Deep Learning

Recurrent Neural Networks

Technische Hochschule Rosenheim
Sommer 2023
Prof. Dr. Jochen Schmidt

Acknowledgements

Many of the slides presented here are based on the Deep Learning Slides Summer Semester 2020, courtesy of
A. Maier, V. Christlein, K. Breininger, F. Denzinger, F. Thamm,
Pattern Recognition Lab, Friedrich-Alexander-University Erlangen-Nürnberg.
https://lme.tf.fau.de/

Overview

- Simple Recurrent Neural Networks (RNNs)
- Long Short-Term Memory Units (LSTMs)
- Gated Recurrent Units (GRUs)
- Comparison of simple RNN units, LSTM units and GRUs

Simple Recurrent Neural Networks (RNNs)

Motivation

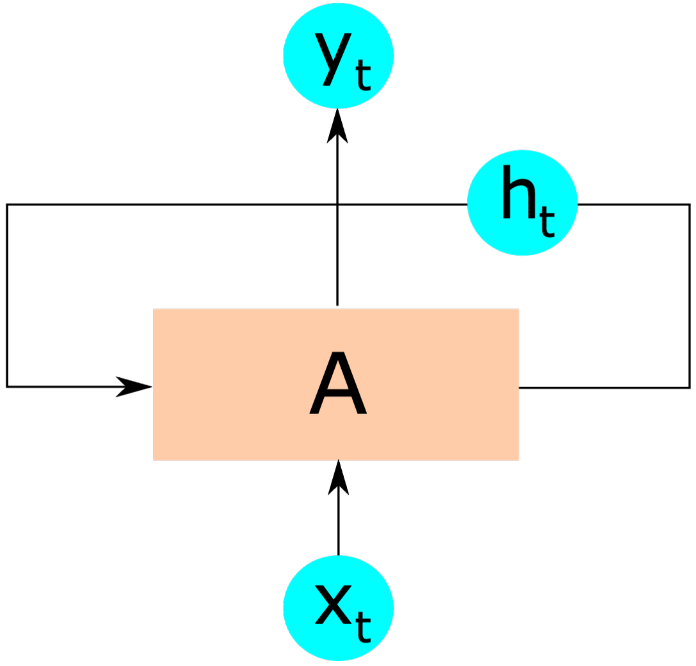
- So far: **One** input, e.g., single image
- Feed forward neural networks: input → processing → result
- But: there are lots of sequential or time-dependent signals, e.g.
  - Speech/Music (translation, music classification)
  - Video (object detection/face recognition)
  - Sensor data (speed, temperature, energy consumption,…)
- "Snapshots" often not informative (single word → translation?)

→ **Temporal context** is important!

Motivation

- How can we integrate this context in the network?
- Simple approach: Feed the whole sequence to a big network → Bad idea!
  - Inefficient memory usage
  - Difficult/impossible to train
  - Difference between spatial and temporal dimensions?
  - **Not real-time**! (translation,...)
- Better approach: Model sequential behavior within the architecture:
  → **Recurrent Neural Networks** (RNNs)

Basic RNN Structure (Elman Unit)



“Elman Unit”

- Current input x_t multiplied by weight
- **Additional input**: **Hidden state** h_{t-1} of the unit

→ Feedback loop:
use information from present and recent past to compute output y_t

- First models in 1970’s [Lit74] and early 1980’s [Hop82] (Hopfield Network)
- Simple recurrent neural network or Elman network introduced in *Finding Structure in Time* by Jeff Elman in 1990 [Elm90]

RNNs vs Feed-forward Networks

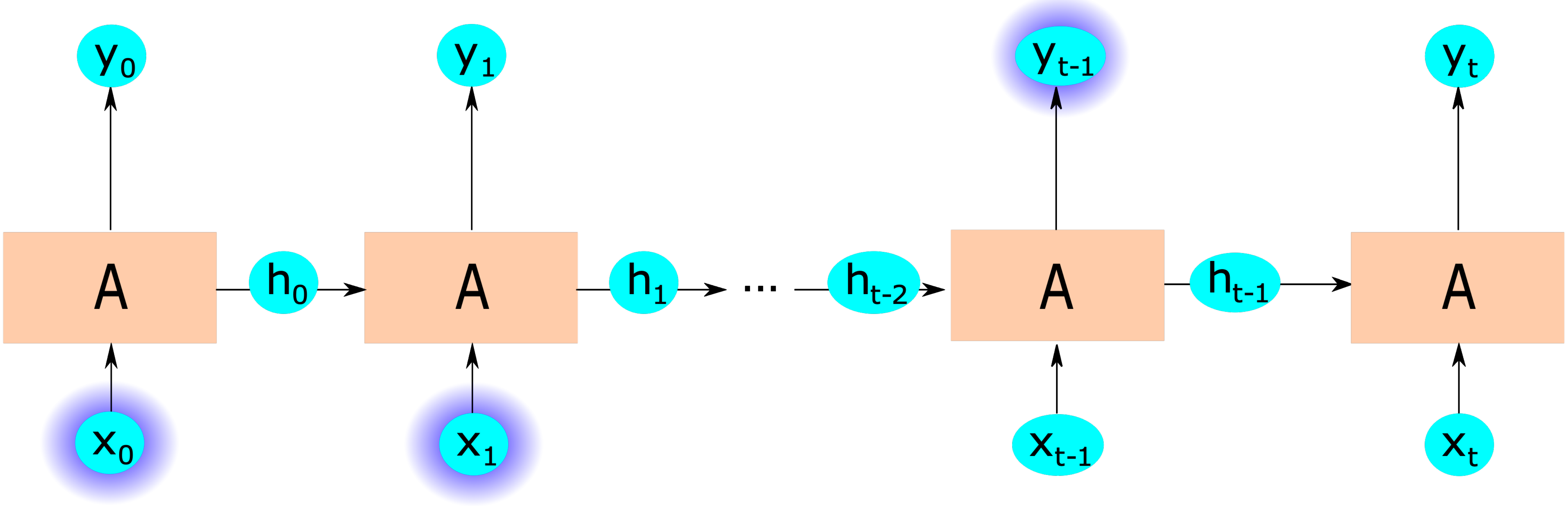
Technische Hochschule Rosenheim

- Feed-forward networks only feed information forward through the net
  - they do not have an internal state
- With recurrent neural networks, we can:
  - model loops
  - model memory and experience
  - learn sequential relationships
  - provide continuous predictions as data comes in → real-time

Unfolded RNN

- “Unfolded” RNN unit: sequence of copies of the **same** unit (= same weights)
  - Parameter-sharing over time
- Each unit passes hidden state as additional input to successor

→ Previous input can influence current output



Types of Sequences

one to one

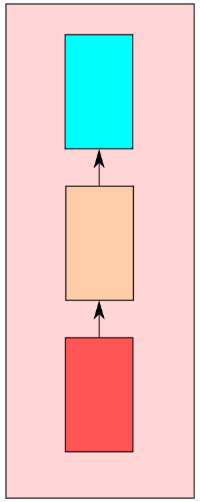


image classification
in: image
out: class label (fixed-size vector)
classic feed-forward

one to many

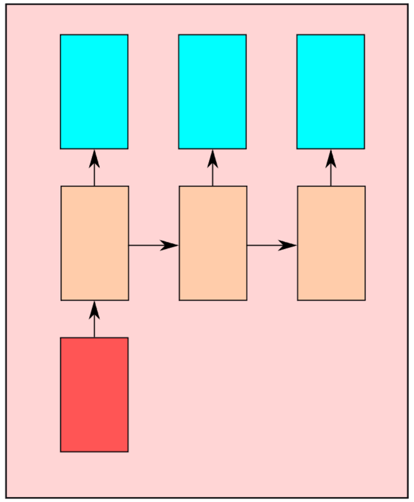
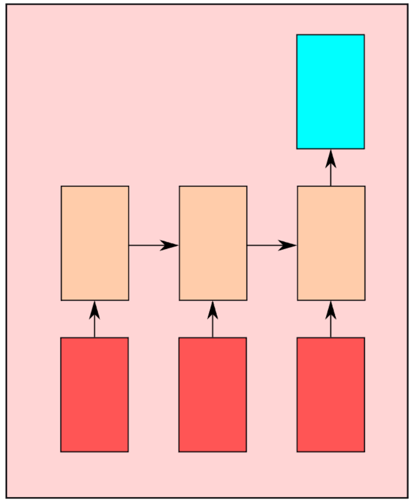


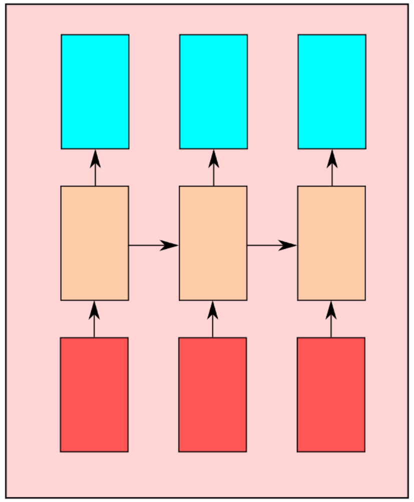
image captioning
in: image
out: text description (sequence)

many to one



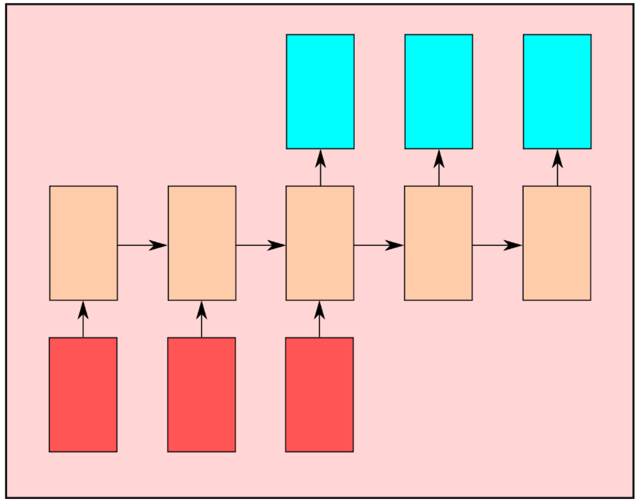
sentiment analysis
in: text (sequence)
out: class label (fixed-size vector)

many to many (sync)



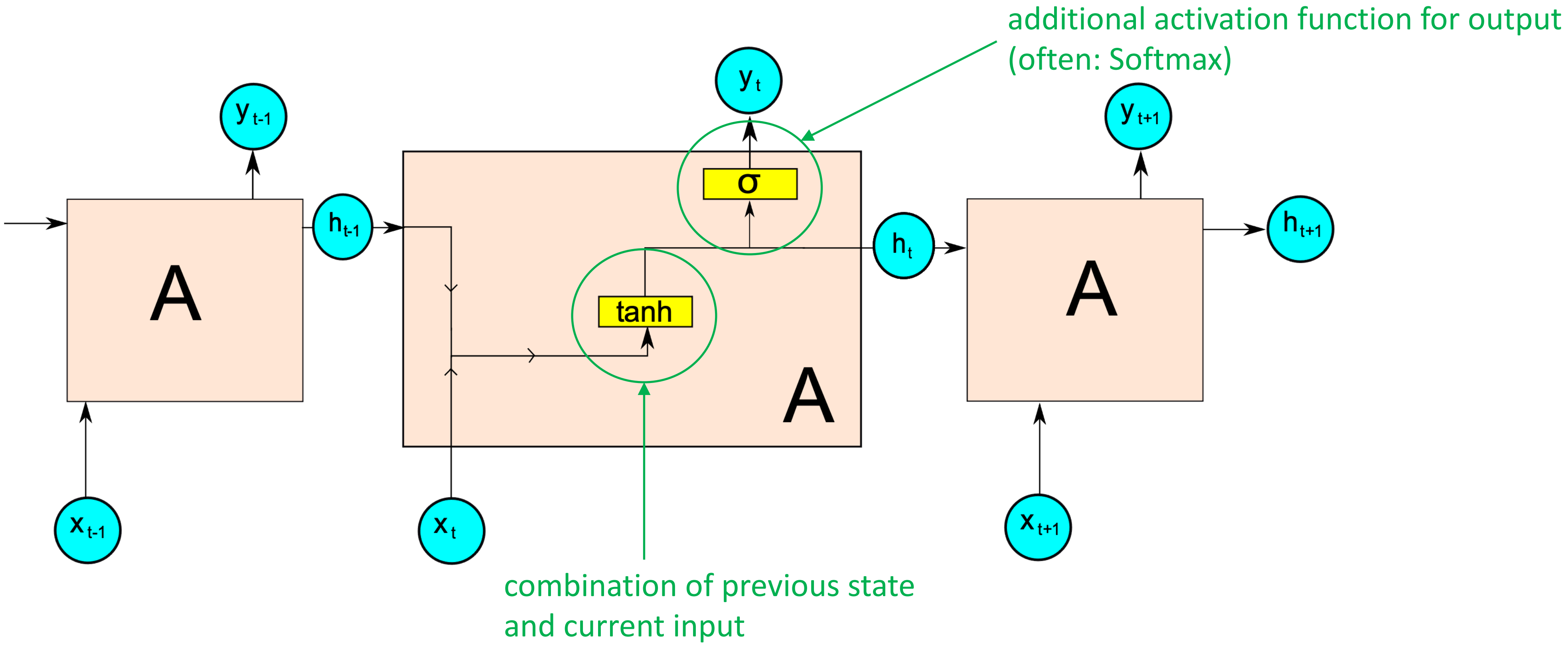
video classification
in: image stream (sequence)
out: label for each frame (sequence)

many to many (no sync)



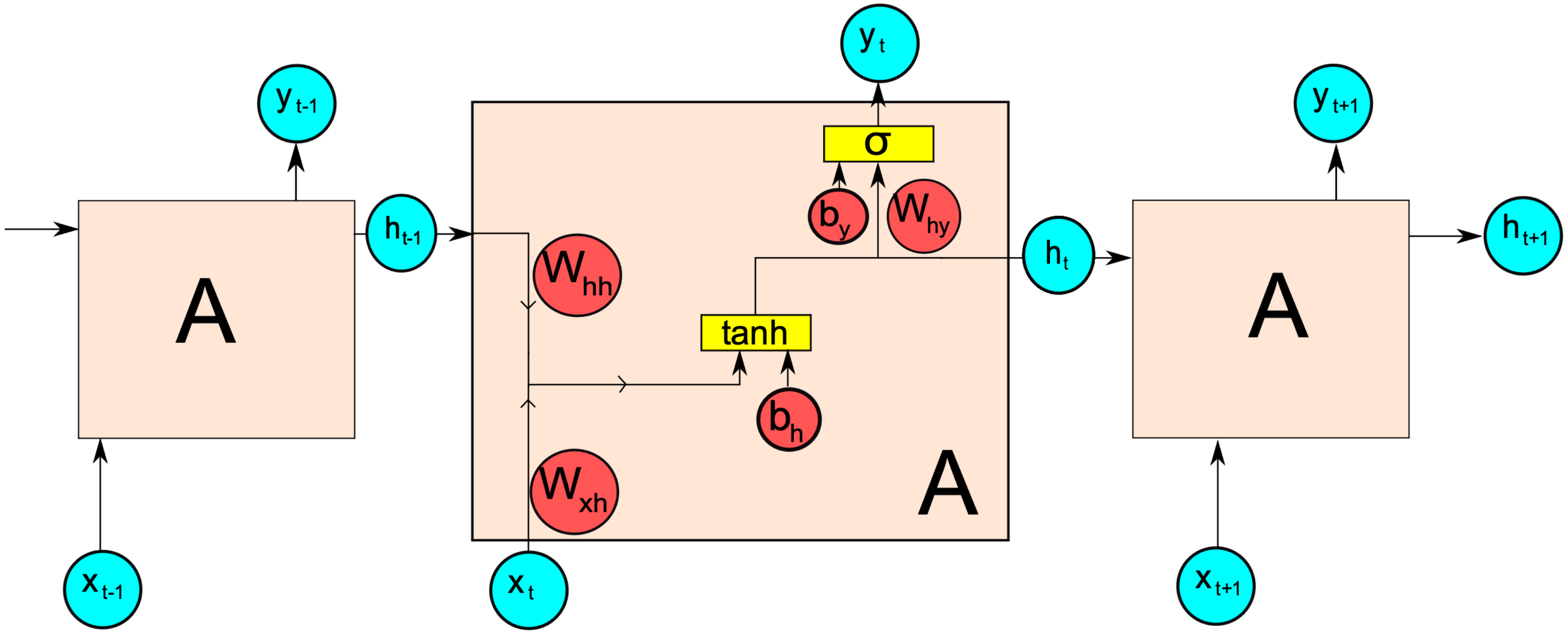
machine translation
in: German text (sequence)
out: English text (sequence)
input and output sequences not necessarily of same length

Internal Structure of RNN Unit



Question 1: How do we update the hidden state?
Question 2: How do we combine input and hidden state to compute output?

