

Computational Intelligence

Winter Term 2024/25

Prof. Dr. Günter Rudolph
 Computational Intelligence
 Fakultät für Informatik
 TU Dortmund

- Deep Neural Networks
 - Model
 - Training

- Convolutional Neural Networks
 - Model
 - Training

DNN = Neural Network with > 3 layers

we know: L = 3 layers in MLP sufficient to describe arbitrary sets

What can be achieved by more than 3 layers?

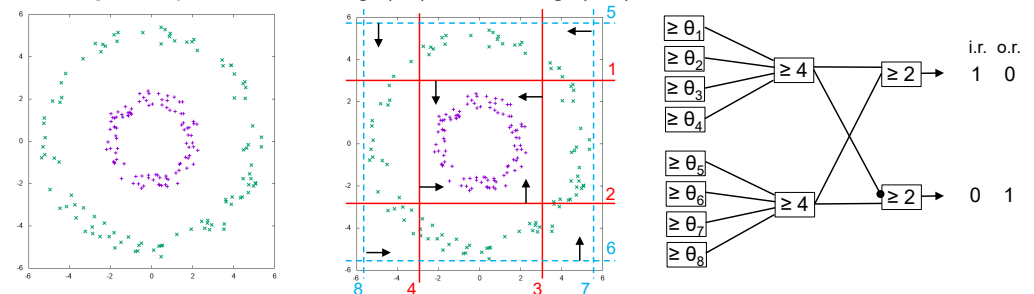
information stored in weights of edges of network
 → more layers → more neurons → more edges → more information storable

Which additional information storage is useful?

traditionally : handcrafted features fed into 3-layer perceptron
 modern viewpoint: let L-k layers learn the feature map, last k layers separate!

advantage:
 human expert need not design features manually for each application domain
 ⇒ no expert needed, only observations!

example: separate 'inner ring' (i.r.) / 'outer ring' (o.r.) / 'outside'



⇒ MLP with 3 layers and 12 neurons

Is there a simpler way?

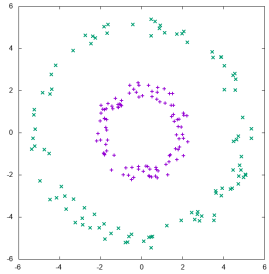
observations $(x, y) \in \mathbb{R}^n \times \mathbb{B}$ feature map $F(x) = (F_1(x), \dots, F_m(x)) \in \mathbb{R}^m$

feature = measurable property of an observation or
 numerical transformation of observed value(s)

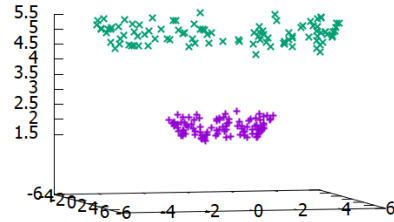
⇒ find MLP on transformed data points $(F(x), y)$

example: separate 'inner ring' / 'outer ring'

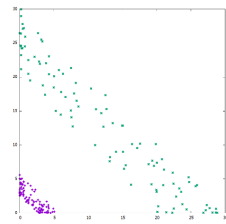
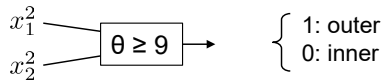
- feature map $F(x) = (x_1, x_2, \sqrt{x_1^2 + x_2^2}) \in \mathbb{R}^3$



2D → 3D



- feature map $F(x) = (x_1^2, x_2^2) \in \mathbb{R}^2$



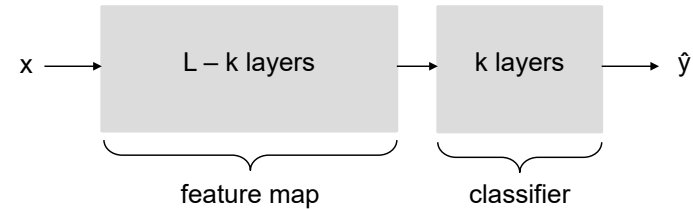
but: how to find useful features?

