# Computer Vision: Foundations

*Lecture Notes*

Lecturer: Prof. Dr. Fred Hamprecht

Edited by: Nils Friess

Last updated: July 30, 2020

# Contents

# 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 endeavour and only two hand full 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 full 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

## Lecture 1.1 Linear Filters: Convolution

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

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

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).

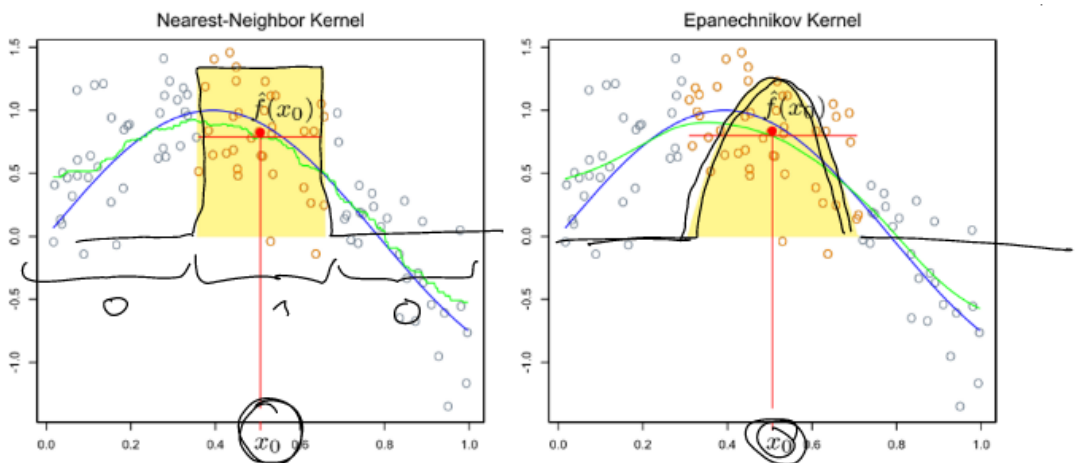



Figure 1.1: Box kernel and Epanechnikov kernel

Different filters can produce very different results ($\rightsquigarrow$ filter optimisation). 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 **separable** 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 $b$)[1]. 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.

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

# Week 2 Sensing, Subsampling and Interpolation

**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 2.1 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* approach 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 applicable 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

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

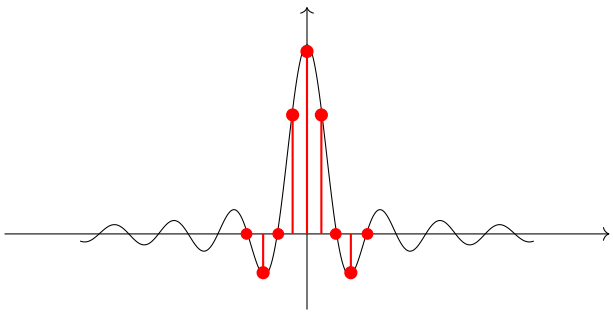


Figure 2.1: The sinc function and the values we would use in a discrete filter.

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 2.2 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

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

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 $\Pi$, 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 $\Pi^3$ or $\Pi^4$ resemble a Gaussian kernel with the exception that they both have finite support (the box filter has a support of width 1, $\Pi^2$ has a support of width 2 etc.).

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

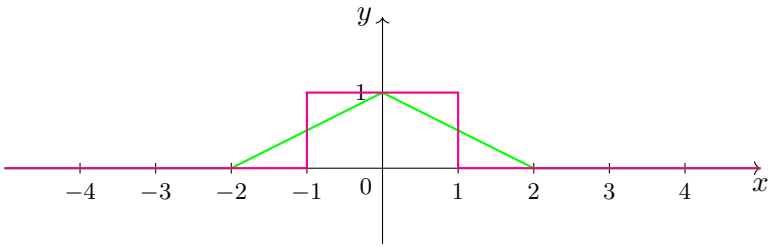



Figure 2.2: Box filter and tent filter.

of (non-interpolating) B-splines. Note that $\Pi^n$ 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

**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:

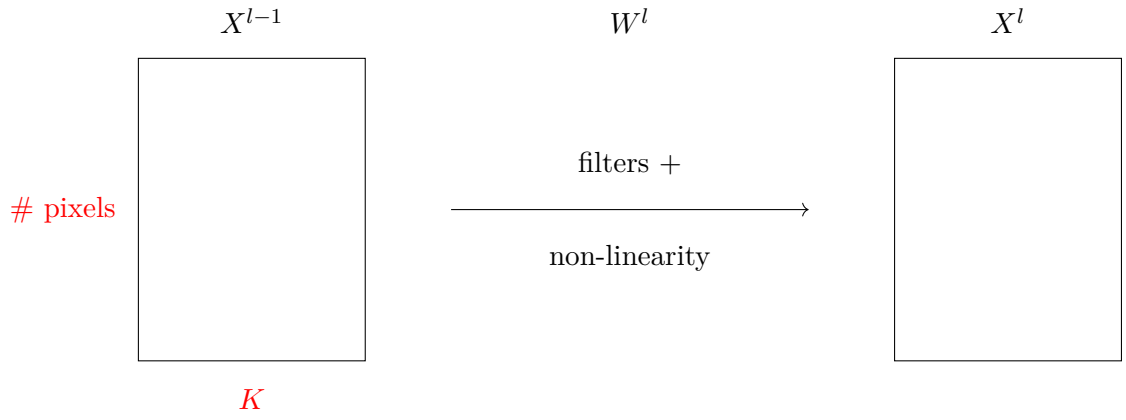
1. Take image
2. Compute features
3. Construct decision rule **by hand** (e. g. select foreground and background)
4. Mark pixels according to decision rule

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 $\sigma$ 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.

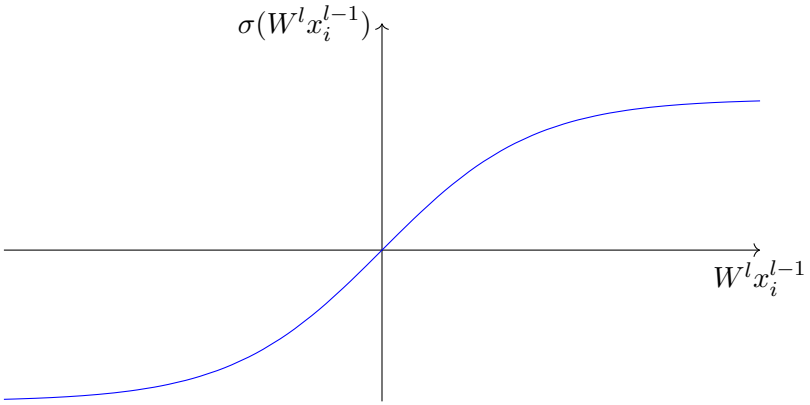
2. A reasonable simplifying assumption would be to make the decisions rules local, i. e., only pixels that are spatially close to the target contribute to the result. $W^l$ would then have band structure.

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.

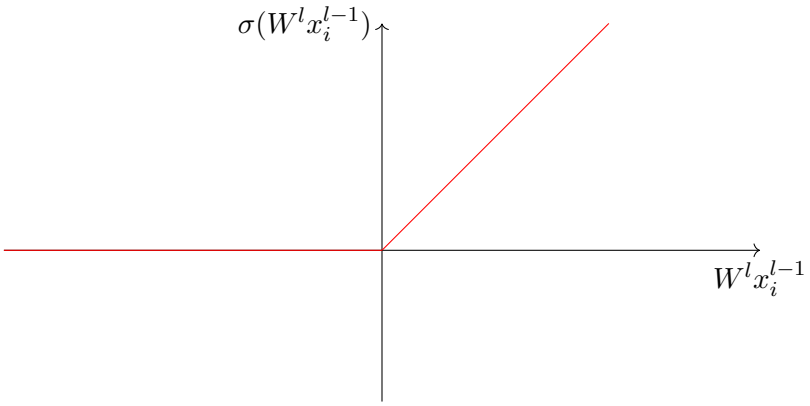
[1] Similarly, although the sinc theoretically is the perfect low pass filter, it is often not plausible why a pixel that is more than a few hundred pixels away of the current pixel should have an impact on this very pixel (which would be the case with infinitely-supported filters such as the sinc filter).

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

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 $i$th 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 or 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

**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.

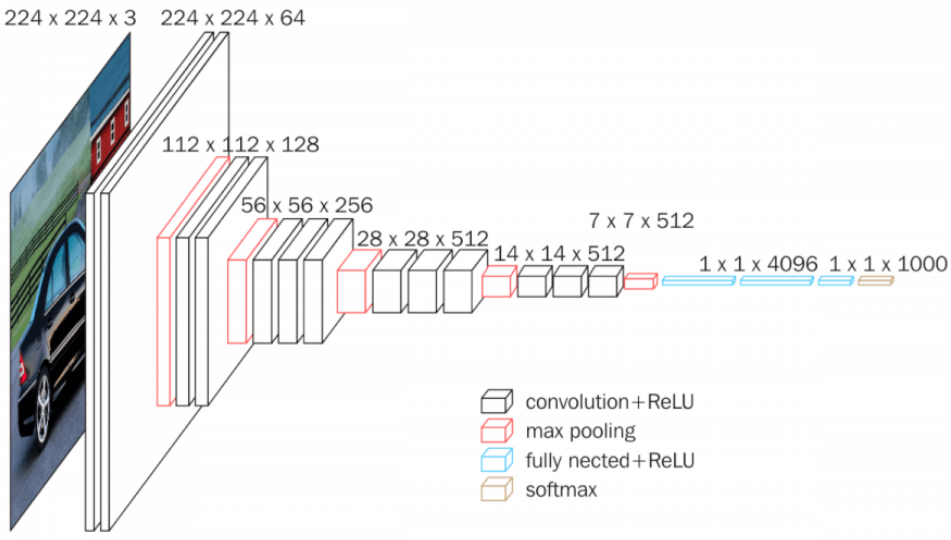



Figure 4.1: Structure of the VGG16 network.

