Algorithms for Image Processing

Winter Term 2021/22

Prof. Andreas Weinmann

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

fbmn
FACHBEREICH MATHEMATIK
UND NATURWISSENSCHAFTEN

Parts of these lecture notes are based on material designed and compiled by Konrad Sandau. Many thanks to him for the support.

Contents

Literature

- R. Gonzales and R. Woods. Digital Image Processing. Addison-Wesley, 1993.
- J. Beyerer, F. Puente Léon, und C. Frese. Automatische Sichtprüfung: Grundlagen, Methoden und Praxis der Bildgewinnung und Bildauswertung. Springer, 2016.
- K. Bredies and D. Lorenz. Mathematical Image Processing. Springer, 2018.
- D. Forsyth and J. Ponce. Computer vision: a modern approach. Pearson, 2012.
- B. Jähne. Digitale Bildverarbeitung. Springer, 2005.
- D. Knuth. The Art of Computer Programming. Addison Wesley, 1997.
- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. Numerical Recipes. Cambridge University Press, 2007.
- M. Sonka, V. Hlavac, and R. Boyle. Image Processing, Analysis, and Machine Vision. Thomson Learning, 2008.

1 Basics on Algorithms

1.1 Definitions and Basics on Algorithms

We start to point out the difference between our understanding of a *method* and an *algorithm*. A method is a plan to reach a scientific goal. An algorithm is more specific: it is a step by step description yielding a (desired) result within finite time and finitely many steps.
Quoting Wikipedia, most algorithms are intended to be implemented as computer programs. However, algorithms are also implemented by other means, e.g., in a biological neural network such as a human brain implementing arithmetic. Further examples are electrical circuits which were in fact be used to solve differential equations for instance.
A more formal computer science definiton of an algorithm (cf. also Wikipedia) is given as follows.

> A scheme to solve a problem is called *algorithm* whenever there is a Turing machine implementing this scheme such that this Turing machine produces a solution for any admissible input.

In consequence, a problem is called *computable* (formal definition), if there is an algorithm, and thus a related Turing machine, which computes a solution of the problem for every input.
We do not have the necessary time and background in theoretical computer science to further discuss the concept of a Turing machine and related automata theory and formal languages. However, we stress the following consequences implied by this definition.

- The scheme can be described by a finite sequence of characters (static finiteness).
- Every step of the scheme must be executable (executability).
- In every step, the scheme only needs finite memory (dynamic finiteness).
- The scheme only needs finitely many steps (termination.)

In addition, in many practical situations the term *algorithm* frequently tacitly implies

- The algorithm is deterministic in the sense that it produces a unique output, i.e., running twice for the same input yields the same output (no random output).
- The algorithm is deterministic in the sense that, in every step, the rule to apply is uniquely determined.

Example 1.1. One of the maybe oldest algorithms is the algorithm for finding the greatest common divisor (gcd) of two positive integers.
The underlying observation is that, for two nonnegative integers a, b with $a > b$,

$$\gcd(a,b) = \gcd(a-b,b) \quad \text{, and} \quad \gcd(a,0) = a. \tag{1}$$

We use this observation for the numerical example of determining $\gcd(111, 27)$:

$$\begin{aligned}\gcd(111, 27) &= \gcd(84, 27) = \gcd(57, 27) = \gcd(30, 27)\\ &= \gcd(3, 27) = \ldots = \gcd(3, 3) = \gcd(3, 0) = 3.\end{aligned}$$

We may realize the formula for computing the greatest common divisor (1) by two means, either *iteratively* or *recursively*.

Recursive and iterative schemes. An *iterative scheme* repeats the same step several times. (In many programming languages it is realized by for/while-loops etc.) For determining the greatest common divisor $\gcd(a, b)$, an iterative scheme may look as given by Algorithm 1. Note that we use pseudo code

Algorithm 1: GCD iteratively

Input: nonnegative integers a, b

Output: greatest common divisor $\gcd(a, b)$ of a, b

```
1  if a < b then
2  |  swap(a, b)
3  end
   /* now a ≥ b                                   */
4  while b > 0 do
5  |  a ← a − b
6  |  if a < b then
7  |  |  swap(a, b)
8  |  end
   |  /* now a ≥ b again                         */
9  end
10 return a
```

notation here emphasizing the important steps.
A *recursive scheme* is based on a function calling itself (with typically "smaller" arguments such that it terminates). This is a concept from functional programming. An example for determining the greatest common divisor is given by Algorithm 2. Recursion is –loosely speaking– the inverse of induction and as such in particular suited to calculate quantities defined inductively.
Further examples are the *factorial function* and the *Fibonacci numbers*. Using the symbol n to denote a positive integer, the factorial function is given by

$$n! = \begin{cases} n \cdot (n-1)! & \text{if } n > 0,\\ 1 & \text{if } n = 0. \end{cases} \tag{2}$$

Think about a recursive as well as about a iterative scheme to realize this function.
The Fibonacci numbers are defined via

$$f_n = f_{n-1} + f_{n-2}, \quad f_0 = 0, f_1 = 1. \tag{3}$$

Fibonacci used this sequence to describe the time evolution of a rabbit population. **Explain why a recursive scheme gets slow for large n. Propose an iterative scheme overcoming the problem.**

Algorithm 2: GCD recursively

Input: nonnegative integers a, b

Output: greatest common divisor $\gcd(a, b)$ of a, b

```
if a < b then
    swap(a, b)
end
if b=0 then
    return a
end
else
    return gcd(a − b, b)
end
```

Time and space complexity. The efficiency of an algorithm may be described by its time and space complexity. Since exact numbers are hard to calculate (and potentially do not yield further deeper insights) one sticks to consider complexity classes.
We start out to consider an example.

Example 1.2. Consider n real numbers stored in a vector x. The task is to sort them in ascending order.

As a first attempt we consider Algorithm 3. This algorithm is frequently termed

Algorithm 3: Bubble Sort

Input: vector x of n real numbers

Output: sorted vector x

```
for i = 0 : n − 2 do
    for j = i + 1 : n − 1 do
        if x[j] < x[i] then
            swap(x[j], x[i])
        end
    end
end
return x
```

bubble sort. (**Explain why the result is ordered.**) It needs $f_1(n) = \frac{n(n-1)}{2}$ comparisons. This number does not depend on the numbers stored in x. Further, in the worst case, we need $\frac{n(n-1)}{2}$ swaps, and each swap implies three assignments. Hence, for the number of swaps $f_2(n)$, we have $f_2(n) \leq \frac{3n(n-1)}{2}$. Note, that the actual number of swaps depends on the data in x. Thus the overall number of operations may be estimated from above by $2n(n-1)$. (At this point in this simple example, we already gave up finding exact numbers but started to estimate the complexity from above. Further, even summing the number of assignments and comparisons may be discussed since they may

take different time. If we start this discussion, we discuss the meaningfullness of the constants. Further determining such constants in harder problems may be very difficult.) However, the quintessence is that the time complexity grows quadratically in n.
A resonable means to formalize the above observations is to assign complexity classes to algorithms. This is done by using so-called Landau symbols. In the above example, we would say

$$f \text{ is in / belongs to the class } \mathcal{O}(n^2), \quad \text{in symbols: } f \in \mathcal{O}(n^2) \text{ or } f = \mathcal{O}(n^2) \tag{4}$$

Here, $\mathcal{O}(n^2)$ is actually a set functions, namely those which do not grow faster than quadratically (with the functions representing the time/space complexity of the algorithms dependent on the input size n.)
Instead of n^2 (which is short for the function $n \to n^2$) we may use any other function g to define the class $\mathcal{O}(g)$, of functions (interpreted as time/space complexity functions of algorithms.) Examples of g are, e.g., $n \log n$ or n^3. The definition is as follows:

$$f = \mathcal{O}(g) \qquad \Leftrightarrow \qquad \sup_n \left| \frac{f(n)}{g(n)} \right| < \infty \tag{5}$$

(**Is the [time complexity of the] above algorithm in $\mathcal{O}(n^3)$? Is it in $\mathcal{O}(n \log n)$? What is the space complexity of the algorithm above? Is it in $\mathcal{O}(n \log n)$?**)
The symbol $f = \mathcal{O}(g)$ formalizes the fact that the complexity of f is lower or equal to that of g. In order to formalize that f has lower complexity than g, one frequently uses the symbol $o(g)$. The corresponding complexity class is defined by

$$f = o(g) \qquad \Leftrightarrow \qquad \lim_{n \to \infty} \left| \frac{f(n)}{g(n)} \right| = 0. \tag{6}$$

(**Is the (time complexity of the) above algorithm in $o(n^3)$? Is it in $o(n^2)$?**)

1.2 Classes of Algorithms

One may try to distinguish algorithms with respect to various criteria. We here consider some of them in order to give the reader an impression.

Classes of algorithms w.r.t. machine capability. We may distinguish *deterministic* and *randomized* algorithms.
A *deterministic algorithm* only depends on the input only. Any intermediate quantities produced are uniquely determined and do not alter when running the algorithm twice with the same input.
A randomized algorithm employs random quantities during execution. Hence, intermediate quantities may vary when running the algorithm twice with the same input. Among randomized algorithms we may further distinguish *Las Vegas algorithms* and *Monte Carlo algorithms*. Las Vegas algorithms always yield a correct solution (example: random quick sort) while Monte Carlo algorithms may sometimes yield wrong solutions or approximations to solutions (which is typically not problematic since they are run several times and the wrong solutions are seldom events.)

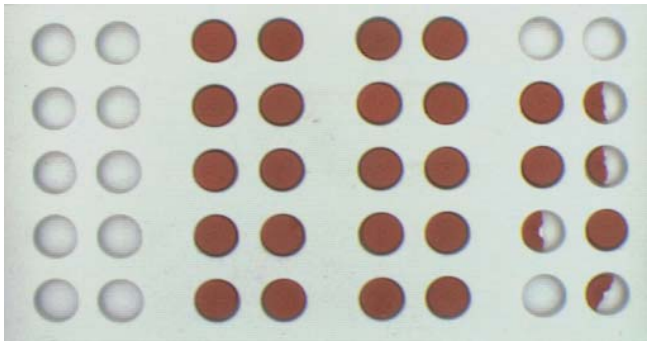


Figure 1: Example of data for a decision algorithm: a blister pack of pills. The decision task is to determine whether the blister is correctly filled. (Image Source: Lecture Notes Konrad Sandau)

Example 1.3. We explain a Monte Carlo Algorithm for determining π : consider the unit square together with its inscribed circle of radius $1/2$. According to a uniform distribution on the square draw a point and determine whether it is in the interior of the circle or not. Repeat this procedure n times and denote the number of points within the circle by Z. Return $4Z/n$.
From statistics we know that

$$\frac{\mathbb{E}(Z)}{n} = \frac{\pi}{4}.$$

(**why?**) which justifies the algorithm. (**why?**)

Classes of algorithms w.r.t. problem classes. Important examples are *decision algorithms* and *optimization algorithms.*
The task of a decision algorithm is to make a decision for the given input; for example to classify with respect to i.o./n.i.o in an inspection task based on an input image, see Figure 1.
The goal of an optimization algorithm is to find a local or global maximum or minimum of a target function f.

Example 1.4. Consider a univariate real convex function f defined on an iterval $[a, b]$, i.e.,

$$f : [a, b] \to \mathbb{R}.$$

Then f has a minimum. (**why?**) Further, due to convexity, every lokal minimum is a global minimum. (**why?**) We want to localize the minimum of f. (without loss of generality, we assume that a, b are no minima, because in this case we are already done.)
As algorithmic approach, we use "trisection of the interval":

- Divide the interval into three equi-spaced subintervals and calculate the function values at the boundaries.
- Determine the signs of the differences of the function values: if the sequence is $-/+/+$ remove the right interval, if the sequence is $-/-/+$ remove the left interval.

- Iterate this step until a given tolerance is reached.

Then a minimum is within the remaining interval which has width $(2/3)^k(b-a)$ after k steps. (**Why is this algorithm correct?**)

Concerning the above algorithm, we note that a particular subdivision of the interval according to the golden section is faster [21].
In order to solve optimization problems often *approximative algorithms* are employed: they do not solve the problem exactly but approximtely. (Typically, they are faster which is in particular important when the optimization problem has such a high complexity that it cannot be solved with the present computer architectures.)
Another class often used in optimization problems are *greedy algorithms*. One example is gradient descent. Greedy algorithms try to find the best gain in every step (which sometimes yields a solution of the minimization problem.)
Finally we mention *genetic algorithms* and refer to the literature for details [18].

Classes of algorithms w.r.t. application area. We first consider examples of *algorithms in geometry and graphics*. One class are algorithms for gridding (into pixel images) of areas, lines, circles or polygons. A prominent example is *Bresenham's (line) algorithm* (which may be used to draw a line in an image.) It is simple to implement and rounding errors caused by discretization of coordinates are minimized. It only uses additions of integers (and no multiplications, divisions) which makes it computationally cheap.

Algorithm 4: Bresenham Algorithms for angles between 0 and 45 degrees

Input: image

Output: image with line

$dx = x_{end} - x_{start}$; $dy = y_{end} - y_{start}$;
$x = x_{start}$; $y = y_{start}$;
Setpixel$(x, y, image)$;
$errorXdx = 0$;
while $x < x_{end}$ **do**
    $x = x + 1$;
    $errorXdx = errorXdx + dy$;
    if $errorXdx > 1/2 \cdot dx$ **then**
        $y = y + 1$;
        $errorXdx = errorXdx - dx$;
    end
    Setpixel$(x, y, image)$;
end
return *image*

Please note that the slope of the above line is given by dy/dx. To understand the algorithm while drawing a graphics it might be advantageous to divide $errorXdx$ by dx.
Further algorithms used in geometry, in particular in metric geometry, are, e.g.,

algorithms to compute Voronoi diagrams and their duals called Delaunay triangulations, or meshing algorithms.
An *algorithm* frequently used in *image processing* is the *floodfill algorithm*. We consider four neighborhoods and want to fill a respective connected region. The algorithm gets a start node (x_{start}, y_{start}) , a target color, and a replacement color. It finds all pixels in the image that are connected to the start node w.r.t. the target color and colors them using the replacement color. A recursive scheme is given by Algorithm 6.

Algorithm 5: floodfill (recursively)

Input: $image, (x_{start}, y_{start}), target\_color, replacement\_color$

Output: image

```
1  if image(x_start, y_start) != target_color then
2  |   return image;
3  end
4  else
5  |   image(x_start, y_start) = replacement_color);
6  |   floodfill(image, (x_start − 1, y_start), target_color, replacement_color);
7  |   floodfill(image, (x_start + 1, y_start), target_color, replacement_color);
8  |   floodfill(image, (x_start, y_start − 1), target_color, replacement_color);
9  |   floodfill(image, (x_start, y_start + 1), target_color, replacement_color);
10 end
```

Further algorithm used in image processing are the topic of this lectures.

Remark 1.1. There is a close relation between algorithms and data structures. In particular, for having fast algorithms, often adapted data structures such as linked lists or trees and graphs are necessary. We refer the interested reader to the computer science literature, e.g., [16]. This will not be a primary topic in the context of the image processing algorithms discussed in this course.

Remark 1.2. Further, we point out that the focus of this course is on the algorithms and not on the related software engineering. We acknowledge the importance of software engineering and refer to the courses of the study program more related to this topic for further details.

1.3 Filters and the Image Boundary

When filtering an image it is typically necessary to deal with its boundary in a suitable way. We discuss treating the boundary here.
There are different approaches:

1. Reducing the size of the resulting filtered image such that the filter operation is well-defined on the coordinates of the reduced image. (Typically, the size is reduced by the filter size, and about half the filter size is "lost" at each boundary.) We tacitely accept that the resulting image has smaller size.

2. Keeping the size of the resulting filtered image and defining suitable extra rules at the boundaries. Examples are (i) using a predefined value

(such as 0), (ii) using the input image value, (iii) using some "averaging" w.r.t. the valid pixels, e.g., mean or median, or a adapted rule.

3. Keeping the size of the resulting filtered image by first extending the image over the boundary in a suitable way, and applying the filter to the extended image. Examples of extension rules are (i) extension by a predefined value (such as 0), (ii) extension by the value of the nearest image coordinate, (iii) periodic extension, (iv) extension by reflection.

2 Rank Order Filters

Rank Order Filters are discussed as an example for the development of (efficient) algorithms here.
A rank order filter takes a local window w.r.t. some central pixel, e.g., the eight-neighborhood, or the "civilization" neighborhood, and computes the α-quantile in this neighborhood and sets the central pixel value by the computed value.
We point out that besides their importance in their own right, in particular of the importance of the median filter, the maximum and the minimum filters are the basic building blocks of morphological filters.

2.1 Basics

Quantiles and Sorting. The simplest means to compute rank order filters is to sort the whole neigborhood and then to take the corresponding item / quantile from the sorted list. However this is expensive, and faster methods are developed.
Let us formalize notions. A list X,

$$X = \{x_0, \ldots, x_{N-1}\}, \tag{7}$$

consisting of N elements is ordered, if $x_i \leq x_j$ for any i, j with $i < j$ between 0 and $N-1$. A median of a list X consisting of N elements is given by an element x_{med} which fulfills

$$\sharp\{x \in X : x \leq x_{\text{med}}\} \geq N/2, \quad \textbf{and} \quad \sharp\{x \in X : x \geq x_{\text{med}}\} \geq N/2, \tag{8}$$

i.e., at least half of the elements are smaller or equal to , and half of the elements are larger or equal to the median. If sorted, a median is a middle element. More general, an $\alpha-$quantile x_α of X is given by

$$\sharp\{x \in X : x \leq x_{\text{med}}\} \geq \alpha N, \quad \textbf{and} \quad \sharp\{x \in X : x \geq x_{\text{med}}\} \geq (1-\alpha)N. \tag{9}$$

If $X = \{x_0, \ldots, x_{N-1}\}$ is sorted,

$$x_\alpha = [x_{\lfloor \alpha(N-1) \rfloor}, x_{\lfloor \alpha N \rfloor}] \tag{10}$$

where $\lfloor \cdot \rfloor$ denotes the largest integer smaller or equal to the real number within the bracket. For instance, $\lfloor 1.75 \rfloor = 1$. Note, that the right-hand side of (10) yields an interval. Any point in the interval fulfills the balance condition in (9), and we call any point in the interval α-quantile. If the interval consists of only one element, the α-quantile is unique. For instance, for $X = \{1, 5, 4, 2\}$, we have $x_{0.25} = [2, 4]$, and $x_{0.3} = [4]$. For any element in x_α we also use the symbol x_α by a slight abuse of notation.
Note that the maximum of a list is a 1-quantile, and that the minimum is a 0-quantile.

Quick Sort. In order to sort a list X we already learned about the bubble sort algorithm, see Algorithm 3. A widely used scheme for sorting is the *quicksort algoritm.* We explain the idea. We start out choosing a pivot element, e.g., the

```python
def partition(array, low, high):

  pivot = array[high]
  i = low - 1

  for j in range(low, high):
    if array[j] <= pivot:
      i = i + 1
      (array[i], array[j]) = (array[j], array[i])

  (array[i + 1], array[high]) = (array[high], array[i + 1])
  return i + 1

def quickSort(array, low, high):

  if low < high:
    pi = partition(array, low, high)
    quickSort(array, low, pi - 1)
    quickSort(array, pi + 1, high)

```

Listing 1: Python-Code for QuickSort.

last element of the present list. Then, the present list is manipulated such that all elements smaller than or equal to the pivot are to the left, and all elements larger than the pivot are to the right of the pivot. This can be accomplished by passing from left to right until we meet an element larger than the pivot, and from right to left until we meet an element smaller than or equal to the pivot. Then we exchange them and proceed analogously until we meet. (For an slight modification, see Listing 1.) Then the list is separated into two lists, the elements left to the pivot (together with the pivot), and the elements to the right. On these both sublists, the same procedure is applied. A code example is given in Listing 1; an illustration can be found in Figure 2 .
The worst-case situation (with the rule of choosing the last element as a pivot) is that the list is already sorted. Then the tree degenerates into a list, and we need $O(n^2)$ comparisions for sorting. However, the average complexity is $O(n \log n)$. (**Question: Why?**) Although there are algorithms with worst case runtime $O(n \log n)$ such as MergeSort, QuickSort is widely used. In case of degenerate situations, a randomized quicksort is an option: this means that the pivot is NOT chosen deterministically (by the last element) but by random. In practice, if the randomized quicksort algorithm takes too much time, just stop and restart it.

Quicksort and α-quantiles. In order to find an α-quantile (the median is the 0.5-quantile), it is not necessary to perfom the whole quicksort algorithm to obtain a fully ordered result. In every step, only that half branch the α-quantile is in, has to be sorted.
In the worst case scenario, this still take $O(n^2)$ comparisons, however, in aver-

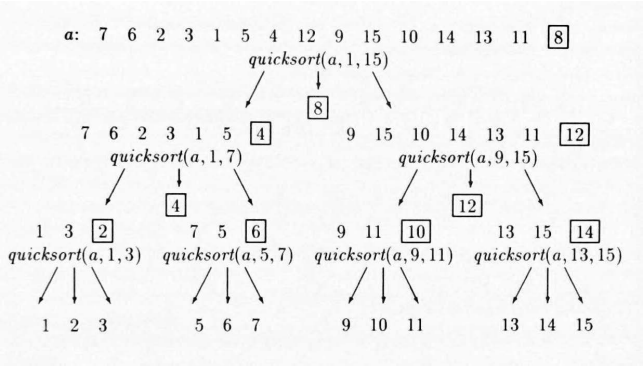


Figure 2: Illustration of QuickSort working on an example (Source: Ottmann & Widmayer).

age, it is in $O(n)$. (**Question: Why?**)
We point out the Blum-Floyd-Pratt-Rivest-Tarjan algorithm which computes an α-quantile (and thus a median) within $O(n)$ worst case time. For details we refer the interested reader to [3], or to the slides of E. Mayr on it `http://wwwmayr.in.tum.de/lehre/2006WS/ea/2006-12-12.pdf`.

2.2 Median filtering and rank order filtering

As already mentioned, a naive approach for median and rank order filtering would be to compute a median or an α-quantile in a local window for each pixel.
However, this approach ignores that neigboring pixels typically share a large part of their neighborhood pixels. For example consider two neighboring pixels and an eight-neighborhood. Then these pixels share 6 neighborhood pixels. Assume we have sorted the neighborhood U_p of one pixel and computed the median w.r.t. this neighborhood. Then we may remove the 3 pixels in $U_p \setminus U_q$ from the sorted list and update the median. (**Question: How?**) Next, we may insert the 3 pixels in $U_q \setminus U_p$ to get a sorted list of the neighborhood U_q of q and update the median. This way we obtain the median of U_q. One means to realize the ordered list with multiple entries is a histogram.

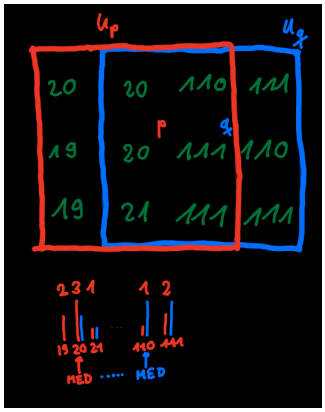


Figure 3: Idea of Huang's algorithm.

Now we may scan an image linewise by once sorting the neigborhood for the leftmost item and applying this update stategy. (For completeness, notice that the leftmost item can be computed from the above leftmost item by the same strategy detailed above.)

The approach above works for any α-quantile instead of the median.
The number of comparisons needed is $O(n^2k)$ where k is the edge length of the filter and n is the (longer) edge length of the image. (**Question: why?**) The naive approach would be $O(n^2k^2)$. In a 1D setting, the Huang algorithm only adds and removes one element per pixel and thus is in $O(n)$. An $O(n^2)$ algorithm not depending on k in 2D may be found in [19] (for finite value set.) An $O(n^2 \log k)$algorithm which is faster for smaller k may be found in [26].

Algorithm 6: Huang's median filtering algorithm.

Input: $m \times n$ image X, "radius" r

Output: Image Y

```
Initialize histogram H, median md
for i = 1 : m do
    for j = 1 : n do
        for k = −r : r do
            remove X_{i+k,j−r−1} from H
            add X_{i+k,j+r} to H
            update md
        end
        Y_{i,j} = m
    end
end
return Y
```

2.3 Maximum filtering

In the following we deal with maximum filters.
The maximum of a list is given by the smallest 1.0-quantile. In this respect a maximum filter can be seen as a special kind of rank order filter. Further, the minimum of a list can be found by the maximum of the list of all entries muliplied by -1 multiplied by -1. Thus any algorithm for finding a maximum of real numbers is one for finding a minimum as well.
The simplest algorithm for finding a maximum of a list is given as follows: initialize the maximum m by the first value; run through the list and compare the present item with m; if larger, set it to m. This algorithm needs as many comparisons as number of list entries (minus one) and is thus faster than finding a median or a general quantile.

Naive maximum filter. A naive maximum filter is obtained by iterating w.r.t. all image pixels and computing the maximum for each local neighborhood as above.
This approach is not that fast, since there are many redundancies. The asymptotic runtime is $O(n^2k^2)$ with n^2 denoting the number of image pixels and k^2 the number of pixels in the filter mask. In the following we will present successively improved approaches exploiting the following partitioning property.

Partitioning property. The maximum has the following property (not shared by a general α-quantile):

$$\max(A \cup B) = \max(\max A, \max B) = \max A \vee \max B, \tag{11}$$

where we also use the symbol $\vee$ to denote the maximum of two numbers in infix notation. This property is key for fast algorithms for maximum filtering and motivates why in general maximum filtering can by done faster than median filtering.

