Ullrich Köthe

Machine Learning for Natural and Life Sciences

Machine learning for Natural- and Life-Science

- Goal: Teach the background of ML, not just recipes.
  ⇒ help you understand ML literature
- Setting: – we are interested in properties or quantities Y, but these are ~~hard to~~ impossible or impractical to measure directly ≙ "response"
  – properties / quantities X are related to Y and are easier to measure / observe ≙ "features"

  ⇒ approximate Y^* (true values, "ground truth") by a formula $\hat{Y} = f(X) \approx Y^*$

  $\hat{Y}$: approximate responses of our model, "predictions"
  – problem: the formula $f(X)$ is not known
  ⇒ collect training data and fit the model to $f(X)$ reproduce the behavior of that data
  – supervised learning: we cannot measure Y under normal field conditions, but we can obtain it in an experimental setting
  • example: crash test to determine car safety

$$TS = \{(X_i, Y_i)\}_{i=1}^{N}$$ N training instances

i: instance index

X_i, Y_i: features and response of instance i

X ≙ feature matrix is simply a table

Instance	Instance index	Height	Weight	Gender	
Alice	$i=1$	1.60	80	f	
Bob	$i=2$	1.85	85	m	...
Carol	$i=3$	1.70	70	f	
⋮			$x_{3,2}$		$\hat{=} X$
Feature index →		$j=1$	$j=2$	$j=3$	

$j \in 1, \dots, D$ D-dimensional feature space

Problem: for simplicity of the math and software, we want X to be real-valued, but many features are discrete / categorial, e.g. gender

⇒ "one-hot encoding": one flag per label

modified table/matrix

	gender	
	male	female
Alice	0	1
Bob	1	0
Carol	0	1

(... between the name and gender columns)

⇒ $X \in \mathbb{R}^{N \times \Delta}$, likewise Y

[if only two labels, we need only ~~one~~ encode one, because the other is implied, e.g. male = 1 − female]

- simple solutions to the problem
  - threshold classifier.
    - e.g. ~~bla~~ $Y = \begin{cases} 0 & \text{patient is healthy} \\ 1 & \text{patient suffers from diabetes} \end{cases}$

cannot be measured directly ⇒ measure a bio-marker

~~X: sugar content in blood~~

X: blood sugar concentration

classifier: $Y = \begin{cases} 0 & \text{if } X < 100\ mg/dl \\ 1 & \text{if } X > \end{cases}$

(simplified for illustration purposes)

- problem: often, there is no single feature that indicates the response
  ⇒ combine several features in some formula returning a "score"

e.g. $Y = \begin{cases} 0 & \text{person is normal} \\ 1 & \text{person is ~~bla~~ overweight} \end{cases}$

$$\text{score} = \frac{\text{weight [kg]}}{\text{height}^2\ [m^2]}$$ "body mass index"

$$Y = \begin{cases} 0 & \text{if } BMI < 25 \\ 1 & BMI > 25 \end{cases}$$

- problem: hand-crafted formulas have disadvantages
  - hard to find, expensive (may require decades)
  - in complex situations, simplifications are necessary
    ⇒ loss of accuracy

⇒ overcome these difficulties using <u>machine learning</u>

(downside: machine learning is often less explainable than an explicit formula ⇒ explainable ML is hot research area)

- nearest-neighbor classifier
  - classify a new instance x_{test} like the most similar instance in the training set
  - define similarity by a distance function of the features

$$dist(x, x') = \begin{cases} \text{small} & \text{if } x \text{ and } x' \text{ are similar} \\ \text{large} & \text{— " — different} \end{cases}$$

e.g. Euclidean distance between features

$$dist(x, x') = \sqrt{\sum_{j=1}^{D} (x_j - x'_j)^2}$$

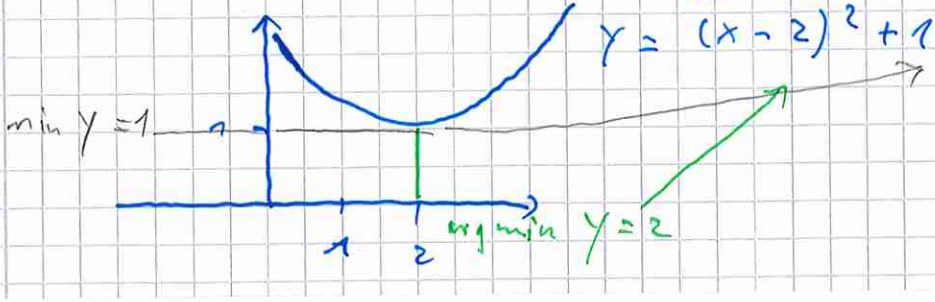
↑ sum over features = index j

- arg min - Notation:

$$i_{nearest} = \arg\min_{i=1}^{N} dist(x_{test}, x_i)$$

$\min_{i=1}^{N} dist(x_{test}, x_i)$: - iterate over all training inst.
- compute distance
- return the minimal distance

$\arg\min_{i=1}^{N}$ - iterate over all training instances
- compute distance
- return <u>index</u> of instance with minimal distance



$\Rightarrow \hat{y}_{test} = y^*_{nearest} = y_{i_{nearest}}$ classifier response

- often works pretty well, but has a number of shortcomings
  - need to memorize TS and search with $O(N)$ in each query $\Rightarrow$ scales poorly to large TS
    - ~~steal~~ memorize only important training instances $\Rightarrow$ hard to identify
    - speed-up by approximate nearest-neighbor search
  - $\hat{y}$ is not guaranteed to converge toward y^* even if $N \to \infty$ (error may be twice as big as the best achievable)
    - k-nearest neighbor $\Rightarrow$ more expensive
  - requires hand-crafted distance function
    $\Rightarrow$ same problems as hand-crafted stores
    $\Rightarrow$ metric learning (Bjön Ommer)
- human similarity judgements are hard to model
- show Voronoi diagram (animation)

Loss functions

Classical models: – define a specific formula for every situation
(e.g. physics: Newtons laws, law of gravity, Maxwells laws, etc.)
– theoreticians hand-craft these models

machine learning – use generic model families that apply to many situations and have free parameters Θ: $Y = f_\Theta(X)$

e.g. Linear: $Y = aX + b$ ↳ $\Theta = (a, b)$

– among all possible choice for Θ, pick $\hat{\Theta}$ such that $Y_i \approx f_{\hat{\Theta}}(X_i)$ works as well as possible for the TS $\{(X_i, Y_i)\}_{i=1}^{N}$
"model fitting / training"

⇒ need a way to quantify the "goodness of fit", i.e. to compare the performance of different choices of Θ

Let $\hat{Y} = f_\Theta(X)$ the prediction for a data set X, given Θ
$Y^* = f^*(X)$ the ~~function~~ true relationship
(we know Y^*, but not f^*)

$\text{loss}(Y^*, \hat{Y}) \to \mathbb{R}^+$ measures the difference between truth and predictions

$$\begin{cases} \approx 0 & \text{if } Y^* \approx \hat{Y} \\ \text{big} & \text{otherwise ("loss")} \end{cases}$$

if the data instances are iid (independent of each other, but generated from the same true model f^*)
this simplifies to ~~a~~ additive loss over all instances

$$\text{loss}(Y^*, \hat{Y}) = \frac{1}{N} \sum_{i=1}^{N} \text{loss}(Y_i^*, \hat{Y}_i)$$

The particular formula for the Loss(..) determines what is considered a good fit and must be selected by the model designer / data scientist according to the application.

- most common choice: squared loss

$$\text{loss}(y^*, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i^* - \hat{y}_i)^2$$

$$= 0 \iff y_i^* = \hat{y}_i \text{ for all } i$$

e.g. for 2-class classification

$$y = \begin{cases} 0 & \text{if healthy} \\ 1 & \text{if diabetes} \end{cases}$$

$$(y_i^* - \hat{y}_i)^2 = \begin{cases} 0 & \iff y_i^* = \hat{y}_i \text{ correct prediction} \\ 1 & \iff y_i^* \neq \hat{y}_i \text{ wrong prediction} \end{cases}$$

$\hat{=}$ count the wrongly predicted instances $\hat{=}$ 0/1 error

- $\hat{y}_i = f_\Theta(x_i)$ depends on the choice of Θ, and so will the loss $\Rightarrow$ choose Θ to minimize the loss ~~of~~ on the TS

$$\hat{\Theta} = \arg\min_{\Theta} \frac{1}{N} \sum_{i=1}^{N} \text{loss}(y_i^*, \hat{y}_i = f_\Theta(x_i))$$

Finding the optimal $\hat{\Theta}$ is in general a complicated optimization problem $\Rightarrow$ later.