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 $224 \times 224$ to $1 \times 1$, so called max pooling layers are used. In this particular case $2 \times 2$ max pooling with a stride of 2 was used, which means that the image is partitioned into $2 \times 2$ 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.

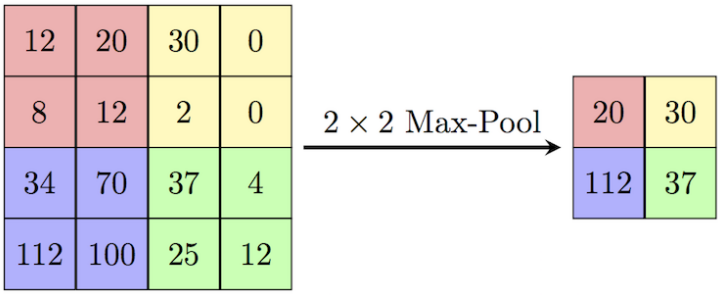



Figure 4.2: Effect of a $2 \times 2$ max pool.

Most layers of the network are convolutional layers with a small receptive field of $3 \times 3$. 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 $4096 \times 4096$ parameters that need to be trained (which is much more than the number of parameters in the convolutional layers). At the end of Section 4.3 we compute the total number of a CNN in more detail.

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

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

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.

modern network use the (stochastic) gradient descent algorithm to minimise the loss function. 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

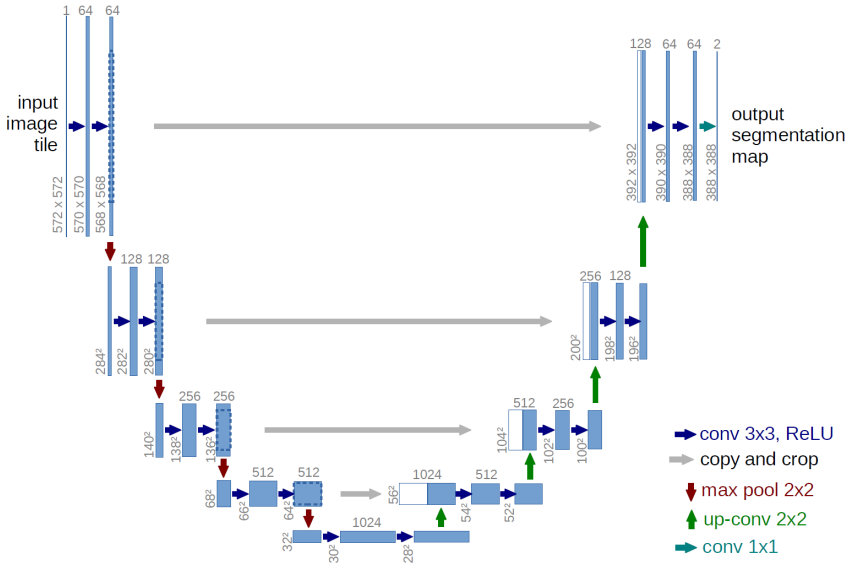



Figure 4.3: Architecture of the UNet

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 (grey, 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.

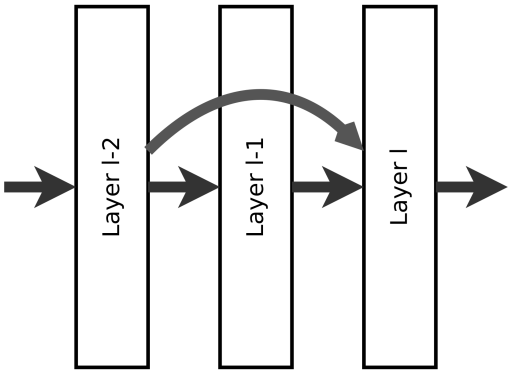



Figure 4.4: Residual block with an additional connection from the layer before the previous layer.

### Batch and Group Normalisation

Batch normalisation is a method that normalises activations in a network across the current mini-batch. For each feature, batch normalisation 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.

### Performance measures

We will now briefly discuss the importance of choosing a meaningful performance measure to evaluate the performance of a neural network that is used for semantic segmentation. A simply straightforward measure is the *accuracy* of the predictions. We simply compare the prediction and the corresponding ground truth on a pixel wise basis. The ratio of correctly classified pixels divided by the number of total pixels then gives the accuracy. This, however, has one big flaw. Typically, the majority of an image consists of background, say 90% percent, and the remaining 10% consist of the pixels that we want to classify into different classes. If we now simply classify each pixel in our prediction as background, we have an accuracy of 90% although our prediction is completely meaningless. Thus, we require a performance measure that incorporates the fact that much of the image is background, i. e., that specifically "targets" pixels in the foreground class. One that is widely used is the *intersection over union* or IoU, for short.

It is based on the following idea. Suppose we have some object that we want to recognise in the image, say a cat. Now we use the network to predict the location of the cat, i. e., we have a certain region of pixels classified as cat. If the prediction was perfect, the classified pixels and the corresponding ground truth would completely overlap. This is unlikely in practice, but this intersection should be maximised. At the same time we want to reduce the false positive errors, i. e., the pixels classified as cat that do not belong to the cat. We of course also want to reduce the false negative errors, i. e., the pixels that belong to the cat but are not classified as such. Thus, to evaluate how good of a prediction that network has made, we divide this intersection by the area of cat in the ground truth and in our prediction, i. e., the union between both. In summary, we compute the ratio between the intersection and the union, or the intersection over union. If we know simply classify everything as background, the intersection between classified pixels and ground truth is empty and thus the IoU score of this prediction would be zero, which much better represents the quality of the prediction.

### Number of Parameters and the Receptive Field of a CNN

To finish this chapter we will discuss how the total number of parameters of a particular network can be computed and we will also introduce and compute the notion of the receptive field. We consider the network given in Figure 4.5.

The only layers that contain learnable parameters are the convolutional layers and the fully connected layers. If the filter inside a convolutional layer has size $k \times k$ and the filter has $in$ input and $out$ output channels, we have a total of $in \cdot k \cdot k \cdot out$ parameters. In a fully connected layer, the number of parameters is simply $in \cdot out$. Note, however that in this case, the number $in$ depends on how the network has previously altered the input size, e. g. by max-pool layers. We begin by

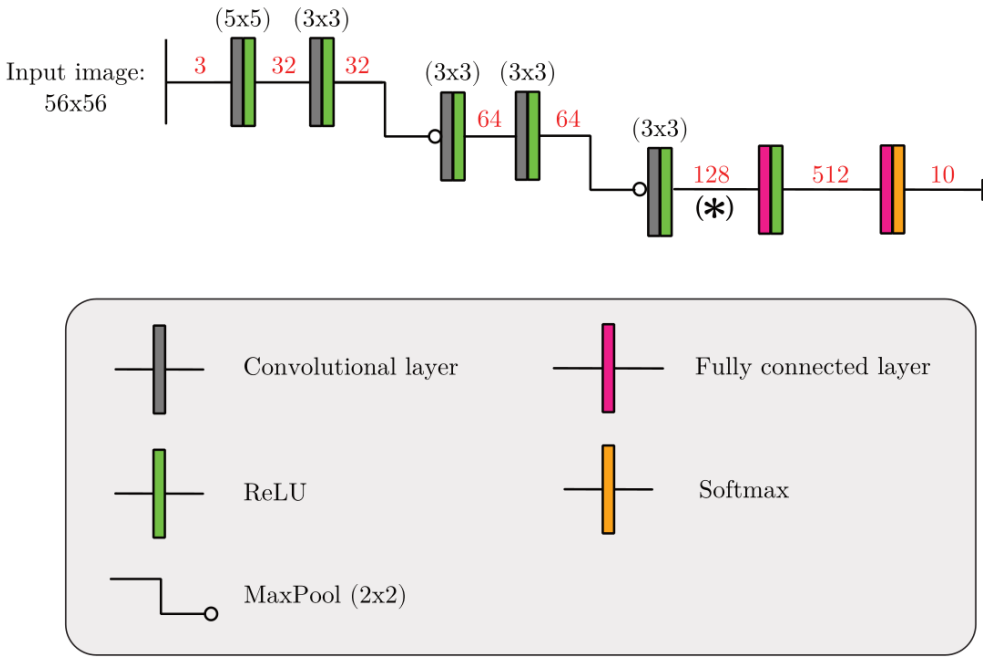



Figure 4.5: Example of very simple architecture for image classification. The input is an RGB image (three channels) with size $56 \times 56$. The numbers of channels are written in red: the number of output classes is 10. For each convolution layer, the dimension of the 2D filter is specified. In the architecture we always use "same" convolutions, so that the input is padded with zeros and the convolution output has the same spatial dimension.

computing the number of parameters for each of the convolutional layers and add them up:

$$3 \cdot 5 \cdot 5 \cdot 32 + 32 \cdot 3 \cdot 3 \cdot 32 + 32 \cdot 3 \cdot 3 \cdot 64 + 64 \cdot 3 \cdot 3 \cdot 64 + 64 \cdot 3 \cdot 3 \cdot 128\,,$$

which yields a total of 140.640 parameters in the convolutional layers. To compute the number of parameters in the fully connected layers, we first have to calculate the size of the input at the output of the convolutional part of the model, which is indicated with $(*)$ in the Figure. Since we are using same convolutions, the only layers that alter the size of the input are the $2 \times 2$ max-pool layers. Each of these layers reduces the size of the input by 2 along each axis. Since we have two of these layers in the network, the input size at $(*)$ is $14 \times 14$ since $56/2/2 = 14$. Since we have 128 input channels, the total number of inputs of the first fully connected layer is $14 \cdot 14 \cdot 128 = 25088$ and consequently the number of parameters in the first fully connected layer is $25088 \cdot 512$. The number of parameters in the second fully connected layer is simply given by $512 \cdot 10$, which results in a total of 12.850.176 parameters. All together, the network has $140.640 + 12.850.176 = 12.990.816$ learnable parameters.[2] We see that in general, fully connected layers involve many more parameters as compared to convolutional ones.

We will now discuss the notion of the receptive field and compute the receptive field of the CNN from Figure 4.5 at the output of the convolutional layer, marked with $(*)$. The receptive field is defined as that part of the input image that a particular feature of the CNN is affected by. In a fully connected layer, each neuron is connected to each neuron of the previous layer. Consequently, each feature in the input has some influence on every feature of the output. With a convolutional layer this is obviously not the case. If the convolution filter is of size $3 \times 3$, the output features are affected by 9 of the input features. Now, if we feed this output again into another convolutional layer, the convolution operation there will combine features that have themselves been computed by different parts of the input image. Thus, the receptive field was increased since each output of this layer is affected by a larger number of input features than the previous layer.

To compute the receptive field we go backwards through the network starting from the location at which we want to compute the receptive field. Each convolutional layer with a kernel of size $2h + 1 \times 2h + 1$ will increase the receptive field by $2h$ in each direction. Similarly, each max pool layer with size $2 \times 2$ will double the receptive field. We want to know by how many pixels of the input one pixel (or a patch of size $1 \times 1$) of the output of layer five is affected. Thus we begin with this pixel, adding two for the convolutional layer (layer V), multiplying this value by two to account for the max pool, then again adding two for layer IV etc. This yields

$$(((1 + 2) * 2) + 2 + 2) * 2 + 2 + 4 = 26\,,$$

thus the receptive field is $26 \times 26$ (since all convolutional layers are symmetrical).

[2] Actually, the number is even slightly higher since we neglected the so called bias terms.

# Week 5 Instance segmentation

**Summary** In this Chapter we discuss different techniques to segment different instances of the same class in an image. To that end, we introduce the notion of bottom-up and top-down methods which are then both discuss an more detail.

## Lecture 5.1 Fundamentals

We previously discussed semantic segmentation, where we want to assign a class to each pixel in an image. If our image contains a group of people then the semantic segmentation network might be able to find this group of people but it is unable to decide where one person ends and where the next person begins. This is precisely the goal of instance segmentation: we want to find all individual instances of a particular class. More precisely, we will call this task semantic instance segmentation or panoptic segmentation (since we try to identify different classes and within each class identify the individual instances). If there is only one class present, then we will refer to the task of identifying the individual instances of this single class as instance segmentation without segmentation or image partitioning; one example is illustrated in the Figure below. Note that the

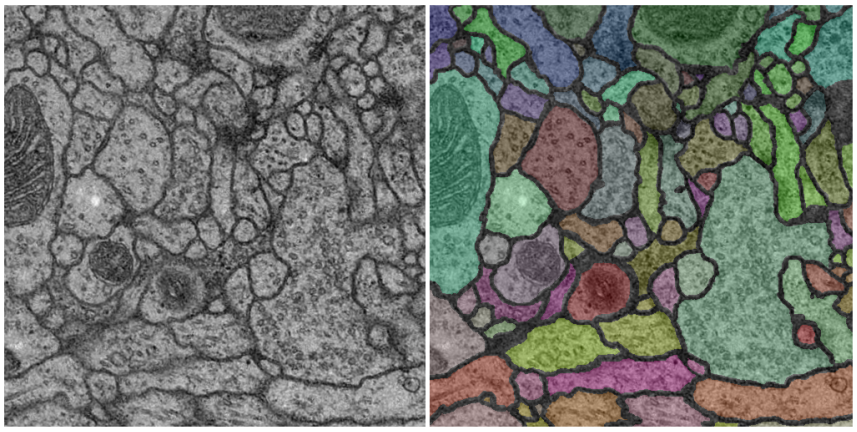


Figure 5.1: Example of an image of neural structures in a brain and the corresponding instance segmentation.

colours in the partitioned image are chosen randomly and are not important; what is important is that two pixels that have the same colour belong to the same instance. Another example where a little bit of semantics have to be learned is instance segmentation with the additional task of deciding whether a pixel is background or foreground. The instance segmentation is then done only for the foreground pixels; the background that has only to be identified as such. What all of these problems have in common is that the number of instance that we try to identify is not

known beforehand. If we know that there will be a fixed number of instances in the image, different methods should be used that we will not discuss in this lecture.

The general strategies to solve these kinds of problems can be broadly categorised into *top-down* and *bottom-up* approaches.

- Top-down This approach is also called *proposal-based*. Here, we first try to find one seed (i. e., , the centre of the instance) or one bounding box for each instance and only then try to estimate the precise shape of that instance. One approach to find the exact shape given a bounding box would be to divide the bounding box into, for example, $16 \times 16$ pixels and classify each pixel (does it belong to the instance or not) using techniques discussed earlier.
- Bottom-up This *proposal-free* strategy can be again divided into two sub-strategies. The first can be seen as a generalised Hough transform (discussed below) where each pixel directly votes for instance membership. This then implicitly also gives the shape of the particular instance. In the second approach we try to decide for pairs of pixels whether or not they belong to the same instance and we were to do this for all of the pixels and all possible pairs this would give as a signed graph. This signed graph can then be partitioned in the "least hurtful" way.

## Lecture 5.2 Proposal-based methods

There is a wide variety of top-down strategies that have successfully been used in the past. The following generic algorithm tries to summarise the general underlying idea which in essence can be interpreted as a sliding window approach.

1. For each location answer one or more of the following questions.
   - Am I a seed? Classification
   - If I am a seed, to what class do I belong? Classification
   - Relative to me, where is the closest seed? Regression
   - What is the bounding box relative to my location? Regression
2. Usually, the first step will produce several seeds or several bounding boxes for each instance but of course we only want one seed or one bounding box. Therefore, we have to somehow decimate them. We call this step the inference step; it is discussed in more detail below.
3. Having now only one seed or one bounding box, we can estimate precises mask or outline of the respective instance conditioned on the surviving seeds from the previous step.

A simple technique for improving the first step is to slide windows of several aspect ratios and sizes across the image simultaneously (instead of just a single window) and asking these questions for every size.

Below we will discuss some strategies that can be used in the inference step. As mentioned above, we will likely have multiple bounding boxes or multiple seeds that where there should only be one. The following is a very simple method to "clean up" the additional, unwanted seeds and boxes. There are more advanced variants of this algorithm and also other methods to solve this problem.

### Non-maximum suppression

After the first step in the generic algorithm above, we have possibly multiple detections for each instance. To each of these detection a probability is assigned and we start by taking the overall detection with the largest probability and set it to be the first actual detection. When the "kill" every detection in a fixed neighbourhood of this detection. Thereafter, we do the same thing again but with the second most plausible detection. Since we removed all the detections in the vicinity of our first true detection, we hope that the second largest detection belongs to a different instance in the image. This is repeated until no detections are left. We do also apply a threshold and detections below that threshold will not be considered (they are considered as being too unlikely).

## Lecture 5.3 Hough Transform

In general, Hough transform is a feature extraction technique; more precisely it can be used to find imperfect instances of parameterised objects by a voting procedure. This voting procedure is carried out in a parameter space, from which object candidates are obtained as local maxima in a so-called accumulator space.

In the simplest case of the Hough transform, the goal is to detect straight lines. Since the "classic" parametrisation $y = mx + b$ leads to unbounded values in the case of vertical lines, one rather uses the form $r = x\cos\theta + y\sin\theta$, where $r$ is the distance from the origin to the closest point of the straight line, and $\theta$ is the angle between the $x$ axis and the line connecting the origin with that closest point (of course other parametrisations are also possible). It is now possible to assign to each line a pair $(r, \theta)$, or a point in the $(r, \theta)$ plane.

