Deep Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-08 Dozent: Karsten Flügge

Themen des Tages

Tag 11

Recurrent Neural Networks (RNN)

Sequenzanalyse, Analyse von Zeitreihen
Backpropagation through time (BPTT)

Recurrent Neural Networks (RNNs)

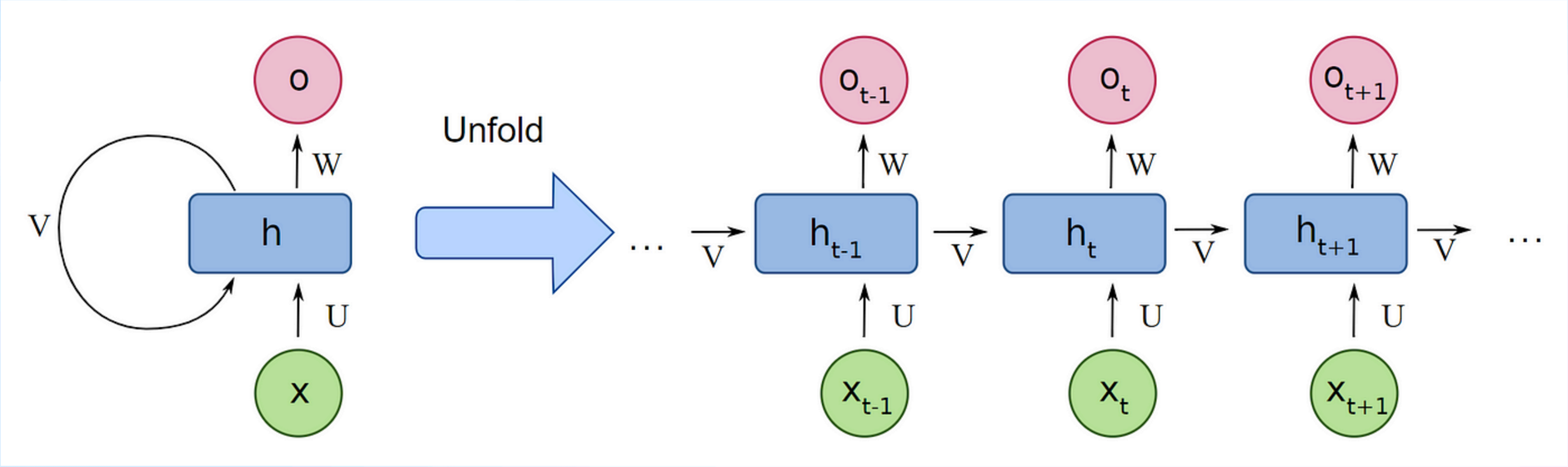
Rekurrenz: RNNs verwenden rekurrente (rückläufige, wiederkehrende) Schichten, bei denen die Ausgabe eines Neurons nicht nur an die nächste Schicht weitergegeben wird, sondern auch **zurück in die gleiche Schicht.** Dies ermöglicht das Erinnern an frühere Eingaben.

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (**RNNs**) sind eine Klasse von neuronalen Netzwerken, die speziell für **sequenzielle Daten** entwickelt wurden. Sie haben die Fähigkeit, Informationen über **vorherige** Einträge in einer Sequenz zu speichern und zu nutzen. Dies macht sie besonders nützlich für Aufgaben wie **Spracherkennung**, maschinelles Übersetzen und **Zeitreihenanalyse.**

