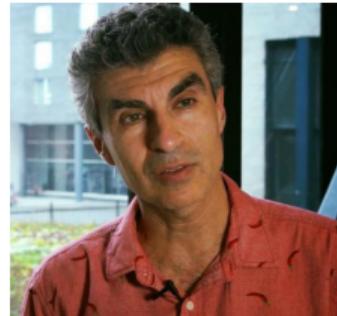


Künstliche Intelligenz (Sommersemester 2024)

Kapitel 06: Neuronale Netze

Prof. Dr. Adrian Ulges

Neuronale Netze: Material



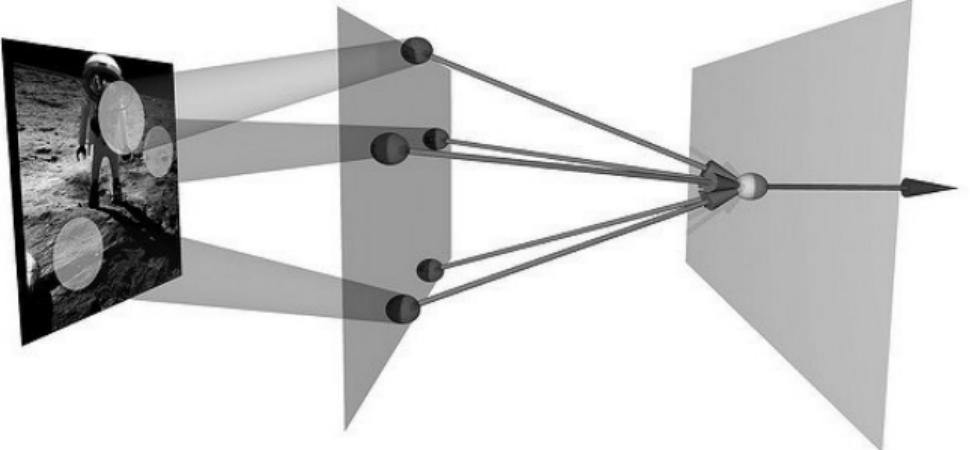
Bücher

- ▶ **Nielsen**: “Neuronale Netze und Deep Learning”
<http://neuralnetworksanddeeplearning.com>
- ▶ Goodfellow, **Bengio**, Courville: “Deep Learning”
<https://www.deeplearningbook.org>

Tools

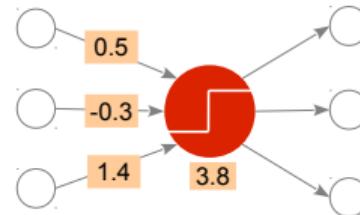
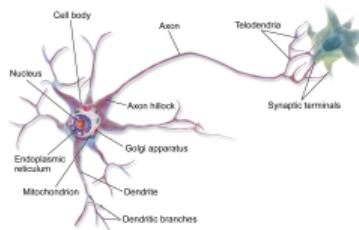
- ▶ PyTorch
<https://pytorch.org>

Neuronale Netze.... Bild: [1]



- ... bilden die Grundlage von “**Deep Learning**”, und somit die Grundlage der KI-Welle seit 2012.
- ... bilden die Informationsverarbeitung im **menschlichen Gehirn** mittels **Graphen** von vereinfachten **Neuronen** nach.
- ... definieren ihr Verhalten durch ihre **internen Gewichte**. Diese werden durch **überwachtes Lernen** bestimmt.

Inspiration: Biologische neuronale Netze



Biologische Neuronen (links)

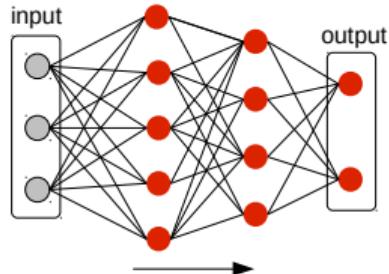
- ▶ ... sind mit vielen (z.B. 7000) anderen Neuronen verbunden.
- ▶ ... empfangen **elektrische Signale** von anderen Neuronen, kombinieren sie, und leiten sie ggf. weiter.
- ▶ ... passen sich durch Sättigung und Sensibilisierung an (= *Lernen*).

Künstliche Neuronen (rechts)

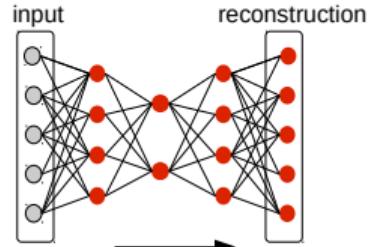
- ▶ ... sind eine sehr einfache Abstraktion biologischer Neuronen (*gewichtete Summe + nicht-lineare Aktivierungsfunktion*).
- ▶ ... definieren ihr Verhalten durch **Eingabegewichte** (orange).

Neuronale Netze: Architekturen

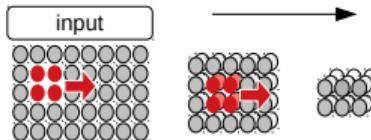
MLPs (Multilayer Perceptrons)



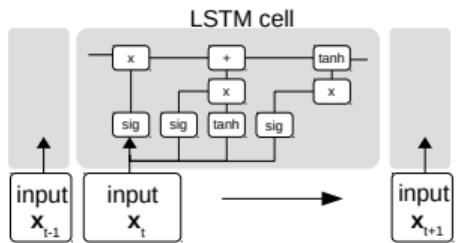
Autoencoders



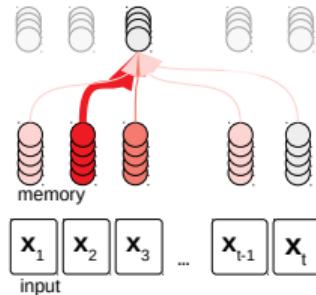
CNNs
(Convolutional Neural Networks)



