Deep Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-08 Dozent: Karsten Flügge

Themen des Tages

Tag 12

Recurrent Neural Networks (RNN) Teil 2:
Exploding und Vanishing Gradient Probleme #2
LSTM (Long Short-Term Memory)
GRU (Gated Recurrent Unit)
Deep RNN, Deep LSTM

Recurrent Neural Networks (RNN) II

nächster output vom RNN:

$$h_{t+1} = \sigma(W \cdot x + W_h \cdot h_t + b)$$

W Gewichtsmatrix für Input x
W_h Gewichtsmatrix für Inneren Zustand 'hidden' h_t
b bias

=> Mathematik kann übersichtlicher sein als Code

Recurrent Neural Networks (RNN) II

Achtung: "number of layers" ≠ sequence length

$$\mathbf{h_{t+1} = \sigma(W \cdot x + W_h \cdot h_t + b) \Rightarrow \text{into next layer:}}$$

$$h2_{t+1} = \sigma(W2 \cdot \mathbf{h1_t} + W2_h \cdot h2_t + b2)$$

$$h3_{t+1} = \sigma(W3 \cdot \mathbf{h2_t} + W3_h \cdot h3_t + b3)$$

Die RNN Schichten werden also nicht nur vertikal sondern auch horizontal gestapelt. Die Ausgabe ist dann das was rechts oben herauskommt.

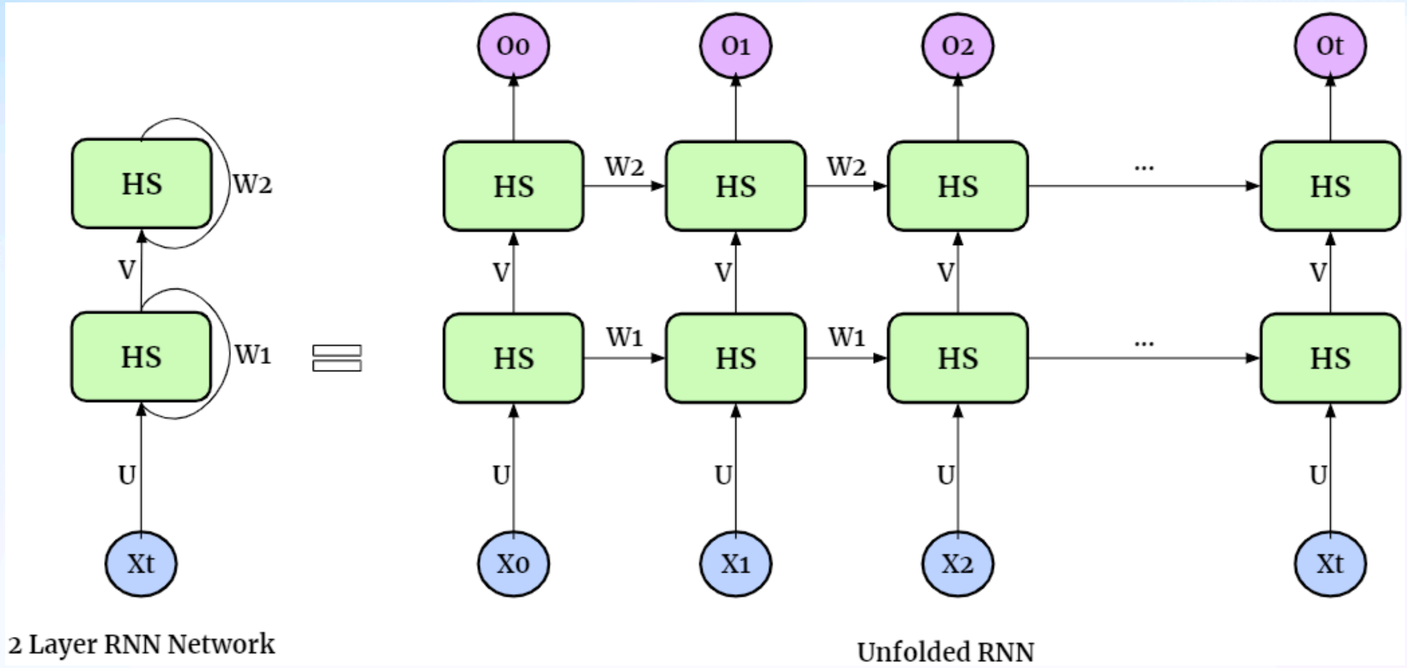
Recurrent Neural Networks (RNN) II

Achtung: "number of layers" ≠ sequence length

$$\mathbf{h}_{t+1} = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{W}_h \cdot \mathbf{h}_t + \mathbf{b}) \Rightarrow \text{into next layer:}$$

$$h2_{t+1} = \sigma(W2 \cdot \mathbf{h1}_t + W2_h \cdot h2_t + b2) \quad \# \text{ memory 2}$$

$$h3_{t+1} = \sigma(W3 \cdot \mathbf{h2}_t + W3_h \cdot h3_t + b3)$$



2 Layer RNN Network

Unfolded RNN

Exploding und Vanishing Gradients in RNN

Wir erinnern uns an Exploding und Vanishing Gradient Probleme. Diese treten bei RNNs besonders auf. Lösung:

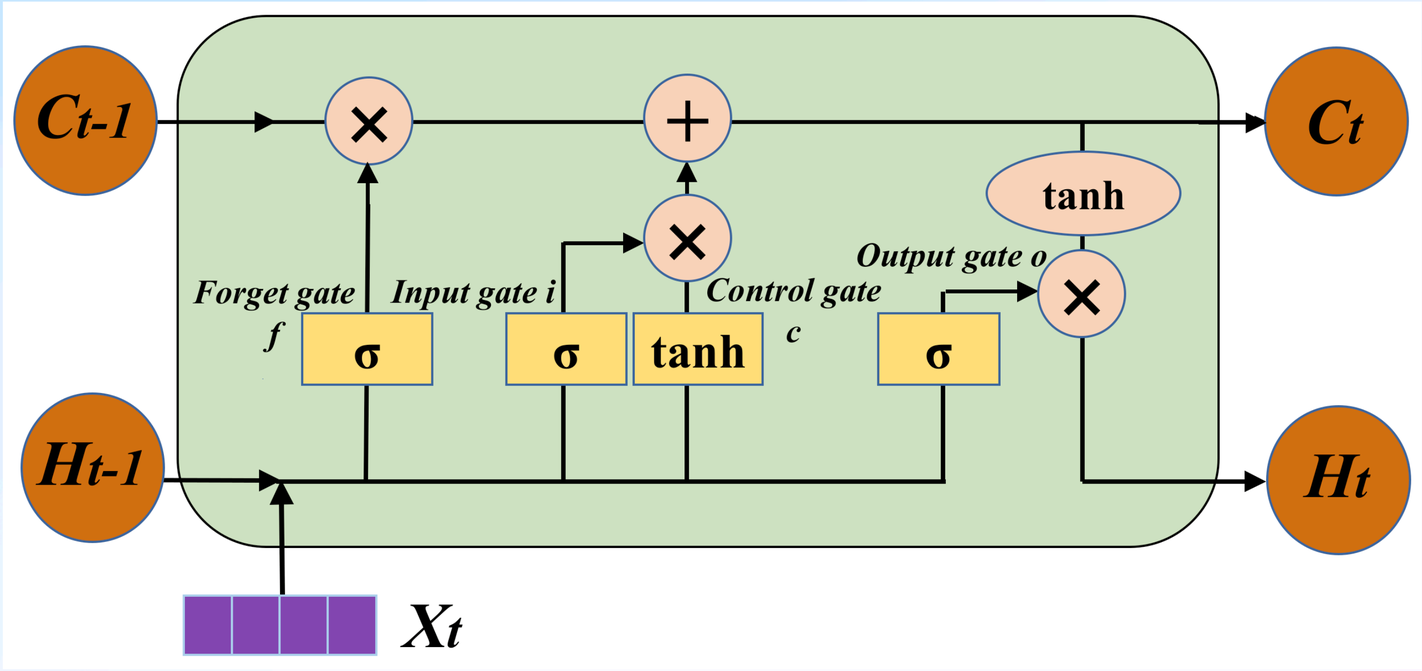
Modifizierte RNNs:

LSTM (Long Short-Term Memory)

Ein LSTM besteht aus sogenannten LSTM-Zellen, die über die Zeit hinweg verbunden sind.

Jede **LSTM-Zelle** hat drei Hauptkomponenten:

- **Vergessen-Gate (Forget Gate)**: Bestimmt, welche **Informationen** aus der Zelle **entfernt** werden sollen.
- **Eingangs-Gate (Input Gate)**: Entscheidet, welche neuen **Informationen hinzugefügt** werden.
- **Ausgangs-Gate (Output Gate)**: Bestimmt, welche **Informationen** aus der Zelle **weitergegeben** werden.

