PROBABILISTIC AND DIFFERENTIABLE PROGRAMMING

V2: Gradient Descent

Özgür L. Özçep

Universität zu Lübeck

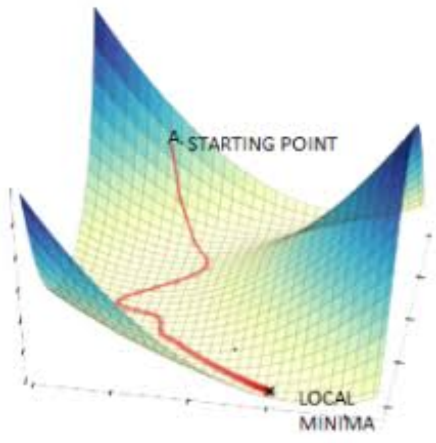
Institut für Informationssysteme

Agenda for today‘s lecture

Gradient descent (GD)

1. Differentiation

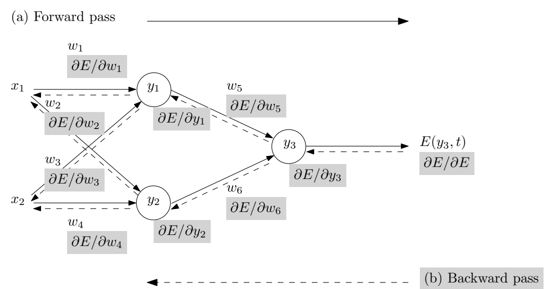
$$\frac{df}{dx}$$



2. Basic GD and variants

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla_\theta L$$

3. Backpropagation



The big idea: follow the gradient

- Fundamentally, we're interested in machines that we train by
  - optimising parameters
- How do we select those parameters?
- In deep learning/differentiable programming we typically define an objective function to optimize
  - *minimise (in case of error or loss say)* or
  - *maximise* with respect to those parameters
- We're looking for points at which the gradient of the objective function is zero w.r.t. the parameters

The big idea: follow the gradient

- Gradient based optimisation is a BIG field!
  - First order methods, second order methods, subgradient methods...
  - With deep learning we're primarily interested in first-order methods[1].
- Primarily using variants of gradient descent:
  - function $F(\boldsymbol{x})$ has *a (not necessarily unique or global)* minimum at a point $\boldsymbol{x} = \boldsymbol{a}$ where $\boldsymbol{a}$ is given by applying

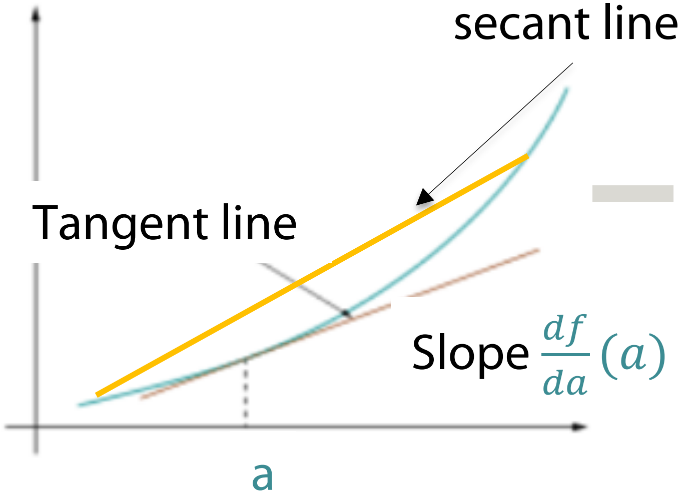
$$\boldsymbol{a}_{n+1} = \boldsymbol{a}_n - \alpha \nabla F(\boldsymbol{a}_n)$$

    until convergence

1) Second order gradient optimisers are potentially better, but for systems with many variables are currently impractical as they equire computing the Hessian.

DIFFERENTIATION

Gradient in one dimension



- Gradient of a straight line is $\Delta y / \Delta x$
- For arbitrary real-valued function $f(x)$

  approximate the derivative, $\frac{df}{dx}(a)$ using the gradient of the secant line trough $(a, f(a))$ and $(a+h, f(a+h))$ for small h

$$f'(a) = \frac{df}{dx}(a) \approx \frac{\Delta f}{\Delta a} \approx \frac{f(a+h)-f(a)}{h}$$ (Newton‘s difference quotient)

$$\frac{df}{dx}(a) = \lim_{h \to 0} \frac{f(a+h)-f(a)}{h}$$ (Derivate of f at a)

Example: Derivative of a quadratic function

$$y = x^2$$

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{(x+h)^2 - x^2}{h}$$

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{x^2 + 2hx + h^2 - x^2}{h}$$

$$\frac{dy}{dx} = \lim_{h \to 0} \frac{2hx + h^2}{h}$$

$$\frac{dy}{dx} = \lim_{h \to 0} 2x + h$$

$$\frac{dy}{dx} = 2x$$

Derivatives of „deeper“ functions

- Deep learning is all about optimising deeper functions: functions that are compositions of other functions, e.g.

$$h = (f \circ g)(x) = f(g(x))$$

- Derivative can be calculated by chain rule

Chain rule (1-dim)

$$\frac{dh}{dx} = \frac{df}{dg}\frac{dg}{dx} \qquad \text{for} \quad h(x) = f(g(x))$$

Example for chain rule

$$h(x) = x^4 = (x^2)^2 \; = f\big(g(x)\big)$$
$$\frac{dh}{dx} = 2 \; \cdot x^2 \cdot 2x = 4x^3$$

You may verify this also directly

$$\frac{dh}{dx} = \lim_{h \to 0} \frac{(x+h)^4 - x^4}{h}$$
$$\frac{dh}{dx} = \lim_{h \to 0} \frac{h^4 + 4h^3x + 6h^2x^2 + 4hx^3 + x^4 \; - x^4}{h}$$
$$\frac{dh}{dx} = \lim_{h \to 0} h^3 + 4h^2x + 6hx^2 + 4x^3 \; = 4x^3$$

Generalization: Vector functions $\boldsymbol{y}(t)$

- Split into its constituent coordinate functions:

$$\boldsymbol{y}(t) \;=\; (y_1(t), \ldots, y_n(t))$$

- Derivative is a vector (the tangent vector),

$$\boldsymbol{y}'(t) \;=\; ({y_1}'(t), \ldots, {y_n}'(t))$$

  which consists of the derivatives of the coordinate functions.

