Computer Vision

Cristian Perez Jensen

January 25, 2024

Contents

SUMMARY OF NOTATION

$\doteq$	equality by definition
$\approx$	approximate equality
$\propto$	proportial to
$\mathbb{N}$	set of natural numbers
$\mathbb{R}$	set of real numbers
$[m]$	set of natural numbers from 1 to m, $\{1, 2, \ldots, m\}$
$i : j$	set of natural numbers between i and j, $\{i, i+1, \ldots, j\}$
$f : A \to B$	function f that maps elements of set A to elements of set B
$\mathbb{1}\{\text{predicate}\}$	indicator function (1 if predicate is true, otherwise 0)

LINEAR ALGEBRA

$\boldsymbol{v} \in \mathbb{R}^n$	vector
$\boldsymbol{M} \in \mathbb{R}^{m \times n}$	matrix
$\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_n}$	tensor
$\boldsymbol{M}^\top$	transpose of matrix $\boldsymbol{M}$
$\boldsymbol{M}^{-1}$	inverse of matrix $\boldsymbol{M}$
$\det(\boldsymbol{M})$	determinant of $\boldsymbol{M}$

CALCULUS

$\frac{\mathrm{d}}{\mathrm{d}x} f(x)$	ordinary derivative of $f(x)$ w.r.t. x
$\frac{\partial}{\partial x} f(\boldsymbol{x})$	partial derivative of $f(\boldsymbol{x})$ w.r.t. x
$\boldsymbol{\nabla} f(\boldsymbol{x}) \in \mathbb{R}^n$	gradient of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$

MACHINE LEARNING

$\boldsymbol{\theta} \in \Theta$	parameterization of a model
$\mathcal{X}$	input space
$\mathcal{Y}$	output space
$\mathcal{D} \subseteq \mathcal{X} \times \mathcal{Y}$	labeled training data

1 Local features

For many computer vision tasks, we need to be able to identify distinctive points that can be matched to points of other images.[1] These tasks require that points that depict the same object must be detected independently in different images, and it must be possible to match such corresponding points between images. This leads to the following requirements,

- Translation, rotation, and scale invariance;
- Robustness to affine transformations, noise, and occlusion;
- A sufficient amount of points have to be detected over an object;
- Regions must contain distinctive structure;
- Real-time performance.

The following is a general approach for finding and matching keypoints between images,

1. Find a set of distinctive keypoints;
2. Define a region around each keypoint, dependent on scale;
3. Extract and normalize the region content to make it invariant to affine transformations and noise;
4. Compute a local descriptor from the normalized region;
5. Match local descriptors between images.

[1] Local features are used in tasks such as object recognition (section 4), object tracking (section 7), and structure reconstruction (section 11).

1.1 Harris detector

The *Harris detector* [Harris et al., 1988] is an algorithm for keypoint localization (step 1 of the general keypoint matching framework). We are looking for points that have significant change in both the x- and y-direction, *i.e.* corners, such that the point will be localizable at any orientation. A good illustration of this is Figure 1.

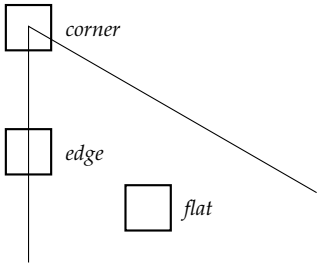


Figure 1. The Harris detector seeks to find corner points, which have a significant change in both the x- and y-direction. Edges are not localizable, because if we rotate the patch, we will not be able to match the two rotated patches together.

To find corners, we want to find points such that if we shift the window in any direction, it should give a large change in intensity. Let's define an error function as the following,

$$E(u,v) \doteq \sum_{(x,y)\in W} (\mathbf{I}[x+u, y+v] - \mathbf{I}[x,y])^2,$$

where W is the set of all points in the window that is a potential corner. We want to find Ws such that $E(u,v)$ is large for any (u,v). We can

rederive $E(u, v)$ with its first-order Taylor expansion to make this easy and efficient:

$$\begin{aligned} E(u,v) &\doteq \sum_{(x,y)\in W} (\mathbf{I}[x+u, y+v] - \mathbf{I}[x,y])^2 \\ &\approx \sum_{(x,y)\in W} \left(\mathbf{I}[x,y] + \begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} - \mathbf{I}[x,y] \right)^2 \\ &= \sum_{(x,y)\in W} \left(\begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \right)^2 \\ &= \begin{bmatrix} u & v \end{bmatrix} \mathbf{M} \begin{bmatrix} u \\ v \end{bmatrix}, \end{aligned}$$

where $\mathbf{I}_x$ and $\mathbf{I}_y$ are the gradients in the x- and y-direction at the implicit $[x, y]$ point, and $\mathbf{M}$ is the structure tensor,[2]

$$\mathbf{M} = \sum_{(x,y)\in W} \begin{bmatrix} \mathbf{I}_x^2 & \mathbf{I}_x\mathbf{I}_y \\ \mathbf{I}_x\mathbf{I}_y & \mathbf{I}_y^2 \end{bmatrix}.$$

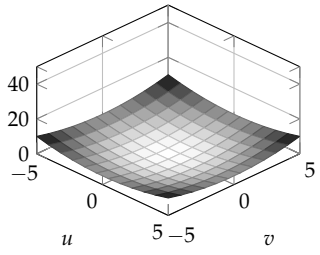
[2] This can be computed using convolutions, which makes it efficient.

We want to find points with an error surface as in Figure 1c. The eigenvalues of a matrix tell us how much it changes in the direction of their eigenvectors. Since we found that $E(u, v)$ is approximately equal to a matrix multiplication with $\mathbf{M}$, the eigenvalues of $\mathbf{M}$ tell us how much the error function changes as we shift the window in the direction of the eigenvectors. Thus, we want to find points where both eigenvalues are large. The Harris detector uses the following response function, because it can be rederived to not need the (computationally inefficient) eigendecomposition,

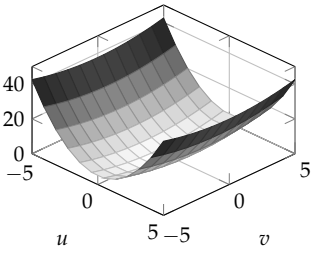
$$\begin{aligned} R &= \lambda_1\lambda_2 - \kappa(\lambda_1 + \lambda_2)^2 \\ &= \det(\mathbf{M}) - \kappa \mathrm{tr}(\mathbf{M})^2. \end{aligned}$$

The response function R is a measure of a window's "cornerness". If $R > 0$, its window is considered to be a corner for the Harris detector.

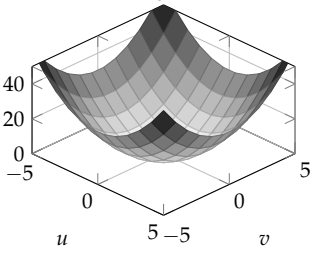
Lastly, the Harris detector applies non-maximum suppression, which reduces clusters of points with $R > 0$ to a single point, since they capture the same corner.



(a) Error surface of a patch containing a flat area.



(b) Error surface of a patch containing an edge.



(c) Error surface of a patch containing a corner.

Figure 2. Different types of error surfaces.

1.2 *Scale-invariant region selection*

The Harris detector is only able to find keypoints at a fixed scale. However, we would like to be able to detect keypoints at any scale and use this scale to match keypoints of different scales. In other words, we want to be able to match keypoints in a zoomed-in version of an image to points in a zoomed-out one. The naive approach would be to

compare descriptors while varying the patch size, but this results in a combinatorial search problem.

A better idea is to do detection over a scale space, which has three dimensions x, y, σ, where the σ parameterizes the convolution with a Laplacian of Gaussian (second derivative of the Gaussian). This "blurs" the image so only large structures remain. Then, we compute the response function for all points (x, y, σ), and call points that are a maximum in a $3 \times 3 \times 3$ grid around them a feature point. The Laplacian of Gaussian can be approximated by the difference of Gaussian operator, which is more efficient.

1.3 Local descriptors

SIFT [Lowe, 2004] is an algorithm that uses the orientation of gradients within a patch to compute a descriptor (step 4 in the general approach). It does so by dividing the found patch into a 4×4 grid and computing the 8-bin histogram of gradient orientations, weighted by their magnitude, within each gridcell. This results in a $4 \times 4 \times 8 = 128$-dimensional vector representation of a keypoint.

We can assign an orientation to a keypoint by creating a histogram of local gradient orientations, weighted by a Gaussian window, within the patch and assigning the orientation to be the maximum gradient orientation. We can then align the descriptor to this orientation, such that we can directly compare two descriptors.

1.4 Matching

There are three main ways of matching keypoints between two images,

- *One-way matching*: for each keypoint in image 1, match it to the keypoint in image 2 with the smallest descriptor difference;
- *Mutual matching*: for each keypoint in image 1, match it to the keypoint in image 2 with the smallest descriptor difference only if this condition also holds the other way;
- *Ratio-threshold matching*: for each keypoint in image 1, match it to the keypoint in image 2 with the smallest descriptor difference only if the ratio between the smallest and second smallest difference is less than some threshold.

2 Deep learning

In this course, we will see many computer vision tasks that predate the deep learning era. However, these tasks have seen much improve-

ment by applying machine learning to them. Thus, we will first be introduced to the tasks, followed by how the problem was dealt with initially, and then how machine learning is applied to it.

2.1 Convolutional neural networks

TODO

2.2 Sequence models

A *recurrent neural network* (RNN) [Rumelhart et al., 1985] is a neural network that maps from an input space of sequences to an output space of sequences in a stateful way. That is, the prediction of the output $\boldsymbol{y}_t$ depends on all inputs $\boldsymbol{x}$ that came before t. This is done by keeping track of a hidden state $\boldsymbol{h}_t$, which is a function of all previous inputs and the current input,

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{W}_x \boldsymbol{x}_t).$$

An application of RNNs in computer vision is processing video, where each frame is processed by a CNN. Then, each frame's vector representation is used to update the hidden state of the RNN to obtain a representation of the video.

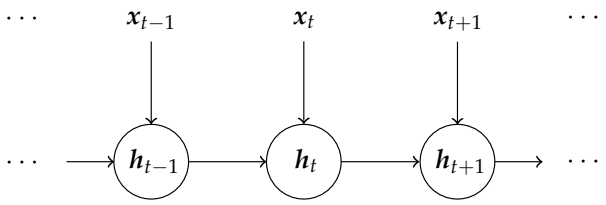


Figure 3. Computation graph of an RNN.

