Computer Vision: Foundations

Lecture Notes

Lecturer: Prof. Dr. Fred Hamprecht

Edited by: Nils Friess

Last updated: July 24, 2020

Contents

Todo list

Week 1 Introduction

Summary In this chapter, a short introduction to the topic of computer vision is given. Thereafter, convolution is introduced. It is discussed how (discrete) convolution can be written as a matrix product and the notion of separability is discussed.

Overview

Computer Vision is a very broad field with applications in many areas. For instance, computer vision is

- useful in consumer devices (e. g. fingerprint login on smartphone);
- used as machine vision in quality control (e. g. check that all pixels of a screen work which is a rather easy task, or to check that there are no scratches, which is a rather hard task);
- making life-and-death decisions (e. g. in autonomous emergency braking, cyclist and pedestrian detection. Applications in this area needs extremely high accuracy.

Considering the last point, an example procedure would be to take last n frames (e. g. $n = 11$) and decide based on them whether or not to brake. Clearly, the false positive rate must be extremely low (i. e., , do not want to brake if it is not necessary). Additionally, we would like a low false negative rate. Finally, the process should process the data in real-time which corresponds to $\sim$ 30 megabyte of input per second.

Computer Vision is a compute-intensive endeavor and only two hand ful of algorithms are sufficiently efficient, i. e., work at high scale and will make it into consumer devices. We will therefore study these two hand ful of algorithms in-depth. They can then be combined in complicated pipelines. Although we will study some of these pipelines we will mainly focus on the building blocks (see Table 1.1).

Input	Output	Task
Image	0/1	Image classification
Image	One class per pixel	Semantic/pixel segmentation
Image	Which pixel belong to which instance	Instance segmentation
Image	Pose of one or more humans	Pose estimation
Video	Tracks of all targets	tracking

Table 1.1: Example tasks in computer vision {tab:ex:tas

Lecture 1.1 Linear Filters: Convolution[1]

Convolution is useful for

- Smoothing (Not SOTA)
- Edge/Blob detection
- General: Feature extraction

Has been mainstay of image analysis since it's very cheap (still matters now) and is well-understood.

1D Convolution

Consider the mean square estimator for $\{y_i\} \in \mathbb{R}$

$$\hat{y} = \arg\min_y \sum_{i=1}^{n} (y_i - y)^2$$

that is given by (can be seen by letting the derivative of the sum above be zero)

$$\hat{y} = \frac{1}{n} \sum_{i=1}^{n} y_i \,.$$

Using the same idea but introducing weights $w_i \geq 0$, i. e.,

$$\hat{y}_w = \arg\min_y \sum_{i=1}^{n} w_i (y_i - y)^2$$

yields

$$\hat{y} = \frac{\sum_i w_i y_i}{\sum_i w_i} \,.$$

If, in addition, the weights depend on the distance from x only, this can be rewritten as (note that this no is a function of x)

$$\hat{y}_w(x) = \arg\min_y \sum_{i=1}^{n} w_i (x - x_i)(y_i - y)^2$$

with solution

$$\hat{y}(x) = \frac{\sum_i w_i (x - x_i) y_i}{\sum_i w_i (x - x_i)}$$

[1] See also Appendix B

and in the case of equidistant observations this simplifies to

$$\hat{y}_l = \frac{1}{\sum_i w_{l-i}} \sum_i w_{l-i} y_i \,,$$

which can be interpreted as the convolution of the signal y and a weight function w. The Figure below demonstrates the results of smoothing a perturbed signal using two different weight functions, one being a NN kernel (that corresponds to a characteristic function of an interval) and one being a Epanechnikov kernel (that corresponds to a parabola that has been cut off at its roots).

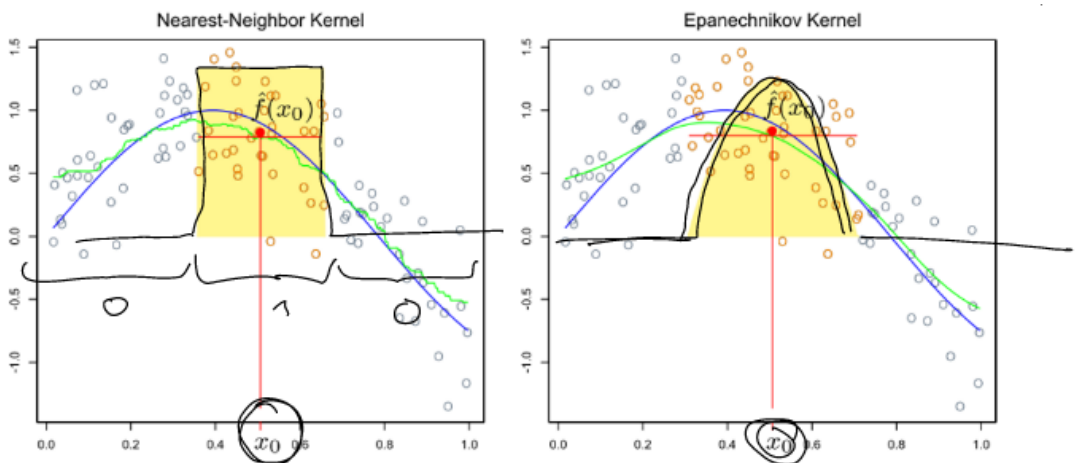


Figure 1.1: Box kernel and Epanechnikov kernel

{fig:conv:k

Different filters can produce very different results ($\rightsquigarrow$ filter optimazation). This is one instance of *discrete convolution*

$$\sum_{i=0}^{n-1} f_{l-1} g_i =: (f * g)_l$$

Properties:

- Convolution is **commutative**: $f * g = g * f$
- Convolution is **associative**: $f * g * h = f * (g * h) = (f * g) * h$
  - Important in practice, especially for image analysis
- Convolution is **distributive**: $f * (g + h) = f * g + f * h$

Important convolution filters include

- $f_i \geq 0$ smoothing (see above)
- $f_i = \delta_{i-s}$ shifting

- $f = 1/2(1\ 0\ -1)$ central finite difference (analogously non-central FD)
- $f = [1 \quad -1] * [1 \quad -1] = [1 \quad -2 \quad 1]$ second derivative

Note: 1D convolution can be written as matrix multiplication with a *Töplitz matrix*

$$\sum_i f_{l-1} g_i \qquad \text{vs.} \qquad \sum_i M_{li} v_i = [M \cdot v]_r$$

Example: Consider the convolution

$$\begin{aligned} &[1 \quad 0 \quad -1] * [g_0 \quad g_1 \quad g_2 \quad g_3 \quad g_4] \\ &= \begin{bmatrix} 0 & 1 & & & -1 \\ -1 & 0 & 1 & & \\ & -1 & 0 & 1 & \\ & & -1 & 0 & 1 \\ 1 & & & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix}, \end{aligned}$$

where the terms in the corners arise from periodic boundary conditions (i. e., the "-1-th" term is the last term and the "$n+1$-st" term is the first term).

2D Convolution

Convolution in higher dimensions can be defined analogously. Here, we have

$$\sum_i \sum_j f_{i,j} g_{u-i,v-j} =: (f * g)(u, v)\,.$$

In image analysis f is often small (e. g. 3×3) and g large (e. g. 3840×2160).

Some filters are **seperable** which means they can be written as an outer product $f_{i,j} = a_i \cdot b_j$ This allows for storage reduction (instead of storing the full matrix it suffices to store the vectors a and b)[2]. In practice you would use libraries because with the right memory layout, clever use of co-processors and GPUs the speed can be drastically increased. If the filter is separable into a and b as above, we can write

$$\underbrace{\sum_i a_i \quad \underbrace{\sum_j b_j g_{u-i,v-j}}_{\text{1D convolution of rows}}}_{\text{1D convolution of columns}}$$

There are different options to extrapolate at the boundary of the image.

[2] Aside: Some filters are not seperable but of low rank and using singular value decomposition we can find a suitable represantation such as $f_{i,j} = \sum_{k=1}^{r} a_i^k b_j^k$ with r small.

Week 2 Sensing, Subsampling and interpolation

{chap:02}

Summary This week's lecture is concerned with downsampling and upsampling. We show that the naïve approaches should not be used and discuss the different alternatives that should be applied. Further, we discuss how the operations can be written in terms of matrix multiplications.

Lecture 2.2 On Downsampling

To deal with very large images one often needs to downsample the image so that it fits into the memory of the GPU. Within the context of neural networks, one approach is to feed only a small patch of the image into the network. Of course, the network then ignores everything outside of this patch, even when we "slide" the small patch across the image. Another approach that makes use of the information across the whole image is to downsample the image. The *wrong* appproach would be to take the first pixels, then leave out a fixed number, then take the next pixel and repeat (see lecture notebook for an example). The correct way would be to first smooth (blur) the image and only then subsample.

The effect of the naïve approach can be explained as follows. Subsampling is a linear operation and can thus be written as a matrix multiplication. If we denote by a preceding downwards arrow the signal after subsampling, we can write

$$\downarrow g = h = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_7 \end{bmatrix} .$$

Every other sample is **completely lost**! Idea: Average two adjacent samples. This can be written as a convolution of a blur b with the signal g

$$\downarrow (b * g) = h' = \frac{1}{2} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_7 \end{bmatrix} = \begin{bmatrix} \frac{1}{2}g_0 + \frac{1}{2}g_1 \\ \frac{1}{2}g_2 + \frac{1}{2}g_3 \\ \vdots \\ \frac{1}{2}g_6 + \frac{1}{2}g_7 \end{bmatrix} .$$

All observations contribute to the result, i. e., less information is lost. There are also other

approaches; for periodic boundary conditions, the following matrix could be used

$$\frac{1}{4}\begin{bmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 \end{bmatrix},$$

which has some nice properties such as constant row and columns sums; also, unlike the previous filters, this filter is symmetric around the particular target sample.

We could take more samples around the target sample into account (fewer zero entries per the rows) which gives rise to the question of whether or not there optimal filters exist? In general, these filters will be data-dependent (this essentially is PCA), i. e., different filters for different data sets. To obtain a generally appllicable filter which is not data-dependent, we need to make some assumptions. A typical choice would be to assume the signal to be mostly low-pass (such filters will predominantly preserve low frequencies; we therefore assume the signal mostly consists of low frequencies). A perfect low-pass filter

- eliminates all frequencies above the new Nyquist limit (highest frequency that can be represented by sampling a signal uniformly), and
- leaves frequencies below the Nyquist limit completely untouched.

The optimal choice is (no proof given)

$$\operatorname{sinc} x = \frac{\sin \pi x}{\pi x}.$$

A plot is given in Figure 2.1. A possible discrete filter that we would use for discrete signals could then be approximately

$$\begin{bmatrix} 0 & -\frac{1}{5} & 0 & \frac{2}{3} & 1 & \frac{2}{3} & 0 & -\frac{1}{5} & 0 \end{bmatrix}$$

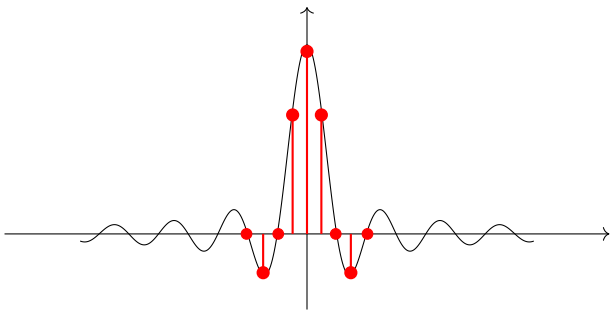


Figure 2.1: The sinc function and the values we would use in a discrete filter. {fig:sinc}

Note that the filter (or its upper envelope) decays very slowly as x becomes large. In an image that contains a "step" this can lead to ringing artefacts far away from the step.

Lecture 2.3 Upsampling/ Interpolating an Image

Interpolation is needed in upsampling and resampling of images but also when rotating images (as long as the angle that the image is rotated by is not a multiple of 90°). We again consider a simple $1D$ example. To that end we consider the signal $g = [g_1, g_2, g_3, g_4]$ which we want to upsample to twice its length. One straightforward approach would be to simply duplicate each component of the signal. In terms of a matrix operation, this can be written as follows

$$Mg = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ g_1 \\ g_2 \\ g_2 \\ g_3 \\ g_3 \\ g_4 \\ g_4 \end{bmatrix} = g^{\uparrow}. \tag{2.1}$$ {eq:box_fil

We see that this results in a piece-wise constant signal but often we would like something smoother. If we still require that the resulting upsampled image interpolates, i. e., that the following holds

$$(Mg)_{2i} = g_i$$

the matrix generally looks as follows

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 1 & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \text{e. g.} \qquad M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

We can also plot such filters as in Figure 2.2. It turns out that the hat filter, i. e., the filter described by the matrix above, arises when we convolve the box filter from (2.1) wit itself. That is, if we denote the box filter by Π, the tent filter is given by $\Pi * \Pi$. If we carry on with convolving the results again with the original box filter, i. e., if we compute $\Pi * \Pi * \Pi$ etc., the results become very smooth very quickly. More precisely, already Π^3 or Π^4 resemble a Gaussian kernel with the exception that they both have finite support (the box filter has a support of width 1, Π^2 has a support of width 2 etc.).

Both filters discussed above can be interpreted as to particular members of a larger family

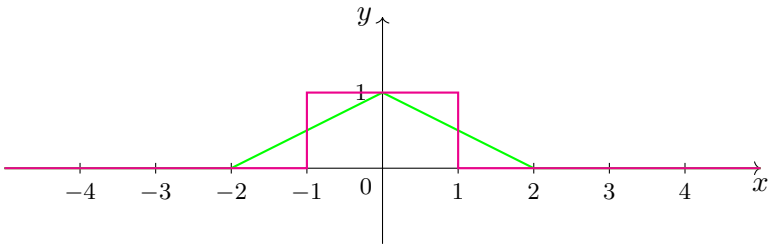


Figure 2.2: Box filter and tent filter. {fig:filter

of (non-interpolating) B-splines. Note that Π^n is only interpolating for $n \leq 2$; thus B-splines of higher order are smoothing filters but not interpolating filters. One way to look at these filters is as follows. We begin by expressing the interpolated continuous signal by a finite number of continuous filters multiplied with a discrete coefficient, i. e., a scalar

$$g^c(l) = \sum_i c_i b(l)\,.$$

As mentioned above, if $b(l)$ is a B-spline of higher order, it is not interpolating and thus, we need to account for this by choosing the coefficients c_i accordingly. If we want the signal to interpolate, the left and right hand sides of the equation above must be equal at the interpolation points. If we again write the filter as a Toeplitz matrix, this means that $g = Bc$ has to hold (here we collected the c_i into a vector). Solving for c yields

$$c = B^{-1}g\,.$$

Of course, in general the matrix inversion is very expensive. Using the fact that B (and also its inverse) is Toeplitz, the inversion can be implemented in terms of an infinite impulse response filter (IIR). This gives rise to so called cardinal B-splines. Another approach which tries to approximate sinc resampling is the Lanczos filter. The methods discussed above can also be regarded as computationally efficient approximations to Lanczos resampling.

Note that the "subjectively" best filter in general depends on the content of the image. While in some cases the sinc might be the best choice, sometimes the bilinear filter might produce better results. Thus, we would ideally like a filter that adapts to the image content, e. g. a neural network.

Week 3 CNNs: Deep Learning

{chap:03}

Summary In this chapter, we begin discussing the main part of the lecture: Deep neural networks. We start by briefly discussing the developments of deep learning within the last 20 years. We then motivate why we are only considering convolutional neural networks in this lecture and briefly look at the individual parts of a deep net.

Lecture 3.1 Shallow and Deep Learning

Example: Pixel classification/ semantic segmentation (assign each pixel to one of several classes). State of the art before 2000:

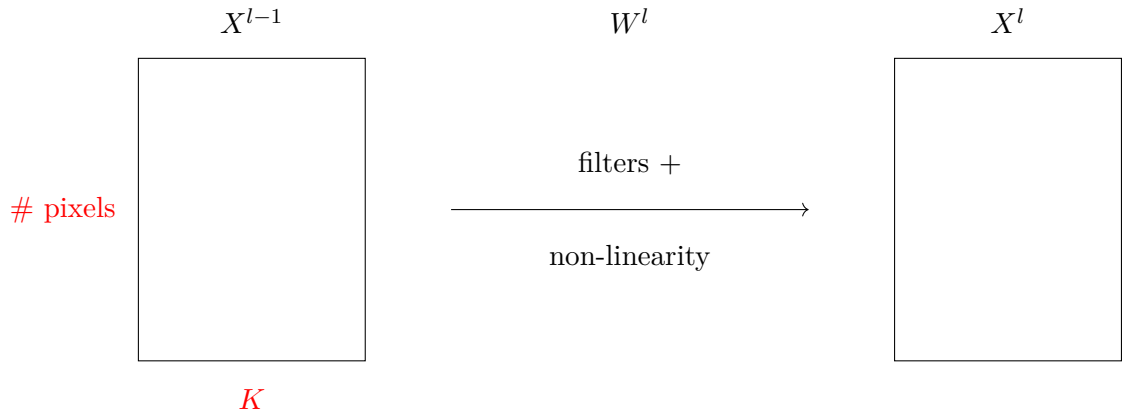
1. Take image
2. Compute features
3. Construct decision rule **by hand** (e. g. select foreground and background)
4. Mark pixels according to decision rule

State of the art 2000–2012:

1. Take image
2. Put into (nonlinear) filter bank that produced lots of images that summarise local contexts (i. e., produce features)
3. Take all of the images and classify using a shallow classifier (Support vector machine, random forest)

The second step is done by the user, i. e., the features are selected by hand. The latter part (the classifiers) are learned. After 2012 the advent of deep learning began. Take the image and repeatedly apply a set of filters and certain "non-linearities" (the *layers*; essentially the same as the filter banks as above). At the very end, a shallow classifier is used (in neural networks this would be a perceptron or logistic regression). The layers are learned and more importantly they are trained jointly! However, there are many hyper-parameters that need to be selected by the user; these are often very arbitrary.

The individual building blocks can be interpreted as follows. We have the input tensor X^{l-1} that gets mapped to the output tensor X^l by the filters and non-linearity W^l (which can also be seen as a tensor) to obtain the resulting tensor X^l



where K denotes the number of features. For simplicity, we assume the tensors are matrices. We can then write

$$x^l = \sigma\left(W^l x^{l-1}\right),$$

with x^{l-1}, x^l now being vectors. The matrix W^l corresponds to the (linear) filter bank and σ denotes the non-linear activation function.

How do we choose the linear operator W^l?

1. If every component of the output is a linear combination of all input components, the matrix W^l is dense. If we graph this relation as a network connecting the input and output components, the network would be *fully connected*. If the input is an image then this means that every pixel of the output depends on every pixel in the input. Often, e. g. for detecting boundaries of cells, it suffices to consider pixels in the vicinity of the pixel we currently try to classify and the rest of the image is not important.[1] Further, this approach is obviously also very computation-intensive since a huge number of parameters (the matrix entries) would need to be learned.

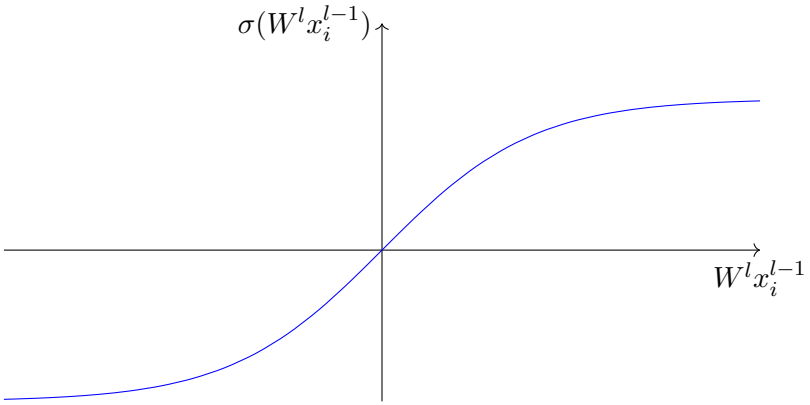
2. A reasonable simplifying assumption would be to make the decisions rules local, i. e., only pixels that are spatially close to the target contribute to the result. W^l would then have band structure.

3. Finally, if the image consists of similar structures (e. g. cells) then there is no need to use a different decision rule in different parts of the image. For instance, when we try to detect boundaries of cells, we want to make the same decision across the whole image. This corresponds to W^l being a convolution (that is reasonably small) which in turn means that W^l is of band Toeplitz structure. A typical choice are $3 \times 3 \times K$ convolution filters.

