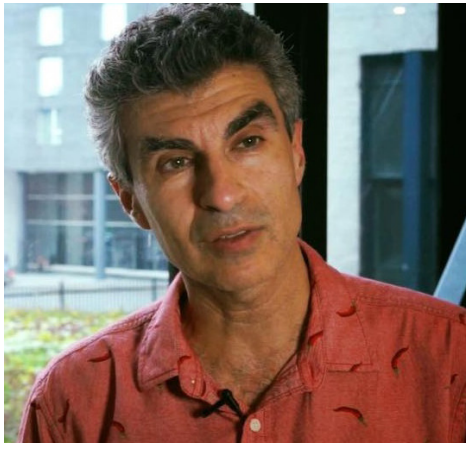
Anwendungen der KI

– Sommersemester 2018 –

Kapitel 06: Neuronale Netze

Prof. Dr. Adrian Ulges
B.Sc. Informatik (AI, ITS, MI, WI)
Fachbereich DCSM
Hochschule RheinMain

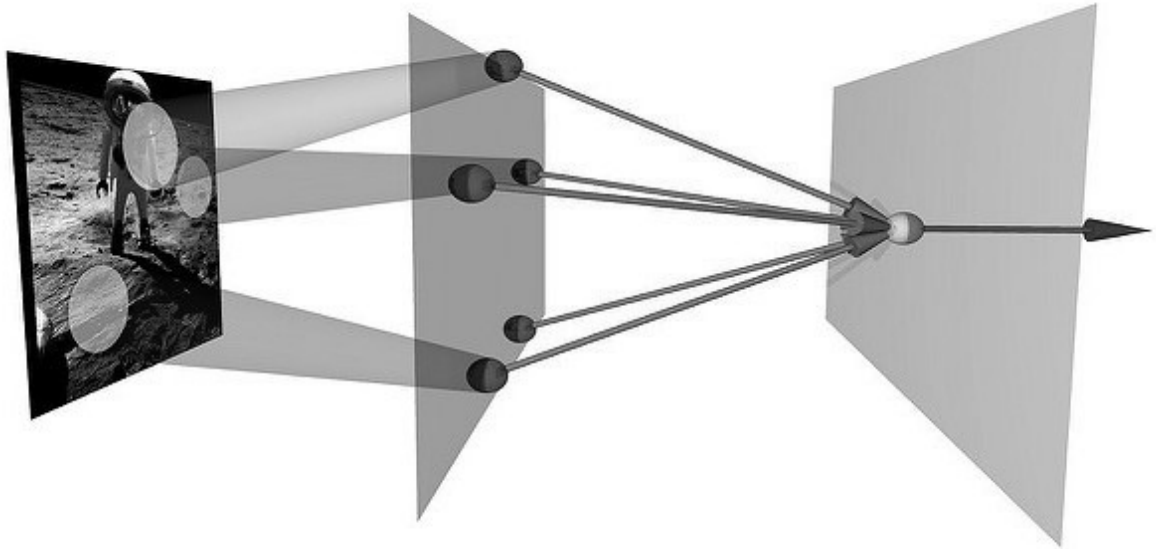
Neuronale Netze: Material


Bücher und Resourcen

- **Nielsen**: "Neural Networks and Deep Learning"
  `http://neuralnetworksanddeeplearning.com`
- Goodfellow, **Bengio**, Courville: "Deep learning"
  `https://www.deeplearningbook.org`
- **Goldberg**: "A Primer on Neural Network Models for Natural Language Processing"
  `http://u.cs.biu.ac.il/~yogo/nnlp.pdf`
- **Toolkit**: Tensorflow → `https://www.tensorflow.org/`

(Künstliche) Neuronale Netze.... Bild: [1]



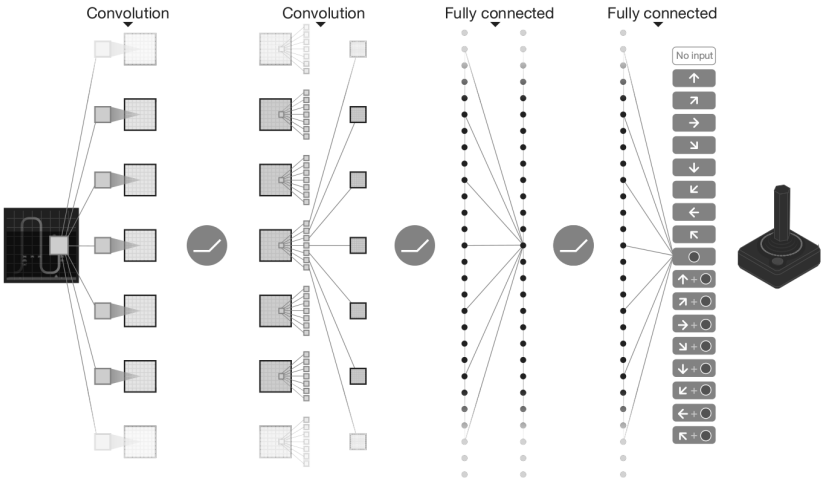
... sind eines der Kernthemen der künstlichen Intelligenz.

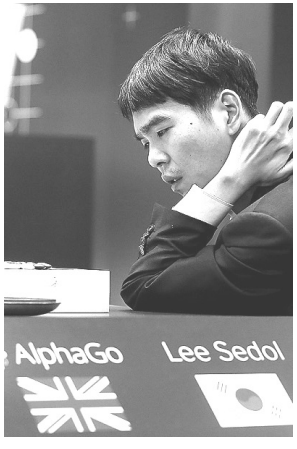
... sind **biologisch inpririerte** Modelle und simulieren Prozesse von Wahrnehmung und "Denken" in lebenden Organismen.

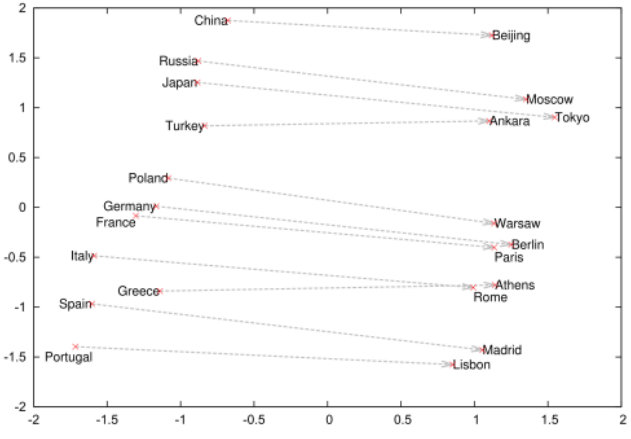
... sind **Graphen** aus einfachen, verknüpften Berechnungseinheiten *(Neuronen)*.

... definieren ihr Verhalten nicht durch Regeln, sondern durch **interne Gewichte**. Diese werden per *Training* auf Beispieldaten ermittelt.

Trend: “Deep Learning” Bilder: [5] [6] [2] [4] [7]

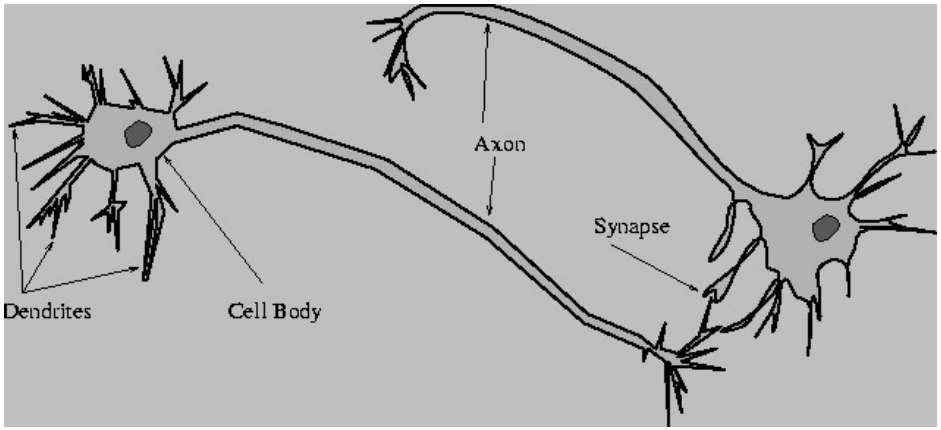


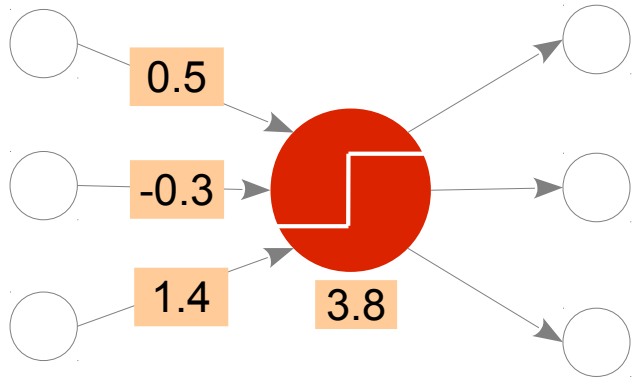




Outline

Inspiration: Biologische Neuronale Netze





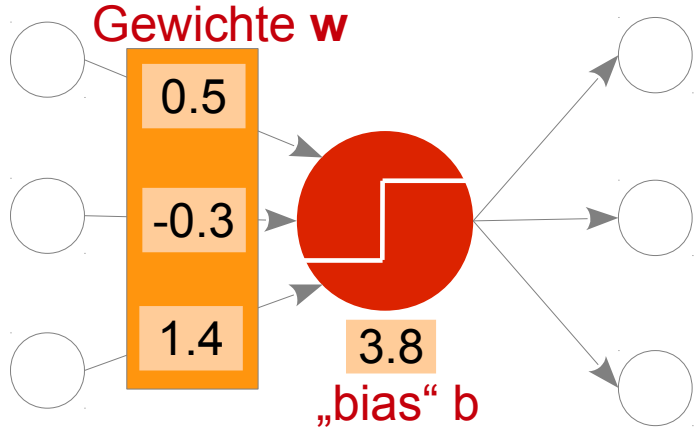
Biologische Neuronen (links)

- ... nehmen **elektrische Impulse** von anderen Neuronen auf, verarbeiten sie und geben sie weiter.
- ... sind mit vielen (z.B. 7000) anderen Neuronen verbunden
- ... passen sich durch Saturierung/Sensitivierung an *(Lernen)*

Künstliche Neuronen (rechts)

- ... bilden biologische Neuronen einfachstmöglich nach *(gewichtete Summe + Aktivierungsfunktion)*
- ... definieren ihr Verhalten durch ihre **Gewichte** (orange)

Neuron: Berechnungsmodell



- Gegeben: Eingabewerte
  $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$
- Das Neuron φ besitzt...
  - Gewichte $\mathbf{w} = (w_1, \ldots, w_n) \in \mathbb{R}^d$
  - einen “Bias” $b \in \mathbb{R}$
  - eine **Aktivierungsfunktion** f,
    z.B. die Treppenfunktion $\mathbf{1}_{x \geq 0}$
- Dann lautet die **Ausgabe** des Neurons:

