

# Large Language Models in Theorie und ~~Praxis~~ Python

Timo Baumann

Folien verfügbar:

[https://www.timobaumann.de/  
work/Main/StatWoLLMs](https://www.timobaumann.de/work/Main/StatWoLLMs)

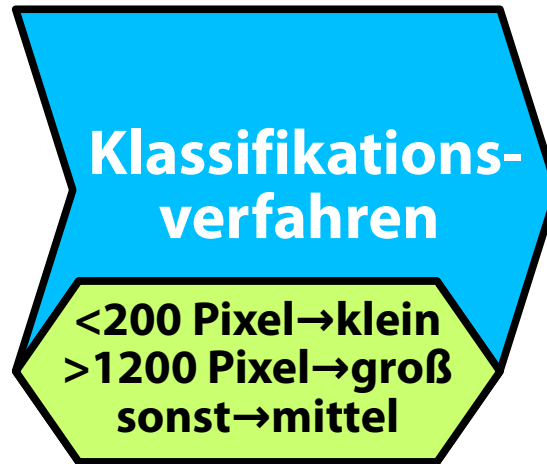
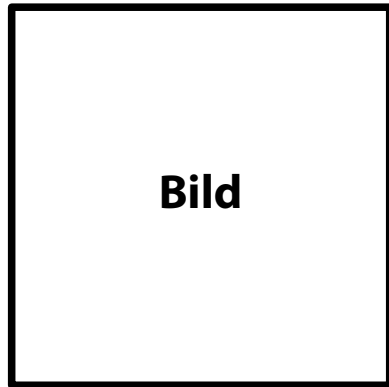
# Inhalt 1. Block

- Maschinelles Lernen
- Neuronale Netze und Berechnungsgraphen
- Fehlerrückführung und datenbasierte Vorgehensweise; funktionale Sicht auf maschinelles Lernen
- Aufbau von NN-Toolkits

# Maschinelles Lernen

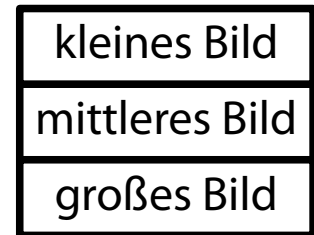
# Klassifikation

Eingabe



Experte

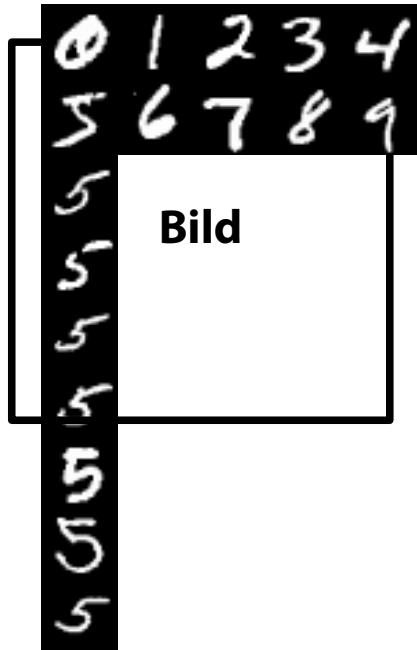
Ausgabe



- einfache Fallunterscheidung
- konkret benennbare Kriterien
- oft **kein** maschinelles Lernen nötig

# Klassifikation

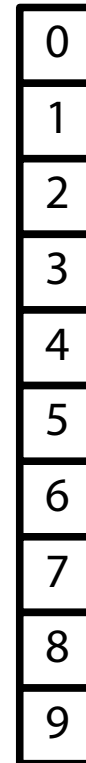
Eingabe



**Experte**

komplizierte Muster  
→ maschinelles Lernen

Ausgabe



# (überwachtes) Maschinelles Lernen

Eingabe  
Beispiel

Trainingsdaten

Ausgabe  
Lösung

Klassifikations-  
verfahren

“Prinzip” des  
Verfahrens

Lernen bzw. Training:  
Parameter so einstellen,  
dass möglichst oft die  
Ziellösung erreicht wird

Parameter

“Einstellungen”  
des Verfahrens

0	0	0
1	1	1
2	2	2
3	3	3

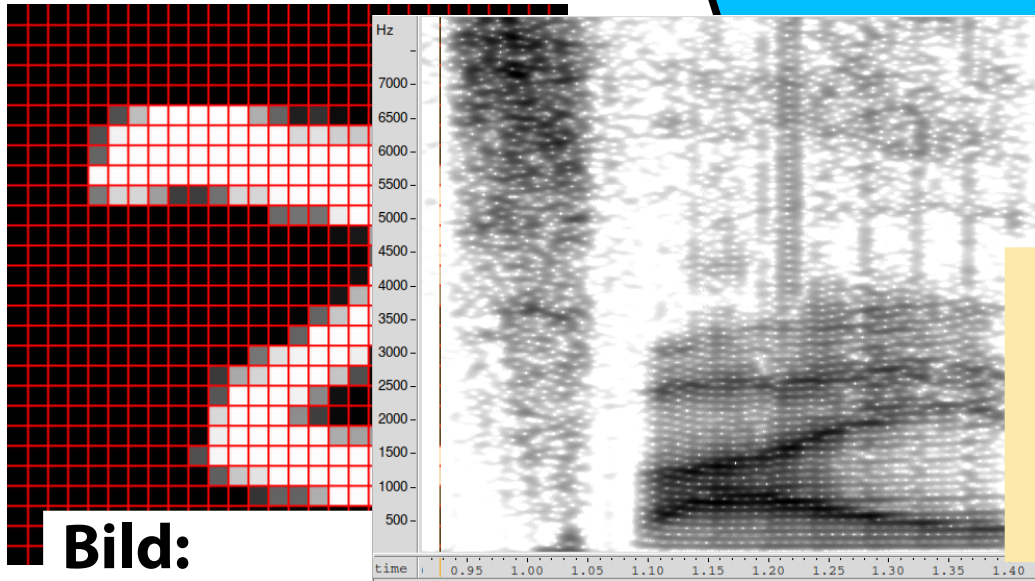
7	7	7
8	8	8
9	9	9

# Generische Lernmodelle: vielfältige Anwendungsmöglichkeiten

Eingabe  
Beispiel

Trainingsdaten

Ausgabe  
Lösung



**Bild:**

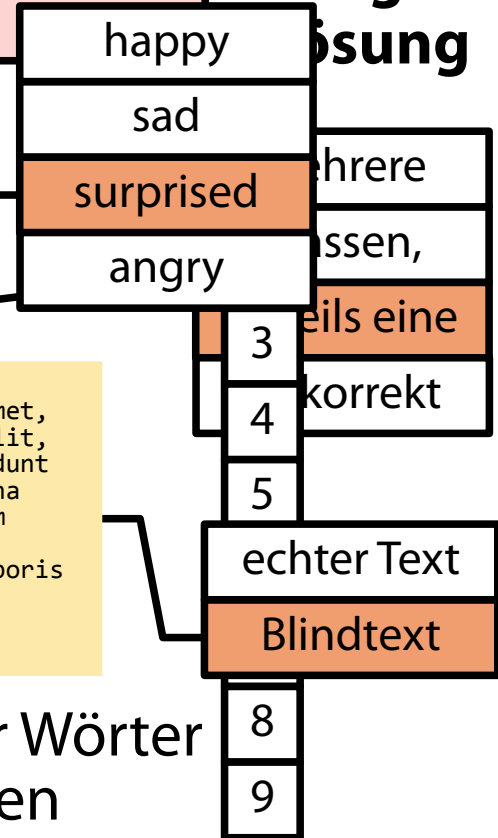
z.B. 28×28  
Pixelwerte

**Ton:** Zeitverlauf von  
Spektraleigenschaften

emotions-  
ren

Lorem ipsum dolor sit amet,  
consectetur adipisicing elit,  
sed eiusmod tempor incididunt  
ut labore et dolore magna  
aliqua. Ut enim ad minim  
veniam, quis nostrud  
exercitation ullamco laboris  
nisi ut aliquid ex ea  
commodi consequat.

