# Deep Feedforward Netze und Rekurrente Neuronale Netze

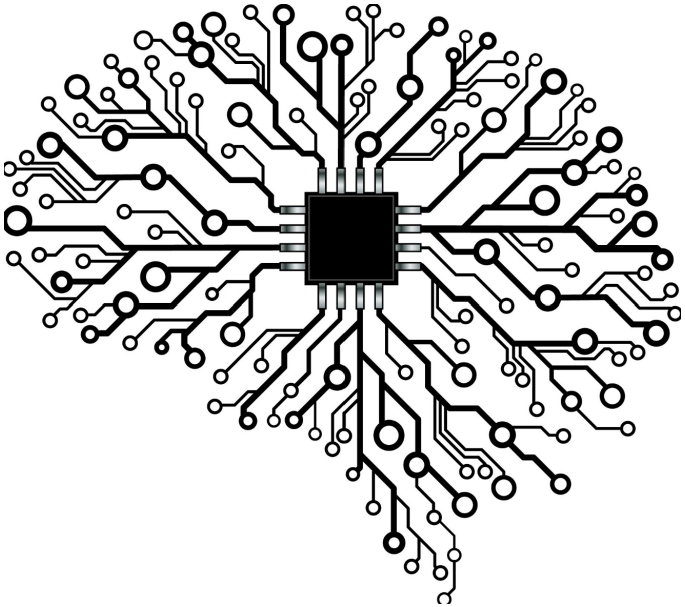
Nico Frisch, Wenhao Peng, Tim Vieth und Mattes Westdörp

30. Dezember 2020

# Deep Feedforward Netze

# Einführung

- Meiste Künstliche Neuronale Netze sind eine Klasse nichtlinearer Lernalgorithmen, die ein biologisches Vorbild haben
- Anwendbar auf große Breite von Problemen, z.B. Klassifizierungsprobleme und Regression.



https://istd.sutd.edu.sg/files/Artificial-Intelligence.jpg

- Künstliche Neuronale Netze können grob als Zusammensetzung von multivariaten Funktionen mit veränderlichen Parametern angesehen werden
- Ziel ist es, einen Input in einen Output zu überführen, um so eine unbekannte Funktion darzustellen
- Zunächst geben wir eine Einführung von *Multi-Layer-Perceptrons (MLPs)* als Art von Deep Feedforward Netzen

# Allgemeine Architektur

Ein MLP besteht aus

## Layer

- einer Anzahl *Layern* $1, \ldots, L$, mit Neuronen $a_1^{[i]}, \ldots, a_{n_i}^{[i]}$ in Schicht $i$, wobei $n_1, \ldots, n_L \in \mathbb{N}$ für die Anzahl der Neuronen der Schicht steht.
- Der erste Layer ist der *Input Layer*, der letzte heißt *Output Layer* und dazwischen befinden sich die *Hidden Layer*

Die Werte der Neuronen jeder Schicht werden anhand der Outputs der Neuronen der vorherigen Schicht berechnet.

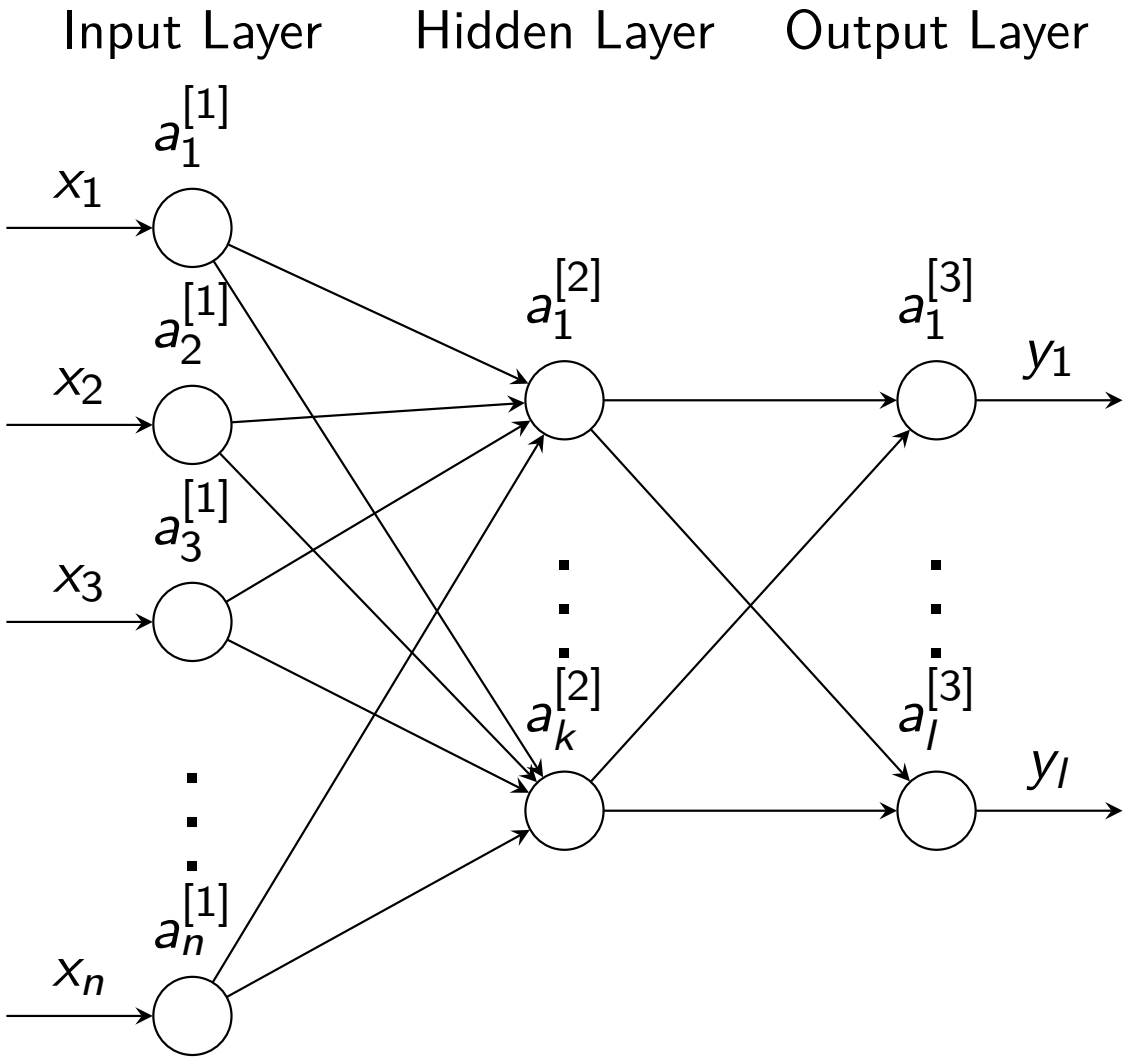



Abbildung: Neuronales Netz mit 3 Schichten

# Outputs der Neuronen

Für die Berechnung zwischen den Schichten brauchen wir

- *Gewichtsmatrizen* $W^{[2]}, \ldots, W^{[L]}$ , mit $W^{[i]} \in \mathbb{R}^{n_i \times n_{i-1}}$ für $i \in \{2, \ldots, L\}$,
- *Bias* $b_i \in \mathbb{R}^{n_i}$ für $i \in \{2, \ldots, L\}$.