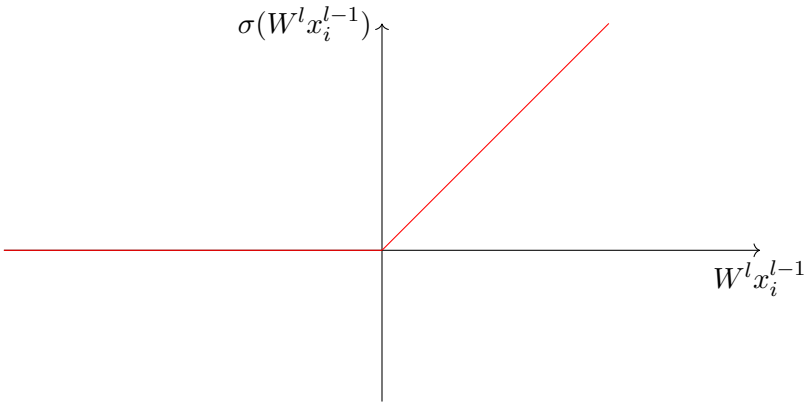
[1] Similarly, although the sinc theoretically is the perfect low pass filter, it is often not plausible why a pixel that is more than a few hundred pixels away of the current pixel should have an impact on this very pixel (which would be the case with infinitely-supported filters such as the sinc filter).

Thus, in the following, we will only consider convolutions as the linear operations W^l. The second ingredient in each layer is the non-linearity. One might first ask why there is the need for a non-linearity at all. A simple reason is that if we would have no non-linearities, the network would consists of linear functions applied after each other which could be expressed as a single linear map and thus we would not gain any benefits from using multiple layers (also we could only construct linear functions)[2].

For decades, people were using non-linearities of the following form



However, this can sometimes lead to difficulties for the learning algorithm, e. g. in cases where the value is high (i. e., in the above picture the value of $W^l x_i^{l-1}$ might lie somewhere on the very right) but the target value should actually be low. Since the curve is very flat at high values, it is difficult for the learning algorithm to decide what to change. Thus, people started using different activation functions and one that turned out to work particularly well is the so called ReLU (rectified linear unit)



which simply sets negative values to zero and leaves positive values unchanged, i. e., $\sigma(x) =$

[2] A more elaborate answer is given by Cover's Function Counting Theorem: The feature space of points we want to classify has to be larger when trying to separate them using a shallow or linear classifier as opposed to using a non-linear classifier such as a deep net.

$\max(0, x)$. There exist various variations of this functions with different aims, e. g. there are variants that "smoothen" the curve around the origin or some that do not set negative values to zero but assign them a (small) negative value etc.

We close this section with a short summary of the advantages of deep neural nets over traditional shallow classifiers. Most importantly, neural nets can achieve much more accuracy as compared to traditional methods. They do, however, require special hardware to be trained (such as GPUs) since the number of parameters is typically very large and the underlying optimisation problem is non-convex which makes is hard to solve efficiently. But this also leads to the following benefit: if you have better hardware, your network will train faster. With shallow classifiers this is not the case. Both approaches require much expertise but especially with deep networks people are trying to provide simple-to-use plug-and-play solutions that can be used by everyone.

Lecture 3.2 Training of a neural network

{sec:train-

The training of deep network is easy to write down but is a very hard problem to solve efficiently. In essence, a neural network is a non-linear function that we try to fit to given data using the input x_i (the ith input image) and the parameters W^l. We can write this as

$$f(x_i; \{W^l\}_{l=1}^L) = \sigma(W^L(\sigma(W^{L-1} \cdots \sigma(W^2\sigma(W^1 x_i))) \cdots)) \,.$$

Depending on the specific task, f can output a single number (e. g. the number of pedestrians, the number of cells), a vector of responses, an output image (e. g. where each pixels is colour-coded to a specific class) etc. At train time we are trying to find parameters W^l that make output useful, where useful in this context means that the output is in some sense close to a given target. In other words, we are trying to find

$$\underset{\{W^l\}_{l=1}^L}{\arg\min} \sum_{t_i \in \mathcal{T}} \text{loss}\left(t_i,\ f(x_i; \{W^l\}_{l=1}^L)\right) \,,$$

where t_i is the desired **target**/output for input image i which is part of the **training set** $\mathcal{T}$. The **loss function** measures the discrepancy between the target and the prediction produced by the current choice of f.

Obviously we have to make certain choices before we can start to train the network, which we discuss briefly below.

1. Architecture Choosing a certain architecture essentially is the task of choosing f. As part of this decision, we have to decide on the dimensions of the matrices W^l (this is only one part, there are various other possible decisions, e. g. should a layer only depend on the previous layer? Should it depend on the output of the layer before that?). Due to this huge number of possible design choices, it is essentially impossible to know whether or not a

certain design decision is optimal. As we have discussed earlier, you are not completely free in your decisions because some are simply not possible due to hardware restrictions (more aspects of the architecture of CNNs are discussed in the next lecture). Another important part which turned out to greatly influence the performance of network is the choice of normalisations (discussed below).

2. Loss function There are a few popularly used loss functions (of course there also exists special loss functions for very complicated tasks). For classification, one would typically use a cross-entropy loss. If one class appears much more often than the others, the so called Sørensen-Dice loss is often a more suitable choice. For regression tasks, usually the loss is measured by the sum of squared deviations. It can also be very useful to solve so called *side tasks* simultaneously to solving the main task. This is discussed in more detail below.

3. Training set In general, one wants as much training data as one can get. To artificially increase the amount of training data, so called augmentations can be used. Typical approaches in image classification orr semantic segmentation include flipping or cropping the training image (and possibly its corresponding ground truth). Additionally, it can be beneficial to add noise or distort the image.

4. Optimisation Modern neural networks often have millions, sometimes billions, of parameters and consequently the goal of finding the optimal set of parameters that minimise the loss is a obviously compute-intensive problem. It is therefore crucial to use an efficient optimisation algorithm. This point is discussed in more detail in the next chapter.

Side Tasks for Self-Supervision

As mentioned above, it can be beneficial to force the network to learn additional side tasks. For instance, suppose we are training a network that finds certain things in a video sequence. A possible side task would be to try to predict one frame based on some of the previous frames. If the network is able to do this, we could argue that the it actually learned something about the structure of the data set. Another popular side task is denoising. Here, artificial noise is added to the training image, for example, one might replace one pixel of the image by a randomly drawn value. The side task would then be to predict the original value of the pixel. Other side tasks include

1. Figuring out rotations and arrangement of patches.

2. Estimating optical flow in a video (i. e., , displacement between frames).

3. Compression (Auto-Encoding) Here we ask the network to additionally train an encoder and a decoder. The encoder compresses the input vector, i. e., it reduces its dimension

while the decoder increases the dimension, i. e., it decompresses the smaller vector to the original size. The loss is the difference between the original image and the image after it was encoded and decoded. If this task is solved successfully then the network has found a compact representation of the dataset.

Such side tasks can often improve the performance of the network, in particular in cases where not much training data is available. Of course ideally we would want self-supervising side tasks, i. e., side tasks that do not require additional ground truth.

Week 4 CNN Architectures

{chap:04}

Summary This week, we discuss two particular variants of convolutional neural networks. The first is AlexNet, one of the most influential networks that was one of the first CNNs to outperform traditional methods in an image classification challenge. The second is a network for semantic segmentation, called the UNet. As part of improving UNet's performance, we also discuss two normalisation techniques.

Lecture 4.1 Convolutional Neural Networks (CNNs) for Image Classification

In 2012, the convolutional neural network AlexNet achieved an outstanding performance on the ImageNet Visual Recognition Challenge, scoring more than 10 percentage points lower in the top-5 error than the runner up. The associated research paper has been cited more than 60,000 times and is considered one of the most influential papers in computer vision. In the same year of the AlexNet publication, deep neural networks started to supersede and replace traditional methods.

The following image shows the structure of the VGG16 network. It can be seen as an improvement of AlexNet but the overall structure is very similar.

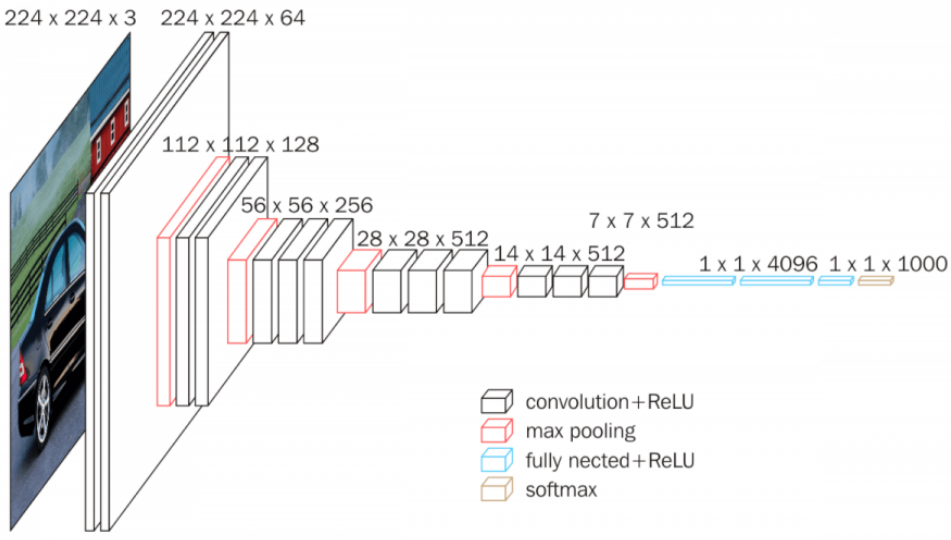


Figure 4.1: Structure of the VGG16 network.

{fig:vgg16}

This network was used to classify images with 1000 classes; this is why the output is $1 \times 1 \times 1000$[1].

[1]The training data is labelled using one-hot encoding. This means the to each image a 1000 dimensional vector

To gradually reduce the spatial dimension from 224×224 to 1×1, so called max pooling layers are used. In this particular case 2×2 max pooling with a stride of 2 was used, which means that the image is partitioned into 2×2 blocks of which the maximum of the respective four values is taken to replace this block. The result is a smaller image, more precisely the spatial dimension is reduced by two in each direction. The effect is visualised in the Figure below.