Pitas' scheme. We explain the idea in the univariate setting (1D image) with filter size $k = 2^l$. In a first step, for each pixel x_i, compute the maximum of the pixel and its successor to obtain the first iterate 1D image $y^{(1)}$ at position i by

$$y_i^{(1)} = x_i \vee x_{i+1}. \tag{12}$$

In the second step, take the maximum of $y_i^{(1)}$ and $y_{i+2}^{(1)}$ to obtain the second iterate 1D image $y^{(2)}$

$$y_i^{(2)} = y_i^{(1)} \vee y_{i+2}^{(1)}. \tag{13}$$

The *crucial observation* is that, by (11),

$$y_i^{(2)} = y_i^{(1)} \vee y_{i+2}^{(1)} = \max(x_i, x_{i+1}, x_{i+2}, x_{i+3}). \tag{14}$$

Next, we get $y^{(3)}$ by

$$y_i^{(3)} = y_i^{(2)} \vee y_{i+2^2}^{(2)}, \tag{15}$$

and, by (11),

$$y_i^{(3)} = \max(x_i, x_{i+1}, \ldots, x_{i+7}). \tag{16}$$

We proceed analogous and obtain, for the last step ,

$$y_i^{(l)} = y_i^{(l-1)} \vee y_{i+2^{l-1}}^{(l-1)}, \tag{17}$$

and $y_i^{(l)}$ stores the maximum $\max(x_i, x_{i+1}, \ldots, x_{i+2^l-1})$. This yields the desired maximum filter. Instead of $O(nk)$ for the naive approach, this approach takes $O(n \log k)$ comparisons. This is because we compute $l = \log k$ iterates y^i, $i = 1, \ldots, l$ and computing each y^i takes less than n comparisions where n denotes the length of the 1D image.
Concerning filter sizes not of the form $k = 2^l$ we may adapt the shifts s_r in

$$y_i^{(r)} = y_i^{(r-1)} \vee y_{i+s_r}^{(r-1)} \tag{18}$$

to obtain the desired size. Note that we had $s_r = 2^{r-1}$ above. For instance, a maximum filter of size 7 is obtained by replacing (15) by

$$y_i^{(3)} = y_i^{(2)} \vee y_{i+3}^{(2)}. \tag{19}$$

Bivariate Setting. Now let us lift this to the bivariate setting (2D images) for *rectangular neigborhoods*. For a maximum filter w.r.t. a $k \times k$ neigborhood we first compute the univariate maximum filter on the lines to obtain an intermediate image with the maxima on the lines. On the intermediate result, we apply the maximum filter columnwise. This yields the maximum w.r.t. the recangular neigbohood. (**Question: Why?**)
We further consider *v-normal neighborhoods*. They are important since they allow for better approximation of circles. A v-normal neighborhood is a neigborhood consisting of n_v successive linewise intervals. For instance, the ball-approximationg "civilization neigborhood" is v-normal and consists of five lines, the first and the last of length 3 and the three in the middle of length 5. To compute the maximum filter w.r.t. a v-normal neighborhoods, (i) first compute, for each line, the linewise maxima w.r.t all intervals of the neighborhood system. (ii) Then consider a pixel in the j-th line, the corresponing precomputed maxima for the neigboring lines (w.r.t. the length of the corresponding interval in the v-normal neigborhood) and compute their maximum.
Observe that dealing with v-normal neighborhoods is computationally more expensive than dealing with rectangular neigborhoods.

Van Herk-Gil-Werman algorithm. The following idea goes back to van Herk [25] and Gill and Werman [8].
We again consider the univariate situation (1D image). The idea is to consider every kth pixel and to compute left and right intervals starting at the pixels up to length k. More precisely, we let

$$R(i)_l = \max(x_{i-l}, \dots, x_i), \qquad S(i)_l = \max(x_i, \dots, x_{i+l}), \tag{20}$$

and compute $R(i)_l$ for $l = 0, \dots, k-1$ and $S(i)_l$ for $l = 0, \dots, k-1$ for every kth point $i = nk - 1$. For a visualization for $k = 7$ see Figure 4. The computation of the $R(i)$ and the $S(i)$ is called the *preprocessing step*.

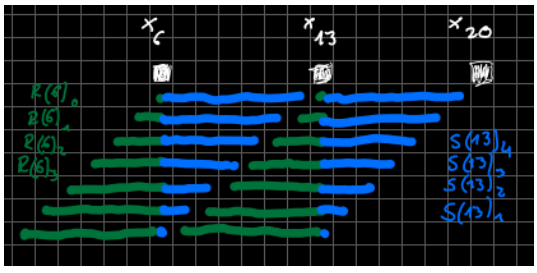


Figure 4: Idea of the van Herk-Gil-Werman algorithm.

For the second step, the *merging step*, we observe that $R(i)_k$ and $S(i)_k$ already equal the maximum filter at positions between $i-1$ and i resp. i and $i+1$. The further maxima around position i are computed as the maximum of $R(i)_l$ and $S(i)_{k-1-l}$, $l = 1, \dots, k-2$. More precisely, the maximum filter y of x near i is given by

$$y_{i+\lfloor \frac{k+1}{2} \rfloor - l} = R(i)_l \vee S(i)_{k-1-l}, \qquad l = 1, \dots, k-2. \tag{21}$$

Let us count operations by counting the number of comparisons: For each "macropixel"-index i, we need $2(k-1)$ comparisons for computing the $R(i)$'s and the $S(i)$'s for the preprocessing. For the merging step, we need $k-2$ comparisons. In total, this amounts to $3k-4$ operations per "macropixel". If N

denotes the input signal length, there are in the order of N/k "macropixels" and since $\frac{N}{k}(3k-4) \leq 3N$, the computational cost amounts to less than 3 comparisons per input pixel, independent of the filter size. Hence, the van Herk-Gil-Werman algorithm is an $O(N)$ algorithm for computing a maximum filter independent of the filter size and the constant hidden in the $O(N)$ notation is less than or equal to 3, independent of k.
The extension to a bivariate setting (2D images) is accomplished as described above.

Accelartion by Gil and Kimmel. A further acceleration of the van Herk-Gil-Werman algorithm by Gil and Kimmel can be found in [9]. We briefly explain the ideas.

- *Accelating the preprocessing step:* We first observe that the intervals for the $R(i+1)$'s and the $S(i)$'s overlap (cf. Figure 4). We first compute only half of the $R(i+1)$'s and the $S(i)$'s, i.e., $S(i)_l$ for $l = 0, \ldots, \lfloor \frac{k}{2} \rfloor$ and $R(i+1)_l$ for $l = 0, \ldots, \lfloor \frac{k-1}{2} \rfloor$. Then we compare the maxima of the largest computed intervals $s = S(i)_{\lfloor \frac{k}{2} \rfloor}$ and $r = R(i+1)_{\lfloor \frac{k-1}{2} \rfloor}$. (In Figure 4, these would be $r = R(13)_3$ and $s = S(6)_3$.) If $s \leq r$ the $R(i+1)$'s corresponding to larger intervals will not increase and we may set

$$R(i+1)_l := r \qquad \text{for} \quad l \geq \lfloor \frac{k-1}{2} \rfloor \tag{22}$$

without further comparison. It remains to compute the remaining $S(i)_l$, for $l > \lfloor \frac{k}{2} \rfloor$. Similarly, if $s \geq r$ the $S(i)$'s corresponding to larger intervals will not increase and we may set

$$S(i)_l := s \qquad \text{for} \quad l \geq \lfloor \frac{k}{2} \rfloor \tag{23}$$

without further comparison. It remains to compute the remaining $R(i)_l$, for $l > \lfloor \frac{k-1}{2} \rfloor$. Overall, instead of the order of $2k$ comparisons, only of the order of $1.5k$ comparisons are necessary.

- *Accelating the merging step:* We consider the $R(i)$'s and the $S(i)$'s (not the $R(i+1)$'s and the $S(i)$ as above!). We compare $\bar{s} = S(i)_{\lfloor \frac{k}{2} \rfloor}$ and $\bar{r} = R(i)_{\lfloor \frac{k-1}{2} \rfloor}$, (e.g., $\bar{s} = S(6)_3$ and $\bar{r} = R(6)_3$ in Figure 4.) If $\bar{s} \geq \bar{r}$,

$$R(i)_l \vee S(i)_{k-1-l} = S(i)_{k-1-l}, \qquad \text{for} \quad k-1-l > \lfloor \frac{k}{2} \rfloor, \tag{24}$$

the indices corresponding to the $S(i)$'s for larger intervals. Similarly, if $\bar{s} < \bar{r}$, we may set

$$R(i)_l \vee S(i)_{k-1-l} = R(i)_l, \qquad \text{for} \quad l > \lfloor \frac{k-1}{2} \rfloor, \tag{25}$$

the indices corresponding to the $R(i)$'s for larger intervals. In any case we need one comparison for setting half of the values. For the remaining half, we may proceed analogous comparing $R(i)_l$ and $S(i)_{k-1-l}$ w.r.t. the

suitable middle index l, to reduce to a quarter of values. The described procedure corresponds to the first steps of a binary search. It requires $\log_2 k$ comparisons and thus reduces the number of comparisons from k to $\log_2 k$.

For details, we refer to the paper [9].
We point out that, as detailed in the following section, that maximum filtering amounts to the basic operation of dilation in grey-value morphology. Further important operations are erosion as well as opening and closing. From an algorithmic side, erosion is given by a minimum filter and fast algorithms are given analogous to that for maximum filters. Opening and closing are given by an erosion followed by a dilation and by a dilation followed by an erosion, respectively. So they can be directly realized by the combination of minimum and maximum filters. However, as discussed in [9], there are –due to monotonicity– algorithms faster than that.

3 Morphological Filters

3.1 Basic Binary Morphology.

We first recall basics on morphology on binary images. To fix ideas, we always consider subsets of the pixel grid, i.e., integer tuples $x = (x_1, x_2)$ with $x_1, x_2 \in \mathbb{Z}$.

A *structuring element* is a set of integer tuples with a distinguished center, the coordinates $(0,0)$; see Figure 5 for an example of the important four neighborhood and eight neigborhood structuring elements. The (non-shifted) structuring elements in the figure are symmetric w.r.t. their center $(0,0)$. More precisely, with $x = (x_1, x_2) \in A$ we have that $-x = (-x_1, -x_2) \in A$ as well (which is the definition of a symmetric structure element.)

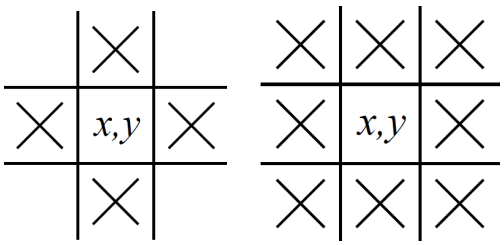


Figure 5: Schematic depiction of four neighborhood and eight neigborhood structuring elements shifted to x, y. The basic unshifted structuring element is given by the particular instance $(x, y) = (0, 0)$.

Dilation and erosion. These are the basic operations. The *dilation* of Y (often the foreground of a binary image) by a structuring element A is given by

$$D_A(Y) = \{x \mid Y \cap A_x \neq \emptyset\} \tag{26}$$

where

$$A_x = \{a + x : a \in A\} \tag{27}$$

denotes the structuring element A shifted by $x = (x_1, x_2)$, see Figure 5. For a visualization of the dilation see Figure 6; for an example see Figure 7.

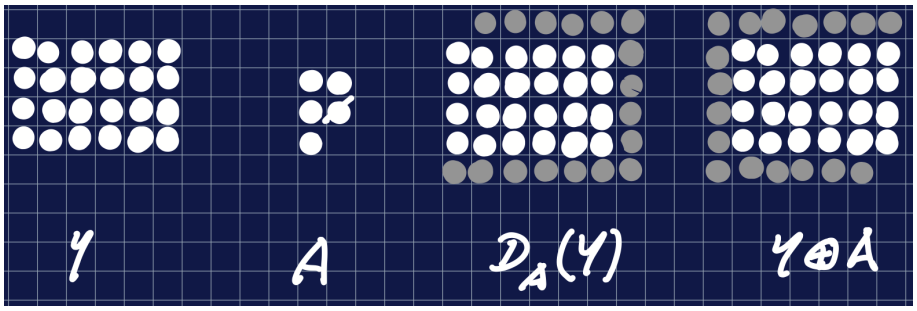


Figure 6: Visualization of the difference of dilation $D_A(Y)$ and Minkowski sum $A \oplus Y$ for non-symmetric structuring elements. The results on the right hand side are the union of the gray and white coordinates.

There is a close relation with the *Minkowski sum*

$$Y \oplus A := \{y + a : y \in Y, a \in A\} = \cup_{a \in A} Y_a = \cup_{y \in Y} A_y.$$

Figure 7: *left:* binarized input; *middle:* dilation with a ball of radius 10; *right:* erosion with the same ball.

of sets. The Minkowski sum is obviously symmetric in Y and A, in symbols $Y \oplus A = A \oplus Y$.
We have that

$$D_A(Y) = Y \oplus A^-, \qquad \text{where } A^- = \{-a : a \in A\} \tag{28}$$

denotes the reflection of A. Hence,

$$D_A(Y) = Y \oplus A \qquad \text{if } A \text{ is symmetric.} \tag{29}$$

Thus, the dilation $D_A(Y)$ equals the union of all shifts of the reflected structuring element A_y^- (which equals A_y when A is symmetric) for shift parameters in Y. (**Use the four neigborhood and a small three-pixel image foreground and concretely write down the involved quantities accompanied by drawing. Then replace the four neigborhood by a non symmetric neigborhood and proceed similarly.**)
The *erosion* of Y by a structuring element A is given by

$$Y \ominus A = \{x \in Y | A_x \subset Y\}. \tag{30}$$

For a visualization, see Figure 8.

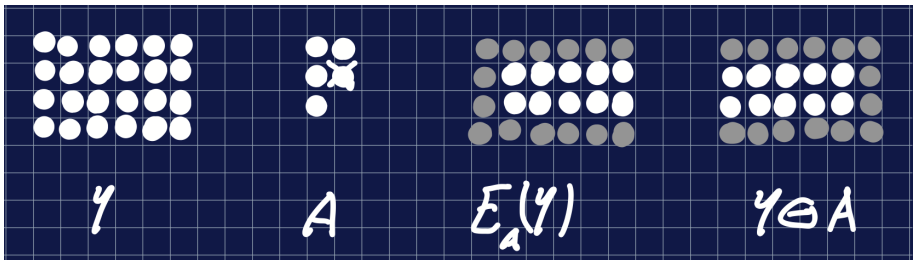


Figure 8: Visualization of the difference of dilation $E_A(Y)$ and Minkowski difference $A \ominus Y$ for non-symmetric structuring elements. (The results on the righthand-side are the white coordinates.)

The *Minkowski subtraction* of Y by A is given by

$$Y \ominus A = \cap_{a \in A} Y_a. \tag{31}$$

We have the connection

$$E_A(Y) = Y \ominus A^- \tag{32}$$

Figure 9: Erosion of Sylt island with a non-symmetric structuring element (Source: Sandau.)

between erosion and Minkowski subtraction. In particular, if A is symmetric, then

$$E_A(Y) = Y \ominus A.$$

We note that non-symmetric structuring elements are important and have to be considered therefore; cf. Figure 9.

Relation between dilation and erosion. We have that

$$D_A(X^C) = [E_A(X)]^C. \quad (33)$$

We refer to Figure 10 for an illustration of this most important relation. This may be interpreted as: Dilation on the foreground acts as erosion on the background and vice versa. (**Explain this fact/formula.**) Note that one reason for the above definition of the dilation is the fact that this equation still holds for nonsymetric structuring elements.

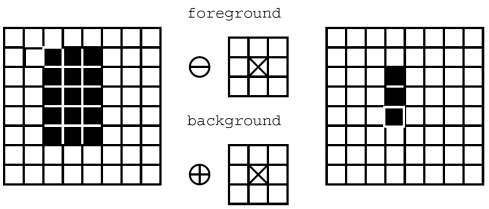


Figure 10: Dilation on the foreground acts as erosion on the background. (Source: Aurich)

Boundary. Using erosion, we define the boundary ∂Y of a set Y by

$$\partial Y := Y \setminus E_N(Y), \quad (34)$$

it is the difference of Y and its erosion $E_N(Y)$, where N is a symmetric neighborhood, typically the four or eight neighborhood from above. As a consequence, the boundary is a subset of Y and maybe seen as an inner boudary. An example is given in Figure 11 (The corresponding outer boundary can be obtained by duality. **How?**).

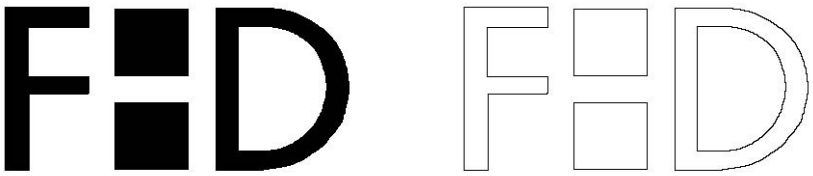


Figure 11: Set Y (*left*) and its boundary ∂Y (*right*) using the four.neighborhood. The boundary of a connection component is connected w.r.t. the eight neigborhood. (Source: K. Sandau)

Remark 3.1. A morphological variant of the Laplace operator maybe defined by

$$D_N(Y) \setminus Y - Y \setminus E_N(Y), \tag{35}$$

i.e., by subtracting the outer boundary from the inner boundary.

Properties of dilation and erosion. Dilation and erosion have the following properties:

(i) D_A is increasing, i.e., $D_A(X) \subset D_A(Y)$ whenever $X \subset Y$. (By symmetry, E_A is decreasing, i.e., $E_A(X) \subset E_A(Y)$ whenever $X \subset Y$.)

(ii) D_A is extensive, i.e., $Y \subset D_A(Y)$ for all Y if the structuring element A contains the origin $(0, 0)$. (By symmetry, E_A is anti-extensive , i.e., $E_A(Y) \subset Y$ whenever A contains the origin.)

(iii) E_A is convexity preserving, i.e., if Y is convex, so is $E_A(Y)$.

(iv) Translation invariance: $E_A(Y)_h = E_A(Y_h)$, i.e., translation and erosion commute. The same holds for the dilation.

Further, there are the following rules of calculus:

(a) $D_{A\cup B}(Y) = D_A(X) \cup D_B(Y)$;

(b) $D_{A\oplus B}(Y) = D_A(D_B(Y))$;

(c) $E_{A\cup B}(Y) = E_A(X) \cap E_B(Y)$;

(d) $E_{A\oplus B}(Y) = E_A(E_B(Y))$.

(**Try to verify the validity of these properties.**) Erosion und dilation are also montonic w.r.t. the structuring elements, i.e.,

$$A \subset B \qquad \Rightarrow \qquad D_A(Y) \subset D_B(Y) \quad (E_A(Y) \supset E_B(Y)). \tag{36}$$

for any $Y \subset \mathbb{Z}^2$.
These rules deal with two structuring elements. From the representation as Minkowski sum and differences we may also derive corresponding rules of calculus for the case of two images and one structuring elements. For instance, we have

$$E_A(X \cap Y) = E_A(X) \cap E_A(Y)$$

which follows from (a) above (by passing to the complement).

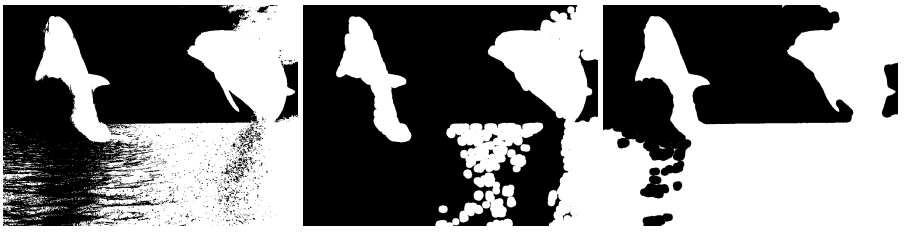


Figure 12: *left:* binarized input; *middle:* opening with a ball of radius 10; *right:* closing with the same ball.

Figure 13: *From left to right:* input; opening with a square of size 3; closing with the same square; opening followed by closing; closing followed by opening.

Opening and closing. The *opening* of a set Y using the structuring element A is given by

$$O_A(Y) = D_{A^-}(E_A(Y)), \tag{37}$$

i.e., the concatenation of the erosion using the structuring element A with the dilation using its reflection A^- as the structuring element. For an example, see Figure 12.
The *closing* of a set Y using the structuring element A is given by

$$C_A(Y) = E_{A^-}(D_A(Y)), \tag{38}$$

i.e., the concatenation of the dilation using the structuring element A with the erosion using its reflection A^- as the structuring element.
Similar to dilation and erosion, we have the following duality for opening and closing

$$C_A(Y^C) = O_A(Y)^C, \qquad O_A(Y^C) = C_A(Y)^C. \tag{39}$$

Typically, when applying openings, small objects disappear, and thin connections are erased. Closings, on the other hand, fill holes and connect nearby objects.
Opening and closing have the following properties:

(i) C_A is increasing, i.e., $C_A(X) \subset C_A(Y)$ whenever $X \subset Y$. (O_A is decreasing, i.e., $O_A(X) \subset O_A(Y)$ whenever $X \subset Y$.)

(ii) C_A is extensive, i.e., $Y \subset C_A(Y)$ for all Y. (O_A is anti-extensive , i.e., $O_A(Y) \subset Y$ whenever A contains the origin.)

(iii) Translation invariance: $C_A(Y)_h = C_A(Y_h)$, and $O_A(Y)_h = O_A(Y_h)$, for any set Y and any translation parameter h.

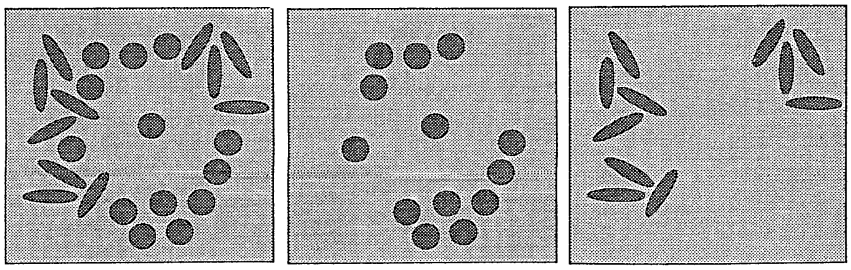


Figure 14: *left:* binary input; *middle:* opening with a ball shaped structuring element; *right:* union of openings with rotated bar-shaped structuring elements. (Source: Serra)

(iv) Opening and Closing are *idempotent*:

$$C_A(C_A(Y)) = C_A(Y), \qquad O_A(O_A(Y)) = O_A(Y).$$

This means that applying the respective operation twice yields the same result as applying it once. In other words, the second application does not alter the result.

There is the following characterization

$$O_A(Y) = \cup_{x:A_x \subset Y} A_x, \tag{40}$$

i.e., $O_A(Y)$ equals the union of all shifts of the structuring element which are completely contained in Y.

Remark 3.2. Defining an opening as concatenation of erosion with dilation, we obtain the properties (i)-(iv) above. Conversely, an *algebraic opening* is defined as any mapping fulfilling these properties. An example is the ring opening

$$\gamma(Y) = Y \oplus A \cap Y,$$

where A denotes a circular ring. The notion of an algebraic opening is more general.

Direct applications. A first application is cleaning images. If the objects of interest are sufficiently wide and the perturbations are rather small or lengthy, a means to get rid of the perturbations is to employ an opening with a disc shaped structuring element of suitable size. If the objects of interest contain small holes, a means to get rid of the holes is to employ a closing with a disc shaped structuring element of suitable size. As an example, see Figure 13.
Another application is the classification of objects w.r.t. their form. One may choose different structuring elements fitting with the respective objects, and apply each of these different structuring elements. For an example we refer to Figure 14.

3.2 Gray Value Morphology

Instead of considering sets, we now shift to considering functions.
A gray value image I is a function $I : \mathbb{Z}^2 \to \mathbb{R}$ or $\mathbb{Z}$ (at least subsets of it depending on the viewpoint.) Its graph is a "surface" separating three-dimensional space into two parts, the part above, and the part below the graph, the latter called *umbra*.

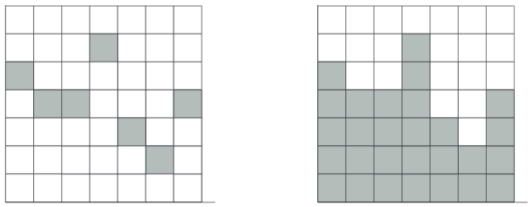


Figure 15: Umbra of a 1d discrete function

More formally, starting from an image I, its umbra Y_I is given by

$$Y_I = \{(x_1, x_2, t) : t \leq I(x_1, x_2)\}. \tag{41}$$

Consider Figure 15 for a visualization in 1d. The umbra is a set (instead of a function as is I), now in 3d space. The morphological operations previously defined do not depend on space dimension, and can therefore be applied to the umbra Y_I, and, in consequence, to gray value images by applying them to the corresponding umbrae. Given an umbra Y, a corresponding image I_Y can be reconstructed by

$$I_Y(x_1, x_2) = \max\{t : (x_1, x_2, t) \in Y\}. \tag{42}$$

Intersection on the side of umbrae, i.e., $Y_{I_1} \cap Y_{I_2}$ translates into forming the minimum on the functions side. More precisely,

$$Y_{I_1} \cap Y_{I_2} = Y_{I_1 \wedge I_2}, \qquad \text{where } I_1 \wedge I_2(x, y) := \min(I_1(x, y), I_2(x, y)) \tag{43}$$

and the minimum is the pixelwise minimum of the images. Analogously, forming the union on the umbra side $Y_{I_1} \cup Y_{I_2}$ translates into forming the maximum on the functions side. More precisely,

$$Y_{I_1} \cup Y_{I_2} = Y_{I_1 \vee I_2}, \qquad \text{where } I_1 \vee I_2(x, y) := \min(I_1(x, y), I_2(x, y)) \tag{44}$$

and the maximum is the pixelwise maximum of the images.
We consider sets Y in $\mathbb{Z}^2$ as discussed before. They may be interpreted as binary images taking two values. Here, one possibility is to choose function values as

$$\chi_Y(x_1, x_2) = \begin{cases} 1 & \text{if} (x_1, x_2) \in Y \\ 0 & \text{if} (x_1, x_2) \notin Y. \end{cases} \tag{45}$$

In other terms, we identify the set Y with its characteristic function χ_Y. (Note that also replacing 1 and 0 by other values can be reasonable.) Structuring elements $A \subset \mathbb{Z}^2$ are embedded into $\mathbb{Z}^3$ via

$$\{(x_1, x_2, 0) : x \in A\} \tag{46}$$

and are also denoted by A.

Remark 3.3. We note that structuring elements $A \subset \mathbb{Z}^2$ are called flat in the literature. We here restrict our considerations to such flat structuring elements $A \subset \mathbb{Z}^2$ and refer the interested reader to [10] for details on the non flat case.

Figure 16: *left:* input; *middle:* dilation with a ball of radius 10; *right:* erosion with the same ball.

Dilation and erosion. *Dilation and erosion* of the gray value image I are defined as dilation and erosion of the corresponding umbrae, i.e.,

$$D_A(I) = I_{D_A(Y_I)}, \qquad E_A(I) = I_{E_A(Y_I)}. \tag{47}$$

This means we form the umbra Y_I of I according to (41). Then we dilate/erode Y_I in $\mathbb{Z}^3$ using the structuring element A embedded into 3d via (46). Finally, we identify the resulting umbra with a function via (42). For an example, see Figure 16.
Writing down the above definition in terms of functions yields for the dilation

$$D_A(I)(z_1, z_2) = \max_{(x_1,x_2)\in A_z^-} [I(x_1, x_2)]\,. \tag{48}$$

For the erosion we have

$$E_A(I)(z_1, z_2) = \min_{(x_1,x_2)\in A_z} [I(x_1, x_2)]\,. \tag{49}$$

Hence, dilation and erosion in a gray value image amount to maximum filtering and minimum filtering, respectively.
The analogous properties of erosion and dilation derived for binary images hold via the interpretation as umbrae. On the function side, subset symbols $\subset$ have to be replaced by $\leq$ symbols etc.

