LMU

LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK

LEHRSTUHL FÜR DATENBANKSYSTEME
UND DATA MINING

AI beyond

Lecture Notes on

Deep Learning and Artificial Intelligence

Winter Semester 2024 /2025

Neural Networks and their mathematical foundations

Script © 2024 Matthias Schubert

Chapter Overview

- Mathematical foundations:
  - vectors, matrices, and tensors
  - similarity, norms, and distance metrics
  - derivatives and gradients
  - 2nd order derivatives and extreme values
  - distributions and probability variables
- Basic Neural Networks
  - non-linearities
  - loss functions
  - weight initialization and input normalization
  - gradient descent
  - optimization techniques

Vectors, Matrices and Tensors

- Vectors, Matrices and Tensors $x \in \mathbb{R}^d, X \in \mathbb{R}^{n \times d}, \mathbb{X} \in \mathbb{R}^{d_1 \times \ldots \times d_l}$
- inner product (dot product): $x^T u = \sum_{i=0}^{d} x_i \cdot u_i$
- outer product: $xu^T = \mathbf{x} \otimes \mathbf{u} = \begin{bmatrix} x_1 u_1 & x_1 u_2 & \cdots & x_1 u_m \\ x_2 u_1 & x_2 u_2 & \cdots & x_2 u_m \\ \vdots & \vdots & \ddots & \vdots \\ x_n u_1 & x_n u_2 & \cdots & x_n u_m \end{bmatrix}$
- matrix multiplication:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Uses of Matrix Algebra

- linear function:

$$f : \mathbb{R}^d \rightarrow \mathbb{R}^n, f(x) = x^T W + b \text{ where } W \in \mathbb{R}^{d \times n}, b \in \mathbb{R}.$$

- cosine of angle α between two vectors $x, y \in \mathbb{R}$

$$cos(\alpha) = \frac{x^T y}{||x|| \cdot ||y||}$$

- kernel matrix matrix of a data set $X \in \mathbb{R}^{n \times d}$

$$K = X \cdot X^T \text{ with } x_{i,j} = X_i^T x_j$$

Metrics and Scalar Products

- Euclidian scalar product:

$$x, y \in \mathbb{R}^d : x^T y = \sum_{i=0}^{d} x_i \cdot y_i = \langle x, y \rangle$$

  – properties:
    - $\langle x, y \rangle = \langle y, x \rangle$ commutative
    - $\langle x + y, z \rangle = \langle x, z \rangle + \langle y, z \rangle$ distributive
      $\langle x, y + z \rangle = \langle x, y \rangle + \langle x, z \rangle$

- Euclidian Norm: $\sqrt{\langle x, x \rangle} = ||x||_2$
- Euclidian Metric:

$$Dist(x, y) = ||x - y|| = \sqrt{\langle x, x, \rangle + \langle y, y, \rangle - 2\langle x, y \rangle}$$

Differential Calculus

- given $f : \mathbb{R} \to \mathbb{R}$, the derivative is defined as:

$$f'(x) = \frac{d}{dx} f\left(x\right) = \lim_{\Delta \to 0} \frac{f\left(x + \Delta\right) - f\left(x\right)}{\Delta}$$

- given $f : \mathbb{R}^d \to \mathbb{R}$, the partial derivate is defined as:

$$\frac{\partial f}{\partial x_i} = \lim_{\Delta \to 0} \frac{f\left(x_1, \ldots, x_i + \Delta, \ldots, x_d\right) - f\left(x_1, \ldots, x_d\right)}{\Delta}$$

- Jacobian of $f : \mathbb{R}^n \to \mathbb{R}^m$

$$\mathbf{J} = \frac{d\mathbf{f}}{d\mathbf{x}} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \cdots \frac{\partial \mathbf{f}}{\partial x_n}\right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

2nd order Derivatives

- for $f : \mathbb{R} \to \mathbb{R}$: $\dfrac{\partial^2 f}{\partial x^2} = \dfrac{\partial f'(x)}{x}$
- for $f : \mathbb{R}^d \to \mathbb{R}$: $(\text{Hess}\, f)_{ij} = \dfrac{\partial^2 f}{\partial x_i \partial x_j} = \begin{bmatrix} \frac{\partial^2 f}{\partial^2 x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \cdots & \frac{\partial^2 f}{\partial^2 x_d} \end{bmatrix}$
- for $f : \mathbb{R}^n \to \mathbb{R}^m$: we a matrix per ouput channel

Optimization

- an optimization problem is defined as :

$$argmin_\theta J(\theta)$$
where $\theta \in \mathbb{R}^p$ and $J : \mathbb{R}^p \rightarrow \mathbb{R}$

- constrained optimization:

$$argmin_\theta J(\theta)$$
subject to

$$c_i(\theta) = 0, i \in \mathbb{I} \tag{1}$$
$$c_k(\theta) \geq 0, k \in \mathbb{K} \tag{2}$$

where $\theta \in \mathbb{R}^p$ and $J : \mathbb{R}^p \rightarrow \mathbb{R}$

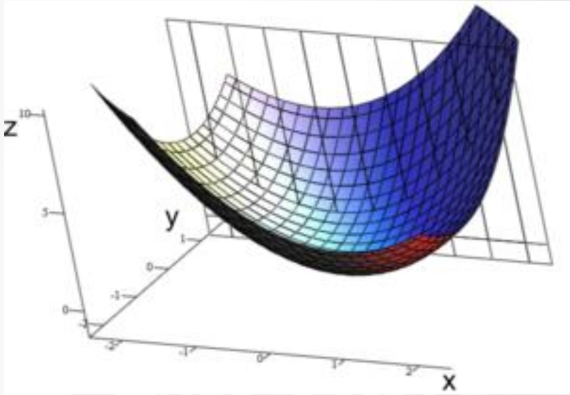
Optimization and Gradients

- Convex functions:

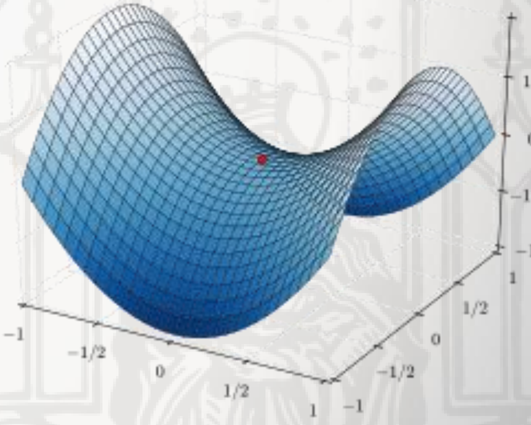
$$\text{for } 0 \leq t \leq 1 \text{ and } x_1, x_2 \in \mathbb{X}\text{:}$$
$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

  - alternative definition, the Hessian is positive semidefinite
  - convex functions have a unique minimum
  - $J'(\theta) = 0$ yields the solution for $argmin_\theta J(\theta)$

- Non-Convex functions:
  - $J'(\theta) = 0$ could be a saddle point
  - Second partial derivative test based on $det(\text{Hess } f(\theta))$ (<0 indicates saddle point)



https://en.wikipedia.org/wiki/Convex_function



https://en.wikipedia.org/wiki/Saddle_poin
t

Finding the root of a function

2-order methods like Newton-Raphson (simple Newton)

- for univariate functions:

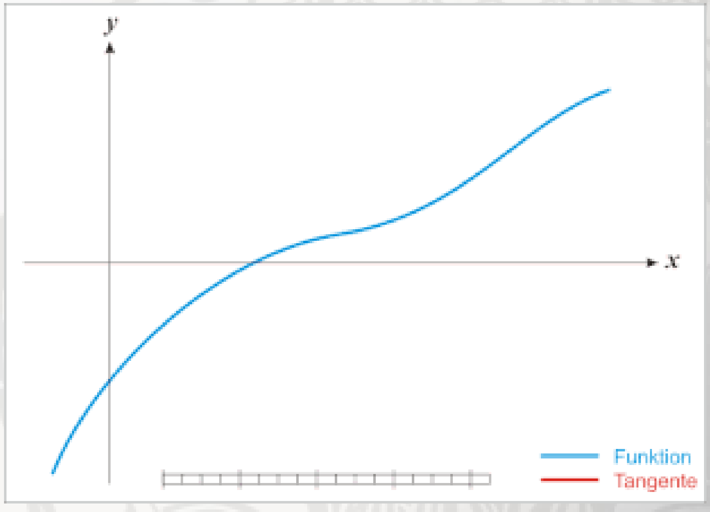
$$f:\mathbb{R}\rightarrow\mathbb{R}: f'(\theta_n) = \frac{f(\theta_n)-0}{\theta_n-\theta_0} \Rightarrow \theta_0 = \theta_n - \frac{f(\theta_0)}{f'(\theta_0)}$$

- for multivariate functions:

$$f:\mathbb{R}^d\rightarrow\mathbb{R}: \theta_0 = \theta_n - \left(\frac{\partial f(\theta)}{\partial\theta\partial\theta^T}\right)^{-1}\frac{\partial f(\theta)}{\partial\theta}$$

Caution: For optimization, $f(\theta) = J'(\theta)$.

We need to compute the Hessian of $J'(\theta)$ to apply 2nd order optimization!



First order optimization

gradient descent or linear search:

- does not try to find the root of the derivative
- instead: walk along the gradient in small steps until it is 0.
- the derivative describes the slope of the tangent
- moving along the negative derivative for a small enough step α, leads to a smaller function value: $\theta_i = \theta_{i-1} - \alpha \cdot f'(\theta_{i-1})$

$$\exists \alpha \in \mathbb{R} : f(\theta_i) \leq f(\theta_{i-1}$$

If α is too big, I won't work properly

More, when we talk about training Neural Networks.

0-Order Optimization

- 1^{st} and 2^{nd} order optimizers require differential functions
- works on any function
- Idea: Instead of computing a gradient, sample k update steps and take the best ones. (Monte Carlo Search)

$$\theta_i = \theta_{i-1} - u \text{ where } u = argmin_{u \in U} f(\theta_{i-1} - u)$$

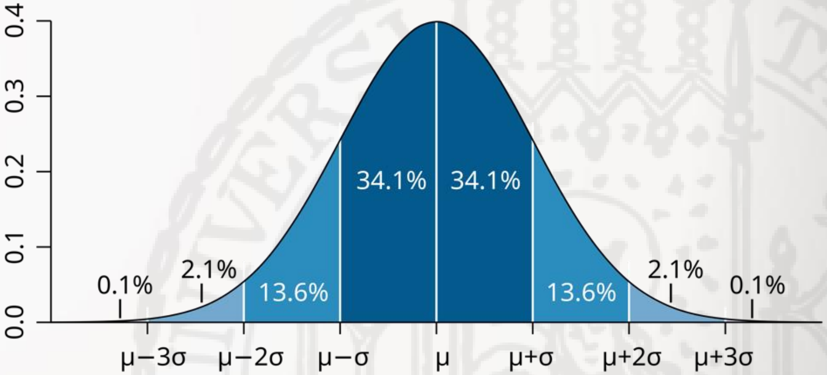
- u is drawn from a random distribution
- the larger U the more likely we find an improvement.
- more sophisticated methods use populations of parameters and more fancy methods to generate updates
- for example: *Genetic Algorithms*

Probability Distributions

- sample space Ω
- for discrete Ω:
  - $\mathrm{Pr}(\omega)$ corresponds to the relative frequency when drawing ω from Ω.
  - $\sum_{i=0}^{|\Omega|} Pr(\omega_i) = 1$ and $0 \leq Pr(\omega_i) \leq 1$
- for continuous Ω :
  - $\mathrm{Pr}(\omega_i) = 0$ because $|\Omega| = \infty$
  - probability density function $p(\omega_i)$:

    $$\int_{-\infty}^{\infty} p(\omega)d\omega = 1$$
  - cumulated Probability:

    $$\mathrm{Pr}(\omega < x) = \int_{-\infty}^{x} p(\omega)d\omega$$



https://en.wikipedia.org/wiki/Probability_distribution

Mean and Variance

- a random variable X is a variable following a distribution over a sample space Ω.
- the expectation value of random variable X is: $\sum_{i=o}^{|\Omega|} p(x_i)x_i$ or $\int_{-\infty}^{\infty} p(x)\, x\, dx$
- the variance of X is: $Var(X) = \mathrm{E}\left(\left(X - \mathrm{E}(x)\right)^2\right) = \frac{\sum_{i=0}^{|\Omega|}(x - E(x))^2}{|\Omega|}$
- covariance of two random variable X,Y:

$$\mathrm{Cov}(X,Y) = \frac{\sum_{i=0}^{|\Omega|}(x - E(X))(y - E(y))}{|\Omega|}$$

Mean and Variance for Random Vectors

- a random vector *X* consist of random variables x_i in each dimension $1 \leq i \leq d$

- mean vector: $\mathrm{E}(X) = \begin{pmatrix} E(x_1) \\ \vdots \\ E(x_d) \end{pmatrix}$

- Co-Variance matrix of *X* :

$$Cov(X) = \begin{pmatrix} Var(x_1) & \dots & Cov(x_1, x_d) \\ \vdots & \ddots & \vdots \\ Cov(x_d, x_1) & \dots & Var(x_d) \end{pmatrix}$$

Estimating mean and covariance

- given a sample space Ω and data X drawn from the distribution $\mathrm{Pr}(x)$ over Ω:

- the empirical mean is computed as: $\hat{E}(X) = \frac{\sum_{i=0}^{|X|} x_i}{|X|}$
- sample bias: $\hat{E}(X) - E_{\mathrm{Pr}(x)}$

- covariance matrix of data matrix $X \in \mathbb{R}^{n \times d}$:

$$\frac{1}{|X|}(X - E(X))^T(X - E(X))$$

(normalized matrix product of the centered data matrix)

Estimating function parameters

given:

- observation $X \in \mathbb{R}^{n \times d}$, and labels $\mathrm{Y} \in \mathbb{R}^{n \times l}$
- a prediction function $f_\theta(x) = \hat{y}$
- Likelihood function $\mathcal{L}(\theta, Y)$ describes how likely Y would be observed for parameters θ when feedinng X to $f_\theta(x)$.

A maximum likelihood estimator determines θ^*

$$\theta^* = argmax_\theta \mathcal{L}(\theta, Y)$$

Universal function approximators

- **Input domain**: $x \in X \subseteq \mathbb{R}^{d_1 \times .. \times d_l}$
  (l=1 for vectors, l=3 for images, i.e., width, height, RGB channels)

- **Output domain**: y $\in y \subseteq \mathbb{R}^{d_1 \times .. \times d_l}$
  (class probability, continuous prediction targets, images, text tokens, ..)

- A neural network is a parametric function $f_\theta : X \rightarrow Y$
  where $\theta \in \Theta \subseteq \mathbb{R}^{d_1 \times .. \times d_l}$ is called weights/parameters

- For a dataset $\mathrm{D} = \{\mathrm{x_i}, \mathrm{y_i}\}_{i=1}^{n} \in X \times Y$ and $f(x_i, \theta) = \hat{y}_\mathrm{i}$, we want to optimize an objective function $\mathrm{J}_\theta : Y \times Y \rightarrow \mathbb{R}$ describing the similarity between $\hat{y}_\mathrm{i}$ and $\mathrm{y_i}$ for $1 \leq i \leq n$
  - as D is given, optimization needs to modify the weights θ:
    for training, we consider $J_D(\theta)$ and in particular, its gradient $\nabla_\theta \, J_D(\theta)$

Neural Network Architecture

- the exact design of $f_\theta : X \rightarrow Y$ is called network architecture
- for now, we assume linear or dense functions:

$$f_\theta(x) = W^T x + b$$

- most architectures stack layers of functions:

$$f_\theta(x) = f^1(f^2(\dots f^n(x) \dots)$$

- stacking multiple linear layer functions results in a single linear function
  $\Rightarrow$ separate layers by non-linear functions σ

$$f_\theta(x) = f^1(\sigma_2 f^2(\dots \sigma_n f^n(x) \dots)$$

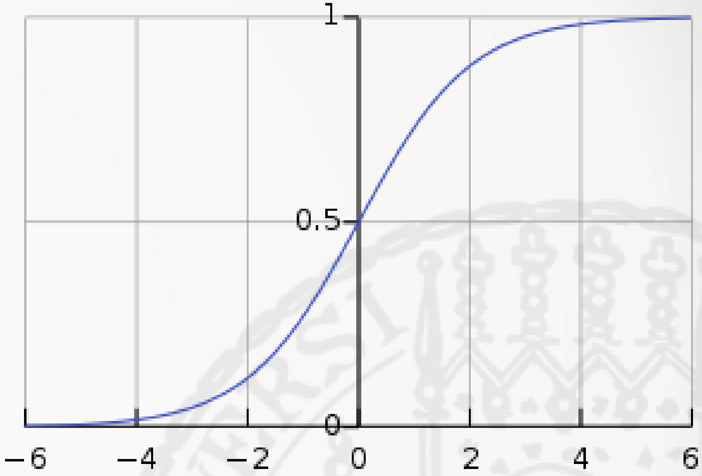
  σ is also called **non-linearity** or **activation function**

Non-Linearities

bounded logistic functions aka sigmoid functions:

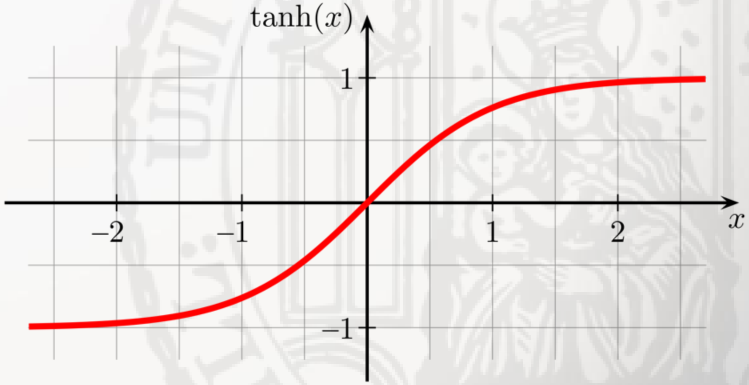
- logistic function:

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



- hyperbolic tangent:

$$\tanh(x) = 1 - \frac{2}{e^{2x}+1}$$



Rectified Linear Units (ReLU)

- ReLU:

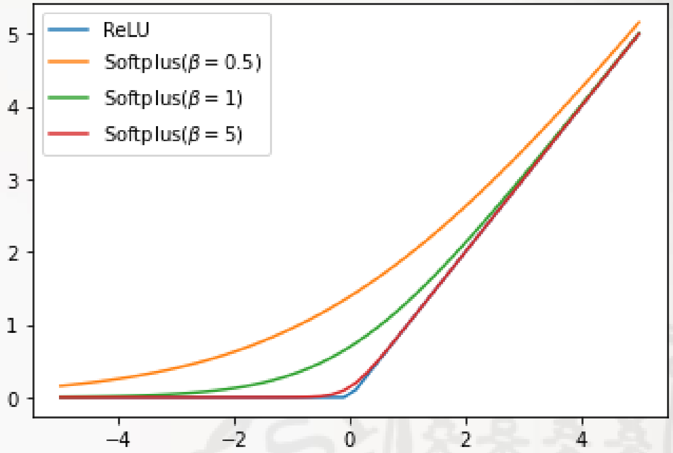
$$\sigma(x) = \begin{cases} x \; if x > 0 \\ 0 \; if \; x \leq 0 \end{cases}$$

- Softplus:

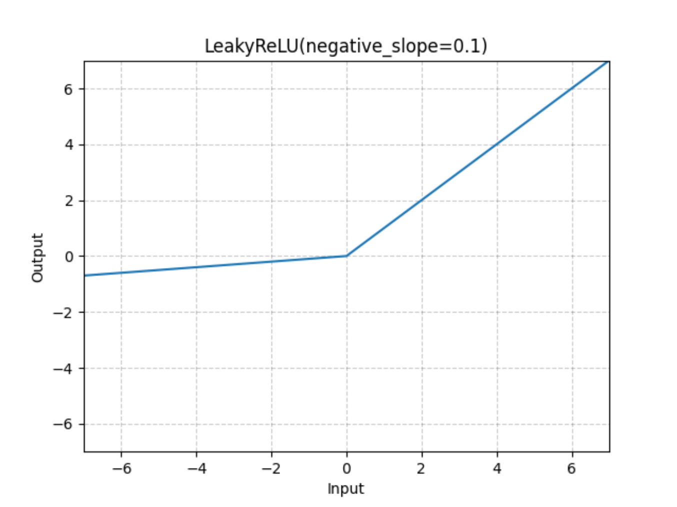
$$\sigma(x) = \frac{1}{\beta}\log\left(1 + e^{\beta x}\right)$$

- leaky ReLU:

$$\sigma(x) = \begin{cases} x \; if x > 0 \\ cx \; if \; x \leq 0 \end{cases} with \, c < 1$$



1.

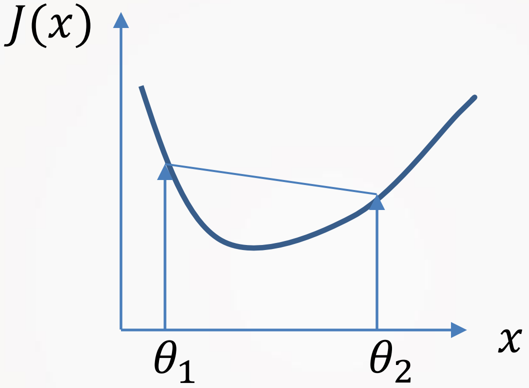


2.

1. https://pat.chormai.org/blog/2020-relu-softplus#
2. https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html

Convex and Non-Convex Objective Functions

- $\forall(\theta_1, \theta_2) \in \Theta \times \Theta, \alpha \epsilon [0,1] : \mathrm{J}_D(\theta)$ is convex if
$$J_D(\alpha\theta_1 + (1-\alpha)\theta_2) \leq \alpha J_D(\theta_1) + (1-\alpha) J_D(\theta_2)$$



- for a convex function, any local minimum is global
- a univariate function is convex if the second derivative is non-negative for any parameter value
- a multivariate function is convex if the Hessian is positive semidefinite (remember: the Hessian is the matrix of 2nd derivatives $\frac{\partial^2 f}{\partial x_1 \partial x_2}$)

Remarks on Convex Objectives

- objective functions for neural networks are usually non-convex
  ⇒ convexity of $J_D(\theta)$ often depends on the complexity of $f_\theta(x)$
  ⇒ optimisation of $J_D(\theta)$ often leads to local minima only

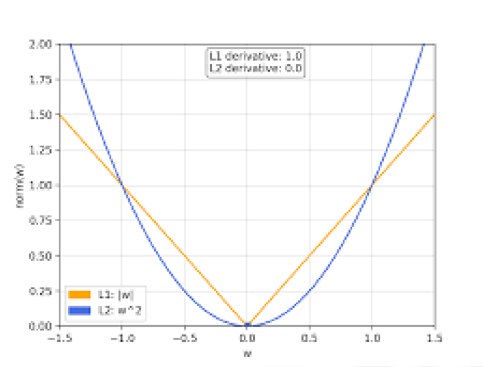
- optimal parameters $\boldsymbol{\theta}^*$ are depending on *D*
  ⇒ if you resample *D*, $\boldsymbol{\theta}^*$ might not be optimal anymore.

- convexity can be exploited in optimisation algorithms

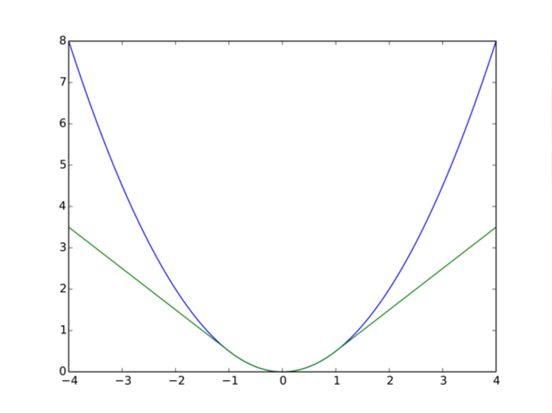
Objectives Continuous Outputs

for y ∈ $\mathbb{R}^n$ (if y ∈ $\mathbb{R}^{d_1 \times .. \times d_l}$, we can flatten it to ý ∈ $\mathbb{R}^{d_1 + .. + d_k}$)

- L1-loss: $L(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i|$
- L2-Loss: $L(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2$



- Huber-Loss: $L(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} \frac{1}{2}(\hat{y}_i - y_i)^2 & if |\hat{y}_i - y_i| < \delta \\ \delta\left(|\hat{y}_i - y_i| - \frac{1}{2}\delta\right) & otherwise \end{cases}$



Categorical Objectives (1)

Categorical data is usually encoded as a distribution over discrete outputs. For *n* categorical outputs, we consider $\hat{y} \in [0,1]^n$ and y $\in \{0,1\}^n$ as the output space:

Case 1: disjunctive categories (**Cross Entropy**)

- as y is a one hot encoding $\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} \hat{y}_i = 1.$
- f(x) is an arbitrary vector in $\mathbb{R}^n$
- use softmax function to resemble a quasi-one-hot distribution:

$$\hat{y}_i = \hat{f}_i(x) = \frac{e^{f_i(x)}}{\sum_{j=1}^{n} e^{f_j(x)}}$$

- Cross-Entropy Loss:

$$J_\theta(X,Y) = \mathbb{E}_{(x,y)\sim D}[-y^T \log\left(\hat{f}(x)\right)] \approx -\frac{1}{k}\sum_{i=1}^{k} y_i^T \log\left(\hat{f}(x_i)\right)$$

note: cross entropy only considers dimension *j* with $y_j = 1.$

If $\hat{f}_i(x) \rightarrow 0$ then $-\log\left(\hat{f}(x_i)\right) \rightarrow \infty$.

Categorical Objectives (2)

Case 2: overlapping categories (**Binary Cross Entropy**)

- each category needs to be treated independently $\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} \hat{y}_i \leq n$
- each output dimension is scaled to [0,1] by a logistic function $\hat{f}_i(x) = \frac{1}{1+e^{-f_i(x)}}$
- Binary Cross-Entropy for k samples and n categories

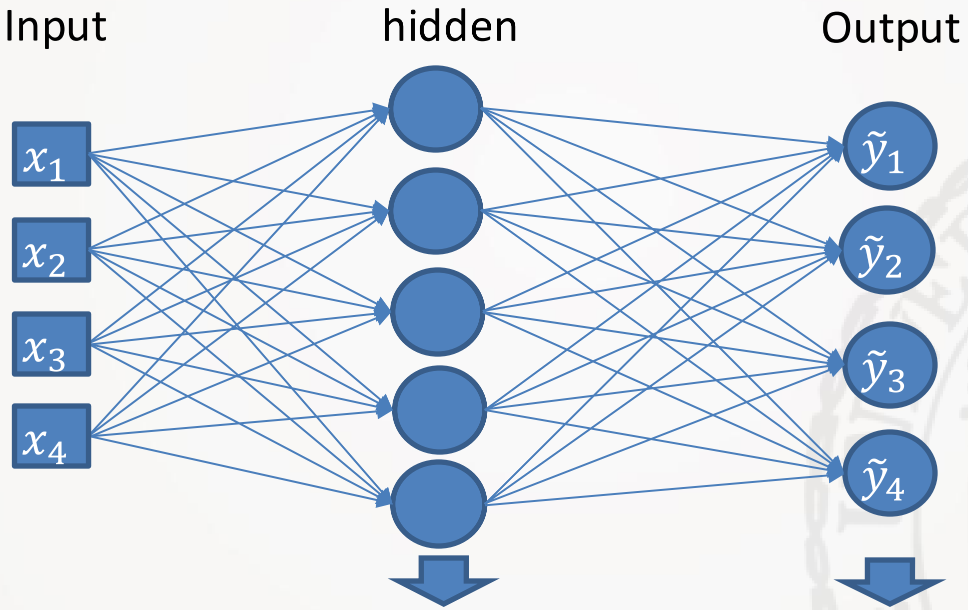
$$J_\theta(X,Y) = -\mathbb{E}_{(x,y)\sim D}[y^T \log\left(\hat{f}(x)\right) + ((1-y)^T \log(1-\hat{f}(x))]$$

$$\approx -\frac{1}{k}\sum_{i=1}^{k} y_i^T \log\left(\hat{f}(x_i)\right) + (1-y_i)^T \log(1-\hat{f}(x_i))$$

note: $(1-y)\log(1-\hat{f}(x))$ is required to punish large values for categories *x* does not belong to ($y_j = 0 \; but \; \hat{f}(x_i) > 0$).

Multilayer Perceptron Architecture

neural network architecture consisting of *n* linear layers with an arbitrary non-linearity. (often 2 layers & logistic function)



$$f(x) = \sigma(W_h x^T + b_h) \qquad \tilde{y}\,(x) = \sigma(W_{out} f(x)^T + b_{out})$$

Parameter Optimization

Various optimisation methods are available for different categories of objective functions:

- linear solvers (only for linear objectives)
- quadratic solvers (for quadratic objectives)
- Newton-Raphson (2. order optimisation requires the Hessian)
- ***gradient descent*** *(1. order optimisation* requires *gradient)*
- genetic and random search algorithms (0. order optimisation: not requirements)
- various types of conjugate gradients (various variants, good for sparse linear systems)

Note: Methods vary in resource requirements, convergence time, parameter sensitivity and whether they are guaranteed to converge.

Optimization for Neural Networks

- architectures are based on stacking non-linear functions
  - objectives are usually not convex
  - most architectures do not belong to a specific type of function for which dedicated optimisation algorithms are available

- architectures are generally continuously differentiable
  - methods like gradient descent and Newton-Raphson can be applied
  - Newton-Raphson requires the Hessian, which is quadratic in the number of parameters (infeasible for very large models)
  - if not, only genetic and randomised search is applicable

Gradient Descent (GD)

- recapitulate: the **derivative** measures how much a function f(x) increases relative to a change in x, i.e., $f'(x) = \lim_{\Delta \to 0} \frac{f(x+\Delta) - f(x)}{\Delta}$ and the **gradient** is the derivate at a particular point x_0.
- going into the opposite direction decreases f(x):

$$\exists \alpha > 0\colon\ x_{t+1} \leftarrow x_t - \alpha f'(x_t) \Longrightarrow f(x_{t+1}) \leq f(x_t)$$

  – α is a hyperparameter called learning rate or step size

  – adding the gradient is used for maximization (gradient ascent)

  – gradient descent: iterate steps until $f(x_{t+1}) = f(x_t)$ or max. steps

  – GD converges if $f(x)$is differentiable and convex

  – If $f(x)$ is not differentiable, we encounter an error when reaching an x where $f'(x_t)$ is undefined.

  – If $f(x)$ is not convex, GD might not reach a global minimum but a local minimum or a saddle point.

Stochastic Gradient Descent (GD) Algorithms

```
Input: $\mathrm{D} = \{\mathrm{x_i}, \mathrm{y_i}\}_{i=1}^{n}, \alpha, f(x, \theta)$
output: $\theta^*$ (local) optimal function parameter
init $\theta$ $(weight\ initialization)$
for e in episodes:
    for $\mathrm{x_i}, \mathrm{y_i}$ in samples D:
        $\hat{y}_i = \mathrm{f}(\mathrm{x_i}, \theta)$ #forward path
        $\mathrm{L}_\theta = \mathrm{J}(\theta, \hat{y}_i, \mathrm{y}_i)$ #compute loss
        $\theta = \theta - \alpha \nabla_\theta L_\theta$ # backward path
return $\theta$
```

GD for Neural Networks

- neural network f(x, θ) is a multi-layer non-linear function
- θ is a simplification for multiple groups of parameters

Example:

For $\tilde{y}\,(x) = \sigma(w_o(\sigma(W_i\, x^T + b_i))^T + b_o)$:

4 types of parameters: $\theta = (w_o, b_o, W_i, b_i)$

Each has a different derivative:

let $L_0^T = \sigma(W_i\, x^T + b_i)$

$$\frac{\partial\sigma(w_o L_0^T + b_o)}{\partial W_{o,i}} = \frac{\partial\sigma(w_o L_0^T + b_o)}{\partial w_o L_0^T + b_o}\,\frac{\partial w_o L_0^T + b_o}{\partial w_{o,i}} = \frac{\partial\sigma(w_o L_0^T + b_o)}{\partial w_o L_0^T + b_o}\,\frac{\partial \sum_{j=0}^{n} w_{o,j} L_j + b_o}{\partial w_{o,i}} =$$

$$\frac{\partial\sigma(w_o L_0^T + b_o)}{\partial w_o L_0^T + b_o}\left(\frac{\partial \sum_{j\neq i} w_j L_j + b_o}{\partial w_{o,i}} + \frac{\partial w_{o,i} L_i}{\partial w_{o,i}}\right) = \frac{\partial\sigma(w_o L_0^T + b_o)}{\partial w_o L_0^T + b_o}\,L_i$$

…

Computing Gradients in Neural Networks

- Neural Networks stack functions on top of each other
- use chain rule to compute gradients for parameter θ_i on the i-th layer:
- for 2 layers: $\frac{\partial f_1(f_0(x,\theta_0),\theta_1)}{\partial \theta_0} = \frac{\partial f_1(f_0(x,\theta_0),\theta_1)}{\partial f_0(x,\theta_0)} \frac{\partial f_0(x,\theta_0)}{\partial \theta_0}$

- for *d* layers:

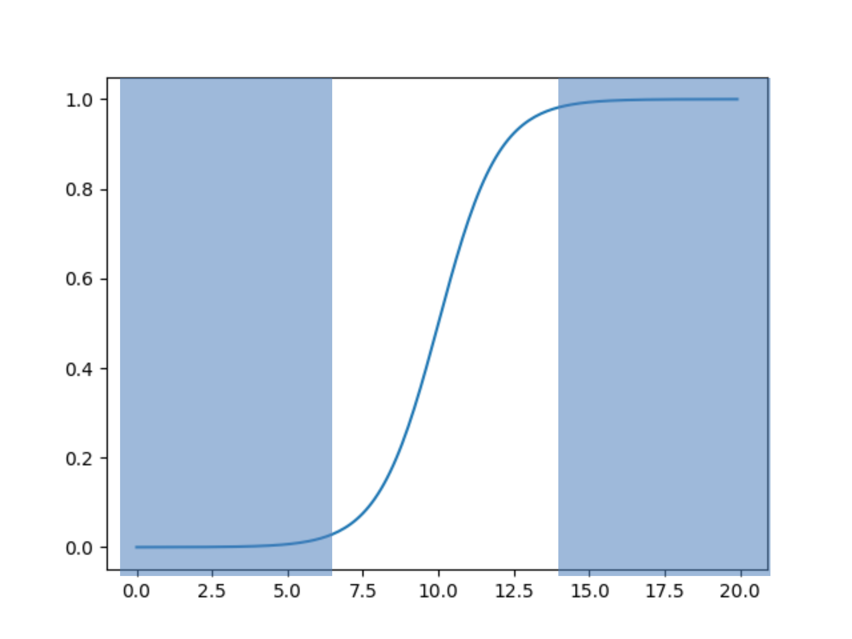
$$\frac{\partial f_d(f_{d-1}..f_0(x,\theta_0),)}{\partial x} = \frac{\partial f_d(f_{d-1}..f_0(x,\theta_0))}{\partial f_{d-1}..f_0(x,\theta_0)} \frac{\partial f_{d-1}(f_{d-2}..f_0(x,\theta_0))}{\partial f_{d-2}..f_0(x,\theta_0)} \dots \frac{\partial f_0(x,\theta_0)}{\partial \theta_0}$$

note:

- for any θ_i the levels i>0 , $f_{i-1}(f_{i-2}(..),\theta_{i-1})$ yield a constant input
- intermediate results can be stored in the forward path and used for the gradients of the higher levels.
- gradients for higher levels are multiplied:
  ⇒ if gradients at higher layers are all < 1, the gradient might vanish
  ⇒ if gradients at higher layers are all > 1, the gradient might explode
  (this will become a problem in recurrent architectures)

Weight Initialization in Neural Networks

- multiplying gradients yields a problem for consistently small or large values
- the value of the gradient depends on the current parameters
- if parameters are not properly initialised,
  GD might start with unfavourable gradient sizes.



shaded blue areas
yield small gradients
⇒ slow convergence

Glorot Initialization

- weights are drawn iid from a distribution with $0\ mean$.
- non-linearity *f* is odd: f(-x) = -f(x). (e.g. tanh)
- What is a good variance/range?
  - We want to achieve that the variance of signals z_i/gradients is the same between layers *i* where W,b represent parameter of a linear layer :
    - $VAR\left[z_k^{i+1}\right] = Var\left[\sigma(W_{\circ,k}^{i} + \mathrm{b}_\mathrm{k}^{i})\right]$ (forward)
    - $VAR\left[\frac{\partial L}{\partial s_k^1}\right] = Var\left[\sum_{j=1}^{n^{i+2}} W_{j,k}^{i+1} \frac{\partial L}{\partial s_k^{j+1}} f'(s_k^i)\right]$ (backward)
  - this can be achieved by: $VAR\left[W^i\right] = \frac{1}{n^i}\ (forward)$

$$VAR\left[W^i\right] = \frac{1}{n^{i+1}}\ (backward)$$

Understanding the difficulty of training deep feedforward neural networks, Glorot *et al.* (2010)

Weight Initialization in Practice

odd non-linearities:

- for normal distribution:

$$W^i \sim N\left(0, \frac{2}{n^i + n^{i+1}}\right)$$

- for uniform distribution $U(-a, a)$:

$$a = \sqrt{\frac{6}{n^i + n^{i+1}}}$$

- Kaiming or He Initialization: (better for ReLU)

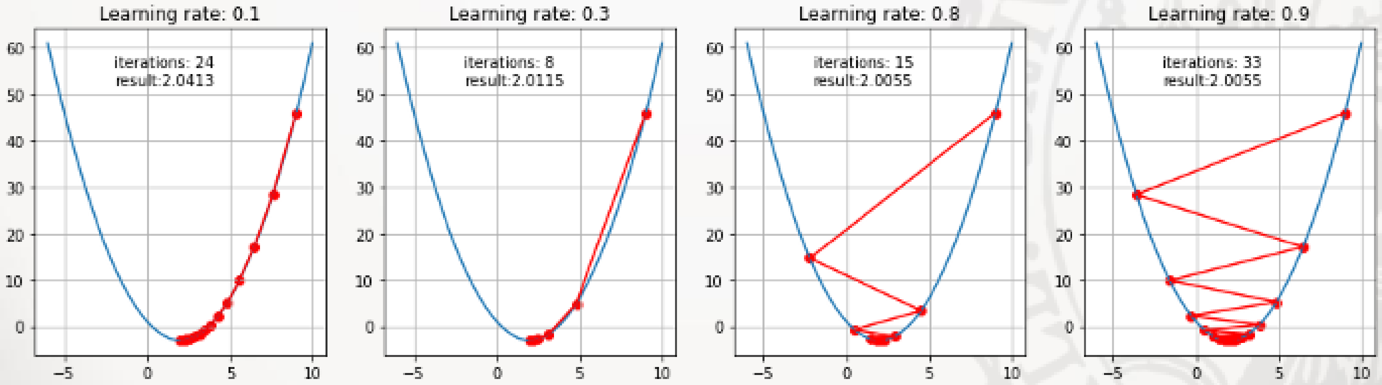
$$W^i \sim N\left(0, \frac{2}{\sqrt{n^i}}\right)$$

note:

- Frameworks like pytorch or tensorflow use these methods as default.
- Input *x* should be normalized between -1 and 1 to keep

Learning Rates and starting point

- learning rate α:
  - if α too big, $f(x_{t+1}) > f(x)$. GD might not converge.
  - if α too small, convergence is too slow to reach local minimum within given time frame.
- initialization x_0:
  - might decide whether GD converges for non-convex functions.
  - the closer x_0 is to the minimum, the faster GD converges.
- 2nd order optimisation like Newton-Raphson choose the step size w.r.t. root of the derivative and avoid these problems. But computing the Hessian and its root is quadratic in the number of parameters.



https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21

Variants of Gradient Descent

- **Stochastic Gradient Descent**: Compute loss on a single instance
- **Gradient Descent:** Compute Average Loss on the complete dataset D.
- **Batch Stochastic Gradient Descent**: Compute loss on a subset of D (batch) and average objective over this batch.

- gradient of the average is the average of the gradients:

$$\frac{\partial \frac{1}{n}\sum_{(x,y)\in D} J(x,y,\theta)}{\partial\theta} = \frac{\partial \frac{1}{n} J(x_0,y_0,\theta)+..+\partial\frac{1}{n}J(x_n,y_n,\theta)}{\partial\theta}$$

$$= \frac{1}{n}\frac{\partial J(x_0,y_0,\theta)}{\partial\theta}+..+\frac{1}{n}\frac{\partial J(x_n,y_n,\theta)}{\partial\theta} = \frac{1}{n}\sum_{i=0}^{n}\frac{\partial f(x_i,\theta)}{\partial\theta}$$

Learning rates in Deep Learning

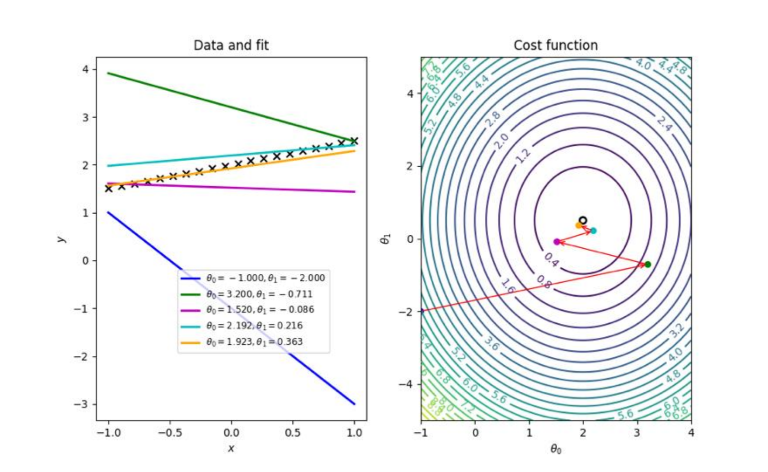
- optimal learning rate depends on the steepness of the derivative
- deep NNs have millions of parameters: derivatives are high-dimensional and steepness depends on the parameters
- improved optimizers for GD adjust the learning rate:
  - use individual learning rates for different parameters/layers
  - use momentum to stabilize the direction of the training (gradients are floating averages over the last couple of steps)
- most common optimizers today
  - AdaGrad
  - RMSPROP
  - ADAM

GD with Momentum

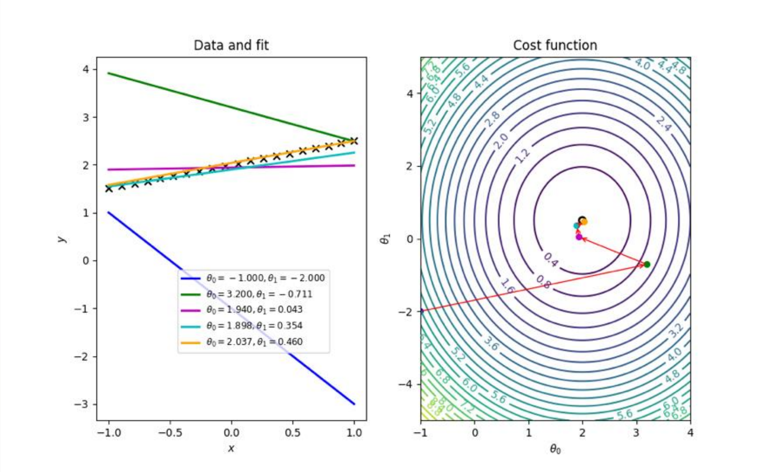
For too large learning rates α, GD starts to oscillate.

Idea: Update gradients with a floating average (momentum) to keep direction more stable. (γ: momentum, α: learning rate)

$$v_i^t = \gamma v_i^{t-1} + \alpha \frac{\partial f(\theta_i^t)}{\partial \theta_i^t}, \qquad \theta_i^{t+1} = \theta_i^t - v_i^t$$



γ:0, α:1.4



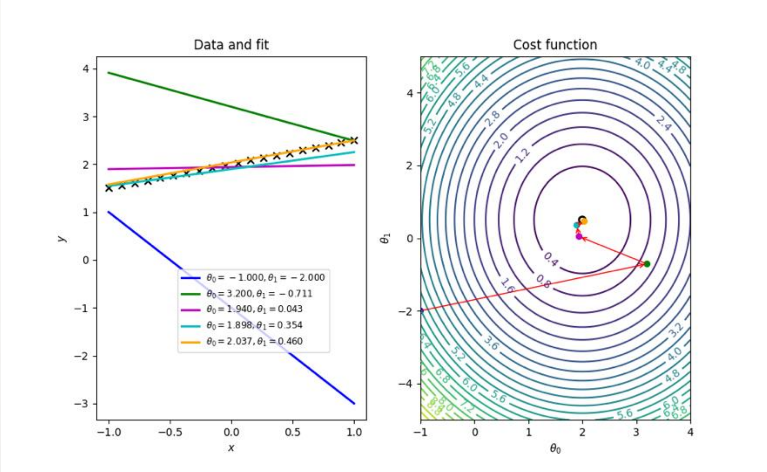
γ:0.1, α:1.4

https://distill.pub/2017/momentum/

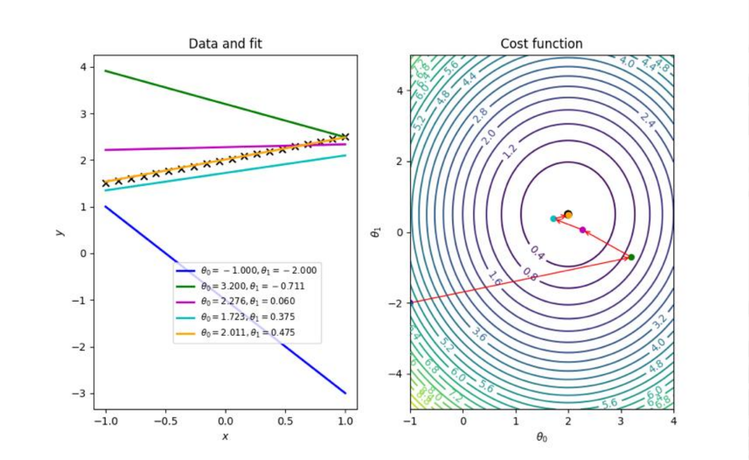
Gradient Descent with Nesterov Momentum

- gradient is computed after the update
- anticipates the update from momentum in the gradient

$$v_i^t = \gamma v_i^{t-1} + \alpha \frac{\partial f(\theta_i^t + \gamma v_i^{t-1})}{\partial \theta_i^t}, \qquad \theta_i^{t+1} = \theta_i^t - v_i^t$$



ordinary momentum γ:0.1, α:1.4



Nesterov momentum γ:0.04, α:1.4

AdaGrad

- introduces dedicated learning rates for each parameter
- learning rates are monotonically decreased
- computes the square sum of gradients for all parameters (large sum a lot of change happened ⇒ decrease LR)
- suitable for sparse data (no gradients of 0 inputs)

$$s_i^t = \sum_{\tau=1}^{t} \left(\frac{\partial f(\theta_i^\tau)}{\partial \theta_i^\tau}\right)^2$$

$$\theta_i^{t+1} = \theta_i^t - \frac{\alpha}{\sqrt{s_i^t + \epsilon}} \frac{\partial f(\theta_i^t)}{\partial \theta_i^t}$$

RMSProp

- individual learning rates with floating average

$$s_i^t = \gamma s_i^{t-1} + (1-\gamma)\left(\frac{\partial f(\theta_i^t)}{\partial \theta_i^t}\right)^2$$

$$\theta_i^{t+1} = \theta_i^t - \frac{\alpha}{\sqrt{s_i^t + \epsilon}} \frac{\partial f(\theta_i^t)}{\partial \theta_i^t}$$

Adam (ADAptive Moment estimation)

- Uses momentum and RMSProp in combination
- currently, the gold standard

$$v_i^{(t)} = \beta_1 v_i^{(t-1)} + (1 - \beta_1) \frac{\partial f(\theta_i^{(t)})}{\partial \theta_i^{(t)}}$$

$$s_i^{(t)} = \beta_2 s_i^{(t-1)} + (1 - \beta_2) \left(\frac{\partial f\left(\theta_i^{(t)}\right)}{\partial \theta_i^{(t)}} \right)^2$$

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\alpha}{\sqrt{\frac{s_i^{(t)}}{1-\beta_2^t}}+\epsilon} \frac{v_i^{(t)}}{1-\beta_1^t}$$

note: the t-th power of β is used for bias correction

Newer Adam approaches

- **Adan[1]** extends Adam with Nestorov Momentum.

- **AdamW[2]** is a stochastic optimization method that modifies the typical implementation of weight decay in Adam, by decoupling weight decay from the gradient update.

- **Rectified Adam[3]**, or **RAdam**, is a variant of the Adam stochastic optimizer that introduces a term to rectify the variance of the adaptive learning rate.