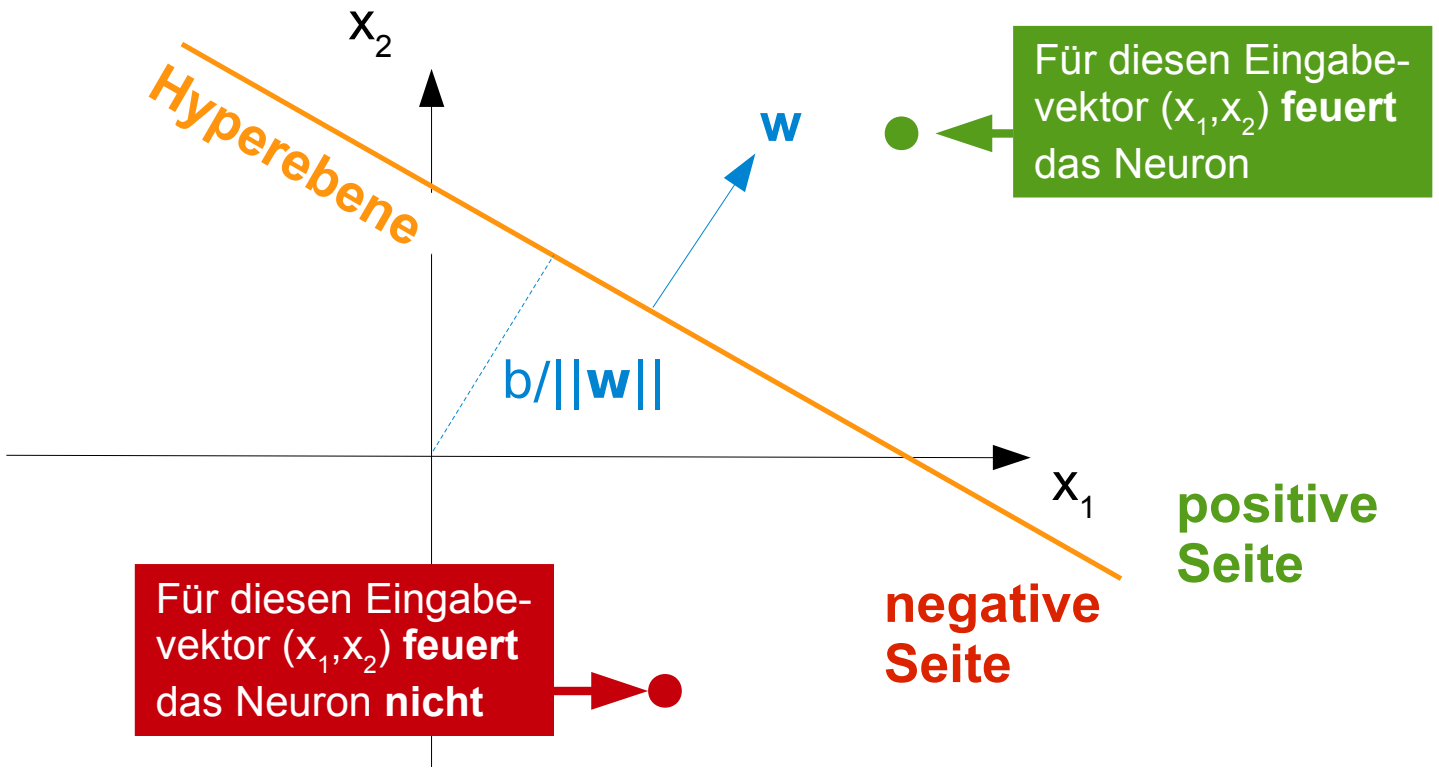
$$y = \varphi(\mathbf{x}) = f(\mathbf{x} \cdot \mathbf{w}) = \begin{cases} 1 & \text{falls } \mathbf{w} \cdot \mathbf{x} \geq \theta \\ 0 & \text{sonst} \end{cases}$$

Anmerkungen

- Das Neuron φ stellt also eine Funktion $\varphi : \mathbb{R}^d \to \mathbb{R}$ dar.
- Diese Funktion besitzt die **Parameter** $(w_1, \ldots, w_d, b)$.
- Ist $\varphi(\mathbf{x}) = 1$, sagen wir “Das Neuron **feuert**”.

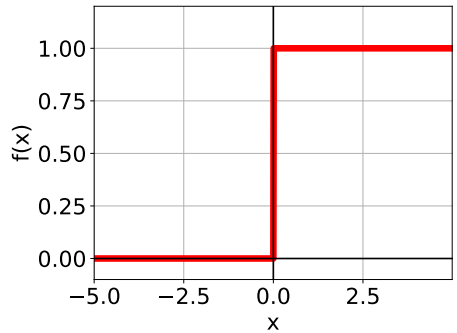
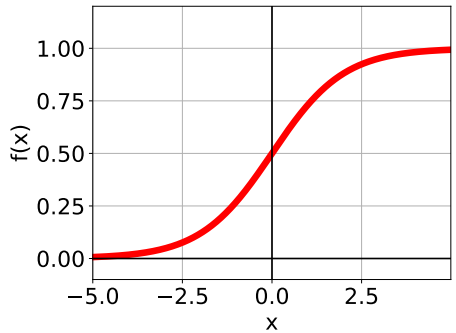
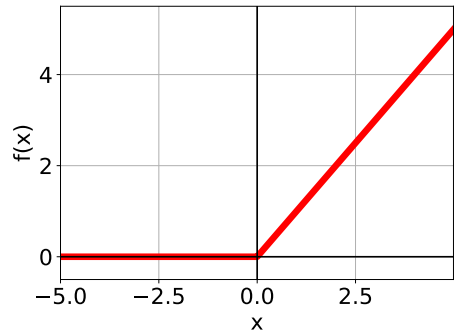
Neuron: Graphische Interpretation

- Das Neuron φ stellt eine **Hyper-Ebene** dar!
- $\mathbf{w} \cdot \mathbf{x} + b \geq 0$: Das Neuron feuert.
- $\mathbf{w} \cdot \mathbf{x} + b < 0$: Das Neuron feuert nicht.
- Das Neuron ist also ein *(linearer, binärer)* Klassifikator.
- Die *Lage* der Hyperebene wird durch $\mathbf{w}$ und b bestimmt.



Andere Aktivierungsfunktionen

Wir können (neben der Treppenfunktion) auch **andere Aktivierungsfunktionen** nutzen:

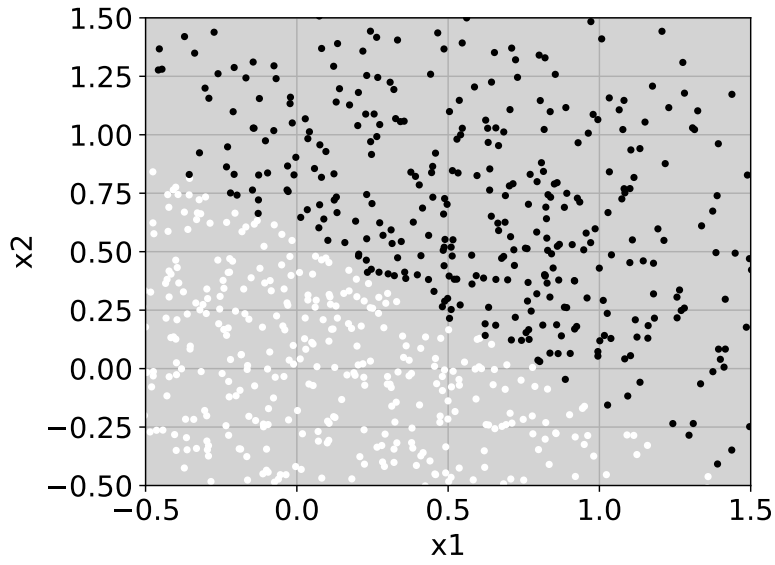
Treppenfunktion	**Sigmoid**	**RELU**
		
$f(x) = \begin{cases} 1 & \text{if } x \geq b \\ 0 & \text{else.} \end{cases}$	$f(x) = \frac{1}{1+e^{-x}}$	$f(x) = max(0, x)$

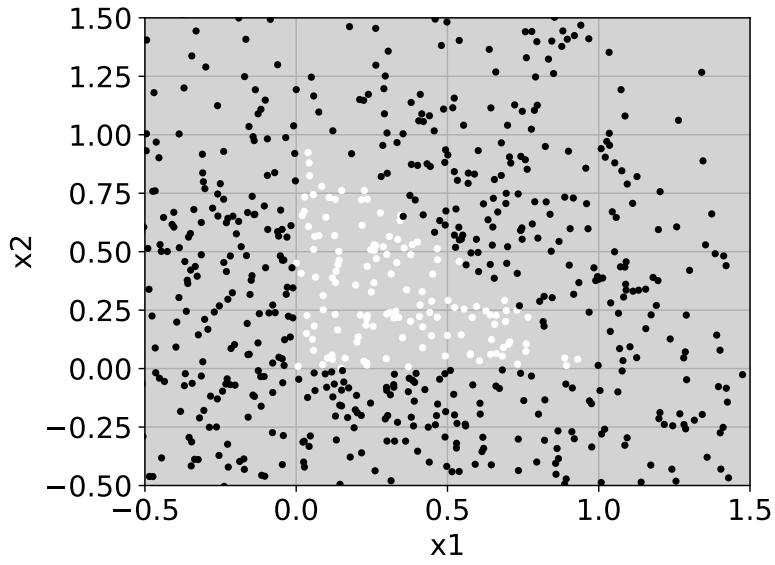
Anmerkungen

- Der Sigmoid approximiert die Treppenfunktion, ist aber darüber hinaus **differenzierbar** (mit $f'(x) = f(x) \cdot (1 - f(x))$)
- Wir bleiben (vorerst) beim Sigmoid.

Outline

Neuronen: Ausdrucksmächtigkeit

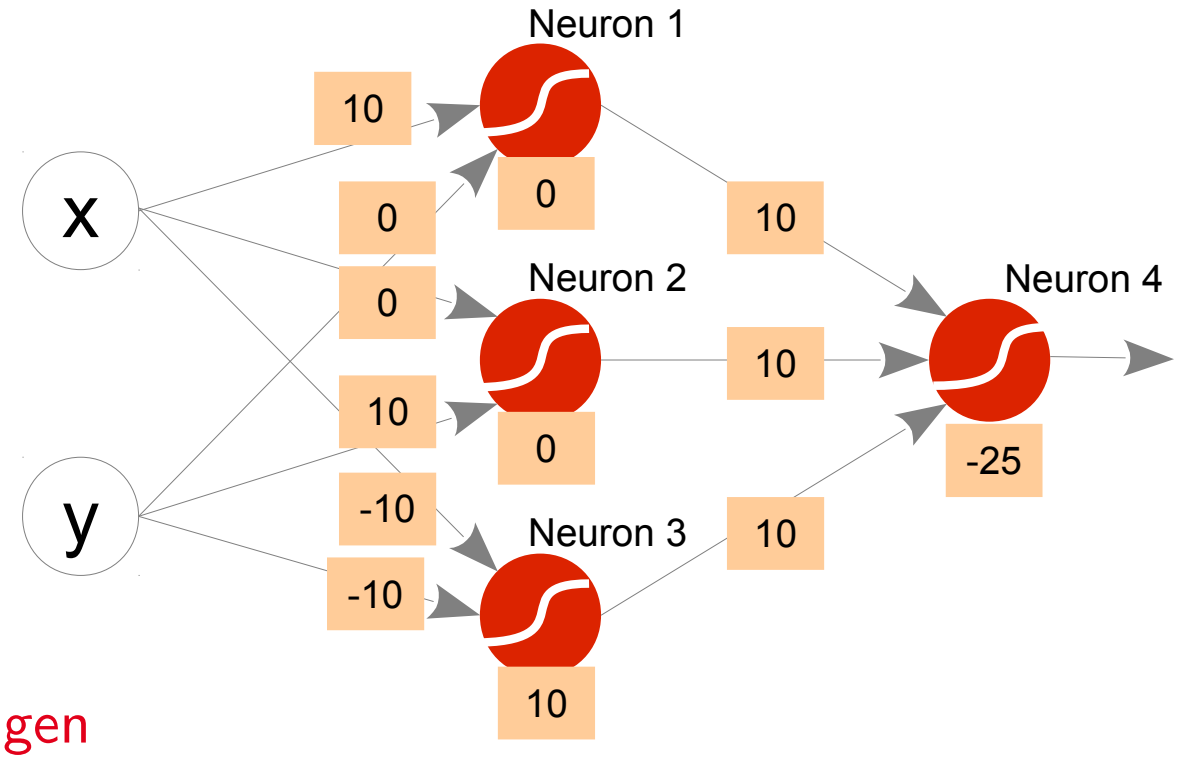




Welches der Probleme kann ein Neuron lösen?