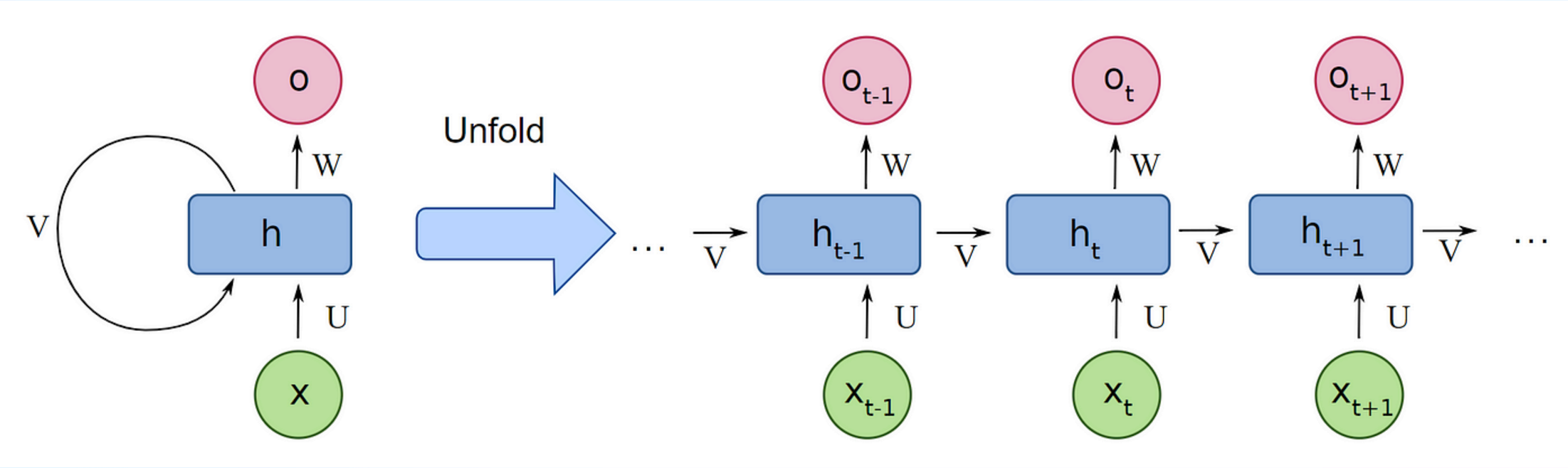
Recurrent Neural Networks (RNNs)

Gedächtnis: RNNs speichern Informationen über die Sequenz, die sie bisher gesehen haben, indem sie ihren versteckten Zustand (**hidden state**) kontinuierlich aktualisieren.
Schleifenstruktur: Jedes Element in der Sequenz beeinflusst den versteckten Zustand, der dann die Verarbeitung des nächsten Elements beeinflusst.
Parameterteilung: Dieselben Gewichte werden auf jedes Element der Sequenz angewendet, was den Speicherbedarf reduziert und zu Resonanzeffekten führen kann.



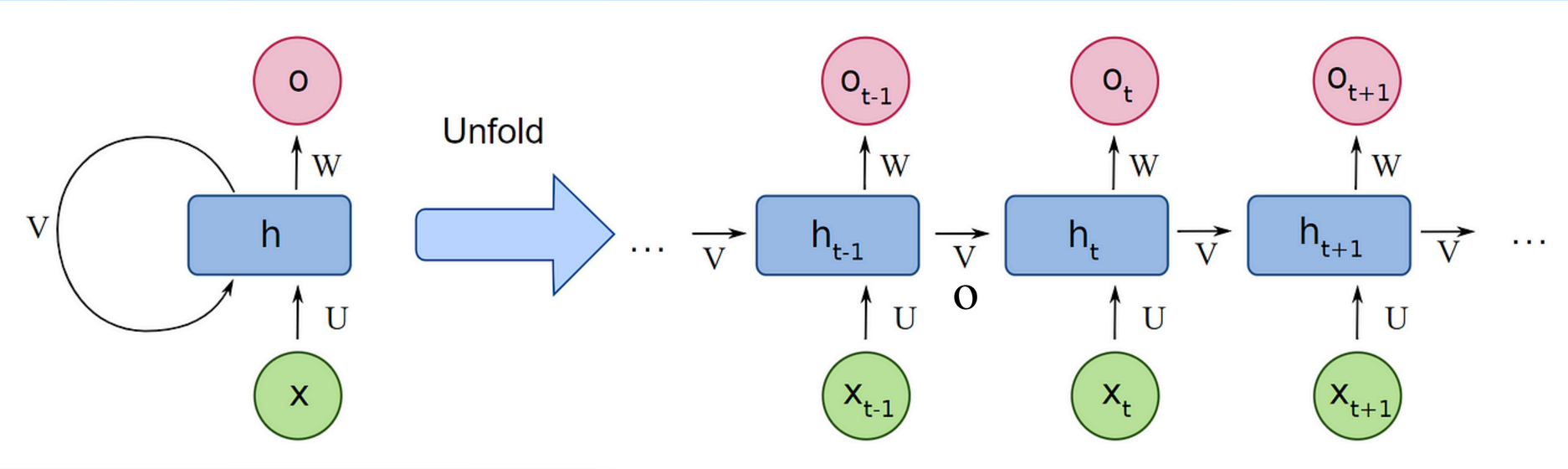
Recurrent Neural Networks (RNNs)