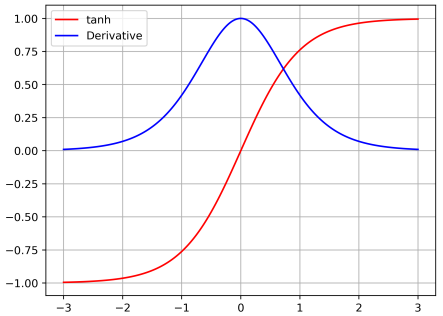
Die Gewichtsmatrizen skalieren den Input, während die Biases den Input verschieben.

- Außerdem brauchen wir für die Skalierung der Outputs auf ein Intervall in $\mathbb{R}$ die sogenannte *Aktivierungsfunktion*. $\phi$
- Diese kann als Grad der Aktivierung eines Neurons verstanden werden.

# Typische Aktivierungsfunktionen:

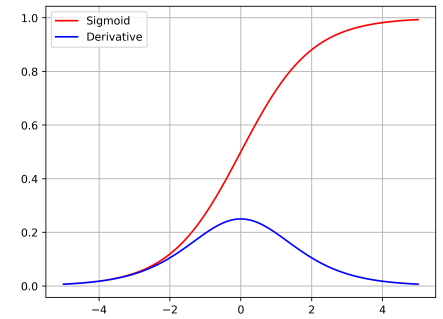
- Tangens hyperbolicus

$$\tanh\colon \mathbb{R} \to (-1,1)$$
$$x \mapsto \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



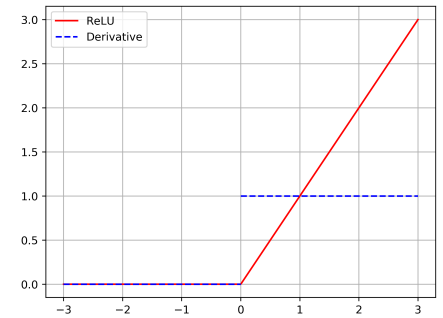

- Sigmoid

$$\sigma\colon \mathbb{R} \to (0,1)$$
$$x \mapsto \frac{1}{1 - e^{-x}}$$




- Rectified Linear Unit („ReLU“)

$$\mathrm{ReLU}\colon \mathbb{R} \to \mathbb{R}_0^+$$
$$x \mapsto \max\{x, 0\}$$



Die Berechnung der Werte der Neuronen in Schicht $i$ ergibt sich dann aus

$$a^{[i]} = \phi(W^{[i]} a^{[i-1]} + b^{[i]}) \in \mathbb{R}^{n_i} \quad \text{für } i \in \{2, ..., L\},$$

und explizit für Neuron $k \in \{1, ..., n_i\}$ in Schicht $i$

$$a_k^{[i]} = \phi\left(\sum_{j=1}^{n_{i-1}} w_{kj}^{[i]} a_j^{[i-1]} + b_k^{[i]}\right).$$

Zur Vereinfachung der Notation schreiben wir auch

$$a^{[i]} =: \phi(z^{[i]}),$$

wobei wir $z^{[i]}$ als *gewichteten Input* bezeichnen.

# Beispiel Ziffernerkennung

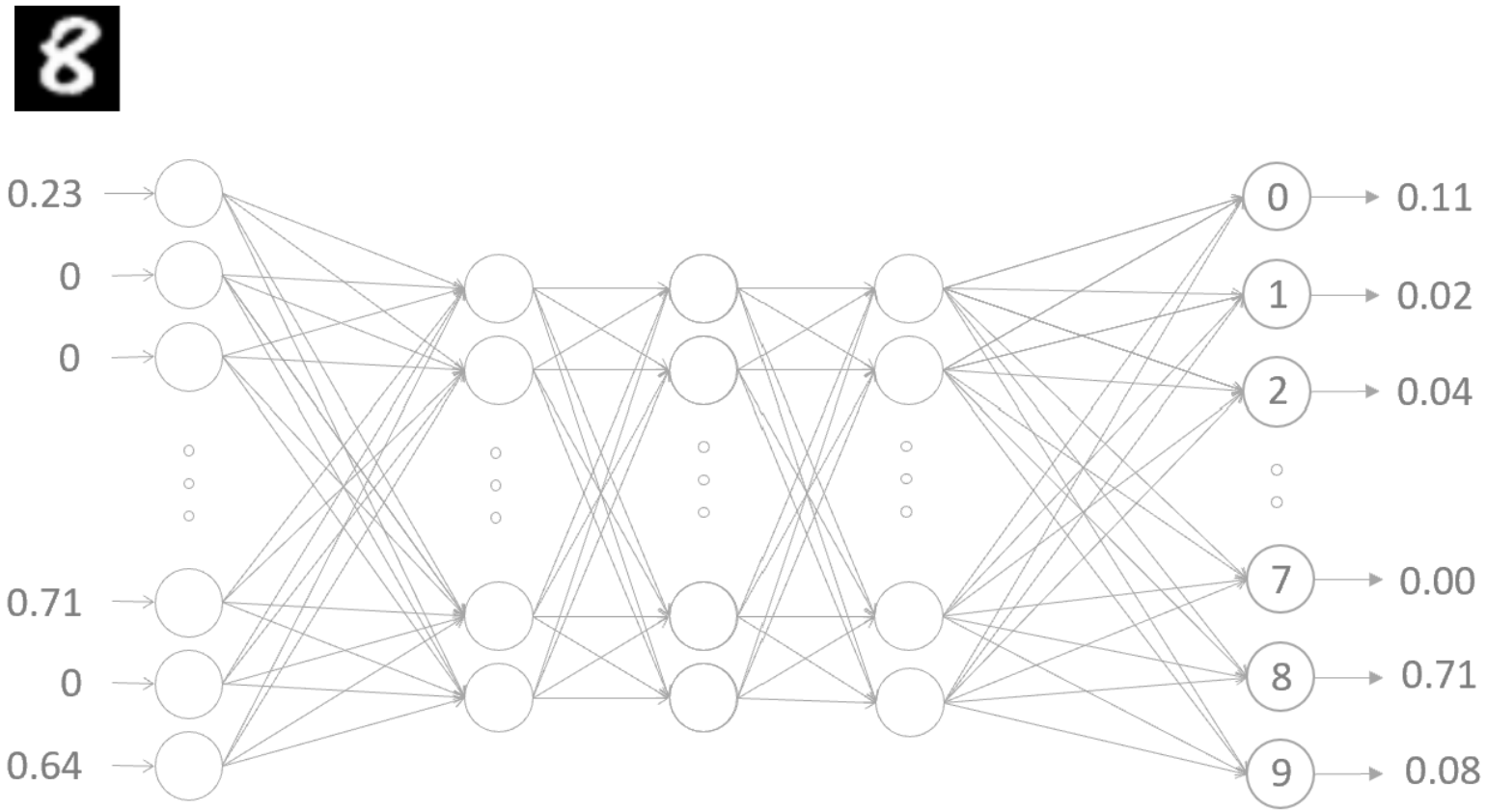



Abbildung: Beispiel für ein neuronales Netz zur Ziffernerkennung.