Close-up: How to Update the Hidden State?



apply element-wise

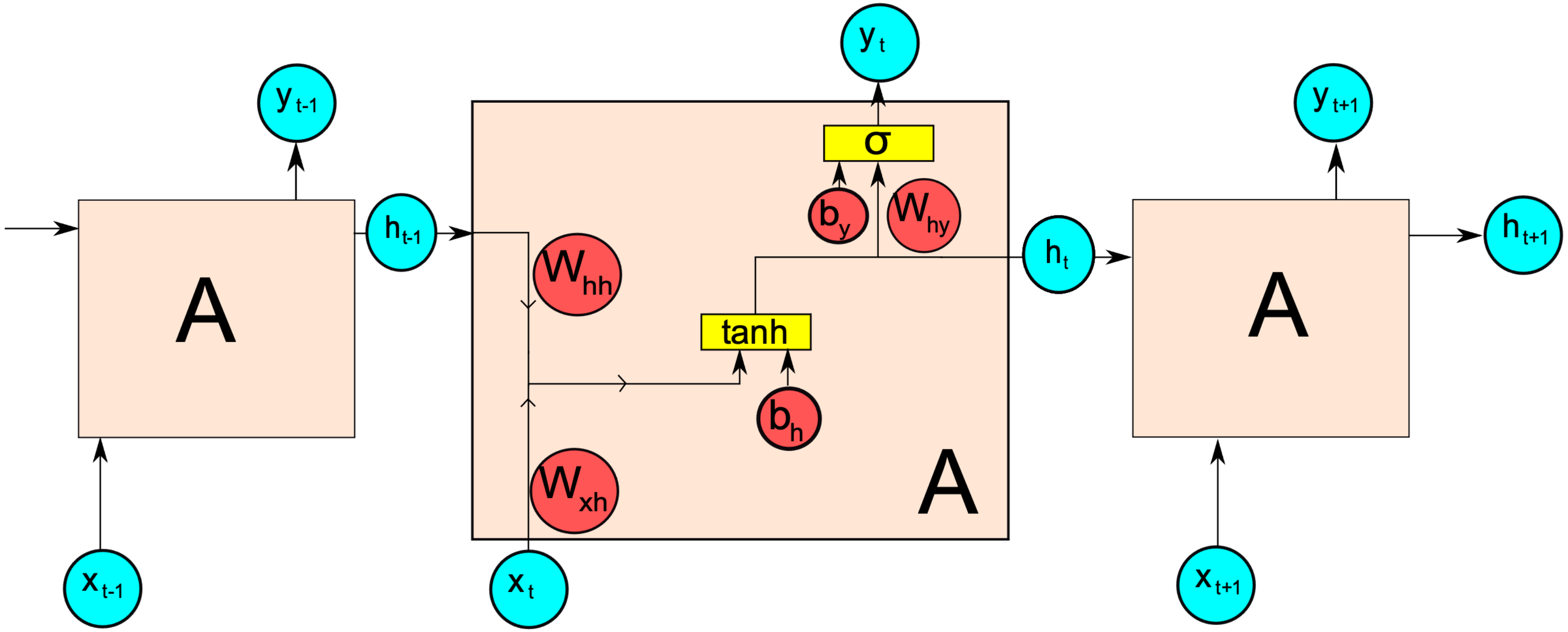
weight matrix for current input (dim($\boldsymbol{h}$) x dim($\boldsymbol{x}$))

$$\boldsymbol{h}_t = \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_h)$$

weight matrix for previous hidden state (square, matching $\boldsymbol{h}$)

update bias

Close-up: How to Compute the Output?



$$y_t = \sigma(W_{hy} h_t + b_y)$$

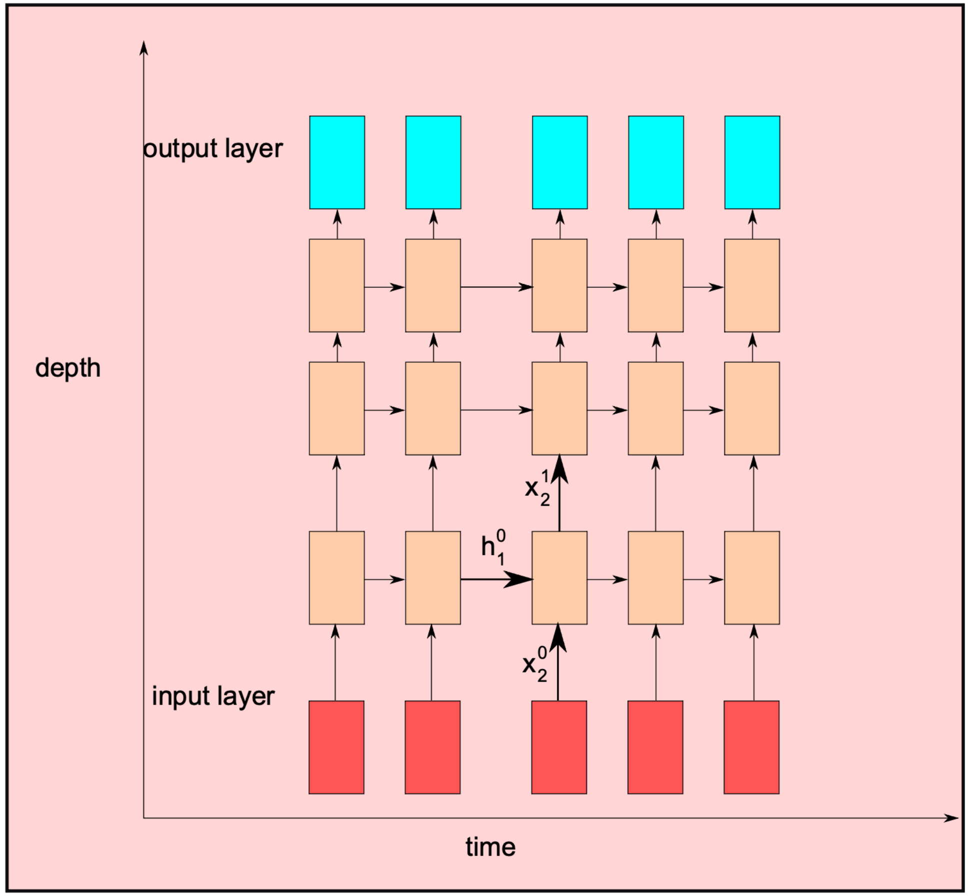
weight matrix for current hidden state

output bias

Deep RNNs

Technische Hochschule Rosenheim

Stack multiple Elman units



$$\boldsymbol{h}_t^l = \tanh\left(\boldsymbol{W}_{hh}^l \boldsymbol{h}_{t-1}^l + \boldsymbol{W}_{xh}^l \boldsymbol{x}_t^l + \boldsymbol{b}_h^l\right)$$

t: time index
l: layer index

- as in feed-forward networks: additional layers for each time step
- higher learning capacity
- more training data required
- in practice you find deep RNNs only very rarely, if at all

Example: Character Level Language Model

Task: Learn character probability distribution from input text

- Vocabulary: {h, e, l, o}
- Characters encoded as one-hot vectors:

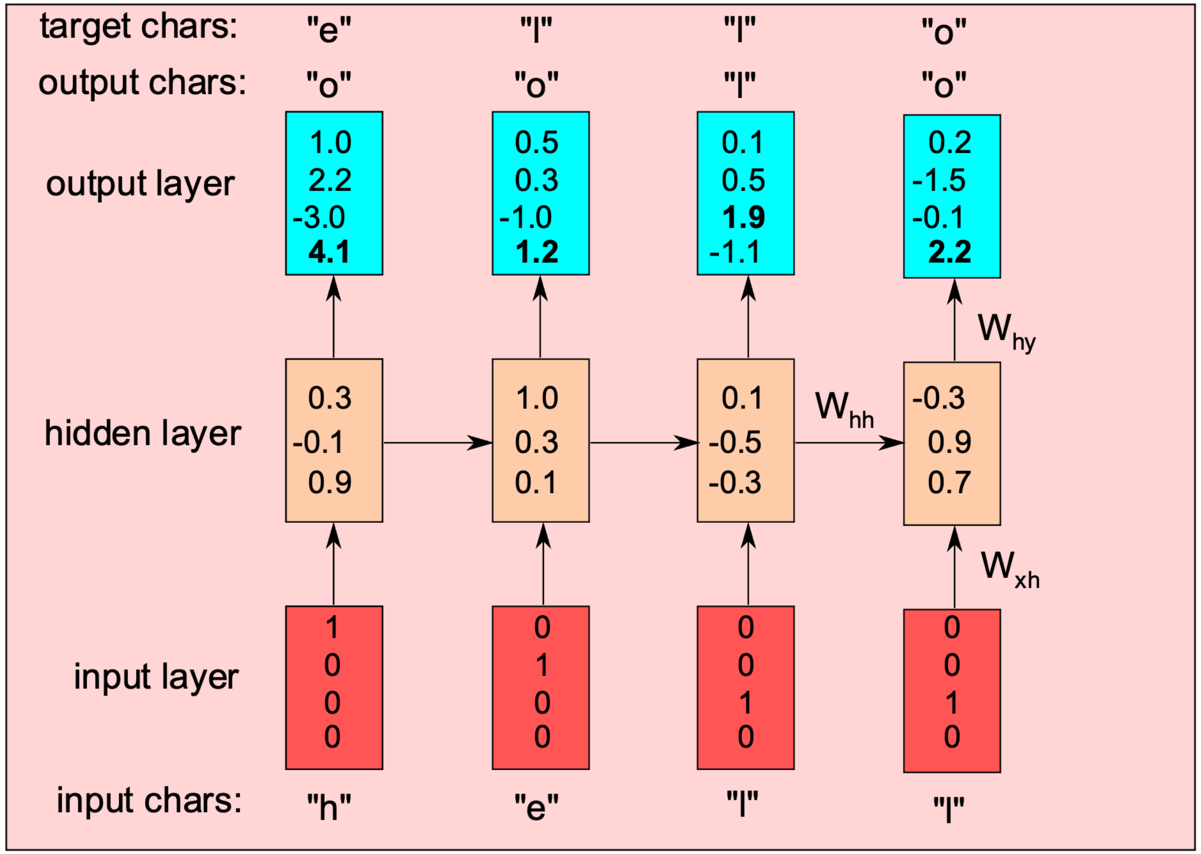
h	e	l	o
$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

- Train RNN on the sequence "*hello*":
  Given 'h' as first input, the network should generate the sequence "*hello*"
- Network needs to know **previous inputs**
  - when presented with 'l': Do we need another 'l' or an 'o'?
- Adapted from http://karpathy.github.io/2015/05/21/rnn-effectiveness

Example: Character Level Language Model

Technische Hochschule Rosenheim

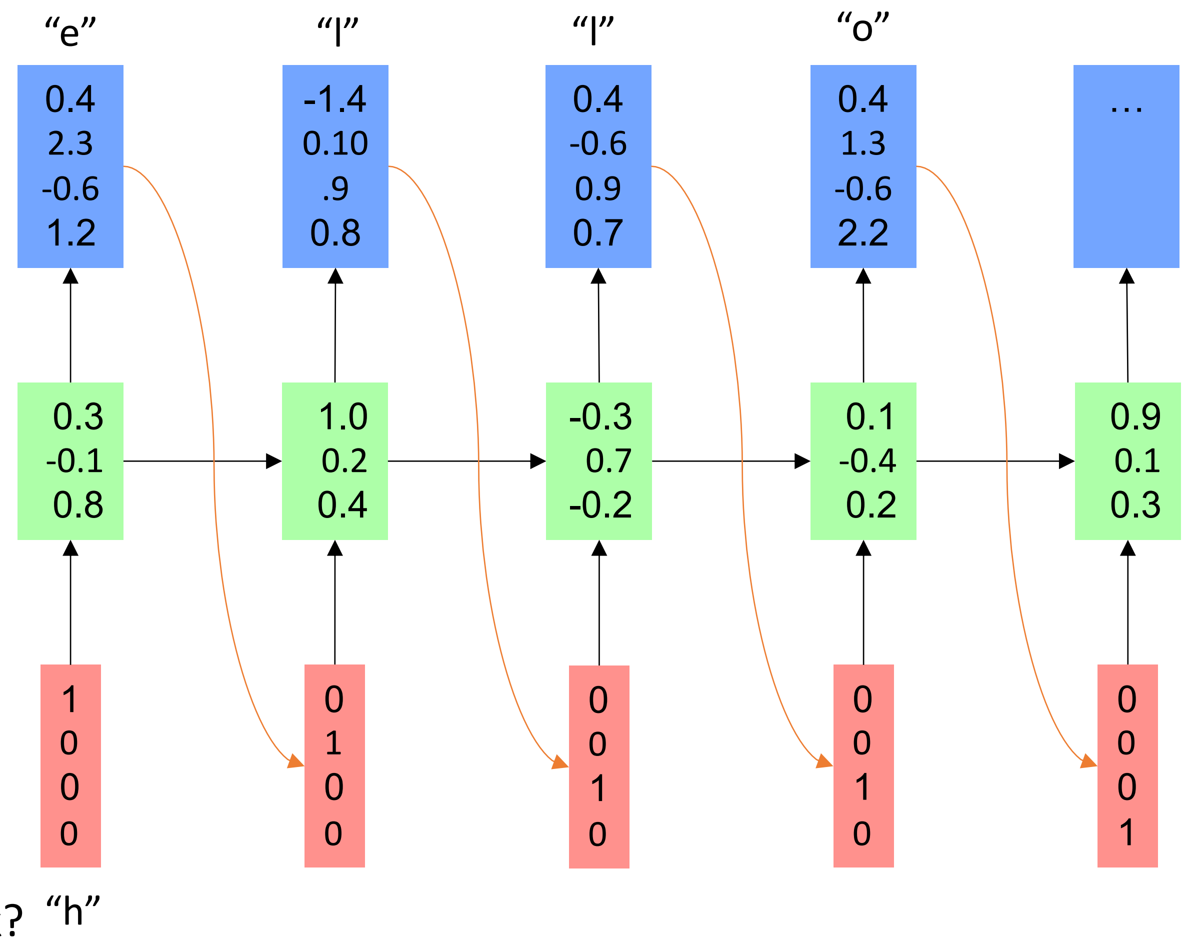
Prediction with random initialization:



Character Level Language Models at Runtime

Technische Hochschule Rosenheim

Desired result after training (example) – Goal: Maximize prediction for correct component



How can we now train this network?