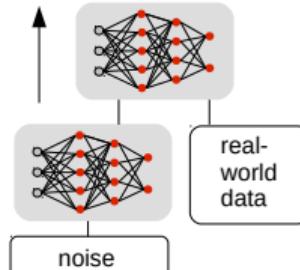
Recurrent Networks



Attention Models



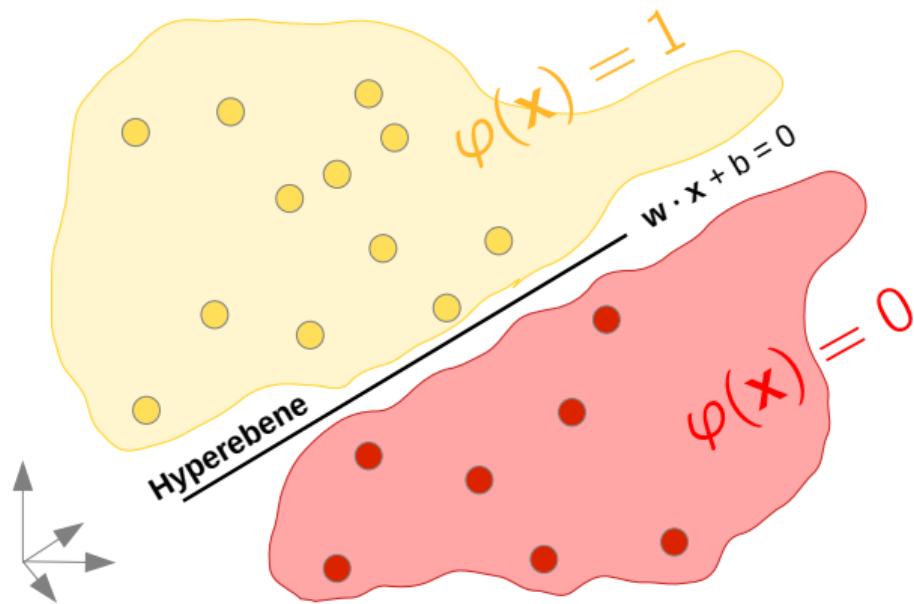
GANs
(Generative Adversarial Networks)



Outline

1. Neuronen
2. Von Neuronen zu Neuronalen Netzen
3. Training: Der Backpropagation-Algorithmus

Definition eines einzelnen Neurons φ



Definition eines einzelnen Neurons φ

Definition (Neuron)

Gegeben einen Eingabevektor $\mathbf{x} = (x_1, \dots, x_d) \in \mathbb{R}^d$ sowie Parameter $\theta = (w_1, \dots, w_d, b)$ und eine Aktivierungsfunktion $g : \mathbb{R} \rightarrow \mathbb{R}$, definieren wir die Ausgabe eines Neurons $\varphi_\theta : \mathbb{R} \rightarrow \mathbb{R}$ als

$$\varphi_\theta(\mathbf{x}) = a = g\left(\underbrace{w_1 x_1 + w_2 x_2 + \dots + w_d x_d + b}_{=: z}\right)$$

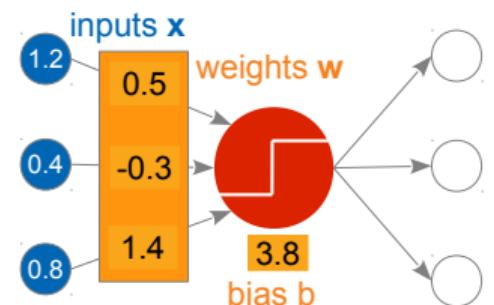


$$= g(\mathbf{w} \cdot \mathbf{x} + b).$$

$$z \\ 1,2 \cdot 0,5 + 0,4 \cdot (-0,3) + \dots$$

Anmerkungen und Graphische Darstellung

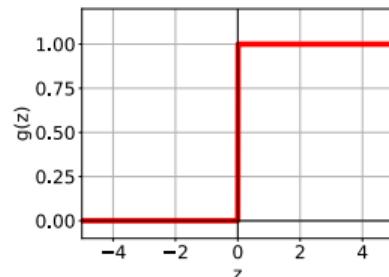
- Wir nennen w_1, \dots, w_d die **Gewichte** des Neurons, und b seinen **Bias**.
- z bezeichnet die **gesammelte Eingabeenergie** des Neurons, und a seine Ausgabe.



Andere Aktivierungsfunktionen

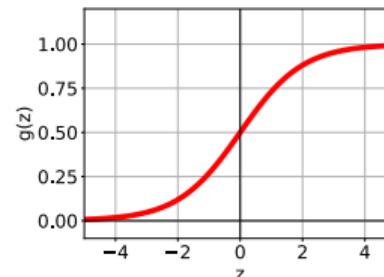
Die Aktivierungsfunktion g muss **nicht** die Schritt-Funktion sein: Es gibt viele **andere Optionen**, z.B. ...

Schritt-Funktion



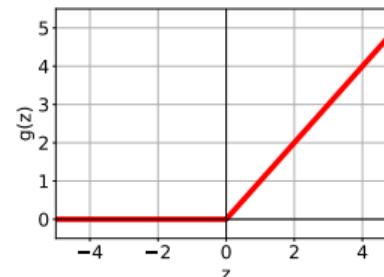
$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{sonst.} \end{cases}$$

Sigmoid



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

RELU



$$g(z) = \max(0, z)$$

Anmerkungen

- ▶ Aktivierungsfunktionen müssen **nicht-linear** sein.
- ▶ Im Moment bleiben wir beim **Sigmoid**.

