Computer Vision

Book · October 2019

CITATION
1

READS
20,068

1 author:

Christoph Rasche
Universitatea Națională de Știință și Tehnologie Politehnica București
86 PUBLICATIONS **685** CITATIONS

SEE PROFILE

Computer Vision

An Overview For Enthusiasts

Bildverarbeitung: eine Übersicht für Begeisterte

C. Rasche

May 11, 2022
[improved introduction to features]

This is a dense introduction to the field of computer vision. It covers all three approaches, the Structural Description approach based on contours and regions; the Local Feature approach; and the Deep Learning approach; summarized in Table 1. The book provides plenty of code snippets and copy-paste examples for Matlab, Python and OpenCV, the latter accessed through Python. We use PyTorch to introduce Deep Neural Networks. Our code notation (variable names, section titles, etc.) is the same throughout all examples and so makes understanding code easy (see paragraph P).

We firstly sketch some of the basic feature extraction methods as those help us to understand the architecture of Deep Neural Networks. Then we introduce those networks, their technique of transfer learning and their use for segmentation using autoencoders. Following that, we sketch object detection and recognition techniques. Then we turn toward the more classical techniques, such as feature extraction and matching based on histograms of gradients - the Local Feature approach. It follows a treatment of image processing techniques - segmentation and morphological processing - and of shape and contour recognition techniques. We overview the essential tracking and motion estimation methods as well as networks for pose estimation. All this culminates into a succinct introduction to robot vision and self-driving cars. We close with an overview of video surveillance, text recognition and remote sensing.

Prerequisites basic programming skills; enthusiasm to write a lot of code (ideally Programmierwut)
Recommended basic statistical pattern recognition, basic linear algebra, basic signal processing

Speed Links		
	Code Examples	P
	Filtering Compact	D
	Resources	M

Contents

Preface

I have written this overview for several reasons. One is, that I tended to forget the function names and their application details in Matlab and Python. Instead of browsing the documentation pages again, I have decided to assemble my own documentation and example scripts to generate a faster look-up table. Another reason is, that there exists little comparison between approaches in the textbook literature, partially due to the rapid evolvement of novel techniques, such as the Deep Network methodology. Making such a comparison helped me understanding the advantages and disadvantages of those techniques. Occasionally I re-read my own sections to refresh some of the issues. Another reason was to collect the important websites into a single document, again for easier look-up. All in all, it serves to me as a Nachschlagewerk (reference book) for the essentials in Computer Vision.

I have placed the overview on ResearchGate to share it with those researchers struggling with similar (Erinnerungs-) challenges. I am not surprised that it has received quite a number of views, but that it would garner thousands of views is a bit unexpected nevertheless. It is proof, that there is need for a more comprehensive book.

I will try to find the time to add some more explanations and example code, because I feel the overview could be a little bit rounder in content and smoother in some sections. For the moment however, sunt foarte ocupat (I'm very busy) with my own implementations. I have surpassed the classification accuracy for some small databases (set by Deep Networks) with my Structural Description approach and now I am hunting the ImageNet benchmark, check out also http://www.sehbau.com/.

Some of my techniques are mentioned in this overview, but a full description will rather appear in a separate, short book. Yes, stay tuned for more novel models and techniques to come!

The overview was assembled mostly in Eastern Europe:
Bucharest, Wallachia (RO)
Berlin, Prussia (DE)

1 Introduction

Computer Vision is the field of interpreting image content. It is concerned with the classification of the entire image, such as in a system classifying photos uploaded to the internet (Facebook, Instagram). Or Computer Vision is concerned with the recognition of objects in an image, such as detecting faces or car license plates (Facebook, GoogleStreetView). Or it is concerned with the detection of aspects of an image, such as cancer detection in biomedical images.

Origin Computer Vision was originally founded as a sub-discipline of the field of Artificial Intelligence in the 1970s. The founding goal was to create a system that has the same perceptual capabilities as the human visual system has - your eyes and most of your brain. The human visual system can easily interpret any scene with little effort: it perfectly discriminates between thousands of categories, and it can find objects in scenes within a time span of several hundred milliseconds only; it easily switches between several types of recognition processes with a *flexibility* and swiftness, whose complexity and dynamics have not been well understood yet. It quickly turned out, that that goal was rather ambitious.

Instead, Computer Vision has focused on a set of specific recognition challenges, to be introduced in Section 2. Those challenges can be often implemented in different ways, with each implementation having advantages and disadvantages. Throughout the decades, many applications have been created (Section 1.2), and some of those implemented tasks begin now to outperform a human observer - such as face identification, letter recognition, or the ability to maneuver through traffic (autonomous vehicles). And that itself is astounding, even though the original goal of an omni-vision system has been elusive. Today, Computer Vision is considered its own field.

Frontier Computer Vision is still considered a frontier, despite its evolvement over almost 50 years. The success of modern Computer Vision is less the result of truly novel algorithms, but rather a result of increasing computer speed and memory. In particular shape recognition is - despite its simple sounding task - still not properly understood. And even though there exist neural networks that can recognize thousands of classes, the system occasionally fails so bluntly, that one may wonder what other algorithms need to be invented in order to achieve a flawless recognition process. If those algorithms will not be invented, then household robots may always make some nerve-wrecking errors, such as mistaking the laundry basket for the trash bin, confusing the microwave with the glass cabinet, etc. Thus, despite all the progress that has been made, it still requires innovative algorithms.

In particular in the past decade, Computer Vision has received new impetus by the use of so-called Deep Learning algorithms, with which one can classify decently large image collections. Google and Facebook are competing to provide the best interface to those learning algorithms: Google offers *Tensorflow*, Facebook provides *PyTorch*, both of which are Python-based libraries. And that is the reason why we treat that topic relatively early (Section 8), after a quick warm up with the classical methods. We then proceed with a method that had been popular just before the arrival of Deep Learning algorithms, namely feature extraction and matching (Sections 12 and 13). Later on, we continue with traditional techniques (Section 14), and we also mention approaches to the most enigmatic challenge of Computer Vision, namely shape recognition (Section 16) and contour description (Section 17).

1.1 Related Fields

Several fields are related to Computer Vision, two of which are closely related, namely *Image Processing* and *Machine Vision*; in fact, those two fields overlap with Computer Vision to such an extent, that their names are sometimes used synonymously. Here is an attempt to discriminate between them, although there exist no agreed definitions and distinctions:

Image Processing is concerned with the transformation or other manipulation of the image with the goal to emphasize certain image aspects, e.g. contrast enhancement, or extraction of low-level features such as edges, blobs, etc; in comparison, Computer Vision is rather concerned with higher-level feature extraction and their interpretation for recognition purposes.

Machine Vision is concerned with applying a range of technologies and methods to provide imaging-based automatic inspection, process control and robot guidance in industrial applications. A machine-vision system has typically 3 characteristics:

1) objects are seen against an uniform background, which represents a 'controlled situation'.
2) objects possess limited structural variability, sometimes only one object needs to be identified.
3) the exact orientation in 3D is of interest.

An example is car license plate detection and reading at toll gates, which is a relatively controlled situation. In comparison, computer vision systems often deal with objects of larger variability and objects that are situated in varying backgrounds. An example would be car license plate detection in GoogleStreetView: the object itself appears with limited variability but in varying context.

There exist two other fields that overlap with Computer Vision:

Pattern Recognition (Machine Learning) is the art of classification (or categorization). To build a good computer vision system, it requires substantial knowledge of classification methodology. Sometimes it is even the more significant part of the computer-vision system, as in case of image classification, for which so-called Deep Neural Networks have produced the best classification accuracy so far (Section 8). Clearly, we cannot treat classification in depth in this course and we will merely point out how to use some of the classifiers (see also Appendices F and H). A key issue will be pointed out in Section 3.3. A straightforward introduction can be found here: https://www.researchgate.net/publication/336718267

Computer Graphics is sometimes considered as part of Computer Vision. The objective in Computer Graphics is to represent objects and scenes as compactly and efficiently as possible; however there is no (semantic) recognition of any kind involved.

1.2 Areas of Application (Examples)

Sze p7

The following list of areas merely gives an overview of where computer vision techniques have been applied so far; the list also contains applications of image processing and machine vision, as those fields are related:

Medical imaging: registering pre-operative and intra-operative imagery; performing long-term studies of people's brain morphology as they age; tumor detection, measurement of size and shape of internal organs; chromosome analysis; blood cell count.

Automotive safety: traffic sign recognition, detecting unexpected obstacles such as pedestrians on the street, under conditions where active vision techniques such as radar or lidar do not work well.

Surveillance: monitoring for intruders, analyzing highway traffic, monitoring pools for drowning victims.

Gesture recognition: identifying hand postures of sign level speech, identifying gestures for human-computer interaction or teleconferencing.

Fingerprint recognition and biometrics: automatic access authentication as well as forensic applications.

Retrieval: as in image search on Google for instance.

Visual authentication: automatically logging family members onto your home computer as they sit down in front of the webcam.

Robotics: recognition and interpretation of objects in a scene, motion control and execution through visual feedback.

Cartography: map making from photographs, synthesis of weather maps.

Radar imaging: target detection and identification, guidance of helicopters and aircraft in landing, guidance of remotely piloted vehicles (RPV), missiles and satellites from visual cues.

Remote sensing: multispectral image analysis, weather prediction, classification and monitoring of urban, agricultural, and marine environments from satellite images.

Machine inspection: defect and fault inspection of parts: rapid parts inspection for quality assurance using stereo vision with specialized illumination to measure tolerances on aircraft wings or auto body parts; or looking for defects in steel castings using X-ray vision; parts identification on assembly lines.

The following are specific tasks which can be often solved with image processing techniques and pattern recognition methods and that is why they are often marginally treated only in computer vision textbooks - if at all:

Optical character recognition (OCR): identifying characters in images of printed or handwritten text, usually with a view to encoding the text in a format more amenable to editing or indexing (e.g. ASCII). Examples: mail sorting (reading handwritten postal codes on letters), automatic number plate recognition (ANPR), label reading, supermarket-product billing, bank-check processing.

2D Code reading reading of 2D codes such as data matrix and QR codes.

A growing list of applications can be found here:
http://homepages.inf.ed.ac.uk/rbf/CVonline/applic.htm
The list also contains some (academic) research projects.

For more application examples one can visit Google's sites that offer various products:
https://cloud.google.com/vision/
https://js.tensorflow.org/: recognition for web-browers

1.3 From Development to Implementation

Usually one develops a novel algorithm in a higher-level language first, such as Matlab, Python, GNU Octave, R, Scilab, etc. Once this testing phase is completed, then one would 'translate' the algorithm into a lower-level language such as C, C++ or Cython for instance, to make the application run in real-time, if that is necessary.

Matlab: (http://www.mathworks.com/) Is extremely convenient for prototyping (research) because its formalism is very compact and because it probably has the largest set of functions and commands. It offers an image processing toolbox that is very rich in functionality, use `doc images` do get the overview. And it offers a computer vision toolbox with a lot of tutorials for complex systems; use `doc vision` for the overview.

Octave, R: (https://www.gnu.org/software/octave/, https://www.r-project.org/) For training purposes one can certainly also use software packages such as R and Octave, in which most functions have the same name as in Matlab.

Python: (https://www.python.org/) Coding in Python is slightly more elaborate than in Matlab and does not offer the flexibility in image display as Matlab. Python's advantage is, that it can be relatively easily interfaced to other programming languages that are suitable for mobile app development for instance, whereas for Matlab this is very difficult. In Python, the initialization process is a bit more explicit and the handling of data-types (integer, float, etc.) is also a bit more elaborate, issues that make the Python code a bit lengthier than in Matlab.

If one switches between any of those high-level languages, then the following summary is useful:
http://mathesaurus.sourceforge.net/matlab-python-xref.pdf.

If you process videos, build robots or deal with millions of images, then you need to implement your time-consuming routines into one of the following lower-level languages.

C: Is the language for obtaining maximum speed, or when building systems for embedded devices where memory is often a limiting factor.

C++: Is the language of choice for larger projects, ie. when multiple persons work with the code. It comes at the downside of a sightly slower execution time than in C and the code becomes a bit more verbose than pure C. A lot of code examples and libraries on the web are written in C++. The most prominent library is called OpenCV, see wiki OpenCV or https://opencv.org/. Many of the routines offered by the OpenCV library can also be easily accessed through Python by importing them.

Cython: (https://www.cython.org/) Is essentially the same code as Python, but offers to specify certain variables and procedures in more detail with a notation similar to C (C++). That additional notation

can speed up the code by several factors, up to the speed of C++. Cython is included in the Anaconda distribution. (Not to be confused with *CPython*, which is the canonical implementation of the Python syntax).

For more pointers to coding we refer to Appendix M.

1.4 Reading

I survey some of the books that helped me to obtain the broader picture (and for which I had time to read or browse them). The list is not to be understood as complete or as an endorsement of specific books.

Concepts

Sonka, M., Hlavac, V., and Boyle, R. (2008). *Image Processing, Analysis, and Machine Vision*. Thomson, Toronto, CA. Introductions to topics are broad yet the method sections are concise. Contains many, precisely formulated algorithms. Exhaustive on texture representation. Oriented a bit towards classical methods, thus, not all newer methods can be found. Written by three authors, but reads like if authored by one person only.

Szeliski, R. (2011). *Computer Vision: Algorithms and Applications*. Springer. Meticulous and visually beautiful exposure of many topics, including on graphics and image processing; Strong at explaining feature-based recognition and alignment, as well as complex image segmentation methods with the essential equations only. Compact yet still understandable appendices explaining matrix manipulations and optimization methods.

Forsyth, D. and Ponce, J. (2010). *Computer Vision - A Modern Approach*. Pearson, 2nd edition. Exhaustive on topics about object, image and texture classification and retrieval, with many practical tips in dealing with classifiers. Equally exhaustive on tracking. Strong at explaining object detection and simpler image segmentation methods. Slightly more praxis oriented than Szeliski. Only book to explain image retrieval and image classification with feature methods.

Davies, E. R. (2012). *Computer and Machine Vision*. Elsevier Academic Press, Oxford. Rather machine vision oriented (than computer vision oriented). Contains extensive summaries explaining advantages and disadvantages of each method. Summarizes the different interest points detectors better than any other book. Treats video surveillance and automotive vision very thoroughly. Only book to contain automotive vision.

Prince, S. (2012). *Computer Vision: Models, Learning, and Inference*. Computer Vision: Models, Learning, and Inference. Cambridge University Press. Also a beautiful exposure of some computer vision topics; very statistically oriented, starting like a pattern recognition book. Contains up-to-date reviews of some topics.

Wikipedia Always good for looking up definitions, formulations and different viewpoints. Even textbooks sometimes point out wikipedia pages. But wikipedia's 'variety' - originating from the contribution of different authors - is also its shortcoming: it is hard to comprehend the topic as a whole from the individual articles (websites). Wikipedia is what it was designed for after all: an encyclopedia. Hence, textbooks remain irreplaceable.

Furthermore, because different authors are at work at wikipedia, it can happen that an intuitive and clear illustration by one author is being replaced by a less intuitive one by another author. I therefore recommend to copy/paste a well illustrated problem into a word editor (e.g. winword) in order to keep it.

Software Implementation - OpenCV

There are many more books providing details on implementations, in particular on using the software provided on OpenCV. There is one by two leading contributors to the OpenCV software:

Kaehler, A. and Bradski, B. (2016). *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media.

I recently found the following early manual for OpenCV (2000), that contains very beautiful illustrations for some of the algorithms:
www.cs.unc.edu/~stc/FAQs/OpenCV/OpenCVReferenceManual.pdf

Image Processing

I recommend the following books as they do not treat topics in too much mathematical detail, and that therefore are sufficient for our purposes:

Gonzalez, R. C. and Woods R.E. (2017). *Digital Image Processing: An introduction oriented toward Matlab*. Pearson. http://www.imageprocessingplace.com/

Marques, O. (2011). *Practical Image and Video Processing Using Matlab*. Wiley, IEEE Press.

Deep Learning

With regard to Deep Learning I have not made an effort yet to read books. There must have appeared many books by now, but since the field is so quickly evolving, they tend to be outdated within months. But there are signs that the dust appears to settle. If one is interested rather in implementation than theory, then perhaps the following book is of good use, the author appears to continuously update his writing:
https://www.pyimagesearch.com/deep-learning-computer-vision-python-book/

Human Vision

If there should be interest in understanding human vision (Sections 2.4, 27), then these references are good starting points:

Active Vision: The Psychology of Looking and Seeing, John M Findlay and Iain D Gilchrist
1) Compact overview of essential eye-movement studies.
2) A convincing summary of arguments against the excessive use of attentional shifts.

Vision Science: Photons to Phenomenology, Stephen E Palmer
1) An opus, settled at the interface of psychology, neuroscience and computer science.
2) Explains the topic as a whole.

Society of Mind, Marvin Minsky
On the representational complexity of the brain

Vision: A Computational Investigation into the Human Representation and Processing of Visual Information, David Marr
An early attempt to provide a 'holistic' picture of various disciplines (psychology, neuroscience, computer science)

More References

Further references can be found here, a list that also contains books on human vision:
http://homepages.inf.ed.ac.uk/rbf/CVonline/books.htm

2 Recognition Processes

There exist different recognition processes and in order to understand the differences between them, we start by dividing them into 'static' versus 'moving'. By static we mean recognition processes that interpret a static scene; the input to the recognition process is a single image (Section 2.1). By moving we mean processes that interpret moving objects or scenes, also called *motion analysis*; the input to the recognition process is a sequence of images (Section 2.2). In real life of humans (or animals) there exist combinations of these processes, something we mention at the end of the motion section (Section 2.2.1). Then we explain the challenges of collecting training material (Section 2.3) and explain shortly the type of comparisons that are made between human and computer performance (Section 2.4).

2.1 Static Scenes

The principal recognition processes are classification, identification and detection. We list those three first, then we mention tasks that can be considered a combination of them.

Classification (Categorization): is the assignment of an object or scene to a category label (class label), such as 'car', 'apple', 'beach scene', etc. The challenge here is to deal with the so-called *intra-class variability*, the structural (configurational) variability among class instances (Section 8). In some classes this variability is small, e.g. bananas look relatively the same; in some classes, the variability is nearly endless - think of how differently chairs can look like, including designer chairs.

Image Classification is the assignment of the entire image to a single label. If the image depicts a single object, then this can also be considered object classification; that should however not be confused with object search, a scenario where the target object can be anywhere in the image (and is thus often much smaller than when depicted for presentation); in that scenario the task becomes a detection or localization problem, introduced further below.

Categories come at different levels of abstraction (Fig. 1). When a human categorizes an object then s/he will usually assign it to a so-called *basic*-level category (Palmer, Vision Sciences), such as the ones mentioned already (car, apple, chair, etc). Categories that are less abstract are also called *subordinate*-level, such as sports car, Granny Smith apple, kitchen chair, etc. Categories that are more abstract are also called *superordinate*-level, such as vehicle, fruit, furniture, etc. Of course there exists no unified categorization of objects as every person has a different motivation and therefore creates (slightly) different categories of its environment. It is probably best to think of it as a continuum from least to most abstract; sometimes the terms *fine* and *coarse* are used to describe the continuum.

Identification: is the recognition of an individual object *instance*, such as in face identification, fingerprint identification, identification of one's own car, etc. In principle, this process can be regarded as a classification process operating at the least abstract (most specific) level of the category continuum.

In Robotics there exists the task of *place recognition*, the ability to re-identify a previously seen scene; an ability important for successful navigation. Place recognition is an identification process.

Detection/Localization: is the search for a specific object class in a scene; or it is the detection of a specific condition in an image. The challenge is to create an efficient search process that can find the object irrespective of its size: does the object cover the entire image?; or is it small and therefore difficult to detect? Examples: face detection, vehicle detection in an automatic road toll system, detection of possible abnormal cells or tissues in medical images. One uses the term detection, if one is interested in finding a single class in a scene (Section 10). The term localization is rather used when one attempts to find multiple object classes (Section 11).

Humans carry out search by an extremely dynamic process involving saccadic eye movements - shifts of focus across the image. But it also uses 'internal' shifts of focus; humans are able to look straight - and seemingly focus - and still be able to process scene parts outside focus. The process is so dynamic, it is hard to comprehend its detailed functioning.

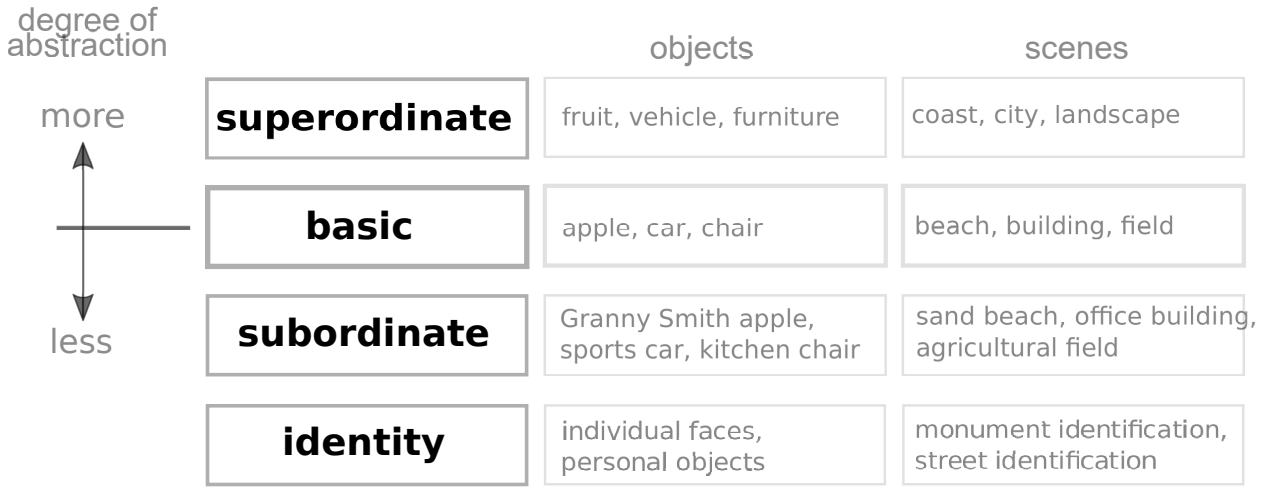

Figure 1: Levels of abstraction for categories. The **basic** level is assumed to be the commonly known degree of abstraction (for humans); Irving Biederman estimates that ca. 5000 basic-level categories exist, taking the number of nouns in a dictionary as reference. The **superordinate** level is more abstract (less specific; larger intra-class variability), the **subordinate** level is less abstract (more specific; smaller intra-class variability). The **identity** level represents unique objects, scenes, shapes, etc.
Computer vision systems are particularly good at the identity level for some tasks - sometimes exceeding human recognition performance such as in face identification or hand-written digit identification. Humans are unbeatable at their flexibility: they categorize an image into one of thousands of categories within a glance (ca. 150 milliseconds, see research by Simon Thorpe).
When training a classification system, the challenge is to collect images and to sort them properly - one will quickly encounter ambiguous cases; cases that can not be easily assigned to a single class.

In the literature, the term *object recognition* often implies some combination of those three processes - occasionally it stands for only one of them.

Another process that might be considered a (static) recognition process is **Supervised Segmentation**, the partitioning of a scene into its constituent objects (Section 9). This process can be understood as a combination of the two processes localization and categorization, whereby the localization process aims at high accuracy: its goal is often to delineate the object silhouettes.

Occasionally, the topic of **Image Retrieval** is considered its own task. Image retrieval is the search for a similar image in a (large) image collection using a given image as cue. For instance, some search engines provide a function, that allows to upload an image and in response the most similar images are retrieved. Depending on the exact search goals, this retrieval can range between identification and abstract categorization. For example, if we search brand logos to detect potential copyright breach, then that would be identification; if we search for fruit images on Google, then that would be basic-level categorization essentially (Fig. 1). The search process is often aided by methods from the field of Information Retrieval (Section 18).

2.2 Motion Analysis

Motion analysis is a general term for reconstructing where to and how something has moved between two moments in time. There are two principal types of movement. In one, the observer stands still and registers when objects in the scene move or have moved. In the other, the scene remains still but the observer moves, in which case the entire scene appears as moving. It becomes really challenging when both types of movements occur, i.e. during driving. We illustrate the first type in Fig. 2, a square shape moving from ‘a’ to ‘f’ thereby undergoing a rotation.

Tracking: is the pursuit of one or several (moving) objects in an image - it can be regarded as frame-by-

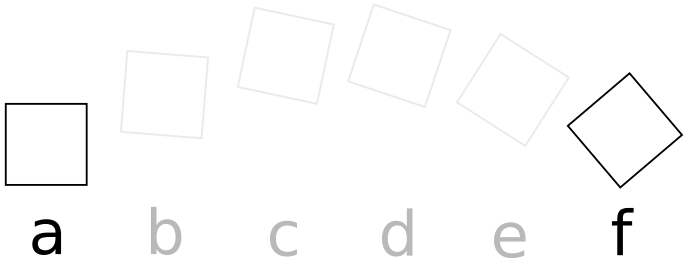

Figure 2: Motion analysis. Motion tracking is the pursuit of an object, if possible frame-by-frame, exemplified here as a square moving from 'a' through 'f' (that is a smooth pursuit 'a'-'b'-'c'-...-'f'). Motion estimation concerns the change in pose between two moments in time, e.g. between 'a' and 'f' ('b' through 'e' not available). Motion classification is the categorization of the trajectory, i.e. as the trajectory is curved it could be classified typical for 'the object was thrown'.

frame object detection (Section 19). In our illustration that would mean pursuing the object from 'a' through 'f'. Examples: tracking a user's face for human-computer interaction; tracking pedestrians in surveillance; etc. Tracking answers *where* the object is moving. Humans can track objects too - the process is called *smooth pursuit* in the neurophysiological literature.

Motion Estimation: is the reconstruction of the precise movement between successive frames or between frames that are separated by a short duration. It answers *how* the object or scene has moved; in our illustration that would mean how the object has changed its pose from 'a' to 'f', assuming that moments 'b' to 'e' are not available (Section 21).

Viewpoint Estimation is the challenge to determine from which point of view a specific object is seen; the task can therefore be regarded as a complex form of motion estimation. This is better illustrated with another graph: figure 3 shows a cube from slightly different viewpoints and should illustrate two scenarios: either the cube has rotated and the observer has remained in place, or the observer moves around the cube (Section 22.2). In either scenario, the challenge to determine the exact viewpoint is essentially the same and can be regarded as a problem of motion estimation.

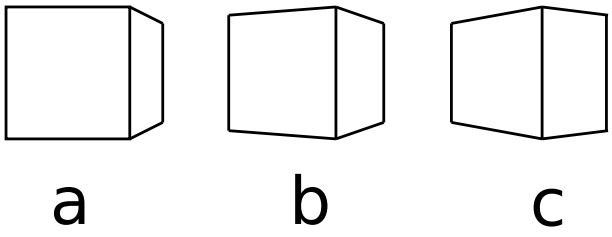

Figure 3: Viewpoint estimation. A cube seen from three slightly different viewpoints: the task is to determine from which angle the object is seen; the task is often regarded as the estimation of a motion, irrespective of whether the object or the observer moves.

Motion Classification: is the challenge of recognizing entire, completed motions as in gesture recognition or human body movements (Section 23). It answers *what* type of movement the object has carried out. When speaking of recognizing human motions, then this is also referred to as *action recognition* sometimes. In our illustration, this would mean to follow the trajectory from 'a' through 'f' and classify this information together with the change in pose, i.e. the object appears to have been kicked and it rotates during the motion.

Related to motion analysis there exists also the challenge to determine whether something has changed at

all. This is called *Change Detection* and is often the first step when carrying out motion analysis (Appendix L).

2.2.1 Combination of Recognition Processes

Some tasks can be considered a combination of static and dynamic recognition processes. For example, identifying a flying airplane means to track and categorize it simultaneously. Identifying a bird that is taking a lot of turns in the air, can be regarded as a task for which both, classification of motion as well as classification of 'static' properties, are necessary; i.e. a bat has a rather jerky flight trajectory and an unusual wing silhouette.

2.3 Training Images

To instruct a system with a specific classification task, the first challenge is to collect appropriate learning material, that will be used for training the system. This collecting phase is laborious work and it is sometimes more challenging than one expects - it is often by inspection of individual images, when the classification intricacies become evident.

For an image classification task, that collection phase is a relatively straightforward process - in comparison to object detection tasks. For example if we intend to train a system to distinguish between outdoor and indoor scenes, then we collect images for both classes and assign them to two folders; software packages offer functions to read those folders, explained below. One should thereby pay attention to several issues:

Class Balance: the image count for the classes should be balanced: each class should have a roughly equal number of images, i.e. 180 outdoor scenes and 220 indoor scenes. If it is not, then we deal with a so-called *class imbalance* problem. We need to pay attention to potential biases during applications and counter weigh if necessary. This is in particular an issue in medical images, where one typically has many benign cases from healthy patients, but only few tens of malign cases from affected patients.

The easiest way to deal with that is to perform oversampling and undersampling. We increase the class with fewer cases by generating artificially new images using the technique of augmentation (Appendix J), called oversampling. And we decrease the class with abundant cases by eliminating some of its images, called undersampling. This is a relatively straightforward procedure. For detailed techniques one can start by exploring the wiki pages, wiki Oversampling_and_undersampling_in_data_analysis.

Image Borders: some images contain stripes of specific color at their image borders; a black border of width equal one pixel can easily go unnoticed to the human observer. If the image borders of a particular class show a certain color, it is possible that the system may learn to discriminate that class solely based on those border colors. In particular Deep Nets are known to exploit any unusual attributes, which speaks for the effectiveness of the method, but must be kept in mind in applications.

Scene Background: this is analogous to the issue of image borders. If images of an object class always contain the same scene background, then the system may learn rather the background, than the object itself. For example, depictions of a mountain animal in front of only a snowy background, may lead to learning the white background, if there are no other classes with such background.

If we intend to train an object recognition system that can detect or localize images anywhere in the image, then we have two choices: we either collect images where the object is placed approximately in the image center; or we collect images where the object is in arbitrary positions - i.e. upper left corner - and then mark their location. How this is done exactly will be detailed in later sections, but here we merely mention that this is a laborious process and requires again careful selection of those images.

For a few classes, images can be collected individually without much effort. A few tens of images can give a decent classification performance and may suffice for consumer applications, where an occasional error does not have consequences. However if highest accuracy is necessary, the image count can easily reach five thousand or more, such as for pedestrian detection for automotive vision, or medical images for diagnosis. In order to maximally exploit the available image material, one often enlarges the material by

introducing slight manipulations of the images, a process called *data augmentation*; the procedure for that will be detailed in later Sections, see also Appendix J.

For the discrimination of tens or hundreds of classes, or for high precision tasks, the collection of image material is often done at universities, tech companies or other institutions; sometimes they employ volunteers or they pay workers for labeling, e.g. wiki Mechanical_Turk.

Existing Collections For many tasks there exists a good number of collections, an overview of those can be found here:

`http://homepages.inf.ed.ac.uk/rbf/CVonline/Imagedbase.htm`
`https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research`

Many of those were created considering the above issues, but for older data sets it is better to carefully study the image material to verify that it meets the task requirements. It is not unusual that one takes an existing data set and then drops those classes that do not fit for the problem at hand.

In PyTorch the function `ImageFolder` in module `torchvision.datasets` reads the list of images automatically. Thus we hand over our folders (i.e. folder with indoor images, folder with outdoor images) and then can proceed to learning immediately. More details will be revealed in later sections (Section 8).

In Matlab there exists the function `imageSet` to read a list of image. It also allows partitioning the image material for cross-foldings. For handling file paths there exist functions such as `dir`, `fileparts`, `filesep`, etc.

Notes A more thorough discussion of issues that one should be aware of, is found here:

`https://link.springer.com/article/10.1007/s11263-017-1020-z`
`https://vitro-testing.com/cv-hazop/`

Some companies have started to offer a near-automated image collection process and also provide learned Deep Net weights, i.e. `https://github.com/idealo/imageatm`.

2.4 Human-Computer Comparison

Computer vision systems have become so high-performing, that comparisons to human performance have been made. It is in particular for identification tasks, for which computers have started to outperform human identification accuracy, such as face identification or hand-written digit identification. It must be added however, that if a human was trained solely on those tasks - without any other earthly distractions - , then a human may well be able to surpass those systems. What the human visual system (HVS) excels at are four aspects:

Massive Storage: the HVS is able to memorize a sheer amount of seen image material - think about your life-time's experiences.

Effortless Categorization: the HVS is able to use that memory in a way to classify images within 150 milliseconds; think of how effortlessly you watch rapid-paced television commercials or pop-culture music videos.

Efficient Search: when we search for a venue while driving, we are able to browse vast image material in extremely short time.

Energy Efficiency (at Application Time): after the HVS has acquired its representations throughout years of learning, it is extremely efficient at applying its system: guiding through traffic is carried out at computational costs that are orders of magnitude smaller than that of the computational hardware built into autonomous vehicles.

In short, the HVS shows an unbeatable efficiency and *flexibility* that has not been achieved yet by any computer vision system, not even in its approach.

A severe downside of the HVS is that it can suffer from fatigue and distraction and those are the reason for many traffic accidents. This is not an issue for the vision system of an autonomous vehicle: it may lack the efficiency and flexibility, but does not suffer from those physiological and psychological constraints.

3 Engineering Approaches

Any approach to recognition starts by identifying pixels that belong together making up a region segment, a contour segment, a texture element, etc. Those groups of pixels are called *features*, sometimes also primitives or elements. Often they are considered *local* meaning that the occupy a small part of the image. Those features are then integrated toward more complex features, constituting scene parts, object parts, shape parts, etc. Those integrated features are sometimes considered *global*, because they occupy a larger region, perhaps the entire image. Finally, that integration process arrives at a point of decision, that assigns a label to the image content, the classification decision. This recognition evolvement is often displayed as a hierarchy, see Fig. 4. The image enters from the bottom, from which then the features are extracted and integrated. Information flow upward is called *bottom-up*; information flow downward is called *top-down*. This flow is sometimes depicted along a horizontal axis, from left to right, in particular for large neural network architectures (not depicted here); in that horizontal layout one uses rather the terms *forward* and *backward* than upward and downward, respectively. That rotated layout does not change the use of the terms local and global.

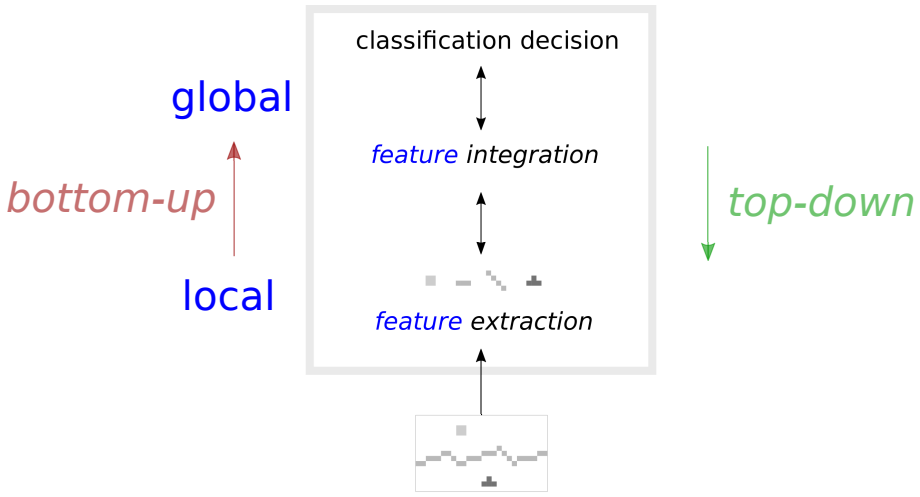

Figure 4: Terminology of process flow for recognition evolvement, typically arranged as a hierarchy. The image pixels enter from the bottom. In a first stage, features are extracted, symbolically indicated by small groups of pixels (elements of the input image). At this stage, features tend to be called *local*. In a second stage, those local features are integrated toward more complex groups, now tendentially called *global* features. Finally, a classification decision is made.
As this flow occurs 'upward' (in a vertical depiction as shown here), it is also referred to as a *bottom-up* flow. For some tasks there occurs also a 'downward' flow, in which case one also refers to *top-down*. Sometimes, the flow is depicted horizontally from left-to-right, in which case one use rather the terms forward and backward (than upward and downward, resp.).

In the following we firstly survey the types of feature that exist (Section 3.1). Of those types some have been more successful than others (for the purpose of classification) and we sketch the three principal approaches to feature extraction and integration (Section 3.2). All of those approaches use a classification step that is rather similar - despite the differences in feature integration -, explained in Section 3.3. As the Deep Neural Network approach has become so popular, it tries to distance itself from the other approaches by arguments as presented in Section 3.4. Lastly, we will give shy recommendations on what type of approach might be suitable for an application (Section 3.6).

3.1 Feature Types

Feature types are very diverse and can be classified differently. We do this here according to their degree of parameterization (Fig. 5). On the one end of this spectrum lies the template (left in Figure) that shows no actual parameterization as such; on the other end lies a full parameterization of the feature (right in Figure). In between are feature types based on histograms, codes, mathematical functions, etc. Most of these features can be taken at a local level, at the bottom of the hierarchy, or they can also be taken at more global level, even for the entire image. We start the feature journey with templates, working our way toward the other end of the spectrum.

Template In this type, the feature is expressed as a two-dimensional array of values holding only binary (logic) values, or floating point (scalar) values. This feature representation is memory intensive as an entire array is used. For example for the 8x8 pixel array shown in the figure, we have 64 values to remember. The advantage of this template is that it is easy to compare to other templates, by mere pixel-by-pixel comparison, i.e. sum of distances (sum of 64 pixel (squared) differences). That comparison is not accurate in a geometric sense and therefore somewhat loose, but this looseness naturally tolerates variability in input, a tolerance that is useful for buffering intra-class variability (structural variability) to some extent. Templates are used in those infamous Deep Neural Networks (Section 8 and 9), where they - genau genommen - are 'weight' templates.

In some tasks, the entire image is taken as a template, then occasionally also called *holistic* representation, as the image as a whole is used for computation without any further preprocessing. This seems simple, but has proven to be quite effective, in particular when comparing thousands or millions of images, as in image retrieval or place recognition (Sections 5, 18 and 25.2).

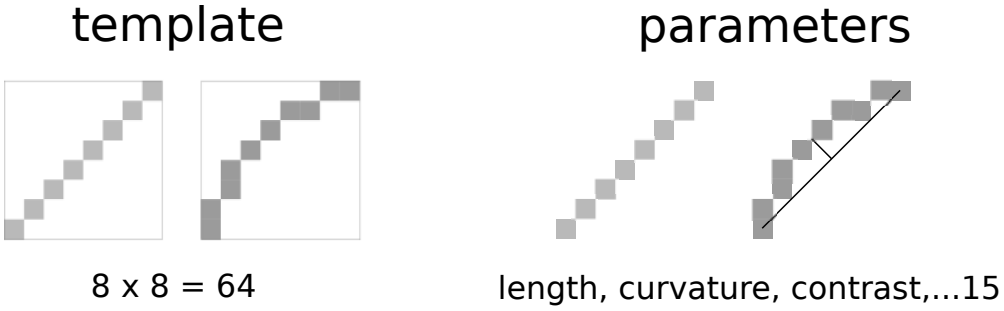

Figure 5: Two extreme types of features - the two ends of a spectrum representing the degree of parameterization (left half 'template': no parameterization at all; right half 'parameters': maximal parameterization).
Template (left): the feature is expressed as a small 2D array, a small image patch or neighborhood illustrated here as two 8 x 8 pixel patches, one containing a straight segment, the other a curved segment. Templates are easy to compare by (pixel-wise) distance measurements, but are memory intensive. The template is at the basis for Deep Neural Networks (i.e. a Convolutional Neural Network); in image retrieval of millions of images, one sometimes treats the entire images as template and uses distance measurements as will be explained in Section 5.1.1.
Parameters (right): the feature is described by a list of parameter (values), such as its arc length, curvature, orientation, contast, etc. (up to 15 parameters for example); those measurements are determined using algorithms. Features are then expressed as a vector with components made of those parameter values, also called *feature descriptor* sometimes. Parametric representation is precise and compact, but bears the challenge to find the appropriate distance measurements in that multi-dimensional space.

Histograms (Local Feature, Global Descriptor) Histograms represent a first step toward abstraction. Histograms are formed by observing pixel values, be that raw intensity values or some computed value at each pixel. Or a histogram is formed for some image attributes. Histograms can be taken for a small image

patch, then often called a *local feature*. They can also be take for the entire image, then often called a *global descriptor*.

A particularly succesful feature is one that takes a histogram of gradients for an image patch, generally referred to as *the* "Local Feature". The choice of where in the image such histograms are formed is based on some measure of interestingness. This feature will be introduced in Sections 12 and 13.

Code (Local Binary Patterns) In that approach, a code is generated. The most succesful example is the so-called *Local Binary Pattern*. It observes the neighboring pixels in reference to some chosen pixel and generates a binary code or some other code. It will be mentioned in Section 7.3.3.

Spectral Filtering Here the image is analyzed with mathematical functions, ranging from simple step-like functions to complex functions such as sinusoidal functions, exponential functions, etc. Examples are Gabor functions (Fig. 20), wavelets, haarlets (Haar features, Fig. 25), etc. In most of these cases, it corresponds to a convolution, the 'folding' of the image with a so-called filter or kernel (Appendix B). This is often done very systematically for a range of filter sizes to cover different object sizes or zoom, hence the term 'spectrum'.

Those filter functions are useful in certain tasks for specific object classes, i.e. rapid face detection. However, for general categorization, this type of filtering has been outperformed by the other approaches and has therefore lost its appeal. We mention this approach in Section 7.3.2.

Parameterization (Full, Structural) Here the feature is characterized with statistical or structural methods. The most accurate characterization would be to describe the structure of the feature using geometric parameters, such as its arc length, its degree of curvature, its angular orientation in the 2D plane, its contrast, etc. (i.e. Section 17). Those parameters are then concatenated to a vector, called *feature descriptor* sometimes (or simply descriptor), with which one can operate using methods from the field of Pattern Recognition. The challenge with this feature is the process of comparison, its multi-dimensional space in particular: the more parameters we use - and therefore the more accurate we describe the feature -, the more difficult it is to find the appropriate degree of abstraction that buffers intra-class variability. This challenge is akin to the famous problem of 'Curse of Dimensionality' in the field of Pattern Recognition. The advantage of this approach is its compactness; even if we described a contour segment with as many as 15 parameters, then that is by far more memory efficient than the template.

Parameterization is not confined to a particular size, i.e. the parameterization of a contour occurs irrespective of its length. In the figure, the segment is shown of same size as the template for reason of illustration, the array outline is left away to indicate the independence of any specified size.

Summary Local/Global A local feature (or descriptor) can be anything that expresses the statistics of an image patch, or a code as in case of Local Binary Patterns. In the literature the term Local Feature (now capitalized) typically refers to the above mentioned feature type that is formed with histograms of pixel values (intensity, gradient, etc.). That expression also refers to its class representation, introduced next. A global feature (or descriptor) can be a histogram of values taken from pixels, or the output of spectral filtering. It then often stands for a description of the entire image. Confused? So am I occasionally. A more precise meaning of the term local/global can only be derived from context, i.e. textual explanations or citations pointing out the specific approach discussed.

3.2 Feature Integration - Class Representation

Feature integration is the process of assembling local features toward more global features that represent a larger part or portion of the image. Sometimes those global features are also called complex features or groups of features. The integration process constitutes also the class representation per se, and it thus represents the most challenging part of the recognition process. Integration is introduced here as a separate topic for reason of simplicity, but is in fact dependent on the types of features extracted; integration and representation is an intertwined issue.

Integration can be carried out minimally, i.e. features can be taken as a mere set (list) of features, without further assembly. Integration can be carried out systematically and thoroughly, by assigning features to image sections, that in turn are combined hierarchically. And there exists intermediate forms between those two extremes. But instead of listing those separately, we directly sketch the three principal, successful recognition approaches.

3.2.1 Deep Nets (Deep Learning)

Deep Neural Networks, or simply Deep Nets, integrate features gradually across several layers (Section 8 and 9). In computer vision, most of those networks are so-called *Convolutional Neural Networks (CNN)*. Those networks learn what type of local templates are suitable to represent their input. The smallest templates start with sizes typically around 5 x 5 pixels. The image is analyzed with multiple layers of such templates and they cover the entire image; this is akin to a convolution with a filter - the Filtering approach as mentioned above. The crucial difference is that those filters are learned and that they can be very individual, depending on the input presented during the learning phase. One could also call it individual filtering.

Those local templates are then integrated into more global templates that become increasingly larger in size until they cover large portions of the image. This hierarchical integration occurs partly by subsampling the image at different levels of the hierarchy, thus forming a pyramid (left graph in Fig. 6). The representation in Deep Nets can therefore be regarded as a pyramid of local, individual templates.

The great advantage of this representation is that it is capable of discriminating almost any set of classes decently, basic-level and super-ordinate in particular (see again Fig. 1). It has near omni-potent classification capacity. The downside of this representation that it requires a lot of memory to remember those templates, which in case of neural networks are called weights. Along with the huge memory comes the need for a lot of computational power to learn and update the network.

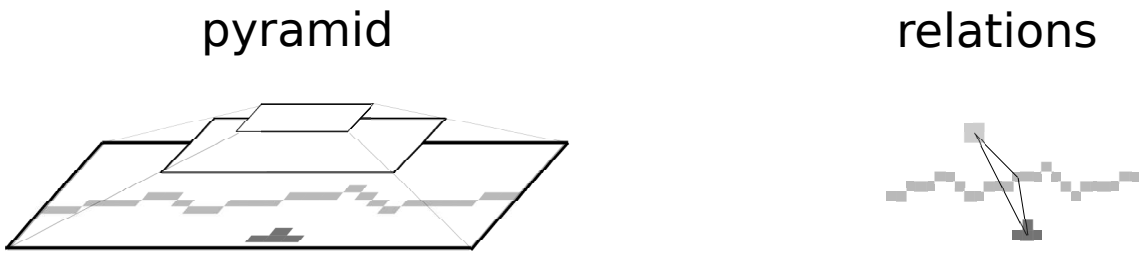

Figure 6: Principles of feature integration.
Pyramid: features are gradually integrated in a hierarchy that becomes increasingly smaller toward the top (3 levels are shown). Used in Deep Nets, i.e. Convolutional Neural Networks, where pyramids consists of up to 10 layers.
Relations: features are set in relation to each other using quantitative or qualitative measurements, i.e. vectors or relational aspects (above, below, left, right, etc.), respectively. The graph illustrates the full graph between three features by delineating the straight line between their midpoints.

3.2.2 Local Feature (Rectangular Patches)

This approach uses as feature the so-called *Local Feature* as it was mentioned above under 'Feature Types'. It will be introduced in Sections 12 and 13), see Fig. 31, for the idea. The feature type is also the name for the representation. From an image, one extracts several hundreds or thousands of small (local) patches. Those are then integrated in three principal ways:

List: no actual integration takes place and the list of local features is used as such. This then leads to the term *bag-of-words*, something that will come up in Section 13.

Relations: the features are related to each other with explicit distance measurements, angles, vectors, etc (right graph in Fig. 6). One set of techniques would be to use Graph methods wiki Graph_theory.

Pyramidal Statistics: the features are assigned to cells in a map, whose lay-out is pyramidal (left in Fig. 6). This can be loosely regarded as an integration, albeit a rather moderate one compared to Deep Net architectures.

Of those three types of integration, it is only the first and third one (list and pyramidal statistics, resp.), that showed some success; the second one (relations) has been rather brittle as object or scene classes show such a large degree of intra-class variability.

The Local Feature approach was primarily developed in the late 1990's until ca. 2010. It has been quite successful in a large number of tasks, but has been surpassed in accuracy by Deep Nets in many but not all tasks. It has lost its appeal for the task of basic-level categorization, but maintains its position in particular for the task of place recognition (Section 25.2).

3.2.3 Structural Description (Contours and Regions)

The Structural Description approach focuses on segments, for example contour or region segments, scene parts, object parts, etc. Those segments are parameterized and then set in relation to each other (as depicted in the right graph of Fig. 6).

It is natural to think of shapes, object and scenes as a structure consisting of contour and region segments; after all, a human can sketch an object or scene with a few lines only. It is not difficult to identify those features, and methods for extracting them will be introduced in Sections 7, 14, 15, 17 and 16. But it has been less obvious how these features should be efficiently represented for classification, let alone to relate them to describe the structure as a whole. The approach has been successful mostly in machine vision applications, where the scenarios are restricted, the structural variability is limited and the illumination is controlled. Structural description was pursued primarily before the year 2000. My recent research shows however, that this approach is in principle competitive with Deep Nets and that it can serve equally well for near omni-potent classification.

Contour and region features are sometimes used as object proposals for the other two approaches (Section 10.5), for Deep Nets and for the Local Feature approach.

3.3 Classification Essence

After the features were integrated, it requires a classification step that assigns the features to a category label. This classification step is essentially the same for most (visual) recognition systems, no matter the exact feature integration approach.

The processes of feature extraction and feature integration transform the image into a single vector of n components, also called *image vector* sometimes (Fig. 7). If only part of an image is analyzed - which then probably is supposed to correspond to an object - , it may also be called an object vector. That image (or object) vector can be a histogram of features, in which case the components represent a feature count (as in bag-of-words with Local Features); the vector can also hold floating values representing the degree of presence of a (global) feature, or the degree of combination of several features (relations); or it can represent any combinations of those two formats. The length (dimensionality) of the image vector can range from several components to tens of thousands of components. The image vector is then handed over to an algorithm from the field of pattern recognition, that carries out the final phase of classification, namely the selection of the most appropriate category label; often a so-called *confidence score* is calculated for each category, and the maximum value indicates the most appropriate label.

The use of such a fixed-sized image vector may appear counter-intuitive, because scenes (or objects or shapes) can have different amounts of contours and regions, different degree of agglomeration or density. For instance a coastal sunset may have only one horizontal straight line with a circle, a busy city scene may have many vertical straight lines; to compare those two scenes, it may appear odd to use a fixed-size representation. Practically, it has been difficult to build a more efficient classifier than those traditional pattern recognition methods (Appendix H) and for that reason one often generates such a fixed-size vector (Section 13).

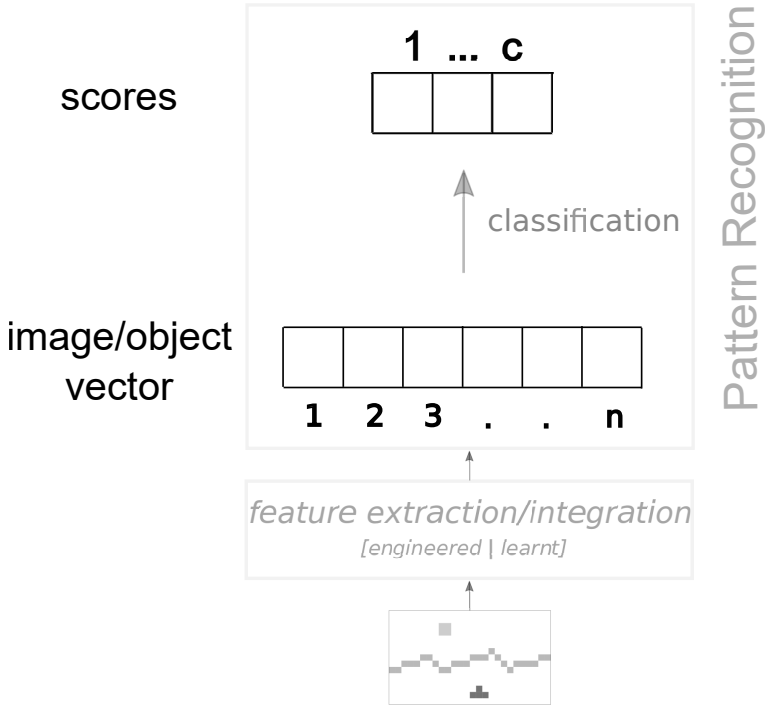

Figure 7: The feature extraction and integration process results in an image vector with n dimensions (components), that represents the image content, also called *image vector* sometimes. The image vector is then handed over to a traditional classifier as they were developed in the field of Pattern Recognition; the algorithm outputs a *score* value for each class, which expresses the degree of class confidence (c is the number of classes). The reason for the choice of those algorithms is that they are so efficient, that it has been difficult to build a more efficient classifier for visual recognition. n can be up to several thousand dimensions in size; ideally n would be as small as possible, as this saves computation time during classification (and still be able to discriminate the classes as good as possible).
Note that in the field of Pattern Recognition, a feature is sometimes considered to be a dimension or component of a vector, a *column* of a data matrix arranged in [samples x attributes]; whereas in Computer Vision a feature is a description of a piece of image - often a vector itself and therefore a *row* of that data matrix.

Not only images are expressed with such vectors, but also objects or shapes. For object recognition, one therefore extracts the vector at different locations in the image, meaning the feature extraction process is applied to each subset of the image; to each subset, the object classifier is applied separately.

Even Deep Neural Networks apply this principle: the network architectures hold in their last, hidden layer that image vector. What sets Deep Nets apart from the other (engineered) approaches is, that this classification decision is an integral and final part of an architecture, that adjusts the parameters for feature extraction and feature classification simultaneously. This distinction will be further elaborated in the next section.

3.4 Feature Learning Versus Engineering

The overwhelming success of Deep Nets has created such a co-motion within the field of Computer Vision, that it has led to a philosophical dichotomy of *Feature Learning* versus *Feature Engineering* (Fig. 8). With the term Feature Learning is meant that representations are learned completely automatically as done in a Deep Net. Proponents of the Deep Net approach consider any other approach as Feature Engineering, as the manual or hand-crafted design of features and representations, such as pursued in the Structural Description approach, the Local Feature approach, the Spectral Filter approach, etc. We elaborate on the Feature Engineering approach first, then on the Feature Learning approach.

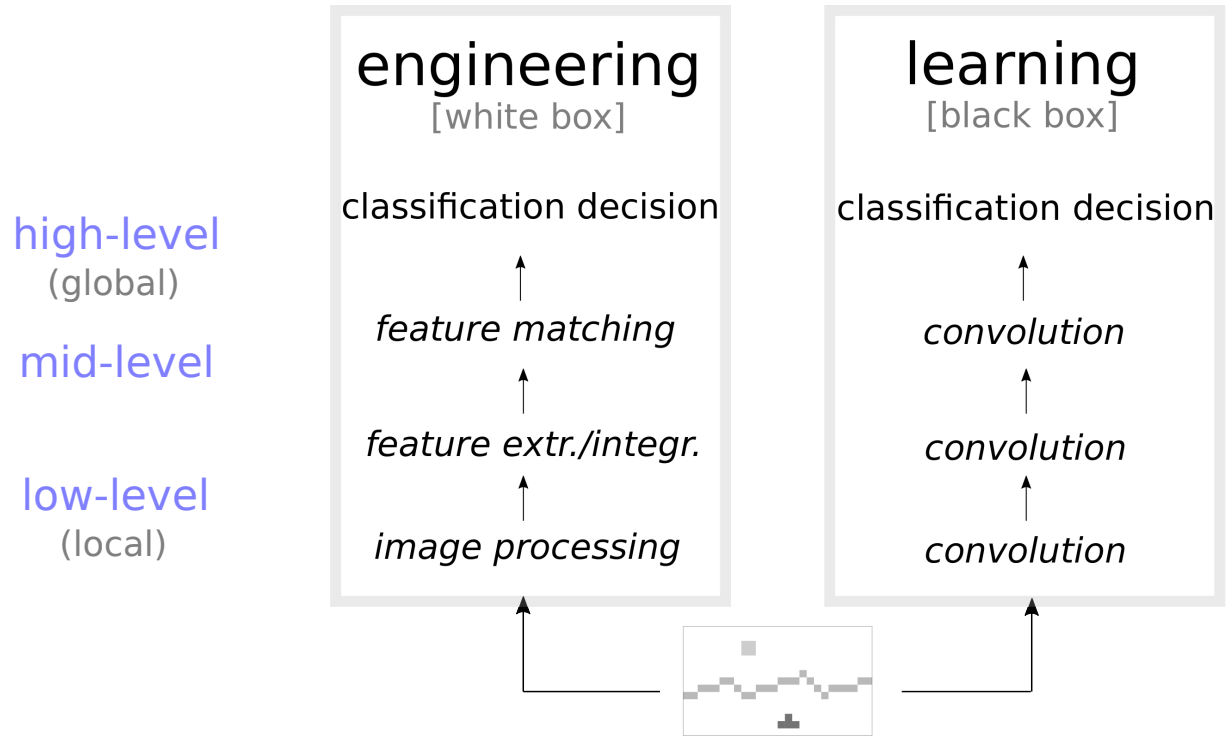

Figure 8: Two philosophies of solving recognition tasks: *Feature Engineering* (left side) versus *Feature Learning* (right side). For simplicity we drew only bottom-up arrows - flow can also proceed top-down in either approach.
Feature Engineering: is the more traditional approach represented by Filtering, Local Features and Structural Description. Feature engineering is considered the harder approach, as it requires more design effort and time, but it is computationally more efficient - it is eco-friendly. Sometimes the terms *low-level*, *mid-level* and *high-level* vision are associated with the hierarchy, expressing the local/global scale as introduced before.
Feature Learning (Deep Nets): is the modern approach (Section 8) and learns the features and representations completely automatically. Feature learning is more convenient in application, but also uses a lot of computational power - it is the equivalent of a SUV (sport utility vehicle). Occasionally, one encounters the terms *local* and *global* that specify in what range of the hierarchy certain computations take place.

Feature Engineering It is worth starting this with a short historical note. In the early years of computer vision, the paradigm for recognition was formulated as a process, that - starting with a 2D image - gradually and meticulously reconstructs the spatial 3D layout of the scene. This 3D reconstruction process was often divided into low-level, mid-level and high-level vision process, corresponding roughly to the three phases depicted in the left pathway of Fig. 8. It was inspired by the fact that we humans perceive the world as a 3D space. Over the years, it has become clear that this paradigm is too elaborate and too complicated, in particular if one intends to infer only a category label. One then shifted to brute-force approaches using feature matching, such as the Local Feature approach and the Structural Description approach with modern techniques.

Feature engineering is explicit in its formulation and design of the individual phases. It typically starts with an *image processing* phase (as mentioned under 'Image Processing' in Section 1.1), whose aim is to prepare the image for feature extraction. In a second phase, the feature extraction and integration processes, the pre-processed image is searched with designed feature detectors that identify straight lines, corners, regions, etc (Local Features, parameterized segments, etc.); those are then integrated to more complex features. Those features are then matched in a third phase, the *feature matching* phase, to some sort of representation in order to arrive at a classification decision. As opposed to previous illustrations, we now have inserted a feature-matching phase, that addresses the comparison between features.

The advantage of this approach is that it is lean: it uses relatively few parameters, consumes little power

and in case of failure, it is possible to pinpoint to the responsible processing step. For the latter reason, this approach is sometimes labeled a white box system: it shows transparency. The downside of this approach is that the design process takes time.

Older text books start with an introduction of classical features and segmentation (parameterization). More recent text books have moved those issues a bit into the background. For instance, Forsyth and Ponce's book follows the structure of the classical paradigm (low/mid/high-level vision), but the treatment of edge detection and image segmentation is rather marginal. Szeliski's book organization is centered around the Local Feature approach, but still contains substantial material on image segmentation for instance.

Feature Learning is the approach that performs feature extraction, integration and matching all together in a gradual flow through the pyramidal network, nowadays simply referred to as *Deep Learning*. This is carried out by an elaborate series of *convolution* phases. The learning procedure finds the innumerous templates, here the neural weights, in an automatic manner.

The advantage of this approach is that it is convenient: the automatic acquisition of new representations is certainly preferred over long design time, but it suffers also from a number of downsides. One is that it has a large memory foot print due to the use of templates (Fig. 5), it uses at least 5 MB of parameters for small (and less accurate) networks; for larger networks it grows to tens of MB, sometimes a few hundred MB. Consequently, a Deep Net carries out a lot of computation and hence uses excessive power, which makes it less suitable for embedded systems, think of a robot operating on Mars. In addition, as the processing phases are less explicit and are distributed across many network layers, it is difficult to pinpoint an exact location of weakness in case of failure.

The arguments for both approaches are exaggerated formulations - created by the proponents of the respective philosophies. In the engineering approach, not everything is as fully transparent as the label 'white box' would suggest and the learning approach is not entirely black; more appropriate terminology would 'brighter' and 'darker' box, or 'more transparent' and 'less transparent' box. Analogously, in the learning approach, not everything is fully automatically learned - there is substantial tuning involved sometimes. For example to make a network independent of the orientation of its input, e.g. satellite images contain their image content at any orientation, the networks are equipped with additional layers learning independence of orientation, whereas in the Structural Description approach one would omit any parameters describing angles; that is, in this instance, there would be more engineering in the Deep Net approach. Furthermore, the search for the appropriate architecture is quite a heuristic endeavor, as evidenced by the success of the recent, so-called EfficientNet; some of the network architectures are partially inspired by engineered architectures. Some of the proposed differences can be considered nuances. Because both approaches have their advantages, research has started to create hybrid approaches.

3.5 Concise Comparison of Approaches

We now provide a compact characterization of the three principal approaches introduced so far. It is somewhat early to make such a comparison as we have not introduced all performance aspects yet, but it fits into this section. Some more aspects will be introduced now, followed by the comparison, which is also summarized in Table 1.

Performance Aspects Some of the performance aspects are easy to comprehend in principle, such as the number of parameters necessary to get a system running, or the robustness of the system to (visual) clutter. Some aspects however are a bit more intricate and we introduce three of them, complexity, scalability, and interactability. For that we will refer to the template feature and the parameterized feature (Fig. 5).

Complexity is a measure of complicatedness or intricacy and is an aspect that rather addresses an individual algorithm or technique. The complexity of algorithms often determines also the complexity of the entire recognition process. There are different types of complexities, also elaborated in Appendix I.2. For example creating a feature template is simple (left in Fig. 5), however determining the parameters of a

feature is complex (right in Fig. 5), because it requires an algorithm calculating them. Vice versa, a feature template requires more memory and that can be regarded as complex; a parameterized feature on the other hand uses less memory and is therefore less complex in terms of memory requirements.

Scalability is an aspect that expresses how well a system can deal with an increase in input, when one tries to compensate for that increase by using more resources. When a system is complex it is less likely to be scalable, but not necessarily. For example if we intend to classify thousands of images in reasonable time, then we can try adding more computers to complete the task in time; it that works, then the system would be considered scalable, no matter how complex the actual recognition process is. Like in the case of the complexity aspect, there exist different types of scalability (wiki Scalability); in this context it often means whether a recognition process can deal with large amounts of image material; or if it can deal with very large images, e.g. is able to process a satellite or medical image that are multiple times larger than a regular image.

Interactability is an aspect that characterizes how useful the feature extraction output is for interaction. For a robot trying to grasp for an object, it would be beneficial to know its precise silhouette or to understand its parts. For instance, when grasping for a cup, then knowing where the handle is located is of high practicality. For a template this interactabililty is low; for a structural feature the interactability is much higher.

Deep Nets: the design of an appropriate network architecture takes a long time; the recent creation of the so-called 'EfficientNet' shows that it can take years. Once designed, the network can be deployed for different types of inputs. For a given type of task, one typically chooses a certain type of network and then operates with it. Deep Nets have a large support by Big Tech companies.

The duration for training a blank network is long, learning can take days or weeks. And the amount of necessary image material to fully train a network is large, several hundreds of images per category. But for many applications it is sufficient to take a trained network and adjust it to the specific classes of the application. That adjustment is called *transfer learning* of which there exists two kinds: one is to retrain only the last layer of the network, and that takes only minutes; the classification accuracy with such a retrained network is good, but can be optimized by adjusting the entire architecture, in which case training takes a bit longer, i.e. up to several hours. The ease of transfer learning makes this approach so convenient for creating consumer applications. failure does not carry any large risks.

The duration for feature extraction, also called *inference time*, is relatively long, as the image is passed through a large architecture with Megabytes of weight parameters. The image vector is typically of size n=512, 1024, 2048 or 4096, multiples of two to facilitate computation.

Deep Nets are quasi omnipotent classifiers and can be easily deployed to basic-level, super-ordinate and sub-ordinate categories. The networks are enormously robust to fragmented input due to their gradual, hierarchical feature extraction and integration; for example a street sign hidden partially behind the leaves of a tree is more reliably recognized with a Deep Net than with any other approach.

Deep Nets also provide segmentation abilities with a convenience, that no other approach had offered before: a segmentation network is able to locate the silhouette of objects very well after training it with a few example images.

A downside of Deep Nets is that their output is not readily usable for interaction as the network itself has never precisely located any contours or regions due to its gradual local-global feature extraction process. In order to obtain contour information, one has to analyze the many units and layers of the network; even a learned segmentation network provides only the silhouette in a map, and not an interpretation of its contours. The interactability of the network output is therefore moderate only.

The scalability of a Deep Net is challenging due their huge architecture and the vast amount of weights involved (Megabytes). It can typically take only moderate image sizes, up to several hundred pixels per side. Large image sizes are downsized to the networks maximum allowable input size. For very large image sets, Deep Nets are deployed only with much optimization, i.e. the use of CUDA cores to accelerate computation.

Local Features: the methodology is relatively general enough, that it can be fairly easily adapted to various tasks, in some sense easier than Deep Nets, but the approach has not provided the same performance accuracies. The learning duration is however much shorter than that for Deep Nets and is often based on

Table 1: Characteristics of the three approaches. The aspect label 'variable' means task-dependent or algorithm-dependent. The comparison is vague of course, as different approaches excel at different tasks, but it should convey the gist of the approaches.

aspect	**Deep Nets**	**Local Feature**	**Structural Desc.**
[0] representation	pyramid of local, individual templates	transformed local patches as lists or pyramidal statistics	spatial relations of parameterized segments
[1] duration design	long	moderate	moderate
[2] duration learning	long (from scratch) moderate (transfer total) short (transfer last layer)	short	shortest
[3] # images for training	many	one-shot,..,several	few
[4] duration feature extraction (inference time for one image)	long	short	shortest
[5] image vector (dimensionality)	512-4096	500 to 20K	several 100 to 1000
[6] weight parameters	MegaBytes	-	-
[7] hyper-parameters	few	few	many
[8] recognition strength	super, basic, sub	place recognition Robot Vision	shapes & silhouettes Machine Vision
[9] robustness to clutter	high	high	low
[10] segmentation	top-down	-	bottom-up
[11] interactability	moderate	-	high
[12] image size	limited	unlimited	variable
[13] scalability	moderate	best	variable
[14] resource requirements	large	medium	tendentially small
climate compatibility (eco friendliness)	low	acceptable	high

clustering and Nearest-Neighbor Search, two issues that are part of the field of Pattern Recognition. In some tasks, one-shot learning is possible; in general it requires only few tens of images to fully exploit the discriminative capacity of local features. There are no weights involved and only a few hyper-parameters to be set, but not notably more than for Deep Nets.

Feature extraction is completed in relatively short time and from images of arbitrary size; inference time is relatively short. An image vector can contain several hundred or thousands of components, sometimes up to 20K; it can therefore be larger than that of Deep Nets, depending on the application.

The approach is not as discriminative as a Deep Net trained for basic-level or super-ordinate categories. The recognition strength with this approach lies rather with more specific categories, and in particular it is useful for place recognition, the ability of a robot or vehicle to recognize its environment. The approach deals well with cluttered objects, because representations are based on local patches: an occluded object still exhibits sufficient local patches that allow proper recognition. The approach is not suitable for segmentation, neither does it permit an interaction with the image content.

The approach can be easily applied to large images and for that reason also scales easily to huge image collections. The computational resources are moderate.

Structural Description: is the classical engineering approach that relies on the detection of segments such as contours and regions. Its great strength is its interactability, because it is designed to understand directly the outline of objects, and for that reason it is used in particular in Machine Vision implementations; the design time is relatively long. For Computer Vision, I have now developed a prototype that is competitive with the other two approaches and shows for the moment a moderate effort to adjust the system to individual tasks; the design time is moderate. The amount of image material necessary to train the system is comparable to that for the Local Feature approach: with a few tens of images, the system has been trained optimally. Learning duration is short as it uses merely traditional methods from Pattern Recognition.

Segment extraction takes relatively long for classical techniques but is much shorter for my prototype techniques. The dimensionality of the image vector for the prototype is comparable to that of Deep Nets. There are no weight parameters, but many hyper-parameters; their tuning is however not more complicated than the adjustment of the hyper-parameters in the other two approaches.

The recognition strength lies rather in basic-level and super-ordinate categorization so far. The classical systems are rather brittle for input that is noisy or partly occluded; the prototype has not been tested yet with regard to that aspect. Segmentation exists only as bottom-up flow and the techniques range from simple to complex.

The Structural Description approach can tackle any image size in principle but only our prototype techniques show sufficient speed. The approach is scalable in principal. The resource requirements are somewhat smaller than that of the Local Feature approach.

A summary of the prototype is describe here:

https://www.researchgate.net/publication/360033329
https://www.researchgate.net/publication/323756034

We partially introduce it in Sections 14.4, 17.2, 17.4, 17.5.2 and 24.

3.6 Choice of Approach, Trade-Offs

The choice of which approach is to be pursued for a task is guided by aspects such as availability of computational power, the amount of image material to be processed, the amount of annotated image material available for training (see again Section 2.3), the desired recognition accuracy, etc.

If computational power is available, such as a recent cell phone or a PC, then Deep Learning is the immediate choice. For the occasional processing of a single image or a short video sequences, a cell phone might suffice; for processing many images or multiple video sequences, a PC is necessary.

In some scenarios, processing speed is of particular concern, such as in robotics or autonomous vehicles, where one desires real-time processing (Fig. 9). In those scenarios, engineering methods are still preferred. Or if one intends to classify the image material of the internet, then even simpler methods are sometimes deployed, such as whole-image template matching (of thumbnail images).

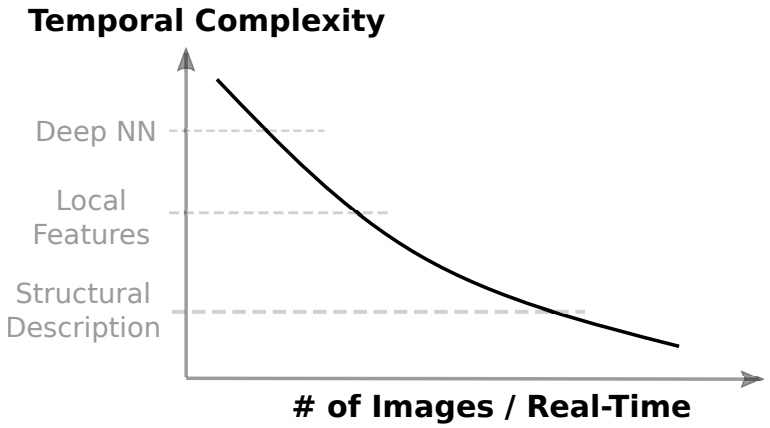

Figure 9: Choice of method in relation to the amount of image material to be processed (aspects scalability and complexity). y-axis: temporal complexity of the method; x-axis: number of images to be processed, or real-time requirement. If few images are to be processed, then Deep Learning (Deep Neural Networks; Deep Nets) is probably the preferred choice. The more images we intend to process however, - millions of images or video processing - , the faster must be the methods, in which case Feature Engineering methods still hold their ground to some extent. There exist also Deep Network implementations that can classify images in microseconds.

We now comment on some other issues: one is a limitation of the classification process, and another is the trade-offs we face for object recognition.

The **classification** accuracy decreases with increasing number of categories we wish to discriminate. A few categories can easily be discriminated to more than 95 percent perhaps, depending on the amount of intra-class variability and the context in which they appear. The less variability is permitted and the more homogeneous the context, the easier it is to discriminate between the classes. Because we can discriminate a few classes relatively easily, it is tempting to think we can build one-versus-all classifiers and then combine them, as is done in ensemble classifiers (Section H.1.5). This is however not quite as straight forward for a number of reasons, one being that the discriminative capacity not really grows with that trick. Yet it is a direction worth pursuing.

For **object recognition**, we are faced with a speed/accuracy trade-off (Fig. 10). The faster we are required to find our target objects, the less accurate will be the method inevitably, as it is computationally expensive to apply a good face detector at every location in the image. Furthermore, if our system also requires to find the exact object outline, a segmentation process, then this trade-off is aggravated. The more classes we wish to detect, the more complicated becomes the entire system, as we need more representational capacity to discriminate the classes.

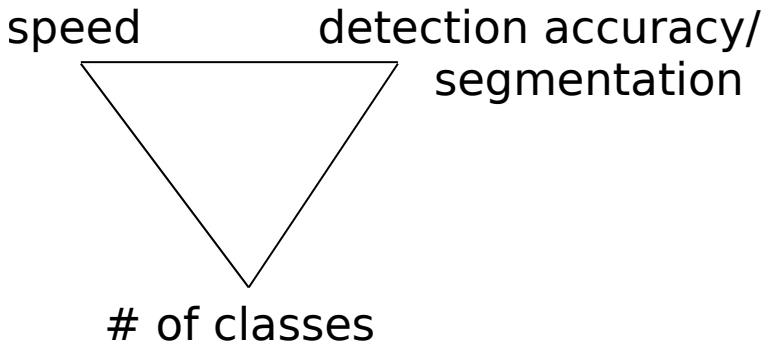

Figure 10: The trade-offs in an object recognition task. Ideally, the algorithm would detect objects as fast as possible, but fast methods also suffer from lower detection accuracy (or segmentation accuracy); this trade-off is indicated by the horizontal line (top line of triangle). The more object categories (classes) we would like to discriminate, the more difficult is it to maintain a high detection rate and the slower becomes the system - those two trade-offs are indicated by the two oblique lines. When we start to develop an object recognition system, a good starting point is to deploy the YOLO network for reference (Section 11.3).

We give two examples to understand some of these conflicts. One is face detection in photographs of street views, as recorded by a company providing maps (Google, HERE, etc.). The amount of image material is so vast, that applying a Deep Net everywhere is computationally too expensive; instead one chooses to run a simple and fast detector that completes the blurring in reasonable time, at the downside of some misses. In order to avoid misses (and potential law-suits), the detector's parameters are set such that it tolerates a lot of false alarms, and hence the street view imagery contains quite a few unnecessarily blurred regions.

Another example is the detection of crosswalks (zebra crossings) for autonomous vehicles. Because those objects appear in specific position of the 2D image - and is thus hardly an actual detection task - , we can set the focus of the detector on that position and run a relatively expensive but accurate Deep Net.

The two examples were tasks for which only one category is detected. If we develop a system for multiple categories, then a good starting point might be to deploy the so-called YOLO network (Section 11.3). It can serve as a reference to observe which aspect one wishes to improve. More details follow in that section.

4 Image Processing 0: Simple Manipulations

To get acquainted with some of the basics, we perform a few simple image manipulations in this section. Firstly, we learn about the image format and some basic operations such as thresholding and data type conversions (Section 4.1). Then we mention some of the image enhancement techniques that are occasionally used for computer vision systems (Section 4.2). Finally, we point out that for carrying out simple arithmetic operations with the entire image, there exist sometimes specific software functions (Section 4.3).

4.1 Image Format, Thresholding, Conversion

A typical digital image, for instance a jpeg image, comes as a three-dimensional array. The first two dimensions correspond to the spatial axes x and y. The third dimension holds color information in three chromatic channels, namely red, green and blue (RGB). The color values generally range from 0 to 255 and are stored as unsigned integers, specifically as `uint8`, the number 8 standing for 8 bits, a data-type designed to hold exactly that range ($255 = 2^8 - 1$). For each pixel then, there exist 24 bits (3*8 bits), which allows to store 256 x 256 x 256 $\approx$ 16.7 million colors. Despite that rich color information, it is often more convenient to computer only with a gray-scale version of that image, thus reducing the information back to 8 bits per pixel. To convert a RGB image into gray-scale image, one converts the color values to a gray value L by adding the three components according to a specific ratio, for instance:

$$L = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B \tag{1}$$

In Matlab one can load an image with the command `imread`. To convert it into a gray-level image there exists the function `rgb2gray`. To display the image we use the function `imagesc` (image scale) and for that purpose we initialize a figure with the function `figure`. The function `clf` clears the figure. With the command `subplot` we can pack several images into the same figure. The following code shows how to use those commands, its output is shown in Figure 11.

```
clear;                              % clear memory
Irgb    = imread('yellowlily.jpg'); % load jpg image
Igry    = rgb2gray(Irgb);           % convert it to gray-scale
IrgbCen = Irgb(400:1200,300:900,:); % zoom into center
Igreen  = Irgb(:,:,2);              % green channel only
BWflw   = Igry>100;                 % thresholded (black-white image)
Iblur   = conv2(single(Igry),ones(25,25)/625);    % blurring the image

%% -----    Plotting    ------
figure(1); clf; [nr nc] = deal(3,2);
subplot(nr,nc,1); imagesc(Irgb); title('Original');
subplot(nr,nc,2); imagesc(Igry); colormap(gray); title('Gray-Scale');
subplot(nr,nc,3); imagesc(IrgbCen); title('Sub-Selection (Zoom)');
subplot(nr,nc,4); imhist(Igreen);   title('Histogram of Green Channel');
subplot(nr,nc,5); imagesc(BWflw);   title('Black-White (Logical) Image');
subplot(nr,nc,6); imagesc(Iblur);   title('Blurred Image');
```

To select a part of the image - see comment 'zoom into center' -, we specify the row numbers first - vertical axis first -, followed by specifying the column numbers - horizontal axis. That is, one specifies the indices as for matrices in mathematics.

Black-White (Binary) Image We can threshold an image by applying a relational operator, see line `BWflw = Igry>100`, in which case the image is automatically converted into a *logical* data-type, that is `true` or `false`, namely one bit (value one and zero respectively). An image of that data-type is also called a *binary* or *black-white* image sometimes, hence the variable's name `BW`. Pixels with value equal one are also called ON pixels, those with value equal zero also called OFF pixels.

In the code example above we attempted to separate the flower from its background, a process also called *foreground/background segregation*. We have chosen the threshold somewhat arbitrarily and of course it would make sense to choose a threshold based on a histogram, coming up in the next paragraph.

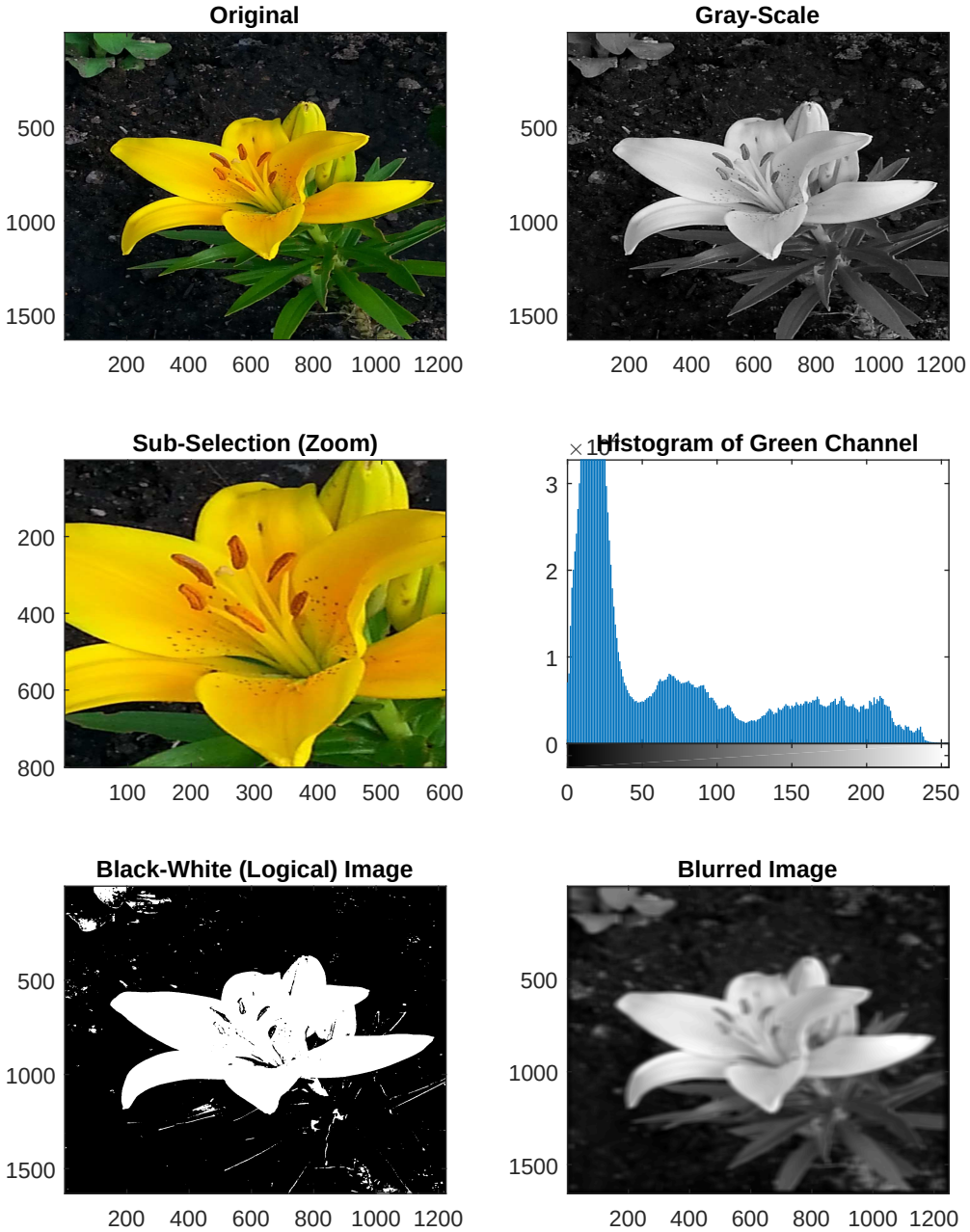

Figure 11: Some simple image manipulations. See also **P.1** for loading and saving images and videos.
Upper Left: original RGB image; each pixel is represented by 24 bits (3 chromatic channels $\times$ 8 bits).
Upper Right: gray-level: RGB values combined to a single luminance value according to Eq. 1; each pixel is now represented by 8 bits only (values ranging from 0 to 255).
Center Left: zoomed image; selection occurs with matrix-style indexing (rows/columns).
Center Right: image histogram of the green channel: the x-axis represents the chromatic value, ranging from 0 to 255; the y-axis represents the pixel-count.
Lower Left: result of thresholding the gray-level image at value equal 100: white represents ON pixels (true, value equal one), black OFF pixels (false, value equal zero).
Lower Right: gray-level image blurred with an average filter, also called *box filter*: at each pixel the average over some neighborhood is taken; in our case we chose a 25x25 pixel-neighborhood.

Histogramming Intensity Values A histogram of intensity values sometimes reveal the dominant color or gray-level of certain regions. If we have a gray-level image only, then we form a histogram with only those values. If we have a color image, we can form histograms for each chromatic channel. In Fig. 11, center right, the histogram for the green channel is plotted. Such a histogram can be exploited to find an optimal thresholding value for segmentation, something which we elaborate on in Section 14. Or the histogram can be used for classification, as it already represents an image vector (Section 3.3); that is a rather crude abstraction of an image, but one that was used in early image retrieval systems (Section 18). The color histogram of an image is also called a *global descriptor* sometimes, as it describes the image as a whole, neglecting local detail (introduced already in Section 3.1).

Blurred Image Sometimes it is useful to blur an image because a blurred image helps analyzing the 'coarse' structures of an image, which otherwise are difficult to detect in the original image with all its details. We can blur an image by averaging over a local *neighborhood* at each pixel in the image, in Matlab done with the function `conv2`. In our example we take a 25x25 pixel neighborhood, generated with `ones(25,25)` and merely sum up its 625 pixel values: this summation operation is done for each pixel in the image. Normally one divides the sum by the filter size in which case one speaks of a *box filter*. We will come back to more blurring filters in Section 6.

Data-Type Conversion (Casting) The function `imread` returns a jpeg image as data-type `uint8`, which is not very practical for certain computations. For many image-processing functions we need to cast (convert) the image into a floating-number data-type. In Matlab that can be done with the functions `single` or `double`, returning lower and higher precision respectively. In the above code, we did that casting for the function `conv2`, but we could also write a separate line if desired, `Irgb = single(Irgb)`.

Many functions are flexible and will produce an output of the same data-type; others expect a specific data-type as input; some functions produce a specific data-type as output such as the thresholding operation. It is best to be always aware of what data type an image is - or any variable -, and what type the functions expect and produce.

In Python the code looks very similar, but we need to import those functions from modules. In particular the module `skimage` holds a lot of functions for computer vision and image processing, see also Appendix O:

```
from numpy           import arange, ones, histogram
from skimage.io      import imread
from skimage.color   import rgb2gray
from scipy.signal    import convolve2d

Irgb    = imread('someImage.jpg')   # loading the image
Igry    = rgb2gray(Irgb)            # convert to gray-scale
IrgbCen = Irgb[100:500,400:700,:]       # zoom into center
Igreen  = Irgb[:,:,2]                   # green channel only
BWflw   = Igry>0.3                      # thresholding (black-white image)
Iblur   = convolve2d(Igry,ones((25,25))/625)  # blurring the image

#%% ---- Plotting
from matplotlib.pyplot import figure, subplot, imshow, plot, hist, title, cm
figure(figsize=(10,6)); (nr,nc) = (3,2)
subplot(nr,nc,1); imshow(Irgb); title('Original')
subplot(nr,nc,2); imshow(Igry, cmap=cm.gray); title('Gray-Scale')
subplot(nr,nc,3); imshow(IrgbCen); title('Sub-Selection (Zoom)')
subplot(nr,nc,4); hist(Igreen.flatten(),bins=arange(256));   title('Histogram of Green Channel')
subplot(nr,nc,5); imshow(BWflw);   title('Black-White (Logical) Image')
subplot(nr,nc,6); imshow(Iblur, cmap=cm.gray);   title('Blurred Image')
```

Note that in Matlab the function `rgb2gray` returns a map of data-type `uint8` - if the input was of type `uint8`. In Python however there exist many functions to read an image and some of them perform scaling to an interval $\in$ [0, 1.0].

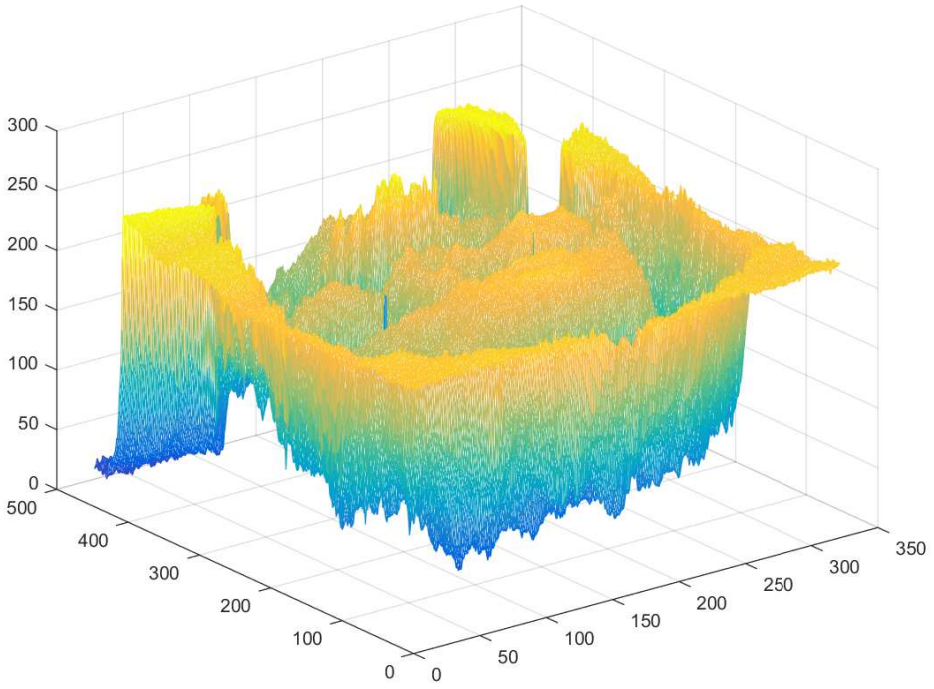

Figure 12: An image as observed from a three-dimensional perspective (command `mesh` in Matlab). The image appears as a landscape whose elevation - the vertical axis - represents intensity with values ranging typically from zero to 255 (for images coded with 8 bits). It is not easy for a human to understand the semantic content of a scene from this perspective, because the human visual system is trained to interpret two-dimensional arrays. But this perspective illustrates better the challenges faced by a vision system. (To what image in this book does this landscape correspond to?)

Topology To obtain an impression what type of stimulus an image represents, we recommend to observe the image with the function `mesh`, as given in the last line in the Matlab code block. This illustrates better that the image array holds an 'intensity landscape' (Fig. 12). For that reason computer vision scientists often borrow terminology from the field of topology to describe the operation of their algorithms, such as the term *watershed* in case of a segmentation algorithm; or *ridge* and *river* contours for lines representing those features in the landscape (Section 17.2).

4.2 Image Enhancement

Image enhancement serves to improve the visual appeal of an image. It therefore serves rather the human observer, but some of the techniques are also used in preparation for recognition processes - as an preprocessing to image processing. Two techniques are particularly noteworthy: contrast manipulation and filtering to eliminate noise.

4.2.1 Contrast Manipulation

wiki Histogram_equalization

If an image shows low contrast, then it can be manipulated to show higher contrast by shifting its pixel values according to some distribution. The manipulations are typically based on the image histogram (Fig. 11 center right). In an image with low contrast, the histogram shows a peak at either side, see Fig. 13 left side. During the equalization process, that histogram peak is moved more toward the center (right side in

Figure), based on an *input histogram* distribution. This technique is sometimes used to manipulate images in collections to show roughly equal intensity distribution, as this helps improving learning results when training recognition systems.

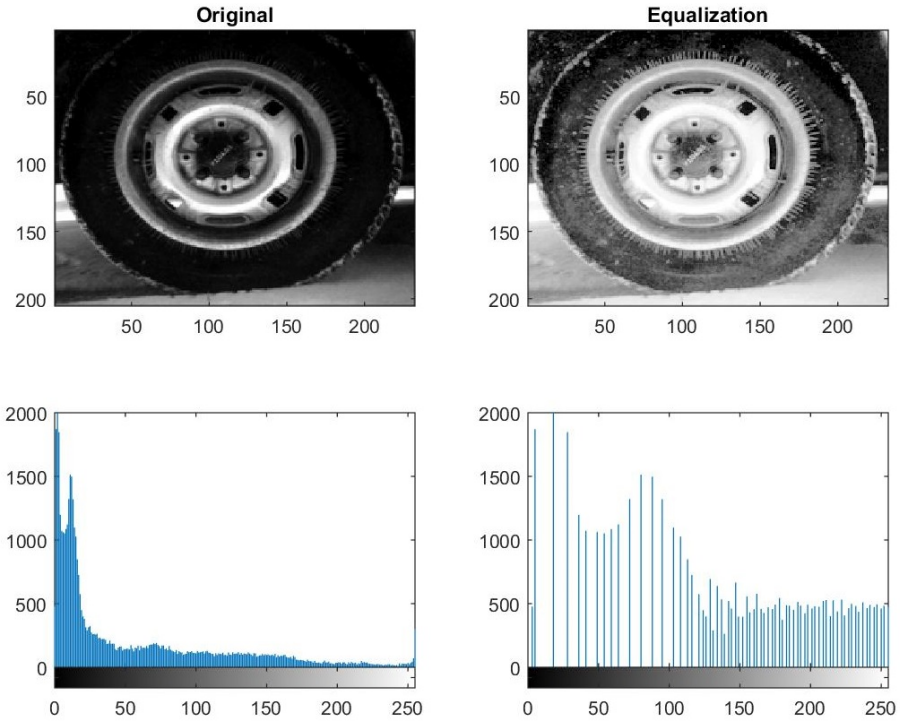

Figure 13: Image enhancement with simple manipulations using its histogram for redistribution. This has visual appeal primarily, but is also carried out for learning classification tasks sometimes.
Left Column: original image (top) and its corresponding intensity histogram (bottom).
Right Column: the equalized histogram (bottom), which serves to reassign the pixels values in the original image resulting in an enhanced image (top).

In Matlab the functions `imadjust`, `histeq` and `adapthisteq` carry out such image enhancements. For instance when using `histeq` without any parameters,

```
Iheq    = histeq(Igry);          % histogram equalization with a flat 64-bin histogram
```

it will automatically assume a flat input histogram made of 64 bins.

In Python those functions are within module `skimage.exposure`. For instance with `equalize_hist` the histogram equalization is carried, whereby there the default input histogram consists of 256 bins.

In OpenCV [py] available under `equalizeHist`.

4.2.2 Noise Elimination

Sometimes, images happen to be 'noisy'. For example, digitization of old photographs leads often to images in which single pixels stand out of their immediate neighborhood, a type of noise that is also called *salt-and-pepper* noise, because those stand-out pixels appear like grains of salt (bright) or pepper (dark) on an image. In that case we can take a median filter, a filter that replaces a pixel value with the median value of its neighborhood. Often, one chooses a 3x3 neighborhood only, that is the median is taken for those 9 values only. Larger neighborhoods can also be used.

In Matlab the function for median filtering is called `medfilt2` and by default it uses a 3x3 neighborhood:

```
    Igry    = medfilt2(Igry);          % median filter for a gray-level map
```

If one intends to denoise a color image, then one would apply the median filter to the individual chromatic channels.

In Python the function `medfilt2d` can be found in module `scipy.signal`.

In OpenCV [py] available as `medianBlur`.

The median filter is sometimes used in video processing as pixel values can show larger intensity variations due to the longer exposure of the sensor and the camera movement.

4.3 Image Arithmetics

Image arithmetics is a term for the pixel-wise arithmetic operations between two images; or between an image and a scalar; or between two patches. Such operations are used heavily for template matching, coming up in the next Section.

In principal these operations can be carried out using the usual operation symbols in software packages, '+' for addition, '-' for subtraction, '*' for multiplication, etc. Using those general operators can be slow however. To speed up image arithmetics, software packages often provide specific functions that carry out these operations a bit faster.

When using one of the packages/libraries, one needs to pay attention how the software deals with operations where the result exceeds the limits of the data type (depth). This is handled differently for each software.