Putting everything together, backpropagation computes the following gradient w.r.t. $\boldsymbol{W}_h$,

$$\begin{aligned}
\frac{\partial \ell}{\partial \boldsymbol{W}_h} &= \sum_{j=0}^{T} \frac{\partial \ell_j}{\partial \boldsymbol{W}_h} \\
&= \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial \ell_j}{\partial \boldsymbol{h}_k} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}_h} \\
&= \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial \ell_j}{\partial y_j} \frac{\partial y_j}{\partial \boldsymbol{h}_j} \frac{\partial \boldsymbol{h}_j}{\partial \boldsymbol{h}_k} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}_h} \\
&= \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial \ell_j}{\partial y_j} \frac{\partial y_j}{\partial \boldsymbol{h}_j} \left(\prod_{m=k+1}^{j} \frac{\partial \boldsymbol{h}_m}{\partial \boldsymbol{h}_{m-1}} \right) \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}_h},
\end{aligned}$$

where T is the amount of timesteps in the sequence. If we use the following approximation $\frac{\partial \boldsymbol{h}_m}{\partial \boldsymbol{h}_{m-1}} \approx \boldsymbol{W}_h$, then we see that there are a large

amount of matrix multiplications of $\boldsymbol{W}_h$ with itself. This will lead to either vanishing or exploding gradients, depending on whether $\det(\boldsymbol{W}_h)$ is greater or less than 1. Possible solutions for this are gradient clipping and gated recurrent units [Hochreiter and Schmidhuber, 1997, Cho et al., 2014].

2.3 *Transformers*

Attention is a mechanism in neural networks that a model can learn to make predictions by selectively attending to a given set of data by using query $\boldsymbol{q}$, key $\boldsymbol{k}$, and value $\boldsymbol{v}$ vector representations. The query and key vectors are used to determine how much weight should be given to the value vector.[3] The weights are computed by $\boldsymbol{A}_i = \text{softmax}(\boldsymbol{q}_i^\top \boldsymbol{k}_i)$, so the values after the attention block can be computed as follows,

$$\text{att}(\boldsymbol{x}_i) = \sum_j \boldsymbol{A}_{ij} \boldsymbol{v}_i.$$

Self-attention blocks learn the query, key, and value representations from data. More specifically, it learns matrices $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, and $\boldsymbol{W}_V$ and computes the vectors from these matrices,

$$\begin{aligned}\boldsymbol{Q} &= \boldsymbol{W}_Q^\top \boldsymbol{X} \\ \boldsymbol{K} &= \boldsymbol{W}_K^\top \boldsymbol{X} \\ \boldsymbol{V} &= \boldsymbol{W}_V^\top \boldsymbol{X}.\end{aligned}$$

Then, we can use these to compute the output of the self-attention block,

$$\text{selfatt}(\boldsymbol{X}) = \text{softmax}\left(\frac{\boldsymbol{Q}^\top \boldsymbol{K}}{\sqrt{d_q}}\right)\boldsymbol{V},$$

where d_q is the dimensionality of the query and key vectors. Furthermore, we need to add a positional encoding to provide ordering information to the model, because the self-attention operation is permutation equivariant. This is done by a sinusoidal positional encoding and are simply combined with $\boldsymbol{X}$ by addition.

Transformers use multi-headed self-attention, which is a module where self-attention is applied M times independently to the data. Thus, this module learns M different ways of looking at the same dataset. The outputs of each self-attention block is concatenated and linearly transformed to the expected dimensionality. Transformers follow this by normalization and MLP layers, as can be seen in Figure 5.

2.4 *Latent variable models*

Latent variable models map between an observation space $\boldsymbol{x} \in \mathbb{R}^D$ and latent space $\boldsymbol{z} \in \mathbb{R}^Q$ with $Q < D$. The models then learn two functions:

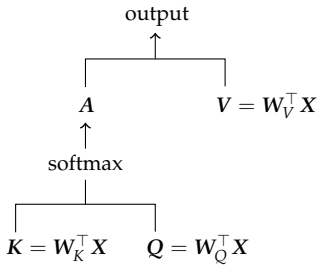


Figure 4. Self-attention mechanism.

[3] Note the parallel with dictionaries/hashmaps in programming languages, but, in the attention mechanism, we do a "soft-lookup".

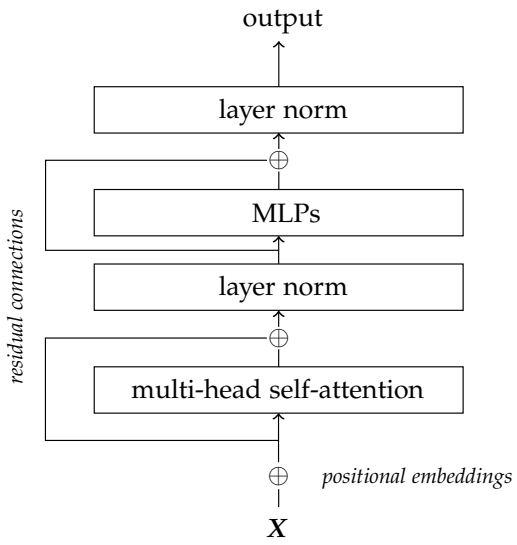


Figure 5. Transformer encoder architecture.

encoder $f_{\boldsymbol{\theta}} : \mathbb{R}^D \to \mathbb{R}^Q$ and decoder $g_{\boldsymbol{\theta}} : \mathbb{R}^Q \to \mathbb{R}^D$. These models are called *autoencoders*, because they predict their input as output.

Generative latent variable models are Bayesian probabilistic functions,

$$p(\boldsymbol{x}) = \int_{\boldsymbol{z}} p(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z}) d\boldsymbol{z}.$$

So, the probability of a sample is the "weighted average" of this sample given a latent variable weighted by the probability of this latent variable. Generally, the goal is to be able to sample from this probability distribution.

Variational autoencoders (VAE) [Kingma and Welling, 2013] sample $\boldsymbol{z}$ from a simple distributions such as the Gaussian and learn decoders that map from this distribution to data points $\boldsymbol{x}$. The aim of training is to maximize the probability of each $\boldsymbol{x}$ in the training set, *i.e.*, maximize the likelihood of seeing our data. Since the likelihood (also known as evidence) is an intractable integral, we need to derive a tractable lower bound,

$$\begin{aligned}
\log p(\boldsymbol{x}) &= \log \int p(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z}) d\boldsymbol{z} \\
&= \log \int p(\boldsymbol{x} \mid \boldsymbol{z}) \frac{p(\boldsymbol{z})}{q(\boldsymbol{z})} q(\boldsymbol{z}) d\boldsymbol{z} \\
&\geq \int q(\boldsymbol{z}) \log \left(p(\boldsymbol{x} \mid \boldsymbol{z}) \frac{p(\boldsymbol{z})}{q(\boldsymbol{z})} \right) d\boldsymbol{z} \qquad \text{Jensen's inequality} \\
&= \mathbb{E}_{q(\boldsymbol{z})} \left[\log \frac{p(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z})}{q(\boldsymbol{z})} \right],
\end{aligned}$$

which is called the *evidence lower bound* (ELBO). The probability distribution q_{theta} is the encoder and can be defined to be anything, *e.g.*, a Gaussian distribution $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ or a probabilistic deep learning model q_{theta} dependent on $\boldsymbol{x}$.

An objective function can be derived from the ELBO, which maximizes the lower bound of the likelihood,

$$\begin{aligned}\mathbb{E}_{q(z)}\left[\log \frac{p(\boldsymbol{x} \mid z)p(z)}{q(z)}\right] &= \mathbb{E}_{q(z)}[\log p(\boldsymbol{x} \mid z)] - \mathbb{E}_{q(z)}\left[-\log \frac{p(z)}{q(z)}\right] \\ &= \mathbb{E}_{q(z)}[\log p(\boldsymbol{x} \mid z)] - D_{\mathrm{KL}}(q(z) \parallel p(z)).\end{aligned}$$

The negative log-likelihood is thus upper-bounded by

$$D_{\mathrm{KL}}(q(z) \parallel p(z)) + \mathbb{E}_{q(z)}[-\log p(\boldsymbol{x} \mid z)].$$

VAEs minimize this bound,

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmin}} \sum_{\boldsymbol{x} \in \mathcal{D}} \underbrace{D_{\mathrm{KL}}(q_{\boldsymbol{\theta}}(z \mid \boldsymbol{x}) \parallel p(z))}_{\text{Encoder loss}} + \underbrace{\mathbb{E}_{q_{\boldsymbol{\theta}}(z|\boldsymbol{x})}[-\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid z)]}_{\text{Decoder loss}},$$

where we now learn a latent representation z given the datapoint $\boldsymbol{x}$.

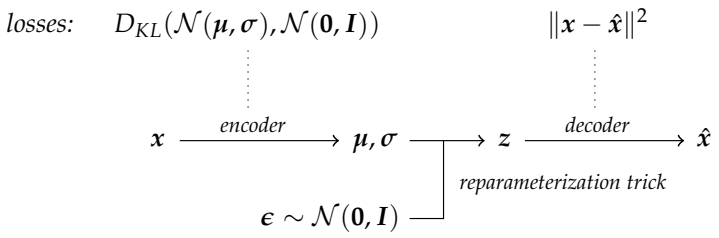


Figure 6. Variational autoencoder architecture.

Because of problems with the gradient, the encoder does not directly predict z from $\boldsymbol{x}$, but rather predicts a Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ from $\boldsymbol{x}$, which is used to sample z. This splits the model into an encoder and a decoder network that are updated separately, as can be seen in Figure 6.

Generative adversarial networks (GAN) [Goodfellow et al., 2014] do not explicitly model the likelihood. Instead, they use an adversarial process in which two models are trained simultaneously; a generator $G : \mathbb{R}^Q \to \mathbb{R}^D$ and a discriminator $D : \mathbb{R}^D \to [0, 1]$. The generator captures the data distribution, while the discriminator estimates whether a sample comes from the data distribution or the generator. At a high level, we train D to assign probability 1 to samples from the training

data and 0 to samples from the generator, while training G to fool D such that it assigns probability 1 to its samples,

$$\operatorname*{argmin}_{\boldsymbol{\theta}_G} \operatorname*{argmax}_{\boldsymbol{\theta}_D} \mathbb{E}_{p_{\text{data}}(\boldsymbol{x})}[\log D_{\boldsymbol{\theta}_D}(\boldsymbol{x})] + \mathbb{E}_{p(\boldsymbol{z})}\big[\log(1 - D_{\boldsymbol{\theta}_D}(G_{\boldsymbol{\theta}_G}(\boldsymbol{z})))\big].$$

The loss function of G solely depends on D, which is learned. So, GANs are a way of learning the loss function.

$$\textit{generator} \longrightarrow \hat{\boldsymbol{x}} \xrightarrow{\textit{discriminator}} 0$$

$$\mathcal{D} \longrightarrow \boldsymbol{x} \xrightarrow{\textit{discriminator}} 1$$

Figure 7. Generative adversarial network architecture. The discriminator must predict 0 for data points generated by the generator, and 1 for data points from the dataset. The generator wants the discriminator to predict 1 for its generations.