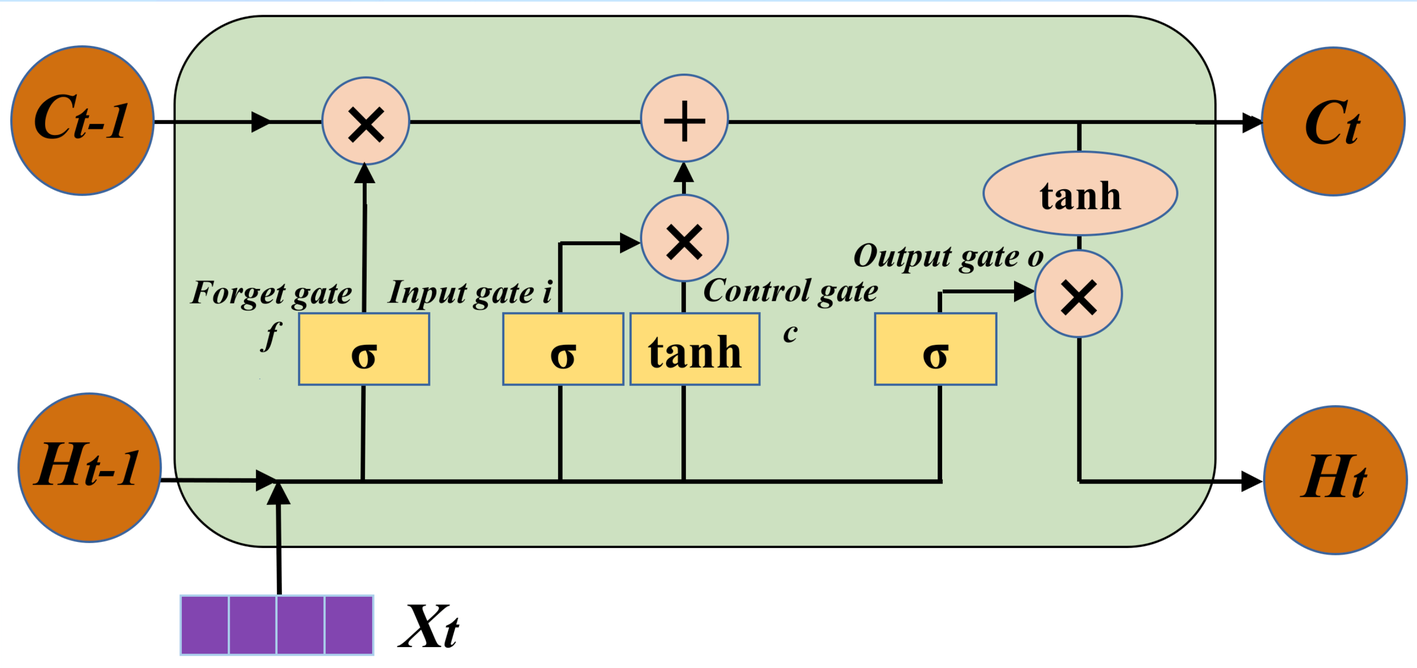


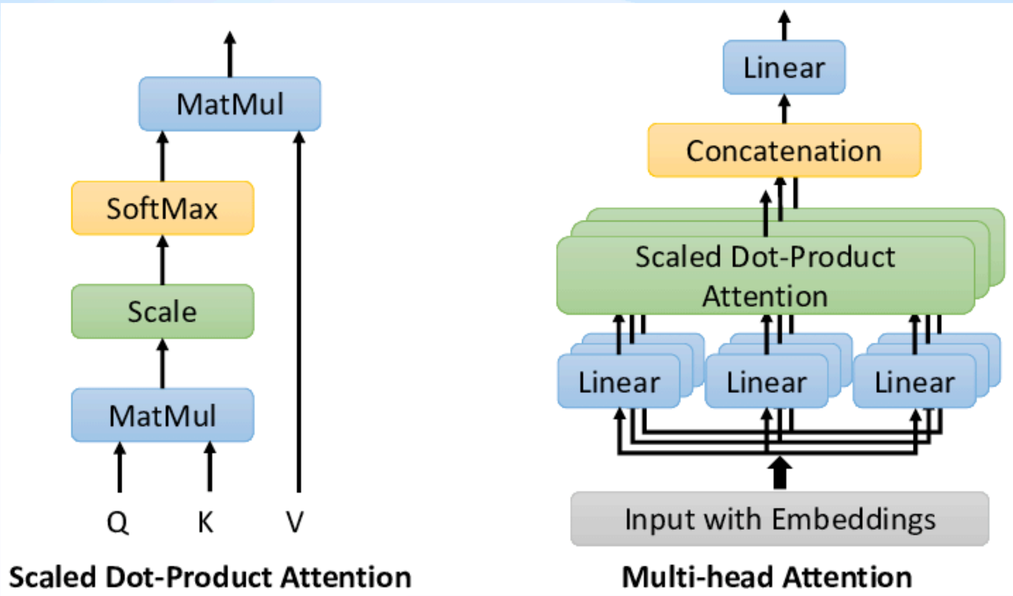
LSTM vs Attention

- Entsprechen die Berechnungen im LSTM UNGEFÄHR den Attentions?