In Matlab :

```
imadd(I1,I2);
imsubtract(I1,I2);
imabsdiff(I1,I2);
immultiply(I1,I2);
imdivide(I1,I2);
imcomplement(s1, I1, s2, I2, a);  % s1*I1 + s2*I2 + a
```

In Python there does not seem to exist a specific set of functions. One would use the OpenCV functions instead, see next.

In OpenCV [py] :

```
In    = add(I1, I2)
In    = subtract(I1, I2)
In    = addWeighted(I1, 0.5, I2, 0.5, 0)
```

Some arithmetics are particularly frequently used:

Sum-of-Absolute Differences (SAD): is a measure of dissimilarity between image blocks - or between two entire images. It is calculated by taking the absolute difference between each pixel in the original block and the corresponding pixel in the block being used for comparison, `Df=imabsdiff(B1,B2)`. These differences are summed to create a simple metric of block similarity, the L1 norm of the difference image or Manhattan distance between two image blocks. wiki Sum_of_absolute_differences

Sum-of-Squared Differences (SSD): the square of the above SAD measure. Deals better with outliers, but is also costlier to compute due to the use of the square operation, see Appendix K for a discussion of distances. This increase in computation may seem negligible for a few images, but shows its effect when we manipulate videos for example or deal with millions of images.

5 Recognition 0: Feature-Less Matching, Profiles

Here we introduce some simple techniques to compare entire images or to spot conspicuous features in an image, without carrying out any particular feature extraction. The image is treated quasi as a feature template (see Section 3.1), and we match the corresponding pixel values between images. For recognition, these are rather outdated techniques, but still occasionally used, in particular when retrieving or classifying the vast amount of image material on the internet; or when carrying out place recognition (in a robot application); or when tracking objects. These simple techniques can also be deployed as a reference; should our engineered (or learned) feature-extraction implementation not be substantially better than the use of such simple matching techniques, then it might not be worth employing our feature-extraction implementation, or we may not have tuned our implementation properly. For binary (black-white) images, there exist particular matching techniques (Section 5.2). We also introduce a simple detector for face part localization to demonstrate that even simple image manipulations can sometimes reveal useful information (Section 5.3).

5.1 Color/Gray-Scale Images

The simplest measure to compare two images is to take their pixel-by-pixel differences and to sum those differences, such as the Sum-of-Absolute Differences or the Sum-of-Squared Differences, as introduced in the previous Section 4.3. If the the images are not of the same size, then we resize one image to match the other in its dimensionality.

This type of pixel-by-pixel comparison is essentially one-dimensional array matching, as the images are flattened to a single vector. The pixels within the same image are not related to each other at all and no feature is extracted. Hence, this is sometimes also called *feature-less* matching or *non-parametric* matching. This form of matching can be used for retrieval and is now called whole-image matching (Section 5.1.1); it can also be carried out more locally for a variety of different tasks, now called partial-image matching (Section 5.1.2).

Perhaps the second simplest measure to compare images is to take their histogram of chromatic pixel values (Section 5.1.3), and then to match those histograms. That type of histogram matching is considerably faster, as we do not match the entire image array, but only a summary of it. Some implementations of histogram matching use the so-called earth mover's distance (EMD), see wiki Earth_mover's_distance.

Histogram matching can be considered already a first step toward feature extraction, but as we do not have any (geometric) parameters, this can be still considered non-parametric.

5.1.1 Whole-Image Matching for Primitive Image Retrieval and Classification

Many of the images on the internet are annotated by some key words, often provided by the user who uploaded them; sometimes the tags are included in the header of an image format. This tagging information can be exploited to classify a given image: we compare our *query* image against many internet images - our database so to say - and then retrieve the most similar images, let us say the 50 most similar images. If of those 50 images 22 images contain a text tag 'beach', then our image is likely to be of that category. This classification decision is also called *majority vote* in Pattern Recognition.

This is an unusual form of classification as we do not have an actual model, but exploit an abundance of tagging data. And it can only work if we compare our query image against tens of thousands of images, ideally millions of images. Such a collection is provided in:

`http://horatio.cs.nyu.edu/mit/tiny/data/index.html`

described in the study

Torralba A, Fergus R, Freeman WT. 80 million tiny images: A large data set for non-parametric object and scene recognition.

Each image of that collection contains a scene or object label; each image was reduced to a size of only 32x32 pixels to allow matching in reasonable time. In the following, we give a code example of the retrieval method, using - for simplicity - an artificial database of small image count, that we call `AIMG`, see now code example in **P.2.1**. We also create two artificial query (testing) images `Iqu1` and `Iqu2`.

To perform the matching, the database and query images are flattened to row vectors. For the database, the resulting matrix `VIMG` then holds the number of images, times the number of pixels (1024) multiplied by

the number of chromatic channels (3):

```
VIMG    = single(cat(4,AIMG{:}));              % [32 32 3 nImg]
VIMG    = reshape(VIMG,[prod(szImg) nImg])';  % [nImg 3072]
```

This matrix arrangement is the general format for Machine Learning (Pattern Recognition) applications: number of samples times number of feature dimensions (Appendix H). Most Machine Learning algorithms in software applications take this type of format as input - occasionally, some applications require the matrix to be rotated (resulting in `[nDim nImg]`). Continuing with our example, we also flatten the two query images:

```
Vqu1 = single(Iqu1(:)'); Vqu2 = single(Iqu2(:)');
```

Now we can conveniently measure the similarity between images using the function `bsxfun`; the function takes the query vector to each row in the matrix:

```
Df1 = sum(bsxfun(@minus, VIMG, Vqu1).^2,2);     % diff to 1st query image
```

and as this implements the Sum-of-Squared Differences; it is precisely speaking a *dissimilarity*. `Df1` is a vector that holds the dissimilarity between query image 1 and all database images. We order `Df1` in increasing manner and obtain a ranking in variable `Rnk1`, which holds the indices of the nearest images in ascending order. Now we observe what label the nearest images have. We will have more on this type of analysis in Section 18.

Taking these dissimilarity measurements is a relatively time-consuming procedure, and one can speed up the comparison by different tricks. One trick is to use the Principal Component Analysis, a transformation that reduces the matrix by tens of percent, thereby loosing also a bit of precision, but gaining a tremendous reduction in calculation duration. This transformation is now a serious step into Pattern Recognition, and we rather refer to other sources for further explanations (Section H.1.4).

Another trick to speed up the comparison is to use techniques of Nearest-Neighbor Search, a technique that determines only the nearest neighbors, and not the entire ranking; the code example shows how to use a KD-Tree, see also Appendix K.2 for an introduction to that topic. This type of distance measurement will come up later again for matching of descriptors (Section 12.3).

This matching scheme will favor only images, whose structure is - not surprisingly - exactly the same as that of the query image. But perhaps there is no such similar scene but only slightly shifted variants, or mirrored variants in the database. In order to to catch such variants we can present the query image with corresponding manipulations, that is once shifted, once mirrored, etc. This corresponds to creating an augmented dataset, first mentioned under Section 2.3 (elaborated in Appendix J).

5.1.2 Partial-Image Matching

Pixel-by-pixel matching can also be used to find an object in an image. We have a target object that we try to find in an image and the target is specified as an image, our template, and we move that template across the image to find the best matching location, a simple but precise scheme. When only a small (local) part of an image is matched in this pixel-wise manner, then this is also called block-matching sometimes, and is still used for tasks such as tracking (Section 19), registering two images (Section 21.4) and for establishing correspondence (Appendix E).

For object detection, Matlab offers the function `normxcorr2`, that moves the template across the image, whereby the distance measurement is the normalized cross-correlation, a rather sophisticated measure; and under certain circumstances, the function preprocesses the image using the Fourier Transform as this increases the matching speed. We skip this entirely and refer merely to a code example for that, see **P.2.2**.

5.1.3 Histogramming

A first step toward abstraction of an image is to take the intensity histogram of an image (as introduced in Section 4.1); if it is a color image with three chromatic channels, then we generate one histogram for each

chromatic channel and then concatenate the three histograms (**P.2.3**). Matching histograms is also useful for retrieving images; it is also deployed for tracking (Section 19.3). As explained previously, such an image histogram is also called a global descriptor sometimes, as it expresses a characteristic of the entire image.

Scaling and Metrics For all these matching schemes, the output of the comparison can significantly vary depending on what type of scaling technique and metric is applied. A safe way to obtain reasonable results is the use of the Euclidean distance measure applied on normalized vectors, normalized to unit length. There exist different scaling possibilities, and one may obtain better results with other scaling techniques - one simply has to try out. Instead of the typical distance metric, we can also try similarity metrics, in which case we need to revert the sorting, i.e. `[SimO Rnk] = sort(Sim,'descend')`, see also Appendix K. For matching histograms, the use of the χ^2 measure typically returns better results wiki Chi-square_distribution. However, not all metrics are suitable for KD-Tree search.

5.2 Binary Images

Binary images are sometimes obtained after a segmentation process (Fig. 11) or we have shapes or scene that are expressed in a black-white image, i.e. after contour pixel detection. To compare the pixels of such black-white maps, one can deploy the above mentioned pixel-wise differences of intensity values, but we likely obtain better results, if we either manipulate the image, or we use only the ON pixels for matching. For example we can generate the distance map for a black-white image, as will be introduced in Section 16.4.1, whose output is analogous to a gray-scale images as it holds scalar values in the entire map. That distance map is then matched pixel-wise as for gray-scale images (as explained above). This type of matching is also called *Chamfer* matching, a term that will be explained in that section (16.4.1).

Another way of comparing binary images is to use only the coordinates of the ON pixels and measure the pair-wise distances between them. We firstly extract the row and column indices `[Rw Cl] = find(BW);`, and ignore the OFF pixels. Then we take distance measurements between those ON points of two images. As the amount of ON pixels differs between (most) images, the distance measure needs to be able to deal with unequal list lengths. Such a distance measure is usually based on pairwise point-matching, whose (pairwise) distances are then analyzed to arrive at a scalar distance value expressing the total distance between the two maps. One such analysis is the so-called Hausdorff distance (wiki Hausdorff_distance), that observes in particular the maximum distance of all pair-wise (point) distances.

5.3 Profile - Face Part Detection

A profile is a cross-section of the intensity landscape along specified coordinate values. It is a one-dimensional intensity signal that holds valuable information in controlled situations; bar-code detection is a use case. For example taking the row `I(25,:)` of an image, would represent a horizontal profile; taking the column `I(:,10)` would be a vertical profile. Profiles can also be taken along any sequence of line segments. The advantage of such profiles is they are easy to analyze, but this only makes sense if we know where to measure the profile values. In case of bar code identification, the scanned profile results in a rectangular wave function of varying widths, that can be relatively easily analyzed; Matlab contains an example for that task.

In Matlab there exists the function `improfile` for which we specify the `x` and `y` coordinates of the line segments.

In Python an equivalent function can be found in module `skimage.measure` with name `profile_line`.

In the following we introduce the example of facial feature detection in a portrait image. It can be regarded as feature extraction, but does not compare to the feature extraction efforts as in the three principal approaches introduced in earlier sections. To do that we sum the pixel intensity values along each dimension, see Fig. 14, resulting in two summation profiles: one by integrating the pixel values along the horizontal axis (dimension), and one by integrating along the vertical axis. In Matlab this is easily done as follows:

```
Pver    = sum(Ig,1);                    % vertical intensity profile
Phor    = sum(Ig,2)';                   % horizontal intensity profile
```

Those profiles are shown in black in the Fig. 14. In such profiles it is - in principle - relatively easy to locate facial features by observing local maxima, minima, etc. The 'raw' profile sums are a bit 'noisy' however - like most raw signals: they contain too many 'erratic' extrema that make detection of the facial features difficult. We therefore smoothen the profiles a bit by low-pass filtering them. The smoothened versions are shown in magenta in the Figure; now it is easier to locate facial features. Smoothening can be done by averaging

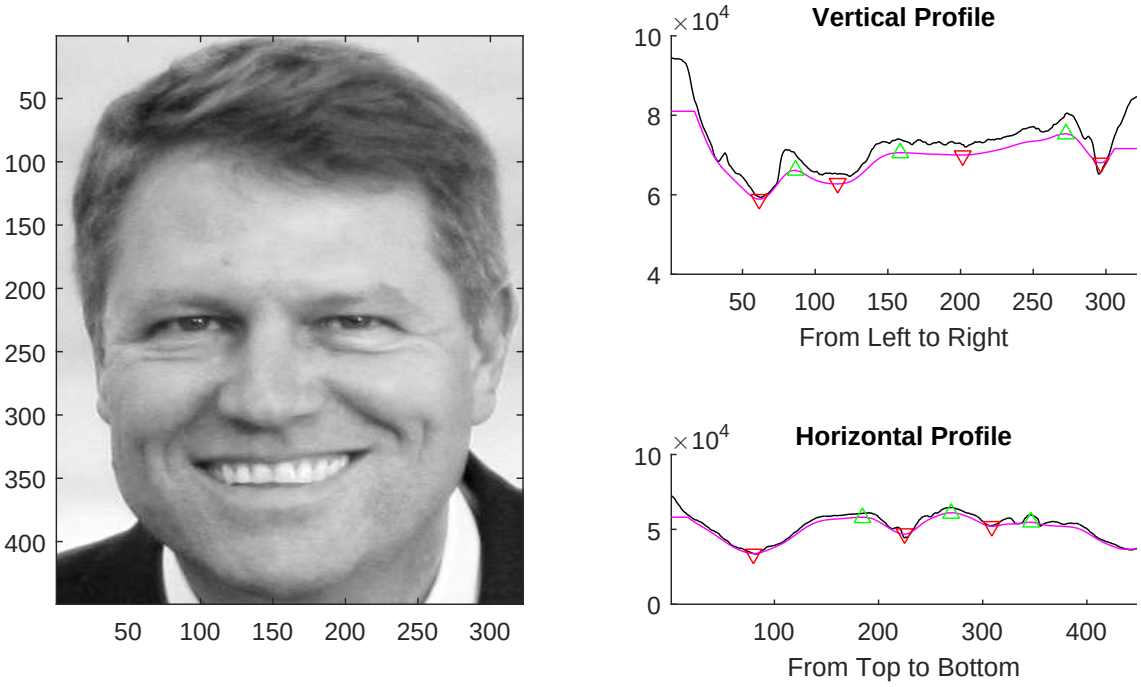

Figure 14: Intensity summation profiles for a face image. The vertical summation profile is generated by summing the pixel intensity values along the y-axis (vertical; column-wise); the horizontal summation profile is generated by summing along the x-axis (horizontal; row-wise). To what face parts do the extrema in the summation profiles correspond to? Code in Appendix **P.2.4**.

over a local neighborhood in the signal at each pixel in the profile. We have done this in an earlier section already to obtain a blurred image in two dimensions with the function `conv2`. Here we do it in one dimension only. And we do so with a so-called Gaussian filter, a function whose shape is a 'bump': it is an elegant way to filter a signal, see Appendix C.2 for its exact shape.

The size of the filter matters: a small size may smoothen the signal insufficiently, a large size flattens the signal too much. We therefore make the filter size dependent on the image size by choosing a fraction of it. The Gaussian values are generated with function `pdf`.

```
nPf     = round(w*0.05);                % # points: fraction of image width
LowFlt  = pdf('norm', -nPf:nPf, 0, round(nPf/2)); % generate a Gaussian
Pverf   = conv(Pver, LowFlt, 'valid');  % filter vertical profile
Phorf   = conv(Phor, LowFlt, 'valid');  % filter horizontal profile
```

If you are not familiar with the convolution process you should consult Appendix B - for the moment the Appendix B.1 on one-dimensional convolution is sufficient. Those concepts are part of the field of Signal Processing.

For extrema detection we can use the function `findpeaks`, which returns the local maxima as well as their location in the signal. In order to find the local minima we invert the distribution. The code in Appendix **P.2.4**

shows how to apply those functions. An explicit code version is available here:
https://de.mathworks.com/matlabcentral/fileexchange/25500-peakfinder-x0-sel-thresh-extrema-includeendpoints-int

In Python we find the equivalent functions in module `scipy.signal`, called `argrelmax` and `argrelmin` for finding maxima and minima respectively (see **P.2.4**).

In OpenCV [py] there does not seem to exist an equivalent function but C code can be found here:
https://github.com/claydergc/find-peaks
which is a translation of the explicit Matlab code mentioned a few lines above.

Note This type of face part localization is somewhat simple, but its temporal complexity is low and that is why this approach is often used as a very first step of an elaborate facial feature tracking system. Many applied computer vision systems consist of cascades of recognition algorithms to progressively carry out a task; such primitive recognition techniques are still used for initializing a recognition process.

6 Image Processing I: Filtering

For many computer vision tasks, it is useful to blur the image and to analyze those different blurs separately. We have introduced the idea of a blurred image already in the previous section, but here we introduce a more systematic approach that generates this blur repeatedly with increasing filter sizes arriving at a so-called *scale space* as shown in Fig. 15. This space will be introduced in the first Section 6.1. Then we introduce the image pyramid, Section 6.2, which is - coarsely speaking - a reduction of the scale space.

For many computations it is also useful to know how the intensity landscape is 'oriented' at each pixel. Specifically, we would like to know the slope of the surface around each image pixel for a small pixel neighborhood. This is expressed with the gradient image, to be explained in Section 6.3. A summary of many of the filtering functions that will be introduced in this section is available in Appendix D.

6.1 Scale Space

wiki Scale_space

Sze p127, s3.5, p144
FoPo p164, s4.7, p134
SHB p106, s4.3

An image is blurred by convolving it with a two-dimensional filter that averages across a small neighborhood. In our introductory example of the previous Section we merely used the mean operation for the neighborhood, also known as box filter. The box filter is fast but simple: it weighs every pixel equally but it makes more sense to weight the center pixel more and to weigh pixels in the periphery (of the neighborhood) less. This is typically done with a Gaussian filter, as we did already for filtering the face profiles (Section 5.3). Here, the Gaussian filter is a two-dimensional function and looks like shown in the first four patches of Fig. 20. It is expressed as $g(x, y, \sigma)$, where x and y are the image axes and where σ is the standard deviation regulating the amount of blur - also called the smoothing parameter. In the language of signal processing one expresses the blurring process as a convolution, indicated by the asterisk $*$. One says, the image $I_o(x, y)$ is convolved by the filter $g(x, y, \sigma)$

$$I_c(x, y) = I_o(x, y) * g(x, y, \sigma), \tag{2}$$

resulting in a coarser image I_c. If you are unfamiliar with the 2D-convolution process, then consult now Appendix B.2 to familiarize yourself with it.

If this blurring is done repeatedly, then the image becomes increasingly coarser, the corresponding intensity landscape becomes smoother, illustrated in Fig. 15. Practically, there are different ways to arrive at the scale space. The most straightforward implementation is to low-pass filter the original image repeatedly with 2D Gaussians of increasing size, that is, firstly with $\sigma = 1$, then with $\sigma = 2$, etc. That approach is however a bit slow and there exist faster implementations, for which we refer to the Appendix B.2.

The resulting stack of images, $\{I_o, I_1, I_2, \dots\}$, is called a *scale space*. In a typical illustration of the scale space, the bottom image corresponds to the original image; subsequent, higher images correspond to the coarser images. In other words, the images are aligned vertically from bottom to top and one can regard that alignment as a *fine-to-coarse* axis. That axis is now labeled σ. The resulting space is then expressed as

$$S(x, y, \sigma). \tag{3}$$

The axis σ can be understood as a smoothing variable: a σ-value equal zero corresponds to the original image, that is, there is no smoothing; a σ-value equal one corresponds to the first coarse image, etc. In applications, sigma values typically range from one to five, $\sigma = 1, 2, .., 5$.

This fine/coarse axis is not to be confused with the terms local/global. Local/global refers to image resolution and feature size, and not to the degree of blur of an image.

Application The scale space is used for the following feature detection processes in particular:

1. Region finding: we can easily determine brighter and darker regions by subtracting the scales from each other, a technique we will introduce in Section 7.1.
2. Verification of structures across the scale (axis): for instance, if a specific feature can be found at different scales, then it is less likely to be accidental, a technique to be used in feature detection in general (Section 12).

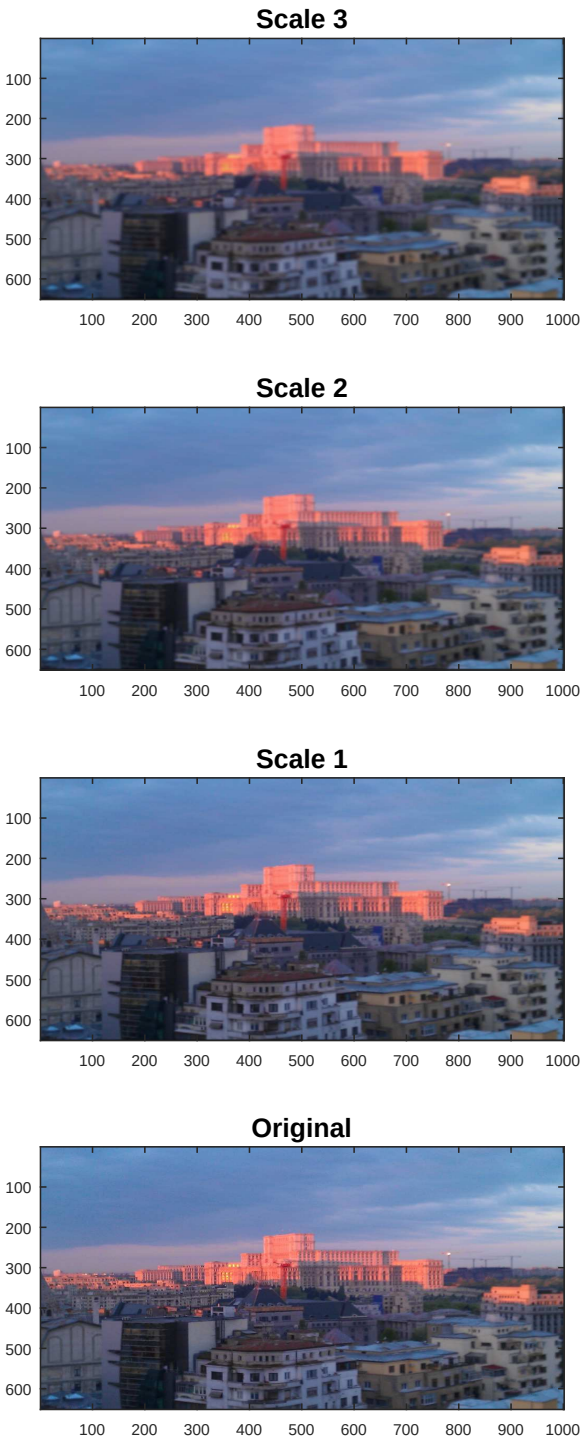

Figure 15: Scale space of an image. Observe the increasing blur from bottom to top.

Original: unfiltered image I_o.
Scale 1: I_o was smoothened with a Gaussian filter of sigma equal one, $\sigma = 1$.
Scale 2: I_o was smoothened with $\sigma = 2$.
Scale 3: I_o was smoothened with $\sigma = 3$.

It is called a space because one can regard it as a three-dimensional space, with the third dimension corresponding to a **fine-to-coarse** axis with variable σ.
At coarser scales (larger values of σ) it is easier to find contours and regions as a whole, but structures are sometimes smeared with other different structures. Coarser scales are often down-sampled to obtain a more compact representation of the scale space, which so forms a pyramid, see Fig. 16. Code in Appendix **P.3**.

(In this case, the filtering was performed for each color channel separately, once for the red, once for the green and once for the blue image.)

[motif: Palatul Parlamentului during a sunrise]

3. Finding more 'coherence': for instance, contours appear more continuous at coarser scales (Sections 7.2 and 17).

In Matlab we can create a Gaussian filter with the function `fspecial` and convolve the image with the command `conv2`:

```
Fsc1 = fspecial('gaussian', [3 3], 1); % 2D gaussian with sigma=1
Fsc2 = fspecial('gaussian', [5 5], 2); % 2D     "      "  sigma=2
Isc1 = conv2(Io, Fsc1, 'same');        % filtering at scale=1
```

The image processing toolbox also offers commands such as `imgaussfilt` and `imfilter` to generate blurs of images and are probably the preferred functions, because those are optimized for speed.

In Python we can use the submodule `scipy.ndimage` that contains the function `gaussian_filter` to blur images:

```
Iblr = scipy.ndimage.gaussian_filter(I, sigma=1.0)
```

In OpenCV [py] the functions are listed under the topic called *Image Filtering*: for Gaussian filtering there is `GaussianBlur`, for general filtering there exists `filter2D`, for box filtering - as done in the introduction section on image processing - there exists `boxFilter`. See for example:
`https://docs.opencv.org/4.0.0/d4/d86/group__imgproc__filter.html`

Generating such a scale space gives us more information, that we then exploit to improve our feature extraction process. This gain in information comes however at the price of more memory usage, as we now have multiple images. And multiple images means more computation when searching for features. But we can reduce this scale space, without much loss of information by 'compressing' the coarse scales, which leads to the *pyramid*, treated in the next section.

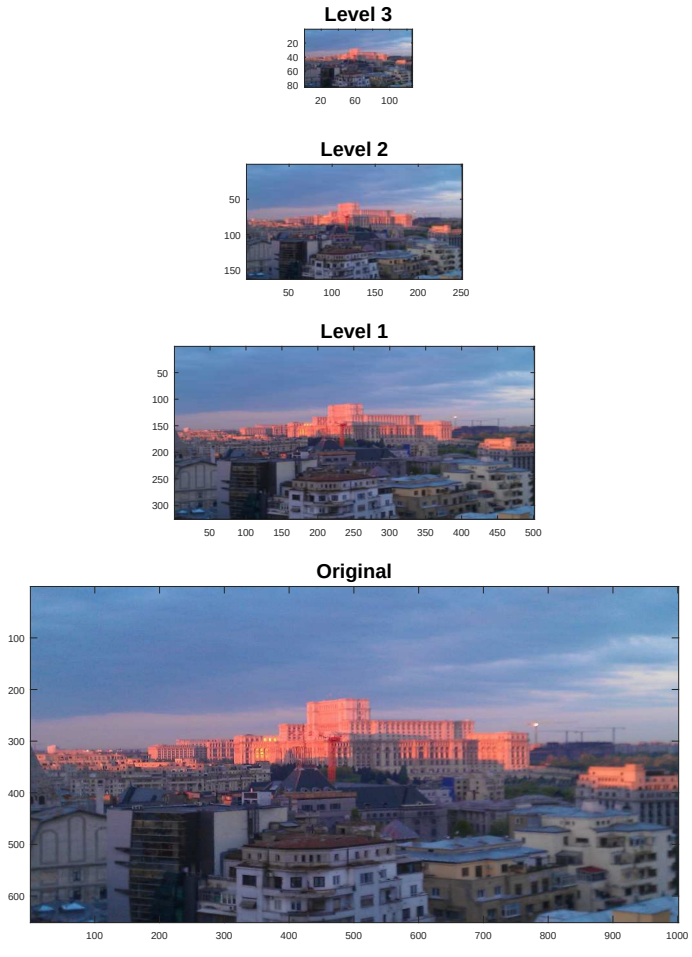

Figure 16: Multi-resolution pyramid of an image. The images of the scale space (Figure 15) are down-sampled by every second pixel: the resolution is halved with each scale along the fine-to-coarse axis.

Original: P_0 = original image I_o
Level 1: P_1 = sub-sampled of I_{c1} ($I_{c1} = I_o * g(1)$)
Level 2: P_2 = sub-sampled of I_{c2} ($I_{c2} = I_o * g(2)$)
Level 3: P_3 = sub-sampled of I_{c3} ($I_{c3} = I_o * g(3)$).

It is called a pyramid, because one could align the images such that they form a pyramid - for simplicity we show the images frontally. Code in Appendix **P.3**.

6.2 Pyramid

wiki Pyramid_(im

Operating with the entire scale space S is computationally expensive, as it is rather large and to some extent redundant. Because coarser scales do not possess any fine structure anymore, it makes sense to subsample them: for each new, coarser scale I_c, only every second pixel is taken, once along the horizontal axis and once along the vertical axis (illustrated already in Fig. 3.2). We so arrive at the (octave) pyramid with levels P_0 to P_3 for instance, see Fig. 16. In higher-level languages we can down-sample 'manually' by selecting every second row and column, i.e. sub-sampling rows with:

```
Idwn = Isc1(1:2:end,:); % sub-sampling rows
```

followed by sub-sampling columns by

```
Idwn = Idwn(:,1:2:end); % sub-sampling columns
```

In Matlab there exists also the function `downsample`, which however works for rows only - if the input is a matrix; we then need to flip the matrix to down-sample it along the columns. Matlab also offers the function `impyramid`, which carries out both the Gaussian filtering as well as the down-sampling. But we can also operate the other direction and up-sample:

```
I1  = impyramid(I0,'reduce');  % filter and downsampling every 2nd
I0  = impyramid(I1,'expand');  % upsampling every 2nd
```

If other down- or up-sampling steps - than halving and doubling- are required, then the function `imresize` may be more convenient.

In Python we find those functions in the module `skimage.transform`, namely as `.pyramid_reduce` and `.pyramid_expand`. See also Appendix D for an overview.

Application In many matching tasks, it is more efficient to search for a pattern starting with the top level of the pyramid, the smallest resolution, and then to work downward toward the bottom levels, a strategy also called coarse-to-fine search (or matching). More specifically, only after a potential detection was made at the coarse level, then one starts to verify by moving towards finer levels, where the search is more time consuming due to the higher resolution.

Combinations If computation duration is not of concern and accuracy is desired, one may also use both, the scale space as well as the pyramid. For each level of the pyramid one would filter the map with two or three different values for sigma.

6.3 Gradient Image

wiki Image_gradient

The gradient image describes the steepness at each point in the intensity landscape, more specifically how the local 'surface' of the landscape is inclined. This is the basis for many implementations of the Local Feature approach (Sections 3.1 and 3.2.2).

At each pixel, two measures are determined: the direction of the slope, also called the gradient; and the magnitude - the steepness - of the slope. Thus the gradient image consists of two maps of values, the direction and the magnitude. That information is most conveniently illustrated with arrows, namely as a vector field, see Figure 17: the direction of the arrow corresponds to the gradient; the length of the arrow corresponds to the magnitude.

To determine the gradient, one takes the derivative in both dimensions, that is the difference between neighboring pixels along both axes, $\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$ respectively. This operation is typically expressed with the nabla sign ∇, the gradient operator:

$$\nabla I = (\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y})^T. \tag{4}$$

whereby the gradient information is expressed as a vector. We give examples: a point in a plane has no inclination, hence no magnitude and an irrelevant direction - the gradient is zero; a point in a slope has a certain direction - an angle value out of a range of $[0, 2\pi]$ - and a certain magnitude representing the steepness. The direction is computed with the arctangent function using two arguments (atan2), returning a value $\in [-\pi, \pi]$ in most software implementations. The magnitude, $\| \nabla I \|$, is computed using the Pythagorean formula.

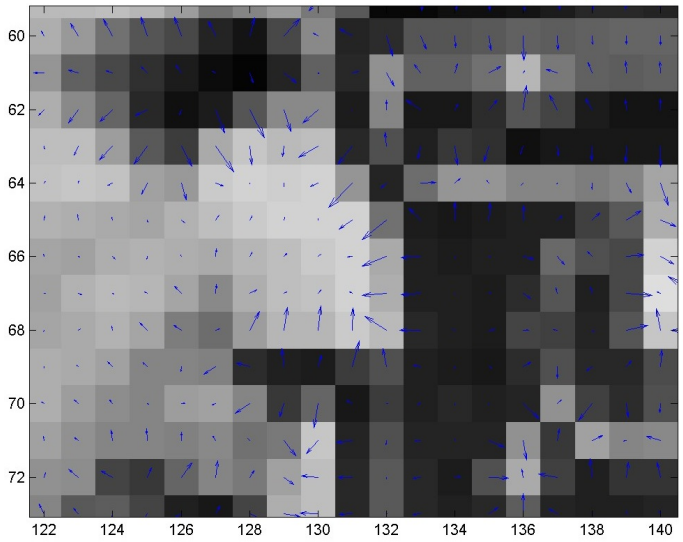

Figure 17: Gradient image. The arrows point into the direction of the gradient - the local slope of the neighborhood; the length of the arrows corresponds to the magnitude of the gradient. The gradient direction points toward high values (white=255; black=0).
(What does the picture depict? Hint: generate a coarse scale by squinting your eyes.)

Determining the gradient field directly on the original image does not return 'useful' results typically, because the original image is often too noisy (irregular). Hence the image is typically firstly low-pass filtered with a small Gaussian function, e.g. $\sigma = 1$, before determining its gradient field.

In Matlab there exists the routine `imgradient` which returns the magnitude and direction immediately (Image Processing Toolbox); if one needs also the derivatives later again in the code, then one can use `imgradientxy` to obtain two matrices corresponding to the individual gradients. If we lack the toolbox, we can use the following piece of code, whereby the function `gradient` returns two matrices - of the size of the image -, which represent the gradients along the two dimensions:

```
Fg          = fspecial('gaussian',[3 3],1); % 2D gaussian of size 3x3 with sigma=1
Isc1        = conv2(I, Fg, 'same');
[Dx Dy]     = gradient(single(Isc1));       % gradient along both dimensions
Dir         = atan2(-Dy, Dx) + pi;          % [-pi,pi]
bk0         = Dir<0;                        % negative values
Dir(bk0)    = Dir(bk0)+2*pi;                % [0,2*pi]
Mag         = sqrt(Dx.^2+Dy.^2);            % gradient magnitude
```

In the example code, the angle values are shifted into the positive range $[0, 2\pi]$. One can plot the gradient field using the function `quiver`:

```
% --- Plotting:
X = 1:ISize(2);  Y = 1:ISize(1);
figure(1); clf; colormap(gray);
imagesc(I, [0 255]); hold on;
quiver(X, Y, Dx, Dy);
```

In Python we can use the function `gradient` of the `numpy` module:

```
Dx, Dy = numpy.gradient(I)
```

Application The gradient image is used for a variety of tasks such as edge detection, feature detection and feature description (coming up). For tasks dealing with large amounts of images, i.e. huge databases or video processing, computing a gradient image is already a relatively expensive process.

7 Feature Extraction I: Regions, Edges & Texture

Now we make a first step toward reading out the structure in images, a first step toward Structural Description but also a first Local Feature type will be introduced. We will try to find edges in the intensity landscape, dots, regions, repeating elements called texture, etc. Some of this is also pure Filtering and therefore considered outdated, it is useful to understand as Deep Neural Networks might also focus on some of those aspects during its intensive learning process. Some of those features are also used for locating objects, i.e. finding their outlines.

Regions can be relatively easily obtained by subtracting the images of the scale space from each other. This is an operation called band-pass filtering and will be introduced in Section 7.1. Edges can be obtained by observing where a steep drop in the intensity landscape occurs, to be treated in Section 7.2. And for texture detection, the techniques are essentially a mixture of region and edge detection to be highlighted in Section 7.3.

7.1 Regions (Blobs, Dots) with Bandpass Filtering

Many regions are often either darker or brighter in comparison to their surround. One way to identify them would be by applying a threshold to the intensity values. That would be a technique that works well if the image consist of a homogenous background and foreground objects that stand out clearly from the background; we elaborate on that a bit later (Section 15). Here we use a technique that is slightly more flexible, but also more complicated. Specifically, we intend to exploit the scale space that we have developed previously. We subtract the images of the scale space S (Equation 3). That is for two images of S, one fine I_{fine} (e.g. $\sigma = 1$) and one coarse I_{coarse} (e.g. $\sigma = 2$), we take their difference

$$I_{\text{band}} = I_{\text{fine}} - I_{\text{coarse}} \tag{5}$$

in which case positive values correspond to brighter regions, and negative values correspond to darker regions. Figure 18 shows an example where those differences are taken between levels of the scale space.

As the fine and coarse image correspond to low-pass filtered images, their subtraction corresponds to the process of *band-pass filtering*. During band-pass filtering a certain center range ('band') of values is selected, as opposed to the low-pass filtering, for which a lower range of values is selected. See also Appendix C for more explanations.

If the low-pass filtering occurred with Gaussian functions, that is g in Equation 2 is a Gaussian filter, then the corresponding band-pass filter is a difference-of-Gaussian filter, a DOG filter. It is perhaps the most common band-pass filter in image processing.

In Matlab we can generate this 'band space' by applying merely the difference operator to the scale space `SS`, in which case however the signs for brighter and darker regions are switched due to the implementation of `diff`:

```
%% =========    DOG     ==========
DOG         = diff(SS,1,3);     % brighter is negative, darker is positive
% --- set borders to 0:
for i=1:size(DOG,3), p=i*3;
    DOG(:,:,i) = padarray(DOG(p:end-p+1,p:end-p+1,i),[p p]-1);
end
%% =========    BRIGHTER/DARKER   ========
[BGT DRK]  = deal(DOG);    clear DOG; % clear DOG to save memory
BGT(BGT>0)  = 0;                % all positive values to 0
DRK(DRK<0) = 0;                % all negative values to 0
BGT        = -BGT;             % switch sign to make brighter positive
```

The band-pass space `DOG` contains one level less than the scale space. In this piece of code we also set the border values to zero for reason of simplicity. The variables `BGT` and `DRK` hold then only positive and negative values (left and right column in Figure 18). Of course one could also take other threshold values than zero to select perhaps only very bright or very dark regions.

In Python we can utilize submodule `skimage.feature`, which offers the function `blob_dog`.

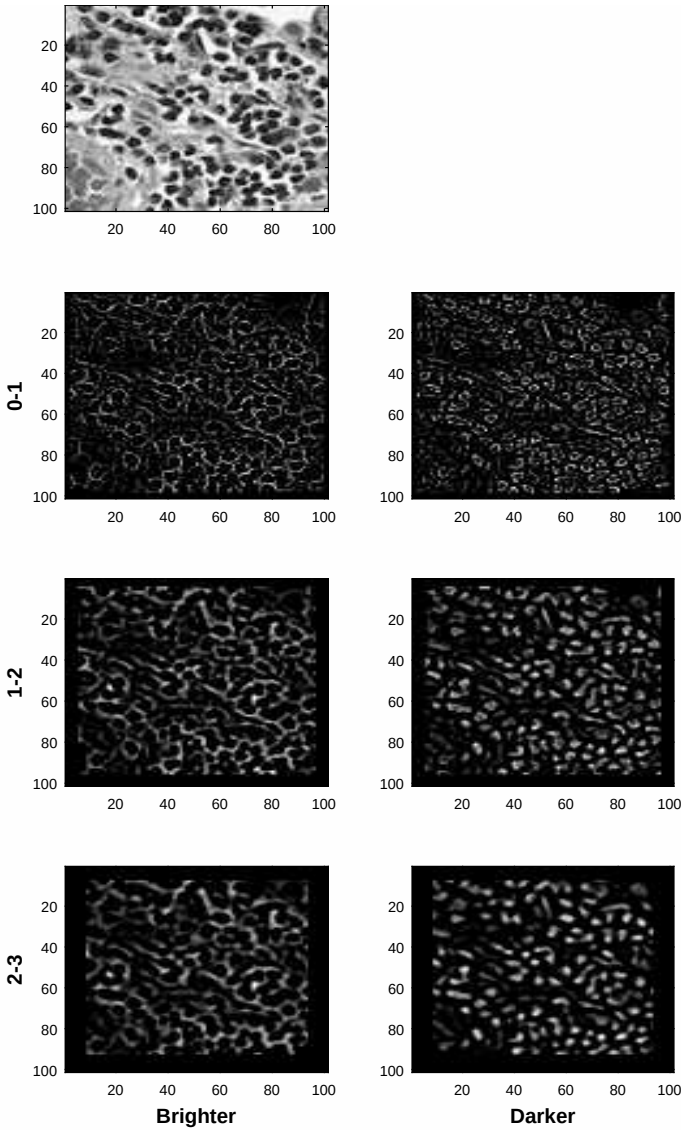

Figure 18: Example of a band-pass space, in this case a Difference-of-Gaussian (DOG) space: it is suitable for detecting regions (example code in **P.4.1**).
Left column represents brighter regions (`BGT` in code): negative values were set to zero.
Right column represents darker regions (`DRK` in code): positive values were set to zero and the absolute value of the negative values is displayed.

The second row labeled **0-1** is the subtraction of the input image and its low-pass filtered version. The third row labeled **1-2** is the subtraction of the corresponding two adjacent levels in the scale space; etc.

If one intended to select only the nuclei in this image of cell tissue, then the lower right image (2-3, darker) might be a good starting point and one would continue with morphological processing as will be presented in Section 15.

Now that we have regions, we are interested in manipulating them. For example we wish to eliminate small regions and would like to know the approximate shape of the remaining larger regions. We continue with that in Section 15.

Generating and dealing with the entire band-pass space is time consuming. If we develop a very specific task it might suffice to chose a few 'dedicated' scales. For instance to select nuclei from histological images that have image sizes of several thousands pixel (for either row or column), then generating the entire space might be too expensive. Rather one would design a bandpass filter whose size matches approximately the size of a typical nucleus. The original image would then be low-pass filtered with the corresponding two suitable sigmas, the resulting two filtered images subtracted from each other, and then a threshold is applied to the difference image.

7.2 Edge Detection

An edge is a steep drop in the intensity image, see again Figure 12 where we can clearly recognize huge cliffs. More precisely, an edge is an abrupt change in contrast with a certain orientation or direction; it is observed locally only, that is in a neighborhood of limited size. One could simply exploit the gradient image, see again 17, and apply a threshold to select the large gradients - the large arrows. Indeed, that represents

a first processing step in more sophisticated edge detectors. But thresholding the gradient image does not always find the precise location of the edge, as the edge could be very large and that would result in the detection of neighboring pixels as well.

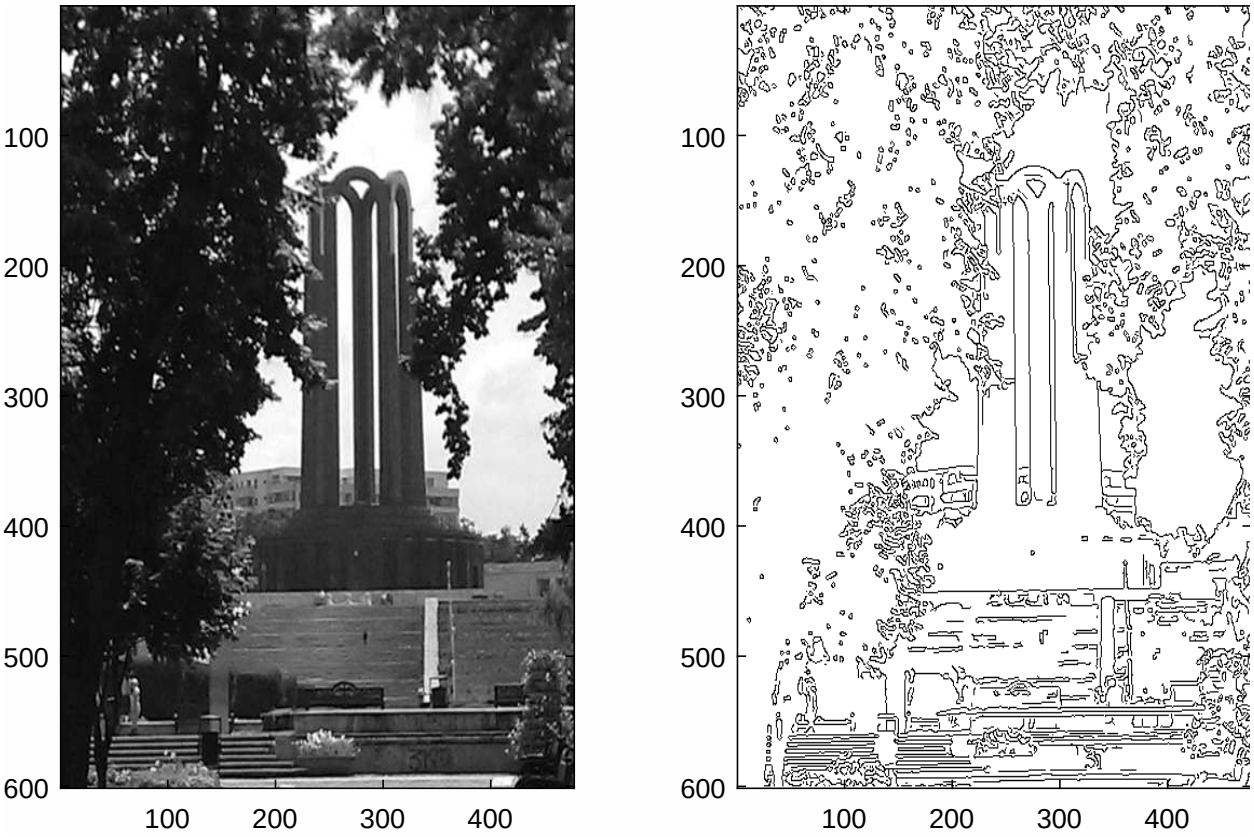

Figure 19: **Left**: input image. **Right**: binary map with ON pixels representing edges as detected by an edge-detection algorithm. There are different algorithms that can do that, the most elegant one is the Canny algorithm which is based on the gradient image as introduced in Section 6. An edge following algorithm traces the contours in such a binary map.

To find edges more precisely, the principal technique is to convolve the image with two-dimensional filters that emphasize such edges in the intensity landscape. The filter masks of such oriented filters therefore exhibit an 'orientation' in their spatial alignment. Here are two primitive examples for a vertical and a diagonal orientation filter respectively.

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}. \tag{6}$$

Thus, the image is convolved multiple times with different orientation masks to detect all edge orientations, resulting in a corrsponding number of different output maps. Those maps are then combined to a single output map by a mere logical AND operation. More sophisticated techniques interpolate between those principal orientations.

Matlab provides the function `edge` to perform this process and it provides different techniques. Python offers pretty much the same set of techniques, but in different modules, `skimage.feature` and `skimage.filters`. Those functions return as output a binary map where ON pixels correspond to locations of pixels. The code example in Appendix **P.4.2** displays the output of different edge detection techniques.

- Robert detector: this detector is rather primitive and is outdated by now, but it is still used in some industrial applications: its advantage is its low processing duration, its downside is that it does not

detect all edges.
- Prewitt and Sobel detector: those detectors find more edges, but at the price of more computation.
- Canny detector: this is most elaborate detection technique.

```
Medg = edge(I, 'canny', [], 1);
```

whereby the empty brackets [] stands for no particular choice of threshold parameters, and thus the threshold is determined automatically; the value 1 is the scale value. Typically, scale values up to 5 are specified. Hence, with this technique one can easily perform low-pass-filtering and edge detection using one function call only.

In Matlab there is a single function script for all techniques, called `edge`.

In Python the methods are found in different modules: the Canny algorithm in `skimage.feature.canny`; other techniques in `skimage.filters.roberts/sobel/prewitt`

In OpenCV [py] they are found in the main module, e.g. `cv2.Canny`.

A code example is given in **P.4.2**.

7.3 Texture

Dav p209, ch8

FoPo p194, pdf 164

Texture is observed in the structural patterns of object surfaces such as wood, grain, sand, grass, and cloth. But even scenes, that consist of many objects, can be regarded as texture.

The term texture generally refers to the repetition of basic texture elements called *textons* (or *texels* in older literature). Natural textures are generally random, whereas artificial textures are often deterministic or periodic. Texture may be coarse, fine, smooth, granulated, rippled, regular, irregular, or linear.

One can divide texture-analysis methods into four broad categories:

Statistical: these methods are based on describing the statistics of individual pixel intensity values. We only marginally mention these methods (Section 7.3.1), as they have been outperformed by other methods.

Structural: in structural analysis, primitives are identified first, such as circles or squares, which then are grouped into more 'global' symmetries (see SHB pch 15 for discussion). We omit that approach in favor of better performing approaches.

Spectral: in those methods, the image is firstly filtered with a variety of filters such as blob and orientation filters, followed by a statistical description of the filtered output. Caution: these methods are sometimes also called 'structural'.

Local Binary Patterns: turns statistical information into a code. It is perhaps the most successful texture description to date (Section 7.3.3). Fore some databases - in particular those containing textures - the description performs almost as well as Deep Nets. In comparison to Deep Nets, it has a smaller representation and a negligible learning duration.

7.3.1 Statistical

ThKo p412

One can distinguish between first-order statistics and second-order statistics. First-order statistics take merely measures from the distribution of gray-scale values. In second-order statistics, one attempts to express also spatial relationships between pixel values and coordinates.

First-Order Statistics Let v be the random variable representing the gray levels in the region of interest. The first-order histogram $P(v)$ is defined as

$$P(v) = \frac{n_v}{n_{tot}} \tag{7}$$

with n_v the number of pixels with gray-level v and n_{tot} the total number of pixels in the region (`imhist` in Matlab for an entire image). Based on the histogram (equ. 7), quantities such as moments, entropy, etc. are defined. Matlab: `imhist, rangefilt, stdfilt, entropyfilt`

ThKo p414
Dav p213, s8.3
SHB p723, s15.1.2

Second-Order Statistics The features resulting from the first-order statistics provide information related to the gray-level distribution of the image, but they do not give any information about the relative positions of the various gray levels within the image. Second-order methods (cor)relate these values. There exist different schemes to do that, the most common one is the gray-level cooccurrence matrix , see Dav p213, s8.3 SHB p723, s15.1.2 for more information. In Matlab the features can be generated with the function `graycomatrix`.

The use of the cooccurence matrix is memory intensive but useful for categories with low intensity variability and limited structural variability - or textural variability in this case. For 'larger' applications, the use of a spectral approach may return better performance.

7.3.2 Spectral [Structural]

FoPo p202
Pnc p277, s.13.1.5

In the spectral approach the texture is analyzed at different scales and described as if it represented a spectrum, hence the name. One possibility to perform such a systematic analysis is the use of wavelets (wiki Wavelet), which has found great use in image compression with the jpeg format. There are two reasons why wavelets are not optimal for texture representations. First, they are based on a so-called mother wavelet only, that is, they are based on a single filter function. Second, they are useful for compression, but less so for abstraction. To represent texture, it is therefore more meaningful to generate more complex filters.

In the following we introduce a bank of filters that has been successful for texture description, see Figure 20, namely the one by authors Leung and Malik (see Appendix **P.4.3** for code). Other spectral filter banks look very similar. All these filters are essentially a mixture of the filtering processes we have introduced before. There are four principally different types of filters in that bank:

Filter no. 1-4: those filters are merely Gaussian functions for different sigmas as we used them to generate the scale space (Section 6.1).

Filter no. 5-12: those eight filters are bandpass filters as we mentioned them in Section 7.1. In this case the blob filter is much larger and is a Laplacian-of-Gaussian (LoG) filter.

Filter no. 13-30 (rows 3, 4 and 5): these are oriented filters that respond well to a step or edge in an image - it corresponds to edge detection as introduced above in Section 7.2. In this case, the first derivative of the Gaussian function is used.

Filter n. 31-48 (bottom three rows): those filters correspond to a bar filter. It is generated with the second derivative of the Gaussian.

To apply this filter bank, an image is convolved with each filter separately, each one returning a corresponding response map. In order to detect textons in this large output, one applies a quantization scheme as discussed in Section 13. Algorithm 1 is a summary of the filtering procedure.

1. In a first step, each filter is applied to obtain a response map R_i $(i = 1..n)$. R_i can have positive and negative values.
2. In order to find all extrema, we rectify the maps obtaining so $2n$ maps R_j $(j = 1..2n)$.
3. We try to find the 'hot spots' (maxima) in the image, that potentially correspond to a texton, by aggregating large responses. We can do this for example by convolving R_j with a larger Gaussian; or by applying a max operation to local neighborhoods.
4. Finally, we locate the maxima using an 'inhibiton-of-return' mechanism that starts by selecting the global maximum followed by selecting (the remaining) local maxima, whereby we ensure that we do not return to the previous maxima, by inhibiting (suppressing) the neighborhood of the detected maximum.

Thus, for each image we find a number of 'hot spots', each one described by a vector $\mathbf{x}_l$. With those we then build a dictionary as in the quantization of Local Features (Section 13).

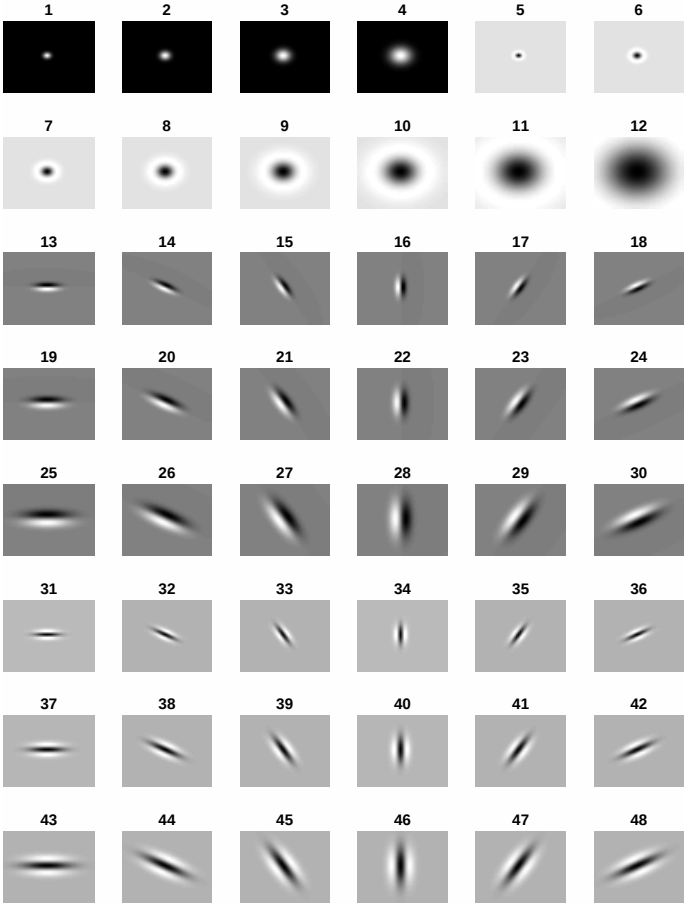

Figure 20: Filtering with a bank of filters for texture analysis (the Leung-Malik filter bank): blob and orientation filters for different scales (see Appendix **P.4.3** for code). Filter patches are generated for a size of 49 x 49 pixels.
Filters 1-4: Gaussian function (low-pass filter) at four different scales (sigmas).
Filters 5-12: a blob filter - a LoG in this case - at eight different scales.
Filters 13-30 (rows 3, 4 and 5): an edge filter - first derivative of the Gaussian in this case - for six different orientations and three different scales.
Filter 31-48 (bottom three rows): a bar filter - generated analogously to the edge filter.

Algorithm 1 Local filtering for texture.

Input : image I, set of n filters F_i (see Figure 20) ($i = 1..10$)
Parameters : radius for suppression in maximum search
1) Apply each filter F_i to the image to obtain a response map $R_i = I * F_i$
2) Rectification: for each R_i compute $\max(0, R_i)$ and $\min(0, -R_i)$, resulting in $2n$ maps R_j ($j = 1..2n$)
3) For each R_j, compute local summaries R_j^s by either
    a) convolving with a Gaussian of scale approximately twice the scale of the base filter
    b) taking the maximum value over a certain radius r.
4) Locate the maxima (in each map?) using an inhibition-of-return mechanism:$l = 1..n_{max}$ with coordinates $[x_l, y_l]$
Output : set of 'hot spot' vectors $\mathbf{x}_l(j)$ whose components j correspond to the filter response R_j at location (x_l, y_l)

7.3.3 Local Binary Patterns

wiki Local_binary_patterns

This texture descriptor observes the neibhouring pixels, converts them into a binary code and that code is transformed into a decimal number for further manipulation. Let us take the following 3x3 neighborhood. The center pixel - with value equal 5 here - is taken as a reference and compared to its eight neighboring pixels:

$$\begin{bmatrix} 3 & 4 & 6 \\ 1 & \mathbf{5} & 4 \\ 7 & 6 & 2 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 0 & 0 & 1 \\ 0 & - & 0 \\ 1 & 1 & 0 \end{bmatrix} \tag{8}$$

Neighboring values that are larger than 5, are set to 1, neighboring values that are smaller, are set to 0. This results in a 8-bit code, which in turn can be converted into a decimal code. This descriptor is constructed for each pixel of some window, sometimes even for the entire image. The descriptor codes are then histogrammed: in case of the 8-bit code turned into a decimal code, that would result in a 256-dimensional vector. That vector can be taken already as an image vector (Section 3.3). There exist many variations of this texture descriptor.

Matlab `extractLBPFeatures`
Python `skimage.feature.local_binary_pattern`

This descriptor is sometimes also deployed in the Local Feature approach (Section 3.2.2; Sections 12 and 13).

7.4 Literature

The most recent review on texture appears to be the one by Cavalin and Oliveira. It also reviews approaches using Deep Neural Networks:

Cavalin, Paulo, and Luiz S. Oliveira, 2017. A review of texture classification methods and databases.

A more recent database is the one provided by Cimpoi et al. Their study also includes a thorough comparison between the classical feature extraction methods, in particular Spectral Filtering, and Deep Net approaches:

Cimpoi et al. 2015. Deep filter banks for texture recognition, description, and segmentation.
`http://www.robots.ox.ac.uk/~vgg/research/deeptex/`

8 Deep Neural Networks

Deep Neural Networks are elaborations of traditional Artificial Neural Networks (ANNs), a methodology that exists already since decades. For a primer on ANNs we refer to Appendix F. In this section we immediately start with Deep Neural Networks. Its most popular type for computer vision is the so-called Convolutional Neural Network (CNN), which we introduce in Section 8.1. For this, we will switch to the programming language Python, as it has become the dominant language to explore neural network architectures. The big tech companies provide libraries to run such networks. Those libraries use similar terminology, but have slightly different formalism to build and learn the networks. One term that is often used is *tensor*. Mathematically speaking, a tensor is a function manipulating vectorial functions. In the context of Deep Nets, a tensor is essentially an array representing a batch (subset) of images, which makes the tensor three-dimensional or four-dimensional in size:

$$\text{Tensor} = [\text{batch size} \times \text{\# chromatic channels} \times x \times y] \tag{9}$$

the first dimension represents the number of images in the batch; the second dimension the number of channels (three for a RGB image; one for a gray-scale image; etc.); the last two dimensions stand for the spatial dimensions x and y. A tensor can also be two- or even one-dimensional, in which case they are sometimes called simple tensors. Practically, we do not need to learn novel algebra with this term, it is merely a different label for arrays, see also https://pytorch.org/tutorials/beginner/pytorch_with_examples.html.

Some of the most popular Deep Net packages are:

PyTorch, https://pytorch.org/: This package is provided by Facebook. It is perhaps the most trending package and for that reason we use it in this book. In particular *transfer learning* can be conveniently done with those libraries, coming up in Section 8.2.

Tensorflow, https://www.tensorflow.org/: This package is provided by Google. It is considered somewhat intricate to understand and to apply, and that is why a simpler API was developed for it, called **Keras**.

Caffe, http://caffe.berkeleyvision.org/, https://github.com/BVLC: Provided by UC Berkely. It was one of the early packages made public and has lost in appeal after the emergence of PyTorch and Tensorflow, but is still a reference. Many implementations of tasks are based on it.

Darknet, https://pjreddie.com/darknet: A package suitable for embedded systems, because it has a smaller memory footprint. It runs a very efficient object detection system.

For other packages we refer to the wiki pages: wiki Comparison_of_deep_learning_software.

A fundamental downside of Deep Learning is that it takes very long to train a Deep Net. One solution to the problem is to apply massive computing power, available however mostly to big tech companies and universities. Another way to mitigate the downside is to exploit the so-called CUDA cores of the graphic cards in a PC. Software packages (PyTorch, Tensorflow) provide routines to exploit the available CUDA cores of a computer. A third way to counteract the problem is to use the trick of *transfer learning*. The latter two tricks will be addressed in particular in Section 8.2.

8.1 The Convolutional Neural Network (CNN)

wiki Convolutional_neural_network

A Convolutional Neural Network (CNN) can be regarded as an extension of a Multi-Layer-Perceptron (MLP), a classical neural network. A code example for a MLP is given in Appendix F.1 and should be studied if one is not familiar with Neural Networks. In a CNN, that MLP is extended by a feature extraction process that consist of the application of two operations: a convolution and a pooling operation, see also Figure 21. By use of the convolution operation, a CNN does make actual use of the two-dimensionality of images, unlike most ANNs (such as the MLP). That way, the network tries to find discriminative, local features, similar to the feature extraction process discussed in Section 7 (Section 7.3.2 in particular). The convolution here is however slightly different from the convolution as used in Section 6 (see again Appendix B.2). The convolution here is more individual and that makes the CNN so powerful.

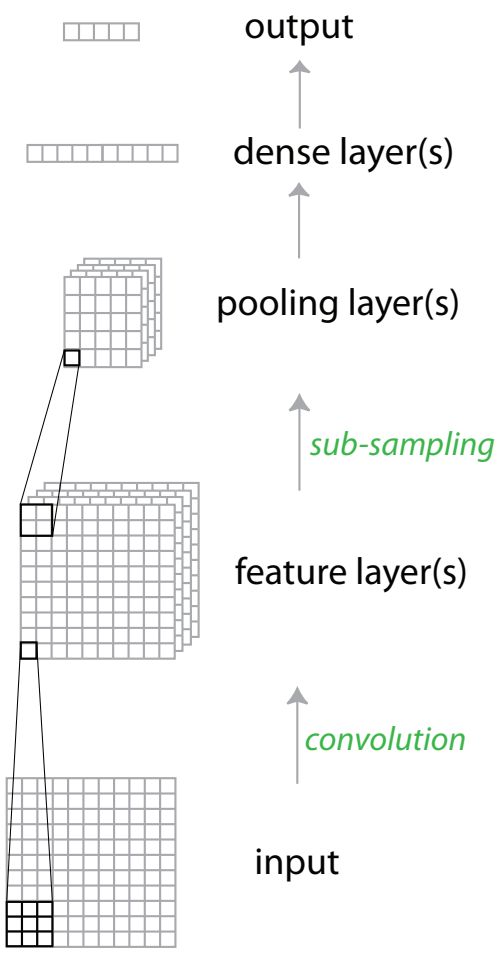

Figure 21: The typical (simplified) architecture of a Convolutional Neural Network (CNN) used for classifying images. It is essentially a MLP (Appendix F.1), elaborated by convolution and subsampling operations that can be regarded as preprocessing. The architecture has in particular several alternating feature and pooling layers. The term 'hidden' layer is not really used anymore in a CNN, as there are so many different types of hidden layers, that more specific names are needed.

The **feature layer(s)** is sometimes also called *feature map(s)*: it is the result of a convolution of the image, however not one with a single, fixed kernel, but one with many individual, learned kernels: each unit observes a local neighborhood in the input image and learns the appropriate weights.

The **pooling layer** is merely a lower resolution of the feature layer and is obtained by the process of sub-sampling. This sub-sampling helps to arrive at a more global 'percept'.

In a typical CNN, there are several alternations between feature and pooling layer. The learning process tries to find optimal convolution kernels K_i for each neighborhood i that help to separate the image classes.

The top two (final) layers - labeled 'dense' and 'output' - correspond to the scenario of image vector classification as introduced in Section 3.3.

Convolution In a 'traditional' convolution there is only a single kernel mask (K in equation 42). In the convolution operation of a CNN there are many individually learned kernel masks K_i, one for each local neighborhood i. And because a single, individual convolution per neighborhood is not enough for the discrimination of several categories, one therefore generates several layers of convolutions, that are also called *feature layers* or *feature maps*, analogous to using multiple different features (Section 7.3.2). In PyTorch the parameters for those convolution operations are given as follows:

```
torch.nn.Conv2D( # input maps, # output maps, kernel size, stride )
```

where we first specify the number of input and output maps, then the kernel size, followed by the parameter stride that specifies the step size of the convolution. The following gif files help understanding the use of that function:

https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

Sub-Sampling A CNN subsamples the feature layers to so-called *pooling layers* akin to building a spatial pyramid discussed in Section 6.1. This can be done in two ways: we either specify a stride in the convolution function (as just mentioned), or we use an explicit function; or one uses both options. A typical explicit function is called `.max_pool2d`,

```
torch.nn.functional.max_pool2d(I, kernel size, stride)
```

which takes the maximum value of its neighborhood and that value feeds into the next layer. It is more precise to use a small stride in the convolution function and then subsample explicitly, but that is also costlier as it requires more computation.

A typical CNN has many alternations between feature and pooling layers, or put differenty, it cycles through many such convolution and sub-sampling operations. For databases with small image size, a CNN may have only 2 such alternations, resulting in a 5-layer network or larger networks; for larger image sizes, the CNN can grow to several hundreds of layers. Between the last pooling layer and the output layer, there lies typically a dense layer: it is a flat (linear) layer, all-to-all connected with its previous pooling layer and its subsequent output layer, again as a complete bipartite graph.

Example To exemplify all that, we now build a small network for the MNIST database, a collection of hand-written digits, 0 to 9, with images of size 28 x 28 pixels. The full code is given in **P.5.1**, here we explain excerpts of it.

The network performs two cycles of convolution and subsampling, followed by two dense layers; the two dense layers are the equivalent of a MLP network as introduced in Appendix F.1. All these operations are firstly initialized in `__init__`, then one defines a function `forward` that carries out the forward pass during operation of the entire network; this specifies the exact flow of the entire network. The forward function is used for training and testing.

```
class NN_MLPconv(nn.Module):
    def __init__(self):
        super(NN_MLPconv, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1) # grayscale, out 20, szKrn, stride
        self.conv2 = nn.Conv2d(20, 50, 5, 1) # in 20, out 50, szKrn, stride
        self.fc1 = nn.Linear(4*4*50, 500)   # [(map size * #layers) 500]
        self.fc2 = nn.Linear(500, 10)       # [500 nClasses]

    def forward(self, I):           # [28 28]   image size
        I = F.relu(self.conv1(I))   # [24 24]   map size after 1st convolution
        I = F.max_pool2d(I, 2, 2)   # [12 12]   map size after 1st pooling
        I = F.relu(self.conv2(I))   # [8 8]     map size after 2nd convolution
        I = F.max_pool2d(I, 2, 2)   # [4 4]     map size after 2nd pooling
        I = I.view(-1, 4*4*50)      # reshape to linear for dense layers
        I = F.relu(self.fc1(I))
        I = self.fc2(I)
        return F.log_softmax(I, dim=1)
```

Since the MNIST database contains only gray-scale images - one color channel - , the very first convolution operation therefore takes the value equal one as its first argument; it would be three for a color image. The second argument is set to 20 maps. A kernel size of 5x5 is set and a stride of 1. The second convolution operation takes the 20 maps generated from the first convolution and generates another 50 maps, again with kernel size 5 and a stride with 1. Then the output of the second convolution is fed into a first linear dense layer, followed by a second even denser layer.

To understand the input size to the first dense layer, we need to understand how exactly the convolutions reduce the image, which is best done by understanding the forward flow. After two alternations of convolution and pooling, the map size has been reduced to only 4x4 pixels. The first dense layer therefore takes as input 4*4*50 parameters. The size of the second dense layer is set to 500. Those architecture parameters are found heuristically.

The rest of the code is as in Appendix F.1.

8.2 Transfer Learning (PyTorch)

The above example works well for small image sizes, but if we intend to classify larger images, then that increases learning duration significantly, because we convolve larger arrays. To train a network, it can take weeks and that is also called learning from *scratch*. Fortunately, there exists a trick called *transfer learning*, which allows us to re-use a from-scratch trained network for other tasks. That seems to work because the lower layers of a from-scratch trained network appear to be general feature extractors. In transfer learning, one takes such a network, cuts off its top weight layer, and replaces it with a weight layer for the new tasks that one intends to solve. The from-scratch trained networks are then called *pre-trained* networks in this

context. We introduce those pretrained networks in the subsequent Section 8.3; here, we merely learn how to apply them.

There are two modes of exploiting a pre-trained network, one as fixed feature extractor, and one as fine-tuning its weights to the specific task at hand.

Fixed Feature Extraction here we merely exchange the top weight layer and re-train only that top-layer with our images at hand. This is called fixed feature extraction, because the rest of the network remains untouched - we use it as is, namely fixed. That would be the quickest and simplest way of doing transfer learning and will be introduced in Section 8.2.1.

Fine-Tuning here we go a step further by re-adjusting the whole network, in addition to retraining the top weight layer. This takes more time, but we gain a bit of prediction accuracy. This mode requires in fact less code modifications to the network than fixed feature extraction (Section 8.2.2).

In Appendix **P.5.2** we give an example for both modes. One can switch between the modes using the boolean variable `bFine`. We now explain excerpts of that code.

One specifies a desired pre-trained model with a single line, in this case we select the 'resnet18' model:

```
from torchvision import models
MOD         = models.resnet18(pretrained=True)
```

Now we perform *data augmentation*, a process to artificially enrich the data set, see also Appendix J. This is carried out with the module `transforms`. The various types of transforms can be lumped together to a single object, called `TrfImgTrain` in the following code snippet:

```
from torchvision import transforms
TrfImgTrain   = transforms.Compose([
        transforms.RandomResizedCrop(szImgTarg),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(ImgMean, ImgStdv)  ])
```

Furthermore, PyTorch provides a routine `ImageFolder` in module `torchvision.datasets` to extract images and class labels from your collected data set automatically. You need to pass only the path of the folder that contains your classes, labeled accordingly, ie. all dog images placed into a folder called 'dog'. The images can be of varying size. A routine called `DataLoader` in module `torch.utils.data` will automatically prepare the data set.

```
FOLDtrain   = ImageFolder(dirImgs+'train', TrfImgTrain)
LOADtrain   = DataLoader(FOLDtrain, batch_size=szBtch, shuffle=True)
```

8.2.1 Fixed Feature Extraction

As introduced already, the fastest way to arrive at some results is to use the pre-trained network as is - without making further adjustments to it. To do that we proceed with two steps. One is to freeze the layers by preventing it to calculate gradients during the learning process: `requires_grad = False`. The second step is to replace the top layer with our problem set, for instance for a 5-class problem we need to insert a final weight layer that connects from the last dense layer to the final 5-unit output layer; function `nn.Linear` does that (`nClss` is the number of classes):

```
# freezing gradient calculation
for param in MOD.parameters():
    param.requires_grad = False
# replaces last fully-connected layer with new random weights:
MOD.fc  = nn.Linear(MOD.fc.in_features, nClss)
```

Now we initialize the training parameters. With the function `SGD` we select an optimization procedure called *stochastic gradients*. Note that we specify the training only for the final layer by selecting `.fc`.

By calling `lr_scheduler.StepLR` we specify more learning parameters. Then we choose a loss function, a function that calculates the classification accuracy.

```
# only parameters of final layer are being optimized
optim   = SGD(MOD.fc.parameters(), lr=lernRate, momentum=momFact)

# Decrease learning rate every szStep epochs by a factor of gamma
sched   = lr_scheduler.StepLR(optim, step_size=szStep, gamma=gam)

crit    = nn.CrossEntropyLoss()
```

8.2.2 Fine Tuning

To adjust the weights of the entire networks, we modify the steps of the previous section in two ways. One is, we refrain from freezing the parameters, so we drop the `for param` loop. Another step, is to omit the attribute `.fc`, which in turn instructs that the weights of the entire net are being re-learned:

```
# .fc is omitted - as opposed to above:
optim   = SGD(MOD.parameters(), lr=lernRate, momentum=momFact)
```

Adjusting the parameters of the entire net will take more time, but we obtain a slightly higher prediction accuracy. Other than that we can use the same code example as in **P.5.2**.

8.3 Network Architectures

There exists a number of popular network architectures, such as ResNet, DenseNet, Inception, etc. In PyTorch they are provided with module `torchvision.models`:

`https://pytorch.org/docs/stable/torchvision/models.html`

Most of them come with varying number of layers, for example network type 'ResNet' comes with 18, 34, 50, 101, etc. layers. Few layers means the network is smaller and learns faster; many layers means the network is larger and learns slower, but also shows a better classification accuracy. Hence, there is a tradeoff between size and performance. A large architecture has also the disadvantage that the processing duration for classifying an image is larger than that for a smaller architecture; this duration is also called *inference time*. This is of concern for real-time applications - e.g. autonomous driving -, but also for classifying vast amounts images - e.g. the image material on the Internet.

To compare the performance between network architectures, a network is trained on the ImageNet dataset, `http://image-net.org/`, an image set with 1000 categories such as dogs, cars, lamps, flowers, etc. The performance for those networks is listed on the above mentioned site, and it can be seen that many approximately perform similar if compared across their degree of detail (number of layers). The weights of those networks - trained on ImageNet - are the weights used when doing transfer learning.

We summarize the main network architectures:

ResNet (Deep Residual Learning for Image Recognition): by Microsoft. Comes with 18, 34, 50, 101 and 152 layers, ranging from 46 to 236 MB of weights. They show the best prediction accuracies on the testing sets.

DenseNet (Densely Connected Convolutional Networks): by Cornell University and FaceBook. Comes with 121, 169, 201 and 264 layers, with 32 to 114 MB of weights. They show competitive (near best) prediction accuracies on the training sets, but their weight set is smaller - about half of that for ResNet at approximately the same prediction accuracy.

InceptionV3 mainly by Google. Comes as a single instantiation with 107 MB of weights. Shows competitive performance. Appears to use the fewest weights.

EfficientNet this is a novel network that is smaller than the other networks and nevertheless shows a slightly better performance. Google has already implemented and applied it with great success. I suspect it will soon appear in a new PyTorch version.

8.4 Ensembles

A relatively easy way to improve the classification accuracy using Deep Nets is to combine them. The combination is carried out by a mere mathematical operation or a statistical function applied to their output. The combination can be understood as the pooling of expert opinions. The process of combining decisions from different implementations is also called *ensembles*. This increases the overall complexity of the system of course, because we now train several Deep Nets and not only one for solving a given task.

Combining Outputs The simplest combination is to use histogramming of the labeled output. Assume we have trained a task with a ResNet, a DenseNet, an EfficientNet and the Inception network. Then for each classificed image, we observe to which category it has been assigned to most frequently, or in short we histogram the labeled output of the four networks. Histogramming labels has the down side we face sometimes par situations, when there exists no majority.

To avoid par situations, we could simply work with the graded outputs of the networks. In the example of the MLP network (Section **P.5.1**), the graded values are in `Out.data`, in the example of transfer learning (Section **P.5.2**), the graded values are in variable `Post`. Those graded values are sometimes also called posteriors. For an image with four network predictions, we would take the label for which the posterior label is largest across all networks. And there are other ways to combine the posterior values, such as using the mean operation on the four posterior values across the networks, or the sum of them, etc.

The intriguing issue with ensemble decisions is that they tend to work better when a network is trained only partially: it is not trained to its fullest extent possible, but sub-optimal using fewer epochs for instance. This leads to the situation where one prefers networks with fewer layers, sometimes trained only partially. It is not unusual to take a single network architecture, e.g. ResNet50, and train it several tens of times individually with few epochs only and then combine the individual decisions.

8.5 Summary and Notes

Deep Neural Networks are the most successful classifiers to date - they are able to discriminate any small set of classes decently. The Convolutional NN is the most popular classifier, but also the Deep Belief Network is used occasionally, mentioned in Section F.2. There exists a size-performance tradeoff for networks and in some situations, i.e. embedded systems or cell phones, one may be limited too smaller and less accurate networks. Ensembles of networks often outperform individual networks, but also increase the complexity substantially.

Although Deep Nets are a relatively complex method, they can process easily hundreds or thousands of images in reasonable time, thanks to the effort to make them run on particular processing units such as CUDA cores wiki CUDA. They merely consume a lot of power (but therefore marginally contribute to the increasing global temperatures).

DNNs have several disadvantages. One is that the design of the appropriate architecture is a bit of a heuristic endeavor, as well as the tuning of the learning parameters; the Belief Network is in that aspect more elegant. Another reason is that learning itself is a very slow process and one therefore employs transfer learning to speed up the training process. Another disadvantage is that they use a lot of memory for the millions of weight parameters and therefore consume a lot of power. There exist Deep Networks that are much more efficient with regard to those latter aspects, but they also show somewhat lower prediction accuracy compared to the full networks, see for instance `https://www.xnor.ai/technology/`. Those networks might be the appropriate choice for embedded systems or cell phones.

An issue we have not really addressed is how to properly analyze the classification outcome and to how to properly estimate the prediction accuracy. We refer for that to Appendix I.1.1. This was not of such a concern so far, because we have introduced systems that discriminate multiple classes and dealt with image sets that have separate training and testing sets. If however we test novel data sets, we need to ensure we provide a proper classification estimate.

8.5.1 Resources

For understanding the convolution operations in more detail the following links certainly help:
https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md: animated gif files
https://arxiv.org/abs/1603.07285: a paper explaining the operations.

Companies offering dedicated hardware:
https://www.aime.info/ https://www.xnor.ai/technology/

9 Segmentation (Supervised)

Segmentation is the partitioning of a scene into its components. For example we segment a street scene into its objects and parts, such as cars, buildings, pedestrians, walk-way, etc. Or we segment the image of a melanoma into two regions, the healthy skin and the lesion itself, the affected skin. Segmentation can be carried out with or without supervision:

Supervised: in this format, we learn what types of objects to segment using training images. The training images come as pairs of images, see Fig. 22. This learning process is therefore analogous to the classification task as introduced in the previous section and for that reason we call this *supervised*, a term that stems from the field of Machine Learning (see also Appendix J). This is the type of segmentation we address in this section. In a wider sense, one can also call this *top-down* segmentation, see again Fig. 4, because information is firstly integrated in a hierarchical manner toward the top. Then, in a second phase, it is unfolded back to the size of the image, toward the bottom. The output is a segmentation map.

Unsupervised: in this format, no or hardly any learning occurs - as opposed to supervised segmentation and for that reason it is also called *bottom-up* segmentation. The unsupervised approach will be introduced in Section 14.

The methodology for supervised segmentation introduced here is based on Deep Nets and is similar to the one for classification, meaning we deal with network architectures that use a lot of convolutional layers. The detailed differences are explained in Appendix F. The particular type of network that is used for segmentation is called an *autoencoder* (Appendix F.3). In order to train such an autocoder we need not only images, but also a so-called *segmentation mask* for each image. The mask is a map of the same size as its corresponding image and outlines the desired segmentation result by use of a numeric label. If the task is to locate a single object, then the mask is a binary map with 1s representing the target object, see Fig. 22; it is a black-white image as introduce in Section 4.1. We now first explain the purpose of the autoencoder (Section 9.1) and then introduce autoencoders for segmentation (Section 9.2).

9.1 Autoencoder - General

An autoencoder is a network designed to learn a code of its input by itself, without any labels. In its original form it is therefore an unsupervised learning algorithm that uses no masks; in the subsequent section we will exploit masks to use the autoencoder in a supervised manner. Irrespective of its use - unsupervised or supervised -, the autoencoder network consists of two phases, an encoding and a decoding phase:

The encoding phase represents a feature extraction process, whose output is here called a code.

The decoding phase is the inverse of the encoding phase, a feature 'generation' process; it unfolds from the code toward the pixel array, often with exactly the same architecture the encoding uses - applied inverse.

This autoencoder system is sometimes used to learn features of its input by itself - analogous to the feature learning process that happens in a neural network as introduced in previous sections, but in this case an explicit teacher is absent. In the example in Appendix F.3, we use the MNIST database to learn a code consisting of only three units, whose graded values therefore express the 10 digits. One could try to use those three units for classification, but learning a classifier directly is typically better at predicting class labels (i.e. F.1).

In this unsupervised form of the autoencoder, the criterion function uses the input image `Img` as a target, unlike in classification networks where the target is a class label:

```
Iout   = NET(Img)              # forward image through network
loss   = criterion(Iout, Img)  # compare autoencoder output with input image
```

where `Iout` is the output of the encoder (the variable names are slightly different in the full code example). In other words, the autoencoder compares its network output to its input - hence the term *auto*.

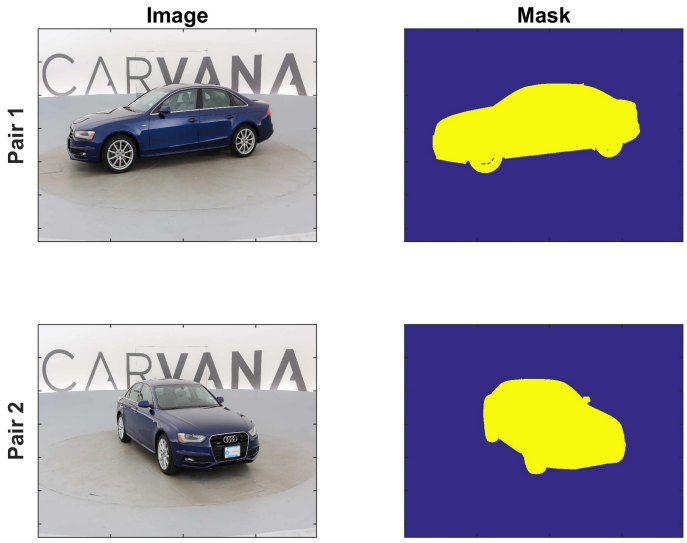

Figure 22: Training for supervised segmentation using pairs of images. Two pairs are shown, row-wise: the RGB image (left) and its corresponding mask (right) [From the Carvana competition on Kaggle]. The mask in this case is a binary image delineating the silhouette of the target object - a label matrix with 0s and 1s (also called black-white image with ON and OFF pixels). Training an autoencoder with such pairs will generate a segmentation system that is able to locate and segment cars in novel images with similar scene types.

The code example in F.3 does not involve any convolutions. One could include (two-dimensional) convolutions into that example to make it better performing, but we skip that in favor of moving directly to autoencoders for segmentation, coming up next.

9.2 Autoencoder for Segmentation

Now we return to the masks as introduced in Fig. 22. To exploit an autoencoder for a segmentation task, we compare the network's output, `Iout`, with the input's corresponding segmentation mask, `Msk`, for instance:

```
Iout    = NET(Img)
loss    = criterion(Iout, Msk)       # compare output with target
```

and thus educate the network to focus only on that labeled region. By using annotated masks, the process has turned into a supervised classification with labels, but one still speaks of autoencoders in this context.

Such networks can be configured to learn to segment only one label (object or scene part) or multiple labels (objects or scene parts) simultaneously. For multiple labels, we need to adapt the calculation of the loss function a bit and we therefore introduce those two modi in separate subsections (Sections 9.2.1 and 9.2.2, respectively).

In PyTorch (version 1.1) such autoencoder networks are offered in module `torchvision.models.segmentation`, see also:
`https://pytorch.org/docs/stable/torchvision/models.html#semantic-segmentation`.
Those networks are often based on one of the classification networks as introduced previously - the model names indicate already on what classification model they are based on. The different types of models come with different number of layers. And as with classification models, the larger the number of layers, the more accurate will be the segmentation, but the longer will also be the training duration, and the larger will be the weight set and the larger the inference duration. The idea of transfer learning is used in this process as well. Thus, from an application point of view, there is little novelty coming from the classification format.

9.2.1 Single Label

We take the task of car segmentation as an example, as displayed in Fig 22, available from
https://www.kaggle.com/c/carvana-image-masking-challenge
This specific task is also called a foreground-background segregation task, or a figure-ground segregation task (see also Section 14). The code example is given in **P.6.1**, we now mention a few key aspects.

As we now load not only an image, but also a corresponding mask, we need to instruct the dataset function accordingly. This is done with the function `f_DATtren`.

In function `BCELoss2d` we apply the loss function. This function compares the image pixel by pixel, that is, no particular use is made of the two-dimensionality of the input.

An autoencoder network is initalized - very similar to a classification network - by specifying the number of classes present, which in this case means the number of (unique) labels. For our object segmentation task, where only one object per image is present (Fig. 22), one therefore specifies `num_classes=1`. We also indicate whether transfer learning is desired by using the parameter name `pretrained`.

```
    NET = fcn_resnet50(num_classes=1, pretrained=False).to(device).
```

Both images and masks are provided as batches. The output is in this case a dictionary and the map estimates are accessed with the key `out`: `IOut['out']`.

```
    IOut    = NET(Imgs)                    # IOut is [szBtc 1 h w]
    loss    = crit(IOut['out'], Msks)      # compare output with target
```

The map for a single image shows graded values in the negative and positive domain and the map is then thresholded at a suitable value, e.g. at value equal zero or the mean intensity value. The resulting black-white image is then our predicted mask. If we wish to compare that to some annotated image, then there exists a variety of measures for which we refer to Appendix I.1.3.

9.2.2 Multiple Labels

Figure 23 shows an example for a task with multiple labels. A road scene is annotated with various class labels, e.g. car, pedestrian, vegetation, etc. Sometimes, this type of segmentation is also called *semantic* segmentation, because we deal with rich and individual scenes as opposed to the situation of discriminating merely between two classes such as in a melanoma task.

Each class label is assigned a positive integer. The mask is therefore not binary anymore - as in our previous example -, but is a so-called label matrix. The network should then predict where those classes occur in a scene.

In such a multi-label task, our input-output images change in type and number:

`Msks` is -as already mentioned - now a label matrix (Section 15.3) The label matrix for the Cityscape database holds values ranging from 0 to the number of labels, where a positive integer stands for one of the labeled classes. Other databases may use a slightly different labeling organization.

`Imaps = IOut['out']`: `Imaps` is now a list of images, one image for each label. The list is returned as a three-dimensional array for convenience, of shape [#labels height width]; and because we deal with batches the entire output is actually a four-dimensional array, of shape [batchSize #labels height width].

This means we need to modify our loss function. One could think of several modifications, a very simple one is to turn the label matrix into a multi-dimensional, binary array. As we deal with batches usually, we write two loops, one outer loop for the images in a batch, an inner loop for the masks of one image:

```
# ====  creating binary masks   =====
MskB    = torch.zeros((szBtc,nLab,sidImg,sidImg), dtype=torch.float)
for b in range(szBtc):   # --- loop batches
     Ib  = Msks[b,:,:,:].squeeze(0) # list of masks for this image
     for l in range(nLab):   # --- loop labels
```

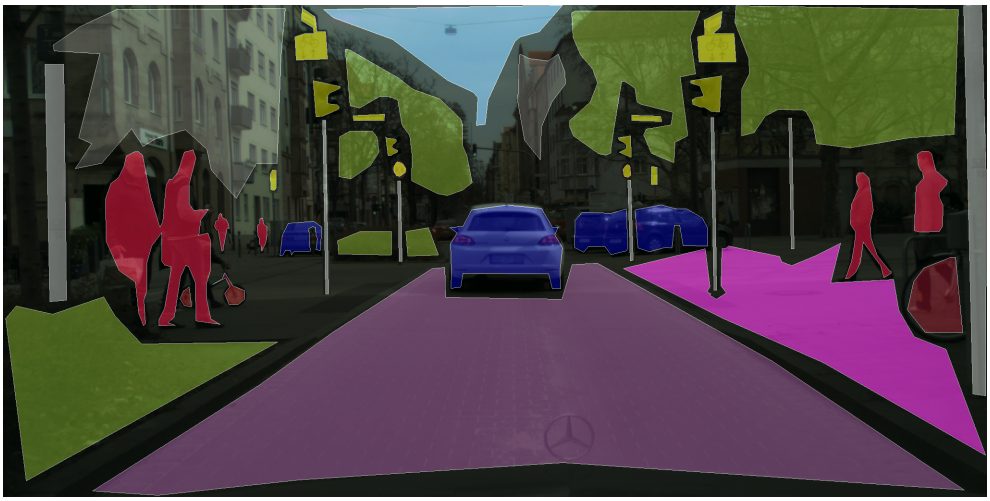

Figure 23: An example of the Cityscape dataset (Appendix M): the masks, a label matrix essentially, are overlayed on top of the gray-scale image. Examples of labels are road surface, car, pedestrians, vegetation, etc. A total of 34 labels exist for that database.

```
        Ip      = torch.zeros((sidImg,sidImg))
        BW      = Ib==l+1       # black-white with 1 where label present
        Ip[BW]  = 1             # place into image
        MskB[b,l,:,:] = Ip      # place into mask array
```

In our code example we take the Cityscapes database, see **P.6.2**. The database needs to be downloaded manually:

`https://www.cityscapes-dataset.com/`

Python provides functions to access the downloaded files and that is the reason why the code is now shorter than that for the single-label example.

9.2.3 Ensembles

As with classification ensembles (Section 8.4), we can typically improve segmentation by using ensembles of such networks. Here, the graded output is an entire map - and not only class scores -, but the principles of combining remain the same as for the classification models. The easiest way to combine two outputs, is to use the logical AND function on the thresholded maps. For example, if we trained two different networks, `NET1` and `NET2`,

```
Iout1   = NET1(Img.unsqueeze(0).to(device))  # add 4th dimension to make it a 4-dim tensor
Iseg1   = Iout1['out'].data[0].squeeze(0).cpu().numpy()
Iout2   = NET2(Img.unsqueeze(0).to(device))  # add 4th dimension to make it a 4-dim tensor
Iseg2   = Iout2['out'].data[0].squeeze(0).cpu().numpy()
Icomb   = np.logical_and(Iseg1>0, Iseg2>0)
```

then `Icomb` is the binary map that is the union of the two thresholded maps.

10 Object Detection

FoPo p549, ch 17

Object *detection* is the localization and count of a specific object category in a scene. For example, we determine how many faces there are in a group photo; or how many pedestrians there are in a street scene. It is a binary (two-class) classifier, that discriminates between target and non-target, or object versus non-object. The non-targets or non-objects are also called *distracters* sometimes. To construct such a system, one collects images of target objects, and images of non-target objects, and then trains a classifier to discriminate between the two target groups. The methodology for this binary classification task is essentially the same as for image classification; to find the object one applies the classifier to different locations in the image. The challenge is to find those potential target locations as efficiently as possible, a spatial search process that can be very time-consuming and that one attempts to reduce as much as possible.

Although such tasks are approached with Deep Nets nowadays, we here introduce systems that are particularly fast, whose detection speed is hard to beat by any Deep Net. Such systems consist of two separate processes, the spatial search process and the classification process. The search process browses the image for potential object candidates. The most accurate approach would be to scan the entire image pixel by pixel, but that is also the costliest - it can be considered greedy search using a term from the field of algorithm development. To speed up the search process we can scan the image step-wise along a grid, but that also deteriorates the classification accuracy, because we risk glancing over optimal positions (see again trade-offs as introduced in Fig. 10). One therefore deals with a speed-accuracy trade-off, whose terminology will be introduced in Section 10.2. Before introducing that terminology, we start with face detection, for which there exists a particularly efficient search algorithm (Section 10.1). Then we introduce a popular pedestrian detector (Section 10.3). Finally we mention some attempts to accelerate the search process by use of image preprocessing (Section 10.5).

10.1 Face Detection

Sze p578, s14.1.1, pdf658

FoPo p550, s17.1.1, pdf520

Face detection is ubiquitous nowadays: it is run in most of today's digital cameras to enhance auto-focus; on social-media sites to tag persons; in Google street view to blur persons, etc. There exist many algorithms each one with advantages and disadvantages. A good face detection system combines different algorithms, but most of them will run the Viola-Jones algorithm (Section 10.1.2). But first we mention general tricks used for training a face detection system.

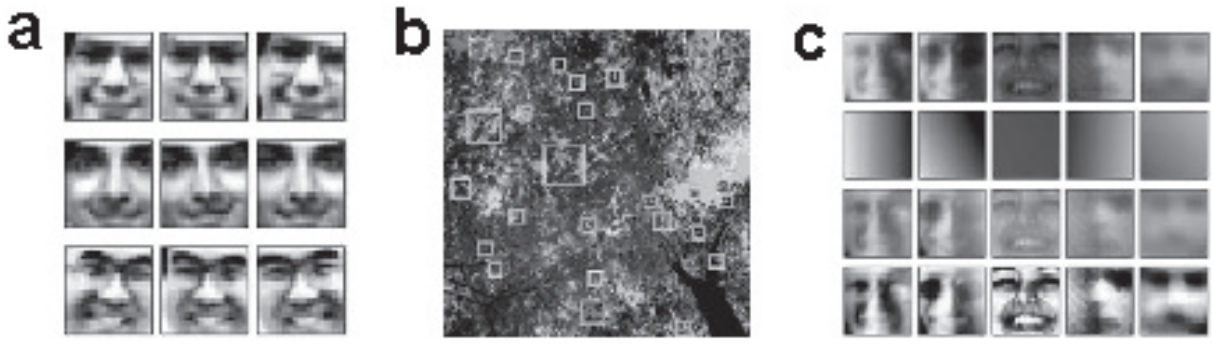

Figure 24: Training a face detector (Rowley, Baluja, and Kanade 1998a):
a) Data augmentation: artificially mirroring, rotating, scaling, and translating training images to generate a training set with including larger variability.
b) Hard negative mining: using images without faces (looking up at a tree) to generate non-face examples.
c) Image enhancment: pre-processing the patches by subtracting a best fit linear function (constant gradient) and histogram equalizing.
[Source: Szeliski 2011; Fig 14.3]

10.1.1 Optimizing Face/Non-Face Discrimination

A typical face detection system uses the following tricks to improve performance. For the first two tricks see also Appendix J.

a) Hard Negative Mining: non-face images are collected from aerial images or vegetation for instance (Figure 24b).

b) Data Augmentation: the set of collected face images is expanded artificially by mirroring, rotating, scaling, and translating the images by small amounts to make the face detectors less sensitive to such effects (Figure 24a).

c) Image Enhancement: after an initial set of training images has been collected, some optional preprocessing can be performed, such as subtracting an average gradient (linear function) from the image to compensate for global shading effects and using histogram equalization to compensate for varying camera contrast (Fig. 24c), see again Section 4.2.

10.1.2 Viola-Jones Algorithm

wiki ViolaJones_object_detection_framework

The most frequently used face detection algorithm is probably the one by Viola and Jones. It describes faces by combinations of brighter and darker rectangles of different size and orientation, see upper row of Fig. 25 for the most salient combinations. It suffices to use two to four rectangular patches of different polarity to find faces efficiently in an image; the pixels inside the white rectangles are subtracted from the pixels inside the black pixels.

Using such combinations of rectangles is a rather simplistic description of an object in comparison to the complex features introduced in previous sections, but it is one that has worked exceptionally well, in particular also because the intensity values of rectangles can be very efficiently computed with the so-called integral image (Section 10.1.3).