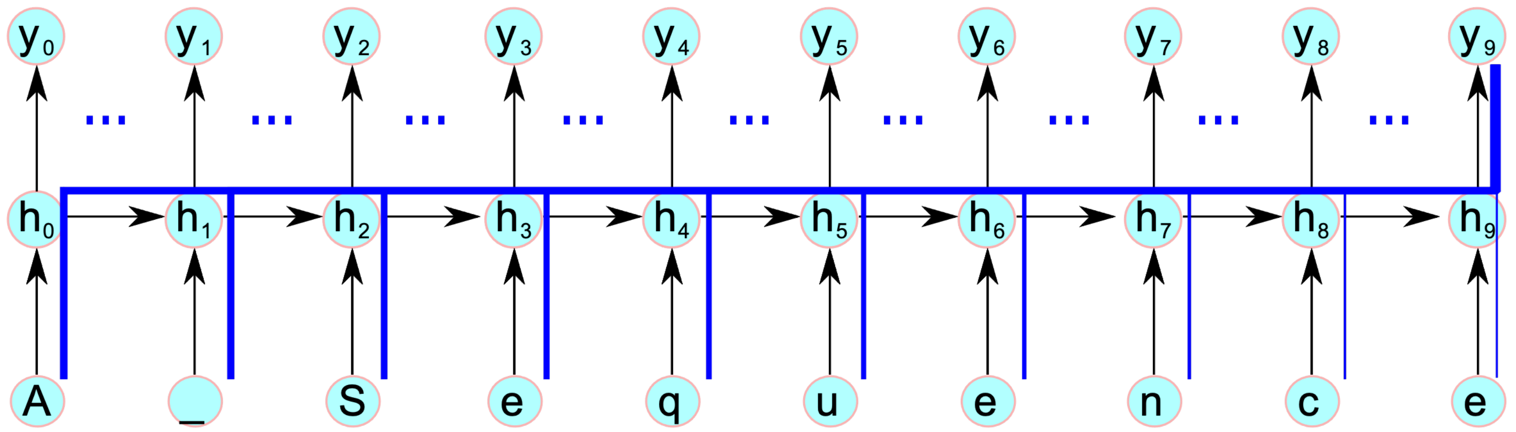
→ **Backpropagation through time (BPTT)**: train “unfolded” network

based on silde courtesy of Tobias Bocklet [Boc20]

Backpropagation Through Time (BPTT)

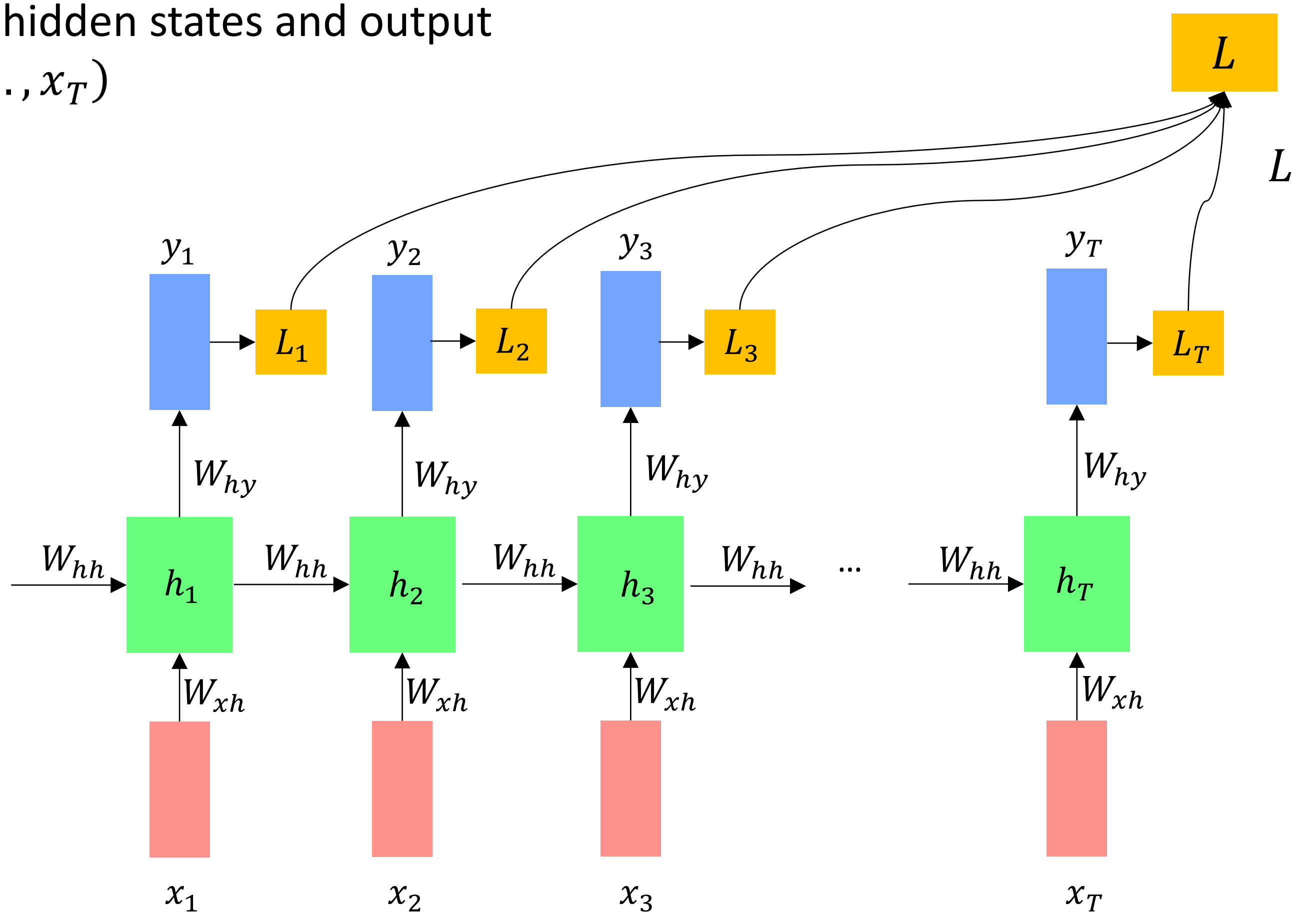
Concept: Train the unfolded network

- Compute forward pass for the full sequence → loss
- Compute backward pass for the full sequence to get gradients → weight update



BPTT – Forward Pass

Forward pass: Computation of hidden states and output using input sequence $(x_1, x_2, \dots, x_T)$



$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{t=1}^{T} L_t(\widehat{\boldsymbol{y}_t}, \boldsymbol{y}_t)$$

$\hat{\boldsymbol{y}}$: predicted output
$\boldsymbol{y}$: ground truth

based on silde courtesy of Tobias Bocklet [Boc20]

BPTT – Backward Pass

- Loss function, e.g., cross-entropy loss: $L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{t=1}^{T} L_t(\widehat{\boldsymbol{y}_t}, \boldsymbol{y}_t)$

- Compute gradient of the loss function L with respect to network parameters $\boldsymbol{\theta}$

$$\nabla_{\boldsymbol{\theta}} L = \left[\nabla_{\boldsymbol{W}_{xh}} L,\ \nabla_{\boldsymbol{W}_{hh}} L,\ \nabla_{\boldsymbol{W}_{hy}} L,\ \nabla_{\boldsymbol{b}_h} L,\ \nabla_{\boldsymbol{b}_y} L,\ \nabla_{\boldsymbol{h}_0} L\right]$$

- Update parameters using a learning rate α

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L$$

- How do we get these derivatives?
  Go “back in time” through the network

BPTT – Backward Pass

Go “backwards” through the unfolded unit, starting at final time step $t = T$
and iteratively compute gradients for $t = T, \dots, 1$
Reminder: $\hat{\boldsymbol{y}}_t = \sigma(\boldsymbol{o}_t) = \sigma\left(\boldsymbol{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y\right)$ $\qquad$ $\boldsymbol{h}_t = \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_h)$

$$\nabla_{\boldsymbol{o}_t} L = \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{o}_t}\frac{\partial L}{\partial \hat{\boldsymbol{y}}} = \sigma'(\boldsymbol{o}_t)\frac{\partial L}{\partial \hat{\boldsymbol{y}}} \qquad \underline{\nabla_{\boldsymbol{W}_{hy,t}} L} = \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{o}_t}\frac{\partial L}{\partial \hat{\boldsymbol{y}}}\frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{W}_{hy}} = \nabla_{\boldsymbol{o}_t} L\, \boldsymbol{h}_t^T \qquad \underline{\nabla_{\boldsymbol{b}_{y,t}} L} = \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{o}_t}\frac{\partial L}{\partial \hat{\boldsymbol{y}}}\frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{b}_y} = \nabla_{\boldsymbol{o}_t} L$$

The gradient $\nabla_{\boldsymbol{h}_t} L$ depends on two elements – the hidden state influences $\boldsymbol{o}_t$ as well as the next hidden state $\boldsymbol{h}_{t+1}$

$$\underline{\nabla_{\boldsymbol{h}_t} L} = \left(\frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}\right)^T \nabla_{\boldsymbol{h}_{t+1}} L \qquad + \left(\frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{h}_t}\right)^T \nabla_{\boldsymbol{o}_t} L$$

$$= \boldsymbol{W}_{hh}^T \tanh'(\boldsymbol{W}_{hh}\boldsymbol{h}_t + \boldsymbol{W}_{xh}\boldsymbol{x}_{t+1} + \boldsymbol{b}_h)\ \nabla_{\boldsymbol{h}_{t+1}} L \ + \boldsymbol{W}_{hy}^T\ \nabla_{\boldsymbol{o}_t} L$$

Note: For $t = 0$ (to get $\nabla_{\boldsymbol{h}_0} L$) and $t = T$, we only need one element of the sum.

BPTT – Backward Pass

Using $\nabla_{\boldsymbol{h}_t} L$ we get the remaining gradients.

Reminder: $\boldsymbol{h}_t = \tanh(\boldsymbol{u}_t) = \tanh(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_h)$

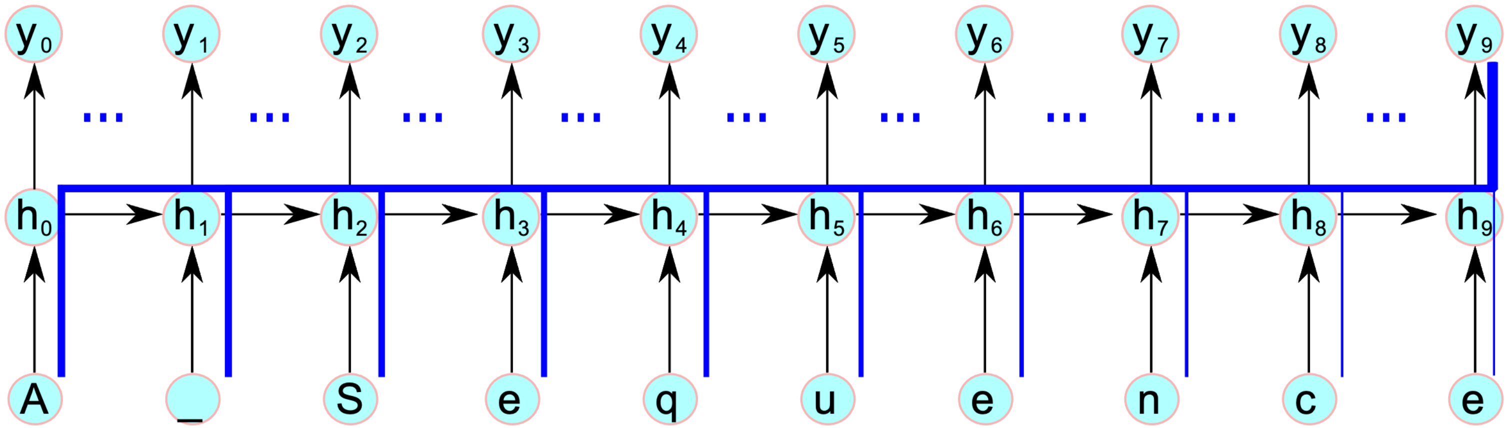
$\nabla_{\boldsymbol{W}_{hh,t}} L = \nabla_{\boldsymbol{h}_t} L \cdot \tanh'(\boldsymbol{u}_t) \cdot \boldsymbol{h}_{t-1}^T$ $\qquad$ $\nabla_{\boldsymbol{W}_{xh,t}} L = \nabla_{\boldsymbol{h}_t} L \cdot \tanh'(\boldsymbol{u}_t) \cdot \boldsymbol{x}_t^T$ $\qquad$ $\nabla_{\boldsymbol{b}_{h,t}} L = \nabla_{\boldsymbol{h}_t} L \cdot \tanh'(\boldsymbol{u}_t)$

Currently, the gradients depend on t. How do we get the gradient for the sequence?

- The unrolled unit is a network with shared weights (over time)
- For each gradient, simply sum over all time-steps $t = 1, \dots, T$

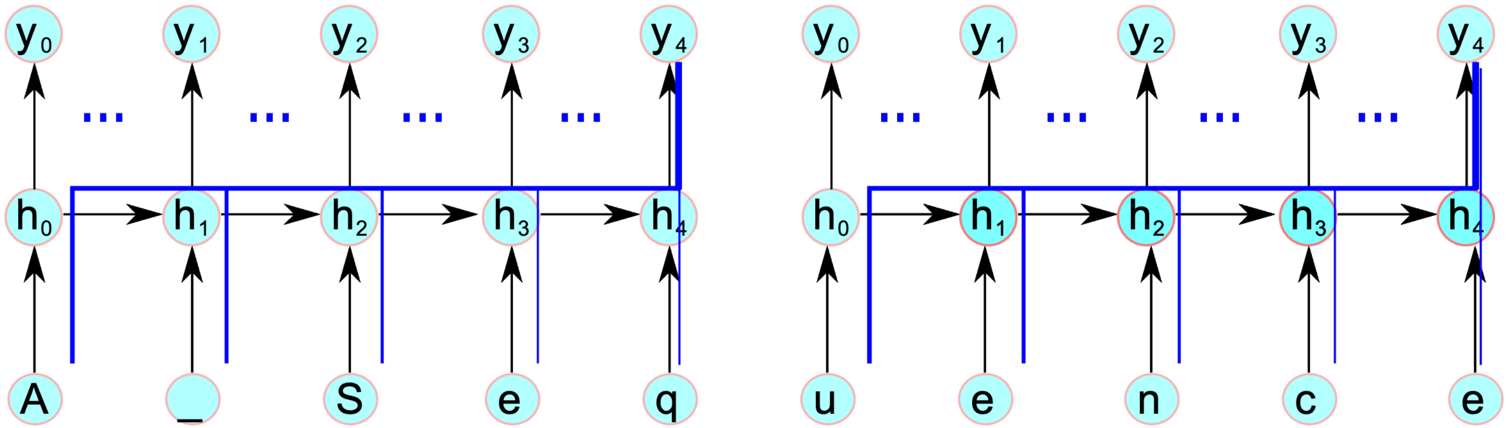
Normal BPTT

Technische Hochschule Rosenheim



One update requires backpropagation through a complete sequence
→ Single parameter update is very expensive!