- Equivalently

$$\boldsymbol{y}'(t) = \lim_{h \to 0} \frac{\boldsymbol{y}(t+h) - \boldsymbol{y}(t)}{h}$$

  (if the limit exists)

Differentiation with multiple variables

$f(x,y) = x^2 + xy + y^2$

$\frac{\partial f}{\partial x} = 2x + y$

$\frac{\partial f}{\partial y} = x + 2y$

Partial derivative of $f(x_1, \ldots x_n): \mathbb{R}^n \to \mathbb{R}$ w.r.t. x_i at $\boldsymbol{a} = (a_1, \ldots, a_n)$

$$\frac{\partial f}{\partial x_i}(\boldsymbol{a}) = \lim_{h \to 0} \frac{f(a_1,\ldots,a_i+h,\ldots,a_n) - f(\boldsymbol{a})}{h}$$

Gradient of $f(x_1, \ldots x_n): \mathbb{R}^n \to \mathbb{R}$ at $\boldsymbol{a} = (a_1, \ldots, a_n)$

$$\nabla f(\boldsymbol{a}) = \left(\frac{\partial f}{\partial x_1}(\boldsymbol{a}), \ldots, \frac{\partial f}{\partial x_n}(\boldsymbol{a})\right)$$

Jacobian of $\boldsymbol{f}(x_1, \ldots x_n): \mathbb{R}^n \to \mathbb{R}^m$ at $\boldsymbol{a} = (a_1, \ldots, a_n)$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}(\boldsymbol{a}) = \begin{pmatrix} \nabla f_1(\boldsymbol{a}) \\ \vdots \\ \nabla f_m(\boldsymbol{a}) \end{pmatrix} = \left(\frac{\partial f_i}{\partial x_j}(\boldsymbol{a})\right)_{1 \le i \le m; 1 \le j \le n} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\boldsymbol{a}) & \cdots & \frac{\partial f_1}{\partial x_n}(\boldsymbol{a}) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(\boldsymbol{a}) & \cdots & \frac{\partial f_m}{\partial x_n}(\boldsymbol{a}) \end{pmatrix}$$

Linear algebra reminder

- Given vectors $\boldsymbol{x} = (x_1, \dots, x_n)$ and $\boldsymbol{y} = (y_1, \dots, y_n)$
- Scalar product : $\boldsymbol{x} \cdot \boldsymbol{y} = \sum_{i=1}^{n} x_i y_i$

- Jacobian is given as an $m \times n$ matrix
  $\mathrm{A} = \left(\mathrm{a}_{ij}\right)_{1 \le i \le m, 1 \le j \le n}$ (m rows, n columns)
- An $m \times n$ matrix A defines a linear mapping
  $\mathrm{A}\colon \mathbb{R}^n \to \mathbb{R}^m$ via

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} \mapsto A\,x = \begin{pmatrix} \sum_{i=1}^{n} a_{1,i} x_i \\ \sum_{i=1}^{n} a_{2,i} x_i \\ \vdots \\ \sum_{i=1}^{n} a_{m,i} x_i \end{pmatrix}$$

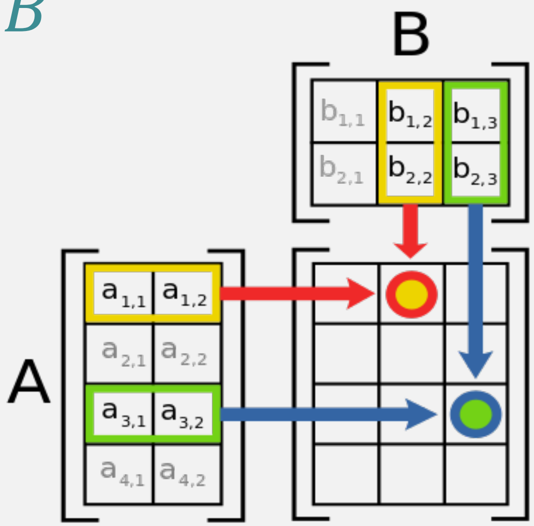
(Linearity: $\mathrm{A}(\lambda \mathbf{x} + \mu \boldsymbol{y}) = \lambda A x + \mu A y$ where x,y vectors and λ, μ scalars

Linear algebra reminder

Matrix multiplication $C = A\,B$ for

$m \times n$ matrix A and $\mathrm{n} \times p$ matrix B

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$



Transposed matrix $A^{\top}$: change columns and rows

Gradients in Machine Learning

The kinds of functions (and programs) that are usually optimized in ML have following properties:

- They are scalar-valued
- There are multiple losses, but ultimately we can just consider optimising with respect to the sum of the losses.
- They involve multiple variables, which are often wrapped up in the form of vectors or matrices, and more generally tensors.

How will we find the gradients of these?

The chain rule for vectors

Given functions f, g with

- $\mathbb{R}^m \xrightarrow{g} \mathbb{R}^n \xrightarrow{f} \mathbb{R}$
- $\boldsymbol{x} \mapsto \boldsymbol{y} = g(\boldsymbol{x}) \mapsto z = f(\boldsymbol{y})$

the chain rule gives the partial derivatives

$$\frac{\partial z}{\partial \boldsymbol{x}} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

(in short form: $\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z$

where $\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)$ is the n x m Jacobian matrix of g)

Chain rule for Tensors (Informal)

- Tensors (as understood in the ML literature) generalize vectors (1D-tensors) and matrices (2D-tensors)
  - 3D-tensor: Layer of matrices
  - nD-tensor $A_{i_1 \dots i_n}$ is indexed by n-tuples $(i_1 \; \dots \, i_n)$
    - Needed e.g. to model layers of convolution matrices etc.
- Gradients of tensors by
  - flattening them into vectors
  - computing the vector-valued gradient
  - then reshaping the gradient back into a tensor.
- This is just multiplying Jacobians by gradients again

The chain rule für tensors (formally)

- Aim: Calculate: $\nabla_{\boldsymbol{X}} z$ for scalar z and tensor $\boldsymbol{X}$
  - Indices into $\boldsymbol{X}$ have multiple coordinates, but we can generalise by using a single variable i to represent the complete tuple of indices.
  - For all index tuples i:
    $$(\nabla_{\boldsymbol{X}} z)_i = \frac{\partial z}{\partial X_i}$$

For $\boldsymbol{Y} = g(\boldsymbol{X})$ and $z = f(\boldsymbol{Y})$

$$\nabla_{\boldsymbol{X}} z = \sum_j (\nabla_{\boldsymbol{X}} Y_j) \frac{\partial z}{\partial Y_j}$$

Example for tensor chain rule

- Let $\boldsymbol{D} = \boldsymbol{X}\boldsymbol{W}$ where the rows of $\boldsymbol{X} \in \mathbb{R}^{n \times m}$ contains some fixed features, and $\boldsymbol{W} \in \mathbb{R}^{m \times h}$ is a matrix of weights.
- Also let $L = f(\boldsymbol{D})$ be some scalar function of $\boldsymbol{D}$ that we wish to minimise.
- What are the derivatives of L with respect to the weights $\boldsymbol{W}$?