Andere Aktivierungsfunktionen (cont'd)

- ▶ Eine weitere häufig verwendete Aktivierungsfunktion ist der **Softmax**. Dieser wird nicht auf einen **einzigem Wert** z angewendet, sondern auf einen **Vektor** (z_1, \dots, z_m) .
- ▶ Der Softmax produziert **Wahrscheinlichkeiten** (*nicht-negativ, Summe=1*):

$$g(z_1, \dots, z_m) = \left(\frac{e^{z_1}}{\sum_i e^{z_i}}, \frac{e^{z_2}}{\sum_i e^{z_i}}, \dots, \frac{e^{z_m}}{\sum_i e^{z_i}} \right)$$

Beispiel

$$g(1, 3, 1, 7) \approx (2\%, 11\%, 2\%, 85\%)$$

$$g(-3, 0, 0.5, -15) \approx (2\%, 37\%, 61\%, 0\%)$$

Anmerkungen

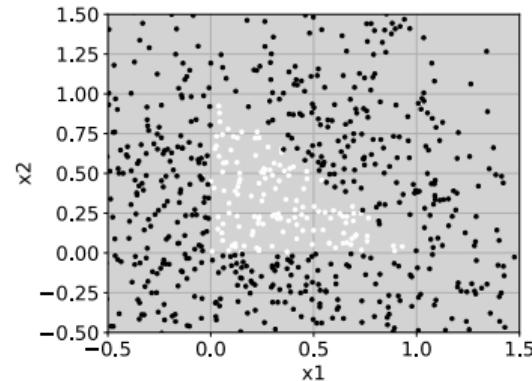
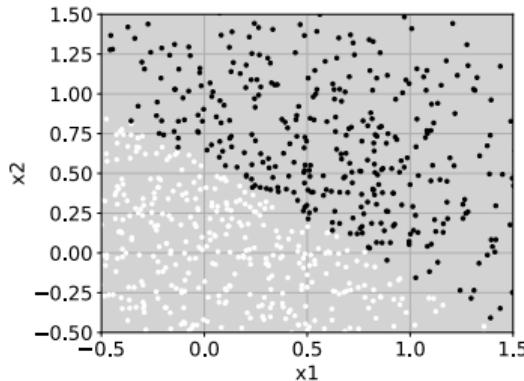
- ▶ Da neuronale Netze oft **Wahrscheinlichkeiten ausgeben**, wird der Softmax gerne in der **letzten Schicht** verwendet.



Outline

1. Neuronen
2. Von Neuronen zu Neuronalen Netzen
3. Training: Der Backpropagation-Algorithmus

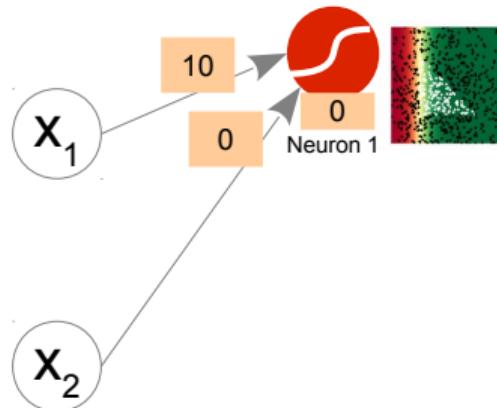
Neuronen: Ausdrucksmächtigkeit



Welches Problem kann ein einzelnes Neuron lösen?

- ▶ Nur das linke! Auf der rechten Seite sind die beiden Klassen "schwarz" und "weiß" nicht linear separierbar (*d. h., keine Hyperebene – und somit kein Neuron φ – kann die Klassen perfekt trennen*).
- ▶ Wie könnten wir mehrere Neuronen verbinden, um das Problem zu lösen?

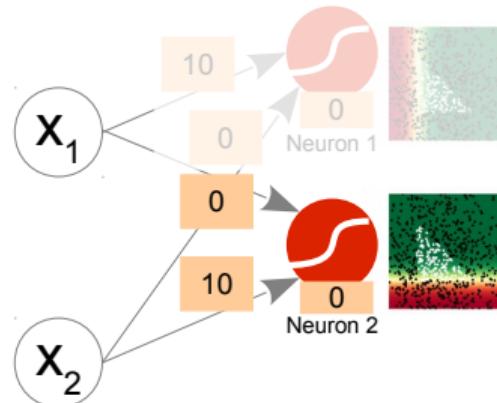
Beispiel: Lösung



Neuron 1

- ▶ Neuron 1 rechnet: $g(10 \cdot x_1 + 0 \cdot x_2 + 0)$.
- ▶ Es modelliert die linke Kante des Dreiecks.
- ▶ Hinweis: **Sigmoid**-Aktivierungen (= sanfte Übergänge).

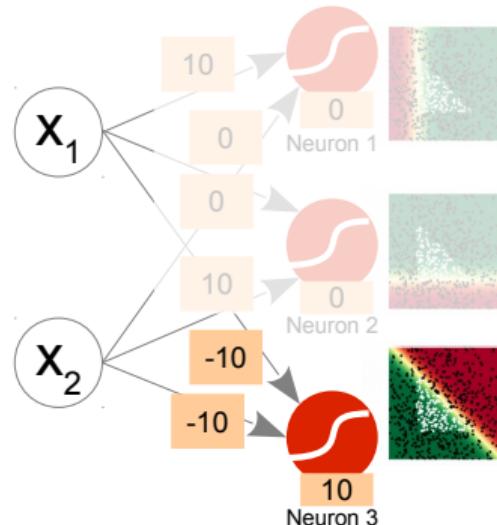
Beispiel: Lösung



Neuron 2

- ▶ Neuron 2 rechnet: $g(0 \cdot x_1 + 10 \cdot x_2 + 0)$.
- ▶ Es modelliert die untere Kante des Dreiecks.

