COMPUTER VISION NANODEGREE | Udacity

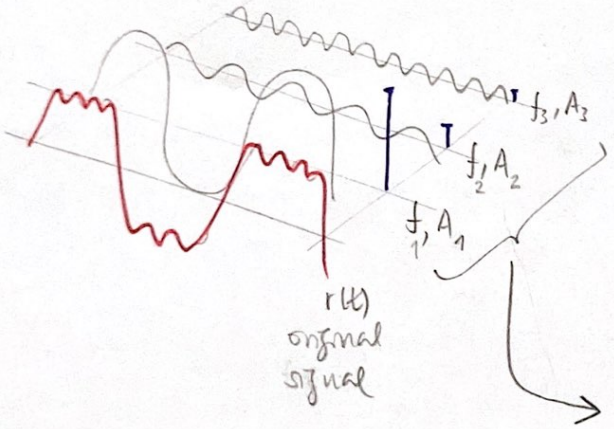
① INTRODUCTION TO COMPUTER VISION

IMAGE REPRESENTATION & CLASSIFICATION (see TXT & notebooks)

- Pixelmaps
- Thresholding
- Color maps / color transformations (HSV)
- Image features (eg. brightness)
- Classification example (& standardization of images): Day and Night image classification
- Image processing & Computer Vision Pipeline

CONVOLUTIONAL FILTERS & EDGE DETECTION

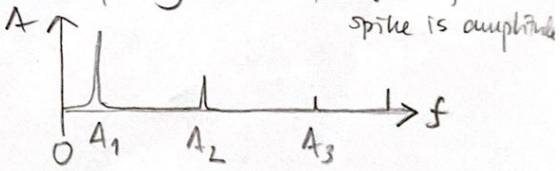
Fourier Transforms



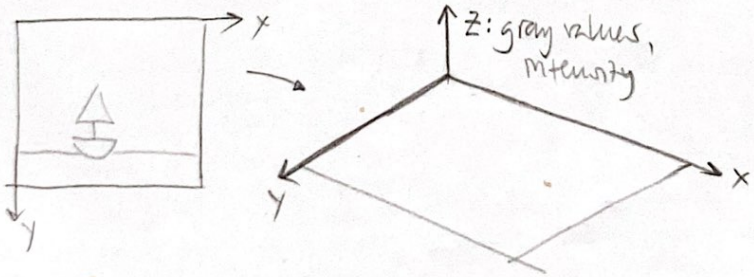
- Signal decomposed as sum of sinoidals with different amplitudes A_i and frequencies f_i:

$$r(t) = \sum A_i \sin(2\pi f_i t + \varphi_i)$$

- Fourier Transform: spikes in frequency domain, height of spike is amplitude



- Image: it's also a varying function but not in time, in 2D space



- an image can be understood as a 2D surface
- it's decomposition elements would have this form:

$$a_i \sin(h \cdot x + k \cdot y)$$

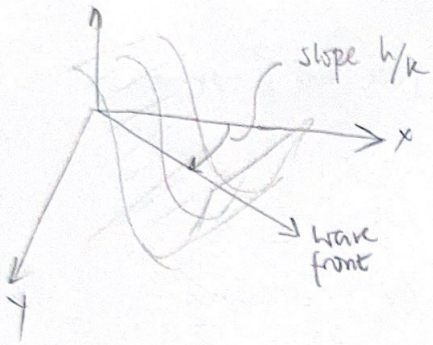
amplitude

frequency (of repetition of the wave in x / y dir)

spatial pixel coords

$$z = \sum a_i \sin(h_i x + k_i y)$$

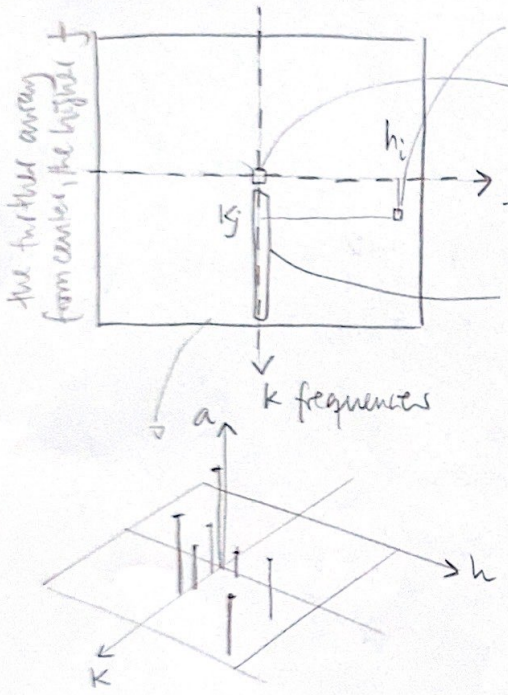
$h_i, k_i \neq 0 \Rightarrow$ wave i travels in angle with slope h_i/k_i (its front)



- Therefore, an image is decomposed as a sum of 2D sine waves.

the amplitude^2 is the intensity

- Interpretation of the FT: the brighter the pixels, the bigger their amplitude



amplitude for frequency pair (h_i, k_j)

$(h,k) = (0,0)$: usually bright: flat plane with no ripples – its intensity/brightness is the average value of the gray values on the image

waves with $h = 0$ frequencies; usually there's many white pixels here (and also on $k=0$), because human environments have many vertical ($h=0$) horizontal ($k=0$) lines

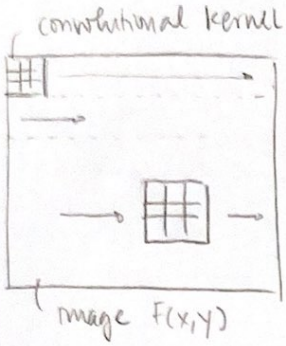
↦ the image is the sum of all these 2D sine waves, with their amplitudes a and frequencies h & k

- Practical tricks

- If FT image brighter in center ⇒ image mostly solid colors, few texture (low f)
- If FT image horizontal line in center ⇒ vertical structures in image, eg human beings!
- " " vertical line " " ⇒ horizontal structures in image

NOTE: in the FT image, each pixel is not linked to the same pixel in the original image... in FT images pixels are frequency pairs and their value is the amplitude associated to the frequency!

Convolution



0	-1	0
-1	4	-1
0	-1	0

Convolution kernel for edge detection

$0 + (-1) + 0 + (-1) + 4 + (-1) + 0 + (-1) + 0 = 0$

weights

the sum of all elements must be 0, because this filter is computing differences between neighbor pixels!

$K * F(x,y)$:

Convolution operator: kernel is applied on all pixels of the image

(x_i, y_i) is the center pixel

K

0	-1	0
-1	4	-1
0	-1	0

$F(x_i, y_i)$

120	140	120
225	220	205
255	250	230

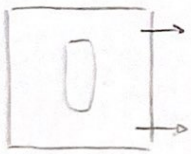
$0 \cdot 120 + (-1) \cdot 140 + 0 \cdot 120$
$+ (-1) \cdot 225 + 4 \cdot 220 + (-1) \cdot 205$
$+ 0 \cdot 255 + (-1) \cdot 250 + 0 \cdot 230 = 60$

that's the value of the output pixel!

- Kernel cannot be overlaid perfectly on edges & corners – what to do?
  - extend: border pixel values extended outwards
  - padding: 0 values (black) set outwards
  - cropping: borders are removed → output image slightly smaller
- Weights: the higher the relative value, the higher the importance
  - their spatial disposition defines the feature we want to remove / enhance.

Sobel Filters

Approximation of image gradient



$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ →  image derivative in X direction (intensity), emphasis of VERTICAL lines

$S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$ →  image intensity derivative in Y dir, emphasis of HORIZONTAL lines

The strongest edges are found with the magnitude:

$|F_x'| = |S_x \cdot F| = \sqrt{Sobel_x^2}$

$|F_y'| = |S_y F| = \sqrt{Sobel_y^2}$

$|F_{xy}'| = \sqrt{Sobel_x^2 + Sobel_y^2}$ → gradient magnitude in both directions

Direction of gradient: $\underbrace{atan(Sobel_y / Sobel_x)}_{\text{angle of lines}}$ → to find lines with an angle upwards...

Lowpass Filters) → usual procedure: ① apply lowpass filter to remove noise ② apply highpass filter to enhance features

They remove high frequency components; usually some sort of averaging is done with pixel values

$$\frac{1}{9}\cdot\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix} \rightarrow \frac{1}{9}\cdot(1+\dots+1) = \underline{1} \quad \text{sum must be 1!}$$

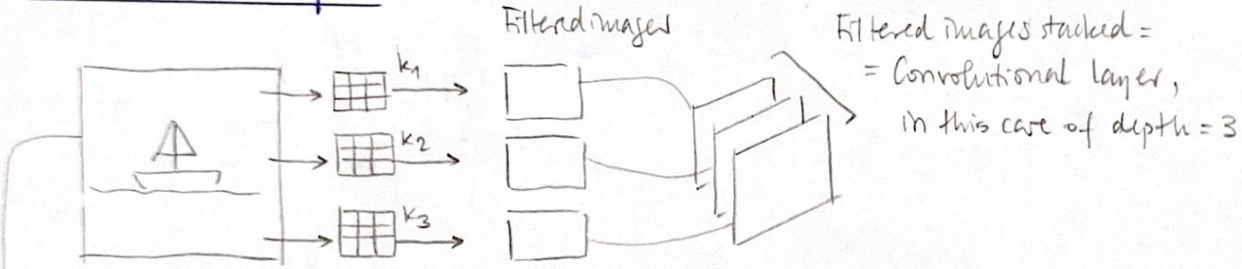
$$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix} \rightarrow \Sigma = 1!$$

→ since it's so common, it has its own function in OpenCV

→ it's basically a weighted sum

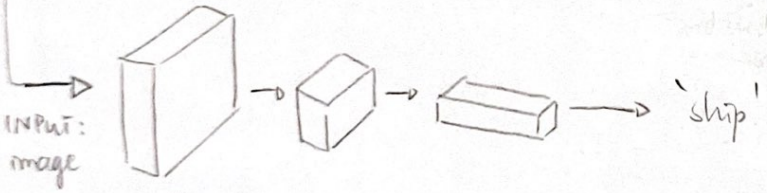
Gaussian Blur: high-freq. removed BUT edges are preserved!

Convolutional Layers



Filtered images stacked = = Convolutional layer, in this case of depth = 3

- A convolutional neural network (CNN) can have several convolutional layers, each with a depth; some other layer types are also possible



convolutional layer | pooling layer | fully connected layer | predicted class

- Weights of convolutional layer are learned during training, but basically they converge to be
  - lowpass filters, eg to compress image by filtering out irrelevant info
  - highpass filters, eg to enhance features like edges

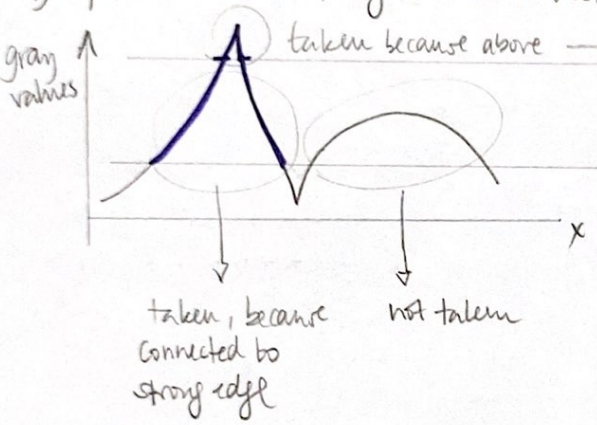
Canny Edge Detector

we face up many issues here

- Edge detection process: grayscale → lowpass → highpass → binary threshold
  - what level of intensity is an edge?
  - how can we consistently represent thick & thin edges?

solutions ↓

- CANNY EDGE DETECTOR: Steps (4)
  1) Filter out noise with Gaussian blur
  2) Sobel X & Y to compute strength & direction of edges
  3) Non-maximum suppression: strongest edges selected and thinned
  4) Hysteresis thresholding to isolate best edges

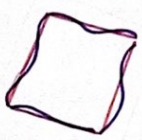


Selecting upper/lower threshold RATIO

- upper/lower ∈ [1, 3] (eg, 2)
- lower: start with middle of gray value range
- the smaller is lower, the more edges detected

Hough Transform → Line Detection as base of Shape Detection

- Most shapes are a collection of line segments
- Hough transform helps detect lines on image



~ original binary edges
~ detected line segments

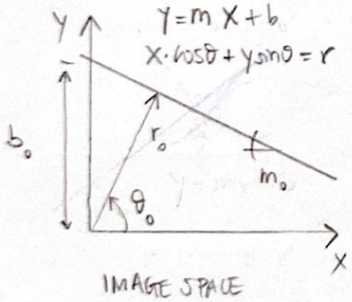
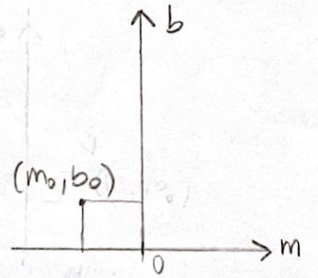


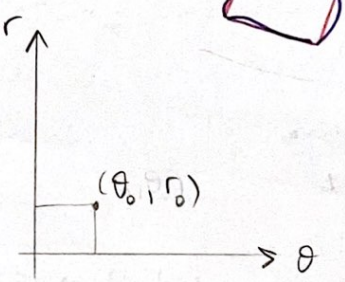
IMAGE SPACE

- we represent a line parametrically
- two options:
  (m, b) cartesian
  (r, θ) polar



HOUGH SPACE = PARAMETER SPACE

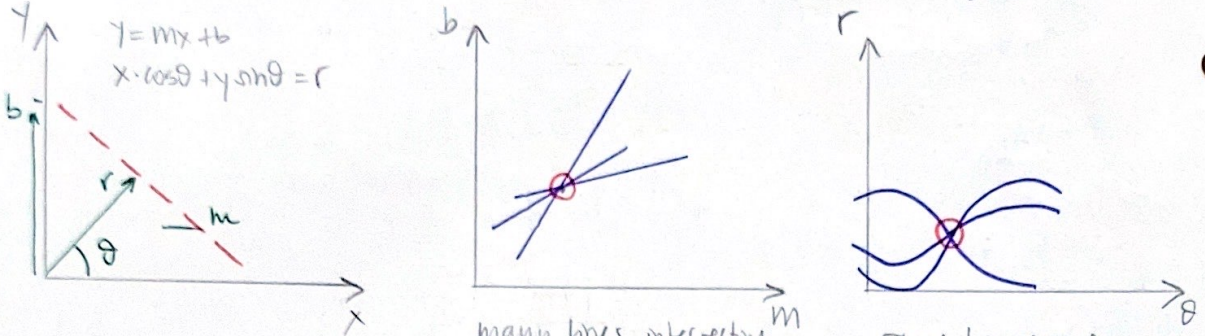
- the parameters of a line are represented
- a line becomes a point!



HOUGH SPACE with polar parameters

- a better representation is with polar coordinates, because otherwise slope of vertical lines $m \to \infty$

- Intersection points in Hough space are discontinuous lines in image space!



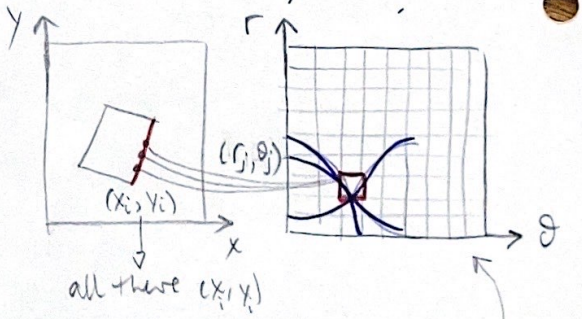
many lines intersecting almost in one same point represent a line segment consisting of disconnected points.

→ same happens in polar Hough space, but here sinusoids intersect

⇩

Therefore, for finding lines in image space, we find intersections in Hough space!

- How does the Hough transform work?
  - We apply an edge detector to image, eg Canny
  - Each point (x_i, y_i) of the binarized image is transformed as (θ_i, r_i)
  - The Hough space is discretized in accumulator cells that contain how many (θ_i, r_i) points lie in them
    - → cells with resulting high values are related to intersections ⇒ lines!



- The same approach can be applied to other analytical shapes, eg. circles:

$$(x-a)^2 + (y-b)^2 = R^2$$

↳ 3 params: a, b, R ⇒ 3D Hough space with a 3D accumulator

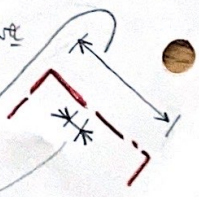
- Parameters for OpenCV:

cv2.HoughLinesP(cannyImage, ...)

rho = 1
theta = np.pi/180
threshold = 60
min_line_length = 50
max_line_gap = 5

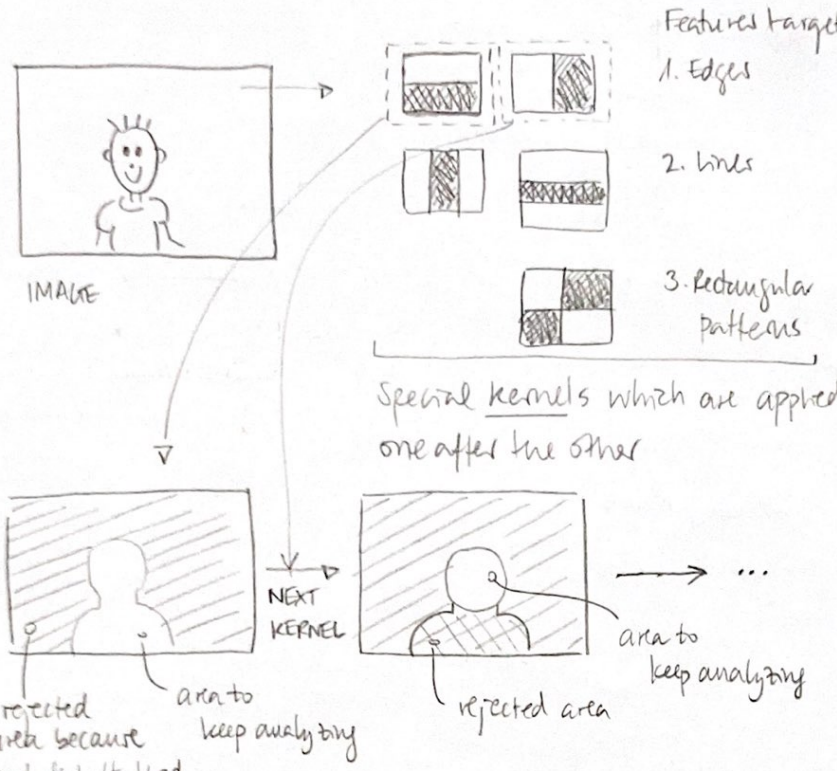
resolution

(# intersections)



→ Note: in cv2.HoughCircles(), the gray blurred image is passed, not the canny image!