- Start by considering a specific weight W_{uv}
- $\frac{\partial L}{\partial W_{uv}} = \sum_{i,j} \frac{\partial L}{\partial D_{ij}} \frac{\partial D_{ij}}{\partial W_{uv}}$ (by chain rule)
- $\frac{\partial D_{ij}}{\partial W_{uv}} = 0$ iff $j \neq v$ because D_{ij} is the scalar product of row i of $\boldsymbol{X}$ and column j of $\boldsymbol{W}$.
- Therefore: $\sum_{i,j} \frac{\partial L}{\partial D_{ij}} \frac{\partial D_{ij}}{\partial W_{uv}} = \sum_{i} \frac{\partial L}{\partial D_{iv}} \frac{\partial D_{iv}}{\partial W_{uv}}$
- What is $\frac{\partial D_{iv}}{\partial W_{uv}}$?
  - $D_{iv} = \sum_{1 \le k \le m} X_{ik} W_{kv}$
  - $\frac{\partial D_{iv}}{\partial W_{uv}} = \frac{\partial}{\partial W_{uv}} \sum_{1 \le k \le q} X_{ik} W_{kv} = \sum_{1 \le k \le m} \frac{\partial}{\partial W_{uv}} X_{ik} W_{kv} = X_{iu}$

- Putting every together, we have: $\frac{\partial L}{\partial W_{uv}} = \sum_i \frac{\partial L}{\partial D_{ij}} X_{iu}$
- $= \sum_i X_{iu} \frac{\partial L}{\partial D_{ij}} = \sum_i X_{ui}^\top \frac{\partial L}{\partial D_{ij}}$
- Doing this for arbitrary W_{iu} leads to
- $\frac{\partial L}{\partial \boldsymbol{W}} = \boldsymbol{X}^\top \frac{\partial L}{\partial \boldsymbol{D}}$

VANILLA GRADIENT DESCENT, VARIANTS AND BEYOND

Vanilla Gradient Descent (VGD)

- Given: loss function l, dataset D, and model g, parameters θ; number of passes (epochs) over the data, learning rate η
- Total loss: $L = -\sum_{(x,y)\in D} l(g(x,\theta),y)$

VGD: $\theta_{t+1} \leftarrow \theta_t - \eta \nabla_\theta L$

\+ Good statistical properties (very low variance)
\- Very data inefficient (particularly when data has many similarities)
\- Doesn't scale to infinite data (online learning)

Problems of *GD
\- ...

Why the hell follow the gradient?

- Make shift in parameter space $\Delta\theta = (\Delta\theta_1, \Delta\theta_2)$
- Calculus says : $\Delta L \approx \frac{\partial L}{\partial \theta_1}\Delta\theta_1 + \frac{\partial L}{\partial \theta_2}\Delta\theta_2 = \nabla \mathrm{L} \Delta\boldsymbol{\theta}$
- Loss should decrease: $\Delta L \leq 0$
- Try: $\Delta\theta = -\eta \nabla L$
- Helps, because $\Delta L \approx -\eta\, \nabla L \cdot \nabla L = -\eta\, \left||\nabla L|\right|^2$

  and $\left||\nabla L|\right|^2 \geq 0$

Linear algebra reminder:

- Norm of v: $\left||v|\right| = v \cdot v$ (for scalar product $\cdot$)

Let‘s talk abut loss – only roughly for now

- Gradient descent algorithms depend on loss function l
- For now think of loss function l as mean squared error l_{MSE}
- We will see other ones and their interplay with activation functions in the next lecture

Mean squared error on one singel training example

$$
\begin{array}{llcl}
l_{MSE}: & \mathbb{R}^n \times \mathbb{R}^n & \overset{\boldsymbol{l}}{\rightarrow} & \mathbb{R} \\
 & (\hat{y}, y) & \mapsto & ||\hat{y} - y||^2
\end{array}
$$

Stochastic Gradient Descent (SGD)

- Given: loss function l, dataset D, model g, parameters θ, number of epochs, learning rate η

SGD : $$\theta_{t+1} \leftarrow \theta_t - \eta \nabla_\theta\, l(g(\boldsymbol{x}, \theta_t), y)$$

+ Faster than VGD
+ Online learning
- Poor statistical properties (high fluctuation)
- computational inefficiency

Problems of *GD
- ...

Mini-Batch SGD (MGD)

- Given: mini-batch size m (common: 50-256), loss function l, dataset D, model g, parameters θ, number of epochs , learning rate η
- Batch loss: $L_{b(t)} = \sum_{(x,y) \in b(t)} l(g(x,\theta), y)$

  where $d(t)$, a subset of D of cardinality m.

MSGD : $\theta_{t+1} \leftarrow \theta_t - \eta \nabla_{\theta_t} L_{d(t)}$

+ reduces the parameter-updates' variance
+ stable convergencevery
+ computational efficiency

Problems of *GD

1. How to choose rate
2. No learning rate schedules
3. Trapping in local minima
4. Inefficient for sparse data set

Problem 1: Choosing the learning rate η [1)]

- Choice of learning rate is extremely important
- But we have to reason about the ‘loss landscape’
  - Types of cost functions (see next lecture)
  - Most convergence analysis of optimisation algorithms assumes a convex loss landscape
    - Easy to reason about
    - (S)GD converges to optimal solution for a variety of ηs
    - Insights into potential problems in the non-convex case
  - Deep Learning is highly non-convex
  - Many local minima; Plateaus; Saddle points; Symmetries (permutation, etc)

1) One form of hyper-parameter magic

„Beyond": Accelerated Gradient Methods

- Accelerated gradient methods use a *leaky* average of the gradient, rather than the instantaneous gradient estimate at each time step

- A physical analogy would be one of the momentum a ball picks up rolling down a hill...

- Helps addressing the *GD problems

Mini-Batch SGD with Momentum (MSGDM)

- Given: momentum parameter β (0,9 is good choice), batch size m, batch loss $L_{d(t)}$, number of epochs, learning rate η

MSGDM : update θ by accumulated velocity

$$v_{t+1} \leftarrow \beta v_t + \nabla_\theta L_{d(t)}$$

$$\theta_{t+1} \leftarrow \theta_t - \eta v_{t+1}$$

\+ The momentum method allows to accumulate velocity in directions of low curvature that persist across multiple iterations

\+ This leads to accelerated progress in low curvature directions compared to gradient descent

Problem 2: Scheduling learning rates

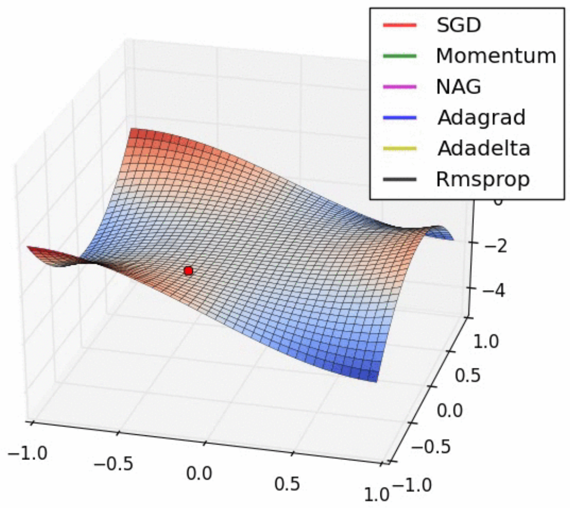
- In practice you want to decay your learning rate over time
- Smaller steps will help you get closer to the minima
- But don't do it to early, else you might get stuck
  Something of an art form!
- 'Grad Student Descent' or GDGS ('Gradient Descent by Grad Student')
- Tackling Plateaus (Common Heuristic approach)
  - if the loss hasn't improved (within some tolerance) for *k* epochs then drop the lr by a factor of 10

Problem 3: Stucking into local minima

- Cycle the learning rate up and down (possibly annealed), with a different lr on each batch
- See L. N. Smith. Cyclical Learning Rates for Training Neural Networks. arXiv e-prints, page https://arxiv.org/abs/1506.01186 , June 2015.