Naive BPTT

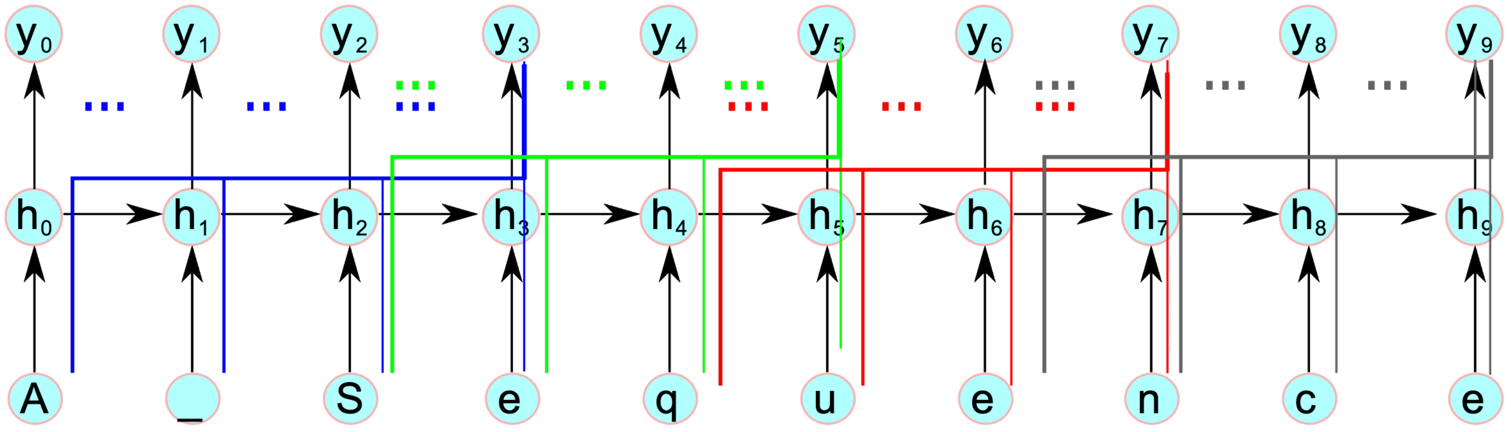


Naive Solution:

- Split long sequences into batches of smaller parts
- Might work ok in practice, but blind to long-term dependencies
- Can we do better? Yes!

→ **Truncated backpropagation through time** (TBPTT)

Truncated Backpropagation Through Time (TBPTT)



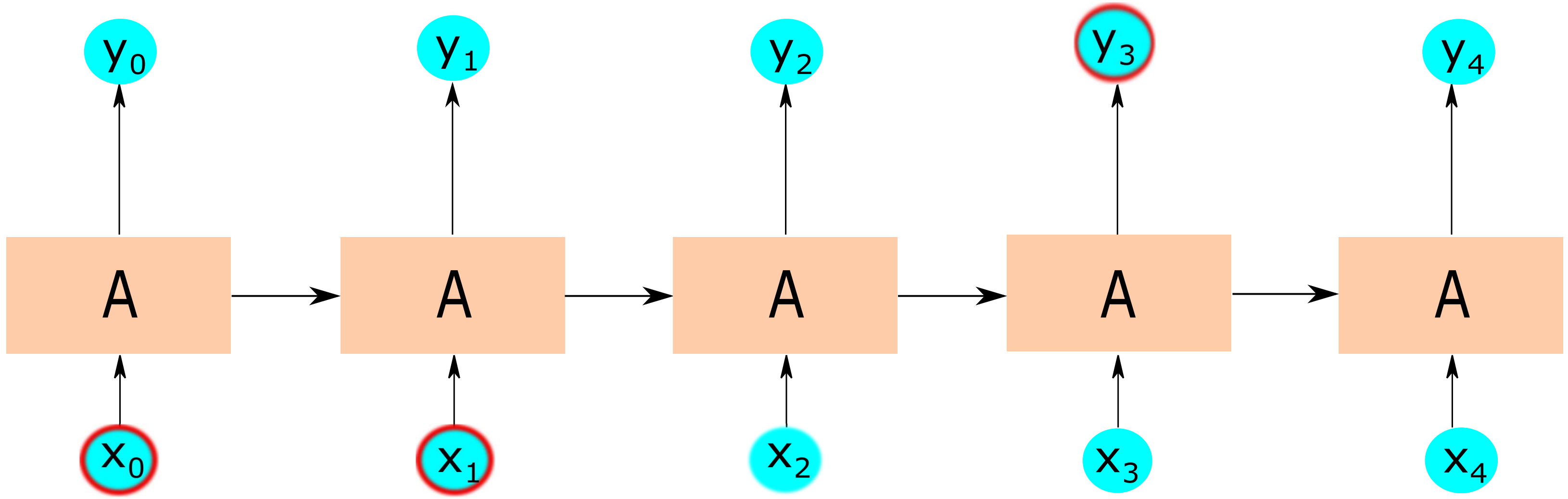
- Main idea: Keep processing sequence as a whole
- Adapt frequency and depth of update:
  - Every k_1 time steps, run BPTT for k_2 time steps
  - Parameter update cheap if k_2 small
- Hidden states are still exposed to many time steps

Algorithm:
for $t = 1$ to T do:
  Run RNN for one step, computing $\boldsymbol{h}_t$ and $\hat{\boldsymbol{y}}_t$
  if $t \bmod k_1 == 0$:
    Run BPTT from t down to $t - k_2$

So can we successfully train RNNs now? Still no...

The Long-Term Dependency Problem with Basic RNNs

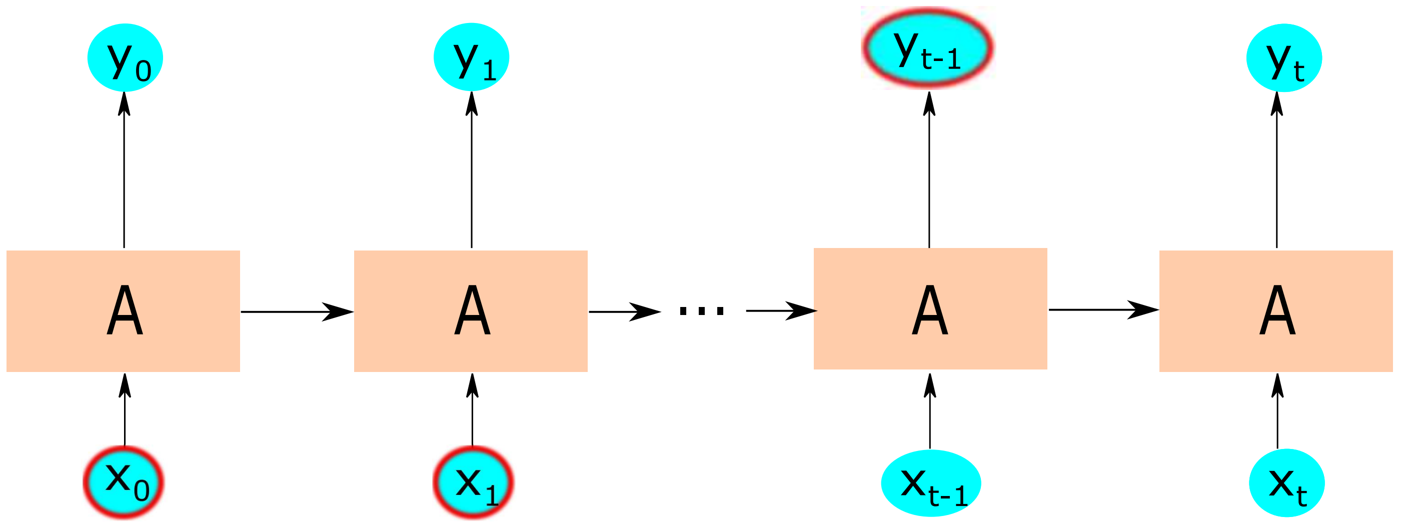
- Short term dependencies work fine
- Example: Predict next word in "the clouds are in the [*sky*]"



Contextual information nearby → can be encoded in hidden state easily

The Long-Term Dependency Problem with Basic RNNs

- It is harder to connect relevant past and present inputs for longer time spans
- Example: Predict next word in “I grew up in Germany … I speak fluent [*German*]”



Contextual information far away
Why does this make a difference?

The Long-Term Dependency Problem with Basic RNNs

Old acquaintances: vanishing and exploding gradients

- Layers and time steps of deep RNNs are related through multiplication
  → Gradients prone to vanishing or exploding
- **Exploding gradient** relatively easy to solve by clipping gradient
- **Vanishing gradient** harder to solve!

Additional problem: memory overwriting

- Hidden state is overwritten each time step
  → Detecting long-term dependencies even more difficult
- Can we do better? Again, yes!

Long Short-Term Memory Units (LSTMs)

LTSM Motivation

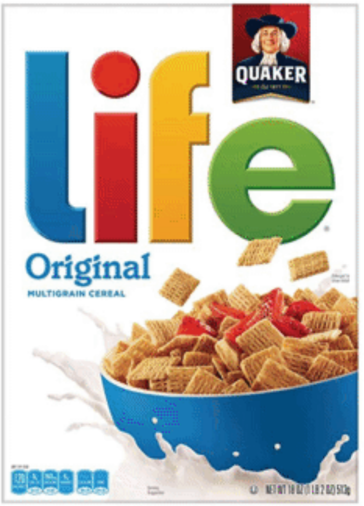
Customers Review 2,491

Thanos

September 2018

Verified Purchase

Amazing! This box of cereal gave me a perfectly balanced breakfast, as all things should be. I only ate half of it but will definitely be buying again!



A Box of Cereal
$3.99

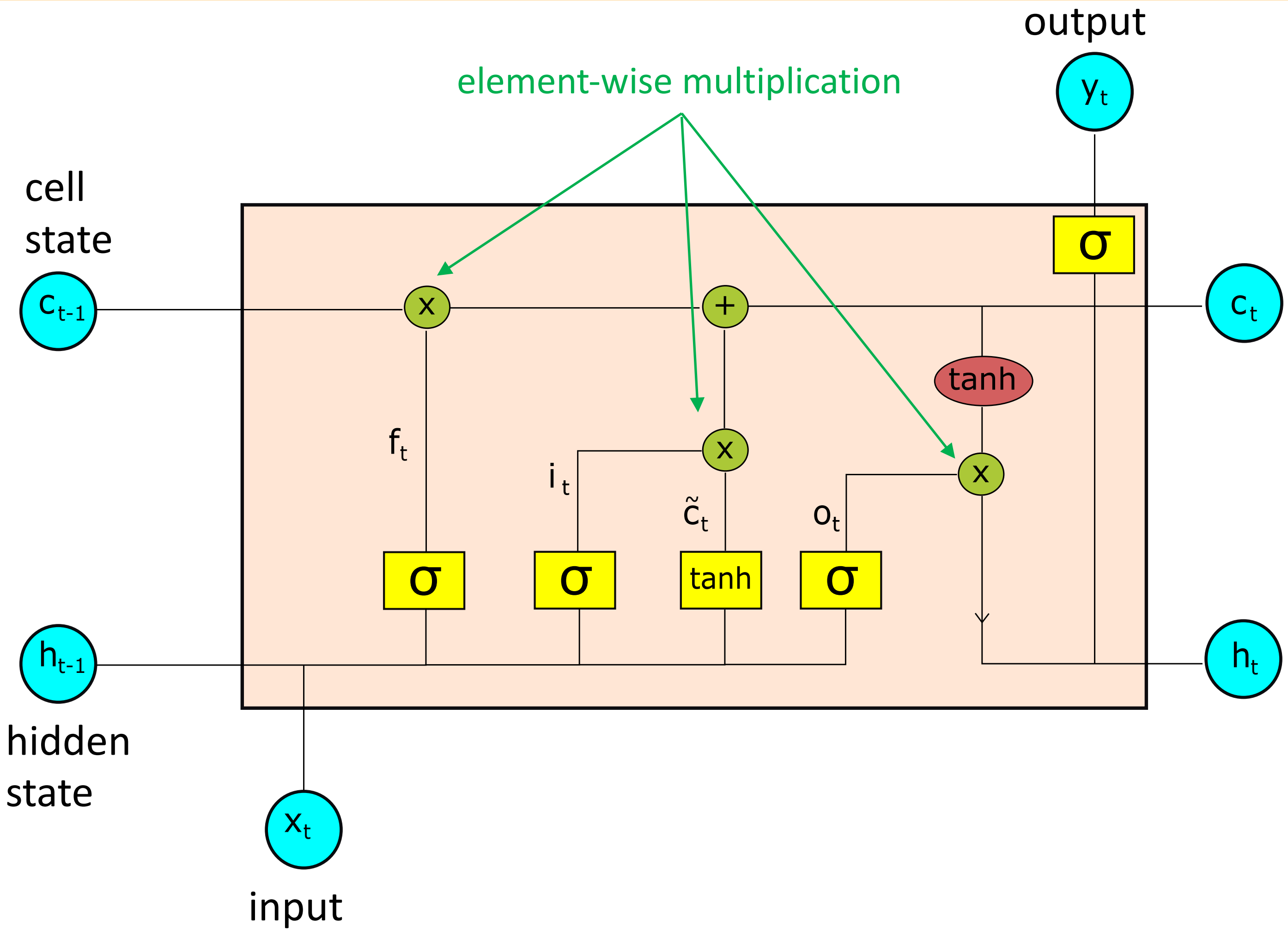
https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Background

Technische Hochschule Rosenheim

- **Long Short-Term Memory Units (LSTMs)** introduced by Sepp Hochreiter & Jürgen Schmidhuber in 1997 [Hoc97]
- Designed to solve vanishing gradient and learning long-term dependencies
- Main idea: introduction of **gates** that control writing and accessing "memory" in additional **cell state**
  - "forgetting" and "memorizing" information in separate steps