3 *Optical flow*

Optical flow is the pattern of apparent motion of objects, sufaces, and edges in a visual scene caused by the relative motion between an observer and the scene.[4] In other words, optical flow does not represent the actual motion of the objects in the scene (that would not be computable from images), but rather the motion of the pixels in the image plane. Optical flow is merely an approximation of the motion field. But, in most cases, it is a pretty good approximation.

[4] Contrast this with *motion field*, which is a 2D image presenting the projection of the 3D motion of points in the scene onto the image plane.

The problem statement of optical flow is to estimate the motion, *i.e.*, velocity, of each pixel, given two consecutive image frames. To solve this problem, optical flow makes the following two assumptions,

- *Brightness constancy*: when a point $\boldsymbol{x} = [x, y]$ in the image at timepoint t moves to point $\boldsymbol{x} + \delta_{\boldsymbol{x}}$ in the image at timepoint $t + \delta_t$, its luminance does not change,

$$\mathbf{I}[x + \delta_x, y + \delta_y, t + \delta_t] = \mathbf{I}[x, y, t];$$

- *Small motion*: the pixels do not move far between image frames. The implication of this is that we can use the Taylor expansion for a very good approximation around the point we care about.

Now, we can derive how to determine the velocity $\boldsymbol{v} = [u, v]$ at point $\boldsymbol{x} = [x, y]$ as follows,

$$\begin{aligned} \mathbf{I}[x, y, t] &= \mathbf{I}[x + \delta_x, y + \delta_y, t + \delta_t] && \text{brightness constancy} \\ &= \mathbf{I}[x, y, t] + \mathbf{I}_x \delta_x + \mathbf{I}_y \delta_y + \mathbf{I}_t \delta_t && \text{Taylor approximation,} \end{aligned}$$

where $\mathbf{I}_x, \mathbf{I}_y, \mathbf{I}_t$ are the derivatives at the implied location $[x, y, t]$. From this, we can derive the *brightness constancy equation*,

$$\begin{aligned}
\mathbf{I}[x, y, t] + \mathbf{I}_x\delta_x + \mathbf{I}_y\delta_y + \mathbf{I}_t\delta_t &= \mathbf{I}[x, y, t] \\
\mathbf{I}_x\delta_x + \mathbf{I}_y\delta_y + \mathbf{I}_t\delta_t &= 0 \\
\mathbf{I}_x\frac{\delta_x}{\delta_t} + \mathbf{I}_y\frac{\delta_y}{\delta_t} + \mathbf{I}_t &= 0 \\
\mathbf{I}_x u + \mathbf{I}_y v &= -\mathbf{I}_t.
\end{aligned}$$

However, it cannot be solved, because we have two unknowns u and v with one equation per pixel. This is called the *aperture problem*. The way this is dealt with is the difference between the Lucas-Kanade and Horn-Schunck algorithms, *i.e.*, their difference is how they add more constraints to make the problem tractable.

3.1 Lucas-Kanade

Lucas-Kanade applies a local method that assumes that pixels in the same area, *i.e.*, a patch around the pixel, have the same velocity. This gives us an additional constraint for every pixel in the area, which we can solve by least-squares estimation:

$$\begin{aligned}
\begin{bmatrix} \mathbf{I}_x[x_1, y_1, t] & \mathbf{I}_y[x_1, x_2, t] \\ \vdots & \vdots \\ \mathbf{I}_x[x_n, y_n, t] & \mathbf{I}_y[x_n, x_n, t] \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} &= - \begin{bmatrix} \mathbf{I}_t[x_1, y_1, t] \\ \vdots \\ \mathbf{I}_t[x_n, y_n, t] \end{bmatrix} \\
\boldsymbol{A}\boldsymbol{v} &= -\boldsymbol{b}.
\end{aligned}$$

This is equivalent to solving:

$$\begin{aligned}
\boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{v} &= -\boldsymbol{A}^\top\boldsymbol{b} \\
\sum_{(x,y)\in P} \begin{bmatrix} \mathbf{I}_x^2 & \mathbf{I}_x\mathbf{I}_y \\ \mathbf{I}_x\mathbf{I}_y & \mathbf{I}_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} &= - \sum_{(x,y)\in P} \begin{bmatrix} \mathbf{I}_x\mathbf{I}_t \\ \mathbf{I}_y\mathbf{I}_t \end{bmatrix}
\end{aligned}$$

This is the same matrix as the structure tensor we saw in the Harris corner detector. Thus, corners are good places to compute flow. If there are no corners, and *e.g.* only a line, then we have the aperture problem.

3.2 Horn-Schunck

Horn-Schunck applies a global method that assumes optical flow fields to be smooth. It seeks to minimize the following energy function,[5]

$$E = \int\int \underbrace{(\mathbf{I}_x u_{xy} + \mathbf{I}_y v_{xy} + \mathbf{I}_t)^2}_{\text{brightness change penalty}} + \underbrace{\lambda(\|\nabla u_{xy}\|^2 + \|\nabla v_{xy}\|^2)}_{\text{flow change penalty}}\, dxdy,$$

[5] In the discrete case that we care about, the integrals are replaced by sums.

where u_{xy} and v_{xy} are the velocities at $[x, y]$ forming a flow field. The first term minimizes the brightness change, which enforces the brightness constancy assumption. The second term minimizes the change in flow, which enforces the smooth flow field assumption. In the discrete case, this can be seen as minimizing the difference in flow between neighbors, resulting in smoother flow. A larger λ leads to a smoother flow field.

The derivatives of this function are the following,

$$\frac{\partial E}{\partial u_{kl}} = 2(u_{kl} - \bar{u}_{kl}) + 2\lambda(\mathbf{I}_x u_{kl} + \mathbf{I}_y u_{kl} + \mathbf{I}_t)\mathbf{I}_x$$
$$\frac{\partial E}{\partial v_{kl}} = 2(v_{kl} - \bar{v}_{kl}) + 2\lambda(\mathbf{I}_x u_{kl} + \mathbf{I}_y u_{kl} + \mathbf{I}_t)\mathbf{I}_y,$$

where $\bar{u}_{kl}$ and $\bar{v}_{kl}$ are the average flow of the 4 neighbors. The extrema of E can be found by setting the derivatives to zero and solving for the unknowns u and v. These form a linear system that can be solved iteratively using update equations,

$$\hat{u}_{kl} = \bar{u}_{kl} - \frac{\mathbf{I}_x \bar{u}_{kl} + \mathbf{I}_y \bar{v}_{kl} + \mathbf{I}_t}{\lambda^{-1} + \mathbf{I}_x^2 + \mathbf{I}_y^2}\mathbf{I}_x$$
$$\hat{v}_{kl} = \bar{v}_{kl} - \frac{\mathbf{I}_x \bar{u}_{kl} + \mathbf{I}_y \bar{v}_{kl} + \mathbf{I}_t}{\lambda^{-1} + \mathbf{I}_x^2 + \mathbf{I}_y^2}\mathbf{I}_y.$$

These update equations are used to update the flow field iteratively until convergence, starting from zero flow field.

This can also be used to compute flow field of frames with larger motions by using coarse-to-fine flow estimation. This is done by first downscaling the image until the displacement is 1 pixel. Then, compute flow of that and upscale the image and flow field. Repeat this step until you are back at the original image.

4 *Recognition*

In *recognition*, we want to be able to recognize whatever (parts of) an image depicts. For example, determining where all the humans are in an image. The challenges of this task are the following,

- Background clutter;[6]
- Intra-class variations.[7]

At first (~1960–1990), the geometric area saw recognition as an alignment problem, where everything consists of shape primitives. Then, you only had to recognize the primitives and figure out their alignment to figure out what object is depicted. Later (~1990–2000), appearance-based models estimate the appearance by combining a lot of images

[6] *E.g.* if the background is similar to the object, the object may not be recognized.

[7] *E.g.* if you want to recognize whether an object is a chair, you need to be able to detect any type of chair, which can vary greatly.

that depict the same thing, *e.g.*, human faces. Then, new data points are compared to this estimate of appearance. Sliding window approaches were also used, which compare every patch of an image to a template. Currently, image classifiers are implemented as machine learning problems that take an image as input and outputs one of a set of predefined labels.

4.1 Bag-of-words

An example of a machine learning algorithm for classification is *bag-of-words*. At a high level, it extracts local features from an image, which are compared to the local features of the training images in an efficient and robust way. Each matching patch is weighted according to the TF-IDF metric and used to determine a classification of the input image.

k-means clustering is a method of partitioning observations into k clusters. It does so by iteratively alternating between updating the clusters and assigning points to clusters. It represents the clusters by their mean, and it assigns points by assigning it to the cluster representation with the smallest distance. It repeats this until convergence, which is when none of the points update their cluster assignment.

1. *Feature extraction*: The goal of this step is to find patches that would be interesting for the classification of its image. For this, algorithms such as SIFT and Harris detection are used to find such interesting patches. These patches are then extracted and used in the following step;
2. *Dictionary learning*: Patches that depict the same usually do not contain the exact same pixels. Thus, we need some way of mapping patches that depict the same together. This is done by k-means clustering, which iteratively updates its k clusters by assigning each object to the cluster with the nearest centroid and updating their representation (centroid) to be the mean of its objects. This step is repeated until convergence. Thus, now each feature is mapped to its closest centroid, which means that similar features are mapped together;
3. *Feature encoding*: The images are encoded by the bag-of-words vectors of their features, *i.e.*, an histogram of the counts of the number of feature occurences;
4. *Classification*: The classification of an image is the class that maximizes the TF-IDF metric,[8] which weights each feature by its inverse document frequency. *I.e.* if a feature occurs in a lot of images, it has a lower weight, since it is less discriminative.

[8] Note that each feature maps to the class of which its image is part.