SOTA: More advanced optimisers

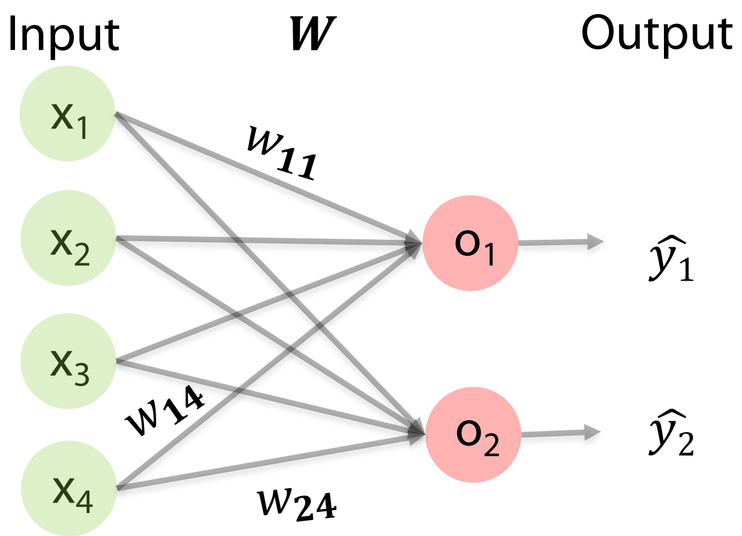
- Here only name dropping and some fancy gif from here
  - Adagrad (dynamic decrease, second moment used)
  - RMSProp (decouple learning rate from gradient)
  - Adam (BestOf(RMSProp,MSDGM))



- J. Hare says:
  - If you're in a hurry to get results use Adam
  - If you have time (or a Grad Student at hand), then use SGD (with momentum) and work on tuning the learning rate
  - If you're implementing something from a paper, then follow what they did!

BACKPROPAGATION

Network view of single function[1)]



Network model

$\boldsymbol{b}$: Bias vector (b_1, b_2)

$\boldsymbol{W}$ = weight matrix $(w)_{1\le i\le 2; 1\le j\le 4}$:

z: $\boldsymbol{Wx} + \boldsymbol{b}$ linear output

$\boldsymbol{\sigma}$: activation function

$$\mathbb{R}^4 \xrightarrow{\boldsymbol{g}} \mathbb{R}^2$$

$$\boldsymbol{x} \mapsto \hat{\boldsymbol{y}} = \boldsymbol{g} = (\boldsymbol{g_1}(\boldsymbol{x}), \boldsymbol{g_2}(\boldsymbol{x}))$$

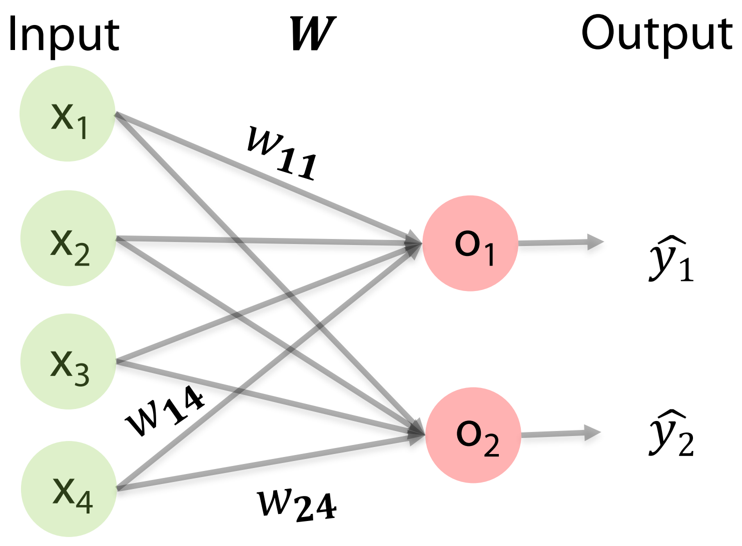
Vector-valued function in four arguments

$$\hat{\boldsymbol{y}} = \boldsymbol{g}\,(\boldsymbol{x}\,;\boldsymbol{W},\boldsymbol{b}) = \boldsymbol{\sigma}\,(\boldsymbol{Wx} + \boldsymbol{b}) = \sigma(\boldsymbol{z})$$

Decomposition into linear and activation part

1) You may find this also under the term perceptron in the literature

Network view of single function



Example linear output

$$\mathbf{W} = \begin{pmatrix} 1 & 2 & -1 & -2 \\ 3 & 4 & -3 & 4 \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \quad \boldsymbol{b} = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$

$$\boldsymbol{Wx} = \begin{pmatrix} 1 \cdot 1 + 2 \cdot 2 + -1 \cdot 3 - 2 \cdot 4 \\ 3 \cdot 1 + 4 \cdot 2 + -3 \cdot 3 + 4 \cdot 4 \end{pmatrix} = \begin{pmatrix} -6 \\ 18 \end{pmatrix}$$

$$\boldsymbol{Wx} + \boldsymbol{b} = \begin{pmatrix} -6 \\ 18 \end{pmatrix} + \begin{pmatrix} 5 \\ 6 \end{pmatrix} = \begin{pmatrix} -1 \\ 24 \end{pmatrix}$$

$$\mathbb{R}^4 \xrightarrow{\boldsymbol{g}} \mathbb{R}^2$$

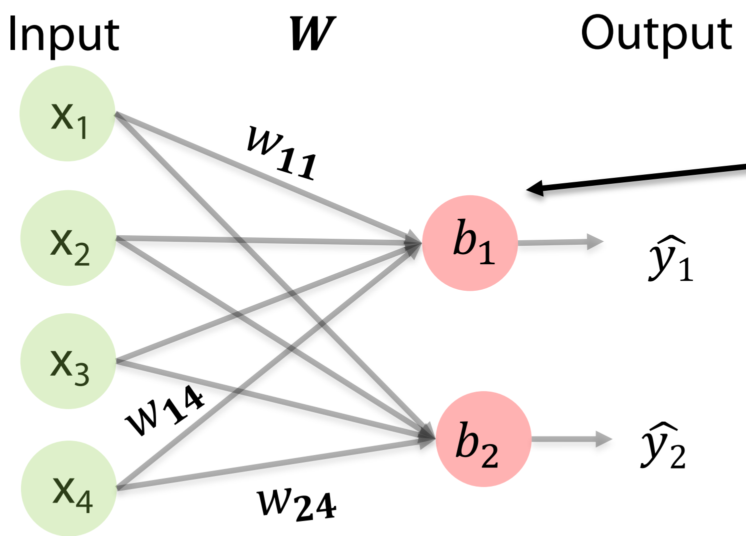
$$\boldsymbol{x} \mapsto \hat{\boldsymbol{y}} = \boldsymbol{g} = (\boldsymbol{g_1}(\boldsymbol{x}), \boldsymbol{g_2}(\boldsymbol{x}))$$

$$\hat{\boldsymbol{y}} = \boldsymbol{g}\,(\boldsymbol{x}\,;\boldsymbol{W},\boldsymbol{b}) = \boldsymbol{\sigma}\,(\boldsymbol{Wx} + \boldsymbol{b}) = \sigma(\boldsymbol{z})$$

