Grundlagen der Künstlichen Intelligenz

Wintersemester 25/26

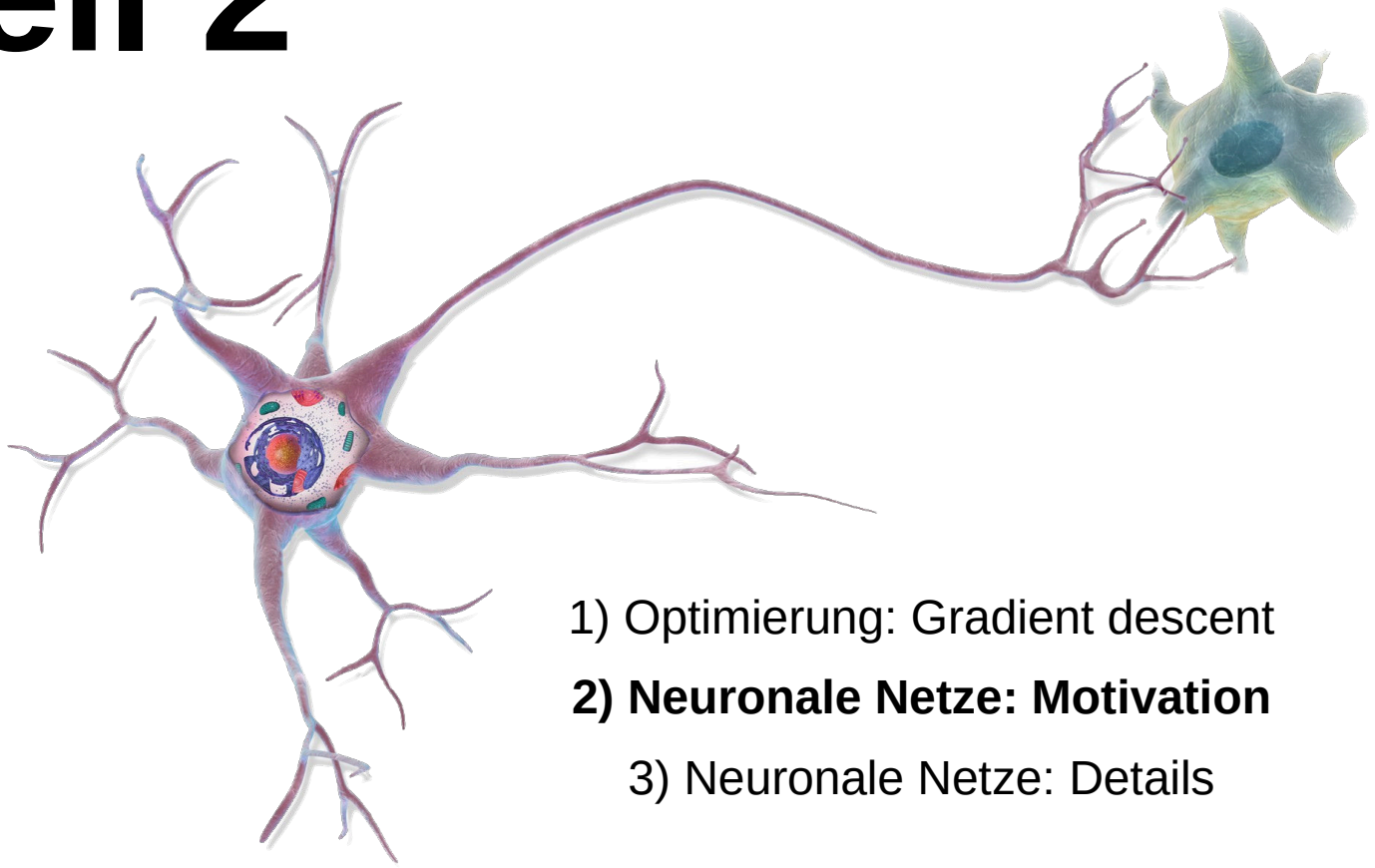
Vorlesung 5

Neuronale Netze, Backpropagation

Teil 2

Prof. Dr. Pascal Friederich
T.T.-Prof. Dr. Peer Nowack

Teil 2



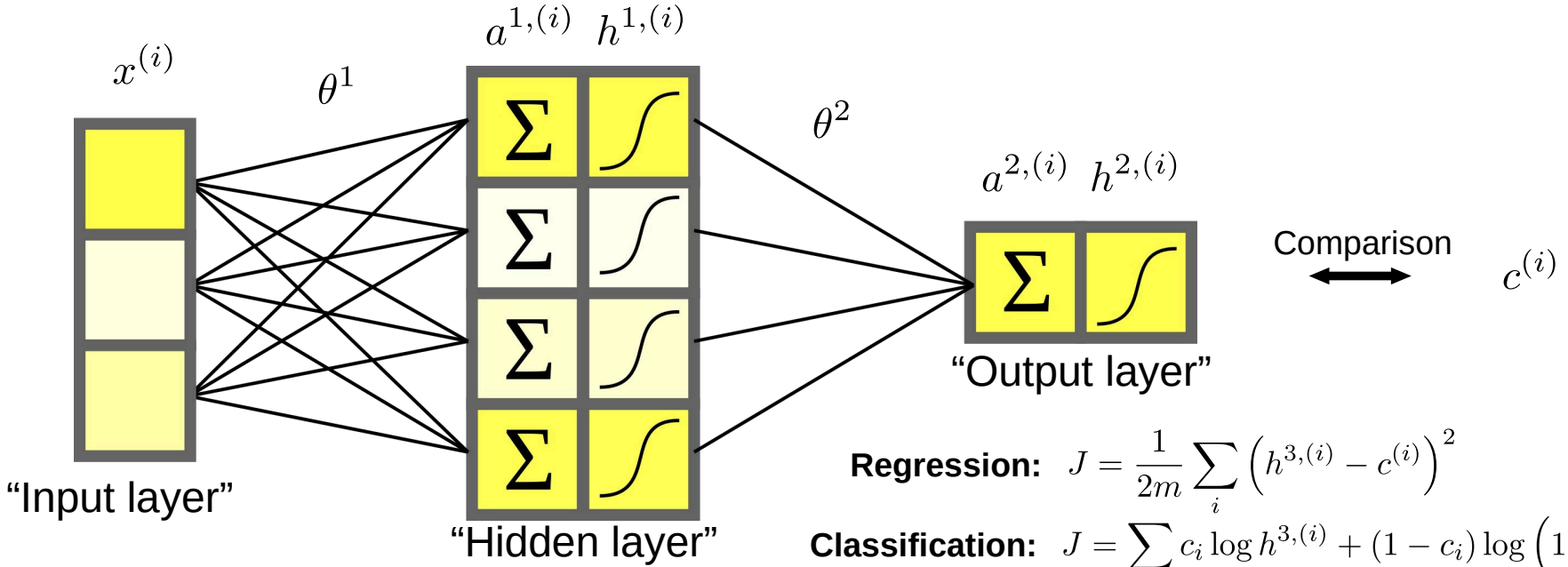
1) Optimierung: Gradient descent

2) Neuronale Netze: Motivation

3) Neuronale Netze: Details

Neuronale Netze: Backpropagation

$$x^{(i)} \xrightarrow{\theta^1} a^{1,(i)} = \left(\theta^1\right)^T x^{(i)} \longrightarrow h^{1,(i)} = \sigma\left(a^{1,(i)}\right) \xrightarrow{\theta^2} a^{2,(i)} = \left(\theta^2\right)^T h^{1,(i)} \longrightarrow h^{2,(i)} = \sigma\left(a^{2,(i)}\right)$$



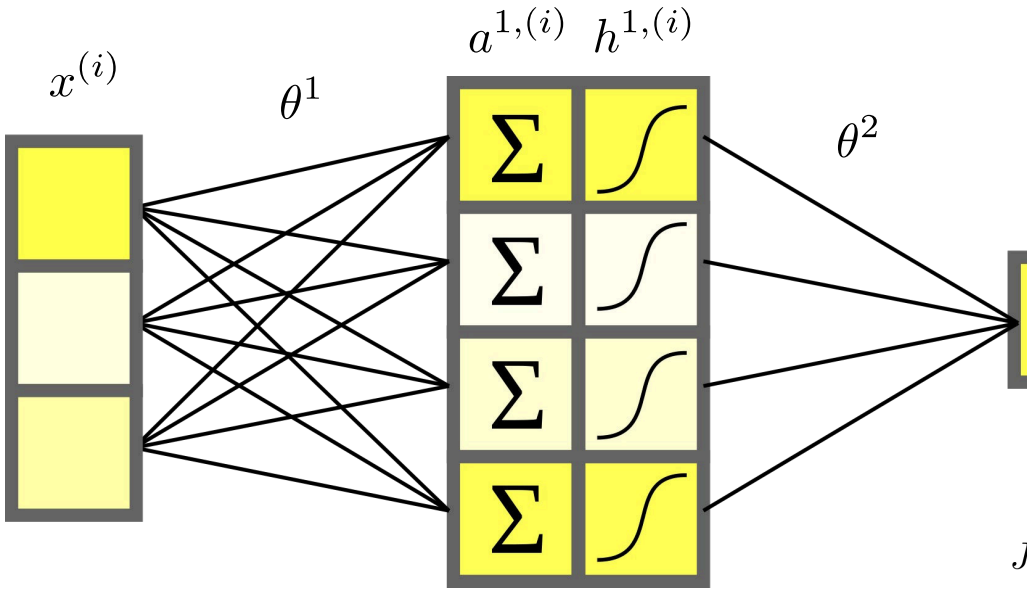
Regression: $J = \frac{1}{2m} \sum_i \left(h^{3,(i)} - c^{(i)}\right)^2$