The loss after optimizing $\hat{\Theta}$ is the "training error"

$$\hat{err} = \frac{1}{N} \sum_i \text{loss}(y_i^*, \hat{y}_i = f_{\hat{\Theta}}(x_i))$$

$$= \min_{\Theta} \frac{1}{N} \sum_i \text{loss}(y_i^*, y_i = f_\Theta(x_i)) \Big] =$$

because we chose $\hat{\Theta}$ such that it achieves the minimum

- what we are actually interested in is the performance of our model in the field, i.e. on arbitrary data beyond the TS

$$Err = \mathbb{E}_{x,y^* \sim p^*(x,y)} \left[\text{loss}(y^*, \hat{y} = f_{\hat{\Theta}}(x)) \right]$$

"generalization ~~error~~ error"

- fundamental insight of ML and statistics:

$$Err > \widehat{err}$$

generalization error is usually bigger than training error, sometimes much bigger $\hat{=}$ "overfitting"

$\Rightarrow$ take home message: fundamental insight 1

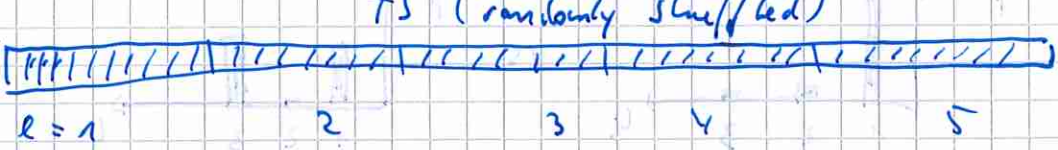
> IT MAKES NO SENSE TO REPORT MODEL PERFORMANCE IN TERMS OF THE TRAINING ERROR $\widehat{err}$

- Instead we must estimate and report the generalization error. In practice, empirical estimation works best. 2 possibilities:
  - independent test set $Test = \{(x_i^{test}, y_i^{test})\}_{i=1}^{N'}$

    Compute the average loss of the test set

$$\widehat{Err} = \frac{1}{N'} \sum_{i=1}^{N'} Loss\left(y_i^{test}, \hat{f}(x_i^{test})\right)$$

    Note: - Never use the test data for training.
    - $\hat{\vartheta}$ remains fixed now.
  - cross-validation: if no independent test set is available, create it from the training set:



    Create K random subsets ("folds"), here $K = 5$

    $\Rightarrow$ "K-fold cross-validation"

    Let TS_ℓ denote fold ℓ, $TS_{\setminus \ell}$ the remaining data for $\ell = 1 \dots K$
    - train on $TS_{\setminus \ell}$ to get $\hat{\vartheta}_\ell$
    - estimate $\widehat{Err}_\ell$ on TS_ℓ, using $\hat{\vartheta}_\ell$ fixed

    estimate $\widehat{Err} = \frac{1}{K} \sum_\ell \widehat{Err}_\ell$

    typical values for K: 10, 5 (2 if training is very expensive); $K = N$ LOO-CV for theory

- Infrastructure for CV is available in all credible ML libraries (sk-learn, pytorch, ...)

⇒ don't implement it yourself (except as an exercise)

Bayes decision rule

- instead of a hard class decision, we can return a soft response: the probability of each label, given the features

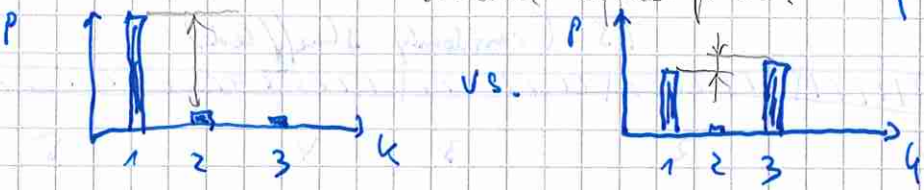
$$\forall k: \quad p(Y_i = k \mid X_i) \qquad \text{with} \sum_{k=1}^{C} p(Y_i = k \mid X_i) = 1$$

- a hard decision is easily obtained by deciding for the most probable class: $\hat{Y}_i = \arg\max_k p(Y_i = k \mid X_i)$

- many methods learn to approximate $\hat{p}(Y \mid X)$ (the "posterior probability" ≙ after measuring the features) as well as possible ⇒ more information than a hard decision rule, e.g. uncertainty estimate via margin:

$$p(\hat{Y}_i \mid X_i) - \arg\max_{k \neq \hat{Y}_i} p(Y_i = k \mid X_i) = \begin{cases} \text{large} & \text{if no doubt} \\ \approx 0 & \text{if uncertain} \end{cases}$$

(second highest prob.)



e.g. Brexit

- the true posterior $p^*(Y_i \mid X_i)$ is called the "Bayesian ~~decision~~ posterior". The Bayes decision rule

$$\hat{Y}_i = \arg\max_k p^*(Y_i = k \mid X_i)$$

is the theoretically best possible classifier

⇒ learning attempts to ensure $\hat{p}(Y_i \mid X_i) \approx p^*(Y_i \mid X_i)$

- the ~~limit~~ Bayesian posterior and thus the limit is generally unknown in practice, but plays an important role as an ~~upper~~ lower bound for the errors in theoretical analysis

Explain confusion matrix, difference between false positives and false negatives ⇒ see later

Linear Models

- simplest model family, applicable to
  - regression: $Y_i \in \mathbb{R}$
  - classification ~~$Y_i \in \{1, \dots, C\}$~~ $Y_i \in \{0, 1\}$

    (extensions to more classes possible by coupling several linear models in one-against-the-rest fashion)
- compute a weighted sum of the feature values plus offset
  - matrix notation for regression: $\hat{Y}_i = X_i \cdot \beta + \delta$

    (β: column vector of D weights; δ: offset)

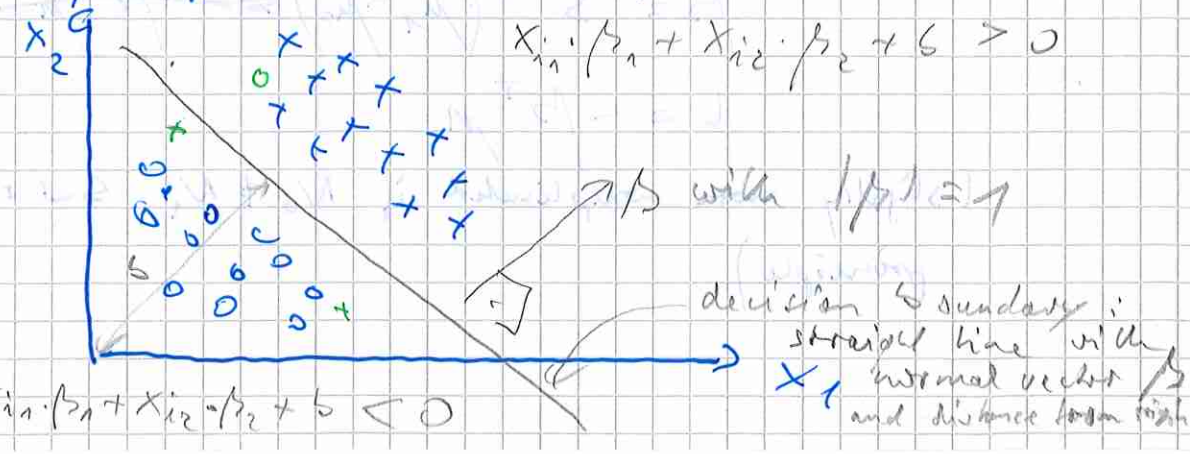
    element notation: $\hat{Y}_i = \sum_{j=1}^{D} X_{ij} \cdot \beta_j + \delta$

    β_j: weight of feature j

    $|\beta_j|$ { big: feature j is important; ≈ 0: — " — unimportant
  - matrix notation for classification: $\hat{Y}_i = \mathbb{1}[X_i \cdot \beta + \delta > 0]$

    $\mathbb{1}[\text{cond}] = \begin{cases} 1 & \text{if "cond" is true} \\ 0 & \text{if "cond" is false} \end{cases}$ } simply converts true/false to numbers 0/1

    $\Rightarrow X_i \cdot \beta + \delta$ should be positive if $Y_i^* = 1$, negative if $Y_i^* = 0$
  - the parameters are $\Theta = \{\beta, \delta\}$ ($D+1$ numbers)
- linear classification works well if data are linearly separable, or nearly so



- many ways to find good decision planes, e.g. optimal parameters $\hat{\beta}, \hat{b}$, controlled by the choice of loss function.
  - If the data are suitable for linear classification, all solutions will be similar.
  - squared loss: $\Rightarrow$ LDA (linear discriminant analysis)

define $\tilde{y}_i = \begin{cases} 1 & \text{if } y_i = 1 \\ -1 & \text{if } y_i = 0 \end{cases} = 2y_i - 1$

optimize $\hat{\beta}, \hat{b} = \underset{\beta, b}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} (\tilde{y}_i - x_i\beta - b)^2$

reduces learning to ordinary least squares and ~~looks complicated, but~~ has a simple solution (assuming balanced classes, i.e. $N_0 = N_1 = \frac{N}{2}$)

1) ~~centre~~ compute the class means

$$\mu_1 = \frac{1}{N_1} \sum_{i: y_i = 1} x_i \qquad \mu_0 = \frac{1}{N_0} \sum_{i: y_i = 0} x_i$$

2) and total mean

$$\mu = \frac{1}{N} \sum_i x_i = \frac{\mu_0 + \mu_1}{2} \quad (N_0 = N_1)$$

2) compute within-class covariance matrix

$$S = \frac{1}{N} \sum_i (x_i - \mu_i)(x_i - \mu_i)^T$$

↑ outer product

with $\mu_i = \begin{cases} \mu_1 & \text{if } y_i = 1 \\ \mu_0 & \text{if } y_i = 0 \end{cases}$ cluster mean

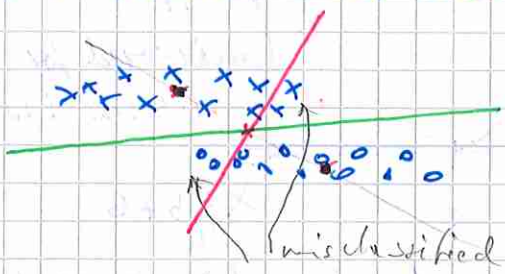
3) compute parameters

$$\beta = S^{-1}(\mu_1 - \mu_0) \quad \sout{b = \beta^T \mu}$$

$$b = -\beta^T \mu$$

(slightly more complicated if $N_0 \neq N_1$, but same principle)

- comparison between LDA and a naive rule: choose direction between class means



normal vector of decision boundary

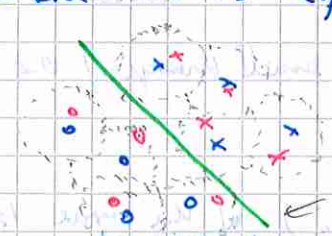
$\beta = S^{-1}(\mu_1 - \mu_0)$

↑ rotates decision boundary according to cluster shape (no rotation if clusters are round)

- strong assumptions: – cluster shapes are close to Gaussian bell (elliptic)
  – both classes have same cluster shape
  – somewhat robust if not fulfilled exactly

Linear support vector machines (SVM)

- insight: features are not exact, but noisy ⇒ slightly perturbed TS is equally plausible