Vector-valued function in four arguments

Decomposition into linear and activation part

Network view of single function



Sometimes just write biases into state

$\mathbb{R}^4 \xrightarrow{\boldsymbol{g}} \mathbb{R}^2$

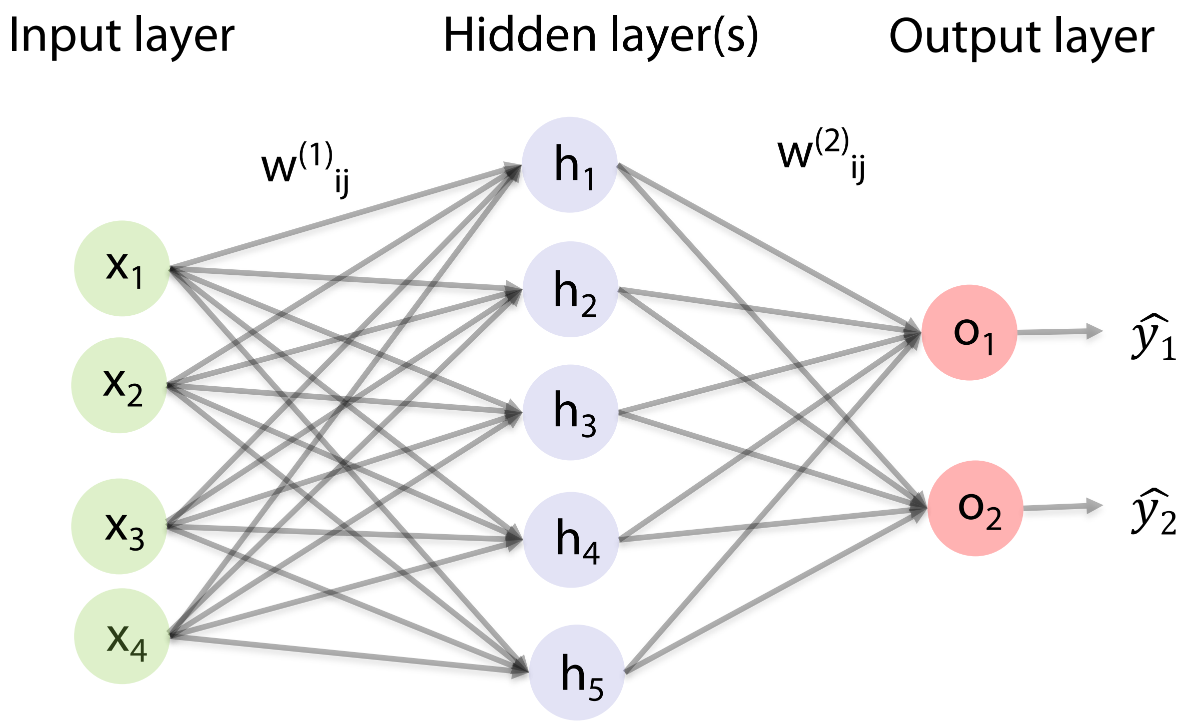
$\boldsymbol{x} \mapsto \hat{\boldsymbol{y}} = \boldsymbol{g} = (\boldsymbol{g_1}(\boldsymbol{x}), \boldsymbol{g_2}(\boldsymbol{x}))$

$\hat{\boldsymbol{y}} = \boldsymbol{g}\,(\boldsymbol{x}\,; \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{\sigma}\,(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\,) = \sigma(\boldsymbol{z})$

Vector-valued function in four arguments

Decomposition into linear and activation part

Network view of composed functions1)



$$\mathbb{R}^4 \xrightarrow{\boldsymbol{g}} \mathbb{R}^5 \xrightarrow{\boldsymbol{f}} \mathbb{R}^2$$

$$\boldsymbol{x} \mapsto \boldsymbol{h} \mapsto \hat{\boldsymbol{y}}$$

i:	layer i
$\boldsymbol{b}^i$:	Bias in i
$\mathbf{W}^{(i)}$:	weight matrix in i
$\boldsymbol{\sigma}_i$:	activation function in i

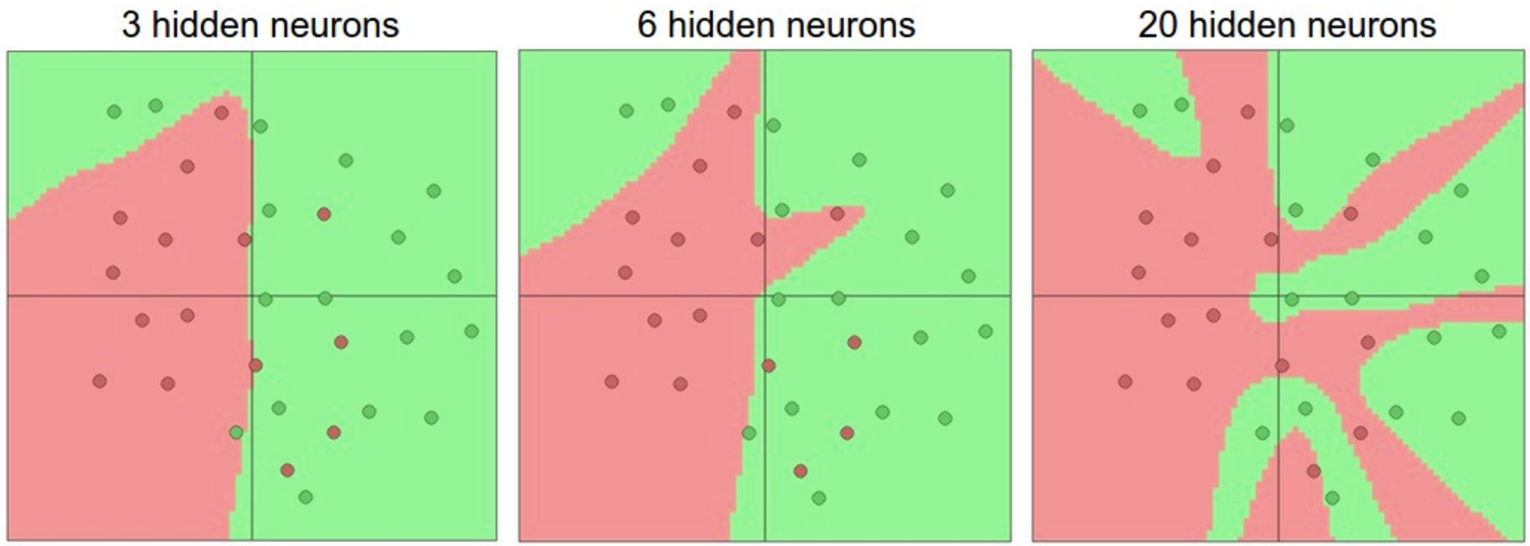
$\boldsymbol{a}^i$:	$\boldsymbol{\sigma_i}(\boldsymbol{W}^{(i)}\boldsymbol{a}^{i-1} + \boldsymbol{b}_i)$ activation in i
$\boldsymbol{z}^i$:	$\boldsymbol{W}^{(i)}\boldsymbol{a}^{i-1} + \boldsymbol{b_i}$ linear ouptut in layer i