Figure 25: Face detection with groupings of rectangular patches. Top row: the 2 most significant rectangle-based combinations (in isolation). The horizontally oriented feature represents the eyes and the cheekbones; the vertically oriented ones represent the region covering left eye-nose bridge-right eye. [Source: Szeliski 2011; Fig 14.6]

To find out which combinations of rectangles are representative for a category, it is necessary to try out all combinations. This learning process can be done with a so-called *boosting* classifier. The classifier identifies which combinations are the most discriminative and then ranks them in decreasing order. When the classifier is applied to an image, it will firstly search for the most discriminative features, and if those are found, it will apply the less discriminative features to confirm that it has found a face. This is also called a *cascade* classifier.

This scheme of learning and detecting using rectangles is so efficient, that it has also been applied to face parts, such as eyes, nose and mouth part, but also to upper body (torso). The primary advantage of this object detection algorithm is that it is extremely fast and runs in real time. The downside of the system is that it detects preferentially vertically oriented faces, see Fig. 26 for an example.

In Matlab we can experiment with such a classifier using the package `vision`, that offers the function `CascadeObjectDetector`. With that function we can initialize a variety of instantiations of the mentioned classifier scheme, here shown only for face detection:

```
FaceDet = vision.CascadeObjectDetector('FrontalFaceCART');
```

We apply the face detector object by using its function `step`:

Figure 26: Face detection with the Viola-Jones algorithm. The algorithm finds many faces in almost no time, but tends to miss any faces that do not exhibit frontal view, with very low contrast, etc. Code examples in **P.7.1**. The yellow boxes are called the bounding boxes and outline the localization estimate by the detection algorithm.

```
BBox    = step(FaceDet,Irgb);  % bounding boxes [nBox 4]
```

and that returns a list of bounding boxes. A bounding box is the rectangle that encompasses its target object. It is typically specified by four values, its upper left point and a width and a height value. A full example is given in **P.7.1**.

In Python we can apply the algorithm through OpenCV, initializing it with function `CascadeClassifier`; here we take the default variant `haarcascade_frontalface_default.xml`. The classifier is applied with the method `detectMultiScale`:

```
FaceDet = cv2.CascadeClassifier(cascPth + 'haarcascade_frontalface_default.xml')
aFaces  = FaceDet.detectMultiScale(Igry, 1.3, 5)
```

A full code example is given in **P.7.1**.

Other Applications Face detectors are built into video conferencing systems to center on the speaker. They are also used in consumer-level photo organization packages, such as iPhoto, Picasa, and Windows Live Photo Gallery.

Further Leads

`http://www.ipol.im/pub/art/2014/104/article_lr.pdf`: a very explicit description of the algorithm.
`http://vis-www.cs.umass.edu/lfw/`, a database with 13k faces.
`https://facedetection.com/`
`https://www.mathworks.com/matlabcentral/fileexchange/39627-cascade-trainer-specify-ground-truth-train-a-detector`

10.1.3 Rectangles

Sze p106, pdf 120
Pnc p275
Dav p175
SHB p101, alg 4.2

Rectangular regions can be detected rapidly by use of the integral image, aka *summed area table*. It is computed as the running sum of all the pixel values from the origin:

$$I_s(i,j) = \sum_{k=0}^{i} \sum_{l=0}^{j} I_o(k,l). \tag{10}$$

To find now the summed area (integral) inside a rectangle $[i_0, i_1] \times [j_0, j_1]$, we simply combine four samples from the summed area table:

$$R_s(i_0..i_1, j_0..j_1) = I_s(i_1, j_1) - I_s(i_1, j_0) - I_s(i_0, j_1) + I_s(i_0, j_0) \tag{11}$$

Matlab

```
Is = cumsum(cumsum(Io,1),2);                % integral image (same size as original image)
Rs = Is(i1,j1)-Is(i1,j0)-Is(i0,j1)+Is(i0,j0); % summed intensity values for rectangular patch (scalar)
```

Python skimage.transform.integral_image

10.2 Sliding Window Technique

The sliding window is a technique for finding objects with high accuracy. It essentially corresponds to scanning the image with a window, a rectangular subset of the image, a large neighborhood in some sense. This window is moved across the image at selected column and row positions, a search called *sliding*. The selected column and row positions are typically taken from a grid.

After we have trained a classifier, we pass $n \times m$ windows of a new (testing) image to the classifier; the window is moved along the grid, by a step size of few pixels e.g. Δx and Δy=3 pixels. There are three challenges with this technique:

1) Size invariance: the detection system should be invariant to object size. This can be achieved by a search over scale, meaning by using the pyramid (Section 6.1): to find large objects, we search on coarser scales (layers), to find small objects we search on a finer scales. Put differently, we apply the $n \times m$ window in each layer of the pyramid.
2) Avoiding multiple counts: the very same object instance in an image should not be counted multiple times, which may happen due to the sliding search: the smaller the step sizes, the higher the chance for repeated detection. To avoid multiple counts, the neighboring windows are suppressed, when a local maximum was detected, also called non-maximum suppression (to be explained in more detail in Section 11.2).
3) Accelerating spatial search: searching for a match in the highest image resolution is time consuming and it is more efficient to search for a match in the top pyramidal layers first and then to verify on lower layers (finer scales), that means by working top-down through the pyramid, e.g. first P_3, then P_2, etc. This strategy is also known as coarse-to-fine matching.

The technique is summarized in Algorithm 2:

There are obviously trade-offs between search parameters (e.g. step sizes) and system performance (e.g. detection and localization accuracy). For example, if we work with training windows that tightly surround the object, then we might be able to improve object/distractor discrimination, but we will have to use smaller step sizes for an actual performance improvement. Vice versa, if we use windows that surround the object only loosely, then we can use smaller steps sizes but our discrimination and localization performance suffers.

In software packages, there are some functions to facilitate the search processes:

Algorithm 2 Sliding window technique for object detection.

=== TRAINING: Train a (binary) classifier on $n \times m$ image windows with positive (object) examples and windows with negative (non-object) examples.
=== TESTING:
Parameters detection threshold t, step sizes Δx and Δy
1) Construct an image pyramid.
2) For each level of the pyramid:
- apply the classifier to each $n \times m$ window (moving by Δx and Δy) and obtain strength c.
- if $c > t$, then add window to a list $\mathcal{L}$ including response value c.
3) Rank list $\mathcal{L}$ in decreasing order of c values $\rightarrow \mathcal{L}^{seq}$.
4) For each window $\mathcal{W}$ in sequence $\mathcal{L}^{seq}$ (starting with maximal c):
- remove all windows $\mathcal{U} \neq \mathcal{W}$ that overlap $\mathcal{W}$ significantly where the overlap is computed in the original image by expanding windows in coarser scales $\rightarrow \mathcal{L}^{red}$.
$\mathcal{L}^{red}$ is the list of detected objects.

In Matlab this can be found in particular under 'Neighborhood and Block Operations' in the image processing toolbox. In particular function `blockproc` and `nlfilter` are useful here.

In Python one would have to look into module `numpy.lib.stride_tricks`, but that does not appear to be a much pursued direction.

10.3 Pedestrian Detection

According to Dalal and Triggs, one can typify the structure of pedestrians into 'standing' and 'walking':
- standing pedestrians look like lollipops (wider upper body and narrower legs).
- walking pedestrians have a quite characteristic scissors appearance.

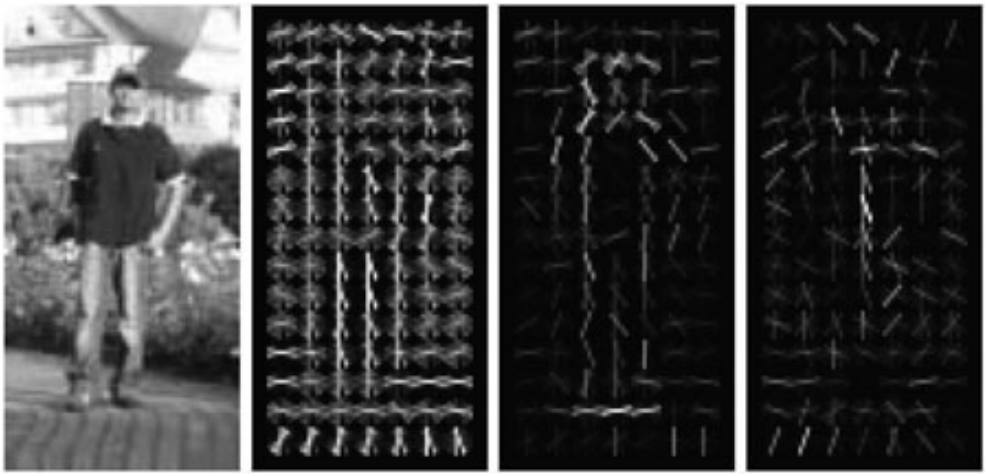

Figure 27: **Left**: typical pedestrian window.
Center left: HOG descriptor. Each of the orientation buckets in each window is a feature, and so has a corresponding weight in the linear SVM.
Center right: HOG descriptor weighted by positive weights, then visualized (so that an important feature is light). Notice how the head and shoulders curve and the lollipop shape gets strong positive weights.
Right: HOG descriptor weighted by the absolute value of negative weights, which means a feature that strongly suggests a person is not present is light. Notice how a strong vertical line in the center of the window is deprecated (because it suggests the window is not centered on a person). [Source: Forsyth/Ponce 2010; Fig 17.7]

Dalal and Triggs used histograms of gradients (HOG) descriptors, taken from a regular grid of overlapping windows (Fig. 27). Windows accumulate magnitude-weighted votes for gradients at particular orientations, just as in the SIFT descriptors (Section 12). Unlike SIFT, however, which is only evaluated at interest point locations, HOGs are taken from a regular grid and their descriptor magnitudes are normalized using an even coarser grid; they are only computed at a single scale and a fixed orientation. In order to capture the subtle variations in orientation around a person's outline, a large number of orientation bins is used and no smoothing is performed in the central difference gradient computation.

Figure 27 left shows a sample input image, while Figure 27 center left shows the associated HOG descriptors. Once the descriptors have been computed, a support vector machine (SVM) is trained on the resulting high-dimensional continuous descriptor vectors. Figures 27 center right and right show the corresponding weighted HOG responses. As you can see, there are a fair number of positive responses around the head, torso, and feet of the person, and relatively few negative responses, mainly around the middle and the neck of the sweater. Figure 28 shows the output of a pedestrian detector algorithm.

In Matlab we can extract those features with function `extractHOGfeatures`. A complete detector is provided in `vision.PeopleDetector`, initialized for instance with:

```
PeopDet = vision. PeopleDetector('UprightPeople_96x48');
```

We do not provide an example, as it is exactly the same as for face detection (Appendix **P.7.1**).

In Python HOG features can be extracted with `skimage.feature.hog`.

In OpenCV [py] the feature detector is called `cv2.HOGDescriptor()`. A full code example is given in **P.7.2**.

Applications Needless to say, that pedestrian detectors can be used in automotive safety applications. Matlab even has a code example for pedestrian detection.

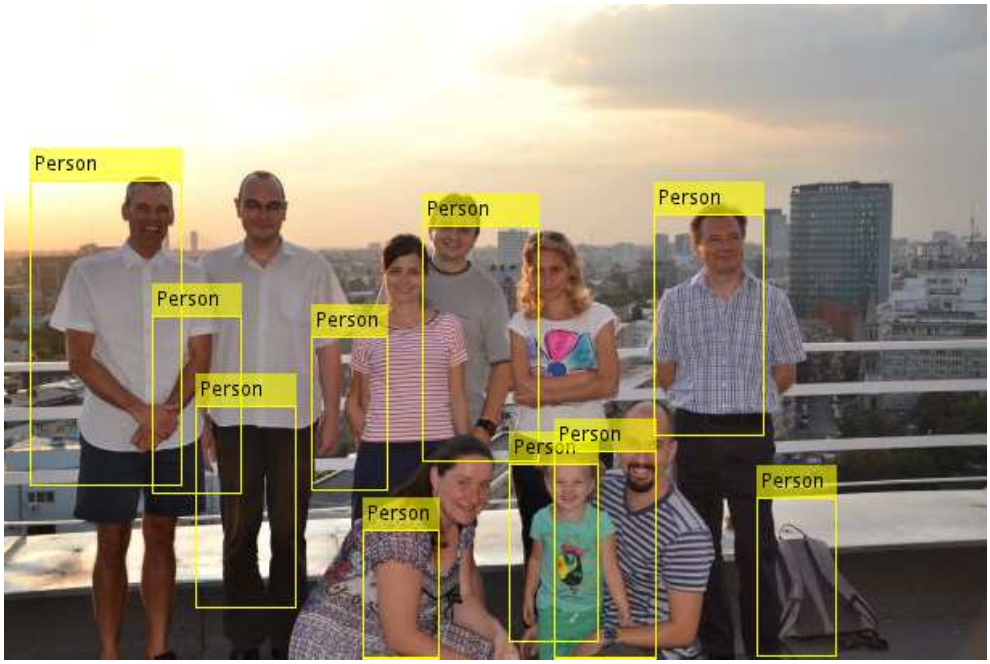

Figure 28: Example for the pedestrian detector; this picture may not exhibit the characteristic context for typical pedestrians, and hence we have quite a few misses and false alarms. Python code example in **P.7.2** (accessing OpenCV).

10.4 Evaluation

Detection algorithms sometimes fail to find all targets, e.g. in Fig. 26 two faces are not recognized. Those undetected targets are also called *misses*. Sometimes the algorithms make too many predictions, as seen in Fig. 28 where some structures are mistaken as persons. Those falsely identified locations are also called *false alarms*. In order to express how successful an detection algorithm is, one therefore needs to have labeled images, in which the target objects occur and ideally a lot of images where the target is absent. Then we determine the number of correctly identified targets, also called *hits*; we determine the number of missed targets (misses); and we count the number of false alarms. With that we can calculate the so-called precision and recall values, for which we refer to Appendix I.1.2.

10.5 Spatial Search (Accelerated)

Spatial search is the process that finds candidate object locations without performing a full feature extraction. The sliding window technique introduced above is a safe but time-consuming technique to detect objects, as it performs full feature extraction at every node of the search grid and the technique is therefore also called *exhaustive* search sometimes. We can distinguish two types of accelerated search: one using contextual cues, the other using a partial feature extraction.

Contextual Cues Some simple search improvements can be made by using what is also called *global* cues. For example for pedestrian detection we know that persons occur only in certain positions of a street scene (see Fig. 75). We can therefore exclude the zones where they are unlikely to occur. Furthermore, if we have a measure of spatial depth, then we can include size assumptions, reducing the scale search.

Partial Feature Extraction It is tempting to think that one can accelerate spatial search by directly jumping to potential object locations based on some combination of lower features. This works well if objects exhibit conspicuous color information, such as traffic signs or tail lights of vehicles: we take the corresponding chromatic channels and search for high intensities, see Appendix N.2. If objects exhibit simple shapes or occur at specific orientations, then we can exploit contour information as introduced in Sections 7.2 and 17; for example traffic signals and license plates show a rectangular silhouette whose elongated, parallel sides can be detected relatively reliably with edge contours.

The more complex objects are, the more challenging is it to define structural information that accelerates the search. There exist some approaches that pursue that and they return what is called *object proposals*. They are generally slow in comparison to the inference time of Deep Neural Networks, but one approach that has been reasonably successful is the ‘edge box’ by Zitnick and Dollar.

There exists some attempts to define a search based on saliency, that combines chromatic and structural information, but that has not produced convincing search algorithms that would directly point to object locations more accurately, than if we used the sliding window technique; we refer to Appendix G for more information.

If we attempt to accelerate search using cues from the object class, then Deep Nets become more convenient as they learn the necessary cues automatically. This is coming up next.

11 Object Recognition

Object recognition is a general term for detecting, classifying and localizing objects in arbitrary scenes; sometimes the process is also referred to merely as object localization. Whereas in the previous section we had introduced the binary classification task of detecting one object class in a scene, here we introduce systems that detect *multiple* object classes simultaneously. Figure 29 shows the output of such a system: it has correctly localized and categorized the three main objects in that image, as well as some parts of the objects.

The task is similar to semantic segmentation (Section 9.2.2) and so is the training method as well as the methodology. The crucial differences is that whereas in segmentation we often work with a specific scene type, in object recognition in contrast we attempt to detect objects in any type of scene and that makes the process very challenging. Today's object recognition systems recognize thousands of categories, not perfectly, but sufficiently well for some applications.

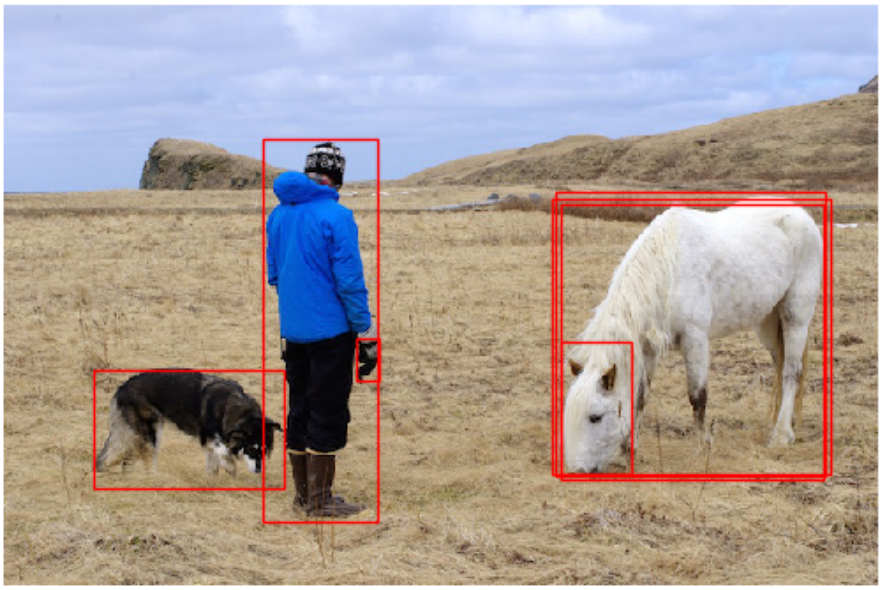

Figure 29: Object recognition is the localization of (multiple) trained object classes in arbitrary scenes. The output of a typical network will return the bounding boxes for localized object candidates. For this image, 7 object candidates are found, three of which strongly overlap (around the horse). For each candidate, the system calculates a confidence value, a so-called *score*, that expresses how certain it is of the predicted class type.

To train such a system we collect training material in similar way as we did for supervised segmentation (Section 9): we collect images containing the targets for each class type we wish to discriminate and we provide information where those targets are located in the image. In the early days of object recognition, that position information consisted merely of a so-called bounding box, the rectangle that comprises the object silhouette. This is of course the easiest way to provide position information. A more elaborate would be to provide also the object masks, analogous to the masks as provided for supervised segmentation. With that information, we can train the Deep Nets also to perform *instance segmentation*, the segmentation of individual object occurrences in the scene.

Before the arrival of Deep Net approaches, object recognition systems were similar to the pedestrian detector as introduced previously (Section 10.3), in particular the Local Feature approach was used (Section 3.2.2; Sections 12 and 13); their (spatial) search phase was essentially the sliding window technique. The entire recognition process was therefore a 'classic' object detection system where the classification step consisted of a multi-class classifier. In Deep Net approaches in contrast, both parts are more intertwined by the use of hierarchical architectures. The classification part is the same as introduced in Section 8;

the search phase is partly carried out by the hierarchical integration of spatial information. The Deep Net approaches vary quite substantially in their detailed approach to perform search.

Object recognition systems are typically evaluated on the so-called *COCO* dataset (http://cocodataset.org). To compare a system against others, it is trained for an agreed set of classes, several tens or even more. There are several measures to compare the systems: the absolute recognition accuracies, the precision of the object localization, the precision of the object segmentation, the inference time of the system (recognition speed), etc., see again the mentioning of the trade-offs in Fig 10. In the following we introduce first systems, that show relatively high recognition accuracies but that have a relatively long inference time (Section 11.1). Then we mention the intricacies of candidate selection, the process to select the principal candidates and to discard overlapping candidates (Section 11.2). Then we introduce a little miracle system, a network with a catchy name, that runs recognition in real-time (Section 11.3).

11.1 Accurate

Any object recognition system tries to classify its target objects with maximal accuracy possible, for a chosen (spatial) search algorithm. The localization however can be done with different degrees of precision. For the (binary) object detection systems as introduced in the previous section (Section 10), the precision did not matter much, the correct object count was given higher priority. In the systems introduced here, the localization precision matters more. One can distinguish three degrees:

Bounding Box: the object is framed by a rectangle that encompasses its outline (red rectangles in Fig. 29). The bounding box is specified with four values, either with two points (x- and y-coordinate), or with one point and the rectangle's width and height, depending on the software used.

Mask: the precise object outline is determined, as was done in segmentation networks (Section 9). A black-white image is generated with 1s representing the object. In this context, that process is also called *instance* segmentation. Figure 30 shows the masks from Fig. 29 combined to a single map containing all confidence values.

Key-Point Detection: the object's individual parts are detected. This is a first step toward pose estimation, a topic on which we elaborate in Section 26.4.

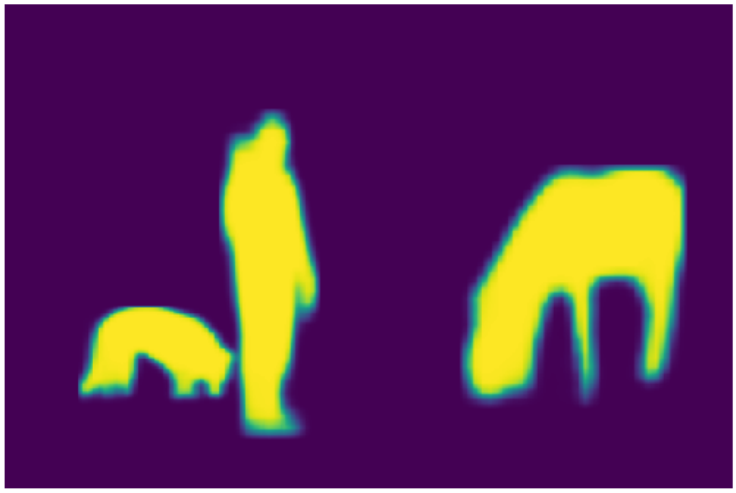

Figure 30: Instance segmentation. For each instance, one obtains a separate segmentation mask of the size of the image. Here, the estimated segmentation masks of the three best scoring candidates (from Fig. 29) are placed into a single map.

A network is typically trained for one of those three types of localization tasks. The larger the degree of desired precision, the larger is the network, the longer is its learning duration, the longer is its inference time.

We do not introduce the details of those network architectures. As one can imagine, those architectures are a melange of the systems introduced so far, the systems for classification, segmentation and search (Sections 8, 9 and 10, respectively). We merely introduce how to use them in PyTorch.

PyTorch offers models for object recognition in module `torchvision.models.detection`. The one for localizing the bounding box only is called `fasterrcnn_resnet50_fpn`, the one for instance segmentation `maskrcnn_resnet50_fpn`. We initialize the network as we would initialize a classification or segmentation network, then we change its mode to evaluation, and we feed it a list of images `aImg`.

```
NET     = ObjRec.fasterrcnn_resnet50_fpn(pretrained=True)
NET.eval()
aImgPred    = NET(aImg)   # [nImg][nChannels nRows nCols]
```

The output is a list of predictions for each image, `aImgPred`, containing the bounding boxes for each prediction, the assigned category label and the score value associated with that category. Appendix **P.8.1** gives a full example. The variable `COCO_INSTANCE_CATEGORY_NAMES` contains the category labels for which the network was trained.

11.2 Candidate Selection

Since we deal with multiple classes, it is possible that one object is assigned multiple classes, as shown in Fig. 29 with the three overlapping bounding boxes placed on the horse. Which one we eventually choose as our preferred candidate is a matter of context. If we lack any context information, it is easiest to select the one with the maximal score and suppress other overlapping candidates. This strategy is also called *non-maximum suppression*.

When we suppress other candidates, we can be very restrictive and prohibit any overlap with other candidates, but that would also mean that parts of a larger object may not be recognized; for example in Fig. 29, the glove of the person was properly identified as such, but if we did not permit any overlap between the candidates, it could be suppressed and therefore ignored. On the other hand, if we permit some overlap, then we risk also multiple guesses for the same scene object, such as the horse's head that was mistaken as sheep. Thus, in trying to eliminate overlapping candidates, we deal with a trade-off between identification accuracy and scene-labeling comprehensiveness.

When specifying the degree of overlap, then there exist several measures. A straightforward one is to determine the overlapping area, also called the *union of intersection* in some applications, abbreviated *IoU*.

In PyTorch such algorithms are found in module `torchvision.ops`. The one for non-maxima suppression is called `nms`. Its use is shown in the lower part of the example script **P.8.1**, where we feed the bounding boxes and the scores to the function and obtain a list of indices `IxKeep`, that are our remaining (selected) candidates:

```
from torchvision import ops as Ops
Prd     = aImgPred[0]           # prediction dictionary for this image
IxKeep  = Ops.nms(Prd['boxes'], Prd['scores'], 0.05)
```

In OpenCV [py] one such selection process is offered in the DeepNet module `dnn`:

```
IxSel   = cv.dnn.NMSBoxes(aCanBbox, aCanConf, thrConf, thrNonMaxSupp)
```

Its use is slightly different from the function in PyTorch and is explained in the next Section.

11.3 Fast: YOLO (You Only Look Once)

This network manages to find objects by an extremely efficient search: the classifier is applied only to an array of cells placed at a very coarse grid at multiple resolutions. Run on the appropriate hardware with CUDA cores, the system can classify at a rate of several images per second. The network's name is YOLO:

you only look once. It is available from the same provider of the DeepLearning software called DarkNet: https://pjreddie.com/darknet/yolo/.

The network's recognition accuracies are of course worse than those of systems with extensive spatial search, but the accuracies are very good nevertheless. The network is so efficient, that it is sometimes used as a reference: if a system does not show either better classification accuracy or faster search, then YOLO can be considered the better choice.

To apply the network for probing purposes, one needs to download only three files (May 2019):

- coco.names: the class labels. The network was trained on 80 classes.
- yolov3.cfg: a text file specifying the network architecture.
- yolov3.weights: the network weights. ca. 230 MB.

We can run the network with OpenCV, accessed through Python using the DeepNet module dnn. We firstly load the network:

```
NET         = cv.dnn.readNetFromDarknet(fileConfig, fileWeights)
```

An image is transformed into a blob image as follows:

```
Iblob    = cv.dnn.blobFromImage(Irgb, 1/255, (wthInput, hgtInput), \
                        [0,0,0], 1, crop=False) # [1 3 hgtInp wthInp]
```

Then we send the image through the network in Python itself with functions as introduced under image classification. The output of the network are three sets of arrays representing the class confidence values of every cell. One then performs a non-maxima suppression to find the most promising candidates:

```
IxSel   = cv.dnn.NMSBoxes(aCanBbox, aCanConf, thrConf, thrNonMaxSupp)
```

The full code is available in Appendix **P.8.2**.

12 Local Feature Approach

Sze p181, ch4, p205

Dav p149, ch6

We now turn toward the Local Feature approach, already mentioned in Section 3.2.2. It operates with small rectangular patches: from an image, one extracts a set of patches and then transforms each one (Fig. 31). To recognize objects or images, one would compare those transformed patches across images. The approach can be broken down into three phases:

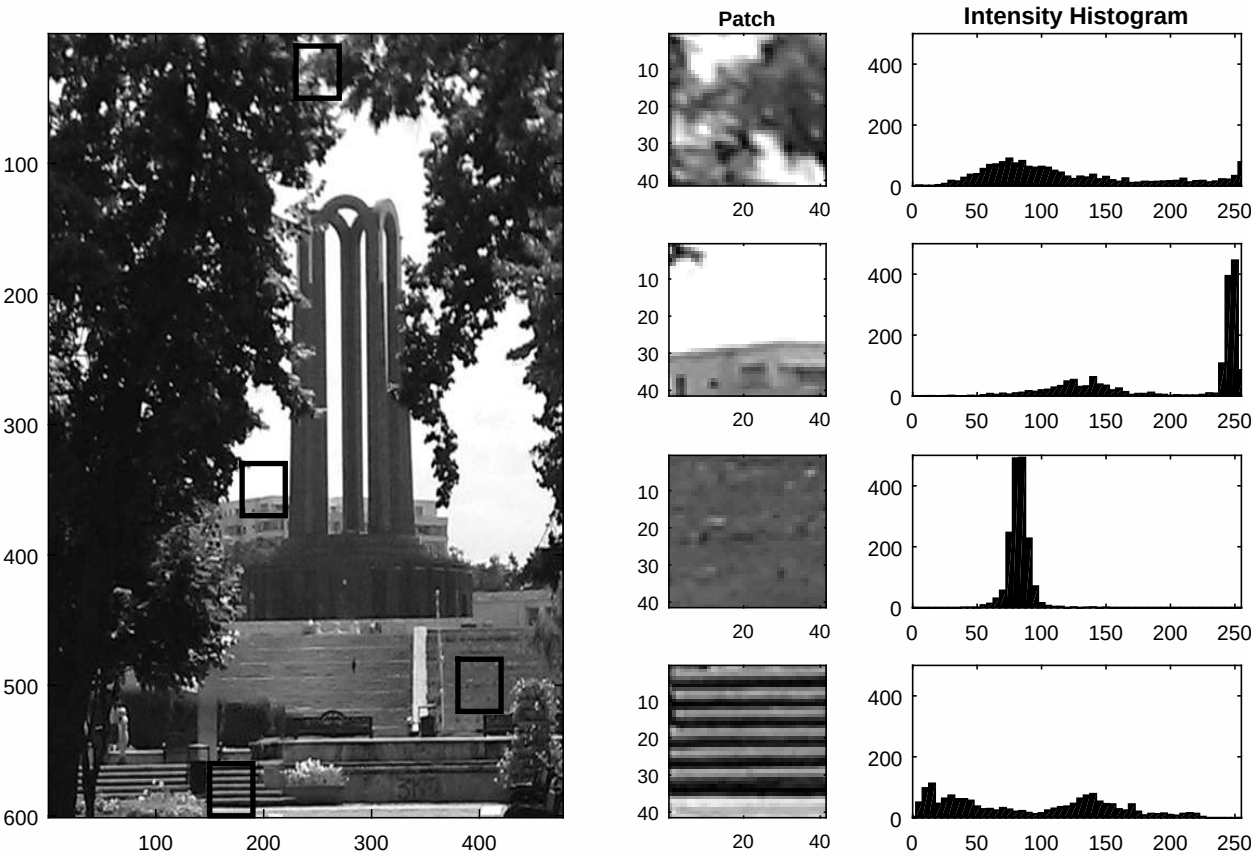

Figure 31: Examples of image patches and their corresponding description, in this case a mere intensity histogram - observe the four black rectangles in the image.
Center column: four patches detected in the picture. In this case the patches were selected randomly to illustrate the potential information they can offer. To compare objects or images, hundreds or thousands such patches are extracted from each image or object.
Right column: histogram of intensity values for each patch. This extraction and description process is rather simple but shown for illustration. In this section, more complex transformations will be introduced, for example based on the intensity gradients as introduced in Section 6.3.

Detection: the image is searched for unique point locations, whose context - the patch - represent the image or object as distinctively as possible. There are different names for the detectors, such as corner detector, interest point detector and key-point detector. Typically, several hundreds of such points are identified per image; the count between images varies.

Extraction/Description: transformation of the patch into a more abstract representation that can sustain certain variances, i.e. slight changes in viewpoint or illumination. The resulting vector is sometimes called a *descriptor*. An image or object is then represented by a matrix of vectors. In case of object representations, one describes the spatial relations between points/descriptors.

Matching: descriptors are matched in a pairwise manner. This can be done between two images to find correspondence, or between one image and a category representation that is based on such descriptors.

In the following three sections we will elaborate on these processes, but mention here already how the corresponding functions are called in Matlab and Python.

In Matlab functions carrying out these processes start with a corresponding keyword, e.g. `detectFeatures`, `extractFeatures` and `matchFeatures`.

In Python those processes are found in module `skimage.feature`, ie. `corner_xxx` and `match_descriptors`.

In OpenCV [py] those processes are found under the 'framework' `Feature2D` and hold the keywords `FeatureDetector` and `DescriptorExtractor`; more functions can be found under `ximgproc.xfeatures2d`.

12.1 Detection

Sze p185
FoPo p179, pdf 149
Dav p158, s6.5 s6.7
SHB p156, s5.3.10
Pnc p281, s13.2.2

Feature detection is the challenge of finding those points in the image, that are the most representative for the scene part or object part. This search is to some degree task dependent. If we intend to find the same object instance across images, then we wish to have as many reliable points as possible, as that facilitates object detection. If we intend to classify images into categories, that reliability is of less concern. In some classification studies, an explicit detection phase is omitted and instead patches are taken from points lying on a grid, because the difference in classification accuracy can be negligible. In some robotic applications, the detection phase is omitted to save on processing time and to accelerate the entire recognition process. In tasks such as tracking, useful points are important, see 'Good Features to Track' (Shi 1994).

We here introduce a corner detector, a feature that has been useful in a broad range of tasks. One way to attempt to find corners is to find edges - as introduced in Section 7.2 - , and then along walk the edges looking for a corner. This approach sometimes works poorly, because edge detectors often fail at corners. Also, at very sharp corners or at oddly oriented corners, gradient estimates are poor, because image smoothing smears any corner. At a 'regular' corner, we expect two important effects. First, there should be large gradients. Second, in a small neighborhood, the gradient orientation should swing sharply. We can identify corners by looking at variations in orientation within a window, which can be done by auto-correlating the gradients:

$$\mathcal{H} = \sum_{window} \{(\nabla I)(\nabla I)^T\} \tag{12}$$

$$\approx \sum_{window} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{13}$$

whereby $I_x = I_o * \frac{\partial g}{\partial x}$ and $I_y = I_o * \frac{\partial g}{\partial y}$ (g is a Gaussian). In a window of constant gray level and hence without any strong gradient, both eigenvalues of this matrix are small because all the terms are small. In a window containing an edge, we expect to see one large eigenvalue associated with gradients at the edge and one small eigenvalue because few gradients run in other directions. But in a window containing a corner, both eigenvalues should be large. The Harris corner detector looks for local maxima of the following formula

$$C = \det(\mathcal{H}) - k\Big(\frac{\text{trace}(\mathcal{H})}{2}\Big)^2 \tag{14}$$

where k is some constant, typically set between 0.04 and 0.06. The detector tests whether the product of the eigenvalues ($\det(\mathcal{H})$) is larger than the square of the average ($(\text{trace}(\mathcal{H})/2)^2$). Large, locally maximal values of this test function imply the eigenvalues are both large, which is what we want. These local maxima are then tested against a threshold. This type of detector is unaffected by translation and rotation. Algorithm 3 summarizes the method:
A code example of this detector is given in Appendix **P.9.1**. There are many variants of this type of feature detection, see for instance Dav p177, s6.7.6. For more code examples see also website links in FoPo p190,s5.5.

In Matlab the Harris detector is available with the function `detectHarrisFeatures`,

Algorithm 3 Feature detection. Sze p190, pdf 214

1. Compute the horizontal and vertical derivatives I_x and I_y of the original image by convolving it with derivatives of Gaussians.
2. Compute the three images corresponding to the outer products of these gradients.
3. Convolve each of these images with a larger Gaussian.
4. Compute a scalar interest measure using for instance equation 14.
5. Find local maxima above a certain threshold and report them as detected feature point locations.

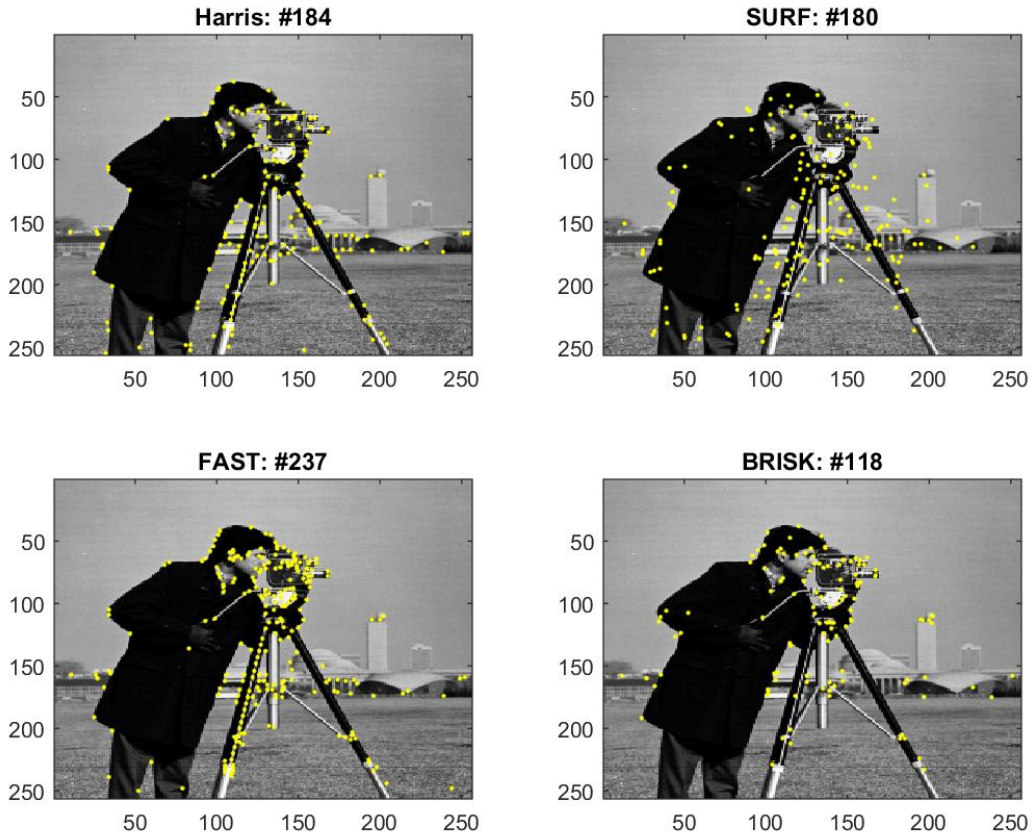

Figure 32: Comparison of the output of various feature detector algorithms, the localization of potentially useful pixels. Points can be used to establish correspondence between two similar images (Appendix E); or a small patched around them is extracted and transformed for matching (next Section). Each detector has advantages and disadvantages: some are accurate and therefore useful for both correspondence and matching; but those tend to be slow; other detectors are fast but less consistent across similar images. Thus, as usual, there is a speed-accuracy tradeoff. For some tasks, such as classification, no such feature detection algorithms are employed at all, and instead, points are taken from a grid, because it has been shown that this regular sampling can provide equal (or almost equal) classification accuracy; for that reason, detection is also omitted in some robotic tasks as this saves time.

```
FHar    = detectHarrisFeatures(I);
```

where output `FHar` is a structure that contains the locations and the significance of the detected features. An example of the detection output is shown in Fig. 32, along with the output of three other feature detectors. As one can see there are substantial differences in location as well as in detection count. Both, location and count, depend on the parameters settings.

Feature Tracking (in Scale Space): Most features found at coarse levels of smoothing are associated with large, high-contrast image events because for a feature to be marked at a coarse scale, a large pool of pixels need to agree that it is there. Typically, finding coarse-scale phenomena misestimates both the size and location of a feature. At fine scales, there are many features, some of which are associated with smaller, low-contrast events. One strategy for improving a set of features obtained at a fine scale is to track features across scales to a coarser scale and accept only the fine-scale features that have identifiable parents at a coarser scale. This strategy, known as feature tracking in principle, can suppress features resulting from textured regions (often referred to as noise) and features resulting from real noise.

12.2 Extraction and Description

FoPo p187, s5.4.1, pdf157
Dav p173, s6.7.3
Pnc p284, s13.3.2

The size of the patch to be extracted and its type of transformation varies and is also to some extent task dependent. In early approaches to local feature matching, one used intensity histograms as depicted in Fig. 31. With increasing computing power, more complex transformations were probed. There are two particularly successful types: one based on histogram of gradients, and the other the Local-Binary Patterns as introduced already in Section 7.3.3. The latter type does not need further introduction and we continue elaborating on the former type.

To generate a descriptor based on gradients, we preprocess the image as described in Section 6.3. For a given patch, we extract the corresponding gradient values (for each pixel) and form a histogram by binning the angle values. There are several exact schemes one can think of creating such a histogram. One successful histogram was introduced already when we introduced the pedestrian detector by Dalal and Triggs (Section 10.3), who called their descriptor simply Histogram-of-Gradients (HOG). It is taken from a relatively large, rectangular patch.

Most other descriptor types, are taken from smaller, square-size patches. The most famous one is the SIFT descriptor by Lowe, the scale-invariant feature transform (SIFT; wiki Scale-invariant_feature_transform). We introduce that one in more detail:

a) take the gradient (0-360 deg) at each pixel (from ∇I) in a 16×16 window around the detected keypoint (Section 6.3), using the appropriate level of the Gaussian pyramid at which the keypoint was detected.

b) the gradient magnitudes are downweighted by a Gaussian fall-off function (shown as a circle in Figure 33) in order to reduce the influence of gradients far from the center, as these are more affected by small misregistrations.

c) in each 4×4 quadrant, a gradient orientation histogram is formed by (conceptually) adding the weighted gradient value to one of 8 orientation histogram bins. To reduce the effects of location and dominant orientation misestimation, each of the original 256 weighted gradient magnitudes is softly added to $2 \times 2 \times 2$ histogram bins using trilinear interpolation. (Softly distributing values to adjacent histogram bins is generally a good idea in any application where histograms are being computed).

d) form an $4 \cdot 4 \cdot 8$ component vector $\mathbf{v}$ by concatenating the histograms: the resulting 128 non-negative values form a raw version of the SIFT descriptor vector. To reduce the effects of contrast or gain (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length: $\mathbf{u} = \mathbf{v}/\sqrt{\mathbf{v} \cdot \mathbf{v}}$.

e) to further make the descriptor robust to other photometric variations, values are clipped to $t = 0.2$: form $\mathbf{w}$ whose i'th element w_i is $\min(u_i, t)$. The resulting vector is once again renormalized to unit length: $\mathbf{d} = \mathbf{w}/\sqrt{\mathbf{w} \cdot \mathbf{w}}$.

The following code fragments give an idea of how to implement steps a-c:

```
EdGrad  = linspace(0,2*pi,8);           % edges to create 8 bins
[yo xo] = deal(pt(1),pt(2));            % coordinates of an interest point
Pdir    = Gbv.Dir(yo-7:yo+8,xo-7:xo+8); % 16 x 16 array from the dir map
Pmag    = Gbv.Mag(yo-7:yo+8,xo-7:xo+8); % 16 x 16 array from the mag map
Pw      = Pmag .* fspecial('gaussian',16, 4);     % weighting center
BLKmag  = im2col(Pw,   [4 4], 'distinct'); % quadrants columnwise for magnitude
BLKdir  = im2col(Pdir, [4 4], 'distinct'); % quadrants columnwise for direction
Gh      = [];
```

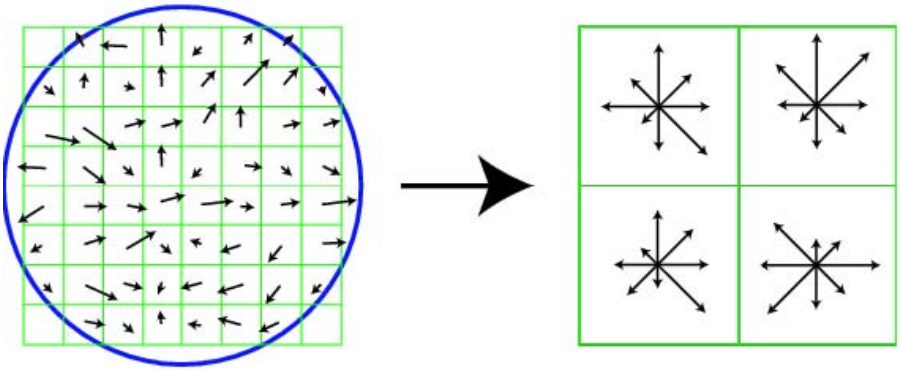

Figure 33: Forming SIFT features, shown for a 8x8 pixel patch in this illustration.
Left: Black arrows indicate the gradient; the blue circle represents the Gaussian weighting for histogram formation.
Right: formation of histograms demonstrated on 2x2 quadrants. Each quadrant represents a compass graph of the gradients in its domain.
[Source: Szeliski 2011; Fig 4.18]

```
for k = 1:16
    [HDir Bin]  = histc(BLKdir(:,k), EdGrad);
    HDw         = accumarray(Bin, round(BLKmag(:,k)*10000), [8 1]);
    Gh  = [Gh;  HDw];
end
```

In Matlab the function `extractFeatures` performs this transformation and one provides both the image and the detected points as arguments:

```
Dsc    = extractFeatures(I, FHar);
```

The output variable `Dsc` is a structure containing the descriptor vectors in array `.Features`

`Dsc.Features` of size `[nDsc nDim]` (15)

namely number of feature (descriptors) $\times$ number of dimensions. The default dimensionality is 64 in Matlab's implementation, but can be changed according to task demands.

12.3 Matching

To compare images, or to compare an image with a category representation, we match the descriptors of two lists. One list represents one image, for instance `L1 = Dsc1.Features`; the other list represents another image for instance `L2 = Dsc2.Features`, or a category represented by some list of descriptor vectors. In that matching procedure we are typically interested in the nearest neighbor, the closest descriptor to another descriptor vector; the second closest and farther descriptors are not of interest. This becomes then essentially the problem of nearest neighbor search, a topic we have elaborated on in Appendix K. Here it is introduced in short.

Matching descriptors can be carried out accurately and approximately, Appendices K.1 and K.2, respectively. The former is also called *exhaustive* matching because we take the distance between all pairs and that is very time consuming; it is therefore suitable for few descriptors only. The latter is necessary when we deal with large lists.

Accurate (Exhaustive) To compare two descriptor lists, $\mathbf{d}_i$ and $\mathbf{d}_j$ ($i = 1,..k, j = 1,..l$), we take the pairwise distances and form a $k \times l$ distance matrix D_{ij}. Matlab offers the function `pdist2` for that. Then we take the minimum in relation to one descriptor list, e.g. $\min_i D_{ij}$, and obtain the closest descriptor from

the other descriptor list. That would be a simple correspondence and may suffice if we compare a list of image descriptors and a list of category descriptors, as we will do for image classification (Section 13). If we intend to establish correspondences for registration (Section 21), we want to find the mutual matches.

```
L1, L2: the 2 descriptor lists, [nD1 x nDim] and [nD2 x nDim]
% ----- Compact version:
DM = pdist2(L1,L2);
% ----- Explicit version: (building DM ourselves)
DM  = zeros(nD1,nD2);
for i = 1 : nD1
   iL1rep  = repmat(L1(i,:), nD2, 1);         % replicate individual vector of L1 to size of L2
   Di      = sqrt(sum((iL1rep - L2).^2,2)); % Euclidean distance
   DM(i,:) = Di;                             % assign to distance matrix
end
% ----- Correspondence with respect to one list
[Mx2 Ix1] = min(DM,[],1);      % [1 x nD2]
[Mx1 Ix2] = min(DM,[],2);      % [nD1 x 1]
% ----- Correspondence mutual
IxP1to2 = [(1:nD1)' Ix2];      % [nD1 x 2] pairs with indices of 1st list and minima of 2nd list
IxP2to1 = [(1:nD2)' Ix1'];     % [nD2 x 2] pairs with indices of 2nd list and minima of 1st list
bMut1   = ismember(IxP1to2, IxP2to1(:,[2 1]), 'rows'); % binary array of mutual matches in list 1
IxMut1  = find(bMut1);         % mutually corresponding pairs with indexing to list 1
IxPMut1 = IxP1to2(IxMut1,:);   % [nMut1 x 2] mutal pairs of list 1
```

One may also want to use a for-loop for the maximum operation, that is to take the maximum row-wise, to avoid costly memory allocation, see Appendix K.

For Large Systems For large image collections, one easily extracts thousands or even millions of vectors. In that large-scale scenario, these accurate distance measurements are too slow and one resorts to nearest-neighbor search techniques that are slightly less accurate but allow to manage such large quantities (K.2). There are two principal techniques, one called *hashing function*, which works similar to a look-up table. Another technique is called KD-Tree: it constructs a decision tree of one list, which in turn is applied to the other list. Both tree construction and application occur in very short time due to the use of mere decision thresholds and not explicit distance measurements. The next section 12.4 gives a primitive example of how to apply that technique. Although that represents a significant speed-up, this is still not the fastest way of retrieving images and to further simplify descriptor matching one usually quantizes the feature space, a process introduced in the next Section 13.

In Matlab both methods, accurate and approximate, are implemented with function `matchFeatures`.

12.4 Examples, Notes

Now that we have learned about the three phases feature detection, feature extraction and feature matching, we can utilize that to find corresponding matches between two images (see also Appendix E). In a first example we do this for two very similar images; in a second example we find an object in another image.

Similar Images Given is a pair of images, depicting the same scene, but seen from two slightly different viewpoints. For each image we select and extract features, which then are compared to find pairs of identical points across the two image. This type of establishing correspondence is the starting point for many tasks such as optic flow (Section 20), image registration (Section 21.4) or depth perception (stereo correspondence, Section 22.1.1), Structure-from-Motion (Section 22.1.2). An example is given in Appendix **P.9.2**.

Object Finding In this example, Appendix **P.9.3**, we recognize a previously seen object in another image. The images in list `aImgNames` represent objects that we have seen at some point: we extract feature descriptors for each of those images and those descriptors, `AIMG`, are our object representation. The image in `Q.Iwhole` contains one of those objects. A search algorithm would firstly find the object location, but is

here provided manually for reason of simplicity, and assigned variable `Q.I`. Then we extract features for that query region and we match those by an efficient nearest neighbor search using a kd-tree, `KDTreeSearcher` and `knnsearch`, see also K.2.

A key advantage of using matching with sets of keypoints is that it permits finding correspondences even in the presence of clutter (occlusion), large scale differences and orientation changes. Patches used to be employed also for image classification, but were meanwhile surpassed by Deep Neural Networks in prediction accuracy, but not yet in inference time. To classify images, we need however to bring the list of descriptors firstly into a format suitable for pattern recognition algorithms, an image vector (Section 3.3). This is best done by clustering the space, a procedure we introduce next (Section 13).

Implementations Apart from the implementations offered in OpenCV, there exist many other software packages. For example, here are two that offer a Matlab interface:

`http://www.vlfeat.org/`

`http://www.aishack.in/2010/05/sift-scale-invariant-feature-transform/`

Literature

Tuytelaars and Mikolajczyk, 2008. Local invariant feature detectors: a survey

Miksik and Mikolajczyk, 2012. Evaluation of Local Detectors and Descriptors for Fast Feature Matching

13 Feature Space Quantization

FoPo p203, s6.2, pdf 164
Sze p612, s14.4.1, pdf 718

Feature space quantization is the compression of the feature space that we have created with some local feature extraction method. This is necessary when we extract huge numbers of local features such as the gradient-based descriptor introduced in the previous Section 12. For example when we collect feature descriptors (eq. 15) from hundreds of thousands of images, then we end up with millions of vectors. In that 'big-data' scenario, employing a matching scheme as introduced in Sections 12.3 and 12.4 is too slow, even if we measure approximate distances using a kd-tree. But as we have millions of features, the feature space is so dense, that some features are nearly identical and we can therefore compress the space by joining features into clusters. In information theoretic terms, we quantize the space. To illustrate the idea, we look at an example from image compression, specifically color encoding:

Example Quantization An image is stored with 24 bits/pixel and can have up to 16 million colors. Assume we have a color screen with 8 bits/pixel that can display only 256 colors. We want to find the best 256 colors among all 16 million colors such that the image using only the 256 colors in the palette looks as close as possible to the original image. This is color quantization where we map from high to lower resolution. In the general case, the aim is to map from a continuous space to a discrete space; this process is called vector quantization. Of course we can always quantize uniformly, but this wastes the colormap by assigning entries to colors not existing in the image, or would not assign extra entries to colors frequently used in the image. For example, if the image is a seascape, we expect to see many shades of blue and maybe no red. So the distribution of the colormap entries should reflect the original density as close as possible placing many entries in high-density regions, discarding regions where there is no data. Color quantization is typically done with the K-Means clustering technique.

The Principal Applied to Features In our case, we aim to find clusters among our features that represent typical characteristics of objects or scenes. In the domain of image classification and object recognition, these clusters are sometimes also called *(visual) words*, as their presence or absence in an image, corresponds to the presence or absence of words in a document; in texture recognition they are also called *textons*. The list of words represents a 'pooled' representation or a *dictionary*, also called *bag of words*, with which we then recognize objects and scenes. Thus, in order to apply this principal, there are two phases (Fig. 34): one phase builds the dictionary - upper half of the Figure; the other phase applies the dictionary. The two phases are now introduced in Sections 13.1 and 13.2, respectively. A full example is given in **P.10**, in the following we discuss sections of this code.

13.1 Building a Dictionary

Our starting point is our list of vectors, the features extracted from the entire image set, all concatenated to a single list. In the code example **P.9.3** this list was called `DSC`:

```
DSC         = double(vertcat(AIMG.Dsc));  % vectors for entire data set
```

where `ntDsc` is our total feature count for the entire image set, `nDim` the dimensionality of our feature (space). Now we apply a clustering procedure to that list, a procedure called K-Means, for which we specify a number k of desired clusters. In our case, k represents the number of words, the dictionary size; in the above example it is 256 colors. How many words we need in order to classify our images properly is a matter of experimenting. We start with 500 words:

```
nWrd        = 500;
[WrdNo WRDc]= kmeans(DSC, nWrd, 'onlinephase','off');
```

The function returns the array `WrdNo` of length `ntDsc` containing word numbers from `1` to `nWrd`. Now we replace the feature vectors of each image with histograms for those words. For each image we generate a feature histogram of length `nWrd`, which is our image vector (Section 3.3):

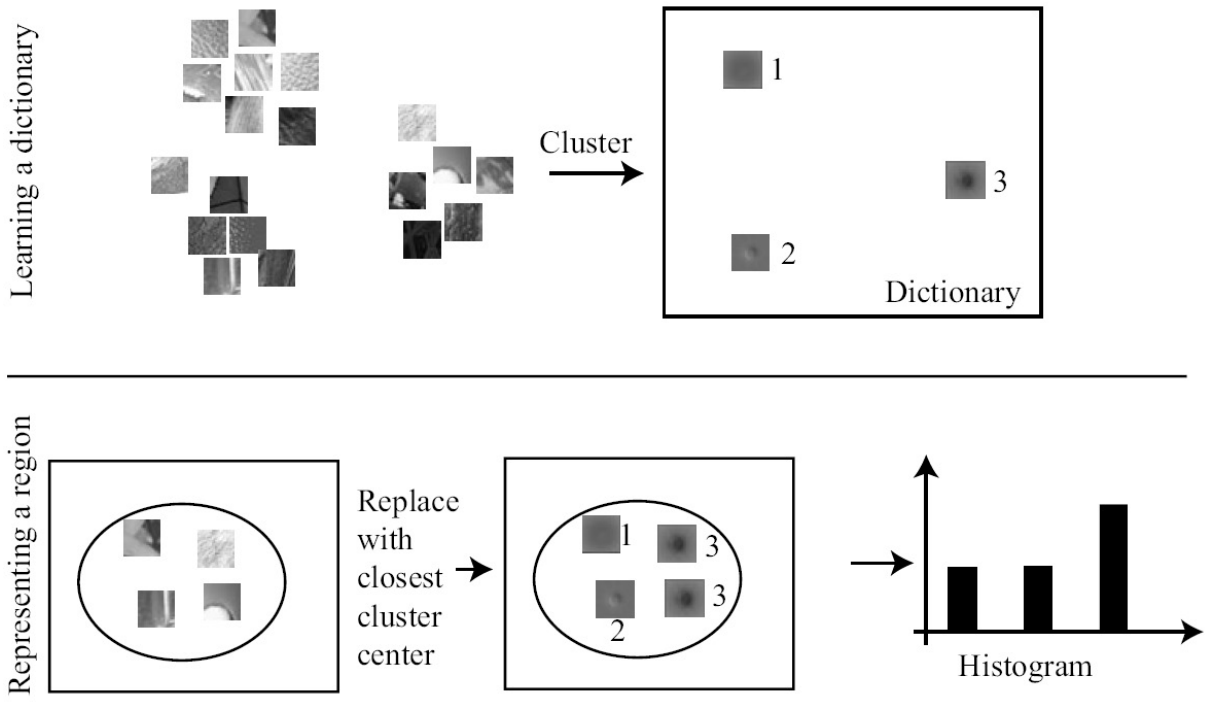

Figure 34: Using local feature vectors for classification.
Upper part Learning a dictionary: 3 clusters were found in the training set (using some clustering technique) that are called *words*; the set of words represent the dictionary.
Lower part Representing a region (object or entire scene): replace detected features with the closest cluster label and create a histogram. Histogram length corresponds to the number of words in our dictionary (3 in this illustration); the histogram represents the image vector suitable for pattern recognition algorithms (Section 3.3). [Source: Forsyth/Ponce 2010; Fig 6.8]

```
IWHST  = zeros(nImg,nWrd);
for i = 1:nImg
    Bimg        = LbImg==i;
    IWHST(i,:) = histcounts(WrdNo(Bimg)',0:nWrd);
end
```

where array `LbImg` is of length `ntDsc` and holds the indices to the images, see first (computational) section in code example of Appendix **P.10**.

Array `IWHST` is our new, compressed representation of `DSC`. To `IWHST` we can apply (traditional) classification algorithms, such as Support Vector Machines, Random Forest, etc.. See again Section 3.3 and also Appendix H for both, more details on the K-Means algorithm and on how to apply classifiers. In Section 14.3.1 we will explain the K-Means procedure in more detail.

13.2 Applying the Dictionary to Novel Images

When we apply our system to novel images, we need to obtain a word histogram in order to compare it to our classifier that was trained on `IWHST`. To do that we need to know the actual word vectors, which we have obtained from Matlab's K-Means implementation with array `WRDc`. In the code example in **P.10.1** we show how to do that explicitly - in some software packages those cluster centers are not provided and need to be computed.

With the actual word vectors (`WRDc` or `WRD`), we then generate the histogram of words for a given image using `knnsearch` as we have used it previously, see examples in **P.9.3**:

```
[NN DIS]    = knnsearch(WRD, Dsc);
Iwhst       = histcounts(NN, 0:c);
```

Histogram `Iwhst` is now of the format as array `IWHST` and can be used for retrieval or classification.

13.3 Refinement and Scalability

The methodology for quantization as explained so far is a coarse implementation. There are many refinements that one can make to improve classification and retrieval results. One is to apply a threshold that determines when a feature is close enough to its center (word). That threshold would be based on the distances of features within a cluster, proportional to the cluster size for example. Features that are assigned to a given word, but whose distances are larger than the corresponding threshold would be ignored.

Another refinement is to ignore overly frequent words, because they may not provide a lot of discriminatory power for classification and may be even prohibitive. Those high-frequency words are also called *stop words* and in order to identify those one has to experiment a bit to find an optimal threshold that improves recognition.

In Matlab we can employ the functions `bagOfFeatures` and `trainImageCategoryClassifier` to perform the quantization for us. Appendix **P.10.2** gives an example.

Scalability The great strength of the Local Feature approach is its scalability, the ability to apply a method to large image collections. The automotive industry builds systems that can recognize street scenes accurately, in order to facilitate (automatic) vehicle navigation (Section 25.2). This is essentially an image retrieval problem and further tricks how to exploit this method are given in Section 18.3.

14 Segmentation (Unsupervised, Image Processing II)

SHB p176, ch6
FoPo p285, ch9, pdf255
Sze p235, ch5,pdf267

Image segmentation is the task of partitioning a scene into its constituent regions. We had already introduced the supervised approach of segmentation using Deep Autoencoders in Section 9. Here we introduce the unsupervised approach and also mention approaches that use semi-supervised guidance, i.e. segmentation initiated by a user in an interactive system.

In the early days of computer vision, unsupervised segmentation was considered to be an essential, early step in a systematic and deterministic reconstruction of a scene (see again the historical note under 'Feature Engineering' of Section 3.4). This is sometimes referred to as *semantic* or *general* segmentation and is still pursued in particular in Content-Based Image Retrieval (CBIR) and video analysis, where one is often faced with a large number of different scenes. But in general, image segmentation nowadays is pursued for more specific tasks.

If the task is to separate an object in the center of the image from its background, then this is also called *figure-ground segregation*. If there are multiple objects in the scene, one speaks rather of *foreground-background* segmentation. In video analysis, segmentation is often considered the task of segregating the moving objects from the stationary background.

Functionally speaking, segmentation is often defined as a search for groups of pixels of a certain coherence. This loose definition makes certainly sense if regions are homogeneous, but regions themselves can also exhibit certain patterns sometimes, which makes that definition already a bit fuzzy.

In the following, the presentation of segmentation methods is ordered along their degree of complexity. Simple segmentation processes carry out mere thresholding of images, a technique that does not always provide satisfactory results, but that is completed quickly and is therefore suitable for applications with time constraints, such as video processing (Section 14.1). Then there exist segmentation processes that regard the image as a landscape (Section 14.2). Another class of segmentation algorithms regards the challenge as a data analysis process and applies clustering algorithms, which essentially are statistical methods (Section 14.3). In the final section, we introduce a general image segmentation procedure (Section 14.4).

14.1 Thresholding (Histogramming)

SHB p177, s6.1
Dav p82, ch4, pdf119

Segmentation by thresholding is often also called histogram segmentation, because one typically starts by generating a histogram of the pixels' intensity values. The histogram was introduced in Section 4.1), another one is shown in the upper right of Fig. 35 (see also SHB p24, s.2.3.2). In many (simpler) scenes, the modes (maxima) of such an intensity distribution correspond to objects and backgrounds and one therefore searches for optimal thresholds by analyzing those modes. As the typical image is encoded in 8 bits, the histogram is generated with 256 bins, with values ranging from 0 to 255.

Global If the objects and background are of distinct intensity, then the distribution in the histogram will appear bimodal, as in Fig. 35. The most straightforward way to determine a threshold would then be to take the minimum between the two maxima. But there exists more clever analysis that investigates the cumulative distribution function (CDF) of the histogram, such as Otsu's method, see the (vertical) blue stippled line in the histogram. Applying a single threshold to the image is also called *global* thresholding.

Band, Multi-Level Instead of splitting the intensity distribution in two parts as in global thresholding, it may also make sense to specify ranges. If we specify a range that constitutes one class, and the values outside that range another class, then that would be called a *band* threshold.

If a histogram shows multiple modes, as is the case for many complex scenes, then we can attempt to specify several thresholds, in which case we talk of *multi-level* thresholding. That is unlikely to give us good segmentation results, but it may work if we search for specific regions parts or objects that possess a relatively distinct gray-level. For instance thresholding has been applied for road segmentation and for vehicle location in in-vehicle vision systems. To find the appropriate thresholds we can think of different techniques. One would be to extend the technique for bimodal thresholding - as mentioned under global thresholding - to the individual modes of the multi-modal distribution. Another technique would be the use

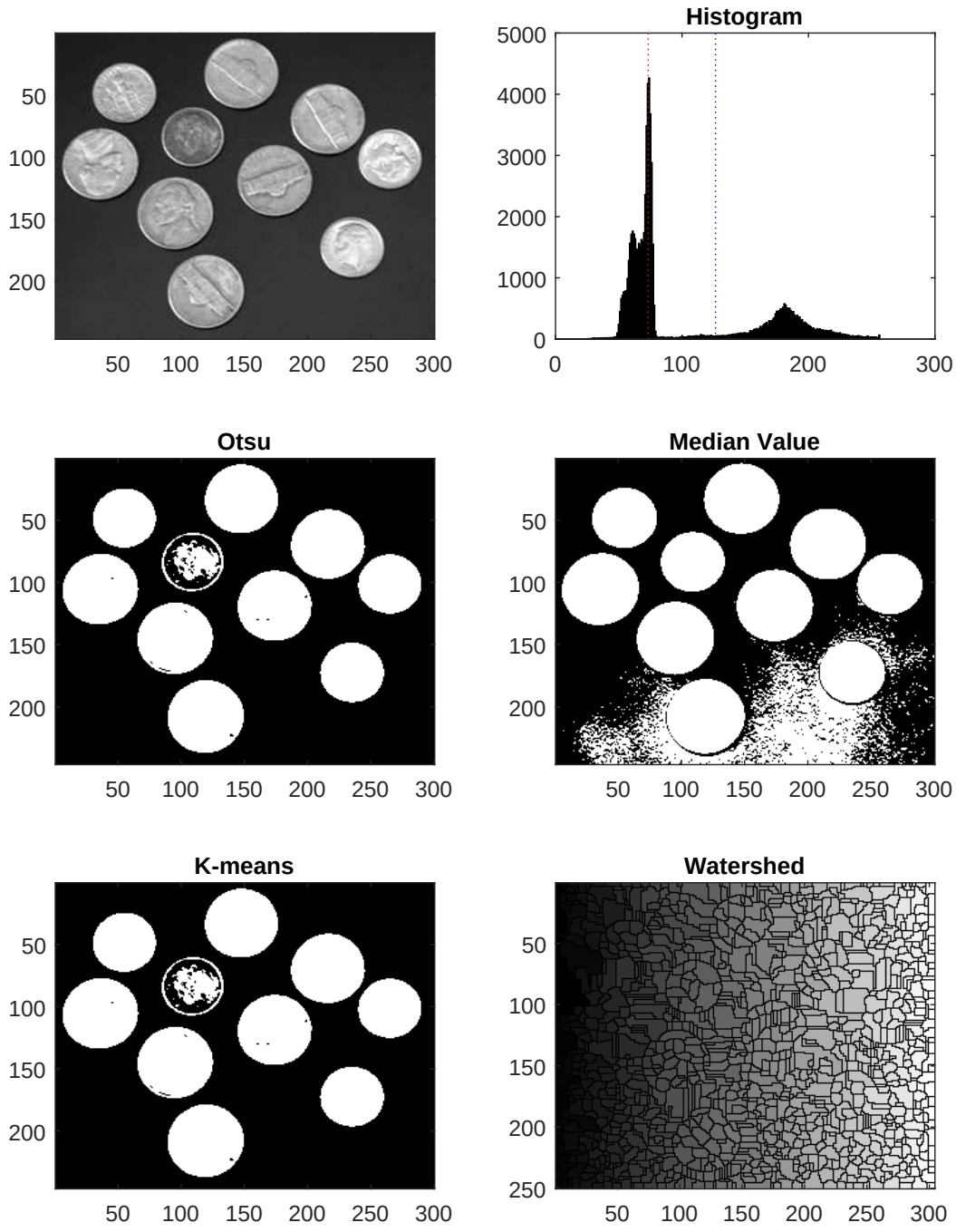

Figure 35: Segmentation methods in comparison (code in **P.11.1**).
Upper Left: image 'coins' from Matlab, `imread('coins.png')`.
Upper Right: intensity histogram distribution; the vertical blue stippled line is the threshold value of Otsu's method (in Matlab `graythresh`); the vertical red stippled line is the median value.
Center Left: binary map as thresholded with Otsu's method.
Center Right: output as produced with a median threshold (shown for comparison).
Lower Left: clustering result with K-Means algorithm (`kmeans`).
Lower Right: output of the watershed algorithm. The watershed algorithm typically over-segments. In this case, the outline of the coins can be recognized nevertheless (with some effort).

of mode detection by means of a maxima search, a technique we applied for face part detection (Section 5.3).

Adaptive/Local There exists scenes that exhibit a slowly changing background color, which - seen as an intensity landscape - constitutes a shallow slope (gradient) across the entire image. The image in Fig. 35 shows signs of such a gradient, which is evident when we apply a median threshold, see center right graph. It then would make sense to search for a threshold value that is based on only a neighborhood and not the entire image, hence also called *local* or *adaptive* thresholding. This has proven to be particularly useful in document analysis, where photographed documents can exhibit a strong gradient due to light shining from one direction (Section 26.2.1).

In Matlab one can look at the intensity histogram with the function `imhist`; the function `graythresh` finds an optimal threshold between two peaks; the function `im2bw` thresholds the image using the level as specified by `graythres`. The function `multithres` is based on the method used in function `graythres`.

> **Caution** `graythresh/im2bw`**.** The function `graythresh` always generates the level as a scalar between 0 (black) and 1 (white) irrespective of the image's class type ('uint8', 'single', etc.). Correspondingly, the function `im2bw` expects a level specified between 0 and 1 and the original image class type. If you have converted the original image to class 'single' already - as recommended in previous exercises - then this may produce wrong results. Here it is better to work with the original image class type.

In Python those functions can be found in particular in sub-module `skimage.filters`, e.g. `threshold_otsu`. Python offers more methods to determine global thresholds than Matlab does.

In OpenCV [py] there exist the function `calcHist` for calculating the histogram, for thresholding one would use the function `threshold`:

```
H = cv.calcHist([Igry.flatten()],[0],None,[256],[0,256]) thr, BW =
cv.threshold(Igry,0,255,cv.THRESH_BINARY + cv.THRESH_OTSU)
```

Some of this is very well illustrated on the OpenCV documentation pages:
https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

Application Thresholding is particularly appealing for tasks that require fast segmentation, as in video processing for example.

14.2 Region Growing: The Watershed Algorithm

SHB p233, s6.3.4
Sze p251, s5.2.1, pdf283

In segmentation by region growing, one starts with several, selected pixels and then keeps expanding from those pixels until some stopping condition is fulfilled. The most popular algorithm is the watershed algorithm and as its name implies, one floods the intensity landscape until the rising water level meets the watersheds. To imagine that, we observe the image as an intensity landscape, see Fig. 12 again. In a first step, the algorithm determines the landscape's (local) minima and then proceeds by growing from those minima until the flood front encounters another growing flood front. The points of encounter form lines that correspond to watersheds. The resulting regions can be regarded as catchment basins, where rain would flow into the same lake.

Watershed segmentation is usually applied to a smoothed version of the gradient magnitude image ($\| \nabla I \|$, Section 6.3), thus finding smooth regions separated by visible (higher gradient) boundaries.

Watershed segmentation often leads to over-segmentation (see lower right in Figure 35), that is a segmentation into too many regions. Watershed segmentation is therefore often used as part of an interactive system, where the user first marks seed locations (with a click or a short stroke) that correspond to the centers of different desired components.

Matlab `watershed`
Python `skimage.segmentation.watershed`

Advantages no specification of the number of clusters necessary or any other parameter.
Disadvantages over-segmentation, slow

14.3 Clustering [Statistical Methods]

Segmentation can also be carried out with statistical methods, in particular with *clustering* methods. Clustering is part of the field of machine learning and is a so-called *unsupervised* classification method, as the method does not exploit any labels for finding groups. In principle, any clustering method can be used for image segmentation; practically we prefer the ones that complete their search process in reasonable time. The fastest method is the *K-Means* algorithm (Section 14.3.1), a method already used for feature quantization (Section 13). Another fast method is the *Mean-Shift* and its variants (Section 14.3.2). A slower method is the *Normalized-Cut* method, which is often used for foreground-background segregation (Section 14.3.3).

For all those statistical methods the image is reshape to a single array whose length n corresponds to the total number of pixels. A clustering method therefore treats pixels as individual data points in principle. For each pixel we specify our desired features. In the simplest case, the feature corresponds solely to the gray-scale intensity, in which case we deal with a one-dimensional array only of length n. If we use all three chromatic channels RGB, then we would deal with a $n \times 3$ matrix, a data matrix with feature dimensionality equal three. In those two cases, it is clustering purely in intensity space. We can also add the pixel coordinates as x and y values, thus creating a $n \times 5$ matrix, in which case we put back neighboring information into the segmentation task. We can also add local texture measurements, as discussed in Section 7.3, in which case we move toward an even higher-dimensional space. In short, a clustering algorithm expects a $n \times f$ matrix D, where f is the number of features under investigation. The clustering algorithm then tries to find groups in that f-dimensional space. The more features we take however, the larger becomes the search space and the slower becomes the clustering process.

14.3.1 K-Means

wiki K-means_clustering

The K-Means algorithm is perhaps the most used clustering procedure for segmentation. We used the K-Means procedure already for feature quantization (Section 13.1), namely on a 64-dimensional problem. Here we use it to on a much lower-dimensional space, for instance in Fig. 35 we used merely a single dimension, namely gray-scale intensity. If we deal with a color image, then it would be obvious to start clustering with the three chromatic channels (RGB). We use that now as an example. The first step is then to reshape the image `Irgb` to a column-wise representation that would correspond to the data matrix D, called `ICol` here:

```
szI   = size(Irgb);             % [rows columns 3]
nPix  = szI(1)*szI(2);          % total number of image pixels
ICol  = reshape(Irgb,[nPix 3]); % turn image into data matrix for clustering [nPix 3]
```

To run a K-Means algorithm we need to specify an expected number of clusters, k. The algorithm is initiated by choosing k pixels randomly from the image, where k is named `nCls` for clarity:

```
IxCtr   = randsample(nPix,nCls);    % random pixel indices
Ctr     = ICol(IxCtr,:);            % initial centroids
```

The points in `Ctr` are our so-called *centroids*, that represent our group centers. As we chose them randomly, they unlikely represent the true group centers, but we move toward those by an iterative procedure consisting of two steps. In a first step, we determine which pixels are nearest to which centroid and we therefore compare each centroid with all other pixels:

```
D       = zeros(size(ICol));        % [nPix 3]
% === Distances Centroid to All
for c = 1:nCls
    ctr     = Ctr(c,:);         % one centroid [1 nDim]
    Df      = bsxfun(@minus, ctr, ICol);   % difference only
    D(:,c)  = sum(Df.^2,2);     % square suffices (we don't root)
```

```
end
% === Find Nearest Centroid
[v LbCls] = min(D,[],2);
```

We compute only the square and not the full distance (in f-dimensional space), as we need only to know the nearest cluster and not its actual distance to it. Array `LbCls` now holds the cluster labels $\in$ [1,nCls].

In a second step, we then compute the mean values of those clusters:

```
for c = 1:nCls
    Ctr(c,:)    = mean(ICol(LbCls==c,:));
end
```

We take the new centroids `Ctr` and repeat the two steps. We continue iterating those two steps until the centroids have moved toward the actual group centers. This is usually achieved with a few tens of iterations. To visualize the obtained clusters we need only to reshape the label vector to image size and display it with `imagesc` for instance (in Matlab); it can be regarded as a *label matrix*.

Complexity Although this iterative procedure appears complex, it is in fact computationally relatively efficient as opposed to other clustering algorithms. Its complexity is only, $O(nkfi)$, where n is the number of image pixels; k the number of specified clusters; f the feature dimensionality and i the number of iterations. The important issue here is that the complexity is linear to n and not square as in some other algorithms.

There are two downsides to the K-Means algorithm. One is that the initial cluster centroids are chosen randomly and that sometimes result in relatively different partitions when we run the algorithm repeatedly. There exist simple tricks to improve that initial selection and software implementations of the K-Means algorithm include those by default, called K-Means++ initialization. Nevertheless, the problem persists to some degree. The problem can be further mitigated by running the K-Means procedure repeatedly and choosing the run with the smallest overall distance for the final clustering; the disadvantage of that is that it multiplies the complexity by the number of runs.

Another downside of the procedure is that one needs to specify a number of clusters k a priori, meaning one needs to know how many different objects one expects. Suppose we have an image of a flower with a homogeneous background (leaves of other plants). Then perhaps we could start the segmentation with $k = 3$: one cluster for the flower leaves, one cluster for the background, and one for the remaining parts of the flower. But that inspection is not possible when we dealing with large databases and one possibility is to run the algorithms for varying k and choosing the smallest distance.

In Matlab there exists a function that offers a highly optimized iteration procedure. We can apply it as follows:

```
szI   = size(Irgb);                       % [rows columns 3]
ICol  = reshape(Irgb,[szI(1)*szI(2) 3]);  % turn image into data matrix for clustering [nPix 3]
Lb    = kmeans(single(ICol),3, 'MaxIter',50, 'online','off'); % Labels E [1,2,3]
Lmx   = reshape(Lb, szI(1:2));    % turn into label matrix
```

It is recommended to experiment first with the online phase turned `off`; the default is set to `on` and that takes a very long time, for relatively little improvement, even for images of moderate size.

In Python we can use the function `KMeans` from the module `sklearn.cluster`:

```
from sklearn.cluster import KMeans
...
ICol    = Irgb.reshape((nPix,3)).astype(float)
RKM     = KMeans(n_clusters=nK, max_iter=20, n_init=1).fit(ICol)
```

The variable `n_init` determines how many times the algorithm is run. The output `RKM` is a structure (class) that contains variables for the labels. See Appendix **P.11.2** for full examples.

In OpenCV [py] we need to be a bit more explicit. It requires also the specification of an epsilon value, a convergence limit: when the centroids do not move anymore during updating, then the iteration cycle terminates.

```
termCrit    = (cv.TERM_CRITERIA_MAX_ITER, 10, 0.1)     # iter, epsilon
flagInit    = cv.KMEANS_PP_CENTERS                     # kmeans++
mes, Lb, Cen= cv.kmeans(ICol, k, None, termCrit, nRep, flagInit)
```

Advantages relatively fast: faster than other clustering algorithms, but slower than thresholding
Disadvantages specification of k: number of expected clusters needs to be specified beforehand

14.3.2 Mean-Shift, Quick-Shift

The *Mean-Shift* procedure is similar to the K-Means in its use, but here one specifies a bandwidth value h and not an expected number of clusters. That has the obvious advantage that we do not need to provide a particular $\mathtt{k}$, yet if we do not specify an adequate bandwidth value, then the results can be unsatisfying. Thus the challenge now is to estimate a reasonable h value, but in some cases that is more convenient than providing a fixed $\mathtt{k}$.

The mean-shift algorithm does not only compute distance values in f-dimensional space, but it calculates gradient values. The idea is that data points around cluster centers show a higher density, which practically is the case in most situations. And to move toward those densities during the search procedure, one would therefore follow the gradient. Although elegant, it has two disadvantages. One is that in practice this gradient search has been shown useful only to a dimensionality f of up to 6 or 7. Another disadvantage is that calculating gradients is more complicated than calculating distances and for that reason the algorithm is slower than the K-Means algorithm. For the latter reason, a fast variant was developed named *Quick-Shift*. The Mathworks site contains examples of those algorithms.

This type of segmentation is often used for tracking moving objects. An object in a scene often has a characteristic color histogram 'signature', that is distinct from its context, and that is exploited with this type of algorithm (Section 19.3).

14.3.3 Normalized Cut

The normalized-cut algorithm is a clustering algorithm that is used in particular for generating two clusters. It searches for those two clusters by using an algorithm based on Graph Theory and that involves calculating the distances between all pixels (in f-dimensional space). That means it shows square complexity $O(n^2)$ and that makes the algorithm substantially slower than the previous algorithms. The advantage of that thoroughness is that one arrives at better segmentation results, but the procedure also suffers from initialization issues as it is a priori not clear which regions are to be segregated into two groups. Practically, the algorithm is therefore often used for a segregation between foreground and background, for example in a graphics programs to extract an object from its background. To initiate the procedure, it requires two samples, one from each cluster. Practically, those two pixels are provided by the user, for instance manually by mouse-clicks, one pixel from the background and one from the foreground. Due to those circumstances, the algorithm is rather used for analysis of selected images, less so for analysis of large image sets.

Matlab does not offer a specific function, but the Mathworks website offers implementations provided by users. Python however offers an implementation with module SciKit-Learn, see Section 9.4 of the SciKit-Learn documentation.

14.4 General Segmentation through Hierarchical Segregation

Now we introduce a general segmentation procedure that partitions the image into regions of similar color. In contrast to the above methods, it generates not one, but multiple segmentation maps of increasing detail. The procedure starts with a fast, simple segmentation method as we had introduced with global thesholding or clustering (Sections 14.1 and 14.3.1, resp.). This will typically result in an under-segmentation. Here we

create a finer segmentation map by segmenting each one of those regions again. We can continue this repeated segmentation until the image is sufficiently segmented. An example is shown in Fig. 36.

This type of segmentation works well only if we use a segregation into two groups, a binary segregation. Two methods are suitable for binary splitting: global thresholding and K-Means clustering with $k = 2$. This procedure creates a hierarchial tree, whose depth corresponds to the number of segregation steps. Typically, three segregation steps are sufficient, sometimes it requires four to find our desired regions. The tree has therefore depth equal three or four. The resulting segmentation maps contain scene parts, object parts and sometimes entire objects.

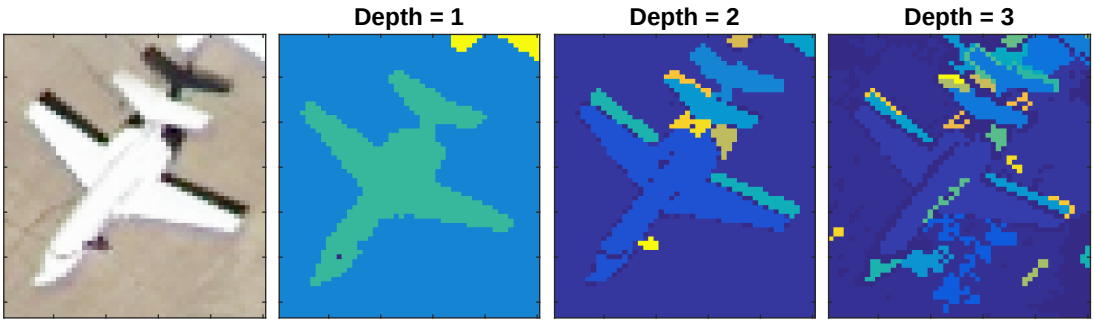

Figure 36: Segmentation by hierarchical segregating individual regions. Code example in **P.11.3**.
Depth=1: connected components (contiguous regions) for segregation of the entire image (ie. kmeans with $k = 2$ or global thresholding). Region brightness corresponds to the inverse of its area.
Depth=2: result of segregating each region of depth=1 again the same manner.
Depth=3: result of segregating each region of depth=2 again. A hierarchical tree is grown that way.

To determine the regions of a segmentation map, we need to run an algorithm that finds adjacent pixels, something which we will introduce in more detail in the upcoming Section 15. A code example is shown in **P.11.3**. It uses gray-level thresholding for segregation. It can be easily modified to used 2-Means as segregation technique. 2-Means segregation may be more suitable for segregating scenes that contain subtly different colors. For most images, segregation by gray-level thresholding appears quite fine.

Complexity The appealing aspect of this procedure is its relatively low complexity. The complexity is roughly proportional to depth. If we use global thresholding as segregation, then connected-component labeling is the most time consuming process. If we use 2-Means clustering, then the complexity is determined by the number of iterations in the clustering procedure.

Further motivation is given in

https://www.researchgate.net/publication/334393498

15 Morphology and Regions (Image Processing III)

Sze p112, s3.3.2, p127

Dav p185, ch7

Morphological processing is the local manipulation of the structure in an image - its morphology - toward a desired goal. Often, the manipulations are a preparatory step for extracting regions, contours, shapes, etc. Morphological processing can be thought of as complex convolution processes; they can carry out complex filtering processes that are the result of algorithmic processing and not just filtering.

After we have detected regions or contours, the resulting segmentation maps may contain 'irregularities' that we would like to eliminate. We may want to eliminate isolated pixels or small regions immediately. That is already considered morphological processing. Or regions and contours may have frequent one-pixel protrusions that we may want to polish without affecting the actual region of contour segment. Such examples are part of so-called *binary* morphology, coming up in Section 15.1. But one can also apply such manipulations to gray-scale images, called *gray-scale* morphology, which we will quickly introduce in Section 15.2.

After we have completed morphological processing, we would like to localize the regions in the black-white image and start describing them. The localization process is also called *region finding* or *region labeling*, to be introduced in Section 15.3. The description of a region is a topic for shape description (the next section), but we will mention in this section the use of some of the functions of software packages.

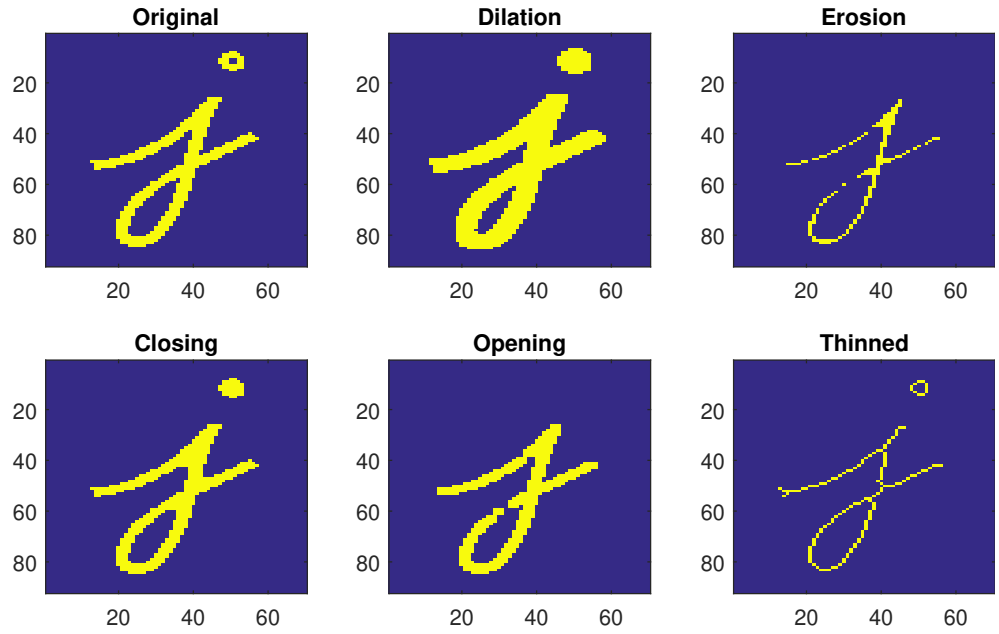

Figure 37: Binary morphological processing exemplified on a hand-written letter 'j'.
Original: the two objects to be manipulated: the main shape and a dot.
Dilation: a thickened version of the original - dilated by one.
Erosion: a slimmed version of the original object - eroded by one. Note that the main shape is now ruptured. Erosion by two would erase the shape completely.
Closing: this process consists of two operations: first dilation of the original, followed by erosion. The dot is closed now while the main shape is back at its original contour width.
Opening: first erosion of the original, followed by dilation. Note that some of the ruptures are now more evident; the dot has disappeared.
Thinned: the objects are repeatedly eroded until their 'skeletons' remain.

15.1 Binary Morphology

Binary morphology manipulates only binary (black-white) images whereby the result is again a binary image. There exist two basic operations, the erosion and the dilation operation, which - as their names imply - make the *object(s)* shrink or grow in some specified way. Those two basic operations are then combined to form more complex operations. We introduce those operations with Figure 37:

Dilation: the object gains one or several layers of pixels along its boundary (center graph in Fig.37). The equivalent convolution process would be to run a summation filter and then threshold at value equal 0; an example is given below in the Matlab section.

Erosion: the object looses one or several 'layers' of pixels along its boundary. The equivalent convolution process would be to run a summation filter with the central pixel turned off and then accept only those pixels whose summed value corresponds to the maximum filter value.

If one applies the above two operations in sequence, then that causes the following modifications:

Closing: is the dilation followed by an erosion; this fuses narrow breaks in the black-white pattern and fills small holes and gaps.

Opening: is the erosion followed by a dilation; this eliminates small objects and sharpens peaks in an object.

The last two sequential operations can be summarized as follows: they tend to leave large regions and smooth boundaries unaffected, while removing small objects or holes and smoothing boundaries.

Thinning: is the process of reducing the object toward its skeleton by repeated erosion (lower right in Fig. 37). This process is particularly useful for contour tracing: after one has detected edges (or other contour pixels), the edge map often exhibits locations, where the structure is more than one pixel wide; thinning helps eliminating those locations.

Other combinations of those basic operations are possible, leading to relatively complex filtering operations such as the **tophat** and **bothat** operations, which however can also become increasingly time-consuming. When developing an application, it is difficult to foresee which morphological operations are optimal for the task: one simply has to try out a lot of combinations of such operations and observe carefully the output.

More formally speaking, the image is modified by use of a *structuring element*, which is moved through the image - very much like in the convolution process. The structuring element can be any shape in principal: it can be a simple 3×3 box filter as in the examples above; or it can be a more complicated structure, for instance some simple shape that one attempts to find in the image, leading so essentially to a filter process.

If one deals with large images, then one of the first steps toward object characterization or region finding might be to eliminate very small regions, which in software programs we can do with a single function. This is of benefit because morphological operations are carried out much faster after one has eliminated any unlikely object candidates.

In Matlab binary manipulations are carried out with commands starting with the letters `bw` standing for black-white. Here are examples of some essential manipulations, whereby most manipulations can be found in function `bwmorph`:

```
BW = imclearborder(BW); % deletes regions touching image border BW =
bwareaopen(BW,4); % deletes regions < 4 pixels (4-pix remain!)
areaObjs = bwarea(BW); % summed areas of all objects

BW      = bwmorph(BW,'dilate',1);   % dilates by one
BW      = bwmorph(BW,'erode',2);    % erodes by two
BWper   = bwperim(BW);              % returns boundary pixels of objects
BWrem   = bwmorph(BW,'remove');     % removes interior of pixels
BWthn   = bwmorph(BW,'thin',inf);   % thinned
```

The dilation and erosion operation by a value of 1 can be expressed as convolution as follows:

```
SeDil   = ones(3,3);
SeEro   = ones(3,3); SeEro(2,2)=0;              % creates a 'hole'
BWdil = conv2(single(BW), SeDil,'same')>0;      % same as bwmorph(BW,'dilate',1)
BWero = conv2(single(BW), SeEro,'same')==8;     % same as bwmorph(BW,'erode',1)
```

Thinning is more complicated as it should not eliminate any structure. There exist several methods. In **P.12.1** we give an example of an algorithm without further explanations.

More sophisticated manipulations can be achieved using a structural element defined with `strel`. There also exist functions such as `bwareafilt` and `bwpropfilt` to carry out complex filtering operations.

In Python the functions are found in different sub-modules, in `scipy.ndimage.binary_xxx` as well as in `skimage.morphology`. We directly give examples:

```
from skimage.morphology import remove_small_objects, remove_small_holes, \
                               binary_erosion, binary_dilation, thin, disk
BWlrg   = remove_small_objects(BW, 4)
BWnoho  = remove_small_holes(BW, 4)
BWdil   = binary_dilation(BW)           # dilation by one (=disk(1)=5-pix cross)
BWero   = binary_erosion(BW,disk(2))    # erosion by two
BWthin  = thin(BW)                      # thins automatically to inf
```

In OpenCV [py] we need to define the structuring element first:

```
StcElm   = cv.getStructuringElement(cv.MORPH_ELLIPSE,(3,3))
BWero    = cv.morphologyEx(BW, cv.MORPH_ERODE, StcElm, iterations=2)
BWdil    = cv.morphologyEx(BW, cv.MORPH_DILATE, StcElm, iterations=2)
BWopn    = cv.morphologyEx(BW, cv.MORPH_OPEN, StcElm, iterations=2)

# in-place
cv.morphologyEx(BWero, cv.MORPH_ERODE, StcElm, dst=BWero, iterations=2)
```

15.2 Grayscale Morphology

The above introduced operations also exist for gray-scale morphology, but here the operation is not the change of a bit value, yet the selection of an extrema value in the neighborhood under investigation. In gray-scale morphology one talks of the *structural function*: in the simplest case, the function takes the maximum or minimum.

In Matlab those operations are found under the initial letters `im`, for instance `imdilate`, `imerode`, `imopen` and `imclose`.

In Python the operations can be found in `scipy.ndimage.grey_xxx` or some of them in `skimage.morphology`. For instance `skimage.morphology.dilation` carries out dilation for a gray-scale image and can also be applied to a black-white image, but for the latter the above functions starting with `binary_` are faster.

15.3 Region Finding (and Simple Description)

After segmentation and morphological processing, we need routines that search the map for connected pixels that make up the individual regions. There are two principal approaches. We either collect all region pixels, or we trace only the region boundary. Either approach results in different region information, from which we can derive different types of region parameters. Of course, one can pursue both approaches, but that also increases the complexity.

To collect all region pixels, one generates a *label matrix* in which each region receives a separate label value. In that context, one talks of *connected components*, which are defined as regions of adjacent pixels

that have the same input value (or label). Connected components are the objects of interest and receive value 'true' or some other label; the region between the connected components is considered background and may consist of one or more regions that receive the value 'false'. We will give an algorithm of this process in Section 15.3.1.

To trace the region boundary, we search for some initial pixel of the region boundary, and then walk along the boundary until we meet the initial, starting pixel again. This procedure is also called *boundary following* or *boundary tracing*. We will introduce an algorithm in Section 15.3.2.

In the following we will firstly introduce how to use the routines in software packages. We thereby also introduce the function that returns a simple parameterization of regions, also called *region properties* sometimes. That function can be called immediately after morphological processing (or segmentation) and carries out region labeling as a first step, without the need to explicitly trigger the process.

In Matlab we can find and characterize connected components in different ways. Region labels and indices can be obtained in two ways:

1. `bwlabel`: returns a label matrix with connected components numbered 1,..$n_{objects}$. To obtain the region pixels, we then would write a loop to find the indices using function `find`.

   ```
   [Lmx nCC] = bwlabel(BW);
   aRegIx = cell(nCC,1);
   for i=1:nCC,
       aRegIx{i} = find(Lmx==i);
   end
   ```

   This procedure is not a very efficient way of finding the region pixels, but should illustrate the idea behind the process. The procedure is inefficient, because we generate and scan `nCC` black-white maps, which is rather costly. The following functions are more efficient.

2. `bwconncomp`: returns a structure with fields corresponding to the objects' indices, information that the function `bwlabel` does not provide. If one desires a labeled matrix, as it is created with `bwlabel`, then we apply the function `labelmatrix`.

The advantage of using `bwconncomp` is, that it requires less memory than `bwlabel`.

To obtain the region boundaries we can use the function `bwboundaries`, applied to the entire black-white map and receive a set of lists of boundary pixels, `aBon`:

```
[aBon MLbon] = bwboundaries(BW, 8, 'noholes');
```

`MLbon` holds the label matrix.

