technische universität
dortmund

Computational Intelligence

Winter Term 2019/20

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering (LS 11)

Fakultät für Informatik

TU Dortmund

- Deep Neural Netwoks
  - Model
  - Training

- Convolutional Neural Netwoks
  - Model
  - Training

DNN = Neural Network with > 3 layers

we know: 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-1 layers learn the feature map, last layer separates!

advantage:
human expert need not design features manually for each application domain

contra:

- danger: overfitting
  - → need larger training set (expensive!)
  - → optimization needs more time
- response landscape changes
  - → more sigmoidal activiations
  - → 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:

forward pass $y = f_3(f_2(f_1(x; w_1); w_2); w_3)$

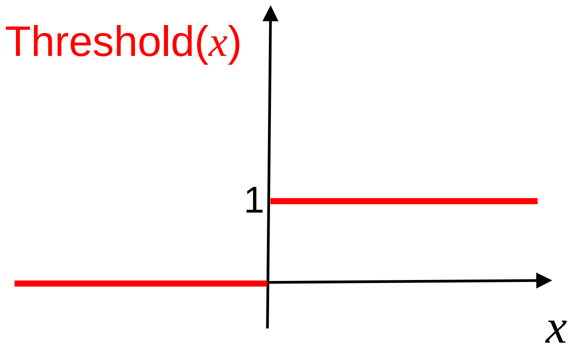
backward pass $(f_3(f_2(f_1(x; w_1); w_2); w_3))' =$

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

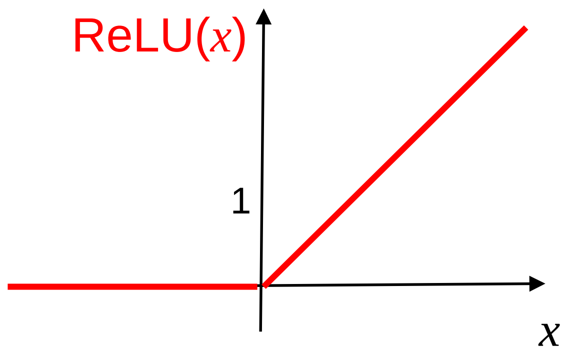
→ repeated multiplication of values in (0,1) → 0

non-sigmoid activation functions

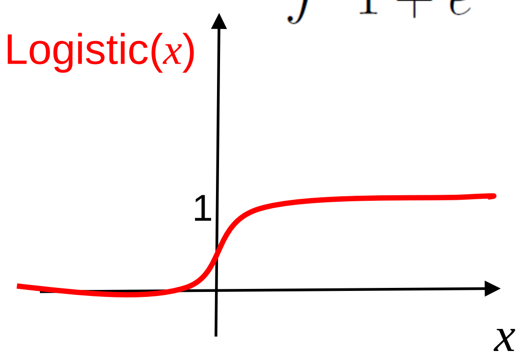
$$\int \mathbb{1}_{[x \geq 0]}(x)\, dx = \left\{ \begin{array}{ll} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{array} \right\} = \max\{0, x\} = \text{ReLU}(x)$$



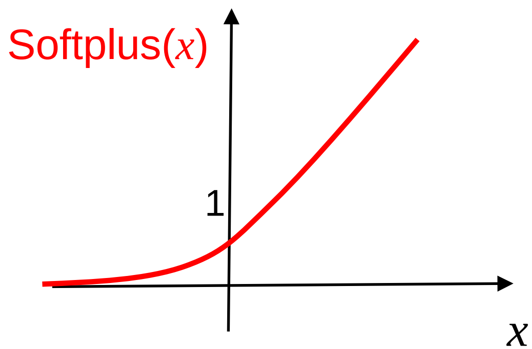
$\Rightarrow$



$$\int \frac{e^x}{1+e^x}\, dx = \log(1+e^x) = \text{softplus}(x)$$



$\Rightarrow$



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 sophistically

data augmentation

→ extending training set by slightly perturbed true training examples

- best applicable if inputs are **images**: translate, rotate, noise, …
- if x is **real vector** then adding e.g. small gaussian noise
  → here, utility disputable (actually needs sample from unseen subsets)

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

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

often $b \approx 100$ (empirically)

cost functions

- *regression*

  N training samples (x_i, y_i)

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

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

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

  $\Rightarrow (X \theta - y)^T (X \theta - y) \to \min\limits_{\theta}!$

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

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

  $\Rightarrow \sum\limits_i (f(x_i; \theta) - y_i)^2 \to \min\limits_{\theta}!$

cost functions

- *classification*

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

  → want to estimate probability of different outcomes

  → decision rule: choose class with highest probability

  idea: use maximum likelihood estimator (MLE)

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

  likelihood function

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

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

→ we use relative frequencies of training set x_1, …, x_N as estimator of q_i

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

⇒ the neural network should output $\hat{p}$ as close as possible to $\hat{q}$!

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

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

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

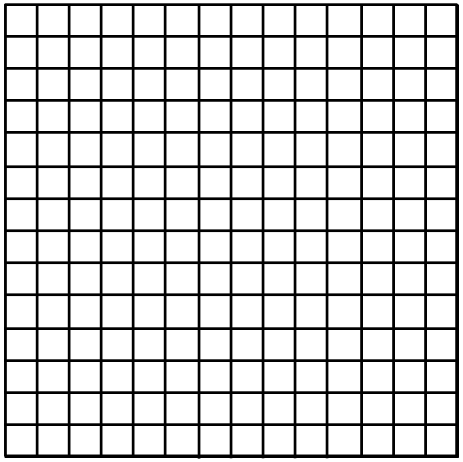
in case of *classification*

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

I(x,y)



K(i,j)

-1	-2	-1
1	1	1
-2	1	-2

example

1. Convolution

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

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

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

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

3. pooling

in principle: summarizing statistic of nearby outputs

e.g. **max-pooling** $m(i,j) = \max(\ z(i+a, j+b) : a,b = -d, \ldots, 0, \ldots d\ )$ for $d > 0$

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

- can be used to reduce matrix / output dimensions

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: