Neuronale Netze und PyTorch

Helmut Schmid

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München

Stand: 27. Juni 2023

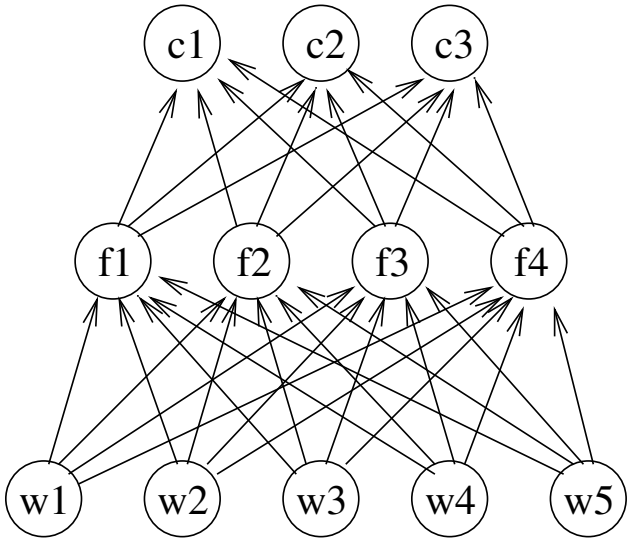
Lernen von Repräsentationen

- Conditional Random Fields und andere Modelle mit log-linearen Klassifikatoren arbeiten mit einer manuell definierten Menge von Merkmalen
- Gute Merkmale zu finden ist schwierig.
- Die gefundene Menge ist selten optimal.
- Die Suche muss wiederholt werden, wenn sich die Aufgabe ändert.

Könnte man die Merkmale automatisch lernen?

Von log-linearen Modellen zu neuronalen Netzen

grafische Darstellung eines log-linearen Modelles



Klassen: c1, c2, c3
Merkmale: f1, f2, f3, f4
Eingabewörter: w1, w2, w3, w4, w5

Jedes Merkmal f_k wird aus w_1^5 berechnet

Jede Klasse c_k wird aus f_1^4 berechnet mit

$$c_k = \frac{1}{Z} e^{\sum_i \theta_{ki} f_i} = \mathit{softmax}\left(\sum_i \theta_{ki} f_i\right)$$

Anmerkung: Hier sind die Merkmale unabhängig von den Klassen. Stattdessen gibt es für jedes Merkmal klassenspezifische Gewichte.

Idee: Die Merkmale f_i sollen analog zu den Klassen c_i berechnet/gelernt werden:

$$f_k = \mathit{act}\left(\sum_i \theta_{ki} w_i\right) = \mathit{act}(\mathit{net}_k)$$

$\Rightarrow$ neuronales Netz

Embeddings

$$f_k = act\left(\sum_i \theta_{ki} w_i\right)$$

Die Eingabe w_i muss eine numerische Repräsentation des Wortes sein.

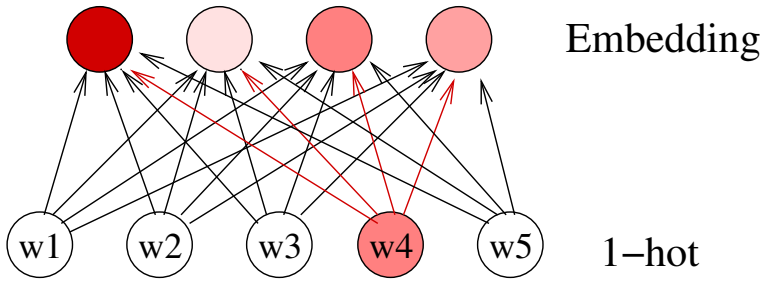
Möglichkeiten

1. binärer Vektor (1-hot-Repräsentation)
   - Beispiel: $w_i^{hot} = (0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0)$
   - Dimension = Vokabulargröße
     $\Rightarrow$ Alle Wörter sind sich gleich ähnlich/unähnlich

2. reeller Vektor (verteilte Repräsentation, distributed representation)
   - Beispiel: $w_i^{dist} = (3.275, -7.295, 0.174, 5.7332)$
   - Vektorlänge beliebig wählbar
   - Ziel: Ähnliche Wörter durch ähnliche Vektoren repräsentieren

Embeddings

Wir können die 1-hot-Repräsentation in einem neuronalen Netz mit einer Embedding-Ebene auf die verteilte Repräsentation (Embedding) abbilden:



Da nur ein 1-hot-Neuron den Wert 1 und alle anderen den Wert 0 haben, ist die verteilte Repräsentation identisch zu den Gewichten des aktiven Neurons. (Die Aktivierungsfunktion ist hier die Identität $act(x) = x$)

Die Embeddings werden wie die anderen NN-Parameter im Training gelernt.

Lookup-Tabelle

Wir können die verteilten Repräsentationen der Wörter zu einer Matrix L (Embedding-Matrix, Lookup-Tabelle) zusammenfassen. Multiplikation eines 1-hot-Vektors mit dieser Matrix liefert die verteilte Repräsentation.

$$\boldsymbol{w}^{dist} = L\boldsymbol{w}^{hot}$$

Matrix-Vektor-Multiplikation:

$$\begin{array}{cccccc|c} & & & & & & 0 & \\ & & & & & & 0 & \\ & & & & & & 1 & \boldsymbol{w}^{hot} \\ & & & & & & 0 & \\ & & & & & & 0 & \\ \hline & 3.3 & 8.5 & 7.3 & 1.9 & -7.7 & 7.3 & \\ & 4.8 & 5.1 & 4.7 & -5.3 & 3.1 & 4.7 & \\ L & -8.6 & 8.7 & 9.5 & 3.5 & 5.6 & 9.5 & \boldsymbol{w}^{dist} \\ & 6.9 & 6.4 & -4.3 & 5.7 & 3.3 & -4.3 & \\ & 7.7 & 6.7 & -8.2 & 9.7 & -9.1 & -8.2 & \end{array}$$

Aktivierungsfunktionen

Neuron: $$f_k = act\left(\sum_i \theta_{ki} w_i\right) = act(net_k)$$

In der Ausgabeebene eines neuronalen Netzes wird (wie bei log-linearen Modellen) oft die Softmax-Aktivierungsfunktion verwendet.

$$c_k = softmax(net_k) = \frac{1}{Z} e^{net_k} \quad \text{mit } Z = \sum_{k'} e^{net_{k'}}$$

Spezialfall: 2 Klassen

SoftMax: $c_k = \frac{1}{Z} e^{net_k}$ mit $net_k = \sum_i \theta_{ki} f_i = \boldsymbol{\theta}_k^T \boldsymbol{f} = \boldsymbol{\theta}_k \cdot \boldsymbol{f}$

Im Falle von zwei Klassen c_1 und c_2 ergibt sich:

$$\begin{aligned} c_1 &= \frac{e^{\boldsymbol{\theta}_1 \cdot \boldsymbol{f}}}{e^{\boldsymbol{\theta}_1 \cdot \boldsymbol{f}} + e^{\boldsymbol{\theta}_2 \cdot \boldsymbol{f}}} \frac{e^{-\boldsymbol{\theta}_1 \cdot \boldsymbol{f}}}{e^{-\boldsymbol{\theta}_1 \cdot \boldsymbol{f}}} \\ &= \frac{1}{1 + e^{-(\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2) \cdot \boldsymbol{f}}} \end{aligned}$$