To describe the regions with a few parameters, we deploy `regionprops`. It can be applied in a variety of ways. Here are explained three ways. We can apply the function `regionprops` directly to the black-white image `BW`, in which case we have skipped the function `bwconncomp` - it is carried out by `regionprops` in that case. Or we apply the function `bwconncomp` first and then feed its output to `regionprops`. Or we use the function `bwboundaries`, which is useful if we intend to describe the silhouette of shapes, for example with a radial description (as will be introduced in the next Section 16).

```
I       = imread('cameraman.tif');
BW      = im2bw(I,128/255);

%% =====    RegionProps directly from BW   ========
RPbw  = regionprops('table',BW,I,'maxintensity','area');

%% =====    RegionProps via bwconncomp   ========
CC      = bwconncomp(BW);
RPcon  = regionprops('table',CC,I,'maxintensity','area');

%% =====    RegionProps via bwboundaries   ========
[aBon MLbon]    = bwboundaries(BW,8,'noholes');
RPbon  = regionprops('table',MLbon,I,'maxintensity','area');

%% -----    Verification  --------
assert(all(RPbw.Area==RPcon.Area));     % compare direct and bwconncomp
```

```
% via bwboundaries: only same for bwboundaries(BW,8,'noholes');
assert(all(RPcon.Area==RPbon.Area));    % compare bwconncomp and bwboundaries
```

The function `regionprops` provides only simple region descriptions. Many of them can be computed very quickly, but the following four will increase computation duration significantly: 'ConvexHull', 'ConvexImage', 'ConvexArea', 'FilledImage'. For more complex descriptions, one inevitably moves toward the topic of shape description and that will be introduced in the upcoming section.

In Python there exists slightly less flexibility than in Matlab (as of 2019). The sub-module `skimage.measure` provides the functions `label` and `regionprops`. The label function requires type `int` as input, and the function `regionprops` takes only a label matrix as input:

```
from skimage.measure import label, regionprops
...
LB = label(BW)        # we call 'label' first, because...
RG = regionprops(LB)  # ...input to 'regionprops' must be of type int! (not logical)
nShp = len(RG)
print('# Shapes', nShp)
```

To obtain the list values as a table - as a single array -, we write a loop as follows, a formulation that is called *list comprehension* in Python:

```
Ara = asarray([r.area for r in RG])        # obtaining all area values
```

In Python, there exists no function that does boundary tracing explicitly, but we can use a function that is designed to trace iso-contours, which we will introduce in Section 17.2.2.

In OpenCV [py] there are several functions. The bare function script `connectedComponents` returns only the number of detected regions and the label matrix `Lmx`. With `connectedComponentsWithAlgorithm` we can specify a different labeling algorithm. With `connectedComponentsWithStats` we obtain minimal region properties. In OpenCV, the background is included in the region count as well, and appears as the first entry in the list of region properties and centers.

```
BW        = BW.astype(uint8)    # source must be uint
nC1, Lmx1 = cv2.connectedComponents(BW, connectivity=8)
nC2, Lmx2 = cv2.connectedComponentsWithAlgorithm(BW, connectivity=8, \
                ltype=cv2.CV_16U, ccltype=cv2.CCL_DEFAULT)
# Prop is array [nReg 5]: left, top, width, height, area
nC3, Lmx3, Prop, Cen = cv2.connectedComponentsWithStats(BW)
# exclude background (1st entry)
nC3     -= 1                # now we have number of conn comps
Prop    = Prop[1:,:]
Cen     = Cen[1:,:]
# sort by decreasing area
Odec    = (Prop[:,-1]).argsort()[::-1]
Prop    = Prop[Odec,:]      # region properties ordered
Cen     = Cen[Odec,:]       # region centers ordered
```

15.3.1 Connected Components

wiki Connected-c

Connected components can be determined with different degrees of *connectivity*:

8-connectivity: this type of connectivity regards *any* of its 8 adjacent pixels as neighbors. A checkerboard pattern is regarded as one region with this type of connectivity. This choice of connectivity results in fewer and larger regions than the use of the 4-connectivity neighborhood. It is the default connectivity in many software packages.

4-connectivity: this type regards only the two neighbors along the vertical or the horizontal axis as neighbors (four in total). This connectivity is tighter than the 8-connectivity and therefore results in more regions in total than the 8-connectivity. Applied to a checkerboard pattern, this connectivity sees only isolated pixels.

The goal is of course to find the connected components as quickly as possible and with low memory demands. As usual, there is no algorithm that is optimal for all scenarios. For example, if we know that our map contains only few pixels of interest (foreground), then we deploy a different algorithm than if the regions of interest cover the entire map. In addition, there are trade-offs concerning memory and computation - even for a single scenario. Those details are better explained by an expert and here we merely introduce a generic scheme that is suitable when regions cover most of the image and background is sparse.

The algorithm to be introduced is called the *two-pass* algorithm. It determines the labels using two scans of the entire image. In a first pass (scan), pixels to the right and to the bottom are connected. This will label only a few regions properly (of a general segmentation map), but the majority of regions are labeled with multiple, unique labels. In a second pass, those multi-label regions are unified to a label and marked as such in the amp. The (current) wikipages explain the procedure well, but do not offer a full example.

A crucial part of the procedure are the set functions 'union' and 'find' that turn the multi-label regions into a single (final) label region. The union function is deployed in the first pass. The find function in the second pass. To understand the detailed operations, our example treated the union-find operations separately, as a separate step between the first and the second phase, see now example in **P.12.2**. The function `f_SetDisJointSimp` carries out that union-find procedure.

The first pass collects the candidate labels in list `aLnk=;`. That list of candidate labels is then fed to function `f_SetDisJointSimp` and returns a shortened list with unique labels. In a second pass, the final labeling is generated. Our implementation works row-wise - as in a C program -, which is not optimal for Matlab, but serves as a template to translate it to C. In order to compare our implementation with that of Matlab, we need to rotate our input, as Matlab's implementation progresses through a map row-wise (as in Fortran).

If one intends to dive into the details of different labeling algorithms then the following website may be a good starting point, which also lists recent papers:

https://github.com/prittt/YACCLAB

15.3.2 Boundary Following

To follow the boundary of a region, we firstly need to identify a suitable starting pixel. From that starting pixel on, one then would walk along the boundary identifying adjacent border pixels. A starting pixel can be identified for instance with the above method for locating connected component candidates utilized during the first pass, i.e. finding a pixel that constitutes an upper left corner of a region. That upper left corner serves as starting pixel.

There exist different algorithms for boundary tracing and some of them are explained and compared well on the following web pages:

http://www.imageprocessingplace.com/downloads_V3/root_downloads/tutorials/contour_tracing_Abeer_George_Ghuneim/alg.html

The example we give, **P.12.3**, is the one based on the Moore algorithm. The code example processes only one contour for simplicity of illustration. Our implementation relies on a starting pixel as detected with the first pass method for connected component labeling. This is emulated here by using the following line:

```
[rw cl] = find(BW==1, 1, 'first');
```

The next line, `dir1st = [1; 0];`, instructs the algorithm to start walking north. The algorithm proceeds to walk in a clock-wise manner around the region boundary.

16 Shape

Dav p227, ch 9 and 10

Shape means the geometry or form of a region or object. In its simplest definition, it is the shape of a region as obtained by a segmentation process, i.e. a connected component. Shape can also be understood as a set of such regions. Or it can be defined as a set of points only. Shape can also be regarded as a structure consisting of several segments. Obviously, there does not exist a strict definition of shape, because visual patterns can be very diverse. But texture or appearance is typically not included when talking of shape. For a shape recognition task, one typically designs a set of segmentation steps that manipulate the image to produce shape regions suitable for a specific shape description technique. For example we would apply morphological processing - as introduced in the previous section - to select certain shape candidates for further processing.

Shape description techniques are used in the following applications for example:

- Medical imaging: to detect shape changes related to illness (tumor detection) or to aid surgical planning
- Archeology: to find similar objects or missing parts
- Architecture: to identify objects that spatially fit into a specific structure
- Computer-aided design, computer-aided manufacturing: to process and to compare designs of mechanical parts or design objects.
- Entertainment industry (movies, games): to construct and process geometric models or animations

In many applications, the task to be solved is the process of *retrieval*, namely the ordering (sorting) of shapes, and less so the process of classification, see Section 2 again to understand the difference. In a retrieval process, one shape is compared to all other shapes, and that is computationally much more intensive than just classification. Thus, one major concern in a retrieval task is the speed of the entire matching process.

The number of shape matching techniques is almost innumerous, two crude ones were mentioned already in Section 5. Each technique has its advantages and disadvantages and works often for a specific task and merely under certain conditions. Textbooks are typically shy of elaborating on this topic - with the exception of Davies' book - , because there is no dominating method and it is somewhat unsatisfactory and endless to present all techniques. Perhaps the two most important aspects in choosing a shape matching technique are its matching duration and its robustness to shape variability.

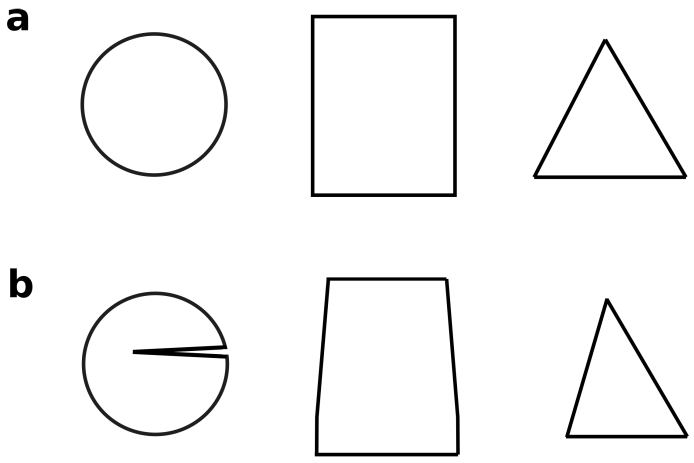

Figure 38: Shapes and their deformations. **a**. Shapes in 'original' form, for example as we would draw them. From left to right: circle, ie. a celestial body (sun or moon); rectangle, ie. a window frame; triangle, ie. a coniferous tree. **b**. Example of deformations. Left to right: the celestial body may be partly covered by a thin cloud; the window may be covered by blinds hanging into the rectangular frame; the tree has been struck by a Blitz and is deformed since.

Shape Variability Shapes can vary in size or in spatial orientation; they can be at different positions in the image; they can appear mirrored; their parts may be aligned slightly differently amongst class instances; their context may differ. These different types of 'shape condition' are also called *variability*. Among those variabilities, the deformations are perhaps the most challenging issue, see Fig. 38.

Ideally, a shape matching technique would be invariant to all those variabilities. The table below shows the terminology used with respect to those desired shape matching properties:

Variability	**Invariance**
size	scaling
orientation	rotation
position	translation
laterality (mirroring)	reflection
alignment of parts	articulation
blur, cracks, noise	deformation
presence of clutter	occlusion

Practically, it is impossible to account for all these invariances and therefore one needs to observe what type of variability is present in the shape database and make a choice of the most suitable technique. This choice is also important for classification techniques.

Description Testing Shape descriptions are often tested on the MPEG7 shape database, a shape set comprising a total of 1400 silhouettes, 70 classes with 20 instances each:

https://dabi.temple.edu/external/shape/MPEG7/dataset.html

The retrieval accuracy of the various methods are summarized here:

https://dabi.temple.edu/external/shape/MPEG7/results.html

In this section we introduce some of those methods; some other methods will be introduced in the section on contours (Section 17).

The first three subsections will introduce shape descriptions of increasing complexity. Firstly, we introduce simple parametric measures suitable for rapid retrieval (Section 16.1). Then, we introduce a description suitable in particular for shape boundaries (Section 16.2). Section 16.3 introduces techniques based on point comparisons: those techniques have a longer matching duration but show better retrieval accuracy. Section 16.4 introduces part-based descriptions: they can be even more accurate, but the matching techniques are rather complicated. Thus, the choice of a matching technique is also a matter of dealing with a speed-accuracy tradeoff. In the final section 16.5, we have a word on shape classification systems.

16.1 Simple Measures

The measures introduced in this section, are parametric measures that are in particular suitable for a rapid description of a single connected component (region). One can use those measures to form a feature vector, which allows to apply classical pattern recognition methodology. The simplest measures are:

- **area**: number of pixels
- **perimeter**: arc length of the region's boundary
- **major and minor axis length**: measured as the two axes of an ellipse encompassing the shape
- **equivalent diameter**: using the area as a parameter, one assumes the shape to be a circle and calculates the corresponding diameter

With those parameters we can calculate other parameters:

- **circularity**: area divided by equivalent diameter
- **axis ratio (aspect ratio; eccentricity)**: ratio between minor and major axis

Another useful measurement is the convex hull, which however takes a bit longer to calculate. The convex hull is the outline of a shape that shows no concavities. For a circle, rectangle, triangle or any other

symmetric (non-intersecting) polygon, the convex shape is the same as the shape itself. But if those shapes showed indentations (concavities), then the convex shape would correspond to the outline without any identations. For an 'N'-shape, the convex hull would be the rectangle made of its four corners. Based on the convex hull we can gain another two shape parameters:

- **convex area**: area of the shape's convex hull
- **solidity**: ratio between (actual) area and convex area

In software packages we obtain those parameters with functions called `regionprops` and we have introduced those already previously, for example in Section 15.3. We point them out again:

Matlab `regionprops`, introduced in Section 15.3 already
Python `regionprops` in submodule `skimage.measure`

We then concatenate those parameter values to a vector an perform distance measurements between shapes. The code examples in **P.13.1** and **P.13.2** show how to do that.

Such feature vectors are not sufficiently discriminative for complex shapes such as in the MPEG7 dataset, but can be useful for describing texture made of small shapes.

16.2 Radial Description (Centroidal Profile)

Dav p269, s 10.3

The radial description is suitable for a shape, whose boundary is a single, closed curve (or nearly closed), e.g. obtained using `bwboundary` in Matlab or `find_contours` in Python (see Section 15.3 or 17). With the boundary coordinates we firstly calculate the *pole* (centroid) of the shape by taking the mean of the coordinates. Then we measure the radii from the pole to its boundary pixels arriving so at the *radial signature* also called *centroidal profile*. The signature is the sequence of distances $R(s)$ from the pole to each boundary pixel s.

Figure 39 shows the signatures for four simple shapes. For a circle, the radial signature is a constant value, whose magnitude corresponds to the radius. For an ellipse, the signature is undulating showing two peaks. For a square, there are four peaks. For a rectangle the signature is a mixture of the ellipse signature and the square signature. Figure 40 shows the shape for a complex shape, a pigeon shape of the MPEG7 database.

There are two relatively straight-forward analyses we can do with the radial signature. One is a Fourier analysis, meaning we express the signature as a spectrum of frequencies. The other analysis is an investigation of the extrema present in the signature, as we did with the face profile (Section 5.3).

Fourier Analysis The Fourier analysis transforms a signal into a spectrum, which in digital implementation is a sequence of so-called Fourier descriptors (FD). This is an enormously powerful analysis, which merits its own lecture, but is rather the topic of a signal processing course. We apply this discrete Fourier transform to the (unmodified) radial signature R, called `Rad` here, and normalize the transform's output by its first value:

```
FDabs    = abs(fft(Rad));         % fast Fourier
FDn      = FDabs(2:end)/FDabs(1); % normalization by 1st FD
```

In the lower right of Fig. 40, the first 50 Fourier descriptors are shown. But typically, the first 5 to 10 Fourier descriptors are sufficient for discriminating the shapes.

Extrema Analysis Finding extrema in a signature is easier if we first low-pass filter the signature, as we did for profile analysis of a face, see Section 5.3. The number of maxima corresponds to the number of corners in a shape, whereby here corner means a curvature higher than its context. Two corners would correspond to an ellipse or bicorn shape, three corners to a triangle or trident shape, etc.

For really complex shapes, the radial profile fails to be specific for boundary segments that lie radial. This can be seen on the pigeon shape in Fig. 40: the signature has three major peaks that represent the head,

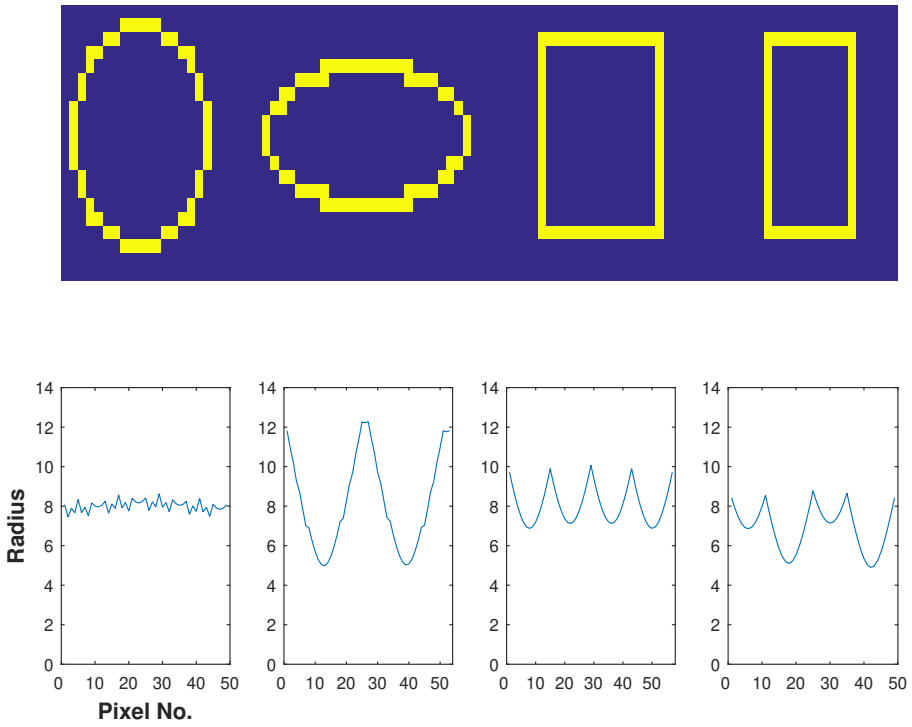

Figure 39: Radial signature for four simple shapes. This is an efficient way of representation and can easily discriminate between some shapes, e.g. the ones shown in row **a** of Fig. 38, but struggles already with shapes in row **b** of same figure.
Upper Panel: circle, ellipse, square and rectangle (shapes traced with `bwboundary`; displayed distorted unfortunately [unequal axis lengths]).
Lower Panel: radial signature: the distances (radii) between pole and the individual boundary points. The x-axis describes pixel number, or more precisely expressed *arc length*, the cumulative length integral of the contour.

the tail and the feet. The subtle boundary changes between those major structures are still reflected in the signature, but are difficult to measure. If one intended to capture those subtleties, one had to work perhaps with multiple poles, one could call them subpoles. Or one used a contour description method (Section 17).

Appendix **P.13.3** shows an example of how to generate the radial and Fourier descriptors. The parameter output of the Fourier analysis is more discriminative than the output of the extrema analysis in general, but combining both can yield even better results. The presented descriptions achieve satisfactory results on the MPEG7 shape data set (see introduction of this section); satisfactory in the sense that they are relatively efficient due to their relative compactness, the few number of parameters.

If one intended to carry out retrieval only (and not classification), then one could also try matching the individual signatures point-wise. This is the topic of the next subsection.

16.3 Point-Wise

There are two cases of point-wise formats one can distinguish. In one format, a shape is expressed as a single boundary, a sequence of points, which typically corresponds to its silhouette (Section 16.3.1). In the other format, a shape is considered as a set of points (Section 16.3.2). For both we can apply for example the Hausdorff distance (mentioned in Section 5 already). Because the shape comparison with such formats is relatively time-consuming, one prefers too know the approximate alignment between the two shapes before an accurate similarity is determined. This is also known as the *correspondence problem*, meaning which points in one shape correspond to some points in another shape, at least in an approximate sense (Appendix E).

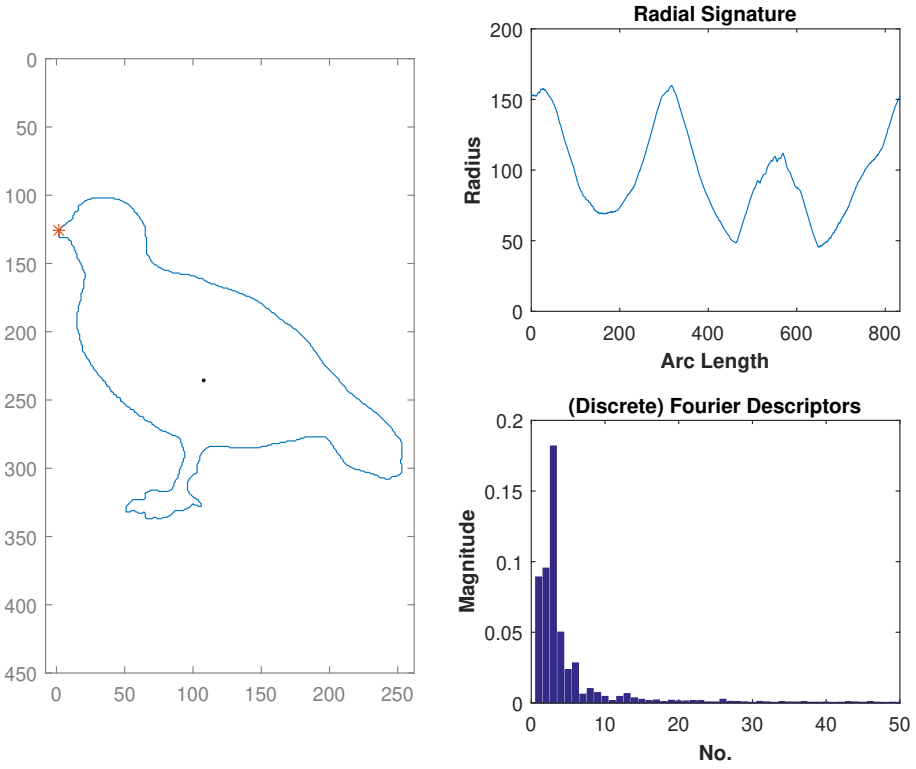

Figure 40: **Left:** boundary of a pigeon shape; the black dot inside the shape is the (calculated) *pole* of the shape; the asterisk on the boundary marks the starting pixel of the boundary.
Upper Right: radial signature: the distances (radii) between pole and the individual shape points. The x-axis describes pixel number, or more precisely expressed *arc length*, the cumulative length integral of the contour.
Lower Right: discrete Fourier descriptors of the radial signature. The x-axis describes the number of the Fourier descriptor. An example code can be found in Appendix **P.13.3**.

16.3.1 Boundaries

In this case, the list of points of one shape, are compared to the list of points of another shape, by somehow determining a distance (or similarity) measure between the two using each pixel of each shape. The simplest way would be to take the pairwise distances between the two point lists and to sum the corresponding minima to arrive at a measure of similarity. The pairwise pixel matching is computationally costly and the computational complexity is said to be square, expressed also as $O(N^2)$, where N is the number of pixels and O is the symbol for complexity. And to solve the correspondence problem one could simply shift the two shapes against each other to find the absolute minimum. That shift would increase the complexity to $O(N^3)$, that is, the matching is now cubic and thus rather impractical as such. There are several optimizations one can think of. A simple one would be to use *landmarks* or *key-points* to determine the shape correspondence. Such key-points are typically points of high curvature in a shape, e.g. we can use the radial, the curvature or the amplitude signatures, see Sections 16.2 and 17.5. The problem is that such key-points are difficult to determine consistently.

The most efficient boundary matching technique is based on observing the local orientations along the boundary and including them in the matching process; on the MPEG7 dataset it is perhaps the most efficient technique considering the speed-accuracy tradeoff. The detailed steps are as follows:

1. Sample an equal number of points $i = 1, .., N$ from each shape, equally spaced along the boundary.
2. Determine the local orientation o at each point, for instance the angle of the segment spanning several pixels on both sides of the center pixel. Thus the shape is described by a list of N points with three values per point: x- and y-coordinate, as well as orientation o.

3. Determine the farthest point using the radial description. This will serve as a correspondence.

When matching two shapes, one would take the point-wise distances (including orientation o) using the farthest point as reference. The point-wise distances are also taken in reverse order to account for asymmetric shapes. Thus, the matching complexity is $O(2N)$ only.

16.3.2 Sets of Points

Here again we can distinguish between two cases: 1) the shape can consist of multiple boundaries - and not only of one as assumed above; 2) the shapes of one class have limited intra-class variability and consist of few points.

1) Multiple Boundaries The most successful approach is called *Shape Context* (Belongie, Malik & Puzicha, 2002) and is based on taking local radial histograms at selected points of the shape. The selection of such key-points may not be completely consistent, but that would be compensated by a flexible matching procedure. At each key-point, a circular neighborhood of points is selected and a one-dimensional histogram is generated counting the number of ON pixels as a function of radial distance.

2) Limited Variability; Few Points This case is rather useful for localization and less for retrieval (or classification). We assume that we know the shape's key-points and that its articulation is limited (see again property list given in the introduction). The goal is then to find the target shape in another image. We are then faced with two tasks: the correspondence problem and the transformation problem. The transformation issue will be introduced in Section 21.

16.4 Toward Parts: Distance Transform & Skeleton

Because we humans interpret a shape as an alignment of segments or parts, it was assumed from the early days of computer vision, that a shape description should also consist of segments or parts somehow. Decades of trial-and-error has taught computer vision scientists, that such a description is difficult to achieve. What exactly is supposed to be a segment or a part and how those should be represented, are still unsolved problems. But if one intended to work toward that direction, then the distance transform could be a piece of the puzzle, because it appears a 'natural' step toward extracting parts. After we introduced that distance transform (Section 16.4.1), we mention the difficulties with obtaining segments and parts (Section 16.4.2).

16.4.1 Distance Transform

Sze p113, s 3.3.3, pdf 129
Dav p240, s 9.5
SHB p19, s 2.3.1

The distance transform is typically determined for a binary image. The transform calculates at each background (off) pixel the distance to the nearest object (on) pixel. This results in a scalar field called *distance map* $D(i, j)$. The distance map looks like a landscape observed in 3D, which is illustrated in Figure 41. The distance values inside a rectangular shape form a roof-like shape, a *chamfer* (shapes used in woodworking and industrial design); the interior of a circle looks like a cone. The distance transform is also sometimes known as the *grassfire transform* or symmetric-axis transform, since it can also be thought of a propagation process, namely a fire front that marches forward until it is canceled out by an oncoming fire front. Wherever such fronts meet, that is where they form symmetric points, which correspond to the ridges in the distance map: it is a roof-like skeleton for the rectangle and a single symmetric point for a circle - the peak of the cone. Sometimes that skeleton is also called *medial axis*.

The distance transform can be calculated with different degrees of precision. Simpler implementations provide less precision but calculate quicker; they are based on the Manhattan distance for instance. Precise implementations provide the Euclidean distance.

In Matlab the function `bwdist` calculates the distance values for OFF pixels, meaning ON pixels are understood as boundaries (as introduced above). It calculates Euclidean distances by default, but a simpler implementation can be specified as option.

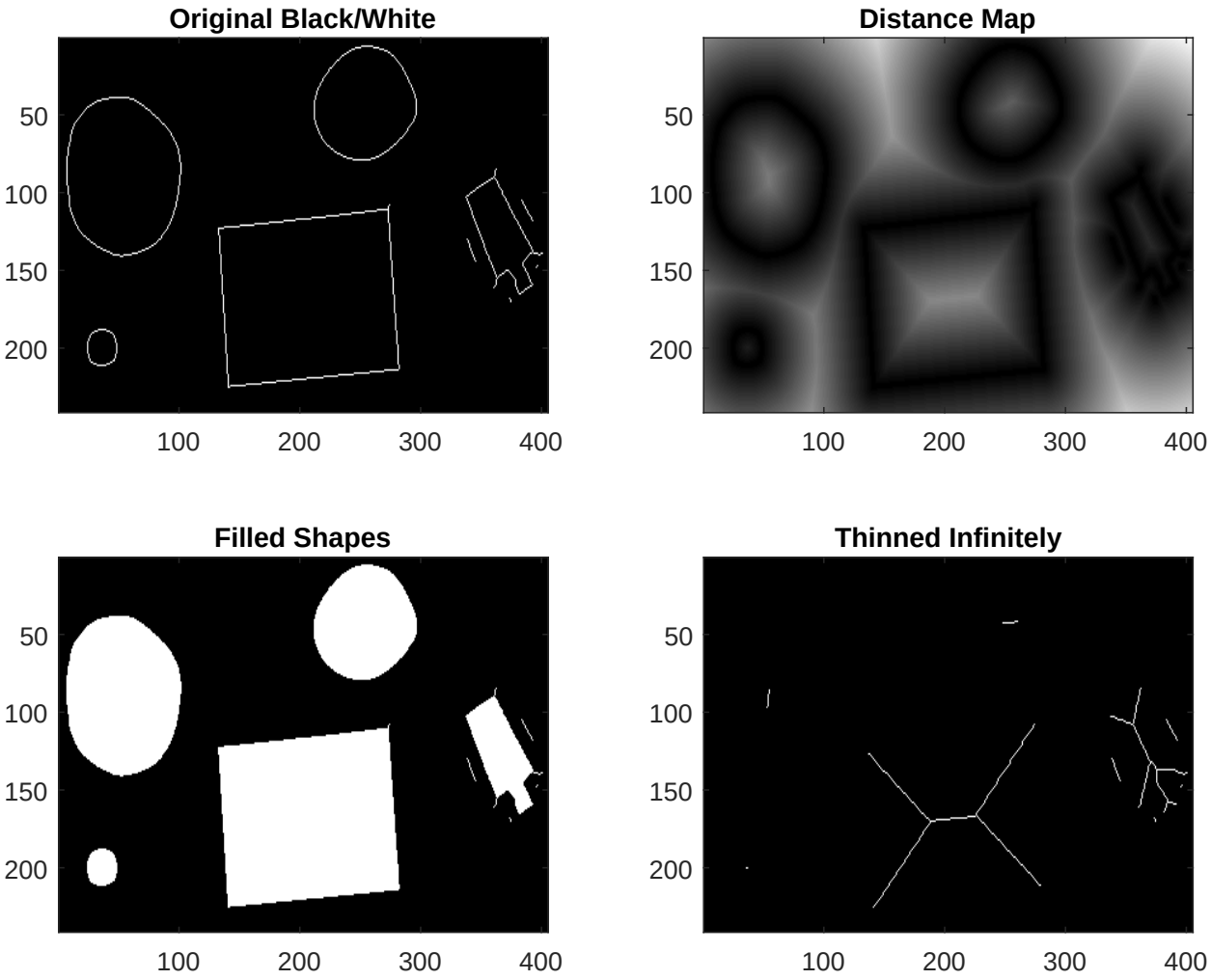

Figure 41: Distance map and skeleton.
Upper Left: input image: a set of boundaries.
Upper Right: its distance map obtained with `bwdist`.
Lower Left: Filled shapes of the original image.
Lower Right: quasi skeletons of the shapes, obtained using the 'thin' option of `bwmorph`.

```
DM  = bwdist(BW);                    % distance calculated at OFF pixels
```

In Python the distance transform can be found in the `scipy.ndimage` module, specifically `scipy.ndimage.distance_transfo` for Euclidean precision. It calculates the distances at ON pixels (!), not at OFF pixels as in Matlab. Simpler implementations are provided by separate function scripts, ie. `distance_transform_cdt`.

```
from scipy.ndimage import distance_transform_edt
DM  = distance_transform_edt(BW)        # distance calculated at ON pixels
```

In OpenCV [py] there is one function, for which one specifies with parameters the type of desired implementation. The distance is calculated at ON pixels as in Python (not as in Matlab).

```
DM = cv2.distanceTransform(BW.astype(uint8), distanceType=cv2.DIST_L2, maskSize=cv2.DIST_MASK_PRECISE)
```

Applications: the transform tends to be used in Machine Vision applications, where the number of shapes is limited, such as in binary image alignment (fast chamfer matching; see again Section 5), nearest point alignment for range data merging, level set; or in computer graphics, feathering in image stitching and blending.

16.4.2 Symmetric Axes (Medial Axes), Skeleton

The symmetric axes are the ridges in the distance map, the contours that look like veins (upper right in Figure 41). Extracting those coherently is rather difficult and there exist a variety algorithms, each one with advantages and disadvantages. One is a thinning algorithm, similar to the erosion operation for morphological processing (Section 15.1), which when carried out infinitely results in a skeleton resembling the sym-axes, see lower right in Figure 41 whereas the starting point are the filled shapes (lower left in figure). Such skeletons are then fragmented and a shape is expressed as a structural description, a description by parts in a certain alignment.

In Matlab implementations of axes and skeletons are accessed throught the function script `bwmorph`:

```
BWs  = bwmorph(BW,'skel',Inf);
BWs  = bwmorph(BW,'thin',Inf);
```

In Python there exist separate function scripts in module `skimage.morphology`

```
BWs  = medial_axis(BW)
BWs  = skeletonize(BW)
```

In OpenCV [py] available in module `ximgproc` as `thinning`. There exist two variants, specified with constants `THINNING_ZHANGSUEN` and `THINNING_GUOHALL`.

The differences between those implementations are subtle and it is difficult to foresee when what implementation is more appropriate for a specific task. One general difficulty with any of those implementations is that they are limitedly robust to subtle variability (see table 16 again). In particular, they are limitedly invariant to articulation and deformation, but even for rotation of the same shape, the output of an algorithm may change.

16.5 Classification

When it comes to classification, the most successful systems are not the above introduced techniques, but Deep Neural Networks as introduced in Section 8. Even for the digit database MNIST, the use of symmetric axes for instance, has not provided better prediction accuracies than those Deep Nets, see the list of accuracies on

`http://yann.lecun.com/exdb/mnist/`.

There are two disadvantages with CNNs for shape recognition. One is, that they require the shape to be fairly well centered in the image; thus, a search algorithm is necessary that finds the exact shape center. The other disadvantage is, that they require fairly long to learn the features, as pointed out already in Section 8, though with transfer learning that short coming is almost eliminated.

The power of those networks comes from their robustness to local changes: small changes in the boundary do have little consequences in networks. In contrast, for any of the shape description techniques introduced above, such small changes can result in relatively different features and thus tendentially more wrong classifications than with DNNs. Take for instance the two rectangular shapes in Figure 41: the corresponding skeletons in the lower right graph, show sufficient differences that make a robust comparison difficult.

16.6 Literature

Surveys on shape tend to be older, as the focus lay on Deep Learning in the past decade:

Yang, Kpalma and Ronsin, 2008
https://hal.archives-ouvertes.fr/hal-00446037/file/ARS-Journal-SurveyPatternRecognition.pdf

Zhang and Lu, 2004. Review of shape representation and description techniques.
https://www.sciencedirect.com/science/article/abs/pii/S0031320303002759

17 Contour

Contours delineate the structure in a scene or object. In a scene, contours delineate object silhouettes or borders of adjacent scene parts. In an object, the contours delineate its parts. Contours can be obtained from different processes. One is by a region segmentation process, in which case we often obtain also many small region shapes that can be conveniently described with the techniques introduced in the previous section. But region segmentation does not capture all contours, in particular not those of texture or of thin, elongated structures. For example an image of an agricultural field seen from a satellite is best analyzed by contour detection techniques (see Fig. 42). It is therefore desirable to develop contour detection techniques that are able to find those directly.

Such contours can be detected by various techniques. One technique was introduced in Section 7.2, and concerned in particular the detection of edges, a cliff or steep slope in the intensity landscape. But an intensity landscape consists of more topological features, such as ridges and rivers, see again Fig. 12. A landscape can also be described by iso-contours as it is done in geological maps. Those types will be introduced in Section 17.2. Having detected contour pixels, we then want to connected them to entire segments, a process that is called *contour following* or *contour tracing*, analogous to boundary following (or tracing) for regions (Section 15.3).

After we have traced the contour segments, we wish to characterize them, for example how straight or curved they are. A simple description is given in Section 17.4, which is sufficient for many contour segments in a gray-scale image, in particular short ones and long, straight ones. For longer segments with complicated geometry, it requires a more complex description, explained in Section 17.5. All these processes are necessary for a precise localization and understanding of the structure within an image. If one however intends to detect only straight lines and circles in an image - without regard for the rest of the contours -, then one can resort to dedicated algorithms. We introduce those first (Section 17.1).

17.1 Straight Lines, Circles

If an application requires the extraction of straight lines in particular, then there exist some dedicated techniques to find them. The most popular one is the Hough transform, which places straight line equations on each point of the diagonal toward all directions. It is typically run on a black-white image, i.e. an edge map as obtained in Section 7.2 or coming up below in Section 17.2.

The transform returns a matrix whose axes correspond to two variables: one is the distance ρ of the straight line from the image center; and the other is the angle θ. The matrix is a 2D histogram, called an *accumulator* here. Its maximum value corresponds to the longest, straightest line in the image. For both variables, ρ and θ, the bin size needs to be specified, which requires some tuning by the user.

Application: Straight lines are used to find vanishing points in road scenes for instance (Figure 65). Or they are used for calibration of internal and external camera parameters. The Hough transform can also be modified to detect circles. There exists also a modification that extends the transform to ellipse detection. The transform can be regarded as a grouping process, a topic that we will introduce in a later section (24). The transform has the advantage that it is relatively robust to contour fragmentation, but it is also relatively complex process in comparison to the grouping processes introduced in Section 24.

In Matlab the transform is implemented with function scripts called `hough`, `houghpeaks` and `houghlines`. With `BW` our edge map, the function `hough` returns the accumulator in `AC`, and the edge values used for the histogram in variables `The` and `Rho`:

```
[AC The Rho]= hough(BW);            % AC=[rho theta]
Pek         = houghpeaks(AC, nLin); % [nPek 2] row/column
aLines      = houghlines(BW, The, Rho, Pek, 'FillGap',5, 'MinLength',minLen);
```

With function `houghpeaks` we find the peaks within the accumulator and here we limit it to be `nLin` lines. The function carries out a search procedure like any other search procedure such as in non-maxima suppression (Section 11.2) and it therefore returns the coordinates for the peaks within the accumulator. With that information we apply function `houghlines`, which returns the detected line segments. A full example is given in **P.14.1**.

For circles one can use `imfindcircles`.

In Python the functions reside in module `skimage.transform` as `hough_xxx`.

In OpenCV [py] under `cv2.HoughXXX` and also under `ximgproc` as `FastHoughTransform` and there exists also the function `ximgproc.HoughPoint2Line`.

17.2 Contour Types

In the early section on feature detection, we had introduced a contour detection technique that focused on edges (Section 7.2). That type of contour is very informative in many scenes, but does not do well in other scenes. In particular when an object is long and thin. If that elongated object is brighter than its surround, then it will appear like a ridge in the intensity map and we detect it with a technique that identifies the ridge line. Conversely, if that elongated object is darker than its surround, then it will appear like a river in the intensity map. Figure 42 shows an example where river contours capture the scene structure better than other contour types. In some other scenes, iso-contours are more suitable.

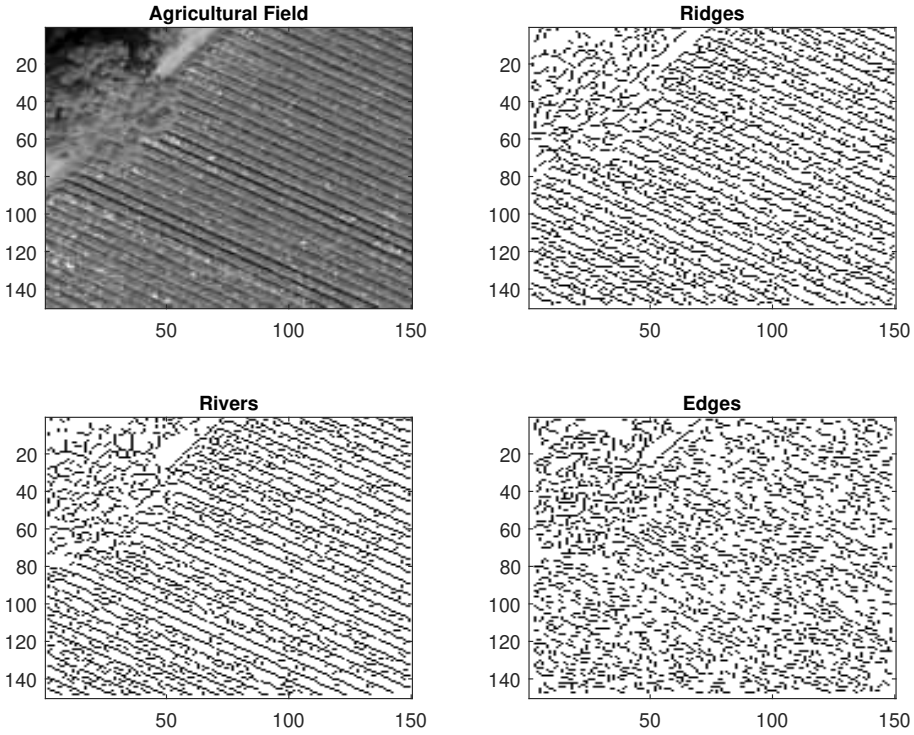

Figure 42: Ridge, river and edge contours of a satellite image. In this example river contours provide the most continuous segments; ridge segments are more fragmented, edge contours are detected only as short segments.

17.2.1 Ridge and River

Ridge and river pixels represent extrema in the intensity landscape. They can therefore be detected with a very simple technique that observes only relational operations between neighboring pixels. At a given pixel, we analyze its eight neighbors along the four principal orientations, horizontal, vertical and the two diagonals. We take the following neighborhood as an example and determine whether the center pixel (value 7) is a ridge pixel,

$$\begin{bmatrix} 3 & 4 & 6 \\ 5 & \mathbf{7} & 4 \\ 8 & 6 & 2 \end{bmatrix} \tag{16}$$

by observing its two neighbors along the four principal orientations:

First diagonal, [3 7 2]: the center value is larger then both of its neighboring values.
Vertical axis, [4 7 6]: the center value is larger than both of its neighbors.
Second diagonal, [8 7 6]: the center value is *not* larger than both neighbors.
Horizontal axis, [5 7 4]: the center pixel is larger than its two neighboring values.

For this neighborhood we have found three maxima and the pixel can be considered a ridge pixel, running along the second diagonal.

One can apply the same technique to count minima and that would then be an analysis of river pixel candidates. We can deploy this technique to detect also edge contours: firstly, we generate a range image of the intensity map and then observe the ridges in that range image. An example of this technique is given in **P.14.2**. In Fig. 42 the output of that technique is shown, where river contours provide the most continuous segments for a photograph of an agricultural field. Ridge contours have also been used for finger print identification for example.

More background on those types of contours can be found in
https://www.researchgate.net/publication/321218540_Rapid_Contour_Detection_for_Image_Classification

17.2.2 Iso-contour

Iso-contours correspond to the lines of equal height in a topological map: an iso-contour has the same intensity value for each of its pixels. Examples of those are shown on the right in Fig. 43. Iso-contours have been used for nuclei detection and classification in histological images.

In Matlab isocontours can be detected with the function `contourc`,

```
A = contourc(double(Img), double(LevInp));    % [2 ntPts]
```

for which one specifies the levels `LevInp` at which the image is thresholded. The function returns a single array for all iso-contours of all levels, consisting of two rows for row (y) and column (x) coordinates. The contour beginnings a marked by one entry that contains the level and the number of pixels:

```
z_level     = A(1,ix);
nP          = A(2,ix);          % # of pts of that isoline
```

For the very first contour we find those values at `ix=1`; for the second contour we find them at `ix=A(2,1)+1`, etc.

In Python we can use the function `find_contours` in module `skimage.measure`. It however only processes one level, unlike in Matlab, and one therefore needs to write a loop to obtain iso-contours from different intensity levels.

One specifies a height value at which the map is thresholded and then the corresponding region boundaries are taken. This can be exploited to emulate boundary following of connected components in binary images (Section 15.3.2). If one specifies a value of 0.8 (between 0.5 and 1.0), then the contours lie closer to the region pixels; if one specifies a value of 0.2 (between zero and 0.5), then the contours lie closer to the adjacent exteriors pixels of the region. The example in Section **P.14.3** demonstrates that.

In OpenCV [py] we can use `cv2.findContours`, which takes a binary image as input. We therefore need to threshold the intensity map ourselves before feeding it to the function.

17.3 Contour Following (Curve Tracing)

Dav p257, s 9.8

Contour following is the process of connecting the detected contour pixels to entire segments. A segment is then expressed as a list of adjacent pixel coordinates. This process is also called *tracing* or *tracking* sometimes. In principle one could use the boundary following algorithms introduced previously (Section 15.3.2). The boundary following algorithm treats the contour segment as a region and returns the coordinates twice

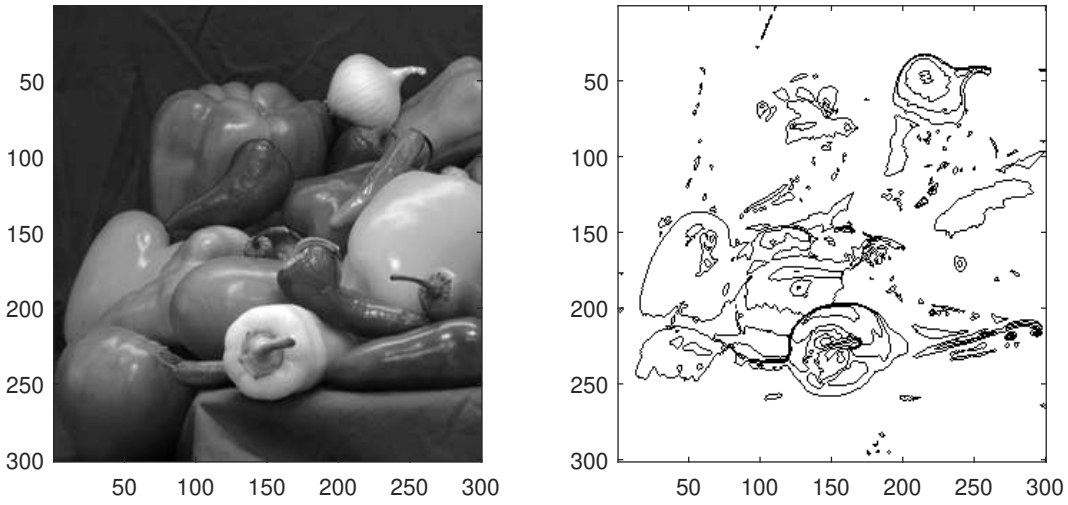

Figure 43: Iso-contours. The intensity image is thresholded at varying values and the resulting boundaries are traced. These contours capture well large-area mounds and basins.

as it goes around the contour completely. One therefore had to determine the two endpoint pixels in the obtained list of coordinates.

A more direct approach would be to trace the contours starting from their ends and to stop tracing at the other end, without returning as in the boundary following approach. There are two issues to consider with this approach. One is that the detected contours are occasionally broader than one pixel, which can make tracing difficult, because such clusters represent ambiguous situations. Those clusters occur in particular when the contour runs along the diagonal axes. To avoid such thickened contours, one can employ the operations of morphological processing as introduced previously, binary morphology in particular (Section 15.1). With the so-called *thinning* operation we can 'slim' those thick contour locations. Or perhaps we wish to remove isolated pixels immediately by using an operation *clean*:

```
Medg = edge(I, 'canny', [], 1);
Medg = bwmorph(Medg,'clean');   % removes isoloated pixels
Medg = bwmorph(Medg,'thin');    % turns 'thick' contours into 1-pixel-wide contours
```

This is however relatively costly, as we scan the entire image multiple times. Alternatively, and less costly, one would make the contour tracing algorithm more robust to ambiguous situations by including walking direction to 'stumble' across those.

Another issue to consider, is that we need to decide how to deal with junctions. Let us take a T-junction as an example: should we break it up at its branching point and trace the three segments individually? Or should trace around it - and obtain a boundary - which we later partition into appropriate segments? This leads to the question of representation, which has still not been properly solved.

In Matlab we can detect endpoint and branchpoints through the function script `bwmorph`, i.e.

```
Mept = bwmorph(Medg,'endpoints');
Mbnh = bwmoprh(Medg,'branchpoints');
```

The function script `bwtraceboundary` traces a thinned contour from start to end without returning to start, if the contour is thinned. For that one specifies a starting point by its coordinates. Hence, we write a loop which loops through the detected endpoints of `Mept`.

In Python there exist no equivalent routines to our knowledge, at least not in the commonly known module.

We place the traced coordinates into an array, i.e. `Bon` of size `[nPts 2]`, where first column holds the row indices and the second column the column indices. For later convenience we add the inter-point spacing as a third column, which in Matlab we can calculate using `diff`:

```
Bon    = [Bon [0; sqrt(sum(diff(Bon,1).^2,2))]];  % adding inter-point spacing as 3rd column
```

17.4 Basic Description

Many contours in gray-scale images are short segments made of only few pixels. It suffices to describe them by a pair of parameters, such as the degree of straightness and its contour length. Even for longer contours this can be a resonable abstraction to start with. Figure 44a shows the keypoints that are useful to characterize the geometry. The contour length is calculated as the sum of all inter-pixel distances,

```
len    = sum(Bon(:,3));  % arc length of contour/boundary
```

also called the total arc length. In case of contour pixels described by integer values, the inter-pixel distances show only values 1 or $\sqrt{2}$. The chord length is the distance between the contour's endpoints, `ep1` and `ep2`:

```
lch   = sqrt(sum((ep1-ep2).^2));
```

A simple form of measuring the straightness of a contour is to take the ratio between the chord length and the segment length.

```
str    = lch/len;  % straightness [0 1]
```

A value of 1 means the segment is completely straight; a value of 0 means it is closed and the contour would form a shape. We can gain some more geometric information by measuring the distance between the mid- and halfpoint as well as the orientation angle of straight line running through those two points.

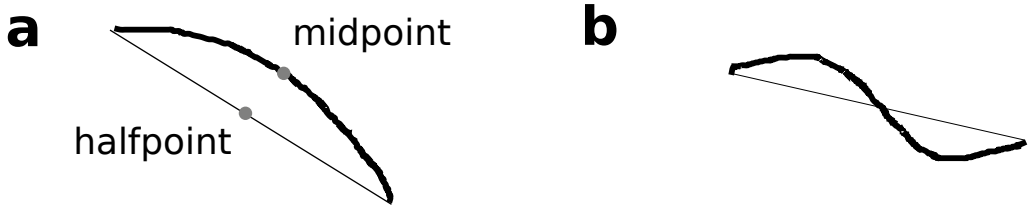

Figure 44: Basic contour (segment) description. The thin line represents the chord, the straight line connecting the segment's endpoints. Contour length is the sum of inter-pixel distances, also called the total arc length.
a. an almost smooth arc. Midpoint: the center pixel of the contour coordinates. Halfpoint: the center point of the chord.
b. contour describes an inflexion. Pixels lie on both sides of the chord.

A more specific geometric characterization requires more complex computation and we need to calculate the distances between the contour pixels and some reference point. In case of elongated segments, one needs to know on which side of the chord the pixels lie, see Fig. 44b, thus we calculate N distances (N the number of pixels). In case of a closed segment we would use the radial signature (Section 16.2). For cases between those two extremes, it requires a systematic description by a space of signatures, coming up next.

17.5 Description by Signatures

To properly describe an open or closed contour it is necessary to move a window through the contour and take some measure of the window's subsegment. This results in a signature analogous to the radial signature introduced for shapes 16.2. There are two types of contour signatures that have been pursued so far: curvature and amplitude.

17.5.1 Curvature

The curvature measure is calculated by taking the derivatives along the contour, pixel-to-pixel. The second derivative is a reasonable measure already, but there exist more precise definitions.

The derivatives are firstly taken along the individual coordinates, `Rf` and `Cf` (row and column indices), and then combined. In case of the simple measure we merely add the two second derivatives; in case of the complex measure, we combine first and second derivatives as given in the code fragment below:

```
%% =====    Derivatives      =======
Y1      = [0; diff(Rf)];        % 1st derivative
X1      = [0; diff(Cf)];
Y2      = [0; diff(Y1)];        % 2nd derivative
X2      = [0; diff(X1)];

%% =====    Curvature Simple    =====
Drv2    = Y2+X2;                    % adding 1st derivatives

%% =====    Curvature Accurate  ======
K       = (X1.*Y2 - Y1.*X2) ./ ( (X1.^2 + Y1.^2).^1.5);
```

Peaks in such a signature represent *points of highest curvature* (Fig. 45, upper right). If there is no peak, then we deal with a perfectly smooth arc or a circle. If the coordinates are integer values, then it is better to firstly smoothen the coordinate values. Appendix **P.14.4** demonstrates a complete example. The signature is plotted against arc length, the cumulative integral of the interpoint spacing.

Those high-curvature points are sometimes used for shape finding and identification. The shortcoming with this signature is that it is suitable only for a certain scale: we are not able to detect all points of highest curvature in arbitrarily sized contours. Broad curvatures are easier detected when the signature has been smoothened with a correspondigly large low-pass filter. In order to capture all potential curvatures, one therefore generates a space, the *curvature-scale space* (CSS). The space is created by low-pass filtering the curve for a range of sigmas, analogous to the image scale space as introduced in Section 6.1. The curvature scale space is part of the MPEG7 definition. On the MPEG7 shape data set (Section 16), the curvature scale space achieves good results using a technique as mentioned in Section 16.3.1, but the matching duration is much longer than that for radial descriptions.

17.5.2 Amplitude

To obtain an amplitude signature, we move a window through the contour and measure the displacement between pixels and the chord - as we did for the entire contour (Section 17.4). A window selects a sequence of pixels, a subsegment, and the maximum distance (displacement) between the subsegment and its chord is taken, the amplitude. As this window is moved through the contour, it creates an amplitude signature, see Fig. 45 (lower right). This is computationally more intensive than taking a curvature measure, but it is also more accurate, as it does not modify the signal by any filtering process. As can be seen in the amplitude signature, it maintains detail in particular around the cupolas of the signature, as it does not involve any filtering.

Analogous to the curvature signature, the amplitude signature can only describe a contour properly by using a range of window sizes. This results in an amplitude space, that allows a more reliable detection of points of highest curvature. A full method is described in

https://www.researchgate.net/publication/323541525

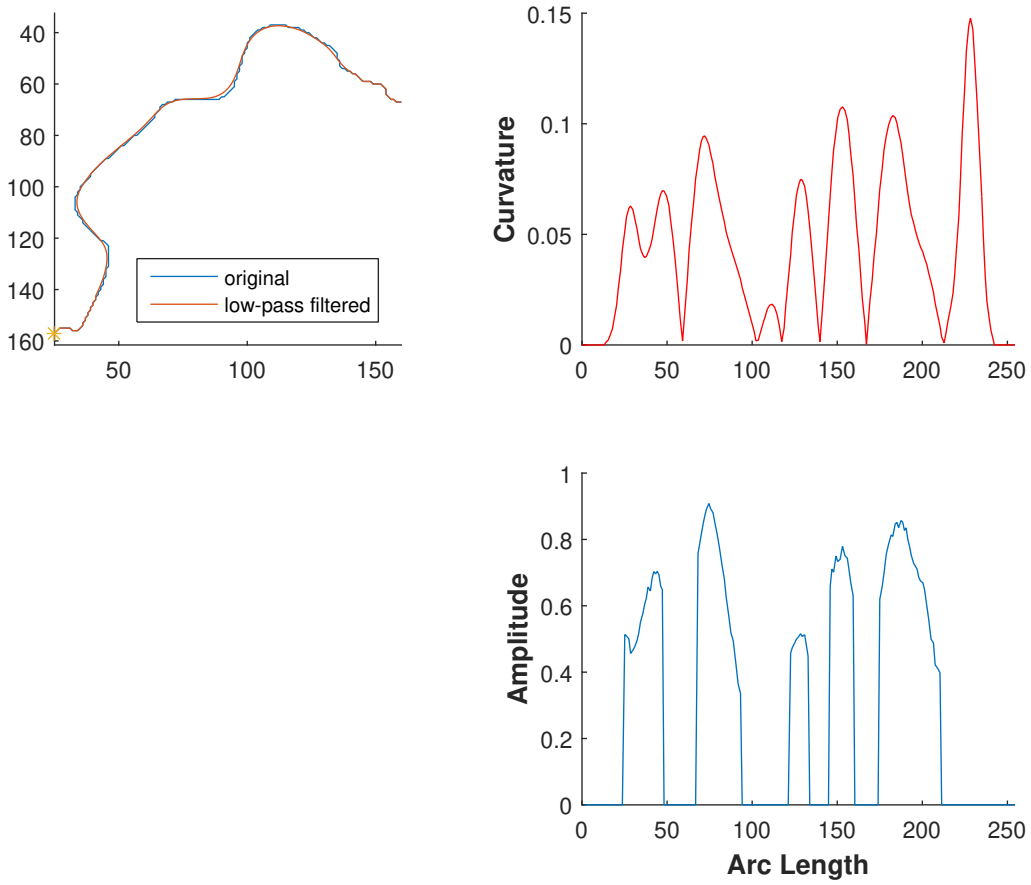

Figure 45: Signatures for contour description.
Upper Left: example contour. The asterisk marks the starting point for the signature.
Upper Right: signature based on differentiation along coordinates.
Lower Right: signature based on determining the maximal distance (amplitude) between chord and subsegment.

18 Image Search & Retrieval

An image retrieval system is a computer system for browsing, searching and retrieving images from a large database of digital images, as introduced in Section 5.1.1. Browsing, searching and retrieving are search processes of increasing specificity:

- browsing: the user looks through a set of images to see what is interesting.
- searching: the user describes what he wants, called a *query*, and then receives a set of images in response.
- retrieval: the user uploads an image and in return obtains the most similar images, ranked by some similarity measure.

Traditionally, methods of image retrieval utilized metadata such as captioning, keywords, or (textual) descriptions to find similar images, that is they would not use any computer vision. Then, with increasingly powerful computer vision techniques, the actual 'pixel content' was analyzed too, using simple image histogramming in the early days (mentioned in Section 4.1 already). This new approach was then called *content-based image retrieval* (CBIR). For some time, CBIR was based on (engineered) features as introduced in early sections (Section 7), but modern CBIR uses also Deep Nets of course.

In image retrieval, images are selected by some measure of similiarity, they are not classified as in image classification, see again Section 2 to understand the difference or the introduction on shapes in Section 16. To retrieve an image, a similiarity between the query image and all other images is calculated, and then the images are ranked according to that similarity measure, or dissimilarity measure in our introductory example in Section 5.1.1. The similarity is measured with image vectors, obtained with the techniques as introduced in previous sections; it is the same principal as introduced in Section 3.3, but instead of classifying the image vectors, we sort them. We repeat in Section 18.1 how the image vectors can be obtained.

As with shape retrieval, in image retrieval too, one is concerned with the computational burden of the retrieval process. If our database is small, we can take elaborate feature descriptor vectors that yield high retrieval accuracy, and we can calculate the distances between all image vectors beforehand (off-line). If our database is large, we are forced to make some simplifications; for example for browsing vast amounts of images from the internet, we are forced to make the retrieval process a simple one. Retrieval methods can be tested on the following database for instance:

http://horatio.cs.nyu.edu/mit/tiny/data/index.html

In the following we firstly explain how to perform retrieval in general (Section 18.1). Then we introduce how the ranking success is quantified (Section 18.2). Finally, we mention that there exist a set of particular tricks for doing retrieval with local features (Section 18.3). But before that, we give examples of specific applications.

Applications

Finding Near Duplicates

1) Trademark registration: A trademark needs to be unique, and a user who is trying to register a trademark can search for other similar trademarks that are already registered (see Figure 46 below).
2) Copyright protection.

Semantic Searches: Other applications require more complex search criteria. For example, a *stock photo library* is a commercial library that survives by selling the rights to use particular images. An automatic method for conducting such searches will need quite a deep understanding of the query and of the images in the collection.

Trends and Browsing: In data mining, one uses simple statistical analyses on large datasets to spot trends. Such explorations can suggest genuinely useful or novel hypotheses that can be checked by domain experts. Good methods for exposing the contents of images to data mining methods would find many applications. For example, we might data mine satellite imagery of the earth to answer questions like: how far does urban sprawl extend?; what acreage is under crops?; how large will the maize crop be?; how much rain forest is left?; and so on. Similarly, we might data mine medical imagery to try and find visual cues to long-term treatment outcomes.

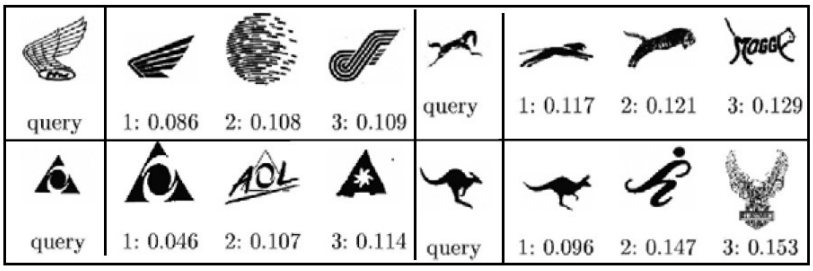

Figure 46: A trademark identifies a brand; customers should find it unique and special. This means that, when one registers a trademark, it is a good idea to know what other similar trademarks exist. The appropriate notion of similarity is a near duplicate. Here we show results from Belongie et al. (2002), who used a shape-based similarity system to identify trademarks in a collection of 300 that were similar to a query (the system mentioned in Section 16.3.2). The figure shown below each response is a distance (i.e., smaller is more similar). *This figure was originally published as Figure 12 of Shape matching and object recognition using shape contexts, by S. Belongie, J. Malik, and J. Puzicha, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2002, ©IEEE, 2002.*

18.1 Image Vector, Distance and Similarity

To perform retrieval, the images are firstly described by one of the feature description methods as introduced in previous sections (Section 3.3). If we deal with color images and if the image collection is not massive, it is perhaps most convenient to use one of the pretrained Deep Nets and then use the output of the last hidden layer as an image vector (Section 18.1.1). If we deal with massive image sets, then retrieval of words is inevitable (Section 18.1.2). If we deal with only textures, perhaps a histogram of Local Binary Pattern may be sufficient (Section 7.3.3). If we deal with shapes, then we can consider shape descriptors as discussed in Section 16.2 for instance.

18.1.1 Deep Net

To obtain the image vector from a Deep Net we firstly initialize a network as if we intended to train a network with pretrained weights (Section 8.2). But here we set the network immediately to evaluation mode, because we use it merely as a fixed feature extractor:

```
NET     = models.resnet50(pretrained=True)  # pretrained weights
NET.eval()
```

By doing so, the last layer, the output layer, is set to 1000 units corresponding to the 1000 classes on which the network was trained. That output layer is irrelevant for our purpose and instead we will use the second last layer, the last *hidden* layer, as a feature vector. Depending on the network type, this last hidden layer holds several hundreds to several thousands units. In order to access those values, we need to instruct the network to store them, because by default it does not keep those values for reason of efficiency. To store those values, we need to define a so-called *hook*:

```
hidImg = [None]
def hookHid(module, input, output):
    hidImg[0] = output
```

After having defined the hook, we register it with the last hidden layer, here shown for the ResNet:

```
NET.avgpool.register_forward_hook(hookHid)              # resnet
```

Then we feed an image to the network and extract the feature values of that last hidden layer, which will serve as our feature vector $\mathbf{f}$. A full example is given in **P.15.1**.

18.1.2 Bag of Features

This method performs retrieval with local features, such as the gradient-based features introduced in Section 12 or the Local-Binary Patterns (Section 7.3.3). The matching principal was demonstrated in Section 12.4, see code example **P.9.3**. But one usually applies the method of quantization to such features (Section 13) and the resulting histograms of words are our feature vectors $\mathbf{f}$ for retrieval. Here we mention that Matlab offers functions performing image retrieval with all tricks applied.

The easiest way to perform retrieval in Matlab is by the use of the two functions `indexImages` and `retrieveImages`. The function `indexImages` takes a list of images and creates dictionary, a bag of features (words), which we then apply with `retrieveImages`. With function `evaluateImageRetrieval` we can evaluate the retrieval result.

If we wish more control over individual processes, then we would use function `bagOfFeatures`, which allows one to perform retrieval with any type of feature.

18.1.3 Distance, Similarity

For any choice of feature description, we end up with a feature vector, $\mathbf{f}_q$, that we compare with the feature vectors of all other images, $\mathbf{f}_i$. The most convenient comparison is by use of some distance metric. The Manhattan distance would be the simplest and therefore fastest metric, but does not penalize outlier feature values. The Euclidean distance deals better with outlier feature values and most likely would return better retrieval results, but the squaring operation is already relatively expensive and might take too long to be computed for large databases.

Practically, similarity measures have been preferred in image retrieval, because similarity measures facilitate certain type of matrix manipulations useful for improving the ranking. One popular similarity measure takes the dot product and then normalizes by their respective lengths,

$$\text{sim} = \frac{\mathbf{f}_1 \cdot \mathbf{f}_2}{\| \mathbf{f}_1 \| \| \mathbf{f}_2 \|}, \tag{17}$$

a measure, which is also referred to as the cosine similarity metric.

After we have computed the similarity (or distance) measure for one image against all other images, sim_i, we sort those values in decreasing order and that is our ranking result. Next, we want to quantify the ranking success.

18.2 Ranking Performance

FoPo p535, s16.2.2, pdf 505

When we return similar images in response to a user query, then we will choose a small subset of images, perhaps the first 20 or first 100 images. That small subset of images should contain as many similar images as possible. It is not helpful if only the first image is similar, but the remaining similar images are outside that subset. It is more desirable to have many similar images among the first few images - even if the very first or second image are dissimilar from the query image. This can be quantified using two measures, precision and recall. Those measures are essentially the same as for detection, i.e. object detection as in Section 11, and we have summarized those measures in Appendix I.1 already. But here we introduce them using exactly the terminology for retrieval. We determine the following numbers:

N_b the number of images returned to the user [query set]
N_r the total number of relevant items for this query [total]
n_r the number of recovered relevant items in the subset [relevant]

With those numbers, we calculate the percentage relevant to the query set size and relevant to the total relevant number:

Recall: the percentage of relevant items that are actually recovered

$$R = \frac{n_r}{N_r}, \qquad \frac{\text{relevant}}{\text{total}} \tag{18}$$

Example: if the database has a total of 200 relevant items for the user query and the selection returned 60 relevant items, then the recall value is 60/200.

The larger we make our selection N_b, the larger will be R and if we return all documents then R will be value equal one.

Precision: the percentage of recovered items that are actually relevant

$$P = \frac{n_r}{N_b}. \qquad \frac{\text{relevant}}{\text{query set}} \tag{19}$$

Example (continued): the precision value would be 60/100. The larger we make the selection N_b, the lower will be the precision value typically: if N_b is set to be 1000, and if this happen to return us all 200 relevant items, the precision is 200/1000.

Having two values as an indication for performance is awkward and one typically prefers a single number. There are two ways to obtain one. One way is to combine the two measures and we will introduce the *F-measure*. Another way is to systematically vary N_b and obtain a so-called *Precision-Recall* curve, from which one also extracts a single number.

F measure: To summarize the recall and precision values, different formulas can be used. A popular one is the F_1-measure, which describes the weighted harmonic mean of precision and recall:

$$F_1 = 2\frac{PR}{P+R}. \tag{20}$$

Precision-Recall Curve: To obtain a more comprehensive description of the performance, one calculates the precision values for increasing recall by systematically increasing N_b. Then, one plots the recall values on the x-axis against the precision values on the y-axis and this curve is called the precision-recall curve.

An important way to summarize a precision-recall curve is the *average precision*, which is computed for a ranking of the entire collection. This statistic averages the precision at which each new relevant image appears as we move down the list. $P(r)$ is the precision of the first r documents in the ranked list, whereby r corresponds to N_b in equation 19; N_r the total number of images in the collection. Then, the average precision value is given by

$$A = \frac{1}{N_r}\sum_{r=1}^{N_r} P(r), \tag{21}$$

which corresponds to the area under the curve.

Example In response to a query the (total of) 3 relevant items are found at positions 2, 13 and 36. The average precision value is then:

$$A = \frac{1}{3}\Big(\frac{1}{2} + \frac{2}{13} + \frac{3}{36}\Big) = 0.2457$$

Implementation Given an index vector with *sorted* retrieval positions, `Ix`, the measure is computed as follows:

```
Ixs    = sort(Ix);                          % sort in increasing order
nItm   = length(Ixs);                       % number of relevant items (N_r)
A      = sum( (1:nItm) ./ Ixs) / nItm;      % average precision
```

Trade-Offs A ideal retrieval system would show high recall and high precision. But obtaining high values can be costly and the following examples illustrate that sometimes trade-offs are made:

- Patent searches: Patents can be invalidated by finding prior art (material that predates the patent and contains similar ideas). A lot of money can depend on the result of a prior art search. This means that it is usually much cheaper to pay someone to wade through irrelevant material than it is to miss relevant material, so very high recall is essential, even at the cost of low precision.

- Web and email filtering: Some companies worry that internal email containing sexually explicit pictures might create legal or public relations problems. One could have a program that searched email traffic for problem pictures and warned a manager if it found anything. Low recall is fine in an application like this; even if the program has only 10% recall, it will still be difficult to get more than a small number of pictures past it. High precision is very important, because people tend to ignore systems that generate large numbers of false alarms.

18.3 Retrieval Tricks with Word-Like Features

Sze p604, s 14.3.2, pdf687

FoPo p662, s 21.2, pdf632

When image retrieval is carried out with detected features as introduced in Sections 12 and 13, then there exists an analogy to methods in document retrieval. In document retrieval, an individual feature is represented by a word; feature vectors would represent the vocabulary; a document is thus represented by a sparse feature vector. In case of images, a feature would represent a quantized patch - also called visual word in this context; the feature vector would represent the visual vocabulary (Section 13); an image would corresond to a document. We explain the methods in terms of document retrieval.

Much of text information retrieval is shaped by the fact that a few words are common, but most words are rare. The most common words - typically including 'the', 'and', 'but', 'it' - are sometimes called *stop words* and are ignored because they occur frequently in most documents. Other words tend to be rare, which means that their frequencies can be quite distinctive. Example: documents containing the words 'stereo', 'fundamental', 'trifocal' and 'match' are likely to be about 3D reconstruction.

Assume now that the total number of non-stop words is N_w; and we work with N_d documents. For each document j we determine the frequency f with which a word t occurs. This will generate a $N_w \times N_d$ table $\mathcal{D}_{tj}$ in which an entry represents the frequency f for a word t in a document j:

$$\{f_{t,j}\} = D(t, j) \tag{22}$$

The table is sparse as most words occur in few documents only. We could regard the table as an array of lists. There is one list for each word, and the list entries are the documents that contain that word. This object is referred to as an inverted index, and can be used to find all documents that contain a logical combination of some set of words. For example, to find all documents that contain any one of a set of words, we would take each word in the query, look up all documents containing that word in the inverted index, and take the union of the resulting sets of documents. Similarly, we could find documents containing all of the words by taking an intersection, and so on. This represents a coarse search only, as the measure f is used as a binary value only (and not as an actual frequency). A more refined measure would be the word count, or even better, a frequency-weighted word count. A popular method is the following:

tf-idf stands for 'term frequency-inverse document frequency' and consists of two mathematical terms, one for 'term frequency', the other for 'inverse document frequency'. With N_t as the number of documents that contain a particular term, the idf is

$$\text{idf} = \frac{N_d}{N_t}. \qquad \frac{\text{total}}{\text{containing term}} \tag{23}$$

Practical tip: add a value of one to the denominator to avoid division by zero. With
$n_t(j)$ for the number of times the term appears in document j and
$n_w(j)$ for the total number of words that appear in that document

the tf-idf measure for term t in document j is

$$f_{t,j} = \Big(\frac{n_t(j)}{n_w(j)}\Big) / \log\Big(\frac{N_d}{N_t}\Big). \tag{24}$$

We divide by the log of the inverse document frequency, because we do not want very uncommon words to have excessive weight. The measure aims at giving most weight to terms, that appear often in a particular document, but seldom in all other documents.

Because the histogram distributions for those words are generally different from those for gradients (as in SIFT), alternate distance measures such as the χ-squared kernel may perform better than the cosine similarity measure.

19 Tracking

Tracking is the close pursuit of one or multiple moving objects in a scene. Tracking is used in many applications:

Surveillance: traffic analysis: counting traffic participants, such as cars, pedestrians, cyclists, etc.; monitoring objects to report when something is suspicious, trucks at airports with unusual movement patterns.

Human Computer Interaction (HCI): continuously localizing the precise position of a user's head and orientation to facilitate the recognition of an user's emotions or the direction of his gaze.

Automotive Vision: anticipating when a moving object might enter the driving direction, i.e. a pedestrian.

Tracking is computationally intensive because we deal with a (rapid) series of images, called *frames* in this context. Image analysis has to occur very quickly in order to achieve real-time tracking; algorithms need to be optimized for speed; complex algorithms can only be applied intermittently (skipping a number of frames). The more objects we track, the more computing power is required. This multi-object tracking is manageable, if the camera is stationary; however when it is moving, as in an autonomous vehicle, then tracking becomes enormously challenging, because now one deals also with the motion of the camera, called *egomotion*, in addition to the moving objects. And if one attempts to maximize tracking accuracy with DeepNets, then we require also a lot of memory and power, something that is still a hindrance for embedded systems.

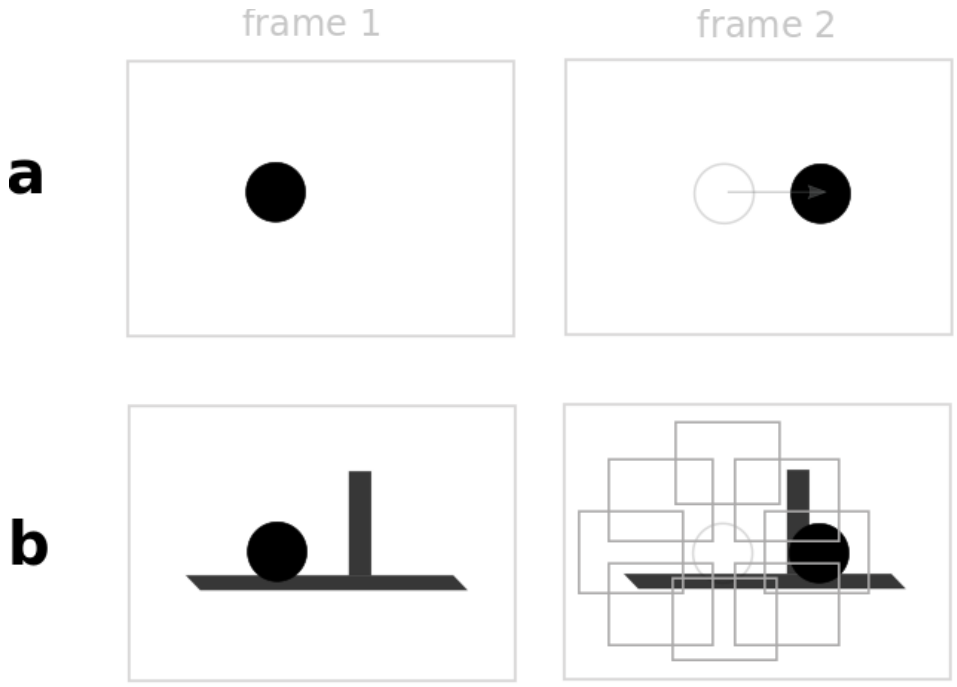

Figure 47: Tracking an object, a black disk moving to the right; two frames are shown.
a. In simple scenarios, such as surveillance, tracking can be done by firstly detecting regions that have moved, for instance by comparing frames pixel-wise. We then merely link up the detected, moving objects. This is also referred to as *tracking-by-detection*.
b. In complicated scenarios, when the background is a distractor, then tracking is done by knowing already what we intend to track, i.e. we apply an object localization algorithm as introduced in Section 11. As this involves a relatively costly search procedure, one attempts to reduce the search effort by using a limited range, here illustrated by 8 grey neighborhoods, radially placed around the object's location of the first frame. In this case one also speaks of *tracking-by-matching*, because one uses the object's properties to search for matching neighborhoods in the next frame.

For surveillance, the task is typically a relatively controlled situation, partly by design. The camera is often placed high above the zone to be observed, which results in small objects on an often homogeneous background. The size of the moving region corresponds to the size of the object. As the camera is stationary, so is the scene, and we can therefore use so-called *background subtraction* to easily detect moving objects. Tracking in that situation is sometimes described as tracking-by-detection; Figure 47a depicts the situation. We firstly introduce the technique of background subtraction (Section 19.1), then the techniques for pursuing multiple objects (Section 19.2).

Tracking becomes a bit more challenging in the scenario of face tracking for Human-Computer Interaction (HCI) for instance. Not only does the target move - the user's head - but also the target's context, namely the user's throat and shoulders, resulting in one large contiguous region moving across the display. It now requires a more elaborate tracking scheme, namely one in which we pursue the object's properties. This is done by searching in a small neighborhood around the tracking target, depicted in Fig. 47b. This is also referred to as *tracking-by-matching* sometimes, because we try to match object properties to small neighborhoods in the next frame. Tracking-by-matching also means that we now need to find the object first - for which we apply object detection algorithms as introduced in Section 10. In case of face-tracking, there exists a particularly succesful tracking technique that uses a clustering algorithm, the so-called mean-shift algorithm (Section 19.3).

Tracking becomes even more challenging, when the target object also drastically changes its appearance during the motion. For instance, seen from an autonomous vehicle, pedestrians increase in size very quickly. In that case, we wish to have a tracker that also adapts to the appearance changes during target pursuit (Section 19.4). That makes tracking computationally even more expensive. To reduce the search over a small neighborhood, it is useful to make predictions of where the objects might go next. For that purpose we carry a number of observations with each tracked object, for example motion direction and magnitude (speed). This would also help us resolve ambiguities when two objects cross their paths, a situation which easily leads to confusions of tracks. Two popular prediction schemes are mentioned in Section 19.5.

19.1 Background Subtraction

FoPo p291, s 9.2.1

SHB p776, s 16.5.1

Background subtraction is a method to determine which parts of the scene are stationary in order to facilitate the detection of significant motion, see also Appendix L. It makes sense to use it when the camera is stationary. Background subtraction is challenging for a number of reasons: changing daylight affects the overall scene luminance; clouds introduce temporary luminance changes; vegetation flutter is a source of noise; wandering object shadows cause easily confusions, etc. Background subtraction therefore requires continuous updating of the background parameters and is for that reason not quite as trivial as it sounds. One therefore needs to specify a set of parameters, such as the number of frames across which the background is calculated or a threshold to suppress noise. The most common technique to actually determine the background image is to take the median value for a pixel, taken across a number of frames.

In Matlab this process of background monitoring is called `ForegroundDetector` - the inverse of the background. It is initialized as follows for instance - before any tracking:

```
ForeGnd  = vision.ForegroundDetector('NumGaussians', 3, 'NumTrainingFrames', 40, ...
                                     'MinimumBackgroundRatio', 0.7);
```

During tracking, when we inspect each frame, we update the detector by providing the frame `Frm` itself:

```
Msk      = ForeGnd.step(Frm); % foreground [m n nCh]
```

where the output is called a mask, `Msk`, and is of the same size as the frame (number of pixels, number of channels).

In Python we can access methods for background subtraction through OpenCV. They are initialized with function names called `createBackgroundsubtractorXXX`:

```
import cv2 as cv
BckSub  = cv.createBackgroundSubtractorKNN()
```

When looping frames we call the method `.apply` to generate the mask:

```
Msk     = BckSub.apply(Frm)   # generates a mask
```

When one observes the output of the mask for an outdoor scene, one will notice that the mask can be very noisy. For that reason, the mask is often post-processed with morphological operations as introduced in Section 15. Two examples are given in the code section **P.16.1**. The parameters for the morphological operations are chosen such, that we avoid the elimination of potential target objects. The remaining regions are then our objects, our detections. That result is also called the *detector response*. To obtain the actual object outlines, we label and measure the actual region pixels as introduced in Section 15.3. Matlab provides the function `vision.BlobAnalysis` for this situation.

The Matlab movie `atrium.avi` shows well that this type of mere motion detection works well. It suffices however only for a coarse estimation of events: it is sufficient to estimate the number of people and follow their paths coarsely. Looking more closely, we observe that there are potential confusions, for instance when two persons cross their paths. Or if a person stands still, then the corresonding track might disappear.

19.2 Maintaining Tracks

FoPo p357, s 11.1.1

After we have found moving regions in individual frames, we try to link them up to obtain the trajectory. If there is only one object to pursue, then that linking is relatively straightforward. Even if the object is not detected in each frame - due to noise or occlusion for example -, then we can manage the pursuit by keeping a record of previous movements in a so-called *track*. In the first frame, we initiate the track by assigning the detected object. In the second frame, we match the new detection with the track of the previous frame. To facilitate continued tracking, it helps to store for how long the object had appeared in previous frames.

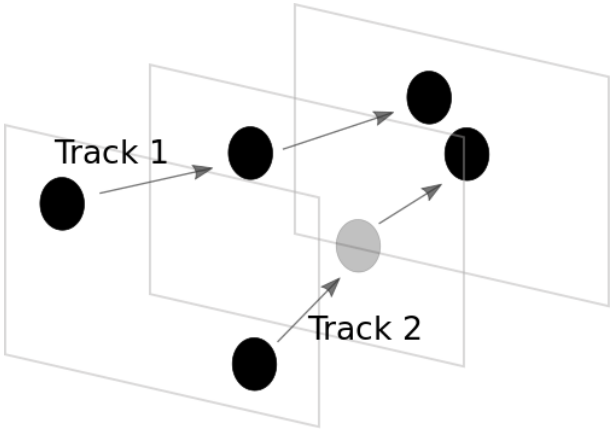

Figure 48: Pursuing multiple objects using *tracks*. Three frames are shown with two dots converging; one dot is not detected in the 2nd frame (shown as gray). In such situations it is necessary to work with records, called tracks, and find the corresponding matches from frame to frame, also known as a bipartite graph matching problem.

Tracking becomes more challenging with multiple objects (Fig. 48). Now we need to sort out, which ones are most likely to match and the obvious way is to take the spatial distance as a measure. We create a distance matrix in which we measure the distance between the last locations of the previous tracks with the locations of the new object detections of the present frame. Then we take the minimum. That assignment is complicated by the occasional lack or sometimes abundance of detections. In that case we require a more sophisticated scheme: the tracks from the previous frame are copied to the next frame, and then the new object detector responses are allocated to the tracks. Each track will be assigned at most one detector

response, and each detector response will get at most one track. However, some tracks may not receive a detector response, and some detector responses may not be allocated a track. Finally, we deal with tracks that have no response and with responses that have no track. For every detector response that is not allocated to a track, we create a new track, because a new object might have appeared. For every track that has not received a response for several frames, we prune that track, because the object might have disappeared. Finally, we may postprocess the set of tracks to insert links where justified by the application. Algorithm 4 breaks out this approach.

Algorithm 4 Tracking multiple objects (or tracking with unreliable object detector). i=time; t=track.

Notation:

Write $\mathbf{x}_k(i)$ for the k'th response of the detector in the ith frame
Write $t(k, i)$ for the k'th track in the ith frame
Write $*t(k, i)$ for the detector response attached to the k'th track in the ith frame
(Think C pointer notation)

Assumptions: Detector is reasonably reliable; we know some distance d such that $d(*t(k, i-1), *t(k, i))$ is always small.

First frame: Create a track for each detector response.

N'th frame:

Link tracks and detector responses by solving a bipartite matching problem.
Spawn a new track for each detector response not allocated to a track.
Reap any track that has not received a detector response for some number of frames.

Cleanup: We now have trajectories in space time. Link anywhere this is justified (perhaps by a more sophisticated dynamical or appearance model, derived from the candidates for linking).

The main issue in allocation is the cost model, which will vary from application to application. We need a charge for allocating detects to tracks. For slow-moving objects, this charge could be the image distance between the detect in the current frame and the detect allocated to the track in the previous frame. For objects with slowly changing appearance, the cost could be an appearance distance (e.g., a χ-squared distance between color histograms). How we use the distance again depends on the application. In cases where the detector is very reliable and the objects are few, well-spaced, and slow-moving, then a greedy algorithm (allocate the closest detect to each track) is sufficient. This algorithm might attach one detector response to two tracks; whether this is a problem or not depends on the application.

The more general algorithm solves a *bipartite matching problem*, meaning tracks on one side of the graph are assigned to the detector responses on the other side of the graph. The edges are weighted by matching costs, and we must solve a maximum weighted bipartite matching problem, which could be solved exactly with the Hungarian algorithm, but the approximation of a greedy algorithm is often sufficient. In some cases, we know where objects can appear and disappear, so that tracks can be created only for detects that occur in some region, and tracks can be reaped only if the last detect occurs in a disappear region.

A full Matlab example is given in Appendix **P.16.2**. That example includes a Kalman filter to predict a track's location for the next frame.

19.3 Face Tracking / CAM-Shift

We now introduce the first example of tracking-by-matching. As explained, in face tracking detection of moving regions alone is not sufficient: it requires something more specific. First, one performs face localization with an algorithm as introduced in Section 10.1: this is done in the first frame. In a subsequent frame, we want to track only the face region. We could therefore apply the face detector again just in a small neighborhood around the location of the detected face of the first frame. With todays computing power that is certainly doable and is in fact the idea of one type of tracking algorithm (next section), but here we introduce a succesful, venerable technique, to illustrate its elegance. The idea is to generate an intensity histogram of

the face and to track just that histogram in subsequent frames. This is computationally simpler, than trying to match entire neighborhoods. The histograms show characteristic peaks which we match across frames with the principle of mean-shift. The mean-shift is technique for clustering data points. It works similar to the K-Means algorithm, which uses distance measurements between points to find clusters. The mean-shift use gradients instead and that has proven to be quite effective in tracking faces.

Because a user might change his seating position - in particular if sitting in front of a home PC - tracking the initial histogram only is not reliable. One therefore adapts the entire procedure to the changes throughout the entire recording, analogous to the monitoring of the background for background subtraction (Section 19.1). This leads us to the CAM-Shift algorithm, the continuously adapting mean-shift algorithm. An example is shown in **P.16.3**. Matlab calls that tracker `HistogramBasedTracker`.

Face tracking nowadays is done with more capable detectors of course, introduced next.

19.4 Tracking by Matching and Beyond

FoPo p360, s11.1.2 or s11.2
Sze p337, s8.1, pdf384

Tracking by matching intensity histograms fails when we are faced with 'wilder' scenarios, for example, when objects are too small on the screen, or when their size changes substantially, for instance if the object is oncoming as in case of an autonomous vehicle tracking a pedestrian. There exists a large range of more sophisticated matching methods, but the essence remains the same: one searches over a small neighborhood to keep the computations at a possible minimum. Examples: some methods use Local Binary Patterns (LBP) as mentioned in Section 7.3.3; some use HOG features as introduced in Section 10.3; or they use other features as introduced in 12.1.

More ambitious trackers include learning. The object itself is learned at the beginning, such as in the boosting tracker; and/or the object itself is relearned during tracking as it might change its appearance during the motion. The most capable trackers are most likely the ones based on DeepNets - they come at the price of heavy resources; for a DeepNet one needs to store easily hundreds of MB of weights. Thus there is always a speed-accuracy tradeoff: the more robust a tracker, the slower is its operation.

In Python we can initiate various trackers in OpenCV with functions called `TrackerXXX_create()` and then calling its method `init`, for example:

```
Trkr    = cv.TrackerKCF_create()
bOk     = Trkr.init(Frm, bbox)
```

where variable `bbox` is the bounding box of our starting position. In the loop cycling through the frames, we update the tracker by calling the method `update`:

```
bOk, bbox = Trkr.update(Frm)
```

A full example is given in **P.16.4**.

19.5 Prediction and Update

Tracking can be optimized by predicting where the object will move in the next frame. That helps to reduce the search effort and also allows measuring higher tracking accuracy. We first observe how the object moves, for example we measure the motion vector across two frames, see large gray box in Fig. 49. We use that vector to predict the object's position for the next frame, shown as gray circle labeled 'predicted' (for Frame 3). Then we detect the actual object position, shown as black circle labeled 'measured'. Should the measured new motion vector be different from the previous ones, then we take that as our new prediction for frame 4. This is the obvious way to perform the simplest type of prediction.

We can increase the accuracy of prediction if we measure the object's trajectory across several frames. Furthermore, if we intend to track with maximal accuracy, then we should include a noise model, for both, the moving object, as well as the measurement. For example if we track a vehicle then it is likely that the vehicle will show some slightly irregular trajectory due to uneven road surface, or if we track a aerial vehicle its trajectory might be perturbed by air turbulence; that is also called the *process* noise. The actual sensor also contain noise, which comes into play in particular for low contrast objects; that is also called

the *measurement* noise. In that more refined tracking scenario, prediction and updating can become rather complex. One speaks of the prediction-correction cycle. There are two popular methods that implement such an iterative cycle, the Kalman filter and the Particle Filter, upcoming in the next two sections.

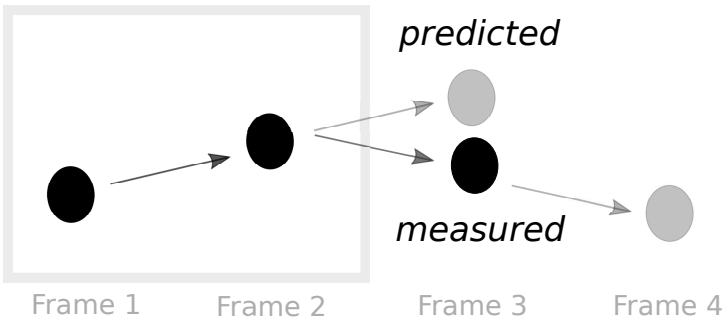

Figure 49: Tracking with prediction and error update. In frames 1 and 2, we make an initial motion estimate; the basis for this initial estimate is indicated by a gray rectangle including frames 1 and 2. Then we predict the position for the upcoming frame 3 (upper gray blob); after detection in frame 3 we measure the actual position (lower black blob). The difference between predicted and measured will be used to adjust the prediction model and to generate an update prediction for frame 4 (gray blob).

19.5.1 Kalman Filter

SHB p793, s 16.6.1

FoPo p369, s 11.3, pdf 339

The Kalman filter assumes that the underlying system is a linear dynamical system and that all error terms and measurements have uni-modal distribution, which is often taken to be a (multi-variate) Gaussian distribution. It works well for objects that do not move too erratically and that appear on a relatively homogeneous background; on cluttered background it starts to become unreliable wiki Kalman_filter. The equation for the linear update is often written as follows:

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{w}_t \tag{25}$$

where $\mathbf{x}_t$ and $\mathbf{x}_{t-1}$ are the current and previous state variables, $\mathbf{A}$ is the linear transition matrix, and $\mathbf{w}$ is a noise (perturbation) vector, which is often modeled as a Gaussian (Gelb 1974). The matrices $\mathbf{A}$ and the noise covariance can be learned ahead of time by observing typical sequences of the object being tracked (Blake and Isard 1998). We here summarize the entire model following the book by SHB, whereby k now represents time.

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}_k\mathbf{x}_k + \mathbf{w}_k, \\ \mathbf{z}_k &= H_k\mathbf{x}_k + \mathbf{v}_k. \end{aligned} \tag{26}$$

$\mathbf{A}_k$ describes the evolution of the underlying model state
$\mathbf{w}_k$ is zero mean Gaussian noise with assumed covariance $Q_k = E[\mathbf{w}_k\mathbf{w}_k^T]$
H_k is the measurement matrix, describing how the observations are related to the model
$\mathbf{v}_k$ is another zero mean Gaussian noise factor, with covariance $R_k = E[\mathbf{v}_k\mathbf{v}_k^T]$

In Matlab the Kalman filter is available under `vision.KalmanFilter`. It is initialized for example with

```
Klm     = vision.KalmanFilter(ModState, ModMesrm, 'ProcessNoise',1e-5, 'MeasurementNoise',1e-2);
```

whereby `ModState` corresponds to $\mathbf{A}$ and `ModMesrm` to H. We need to specify those matrices according to our task; and if we have knowledge of the process and measurement noise, then we specify those two values, corresponding to motion jitter and tracking inaccuracy in our case. During tracking we predict the next object location by calling

```
predict(Klm);                    % predict next location
```

After we have detected the object in a new frame and determined its new location `detLoc`, we hand it over to the Kalman tracker

```
trkLoc  = correct(Klm, detLoc);    % adjust prediction
```

and it returns the adjusted location in `trkLoc`, taking the previous motion dynamics into account.

Matlab provides several examples for the Kalman filter. We have rewritten the two simplest one as those provide a good intuition of how the algorithm works. In the first example, **P.16.5**, it is demonstrated that one property of the Kalman filter is too smoothen the recorded signal; there is no moving object involved but it simulates only the measurement of a constant signal and assumes that the measurement model is noisy, i.e. a camera pixel sensing a light value showing fluctuation. The Kalman filter smoothens that fluctuation.

The second example, **P.16.6**, a one-dimensional motion is simulated. It is simulated that the motion stimulus lacks some measurements. During the missing measurements (detections), the Kalman filter fills in the predictions - the situation that occurs when we drive through a tunnel and the navigation system estimates the location due to the lacking GPS signal.

In Python there does not appear an implementation within one of the larger modules. Perhaps a good module is the following, https://pykalman.github.io/.

In OpenCV [py] available as KalmanFilter.

19.5.2 Condensation Algorithm (Particle Filters)

Sze p243, s 5.1.2, pdf 276
SHB p798, s 16.6.2
FoPo p380, s 11.5, pdf 350

The condensation algorithm maintains a swarm of particles to pursue an object. The swarm follows a specified target in a way similar to the physical process of drift. The idea is presented in Fig. 50, where the first three frames of a (primitive) face tracking process are shown.

In the first frame (left image), we select a few pixels at random as our initial swarm, shown in blue. Those pixels are our initial set of particles, $n_{particles}$ in count. Then we specify a target value, in our case a pink value in RGB space, i.e. [255 20 100], as we wish to track the face. Now we calculate the distance between our particle pixel values and the target value and use that 'error' value to determine a probability that the pixel is on target, some inverse of that distance (error). Then we select those particle pixels, that show a high probability (low distance/error), at random to some degree. With that sub-selection, two steps occur: in a first step, we split the individually select particles, such that the total count equals the original particle count $n_{particles}$. In a second step, we add a drift to those particles, a small movement in space in a random direction. The result is shown in the second frame (center image).