→ typically designed by experts with domain knowledge

→ traditional approach in classification:

1. design & select appropriate features
2. map data to feature space
3. apply classification method to data in feature space

modern approach via DNN: learn feature map and classification simultaneously!



proven: MLP can approximate any continuous map with arbitrary accuracy

contra:

- danger: overfitting
 - need larger training set (expensive!)
 - optimization needs more time
- response landscape changes
 - more sigmoidal activations
 - gradient vanishes
 - small progress in learning weights

countermeasures:

- regularization / dropout
 - data augmentation
 - parallel hardware (multi-core / GPU)
- not necessarily bad
 - change activation functions
 - gradient does not vanish
 - progress in learning weights

vanishing gradient: (underlying principle)

forward pass $y = f_3(f_2(f_1(x; w_1); w_2); w_3)$ $f_i \approx$ activation function

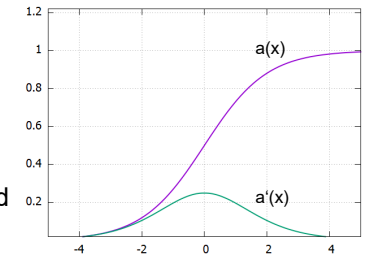
backward pass $(f_3(f_2(f_1(x; w_1); w_2); w_3))' = f_3'(f_2(f_1(x; w_1); w_2); w_3) \cdot f_2'(f_1(x; w_1); w_2) \cdot f_1'(x; w_1)$ **chain rule!**

→ repeated multiplication of values in (0,1) → 0

vanishing gradient: $a(x) = \frac{e^x}{e^x + 1} = \frac{1}{1 + e^{-x}} \rightarrow a'(x) = a(x) \cdot (1 - a(x))$

$\forall x \in \mathbb{R} : a(x) \cdot (1 - a(x)) \leq \frac{1}{4} \Leftrightarrow \left(a(x) - \frac{1}{2}\right)^2 \geq 0$

\Rightarrow gradient $a'(x) \in [0, \frac{1}{4}]$



principally: desired property in learning process!

if weights stabilize such that neuron almost always either fires [i.e., $a(x) \approx 1$] or not fires [i.e., $a(x) \approx 0$] then gradient ≈ 0 and the weights are hardly changed

\Rightarrow leads to convergence in the learning process!

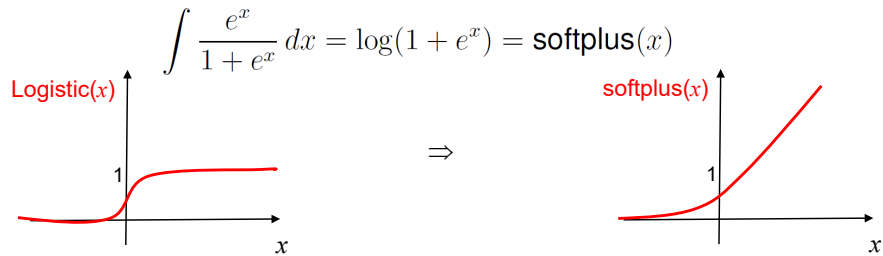
while learning, updates of weights via partial derivatives:

$\frac{\partial f(w, u; x, z^*)}{\partial w_{ij}} = 2 \sum_{k=1}^K [a(u'_k y) - z_k^*] \cdot \underbrace{a'(u'_k y)}_{\leq \frac{1}{4}} \cdot u_{jk} \cdot \underbrace{a'(w'_j x)}_{\leq \frac{1}{4}} \cdot x_i$ (L= 2 layers)

\Rightarrow in general $f_{w_{ij}} = O(4^{-L}) \rightarrow 0$ as $L \uparrow$ $L \leq 3$: effect neglectable; but $L \gg 3$

non-sigmoid activation functions

$$\int \mathbb{1}_{[x \geq 0]}(x) dx = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} = \max\{0, x\} = \text{ReLU}(x)$$



dropout

- applied for regularization (against overfitting)
- can be interpreted as inexpensive approximation of **bagging**



aka: bootstrap aggregating, model averaging, ensemble methods

create k training sets by drawing with replacement
 train k models (with own exclusive training set)
 combine k outcomes from k models (e.g. majority voting)

- parts of network is effectively switched off
 e.g. multiplication of outputs with 0,
 e.g. use inputs with prob. 0.8 and inner neurons with prob. 0.5
- gradient descent on switching parts of network
 → artificial perturbation of greediness during gradient descent
- can reduce computational complexity if implemented sophisticatedly

data augmentation (counteracts overfitting)

- extending training set by slightly perturbed true training examples
- best applicable if inputs are **images**: translate, rotate, add noise, resize, ...



