Exam notes

Advanced Robotic Perception

Victor Risager

January 19, 2024

Contents

Pool 1

1. Harris Corner, HOG, SIFT
2. Linear regression, RANSAC
3. PCA, LDA
4. Background subtraction, image differencing
5. Optical flow
6. Tracking - Kalman filter, mean-shift
7. Multi-Object tracking
8. Depth sensing
9. Camera calibration, hand-eye calibration

Pool 2

1. Clustering algorithms
2. Classification algorithms
3. CNN: Convolutions, pooling
4. CNN: Training and fine-tuning
5. CNN: Object detection
6. CNN: Semantic and instance segmentation
7. CNN: Pose estimation
8. Point cloud processing
9. Deep learning for point clouds

Short recap/Edge filters

Types of classic features of e.g. BLOB's:

- Area
- Height/width ratio
- Circularity
- Holes
- Perimeter
- Bounding box
- ...

What are features used for?

- Object detection / recognition
- Stitching, matching or aligning images
- Tracking from frame to frame in motion analysis
- Model construction
- Stereo vision
- ...

Image correlation

The process of moving a kernel across an image, and using that to extract or enhance certain features of an image. This vcould for example include:

- Blurring or smoothing with e.g. a gaussian filter
- Edge detection using e.g. sobel or canny kernels
- Template matching, where a template (kernel) is correlated accross an image.

The formula for correlation is:

$$g(x,y) = h \circ f(x,y) = \sum_{-n}^{n}\sum_{-m}^{m} h(m,n)f(x+m, y+n) \tag{1}$$

where m is the radius width of the kernel, and n is the radius height of the kernel. h is the filter, and $f(x,y)$ is the input, and $g(x,y)$ is the output.

Image convolution

The concept of convoluting an image is often used interchangably in the context of image processing. Convolution is done by flipping the array/kernel and then gliding it across the input image $f(x,y)$. Note, **when the filter is symmetric $\rightarrow$ correlation = convolution** The equation for convolution of a 1D array is given as:

$$g(x) = h * f(x) = \sum_{-n}^{n} h(n)f(x-n) \tag{2}$$

The assumed equation for the 2D image is given as

$$g(x,y) = h * f(x,y) = \sum_{-n}^{n}\sum_{-m}^{m} h(m,n)f(x-m, y-m) \tag{3}$$

Edge detection

Edges constitue rapid changes in pixel intensity, thus the gradient between two pixels is high. It usually requires noise reduction, because else there may be great edges in between pixels which may not have edges. The process is usually:

1. Noise reduction
2. Edge extraction
3. Edge refinement.

Usually done with Sobel, or Canny.
It could also be done with using 2.nd derivatives, as they cross 0 at extrema points of the derivative points.

Laplacian

The laplacian is a edge detection filter that uses double partial derivatives to conduct edge detection. There is both negative and positive laplacian. 2'nd order partial derivatives are very sensitive to noise, thus before using the Laplacian, the image should be smoothed first, by using a gaussian filter. This combination is called LoG (Laplacian of Gaussian).

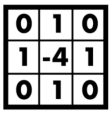

Figure 1: Laplacian kernel

Canny Edge detector

This edge detector is a sequence of steps:

1. Smooth the image with gaussian filter
2. Calculate gradient direction and magnitude
3. Non-maximum suppression perpendicular to edge
4. Threshold into *strong, weak, no edge*
5. Connect components

Here are the sobel kernels:

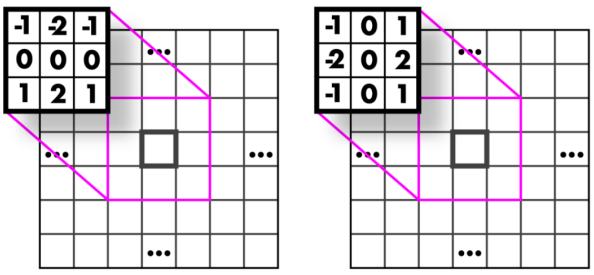

Figure 2: Horizontal and vertical sobel kernels

Compute gradients Compute gradients which are perpendicular to edges, and then look in that direction, but keep only the highest pixel. The goal is to get single pixel edges and not blurry and thick lines.
Connect components This step involves connecting the classes of edges.

- Strong edges are edges!

- Weak edges are edges **if they connect to strong edges**
-
-
-
-

1 Pool

1.1 Harris Corner, HOG, SIFT

Features are good and still relevant. They are used in recognition, detection, image stitching and visual odometry and SLAM. They do not need training, as they are extracted with simple algorithms. the main components of features are:

1. Detection - Detect the features, and extract their information and position.
2. Description - The feature descriptor is what describes the feature. This should be as unique as possible.
3. Matching - E.g. determining corresponding between two describtors in two views/images.

Examples of features:

- Sky - Bad (little variation → easily mistaken to be somewhere else)
- Edges - mid (provide uniqueness in one direction)
- Corners - Best (provide positional uniqueness across entire image. Very few potential matches.)

You could compare every feature to every other feature, but this scales horibly, On^2
To get a measure of uniqueness, you could do self difference. Compute the self of the image with a template of the feature, and do template matching with distances.

$$\sum_{d} \sum_{x,y} (I(x,y) - I(x+dx, y+dy))^2 \tag{4}$$

Which computes the distance between the patch and other pixels in the kernel for the entire image.

1.1.1 Gradient analysis

To analyse gradients, eigenvectors and eigenvalues can be used for analysing gradients around a pixel p.

$$S_w[p] = \begin{bmatrix} \sum_r w[r](I_x[p-r])^2 & \sum_r w[r]I_x[p-r]I_y[p-r] \\ \sum_r w[r]I_x[p-r]I_y[p-r] & \sum_r w[r](I_y[p-r])^2 \end{bmatrix} \tag{5}$$

Where r is the radius from pixel p that determines the weight of the gradients I_x and I_y round the pixel. The eigenvalues of (5) summarize the distribution of gradients of the pixel p. if

- λ_1 and λ_2 are both small → then there is no gradient.
- $\lambda_1 >> \lambda_2$ or opposite, gradient in one direction.
- λ_1 and λ_2 are both large, multiple gradient directions = corner

Eigenvalue computation is expensive however, therefore does the Harris Corner apporoximate the eigenvalues.

1.1.2 Harris corner

The algorithm:

1. Calculate derivatives I_x and I_y.
2. Calculate products I_xI_x, I_yI_y, I_xI_y.
3. Compute the weighted sums based on (5) .
4. Estimate response based on smallest eigenvalue.
5. Non-maximum suppression (like Canny-edge).

It IS rotation invariant → eigenvalues are the same.

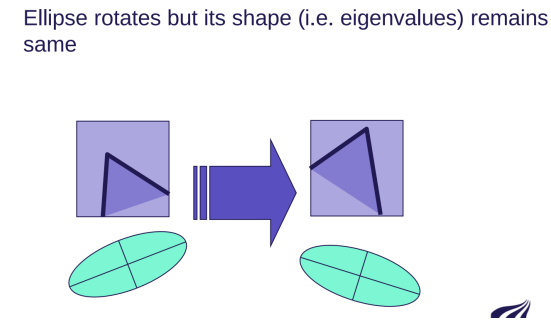

Figure 3: Harris corner rotation invariance

It is not Intensity invariant → because of fixed threshold on non maximum suppression.
It is not Scale invariant → The eigenvalues scale with the window size used for gradient computation, and because the gradients magnitude affects the eigenvalues.

1.1.3 HOG - Histogram of Oriented Gradients

Objects can be described by the distribution of local intensity gradients. Originally proposed as a pedestrian detector. It is computed in windows which are slid across the entire image.

1. Normalize color
   Make every color between 0 and 1
2. Compute gradients
   Use sobel kernels $\begin{bmatrix}-1 & 0 & 1\end{bmatrix}$ and $\begin{bmatrix}-1 & 0 & 1\end{bmatrix}^T$
3. Weighted vote into spatial orientation cells
4. Contrast normalize over overlapping spatial blocks We make blocks that span multiple cells. Compute the L2-norm and divide each gradient with that.
5. Collect HOG's over detection window
   Concatenate all gradient bins into one long feature vector for a single window.
6. *Run linear SVM classifier*
   Could be used to classify e.g. if it is a person.

As seen in Figure (4), the longer arrows in a cell (usually 8×8) means more predominant edge in that particular direction. Then we assemble the gradients in histograms with angle along the horizontal axis and number of edges at that angle along the vertical axis.

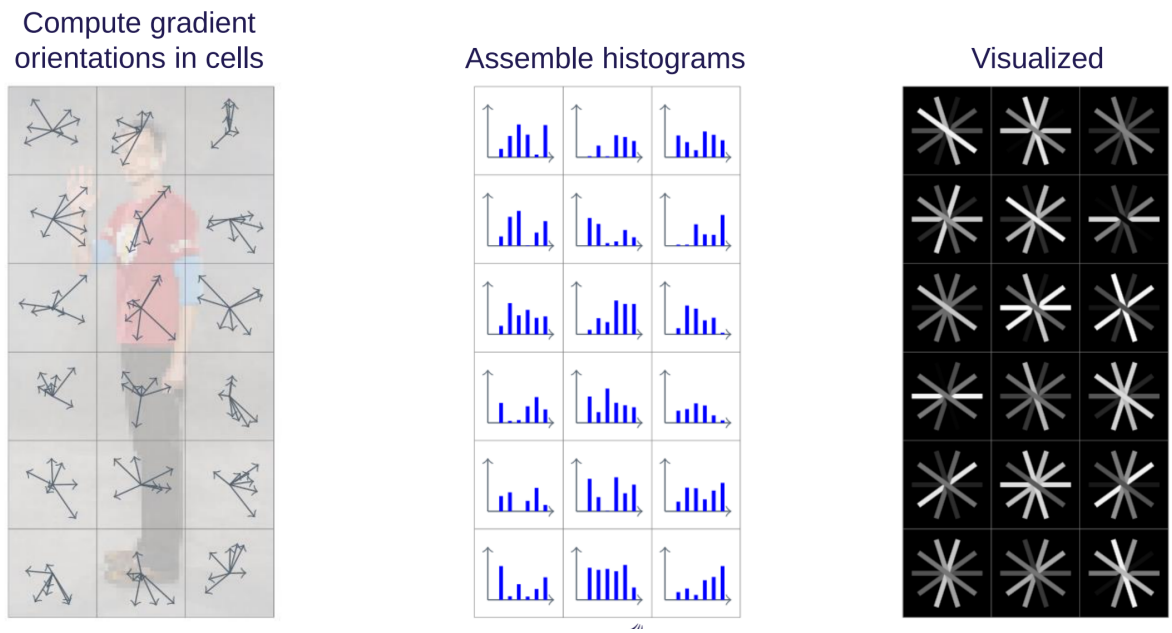

Figure 4: HOG features

Lighting invariant
Scale invariant
Not rotation invariant

1.1.4 SIFT - Scale Invariant Feature Transform

We try to compute scale invariant features.

- Scale-space extrema detection
- Keypoint localisation
- Orientation assignment
- Keypoint descriptor

SIFT focusses on both *detection* and *description*
The image is asigned different scales. These are assigned into what is know as *octaves*. In each octave we convolve gaussians with different σ-values, which affects the degree of blurring. Two adjecent gaussians are subtracted, which gives the *DoG*. This results in images that containes edges, and works like a low and high pass filter depending on the σ-value. For large σ-values, the large changes (low frequencies) are extracted, and vice versa.

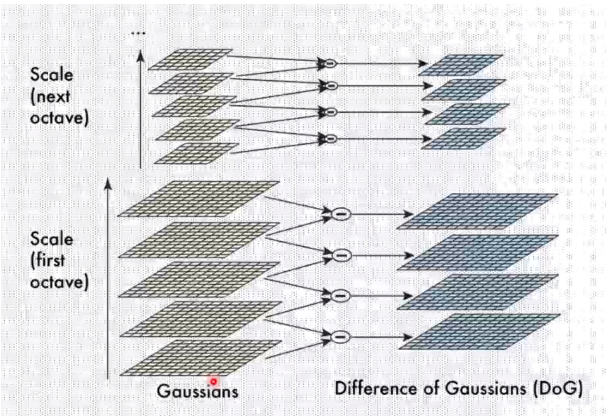

Figure 5: SIFT extrema detection

In this step we determine if the keypoint is an extrama, by considering the scale above and the scale below. If these are all above or below the pixel, then we have an extrema.

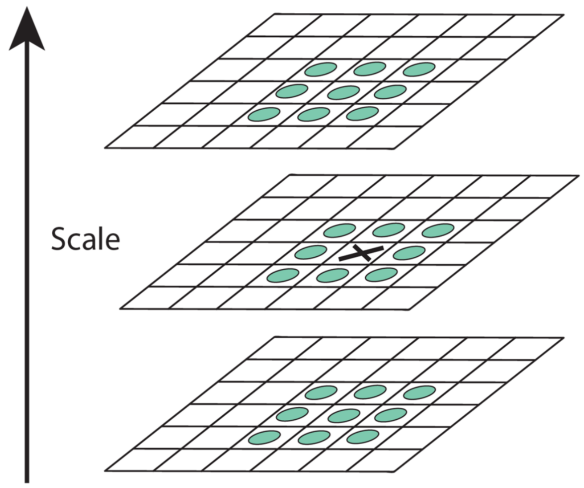

Figure 6: The local extrema location across adjecent scales.

In this step, we assign orientation based on the patches in the image. We assign 36 orientation histogram bins, and these are weighted by the gradient magnitude and gaussian-weighted distance from the keypoint. We assign the most dominant direction as the orientation of the keypoint, and we also assign keypoint descriptors for bins above 80%.