**Text:** Folge der Wörter  
oder Buchstaben





# Maschinelles Lernen



*“Generierung von Wissen aus Erfahrung”*

*Erfahrung* → Trainingsdaten

**Wissen** → Modellparameter

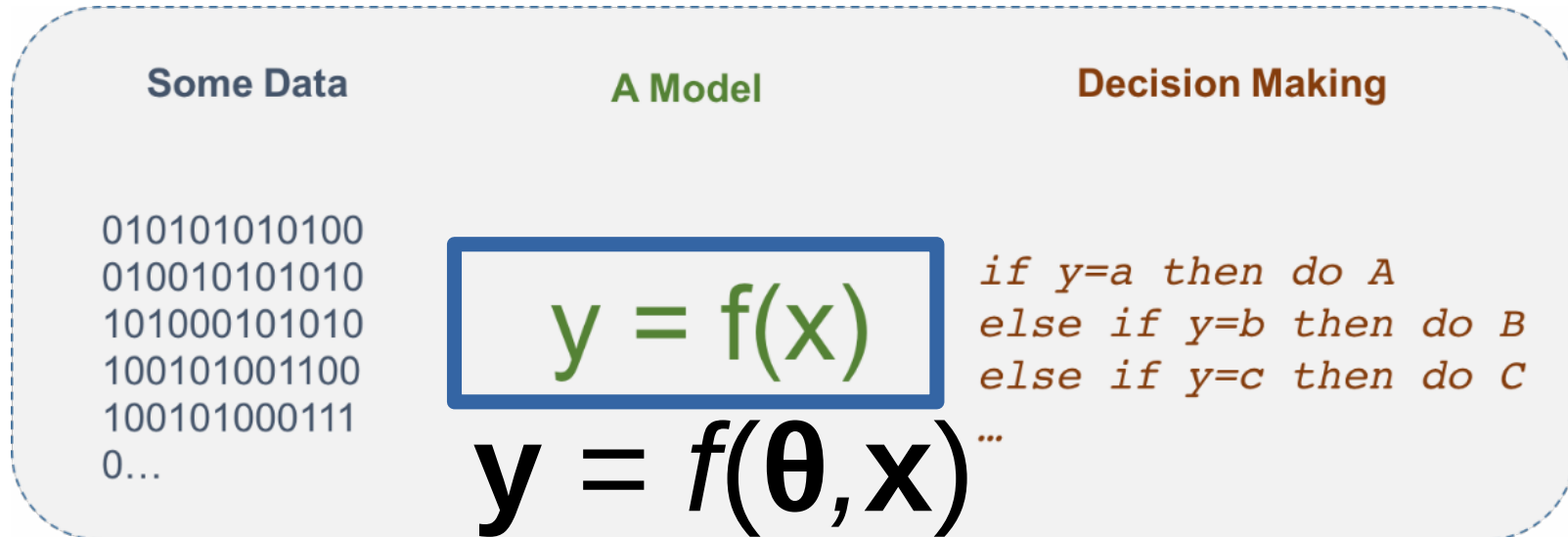
*Generierung* → Einstellen der Parameter

[en.wikipedia.org](https://en.wikipedia.org):

*“ability to effectively perform a specific task”*

# What is Machine Learning?

- Machine learning (ML) refers to algorithms used to extract patterns from data and learn a mathematical model that could be used by a computer program to make intelligent decisions.



# ML Problems

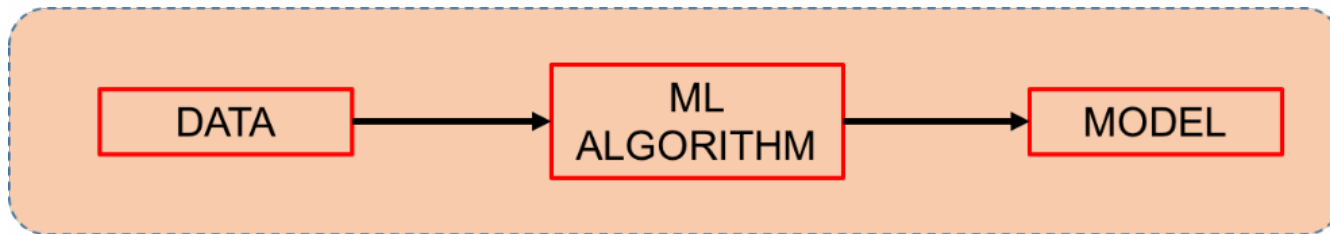
$$\mathbf{y} = f(\boldsymbol{\theta}, \mathbf{x})$$

- we have pairs  $(x,y)$   
→ **supervised** learning
- we only know some  $x$ 's, want to learn useful(!)  $y$ 's  
→ **unsupervised** learning
- in-between:
  - we have lots of  $x$ 's and lots of  $y$ 's (but no pairing)
  - $x$ 's and  $y$ 's are of variable dimensionality
  - $x$  and  $y$  are part of a long sequence. We want to learn  $f$  (or its parameters  $\theta$ ) such that we optimize our performance in the long run.  
→ **reinforcement** learning

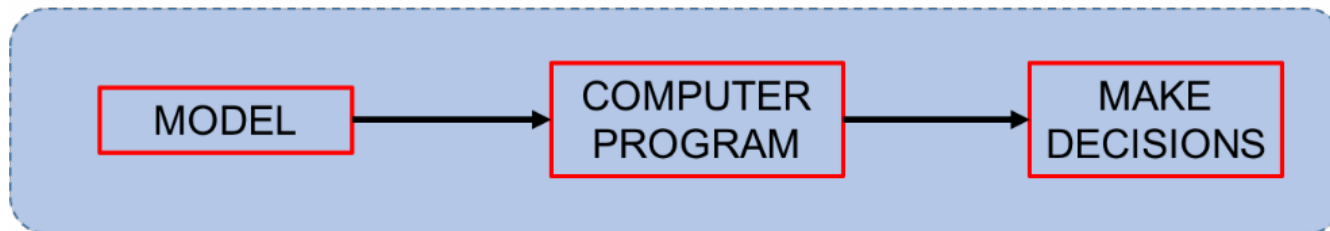
# Practical aspects of machine learning

- Using machine learning algorithms usually involves at least two phases:

## LEARNING PHASE



## APPLICATION PHASE



# Aufgabenarten für maschinelles Lernen

- Zuordnen
  - klassifizieren: Spam-E-Mail ja/nein, ...
  - ranking: Suchergebnisse, Kundenranger, ...
- Transformieren:
  - Text-zu-Sprache, Sprache-zu-Text, ...
  - Stimmadaptierung, Übersetzung, ...
- Struktur induzieren und generalisieren
  - Gruppieren (Clustering)
  - Kondensieren (Dimensionsreduktion), ...
- Generieren: Bilder, Geschichten, ...
- Pläne entwickeln: Kochrezepte, ...
- Interagieren: Dialogsysteme, selbstfahrende Autos, ...

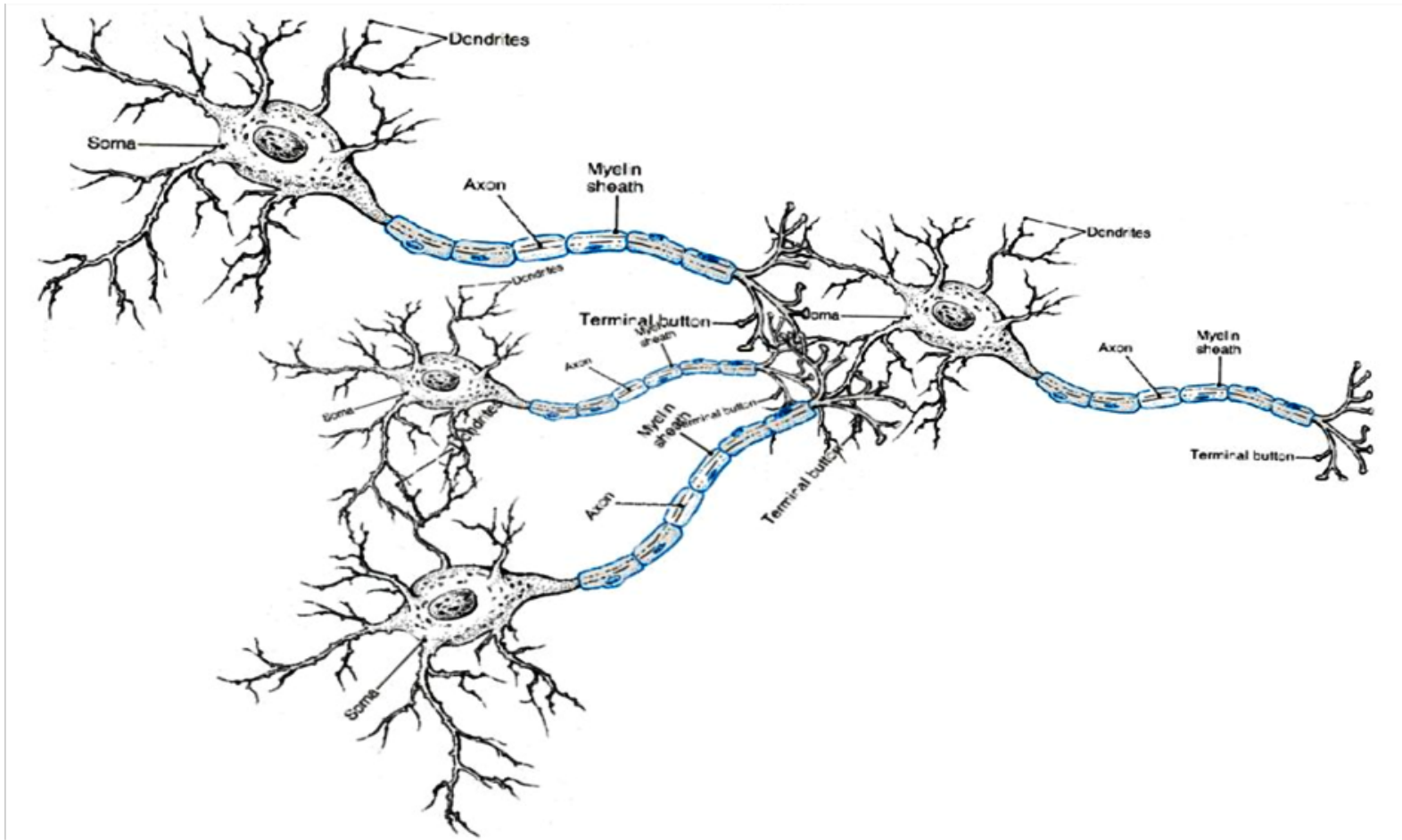
**überwacht:**  
Lernbeispiele  
mit Musterlösung