Classification: $J = \sum_i c_i \log h^{3,(i)} + (1 - c_i) \log\left(1 - h^{3,(i)}\right)$

$$\theta^1 \to \theta^1 - \alpha \frac{\partial J}{\partial \theta^1}$$

$$\theta^2 \to \theta^2 - \alpha \frac{\partial J}{\partial \theta^2}$$

Neuronale Netze: Backpropagation



$$J = \frac{1}{2m} \sum_i \left(h^{3,(i)} - c^{(i)} \right)^2$$

$$J = \sum_i c_i \log h^{3,(i)} + (1 - c_i) \log \left(1 - h^{3,(i)} \right)$$

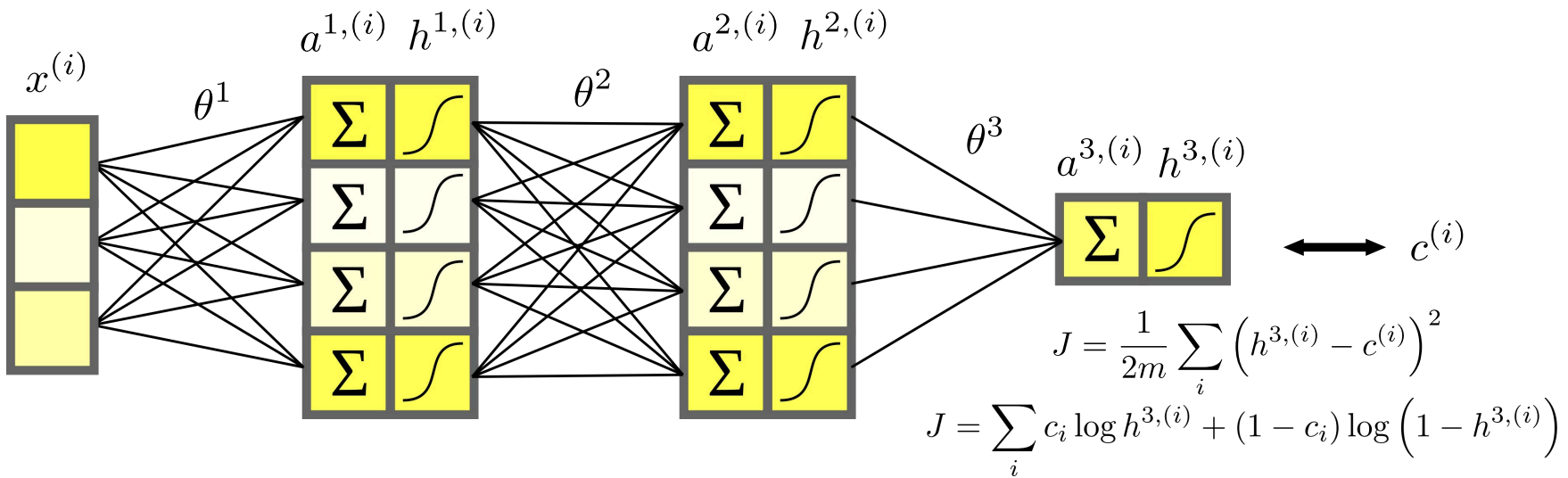
$$\frac{\partial J}{\partial \theta^1} = \frac{\partial J}{\partial h^2} \frac{\partial h^2}{\partial a^2} \frac{\partial a^2}{\partial h^1} \frac{\partial h^1}{\partial a^1} \frac{\partial a^1}{\partial \theta^1}$$

$$\frac{\partial J}{\partial \theta^2} = \frac{\partial J}{\partial h^2} \frac{\partial h^2}{\partial a^2} \frac{\partial a^2}{\partial \theta^2}$$

$$\theta^1 \rightarrow \theta^1 - \alpha \frac{\partial J}{\partial \theta^1}$$

$$\theta^2 \rightarrow \theta^2 - \alpha \frac{\partial J}{\partial \theta^2}$$

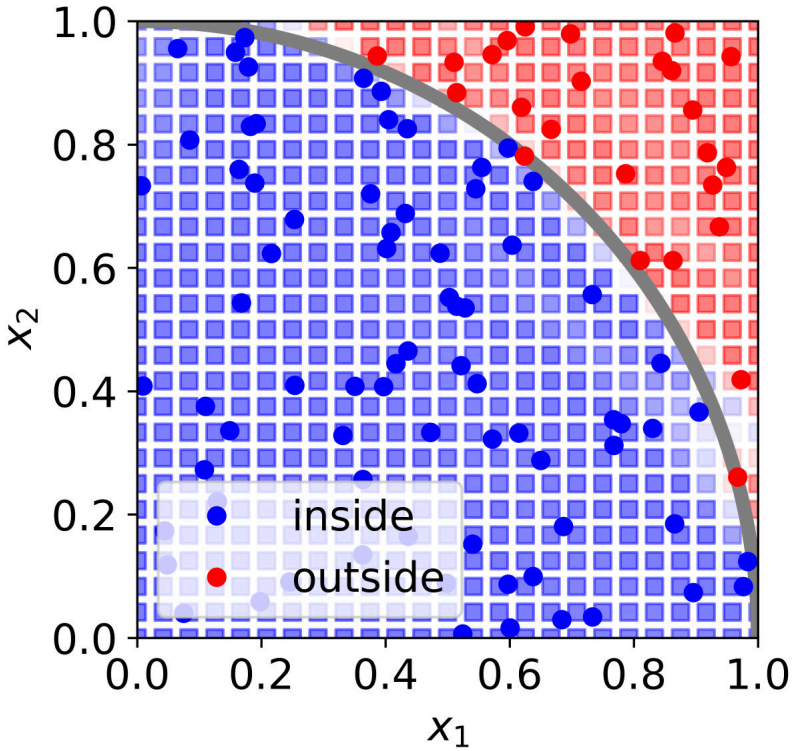
Neuronale Netze: Backpropagation



$$\frac{\partial J}{\partial \theta^1} = \frac{\partial J}{\partial h^3} \frac{\partial h^3}{\partial a^3} \frac{\partial a^3}{\partial h^2} \frac{\partial h^2}{\partial a^2} \frac{\partial a^2}{\partial h^1} \frac{\partial h^1}{\partial a^1} \frac{\partial a^1}{\partial \theta^1} \qquad \frac{\partial J}{\partial \theta^2} = \frac{\partial J}{\partial h^3} \frac{\partial h^3}{\partial a^3} \frac{\partial a^3}{\partial h^2} \frac{\partial h^2}{\partial a^2} \frac{\partial a^2}{\partial \theta^2} \qquad \frac{\partial J}{\partial \theta^3} = \frac{\partial J}{\partial h^3} \frac{\partial h^3}{\partial a^3} \frac{\partial a^3}{\partial \theta^3}$$

$$\theta^1 \to \theta^1 - \alpha \frac{\partial J}{\partial \theta^1} \qquad \theta^2 \to \theta^2 - \alpha \frac{\partial J}{\partial \theta^2} \qquad \theta^3 \to \theta^3 - \alpha \frac{\partial J}{\partial \theta^3}$$

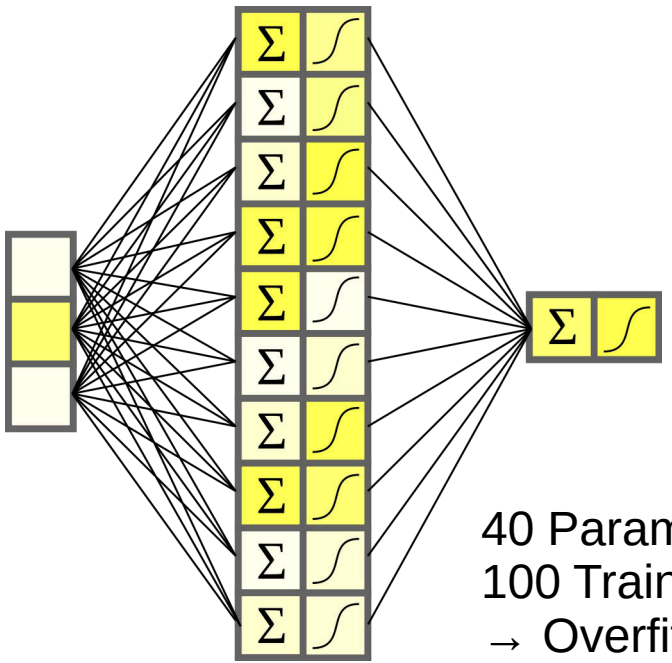
Neuronale Netze: Ergebnisse



Genauigkeit
Training: 100 %
Test: >98 %

Neuronales Netz

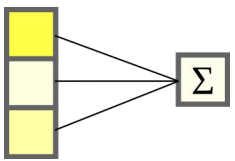
$$a^1(x) = \left(\theta^1\right)^T x$$
$$h^1(x) = \sigma\left(a^1(x)\right)$$
$$a^2(x) = \left(\theta^2\right)^T h^1(x)$$
$$f(x) = h^2(x) = \sigma\left(a^2(x)\right)$$

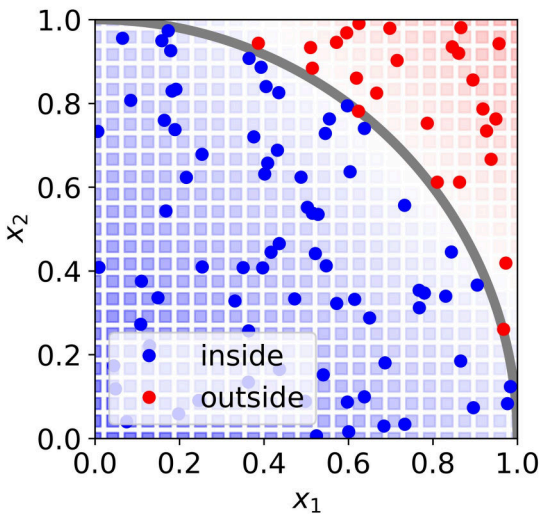


40 Parameter
100 Trainingspunkte
→ Overfitting?

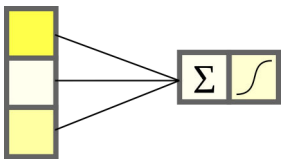
Vergleich mit letzter Vorlesung: Lineare und logistische Regression