5 Segmentation

Segmentation involves the grouping of pixels, where the groupings indicate a semantic relationship. For example, segmenting pixels into background and foreground. The purpose of segmentation is that the found groupings can be used in further algorithms.

5.1 *k-means*

The most simple way of performing segmentation is using the k-means algorithm. First, we encode all pixels into a feature space, after which we apply k-means to the features. Then, if two pixels are part of the same cluster, they are part of the same grouping in the segmentation.

Examples of features are RGB values or filter bank responses.[9] However, these do not give us spatial coherence, because spatial information is not taken into account. If there is a lot of noise, the groupings may not be very good. Thus, we want pixels that are spatially close to have a greater chance of being in the same grouping. The way to do this with k-means is to add two dimensions to the features, x and y position. This enforces spatial coherence.

[9] Apply 24 common filters by convolution.

The advantage of k-means is that it is simple and fast. However, it has many problems,

- There is no way of knowing what the value of k should be;
- k-means is sensitive to initial centers;
- k-means is sensitive to outliers;
- It only detects spherical clusters, because points are assigned to clusters by their squared distance.

5.2 *Mixture of Gaussians*

Mixture of Gaussians (MoG) solves the problem of k-means' sensitivity to outliers. Instead of treating the pixels as a bunch of points, we will assume that they are all generated by sampling a continuous function. Then, we will be able to remove outliers, because their likelihood will be very low.

MoGs assume that the data is generated by k weighted d-dimensional Gaussians with means $\boldsymbol{\mu}_b \in \mathbb{R}^d$, covariance matrices $\boldsymbol{\Sigma}_b \in \mathbb{R}^{d\times d}$, and weights α_b. The likelihood of observing $\boldsymbol{x}$ is the following,

$$p(\boldsymbol{x} \mid \boldsymbol{\theta}) = \sum_{b=1}^{k} \alpha_b \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b).$$

We then want to optimize the model w.r.t.

$$\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\mu}_1 & \cdots & \boldsymbol{\mu}_k & \boldsymbol{\Sigma}_1 & \cdots & \boldsymbol{\Sigma}_k & \alpha_1 & \cdots & \alpha_k \end{bmatrix},$$

which is done with the *expectation-maximization* (EM) algorithm. EM works by estimating the "ownership" each Gaussian has over the points, given the current estimate of $\boldsymbol{\theta}$ (E-step). Then, we update $\boldsymbol{\theta}$ given the "ownership" (M-step). This is done iteratively.

In more details, given the current estimate $\boldsymbol{\theta}$, the E-step computes the probability that point $\boldsymbol{x}$ is in blob b, which we call the "ownership" of b over $\boldsymbol{x}$,

$$p(b \mid \boldsymbol{x}, \boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b) = \frac{\alpha_b p(\boldsymbol{x} \mid \boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b)}{\sum_{b'=1}^{k} \alpha_{b'} p(\boldsymbol{x} \mid \boldsymbol{\mu}_{b'}, \boldsymbol{\Sigma}_{b'})}.$$

Given the "ownership" information, the M-step computes the new means, covariance matrices, and weights,

$$\begin{aligned}
\alpha'_b &= \frac{1}{n} \sum_{i=1}^{n} p(b \mid \boldsymbol{x}_i, \boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b) \\
w_i^{(b)} &= \frac{p(b \mid \boldsymbol{x}_i, \boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b)}{\sum_{j=1}^{n} p(b \mid \boldsymbol{x}_j, \boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b)} \\
\boldsymbol{\mu}'_b &= \sum_{i=1}^{n} w_i^{(b)} \boldsymbol{x}_i && \text{weighted mean} \\
\boldsymbol{\Sigma}'_b &= \sum_{i=1}^{n} w_i^{(b)} (\boldsymbol{x}_i - \boldsymbol{\mu}_b)(\boldsymbol{x}_i - \boldsymbol{\mu}_b)^\top && \text{weighted variance.}
\end{aligned}$$

Then, after running the EM algorithm, we assign each pixel to the blob with the highest "ownership" over the pixel.

The advantage of this approach is that it is probabilistic and can predict new data points, because it is generative. This also has the advantage that we can detect outliers. Furthermore, it can be stored in $\mathcal{O}(kd^2)$, which is quite compact. The problems are that we still need to know the number of components k, and we need a good initialization.[10]

[10] It is often a good idea to start from the output of k-means.

5.3 *Mean-shift*

Mean-shift is an algorithm that does not require a predetermined amount of clusters, like k-means and MoG. Instead, it localizes the histogram modes of the data. Then, all identified modes are considered a cluster. It works by starting from random points and iteratively moving toward the center of mass of the window around the point. This will cause the algorithm to move toward high density areas. Ultimately, all points that move toward the same area, *i.e.* same mode, are considered a cluster.

The iterative update rule is the following,

$$f(\boldsymbol{x}) = \frac{1}{\sum_{i=1}^{n} k(\boldsymbol{x}, \boldsymbol{x}_i)} \sum_{i=1}^{n} \boldsymbol{x} k(\boldsymbol{x}, \boldsymbol{x}_i),$$

where $\boldsymbol{x}$ is the current point, k is a distance function, and $f(\boldsymbol{x})$ is the new point. Intuitively, it is a weighted average of its distance to all

other points, which makes it move toward high density areas. The attraction basin is the region for which all trajectories lead to the same mode, and all data points in the same attraction basin of a mode are considered a cluster.

The advantage of this method is that it does not require a predetermined amount of clusters. Another advantage is that it does not assume any prior shape on data clusters. Furthermore, it has just a single parameter h, which is the window size, so it has a physical meaning. However, the selection of h is not trivial, and the method is computationally quite expensive.

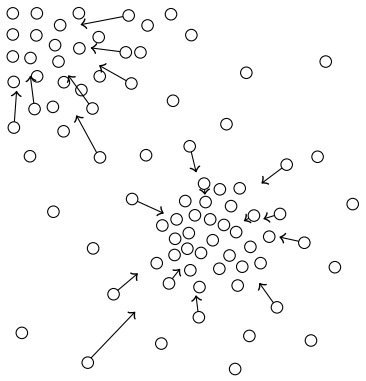


Figure 8. Iteration of mean-shift where points move toward a mode.

5.4 *Hough transform*

Remark 1. This is not an algorithm for segmentation, but rather for finding features.

The *Hough transform* uses the structure of shapes to recognize them in an image. First, we have to find the edges of the objects in the image, so we can fit shapes to them. Now, let's say that we want to detect straight lines for example.[11] We use the following equation for a line,[12]

$$x\cos\theta + y\sin\theta = \rho.$$

Then, we represent all lines through all the points on the edges in a parameter space. *E.g.*, the point $[0, 1]$ lies on all lines with $\rho = 1$, thus all lines in this point are repsented by the line $\rho = 1$ in parameter space. Then, we need to identify peaks in the parameter space. These peaks make up the geometry of the objects in the image.

[11] These are important, because of their omnipresence in man-made scenes.

[12] We use this instead of $y = ax + b$, because this has a bounded parameter domain.

The advantage of this approach is that it can be done for any shape, such as a circle, but the problem is that as the parameters grow, it gets exponentially more expensive.

5.5 *Interactive segmentation*

In *interactive segmentation*, we assume that we also have information from the user about the segmentation. *I.e.*, the user has identified some pixels as being part of some grouping. We can use this information to provide a better segmentation.

In this approach, we will assume that the pixels of the image form a *Markov random field* (MRF). We assume that there is a defined relationship $\psi(\boldsymbol{x}_i, \boldsymbol{x}_j)$ between adjacent pixels and a defined relationship $\phi(\boldsymbol{x}_i, y_i)$ between pixels and their hidden state (label). The joint prob-

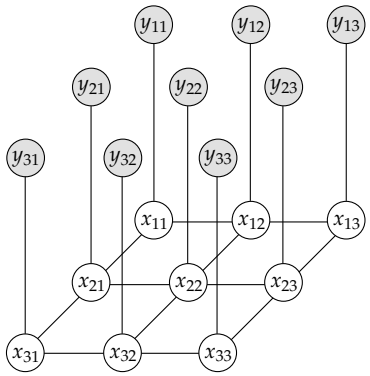


Figure 9. Markov random field.

ability of the field is then defined as the following,

$$p(\boldsymbol{X}, \boldsymbol{y}) = \prod_i \phi(\boldsymbol{x}_i, y_i) \prod_{i,j} \psi(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

We then want to minimize the negative log-likelihood,

$$E(\boldsymbol{X}, \boldsymbol{y}) = -\sum_i \phi(\boldsymbol{x}_i, y_i) - \sum_{i,j} \psi(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

We call this the energy function. We can then use an inference algorithm such as Gibbs sampling to minimize the energy function. However, the most popular one is GraphCut.

The idea of *GraphCut* is to turn the MRF into a source-sink graph, where the edge weights are defined by ϕ and ψ. Let's say we want to segment the image into foreground/background[13] and we have input from the user about some pixels that are background and some that are foreground. We now want to know how to assign the surrounding pixels. First, we convert all the known background pixels into sources and foreground pixels into sinks in the source-sink graph. Then, we find the minimum cost cut that separates the background from the foreground and label the pixels appropriately. This enforces spatial coherence.

[13] If there are more than 2 possible labels, there will not be a unique global solution.

We can also minimize the energy function further by iteratively alternating between using GraphCut and fitting an MoG. Starting from the user-provided bound, each iteration, we get a new bound from GraphCut, which is used to refine the MoG, which is used to refine the bounds with GraphCut, etc.

5.6 *Learning-based approaches*

In fully supervised learning, we assume we have access to a training dataset of images with full segmentation mask as labels. Thus, whatever type of segmentation that will be done by a model will depend on the training data. If the training data consists of background/-foreground segmentation, the model will do background/foreground segmentation.

Models are evaluated with the intersection over union (IoU) metric (see Figure 10),

$$\frac{1}{C} \sum_{i=1}^{C} \frac{n_{ii}}{n_i + \sum_{j=1}^{C} n_{ji} - n_{ii}},$$

where C is the amount of labels, n_i is the amount of pixels of label i, and n_{ij} is the amount of pixels of class i predicted to have class j.

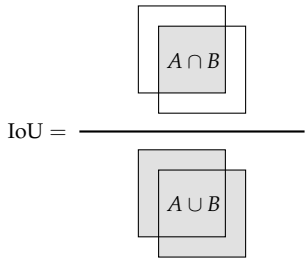


Figure 10. Illustration of the intersection over union metric.