Haar Cascades: Applied to Face Detection (LaVola-Jones Paper/Algorithm)



The algorithm works rejecting irrelevant areas, so it's very fast: Video face detection possible.

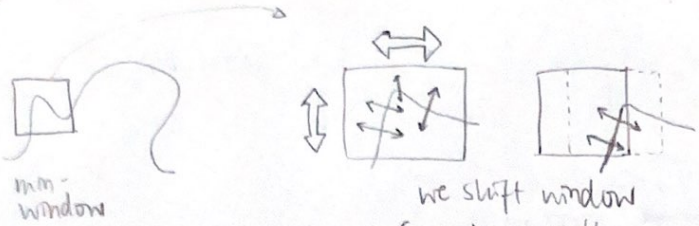
selection/rejection happens through a classificator for the particular feature (kernel) that was trained with face/no face images; basically classificators applied to sub windows

- Haar cascades can be used for segmenting areas of interes for further processing.
- We can define Haar cascade classificators for different objects, not only faces, but features and kernels must be defined accordingly, and the classificators trained! (these are trained with yes/no labelled images beforehand)

TYPES OF FEATURES AND IMAGE SEGMENTATION

Corner detection: Harris

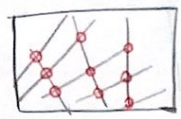
- Typical features can be
  1. Edges: high gradients or intensity changes
  2. Blobs: regions or areas with a color/intensity/texture
  3. Corners: intersections of edges → this are the most salient! ⇒ one of the most important! because they can match unique parts of the image
- Corner detection can work as follows:
  - We take a mini-window and observe the image gradient magnitude and direction
  - Then we shift the window in all directions
  - If the magnitude and direction change considerably → CORNER!



· magnitude of the gradient: $g = \sqrt{S_x^2 + S_y^2}$ (Sobel)

· direction of the gradient: $\varphi = \text{atan}\left(\frac{S_y}{S_x}\right)$

— This is implemented in the **Harris** corner detector.



Morphological operations

Often performed on binary images to improve their properties

* Dilation → cv2.dilate(image, kernel, iterations)
* Erosion → cv2.erode(image, "", "")
  - "" → None, default
  - usually 1.

* Opening: erosion + dilation
  - erosion: remove noise and shrink region
  - dilation: enlarge object again
  - → remove salt noise

cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)

* Closing: dilate + erode
  - dilate: close small gaps and enlarge region
  - erode: shrink object again
  - → close holes = remove pepper noise

cv2.morphologyEx(..., cv2.MORPH_CLOSE, ...)

Contours (Motivation: Image Segmentation)

— Contours: complete closed boundaries that contain objects
- Continuous curves that follow the edges along a boundary
- Computed after edges or a binary threshold
- Provide info on contained object: area, shape, center, perimeter, bounding box

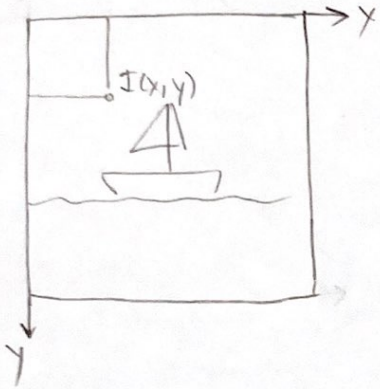
contour features → there are much more... see TXT

these can be used to analyze shape of objects

cv2.threshold() → cv2.findContours() → cv2.drawContours()

- threshold(): try to use white background, then apply inverse threshold
- findContours(): retrieval contour tree or hierarchy + list of contours → each contour is a list of coordinates in the image — I understand that, joining them, we draw the contours! (so we don't have subpixel precision?)

Image Moments



$$M_{pq} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x^p y^q I(x,y)\,dx\,dy \quad \text{(continuous)}$$

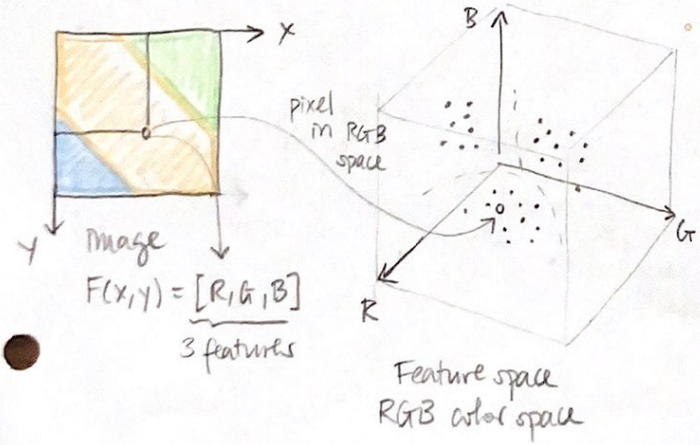
$$\left[M_{ij} = \sum_x \sum_y x^i y^j I(x,y) \right] \quad \text{(discrete, image)}$$

- Sum of gray values: M_{00}
  - for BINARY images: area = M_{00}
- centroid: $(\bar{x}, \bar{y}) = \left(\frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}}\right)$

```
M = cv2.moments(contour)
m00 = M['m00'] → dictionary with all moments
```

K-means Clustering (Motivation: Image Segmentation)

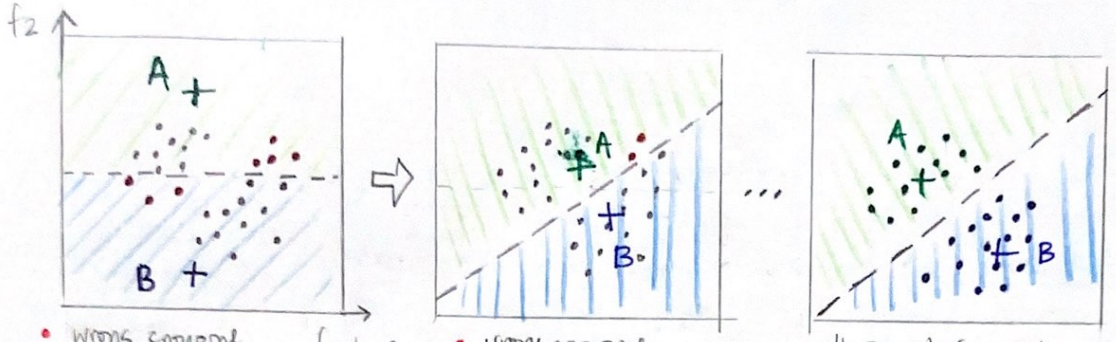
- It separates an image into segments by clustering data points that have similar features/traits.
- Unsupervised learning: no need of labelling
- Input variable: K groups to cluster data



Example:

- Color segmentation of a flag
- K = 3
- (x,y) pixels set in RGB color space
- K = 3 clusters identified
- Assumption: there are 3 natural or expected color groups!

- Clustering (2D example):



• wrong grouping

• wrong grouping

all samples correctly grouped!

1) k = 2 centerpoints A, B are randomly set in feature space

2) Assing each data point to closest centerpoint A/B

3) Move centerpoints A & B to center of mass of clustered points

4) Repeat 2): Assing each point to closest centerpoint A/B and then repeat 3)

- The algorithm is expected to converge if there's an underlying structure.
- STOP criteria:
  → num iterations
  → if centerpoints move less than a threshold

The notebook with the example is very interesting because it has non-straightforward reshaping & masking operations that are basic...

FEATURE VECTORS (& OBJECT IDENTIFICATION)

→ Feature vectors are used to detect objects, going beyond object parts such as corner

- Corners are distinct image features that can be used as matching patterns
- BUT: we want to find whole objects, not just corners
  ↳ SOLUTION: we implement feature vectors that contain several (eg corner) feature data
- Example

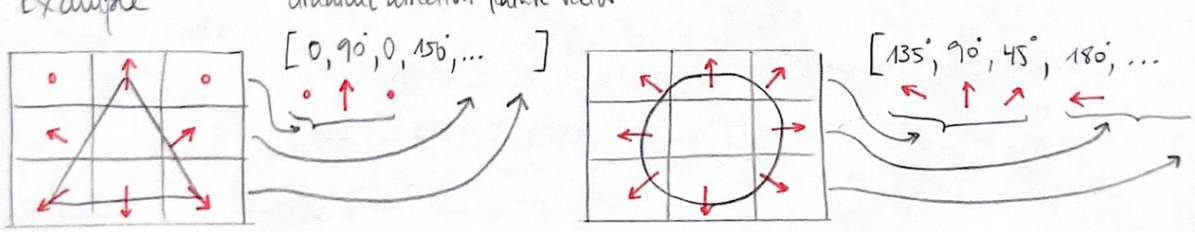


image divided in cells and average gradient direction computed

- ADDITIONALLY: we want to detect objects in REALTIME — for applications like { • Self-driving cars • AR }

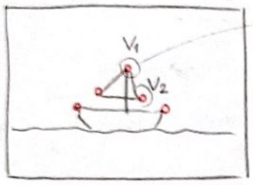
Rublee, Rabaud, Konolige, Bradski – 2011, Willow Garage

ORB = Oriented FAST and rotated BRIEF

- FAST: Features from accelerated segment test
- BRIEF: Binary robust independent elementary features → Object detection of clear inherent shape cues (eg, faces)

↳ State-of-the-Art algorithm feature detector (2011):
- used for real-time object recognition or 3D reconstruction
- alternative to SIFT & SURF: scale-invariant feature transform] → because these are patented!
- robust wrt illumination changes, noise & image transformations!

1) FAST: feature detection



Keypoints: small distinctive regions, eg, corners

2) BRIEF: feature vector creation

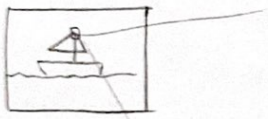
$V_1 = [0101110\ldots]$

$V_2 = [10010011\ldots]$

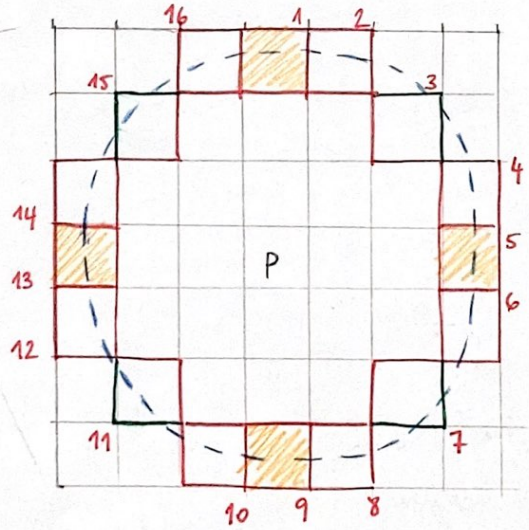
Keypoints encoded as bitstrings: Binary feature vectors

sequence depicts properties of keypoint area: patterns of intensity

(1) FAST: Feature or keypoint detection (Features from Accelerated Segment Test)



We stop in a pixel P



- I_p: intensity of pixel p
- h: threshold intensity
- 16 pixels in a circle around p are classified to be:
  1) BRIGHTER than p if $I > I_p + h$
  2) DARKER if $I < I_p + h$
  3) SIMILAR otherwise

→ Pixel P is a keypoint iff >8 CONNECTED pixels in this set of 16 are either BRIGHTER or DARKER

→ Faster version with same result: consider set of 4 equidistant pixels, (4x faster) eg, 1, 5, 9, 13

↳ p is keypoint iff 2 consecutive pixels are either BRIGHTER or DARKER

- keypoints found by FAST account for the location (not orientation) of OBJECT DEFINING EDGES

(Binary Robust Independent Elementary Features)

② BRIEF : Feature vector creation

mem. efficient & fast to manipulate

- keypoints are encoded as binary vectors or bit-strings of 128 to 512 bits or descriptors

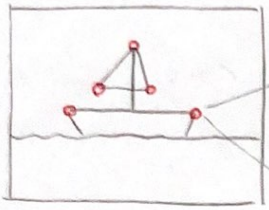
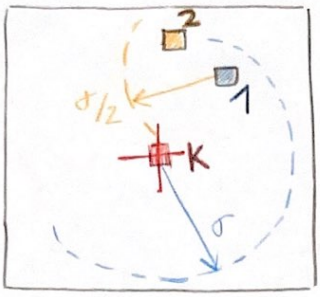


image with keypoints + BLURRED to remove high freq. noise (Gaussian kernel)

for each keypoint K



patch: neighborhood of WxH pixels around keypoint K

- Two random pixels are selected around K
  1) First pixel within the Gaussian spread with σ around K
  2) second pixel within the Gaussian spread σ/2 around pixel 1
- Then, intensities of 2 pixels are compared:

$I_1 > I_2 \rightarrow 1$, 0 otherwise

(1 brighter than 2)

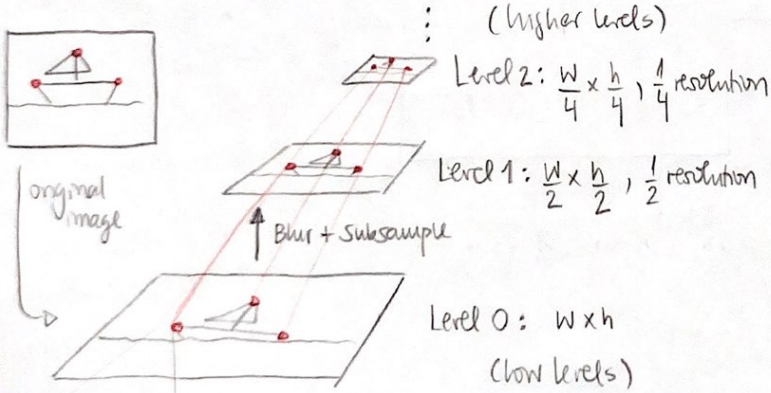
? BUT random pixels are always the same or each time different?

- The process is repeated N times for each key_point (N = length of binary string):

$$V(K) = [\overset{N}{\overbrace{1\,0\,0\,1\,1\,0\ldots}}]$$

↳ it contains intensity/brightness comparisons of N (128..512) random pixel pairs around keypoint K

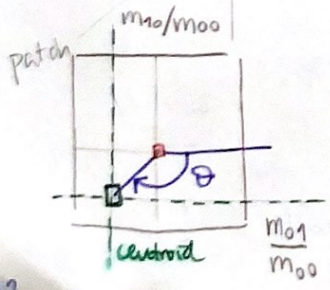
SCALE & ROTATION INVARIANCE : how is it achieved?



- Image pyramids are computed
- For each level image, keypoints are located

→ this way, we are scale invariant!

? Basically, keypoints of original level are scaled to higher levels without recomputing them, right?



- For each keypoint of each level, its centroid (center of mass of intensities) is computed in a patch of constant size (number of pixels)
  - We take angle θ of centroid

↳ constant for all levels, thus, keypoints in higher levels are bigger!

- After computing θ, a modified version of BRIEF is used
- RBRIEF = Rotation aware BRIEF : it introduces rotation invariance
  - It starts like BRIEF: N (= 256, eg) random pixel pairs are created
  - Then, the set of pixel pairs is rotated θ !
  - This way, we always compute the bit-string with respect to the axis given by the centroid !
- The set of keypoints and feature vectors is known as ORB descriptor: all BRIEFS stacked together.

Feature matching

(?) Why is it called training if there's no iterative optimization?

- Keypoints of [training] image are saved
- Keypoints of query image are computed (that's what I understand (?)) → correct.
- Training & query keypoints are compared: their similarity is measured with a metric
  - Hamming distance is commonly used = # positions at which symbols are different
- Best keypoint matches are returned] → (?) Is the relative position of the matched keypoints analyzed?
- If # matches above [a threshold] ⟹ object found!
  - ↳ usually a % of all keypoints; it's a user parameter (eg 35 %)

(?) If BRIEF is done with random pixel positions each time, how is that going to work?