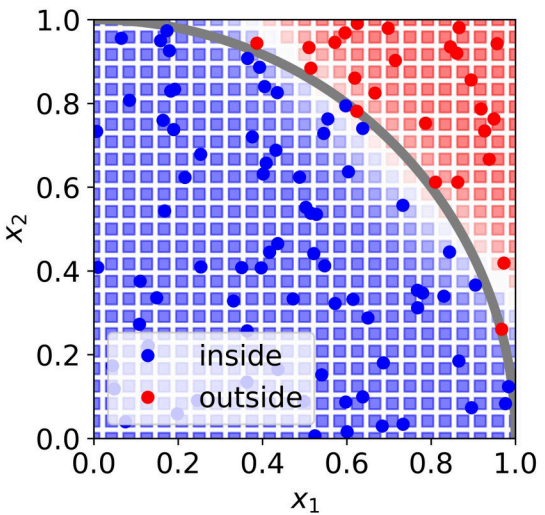
Lineare Regression



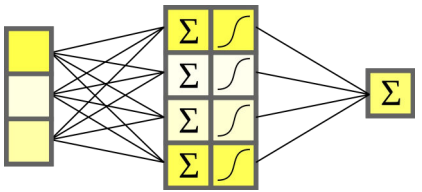


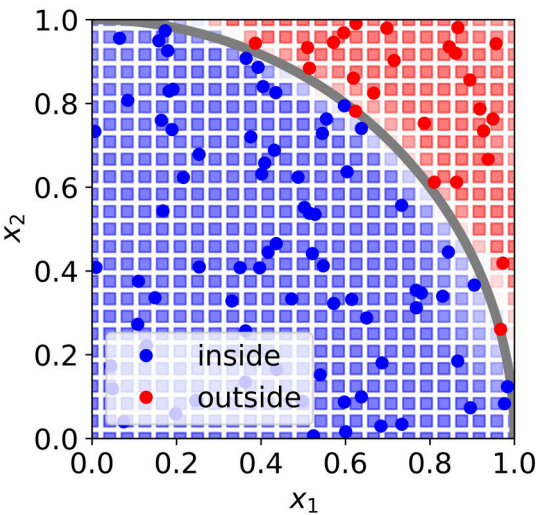
Logistische Regression





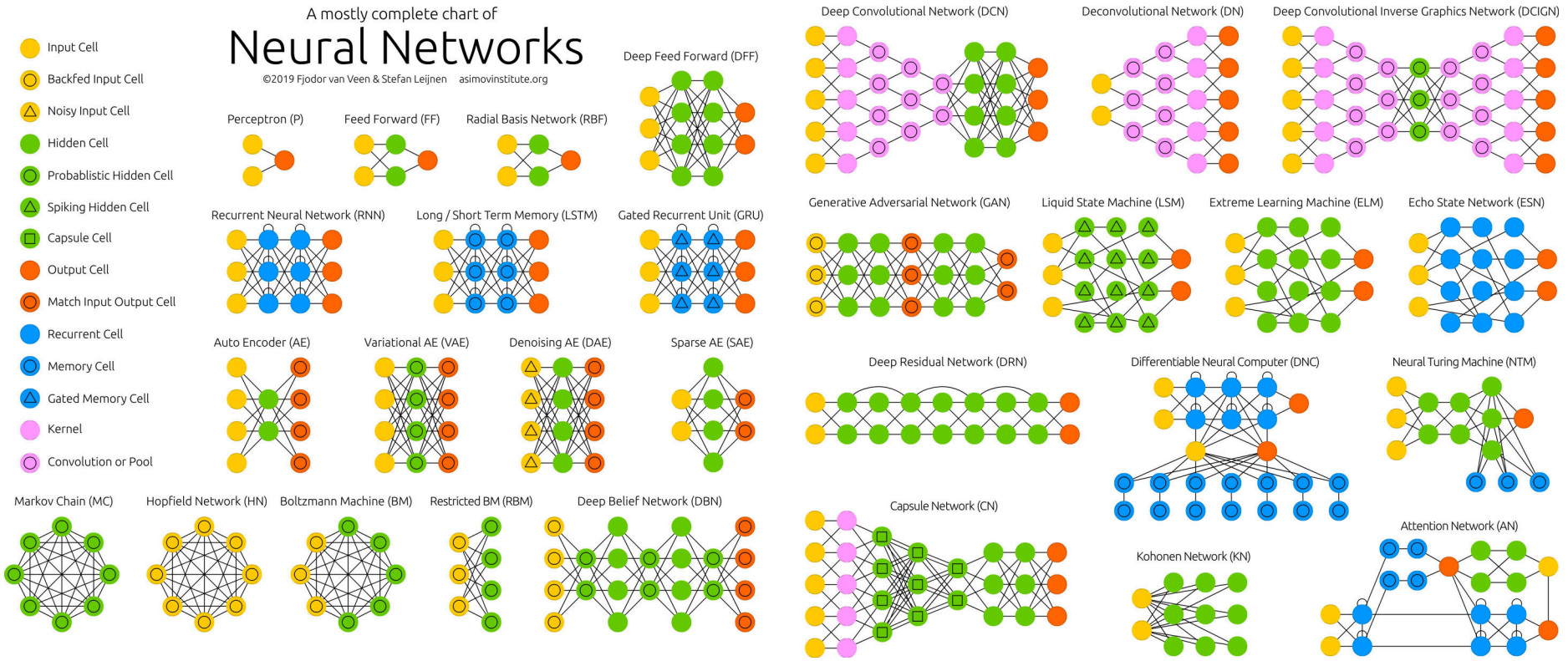
Neuronales Netz





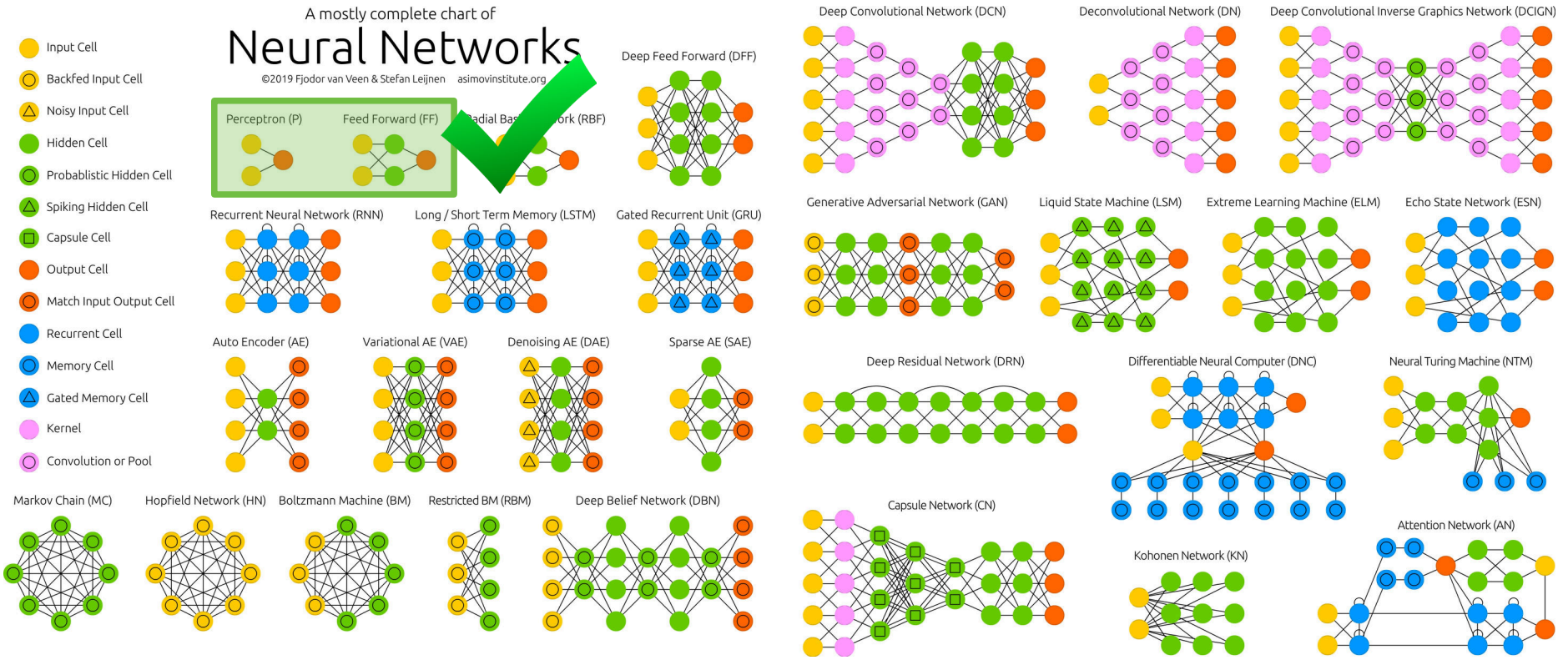
Und jetzt?

A “mostly” complete chart of neural networks *(Asimov institute)*

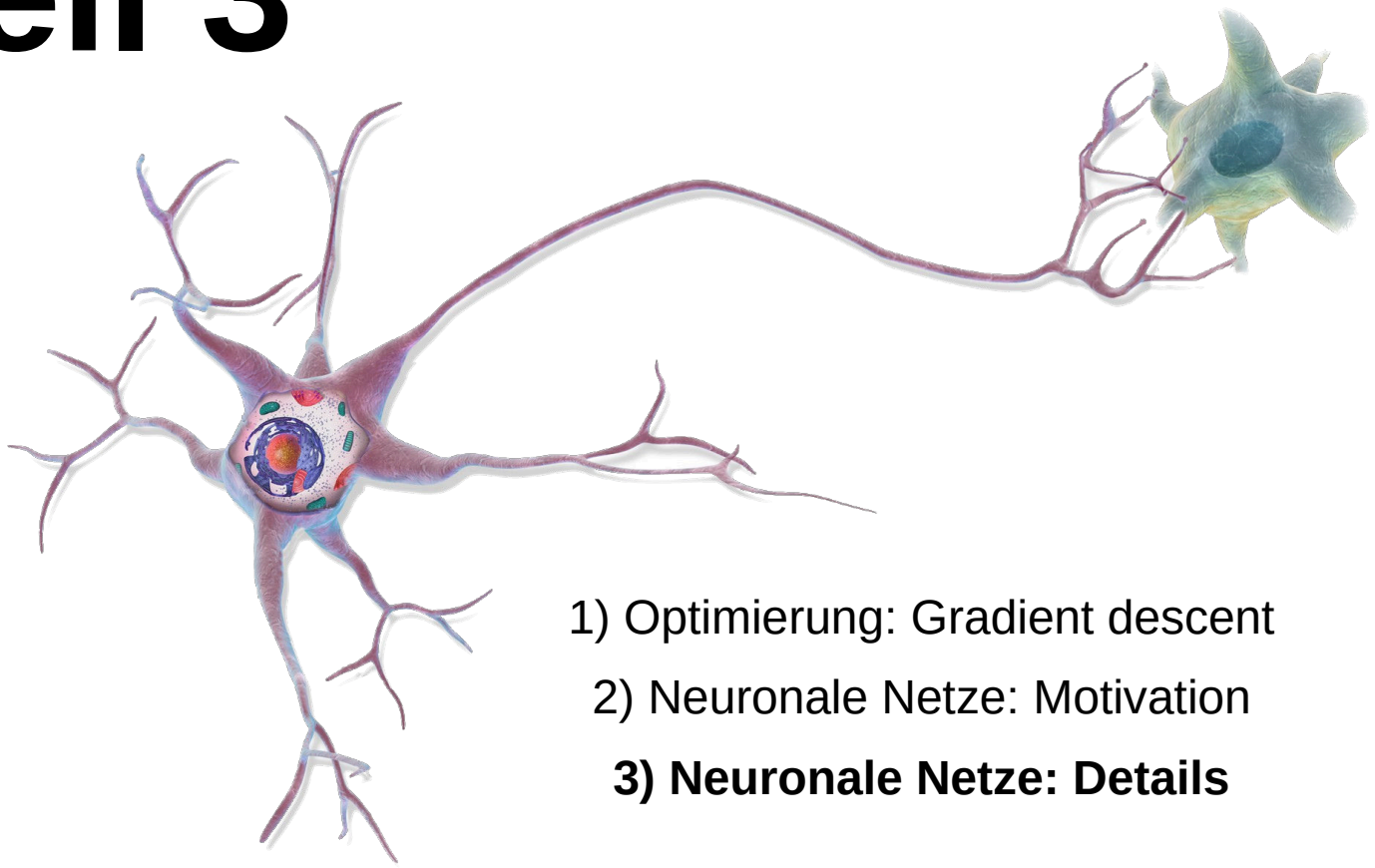


Und jetzt?

A “mostly” complete chart of neural networks *(Asimov institute)*



Teil 3



1) Optimierung: Gradient descent

2) Neuronale Netze: Motivation

3) Neuronale Netze: Details

Warum benötigt man die Aktivierungsfunktion?