Figure 50: Tracking with a swarm of particles (condensation algorithm); the target is a pink pixel value, which will lead to tracking the face in this example. Frame 1 (left): random selection of a set of pixels, our particles. Error calculation (particle pixel value minus target value) and selection of those particles with low error. Splitting of that sub-selection and making them drift into random directions; shown in Frame 2 (center). We repeat the low-error selection, splitting and drifting, shown in Frame 3 (right). The yellow dot marks the mean value of all particle positions.

The procedure is repeated: first the sub-selection of fitter pixels; second the splitting of fitter pixels to the total particle count $n_{particles}$; thirdly the drift of those new pixels. In our example frames, the particles have gathered around the face in the third frame already (right image). A code example is given in **P.16.7**.

More formally speaking, a particle filter is a sampling method that approximates distributions by exploiting their temporal structure; in computer vision they were popularized mainly by Isard and Blake in their CONditional DENSity propagATION algorithm, short CONDENSATION wiki Condensation_algorithm. It was introduced as an alternative to the Kalman filter; the CONDENSATION algorithm allows dealing with multimodal density distributions as no restrictive assumption about the dynamics of the state-space or the density function to be estimated are made. In the field of Artificial Intelligence, the algorithm is known as 'survival of the fittest'.

20 Optic Flow (Motion Estimation I)

Dav p506, s19.2
Sze p360, s8.4
FoPo p343, s10.6, pdf 313
SHB p757, s16.2

Optic flow is the estimation of motion between two successive frames, resulting in a pattern of motion vectors that is also called the *vector flow-field*. Figure 51 shows that flow-field for two directions of motion for an object. In the left scenario, the object moves as depicted already in Fig. 2 (toward right and rotating, as if kicked). If we manage to make a correspondence between the points in the first frame and those in the second frame - indicated by four arrows in this case - then we have information that precisely describes the motion. In the right scenario, the object moves toward us: the small rectangle (in light gray) depicts the first frame, the larger rectangle (in dark gray) depicts the second frame: now, the arrows point radially outward.

The optic flow for an individual object is also called a *local-flow* field; it can be understood as the tracking of an object's individual parts. Optic flow is difficult to determine due to the challenge of finding the corresponding features or pixels across frames and for that reason it is also computationally intensive. For objects the flow is particularly difficult to determine, as they occupy only a part of the image and for that reason it is rarely exploited in its full.

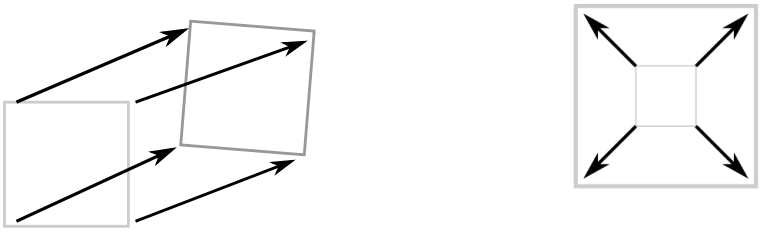

Figure 51: Local flow field for an object moving between two frames. **Left:** object moves as in Fig. 2: from left to right with a slight rotation. **Right:** object moves toward observer.

Optical flow is of particular use in the scenario of *egomotion*, when the camera (observer) moves such as in a moving vehicle (robot, drone, self-driving automobile, etc). Figure 52 shows two examples:

Forward Looking (a) the observer moves forward and looks forward. This creates a radial flow pattern as objects will move to the periphery of the visual field, away from the vanishing point. Near objects move faster than far objects - if they are fixed in position (or slowly moving) -, resulting in longer vectors.

Side-Ways Looking (b) the observer moves forward but looks toward one side, e.g. observing the landscape from a (running) train. This creates a horizontal flow pattern with near objects again generating longer vectors than far objects.

Methods

Optical flow can be calculated with a variety of methods and - one may have guessed it already - those methods are similar to the ones for tracking. Optical flow can be calculated pixel-by-pixel, which is computationally intensive but precise; it can be calculated feature-by-feature, which is much faster but we deal with the challenge of finding good feature matches; see also Appendix E for the issue. We sketch some prominent, distinct approaches:

Block-Based: here one searches in a small neighborhood, analogous to the search for the object in tracking, see again Fig.47b. One minimizes the sum-of-squared differences or sum-of-absolute differences (Section 5.1.2), or one maximizes the normalized cross-correlation. It is perhaps the simplest approach with respect to spatial complexity, but it is also a costly one.

Differential, Global: this method calculates optical flow for the entire image, in some sense at once, hence the term global. It does so by using the two derivatives of an image, in each direction, akin to calculating gradients in Section 6.3. Its most famous representative method is the Horn-Schunck algorithm,

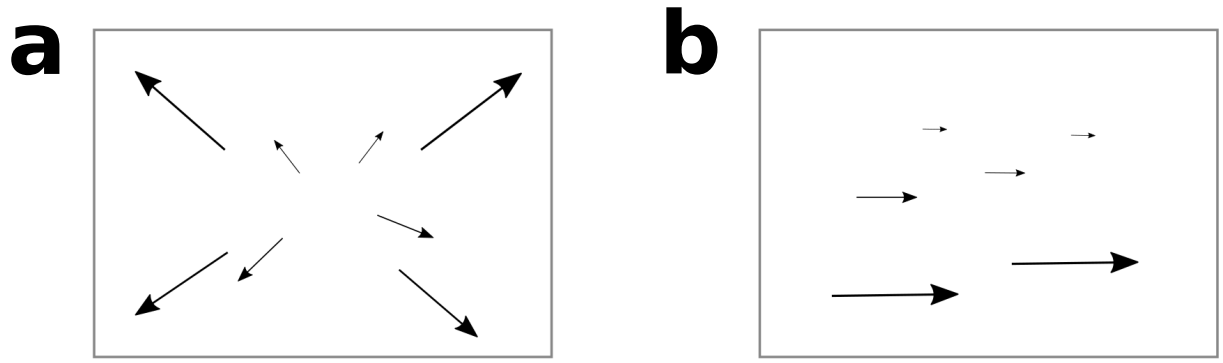

Figure 52: Optical flow fields are velocity flow fields typically measured for the entire scene during ego-motion (camera moves), but can be equally interesting for objects moving across the scenes (camera stationary). The two examples are for ego-motion. Long vectors correspond typically to objects near the observer.
a. Optic flow as generated when moving forward *and* looking forward, i.e. driving a car.
b. Optic flow as generated when moving forward *but* looking sideways, i.e. observing the landscape from a train.

abbreviated HS in software packages: it yields a high density of flow vectors, i.e. the flow information missing in inner parts of homogeneous objects is filled in from the motion boundaries. On the downside, the algorithm is sensitive to noise. wiki Horn-Schunck_method

Differential, Local: is a method also based on the image derivatives, and hence the same term 'differential'. But unlike the global method, it partitions the image to avoid the sensitivity to noise. The best known method for this approach is the Lucas-Kanade method, abbreviated LK in software packages. The method cannot provide flow information in the interior of uniform regions. wiki Lucas-Kanade_method

And there exist of course combinations of local and global methods that attempt to combine the advantages of both methods. The OpenCV library contains a large set of implementations. Optical flow algorithms can be evaluated on the following databases for instance:

`http://vision.middlebury.edu/flow/`
`http://of-eval.sourceforge.net/`

In Matlab we can calculate optical flow with two functions: one for initiating a structure whose name starts with `opticalFlow`; and one function for updating the flow-field computation during looping, called `estimateFlow`. A full example is given in Appendix **P.17**.

In Python we can access some of the methods through OpenCV, whose function names start with `calcOptialFlow`, for instance:

```
import cv2 as cv
Pts1, St, err = cv.calcOpticalFlowPyrLK(Iold, Inew, Pts0, None, **PrmOptFlo)
```

where `Iold` is the previous frame, `Inew` is the next frame, `Pts0` are a list of feature coordinates from the previous frame and `PrmOptFlo` is a list of parameters for that algorithm (full example in Appendix **P.17**).

Applications Optical flow is not only exploited during ego-motion (moving camera), but it can also be employed to detect change in sequences (see also Appendix L), and that in turn is exploited in video compression.

21 Alignment (Motion Estimation II)

FoPo p397, ch 12, pdf 446

Sze p273, ch 6, pdf 311

Alignment is the process of estimating object or camera motion. This motion estimation is typically carried out after the object or observer has completed its movement; it is therefore rather a reconstruction process, as opposed to tracking or calculating optic flow, two processes that can be considered continuous processes designed for monitoring (following). In the alignment process, one is interested in how the object or observer has moved precisely across some period of time, in some sense between two frames separated by some duration.

The motion can be estimated with various methods. In this section, we pursue an estimation that uses the analytical, geometric approach, namely by finding a geometric transformation for the motion. In mathematics one speaks of how the object *transformed*: we seek the transform that describes the motion as informative and precise as possible. Figure 53 shows the common 2D transformations.

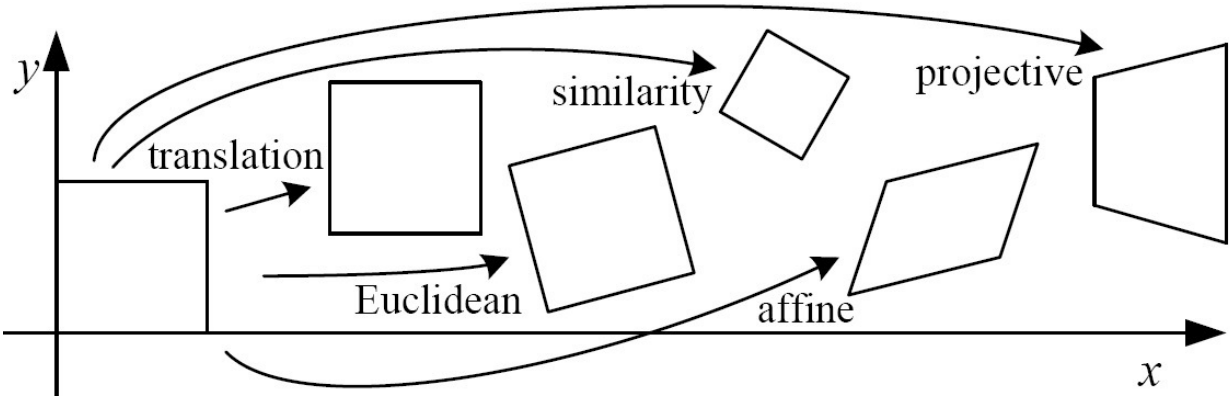

Figure 53: Basic set of 2D planar transformations. Used for estimating object motion (fixed observer) or camera motion (observer moved). The square shape at the origin is also called *moving* image/points or source; the transformed shapes are the target, *fixed* or sensed image/points.
translation: a shift in position.
Euclidean: translation and rotation, also known as rigid body motion.
similarity: translation, rotation and scaling.
affine: including distortion.
projective: including change of viewpoint.
[Source: Szeliski 2011; Fig 3.45]

As introduced in Section 2.2 (Fig. 3), there are two principal scenarios where this motion estimation is calculated:

Object Motion: is the reconstruction of how precisely an object has moved, we attempt to match the two poses by some transformation.

Camera Motion: is the reconstruction of how the camera (observer) has moved. In other words, we are given two images of some scene or object, but taken from different viewpoints. We estimate the observer (camera) motion; we estimate the observer's new viewpoint.

Both scenarios are sometimes more generally referred to as *pose estimation*, which however should not be confused with posture estimation for humans (Section 26.4).

The methods to estimate the transforms are analogous to the methods for estimating optical flow. We operate with a vector flow-field and we can attempt to make the estimation with all pixels, or with a subset of pixels, that were identified by feature detectors for instance. If done with features, then we also need to determine which points or features correspond to each other, an issue also know as the *correspondence problem* (Appendix E). Once we have established that correspondence, then we can determine the exact

motion parameters by using geometric transformations. This will be introduced later; for the moment we continue the introduction of the geometric transformations.

In the simplest case, the object has moved straight from one location to another and otherwise it did not change: that would be a mere *translation*, see Fig. 53. But the object may have also rotated during the *motion*, in which case the motion corresponds to a so-called *Euclidean* transformation. The object may have also shrunk or enlarged during motion, in which case the transformation is expressed with *similarity*. The object may have deformed during motion, in which case we can attempt to reconstruct the motion with a so-called *affine* transformation. Or in case of camera motion, the scene may appear from a different perspective, something which can be reconstructed with a *projective* transform. The latter two, the affine and projective transform, are however very complex and with those it is not easy anymore to infer the parameters translation, rotation and scaling, although those three parameters are very informative and intuitive.

In the most straightforward form, the two datasets have the same dimensionality, for instance we are registering 2D data to 2D data or 3D data to 3D data, and the transformation is rotation, translation, and perhaps scale. Here are some example applications:

Shape Analysis: A simple 2D application is to locate shapes, for instance finding one shape in another image as employed in biology, archeology or robotics. If the transformations involve translation, rotation, scaling and reflection, then it is also called *Procrustes* analysis wiki Procrustes_analysis.

Medical Support: We have an MRI image (which is a 3D dataset) of a patient's interior that we wish to superimpose on a view of the real patient to help guide a surgeon. In this case, we need to know the rotation, translation, and scale that will put one image on top of the other.

Cartography: We have a 2D image template of a building that we want to find in an overhead aerial image. Again, we need to know the rotation, translation, and scale that will put one on top of the other; we might also use a match quality score to tell whether we have found the right building.

In the following section we explain how some of the 2D motions of Figure 53 are expressed mathematically (Section 21.1). Then we learn how to estimate them (Section 21.2), assuming that the correspondence problem was solved. Finally, we learn how to robustly estimate them, that is even in the presence of noise and clutter when the correspondence problem is a challenge (Section 21.3). At the very end, we have a few words on the topic of image registration (Section 21.4).

21.1 2D Geometric Transforms

Sze p33, s 2.1.2, pdf 36

Figure 53 showed the global parametric transformations for rigid 2D shapes. Now we introduce the formulas that express those transformations with matrix multiplications, whereby two frequent notations are given in Table 2: notation 1 is more explicit with respect to the individual motions; notation 2 concatenates the individual motions into a single matrix such that the entire motion can be expressed as a single matrix multiplication, which can be convenient sometimes.

Translation: simplest type of transform - merely a vector $\mathbf{t}$ is added to the points in $\mathbf{x}$. To express this as a single matrix the identity matrix is used (unit matrix; square matrix with ones on the main diagonal and zeros elsewhere) and the translation vector is appended resulting in a 2×3 matrix, see also Figure 54.

2D Euclidean Transform: consist of a rotation and translation. The rotation is achieved with the a so-called orthonormal rotation matrix $\mathbf{R}$ whose values are calculated by specifying the rotation angle θ. In notation 2, the single matrix is concatenated as before and it remains a 2×3 matrix.

Scaled Rotation: merely a scale factor is included to the previous transform. The size of the single matrix in notation 2 does not change.

Affine: here a certain degree of distortion is allowed. The single matrix for notation 2 still remains of size 2×3.

Projective: is also known as *perspective transform* or *homography*. The transformation matrix is now of size 3×3 and is typically given letter H.

Transform	Notation 1	Notation 2	Comments
Translation	$\mathbf{x}' = \mathbf{x} + \mathbf{t}$	$\mathbf{x}' = \begin{bmatrix}\mathbf{I} & \mathbf{t}\end{bmatrix}\mathbf{x}$	$\mathbf{I}$ is the 2×2 identity matrix
Rotation + Translation (2D rigid body motion, 2D Euclidean transformation)	$\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$	$\mathbf{x}' = \begin{bmatrix}\mathbf{R} & \mathbf{t}\end{bmatrix}\mathbf{x}$	$\mathbf{R} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$ = orthonormal rotation matrix: $\mathbf{R}\mathbf{R}^T = \mathbf{I}$ and $\lvert\mathbf{R}\rvert = 1$ Euclidean distances are preserved.
Scaled Rotation (Similarity Transform)	$\mathbf{x}' = s\mathbf{R}\mathbf{x} + \mathbf{t}$	$\mathbf{x}' = \begin{bmatrix}s\mathbf{R} & \mathbf{t}\end{bmatrix}\mathbf{x}$ $\mathbf{x}' = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \end{bmatrix}\mathbf{x}$	no longer requires $a^2 + b^2 = 1$
Affine	$\mathbf{x}' = \mathbf{A}\mathbf{x}$	$\mathbf{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}\mathbf{x}$	parallel lines remain parallel
Projective (Perspective Transform, Homography)	$\mathbf{x}' = \mathbf{H}\mathbf{x}$	$\mathbf{x}' = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}\mathbf{x}$	straight lines remain straight

Table 2: The starting point of the shape (or set of points) is expressed with coordinates in variable $\mathbf{x}$; the transformed (moved) shape is assigned to variable $\mathbf{x}'$.

It is instructive to understand how we can apply these transforms, in order to understand how we can reconstruct them. We do this first for a set of points (Section 21.1.1), then for an entire image (Section 21.1.2).

21.1.1 Moving Points

Assume our shape is a set of 2D points, expressed in variable `Pts`, holding x- and y- coordinates. We intend to translate it by vector `[tx ty]`. By notation one, we would simply add the vector's components, e.g.

```
Ptf1(:,1) = Pts(:,1)+tx;
Ptf1(:,2) = Pts(:,2)+ty;
```

Using notation two, we firstly build the transformation matrix, which in our case is:

```
Ttrs    = [1 0 tx; 0 1 ty];
```

Then we can simply multiply the points by the transformation matrix, whereby we need to consider the position of the shape, called `Cen` here:

```
Ptf2    = [Pts-Cen ones(np,1)] * Ttrs' + Cen;
```

Appendix **P.18.1** gives a full example.

In Matlab the transformations can be carried out with function `affine2d`. It requires a 3x3 matrix, of which the first two rows would specify the values as in notation two; the last row is required to contain the values `[0 0 1]`. For projective transformations one would use `projective2d`. We specify the transformation matrix and then apply the method `transformPointsForward`. An example is shown in Appendix **P.18.1**.

In Python the transformations can be carried out with `skimage.transform.matrix_transform` of module `skimage.transform`. It requires the specification of a homogeneous 3x3 matrix.

21.1.2 Moving an Image

The transformations can also be applied to an entire image. That is what is frequently done for data augmentation when training neural networks (Section 8), see Appendix J in particular. The transformation is applied to each pixel position and by doing so we obtain a moved image that has left its original frame;

we therefore need to pad missing pixel positions in order to be able to display the image properly. Software packages provide functions doing this padding automatically.

In Matlab there exists single functions to carry out individual transformations such as `imtranslate`, `imrotate` and `imresize`; `imcrop` is useful too for data augmentation. More complex transformations, such as affine and projective, are carried out in two steps: first we generate the transformation matrix with functions such as `affine2d` or `projective2d` - as introduced above; then, we apply the function `imwarp` to carry out the transformation. Some examples are given in Appendix **P.18.2**. Older Matlab versions use function scripts called `maketform` and `imtransform`.

In Python all those functions are found in module `skimage.transform`, e.g. `skimage.transform.warp`.

In PyTorch all those functions are packed into the module `torchvision.transforms`.

21.2 Motion Estimation with Linear-Least Squares

Sze p33, s 2.1.2, pdf 36

To estimate the motion we require two types of information. One is the correspondence between the two sets of points, which is not so trivial to establish in real-word scenes (see feature detection before), but for the moment we assume that the correspondence is known. The other type of information is the kind of transformation we expect. To address this we could simply work with complex transformations in order to be prepared for any type of transformation (i.e. affine transformation), which however do not return us the individual motion parameters explicitly; in scaled rotation in contrast we have explicit parameters for rotation, scale and translation.

Formulated mathematically, given a set of matched feature points $\{\mathbf{x}_i, \mathbf{x}'_i\}$ and a chosen planar transformation $\mathbf{f}$ with parameters $\mathbf{p}$ (e.g. $\mathbf{t}, \mathbf{R}$...),

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}; \mathbf{p}), \tag{27}$$

how can we reconstruct the best estimate of the motion parameters values? The usual way to do this is to use least squares, i.e., to minimize the sum of squared residuals

$$E_{LS} = \sum_i \|\mathbf{r}_i\|^2 = \sum_i \|\mathbf{f}(\mathbf{x}_i; \mathbf{p}) - \mathbf{x}'_i\|^2, \tag{28}$$

where

$$\mathbf{r}_i = \mathbf{x}'_i - \mathbf{f}(\mathbf{x}_i; \mathbf{p}) = \hat{\mathbf{x}}'_i - \tilde{\mathbf{x}}'_i \tag{29}$$

is the residual between the measured location $\hat{\mathbf{x}}'_i$ and its corresponding current predicted location $\tilde{\mathbf{x}}'_i = \mathbf{f}(\mathbf{x}_i; \mathbf{p})$.

For simplicity we assume now a linear relationship between the amount of motion $\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x}$ and the unknown parameters $\mathbf{p}$:

$$\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x} = \mathbf{J}(\mathbf{x})\mathbf{p}. \tag{30}$$

where $\mathbf{J} = \partial\mathbf{f}/\partial\mathbf{p}$ is the Jacobian of the transformation $\mathbf{f}$ with respect to the motion parameters $\mathbf{p}$. $\mathbf{J}$ is shown in figure 54 and has a particular form for each transformation. In this case, a simple *linear* regression (linear-least-squares problem) can be formulated as

$$E_{LLS} = \sum_i \|\mathbf{J}(\mathbf{x}_i)\mathbf{p} - \Delta\mathbf{x}_i\|^2, \tag{31}$$

which - after some reformulation - equates to

$$\mathbf{p}^T\mathbf{A}\mathbf{p} - 2\mathbf{p}^T\mathbf{b} + c. \tag{32}$$

The minimum can be found by solving the symmetric positive definite (SPD) system of normal equations:

$$\mathbf{A}\mathbf{p} = \mathbf{b}, \tag{33}$$

where

$$\mathbf{A} = \sum_i \mathbf{J}^T(\mathbf{x}_i)\mathbf{J}(\mathbf{x}_i) \tag{34}$$

is called the Hessian and

$$\mathbf{b} = \sum_i \mathbf{J}^T(\mathbf{x}_i)\Delta\mathbf{x}_i. \tag{35}$$

Transform	Matrix	Parameters p	Jacobian J
translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$	(t_x, t_y)	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Euclidean	$\begin{bmatrix} c_\theta & -s_\theta & t_x \\ s_\theta & c_\theta & t_y \end{bmatrix}$	(t_x, t_y, θ)	$\begin{bmatrix} 1 & 0 & -s_\theta x - c_\theta y \\ 0 & 1 & c_\theta x - s_\theta y \end{bmatrix}$
similarity	$\begin{bmatrix} 1+a & -b & t_x \\ b & 1+a & t_y \end{bmatrix}$	(t_x, t_y, a, b)	$\begin{bmatrix} 1 & 0 & x & -y \\ 0 & 1 & y & x \end{bmatrix}$
affine	$\begin{bmatrix} 1+a_{00} & a_{01} & t_x \\ a_{10} & 1+a_{11} & t_y \end{bmatrix}$	$(t_x, t_y, a_{00}, a_{01}, a_{10}, a_{11})$	$\begin{bmatrix} 1 & 0 & x & y & 0 & 0 \\ 0 & 1 & 0 & 0 & x & y \end{bmatrix}$

Figure 54: Jacobians of the 2D coordinate transformations $\mathbf{x}' = \mathbf{f}(\mathbf{x}; \mathbf{p})$ (see table before), where we have re-parameterized the motions so that they are identity for $\mathbf{p} = 0$. [Source: Szeliski 2011; Tab 2.1]

Practically, we write a loop that takes each point and calculates the dot product and accumulates them to build $\mathbf{A}$. Then we call a script that does the least-square estimation. This is shown in the second part in the code of **P.18.3**. The function `lsqlin` carries out the least-square estimation. We also show how to use Matlab's function scripts `fitgeotrans` that does both in a single script - building $\mathbf{A}$ and $\mathbf{b}$, as well the estimation of the motion parameters. In that code we also show how to use the function script `procrustes`, which does also everthing at once.

When using functions in software packages the preferred terminology is as follows: the first image is referred to as the *moving* image (or points), or the *source*; the second image is referred to as the *target*, *fixed* or *sensed* image (points).

In Python the procrustes analysis can be found in module `scipy.spatial`. I am not aware of a single script in Python, but OpenCV provides functions that do the entire alignment process.

21.3 Robust Alignment with RANSAC

FoPo p332, s 10.4.2, pdf 30
Sze p281, s 6.1.4, pdf 318
Pnc p342, s 15.6
SHB p461,s.10.2

As pointed out already, in complex situations it is not straightforward to establish the correspondence between pairs of points. Often, when we compare two images - or two shapes in different images - we find only a subset of the shape points and sometimes wrong points or features were determined. One can summarize such 'misses' as outliers and noise, and they make motion estimation in real world applications difficult. It therefore requires more robust estimation techniques, for instance an iterative process consisting of the following two steps:

1) selection of a random subset of points and estimation of their motion.
2) verification of the estimate with the remaining set of points.

Example: We are fitting a line to an elliptical point cloud that contains about 50% outliers - or that is embedded in random noise. If we draw pairs of points uniformly and at random, then about a quarter of these pairs will consist of 'inlier' data points. We can identify these inlier pairs by noticing that a large proportion of other points lie close to the fitted line. Of course, a better estimate of the line could then be obtained by fitting a line to the points that lie close to our current line.

Algorithm Fischler and Bolles (1981) formalized this approach into an algorithm called RANSAC, for RANdom SAmple Consensus (algorithm 5). It starts by finding an initial set of inlier correspondences, i.e., points that are consistent with a dominant motion estimate: it selects (at random) a subset of n correspondences, which is then used to compute an initial estimate for $\mathbf{p}$. The *residuals* of the full set of correspondences are then computed as

$$\mathbf{r}_i = \tilde{\mathbf{x}}'_i(\mathbf{x}_i; \mathbf{p}) - \hat{\mathbf{x}}'_i, \tag{36}$$

where $\tilde{\mathbf{x}}'_i$ are the *estimated* (mapped) locations and $\hat{\mathbf{x}}'_i$ are the sensed (detected) feature point locations.

Algorithm 5 RANSAC: Fitting structures using Random Sample Consensus. FoPo p332, pdf 305

Input : $\mathcal{D}$, $\mathcal{D}^*$
Parameters:
n the smallest number of points required (e.g., for lines, n = 2, for circles, n = 3)
k the number of iterations required
t the threshold used to identify a point that fits well
d the number of nearby points required to assert a model fits well
Until k iterations have occurred:
- Draw a sample of n points from the data $\mathcal{D}$ uniformly and at random $\rightarrow \mathcal{D}^s$; $\mathcal{D}^c = \mathcal{D} \setminus \mathcal{D}^s$
- Fit to $\mathcal{D}^s$ and obtain estimates $\mathbf{p}$
- **For** each data point $\mathbf{x} \in \mathcal{D}^c$: if point close,
  that is smaller than a threshold: $\|\mathbf{r}_i\|^2 < t$, then $\mathbf{x}_i \rightarrow \mathcal{D}^{good}$
  end
- If there are d or more points close to the structure ($|\mathcal{D}^{good}| \geq d$), then $\mathbf{p}$ is kept as a good fit.
  - refit the structure using all these points.
  - add the result to a collection of good fits $\rightarrow \mathcal{P}^{good}$

end
Choose the best fit from $\mathcal{P}^{good}$, using the fitting error as a criterion

The RANSAC technique then counts the number of inliers that are within ϵ of their predicted location, i.e., whose $\| \mathbf{r}_i \| \leq \epsilon$. The ϵ value is application dependent but is often around 1-3 pixels. The random selection process is repeated k times and the sample set with the largest number of inliers is kept as the final solution of this fitting stage. Either the initial parameter guess $\mathbf{p}$ or the full set of computed inliers is then passed on to the next data fitting stage.

Matlab's computer vision toolbox provides a function. In Appendix **P.19** we provide a primitive version to understand it step by step. Python offers the function `ransac` in submodule `skimage.measure`.

21.4 Image Registration

In image registration we aim at aligning entire scenes. For example in medical image analysis, one would like to align the scans of the human body, and because organs deform, this is a rather challenging task. In addition, the scans are carried out with different scanning modes, aggravating the registration effort.

There exists a large range of techniques, each one with advantages and disadvantages. Those techniques can be classified according to different aspects. One distinction is made based on the number of points selected for registration. In feature-based methods, one detects characteristic features as introduced in Section 12 and then estimates the motion using RANSAC for instance. That would be a registration based on a subset of points. In intensity-based methods, one uses all pixel values at hand - the entire image - and that can be more accurate but also requires more computation, analogous to the computation of optical flow.

In Matlab, image registration can be carried out with `imregister`. In Python, one rather uses functions from OpenCV.

22 3D Vision

SHB p546, ch11, ch12

3D Vision is a general term for tasks related to understanding a scene as a 3D space, the ultimate goal if one intends to interact with the world. In a first step, it is beneficial to perceive depth, the ability to sense the distance to objects in a scene (Section 22.1). Depth perception is useful, for example, for industrial robots reaching for tools and for autonomous vehicles trying to avoid obstacles. Depth information is also exploited in gesture recognition sometimes, with the most famous example being the Kinect system (to be described in Section 23.2). With depth information, we can go further and perform viewpoint or pose estimation (previous Section or Fig. 3); some more explanations on that topic are given in Section 22.2.

22.1 Depth Perception

Depth information can be obtained with different cues. One can try to obtain depth from the two-dimensional image only by elaborate reconstruction. Or one can simply use a camera that senses depth directly, a technique called *range imaging*, see again Appendix A. Each cue and technique has advantages and disadvantages. We here mention only the popular ones, of which the first one, stereo correspondence, uses two-dimensional information only:

Stereo Correspondence: takes two or more images taken from slightly different positions and then determines the offset (disparity) between the images - a process also done by the human visual system to estimate depth. To determine disparity one needs to find the corresponding pixels across images, a process that is relatively challenging. The advantage of the technique is that is can be carried out with cheap, light-sensitive (RGB) cameras. We elaborate on that method in Section 22.1.1.

Lidar (Light Detection And Ranging): is a surveying method that measures distance to a target by illuminating the target with a pulsed laser light, and measuring the reflected pulses with a sensor. It typically works for a specific depth range. This sensor is particularly used for autonomous vehicles and in order to cover the entire depth of the street scene, one employs several Lidar sensors each one tuned for a specific depth.

Time-of-Flight (ToF): a relatively new technique that uses only a single pulse to scan the environment - as opposed to Lidar (or Radar). Early implementations required relatively much time to compute the distance - as opposed to Lidar and Radar -, but recent implementations can provide up to 160 frames per second.

The output of such a technique is typically a two-dimensional map, in which the intensity values correspond to depth measured in some unit (i.e. centimeters). That map is also called *depth map*, or *range map* if obtained with range imaging. The analysis of such a depth map does not pose any novel computational challenges: to obtain a segregation between foreground and background one can use the segmentation techniques as introduced previously. For that reason there is not more to explain here and in the following we merely elaborate on the process of stereo correspondence as that is an algorithmic issue (Section 22.1.1).

Depth information can also be obtained from single images in principle - called *shape from X* sometimes -, but that has not been as practical yet as the above techniques. We mention those methods nevertheless (Section 22.1.2).

22.1.1 Stereo Correspondence

Sze p469, ch11, pdf533
SHB p573, s11.5, s11.6.1, p
FoPo p227, ch7

Stereopsis is the perception of depth using two (or more) images taken from slightly different viewpoints. The corresponding points between the two images have a slight, horizontal offset (motion), so-called *disparity*, which is inverse proportional to depth; it can be assumed that all animals with two forward oriented eyes exploit that for precise depth perception. The main challenge is to determine the correspondence between the two images, a problem similar to determining optic flow (Section 20), see also Appendix E. Whereas in optic flow we do not know how the two frames are related, in stereo correspondence we know the setup

of the two cameras and that allows us to facilitate the search for correspondence by assuming some constraints, see SHB for a nice overview (SHB p584, s11.6.1). To assign a proper depth value to corresponding points we need to calibrate our system by presenting it with a pattern of known size.

In the following we firstly introduce the most popular constraint. Then we give an example for calibration. Finally, we explain what methods are useful for establishing correspondence.

Epipolar Constraint A popular constraint to compute stereo correspondence is the so-called epipolar constraint (wiki Epipolar_geometry). Figure 55 introduces the idea. $\mathbf{c_0}$ and $\mathbf{c_1}$ are the optical centers of the (camera) lenses; the line connecting those two optical centers is the *baseline*, also known as *parallax* (wiki Parallax). The baseline intersects the image planes in the epipoles $\mathbf{e_0}$ and $\mathbf{e_1}$. Any scene point $\mathbf{p}$ observed by the two cameras and the two corresponding rays from optical centers $\mathbf{c_0}$, $\mathbf{c_1}$ define an epipolar plane. This plane intersects the image planes in the epipolar lines (or just epipolars) $\mathbf{l_0}$ and $\mathbf{l_1}$. Pixel $\mathbf{x}_0$ in the left image corresponds to epipolar line segment $\overline{\mathbf{x_1 e_1}}$ in the right image, and vice versa, point $\mathbf{x}_1$ corresponds to $\overline{\mathbf{x_0 e_0}}$.

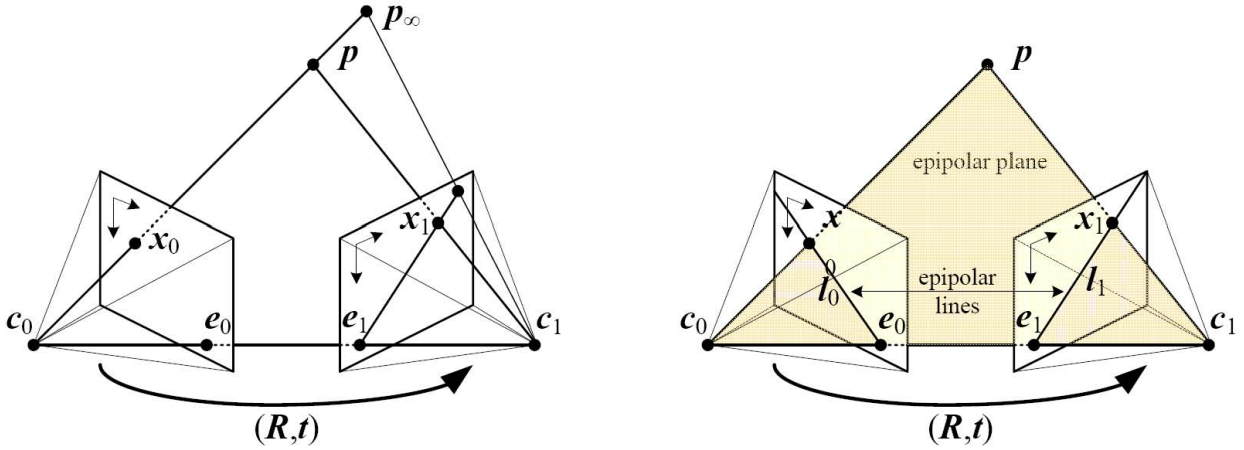

Figure 55: Epipolar geometry. $\mathbf{c}$ camera center; $\mathbf{e}$ epipole. $\mathbf{R}$ and $\mathbf{t}$ are the rotation and translation parameters as introduced in Table 2.
Left: epipolar line segment corresponding to one ray.
Right: corresponding set of epipolar lines and their epipolar plane. [Source: Szeliski 2011; Fig 11.3]

The epipolar constraint in algebraic form:

$$\mathbf{x}_0^T F \mathbf{x}_1 = 0, \tag{37}$$

where T stands for the transpose. Matrix F is called the **fundamental matrix**, a slightly misleading name widely used for historical reasons; more appropriate names like **bifocal matrix** are used by others.
SHB p578 Epipolar geometry has seven degrees of freedom. The epipoles $\mathbf{e_0}, \mathbf{e_1}$ in the image have two coordinates each (giving 4 degrees of freedom), while another three come from the mapping of any three epipolar lines in the first image to the second. Alternatively, we note that the nine components of F are given up to an overall scale and we have another constraint $\det F = 0$, yielding again $9 - 1 - 1 = 7$ free parameters.

The correspondence of seven points in the left and the right image allows the computation of the fundamental matrix F using a non-linear algorithm [Faugeras et al.,1992], known as the **seven-point** algorithm. If eight points are available, the solution becomes linear and is known as the **eight-point** algorithm.

Calibration In order to setup up a stereo correspondence system, we need to calibrate the two cameras to a reference pattern. The reference pattern is of known size and easy to detect. A commonly used pattern is a checkerboard pattern presented on a piece of paper (wiki Chessboard_detection). From those we obtain their so-called *world coordinates*, the coordinates in the 3D space. With those in turn we obtain the

rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$, see again Fig. 55. Matlab has an example for doing all that, which we have rewritten according to our code notation, **P.20**. It is based on the correspondence of local features as introduced in Section 12.

A list of calibration tool boxes can be found here:
http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/links.html

Correspondence The correspondence can be calculated with different methods - not only with features as suggested by the code example. If the baseline is short, then low-level methods are considered better; if the baseline is long/wide, high-level methods are preferred (see also E):

- Low-level, correlation-based, bottom-up, dense correspondence
- High-level, feature-based, top-down, sparse correspondence. Features are for instance the local features as introduced in Section 12; as in the code example.

22.1.2 Shape from X

Sze p580, s12.1

SHB p551, s11.1.2, pdf588

Shape from X is a generic name for techniques that aim to extract shape from intensity images. Many of these methods estimate local surface orientation (e.g., surface normal) rather than absolute depth. If, in addition to this local knowledge, the depth of some particular point is known, the absolute depth of all other points can be computed by integrating the surface normals along a curve on a surface [Horn, 1986]. There exist:

- from Stereo: as mentioned above.
- from Shading: see below.
- from Photometric Stereo: a way of making 'shape from shading' more reliable by using multiple light sources that can be selectively turned on and off.
- from Motion: also known as Structure-from-Motion, abbreviated SfM or SFM (wiki Structure_from_motion).
- from Texture: see below.
- from Focus: is based on the fact that lenses have finite depth of field, and only objects at the correct distance are in focus; others are blurred in proportion to their distance. Two main approaches can be distinguished: shape from focus and shape from de-focus.
- from Contour: aims to describe a 3D shape from contours seen from one or more view directions.

FoPo p89, s2.4

Dav p398, s15.4

Shape from Shading **Albedo** (latin for "whiteness"): or reflection coefficient, is the diffuse reflectivity or reflecting power of a surface. It is the ratio of reflected radiation from the surface to incident radiation upon it. Its dimensionless nature lets it be expressed as a percentage and is measured on a scale from zero for no reflection of a perfectly black surface to 1 for perfect reflection of a white surface.

Algorithm Most shape from shading algorithms assume that the surface under consideration is of a uniform albedo and reflectance, and that the light source directions are either known or can be calibrated by the use of a reference object. Under the assumptions of distant light sources and observer, the variation in intensity (irradiance equation) become purely a function of the local surface orientation, which is used for instance for scanning plaster casts. In practice, surfaces are rarely of a single uniform albedo. Shape from shading therefore needs to be combined with some other technique or extended in some way to make it useful. One way to do this is to combine it with stereo matching (Fua and Leclerc 1995) or known texture (surface patterns) (White and Forsyth 2006). The stereo and texture components provide information in textured regions, while shape from shading helps fill in the information across uniformly colored regions and also provides finer information about surface shape.

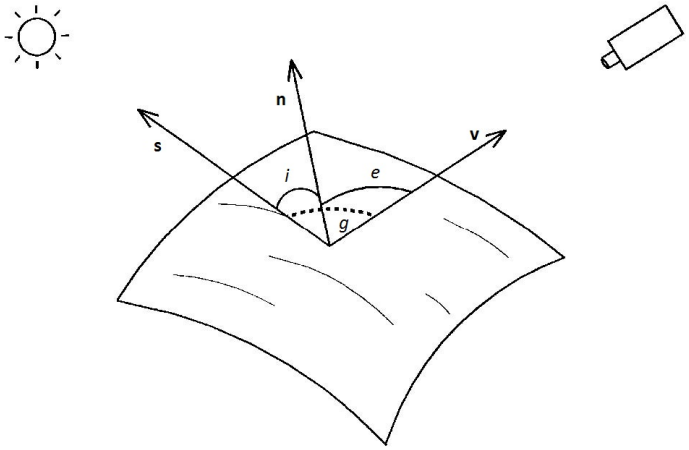

Figure 56: Geometry of reflection. An incident ray from source direction **s** is reflected along the viewer direction **v** by an element of the surface whose local normal direction is **n**; *i*, *e*, and *g* are defined respectively as the incident, emergent, and phase angles. [Source: Davies 2012; Fig 15.8]

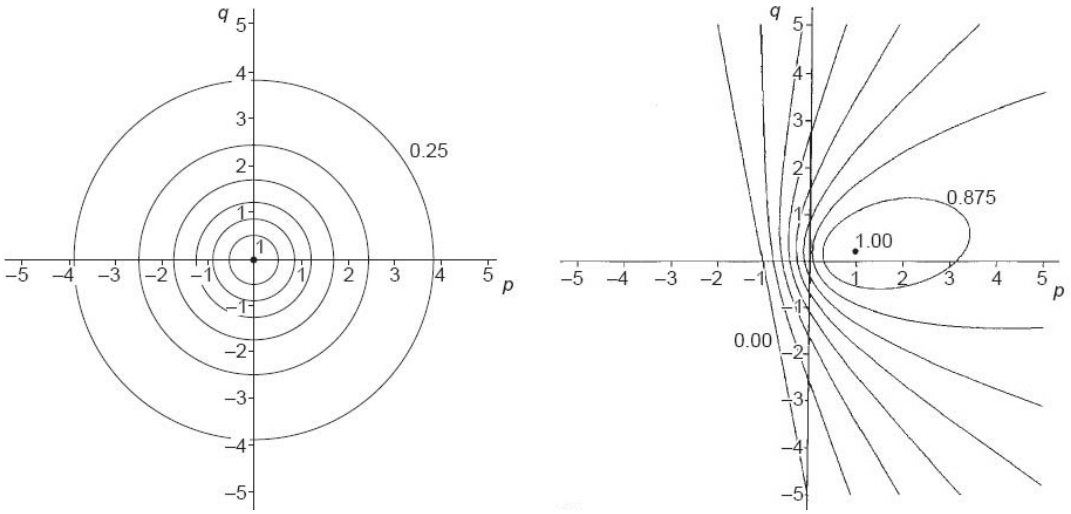

Figure 57: Reflectance maps for Lambertian surfaces: **Left**: contours of constant intensity plotted in gradient (p, q) space for the case where the source direction **s** (marked by a black dot) is along the viewing direction **v** (0, 0) (the contours are taken in steps of 0.125 between the values shown); **Right**: the contours that arise where the source direction (p_s, q_s) is at a point (marked by a black dot) in the positive quadrant of (p, q) space: note that there is a well-defined region, bounded by the straight line $1 + pp_s + qq_s = 0$, for which the intensity is zero (the contours are again taken in steps of 0.125). [Source: Davies 2012; Fig 15.9]

FoPo p217, s6.5
Sze p510, s12.1.2
SHB p613, s12.1.2

Shape from Texture The variation in foreshortening observed in regular textures can also provide useful information about local surface orientation. Figure 58 shows an example of such a pattern, along with the estimated local surface orientations. Shape from texture algorithms require a number of processing steps, including the extraction of repeated patterns or the measurement of local frequencies in order to compute local affine deformations, and a subsequent stage to infer local surface orientation. Details on these various stages can be found in the research literature (Witkin 1981; Ikeuchi 1981; Blostein and Ahuja 1987; Garding 1992; Malik and Rosenholtz 1997; Lobay and Forsyth 2006).

22.2 Viewpoint Estimation, Pose Estimation

Viewpoint estimation is the task of localizing the precise position of the observer in space by comparing the present viewpoint with previously stored viewpoints. Taking our illustration in Fig. 3 the challenge is as follows: we have seen views ‘a’ and ‘b’ in a learning phase, then we are confronted with ‘c’ in a testing

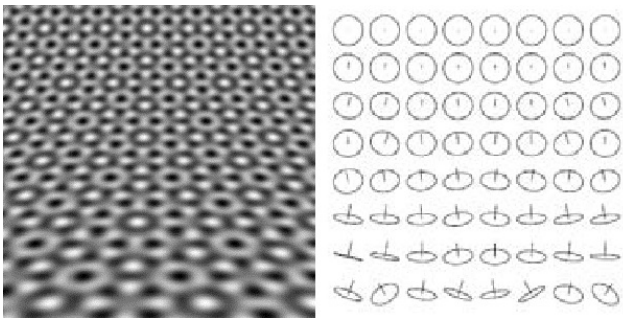

Figure 58: Shape from texture.
Left: the texture.
Right: corresponding estimations of the local surface normal.
[Source: Szeliski 2011; Fig 12.3]

phase: are we able to infer from which point in 3D space we observe 'c'? A more practical example is: we recognize in a photo a known landmark (i.e. a famous monument), but the shot was taken from a novel viewpoint, a viewpoint from which we have not seen the landmark before: can we figure out the new viewpoint from our previously seen viewpoints? We can invert the problem formulation and say we estimate the *pose* of the object in space for which we require a different reference point than the one of the current observer.

In Computer Vision, the traditional approach to perform this localization is to detect features between the two photographs and then reconstruct the motion in order to make an estimate as precise as possible, as introduced in Sections 12 and 21, respectively. This is computationally intensive both in calculation as in memory requirements. A recent DNN has however achieved stunning results by solving that task quicker and with less memory, called PoseNet. It is a network that does not allow transfer learning, as the task per se is rather an identification task and it requires complete relearning for each new location (neighborhood).

23 Classifying Motions, Action Recognition

Classifying motions is the challenge to understand a sequence of individual motions. Recognizing human motions is also referred to as *action recognition*, a broad term that can be subdivided into gesture recognition, human-object interaction and human-human interaction.

Recognizing such motions means tracking the individual motion components and describing the resulting trajectories. This is doable for hand-gestures made on the cell phone (Section 23.1), but becomes very complex for motions against a heterogeneous background. The challenge is the proper segragation of the motion from its background. For the tracking task as introduced in Section 19, this segragation was a lesser problem as the objects in the scene are relatively small and embedded in a homogeneous background - at least phase-wise and locally. But when following human motions, one typically zooms into the silhouette for reasons of resolution, and in that situation chances are higher that the gesture's background is heterogeneous, which in turn makes segragation more difficult: think of the background that a laptop camera faces when the user sits in his room at home. For that reason one often seeks additional information by using depth camera (Appendix A), but even with that depth information segragation remains challenging sometimes. It is therefore not surprising that recognizing human motions works best on a homogeneous background with fairly constant illumination. Skeptical voices even say that there still does not exist a convincing implementation of motion classification.

Systems solving action recognition tasks are complex systems - though there exist exceptions such as the Kinect user interface (Section 23.2). The Kinect interface relies on a depth sensor however. If one uses only RGB cameras as input, then the presently best performing systems are Deep Nets. They are trained on a variety of databases and there exists no comparison reference as there is for classification models (Section 8.3). In Section 23.3 we briefly mention those systems.

23.1 Gesture Recognition

The most common gestures that are analyzed are hand gestures and facial gestures, the latter often reflecting emotions. Hand motions are recognized for the purpose of human-computer interaction. A daily example is the unlocking of a cell phone. This works well because one tracks the finger tip only on a set of predefined positions aligned on an orthogonal grid. Motion tracking becomes much more challenging for recognizing hand motions in the air. The automotive industry tries to make gesture recognition work for the control of the dash board.

23.2 Body Motion Classification with Kinect (Depth Sensor)

FoPo p476, s14.5, pdf446

Kinect is a video game technology developed by Microsoft that allows its users to control games using natural body motions. Kinect's sensor delivers two images: a 480x640 pixel depth map and a 1200x1600 color image. Depth is measured by projecting an infrared pattern, which is observed by a black-and-white camera. The two main features of this sensor are its speed - it is much faster than conventional range finders using mechanical scanning - and its low cost (only ca. 200 Euros).

Range images have two advantages in this context:

1) They relatively easily permit the separation of objects from their background, and all the data processed by the pose estimation procedure presented below, is presegmented by a separate and effective background subtraction module.
2) They are much easier to simulate realistically than ordinary photographs (no color, texture, or illumination variations). In turn, this means that it is easy to generate synthetic data for training accurate classifiers without overfitting.

23.2.1 Features

Kinect features simply measure depth differences in the neighborhood of each pixel. Concretely, let us denote by $z(\mathbf{p})$ the depth at pixel $\mathbf{p}$ in the range image. Given image displacements $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$, a scalar is

computed as follows:

$$f_{\boldsymbol{\lambda},\boldsymbol{\mu}}(\mathbf{p}) = z\Big[\mathbf{p} + \frac{1}{z(\mathbf{p})}\boldsymbol{\lambda}\Big] - z\Big[\mathbf{p} + \frac{1}{z(\mathbf{p})}\boldsymbol{\mu}\Big] \tag{38}$$

In turn, given some allowed range of displacements, one can associate with each pixel $\mathbf{p}$ the feature vector $\mathbf{x}(\mathbf{p})$ whose components are the D values of $f_{\boldsymbol{\lambda},\boldsymbol{\mu}}(\mathbf{p})$ for all distinct unordered pairs $(\boldsymbol{\lambda}, \boldsymbol{\mu})$ in that range.

As explained below, these features are used to train an ensemble of decision tree classifiers, in the form of a random forest. After training, the feature $\mathbf{x}$ associated with each pixel of a new depth map is passed to the random forest for classification.

23.2.2 Labeling Pixels

The objective is to construct a classifier that assigns to every pixel in a range image one out of a few body parts, such as a person's face, left arm, etc. There are 10 main body parts in Kinect (head, torso, two arms, two legs, two hands, and two feet), some of which are further divided into sub-parts, such as the upper/lower and left/right sides of a face, for a total of 31 parts.

Random Forest Research in classification has shown that some data sets are better classified with multiple classifiers, also called ensemble classifiers. In this case, the pixel/bodypart classification takes place with multiple tree classifiers, a random forest, see figure 59.

The feature $\mathbf{x}(\mathbf{p})$ (from above) is passed to every tree in the forest, where it is assigned some (tree-dependent) probability of belonging to each body part. The overall class probability of the pixel is finally computed as an average probability of the different trees.

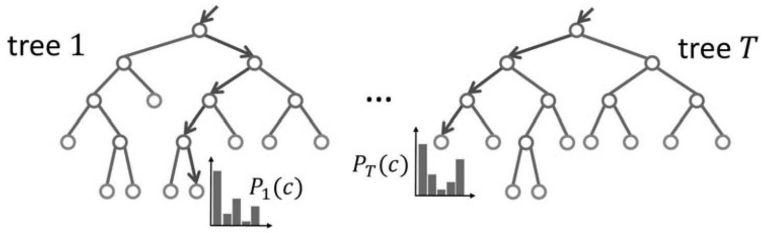

Figure 59: Random Forest: multiple trees are used to classify the same input and pool their decisions.

Creation of Training Set The primary source is a set of several hundred motion capture sequences featuring actors engaged in typical video game activities such as driving, dancing, kicking, etc. After clustering close-by pictures and retaining one sample per cluster, a set of about 100K poses is obtained. The measured articulation parameters are transferred (retargeted) to 15 parametric mesh models of human beings with a variety of body shapes and sizes. Body parts defined manually in texture maps are also transferred to these models, which are then skinned by adding different types of clothing and hairstyle, and rendered from different viewpoints as both depth and label maps using classical computer graphics techniques. 900k labeled images in total were created this way.

Training Classifier The classifier is trained as a random forest, using the features described above, but replacing the bootstrap sample used for each tree by a random subset of the training data (2,000 random pixels from each one of hundreds of thousands of training images).

The experiments described in Shotton et al. (2011) typically use 2,000 pixels per image and per tree to train random forests made of 3 trees of depth 20, with 2,000 splitting coordinates and 50 thresholds per node. This takes about one day on a 1,000-core cluster for up to one million training images. The pixelwise classification rate is ca. 60% (error of 40%!), which may appear as low but chance level is much lower (what is it?).

23.2.3 Computing Joint Positions

The random forest classifier assigns to each pixel some body part, but this process does not directly provide the joint positions because there is no underlying kinematic model. Instead, the position of each body part k could (for example) be estimated as some weighted average of the positions of the 3D points corresponding to pixels labeled k, or using some voting scheme. To improve robustness, it is also possible to use mean shifts to estimate the mode of the following 3D density distribution:

$$f_k(\mathbf{X}) \propto \sum_{i=1}^{N} P(k|\mathbf{x}_i) A(\mathbf{p}_i) e^{-\frac{1}{\sigma_k^2}\|\mathbf{X}-\mathbf{X}_i\|^2}, \tag{39}$$

where $\mathbf{X}_i$ denotes the position of the 3D point associated with pixel $\mathbf{p}_i$, and $A(\mathbf{p}_i)$ is the area in world units of a pixel at depth $z(\mathbf{p}_i)$, proportional to $z(\mathbf{p}_i)^2$, so as to make the contribution of each pixel invariant to the distance between the sensor and the user. Each mode of this distribution is assigned the weighted sum of the probability scores of all pixels reaching it during the mean shift optimization process, and the joint is considered to be detected when the confidence of the highest mode is above some threshold. Since modes tend to lie on the front surface of the body, the final joint estimate is obtained by pushing back the maximal mode by a learned depth amount.

The average per-joint precision over all joints is 0.914 for the real data, and 0.731 for the synthetic one (tolerance of 0.1m). Thus, the voting procedures are relatively robust to 40% errors among individual voters. The synthetic precision is lower due to a great variability in pose and body shape. In realistic game scenarios, the precision of the recovered joint parameters is good enough to drive a tracking system that smoothly and very robustly recovers the parameters of a 3D kinematic model (skeleton) over time, which can in turn be used to effectively control a video game with natural body motions.

Closing Note

The final component of the system is a tracking algorithm whose details are proprietary but, like any other approach to tracking, it has temporal information at its disposal for smoothing the recovered skeleton parameters and recovering from joint detection errors.

23.3 Action Recognition on RGB Frames

Deep Nets for action recognition are based on the type of convolution networks as introduced in 8.3, but are modified to include time. The two-dimensional convolution is extended to a third dimension representing time but the principles of learning remain the same. As one now operates on a third dimension, training and applying such a network becomes extremely time-intensive; training therefore occurs on lower image sizes than for classification of static images. We merely explain how such video classification networks are applied in PyTorch. Such networks are trained on databases such as:

Kinetics-400, https://deepmind.com/research/open-source/kinetics: 700 action categories, each category has 600 sequences or more, each one lasting 10 seconds approximately.

UCF 101, https://www.crcv.ucf.edu/data/UCF101.php: 101 categories.

HMDB 51, http://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database/: 51 action categories.

In Python (version 1.2) three types of networks are offered to classify motion sequences, placed in module `torchvision.models.video`. They were all trained on the first data set, the Kinetics-400 set, which originally had 400 classes, but meanwhile contains 700. We can load such a network with its pretrained weights, as is done for any other network (in classification, segmentation or object recognition), i.e.:

```
import torchvision.models.video as ModVid
NET     = ModVid.r3d_18(pretrained=True)
```

To load a video we can use the reader function from the module `torchvision.io`. This function loads the video directly into data type `torch`, of format:

$$\text{Video Tensor} = [\text{\# frames} \times \text{\# chromatic channels} \times y \times x] \tag{40}$$

with y and x being frame height and frame width. As the video is already in torch type, we cannot apply the typical transformation functions anymore as those expect the `PIL` format. Instead we need to write preparatory functions. An example is shown in Appendix **P.22**.

24 Grouping

Grouping is the process of associating features into potential groups, that outline an object, scene part or that constitute some pattern. Grouping processes have been developed for different tasks. There are two scenarios where it has been particularly successful:

Pose Estimation in Machine Vision: Some robots in industry grasp for specific material and tools. Those objects are limited in variability and their shape is known in advance, and the scene illumination is controlled. In this typical machine vision scenario, grouping operations serve two purposes: they first identify object candidates; then they are used to estimate the object's pose (see also 22.2). An early successful work on this topic is the one by Lowe, available in book form as well as in its original form, as PhD thesis:
https://www.amazon.com/Perceptual-Organization-Recognition-International-Engineering/dp/089838172X
https://apps.dtic.mil/dtic/tr/fulltext/u2/a150826.pdf

Object Detection in Satellite Images: Contour segments are grouped to find objects such as houses, vehicles, roads etc. Many of those objects are outlined by two parallel contour segments lying vis-a-vis.

Many grouping principles have been proposed, motivated computationally as well as psychologically. The two basic ones are proximity and similarity, see Fig. 60. Features that lie near each other, probably form a group that belongs to some pattern, object or scene part. Features that show visual similarity - be that appearance or geometry - also tend to belong to the same pattern.

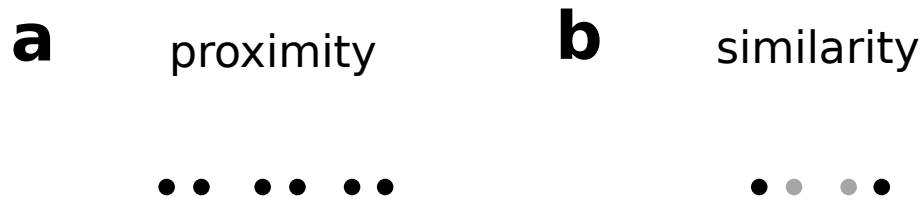

Figure 60: Basic grouping principles (wiki Principles_of_grouping).
a. Features close to each other, tend to belong to each other as part of the same object or scene part or pattern.
b. Features similar in appearance, tend also to group together. Note that in this example the spacing between the points is the same as in **a**.

Grouping becomes richer, when one considers segments, see Fig. 61, because now the feature includes elongation and that leads to a range of conspicuous alignments. For a pair of segments one can consider the following cases:

Ribbon: two parallel segments, closely spaced. An easy way to identify is too determine the spacing between midpoints (see Fig. 44 for terminology).
L-feature: two segments whose endpoints lie near. The angle between the two segments typically shows a minimum and a maximum value: angles below or above those extrema become either a ribbon feature or a colinear alignment.
T-feature: two segments with their intersection point being on either segment.
Colinear: two segments aligned along the same line equation.

How those pairs are defined exactly is a matter of choice and also a matter of context. It is a matter of choice, because if we deal with a specific type of scene, then some features may dominate and one would therefore tailor the grouping operations accordingly. For instance, in machine vision applications in industry, object contours are often smooth and objects exhibit vertices at their corners. Thus focusing on the above suggested list of pair alignments is very helpful in recognition. In satellite imagery however, the range of structures is larger and those suggested alignments are only part of a larger set of features one needs to

consider. It is a matter of context, because in a more random alignment of segments, certain groupings may be conspicuous that would not be considered a group amidst a regular alignment of segments.

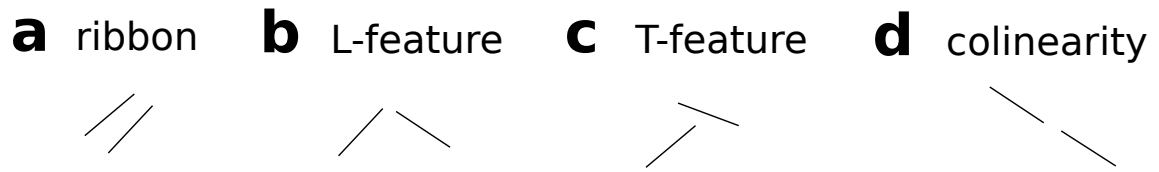

Figure 61: Conspicuous alignments of a pair of segments.
a. approximately parallel and closely spaced.
b. two endpoints lie proximal and the segments form a certain angle.
c. the intersection point of the two segments lies on either one.
d. the two segments share similar line equation and lie in sequence.

Grouping typically starts with an edge map, a binary map containing on-pixels where an edge was detected (Section 17). One could start associating the individual edge pixels without regard for any traced segments, but that is of square complexity and relatively slow; it has however the advantage that it can be very precise and include single pixels: for specific structures, such as straight lines and circles, one may consider the grouping processes as introduced in Section 17.1.

If one intends to search a large number of groups, it is more efficient to work with traced segments and start with a description as introduced in Section 17.4. The most elegant way to perform grouping would be to analyze each pair of segments and determine a proximity. That can however also be complex, for example in large images with more than ca 100k pixels or if we exploit and entire scale space, see Appendix K for the complications. Instead, we will try to make use of Nearest-Neighbor Search.

24.1 Points

We may want to group individual points, be that individual pixels, the midpoint of segments, the center point of regions, etc. As just mentioned above, for large sets of points, nearest-neighbor search is the most efficient way to do that. With array `Pts` our set of coordinates, be that two- or three-dimensional, we search for the nearest points as follows:

```
[NEB DI]  = knnsearch(Pts, Pts, 'k', 3);
```

where `NEB` are the indices to the neighbors and `DI` are the distances. Since this is a search on its own points, the (first) nearest neighbor is own index and we therefore exclude it from the arrays:

```
NEB(:,1)    = [];             % exclude own
DI(:,1)     = [];
IxNN        = NEB(:,1);          % nearest neibor
mnDiNN      = mean(DI(:,1));
```

Variable `IxNN` is the actual, first nearest neighbor. The variable `mnDiNN` is the average nearest neighbor distance. It could be used to define a threshold for grouping pixels. Appendix **P.23.1** gives a full example.

25 Robot Vision

Robot vision is the recognition process for an agent that interacts with its environment: a robot often receives sequences of images as input and in order to efficiently wade through that stream of information, recognition needs to be fast and guided, a challenge similar to identifying object proposals in object recognition (Section 10.5). For static robots in industry, the scenario is often controlled. For example for a robot picking up objects from a conveyor belt, the set of objects is known and has been trained in advance; the illumination is fixed and the background is kept homogeneous to facilitate localizing the objects (see again 'Machine Vision' under Section 1.1). For mobile (dynamic) robots, the difficulties grow immensely and there are two types of challenges in particular: navigation and reliable recognition in face of an ever changing environment.

Navigation In order to move through an environment, a mobile robot needs to have some sort of map. The easiest way is to deploy an existing map, such as satellite-based radio navigation (GPS) for automotive vehicles, radio-frequency identification (RFID) for robots operating in industrial settings (wear houses), etc. In the case of absence of such a predefined map, the map needs to be built. The map building process is complex and comparable to the human situation when one moves to another city or neighborhood, and then starts orienting him/herself (without cell phone): one starts remembering places and landmarks and links them to a representation that allows us to navigate efficiently through our new neighborhood. One can think of different strategies how this map-making process is accomplished (Section 25.3).

Environmental Changes In industrial settings - where Machine Vision methods are deployed -, the environment is kept static: the set of objects that a robot is supposed to manipulate were presented during the training phase - no other objects will appear during application. Recognition in such a scenario is also called *closed-set* recognition because at run-time, the system will not face anything unexpected. However robots operating in outdoor or domestic environments will face a permanently changing environment. There are two classes of changes: changes in (texture) appearance, such as described already for tracking systems (Section 19); and object changes, such as displacement or modification of present objects, disappearance and new appearance of objects, e.g. a domestic robot needs to be prepared that some furniture is being moved (i.e. chairs). The presence of the latter type of (potential) changes, has led to the term *open-set* recognition, the challenge to face novel object classes that were not present during the training phase.

From a methodological point of view, there is nothing novel in terms of representation of structure. Scenes and objects are represented with methods as introduced in previous sections and in the following we merely introduce novel terminology and details on some of the tasks related to visual navigation. Firstly we introduce the process of visual odometry, which helps a robot to understand its heading direction (Section 25.1). Then we explain the task of place recognition, which helps a robot to actually memorize its environment (Section 25.2). Both of these tasks enable the robot to construct a map of its surround (Section 25.3). Finally, we have a few words on Self-Driving Cars (Section 25.4).

Sensors A robot can be equipped with different types of sensors. *Exteroceptive* sensors observe the environment: there exist sonar, rangefinders, cameras and global positioning systems (GPS), see also Appendix A for their characteristics. GPS is useful for a coarse localization but the signal is often absent in urban scenarios due to the presence of large trees, high buildings, tunnels, indoors, etc; it is absent on other planets. Rangefinders are a convenient method to measure depth, but are expensive. Cameras are cheap, light and provide rich information, but one copes with motion blur and the above mentioned appearance changes.

Proprioceptive sensors can be encoders, accelerometers and gyroscopes; the latter two are sometimes combined into a *inertial measurement unit* (IMU). They allow to measure velocity, position change and acceleration. These measurements allow obtaining an incremental estimate of the robots movements by means of *dead-reckoning*, the navigation technique to predict the immediate route using the presently measured values. Due to their inherent noise they are not sufficient for accurate (long-term) path integration.

Because proprioception is not reliable and GPS signal is occasionally absent, accurate odometry therefore requires visual odometry, coming up next.

25.1 Visual Odometry (VO)

wiki Visual_odometry

Visual odometry is the process of estimating the egomotion of an agent (robot, vehicle, human) for the purpose of path integration, based solely on visual input, without using proprioceptive sensors (i.e. without an IMU). The essential techniques are the calculation of optic flow to estimate the heading direction (Section 20); its optimization using tracking techniques such as Kalman and Particle filters (Section 19.5); and the integration of the estimates to reconstruct the covered route. Because those successive motion estimates are not perfect due to several noise sources, they accumulate an error that expresses itself as *drift*, the offset of the estimated trajectory from the real route (Fig. 62). This drift can be corrected in various ways; the simplest way is to apply - if present - another navigation device (GPS), a depth camera (laser) or an IMU. In case of absence of such aid, one analyzes the sequence of recorded frames using a method called *sliding window bundle adjustment* or simply *bundle adjustment* (wiki Bundle_adjustment).

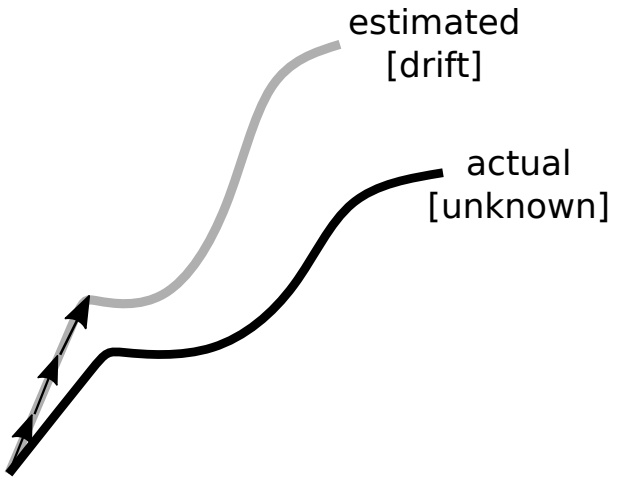

Figure 62: Drift during visual odometry. Actual driven route is shown in black. Path integration consist of estimating the ego-motion between key-frames (see arrows), but includes the accumulation of small errors that result in a drift, the difference between actual and estimated route (latter in gray). Drift is corrected by methods such as bundle adjustment (or GPS if available).

The computations for visual odometry improve if one selects *keyframes*, for both the gradual integration and the bundle adjustment. The reason is that the estimation becomes more robust if some frames are dropped; for that reason frame-by-frame optic flow is considered inept by some researchers and motion estimates are carried out with a feature-based correspondence scheme (Appendix E).

Visual odometry works best when scenes are static, i.e. no object movement, because they make estimation of egomotion difficult; when scenes are sufficiently illuminated in order to find visual cues that allow establishing correspondence; and when scenes show enough texture, also for the purpose to establish image correspondence.

Visual odometry is considered a particular case of Structure-from-Motion (Section 22.1.2). A tutorial by Scaramuzza and Fraundorfer (2011) introduces that topic very smoothly.

25.2 Place Recognition

Place recognition is the ability to re-identify a previously seen place or scene (Section 2.1). The exact definition of place is dependent on context: for an automobile a place can be an intersection; for a household robot it can be a room or perhaps only a corner of a room. It is in fact not straightforward to decide when the scene has sufficiently changed in order to label it as a new place: location '2' in front of the tree might be considered very different from the location '0' in front of the house; would location '1' be also different? It is similar to the issue of *change-point* detection in video segmentation.

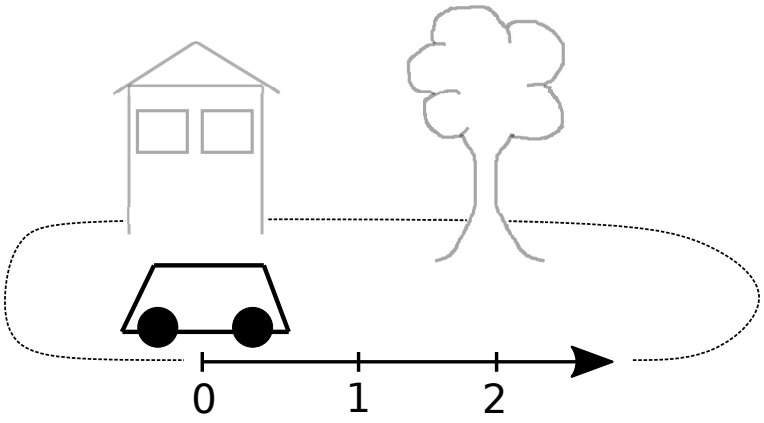

Figure 63: An intelligent, moving robot will remember the route it has covered and integrate the path using methods from visual odometry. Later it will re-identify the places it has visited before (place recognition) and then link its starting point with the revisited places (loop closure) to form a topological map of its environment. The prevailing concept to accomplish this is by a framework called 'Simultaneous Localization and Mapping' (SLAM).

Place recognition has been tried with a large variety of methods. They can be broadly divided into global descriptors and those from the Local Feature approach. Global descriptors describe the image as a whole and can be as simple as histograming intensity values as mentioned already in Section 4; exploit global thresholding as introduced in Section 14.1; but can be any imaginable description that generates the fixed-size image vector (Section 3.3). A popular descriptor is the so-called GIST descriptor that is based on filtering the image with filters as presented in Section 7.3.2, review Fig. 20 for brevity.

The Local Feature approach was introduced in Sections 12 and 13. Recent efforts have also started to exploit Deep Learning. As of now, it is not clear whether it is really outperforming the other approaches, as Deep Nets appear better suited for abstraction than for identification.

What makes place recognition challenging is the ever-present appearance changes. For place recognition in outdoor scenes, it has been proposed to use three to four scene captures under different illumination conditions, in order to reliably recognize a scene. Another challenge is to cope with different viewpoints: one can enter a place from different directions resulting in relatively different scenes (perspectives). When scenes get confused based on solely their visual properties, then one talks of *Perceptual Aliasing*. Proper place recognition is important otherwise the robot is not able to construct a correct map and it will get quickly lost.

Literature The most comprehensive review is by Lowry et al. More literature will be mentioned in the next Section.

Lowry et al., 2016. Visual Place Recognition: A Survey

25.3 Map Construction (SLAM)

When a robot is set off to explore its environment and starts building a map of its surround, then that requires two problems to be solved: localization and mapping. Localization is the process to identify its position in space; mapping is the process to relate the surround to the map. The two processes are intertwined: localization is difficult to achieve without a proper map, mapping is hard to achieve without accurate localization; it is a chicken-egg challenge. For that reason, it is assumed that the optimal approach is the simultaneous use of both processes, which has led to the term *Simultaneous Localization and Mapping* (SLAM), wiki Simultaneous_localization_and_mapping. There are innumerous attempts to realize this concept; a solution depends on the intended operation range of the robot, its deployed sensors, its operation speed, etc. When the process is designed for visual input only, it is referred to as visual SLAM or *v-SLAM*. It is

perhaps the most complex issue introduced so far (in this discourse on computer vision) as it involves both recognition and 3D reconstruction - essentially everything introduced until now.

The map is constructed by associating places, very much like what humans do: think of how we explain heading directions to each other using conspicuous cues of the environment i.e. T-intersection, white-painted house, narrow street, etc. This will result in a *topological* map whose arrangement corresponds to a graph (made of vertices and edges). A topological map can be suitable for an individual (robot or human), but is inept for communication to other individuals. For that reason it is preferred to endow a map with metric information, resulting in a *metric* map. Humans also use metric maps to communicate heading directions effectively on a large scale, i.e. city maps, geographic maps, etc.

When a roaming robot recognizes a previously seen place, then that is used to create a so-called *loop closure* into the map - speaking in graph terminology: a vertex added to the graph and its corresponding edges are inserted. For example, after a robot has circled once around the block (Fig. 63), it should recognize its starting point, such that it can add the proper vertex to its growing map.

Proper map construction requires a lot of parameters to be set and the complexity of tuning can be so daunting, that some researchers have labeled map construction also the “curse of manual tuning”, in reference to the term “curse of dimensionality” in the field of Pattern Recognition. For that reason some researchers have turned toward reinforcement learning as a method to partially self-tune such systems.

Literature The most recent review is by Younes et al. 2017, which focuses in particular on single-camera SLAM. For older approaches and reviews, a good starting point might by the tutorial by Durrant-Whyte and Bailey (2006). The review by Fuentes-Pacheco et al. (2015) contains links to large datasets; the one by Garcia-Fidalgo and Ortiza discusses differences in the exact use of descriptors.

Younes, G., et al., 2017. Keyframe-based monocular SLAM: design, survey, and future directions
Fuentes-Pacheco et al., 2015. Visual simultaneous localization and mapping: a survey
Durrant-Whyte and Bailey, 2006. Simultaneous Localization and Mapping: Part I/II
Garcia-Fidalgoa and Ortiza, 2015. Vision-based topological mapping and localization methods: A survey

Training Sets If one intends to practice navigation in a small environment, then the following indoor scenes might be a good choice:
https://www.nada.kth.se/cas/IDOL/.

25.4 Self-Driving Cars

wiki Self-driving_car Dav p636, ch23

Self-Driving Cars rely on existing maps for reason of efficiency; a full map construction during driving would be unfeasible. Those maps are created before the operation of the car, termed *off-line*, marked as ‘Offline Maps’ in Fig. 64. Existing maps can be those as created by professional map providers such as TomTom, HERE, Google, Amazon, etc., but also on collaborative projects such as OpenStreetView. Maps can also be generated by driving all routes with a prototype of the self-driving car to obtain maps that correspond to the sensors, a scouting/reconnaissance process in some sense. The type of maps can be as diverse as the sensors themselves: RGB, depth from stereo, depth from lidar, depth from radar, etc. Such predefined maps allow for localization accuracy that a GPS cannot provide; the GPS is deployed only for coarse localization.

When a car is in operation, operating *on-line* as it is called, the localizer module compares its percept with its corresponding maps to localize itself as accurately as possible; for some methods and maps, the accuracy can be as little as few centimeters. The mapper module updates the maps with observed changes. A typical autonomous car has two other import modules: a traffic-signalization detection system (TDS) and a moving-objects tracking (MOT) system. We have a few words on some of these modules below.

Since cars move fast, implementations of any of those perception processes have to be optimized for speed. Efficient object search becomes essential. Deep Nets struggle to keep up with that speed: general segmentation processes as introduced in Section 9 might be useful for a coarse localization and for generating object proposals, i.e. when GPS is absent or when the car takes a sharp turn in a city resulting in rapid

scene changes. More successful are dedicated Deep Nets for detection of a single class or a few classes, that appear in certain locations of the scene.

Due to the huge competition between auto-manufacturers, it is hard to identify best performing algorithms - training data sets are handed out reluctantly. It can be assumed that the classification part is a combination of different methodologies in the sense of ensemble classification (Appendix H.1.5).

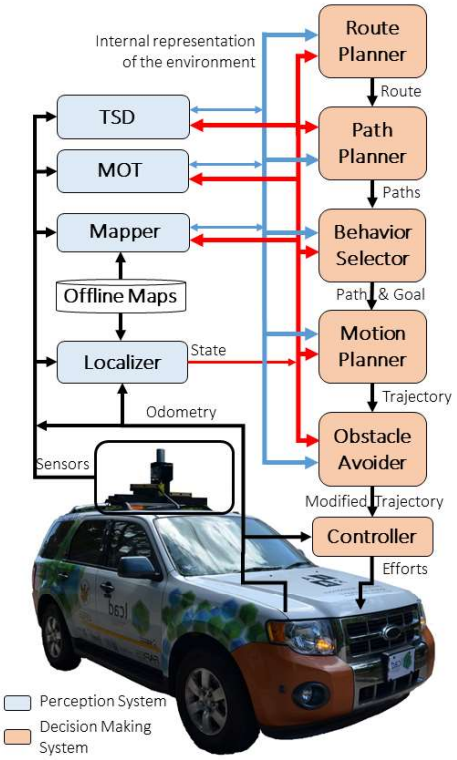

Figure 64: Overview of the typical hierarchical architecture of self-driving cars (TSD = Traffic Signalization Detection; MOT = Moving Objects Tracking). [Source: Badue et al. 2019, Survey Self-Driving Cars]

25.4.1 Localization

LIDAR sensors offer measurement accuracy and easiness of processing but are pricey; camera-based localization approaches are cheap and convenient, but typically less precise and/or reliable. Some mixed approaches use LIDAR data to build the map, and camera data to estimate the car's position relative to the map. Some camera-only approaches use stereo vision to obtain depth; some studies have suggested that observing vertical structures is of use, such as lamp poles, traffic sign poles, tree trunks, etc.

The localization of the roadway is for example addressed with multilevel thresholding (Section 14.1); or with vanishing point detection using RANSAC on edge points (see Figure 65). The most comprehensive review on this topic is by Hillel et al.

Hille et al., 2014. Recent progress in road and lane detection: a survey

25.4.2 Traffic-Signalization Detection (TDS)

Signalization includes traffic lights, traffic signs, pavement markings, etc.

- Traffic light recognition involves their detection and the identification of their state (red, green, and yellow). Deep Net detectors tend to be more robust to over-exposure, color distortions, occlusions, etc. than any

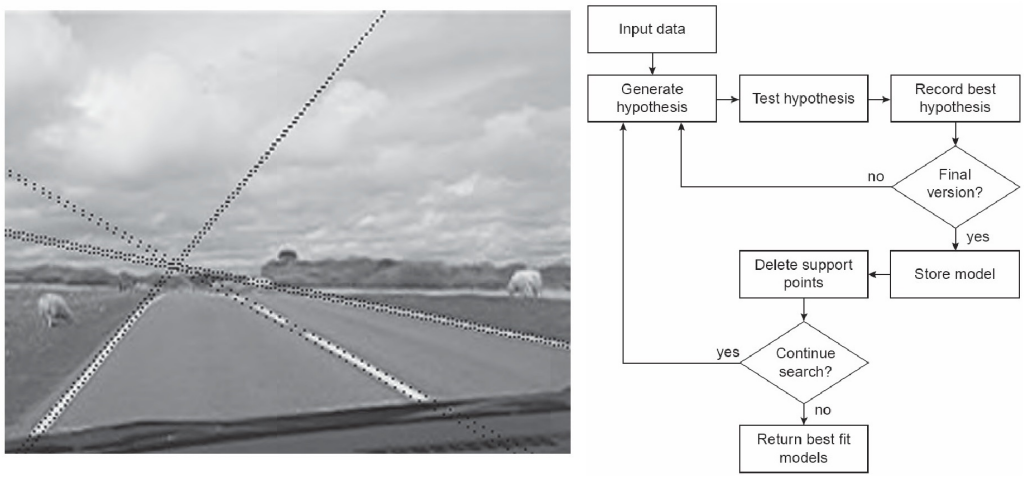

Figure 65:
Left: As the lane markings converge to a point on the horizon, the vanishing point, it is useful to determine that point using RANSAC (Section 21.3), applied to hypotheses of straight lines drawn through detected edge points.
Right: Flow chart of a lane detector algorithm.
[Source: Davies 2012; Fig 23.2], [Source: Davies 2012; Fig 23.4]

other approach; most likely because they store all these conditions with their innumerous weights. A dataset and associated publications can be found here:

http://cvrr.ucsd.edu/LISA/lisa-traffic-sign-dataset.html

- Traffic sign recognition is best performed with a Deep Net, that is, it has been demonstrated to be the clear winner. The largest collection with 100k images is:

https://cg.cs.tsinghua.edu.cn/traffic-sign/

25.4.3 Moving-Objects Tracking (MOT)

The detection and tracking of vehicles, pedestrians, motor cycles and cyclists is the foremost issue as those objects are the most frequent. The principal approaches to MOT were introduced previously (Sections 10, 11 and 19); in the following we mention some techniques that might be old, but perhaps still existent in systems as part of ensemble classification:

- Locating Vehicles: there are two very simple tricks to detect cars in 2D gray-scale images:
  a) shadow induced by vehicle: Importantly, the strongest shadows are those appearing beneath the vehicle, not least because these are present even when the sky is overcast and no other shadows are visible. Such shadows are again identified by the multilevel thresholding approach (Section 14.1).
  b) symmetry: The approach used is the 1-D Hough transform, taking the form of a histogram in which the bisector positions from pairs of edge points along horizontal lines through the image are accumulated. When applied to face detection, the technique is so sensitive that it will locate not only the centerlines of faces but also those of the eyes. Symmetry works also for plant leaves and possibly other symmetric shapes.

- Locating Pedestrians: We have introduced a pedestrian detection algorithm already in Section 10.3, but here we mention some other techniques that ensure a high recognition accuracy:
  a) detection of body parts, arms, head, legs. The region between legs often forms an (upside-down) V feature.
  b) Harris detector (Section 12.1) for feet localization.
  c) skin color (see also Appendix N.1).

Figure 66: Detecting cars by exploiting the symmetry of vertical segments. [Source: Davies 2012; Fig 23.6]

While detection of vehicles and pedestrians is perhaps manageable, the real challenge is the detection and tracking of unexpected or unusual movement, a baby-carriage, a ball rolling toward the street - lost from a child play ground; an approaching kangaroo (unusual trajectory), etc. It is hard to beat the experience that a human has acquired throughout his/her life time.

26 More Domains and Tasks

We touch on some domains and their tasks to illustrate their key issues and to provide some pointers to on-going research.

26.1 Video Surveillance

Dav p578, ch22

Surveillance is useful for monitoring road traffic, monitoring pedestrians, assisting riot control, monitoring of crowds on football pitches, checking for overcrowding on underground stations, and generally is exploited in helping with safety as well as crime. We already mentioned some aspects of surveillance (e.g. in Sections 19 and 14). Here we round the picture by adding some more aspects and by giving some examples.

The Geometry The ideal camera position is above the pedestrian, at some height H_c, and the camera's optical axis has a declination (angle δ) from the horizontal axis (see Fig. 67). This is simply the most suitable way to estimate the distance and height H_t of the pedestrian, thereby exploiting triangulation and the knowledge of where the position of the pedestrian's feet.

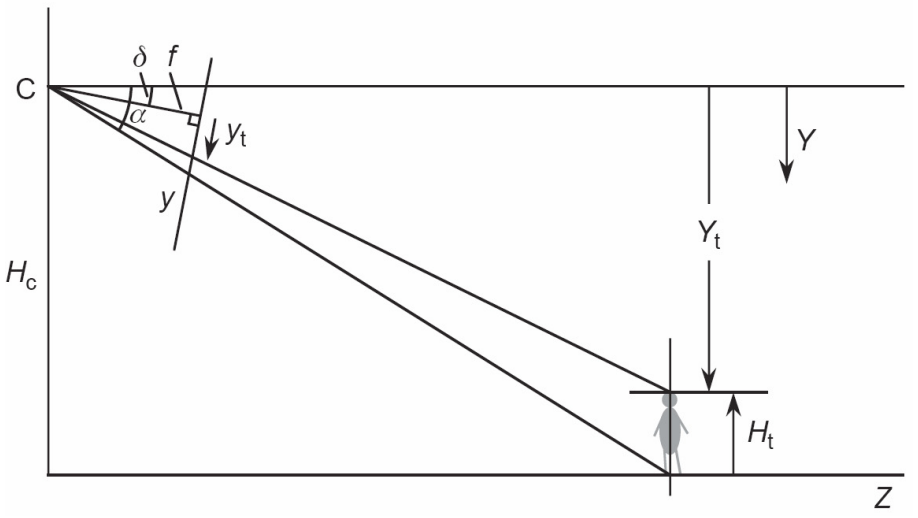

Figure 67: 3-D monitoring: camera tilted downwards. δ is the angle of declination of the camera optical axis. [Source: Davies 2012; Fig 22.2]

Vehicle License Plate Detection License plate recognition is a challenging task due to the diversity of plate formats and the nonuniform outdoor illumination conditions during image acquisition. Therefore, most approaches work only under restricted conditions such as fixed illumination, limited vehicle speed, designated routes, and stationary backgrounds. Algorithms (in images or videos) are generally composed of the following three processing phases:

1) Extraction of a license plate region. An effective method is described in Figure 68.
2) Segmentation of the plate characters. Thresholding as described previously (Section 14.1)
3) Recognition of each character, i.e. using a Deep Neural Network.

Under the name "Anagnostopoulos" one can find a review on this topic on GoogleScholar.

Classifying Traffic Participants To distinguish between traffic participants, such as car, pedestrian, truck, bicycle, one can take a Deep Neural Network. A useful database to verify the localization and classification performances can be found here:

http://podoce.dinf.usherbrooke.ca/challenge/dataset/

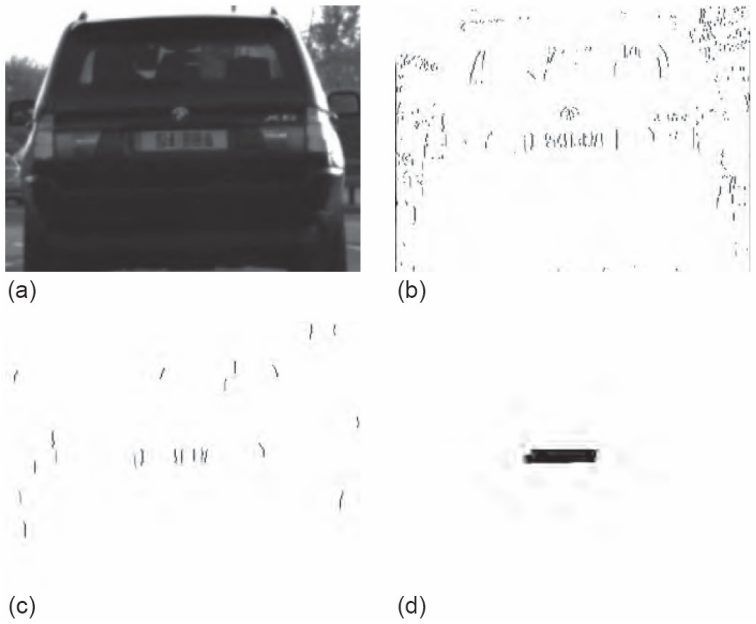

Figure 68: Simple procedure for locating license plates. (a) Original image with license plate pixellated to prevent identification. (b) Vertical edges of original image (e.g. with Sobel detector). (c) Vertical edges selected for length. (d) Region of license plate located by morphological operations, specifically horizontal closing followed by horizontal opening, each over a substantial distance (in this case 16 pixels). [Source: Davies 2012; Fig 22.13]

Vehicle (Model) Identification For make/model identification, many systems have focused on analyzing the car's front, which can be regarded as its face. A typical system locates the license plates first and then measures other front features in relation to it, such as head lights, radiator grill, etc.:

Figure 69: Make and model identification using the fronts features and their relations to the license plate.

So far, (exploratory) studies have used up to 100 auto model tested on still images. Scholar google 'Pearce and Pears' for a recent publication.

26.2 Text Recognition

For this task, one generally distinguishes between two types of applications, optical character recognition (OCR) and recognition of text in the wild. In optical character recognition, the goal is to identify characters in documents or in handwriting; one generally assumes that characters are fairly easy to find - in most documents that is the case. Nowadays, a Deep CNN is used for recognition, in Section 26.2.1 we introduce how to use a package for Python. In recognition in the wild, one attempts to find text in arbitrary scenes, i.e. outdoor scenes, in which case we now are faced with the challenge to firstly locate the text, i.e. a street sign with elaborate instructions (Section 26.2.2).

The most powerful and fastest text recognizers are perhaps the ones provided by the Big-Three - exploiting their clouds -, for which one needs an internet connection however:

`https://cloud.google.com/vision/docs/ocr`
`https://docs.aws.amazon.com/rekognition/latest/dg/text-detection.html`
`https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/`

26.2.1 Optical Character Recognition

For localizing the letters in documents it is obvious to start with an image analysis first. For example, we threshold the image to obtain candidates for lines of text (Section 14.1); then we proceed to refine the search

for letter centers at which locations one would apply the CNN to perform character identification. In case of scanned documents, it is probably necessary to apply an adaptive threshold, as the brightness might change across the page. With help of morphological operations we can generate letter candidate regions. Identifying handwritten letters is of course much more challenging than identifying printed characters. To compare methods, the following famous dataset set is used in research:

`http://yann.lecun.com/exdb/mnist/`

In Matlab there exists a function script called `ocr` that does character recognition, an example is given in Appendix **P.24.1**.

In Python the most popular package is `tesseract`, which - due to its complexity - offers also various wrappers in Python. Appendix **P.24.1** gives a simple example. Below is listed a wrapper package.

`https://github.com/tesseract-ocr/tesseract/` (the original one)
`https://pypi.org/project/tesserocr/` (pip install tesserocr)

26.2.2 Detection in the Wild

Detecting text in an arbitrary scene requires now a systematic whole image search. Matlab gives an example of how to do that. It is based on using so-called MSER features (maximally stable extrema regions). The actual character identification part uses then its `ocr` function. As one can imagine, an efficient search would be useful to have. Here is a list of systems:

`https://github.com/chongyangtao/Awesome-Scene-Text-Recognition`

The example for OCR given in **P.24.1** - using tesseract - includes detection in the wild, but can perhaps be improved using one of the newer search algorithms.

26.3 Remote Sensing

wiki Remote_sensing

Dav p738, ch23

Remote sensing is the analysis of images taken by satellites or aircrafts (wiki Satellite_imagery, wiki Aerial_photography). It is used for example in meteorology, oceanography, fishing, agriculture, biodiversity conservation, forestry, landscape, geology, cartography, regional planning, education, intelligence and warfare.

The images one obtains are huge and manipulating them therefore takes a lot of time - it is practically impossible to apply sophisticated image-processing methods on large areas; for instance scanning the ocean for floating debris is unfeasible at this point. From a methodological viewpoint, there is no new technique to explain here. We now merely give some background information on this topic, in particular on satellite imagery.

In satellite imagery one can distinguish between four types of resolution: spatial, spectral, temporal and radiometric. The more modern the satellite, the higher are those resolutions in general.

- Geometric Resolution: specifies the optical precision and is typically given as the Ground Sample Distance (GSD). Modern commercial satellites provide a GSD of 31 centimeters for gray-scale images, thus a car would be ca. 5 x 15 pixels large. A coverage of one square kilometer would fit in a 3300 x 3300 pixel image approximately.

- Spectral Resolution: satellite images can be simple RGB photographs, but also a broad range of electromagnetic waves is typically measured. Early satellites recorded so-called multi-spectral images, where at each pixel several bands of the electromagnetic spectrum were recorded, sometimes up to 15 bands wiki Multispectral_image. Table 3 gives an impression of how some of those bands can be exploited. Meanwhile there exist satellites that record several tens or even hundreds of bands, generating so-called hyperimages, which permit detailed selections; the amount of storage required for such images is very large however.

- Temporal Resolution: specifies the time with which a satellite revisits the same location, called revisiting frequency or return period. This is of interest if one intends to track changes over time. The return period can be several days.

Table 3: Example of bands and their use. Given ranges are approximate - exact values depend on satellite.

Band Label	Range (nm)	Comments
Blue	450-520	atmosphere and deep water imaging; depths up to 150 feet (50 m) in clear water.
Green	515-600	vegetation and deep water; up to 90 feet (30 m) in clear water.
Red	600-690	man-made objects, in water up to 30 feet (9 m), soil, and vegetation.
Near-infrared (NIR)	750-900	primarily for imaging vegetation.
Mid-infrared (MIR)	1550-1750	imaging vegetation, soil moisture content, and some forest fires.
Far-infrared (FIR)	2080-2350	imaging soil, moisture, geological features, silicates, clays, and fires.
Thermal infrared	10400-12500	emitted instead of reflected radiation to image geological structures, thermal differences in water currents, and fires, and for night studies.
Radar & related tech		mapping terrain and for detecting various objects.

- Radiometric Resolution: concerns the 'range' of values and starts typically at 8 bits (256 values) and goes up to 16 bits (65535 values).

There exist software tools that preprocess the raw satellite images in order to transform them into a format that is more suitable for object detection and classification, see wiki Remote_sensing_application. The larger the area under investigation, the more time consuming is this transformation.

To compare methods one can participate in the following classification competitions:
https://www.kaggle.com/c/dstl-satellite-imagery-feature-detection/data

26.4 Posture Estimation (Pose Estimation of Humans)

Posture estimation is the recognition of a person's pose, the detailed alignment of the body parts (Fig. 70); the task is also called simply *pose estimation* sometimes. Such tasks used to be solved with feature point detection (Section 12): one would learn the configuration of key-points in a set of training images, and then find and match those points in the testing images. Such systems can be fairly complex. Nowadays, Deep Nets provide an easier way to train such poses (Section 8): those networks are not less complex than traditional approaches, but they are easier to use and achieve better results - sometimes much better results.

Posture estimation systems typically localize keypoints first, such as the limbs or joints or major body parts, from which one then can estimates their precise alignment. There exists a variety of (traditional) approaches that often model explicit structural relations, ranging from few relations expressed deterministically to multiple relations expressed probabilistically. CNNs have taken the latter approach to the extreme and do not formulate any relations: instead those are learned automatically, in particular in the more global (upper) maps of the network hierarchy. The (presently) best performing network is called *OpenPose*, the corresponding article entitled 'Convolutional Pose Machines'.
Figure 70 shows the output of that network. With that network, 15 key-points can be detected; with some network variants even more. With the same network architecture one can also train to estimate the pose of faces (Fig. 71), hands (Fig. 72), and in the near future feet (as the authors claim):
https://github.com/CMU-Perceptual-Computing-Lab/openpose

The code in Appendix **P.21** shows how to feed a single image to the network and how to read out the key-points from the network output. The ordering of key-points can be looked up on:
https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/output.md

Now that we have a list of points, we could try to classify poses. There are several variants that one can think of, which in itself is an iteration of all the methods introduced so far: we can attempt to apply traditional pattern recognition methods (Section 3.3, Appendix H), run a CNN, or perhaps apply some shape recognition method (Section 16).