Nach einer Umparametrisierung erhalten wir die Sigmoid-Funktion

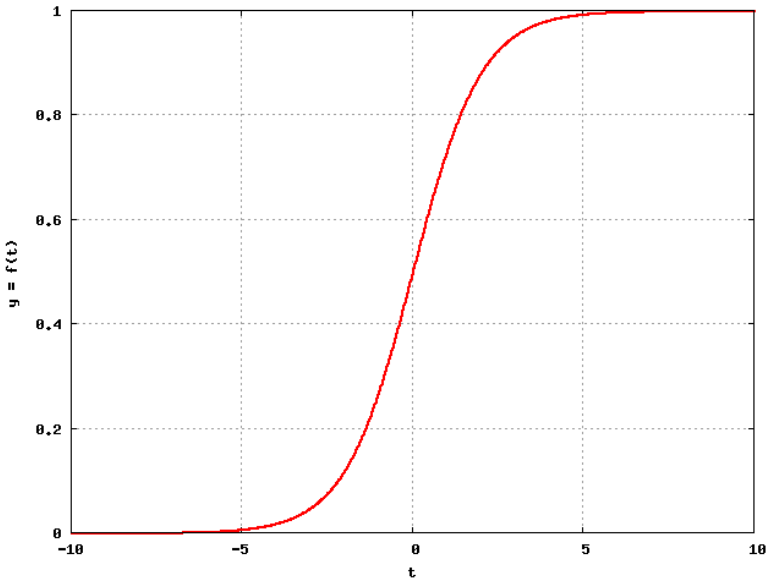
$$\begin{aligned} c_1 &= \frac{1}{1 + e^{-\theta f}} \quad \text{mit } \theta = \theta_1 - \theta_2 \\ c_2 &= 1 - c_1 \end{aligned}$$

$\Rightarrow$ Bei 2-Klassen-Problemen genügt ein Neuron.

Sigmoid-Funktion

Die Sigmoid-Funktion wird häufig als Aktivierungsfunktion der Neuronen in den versteckten Ebenen (hidden layers) genommen.

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



Man könnte also sagen, dass die versteckten Neuronen die Wahrscheinlichkeiten von binären Merkmalen berechnen.

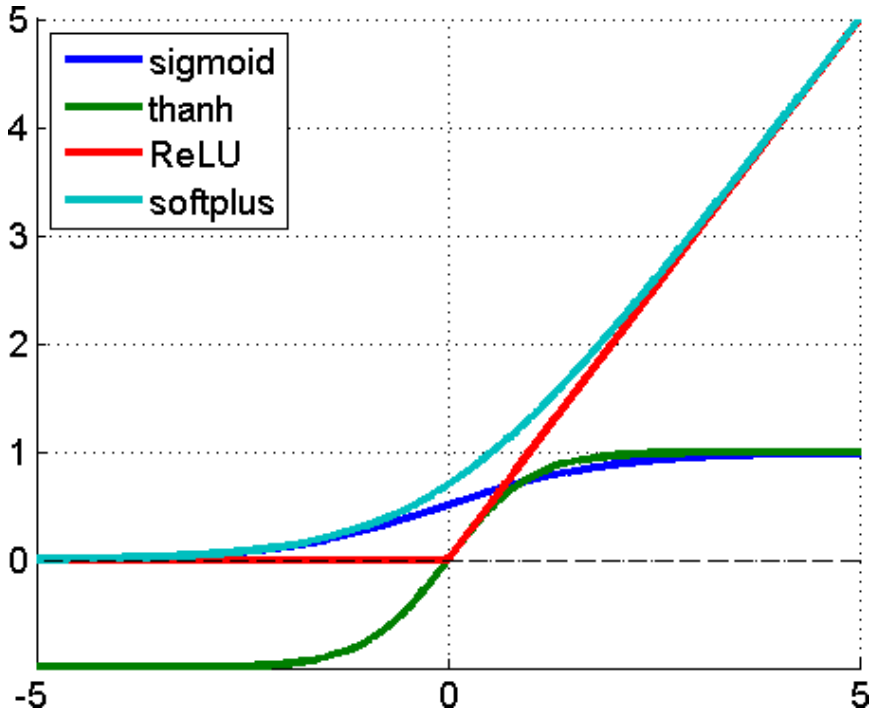
Verschiedene Aktivierungsfunktionen

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

tanh: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$

ReLU: $ReLU(x) = max(0, x)$

Softplus: $softplus(x) = ln(1 + e^x)$



Tangens Hyperbolicus tanh: entsteht durch Skalieren und Verschieben aus der Sigmoidfunktion, oft besser als Sigmoid, liefert 0 als Ausgabe, wenn alle Eingaben 0 sind

ReLUs (rectified linear units): einfach zu berechnen, bei 0 nicht differenzierbar.

Softplus ist eine überall differenzierbare Annäherung an ReLU.

Ableitungen der Aktivierungsfunktionen

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

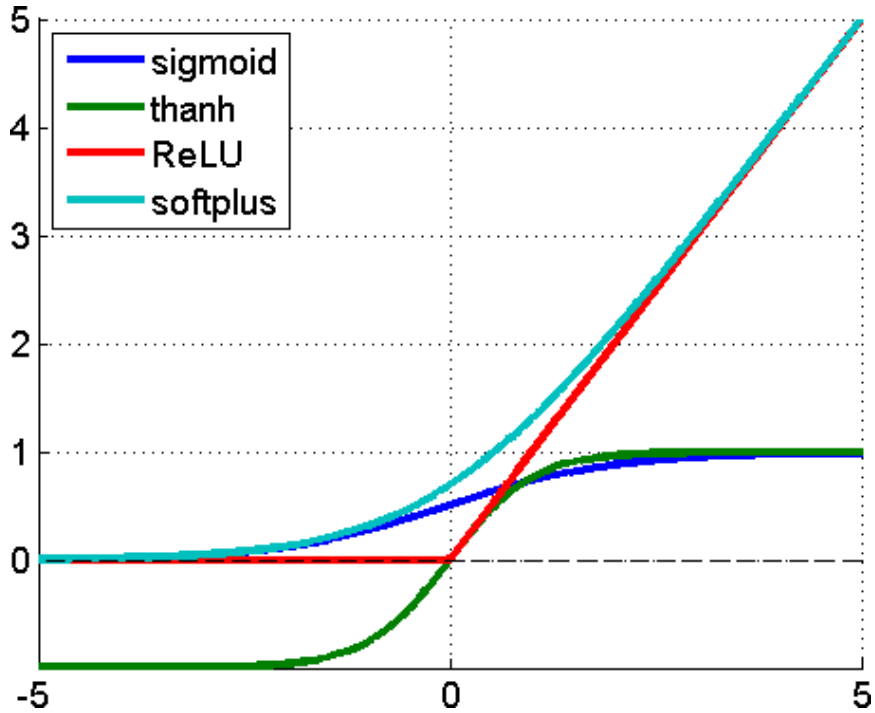
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

tanh: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$

$$tanh'(x) = 1 - \tanh(x)^2$$

ReLU: $ReLU(x) = max(0, x)$

$$ReLU'(x) = \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{falls } x < 0 \end{cases}$$



Warum ist die Aktivierungsfunktion wichtig?