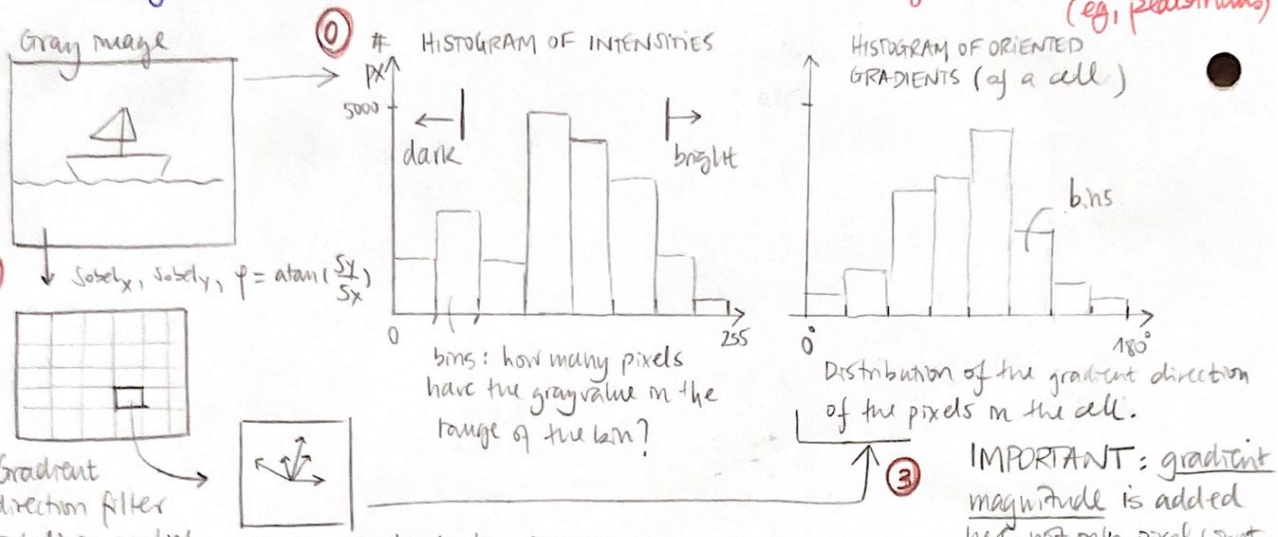
- Applications
  - Video (because it's very fast)
  - Face detection
  - Concrete object detection, no matter background
- BUT: ORB doesn't work well for general object detection. In other words, to use ORB, objects need to have clear distinctive and consistent internal features → for other cases, we can use HOG with a classifier.

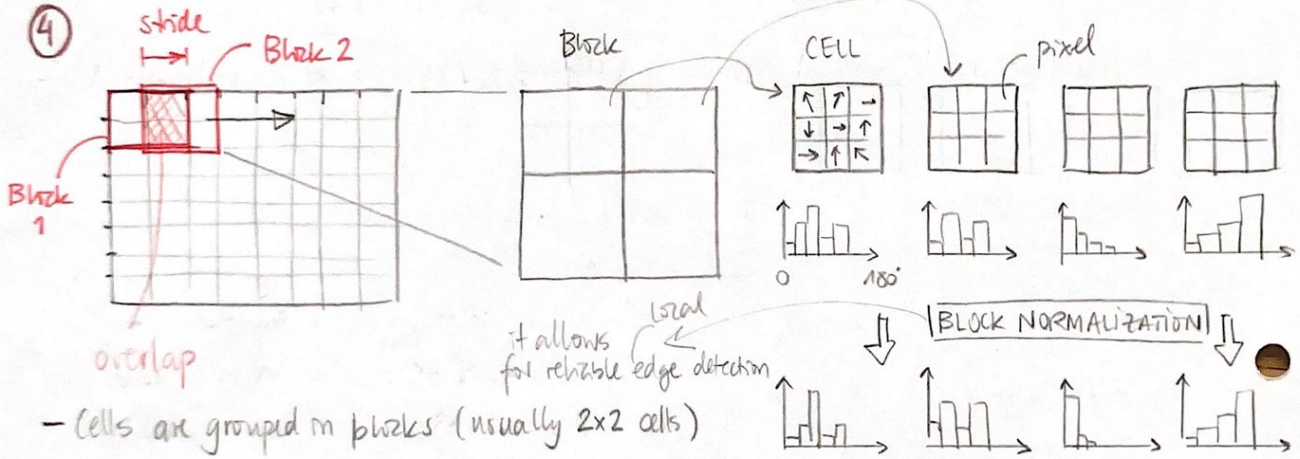
- In the Notebook (Jupyter), we see that ORB is:
  - scale invariant
  - rotation invariant
  - illumination invariant
  - noise invariant

(?) The real useful application would be to determine the scaling & pose of the template in the image — is there a function for that?

HOG = Histogram of Oriented Gradients (Dalal & Triggs 2005)

→ Object detection of generic object classes (eg, pedestrians)





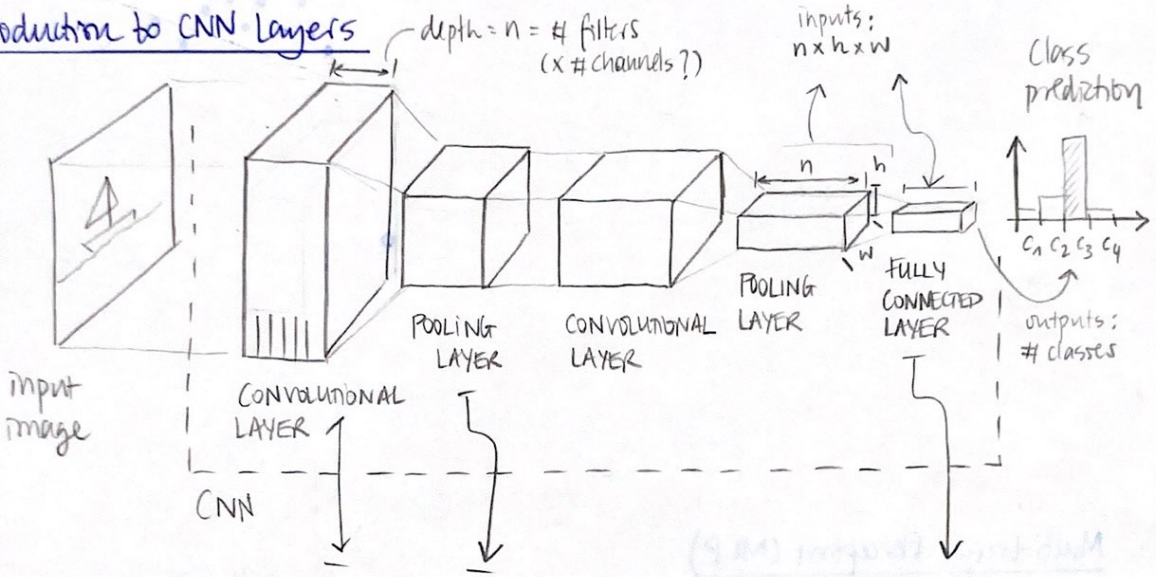
- Cells are grouped in blocks (usually 2x2 cells) that usually overlap
  - The distance between blocks is called stride; it's recommended having a stride that generates overlap (typically: 1 cell)
- The HOGs of a block are normalized together
  - Thus, assuming blocks overlap, most cells will be normalized wrt several blocks → this increase accuracy
- HOG descriptor: feature vector that contains all normalized HOG histograms of all the ~~cells~~ block-cells in the image
- A classifier, eg a Support Vector Machine (SVM) is trained with many HOG descriptors of the object class (eg. pedestrian) and many HOG descriptors of other object classes
- Detection is done with a sliding window over image trying to look for image part similar to HOG descriptor

- Note: the # of histograms in the HOG descriptor ≠ # cells, but
  = # blocks × cells/block
  Therefore, if there's block overlap (as recommended), the histogram of a cell will come up in the vector several times with a different normalization.

CNN LAYERS AND FEATURE VISUALIZATION

→ continues on page 28

Introduction to CNN Layers



In a CONVOLUTIONAL LAYER we have filters that are applied to the image

- filter weights are learned during training with backpropagation
- filters highlight important features (edges, color, etc.)
- if we have n filters, we'll produce n convoluted images and the convolutional layer will have depth = n.

In a POOLING LAYER the size of the filtered images is reduced by taking relevant pixels. For instance, if we do maxpooling, the max pixel value of a cell of pixels is taken; example

20	30	80	90
140	45	40	50
200	100	6	7
15	16	50	250

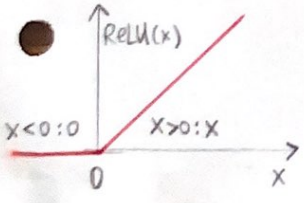
⇨

140	90
200	250

This:
- reduces dimensionality
- increases robustness wrt small gray value changes
- we increase field of view to later filters

In a FULLY CONNECTED layer we take as input all the pixels from last layer and map them to class nodes in a fully connected manner

ACTIVATION



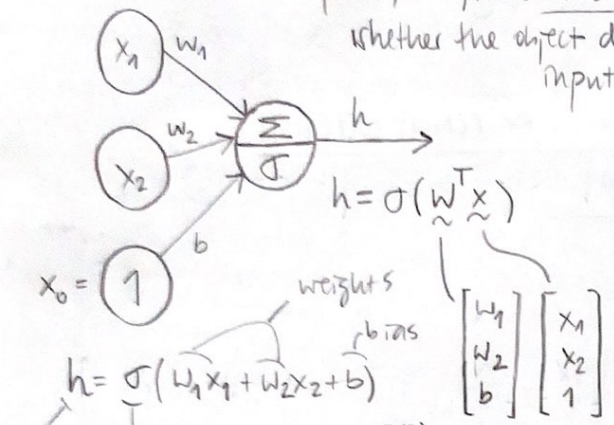
- Images are normalized to [0,1] region
- Negative pixel values are set to 0 with Relu activation

NEURAL NETWORKS | Optional exercises (from several sources)

but, in general, following Udacity's guideline

Perceptron

: binary classifier: given n inputs, it yields PROBABILITY whether the object described with inputs belongs to a class or not.



$$h = \sigma(w^T x)$$

$$\begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

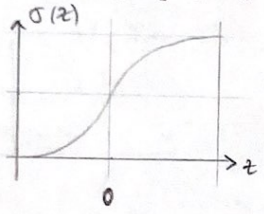
$$h = \sigma(w_1 x_1 + w_2 x_2 + b)$$

weights, bias

output

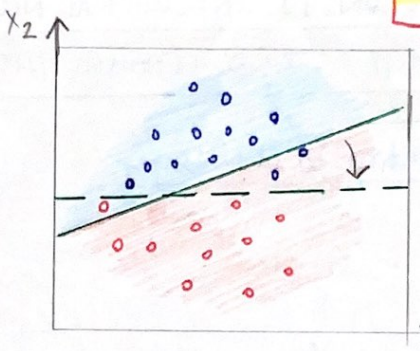
activation function: sigmoid

$$\sigma(z) = \frac{1}{1+e^{-z}}$$



It maps $[-\infty, \infty]$ to $[0, 1]$ because h should give a probability!

Neural networks with non-linear activations work like universal function approximators.



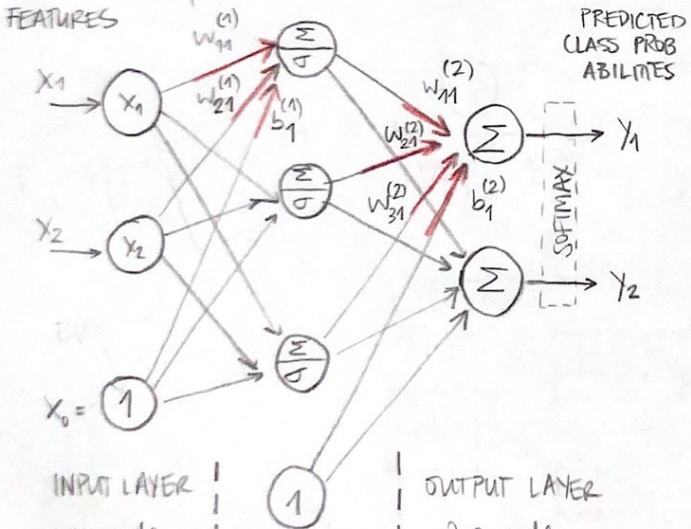
- Training sample $h = 1$ (ground truth)
- " " $h = 0$ (" ")
- Region predicted as $h = 1$
- " " $h = 0$

Boundary trained, composed of the weights

During training the weights are modified so that the boundary better splits the training samples in their categories

Multi-Layer Perceptron (MLP)

Several perceptrons stacked one after the other



- All layers are fully connected
- All layers except output have bias b (= x_0)
- Several activation functions are possible:
  - Heaviside step: $\sigma_h(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$
  - ReLU: $\sigma_r(z) = \max(0, z)$
    - it's non-linear, but the most effective & used
  - ...
- For classification, softmax is applied with outputs: normalized exponential function

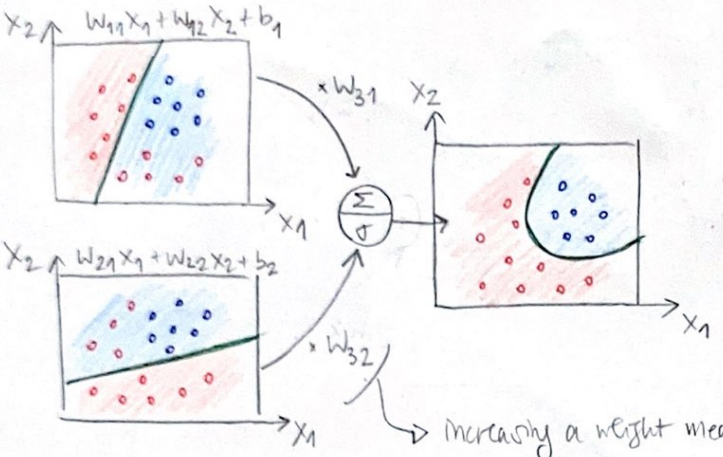
$$\sigma_i(\underset{\sim}{z}) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad i = 1, \dots K, \quad \underset{\sim}{z} = [z_1, \dots z_K]$$

(K: # classes)

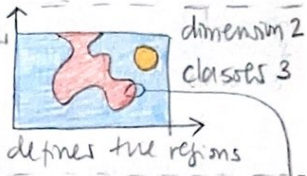
- While a single perceptron draws a linear boundary between 2 regions, a MLP is a linear combination of perceptrons that achieves **non-linear boundaries**

The weights and biases define non-linear boundaries in feature space!

Each region is a class!



- the # features defines the dimension
- the # classes defines the regions

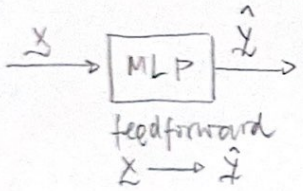


Increasing a weight means giving more importance to the output of that node, as it happens with neurons in the brain: as Hebb suggested: when a neuron triggers another, their connection grows stronger.

More neurons & layers result in more non-linear boundaries

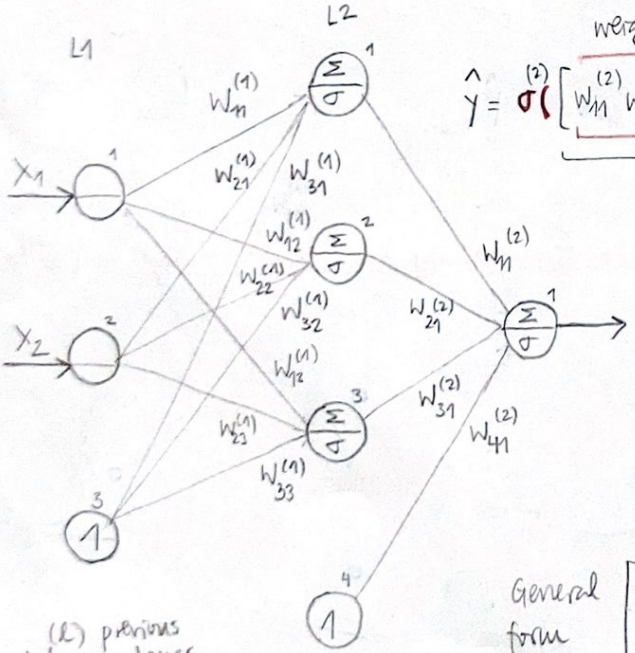
Feedforward: the process of obtaining a prediction after applying the network to the input

Notation: Function composition: $g \circ f(x) \equiv g(f(x))$



$\underset{\sim}{X} = [X_1\ X_2\ X_3]^T$ ← inputs: features (eg 3)

$\hat{\underset{\sim}{y}} = [Y_1\ Y_2]^T$ ← # classes: probabilities (eg, 2 classes)



$$\hat{y} = \sigma^{(2)}\left(\underbrace{\begin{bmatrix} W_{11}^{(2)} & W_{21}^{(2)} & W_{31}^{(2)} & W_{41}^{(2)} \end{bmatrix}}_{\underset{\sim}{W}^{(2)}} \cdot \sigma^{(1)}\left(\underbrace{\begin{bmatrix} W_{11}^{(1)} & W_{21}^{(1)} & W_{31}^{(1)} & 0 \\ W_{12}^{(1)} & W_{22}^{(1)} & W_{32}^{(1)} & 0 \\ W_{13}^{(1)} & W_{23}^{(1)} & W_{33}^{(1)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\underset{\sim}{W}^{(1)}} \begin{bmatrix} X_1 \\ X_2 \\ 1 \\ 1 \end{bmatrix}\right)\right)$$

(weights | BIAS)

for consistency (2nd layer has one node more than input)

$$\hat{y} = \sigma \circ \underset{\sim}{W}^{(2)} \circ \sigma \circ \underset{\sim}{W}^{(1)} (\underset{\sim}{X})$$

General form:

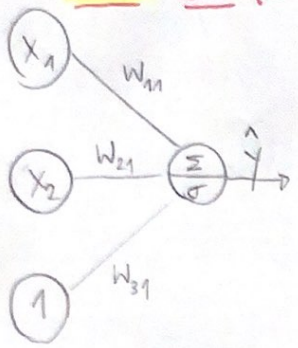
$$\hat{\underset{\sim}{y}} = \underset{S}{\sigma} \circ \underset{\sim}{W}^{(L)} \circ \underset{\sim}{\sigma} \circ \underset{\sim}{W}^{(L-1)} \circ \cdots \circ \underset{\sim}{\sigma} \circ \underset{\sim}{W}^{(1)} (\underset{\sim}{X})$$