- **Vergessen-Gate (Forget Gate)**: Bestimmt, welche **Informationen** aus der Zelle **entfernt** werden sollen.
- **Eingangs-Gate (Input Gate)**: Entscheidet, welche neuen **Informationen hinzugefügt** werden.
- **Ausgangs-Gate (Output Gate)**: Bestimmt, welche **Informationen** aus der Zelle **weitergegeben** werden.

LSTM: sigmoid 'conditions/gates'
Attention: sigmoid + softmax 'condition/gate'



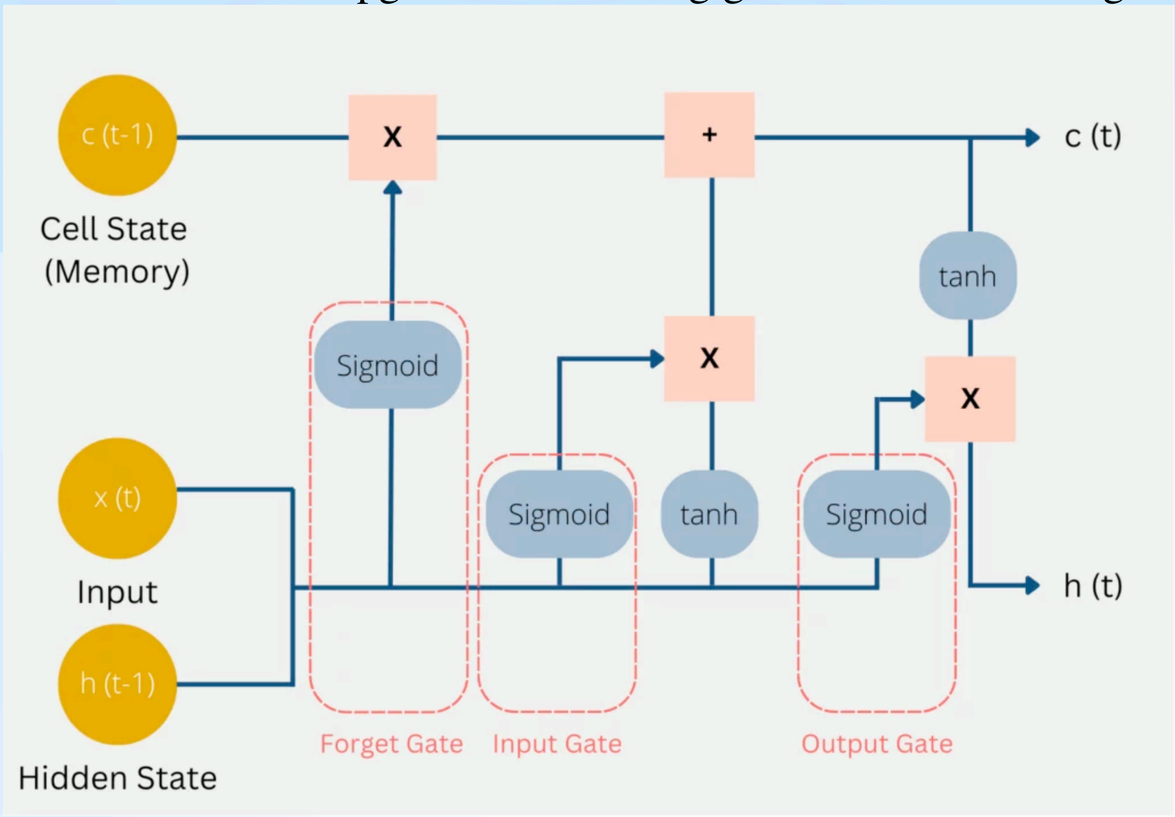


LSTM (Long Short-Term Memory)

Anders als bei dem einfachen RNN werden beim LSTM **zwei zeitabhängige Zustände** gespeichert:

Cell State entspricht so etwas wie **Langzeitgedächtnis**
Hidden State entspricht etwa **Kurzzeit**gedächtnis und der **aktuellen Aktivierung** (⚠️ anders als normale Konvention)

In jedem Zeitschritt werden die Werte upgedated in Abhängigkeit von den vorherigen Werten und dem aktuellen Input:



LSTM (Long Short-Term Memory)

Idee als Rechenschema:

für jeden Zeitschritt **t**:

Forget-Gate $\mathbf{f}_t = \sigma(Wf * [h_{t-1}, x_t] + bf)$

Input-Gate $\mathbf{i}_t = \sigma(Wi * [h_{t-1}, x_t] + bi)$

Candidate-Zustand $C'_t = \tanh(Wc * [h_{t-1}, x_t] + bc)$

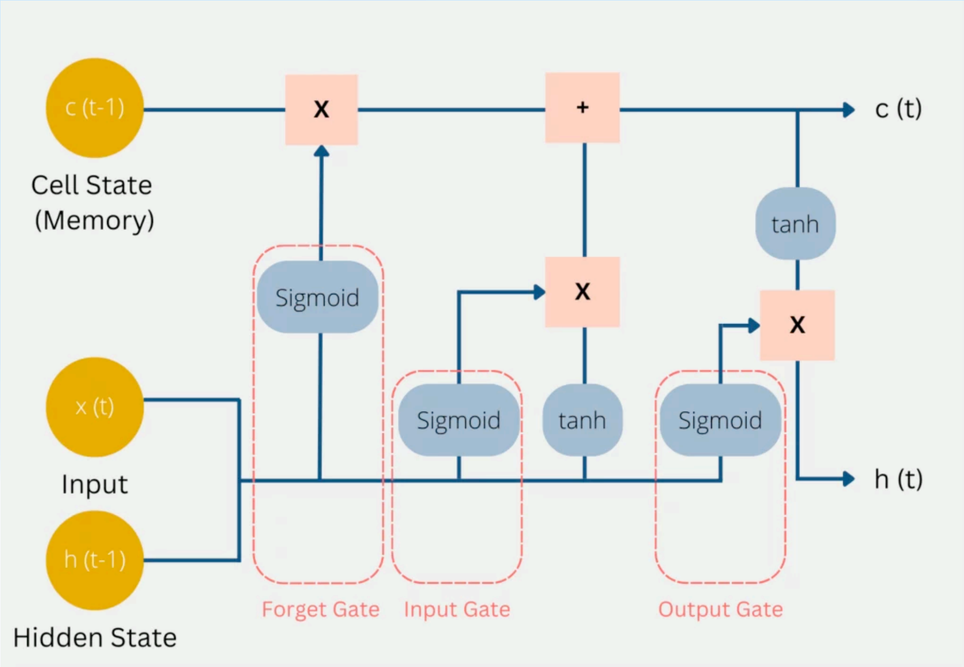
Cellzustand $\mathbf{C}_t = f_t * C_{t-1} + i_t * C't$

Output-Gate $o_t = \sigma(Wo * [h_{t-1}, x_t] + bo)$

Cell-Ausgabe $\mathbf{h}_t = o_t * \tanh(C_t)$

zum Vergleich einfacheres RNN Schema:

$$\mathbf{h}_{t+1} = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{W}_h \cdot \mathbf{h}_t + \mathbf{b})$$



Deep RNN, Deep LSTM

Deep RNN, Deep LSTM

wie Lego-Blöcke lassen sich RNN **Schichten** / LSTM **Zellen** hintereinander schalten, indem man schlicht die Ausgabe einer Zelle als Eingabe für die nächste Zelle verwendet.

Natürlich darf man die Ausgaben zwischendurch auch mit anderen Netzwerkkomponenten bearbeiten, besonders linear / fully connected layers. Wir haben auch gesehen wie man die Ausgaben splitten und später wieder re-kombinieren kann. Der Kreativität sind quasi keine Grenzen gesetzt? Doch: normale RNNs leiden unter Vanishing Gradient Problemen, Aber: dank LSTM und anderen Techniken welche wir gesehen haben lassen sich diese Grenzen doch meist überwinden.

Tatsächlich gab es und gibt es Ansätze, statt sich ad-hoc Netzwerk Architekturen auszudenken, diese lieber systematisch von Computern selber erforschen zu lassen.

Deep RNN, Deep LSTM

- ResNet: Often used with depths of 50, 101, or 152 layers.
- DenseNet: Commonly used with 121, 169, or 201 layers.
- EfficientNet: Variants range from 25 to 66 layers.
- Transformers (e.g., BERT): Often have depths of 12 to 48 layers or more.
  - **GPT-2**: 48 layers.
  - **GPT-3**: 96 layers.
  - **GPT-4**: wahrscheinlich 100te Schichten (Details proprietär)

LSTM already outdated?

Apropos Kreativität:

LSTM Rechenschema ist **kompliziert**, was ist tatsächlich die **Essenz** dieser Zellen?

Memory, in Abhängigkeit vom vorherigen Zustand um vom aktuellen Input

Warum darf man diesen Zustand nicht einfach irgendwo anders im Speicher oder auf der Festplatte abspeichern?