- Nur das linke! Rechts sind die zwei Klassen "schwarz" und "weiß" **nicht linear separierbar** (d.h., es existiert keine Hyperebene – und somit kein Neuron φ – das die Klassen perfekt trennt).
- Wie könnte man **mehrere** Neuronen verknüpfen, um auch das rechte Problem zu lösen?

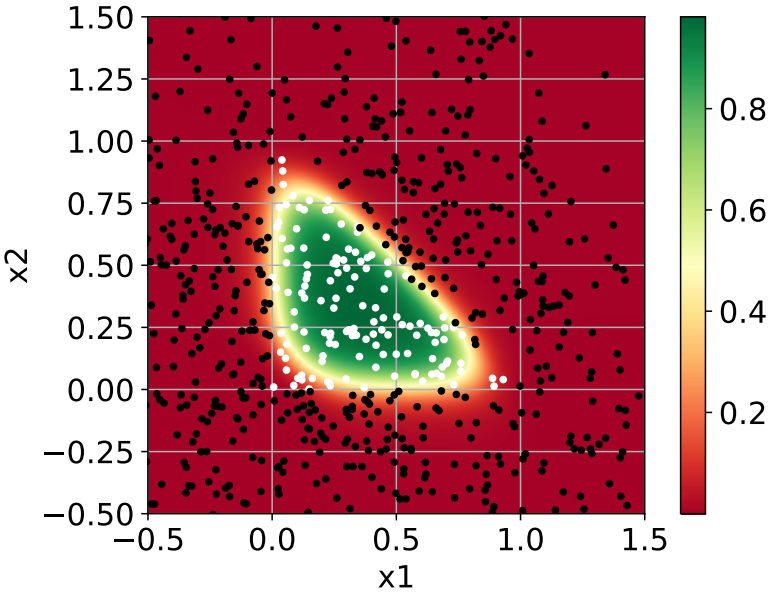
Lösung

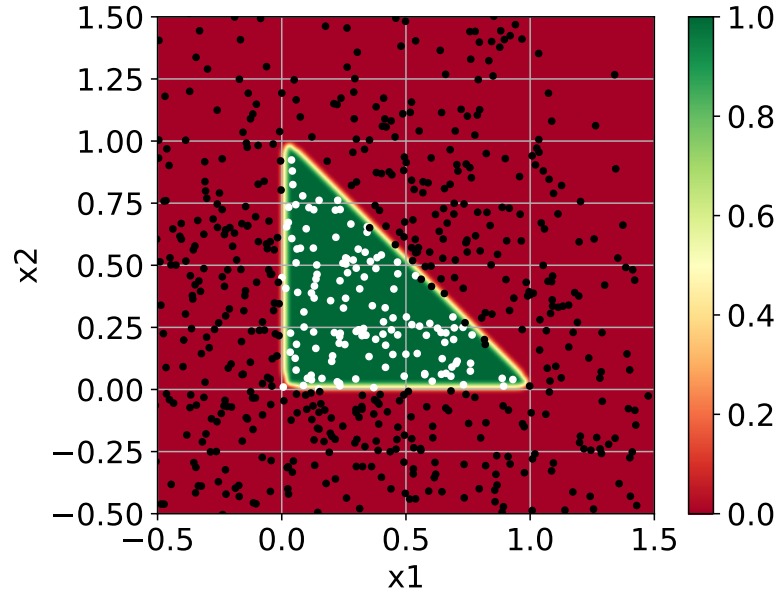


Anmerkungen

- Die Neuronen 1-3 modellieren die **Seiten des Dreiecks**
- Bsp.: Die Entscheidungsgrenze von Neuron 3 liegt bei: $-10 \cdot x_1 - 10 \cdot x_2 + 10 = 0$
- **Neuron 4** enthält von den Neuronen 1-3 jeweils das Signal, ob die Eingabe auf der "korrekten" Seite liegt
- **Neuron 4** modelliert eine **'Und'-Verknüpfung**
- Wir haben ein Netz mit 3 Schichten *(Input, Hidden, Output)*

Lösung: Illustration





- Die Plots zeigen die Ausgabe des obigen Netzes per Farbe
- Es wurden **Sigmoid-Neuronen** verwendet *(keine harte Entscheidungsgrenze, sondern ein glatter Übergang)*
- Das **linke Bild** zeigt das Netz auf der vorherigen Seite
- Im **rechten Bild** wurden alle Gewichte und Biases mit 10 **multipliziert**.

Können Neuronale Netze alles lernen?

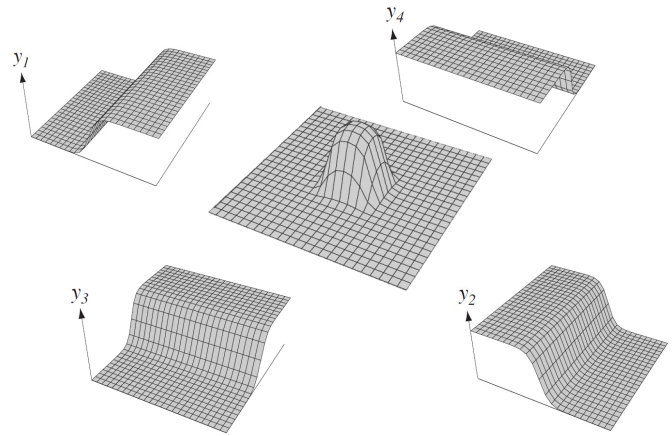
Theorem (Ausdrucksmächtigkeit neuronaler Netze)

Durch Verknüpfung mehrerer Neuronen mit sigmoidaler Aktivierungsfunktion können wir jede stetige Funktion $g : [0,1]^d \to [0,1]$ *mit beliebiger Genauigkeit approximieren.*

Anmerkungen

- Wir können somit in der Tat jedes Klassifikationsproblem lösen
- Wir brauchen hierzu nur **eine Hidden-Schicht**.
- Problem: **Wieviele** Hidden-Neuronen brauchen wir?
- Problem: Wie **finden** wir Lösungsgewichte?

"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. Only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single bidden layer neural networks. 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. "

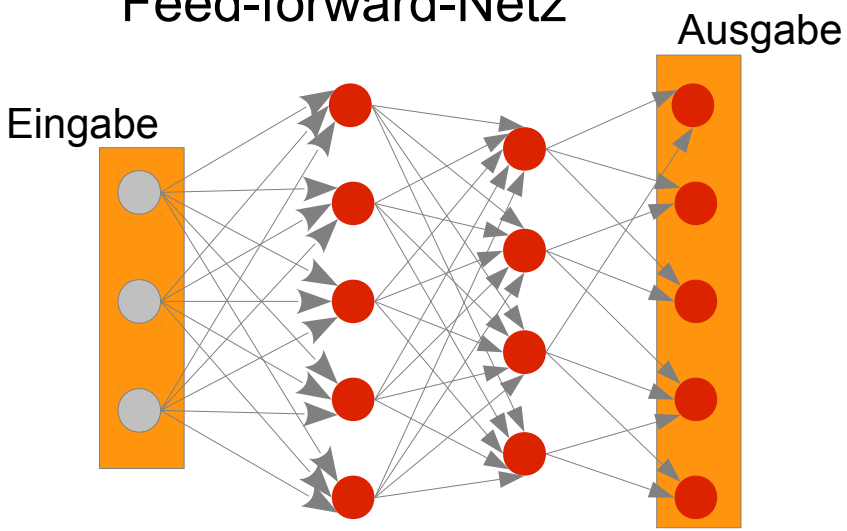


(Cybenko., G. [3])

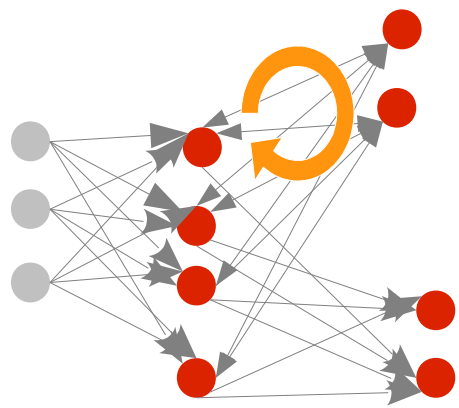
Neuronales Netz (NN): Definition

- Ein NN ist eine Menge **(partiell) verknüpfter** Neuronen, d.h. die Ausgabe eines Neurons kann als Eingabe anderer Neuronen dienen.
- Die **Eingabe**/**Ausgabe** des Netzes besteht aus allen Signalen die nicht von/zu anderen Neuronen kommen/gehen. Hierfür werden eigene **Input-/Output-Knoten** verwendet.
- Wir unterscheiden zwei Topologien: (1) **Feedforward-Netze**, (2) **rekurrente Netze**.

Feed-forward-Netz



Rekurrentes Netz
(mit Zyklen!)

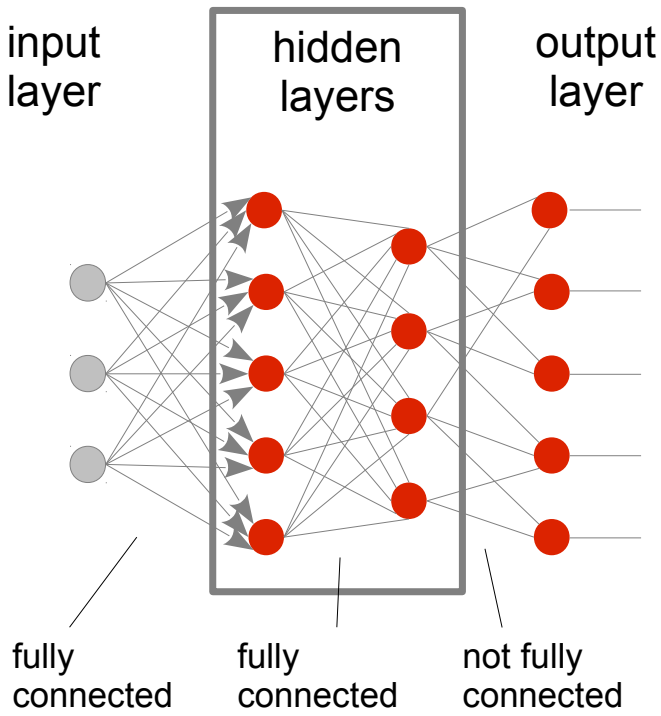


Feedforward-Netze

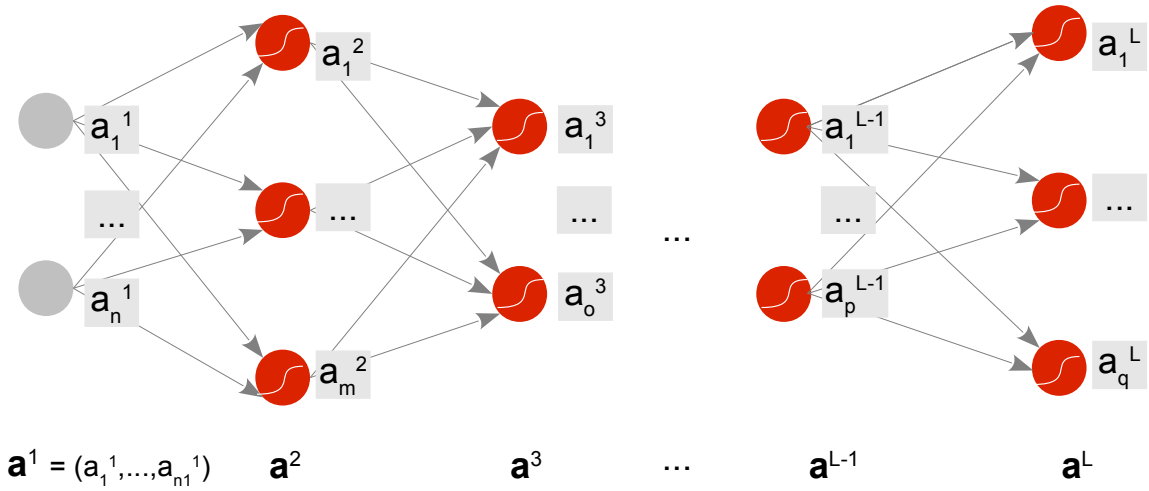
- **Feedforward-Netze** sind enthalten keine Zyklen!
- **Konvention**: Wir organisieren sie in **Schichten**.