**unüberwacht:**  
Lernbeispiele  
ohne  
Musterlösung


**teilüberwacht:**  
Lerndaten mit  
Teillösungen oder  
nur teilweise  
mit Lösung

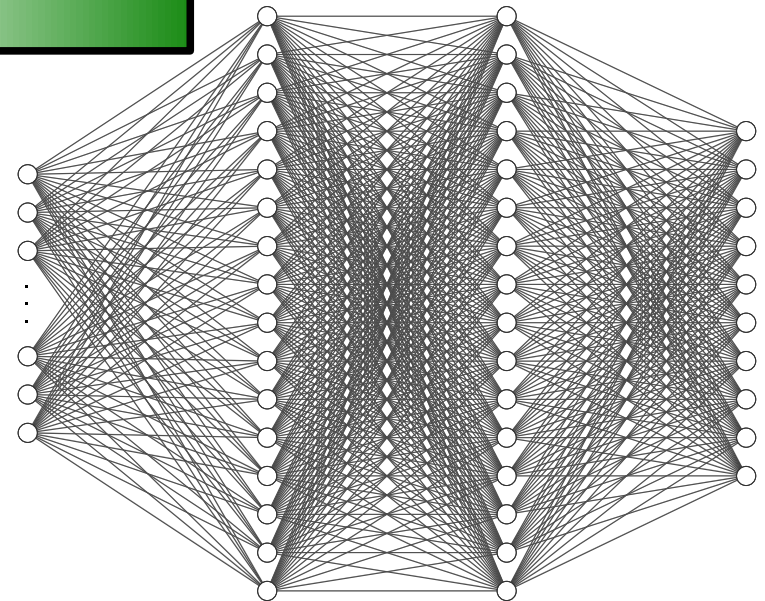
# Neuronale Netze

# Inspired by Nature



# das künstliche Neuron

- zunächst einmal nur eine “Stelle” / ein Platzhalter für einen Wert (= **Aktivierung**)
- Wertebereich z.B.  $[0;1]$  
- Eingabeneuronen, Ausgabeneuronen und innere Schichten

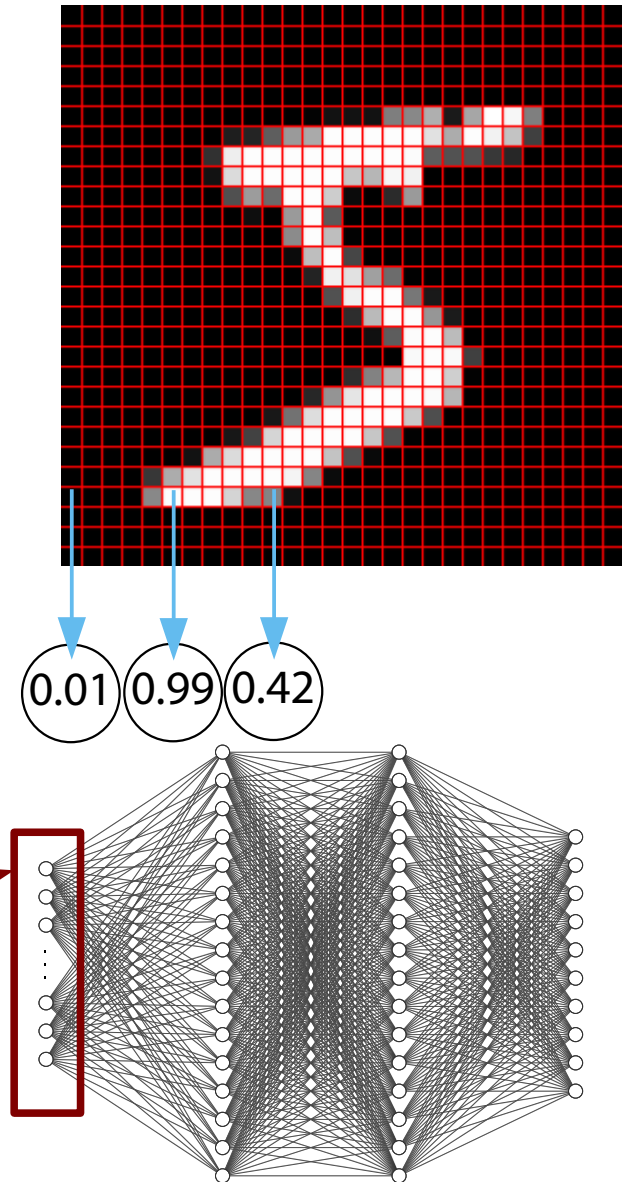




# das (künstliche) Neuron

- **Eingabeneuronen**

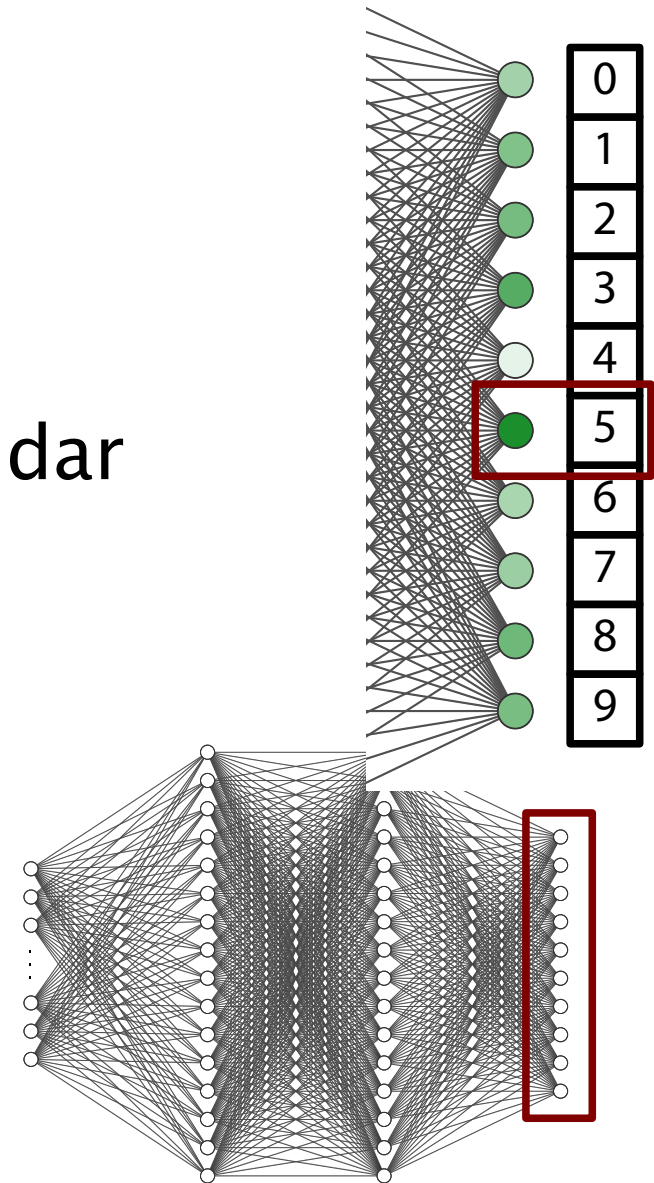
- repräsentieren die Eingabe
- Beispiel Handschrifterkennung:
- ein Eingabeneuron pro Bildpunkt
- speichert den Grauwert (schwarz  $\rightarrow$  0.0, weiß  $\rightarrow$  1.0)
- $28 \times 28$  Eingabeneuronen = Eingabeschicht mit 784 Neuronen



# das (künstliche) Neuron

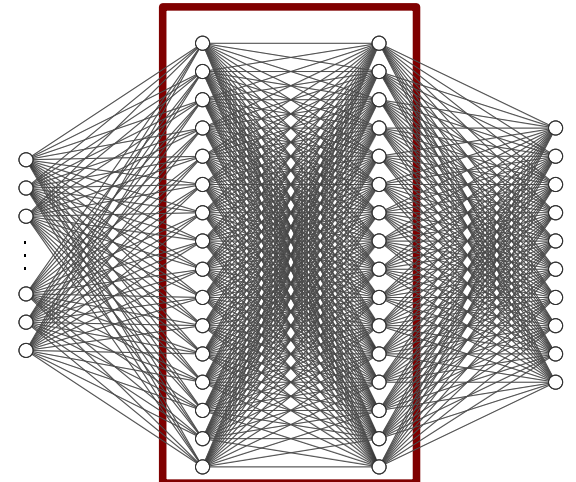
- **Ausgabeneuronen**

- stellen das “Ergebnis” des Netzes dar
- für Klassifikation:  
ein Neuron pro Klasse
- Wertebereich wieder  $[0;1]$   
(Grüntöne)  
→ stärkste Aktivierung gewinnt