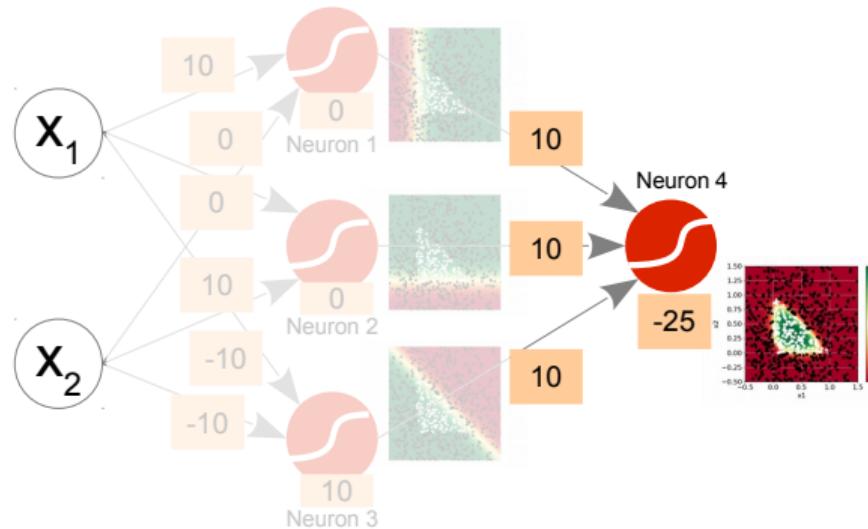
Beispiel: Lösung



Neuron 3

- ▶ Neuron 3 rechnet: $g(-10 \cdot x_1 - 10 \cdot x_2 + 10)$.
- ▶ Es modelliert die diagonale Kante des Dreiecks.

Beispiel: Lösung



Neuron 4

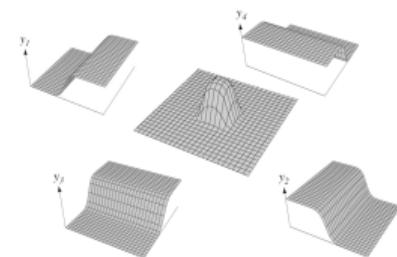
- ▶ Neuron 4 empfängt die Signale a_1, a_2, a_3 der Neuronen 1-3. Diese zeigen an ob die Eingabe x auf der "Innenseite" jeder Kante ist.
- ▶ Neurons 4 rechnet: $g(10a_1 + 10a_2 + 10a_3 - 25)$.
- ▶ Neuron 4 approximiert ein **logisches AND**.

Können neuronale Netze alles modellieren?

Satz (Ausdrucksmächtigkeit neuronaler Netze)

Indem wir mehrere Neuronen mit sigmoid-Aktivierungen verbinden, können wir jede kontinuierliche Funktion $f : [0, 1]^d \rightarrow [0, 1]$ mit infinitesimalem Fehler approximieren.

“In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube. [...] In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity.”
(Cybenko., G. [2])

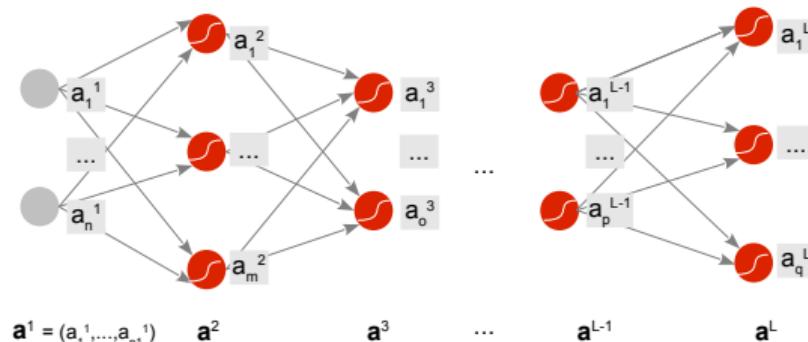


Anmerkungen

- Das bedeutet: Wir können tatsächlich jedes Klassifikationsproblem lösen.
- **Offene Frage:** Modellauswahl → Wie viele Neuronen benötigen wir?
- **Offene Frage:** Lernen → Wie finden wir die Lösung?

Das MLP: Notation

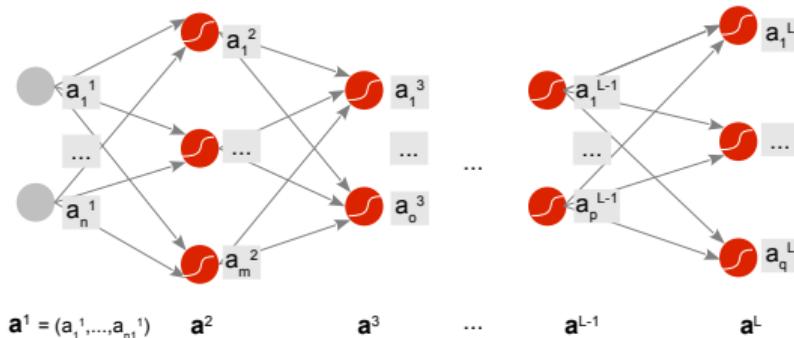
Wir fokussieren uns zunächst auf das “Einsteiger-Netz”, das sogenannte **Multilayer-Perceptron** (MLP). Dieses besteht aus vollständig verbundenen, vorwärtsgerichteten **Schichten** von Neuronen.



Notation

- $z_i^l/a_i^l/b_i^l$ bezeichnen Eingabeenergie/Ausgabe/Bias des *i*-ten Neurons in Schicht *l*.
- Insgesamt gibt es **L Schichten** $l = 1, \dots, L$ mit n_1, \dots, n_L Neuronen.
- w_{ji}^l ist das **Gewicht** der Kante zwischen Neuron *i* in Schicht $(l-1)$ und Neuron *j* in Schicht *l*.