Ohne (bzw. mit linearer) Aktivierungsfunktion gilt:

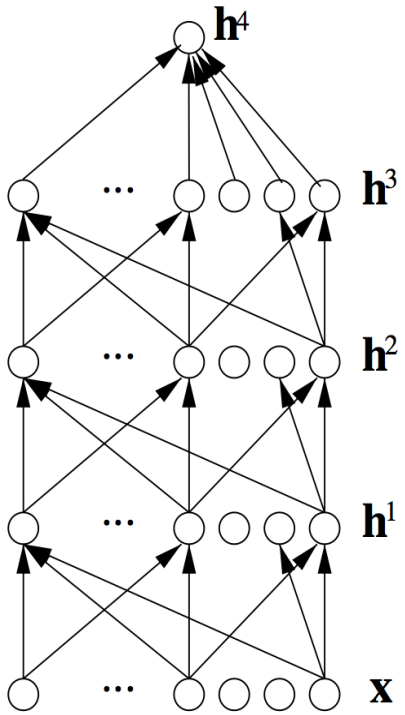
- Jede Ebene eines neuronalen Netzes führt eine lineare Transformation der Eingabe durch (= Multiplikation mit einer Matrix).
- Eine Folge von linearen Transformationen kann aber immer durch eine einzige lineare Transformation ersetzt werden.

$$W_1 W_2 W_3 x = (W_1 W_2 W_3) x = W x$$

$\Rightarrow$ Jedes mehrstufige neuronale Netz mit linearen Aktivierungsfunktionen kann durch ein äquivalentes einstufiges Netz ersetzt werden.

$\Rightarrow$ Mehrstufige Netze machen also nur Sinn, wenn nicht-lineare Aktivierungsfunktionen verwendet werden.
Ausnahme: Embedding-Schicht mit gekoppelten Parametern

Neuronale Netze



Die Aktivierung a_k eines Neurons ist gleich der Summe der gewichteten Eingaben e_i moduliert durch eine Aktivierungsfunktion *act*.

$$a_k = act(\sum_i w_{ik} e_i + b_k)$$

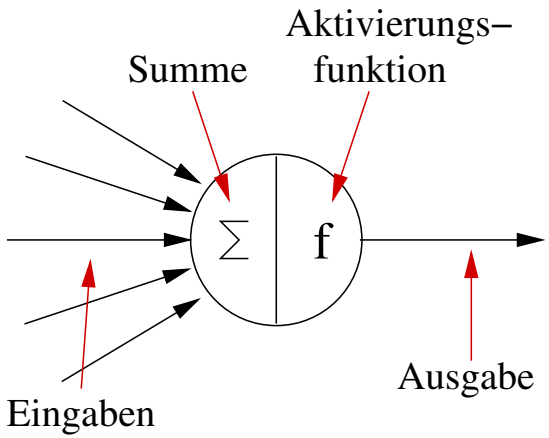
Der zusätzliche Biasterm b_k ermöglicht, dass bei einem Nullvektor als Eingabe die Ausgabe nicht-null ist.

Vektorschreibweise $a_k = act\left(\boldsymbol{w_k}^T \boldsymbol{e} + b_k\right)$
Matrixschreibweise $\boldsymbol{a} = act\left(W\boldsymbol{e} + \boldsymbol{b}\right)$

wobei *act* elementweise angewendet wird:

$$act([z_1, z_2, z_3]) = [act(z_1), act(z_2), act(z_3)]$$

trainierbare Parameter: $W, \boldsymbol{b}$



Training von neuronalen Netzen

Neuronale Netzwerke werden (ähnlich wie CRFs) trainiert, indem eine Zielfunktion, welche die Qualität der aktuellen Ausgabe misst, optimiert (d.h. minimiert oder maximiert) wird.

Die Optimierung erfolgt mit stochastischem Gradientenabstieg oder -anstieg.

Dazu wird die Ableitung der Zielfunktion nach den Netzwerkparametern (Gewichte und Biaswerte) berechnet, mit der Lernrate multipliziert und zu den Parametern addiert (Gradientenanstieg) bzw. davon subtrahiert (Gradientenabstieg).

Zielfunktionen

Die verwendeten Ausgabefunktionen und zu optimierenden Kostenfunktionen hängen von der Aufgabe ab.

- **Regression:** Jedes Ausgabeneuron liefert eine Zahl.
  Beispiel: Vorhersage der Restlebenszeit eines Patienten in Abhängigkeit von verschiedenen Faktoren wie Alter, Blutdruck, Rauch- und Essgewohnheiten, Sportaktivitäten etc.
  Aktivierungsfunktion der Ausgabeneuronen: $o = act(x) = e^x$
  Kostenfunktion: $(y - o)^2$ quadrierter Abstand zwischen gewünschter Ausgabe y und tatsächlicher Ausgabe o

- **Klassifikation:** n mögliche disjunkte Ausgabeklassen repräsentiert durch je ein Ausgabeneuron
  Beispiel: Diagnose der Krankheit eines Patienten anhand der Symptome
  Aktivierungsfunktion der Ausgabeneuronen: $o = softmax(x)$
  Kostenfunktion: $\log(y^T o)$ (Loglikelihood, Crossentropie)

Deep Learning

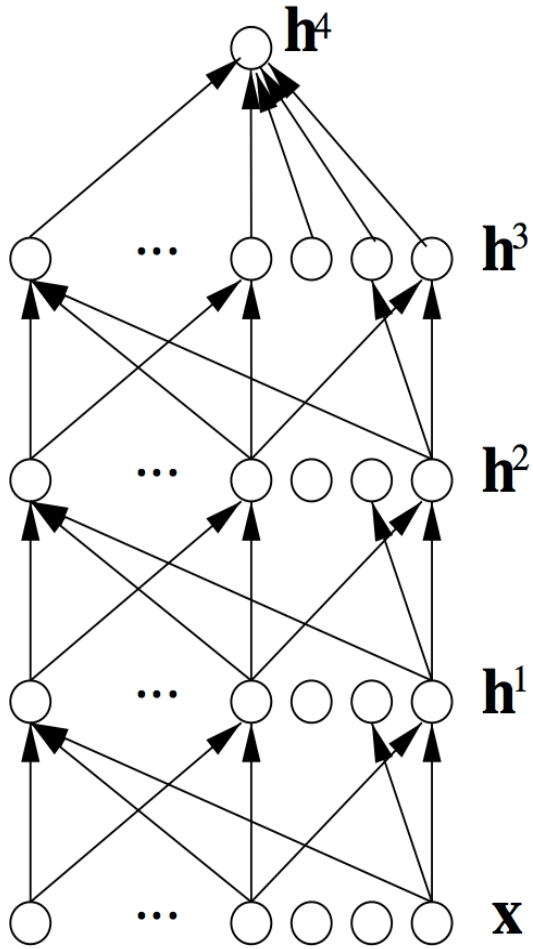
Neuronale Netze mit mehreren versteckten Ebenen nennt man tiefe Netze → Deep Learning.