Statt einer echten **Schleife**, wird in der Praxis die Zelle **n-mal** wiederholt.
Zeit entspricht einfach **n-Kopien** des selben Elementes, also ähnlich der Zusatzdimension batch beim Tensor oder rolling average.
Diesen Trick nennt ausrollen / entfalten / "**unfold**ing"



Recurrent Neural Networks (RNNs)

Erster Ansatz: triviales RNN funktioniert nur sehr schlecht wegen **exploding / vanishing gradients**



Recurrent Neural Networks (RNN) II

nächster output vom RNN:

$y = \sigma(W \cdot x + b)$ klassisches Perceptron

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

W Gewichtsmatrix für Input x

$\mathbf{h_t}$ **'hidden' Inneren Zustand,** wird auch als **output y** benutzt

$\mathbf{W_h}$ Gewichtsmatrix für Inneren Zustand 'hidden'

b bias

σ beliebige Aktivierung, z.B. sigmoid, tanh (typisch)

=> Mathematik kann übersichtlicher sein als Code

Recurrent Neural Networks (RNN) II

Achtung: "number of layers" ≠ sequence length

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

Wir können RNNs auch stapeln,
dann ist einfach output vom letzten input vom nächsten

Achtung: "number of layers" ≠ sequence length

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

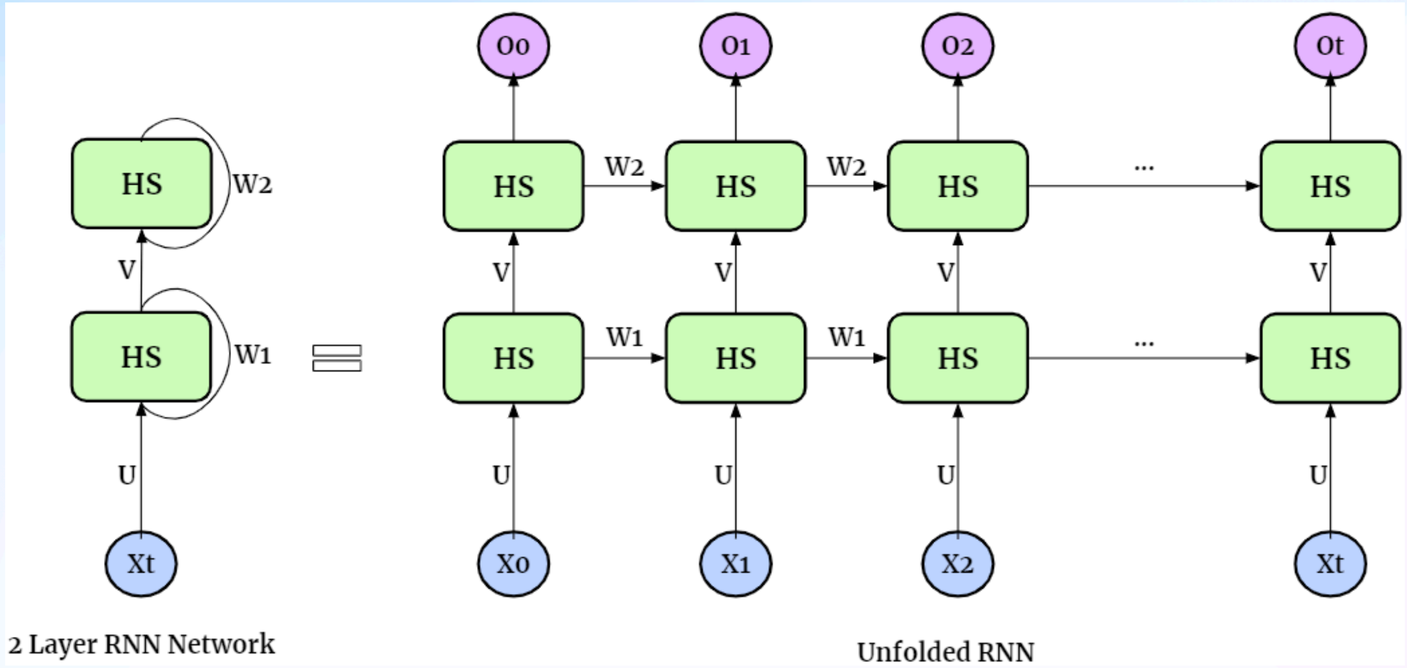
Recurrent Neural Networks (RNN) II

Achtung: "number of layers" ≠ sequence length

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

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

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



Exploding und Vanishing Gradients in RNN

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

Modifizierte RNNs:

Aufgabe

Experimentiert mit sine_gru.py vom root ordner.

Verstehe manual_rnn_block.py und baue diesen als Ersatz ein in sine_gru.py
(statt torch.nn.RNN / torch.nn.GRU)

Vorteile und Nachteile

Vorteile:

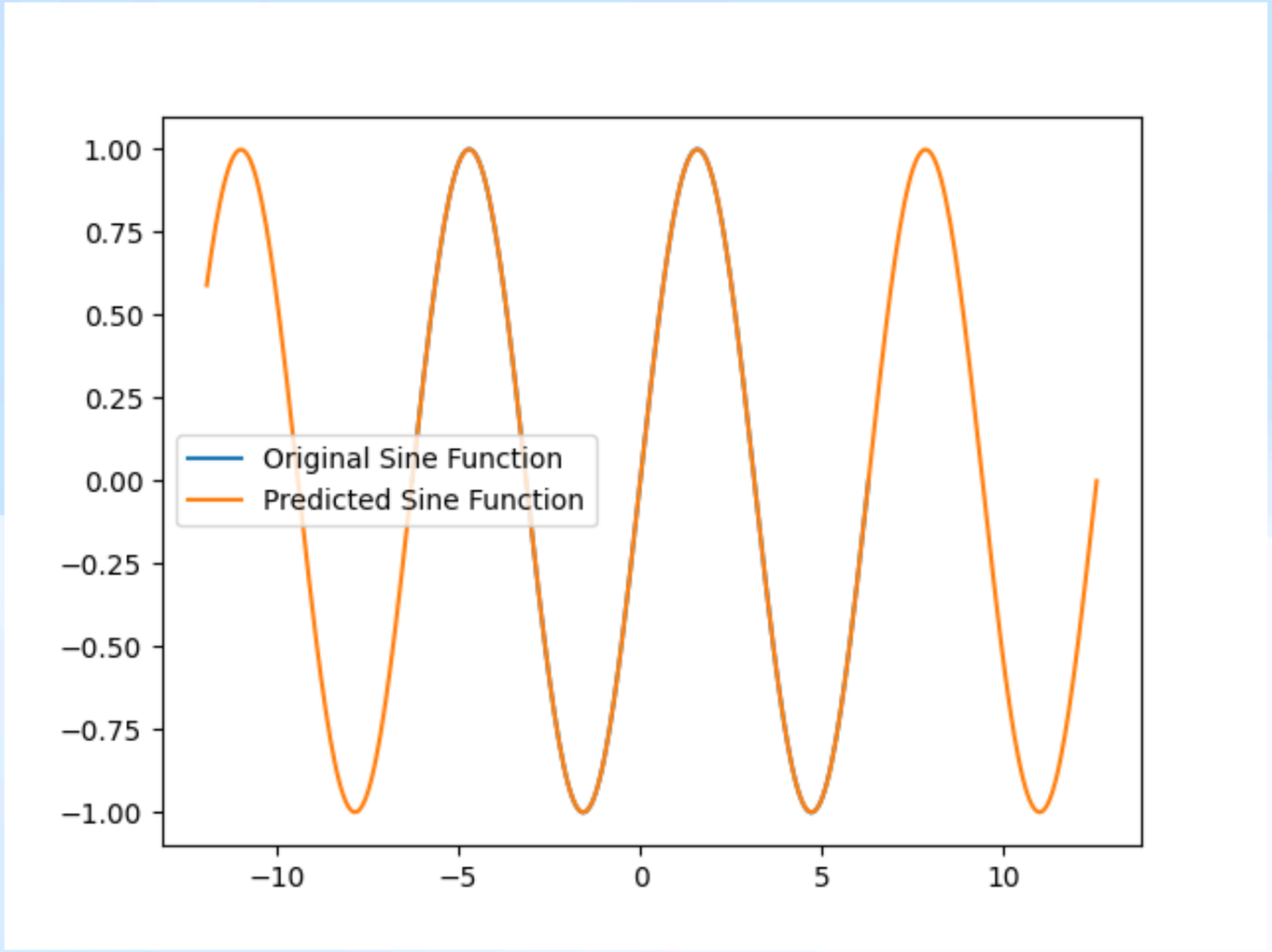
Fähigkeit, **Sequenzabhängigkeiten** zu modellieren.
Flexibilität in der Verarbeitung von Eingaben variabler Länge.
Gut fur Sprachmodellierung, maschinelle Übersetzung und Zeitreihenanalyse