# Kostenfunktion

Wir führen die *Kostenfunktion* ein, um die Güte der Vorhersage auf den Trainingsdaten zu bestimmen. Wenn die Gesamtzahl der Gewichte und Parametern $d \in \mathbb{N}$ ist, gilt

$$C \colon \mathbb{R}^d \to \mathbb{R}.$$

Ein einfaches Beispiel ist der *Mean Squared Error*

$$MSE = \frac{1}{n} \sum_{i=1}^{n} \left|\left| y_i - a_i^{[L]} \right|\right|_2^2.$$

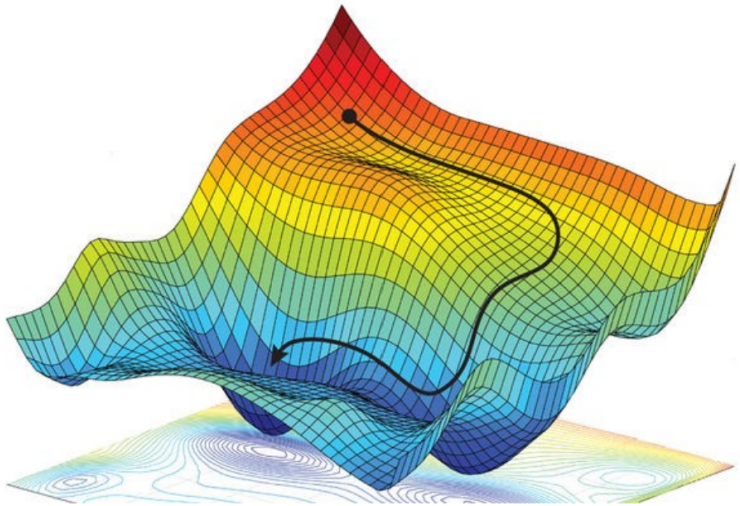
für Trainingsdaten mit Zuordnung $y_1, ..., y_n$ und Vorhersagen $a_1^{[L]}, ..., a_n^{[L]}$.

# Gradientenverfahren

- Betrachte Veränderung der Gewichte und Biases in Abhängigkeit zur Kostenfunktion.
- Für Gewichte und Biases der Anzahl $d \in \mathbb{N}$ wird ein Vektor in $\mathbb{R}^d$ gesucht, der die Kostenfunktion minimiert.

$\rightarrow$ Minimierung mit Hilfe des Gradientenverfahren

- Starte mit Initialisierung $p$ der Gewichte
- Berechne Gradienten der Kostenfunktion, und wähle $p \leftarrow p - \eta \nabla C(p)$
- Dabei ist $\eta > 0$ die *Lernrate*
- Wiederhole, bis lokales Minimum gefunden ist



Source: https://reconsider.news/2018/05/09/ai-researchers-allege-machine-learning-alchemy/

Bedeutung der Trainingsdaten im Gradientenverfahren

- Verfahren kann auf gesamten Trainingsdaten ausgeführt werden $\rightarrow$ sehr teuer
- Vereinfachung durch Partition in *Batches* und bilden eines durchschnittlichen Gradienten
- Ausführung über Gesamtheit der Batches nennt sich *Epoche*

Verfahren zur Berechnung des Gradienten in Richtung der Gewichte und Biases. Sei hier

$$C := \frac{1}{2} \left|\left| y - a^{[L]} \right|\right|_2^2 .$$

## Backpropagation Lemma

Für die Ableitung von C in Richtung des gewichteten Inputs der Neuronen ergibt sich mit $\delta^{[i]} := \frac{\partial C}{\partial z^{[i]}}$

$$\delta^{[L]} = \phi'(z^{[L]})(a^{[L]} - y),$$

sowie

$$\delta^{[i]} = \phi'(z^{[i]})(W^{[i+1]})^T \delta^{[i+1]} \qquad \text{für } i \in \{2, \ldots, L-1\}.$$

## Backpropagation Lemma

Für die Ableitungen in Richtung der Bias, bzw. Gewichte gilt

$$\frac{\partial C}{\partial b_j^{[i]}} = \delta_j^{[i]}$$

und

$$\frac{\partial C}{\partial w_{jk}^{[i]}} = \delta_j^{[i]} a_k^{[i-1]} \qquad \text{für } i \in \{2, \ldots, L\}.$$

$\rightarrow$ Anwendung des Gradientenverfahrens möglich

Update Gewichte und Bias durch

$$b_j^{[i]} \leftarrow b_j^{[i]} - \eta \frac{\partial C}{\partial b_j^{[i]}} = b_j^{[i]} - \eta \delta_j^{[i]},$$

und

$$w_{jk}^{[i]} \leftarrow w_{jk}^{[i]} - \eta \frac{\partial C}{\partial w_{jk}^{[i]}} = w_{jk}^{[i]} - \eta \delta_j^{[i]} a_k^{[i-1]},$$

wobei $\eta$ die Lernrate ist.

# Vanishing Gradient Problem

Betrachte neuronales Netz mit 4 Schichten aus je einem Neuron und Sigmoid als Aktivierungsfunktion, also

$$f(x) = \sigma(w^{[4]}\sigma(w^{[3]}\sigma(w^{[2]}x + b^{[2]}) + b^{[3]}) + b^{[4]}).$$

Das Backpropagation Lemma ergibt

$$\frac{\partial C}{\partial w^{[2]}} = a^{[1]}\sigma'(z^{[2]})w^{[3]}\sigma'(z^{[3]})w^{[4]}\sigma'(z^{[4]})\left(a^{[4]} - y\right),$$

wobei $\sigma'(x) = \sigma(x)\left(1 - \sigma(x)\right) \in (0, 0.25]$.