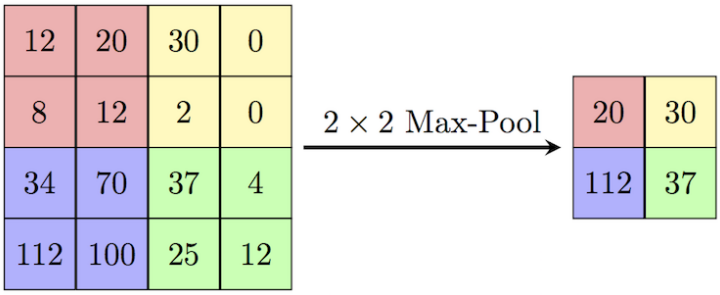


Figure 4.2: Effect of a 2×2 max pool. {fig:maxpoo

Most layers of the network are convolutional layers with a small receptive field of 3×3. For instance, the first two layers after the input layer consist of 64 distinct convolution filters (typically, in these early layers, these are edge detectors and the like).

After the stack of convolutional layers, three fully connected layers are trained. The first two have 4096 channels each, which results in 4096×4096 parameters that need to be trained (which is much more than the number of parameters in the convolutional layers).

The non-linearity used in all of the hidden layers was a rectified linear unit (ReLU). The last layer consists of a so called softmax function. The goal of this function is as follows. Prior to applying softmax, some vector components could be negative, or greater than one; and might not sum to 1; but after applying softmax, each component will be in the interval $(0, 1)$ and the components will add up to 1, so that they can be interpreted as probabilities. Furthermore, the larger input components will correspond to larger probabilities.

Lecture 4.2 Training of a Neural Network: Optimisation {sec:train_

When we have decided on which architecture we want to use (cf. the last chapter), we can start to train the network. We begin by initialising the weights randomly which will obviously not generate any meaningful results. The goal is now to (iteratively) find a set of weights that correspond to a better-performing network, or in other words weights that correspond to a lower loss. Most modern network use the (stochastic) gradient descent algorithm to minimise the loss function.

is associated where each component corresponds to one class that we want to predict and that contains a 1 in the corresponding entry and is zero everywhere else. The network will usually not be 100% certain about its decision and thus the final output of the layer will be a 1000 dimensional vector of which the entries sum to one and which has large values in the components that correspond to the predicted classes.

To find a minimum of a function starting from a random point on the graph, we can go in the direction of a negative gradient. This is the basic idea of gradient descent. At each step in learning we go one step in the direction of a negative gradient; of course theoretically the derivative only guarantees that the function will decrease if we take an infinitely small step which is obviously not possible in practice. The "size" of the step is a hyper-parameter called the learning rate which has to be adjusted beforehand.

In classic gradient descent, we would do this for every image in the training set which is usually very inefficient. Especially in the beginning, where the weights have random values, it is not necessary to incorporate every training image in computing the next (small) step towards a minimum. Therefore we only optimise over so called mini-batches of the dataset which are usually chosen randomly from the set of training images. The size of the mini-batch is another hyper-parameter that has to be chosen before training; usually batches are not larger than 100 (in some contexts only the extreme case of mini-batches consisting o a single image is called stochastic gradient descent; sometimes also mini-batch gradient descent is referred to as stochastic gradient descent).

Lecture 4.3 Convolutional Neural Networks for Semantic Segmentation

The goal of semantic segmentation is to understand what is in an image on a pixel level, i. e., to assign a class to each pixel in an image. This is visualised in the following image which is taken from the CityScapes dataset.

Another application of semantic segmentation is pose estimation. Here, the network is asked to find, for instance, all left hands in an image, or all right feet etc.

One very famous and very successful architecture of a CNN for semantic segmentation is the UNet given in Figure 4.3. One main difference when compared to the networks we previously

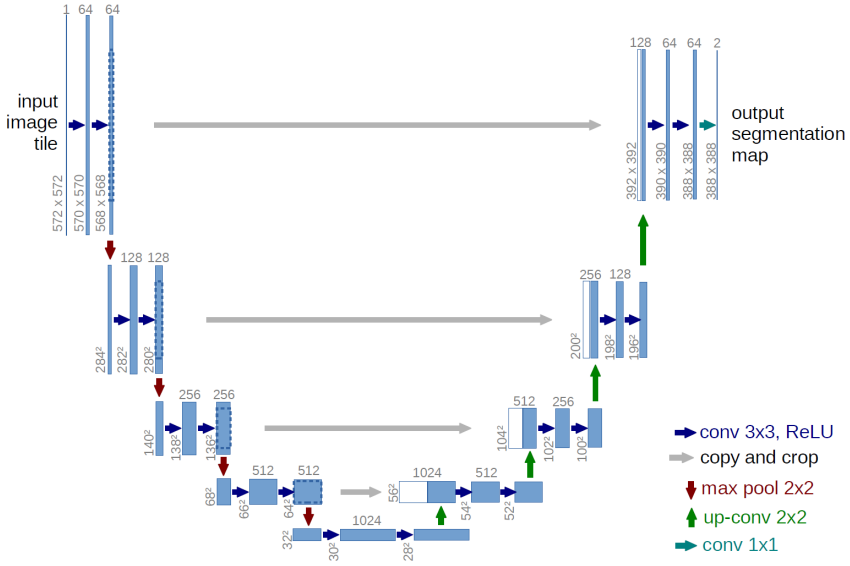


Figure 4.3: Architecture of the UNet {fig:unet}

discussed is that some layers or not only connected to the previous layer but also to other layers. More precisely, the output of some of the early layers is fed into the corresponding late layer of the same resolution. Further, instead of using max pool to reduce the dimensions of the image, smoothing and downsampling convolutional filters are used. They are not prespecified to a certain filter but are also learned. Similarly, upsampling convolution filters are used to increase the dimension again. The approach of the UNet can be interpreted as follows. The layers on the bottom in the Figure are very low-resolution but have many channels (up to 2014). The path through these layers is therefore referred to as the semantic pathway. Due to the high number of channels, its aim is to learn *what* is in each part of the image. However, the low resolution of $\sim 30 \times 30$ prohibits the network form learning exactly *where* these things are (for example in the CityScapes dataset, the network might be very certain that somewhere in the left side of the image there is a tram but it is unable to exactly pin down its location). This is where the skip connections (gray, from left to right) come into play. Although they obviously have not yet understood the difference between, e. g. , a tram and a pedestrian, they have learned where the boundaries of each of these objects are, as they typically act as edge detectors. This information from this so called geometric pathway is then combined with the information from the semantic pathway.

In practice, there are several techniques that should be used to improve the performance of the UNet. First, one should definitely use some form of normalisation, either group or batch norm (described below). Also, residual connections should be used in each of the individual blocks of the UNet. A simple schematic is given in Figure 4.4. In addition to the output from the previous layer, also the output of the layer before that is fed into Layer I. While one could use certain

linear transformations to transform the skip connection, often the two outputs are simply added to form the new input. This is sometimes referred to as a highway network. For more layers inside a block, additional connections are added. For instance, in a block of four layers, also the output of the second layer would be fed into the fourth. Furthermore, one could also add second-order connections, i. e., in the example of four layers, one could connect the first and fourth layer, skipping two layers instead of just one. In practice, these residual connection made it possible to train very deep network with 50–100 layers.

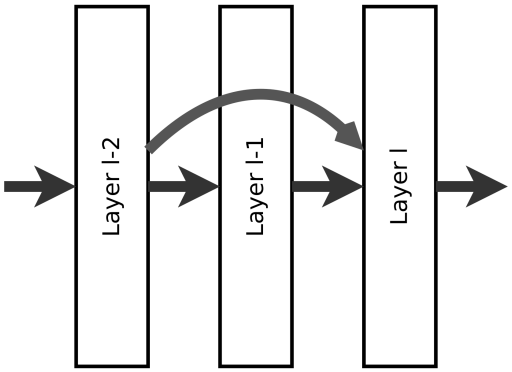


Figure 4.4: Residual block with an additional connection from the layer before the previous layer. {fig:resblo

Batch and Group Normalisation

Batch normalisation is a method that normalises activations in a network across the current mini-batch. For each feature, batch normalization computes the mean and variance of that feature in the mini-batch. It then subtracts the mean and divides the feature by its mini-batch standard deviation. After the normalisation, the activations have zero mean and unit standard deviation. In practice it was observed that batch normalisation has a huge impact on the effectiveness of the training of the network. Although there are many contributions discussing the actual effect of batch normalisation and explaining why it works so well, it is not fully understood.

With very large networks, the size of the mini-batches is often restricted by the available GPU memory and in the extreme case of a batch size of one, batch normalisation is obviously useless (also, for very small batch sizes batch normalisation does not work very well either). There is a variety of other normalisation techniques, on more recent one being group normalisation. Here, the mean and standard deviation are computed over groups of channels for each training example. In particular for cases where batch norm was not applicable, group norm has shown to produce very good results.

Week 5 Instance segmentation

{chap:05}

Week 6 Shortest Paths

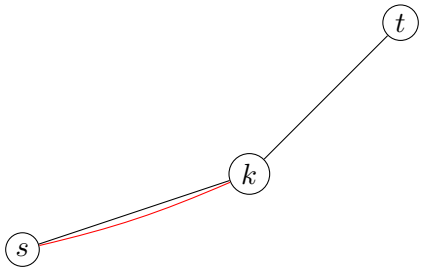
{chap:06}

Examples where shortest path algorithms are used in computer vision

- Intelligent scissors (GIMP)/ Magnetic lasso (Photoshop)
- Geodesic matting/ seeded segmentation
- segmented least squares (estimate piecewise constant best approximation, but a cost is imposed on jumps in the resulting function)

Basic approach to tackle shortest path problems: **dynamic programming**. Generally problems where dynamic programming techniques are applied are required to have the *optimal subproblem* property, which means that the optimal solution to a partial problem has to be part of the overall optimal solution.

Example (Shortest path)**.** Consider the following graph:



We do not know if the shortest path from s to t will go through k; if it does go through k, however, the the shortest path $s \to k$ must be part of the globally optimal shortest path $s \to t$ (red path).

It only makes sense to use dynamic programming if the solutions of the subproblems can be re-used multiple times. In methods where the paths are acyclic, the dynamic programming approach grows only linearly with the problem size which is the best possible complexity one can expect.