[1] https://arxiv.org/abs/2208.06677v4
[2] https://arxiv.org/abs/1711.05101v3
[3] https://arxiv.org/abs/1908.03265v4

Input Normalization

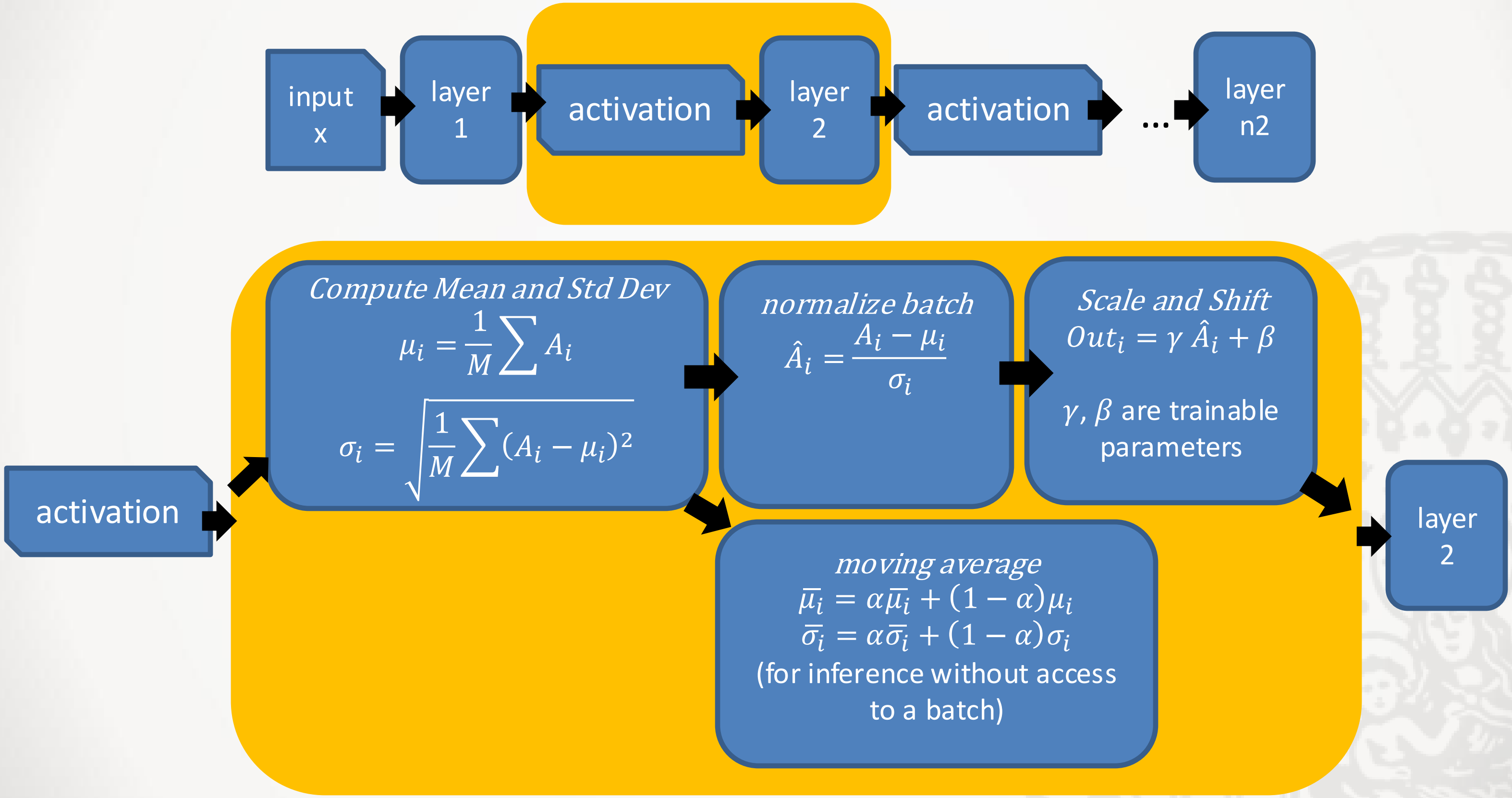
- inputs to the NN should be normalised to make all input features share the same scale (e.g. $\bar{x} = \frac{x-\mu}{\sigma}$)
- if raw features have varying scales, the gradients vary in scale as well. Updates progress at different speeds for varying parameters.
- normalisation counters this effect

Note:

- individual learning rates could be used here as well, but in Adagrad, RMSProp etc. this is not directly considered
- Input Normalisation helps with the input layer, but this effect could happen in internal layers as well

Batch-Normalization

batch normalisation layers adjust feature scaling for inner layers:



input x → layer 1 → activation → layer 2 → activation → … → layer n2

activation

Compute Mean and Std Dev

$$\mu_i = \frac{1}{M}\sum A_i$$

$$\sigma_i = \sqrt{\frac{1}{M}\sum (A_i - \mu_i)^2}$$

normalize batch

$$\hat{A}_i = \frac{A_i - \mu_i}{\sigma_i}$$

Scale and Shift

$$Out_i = \gamma\, \hat{A}_i + \beta$$

γ, β are trainable parameters

moving average

$$\bar{\mu}_i = \alpha\bar{\mu}_i + (1-\alpha)\mu_i$$

$$\bar{\sigma}_i = \alpha\bar{\sigma}_i + (1-\alpha)\sigma_i$$

(for inference without access to a batch)

layer 2

batch normalisation can be applied before or **after** activation.

Overfitting and Deep Architectures

- NN are universal function approximators with large amounts of parameters ⇒ model bias is pretty low
- generally, low-bias models allow to exactly fit the test data

So why do these models generalise to new data at all:

- SGD training helps:
  - gradient points to the optimum of the batch, not the complete data set, but the batch always changes
  - SGD takes a batch only for one step but does not converge to the minimum for one batch
- Usually, NNs are trained on very large data sets
- Deep architectures often contain layers for removing irrelevant information. (information bottlenecks, drop out, pooling etc.)

Note: Despite all, overfitting still might be a problem in NN.

Regularization via Dropout

What is a dropout layer?

- training: with probability $\left(1 - p_{keep}\right)$ set an input value to zero⇒ part of the network before dropout remains unused
- inference: all weights are used but with $p_{keep}$to keep the input scale

Why does it work?

- a way to reduce overfitting would be to train an ensemble of models, which is too expensive for larger architectures.
- usually, deep NNs yield a lot of parameters which implies that the network in itself encodes redundant information
- dropout emulates an ensemble by dynamically selecting subnetworks during training

Early Stopping

- SGD is a stepwise optimisation, and initial weights usually don’t overfit but generalise well
- in general, learning general concepts which hold for the majority of data decreases the joint loss stronger than fitting parameters to decrease the loss for particular instances
- after the weights sufficiently represent the general concepts, fitting particular instances yields the majority of loss improvements

⇒ Early Stopping tries to end training at the sweet spot

Technically:

- extract a validation set to test performance independently
- if performance on the validation set consistently drops, stop training. (depending on the batch size, several tries to improve might be allowed)

Summary

- building blocks: linear layers, non-linearities and loss functions
- gradients and optmization
- Gradient Descent, batch GD, Stochastic GD
- Momentum, Adagrad, RMSProp and ADAM
- Weight Initialization, Input Scaling and Batch Normalisation
- Overfitting, Dropout and Early Stopping