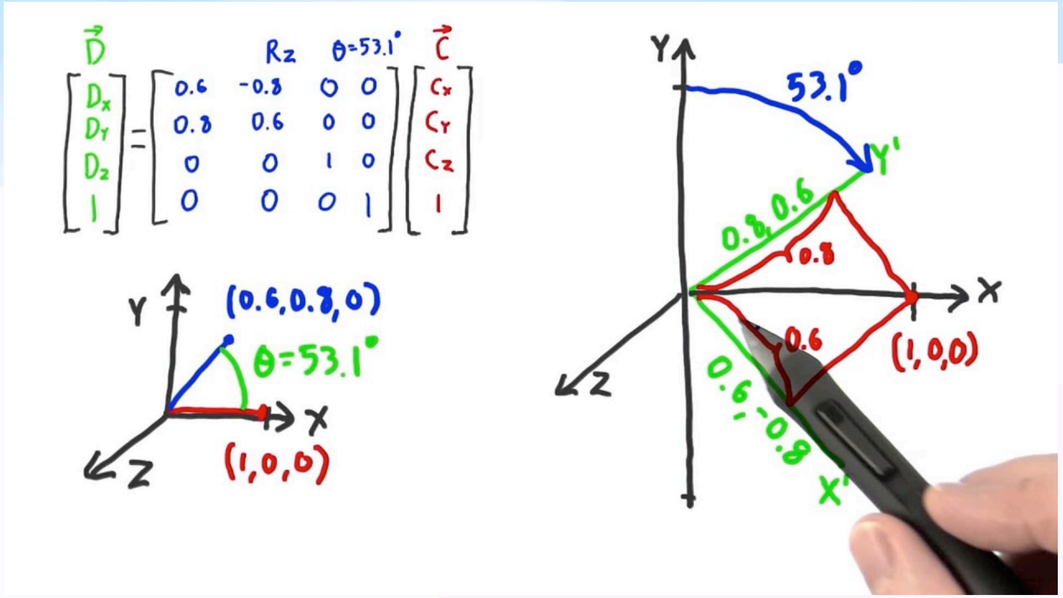
Nachteile:

Begrenztes Langzeitgedächtnis, was in der Praxis die Modellierung langer Sequenzen erschwert.
Komplexität und **längere Trainingszeiten** im Vergleich zu Feedforward-Netzwerken.
Training kann aufgrund von Problemen wie verschwindenden oder explodierenden Gradienten **schwierig** sein.

Jeder Zustand hängt von der Fertigstellung der Berechnungen des vorherigen Zeitschritts ab. Deswegen lassen Sie sich **nicht parallelisieren**! Das ist der Hauptgrund warum sie (fast) **komplett von *Attention*-architekturen abgelöst** werden (sollten).

Sequenzanalyse, Analyse von Zeitreihen





RNNs können als "**Resonatoren**" agieren, weil V den Zustand rotieren kann.