Bei der Anwendung eines neuronalen Netzes werden die Eingabewerte **x** über die Neuronen der versteckten Ebenen $h_1, ..., h_3$ bis zur Ausgabeebene h_4 propagiert.

Je höher die versteckte Ebene, desto komplexer sind die darin repräsentierten Merkmale.

Im Training werden alle Parameter (Gewichte, Biases) gleichzeitig optimiert. Dazu wird der Gradient der Zielfunktion an den Ausgabeneuronen berechnet und bis zu den Eingabeneuronen zurückpropagiert.

→ Backpropagation

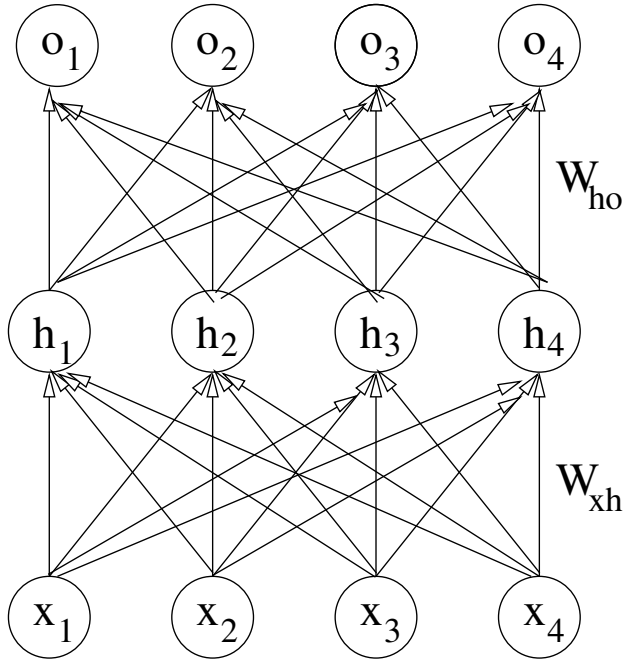


Arbeitsweise eines neuronalen Netzes

- Forward-Schritt: Die Aktivierung der Eingabeneuronen wird über die Neuronen der versteckten Ebenen zu den Neuronen der Ausgabeebene propagiert.
- Backward-Schritt: Im Training wird anschließend der Gradient einer zu optimierenden Zielfunktion an den Ausgabeneuronen berechnet und zu den Eingabeneuronen zurückpropagiert.

Forward-Propagation

Ein neuronales Netz mit Eingabevektor $\boldsymbol{x}$, verstecktem Vektor $\boldsymbol{h}$ und skalarem Ausgabewert o kann wie folgt definiert werden:



$$
\begin{aligned}
\boldsymbol{h}(\boldsymbol{x}) &= \tanh(W_{hx}\boldsymbol{x} + \boldsymbol{b}_h) \\
\boldsymbol{o}(\boldsymbol{x}) &= \mathrm{softmax}(W_{oh}\boldsymbol{h} + \boldsymbol{b}_o) \\
&= \mathrm{softmax}(W_{oh}\tanh(W_{hx}\boldsymbol{x} + \boldsymbol{b}_h) + \boldsymbol{b}_o)
\end{aligned}
$$

Das Training maximiert die Log-Likelihood der Daten

$$
\begin{aligned}
LL(\boldsymbol{x}, \boldsymbol{y}) &= log(\boldsymbol{y}^T \boldsymbol{o}) \\
&= log(\boldsymbol{y}^T \mathrm{softmax}(W_{oh}\tanh(W_{hx}\boldsymbol{x} + \boldsymbol{b}_h) + \boldsymbol{b}_o))
\end{aligned}
$$

$\boldsymbol{y}$ repräsentiert die gewünschte Ausgabe als 1-hot-Vektor

Backward-Propagation

Für die Parameteroptimierung müssen die Ableitungen des Ausdruckes

$$LL(\boldsymbol{x}, \boldsymbol{y}) = log(\boldsymbol{y}^T \text{softmax}(W_{oh} \tanh(W_{hx}\boldsymbol{x} + \boldsymbol{b}_h) + \boldsymbol{b}_o))$$

bzgl.der Parameter W_{hx}, $\boldsymbol{b}_h$, W_{oh} und $\boldsymbol{b}_o$ berechnet werden.

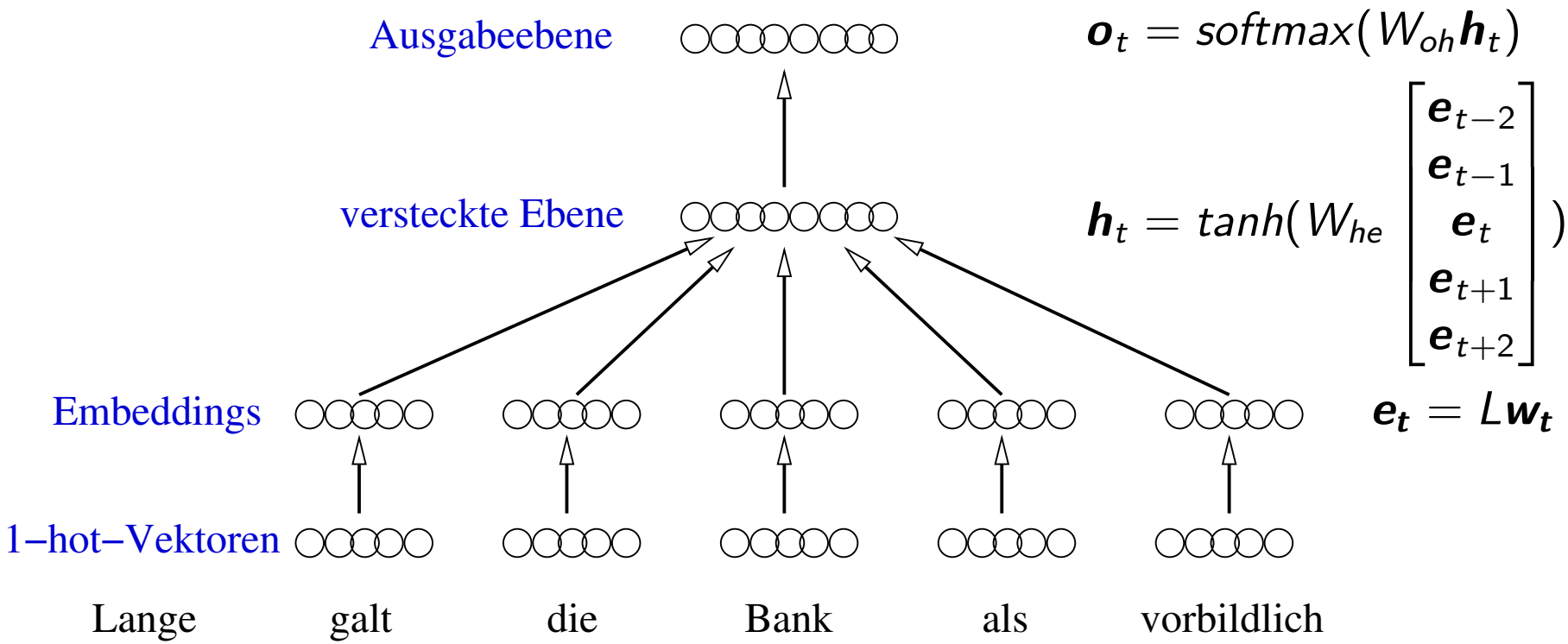
Am besten benutzt man dazu **automatische Differentiation**.