Opening and closing. Similar to dilation and erosion, *opening and closing of gray value images* can be defined via the corresponding umbrae, i.e.,

$$O_A(Y_I) = D_{A^-}(E_A(Y_I)), \qquad C_A(Y_I) = E_{A^-}(D_A(Y_I)). \tag{50}$$

Writing this down in terms of functions yields for the opening

$$O_A(I)(z_1, z_2) = \max_{(x_1,x_2)\in A_z} [E_A I(x_1, x_2)]\,. \tag{51}$$

For the closing, we have

$$C_A(I)(z_1, z_2) = \min_{(x_1,x_2)\in A_z^-} [D_A I(x_1, x_2)]\,. \tag{52}$$

Figure 18: *left:* input; *middle:* opening with a ball of radius 10; *right:* closing with the same ball.

Hence, an opening corresponds to first applying a minimum filter followed by a maximum filter. A closing corresponds to first applying a maximum filter followed by a minimum filter. For a visualization, see Figure 17; for an example, see Figure 18.

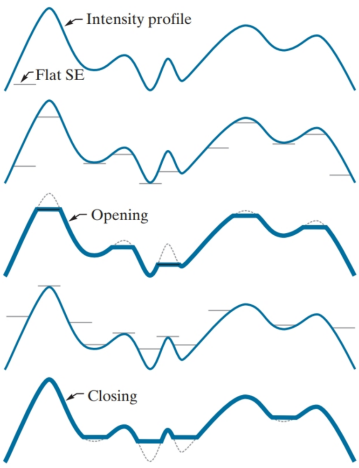


Figure 17: Illustration of opening and closing for 1D signals. (Source: Gozales/Woods)

Due to the definition via umbrae, the properties of openings and closings derived for binary images hold for gray value images mutas mutandi. In particular, C_A is increasing and O_A is decreasing, C_A is extensive and O_A is anti-extensive, both operations are translation invariant, and both operations are idempotent.

Morphological gradient. For binary images we defined the boundary (of the foreground Y) by $Y \setminus E_A(Y)$ where A denotes a 'unit disc' structuring element. We now discuss this operation for gray value images. Since the erosion $E_A(I)$ of an image I at position x, i.e., $E_A(I)(x_1, x_2)$ equals the minimum of I in A_x in the A-neigborhood of x, the expression $I - E_A(I)(x_1, x_2)$ equals the absolute value of the difference between I at pixel (x_1, x_2) and the value in steepest descend direction. This motivates the definition of the morphological gradient

$$\rho^-(I) = I - E_A(I) \tag{53}$$

which clearly depents on the 'unit disc' A. Analogously, we may define

$$\rho^+(I) = D_A(I) - I, \qquad \rho(I) = D_A(I) - E_A(I) \tag{54}$$

as the outer and central variant, respectively. The right-hand gradient ρ is frequently called Beucher gradient. For a visualization we refer to Figure 19. We point out that carrying out the above argument in $\mathbb{R}^n$ actually yields a formula for the absolute value of the gradient.

Figure 19: *From top left to bottom right:* data, morphological gradient ρ^-, morphological gradient ρ^+, Beucher gradient.

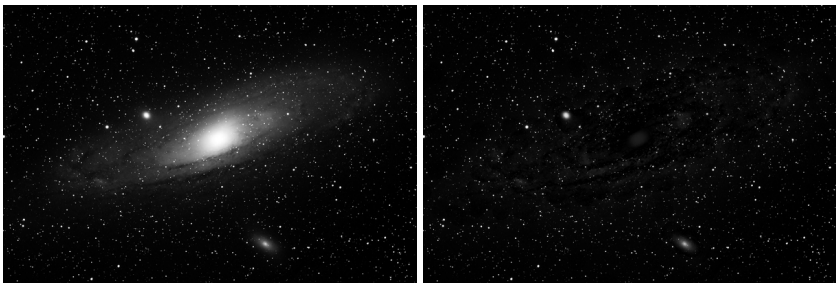


Figure 20: Result of the white tophat transform using a disc shaped structuring element of size 12; image: 602×404.

Similarly, we may define a morphological Laplace operator for (bivariate) gray value images by

$$L(I) := D_A(I) + E_A(I) - 2I. \tag{55}$$

Tophat transforms. In analogy to the gradient we define the white tophat of the image I by

$$WTH(I) = I - O_A(I). \tag{56}$$

In contrast to the gradient, A may denote an arbitrary (potentially larger) structuring element here. The dual transform is called black tophat. It is given by

$$BTH(I) = C_A(I) - I. \tag{57}$$

The joint self-dual transform is given by

$$TH(I) = WTH(I) + BTH(I) = C_A(I) - O_A(I). \tag{58}$$

It contains all structures which can be observed using the respective structuring element A.
The white tophat transform returns that bright parts of the image which are smaller than the structuring element; see Figure 20. The same statement holds for black tophat transform for the dark parts. A typical application is morphological anti-shading. We remark that a tophat transform is often followed by a binarization in which case these operations may also be combined.

3.3 Distance Transforms

We consider a binary image I. The distance function derived from I is a mapping/ an image assigning each coordinate $x = (x_1, x_2)$ its distance to the next background pixel, i.e.,

$$D_I(x) = \min_{y \in A^c} d(x, y), \tag{59}$$

where $I = 1_A$, i.e., A represents the foreground in I and A^C its background. where d is a distance on the pixel grid whose choice is discussed next. Since every binary image has an associated distance function, the above mapping $I \mapsto D_I$ mapping an image to its distance function is frequently called *distance transform*.

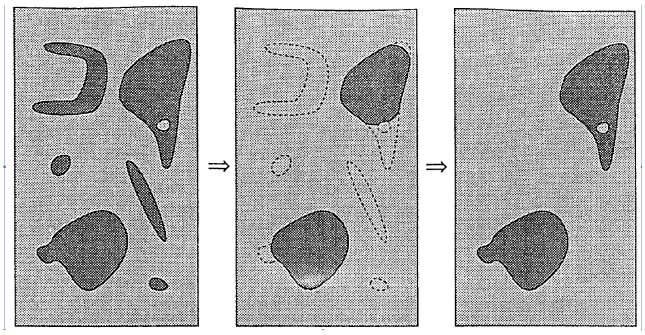


Figure 22: *left*. Binary image. *middle.* Opening using a ('large') disk as structuring elements. *right*. Opening by reconstruction (which is an erosion followed by geodesic reconstruction by dilation). (Source: Soille).

Examples of the distance d on the pixel grid employed in (59) are the euclidean distance

$$d_2(x,y) = \sqrt{\sum_i |x_i - y_i|^2},$$

and the city block or Manhatten metric

$$d_1(x,y) = \sum_i |x_i - y_i|;$$

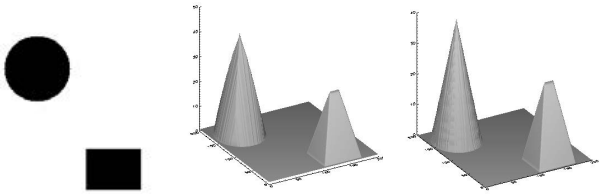


Figure 21: Distance transform. *left:* image as negative; *middle:* distance transform w.r.t. the Manhatten metric d_1; *right:* distance transform w.r.t. the (rounded) euclidean metric d_2. (Source: Sandau)

cf. Figure 21. The result of a distance transform obviously depends on the employed metric.
The connection with erosion is as follows: the distance map encodes the result of various erosions with balls of increasing radii (at least, approximately.) This is seen by thresholding the distance function; the structuring element is a corresponding ball in the employed metric. Hence, we may employ the distance function in order to perform erosions with balls of increasing radii. Using the negative image (i.e., changing foreground/background, black/white, $0/1$) the same statement applies to dilations.
Applications are in connection with segmentation based on the watershed algorithm (see below) and with forming skeletons [10].
Details on computing distances transform are given at the end of the section.

3.4 Geodesic Reconstruction

Using erosion small objects which are not intended to be reconstructed disappear. However, the remaining objects shall be reconstructed in original size. A means to accomplish this task is reconstruction by dilation. For a visualization, see Figure 22.

Geodesic reconstruction by dilation. We here first consider the binary case.

As an operation reconstruction by dilation is defined as follows: the input is a binary image (think of the original image) with a foreground set Y and a marker image (think of the eroded image) with foreground $M \subset Y$. We dilate M relative to Y using the unit disc A as structuring element to obtain

$$D^{1,Y}(M) = D_A(M) \cap Y,$$

and iterate this process

$$D^{k,Y}(M) = D_A(D^{k-1,Y}(M)) \cap Y, \quad k \in \mathbb{N}$$

until no new pixels are added. The result

$$R_Y(M) = \lim_{k\to\infty} D^{k,Y}(M)$$

is called *reconstruction by dilation* of M in Y. In a nutshell, we iterate dilating the marker set M relative to the foreground Y until stability.
We point out that the results of reconstruction by dilation using a 'unit disc' as structuring element just yields the corresponding connection components of Y which contain points of M. This can be used for computing the reconstruction by dilation.

Opening by reconstruction. Opening by reconstruction is defined as the concatenation of an erosion with reconstruction by dilation. For an illustration see Figure 22.

Reconstruction by erosion and closing by reconstruction. *(Geodesic) reconstruction by erosion* is dual to reconstruction by dilation. Here, the marker M is a superset of Y, and M is eroded in Y until stability. In symbols,

$$\bar{R}_Y(M) = \lim_{k\to\infty} E^{k,Y}(M),$$

where, for $k \in \mathbb{N}$,

$$E^{k,Y}(M) = E_A(E^{k-1,Y}(M)) \cup Y, \qquad \text{and} \qquad E^{1,Y}(M) = D_A(M) \cup Y.$$

Closing by reconstruction is defined as the concatenation of a dilation with reconstruction by erosion.

Gray scale images. Gray-scale morphological reconstruction is defined similarly as for binary images. We use the symbols f and g to denote the marker and the mask grayscale images respectively, and assume that $f \leq g$.
The geodesic dilation of size 1 of the marker f with respect to the mask g is defined as

$$D_A^{1,g}(f) = D_A(f) \wedge g \tag{60}$$

with the pixel-wise minimum operator $\wedge$. Accordingly, the geodesic dilation of size n of f with respect to g is given by

$$D_A^{n,g}(f) = D_A(D_A^{n-1,g}(f)) \wedge g, \tag{61}$$

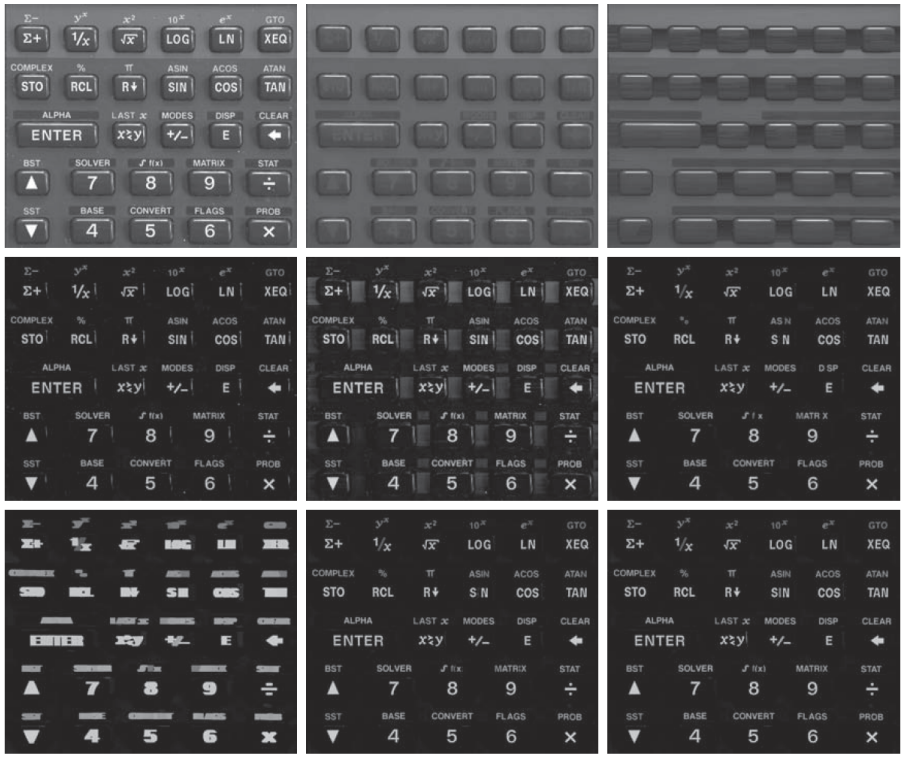


Figure 23: Flattening a complex background *Top, left to right.* Data (1134 × 1360), opening by reconstruction using a horizontal line of 71 pixels as a structuring element, opening of the original data using the same SE. *Middle, left to right.* Top-hat by reconstruction, top-hat transformation, Opening by reconstruction of the top-hat by reconstruction using a horizonal line of 11 pixels. *Middle, left to right.* Dilation of the middle right image using a horizontal line of 21 pixels, minimum of the right neighbor and right middle image, final reconstruction result. The horizontal reflections on top of the keys are suppressed by the tophat by reconstruction, i.e., subtracting the opening by reconstruction (employing a long horizontal line in the erosion operation) from the image. Further, variations in the background are suppressed. Vertical reflections on top of the keys are suppressed by an opening by reconstruction with a shorter horizontal line. In order to reconstruct strokes missing after the opening by reconstruction, the tophat by reconstruction is dilated and the minimum with the opening by reconstructionis taken. Then the result is employed as a marker for reconstruction by erosion with the dilated tophat by reconstruction as mask. (Source: Gonzales/Woods, Steve Eddins).

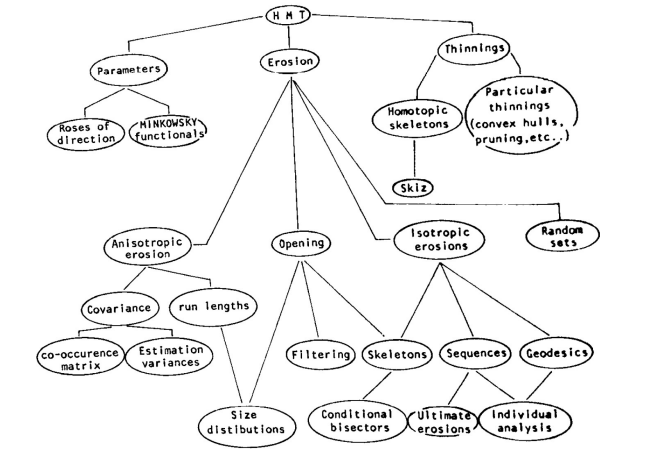


Figure 24: Hit-and-Miss transforms (Source: Serra).

and its stable limit as $n \to \infty$ is called *geodesic reconstruction by dilation* of the marker f w.r.t. the mask g, in symbols $R_g(f)$.
Geodesic reconstruction by erosion considers the dual case. Here, $f \geq g$ is assumed. Instead of iterated geodesic dilation of the marker f with respect to the mask g, iterated geodesic erosion of the marker f with respect to the mask g is applied until stability.
Closing by reconstruction and *opening by reconstruction* are defined similar to the binary case as dilation followed by reconstruction by erosion and as erosion followed by reconstruction by dilation, respectively.
(**How would you define a tophat by reconstruction?**)
An opening followed by a closing (or vice versa) performs a morphological smoothing. Also an opening by reconstruction followed by a closing by reconstruction performs a morphological smoothing (with frequently better results). For a visualization involving reconstruction we refer to Figure 23.

3.5 Hit and Miss Transforms

We here have a brief look at a more general class of transformations, the hit and miss transformations (HMT). This important class has many applications and it extends erosion and dilation, see Figure 24.
We consider two structuring elements A, B (think of A for the foreground and of B for the background) The *hit and miss transform* is defined as

$$Y \otimes [A, B] = \{x \in \mathbb{Z}^2 : A_x \subset Y, B_x \subset Y^C\} = E_A(Y) \cap E_B(Y^C). \qquad (62)$$

The name may be explained by the fact that the transform looks for that points x such that A_x hits Y and B_x misses Y in the sence that $B_x \cap Y = \emptyset$.
We first observe that ordinary erosion is obtained as the special case where B equals the empty set. Further if A and B are not disjoint, i.e., $A \cap B \neq \emptyset$, then the hit and miss transform is completely uninteresting, since then $Y \otimes [A, B] = \emptyset$.

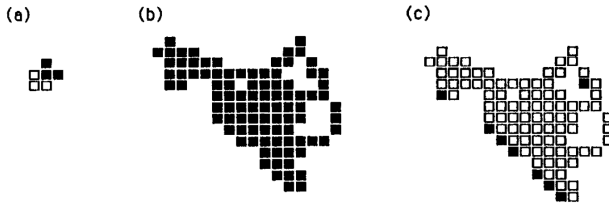


Figure 25: Hit and miss transform with the left hand structuring element yields the lower left boundary pixels. (Source: Sandau)

A first application is finding for instance only the lower left boundary pixels, see Figure 25.
A further direct applicaction is finding small holes of prescribed structure (such as, e.g., one pixel holes in a binary image. **Which structuring elements would you use to do this?**). A more advanced application is the determination of the Euler number for instance.

Thinning and thickening. We consider a pair of structuring elements $[A, B]$ together with a set Y. Then the thinning of Y using A, B is defined by

$$Y \circ [A, B] = Y \setminus (Y \otimes [A, B]). \tag{63}$$

The thickening of Y using A, B is defined by

$$Y \odot [A, B] = Y \cup (Y \otimes [A, B]). \tag{64}$$

The thinning and thickening operations are only interesting if A and B are disjoint, since otherwise $Y \otimes [A, B] = \emptyset$ as discussed above. Further, thinning is only interesting if the center $(0, 0) \notin B$ since otherwise $Y \circ [A, B] = Y$ as well (**Why?**). Similarly, thickening is only interesting if the center $(0, 0) \notin A$.
We further observe that the thinning operation is anti-extensive and that the thickening operation is extensive.
In order to deal with duality, we first observe

$$Y^c \otimes [A, B] = \{x : A_x \subset Y^C, B_x \subset Y\} = Y \otimes [B, A].$$

This implies

$$\begin{aligned} Y^c \circ [A, B] = Y^c \cap (Y^c \otimes [A, B])^c &= Y^c \cap (Y \otimes [B, A])^c \\ &= [Y \cup (Y \otimes [B, A])]^c = [Y \odot [A, B]]^c \end{aligned}$$

Hence, the dual of thinning is given by thickening with the structure elements interchanged.
A central application of thinning is skeletonization or homotopic thinning. Loosely speaking, the task is to thin the foreground in a binary image while keeping its topological structure, i.e., preserving connectivity, holes, etc. This can be done by iteratively using thinings with the masks depicted in Figure 26. More precisely,

$$HT(Y) = \Psi^\infty(Y), \quad \text{where} \quad \Psi(Y) = (Y \circ [A_1, B_1]) \circ \cdots \circ [A_8, B_8].$$

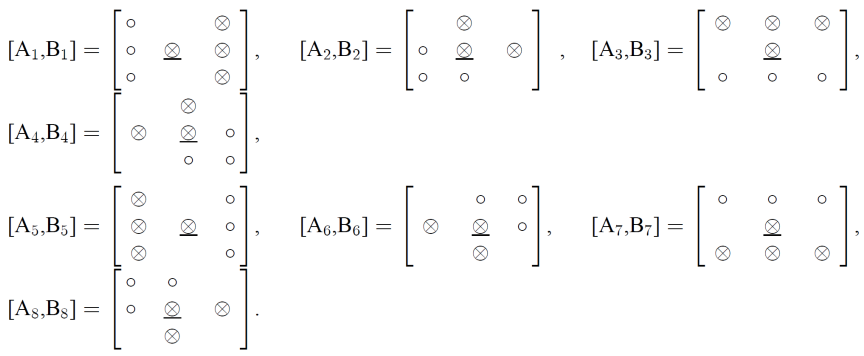


Figure 26: Masks for iterative thinning resulting in a skeleton.

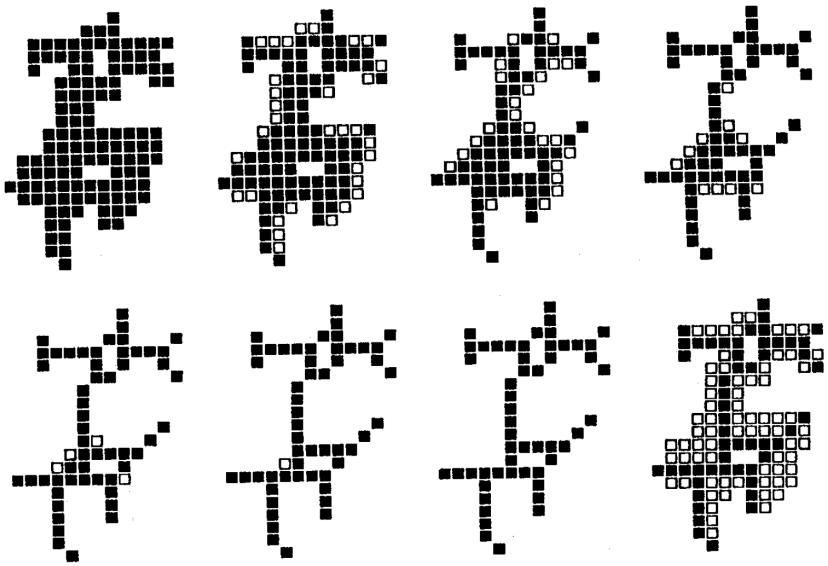


Figure 27: Visualization of the homotopic thinning using thinning with the structure elements of Figure 26. (Source: Sandau)

Figure 28: The watershed algorithm applied to the negative distance map is capable of seperating the connected foreground of the binary image into three segments.

The resulting algorithm is called Levialdi's algorithm. A visualization is given in Figure 27. Due to time and space reasons, we refer to [23] for a detailed account on skeletonization or homotopic thinning.

3.6 Regional Minima and Related Concepts

Frequently, objects in a binary image are overlapping; see for instance Figure 28, left. One means to partition them into disjoint regions is to consider the negative distance function of the foreground; cf. Figure 28, middle. Its regional minima –the regional maxima of the distance map – may be seen as a kind of centers of the segments. Suitable boundaries may then be detected using the watershed transform, cf. Figure 28, right. (For details on the watershed transform we refer to the next chapter.)
We here discuss regional minima and related concepts. We use the symbol regmax I to denote the regional maxima of I which are pixels or sets of pixels of equal value which have a gray value higher than their neighborhood.
There is a link to ultimate erosion which is derived as follows. When iterating erosion, i.e., considering

$$E_A^k(Y) := E_A(E_A^{k-1}(Y)),$$

the sets become successively smaller. Frequently, whole connected components vanish. When this happens, in say the $k+1$th erosion step, then $E_A^k(Y)$ and the part of the reconstruction $R_Y(E_A^{k+1}(Y))$ of the $k+1$th erosion in $E_A^k(Y)$ are not identical, but there is 'something' missing. This 'something' equals the vanishing connected components and can be written as $E_A^k(Y) \setminus R_Y(E_A^{k+1}(Y))$. Forming the union w.r.t. all dilation steps, we end up with

$$UE_A(Y) = \cup_{k\in\mathbb{N}_0} \left[E_A^k(Y) \setminus R_Y(E_A^{k+1}(Y))\right] \tag{65}$$

which is called the ultimate erosion of Y.
The connection with the distance map is given as follows.

$$UE_A(Y) = \text{regmax}\, D(Y) \tag{66}$$

where D denotes the distance map of the set Y, and regmax $D(Y)$ denotes regional maxima of $D(Y)$. As structuring element we use the unit disc A. A regional maximum of height h is a connected set of pixels with value h whose outer boundary only has values smaller that h.
Note that, by the use of different types of algorithms, the equality (66) holds only approximately in practice (which also explain that two symbols are employed.) However, the results are comparable.

Determining regional maxima. In order to algorithmically determine the regional maxima regmax I of a (gray value) image I we proceed as follows: we first form the image $I_2 = I + 1$; then, for each gray value g, we look which pixels $\{x : I(x) \leq g\}$ cannot be reached by reconstruction (by dilation) of $\{x : I_2(x) \leq g\}$. Doing this together for all gray values g, yields

$$\text{regmax } I = I_2 - R_{I_2}(I) \tag{67}$$

where R denotes the reconstruction (by dilation) for gray value images. The result is a 0-1 image with the 1's encoding the regional maxima.
Coming back to the ultimate erosion of a binary image, we conclude that its ultimate erosion can be computed by reconstruction using its distance map as image I in (67).

***h*-maxima.** Using a picture as analogy, every small stone on an absolutely flat plain part yields a regional maximum in the considered landscape. In order to deal with such 'not that interesting' regional maxima or to better quantify regional maxima, the term of a h maximum is employed.
A h-maximum of the image I is a connected region M fulfilling the following conditions: (i) the gray value is between m (the maximal value in M) and $m - h + 1$ for all pixels in M, (ii) the gray value is smaller than or equal to $m - h$ on the outer boundary of M. Similar to (67), h-maxima can be computed as

$$\text{h} - \max(I) = \bar{I} - R_{\bar{I}}(I), \qquad \text{with} \quad \bar{I} = I + h. \tag{68}$$

3.7 Computing Distance Transforms

Computing distance transforms efficiently is an important topic not restricted to morphology; see for instance [6]. We consider the discrete situation here and discuss two basic schemes to compute distance transfoms w.r.t. the Manhatten/ℓ^1 metric (together with generalizations) and w.r.t. the euclidean metric.

Distance transfoms w.r.t. the Manhatten metric and generalizations. The basis for the algorithm is the following observations: (i) In the Manhatten metric the distance to the central point from its eight neighbors is given by

$$M = \begin{pmatrix} 2 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 2. \end{pmatrix} \tag{69}$$

(ii) Crucial point: The distance between two coordinates equals the shortest path between the coordinates w.r.t. the eight neighborhood where horizontal/vertical steps cost 1 and diagonal steps cost 2. (Note that the Euclidean distance does not have this property.)

This observations can be used to derive an algorithm to compute the distance to the background (in the Manhatten metric) as follows:

- *Initialization:* Initialize the image y by setting background pixels of the image x by 0, and foreground pixels by ∞ (realized by a number larger than the largest distance in the image x).
- *Forward pass:* Pass from top left to bottom right and set

$$\begin{aligned} y(i,j) &= \min M + y(i-1:i+1, j-1:j+1) \\ &:= \min_{k,l\in\{-1,0,1\}} M(k,l) + y(k,l) \end{aligned} \tag{70}$$

  for any coordinate (i,j).
- *Backward pass:* Pass from bottom right to top left employing (70).