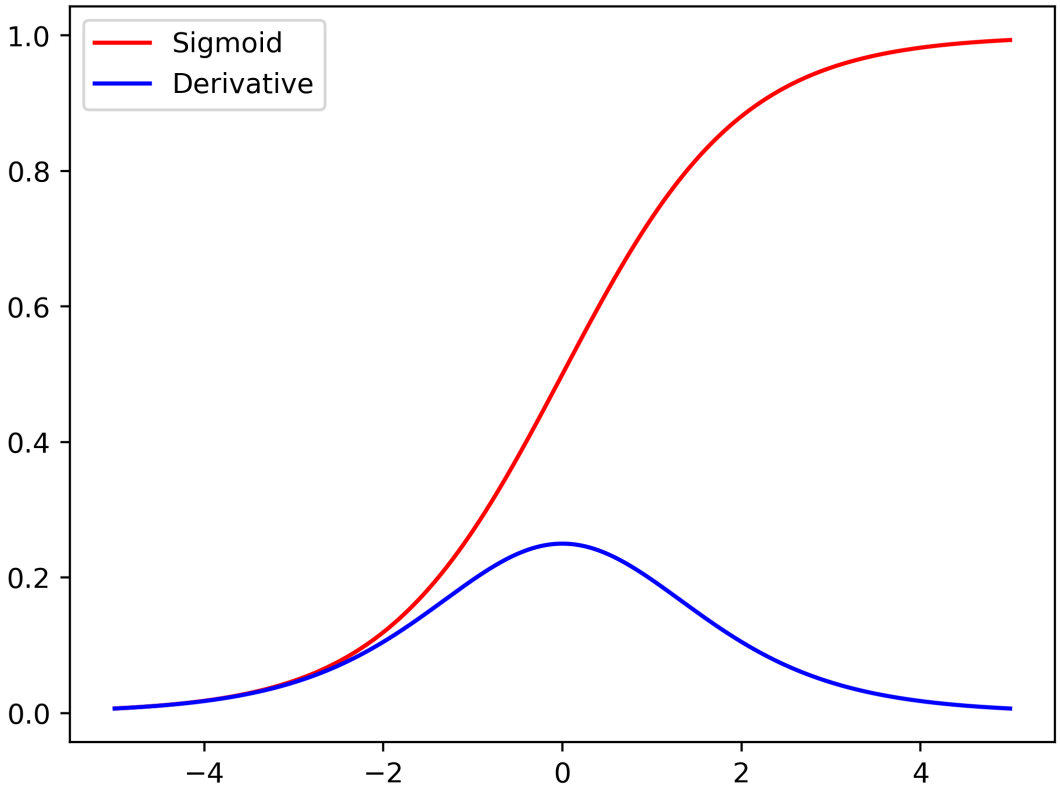



Abbildung: Sigmoid und Ableitung

Wir erhalten die Abschätzung

$$\frac{\partial C}{\partial w^{[2]}} \leq 0.25^3 \cdot a^{[1]} w^{[3]} w^{[4]} \left( a^{[4]} - y \right).$$

Die Werte von $W$ werden typischerweise klein initialisiert
$\rightarrow$ Der gesamte Ausdruck wird vom Term der Aktivierungsfunktion dominiert. Der Effekt wird verstärkt, je tiefer das Netz ist.

- Problem: Ableitung kann sehr klein werden, insb. bei tiefen Netzen $\rightarrow$ *Vanishing Gradient*
- Erhöhung der Werte der Gewichte $\rightarrow$ *Exploding Gradient*
- Lösungsansätze: ReLU als Aktivierungsfunktion, oder *Skip Connection*

# Implementierung in R

Beispiel für ein Deep Feedforward Netz mit einem Hidden Layer in R.

```
1 inputs <- layer_input(shape = c(4))
2 prediction <- inputs %>%
3     layer_dense(units = 5, activation = "relu") %>%
4     layer_dense(units = 1, activation = NULL)
5
6 model <- keras_model(inputs = inputs, outputs =
      prediction)
7
8 model %>% compile(optimizer, loss)
9 model %>% fit(x_train, y_train, batch_size, epochs)
```

```
> summary(model)
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 4)]               0
_________________________________________________________________
dense (Dense)                (None, 5)                 25
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 6
=================================================================
Total params: 31
Trainable params: 31
Non-trainable params: 0
_________________________________________________________________
```

# Rekurrente Neuronale Netze

# Motivation

MLPs eignen sich gut zur Modellierung von vielen Problemstellungen. Bei manchen stoßen sie aber an ihre Grenzen. Beispiele dafür sind:

- Bildverarbeitung/-analyse
  $\rightarrow$ Convolutional Neural Network (CNN)
- Modellierung von Sequenzen mit zeitlicher Abhängigkeit
  $\rightarrow$ Recurrent Neural Networks (RNN)

# Simples Rekurrentes Netz

Der Einfachheit halber betrachten wir zunächst ein Netzwerk mit einem Hidden Layer. Gegeben ist ein Input an $T$ Zeitschritten $(x_1, \ldots, x_T)$, wobei $x_i \in \mathbb{R}^d$ für $i = 1, \ldots, T$. Die Idee des rekurrenten Netzes basiert auf dem dynamischen System

$$h_t = f(x_t, h_{t-1}; \theta),$$

dabei wird $h_t$ als Hidden State bezeichnet und $\theta$ sind die Parameter. Für $h_0$ wird ein Wert vorinitialisiert, zum Beispiel 0.

Die obige Gleichung beschreibt die folgende Netzwerkarchitektur mit $n$ Neuronen in der rekurrenten Schicht und einem Output für jeden Zeitschritt der Dimension $m$.

$$\begin{aligned} a_t &= b + Wh_{t-1} + Ux_t, \\ h_t &= \phi(a_t), \\ o_t &= c + Vh_t, \\ \hat{y} &= \gamma(o_t), \end{aligned} \tag{2.1}$$

mit den Bias $b \in \mathbb{R}^n$, $c \in \mathbb{R}^m$, den Gewichtsmatrizen $W \in \mathbb{R}^{n \times n}$, $U \in \mathbb{R}^{n \times d}$ und $V \in \mathbb{R}^{m \times n}$ und beliebigen Aktivierungsfunktionen $\phi$ und $\gamma$.