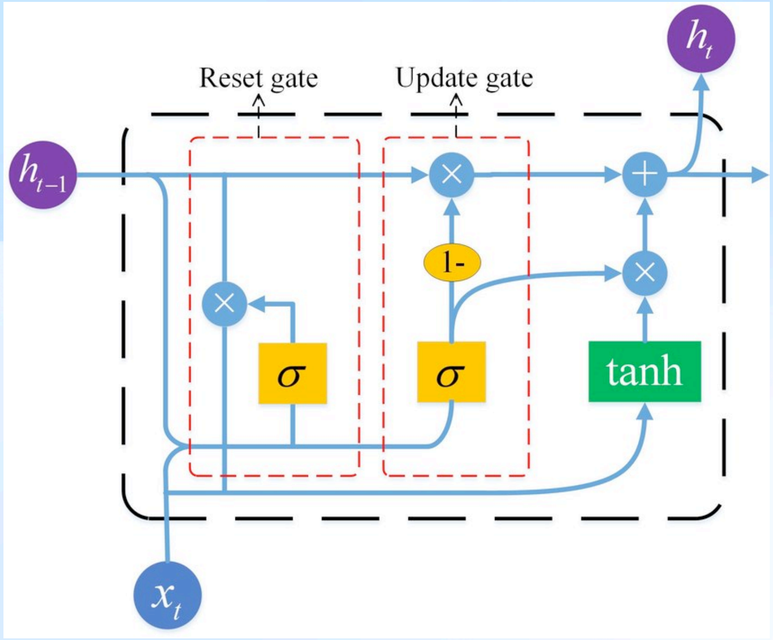
Um trainierbar zu sein muss jede Aktion in einem neuronalen Netz ableitbar sein oder zumindest brauchen wir so etwas für einen Pseudo **Gradienten**. Es gibt Forschungsansätze solche **allgemeinen Schreib und Lesevorgänge** als ableitbare Funktion zu approximieren, zunächst gucken wir aber auf eine Vereinfachung des Rechenschemas:

GRU (Gated Recurrent Unit)

GRU (Gated Recurrent Unit): **Vereinfachung** von LSM und Beibehaltung der selben Grundfunktion:
Zumindest für tiefe Netze hat sich herausgestellt dass GRUs keinerlei Genauigkeitsverluste gegenüber LSTMs haben.