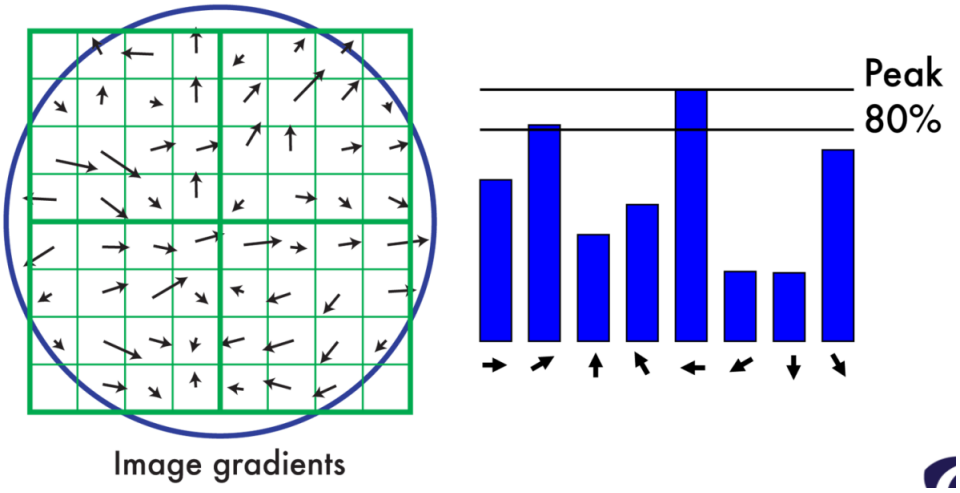

Figure 7: Patch orientation for SIFT keypoints

We rotate the the feature relative to the keypoint orientation. This makes it rotation invariant. The feature descriptor must be illumination invariant as well, this is done by normalizing the feature descriptor vector to unit length, thresholding the descriptor above 0.2, so we disregard any light intensities below 0.2 and we normalize once again.

Figure 8: The SIFT algorithm visualized

Problems:

- Computationally heavy → slow in some scenarios

Alternatives

- SURF
- ORB (fast for realtime.)

1.2 Linear regression, RANSAC

1.2.1 RANSAC - RANdom Sample Consensus

E.g. a way of matching two images, where we consider inliers and outliers. If there is enough inliers, we have can make an image transformation, because there may be a lot of feature matches agree on some image transformation, then we may use this image transformation.
Algorithm:

1. Select a random subset of the of original data
2. A model is fitted to this subset (e.g. Linear regression)
3. Determine how many points fit within a predefined tolerance (inliers vs outliers)
4. If the fraction of the total number of inliers relative to the number of data points in the subset are above some threshold, we refit the model to the set of inliers, and then we terminate.
5. Otherwise repeat step $1-4$ (max N times)

We select e.g. 2 points and fit a model to these two points $\rightarrow$ A single line between the two points. Then we compute the number of inliers of that model based on the predefined tolerance threshold. If we see enough inliers, then terminate.

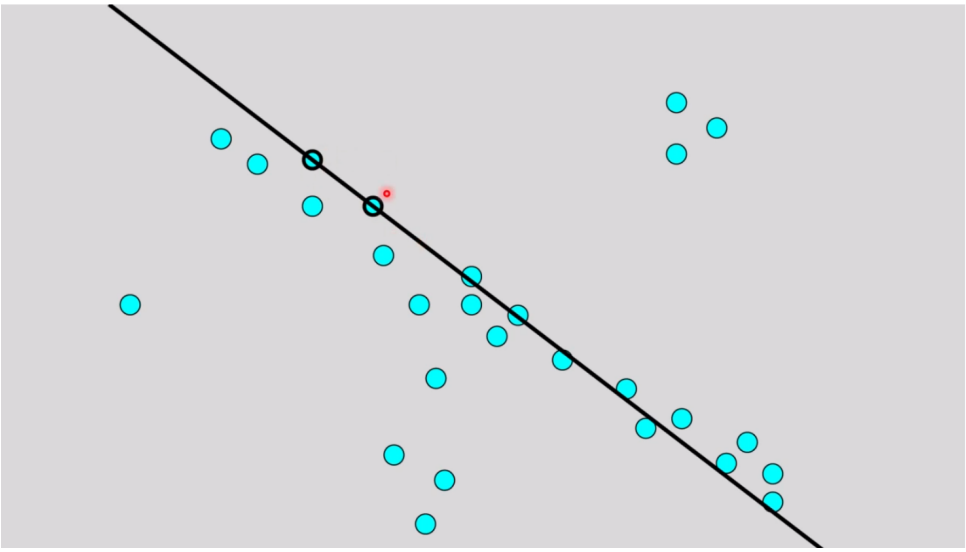

Figure 9: Illustration of the RANSAC algorithm

It has a lot of different hyper parameters.

- t threshold from the model $\rightarrow$ quite small
- n number of points to fit the model $\rightarrow$ 2 for a line
- k is the number of itterations $\rightarrow$ can be high
- d good fit cutoff. How many inliers are considered engough to consider a good model.

Note may need to elaborate on linear regression

1.3 PCA-LDA - Principal Component Analysis, Linear Discriminant Analysis

Dimensionality reduction. We have some featurevector with a high dimensionality of features, and the goal is to reduce the dimensionality of the feature vector, while still preserving the variance of the data, which says something about the information of the data.

1.3.1 PCA

Algorithm

- **Standardize** the data (compute mean and subtract, to center around 0)
- Compute **covariance** matrix of standardized data.
- Compute **eigenvectors** and corresponding **eigenvalues** of the covariance matrix

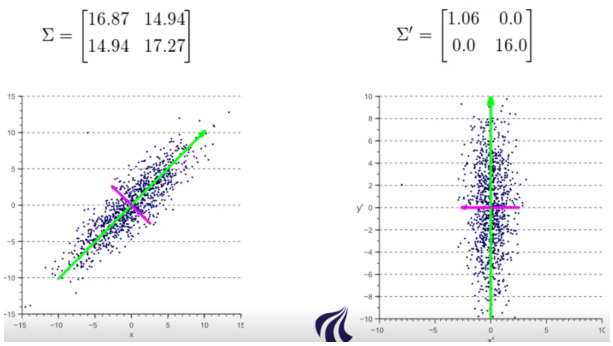

Figure 10: Covariance computation and Transformation.

- **Transform** the data using the n most significant variances.

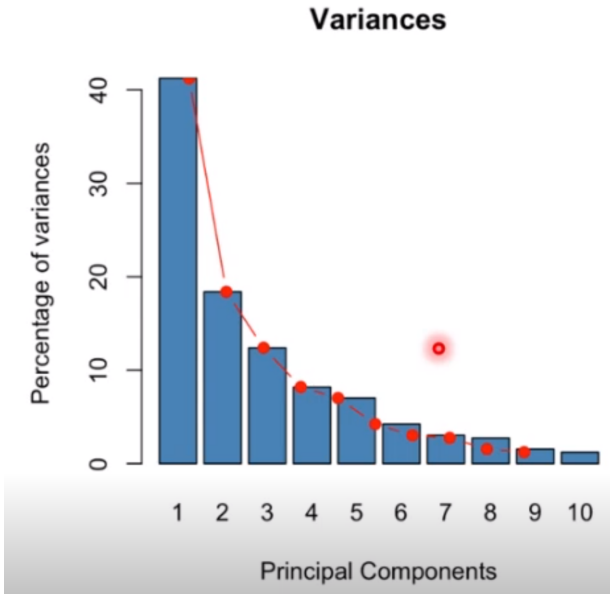

Figure 11: PCA percentage of variances

Problems:

- Unsupervised → no knowledge of the classes of the data

1.3.2 LDA

Uses labelled data. Finds the direction that best discriminates between classes. I.e. Best way of splitting the classes up for easy classification.
Also known as Fishers Linear Discriminant. (small difference in assumptions)
Based on 2 different computations

- Within-class scatter:

$$S_w = \sum_{i=1}^{c} \sum_{x \in C_i} (x - m_i)(x - m_i)^T \tag{6}$$

- Between-class scatter:

$$S_B = \sum_{i=1}^{c} n_i (m_i - m)(m_i - m)^T \tag{7}$$

Where x is a sample of class i, c is the total number of classes and m_i is the sample mean of class i. n_i is the number of samples in class i, and m is the total sample mean.
So in (6) the difference between the sample and the mean of that class is comuted and summed. Basicly the same as computing variance.
In (7) the computed mean of each class vs the total mean is computed. This tells us the magnitude of the variance of all the classes, i.e. how far appart these classes are.
We then want to find the transformation that maximises the ration between the within-class scatter and between class scatter.

$$W_{opt} = argmax_w \frac{|W^T S_B W|}{|W^T S_W W|} \tag{8}$$

1.4 Background subtraction, image differencing

1.4.1 Background Subtraction

Video is a function $f(x, y, t)$. Use a time dimension to segment the video frames.
Segmentation $\rightarrow$ seperation of forground and background. (binary images.) You can do background subtraction in videoes.
Basic algorithm

1. Capture image containing the stable background.
2. Capture a new frame sometime in the future and compare them.
3. Subtract frame from background (use absolute difference to avoid negative values.)
4. Threshold difference (there may be small differences due to noise)
5. Delete the noise (use median filter, bigger kernels remove larger spots, or use Morphology(opening/closing))

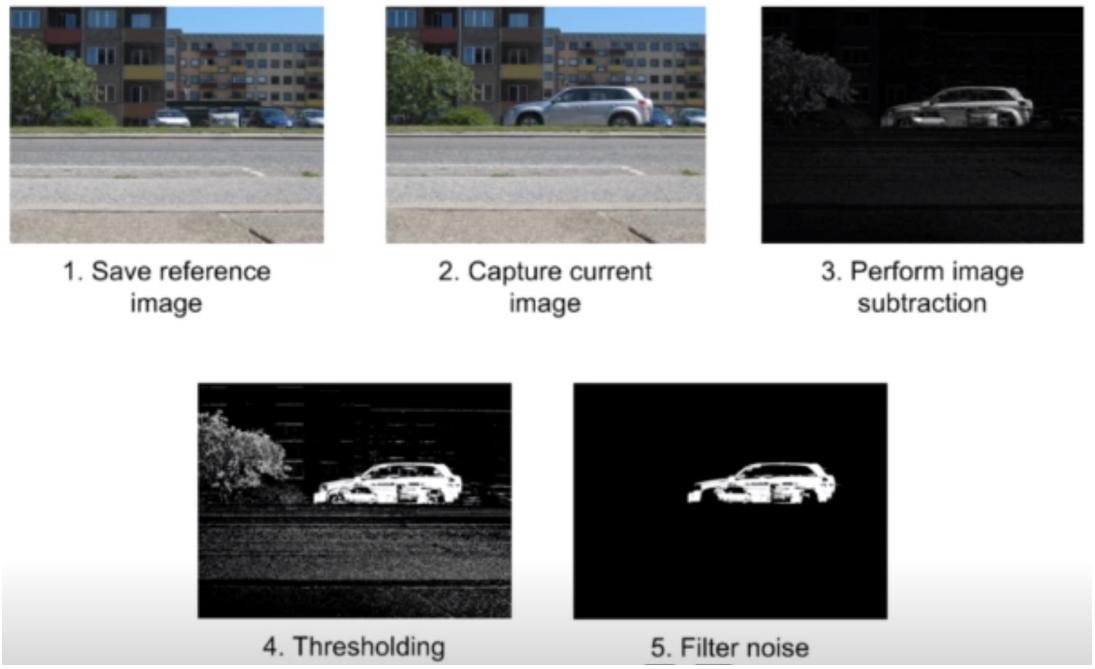

Figure 12: Basic background subtraction algorithm

Requires **completely static background**
Advanced background subtraction Almost static background (noise, light changes, small motion (wind blowing in trees))
Compute **average background image**.

$$B(x, y) = \sum_{i=1}^{N} \frac{I_i(x, y)}{N} \tag{9}$$

where I_i is the i'th image and N is the total number of images.
Threshold should be different for different pixels in the image, as they may vary in variance.
We can learn these thresholds:

Gaussian models (mean and variance) for each pixel.

$$TH = K \cdot \sigma(x, y) \tag{10}$$

You could also **Update the reference image.**

1.4.2 Image differencing

Very dynamic scenes → Background cannot be learned.
Use any previous image as the reference image. depending on how close to the captured frame you want.
Subract them and look at what moved.
You can look at two different reference frames and **AND** them together.

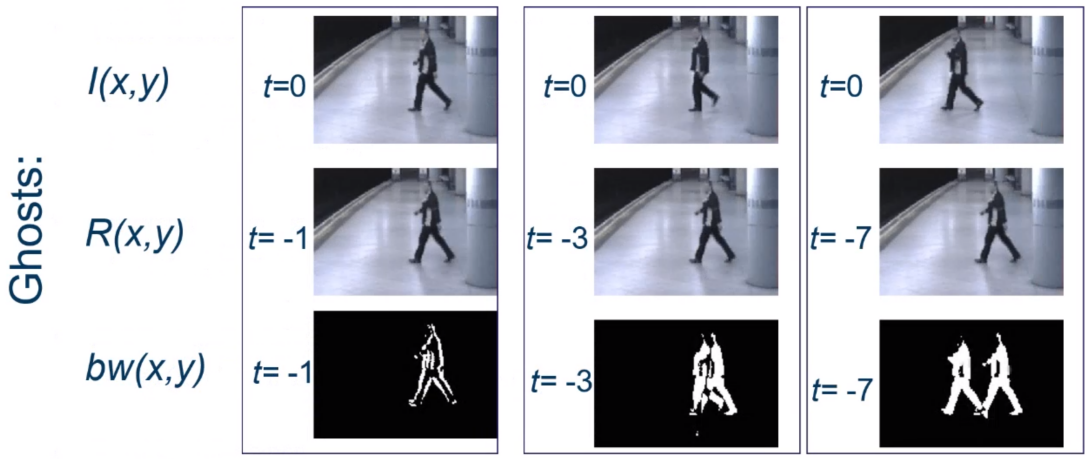

Figure 13: Image differencing at 1. 3 and 7 frame differences.