Like in unsupervised approaches, we first need to extract features of the pixels, which can be learned. These features are then given as input to the models, which are optimized to predict the training labels.

k-nearest neighbors. The simplest learning-based approach is k-nearest neighbors (kNN). In kNN, every point is labeled to be the mode of its k nearest neighbors in the training data. The "nearest" neighbors refer to the points in the feature space with the smallest distance to the current feature vector, not the nearest point in the image.

Transfer learning. Transfer learning is a method that does not require a large dataset.[14] In transfer learning, the weights are first learned on a related task, such as classification, and dataset (pretraining). Then, we initialize our segmentation model with these weights as much as possible and train on the actual dataset (finetuning). This makes the assumption that pretrained weights are easier to train for segmentation than a random initalization.

[14] Which is good, because, in segmentation, training data is very expensive.

Hypercolumns. It is common to pretrain on a classification task (one label per image) and then augmenting the model for segmentation (one label per pixel). Hypercolumns do this by taking the pixel activations for each pixel at each layer of the classification network, concatenating them, and calling that the pixel's embedding (see Figure 11).[15] This embedding is then used as input to a pixel classification network that actually predicts the grouping of the pixel.

[15] For convolutional layers, this means taking the output of the convolution at that pixel, and, for downsampling layers, multiple pixels get mapped to the same value.

This implicitly enforces spatial coherence, because convolutional layers work locally. Thus, close activations are closely related. The results of hypercolumns are pretty good, but the boundaries are very coarse, due to the downsampling layers of the classification model, which cause many features to be the same value for close pixels.

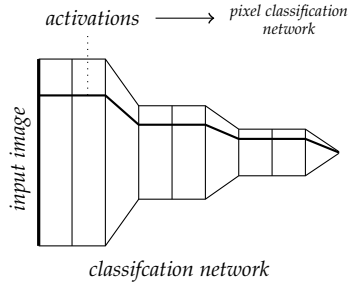


Figure 11. Hypercolumns.

Fully convolutional networks. In fully convolutional networks (FCN), we change the usually used linear layers of the classification network to 1×1 convolutional layers. This allows for pretraining on a classification task, because linear layers and 1×1 convolutional layers have the same parameters. Then, at the end of the model, the images are not turned into vectors, but rather feature maps. Then dependent on how much the images were downsampled by the model, we upsample the feature map from different layers of the nerwork to be of same size as the input image with a transposed convolution. Now, we have features per pixel that can be used to predict the label per pixel with 1×1 convolution.

Refinement with conditional random fields. The outputs of FCNs are not sharp. But, we can use an additional refinement step with a conditional random field (generalization of Markov random field). We can use the FCN output as the unary term ϕ and an edge-aware pairwise term ψ to refine the segmentation.

U-nets. This model is similar to FCNs, but rather than doing the upsampling in a single transposed convolutional layer, we use the same amount of upsampling layers as downsamping layers. Additionally, U-nets also use skip-connections between downsampling and upsampling layers of the same level.

Dilated convolutions. In computer vision, we need convolutional layers to process an input image for a task, and downsampling layers to make the receptive field large. However, because the convolutional layers work very locally (usually 3×3 or 5×5), we miss out on contextual information, and because the downsampling layers remove data, we lose information. We want a layer that does not downsample, but still has a large receptive field. The dilated convolution is a generalization of the convolutional layer that skips values for an exponentially large receptive field without a change in the amount of weights (see Figure 12). Thus, we can still pretrain a classification network and reuse the weights in the dilated layers. Another advantage of not needing downsampling layers is that we get an output for every pixel without needing to upsample, which is what we want in segmentation.

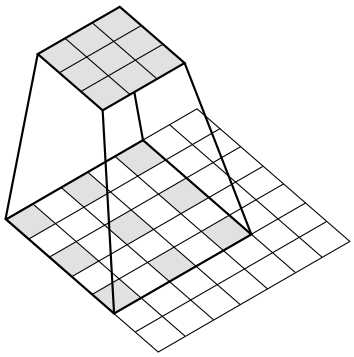


Figure 12. Dilated convolution

However, simply only using dilated layers causes artifacts, because the learned filters are often discontinuous. To mitigate this, we can add additional convolutional layers to the beginning and end of the model to smooth out the artifacts. This results in much better segmentations.

6 *Object detection*

In *object detection*, we want to detect objects within an environment. We are given an image as input, and the output should be a set of objects with their location within the image.

6.1 *Detection via classification*

The most naive idea is to run a classification model on every patch within the image. We can use all previously discussed idea for the representation of the patches, such as a gradient histogram or pixel intensities.

6.2 *Boosting with weak classifiers*

An example of a weak classifier is the rectangle filter. It is essentially a large gradient calculation, where we compute the sum of one patch minus the sum of another next to it. We can compute these very quickly using a sliding window approach, for which we only need to compute a cumulative sum over the integral image in $\mathcal{O}(H \cdot W)$. Then, we can compute the sum of any patch in $\mathcal{O}(1)$. We then classify the patch based on the difference of sums of the patches.

This will result in a single very weak classifier, but if we combine a lot of them, we can use boosting to combine them linearly to be a strong classifier. This is fast, because each weak classifier is very simple. Boosting works by iteratively finding the weak classifier that achieves the lowest weighted training error (weights are initialized to be uniform). Then, we raise the weights of training samples that were misclassified by the best weak classifier. We repeat this, while collecting the best weak classifier every iteration to get a subset of weak classifier that collectively are strong. We weight each weak classifier by their accuracy on the training dataset.

This will result in a subset of weak classifiers that each might have a little better than expected accuracy, but together they form a strong classifier that has good accuracy.

In more details, AdaBoost [Freund et al., 1996] runs for M iterations, each resulting in a weak classifier. Each iteration m, it first trains a new weak classifier h_m that minimizes the weighted error function,

$$J_m = \sum_{i=1}^{n} w_i^{(m)} \mathbb{1}\{h_m(x_i) \neq y_i\}.$$

Then, it estimates the weighted error of this classifier on the dataset,

$$\epsilon_m = \frac{1}{\sum_{i=1}^{n} w_i^{(m)}} \sum_{i=1}^{n} w_i^{(m)} \mathbb{1}\{h_m(x_i) \neq y_i\}.$$

Then, it calculates the weighting coefficient for $h_m(x)$, which is large if the weak classifier is good, and otherwise small,

$$\alpha_m = \log\left(\frac{1-\epsilon_m}{\epsilon_m}\right).$$

Lastly, it updates the weighting coefficients,

$$w_i^{(m+1)} = w_i^{(m)} \exp(\alpha_m \mathbb{1}\{h_m(x_i) \neq t_i\}).$$

The final classifier is then the weighted linear combination, according to α_m, of the weak classifiers.

6.3 Implicit shape model

An implicit shape model (ISM) learns a star-topology structural model, which means that features are considered independent given the center of the object. It learns a visual codebook with displacement vectors that encode where the center of the object must be that this visual word is a part of. For example, we might encode that a car must have two visible wheels. Then, if two wheels point at the same center with their displacement vectors, there is a car object at that center.

6.4 Learning-based approaches

The difficulty in using learning-based approaches in object recognition is that the output has a variable size, since an image may contain any number of objects. Thus, we must design new models that can output a variable amount of outputs. Also, there are an exponential amount of possible regions within an image, which makes classifying every possible patch infeasible.

The solution to this is selective search [Uijlings et al., 2013]. Selective search first segments the image into a large number of regions using a graph-based segmentation technique. Then, it iteratively alternates between collecting segmentation regions as region proposals, and combining segmentation regions into larger ones. This results in a wide spectrum of region proposal sizes. This enables learning-based approaches, because now we only need to classify a small number of patches.

R-CNN. R-CNN [Girshick et al., 2014] works by first extracting approximately 2000 region proposals using selective search. Then, it warps the region proposals to constant-sized images patches and passes them to a classification network, which classifies the region (CNN followed by support vector machine).

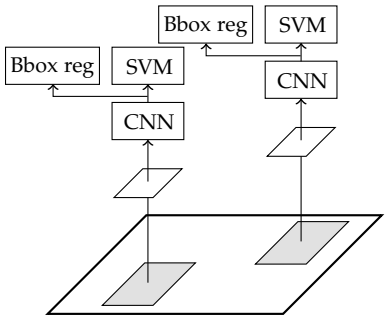


Figure 13. R-CNN architecture.

The problem with R-CNN is that the training and testing is still slow (47 seconds per test image).

Fast R-CNN. Fast R-CNN [Girshick, 2015] solves the problem of R-CNN by first feeding the entire input image into a CNN to generate a feature map. Then, the region proposals are identified within this feature map. This is then reshaped to a constant-sized representation which is passed to a fully connected layer that outputs a probability distribution over classifications. The reason for that this model is faster is because we only need to run a single CNN, instead of 2000 for each region proposal.

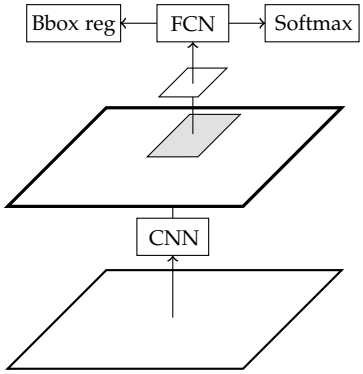


Figure 14. Fast R-CNN architecture.

Faster R-CNN. The largest bottleneck in Fast R-CNN is the selective search, which is very slow. Therefore Ren et al. [2015] came up with Faster R-CNN that uses a region proposal network (RPN) that lets the model learn the region proposals. The RPN works by taking the feature map as input and outputting a bounding box and "objectness" score.

This network has four loss functions that need to be optimized: Region proposal "objectness" classification loss, Region proposal bounding-box regression loss, Region classification loss, and another bounding-box regression loss on the eventual classification network.

You only look once. All R-CNN approaches require a call to a classification network for every region proposal. However, this slows down the algorithm. YOLO [Redmon et al., 2016] predicts bounding boxes and class probabilities in a single pass. It works by dividing the input image into an $S \times S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell then predicts B bounding boxes ((x, y, w, h, c), where c is the confidence score), and C conditional class probabilities.[16] It then picks the boxes with the greatest confidence and removes boxes if their IoU is lower than some threshold.

YOLO is much faster than the R-CNN approaches with a real-time performance of approximately 45 frames per second. A minor problem is that it struggles with small objects and it imposes strong spatial constraints (since each grid cell can only have one class prediction), which limits the number of nearby objects it can predict.