Lecture 6.2 Shortest path algorithms

{sec:shortp

In the following, the length of a path is defined to be the sum of the edge weights.

Dijkstra and Viterbi with example from lecture

Lecture 6.3 Scanline optimization/ Stereo disperity estimation

Setting: Two cameras, slighlty shifted or rotated and rectified, take two pictures of the same scene. We want to estimate the "depth" of the elements in the scene. As illustrated in the image

below, the two pictures are evaluated along a horizontal scanline (the example is taken from `http://lunokhod.org/?p=1356`).

Figure 6.1: Original rectified stereo images

For each pixel along that line, each possible disparity result is evaluated. The cost of each pixel along a scanline can then be stacked into a matrix; the cost for each pixel on the scanline in the example image versus each possible disparity value is shown in the Figure below. The solution for the disparity along this scanline is then the path through this matrix (image) that has minimum costs (dark areas) with some smoothness constraint imposed (i. e., , the shortest path). The process is repeated for every horizontal scanline; the resulting disparity map is given in Figure 6.3. Since each scanline is treated individually, the result might be not very smooth in the y-direction and one can apply different smoothing techniques to obtain a disparity map this is smooth in both the x- and y-direction.

Figure 6.2: All possible disparities along one scanline

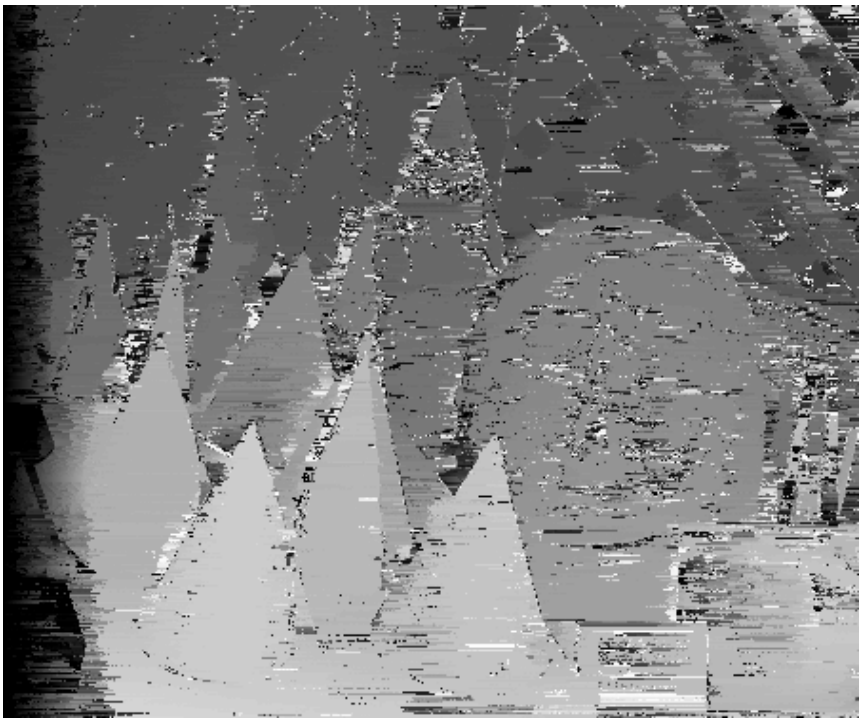


Figure 6.3: Disparity map for the example above {fig:disper

MAP – maximum a posteriori estimation

The problem above can be rewritten as: Find

$$\begin{aligned}\min_{z_1,\dots,z_n} e(z) &= \min_{z_1,\dots,z_n} \sum_{i=1}^{n-1} \psi_{i,i+1}(z_i, z_{i+1}) \\ &= \min_{z_1,\dots,z_n} \psi_{n-1,n}(z_{n-1}, z_n) + \cdots + \psi_{2,3}(z_2, z_3) + \psi_{0,1}(z_0, z_1) \\ &= \min_{z_n} \left(\min_{z_{n-1}} \psi_{n-1,n}(z_{n-1}, z_n) + \cdots + \min_{z_2} \Big(\psi_{2,3}(z_2, z_3) + \underbrace{\min_{z_1} \psi_{1,2}(z_1, z_2)}_{m_{1\to 2}(z_2)} \Big) \cdots \right),\end{aligned}$$

where the vectors z_i denote the "state" at time i (i. e., , the ith column of the graph). Note that the components of these vectors are either one or zero, and they have to sum up to one. In other words, these state vectors denote in the end which of the respective nodes of the graph lie in the shortest path (see also the Figure below).

Lecture 6.4 Segmented least squares

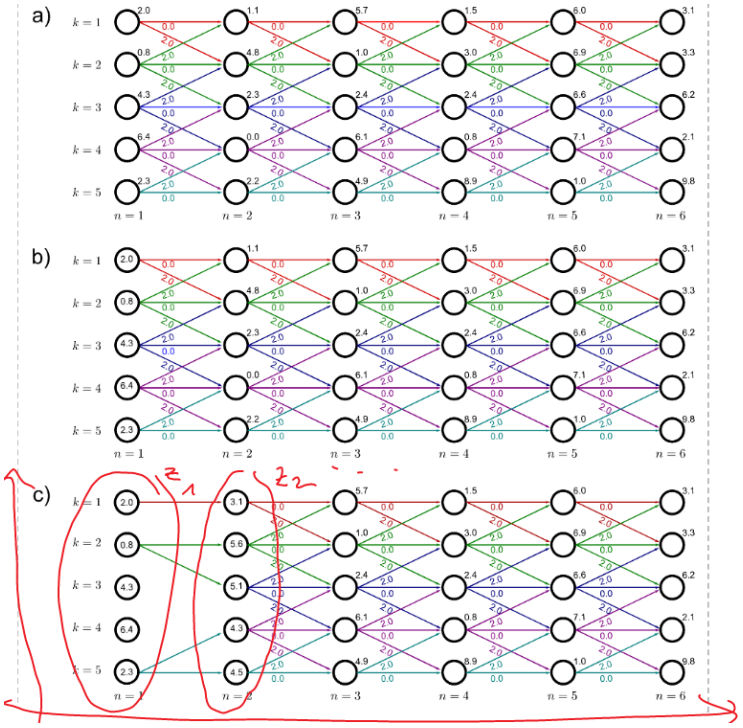


Figure 6.4: State vectors

Week 7 Message passing

{chap:07}

Lecture 7.1 Why dynamic programming on trees?

In many applications in computer vision we want to model connectivity. Consider the segmentation of blood vessels in a lung. These vessels form a tree and thus it is natural to pose a connectivity constraint in a segmentation task. For example, this leads to the connections of previously unconnected regions which in turn reduces the noise in the computed segmentation.

Trees are also a very natural choice for modeling hierachies. For instance, in cell segmentation, one can represent the segmented cells in a tree structure where children correspond to the individual parts of a "clumped" cell, see also the Figure below.

We can also model (simple) constellations; applications are facial feature detection in images. Here, the parts of a face (eyes, nose, mouth etc.) are modeled in a tree structure where the edges can be interpredted as springs in a star configuration to the nose, see the Figure below. Extending these springs or posing them in certain positions that do not make sense has a high cost, and thus, plausible configurations that correspond to realistic facial features correspond to low costs.

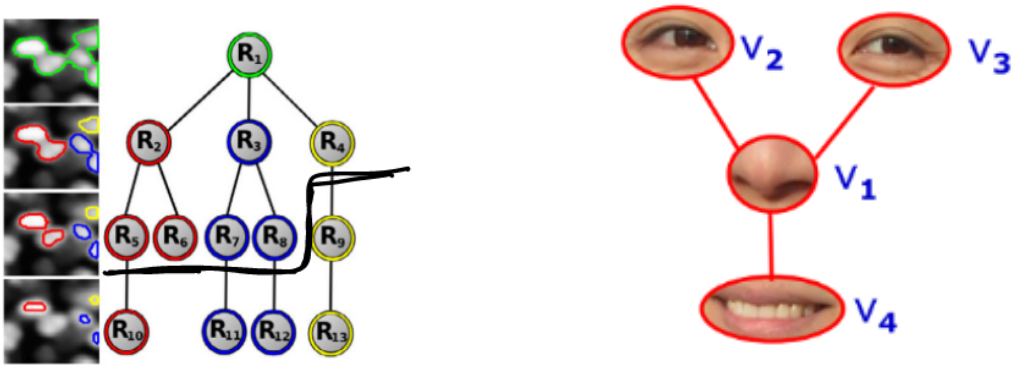


Figure 7.1: Two example of how trees can be used to model hierachies and simple constellations in computer vision applications.

Week 8 Widest paths

{chap:08}

Lecture 8.2 Shortest vs. widest paths

In the "classical" shortest path algorithms discussed so far the length of a path was given by the **sum** of the edge weights. In this chapter we discuss widest path (bottleneck shortest path/ maximum capacity path/ minimax path) methods. Here, the length of a path is given by the **single largest** (smallest) edge weight.

1D example from lecture

Lecture 8.3 Minimax paths and Prim's algorithm

Recall Disjkstra's Algorithm from a previous lecture. At each iteration we do the following: take the node v from the list of nodes that have not yet been visited with the smallest distance from the starting node. Denote by $e = (v, u)$ the edge from v to u. In the "classical" Dijkstra algorithm (where we want to compute the shortest path from the starting vertex to each of the other nodes) we now do the following: For each edge $e(v, u)$, u in the forward-star (i. e., the set of edges connecting "outgoing" nodes) of v, update the distance to the node u according to

$$d(u) = \min\{d(u), d(v) + w(e)\},$$

where $w(e)$ is the edge weight of e. If we want to compute the minimax path, we can still use the same idea; however, we instead update $d(u)$ by

$$d(u) = \min\{d(u), \max\{d(v), w(e)\}\},$$

since we are only interested in the single most expensive edge on the way from the starting node to u. This change essentially leads to a related algorithm, called (eager) Prim's algorithm. Instead of finding the shortest path we now build a spanning tree, i. e., a minimal tree that contains every node of the graph. We see that the shortest path problem and the minimax problem are in essence the same problem only with different definition of what we mean by the distance between two nodes. In other words: both a shortest path problems with a different notion of what it means to be short. Algebraic graph theory, which is discussed in the next chapter, provides a unified framework to better understand this relationship.