Backpropagation Through Time (BPTT)

Backpropagation Through Time (BPTT): Ein spezieller Algorithmus zur Anpassung der Gewichte in RNNs, der den Gradienten über mehrere Zeitschritte zurückpropagiert.

Time is just an *extra dimension*! Klassischer BP.

Weitere Aspekte

- ***Bidirektionale* RNNs** können Informationen aus der Vergangenheit und der Zukunft nutzen.
- **Long Short-Term Memory (*LSTM*)** und **Gated Recurrent Unit (GRU)** sind erweiterte Varianten von RNNs, die entwickelt wurden, um die Probleme des einfachen RNNs zu adressieren.
- **Attention Mechanismen** und **Transformers** haben in vielen Bereichen RNNs ersetzt, da sie effizienter sind und längere Abhängigkeiten besser modellieren können und parallelisierbarer sind.

💡in pytorch kann man einfach **nn.RNN** mit **nn.GRU** austauschen
Die API ist kompatibel aber GRU zeigt viel besseres Lernverhalten

RNN ≠ RvNN

⚠️ Verwechslungsgefahr und Vorausschau:

Recurrent Neural Networks (RNNs) ≠

Recursive Neural Networks (RvNNs) sind eine Erweiterung von neuronalen Netzwerken, die auf rekursive Datenstrukturen wie Bäume angewendet werden. Sie sind besonders nützlich für die Verarbeitung hierarchischer Daten, wie sie oft in der natürlichen Sprachverarbeitung (z.B. Parsing) und Computer Vision vorkommen.

… ≠ Reinforcement Learning Netze (RL)

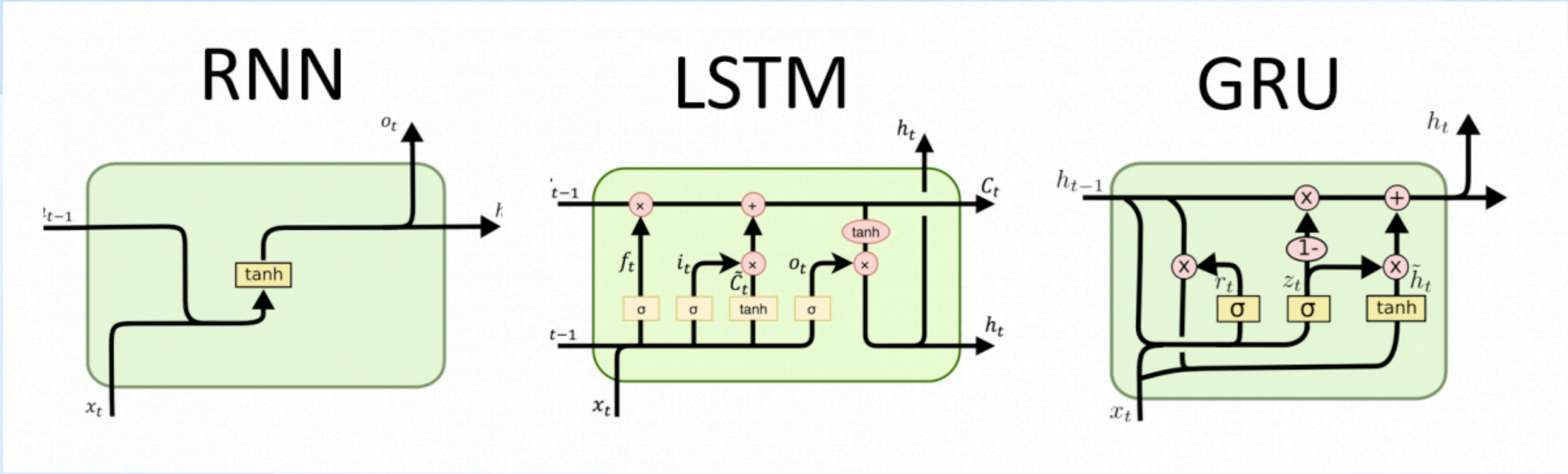
RNN ≠ RvNN

⚠️ Rekursion gibt es im Hirn!

⚠️ aber können wir **nicht** mit Neuronalen Netzen emulieren

Warum? Weil Backpropagation nur auf deterministischen Gradienten funktioniert.