Note that if the time difference is great, it may be difficult to do data association between two BLOB's.
This is considered **Ghosting**

Image differencing	Background subtraction
• Handles dynamic scenes	• Requires static background
• Detects only motion	• Detects stationary objects
• Detection results in ghosts	• Detects whole objects

Figure 14: Background subtraction and image differencing comparison.

1.5 Optical Flow

Motion analysis.
Can detect both object motion and camera motion.
Can be used in **Visual Odometry**.
We can detect type of motion by looking at the difference vectors and their magnitudes.

- Rotation
- Horizontal translation
- Larger vectors appear to move faster, or are closer to the camera.

We can **not** just use **feature matching** because:

- Sparse (needs good features)
- Feature alignment may not be exact.
- Low accuracy for entire image.
- Assumes that everything is moving with the same transformation!

1.5.1 Lucas-Kanade Algorithm

Based on the idea that when the object is moving in one direction, it is the same as moving the observed pixel in the opposite direction.

$$f(p - \Delta p, t) = f(p, t + \Delta t) \tag{11}$$

where $p = \begin{bmatrix} x \\ y \end{bmatrix}$ is the observation point. t is the first image and $t + \Delta t$ is the second image.
Approximate the model with taylor expansion and do some deriviation.

$$-dx \cdot u - dy \cdot v \approx I_t[x, y] - I_{t+\Delta t}[x, y] \tag{12}$$

where $u = \Delta x$ and $v = \Delta y$ and I_t is the image at time t Solve for u and v. $\rightarrow$ 1 equation, 2 unknowns.
Therfore we look at the pixels in the **neighborhood** $\rightarrow$ We assume that the pixels in the neighborhood **move together**
We express the (12) for all neighbors, and concatenate them into vectors.
let

$$S = \begin{bmatrix} dx_1 & dy_1 \\ dx_2 & dy_2 \\ \vdots & \vdots \\ dx_9 & dy_9 \end{bmatrix} \tag{13}$$

$$\Delta p = \begin{bmatrix} u \\ v \end{bmatrix} \tag{14}$$

$$T = \begin{bmatrix} I_t[x_1, y_1] - I_{t+\Delta t}[x_1, y_1] \\ I_t[x_2, y_2] - I_{t+\Delta t}[x_2, y_2] \\ \vdots \\ I_t[x_9, y_9] - I_{t+\Delta t}[x_9, y_9] \end{bmatrix} \tag{15}$$

Then the solution expression is given as:

$$S\Delta p = T \tag{16}$$

Which is solved with least squares giving:

$$\Delta p = (S^T S)^{-1} S^T T \tag{17}$$

$S^T S$ must be invertible. $\rightarrow$ Not invertible when there is no structure around x and y. E.g. if the middle of the sky is selected.
Good selections are corners.

Problems:

- Not lighting invariant
- Large movement → Requires high framerate
- Needs good feature.
- Appature problem

Appature problem Because of looking only in the neighborhood, it may not be possible to see the actual motion.

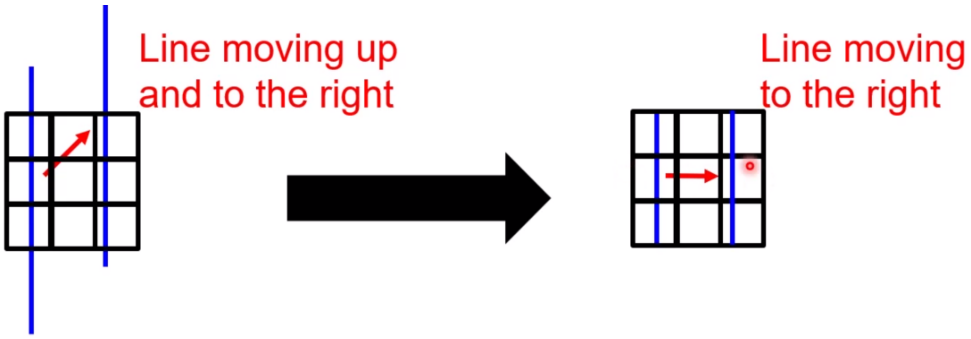

Figure 15: Appature problem

Same with the **barber pole illusion**.

1.6 Tracking - Kalman filter, mean shift

Using temporal information in object tracking.
One way is doing:
Predict → **Match** → **Update** This can take in no linear motion, or 1'st order linear motion → velocity to predict next position. Or lastly 2'nd order linear motion → Acceleration, and predicted acceleration at next time step.

- 0 order linear motion → same position
- 1'st order linear motion → use velocity to predict next position
- 2'nd order linear motion → Use acceleration to predict next position.

The matching state is using a **ROI** dependent on uncertainty of where the next position is. We can **scale the ROI with uncertainty** α

$$radius(t+1) = \alpha \cdot (p(t) - p(t)) + (1-\alpha) \cdot radius(t) \tag{18}$$

Can track based on **appearance** or **motion model**

1.6.1 Kalman filter

Set up a state vector containing fx:

- Position
- Velocity
- Acceleration
- Size
- Shape
- Colour
- $\vdots$

Prediction

1. Predicting the next state:
$$\hat{x}_k = A\hat{x}_{k-1} + Bu_k + w_k \tag{19}$$
2. Error covariance propagation.
$$P_k = AP_{k-1}A^T + Q \tag{20}$$

Measurement

$$z_k = Hx_k + v_k \tag{21}$$

where H is the observation model → maps from state-space to measurement space
w_k and v_k are random gaussian noise random variables.

1. Compute kalman gain:
$$K_k = P_k H^T (HP_k H^T + R)^{-1} \tag{22}$$
2. Update estimate:
$$\hat{x}_k = \hat{x}_k + K_k(z_k - H\hat{x}_k) \tag{23}$$
3. Update error covariance:
$$P_k = (I - K_k H)P_k \tag{24}$$

Assumes

- Gaussian noise
- Linear motion model

Non-linear → EKF

1.6.2 Mean shift tracking

Tracking **non-rigid objects** like walking person. → Appearance represented as histograms (gradient, color, etc.)
Shift the center of **ROI** towards the mean of data

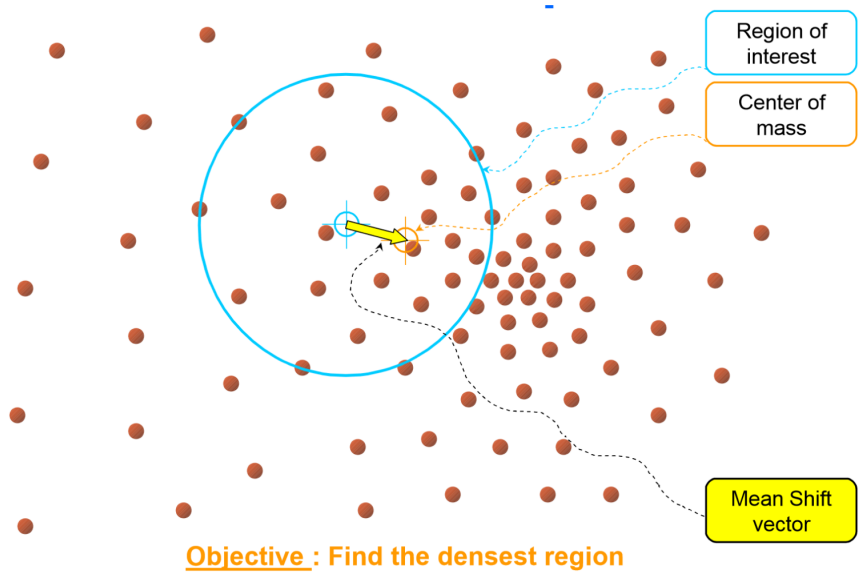

Figure 16: Shifting the center of the ROI towards the mean of the center of mass

The framework of the mean shift tracking is based on choosing some position of the object in the current frame, and searching for it in the neighborhood of it, in the next frame. Maximise the similarity function which can be based on **Color likelihood (best for single color objects) or represent color distribution with histograms and find the most similar histogram.**

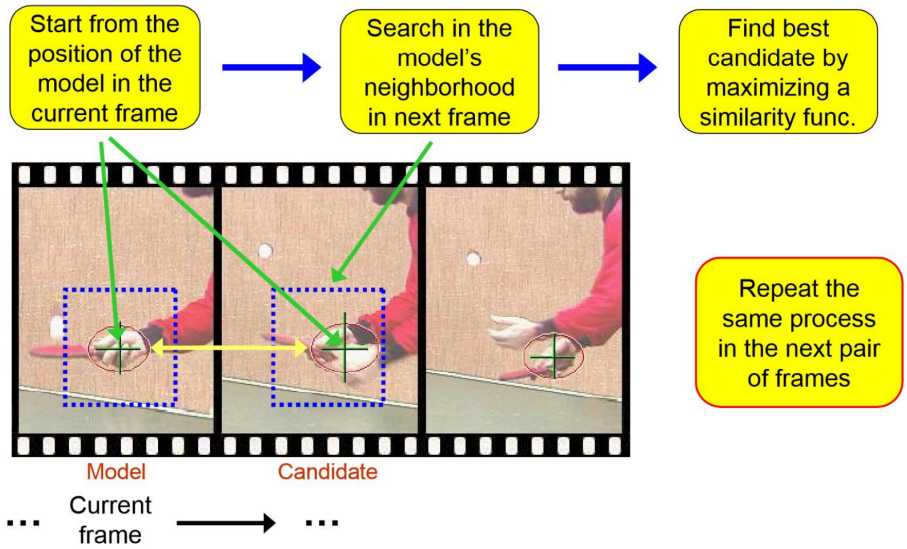

Figure 17: Framework of mean shift

The 1'st method:

- Requires good segmentation

- Or we compute a likelihood of pixels (what is the probability of this pixel being in the object we are tracking.)

The 2'nd method

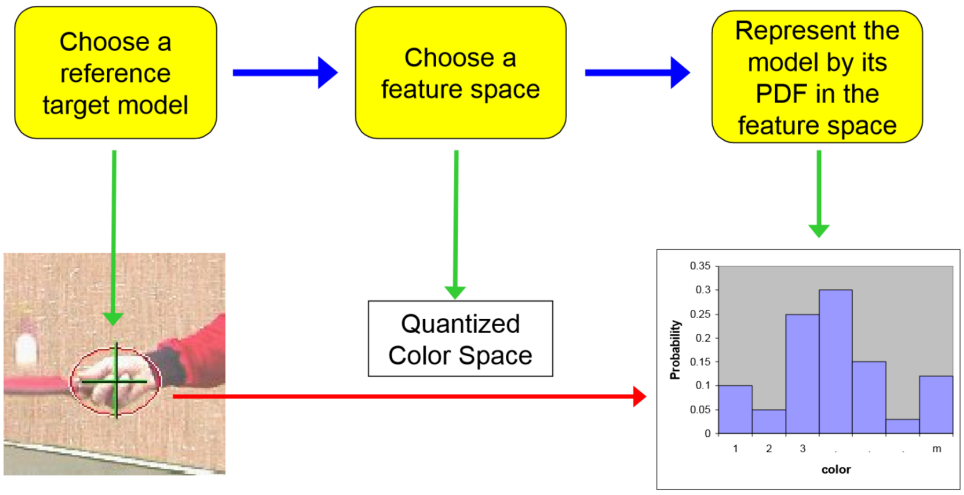

Figure 18: Choosing the featurespace and representing the likelihood of the model being the best match based as a PDF.

1.7 Multi Object tracking

We can just run several independent trackers but there are some problems that must be adressed:
Problems:

- Similar / identical objects
- Splits
- Merge/ intersections
- Occlusions
- Noise

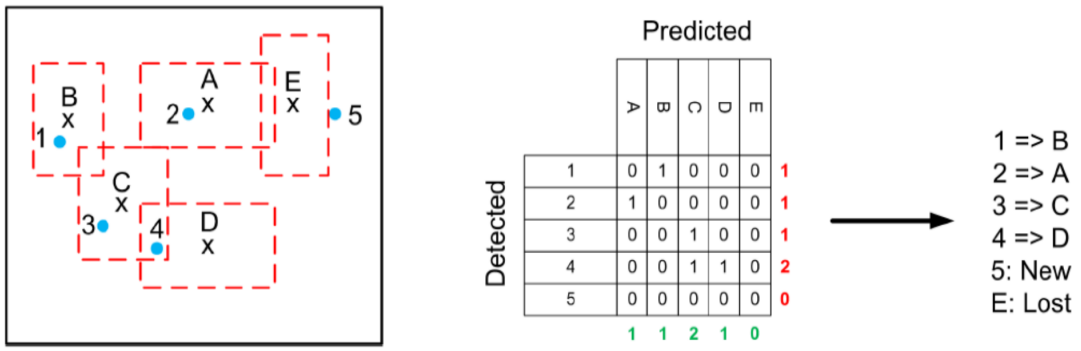

Figure 19: Multi Object Tracking example illlustrating issues.