As an example, consider the univariate data (with 0 denoting the background and 1 denoting the foreground)

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1. \end{pmatrix}$$

The matrix M for the neighborhood distances is given by $(1,0,1)$. The initialization reads

$$\begin{pmatrix} 0 & 0 & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty \end{pmatrix}$$

After the forward pass, we have (extending the image by ∞'s at the boundaries)

$$\begin{pmatrix} 0 & 0 & 1 & 2 & 3 & 1 & 0 & 0 & 1 & 2 \end{pmatrix},$$

and, after the backward pass, we have the distance map

$$\begin{pmatrix} 0 & 0 & 1 & 2 & 2 & 1 & 0 & 0 & 1 & 2 \end{pmatrix}.$$

We consider a bivariate example for the binary input image

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

The initialization reads (instead of ∞, we use 40 which is Ok, since the maximal distance in the image is 14. (**Question: Why?**))

$$\begin{pmatrix} 0 & 0 & 40 & 40 & 40 & 40 & 0 & 0 & 40 & 40 \\ 0 & 0 & 40 & 40 & 40 & 40 & 0 & 0 & 40 & 40 \\ 0 & 0 & 40 & 40 & 40 & 40 & 40 & 40 & 40 & 40 \\ 0 & 0 & 40 & 40 & 40 & 40 & 40 & 0 & 0 & 0 \end{pmatrix}.$$

Outside the image, we assume foreground. Thus we extend the image by an outer ring of 40's. After the forward pass, we have

$$\begin{pmatrix} 0 & 0 & 1 & 2 & 3 & 1 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 2 & 3 & 1 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 2 & 3 & 2 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 & 3 & 3 & 1 & 0 & 0 & 0 \end{pmatrix}$$

and, after the backward pass, we get the distance map

$$\begin{pmatrix} 0 & 0 & 1 & 2 & 2 & 1 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 2 & 2 & 1 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 2 & 3 & 2 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 & 3 & 2 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

Generalizations. Instead of starting with the metric, here the Manhatten metric, a means to define a distance/metric is to define a matrix M for the neighbors of the form

$$M = \begin{pmatrix} b & a & b \\ a & 0 & a \\ b & a & b \end{pmatrix} \tag{71}$$

first. Then the distance is defined as the shortest pass length w.r.t. this M, i.e. diagonals are measured by b and horizontal and vertical steps by a. A prominent example is the Chamfer distance given by the matrix

$$M = \begin{pmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & 0 & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{pmatrix}. \tag{72}$$

Note that this distance does not equal the euclidean distance but provides an approximation of it. However, using the approximation has the advantage of higher computational speed (see next paragraph.)

Distance transfoms w.r.t. the euclidean metric. The crucial point (ii) used above does not apply to the euclidean distance. Hence, the fast scheme above does not apply. Nonetheless, a more expensive forward and backward scan algorithm still works.

- *Initialization:* Initialize the image y by setting background pixels of the image x by 0, and background pixels by ∞ (realized by a number larger than the square of the largest distance in the image x).
- *Linewise forward and backward pass:* Compute the linewise squared euclidean distances to the background by (i) first passing through the foreground from left to right setting $1, 4, 9, 16, 25, \ldots$ in the foreground until we meet a background pixel; (ii) do the same from right to left and take the minimum. Store the result in y.
- *Columnwise forward and backward pass:* The target euclidean squared distance e is obtained from y by

$$e_{i,j} = \min_k y_{k,j} + (k-j)^2. \tag{73}$$

  We point out that this is the time critical part: a naive approach would require a column scan for each pixel resulting in full cubic runtime. A speed up can be gained by the following mononicity property we can use for pruning: if we are in the ith line and have computed a minimizing k'

for line $i-1$, and $k \in \{0,\dots,i-1\}$ we can start the search at k' for line line i, and $k \in \{0,\dots,i\}$, i.e.,

$$\min_{k=0,\dots,i} y_{k,j} + (k-j)^2 = \min_{k=k',\dots,i} y_{k,j} + (k-j)^2. \tag{74}$$

(**Question: Why?**) This observation can be cast into a forward and backward pass algorithm. For each column with index j pass from top to bottom w.r.t. the line index i, storing the optimal index k' : compute the right hand side of (74) and store it in $e_{i,j}$ and update k'.

The backward pass does the same where here $k' > i$, and the analogue of (74) is given by

$$\min_{k\geq i} y_{k,j} + (k-j)^2 = \min_{k=i,\dots,k'} y_{k,j} + (k-j)^2. \tag{75}$$

The result is compared with the minimum already stored in $e_{i,j}$ and the overall minimum is written into $e_{i,j}$ which yields the squared euclidean distance map.

We reconsider the example

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

from above. The initialization reads (instead of ∞, we use 400 which is OK, since the maximal distance in the image is 14. (**Question: Why?**))

$$\begin{pmatrix} 0 & 0 & 400 & 400 & 400 & 400 & 0 & 0 & 400 & 400 \\ 0 & 0 & 400 & 400 & 400 & 400 & 0 & 0 & 400 & 400 \\ 0 & 0 & 400 & 400 & 400 & 400 & 400 & 400 & 400 & 400 \\ 0 & 0 & 400 & 400 & 400 & 400 & 400 & 0 & 0 & 0 \end{pmatrix}.$$

Outside the image, we again assume foreground. After the linewise forward and backward pass, we have

$$\begin{pmatrix} 0 & 0 & 1 & 4 & 4 & 1 & 0 & 0 & 1 & 4 \\ 0 & 0 & 1 & 4 & 4 & 1 & 0 & 0 & 1 & 4 \\ 0 & 0 & 1 & 4 & 9 & 16 & 25 & 36 & 49 & 64 \\ 0 & 0 & 1 & 4 & 9 & 4 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

And after the columnwise forward and backward pass we have

$$\begin{pmatrix} 0 & 0 & 1 & 4 & 4 & 1 & 0 & 0 & 1 & 4 \\ 0 & 0 & 1 & 4 & 4 & 1 & 0 & 0 & 1 & 4 \\ 0 & 0 & 1 & 4 & 5 & 2 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 4 & 8 & 4 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

3.8 Concluding Remarks.

As a reference discussing further morphological topics in image processing we refer the interested reader to the book [10] and, for a more mathematical treatment, to [23].

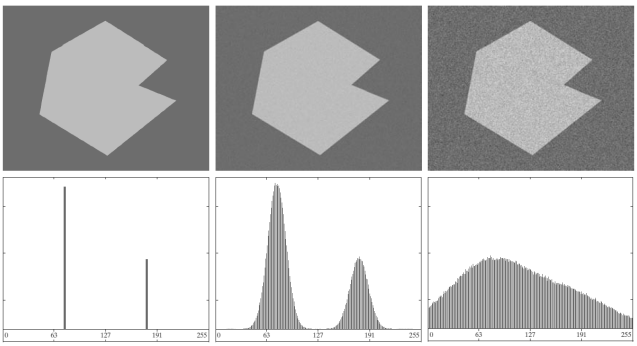


Figure 29: Histograms and noise. (Source: Gonzalez/Woods.)

4 Segmentation Algorithms

A basic step in image processing is the segmentation of images in regions whose pixel belong together w.r.t. some criterium.
We may distinguish binary segmentation from multilabel segmentation. Binary segmentation tries to segment an image (binary, gray scale, color,...) into two regions – say the foreground/ object of interest and background. Multilabel segmentation tries to segment the image into more than two parts; here one frequently distinguishes between the case where the number of labels has to be prespecified and the case where no a priori number of labels has to be imposed. Another means w.r.t. which one may try to categorize segmentation approaches is whether they are supervised or unsupervised, i.e., whether labeled training data is available or not.
There is a vast amount of different techniques originating from various different branches of mathematics. We here discuss a few basic approaches frequently applied in the image processing context. References for further discussion of segmentation techniques are [14, 10].

4.1 Thresholding

Thresholding is a first approach for segmentation. In its basic form, it binarizes an image I according to whether $I(x) < T$, and $I(x) \geq T$, i.e., a segment Y is generated by

$$Y = \{(x_1, x_2) : I(x_1, x_2) \geq T\}. \tag{76}$$

The underlying assumption is that object and background occupy a different range of gray-values. Besides applying thresholding as a stand-alone segmentation technique, thresholding is applied as a step in many segmentation pipelines (for e.g., pre- or postprocessing) or binarization routines. A simple example: referring back to the lecture 'Bildverarbeitung' the local adaptive binarization a la Berensen may be interpreted as global thresholding on the image obtained from subtracting its Beucher gradient image (with a sufficiently large disc as structuring element.) Further, when noise is present, see for instance Figure 29 for an illustration, at least some preprocessing or more sophisticated methods are necessary.

When performing *global thresholding* the thresholds are applied to a whole image. Hence, no information on the regional structure is used. In order to avoid effects from local illumination changes etc. thresholding is frequently applied locally, i.e., the image is partitioned into subimage, and on each subimage, thresholding is applied.
We first consider binary global thresholding. For binary segmentation, one threshold is sufficient. Finding a suitable threshold is the major challenge. One means to find a threshold is to "stare at the histogram and to guess T" (which frequently works well in a supervised setting.) Finding the threshold automatically requires a rule: one means to define such a rule is to define a target function f, such as the intra-class variance, and to find a threshold which is optimal w.r.t. this target function. More precisely, for each t in the pixel range, we threshold the image I using the threshold t to obtain the binarized image I_t consisting of two objects Y_t and Y_t^c. The classes C_t, C_t^c are the gray values in Y_t and in Y_t^c, respectively. (They don't have to be computed from the image, they are already encoded in the histogram.) For these classes we can compute the (weighted sum of) intraclass variances

$$\mathrm{ICV}(C_t, C_t^c) = |Y_t|\mathrm{var}(C_t) + |Y_t^c|\mathrm{var}(C_t^c), \tag{77}$$

where $|Y_t|$ and $|Y_t^c|$ denotes the number of pixels in the correspond class (which may be read of the histogram) and $\mathrm{var}(C_t)$ and $\mathrm{var}(C_t^c)$ denote the variances within the classes.
Then the target function f defined on the gray values is given by

$$f(t) = \mathrm{ICV}(C_t, C_t^c). \tag{78}$$

This function can be computed for all t in the gray range (extensive search) and the minimal t can be chosen:

$$T = \mathrm{argmin}_t\, f(t). \tag{79}$$

The way choosing the threshold is called *Otsu's rule.* The method is optimal in the sense that it maximizes the between-class variance as well. The functional is well-known in linear discriminant analysis (LDA) and actually the rule arises as a special case of LDA in 1d. Another interpretation is: it performs 2-means clustering on the histogram. A straight forward extension to more than two classes, is to employ k-means clustering on the histogram to obtain $k-1$ thresholds.
Finding a suitable threshold is a rather pronounced topic in the literature, we refer to [22] for an overview; also see the documentation of scikit image for instance.
We point out that for adaptive thresholding the image may be divided into subimages and a separate threshold for each sub-image may be computed automatically (using e.g. Otsu's or any other rule) to better address local features. Concerning limitations, see Figure 30.

Preprocessing. Next, we point out preprocessing steps. As illustrated in Figure 29, noise can seriously complicate thresholding problems or even make them unsolvable. When thresholding is the method of choice, smoothing the image prior to thresholding can improve the result; see Figure 31. Further, if

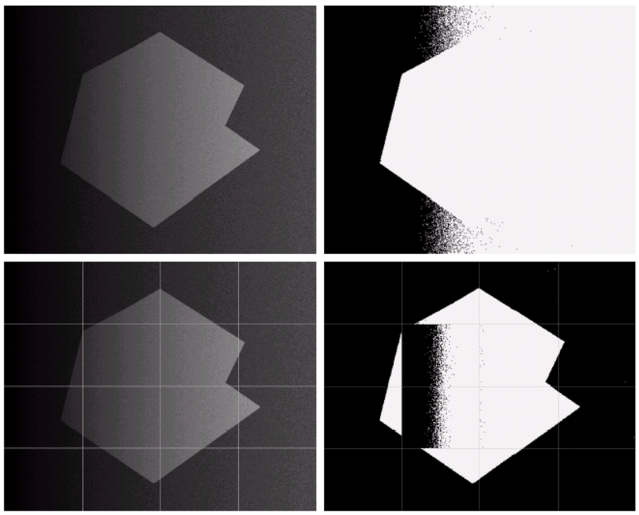


Figure 30: Global thresholding (*top*); Adaptive thresholding (*bottom*). The image is divided into subimages and Otsu's rule is applied to each subimage seperately (Source: Gonzales/Woods).

there are too many background pixels one approach for improving the shape of histograms is to consider only those pixels that lie on or near the edges (which are assumed to be the boundaries between regions; for detection the application of an edge detector is typically necessary.) Further preprocessing topics include shading correction, e.g., using a tophat filter; for further details see [10].

Hysteresis thresholding. Finally, we mention the concept of Hysteresis thresholding. We consider binary segmentation from a gray-range image I and assume that the foreground is dark. We employ two thresholds T_1, T_2 in the following way: If $I(x_1, x_2) < T_1$ the pixel belongs to the foreground, and if $I(x_1, x_2) > T_2$ the pixel belongs to the background. If the gray value $I(x_1, x_2)$ is between T_1 and T_2, and if a neighboring pixel belongs to the foreground, then (x_1, x_2) belongs to the foreground as well; otherwise, it belongs to the background.

4.2 Segmentation based on Clustering

We may interpret global thresholding as a means of clustering using the histogram and thus neglecting any spacial structure or information. That this may not always be a good idea has been observes in various places. In particular, incorporating denoising and edge detection constitute means to incorporate spacial neighborhood structure. Further approaches aiming at incorporating locality include adaptive thresholding by partitioning the image or by employing sliding windows.
We here discuss another rather successful technique: clustering on the image graph. Here, the discrete graph of an image I consisting of triples of the form $p_{ij} = (i, j, I(i, j))$ is considered as point cloud in 3d, where $I(i, j)$ denotes the

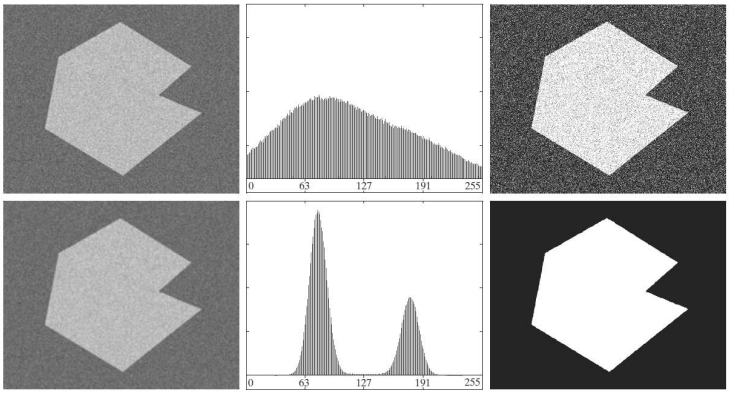


Figure 31: Thresholding using Otsu's rule without and with denoising with a 3×3 moving average filter (Source: Gonzales/Woods).

grey value for a grey value image, (or, in 5d for color images, where $I(i,j)$ denote three color channels.) Then a clustering procedure such as k means clustering is employed. (In principle, one may take any clustering procedure.) We consider k means clustering and try to find k class means m_l, $l = 1, \ldots, k$ which minimize

$$\sum_{i,j} \min_{l=1,\ldots,k} \| p_{ij} - m_l \|^2. \tag{80}$$

Given minimal means m_l the corresponding lth segmentation is obtained by assigning pixel p_{ij} with class l^* if m_{l^*} is that mean which minimizes the above expression, i.e,

$$l^* = \text{argmin}_{l=1,\ldots,k} \| p_{ij} - m_l \|^2. \tag{81}$$

Compare this definition with that of Otsu's rule to convince yourself that the latter defines a particular instance. Concerning the norm $\| \ \|$ applied above we notice that the spacial components and the intensity components have no natural geometric relation such that it is reasonable to apply a weighted square norm

$$\| (i,j,I_{ij}) \| = \sqrt{\alpha(i^2+j^2) + \beta I(i,j)^2}. \tag{82}$$

with parameters α, β weighting spacial and intensity distance. Unfortunately (80) is a NP hard problem (i.e., it is not computable with present computers for larger problem sizes). However, algorithms producing approximate solutions exist:

1. Initialization: Choose m_l, $l = 1, 2, \ldots, k$.
2. Assign data points p_{ij} to cluster C_{l^*} via (81).
3. Update means as new centers of their cluster, i.e.,

$$m_l = \sum_{(i,j) \in C_l} p_{ij}, \qquad l = 1, 2, \ldots, k, \tag{83}$$

   where the sum is taken w.r.t. the members of the corresponding cluster C_l.

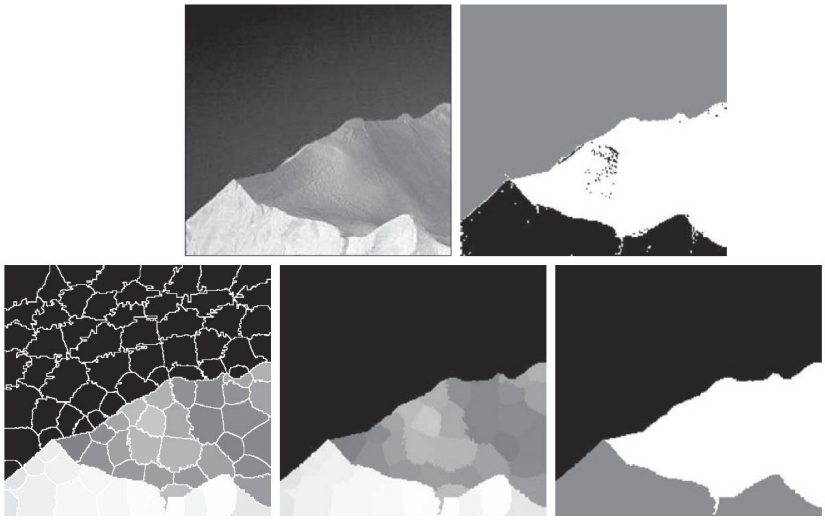


Figure 32: *From top left to bottom right:* data, 3-means, SLIC using 100 superpixels, superpixels without boundary, 3-means applied to the superpixel image. (Source: Gonzales/Woods)

4. Test for completion: Compute the norms of the differences between the mean vectors (centers) in the current and previous steps, and then denote the sum of these k norms by E. If the centers show sufficiently small change, i.e., if $E \leq t$, with a specified small threshold t, STOP. Else, GOTO Step 2.

Clustering methods are particularly suited to deal with super pixels: instead of pixels $i, j, I(i, j)$ the data fed to the clustering algorithm are already super pixels represented by a data point (x, y, z) (which may be a local average within the super pixel which was already computed) together with a potential weight of the superpixel.
One means to generate superpixels is called simple linear iterative clustering (SLIC). Here, a number n of super-pixels (significantly larger than the number of clusters finally aiming at) is generated. In the beginning the n centers of super-pixels are chosen in a nearly equi-spaced way on the pixel domain. Similar to k-means clustering then image pixels are assigned to superpixels and super pixel centers are updated; the major difference is that for the assignment of image pixels to a superpixel only pixels in a certain neighborhood of the center are allowed. For precise details, we refer to [10].
A possible segmentation pipeline is given by applying SLIC to generate superpixels followed by k-means clustering to obtain a segmentation; see Figure 32 for a visualization.

4.3 Segmentation based on Edge Detection

We consider means of segmentation based on edge detection. In such a pipeline, typically, first an edge detector is applied. The edges are interpreted as segment boundaries. The segments are therefore obtained by filling the areas encircled by edges. Problems arise when edges are broken due to low gradient strength;

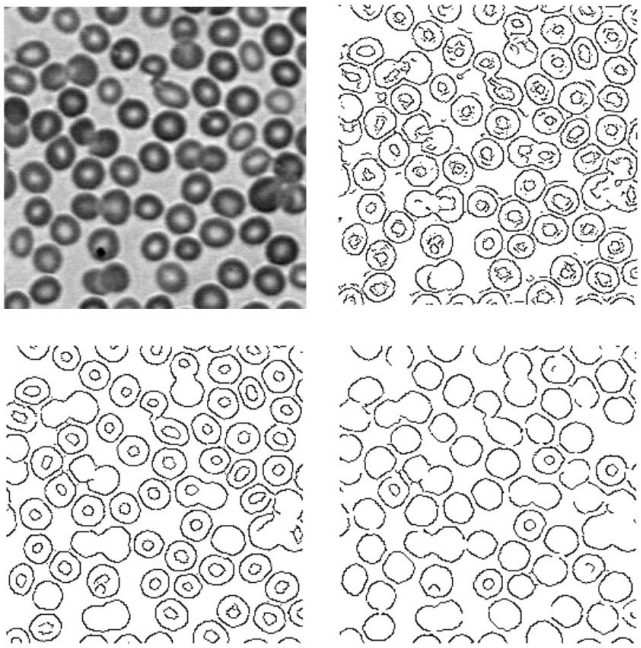


Figure 33: Canny edge detection for different smoothing parameters $\sigma = 0.6$, $\sigma = 1.5$, and $\sigma = 2.3$.

then, some edge linking is employed before filling. The steps are (i) edge detection, (ii) edge linking and filling.
We start with edge detection. As an edge detector we here employ the Canny edge detector. (For a discussion of edge detection including further edge detectors such as the Marr-Hildreth edge detector, we refer to [10].)
The Canny edge detection algorithm is composed of three steps:

1. Calculation of the gradient of the smoothed image;
2. non-maximum suppression;
3. hysteresis thresholding.

For a visualization, see Figure 33
Concerning the gradient calculation, one form is to apply the Sobel filter which yields a (slightly smoothed) difference image in $x-$ and $y-$direction. Instead, and also for more smoothing, we may apply filtering with a sampled gradient of the Gaussian kernel G' in x-direction while smoothing with a Gaussian G in y-direction to obtain the x-component of the smoothed gradient ∇I_x. For the y-component ∇I_y of the smoothed gradient we proceed analogously be interchanging G' and G.
Equipped with the smoothed (discrete) gradient image ∇I of I, we consider the pixelwise polar coordinates, i.e., gradient magnitude ∇I_r and gradient direction ∇I_φ. Nonmaximum suppression then works as follows. Consider a pixel coordinate (i, j) together with gradient direction $\nabla I_\varphi(i, j)$. This yields two neighbor pixels in gradient direction and opposite. If one of this pixel has a

higher gradient magnitude ∇I_r, then the pixel (i, j) is considered a being no maximum and is suppressed by setting the coordinate to 0.
As a final step of edge detection the obtained non-maximum suppressed gradient image is thresholded by using hysteresis thresholding as described in the subsection on thresholding. The remaining binarized foreground set is the output of Canny edge detection. Canny's experiments suggest a ratio of the upper to the lower threshold in the range of $2:11$ to $3:1$.
It may happen that, due to low gradient strength, frequently

4. linking broken edges

is necessary. We here briefly explain the idea of a basic local linking procedure to give the reader an impression: considering open ends of the Canny edge set and choosing a neighborhood (e.g., a disk with a five pixel radius), we may search the neighborhood of an open end for pixels with similar gradient strength and in particular edge direction (differences smaller than a threshold) and then, when having found two pixels, we may link them. There are efficient refinements and speed-ups for local linking procedures and global approaches based on the Hough/Radon transform. For a more elaborate discussion we refer to [10]. Please note that this step is crucial for obtaining a good segmentation.
Now, to complete the pipeline, we perform

5. filling the complement of the edge set and potentially labeling it

if necessary.

4.4 Segmentation based on the Watershed Transform

We consider the image, more precisely the graph of the image as a mountain range. Then the points in this landscape can be divided into three categories:

- local/regional minima (cf. the discussion in the previous section),
- watershed lines/ ridge lines: points where water may run-off to more than one minimum.
- points of catchment basins: if water is placed on these points it will run-off to exactly one minimum.

We are interested in finding watershed lines as well as catchment basins. We may think of the watershed lines as the segment boundaries, and of the catchment basins as the segments.
The watershed algorithm may be applied to different segmentation maps: examples are the negative distance map of a binary image (see Figure 28) or the (smoothed) gradient image of a gray value image (see Figure 36).

Watershed algorithm based on flooding. We explain the idea of a flooding algorithm for performing the watershed transform. As support, consider the schematic Figure 34 to get a first impression.
To formalize the flooding, we need the level sets

$$T_n(I) := \{(x_1, x_2) : I(x_1, x_2) \leq n\} \tag{84}$$

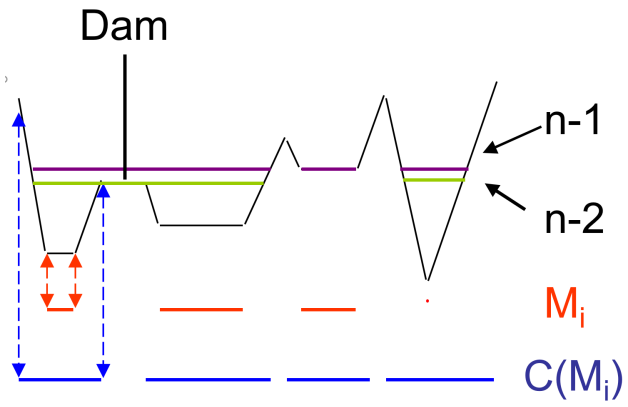


Figure 34: The level sets $T_{n-1}(I)$ and $T_{n-2}(I)$ are shown in violet and green, respectively

of those coordinates with gray value smaller than or equal to n. Obviously, increasing n corresponds to successive flooding.
We will also need the regional minima M whose connected components are denote by M_i. Note that the value of the respective M_i is also important, since the value determines when the minimum is flooded.
Further we need notation for the catchment basis we want to compute: we denote by $C(M_i)$ the catchment basin associated with the regional minimum M_i (which we do not know, but which we want to compute). Actually, we will explain how we obtain

$$C_n(M_i) = C(M_i) \cap T_n \tag{85}$$

which is the part of the catchment basin of the regional minimum M_i flooded at level n from the $C_{n-1}(M_i)$ from one flooding level below.
Starting from the lowest gray value n_0 in the image we start flooding. The $C_{n_0}(M_i)$ are just the M_i which have value n_0.
Now assume we have constructed the $C_{n-1}(M_i)$, i.e., we have partitioned the level set $T_{n-1}(I)$ into catchment basins w.r.t. the regional minima M_i. We explain how to construct the $C_n(M_i)$ and consider the level set $T_n(I)$. We may categorize the connection components of $T_n(I)$ as follows:

- There is a regional minimum (set) M_i with gray value n. Then, each M_i with gray value n is a (new) connected component of $T_n(I)$ (**why?**) and we have $C_n(M_i) = M_i$ as a start for this new catchment basin. We may subtract these M_i from $T_n(I)$ and consider the remainder

$$Y_n = T_n(I) \setminus \cup\{M_i : I = n \text{ on} M_i\}. \tag{86}$$

- A connection component of $T_n(I)$ contains precisely one M_i. Then this connection component equals $C_n(M_i)$.