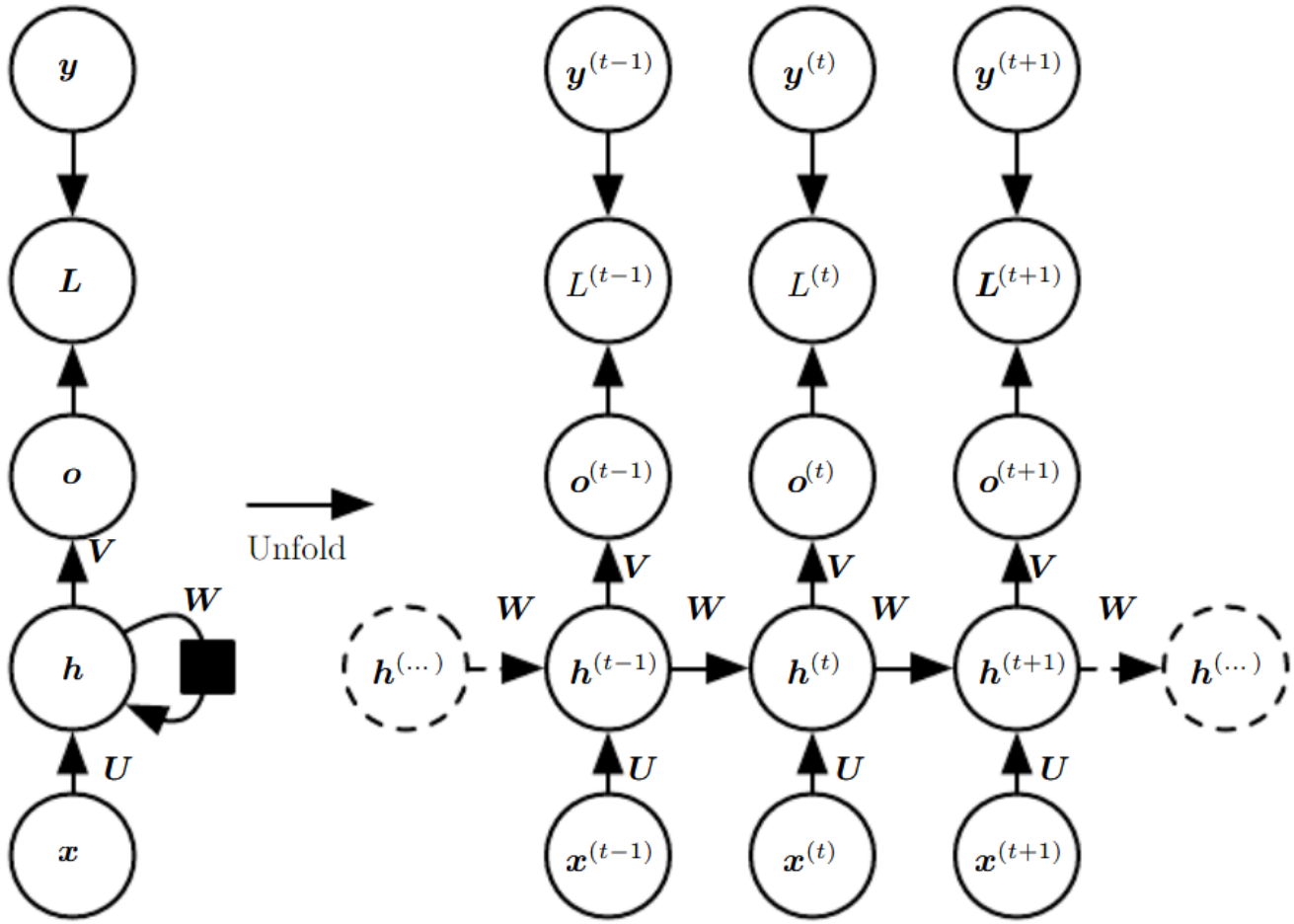



Abbildung: Das Netzwerk aus Gleichung (2.1) und seine zeitlich entfaltete Darstellung. Das schwarze Quadrat steht für die Verzögerung von einem Zeitschritt.

Oftmals ist nur ein Output nach dem letzten Zeitschritt $T$ gesucht. Die Gleichung (2.1) reduzieren sich zu

$$\begin{aligned} a_t &= b + Wh_{t-1} + Ux_t, \\ h_t &= \phi(a_t), \end{aligned} \tag{2.2}$$

für $t = 1, \ldots, T$ und

$$\begin{aligned} o &= c + Vh_T, \\ \hat{y} &= \gamma(o). \end{aligned}$$

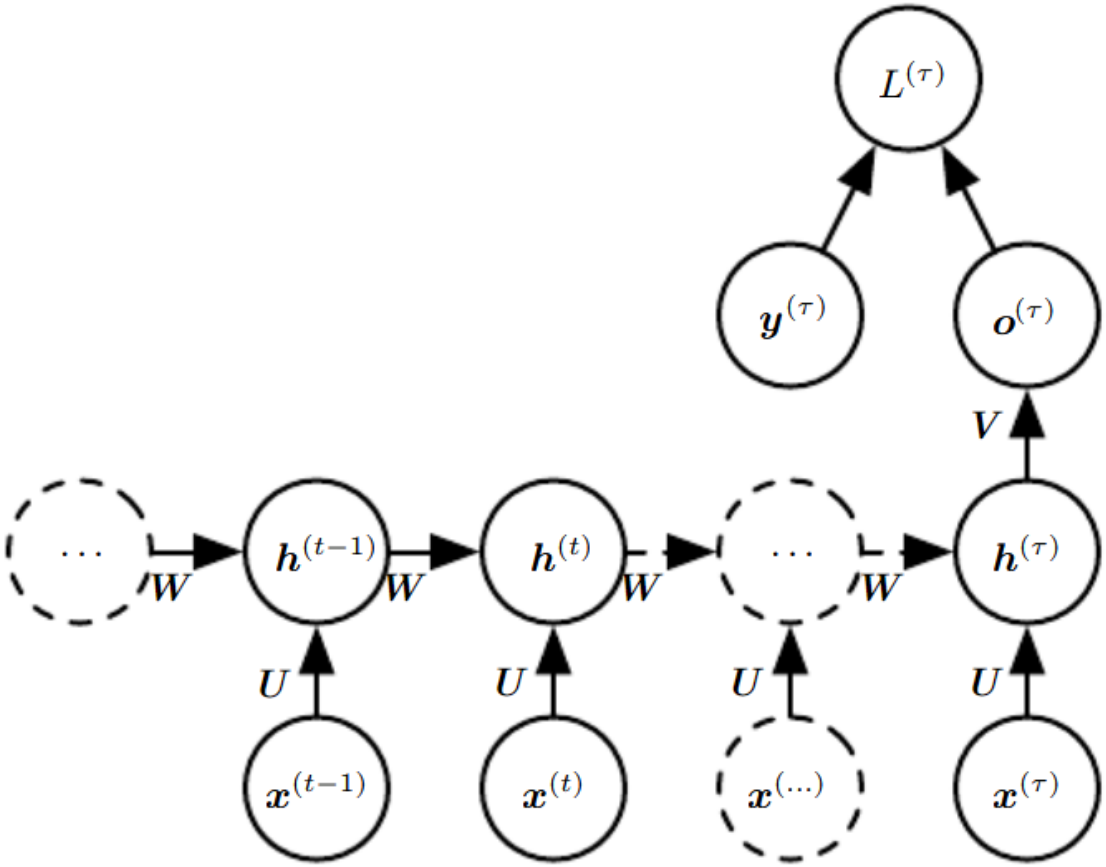



Abbildung: Das Netzwerk aus (2.2) in zeitlich entfalteter Darstellung.

Bisher nur ein Netzwerk mit einem Hidden Layer. Mögliche Varianten für ein Netzwerk mit zwei Hidden Layern.

- *1. Variante:* Nur Rückkopplungen im Hidden Layer selbst

$$h_t^{(1)} = f^{(1)}(x_t, h_{t-1}^{(1)}; \theta^{(1)}),$$
$$h_t^{(2)} = f^{(2)}(h_t^{(1)}, h_{t-1}^{(2)}; \theta^{(2)}).$$

- *2. Variante:* Rückkopplungen zur ersten Schicht

$$h_t^{(1)} = f^{(1)}(x_t, h_{t-1}^{(1)}, h_{t-1}^{(2)}; \theta^{(1)}),$$
$$h_t^{(2)} = f^{(2)}(h_t^{(1)}, h_{t-1}^{(2)}; \theta^{(2)}).$$

- *3. Variante:* Zusätzliche Skip Connection vom Input zur zweiten rekurrenten Schicht

$$h_t^{(1)} = f^{(1)}(x_t, h_{t-1}^{(1)}, h_{t-1}^{(2)}; \theta^{(1)}),$$
$$h_t^{(2)} = f^{(2)}(x_t, h_t^{(1)}, h_{t-1}^{(2)}; \theta^{(2)}).$$

# Backpropagation Through Time

Wie wird ein solches Netz trainiert?
$\rightarrow$ Backpropagation Through Time (BPTT)

- Insgesamte Kosten: $C_{total} = \Sigma_t C_t$.
- Nutze den zeitlich entfalteten Graph, um die Ableitung von $C_t$ nach den Gewichten mittels Backpropagation zu bestimmen.
- Summiere Gradienten der Gewichte.