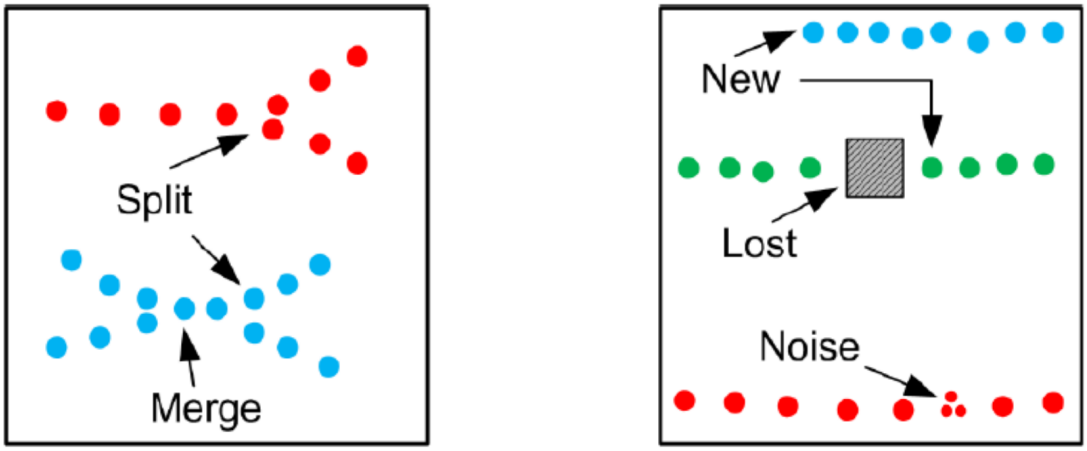

Figure 20: Multi Object Tracking issues.

The problem is **Data association**

1.7.1 Hungarian algorithm

Optimal assignment based on a given cost matrix.

- Example: Weights between three trackers and three detections

	T1	T2	T3
D1	250	400	350
D2	400	600	350
D3	200	400	250

- Cost matrix: $\begin{bmatrix} 250 & 400 & 350 \\ 400 & 600 & 350 \\ 200 & 400 & 250 \end{bmatrix}$

Figure 21: Hungarian algorithm cost matrix

1.8 Depth sensing

Goal → recover 3D coordinates from 2D images.
In 2D images, size and distance is unknown.

Figure 22: Depth illusion

Different technologies exist in distance measurement. We are working with **reflective** (transmissive would be the object them selfs transmitting data that can be measured.) Active optical distance measurement meanst that the camera itself emits some kind of radiation. Passive is when the ambient light is used and measured.

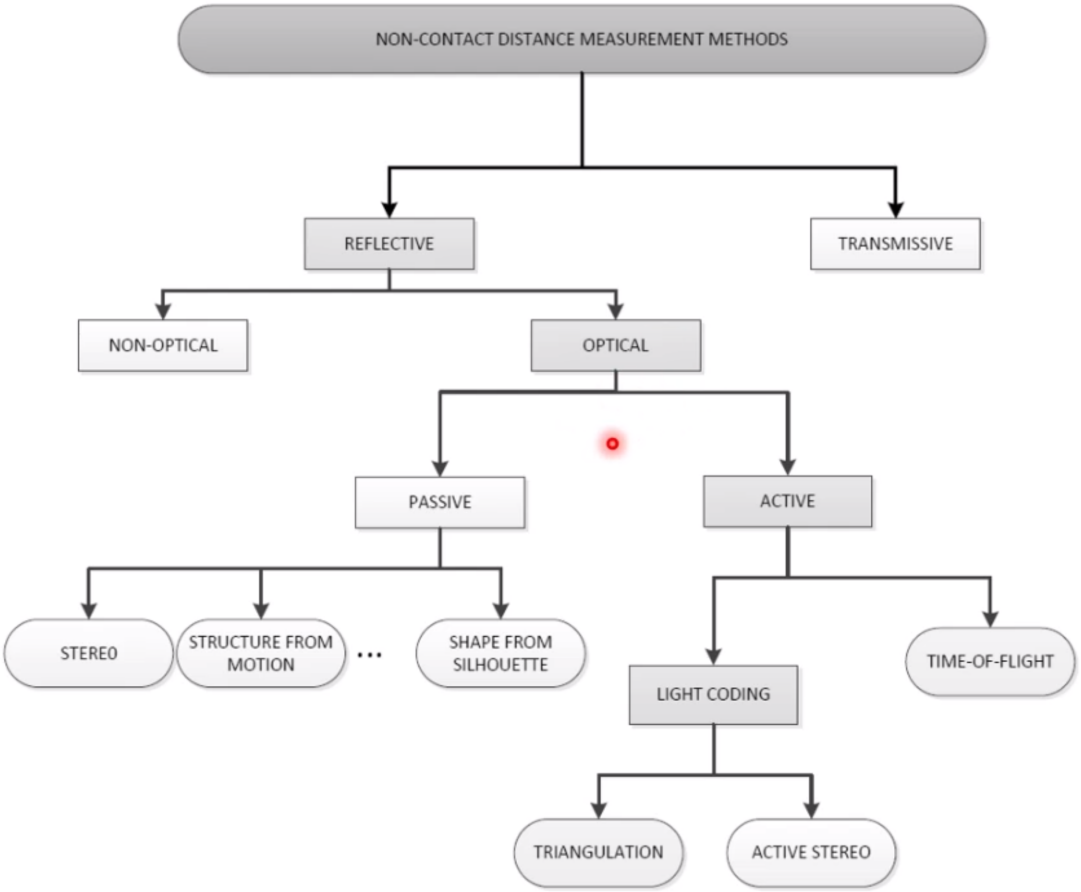

Figure 23: Types of distance measurements

1.8.1 Two view geometry

Aquire image pair

1. Stero rig (two cameras)
2. Move one camera and take two fotos.

A keypoint can be triangulated but it takes two steps:

1. Find the keypoint from the 1'st image in the 2'nd image. **Search problem**
2. Estimate the 3D point by triangulation. **Estimation problem**

We can reduce the search problem from 2D to 1D using **Epipolar constraints**

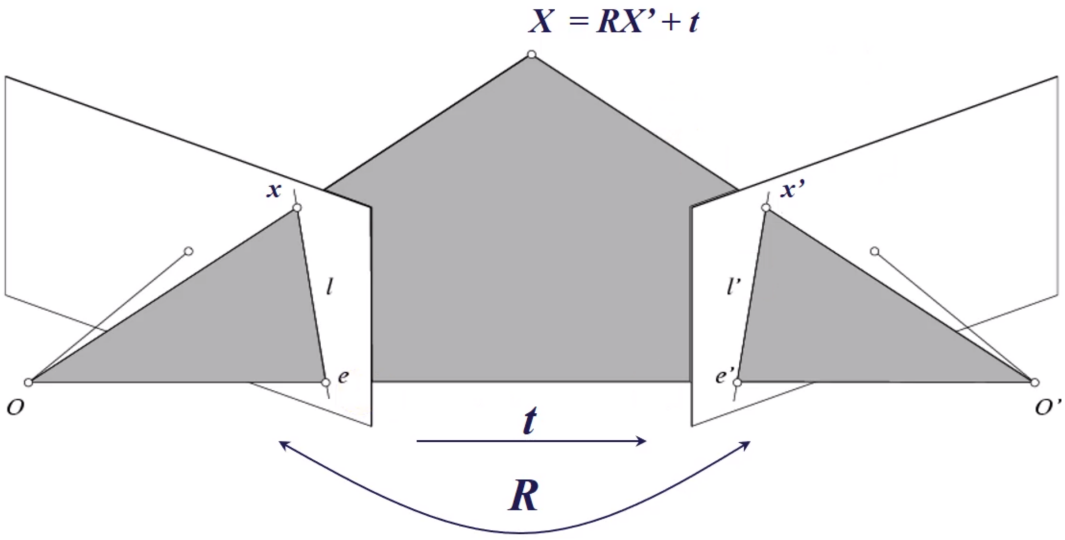

Figure 24: Epipolar geometry

- **Baseline:** line between the two cameras
- **Epipolar plane:** plane between the two point camera's. Contains the baseline. Rotates around the baseline
- **Epipoles:** intersections of baseline with image planes. (may be outside of the frame)
- **Epipolar Lines:** Intersection between the epipolar plane and the epipolar image planes.

Epipoles are at infinity when the two image planes are parallel. But when we have **Converging** cameras, then the epipoles exist.
Assumes we know extrinsic and intrinsic parameters (Camera is calibrated).
We compute the **Essential matrix**

$$E = [t_\times]R \tag{25}$$

$$[t_\times] = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix} \tag{26}$$

Where $[t_\times]$ is a decomposition of the translation vector.
Using the essential matrix we get an equation for the epipolar line for point x' in the other image.

$$l = Ex' \tag{27}$$

We use the **Fundamental** matrix F if the camera is **uncalibrated**

1.8.2 Depth sensing with two view geometry

Using a calibrated camera pair (know both extrinsic and intrinsic)

1. Calibrate camera (offline)
2. Aquire image pair (online →)
3. Rectify images (Align points along the same horizontal scanline.)
4. Compute horizontal disparity (unit = pixels)
   - Dense (for every pixel) → scan along the scanline and do e.g. Sum of squared differnces or normalized correlation and find the best match.
   - Sparse (for keypoints)
5. Triangulate to optain 3D coordinate. (unit = meters)

Requires parallel camera planes. For converging cameras, we can project the image planes onto a common plane. (Stereo image rectification)

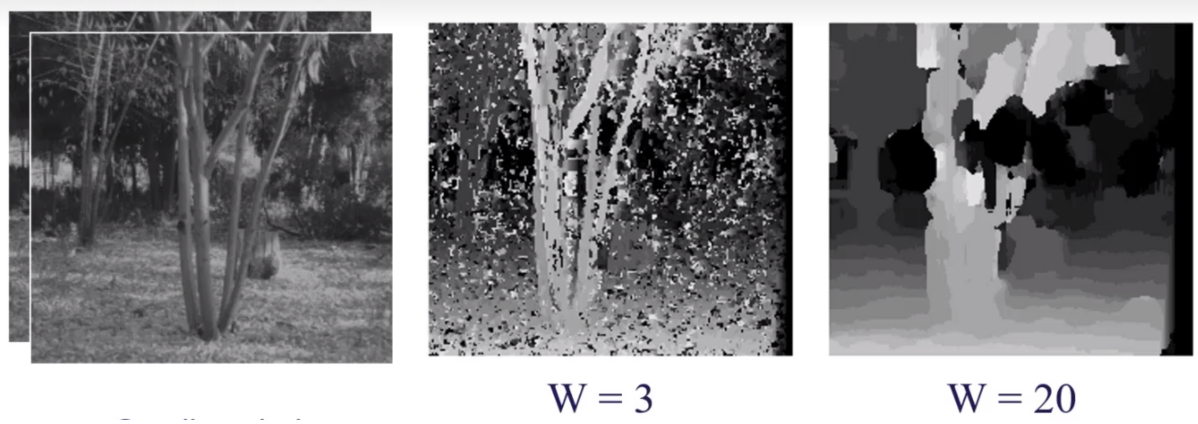

Figure 25: The efffect of window size when doing correspondence in horizontal disparity map

- Large window → less noise and less detail
- Small window → more noise and more detail

The problems of correspondence search **Problems:**

- Occlusions
- No texture on surfaces → sacrified uniquness of non-local constraints.
- There may be repititions of the same object in the image. (Bars in a fence)
- Light may reflect differently in both images. (The images are not similar)

Depth from disparity

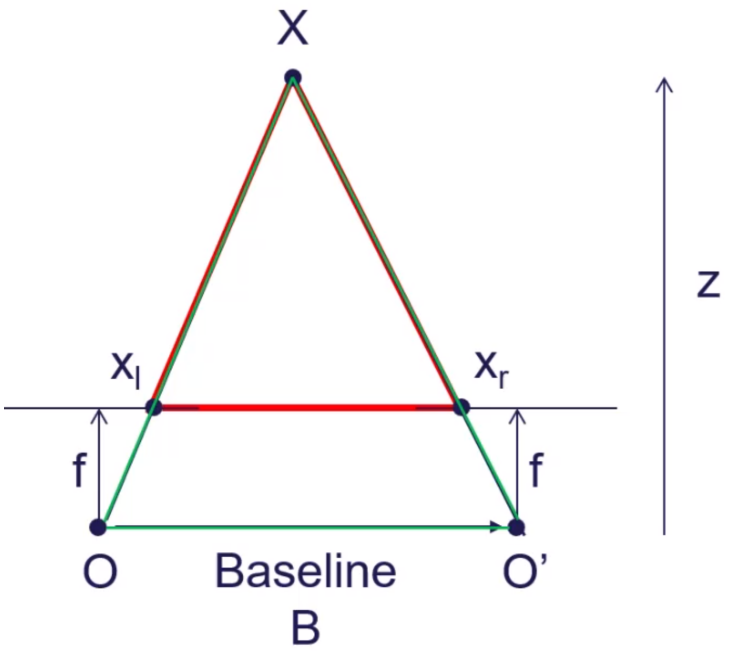

Figure 26: Depth from disparity

Based on equal triangles, we can compute the ratio, and then get the detph coordinate z

$$z = \frac{fB}{x_r - x_l} \tag{28}$$

$x_r - x_l$ is the disparity.
It results in discretisisation where the steps gets larger the further it goes, the lower the resolution of the depth estimate.

1.8.3 Active depth sensors

Structured Light

- Projects light patterns onto the object
- Can estimate depth from **textureless** objects

Time of flight

- Measure the reflection time of a light pulse.
- Measure the reflection based on continuously emitted light, and measure **phase shift**.
- Both require very high clock speed.

1.9 Camera calibration, Hand-eye calibration

1.9.1 Camera calibration

In the **Projective world** parallel lines may meet, and circles may be eliptic (formally known as **conic**), angles between lines may also change.
The mapping from euclidean space to projective space, is done using homogeneous coordinates.

- **Intrinsic:** estimating internal propoties of the camera that affects the imaging process.
- **Extrinsic:** Estimating the relationship between the world coordinates and the camera coordinates, and object coordinates.

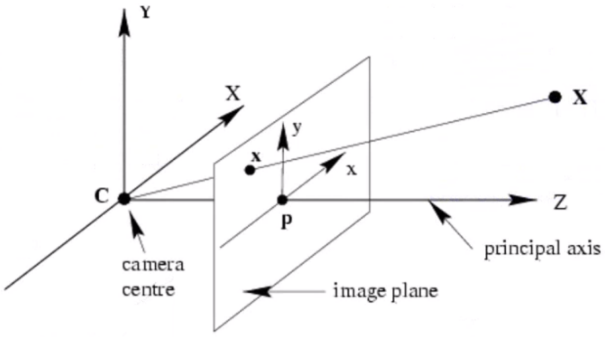

Figure 27: Pinhole camera model

For the pinhole camera model we have:

- **Principal axis:** the axis from the camera center perpendicular to the image plane.
- **Normalized (camera) coordinate system:** is the camera center at the origin. Principal axis as the z-axis.

We want to estimate the projection matrix $P \in \mathbb{R}^{4\times 3}$ s.t.

$$x = PX \tag{29}$$

where $x \in \mathbb{R}^{3\times 1}$ is the image point and $X \in \mathbb{R}^{4\times 1}$ is the scene point. So these are in homogeneous coordinates.

Homogeneous coordinates

We introduce a new variable w so 3D coordinates in euclidean space is 4D in homogeneous coordinates and 2D $\rightarrow$ 3D

$$x = \begin{bmatrix} x \\ y \end{bmatrix} \leftrightarrow \begin{bmatrix} u \\ v \\ w \end{bmatrix} = k \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{30}$$

We still only have 2 degrees of freedom, so we can extract euclidean points

$$x = \frac{u}{w} \qquad y = \frac{v}{w} \tag{31}$$

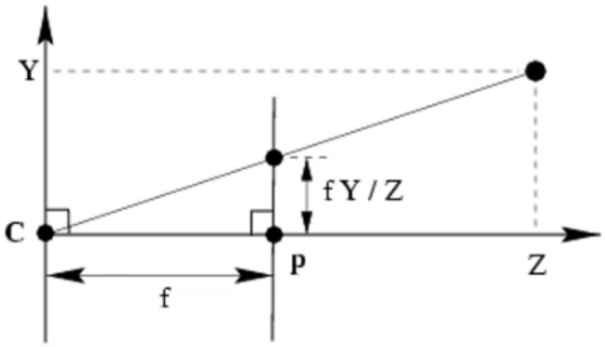

Figure 28: Pinhole sideview used in intrisic parameter estimation

From similar triangles between the scene point and the image point, we can estimate the image point.

$$x_{im} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f\frac{x_s}{z_s} \\ f\frac{y_s}{z_s} \\ 1 \end{bmatrix} \tag{32}$$

and converting to homogeneous coordinates using the projection matrix P:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} \tag{33}$$

f is the focal length.
Thus the intrinsic paramers are:

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} \alpha_x & s & x_0 & 0 \\ 0 & \alpha_y & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} \tag{34}$$

where

- α_x, α_y "focal length" measured in pixels
- x_0, y_0 coordinates of image center
- s skew parameter for non-orthogonal axis (usually 0)
- u', v', w' homogeneous image coordinates
- x_s, y_s, z_s scene points in camera coordinate system.

The extrinsic parameters are then:

$$P = \begin{bmatrix} \alpha_x & s & x_0 & 0 \\ 0 & \alpha_y & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & T \\ 0_3^T & 1 \end{bmatrix} \tag{35}$$

Which projects any coordinate from the world onto the image frame.

Figure 29: Radial distortion

Camera calibration in practice

Using a bunch of images of a checkerboard:

1. Find edges
2. Perform straight line fitting
3. Intersecting lines to find corners
4. Normalize image + space points
5. Use Linear transformation
6. Find solution + error
7. Adjust parameters and reiterate

1.9.2 Hand-eye calibration

It is important to know the mapping between the camera on the end effector and the flange. If this transformation is not known, we see failures to grasp stuff, and collisions with the environment.

Eye-on-hand

The camera is positioned on the end-effector.

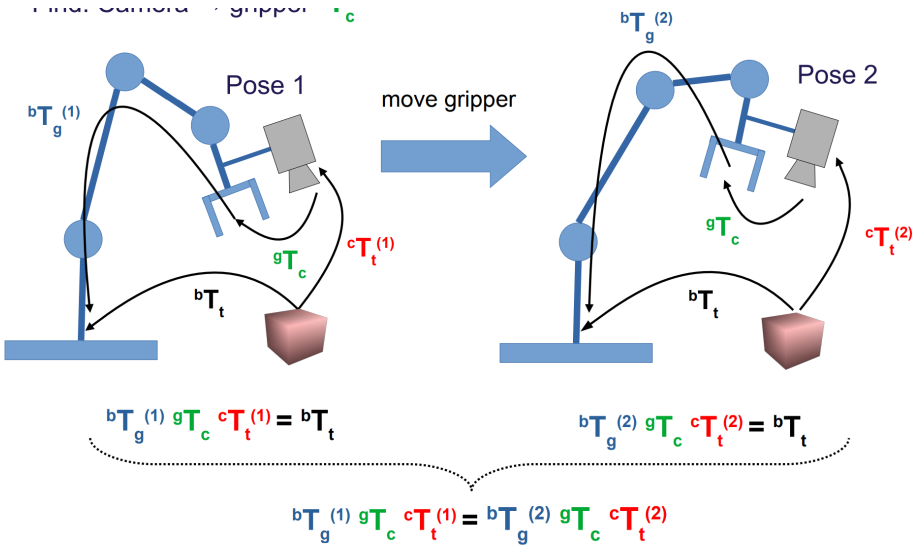

Figure 30: Eye on hand calibration

we want to find the transformation from the gripper to the camera.
Use the red transformation from the camera to the target, but this may be unknown, so we can use a checkerboard as a target. We isolate the types of transformations such that $AX = XB$

- X camera $\rightarrow$ gripper (unknown)
- A gripper $\rightarrow$ base (forward kinematics)
- B target $\rightarrow$ gripper (checkerboard)

This equation can be solved for X using the OpenCV function *calibrateHandEye()* **Important:** the checkerboard must stay fixed during calibration. It is only relative transformations, not absolute.

Eye-on-base

Camera relative to base. Mount the checkerboard on the end-effector, and the camera on somewhere else. The idea is the same.

2 Pool

2.1 Clustering algorithms

Unsupervised learning.

Note on fitting

For sparse datasets it is a good idea to use cross validation, where the model is trained on e.g. 4/5 parts of the data, and validated across the last 1/5. Then during the test phase, no additional tweaks to the hyper paramereters are permitted.

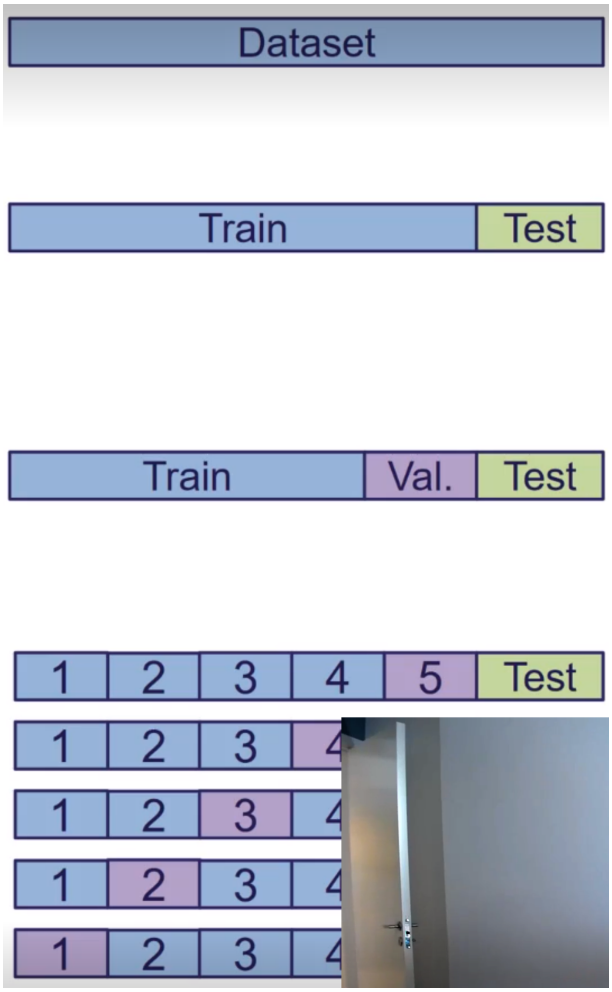

Figure 31: Splitting the dataset into test, validation and training set.

Note on Performance

- **Precision** $\rightarrow$ Measure of positive prediction values $P = \frac{TP}{TP+FP}$
- **Recall** $\rightarrow$ Measure of sensitivity $R = \frac{TP}{TP+FN}$

ROC - Reciever operating curves: A graph displaying the FP to TP ratio (The further up in the left corner, the better.)
Precision-recall curves: A curve displaying the recall (horizontal axis) compared to the precision (vertical axis)

2.1.1 K-means clustering

Assuming that points of the same group are in close proximity of each other.
Algorithm

1. Randomly **initialize** k cluster centers.

2. Compute the **distance** between points and all the centers.
3. **Assign** each point to the closest center.
4. **Update cluster centers:** average of cluster points in a group
5. **Repeat** until **convergence**

can be used on featurevectors, or pixel values directly.
Problems:

- If a cluster is confined within another cluster, it will assign the datapoints to the outer cluster. I.e. it cannot differ.
- You must choose k
- Sensitive to **outliers**
- Local minima in the *within-cluster squared error* cost function may exist.
- Detects only spherical clusters

2.1.2 Mean-shift clustering

Based on mean shift tracking
Pros: does not need predefined number of clusters.
Cons: need to choose windowsize/radius.
Start with a grid of centers, and then let run the algorithm and let the grid points converge towards the center of mass in each iteration. Lastly overlaps are removed, such that there is only the corrent number of clusters left.

2.1.3 DBSCAN - Density-based Spatial Clustering of Applications with Noise

Works by connecting close points within some distance from each other. It also focusses on defining core points with the most neighbors. They are chosen based on a threshold, such that they must have mroe than minPts neighbors to be considered a core point. Neighboring core points along with their respecitve non core points are added to the the cluster.
Algorithm:

1. Compute neighbors within ϵ distance of each point and identify core points with more than minPts neighbors
2. Join neighboring core points into clusters
3. For each non-core point do:
    - Add to a neighboring core point if possible
    - Otherwise add to noise

This algorithm can handle noise well, and has no limitations of shapes on the clusters. but you need to choose $\epsilon$and minPts.

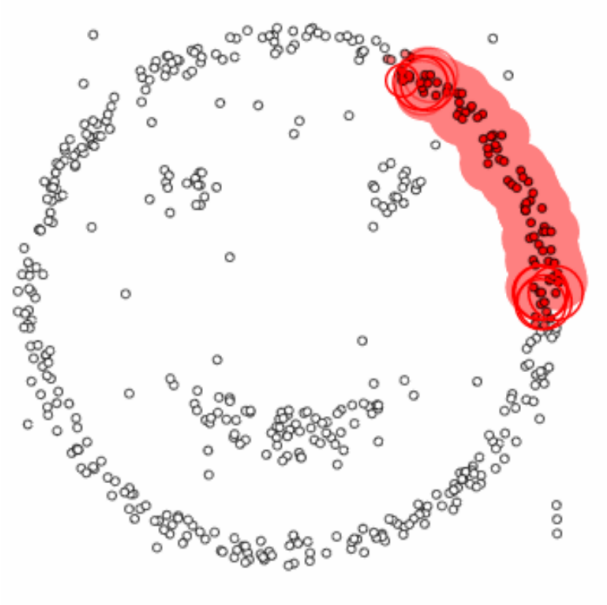

Figure 32: Illustration of clustering with DBSCAN.

2.2 Classification algorithms

You can use partitions, which is splitting single variables e.g. features, into two or more partitions. This could be done using thresholds.

2.2.1 Decision tree classifiers

Using partitions in the featurespace, we can use trees to decide wether or not a particular datapoint should be in partition x. So when we get new data in, then we will be able to decide which partition it resides in. But there may be outliers, therefore we can use the probability of the sample being in partition x, and choose the highest probable partition.
The tree is determined by training, by starting with all the features in the root node. Then every possible binary split is computed. For each split, then entropy and thereby information gain is computed. The highest information gain is chosen for that node. If a node ends up having only one features belonging to one class, it is left as a leaf node, and thereby we are sure that there is no more splits needed.