- A connection component of $T_n(I)$ contains more than one connection component of $T_{n-1}(I)$, and hence more than one of the M_i's. Then, dams are constructed and corresponding ridges/watersheds emerge. More precisely, we take the geodesic zones of influence of the $C_{n-1}(M_i)$ as $C_n(M_i)$.

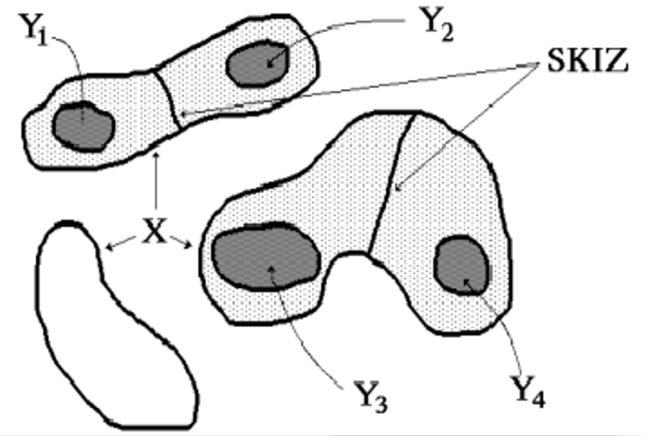


Figure 35: The skeleton of influence zone (SKIZ) of the marker set $Y = \cup_i Y_i$. The dotted areas depict the influece zones of the respective Y_i. (Source: Beucher /Meyer)

The corresponding skeleton of influence are the ridges. We elaborate on this concepts right below.

We have computed all $C_n(M_i)$, and may proceed to compute the $C_{n+1}(M_i), \ldots$ inductively. When the index k is larger than the highest gray value, we have computed all $C_k(M_i) = C(M_i)$ which was the task.

Geodesic zones of influence and skeleton of influence. Consider l connected components Y_i, $i = 1, \ldots, l$ all contained in a larger set X. (Above, the Y_i are the flooded parts of the catchment basin $C_{n-1}(M_i)$, and the set X is the connected component of $T_n(I)$ containing $C_{n-1}(M_i)$.)
The geodesic zone of influence $IZ_X(Y_i)$ of Y_i in X is the set of points x in X whose finite distance is closest to Y_i (among all components), i.e.,

$$d_X(x, y_i) < d_X(x, y_k), \quad \text{for all } k \neq i. \tag{87}$$

The boundaries between the various zones of influence give the geodesic skeleton by zones of influence (SKIZ) of the Y_i in X. For a visualization, see Figure 35

Further approaches. The watershed algorithms can be divided into different groups. The first group consists of algorithms simulating the flooding process. The idea of flooding was introduced by Beucher and Lantéjoul in 1979 and developed further in the following years. A result is Beucher and Meyer's flooding algorithm of the early 1990s [1]. Since then a number of improvements has been made to this type algorithms. A second group of watershed algorithms consists of procedures aiming at direct detection of the watershed lines; see for instance [5].

Dealing with oversegmentation. The oversegmentation which is produced by directly using the watershed transform on the gradient image (see Figure 38)

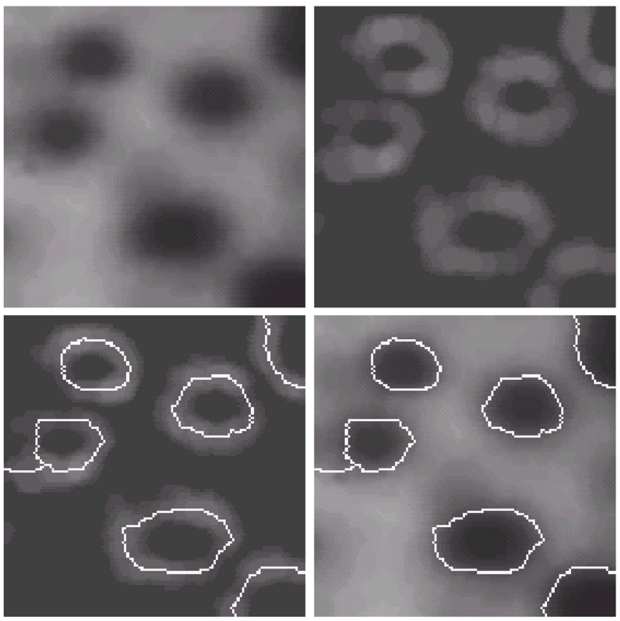


Figure 36: The watershed transform applied to the gradient of a toy image (top left) directly: The watershedline in the gradient as well as in the original image are shown at the bottom. (Source: Beucher /Meyer)

stems from the fact that every regional minimum becomes the center of a catchment basin. Some of them are produced by noise and others by minor structures in the image which may be unimportant for the task to address. Hence, a first important task is to find the 'important' regional minima. Having found them, one may proceed as follows.

(i) Given a marker set M, modify the (gradient) image G such that the regional minima of the modified (gradient) image $\bar{G}$ are contained in the previously defined marker set M. $\bar{G}$ is similar to G but with regional minima contained in M.

(ii) Apply the watershed transform to $\bar{G}$. Then there are only catchment basins corresponding to points which are regional minima of $\bar{G}$ and which are thus all contained in M.

When choosing M as a subset of regional minima of G, the catchment basins correspond to a subset of the regional minima of G. However, we may want M to contain additional points which is possible as we will see in the following.) We start out to explain means to realize (i) above. For an illustration, see (37). We use the marker set to define the marker image G' by

$$G'(x_1, x_2) = \begin{cases} 0, & \text{if } (x_1, x_2) \in M, \\ \max(G) + 1, & \text{else.} \end{cases} \tag{88}$$

where $\max(G)$ is the maximal gray value in G. Then, we perform reconstruc-

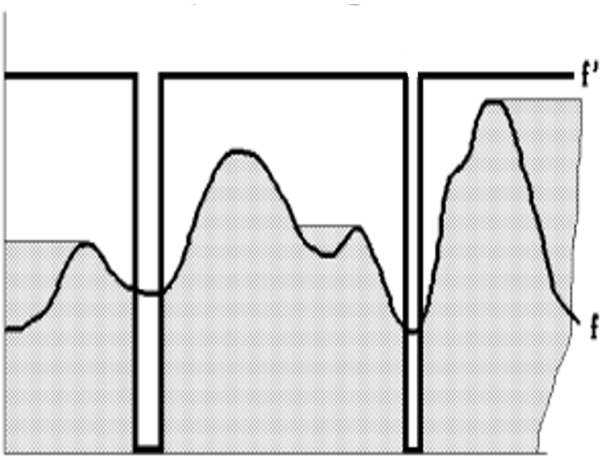


Figure 37: The marker function f' is generated from a marker set M by letting $f'(x) = 0$ if $x \in M$ and ∞ else. Starting with f' as marker and $f' \wedge f$ as mask we perform reconstruction by erosion to obtain the topology modification $\bar{R}_{f \wedge f'}(f')$ (dotted). (Source: Beucher/Meyer)

tion by erosion with marker G' and mask $G \wedge G'$ to obtain $\bar{G}$, i.e.,

$$\bar{G} = \bar{R}_{G \wedge G'}(G'). \tag{89}$$

This transformation is called *homotopy modification*.
We emphasize that the watershed transform based on flood filling such as described before may be modified such that it performs step (i) and (ii) above jointly. This means the user supplies G and a marker set M, and the algorithm produces the watersheds and catchment basins of the reconstruction $\bar{G}$. To this end, the algorithm is modified as follows: Instead of the regional minima, the marker set M is used. We start with the initialization

$$C_{-1}(M_i) = M_i, \tag{90}$$

where M_i is the label i of the marker image M. Then, the $(n+1)$th level basin of marker label i is given by

$$C_{n+1}(M_i) = IZ_{T_I(n+1) \cup M}(C_n(M_i)), \tag{91}$$

i.e., the geodesic influence zone of $C_n(M_i)$ on flood level $n+1$. This has the following effects: The catchment basins of minima which are not marked are filled up by overflowing of the neighboring catchment basin as soon as the water reaches the saddle point between both basins, the water rushes through the pass and fill the previously empty basin and no dam is constructed between these two basins. Further note that a dam is only constructed if water from the raising flood has different label/color.

Finding suitable markers is essential for the quality of the resulting segmentation. However, the task of finding adequate markers depends on the problem and has not general best solution.

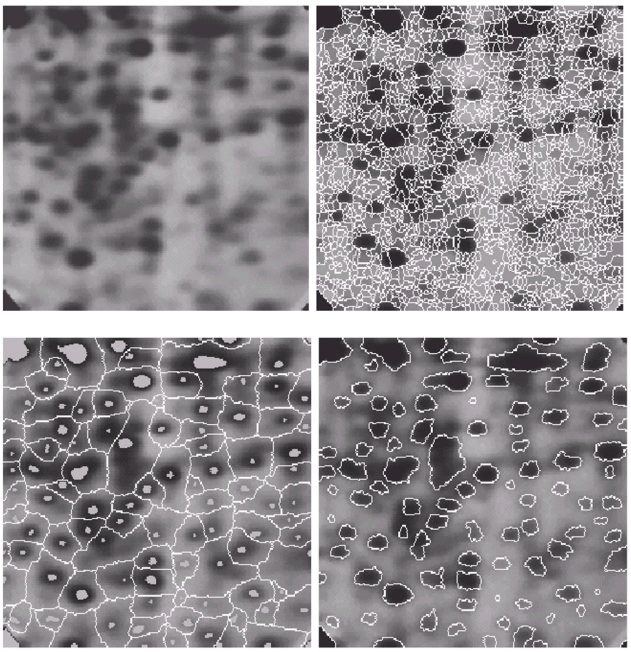


Figure 38: Direct application of the watershed transform yields oversegmentation (top left). Using a marker set (bottom left) improves the segmentation result (bottom right.) (Source: Beucher/Meyer)

We point out some possibilities. Often the image is smoothed, e.g., by applying an opening by reconstruction followed by a closing of reconstruction. If the objects of interest are bright, then local extrema of the resulting smoothed image are extracted. H-maxima/h-minima are also a possibility. Thresholding is a further possibility. The resulting sets are regularized further, e.g., by erosion and opening. The output can be used as a foreground marker. We note that typically, one distinguishes background markers and foreground markers (Note that the algorithm does not distiguish them.) Background markers may then be obtained using the SKIZ of the foreground markers for instance. With the marker set consisting of foreground and background markers, we may apply the Watershed transform.
We point out that a present trend in the literature is to learn the markers for the watershed transform.

4.5 Concluding Remarks

We have not given a complete overview on segmentation methods here. Instead, we have discussed a few methods, and one in more detail. In particular, we have not considered variational methods including graph cuts, the Chan-Vese model or other active contour methods. A more complete treatment can be found in [10] or in [2].

5 Linear Transformations

We here consider algorithms to compute the Fourier and the Radon transform. In particular, we consider the Cooley-Tukey algorithm for the fast Fourier transform and discuss rearranging the data for an implementation avoiding explicit recursion. A related scheme will be revealed in connection with the Götz-Druckmüller Radon transform.
Some scientific communities extensively discuss differences between the Radon and the Hough transform. We here notice that, from an algorithmic side, both the discrete classical Radon and the Hough transform for lines in 2D (classical image) compute the sums of image pixels w.r.t. straight lines.

5.1 Fast Fourier Transform

Recall that the Fourier transform computes a representation of a signal in terms of coefficients w.r.t. trigonometric poylnomials. In formulae, the Fourier transform of the signal f of length N is given by

$$\hat{f}_k = \sum_{l=0}^{N-1} e^{-2\pi i \frac{lk}{N}} f_l. \tag{92}$$

If the signal has even length $N = 2n$ we may rewrite

$$\begin{aligned} \hat{f}_k = \sum_{l=0}^{N-1} e^{-2\pi i \frac{lk}{N}} f_l &= \sum_{l=0}^{2n-1} e^{-2\pi i \frac{lk}{2n}} f_l \\ &= \sum_{m=0}^{n-1} e^{-2\pi i \frac{mk}{n}} f_{2m} + e^{-2\pi i \frac{k}{2n}} \left(\sum_{m=0}^{n-1} e^{-2\pi i \frac{mk}{n}} f_{2m+1} \right). \end{aligned} \tag{93}$$

The first sum on the right-hand side equals the Fourier transform of the even-indexed part of f denoted by $f^0 = (f_0, f_2, f_4, \ldots, f_{2n-2})$. Let us consider the second sum. For $k < n$ we identify the Fourier transform of the odd-indexed part of f denoted by $f^1 = (f_1, f_3, f_5, \ldots, f_{2n-1})$. If k is such that $n \leq k < 2n$, we have $\sum_{m=0}^{n-1} e^{-2\pi i \frac{mk}{n}} f_{2m+1} = \sum_{m=0}^{n-1} e^{-2\pi i \frac{m(k-n)}{n}} f_{2m+1}$, and $e^{-2\pi i \frac{k}{2n}} = -e^{-2\pi i \frac{k-n}{2n}}$. Hence, we may compute the Fourier transform of the signal f of length $2n$ by the Fourier transforms of its even indexed part f^0 and its odd indexed part f^1 of half length by

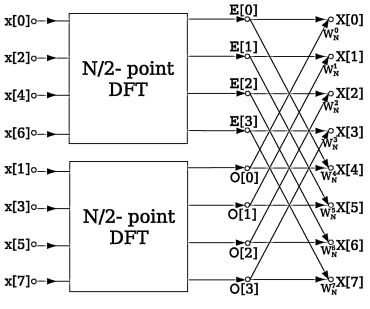


Figure 39: Schematic for the last step of the FFT for input x. (Source: Wiki).

$$\begin{aligned} \hat{f}_k &= \widehat{f^0}_k + e^{-2\pi i \frac{k}{2n}} \widehat{f^1}_k, \qquad \text{for} \quad k = 1, \ldots, n, \\ \hat{f}_{n+k} &= \widehat{f^0}_k - e^{-2\pi i \frac{k}{2n}} \widehat{f^1}_k, \qquad \text{for} \quad k = 1, \ldots, n. \end{aligned}$$

Hence, the cost $\phi(2n)$ of computing the Fourier transform for a signal of length $2n$ is given by

$$\phi(2n) = 2\phi(n) + \tau n,$$

where τ is the constant amount of effort given by a complex multiplication and two additions. Thus if $N = 2^r$ for some r we may successively apply the recursion to obtain an $O(N \log N)$ algorithm.
To avoid a recursion there are some means to resort the input data according to reverse bit ordering of the indices; cf. Figure 40. More precisely, consider $N = 2^k$, then any data item is indexed by k digits. We rearrange the data such that, in the lth position of the rearanged data, we find the data item with the reverse of the bit-expansion of l as index. Then, we may apply the scheme for increasing size blockwise.
The above variant is called Cooley–Tukey algorithm. It works best for $N = 2^k$. There are various FFT variants which have been developed (to deal with different data length including prime numbers.) For details we refer the interested reader to [7]. There, implementations are discussed and various references are given.

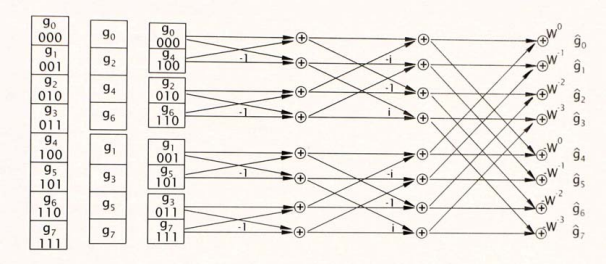


Figure 40: Schematic of organizing FFT non-recursively (Source: Jähne.)

Finally, we note that the inverse Fourier transform is given by the Fourier transform followed by a reflection and scaling by a factor of N. Therefore, the inverse Fourier transform can be realized similarly.

5.2 The Radon Transform

We first recall the concept of a Radon transform in 2D. On the one hand side, Radon transforms appear in Computed Tomography. For instance, the measured data corresponds to the Radon transform of a slice image of say a knee. There the task is to reconstruct the slice to be able to do diagnosis for the knee say. On the other hand the forward transform, represents an image in terms of averages w.r.t. to lines, the linear "Hough buffer."

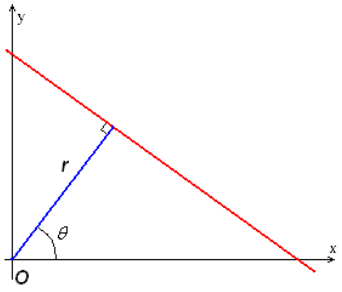


Figure 41: Parametrization of the lines in the Radon transform.

For defining the Radon transform we first parametrize the set of lines in the plane via their distance r to the origin, and the angle θ of their normal vectors; see Figure 41.

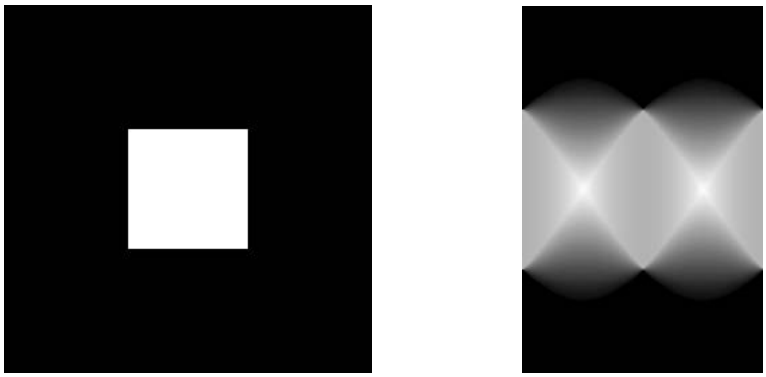


Figure 42: Radon transfor of a square. Origin in the image center (in the image). Angles beween 0 and π hortizontally, zero-offset vertically in the middle in the sinogram. (Source: Sandau)

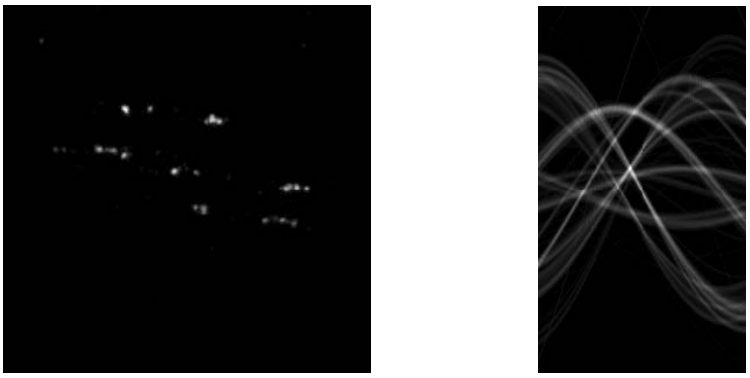


Figure 43: Damaged tissue (left) and corresponding sinogram (right.) (Source: Sandau)

The Radon transform $Rf(r,\theta)$ at position r,θ corresponds to the line parameterized by r,θ and computes the non-normalized average a.k.a. the integral of the signal f along this line:

$$Rf(r,\theta) = \int_{-\infty}^{\infty} f(r\cos\theta + t\sin\theta, r\sin\theta - t\cos\theta)\,\mathrm{d}t. \tag{94}$$

Thus the transform is well-defined whenever the integrals exist (which is for instance guaranteed for bounded functions with a bounded support.) The domain of the transformed function Rf is $[0,\pi) \times \mathbb{R}$ allowing the angle θ of the normal vector vary between $[0,\pi)$ and the offset r being a real number.
Examples can be found in Figure 42, in Figure 43 and in Figure 44. (Note that the images are scaled: the offset units in the Radon transform do not match the pixel units in the images.)

Fourier slice theorem. The Radon transform at (r,θ) encodes the line integral w.r.t. the line with normal vector angle θ and offset r. So keeping the angle θ

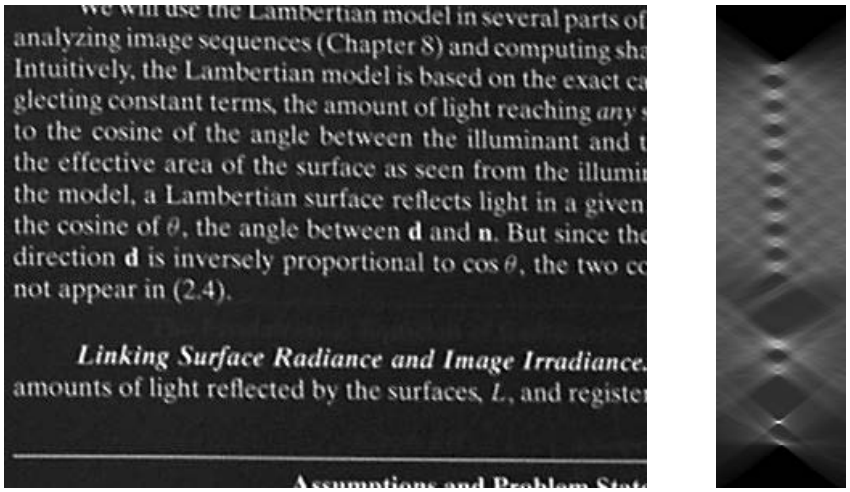


Figure 44: Rotated text. Method for finding the rotation angle. (Source: Sandau)

fixed (which corresponds to fixing a column in the sinogram), the function $r \mapsto Rf(r,\theta)$ encodes the line integrals w.r.t. fixed angle ϕ. On the other hand, consider the Fourier kernel $\exp 2\pi i\langle x, n\rangle$ where we integrate a bivariate signal f w.r.t. x. The Fourier integral

$$\int \exp 2\pi i\langle x, \|n\| \frac{n}{\|n\|}\rangle f(x)dx \tag{95}$$

first integrates the signal f along the lines perpendicular to n and then performs 1D Fourier transform along the direction evaluated at $\|n\|$. Hence,

$$\mathcal{F}[Rf(\cdot,\theta)](\xi) = \hat{f}(\xi\cos\theta, \xi\sin\theta). \tag{96}$$

This result of geometric considerations is called Fourier slice theorem. In particular, we get the Fourier tranform of f by 1D Fourier transforms of the Radon transform for fixed angles. Hence the Randon transform is invertible since the Fourier transform is.
However, in contrast to the Fourier transform, the Radon transform is ill-posed. This means that small perturbations in the sinogram result in potentially huge perturbations of the reconstructed images. To avoid such effects the reconstruction is regularized as detailed below.

The adjoint Radon transform. The adjoint of the Radon transform is given by

$$R^*g(x,y) = \int_0^\pi g(x\cos\theta + y\sin\theta, \theta)\,\mathrm{d}\theta. \tag{97}$$

It takes a sinogram g and yields an image R^*g. If we apply R^* to Radon transformed data $g = Rf$, the result $R^*Rf(x,y)$ takes all lines passing through x, y and averages the line integrals.

So R^* is not the inverse transform. However, (intuitively, R^*Rf is an integrated version of f. This can be made formal. One can show that R^*R equals the inverse of the square root of the negative Laplacian up to constants, hence $(-\Delta)^{1/2}R^*R$ equals the original image up to scaling. Or) in terms of the Fourier transform,

$$\widehat{R^*Rf}(\xi) = \frac{1}{\|\xi\|}\widehat{f}(\xi). \tag{98}$$

Hence, the singular values of the Radon transform become small, and –when reconstructing– large frequencies are significantly amplified.
The differentiation can be transported to the sinogram. ($\Delta^{1/2}R^*$ equals differentiation plus Hilbert transform [=ramp filter] w.r.t. to the offset variables followed by applying R^*.) We consider a sinogram $g(r,\theta)$ and apply convolution only w.r.t. the offset variable r writing $g(\cdot,\theta) * h$, where $h : \mathbb{R} \to \mathbb{R}$ is the function with $\hat{h}(\xi) = |\xi|$. Then

$$f = R^*(Rf * h). \tag{99}$$

Hence, reconstruction amounts to univariate convolution with h w.r.t. the offset variable followed by applying the adjoint Radon transform. This process is called filtered backprojection. Due to ill-posedness, $\hat{h}$ is often multiplied by a damping function to damp higher frequencies. Examples are the Shepp-Logan (multiplication with sinc) or the Hamming filters (multiplication with $0.5 + 0.5\cos(2\pi\cdot)$.)

5.3 Discrete Radon Transforms

We here deal with means to implement the forward transform. We first discuss straight forward approaches taking $O(N^3)$ where N is the maximal side length of the image. (Such approaches are actually implemented in Python or Matlab.) Then we discuss an $O(N^2 \log N)$ algorithm going back to Götz/Druckmüller and Brady.

Direct approaches. The first approach is to rotate the image by the the respective angle θ and then to sum along the coordinate axes (as done in Scikit Image.)
A related approach is to subdivide the image pixels and to project to a line in a semi-discrete way (as done in Matlab.) The value of the micropixel is proportinally binned to the pixels on the line.
Another related approach is, given an angle θ, to figure out discrete lines by taking those pixels a continuous line hits and to sum these (potentially in a weighted fashion.) A reference is [24].