- $\hat{\underset{\sim}{y}}$: probabilities of classes
- $\underset{S}{\sigma}$: softmax
- $\underset{\sim}{\sigma}$: sigmoid
- $\underset{\sim}{W}^{(L)}$: weights of layer L

$W_{ij}^{(L)}$: previous layer; previous neuron → next neuron

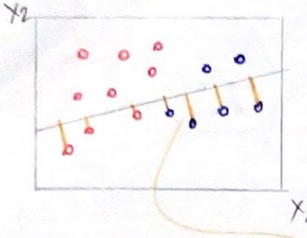
Error function: it measures the error between the ground truth and the prediction

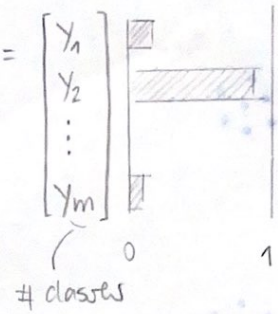
aka **LOSS** or **COST** function



$\hat{y} = \sigma(\underset{\sim}{W}\underset{\sim}{x})$ (Perceptron)

$\hat{\underset{\sim}{y}} = \sigma \circ \underset{\sim}{W}^{(L)} \circ \dots \circ \sigma \circ \underset{\sim}{W}^{(1)}(\underset{\sim}{x}) =$ (general)





– The error is a metric that measures the distance between $\underset{\sim}{y}$ (ground truth) and $\hat{\underset{\sim}{y}}$ (predicted)

- One metric would be the Mean Squared Error (MSE)

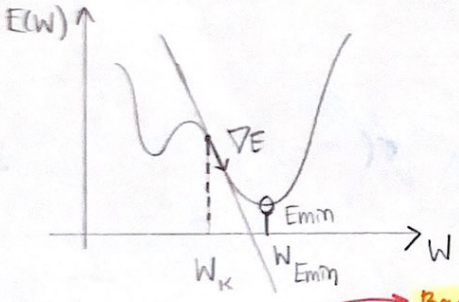
$$E_{MSE}(\underset{\sim}{W}) = \frac{1}{m}\sum_{j=1}^{m}(y_j - \hat{y}_j)^2$$ (that's actually related to the distance between $\underset{\sim}{y}$ & $\hat{\underset{\sim}{y}}$)

- Since $\underset{\sim}{y}$ measures probabilities of classes, it's a better to use the Cross-Entropy:

$$E_{CE}(\underset{\sim}{W}) = -\frac{1}{m}\sum_{j=1}^{m} \underbrace{y_j \ln(\hat{y}_j)}_{\text{one class}} + \underbrace{(1-y_j)\ln(1-\hat{y}_j)}_{\text{the other class}}$$

There are more error functions

Gradient Descend: we modify $\underset{\sim}{W}$ in the direction in which $E(\underset{\sim}{W})$ decreases



$$W_{k+1} = W_k - \alpha \nabla E(W_k)$$

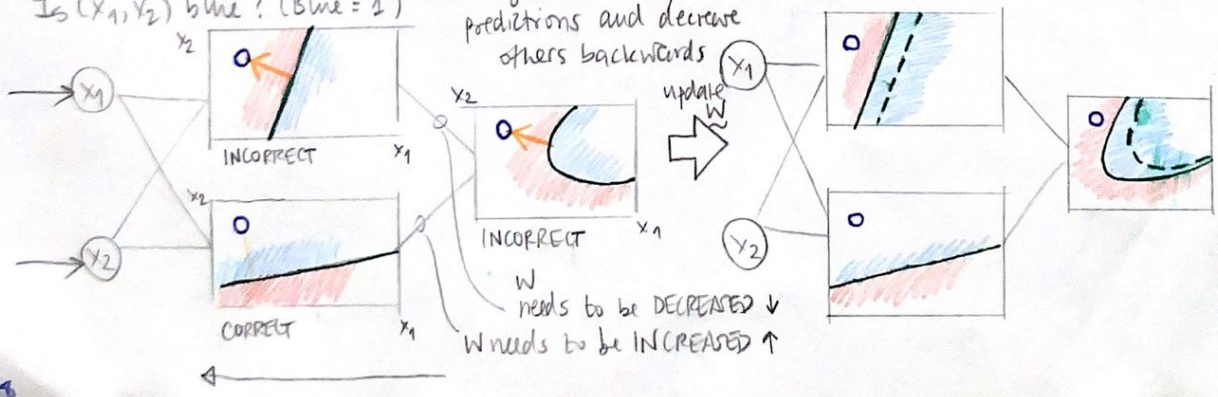
update · learning rate · gradient: direction of max decrease of E

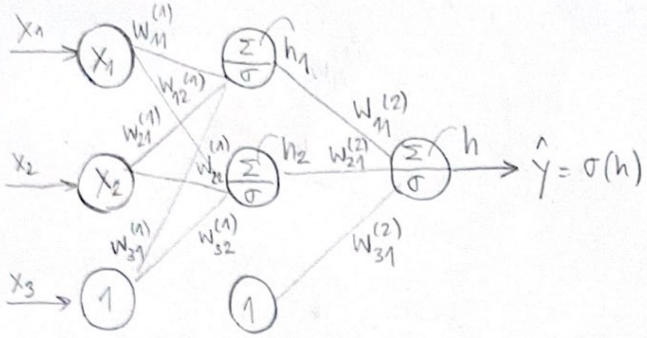
the gradient for gradient descent

Backpropagation: Intuition: we start in the output and check whether the prediction corresponds with ground truth; we increase the weights that contribute correct predictions and decrease others backwards

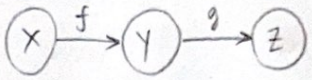
→ Backpropagation is essentially an application of the chainrule to find

Is (x_1, x_2) blue? (Blue = 1)





Chain rule



$y = f(x)$ $\quad z = g(y) = g(f(x)) = g \circ f(x)$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

- $\hat{y} = \sigma \circ \underset{\sim}{W}^{(2)} \circ \sigma \circ \underset{\sim}{W}^{(1)} (\underset{\sim}{x})$
  - usually they are vectorized
- $\nabla E(\underset{\sim}{W}) = \left[\dots, \frac{\partial E}{\partial w_{ij}^{(l)}}, \dots \right]$ The gradient of E is also vectorized
- Weights are updated like this: $w_{ij}^{(l)'} = w_{ij}^{(l)} - \alpha \frac{\partial E}{\partial w_{ij}^{(l)}}$ (Gradient descend)
  - α: learning rate
  - $\frac{\partial E}{\partial w_{ij}^{(l)}}$: which are these?
- Computation of $\partial E / \partial w_{ij}^{(l)}$ (with the example):

$$h_1 = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)}$$

$$h_2 = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{32}^{(1)}$$

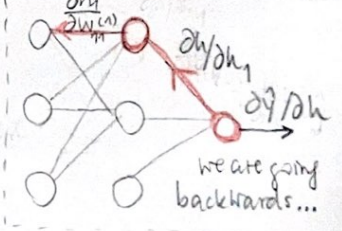
$$h = w_{11}^{(2)} \sigma(h_1) + w_{21}^{(2)} \sigma(h_2) + w_{31}^{(2)}$$

$$\hat{y} = \sigma(h) = \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)} (\underset{\sim}{x})$$

$$E(\underset{\sim}{W}) = -\frac{1}{m} \sum_{k=1}^{m} y_k \ln(\hat{y}_k) + (1 - y_k) \ln(1 - \hat{y}_k) = E\left(\underbrace{w_{11}^{(1)}, w_{21}^{(1)}, w_{31}^{(1)}}, \underbrace{w_{12}^{(1)}, w_{22}^{(1)}, w_{32}^{(1)}}, \dots \right)$$

$$\nabla E(\underset{\sim}{W}) = \left[\frac{\partial E}{\partial w_{11}^{(1)}}, \frac{\partial E}{\partial w_{21}^{(1)}}, \frac{\partial E}{\partial w_{31}^{(1)}}, \dots \right]$$

BACKPROPAGATION VISUALLY



$$\frac{\partial E}{\partial w_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h} \cdot \frac{\partial h}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_{11}^{(1)}}$$

all terms here:

$$\frac{\partial E}{\partial \hat{y}} = \hat{y} - y \quad , \quad \frac{\partial h}{\partial h_1} = w_{11}^{(2)} \frac{\partial \sigma}{\partial h_1} = w_{11}^{(2)} \underbrace{\sigma(h_1)(1 - \sigma(h_1))}$$

$\partial \sigma / \partial h_1$, σ' sigmoid derivative!

$$\frac{\partial \hat{y}}{\partial h} = \sigma(h)(1 - \sigma(h)) = \hat{y}(1 - \hat{y}) \quad , \quad \frac{\partial h_1}{\partial w_{11}^{(1)}} = x_1$$

- So, basically, each $\partial E / \partial w_{ij}^{(l)}$ is computed following the chain rule and the vector $\nabla E(W)$ is assembled
- Training happens by updating the weights with the gradient:

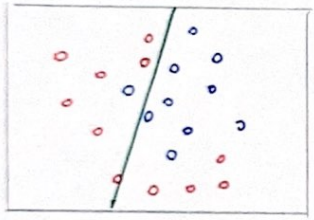
$$W' = W - \alpha \nabla E(W)$$

- In short, backpropagation consists in applying gradient descend to the $E(W)$ function, and for that, using autodifferentiation and the chain rule.

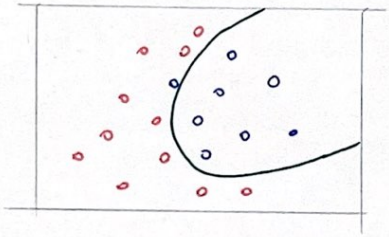
TRAINING: Test split & complexity

- In the dataset, ALWAYS define a training split and a test split.
- Use the test split to evaluate how the model works.
- Prefer simple models over complex:

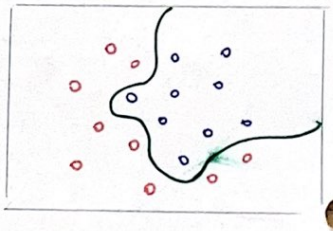
TRAINING: Overfitting & Underfitting : we check our fitting with the test split !!!



UNDERFITTING
- Too simple model
- Error by bias because we have a high bias
- Usually simplistic architectures



CORRECT
- Incorrect prediction is treated as noise
- Model generalizes correctly
- Slightly complex architectures



OVERFITTING
- Too complex model
- Error by variance, because high variance is learned
- Model is TOO SPECIFIC and has problems to generalize
- Noise is fitted
- Very (too) complex architectures

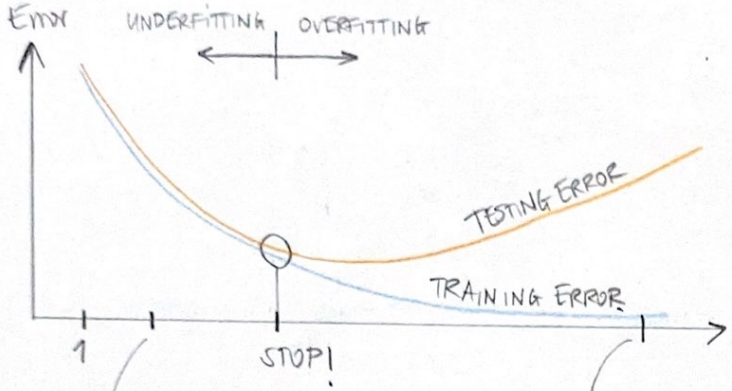
usually we start with a complex architecture and prevent it to overfit.

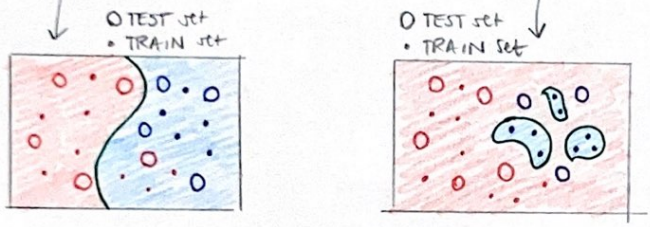
Methods to prevent overfitting:
1. Early Stop
2. Regularization
3. Dropout, 4. Batch Normalization → page 31

TRAINING: Early Stopping (it prevents overfitting)

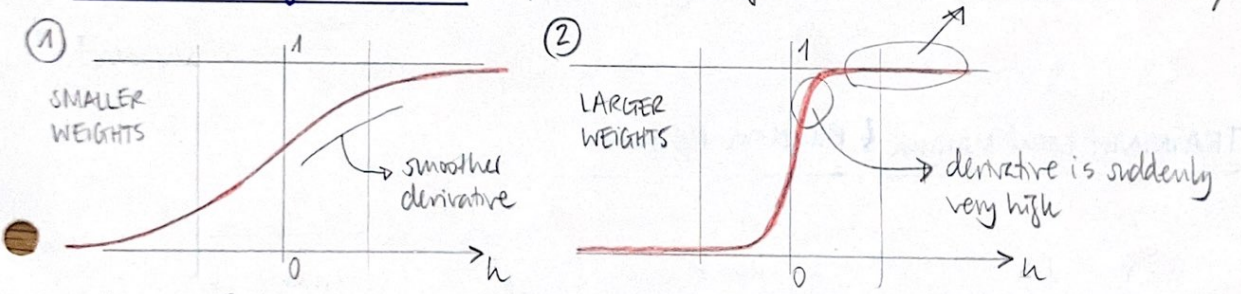
- Divide the dataset in randomly distributed TRAINING & TESTING sets ← not seen by network for computing weights
- Train with the TRAINING set + compute $Error_{train}$
- Test -"- TESTING set + compute $Error_{test}$

→ Stop test when $Error_{test}$ starts to increase while E_{train} decreases





TRAINING: Regularization (it prevents overfitting)



$\hat{y} = \sigma(\underset{\sim}{W}^T \underset{\sim}{x})$

$\hat{y} = \sigma(x_1 + x_2)$, $\underset{\sim}{W} = [1\ 1\ 0]^T$

$\hat{y} = \sigma(K \underset{\sim}{W}^T \underset{\sim}{x})$, $K > 1$

$\hat{y} = \sigma(10x_1 + 10x_2)$, $\underset{\sim}{W} = [1\ 1\ 0]^T$, $K = 10$

- Larger weights (②) shrink the sigmoid function and it becomes more sensitive and unstable
  - Derivative is almost ∅ very soon
  - and it suddenly increases a lot close to the origin
- Too large weights stretch the output of σ to the border values (1 or 0), so the model is TOO CERTAIN = WE ARE OVERFITTING!

→ SOLUTION: REGULARIZATION: Penalize large weights by adding them to the error

↳ 2 major methods: { L1: $\Sigma |W_{ij}|$; L2: $\Sigma |W_{ij}|^2$

L1 REGULARIZATION

error

$$E(W) = -\frac{1}{m}\sum_{j=1}^{m} y_j \ln(\hat{y}_j) + (1-y_j)\ln(1-\hat{y}_j) + \lambda \sum |w_{ij}|$$

L1 Regularization term

$\lambda \uparrow$ if we want to penalize more

- We tend to end up with sparse vectors:
  - small weights become 0
  - bigger weights become 1
- → Good for FEATURE SELECTION

L2 REGULARIZATION

$$E(W) = -\frac{1}{m}\sum_{j=1}^{m} y_j \ln(\hat{y}_j) + (1-y_j)\ln(1-\hat{y}_j) + \lambda \cdot \sum (w_{ij})^2$$

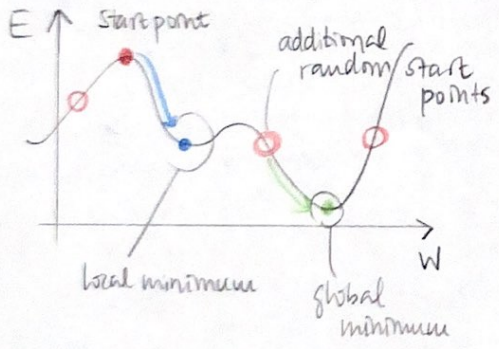
L2 regularization term

- We tend to have homogeneously small weights in the vectors
- It's the most used, because usually we want to train models with homogeneously SMALL WEIGHTS

TRAINING: Dropout (It prevents overfitting)

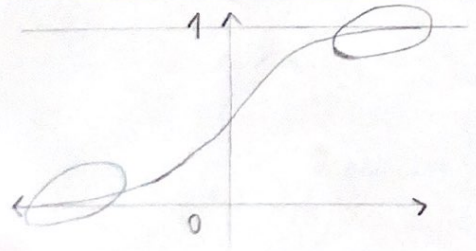
- Sometimes some weights are considerably larger than others and they end up dominating the training
- Solution: we randomly deactivate nodes during training & let the rest train = DROPOUT.
- The probability of any node to deactivate is fed as parameter, eg 0.2.

TRAINING: Local Minima & Random Restart



- Gradient descent has the flaw or risk of leading to a local minimum.
- SOLUTION: start from several random initial sets = RANDOM RESTART
  - We cannot guarantee we achieve the global min
  - BUT: the local min will be better than before!