Schichtenmodell

- Jedes Neuron ist ausschließlich mit Neuronen der vorherigen und nächsten Schicht verbunden.
- Zwei Schichten sind vollständig verbunden *(engl. "fully connected")* wenn alle Ihrer Neuronen paarweise verbunden sind.



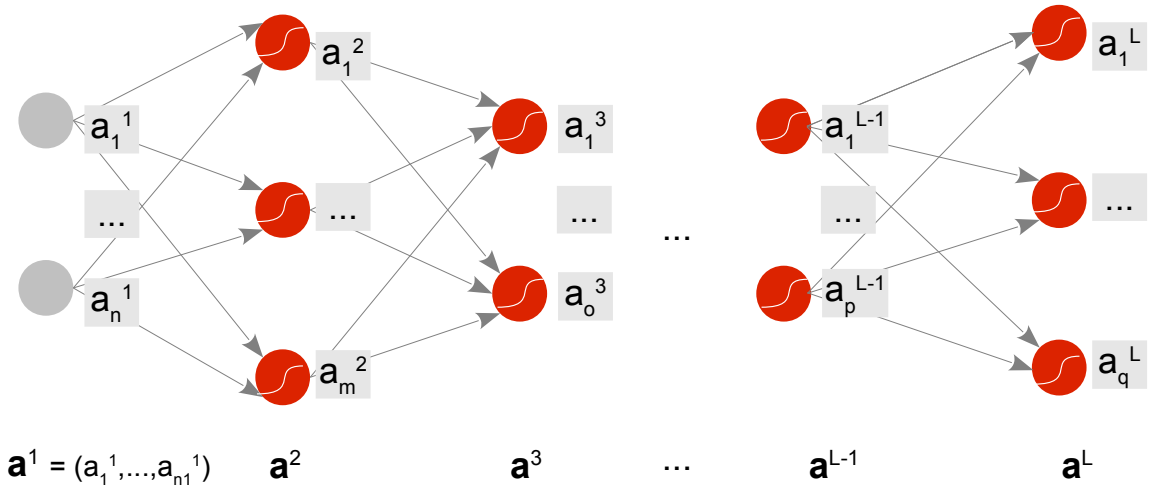
Feedforward-Netze: Notation



Notation

- a_i^l (bzw. b_i^l) bezeichnet die **Aktivierung** (bzw. den **Bias**) des i-ten Neurons in der l-ten Schicht
- w_{ik}^l bezeichnet das **Gewicht** der Kante zwischen dem i-ten Neuron in der $(l-1)$-ten Schicht und dem k-ten Neuron in der l-ten Schicht.
- Es gibt **Schichten** $l = 1, ..., L$ mit jeweils $n_1, ..., n_L$ Neuronen.

Feedforward-Netze: Notation

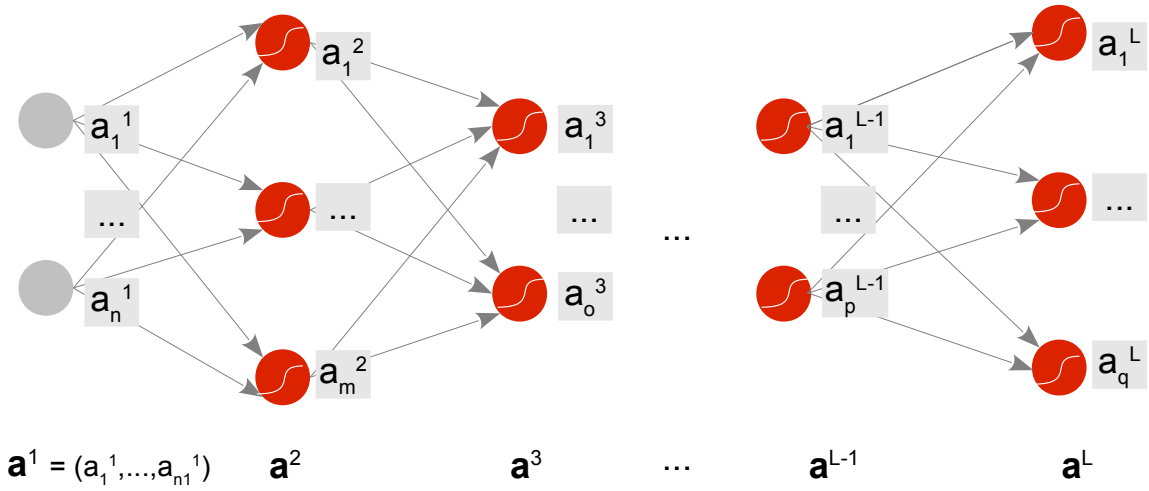


Kompaktere Notation: Vektoren/Matrizen

- Vektor mit Aktivierungen je Schicht: $\mathbf{a}^l := (a_1^l, ..., a_{n_l}^l)$
- Analog für den Bias: $\mathbf{b}^l := (b_1^l, ..., b_{n_l}^l)$
- Matrix mit Gewichten zwischen Schicht $l-1$ und l:

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

Feedforward-Netze: Berechnungsmodell



- Gegeben den Eingabevektor $\mathbf{a}^1$, wird das Signal Schicht für Schicht durch das Netz propagiert *(forward-pass)*
- In jeder Schicht sammeln wir zunächst den Input z_i^l jedes Neurons ...

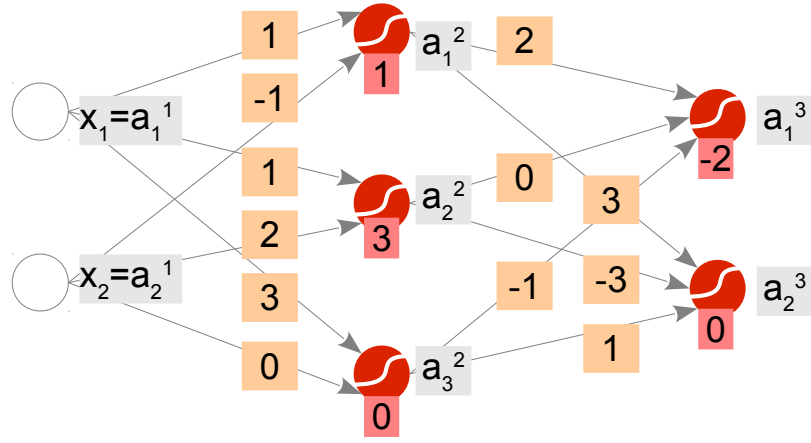
$$\mathbf{z}^l = W^l \cdot \mathbf{a}^{l-1} + \mathbf{b}^l$$

- ... und wenden hierauf für jedes Neuron die Aktivierungsfunktion f an:

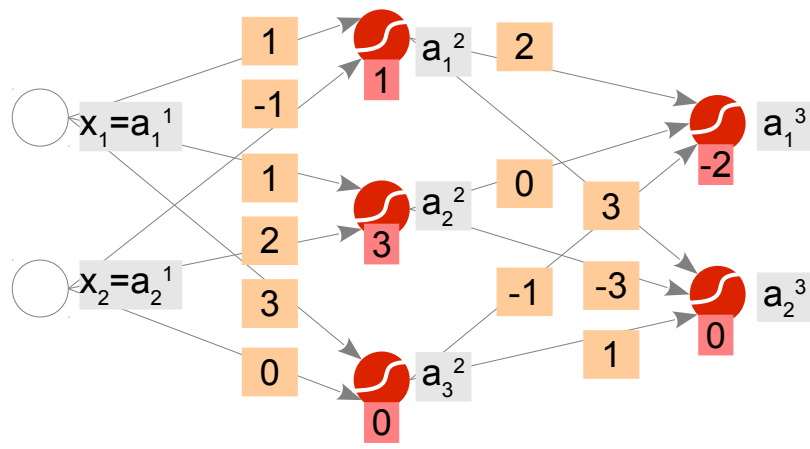
$$\mathbf{a}^l = \left(f(z_1^l), f(z_2^l), \ldots\right) = f(\mathbf{z}^l)$$

Forward-Pass: Do-it-yourself

Gegeben ist das folgende Netz mit Treppen-Aktivierungsfunktionen f und Input $\mathbf{x} = (1, 0)$. Berechne den Output $\mathbf{a}^3$.