Götz-Druckmüller-Brady Discrete Radon Transform. This approach was independently proposed by Götz and Druckmüller [11] and Brady [4]. A summary and further information on respective filtered backprojection algorithms can be found in [20].
We consider square images of size $2^k \times 2^k$. Smaller images can be brought to these dimensions by filling with zeros. We first consider lines with slope be-

tween 0 and 1. This corresponds to angles between 0 and $\pi/2$ for the lines resp. between $\pi/2$ and $3/4\pi$ for the normal vectors. Later we describe how to address the other slopes/angles by transforming the image. The algorithm is based on particular sets of discrete lines. They have the feature that they can be generated recursively. We produce discrete lines $D_N(h,s)$ consisting of N points, having intercept h and having rise s, with $0 \leq s \leq N-1$, see Figure 45. (We want to compute the sums w.r.t. these lines. The particular feature is that sums can be just computed from the sums of shorter lines.) The lines are inductively defined by

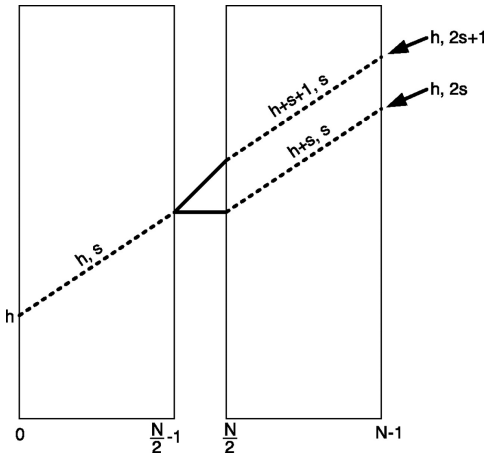


Figure 45: Generation of discrete lines (Source: Press.)

$$D_N(h,2s) = D_{N/2}(h,s) \cup D_{N/2}(h+s,s) \tag{100}$$

$$D_N(h,2s+1) = D_{N/2}(h,s) \cup D_{N/2}(h+s+1,s), \tag{101}$$

where the right hand indices refer to images of half width.
For a visualization of some discrete lines see Figure 46. Note that in a square image of size $2^k \times 2^k$, we have 2^k horizontal lines and 2^k discrete diagonal lines which are (potentially) non-zero. The discrete line $D_N(h,s)$ approximates a continuous line with intercept h (y-coordinate the left image boundary) and slope $\frac{s}{N-1}$.
The discrete Radon transform with respect to the discrete lines parametrized by h,s is given by the sum of all pixels belonging to that line

$$R^a(h,s) = \sum_{(i,j)\in D_N(h,s)} f_{ji}, \tag{102}$$

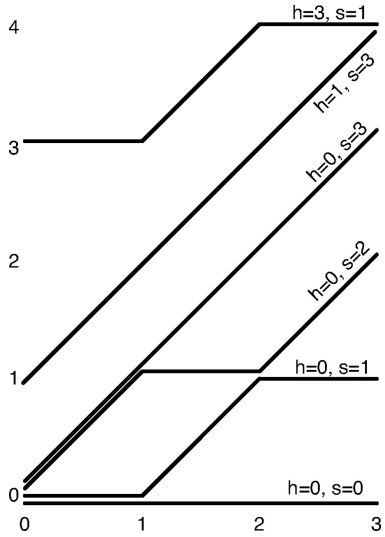


Figure 46: Example of discrete lines for $k = 2$ (Source: Press).

where the image is set 0 outside its bounds. This definition together with the definition of the discrete lines yields a direct recurive scheme for computing R^a. Starting

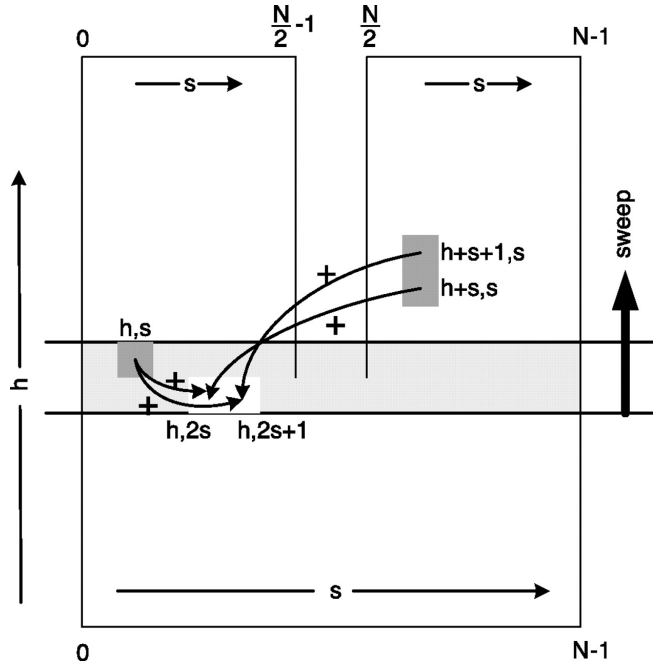


Figure 47: Schematic for an algorithm realizing the recursion to compute the discrete Radon transform. (Source: Press)

with lines of length 1, we may start with the image (extended by additional $2^k - 1$ additional rows initialized by 0 to take care of the fact that there are more diagonal nonzero lines.) To deal with lines of length 2 we add neigboring even and uneven columns (the uneven columns non-shifted and shifted) and write the result to these columns (the result for the nonshifted part goes to the even-indexed column.) As a result, the even indexed columns correspond to $s = 0$, the odd-indexed columns to $s = 1$ and the corresponding h encodes the intercept. For lines of length 4 we proceed analogous using (100) to obtain 4-column-pairs for $s = 0$, $s = 1$, $s = 2$, and $s = 3$. For a visualization, see Figure 47. More precisely, we get the first column corresponding to $h, 2 \cdot 0$ by summing the first $[(h, 0)]$ and the third $[(h + 0, 0)]$. We get the second column corresponding to $h, 2 \cdot 0 + 1$ by summing the first $[(h, 0)]$ and the third $[(h + 0 + 1, 0)]$, shifted by one. We get the third column corresponding to $h, 2 \cdot 1 + 0$ by summing the second $[(h, 1)]$ and the fourth $[(h + 1, 1)]$, shifted by one. Finally, we get the fourth column corresponding to $h, 2 \cdot 1 + 1$ by summing the second $[(h, 1)]$ and the fourth $[(h + 1 + 1, 1)]$, shifted by two. We proceed similarly for larger k; cf. again Figure 47.

Note that only additions are employed. After k sweeps, we are done. Each line in the image contains the sums w.r.t. the different slopes, and the line index encodes the intercept (w.r.t. the left boundary.) The result is on $O(L^2 \log L)$ algorithm where $L = 2^k$ denotes the image side length. Python code is displayed in Listing 2.

Covering the whole angle range. The definition of R^a covers an angular range between 0 and $\pi/4$ of the slope of the line (between $\pi/2$ and $3\pi/4$ of the slope

```python
def radon_gdb(img):

    ys, m = img.shape
    n = np.log2(m).astype(int)

    P0 = np.zeros((2 * m - 1, m), dtype='int64')
    P1 = P0.copy()

    P0[0:m, :] = img
    TwoPowKp1 = 1

    for k in np.arange(0, n, dtype='int'):
        TwoPowK = TwoPowKp1
        TwoPowKp1 = TwoPowK * 2

        for s in np.arange(0, TwoPowKp1):
            s_half = s // 2
            P1[:, s::TwoPowKp1] = P0[:, s_half::TwoPowKp1]
            + np.roll(P0[:, s_half + TwoPowK::TwoPowKp1],
            ((s + 1) // 2), axis=0)

            # Setting params w.r.t. image center (if desired)
            #if TwoPowKp1 == m:
            # P1[:, s] = np.roll(P1[:, s], (m - s - 1) // 2)

        P0 = np.copy(P1)
    return P1
```

Listing 2: Python-Code for the Götz-Druckmüller-Brady Discrete Radon/Hough transform.

of the line).
Following [20] and the original works [11, 4], we define the discrete Radon transform for the complete angular range. First, between $\pi/4$ and $\pi/2$, we let

$$R^b f = R^a M f, \tag{103}$$

where $(Mf)_{ji} = f_{ij}$ denotes the reflection at the line bisecting the first sector. We stich the results by flipping the present result w.r.t. both axis; see Figure 48. Then, between $-\pi/4$ and 0, we reverse the columns and apply R^a, i.e.,

$$R^d f = R^a F f, \tag{104}$$

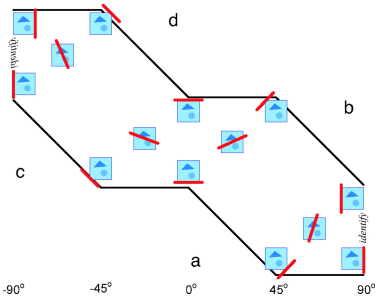


Figure 48: Stitching R^a to obtain a full angle DRT. (Source:Press)

where $(Ff)_{ji} = f_{2^k-1-j,i}$. We again flip the result twice and stitch according to Figure 48. Finally, we interchange coordinates and flip columns (which results in flipping lines), i.e.,

$$R^d f = R^a X f, \tag{105}$$

where $(Xf)_{ji} = f_{i,2^k-1-j}$. This representation allows for "continuously passing" over the patches while being able to apply R^a and not using multiplications. Further, the lines passing through the corners of the image are represented as straight lines. (**Question: Which lines are these?**) .

Deriving the classical sinogram. Another idea is to keep the image center fixed as a line as in the sinogram. Then one may stich the image similarly as discribed obtaining a DRT with shifts within the columns. Besides these the most important issues in connection with the sinogram are (i) angles are not sampled equidistantly but according to $\tan(\theta)$, (ii) values are just added, the "line element" is missing. These issues can be corrected. We consider R^a and the corresponding range between 0 and $\pi/4$. The other patches are treated analogously. First, let

$$\theta_s = \arctan \frac{s}{2^k - 1}. \tag{106}$$

Then $R^a(h,s)$ corresponds to the column θ_s in the sinogram. Further, the result of $R^a(h,s)$ has to be scaled by $\frac{1}{\cos\theta_s}$ to account for the fact that a line with slope $\frac{s}{2^k-1}$ has length $\sqrt{1+\left(\frac{1}{\theta_s}\right)^2}$ per unit. The off-set of a line indexed by h, s w.r.t. the center of the image equals $\cos(\theta_s)\left(h - \frac{2^k-1-s}{2}\right)$; see the comment part of Listing 2.

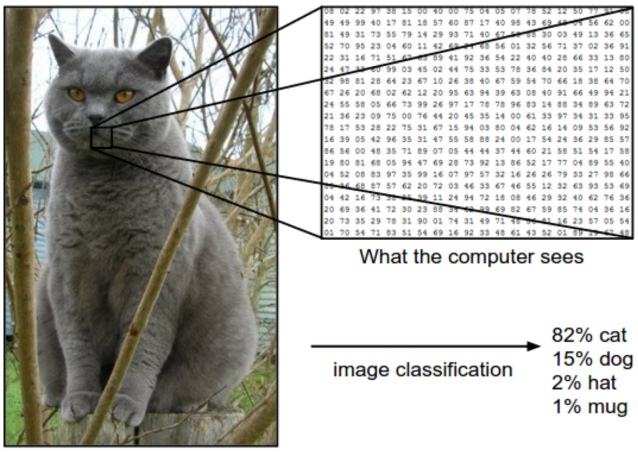


Figure 49: An image classification model takes a single image and assigns probabilities to 4 labels, here $label \in \{cat, dog, hat, mug\}$. (Source: Fei-Fei Li et al.)

6 Machine Learning Algorithms

In this section, we start out to recall the image classification problem. Then, we will discuss different learning-based classification algorithms.

Recap: The Image Classification Problem. Recall that the image classification problem is the task of assigning a label from a fixed set of classes to an input image. This is one of the core problems of Computer Vision. Many other seemingly distinct computer vision tasks such as object detection, segmentation, or semantic segmentation can be reduced or may be seen as generalizations of image classification.
Frequently (as shown in Figure 49), instead of assigning a single label, probabilities (or quantities summing up to 1 which are then interpreted as probabilities) of the image belonging to a certain class are also termed classification. E.g., taking the highest probability as class label is a means of obtaining a classification in the strict sense if required. Conversely, given an assignment of one label to an image (e.g., annotated training data) a probabilistic representation is obtained by letting this class equal 1 and the other classes equal 0.
Major Challenges are as follows:

- Invariance w.r.t. viewpoint variations: a single instance of an object can have different orientation w.r.t. the camera which should not change the classification result.
- Invariance w.r.t. scale variation: classes often exhibit variation in their size in the real world and in the image. This should neither alter classification results.
- Invariance w.r.t. deformation. Many objects are not rigid bodies and may appear deformed in quite extreme ways.
- Ability to deal with occlusion: sometimes only a part of an object is visible.

- Invariance w.r.t. illumination conditions.
- Ability to deal with background clutter. The objects of interest may blend into their environment, making them hard to identify.
- Intra-class variation. The classes of interest can be relatively large such as car, chair or cat. There are many different types of these objects each of which have their own appearance.

A good image classification model addresses all these points. (It is clearly understood that not all these issues can be dealt with in a 100% manner but are a statement of desirable properties which should be addressed in a best possible way.)
We formalize the image classification pipeline. We consider classification for K different classes with the classes a priori fixed.

- The **input** consists of a set of N images, each labeled as one of the K classes. This data is called training set.
- **Learning:** the task is to use the training set to learn the classes, more precisely, to learn a function which assigns an image to a class. This step is refered to as training a classifier, or as learning a model.
- **Evaluation:** Finally, we evaluate the quality of the classifier by letting it predict labels for a new set of images, i.e, images which have not been used for training yet.

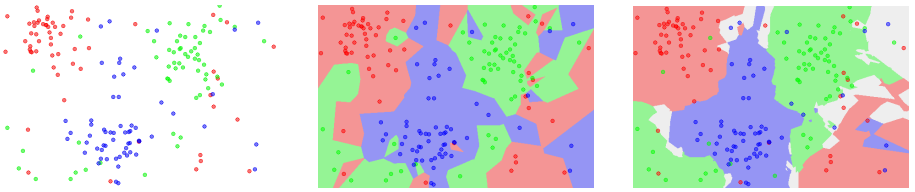
  We then compare the true labels of these images to the ones predicted by the classifier. The *accuracy* of the classifier is the rate of well-predicted classes, and thus a number between 0 and 1. The higher the value, the " better" the classification result.

Putting this into context, we consider classification as a supervised learning task. Here, the term "supervised" refers to the fact that the training data is annotated, i.e., the class labels are presented. (Unsupervised classification in this setup [without labeled training data] makes no sense, but note that a related unsupervised task consists of clustering.)
We now discuss different approaches for classification.

6.1 Nearest Neighbor Classifiers

We start out to discuss nearest neighbor classifiers, and develop an idea about the basic approach to an image classification problem.
For the time being and to stress the basic idea, we apply nearest neighbor classification to the image space itself. (We point out that significantly better results may be obtained by determining a suitable feature space beforehand; cf. also the problem session.)
In this setting, the nearest neighbor classifier will take a test image, compare it to every single one of the training images, and predict the label of the closest training image. In order to quantize the term "closest" one typically introduces a distance, for instance the L^1 distance between images given by

$$\mathrm{d}_1(I, J) = \sum_{ij} |I_{ij} - J_{ij}| \tag{107}$$



2d data of three classes | nearest neigbor | 5-nearest neighbor

Figure 50: k-nearest neighbor classification of 2d data (two real-valued features) with three classes for $k = 1, 5$. (Source: Wikipedia.)

where I, J denote images with coordinates I_{ij}, J_{ij}. Other choices are the p-distances

$$d_p(I, J) = \left(\sum_{ij} |I_{ij} - J_{ij}|^p \right)^{\frac{1}{p}} \tag{108}$$

with $p \in [1, \infty)$. (For $0 < p < 1$ remove the $\frac{1}{p}$ part.) Note that if I, J are color images, $|I_{ij} - J_{ij}|$ denotes a distance in three-dimensional space, e.g., the euclidean one given by taking the square-root of the squared sum of the absolute values of the channel differences. For $p = 2$, we obtain the L^2 distance. L^1 and L^2 distances are the most commonly used instance of L^p-distances. The L^2 distance penalizes a large difference in single pixels significantly more severely than the L^1 distance. Hence, nearest neighbors in the L^2 setting tend to have many medium disagreements rather than a few big ones.
We again point out that it is in general a bad idea to apply nearest neighbor classification to the image space itself (**Convince yourself.**) Better results may be obtained by determining a suitable feature space beforehand.

k - Nearest Neighbor Classifier. It is not very robust to only use the label of the nearest image to make a prediction. Instead incorporating some neighborhood seems more promising. The k-Nearest Neighbor Classifier realizes this as follows: instead of finding the single closest image in the training set, it finds the k closest images; among these k closest test images, a vote on their labels takes place, e.g., the label appearing most is chosen. The special case $k = 1$ corresponds to the nearest neighbor classifier. Higher values of k make the classifier more resistant to outliers but also make boundaries between classes less distinct, or in other words, may smooth to much. The choice of a suitable k is discussed below.
Concerning the learning step of the k-nearest neighbor classifier, we note that basically the labeled input data has to be stored. So learning is cheap. In the evaluation step, however, we have to compute the distance between the test sample and each stored labeled training sample.
From a statistical point of view, k-nearest neighbor classification enjoys strong consistency whereas plain nearest neighbor classification does not.

Hyperparameter Tuning and Validation Sets.

Example 6.1 (Image classification dataset CIFAR-10.). A popular toy image classification dataset is the CIFAR-10 dataset. This dataset consists of 60000 tiny

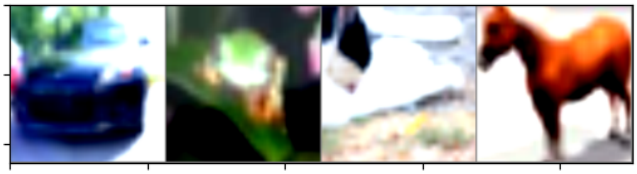


Figure 51: Four sample images from the CIFAR-10 data set; see https://www.cs.toronto.edu/~kriz/cifar.html.

RGB images of size 32×32; see Figure 51. There are ten classes ("airplane, automobile, bird, etc") one of which each image is labeled with. The $60,000$ images are partitioned into a training set of $50,000$ images and a test set of 10,000 images; see https://www.cs.toronto.edu/~kriz/cifar.html.

The positive integer k in the k-nearest neighbor classifier is a hyperparameter, i.e., a free parameter of the model which has to be specified before one can apply an algorithm to test data. (Another potential hyperparameter is the parameter p of the L^p distance, potentially quantized to $p \in \{1,2\}$.) We explain data-driven means to find suitable hyperparameters. Basically, one tries out different values of the hyperparameters and takes that values which work best, i.e., which produce the highest accuracy. It is important to find a proper set on which this hyperparameter tuning takes place. We cannot use the test set for this purpose since it would bias the result. Otherwise hyperparameters may work well on the test set (we incorporated in the parameter tuning process,) but if we apply our model to further data we significantly lose performance. (The test set should not be abused for training: ideally, it should only be touched in the very end after all hyperparameters/parameters have been determined for evaluation.)
A proper way of tuning the hyperparameters is as follows: we split the training set in two parts: a slightly smaller training set, and a validation set. This validation set is now used to tune the hyperparameters. Let us consider a more concrete example: Consider the CIFAR-10 data set and the k of the k-nearest neighbor classifier as hyperparameter. Consider, say 40000 of the training images for training, and 10000 for validation. For each sample in the validation set, we apply k-nearest neighbor classification w.r.t. the 40000 training images for $k = 1, \ldots, 10000$. Then, for each k, we may check the number of well-labeled validation images (This is a number between 0 and 10000.) We may choose the hyperparameter k^* which has the highest score, i.e., the highest number of well-labeled validation images. Then we apply the algorithm with this hyperparameter k^* to the test images.
As a general principle for tuning hyperparameters we record the following approach: split the training data into an actual training set and a validation set. Use the validation set to find suitable hyperparameters. Only use the test set for evaluating performance: run the algorithm with the found hyperparameters on the test set and report performance.
Recall that the above evaluation criterion is called *accuracy*, the fraction of correct predictions.

Cross Validation. If there are only few training data (and therefore also few validation data), a more sophisticated technique for tuning hyperparameters is cross-validation.
Instead of dividing the training data into actual training data and validation data once, we consider the following process generating several instances. First, we divide the training data into l folds, e.g., $l = 5$. Then we consider $l - 1$ folds as actual training data and the remaining fold as validation set. More precisely, in the first step, folds $1, \ldots, l - 1$ are considered as actual training data and fold l as validation set; next, folds $1, \ldots, l - 2$ and fold l are considered as actual training data and fold $l - 1$ as validation set; Doing so for all l we end up with l instances of the previously considered setup. For each instance we can proceed as described above, average the score w.r.t. the different instances and choose optimal hyperparameters accordingly.
We note that cross validation can be computationally expensive for larger training sets. However in this case the approach of dividing into a single test and validation set may be sufficient, and cross validation may not be necessary. Typically no computational cost problems arise for a few hundred training data samples and then cross-validation is reasonable.
As a rule of thumb, between 50% and 90% of the training data are typically used for training and the rest is used for validation. If there are many hyperparameters, one should use a higher percentage of the data as validation data. If one uses cross-validation, one may typically divide the data into three, five or ten folds. (Typically, the more folds there are the better the tuning works, but it is also computationally more expensive).

Summary on nearest neighbor classifiers. It is simple to implement and understand. The classifier takes no time to train, since all that is required is to store and possibly index the training data. However, this comes with computational cost at test time: classifying a test example requires a comparison to every single training example. (Notice that test time efficiency is in many cases significantly more important than the efficiency at training time.)
We point out that the computational complexity of nearest neighbor classifiers is still an active area of research. There are approximate nearest neighbor (ANN) algorithms and libraries such as FLANN exist that accelerate finding nearest neighbors. ANN algorithms allow one to trade off the correctness of the nearest neighbor retrieval with its space/time complexity during retrieval.
The nearest neighbor classifier may sometimes be a good choice in some settings (in particular, if the data is low-dimensional), but it is not appropriate for use on images. When using it in the context of computer vision, it is typically used on a (low-dimensional) feature space. Finding a suitable feature space is then key.
Before application, it is recommended to normalize the features to have zero mean and unit variance. If the data is higher-dimensional (already at dimensions of more than 10) one might consider using a dimensionality reduction technique such as PCA to avoid the effects of the curse of dimensionality. (The curse of dimensionality in the kNN context basically means that at least the euclidean distance is somewhat problematic in high dimensions because almost all training vectors are almost equidistant to the candidate vector.) Train and evaluate the kNN classifier for various choices of k and across different dis-

tance types, e.g. L^1 and L^2. If necessary, a potential speed-up is obtained using an ANN library (at the cost of some accuracy.)

6.2 Linear Classification

As we saw, kNN has some disadvantages: it has to store the training data for future comparison with the test data. This is rather space inefficient (in particular for large datasets) and classifying a test image is rather expensive.
In the following we present approaches where the model has parameters (not to confuse with the hyperparameters) and where, in the training stage, the model parameters are learned from the training data. Equipped with the found model parameters the training data is no longer necessary for running the classifier.
In order to determine the model parameters from the training data in the training phase, we need the notions of a *score function* and a *loss function*. Basically, the score function assigns each data item/image a class score (i.e., a number for each class, you may think of these scores as a pre-stage of classification probabilities; cf. Figure 49). The loss function assigns each class score a real valued number so that overall each data item/image is a assigned a real valued loss by the concatenation of score and loss function. This number depends on the model parameters, i.e., it is a function of the model parameters. Suming up over all data items results in a mapping assigning the parameters of the model a real-valued overall loss. The model parameters are determined by minimizing this mapping.
In the following we discuss these issues in detail; in this subsection we start out with simple parametric models.

Linear classifiers. A very simple parametric model is given by the linear mapping

$$f(x_i, W, b) = Wx_i + b. \tag{109}$$

Here, x_i are the data and W, b are model parameters which have to be determined by training. If the x_i are images we assume that they are flattened out to a single $D \times 1$ column vector. The matrix W has size $K \times D$, and the vector b has size $K \times 1$, where K denotes the number of different classes. The output of f are the scores as a function of W and b.
For example, in the CIFAR-10 data set, x_i corresponds to the i-th image of size $32 \times 32 \times 3$ flattened into a single column of size 3072×1. W is of size 10×3072 since the data set has 10 different classes labeled, b is of size 10×1. The parameters in W are often called weights, and b is called the bias vector. The terms weights and parameters are frequently used interchangeably.
Each row of W together with the corresponding entry of b defines a hyperplane in $\mathbb{R}^D$, and thus a linear (binary) classifier for the corresponding class. This way, we face K separate classifiers –one for each class– in parallel (in the example, one for each of the $K = 10$ classes). This leads to the following interpretation as template matching. Each row of W corresponds to a template (also called a prototype) for one of the classes. The score of each class for an image is then obtained by comparing each template with the image using an inner product. From this point of view, a linear classifier is doing template

matching, where the templates are learned. The training is then interpreted as finding suitable prototypes.
Another way to think of it is that we are still effectively doing nearest neighbor classification, but instead of having training images we only use a single image per class (which we learn, and which does not have to be an image from the training set), and we use the (negative) inner product as a kind of distance instead of L^1 or L^2 distances.
We note that the classification of a test image basically has the cost of a matrix multiplication (where one dimension corresponds to the number of labels) which can be done rather fast.
It is very important to notice that the input data (x_i, y_i) is given and fixed, and in concrete applications these symbols denote numerical values. The parameters W, b are the actual objects of interest we want to determine and we consider f as a function of W and b.

Bias trick. Instead of always writing down the two parameters W, b there is a means to write them as one. Basically, going from affine to projective (synonymously, homogeneous) coordinates, allows to rewrite the score

$$f(x_i, W, b) = Wx_i + b \tag{110}$$

by

$$f(x_i, \tilde{W}) = \tilde{W}\tilde{x}_i \qquad \text{with} \quad \tilde{W} = \begin{pmatrix} W & b \end{pmatrix} \quad \text{and} \quad \tilde{x}_i = \begin{pmatrix} x_i \\ 1 \end{pmatrix}. \tag{111}$$

In the following we will just omit the tilde and understand that if we face expressions of the form $f(x_i, W) = Wx_i$ that the bias is incorporated into W and x_i.
In case of the CIFAR-10 example, x_i is now of size 3073×1 instead of 3072×1. (with the extra dimension being constant equal to 1), and W is now of size 10×3073 instead of 10×3072. The extra column in W contains the bias b.

Image data preprocessing. It is very common to always perform normalization of the input features (e.g., in the case of images, every pixel might be interpreted as a feature).
First, the data is centered by subtracting the mean from every feature. In the case of image pixels as features, this means computing a mean image across the training images (pixelwise means) and subtracting this mean image from every training image. (Note that this transforms the pixels range to have negative entries.)
Secondly, one may scale each input feature so that its values range from $[-1, 1]$. Zero mean centering is more important and, in case of image pixels as features, all features essentially live on the same scale which makes scaling less important.

Loss functions. Also other more elaborate models such as neural networks map images to scores using a function f as above (whose implementation is however different). The principle is similar: the model yields a score function that maps the raw data to class scores. The second ingredient is a loss function

that quantifies the agreement between the predicted scores and the ground truth labels. The concatenation of score and loss function yields an optimization problem in which we will minimize the loss function with respect to the parameters of the score function.
We here consider loss functions together with the linear model (109) first and point out that they may be applied to more complex models with minor changes (if necessary at all.)

Multiclass support vector machine loss. We start out with the multiclass support vector machine (SVM) loss. The loss function quantifies our unhappiness with predictions on the training set. We want the correct class for each image to have a score higher than the incorrect classes. In particular, the SVM loss incorporates a fixed margin Δ.
More precise, given the ith test image x_i and its class label y_i that specifies the index of the correct class, the score function computes the vector $f(x_i, W)$ of class scores s^i, where

$$s^i_j = f(x_i, W)_j = (Wx_i)_j, \tag{112}$$

and the index j denotes the jth component of score of the ith data sample corresponding to class j. The Multiclass SVM loss for the ith test sample is then defined by

$$L_i(W) = \sum_{j \neq y_i} \max\left(0, s^i_j - s^i_{y_i} + \Delta\right) \tag{113}$$

where the sum is taken w.r.t. the incorrect classes. (Recall that y_i was the annotated correct class.) Note that (113) states that the jth item does not contribute to the loss if the score component $s^i_{y_i}$ w.r.t. the correct class is by Δ larger than the other score components; otherwise, penalization is linear. (**Construct your own small toy example and calculate it by hand.**) The loss above is frequently called hinge loss, the corresponding squared hinge loss SVM, or L^2-SVM, is obtained by replacing $\max(0, \cdot)$ by $\max(0, \cdot)^2$. The non squared version is more standard, but in some cases the squared hinge loss can work better. This can be determined by hyperparameter tuning, e.g., by cross-validation.
The objective of optimization is now to find model parameters which minimize the loss. From this perspective, it would be optimal if the loss were zero, i.e., the score for the correct label is for all test images by Δ higher than the score or the incorrect classes. If there are less parameters in the model than test samples this is typically not possible, and the optimization tries to reach this goal "as best as possible," with this phrase quantified by the loss function.