Das MLP: Notation

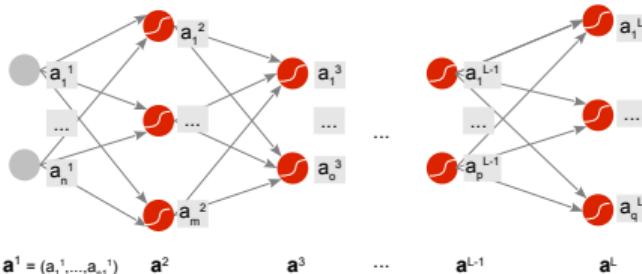


Einfachere Notation: Vektoren/Matrizen

- Wir sammeln die Aktivierungen jeder Schicht in einem **Vektor**: $\mathbf{a}^l := (a_1^l, \dots, a_{n_l}^l)$
- Ähnliche Vektoren gibt es für den Bias \mathbf{b}^l und Eingabeenergie \mathbf{z}^l .
- Die Gewichte zwischen den Schichten $l-1$ und l bilden eine **Matrix**:

$$W^l := \begin{pmatrix} w_{1,1}^l & \dots & w_{1,n_{l-1}}^l \\ \dots & \dots & \dots \\ w_{n_l,1}^l & \dots & w_{n_l,n_{l-1}}^l \end{pmatrix} \in \mathbb{R}^{n_l \times n_{l-1}}$$

Das MLP: Notation



- Gegeben den Eingabevektor $\mathbf{x} := \mathbf{a}^1$, wird das Signal schichtweise durch das Netzwerk propagiert (*Forward Pass*).
- In jeder Schicht sammeln wir die **Eingabeenergie** aller Neuronen ...

$$\mathbf{z}^l = W^l \cdot \mathbf{a}^{l-1} + \mathbf{b}^l$$

- ... und wenden die **Aktivierungsfunktion g** elementweise auf \mathbf{z}^l an:

$$\mathbf{a}^l = g(\mathbf{z}^l) = (g(z_1^l), g(z_2^l), \dots)$$

- Das **Endergebnis** entspricht der Ausgabe der letzten Schicht, d.h. $f_{\theta}(\mathbf{x}) := \mathbf{a}^L$.
- Die **Parameter** des Netzes sind Gewichte und Biases aller Schichten:
 $\theta = (W^2, \dots, W^L, \mathbf{b}^2, \dots, \mathbf{b}^L)$.

Forward Pass: Beispiel

Gegeben sei das Netz unten, mit Schritt-Aktivierungsfunktion g .

Wir geben $x = (1, 0)^T$ ein, und berechnen die Ausgabe a^3 .

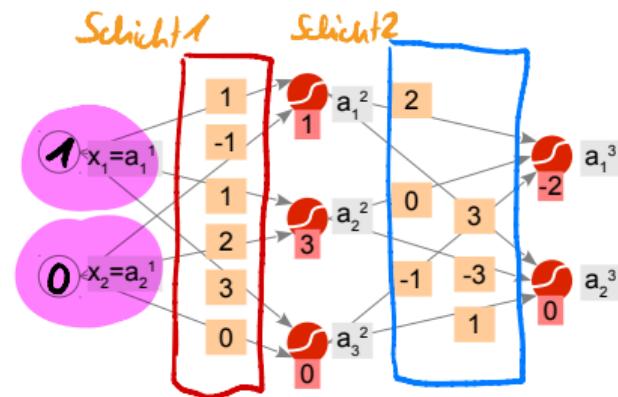
$$\omega^2 = \begin{pmatrix} 1 & -1 \\ 1 & 2 \\ 3 & 0 \end{pmatrix}; b^2 = \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix}; \omega^3 = \begin{pmatrix} 2 & 0 & -1 \\ 3 & -3 & 1 \end{pmatrix}; b^3 = \begin{pmatrix} -2 \\ 0 \end{pmatrix}$$

$$z^2 = \omega^2 \cdot a^1 + b^2 = \begin{pmatrix} 1 & -1 \\ 1 & 2 \\ 3 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 5 \\ 3 \end{pmatrix}$$

$$a^2 = g(z^2) = \underline{\underline{\begin{pmatrix} 1 \\ 1 \end{pmatrix}}}$$

$$z^3 = \omega^3 \cdot a^2 + b^3 = \begin{pmatrix} 2 & 0 & -1 \\ 3 & -3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} -2 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