Anschließend Anpassung der Parameter θ mit Gradientenanstieg:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \nabla LL_{\boldsymbol{\theta}_t}$$

θ_t steht dabei für die trainierbaren Parameter W_{hx}, W_{oh}, b_h und b_o.

Einfaches neuronales Netz für Wortart-Annotation



Numpy

- Python-Bibliothek für numerische Aufgaben
- Datentypen: Skalare, Vektoren, Matrizen und Tensoren
- viele Operatoren (z.B. Matrixprodukte)
- effiziente C-Implementierungen der Operatoren

Programmierung von neuronalen Netzen

PyTorch

- Python-Bibliothek für neuronale Netze und andere komplexe mathematische Funktionen
- Syntax ähnelt Numpy
- automatisches Differenzieren von Funktionen ($\rightarrow$ NN-Training)
- GPU-Unterstützung

PyTorch-Beispiel: Netzwerkparameter

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from functools import reduce
import operator

class Net(nn.Module):

    def __init__(self):
        super().__init__()

        # 1 input channel, 6 output channels, 5x5 convolution kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)

        # Feedforward layers
        self.ff1 = nn.Linear(16 * 5 * 5, 120)
        self.ff2 = nn.Linear(120, 80)
        self.ff3 = nn.Linear(80, 10)
```

PyTorch-Beispiel: Forward-Funktion

```
...
class Net(nn.Module):
        ...
    def forward(self, x):
        # Convolution and max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))

        # Flatten the feature vectors
        x = x.view(x.size()[0], -1)  # 0 is the batch dimension

        # Feedforward layers
        x = F.relu(self.ff1(x))
        x = F.relu(self.ff2(x))
        x = self.ff3(x)

        return x
```

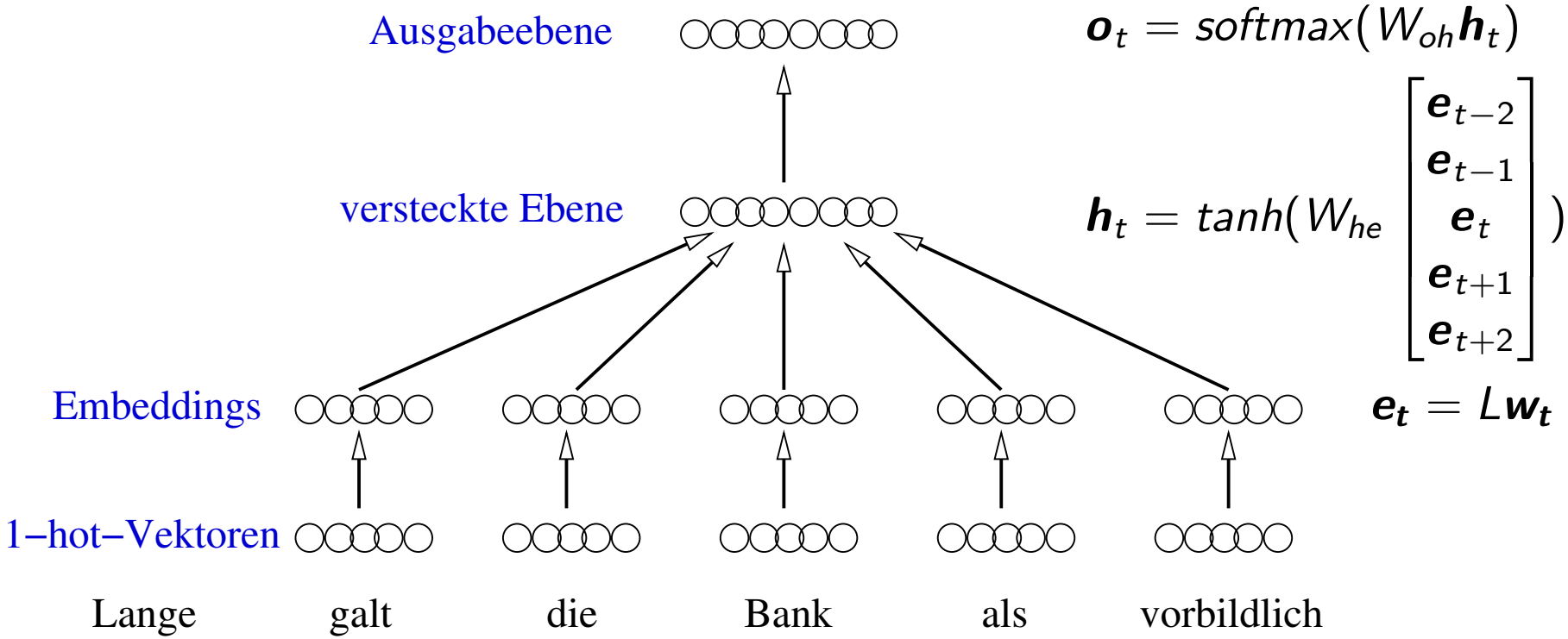
PyTorch-Beispiel Training

```
net = Net()  # Create a network instance
criterion = nn.MSELoss()  # Create a loss function
optimizer = optim.SGD(net.parameters(), lr=0.01)  # Create an optimizer

# Create and process a random input matrix (just for demonstration)
input  = torch.randn(1, 1, 32, 32)
output = net(input) # Call the forward method

# Define a random output vector (for demonstration)
target = torch.arange(1, 11, dtype=torch.float)
target = target.unsqueeze(0)  # add a batch dimension