⇒ decision should be robust against such perturbations

⇒ we should maximize the "safety margin" between data points and decision boundary

$$\hat{\beta}, \hat{b} = \underset{\beta, b}{\arg\min} \; \frac{\beta^T \beta}{2} + \frac{\lambda}{N} \sum_i \text{Hinge Loss}(y_i^*, x_i \beta + b)$$

$\frac{\beta^T\beta}{2}$: maximizes margin for possible TS perturbations ("regularization term")

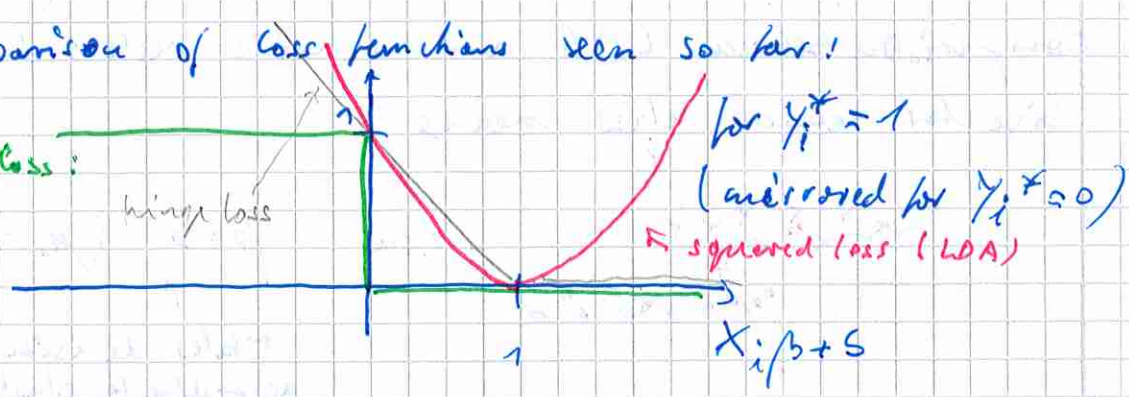
$\sum_i \text{Hinge Loss}(y_i^*, x_i\beta + b)$: measures quality of prediction on the TS ("data term")

λ: regularization parameter: adjusts trade-off between training error and robustness (hyper parameter, must be chosen by user)

$$\text{Hinge Loss}(y_i^*, x_i\beta + b) = \begin{cases} \max(0, 1 - (x_i\beta + b)) & \text{if } y_i^* = 1 \\ \max(0, 1 + (x_i\beta + b)) & \text{if } y_i^* = 0 \end{cases}$$

no analytic solution, but efficient iterative algorithms

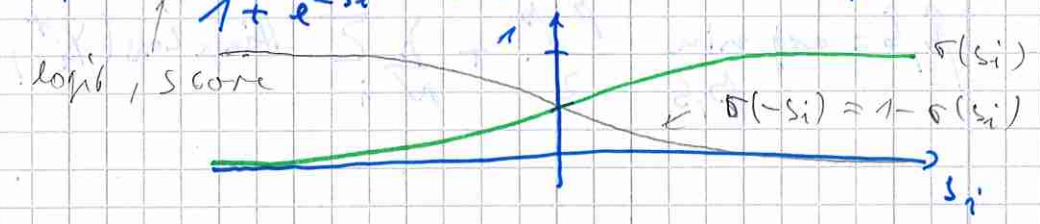
- comparison of loss functions seen so far:



- 0/1 loss: no penalty if correct sign (i.e. correct decision)
  constant penalty if wrong sign
- squared loss: quadratic penalty if $x_i\beta + b \neq 1$
  (i.e. over confident responses are also penalized)
- hinge loss: no penalty for sufficiently confident correct answers: $x_i\beta + b \geq 1$
  linearly increasing penalty for incorrect answers: $x_i\beta + b < 0$
  small penalty for correct answers with low confidence (i.e. small margin) $0 < x_i\beta + b < 1$

Logistic Regression (LR)

- learn an approximation $\hat{p}(y_i | x_i)$ of the true Bayesian posterior $p^*(y_i | x_i)$
- combine linear model $s_i = x_i\beta + b \in \mathbb{R}$ with subsequent non-linearity $\sigma(s_i) = \frac{1}{1 + e^{-s_i}} \in [0, 1]$ ⇒ "logistic sigmoid fct."



$$\hat{p}(y_i = 1 | x_i) = \sigma(x_i\beta + b)$$

$$\hat{p}(y_i = 0 | x_i) = 1 - \hat{p}(y_i = 1 | x_i) = 1 - \sigma(x_i\beta + b) = \sigma(-(x_i\beta + b))$$

- optimal decision:

$$\hat{y}_i = \begin{cases} 1 & \text{if } \hat{p}(y_i = 1 | x_i) > \hat{p}(y_i = 0 | x_i) \Leftrightarrow x_i\beta + b > 0 \\ 0 & < \end{cases}$$

⇒ posterior is non-linear, but decision rule is linear

- optimization problem: define $\hat{p}$, i.e. $\hat{\beta}$ and $\hat{b}$ such that the combined posterior of the training labels is maximized
  ≙ maximum likelihood principle

$$\hat{\beta}, \hat{b} = \underset{\beta, b}{\arg\max} \prod_{i: y_i = 1} \hat{p}(y_i = 1 \mid x_i) \cdot \prod_{i: y_i = 0} \hat{p}(y_i = 0 \mid x_i)$$

(due to i.i.d.)

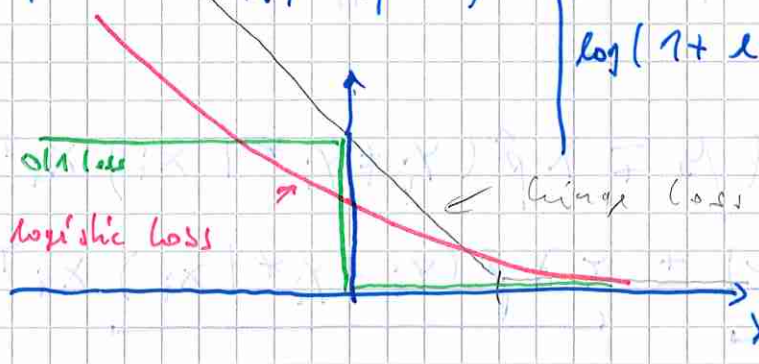
$$= \underset{\beta, b}{\arg\max} \prod_{i: y_i = 1} \sigma(x_i \beta + b) \cdot \prod_{i: y_i = 0} \sigma(-(x_i \beta + b))$$

- after some manipulations and introduction of regularization (i.e. maximization of safety margin), this results in

$$\hat{\beta}, \hat{b} = \underset{\beta, b}{\arg\min} \frac{\beta^T \beta}{2} + \frac{1}{\alpha} \sum_i \text{logistic Loss}(y_i, x_i \beta + b)$$

solve iteratively, similar to SVM

$$\text{logistic Loss}(y_i, x_i \beta + b) = \begin{cases} \log(1 + e^{-(x_i \beta + b)}) & \text{if } y_i = 1 \\ \log(1 + e^{x_i \beta + b}) & \text{if } y_i = 0 \end{cases}$$



for $y_i = 1$
(mirrored for $y_i = 0$)

⇒ logistic loss is a smooth version of the hinge loss, similar behavior

- for large dataset ($N > 10^4 \ldots 10^5$), stochastic gradient descent (SGD), or one of its improved variants, is the fastest optimization algorithm
  (for small datasets, $N \leq 10^3 \ldots 10^4$, Newton or quasi-Newton algorithms are faster)
  - SGD selects a single random instance or a mini-batch to estimate the loss in every iteration
    $i \sim \text{uniform}(1, \ldots N)$

loss in current iteration:

$$\text{loss}_i = \frac{\beta^T\beta}{2} + \lambda \log\left(1 + e^{\mp(x_i\beta + s)}\right)$$

with $-$ if $y_i = 1$ and $+$ if $y_i = 0$

$$\frac{\partial\, \text{loss}_i}{\partial \beta} = \frac{\beta}{} + \frac{\lambda}{1 + e^{\mp(x_i\beta+s)}} \cdot e^{\mp(x_i\beta+s)} \cdot \left(\mp x_i^T\right)$$

$$= \beta \mp \lambda \cdot \underbrace{\sigma\left(\mp(x_i\beta + 1)\right)}_{= 1 - \hat{p}(y_i = y_i^* \mid x_i) = \hat{p}(y_i \neq y_i^* \mid x_i)} x_i^T$$

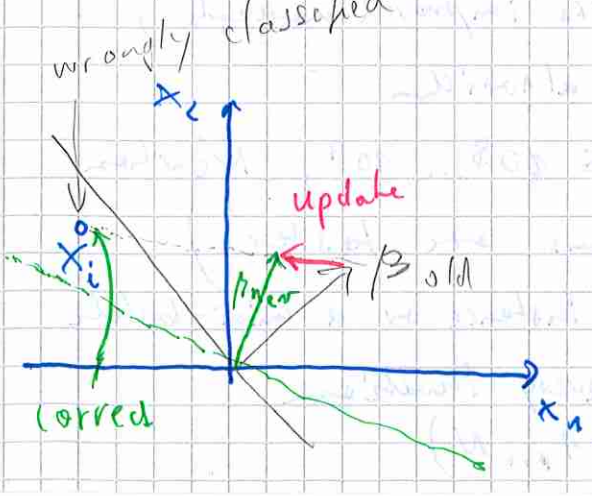
$$= \beta \mp \lambda \cdot \hat{p}(y_i \neq y_i^* \mid x_i)\, x_i^T$$

Since we want to minimize the loss, we perform gradient descent, i.e. the update goes in the opposite direction with step size τ:

$$\beta \leftarrow \beta - \tau\left(\beta \mp \lambda \hat{p}(y_i \neq y_i^* \mid x_i)\, x_i^T\right)$$

$$\beta \leftarrow \underbrace{\beta(1-\tau)}_{\text{regularization: reduce magnitude of } \beta} \pm \underbrace{\tau\lambda \hat{p}(y_i \neq y_i^* \mid x_i)\, x_i^T}_{\text{update weight}}$$