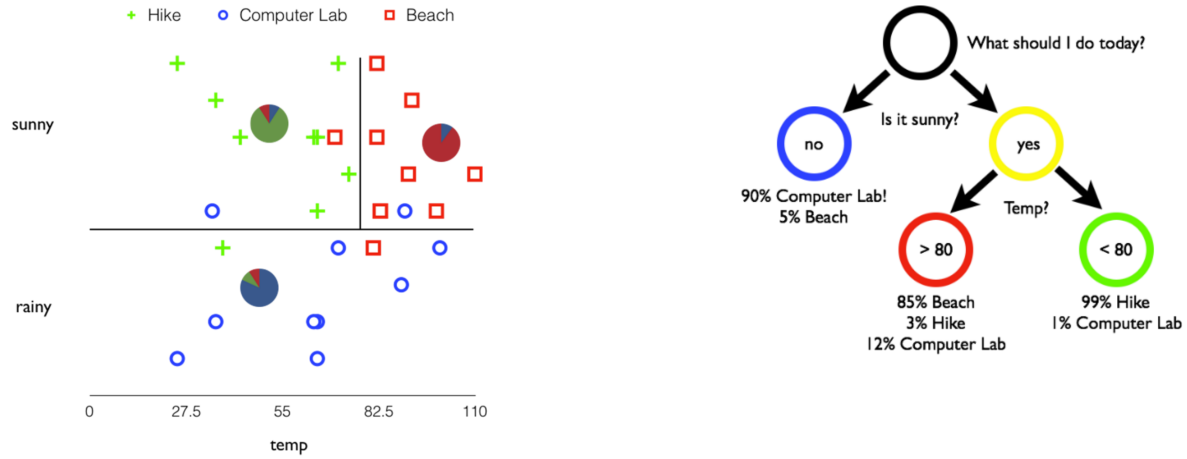

Figure 33: Decision tree example

2.2.2 Random Forest

It creates multiple decision trees.
When a new datapoint needs to be classified, it is run through all the decision trees, which each gives their own independent classification. Then we use **majority voting** by choosing the class which constituted the highest number of classifications.
Based on a labelled training set of N cases.
Algorithm

1. If the number of cases in the set is N, sample N cases at random $\rightarrow$ with replacement. This sample is used to grow the tree.
2. If there are M input features, a number $m << M$ is specified at each node. m variables are selected, at random out of the M features, and the best split on these m is used to split the node. m is constant throughout the forest growing.
3. Each tree is grown to the largest extent possible. No branches are removed (no pruning)
4. Predict new data by aggregating (grouping) the predictions of the n trees, and use majority voting to classify. Use the average for regression problems.

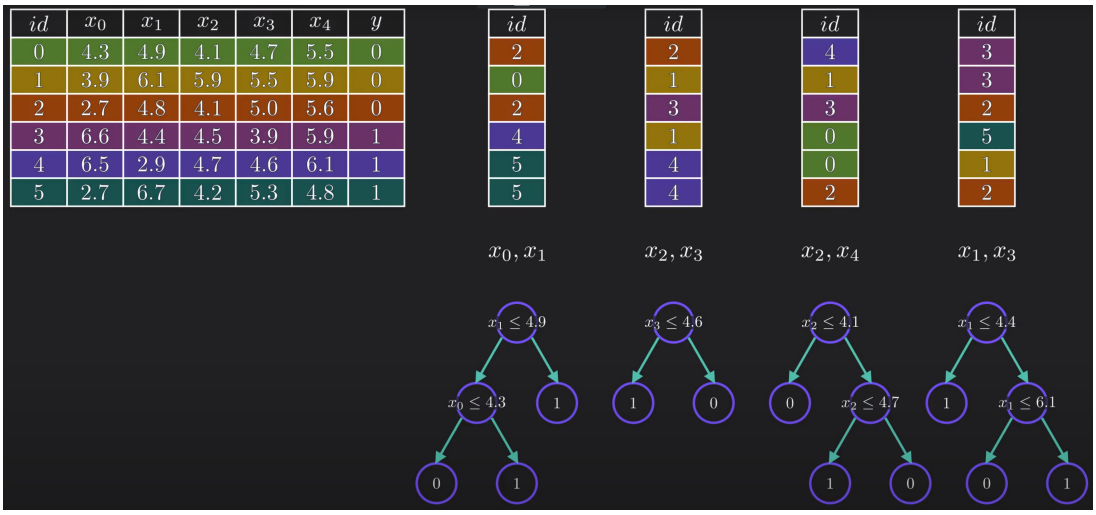

Figure 34: Random Forest algorithm

2.2.3 SVM - Support Vector Machines

Using hyperplanes to seperate datapoints. It tries to **maximize margin** towards the nearest datapoints.
For linear case, it is called linear SVM.
For every chosen line, increase the margin until a datapoint is reached. The reached datapoint is called **support vectors**.
designed for **binary** classifiers, but can be combined to solve multi-class problems.

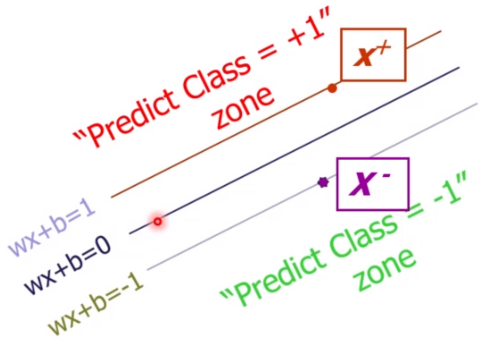

Figure 35: SVM

We label every datapoint beneath the line -1 and +1 for datapoints above.
We compute the margin by:

$$M = \frac{(x^+ - x^-)w}{|w|} = \frac{2}{|w|} \tag{36}$$

Where w is the vector from the center to the margin line.
We want to maximise the margin M, which is the same as:

$$minimize \ \frac{1}{2}w^T w \tag{37}$$

Which is solved with quadratic programming. Where ever datapoint y_i must satisfy:

$$y_i(wx_i + b) \geq 1 \forall i \tag{38}$$

noisy trainingset → slack variables
Non-linear case → map to higher dimension using **Kernel functions** .
Proporties of SVM

- Great with large datasets/featurespaces → only uses support vectors
- Overfitting can be handled with slack variables.
- Convex problem → one solution

2.3 CNN: Convolutions, pooling

- We **learn features** instead of handcrafting them.
- We learn the classification.
- Handcrat networks instead.

A convolution could be a 3×3, which is convoluted with the entire input image.

- Early convolutions comptue the more **global features**

Each entry in the convolution kernel is a weight. This weight is learned through backpropagation.
The process of convoluting the input image is also known as filtering. It extracts features

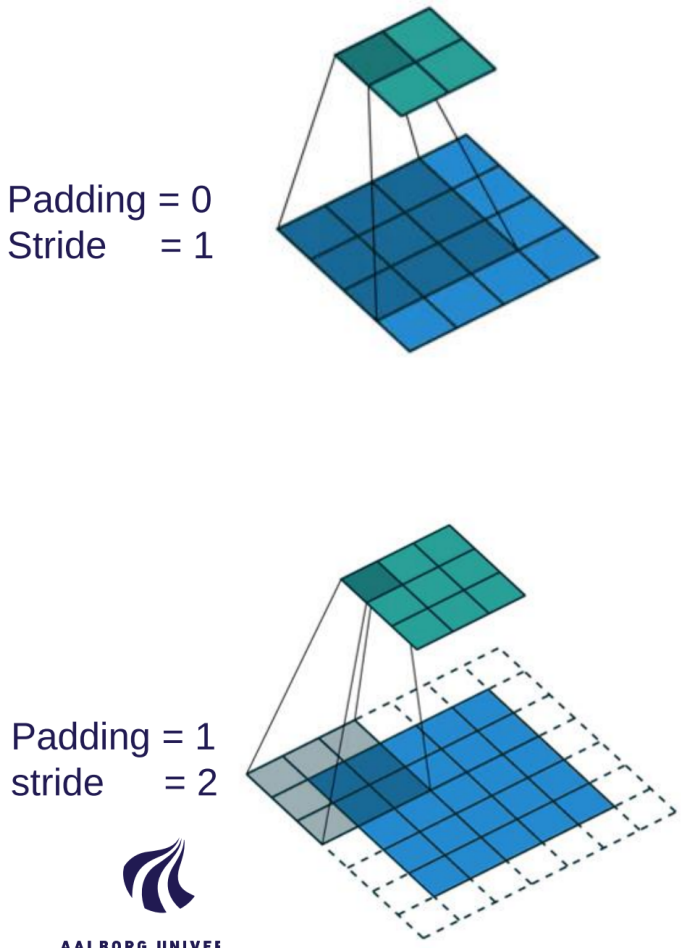

Figure 36: Illustration of filter size, padding and stride.

To preserve the size of the input feature map, then we use **padding**, which is adding 0's to the edge.
Stride is the step size.
We stack filters into channels.

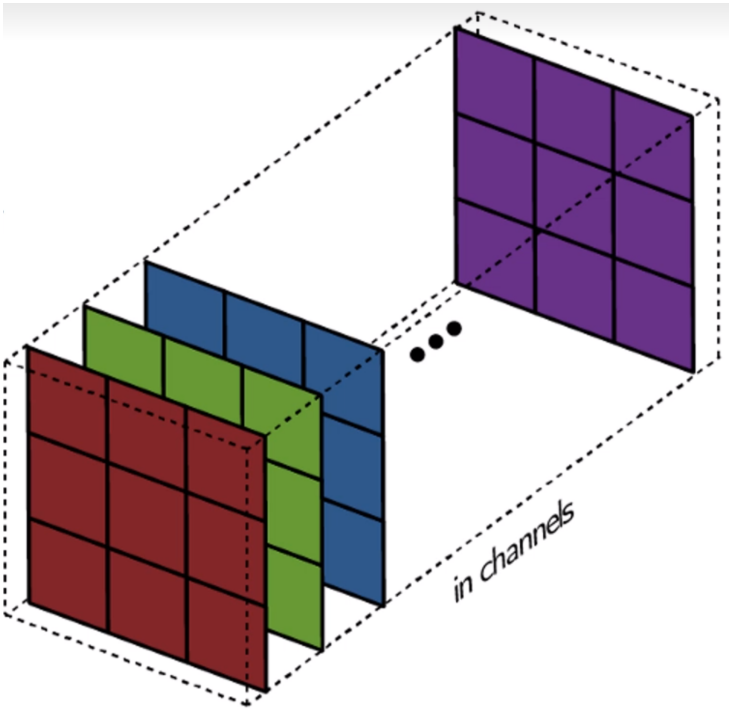

Figure 37: Filter channels by stacking them

The more layers of convolutions we have, the higher level features we genererate.

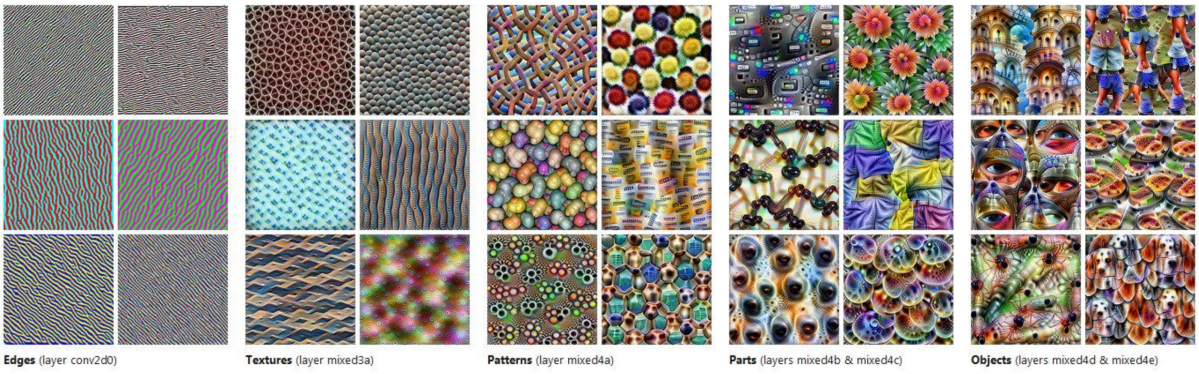

Figure 38: Different levels of details in the layers of convolutional featuremaps.

Stacking Convolutional layers

VGG-Net (2015)
By stacking convolutional layers it is possible to **simulate a larger receptive field**. This means that for each kernel size it gets increased because we have a stride of 2. This results from a 3×3 kernel, to a 5×5 without having the same number of convolutional weights (parameters) that should be learned. This would be $2 \cdot 9 = 18$instead of just using a standard 5×5 kernel, which results in 25 parameters.

Parallel convolutional layers

Inception (2015)
By doing parallel convolutional layers, it is possible to choose in the end what layer worked the best.

Skipping convolutional layers

RES-Net (2016)

By skipping convolutional layers, it is possible to compute the residual of the featuremap, which is the difference that the actual layers does. This effectively reduces the noise in the bottom of the layer during back propagation known as the **vanishing gradient problem**.

Dropout

To avoid overfitting we can use dropout. Dropout is **randomly removing 50% of the network**, to force the network to not rely on single neurons → Spread the knowledge out on multiple neurons.
3 options for using a network:

- Use a pretrained network
- Transferlearning (e.g. discarding unnecessary classes, retraining part of the weigthts by **freezing layers** , etc.)
- Train from scratch

2.3.1 Pooling

Also known as **subsampling** → reduces the dimensions of a layer.
The types of pooling are:

- MAX-pooling
- Average pooling
- Sum-pooling

Pooling is a hyperparameter, you can not learn it.

MAX-pooling

Most popular type of pooling. It finds the maximum value within the pooling kernel size, often 2×2.

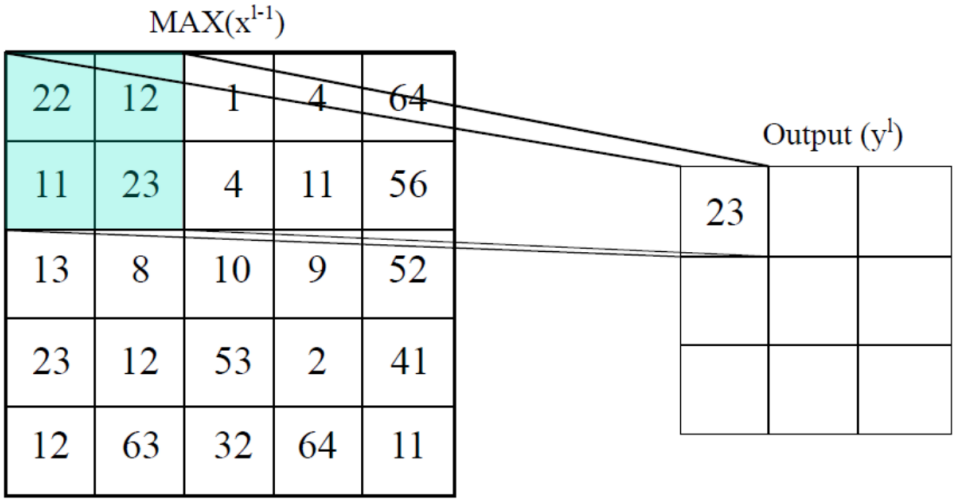

Figure 39: Illustration of max pooling

2.4 CNN: Training and fine-tuning

Training

The components of training CNN's are:

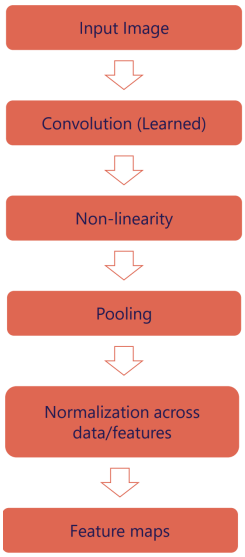

Figure 40: Components in CNN's

This is known as feed-forward.

1. Input image
2. Convolution: here we conduct the convolutions. They can be done multiple times to generate more featuremaps.
3. Non-linearity: Here we apply the ReLU function which non-linearises the loos function. Done per pixel.
4. Pooling: (optional) Here we do subsampling, by e.g. max pooling
5. Batch Normalization across data/features: (optional) Normalize each input for each layer and it **improves training speed, less sensitive bad initialization, and improves regularisation**. It strikes a **balance between full gradient computation and SGD**.
6. Fully connected layer resulting in a featuremap for classification.

We can have multiple convolutional layers. This is also refered to as **depth**. Width is the number of filters in that convolutional layer.

Training steps

1. Define the problem → **Classification / Object detection / segmentation etc.**
2. Collect dataset that represent the problem. → Split into **Training, Validation, Test**
3. Preprocess the data → **Normalize** or **standardize** the data
4. Choose the **architecture** of the network. → Choose layers, (Convolutional, Pooling, fully connected). Specify **activation functions, input/output dimensions**
5. Compile the model → Choose **Loss function, optimizer (SGD), learning rate, evaluation metrics (precision, recall etc.)**
6. Train the model → Feed data through, compute loss, and update weights in back propagation. Monitor accuracy to identify potential overfitting.

7. Tune hyper parameter tuning → **learning rate, batch size**
8. Evaluate on test set.
9. Finetune → Adjust architecture, collect more data.

Backpropagation

Done using errorpropagation through the network using the chainrule.
Stochastic gradient descent Due to the loss of gradient problem, it is necessary to use stochastic gradient descent. This is done by introducing stochastic noise into the gradients, and thus manipulating both the descent direction and the magnitude of descent.
Momentum

2.5 CNN: Object detection

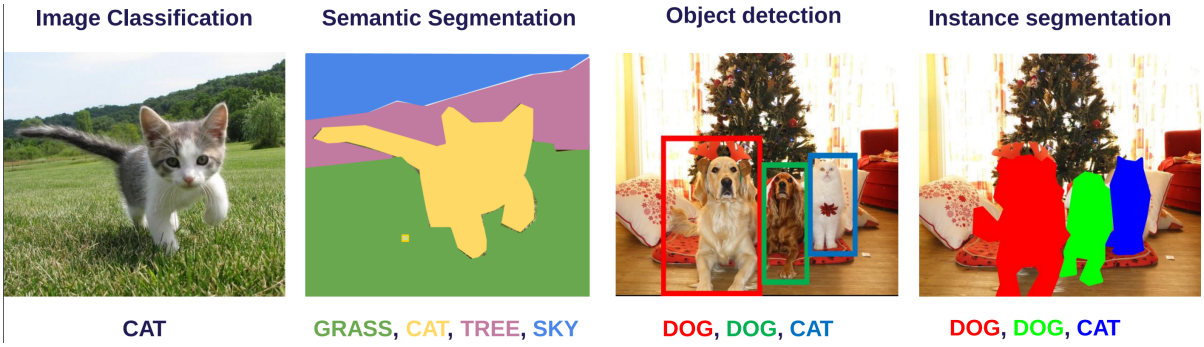

Figure 41: Different types of CNN's

Single Object detection

Two tasks:

- **What** is the object?
- **Where** is the object?

From the fully connected layer, we define the output layer with probabilities between 0 and 1, for each class. Compute the loss e.g. by using softmax loss. To figure out where the object is, we have a fully connected layer with regression (continuous variables) for which we compute the bounding box paramters. Here we can use the L2 loss function to figure out how far from the real bounding box it is, and by combining these two losses with a weighted sum, we get what is known as a **multitask-loss**. This is used to back propagate the CNN.

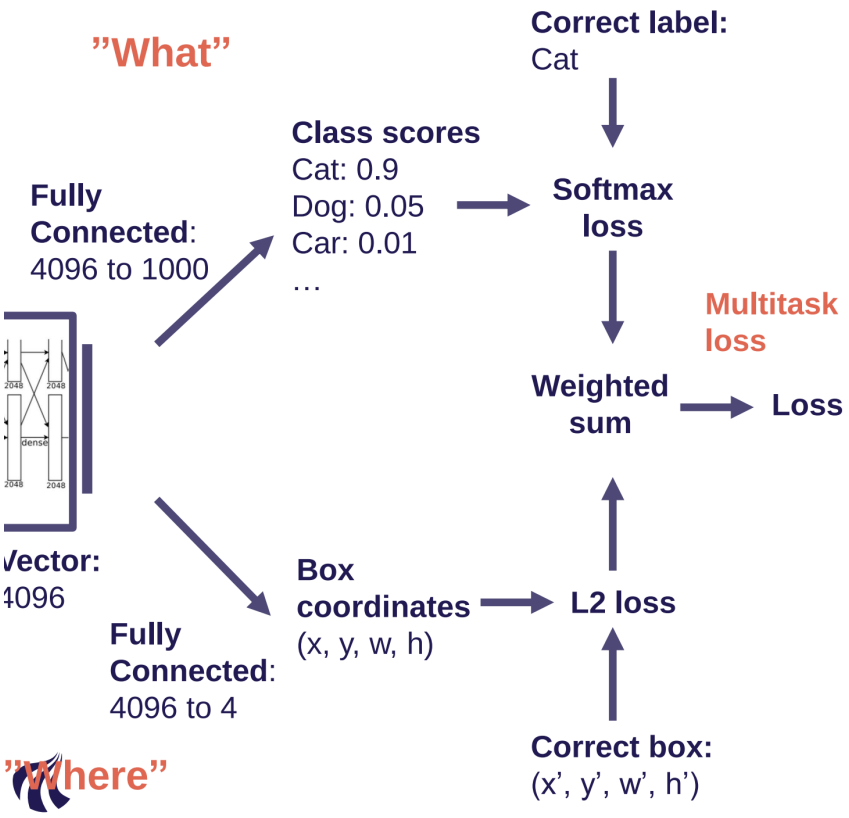

Figure 42: Illustration of the head layers on single object detection.

Multi Object detection

Done by Regional-based CNN's (**R-CNN**)
By running **Selective search** we get **region proposals**, which are boxes covering all objects.

1. Warp each region to be 224×224 pixels (input size of ImageNet)
2. Each region can be passed through a single convNet classifier network.
3. Adjust region positons using regression.

Slow $\rightarrow$ feedforward for each region

Solved by **Fast R-CNN**
This network takes the convolutional part and then it algins the regions, and they are cropped using **RoI Pool**. Then each region is run through a small neural network, called **per-region network**. The **seletive search** is computed on the **CPU**, so it takes a few seconds.

Solved by **Faster R-CNN**
Here the regions are found using a CNN instead of using selective search. Known as **RPN (Regional Proposal network)**
We imaginge anchor boxes, for each pixel in the featuremap. Is this anchor box an object?
Training the Faster R-CNN

1. **RPN classification** Is the anchor box an object?
2. **RPN Regression** Predict transform from anchor box to proposal box
3. **Object Classification** Classify into backgrond /object classes
4. **Object regression** Predict transform from proposal box to object box.

These are known as **Two-stage** proposals. There exist single stage proposals as well. These go directly to clasifying the anchor boxes instead of figuring out if the anchor box is part of an object and then figuring out what object.

Evaluating Object detection performance

By computing the intersection over union, we get at measure between 0 and 1 of how well the prediction fits the ground truth.

$$\frac{Areaofintersection}{Areaofunion} \tag{39}$$

- IoU $> 0.5 \rightarrow$ Decent
- IoU $> 0.7 \rightarrow$ Great
- IoU $> 0.9 \rightarrow$ Almost perfect

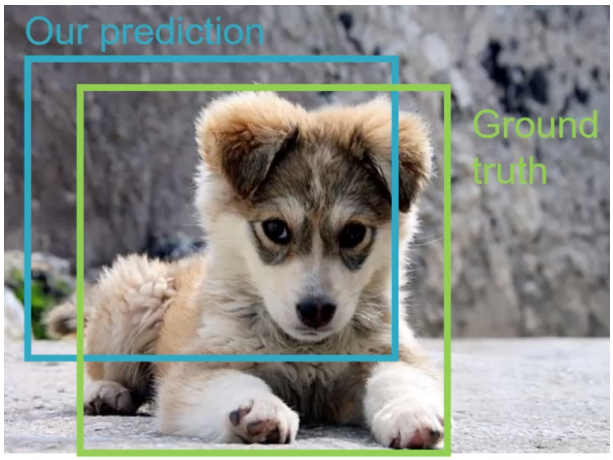

Figure 43: Comparing predicted object to the actual object (manually annotated)

- There may be many **oberlapping** boxes. Solved by **Non-maximum suppresion** :
    1. Select highest scoring box
    2. Eliminate lower IoU scoring boxes by comparing it to every other bounding box. If they are above 0.7.
    3. If overlapping boxes remain, repeat.

Mean Average Precision - mAP
For each category compute **Average Precision** = Area under precision recall curve.
The compute the overall mean of the average precessions of all the categories. Gives an overall metric.

2.6 CNN: Semantic and instance segmentation

2.6.1 Semantic segmentation

Label each pixel with a category, no instances of objects.

Figure 44: Illustration of semantic segmentation

Output image is same resolution as input image. → apply 0 padding

Done with fully convolutional network

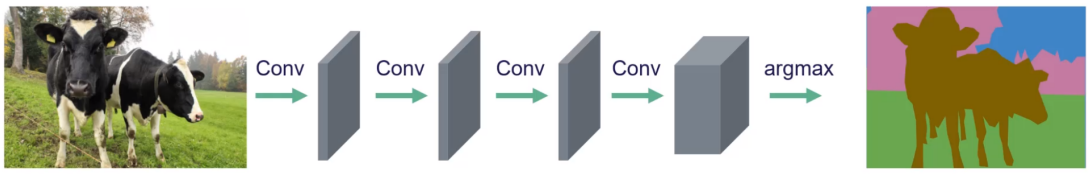

Figure 45: Fully convolutional network that preserves the image resolution through the network.

Problems:

- We have relatively **small receptive field**. Many sequential convolutions are required to expand receptive field → expensive
- High res images are expensive.

Could be solved by reducing image size and expanding it in the end

Done using **downsampling (pooling)** and **upsampling**

Upsampling

Transposed convolution. Done by introducing padding in the lower resolution input image. Can be **learned**.
Use **skip connections** to preserve fine-grained textural information.

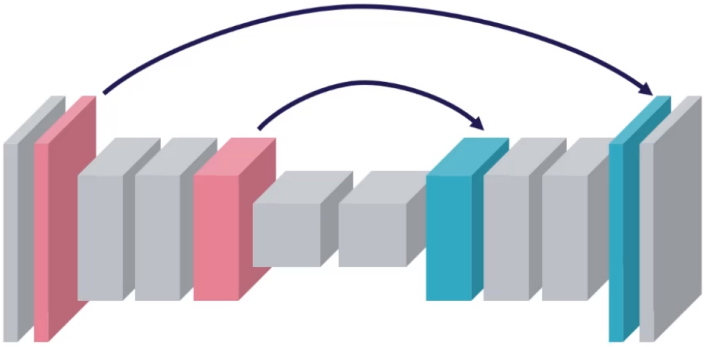

Figure 46: Illustration of a downscaling and upscaling network with **skip connections**

MS-COCO is a dataset that includes Instance segmentation. Not just single classifications for an image.

2.6.2 Instance segmentation

Decide what object instance a pixel belongs to.