$$a^3 = g(z^3) = \underline{\underline{\begin{pmatrix} 0 \\ 1 \end{pmatrix}}}$$

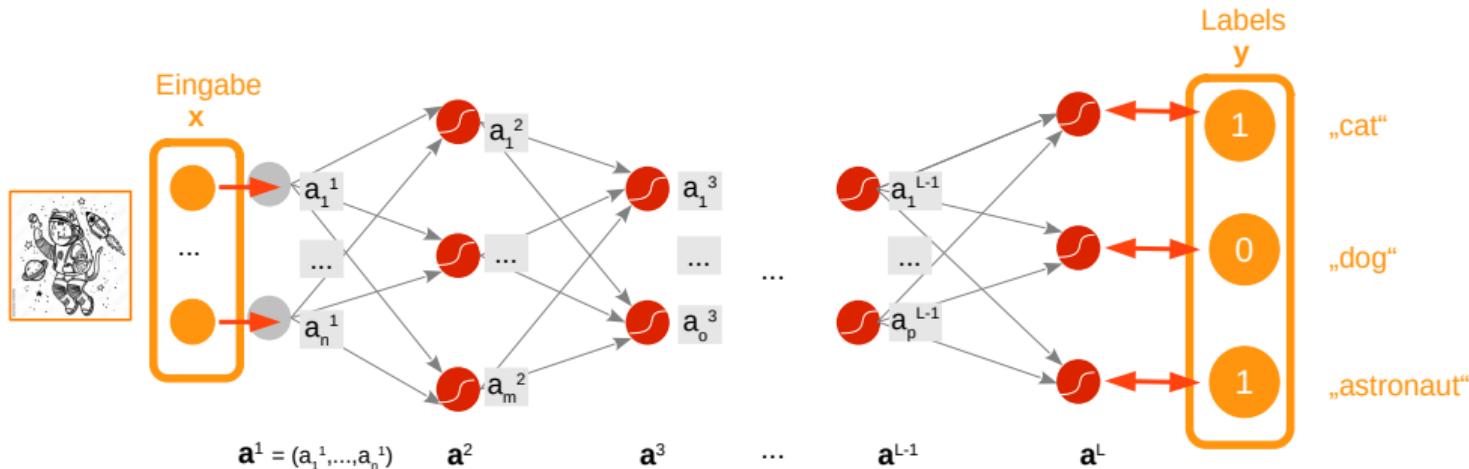




Outline

1. Neuronen
2. Von Neuronen zu Neuronalen Netzen
3. Training: Der Backpropagation-Algorithmus

Lernen: Mechanik



- Das Training ist **überwacht**: Die **Trainingsdaten** bestehen aus den **Eingaben** (=Merkmalsvektoren) x_1, \dots, x_n und den entsprechenden **erwarteten Ausgaben** (=Labels) y_1, \dots, y_n .
- **Frage:** Wie finden wir Gewichte und Biases, so dass das Netz – gegeben x_i – die **gewünschte Ausgabe** erzeugt, d.h. $f_\theta(x_i) \approx y_i$?

Lernen: Mechanik

Wir **initialisieren** die Gewichte und Schwellenwerte des Netzes **zufällig** und alternieren dann zwischen zwei Phasen:

1. Forward Pass:

Wähle ein Trainingsbeispiel x als Eingabe. Berechne die Aktivierungen aller Neuronen. Wir erhalten einen Ausgabevektor $\hat{y} = f_\theta(x)$.

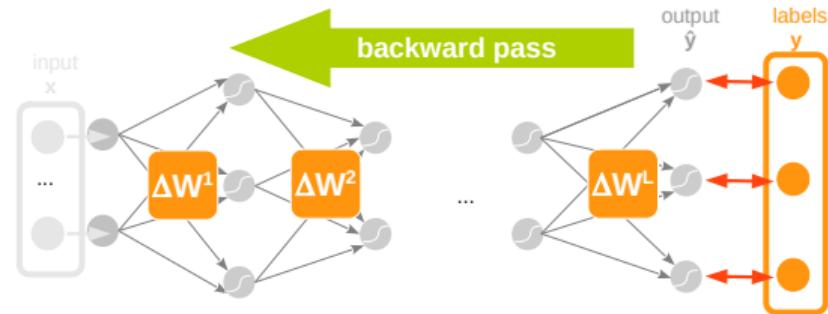
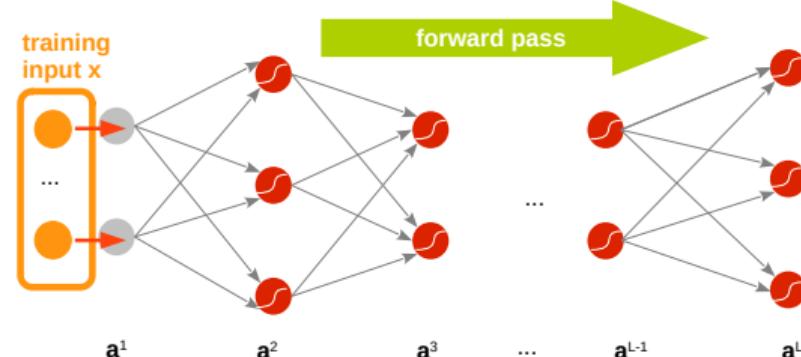
2. Backward Pass:

Vergleiche \hat{y} mit den Labels y und passe Gewichte+Biases “sinnvoll” an.

für alle $l=L,\dots,2$:

$$\mathbf{W}^l := \mathbf{W}^l + \Delta \mathbf{W}^l$$

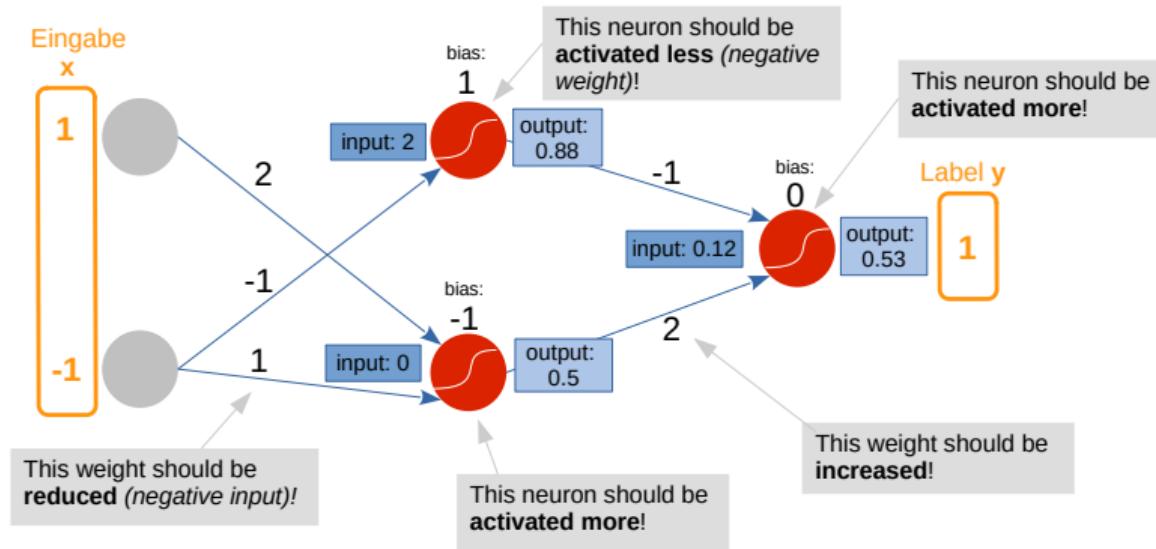
$$\mathbf{b}^l := \mathbf{b}^l + \Delta \mathbf{b}^l$$