# Training steps
net.zero_grad()   # No accumulation of gradients
loss = criterion(output, target)
loss.backward()   # Gradient computation
optimizer.step()  # Parameter update
```

Batchverarbeitung

- Die vordefinierten PyTorch-NN-Module erwarten eine Folge von Trainingsbeispielen (Minibatch) als Eingabe.
- Die Minibatch-Verarbeitung lastet eine GPU besser aus.
- Wenn auf Sätzen trainiert wird, müssen alle Sätze dieselbe Länge haben, damit sie zu einer Matrix zusammengefasst werden können.
- Kürzere Sätze müssen daher mit Dummysymbolen aufgefüllt werden (Padding).

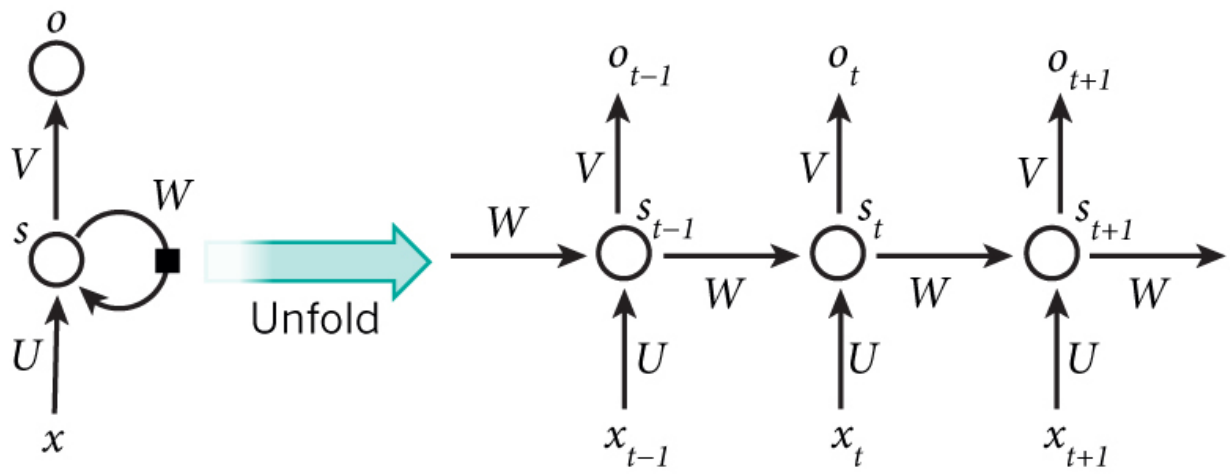
Einfaches neuronales Netz für Wortart-Annotation



"Feedforward"-Netze haben ein beschränktes Eingabefenster und können daher keine langen Abhängigkeiten erfassen.

→ rekurrente Netze

Rekurrente Neuronale Netze

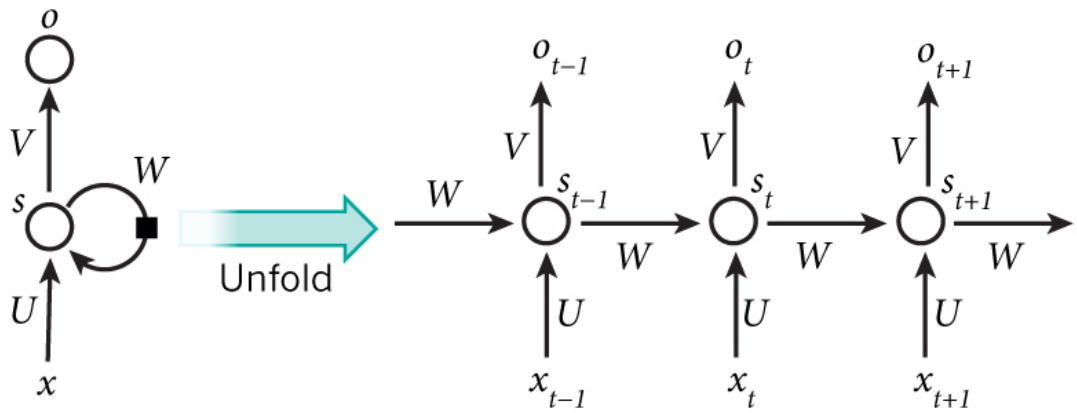


http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns

Ein RNN ist äquivalent zu einem sehr tiefen Feedforward-NN mit gekoppelten Gewichtsmatrizen.

Das neuronale Netz lernt, relevante Information über die Vorgängerwörter im Zustand s_t zu speichern.

Rekurrentes neuronales Netz für Wortart-Annotation

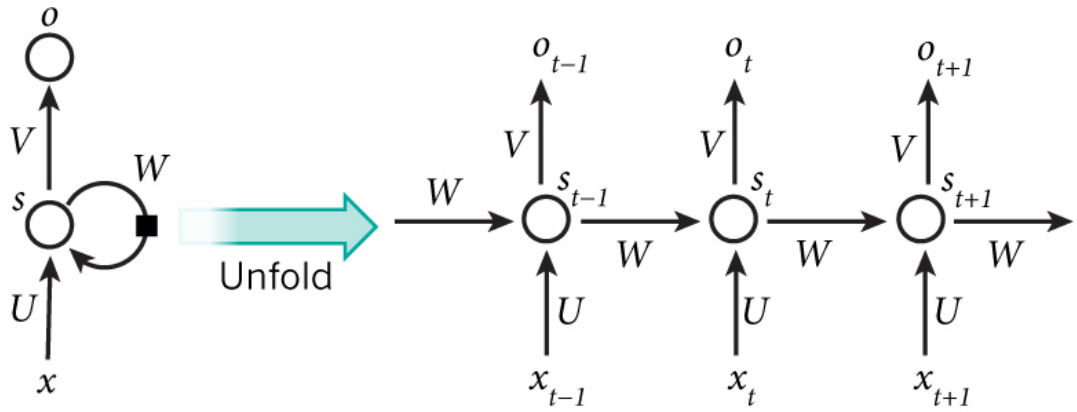


In jedem Schritt

- wird ein Wort x_t gelesen
- ein neuer Hidden State s_t (= Aktivierungen der versteckten Ebene) berechnet
- eine Wahrscheinlichkeitsverteilung über mögliche Tags ausgegeben o_t

(Das Netzwerk hat noch keine Information über den rechten Kontext. Dazu später mehr.)

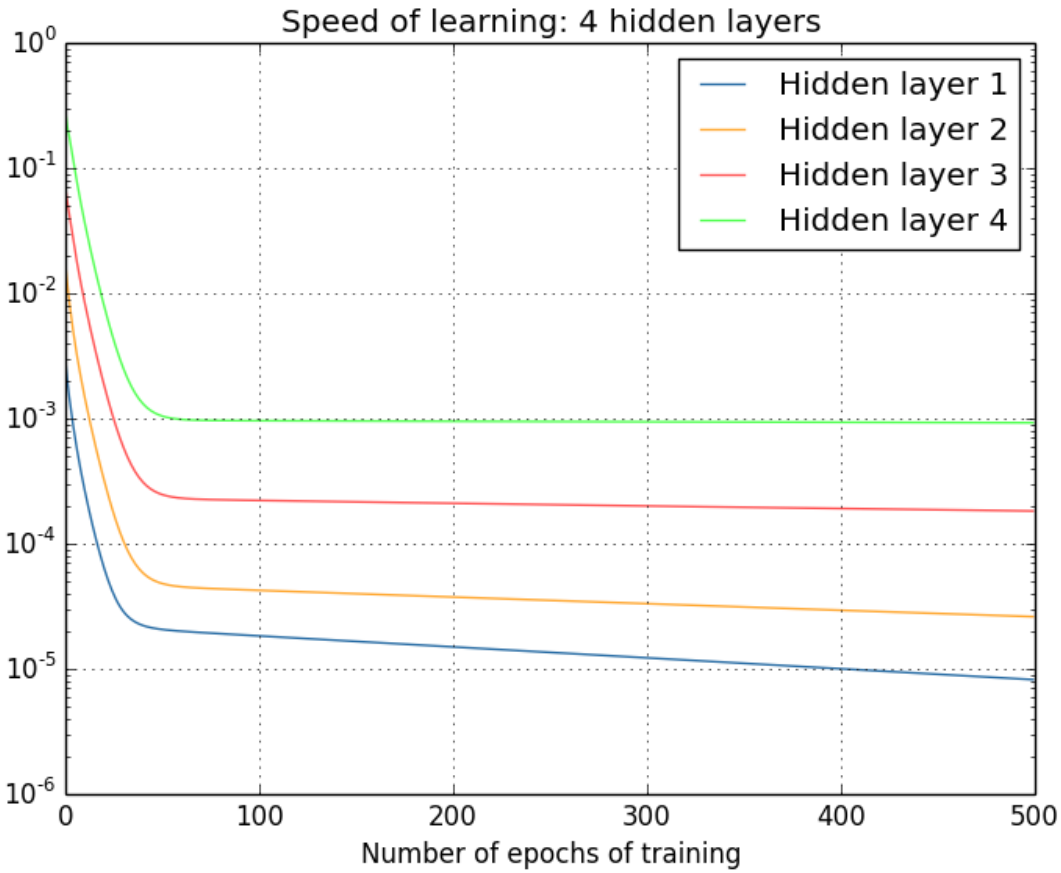
Backpropagation Through Time (BPTT)



- Zum Trainieren wird das rekurrente Netz zu einem Feedforward-Netz aufgefaltet.
- Der Gradient wird von den Ausgabeneuronen o_t zu den versteckten Neuronen s_t und von dort zu den Eingabeneuronen x_t und den vorherigen versteckten Neuronen s_{t-1} propagiert (Backpropagation through time).