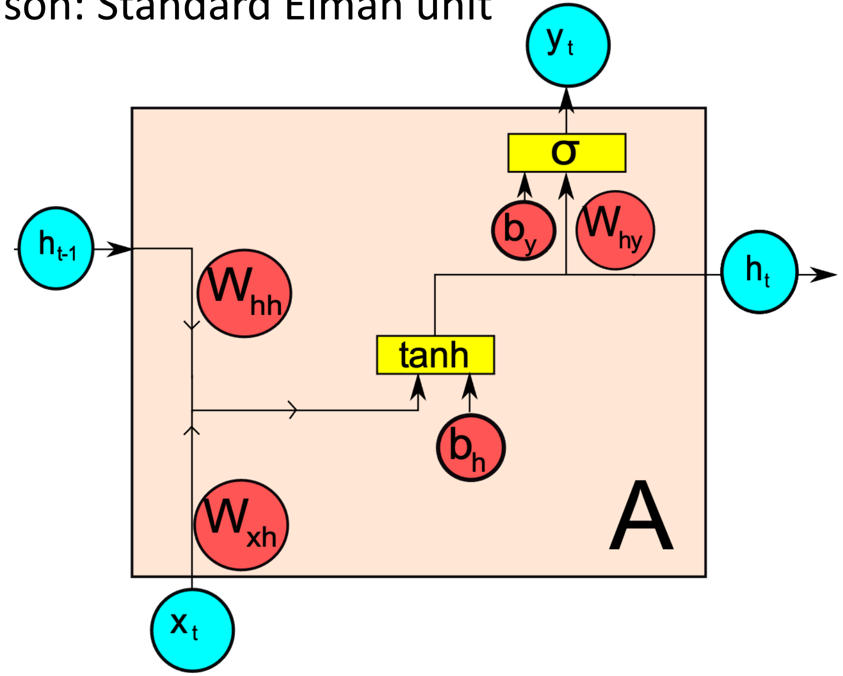
Overview LSTM Unit



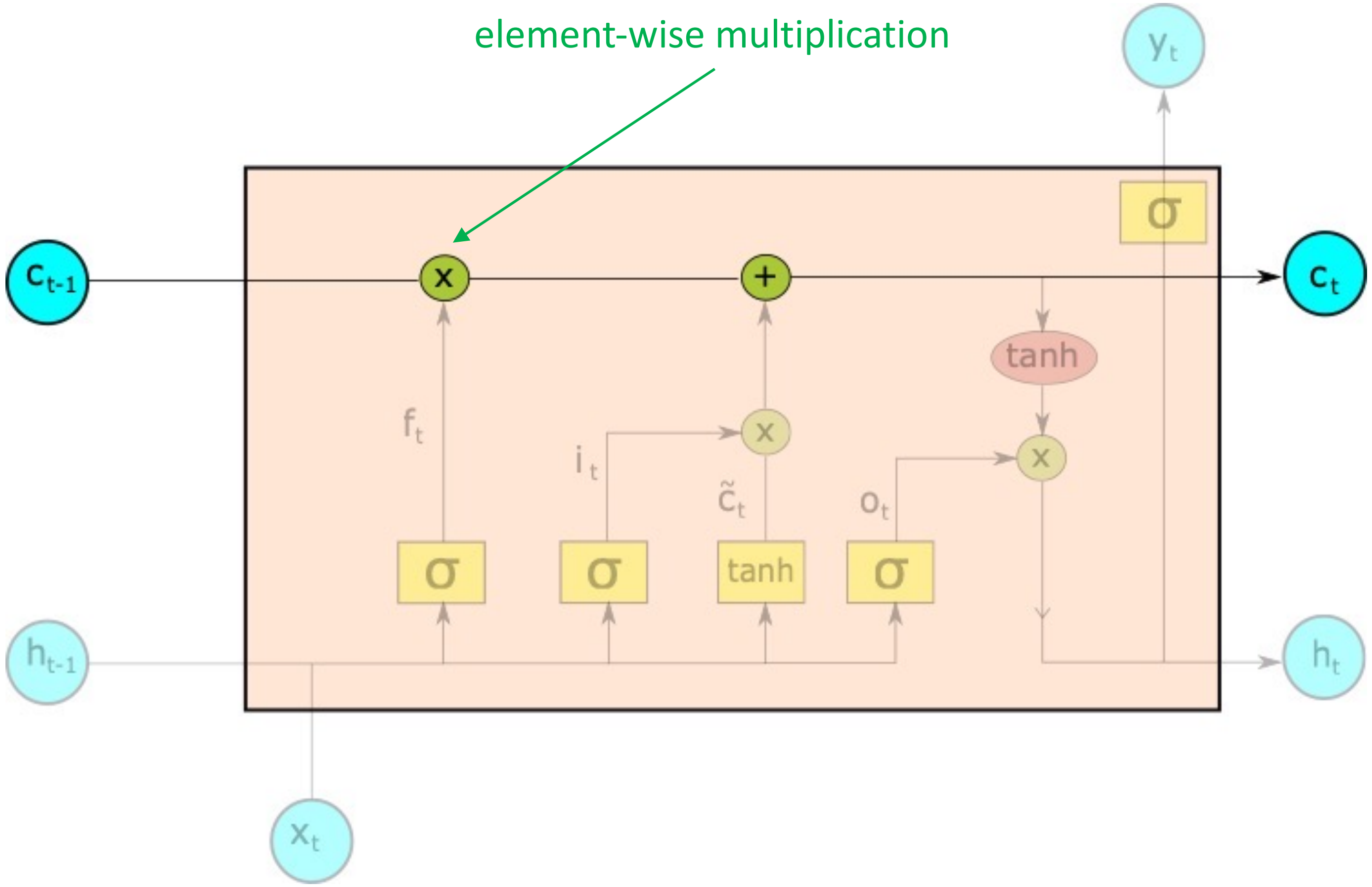
Update of internal states in multiple steps:

1) **Forget gate**: Forgetting old information in cell state
2) **Input gate**: Deciding on new input for cell state
3) Computing the updated cell state
4) Computing the updated hidden state

For comparison: Standard Elman unit



LSTM Cell State

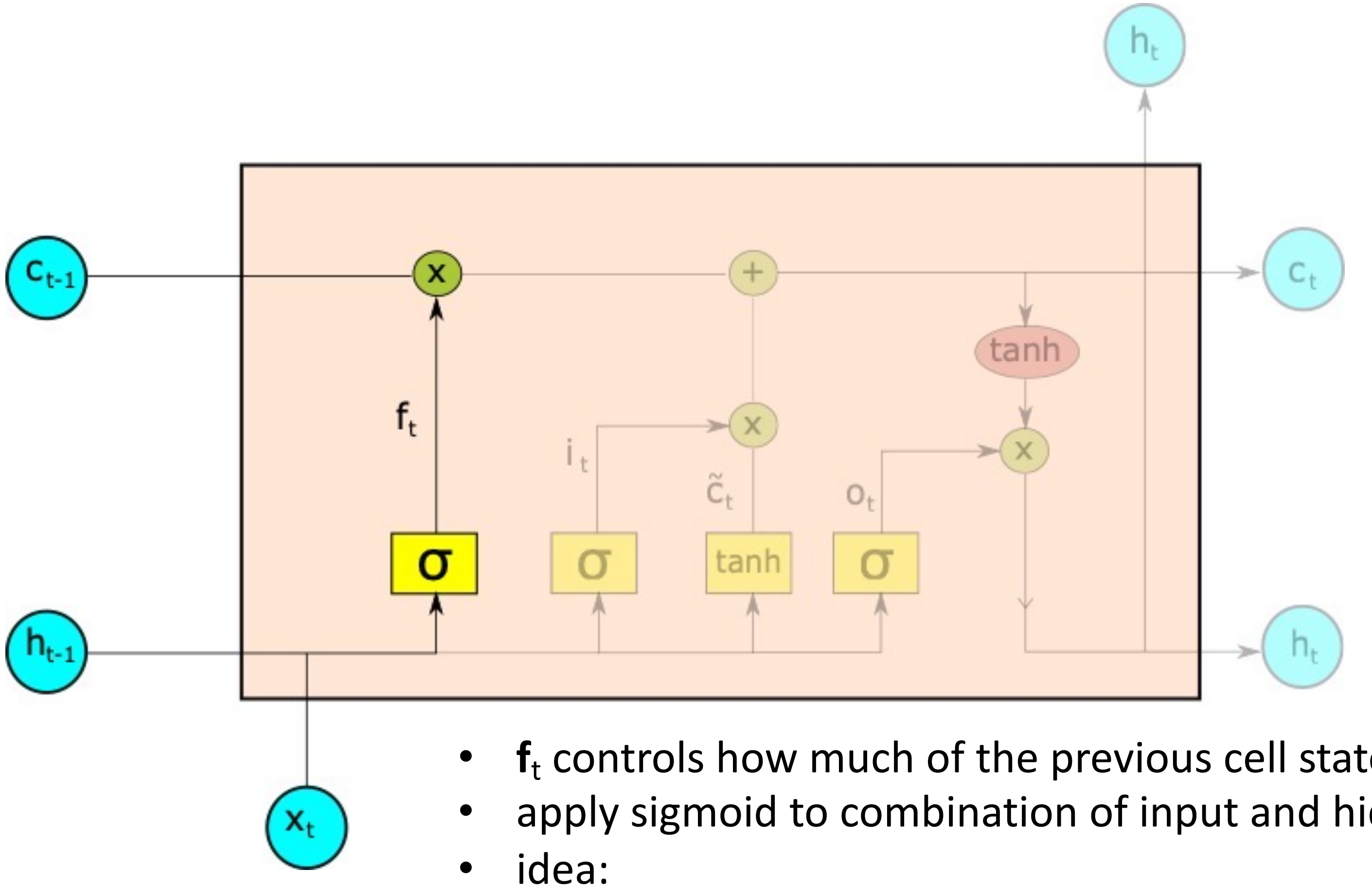


Cell state after time step t

- Undergoes only linear changes: no activation function!
- can flow through a unit unchanged → cell state can stay constant for multiple time steps
- this is the memory of the network

Forget Gate: Forgetting Old Information

Technische Hochschule Rosenheim

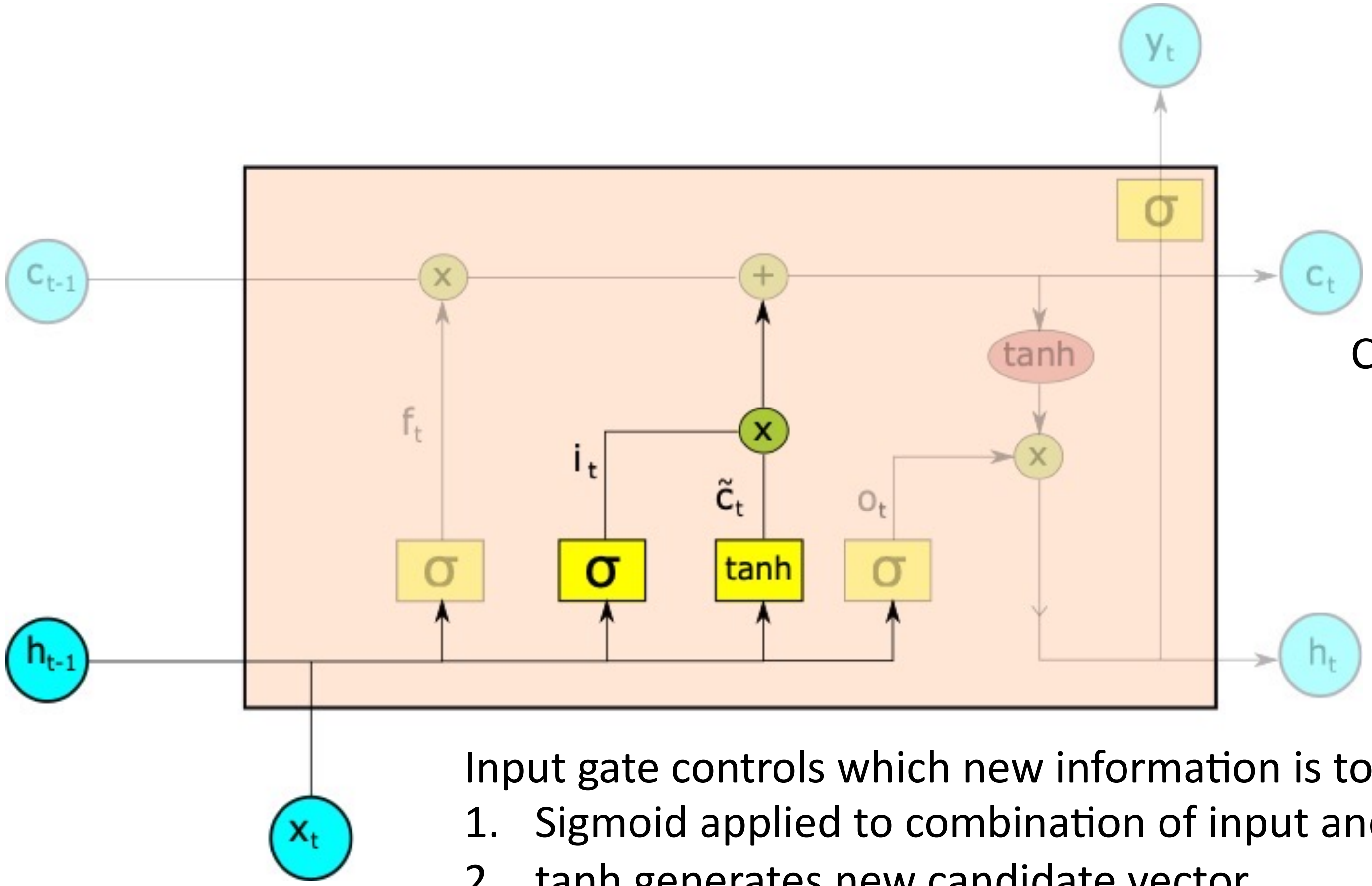


$$\mathbf{f}_t = \sigma\left(\boldsymbol{W}_{fh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{fx}\boldsymbol{x}_t + \boldsymbol{b}_f\right)$$

- $\mathbf{f}_t$ controls how much of the previous cell state is forgotten
- apply sigmoid to combination of input and hidden state (with weights and bias)
- idea:
  - Sigmoid output = 0 → forget
  - Sigmoid output = 1 → remember

Input Gate: Deciding on New Input

Technische Hochschule Rosenheim



Combination of input and hidden state on two paths:

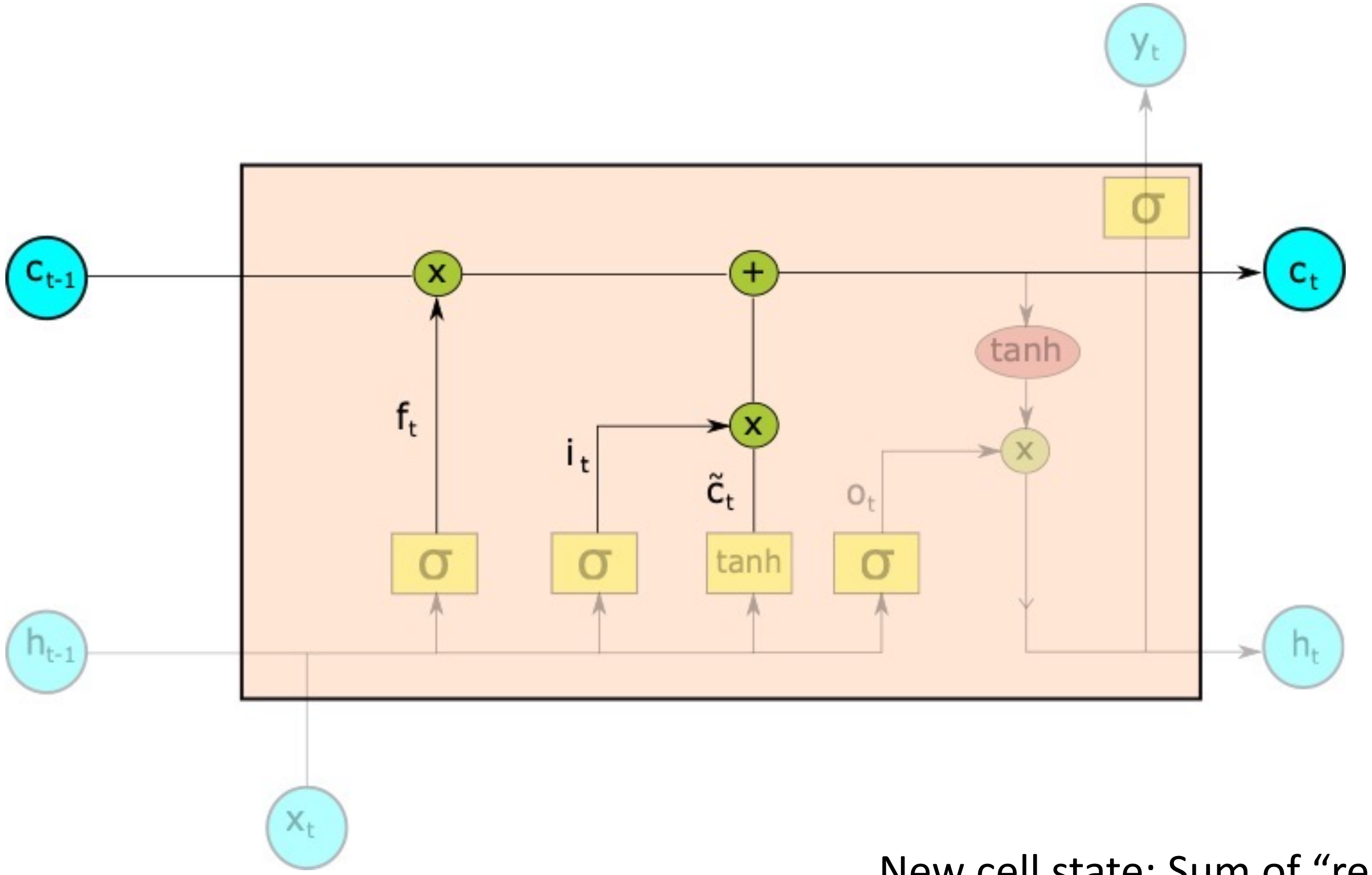
$$\mathbf{i}_t = \sigma(\boldsymbol{W}_{ih}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ix}\boldsymbol{x}_t + \boldsymbol{b}_i)$$

$$\widetilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}_{ch}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{cx}\boldsymbol{x}_t + \boldsymbol{b}_c)$$