Figure 70: Posture estimation, here focusing in particular on joints. Estimate generated by OpenPose, a deep CNN that was trained on thousands of poses. See P.21.

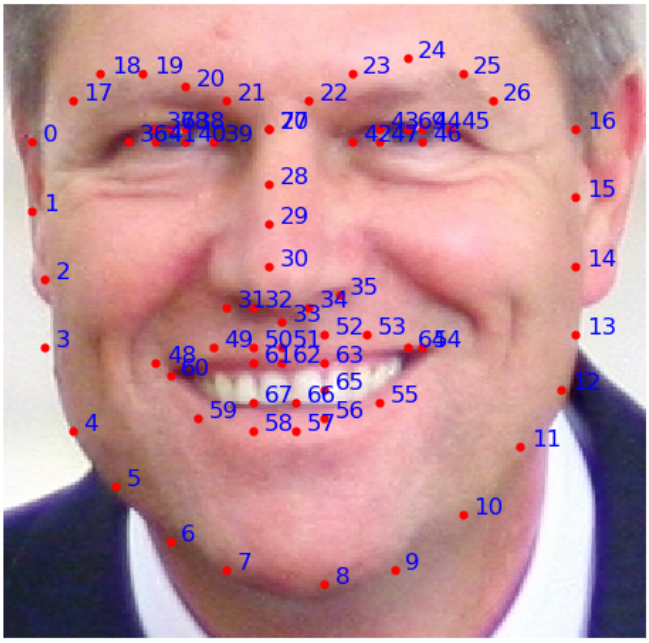

Figure 71: Face pose estimation, also estimated by the OpenPose network, trained on images of faces. See P.21.

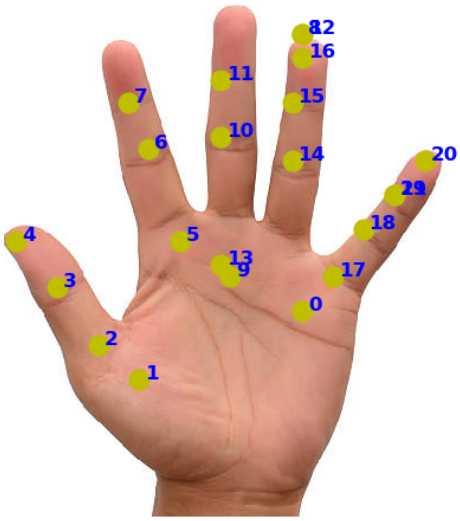

Figure 72: Hand pose estimation. Also estimated by the OpenPose system, trained on images containing hands. See P.21.

27 Humanoid Vision

This is an outlook what needs to be implemented in order to achieve a system operating similarly to the human visual system (HVS). Such systems are of interest where speed and flexibility is required, such as a household robot, an automotive system or an unmanned vehicle, that would operate using mainly RGB images as input. We distinguish here between the representational and the algorithmic level, a distinction loosely inspired by the levels of analysis proposed by David Marr (wiki wiki/David_Marr_(neuroscientist)). The representational level is concerned with how the visual environment is represented and understood (Section 27.1). The algorithmic level addresses how exactly the information is processed (Section 27.2).

27.1 Representational Level

Firstly, one needs to realize that the amount of scene information is nearly endless and that it is impossible to process everything at once in detail (see again the historical note in Feature Engineering of Section 3.4). In order to cope with the vast amount of information, the HVS acquires scene information in two phases (Fig. 73): in a first phase it acquires a coarse understanding of the scene, also called the *gist*. This understanding is achieved in about 150 milliseconds, within a single glance so to speak (Thorpe); the process is also called *gist perception* (Oliva, Torralba). In a second phase, the HVS starts browsing the image for details, which can be easily measured using an eye-tracker, a device that records the eye movements. Figure 74 shows those eye movements when watching a face: for a face, there is a clear tendency toward fixating the eyes, the nose and the mouth.

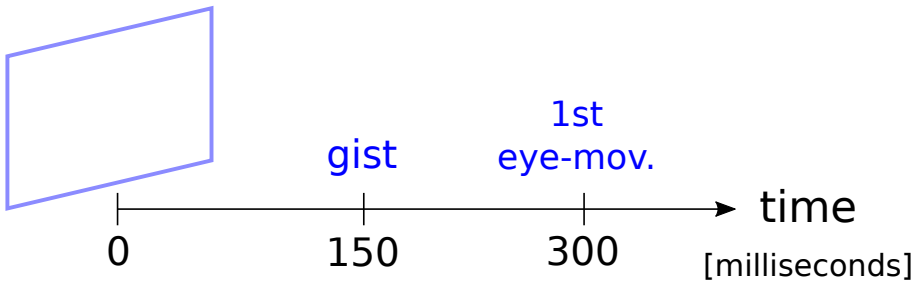

Figure 73: Recognition evolvement when viewing an image: 0 milliseconds represents the onset time of image display. In a first phase the gist of the scene is acquired, also known as *gist perception*; it is completed within ca. 150 milliseconds. Gist perception retrieves a so-called scene skeleton, which contains spatial information that helps performing precise eye movements to scene parts. In a second phase, those eye-movements will analyze and learn details of the scene by shifting from one image location to another.

Gist perception is the phenomenon to classify a scene correctly even without having verified all details. Psychophysics research has shown that the HVS easily understands scenes even when the presented scenes were artificially manipulated to exhibit small visual illusions, such as physical anomalities, e.g. an object not standing on the ground and thus seemingly floating in the air (Biederman). The gist perception process is robust in a way that may suggest insensitivity to peripheral detail, yet it is still extremely reliable in classification.

During the browsing phase, the HVS rests approximately 300-400 milliseconds on an image location, also called a *fixation*. It then moves the eyes in a ballistic manner toward another location; that ballistic movement is also called a *saccade*. For example, when one reads text, the eyes move roughly from word to word, but it will also occasionally skip small words. This example shows too, that the HVS is relatively robust to irregularities, in this case orthographic mistakes - sometimes it does not even note them.

Eye-movement recordings also show that the fixation pattern changes substantially, when the subject was instructed with specific search goals. For example it the subject was asked to judge whether the face shows a smile, then the fixations would primarily land around the mouth ends. Analogously, when a automobile driver takes a curve, the driver will focus on specific points of the curved lane. This shows again that saccades hardly follow an erratic search, but instead know precisely where to land for detailed analysis.

Figure 74: Eye movements and fixations recorded when viewing an image of a face. The human visual system moves the eye every 300 to 400 milliseconds. The actual eye movement is of ballistic nature and occurs within few tens of milliseconds; it is called a saccade. The majority of time is spent for analysing some location in the scene. Image source: Wikipedia

This knowledge where to land is stored in some construct that expresses a relatively well defined skeleton of spatial relations between scene parts or objects. This construct is now called *skelet*. Depending on the goal, it can provide relatively precise coordinates for the next fixation.

In summary, the HVS understands a scene within a glance by a process called gist perception. During that process the scene has been reliably classified, but the scene's details have not been recognized at all, by far not. Gist perception activates a skelet that contains information about the scene details, the spatial relations between scene parts. This enables the HVS to shift its focus relatively precisely to new image locations (using saccades) for the purpose of learning about the scene. In short, the HVS uses a gist-skelet-shift process to orient itself in a scene:

$$\text{Gist} \rightarrow \text{Skelet} \rightarrow \text{Shift} \tag{41}$$

If the HSV is instructed with a search goal, then it will select very specific points of analysis.

Realization How can we apply the principle of a gist-skelet-shift process to build a computer vision system that performs like a HSV? Ideally, one would develop a self-learning system that would acquire all the knowledge on its own, as a human does throughout his/her early life. This may perhaps still be a step too large, but we can learn from focusing on specific scene types, such as the above mentioned scenarios of automotive vision and household robot vision. Even for such narrower scenarios, the challenge remains huge.

A good starting point is a labeled database, such as the City-Scapes set (https://www.cityscapes-dataset.com/), that contains individually annotated objects and scene parts. If we look at the sum of annotations for one category (across images), then we observe - not surprisingly - , that categories tend to occur in certain zones, see Fig. 75. The HVS will automatically learn the locations and spatial extent of those zones. In addition, it will also know relatively precise points of fixation where those categories occur for individual scenes types. For instance, the locations of object candidates are different for a narrow one-way street than for a two-lane street. One would therefore sub-categorize the scenes and analyze the spatial statistics of object locations separately. Those spatial statistics would constitute the skelet. This would allow us to apply object classification algorithms at precise locations and therefore reduce the search time (see again Section 10).

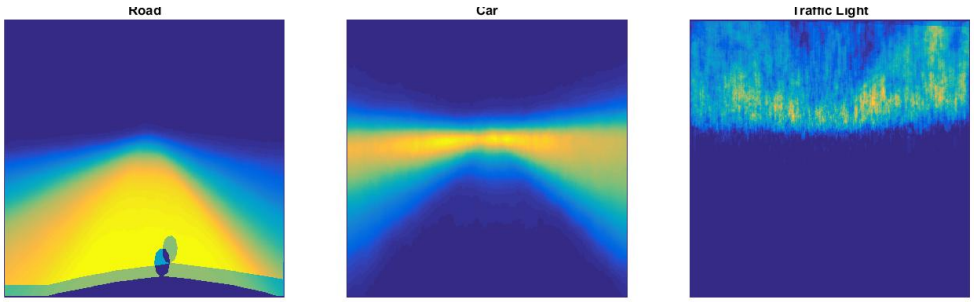

Figure 75: Heat maps for three categories of the CityScape data set, road (left), car (center) and traffic light (right). As expected, objects appear in certain zones. For different scene types, those zones become more specific.

27.2 Algorithmic Level

The algorithmic level now addresses how exactly gist perception is achieved, how the skelet looks like, what features could guide the shifts. Whereas the representational level has drawn a scheme that many scientists can agree on - the gist-skelet-shift process -, for the algorithmic level there exists less agreement. Of the three existent approaches to recognition (Section 3), only Deep Nets and Structural Description are contenders.

The Deep Learning approach certainly bears some resemblance with the HVS in the sense that it gradually integrates information from pixels to some 'holistic' percept, also called a local-to-global recognition evolvement. But that resemblance is as good as the comparison between airplanes and birds: there are certain obvious similarities, but at an algorithmic level, the comparison does not hold anymore. This may also be the case for neural networks; many neuroscientist doubt that artificial neural networks are an actual reflection of the algorithmic level.

The advantage of Deep Nets is their robustness and convenience. They are robust to fragmented input as they gradually integrate their input. They are convenient because there exist software packages that make their use easy, e.g. PyTorch, Tensorflow. Their downside is that they are not explicit in their feature output: to detect a straight line or a region, one has to analyze the entire network architecture; their interactiability is limited.

The Structural Description approach operates with contours and regions directly and many of those outline the objects or scene parts themselves. The approach has however not produced segmentation results as Deep Neural Networks have (Section 9). With recent developments however, namely our prototype, there are signs that a more capable system is possible.

A Image Acquisition

Digital image acquisition is the conversion of the 'outer', analog signal to a digital representation, a number. This analog-to-digital conversion can be carried out by different types of image sensors (cameras). The conversion can be very complex and often involves the generation of an output that is 'visible' to the human eye.

One principal distinction between acquisition methods is passive versus active. In passive methods, the scene is observed based on what it offers: the simplest case is the regular light-sensitive camera that measures the environment's luminance - the reflection of the illuminating sun. In active methods, a signal is sent out to probe the environment, analogous to a radar. Some sensors combine both methods.

The following are the principal sensors used for acquiring images.

Light-Sensitive Camera: measures from the visible part of the electromagnetic spectrum, typically red, green and blue dominance; the RGB camera is an example. Light is the preferred energy source for most imaging tasks because it is safe, cheap, easy to control and process with optical hardware, easy to detect using relatively inexpensive sensors, and readily processed by signal processing hardware.

Multi/Hyper-Spectral Sensors: measure from a broader part of the electromagnetic spectrum (than the light-sensitive cameras) with individual sensors tuned to specific bands. Originally it was developed for remote sensing (satellite imagery, Section 26.3), but is now also employed in document and painting analysis.

Range Sensor (Rangefinder): is a device that measures the distance from the observer to a target, in a process called ranging or rangefinding. Methods include laser, radar, sonar, lidar and ultrasonic rangefinding. Applications are for example surveying, navigation, more specifically for example ballistics, virtual reality (to detect operator movements and locate objects) and forestry. Data for lidar can be found here: http://www.lidarbasemaps.org/.

Tomography Device: generates an image of a body by sections or sectioning, through the use of any kind of penetrating wave. The method is used in radiology, archeology, biology, atmospheric science, geophysics, oceanography, plasma physics, materials science, astrophysics, quantum information, and other areas of science.

The obtained 'raw' image may require some manipulation such as re-sampling in order to assure that the image coordinate system is correct; or noise reduction in order to assure that sensor noise does not introduce false information.

Summarizing, the output of sensors is an array of pixels, whose values typically correspond to light intensity in one or several spectral bands (gray-scale, colour, hyper-spectral, etc.), but can also be related to various physical measures, such as depth, absorption or reflectance of sonic or electromagnetic waves, or nuclear magnetic resonance.

B Convolution [Signal Processing]

Expressed in the terminology of applied mathematics, convolution is the repeated multiplication of one function on the domain of another function producing a third function; it is considered an 'operation' and is similar to cross-correlation. In signal processing terms, the first function is the signal - in our case often an image -, and the second function is a so-called *kernel* and manipulates a local neighborhood of that image at each location. This is easier to understand in one dimension first (Section B.1), then we introduce this for two dimensions (Section B.2).

B.1 In One Dimension

Our signal is for instance the face profile as in Figure 14. We wish to determine the extrema of that signal but that is easier if we smoothen the signal first with a low-pass filter. In plain words, we average the signal over a small neighborhood at each pixel. Let us clarify the convolution operation on a very simple signal `S`: our signal is zero everywhere except for one value in center which holds the value 2. We convolve it with three different kernels, `Ka`, `Kb` and `Kg` using the function `conv` (Python: `scipy.signal.convolve`). Then we plot the three new functions.

```
clear;
S   = [0 0 0 0 2 0 0 0 0];      % the signal
Ka  = [1 1 1]/3;                % averaging kernel
Kb  = [.25 .5 .25];             % triangle function
Kg  = pdf('norm',-2:2,0,1);     % 5-pixel Gaussian

Sa  = conv(S, Ka, 'same');
Sb  = conv(S, Kb, 'same');
Sg  = conv(S, Kg, 'same');

%% ------   Plotting
figure(1);clf;
Xax     = 1:nPix;
plot(Xax, S, 'k');   hold on;
plot(Xax, Sa, 'r*');
plot(Xax, Sb, 'bt');
plot(Xax, Sg, 'g.');
```

So far, not much has happened. The new function looks like its kernel, but scaled in amplitude. It becomes more interesting if we make the signal more complicated: turn on another pixel in signal `S` and observe. To ensure that you understand the detailed convolution process, look at the following explicit example:

```
Sa2     = zeros(1,nPix);
for i = 2:nPix-1
    Nb      = S(i-1:i+1);       % the neiborhood
    Sa2(i)  = sum(Nb .* Ka);    % multiplication with kernel and summation
end
assert(nnz(Sa2-Sa)==0);         % verify that it is same as 'conv'
```

The example is a simplified implementation of the convolution process, namely we do not observe the boundary values and it works only for kernel of length equal three pixels, with `i` running from 2 to `nPix-1`. But it contains the gist of the convolution operation.

When applying a convolution function you need to pay attention to what type of treatment you prefer for the boundary values. Matlab offers three options: full, valid and same; they return outputs of different sizes. We refer to the documentation for details. They do matter, so when you apply a convolution you need to think about the boundary values. If you prefer to set your own boundary values, then you compute only the central part of the convolution - in Matlab with option `valid` - and then use `padarray` to set the boundary values. We did that in the example of the face profiles, see Appendix **P.2.4**.

Mathematical Formulations In engineering equations the convolution can be written as:

$$(S * K)[k] = \sum_{i}^{n} S[i]K[k-i] \tag{42}$$

where `*` is the convolution symbol, `i` is the signal's variable, `n` the number of pixels of the signal and `k` is the Kernel's variable. That was the formulation for the discrete convolution. The continuous convolution is written as:

$$(S * K)(k) = \int_{-\infty}^{\infty} S(i)K(k-i)\delta i \tag{43}$$

B.2 In Two Dimensions [Image Processing]

In two dimensions the convolution process integrates over a local two-dimensional neighborhood. In case of an image the neighborhood can be a 3x3 pixel neighborhood, or a 5x5 pixel neighborhood, etc. The mathematical formulation remains essentially the same as above; one can understand the kernel as of any dimension, but the signal and kernel need to be defined clearly from the beginning.

In Matlab we now use the function `conv2`, in Python we use `scipy.signal.convolve2d`:

```
clear;
% --- the image
I       = zeros(10,10);
I(5,5)  = 1;
% --- kernels
Ka      = ones(3,3)/9;                  % averaging kernel
Kg      = fspecial('gaussian',5,1);     % 5x5 Gaussian kernel
% --- convolutions
Ia      = conv2(I, Ka);
Ig      = conv2(I, Kg, 'same');

%% ------   Plotting
figure(1);clf;[nr nc] = deal(2,2);
subplot(nr,nc,1); imagesc(I,[0 1]);
subplot(nr,nc,2); imagesc(Ia);  colorbar;
subplot(nr,nc,3); imagesc(Ig);  colorbar;
```

Speeding Up Because image convolution is a relatively time-consuming operation due to the repeated multiplication of two matrices, there exist methods to speed up image convolution. Those speed-ups work only if the kernel shows specific characteristics, in particular it needs to be symmetric. The Gaussian function for instance is suitable for speed up. In that case, an image convolution with a 2D Gaussian function can be separated into convolving the image twice with the 1D Gaussian function in two different orientations:

```
Kg1     = normpdf(-2:2);                          % a one-dim Gaussian
Iblr1   = conv2(I,Kg1'*Kg1,'same');               % with 2D function
Iblr2   = conv2(conv2(I,Kg1,'same'),Kg1','same'); % with 1D functions
Iblr3   = imgaussfilt(I,1);                       % special function
```

The product `Kg1'*Kg1` creates a 2D Gaussian - we could have also generated it using `fspecial('gaussian',[5 5],1)` for instance.

For small images, the durations for each of those three different versions are not much different - the duration differences become evident for larger images. Use `tic` and `toc` to measure time; or use `clock`.

C Filtering [Signal Processing]

An image can be filtered in different ways depending on the objective: one can measure emphasize or even search certain image characteristics. The term filter is defined differently depending on the specific topic (signal processing, computing, etc.). Here the term is understood as a function, which takes a small neighborhood and computes with its pixel values a certain value; that computation is done for each neighborhood of the entire image. That local function is sometimes called kernel.

For educational purposes, we differentiate here between five types of filtering. In the first one, the kernel calculates simple statistics with the neighborhood's pixel values (Section C.1). Then there are three basic techniques to emphasize a certain 'range' (band) of signal values: low-pass, band-bass and high-pass filtering (Sections C.2, C.3 and C.4, respectively). Finally, there exist even more complex filter kernels.

C.1 Measuring Statistics

Sometimes we intend to measure very simple statistics. Those in turn can be used for a variety of tasks, in particular texture description. One can determine the range of values within the neighborhood, their mean, their standard deviation etc. In Matlab those statistical measurements are provided with `rangefilt`, `stdfilt`, `medfilt2`, `ordfilt2`, etc.

There are various ways to apply your own specialized filters, for which it is easiest to arrange the image first into a matrix in which your neighborhoods are aligned column-wise using Matlab's `im2col/col2im` functions. That allows you to conveniently process the neighborhoods. Here is an example taking the average value for a neighborhood:

```
s   = 5;                                % side length
I   = reshape(linspace(0,1,s*s),[s s])'; % [s s] stimulus
C   = im2col(I,[3 3]);                  % extract [3 3] patches
Mn  = mean(C,1);                        % take mean
Ims = col2im(Mn,[1 1],[s s]-2);         % rearrange to matrix
Im  = padarray(Ims,[1 1]);              % pad to make it same size as I
```

C.2 Low-Pass Filtering

Because signals are often noisy, applying a low-pass filter to the image is often one of the first steps and that serves to squash the 'erratic' values. That is why images are typically filtered with at least a 3x3 Gaussian filter initially, no matter the exact task. The Gaussian filter is perhaps the most frequently used low-pass filter and it is worth looking at its one-dimensional shape, see Figure 76.

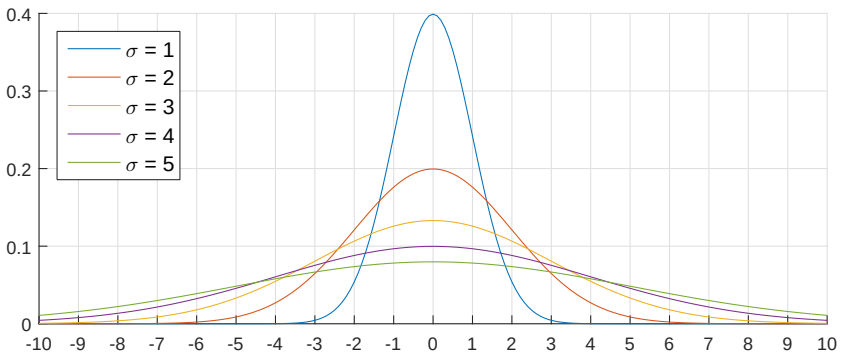

Figure 76: Gaussian function for five different values of sigma (1,..,5) placed at x=0. The parameter sigma determines the width of the bell-shaped curve.

In two dimensions, the Gaussian looks as depicted in Figure 20, namely in the first four patches (filter no. 1-4).

One can take also other functions for low-pass filtering, such as a simple average, see code block above or code in the Appendix on Convolution above, where `Ka = ones(3,3)/9` was used. The reason why the Gaussian function is so popular, is that it has certain mathematical advantages.

C.3 Band-Pass Filtering

One example of band-pass filtering is region detection with the Difference-of-Gaussian (DOG) as we introduced in Section (7.1). Another example are the Laplacian-of-Gaussian (LoG) filters used for texture detection (Figure 20).

C.4 High-Pass Filtering

Edge detection is an example for high-pass filtering. In some algorithms the first derivative of the Gaussian is used as high-pass filter, see also Figure 77.

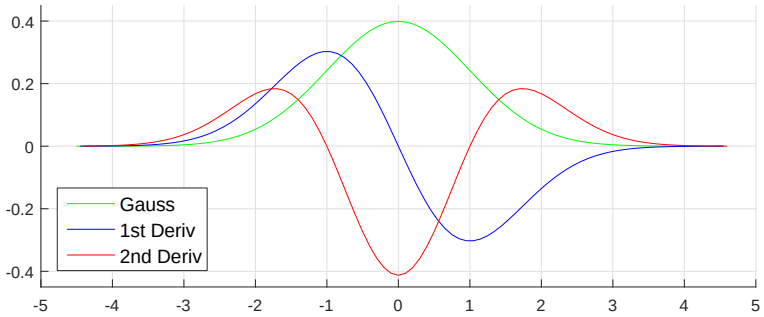

Figure 77: The Gaussian function (green) and its first (blue) and second (red) derivative. The Gaussian function itself is often used as low-pass filter. The first derivative of the Gaussian is often used as high-pass filter, for example in edge detection. The second derivative is occasionally used as band-pass filter, for example as region detection.

C.5 Function Overview

The variety of filtering commands can be confusing sometimes, that is why we provide here a short summary of frequently used filtering operations:

Table 4: Overview of commonly used image filtering methods. `I` = image.

Command	Comments
`rangefilt(I,true(rad*2+1))`	local range. Neighborhood is logical matrix.
`stdfilt(I,true(rad*2+1))`	local std dev. Neighborhood is logical matrix.
`Flt = fspecial('gaussian',[3 3],0.5)`	creates a filter. To be used with `conv2`, `imfilter` ...
`conv2(I,Flt,'same')` `convolve2d(I,Flt,mode='same')`	convolution in `scipy.signal`
`im2col(I,[3 3])`	extracts blocks from image, `[nPix 9]`.
`col2im(B,[3 3])`	inverse of `im2col`.
`colfilt`	column-wise filtering, uses `im2col` and `col2img`.
`imfilter`	filter multi-dimensional images.
`nlfilter`	general non-linear sliding filter.
`filter2(Flt, I, 'same')`	filtering with a FIR filter.

Table 5: Filtering of one-dimensional signals. `S` = one-dimensional signal.

Command	Comments

`F1d = normpdf(-2:2)`
`= pdf('norm',-2:2,0,1)`
`F1d = get_window(('gaussian',1),5)`
	generates a Gaussian (normal) filter for five pixels with μ=0 (mean) and σ=1 (standard deviation)
	in `scipy.signal`

`conv(S,F1d,'same')` `convolve(S,F1d,'same')`	1D-convolution. Watch the orientation of vectors! in `scipy.signal`

D Filtering Compact

A quick reference for filtering in Matlab, Python and OpenCV (accessed through Python), as introduced in Section 6 mostly.

```
Isum    = conv2(I, ones(3,3), 'same');  % summation
Igss    = imgaussfilt(I, 2);            % Gaussian
Imed    = medfilt2(I, [3 3]);           % median
Irng    = rangefilt(I, ones(3,3));      % range
Istd    = stdfilt(I, ones(3,3));        % standard deviation
Ired    = impyramid(I, 'reduce');       % downsampling by 2
Iexp    = impyramid(I, 'expand');       % upsampling by 2
```

```
from numpy import ones, gradient
from scipy.signal  import convolve2d
from scipy.ndimage import gaussian_filter, median_filter
from skimage.transform import pyramid_reduce, pyramid_expand

Isum    = convolve2d(I, ones((3,3)), 'same')
Igss    = gaussian_filter(I, sigma=2.0)
Imed    = median_filter(I, (3,3))
Ired    = pyramid_reduce(I)
Iexp    = pyramid_expand(I)
Dx, Dy  = gradient(I)
```

For OpenCV, there exists a nice overview:

https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html

Input and output depths at one glance; as well as the border extrapolation types:

```
CV_8U           -1/CV_16S/CV_32F/CV_64F
CV_16U/CV_16S   -1/CV_32F/CV_64F
CV_32F          -1/CV_32F/CV_64F
CV_64F          -1/CV_64F

BORDER_REPLICATE     aaaaaa|abcdefgh|hhhhhhh
BORDER_REFLECT       fedcba|abcdefgh|hgfedcb
BORDER_REFLECT_101   gfedcb|abcdefgh|gfedcba
BORDER_WRAP          cdefgh|abcdefgh|abcdefg
BORDER_CONSTANT      iiiiii|abcdefgh|iiiiiii  with some specified 'i'
```

```
Isum    = cv2.boxFilter(I, -1, (3,3), normalize=False)
Isum2   = cv2.filter2D(I, -1, ones((3,3)))
Igss    = cv2.GaussianBlur(I, (3,3), 1)
Imed    = cv2.medianBlur(I.astype(uint8), 3)
Ired    = cv2.pyrDown(I)
Iexp    = cv2.pyrUp(I)
Dx, Dy  = cv2.spatialGradient(I.astype(uint8))
```

E Correspondence

Correspondence is the task of determining which parts of two similar structures correspond to each other. The structures can be shapes, objects or scenes. This correspondence is necessary in particular in motion estimation methods, such as optic flow, stereo vision, 3D pose estimation, structure-from motion, etc.; as well as in shape matching. The method for finding correspondence depends on how different the two images (shape) are and on the time constraints of the task. Correspondence can be established using individual pixel values or extracted features (corners, contours, etc.), or a combination of both.

Pixel-Based: this is also known as *direct* or *dense* approach as all or most pixels are involved; occasionally also the term *image matching* is used. This method is useful when images have only a slight offset, a few pixels, and if appearance changes are not too different, as it is often the case in optic flow or in stereo matching. The method compares the two images patch-wise using image arithmetics as introduced in Section 4.3. The choice of patch size is often subject to the speed-accuracy trade-off: the larger the patch, the more accurate the correspondence but the slower it is achieved; the smaller the patch, the inverse holds. In short baseline stereo correspondence that size is typically around 9x9 or 11x11 pixels.

As this patch-wise matching is computationally intensive, a short-cut is to firstly identify conspicuous points in both images using a feature detector method (Section 12) and then to apply patch matching only at those points. This short-cut is already a step toward feature-based correspondence.

Feature-Based: the further apart the two structures are in their respective images, the more appropriate is it to search correspondence with features, such as corners, contours, regions, etc. This is also called *sparse* correspondence and is faster than dense matching; it can also be more accurate depending on the task. A well established and efficient method is to use the local features as introduced in Section 12, see Section 12.4 in particular.

Shape-Matching When matching shapes, expressed as a list of contour pixels (Section 16.3), an analogous speed-accuracy trade-off is faced. Pixel-wise matching can be very accurate but is computationally costly. To speed up the matching, one seeks firstly features on a shape, such as corners or straight line segments, also known as landmarks in this context. Then one aligns the two shapes based on those detected features, which is essentially the search for a coarse correspondence. Finally, one fine-tunes the matching by returning to pixel-wise matching along the found coarse alignment.

F Neural Networks

The element of a neural network (NN) is a so-called *Perceptron* wiki Perceptron. A perceptron essentially corresponds to a model of a binary, linear classifier as in traditional Machine Learning (left in Fig. 78); a multi-class perceptron discriminates between multiple classes like any multi-class linear classifier (right in Fig. 78). Formulated in the terminology of neural networks, a (multi-class) Perceptron consist of a (neural) *unit* that receives input from other units. The input is weighted and then summed, in mathematical terms that is the dot product. That sum c is then passed to a so-called *activation* function that thresholds its input given by parameter θ:

$$a = f(c, \theta) \tag{44}$$

The output a of the activation function is then passed to the next layer. There are many types of activation functions (wiki Activation_function). A popular one for hidden units is the rectified linear unit (relu) which sets values below θ to zero, and leaves values above as they are. The sigmoid function is often used in the last layer.

For a two-class problem - a binary decision -, there is one integrating unit; for a multi-class problem there are several integrating units, whose count corresponds to the number of classes to be discriminated.

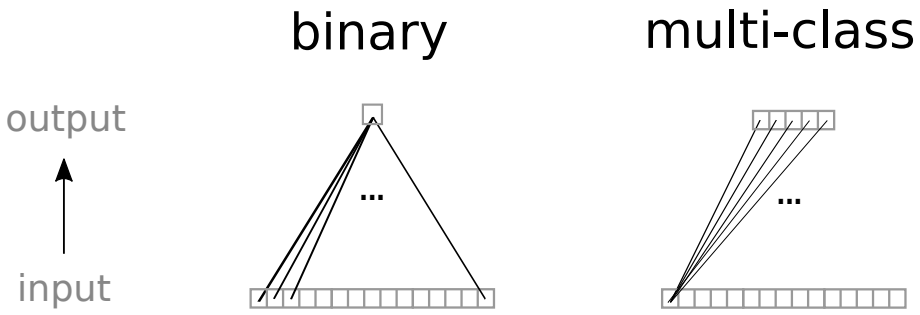

Figure 78: Perceptron - the element of a neural network. **Left:** the binary Perceptron corresponds to a binary, linear classifier; the input units are at the bottom; the integrating, output unit is at the top. **Right:** multi-class Perceptron, equivalent of a multi-class linear classifier. Such elementary networks are stacked to form larger networks.

If such a Perceptron is stacked, then we talk of a *Multi-Layer* Perceptron (MLP), which is introduced next (Section F.1). There are also 'richer' ways to wire neural units and we introduce one such variant in Section F.2. In Section 8.3 we survey the available, pretrained networks that are used for transfer learning and image classification. In Section F.3 we introduce a network that performs a certain type of clustering.

F.1 A Multi-Layer Perceptron (MLP)

wiki Multilayer_perceptron

An MLP is a neural network consisting of three or more layers (Figure 79): an input layer, receiving the image in our case; one or more hidden layers that combine information; and an output layer, that holds the score (confidence) for each trained class. The unit count of the input layer typically corresponds to the number of pixels of the input image. The unit count of the hidden layers varies and depends on the task. The unit count of the output layer corresponds to the number of classes to be distinguished.

The units between two layers are typically all-to-all connected that is the connections build a *complete bipartite graph* (wiki Bipartite_graph). Each connection holds a weight value that will be learned during the training process.

An Example We demonstrate how such a network can be programmed in PyTorch, https://pytorch.org/. We use a database of handwritten digits (0-9), the so-called MNIST database, containing 60000 training images and 10000 testing images. Note that the function loads the images as two dimensions (28 x 28). The training images are therefore stored as a three-dimensional array 60000 x 28 x 28. A single digit image can be viewed by `DATtren.train_data[0,:,:].numpy()`.

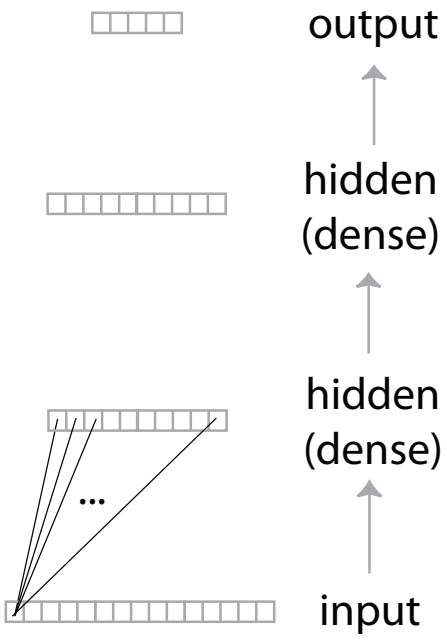

Figure 79: A Multi-Layer Perceptron (a general classifier in fact). This architecture has four layers: input layer, two hidden (dense) layers, and an output; classification flow occurs from bottom to top.
The **input layer** is linear and corresponds to the number of image pixels.
The **(first) hidden layer** receives the values from the input layer with all-to-all connectivity (only the connections from one pixel to all hidden units are indicated). Each connection has a weight whose exact value will be learned during the learning process. The corresponding weight matrix has the size of number of input pixels times number of hidden units.
The **(second) hidden layer** receives the output of the first hidden layer. The corresponding weight matrix has dimensions corresponding to the sizes of the two hidden layers.
The **output layer** has the same number of units as the number of classes to be discriminated. For instance if we intend to discriminate the digits 0 to 9, then there would be 10 output units. The output layer receives the values of the previous hidden layer.

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

pthDatBase  = 'c:/IMGdown'      # path of the MNIST database on your PC

# =====     Topology Parameters    =====
szInp       = 784       # input size (number of input units)
szHid       = 500       # number of hidden units
nClass      = 10        # number of classes
# =====     Learning Parameters     =======
nEpoch      = 5         # number of epochs
szBtch      = 100       # batch size (number of training samples per epoch)
lrnRate     = 0.001     # learning rate

#%% %%%%%%%%%%%%%      PREPARE DATA     %%%%%%%%%%%%%%%%%%%
TrfImg  = transforms.ToTensor()     # uint8 [0,255] > torch float [0,1]

DATtren = MNIST(pthDatBase, train=True,  transform=TrfImg, download=False)
DATtest = MNIST(pthDatBase, train=False, transform=TrfImg)
# DATtren.data[0,:,:].numpy()       # first digit of database as numpy
# DATtren.targets[0].item()         # corresponding label

LOADtren = DataLoader(DATtren, batch_size=szBtch, shuffle=True)
LOADtest = DataLoader(DATtest, batch_size=szBtch, shuffle=False)

#%% %%%%%%%%%%%%%      NETWORK    %%%%%%%%%%%%%%%%%%%
class NN_MLP(nn.Module):
    def __init__(self, szInp, szHid, nClass):
        super(NN_MLP, self).__init__()
        self.fc1    = nn.Linear(szInp, szHid)
        self.relu   = nn.ReLU()
        self.fc2    = nn.Linear(szHid, nClass)

    def forward(self, In):
        Out = self.fc1(In)
        Out = self.relu(Out)
        Out = self.fc2(Out)
```

```
        return Out
NET         = NN_MLP(szInp, szHid, nClass)

#%% %%%%%%%%%%%%%      OPTIMIZATION    %%%%%%%%%%%%%%%%%%
criterion   = nn.CrossEntropyLoss()
optimizer   = torch.optim.Adam(NET.parameters(), lr=lrnRate)

#%% %%%%%%%%%%%%%      LEARNING     %%%%%%%%%%%%%%%%%%
nImgTren    = DATtren.data.shape[0]
NET.train()                                 # ensure training mode
# =====     LOOP EPOCHS    ======
for e in range(nEpoch):
    # =====     LOOP BATCHES    ======
    lossInt=0
    for i, (IMG, Lbl) in enumerate(LOADtren):
        IMG     = IMG.view(-1, szInp)       # batch [szBtch 784]
        # ====  forward  ======
        optimizer.zero_grad()               # zero the gradient buffer
        Out     = NET(IMG)                  # forward
        loss    = criterion(Out, Lbl)       # error calculation
        # ====  backward  ======
        loss.backward()                     # calculate gradients
        optimizer.step()                    # adjust weights
        lossInt += loss.data.item()         # accumulate loss value

    print ('epoch %d, loss: %.4f' % (e+1, lossInt/nImgTren))

#%% %%%%%%%%%%%%%      EVALUATION BATCH-WISE   %%%%%%%%%%%%%%%%%%
NET.eval()                                  # switch to testing mode
cHit=0; cImg=0
for IMG, Lbl in LOADtest:
    IMG         = IMG.view(-1, szInp)
    Out         = NET(IMG)
    _, Pred     = torch.max(Out.data, 1)        # max across class scores
    cImg        += Lbl.size(0)
    cHit        += (Pred.cpu()==Lbl).sum()      # number of hits

assert cImg==LOADtest.dataset.data.shape[0], 'not same count' # verify
print('accuracy: %1.2f' % (100 * cHit.numpy() / cImg)) # should be ca. 97.77

#%% %%%%%%%%%%%%%      EVALUATION IMAGE-WISE   %%%%%%%%%%%%%%%%%%
nImgTest    = DATtest.data.shape[0]
cHit2=0;
for i in range(0,nImgTest):
    Img         = DATtest.data[i,:,:].float()   # from uint8 to float
    lb          = DATtest.targets[i].item()
    out         = NET(Img.view(-1, szInp))
    _, pred     = torch.max(out.data, 1)        # max across class scores
    cHit2       += pred.item() == lb            # hit

print('accuracy: %1.2f' % (100 * cHit2/ nImgTest)) # should be ca. 97.77
```

The MLP network ignores the fact that the image has two dimensions and 'linearizes' the image by aligning all pixels in a single column: the input to the network is therefore a vector of length 784. We now discuss the four sections 'Prepare Data', 'Build Network', 'Learning' and 'Evaluation'.

Prepare Data The function `TrfImg` is used to instruct that images will be converted from uint8 (or whatever type the image was saved in) to tensors when loaded. The function `MNIST` loads the images and is specific for that database. The function `DataLoader` prepares the indices for learning: one specifies a batch size, the size of a cluster of images used for a learning step, similar to batch-based processing in the Kmeans algorithm; `shuffle` concerns the permutation during learning: it is important for learning, but irrelevant for testing.

Build Network The network is configured with two functions, `__init__` and `forward`. In the first we specify the layers of the architecture and the precise functions they use, in the second we specify the exact flow. A hidden (dense) weight layer is specified by `linear`, which uses the above mentioned all-to-all connections. To such weight layers are used, one with 784-to-500 connections, and another one with 500-to-10 connections. The function `relu` specifies the activation function (eq. 44), here the rectified linear 'unit'.

We initialize the network by instantiating it, called `NET` here, and also specify weather we have CUDA cores available or not. We further initialize the network by defining a loss function, a function that specifies what exactly is considered an error when learning samples, e.g. `nn.CrossEntropyLoss`, which is called `criterion` here. And we need to specify an optimization function that adjusts the weights to be learned, called `optimizer`.

Learning Learning occurs in epochs and is performed strictly on the training set `LOADtren` only. For each epoch a batch (subset) of images is employed, whose size is set with parameter `szBtc`. There exists a rough optimum: too few or too many images per batch results in less efficient learning. The parameter `nEpoch` corresponds to the learning duration essentially. As we ignore the spatial relations of the digit image, we need to reshape manually the images to a linear vector using the function `view`. For each image, one calculates the forward flow and afterwards the errors, followed by adjusting the weights.

Evaluation Evaluation takes place with the testing set `LOADtest`. We only apply the forward operation and obtain the class scores in `Out.data`. One then typically takes the maximum score as the corresponding predicted class label.

The NNs as introduced so far, are considered meanwhile traditional Neural Networks or *shallow* NNs, as they use few layers as opposed to the many layers used by a Deep Net.

F.2 Deep Belief Network (DBN)

wiki Deep_belief_network

A Deep Belief Network (DBN) is another type of Deep Neural Network. It is employed more rarely than the CNN, because its learning duration is slow in comparison to a CNN, but has the advantage that its overall architecture is simpler: often, with two layers you can achieve almost the same performance as with a CNN. Such a DBN has essentially a similar architecture to the Multi-Layer Perceptron, but the DBN contains also connections within the same layer. That particular characteristic makes it extremely powerful, but the downside is that learning is terribly slow.

A Belief Network is a network that operates with so-called *conditional dependencies*. A conditional dependence expresses the relation of variables more explicitly than just by combining them with a weighted sum - as is done in most other classifiers. However determining the full set of parameters for such a network is exceptionally difficult. Deep Belief Networks (DBNs) are specialized in approximating such networks of conditional dependencies in an efficient manner, that is at least partially and in reasonable time. Popular implementations of such DBNs consist of layers of so-called *Restricted Boltzmann Machines* (RBMs).

Architecture The principal architecture of a RBM is similar to the dense layer as used in an MLP or CNN. An RBM however contains an additional set of *bias* weights. Those additional weights make learning more difficult but also more capable - they help solving those conditional dependencies. The typical learning rule for a RBM is the so-called *contrast-divergence* algorithm.

The choice of an appropriate topology for the entire DBN is relatively simple. With two hidden layers made of RBMs, one can obtain already fantastic results. A third layer rarely helps in improving classification accuracy.

Learning Learning in a DBN occurs in two phases. In a first phase, the RBM layers are trained individually one at a time in an unsupervised manner (using the contrast-divergence algorithm): the RBMs perform quasi a clustering process. Then, in the second phase, the entire network is fine-tuned with a so-called back-propagation algorithm. As with CNNs, a Deep Belief Network takes much time to train. The downside of a DBN is, that there exists no method (yet) of speeding up the learning process, as is the case for CNNs.

F.3 Autoencoder

An autoencoder is a type of network that learns a (compressed) code of its own input without any external help (hints or labels), or speaking in Machine Learning terminology, in an unsupervised manner. Such an autoencoder can be used for various tasks and depending on the task, the code has a different meaning. In computer vision, autoencoders are used in particular for segmentation (Section 9).

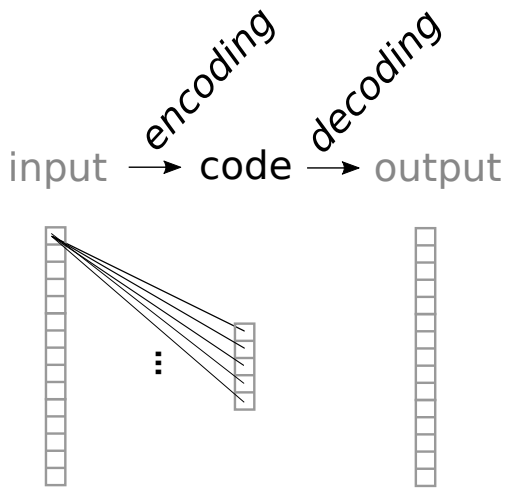

Figure 80: Autoencoder - simplest version. The architecture is often depicted with its flow running from left to right (as opposed to the typical hierarchical layout of classification networks). Three layers: input layer, code layer and output layer. The projections from input to the code layer represent the *encoding* phase: here they are equivalent to a multi-class perceptron (see Fig. 78). The projections from the code to the output layer represent the *decoding* phase and are the reverse of the encoding phase. The input layer and output layer have the exact same dimensionality.

The architecture of an autoencoder consists of two phases: an encoding phase and a decoding phase (Fig. 80). The encoding phase integrates its input to a small layer that represents the code and that is the central (middle) layer of the entire architecture. The encoding phase works essentially like any neural network performing classification as we had treated so far, be that a Perceptron or MLP or DeepNet - depending on the autoencoder's task, size and format. The decoding phase is the inverse of the encoding process and 'unfolds' the code toward the output layer with the exact same connections as for the encoding phase. The output layer has exactly the same size as the input layer.

F.3.1 Example on MNIST

To get acquainted with the operation of an autoencoder we look at an example using the MNIST dataset. It does not involve any convolutions and transforms its input to a single vector (28*28), very much as in Fig. 80. The vector image is then encoded to just three units in three steps: $128 \rightarrow 64 \rightarrow 12 \rightarrow 3$. Then, it is decoded back to its original size. Explained visually, one would take the autoencoder of Fig. 80 and add two more layers on each side.

The example shows only training - no testing on novel images is carried out. The way we utilize the functions is analogous to the classification example given above.

```
# https://github.com/L1aoXingyu/pytorch-beginner/blob/master/08-AutoEncoder/simple_autoencoder.py
import os
import torch
import numpy as np
from torch import nn
from torch.autograd import Variable
from torch.utils.data import DataLoader
```

```
from torchvision import transforms
from torchvision.datasets import MNIST
from torchvision.utils import save_image

if not os.path.exists('./Log'):  os.mkdir('./Log')
import matplotlib.pyplot as plt; plt.show(block=False)

# =====     Learning Parameters     =======
nEpo        = 100
szBtch      = 64
learnRate   = 1e-3

# ======    Select Device (GPU | CPU)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

#%% %%%%%%%%%%%%%      PREPARE DATA     %%%%%%%%%%%%%%%%%%%
TrfImg = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])

DATA = MNIST('C:/IMGdown', transform=TrfImg, download=False)
LOAD = DataLoader(DATA, batch_size=szBtch, shuffle=True)

def f_Batch2Idisp(I): # from image vector to 2D array (per image, not as subplot)
    I = 0.5 * (I + 1)
    I = I.clamp(0, 1)       # trim to range [0,1]
    I = I.view(I.size(0), 1, 28, 28)
    return I

#%% %%%%%%%%%%%%%      NETWORK    %%%%%%%%%%%%%%%%%%%
class NN_AutoEnc(nn.Module):
    def __init__(self):
        super(NN_AutoEnc, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28*28, 128),  nn.ReLU(True),
            nn.Linear(128, 64),     nn.ReLU(True),
            nn.Linear( 64, 12),     nn.ReLU(True),
            nn.Linear( 12,  3))
        self.decoder = nn.Sequential(
            nn.Linear(  3,  12),    nn.ReLU(True),
            nn.Linear( 12,  64),    nn.ReLU(True),
            nn.Linear( 64, 128),    nn.ReLU(True),
            nn.Linear(128, 28*28),  nn.Tanh())

    def forward(self, I):
        I = self.encoder(I)
        I = self.decoder(I)
        return I

NET         = NN_AutoEnc().to(device)

#%% %%%%%%%%%%%%%      OPTIMIZATION    %%%%%%%%%%%%%%%%%%%
criterion   = nn.MSELoss()
optimizer   = torch.optim.Adam(NET.parameters(),
                               lr           = learnRate,
                               weight_decay = 1e-5)

#%% %%%%%%%%%%%%%      LEARNING     %%%%%%%%%%%%%%%%%%%
for e in range(nEpo):
    for Dat in LOAD:
        IN, _   = Dat             # batch of images [szBtc 1 28 28], (labels)
        IN      = IN.view(IN.size(0), -1)   # reshaped to [szBtc 28*28]
        IN      = Variable(IN).to(device) # .cuda()
        # ==========    forward      ===============
        OUT     = NET(IN)               # IN and OUT are [szBtc 28*28]
        loss    = criterion(OUT, IN)    # scalar
```

```
        # ==========    backward     ===============
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # ---- display -----
    print('epoch [{}/{}], loss:{:.4f}' .format(e+1, nEpo, loss.data.item()))
    Iin     = f_Batch2Idisp(IN.cpu().data)     # input digits
    Iout    = f_Batch2Idisp(OUT.cpu().data)    # output digits
    # save to file every 10 epochs
    if e % 10 == 0:
        Isv     = torch.cat((Iin,Iout),dim=0)
        save_image(Isv, './Log/Epo_{}.png'.format(e))
    # for immediate display
    Dg1     = np.array(Iin[0][0]);
    Dg2     = np.array(Iout[0][0]);
    Idisp   = np.vstack((Dg1,Dg2))
    plt.figure(1);
    plt.imshow(Idisp); plt.pause(.1)
```

G Saliency (Attention)

Saliency is the process that determines conspicuous points (or regions) in a scene and then ranks them according to some measure of importance, the degree of saliency. The points are then processed sequentially selected according to decreasing saliency. The idea is to pinpoint to scene details in a prioritized way without processing the entire image in full detail, similar to spatial search of objects (Section 10.5). Saliency can be understood as a generalization of the spatial search for objects.

The concept of saliency is inspired by psychophysical research on human eye movements (Section 27). Spatial search can be divided into bottom-up and top-down, analogous to bottom-up segmentation and top-down segmentation, Sections 14 and 9, respectively. In bottom-up saliency, we preprocess the image without any prior knowledge of object representations; in top-down saliency we search a preprocessed image for certain characteristics that correspond to our expected objects.

The output of the spatial search process is an intensity map, called *saliency map*, whose values represent the degree of saliency for a location. The size of the saliency map is often smaller than the image size. One selects the peaks from the maps and those are then our coordinates.

Bottom-Up Search Models for bottom-up saliency perform a feature extraction similar to the one introduced in Section 7. Then, the features are graded based on their chromatic contrast and intensity. Those grades are then combined to form the saliency.

There exist different models for saliency. Some attempt to model the spatial search behavior of humans, among which the most prominent one was developed by Itti and Koch, e.g.

`http://www.saliencytoolbox.net/`

`http://jevois.org/`

Those models are sometimes employed to estimate the efficiency of product advertisement. They are however too slow for use in robotics due to the use of time-consuming filtering operations. Robots require faster saliency models for which there exist some, but they do not have found widespread use as they do not point quickly enough to object candidates. We refer to Rosebrock's blog for that:

`https://www.pyimagesearch.com/2018/07/16/opencv-saliency-detection/`

Top-Down Search Here we search with prior knowledge of our target objects. The object recognition models introduced in Section 11 essentially correspond to this idea. There exist also a number of Deep Nets that contain an explicit saliency mechanism, but those are difficult to distinguish from the regular networks performing recognition. For very simple objects, we have mentioned some techniques in Section 10.5.

H Pattern Recognition

Traditional pattern recognition methods are particularly used with the feature engineering approach, but also Deep Nets profit sometimes from those methods. After we have extracted features from images (with the engineering approach), we deal with a two-dimensional array, that represents images in two principal ways.

One Vector Per Image: each image is represented by a single vector (of same dimensionality). The list of image vectors is an array `IMG` whose format is `[nImg nDim]`, number of images $\times$ number of feature dimensions. To this matrix we can apply pattern recognition methods as usual.

Multiple Vectors Per Image: each image is represented by multiple vectors (of same dimensionality). For multiple images we amass the array `DSC`, as in Section 13.1. The array is of size `[ntDsc nDim]`, number of total features for all images $\times$ number of feature dimensions. There are two ways to deal with that variable size format. There exists the Adaptive Boosting classifier that can deal with this variable format. Or one finds a transformation that results in the above format 'one vector per image' such as the quantization method introduced in Section 13.

(In correspondence to https://www.researchgate.net/publication/336718267)

H.1 Classification

In a classification task we have - in addition to our list of image vectors `IMG` - also a label array `Lbl`, a one-dimensional array `[nImg 1]` with entries corresponding to the class (group or category) membership. Then we can apply classifiers such as:

H.1.1 Support Vector Machines (SVM)

The Support Vector Machine (SVM) is a binary classifier that discriminates two classes only. To apply it to a multi-class problem with c classes, we train c one-versus-all (other) classifiers. Software packages typically do that automatically for us if array `Lbl` shows more than two classes.

In Matlab a SVM implementation is available as `fitcsvm` as binary classifier. The function `trainImageCategoryClassifier` deploys SVMs as default. If one classifies more than two classes than we can use `fitcecoc`, which uses SVMs as default:

```
MdCv    = fitcecoc(IMG, Lbl, 'kfold',5);
pcSVM   = 1-kfoldLoss(MdCv); % proportion correctly classifed
```

In Python existent as `svm` in module `sklearn`.

H.1.2 Random Forests

Random forests are ensembles of decision trees; they pool the decision of individually trained decision trees. They are particularly useful sometimes when the dimensions express quantized data, such as the word histograms generated in Section 13.

In Matlab run via `TreeBagger`:

```
MdCv    = TreeBagger(100, IMG, Lbl, 'OOBPred','on');
pcRF    = mean(1-oobError(MdCv));
```

In Python available as `RandomForestClassifier` in module `sklearn.ensemble`.

Random forest are sometimes used also for segmentation. For the action classification system introduced in Section 23.2 they work as classifiers applied to the entire image.

H.1.3 PCA & LDA

The Linear Discriminant Analysis (LDA) is a very efficient simple classifier - it is useful to obtain classification results quickly. It often works however only in combination with the PCA (Section H.1.4).

In Matlab it is available as `fitcdiscr`. In case it fails to produce results, one would apply the PCA and use `IMGred` instead of the full dimensionality (`IMG`).

```
MdCv    = fitcdiscr(IMG, Lbl, 'kfold',5);
pcLD    = 1-kfoldLoss(MdCv);
```

In Python available as `LinearDiscriminantAnalysis` in module `sklearn.discriminant_analysis`.

H.1.4 Dimensionality Reduction with Principal Component Analysis

The Principal Component Analysis (PCA) is a procedure that finds a more compact data space for the original data: it reduces the number of dimensions `nDim` of the data. This is not to be confused with quantization of the sample, which reduces the sample size. The following code piece shows how to apply it to some matrix, be that image vectors `IMG` or descriptors `DSC`:

```
coeff  = pca(DSC);              % [nDim, nDim] coefficients
nPco   = round(min(size(DSC))*0.7);  % suggested # of observations
PCO    = coeff(:,1:nPco);            % select the 1st nPco eigenvectors
DSCred = zeros(nDsc,nPco);           % allocate memory
for i = 1 : nObs,
    DSCred(i,:) = DSC(i,:) * PCO;    % transform each sample
end
```

The reduced array `DSCred` is now being used for classification above.

H.1.5 Ensemble Classifiers

An ensemble classifier is a classifier that combines the output of several different classifiers. The classifiers can be of the same type as we had mentioned in Sections 9.2.3 and 8.4. But they can also combine the output of different types of classifiers, e.g. the output of a Deep Net combined with the output of a Feature Engineering approach.

H.2 Clustering

Clustering algorithms are used to find groups in unlabeled data. There exist different types of clustering algorithms. The K-Means is the most popular due to its simplicity and relatively low complexity. It was exploited for image segmentation (Section 14.3.1) and for feature quantization (Section 13). Other clustering algorithms were essentially discussed in the section on unsupervised image segmentation (Section 14).

I Performance Measures

The performance of a recognition system can be characterized in various ways. This helps us understanding where it can be employed. Large systems usually recognize with higher accuracy, but also take more time to arrive at their classification decision and require more computational power - they are feasible to employ only if the computer is well powered. Smaller systems recognize with less accuracy, but consume less energy and can therefore be employed in embedded systems. This and other trade-offs need to be quantified somehow. Here we introduce measures for that. We begin with recognition rates (Section I.1) and then introduce issues of complexity and recognition duration (Section I.2).

I.1 Recognition Rate

For each recognition process there exists a different measure to precisely characterize the recognition success. Those measures are typically from the fields of Information Retrieval and Signal Detection Theory. We firstly introduce how to describe classification rates when three or more categories are involved (Section I.1.1). Then we specify how that is done for detection tasks, which represent a two-class problem with imbalanced class sizes (Section I.1.2). For regions, there exist also specific measures (Section I.1.3). And for retrieval we have already specified those measures and refer directly to Section 18.2.

I.1.1 Classification Accuracy

We can determine a classification accuracy if we know the corresponding class labels for our sample images or sample objects. Those labels are also called the *actual*, *known* or *true* labels; the ones assign by our classification system are called *predicted* or *estimated* labels. If the actual label and the predicted label match, then this is a correctly identified sample; if not, then it is an incorrectly classified sample. Then we can determine the classification rate, the classification accuracy. For example, for a logical array where `0`s and `1`s correspond to incorrectly and correctly classified samples, respectively, we simply take the average of the vector. Matlab offers the function `classperf` to calculate the rate and measures related to it.

This measure is suitable for a system discriminating between three or more classes. If we deal with a two-class problem, a binary classification task, then this is not a sufficient measure, in particular if the class sizes are imbalanced, and we need to calculate other measures, see Section I.1.2 below.

Training and Testing To properly estimate the classification accuracy of a system, we need two sets of samples. One set for training the system only, and one set for testing the system. For each one we can determine a classification accuracy, a training accuracy and a testing accuracy. The testing accuracy is the actual *prediction accuracy* and it tells us more about the true classification capabilities of the system. The training accuracy in contrast is of limited significance, because training a system can be overdone resulting in some cases in 100 percent correct classification. In particular Neural Networks are capable of easily achieving 100 percent as they have so many parameters. This over-training is also call over-fitting sometimes. Over-fitting can be detrimental sometimes. It is better to stop training before it starts over-training the system.

Folding In classification competitions, the organizers provide those two sets in order to determine a prediction accuracy that is comparable across different systems. In case of absence of such a partitioning, then we need to split the data ourselves. There we face a trade-off: if we take much training data, then the value for testing accuracy is unreliable due to the small amount of data left after the split; if we take little training data, then the testing accuracy is more reliable, but we have a less well trained system. The proper procedure is therefore to partition the image set into n equal folds and reserve one fold for testing and use the remaining $n-1$ folds for training; then we cycle through the folds, generating n prediction accuracies, which we then average. This is evaluation procedure is also known as *cross-validation*. Typically one chooses five folds, also called then five-fold cross-validation. In Matlab we can generate the indices for cross-validation with the function `crossvalind`.

Confusion Matrix To further understand how a system classifies, we can look at the confusions between classes. For that we generate a table (matrix) in which we enter predicted versus actual. In Matlab that can be done with function `confusionmat`.

I.1.2 Detection Accuracy

When we characterize a detection system, such as a face-detection algorithm, then counting the correctly identified target locations and calculating an accuracy thereof is only a first step. Observing that accuracy value only can be misleading - also for a binary classification task. The reason is that the detection system may return an excessive amount of candidate locations and therefore cover all actual, true object locations; but it will also have generated many falsely identified locations and those can be detrimental sometimes. For instance, in Google Street View, faces and license plates are blurred to avoid complaints and law-suits; if however all candidate locations were blurred, then Street View imagery may appear rather unspecific - in rare cases it does in fact.

To describe the recognition rate more precisely, we need to include the falsely identified locations. We count the following three occurrences for which there exists different terminology (label 1 and label 2):

	Label 1	Label 2	Example
TP	true positive	hit	target object locations correctly identified
FP	false positive	false alarm	non-objects wrongly identified as target object
FN	false negative	miss	target object locations not identified

We further have the total number of objects, P, which should equal TP + FN. And we have the total number of predictions, P', equaling TP + FP . Now we calculate two values, the precision and recall value:

Precision	TP / P'	P' = TP + FP	
Recall	TP / P	P = TP + FN	also known as True Positive Rate or Sensitivity

From those two measure we calculate a single value, the F1 score:

F1 score	2 * (Precision*Recall) / (Precision + Recall)

Ideally, recall and precision show values that are as high as possible, then the F1 score would also be high. If there is an excessive amount of false positives, then that results in a low precision and that drags down the F1 score.

I.1.3 Region Matching

When we predict regions, such as in segmentation or object localization, then we want a measure that characterizes the degree of overlap between the annotated image - the annotation mask -, and our predicted region, the prediction mask. There exist several measures. A strict, straightforward measure is the Jaccard index, wiki Jaccard_index , also known as *Intersection-Over-Union*, abbreviated *IoU* sometimes. But there are also measures based on Signal Detection Theory as introduced above. We give directly a code example for those, where `BWann` and `BWprd` are our annotation mask and prediction mask, respectively. We use two overlapping rectangles as regions for simplicity.

```
clear;
% ---   Annotation
BWann               = zeros(20,20);
BWann(3:12,7:14)    = 1;               % a region labeled as some object
% ---   Prediction
BWprd               = zeros(20,20);
BWprd(5:16,9:16)    = 2;               % the predicted object
IxReg               = find(BWprd);
nPixPrd             = length(IxReg);   % region size = predicted pixels

% ----  Annotation properties
```

```
szI    = size(BWann);
nPixImg = prod(szI);
nPixAnn = nnz(BWann);           % # of positive instances
nNeg   = nPixImg - nPixAnn;   % # of negative instances

TP   = nnz(BWann(IxReg));    % true positives  (correctly predicted, # of hits)
FP     = nPixPrd - TP;      % false positives (wrongly predicted)
FN   = nPixAnn - TP;       % false negative  (missed)
TN   = nNeg - FP;            % true negatives  (correctly rejected)

%% ======   Sensitivity, Specificity, Precision  =======
FPR         = FP / (FP + TN);       % false-positive rate
sens        = TP / nPixAnn;         % sensitivity
spec        = 1 - FPR;              % specificity
spec2       = TN / (TN + FP);

prec        = TP / (TP + FP);       % precision
reca        = TP / (TP + FN);       % recall = sensitivity

FNR         = FN / (TP + FN);       % false negative rate
PWC         = (FN + FP) / (TP + FN + FP + TN); % prop. wrong classifications
Fmeasure    = (2 * prec * reca) / (prec + reca);

%% ======   Jaccard     =======
nJnt        = length(union(IxReg, find(BWann)));
jacc        = TP / nJnt;

%% -----    Plot    -------
figure(1); clf;
imagesc(BWann + BWprd);
```

In some situations we compare only the bounding box of regions, in which case Matlab offers a function bboxOverlapRatio.

I.2 Complexity, Duration

There exists different measures to characterize the 'convenience' of a recognition system. Here we can again distinguish between the feature engineering and the feature learning approach (Fig. 8). In the engineering approach one sometimes identifies the most complex process with regard to the number of image pixels. In the neural network approach, one specifies learning duration, memory usage and inference time. We elaborate by starting with Deep Nets.

I.2.1 Feature Learning (Deep Nets)

For Deep Networks the most informative indicator of complexity is probably the number of layers. The more layers, the longer the learning duration, the larger the number parameters and the longer the inference time.

- **Learning Duration:** it measures how long it takes to train the network weights. If a Deep Net is trained from scratch, it can take days and the duration is specified in that unit. If a Deep Net is trained with transfer learning, the duration is much shorter and is given in hours or sometimes in minutes.
- **Weight Parameters:** the number of weight parameters is rather given as the amount of memory used, in MegaBytes.
- **Inference Time:** measures how long it takes to propagate an image through the network to arrive at a classification decision or the segmentation output. This number is given in milli-seconds or micro-seconds and is usually measured for a 256x256 image.

The durations are typically given for operations carried out on CUDA cores. Thus, if we ran a Deep Net on a smaller system, an embedded system (without CUDA cores), then we will have much larger inference times.

I.2.2 Feature Engineering

As we deal with individual algorithms and phases in this approach, it is common to specify the most complex algorithm or phase. To specify the complexity of an algorithm, one uses the so-called *Big-O* notation (wiki Big_O_notation). It was used to specify the complexity of the segmentation algorithms (Section 14.3).

Sometimes, also the learning duration is given, measured in minutes. The inference time is specified also in relation to image size.

In Matlab we can use the functions `tic` and `toc`, or `clock` and `etime`, to measure durations:

```
tStart = clock;
sec     = etime(clock, tStart);
ms      = round(sec * 1000);        % milliseconds
```

In Python we employ functions from module `timeit`:

```
from timeit import time
tStart     = time.clock()
sec        = time.clock()- tStart       # duration in seconds
```

These timing measurements are generally inaccurate, as the CPU is reluctant to pay full attention to a single program and the measurements therefore tend to be somewhat longer. It is best to run a measurement multiple times and then take the average.

J Learning Challenges and Tricks [Machine Learning]

Here we overview some of the challenges in learning and its terminology (Section J.1), and then summarize learning tricks that are used to improve the prediction accuracy of a classification system (Section J.2).

J.1 Issues and Terminology

Learning requires some type of guidance and one typically distinguishes between two extreme types of guidance, one extreme actually being a lack thereof. We had introduced those types already previously, but are mentioned again for completion. Then we mention learning modii and the challenges of dynamic learning.

Supervised Learning: is a strict guidance, a teacher that clearly assigns true or false during the learning process. Examples are any classification method, such as Deep Nets for image or object classification (i.e. Section 8), or traditional Pattern Recognition methods using classifiers, see Appendix H.1.

Unsupervised Learning: is learning without guidance; it is a search for structure and densities in the data, as described in Appendix H.2; or the autoencoder as described in Appendix F.3.

Some systems are so complex that a clear distinction between those two types are difficult. For example the top-down segmentation networks used in Section 9 are based on the AutoEncoder, an unsupervised method, but the way they are employed for training segmentation falls under the supervised regime.

Learning occurs typically by presentation of multiple image samples; ideally of as many samples as possible. In some scenarios, when few sample images are available, we can either use transfer learning or are forced to use only one or few images.

Transfer Learning: is a method to train one task thereby profiting from another learned process. For example we can practice a classification task for a few images using a Deep Net that was trained on another set of images (Section 8.2). The experience of one system is transferred to the new system.
Shao et al. 2014, Transfer Learning for Visual Categorization: A Survey

One-Shot/Few-Shot Learning: is learning using a single image as an example, or only a few images as examples, i.e. for learning a new class.

Catastrophic Learning/Inference/Forgetting: is the failure to maintain old learned information during the attempt to acquire new information. In short, it is unlearning during dynamic learning, explained next.

In the simplest case, a system is trained once and then applied. Often however, the set of input classes changes during continued application such that the trained system does not reflect well anymore its original teaching after some time. For example during application, a trained class may not appear anymore, but a new one has started to appear. If those changes occur occasionally, then retraining the system may be sufficient. But if the changes outpace the learning duration, then it requires a more dynamic system. In particular in robotics, the challenge to keep up with environmental changes is huge. A house-hold robot is faced with some objects being continuously relocated, new objects appearing and disappearing; if a newly appeared object class has appeared, then that very much corresponds to the challenge of one-shot learning. Generally speaking, this adaptation challenge faces a *stability-plasticity* dilemma. For example Neural Networks that attempt to adapt to changes in input, risk to forget some of their previously trained classes, a phenomenon called catastrophic inference.

J.2 Techniques to Maximize Classification Accuracy

The following three techniques can be used with any classification method, with traditional or with modern (deep learning) methods.

Data Augmentation is a method to artificially increase the collected learning material (Section 2.3) by slightly modifying the images. Data augmentation aims to increase the (class) variability as to mimic the variability present in real-word classes - think of how many types of cars or chairs there exist. Those manipulations consist of stretching the images, cropping them, rotating, flipping, etc. Some of those manipulations are introduced in Section 21.1.2 (see Appendix **P.18.2** for example code).

In Matlab one would use functions such as `imtranslate`, `imcrop`, `imresize`, `imwarp`, `flipud`, `fliplr`, etc.

In Python all those functions are found in module `skimage.transform`.

In PyTorch the functions are provided in module `torchvision.transform` starting with the term `Random`, for example `RandomResizedCrop`, as introduced in 8.2, see code in **P.5.2**.

Drop Out is a technique to prevent overtraining. It was originally developed for Neural Networks, but can also be applied to other classification methods. In Neural Networks, during each learning step, a small percentage of neural units is eliminated of a layer. This is specified as drop-out rate. The technique prevents the classifier from overtraining, from becoming overly specific to certain input patterns that are not reflecting the classes well anymore.

Hard Negative Mining is a technique to collect images that are difficult to discriminate. In a general form, one can collect an entire image class that appears to be difficult for a target class, such as collecting vegetation images to train face detection (Section 10.1). In a more specific form, one focuses on samples that trigger false alarms in a classifier. For instance, if we train a classifier to categorize digits, than we observe and collect (mine) those digits that confuse one category, that is those samples that trigger false alarms. For instance, we collect those digit samples that confuse class '1', which could be '7's or '4's; or for category '3' it could be '2' or '8'. Then we re-train the classifier with those 'hard negatives'. This is systematically exploited in so-called cascade or boosting classifiers, such as the one trained to detect faces (Section 10.1).

K Distance, Similarity

We can measure distance or similarity between two points or vectors in various ways. A distance measure describes how far apart two objects are, in two or more dimensions; a similarity measure describes how close they are. The measurements can be done with different degree of precision; the more precise the measure, the more complex is its computation. Those measures are sketched in Section K.1.

A complex measure can be an obstacle sometimes, for example for retrieval of shapes or images (Sections 18 and 16), where we sometimes deal with large image sets; or even for taking the pair-wise distances between all segments of a large image (a satellite or medical image for instance), it is impracticable to compute all distances for reason of time and memory. In many of those situations, it is however sufficient to know the distance only to the closest points, also known as *nearest neighbors*; sometimes it is sufficient to know which ones are the nearest neighbors without requiring the computation of the exact distance to them. In that situation, and under certain conditions, one can use methods that also work very fast; the methods are known as *nearest neighbor search* (NNS), to be introduced Section K.2.

K.1 Measures

Distance and similarity can be measured with functions that are called `pdist` in many software packages, pair-wise distance measurements for a single list of vectors, or `pdist2` for two different lists of vectors (of same dimensionality). The default is typically the Euclidean distance, but other distances or similarities can be specified.

The Euclidean distance is the most precise metric, but also complex because taking the square and root operations can become time consuming if one compares thousands of points. In some case we do not need to know the precise distance, but it is sufficient to know which object is the closest, also called *nearest neighbor*; in that case we can try the *CityBlock* distance, also called *Manhattan* or *TaxiCab* distance sometimes. It takes only the sum of the absolute differences between dimensions (coordinates). For example, to find the nearest pixel in relation to some other one, the CityBlock distance is sufficient, as pixels lie on a defined grid.

A downside of the Euclidean distance metric is, that for higher dimensionality, it may become imprecise because in a high-dimensional space points become very distant, an issue related to the *curse of dimensionality* sometimes. From what dimension on this may be the case, depends to some extent on the data size, but from dimensionality 10 onward one should keep this potential downside in mind. From dimensionality 15 on, it is perhaps a serious issue. It is for that reason why nearest neighbor search is only precise and efficient when it is of dimensionality smaller than that limit.

Similarity metrics do not appear to suffer from this issue as variables are multiplied with each other giving rise to a non-linear space. Note that strictly speaking, a similarity measure is not a metric, but the term metric is used nevertheless. A similarity measure is often defined as having value equal one, if both vectors are the same and is often zero if one dimension shows a large difference. The dot product is a popular similarity measure and can be applied with different types of normalization.

We show in Matlab how to program these measures. `DAT` is our (artificial) set of vectors, representing some objects. We loop through each sample and take the distance (or similarity) to all other samples. For distances we set the own sample to infinity, and for similarities to zero. Then we sort and visualize the ordering (per sample).

```
clear;
nDim        = 3;
nSmp        = 50;
DAT         = single(randn(nSmp,nDim));      % artificial data
% --- for normalization/scaling
DLen        = sqrt(sum(DAT.^2, 2));       % length of each observation vector
% --- for cosine similarity
DLN         = DAT ./ DLen(:,ones(1,nDim));
% --- for Pearsons correlation
DF2M        = bsxfun(@minus, DAT, mean(DAT,2)); % difference to mean
DPC         = DF2M ./ DLen(:,ones(1,nDim));     % E [-1 1]
```

```
%% SSSSSSSSSSSSSSSS    LOOP SAMPLES     SSSSSSSSSSSSSSSSSS
% pairwise distances/similarities
for i = 1:nSmp
    % ======    Distances   ======
    Dcbl    =  sum(abs(bsxfun(@minus,DAT,DAT(i,:))), 2);    % city-block
    Deuc    = sqrt(sum(bsxfun(@minus,DAT,DAT(i,:)).^2, 2)); % Euclidean
    Dseu    =      sum(bsxfun(@minus,DAT,DAT(i,:)).^2, 2);  % squared Euclidean

    % ======    Sort Distances   ======
    Dcbl(i) = inf;                      % set own to infinity
    Deuc(i) = inf;
    Dseu(i) = inf;
    [D Ocbl] = sort(Dcbl,'ascend');  % sorted starting with nearest neighbors
    [D Oeuc] = sort(Deuc,'ascend');
    [D Oseu] = sort(Dseu,'ascend');

    % ======    Similarities   ======
    Scos    = 1 - (DLN * DLN(i,:)');  % cosine similarity
    Spec    = 1 - (DPC * DPC(i,:)');  % pearson's correlation

    % ======    Sort Similarities   ======
    Scos(i)     = 0;                    % set own to 0
    Spec(i)     = 0;
    [S Ocos] = sort(Scos,'descend');
    [S Opec] = sort(Spec,'descend');

    % -----     Plot    -----
    if 1
        % visualizing order as image
        I   = [Ocbl Oeuc Oseu Ocos Opec]';
        figi(1);imagesc(I);
        pause();
    end
end
```

This is also called *exhaustive* search, because we determine the distances between all pairs of points. This can be exploited for accuracy or for clustering processes. If we do not need to know all pairs of distances, perhaps only the closest five points, then a nearest neighbor search can be used, coming up next.

K.2 Nearest Neighbor Search (NNS)

wiki Nearest_neighbor_search

Nearest neighbor search is a method for fast search of the closest points for a given query point. If the dimensionality of our points is 9 or less, and/or if there are sufficient points, then one can utilize a technique called KD-Tree. The technique partitions the space as a decision tree does; when a query point is presented, its nearest neighbors can be found by simply applying that decision tree.

If the dimensionality is 10 or larger, then distance measurements become increasingly inaccurate (due to the above mentioned curse of dimensionality). One can also use techniques that are designed as so-called *approximation* methods, such as locality-sensitive hashing (LSH) or best-bin first; the wikipedia pages are a good starting point to find some orientation.

We merely demonstrate how to use the KD-Tree in programming languages, where it is often part of a general search function for nearest neighbor search.

In Matlab we use the function `knnsearch`, which decides automatically whether it merits to use an optimized search or an exhaustive search. If we determine neighbors once, e.g. we compare the two lists `DAT1` and `DAT2`, then we would write:

```
[NEB DI]    = knnsearch(DAT1,DAT2, 'k',2);    % decides automatically
```

where `NEB` and `DI` contain the neighbor indices and distances, respectively. If we compare one list repeatedly to different lists, then we create a NNS model separately, called `Mns` here, and then apply it multiple times:

```
Mns         = createns(DAT1);                   % create 'model'
NEBa        = knnsearch(Mns,DAT2, 'k',2);    % apply model
NEBb        = knnsearch(Mns,DAT3, 'k',2);       % apply again
```

The decision as to which mode it uses is still done automatically.

In Python we work immediately with the explicit model. The term *fit* comes from the field of machine learning: a model is fitted to the data, in our case a decision tree (if appropriate).

```
Md          = NearestNeighbors(n_neighbors=2).fit(DAT1)
DIS, NEB    = Md.kneighbors(DAT2)
```

Output arguments are switched in this case: first distances are returned, then neighbors.

L Change Detection

Change detection is the task of discriminating between crucial and insignificant changes when we observe a scene, either continuously or at different moments in time. Examples are:

In surveillance, the aim is to detect objects entering the scene, while ignoring illumination changes caused by shadows for instance. The target object is of high contrast typically, the unimportant changes can be very distracting. The challenge becomes more intriguing when objects stop in the image for a while and start moving later again, i.e. a pedestrian sitting down and leaving later; in that case we detect motion *onset*.

In remote sensing, the aim is to detect areas that have changed, i.e. we wish to detect forest that has been cut down and turned into agricultural fields. Changes in this task can be obvious, but we deal with alignment issues as satellite photos typically differ in orientation.

In medical image analysis, the aim is to detect subtle changes between recordings taken at different times. The change can be structurally subtle, e.g. detecting when the vessels or arteries have slightly different configurations between images.

In surveillance, the recording is continuous; in remote sensing and medical image analysis we are given perhaps only two images separated in time. The examples should have made clear that the solution to the task requires a specific algorithm and is often linked to subsequent processes such as change understanding, motion tracking, object recognition etc. For that reason change detection is carried out with different methods.

Classical methods use primarily the difference between images combined with image manipulations to emphasize the differences. The difference can be as simple as the absolute difference between the corresponding pixels, e.g. Section 5. Image manipulations consist of filtering or morphological processing, i.e. Sections 7 and 15. A review of those is provided by Radke et al., 2005, available at:

https://msol.people.uic.edu/ECE531/papers/Image%20Change%20Detection%20Algorithms-%20Survey.pdf

Modern methods use Deep Neural Networks and are certainly the preferred choice if there is sufficient labeled data and computational power. We then can train a segmentation network as introduced in Section 9.

There exist several databases, the largest one to our knowledge is:

http://changedetection.net/

M Resources

Databases

- **ImageNet**, the famous one, used for learning the DeepNets:
  http://image-net.org/
- **Cityscapes**, traffic scenes, used for testing segmentation networks:
  https://www.cityscapes-dataset.com/

Sites that list databases:
http://homepages.inf.ed.ac.uk/rbf/CVonline/Imagedbase.htm
https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research

The Kaggle website offers databases and a way to compare results by participating in competitions:
https://www.kaggle.com

Libraries

List pointing toward software packages:
http://homepages.inf.ed.ac.uk/rbf/CVonline/SWEnvironments.htm

Open-CV is probably the largest (open-source) library. It contains binaries and their corresponding source code, written in C and C++ :
https://www.opencv.org/
https://github.com/opencv
There does not exist separate documentation yet for how to use OpenCV through Python, so you need to browse the tutorials:
https://docs.opencv.org/master/d6/d00/tutorial_py_root.html

First steps to program CUDAs in Python:
https://devblogs.nvidia.com/numba-python-cuda-acceleration

Code for Matlab can be found in particular on:
http://www.mathworks.com/matlabcentral/fileexchange

Sites with code resources by individual persons:
https://www.learnopencv.com
https://www.pyimagesearch.com
http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/index.html
http://www.aishack.in
http://www.vlfeat.org

Varia (Programming Support)

Summary of function names for Matlab, Python, etc:
http://mathesaurus.sourceforge.net/matlab-python-xref.pdf

On-line compendium:
http://homepages.inf.ed.ac.uk/rbf/CVonline/

Coding in Python:
https://pythonawesome.com/
http://programmingcomputervision.com/

Datasets, job offers, conference information, etc.:
http://www.computervisiononline.com/

Goes along with the Image Processing book by Gonzales et al.:
http://www.imageprocessingplace.com/

Embedded Systems

Steps toward embedded systems:

https://github.com/uTensor/uTensor

Community:

https://www.embedded-vision.com

Ideal for exploring:

https://openmv.io/

http://www.jevois.org

N Color Spaces

wiki Color_space

There exist multiple color spaces, each one with certain advantages and disadvantages. The most common one is the RGB space and can be exploited to do some (color) segmentation, but segmentation of certain objects is easier carried out in other color spaces, see Table 6. To obtain those color spaces, we convert the RGB values using specific transformations, sometimes very complex ones. In Matlab this is done using commands such as `rgb2hsv` for example:

```
Ihsv     = rgb2hsv(Irgb);
```

In Python we find those conversion functions in module `skimage.color`:

```
Ihsv  = skimage.color.rgb2hsv(Irgb)
```

In OpenCV, those conversions are carried out with the function `cvtColor` and the specifier `COLOR_XXX`:

```
Ihsv    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2HSV)
```

Table 6: Commonly used color spaces. f: some complex function. Some of those transforms are given as code examples in **P.25**.

Space	Calculation	Comments
RGB wiki RGB_color_model	measured	- high correlation between channels; significant perceptual non-uniformity; mixing of chrominance and luminance data
normalized RGB wiki Rg_chromaticity	with $s = R+G+B$, then $r = R/s$ $g = G/s$ $b = B/s$	+ reduces shadow and shading effects (under certain assumptions); essentially a 2D space because the 3rd component is = 1-the other two Usage: computer vision `bsxfun(@rdivide,I,sum(I,3));`
HSI (HSV HSL) Hue Saturation Intensity (HS Value HS Brightness) wiki HSL_and_HSV	$H = f(R,G,B)$ $S = 1 - 3\frac{\min(R,G,B)}{R+G+B}$ $I = \frac{1}{3}(R+G+B)$	+ gives the user a good impression about the resulting color for a certain color value; describes color with intuitive values, based on the artists idea of tint, saturation and tone. - hue discontinuities; computation of I, V or L conflicts with properties of color vision. • Hue: dominant color such as red, green, purple and yellow. • Saturation: colorfulness in proportion to its brightness. • Intensity, Lightness or Value: related to luminance: Usage: design and editing (e.g. within graphics design tools) `rgb2hsv`, **P.25.1**
YC$_b$C$_r$ wiki YCbCr	Y = more complex than eq. 1 $C_b = R - Y$ $C_r = B - Y$	Y also called *luma* here C_b, C_r also called *chroma* here Usage: digital video; image compression `rgb2ycbcr`, **P.25.2**
CIELAB or **CIE Lab** (related: **CIELUV**) wiki Lab_color_space	complex	+ perceptually uniform: small changes to a component do not change perception much - computationally intensive. • L: lightness • a: green-to-magenta • b: blue-to-yellow Usage: design and editing `makecform`, `applycform`

For complex spaces we need to call two commands in Matlab:

```
LabForm = makecform('srgb2lab');       % prepares transformation structure
Ilab    = applycform(Irgb, LabForm);   % applies above structure
```

The detailed formulas for the conversions are listed here

`https://de.mathworks.com/help/vision/ref/colorspaceconversion.html`

but the actual help pages in the Matlab programming environment offer the nicest and most compact overview.

N.1 Skin Detection

Skin detection is the process of segmenting skin from its background. It is used for example for face detection in videos; or in medical applications, i.e. to distinguish between tumor and skin, in which case the skin is background and the tumor is foreground. There are many algorithms for skin detection, often geared towards a specific objective. For instance for face detection in videos, one requires a fast algorithm. Often, the algorithms involve a transformation from the RGB color space into another color space in which the red tones are (hopefully) isolated or emphasized. However, the advantage of those other color spaces has been occasionally questioned and one may therefore start developing an algorithm using the 'original' RGB space first, in order to see whether it is sufficient. For instance the following fast skin-detection algorithm is based on a set of hard rules in RGB space (`I` is a RGB color image):

```
% --- Minimum Range Condition
Mn      = min(I,[],3);
Mx      = max(I,[],3);
Brg     = (Mx-Mn)>15;

% --- Minimum Value Condition
R       = I(:,:,1);
G       = I(:,:,2);
B       = I(:,:,3);
Bmn     = R>95 & G>40 & B>20;

% --- Dominant Red Condition
Bc3     = abs(R-G)>15 & R>G & R>B;

% --- All Conditions
S       = Brg & Bmn & Bc3;
```

`S` is a black-white (binary) image with ON pixels corresponding to skin.

For tumor/skin discrimination in medical images, the following color information has been used (Vezhnevets et al. 2003):
- The blue channel of the (original) RGB space
- The ratio between the red and the green channel of the RGB space
- The a channel of the Lab color space
- The H channel of the HSV space, see the face tracking example in Appendix **P.16.3**

N.2 Other Tasks

The red component of the YCbCr space can be exploited for the detection of tail lights and road signs in road scenes. For instance, turn a frame `Frm` into that color space and threshold its red-chroma values at some appropriate value:

```
Frm     = cv.cvtColor(Frm, cv.COLOR_RGB2YCR_CB)
BWhih   = Frm[:,:,2] > 140
```

The segmentation of natural objects, such as leafs, fruits and landscapes, is sometimes done with the Hue map of the Hue-Saturation-Value (HSV) space.

N.3 Links

FAQ with the conciseness for programmers:
http://poynton.ca/PDFs/ColorFAQ.pdf

O Python Modules and Functions

Python offers functions for computer vision and image processing in various modules. Some functions exist in multiple modules. Most functions can be found in module `skimage`, which also contains a number of graphics routines. Its documentation can be found at `http://scikit-image.org`. A number of functions can be found in submodule `numpy.ndimage`. For signal processing and clustering we employ `scipy.signal` and `scipy.cluster` respectively.

Table 7: Overview of the skimage module

Utility Functions	**Comments**
`img_as_float`, `img_as_xxx`	casting (type conversion)
`dtype_limits`	returns minimum and maximum intensity
Submodule	**Functions/Comments**
`color`	all types of conversions, e.g. `rgb2gray`
`data`	example images such as `coins`
`feature`	detection of blobs: `blob_dog`, `blob_xxx` edges: `canny` corners: `corner_harris`, `corner_xxx` maxima: `peak_local_max`
`filters`	edge detection (Prewitt, Sobel, ...); threshold determination (`threshold_otsu`); complex filters
`io`	input/output, e.g. `imread`
`measure`	various functions, e.g. `find_contours` (iso-contours), `regionprops`, `label`, `points_in_poly`, `ransac`
`morphology`	black-white: `binary_closing`, `binary_xxx` gray-scale: `dilation`, `erosion`, ... extrema: `h_minima`, `h_maxima`, `local_minima`, `local_maxima` various: `label`, `watershed`
`segmentation`	`watershed`, `quickshift`, `clear_border`
`transform`	`pyramid_reduce`, `pyramid_expand`, `resize`

Table 8: Overview of the `scipy` module, `https://docs.scipy.org/`.

Submodule	**Functions/Comments**
`cluster`	`vq.kmeans`
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
signal	`convolve`, `convolve2d`
`spatial.distance`	`pdist`, `squareform`
special	Special functions
stats	Statistical distributions and functions

Table 9: Matlab-Python equivalents.

Matlab	**Python**
`padarray(A,[2 2])`	`numpy.pad(A,2,'constant',constant_values=0)`

P Code Examples

The code examples are written as scripts to be run in a console, e.g. loading them into the Matlab or Spyder editor and then running them with key 'F5'. They are meant to provide a visually obvious access to the code blocks - unlike the majority of code examples on GitHub, that are written as stand-alone examples with parameter specifications to be given on a single command line, a format whose processing flow is sometimes hard to understand. Some of our code examples are modifications of the common examples found on GitHub or in the PyTorch tutorials, and our examples were adapted a bit to the dictionary that we use, explained below. Not all code examples are fully optimized for speed as I gave priority to readability sometimes.

Installation/Versions
For Matlab we had used the 2015a distribution.

For Python/Spyder we used the Anaconda software, https://www.anaconda.com, with Python version 3.6 initially, and recently 3.7; for PyTorch we used version 1.0.1 but switched to 1.2 recently; for OpenCV version 4.0 or higher.
Additional packages were installed by opening an Anaconda prompt and then using `conda` or `pip` commands as follows:

Skimage (https://scikit-image.org): `conda install -c conda-forge scikit-image`

Skilearn (https://scikit-learn.org): `conda install scikit-learn`

PyTorch (https://pytorch.org/): `conda install pytorch torchvision cudatoolkit=9.0 -c pytorch`

OpenCV (https://opencv.org/): `pip install opencv-python`

Code Shading

```
Matlab code.
```

```
Python code.
```

```
Python code in which functions of OpenCV are called. In many of those code snippets, we omit
the plotting part, as it is exactly the same as in Python.
```

Code Dictionary The naming of our variables is most similar to the Leszynski naming convention, which is a variant of the Hungarian notation. Names express intent not data type. We use camel case, which uses both lower and upper case without underscores. We generally start a name with lower case(s) for single values or instantiations, i.e. parameter values or specific object instances. We generally start a name with upper case(s) for matrices, arrays and structures, and as the majority of variables belong to those categories, we use more upper case than lower case; many variables are all capitals. The more specific usage is as follows:

images, frames, maps	`I`	`Irgb`, `Ihsv`, `Igry` (gray-scale), `Iblob` (blob image)
	`F`	`Frgb`, `Fhsv`, `Fgry`
	`M`	`Medg` (edge), `Mrdg` (ridge)
	`BW`	black-white map, `BWrdg`, `BWriv`, ...
vectors, counters, numbers	`B`	logical vectors/arrays
	`c`	counter: `cF` feature counter
	`n`	number of something: `nFrm`, number of frames; `nImg`, number of images
	`N`	array of numbers of something: `Nfet`, number of features for a set of images
indices	`ix`	single index: `ixImg`, index of an image
	`Ix`	array of indices: `IxImg`, image indices
	`rw, Rw`	row indices
	`cl, Cl`	column indices
	`Rg`	interval of indices, e.g. [3,4,5,6]
length, size	`len`	length of X
	`sz`	`szI`, image size; `szKrn`, kernel size
structures	`Stc`	`StcElm`, structuring element; `StcTrk`, track record
lists	`aX`	list of X for one image; `aCnt`, list of contours for an image
	`AX`	list of X for multiple images; `ACnt`, list of contours for image collection

Matrices start with a capital letter, often they are fully capitalized:

`DSC`	descriptors of size `[nDsc nDim]` [# of descriptors x # of dimensions]
`Pos, Pts`	positions or points of size `[nDsc 2]`
`DM`	distance matrix

Special counters are:

`rr, cc`	row and column counters

The sorting output of an array of values `V` is denoted with trailing lowercase 'o' for the values, and capital 'O' for the ordered indices:

```
[Vo Ov] = sort(V);
```

Other abbreviations:

`fet`	feature: `aFet`, list of features
`mtc`	matched: `aPtsMtc`, matched points; `DSCmtc`, matched descriptors
`horz`	horizontal
`vert`	vertical
`cat, catg`	category
`lb`	label: `LbCat`, array of category labels
`dis`, `diq`, `dib`	distance (Euclidean), square Eucl. dist, city-block dist.

P.1 Loading/Saving an Image or Video

Loading and saving an image in Matlab:

```
Irgb  = imread(fileName) % reads rgb it is rgb, otherwise gray-scale
Igry  = rgb2gray(Irgb) % convert to gray-scale
imsave(Irgb, fileName) % will deduce format from extension [or specify]
```

Loading, playing and saving a video in Matlab:

```
%% ---------    Movie Info      ---------
Vid         = VideoReader(pathLoad);            % movie 'handler'
[wth hgt]   = deal(Vid.Width,Vid.Height);
nFrm        = floor(Vid.duration * Vid.frameRate);  % number of frames

%% =========    Load Movie  =========
Vid.currentTime = 0;                        % set to 0 [in seconds]
fprintf('Reading sequence...' )
MOC(1:nFrm) = struct('cdata', zeros(hgt,wth,3,'uint8'), 'colormap',[]);
f=0;
while hasFrame(Vid) && Vid.currentTime<2    % we take only two seconds
    f       = f + 1;
    MOC(f).cdata =  readFrame(Vid);
end
fprintf('done\n');

%% ---------    Play the Movie (within Matlab)  --------
hf = figure(5);
set(hf,'position',[5 5 wth hgt]);
movie(hf, MOC, 1, Vid.frameRate);       % plays sequence

%% =========    Save Movie  =========
fprintf('Now writing...');
VidWrt      = VideoWriter(pathSave);
open(VidWrt);
for i = 1:f
    I       = MOC(i).cdata;
    writeVideo(VidWrt, I);
end
close(VidWrt);
fprintf('done\n');
```

Loading an image in Python in two popular ways, with skimage and with PIL (Python Imaging Library). Note that the conversion to a PyTorch tensor switches the dimensions and normalizes the values to a range between 0 and 1.

```
from skimage.io import imread, imsave
from skimage.color import rgb2gray
from PIL import Image

path   = 'C:/IMGdown/SILU/Images/airplane/airplane00.tif'

#%% ----    Skimage     -----
Irgb    = imread(path)          # numpy array [row/column/color] E [0..255]
Igry    = rgb2gray(Irgb)        # rgb to gray conversion

imsave('C:/ztmp/ImgSave.jpg', Irgb) # saves image as jpeg

#%% ----    PIL     -----
IrgbPIL = Image.open(path)      # reads into PIL format

#%% -----   PyTorch     -------
from torchvision import transforms
Trf2Tens = transforms.ToTensor()    # create transformation method
IrgbTo   = Trf2Tens(IrgbPIL) # switched and normalized: [color/row/column] E [0..1]
```

Loading a mp4 video in Python:

```
import imageio
filename = 'pathToVideo.mp4'

#%% -----   Video Pointer & Info      -------
vid     = imageio.get_reader(filename, 'ffmpeg')
inf     = vid.get_meta_data()       # video info
szI     = inf['size']
nF      = inf['nframes']
print('#frames %i   fps %i  [%i %i] dur=%1.2f seconds' % \
      (nF, inf['fps'], szI[0], szI[1], inf['duration']))

#%% -----   Video Frames      -------
from matplotlib.pyplot import *
IxFrm = [10,20,30,40]               # a few selected frames
for f in IxFrm:
    Irgb    = vid.get_data(f)       # read a frame
    figure()
    imshow(Irgb)
    title(f'frame={f}', fontsize=20)
```

Loading an image with OpenCV through Python. Note that with OpenCV, the three chromatic channels are loaded in reversed order than usual: BGR not RGB.