Lecture 8.4 All-pairs minimax paths and the minimum spanning tree

A minimum spanning tree (MST) is a subgraph with the shape of a tree (i. e., it contains no loops) that is spanning (i. e., each node can be reached from any other in the subgraph) whose sum of edge weights is minimal. Every edge not in the MST is at least as large/ heavy/ costly as all other edges in the loop induced by adding that edge to a MST ("cycle property"), see Figure 8.1.

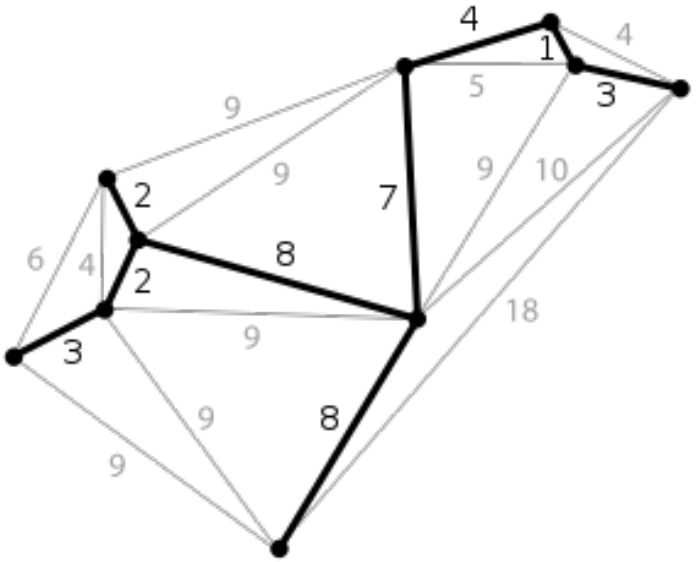


Figure 8.1: A minimum spanning tree. {fig:mst}

It turns out that the path from some node u to some other node v in the MST of an undirected graph is a minimax path between u and v in that graph. The converse is not true in general: the union of the minimax paths between all nodes is not necessarily a tree and hence in particular it need not be the MST.

Further, in an undirected graph with non-negative edge weights, the minimax distance between pairs of vertices defines an **ultrametric**.

Definition. An ultrametric d_{uv} is a metric with the following additional property

$$d_{uv} \leq \max\{d_{uw}, d_{wv}\}\,,$$

the so called *strong triangle inequality* (or ultrametric inequality).

Note that the ultrametric inequality implies that there exists a permutation (i, j, k) of three vertices (u, v, w) such that

$$d_{ij} \leq d_{ik} = d_{kj}\,.$$

This, in trurn, implies that a set of points in an ultrametric space can always be represented as leaves in a binary rooted tree that all have the same distance from the root (a so called *ultramteric tree*). Consider again the MST from Figure 8.1. We can build the corresponding ultrametric tree as follows. Start with the nodes that have the smallest distance, draw them as leaves of a tree and connect them to the same parent that is located at the "height" that corresponds to their distance (see Figure below; we start with the two nodes on the top right that are connected by an edge with weight one). Then we take the node that is next closest and connect the subtree from the previous step to this node (in the Figure 8.2, now all vertices within the green ellipse are processed). We then repeat this process until no more nodes are left. The ultrametric distance between two nodes is then given by the height of their least common ancestor in the ultrametric tree. Note that there exist data structures such that finding the least common ancestor of two nodes is an $\mathcal{O}(1)$ operation.

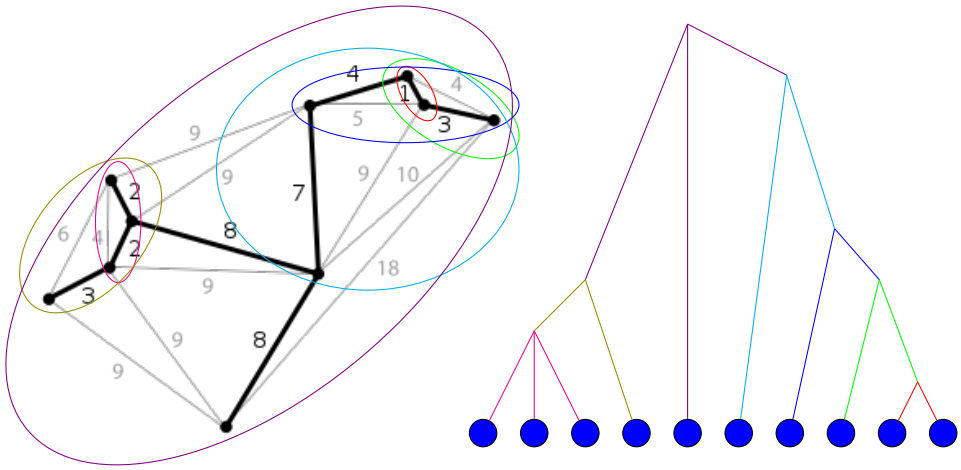


Figure 8.2: Minimum spanning tree and corresponding ultrametric tree. The ultrametric tree is built by repeatedly finding the nodes that have the smallest distance to the nodes already processed (indicated by the ellipses). {fig:ultram

Lecture 8.5 Seeded watershed segmenation

Finish lecture

Lecture 8.6 Learned watershed

Finish lecture

Week 9 Algebraic graph theory

{chap:09}

Summary In this lecture we discuss a unified framework that allows to us to understand many different path problems (shortest paths, widest paths, existence of paths etc.) in a single context. This will be done by generalising the notion of matrix multiplications which will allow us to write shortest path, widest path and path existence problems in terms of matrix powers.

Lecture 9.1 Generic single source shortest paths

Recall again Dijkstra's algorithm. In the inner loop we look at each edge $e = (v, u)$ in the forward star of the current looked-at node v. We discussed two different types of Dijkstra's algorithm that only differ in the way the distance $d(u)$ from the starting vertex is updated (sum of all edge weights in shortest path; single most expensive edge weight in minimax path). We now discuss a third variant. Here, we want to answer the question whether or not there is a path at all. However, this is a special case of the above discussed methods where we encode the edges of the original path as 0 if there is a connection between the nodes and ∞ if there is no edge between two vertices. Alternatively, we can formulate this problem as follows. Consider a fully connected graph with edge weights $\{\texttt{FALSE}, \texttt{TRUE}\}$. In the initialisation phase of Dijkstra's algorithm, we set

$$d(s) = \texttt{TRUE} \quad \text{and} \quad d(u \in V \setminus \{s\}) = \texttt{FALSE}\,.$$

The update of the "distance" $d(u)$ then looks as follows

$$d(u) = \texttt{OR}(d(u),\, \texttt{AND}(d(v), w(e)))\,.$$

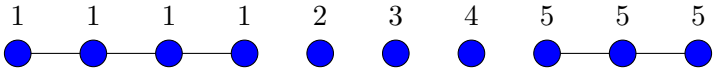
In other words: along a path, each single edge weight has to be `TRUE` (this is the inner `AND`) while of many different paths only one has to be `TRUE` (this is the outer `OR`) for $d(u)$ to be set to `TRUE`. Example applications for this problem are

- In computer vision this is often used when training a network that is supposed to detect some feature and one wants to collect all pixels that are connected (and lie above a certain threshold). This is usually not done using the approach introduced above but it the problem itself can still be solved by an algorithm that falls in this general framework discussed in the following.
- Other problems include: reachability, percolation (simulate if water will reach the bottom when flowing through a certain network), connectedness or maze solving.

We see that all the different kinds of Dijkstra's algorithm are only certain special cases of a generic single-source generalized-shortest-distance algorithm that can find shortest and widest path as well as determine whether or not there is a path at all etc. Note, however, that these algorithms need not be the most efficient ones solving the particular problem (e. g. for the problem of finding if there exists a path one would implement the algorithm using a disjoint-set/ union-find data structure where each node is mapped to an index such that nodes with same index are connected).

In 1D there is simple $\mathcal{O}(N)$ algorithm to determine whether or not two nodes are connected. Consider the following simple graph.

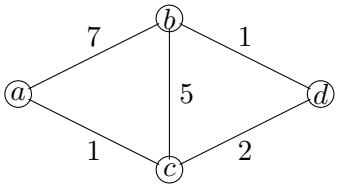
We now start from the left and assign an arbitrary label to the first node. If the next node is connected to the current node then we assign to it the same label. Otherwise we choose a different label. Using numbers as labels, our graph from above might then look like this. Checking if two



nodes are connected is now simply achieved by checking whether or not their assigned node labels are the same. Assigning the node labels is obviously in $\mathcal{O}(N)$, where N is the number of nodes and checking two nodes is in $\mathcal{O}(1)$.

Lecture 9.2 All-pairs shortest paths and the distance product

Consider the following graph and its associated adjency matrix. where the entries (a, d) and (d, a)



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 7 & 1 & \infty \\ 7 & 0 & 5 & 1 \\ 1 & 5 & 0 & 2 \\ \infty & 1 & 2 & 0 \end{bmatrix} \end{array}$$

are set to ∞ since there is no connection between these two vertices. Now, recall that usual matrix multiplication formula. If we multiply two matrices $C = AB$, we have

$$[C]_{ij} = \sum_k [A]_{ik} \cdot [B]_{kj} \,.$$

We will now define different kinds of "matrix multiplications" where we replace at least one of the sum (green) and product (red). We start by replacing the sum by taking a minimum and

replacing the product by a sum. The resulting operation looks as follows.

$$[C]_{ij} = \min_k [A]_{ik} + [B]_{kj} \,.$$

If we compute the "matrix product" $A \cdot A$ for the adjancy matrix from the example above using this formula we obtain

$$A^2 = \begin{bmatrix} 0 & 7 & 1 & \infty \\ 7 & 0 & 5 & 1 \\ 1 & 5 & 0 & 2 \\ \infty & 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} 0 & 7 & 1 & \infty \\ 7 & 0 & 5 & 1 \\ 1 & 5 & 0 & 2 \\ \infty & 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & 3 & 1 \\ 1 & 3 & 0 & 2 \\ 3 & 1 & 2 & 0 \end{bmatrix} .$$

The matrix A obviously contains the cost to go from one node to another with at most one hop. The matrix A^2, computed using the min sum matrix product now contains the cost of the shortest path from all nodes to all other nodes with at most two hops! The entries with the yellow background in the matrix above correspond to paths where in two hops we found a shorter path as compared to using just one hop.[1]

If we now multiply this result again by A, i. e., if we compute $A \cdot A^2$, we obtain the cheapest cost from all nodes to all other nodes while taking at most three hops. The result is as follows

$$AA^2 = \begin{bmatrix} 0 & 7 & 1 & \infty \\ 7 & 0 & 5 & 1 \\ 1 & 5 & 0 & 2 \\ \infty & 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} 0 & 6 & 1 & 3 \\ 6 & 0 & 3 & 1 \\ 1 & 3 & 0 & 2 \\ 3 & 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 4 & 1 & 3 \\ 4 & 0 & 3 & 1 \\ 1 & 3 & 0 & 2 \\ 3 & 1 & 2 & 0 \end{bmatrix} .$$

We again highlighted the costs that have changed. If repeat this process another time, the matrix no longer changes since in a graph with four nodes, we can only do three hops before arriving at nodes that have been visited on the path. Depending on the graph, this "convergence" might happen earlier.