Input gate controls which new information is to be stored in cell state

1. Sigmoid applied to combination of input and hidden state; decides which values are to be updated
2. tanh generates new candidate vector
3. The sigmoid decides which parts of the candidate are to be kept and added to the cell state

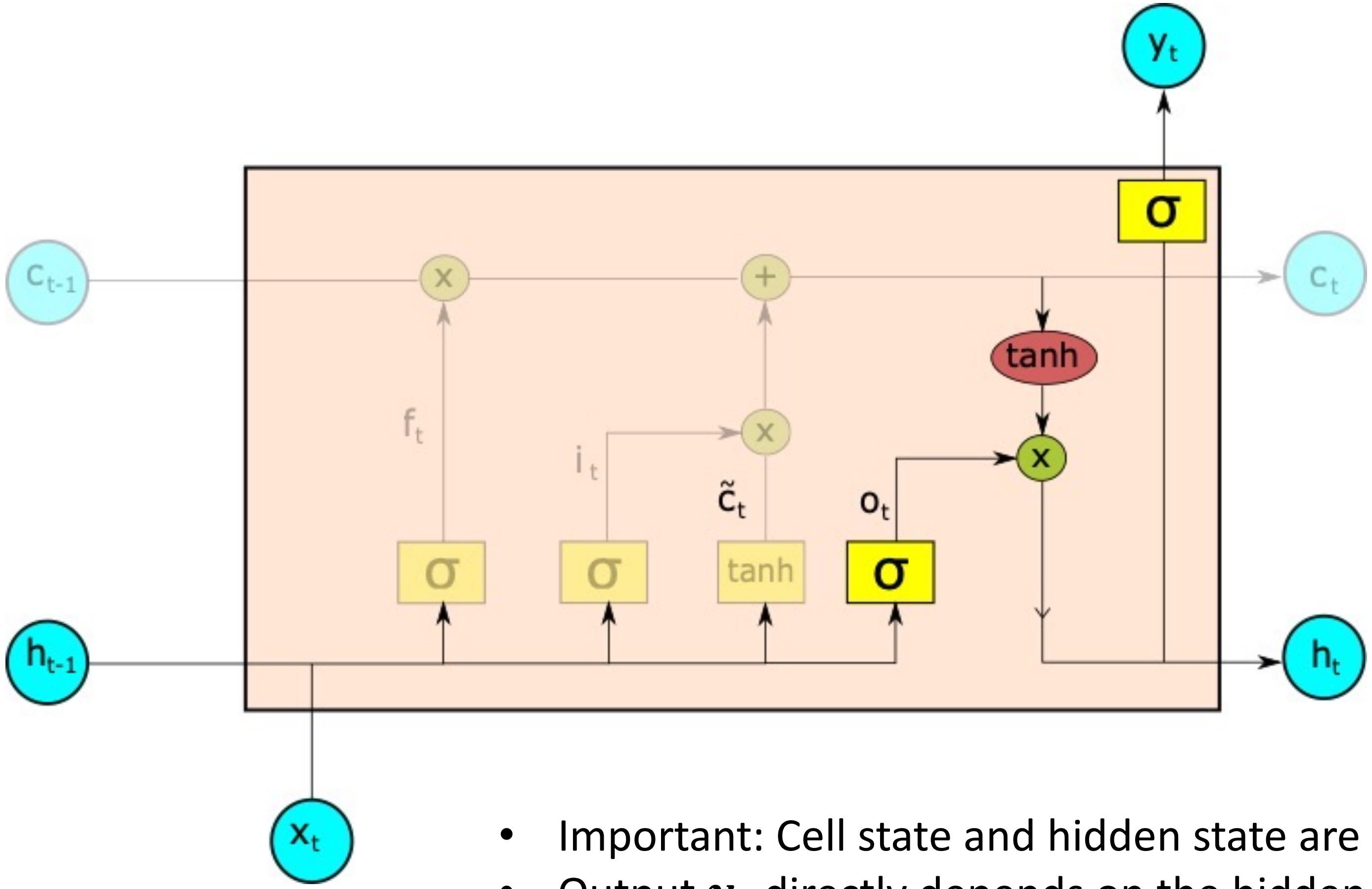
Updating the Cell State



$$\boldsymbol{c}_t = \mathbf{f}_t \odot \boldsymbol{c}_{t-1} + \mathbf{i}_t \odot \tilde{\boldsymbol{c}}_t$$

New cell state: Sum of "remaining information" from $\boldsymbol{c}_{t-1}$ and new information from input and hidden state ($\odot$: element-wise multiplication)

Updating the Hidden State and Computing the Output



$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_{oh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ox}\boldsymbol{x}_t + \boldsymbol{b}_o)$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh \boldsymbol{c}_t$$

$$\boldsymbol{y}_t = \sigma\left(\boldsymbol{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y\right)$$

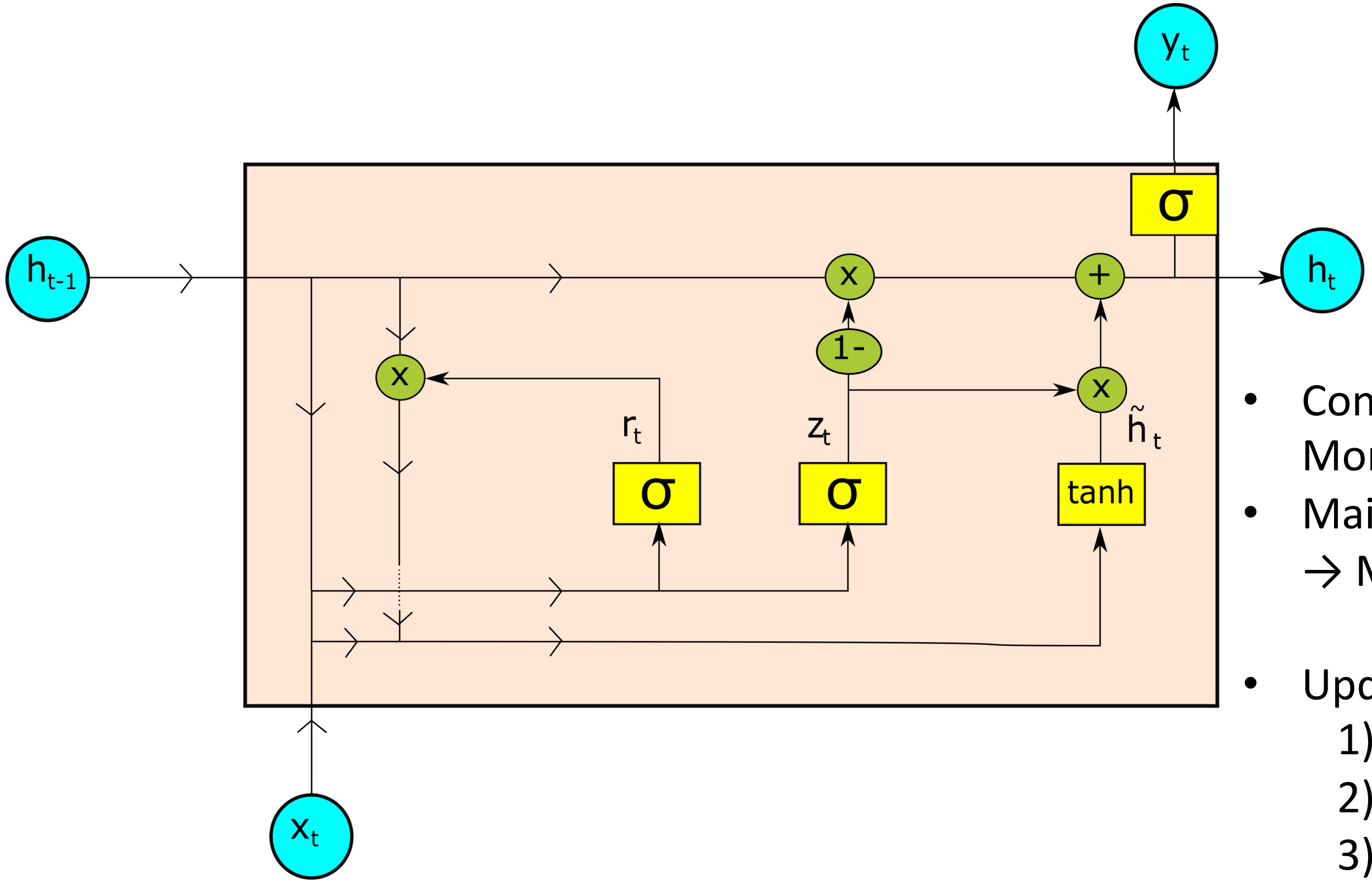
- Important: Cell state and hidden state are updated **separately**
- Output $\boldsymbol{y}_t$ directly depends on the hidden state $\boldsymbol{h}_t$

Gated Recurrent Units (GRUs)

Motivation

- LSTM great idea, but many parameters and difficult to train
  → Gated Recurrent Unit (GRU)
- Originally introduced by Cho et al. in 2014 for statistical machine translation [Cho14]
- Variant of the LSTM unit, but simpler and fewer parameters

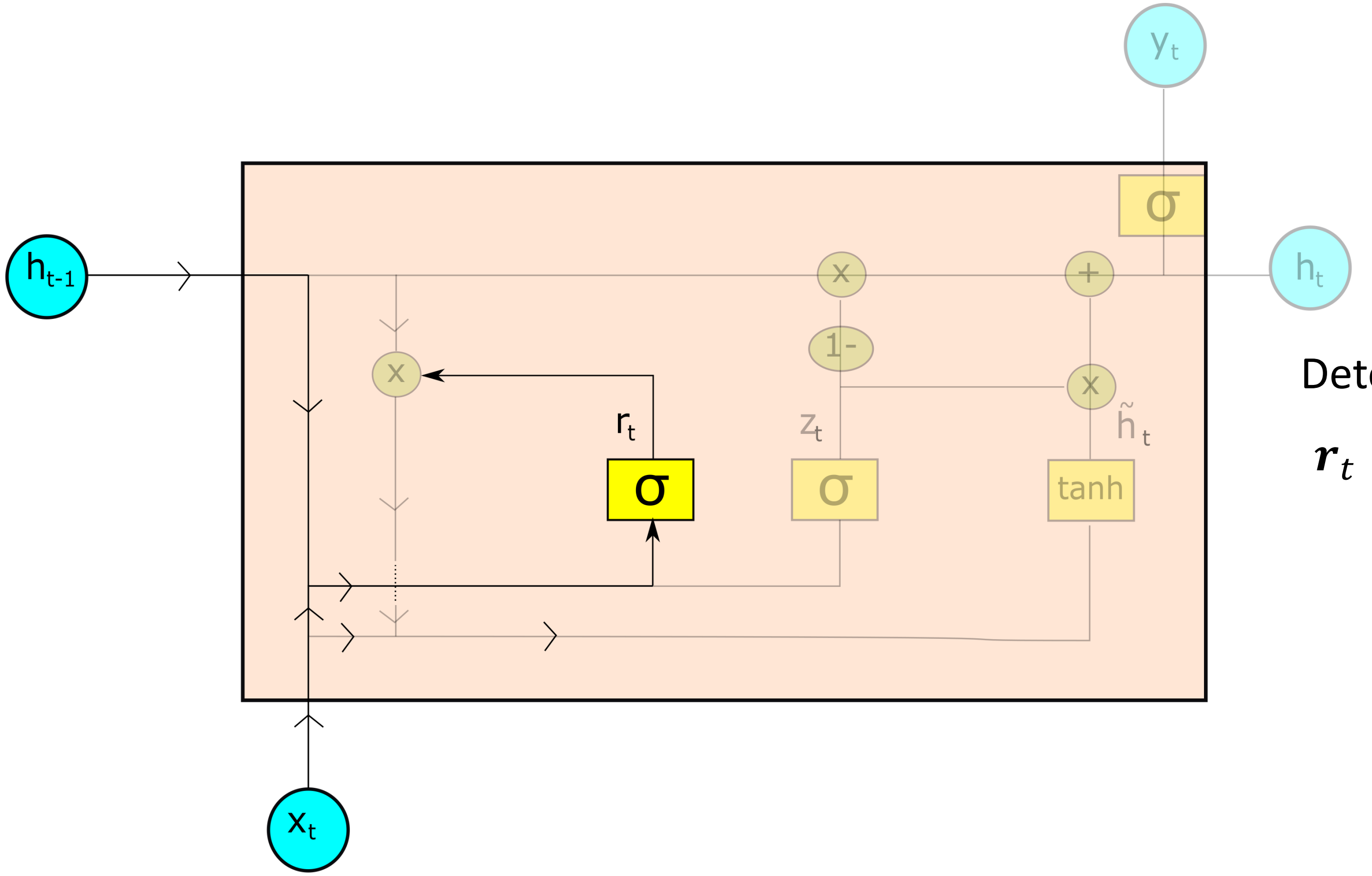
Structure of a GRU Cell



- Concept from LSTM:
  More control over hidden state/memory → gates
- Main difference: No additional cell state!
  → Memory operates only and directly via the hidden state

- Update of the hidden state can be divided into four steps:
  1) Reset gate: Influence of the previous hidden state
  2) Update gate: Influence of a newly computed update
  3) Propose an updated hidden state
  4) Compute updated hidden state