```
import cv2

fp   = 'C:/IMGdown/SILU/Images/airplane/airplane00.tif' # image file path

# ---- cv2.imread loads image into a numpy array:
Igry = cv2.imread(fp,cv2.IMREAD_GRAYSCALE) # gray-scale
Ibgr = cv2.imread(fp,cv2.IMREAD_COLOR)     # loads channels as BGR: blue/green/red
Irgb = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2RGB)    # convert to RGB for displaying

#%% -----   accessing/setting individual pixels     -------
vBGR    = Ibgr[100,100]         # [3,]
Ibgr[100,100] = [255,255,255]
# faster solution:
v       = Ibgr.item(10,10,2)    # red value at 10,10
Ibgr.itemset((10,10,2),100)     # setting red value to 100

#%% -----   accessing/setting regions     -------
Roi = Ibgr[50:100, 50:100].copy()   # extracting a region
Ibgr[30:60,30:60] = [255,255,255]   # setting a region to one value

#%% ---    Separating into BGR channels: discouraged because slow   --------
B,G,R   = cv2.split(Ibgr)
Ibgr    = cv2.merge((B,G,R))

cv2.imshow('color',Ibgr)        # creates a separate window!
cv2.imshow('gray',Igry)
cv2.imshow('selected',Roi)

k   = cv2.waitKey(0) & 0xFF     # waits for a key press
cv2.destroyAllWindows()
```

Playing and writing a video in Python using the OpenCV functions:

```
import cv2 as cv
import numpy as np
#%% ----------------    Video Info      -------------
vidName = 'C:/Program Files/MATLAB/MATLAB Production Server/R2015a/toolbox/vision/visiondata/atrium.avi'
Cap     = cv.VideoCapture(vidName)
Cap.set(cv.CAP_PROP_POS_MSEC, 5*1000)       # set starting time
nFrm    = Cap.get(cv.CAP_PROP_FRAME_COUNT)  # obtain total # of frames
```

```
#%% ===============     LOOP FRAMES       ================
cF=0
while cF < nFrm:
    cF      += 1
    r,Frm   = Cap.read()            # reads one frame
    if r==False: break

    # ----  Plot    -----
    cv.imshow('frame', Frm)
    k   = cv.waitKey(30) & 0xff
    if k==27: break                 # ESC button

Cap.release()
cv.destroyAllWindows()

#%% ===============     LOOP FRAMES to Write      ================
Cap     = cv.VideoCapture(vidName)
fourcc  = cv.VideoWriter_fourcc(*'XVID')    # specify video codec
# specify size as WIDTH/HEIGHT!! (not column/row!)
Wrt     = cv.VideoWriter('c:/ztmp/Output.avi',fourcc, 20, (640,360))

while(Cap.isOpened()):
    r, Frm  = Cap.read()
    if r==False: break

    Wrt.write(Frm.astype(np.uint8))             # write the frame

    cv.imshow('frame',Frm)
    k   = cv.waitKey(1) & 0xFF
    if k==27: break

Cap.release()
Wrt.release()
cv.destroyAllWindows()
```

P.2 Recognition 0

Simple recognition techniques, used primarily in image retrieval (Section 5).

P.2.1 Whole-Image Matching - Primitive Image Retrieval

Retrieval using pixel-by-pixel matching. Two different methods for distance measurements are tested: one method uses the Sum-of-Squared Differences, the other a Nearest-Neighbor Search, specifically a KD-Tree. The Nearest-Neighbor Search method is faster, when deployed on very large image collections; for the small set size used here, the method is in fact slower due to the overhead using the functions.

```
% Example of matching tiny images.
% We create fake images for simplicity, with a center region set to a
% constant value.
%
clear;
nImg    = 1000;
szImg   = [32 32 3];                    % size of a tiny image

%% ------   Fake Image Collection    -------
AIMG    = cell(nImg,1);
for i = 1:nImg
    I     = uint8(rand(szImg)*256);       % random RGB image
    I(14:18,14:18,:) = round(i/nImg*256);   % modify center region
    AIMG{i} = I;
end
%% ------   2 Fake Query Images   ------
Iqu1 = uint8(rand(szImg)*256); Iqu1(14:18,14:18,:) = 20;   % one fake image
Iqu2 = uint8(rand(szImg)*256); Iqu2(14:18,14:18,:) = 220;  % another one

%% ======   Transform to Matrix     =======
VIMG    = single(cat(4,AIMG{:}));               % [32 32 3 nImg]
VIMG    = reshape(VIMG,[prod(szImg) nImg])';  % [nImg 3072]
Vqu1    = single(Iqu1(:)');
Vqu2    = single(Iqu2(:)');

%% =====    Retrieval with SSD    ======
% SSD: sum-of-squared differences
tic
Df1 = sum(bsxfun(@minus, VIMG, Vqu1).^2,2);     % diff to 1st query image
Df2 = sum(bsxfun(@minus, VIMG, Vqu2).^2,2);     % diff to 2nd query image
[Df1 Rnk1]  = sort(Df1,'ascend');
[Df2 Rnk2]  = sort(Df2,'ascend');
toc

%% =====    Retrieval with NNS    =====
Mns         = KDTreeSearcher(single(VIMG)); % create KD-Tree 'model'
tic
Neb1        = knnsearch(Mns,Vqu1,'k',20);   % apply to query 1
Neb2        = knnsearch(Mns,Vqu2,'k',20);  % apply to query 2
toc

%% -----    Verify      -------
assert(all(Rnk1(1:3)==Neb1(1:3)'));     % both distances the same?

%% ------   Plot    -------
figure(1); clf; [nr nc]=deal(2,2);
subplot(nr,nc,1);
    imagesc(Iqu1); title('Query');
subplot(nr,nc,2);
    imagesc(reshape(uint8(VIMG(Rnk1(1),:)),szImg)); title('Nearest');
subplot(nr,nc,3);
    imagesc(reshape(uint8(VIMG(Rnk1(2),:)),szImg)); title('2nd Nearest');
subplot(nr,nc,4);
    imagesc(reshape(uint8(VIMG(Rnk1(end),:)),szImg)); title('Farthest');
```

```
figure(2); clf;
subplot(2,1,1);
plot(Rnk1); hold on; plot(Neb1+0.2);
title('Ranking');

subplot(2,1,2);plot(Rnk2);
```

P.2.2 Partial-Image Matching

An example for object search.

```
% Template matching using normalized cross-correlation.
% Adapted from Matlab's example on normxcorr2
clear;
Ipep    = rgb2gray(imread('peppers.png'));  % entire image
Ioni    = rgb2gray(imread('onion.png'));    % part of the image, the template

%% =====    Match   =====
tic
MC          = normxcorr2(Ioni,Ipep);   % correlation map

[yPek xPek] = find(MC==max(MC(:)));     % find peak in correlation map
xLef        = xPek-size(Ioni,2);        % x left
yTop        = yPek-size(Ioni,1);        % y top

toc

%% -----    Plot    ------
figure(1); imshowpair(Ipep,Ioni,'montage');
rectangle('position',[xLef yTop size(Ioni,2), size(Ioni,1)],'edgecolor','y');
figure(2); surf(MC); shading flat
```

P.2.3 Histogramming

Creating a color histogram for an RGB image.

```
% Histogramming for retrieval.
clear;
Irgb    = imread('peppers.png');
szI     = size(Irgb);
nPx     = szI(1)*szI(2);

%% =====    Histogram   =====
nBins   = 10;                           % number of bins
Edg     = linspace(0, 255, nBins+1);    % edges for histogram

Hred    = histcounts(Irgb(:,:,1), Edg) / nPx;   % [1 nBins]
Hgrn    = histcounts(Irgb(:,:,2), Edg) / nPx;
Hblu    = histcounts(Irgb(:,:,3), Edg) / nPx;

Ihst    = [Hred Hgrn Hblu];             % image histogram [1 nBins*3]

%% -----    Plot    ------
figure(1); [nr nc] = deal(2,2);
    Ylim = [0 0.6];
subplot(nr,nc,1);
    imagesc(Irgb);
subplot(nr,nc,2);
    bar(Hred);
    set(gca,'ylim',Ylim);
subplot(nr,nc,3);
    bar(Hgrn);
    set(gca,'ylim',Ylim);
```

```
subplot(nr,nc,4);
    bar(Hblu);
    set(gca,'ylim',Ylim);
```

P.2.4 Face Profiles

Analysis of the one-dimensional profile of a face image (Section 5.3).

```
clear;
Iorg    = imread('KlausIohannisCrop.jpg');  % image is color [m n 3]
Ig      = rgb2gray(Iorg);                   % turn into graylevel image
Ig      = Ig(35:end-35,35:end-35);          % crop borders a bit more
[h w]   = size(Ig);                         % image height and width

%% ===== Raw Profiles    ======
Pver    = sum(Ig,1);                        % vertical intensity profile
Phor    = sum(Ig,2)';                       % horizontal intensity profile

%% ===== Smoothen Profiles   =======
nPf     = round(w*0.05);                    % # points: fraction of image width
LowFlt  = pdf('norm', -nPf:nPf, 0, round(nPf/2)); % generate a Gaussian
Pverf   = conv(Pver, LowFlt, 'valid');     % filter vertical profile
Phorf   = conv(Phor, LowFlt, 'valid');      % filter horizontal profile
Pverf   = padarray(Pverf,[0 nPf],'replicate'); % extend to original size
Phorf   = padarray(Phorf,[0 nPf],'replicate'); % extend to original size

%% =====    Detect Extrema      =======
[PksVer LocPksVer] = findpeaks(Pverf);     % peaks
[TrgVer LocTrgVer] = findpeaks(-Pverf);    % troughs (sinks)
[PksHor LocPksHor] = findpeaks(Phorf);     % peaks
[TrgHor LocTrgHor] = findpeaks(-Phorf);    % troughs

%% --------     Plotting    -----------
figure(1);clf;
subplot(1,2,1);
    imagesc(Ig); colormap(gray);
subplot(2,2,2); hold on;
    plot(Pver,'k');
    plot(Pverf,'m');
    plot(LocPksVer,PksVer,'g^');
    plot(LocTrgVer,abs(TrgVer),'rv');
    set(gca,'xlim',[1 w]);
    xlabel('From Left to Right');
    title('Vertical Profile');
subplot(3,2,6); hold on;
    plot(Phor,'k');
    plot(Phorf,'m');
    plot(LocPksHor,PksHor,'g^');
    plot(LocTrgHor,abs(TrgHor),'rv');
    set(gca,'xlim',[1 h]);
    xlabel('From Top to Bottom');
    title('Horizontal Profile');
```

This would be the corresponding Python code:

```
from skimage.io    import imread
from skimage.color import rgb2gray
from scipy.signal  import convolve, get_window, argrelmax, argrelmin

#%% ----    Load Image -----
Irgb    = imread('c:/klab/do_lec/compvis/Matlab/KlausIohannisCrop.jpg')  # image is color [m n 3]
Ig      = rgb2gray(Irgb)                # turn into graylevel image
Ig      = Ig[34:-36,34:-36]             # crop borders a bit more
(h,w)   = Ig.shape                      # image height and width
```

```
#%% =====   Raw Profiles    ======
Pver    = Ig.sum(axis=0)                # vertical intensity profile
Phor    = Ig.sum(axis=1)                # horizontal#%% ---- Loading and Analysing

#%% =====   Smoothen Profiles   =======
sgm     = 10
nPf     = round(w*0.10)                     # #points: fraction of image width
LowFlt  = get_window(('gaussian',sgm),nPf)  # generate a Gaussian
LowFlt  = LowFlt / LowFlt.sum()             # normalize the filter
Pverf   = convolve(Pver, LowFlt, 'same')    # filter vertical profile
Phorf   = convolve(Phor, LowFlt, 'same')    # filter horizontal profile

#%% =====    Detect Extrema      =======
LocMaxVer    = argrelmax(Pverf)             # peaks
LocMinVer    = argrelmin(Pverf)             # troughs (sinks)
LocMaxHor    = argrelmax(Phorf)             # peaks
LocMinHor    = argrelmin(Phorf)             # troughs (sinks)

#%% ---- Plotting
from matplotlib.pyplot import figure, subplot, imshow, plot, title, xlim, ylim, cm
figure(figsize=(10,6))
subplot(1,2,1)
imshow(Ig, cmap=cm.gray)
subplot(2,2,2)
plot(Pver,'k')
plot(Pverf,'m')
plot(LocMaxVer[0],Pverf[LocMaxVer],'g^')
plot(LocMinVer[0],Pverf[LocMinVer],'rv')
xlim(1,w)
title('Vertical Profile')
subplot(2,2,4)
plot(Phor,'k')
plot(Phorf,'m')
plot(LocMaxHor[0],Phorf[LocMaxHor],'g^')
plot(LocMinHor[0],Phorf[LocMinHor],'rv')
ylim(1,h)
title('Horizontal Profile')
```

P.3 Image Processing I: Scale Space and Pyramid

Building a scale space (and pyramid) with five levels (Section 6).

```
clear;
Icol    = imread('autumn.tif');     % uint8 type; color
Ig      = single(rgb2gray(Icol));   % turn into single type

%% ------   Initialize
nLev    = 5;
[SS PY aFlt] = deal(cell(nLev,1));
SS{1}   = Ig;         % scale space: make original image first level
PY{1}   = Ig;         % pyramid:      "      "      "     "     "

%% =====    Scale Space and Pyramid
for i = 1:nLev-1
   Flt     = fspecial('gaussian', [2 2]+i*3, i);   % 2D Gaussian
    aFlt{i} = Flt;                                  % store for plotting
    Ilpf    = conv2(Ig, Flt, 'same');      % low-pass filtered image
    SS{i+1} = Ilpf;
    % --- Downsampling with stp
    stp     = 2^i;
    Idwn    = downsample(Ilpf,stp);      % first along rows
    PY{i+1} = downsample(Idwn',stp)';       % then  along colums
end

%% -----    Plotting
figure(1);clf;
[nr nc] = deal(nLev,3);
for i = 1:nLev
    if i<nLev,
        subplot(nr,nc,i*nc-2);
            imagesc(aFlt{i});
    end
    subplot(nr,nc,i*nc-1);
        imagesc(SS{i});
    subplot(nr,nc,i*nc);
        imagesc(PY{i});
end
```

For the Python code, we also wrote a function `fspecialGauss`, which mimics Matlab's function `fspecial`.

```
from numpy import mgrid, exp
from skimage       import data
from skimage.color import rgb2gray
from skimage.transform import resize
from scipy.signal  import convolve2d

def fspecialGauss(size,sigma):
    x, y = mgrid[-size//2 + 1:size//2 + 1, -size//2 + 1:size//2 + 1]
    g   = exp(-((x**2 + y**2)/(2.0*sigma**2)))
    return g/g.sum()

#%% ------   Load & Transform   --------
Irgb    = data.chelsea()
Ig      = rgb2gray(Irgb)    # turn into single type
(m,n)   = Ig.shape

#%% ------   Initialize     ---------
nLev    = 5
SS = {}; PY={}; aFlt={};
SS[0]   = Ig.copy()   # scale space: make original image first level
PY[0]   = Ig.copy()   # pyramid:      "      "      "     "     "

#%% =====    Scale Space and Pyramid    ========
for i in range(1,nLev):
    Flt     = fspecialGauss(2+i*3, i)           # 2D Gaussian
    aFlt[i-1] = Flt                             # store for plotting
```

```
    Ilpf    = convolve2d(Ig, Flt, mode='same')  # low-pass filtered image
    SS[i]   = Ilpf
    # --- Downsampling with stp
    stp     = 2**i
    PY[i]   = resize(Ilpf,(m//stp,n//stp))

#%% -----    Plotting
from matplotlib.pyplot import *
figure()
(nr,nc) = (nLev,3)
for i in range(0,nLev):
    if i<nLev-1:
        subplot(nr,nc,i*nc+1)
        imshow(aFlt[i])
    subplot(nr,nc,i*nc+2)
    imshow(SS[i])
    subplot(nr,nc,i*nc+3)
    imshow(PY[i])
```

P.4 Feature Extraction I

P.4.1 Regions

Obtaining regions using the Difference-Of-Gaussian (DOG) method (Section 7.1).

```
clear;
Icol    = imread('tissue.png');
I       = single(rgb2gray(Icol));

%% ======   Diff-of-Gaussians (DOG)
Fs1     = fspecial('Gaussian',[11 11], 3);  % fine   low-pass filter
Fs2     = fspecial('Gaussian',[21 21], 6);  % coarse low-pass filter

Is1     = conv2(I, Fs1, 'same');            % convolution with image
Is2     = conv2(I, Fs2, 'same');

Idog    = Is2 - Is1;                        % diff-of-Gaussians
BWblobs = Idog > 15;                        % thresholding for blobs

%% ======   Laplacian-of-Gaussian (LoG)
Flog    = fspecial('log',[15 15],5);
Ilog    = conv2(I, Flog, 'same');
BWlog   = Ilog > .6;

%% -----    Plotting
figure(1);clf;
[nr nc] = deal(3,2);
subplot(nr,nc,1),    imagesc(Icol);
subplot(nr,nc,2),    imagesc(I);
subplot(nr,nc,3),    imagesc(Idog);colorbar;
subplot(nr,nc,4),    imagesc(BWblobs);
subplot(nr,nc,5),    imagesc(Ilog);colorbar;
subplot(nr,nc,6),    imagesc(BWlog);
```

For the Python example we assume that the 2D Gaussian filter - as used in the above example P.3 - is placed into a separate module:

```
from skimage       import data
from skimage.color import rgb2gray
from fspecialGauss import *
from scipy.signal  import convolve2d
from matplotlib.pyplot import *

#%% ------  The image   -------
Icol    = data.immunohistochemistry()
I       = rgb2gray(Icol);

#%% ======   Diff-of-Gaussians (DOG) =======
Fs1     = fspecialGauss(11, 3)              # fine   low-pass filter
Fs2     = fspecialGauss(21, 6)              # coarse low-pass filter

Is1     = convolve2d(I, Fs1, 'same')        # convolution with image
Is2     = convolve2d(I, Fs2, 'same')

Idog    = Is2 - Is1                         # diff-of-Gaussians
BWblobs = Idog > 0.02                       # thresholding for blobs

#%% -----    Plotting       -------
figure(figsize=(15,15))
nr, nc = 3,2
subplot(nr,nc,1),    imshow(Icol)
subplot(nr,nc,2),    imshow(I)
subplot(nr,nc,3),    imshow(Idog);colorbar
subplot(nr,nc,4),    imshow(BWblobs)
```

P.4.2 Edge Detection

As introduced in Section 7.2. In addition to the demonstration of edge detection, we also show how to thin the black-white map (last few lines), an action that is useful for contour tracing (coming up in a later section).

```
clear; format compact;
sgm     = 1;            % scale, typically 1-5

%% -----    Load an Image   -----
I       = double(imread('cameraman.tif'));

%% =====    Blurring    ======
Nb      = [2 2]+sgm;
Fsc     = fspecial('gaussian', Nb, sgm);    % 2D gaussian
Iblr    = conv2(I, Fsc, 'same');

%% =====    Edge Detection  ======
BWrob   = edge(Iblr, 'roberts');
BWsob   = edge(Iblr, 'sobel');
BWpwt   = edge(Iblr, 'prewitt');            % similar to Sobel
BWlog   = edge(Iblr, 'log', [], sgm);       % laplacian of Gaussian
BWzex   = edge(Iblr, 'zerocross');
% For Canny we apply the original image I as it performs the blurring
BWcny   = edge(I, 'canny', [], sgm);

%% -------  Plotting    -------
figure(1); clf; colormap(gray); [nr nc] = deal(3,2);
subplot(nr,nc,1); imagesc(BWrob); title('Roberts');
subplot(nr,nc,2); imagesc(BWsob); title('Sobel');
subplot(nr,nc,3); imagesc(BWpwt); title('Prewitt');
subplot(nr,nc,4); imagesc(BWlog); title('Log');
subplot(nr,nc,5); imagesc(BWcny); title('Canny');
subplot(nr,nc,6); imagesc(BWlog); title('Zero-Cross');

%% -------  Cleaning Edge Map for Contour Tracing   -------
Medg = BWcny;
Medg = bwmorph(Medg,'clean');   % removes isoloated pixels
Medg = bwmorph(Medg,'thin');    % turns 'thick' contours into 1-pixel-wide contours
```

```
from skimage           import data
from skimage.feature   import canny
from skimage.filters   import roberts, sobel, prewitt
from skimage.morphology import remove_small_objects, thin

#%% -----   Load    ------
I       = data.camera()          # uint8

#%% -----   One-scale edge detection    -------
Mrob    = roberts(I)
Msob    = sobel(I)
Mpwt    = prewitt(I)

#%% -----   Canny for two scales   ------
ME1     = canny(I, sigma=1)
ME2     = canny(I, sigma=3)

#%% -------  Plotting   -------
from matplotlib.pyplot import *
figure(figsize=(12,10)); nr,nc = 2,3;
subplot(nr,nc,1); imshow(I); title('Input Image')
subplot(nr,nc,2); imshow(Mrob); title('Roberts')
subplot(nr,nc,3); imshow(Msob); title('Sobel')
subplot(nr,nc,4); imshow(Mpwt); title('Prewitt')
subplot(nr,nc,5); imshow(ME1); title('Canny $\sigma$=1')
subplot(nr,nc,6); imshow(ME2); title('Canny $\sigma$=3')
```

```
#%% -------  Cleaning Edge Map for Contour Tracing   -------
Medg = ME1.copy()               # we copy for reason of clarity
remove_small_objects(Medg,2,in_place=True)  # removes isoloated pixels
Medg = thin(Medg)                           # turns 'thick' contours into 1-pixel-wide contours
```

```
import cv2
# ---- Stimulus
fp      = 'C:/xxx/xxx.tif'
Igry    = cv2.imread(fp, cv2.IMREAD_GRAYSCALE) # gray-scale

#%% ======    detect edge pixels  =======
lowThr  =  50       # low threshold
hihThr  = 200       # high threshold
BWcan  = cv2.Canny(Igry, lowThr, hihThr)
```

P.4.3 Texture Filters

```
% A filter bank for texture. (The Leung-Malik filter bank)
% see also: http://www.robots.ox.ac.uk/~vgg/research/texclass/filters.html
%
function F = f_GenTxtFilt(figNo)
sz          = 49;               % filter size
SclBlb      = sqrt(2).^(1:4);   % sigma for blob filters
SclOri      = sqrt(2).^(1:3);   % sigma for oriented filters
nSclBlb     = length(SclBlb);
nSclOri     = length(SclOri);
nBlb        = 12;               % # of blob filters (first 12 filters)
nOri        = 6;             % # of orientations
ntFlt       = nSclOri*nOri*2 + nBlb;   % # of total filters

%% -----    Init Memory & Points
F           = zeros(sz,sz,ntFlt);
rd          = (sz-1)/2;
[X Y]       = meshgrid(-rd:rd, rd:-1:-rd);
PTS         = [X(:) Y(:)]';

%% -----    Blob Filters (first 12 filters)
for i = 1:nSclBlb
    F(:,:,i)   = ff_Norm(fspecial('gaussian', sz, SclBlb(i)));
    F(:,:,4+i) = ff_Norm(fspecial('log',      sz, SclBlb(i)));
    F(:,:,8+i) = ff_Norm(fspecial('log',      sz, SclBlb(i)*4));
end

%% -----    Edge & Bar Filters (filters 13-48)
cc          = 1;
for s = 1:nSclOri,
    for o  = 0:nOri-1,
        ang         = pi*o/nOri;  % Not 2pi as filters have symmetry
        ca          = cos(ang);
        sa          = sin(ang);
        PTSrot     = [ca -sa; sa ca] * PTS;
        F(:,:,12+cc) = ff_Edg(PTSrot, SclOri(s), sz); % edge
        F(:,:,30+cc) = ff_Bar(PTSrot, SclOri(s), sz); % bar
        cc          = cc+1;
    end
end

%% -----    Plotting
figure(figNo); clf; colormap(gray);
for i = 1:ntFlt
    subplot(8,6,i)
        I = F(:,:,i);
        imagesc(I);
        set(gca,'fontsize',4);
        title(num2str(i),'fontsize',6,'fontweight','bold');
end

end % MAIN

%% ==============   SUB FUNCTIONS   ============
function F = ff_Bar(PTS, sgm, sz)
Gx      = normpdf(PTS(1,:), 0, sgm*3);
Gy      = normpdf(PTS(2,:), 0, sgm);
vnc     = sgm^2;
Gy      = Gy.*((PTS(2,:).^2-vnc)/(vnc^2));      % 2nd derivative
F       = ff_Norm(reshape(Gx.*Gy, sz, sz));
end % SUB

function F = ff_Edg(PTS, sgm, sz)
Gx      = normpdf(PTS(1,:), 0, sgm*3);
Gy      = normpdf(PTS(2,:), 0, sgm);
Gy      = -Gy.*(PTS(2,:)/(sgm^2));              % 1st derivative
F       = ff_Norm(reshape(Gx.*Gy, sz, sz));
end % SUB

function D = ff_Norm(D)
D       = D - mean(D(:));
D       = D / sum(abs(D(:)));

end % SUB
```

P.5 Convolutional Neural Networks

P.5.1 MNIST dataset

A network for recognizing handwritten digits, i.e. the MNIST database with 28 x 28 pixel images. Explanations in 8.1 and F.1. The code is almost the same as in F.1, except that the network architecture is preceeded by convolutional and subsampling operations, and that we need to reshape the tensors entering the dense layers slightly differently.

```
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
import torchvision.transforms as transforms

pthDatBase  = 'c:/IMGdown'      # path of the MNIST database on your PC

# =====     Learning Parameters    =====
nEpoch      = 5         # number of epochs
szBtch      = 100       # batch size (number of training samples per epoch)
lrnRate     = 0.001     # learning rate

# ======    Select Device (GPU | CPU)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

#%% %%%%%%%%%%%%%      PREPARE DATA     %%%%%%%%%%%%%%%%%%%
TrfImg  = transforms.ToTensor()         # uint8 [0,255] > torch float [0,1]

DATtren = MNIST(pthDatBase, train=True,  transform=TrfImg, download=False)
DATtest = MNIST(pthDatBase, train=False, transform=TrfImg)

LOADtren = DataLoader(DATtren, batch_size=szBtch, shuffle=True)
LOADtest = DataLoader(DATtest, batch_size=szBtch, shuffle=False)

#%% %%%%%%%%%%%%%      BUILD NETWORK    %%%%%%%%%%%%%%%%%%%
class NN_MLPconv(nn.Module):
    def __init__(self):
        super(NN_MLPconv, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)  # grayscale, out 20, szKrn, stride
        self.conv2 = nn.Conv2d(20, 50, 5, 1) # in 20, out 50, szKrn, stride
        self.fc1   = nn.Linear(4*4*50, 500)  # [(map size * #layers) 500]
        self.fc2   = nn.Linear(500, 10)      # [500 nClasses]

    def forward(self, I):           # [28 28]   image size
        I = F.relu(self.conv1(I))   # [24 24]   map size after 1st convolution
        I = F.max_pool2d(I, 2, 2)   # [12 12]   map size after 1st pooling
        I = F.relu(self.conv2(I))   # [8 8]     map size after 2nd convolution
        I = F.max_pool2d(I, 2, 2)   # [4 4]     map size after 2nd pooling
        I = I.view(-1, 4*4*50)      # reshape to linear for dense layers
        I = F.relu(self.fc1(I))
        I = self.fc2(I)
        return F.log_softmax(I, dim=1)

NET         = NN_MLPconv().to(device)  # move to CUDA (or stay on CPU)

#%% %%%%%%%%%%%%%      OPTIMIZATION    %%%%%%%%%%%%%%%%%%%
criterion   = nn.CrossEntropyLoss()
optimizer   = torch.optim.Adam(NET.parameters(), lr=lrnRate)

#%% %%%%%%%%%%%%%      LEARNING     %%%%%%%%%%%%%%%%%%%
nImgTren    = DATtren.data.shape[0]
NET.train()
# =====     LOOP EPOCHS    ======
for e in range(nEpoch):
    # =====     LOOP BATCHES    ======
    lossInt=0
```

```
    for i, (IMG, Lbl) in enumerate(LOADtren):
        IMG     = IMG.to(device)        # move to CUDA (or stay on CPU)
        Lbl     = Lbl.to(device)
        # ====  forward  ======
        optimizer.zero_grad()           # zero the gradient buffer
        Out     = NET(IMG)              # forward
        loss    = criterion(Out, Lbl)   # error calculation
        # ====  backward  ======
        loss.backward()                 # error finding
        optimizer.step()                # error correction
        lossInt += loss.data.item()     # accumulate error value

    print ('epoch %d, loss: %.4f' % (e+1, lossInt/nImgTren))

#%% %%%%%%%%%%%%      EVALUATION    %%%%%%%%%%%%%%%%%%%
NET.eval()
cHit = 0; cImg = 0
for IMG, Lbl in LOADtest:
    IMG         = IMG.to(device)
    Out         = NET(IMG)
    _, Pred     = torch.max(Out.data, 1)        # max across class scores
    cImg        += Lbl.size(0)
    cHit        += (Pred.cpu() == Lbl).sum()    # number of hits

assert cImg==LOADtest.dataset.data.shape[0], 'not same count' # verify
print('accuracy: %1.2f' % (100 * cHit.numpy() / cImg))  # 99.23

#%% %%%%%%%%%%%%      OBTAIN HIDDEN LAYER   %%%%%%%%%%%%%%%%%%%
hidImg = [None]
def hookHid(module, input, output):
    hidImg[0] = output

NET.fc1.register_forward_hook(hookHid)          # instruct to store

for i in range(0,3):
    Img         = DATtest.data[i,:,:].float()   # from uint8 to float
    out         = NET(Img.unsqueeze(0).unsqueeze(0))
    print(hidImg[0].detach().numpy())
```

P.5.2 Transfer Learning [PyTorch]

Section 8.2

Below are three code blocks. The first code block trains the model. The second code block saves the model - it is a single line. The third code block demonstrates how to apply the model on a single (novel) image.

Block 1 The string variable `dirImgs` is the directory to your database, with two folders: 'train' and 'valid' for training and validation. In each one of those two folders, there is the exact same list of folder names representing the category labels. For those in turn, we provide different images. For small data sets place two thirds of the images into the corresponding training folder (ie. `train/dog/`) and one third into the corresponding validation folder (`valid/dog/`). To obtain optimal results, one would calculate the (normalized) mean and standard devation of the entire dataset - here we take the values of the ImageNet, for which the network was trained for. Using the boolean variable `bFine` we can switch between the two modes of transfer learning, fixed-feature extraction (`False`) and fine tuning (`True`).

```
"""
Transfer Learning with switch between fixed-feature extraction and fine-tuning
    > fixed-feature extraction: re-learns only last layer
    > fine-tuning: all weights are adjusted
"""
import numpy as np
import torch
from torchvision import models, transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
from torch.optim import SGD, lr_scheduler
import torch.nn as nn

import time
import copy

dirImgs     = 'C:/IMGdown/OLTOtv/'
bFine       = True     # True: fine-tuning; False: fixed-feature extraction
# computed mean values for all images of entire database
ImgMean     = [0.485, 0.456, 0.406]  # mean    of ImageNet set
ImgStdv     = [0.229, 0.224, 0.225]  # std dev of ImageNet set
szImgTarg   = 224
#%% ================    Parameters      ================
nEpo        = 20        # number of epochs (learning steps)
szBtch      = 4         # number of images per batch
lernRate    = 0.001     # learning rate
momFact     = 0.9       # momentum factor
szStep      = 7         # step size [in epochs]
gam         = 0.1       # gamma (decay rate)

# ======    Select Device (GPU | CPU)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

#%% %%%%%%%%%%%%%      PREPARE DATA     %%%%%%%%%%%%%%%%%%%
TrfImgTrain   = transforms.Compose([
        transforms.RandomResizedCrop(szImgTarg),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(ImgMean, ImgStdv)  ])
TrfImgValid   = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(szImgTarg),
        transforms.ToTensor(),
        transforms.Normalize(ImgMean, ImgStdv)   ])
# ----------------    Folder and Loader   ---------------
FOLDtrain   = ImageFolder(dirImgs+'train', TrfImgTrain)
FOLDvalid   = ImageFolder(dirImgs+'valid', TrfImgValid)
LOADtrain   = DataLoader(FOLDtrain, batch_size=szBtch, shuffle=True)
LOADvalid   = DataLoader(FOLDvalid, batch_size=szBtch, shuffle=True)
```

```
nImgTrain   = len(FOLDtrain)        # total number of images for training set
nImgValid   = len(FOLDvalid)        # total number of imges for validation set
aLbClass    = FOLDtrain.classes    # class labels
nClss       = len(aLbClass)

#%% %%%%%%%%%%%%%      PREPARE NETWORK    %%%%%%%%%%%%%%%%%%%
MOD         = models.resnet18(pretrained=True)
# freezing layers until last one (to avoid backward computations)
if not bFine:
    for param in MOD.parameters():
        param.requires_grad = False
# replaces last fully-connected layer with new random weights:
MOD.fc  = nn.Linear(MOD.fc.in_features, nClss)

MOD     = MOD.to(device)    # needs to occur BEFORE selection of optimization algorithms

#%% %%%%%%%%%%%%%      OPTIMIZATION    %%%%%%%%%%%%%%%%%%%
crit    = nn.CrossEntropyLoss()
# re-learning process: full re-learning or only last layer is re-learned
if bFine: optim = SGD(MOD.parameters(),    lr=lernRate, momentum=momFact)
else:     optim = SGD(MOD.fc.parameters(), lr=lernRate, momentum=momFact)
# decreases learning rate every szStep epochs by a factor of gamma
sched   = lr_scheduler.StepLR(optim, step_size=szStep, gamma=gam)

#%% #################        TRAIN AND EVALUATE      ###################
t0 = time.time()
WgtsBest = copy.deepcopy(MOD.state_dict())
accuBest = 0.0
PrfVal, PrfTrn   = np.zeros((nEpo,2)), np.zeros((nEpo,2))
# =====     LOOP EPOCHS    ======
for i in range(nEpo):
    print(f'---- epoch {i}/{nEpo} -----')
    # ================    TRAIN    ================
    sched.step()
    MOD.train()                             # set model to training mode
    # =====     LOOP BATCHES    ======
    lossRun = 0.0
    cCrrRun = 0
    for ImBtc, LbBtc in LOADtrain:
        ImBtc = ImBtc.to(device)            # [nImgBtch 3RGB height width]
        LbBtc = LbBtc.to(device)            # [nImgBtch]
        with torch.set_grad_enabled(True):
            # ====  forward  ======
            Post        = MOD(ImBtc)        # posteriors [nImgBtch nClasses]
            _,LbPred    = Post.max(1)       # predicted labels
            loss        = crit(Post, LbBtc)
            # ====  backward  ======
            optim.zero_grad()               # zero the parameter gradients
            loss.backward()
            optim.step()
        # statistics
        lossRun += loss.item() * szBtch
        cCrrRun += torch.sum(LbPred==LbBtc.data)
    # --- performance per epoch
    lossEpo = lossRun / nImgTrain
    accuEpo = cCrrRun.double() / nImgTrain
    print(f'train  loss: {lossEpo:.4f}  acc: {accuEpo:.4f}')
    PrfTrn[i,:] = [accuEpo, lossEpo]        # record performance

    # ================    VALIDATE    ================
    MOD.eval()                              # set model to evaluate mode
    # =====     LOOP BATCHES    ======
    lossRun = 0.0
    cCrrRun = 0
    for ImBtc, LbBtc in LOADvalid:
        ImBtc = ImBtc.to(device)            # [nImgBtch 3RGB height width]
        LbBtc = LbBtc.to(device)            # [nImgBtch]
```

```
        with torch.set_grad_enabled(False):
            # ====  forward  ======
            #set_trace()
            Post        = MOD(ImBtc)        # posteriors [nImgBtch nClasses]
            _,LbPred    = Post.max(1)       # predicted labels
            loss        = crit(Post, LbBtc)
        # statistics
        lossRun += loss.item() * szBtch
        cCrrRun += torch.sum(LbPred==LbBtc.data)
    # --- performance per epoch
    lossEpo = lossRun / nImgValid
    accuEpo = cCrrRun.double() / nImgValid
    print(f'valid  loss: {lossEpo:.4f}  acc: {accuEpo:.4f}')
    PrfVal[i,:] = [accuEpo, lossEpo]        # record performance
    # keep weights if better than previous weights
    if accuEpo>accuBest:
        accuBest    = accuEpo
        WgtsBest    = copy.deepcopy(MOD.state_dict())

# ----  Concluding
tElaps = time.time() - t0
print('Training duration {:.0f}min {:.0f}sec'.format(tElaps // 60, tElaps % 60))
print(f'Max accuracy {accuBest:4f}')

MOD.load_state_dict(WgtsBest)   # loads best performing weights for application

#%% -----   Plot Learning Curves    -------
import matplotlib.pyplot as plt
plt.figure
plt.plot(PrfVal[:,0],'b')
plt.plot(PrfTrn[:,0],'b:')
plt.plot(PrfVal[:,1],'r')
plt.plot(PrfTrn[:,1],'r:')
plt.gca().legend(('valid acc', 'train acc', 'valid err', 'train err'))
plt.show()
```

Block 2 You can save your model as follows:

```
torch.save(MOD.state_dict(), pathVariable)
```

Block 3 Now we reload the model and test a single image:

```
import torch
from torchvision import models, transforms
import torch.nn as nn
from PIL import Image

pathToYourModel  = 'path and name of your model'
imgPath  = 'path and name of your testing image'

ImgMean     = [0.50, 0.50, 0.50]    # we assume them to be 0.5
ImgStdv     = [0.22, 0.22, 0.22]    # we assume them to be 0.22
szImgTarg   = 224

#%% --- Prepare the Model
MOD         = models.resnet18()
MOD.fc      = nn.Linear(MOD.fc.in_features, nClasses)
# --- Now load the weights
MOD.load_state_dict(torch.load(pathToYourModel))
MOD.eval()

f_PrepImg  = transforms.Compose([
        transforms.Resize((szImgTarg,szImgTarg)),
```

```
        transforms.ToTensor(),
        transforms.Normalize(ImgMean, ImgStdv)  ])

#%% --- Load Image and Apply
Irgb        = Image.open(imgPath)   # open as PIL image (not numpy!)
Irgb        = f_PrepImg(Irgb)     # image preparation
# make it 4-dimensional by adding 1 dimension as axis=0
Irgb        = Irgb.unsqueeze(0)
# now we feed it to the network:
Post        = MOD(Irgb)         # posteriors [1 nClasses]
psT,lbT    = Post.max(1)       # posterior and predicted label (still as tensors!)
lbPred      = lbT.item()     # label as a single scalar value (now as scalar in regular python)
```

P.6 Segmentation (Top-Down)

Segmentation using auto-encoder networks (Section 9), see also Section F.3 for a simple auto-encoder example.

P.6.1 One-Label Example

Two sets of images are required: the gray-scale images in directory 'train' and the corresponding label images with the same filenames in directory 'train_masks'.

```
import os
from os.path import isfile, join
from PIL import Image
import torch
import torchvision.models.segmentation as ModSeg
import torch.optim as optim
from torch import nn
from torch.utils.data.dataset import Dataset
from torch.utils.data import DataLoader
from torchvision import transforms
import numpy as np
import matplotlib.pyplot as plt; plt.show(block=False)

pthData     = 'c:/IMGdown/CARVANA/'

#%% ================    Parameters      ================
nEpo        = 20        # number of epochs
szBtc       = 8         # batch size
lrnRate     = 1e-4      # learning rate
nWorkers    = 0         # 0 for simplicity

# ======    Select Device (GPU | CPU)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

#%% %%%%%%%%%%%%%      PREPARE DATA     %%%%%%%%%%%%%%%%%%%
def f_ImgPaths(path):   # obtain full paths for images
    L = [join(path, f) for f in os.listdir(path) if isfile(join(path, f))]
    L.sort()
    return L

class f_DATtren(Dataset):
    def __init__(self, dirSub, transform=None):
        self.transform  = transform
        self.pathImg = f_ImgPaths(pthData + dirSub)
        self.pathMsk = f_ImgPaths(pthData + dirSub + "_masks")

    def __getitem__(self, ix):
        I = Image.open(self.pathImg[ix])
        T = Image.open(self.pathMsk[ix])
        I = self.transform(I)
        T = self.transform(T)
        return I, T

    def __len__(self):
        return len(self.pathImg)

#%% -----   Init DATA  -------
TrfImg   = transforms.Compose([transforms.Resize((256,256)),
                               transforms.ToTensor()]  )
DATtren  = f_DATtren(dirSub='train', transform=TrfImg)

LOADtren = DataLoader(dataset      = DATtren,
                      batch_size   = szBtc,
                      shuffle      = True,
                      pin_memory   = True,
                      num_workers  = nWorkers)
```

```
#%% %%%%%%%%%%%%%      NETWORK    %%%%%%%%%%%%%%%%%%%
NET     = ModSeg.fcn_resnet50(num_classes=1, pretrained=False).to(device)

#%% %%%%%%%%%%%%%      OPTIMIZER     %%%%%%%%%%%%%%%%%%%
class BCELoss2d(nn.Module): # Binary Cross Entropy loss function
    """
    https://www.kaggle.com/c/carvana-image-masking-challenge/discussion/37208
    """
    def __init__(self, weight=None, reduction='mean'):
        super(BCELoss2d, self).__init__()
        self.bce_loss = nn.BCELoss(weight, reduction)

    def forward(self, IOut, Msks):
        Iprob       = torch.sigmoid(IOut)
        Iprob_flat  = Iprob.view(-1)
        Msks_flat   = Msks.view(-1)
        return self.bce_loss(Iprob_flat, Msks_flat)

crit    = BCELoss2d().to(device)
optim   = optim.Adam(NET.parameters(), lr=lrnRate)

#%% -----   Plot    -------
Tens2PIL = transforms.ToPILImage()
def p_ImgEx(aImg):
    if len(aImg) > 9: raise Exception("9 or fewers images allowed")
    for i, I in enumerate(aImg):
        I = np.array(Tens2PIL(I))
        plt.subplot(100 + 10 * len(aImg) + (i+1))
        fig = plt.imshow(I)
        fig.axes.get_xaxis().set_visible(False)
        fig.axes.get_yaxis().set_visible(False)
    plt.show(); plt.pause(.1)

# some example pairs
plt.figure(1)
aImg = []
for i in range(4):
    I, M = DATtren[i]
    aImg.append(I)
    aImg.append(M)
p_ImgEx(aImg)

#%% %%%%%%%%%%%%%      LEARNING     %%%%%%%%%%%%%%%%%%%
for e in range(nEpo):

    NET.train()
    # SSSSSSSSSSSS     LOOP IMAGE/MASK PAIRS   SSSSSSSSSSSSSS
    for i, (Imgs, Msks) in enumerate(LOADtren):
        # batch of images [szBtc 3 h w]
        Imgs    = Imgs.to(device)
        Msks    = Msks.to(device)
        # ==========    forward      ===============
        optim.zero_grad()
        IOut    = NET(Imgs)             # IOut is [szBtc 1 h w]
        loss    = crit(IOut['out'], Msks)      # compare output with target
        # ==========    backward     ===============
        loss.backward()
        optim.step()
        print('epo %d/ %d - loss: %.4f ' % (e + 1, nEpo, loss.data.item()))

    # -----     plot    ------
    I = np.array(Tens2PIL(IOut[0]))     # first example of batch
    plt.figure(2);
    plt.imshow(I); plt.pause(.1)

#%% %%%%%%%%%%%%%      EVALUATING     %%%%%%%%%%%%%%%%%%%
class f_DATtest(Dataset):   # same function as f_DATtren but not loading masks
```

```
    def __init__(self, dirSub, transform=None):
        self.transform  = transform
        self.pathImg = f_ImgPaths(pthData + dirSub)
    def __getitem__(self, ix):
        I = Image.open(self.pathImg[ix])
        I = self.transform(I)
        return I
    def __len__(self):
        return len(self.pathImg)
#%% -----   Init DATA  -------
DATtest  = f_DATtest(dirSub='test', transform=TrfImg)
LOADtest = DataLoader(DATtest, batch_size=1, num_workers=nWorkers, shuffle=False)
NET.eval()  # switch to evaluation mode
#%% ----    Run 1 image     -----
Img     = LOADtest.dataset[0]   # one image
#Img,Msk = LOADtren.dataset[0]  # testing an image from training
Iout    = NET(Img.unsqueeze(0).to(device))  # add 4th dimension to make it a 4-dim tensor
Iseg    = Iout['out'].data[0].squeeze(0).cpu().numpy()
BWseg   = Iseg>Iseg.mean()
plt.figure(3)
plt.subplot(1,2,1); plt.imshow(Iseg)
plt.subplot(1,2,2); plt.imshow(BWseg)
```

P.6.2 Multi-Label Example

```
import torch
import torchvision.models.segmentation as ModSeg
import torch.optim as optim
from torch import nn
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import Cityscapes
import numpy as np
import matplotlib.pyplot as plt; plt.show(block=False)

pthImg     = 'e:/IMGdown/cityscapes_data/'

#%% ================    Parameters      ================
nEpo        = 20        # number of epochs
szBtc       = 4         # batch size
lrnRate     = 1e-4      # learning rate
nWorkers    = 0         # 0 for simplicity
nLab        = 34;
sidImg      = 256
# ======    Select Device (GPU | CPU)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

#%% %%%%%%%%%%%%%      PREPARE DATA     %%%%%%%%%%%%%%%%%%%
TrfImg   = transforms.Compose([transforms.Resize((sidImg,sidImg)),
                               transforms.ToTensor()]  )
IMS = Cityscapes(pthImg, split='train', mode='fine', target_type='instance',
                 transform=TrfImg, target_transform=TrfImg)
IML = DataLoader(IMS, batch_size=szBtc, shuffle=True, pin_memory=True,
                      num_workers=nWorkers)

#%% %%%%%%%%%%%%%      NETWORK    %%%%%%%%%%%%%%%%%%%
NET     = ModSeg.fcn_resnet50(num_classes=nLab, pretrained=False).to(device)

#%% %%%%%%%%%%%%%      OPTIMIZER     %%%%%%%%%%%%%%%%%%%
class BCELoss2d(nn.Module): # Binary Cross Entropy loss function
    """
    https://www.kaggle.com/c/carvana-image-masking-challenge/discussion/37208
    """
    def __init__(self, weight=None, reduction='mean'):
```

```
        super(BCELoss2d, self).__init__()
        self.bce_loss = nn.BCELoss(weight, reduction)

    def forward(self, IOut, Msks):
        Iprob       = torch.sigmoid(IOut)
        Iprob_flat  = Iprob.view(-1)
        Msks_flat   = Msks.view(-1)
        return self.bce_loss(Iprob_flat, Msks_flat)

crit    = BCELoss2d().to(device)
optim   = optim.Adam(NET.parameters(), lr=lrnRate)

#%% %%%%%%%%%%%%%      LEARNING     %%%%%%%%%%%%%%%%%%%
for e in range(nEpo):

    NET.train()
    # SSSSSSSSSSSS     LOOP IMAGE/MASK PAIRS   SSSSSSSSSSSSSS
    for i, (Imgs, Msks) in enumerate(IML):
        Imgs    = Imgs.to(device)   # batch of images [szBtc 3 h w]
        Msks    = Msks.to(device)   # batch of masks  [szBtc 1 h w]
        szBtc   = Imgs.shape[0]
        # ====  creating binary masks   =====
        MskB    = torch.zeros((szBtc,nLab,sidImg,sidImg), dtype=torch.float)
        for b in range(szBtc):   # --- loop batches
            Ib  = Msks[b,:,:,:].squeeze(0) # list of masks for this image
            for l in range(nLab):   # --- loop labels
                Ip      = torch.zeros((sidImg,sidImg))
                BW      = Ib==l+1       # black-white with 1 where label present
                Ip[BW]  = 1             # place into image
                MskB[b,l,:,:] = Ip      # place into mask array
        # ==========    forward      ===============
        optim.zero_grad()
        IOut    = NET(Imgs)             # IOut is a dictionary with key='out'
        Imaps   = IOut['out']           # [szBtc nLabels h w]
        loss    = crit(Imaps, MskB.to(device))
        # ==========    backward     ===============
        loss.backward()
        optim.step()
        print('epo %d/ %d - loss: %.4f ' % (e + 1, nEpo, loss.data.item()))

#%% ----    Save Model     -----
torch.save(NET.state_dict(), pthImg + 'ModLearned')

#%% %%%%%%%%%%%%%      EVALUATING     %%%%%%%%%%%%%%%%%%%
NET.eval()  # switch to evaluation mode
# activate testing set
IMS = Cityscapes(pthImg, split='test', mode='fine', target_type='instance',
                 transform=TrfImg, target_transform=TrfImg)

#%% ----    Run 1 image     -----
Img, Smnt = IMS[0]
Iout    = NET(Img.unsqueeze(0).to(device))  # add 4th dimension to make it a 4-dim tensor
Iseg    = Iout['out'].data[0].squeeze(0).cpu().numpy()
Tens2PIL = transforms.ToPILImage()
plt.figure(3)
plt.subplot(1,2,1); plt.imshow(Tens2PIL(Img))
plt.subplot(1,2,2); plt.imshow(Iseg[6,:,:]>0)
```

P.7 Object Detection

P.7.1 Face Detection

Face and eye detection using the venerable Viola-Jones algorithm, an algorithm that uses so-called Haar features (Section 10.1). First an example of how to use it in Matlab, then one in OpenCV.

```
%% ----     Load Image  ----
imPth   = 'C:\zFotos\PixRomania\LAPIonTerasa\DSC_0791_red.JPG';
Irgb    = imread(imPth);
Igry    = rgb2gray(Irgb);

%% ====     Initialize Detector    ====
FaceDet = vision.CascadeObjectDetector('FrontalFaceCART');

%% ====     Apply Detector    ====
BBox    = step(FaceDet,Irgb);  % bounding boxes [nBox 4]

%% ------   Plot Entire Image    ------
Iann    = insertObjectAnnotation(Irgb, 'rectangle', BBox, 'Face');
figure(1); clf;
imshow(Iann);

%% ====     Extract Faces   =====
figure(2); clf;
nBox    = size(BBox,1);
nr      = ceil(nBox/2);
for i = 1:nBox
    FacePatch    = imcrop(Irgb,BBox(i,:));
    subplot(nr,2,i);
    imagesc(FacePatch);
end
```

An example using OpenCV through Python:

```
import cv2

imPth   = 'pathToSomeImage.JPG'
cascPth = 'C:/SOFTWARE/opencv/build/etc/haarcascades/'
FaceDet = cv2.CascadeClassifier(cascPth+'haarcascade_frontalface_default.xml')
EyeDet  = cv2.CascadeClassifier(cascPth+'haarcascade_eye.xml')

#%% ----    Load Image   ------
Ibgr    = cv2.imread(imPth)
Igry    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2GRAY)

#%% ====    Face Detection  ======
aFaces  = FaceDet.detectMultiScale(Igry, 1.3, 5)
aEyes   = []
for (x,y,w,h) in aFaces:
    Pface   = Igry[y:y+h, x:x+w]                # face patch
    Eyes    = EyeDet.detectMultiScale(Pface)    # eye detection on face patch
    aEyes.append(Eyes)                          # coords relative to face patch

#%% ----    Plot    ------
Irgb    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2RGB) # convert to RGB for displaying
from matplotlib.pyplot import *
import matplotlib.patches as patches
close('all')
figure(1)
imshow(Irgb, cmap=cm.gray)
for i,(x,y,w,h) in enumerate(aFaces):
    rect = patches.Rectangle((x,y),w,h, linewidth=1, edgecolor='r', facecolor='none')
    gca().add_patch(rect)
    # --- adding eyes
    Eyes    = aEyes[i]
    for (xe,ye,we,he) in Eyes:
```

```
        rect = patches.Rectangle((x+xe,y+ye),we,he, linewidth=1, edgecolor='g', facecolor='none')
        gca().add_patch(rect)
```

P.7.2 Pedestrian Detection

An example for pedestrian detection using HOG features (Section 10.3) exploiting the functions of OpenCV through Python. The usage is very similar to the face detection example.

```
import cv2

imPth   = 'C:\images\SomePhotoWithPedestrians.jpg'

PedDet     = cv2.HOGDescriptor()
PedDet.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

#%% ----    Load Image   ------
Ibgr    = cv2.imread(imPth)
Igry    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2GRAY)

#%% ====    Pedestrian Detection  ======
(aPers, Like) = PedDet.detectMultiScale(Igry, winStride=(4,4), padding=(8,8), scale=1.3)

#%% ----    Plot    ------
Irgb    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2RGB) # convert to RGB for displaying
from matplotlib.pyplot import *
import matplotlib.patches as patches
figure(1)
imshow(Irgb, cmap=cm.gray)
for i,(x,y,w,h) in enumerate(aPers):
    rect = patches.Rectangle((x,y),w,h, linewidth=1, edgecolor='r', facecolor='none')
    gca().add_patch(rect)
```

P.8 Object Recognition

The first example is one for (relatively) accurate object localization and recognition (Section **P.8.1**), the second one is a network geared toward speed (Section **P.8.2**).

P.8.1 Accurate

Section 11.1

Initializing and running the network follows the same declarations as for classification and segmentation networks. The novel part here is the analysis of object predictions.

In this specific example we load only one image for reason of simplicity. The network `NET` expects its input to be a list of images and so we place our image in square brackets, `[Irgb]`. The output is a list of predictions for each image, and since we use only one image in this example, we extract its prediction as `Prd = aImgPred[0]`. We firstly plot all candidates; then we apply a non-maximum suppression to select the candidates with the best scores, and then we plot those remaining ones again.

```
import numpy
import torchvision.models.detection as ObjRec
from torchvision import transforms
from skimage.io import imread

#%% ------    image path    ----------
#imgNa           = 'bicycle.jpg'     # the image to be processed
imgNa            = 'person.jpg'     # the image to be processed
#imgNa           = 'horses.jpg'     # the image to be processed
pthDarkNet  = 'c:/IMGdown/DEEPNETS/DarkNet/'
imgPth      = pthDarkNet + imgNa

#%% ------    Load an Image    ---------
Irgb        = imread(imgPth);
Trf2Tens    = transforms.ToTensor()
IrgbTo      = Trf2Tens(Irgb)

#%% %%%%%%%%%%%%%      NETWORK    %%%%%%%%%%%%%%%%%%%
NET         = ObjRec.fasterrcnn_resnet50_fpn(pretrained=True)

#%% %%%%%%%%%%%%%      APPLY    %%%%%%%%%%%%%%%%%%%
NET.eval()
aImgPred    = NET([IrgbTo])   # provide as list using square brackets []

#%% -----   Copied/Pasted from:
# https://pytorch.org/docs/stable/torchvision/models.html#object-detection-instance-segmentation-and-person-keypoint-detection
COCO_INSTANCE_CATEGORY_NAMES = [
    '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop sign',
    'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
    'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/A', 'N/A',
    'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
    'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket',
    'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
    'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
    'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table',
    'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone',
    'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A', 'book',
    'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'
]

#%% -----   Plot All Candidates    ------
from matplotlib import pyplot as plt
from matplotlib.patches import Rectangle
plt.figure(1); plt.imshow(Irgb)   # since we have only 1 image, we plot only that image
Prd     = aImgPred[0]           # prediction dictionary for this image
Lab     = Prd['labels']     # labels for this image
Scor    = Prd['scores'].detach().numpy()    # scores for this image
Boxes   = Prd['boxes'].detach().numpy()
```

```
nLab    = len(Lab)
for j in range(0,nLab):
    l   = Lab[j]
    print('%-10s  %1.6f'%(COCO_INSTANCE_CATEGORY_NAMES[l], Scor[j]))
    Bbx = Boxes[j,:]
    w   = Bbx[2]-Bbx[0]
    h   = Bbx[3]-Bbx[1]
    plt.gca().add_patch(Rectangle((Bbx[0],Bbx[1]),w,h,linewidth=1,edgecolor='r',facecolor='none'))
plt.show()

#%% ====    Maxima Selection    =====
from torchvision import ops as Ops
Prd     = aImgPred[0]           # prediction dictionary for this image
IxKeep  = Ops.nms(Prd['boxes'], Prd['scores'], 0.05)
print('kept %d/%d'%(len(IxKeep),len(Prd['boxes'])))


#%% -----   Plot Max Candidates    ------
plt.figure(2); plt.imshow(Irgb)    # since we have only 1 image, we plot only that image
Lab     = Prd['labels'][IxKeep]    # labels for this image
Scor    = Prd['scores'][IxKeep].detach().numpy()    # scores for this image
Boxes   = Prd['boxes'][IxKeep].detach().numpy()
nLab    = len(Lab)
for j in range(0,nLab):
    l   = Lab[j]
    print('%-10s  %1.6f'%(COCO_INSTANCE_CATEGORY_NAMES[l], Scor[j]))
    Bbx = Boxes[j,:]
    w   = Bbx[2]-Bbx[0]
    h   = Bbx[3]-Bbx[1]
    plt.gca().add_patch(Rectangle((Bbx[0],Bbx[1]),w,h,linewidth=1,edgecolor='r',facecolor='none'))
plt.show()
```

P.8.2 Fast: YOLO (You Only Look Once)

The YOLO network called from OpenCV (Section 11.3):

```
import numpy as np
import cv2 as cv
from skimage.io import imread

imgNa           = 'person.jpg'     # the image to be processed

# Initialize the parameters
thrConf         = 0.5           # confidence threshold
thrNonMaxSupp   = 0.4           # non-maximum suppression threshold
wthInput, hgtInput = 416,416   # height/width of network's input image

#%% ------    file paths    ----------
pthDarkNet      = 'c:/IMGdown/DEEPNETS/DarkNet/'
imgPth          = pthDarkNet + imgNa
fileClassLabels = pthDarkNet + 'coco.names'
fileConfig      = pthDarkNet + 'yolov3.cfg'
fileWeights     = pthDarkNet + 'yolov3.weights'

#%% ------    Load image    ---------
Irgb  = imread(imgPth);   hgtImg=Irgb.shape[0];  wthImg=Irgb.shape[1]

#%% ------    Load DarkNet    ---------
NET         = cv.dnn.readNetFromDarknet(fileConfig, fileWeights)
NET.setPreferableBackend(cv.dnn.DNN_BACKEND_OPENCV)
NET.setPreferableTarget(cv.dnn.DNN_TARGET_CPU)
# obtain names of unconnected layers
IxUnConn    = NET.getUnconnectedOutLayers()
aLayersNames= NET.getLayerNames()
aLayUnconn  = [aLayersNames[i[0]-1] for i in IxUnConn]
```

```
# load class labels
aClassLabels = None
with open(fileClassLabels, 'rt') as f:
    aClassLabels = f.read().rstrip('\n').split('\n')
nClassLabels    = len(aClassLabels)

#%% ============    Process One Image   =================
Iblob    = cv.dnn.blobFromImage(Irgb, 1/255, (wthInput, hgtInput), \
                        [0,0,0], 1, crop=False) # [1 3 hgtInp wthInp]
NET.setInput(Iblob)      # place input to net
ACONF    = NET.forward(aLayUnconn) # 3 arrays: [507 85],[2028 85],[8112 85]

#%% ============    Analyse Cells  =================
aCanClass, aCanConf, aCanBbox = [],[],[]
# ======    Loop Grids [507, 2028, 8112]     ========
for AConf in ACONF:
    # ======    Loop Cells     ========
    for Cand in AConf:
        # candidate vector is [1 85]: x/y/w/h + 80 class score values
        Scores  = Cand[5:]     # conf scores for the 80 classes
        ixClass = np.argmax(Scores)
        conf    = Scores[ixClass]
        if conf > thrConf:
            xCen    = int(Cand[0] * wthImg)
            yCen    = int(Cand[1] * hgtImg)
            wth     = int(Cand[2] * wthImg)
            hgt     = int(Cand[3] * hgtImg)
            left    = int(xCen-wth/2)
            top     = int(yCen-hgt/2)
            aCanBbox.append([left, top, wth, hgt])
            aCanClass.append(ixClass)
            aCanConf.append(float(conf))
            print('Candidate %s'%(aClassLabels[ixClass]))

# select winners: non-maximum suppression of bounding boxes
IxSel   = cv.dnn.NMSBoxes(aCanBbox, aCanConf, thrConf, thrNonMaxSupp)
print('%d winners left'%(len(IxSel)))

# classification (inference) duration
dur, _  = NET.getPerfProfile()
durStr  = 'Inference duration: %.2f ms'%(dur*1000.0/cv.getTickFrequency())

#%% -------   Plot    ---------
for i in IxSel:
    ix      = i[0]
    # display bounding box
    box     = aCanBbox[ix];   left=box[0];    top=box[1]
    right   = left+box[2]
    bottom  = top +box[3]
    cv.rectangle(Irgb, (left,top), (right,bottom), (0,0,255))
    # class label and its confidence
    conf    = aCanConf[ix]
    ixClass = aCanClass[ix]
    label = '%s:%.2f' % (aClassLabels[ixClass], conf)
    szLab, baseLine = cv.getTextSize(label, cv.FONT_HERSHEY_SIMPLEX, 0.5, 1)
    top     = max(top, szLab[1])
    cv.putText(Irgb, label, (left, top), cv.FONT_HERSHEY_SIMPLEX, 0.5, (255,255,255))

# duration in upper left of image
cv.putText(Irgb, durStr, (0, 15), cv.FONT_HERSHEY_SIMPLEX, 0.5, (0,0,255))

from matplotlib.pyplot import *
figure(1); imshow(Irgb)
```

P.9 Feature Extraction

P.9.1 Detection

Section 12.1

An example of Harris feature detection. It is compared to Matlab's instantiation of this type of detector.

```
clear;
I       = imread('cameraman.tif');
sigma   = 1 ;
radius  = 1 ;
thresh  = 0.185;

%% ----------- Step 1
gx      = repmat([-1 0 1],3,1);          % derivative of Gaussian (approximation)
gy      = gx';
Ix      = conv2(single(I), gx, 'same');
Iy      = conv2(single(I), gy, 'same');

%% ----------- Step 2 & 3
Glrg    = fspecial('gaussian', max(1,fix(5*sigma)), sigma); % Gaussian Filter
Ix2     = conv2(Ix.^2,  Glrg, 'same');
Iy2     = conv2(Iy.^2,  Glrg, 'same');
Ixy     = conv2(Ix.*Iy, Glrg, 'same');

%% ----------- Step 4
k       = 0.04;
MHRS    = (Ix2.*Iy2 - Ixy.^2) - k*(Ix2 + Iy2).^2;   % map [256 256]

%% ----------- Step 5
% Extract local maxima by performing a grey scale morphological
% dilation and then finding points in the corner strength image that
% match the dilated image and are also greater than the threshold.

sze     = 2*radius+1;                    % size of dilation mask.
Hmx     = ordfilt2(MHRS,sze^2,ones(sze));  % grayscale dilate.

% Make mask to exclude points within radius of the image boundary.
BWbord  = zeros(size(MHRS));
BWbord(radius+1:end-radius, radius+1:end-radius) = 1;

% Find maxima, threshold, and apply bordermask
Mn      = MHRS-min(MHRS(:));    Mn = Mn / max(Mn(:));
BHRS    = (MHRS==Hmx) & (Mn>thresh) & BWbord;

[r,c]   = find(BHRS);          % find row,col coords.
PtsIts  = [r c];             % list of interest points [nPts 2]
nDet    = length(r);

%% =====    Matlab Function     ======
FHar    = detectHarrisFeatures(I);

%% -----    Plot    ------
figure(1); clf; [nr nc]=deal(1,2); colormap(gray);
subplot(nr,nc,1);
    imagesc(I); hold on;
    plot(PtsIts(:,2),PtsIts(:,1),'y.');
    title(['Harris Own: #' num2str(nDet)]);
subplot(nr,nc,2);
    imagesc(I); hold on;
    plot(FHar.Location(:,1),FHar.Location(:,2),'y.');
    title(['Harris Mb: #' num2str(FHar.Count)]);
```

P.9.2 Finding Feature Correspondence in two Images

This code block is rewritten from Matlab's example on 'finding feature correspondences'. Matlab offers a function that draws a line between a pair of corresponding points, how convenient.

```
% Rewritten Matlab example of
% 'Find Corresponding Interest Points Between Pair of Images'
clear;
I1          = rgb2gray(imread('viprectification_deskLeft.png'));
I2          = rgb2gray(imread('viprectification_deskRight.png'));

%% =====    Detect Features     ======
Pt1det      = detectHarrisFeatures(I1);     % struct with .Locations [nPt1 2]
Pt2det      = detectHarrisFeatures(I2);

%% =====    Extract Features     ======
[D1 Pt1vld] = extractFeatures(I1, Pt1det);  % struct with .Features [nDsc 64]
[D2 Pt2vld] = extractFeatures(I2, Pt2det);

%% =====    Match Features     ======
IxPairs     = matchFeatures(D1, D2);        % [nMatches 2]
Pt1Matched  = Pt1vld(IxPairs(:,1), :);
Pt2Matched  = Pt2vld(IxPairs(:,2), :);

%% -----    Plot    ------
figure(1);
showMatchedFeatures(I1, I2, Pt1Matched, Pt2Matched);
```

P.9.3 Object Recognition

```
% Rewritten Matlab example 'Image Search Using Point Features'
clear;
%% -----    Prepare Collection of Images to Search      -----
aImgNames = {'elephant.jpg','cameraman.tif','peppers.png','saturn.png',...
    'pears.png','stapleRemover.jpg','football.jpg','mandi.tif',...
    'kids.tif','liftingbody.png','office_5.jpg','gantrycrane.png',...
    'moon.tif','circuit.tif','tape.png','coins.png'};
nImg        = numel(aImgNames);
% ----  Create a list of those images
EntryEmp    = struct('I',[],'Ithumb',[]);
AIMG        = repmat(EntryEmp,[1 nImg]);
szThumb     = 200;
for i = 1:nImg
    I       = imread(aImgNames{i});
    if size(I,3)==3,        % if color, then convert to gray-scale
        I = rgb2gray(I); end
    AIMG(i).I       = I;
    AIMG(i).Ithumb  = imresize(I, [szThumb,szThumb]);
end
%% -----    Choose Query Image & Region     -----
Q.Iwhole    = imread('clutteredDesk.jpg');
Bbox        = [130 175 330 365];        % our selection from that image
Q.I       = imcrop(Q.Iwhole, Bbox);   % only that will be queried

%% =====    Detect & Extract Features in Image Collection   ======
for i = 1:nImg
    Img         = AIMG(i);
    AIMG(i).Pts = detectSURFFeatures(Img.I, 'MetricThreshold',600);
    AIMG(i).Dsc = extractFeatures(Img.I, AIMG(i).Pts);
    AIMG(i).cFet = size(AIMG(i).Dsc,1);         % feature count
end
%% =====    Detect & Extract Features in Query Region   ======
% Consider only selected region - not entire image
Q.Pts           = detectSURFFeatures(Q.I, 'MetricThreshold',600);
[Q.Dsc, Q.Pts]  = extractFeatures(Q.I, Q.Pts);

%% =====    Search Query in Image Collection    ======
DSC         = double(vertcat(AIMG.Dsc));  % vectors for entire data set
DSCkdTree   = KDTreeSearcher(DSC);          % kd-Tree
% find for each query feature the 2 closest features
```

```
[IXNN DIST] = knnsearch(DSCkdTree, Q.Dsc, 'K',2);
% distance to the 2nd NN is used for removing outliers
BgoodRatio  = DIST(:,1) < DIST(:,2) * 0.8; % ratio test
BgoodDist   = DIST(:,1) < 0.25;           % distance threshold
Bgood       = IXNN(BgoodDist & BgoodRatio, 1);

% Count neibors and good matches for each image
IxIntvls    = [0, cumsum([AIMG.cFet])] + 1;
CountNN     = histcounts(IXNN(:,1), IxIntvls);
CountGM     = histcounts(Bgood,     IxIntvls);
mxCountNN   = max(CountNN);
mxCountGM   = max(CountGM);
if mxCountGM==0, error('no good features found'); end
if mxCountNN==0, error('no neighbors matches'); end

%% ---------    Plot    ----------
figure(1);
subplot(2,2,1);montage(cat(4,AIMG.Ithumb));
title('Image Collection');

subplot(2,2,2);
Pts1img = detectSURFFeatures(AIMG(1).I);
imshow(AIMG(1).I); hold on;
plot(selectStrongest(Pts1img, 100));
title('Strong Features from 1st Image');

subplot(2,2,3);
imshow(Q.Iwhole); hold on;
rectangle('Position',Bbox,'edgecolor','y');
title('Query Region (yellow)')

subplot(2,2,4);
imshow(Q.I); hold on;
plot(selectStrongest(Q.Pts, 100));
title('Strong Features from Query Image');

%% ---------    Plot    ----------
figure(2);
bar(IxIntvls, [CountNN 0], 'histc')
title('Number of Nearest Neighbor Features from Each Image')

%% ---------    Plot Nearest Neighbors   ----------
szImin    = 20;                   % size for images with no matches
sclFctI  = (szThumb - szImin)/2; % image scaling factor
valWhite  = 255;                  % we assume uint8 images

for i = 1:nImg
    szNew   = round(CountNN(i)/mxCountNN*sclFctI)*2 + szImin;
    szH     = (szThumb-szNew)/2;
    Iscl    = imresize(AIMG(i).Ithumb, [szNew szNew]);
    AIMG(i).IhistNN = padarray(Iscl, [szH szH], valWhite);
end
figure(3);
montage(cat(4,AIMG.IhistNN));
title('Size Corresponds to Number of Nearest Neighbor Occurrences');

%% ---------    Plot Matches    ----------
for i = 1:nImg
    szNew   = round(CountGM(i)/mxCountGM*sclFctI)*2 + szImin;
    szH     = (szThumb-szNew)/2;
    Iscl    = imresize(AIMG(i).Ithumb, [szNew szNew]);
    AIMG(i).IhistGM = padarray(Iscl, [szH szH], valWhite);
end
figure(4);
montage(cat(4, AIMG.IhistGM));
title('Size Corresponds to Number of Matched Features')
```

P.10 Feature Quantization

Building and applying a dictionary of (feature) words as introduced in Section 13.

P.10.1 Quantization Explicit

```
clear;
%% -----    Prepare Collection of Images to Search      -----
aImgNames = {'elephant.jpg','cameraman.tif','peppers.png','saturn.png',...
    'pears.png','stapleRemover.jpg','football.jpg','mandi.tif'};
nImg        = numel(aImgNames);

%% =====    Detect & Extract Features in Image Collection   ======
nDim        = 64;
DSC         = zeros(0,nDim);        LbImg = zeros(0,1);
for i = 1:nImg
    Img         = imread(aImgNames{i});
    if ndims(Img)==3,   % if color image, then turn into gray-scale image
        Img = rgb2gray(Img);
    end
    Pts         = detectSURFFeatures(Img);
    Dsc         = extractFeatures(Img, Pts);
    DSC         = [DSC; Dsc];
    % --- labels for images
    nVecImg     = size(Dsc,1);
    LbImg       = [LbImg; ones(nVecImg,1)*i];
end

%% ======   Quantize    ======
nWrd        = 500;
[WrdNo WRDc]= kmeans(DSC, nWrd, 'onlinephase','off');

%% ======   Word Numbers to Histogram   ========
IWHST  = zeros(nImg,nWrd);
for i = 1:nImg
    Bimg        = LbImg==i;
    IWHST(i,:) = histcounts(WrdNo(Bimg)',0:nWrd);
end
figure(1); imagesc(IWHST); colorbar; ylabel('Image No.'); xlabel('Word No.');

%% =====    LOOP WORD CLUSTERS   ======
c   = 0;
WRD     = zeros(nWrd,nDim);     aIx=cell(nWrd);     MemC=zeros(nWrd,1);
for i = 1:nWrd
    Bwrd  = WrdNo==i;                % identify the cluster indices
    mc      = nnz(Bwrd);               % member count
    if mc==0, continue; end            % skip if empty (or too small)
    c       = c+1;
    cen    = mean(DSC(Bwrd,:),1);   % centroid (cluster center=word)
    WRD(c,:)= cen;
    aIx{c} = find(Bwrd);              % member indices
    MemC(c) = mc;                      % # of members in cluster
end
WRD     = WRD(1:c,:);                  % actual dictionary
MemC    = MemC(1:c);                 % shorten to actual size

%% =====    Test Image      ======
Img         = imread('tape.png');       % unseen (novel/testing) image
if ndims(Img)==3,   % if color image, then turn into gray-scale image
    Img = rgb2gray(Img);
end
Pts         = detectSURFFeatures(Img);
Dsc         = extractFeatures(Img, Pts);

%% =====    Apply Dictionary    ======
[NN DIS]    = knnsearch(WRD, Dsc);
```

```
Iwhst       = histcounts(NN, 0:c);

figure(2); imagesc([IWHST; Iwhst]);  % show in comparison to others
```

P.10.2 Using Matlab's Functions

```
% Adapated from Matlab's example 'Image Category Classification Using
% Bag of Features'
clear;
foldRoot = 'C:\IMGdown\caltech101\';    % downloaded Caltech 101 set
% selecting 3 categories as our image collection to work with
aImgAll  = [ imageSet(fullfile(foldRoot, 'airplanes')), ...
             imageSet(fullfile(foldRoot, 'ferry')), ...
             imageSet(fullfile(foldRoot, 'laptop')) ];

%% -----    Folding   -------
% segregating collection into training and testing set
minCount  = min([aImgAll.Count]);                    % smallest image count
aImgAll    = partition(aImgAll, minCount, 'randomize');% trims to minCount
[aImgTren aImgTest] = partition(aImgAll, 0.3, 'randomize');

%% =====    Build Dictionary    =====
BAG     = bagOfFeatures(aImgTren);

%% ======   Train a Classifier  ======
Classif = trainImageCategoryClassifier(aImgTren, BAG);

%% ======   Predict with Classifier    =======
MxConf  = evaluate(Classif, aImgTest);
prdAcc  = mean(diag(MxConf));

%% =====    Apply Dictionary to 1 Image   =====
I       = read(aImgAll(1), 1);
Iwhst   = encode(BAG, I);
% Plot the histogram of word occurrences
figure(2);
bar(Iwhst);
xlabel('Word No.'); ylabel('Frequency');

%% =====    Apply Classifier to 1 Image   =====
I       = imread(fullfile(foldRoot, 'airplanes', 'image_0690.jpg'));
[IxPrd Scor] = predict(Classif, I);
Classif.Labels(IxPrd)

%% ------   Plot Some Images    ------
Iairpl      = read(aImgTren(1),1);
Iferry      = read(aImgTren(2),1);
Ilaptop     = read(aImgTren(3),1);
figure(1);
subplot(1,3,1);     imshow(Iairpl)
subplot(1,3,2);     imshow(Iferry)
subplot(1,3,3);     imshow(Ilaptop)
```

P.11 Image Processing II: Segmentation

P.11.1 Gray-Scale Images

```
clear;
I       = imread('coins.png');

%% -----    Thresholding with Otsu and Median
tOts    = graythresh(I);
BWots   = im2bw(I,tOts);
I       = single(I);       % the following computation is simpler in single
tMed    = median(I(:))/max(I(:));
BWmed   = im2bw(uint8(I),tMed);

%% -----    K-means
Ix      = kmeans(I(:),2);
BWkmen  = false(size(I));
BWkmen(Ix==2)   = true;

%% -----    Watershed
Ilpf    = conv2(I,fspecial('Gaussian',[5 5],2));
W       = watershed(Ilpf);

%% -----    Plotting
figure(1);clf;[nr nc] = deal(3,2);
subplot(nr,nc,1);
    imagesc(I); colormap(gray);
subplot(nr,nc,2);
    bar(histc(I(:),[0:255])); hold on;
    tOts    = tOts*256;
    plot([tOts tOts],[0 1000],'b:');
    title('Histogram');
subplot(nr,nc,3);
    imagesc(BWots); title('Otsu');
subplot(nr,nc,4);
    imagesc(BWmed); title('Median Value');
subplot(nr,nc,5);
    imagesc(BWkmen); title('K-means');
subplot(nr,nc,6);
    imagesc(W); title('Watershed');
```

```
from numpy             import arange, zeros, ones, median
from scipy.signal      import convolve2d
from sklearn.cluster   import KMeans
from scipy.ndimage     import label
from skimage           import data
from skimage.feature   import peak_local_max
from skimage.filters   import threshold_otsu
from skimage.segmentation import watershed
from fspecialGauss     import fspecialGauss
#%% -----   Load    ------
I       = data.coins()          # uint8
(m,n)   = I.shape
nPix    = m*n
#%% -----    Thresholding with Otsu and Median
tOts    = threshold_otsu(I)         # E [0 255]  uint8
BWots   = I > tOts
tMed    = median(I)                 # E [0 255] uint8
BWmed   = I > tMed

#%% -----    k-Means
Ivec    = I.reshape((nPix,1))
RKM     = KMeans(n_clusters=2, max_iter=30, n_init=1).fit(Ivec)
Cen     = RKM.cluster_centers_      # cluster centers [nK 3]
# Distances & nearest neibors
DIS     = zeros((nPix,2),float)
```

```
for i in range(2):
    DIS[:,i] = (Cen[i] - Ivec[:,0]) ** 2    # squared distance
IxNN    = DIS.argmin(axis=1)        # nearest neighbor
BWkmen  = IxNN.reshape((m,n))       # reshape to image for plotting

#%% -----    Watershed
Ilpf    = convolve2d(I, fspecialGauss(5,2), mode='same')
Mmx     = peak_local_max(Ilpf, labels=I, footprint=ones((3,3)), indices=False)
Mmrk    = label(Mmx)[0]
W       = watershed(Ilpf, Mmrk, mask=I);

#%% -----    Plotting
from matplotlib.pyplot import *
figure(figsize=(12,9));(nr,nc)=(3,2)
# image in upper left
subplot(nr,nc,1)
imshow(I, cmap=cm.gray);
# histogram in upper right
subplot(nr,nc,2)
hist(I.flatten(),arange(0,256))
plot([tOts,tOts],[0,1000],'b:')
plot([tMed,tMed],[0,1000],'r:')
title('Histogram')
# output of otsu
subplot(nr,nc,3)
imshow(BWots); title('Otsu')
# output of median
subplot(nr,nc,4)
imshow(BWmed); title('Median Value')
# output of kmeans
subplot(nr,nc,5)
imshow(BWkmen); title('K-means')
# output of watershed
subplot(nr,nc,6)
imshow(W); title('Watershed')
```

P.11.2 K-Means on Color Images

Section 14.3.1

```
clear;
% ------    Load image      -------
Irgb = imread('peppers.png');
szI  = size(Irgb);                  % [rows columns 3]
nPix    = szI(1)*szI(2);
% ------    Reshape to [nPix 3]      -------
ICol = reshape(Irgb,[nPix 3]);      % [nPix 3] format for clustering

%% =========    Kmeans      ==========
L  = kmeans(single(ICol),5, 'MaxIter',50, 'online','off'); % L E [1,..,5]
Lmx = reshape(L, szI(1:2));         % turn into label matrix

%% -----    Plot    -------
figure(1);clf;[nr nc] = deal(1,2);
subplot(nr,nc,1);
    imagesc(Irgb);
subplot(nr,nc,2);
    imagesc(Lmx);
```

```
from sklearn.cluster import KMeans
from skimage.io import imread

#%% -----   Load & Reshape  -------
Irgb    = imread('C:/IMGdown/SILU/Images/airplane/airplane00.tif')
(m,n,nCh) = Irgb.shape
nPix    = m*n
```

```
ICol    = Irgb.reshape((nPix,3)).astype(float)

#%% =======     Kmeans      =========
nK      = 4
RKM     = KMeans(n_clusters=nK, max_iter=20, n_init=1).fit(ICol)
Cen     = RKM.cluster_centers_      # [nK 3]
LbK     = RKM.labels_               # [nPix,1] E [0..nK-1]
```

```
import cv2 as cv
from numpy import float32

# ------    load image  ------
fp      = 'C:/someImage.tif'
Ibgr    = cv.imread(fp, cv.IMREAD_COLOR) # loads channels as BGR: blue/green/red
# ------    reshape to column-wise [nPix 3]     -------
szI     = Ibgr.shape
nPix    = szI[0]*szI[1]
ICOL    = Ibgr.reshape((nPix,3)).astype(float32)

#%% =======     Kmeans      =========
k           = 3         # number of desired clusters
nRep        = 1         # number of repetitions
termCrit    = (cv.TERM_CRITERIA_MAX_ITER, 10, 0.1)     # iter, epsilon
flagInit    = cv.KMEANS_PP_CENTERS                     # kmeans++
mes, Lb, Cen= cv.kmeans(ICOL, k, None, termCrit, nRep, flagInit)
BW          = Lb[:,0].reshape(szI[0],-1)     # labels to map
```

P.11.3 General Image Segmentation

A rapid procedure for general image segmentation (Section 14.4), which we also call *Fleckmentation*. The result is a list of regions (flecks) obtained from three different maps (`depth=3`). The following code is a slow implementation using connected-component labeling for each fleck separately, which is the most accurate but also the costliest. Much faster versions are possible at the price of slightly eroded regions. To deploy K-Means clustering, we would replace essentially the thresholding operation by a 2-Means clustering process.

```
% Fleckmentation with gray-level thresholding.
% Slow implementation with connected-component labeling for each fleck
% individually.
clear;
Irgb        = imread('peppers.png');
[m n nChB]  = size(Irgb);   szI=[m n];    nPix=m*n;
Igry        = rgb2gray(Irgb);

%% -----    Parameters      -----
depth       = 3 ;
minPixNode  = 10 ;      % min region size when segregating
%% -----    Init for 1st depth  -----
aFprv.PixelIdxList = 1:nPix;    nFprv=1;
%% -----    Init of Maps    -----
LMX         = zeros([szI depth]);   % label matrix for plotting

%% SSSSSSSSSSSSSSSS    LOOP DEPTHS      SSSSSSSSSSSSSSSSSSSSSSSSS
for d = 1:depth
    % ==========    LOOP PREVIOUS FLECKS    ==========
    aFlk=[];
    for f = 1:nFprv
        IxReg   = aFprv(f).PixelIdxList;
        if length(IxReg)<minPixNode, continue; end
        Ireg    = Igry(IxReg);
        thr     = graythresh(Ireg);
        BW      = Ireg > (thr*255);
        LbK     = BW(:)'+1;
```

```
        aIx     = {IxReg(LbK==1); IxReg(LbK==2)};
        % ======   Loop Image Clusters     ========
        for c = 1:2
            BW          = false(szI);   % empty map
            BW(aIx{c})  = true;         % place cluster into map
            aFlk        = [aFlk; regionprops(BW,'PixelIdxList')];
        end
    end
    aFprv   = aFlk;
    nFprv   = length(aFprv);
    fprintf('depth %d  #flecks %d\n', d, nFprv);
    % ---   Label Matrix For Plotting
    Lm  = zeros(szI);
    for f = 1:nFprv
        Ix      = aFlk(f).PixelIdxList;
        Lm(Ix)  = f;
    end
    LMX(:,:,d) = Lm;
end

%% ---------    Plot    --------
figure(1); colormap('default');
subplot(2,2,1); imagesc(Irgb);
for d = 1:depth
    subplot(2,2,d+1);
    imagesc(LMX(:,:,d));
end
```

P.12 Morphological Processing

Section 15

Section 15.1

P.12.1 Thinning

```
% Thinning as in Matlab's bwmorph.
%
% Adapted from:
% https://de.mathworks.com/matlabcentral/fileexchange/27291-thinning-image

BW      = imread('circbw.tif');
I       = BW(185:235,98:118);
Iorg    = I;            % for verification at end

%% =============    Thinning     ===========
% IN    I   black-white image unthinned
% OUT   O   black-white image thinned
[m n]       = size(I);
Ipad        = false([m n]+2);
Ipad(2:end-1,2:end-1) = I;
I           = Ipad;
clear Ipad;
O           = I ;
bChng       = true;
while bChng

    for i = 2:m+1
        for j = 2:n+1
            if I(i,j)>0
                p11 = I(i-1,j-1);  p12 = I(i-1,j); p13 = I(i-1,j+1);
                p21 = I(i,j-1  );                  p23 = I(i,j+1  );
                p31 = I(i+1,j-1);  p32  = I(i+1,j); p33 = I(i+1,j+1);

                %% Condition 1
                b1=0; b2=0; b3=0; b4=0;
                if p23==0 && (p13==1 || p12==1), b1 = 1; end
                if p12==0 && (p11==1 || p21==1), b2 = 1; end
                if p21==0 && (p31==1 || p32==1), b3 = 1; end
                if p32==0 && (p33==1 || p23==1), b4 = 1; end
                b = b1 + b2 + b3 + b4 ;
                if b==1
                    %% Condition 2
                    N1 = double( p23 | p13 ) + double( p12 | p11 ) + double( p21 | p31 ) + double( p32 | p33 );
                    N2 = double( p13 | p12 ) + double( p11 | p21 ) + double( p31 | p32 ) + double( p33 | p23 ) ;
                    cCond2 = min(N1,N2) ;
                    if cCond2>=2 && cCond2<=3
                        %% Condition 3
                        p33 = ~p33 ;
                        if (( p13 | p12 | p33 ) & p23)==0
                            O(i,j) = 0 ;
                        end
                    end
                end
            end
        end
    end
    %O1  = O;
    I       = O;
    bChng   = false;
    for i = 2:m+1
        for j = 2:n+1
            if I(i,j)>0
                p11 = I(i-1,j-1);  p12 = I(i-1,j); p13 = I(i-1,j+1);
                p21 = I(i,j-1  );                  p23 = I(i,j+1  );
                p31 = I(i+1,j-1);  p32  = I(i+1,j); p33 = I(i+1,j+1);

                %% Condition 1
                b1=0; b2=0; b3=0; b4=0;
```

```
                if p23==0 && (p13==1 || p12==1), b1 = 1; end
                if p12==0 && (p11==1 || p21==1), b2 = 1; end
                if p21==0 && (p31==1 || p32==1), b3 = 1; end
                if p32==0 && (p33==1 || p23==1), b4 = 1; end
                b = b1 + b2 + b3 + b4 ;
                if b==1
                    %% Condition 2
                    N1 = double( p23 | p13 ) + double( p12 | p11 ) + double( p21 | p31 ) + double( p32 | p33 );
                    N2 = double( p13 | p12 ) + double( p11 | p21 ) + double( p31 | p32 ) + double( p33 | p23 ) ;
                    cCond2 = min(N1,N2) ;
                    if cCond2>=2 && cCond2<=3
                        %% Condition 3
                        p11 = ~p11 ;
                        if (( p31 | p32 | p11 ) & p21)==0
                            O(i,j)  = 0 ;
                            bChng   = true;
                        end
                    end
                end
            end
        end
    end

    %bChng = sum(O1(:)-O(:))>0;
    I   = O;
end
O   = O(2:end-1,2:end-1);

%% ======   Verify   =====
BWthinMb    = bwmorph(Iorg,'thin','inf');
assert(all(BWthinMb(:)==O(:)));
```

P.12.2 Connected Components

Section 15.3.1

The following example of labeling connected components is an instance of the two-pass algorithm. It is implemented here with the union-find procedures squeezed in-between the first and second pass, as a separate function called f_SetDisJointSimp, appended below. To compare the output of the algorithm with that of Matlab's function, we need to rotate the input matrix.