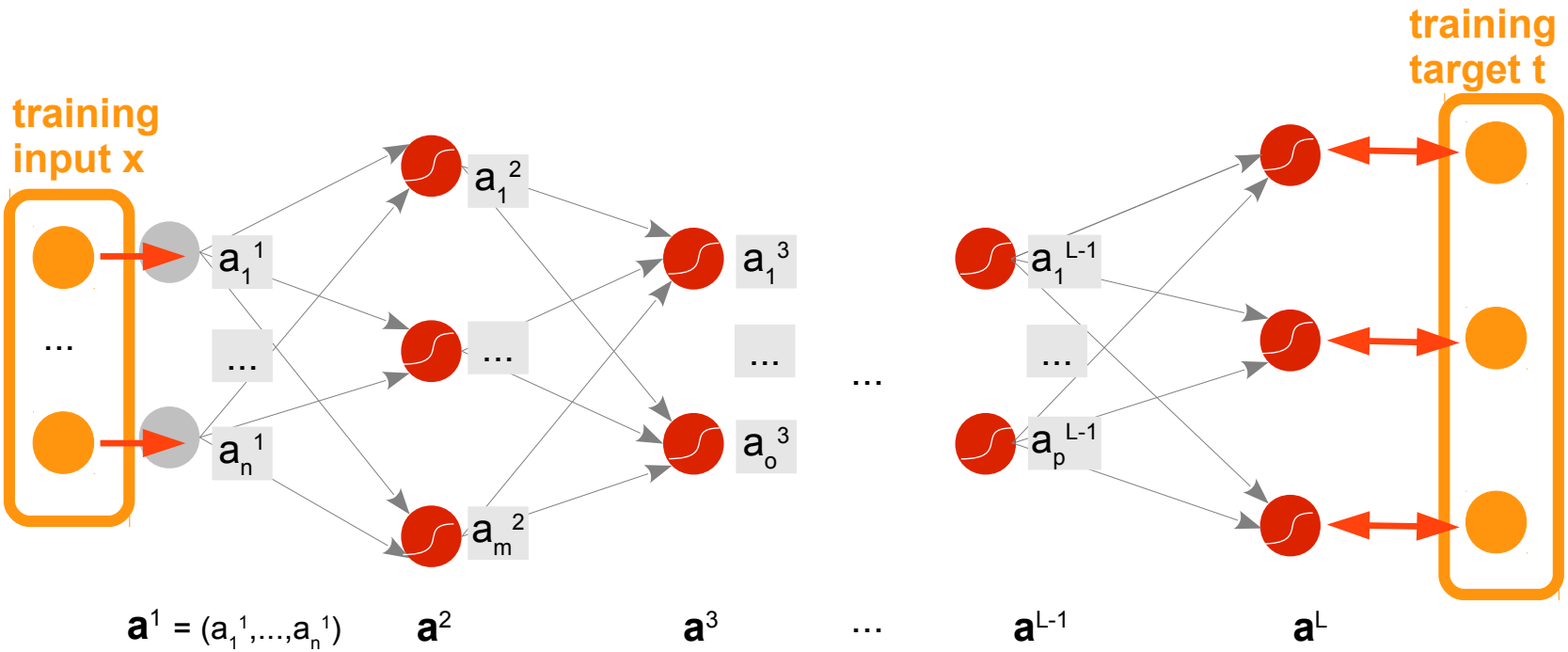


Forward-Pass: Do-it-yourself



Outline

Codierung von Klassifikationsproblemen



- Zum Training des Neuronalen Netzes benötigen wir **Trainingsdaten** aus Paaren von **Inputs** (=Merkmalsvektoren) $\mathbf{x}_1, ..., \mathbf{x}_n$ und erwünschten zugehörigen **Outputs** (=Targets) $y_1, ..., y_n$.
- Wie codieren wir Klassen-Labels in einem neuronalen Netz?

Target-Vektoren

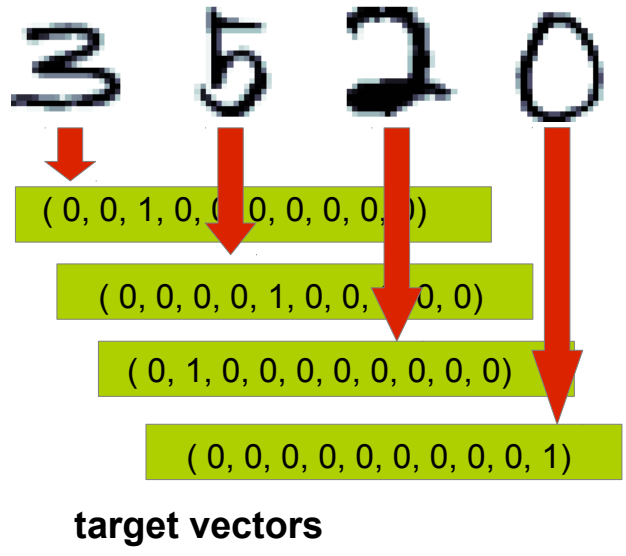
One-Hot Encoding

- Für jede Klasse gibt es einen Ausgabeknoten
- Ausgabeknoten c soll **genau dann** aktiviert sein, wenn der Input zu Klasse c gehört
- Die gewünschte Ausgabe eines Samples $\mathbf{x}$ entspricht also einem binären **Target-Vektor**

$$\mathbf{t} = (t_1, ..., t_C),\ z.B.\ \mathbf{t} = (0,0,0,1,0,0,0,0,0,0)$$

Beispiel: Ziffernerkennung

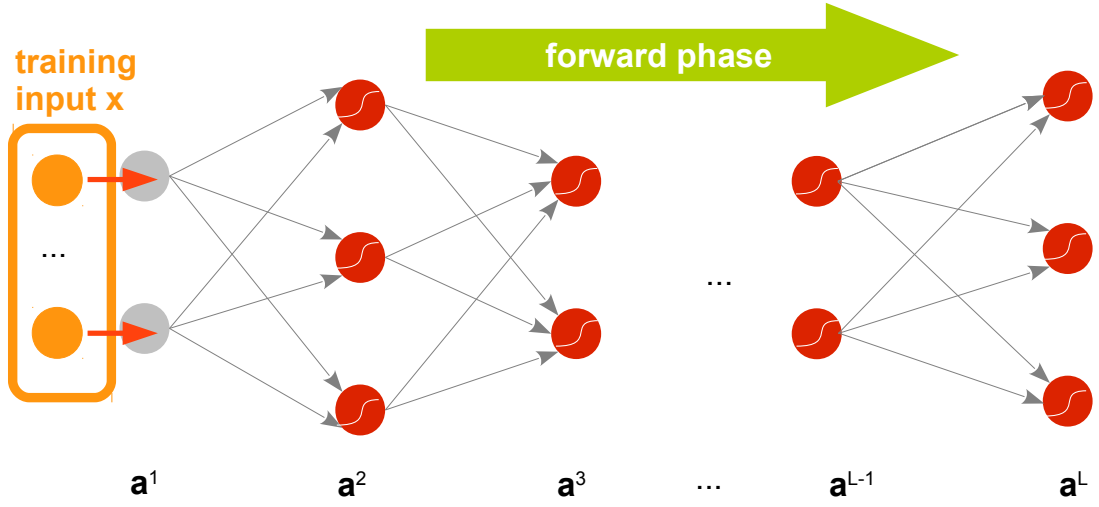
- $\mathbf{x} = (x_1, ..., x_{748})$
  (Pixel eines Bildes)
- target = **one-hot encoding**
  der zugehörigen Klasse
- Beispiel: Bild zeigt eine 4 $\rightarrow$
  $\mathbf{t} = (0,0,0,1,0,0,0,0,0,0)$



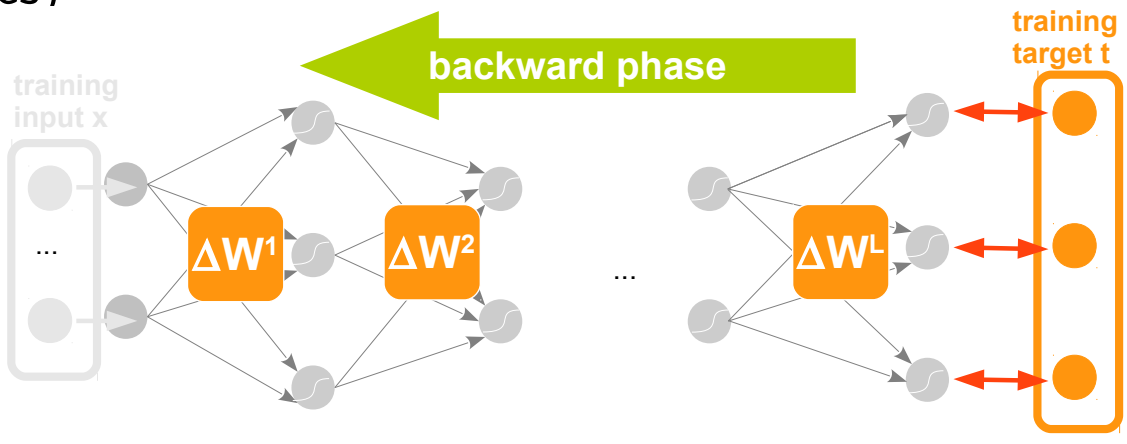
target vectors

Lernen: Ansatz

Wir initialisieren die Gewichte und Biases des Netzes zufällig und alternieren dann zwei Phasen:



1. **Forward-Phase**:
   Wähle ein Sample **x** als Eingabe.
   Berechne die Aktivierungen aller Neuronen, $\mathbf{a}^1, ..., \mathbf{a}^L$.
2. **Backward-Phase**: Vergleiche den Output $\mathbf{a}^L$ mit dem Target-Vektor **t** und verbessere die Gewichte (und Biases) in allen Schichten:
   - $\mathbf{W}^l := \mathbf{W}^l + \Delta\mathbf{W}^l$ (Gewichte)
   - $\mathbf{b}^l := \mathbf{b}^l + \Delta\mathbf{b}^l$ (Biases)



Backpropagation: Algorithmus

Das resultierende Lernverfahren nennt man **Backpropagation**:

```
function BACKPROPAGATION(x_1, ..., x_n, t_1, ..., t_n):
    W^2, W^3..., W^L, b^2, b^3, ..., , b^L := initialize()
    repeat
      (x, t) := choose_sample()
      feed x to the net, compute the activations a^1, ..., a^L  //
    Forward-Phase
      for layer l in L, L-1, ..., 2:
          w_ij^l := w_ij^l + Δw_ij^l      for all i,j    // Gewichte updaten
          b_j^l  := b_j^l  + Δb_i^l       for all i      // Biases updaten
    until weights+biases stop changing
```

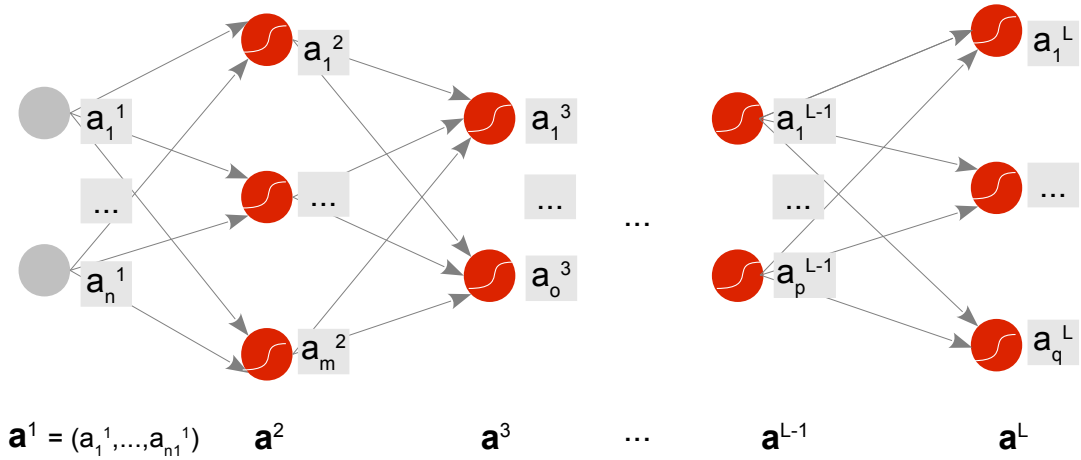
Anmerkungen

- Schlüsselfrage: Wie berechnen wir die Updates Δw_{ij}^l und Δb_i^l?
- Antwort: Wir definieren eine Fehler- *(engl. Loss-)*Funktion E:

$$E(W^2, W^3, ..., W^L, \mathbf{b}^2, \mathbf{b}^3, ..., , \mathbf{b}^L) := \sum_{i=1}^{n_L} \left(a_i^L - t_i \right)^2$$

- Diesen Fehler **minimieren** wir mittels **Gradientenabstieg**!