- An den versteckten Neuronen werden zwei Gradienten addiert.

Verschwindender/Explodierender Gradient

Das Training tiefer neuronaler Netze ist schwierig, weil der Gradient beim Zurückpropagieren meist schnell kleiner (oder größer) wird.



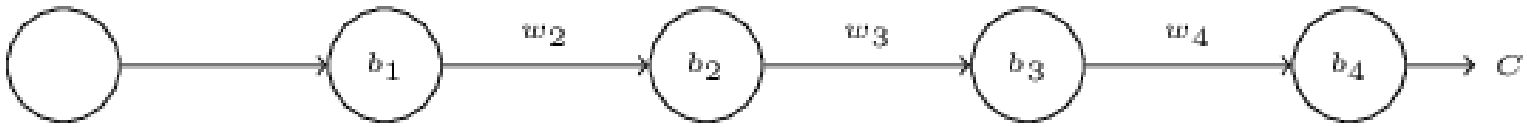
`http://neuralnetworksanddeeplearning.com/chap5.html`

Verschwindender/Explodierender Gradient

Warum wird der Gradient exponentiell kleiner mit der Zahl der Ebenen?

Betrachten wir ein neuronales Netz mit 5 Ebenen und einem Neuron pro Ebene

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



http://neuralnetworksanddeeplearning.com/chap5.html

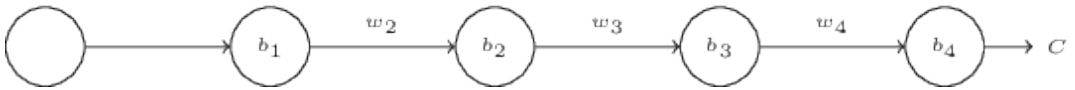
w_i ist ein Gewicht, b_i ein Bias, C die Kostenfunktion, a_i die Aktivierung eines Neurons und z_i die gewichtete Eingabe eines Neurons

Der Gradient wird in jeder Ebene mit dem Ausdruck $w_i \times \sigma'(z_i)$ multipliziert.

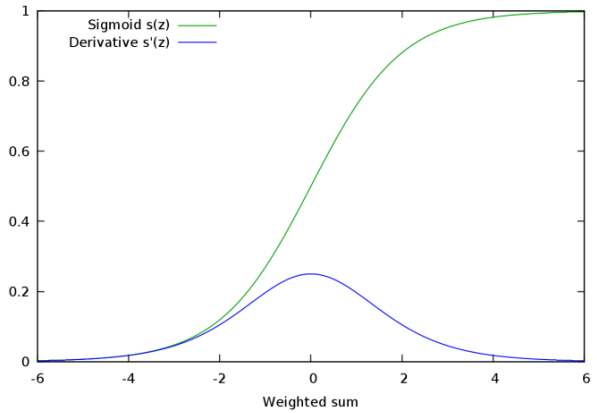
Wie groß ist dieser Ausdruck?

Verschwindender/Explodierender Gradient

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



http://neuralnetworksanddeeplearning.com/chap5.html



http://whiteboard.ping.se/MachineLearning/BackProp

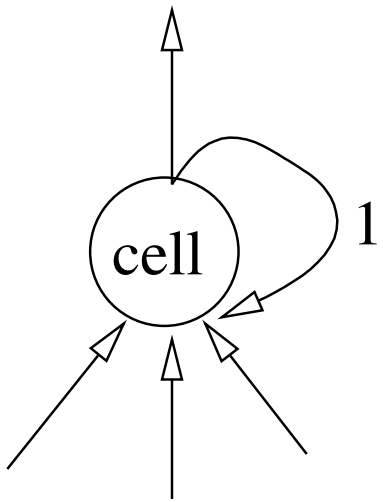
Wenn die Gewichte mit zufälligen Werten im Bereich [-1,1] initialisiert werden, dann gilt $|w_i \times \sigma'(z_i)| < 0.25$, da $|\sigma'(z_i)| < 0.25$.

Damit wird der Gradient exponentiell kleiner mit der Zahl der Ebenen.

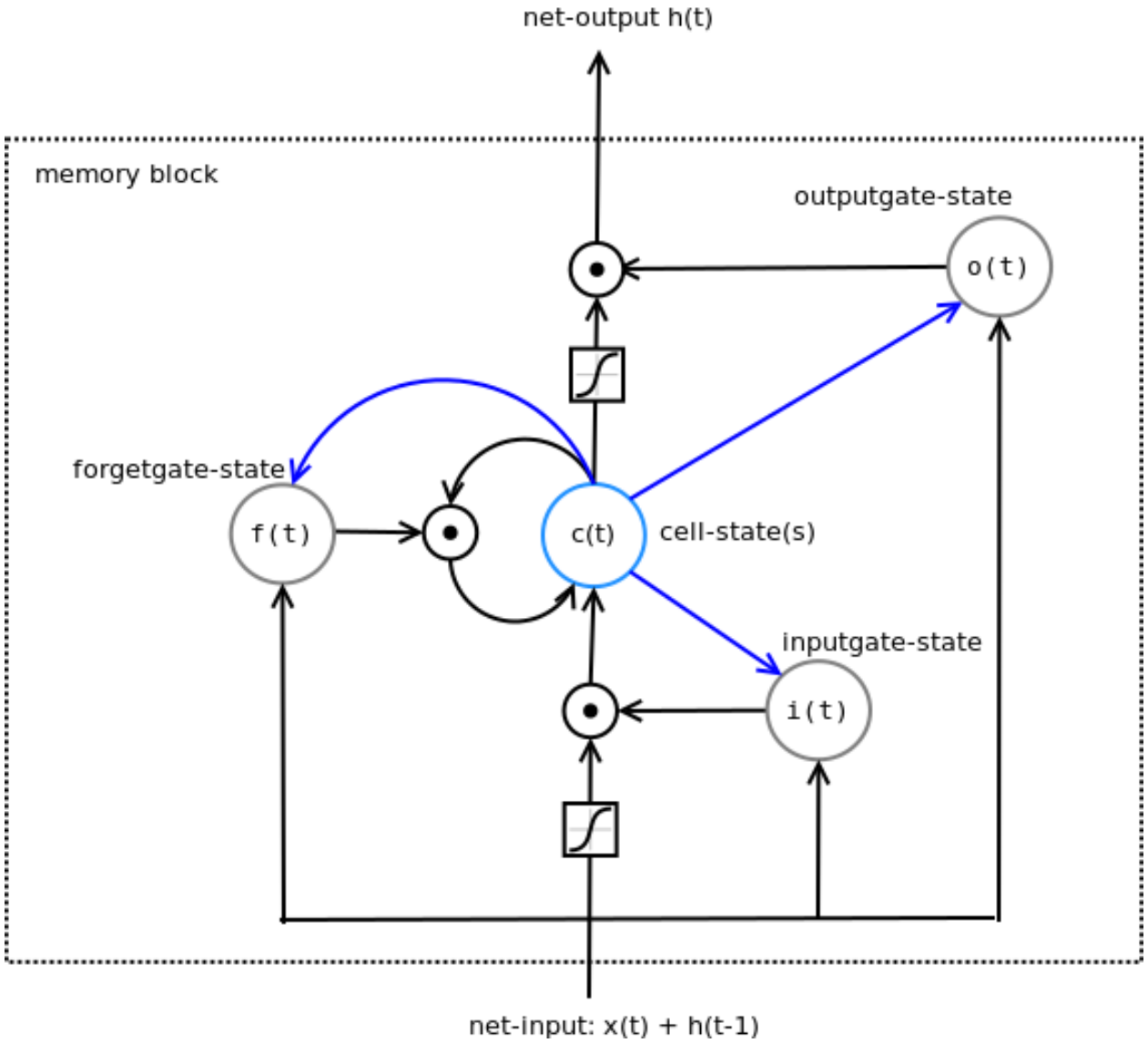
Wenn die Gewichte mit großen Werten initialisiert werden, kann der Gradient auch explodieren.

Long Short Term Memory

- entwickelt von Sepp Hochreiter und Jürgen Schmidhuber (TUM)
- löst das Problem der instabilen Gradienten für rekurrente Netze