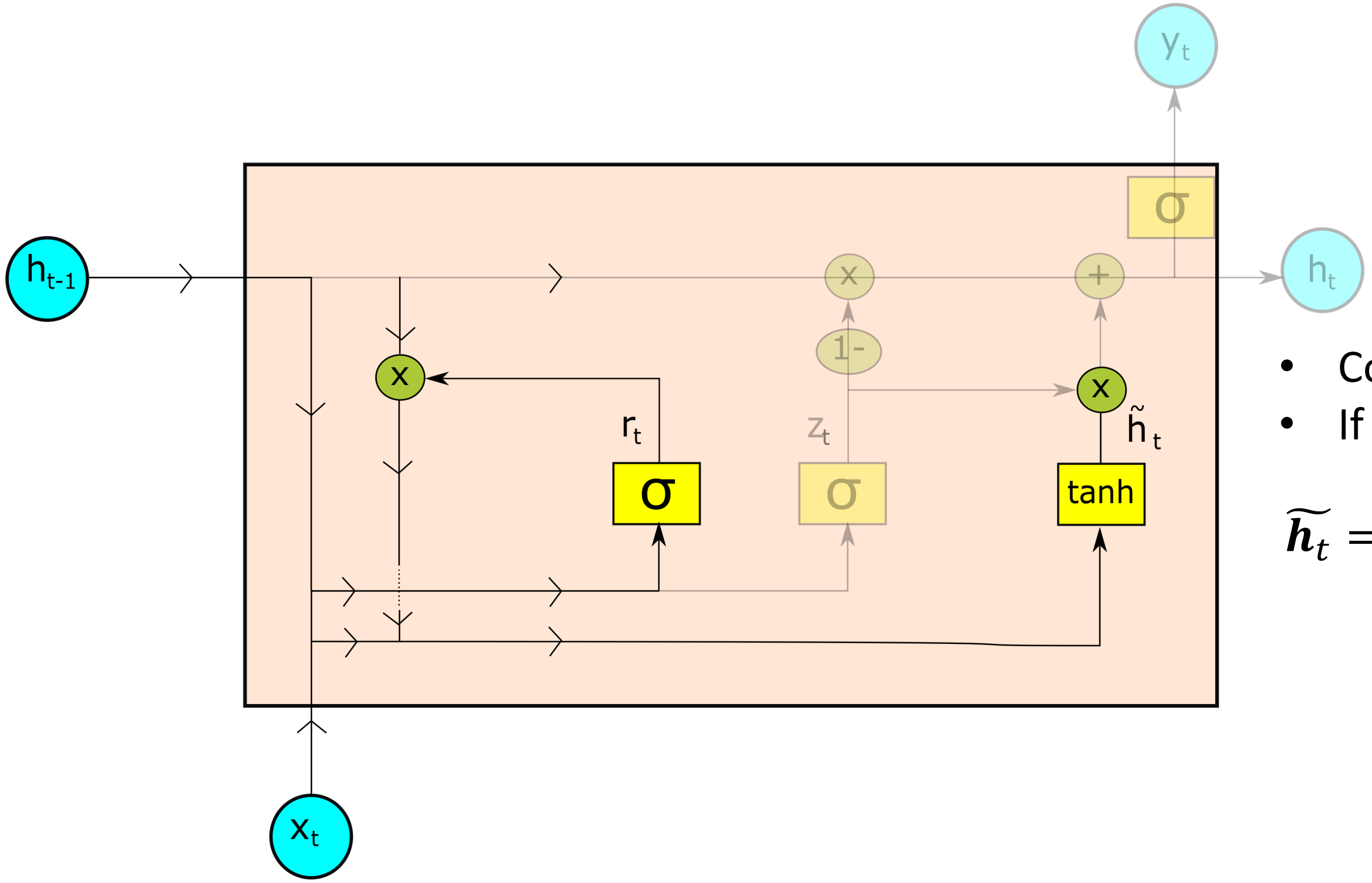
Reset Gate



Determines the influence of the previous hidden state

$$\boldsymbol{r}_t = \sigma(\boldsymbol{W}_{rh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{rx}\boldsymbol{x}_t + \boldsymbol{b}_r)$$

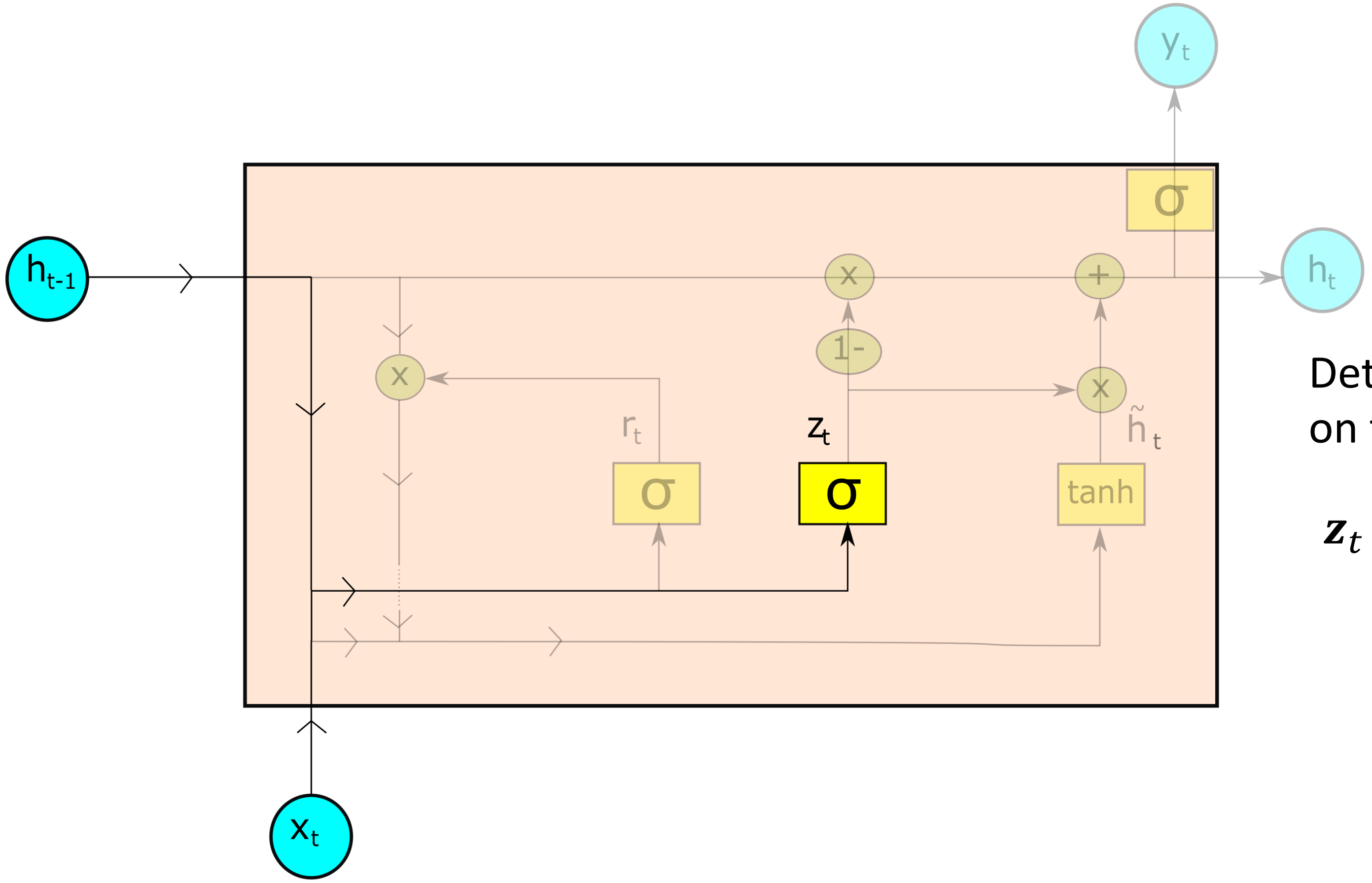
Proposing an Update



- Combination of input and “reset” hidden state.
- If $\boldsymbol{r}_t$ is close to 0 → low influence of previous hidden state

$$\widetilde{\boldsymbol{h}_t} = \tanh(\boldsymbol{W}_{hh}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{W}_{hx}\boldsymbol{x}_t + \boldsymbol{b}_h)$$

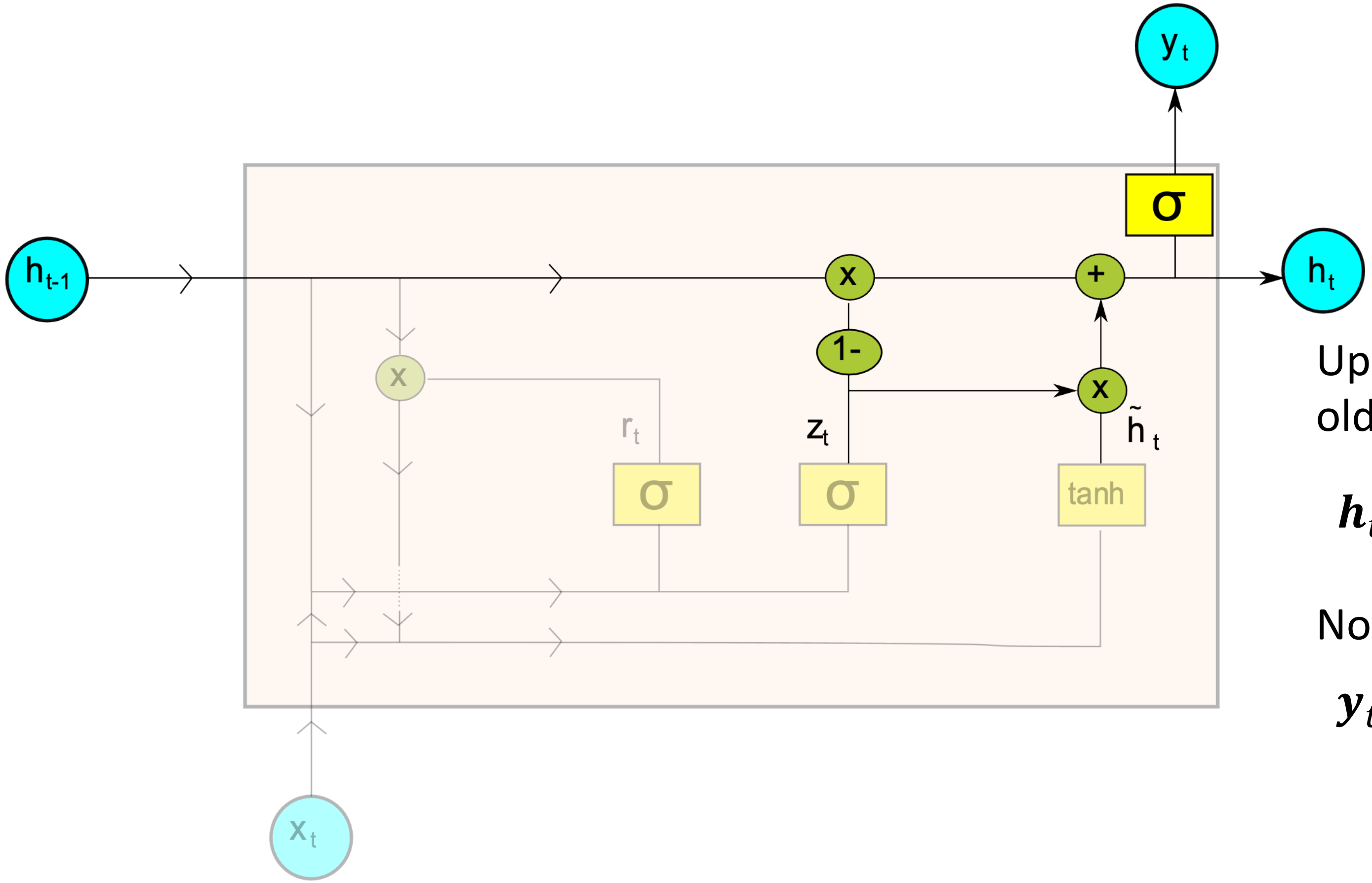
Update Gate



Determines the influence of an “update proposal” on the new hidden state

$$\boldsymbol{z}_t = \sigma(\boldsymbol{W}_{zh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{zx}\boldsymbol{x}_t + \boldsymbol{b}_z)$$

Finally: Compute the Updated Hidden State



Update gate controls combination of old state and proposed update

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \widetilde{\boldsymbol{h}_t}$$

Node output:

$$\boldsymbol{y}_t = \sigma\left(\boldsymbol{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y\right)$$

Remarks

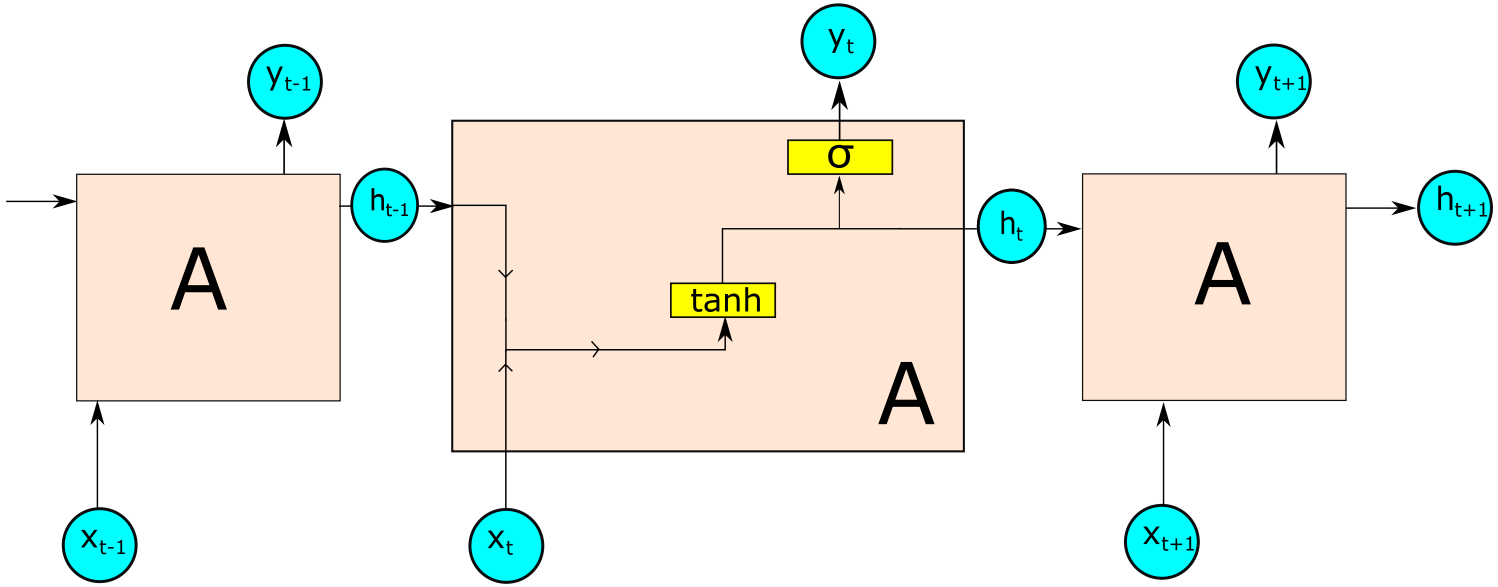
- Add (“+”) essential for preservation of error in backpropagation
  - resolves vanishing gradient problem (which originates from multiplication)
- Gates allow capturing diverse time scales and remote dependencies
- Units learning **short-term** dependencies have restrictive reset gates
  → $\boldsymbol{r}_t$ close to 0: ignore previous hidden state
- Units learning **long-term** dependencies have restrictive update gates
  → $\boldsymbol{z}_t$ close to 0: ignore new input

→ Gates have varying “rhythm” depending on the type of information

Comparison

Recap: Simple RNNs

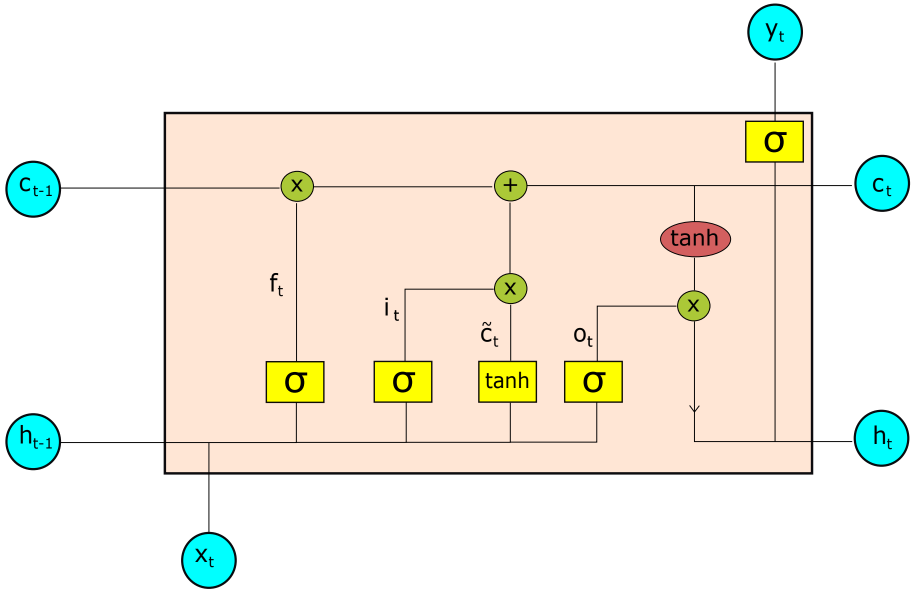
- Gradient-based training difficult (vanishing/exploding gradients)
- Short-term dependencies hide long-term dependencies due to exponentially small gradients
- Hidden state is overwritten in each time step