Regularization. In the above form, the loss and, and in turn the model parameters W, depend on the training data only. If we also want to incorporate a priori assumption, a priori information and/or requirements this is typically done by regularization. Regularization can for instance be performed by adding a regularization term to the loss function. If the a priori information or assumption is that the parameters W_{ij} are typically small, a corresponding loss

function incorporating this assumption would be the full Multiclass SVM loss

$$L(W) = \frac{1}{N}\sum_i L_i(W) + \lambda R(W), \qquad \text{with } R(W) = \sum_{k,l} W_{kl}^2. \tag{114}$$

Here, N denotes the number of training samples, and λ is a hyperparameter regulating the tradeoff between regularization and data part. There is no simple way of setting λ and it is usually determined by cross-validation. Choosing the squared L^2 norm for the regularization part is somewhat arbitrary as also the absolute sum would be a reasonable choice; however, the squared L^2 norm has the nice property that it is differentiable.
Typically, biases are not regularized such that the sum above is taken w.r.t. the weights in W, but not the biases b which are also stored in W.
We note that penalizing large weights tends to improve generalization since so no particular input dimension can itself strongly influence the scores on its own.
Concerning the choice of Δ, we note that it can be set to $\Delta = 1.0$ in all cases. The reason for this is that the hyperparameters Δ and λ both control the same trade-off between the data loss and the regularization loss.
For an interactive web demo we refer to http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/.

Softmax classifier. Besides the SVM classifiers, the other popular choice is the softmax classifier. The only difference is a different loss function. Using the function $f(x_i, W) = Wx_i$ as above we obtain the same definition of scores as for the multiclass SVM classifier. We now interpret these scores as the unnormalized log probabilities of each class. We replace the hinge loss (113) with the cross-entropy loss

$$L_i(W) = -\log\left(\frac{\exp(s^i_{y_i})}{\sum_j \exp(s^i_j)}\right) = -s^i_{y_i} + \log\sum_j \exp(s^i_j). \tag{115}$$

Symbols mean the same as for the SVM classifier above. The full loss for the dataset is given by (114) with the L_i implemented by (115).
The function $g : \mathbb{R}^D \to \mathbb{R}^D$ which maps $z = (z_1, \ldots, z_D)$ to $g(z)$ with lth component $g_l(z) = \frac{\exp(z_l)}{\sum_j \exp(z_j)}$ is called the softmax function. Each component of the resulting vector $g(z)$ has a values between zero and one, and the components sum up to 1.
The cross-entropy between a "true" distribution p and an "estimated" distribution q is given by:

$$H(p, q) = -\sum_i p_i \log q_i \tag{116}$$

The Softmax classifier is minimizing the cross-entropy between the estimated class probabilities $\frac{\exp(s^i_l)}{\sum_j \exp(s^i_j)}$ and the true distributions where all probability mass is on the correct class, i.e., it contains a single 1 at the y_ith position and 0 elsewhere. Moreover, this expression also minimizes the Kullback-Leibler divergence

$$\mathrm{KL}(p||q) = H(p) - H(p, q) \tag{117}$$

where $H(p)$ denotes the entropy of p, since the entropy of the true distributions where all probability mass is on the correct class equals zero.
Additionally, we have the following probabilistic interpretation. We may interpret

$$P(y_i|x_i, W) = \frac{\exp(s^i_{y_i})}{\sum_j \exp(s^i_j)} \tag{118}$$

as the (normalized) probability assigned to the correct label y_i given the image x_i, with parameters W. We hence minimize the negative log likelihood of the correct class which amounts to performing maximum likelihood estimation (MLE), i.e., choosing that parameter set W which maximizes the likelihood. Further, one my then interpret the regularization term $R(W)$ as coming from a Gaussian prior over the weight matrix W; in this interpretation, minimizing the full loss function amounts to maximum a posteriori (MAP) estimation. For further details and derivations we refer to the literature. Finally, we note that softmax classification w.r.t. two classes amounts to performing logistic regression.
Concerning computation some numerical issues arise. The intermediate term $\exp(s^i_{y_i})$ and the sum $\sum_j \exp(s^i_j)$ can become very large. To avoid this we note that

$$\frac{\exp(s^i_{y_i})}{\sum_j \exp(s^i_j)} = \frac{C\exp(s^i_{y_i})}{C\sum_j \exp(s^i_j)} = \frac{\exp(s^i_{y_i} + \log C)}{\sum_j \exp(s^i_j + \log C)} \tag{119}$$

where we can choose C arbitrarily. Choosing $\log C = -\max_j s^i_j$ scales into a reasonable range (the argument of each exponential is smaller than or equal to 1 then) and avoids problems.

6.3 Parameter Optimization

The concatenation of the score function and the loss function yields a real-valued function of the parameters which measures the classification quality on the training data. For example, in the case of a linear model $f(x_i, W) = Wx_i$ and the SVM classifier this overall loss function of the parameters W explicitly reads

$$L(W) = \frac{1}{N}\sum_i \sum_{j\neq y_i} \max\left(0, (Wx_i)_j - (Wx_i)_{y_j} + 1\right) + \lambda\left(\sum_{k,l} W^2_{kl}\right). \tag{120}$$

We want to find parameters W that minimize $L(W)$. We then use the found parameters as model parameters for classification. In other words, the minimization of the loss function performs the parameter estimation for the model from the test data.
Concerning the structure of L in (120) we first observe that we have to minimize a function of M variables, where M denotes product of the number of classes and the image size. (In case of the CIFAR-10 data set the number of variables is 30730). The function in (120) is convex (which however will not be the case for the more complex models considered later). Due to the use of the

max-function, it is not differentiable everywhere. However, it is differentiable in almost all points and the concept of a subgradient is available for the other points. Therefore, for practical issues, a gradient is available and we will not further elaborate on this issue. We continue to use the terms gradient (including such cases as well.)
Concerning optimization strategies, we note that grid search is not practical in high dimension. However, local strategies such as gradient descent are practicable. For convex problems local strategies typically yield global minimizers; for non-convex problems this is no longer true, one typically gets local minimizers instead. However, since the loss is an auxiliary tool, one might also be willing to accept a local minimum as a parameter estimate if it performs well.
Recall that gradient descent is an iterative scheme given by

$$W_n = W_{n-1} - \tau \, \nabla L(W_{n-1}), \tag{121}$$

starting from some initialization W_0. Here the positive τ is called the step size or learning rate. Eq. (121) describes how to obtain the nth iterate W_n from the $n-1$th iterate W_{n-1} by going into the direction of the gradient ∇L of the loss function evaluated at W_{n-1}.
Concerning the step size, we note that the negative gradient tells us the direction of steepest descent, but not how far to step along this direction. Choosing the step size/learning rate is one of the most important hyperparameter settings in training a neural network. If it is too small, one consecutively obtains smaller losses but is slow. If it is too large, there is the risk of "overshooting." We explore this issue in more detail later on.
There are two ways to compute the gradient: A slow, approximate, but easy way to computing the gradient is the numerical evaluation of the differences

$$\frac{\partial L}{\partial W^{ij}}(W) \approx \frac{L(W + hE^{ij}) - L(W)}{h} \tag{122}$$

where W^{ij} denotes the (i,j) component of W, the matrix E^{ij} contains 1 in the (i,j) component and 0 else. Here h is a small positive number such as $10{-}6$. Here the reason for being slow is the required evaluation of L at $W + hE^{ij}$ for all i, j. (In case of the CIFAR 10 data set, these are already 30730 function evaluations.)
A fast, exact but more (human) error-prone way that requires calculus is the computation of the analytic gradient. We consider the SVM loss for a single datapoint as an example. We have

$$L_i(W) = \sum_{j \neq y_i} \max\left(0, (Wx_i)_j - (Wx_i)_{y_i} + \Delta\right) = \sum_{j \neq y_i} \max\left(0, w_j^T x_i - w_{y_i}^T x_i + 1\right)$$

where w_j^T denotes the jth row of the matrix W. We are interested in differentiating the function with respect to the weights W. Taking the gradient with respect to w_{y_i}, we get

$$\nabla_{w_{y_i}} L_i(W) = -\left(\sum_{j \neq y_i} 1_{w_j^T x_i - w_{y_i}^T x_i + \Delta > 0}(W)\right) x_i \tag{123}$$

where 1_A denotes the indicator function of the set A, i.e., $1_A(z) = 1$ if $z \in A$ and 0 else. Implementing this in code you simply count the number of classes that did not meet the desired margin and then the data vector x_i scaled by this number is the gradient with respect to the row of W that corresponds to the correct class. For the other rows $j \neq y_i$ we have

$$\nabla_{w_j} L_i(W) = 1_{w_j^T x_i - w_{y_i}^T x_i + \Delta > 0}(W) x_i. \tag{124}$$

Gathering the rows, summing over the samples indexed by i and adding the gradient of the regularizing part, we obtain the gradient of the SVM loss function.

Batch gradient descent. The basic (or vanilla) variant of gradient descent is given by Algorithm 7.

Algorithm 7: Gradient Descent

Input: Guess/Initialization of Weights W

Output: Optimized Weights W

```
while true do
    weights_grad = evaluate_gradient(loss_fun, data, weights);
    W = W - step_size * weights_grad
end
return W
```

In large-scale applications the training data can have millions of examples. In order to save time by not computing the (full) loss function over the entire training set for only a single parameter update, a common approach is to compute the gradient over batches of the training data. (For example, in current state of the art convolutional neural networks, a typical batch contains 256 examples from the entire training set of 1.2 million.) One batch is then used to perform a parameter update; see Algorithm 8.

Algorithm 8: Batch Gradient Descent

Input: Guess/Initialization of Weights W

Output: Optimized Weights W

```
while true do
    data_batch = sample_training_data(data, batch_size=256)
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights);
    W = W - step_size * weights_grad
end
return W
```

In typical applications, the gradient derived from a mini-batch turns out to be a good approximation of the gradient of the full objective. Hence, a significant speedup is achieved by evaluating mini-batch gradients.

Stochastic gradient descent (SGD) here appears as the extreme special case of the mini-batch consisting of a single sample only. However, the term SGD and batch gradient descent are also often used as synonyms.
The size of the mini-batch is a hyperparameter but it is not very common to cross-validate it. It is usually based on memory constraints, or set to some value such as 64, 128, or 256.

Backpropagation. Backpropagation is a means of systematically computing analytic gradients using the chain rule of calculus. Hence, it is important to understand the multivariate chain rule. We first state the core problem studied: given some function f of the input vector x, compute the gradient $\nabla f(x)$ of f at x.
Considering univariate functions $f, g : \mathbb{R} \to \mathbb{R}$ we have the chain rule

$$\frac{d(g \circ f)}{dx}(x) = \frac{dg}{dy}(f(x)) \cdot \frac{df}{dx}(x). \tag{125}$$

We obtain the derivative of the concatenation of $g \circ f$ at position x by multiplying the derivative of g at position $f(x)$ with the derivative of f at position x. If we have three functions $f, g, h : \mathbb{R} \to \mathbb{R}$ we get by iterating the chain rule

$$\frac{d(h \circ g \circ f)}{dx}(x) = \frac{dh}{dz}(g(f(x))) \cdot \frac{dg}{dy}(f(x)) \cdot \frac{df}{dx}(x). \tag{126}$$

Hence to compute the derivative, we first compute the function values x, $f(x)$, $g(f(x))$ in a forward pass. We assume that we have analytic expressions or formulae for the derivatives of f, g, h. Based on them, we can then compute the derivative of $h \circ g \circ f$ at position x in a backward pass: First compute $\frac{dh}{dz}(g(f(x)))$ by inserting the value $g(f(x))$ computed in the forward pass into your analytic expression for the derivative of h and store the result; secondly, compute $\frac{dg}{dy}(f(x))$ using the same principle: inserting the value $f(x)$ computed in the forward pass into your analytic expression for the derivative of g, multiply it with the stored value and store the value; finally, multiply the lately stored value with the derivative of f at x. (**Perform this backpropagation algorithm manually for simple examples; e.g.,** $x \mapsto 3x + 4$.)
The chain rule is also valid in the multivariate setting. More precisely, for functions $f : \mathbb{R}^n \to \mathbb{R}^d$ and $g : \mathbb{R}^d \to \mathbb{R}^l$ we have

$$\frac{d(g \circ f)}{dx}(x) = \frac{dg}{dy}(f(x)) \cdot \frac{df}{dx}(x). \tag{127}$$

where $\frac{dg}{dy}$ denotes the Jacobi matrix of g, and $\frac{df}{dx}$ that of f. Recall that the Jacobi matrix contains the partial derivatives of the respective component functions with respect to all variables.

Example 6.2. The gradient of $f(x) = x_1 x_2$ is given by $\nabla f(x) = (x_2, x_1)^T$. E.g., the gradient at $(1, 2)$ equals $(2, 1)^T$. The Jacobi matrix at x is given by row vector (x_2, x_1).

Recall that, in general, the gradient is defined for scalar functions and equals the transpose of the Jacobian in this case.

Intuitively, the derivative on each variable x_k (at the corresponding value, e.g. $x = (0, 1, 0)$) tells us the sensitivity of the whole expression $f(x_1, \ldots, x_n)$ w.r.t. this variable x_k (at the corresponding value $f(x_1, \ldots, x_n)$.)

Example 6.3. The gradient of $f(x) = \max(x_1, x_2)$ is given by

$$\nabla f(x) = \begin{cases} (1,0)^T, & \text{if } x_1 > x_2, \\ (0,1)^T, & \text{if } x_1 < x_2. \end{cases}$$

If $x_1 = x_2$, the function is not differentiable and a suitable subgradient is given by $(0, 0)$.

Example 6.4. We consider $f(x, y, z) = (x + y)z^2$ and want to determine $\nabla f(1, 2, 3)$ We write $f = g \circ h$ with $h(x, y, z) = (x + y, z)$ and $g(u, v) = uv^2$. In the forward pass we get $h(1, 2, 3) = (3, 3)$. The analytic expression for the derivative/Jacobi matrix of h is given by

$$\frac{dh}{d(x,y,z)}(x, y, z) = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{128}$$

and that of g is given by

$$\frac{dg}{d(u,v)}(u, v) = \begin{pmatrix} v^2 & 2uv \end{pmatrix}. \tag{129}$$

Using the forward pass, we get that $\frac{dg}{d(u,v)}(h(1, 2, 3)) = \frac{dg}{d(u,v)}(3, 3)$ and plugging this into the analytic expression, $\frac{dg}{d(u,v)}(h(1, 2, 3)) = (9, 18)$. Further,

$$\frac{dh}{d(x,y,z)}(1, 2, 3) = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Multiplying these two matrices (i.e., performing back-propagation) we get that the transpose of the gradient of f at $(1, 2, 3)$ is given by $(9, 9, 18)$.

Example 6.5. We consider the simple neuron

$$f(w, x) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))} \tag{130}$$

and want to find the gradient of f with respect to the weights $w = (w_1, w_2, w_3)$. We may decompose $f = \sigma \circ D$ with the sigmoid function $\sigma : \mathbb{R} \to \mathbb{R}$ given by

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{131}$$

and $d : \mathbb{R}^3 \to \mathbb{R}$ with

$$D(w) = w_0 x_0 + w_1 x_1 + w_2. \tag{132}$$

The analytic expression for the derivative of D is given by

$$\frac{dD}{dw}(w) = (x_0, x_1, 1), \tag{133}$$

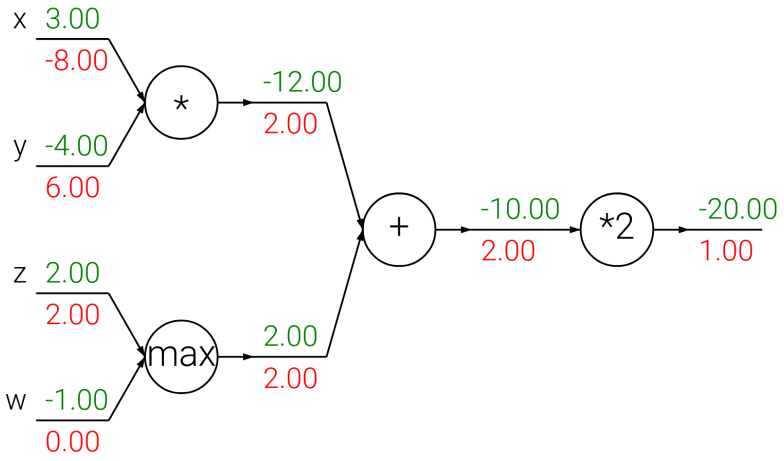


Figure 52: Visualization of backpropagation for basic operations. Green: forward pass. Red: back-propagation. (Source: Li Fei-Fei et al.)

and the analytic expression for the derivative of the sigmoid function σ is given by

$$\frac{d\sigma}{dz}(z) = \frac{\exp(-z)}{(1+\exp(-z))^2} = (1-\sigma(z))\ \sigma(z). \tag{134}$$

Computing the derivative of f w.r.t. the weights w at say $w = (0,1,2)$ for say $x = (3,4)$ now amounts to compute the forward pass $D((0,1,2)) = x_1 - 3 = 0$, followed by backpropagation: we have $\frac{d\sigma}{dz}(0) = (1-\sigma(0))\sigma(0) = (1-0.5)0.5 = 0.25$, and $\frac{dD}{dw}((0,1,2)) = (3,4,1)$ which results in the transposed gradient $(0.75,1,0.25)$ asked for.

In many cases the backward-flowing gradient can be interpreted on an intuitive level; see also Figure 52. Speaking pictorial, the add operation takes the gradient from its output and distributes it to all of its inputs, regardless of what their values were during the forward pass. (Mathematically, the derivative w.r.t. to each variable equals 1.) Further, pictorial, the max gate routes the gradient to the maximal input. Concerning multiplication, the derivatives are the input values switched. Hence, pictorial, while "passing the multiplication gate" the derivative is multiplied with the switched input. Notice that if one of the inputs to the multiply gate is very small and the other is very large, then the multiply gate assigns a large derivative value to the small input and a small derivative value to the large input. Hence, for linear classifiers, where we have expressions $w_j^T x_i$ for input x and class j where w_j^T is the jth row of the weight matrix W, the scale of the training data x_i has an effect on the magnitude of the gradient for the weights. (This explains why data preprocessing is an important issue.)

Finally, we derive derivatives for matrix-operations: Consider the mapping

$$W \mapsto W \cdot X,$$

where $\cdot$ denotes the matrix multiplication. Then the Jacobi-matrix/derivative of this mapping is given by X^T. This is because the mapping is linear and the derivative is just defined as the best linearization. Note that for the mapping $X \mapsto W \cdot X$, the derivative equals W. Further note that the case of vector-vector and matrix-vector multiplication are special cases.

6.4 Fully-connected neural networks

For linear classification we computed scores based on the formulation

$$s = Wx$$

with the weight matrix W (including the biases) and input x (In the case of CIFAR-10, x is a 3073×1 serialized RGB image including an additional entry with value 1 and W is an 10×3073 weight matrix corresponding to 10 classes, and 3073 parameters per class including offset. The output scores is a vector of 10 class scores.
A two layer example neural network would instead compute

$$s = W_2 \max(0, W_1 x) \tag{135}$$

Here, W_1 could be a say, 100×3073 matrix transforming the image into a 100-dimensional intermediate vector. (The 100 is arbitrary, and could be replaced by 50 or 200 say, the 3073 is fixed.) The $\max(0, \cdot)$ function is a non-linearity which we apply elementwise. (Another possible choice could be the sigmoid function introduced above.) The matrix W_2 is of size 10×100 then such that we get 10 numbers which we interpret as class scores.
(Notice that if we left out the non-linearity we would end up in linear classification with matrix $W = W_2 W_1$.) The parameters $W2, W1$ can be learned by stochastic gradient descent, and their gradients are derived with chain rule and computed with backpropagation.
A three layer fully connected neural network can be defined by

$$s = W_3 \max(0, W_2 \max(0, W_1 x)), \tag{136}$$

with parameter matrices W_3, W_2, W_1. The sizes of the intermediate hidden matrices are hyperparameters. Note that the process can be iterated to arbitrary depth beyond 3 by adding additional layers.
In the above examples the nonlinearity or activation function is the relu (rectified linear unit) function $x \mapsto \max(x, 0)$. Using the classical sigmoidal activation function from (130) in (135) we end up with the classical two-layer perceptron, using it in (136) we end up with the classical three-layer perceptron. Any order of multilayer perceptron can be obtained by imposing the respective number of layers.
Using a single neuron (see also Example 6.5) with the sigmoidal activation function from (130) and suitable loss function we obtain the binary softmax classifier, also known as logistic regression. More precisely, we may interpret $\sigma(w^T x_i + b)$ as the probability of the one class $P(y_i = 1 | x_i, w)$. Then the probability of the other class $P(y_i = 0 | x_i, w)$ equals $1 - P(y_i = 1 | x_i, w)$. Using this interpretation, we can formulate the cross-entropy loss and minimize it. By the minimization property, this leads to a binary softmax classifier.

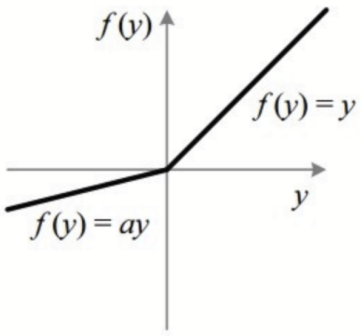


Figure 53: The leaky relu activation function. (Source: paperswithcode.com)

Activation functions. Activation functions, synonymous non-linearities, are scalar functions of a single real number. Commonly used activation functions are the sigmoid function and the relu-function already mentioned, where the sigmoid function has its historic merit and the relu-function is widely used nowadays. Further commonly used activation functions are the hyperbolic tangent function and the leaky relu function as well as the binary max function termed maxout. We discuss these function in a bit more detail now.
Recall that the sigmoid function was defined by $\sigma(x) = 1/(1 + \exp(-x)$. It is monotonously increasing and has range between 0 and 1. Large negative numbers are near 0 and large positive numbers near 1. It has the major drawback that sigmoids saturate and so yield small gradients. This refers to the observation that for large and small values, the gradients are very small, and by the chain rule these small gradients "enter" the overall gradient which in turn becomes small and which is undesired since no changes take place during the iteration any more.
The hyperbolic tangent function $\tanh(x) = \sinh(x)/\cosh(x) = 2\sigma(2x) - 1$ is a scaled and translated version of the sigmoid which faces the same saturation problem but is zero centered.
The rectified linear unit (relu) $f(x) = \max(0, x)$ is widely used nowadays. It was found to accelerate the convergence of stochastic gradient descent compared with sigmoid or hyperbolic tangent functions. It is simple to implement and avoids rather expensive operations such as the exponential. On the flipside, relu units can "die" during training which means that the gradient "flowing" trough that unit is always 0 starting from a certain time point in the training process. This is also a saturation effect (but only one-sided saturation). More precisely, it may happen during training that the weights of a neuron reach a state such that their output for any trainings data is negative and the relu function as well as its gradient constantly yield 0. Thus, it is "dead" since the gradient is 0 and the weights will not change any more. Such a state of weights can be caused by too large step-sizes which can be avoided by a proper setting of the learning rate.
One attempt to fix the dying relu problem is the leaky relu function given by $f_\alpha(x) = x$ if $x \geq 0$ and $f_\alpha(x) = \alpha x$ for small α, e.g., $\alpha = 0.01$ for $x < 0$. For a visualization, see Figure 53
Another approach is maxout. It computes the function $\max(w_1^T x + b_1, w_2^T x +$

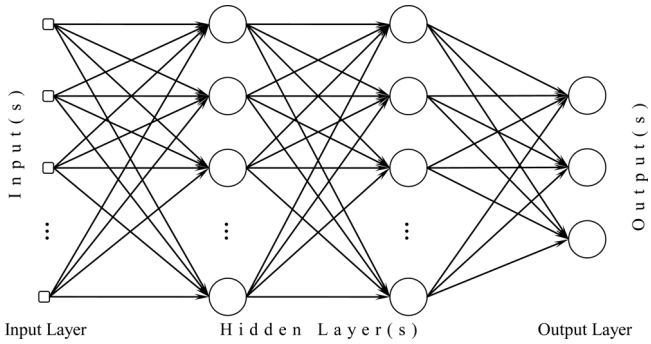


Figure 54: A multilayer perceptron visualized as graph. (Source: TUMwiki.)

b_2) using two parameter sets. Thus it doubles the number of parameters, but enjoys the benefits of the relu function and avoids the dying unit problem.
Concerning the question which activation function to use one strategy can be as follows: use the relu activation, pay attention w.r.t. the learning rate (not to high) and possibly monitor the fraction of "dead units." in a network. If this is a matter of concern, one may try the Leaky relu nonlinearity of the maxout nonlinearity. Using sigmoid or tanh functions is not advised.

Network architecture. A means to display neural networks are (acyclic) graphs with the neurons as notes. see Figure 54 for a visualization as graph of a fully connect network with multiple layers called multilayer perceptron. The organization in form of layers is typical. (Note that in the example, each layer corresponds to a weight matrix W_i as in (135) and (136), and the weights of the jth neuron in the ith layer are stored in the jth row of the matrix W_i.)
Note that typically the output layer neurons have no activation function. The last output layer typically yields the class scores (in classification tasks) and a kind of activation can be incorporated into the loss function.
The size of neural networks is typically either measured by the number of parameters or by the number of neurons. (To get an idea, convolutional networks can contain orders of 100 million parameters.)
The forward pass consists of alternating matrix multiplication and application of the activation function.

Hyperparameter: setting number of layers and their sizes. By increasing the size and number of layers, the space of representable (classification) functions grows, i.e., larger neural networks can represent more complex functions. (To get an impression we refer to the demo https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html.) A larger space of representable functions also makes it easier to overfit the training data, i.e., the noise in the training data is fit into the model. Overfitting hampers generalization, i.e., the application to test data does not work well although the accuracy on the training data is high.
Means to avoid overfitting are regularization techniques. Choosing smaller networks may seem to have a regularizing effect; this is, however, not advised by practitioners in many places in connection with neural networks; instead it

is advised to choose larger networks and to employ other regularization techniques on the larger networks (such as L2 regularization which we already saw, and others we see later) to avoid overfitting.