Neuronales Netz mit einem hidden layer

$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) \qquad y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$$

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)$$

Beide Funktionen sind linear: $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^\top \boldsymbol{x}$

(ohne Bias) $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^\top \boldsymbol{w}$

$$f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{W} \boldsymbol{w}$$

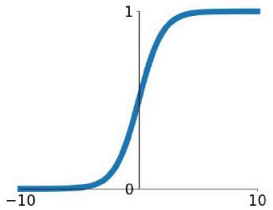
Das ist äquivalent zu einfachem linearen Modell: $f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w}'$ mit $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$

Aktivierungsfunktionen

Viele verschiedene Aktivierungsfunktionen können verwendet werden

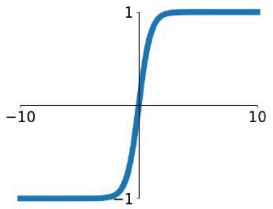
Sigmoid

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



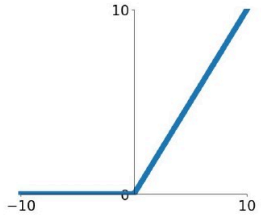
tanh

$$\tanh(x)$$



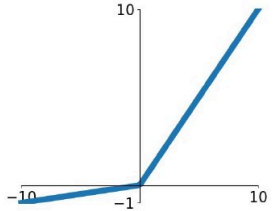
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

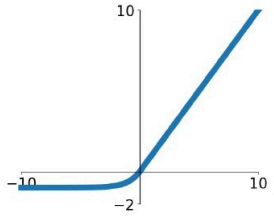


Maxout

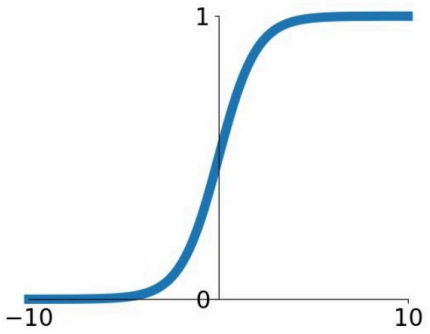
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Aktivierungsfunktionen: Sigmoid



Sigmoid

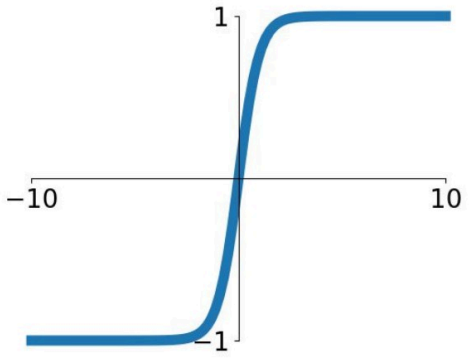
Berechnet: $f(a) = \sigma(a) = \frac{1}{1 + \exp(-a)}$

- Wertebereich [0,1]
- Historisch, Interpretation als saturierende Aktivierungsrate der Neuronen

Probleme

- Wenn Input groß/klein: Flach → Kein Gradient
- Output nicht um 0 zentriert: Macht Initialisierung schwer
- $\exp()$ ist aufwändig zu berechnen

Aktivierungsfunktionen: Tangens hyperbolicus (Tanh)



tanh(x)

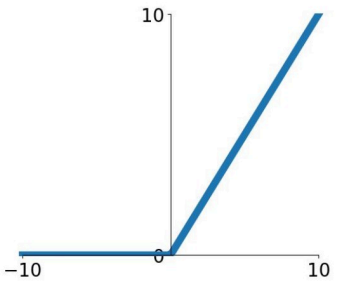
Berechnet: $h(a) = \tanh(a)$

- Um 0 zentriert

Probleme

- Wenn Input groß/klein: Flach → Kein Gradient

Aktivierungsfunktionen: Tangens hyperbolicus (Tanh)



ReLU
(Rectified Linear Unit)

Berechnet: $h(a) = \max(0, a)$

- Saturiert nicht bei großem Input
- Sehr effizient zu berechnen
- Konvergiert typischerweise schneller als sigmoid/tanh

Probleme

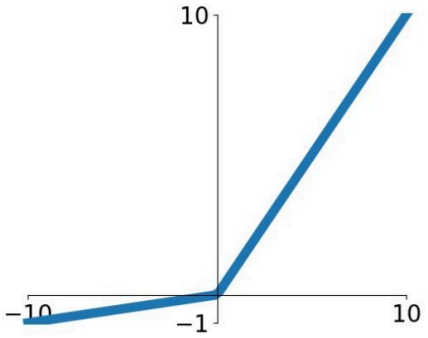
- Wenn Input klein: Flach → Kein Gradient
- Output nicht um 0 zentriert: Macht Initialisierung schwer

Nebenbemerkung: ReLU

- **ReLU** ist ein Akronym für „**Re**ctified **L**inear **U**nit“
- Der Name geht aber zurück auf **Relu** Patrascu, Sysadmin an der Universität von Toronto

Image: Twitter, Relu Patrascu, @ikoflexer

Aktivierungsfunktionen: Leaky ReLU



Leaky ReLU

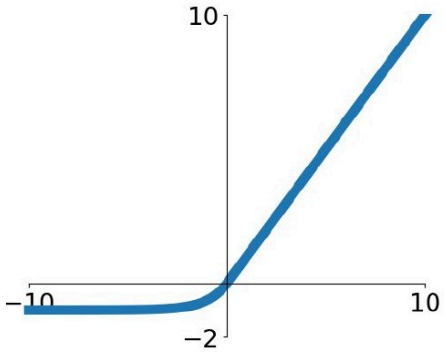
Berechnet: $h(a) = \max(0.1a, a)$

- Saturiert nicht → Hat immer einen Gradienten
- Sehr effizient zu berechnen
- Konvergiert typischerweise schneller als sigmoid/tanh

Erweiterung: Parametric Rectifier (PReLu)

- Berechnet: $h(a) = \max(\alpha a, a)$
- Parameter α muss/kann gelernt werden

Aktivierungsfunktionen: Exponential Linear Unit



Exponential Linear Units (ELU)

Berechnet: $h(x) = \begin{cases} a & \text{if } a > 0 \\ \alpha(\exp(a) - 1) & \text{if } a \leq 0 \end{cases}$

- Alle Vorteile von ReLU
- Outputs im Mittel näher bei 0
- Benötigt die Berechnung von $\exp()$

Aktivierungsfunktionen: Auswahl

Praktisches Vorgehen

- Benutze ReLU, aber justiere die Lernrate und die Initialisierung der Parameter
- Teste Leaky ReLU und ELU
- Teste tanh, aber erwarte nicht zu viel
- Benutze keine sigmoid Funktion (außer als Output für Klassifizierungsprobleme)

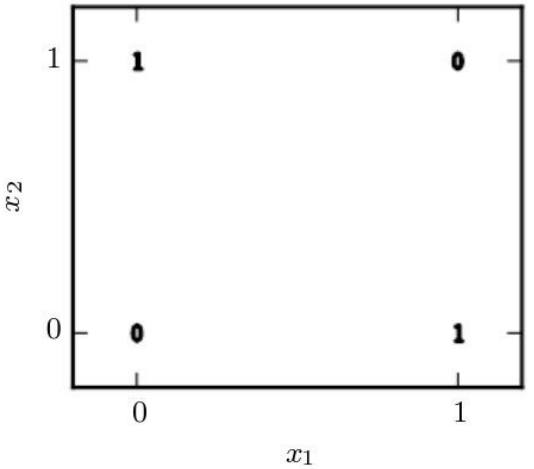
Fazit

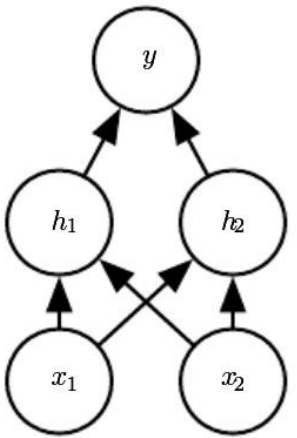
- Aktivierungsfunktion als Hyperparameter: Nächste Woche mehr!

Beispiel: XOR

- Sehr einfaches Problem
- Vier Input Vektoren, vier Labels (0, 1)
- **Nicht lösbar** mit **linearem Modell**
- **Lösbar mit**
  - Neuronalem Netz mit einem hidden layer mit zwei Neuronen
  - Aktivierung: step (nur hier als Beispiel)

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \text{step}\left(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\right) + b$$





Beispiel: XOR

Eine mögliche Lösung: $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \text{step}\left(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\right) + b$

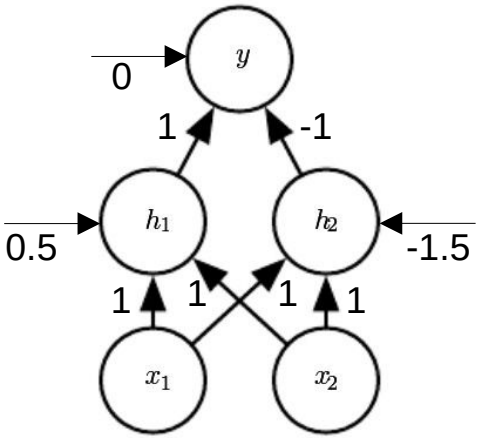
Schritt für Schritt:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{c} = \begin{bmatrix} -0.5 \\ -1.5 \end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad b = 0$$

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \quad \boldsymbol{XW} + \boldsymbol{c} = \begin{bmatrix} -0.5 & -1.5 \\ 0.5 & -0.5 \\ 0.5 & -0.5 \\ 1.5 & 0.5 \end{bmatrix}$$

Für jede Zeile

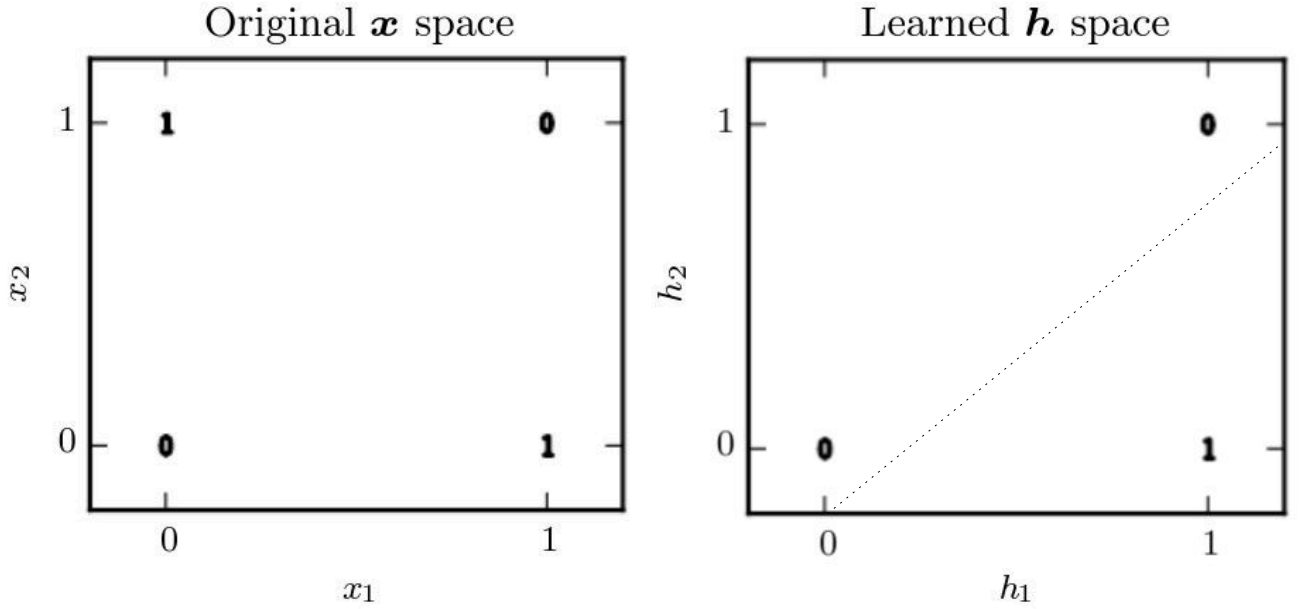
$$\text{step}\left(\boldsymbol{XW} + \boldsymbol{c}\right) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{w}^\top \text{step}\left(\boldsymbol{XW} + \boldsymbol{c}\right) + b = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