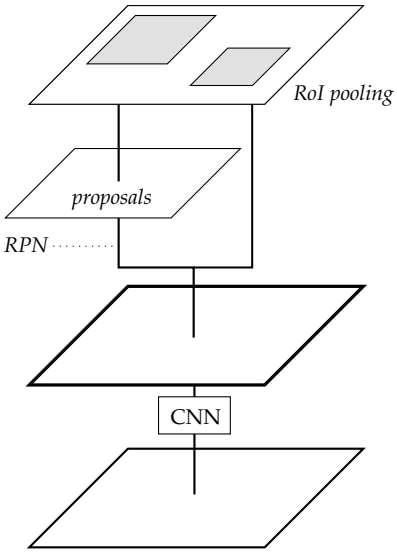


Figure 15. Faster R-CNN architecture.

[16] Thus, the output of the model has dimensionality $S \times S \times (5B + C)$.

7 Tracking

Tracking is the following of the movement of an object (point, region, or template), *i.e.*, we want to find the position of the object in the consecutive frames. Thus, the problem statement is that we want to find the position of an object in frame $t + 1$, given that we know its position in frame t.

To make sure that the object detection location does not "teleport", we can give the models the prior that objects have a constant velocity.[17] We can do this by penalizing points that are not in the direction of the current velocity of the object.

[17] This is especially useful when the camera is stationary.

7.1 Point

If we want to track a point between frames, we can make the assumption that the color of the point will remain the same, and optimize the

following error function (one-dimensional case),

$$\begin{aligned} E(h) &= (\mathbf{I}[x,t]) - \mathbf{I}[x+h,t+1])^2 \\ &\approx (\mathbf{I}[x,t] - \mathbf{I}[x,t+1] - h\mathbf{I}_x[x,t])^2 \\ \frac{\partial}{\partial h}E(h) &= -2\mathbf{I}_x[x,t](\mathbf{I}[x,t] - \mathbf{I}[x,t+1] - h\mathbf{I}_x[x,t]). \end{aligned}$$

At the minimum, where the derivative is 0, we get the following,

$$h = \frac{\mathbf{I}[x,t+1] - \mathbf{I}[x,t]}{\mathbf{I}_x[x,t]}.$$

However, if the gradient is 0, *i.e.*, when the image region is flat, $\mathbf{I}_x[x,t]$ will be 0 and thus we have no information about which direction the point moved in. This is very similar to the problem in optical flow, called the aperture problem.

Furthermore, we assume to always move toward the closest minimum. But, the image is non-convex in general, which has multiple solutions. This can be solved by a high framerate, because then the movement between frames is small.

Like in optical flow, in two dimensions, we need more constraints to solve for the displacement. And, like in optical flow, we can solve this by either solving it globally (Horn-Schunck) or locally (Lucas-Kanade).

7.2 Template

In template tracking, we want to track an object between frames, represented by a bounding box, which we call its template. The most naive solution is to simply compare pixel intensities and minimizing the following error function,

$$E(u,v) = \sum_{x,y}(\mathbf{I}[x+u,y+v] - \mathbf{T}[x,y])^2,$$

where $\mathbf{T}$ is the template. However, the object in the template might make a transformation such as a rotation or scaling. We could also parameterize those transformations, instead of only a translation, by generalizing the previous error function to any transformation family W,

$$E(\boldsymbol{p}) = \sum_{\boldsymbol{x}}(\mathbf{I}[W(\boldsymbol{x};\boldsymbol{p})] - \mathbf{T}[\boldsymbol{x}])^2,$$

where $\boldsymbol{p}$ parameterizes a "warp" W. However, we cannot directly solve for $\boldsymbol{p}$, thus we iteratively improve it by updating it with some δ_p,

$$
\begin{aligned}
&= \sum_{\boldsymbol{x}} \big(\mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p} + \delta_p)] - \mathbf{T}[\boldsymbol{x}]\big)^2 \\
&\approx \sum_{\boldsymbol{x}} \big(\mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] + \delta_p \boldsymbol{\nabla}_{\boldsymbol{p}} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] - \mathbf{T}[\boldsymbol{x}]\big)^2 && \text{Taylor approx} \\
&= \sum_{\boldsymbol{x}} (\mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] + \delta_p \boldsymbol{\nabla}_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \boldsymbol{\nabla}_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p}) - \mathbf{T}[\boldsymbol{x}])^2 && \text{chain rule}
\end{aligned}
$$

$$
\frac{\partial}{\partial \delta_p} E(\boldsymbol{p}) = \sum_{\boldsymbol{x}} 2\Big(\boldsymbol{\nabla}_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \boldsymbol{\nabla}_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p})\Big)^\top \Big(\mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] + \delta_p \boldsymbol{\nabla}_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \boldsymbol{\nabla}_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p}) - \mathbf{T}[\boldsymbol{x}]\Big).
$$

We can then solve for δ_p by setting the gradient to o,

$$
\delta_p = \boldsymbol{H}^{-1} \sum_{\boldsymbol{x}} \Big(\boldsymbol{\nabla}_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \boldsymbol{\nabla}_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p})\Big)^\top (\mathbf{T}[\boldsymbol{x}] - \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})]),
$$

with Hessian matrix $\boldsymbol{H}$ defined as follows,

$$
\boldsymbol{H} = \sum_{\boldsymbol{x}} \Big(\boldsymbol{\nabla}_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \boldsymbol{\nabla}_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p})\Big)^\top \Big(\boldsymbol{\nabla}_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \boldsymbol{\nabla}_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p})\Big).
$$

We iteratively update the warp by δ_p until convergence.

However, this method is not robus to image noise, since it assumes that pixel intensities remain the same between frames. Furthermore, some three-dimensional transformations are impossible to parameterize as a two-dimensional warp of an image. For example, a person turning around w.r.t. the camera cannot be parameterized as a two-dimensional warp.

7.3 Tracking by detection

We can also track by detecting keypoints in each frame and minimizing the distance to the feature descriptor of the point we are tracking. This does not require the assumption that the image intensities remain the same, and can track large displacements.

We can also use this to track a region by matching keypoint descriptors of the template with the next frame. Then, removing outliers with *e.g.* RANSAC. The bounding box in the next frame is then the box that encapsulates all points that are left.

However, in many cases, we do not have an exact template, but we want to track any object of a specific type. This can be done by detecting the objects independently in each frame and then associating the detections over time with a bipartite matching algorithm. This is especially important when there are multiple objects in the scene.

7.4 Online learning

Often, the template changes throughout frames, but, until now we have assumed that the template is constant. However, it is more realistic to assume that the appearance of the tracked object changes. Online learning accounts for this by collecting all detected features of the tracked object throughout the frames and the background. Then, it uses these to train the detector every frame. Thus, the detector improves every frame, and updates it representation of the object.

The advantage of this is that it is robust to changes to the environment, *e.g.*, if we go from a dark to a bright environment. However, the disadvantage is that the representation of the object can gradually drift to something else. But, this can be avoided by not allowing the model to drift too far from our initial template with additional constraints.

8 Projective geometry

SUMMARY OF NOTATION

$\boldsymbol{x}$	two-dimensional vector in real space $\mathbb{R}$
$\tilde{\boldsymbol{x}}$	two-dimensional vector in projective space $\mathbb{P}$
$\overline{\boldsymbol{x}}$	two-dimensional vector in projective space $\mathbb{P}$ with $w = 1$
$\boldsymbol{X}$	three-dimensional vector in real space $\mathbb{R}$
$\tilde{\boldsymbol{X}}$	three-dimensional vector in projective space $\mathbb{P}$
$\overline{\boldsymbol{X}}$	three-dimensional vector in projective space $\mathbb{P}$ with $w = 1$

Using *projective geometry*, we can model the imaging process of a perspective camera,[18] because it satisfies the following constraints,

- Straight lines must be mapped to straight lines;
- The size of an object must be inversely proportional to its distance from the camera;
- We must be able to represent points at infinity to model vanishing points.

In projective geometry, the homogeneous coordinate space is used,

$$\tilde{\boldsymbol{x}} \propto \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \in \mathbb{P}^2,$$

[18] And various other transformations that Euclidean space cannot, such as translation.

which can be converted to Euclidean space,

$$\tilde{\boldsymbol{x}} \mapsto \begin{bmatrix} x/w \\ y/w \end{bmatrix} \in \mathbb{R}^2,$$

where w is the last entry of $\tilde{\boldsymbol{x}}$. Homogeneous coordinates with $w = 0$ are called *ideal points* and serve as a direction toward infinity, rather than points.

In this space, lines can be represented as the following,

$$\boldsymbol{\ell} = \begin{bmatrix} a & b & c \end{bmatrix}^\top,$$

where the points on this line satisfy $\boldsymbol{\ell}^\top \tilde{\boldsymbol{x}} = 0$. Now, if we want to find the point where two lines $\boldsymbol{\ell}, \boldsymbol{\ell}'$ intersect, we need to find a point that satisfies $\boldsymbol{\ell}^\top \tilde{\boldsymbol{x}} = \boldsymbol{\ell}'^\top \tilde{\boldsymbol{x}} = 0$. This is exactly the cross product,

$$\tilde{\boldsymbol{x}}_{\text{intersection}} = \boldsymbol{\ell} \times \boldsymbol{\ell}'.$$

We can find the line between two points $\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2$ in the same way,

$$\boldsymbol{\ell} = \tilde{\boldsymbol{x}}_1 \times \tilde{\boldsymbol{x}}_2.$$

In three dimensions, we can also represent planes,

$$\boldsymbol{\pi} = \begin{bmatrix} a & b & c & d \end{bmatrix}^\top.$$

A point lies on the plane $\boldsymbol{\pi}$ if and only if $\boldsymbol{\pi}^\top \boldsymbol{X} = 0$. If we want to find a plane from three points, we need to solve

$$\boldsymbol{\pi}^\top \begin{bmatrix} \boldsymbol{X}_1 & \boldsymbol{X}_2 & \boldsymbol{X}_3 \end{bmatrix} = \mathbf{0}.$$

We can construct lines as the intersection of two planes, or construct points as the intersection of three planes.

8.1 Transformations

Translations in Euclidean geometry are non-linear, but by using homogeneous coordinates it becomes a linear operation,

$$\begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ w \end{bmatrix}.$$

Furthermore, we can make the following transformations,

- Rigid transformations (rotation and translation) with the following matrix,[19]

$$\begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}.$$

[19] This matrix has 3 degrees of freedom: t_x, t_y, θ.