- Eine Speicherzelle (cell) bewahrt den Wert des letzten Zeitschritts.
- Der Gradient wird beim Zurückpropagieren nicht mehr mit Gewichten multipliziert und bleibt über viele Zeitschritte erhalten.



Long Short Term Memory: Schaltplan



http://christianherta.de/lehre/dataScience/machineLearning/neuralNetworks/LSTM.php

Long Short Term Memory

Berechnung eines LSTMs (ohne Peephole-Verbindungen)

$$
\begin{aligned}
z_t &= \tanh(W_z x_t + R_z h_{t-1} + b_z) && \text{(input activation)} \\
i_t &= \sigma(W_i x_t + R_i h_{t-1} + b_i) && \text{(input gate)} \\
f_t &= \sigma(W_f x_t + R_f h_{t-1} + b_f) && \text{(forget gate)} \\
o_t &= \sigma(W_x x_t + R_o h_{t-1} + b_o) && \text{(output gate)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot z_t && \text{(cell)} \\
h_t &= o_t \odot tanh(c_t) && \text{(output activation)}
\end{aligned}
$$

mit $a \odot b = (a_1 b_1, a_2 b_2, ..., a_n b_n)$

Die LSTM-Zellen ersetzen einfache normale Neuronen in rekurrenten Netzen.

Man schreibt kurz:

$$\boldsymbol{z} = LSTM(\boldsymbol{x})$$

Long Short Term Memory

Vorteile

- löst das Problem mit instabilen Gradienten
- exzellente Ergebnisse in vielen Einsatzbereichen

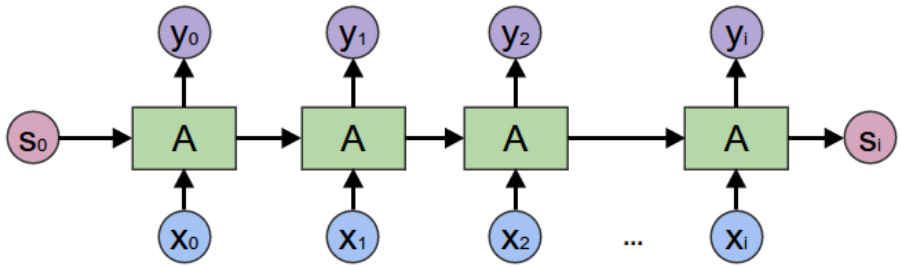
Nachteile

- deutlich komplexer als ein normales Neuron
- höherer Rechenzeitbedarf

Alternative

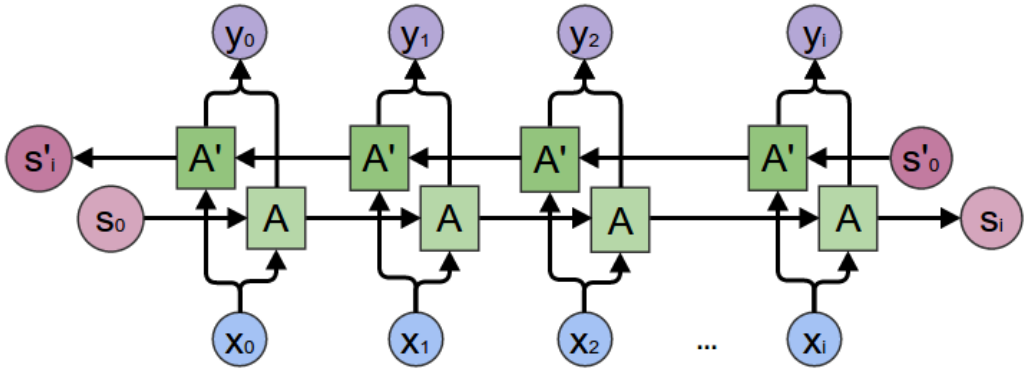
- Gated Recurrent Units (GRU) (Cho et al.)
- etwas einfacher (nur 2 Gates)

Bidirektionale RNNs



colah.github.io/posts/2015-09-NN-Types-FP/img/RNN-general

- Die rekurrenten Neuronen repräsentieren alle bisherigen Eingaben.
- Für viele Anwendungen ist aber auch eine Repräsentation der folgenden Eingaben nützlich. ⇒ bidirektionales RNN



colah.github.io/posts/2015-09-NN-Types-FP/img/RNN-bidirectional

- Bidirektionale RNNs können zu tiefen bidirektionalen RNNs gestapelt werden