TRAINING: Other activation functions



- The problem of the vanishing gradient: the sigmoid function has a very small slope very soon
- Since the gradient is packed of those derivatives, we're going to have VERY SMALL steps!

$$\nabla E(\underset{\sim}{W}) = \left[\dots, \frac{\partial E}{\partial w_{ij}^{(L)}}, \dots \right]$$

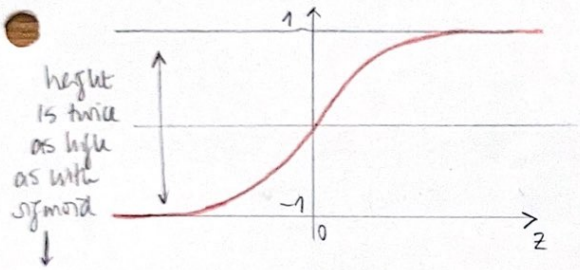
$$\frac{\partial E}{\partial w_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h} \cdot \frac{\partial h}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_{11}^{(1)}}$$

all are derivatives of a sigmoid ⇒ small

product of small values yields a TINY value!

→ The best way to solve this issue are other activation functions with larger slopes:

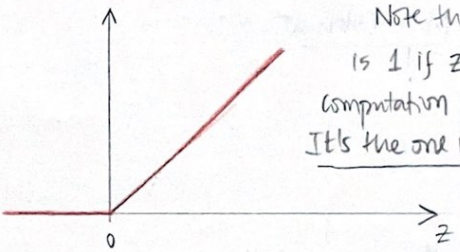
HYPERBOLIC TANGENT



height is twice as high as with sigmoid ↓ max slope

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

RELU: RECTIFIED LINEAR UNIT



Note that its derivative is 1 if $z > 0$! So its computation is much faster! It's the one usually used.

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

⇨ CONCLUSION:

- Usually the ReLU(z) function is used as activation function, not the sigmoid $\sigma(z)$
- The last (output) nodes have a softmax or in the case of a unique class:
  - the sigmoid, if we want probability
  - ReLU if we want some regression

(?) That's basically the same? YES, it's meant the same!

TRAINING: Gradient descent vs STOCHASTIC vs BATCH gradient descents

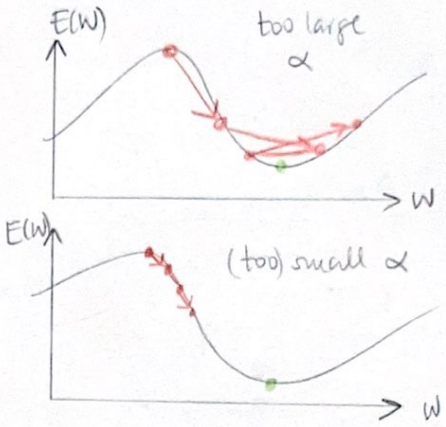
- In regular gradient descent we enter all the samples to the network, compute the error & the gradient and make an update step in the weights. That's called an Epoch.
- BUT: this approach has some drawbacks
  - We need to process all the data to make a small step
  - Loading the whole date every step to memory might be impossible in practice!
- SOLUTION: STOCHASTIC GRADIENT DESCENT: data is processed in batches or subsets
  - Each batch approximates the error function, but it's more memory efficient to do that
  - A mini-step after each batch is taken, which is better than one step after one Epoch

- The mini-steps with batches are less accurate, but more effective at the end

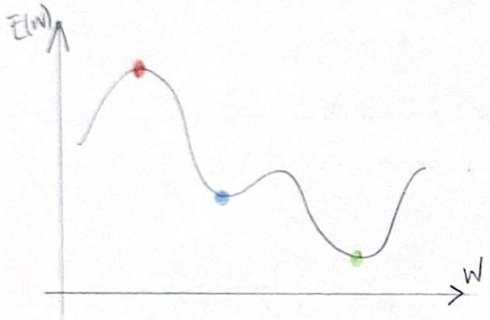
TRAINING: Learning Rate (α)

- Too large learning rate:
  - big steps ; good at beginning,
  - BUT we might miss the minimum
- Too small learning rate:
  - we make too small steps ⇒ too slow
  - BUT: usually, tend to use small learning rates

→ GOOD strategy: DECREASE learning rate as we approach minimum



TRAINING: Momentum



- In order to avoid falling in local minima, the step is extended with the previous steps:
  - Local minima will be left behind
  - Target minima will be found
  - Since the step size is bigger, convergence around target minimum will take a bit longer

$$\left.\begin{aligned} z^{k+1} &= \beta z^{k} + \nabla E(w^{k}) \\ w^{k+1} &= w^{k} - \alpha z^{k+1} \end{aligned}\right\} \rightarrow w^{k+1} = w^{k} - \alpha\left(\nabla E(w^{k}) + \beta \nabla E(w^{k-1}) + \beta^{2} \nabla E(w^{k-2}) + \dots\right)$$

$1 \quad \beta \quad \beta^{2} \quad \dots$

$\beta \in [0, 1)$

- 0 → no momentum
- 0.99 boosts search

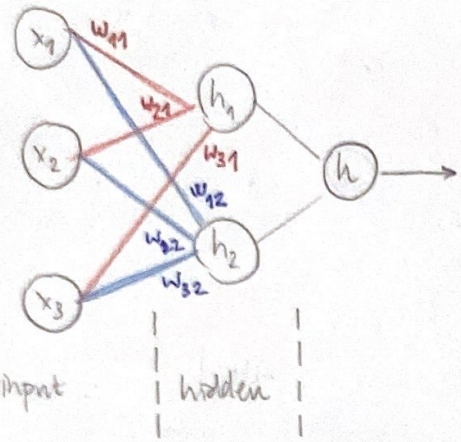
→ that's how it's done in practice: another step variable is accumulated

→ that's the effect: previous steps are summed weighting them with β^{j}, j = # step back in past; older steps have less influence

PYTORCH

Notes on how to use it.

→ See notes on Udacity Guide related to the Notebooks !!!
I have very simple examples in these handwritten notes → ADVANCED STUFF IN JUPYTER !!!



hidden units →

input units ↓

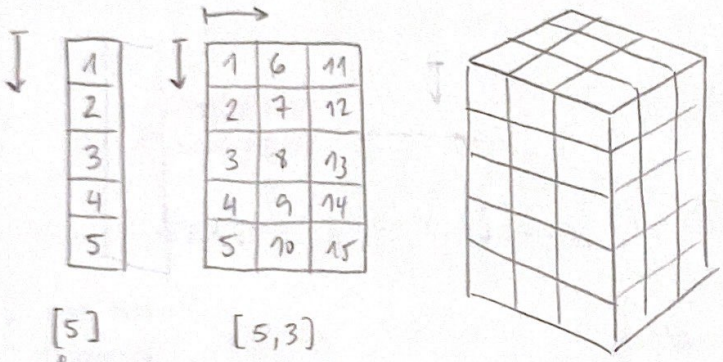
$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{23} \end{bmatrix}$$

the feed forward propagation between 2 layers can be expressed with matrices

$$\vec{h} = [h_1\ h_2] = [x_1\ x_2 \dots x_n] \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ \vdots & \vdots \\ w_{n1} & w_{n2} \end{bmatrix}$$

$$\hat{y} = f_2\left(f_1(\vec{x}\,\underset{\sim}{W_1})\,\underset{\sim}{W_2}\right)$$

Tensors: similar to N-D arrays in numpy



- Datasets represented with tensors
- Samples are also tensors, which are passed usually in batches, eg:

(64, 1, 28, 28)

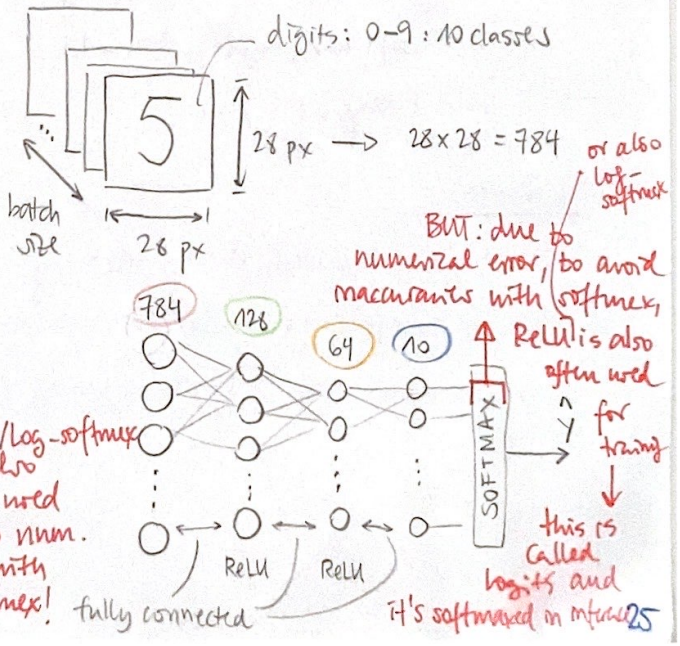
batch size | channels | rows x columns

Networks (import nn, F, etc.)

```
class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)
    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.softmax(x, dim=1)
        return x
```

full connection is like linear transformation!

data batch × swept through all layers



Training & Simple Inference

loss function: MSE, Cross-Entropy ...

```
model = Network(),...
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

optimization alg. of loss function: Stochastic Grad Descent, Adam, ...

SGD which uses momentum

```
epochs = 3
steps = 0
for e in range(epochs):
    running_loss = 0
    for images, labels in iter(trainloader):
        steps += 1
        images.resize_(images.size()[0], images.size[2]*images.size()[3])   # prepare batch
        optimizer.zero_grad()                                                # clear grads
        output = model.forward(images)                                       # 1. forward
        loss = criterion(output, labels)                                     # 2. compute loss
        loss.backward()                                                      # 3. compute grad
        optimizer.step()                                                     # 4. grad descent
        running_loss += loss.item()                                          # output
        if (steps % 40 == 0):
            print("Epoch {}, Loss {}".format(e, running_loss/40))
            running_loss = 0
```

validation() comes here!!!