Given a single point in the plane, the set of all straight lines going through that point corresponds to a sinusoidal curve in the $(r, \theta)$ plane and this curve is unique to this point. A set of two or more points that form a straight line will produce sinusoids which cross at the $(r, \theta)$ point that parameterises this line.

## Lecture 5.4 Similarity Learning

Instead of directly learning whether or not to pixels are similar we learn a mapping $e$ from the observations (i. e., , the image) $X$ into some embedding space $\mathbb{E}$ (e. g. , $\mathbb{R}^k$). We can then compute the similarity of two pixels $x_1$ and $x_2$ by computing their respective distance in the embedding space, i. e., we compute

$$s(x_1, x_2) = s_{\mathbb{E}}(e(x_1), e(x_2)) = s_{\mathbb{E}}(e_1, e_2)\,.$$

This is usually more efficient, since the distance in the metric space is typically simpler to compute (for example, $s_{\mathbb{E}}$ might just be the Euclidean distance). The loss function to train the neural network which in this case is called contrastive loss or associative loss is given by the following equation

$$L = \sum_{x_i, x_j} \delta_{c_i\, c_j}(1 - s(e(x_i), e(x_j))) + (1 - \delta_{c_i\, c_j})[s(e(x_i), e(x_j)) - \alpha]_+\,,$$

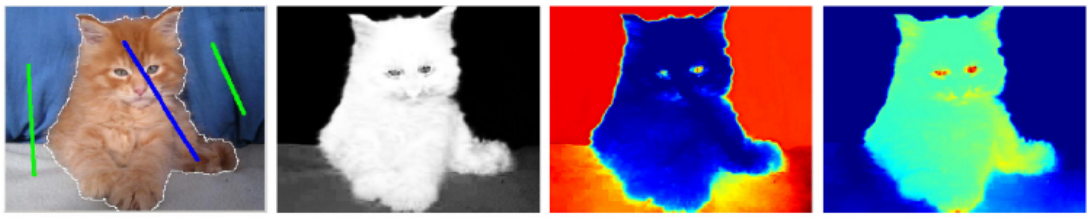
where $c_i$ and $c_j$ are the associated instance of pixel $x_i$ and pixel $x_j$, respectively. Thus, the first part inside the sum is "turned on" if two pixels belong to the same instance whereas the second part is turned on when they belong to different instances. Since in the former case we would like the loss to be low, i. e., their similarity should be high. In the latter case we want their similarity to be low (here $[x]_+ = \max(0, x)$, i. e., similarities below a threshold of $\alpha$ will be cut off).

# Week 6 Shortest Paths

**Summary** In this week's lecture the shortest path problem is introduced. We begin by discussing examples of shortest path problems in computer vision. Thereafter we introduce two algorithms to solve this problem and also discuss one particular example on more detail.

## Lecture 6.1 Examples

Shortest path algorithms are used in different areas in computer vision, in the following we briefly discuss two examples. The first example is seeded segmentation. Consider the Image (a) of the following Figure.



(a) (b) (c) (d)

A user draws the blue and green line to annotate foreground and background, respectively. Image (b) then shows a for each pixel the similarity to the foreground class; white means the probability that the particular pixel belongs to the blue class is high while black means the probability is low (in essence, this can be seen as a very simple classifier). Based on this image a graph connecting the pixels in a grid is generated and for each pixel the shortest path to the closest blue pixel and the closest green pixel is computed. The total costs of these shortest paths, i. e., the distances to the closest blue and green points are given in the last two images. Blue here means low distance, green means intermediate distance and red means high distance. Based on these two images a segmentation into foreground and background can be obtained which is indicated in the first image as the border around the cat.

Another example is computing a depth map from a stereo image which is discussed in more detail in Section 6.3. The fundamental approach to tackle shortest path problems is **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. Consider the graph in Figure 6.1. We do not know if the shortest path from $s$ to $t$ will go through $k$; if it does go through $k$, however, then the shortest path $s \to k$ must be part of the globally optimal shortest path $s \to t$.

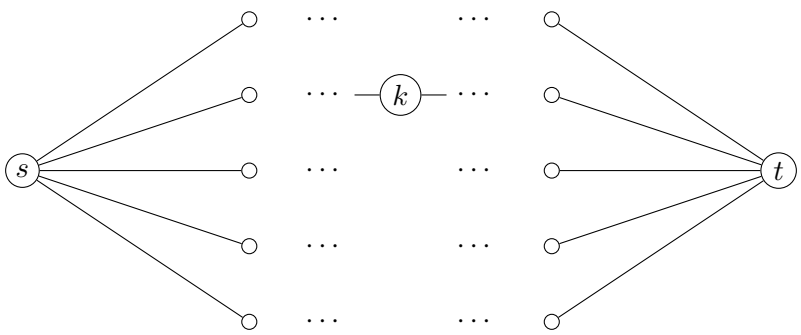



Figure 6.1: We try to find the shortest path from $s$ to $t$. We do not whether or not the shortest path will go through $k$ but if it does,

Note, that it really 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 complexity 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

Below we introduce two dynamic programming algorithms that can be used to solve the shortest path problem. We begin with Viterbi's algorithm which can be defined in different ways; the version we give below is sometimes referred to as the forward update version of Viterbi's algorithm.

**Algorithm 6.1:** Viterbi's algorithm

**Result:** Shortest path starting from node $s$
topologically sort the vertices of $G$;
**foreach** *vertex* $v$ *in topological order* **do**
  **foreach** *edge* $e = (v, u)$ *in the forward star of* $v$ **do**
    $d(u) = \min(d(u), d(v) + w(e))$;
  **end**
**end**

We start from the node $s$ and follow all edges in the forward star of this vertex, i. e., all outgoing edges. All of these edges connect $s$ to some node $u$ and we now compute the distance from $s$ to $u$ which is given by the edge weight and possibly and additional weight imposed by the vertex itself. If have done this for every edge in the forward star of the start node, we now know the cost for reaching each of its outgoing neighbours. We repeat the process for each of these vertices, call them $v$, i. e., we again follow the edges in the forward star and compute the distance from $v$ to all nodes $u$ that we can reach. Now there are two possibilities. If we have not yet visited the node $u$ through some other path, we simply set the distance of reaching $u$ to the distance of reaching

$v$ plus the edge weight of the edge connecting $v$ and $u$ (and possibly also plus the additional weight of $u$ itself). If, however, we have already reached $u$ by some other path, we again compute the distance as above but also compare it with the distance from the other path and only keep the smaller of both. If we additionally remember for each vertex where we came from, we can then easily construct the shortest path from any vertex back to the starting vertex $s$ by simply following this path backwards.

In a later lecture we will discuss this shortest path problem in a more general setting where the notion of distance and the notion of shortness might be defined differently. In this case, the algorithm still applies; however, we have to replace the step where we update the distance by

$$d(u) = d(u) \oplus (d(v) \otimes w(e)) \ ,$$

where $\oplus$ and $\otimes$ are generalised operations of what in our case is the minimum and the sum operation.

Note that then the graph must not contain any negative cycles since this makes the notion of shortest path meaningless (the shortest path would be given by following the negative cycle infinitely many times). Also, Viterbi's algorithm can only be used for directed acyclic graphs, i. e., graphs that do not contain any cycles. If the graph does contain cycles, a similar algorithm called Dijkstra's algorithm can be used.

**Algorithm 6.2:** Dijkstra's algorithm

```
Result: Shortest path starting from node s
Set distances to all nodes but the starting node to ∞;
Initialise queue Q ← V, prioritised by distance;
foreach Q is not empty do
    Extract minimum distance vertex v from Q;
    foreach edge e = (v, u) in the forward star of v that is still in the queue do
        d(u) = min(d(u), d(v) + w(e));
    end
end
```

Although Dijkstra's algorithm will not fail when the graph contains negative edges (or even a negative cycle) it will not necessarily yield the correct result. In practice one would typically also store for each vertex the vertex from which we have reached this node so that in the end we can easily reconstruct the shortest paths from any node to the starting node by following these predecessor pointers beginning from the end node.

## Lecture 6.3 Scanline Optimisation/ Stereo Disparity Estimation

We consider the follow situation. There are two cameras, slightly shifted or rotated and rectified, and they take both take a pictures of the same scene. The goal is to estimate the "depth" of the elements in the scene. As illustrated in the image in Figure 6.2 below, the two pictures are evaluated along a horizontal scanline (the example is taken from `http://lunokhod.org/?p=1356`).

Figure 6.2: 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 Figure 6.3 below.

Figure 6.3: All possible disparities along one scanline.

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 6.4. 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.