Lernen: Beispiel

Ein (winziges) Beispielnetzwerk

- ▶ Eingabe: $x = (1, -1)$, Label: $y = (1)$.
- ▶ **Forward Pass:** Wir berechnen die Ausgabe des Netzwerks.
- ▶ **Backward Pass:** Wie aktualisieren wir die Gewichte und Biases?



Der Backpropagation-Algorithmus

Backpropagation ist der Schlüssel-Lernalgorithmus in neuronalen Netzen:

```

1  function BACKPROPAGATION( $x_1, \dots, x_n, y_1, \dots, y_n$ ):
2       $W^2, W^3 \dots, W^L, b^2, b^3, \dots, b^L := \text{initialize}()$ 
3      repeat
4          ( $x, y$ ) := choose_sample()
5          feed  $x$  to the net, compute activations  $z^1, \dots, z^L$  and  $a^1, \dots, a^L (= \hat{y})$  // forward pass
6          for layer  $l = L, L-1, \dots, 2$ :
7               $w_{ji}^l := w_{ji}^l + \Delta w_{ji}^l$       for all  $i, j$       // update weights
8               $b_j^l := b_j^l + \Delta b_j^l$       for all  $j$       // update biases
9      until weights+biases stop changing

```

- ▶ Schlüsselfrage: Wie berechnet man die Aktualisierungen Δw_{ji}^l und Δb_j^l ?
- ▶ Antwort: Minimiere eine Zielfunktion (oder **Loss**) \mathcal{L} :

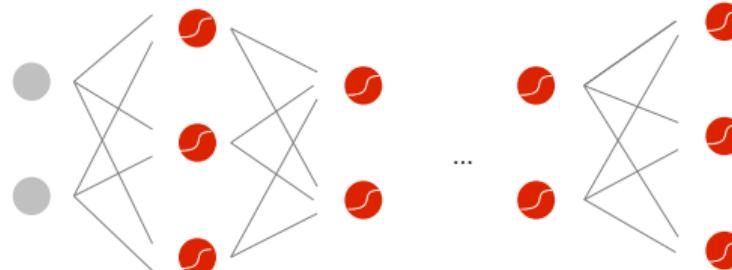
$$\mathcal{L}(\theta) := \frac{1}{2} \cdot \sum_{k=1}^{n_L} (\hat{y}_k - y_k)^2$$

- ▶ Wir minimieren \mathcal{L} mit Hilfe von **Gradientenabstieg**!

Backpropagation im MLP



$$L(\theta) \quad \begin{array}{c} \nearrow \\ \text{Netzwerk} \\ \searrow \end{array} \quad \theta$$



Idee: Minimiere L per Gradientenabstieg

$$\begin{aligned} w_{ji}^l &\leftarrow w_{ji}^l - \lambda \cdot \frac{\partial L}{\partial w_{ji}^l} \\ b_{ji}^l &\leftarrow b_{ji}^l - \lambda \cdot \frac{\partial L}{\partial b_{ji}^l} \end{aligned}$$

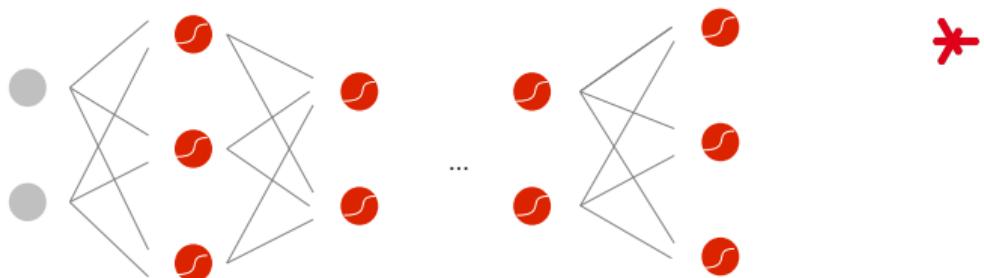
Wir berechnen die partiellen Ableitungen per Kettenregel

$$\begin{aligned} \frac{\partial L}{\partial w_{ji}^l} &= \frac{\partial L}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{ji}^l} = \delta_j^{l-1} \\ \frac{\partial L}{\partial b_{ji}^l} &= \frac{\partial L}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_{ji}^l} = 1 \end{aligned}$$

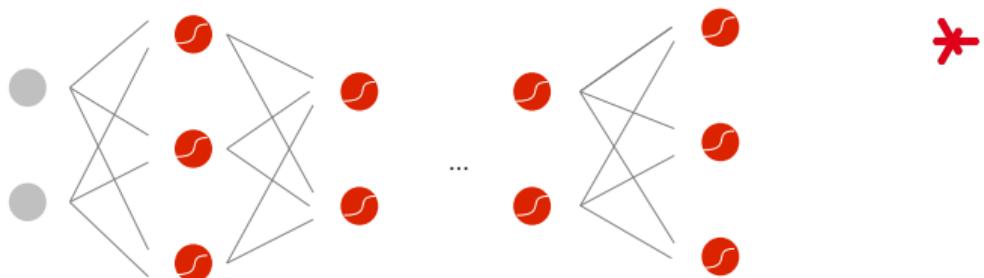
$$z_j^l = \left(\sum_i \delta_{i-1} \cdot w_{ji}^l \right) + b_j^l$$