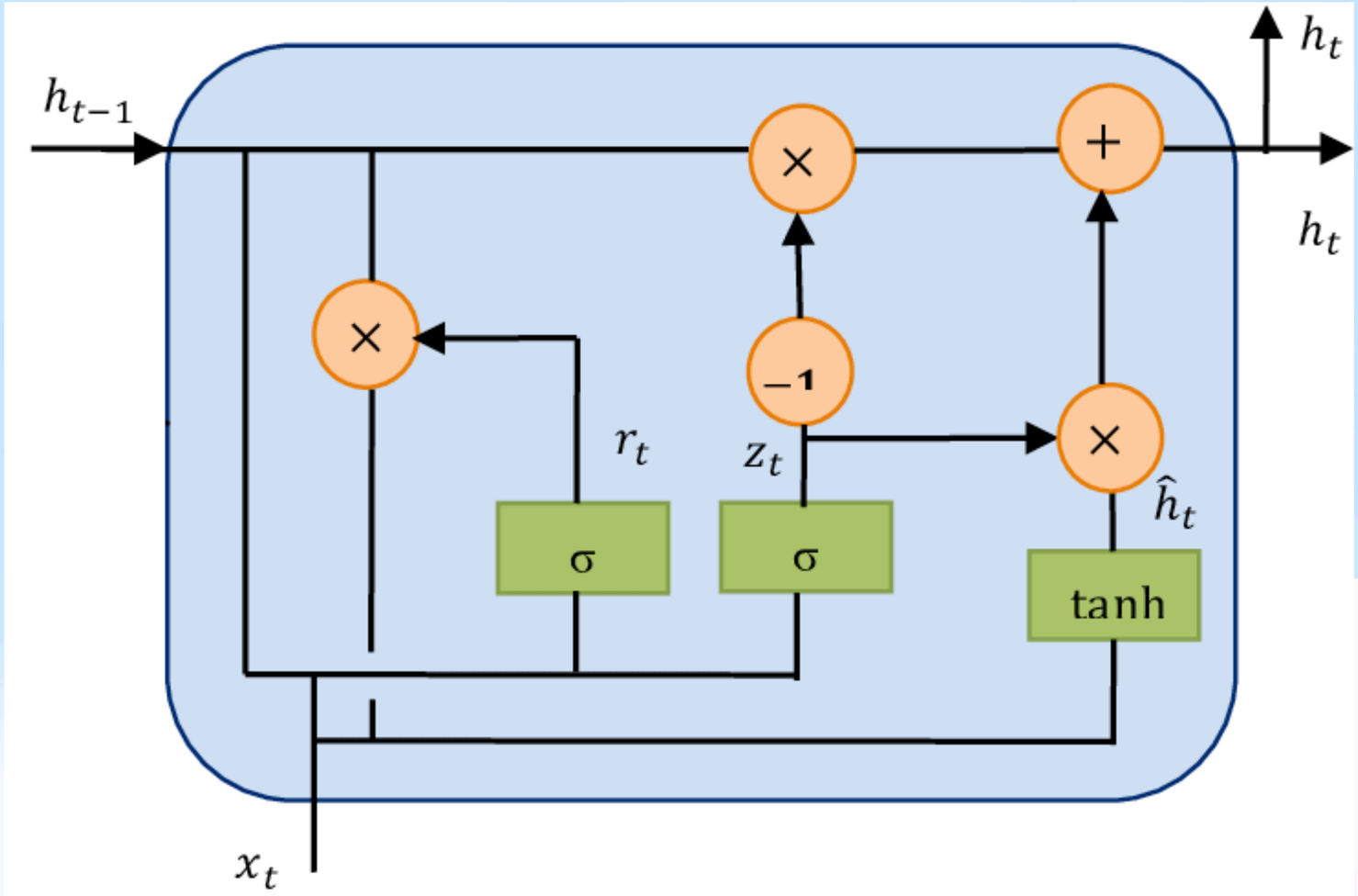
nur zwei Gates

- **Reset Gate**: Entscheidet, wie stark der vorherige Zellzustand **vergessen** wird.
- **Update Gate**: Kontrolliert den Anteil der alten Informationen, die **beibehalten** werden, und den Anteil der neuen Informationen, die **hinzugefügt** werden.



Im Grunde sind wir also fast zurück beim RNN, an der Ausgabe Schnittstelle ist es sogar vereinfacht weil Hidden State und Aktivierung zusammenfallen.
Nur intern sind GRUs etwas inspirierter als klassische RNNs:

GRU (Gated Recurrent Unit)



Für jeden Zeitschritt t:

Reset-Gate:

$\mathbf{r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)}$

W_r: Gewichtsmatrix für das Reset-Gate
b_r: Bias für das Reset-Gate
h_{t-1}: Hidden-State vom vorherigen Zeitschritt
x_t: Eingabe im aktuellen Zeitschritt
$[h_{t-1}, x_t]$ concat von hidden und input
σ: **Sigmoid**-Funktion

Update-Gate:

$\mathbf{z_t = \sigma(Wz \cdot [h_{t-1}, x_t] + bz)}$

selbe Formel nur andere Verwendung
Wz: Gewichtsmatrix für das Update-Gate
bz: Bias für das Update-Gate

Kandidaten-Aktivierung "new hidden"

$\mathbf{\tilde{h}_t = \tanh(W_h \cdot [r_t * h_{t-1}, x_t] + b_h)}$

W_h: Gewichtsmatrix für die Kandidaten-Aktivierung
b_h: Bias für die Kandidaten-Aktivierung
$\mathbf{r_t * h_{t-1}}$: Elementweise Multiplikation des Reset-Gates mit dem Hidden-State vom vorherigen Zeitschritt (**r=0** => old hidden verschwindet **r=1** keep it)

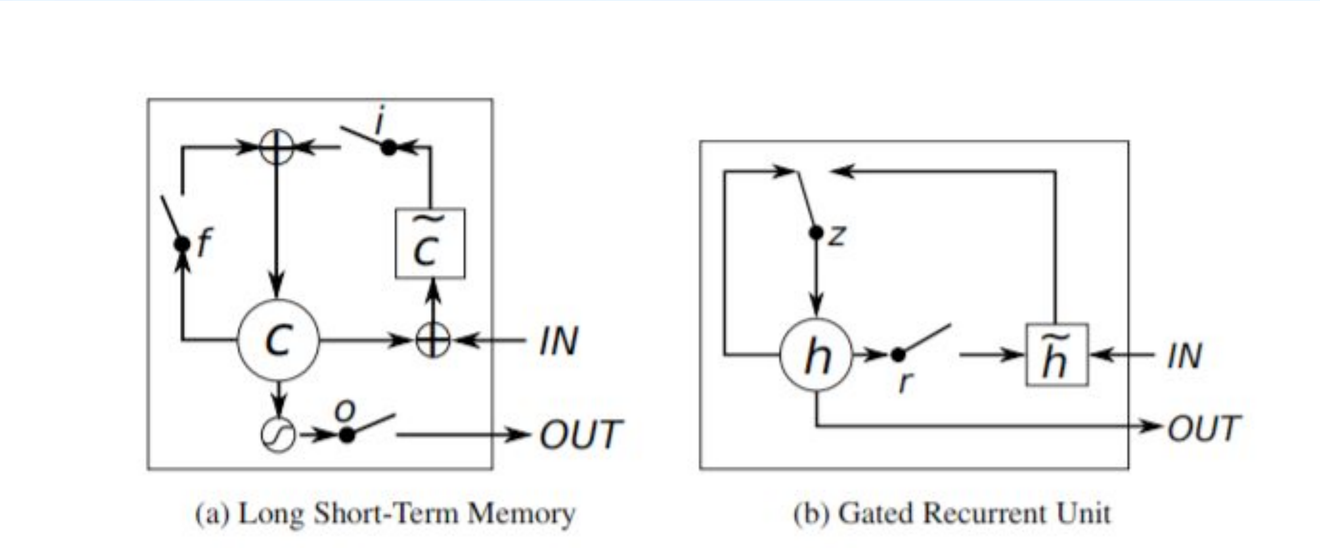
Aktualisierter **Hidden-State:**

$\mathbf{h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t}$ in [-1,1]