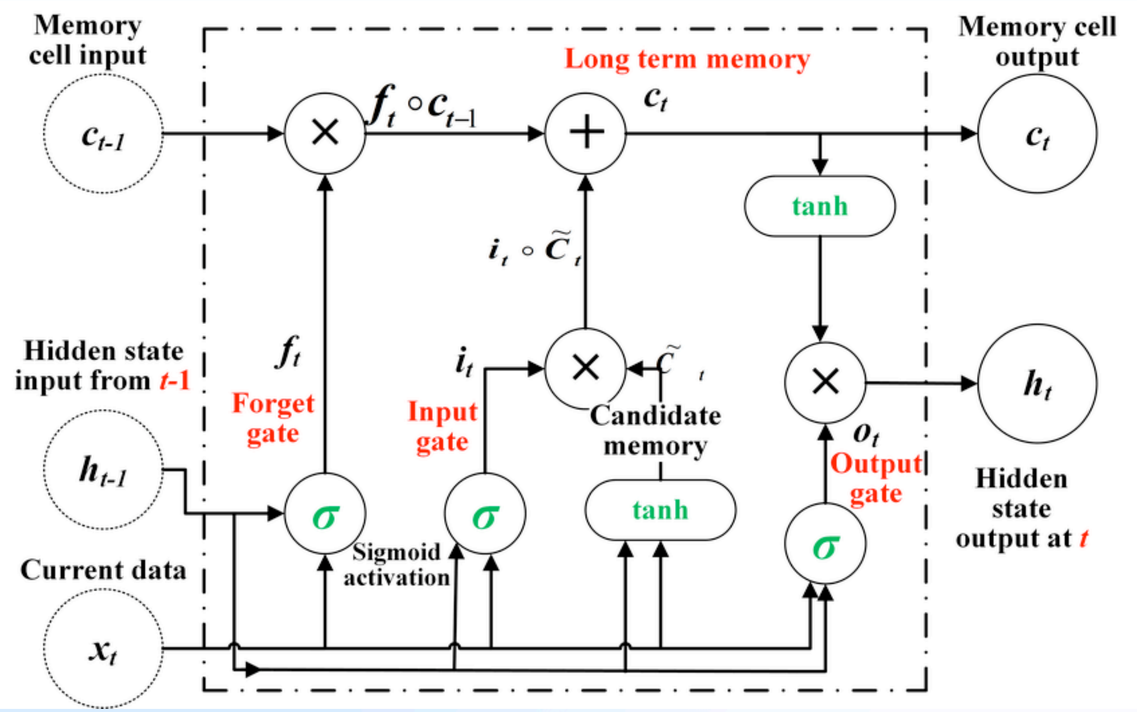
y = f(x) f=f(x,f(x)) => **chaos**



LSTM

Long Short Term Memory Block

Komplexes Rechenschema mit intuitiven Ideen



kann drastisch vereinfacht werden mit der selben 'Mächtigkeit' des Blocks:

GRU

General Recurrent Unit (Block)

Grundideen:

Merke alten **output** als hidden state.

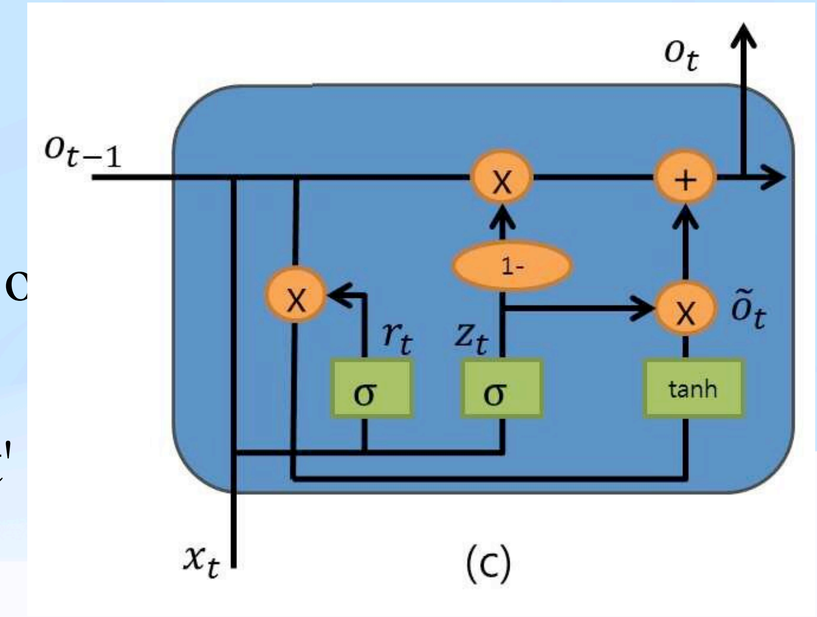
Steuere wie viel vom neuen **Input** x und altem **Gedächtnis** c