Interpretation 1: Lernen einer linear separierbaren Repräsentation

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \text{step}\left(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\right) + b = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Erklärung



$$\max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Interpretation 1: Lernen der Logik

Aufgabe

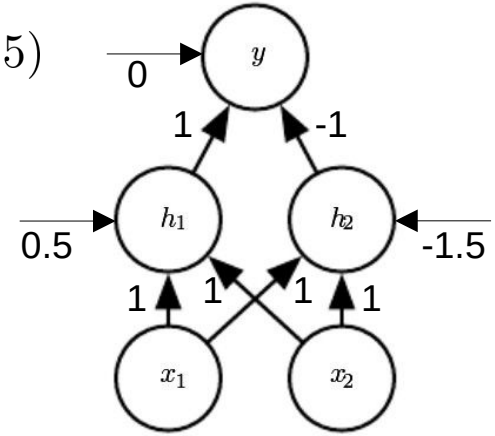
$$\mathrm{XOR}(x_1, x_2) = (x_1 \mathrm{OR}\ x_2)\ \mathrm{AND\ NOT}\ (x_1 \mathrm{AND}\ x_2)$$

Erklärung

$$h_1 = (x_1 \mathrm{OR}\ x_2) \qquad h_1 = \mathrm{step}(1 \cdot x_1 + 1 \cdot x_2 - 0.5)$$

$$h_2 = (x_1 \mathrm{AND}\ x_2) \qquad h_2 = \mathrm{step}(1 \cdot x_1 + 1 \cdot x_2 - 1.5)$$

$$y = h_1\ \mathrm{AND\ NOT}\ h_2 \qquad y = (h_1 - h_2)$$



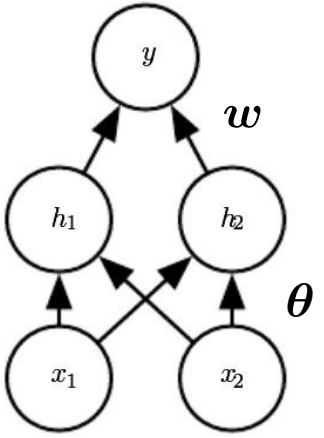
Allgemeiner: Representation Learning

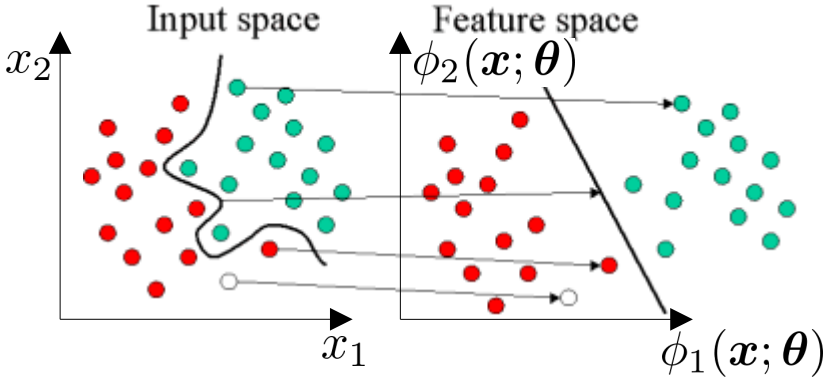
Interpretation von neuronalen Netzen

- Kombination aus:
  - Nicht-lineare Transformation des Inputs (Gewichte $\boldsymbol{\theta}$)
  - Lineares Modell (Gewichte $\boldsymbol{w}$)

$$y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta})^\top \boldsymbol{w}$$

($\boldsymbol{h} = \boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta})$)





Warum mehr als ein Layer?

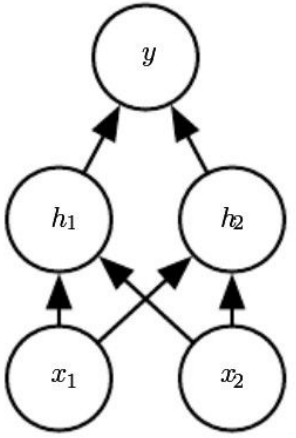
Aufbau des neuronalen Netzes

- **Modular**: Typischerweise in Form von aufeinanderfolgenden Layers
  - Freiheitsgrade: Anzahl, Art und Größe der Layers
- **Universal Approximation Theorem** (Hornik 1989, Cybenko 1989)
  - Neuronales Netz mit **einem hidden layer** kann alle Funktionen approximieren
  - (Natürlich mit Einschränkungen, die hier zu weit gehen)
  - Praktisch sind dafür aber sehr viele Neuronen notwendig
- **Tiefere neuronale Netze** können mit weniger Parametern bessere Modelle liefern
  - Besser: weniger Parameter, kleinere Validierungs- und Testfehler
  - Aber: eventuell schwieriger zu optimieren/trainieren

Welche Gewichte nutzen wir?

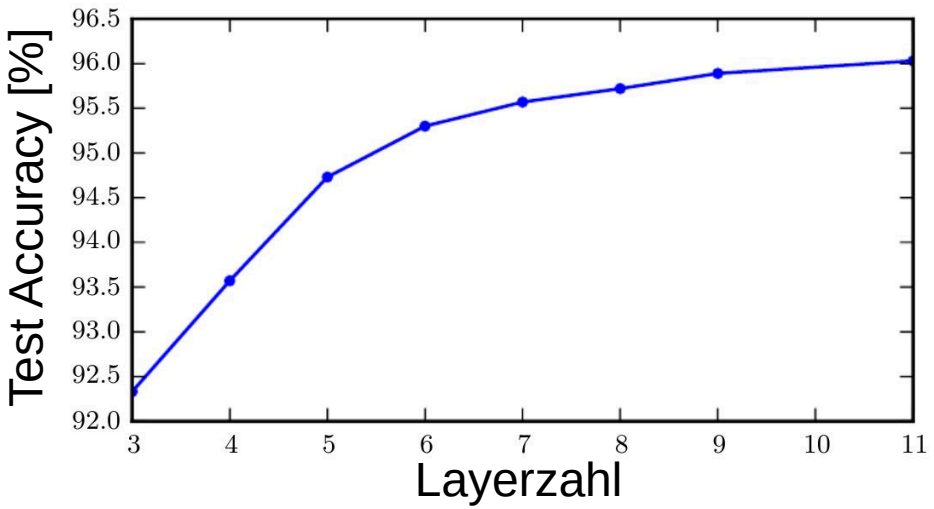
Initialisierung der Gewichte vor dem Training

- **Typisch:** Zufällige Gewichte
- Aber warum?
- Angenommen, wir setzen alle Gewichte w auf den gleichen Wert
- → Symmetrie in allen Gewichten und Informationsflüssen
- → wir werden immer wieder gleiche Gradienten erhalten
- → Initialisieren mit zufälligen (nicht-identischen) Gewichten



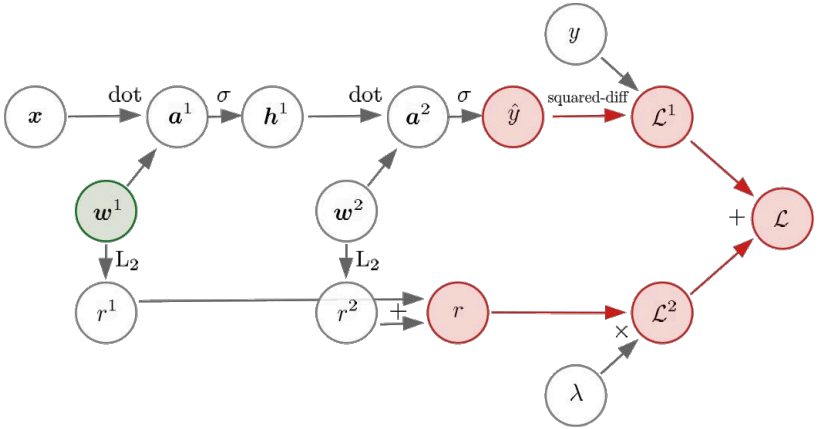
Tiefe neuronale Netze

Klassifizierung von fotografierten Hausnummern



https://arxiv.org/pdf/1312.6082.pdf

Teil 4



$$\nabla_{\boldsymbol{w}^1}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}^1}\nabla_{\boldsymbol{w}^1}\mathcal{L}^1 + \frac{\partial \mathcal{L}}{\partial \mathcal{L}^2}\nabla_{\boldsymbol{w}^1}\mathcal{L}^2 = \dots$$

1) Optimierung: Gradient descent

2) Neuronale Netze: Motivation

3) Neuronale Netze: Details

4) Neuronale Netze: Back-propagation

Gradient descent

- **Bisher:** Training von **logistischer Regression** mit **gradient descent**
- **Jetzt:** Neuronale Netze, fast identischer Algorithmus
- **Unterschiede:**
  - Sehr viel mehr Parameter
  - Nicht konvex: Viele lokale Optima
  - Gradient descent funktioniert hier häufig nicht „einfach so“

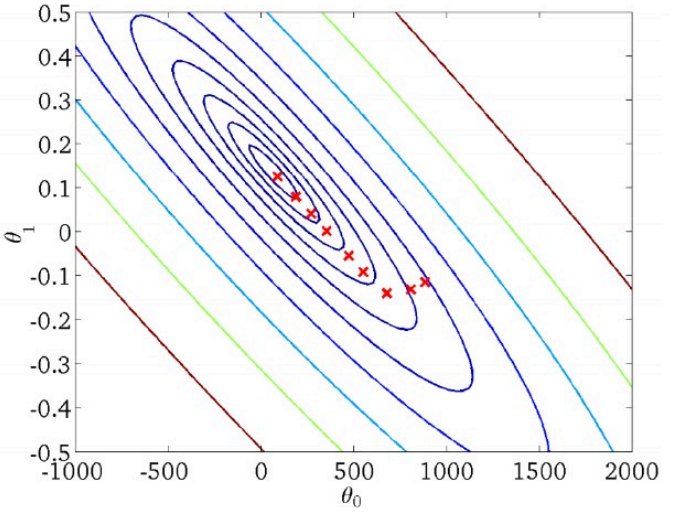
    → NN-Winter von ~2000 bis 2012

- **In den letzten 5-10 Jahren:**
  - Mehr Compute-Power
  - Mehr Daten
  - Einige Tricks