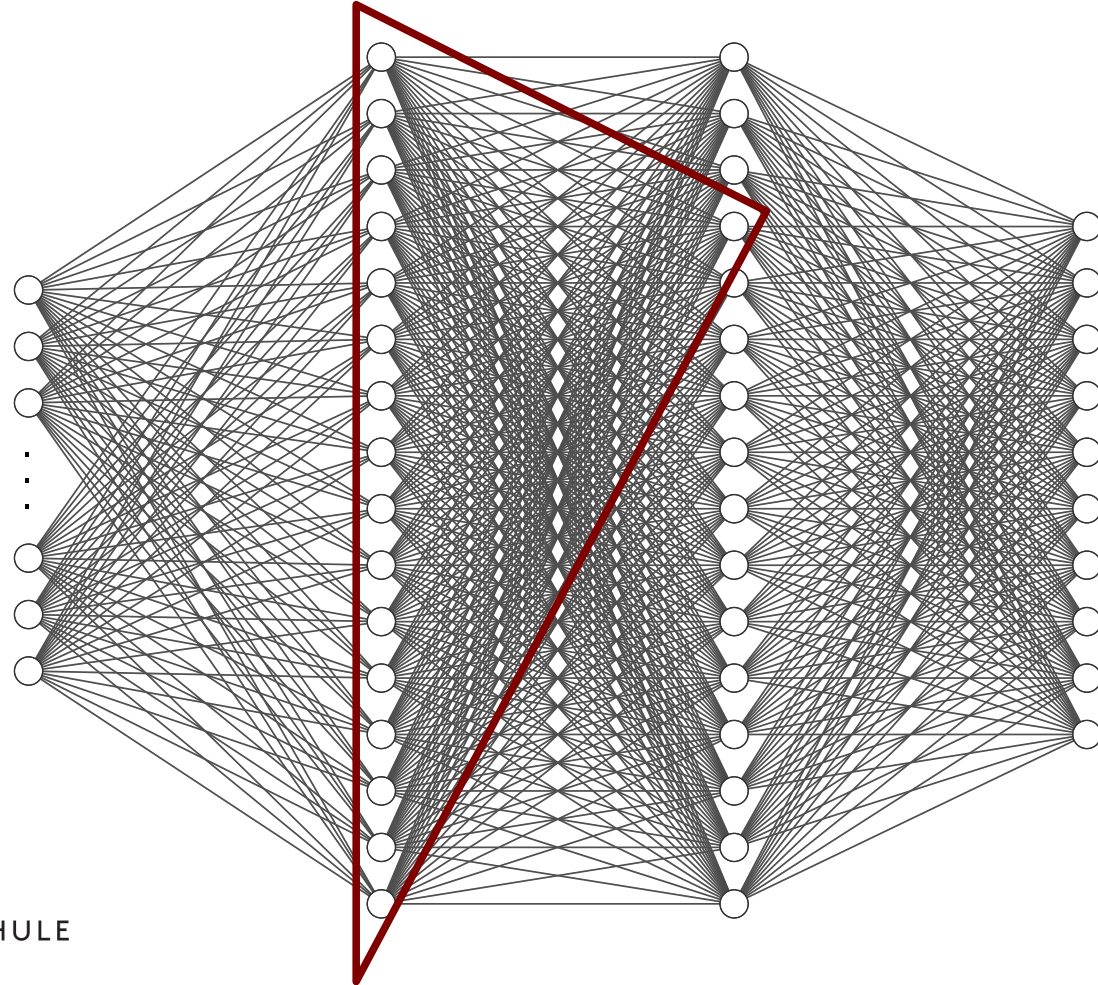


# das (künstliche) Neuron

- **innere Neuronen**
  - ermöglichen Rekombination und Abstraktion der Eingabedaten



# das vernetzte Neuron



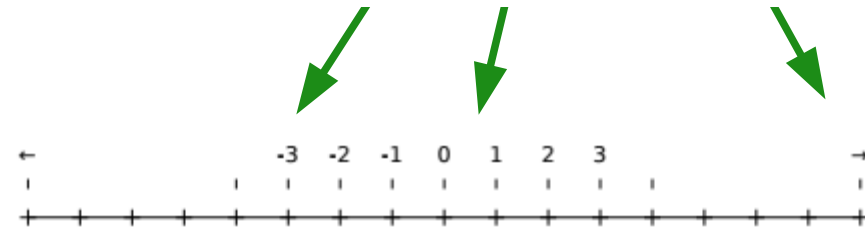
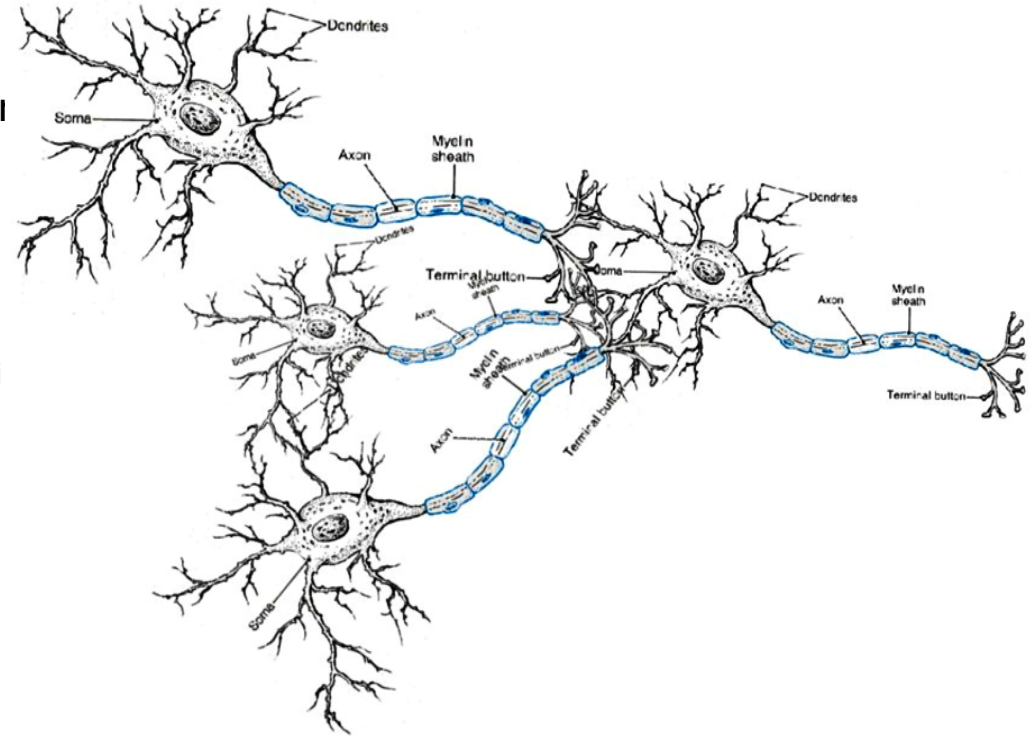
# das vernetzte Neuron

- mit allen Neuronen der Vorgängerschicht verbun

- Verbindungen sind gewichtet:  $w_1, \dots, w_n$
- Aktivierung bestimmt sich aus Summe der gewichteten Eingabeaktivierungen

$$a_1 \cdot w_1 + a_2 \cdot w_2 + a_3 \cdot w_3 + \dots + a_n \cdot w_n = \sum_{i=1..n} a_i \cdot w_i$$

- Summe kann beliebig groß werden  
→ Wertebereich **soll** aber  $[0;1]$  sein





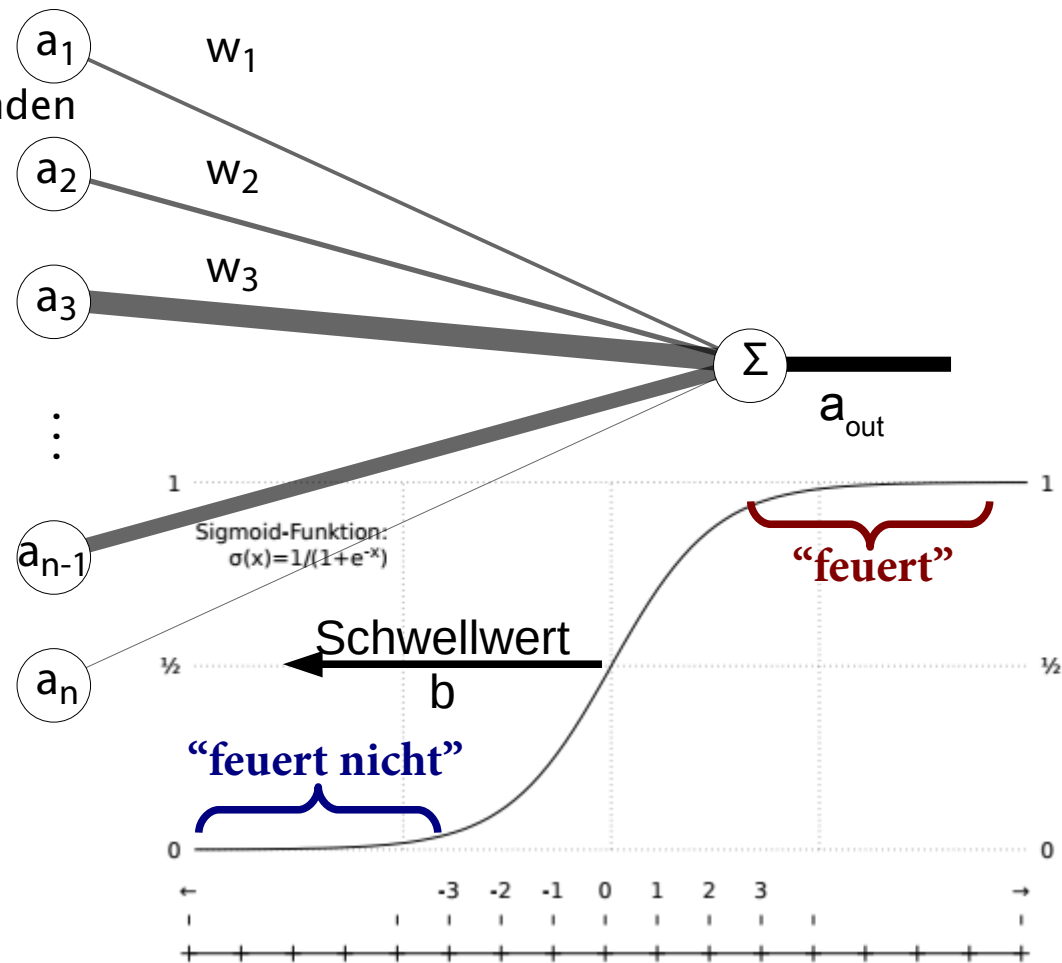
# das vernetzte Neuron

- mit allen Neuronen der Vorgängerschicht verbunden

- Verbindungen sind gewichtet:  $w_1, \dots, w_n$
- Aktivierung bestimmt sich aus Summe der gewichteten Eingabeaktivierungen

$$a_1 \cdot w_1 + a_2 \cdot w_2 + a_3 \cdot w_3 + \dots + a_n \cdot w_n = \sum_{i=1..n} a_i \cdot w_i$$