- if x is **real vector** then adding e.g. small gaussian noise
 → here, utility disputable (artificial sample may cross true separating line)

extra costs for acquiring additional annotated data are **inevitable!**

stochastic gradient descent

- partitioning of training set B into **(mini-) batches** of size b

traditionally: 2 extreme cases

update of weights

- after each training example b = 1
- after all training examples b = |B|

now:

update of weights

- after b training examples
 where 1 < b < |B|

- search in subspaces → counteracts greediness → better generalization
- accelerates optimization methods (parallelism possible)

choice of batch size b

- b large ⇒ better approximation of gradient
- b small ⇒ better generalization

} often b ≈ 100 (empirically)

b also depends on available hardware
 b too small ⇒ multi-cores underemployed

cost functions

- regression

N training samples (x_i, y_i)

insist that $f(x_i; \theta) = y_i$ for $i=1, \dots, N$

if $f(x; \theta)$ linear in θ then $\theta^T x_i = y_i$ for $i=1, \dots, N$ or $X\theta = y$

\Rightarrow best choice for θ : least square estimator (LSE)

$$\Rightarrow (X\theta - y)^T (X\theta - y) \rightarrow \min_{\theta}$$

in case of MLP: $f(x; \theta)$ is nonlinear in θ

\Rightarrow best choice for θ : (nonlinear) least square estimator; aka TSSE

$$\Rightarrow \sum_i (f(x_i; \theta) - y_i)^2 \rightarrow \min_{\theta}$$

cost functions

- classification

N training samples (x_i, y_i) where $y_i \in \{1, \dots, C\}$, $C = \#classes$

\rightarrow want to estimate probability of different outcomes for unknown sample

\rightarrow decision rule: choose class with highest probability (given the data)

idea: use maximum likelihood estimator (MLE)

= estimate unknown parameter θ such that likelihood of sample x_1, \dots, x_N gets maximal as a function of θ

likelihood function

$$L(\theta; x_1, \dots, x_N) := f_{X_1, \dots, X_N}(x_1, \dots, x_N; \theta) = \prod_{i=1}^N f_X(x_i; \theta) \rightarrow \max_{\theta}$$

here: random variable $X \in \{1, \dots, C\}$ with $P\{X = i\} = q_i$ (true, but unknown)

\rightarrow we use relative frequencies of training set x_1, \dots, x_N as estimator of q_i

$$\hat{q}_i = \frac{1}{N} \sum_{j=1}^N \mathbb{1}_{[x_j=i]} \Rightarrow \text{there are } N \cdot \hat{q}_i \text{ samples of class } i \text{ in training set}$$

\Rightarrow the neural network should output \hat{p} as close as possible to \hat{q} ! [actually: to q]

$$\text{likelihood } L(\hat{p}; x_1, \dots, x_N) = \prod_{k=1}^N P\{X_k = x_k\} = \prod_{i=1}^C \hat{p}_i^{N \cdot \hat{q}_i} \rightarrow \max!$$

$$\log L = \log \left(\prod_{i=1}^C \hat{p}_i^{N \cdot \hat{q}_i} \right) = \sum_{i=1}^C \log \hat{p}_i^{N \cdot \hat{q}_i} = N \underbrace{\sum_{i=1}^C \hat{q}_i \cdot \log \hat{p}_i}_{-H(\hat{q}, \hat{p})} \rightarrow \max!$$

\Rightarrow maximizing $\log L$ leads to same solution as minimizing **cross-entropy** $H(\hat{q}, \hat{p})$

in case of *classification*

$$\text{use softmax function } P\{y = j | x\} = \frac{e^{w_j^T x + b_j}}{\sum_{i=1}^C e^{w_i^T x + b_i}} \text{ in output layer}$$

\rightarrow multiclass classification: probability of membership to class $j = 1, \dots, C$

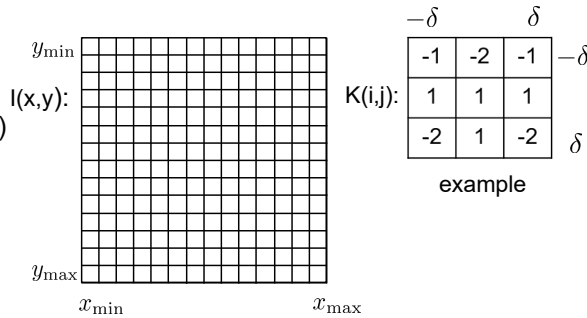
\rightarrow class with maximum excitation $w \cdot x + b$ has maximum probability

\rightarrow decision rule: element x is assigned to class with maximum probability

most often used in graphical applications (2-D input; also possible: k-D tensors)

layer of CNN = 3 stages

1. convolution
2. nonlinear activation (e.g. ReLU)
3. pooling



1. Convolution

local filter / kernel $K(i, j)$ applied to each cell of image $I(x, y)$

$$S(x, y) = (K * I)(x, y) = \sum_{i=-\delta}^{\delta} \sum_{j=-\delta}^{\delta} I(x + i, y + j) \cdot K(i, j)$$

ranges:

- without padding: $x = x_{\min} + \delta, \dots, x_{\max} - \delta, y = y_{\min} + \delta, \dots, y_{\max} - \delta$

example: edge detection with Sobel kernel

→ two convolutions

$$K_x = \begin{pmatrix} -1, 0, 1 \\ -2, 0, 2 \\ -1, 0, 1 \end{pmatrix} \text{ yields } S_x \quad K_y = \begin{pmatrix} -1, -2, -1 \\ 0, 0, 0 \\ 1, 2, 1 \end{pmatrix} \text{ yields } S_y$$

$$S(x, y) = \sqrt{S_x(x, y)^2 + S_y(x, y)^2}$$



original image $I(x,y)$



image $S(x,y)$ after convolution

filter / kernel

well known in image processing; typically hand-crafted!

here: values of filter matrix learnt in CNN !

actually: many filters active in CNN

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{pmatrix}$$

e.g. horizontal line detection

stride

= distance between two applications of a filter (horizontal s_h / vertical s_v)

→ leads to smaller images if s_h or $s_v > 1$

padding

= treatment of border cells if filter does not fit in image

- “valid“ : apply only to cells for which filter fits → leads to smaller images
- “same“ : add rows/columns with zero cells; apply filter to all cells (→ same size)

2. nonlinear activation

$$a(x) = \text{ReLU}(x^T W + c)$$

3. pooling

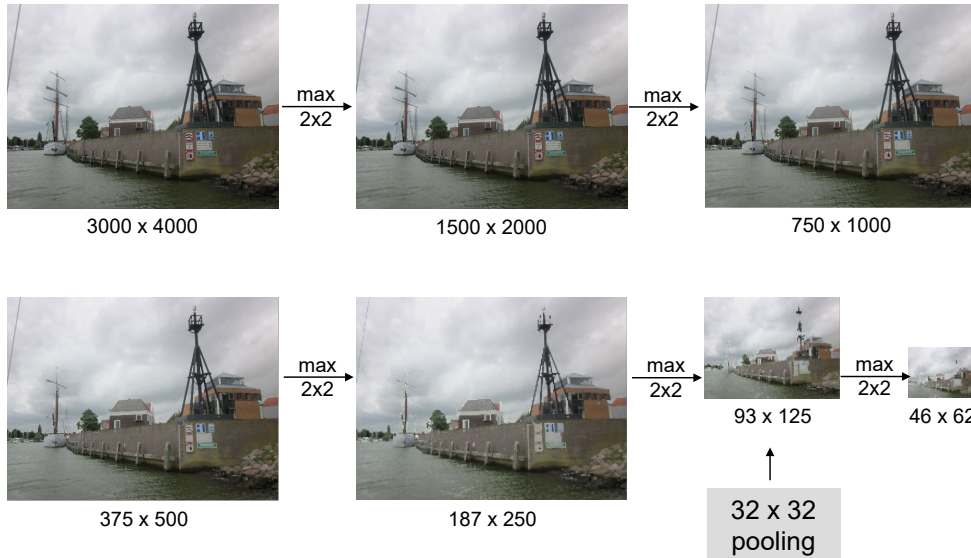
in principle: summarizing statistic of nearby outputs

e.g. **max-pooling** $m(i,j) = \max(I(i+a, j+b) : a,b = -\delta, \dots, 0, \dots \delta)$ for $\delta > 0$

- also possible: mean, median, matrix norm, ...

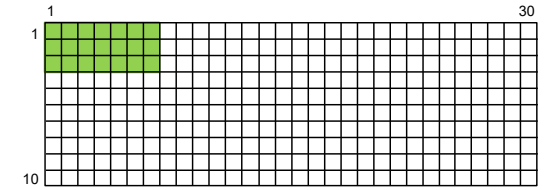
- can be used to reduce matrix / output dimensions

example: max-pooling 2x2 (iterated), stride = 2



Pooling with Stride

c_{in} : columns of input
 r_{in} : rows of input
 f_c : columns of filter
 f_r : rows of filter
 s_c : stride for columns
 s_r : stride for rows



How often fits the filter in image horizontally?

$$\begin{aligned} \text{pos}_1 &= 1 \\ \text{pos}_2 &= \text{pos}_1 + s_c \\ \text{pos}_3 &= \text{pos}_2 + s_c = (\text{pos}_1 + s_c) + s_c = \text{pos}_1 + 2 \cdot s_c \\ &\vdots \\ \text{pos}_k &= \text{pos}_1 + (k - 1) \cdot s_c \end{aligned}$$

thus, find largest k such that

$$\begin{aligned} \text{pos}_1 + (k - 1) \cdot s_c + (f_c - 1) &\leq c_{in} \\ \Leftrightarrow (k - 1) \cdot s_c + f_c &\leq c_{in} \\ \Leftrightarrow k &\leq (c_{in} - f_c) / s_c + 1 \quad (\text{integer division!}) \end{aligned}$$

$$\Rightarrow k = \left\lfloor \frac{c_{in} - f_c}{s_c} \right\rfloor + 1 = c_{out}$$