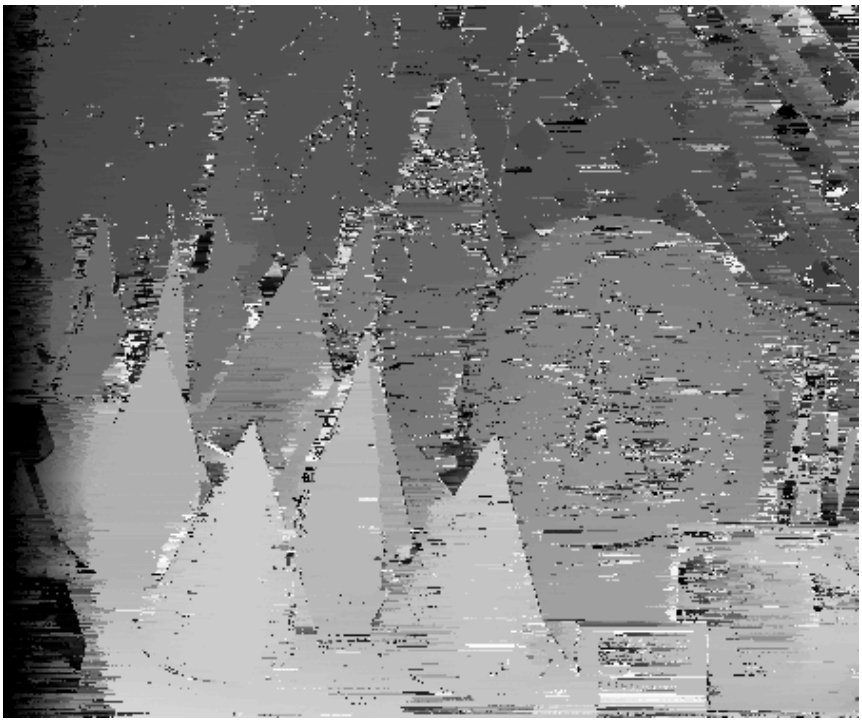


Figure 6.4: Disparity map for the example above.

## 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 $i$th 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).

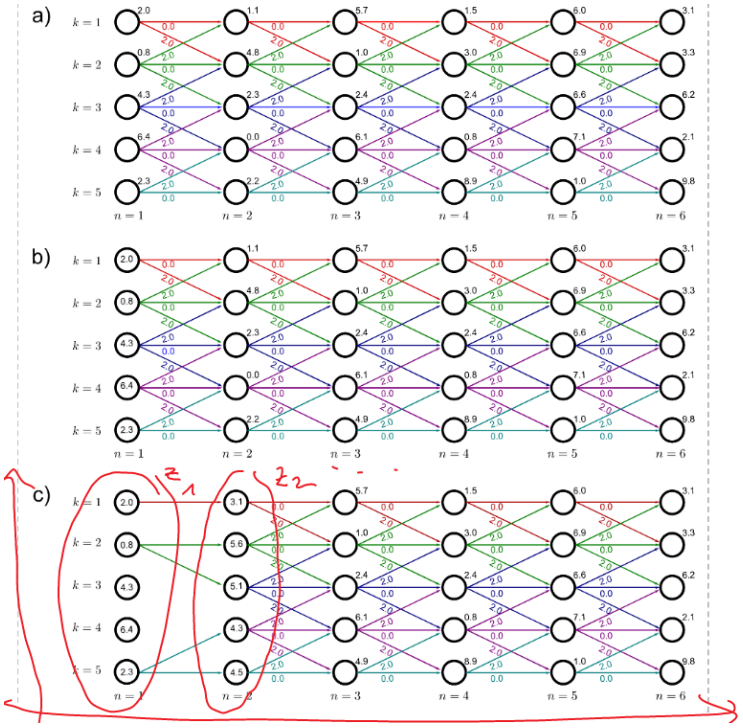



Figure 6.5: State vectors

# Week 8 Widest Paths

**Summary** This chapter is concerned with the widest path problem. We explain its relation to the shortest path problem and introduce an algorithm to solve it. We discuss the notion of a minimum spanning tree (MST) and examine the Watershed algorithm which is based on computing MSTs.

## 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.

## 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.

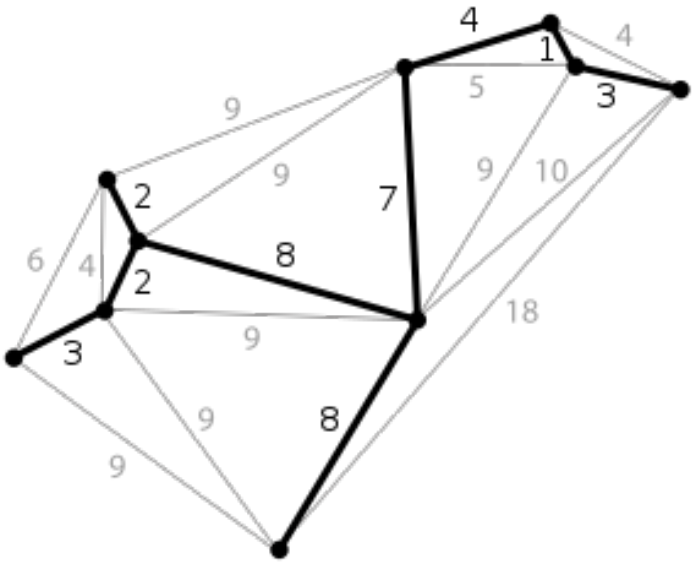



Figure 8.1: A minimum spanning tree.

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 turn, 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 *ultrametric 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.

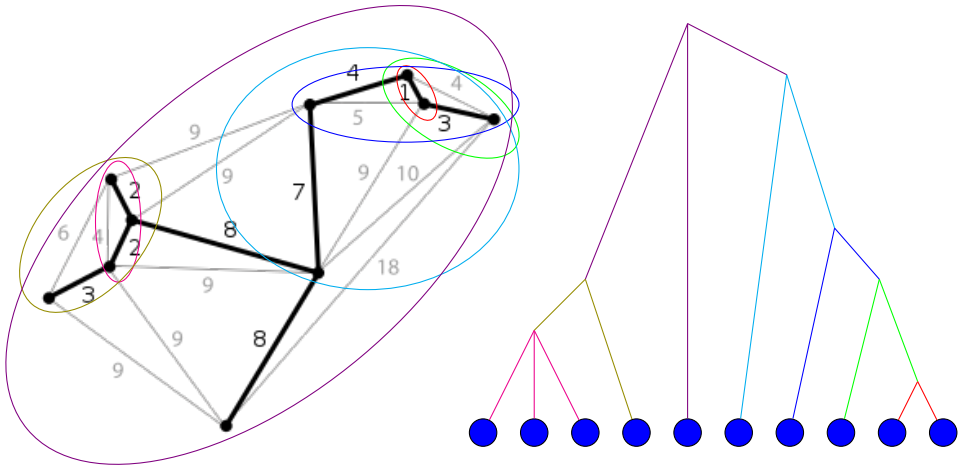
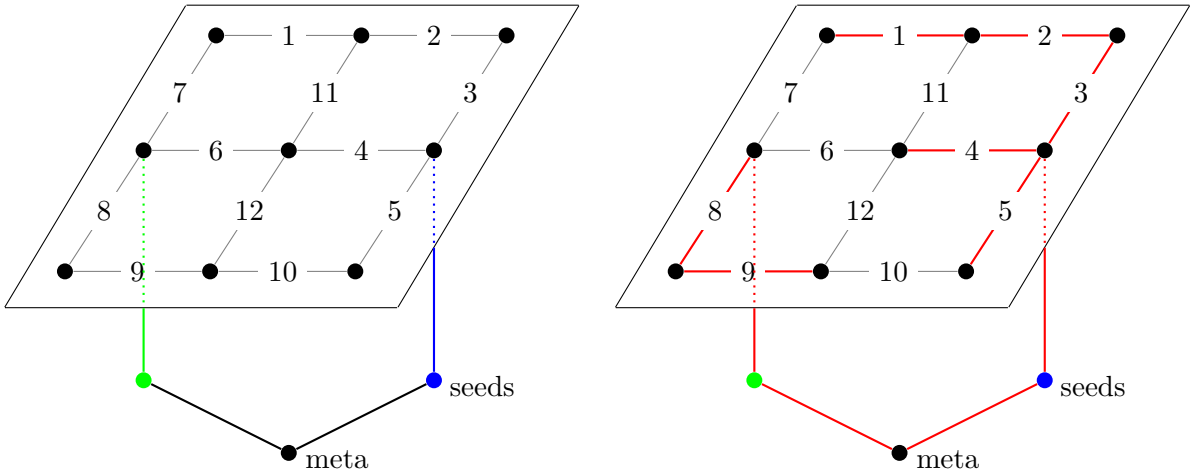



Figure 8.2: Minimum spanning tree and corresponding ultrametric tree. The ultrametric tree is built by repeatedly finding the nodes that have the smallest distance to the nodes already processed (indicated by the ellipses).

## Lecture 8.5 Seeded Watershed Segmentation

The watershed algorithm can be used to segment an image. We begin with a version of the algorithm where the user has to provide seeds for, e. g. foreground and background classes. Consider the example image in Figure 8.3 where we want to partition the image into the two given classes. On the left we have the original image which has already been transformed into a graph where the

edge weights might come from a neural network or some other means. Starting from the artificial meta node we compute the minimum spanning tree using Prim's algorithm. The result is shown on the right.




(a) Original graph

(b) Graph partitioned into two parts using Prim's algorithm starting from the meta node

Figure 8.3: Seeded watershed algorithm applied to a simple $3 \times 3$ image.

This clearly induces a "cut" through the image an we can now assign the upper right pixels to the blue class and the lower left pixels to the green class. One can also use the same idea without providing seeds manually but instead choosing them automatically. In this algorithm which is then simply called *watershed segmentation*, the local minima of the image are used as seeds. This can be used in practice to find so called "superpixels" to coarse-grain a computation. These superpixels should lie within one instance in the image, but they might not fill the entire image; in other words we coarsened the image but still kept the segments separate. On the graph corresponding to these superpixels we can then perform seeded watershed algorithm. Also other variants of watershed are possible where the seeds are found, for example, by a deep learning method.