- Summe kann beliebig groß werden
- Wertebereich **soll**  $[0;1]$  sein
  - “Stauchung” auf  $[0;1]$  durch nicht-lineare Aktivierungsfunktion
- Schwellwert  $b$  stellt Sensibilität des Neurons ein



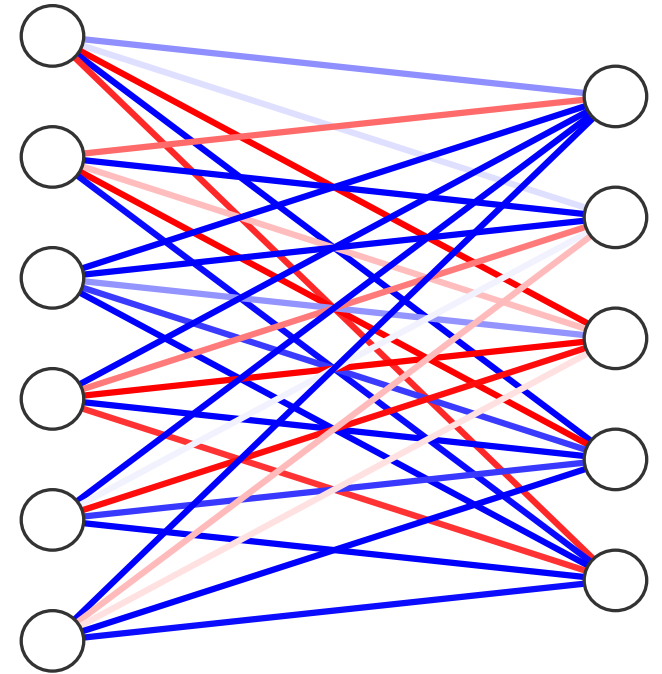
$$a_{out} = \sigma\left(\sum_{i=1..n} a_i \cdot w_i + b\right)$$

# Das Neuron als Funktion

- ein (künstliches) Neuron ist eine Funktion mit mehreren Eingaben, einigen Parametern, und einer Ausgabe (=Aktivierung).

# Die Neuronenschicht als Funktion

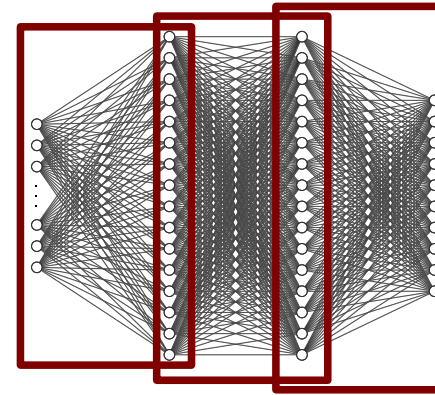
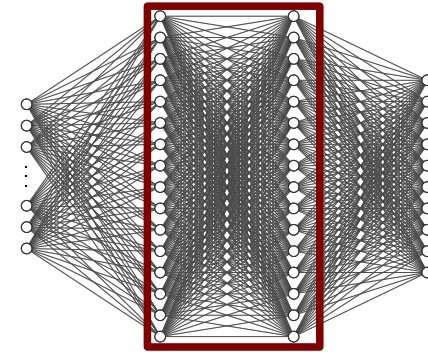
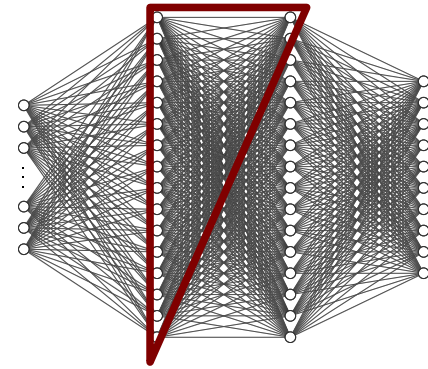
- aktuelle Schicht:  $m$  Neuronen
- Vorgängerschicht:  $n$  Neuronen mit Aktivierungsvektor  $x$
- $m \cdot n$  Gewichte von jedem der  $n$  zu jedem der  $m$  Neuronen  
→ Gewichtsmatrix  $W$
- zusätzlich  $m$  Schwellwerte  
→ Schwellwertvektor  $b$
- Aktivierung der aktuellen Schicht:  $y = \sigma(Wx + b)$
- Matrixoperationen sind sehr schnell mit moderner Hardware (insbesondere Graphikkarten)
- Deep-Learning-Toolkits “denken in Schichten”, nicht bloß in einzelnen Neuronen





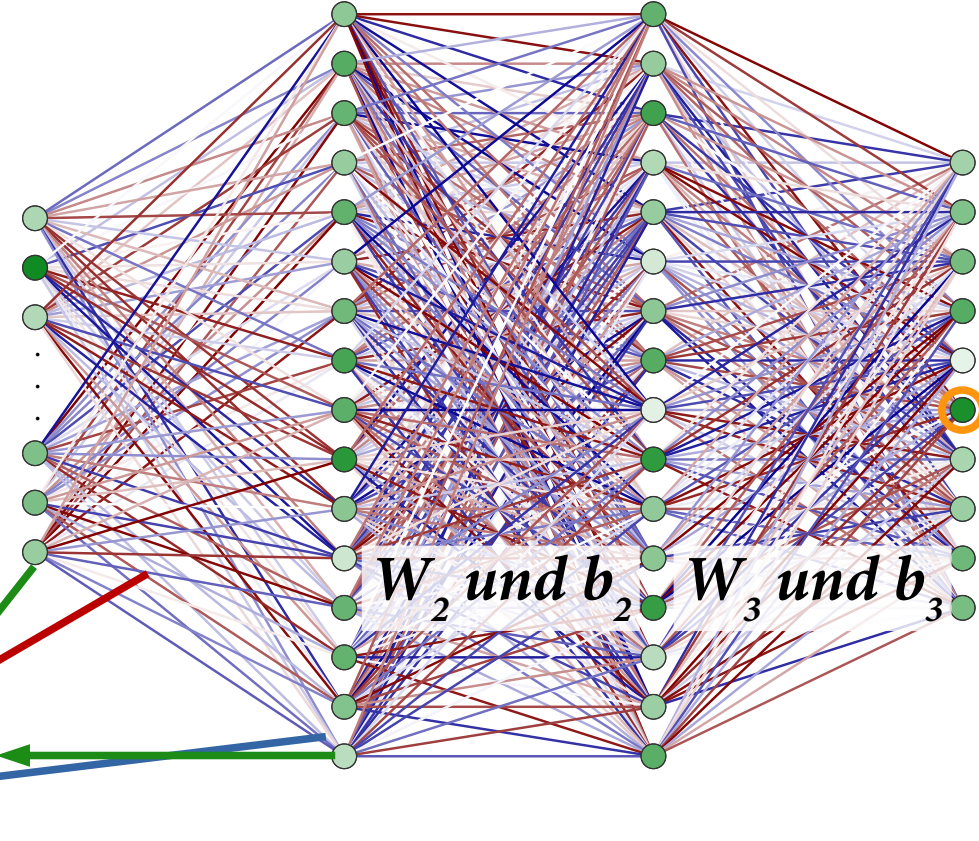
# Neuronales Netz als Funktion

- jedes Neuron kann als Funktion aufgefasst werden:  
Ausgangsaktivierung =  $f(\text{Eingangsaktivierungen})$
- jede Schicht kann als Funktion aufgefasst werden, bei der mehrere Neuronen parallel berechnet werden:  
Ausgangsaktivierungen =  $f(\text{Eingangsaktivierungen})$
- das ganze Netz kann als Funktion aufgefasst werden, bei denen Schichten verkettet werden:  
Ausgabeaktivierungen =  $f(\text{Eingabeaktivierungen})$



# Zwischenergebnis

- Neuronales Netz besteht aus Neuronen und gewichteten Verbindungen dazwischen
- Aktivierung
  - Eingabeschicht: Aktivierung  $a$  repräsentiert die Eingabedaten
  - innere Aktivierung ist gewichtete, reskalierte Kombination der Eingaben und abhängig von einem Schwellwert (bias)
  - für ein Neuron:  $a_{\text{out}} = \sigma(\sum_{i=1..n} a_i \cdot w_i + b)$
  - für jede Schicht:  $y = \sigma(W \cdot x + b)$  ;
  - für das gesamte Netz:  $a^{\text{out}} = \sigma(W_3 \cdot \sigma(W_2 \cdot \sigma(W_1 \cdot a^{\text{in}} + b_1) + b_2) + b_3)$
  - Ausgabeschicht zeigt Ergebnis an
- mit den “richtigen” Gewichten (und Schwellwerten) kann man (beliebige!) Eingabe-Ausgabe-Zusammenhänge abbilden
- wie “lernt” das Verfahren die richtigen Parameter ( $W$  und  $b$ )?