# Long Short-Term Memory

- Auch bei RNN tritt Vanishing/Exploding Gradient Problem auf
- Problem: Weit zurückliegende Informationen werden bei gewöhnlichen RNNs „vergessen“(„Short-Term Memory“)
- Lösung: Gated Architekturen
- Idee: Kontrolliere mithilfe von Gates den Informationsfluss
- LSTM eine der meistgenutzten Architekturen
- Anwendungen: Spracherkennung, Sprachübersetzung, weitere sequentielle Aufgabenstellungen

# LSTM-Modell

- Kernkomponente der LSTM-Einheit ist der Cell State $c_t$, der als Gedächtnis fungiert, indem er relevante Informationen durch die Zeitschritte transportiert $\rightarrow$ Long-Term Memory möglich
- Welche Informationen genau als relevant gelten, entscheiden die Gates, die lernen welcher Teil vergessen bzw. hinzugefügt werden sollte

# Aufbau eines LSTM-Modells

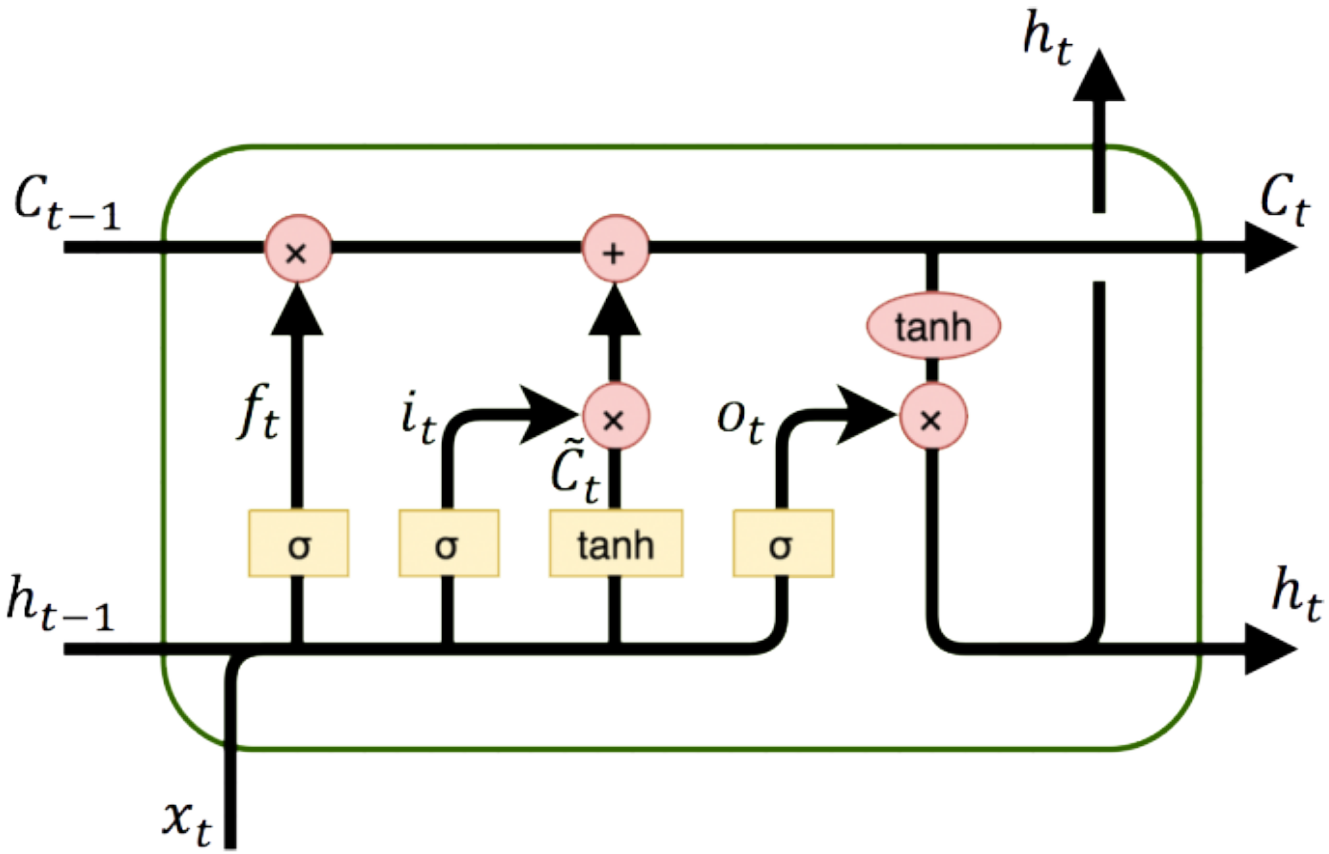



Abbildung: LSTM-Einheit mit den verschiedenen Gates (s. [O]).

# LSTM-Modell

Betrachte einen Hidden LSTM-Layer mit $n$ Neuronen.
**Forget Gate**:

$$f_t = \sigma\left(U_f x_t + W_f h_{t-1} + b_f\right) \in (0,1)^n$$

$(f_t)_k \approx 0 \rightarrow$ Vergiss Informationen zu $(c_{t-1})_k$
$(f_t)_k \approx 1 \rightarrow$ Behalte Informationen zu $(c_{t-1})_k$
**Input Gate**:

$$i_t = \sigma\left(U_i x_t + W_i h_{t-1} + b_i\right) \in (0,1)^n$$

Cell State zum Zeitpunkt $t$:

$$c_t = f_t \circ c_{t-1} + i_t \circ \underbrace{\tanh\left(U_c x_t + W_c h_{t-1} + b_c\right)}_{:=\tilde{c}_t} \in \mathbb{R}^n.$$