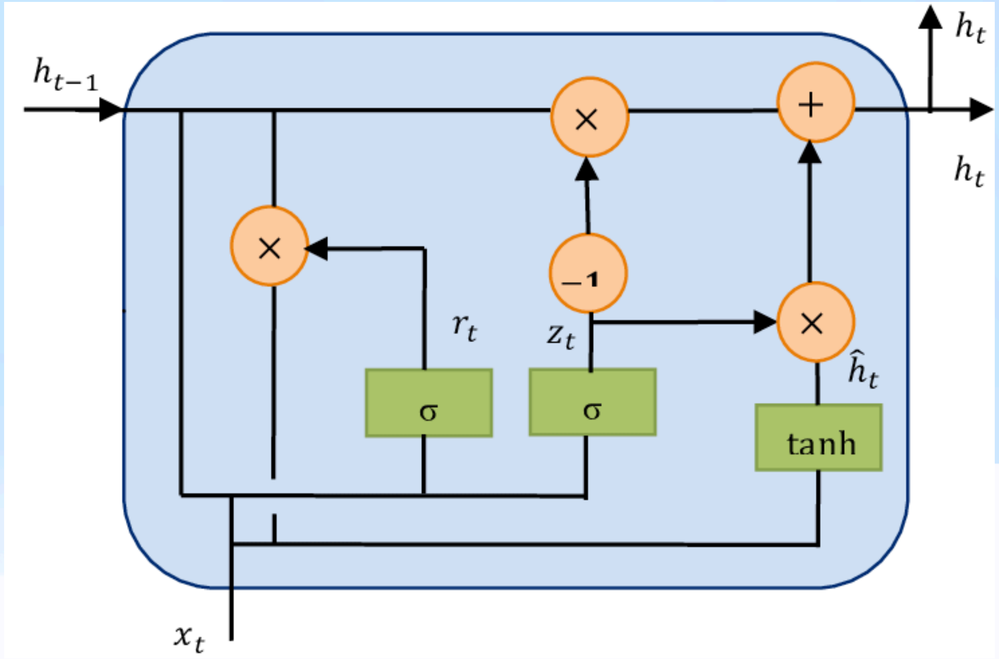
$(1 - z_t) * h_{t-1}$: Beibehaltung eines Teils des vorherigen Hidden-States
$z_t * \tilde{h}_t$: Hinzufügen der **neuen** Informationen aus der Kandidaten-Aktivierung

GRU (Gated Recurrent Unit)

GRU gates ≈ logic gates



(a) Long Short-Term Memory (b) Gated Recurrent Unit



Shakespeare RNN

Shakespeare RNN: Lasse Netzwerk schreiben wie Shakespeare

One hot encoding für Text: ASCII Buchstabe 'a' => 97 'D' => 68 …

Jeder Buchstabe bekommt eine 1 im Vector:
(0,0,0,0,0,0,0,0,0,0, … 1, 0,0,0,0,0,0,0, … 0) = 'a' wenn 1 an 97ster Stelle

Buchstabe zu Vector
Text zu Tensor: Stapel von Buchstaben Vektoren

Damit kann unser Netz mit Texten rechnen.
Output Vorhersage = nächster Buchstabe
Zeitreihen **langfristige Abhängigkeiten**
QUEEN ELIZABET_ sage 'H' voraus aufgrund *aller* vorheriger Zeichen

Char RNNs

The Unreasonable Effectiveness of Recurrent Neural Networks

Andrej Karpathy blog May 21, 2015

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

pure python implementation: **char_rnn_pure.py** !

pytorch implementation: **char_rnn.py**
https://github.com/spro/char-rnn.pytorch

```
self.encoder = nn.Embedding(input_size, hidden_size)
self.rnn = nn.GRU(hidden_size, hidden_size, n_layers) # << das ist alles!
self.decoder = nn.Linear(hidden_size, output_size)
```

Char RNNs

The original RNN produces 'somewhat english' output after a million steps:

DUKE VINCENTIO:

Maday; that the present for whitted it is fair have up my ours I am you and thy broose,

Bans.

Lord.

Warkerd him on

Your queen royall a goods to oned your field

Modern variants with GRU produce 'almost Shakespeare' results after only a few iterations:

Why: my lord, unto them like

By her a loves to the sky.

PROSPERO:

A foul mother's liege and all work