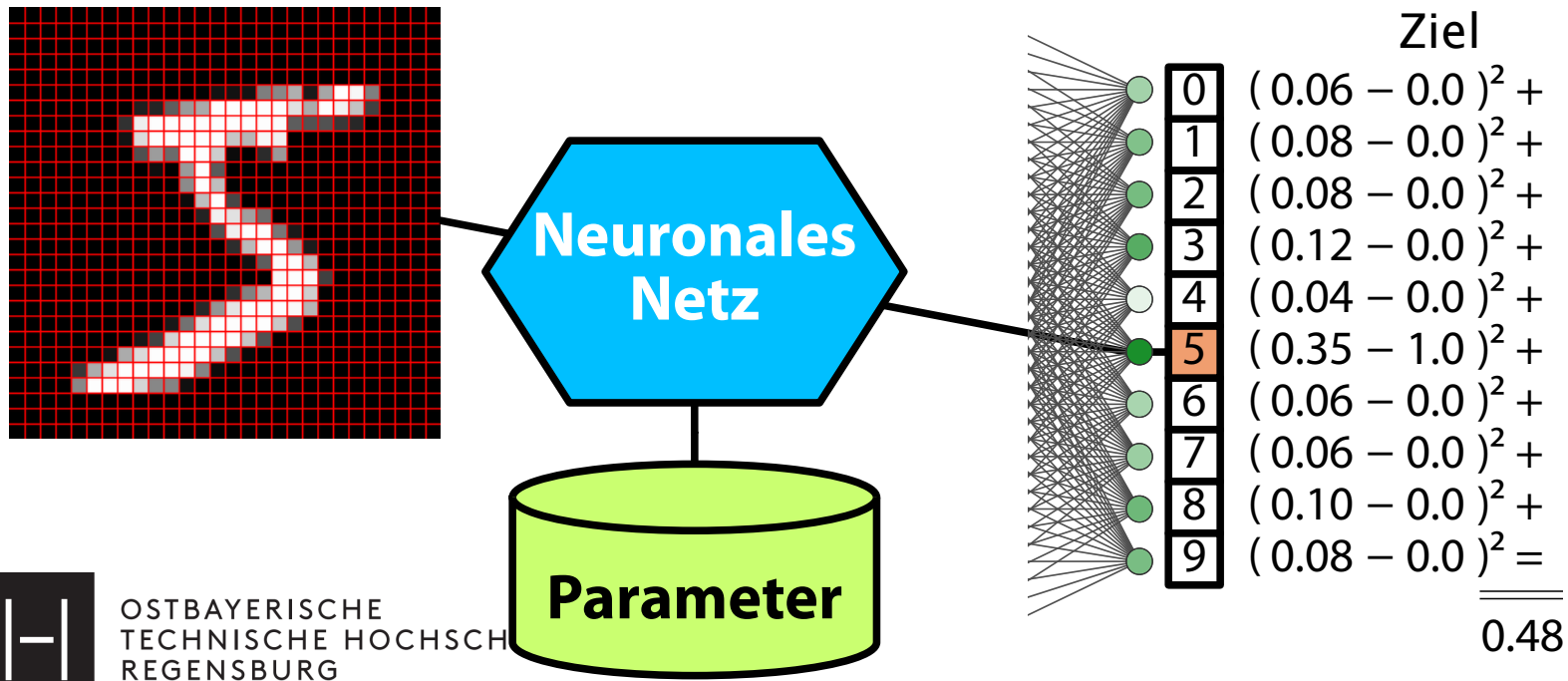
# Grundidee: Lernen im neuronalen Netz

- Qualität der Parametrisierung messen: *Kostenfunktion*
  - misst die “Imperfektion” der Parametrisierung für gegebene Daten
    - $kosten(\text{Parameter}, \text{Trainingsdaten}) \rightarrow \mathbb{R}^+$  (kleiner ist besser!)
  - 1. “bessere” von “schlechteren” Parametrisierungen unterscheiden
  - 2. wenn  $kosten()$  differenzierbar ist, können wir Verbesserungen schrittweise umsetzen

# Grundidee: Lernen im neuronalen Netz

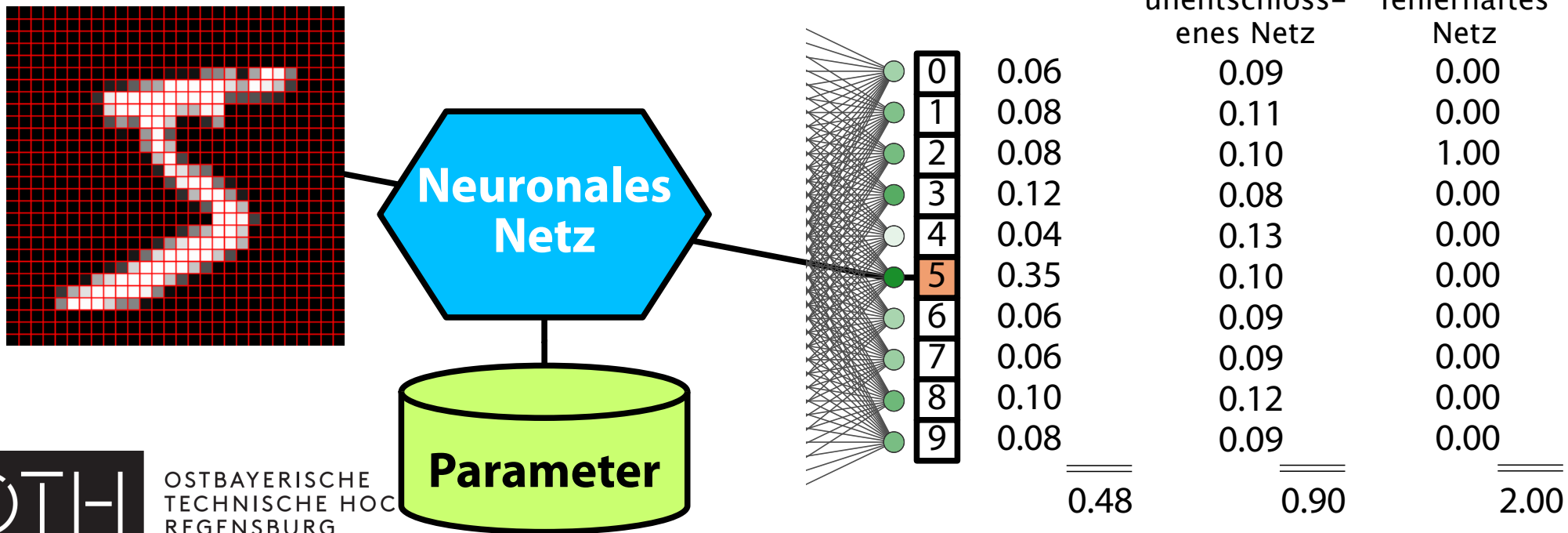
Kostenfunktion:

- z.B.: quadratische Abweichung von Zielwerten



# Grundidee: Lernen im neuronalen Netz

Kosten sind von den Parametern abhängig



# Gradientenabstiegsverfahren

- 1) Parameter irgendwie initialisieren,
- 2) partielle Ableitung der Fehlerfunktion nach allen Parametern ausrechnen,
- 3) dann Parameter einen kleinen Schritt in Richtung niedrigerer Fehler verändern,
- 4) gehe zu 2.

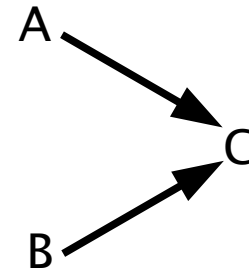
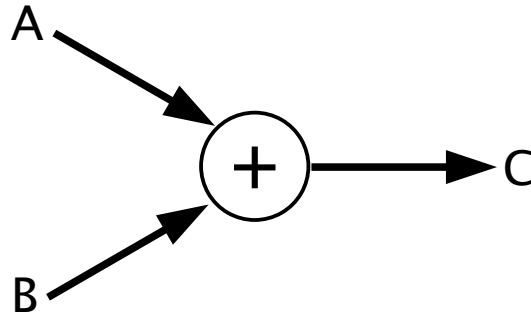
Aber wie geht das genau?  
Insb.: wie werden die Gradienten berechnet?