We have now discussed two different approaches to perform seeded segmentation: one using shortest paths and one using minimax paths (watershed) and they can behave very differently. In the first variant, i. e., when using the sum of edge weights as the cost of a path, the costs obviously accumulate over distance and thus the proximity of pixels to seed very much matters (e. g. very long thin segments might not be classified as one segment). Thus, a single seed will not take us very far. However, a segment with an incomplete boundary will not easily "bleed" through this boundary which often happens in seeded watershed segmentation. When using the maximum of edge weights, the cost of a path depends on the single weakest edge and thus, a single seed can give very large segments.

# Week 9 Algebraic Graph Theory

**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 $\infty$ 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

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

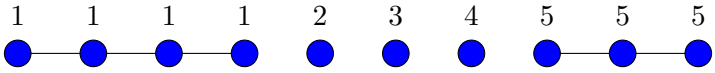
In other words: along a path, each single edge weight has to be `TRUE` (this is the inner `AND`) while of many different paths only one has to be `TRUE` (this is the outer `OR`) for $d(u)$ to be set to `TRUE`. Example applications for this problem are

- 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 generalised-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.

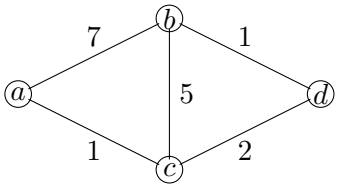
We now start from the left and assign an arbitrary label to the first node. If the next node is connected to the current node then we assign to it the same label. Otherwise we choose a different label. Using numbers as labels, our graph from above might then look like this. Checking if two




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 adjacency matrix. where the entries $(a, d)$ and




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

$(d, a)$ are set to $\infty$ 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 adjacency 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 optimised 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

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$
$$c \otimes (a \oplus b) = (c \otimes a) \oplus (c \otimes b)\,,$$

   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

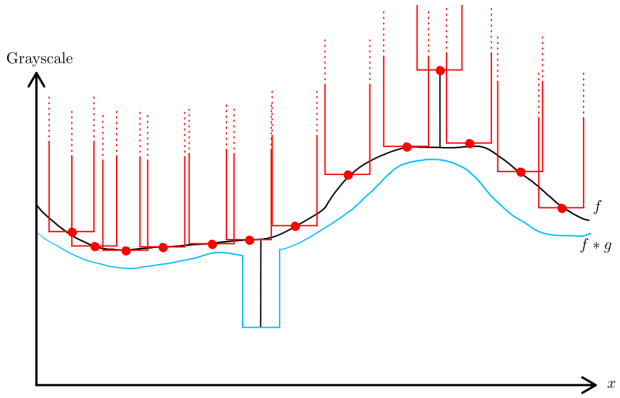
$$f \star g(x) = \sum_y f(x-y) \cdot g(y) \,,$$

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

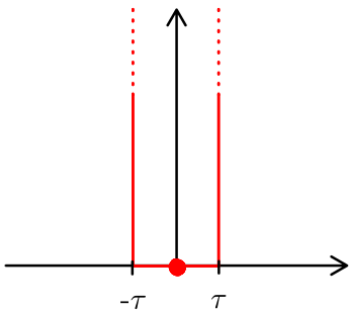
$$f \star g(x) = \inf_y \left( f(x-y) + g(y) \right) \,,$$

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 greyscale image together with the result (light blue line) of the erosion using the structural 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.

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




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




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

Figure 9.1: Example of infimal convolution.

and dilation to obtain

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

and

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

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 objects, 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”.

# Week 10 Tracking

**Summary** In this week's lecture we discuss the tracking problem. We begin with explaining the basic ideas of three fundamental approaches of which one is discussed in more detail, namely tracking by assignment. To that end, we introduce the notion of a min-cost flow problem and show how it can be interpreted as an (integer) linear programs. We also discuss that the hard-to-solve integer linear program that arises in the min-cost flow formulation can be efficiently solved by reducing it to a standard linear program.

## Lecture 10.1 Fundamental approaches

Tracking has many application in computer vision, for example tracking people in a video or tracking dividing cells sequence of microscopy images. The latter poses a particularly difficult challenge, since here targets (the cells) can divide and we want to track for each offspring from which cell it has originated.

We discuss three "school of thoughts" for solving such tracking problems. The first, called *space-time segmentation* we simply treat time as a third space dimension and apply instance segmentation methods similar to those that we have discussed in previous chapters.

In the second approach, called tracking-by-assignment or data association, we first try to detect the objects of interest at each time step individually. If every time step is processed, we then consider possible associations between the detected objects in neighbouring time steps (in practice we do not consider all possible associations since often certain assumptions are made a priori, for instance we know that cells only move with a certain speed and thus, cells that are at a certain position at time $t-1$ can only be within a certain region of this position at time $t$). The last class, which we not discuss in this lecture, are state-space models such as the Kalman filter or generally hidden Markov models. In Table 10.1 we listed some of the advantages and disadvantages of the three approaches.

In all three schools we have a neural network that gets the dense video input which usually has the same (dense) output dimensions. Often, for instance when tracking dividing cells, we would rather like a tree that shows how the different cells and their offsprings behaved and divided over time. That is, we want a sparse output. This dense to sparse conversion is something that convolutional neural network often struggle with.

| | ① | ② | ③ |
|---|---|---|---|
| Can handle large displacements | + | + | + |
| Affords joint segmentation and tracking | + | + | − |
| Can handle unknown number of particles | + | + | − |
| Can handle dividing particles | + | + | + |
| Has representation for the internal state | − | − | + |

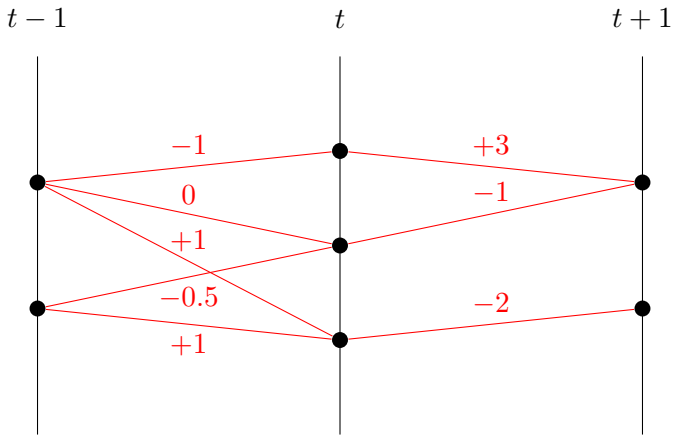
Table 10.1: Advantages and disadvantages of space-time segmentation (①), tracking-by-assignment (②), and state space methods (③) when used for tracking.

## Lecture 10.2 Tracking by association: Min cost flow

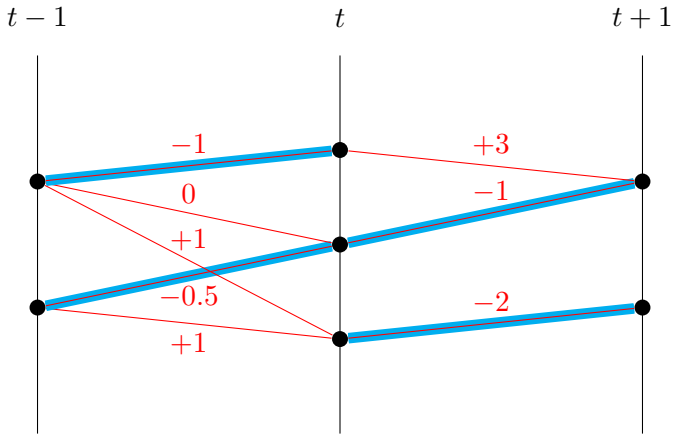
We begin with a simple example of tracking without division (we also assume that no new detections are born and detections do not die). The detections as assignments are given for three time steps in Figure 10.1. The second and third image show the detected tracks when matching detections only between two consecutive frames and when using a shortest path approach, respectively. We can see that both are not applicable for this task since the results are not consistent with our assumptions.

Hence, we need another way to find suitable paths in this graph to which end we introduce the notion of min-cost flows. In this setting we again assume that paths are additive along paths. The min-cost flow formulation of the problem enforces the laws of conservation and we can impose a restriction on the number of times one vertex can be used, which in our case would be one. To demonstrate the idea, we have to slightly modify the graph from above, the results is shown in Figure 10.2. There, we have duplicated each "layer" of the graph and connected the two arsing layers to each other (the blue edges). The purpose of this modification is as follows. Sometimes we might assign a cost (or a reward) to certain implausible (or very plausible) detections. This can now be achieved by assigning a cost (reward) of accepting to the blue edges which. This formulation also allows to easily model spontaneous appearing and disappearing detections. To model appearing detections we can add edges from the starting node to any of the edges in the first parts of the second or third layer and analogously, we can add edges from the second parts of the layers to the target vertex to model spontaneous disappearance. In the figure we also indicated the optimal min-cost flow solution; before discussing how to obtain these solutions in the general case, we first define the precise optimisation problem that we try to solve.