Data preprocessing. In the following we consider the data matrix X of size $N \times D$ consisting of N data items of dimensionality D.
We first consider mean subtraction which is the most common form of preprocessing. We consider the ith feature, compute the mean for the feature and subtract it from each data item. We do this for all features. In case of images, we hence compute a mean image (pixelwise average, not average value of a particular image) and subtract the mean image from each training image.
Next, we consider normalization which amounts to scaling each feature separately such that the features live on approximately the same scale. One way is to divide each feature dimension of the centered data by its standard deviation, the other is to normalize each dimension such that the minimum equals -1 and the maximum equals 1 (seperately for each feature dimension.) This preprocessing is important if the different input features have different scales or meaning. In case of images, the different features are the different pixels, and therefore already live on approximately the same scale which makes scaling less important.
For completeness, we mention the preprocessing techniques of PCA and whitening. Since they are not widely used in our context, we refer the interested reader to the literature for further reading.

Weight initialization. Initializing all weights by 0 (or constant in general) is not a good idea since then every neuron in the network computes the same output in every stage of training. More precisely, initially all weights and thus all neurons are equal and this does not change during the iteration since also same gradients are computed during backpropagation.
Therefore, it is common to initialize the weights of the neurons by small (random) numbers "to break the symmetry." Typically, either independent Gaussian random variables $N(0, \sigma^2)$, e.g., $\sigma^2 = 0.01$ or independent uniformly distributed random variables are used. There is one more thing to consider: the variance of a randomly initialized neuron (Gaussian noise) grows with the number of inputs. This can be avoided by scaling with $1/\sqrt{n})$, where n is the number of its inputs. Doing so, all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence. It is recommended by He et al. [12] for relu neurons, to chose the variance of neurons in the network by $2.0/n$. For fully connected layers, this results in choosing the initializations

$$W_i^{kl} \sim \sqrt{2.0/n_i} \cdot N(0, 1) \tag{137}$$

where n_i is number columns of W_i. It is possible and common to initialize the biases to be zero.

Batch normalization. We can think about batch normalization as doing preprocessing at every layer of the network integrated into the network itself in a differentiable manner. It was developed in [13]. It amounts to inserting a BatchNorm layer after a (fully connected) layers before applying the non-linearity.

In this layer the mean w.r.t. the corresponding minibatch is subtracted and the result is devided by the empirical standard deviation w.r.t. the minibatch. Details may be found in [13].

Regularization. A first means of regularization employed for many learning tasks are L^1 and L^2 regularization we already saw before as well as combinations. L^1 and L^2 regularization are applied to the weights as seen before. We here also point out the possibility of combining them which is the called elastic net regularization; more precisely the regularizer is given by

$$R(W) = \lambda_1 \|W\|_1 + \lambda_2 \|W\|_2^2 \tag{138}$$

where λ_1, λ_2 are positive hyperparameters regularizing the respective regularization strength and $\|W\|_1$, $\|W\|_2^2$ denote the L^1 and squared L^2 norm of the weights (stacked together in on column vector say.) L^1 regularization tends to produce weight vectors which are sparse, i.e., many components are (theoretically) 0 or practically close to 0. This can be important in connection with feature selection. However, it is frequently reported that L^2 regularization yields better performance the L^1 regularization in the general case.
Another mean are using hard max-norm constraints for regulariztion: e.g., one may enforce the weights w of each neuron to fulfill $\|w\|_2 < c$ for a hyperparameter c. This corresponds to performing the parameter update as normal, and then enforcing the constraint by clamping the weight vector w to satisfy $\|w\|_2 < c$. This way of regularization also avoids the weights becoming too large when the learning rate is chosen too large.
Another regularization technique specific to neural network is *dropout*. While training, dropout is implemented by keeping a neuron active with some probability p (hyperparameter), and setting it to zero else. In case of a multilayer perceptron, this is done as follows: In each training step, we compute the output of the first layer $H_1 = \max(0, W_1 X) + b_1$ and now switch it off with probability $(1-p)$ to get $U_1 = \text{pointwiseMult}(H_1, B_1)/p$ where B_1 is an independent Beroulli random field of same size as H_1. The division by p is motivated by keeping expectations constant. We proceed analogously with the other layers (except the final one): $H_k = \max(0, W_k U_{k-1}) + b_k$, followed by $U_k = \text{pointwiseMult}(H_k, B_k)/p$. Testing remains unchanged.
As mentioned for linear classification already it is not common to regularize the bias parameters. However, it is reported that regularizing the bias rarely leads to significantly worse performance. Further, it is most common to use a cross-validated single global L^2 regularization parameter. It is also common to combine this with dropout applied after all layers. The value of $p = 0.5$ is a reasonable default, and can be further tuned on validation data.

Loss functions. Besides the regularization loss we need a data loss term. In analogy to linear classifiers, we interpret its output as score produced by a network and denote it by $s = f(x_i, W)$ with the weights gathered in W. As in the case of linear classification, we consider the hinge/SVM loss and the softmax loss.
We point out that when the set of labels is very large (e.g., words in the English dictionary, or the ImageNet data which contains 22,000 categories), computing

the full softmax probabilities becomes expensive and approximations may be employed.

Hyperparameter optimization. The most important hyperparameters to tune are the initial learning rate, the learning rate decay schedule as well as regularization strength (L^2 penalty or dropout).
It is important to keep in mind that performing hyperparameter search can take many days/weeks for larger networks so that some automation can be needed.
It can save time to use one validation fold instead of cross-validation.
Search for hyperparameters on log scale: For example, a typical sampling of the learning rate could be 10^k with k drawn from a uniform distribution on say $[-6, 1]$. The strategy can also be employed for the regularization strength. (If the best values are near the border, and the parameter is not bounded naturally, extend the interval.) In contrast, the (bounded) dropout parameter is usually searched in the original scale, e.g., using the uniform distribution on $[0, 1]$. Further, prefer random search to grid search.

Hints for debugging. Many useful hints in particular important for debugging may be found on https://cs231n.github.io/neural-networks-3/.

Case study. For a worked out toy example we refer the reader to https://cs231n.github.io/neural-networks-case-study/.

6.5 Convolutional Neural Networks

Convolutional neural networks make the explicit assumption that the inputs are images. An interpretation is that they incorporate the feature extraction into the network, more precisely, the features are learned (initial layers) and (often) combined with the classical fully connected networks (final layers) for classification. (We note that classification however is not the only task CNNs are used for.)
In a nutshell, a CNN is in the basic version a list of layers that transform the image into a output vector containing class scores (in the case of classification.) There are a few distinct types of Layers such as CONV layers, FC layers, RELU layers, and POOL layers. Each layer gets a 3d volume as input and transforms it to an 3d output volume (of different size). (For later optimization purposes notice that it is important to have analytic gradients for the layers.) Each Layer may contain parameters and additional hyperparameters.
Important building blocks of the initial layers of a neural network are the convolutional layers. The input of the first layer typically is an RGB image (CIFAR 10: $32 \times 32 \times 3$.) The layers are basically convolution kernels (with the sliding taking place w.r.t. width and heights which are the first two dimensions and summing w.r.t. to the third dimension called depth.) The "convolution kernels" are of small size so that each neuron in a layer is associated with a spacial location and only connected to the output of the previous layer in a small neighborhood of this spacial location (instead of all previous-layer neurons in a fully-connected network.) Non-locality is reached by down-sampling which is realized by pooling layers. The final layers are typically fully connected and

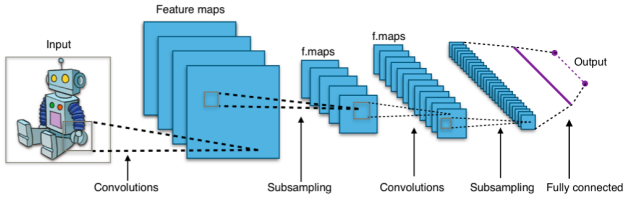


Figure 55: Schematic design of a first generation convolutional neural network (Source: Wikipedia.)

(for classification) the final output layer has dimension $1 \times 1 \times l$ with l being the number of classes and the result representing a score as previously. For a visualization see Figure 55. We discuss different types of layers in more detail next.

Convolutional layers. The parameters of the CONV layer are a set of filters coefficients. (These coefficients are not fixed, but learned from the data by parameter estimation via loss minimization.) Every filter has a small support spatially (width and height dimension), and extends through the full depth of the layer input. A typical filter kernel in the first layer might have size $5 \times 5 \times 3$ (for 3 channel RGB input.) In later layers, the last dimension may be larger, e.g., 32 resulting in filter sizes of $5 \times 5 \times 32$. In the forward pass we convolve each filter across the width and height dimensions and sum w.r.t. the depth which amounts to computing dot products between the filter kernel entries and the shifted input image at any position. Note that using k filters results in an output depth of k; e.g., using 32 filters in the first layer results in an output of depth 32. The lth filter's output is stored in the lth slice.
Three hyperparameters control the size of the output of a convolutional layer: the depth, stride and zero-padding. We already discussed the depth which corresponds to the number of filters used in the layer.
The second parameter is the stride which is another name for the down-sampling rate. A stride of 1 corresponds to "moving the filters one pixel at a time when sliding over the image". A stride of k corresponds to downsampling the filter output at rate k, i.e., only taking every kth pixel of the result. Since the left-out pixels need not be computed, using a stride of k, "the filters jumps k pixels at a time when sliding over the image."
As common for convolution on images padding is a topic here as well. It is used to get outputs with spatial dimensions agreeing with the input (for stride 1, proper halving for stride 2 etc.) Typically, zero-padding is used. The size of this zero-padding is a hyperparameter (which is typically not cross-validated, but set to maintain spatial dimensions for instance.)

Example 6.6. The Krizhevsky et al. architecture (AlexNet) that won the ImageNet challenge in 2012 has images of size $227 \times 227 \times 3$ as input. In the first layer, it uses filter size $F = 11$, stride $S = 4$ and no zero padding. It used 96 filters, resulting in depth $K = 96$. As a result, the first CONV layer output had size 55×55.
In summary, AlexNet contains eight layers: five convolutional layers, some followed by max-pooling, and the last three were fully connected layers. It

used the non-saturating relu activation function. Details may be found in [17].

We summarize the CONV Layer.

- Input: volume of size $W_1 \times H_1 \times D_1$.
- Hyperparameters: number of filters K, filter size F, stride S, amount of zero padding P.
- Output: volume of size $W_1 2 \times H_2 \times D_2$, where $W_2 = (W_1 - F + 2P)/S + 1$, $H_2 = (H_1 - F + 2P)/S + 1$ and $D_2 = K$.
- There are $F \cdot F \cdot D_1$ weights per filter, hence $F \cdot F \cdot D_1 \cdot K$ weights and K biases per layer in total.

The backward pass for a convolution operation (for both the data and the weights) is also a convolution (but with spatially-flipped filters).
In the context of Conv-Nets also 1×1-convolutions makes sense: they sum w.r.t. the channels/depth/previous filter outputs. Also so-called dilated convolutions are sometimes used: here the input is down-sampled before the convolution takes place; e.g. only every second input item is applied to the filter.

Pooling layer. For down-sampling pooling layers are frequently used between successive convolution layers. (Max-)pooling operates independently on every depth slice of the input and reduces the size to a fourth, using the max operation for 2×2 blocks.
In addition to max pooling, pooling can also employ other functions, such as averaging (average pooling). Typically, max pooling is used due to working better in practice.
As we have seen, the derivative for a max function yields only nonzero values on the largest component where it has value 1. During backpropagation it amounts to routing the gradient to the input component with higest value. This can be stored during the forward pass and used in the backward pass later.
Frequently (in particular in later designs), pooling layers are omitted and the down-sampling is obtained by using convolutions with stride 2 or larger strides.

Fully-connected layer. They have the same shape as explained before. Typically, the first fully-connected layer obtains the output of the last convolutional layer (potentially followed by pooling) as an input.

Connection between fully-connected and CONV layers. Each CONV layer (with already trained weights) can be been as a fully connected layer in a analogy to realizing convolution as matrix vector multiplication (cf. lecture on linear system's theory.)
Conversely, a fully connected layer with l neurons acting on an $k \times \ k \times d$ volume can be represented (and even trained) by applying convolution without padding of spacial size $k \times k$ and depth l. Further fully connected layers can then be realized using 1×1 of respective depth, i.e., for K neurons, we may employ $1 \times 1 \times K$ convolution. In this respect, a CONV net can be implemented using no fully-connected layers explicitly.

Relu layer. Finally, the relu layer applies the relu activation function elementwise.

Stacking layers. Common ConvNet architectures stack some CONV-RELU layers followed by pooling, and repeat this until the volume has spatially small size. Then fully-connected layers are employed where the last layer holds the output, e.g., the class scores in classification. Stacks of conv layers with small filters are preferred to one layer with large filter size. This yields the same "view" at the cost of less parameters. As a disadvantage more memory is needed to hold the intermediate layer results when doing backpropagation.
Typically, when approaching a classification problem, one is not starting out to define a new type of architecture. Instead, one might look for architectures working best on the ImageNet data say, download a corresponding pretrained model and tune it on the data to process.
A demo on CNN using CIFAR-10 may be found at https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html.

Hyperparameters: layer sizes. The (spatial) image size the input layer accepts should be divisible by 2^k for larger k. Examples are 32 (e.g., CIFAR-10), 64, 96, or 224 (e.g., ImageNet), 384, and 512.
Especially in the first layers of the discussed ConvNet architectures, the amount of memory required can become large very quickly in the setting presented. For example, starting with images of size $224 \times 224 \times 3$ and three 3×3 CONV layers with 64 filters each and zero-padding 1 would create three consecutive volumes of size $224 \times 224 \times 60$ through the first three layers. This amounts to about 72 MB of memory per image, for both activations and gradients for backpropagation. To avoid problems, frequently downsampling right in the first layer is performed by, e.g., using a stride, e.g., of 2 or 4 reducing memory consumption by a factor of 4 or 16.

Examples. In the in 1990's Yann LeCun and his team developed the first successful applications of CNNs. The best known is the LeNet architecture (reading zip codes, digits).
A break through for CNNs in Computer Vision was AlexNet. AlexNet won the ImageNet Large Scale Visual Recognition Competition (ILSVRC) challenge in 2012. The network architecture is similar to LeNet, but deeper and bigger.
ZF Net won the ILSVRC challenge in 2013 and improved AlexNet by optimizing the architecture hyperparameters.
GoogLeNet won the ILSVRC challenge in 2014. Its main contribution was to significantly reduce the number of parameters in the network (4M, compared to AlexNet with 60M) by developing a so-called inception module. Additionally, this paper uses average pooling instead of fully connected layers at the top of the ConvNet, eliminating a large amount of parameters that do not seem to matter much.
VGGNet was second in the ILSVRC challenge in 2014. A main contribution was in showing that the depth of the network is a critical component for good performance. The design is very appealing being extremely homogeneous and simple: there are only 3×3 convolutions (with stride 1 and pad 1) and 2×2 pooling from the beginning to the end. It is rather expensive to evaluate and

uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer, and it was since found that these FC layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.
ResNet won the ILSVRC challenge in 2015. The major contribution are special skip connections avoiding gradients from "dying" through backpropagation in deep networks. The architecture is also missing fully connected layers at the end of the network.

Visualization. A straight-forward visualization is to show the activations of the network during the forward pass; see https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html. If some activation maps are all zero for many different inputs this may indicate dead filters caused by too high learning rates.
A second strategy is to visualize the weights (which have interpretation as filter kernels on the first layer). However, the weights are useful to visualize because well-trained networks usually display nice and smooth filters without any noisy patterns also in higher layers.
Another visualization technique is to display those images of a large dataset that maximally activate a neuron. This may be interpreted in the direction that we see what the neuron is most activated by in its receptive field.
Another visualization technique consists of investigating images one by one and occluding parts of the respective image; see Figure 56. More precisely, using a sliding occlusion window, one may plotting the classification probability of the result w.r.t. the class of interest as a heat map w.r.t. the sliding window center. if the net works properly the probability should go down when the object from the class of interest is occluded.

Transfer learning. It is common to pretrain a ConvNet on a very large dataset such as ImageNet and then use the ConvNet either as an initialization or as a fixed feature extractor for the classification task of interest.
When using a ConvNet as feature extractor, we remove the last fully-connected layer from the pretrained net (which outputs e.g. 1000 class scores when the net was trained on ImageNet) and keep the weights of the rest fix. As output, we get the output of the second before last layer features for our new data.
The second strategy is to replace the last layer and to retrain the classifier on top as well as to fine-tune the weights of the pretrained network by backpropagation. It is possible to fine-tune all the layers, or it's possible to keep some of the earlier layers fixed and fine-tune later layers only.
Which kind of transfer learning to use primarily depends on the size of the new dataset (few or many saples) and its similarity to the original dataset. If the new dataset has only few examples and the images are similar to the original dataset, the best idea might be to train a linear classifier on the CNN codes, i.e., the second before last level output of the CNN. If the new dataset has many examples and the images are similar to the original dataset, one might try to fine-tune through the full network. If the new dataset has only few examples and the images are very different from the original dataset it could work better to train the SVM classifier from activations somewhere earlier in the network. If the new dataset has many examples and the images are very different

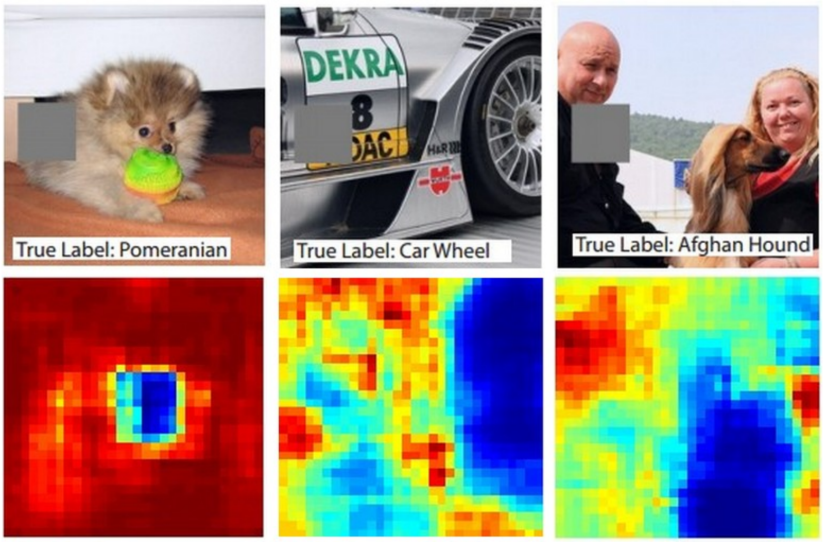


Figure 56: Mostly, the heatmap shows lower probabilities when the objects corresponding to the classes of interest are occluded. In the right image it is interesting that the highest probability for Afgan Hound is reached when the man's head is occluded. (Source: Li Fei-Fei et al.)

from the original dataset one might consider to train a ConvNet from scratch. However, it may be still beneficial to initialize with weights using a pretrained model and fine-tune through the entire network.
It's common to use a smaller learning rate for ConvNet weights that are being fine-tuned, in comparison to the (randomly-initialized) weights for the new linear classifier that computes the class scores of the new dataset.

More advanced optimization strategies. As we have seen the optimization strategy for parameter estimation by minimizing the loss function is a central topic. So far we have discussed stochastic/batch gradient descent (SGD.) Here, the update was

$$x_{n+1} = x_n - \text{learningRate} \cdot \nabla f(x_n)$$

where the learning rate is a hyperparameter.
Momentum update is another approach that typically yields faster convergence in the context of neural networks. A motivation is as follows: By initializing the parameters with random numbers we set a particle with zero velocity to some location. The optimization process can then be seen as rolling the particle on the landscape given by the graph of the loss function. The force on the particle corresponds to the gradient of the potential (defined as the loss function.) the momentum update is given by

$$v_{n+1} = \mu v_n - \text{learningRate} \cdot \nabla f(x_n), \qquad x_{n+1} = x_n + v_{n+1}, \tag{139}$$

We use an additional (velocity) variable v and an additional hyperparameter μ. μ physically corresponds to the coefficient of friction but in this context is

referred to as *momentum*. A typical value is 0.9. Typical values for cross validation are $0.5, 0.9, 0.95, 0.99$.
The Nesterov momentum method yields stronger theoretical converge guarantees for convex functions and also consistently works slightly better than standard momentum in practice. Instead of evaluating the gradient at the previous iterate it is evaluated at an first order approximation $\tilde{x}_{n+1}$ of the resulting (unknown next iterate)

$$\begin{aligned}\tilde{x}_{n+1} &= x_n + \mu v_n,\\ v_{n+1} &= \mu v_n - \text{learningRate} \cdot \nabla f(\tilde{x}_{n+1}),\\ x_{n+1} &= x_n + v_{n+1}.\end{aligned}$$

When training deep networks, it is usually helpful to anneal the learning rate over time. Annealing means that the learning rate is reduced by some factor every few epochs. Typical schemes involve halving the learning rate every 5 epochs, or reducing it by factor 0.1 every 20 epochs. The numbers depend heavily on the type of problem and the model. A heuristic is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (such as 0.5) when the validation error stops improving. There is also exponential decay used classically for annealing which has the form $\alpha = \alpha_0 e^{-kt}$ where α_0 and k denote hyperparameters and t is the iteration number. Finally, there is $1/t$-decay which has the form $\alpha = \alpha_0/(1+kt)$ with hyperparameters a_0 and k and iteration number t. If there is sufficient computational budget it is common in parameter annealing to employ slower decay and train for a longer time to typically obtain better results.
The discussed approaches set the learning rate equally for all parameters. Now we discuss adaptive methods.
Adagrad is an adaptive learning rate method, computing in the jth component of the $i+1$th iterate of the parameter vector x

$$\begin{aligned}\text{cache}_{i+1}^{(j)} &= \text{cache}_i^{(j)} + \nabla_j f(x_i)^2 \\ x_{i+1}^{(j)} &= x_i^{(j)} - \frac{\text{learning\_rate}}{\sqrt{\text{cache}_{i+1}^{(j)} + \text{eps}}} \cdot \nabla_j f(x).\end{aligned} \tag{140}$$

Weights that have high absolute derivative/change rate will have their effective learning rate reduced, while weights that have small change rates will have their effective learning rate increased. The term eps (set in the range between 10^{-4} and 10^{-8}) is employed to avoid division by zero. In case of Deep Learning, the monotonic learning rate turns out to be too aggressive and to stop the learning process too early.
RMSprop is a more effective adaptive learning rate method. The RMSProp update adjusts the Adagrad method: in (140) the first line is replaced by

$$\text{cache}_{i+1}^{(j)} = \text{decay\_rate} \cdot \text{cache}_i^{(j)} + (1 - \text{decay\_rate}) \nabla_j f(x)^2 \tag{141}$$

The decay rate is a hyperparameter with typical values $0.9, 0.99, 0.999$.
Adam is an update scheme that looks a bit like RMSProp with momentum

together with a bias correction. It is given by

$$
\begin{aligned}
m_{i+1}^{(j)} &= \beta_1 m_i^{(j)} + (1-\beta_1)\nabla_j f(x) \\
\tilde{m}_{i+1}^{(j)} &= m_{i+1}^{(j)}/(1-\beta_1^j) \\
v_{i+1}^{(j)} &= \beta_2 v_i^{(j)} + (1-\beta_2)\nabla_j f(x)^2 \\
\tilde{v}_{i+1}^{(j)} &= v_{i+1}^{(j)}/(1-\beta_2^j) \\
x_{i+1}^{(j)} &= x_i^{(j)} - \frac{\text{learning\_rate}}{\sqrt{v_{i+1}^{(j)}} + \text{eps}} m_{i+1}^{(j)}
\end{aligned}
\tag{142}
$$

We refer the reader to the paper [15] for the details.

References

[1] S. Beucher and F. Meyer. "The morphological approach to segmentation: the watershed transformation". In: *Mathematical morphology in image processing* 34 (1993), pp. 433–481.

[2] J. Beyerer, F. P. León, and C. Frese. *Automatische Sichtprüfung: Grundlagen, Methoden und Praxis der Bildgewinnung und Bildauswertung*. Springer-Verlag, 2016.

[3] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, R. E. Tarjan, et al. "Time bounds for selection". In: *J. Comput. Syst. Sci.* 7.4 (1973), pp. 448–461.

[4] M. L. Brady. "A fast discrete approximation algorithm for the Radon transform". In: *SIAM Journal on Computing* 27.1 (1998), pp. 107–119.

[5] J. Cousty, G. Bertrand, L. Najman, and M. Couprie. "Watershed cuts: Minimum spanning forests and the drop of water principle". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.8 (2008), pp. 1362–1374.

[6] P. F. Felzenszwalb and D. P. Huttenlocher. "Distance transforms of sampled functions". In: *Theory of computing* 8.1 (2012), pp. 415–428.

[7] M. Frigo and S. G. Johnson. "The design and implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.

[8] J. Gil and M. Werman. "Computing 2-D min, median, and max filters". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15.5 (1993), pp. 504–507.

[9] J. Y. Gil and R. Kimmel. "Efficient dilation, erosion, opening, and closing algorithms". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.12 (2002), pp. 1606–1617.

[10] R. C. Gonzalez and R. E. Woods. *Digital image processing*. Pearson, 2017.

[11] H. Götz W.and Druckmüller. "A fast digital Radon transform—An efficient means for evaluating the Hough transform". In: *Pattern Recognition* 29.4 (1996), pp. 711–718.

[12] K. He, X. Zhang, S. Ren, and J. Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[13] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[14] B. Jähne. *Digitale Bildverarbeitung*. Springer, 2013.

[15] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[16] D. Knuth. *The art of computer programming*. Addison-Wesley Professional, 1998.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[18] K.-F. Man, K. S. Tang, and S. Kwong. *Genetic algorithms: concepts and designs*. Springer Science & Business Media, 2001.

[19] S. Perreault and P. Hébert. "Median filtering in constant time". In: *IEEE transactions on image processing* 16.9 (2007), pp. 2389–2394.

[20] W. H. Press. "Discrete Radon transform has an exact, fast inverse and generalizes to operations other than sums along lines". In: *Proceedings of the National Academy of Sciences* 103.51 (2006), pp. 19249–19254.

[21] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[22] M. Sezgin and B. Sankur. "Survey over image thresholding techniques and quantitative performance evaluation". In: *Journal of Electronic imaging* 13.1 (2004), pp. 146–166.

[23] P. Soille. *Morphological image analysis: principles and applications*. Springer Science & Business Media, 2013.

[24] P. Toft. "The radon transform". In: *Theory and Implementation (Ph. D. Dissertation)(Copenhagen: Technical University of Denmark)* (1996).

[25] M. Van Herk. "A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels". In: *Pattern Recognition Letters* 13.7 (1992), pp. 517–521.

[26] B. Weiss. "Fast median and bilateral filtering". In: *ACM SIGGRAPH 2006 Papers*. 2006, pp. 519–526.