```
images, labels = next(iter(testloader))
img = images[rd].resize_(1, images.size()[2]*images.size()[3])
with torch.no_grad():
    logits = model.forward(img)
ps = F.softmax(logits, dim=1)
--> plot(img, ps) ...
```

we're passing 1 image!

Validation

- Model is evaluated during training to check that its accuracy is increasing (⇒ no overfitting) while loss decreases.
- Implemented in a `validation()` function that uses the test split/ subset.

Further Topics in Jupyter Notebooks & Udacity Guide TXT.

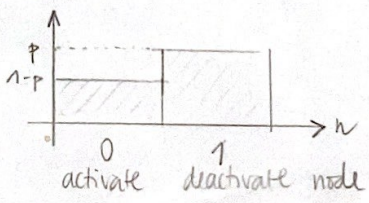
- Creual networks with arbitrary nums of hidden layers created
- Use log probabilities (log-softmax) as output activation/transformation:
  - larger numbers ⇒ more stable
  - multiplication of probs becomes addition of probs ⇒ more efficient
- Dropout is added in forward(); it's activated with model.train()
  - used only during training!
  - deactivated with model.eval() → in validation()
- Save and load networks (state & architecture params): it's saved as a dict!
  - weights & biases of each layer
  - input size
  - output size
  - hidden layer sizes
- Loading any dataset (from folders of images to pre-processed tensor datasets)
  - Data augmentation: introduce variation to generalize true network
  - Normalization: scale gray values to [-1, 1]
- Transfer learning: use pre-trained big networks to leverage DL applications
  - Last layers (classifier) implemented and trained
  - GPU device: use (CUDA) if available: models, images & labels (or any tensor) must be transferred to device: model.to(device)

QUESTIONS

- Sometimes the code produces errors on my computer; for example:
  - transforms.RandomRotation()
- Is the order of the transforms relevant?
- Dropout: [Why is it done after relu(forward(x))?
  [Does here .dropout() mean it's just activated for next passes?

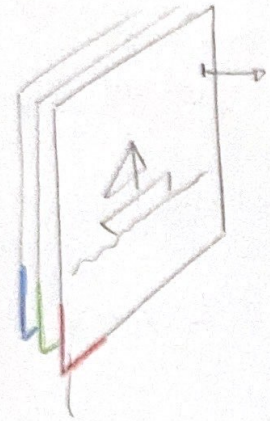
→ look at Jupyter Notebooks & fc_model.py & helper.py!

△ NO; the dropout() function randomly zeroes some of the elements following a Bernoulli distribution

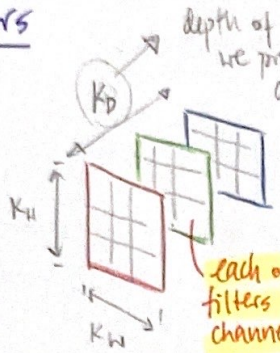


CNN LAYERS AND FEATURE VISUALIZATION (CONT.) (from page 15)

Convolutional Layers



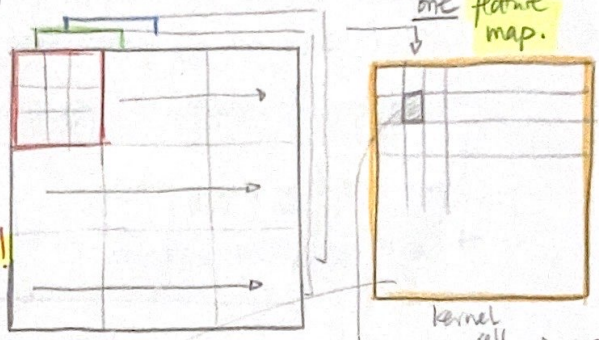
$D \times W \times H$
usually depth is $D = 3$ channels



Filter F_N: 3D tensor which is convolved in m image; usually

N = tens to hundreds

Example: Convolution with one filter F_N in 3 channel image results in one feature map.



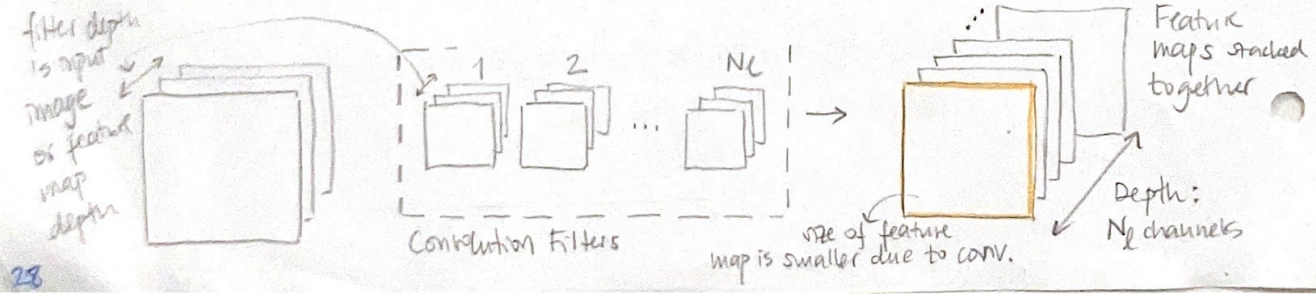
kernel cell value · image grayscale value

$$ReLU(F_N * I) = ReLU\left(\sum\sum\sum f_N \cdot i\right)$$

depth/channel · row/height · column/width

the size of the feature map is size(img) − size(kernel) + 1 — except if padding is used = extending img outwards

- Each convolution layer has N_ℓ filters
- Each of the filters is a 3D tensor and produces one output
  - the depth of the filter is the number of outputs from previous layer
    - each output is called **FEATURE MAP / ACTIVATION MAP**, and is like a channel
    - if the input image is the original preprocessed image, the filters of this first convolution layer are going to have expectedly a depth of 3: R, G, B.
- All outputs or FEATURE MAPS are stacked to form a 3D matrix of N_ℓ channels
- Therefore:
  - each 3D filter produces a feature map, which is like a channel
  - Convolutional layers can be concatenated
    - each feature map highlights patterns
    - successive convolutional layers discover patterns within patterns

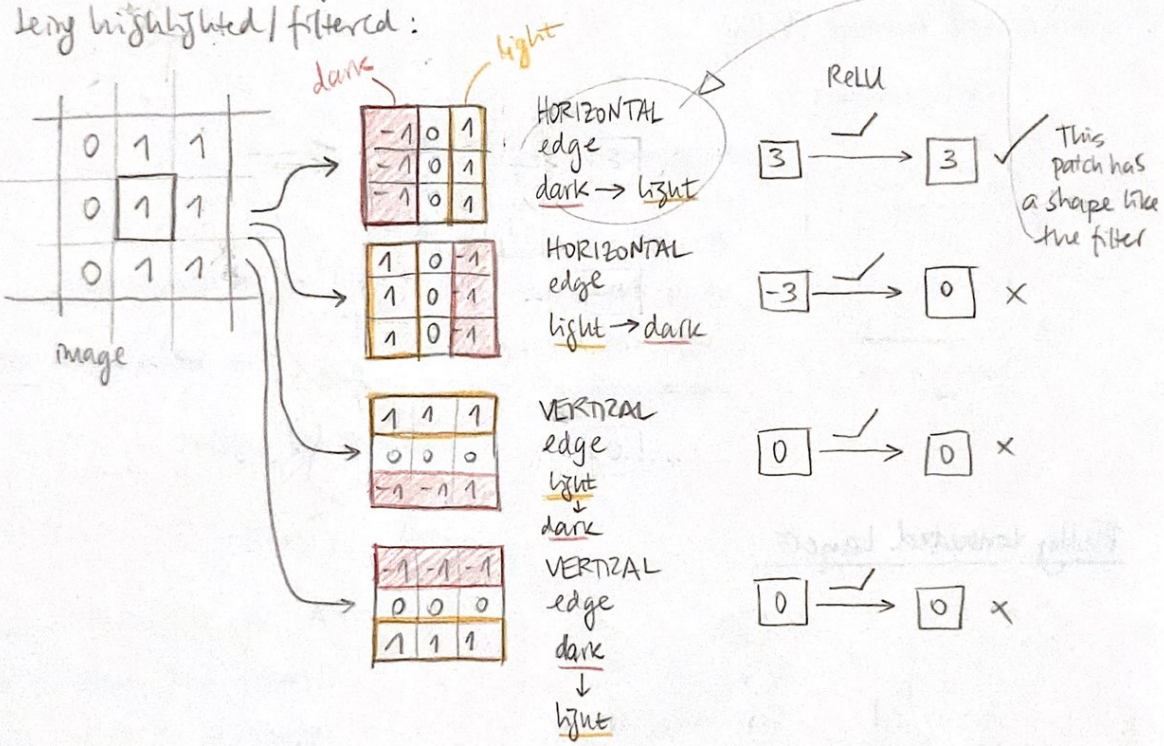


– CNN vs MLP (dense or fully connected layers)

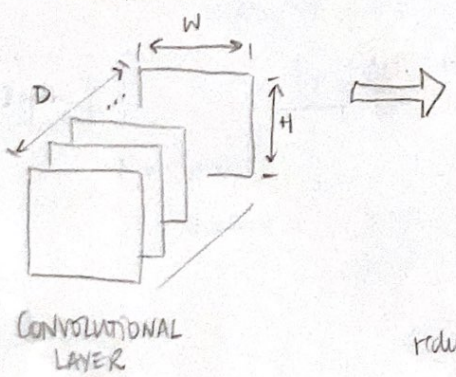
- In a fully connected or dense network all previous and successor nodes are connected, whereas in CNNs nodes are only LOCALLY connected
- Both have weights & biases, but in the case of CNNs these are in the form of FILTERS: the CNN learns which patterns to detect from the data.
- Training & inference works the same way.

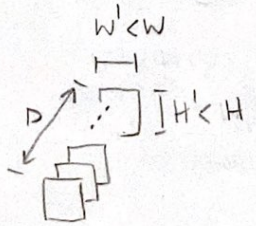
– Notes:

- images are usually normalized either to [0, 1] or even [-1, 1]
  - ÷ 255 ← .ToTensor() does that
  - [0, 1] − (0.5) mean / (0.5) std ← .Normalize() does that
- filter kernel values give an idea of what's being highlighted/filtered:



Pooling Layers: usually come after a convolutional layer and they reduce dimensionality



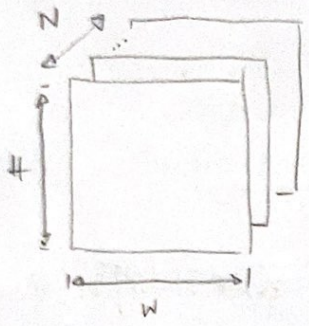


POOLING LAYER:
selection or computation of pixel values
reducing size of feature maps — BUT same depth = # feature maps

- Convolutional layers have usually many channels/filters, which results in many features
  - ⇒ many parameters ⇒
  - ⇒ OVERFITTING
  - ⇒ dimensionality reduction is the solution!

- There are several pooling layer types (see documentation of PyTorch). 2 most known:

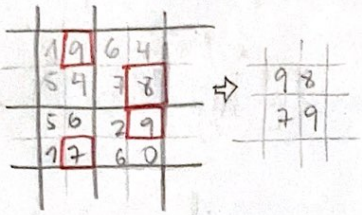
1) MaxPooling

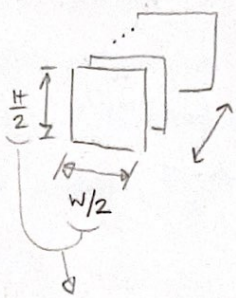


N feature maps

Parameters
- window size, eg 2x2
- stride, eg 2

for each map, separately, max pixel value within a window selected

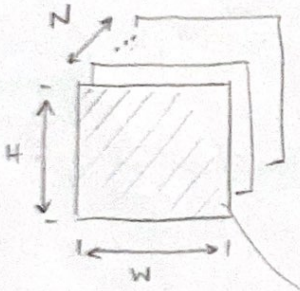




N: number of output feature maps is the same!

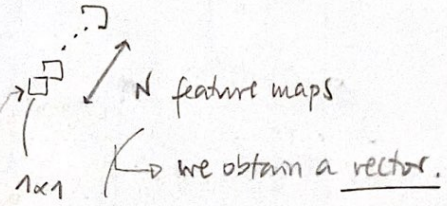
In the particular case of window size 2x2 & stride 2

2) Global Average Pooling



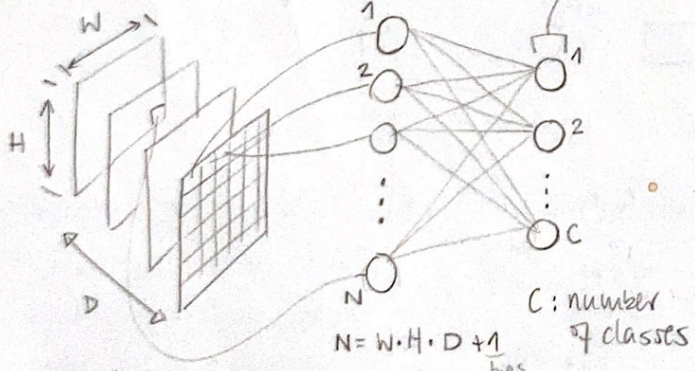
No params defined

Average pixel/node value taken for each feature map independently



average of all pixels

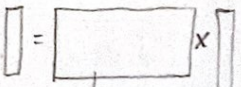
Fully Connected Layers



the output is a vector of classes!

- A fully connected layer maps linearly all pixels of the last layer to C nodes, being usually C the number of classes we want to predict.

$$\underset{\sim}{h} = \underset{\sim}{W} \cdot \underset{\sim}{x}$$



$\underset{\sim}{h}$: $C \times 1$ classes

$\underset{\sim}{W}$: $C \times (W \cdot H \cdot D + 1)$ weights bias

$\underset{\sim}{x}$: $W \cdot H \cdot D \times 1$ feature maps vector

weights & biases

- Several fully connected layers can appear at the end one after the other
  - Intermediate f.c. layers usually have ReLU activation
  - LAST f.c. layers ends up usually with a SOFTMAX or a LOGMAX
    ↳ this converts the output to be class probabilities.

- DROPOUT
  - Dropout consists in randomly deactivating nodes, which reduces overfitting
  - This is implemented in practice adding a layer which zeroes outputs
  - The probability p of deactivating a node is our param, which follows Bernoulli distrib.

Network Structure : which layers should we use when?

- To prevent overfitting
  1. Early stop: track loss & accuracy and stop if loss↓ but accuracy ↓
  2. Regularization (I understand that should be used in torch.optim? How?)
     ↳ L2 Regularization implemented through param [weight_decay] in optim function
  3. Dropout
     - Usually at the end of the network
     - Example: between fully connected layers ⇒ we prevent nodes from overdominating
  4. Batch normalization
     - Usually at the beginning of the network
     - Just as with dropout, a layer is added { nn.BatchNorm1d (after linear) / nn.BatchNorm2d (after conv2d)
     - What's this?
       For each batch, where applied, the mean of the outputs is substracted and the std. deviation divided
     - Which is the effect?
       - The values of the hidden weights don't shift around, they're more stable
       - Each layer learns more independently from others
       - We can use larger learning rates

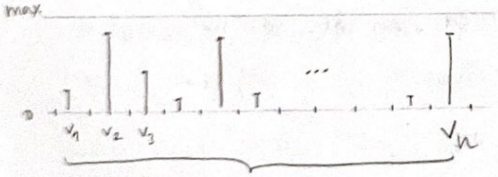
- Which layers?
  - In general: more layers help see more complex structures
    BUT for simple structures, few layers are enough!
  - LOOK AT KNOWN NETWORKS: and inspire on them! People already spent time figuring out how to arrange CNNs

- Always try changes and see what happens
  - Number of layers
  - Size of conv. kernels
  - Loss & optimization functions
  - Hyperparameters: learning rate, momentum
  - Batch size

Feature Visualization

- Sometimes the network learns from the image incorrect ones that seem to work but don't generalize
  - Example: if we show wolfs in snow, the model will learn to detect wolfs when there's snow and a dog in the image!
- Therefore, we need to visualize the feature maps learn, to check the correct ones are learnt.
- First layer
  - Usually highpass filters are learnt, which detect any kind of inclined lines or corners
- Successive layers
  - More complex shapes appear
  - As we go deeper, object parts, objects, start to appear; from layer 4/5, animals & people might appear
  - Field of view is increased in the filters (pooling does that)
  - Important note: deep layer have as input the feature/activation map from previous layers, not images! Therefore, sometimes it's hard to understand