Gradient descent

- **Grundidee aus Teil 1**
  - Berechne Gradienten
  - Bewege die Parameter in die entgegengesetzte Richtung

- **Für neuronale Netze**
  - Grundsätzlich das selbe, aber:
  - … höher-dimensional
  - … schwieriger zu visualisieren



Loss Funktion

- **Ziel: Minimierung der Loss Funktion / Kostenfunktion**
  - Gegeben: Daten $\mathcal{D}$ (*M* Datenpunkte $\boldsymbol{x}_i$)
  - Gesucht: Optimale Gewichte $\boldsymbol{\theta}$
  - → Minimiere Loss Funktion (inkl. Regularisierung)

$$\mathcal{L}(\boldsymbol{\theta}, \mathcal{D}) = \sum_{i=1}^{M} l\left(\boldsymbol{x}_i, \boldsymbol{\theta}\right) + \lambda \text{ penalty } (\boldsymbol{\theta})$$

$$\boldsymbol{\theta} = \left\{ \mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(L)} \right\}$$

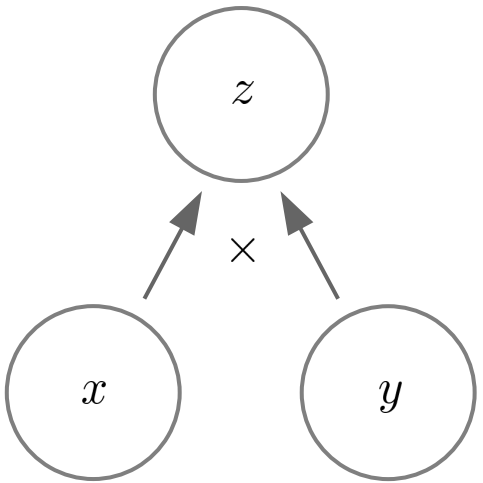
- **Vorgehen: Berechnung der partiellen Ableitungen**
  - Gewichte **W**: $\dfrac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}$
  - Bias **b**: $\dfrac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}$

→ Rekursive Anwendung der **Kettenregel**

$$\frac{d}{dw} L(h(...(w))) = \frac{dL}{dh} \frac{dh}{d\ldots} \frac{d\ldots}{d\ldots} \frac{d\ldots}{dw}$$

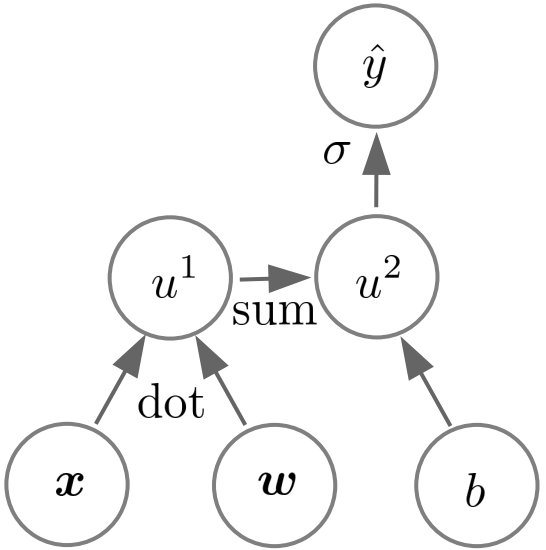
Wie kann man das automatisieren und implementieren?

→ Backpropagation durch Darstellung des computational graphs

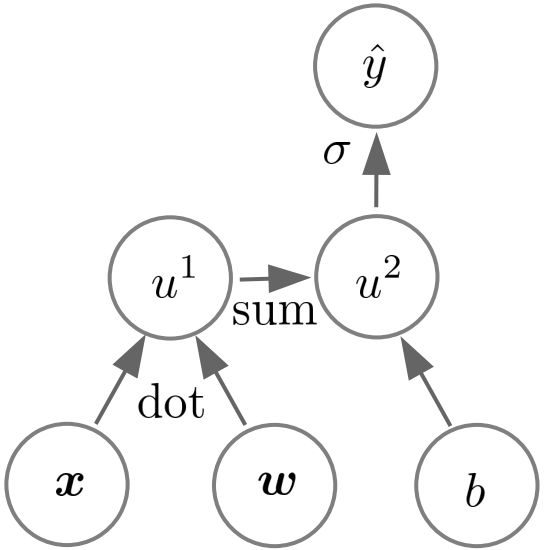


$$z = xy$$

Computational graph



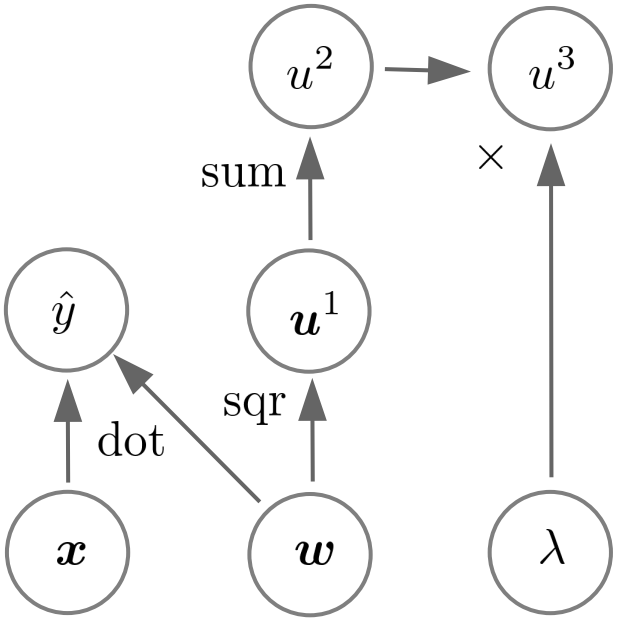
Computational graph



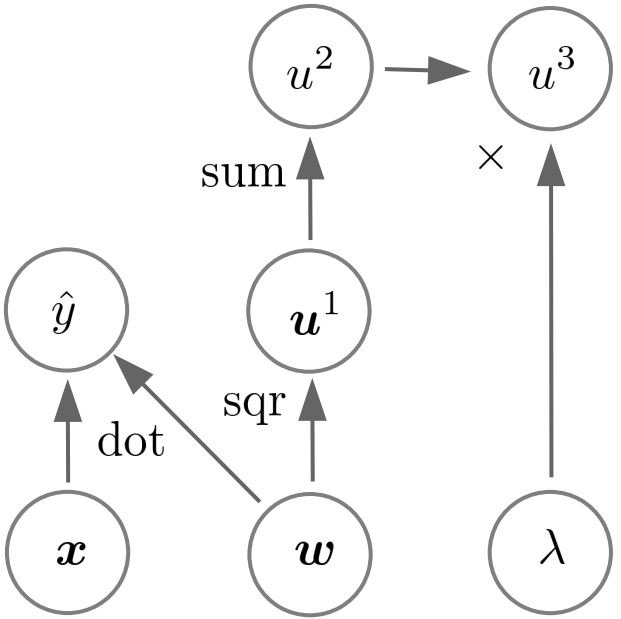
$$\hat{y} = \sigma\left(\boldsymbol{x}^\top \boldsymbol{w} + b\right)$$

Logistische Regression

Computational graph



Computational graph



Lineare Regression $\boldsymbol{x}^\top \boldsymbol{w}$
mit Regularisierung $\lambda \sum_i w_i^2$