```
clear;
% ------    Stimulus
I       = imread('cameraman.tif');
I       = I(235:245,140:180); % small size
BW      = I > 130;
szI     = size(BW);  szV=szI(1); szH=szI(2);

%% ---- Eliminate Borders ----
% our implementation cannot deal with pixels at the border, so we eliminate
BW(:,[1 end]) = 0;  % 1st and last column
BW([1 end],:) = 0;  % 1st and last row

%% SSSSSSSSSSSSSSSSSSSSSSS   1ST PASS   SSSSSSSSSSSSSSSSSSSSSSS
% generates candidate labels in map M
aLnk={};                        % list of candidates
M  = zeros(szV,szH,'uint16');   % marker array
cCan = 0;                       % candidate counter
for i = 2:szV
    for j = 2:szH-1
        if BW(i,j)==0, continue; end % skip if no pixel present
        % upper left corner: [1 2 4 7] E [3x3 indices column wise]
        bAnyNeib = any([BW(i-1,j-1)  BW(i-1,j)  BW(i-1,j+1)  BW(i,j-1)]);
        if ~bAnyNeib                % if no pixels in uplf neibor zone
            cCan   = cCan+1;        % increase counter
            aLnk{cCan}= [];         % ...then initialize as empty
            M(i,j)  = cCan;
```

```
        else
            % upper left corner for M:
            Ln      = [M(i-1,j-1) M(i-1,j) M(i-1,j+1) M(i,j-1)];
            Ln(Ln==0)=[];           % exclude zeros
            M(i,j)  = min(Ln);      % minimum label of that zone
            % ======    Link    ======
            Lu = unique(Ln);        % unique labels
            nU = length(Lu);
            for l = 1:nU,
                lu      = Lu(l);
                aLnk{lu} = unique([aLnk{lu} Lu]);
            end
        end
    end
end
M1st    = M;        % save as '1st pass' for plotting
SzL     = cellfun(@numel,aLnk);
fprintf('max set size %d\n', max(SzL));
fprintf('cL %d\n', cCan);

%% SSSSSSSSSSSSSSSSSSSSSSS   Disjoint Sets   SSSSSSSSSSSSSSSSSSSSSSS
% join multi-label regions to final (single) label
fprintf('Disjoint sets...');
DsJ     = f_SetDisJointSimp(aLnk);
nCC    = max(DsJ);      % # of connected components
fprintf('#lab %d  (max set size %d)-> #conn comp %d\n', cCan, max(SzL), nCC);

%% SSSSSSSSSSSSSSSSSSSSSSS   2ND PASS   SSSSSSSSSSSSSSSSSSSSSSS
% now we relabel the multi-label regions and collect region indices
fprintf('2nd pass...');
aIxLin = cell(nCC,1); % region indices per label
for i = 2:szV
    for j = 2:szH-1
        if BW(i,j)>0
            le      = DsJ(M(i,j));      % adjusted label
            M(i,j)  = le;               % re-assign with adjusted label
            aIxLin{le} = [aIxLin{le} (i-1)*szH+j]; % add linear index
        end
    end
end
fprintf('done\n');

%% SSSSSSSSSSSSSSSSSSSSSS      VERIFICATION      SSSSSSSSSSSSSSSSSSSSSSS
%% ------    Matlab  --------
fprintf('Running Matlab''s bwconncomp...');
CC      = bwconncomp(BW',8);
MlbMb   = labelmatrix(CC)';
NelMb   = cellfun(@numel, CC.PixelIdxList);
NelMbO  = sort(NelMb,'descend');
fprintf('done\n');

%% ------   Verify Matlab-Mine  -------
assert(CC.NumObjects==nCC);
Idf = abs(single(M)-single(MlbMb));
if any(Idf(:)), warning('not same'); end
for i = 1:CC.NumObjects
    IxMb = CC.PixelIdxList{i};
    IxMi = aIxLin{i};
    assert(all(IxMb==IxMi'));
end
```

```
% Simple form of disjoint. Not as elegant as union-find.
% Returns only new labels not unique lists.
%
% IN    aLnk    list of sets {nSets 1}
% OUT   DsJ     array holding new labels [nSets 1] E [1..nUnique]
%
```

```
function DsJ = f_SetDisJointSimp(aLnk)
cL  = length(aLnk);

%% =========    LOOP UNTIL SYMMETRIC    ========
while 1
    bChange = 0;
    for i = 1:cL
        Li  = aLnk{i};
        ni  = length(Li);
        for j = 1:cL
            if i==j, continue; end
            Lj      = aLnk{j};
            Bmem    = ismember(Li,Lj);
            nMem    = nnz(Bmem);
            if nMem>0 && nMem~=ni
                bChange = 1;
                aLnk{i} = union(Li,Lj);
                aLnk{j} = aLnk{i};
            end
        end
    end
    if bChange==0, break;  end
end

%% =========    Identify DisJoints      ==========
cD      = 0;                    % counter disjoint
DsJ     = zeros(cL,1);          % disjoint label
for i = 1:cL
    L1 = aLnk{i};  n1=length(L1);
    if DsJ(i)>0, continue; end  % labeled already
    if n1==0,                   % no partner labels?
        cD      = cD+1;         % ...then disjoint anyway
        DsJ(i)  = cD;
        continue;
    end
    cRem = cL-i;
    fprintf('%d  cRem %d\n', i, cRem);
    Bdsj    = zeros(cL-i,1);
    for j = i+1:cL
        if length(aLnk{j})==n1
            Bsame   = L1==aLnk{j};
            if all(Bsame)
                if DsJ(i)==0,
                    cD      = cD+1;
                    DsJ(i)  = cD;
                end
                DsJ(j)  = DsJ(i);
            else
                Bdsj(j-i) = 1;
            end
        else
            Bdsj(j-i) = 1;
        end
    end
    % Bdsj
    if all(Bdsj)
        cD      = cD+1;
        DsJ(i)  = cD;
    end
end

end
```

P.12.3 Boundary Following

Section 15.3.2

```
% Tracing one boundary.
clear;
Igry    = imread('cameraman.tif');
BW      = Igry < 50;

%% ------    Elim Borders   ------
BW(1,:)   = 0;   BW(:,1)   = 0;
BW(end,:) = 0;   BW(:,end) = 0;

%% ======    1st Pixel & Direction       ======
[rw cl] = find(BW==1, 1, 'first');
dir1st  = [1; 0];

%% ------    Init lists   ------
rwPx    = rw;
clPx    = cl;
nPxImg  = prod(size(BW));
nPxHlf  = round(nPxImg/2);
% pixels:
RwPx    = zeros(nPxHlf,1);
ClPx    = zeros(nPxHlf,1);
RwPx(1) = rwPx;
ClPx(1) = clPx;
% directions:
RwDi    = zeros(nPxHlf,1);
ClDi    = zeros(nPxHlf,1);
RwDi(1) = dir1st(1);
ClDi(1) = dir1st(2);
rwDi    = dir1st(1);
clDi    = dir1st(2);

%% ========   LOOP  =======
% traces one contour and then breaks the loop
i = 1;
while 1
    i = i+1;

    rwDi    = -rwDi;
    clDi    = -clDi;
    rwPx    = rwPx + rwDi;
    clPx    = clPx + clDi;
    if BW(rwPx,clPx) == 1  % ------ TEST 1   -------
        RwPx(i) = rwPx;  ClPx(i) = clPx;
        RwDi(i) = rwDi;  ClDi(i) = clDi;
    else
        d2      = -rwDi;
        rwDi    =  clDi;
        clDi    = d2;
        rwPx    = rwPx + rwDi;
        clPx    = clPx + clDi;
        if BW(rwPx,clPx) == 1   % ------ TEST 2   -------
            RwPx(i) = rwPx;  ClPx(i) = clPx;
            RwDi(i) = rwDi;  ClDi(i) = clDi;
        else
            d2      = -rwDi;
            rwDi    =  clDi;
            clDi    = d2;
            rwPx    = rwPx + rwDi;
            clPx    = clPx + clDi;
            if BW(rwPx,clPx) == 1  % ------ TEST 3   -------
                RwPx(i) = rwPx;  ClPx(i) = clPx;
                RwDi(i) = rwDi;  ClDi(i) = clDi;
            else
                rwPx  = rwPx + rwDi;
                clPx  = clPx + clDi;
```

```
            if BW(rwPx,clPx) == 1  % ------ TEST 4   -------
                RwPx(i) = rwPx;  ClPx(i) = clPx;
                RwDi(i) = rwDi;  ClDi(i) = clDi;
            else
                d2      = -rwDi;
                rwDi    =  clDi;
                clDi    = d2;
                rwPx    = rwPx + rwDi;
                clPx    = clPx + clDi;
                if BW(rwPx,clPx) == 1  % ------ TEST 5   -------
                    RwPx(i) = rwPx;  ClPx(i) = clPx;
                    RwDi(i) = rwDi;  ClDi(i) = clDi;
                else
                    rwPx  = rwPx + rwDi;
                    clPx  = clPx + clDi;
                    if BW(rwPx,clPx) == 1  % ------ TEST 6   -------
                        RwPx(i) = rwPx;  ClPx(i) = clPx;
                        RwDi(i) = rwDi;  ClDi(i) = clDi;
                    else
                        d2      = -rwDi;
                        rwDi    =  clDi;
                        clDi    = d2;
                        rwPx    = rwPx + rwDi;
                        clPx    = clPx + clDi;
                        if BW(rwPx,clPx) == 1  % ------ TEST 7   -------
                            RwPx(i) = rwPx;  ClPx(i) = clPx;
                            RwDi(i) = rwDi;  ClDi(i) = clDi;
                        else
                            rwPx  = rwPx + rwDi;
                            clPx  = clPx + clDi;
                            if BW(rwPx,clPx) == 1  % ------ TEST 8   -------
                                RwPx(i) = rwPx;  ClPx(i) = clPx;
                                RwDi(i) = rwDi;  ClDi(i) = clDi;
                            else
                                d2      = -rwDi;
                                rwDi    =  clDi;
                                clDi    = d2;
                                rwPx    = rwPx + rwDi;
                                clPx    = clPx + clDi;
                                if BW(rwPx,clPx) == 1  % ------ TEST 9   -------
                                    RwPx(i) = rwPx;  ClPx(i) = clPx;
                                    RwDi(i) = rwDi;  ClDi(i) = clDi;
                                else
                                    d2      = -rwDi;
                                    rwDi    =  clDi;
                                    clDi    = d2;
                                    rwPx    = rwPx + rwDi;
                                    clPx    = clPx + clDi;
                                    if BW(rwPx,clPx) == 1  % ------ TEST 10   -------
                                        RwPx(i) = rwPx;  ClPx(i) = clPx;
                                        RwDi(i) = rwDi;  ClDi(i) = clDi;
                                    end % 10
                                end % 9
                            end % 8
                        end % 7
                    end % 6
                end % 5
            end % 4
        end % 3
    end % 2
end % 1
cPix = i;
% check for stopping criterions
if cPix>2 % from 3rd pixel on:
    if  rwPx==RwPx(2) && clPx==ClPx(2) && ... % same pixel?
        rwDi==RwDi(2) && clDi==ClDi(2)        % same direction?
        break;
```

```
        end
    end
end
% shorten and remove the 2nd visit to the 2nd pixel
RwPx  = RwPx(1:i-2);
ClPx    = ClPx(1:i-2);

%% ---------   Plot   ----------
figure(1); clf;
subplot(2,1,1); imagesc(Igry); colormap(gray);
subplot(2,1,2); imagesc(BW); hold on;
plot(ClPx,RwPx,'-');
```

P.13 Shape

Section 16

In subsection P.13.1 we create a set of shapes. In subsection P.13.2 we measure their similarity with simple properties; in subsection P.13.3 we use the radial signature to provide better similarity performance.

P.13.1 Generating Shapes

```
clear;
I       = ones(150,760)*255;       % empty image
sz      = 28;
Ix      = 65:90;            % indices for A to Z
nShp    = length(Ix);
figure(1);clf;
imagesc(I); hold on;  axis off;
for i = 1:nShp
    ix  = Ix(i);
    text(i*sz,20,char(ix),'fontweight','bold');
    text(i*sz,50,char(ix),'fontweight','bold' ,'fontsize',12);
    text(i*sz,110,char(ix),'fontweight','bold','fontsize',12,'rotation',45);
    text(i*sz,80,char(ix),'fontweight','bold' ,'fontsize',14);
    text(i*sz,140,char(ix),'fontweight','bold','fontsize',16);
end
print('ShapeLetters','-djpeg','-r300');
```

```
from numpy             import ones, arange
from matplotlib.pyplot import *

I       = ones((150,760))       # empty image
I[0,0]  = 0                     # to ensure that the colormap uses its range
sz      = 28
Ix      = arange(65,91)         # indices for A to Z
nShp    = len(Ix)
figure(figsize=(12,5))
imshow(I,cmap=cm.gray); axis('off')
xoff    = 8
for i in arange(0,nShp):
    ix  = Ix[i]
    text(i*sz+xoff,20, chr(ix),fontweight='bold')
    text(i*sz+xoff,50, chr(ix),fontweight='bold',fontsize=12)
    text(i*sz+xoff,110,chr(ix),fontweight='bold',fontsize=12,rotation=45)
    text(i*sz+xoff,80, chr(ix),fontweight='bold',fontsize=14)
    text(i*sz+xoff,140,chr(ix),fontweight='bold',fontsize=16)

savefig('ShapeLettersPy.jpeg', facecolor='w', bbox_inches='tight')
```

P.13.2 Simple Measures

```
clear;
BW      = rgb2gray(imread('ShapeLetters.jpg')) < 80;
%% =====    Shape Properties
RG      = regionprops(BW, 'all');
Ara     = cat(1,RG.Area);
Ecc     = cat(1,RG.Eccentricity);
EqD     = cat(1,RG.EquivDiameter);
BBx     = cat(1,RG.BoundingBox);
Vec     = [Ara Ecc EqD];            % [nShp 3] three dimensions
nShp    = length(Ara);
fprintf('# Shapes %d\n', nShp);

%% =====    Pairwise Distance Measurements
DM      = squareform(pdist(Vec));   % distance matrix [nShp nShp]
```

```
DM(diag(true(nShp,1))) = nan;       % inactivate own shape
[DO O]  = sort(DM,2,'ascend');      % sort along rows

%% -----    Plotting First nSim Similar Shapes for Each Found Shape
% Bounding box
UL      = floor(BBx(:,1:2));    % upper left corner
Wth     = BBx(:,3);         % width
Hgt     = BBx(:,4);         % height
mxWth   = max(Wth)+2;
mxHgt   = max(Hgt)+2;
nSim    = 20;               % # similar ones we plot
nShp2   = ceil(nShp/2);
[ID1 ID2] = deal(zeros(nShp2*mxWth,nSim*mxHgt));
for i = 1:nShp

    % --- given/selected shape in 1st row
  Row     = (1:Hgt(i))+UL(i,2);   % rows
   Col     = (1:Wth(i))+UL(i,1);   % columns
   Sbw     = BW(Row,Col);
    Szs     = size(Sbw);
    if i<=nShp2,ID1((1:Szs(1))+(i-1)*mxHgt,    1:Szs(2)) = Sbw*2;
    else        ID2((1:Szs(1))+(i-nShp2)*mxHgt,1:Szs(2)) = Sbw*2;
    end

    % --- similar shapes in rows 2nd to nShp+1
    for k = 1:nSim
        ix      = O(i,k);
      Row     = (1:Hgt(ix))+UL(ix,2);   % rows
        Col     = (1:Wth(ix))+UL(ix,1);   % columns
        Sbw     = BW(Row,Col);
        Szs     = size(Sbw);
        if i<=nShp2, ID1((1:Szs(1))+(i-1)*mxHgt,    (1:Szs(2))+k*mxWth) = Sbw;
        else         ID2((1:Szs(1))+(i-nShp2)*mxHgt,(1:Szs(2))+k*mxWth) = Sbw;
        end
    end
end

figure(1);clf;
subplot(1,2,1); imagesc(ID1); title('First Half of Letters');
subplot(1,2,2); imagesc(ID2); title('Second Half of Letters');
```

```
from numpy import asarray, asmatrix, concatenate, nan, diagflat, floor, ceil, \
                  arange, zeros, ones, ix_, sort, argsort
from skimage.io        import imread
from skimage.color     import rgb2gray
from skimage.measure   import label, regionprops
from scipy.spatial.distance import pdist, squareform

BW      = rgb2gray(imread('ShapeLettersPy.jpeg')) < 0.3

#%% =====    Shape Properties   =====
LB      = label(BW) #BW.astype(int))
RG      = regionprops(LB)
nShp    = len(RG)
print('# Shapes', nShp)

#%% =====   Create Attribute Vectors    ======
# we use 'asmatrix' instead of 'asarray' to create matrix 'Vec'
Ara     = asmatrix([r.area for r in RG])         # called list comprehension
Ecc     = asmatrix([r.eccentricity for r in RG])
EqD     = asmatrix([r.equivalent_diameter for r in RG])
Vec     = concatenate((Ara,Ecc,EqD)).conj().T   # [nShp 3] three dimensions

#%% =====    Pairwise Distance Measurements
DM      = squareform(pdist(Vec))                # distance matrix [nShp nShp]
DM[diagflat(ones((1,nShp),dtype=bool))] = nan   # inactivate own shape
DO      = sort(DM,axis=1)                       # sort along rows
```

```
O       = argsort(DM,axis=1)                    # obtain indices separately

#%% -----    Plotting First nSim Similar Shapes for Each Found Shape
BBx     = asarray([r.bbox for r in RG])     # Bounding box
UL      = floor(BBx[:,0:2])                 # upper left corner
#LR      = floor(BBx[:,2:4])                 # lower rite corner
Hgt     = BBx[:,2]-BBx[:,0]                 # width
Wth     = BBx[:,3]-BBx[:,1]                 # height
mxHgt   = max(Hgt)+2
mxWth   = max(Wth)+2
nSim    = 20                                # similar ones we plot
nShp2   = int(ceil(nShp/2))
ID1     = zeros((nShp2*mxHgt,nSim*mxWth))
ID2     = ID1.copy()
for i in range(nShp):
    # --- given/selected shape in 1st row
    Row    = arange(0,Hgt[i])+UL[i,0]-1    # rows
    Col    = arange(0,Wth[i])+UL[i,1]-1    # columns
    Sbw    = BW[ix_(Row.astype(int), Col.astype(int))]
    Szs    = Sbw.shape
    RgV    = arange(0,Szs[0])           # vertical range
    RgH    = arange(0,Szs[1])           # horizontal range
    if i < nShp2: ID1[ix_(RgV+i*mxHgt,         RgH)] = Sbw*2
    else:         ID2[ix_(RgV+(i-nShp2)*mxHgt, RgH)] = Sbw*2

    # --- similar shapes in rows 2nd to nShp+1
    for k in range(nSim-1):
        ix      = O[i,k]
        Row     = arange(0,Hgt[ix])+UL[ix,0]-1    # rows
        Col     = arange(0,Wth[ix])+UL[ix,1]-1    # columns
        Sbw     = BW[ix_(Row.astype(int),Col.astype(int))]
        Szs     = Sbw.shape
        RgV     = arange(0,Szs[0])              # vertical range
        RgH     = arange(0,Szs[1])              # horizontal range
        if i < nShp2:   ID1[ix_(RgV+i*mxHgt,        RgH+(k+1)*mxWth)] = Sbw
        else:           ID2[ix_(RgV+(i-nShp2)*mxHgt,RgH+(k+1)*mxWth)] = Sbw

from matplotlib.pyplot import *
figure(figsize=(20,20))
subplot(1,2,1); imshow(ID1); title('First Half of Letters')
subplot(1,2,2); imshow(ID2); title('Second Half of Letters')
```

P.13.3 Shape: Radial Signature

```
clear;
BW      = imread('text.png');       % the stimulus
aBonImg = bwboundaries(BW);
nShp    = length(aBonImg);
fprintf('# Shapes %d\n', nShp);

%% =====    Shape Properties
Vec     = zeros(nShp,4);
aBon    = cell(nShp,1);
for i = 1:nShp
    Bon     = aBonImg{i};
    nPix    = length(Bon);
    cenPt   = mean(Bon,1);
    Rsig    = sqrt(sum(bsxfun(@minus,Bon,cenPt).^2,2));
    Fdsc    = abs(fft(Rsig));           % fast Fourier
    FDn     = Fdsc(2:end)/Fdsc(1);      % normalization by 1st FD
    fprintf('%3d  #pix %2d  FFD %2d\n', i, nPix, length(FDn));
    Vec(i,:) = FDn(1:4);
    aBon{i} = bsxfun(@minus,Bon,cenPt-[10 10]);  % move into [1..20 1..20]
    if 0
        figure(1); imagesc(zeros(20,20)); hold on;
```

```
        plot(aBon{i}(:,2), aBon{i}(:,1));
        pause();
    end
end
clear aBonImg

%% =====    Pairwise Distance Measurements
DM      = squareform(pdist(Vec));   % distance matrix [nShp nShp]
DM(diag(true(nShp,1))) = nan;       % inactivate own shape
[DO O]  = sort(DM,2,'ascend');      % sort along rows

%% -----    Plotting First nSim Similar Shapes for Each Found Shape
sz      = 19;
nSim    = 20;               % # similar ones we plot
nShp2   = ceil(nShp/2);
[ID1 ID2] = deal(zeros(nShp2*sz,nSim*sz));
figure(1);clf;
subplot(1,2,1); imagesc(ID1); title('First Half of Letters'); hold on;
subplot(1,2,2); imagesc(ID2); title('Second Half of Letters'); hold on;
for i = 1:nShp

    % --- given/selected shape in 1st row
    Bon     = aBon{i};
    if i<=60,
        subplot(1,2,1);
        plot(Bon(:,2), Bon(:,1)+(i-1)*sz,'b');
    else
        subplot(1,2,2);
        plot(Bon(:,2), Bon(:,1)+(i-61)*sz,'b');
    end

    % --- similar shapes in rows 2nd to nShp+1
    for k = 1:nSim
        ix      = O(i,k);
        Bon     = aBon{ix};
        if i<=60
            subplot(1,2,1);
            plot(Bon(:,2)+k*sz,Bon(:,1)+(i-1)*sz,'k');

            % ID1((1:Szs(1))+(i-1)*mxHgt, (1:Szs(2))+k*mxWth) = Sbw;
        else
            subplot(1,2,2);
            plot(Bon(:,2)+k*sz,Bon(:,1)+(i-61)*sz,'k');
            %ID2((1:Szs(1))+(i-45)*mxHgt,(1:Szs(2))+k*mxWth) = Sbw;
        end
    end
end
```

P.14 Contour

P.14.1 Hough Transform (Straight Lines)

Section 17.1

The following example is a modification of Matlab's code example, adapted to our notation.

```
clear;
nLin    = 8;
minLen  = 17;
% ----  create a stimulus
Igry    = imread('circuit.tif');
Irot    = imrotate(Igry,33,'crop');

%% =====    Edges     =====
BW          = edge(Irot,'canny');

%% =====    Hough Transform & LineDetection     =====
[AC The Rho]= hough(BW);                % AC = [rho theta]
Pek         = houghpeaks(AC, nLin);     % [nPek 2] row/column
aLines      = houghlines(BW, The, Rho, Pek, 'FillGap',5, 'MinLength',minLen);

%% ----------Plot   ----------
figure(1);clf;
subplot(1,2,1);
    imshow(AC,[], 'XData',The, 'YData',Rho, 'InitialMagnification','fit');
    xlabel('\theta (Angle)'), ylabel('\rho (Radius)');
    axis on, axis normal, hold on;
    Xco   = The(Pek(:,2));    Yco = Rho(Pek(:,1));
    plot(Xco,Yco,'s','color','white');

subplot(1,2,2);
    imshow(Irot), hold on
    maxLen = 0;
    for k = 1:length(aLines)
        Co = [aLines(k).point1; aLines(k).point2];
        % line
        plot(Co(:,1),Co(:,2),'LineWidth',2,'Color','green');
        % beginnings and ends of lines
        plot(Co(1,1),Co(1,2),'x','LineWidth',2,'Color','yellow');
        plot(Co(2,1),Co(2,2),'x','LineWidth',2,'Color','red');

        % remember longest line
        len     = norm(aLines(k).point1 - aLines(k).point2);
        if len>maxLen
            maxLen  = len;
            CoLong  = Co;
        end
    end
```

P.14.2 Ridge, River and Edge Contours

Detection of contour pixels by observing relational operations between neighbors (Section 17.2).

```
% Ridge, river and edge detection.
% From 'Rapid Contour Detection for Image Classification'
% http://digital-library.theiet.org/content/journals/10.1049/iet-ipr.2017.1066
% DOI:  10.1049/iet-ipr.2017.1066
% https://www.researchgate.net/publication/321218540_Rapid_Contour_Detection_for_Image_Classification
clear;
I       = imread('cameraman.tif');
[m n]  = size(I);

%% -----    Parameters
s       = 1 ;       % search radius
minCtr  = 0.05;
```

```
%% ------------------   Subimages & Indices     -----------------
rr = s+1:m-s;      rrN = rr-s;     rrS = rr+s;
cc  = s+1:n-s;      ccE = cc+s;     ccW = cc-s;
CEN = I(rr,cc);                        % center
NN  = I(rrN,cc);    SS = I(rrS,cc);   % north, south
EE  = I(rr, ccE);   WW = I(rr, ccW);    % east, west
NE  = I(rrN,ccE);   SE = I(rrS,ccE);    % north east, south east
SW  = I(rrS,ccW);   NW = I(rrN,ccW);    % south west, north west

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                  R I D G E S   &   R I V E R S
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% ============= EXTREMA Along Axes & Diagonals
% --- Maxima (counter-clockwise)
MX1     = padarray(CEN>NN & CEN>SS, [s s]);  % max north-south
MX2     = padarray(CEN>NW & CEN>SE, [s s]);  % max diag 1
MX3     = padarray(CEN>WW & CEN>EE, [s s]);  % max west-east
MX4     = padarray(CEN>NE & CEN>SW, [s s]);  % max diag 2
% --- Minima
MN1     = padarray(CEN<NN & CEN<SS, [s s]);  % min north-south
MN2     = padarray(CEN<NE & CEN<SW, [s s]);  % min diag 1
MN3     = padarray(CEN<WW & CEN<EE, [s s]);  % min west-east
MN4     = padarray(CEN<NW & CEN<SE, [s s]);  % min diag 2

Cmax = uint8(MX1+MX2+MX3+MX4);      % map of maxima count
Cmin    = uint8(MN1+MN2+MN3+MN4);      % map of minima count

%% ============= Suppress Low RR Contrast
R           = colfilt(I,[2 2]+s, 'sliding', @range);    % range image
CtrXtr      = R(logical(Cmax) | logical(Cmin));         % [nExtrema 1]
thrXtr      = max(CtrXtr)*minCtr;       % threshold for low contrast
Blow        = R < thrXtr;               % map with low contrast pixels
Cmax(Blow)  = 0;                        % eliminate low contrast
Cmin(Blow) = 0;

%% ============= Ridge/River Maps
Mrdg        = Cmax >= 2;
Mriv        = Cmin >= 2;
Mrdg       = bwmorph(Mrdg, 'clean');
Mriv       = bwmorph(Mriv, 'clean');
Mrdg      = bwmorph(Mrdg, 'thin', 'inf');
Mriv      = bwmorph(Mriv, 'thin', 'inf');

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                            E D G E S
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% ------------------   Subimages of Range Image
RCN     = R(rr, cc);         % center
Rnn     = R(rrN,cc);           % north
Rss     = R(rrS,cc);           % south
Ree     = R(rr, ccE);         % east
Rww     = R(rr, ccW);         % west
Rne     = R(rrN,ccE);           % north east
Rse     = R(rrS,ccE);           % south east
Rsw     = R(rrS,ccW);           % south west
Rnw     = R(rrN,ccW);        % north west

%% ================= MAXIMA Along Axes & Diagonals
RX1     = padarray(RCN>Rnn & RCN>Rss, [s s]);   % max north-south
RX2     = padarray(RCN>Rnw & RCN>Rse, [s s]);   % max diag 1
RX3     = padarray(RCN>Rww & RCN>Ree, [s s]);   % max west-east
RX4     = padarray(RCN>Rne & RCN>Rsw, [s s]);   % max diag 2
Cedg  = uint8(RX1+RX2+RX3+RX4);              % maxima count

%% ================= Suppress Low Contrast
```

```
Cedg        = padarray(Cedg(rr,cc), [s s]);     % blank borders
BWedg      = logical(Cedg);
CtrEdg      = R(BWedg);                 % range value at edge candidates

thrEdg      = max(CtrEdg)*minCtr*2;     % threshold for low contrast
BlowEdg     = CtrEdg < thrEdg;          % identify low-contrast edges
IxEdg       = find(BWedg);              % linear index of edge candidates
IxLow       = IxEdg(BlowEdg);
Cedg(IxLow) = 0;                        % eliminate low contrast

%% ================= Edge Map (Ridges in RangeImage)
Medg     = Cedg >= 2;
Medg        = bwmorph(Medg, 'clean');
Medg        = bwmorph(Medg, 'thin', 'inf');

%% ---------    Plotting    --------
[RwRdg ClRdg]   = find(Mrdg);
[RwRiv ClRiv]   = find(Mriv);
[RwEdg ClEdg]   = find(Medg);

figure(1); clf; colormap(gray);
imagesc(I); hold on;
plot(ClRdg,RwRdg,'g.');
plot(ClRiv,RwRiv,'b.');
plot(ClEdg,RwEdg,'c.');
```

P.14.3 Iso-Contour Detection (and Following)

Section 17.2.2

```
"""
Closes contour as in Matlab
Finds inside contours (holes) as well
4-connected by default
Coordinates range from [0 sz-1] (not [1 sz] as in Matlab)
1 pix:    5 points
2 pix:    7 points
"""
from numpy import zeros, arange
from skimage.measure import find_contours
from matplotlib.pyplot import *

# ------ BW image with some stimuli
M  = zeros((10,11),bool)
M[1,1]        = 1                       # a single pixel
M[3:5,1]      = 1                       # two pixels
M[[1,4],3:7]=1;  M[1:5,[3,6]]=1         # rectangle
M[6,3:6]    =1;  M[6:9,4]    =1         # T-feature
M[7,7:9]    =1;  M[6,8]=1;   M[8,9]=1;  # Y-feature at 45 deg

#%% =====    Tracing    =======
aBon    = find_contours(M, 0.99)   # closer to contour pixels
aBonE   = find_contours(M, 0.2)    # closer to exterior pixels

#%% -----    Plotting    -------
figure(1)
imshow(M, interpolation='nearest')
nB = len(aBon)
for i in arange(0,nB):
    Bon     = aBon[i]
    BonE    = aBonE[i]
    print('#pix  %d, %d'%(Bon.shape[0],BonE.shape[0]))
    plot(Bon[:,1], Bon[:,0], 'r')
    plot(BonE[:,1], BonE[:,0], 'y')
```

For the OpenCV example (accessed through Python) we omit the plotting part for reason of brevity.

```
"""
Boundary tracing in OpenCV. Results are more similar to Matlab than Python.
Does not close contour (as in Matlab or Python)
Coordinates range from [0 sz-1] (not [1 sz] as in Matlab)
1 pix:     1 point
2 pix:     2 points
"""
import cv2
from numpy import uint8, zeros

# ------ BW image with some stimuli
M  = zeros((10,11),bool)
M[1,1]        = 1                       # a single pixel
M[3:5,1]      = 1                       # two pixels
M[[1,4],3:7]=1;  M[1:5,[3,6]]=1         # rectangle
M[6,3:6]    =1;  M[6:9,4]    =1         # T-feature
M[7,7:9]    =1;  M[6,8]=1;   M[8,9]=1;  # Y-feature at 45 deg

#%% ---------   Measure     --------
BWout, aCnt, Hier   = cv2.findContours(M.astype(uint8), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

cCnt    = len(aCnt)
print(f'#cnt {cCnt}')
# squeeze middle axis and change from x/y to row/col
for i in range(cCnt): aCnt[i]=aCnt[i][:,0][:,[1,0]]
for i in range(cCnt):
    print(aCnt[i].shape[0])
```

P.14.4 Curvature Signature

Section 17.5

```
clear;
%% -----    Boundaries from rice image
I      = imread('rice.png');
BW     = im2bw(I, graythresh(I));
[aBon,L]= bwboundaries(BW,'noholes');
% -----    Take 1st boundary:
Bon    = aBon{1};              % use row/col (matrix) axis
Ro      = Bon(:,2);             % original row (for plotting)
Co    = Bon(:,1);             % original column
% adds inter-point spacing as 3rd column:
Bon    = [Bon [0; sqrt(sum(diff(Bon,1).^2,2))]];

%% -----    Low-PassFilter (Gaussian)   ------
sgm   = 2;
w     = 5;
Lpf     = normpdf(-w:w,0,sgm)';     % Gaussian at 0 with sigma=1

%% =====    Low-Pass Filter to Suppress Noise   =======
Rf    = conv(Ro,Lpf,'same');
Cf   = conv(Co,Lpf,'same');
Rf    = [Ro(1:w); Rf(w+1:end-w); Ro(end-w:end)];
Cf     = [Co(1:w); Cf(w+1:end-w); Co(end-w:end)];

%% =====    Derivatives      =======
Y1      = [0; diff(Rf)];        % 1st derivative
X1      = [0; diff(Cf)];
Y2      = [0; diff(Y1)];        % 2nd derivative
X2      = [0; diff(X1)];

%% =====    Curvature Simple    =====
Drv2    = Y2+X2;                    % adding 1st derivatives

%% =====    Curvature Accurate  ======
K       = (X1.*Y2 - Y1.*X2) ./ ( (X1.^2 + Y1.^2).^1.5);
```

```
%% -----    Plot    -------
figure(1); clf;

subplot(2,2,1);
hold on;
plot(Bon(:,1),      Bon(:,2));
plot(Cf,Rf);
legend('original','low-pass filtered','location','northwest');
plot(Bon(1,1),Bon(1,2),'*');        % starting pixel
axis equal;

subplot(2,2,2); hold on;
ArcLength   = [0; cumsum(Bon(:,3))];
plot(ArcLength,Drv2,'r');
plot(ArcLength,K,'g');
ylabel('Amplitude');
xlabel('Arc Length');
legend('2nd Deriv','K');
```

P.15 Image Retrieval

P.15.1 Feature Vector from Deep Net

Accessing the weights of the last *hidden* layer of a pretrained Deep Net using a hook. The weight values can be used as a feature vector for image retrieval (Section 18). For a ResNet that last hidden layer is called `avgpool`, for the Inception and the Vgg network they are called `Mixed_7c.branch_pool` and `classifier[3]`, respectively.

```
import torch
from torchvision import models
import torchvision.transforms as transforms
from skimage import data
from PIL import Image

Irgb     = data.astronaut()[0:200, 0:200]
Irgb     = Image.fromarray(Irgb)

# ======    Select Device (GPU | CPU)
device   = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# ======    Transform
ImgMean     = [0.485, 0.456, 0.406]  # mean    of ImageNet set
ImgStdv     = [0.229, 0.224, 0.225]  # std dev of ImageNet set
TrfImg   = transforms.Compose([transforms.Resize((256,256)),
                               transforms.ToTensor(),
                               transforms.Normalize(ImgMean, ImgStdv) ])

#%% %%%%%%%%%%%%%      NETWORK    %%%%%%%%%%%%%%%%%%%
NET     = models.resnet50(pretrained=True)  # pretrained weights
NET     = NET.to(device)                    # move to CUDA (or stay on CPU)
NET.eval()                                  # set to evaluation mode

#%% %%%%%%%%%%%%%      PREPARATION   %%%%%%%%%%%%%%%%%%%
# preparatory definition
hidImg = [None]
def hookHid(module, input, output):
    hidImg[0] = output

# instruct to store during forward
NET.avgpool.register_forward_hook(hookHid)              # resnet
# NET.Mixed_7c.branch_pool.register_forward_hook(f_HookHid) # inception
# NET.classifier[3].register_forward_hook(f_HookHid)    # vgg

#%% %%%%%%%%%%%%%      RUN   %%%%%%%%%%%%%%%%%%%
Img         = TrfImg(Irgb).to(device)           # transform the image
out         = NET(Img.unsqueeze(0))             # feed as batch using unsqueeze
Post        = hidImg[0].detach().numpy()[0,:,0,0]   # [2048,]
print(Post)
```

P.15.2 Bag of Words

```
Upcoming.
```

P.16 Tracking

P.16.1 Background Subtraction

Two examples of background subtraction, one for Matlab, one for OpenCV (through Python).

```
clear;
Vid.reader  = vision.VideoFileReader('atrium.avi');
%% --------------    Init Detection    --------------
ForeGnd     = vision.ForegroundDetector('NumGaussians', 3, ...
                'NumTrainingFrames', 40, 'MinimumBackgroundRatio', 0.7);
BlobAly     = vision.BlobAnalysis('BoundingBoxOutputPort', true, ...
                'AreaOutputPort', true, 'CentroidOutputPort', true, ...
                'MinimumBlobArea', 400);
%% --------------    Init Plotting    --------------
Play.Vido = vision.VideoPlayer('Position', [20, 400, 700, 400]);
Play.Mask = vision.VideoPlayer('Position', [740, 400, 700, 400]);

%% SSSSSSSSSSSSSSSSSSS    LOOP FRAMES     SSSSSSSSSSSSSSSSSSS
while ~isDone(Vid.reader)
    Frm      = Vid.reader.step();        % read one frame

    Msk         = ForeGnd.step(Frm);        % foreground [m n nCh]
    % remove noise and fill in holes
    Msk         = imopen(Msk,  strel('rectangle', [3,3]));
    Msk         = imclose(Msk, strel('rectangle', [15, 15]));
    Msk         = imfill(Msk, 'holes');
    [~, CenDet, BbxDet] = BlobAly.step(Msk);   % region analysis

    %% --------------   Plot    --------------
    % convert the frame and the mask to uint8 RGB.
    Frm     = im2uint8(Frm);
    Msk     = uint8(repmat(Msk, [1, 1, 3])) .* 255;
    % place rectangles around detections
Frm     = insertObjectAnnotation(Frm, 'rectangle', BbxDet, ' ');
  Msk     = insertObjectAnnotation(Msk, 'rectangle', BbxDet, ' ');
    % put on screen
    Play.Mask.step(Msk);
    Play.Vido.step(Frm);
end
```

```
import cv2 as cv

#%% ----------------    Video Info      -------------
vidName = 'C:/Program Files/MATLAB/MATLAB Production Server/R2015a/toolbox/vision/visiondata/atrium.avi'
Vid     = cv.VideoCapture(vidName)
Vid.set(cv.CAP_PROP_POS_MSEC, 2*1000)       # set starting time
nFrm    = Vid.get(cv.CAP_PROP_FRAME_COUNT)  # obtain total # of frames

#%% ---------------     Init Background & Detection     -------------
BckSub  = cv.createBackgroundSubtractorKNN()
StcElm  = cv.getStructuringElement(cv.MORPH_ELLIPSE,(3,3)) # for filtering

#%% ===============     LOOP FRAMES       ================
cF=0
while cF < 500:
    cF      += 1
    r,Frm   = Vid.read()          # reads one frame

    Msk  = BckSub.apply(Frm)      # generates a mask
    Msk  = cv.morphologyEx(Msk, cv.MORPH_OPEN, StcElm)  # filter for large patches

    # ----  Plot    -----
    cv.imshow('frame', Msk)
    k   = cv.waitKey(30) & 0xff

Vid.release()
```

```
cv.destroyAllWindows()
```

P.16.2 Maintaining Tracks

This example is a rewritten version of the example provided by Matlab (“Motion-Based Multiple Object Tracking”).

```
% depends on: predict, distance, correct, configureKalmanFilter and
%                       assignDetectionsToTracks
% Detected 'salient' regions are called objects here.
clear;
Vid.reader   = vision.VideoFileReader('atrium.avi');
%% --------------    Init Detection    --------------
Det.ForeGnd  = vision.ForegroundDetector('NumGaussians', 3, ...
    'NumTrainingFrames', 40, 'MinimumBackgroundRatio', 0.7);
Det.BlobAly  = vision.BlobAnalysis('BoundingBoxOutputPort', true, ...
    'AreaOutputPort', true, 'CentroidOutputPort', true, ...
    'MinimumBlobArea', 400);
% ---   more parameters:
costNonAss      = 20;  % cost of non-assignment
durMaxOff     = 20; % maximum off/invisible duration [frames]
durMinTot    = 8;    % minimum total duration        [frames]
% ---track structure:
StcTrk  = struct('id', [], 'bbox', [], ...          % id-number, bound-box
           'KalFlt', [], ...                        % Kalman filter
           'durTot', 1, 'durOnn', 1, 'durOff', 0); % durations
aTrk    = [];           % track list
cIdT    = 0;            % counter track identity
%% --------------    Init Plotting    --------------
durMinOnnPlot       = 8;
Play.Vido = vision.VideoPlayer('Position', [20, 400, 700, 400]);
Play.Mask = vision.VideoPlayer('Position', [740, 400, 700, 400]);

%% SSSSSSSSSSSSSSSSSSS    LOOP FRAMES     SSSSSSSSSSSSSSSSSSS
cF=0;
while ~isDone(Vid.reader)
    Frm      = Vid.reader.step();        % read one frame
    cF          = cF+1;
    %% SSSSSSSSSSSSSSSS DETECT      SSSSSSSSSSSSSSSSSSS
    Msk         = Det.ForeGnd.step(Frm); % foreground [m n nCh]
    % remove noise and fill in holes
    Msk  = imopen(Msk, strel('rectangle', [3,3]));
    Msk     = imclose(Msk, strel('rectangle', [15, 15]));
    Msk     = imfill(Msk, 'holes');
    [~, CenDet, BbxDet] = Det.BlobAly.step(Msk);   % region analysis

    %% SSSSSSSSSSSSSSSS TRACKS      SSSSSSSSSSSSSSSSSSS
    nTrk    = length(aTrk);
    %% =========    Predict Next Location    ==========
    for t = 1:nTrk
        bbox        = aTrk(t).bbox;
        cenPred     = predict(aTrk(t).KalFlt); % next location
        % shift bounding box to that prediction
        cenPred     = int32(cenPred) - bbox(3:4) / 2;
        aTrk(t).bbox = [cenPred, bbox(3:4)];
    end
    %% =========    Assign Detections to Tracks   ==========
    % cost matrix is quasi a distance matrix
    nDet    = size(CenDet,1);
    Cost    = zeros(nTrk, nDet);
    for t = 1:nTrk
        Cost(t,:)   = distance(aTrk(t).KalFlt, CenDet);
    end
    % solve the assignment problem
    [TrkAss TrkUna ObjUn] = assignDetectionsToTracks(Cost, costNonAss);
```

```
%% =========    Update Assigned Tracks    =========
for t = 1:size(TrkAss,1)
    ixT  = TrkAss(t,1);  % index track
    ixO     = TrkAss(t,2);  % index detection/object
    % replace predicted bbox with detected one
    aTrk(ixT).bbox  = BbxDet(ixO,:);
    % correct the estimate of the object's location
    % using the new detection.
    correct(aTrk(ixT).KalFlt, CenDet(ixO,:));

    aTrk(ixT).durTot = aTrk(ixT).durTot + 1; % duration (age)
    aTrk(ixT).durOnn  = aTrk(ixT).durOnn + 1; % duration visible
    aTrk(ixT).durOff    = 0;                    % reset to 0
end

%% =========    Update Unassigned   ========
for t = 1:length(TrkUna)
    ix    = TrkUna(t);
    aTrk(ix).durTot = aTrk(ix).durTot + 1;
    aTrk(ix).durOff = aTrk(ix).durOff + 1;
end

%% =========    Delete Lost/Unusual     ========
if ~isempty(aTrk)
    DurTot      = [aTrk(:).durTot]; % total duration    [nTrk 1]
    DurOnn      = [aTrk(:).durOnn]; % duration visible  [nTrk 1]
    DurOff      = [aTrk(:).durOff]; % duration invisible [nTrk 1]
    PropOn      = DurOnn ./ DurTot; % proportion visible/on
    % eliminate unusual/lost tracks:
    Blost = (DurTot<durMinTot & PropOn<0.6)  |  DurOff>=durMaxOff;
    aTrk(Blost) = [];               % delete lost tracks
end

%% =========    Create New     ========
CenNew  = CenDet(ObjUn,:);        % unassigned centers
BbxNew  = BbxDet(ObjUn,:);
% ======    Loop newly detected objects   ======
for i = 1:size(CenNew,1)
    KalFlt = configureKalmanFilter('ConstantVelocity', ...
        CenNew(i,:), [200, 50], [100, 25], 100);
    % create a new track & add to list
    cIdT        = cIdT + 1;     % increment the next id.
    newTrk      = StcTrk;
    newTrk.id   = cIdT;
    newTrk.bbox = BbxNew(i,:);
    newTrk.KalFlt = KalFlt;
    if cIdT==1,aTrk        = newTrk;
    else      aTrk(end+1) = newTrk;   end
end

%% --------------   Plot    --------------
% convert the frame and the mask to uint8 RGB.
Frm     = im2uint8(Frm);
Msk     = uint8(repmat(Msk, [1, 1, 3])) .* 255;
if ~isempty(aTrk)

    % Noisy detections tend to result in short-lived tracks.
    % Only display tracks that have been visible for more than
    % a minimum number of frames.
    IxReliable  = [aTrk(:).durOnn] > durMinOnnPlot;
    aTrkRel     = aTrk(IxReliable);

    % Display the objects. If an object has not been detected
    % in this frame, display its predicted bounding box.
    if ~isempty(aTrkRel)

        BbxRel  = cat(1, aTrkRel.bbox);
```

```
            ids     = int32([aTrkRel(:).id]);

            % Create tags for objects indicating the ones for
            % which we display the predicted rather than the actual
            % location.
            Tags    = cellstr(int2str(ids'));
            IxPred  = [aTrkRel(:).durOff] > 0;
            isPredicted = cell(size(Tags));
            isPredicted(IxPred) = {' predicted'};
            Tags    = strcat(Tags, isPredicted);

            Frm = insertObjectAnnotation(Frm, 'rectangle', BbxRel, Tags);
            Msk = insertObjectAnnotation(Msk, 'rectangle', BbxRel, Tags);
        end
    end
    % put on screen
    Play.Mask.step(Msk);
    Play.Vido.step(Frm);
end
```

P.16.3 Face Tracking with CAM shift

This code example is a rewritten version of the example provided by Matlab ("Face Detection and Tracking Using CAMShift").

```
% Face tracking using CAMshift with histograms. The exact object to be
% tracked is the nose, because it does not contain any (distracting) back-
% ground pixels - it makes tracking more reliable. The hue channel of the
% HSV space is used, as it represents skin tones better (tendentially).
clear;
%% =========    Detect the face in the 1st frame    =======
DetFace         = vision.CascadeObjectDetector('FrontalFaceCART');
VidReader       = vision.VideoFileReader('visionface.avi');
Frm             = step(VidReader);              % a single frame
BboxFace        = step(DetFace, Frm);           % bounding box [1 4]
DetNose         = vision.CascadeObjectDetector('Nose', 'UseROI', true);
BboxNose        = step(DetNose, Frm, BboxFace(1,:));
% ----  Plot  ----
figure(1); imshow(Frm);
rectangle('Position',BboxFace(1,:), 'EdgeColor',[1 1 0])
rectangle('Position',BboxNose(1,:), 'EdgeColor',[1 0 0])
pause(.5);
%% --------     Init tracker  ---------
Trk             = vision.HistogramBasedTracker; % init tracker object
[Mhue,~,~]      = rgb2hsv(Frm);
initializeObject(Trk, Mhue, BboxNose(1,:));  % hue pixs from nose
% ----  Plot  ----
figure(2); clf; imshow(Mhue); title('Hue Channel Map');
rectangle('Position',BboxFace(1,:), 'EdgeColor',[1 1 0])
pause(.5);
%% ========     Tracking the face   ===========
% ----  Init plotting
VidInfo     = info(VidReader);
VidPlayer   = vision.VideoPlayer('Position',[300 300 VidInfo.VideoSize+30]);
% ----  Tracking
while ~isDone(VidReader)
    Frm         = step(VidReader);
    [Mhue,~,~]  = rgb2hsv(Frm);
    BboxFnow    = step(Trk, Mhue);      % new face bounding box
    % ----  Plot    ----
    Fann        = insertObjectAnnotation(Frm,'rectangle', BboxFnow, 'Face');
    step(VidPlayer, Fann);
end
% ---- release objects
release(VidReader);
```

```
release(VidPlayer);
```

P.16.4 Tracking-by-Matching

```
import cv2 as cv

#%% ------------    Video Prep      -------------
vidPth  = 'C:/Program Files/MATLAB/MATLAB Production Server/R2015a/toolbox/vision/visiondata/'
vidName = 'handshake_left.avi'
Vid     = cv.VideoCapture(vidPth + vidName)

#%% -----------     Find 1st BoundingBox   -------------
ret,Frm     = Vid.read()
PedDet      = cv.HOGDescriptor()
PedDet.setSVMDetector(cv.HOGDescriptor_getDefaultPeopleDetector())
(aPers, Like) = PedDet.detectMultiScale(Frm, winStride=(2,2),
                            padding=(8,8), scale=1.05)
x1,y1,w,h   = aPers[0]
bbox        = (x1,y1,w,h)  # upper left point, width, height

#%% -----------     Init Tracker    -------------
Trkr    = cv.TrackerKCF_create()
bOk     = Trkr.init(Frm, bbox)

#%% ===============     TRACK       ================
cF=1
while cF < 55:
    cF      += 1
    ret,Frm  = Vid.read()
    if ret==False: break

    bOk, bbox = Trkr.update(Frm)

    # ----  Plot    -----
    p1 = (int(bbox[0]), int(bbox[1]))
    p2 = (int(bbox[0] + bbox[2]), int(bbox[1] + bbox[3]))
    Idem = cv.rectangle(Frm, p1, p2, 255,2)
    cv.imshow('Mask',Idem)
    k = cv.waitKey(60) & 0xff
    if k == 27:
        break

cv.destroyAllWindows()
Vid.release()
```

P.16.5 Kalman Filter - The Smoothening Property

Section 19.5.1

The smoothening characteristic of the Kalman filter: the signal has a constant value, `x=1`, but is measured with some fluctuation simulated as `det = x + 0.1*randn`.

```
% Rewritten from Matlab's example on 'Remove Noise from a Signal' under
% the document page for vision.KalmanFilter.
clear;
%% -----    Synthetic Stimulus    ------
x       = 1;            % the signals theoretical (constant) value
nSmp    = 100;          % number of samples

%% =====    Init Tracker        ======
ModState  = 1;
ModMesrm    = 1;
Klm     = vision.KalmanFilter(ModState, ModMesrm, ...
    'StateCovariance',1, 'ProcessNoise',1e-5, 'MeasurementNoise',1e-2);
```

```
%% =====    Track   =======
Trk     = zeros(1,nSmp);            % track prediction
Det     = zeros(1,nSmp);            % detections
for i = 1:nSmp
    predict(Klm);                   % predict next location
    det     = x + 0.1 * randn;      % measure actual location
    Trk(i)  = correct(Klm, det);    % adjust prediction
    Det(i)  = det;
end

%% -----    Prepare Plot    -----
figure(1);clf;
plot(x*ones(1,nSmp), 'g-'); hold on;
plot(1:nSmp,Det,'b+', 1:nSmp,Trk,'r-');
legend('Theoretical signal', 'Detected (Noisy) signal', 'Tracked (Filtered) signal');
```

P.16.6 Kalman Filter - The Prediction Property

Section 19.5.1

The prediction property of the Kalman filter: the signal is a one-dimensional stimulus, that occasionally is not detected by the tracker.

```
% Rewritten from Matlab's example on object tracking using a synthetic
% motion stimulus (under vision.KalmanFilter).
clear;
%% =====    Synthetic Stimulus    =====
nLoc        = 40;
LocDet      = num2cell(2*randn(1,nLoc) + (1:nLoc)); % random vector
LocDet{1}   = [];           % make 1st one missing
for i = 16:25,              % make 16th to 25th missing too
    LocDet{i} = []; end
%% =====    Init Tracker        ======
ModState = [1 1; 0 1];      % model state
ModMesrm = [1 0];           % model measurement
%% -----    Prepare Plot    -----
figure(1); clf; hold on;
ylabel('Location');     ylim([0 50]);
xlabel('Time');         xlim([0 nLoc]);
%% =====    Track   =======
Klm = [];
for i = 1:nLoc
    locDet  = LocDet{i};            % detected/measured location
    isDet   = ~isempty(locDet);     % is detected (not missed)
    if isempty(Klm)
        if isDet
            Klm = vision.KalmanFilter(ModState, ModMesrm, ...
                    'ProcessNoise',1e-4, 'MeasurementNoise',4);
            Klm.State = [locDet 0];         % set initial state
        end
    else
        locTrk  = predict(Klm);                 % PREDICT
        if isDet
            plot(i, locDet,'k+');
            dis     = distance(Klm, locDet);    % error
            title(sprintf('Distance:%f', dis));
            locTrk  = correct(Klm, locDet);     % CORRECT
        else
            title('Missing detection');
        end
        pause(0.2);
        plot(i,locTrk,'ro');                % plot predicted/corrected
    end
 end
legend('Detected','Predicted/Corrected');
```

P.16.7 Condensation (Particle Filter)

Section 19.5.2

A simple example for face tracking.

```
clear;
nPart       = 1000;             % # of particles
rgbT        = [255 20 100]';    % red is our target
noiseRgb    = 50;               % measurement noise
noisePos    = 25;               % sigma value for position variability
noiseMot    = 5;                % sigma value for motion variability
a       = -log(sqrt(2*pi) * noiseRgb);
b       = - 0.5 / (noiseRgb.^2);

%% -----    Load Movie  ------
Vid=VideoReader('C:\Program Files\MATLAB\MATLAB Production Server\R2015a\toolbox\vision\visiondata\tilted_face.avi');
szI=[Vid.Height Vid.Width];   nFrm=floor(Vid.Duration * Vid.FrameRate);

%% =====    Init Tracker    =====
Rw      = randi(szI(1), 1, nPart);  % random row index/coordinate
Cl      = randi(szI(2), 1, nPart);  % random col index/coordinate
POS     = [Rw; Cl];                 % random positions initially
MOT     = zeros(2, nPart);          % motion vector

%% TTTTTTTTTTTTTTTT    Track   TTTTTTTTTTTTTTTTTTTT
figure(1);clf;
for f = 1:nFrm

    Frm     = readFrame(Vid);           % obtain a frame [szI 3]
    FrmPrm  = permute(Frm, [3 1 2]);

    %% ======    Drift      =======
    POS         = POS + MOT;
    POS         = POS + randn(2,nPart) * noisePos;
    MOT         = MOT + randn(2,nPart) * noiseMot;

    %% ======   Probability      =======
    % calculated only for position within image
    PSR     = round(POS);
    BrowIn  = PSR(1,:)>=1 & PSR(1,:)<=szI(1);   % within image height
    BcolIn  = PSR(2,:)>=1 & PSR(2,:)<=szI(2);   % within image width
    Bins    = BrowIn & BcolIn;                  % inside image boundaries
    LL      = -inf(1,nPart);                    % log-likelihood
    for k = find(Bins)
        r       = PSR(1,k);
        c       = PSR(2,k);
        rgbM    = double(FrmPrm(:,r,c));    % [3 1] rgb values measured
        d       = sum( (rgbM-rgbT).^2 );    % distance (measured-target)
        LL(k)   = a + b * d;                % log-likelihood
    end
    L       = exp(LL - max(LL));        % E [0..1]
    Q       = L / sum(L);               % normalized to 1, E [0..small]

    %% ======    Splitting      =======
    R       = cumsum(Q,2);              % E [0..1], large step = high probability
    T       = rand(1,nPart);            % random numbers
    [H,~,Ix2Bin] = histcounts(T,R);     % R=edges
    POS     = POS(:,Ix2Bin+1);
    MOT     = MOT(:,Ix2Bin+1);

    %% ---   Plot    ----
    figure(1);
    imagesc(Frm); hold on;
    plot(POS(2,:), POS(1,:), '.');
    plot(mean(POS(2,:)), mean(POS(1,:)), 'o', 'MarkerSize', 8, 'MarkerEdgeColor', 'y', 'MarkerFaceColor', 'y')
    hold off; drawnow

end
```

P.17 Optic Flow


```
clear;
Vid     = VideoReader('viptraffic.avi');
%% -----    3 Methods:
%OptFlow = opticalFlowHS;
%OptFlow = opticalFlowLK('NoiseThreshold',0.009);
OptFlow = opticalFlowLKDoG('NumFrames',3);

%% SSSSSSSSSSSSS   LOOP FRAMES     SSSSSSSSSSSSSS
while hasFrame(Vid)
    Frgb    = readFrame(Vid);
    Fgry    = rgb2gray(Frgb);
    Flow    = estimateFlow(OptFlow, Fgry);

    imshow(Frgb)
    hold on
    plot(Flow, 'DecimationFactor',[5 5], 'ScaleFactor',25)
    hold off

end
```

```
import numpy as np
import cv2 as cv
#%% ------------    Prepare Video      ------------
vidName = 'someVideo.mp4'
Vid     = cv.VideoCapture(vidName)
Vid.set(cv.CAP_PROP_POS_MSEC, 40*1000)  # set to appropriate starting time

#%% ------------    Parameters      ------------
# corner detection, ShiTomasi technique
PrmCorDet    = dict(maxCorners = 500,
                    qualityLevel = 0.5,
                    minDistance = 7,
                    blockSize = 7 )
# optic flow, LucasKanade method
PrmOptFlo   = dict(winSize  = (15,15),
                   maxLevel = 2,
                   criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 10, 0.03))

#%% ============    1st frame ============
rt, Fold = Vid.read()       # we call the frame old already
Iold     = cv.cvtColor(Fold, cv.COLOR_BGR2GRAY)
Pts0     = cv.goodFeaturesToTrack(Iold, mask = None, **PrmCorDet)

# --- prepare plotting:
Msk     = np.zeros_like(Fold)
Cols    = np.random.randint(0,255,(100,3))  # list of colors
#%% SSSSSSSSSSSSSSSSSSS    LOOP FRAMES     SSSSSSSSSSSSSSSSSSS
while(1):
    rt,Fnew = Vid.read()        # new frame
    Inew    = cv.cvtColor(Fnew, cv.COLOR_BGR2GRAY)

    # === Calculate optical flow ===
    Pts1, St, Err = \
        cv.calcOpticalFlowPyrLK(Iold, Inew, Pts0, None, **PrmOptFlo)
    # === Select good points (status==1) ===
    PtsNew = Pts1[St==1]
    PtsOld = Pts0[St==1]
    # === Update ===
    Iold = Inew.copy()  # new (gray-scale) frame becomes old one
    Pts0 = PtsNew.reshape(-1,1,2)

    # --------   Plot  --------
    # draw the tracks
    for i,(new,old) in enumerate(zip(PtsNew, PtsOld)):
```

```
        a,b  = new.ravel()
        c,d  = old.ravel()
        Msk  = cv.line(Msk, (a,b),(c,d), Cols[i].tolist(), 2)
        Fnew = cv.circle(Fnew,(a,b),5, Cols[i].tolist(), -1)
    Idisp = cv.add(Fnew, Msk)
    cv.imshow('frame',Idisp)
    # keystroke input
    k = cv.waitKey(30) & 0xff
    if k == 27:
        break

cv.destroyAllWindows()
Vid.release()
```

P.18 Geometric Transformations, 2D

We firstly show how to move a shape or image using matrix multiplication (Sections **P.18.1** and **P.18.2**). Then we estimate motions (Section **P.18.3**).

P.18.1 Moving a Set of Points

An example for translation. It shows three implementations: one for notation 1, another for notation 2, and the third, the use of the Matlab functions (Section 21.1). More transformation types are given in the subsequent section.

```
clear;
%% ------   Stimulus: a distorted rectangle
Pts     = [0 0; 0 .8; 0.6 0.75; 1.2 .8; 1.0 0.9; 1.2 0; 0 0]*4+2;
np      = size(Pts,1);       % # of points

% --- choosing some motion parameters
tx  = 0.5;              % x-translation
ty  = 1.5;              % y-translation

%% =====    Transformation Notation 1   =====
Pno1(:,1) = Pts(:,1)+tx;
Pno1(:,2) = Pts(:,2)+ty;

%% =====    Transformation Notation 2   =====
% ====  transformation matrix:
Ttrs    = [1 0 tx; 0 1 ty];
% ====  perform transformations
% we subtract center point from shape and add it later again
cen     = mean(Pts,1);          % center point of shape
Cen     = repmat(cen,np,1);  % replicate center point
% transformation matrix is transposed to match column-wise coordinates:
Pno2    = [Pts-Cen ones(np,1)] * Ttrs' + Cen;

%% =====    Transformation Matlab   =====
Taff    = affine2d([Ttrs' [0 0 1]']);
Paff    = Taff.transformPointsForward(Pts);
Tprj    = projective2d([Ttrs' [0 0 1]']);
Pprj    = Taff.transformPointsForward(Pts);

%% ------   Verify      ------
assert(all(Pno1(:)==Pno2(:)));
assert(all(Pno1(:)==Paff(:)));
assert(all(Pno1(:)==Pprj(:)));

%% -----------      Plotting        -----------
figure(1); clf; hold on;
plot(Pts(:,1), Pts(:,2), 'color', 'k', 'linewidth', 2);
plot(Pno1(:,1), Pno1(:,2), 'color', 'g');
legend('original', 'translation', 'location', 'northwest');
set(gca, 'xlim', [0 8], 'ylim', [0 8]);
```

In Python the function `matrix_transform` takes the homogeneous matrix only, called `H` here.

```
import numpy as np
import numpy.matlib
from skimage.transform import matrix_transform

#%% ------   Stimulus: a distorted rectangle
Pts     = np.array([[0,0],[0,.8],[0.6,0.75],[1.2,.8],[1.0,0.9],[1.2,0],[0,0]])*4+2
nP      = Pts.shape[0]       # # of points

# --- choosing some motion parameters
tx  = 0.5;              # x-translation
ty  = 1.5;              # y-translation
```

```
#%% =====    Transformation Notation 1   =====
Pno1        = np.zeros((nP,2))
Pno1[:,0]   = Pts[:,0]+tx;
Pno1[:,1]   = Pts[:,1]+ty;

#%% =====    Transformation Notation 2   =====
# ====  transformation matrix:
Ttrs    = np.array([[1,0,tx],[0,1,ty]])
# ====  perform transformations
# we subtract center point from shape and add it later again
cen     = Pts.mean(axis=0)          # center point of shape
Cen     = np.matlib.repmat(cen,nP,1);  # replicate center point
# transformation matrix is transposed to match column-wise coordinates:
Pno2    = np.hstack((Pts-Cen,np.ones((nP,1)))) @ Ttrs.transpose() + Cen

#%% =====    Transformation Skimage   =====
H       = np.vstack((Ttrs,np.array([0,0,1])))   # homogeneous transformation mx
Paff    = matrix_transform(Pts, H)

#%% ------   Verify      ------
assert all(Pno1.flatten()==Pno2.flatten())
assert all(Pno1.flatten()==Paff.flatten())

#%% -----------      Plot        -----------
from matplotlib import *
figure
plot(Pts[:,0], Pts[:,1],'k') # 'color', 'k', 'linewidth', 2);
plot(Pno1[:,0], Pno1[:,1], 'g')
```

P.18.2 Moving (Warping) an Image

Some examples of image manipulations useful for augmenting an image data base. We specify a homogeneous (3x3) matrix for reason of convenience.

```
clear;

Igry    = imread('cameraman.tif');

theta   = pi/2 * 0.102; % 9.8039 degree
thC     = cos(theta);   thS = sin(theta);
s       = 0.148;        % shear proportion (both sides)
d       = .0005;        % distortion unilateral

%% =====    Build Transformation Matrix     =====
TshH    = projective2d([1 0 0;     s  1 0;     0 0 1]);  % horiz shear
TshV    = projective2d([1 s 0;    0  1 0;     0 0 1]);  % vertical shear

TsVH    = projective2d([1 s  0;   s  1  0;  0 0 1]);  % h & v shear
TsVL    = projective2d([1 0 -d;   0  1  0;  0 0 1]);
TsHR    = projective2d([1 0  0;    0  1 -d;  0 0 1]);
TsHRVL  = projective2d([1 0 -d/2; 0  1 -d/2; 0 0 1]);

Trot    = projective2d([thC -thS 0; thS thC 0;  0 0 1]);
Trsh    = projective2d([thC -thS d; thS thC d;  0 0 1]);

%% =====    Apply Transformation Matrix     =====
IshH  = imwarp(Igry,TshH);
IshV  = imwarp(Igry,TshV);

IsVH  = imwarp(Igry,TsVH);
IsVL  = imwarp(Igry,TsVL);
IsHR  = imwarp(Igry,TsHR);
IsHRVL  = imwarp(Igry,TsHRVL);

Irot  = imwarp(Igry,Trot);
```

```
Irsh  = imwarp(Igry,Trsh);

%% -----    Plot    ------
[nr nc] = deal(3,3);
figure(1); clf;
subplot(nr,nc,1); imagesc(Igry); title(sprintf('original, [%d %d]', size(Igry,1), size(Igry,2)));
subplot(nr,nc,2); imagesc(IshH); title(sprintf('shear h, [%d %d]', size(IshH,1), size(IshH,2)));
subplot(nr,nc,3); imagesc(IshV); title(sprintf('shear v, [%d %d]', size(IshV,1), size(IshV,2)));

subplot(nr,nc,4); imagesc(IsVH); title(sprintf('h & v shear, [%d %d]', size(IsVH,1), size(IsVH,2)));
subplot(nr,nc,5); imagesc(IsVL); title(sprintf('sh v low [%d %d]', size(IsVL,1), size(IsVL,2)));
subplot(nr,nc,6); imagesc(IsHR); title(sprintf('sh h right [%d %d]', size(IsHR,1), size(IsHR,2)));

subplot(nr,nc,7); imagesc(IsHRVL); title(sprintf('shear hrvl [%d %d]', size(IsHRVL,1), size(IsHRVL,2)));
subplot(nr,nc,8); imagesc(Irot); title(sprintf('rot [%d %d]', size(Irot,1), size(Irot,2)));
subplot(nr,nc,9); imagesc(Irsh); title(sprintf('rot & shear [%d %d]', size(Irsh,1), size(Irsh,2)));
```

P.18.3 Estimating Motions

The script has two sections. In the first section, 'Transforming the Shape', a shape is transformed in various ways to simulate the corresponding types of motion (Section 21.1); in a second section, 'Estimate Motion', those various transformations are then estimated (Section 21.2).

Transforming the Shape: as shape we chose a distorted (self-intersecting) rectangle, specified with coordinates in `Pts`. Its transformed points are called `Prot`, `Pshr`, etc.

Estimate Motion: the simulated motion is then estimated multiple times, once with a similarity and once with an affinity transform, for which we build the corresponding variables `Psim`, `Paff` and `bsim` and `baff`, respectively. The estimation procedure is done with a minimization technique such as `lsqlin` or `lsqnonneg`, whereby $\mathbf{A}$ and $\mathbf{b}$ are passed as arguments.

At the very end, we carry out some of those transformations with function scripts that perform all in one, for instance `fitgeotrans` and `procrustes`.

```
% Examples of 2D transforms and their motion estimation. Sze pdf 36, 312.
clear;
%% ------   Stimulus: a distorted rectangle
Pts = [0 0; 0 .8; 0.6 0.75; 1.2 .8; 1.0 0.9; 1.2 0; 0 0]*4+2;
np = size(Pts,1);       % # of points
cpt = mean(Pts,1);       % center point

%% SSSSSSSSSSSSSSSSS   Transforming the Shape     SSSSSSSSSSSSSSSSS
% --- choosing some parameters
r   = 0.25;             % rotation/scale
tx  = 0.5;              % x-translation
ty  = 1.5;              % y-translation
a   = 0.25;             % scale (for similarity)
b   = 0.25;             % rotation (for similarity)
[a00 a01 a10 a11] = deal(-0.5, 0.25, 0.125, -0.75); % for affinity
% -----------preparing transformation matrices     ------------
TRot    = [cos(r) -sin(r); sin(r) cos(r)]; % clockwise rotation
TEuc    = [TRot [tx ty]'];          % Euclidean (rot&trans) aka rigid body motion
TEuc0   = [TEuc; [0 0 1]];          % with row extended
TShr    = [1 0; 0.5 1];             % shear
TScl    = [r 0; 0 r];               % scaling
TSim    = [1+a -b tx; b 1+a ty];    % similarity
TSim0   = [TSim; 0 0 1];            % with row extended
TAff    = [1+a00 a01 tx; a10 1+a11 ty; 0 0 1]; % affinity (with row extended)
% -----------perform transformations     ------------
Cen = repmat(cpt,np,1);                 % replicate center point
% transf. mx are transposed to match column-wise coordinates:
Poc     = Pts-Cen;                       % subtract cenpt: coordinates 0,0 centered
% --- rot, shear, scale:
Prot    = Poc * TRot' + Cen;            % rot   transformation and add cenpt
```

```
Pshr    = Poc * TShr' + Cen;            % shear         "
Pscl    = Poc * TScl' + Cen;            % scale         "
% --- aff/euc/sim: add row of 1s, then transform
Paff    = [Poc ones(np,1)] * TAff';     % affinity      "       here we add a col of 1s
Peuc    = [Poc ones(np,1)] * TEuc0';    % Euclidean     "       here we add a col of 1s
Psim    = [Poc ones(np,1)] * TSim0';    % similarity    "     here we add a col of 1s
Paff(:,[1 2]) = Paff(:,[1 2])+Cen;      % THEN add cenpt
Peuc(:,[1 2]) = Peuc(:,[1 2])+Cen;
Psim(:,[1 2]) = Psim(:,[1 2])+Cen;

%% -----------      Plotting        -----------
figure(1); clf; [rr cc] = deal(1,1); hold on;
plot(Pts(:,1), Pts(:,2), 'color', 'k', 'linewidth', 2);
plot(Prot(:,1), Prot(:,2), 'color', 'g');
plot(Pshr(:,1), Pshr(:,2), 'color', 'r');
plot(Pscl(:,1), Pscl(:,2), 'color', 'b');
plot(Paff(:,1), Paff(:,2), 'color', 'm');
plot(Peuc(:,1), Peuc(:,2), 'color', 'c');
plot(Psim(:,1), Psim(:,2), 'color', ones(3,1)*.5);
axis equal
legend('original', 'rotation', 'shear', 'scaling', 'affine', 'rigid', 'similar',...
'location', 'northwest');
set(gca, 'xlim', [0 8], 'ylim', [0 8]);

%% SSSSSSSSSSSSSSSSS   Estimate Motion   SSSSSSSSSSSSSSSSS
%% =========    Building A and b    =========
JSim    = @(x,y)([1 0 x -y; 0 1 y x]);
JAff    = @(x,y)([1 0 x y 0 0; 0 1 0 0 x y]);
% JEuc = inline('[1 0 -sin(r)*x -cos(r)*y; 0 1 -cos(r)*x -sin(r)*y]');
Asim    = zeros(4,4); bsim = 0;     % init A and b for similarity case
Aaff    = zeros(6,6); baff = 0;     % init A and b for affinity case
DltSim  = Psim(:,[1 2])-Poc;        % delta (transformed-original 0,0 centered)
DltAff  = Paff(:,[1 2])-Poc;        % delta (transformed-original 0,0 centered)
for i = 1 : np
    pt      = Poc(i,:);           % original 0,0 centered
    Jp      = JSim(pt(1),pt(2));
    Asim    = Asim + Jp'*Jp;
    bsim    = bsim + DltSim(i,:)*Jp;
    Jp      = JAff(pt(1),pt(2));
    Aaff   = Aaff + Jp'*Jp;
    baff    = baff + DltAff(i,:)*Jp;
end
%% =========    Least-Square for A and b for Similarity     =========
disp('Similarity');
[Prm1 resn1] = lsqnonneg(Asim, bsim'); % Prm1(3:4) contain estimates for a and b
[Prm2 resn2] = lsqlin(Asim, bsim');
Prm1'
Prm2'
(Prm1(1:2)-cpt')'       % translation parameters (tx, ty)
%% =========    Least-Square for A and b for Affinity   =========
disp('Affinity');
[Prm resn] = lsqlin(Aaff, baff'); % Prm(3:6) contain estimates for a00, a01, a10, a11
Prm'
(Prm(1:2)-cpt')'        % tx and ty

%% ---------    FitGeoTransform         ----------
Tffit               = fitgeotrans(Pts,Psim(:,1:2),'similarity');
Tffit.T-TSim0'

%% ---------    Applying Procrustes     ----------
[d PtProc Tproc]    = procrustes(Pts, Psim(:,1:2));

figure(2); clf; hold on;
plot(Pts(:,1), Pts(:,2), 'color', 'k', 'linewidth', 2);
plot(Psim(:,1), Psim(:,2), 'color', 'm');
plot(PtProc(:,1), PtProc(:,2), 'r');
```

P.19 RanSAC

Example of a primitve version of the Random Sampling Consensus algorithm (Section 21.3). Below are two scripts: a function script `f_RanSaC` and a testing script that runs the function.

```
% Random Sampling Consensus for affine transformation.
% No correspondence determined - assumes list entries correspond already.
% IN:   - Pt1 list of original    points [np1,2]
%       - Pt2 list of transformed points [np2,2]
%       - Opt options
% OUT:  - TP    struct with estimates
%           - PrmEst    [nGoodFits x 6] parameters from lsqlin
%           - ErrEst    [nGoodFits x 1] error
function TP = f_RanSaC(Pt1, Pt2, Opt, b_plot)
TP.nGoodFits = 0;
if isempty(Pt1) || isempty(Pt2),  return;  end
JAff        = inline('[1 0 x y 0 0;   0 1 0 0 x y]');
np1         = size(Pt1,1);
np2         = size(Pt2,1);
fprintf('Original has %d points, transformed has %d points\n', np1, np2);
nMinPt      = Opt.nMinPts;              % # of minimum pts required for transformation
nCom        = np1-nMinPt;               % # of complement pts
if ~nCom,  warning('no complementing points');  end
cpt         = mean(Pt1);                % center point of original
Pt1Cen      = Pt1 - repmat(cpt,np1,1);  % original 0,0 centered

%% ====================    Iterating    ====================
cIter       = 0;
nGoodFits = 0;
[Prm Err]   = deal([]);
while cIter < Opt.nMaxIter

    %% ------ Random Subset (Samples)
    Ixrs    = randsample(np1, nMinPt);              % random sample indices
    Ixcm    = setdiff(1:np1, Ixrs);                 % complement

    %% ==========   Estimation with Random Samples
    Dlt     = Pt2(Ixrs,:)-Pt1Cen(Ixrs,:);   % delta (transformed-original 0,0 centered)
    Atot    = zeros(6,6);  b = 0;           % init Atot and b
    for i = 1:nMinPt
        pt      = Pt1Cen(Ixrs(i),:);
        Jp      = JAff(pt(1),pt(2));
        Atot    = Atot + Jp'*Jp;                    % building A
        b       = b + Dlt(i,:)*Jp;                  % building b
    end
    [prm err]   = lsqlin(double(Atot), double(b'));     % prm(3:6) contain a00,..
    [tx ty]     = deal(prm(1)-cpt(1), prm(2)-cpt(2));   % tx and ty
    fprintf('tx %1.4f  ty %1.4f ', tx, ty);

    %% ==========   Transform Complement
    [a00 a01 a10 a11] = deal(prm(3), prm(4), prm(5), prm(6));
    TAff        = [1+a00 a01 tx;  a10 1+a11 ty;  0 0 1]; % affinity (with row extended)
    Pt1Com      = Pt1Cen(Ixcm,:);                   % the complement pts
    Caff        = [Pt1Com ones(nCom,1)] * TAff';    % we add a col of 1s
    Caff(:,1:2) = Caff(:,1:2)+repmat(cpt,nCom,1);   % THEN add cenpt

    %% ==========   Close enough?
    DfCo    = Caff(:,1:2)-Pt2(Ixcm,:);              % [nCom,2]
    Di      = sqrt(sum(DfCo.^2,2));
    bNear   = Di < Opt.thrNear;
    nNear   = nnz(bNear);
    fprintf('%3d near of %4d  err %12.10f', nNear, nCom, err);
    if nNear >= Opt.nNear
        nGoodFits = nGoodFits + 1;
        % =========   Refit   =========
        Pt2n    = Pt2(Ixcm(bNear),:);
        Pt1Cenn = Pt1Cen(Ixcm(bNear),:);
        Dltn    = Pt2n - Pt1Cenn;
        % --- building A and b
        A2      = zeros(6,6);  b = 0;           % init Atot and b
        for i = 1:nNear
            pt      = Pt1Cenn(i,:);
            Jp      = JAff(pt(1),pt(2));
            A2      = A2 + Jp'*Jp;              % building A
            b       = b  + Dltn(i,:)*Jp;        % building b
        end
        % --- least squares:
        [prm2 err2] = lsqlin(double(A2), double(b'));
        [tx ty]     = deal(prm(1)-cpt(1), prm(2)-cpt(2));   % tx and ty
        Prm         = [Prm; [tx ty] prm2(3:end)'];
        Err         = [Err; err2];
    end

    %% ==========   Plotting
    if b_plot.dist
        Dis = sort(Di, 'ascend');
        figure(10); clf; [rr cc] = deal(1,2);
        subplot(rr,cc,1);
            plot(Dis);
            set(gca, 'ylim', [0 1500]);
            title([Opt.xlb ' ' num2str(cIter)]);
            xlabel(Opt.Match, 'fontweight', 'bold', 'fontsize', 12);
        subplot(rr,cc,2);
            plot(Dis); hold on;
            set(gca, 'xlim', [0 nCom*0.25]);
            set(gca, 'ylim', [0 Opt.thrNear*2]);
            plot([0 nCom], ones(1,2)*Opt.thrNear, 'k:');
        pause();
    end

    cIter   = cIter + 1;
    fprintf('\n');
end
TP.PrmEst       = Prm;
```

```
TP.ErrEst       = Err;
TP.nGoodFits    = nGoodFits;
fprintf('#GoodFits %d out of %d iterations\n', nGoodFits, Opt.nMaxIter);
end % function
```

This is the testing script calling the above function:

```
%% ==================== TESTING SCRIPT t_Ransac
% Testing Ransac.
clear;
nP      = 40;
%% Simple Pattern
CoOrig  = rand(nP,2)*3+5;
np      = size(CoOrig,1);       % # of points
cpt     = mean(CoOrig,1);       % center point
Cpt     = repmat(cpt,np,1);     % replicated center point

%% ==================   Transform Original   ==================
tx  = 4;          % x-translation
ty  = 1.5;        % y-translation
a   = 0.25;        % scale     (for similarity)
b   = 0.35;       % rotation  (for similarity)
% -------------- Transformation Matrices
TSim    = [1+a -b tx; b 1+a ty];    % similarity
TSim0   = [TSim; 0 0 1];            % with row extended
% -------------- Perform Transformations
Coc     = CoOrig-Cpt;                   % subtract cenpt: coordinates 0,0 centered
Csim    = [Coc ones(np,1)] * TSim0';    % similarity    "     here we add a col of 1s
Csim(:,[1 2]) = Csim(:,[1 2])+Cpt;
CoTrns  = Csim(:,[1 2]) + rand(nP,2)*1.2;   % adding some noise

%% ==================   Plotting   ==================
figure(1); clf; hold on;
plot(CoOrig(:,1), CoOrig(:,2), 'g.');  % original    set of points
plot(CoTrns(:,1), CoTrns(:,2), 'r.');  % transformed set of points
axis equal
legend('original', 'transformed', 'location', 'northwest');

%% ==================   RanSaC   ==================
disp('Verifying f_RanSaC');
Opt.nMinPts     = 6;
Opt.nMaxIter    = 10;
Opt.thrNear     = 0.2;
Opt.nNear       = 3;
Opt.xlb         = '';
Opt.Match       = '';
b_plot.dist     = 1;
RSprm           = f_RanSaC(CoOrig, CoTrns, Opt, b_plot);
RSprm'
RSprm.PrmEst
```

Python offers a function in `skimage.measure.ransac`

P.20 Stereo Correspondence

Section 22.1.1

Based on local-feature correspondence.

```
% Rewritten Matlab example 'Sparse 3-D Reconstruction From Two Views'
clear; clear all;

dirImg  = fullfile(toolboxdir('vision'), 'visiondata','sparseReconstructionImages');
I1      = imread(fullfile(dirImg, 'Globe01.jpg'));
I2      = imread(fullfile(dirImg, 'Globe02.jpg'));
n0      = size(I1,2);

figure(1); subplot(2,2,1);
imshowpair(I1, I2, 'montage'); title('Originals');

%% ------   Load precomputed camera parameters  -----
load sparseReconstructionCameraParameters.mat   % loads cameraParams

%% ------   Remove Lens Distortion   -----
[I1, Org1new] = undistortImage(I1, cameraParams, 'OutputView', 'full');
[I2, Org2new]   = undistortImage(I2, cameraParams, 'OutputView', 'full');

figure(1); subplot(2,2,2);
imshowpair(I1, I2, 'montage'); title('Undistorted');

%% ======   CheckerBoard Reference   =======
szSqu       = 22;       % checkerboard square size in millimeters
[Pts1Ref, szBoard1] = detectCheckerboardPoints(I1);
[Pts2Ref, szBoard2] = detectCheckerboardPoints(I2);
% Translate detected points back into the original image coordinates
Pts1Ref   = bsxfun(@plus, Pts1Ref, Org1new);
Pts2Ref   = bsxfun(@plus, Pts2Ref, Org2new);

subplot(2,2,1); hold on;  % plot into original images
plot(Pts1Ref(:,1),      Pts1Ref(:,2));
plot(Pts2Ref(:,1)+n0,   Pts2Ref(:,2));

PtsWorld   = generateCheckerboardPoints(szBoard1, szSqu); % [54 2]
[R1, t1]   = extrinsics(Pts1Ref, PtsWorld, cameraParams);
[R2, t2]   = extrinsics(Pts2Ref, PtsWorld, cameraParams);

% calculate camera matrices using the |cameraMatrix| function.
MxCam1     = cameraMatrix(cameraParams, R1, t1);
MxCam2    = cameraMatrix(cameraParams, R2, t2);
% camera positions for plotting:
PosCam1     = -t1 * R1';    % x/y/z-coordinates
PosCam2     = -t2 * R2';

%% =====    Local Feature Correspondence  =====
aPts1Surf  = detectSURFFeatures(rgb2gray(I1), 'MetricThreshold', 600);
aPts2Surf   = detectSURFFeatures(rgb2gray(I2), 'MetricThreshold', 600);
fprintf('#Features %d/%d\n', length(aPts1Surf), length(aPts2Surf));
Dsc1        = extractFeatures(rgb2gray(I1), aPts1Surf);
Dsc2        = extractFeatures(rgb2gray(I2), aPts2Surf);
IxPrs       = matchFeatures(Dsc1, Dsc2, 'MaxRatio', 0.4);
aPts1Mtc    = aPts1Surf(IxPrs(:,1));
aPts2Mtc    = aPts2Surf(IxPrs(:,2));

% ------    Plot Matches   ------
subplot(2,2,3);
showMatchedFeatures(I1, I2, aPts1Mtc, aPts2Mtc);
title('All Matches');

%% =====    Transform matched points to the original image's coordinates
aPts1Mtc.Location = bsxfun(@plus, aPts1Mtc.Location, Org1new);
aPts2Mtc.Location = bsxfun(@plus, aPts2Mtc.Location, Org2new);

%% ======   Triangulate     =======
```

```
[Pts3D ErrReproj] = triangulate(aPts1Mtc, aPts2Mtc, MxCam1, MxCam2);
Bvalid      = ErrReproj < 1;        % identify strong correspondences
Pts3D      = Pts3D(Bvalid,:);      % select strong correspondences
aPts1Valid  = aPts1Mtc(Bvalid,:);
aPts2Valid  = aPts2Mtc(Bvalid,:);

% ---   mean distance from camera to features
DisFrom1    = sqrt(sum(bsxfun(@minus,Pts3D,PosCam1).^2,2));
DisFrom2    = sqrt(sum(bsxfun(@minus,Pts3D,PosCam2).^2,2));
fprintf('mean dist from 1: %3d mm\n', round(mean(DisFrom1)));
fprintf('mean dist from 2: %3d mm\n', round(mean(DisFrom2)));

% ------    Plot Strong    ------
subplot(2,2,4);
showMatchedFeatures(I1, I2, aPts1Valid,aPts2Valid);
title('Strong Matches');

%% -----    Plot 3D     ------
figure(2);
plotCamera('Location', PosCam1, 'Orientation', R1', 'Size', 10, ...
    'Color', 'r', 'Label', '1'); hold on; grid on
plotCamera('Location', PosCam2, 'Orientation', R2', 'Size', 10, ...
    'Color', 'b', 'Label', '2');

% get the color of each reconstructed point
[m n nCh]   = size(I1);
PtsR        = round(aPts1Valid.Location);
ICol        = reshape(im2double(I1), [m*n 3]);
IxPts       = sub2ind([m n], PtsR(:,2), PtsR(:,1));   % linear index
ColValid    = ICol(IxPts, :);

% add green point representing the origin
Pts3D(end+1,:)      = [0 0 0];
ColValid(end+1,:)   = [0 1 0];

% plot point cloud
showPointCloud(Pts3D, ColValid, 'VerticalAxisDir', 'down', 'MarkerSize', 45);
xlabel('X (mm)'); ylabel('Y (mm)'); zlabel('Z (mm)')
title('Reconstructed Point Cloud');
```

P.21 Posture Estimation

Section 26.4

In P.21.1 we give an example using the network as provided by PyTorch. In P.21.2 we show an example using the PoseNet system, accessed via OpenCV.

P.21.1 Posture in PyTorch

This example code is analogous to the example given in the object localization section P.8.1.

```
import numpy
import torchvision.models.detection as ObjRec
from torchvision import transforms
from skimage.io import imread

#%% ------    image path    ----------
imgPth      = 'C:/klab/do_lec/compvis/Matlab/Pose_TennisPlayerCrop.jpg'

#%% ------    Load an Image    ---------
Irgb        = imread(imgPth);
Trf2Tens    = transforms.ToTensor()
IrgbTo      = Trf2Tens(Irgb)

#%% %%%%%%%%%%%%%      NETWORK    %%%%%%%%%%%%%%%%%%%
NET         = ObjRec.keypointrcnn_resnet50_fpn(pretrained=True)

#%% %%%%%%%%%%%%%      APPLY    %%%%%%%%%%%%%%%%%%%
NET.eval()
aImgPred    = NET([IrgbTo])   # provide as list using square brackets []

COCO_PERSON_KEYPOINT_NAMES = [
    'nose',
    'left_eye',
    'right_eye',
    'left_ear',
    'right_ear',
    'left_shoulder',
    'right_shoulder',
    'left_elbow',
    'right_elbow',
    'left_wrist',
    'right_wrist',
    'left_hip',
    'right_hip',
    'left_knee',
    'right_knee',
    'left_ankle',
    'right_ankle'
]
nKeys   = len(COCO_PERSON_KEYPOINT_NAMES)
#%% -----   Plot     ------
from matplotlib import pyplot as plt
from matplotlib.patches import Rectangle
plt.figure(1); plt.imshow(Irgb)   # since we have only 1 image, we plot only that image
Prd     = aImgPred[0]           # prediction dictionary for this image
# we select only 1st prediction, note the index [0]
Boxes   = Prd['boxes'][0].detach().numpy()
KeyPt   = Prd['keypoints'][0].detach().numpy()
KeyScor = Prd['keypoints_scores'][0].detach().numpy()
for j in range(0,nKeys):
    print('%-15s  %1.6f'%(COCO_PERSON_KEYPOINT_NAMES[j], KeyScor[j]))
    Pt  = KeyPt[j,:];   x=Pt[0]; y=Pt[1]
    #plt.gca().add_patch(plt.plot(Pt[0],Pt[1],'r'))
    plt.plot(x,y,'yo')
    plt.text(x+6,y,str(j), color='red', fontsize=15, weight='bold')
plt.show()
```

P.21.2 PoseNet in OpenCV

Two files need to be downloaded, see the first two comment lines of the code block below:
- `pose_deploy_linevec_faster_4_stages.prototxt`: this is a text file that contains the architecture of the CNN. It is written in a format that is suitable for the CAFFE software.
- `pose_iter_160000.caffemodel`: this is a file containing the corresponding weight values, ca 200MB.

Those two files will be loaded into the program with the function `cv.dnn.readNetFromCaffe` from OpenCV: it generates the network called here `NET`.

The output of the estimate, `OMPS`, consists of ca. 40 confidence maps, of which the first 15 correspond to body parts, and the 16th to the background.

```
# https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/models/pose/mpi/pose_deploy_linevec_faster_4_stages.prot
# http://posefs1.perception.cs.cmu.edu/OpenPose/models/pose/mpi/pose_iter_160000.caffemodel
from numpy import zeros
import cv2 as cv

ifn     = "C:/IMGdown/POSEHUM/tennisPlayer.jpg"     # the image to be evaluated

# Specify the paths for the 2 files
pthFiles    = 'C:/IMGdown/POSEHUM/models/'
protoFile   = pthFiles + "pose/mpi/pose_deploy_linevec_faster_4_stages.prototxt"
weightsFile = pthFiles + "pose/mpi/pose_iter_160000.caffemodel"

#%% ====    construct network   =====
NET       = cv.dnn.readNetFromCaffe(protoFile, weightsFile)

#%% ====    load image and transform to blob image  =====
Ibgr      = cv.imread(ifn)
# generate blob image...
sclFact   = 1.0 / 255           # scale factor
szInp     = (368, 368)          # size of input
menSub    = (0,0,0)             # mean value to subtract
Iblob     = cv.dnn.blobFromImage(Ibgr, sclFact, szInp, menSub, swapRB=False, crop=False)

#%% =====    sent blob image through net     ======
NET.setInput(Iblob)             # enter image    [nImg nChr hi wi]
OMPS    = NET.forward()         # forward image: [nImg nMaps hMap wMap]

#%% =====    analyze maps     =======
nImg, nMaps, hMap, wMap = OMPS.shape
Iback   = OMPS[0, 15, :, :]     # background map [hMap wMap]
# --- Loop the 15 key points
aLoc=zeros((15,2)); Prob=zeros(15)
for i in range(15):
    Mconf       = OMPS[0, i, :, :]              # confidence map
    dmy, prob, dmy, Loc = cv.minMaxLoc(Mconf)   # global peak
    aLoc[i,:]   = Loc                           # coords [1..sizeMap]
    Prob[i]     = prob
print('Key probs range from %1.2f - %1.2f' % (Prob.min(), Prob.max()))

#%% ----    Plot 1   ------
Irgb    = cv.cvtColor(Ibgr, cv.COLOR_BGR2RGB) # convert to RGB for displaying
hFrm    = Ibgr.shape[0]
wFrm    = Ibgr.shape[1]

from matplotlib.pyplot import *
figure(1)
imshow(Irgb, cmap=cm.gray)
for i in range(15):
    Loc     = aLoc[i,:]                     # coords [1..sizeMap]
    x       = (wFrm * Loc[0]) / wMap
    y       = (hFrm * Loc[1]) / hMap
    plot(int(x), int(y), 'yo', markersize=12)
    text(int(x)+5, int(y), str(i), color='red', fontsize=12)
```

```
BodyPart    = [None]*16
BodyPart[0] = 'Head'
BodyPart[1] = 'Neck'
BodyPart[2] = 'Right Shoulder'
BodyPart[3] = 'Right Elbow'
BodyPart[4] = 'Right Wrist'
BodyPart[5] = 'Left Shoulder'
BodyPart[6] = 'Left Elbow'
BodyPart[7] = 'Left Wrist'
BodyPart[8] = 'Right Hip'
BodyPart[9] = 'Right Knee'
BodyPart[10] = 'Right Ankle'
BodyPart[11] = 'Left Hip'
BodyPart[12] = 'Left Knee'
BodyPart[13] = 'Left Ankle'
BodyPart[14] = 'Chest'
BodyPart[15] = 'Background'
```

For the face pose recognition system, one downloads the following two files - to build the network. Apart from that, the code pretty much remains the same as above.

```
# https://raw.githubusercontent.com/CMU-Perceptual-Computing-Lab/openpose/master/models/face/pose_deploy.prototxt
# http://posefs1.perception.cs.cmu.edu/OpenPose/models/face/pose_iter_116000.caffemodel
```

For the hand pose recognition system, one downloads the following two files:

```
# https://raw.githubusercontent.com/CMU-Perceptual-Computing-Lab/openpose/master/models/hand/pose_deploy.prototxt
# http://posefs1.perception.cs.cmu.edu/OpenPose/models/hand/pose_iter_102000.caffemodel
```

P.22 Action Recognition

Section 23.3

This requires a lot of RAM: 16GB will not do, perhaps 32GB might suffice, 64 should certainly manage. As the network expects a batch of videos, we insert a 5th dimension making it a 5-dimensional tensor. The output is an array of scores, `Scor`, of length 400, the number of classes (of the data base) the network was trained on.

```
import torch
import torchvision.models.video as ModVid
from torchvision.io import read_video

datBase     = 'C:/IMGdown/hmdb51_org/'
vidName     = 'brush_hair/April_09_brush_hair_u_nm_np1_ba_goo_0.avi'

#%% =======     Video Preparation   ======
FrmMean = [0.43216, 0.394666, 0.37645]      # mean values of kinetics-400
FrmStd  = [0.22803, 0.22145, 0.216989]      # std dev values of kinetics-400

def f_NormalizeToTensor(vid):
    return vid.permute(3, 0, 1, 2).to(torch.float32) / 255

def f_Resize(vid, size):
    return torch.nn.functional.interpolate(
        vid, size=size, scale_factor=None, mode='bilinear', align_corners=False)

def f_CenterCrop(vid, output_size):
    h, w    = vid.shape[-2:]
    th, tw  = output_size
    i = int(round((h - th) / 2.))
    j = int(round((w - tw) / 2.))
    return vid[..., i:(i + th), j:(j + tw)]

def f_NormalizeWithBase(vid, mean, std):
    shape = (-1,) + (1,) * (vid.dim() - 1)
    mean = torch.as_tensor(mean).reshape(shape)
    std = torch.as_tensor(std).reshape(shape)
    return (vid - mean) / std

#%% ------    Load a Video  ---------
Vid, Aud, info = read_video(datBase+vidName)
shpV        = Vid.shape
print('#frm %d, size [%d %d]'%(shpV[0],shpV[1],shpV[2]))
# --- prepare video:
Vid     = f_NormalizeToTensor(Vid)
Vid     = f_Resize(Vid,(128, 171))
Vid     = f_CenterCrop(Vid,(112, 112))
Vid     = f_NormalizeWithBase(Vid, mean=FrmMean, std=FrmStd)

#%% %%%%%%%%%%%%%      NETWORK    %%%%%%%%%%%%%%%%%%%
NET     = ModVid.r3d_18(pretrained=True)
#NET     = ModVid.mc3_18(pretrained=True)
#NET     = ModVid.r2plus1d_18(pretrained=True)

#%% %%%%%%%%%%%%%      APPLY    %%%%%%%%%%%%%%%%%%%
NET.eval()
Scor    = NET(Vid.unsqueeze(0))     # insert 5th dimension to make it a batch
ixMx    = Scor.argmax()             # class with maximum score E [0..400]
```

P.23 Grouping

P.23.1 Points

Section 24.1

```
clear;
nPts    = 1000;
Pts     = round(rand(nPts,2)*100);  % we generate artifical pixel coordinates

%% =============    NearestNeighbors      ===============
[NEB DI]  = knnsearch(Pts, Pts, 'k', 3);
NEB(:,1)    = [];             % exclude own
DI(:,1)     = [];
IxNN        = NEB(:,1);          % nearest neibor
mnDiNN      = mean(DI(:,1));

% ======    Mutuals
[IxU b]     = unique(sort([(1:nPts)' IxNN],2), 'rows');   % find isolated
Bmut        = true(nPts,1);                % default is double
Bmut(b)     = false;                       % inactivate unique ones
IxMut       = [find(Bmut) IxNN(Bmut)];
CenMut      = (Pts(IxMut(:,1),:) + Pts(IxMut(:,2),:))/2;

%% -------  Plot    --------
% we plot x/y as we don't have an image
figure(1); clf;
subplot(2,2,1);
    plot(Pts(:,1),Pts(:,2),'k.');
subplot(2,2,2);
    plot(Pts(:,1),Pts(:,2),'k.'); hold on;
    plot(CenMut(:,1),CenMut(:,2),'r.');
    title('Mutual NN');
subplot(2,2,3);
    plot(Pts(:,1),Pts(:,2),'k.'); hold on;
    for i = 1:nPts
        Pt1     = Pts(i,:);
        Pt2     = Pts(IxNN(i),:);   % nearest neighbor
        plot([Pt1(1) Pt2(1)], [Pt1(2) Pt2(2)], 'color',ones(1,3)*0.8);
    end
    title('1NN');
subplot(2,2,4);
    plot(Pts(:,1),Pts(:,2),'k.'); hold on;
    for i = 1:nPts
        Pt1     = Pts(i,:);
        Pt2     = Pts(IxNN(i),:);
        Pt3     = Pts(NEB(i,2),:);
        plot([Pt1(1) Pt2(1)], [Pt1(2) Pt2(2)], 'color',ones(1,3)*0.5);
        plot([Pt1(1) Pt3(1)], [Pt1(2) Pt3(2)], 'color',ones(1,3)*0.8);
    end
    title('2NN');
```

P.24 Text Recognition

P.24.1 Optical Character Recognition (OCR)

Section 26.2.1

```
clear;
Irgb    = imread('businessCard.png');
% Irgb    = rgb2gray(Irgb);

%% =====    OCR     =====
Txt     = ocr(Irgb);

%% -----    Plot    -----
nWords  = length(Txt.Words);
% add annotations to image Irgb
for i = 1:nWords
    word        = Txt.Words{i};
    wordBBox    = Txt.WordBoundingBoxes(i,:);
    Irgb        = insertObjectAnnotation(Irgb, 'rectangle', wordBBox, word);
end
figure(2);
imshow(Irgb);
```

```
# pip install pillow
# pip install pytesseract
# set an environment variable called TESSDATA_PREFIX' to:
#           'C:\\Program Files (x86)\\Tesseract-OCR\\tessdata\\
from PIL import Image
import pytesseract
pytesseract.pytesseract.tesseract_cmd='C:/Program Files (x86)/Tesseract-OCR/tesseract.exe'

#%% ----   Load Image   ----
pthImg  = 'someImageWithText.jpg'
Irgb    = Image.open(pthImg)

#%% ====    Detect  =====
Txt     = pytesseract.image_to_string(Irgb)
print(Txt)
```

P.25 Color

The transformation for RGB to Gray is given here as a single line, with `V` as a matrix made of three columns `[nPix 3]`:

```
G   = 0.2989 * V(:,1) + 0.5870 * V(:,2) + 0.1140 * V(:,3);
```

Examples for another two transformations are given in subsequent sections.

P.25.1 HSV

The code example is given as a single function script. The comment section in the first few lines, contains a subsection called 'USE', that illustrates how to deploy the function.

```
% Color transformation from RGB to HSV.
%
% IN   RGB   values as [nPix 3]
% OUT  HSV   values as [nPix 3]
%
% USE  Ihsv = f_Rgb2HSV(reshape(Irgb, [szI(1)*szI(2) 3]));
%      Ihsv = reshape(Ihsv, [szI(1) szI(2) 3]);
%
function HSV = f_Rgb2HSV(RGB)
[nPx nChr] = size(RGB);
assert(nChr==3);

%RGB        = double(RGB)/255;
RGB         = double(RGB);
[Vmin IxMin] = min(RGB,[],2);
[Vmax IxMax] = max(RGB,[],2);

Rng    = Vmax-Vmin;

H   = zeros(nPx,1);
for i = 1:nPx
    ixMx  = IxMax(i);
    rng   = Rng(i);
    if rng==0,          % gray-level value will be set to 0
        h  = 0;
    elseif ixMx==1
        h  = (       (RGB(i,2)-RGB(i,3)) / rng   ) / 6 ; % G-B
    elseif ixMx==2
        h  = ( 2 + ( (RGB(i,3)-RGB(i,1)) / rng ) ) / 6 ; % B-R
    else
        h  = ( 4 + ( (RGB(i,1)-RGB(i,2)) / rng ) ) / 6 ; % R-G
    end
    if h<0, h = h+1; end
    H(i) = h;
end

S           = Rng ./ Vmax;
S(isnan(S)) = 0;
if (any(isnan(S))),
    fprintf('NaN in S\n');
end
HSV         = [H  S  Vmax/255];
```

P.25.2 YCbCr

Analogous to the above example, this code example is given also as a function script. See the two lines under 'USE' on how to deploy it.

```
% Color transformation from RGB to YCbCr.
%
% IN   RGB   RGB values as matrix [nPix 3]
% OUT  YCC   YCC values as matrix [nPix 3]
%
% USE  Iycc = f_Rgb2YCbCr(reshape(Irgb, [szI(1)*szI(2) 3]));
%      Iycc = reshape(Iycc, [szI(1) szI(2) 3]);
%
function YCC = f_Rgb2YCbCr(RGB)

% This matrix comes from a formula in Poynton's, "Introduction to
% Digital Video" (p. 176, equations 9.6).

% T is from equation 9.6: ycbcr = origT * rgb + origOffset;
T = [65.481 128.553 24.966;...
     -37.797 -74.203 112; ...
     112 -93.786 -18.214] / 255;

Tred    = [ 65.481   -37.797   112] / 255;
Tgrn    = [128.553   -74.203  -93.786] / 255;
Tblu    = [ 24.966    112     -18.214] / 255;

offset  = [16; 128; 128];

YCC   = zeros(size(RGB,1), 3, 'uint8');  % initialize output
RGB     = single(RGB);
for i=1:size(RGB,1)
    red = RGB(i,1);
    grn = RGB(i,2);
    blu = RGB(i,3);
    YCC(i,1) = Tred(1)*red + Tgrn(1)*grn + Tblu(1)*blu + offset(1);
    YCC(i,2) = Tred(2)*red + Tgrn(2)*grn + Tblu(2)*blu + offset(2);
    YCC(i,3) = Tred(3)*red + Tgrn(3)*grn + Tblu(3)*blu + offset(3);
end
```