- if classification is already correct: $\hat{p}(y_i \neq y_i^* \mid x_i) \approx 0$ $\Rightarrow$ almost no update
- if classification is wrong: $\hat{p}(y_i \neq y_i^* \mid x_i) \approx 1$ $\Rightarrow$ move β towards x_i (if $y_i^* = 1$) or away from x_i (if $y_i^* = 0$)



unbalanced loss functions / classes

- in two-class problems, there are two kinds of errors:
  - false positive $\hat{y}_i = 1$, $y_i^* = 0$
  - false negative $\hat{y}_i = 0$, $y_i^* = 1$
- often, consequences of these errors are very different, e.g. access control to a bank safe
  - false negative: authorized person is rejected:
    ⇒ annoying, but simply retry (cf. fingerprint sensor on cell phones)
  - false positive: burgler is admitted
    ⇒ very expensive

⇒ weight each loss term with its cost

$$\hat{\beta}, \hat{s} = \underset{\beta, s}{\operatorname{argmin}} \frac{1}{N} \sum_i w_i \, \text{loss}(y_i^*, x_i \beta + s)$$

for access control: $w_i = \begin{cases} \text{big} & \text{if } y_i^* = 0 \text{ (false positive)} \\ \text{small} & \text{if } y_i^* = 1 \end{cases}$

choose weights according to application

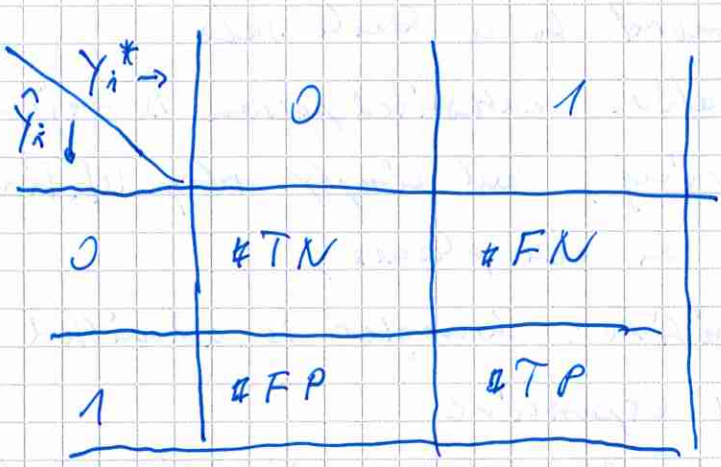
- unbalanced classes: one class (e.g. positive ≙ disease) is much more rare than the other (e.g. healthy)
  ⇒ errors may be misleading
  - consider test for a disease with 1% false positives and 1% false negatives – sounds pretty good
  - positive to negative ratio is 1/100
  - simple calculation shows that if

$$\hat{y}_i = 1 \text{ (suspected disease)} \;\Rightarrow\; p(y_i^* = 1 \mid x_i) = 0.5 \text{ (actual disease)}$$

⇒ many patients are unnecessarily scared
⇒ report precision and recall instead of error rates

more?

- even worse: if only true positive rate (e.g. 99%) / false negative rate (e.g. 1%) is reported, but not the false positive rate, the test quality cannot be judged at all! (cf. ~~blood~~ breast cancer blood test scandal)
- confusion matrix for $C=2$ ($C>2$ analogously)



should be diagonal

true positive rate $\hat{=}$ sensitivity: $\hat{=}$ recall

$$\frac{\#TP}{\#P} = \frac{\#TP}{\#TP + \#FN}$$

true negative rate $\hat{=}$ specificity:

$$\frac{\#TN}{\#N} = \frac{\#TN}{\#TN + \#FP}$$

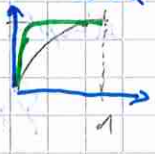
precision:

$$\frac{\#TP}{\#TP + \#FP}$$

- If we have a score or posterior, we can freely choose the threshold, e.g. $\hat{p}(y_i = 1 \mid x_i) > 0.5$ (balanced)
  $\hat{p}(y_i = 1 \mid x_i) > 0.9$ (avoid false positives)

  ⇒ precision, recall, specificity change according to threshold

  ⇒ ROC curve and precision/recall curve depict this dependency ⇒ judge quality of results by "area under curve" (AUC)



fundamental insight 2:

DEFINING APPROPRIATE LOSS FCT. FOR AN APPLICATION IS A MAJOR PART OF A MACHINE LEARNING DESIGN.

⇒ easier than hand-crafted model, because only a critic, not a constructive solution

Regression

- classification: Y is discrete (e.g. C classes)
  regression: Y is a real number
- closely related: $s_i = x_i \beta + \varsigma$ is a real-valued score
  $\hat{y}_i = \operatorname{sign}(x_i \beta + \varsigma)$ a discrete class
  ⇒ regression is often an intermediate part of classification and often a goal of its own

(example:)

- linear regression is most basic approach
  ⇒ ordinary least squares (OLS) method

$$\hat{\beta}, \hat{\varsigma} = \arg\min_{\beta, \varsigma} \frac{1}{N} \sum_{i=1}^{N} \left(y_i^* - \underbrace{(x_i \beta + \varsigma)}_{= \hat{y}_i} \right)^2$$

y_i^*: arbitrary real number

~~$= \arg\min_{\beta, \varsigma = N}$~~

- we can eliminate ς by centralizing the data

$$\tilde{x}_i = x_i - \mu \qquad \mu = \frac{1}{N} \sum_i x_i \quad \text{mean}$$

($\tilde{x}_i$ has now zero mean)

$$\hat{\beta} = \arg\min_{\beta} \frac{1}{N} (y^* - X\beta)^T (y^* - X\beta)$$

(matrix notation)

- find the optimum by setting the derivative to zero

~~$\frac{\partial}{\partial \beta} \frac{1}{N} (y^* - X\beta)^T (y^* - X\beta) = \frac{2}{N} (-y^* X + X^T X \beta) \doteq 0$~~

$$\frac{\partial}{\partial \beta} \frac{1}{N} (X^T y^* - y\beta)^T (y^* - X\beta) = \frac{2}{N} (-X^T y^* + X^T X \beta)$$

$$\doteq 0$$

$$X^T X \beta = X^T y^*$$

$$\hat{\beta} = \underbrace{(X^T X)^{-1} X^T}_{=: X^+} y^* \quad \text{“pseudo-inverse”}$$

Moore-Penrose inverse
generalization of inverse for rectangular matrices

- in order to give all features equal influence in the loss, it makes sense to standardize features ⇒ unit-free quantities

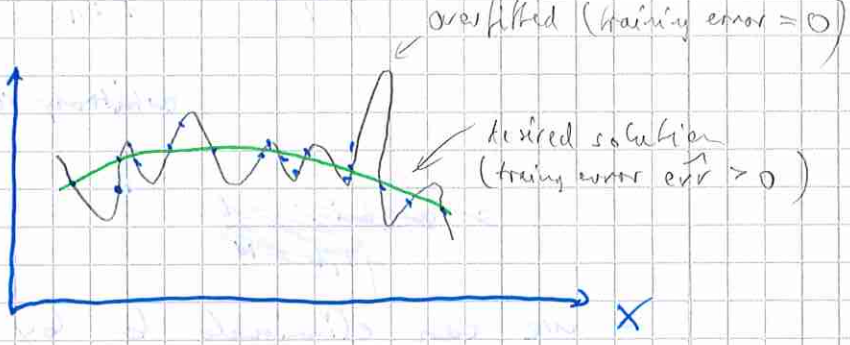
$$\tilde{x}_i = \frac{x_i - \mu}{\sigma} \leftarrow \text{element-wise division}$$

σ: vector of standard deviations (σ_j: stddev of feature j)
(this is also helpful in classification, standard data preprocessing)

- numeric solution of OLS: re-usable algorithms in all ML libraries, using QR decomposition or singular value decomposition (SVD)

Overfitting

- non-linear example



- effective counter-measure: regularization of parameters, e.g. penalize large values of β-coefficients
most common: – L2 regularization: restrict Euclidean (squared) norm

$$\|\beta\|_2^2 = \sum_j \beta_j^2 = \beta^T \beta$$

⇒ ridge regression

$$\hat{\beta} = \arg\min_\beta \frac{\beta^T \beta}{2} + \frac{\lambda}{n} \sum_i (y_i^* - x_i \beta)^2$$

identical to support vector machine, but with squared loss ⇒ robustness against noise

– L1 regularization: restrict L1 norm

$$\|\beta\|_1 = \sum_j |\beta_j|$$

⇒ LASSO regression

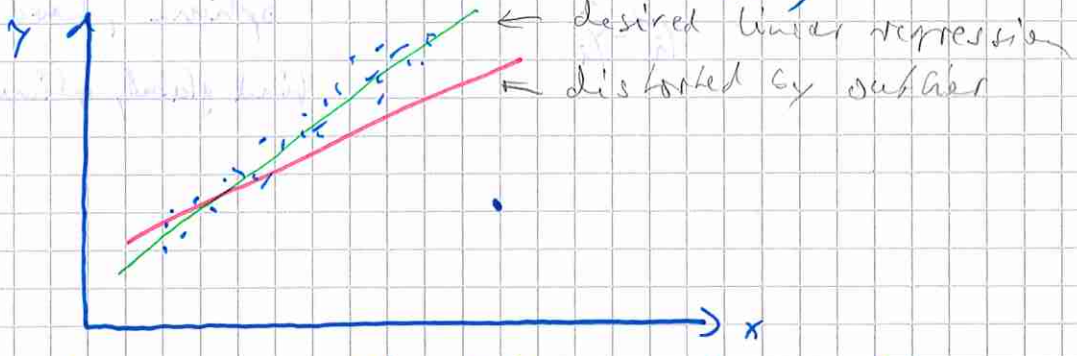
$$\hat{\beta} = \arg\min_{\beta} \|\beta\|_1 + \frac{\lambda}{n} \sum_i (y_i^* - X_i \beta)^2$$