[analog reasoning for rows!]

image size : $r_{in} \times c_{in}$
 filter size : $f_r \times f_c$

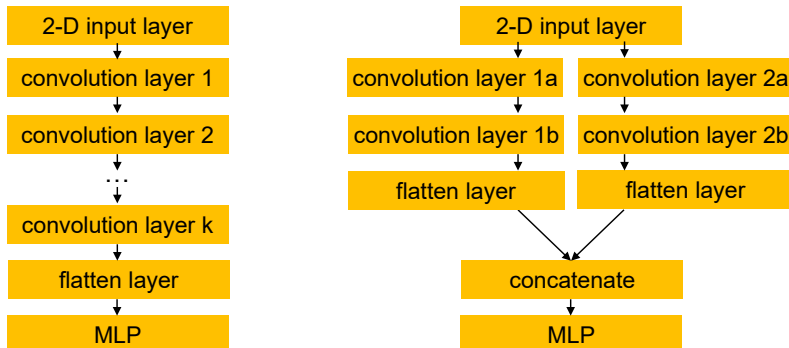
assumptions:

$f_c \leq c_{in}$
 $f_r \leq r_{in}$
 padding = valid

CNN architecture:

- several consecutive convolution layers (also parallel streams); possibly dropouts
- flatten layer (→ converts k-D matrix to 1-D matrix required for MLP input layer)
- fully connected MLP

examples:



Popular CNN Architectures

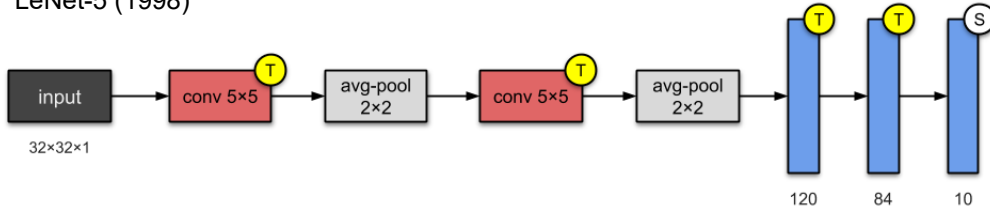
<https://towardsdatascience.com>

Name	Year	Depth	#Params
LeNet	1998		
AlexNet	2012		> 60 M
VGG16	2014	23	> 23 M
Inception-v1	2014		
ResNet50	2014		> 25 M
Inception-v3	2015	159	
Xception	2016	126	> 22 M
InceptionResNet	2017	572	> 55 M
...			

Popular CNN Architectures

<https://towardsdatascience.com>

LeNet-5 (1998)

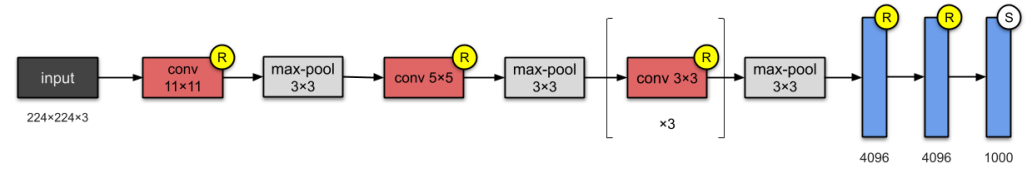


T = tanh
S = softmax

Popular CNN Architectures

<https://towardsdatascience.com>

AlexNet (2012)



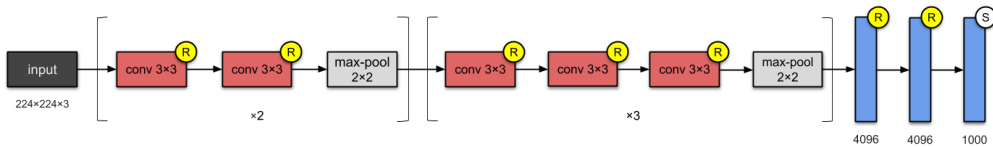
T = tanh
R = ReLU
S = softmax

Used dropout

Popular CNN Architectures

<https://towardsdatascience.com>

VGG-16 (2014)



T = tanh
R = ReLU
S = softmax

Deeper than AlexNet