- Similarity transformations (rotation, translation, and scaling) with the following matrix,[20]

$$\begin{bmatrix} s\cos\theta & -s\sin\theta & t_x \\ s\sin\theta & s\cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}.$$

[20] This matrix has 4 degrees of freedom: t_x, t_y, θ, s.

- Affine transformations (rotation, translation, scaling, and shearing) with the following matrix,[21]

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ 0 & 0 & 1 \end{bmatrix}.$$

[21] This matrix has 6 degrees of freedom: $a_{1:6}$

- Projective transformations with the following matrix,[22]

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

[22] This matrix has 8 degrees of freedom, because it is specified up to scale.

8.2 Homography

Suppose that we have point correspondences between two images. Now, we want to find the projective transformation $\boldsymbol{H}$ that relates the two images. For this, we use *direct linear transformation*. We have the following equations,[23]

$$\begin{aligned} \lambda_i \bar{\boldsymbol{x}}_i' &= \boldsymbol{H}\bar{\boldsymbol{x}}_i \\ \begin{bmatrix} \lambda_i x_i' \\ \lambda_i y_i' \\ \lambda_i \end{bmatrix} &= \begin{bmatrix} h_{11}x_i + h_{12}y_i + h_{13} \\ h_{21}x_i + h_{22}y_i + h_{23} \\ h_{31}x_i + h_{32}y_i + h_{33} \end{bmatrix}. \end{aligned}$$

[23] The homography is constrained to map $\boldsymbol{x}_i$ to $\boldsymbol{x}_i'$.

Now, when we revert them back to the real coordinate system, we get the following,

$$\begin{aligned} \lambda_i &= h_{31}x_i + h_{32}y_i + h_{33} \\ \lambda_i x_i' &= h_{11}x_i + h_{12}y_i + h_{13} \\ &\implies h_{11}x_i + h_{12}y_i + h_{13} - h_{31}x_i'x_i - h_{32}x_i'y_i - h_{33}x_i' = 0 \\ \lambda_i y_i' &= h_{21}x_i + h_{22}y_i + h_{23} \\ &\implies h_{21}x_i + h_{22}y_i + h_{23} - h_{31}y_i'x_i - h_{32}y_i'y_i - h_{33}y_i' = 0. \end{aligned}$$

Singular value decomposition (SVD) decomposes a matrix $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are unitary matrices, and $\boldsymbol{\Sigma}$ is a diagonal scaling matrix. The SVD is very useful for solving (overdetermined) equations of the following form,

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}.$$

Now, when solving such equations, we want to minimize $\|\boldsymbol{A}\boldsymbol{x}\|$ subject to $\|\boldsymbol{x}\| = 1$. We can solve for this by using SVD,

$$\begin{aligned} &\min \|\boldsymbol{A}\boldsymbol{x}\| && \|\boldsymbol{x}\| = 1 \\ &\min \|\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x}\| && \|\boldsymbol{x}\| = 1 \\ &\min \|\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x}\| && \|\boldsymbol{x}\| = 1 \\ &\min \|\boldsymbol{\Sigma}\boldsymbol{b}\| && \|\boldsymbol{V}\boldsymbol{b}\| = 1. \end{aligned}$$

Then, $\|\boldsymbol{\Sigma}\boldsymbol{b}\|$ is minimized if

$$\boldsymbol{b} = \begin{bmatrix} 0 & \cdots & 0 & 1 \end{bmatrix}^\top.$$

Thus, the solution is the following,

$$\boldsymbol{x} = \boldsymbol{V}\begin{bmatrix} 0 & \cdots & 0 & 1 \end{bmatrix}^\top = \boldsymbol{V}_n,$$

where $\boldsymbol{V}_n$ is the last column vector of $\boldsymbol{V}$.

In matrix form, we can set this up as the following for n points,

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 & -y'_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_nx_n & -x'_ny_n & -x'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_nx_n & -y'_ny_n & -y'_n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \mathbf{0}.$$

If we stack $n \geq 4$ points, we can use SVD to find the $\boldsymbol{h}$ vector that minimizes the algebraic distance. However, this does not have a geometric meaning. We would like a cost function that minimizes the distortion of the points. An example of this is the symmetric transfer error,

$$\hat{\boldsymbol{H}} = \underset{\boldsymbol{H}}{\operatorname{argmin}} \sum_{(\boldsymbol{x},\boldsymbol{x}')\in\mathcal{D}} d(\overline{\boldsymbol{x}}, \boldsymbol{H}^{-1}\overline{\boldsymbol{x}}')^2 + d(\overline{\boldsymbol{x}}', \boldsymbol{H}\overline{\boldsymbol{x}})^2,$$

where $\mathcal{D}$ is the collection of point correspondences, and $d(\cdot,\cdot)$ is a distance function in real space. In practice, we use the algebraic solution as a starting point and then refine it to minimize the geometric error.

9 *Camera model*

The pinhole camera is a mathematical model of how perspective cameras work. It assumes that the pinhole is infinitely small, so it only lets one lightray through per point in the space.[24]

[24] If the pinhole camera is not infinitely small, the image would become blurry. However, in reality, an infinitely small pinhole is impossible and a very small pinhole would not let enough light through to see anything. The solution to this are lenses that map all light coming from a point to its corresponding point in the image.

9.1 *Internal camera matrix*

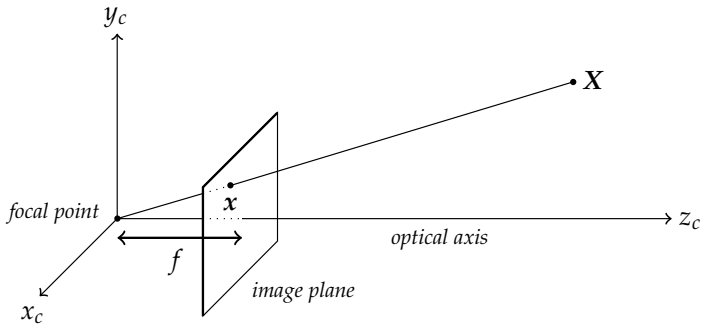


Figure 16. Illustration of how the intrinsic camera matrix works.

The coordinate frame of the pinhole camera is chosen such that the origin is in the center of the pinhole. The image plane behind the

camera is in the plane where $z_c = -f$, where f is the focal distance. The z-axis is called the principal axis and goes in the direction where the camera is pointing toward. If we know a three-dimensional point $\boldsymbol{X}_c$, then it is possible to calculate the image coordinate $\boldsymbol{x}$ of that point projected onto the image plane:

$$x = -f\frac{x_c}{z_c}, \quad y = -f\frac{y_c}{z_c}.$$

The minus-sign indicates that the projected image is upside down. To get rid of this mirroring, we use a virtual pinhole camera in which the image plane is put at $z_c = f$, *i.e.*, in front of the pinhole. In this model, we have

$$x = f\frac{x_c}{z_c}, \quad y = f\frac{y_c}{z_c}.$$

Using homogeneous coordinates, we can model this with the following,

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} f_x & s & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{\boldsymbol{X}}_c = \boldsymbol{K}\tilde{\boldsymbol{X}}_c.$$

In practice, we often set $f_x = f_y$ and $s = 0$. The translation $\boldsymbol{c}$ moves the origin of the image from the middle to the top left of the image plane, which makes computation easier.[25]

[25] Computation becomes easier then, because most libraries, such as `NumPy` and `PyTorch`, start the coordinates from the top left.

9.2 *External camera matrix*

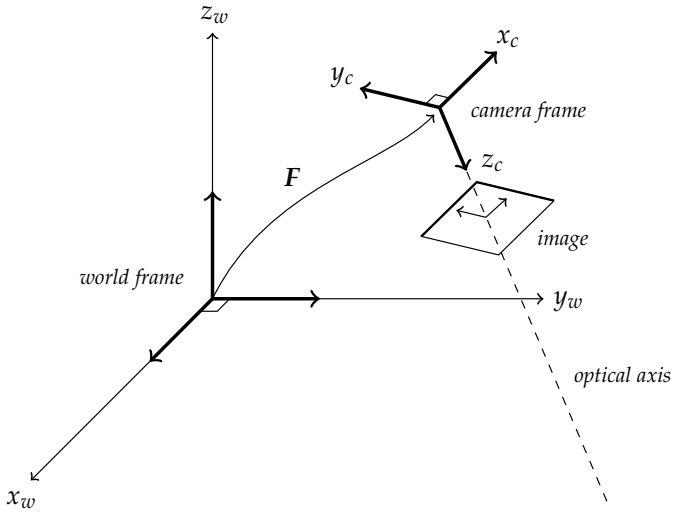


Figure 17. $\boldsymbol{F}$ is the rigid transformation that the camera made within the world space. To convert world coordinates such that the camera is the origin, we transform them by $\boldsymbol{F}^{-1}$.

The camera will generally not be in the origin of the three-dimensional world space. So, we need the transformation that moved the camera from the origin to its point. Let

$$\boldsymbol{F} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^\top & 1 \end{bmatrix}$$

be this transformation, then the projection of a point $\boldsymbol{X}_w$ in world space onto the image plane is given by the following,[26]

$$\tilde{\boldsymbol{x}} = \boldsymbol{K}\boldsymbol{F}^{-1}\overline{\boldsymbol{X}}_w.$$

We denote this matrix by $\boldsymbol{P} = \boldsymbol{K}\boldsymbol{F}^{-1}$.

[26] We take the inverse of $\boldsymbol{F}$, because we want to move the origin of the points to the origin of the camera.

10 *Epipolar geometry*

Remark 2. In this section, all points are in projective space. They are not denoted by $\tilde{\boldsymbol{x}}$ and $\tilde{\boldsymbol{X}}$, but rather by $\boldsymbol{x}$ and $\boldsymbol{X}$.

Epipolar geometry is the geometry of stereo vision, where two cameras view a scene. When two cameras view a scene from distinct positions, there are geometric relations between the 3-dimensional points and their 2-dimensional projections on the image planes. These geometric relations lead to constraints between the image points that we can make use of. These relations are derived based on the assumption that the two cameras are approximated by the pinhole camera model (section 9).