⇒ sparsity enforcing: many coefficients $\beta_j = 0$
non-zero coefficients denote important features
⇒ feature selection

- regularization parameter λ must be manually chosen for each application as a hyper-parameter to optimize the bias-variance trade-off

robust regression

- results can be severely distorted when training data is contaminated by outliers ≙ grossly wrong data points (e.g. satellite data: transmission errors as opposed to measurement noise)

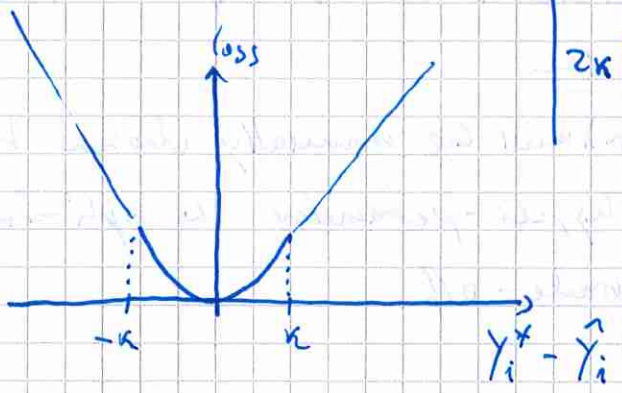


- solution 1: identify inliers and ignore outliers
⇒ RANSAC algorithm
- solution 2: modify the loss function to become robust against outliers

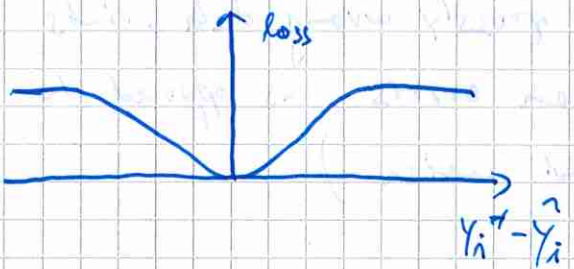
- squared loss: outliers are highly weighted, because error term is squared
- absolute loss: $\text{Loss}(y_i^*, \hat{y}_i) = |y_i^* - \hat{y}_i|$
  - much more robust, influence of outliers greatly reduced
  - disadvantages: • often no unique solution for $\hat{\beta}$
    • higher generalization error on new data

- Huber loss: best of both worlds
  - squared loss for small errors ⇒ high accuracy, uniqueness
  - absolute loss for high errors ⇒ robustness

$$\text{Huber Loss}(y_i^*, \hat{y}_i) = \begin{cases} (y_i^* - \hat{y}_i)^2 & \text{if } |y_i^* - \hat{y}_i| \le \kappa \\ 2\kappa |y_i^* - \hat{y}_i| - \kappa^2 & \text{else} \end{cases}$$



- bi-weight functions: loss for high errors saturates, there is no distinction between "very wrong" and "extremely wrong"



advantage: even more robust!
disadvantage: has many poor local optima, hard to find globally optimal $\hat{\beta}$

- outlier handling $\hat{=}$ out-of-distribution detection $\approx$ novelty detection $\hat{=}$ robustness against adversarial attacks / samples

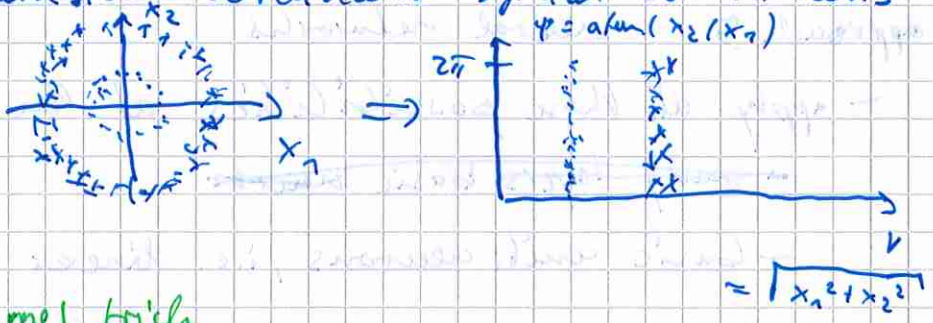
is still a hot research topic ⇒ show adversarial examples

Non-linear Methods

- most data are not adequately represented by linear models
- approach 1: manually design non-linear model and fit its parameters, e.g. non-linear least squares, Levenberg-Marquardt algorithm.
  problems: • has many poor local optima, good initial guess required to find best $\hat{\theta}$
  • manual model design is difficult, esp. for highly complex problems (e.g. medicine, biology, social sciences)
- approach 2: cleverly combine several linear models into a non-linear model
- approach 2: augmented feature spaces
  (1) calculate new features by non-linear functions of the given original features
  (2) apply a linear model in this augmented feature space

example: body mass index = ~~body fat~~ weight / height^2

example: Cartesian coordinates $\Rightarrow$ polar coordinates



Kernel trick

- access features via Gram matrix $G = XX^T$ instead of directly via X
- replace G with kernel matrix K using $x_i x_j^T \Rightarrow k(x_i, x_j)$ $\Rightarrow$ non-linear kernel function

variant: rewrite loss and regularization in terms of a non-linear kernel function $\Rightarrow$ augmented feature space needs not be constructed explicitly, can have high and even infinite dimension

advantage: - training (i.e. finding optimal $\hat{\theta}$) is easier than in approach 1, good optimization algorithms

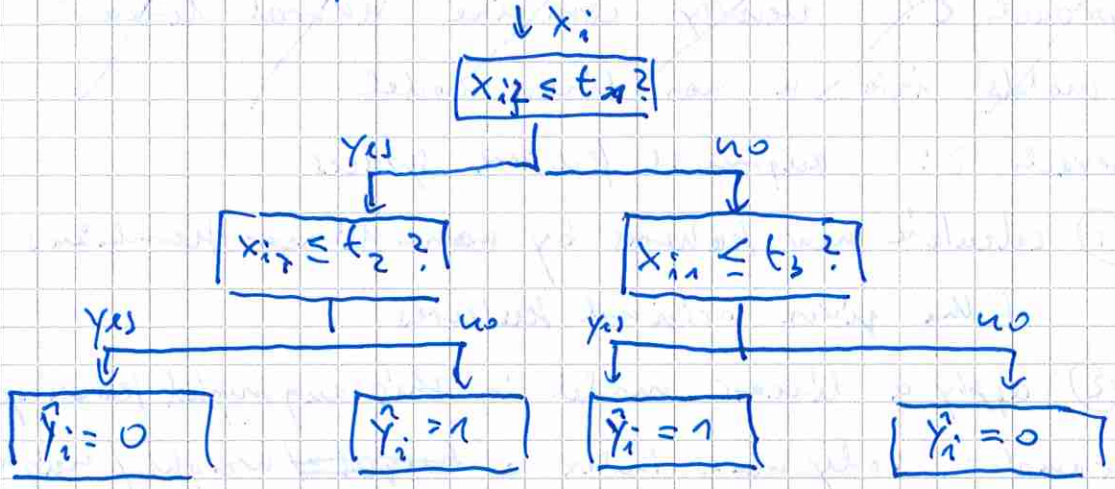
disadvantage: - manual design of good non-linear transformations or kernels is difficult

- approach 3: boosting
  - run several linear models in parallel, then take a weighted vote

$$\hat{y}_i = \text{sign}\left(\frac{1}{L}\sum_{\ell=1}^{L} w_\ell \cdot \text{sign}\left(x_i \beta_\ell + b_\ell\right)\right)$$

$$\Theta = \left\{(\beta_1, b_1, w_1), \dots, (\beta_L, b_L, w_L)\right\}$$

- approach 4: decision trees
  - run several linear models in series => split feature space into subregions, where each subregion behaves simpler than the whole.



- approach 5: neural networks
  - apply all those possibilities at the same time

~~many layers basic neurons~~

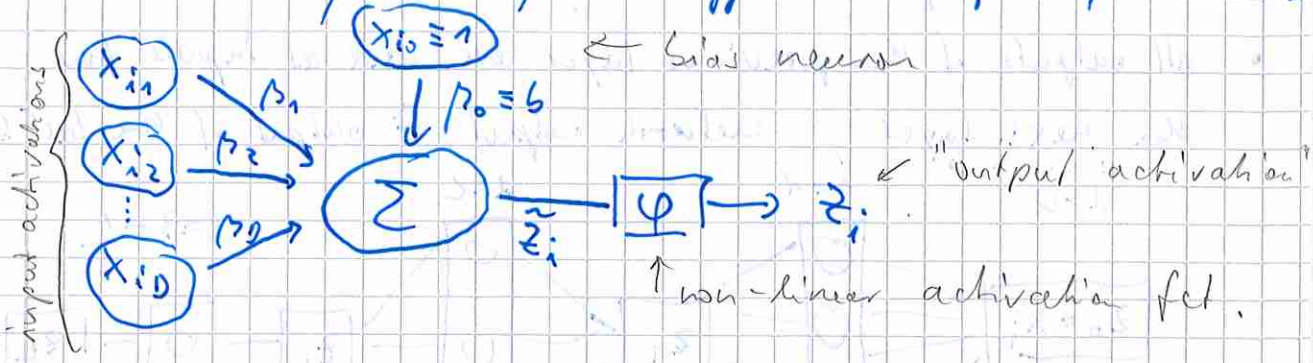
  - basic multi neurons, i.e. linear models, followed by simple non-linearity
  - sequential processing: arrange neurons in layers
  - parallel processing: each layer contains many independent neurons

why does this solve the problems?

- scales to very complex models ("deep" networks, "wide" networks)
- outputs of inferior layers are "learned and transformed feature spaces"
- most local minima are reasonably good => robust optimization
- hand-crafting good architecture and non-linearities is still hard, but performance is relative robust against sub-optimal solutions

Neural Networks

- neurons are simple linear models, followed by a 1-dimensional non-linearity — very crude approximation of biological neurons



bias neuron: add feature 0, which is constant = 1

$\Rightarrow$ absorb bias parameter b into β as β_0

$(\beta \in \mathbb{R}^{D+1}, \; j = 0 \dots D)$

output of the neuron:

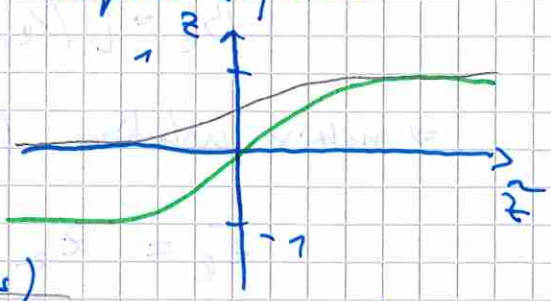
$$z_i = \varphi\left(\sum_{j=0}^{D} x_{ij} \cdot \beta_j \right) \qquad \text{row vector}$$

- activation fct. should be almost everywhere differentiable, so that we can train by gradient descent. ($\operatorname{sign}(\tilde{z})$ is not suitable)

- traditional choices:

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

logistic sigmoid

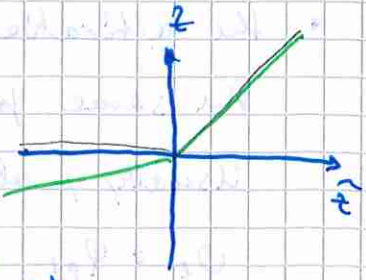


$$z = \varphi(\tilde{z}) = \tanh(\tilde{z})$$

~~$z = \tilde{z}$ (for linear layers, e.g. regression outputs)~~

- modern choices

$$z = \varphi(\tilde{z}) = \mathrm{ReLU}(\tilde{z}) = \max(\tilde{z}, 0)$$



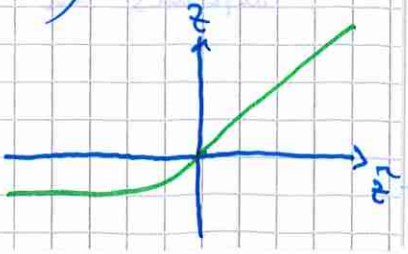
$$z = \mathrm{LeakyReLU}(\tilde{z}) = \begin{cases} \tilde{z} & \text{if } \tilde{z} \geq 0 \\ a \cdot \tilde{z} & \text{if } \tilde{z} < 0, \end{cases}$$

($a \ll 1$ fixed)

$z = \mathrm{PReLU}(\tilde{z})$: like LeakyReLU, but a is learned

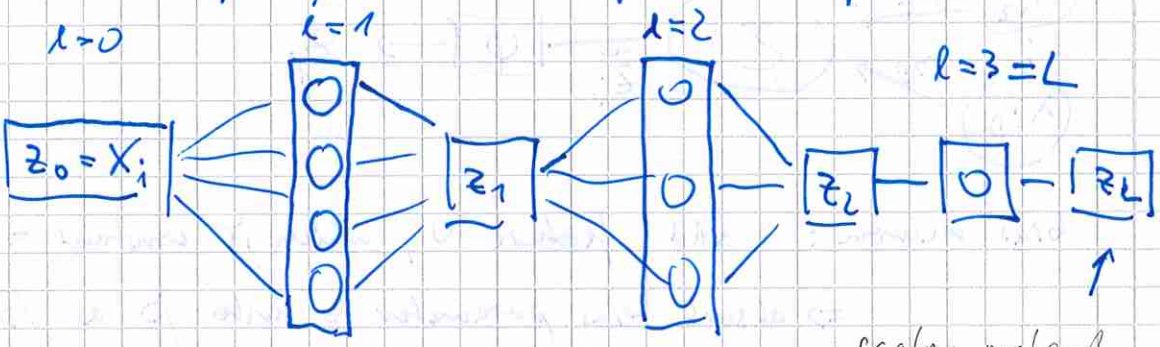
$$z = \mathrm{ELU}(\tilde{z}) = \begin{cases} \tilde{z} & \text{if } \tilde{z} > 0 \\ a(e^{\tilde{z}} - 1) & \text{otherwise} \end{cases}$$

(a fixed)



Fully connected networks

- consist of the input layer ($\hat{=}$ the original feature vector x_i) and L ~~compute~~ neuron layers
- all outputs of the previous layer are used as inputs for the next layer ; network output $\hat{=}$ output of the last layer



scalar output,
e.g. • regression in 1-D
• posterior probab. $\hat{p}(\hat{y}_i = 1 / x_i)$

- layer l contains H_l (M_l) neurons (+ bias neuron - not shown)
(here $H_1 = 4$ (M_1), $H_2 = 3$ (M_2), $H_3 = 1$ (M_3))
⇒ each layer needs H_l different weight vectors β:

$$\beta_{lm}, \quad l = 1 \dots L, \quad m = 1 \dots H_l \ (M_l)$$

Combine the weight vectors of each layer into matrix

$$B_l = [\beta_{l1} \dots, \beta_{lM_l}]$$

⇒ matrix notation for pre-activations of layer l:

$$\tilde{z}_l = z_{l-1} \cdot B_l$$

- the activation functions φ (non-linearities) are usually the same for all neurons in a given layer: φ_l.
Usually, all inner layers have the same activation $\varphi_l = \varphi_{l'}$, except for last layer φ_L, whose activation depends on the application

output of a 3-layer network for feature vector x_i:

$$\hat{y}_i = z_3 = \varphi_3\Big(\varphi_2\big(\varphi_1\big(\underbrace{z_0}_{=x_i}\cdot B_1\big)\cdot B_2\big)\cdot B_3\Big)$$

(non-linearities; parallel processing; serial processing)

[note: the bias channels $z_{\ell 0} \equiv 1$ is always implicitly present, even if not shown explicitly]

- by increasing L and M_ℓ, we can adapt model complexity to the application
- output activation functions depend on the task:
  - regression: y_i is real or real-valued vector

    $\varphi_L(\tilde{z}_L) = \tilde{z}_L$ linear activation

    $\Rightarrow$ last layer performs linear regression, using penultimate activations z_{L-1} as augmented feature space
  - classification with $C = 2$: estimate posterior for positive class

    $z_L = \hat{p}(y_i = 1 \mid x_i) = \sigma(\tilde{z}_L)$ sigmoid fct.

    [$\hat{p}(y_i = 0 \mid x_i) = 1 - z_L$ is trivial, doesn't have to be learned]

    [class label easily obtained from posterior via

    $$\hat{y}_i = \begin{cases} 1 & \text{if } z_L > 0.5 \\ 0 & \text{if } z_L < 0.5 \end{cases}$$

    (or other threshold for assymetric costs)]
  - classification with $C > 2$: z_L is a probability vector of length C, i.e.

    $z_{Lk} = \hat{p}(y_i = k \mid x_i)$

    with normalization $\sum_{k=1}^{C} z_{Lk} = 1$

soft max activation generalizes sigmoid for multi-class problem (reduces to $z_{L1} = \sigma(\vec{z}_{L1} \cdot 2)$ if $C=2$)

$$z_{Lk} = \frac{e^{\vec{z}_{Lk}}}{\sum_{k'=1}^{C} e^{\vec{z}_{Lk'}}}$$

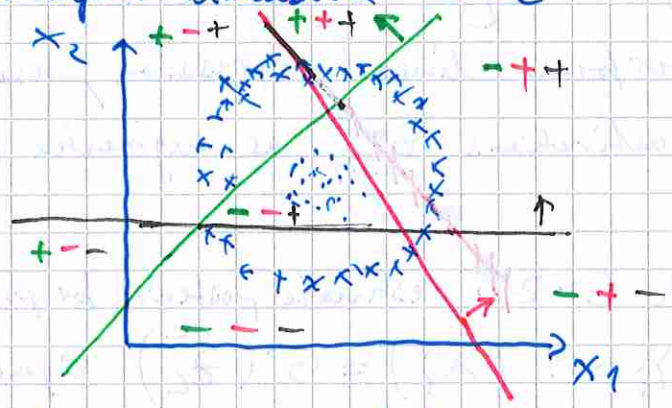
- illustration of how it works, using a 2-layer network with sign( ) activation (not suitable for gradient training, but easy to understand)

We have $$z_2 = \varphi_2\left(\varphi_1(x_i \cdot B_1) \cdot B_2\right)$$

with $\varphi_{1,2}(\vec{z}) = \text{sign}(\vec{z}) = \{-, +\}$

- we use $M_1 = 3$, $M_2 = 1$ (4 neurons in total)

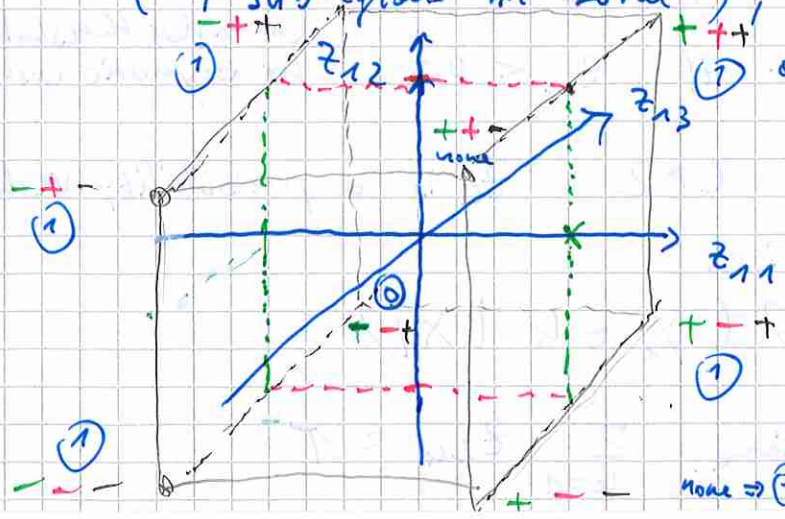
for input dimension $D = 2$



labels: x = 1
• = 0

first layer neurons define 3 decision planes
with arrows indicating the positive side

⇒ each subregion is assigned a particular sign combination (7 subregions in total), activations z_1 are corners of a cube.



Subregions are learned such that each contains a unique label.

second layer neuron must learn a decision plane that cuts off the corner $(-1,-1,+1)$ from the remaining corners of the cube,

e.g. $w_2 = (\underbrace{2}_{\text{bias weight}}, 1, 1, -1)$

with $\hat{y}_i = \begin{cases} 1 & \text{if } z_{1i} + z_{2i} - z_{3i} + 2 > 0 \\ 0 & \text{else} \end{cases}$

- loss functions are essentially the same as for linear models
  - regression: squared loss $\text{loss}(y_i^*, \hat{y}_i) = \|y_i^* - \hat{y}_i\|_2^2$
  - classification with $C=2$:

    ~~$z_{Li} = \hat{p}(y_i = 1 \mid x_i)$~~ scalar posterior prob. value (sigmoid output)

    $(\hat{p}(y_i = 0 \mid x_i) = 1 - z_{Li}$ trivial$)$

    $\Rightarrow$ logistic loss

    $$\text{loss}(y_i^*, z_{Li}) = \begin{cases} -\log z_{Li} & \text{if } y_i^* = 1 \\ -\log(1 - z_{Li}) & \text{if } y_i^* = 0 \end{cases}$$

  - classification with $C > 2$

    $z_{Lik} = \hat{p}(y_i = k \mid x_i)$ full probability vector (softmax output)

    with $\sum_k z_{Lik} = 1$

    $\Rightarrow$ cross-entropy loss

    $\text{loss}(y_i^*, z_{Li}) = -\log z_{Lik}$ with $k = y_i^*$

    (reduces to logistic loss for $C=2$)

    $\hat{p}(y_i = y_i^* \mid x_i)$ should be close to 1 (and the others close to 0)

    $$-\log z_{Li,k=y_i^*} = \begin{cases} \approx 0 & \text{if } z_{Li,k=y_i^*} \approx 1 \\ \text{big} & \text{else} \end{cases}$$

training by stochastic gradient descent

① pick a random instance i

② compute $\text{Loss}(y_i^*, \hat{z}_{Li})$ for current parameters $\Theta^{(t)}$

③ compute $\Delta\Theta = \frac{\partial \text{Loss}(y_i^*, z_{Li})}{\partial \Theta}$ (derivative of loss w.r.t. parameters)

④ update $\Theta^{(t+1)} = \Theta^{(t)} - \tau \Delta\Theta$

τ learning rate
subtraction, because we need descent to minimize loss

- step ③ is the critical part. The derivative is computed by back-propagation, i.e. by passing errors from output towards input.

Introduce auxilliary variable $\tilde{\delta}_\ell$ (row vector)

$$\tilde{\delta}_\ell = \frac{\partial \text{Loss}}{\partial \tilde{z}_\ell}$$

with $\tilde{z}_\ell = z_{\ell-1} B_\ell$ the pre-activations of layer ℓ (before applying nonlinearity $\varphi_\ell(\cdot)$)

← The chain rule gives

$$\tilde{\delta}_\ell = \frac{\partial \text{Loss}}{\partial \tilde{z}_\ell} = \frac{\partial \text{Loss}}{\partial \tilde{z}_{\ell+1}} \cdot \frac{\partial \tilde{z}_{\ell+1}}{\partial z_\ell} \cdot \frac{\partial z_\ell}{\partial \tilde{z}_\ell}$$

the RHS is simple:

$\frac{\partial \text{Loss}}{\partial \tilde{z}_{\ell+1}} =: \tilde{\delta}_{\ell+1}$ by definition of $\tilde{\delta}$. We already know this term because we work back to front.

$$\frac{\partial \tilde{z}_{\ell+1}}{\partial z_\ell} = \frac{\partial (z_\ell \cdot B_{\ell+1})}{\partial z_\ell} = B_{\ell+1}^T$$

$$\frac{\partial z_\ell}{\partial \tilde{z}_\ell} = \frac{\partial (\varphi_\ell(\tilde{z}_\ell))}{\partial \tilde{z}_\ell} = \text{diag}\left(\varphi_\ell'(\tilde{z}_\ell)\right)$$

$$\tilde{\delta}_\ell = \tilde{\delta}_{\ell+1} \cdot B_{\ell+1}^T \cdot \text{diag}\left(\varphi_\ell'(\tilde{z}_\ell)\right)$$

- The start of backpropagation is determined by the output activation:

  - regression: $\varphi_L(\tilde{z}_L) = \tilde{z}_L$

$$\text{Loss}(y_i^*, z_{Li}) = \frac{1}{2}(y_i^* - z_{Li})^2$$

$$= \frac{1}{2}(y_i^* - \tilde{z}_{Li})^2$$

$$\Rightarrow \frac{\partial \text{Loss}}{\partial \tilde{z}_L} = \tilde{\delta}_L = \tilde{z}_L - y_i^*$$

  - classification: $\varphi_L(\tilde{z}_L) = \text{softmax}(\tilde{z}_L)$

$$\text{Loss}(y_i^*, z_L) = \text{cross-entropy}(y_i^*, z_L)$$

$$\Rightarrow \left(\frac{\partial \text{Loss}}{\partial \tilde{z}_L}\right)_k = \begin{cases} z_{Lk} - 1 & \text{if } y_i^* = k \\ z_{Lk} & \text{if } y_i^* \neq k \end{cases}$$

- The derivative w.r.t. the parameters B_ℓ is easily computed from $\tilde{\delta}_\ell$

$$\frac{\partial \text{Loss}}{\partial B_\ell^T} = \left(\frac{\partial \text{Loss}}{\partial \tilde{z}_\ell}\right)^T \cdot \left(\frac{\partial \tilde{z}_\ell}{\partial B_\ell}\right)^T = \tilde{\delta}_\ell^T \cdot z_{\ell-1}$$

$$\left[\frac{\partial \tilde{z}_\ell}{\partial B_\ell^T} = \frac{\partial (z_{\ell-1} \cdot B_\ell)}{\partial B_\ell^T} = z_{\ell-1}\right]$$

⇒ gradient update with learning rate τ

$$B_\ell^{(t+1)} = B_\ell^{(t)} - \tau \cdot \underbrace{\left(z_{\ell-1}^T \cdot \tilde{\delta}_\ell\right)}_{\text{outer product}}$$

- Fortunately, modern neural network libraries (pytorch, tensorflow) contain a module "autograd", which computes these (and more complex) derivatives automatically ⇒ you don't have to do these calculations by hand and implement the resulting algorithm

⇒ eliminated big source of bugs ~~and allows for very complex models~~

fundamental insight 3

NEURAL NETWORKS SUPPORT ~~MODELS~~ NON-LINEAR MODELS OF ALMOST ARBITRARY COMPLEXITY, BECAUSE:

- we can construct highly complex architectures by replicating a single basic unit (neuron = linear model plus non-linear activation) in parallel (layers) and in series (deep)
- auto grad allows to compute the gradients needed for training automatically, without additional programming
- modern GPUs make the forward and backward computation sufficiently fast.

Why do neural networks work so well?

- Theoretically not yet understood. NNs have so many parameters that they should overfit terribly (i.e. have small training error, but large generalization error). But this does not happen in practice.
- Theoretical guarantee: universal approximation theorems
  - 1-D regression: $z_L \in \mathbb{R}$

    2-class classification $z_L = \hat{p}(y_i = 1 | x_i) \in [0,1]$

  can be shown, that a 2-layer network ($L=2$) can approximate arbitrarily complicated true mappings ($y_i^* = f^*(x_i)$ or $p^*(y_i = 1 | x_i)$)

  If the hidden layer is sufficiently wide ($M_1 \to \infty$) and the weights B_1, B_2 are suitably chosen.

  However, this is a pure existence proof, gives no algorithm to determine M_1, B_1, B_2.
- Finding the globally optimal B_ℓ for any network is NP-hard (i.e. virtually impossible)

Fundamental insight 4:

IF THE NETWORK IS SUFFICIENTLY LARGE (WIDE AND DEEP), AND WE HAVE ENOUGH TRAINING DATA, TYPICAL LOCAL OPTIMA TEND TO BE PRETTY GOOD (≈ CLOSE TO THE GLOBAL ONE, I.E. HAVE ~~GOOD~~ LOW GENERALIZATION ERROR).

This is an empirical finding, not yet theoretically understood, why this is the case.

Training tricks

- early stopping: regularly check generalization error during training on an independent validation set. Stop if this error estimate starts to increase (even if the training error would keep decreasing)
- training rate schedule: when training error saturates, decrease training rate $\tau \leftarrow \tau/10$. (repeat 2 or 3 times)
- mini-batch SGD: instead of estimating the loss from a single random instance, as in pure SGD, use the average over a random set of size k ("mini-batch") $k = 32 \ldots 10^4$, as GPU-RAM allows more ?
- ADAM optimizer (and similar relatives): Plain SGD uses uniform learning rate τ for all parameters. ADAM adjusts learning rate for each parameter ⇒ much faster convergence (or divergence, if unlucky).