x_t input

$v_t \approx$ **o**(t-1) **o**ld **o**utput ≈ h(hidden) alter output wird 'gemerkt'

o_t output

stacked: number of layers = "time steps"



Drei Kanäle / **Gates**:

Update Gate: Steuert wie stark o mit neuer Information x_t input upgedated werden soll

GRU

General Recurrent Unit (Block)

Grundideen:

Merke alten **output** als hidden state.

Steuere wie viel vom neuen **Input** x und altem **Gedächtnis** o verwendet wird.

x_t input

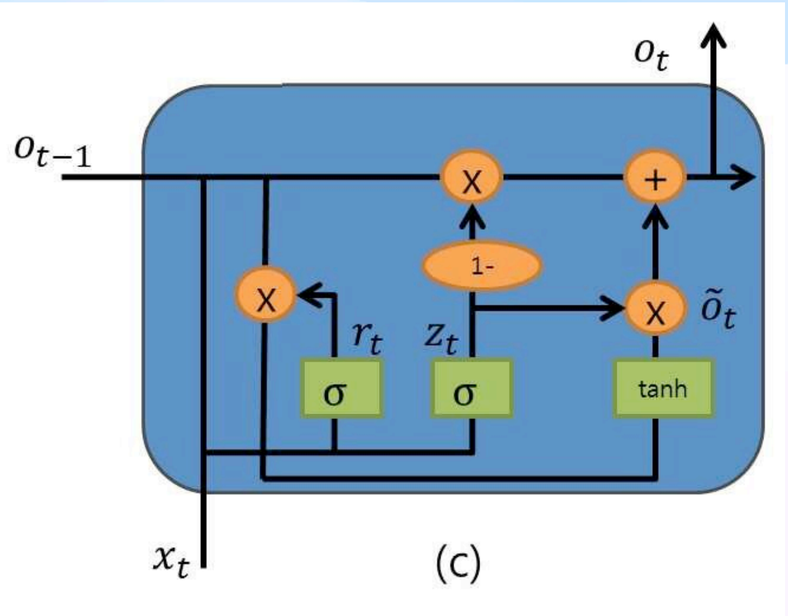
$v_t \approx$ **o**(t-1) **o**ld **o**utput ≈ h(hidden) alter output wird 'gemerkt'

o_t output

stacked: number of layers = "time steps"

r reset gate

z



Vokabeln

Generalisierung:
Verallgemeinerung des Gelernten auf noch nicht gesehene Daten (-Bereiche)
"**Out of distribution**" Ausserhalb der Verteilung:
Test Daten waren nicht in den Trainingsdaten
LSTM = Long Short Term Memory (Cell/Block/Unit/Net)
GRU = **Gate**d Recurrent Unit
Gate = Kontrolle des Informationsflusses (0…1)*x

Buch

Kapitel 15
Verarbeiten von Sequenzen mit RNNs und CNNs 577-616
(noch nicht Kapitel 16)

Aufgabe:
Versucht BasicRNNCell zu verstehen in manual_rnn_block.py
Versucht GRUModel zu verstehen in sine_gru.py

z.B.)

a) `out = self.fc(out[:, -1, :])` warum -1 ?

b) `out = self.fc(out[:, -1, :])` warum nicht `out[:, :, -1]`

Experimentieren!
c) andere Funktionen mit GRU approximieren. Welche gehen gut, welche nicht?
d) eigene Daten / Börsenkurs? Projektarbeit;)