- Last layers / linear / fully connected layers: the output activation maps are vectors!!
- ⟶ See images of conv. kernels in different layers (Matt Zeiler)

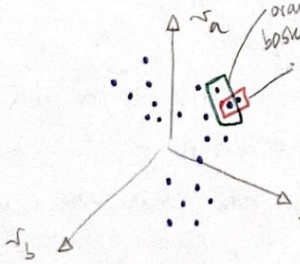
Last Feature Vector, PCA, t-SNE



$\tilde{v}$

feature vector of last layer before class layer

- The feature vector before the class layer encodes objects with a characteristic object signature



- If we plot image sample feature vectors in this feature space, similar objects are going to be close!

→ Applications of that feature space

* Similar objects can be found using k-nearest neighbors
  - as distance metric, L1 or L2 can be applied on samples

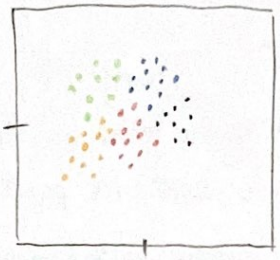
DIMENSIONALITY REDUCTION

* Principal Component Analysis (PCA) could be applied on all samples to reduce the dimensions of the feature space to a smaller significant set of features
* t-SNE (/tee-SNEE/): t-distributed stochastic neighbor embedding
  - non-linear dimensionality reduction
  - data is separated in clusters in a low dimensional feature space, eg 2D.

because t distribution is used

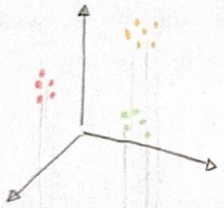
MNIST dataset

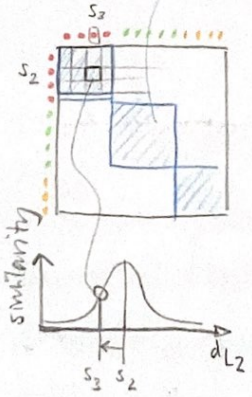
28×28 feature space



2D feature space with 10 clusters (0-9 digits)

- Method intuition



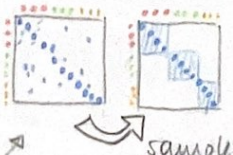


high similarity values

- similarity matrix of all samples against all samples in high dimensional space
- similarity between samples s_i & s_j is the prob. of both belonging to same cluster, which is evaluated by the distance between the samples, eg L2 distance

$d_{L_2} = \|s_i - s_j\|_2$: the scaled normal distribution of samples in cluster is computed

- samples are projected ona low dimensional space and similarity matrix is computed



samples moved in low dim space

- the matrix is expected to be sparse
- the projected samples are moved step by step and similarity matrix is recomputed so that the final matrix resembles the original one from high dim space

Occlusion, Saliency and Guided Backpropagation: What is the network looking at for the decision?

HEATMAPS: ROI relevance for class by experimenting with occlusion

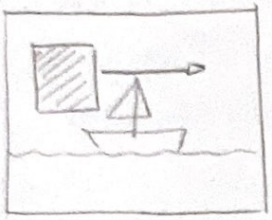
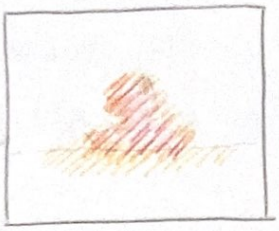


Image is occluded with a mask/box and its class computed for each box position (box is moved)



for each of the pixels in the box, the decrease of the class confidence when overlaying the box is accumulated

Relevant image areas that lead to the predicted class are computed and visualized encoded in color.

- very relevant region
- relevant region
- ...

SALIENCY MAPS

- Which pixels are most important for classifying an image?
- The gradient of the class score wrt pixel values is computed for each pixel
  - this creates a map similar to the heatmap
  - $\nabla C = \left[\frac{\partial C_1}{\partial p}, \frac{\partial C_2}{\partial p}, \dots \right]$
  - Image segmentation can be done this way: the saliency map is overlaid and thresholded to get a mask.

GUIDED BACKPROPAGATION

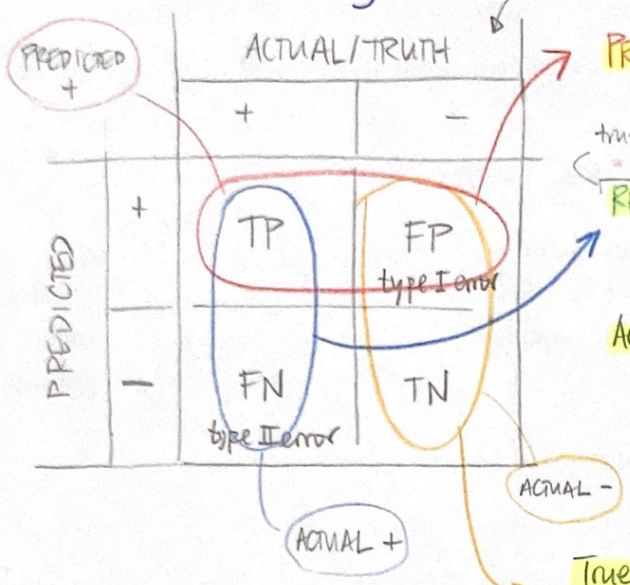
- Same idea as in the saliency map, but we look at the change in the output of a neuron/node, when a pixel value is changed dp.
- Big neuron output changes indicate that the pixel has a big influence in the neuron
- (Basically the chain rule is applied as in backpropagation)?

Some Applications with Feature Maps

- Deep Dream: selected features are amplified in an image
- Style Transfer: content and style are captured with feature maps from 2 images and then they're combined

SKIN CANCER DETECTION (optional excursus) – Mini Project

Precission, Recall, Accuracy & CONFUSION MATRIX



$$\text{PRECISION} = \frac{TP}{TP+FP}$$

"Among all patients PREDICTED to have cancer, how many actually have it?"

true positive rate

$$\text{RECALL} = \frac{TP}{TP+FN}$$

"Among all patients that ACTUALLY have cancer, how many were predicted to have it?"

$$\text{ACCURACY} = \frac{TP+TN}{TP+FP+FN+TN} = \text{TOTAL}$$

↑ how well does our prediction perform?

$$\text{True negative rate} = \frac{TN}{TN+FP} \quad \text{False positive rate} = \frac{FP}{TN+FP}$$

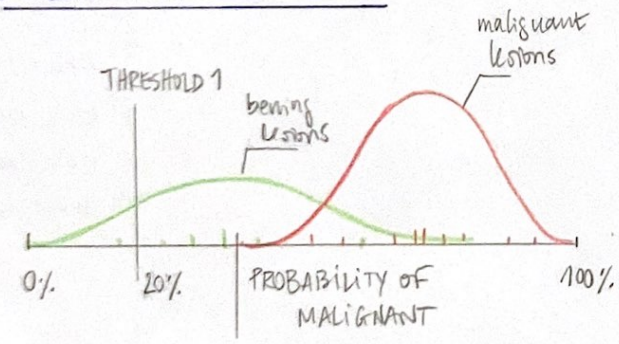
SPECIFICITY — actual negative

- FP: type I error: false alarm
  "cancer prediction in cases where there's no cancer"
- FN: type II error: missed cases
  "sane prediction in cases where there's actually cancer"

⟷ very dangerous!

- In medicine context, often appear:
  - SENSITIVITY = RECALL (= true positive rate) → "of all sick people, how many did we diagnose sick?"
  - SPECIFICITY = True negative rate → "of all healthy people, how many did we diagnose healthy?"

Detection Score Threshold



– If the distributions are known, we take THRESHOLD 2, which eliminates FNs or type II error!

– If the distributions of the benign & malignant lesions for the variable PROBABILITY OF MALIGNANT are unknown, it's better to take a low threshold, because the cost of regret is much lower!: Better to order new tests that missing a cancer diagnosis → THRESHOLD 1

ROC Curve = Receiver Operating Characteristic Curve

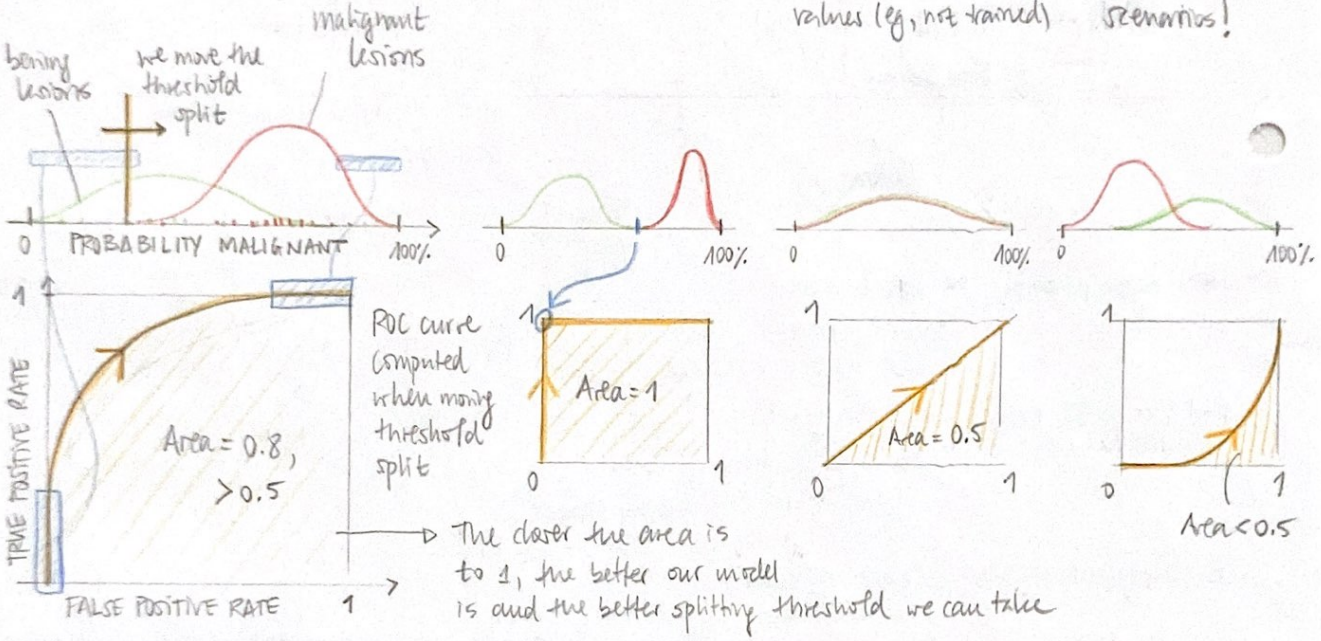
- Plot that illustrates the diagnostic ability of a binary classifier.
- There are several types of ROC plots; in any case, they are created by sweeping the threshold along the classification probability and computing true/false positive/negative rates
- Analysis of 4 cases & 2 types of ROC curves

CASE 1: usual case
Distributions of malignant & benign overlap

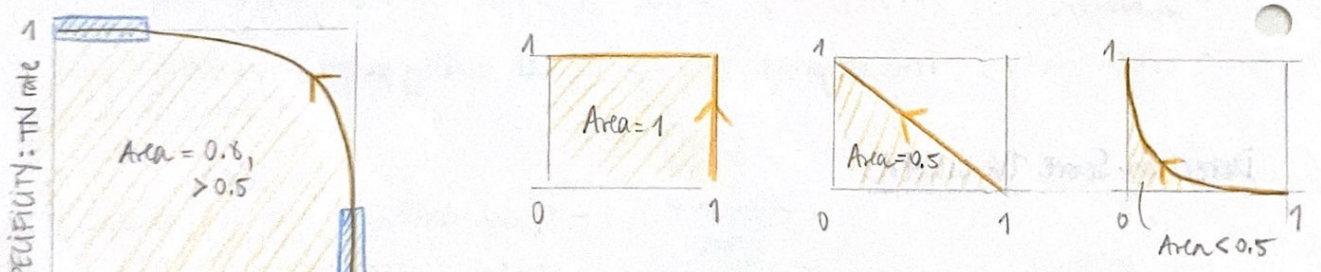
CASE 2: perfectly disjoint — nice to have scenario

CASE 3: perfect overlap — classification doesn't work, random values (eg, not trained)

CASE 4: predicted probability is the inverse to the real scenarios!



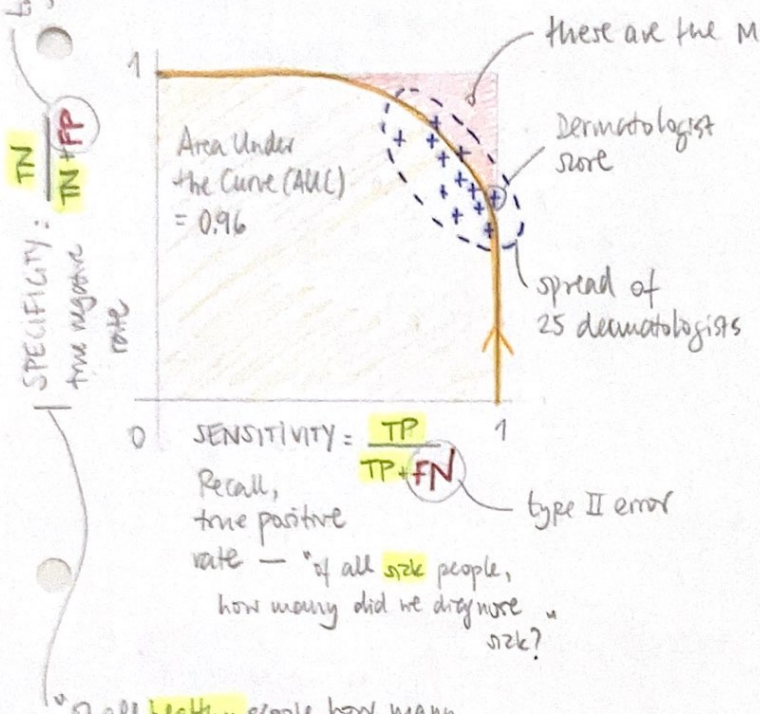
The closer the area is to 1, the better our model is and the better splitting threshold we can take



common ROC plot in medical context

In this scenario, we would actively kill people...

- Interpretation of the ROC curve (in the particular case of medical context)

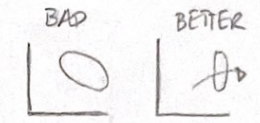


	SPECIFICITY	SENSITIVITY
HIGHER	More accurate ability to say who is healthy	More accurate ability to say who is sick
LOWER	We tend to diagnose healthy people as sick CONSERVATIVE, but more COSTLY	We miss to detect sick people BAD !!! (apparently cheaper)

↓ Increasing SPECIFICITY makes the cost more efficient

↓ SENSITIVITY is critical and should be as high as possible

- Dermatologists are usually below the curve ⇒ less performance
- The spread of the dermatologists is quite large!
- The spread should be as right as possible (high SENSITIVITY)



↳ some dermatologists are to the left: is that so because it's cheaper for the insurance in the short-term?

t-SNE Visualization (see p. 32, 33)

- Very intuitive projection of the feature space on a reduced space that allows for clustered visualization of the samples
- If visualized images of the clustered samples, similarities are clear to the human eye !!!

Visualization of what the network is looking at = SENSIBILITY ANALYSIS (see p. 34)

- What's the network looking at to take that classification decision?
- Sensibility analysis: we take samples, change them and observe how the classification output changes ⟶ this gives us an idea of what's the net looking at
- METHODS: heatmaps, saliency maps, guided backpropagation ⟶ TRY THEM & DRAW CONCLUSIONS!!

Confusion Matrix – How good is our model?

PREDICTION = DIAGNOSIS		TRUTH: sick	TRUTH: healthy
	sick	TP	FP type I error
	healthy	FN type II error	TN

type I error: healthy patients diagnosed as sick → More tests, costly

type II error: sick patients diagnosed as healthy → DANGER!

3 sicknesses

PREDICTION \ TRUTH	A	B	C	Σ
A	0.8	0.08	0.3	≠ 1!
B	0.1	0.9	0.1	
C	0.1	0.02	0.6	
Σ	1	1	1	

$P_{ij} = 0.1$ | $i = B$, $j = A$

probability of patients having A but of being diagnosed with B

i: DIAGNOSIS

j: TRUTH

probability of getting missdiagnosed if you suffer C: 0.4 = 0.1 + 0.3

- Main diagonal should be 1 and rest $\emptyset$ ⇒ identity
- Everything off the diagonal is a confusion!
- Dermatologists had a more diffuse confusion matrix ⇒ more miss classifications (sparse)

POPULAR NETWORKS (personal Review)

- AlexNet 2012
- VGG-16 2014
- ResNet50 2015 (Microsoft)
- Inception-v3 (Google) 2015
- LeNet 1998

AlexNet Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton
U. Toronto
2012

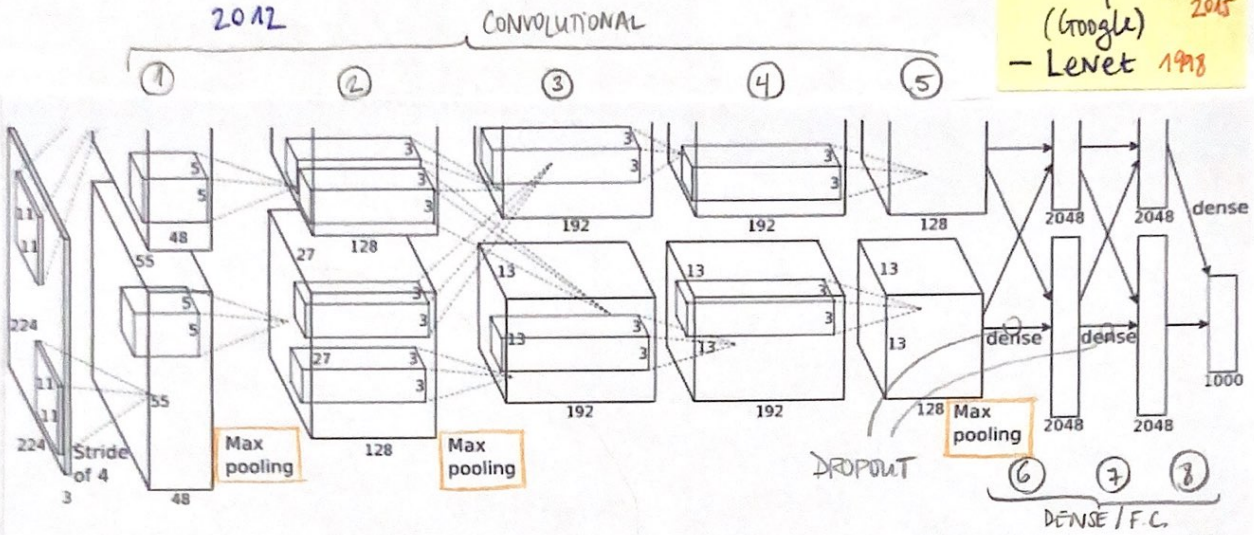
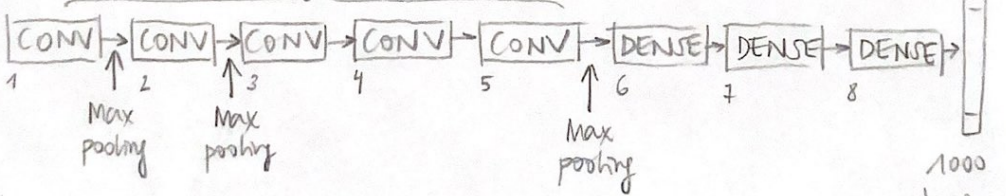


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

- Competed in the ImageNet Large Scale Visual Recognition Challenge (ILSVR) in 2012