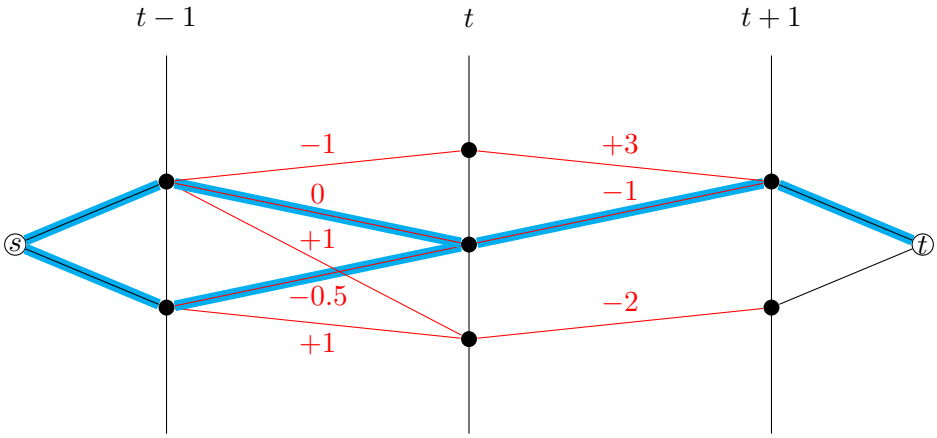
To that end, we consider a graph $G(V, E)$ with vertices $V$ and edges $E$ in which we want to find a flow $x$ (note, that the graphs in the examples are actually directed; we omitted the arrows that would indicate the directions since in our case the direction is always from left to right). The




(a) Detections (black dots) at three time steps with assignments and costs.




(b) Matching detections only in pairs of frames. The results are not consistent across frames.




(c) Matching detections using the shortest path and second shortest path. A single edge is used more than once which is not desired.

Figure 10.1: Example for tracking without division.

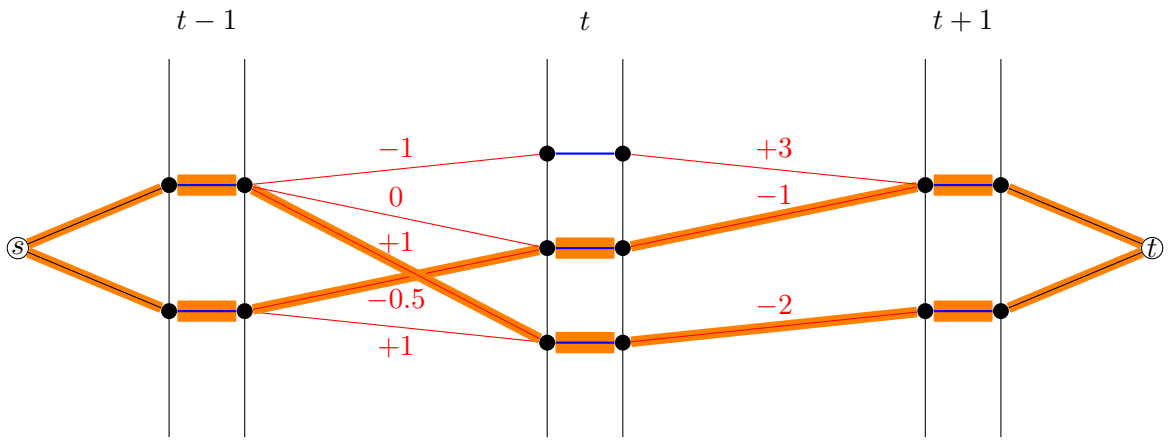



Figure 10.2: Min-cost flow formulation of the previous problem.

optimisation problem can now be written as

$$\text{Find} \quad \min_x \sum_{e \in E} x_e c_e \,,$$

where $x_e$ is the flow across edge $e$ and $c_e$ is the cost of sending one unit of flow across edge $e$. That is, $c_e$ is the cost of using an edge while $x_e$ is the number of times an edge is used. To ensure that flows go from the starting node to the target node, we impose the following conservation law for all $v \in V \setminus \{s, t\}$

$$\sum_{u:\,(u,v)\in E} x_{(u,v)} = \sum_{w:\,(v,w)\in E} x_{(v,w)} \,.$$

This ensures that all for every oncoming connection to a node $v$, there is also an outgoing connection from $v$. To ensure that the flow should not exceed a certain upper limit, i. e., a certain capacity on a particular edge, we add the following constraint for all edges $e$

$$x_e \leq b_e \,,$$

where $b_e$ is the bottleneck associated with the edge $e$. In our example above, we would have $b_e \leq 1$. In many tracking problems, we can also impose an integral constraint such as

$$x_e \in \{0, 1\} \,.$$

This ensures that there cannot be half a person, or half a cell that is tracked but that one detections corresponds to one instance. Note that this last assumptions makes the problem non-convex and thus hard to solve but luckily, this constraint can be replaced by the convex constraint $x_e \geq 0$ for min-cost flow problems.

### Linear Programming

We can rewrite the problem above in the canonical form of an eminent optimisation problem, a so called *Integer Linear Program* or ILP, for short. We start by defining what we mean by a linear program. It is the following optimisation problem

$$\min_x c^{\mathrm{T}} x \qquad \text{s.t.} \quad Ax \leq b \quad \text{and} \quad x \geq 0\,.$$

This is a convex problem and is thus easy to solve (using the Simplex algorithm or the interior point method). If we add the following integer restriction

$$x \in \mathbb{Z}^{\dim(x)}\,,$$

the problem becomes non-convex and solving it is now NP-hard.

In summary, we have seen that we can model tracking-by-assignment as a minimum-cost flow problem and thus as a linear program which can be solved to global optimality efficiently (in polynomial time). Integer linear programs, on the other hand, are usually very hard to solve; however, if the have certain special structures they can sometimes be reduced to easier problems. For instance, if the constraint matrix is totally unimodular and the right hand side is integer-valued, the problem can be reduced to a standard linear program. Also, shortest path problems (or more generally, flow problems) are only one instance of ILPs that happen to be easy to solve.

## Lecture 10.3 Total Unimodularity

As mentioned above, under certain circumstances can an ILP be solved efficiently, for instance when the constraint matrix is totally unimodular.

**Definition 10.1.** Let $A \in \mathbb{Z}^{m \times n}$ be an integer matrix.

(a) If $A$ is square, we say $A$ is *unimodular* if $\det(A) = \pm 1$.

(b) $A$ is called *totally unimodular* if for every submatrix $B$ of $A$ it holds that $\det(B) \in \{-1, 0, 1\}$.

One can show that every for entry $a_{ij}$ of totally unimodular matrix $A$ it holds that $a_{ij} \in \{-1, 0, 1\}$, which is not true for unimodular matrices.

As mentioned above, if the constraint matrix of a ILP is totally unimodular and the right hand side is integer-valued, the ILP can be solved efficiently. This is due to the fact that in this case the feasible region has integer vertices only which implies that the integer linear program solution coincides with the normal linear program solution. It turns out that any min-cost flow problem with integer edge capacities that does not have negative cost cycles has an optimal solution with integer flows on its edges.

A main ingredient of proofing the results above is the following fact. The linear equation $Bx = b$ with a non-singular integer-valued square matrix $B$ and an integer-valued vector $b$ has an integer-valued solution $x$ if and only if $B$ is unimodular.

In summary, "pure" flow problems (max-/min-cost flow problems) can be described by linear programs with totally unimodular constraint matrices. Given integer edge flow bounds (right hand sides in the LP), this guarantees integral solutions of the optimisation problem. Hence, such pure flow problems can be efficiently solved to global optimality.

# Week 11 Optimal Transport

**Summary** In this lecture we discuss optimal transport. After motivating the topic we examine the discrete case where we show that discrete optimal transport can be rewritten as a linear programming problem; more precisely, it can be rewritten as a min-cost flow problem.

## Lecture 11.1 Motivation

Optimal transport is about finding the cost, and possibly the transport plan, to transmogrify one distribution into another; one example is illustrated in the Figure 11.1.

One important special case of this problem is when the bins have a spatial layout, and the transport cost is a metric on that space. Another important special case within this special case is when the cost is given by the Euclidean distance in which case we call this cost the *earth mover's distance*. In this case the problem can be interpreted as to find the optimal strategy to move one pile of dirt to somewhere else. Another example would be the optimal transport plan of moving certain source items (e. g. from a factory) to certain target items (e. g. to distribution centres). Of course, these examples can also be generalised to continuous cases.

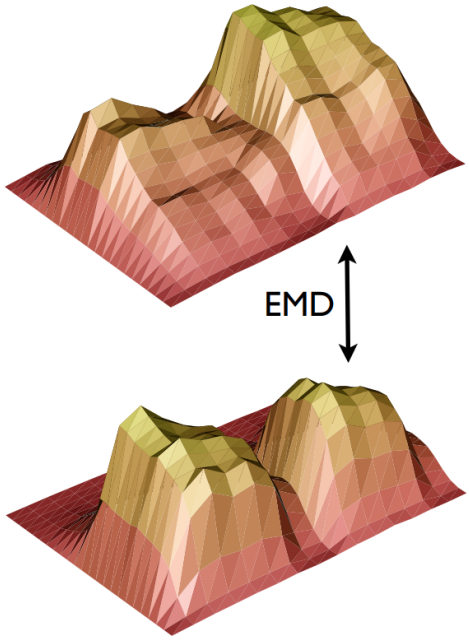



Figure 11.1: Transmogrifying one distribution into another. The transport cost is measured in terms of the earth mover's distance (EMD).