Mask R-CNN

Build on Faster R-CNN

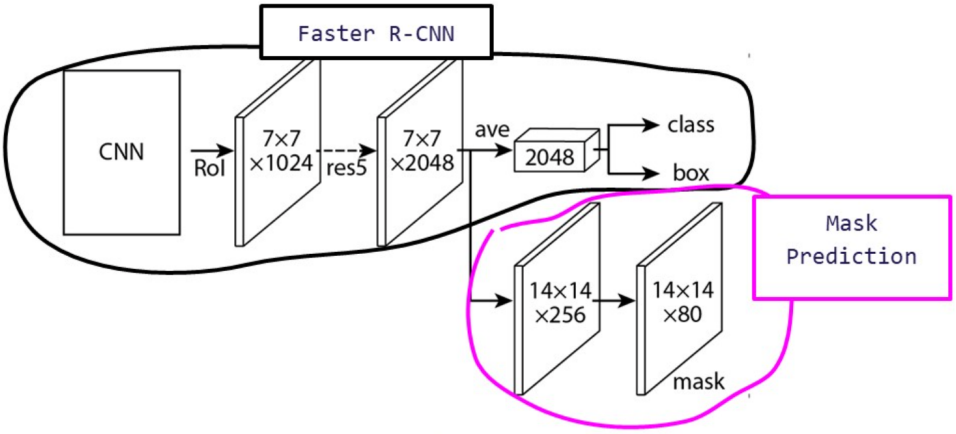

Figure 47: Mask R-CNN

- Fast R-CNN outputs probability of belonging to class x and the bounding box associated with it.
- The pink part predicts the mask (outline) of the instance.

2.7 CNN: Pose estimation

We estimate the human poses with joints and links, such that they look like a skeleton.

Figure 48: Human pose estimation

Goal: estimate 2D or 3D positions of joints.
Problems:

- Many degrees of freedom
- Varying clothes / appearance
- Self occlusions

pre-deep learning: Pictorial structures in 2D. Set relative constraints (springs) between the joints.

2D Pose estimation - DNN

- **DeepPose:** first DNN method. Based on itteratively refining the position of the joint. The input size changes from iteration to iteration.

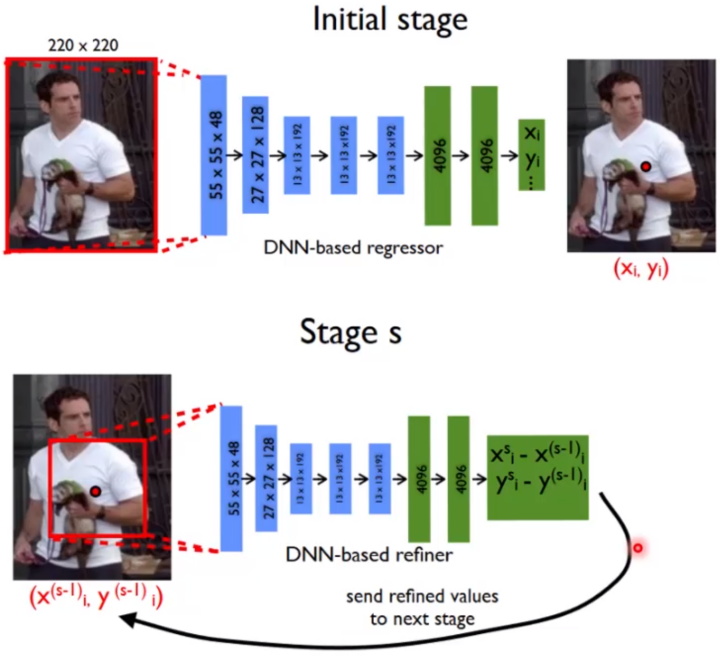

Figure 49: Illustration of the DeepPose algorithm

- **Mask R-CNN:** Can be trained to find only a single pixel to illustrate the joints. So instead of masking the entire object, it masks a single pixel. Predict k joints.

- **HRNet:** High resolution is maintained through the network.

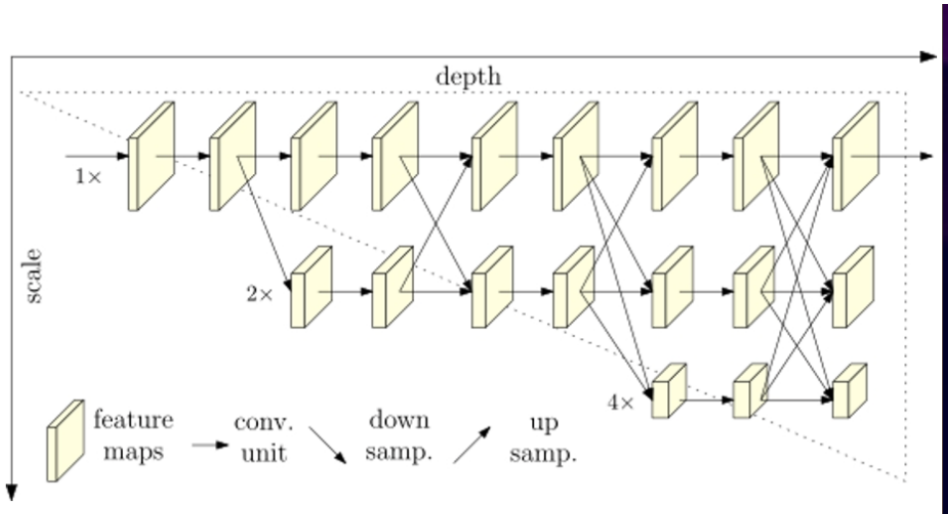

Figure 50: HRNet network

3D Pose estimation

Requires Ground truth with motion tracking systems. Expensive.
You can also input 3D features into the network, e.g. depth camera input or multiview.
DNN approach

1. Use a large dataset of 3D poses and take numerous 2D images from arbitrary angles.
2. Use the 3D, 2D pairs as training data.
3. Given a input image, find 2D pose
4. Find best matching 2D pose from the dataset, and output the corresponding 3D pose.

2.8 Point cloud processing

- Set of 3D coordinates
- May include additional information like **Surface normal, color**

You can compute point clouds from e.g.

- Depth from disparity (stereo vision)
- Directly measure them from sensors
- Can be computed from RGB-D using intrinsic and extrinsic camera paratmers.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = d \begin{bmatrix} \frac{u-u_0}{f_x} \\ \frac{v-v_0}{f_y} \\ 1 \end{bmatrix} \tag{40}$$

  where u, v are the image coordinates with depth d, and u_0, v_0 are the optical center. f_x, f_y being focal length in pixels and X, Y, Z being 3D camera coordinates.

PCL: Point Cloud Library
Each of these methods reside in PCL

Pass through filter

Filter the pointcloud **based on position**. Define the positional threshold you want to keep, and then everything else is discarded.

Outlier removal

Can be done with **StatisticalOutlierremoval**.

1. Measures distance from each point to its 50 closest neighbors
2. Computes the mean and standard deviations from these distances
3. if a point is over 1 std. away from the point, then it is discarded.

Clustering

By specifying a max distance from a point that a point must be to be within a cluster, we can define clusters.
Note you can specify min- and max cluster size.

RANSAC

1. Randomly sample x points from the pointcloud.
2. Fit a plane to the points.
3. Choose inlier distance, and let the model with most inliers win.

2.9 Deep learning for point clouds

Challenges of using DL for pointclouds

1. Images are fixed size with good knowledge of what is to the left and right of a pixel. Pointclouds are not like this.
2. Can vary in size.
3. Unordered list of points. → Unclear neighborhood.

PointNet

Solves the unordered problem by using a symmetric function (output is the same regardless of input).
This is done by taking a fixed number of points (1024), and running them through **individual 1D CNN's** and extracts a 1024 dimensional featuremap from this. From here we do max pooling to extract the best feature, and then run it through a Deep Neural network to resolve a label.

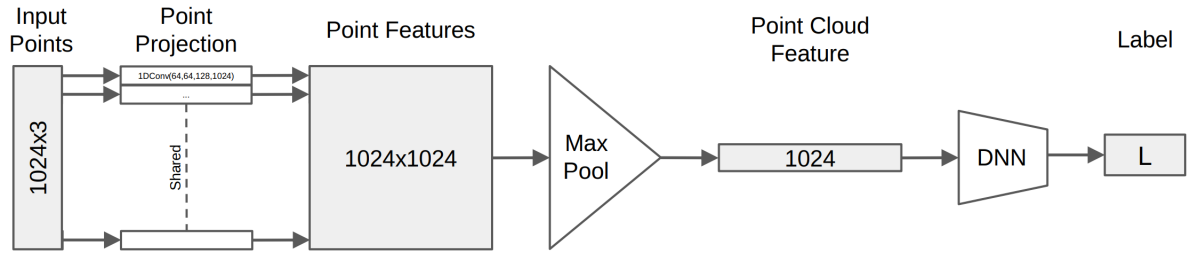

Figure 51: PointNet

Graph Neural Networks

Resolves all problems:

- Permutation invariance
- Cardinality invariance
- Neighbourhood features.

The Dynamic Graph CNN computes a neighbourhood for each pixel, and the uses that neighborhood to compute the next featureset through edge convolutions. As more layers are computed, the features becomes enriched with neighborhood information of larger neighborhoods, and become more semanticly apparent.

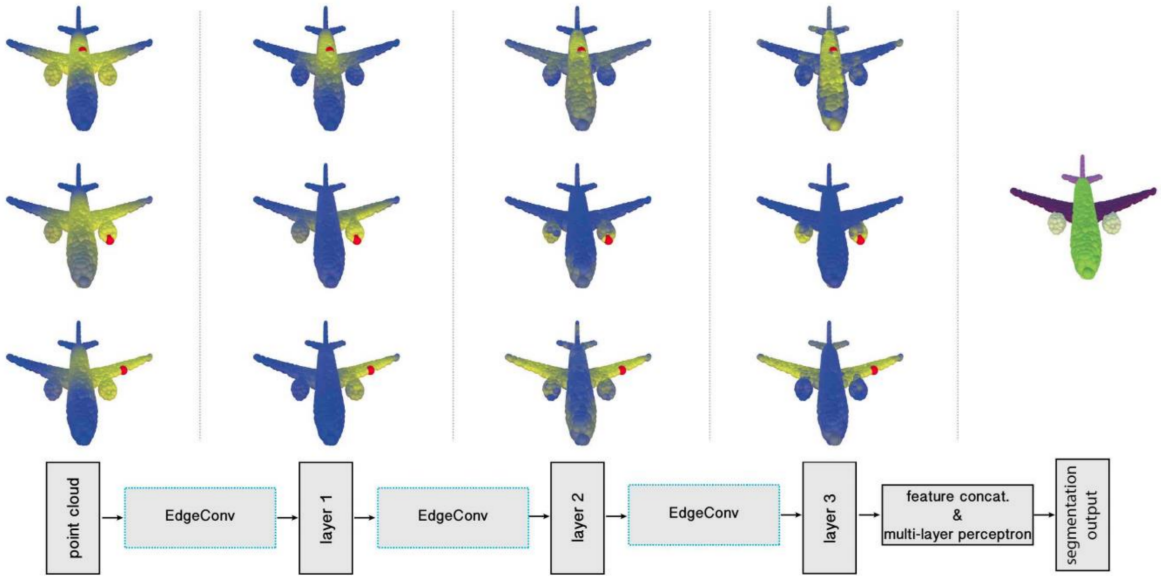

Figure 52: Graph Neural Network - Dynamic graph CNN for learning on point clouds

3 Mini project

3.1 Miniproject

Figure 53: Miniproject results

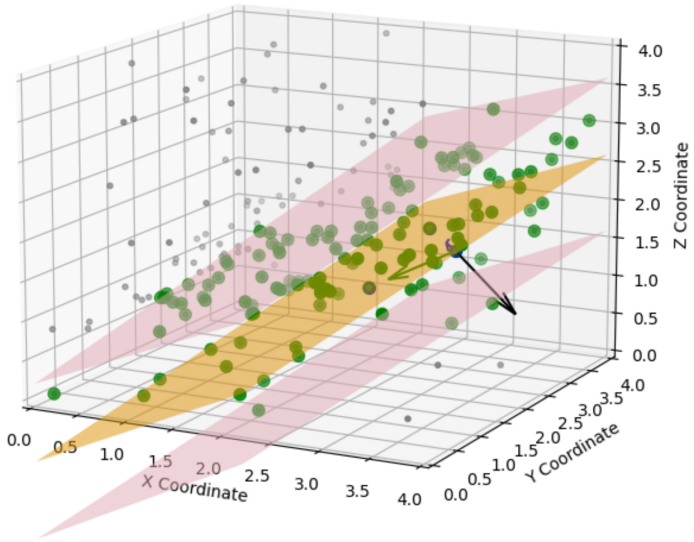

Figure 54: Ransac visualization

Camera: Intel realsense D435. Has an IR projector and IR 2 sensors. Using camera intrinsics to aquisite the image. RGB image is 1920×1080, and Depth is 1280×720 UV mapping takes the RGB image and maps it to the depth map, such that it has the same resolution. We return the Depth image, the color image, and the pointcloud.
HSV:

1. $45 < H < 65$
2. $35 < H < 255$
3. $35 < H < 255$

we set all other verticies to 0,0,0
By using the UV map, we can for each vertex in the pointcloud find the corresponding pixel in the RGB image, and then use the HVS colors on that image to determine if it is green or not.

RANSAC

From the 3 selected verticies, we generate two vectors and from those, the surface normal vector.

3.1.1 Tips

- Use the colors as dimensions in the featurevector in RANSAC
- Use Kalman filter to track the plane
- Use the previous number of inliers to use as a threshold for exiting the current itteration. → No need to find further planes if we found a plane with similar number of inliers as the previous best inlier.