$$\hat{\boldsymbol{y}} = \boldsymbol{f}\,(\boldsymbol{g}\,(\boldsymbol{x}\,;\ \boldsymbol{W}^{(1)}\ ,\boldsymbol{b}^{1}\,);\ \boldsymbol{W}^{(2)}\,,\boldsymbol{b}^{2}) \qquad = \boldsymbol{\sigma}_2\,(\boldsymbol{W}^{(2)}\,\boldsymbol{\sigma}\,1(\boldsymbol{W}^{(1)}\boldsymbol{x}\ + \boldsymbol{b}^{1}\,)\ +\ \boldsymbol{b}^{2})$$

1) You may find this also under the term multilayer perceptron in the literature

Activation functions

Non-linearities needed to learn complex (non-linear) representations of data, otherwise the network would be just a linear function $W_1 W_2 x = Wx$



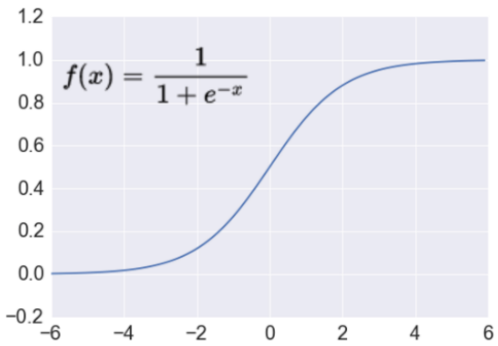
http://cs231n.github.io/assets/nn1/layer_sizes.jpeg

More layers and neurons can approximate more complex functions

Full list: https://en.wikipedia.org/wiki/Activation_function

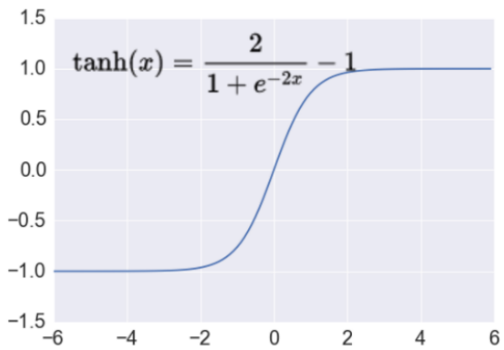
Activation functions

http://adilmoujahid.com/images/activation.png



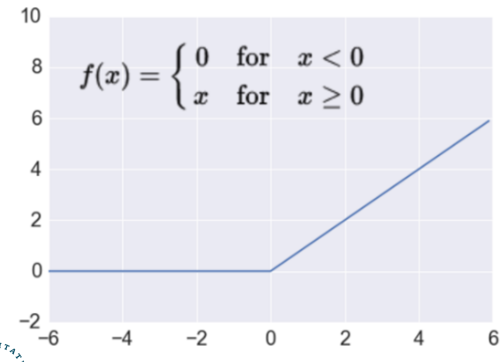
Sigmoid $\mathbb{R}^n \rightarrow [0,1]$

- Takes a real-valued number and "squashes" it into range between 0 and 1.
- Earliest used activation function (neuron)
- Leads to vanishing gradient problem



Tanh: $\mathbb{R}^n \rightarrow [-1,1]$

- Takes a real-valued number and "squashes" it into range between -1 and 1
- Same probem of vanishing gradient
- $\tanh(x) = 2sigm(2x) - 1$



Rectified Linear Unit ReLu: $\mathbb{R}^n \rightarrow \mathbb{R}^n_+$

- Takes a real-valued number and thresholds it at zero
- Used in Deep Learning
- No vanishing gradient
- But: it is not differentiable (need relaxation)
- Dying ReLU

Backprop: efficient implementation of gradient descent

V2

Sample labeled data (**batch**) → **Forward** it through the network, get predictions → **Back-propagate** the errors → **Update** the network weights →

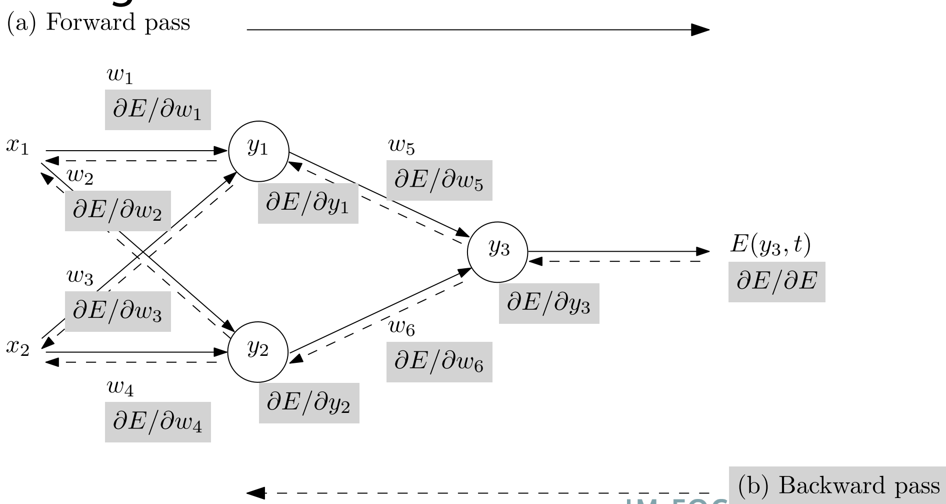
Backpropagation idea

- Generate **error signal** that measures difference between predictions and target values
- Use error signal to change the weights and get more accurate predictions backwards
- Underlying mathematics: chain rule

Chain rule (1-dim)

$$\frac{dh}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

(for $h(x) = f(g(x))$)



Computational graph perspective

Function f

$$f(x, y, z) = (x + y) \cdot z$$
$$= qz$$
$$\text{for } q = x + y$$

Partial Derivatives

$$\frac{\partial f}{\partial z} = q \qquad \frac{\partial f}{\partial q} = z$$
$$\frac{\partial q}{\partial x} = 1 \qquad \frac{\partial q}{\partial y} = 1$$