$\delta_j^l = \begin{cases} \text{siegt oder} \\ \text{siegt das durch, wenn} \\ \text{das Neuron stärker} \\ \text{aktiviert wird} \end{cases}$

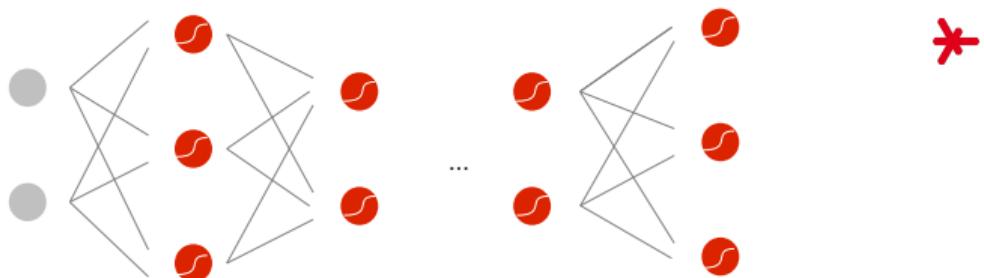
Backpropagation im MLP



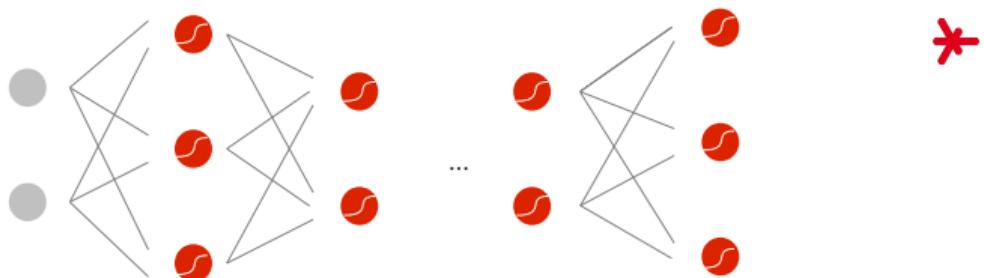
Backpropagation im MLP



Backpropagation im MLP



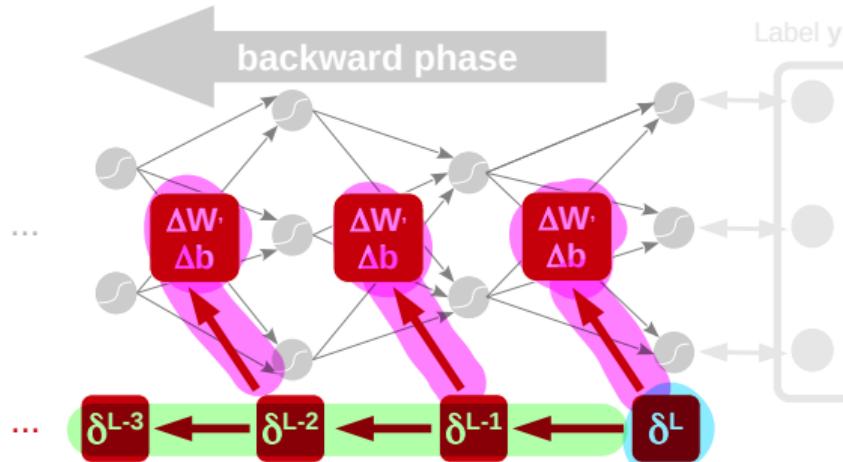
Backpropagation im MLP



Backpropagation im MLP



Backpropagation (Wiederholung)



Backpropagation-Formeln

$$\delta^L = (\hat{\mathbf{y}} - \mathbf{y}) \odot g'(\mathbf{z}^L)$$

$$\delta^l = ((W^{l+1})^T \cdot \delta^{l+1}) \odot g'(\mathbf{z}^l) \quad \forall l$$

$$\Delta w_{ji}^l = -\lambda \cdot \delta_j^l \cdot a_i^{l-1} \quad \forall l, i, j$$

$$\Delta b_j^l = -\lambda \cdot \delta_j^l \quad \forall l, i$$

Der Backpropagation-Algorithmus (vollständig)

```
1  function BACKPROPAGATION( $x_1, \dots, x_n, y_1, \dots, y_n$ ):
2       $W^2, W^3, \dots, W^L, b^2, b^3, \dots, b^L := \text{initialize}()$ 
3      repeat
4          ( $x, y$ ) := choose_sample()
5          feed  $x$  to the net, compute activations  $z^1, \dots, z^L$  and  $a^1, \dots, a^L (= \hat{y})$  // forward pass
6           $\delta^L := (\hat{y} - y) \cdot g'(z^L)$ 
7          for layer  $l = L, L-1, \dots, 2$ :
8               $w_{ji}^l := w_{ji}^l - \lambda \cdot \delta_j^l \cdot a_i^{l-1}$  for all  $j, i$  // update weights
9               $b_j^l := b_j^l - \lambda \cdot \delta_j^l$  for all  $j$  // update biases
10              $\delta^{l-1} := ((W^l)^T \cdot \delta^l) \odot g'(z^{l-1})$  // update deltas
11         until weights+biases stop changing
```



References I

- [1] fdecomite: Minsky & Papert Model.
<https://flic.kr/p/5VsZ1M> (retrieved: Nov 2016).
- [2] G. Cybenko.
Approximation by Superpositions of a Sigmoidal Function.
Mathematics of Control, Signals, and Systems (MCSS), 2(4):303–314, December 1989.