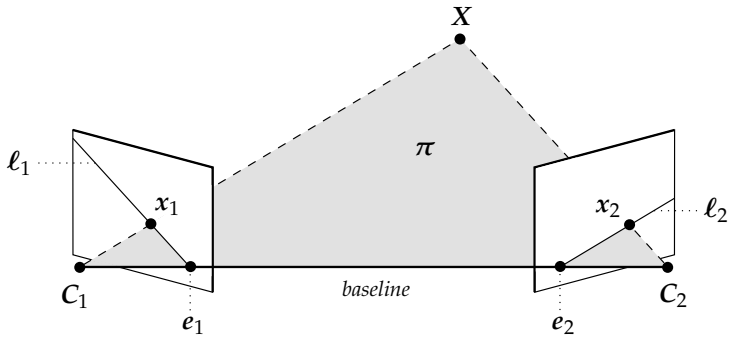


Figure 18. Epipolar geometry.

Figure 18 shows the geometric relationships between the 3-dimensional points and its projections onto the images. $\boldsymbol{X}$ is the 3-dimensional point that is projected onto camera 1 as $\boldsymbol{x}_1$ and camera 2 as $\boldsymbol{x}_2$ (as seen in the pinhole camera model). Together, $\boldsymbol{C}_1$, $\boldsymbol{C}_2$, and $\boldsymbol{X}$ form the *epipolar plane* $\boldsymbol{\pi}$. $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ must also be on this plane. More specifically, the intersections between the epipolar plane and image planes are lines called the *epipolar lines* $\boldsymbol{\ell}_1$ and $\boldsymbol{\ell}_2$. $\boldsymbol{x}_1$ must be on $\boldsymbol{\ell}_1$ and $\boldsymbol{x}_2$ must be on $\boldsymbol{\ell}_2$.

We can also see this from a different perspective. The projections of $\boldsymbol{C}_2$ onto image 1, and $\boldsymbol{C}_1$ onto image 2, are called the *epipoles* $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$.

These are computed as follows,

$$e_1 = P_1 C_2$$
$$e_2 = P_2 C_1,$$

where P_1 and P_2 are the camera matrices of cameras 1 and 2, respectively. The line formed between x_1 and e_1 is the epipolar line (same for x_2 and e_2). This gives us the following constraint,

$$\ell_1 = e_1 \times x_1$$
$$\ell_2 = e_2 \times x_2.$$

Intuitively, this is because both C_1 and C_2 are on the epipolar plane, thus so must be e_1 and e_2. The epipolar lines are the intersection of the epipolar plane with the image planes, and we know two points that are on both, x and e, which form a line.

Notice that x_1 cannot be "behind" e_1, because that would imply that X is behind C_1, which is not the case, because X is visible in image 1. The same holds for camera 2.

10.1 Correspondence geometry

We need to find out in what way x_1 constrains the location of x_2 in image 2, without knowing X. x_1 constrains the point X to be on the projection line from C_1 to x_1, as can be seen in Figures 16 and 19.

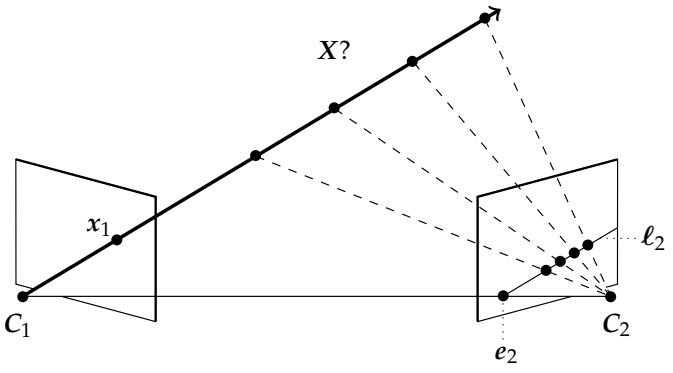


Figure 19. Knowing x_1 constrains its corresponding point x_2 to be on the epipolar line ℓ_2, and not behind e_2, because then X would be behind camera 1, which is not the case.

The key insight into constraining x_2 is that we do not need to know the exact location of X. We only need to find a second point on ℓ_2 to compute it with the cross product (we already know e_2). Furthermore, the projection line emanating from x_1 is on the epipolar plane, thus any point on this projection line is also on the epipolar plane. If we project that point onto image plane 2, we get a second point on ℓ_2. We can compute a point on the projection line by using the pseudo-inverse

$\boldsymbol{P}_1^+$.[27] A 3-dimensional point $\boldsymbol{X}'$ on the projection line between $\boldsymbol{C}_1$ and $\boldsymbol{x}_1$ can be computed as follows,

$$\boldsymbol{X}' = \boldsymbol{P}_1^+ \boldsymbol{x}_1.$$

The epipolar line $\boldsymbol{\ell}_2$ can be computed by the cross product between the epipole $\boldsymbol{e}_2$ and this point on image plane 2,

$$\boldsymbol{\ell}_2 = \boldsymbol{P}_2\boldsymbol{C}_1 \times \boldsymbol{P}_2\boldsymbol{P}_1^+\boldsymbol{x}_1.$$

Definition 1 (Fundamental matrix)**.** The fundamental matrix $\boldsymbol{F}$ is defined as the following,[28]

$$\boldsymbol{F} \doteq [\boldsymbol{P}_2\boldsymbol{C}_1]_\times \boldsymbol{P}_2\boldsymbol{P}_1^+$$
$$\boldsymbol{F} \doteq [\boldsymbol{P}_1\boldsymbol{C}_2]_\times \boldsymbol{P}_1\boldsymbol{P}_2^+.$$

The fundamental matrix relates points in one image plane with lines in the other by the following equation,

$$\boldsymbol{x}_1^\top \boldsymbol{F}\boldsymbol{x}_2 = 0.$$

I.e., $\boldsymbol{x}_1$ must be on the line $\boldsymbol{F}\boldsymbol{x}_2$ and $\boldsymbol{x}_2$ must be on the line $\boldsymbol{x}_1^\top \boldsymbol{F} = \boldsymbol{F}\boldsymbol{x}_1$.

Definition 2 (Essential matrix)**.** The essential matrix $\boldsymbol{E}$ relates points in one image plane with lines in the other by the following equation in the same way as the fundamental matrix (definition 1),[29]

$$\boldsymbol{x}_1^\top \boldsymbol{E}\boldsymbol{x}_2 = 0.$$

The difference is that the essential matrix assumes that the cameras are calibrated. This means that the intrinsic camera parameters are known.

Computing the fundamental matrix. Given that we know 8 point correspondences,[30] we can compute $\boldsymbol{F}$ using direct linear transformation (DLT) derived from $\boldsymbol{x}'^\top \boldsymbol{F}\boldsymbol{x} = 0$,

$$\begin{bmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_nx_n & x'_ny_n & x'_n & y'_nx_n & y'_ny_n & y'_n & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = \mathbf{0}.$$

[27] The pseudo-inverse can be computed with SVD as $\boldsymbol{M}^+ = \boldsymbol{V}\frac{1}{\boldsymbol{\Sigma}}\boldsymbol{U}^\top$.

The cross product matrix is the conversion of a vector to a matrix that would be equivalent to performing a cross product with that vector,

$$[\boldsymbol{a}]_\times \doteq \begin{bmatrix} 0 & a_3 & -a_2 \\ -a_3 & 0 & a_1 \\ a_2 & -a_1 & 0 \end{bmatrix}.$$

[28] The fundamental matrix has 7 degrees of freedom, because it is a 3×3 matrix, is defined up to scale, and $\det(\boldsymbol{F}) = 0$.

[29] The essential matrix has 5 degrees of freedom, because it has the same parameters as the fundamental matrix, minus the focal length and skew.

[30] We assume that all our point correspondences are correct, *i.e.*, they satisfy $\boldsymbol{x}_1^\top \boldsymbol{F}\boldsymbol{x}_2 = 0$.

However, the problem with this is that the values of the matrix grow very large, because of the multiplications. Using DLT, this will always result in the best matrix being very similar to the following,

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

Thus, we need to first normalize the coordinates of the images to be in $[-1, 1] \times [-1, 1]$ before computing $\boldsymbol{F}$.

Singularity constraint. Enforce the rank 2 constraint of $\boldsymbol{F}$. ...

10.2 Camera geometry

10.3 Scene geometry

11 Structure from motion

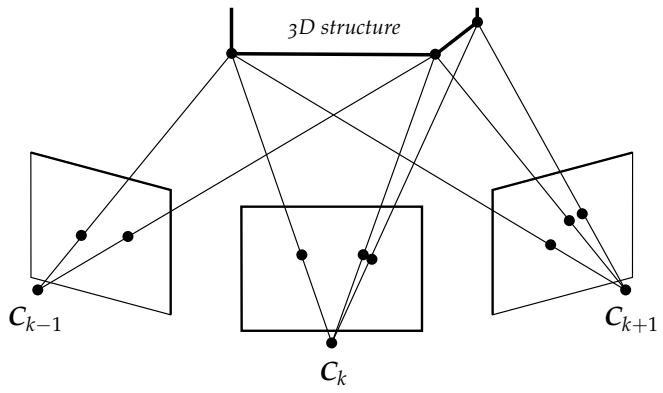


Figure 20. Structure from motion

11.1 RANSAC

TODO

12 Multi-view stereo matching

TODO

References

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

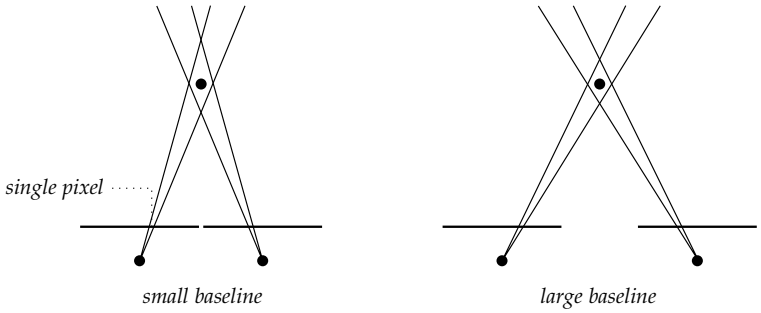


Figure 21. Stereo baseline

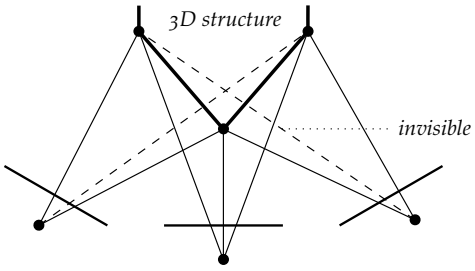


Figure 22. Visibility problem

Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *icml*, volume 96, pages 148–156. Bari, Italy, 1996.

Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

Chris Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Manchester, UK, 1988.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60:91–110, 2004.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning internal representations by error propagation, 1985.

Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International journal of computer vision*, 104:154–171, 2013.