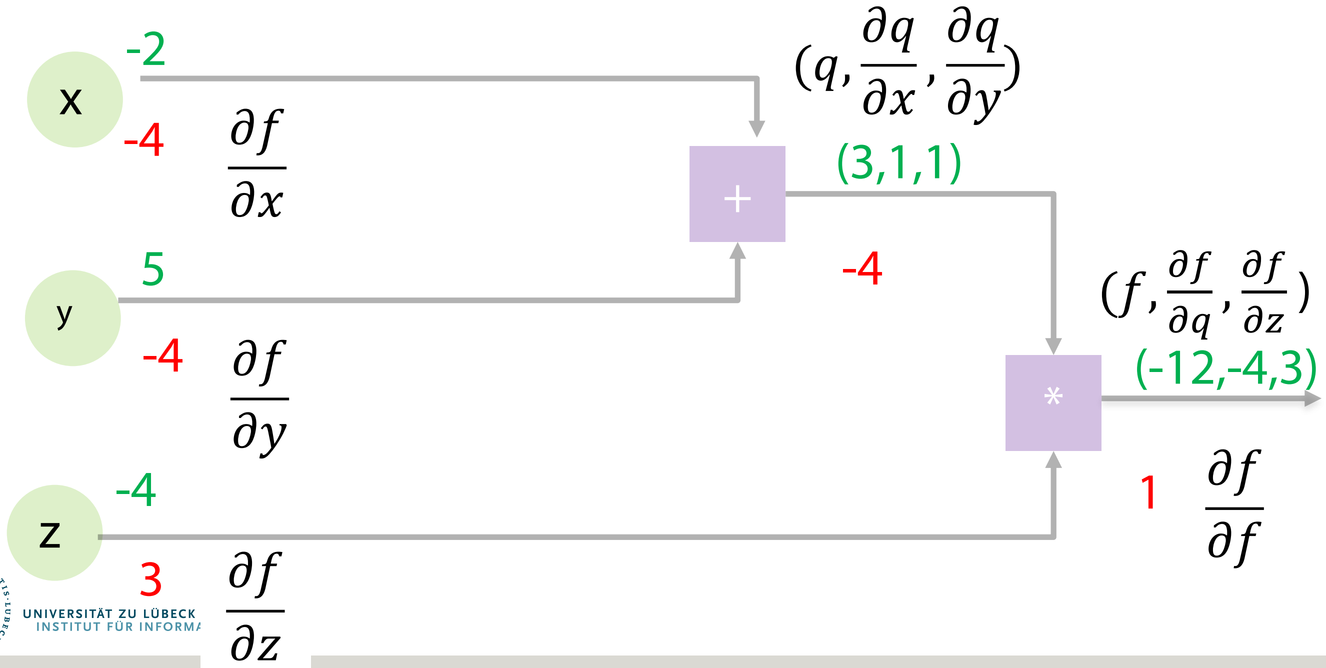
Chain rule applied

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z$$

Gradient

$$\nabla_{x,y,z} f = (z, z, q)$$

(In particular: $(\nabla_{x,y,z} f)(-2, 5, -4) = (-4, -4, 3)$)



Forward pass: function values and local gradients

Backward: chain rule

What this example tells us about backprop

- Every operation in the computational graph given its inputs can immediately compute two things:
  1. its output value and
  2. local gradients of its inputs

- The chain rule tells us literally that each operation should take its local gradients and multiply them by the gradient that flows backwards into it

- Backprop is an instance of ‘Reverse Mode Automatic Differentiation’

Backpropagation: requirements on cost (loss)

1. Cost C (we named it L before) on whole data is sum of costs on training instances
2. Cost is a function of the output $\hat{\boldsymbol{y}}$

- Backpropagation in the following described for cost on single training example
- With 1. assumption backpropagation can be combined with gradiend descent.
- In the following going to use Hadamard product $\odot$

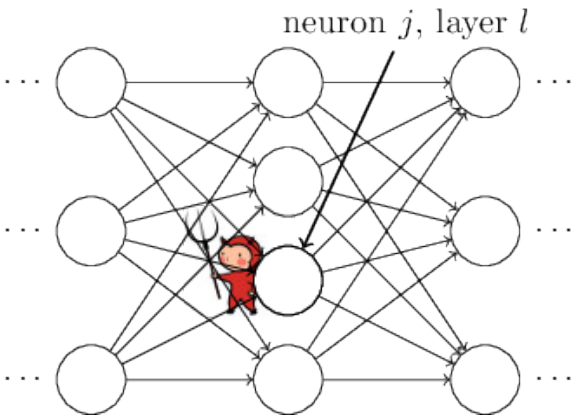
$$\begin{pmatrix} a \\ b \end{pmatrix} \odot \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ bd \end{pmatrix}$$

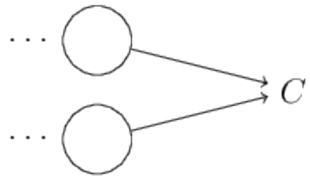
Propagation of errors

Backpropagation works on errors

(from these in the end one gets $\nabla_{\boldsymbol{W},\boldsymbol{b}}\, C$)

$$\delta_j^l := \frac{\partial C}{\partial z_j^l}$$ error in j^{th} component in layer l





Demon changes z_j^l to $z_j^l + \Delta z_j^l$

Resulting cost C changes by $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$

Backpropagation algorithm (on single instance)

1. Input: Initialize input vector $\boldsymbol{x} = \boldsymbol{a}^0$
2. Feedforward: For i = 1,2, … , M

$$\boldsymbol{z^i} = \boldsymbol{W}^{(i)}\boldsymbol{a^{i-1}} + \boldsymbol{b_i} \text{ and } \boldsymbol{a^i} = \boldsymbol{\sigma_i}(\boldsymbol{z^i})$$

3. Compute error on last layer

$$\boldsymbol{\delta}^M = \nabla_{\hat{\boldsymbol{y}}} C \odot \sigma'(\boldsymbol{z}^M) \quad \text{(BP1)}$$

4. Backpropagate error: For i = M-1, M-2, …,

$$\boldsymbol{\delta}^i = (\boldsymbol{w^{i+1}})^\top \boldsymbol{\delta}^{i+1} \odot \sigma'(\boldsymbol{z}^i) \quad \text{(BP2)}$$

5. Compute gradients

$$\frac{\partial C}{\partial w_{jk}^i} = a_k^{i-1}\delta_j^i \quad \text{and} \quad \frac{\partial C}{\partial b_j^i} = \delta_j^i \quad \text{(BP3/4)}$$