MLP: Backpropagation



MLP: Backpropagation

MLP: Backpropagation

MLP: Backpropagation

MLP: Backpropagation

MLP: Backpropagation

Backpropagation: Algorithm

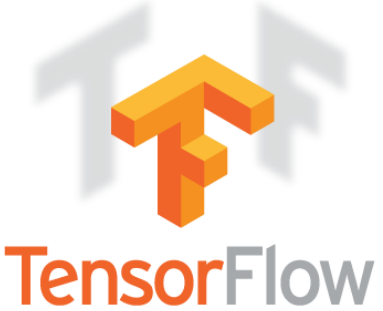
```
1  function BACKPROPAGATION(x_1, ..., x_n, t_1, ..., t_n, λ):
2      W^2, W^3, ..., W^L, b^2, b^3, ..., , b^L := initialize()
3      repeat
4        (x, t) := choose_sample()
5        feed x to the net, compute the activations a^1, ..., a^L  //
       Forward-Phase
6        δ^L := (a^L − t) · f'(z^L)
7        for layer l in L, L-1, ..., 2:
8            w_ij^l := w_ij^l − λ · a_i^(l−1) · δ_j^l   for all i, j    // Gewichte updaten
9            b_j^l := b_j^l − λ · δ_j^l                  for all j      // Biases updaten
10           δ^(l−1) := (W^l · δ^l) ⊙ f'(z^(l−1))          // Fehler updaten
11     until weights+biases stop changing
12
```

Anmerkungen

- $\lambda > 0$ – die *Lernrate* – ist schwierig zu wählen! *(Warum?)*

Outline

Tensorflow



Idee / Historie

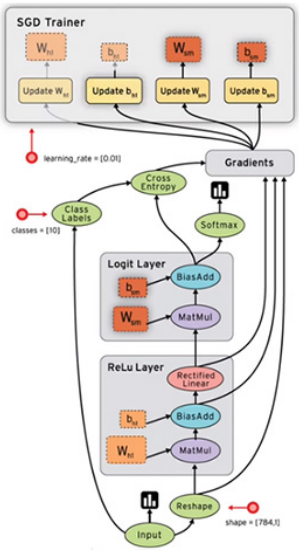
- Eines von mehreren **Frameworks** für Neuronale Netze, die im Zuge des **Deep Learning** – Trends entstanden sind *(Caffe, DeepLearning4J, Theano, Torch, ...)*
- Entstanden im Google Brain Team, seit Nov 2015 open-source
- Weit **verbreitet**: Google (Translate, Deepmind, ...), DropBox, ebay, airbnb, Airbus, Intel, ...

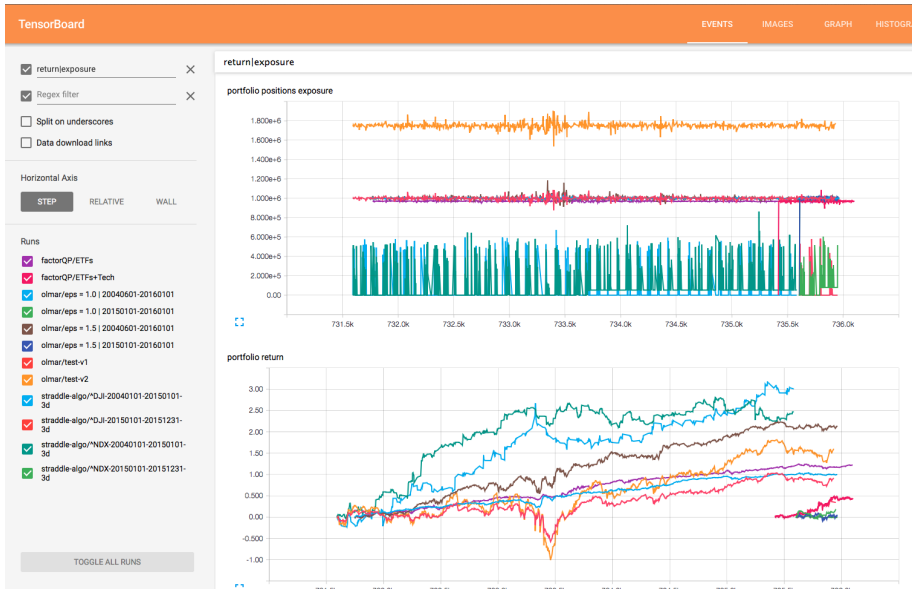
Features

- Aufbau neuronaler Netze als **Flow Graphs**
- **Backpropagation fertig eingebaut!**
- Automatische **Parallelisierung** auf CPUs / GPUs
- **vortrainierte Netze** verfügbar (vgl. Caffe's *Model Zoo*)
- **Visualisierung** des Netzverhaltens (Tensorboard)

Tensorflow

Tensorboard: Illustrationen

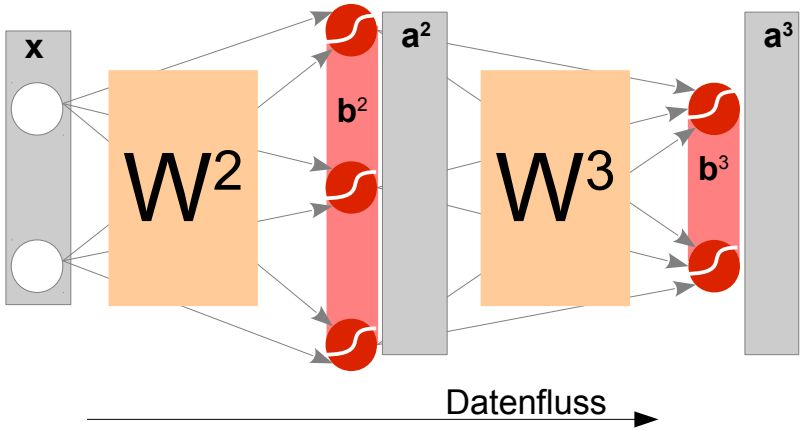


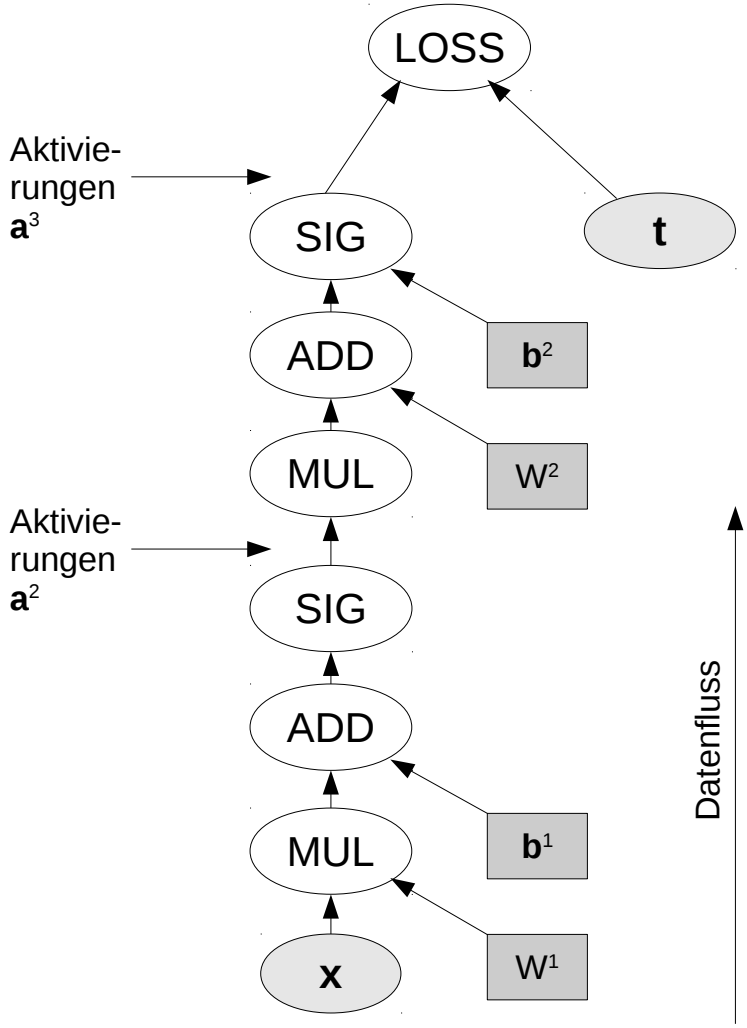


Technische Details

- Lizenz: Apache 2.0
- Interfaces: Python, C/C++
- Plattformen: Linux, Mac OS X, Windows, Android

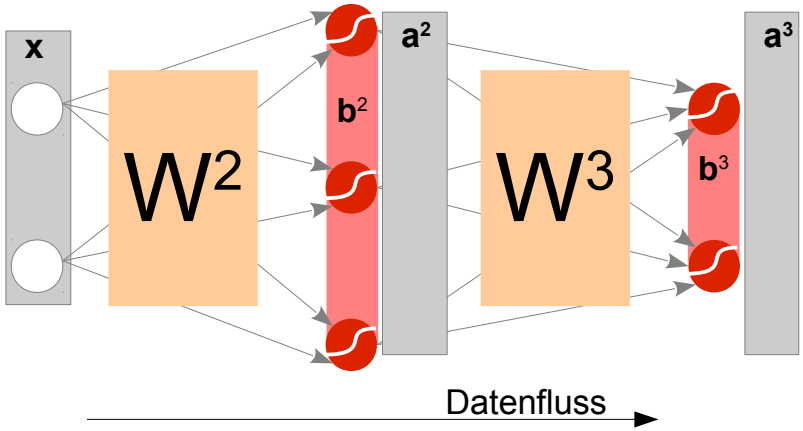
Neuronale Netze als Flowgraphs

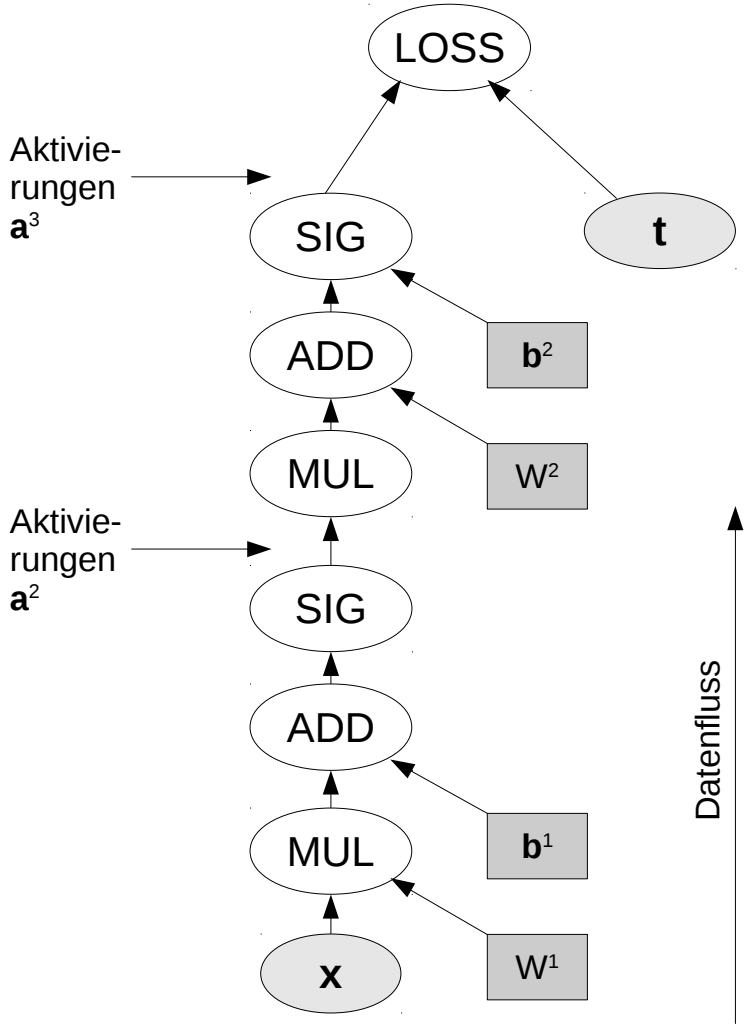




- Ein neuronales Netz (links) kann als sogenannter **“Flow Graph”** (rechts) dargestellt werden
- Hierzu fassen wir üblicher Weise die **Neuronen einer Schicht** zusammen
- Der Flow-Graph enthält Knoten für die **Berechnungsschritte** des Netzes