Backpropagation aus computational graph

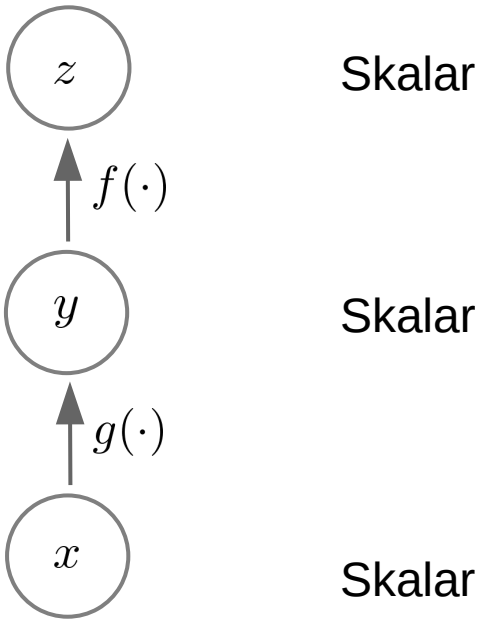
Einfaches Beispiel

$$x \in \mathbb{R}, y \in \mathbb{R}$$

$$y = g(x)$$

$$z = f(y) = f(g(x))$$

$$\rightarrow \quad \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$



Backpropagation aus computational graph

Etwas komplizierter

$$\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$$

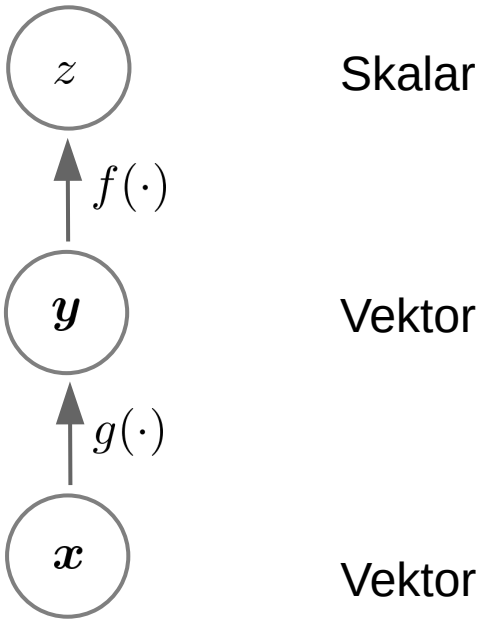
$$\boldsymbol{y} = g(\boldsymbol{x})$$

$$z = f(\boldsymbol{y})$$

$$\rightarrow \quad \frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

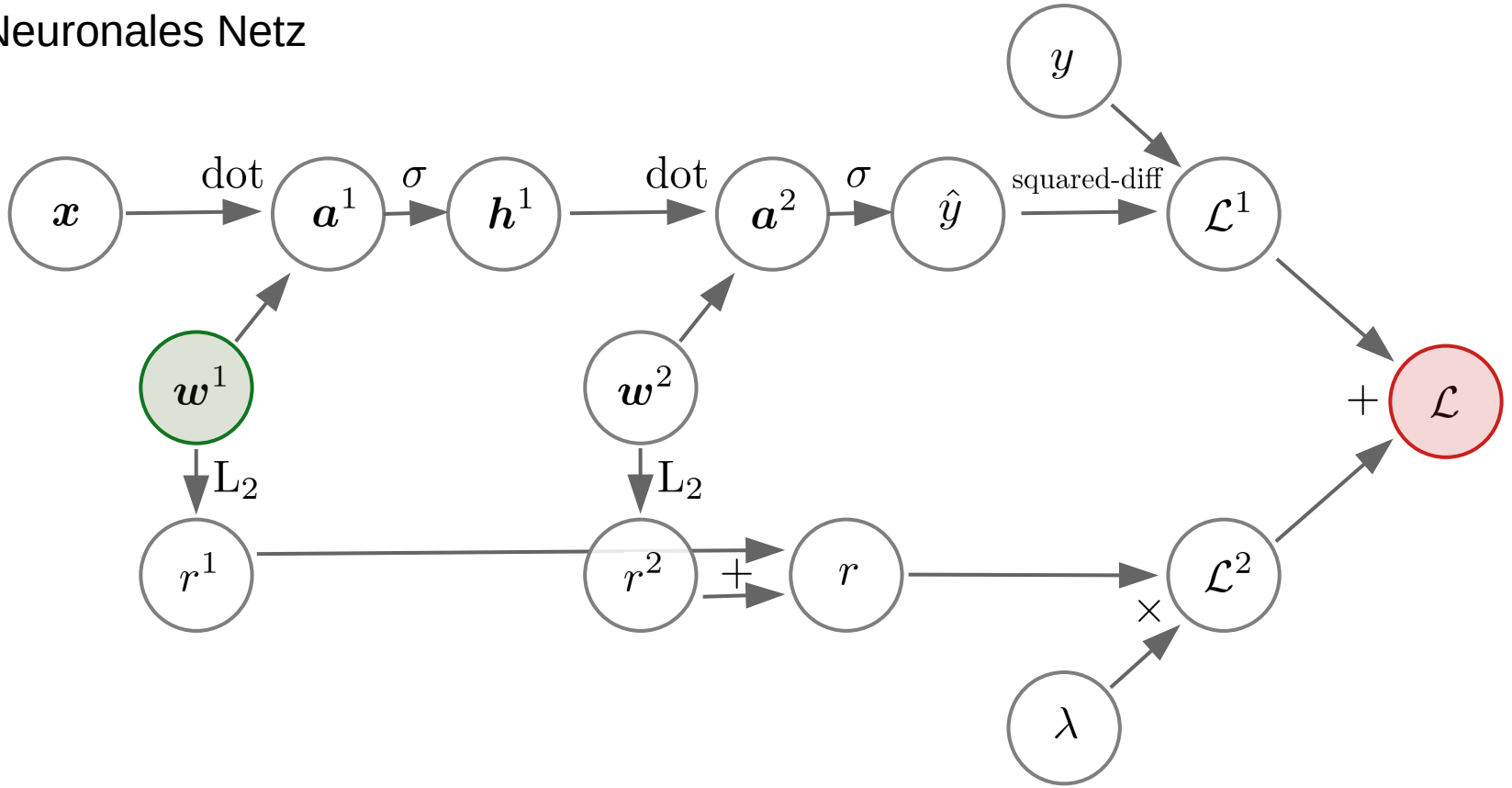
$$\rightarrow \quad \nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \right)^\top \nabla_{\boldsymbol{y}} z$$

Jacobi Matrix — Gradient



Backpropagation aus computational graph

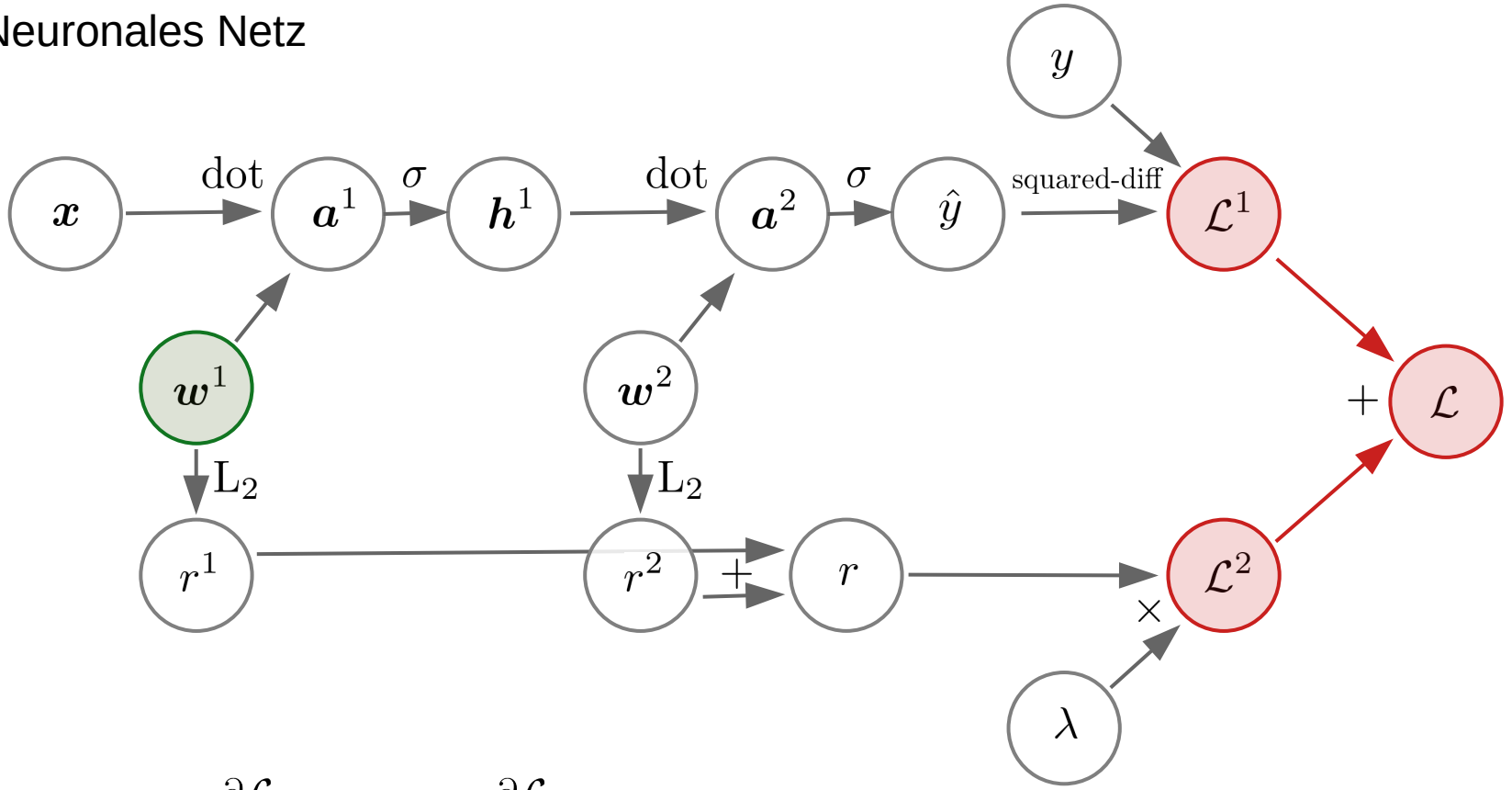
Neuronales Netz



$$\nabla_{\boldsymbol{w}^1} \mathcal{L} = \dots$$

Backpropagation aus computational graph

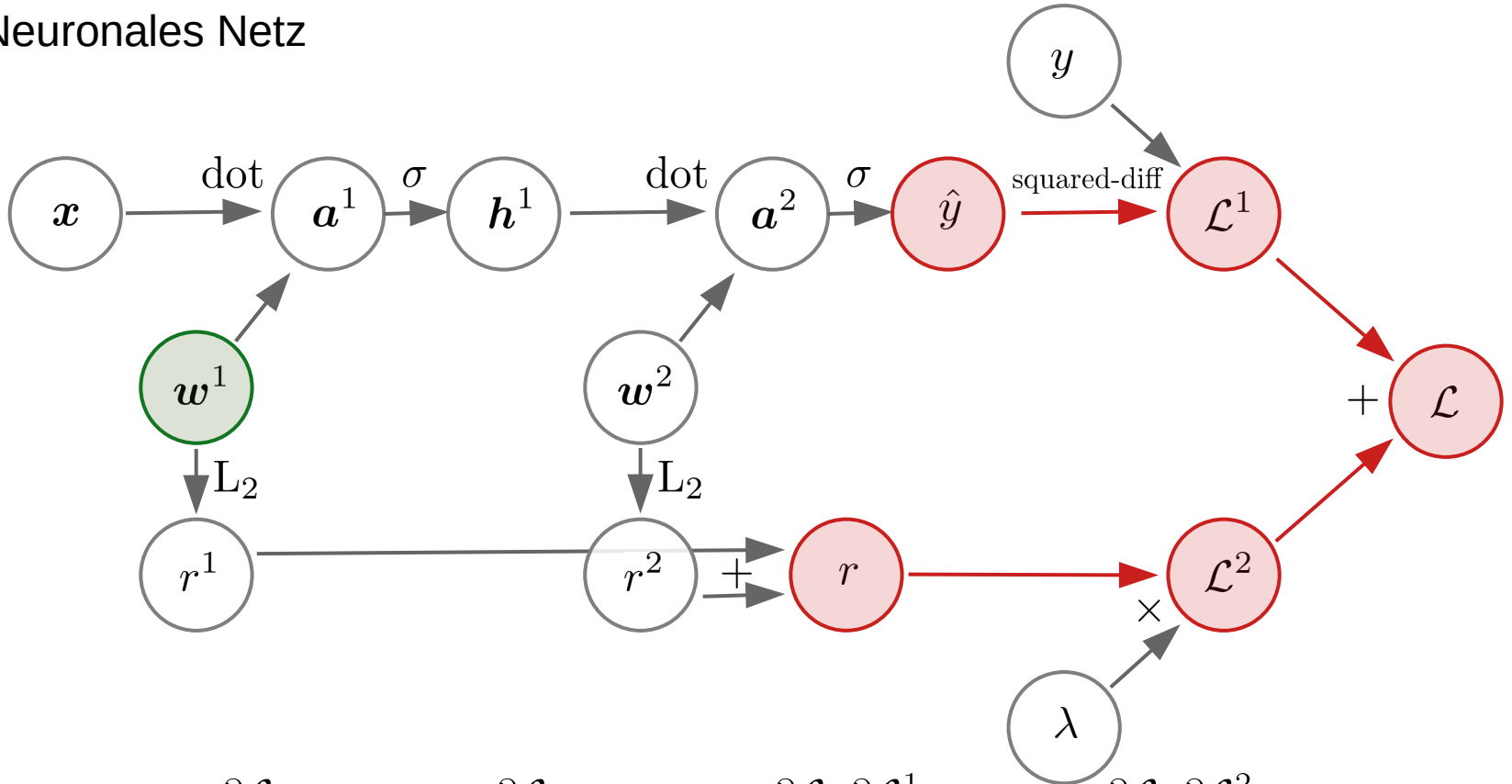
Neuronales Netz



$$\nabla_{\boldsymbol{w}^1}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}^1}\nabla_{\boldsymbol{w}^1}\mathcal{L}^1 + \frac{\partial \mathcal{L}}{\partial \mathcal{L}^2}\nabla_{\boldsymbol{w}^1}\mathcal{L}^2 = \dots$$