  Let $g^{(t)} = \frac{\partial \text{Loss}^{(t)}}{\partial B^{(t)}}$ the loss derivative in iteration t

  Plain SGD: $B^{(t+1)} = B^{(t)} - \tau\, g^{(t)}$

  ADAM maintains a running average of g and g^2!

$$G^{(t)} = \mu_1 G^{(t-1)} + (1-\mu_1)\, g^{(t)}$$

$$(G^2)^{(t)} = \mu_2 (G^2)^{(t-1)} + (1-\mu_2)\,(g^{(t)})^2$$

$(\mu_1 \approx 0.9\,,\ \mu_2 \approx 0.999)$

$$\tilde{G}^{(t)} = \frac{G^{(t)}}{1-\mu_1^t} \qquad (\tilde{G^2})^{(t)} = \frac{(G^2)^{(t)}}{1-\mu_2^t}$$

"burn-in phase correction"

$$B^{(t+1)} = B^{(t)} - \tau \frac{\tilde{G}^{(t)}}{\sqrt{(\tilde{G^2})^{(t)}} + \varepsilon} \qquad \varepsilon = 10^{-8}$$

(avoid div-by-zero)

- batch normalization: normalize pre-activations $\tilde{z}_l$ by their mean and std-dev within current mini-batch
  - standard network $\tilde{z}_l = z_{l-1} \cdot B_l \,,\ z_l = \varphi_l(\tilde{z}_l)$
  - batch norm:

$$z'_{l,i} = z_{l-1,i} \cdot B_l \quad \text{for } i \text{ in mini-batch}$$

$$\mu = \frac{1}{k}\sum_i z'_{l,i} \,,\quad \sigma = \sqrt{\frac{1}{k}\sum_i (z'_{l,i}-\mu)^2 + \varepsilon}$$

$$z''_{l,i} = \frac{z'_{l,i} - \mu}{\sigma}$$

$$\tilde{z}_{l,i} = a_l \cdot z''_{l,i} + b_l \qquad (a_l, b_l: \text{learnable parameters})$$

$$z_{l,i} = \varphi_l(\tilde{z}_{l,i})$$

often significant improvements of training and generalization errors

- dropout: ~~randomly~~ deactivate part (e.g. 50%) of the network in every mini-batch:
  sample dropout masks $\kappa_l \in \{0,1\}^{M_l}$ and replace activations $z_{l,i}$ with $z_{l,i} = \kappa_l \cdot \varphi_l(\tilde{z}_{l,i})$

⇒ reduces overfitting, because subtle interactions between neurons are made impossible

during prediction, downscale weights B_ℓ by dropout factor (e.g. $B_\ell \leftarrow B_\ell \cdot 0.5$) and predict without dropout

- weight initialization:

$$B_\ell \sim \mathcal{N}\left(0,\ \sigma^2 = \frac{1}{M_{\ell-1}+1}\right) \quad \text{(for tanh(.))}$$

$$\sim \mathcal{N}\left(0,\ \sigma^2 = \frac{2}{M_{\ell-1}+1}\right) \quad \text{(for ReLU(.))}$$

$\Rightarrow$ ensures that (expected) norm of the input weights of each neuron (incl. bias weight) is 1.

- piecewise linear activation functions (ReLU, PReLU, Leaky ReLU) tend to work better than δ and tanh, because they do not suffer from vanishing gradients when $\tilde{z}_\ell \gg 0$.

- data augmentation: artificially increase training set size by applying random perturbations to the existing instances, e.g.: - mirror or rotate images
  - add slight noise
  - random (affine or) non-rigid morphing

$\Rightarrow$ make ~~training~~ network robust against those perturbations (because y_i^* remains unchanged or is perturbed according(ly))

- except for data augmentation, predefined modules for all those which are already available in NN libraries

- weight decay: add regularizer to loss that penalizes very large weights: L2: $\lambda \sum_\ell \|B_\ell\|_F^2$

L1: $\lambda \sum_\ell |B_\ell|$ (elementwise abs)

Convolutional Neural Networks

- in many applications, i.e. analysis of audio, images, videos, data are (approximately) translation invariant

human and dog

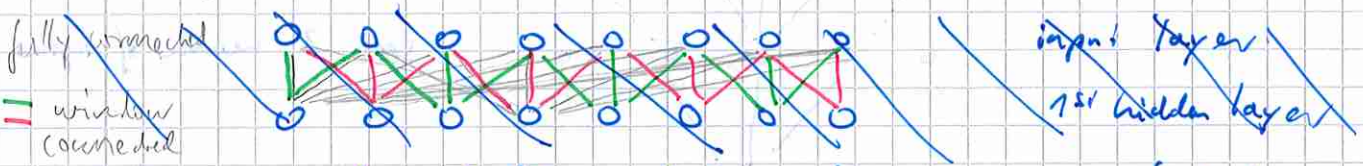
sliding window

~~⇒ important receptive fields should not~~

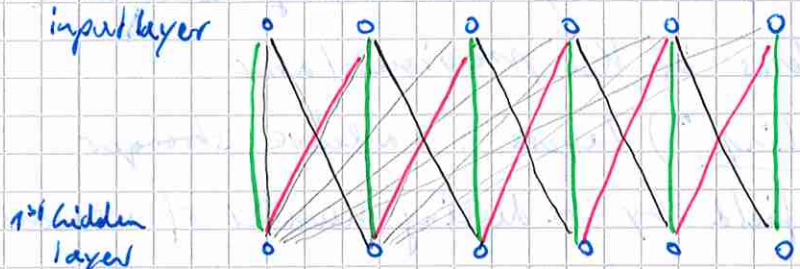
⇒ should not only look at image as a whole, but also at every subregion of a certain (or various) size "receptive field"

- these receptive fields should cover the entire image in a sliding window fashion

- consequences for network architecture (1D illustration)

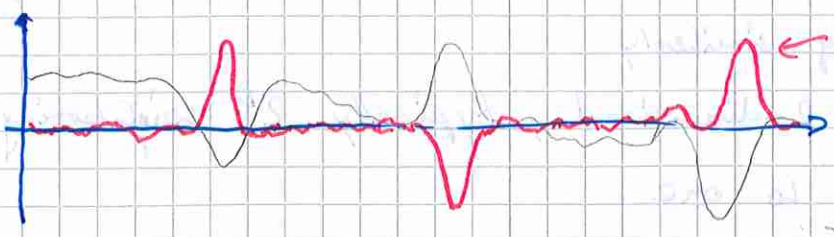


- connections to the next layer become sparse (window of attention)
- connections of same color share weight (translation invariance)



⇒ We do not need to repeat the redundant network architecture, but instead use a loop to actually move the window around. The weights in a window act as a filter that detect patterns of interest.

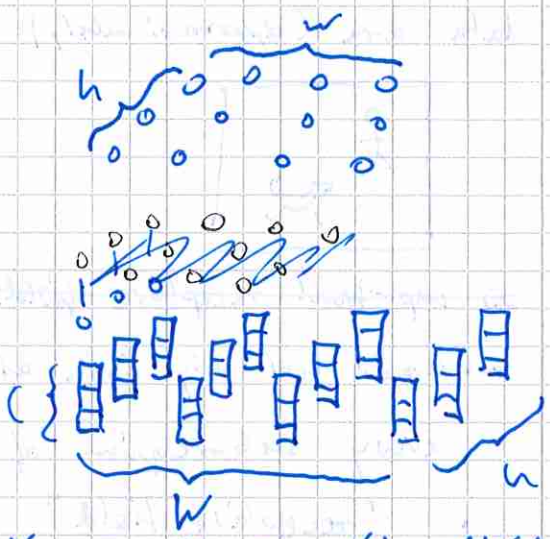
Matched filter with weights (+1) (−2) (+1)



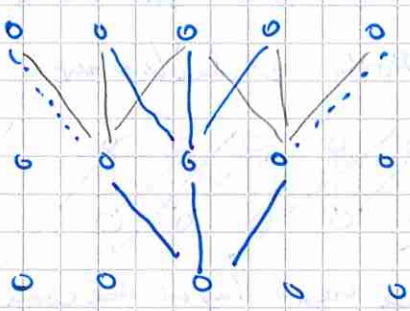
- Images contain many different patterns of interest ⇒ run many filters in parallel. The output of each filter forms a <u>channel</u> of the next neuron layer.

  input layer (gray scale image)
  $w \times h$

  1st hidden layer after C filters:
  $C \times w \times h$



- deep networks: stacking layers with small receptive fields yields a larger receptive field



sees 3 neurons of input

sees 3 neurons of first layer
⇒ 5 neurons of input

⇒ complex patterns are formed by combining activation of several simple patterns in the previous layer

- downsampling ("<u>pooling</u>") layers: achieve stronger increase of receptive field by reducing number of neurons ($\hat{=}$ pixels). 1D example

  $z_{k,i}$ = i^{th} layer value

max-pooling

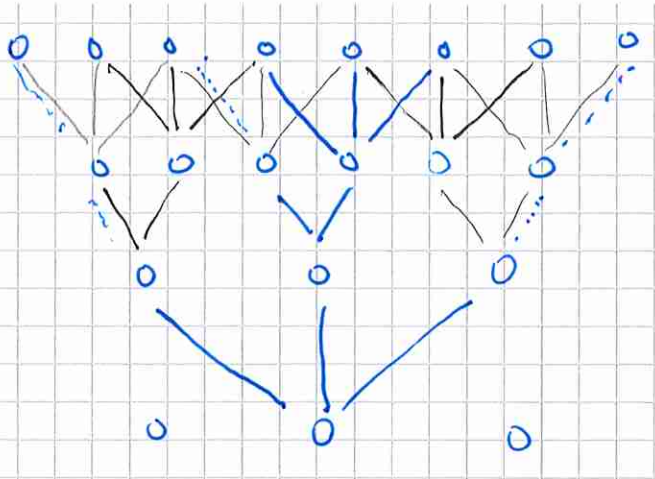
$z_{(k+1),1} = \max(z_{k,1}, z_{k,2})$

$z_{(k+1),2} = \max(z_{k,3}, z_{k,4})$

$z_{(k+1),3} = \max(z_{k,5}, z_{k,6})$

average pooling similarly

- if signal is 0-dimensional, typically 2^D neighboring elements are reduced to one.



neuron sees 8 neurons in the input layer

- strided convolution has a similar effect: don't move window to the next pixel, but skip one or more ≙ "stride"
  ⇒ downsampling according to how many pixels are skipped

- convolutional layers are specified by window size and number of filters:
  - $5 \times 5 \times 8$: 8 filters of size 5×5
  - $3 \times 3 \times 32$: 32 — " — 3×3
  - $1 \times 1 \times 128$: 128 projections of the input channels (just scalar products of the center pixel)

- show famous networks on slides
  - classification networks (detect objects in image)
    LeNet, AlexNet, GoogLeNet, VGG, ResNet
  - segmentation networks: U-net, fully convolutional networks
- show applications on slides