Neuronale Netze als Flowgraphs

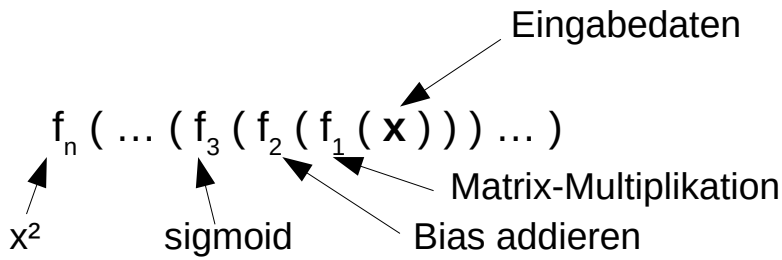




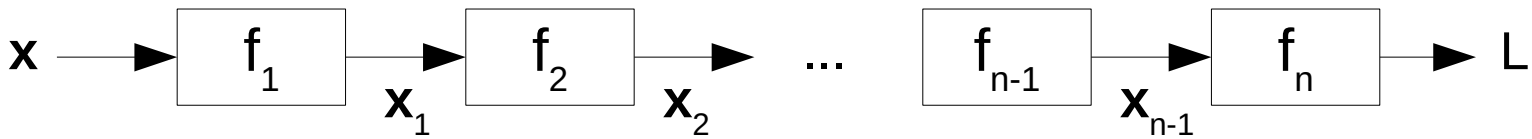
Knoten im Flow Graph

1. **Eingabedaten**
   Merkmalsvektor **x**, Target **t**, ...
2. **Parameter**
   Gewichte (W^2, W^3),
   Biases $(\mathbf{b}^2, \mathbf{b}^3)$, ...
3. **Rechenoperationen**: Matrix-Multiplikation (*MUL*),
   Vektor-Addition (*ADD*), Aktivierungsfunktionen (*SIG*), ...
   - Auch die Loss-Funktion (LOSS) ist eine Rechenoperation!

Wie funktioniert "Automatische Backpropagation"?



- Ein neuronales Netz ist eigentlich eine geschachtelte Funktion.
- Alternative Sicht:



- Zur Optimierung der Loss-Funktion L benötigen wir die Ableitungen $\nabla_{\mathbf{x_1}} L, \nabla_{\mathbf{x_2}} L, \dots, \nabla_{\mathbf{x_{n-1}}} L$.

Wie funktioniert "Automatische Backpropagation"?

Alternative 1: Numerische Differentiation

- Wir verwenden den Differenzenquotienten:

$$g'(x) \approx \frac{g(x+h) - g(x)}{h} \qquad // \text{ mit 'kleinem' } h$$

- **Probleme**: **Fehler!** Ein "gutes" h ist schwer zu bestimmen.

Alternative 2: Symbolische Differentiation

- Manipulation von Symbolen. Berechne aus mathematischen Ausdrücken deren Ableitung.
- **Beispiel**: "e^{3x^2}" $\rightarrow$ "$e^{3x^2} \cdot 6x$"
- wird in Mathe-Packages *(z.B. mathematica)* verwendet.

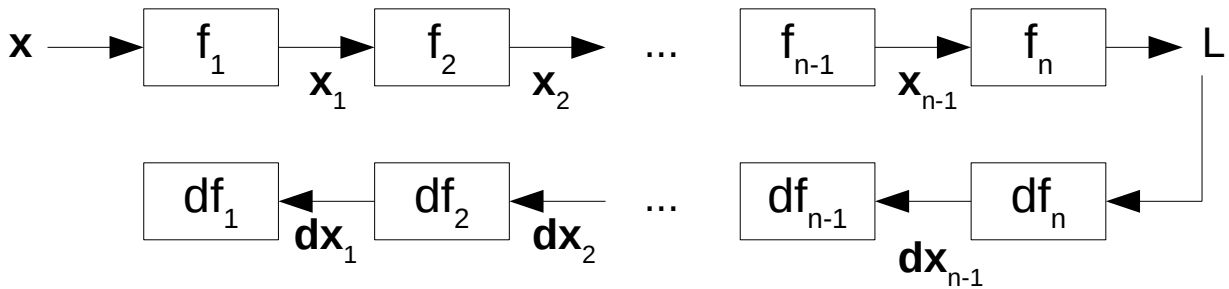
Hier: Automatische Differentiation (autograd)

- Verwendet in Tensorflow, Pytorch, Keras, ...
- **Ansatz**: Nehme eine **Python-Funktion** `f` und generiere eine neue Funktion `df`, die die **Ableitung** von `f` berechnet.
- Geht für alle `numpy`-Funktionen (`exp`, `sin`, ...), Schleifen, ...

```
1  import autograd.numpy as np
2  from autograd import grad, elementwise_grad
3
4  # Beispiel 1: x**2
5  def f(x):
6      return x**2
7
8  df = grad(f)
9
10 print( f(3.), df(3.) )  > 9.0, 6.0
11
12 # Beispiel 2: Sigmoid-Vektor
13 def f(x):
14     return 1. / (1 + np.exp(-x))
15
16 df = elementwise_grad(f)
17
18 x = np.array([-5, 0, 5])
19 print( f(x), df(x) )     > [0.01,0.5,0.99], [0.01,0.25,0.01]
```

Hier: Automatische Differentiation (autograd)

- Wenn wir für die **Einzelschritte** Ableitungsfunktionen haben, können wir sie leicht per Backpropagation **verknüpfen**.
- Wir definieren $\mathbf{dx}_k := \nabla_{\mathbf{x}_k} L$

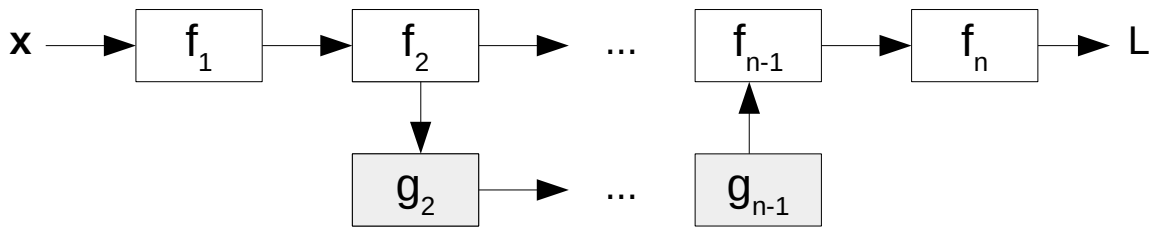


1. **Forward-Pass**: Berechne und speichere $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_{n-1}$.
2. **Backward-Pass**: Berechne und speichere $\mathbf{dx}_{n-1}, \mathbf{dx}_{n-2}, \ldots \mathbf{dx}_1$, denn *(gemäß Kettenregel)*:

$$\mathbf{dx}_k = \mathbf{dx}_{k+1} \cdot df_{k+1}(\mathbf{x}_k).$$

Anmerkungen

- Dies funktioniert auch bei **Branching**, z.B.



Tensorflow: Aufbau eines Flowgraphs

- Tensorflow importieren, Größe der Schichten festlegen

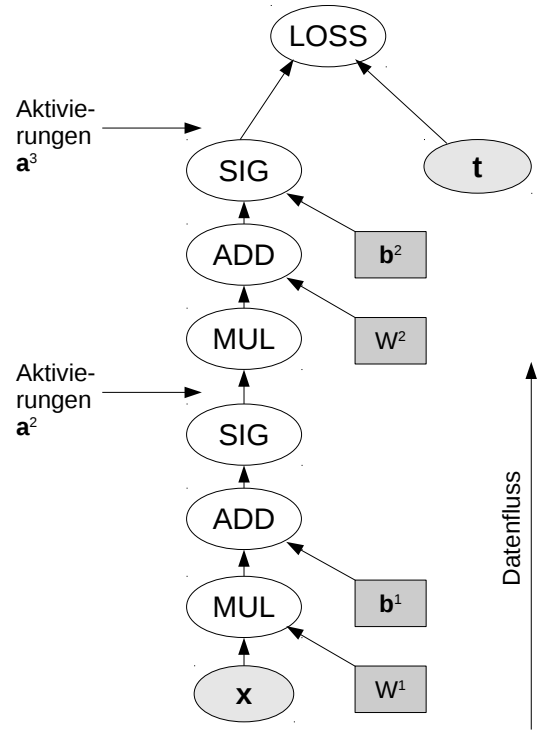
```
1  import tensorflow as tf
2
3  ninput,nhidden,noutput = 2,3,2
```

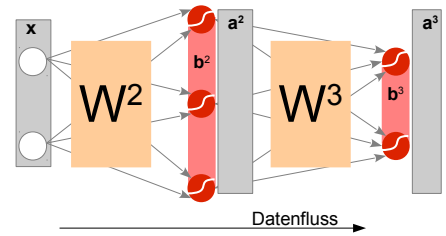
- Schichten mit Eingabedaten sind *placeholder*

```
1  x = tf.placeholder(tf.float32,
2                     shape=[None, ninput])
3  t = tf.placeholder(tf.float32,
4                     shape=[None, output])
```

- Gewichte und Biases sind *Variablen*
  und werden mit Zufallszahlen initialisiert

```
1  # Example: random values for W2
2  shape_W2 = [ninput,nhidden]
3  init_W2 = tf.random_normal(shape_W2,
4                             mean=0.0,
5                             stddev=0.1)
6  ...
7  W2 = tf.Variable(init_W2)
8  W3 = tf.Variable(init_W3)
9  b2 = tf.Variable(init_b2)
10 b3 = tf.Variable(init_b3)
```





Tensorflow: Aufbau eines Flowgraphs

- Aufbau des Netzes (zwei Schichten)

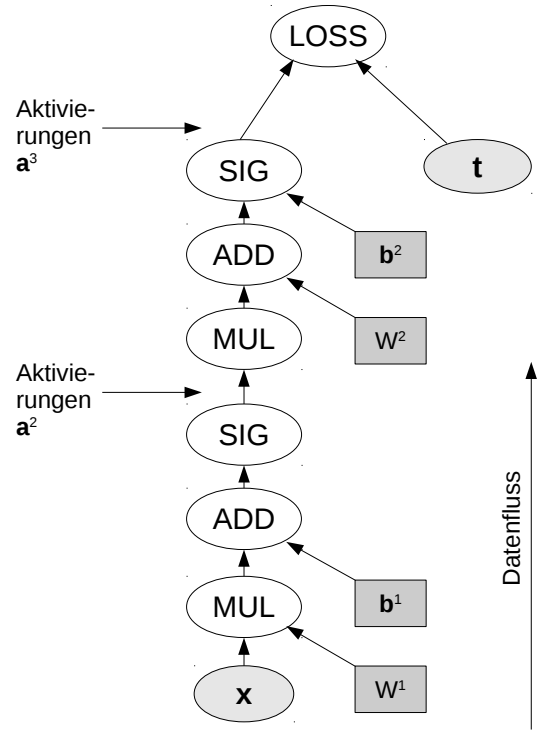
```
1    a2 = tf.sigmoid( tf.matmul( x, W2) + b2 )
2    a3 = tf.sigmoid( tf.matmul(a2, W3) + b3 )
```