Backpropagation aus computational graph

Neuronales Netz



$$\nabla_{\boldsymbol{w}^1}\mathcal{L} = \frac{\partial\mathcal{L}}{\partial\mathcal{L}^1}\nabla_{\boldsymbol{w}^1}\mathcal{L}^1 + \frac{\partial\mathcal{L}}{\partial\mathcal{L}^2}\nabla_{\boldsymbol{w}^1}\mathcal{L}^2 = \frac{\partial\mathcal{L}}{\partial\mathcal{L}^1}\frac{\partial\mathcal{L}^1}{\partial\hat{y}}\nabla_{\boldsymbol{w}^1}\hat{y} + \frac{\partial\mathcal{L}}{\partial\mathcal{L}^2}\frac{\partial\mathcal{L}^2}{\partial r}\nabla_{\boldsymbol{w}^1}r = \dots$$

Neuronale Netze: Computational cost

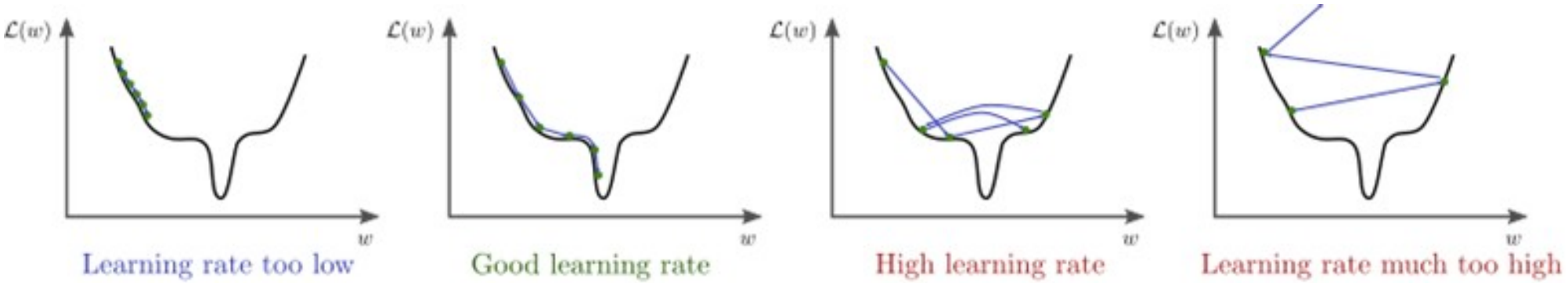
- **Forward propagation**

$$a = wx + b$$

  - Matrix-Vektor Produkt
  - Ca. eine Multiplikation und Addition pro Gewicht
- **Back propagation**
  - Matrix-Vektor Produkt und äußeres Produkt
  - Ca. zwei Multiplikationen und Additionen pro Gewicht
- **Daraus folgt**
  - Kosten sind linear in der Zahl der Schichten
  - Kosten sind quadratisch in der Zahl der Neuronen pro Schicht

Training: Gradient Descent plus Tricks

- **Gradient Descent**
  - Lernrate muss „gut“ gewählt werden



Training: Gradient Descent plus Tricks

- **Verbesserung: Momentum (Impuls/„Schwung“)**
  - Gradient wie vorher

$$\boldsymbol{g}_t = \frac{1}{b} \sum_{i \in I_t} \nabla_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{x}_i; \boldsymbol{\theta}_t\right)$$

Minibatch

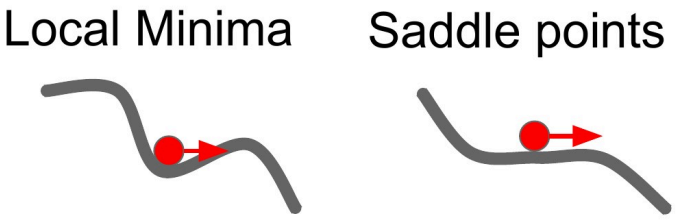
  - Jetzt: Berechne den gleitenden Mittelwert des Gradienten

$$\boldsymbol{m}_0 = \mathbf{0}, \quad \boldsymbol{m}_{t+1} = \gamma \boldsymbol{m}_t + (1 - \gamma)\, \boldsymbol{g}_t$$

  - Parameter Update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{m}_{t+1}$

$\gamma = 0$: Gradient Descent

$\gamma > 0$: Momentum



  - Interpretation: Gradient wirkt sich auf „Geschwindigkeit“ der Anpassung aus, und damit nur indirekt auf den Update-Schritt selbst

Training: Gradient descent plus Tricks

- **Weitere Tricks**
  - **Gradient Normalization**

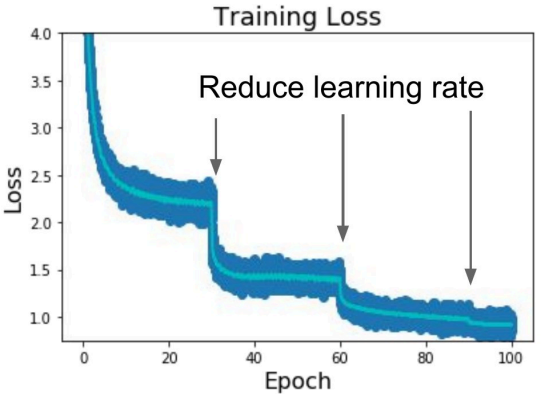
    → Idee: Größere Schritte in flachen Gebieten, kleinere Schritte in steileren Gebieten

Gradient: $\boldsymbol{g}_k = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_k)$ Running avarage gradient norm: $\boldsymbol{v}_{t+1,i} = \gamma \boldsymbol{v}_{t,i} + (1-\gamma)\boldsymbol{g}_{t,i}^2$

Update: $\boldsymbol{\theta}_{t+1,i} = \boldsymbol{\theta}_{t,i} - \frac{\alpha}{\sqrt{\boldsymbol{v}_{t+1,i} + \epsilon}} \boldsymbol{g}_{t,i}$

  - **Learning rate decay**

    → Idee: Passe die Lernrate nach k Epochen an



  - **Adaptive Momentum** (Adam)

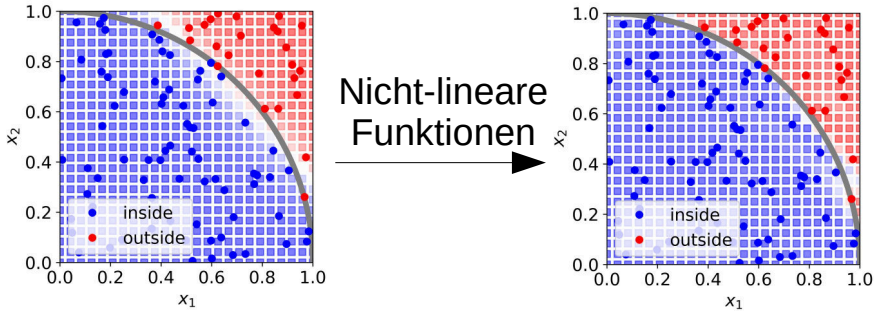
    → Idee: Verbinde Momentum mit adaptiver Lernrate und Gradient Normalisierung

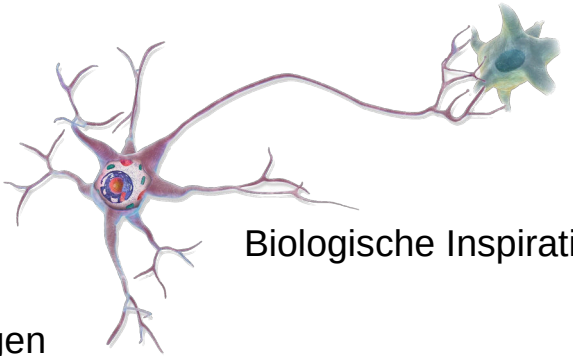
Sollten wir immer Backpropagation nutzen?

- Neuronale Netze: Grob inspiriert durch Biologie
- Aber: Im Gerhirn gibt es (vermutlich) keinen Mechanismus, der Fehler ausrechnet und damit eine Backpropagation erlaubt
- Es gibt Alternativen für Backpropagation (oft durch Biologie motiviert)

- Aber: Bisher ist Backpropagation **der** dominante Algorithmus fürs Training

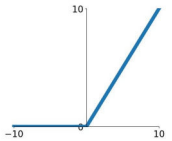
Zusammenfassung





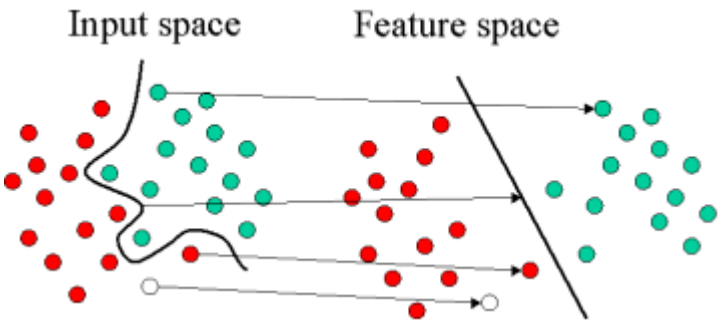
Biologische Inspiration

Aktivierungen



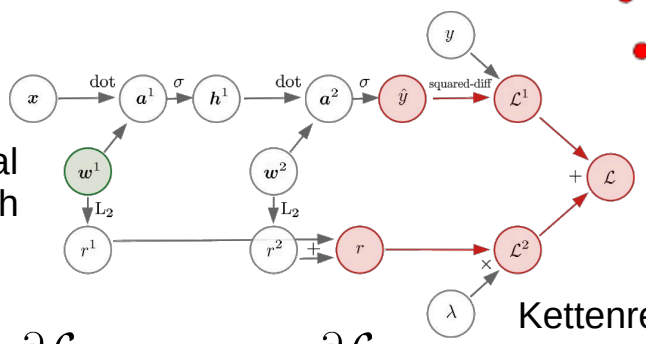
ReLU
(Rectified Linear Unit)

Representation learning



1) Optimierung: Gradient descent
2) Neuronale Netze: Motivation
3) Neuronale Netze: Details
4) Back-propagation: Details

Computational graph



Kettenregel

$$\nabla_{\boldsymbol{w}^1}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}^1}\nabla_{\boldsymbol{w}^1}\mathcal{L}^1 + \frac{\partial \mathcal{L}}{\partial \mathcal{L}^2}\nabla_{\boldsymbol{w}^1}\mathcal{L}^2 = \dots$$