  - top-5 error: 15.3%: 10.8 points better than best ever!
  - people started paying attention to AI / Deep Learning

Created by Prof. Fei-Fei Li, 2007
15 M+ images, labelled; 22.000 classes/categories
→ for the challenge 1.000 classes with ~1000 images each used

- Properties
  - 2 GPUs used: image top & bottom → faster
  - ReLU: better than tanh(x) or sigmoid
  - 8 layers:



  - 60 M parameters, 650K neurons
  - Data augmentation used for training (to prevent overfitting)
  - Dropout with p = 0.5 in the first 2 fully connected (dense) layers
  - Momentum 0.9, weight decay 0.0005

LeNet LeCun, Yann
1989 – 1998

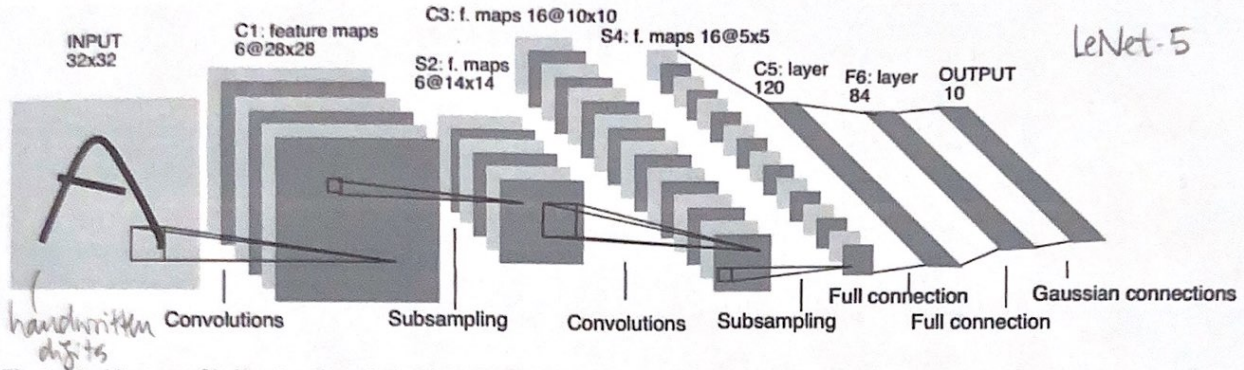
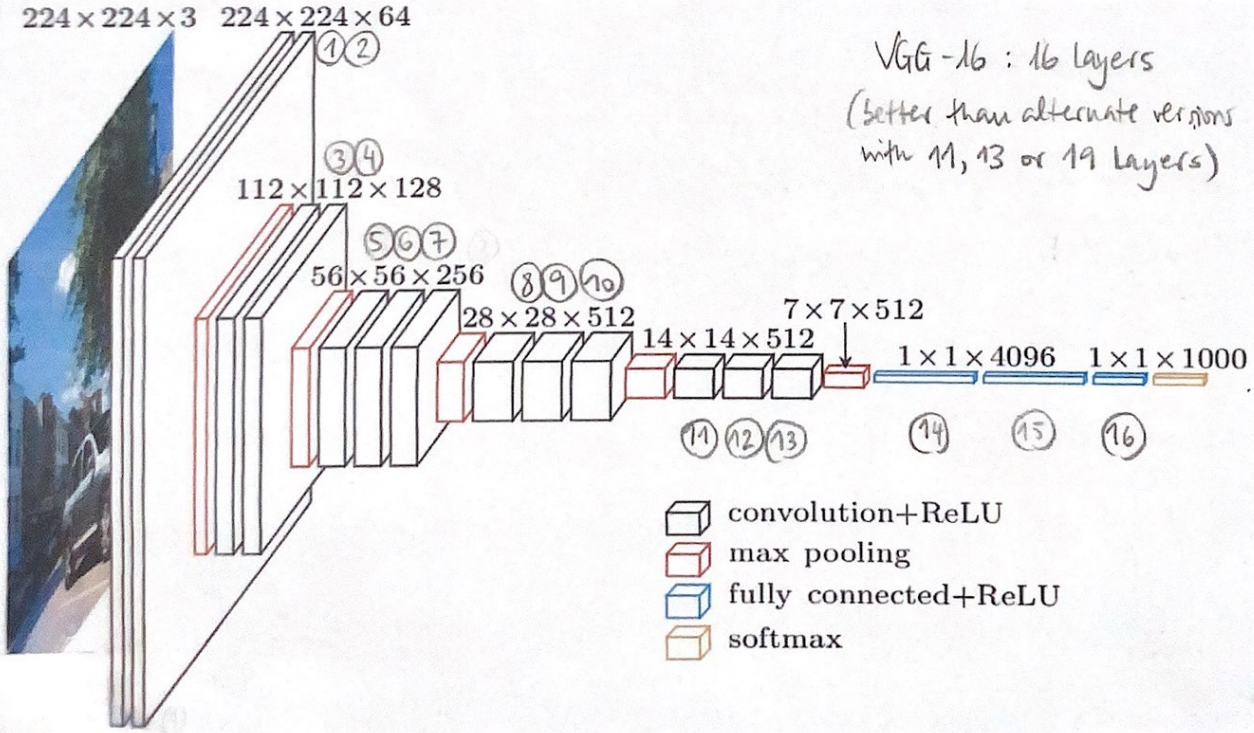


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical. (1998)

- Very old architecture, but the inspiration/base for later architectures like AlexNet
- LeNet was presented by Yann LeCun and the idea evolved between 1989 - 1998; the final implementation is LeNet-5 (the one that's usually meant)
- LeCun applied for the first time
  - Convolutional layers
  - Backpropagation

VGG-16 Very Deep Convolutional Networks for Large-Scale Image Recognition
Simonyan, Zisserman – U. Oxford, UK, 2015



VGG-16 : 16 Layers
(better than alternate versions with 11, 13 or 19 Layers)

– Created by the Visual Geometry Group: VGG

– Properties

• They discovered smaller filters (3×3) are enough ⇒ less params (weights) ⇒ faster & less images needed! and less overfitting!

• They tested increasing and local normalization response (LNR)
  - proposed by AlexNet
  - neuron output inhibition strategy inspired in human biology; ReLU is not bounded and normalizing groups of outputs was supposed to yield better results
  - BUT: nowadays not used anymore, because not efficient; much better: Dropout, Batch Normalization

• RESULT (BEST): VGG with 16 layers (better than 19 layers!)
  ↳ ablation study: remove layers and see what happens

• Input images were 224×224, but multi-scale training was performed to generalize to different object scales.
  - Take bigger picture sizes and crop to 224×224
  - This improves accuracy

• They did something with the final dense layers during testing (but not training) — they were replaced by 1×1 convolutions, ie spatial averages ???

ResNet He, Zhang, Ren, Sun – Microsoft
2015 (from Medium article by Sik-Ho Tsang)

– ResNet can have very deep networks (up to 152 layers)
  ↳ this can be achieved by learning the residual representation instead of the actual signal / feature maps of the images

– They introduce skip or shortcut connections: input from previous layer is taken without any modification
  ↳ They call plain networks all previous that don't do that (AlexNet, VGG, etc) and can't achieve so many layers because of the vanishing / exploding gradients problem

– VANISHING/EXPLODING GRADIENT Problem of plain networks
  - It's not uncommon to have small or large derivatives in the network
  - $\partial E/\partial w$ is a multiplication of derivatives (s. p. 19), due to the chain rule (backpropagation)
  - If the network is deep, this product is VERY SMALL/LARGE!

PROBLEM!

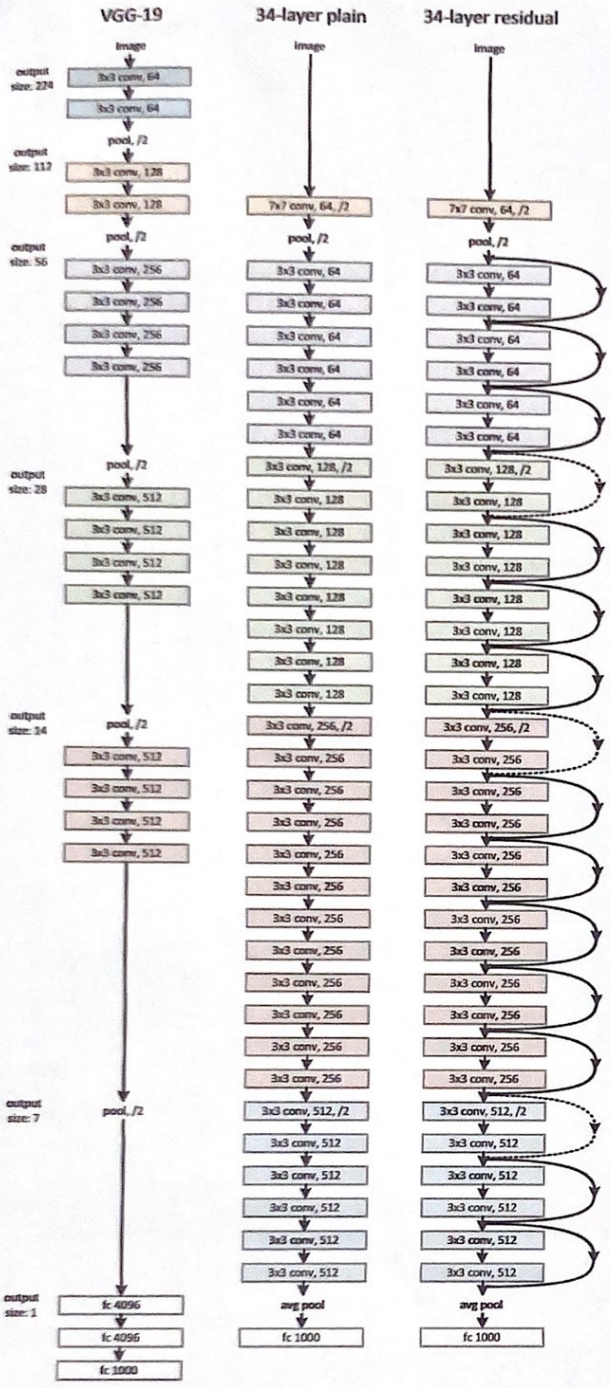
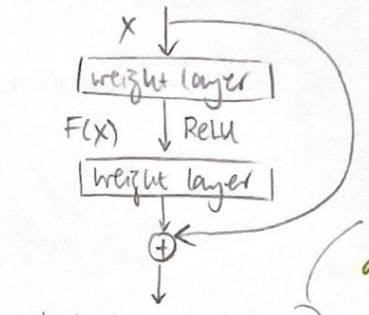


Figure 3. Example network architectures for ImageNet. **Left:** the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle:** a plain network with 34 parameter layers (3.6 billion FLOPs). **Right:** a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

— The skip / shortcut connection solves the problem of the vanishing gradient:



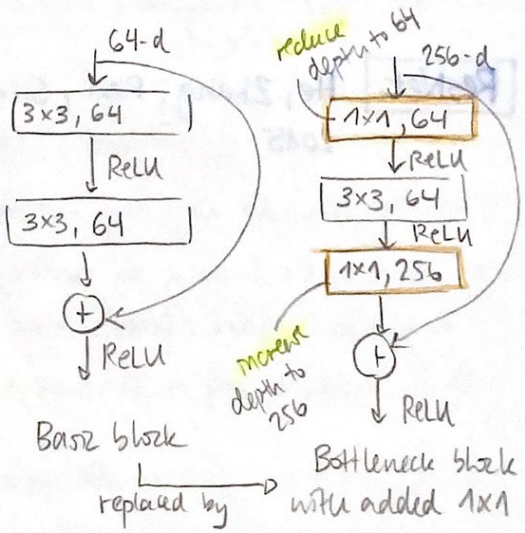
↓

BUT the weights are optimized during learning with the RESIDUAL MAPPING:

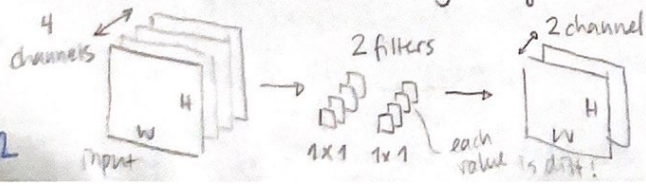
$$F(x) = H(x) - x$$

— They additionally use a BOTTLENECK design with 1x1 convolutions

- Bottleneck designs reduce complexity
- Adding 1x1 convolutions we go from 34 layers to 50 layers: ResNet50



— A 1x1 convolution layer reduces the number of connections (= parameters) without degrading network



- a 1x1 convolution can be used to modify the depth of the signal:
  - reduce / increase

- Result/Conclusion
  - Networks can be deeper while
    - having less parameters ⇒ faster to train and use] → other shallower architectures have much more parameters!
    - increasing accuracy] → in other architectures, increasing layers up from a certain point (eg VGG 16), decreases accuracy due to vanishing gradient issues
  - As we increase layers { accuracy ↑ / speed ↓ } ⇒ ResNet50 is often a good alternative nowadays (2020)

Inception v3

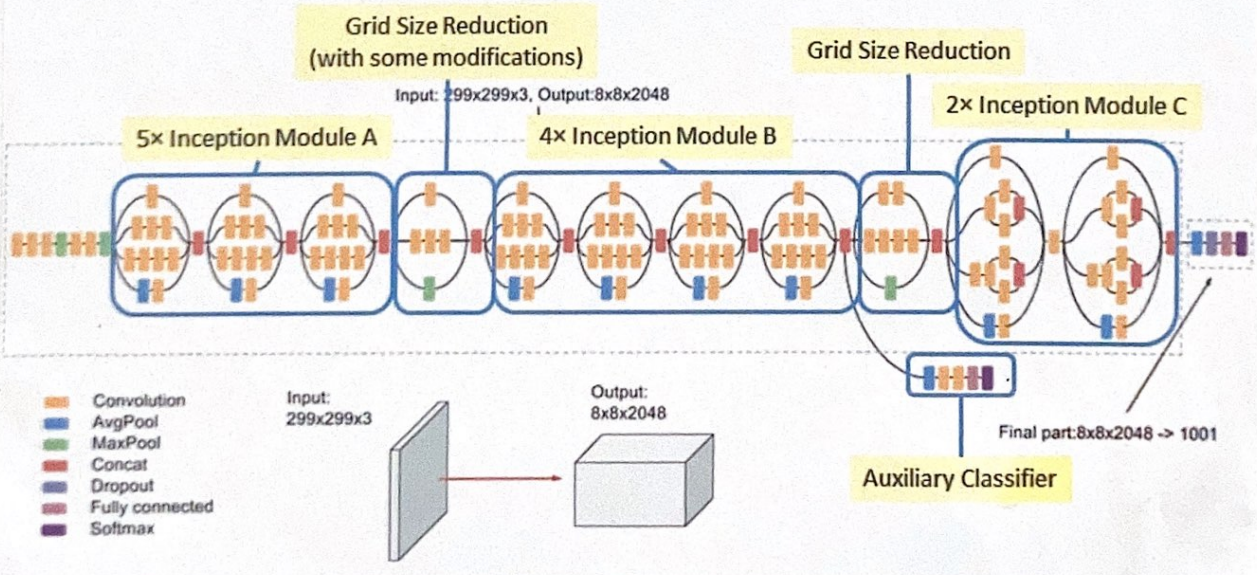
Szegedy, Vanhoucke, Ioffe, Shlens, Wojna — Google 2015 (from Medium article by Sik-Ho Tsang)

- There are several versions
  - The first version (2015), was called GoogLeNet
  - Later versions introduced BATCH NORMALIZATION and extended FACTORIZATION

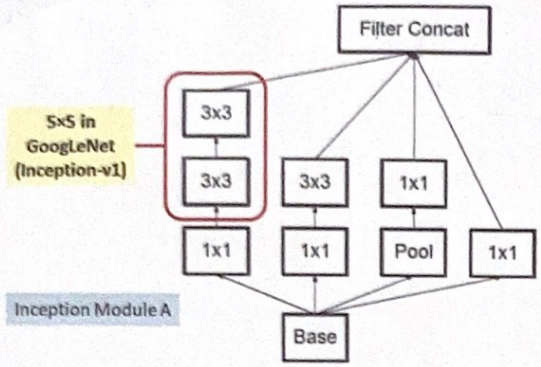
BATCH NORMALIZATION (see p. 31): output of each batch normalized (with mean & std) to avoid shifting of weights

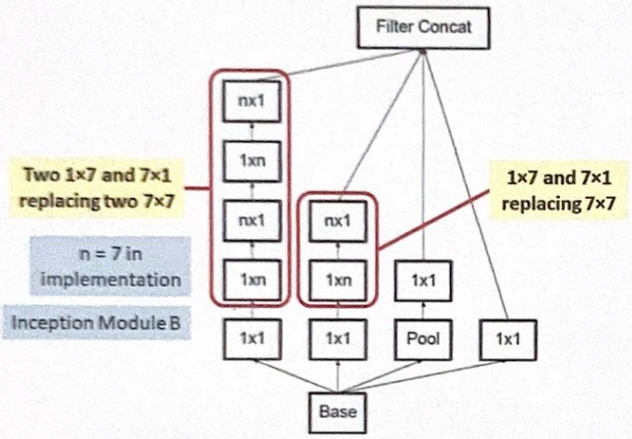
FACTORIZATION: larger filters (eg 5×5) are replaced by smaller ones (eg 3×3) that work in parallel and then result is concatenated → this reduces number of params without decreasing network efficiency

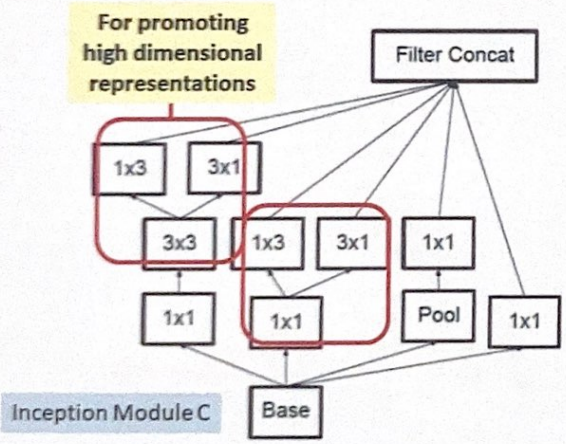
Factorization example
- 5×5 filter : 5×5 = 25 params
- 2 3×3 filters : 2×3×3 = 18 params



- They define 3 types of factorization modules, here informally named A, B, C (i. mog







- They have an extra branch from which an auxiliary classifier emerges
  - It's used as a regularizer (HOW?)
- They also use a grid size reduction module
  - : AlexNet & VGGNet use feature map downsizing with max pooling
  - : Maxpooling after a convolution (can result either coarse (pooling) or expensive (convolution operation done anyway)
  - . Their proposed grid reduction module processes feature maps in 2 parallel branches using convolutions and pooling and it leads to a LESS EXPENSIVE & STILL EFFICIENT network.
- Conclusion (my observations)
  - What they aim is to reduce the cost of training / using the network while still having a complex & deep architecture; FACTORIZATION & GRID SIZE REDUCTION are used for that.

DenseNet Huang, Liu, van der Maaten, Weinberger, 2018/7
Best Paper Award CVPR 2017

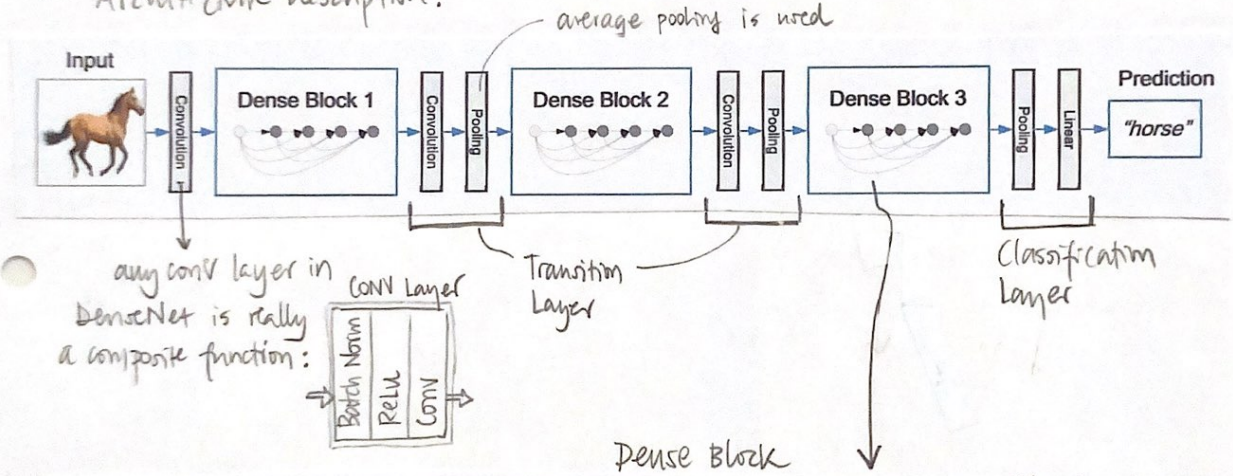
Summarized from the review by Mukul Khanna @medium

- Issue for deep networks: VANISHING GRADIENT: with very deep networks, the gradient becomes small when back-propagating it (recall chain rule) and first layers don't learn efficiently
  - Several architectures have been introduced to tackle this, eg ResNets, Inception,
  - All of them create some kind of paths between layers

- Main idea:
  - The architecture is composed of dense blocks with layers
  - Each layer from a dense block recives feature maps from all preceeding layers and these are fused through CONCATENATION, not summation (like in ResNets)

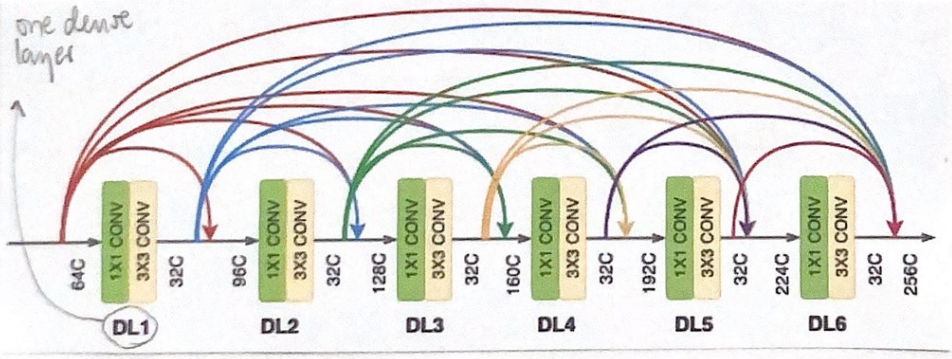
- Consequences / Contributions
  - Vanishing gradient is alleviated
  - Number of parameters is considerably reduced!
  - Features are learnt collectively: there are reused and propagated

- Architecture description:



any conv layer in DenseNet is really a composite function:

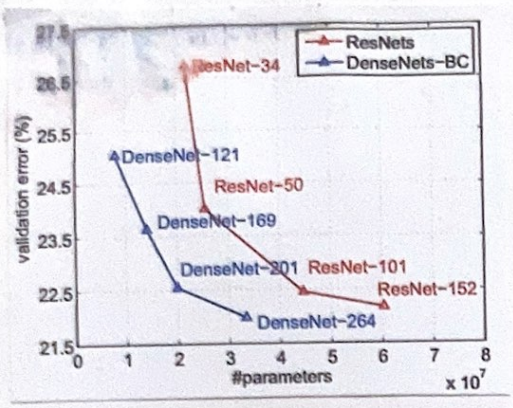
Dense Block



- There are several DenseNets, all have 4 dense blocks, 3 transition layers, 1 classification layer
- The difference is the # of layers within a block

- Example: DenseNet-121:

Block	1	2	3	4
Dense Layers	6	12	24	16



Results: Comparision to ResNet

DenseNets require much less parameters to achieve the same accuracy !!!

recall this architecture also targets the VANISHING GRADIENT problem!

Udacity CV | 1 | INTRO CV 1

- Image representation
  - RGB channels
  - Pixels & image access & histograms
  - Color spaces
  - Basic classification
- Fourier
- Convolutional filters { low-pass: Gauss; high-pass: Sobel; general
- Canny edges
- Hough transform: lines & circles
- Haar cascades: classifiers trained for edge detection

Udacity CV | 1 | INTRO CV 2

- Harris corner detection
- Morphological operations
- Contours: contour features (shape info)
- Image moments
- K-means clustering
- Feature vectors / descriptors
- ORB
- HOG
- → Deep Learning intro

UDACITY CV | 1 | Deep Learning

- Perceptron
- MLP
- Feed-forward
- Error / loss
- Gradient descend
- Backpropagation
- Training
  - split: train / test
  - overfitting
  - early stopping
  - regularization
  - dropout
- (- Training)
  - random start
  - other activation functions
  - SGD
  - learning rate
  - momentum
- Pytorch basic usage