We can use this iterated generalised matrix multiplication to also solve minimax path and path-existence problems that we have discussed earlier. For the minimax path we define the matrix product as

$$[C]_{ij} = \min_k \max \left\{ [A]_{ik} [B]_{kj} \right\} .$$

[1] This is in essence a matrix formulation of the Floyd-Warshal algorithm.

Again using the example from the beginning of the section, after two hops we have

$$A^2 = \begin{bmatrix} 0 & 7 & 1 & \infty \\ 7 & 0 & 5 & 1 \\ 1 & 5 & 0 & 2 \\ \infty & 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} 0 & 7 & 1 & \infty \\ 7 & 0 & 5 & 1 \\ 1 & 5 & 0 & 2 \\ \infty & 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 5 & 1 & 2 \\ 5 & 0 & 2 & 1 \\ 1 & 2 & 0 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix},$$

and after three hops

$$AA^2 = \begin{bmatrix} 0 & 7 & 1 & \infty \\ 7 & 0 & 5 & 1 \\ 1 & 5 & 0 & 2 \\ \infty & 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} 0 & 5 & 1 & 2 \\ 5 & 0 & 2 & 1 \\ 1 & 2 & 0 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}, = \begin{bmatrix} 0 & 2 & 1 & 2 \\ 2 & 0 & 1 & 2 \\ 1 & 2 & 0 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}.$$

Again, the result after four hops is the same as after three. Lastly, for the path existence problem we let

$$[C]_{ij} = \underset{k}{\texttt{OR}} \left\{ [A]_{ik} \ \texttt{AND} \ [B]_{kj} \right\}.$$

As mentioned above, the edge weights are now either `T` = `TRUE` or `F` = `FALSE` depending on whether or not an edge connects the two nodes in consideration. Thus the matrix after one hop is given by

$$A = \begin{bmatrix} \texttt{T} & \texttt{T} & \texttt{T} & \texttt{F} \\ \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \\ \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \\ \texttt{F} & \texttt{T} & \texttt{T} & \texttt{T} \end{bmatrix}.$$

The result after two hops is then given by

$$A^2 = \begin{bmatrix} \texttt{T} & \texttt{T} & \texttt{T} & \texttt{F} \\ \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \\ \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \\ \texttt{F} & \texttt{T} & \texttt{T} & \texttt{T} \end{bmatrix} \begin{bmatrix} \texttt{T} & \texttt{T} & \texttt{T} & \texttt{F} \\ \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \\ \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \\ \texttt{F} & \texttt{T} & \texttt{T} & \texttt{T} \end{bmatrix} = \begin{bmatrix} \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \\ \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \\ \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \\ \texttt{T} & \texttt{T} & \texttt{T} & \texttt{T} \end{bmatrix},$$

after which the result obviously does not change anymore. Note that we can speed the method up by using the fact that after we have computed $A^2 = A \cdot A$ we can use this to compute $A^4 = A^2 \cdot A^2$ etc. In practice one would also use optimized matrix multiplication algorithms and/ or use approximations of the matrices (e. g. one might only use randomly selected rows and columns).

Note that we did consider all-pairs path problems. In practice one does often not have enough memory to store this matrix and thus, instead of applying these algorithms to the original image, they are applied to patches or superpixels.

Lecture 9.3 The algebraic path problem

We have just seen that single source (generalised Bellmann-Ford) and all pairs shortest paths (generalised Floyd-Warshall, distance matrix product) algorithms can be seen as special cases of one basic algorithm. More precisely, they can be seen as multiple incarnations of matrix multiplications on different *commutative semirings*. We start with a short recap on monoids.

Definition. A *monoid* $(\mathbb{K}, \cdot, \overline{e})$ is a set $\mathbb{K}$ with a binary operation $\cdot : \mathbb{K} \times \mathbb{K} \to \mathbb{K}$ such that for all $a, b, c \in \mathbb{K}$, the equation

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

holds and an identity element $\overline{e}$ which satisfies

$$a \cdot \overline{e} = \overline{e} \cdot a\,, \quad \text{for all } a \in \mathbb{K}\,.$$

If additionally we have $a \cdot b = b \cdot a$ for all $a, b \in \mathbb{K}$ we call the monoid *commutative*.

We can now define a semiring.

Definition. A semiring is a system $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ such that

1. $(\mathbb{K}, \oplus, \overline{0})$ is a commutative monoid,
2. $(\mathbb{K}, \otimes, \overline{1})$ is a monoid (not necessarily commutative),
3. $\otimes$ distributes over $\oplus$, i. e., for all $a, b, c \in \mathbb{K}$ we have

$$\begin{aligned}(a \oplus b) \otimes c &= (a \otimes c) \oplus (b \otimes c)\\ c \otimes (a \oplus b) &= (c \otimes a) \oplus (c \otimes b)\,,\end{aligned}$$

   and
4. $\overline{0}$ is an annihilator for $\otimes$, i. e., $a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$ for all $a \in \mathbb{K}$.

The semiring is said to be commutative is $\otimes$ is commutative.

Given a semiring, one can define the associated matrix semiring $M_n(\mathbb{K})$ of $n \times n$ matrices with entries in $\mathbb{K}$. Note that commutativity does not necessarily transfer from $\mathbb{K}$ to $M_n(\mathbb{K})$. This is what we essentially used in the previous section where we discussed different matrix semirings such as the minmax semiring.

We now consider the algebraic path problem. We begin with the following equation that defines the entries of a distance matrix $[D]_{st}$ from a source s to a target t as

$$[D]_{st} = \bigoplus_{P \in \mathcal{P}_{st}} \bigotimes_{e \in P} w_e\,,$$

where $\mathcal{P}_{st}$ is the set of all paths P from s to t and $\otimes$ is taken over all edges within P. Here $\otimes$ is the semiring multiplication and is applied along the path whereas the semiring addition $\oplus$ is done between paths. For instance, when we solve the shortest path problem, we can interpret this as solving the algebraic path problem in the $(\min, \text{sum})$ semiring.

Lecture 9.4 Infimal convolution, morphology and the Euclidean distance transform

We discussed earlier that convolution can be written as a matrix multiplication with a Toeplitz matrix. We can now define a generalised notion of convolution similar to how we generalised matrix multiplication in the previous section. There, we defined matrix multiplication over a semiring as

$$[AB]_{ij} = \bigoplus_k [A]_{ik} \otimes [B]_{kj} \,.$$

Analogously, one can define a generalised inner product

$$\langle a, b \rangle = \bigoplus_k a_k \otimes b_k \,.$$

With the same approach, we can define convolution on different semirings, e. g. on the sum-product semiring

$$f \star g(x) = \sum_y f(x-y) \cdot g(y) \,,$$

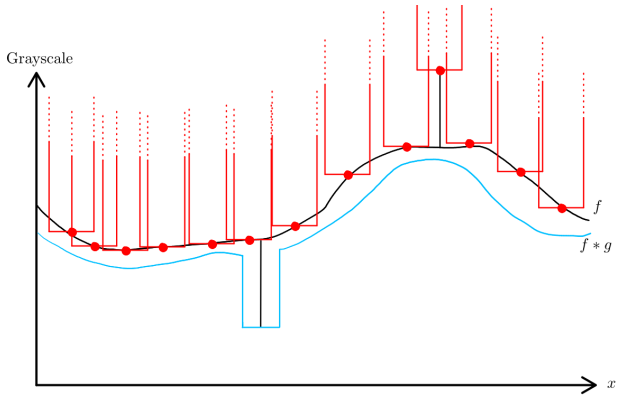
which results in the "standard" convolution that we have seen earlier or on the min-sum semiring (also referred to as the *tropical semiring*)

$$f \star g(x) = \inf_y \left(f(x-y) + g(y) \right) \,,$$

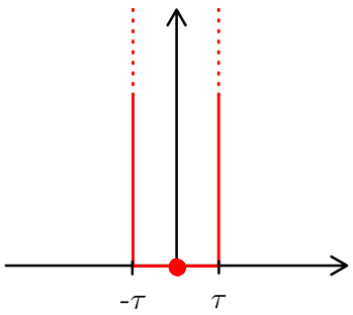
which results in the important generalisation of the *infimal* or *tropical convolution*. In the context of mathematical morphology, this operation is often referred to as *erosion*[2]. The function g is then usually called a *structuring element* (SE). In the figure below we plotted one row of a grayscale image together with the result (light blue line) of the erosion using the strucutral element given on the right. As we can see, the erosion computes a lower envelope of the signal. We also observe that positive peaks shrink; more precisely, peaks that are thinner than the structuring element disappear. (in image analysis, we can for example use this to pre-process images to get rid of bright speckles). Valleys and sinks, on the other hand, are expanded. If we would apply dilation to the same signal (with the structuring element being constant in the interval $[-\tau, \tau]$ and zero everywhere else), we would observe the opposite effects.

Binary erosion example from lecture? Minute 12

[2] Analogously, one can define the max-sum convolution which is usually called *dilation*



(a) Erosion applied to one row of a grayscale image.



(b) Structural element that was used in the example.

{fig:tropco

Figure 9.1: Example of infimal convolution.