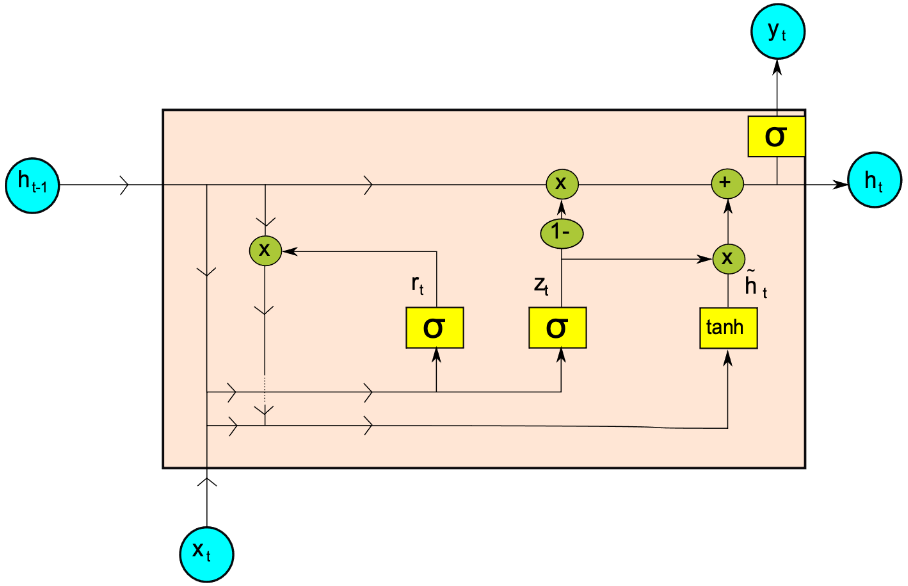


Recap: Advanced structures – LSTM and GRU

Both

- introduce gates that control information flow and operate on memory
- can capture dependencies at different time scales
- additive calculation of state preserves error during backpropagation
  → more efficient training possible





Recap: LSTM and GRU – Differences

LSTM	GRU
Separate hidden and cell state	Combined hidden and cell state
Controlled exposure of memory content through output gate	Full exposure of memory content without control
Independent input and forget gate: $\boldsymbol{c}_t = \mathbf{f}_t \odot \boldsymbol{c}_{t-1} + \mathbf{i}_t \odot \widetilde{\boldsymbol{c}_t}$	Joint update gate: $\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \widetilde{\boldsymbol{h}_t}$

Comparison of Recurrent Units: What Should We Use?

- LSTM and GRU outperform simple RNN units [Agg18]
- LSTM vs GRU inconclusive
  - roughly similar in performance
  - GRU is simpler and more efficient
  - GRU might generalize slightly better with less data due to less parameters
  - LSTM might be preferable with more data

Application Examples

Character-based Language Modeling with RNNs

- Great blog post by Andrej Karpathy: http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- Character-level RNN for text generation trained on Shakespeare
- Example for generated text:

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.
```

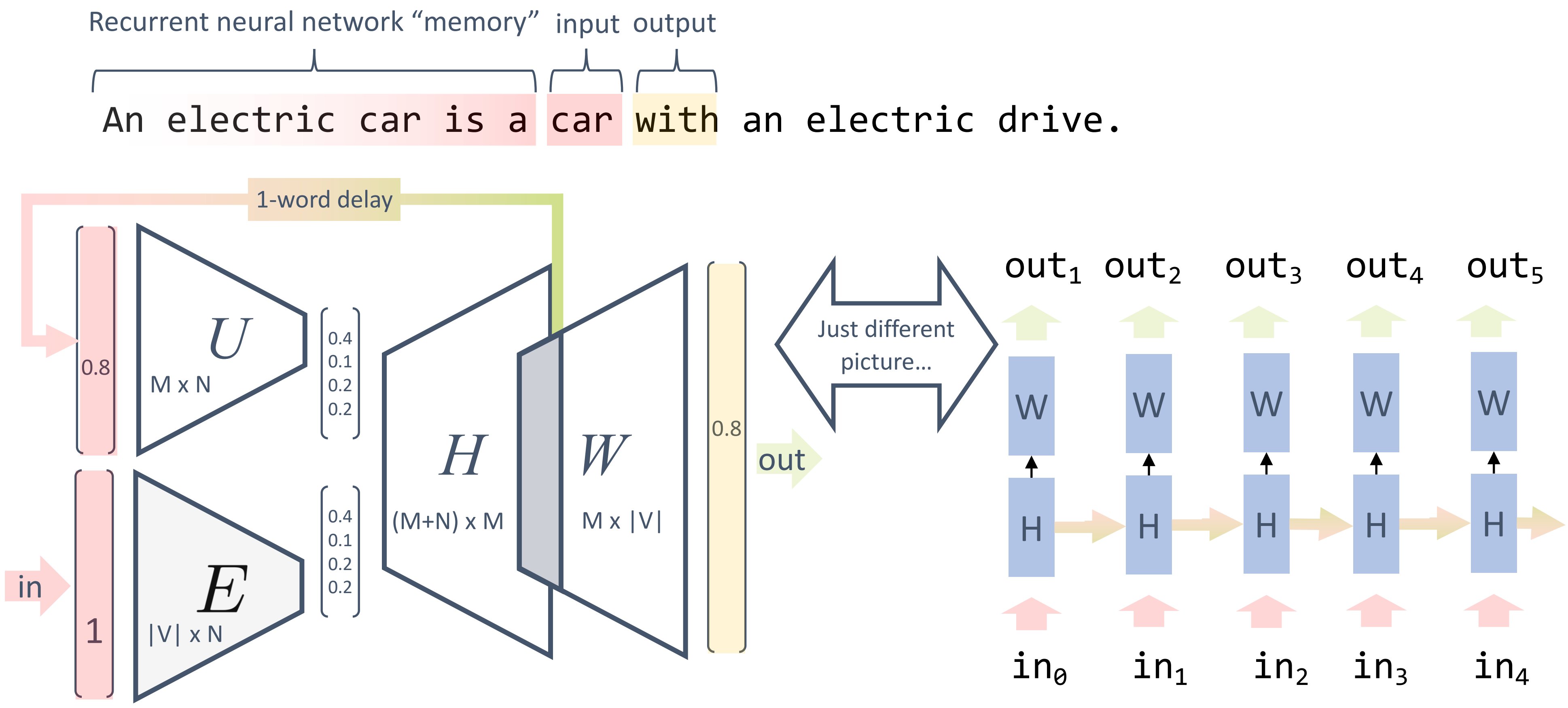
```
Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

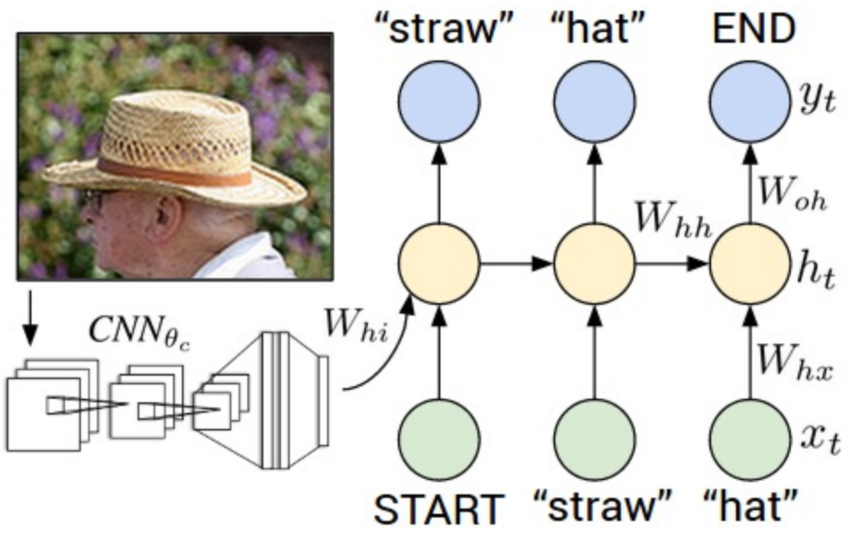
- Experiments with LATEX, Linux code and Wikipedia entries in the blog post

Language Modeling – How Probable is the Next Word?



based on silde courtesy of Tobias Bocklet [Boc20]

Image Captioning

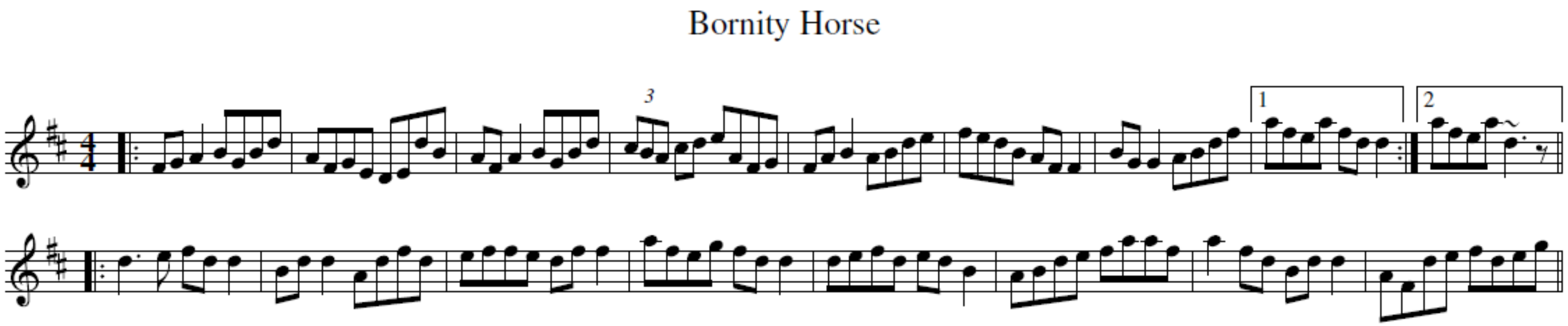


- Encoder:
  - encode image using CNN
  - similar to transfer-learning based on Imagenet trained network using the second to last layer
- Decoder:
  - RNN models language
  - START and END as special tokens

Andrej Karpathy Li Fei-Fei, Deep Visual-Semantic Alignments for Generating Image Descriptions, CVPR 2015
https://arxiv.org/abs/1412.2306

Composing Folk Music

- Music composition tackled frequently with RNNs (e.g. 1989 by Todd [Tod89], 2002 by Eck and Schmidhuber [Eck02], ...)
- Sturm and Ben-Tal [Stu15] use bigger/deeper networks to generate Folk music
- Character-level RNN using ABC format, including generating title
- Example:



- Audio examples online, e.g., https://themachinefolksession.org/tune/31

Concluding Remarks

Summary

- Recurrent neural networks are able to directly model sequential algorithms
- Training via (truncated) backpropagation through time
- Simple units suffer extremely from exploding/vanishing gradients
- LSTM & GRU as improved RNN units that explicitly model “forgetting” and “remembering”
- Many more architectures we have not discussed
  - Memory Networks [Wes14, Suk15]
  - Neural Turing Machines [Gra14]
  - Neural GPUs [Kai16]
  - Transformer [Vas17]

Computational Power

- Recurrent networks are **Turing complete**!
  - not using the simple structures presented here, but with feedback loops in the network
  - proof by Siegelmann and Sontag 1992 [Sie92]
  - 886 neurons with rational weights are sufficient to build a universal neural network [Sie95]
    - this is a conservative estimate; there are probably smaller universal networks
    - this does not mean that we have an algorithm for training
- Neural Turing Machines, Neural GPUs, and Transformers can also be shown to be Turing complete [Per19]

- Simple Elman-Unit RNNs have at least the power of deterministic finite automata (DFAs) [Sie96]
- Allowing infinite precision, [Kor19]
  - RNNs with just one hidden layer and ReLU activation are at least as powerful as pushdown automata (PDAs)
  - GRUs are at least as powerful as DFAs

Links

RNN Folk Music
FolkRNN.org
MachineFolkSession.com
The Glass Herry Comment 14128

Further Reading
Character RNNs
CNNs for Machine Translation
Composing Music with RNNs

References

[Agg18] C. Aggarwal. Neural Networks and Deep Learning, Springer 2018

[Boc20] T. Bocklet: Deep Learning Slides Winter Semester 2020/21. Technische Hochschule Nürnberg.

[Cho14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: arXiv preprint arXiv:1406.1078 (2014).

[Eck02] Douglas Eck and Jürgen Schmidhuber. “Learning the Long-Term Structure of the Blues”. In: Artificial Neural Networks — ICANN 2002. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 284–289.

[Elm90] Jeffrey L Elman. “Finding structure in time”. In: Cognitive science 14.2 (1990), pp. 179–211.

[Gra14] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural Turing Machines”. In: CoRR abs/1410.5401 (2014). arXiv: 1410.5401.

[Hoc97] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: Neural computation 9.8 (1997), pp. 1735–1780.

[Hop82] J. J. Hopfield. “Neural networks and physical systems with emergent collective computational abilities”. In: Proceedings of the National Academy of Sciences 79.8 (1982), pp. 2554–2558. eprint: http://www.pnas.org/content/79/8/2554.full.pdf.

[Kai16] L. Kaiser und I. Sutskever. Neural GPUs Learn Algorithms. In Y. Bengio und Y. LeCun, Hg., 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings. 2016.

[Kor19] S.A. Korsky and R.C. Berwick. On the Computational Power of RNNs. CoRR abs/1906.06349 (2019)

[Lit74] W.A. Little. “The existence of persistent states in the brain”. In: Mathematical Biosciences 19.1 (1974), pp. 101–120.

[Per19] J. Pérez, J. Marinkovic und P. Barceló. On the Turing Completeness of Modern Neural Network Architectures. In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019.

[Sie92] H. T. Siegelmann and E. D. Sontag. On the Computational Power of Neural Nets. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT ’92, S. 440–449. Association for Computing Machinery, New York, NY, USA, 1992.

[Sie95] H. T. Siegelmann und E. D. Sontag. On the Computational Power of Neural Nets. *J. Comput. Syst. Sci.*, 50(1):132–150, Febr. 1995.

[Sie96] H. T. Siegelmann. 1996. Recurrent neural networks and finite automata. Computational Intelligence, 12:567–574.

[Stu15] Bob Sturm, João Felipe Santos, and Iryna Korshunova. “Folk music style modelling by recurrent neural networks with long short term memory units”. In: 16th International Society for Music Information Retrieval Conference, Malaga, Spain, 2015, p. 2.

[Suk15] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, et al. “End-to-End Memory Networks”. In: CoRR abs/1503.08895 (2015). arXiv: 1503.08895.

[Tod89] Peter M. Todd. “A Connectionist Approach to Algorithmic Composition”. In: Computer Music Journal 13 (Dec. 1989).

[Vas17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In NIPS, pp. 5998–6008, 2017.

[Wes14] Jason Weston, Sumit Chopra, and Antoine Bordes. “Memory Networks”. In: CoRR abs/1410.3916 (2014). arXiv: 1410.3916.