# Cell State

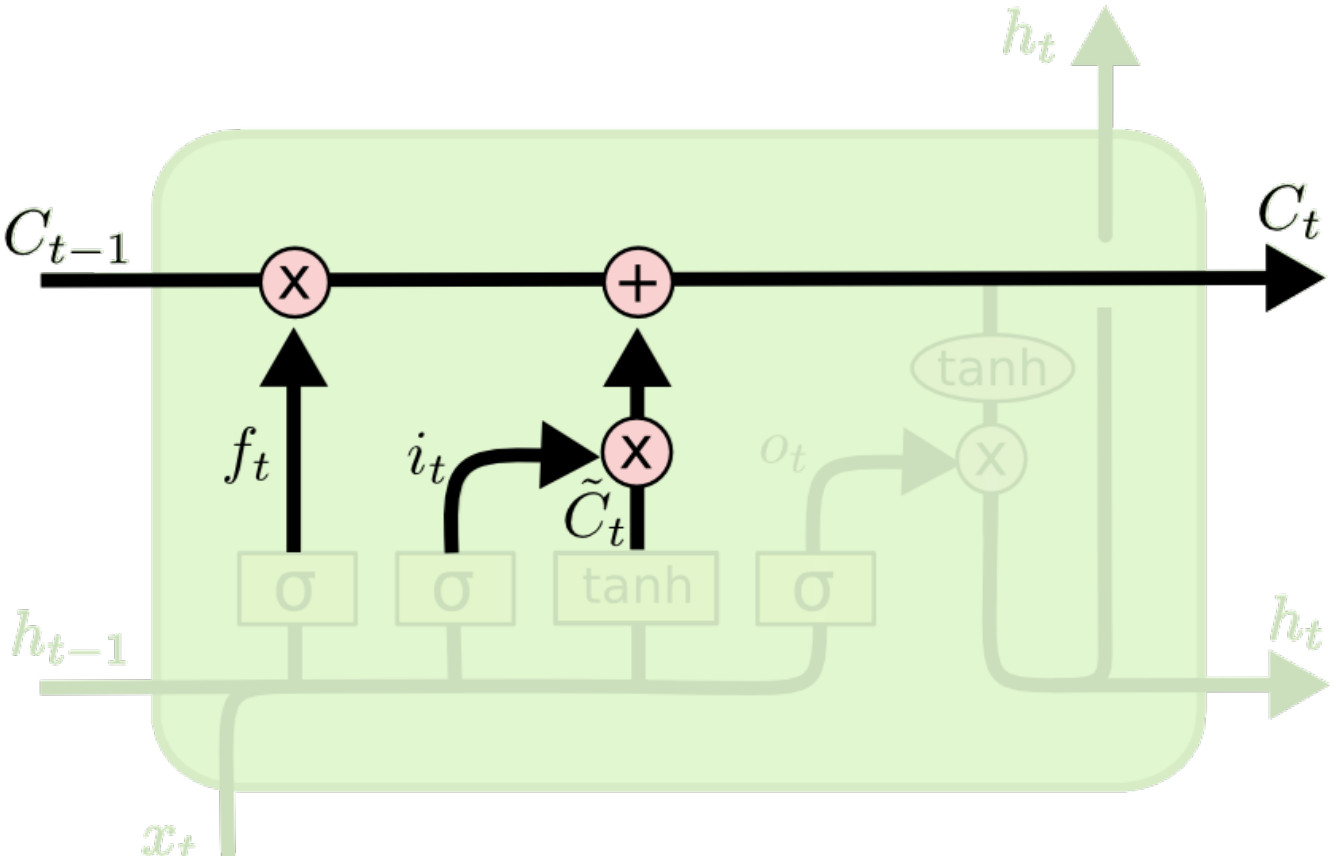



Abbildung: Update des Cell States zum Zeitpunkt $t$.

# Hidden State

Neuer Hidden State:

$$h_t = o_t \circ \phi(c_t) \in \mathbb{R}^n$$

mit beliebiger Aktivierungsfunktion $\phi$ und dem **Output Gate**

$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o) \in (0,1)^n$$

# Hidden State

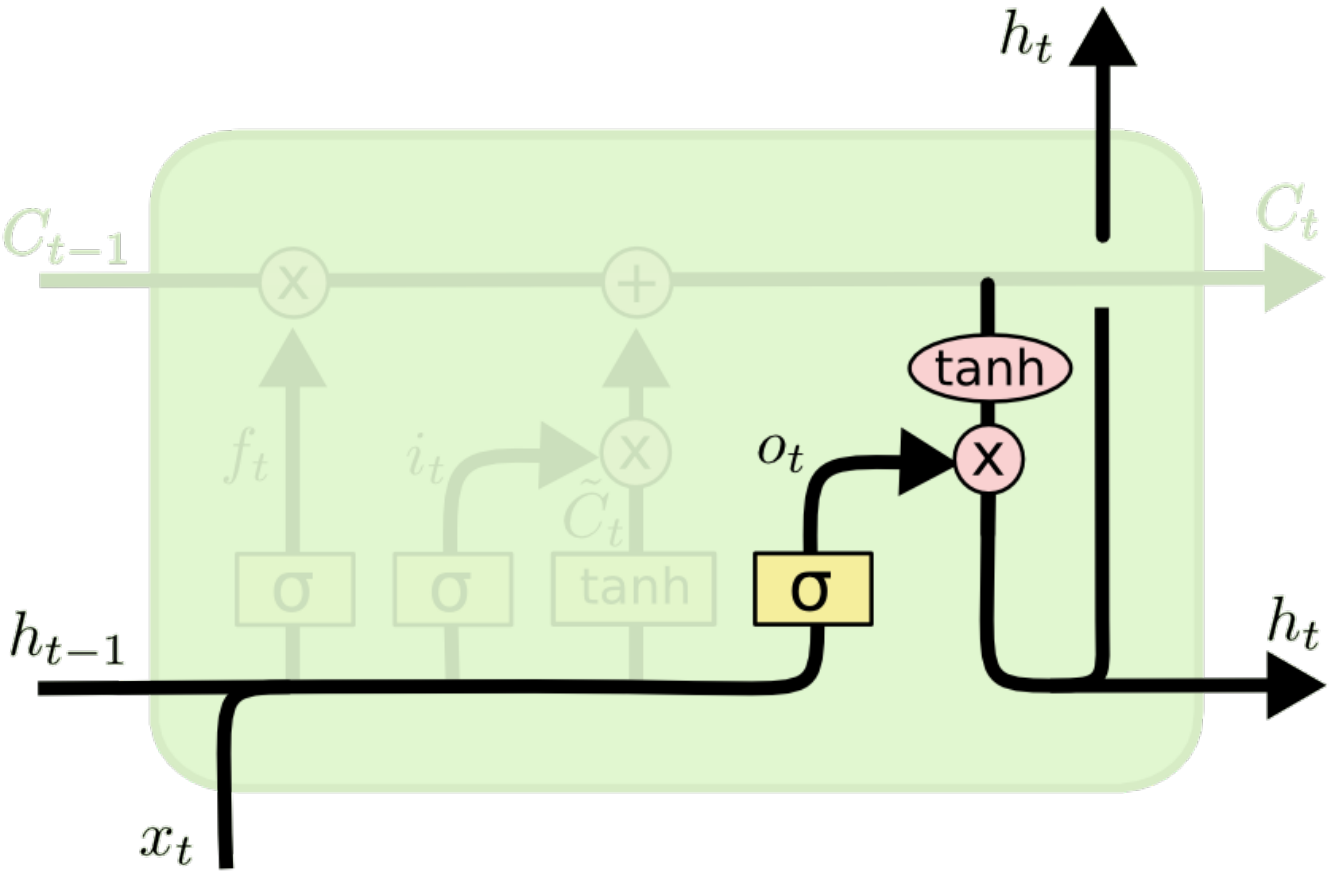



Abbildung: Update des Hidden States zum Zeitpunkt $t$.