Proof of (BP1) in backprop

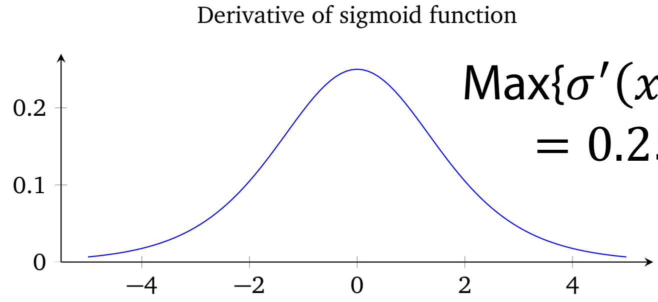
- $\delta_j^M = \frac{\partial C}{\partial z_j^M}$ (by definition)
- $\delta_j^M = \sum_k \frac{\partial C}{\partial a_k^M} \frac{\partial a_k^M}{\partial z_j^M}$ (chain rule; k over all components in output)
- $\delta_j^M = \frac{\partial C}{\partial a_j^M} \frac{\partial a_j^M}{\partial z_j^M}$ ($\frac{\partial a_k^M}{\partial z_j^M}$ vanishes wenn $k \neq j$)
- $\delta_j^M = \frac{\partial C}{\partial a_j^M} \sigma'(z_j^M)$ ($a_j^M = \sigma(z_j^M)$)

Backpropagation algorithm (within MSGD)

1. Input: mini-batch of m training examples x
2. For each training example set corresponding activation $\boldsymbol{a}^{\boldsymbol{x},\mathbf{1}}$ and do the following
   1) Feedforward: For i = 1,2, … , M
      $$\boldsymbol{z}^{\boldsymbol{x},\boldsymbol{i}} = \boldsymbol{W}^{(\boldsymbol{i})}\boldsymbol{a}^{\boldsymbol{x},\boldsymbol{i}-\mathbf{1}} + \boldsymbol{b}_{\boldsymbol{i}} \text{ and } \boldsymbol{a}^{\boldsymbol{x},\boldsymbol{i}} = \boldsymbol{\sigma}_{\boldsymbol{i}}\left(\boldsymbol{z}^{\boldsymbol{x},\boldsymbol{i}}\right)$$
   2) Compute error on last layer
      $$\boldsymbol{\delta}^{\boldsymbol{x},M} = \nabla_{\widehat{\boldsymbol{y}}} C_x \odot \sigma'(\boldsymbol{z}^{\boldsymbol{x},M})$$
   3) Backpropagate error: For i = M-1, M-2, …,
      $$\boldsymbol{\delta}^{i} = (\boldsymbol{w}^{\boldsymbol{i}+\mathbf{1}})^{\top}\boldsymbol{\delta}^{\boldsymbol{x},i+1} \odot \sigma'(\boldsymbol{z}^{\boldsymbol{x},i})$$
3. Gradient descent:
   $$\boldsymbol{w}^{i} = \boldsymbol{w}^{i} - \frac{\eta}{m}\sum_x \boldsymbol{\delta}^{\boldsymbol{x},i}\left(\boldsymbol{a}^{\boldsymbol{x},i-1}\right)^{\top} \text{ and } \boldsymbol{b}^{i} = \boldsymbol{b}^{i} - \frac{\eta}{m}\sum_x \boldsymbol{\delta}^{\boldsymbol{x},i}$$

Problem: Vanishing gradient for sigmoid σ

- Gradient of sigmoid:
  $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

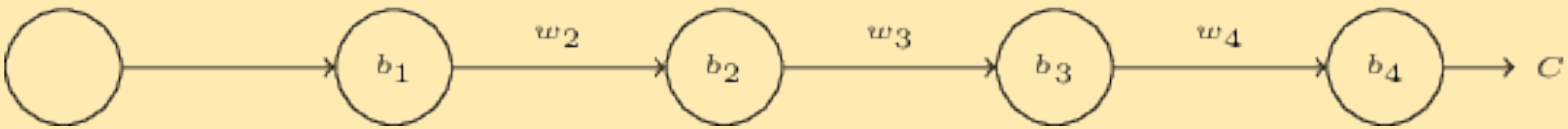


Max$\{\sigma'(x)\}$
$= 0.25$

Gradients in linear network of depth 4

$\leq$ 0,25 $\quad\quad$ $\leq$ 0,25 $\quad\quad$ $\leq$ 0,25

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



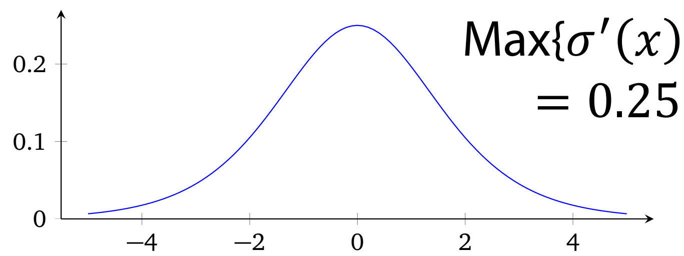
- Assume $|w_i| \leq 1$ (e.g. $w_i \sim N(0,1)$)
- Then: $||w_i\sigma'(z_i)| \leq$ 0,25
- Exponential decrease from later derivatives to earlier ones due to chain rule

Gradient vanishes moving backwards

Problem: Vanishing gradient with large input

- Gradient of sigmoid:
  $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

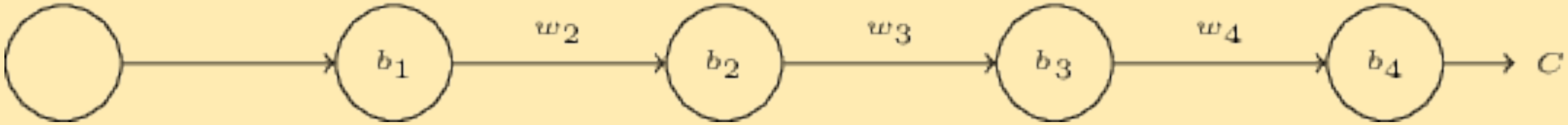
Derivative of sigmoid function



$\text{Max}\{\sigma'(x)\} = 0.25$

Gradients in linear network of depth 4

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



- If $|x|$ very large , then $\sigma(x)$ or (1-- $\sigma(x)$) becomes zero
- So $\sigma'(x)$ becomes zero

- Gradient of sigmoid:
  $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

Gradient vanishes for large inputs to activation functions

NEARLY THE END

Take Home Message

Follow the gradient – with care

Uhhh, a lecture with a hoepfully useful

APPENDIX

Color Convention in this Course

- Formulae, when occurring inline
- Newly introduced terminology and definitions
- Important results (observations, theorems) as well as emphasizing some aspects
- Examples are given with standard orange with possibly light orange frame
- Comments and notes in nearly opaque post-it
- Algorithms
- Reminders (in the grey fog of your memory)

Todays lecture is based on the following

- Jonathon Hare: Lectures 2,3,4,6 of course „COMP6248 Differentiable Programming (and some Deep Learning“)
  http://comp6248.ecs.soton.ac.uk/

- Nielsen: Neural Networks and Deep Learning.
  http://neuralnetworksanddeeplearning.com/, chapters 1,2

- https://medium.com/@ramrajchandradevan/the-evolution-of-gradient-descend-optimization-algorithm-4106a6702d39

- I. Lorentzou: Introduction to Deep Learning, link