# Neuronales Netz als Berechnungsgraph

# Definition: Berechnungsgraph

Gerichteter Graph, der den Einfluss von Variablen (Parameter und Eingaben) auf zu berechnende (Zwischen-)Größen ausdrückt

$$C = A + B$$

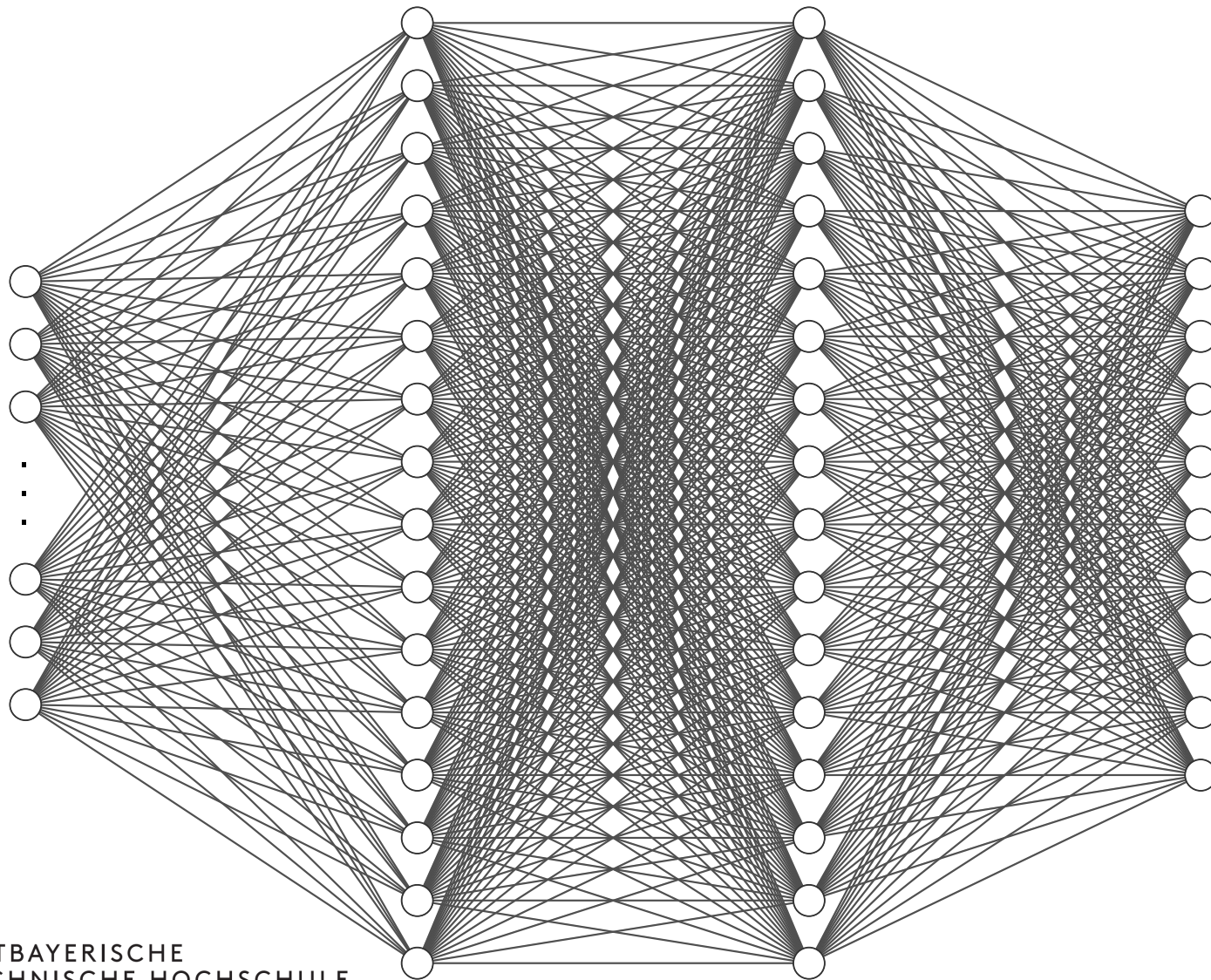


$$C = A + B$$

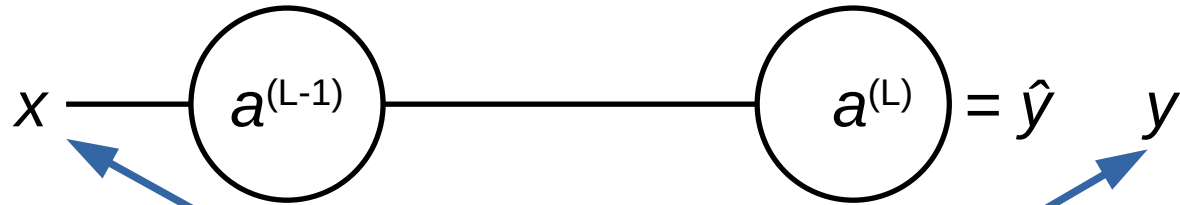


# Motivation: Berechnungsgraph

- der Berechnungsgraph ist praktisch um die Details des Trainings des NNs zu verdeutlichen (=Wiederholung auf andere Weise)
- Deep-Learning-Toolkits erstellen (und errechnen) automatisch den Berechnungsgraphen und die Gradienten für den Abstieg



# Ein minimales neuronales Netz nur zwei Neuronen!

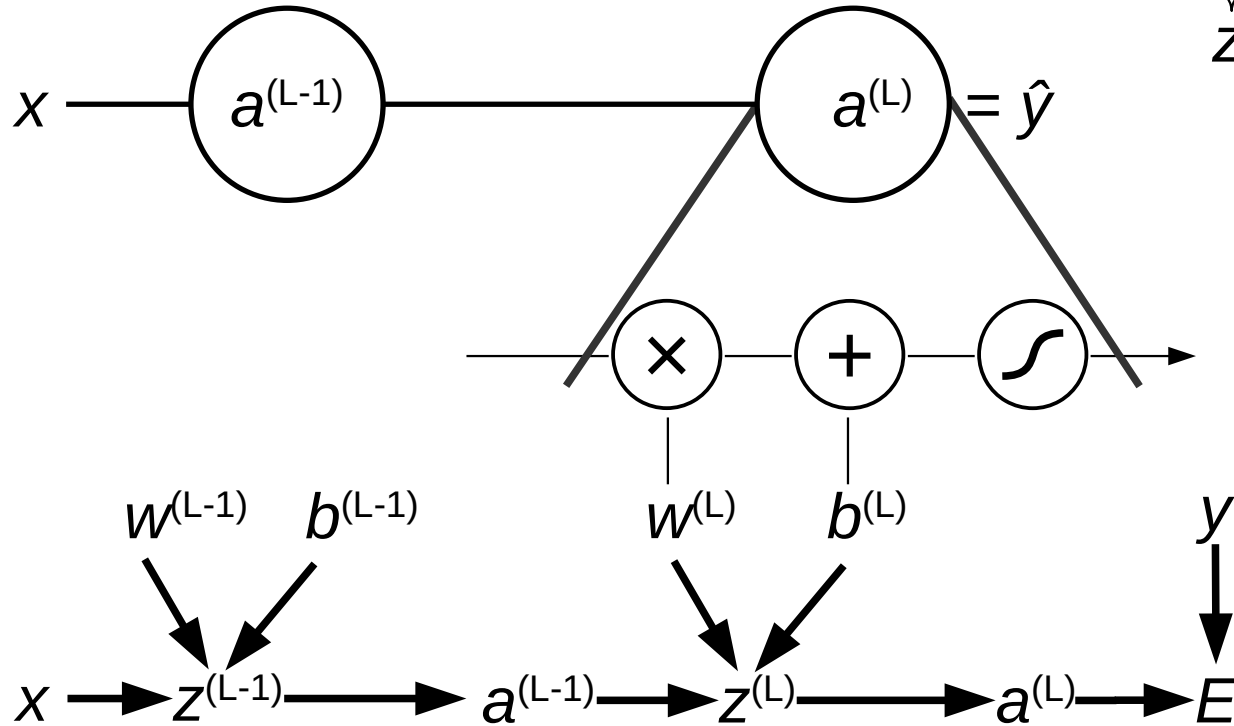


$$E = \frac{1}{2}(\hat{y} - y)^2$$

Gegeben Paare  $(x,y)$   
minimieren wir den  
Vorhersagefehler  $E$

# Ein minimales neuronales Netz

$$a^{(L)} = \sigma(\underbrace{w^{(L)}a^{(L-1)} + b^{(L)}}_{z^{(L)}})$$



$$E = \frac{1}{2}(a^{(L)} - y)^2$$

$$a^{(L)} = \sigma(z^{(L)})$$

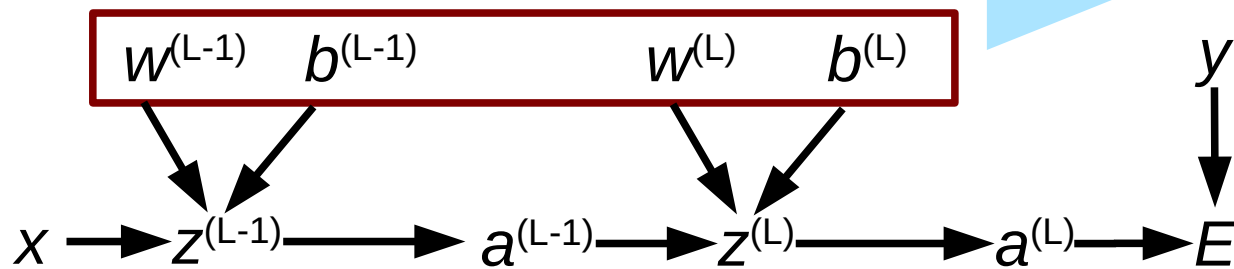
$$z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$$

Berechnungsgraph

# Ein minimales neuronales Netz

Netzwerkausgabe berechnen:

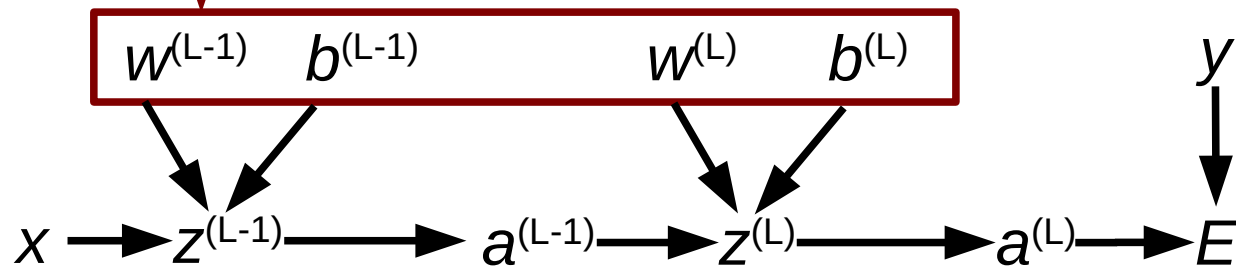
Graph von der Eingabe in Pfeilrichtung folgend bis  $a^{(L)}$  berechnen (unter Beachtung der Parameter und Rechenoperationen)



# Ein minimales neuronales Netz

Training: berechne partielle Ableitungen  $\frac{\nabla E}{\nabla \theta}$ ,  
des Fehlers, dann ändere die Parameter in  
Richtung kleinerer Fehler, wiederhole.

$$\frac{\nabla E}{\nabla \theta} := \frac{\partial E}{\partial w^{(L)}}, \frac{\partial E}{\partial b^{(L)}}, \frac{\partial E}{\partial w^{(L-1)}}, \frac{\partial E}{\partial b^{(L-1)}}$$



# Derivation rules

- Produktregel:  $(x^n)' = n \cdot x^{n-1}$

$$(x^n)' = \frac{d}{dx} x^n = n \cdot x^{n-1}$$

- Kettenregel:

$$((f \circ g)(x))' = (f(g(x)))' = f'(g(x)) \cdot g'(x)$$

$$((f \circ g)(x))' = \frac{d}{dx} (f \circ g)(x) = \frac{df(x)}{dg(x)} \frac{dg(x)}{dx}$$

- Exponentialregel:  $(e^x)' = e^x$

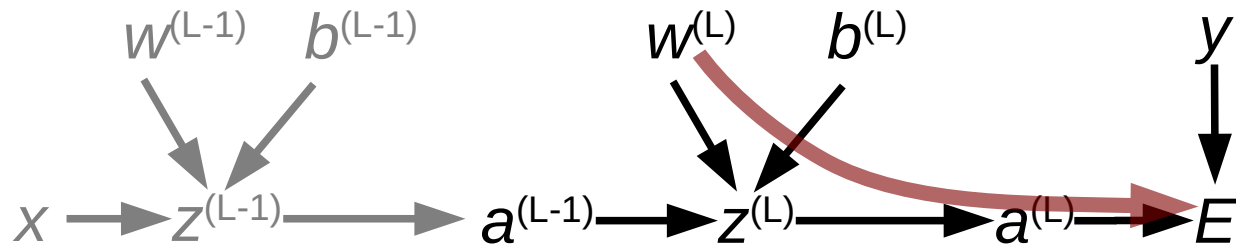
- $y = \sigma(x) = e^x / (e^x + 1)$

- $(\sigma(x))' = \dots = \sigma(x) \cdot (1 - \sigma(x)) = y \cdot (1 - y)$

$\frac{d}{dx}$  ?

# Ein minimales neuronales Netz

$$\frac{\partial E}{\partial w^{(L)}}$$



$$E = \frac{1}{2}(a^{(L)} - y)^2$$

$$a^{(L)} = \sigma(z^{(L)})$$

$$z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$$



# Ein minimales neuronales Netz

$$\frac{\partial E}{\partial w^{(L)}} = \frac{\partial E}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}} = (a^{(L)} - y) a^{(L)} (1 - a^{(L)}) a^{(L-1)}$$

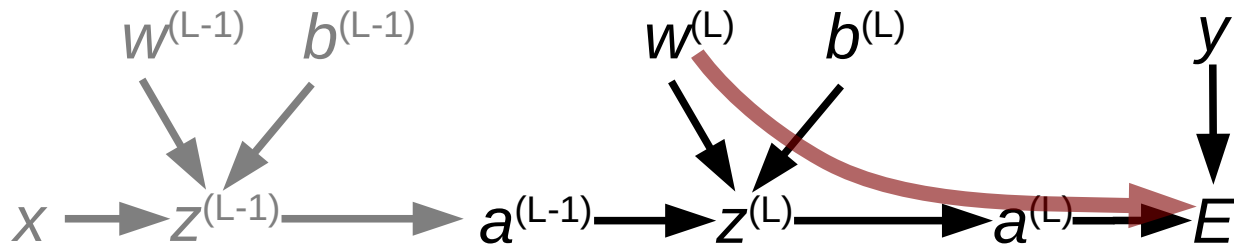
$$\frac{\partial E}{\partial a^{(L)}} = \frac{1}{2} \cdot 2(a^{(L)} - y) \cdot (1 - 0) = (a^{(L)} - y)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)}) = a^{(L)}(1 - a^{(L)}) \quad ; \quad \frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

$$E = \frac{1}{2}(a^{(L)} - y)^2$$

$$a^{(L)} = \sigma(z^{(L)})$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$



# Ein minimales neuronales Netz

$$\frac{\partial E}{\partial b^{(L)}} =$$

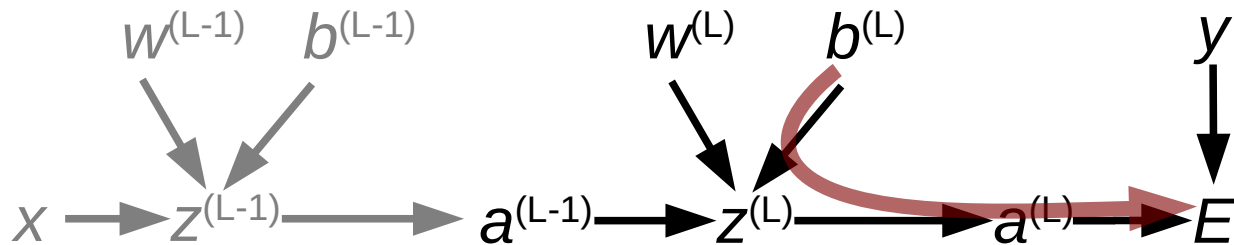
$$\frac{\partial E}{\partial a^{(L-1)}} =$$

$$\frac{\partial E}{\partial w^{(L-1)}} = \frac{\partial E}{\partial a^{(L-1)}} a^{(L-1)} (1 - a^{(L-1)}) a^{(L-2)}$$

$$E = \frac{1}{2}(a^{(L)} - y)^2$$

$$a^{(L)} = \sigma(z^{(L)})$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$



# Backpropagation: key points

- consider NN as computation graph
- forward–compute values in the graph  
using previous layer's activations for next
- backward–compute derivatives  
making use of previously computed derivatives  
*“propagating backwards the error signal”*

# Aufbau von NN-Toolkits

PyTorch, DyNet, Tensorflow, Keras/Theano, ...

- Spezifikation des Berechnungsgraphen durch Angabe der Vorwärtsberechnung:

```
y = torch.functional.sigmoid(x @ W + b)
```

```
y = dynet.logistic(x * W + b)
```

```
y = tf.math.sigmoid(x @ W + b)
```

- automatische Berechnung der Gradienten mittels Backpropagation (ohne, dass die Ableitungen der einzelnen Teilschritte spezifiziert werden müssen!)
- Variable sind keine Zahlentypen (oder Vektoren, wie in Numpy), sondern Objekte
  - zu den Werten der Zahlen wird zusätzlich ihr Gradient ausgerechnet, sobald für einen Zahl die Backpropagation aufgerufen wird (`.backward()`)
  - Vorwärtsberechnung sobald der Wert angefordert wird (*lazy evaluation* in DyNet: `.value()`) oder sofort (*eager evaluation*, PyTorch, TF)
  - Variable müssen üblicherweise aufwendiger initialisiert werden als "normal"
- Rechenoperationen sind Bibliotheksaufrufe, entweder **direkt** oder per **Operator-Overloading**

# Zusätzliche Features von NN-Toolkits

- Objektorientierung und Kapselung häufig genutzter Layers und Rechenoperationen in neuronalen Netzen:

```
self.lin_layer = nn.Linear(input_size, HIDDEN_DIM)
...
y = nn.functional.sigmoid(self.lin_layer(x))
```

- Integration unterschiedlicher Trainingsregimes für Gradientenabstieg
- Unterstützung von “Mini-Batching”, sodass mehrere Dateninstanzen gleichzeitig parallel verarbeitet werden
  - DyNet: automatisches Batching nebenläufiger Operationen
- automatische Auslagerung von Berechnungen auf die GPU (die besonders gut parallelisierbare Operationen verarbeitet)

Zu den Tasten!

[https://www.timobaumann.de/  
work/Main/FobiNeuNe](https://www.timobaumann.de/work/Main/FobiNeuNe)

Vielen Dank! Ihre Fragen?

[timo.baumann@oth-regensburg.de](mailto:timo.baumann@oth-regensburg.de)