In the context of computer vision we can use optimal transport to measure the discrepancy between different *probability* distributions. This can be used, for example, to decide for two pairs of distributions which of the pairs are more similar to each other. A more concrete example is the automatic generating of artificial images from given images, e. g. generating images of bed rooms or faces etc. The connection to optimal transport will be discussed later. Optimal transport can also be used to interpolate between distribution. This often yields better results than simple linear interpolation. This can be used to interpolate between 3D shapes of objects (since they can be interpreted as distributions) and to generate intermediate shapes that partly look like both of the original shapes.

## Lecture 11.2 Discrete Case

As mentioned above, the goal is to transmogrify one distribution (the source) into another distribution (the sink). If both the source and the sink are discrete (i. e., both are probability mass functions which are functions that give the probability that a discrete random variable is exactly equal to some value), then the problem can interpreted as a min-cost network flow problem which can be efficiently solved using linear programming. If either of the distributions is continuous (i. e., it is a probability density function) the problem becomes harder to solve. If both are continuous, the problem becomes yet harder although there are certain special cases where it is easy to solve. We can visualise the task as in Figure 11.2.

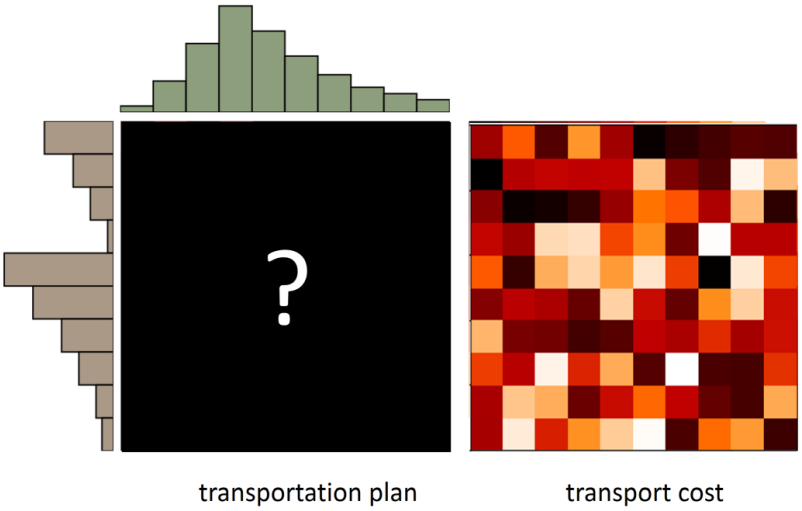



Figure 11.2: Examples for discrete distributions and transport cost matrix.

To derive the min-cost flow problem we assign vertices to each bin of the green distribution and also to the bins of the red distribution (as was the case in the previous chapter, we can duplicate these vertices to impose additional constraints). The we connect the respective edges from one distribution to all edges to the other distributions and set the cost of the new edges as given by the cost matrix on the right. By now adding an additional source and one sink node that connect to the first and the second distribution respectively, we have converted our problem into a form that is similar to the one discussed in the previous chapter which allows us to interpret it in terms of a linear program.

Alternatively we can the derive the linear programming formulation as follows. Let $P \in \mathbb{R}^{n \times m}$ the flow matrix we want to find and let $C \in \mathbb{R}^{n \times m}$ be the (given) cost matrix. The problem is

now given by

$$\begin{aligned} \text{Find} \quad \min_P \sum_{i=1}^{n}\sum_{j=1}^{m} P_{ij}C_{ij} \\ \text{s.t.} \quad \sum_{j=1}^{m} P_{ij} = a_i \quad \forall i \\ \sum_{i=1}^{n} P_{ij} = b_j \quad \forall j \\ \mathcal{P}_{ij} \geq 0 \quad \forall i,j\,, \end{aligned}$$

where $a_i$ and $b_j$ are the values of the respective bins of the first and second distribution, respectively. In other words, this is the amount that has to be transported and so the first to constraints make sure that everything that needs to be transported from the source does really get transported and that the target receives everything it needs to receive. To make it clearer that this is a linear program we first rewrite it as

$$\begin{aligned} \text{Find} \quad \min_{P\in\mathbb{R}_+^{n\times m}} \langle P, C\rangle \\ \text{s.t.} \quad P\mathbb{1}_m = a \quad \text{and} \quad P^{\mathrm{T}}\mathbb{1}_n = b\,, \end{aligned}$$

where $\mathbb{1}_k$ is a $k$-dimensional vector containing only ones. By identifying the matrices as vectors, we could also rewrite this again to obtain the standard form of a linear program. This problem can now be solved using standard linear programming solvers. Note, however, that it is usually more efficient to directly use a solver that is specifically designed to solve min-cost flow problems rather than a general LP solver. Alternatively, one can use an algorithm that approximates the optimal transport solution which is faster than the exact LP solver while yielding sufficiently accurate results. One such algorithm is discussed in the next section.

## Lecture 11.3 Wasserstein Generative Adversarial Networks (GANs)

To conclude this Chapter, we briefly discuss a class of neural networks that can be used to generate new images that are similar to training images that are given to the network. To use the ideas from optimal transport discussed above we first interpret the images as points in a high dimensional feature space that are located on a manifold inside this space (for instance, the feature space might be the space of images with $512 \times 512$ pixels and the manifold might be the subset of images that show bed rooms). The basic idea of GANs is now as follows. We want to use a neural network that tries to learn a probability distribution on this manifold from which we

can draw samples (i. e., new artificial examples of the given class of images). More precisely, the network only learns a generator function that takes samples from a low-dimensional probability distribution (e. g. a 64-dimensional Gaussian) and maps them to the feature space. To measure the discrepancy between our generated images and the training data we cannot use the same ideas as we have used with CNNs since we have no ground truth for the artificially generated images. Hence, we interpret the given training data as a distribution in the feature space and do the same with a number of images generated by our network and use the earth mover's distance as a measure of discrepancy.

This, however, gives rise to a new problem: it is computationally costly to compute the EMD in the high dimensional feature space. Luckily, this task can also be well-approximated using another neural network. This is done by first transforming the linear program discussed above into its associated dual problem, which yields the same minimum as the original problem (we do not discuss this duality in more detail here). This dual problem is then learned by a neural network. That is, the loss function of the first network is (at least partly) learned by a second network.

In summary, a Wasserstein GAN tries to find parameters of an decoder network that maps samples from a primitive low-dimensional distribution to the space of images such that the generated images have a small EMD to the real images. However, the EMD itself is also learned by a second neural network and this network and the decoder are trained simultaneously in an adversarial manner.

# Appendices

# A Disjoint-Set/Union-Find Data Structure

**Summary** In this chapter we briefly introduce the disjoint-set data structure and discuss how it can be used in Kruskal's algorithm to compute minimum spanning trees.

The disjoint-set data structure is a data structure that tracks partition of a set of elements. It provides near-constant-time operations to

- add new set,
- merge existing sets, and to
- determine whether elements are in the same set.

Since this data structure is often used in the context of graphs, the disjoint unions of the partition are often called connected components or simply components. We then say two elements are connected if they belong to the same component.

Each element in the data structure stores an id, a parent pointer and possibly a size or rank value (this is used in certain efficient algorithms). The parent pointers are arranged to form one or more trees, each representing a set. If an element's parent pointer points nowhere, then the element is the root of a tree and becomes the representative member of its set. These *forests* can be represented compactly in memory as arrays in which parents are indicated by their array index.

To make a new set from an element $x$ a new unique id is created and $x$ is added to the disjoint-set tree. Its parent pointer is set to itself, indicating that it is the representative member of its own set.

The find operation returns the root of the set in which an element $x$ is contained (given it is in one of the sets). To find this root, we simply follow the chain of parent pointers until we find a pointer that references itself. Often, to speed up consecutive calls to find, the parent of $x$ is set to the parent that was computed by find (this does not only speed up the find operation for $x$ but also for elements that reference $x$).

To merge the two sets to which two elements $x$ and $y$ belong to, we first use the find operation to obtain their respective roots (if they are the same, we are done because then the elements already belong to the same set). Then we attach one root to the root of the other using the parent pointers.

### Kruskal's Algorithm

The union-find data structure can be used to efficiently implement Kruskal's algorithm which computes the minimum spanning tree in a graph. It is a greedy algorithm which repeatedly finds

an edge of the least possible weight that connects any two trees in the current forest. In the beginning the forest consists of one tree for every vertex that only contains that vertex.

**Algorithm A.1:** Kruskal's algorithm

```
Result: Minimum spanning tree of graph G with vertices G.V and edges G.E
A ← ∅;
foreach vertex v in G.V do
    MakeSet(v);
end
foreach edge (u, v) in G.E sorted by increasing weight do
    if FindSet(u) ≠ FindSet(u) then
        A ← A ∪ {(u, v)};
        Union(FindSet(u),FindSet(v));
    end
end
return A
```