# Überblick

Insgesamt:

$$
\begin{aligned}
f_t &= \sigma\left(U_f x_t + W_f h_{t-1} + b_f\right) && \in (0,1)^n \\
i_t &= \sigma\left(U_i x_t + W_i h_{t-1} + b_i\right) && \in (0,1)^n \\
o_t &= \sigma\left(U_o x_t + W_o h_{t-1} + b_o\right) && \in (0,1)^n \\
\tilde{c}_t &= \tanh\left(U_c x_t + W_c h_{t-1} + b_c\right) && \in (-1,1)^n \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t && \in \mathbb{R}^n \\
h_t &= o_t \circ \phi\left(c_t\right) && \in \mathbb{R}^n
\end{aligned}
$$

wobei $U_k \in \mathbb{R}^{n \times d}$, $W_k \in \mathbb{R}^{n \times n}$ und $b_k \in \mathbb{R}^n (k \in \{f, i, o, c\})$ und somit insgesamt $4\left(nd + n^2 + n\right)$ Parameter

# Gated Recurrent Unit

- Vereinfachtes Modell, ähnlich effizient wie LSTM
- Bei GRU ersetzt ein *Update Gate* das Forget und Input Gate aus dem LSTM-Modell
- Cell State fällt ganz weg, nur noch Hidden State

# Aufbau eines GRU-Modells

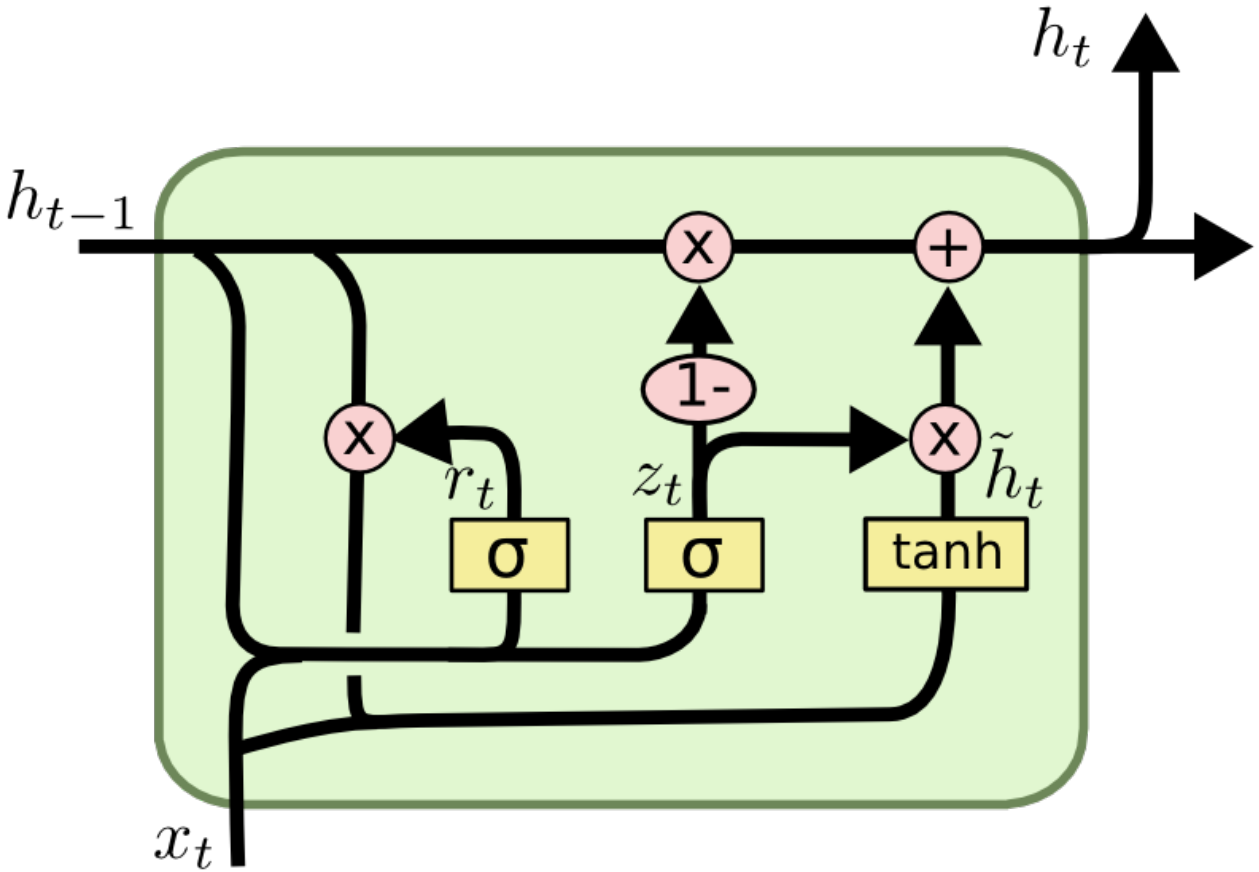



Abbildung: GRU-Einheit mit den verschiedenen Gates.

# GRU

Update Gate $z_t$ und Reset Gate $r_t$:

$$
\begin{aligned}
z_t &= \sigma\left(U_z x_t + W_z h_{t-1} + b_z\right) && \in (0,1)^n \\
r_t &= \sigma\left(U_r x_t + W_r h_{t-1} + b_r\right) && \in (0,1)^n \\
\tilde{h}_t &= \tanh\left(U_h x_t + W_h(r_t \circ h_{t-1}) + b_h\right) && \in (-1,1)^n
\end{aligned}
$$

Neuer Hidden State:

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t \in \mathbb{R}^n$$

Insgesamt nur noch $3\left(nd + n^2 + n\right)$ Parameter.

# Implementierung in R mit Keras

Hier: Inputs $x_t \in \mathbb{R}^5$ mit $t \in \{1, \ldots, 10\}$ sowie zwei LSTM-Layer mit $n_1 = 20$ und $n_2 = 10$ Neuronen.

```
1 library(keras)
2 inputs <- layer_input(shape = c(10, 5))
3 hidden_1 <- layer_lstm(units = 20, return_sequences =
      TRUE)(inputs)
4 hidden_2 <- layer_lstm(units = 10)(hidden_1)
5 output <- layer_dense(units = 1)(hidden_2)
6
7 model <- keras_model(inputs = inputs, outputs = output)
```

# Implementierung in R mit Keras

```
> summary(model)
Model: "model"
```

$4(5 \cdot 20 + 20^2 + 20)$

| Layer (type) | Output Shape | Param # |
|---|---|---|
| inputs (InputLayer) | [(None, 10, 5)] | 0 |
| hidden_1 (LSTM) | (None, 10, 20) | 2080 |
| hidden_2 (LSTM) | (None, 10) | 1240 |
| output (Dense) | (None, 1) | 11 |

```
Total params: 3,331
Trainable params: 3,331
Non-trainable params: 0
```

$4(20 \cdot 10 + 10^2 + 10)$

# Fazit & Ausblick

# Fazit & Ausblick

- Deep Feedforward Netze vielseitig anwendbar
- Nicht optimal bei Zeitreihen-Problemen (z.B. Sterblichkeitsmodellierung)
- Dafür eignen sich RNN
- Aber: Immer noch Vanishing/Exploding Gradient Problem
- Lösung: Gated Architekturen wie LSTM, GRU, etc.
- Allerdings: Auch Gated Architekturen stoßen bei sehr weit zurückliegenden Informationen an ihre Grenzen

# Literatur

# Literatur


Ian Goodfellow and Yoshua Bengio and Aaron Courville, 2016, *Deep Learning*, MIT Press, http://www.deeplearningbook.org


Catherine F. Higham, Desmond J. Higham, 2018, *Deep Learning: An Introduction for Applied Mathematicians*, https://arxiv.org/pdf/1801.05894.pdf


Ronald Richman, Mario V. Wüthrich, 2019, *Lee and Carter go Machine Learning: Recurrent Neural Networks*, Swiss Association of Actuaries SAV


Chris Olah, August, 27, 2015, *Understanding LSTM Networks*, http://colah.github.io/posts/2015-08-Understanding-LSTMs/