One can combine erosion and dilation to obtain

$$\text{opening} \;\coloneqq\; \text{dilation}(\text{erosion}(\text{image}, \text{SE}), \text{SE})$$

and

$$\text{closing} \;\coloneqq\; \text{erosion}(\text{dilation}(\text{image}, \text{SE}), \text{SE})\,,$$

where SE stands for structuring element. Since erosion shrinks all objects which leads to the elimination of small objects and dilation enlarges objects, the opening operation can be used to get rid of very small objects in images. Closing, on the other hand, gets rid of small holes in objets, since these holes get first “filled” by the dilation operation after which erosion than shrinks the objects to their original size, but keeping the holes “filled”.

Euclidean distance transform of a binary set

Week 10 Tracking

{chap:10}

Appendices

Week A Machine Learning primer

This overview is based on the chapter `https://www.deeplearningbook.org/contents/ml.html`.

Lecture A.1 Learning Algorithms

What does the *learning* mean in machine learning? Popular definition is due to Mitchell:

> A Computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

In the following, we give intuitive descriptions and examples of what T, P and E might be in practice.

The Task T

Task is not the process of learning itself. Learning is our means of attaining the ability to perform the task (Example: If we want the robot to walk, walking is the task). Usually, tasks are described in terms of how the machine should process an **example**. An example conists of **features** that have been quantitatively measurred from something that we want the machine learning system to process. Typically, we represent an example as $\mathbf{x} \in \mathbb{R}^n$ where each x_i is another feature (features of images $\hat{=}$ values of the pixels).

Some of the most common ML tasks are

- **Classification.** Here, the system is asked to put the input into one of k categories. Typically, this is modelled as a function $f : \mathbb{R}^n \to \{1, \dots, k\}$. Other variants are possible, where f outputs a probability distribution over the different classes. A popular example is object recognition.
- **Regression.** Predict a numerical value given some input, i. e., output a function $f : \mathbb{R}^n \to \mathbb{R}$.
- **Transcription.** Here, the system is asked to observe a relatively unstructred representation of some data and transcribe into textual form (Example: transform photograph of text into sequence of ASCII characters; Speech recognition).
- **Machine translation.** Convert sequence of symbols from one language into sequence of characters of another language.

- **Structured output.** Broad category, involving any task that outputs a vector with important relationships between different elements. Examples from above also fall in this category; other examples are formation of sentences that describe what is shown in an image (here the words in the output have the relationship that they must form a correct sentence).
- **Anomaly detection.** Flag some events or objects from some set as being unusual (example: credit card fraud detection).

Add the missing tasks (not that important?)

These tasks are only examples; many other types of tasks are possible.

The Performance Measure P

To quantitatively measure the learning process we need to be able to evaluate the algorithm's performance. To that end, a performance measure is required that is usually specific to the task. For tasks as classification we often measure the **accuracy** of the model (= the proportion of examples for which the model produces the correct output). Alternatively, we can measure the proportion of examples that result in incorrect output (**error rate**). We often refer to the error rate as the expected 0-1 loss (0 if an example is correctly classified, 1 if it is not). For continuous examples we need a continuous performance metric (e. g. , the average log-probability the model assigns to some examples). It is often difficult to define a performance measure that corresponds well to the desired behavior of the system (What should be measured? What if measuring is impractical or even intractable? How can alternative criteria be designed?).

We are usually interested in the perfomance of the ML algorithm on data it has never seen before; thus the performance is measured on a **test set**. This set is different from the training data.

The Experience E

ML methods can be broadly categorized as **unsupervised** and **supervised**. They differ in the kind of experience they are allowed to have during the training. Most of the algorithms we consider have access to an entire dataset, i. e., a collection of many examples which themselves consist of features. Sometimes these examples are called data points. Examples for supervised learning algorithms have additional **labels** or **targets**. Note, that unsupervised learning and supervised learning are not formally defined and the lines between them are often blurred.

A common way of describing a dataset is with a **design matrix** that contains in each of its rows a different example. The columns correspond to the features. Of course, this implicitly requires each example to be a vector of the same size which is not always possible (e. g. , photographs with different widths and heights). In these cases, design matrices are not the right choice of representation. In the case of supervised learning, where each example contains a label or target,

we work with a design matrix of observations $\boldsymbol{X}$ and a vector of labels $\mathbf{y}$, with y_i providing the label for example i (Of course, the label can be more than just a single number).

Week B Convolution primer

{chap:appen

This overview is mainly based on the paper `https://arxiv.org/pdf/1603.07285.pdf` and the chapter `https://www.deeplearningbook.org/contents/convnets.html`.

Lecture B.1 Intro

Generally, a discrete convolution is a linear transformation that preserves ordering in the signal (such as width and height axis in images, time axis in sound clips). The figure below shows an example of a discrete convolution. The light blue grid is called the *input feature map* (multiple feature maps are possible, e. g. when convoluting the different color channels of an image). The shaded are is referred to as the *kernel*. The results (in green) are the *output feature maps*.

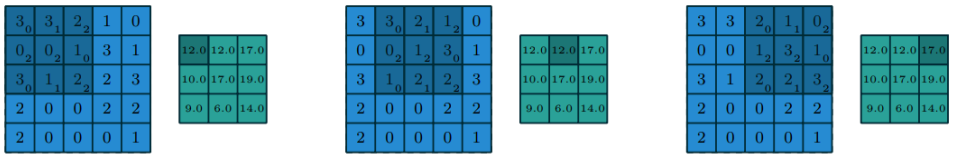


Figure B.1: Discrete convolution

{fig:conv:e

The figure is an instance of a 2-D convolution but it can be generalised to N-D convolutions. The output size o_j of a convolutional layer along an axis j is affected by

- i_j: input size along the axis
- k_j: kernel size along the axis
- s_j: stride along the axis (distance between to consecutive positions of the kernel)
- p_j: zero padding along the axis

For instance, Figure B.2 shows a 3×3 kernel applied to a 5×5 input padded with a 1×1 border of zeros using 2×2 strides. Note that strides can be interpreted also as follows. Moving the kernel

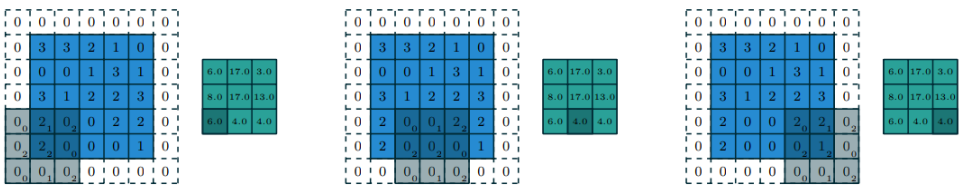


Figure B.2: Different discrete convolution

{fig:conv:e

by, e. g. , hops of two is the same thing as moving it by hops of ibe but retaining only odd output elements.

Lecture B.2 The Convolution Operation

In the most general form, convolution is an operation of two functions of a real-valued argument. Take this example: Suppose we track a spaceship using a laser sensor that provides a single output $x(t)$, the position of the spaceship at time t, where both x and t are real-valued. Now suppose this sensor is somewhat noisy. To obtain less noisy estimates of the position, we want to average over severale measurements. We would like to have a method that weighs more recent measurements more heavy. To that end, we define a weighting function $w(a)$, where a is the age of the measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship

$$s(t) = \int x(a)w(t-a)\,\mathrm{d}a\,.$$

This operation is called **convolution** and is typically denoted with an asterisk, i. e., $s(t) = (x * w)(t)$.

In a machine learning context, the first argument (x, in this example) is usually called the **input**, and the second argument (w) is referred to as the **kernel**. The output is called the **(output) feature map**.

In most of our examples, we will encounter discrete time steps and thus deal with *discrete convolution*

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)\,.$$

The inputs and kernels are tensors that are represented as multi-dimensional arrays (e. g. images $\hat{=}$ matrices). We assume that only for a finite set of points are these functions not zero because otherwise they could not be implemented on a computer with finite storage. The summations then reduce to summation over a finite number of array elements.

Often, we use convolutions over more than one axis at a time. For example, if the input is an image I the kernel K will probably also be 2D

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)\,.$$

Since convolution is commutative, we can equivalently write

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)\,,$$

which is often more straightforward to implement. Convolution is commutative because the kernel is flipped (i. e., as m increases the index into the input increases whereas the index into the kernel decreases). Since the commutative property is often not an important property in a neural network implementation, many libraries implement a related function called the **cross-correlation** and often call this also convolution

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)\,.$$

Discrete convolution can be viewed as matrix multiplication. For example, univariate discrete convolutions are equivalent to multiplication by a **Toeplitz** matrix (each row is a shifted version of the row above). In 2D, convoltion corresponds to a **doubly block circulant matrix**. In most cases, these matrices are sparse.In the most general form, convolution is an operation of two functions of a real-valued argument. Take this example: Suppose we track a spaceship using a laser sensor that provides a single output $x(t)$, the position of the spaceship at time t, where both x and t are real-valued. Now suppose this sensor is somewhat noisy. To obtain less noisy estimates of the position, we want to average over severale measurements. We would like to have a method that weighs more recent measurements more heavy. To that end, we define a weighting function $w(a)$, where a is the age of the measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship

$$s(t) = \int x(a) w(t-a)\,\mathrm{d}a\,.$$

This operation is called **convolution** and is typically denoted with an asterisk, i. e., $s(t) = (x * w)(t)$.

In a machine learning context, the first argument (x, in this example) is usually called the **input**, and the second argument (w) is referred to as the **kernel**. The output is called the **(output) feature map**.

In most of our examples, we will encounter discrete time steps and thus deal with *discrete convolution*

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a) w(t-a)\,.$$

The inputs and kernels are tensors that are represented as multi-dimensional arrays (e. g. images $\hat{=}$ matrices). We assume that only for a finite set of points are these functions not zero because otherwise they could not be implemented on a computer with finite storage. The summations then reduce to summation over a finite number of array elements.

Often, we use convolutions over more than one axis at a time. For example, if the input is an

image I the kernel K will probably also be 2D

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)\,.$$

Since convolution is commutative, we can equivalently write

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)\,,$$

which is often more straightforward to implement. Convolution is commutative because the kernel is flipped (i. e., as m increases the index into the input increases whereas the index into the kernel decreases). Since the commutative property is often not an important property in a neural network implementation, many libraries implement a related function called the **cross-correlation** and often call this also convolution

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)\,.$$

Discrete convolution can be viewed as matrix multiplication. For example, univariate discrete convolutions are equivalent to multiplication by a **Toeplitz** matrix (each row is a shifted version of the row above). In 2D, convoltion corresponds to a **doubly block circulant matrix**