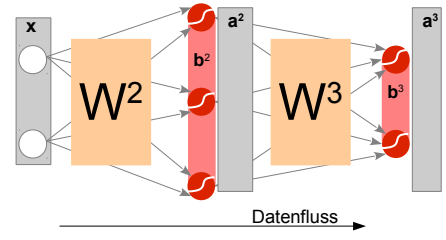
- Loss-Funktion aufbauen: Mittlerer quadratischer Fehler zwischen Output a3 und Target

```
1    # squared error per sample
2    squared_errors = tf.square( a3 - t )
3
4    # average this -> loss function
5    E = tf.reduce_mean(squared_errors)
```

- Weil noch Platz ist: Ein Extra-Knoten *accuracy*, der die Genauigkeit der Klassifikation misst

```
1    # boolean vector with correctness of
      classification per sample
2    correct = tf.equal (tf.argmax(a3, 1),
3                        tf.argmax(t,1) )
4    correct_float = tf.cast(correct, tf.float32)
5
6    # average this -> accuracy rate
7    accuracy = tf.reduce_mean( correct_float )
```

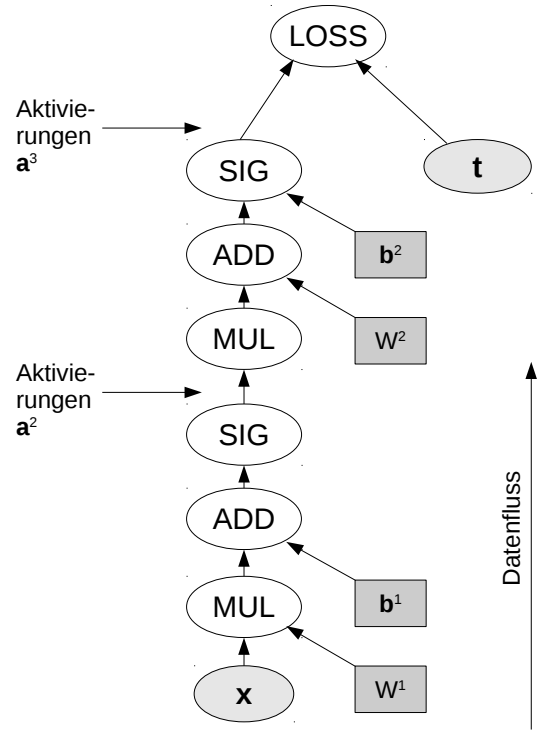


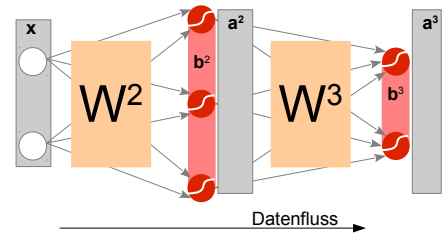


Tensorflow: Inferenz

- Inferenz bedeutet, dass man ein neuronales Netz "benutzt", d.h. man rechnet die Werte einer Schicht aus.
- In Tensorflow: Session initialisieren, `run()` aufrufen. Man erhält ein Numpy-Array mit dem Ergebnis.
- Hierbei muss man die nötigen Eingabedaten übergeben, in einem sogenannten `feed_dict`.

```
1   with tf.Session() as session:
2
3       # Eingabevektor -> Ausgabe
4       _in = {x: [0.2, 0.7]}
5       _out = session.run(a3,
6                          feed_dict = _in)
7
8       # -> Gewichte der Hidden-Schicht
9       _w2 = session.run(W2, feed_dict={})
10
11      # (Eingabevektoren,Targets) ->
12      #  Genauigkeit messen
13      _x = [[0.2,0.7],[0.4,0.9],[0.3,0.8]]
14      _t = [[0,1], [1,0], [1,0]]
15      _in = {x: _x, t: _t}
16      _acc = session.run(accuracy,
17                         feed_dict=_in)
```





Tensorflow: Training

- Zum Training benötigen wir Eingabevektoren und Targets

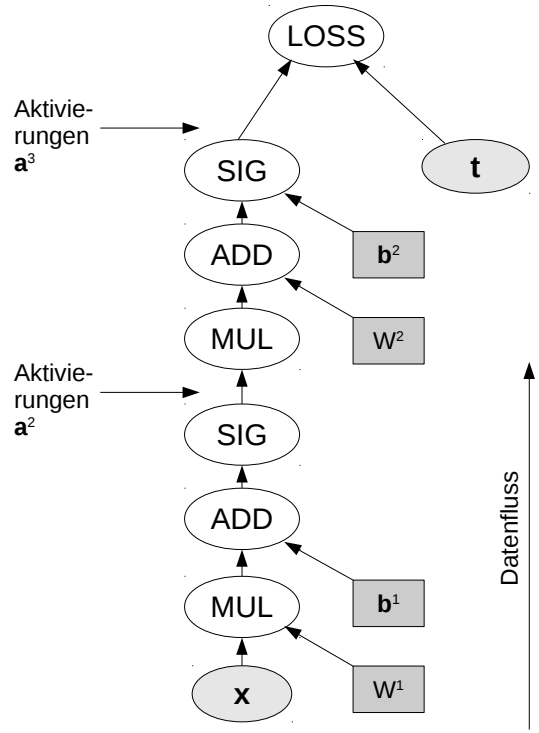
```
X = [ [0.2,0.7],
      [0.4,0.9],
      [0.3,0.8],
      ...
    ]
T = [ [  0,  1],
      [  1,  0],
      [  1,  0],
      ...
    ]
```

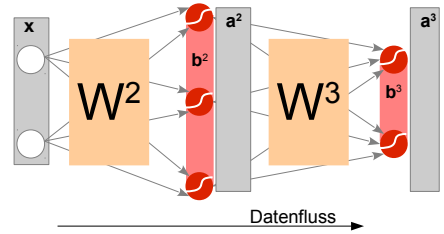
- Wir erweitern das Netz um einen speziellen Knoten.

```
lrate = 0.1 # learning rate
g = tf.train.GradientDescendOptimizer(lrate)
step = g.minimize(E)
```

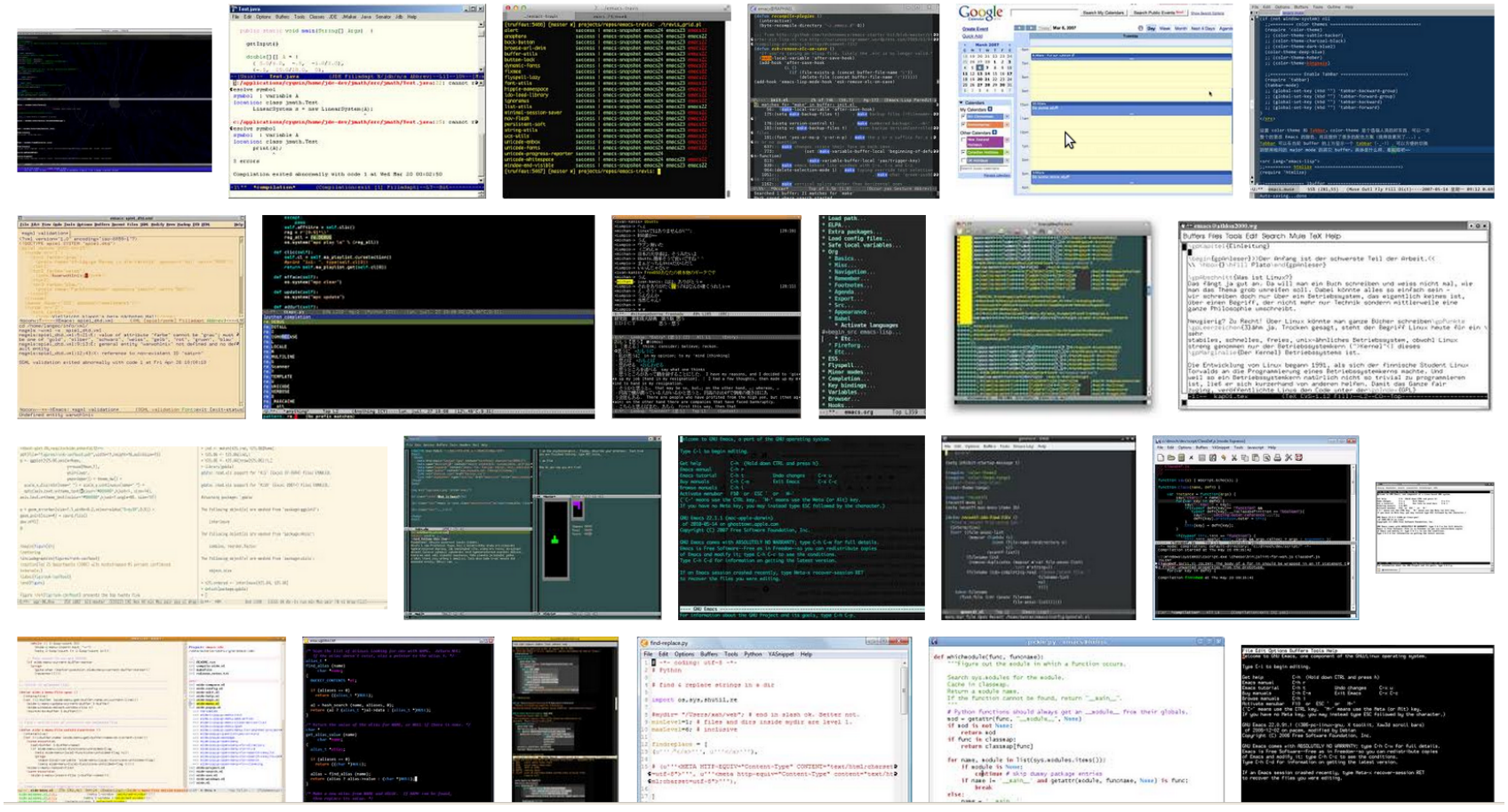
- Zum Training rufen wir einfach step auf den Trainingssamples auf!

```
for iteration in range(100):
  for _x,_t in zip(X,T):
    session.run(step, feed_dict={x:_x, t:_t})
```





Tensorflow: Code Example



References I

[1] fdecomite: Minsky & Papert Model.
https://flic.kr/p/5VsZ1M (retrieved: Nov 2016).

[2] Google DeepDream robot: 10 weirdest images produced by AI 'inceptionism' and users online (Photo: Reuters).
http://www.straitstimes.com/asia/east-asia/alphago-wins-4th-victory-over-lee-se-dol-in-final-go-match (retrieved: Nov 2016).

[3] G. Cybenko.
Approximation by Superpositions of a Sigmoidal Function.
Mathematics of Control, Signals, and Systems (MCSS), 2(4):303–314, December 1989.

[4] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean.
Distributed Representations of Words and Phrases and their Compositionality.
In Advances in Neural Information Processing Systems 26, pages 3111–3119. Curran Associates, Inc., 2013.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis.
Human-level control through deep reinforcement learning.
Nature, 518(7540):529–533, 02 2015.

[6] Mary-Ann Russon.
Google DeepDream robot: 10 weirdest images produced by AI 'inceptionism' and users online.
http://www.ibtimes.co.uk/google-deepdream-robot-10-weirdest-images-produced-by-ai-inceptionism-users-online-1509518 (retrieved: Nov 2016).

[7] C. Szegedy.
Building a deeper understanding of images (Google Research Blog).
https://research.googleblog.com/2014/09/building-deeper-understanding-of-images.html (retrieved: Nov 2016).