



FRIEDRICH-ALEXANDER-
UNIVERSITÄT
ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Recurrent Neural Networks

K. Breininger, V. Christlein, Z. Yang, L. Rist, M. Nau, S. Jaganathan, C. Liu, N. Maul, L. Folle, M. Zinnen,
K. Packhäuser

Pattern Recognition Lab, Friedrich-Alexander University of Erlangen-Nürnberg

January 4, 2023



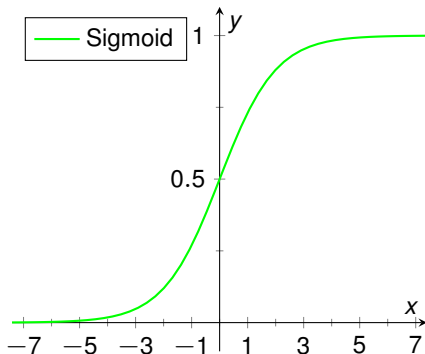


FRIEDRICH-ALEXANDER-
UNIVERSITÄT
ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Activation Functions



Sigmoid Activation Function



Sigmoid (logistic function)

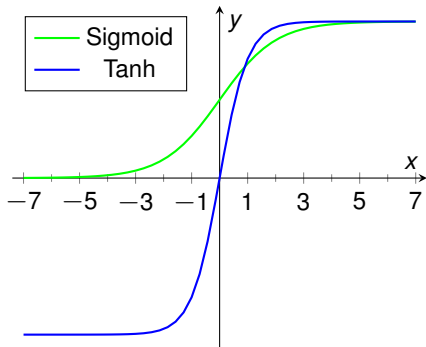
$$f(x) = \frac{1}{1 + \exp(-x)}$$

$$f'(x) = f(x)(1 - f(x))$$

for derivative we only need to store the output in the forward pass. but still you should consider error tensor

→ Observe that the derivative can be solely expressed in terms of the activation!

Tanh Activation Function



Tanh

$$f(x) = \tanh(x)$$

$$f'(x) = 1 - f(x)^2$$

→ The derivative is still a function of the activation!



FRIEDRICH-ALEXANDER-
UNIVERSITÄT
ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Elman Recurrent Neural Network



General strategy

- We interpret the **batch** dimension as **time** dimension now

General strategy

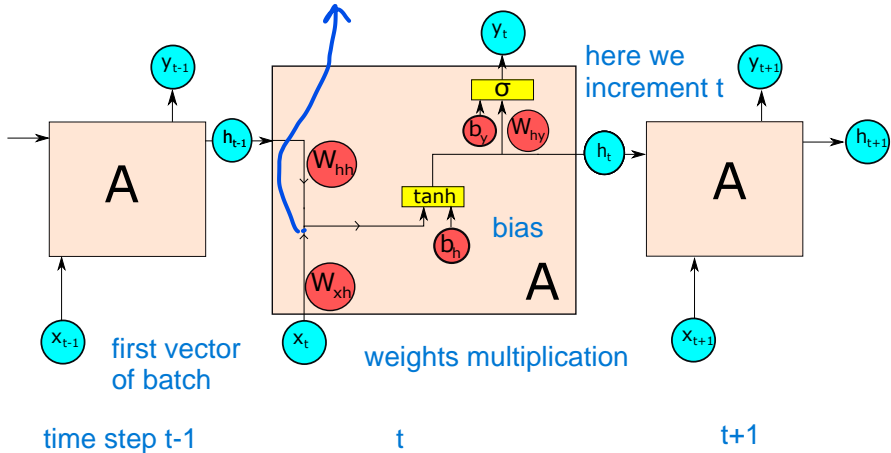
- We interpret the **batch** dimension as **time** dimension now
- Samples are correlated in this dimension

General strategy

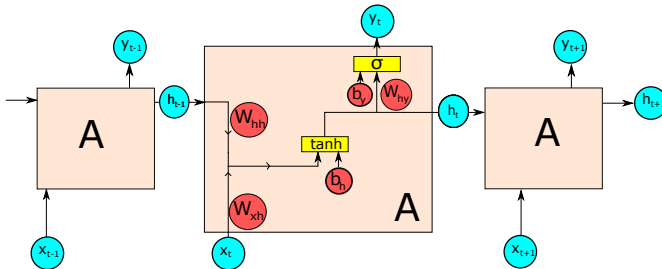
- We interpret the **batch** dimension as **time dimension** now
- Samples are **correlated** in this dimension **they become dependent**
- This allows to **reuse** loss functions, optimizers, initializers, activation functions and the Neural Network class

hidden state

transformed to sigmoid layer



Elman RNN Cell



Output formula:

$$\mathbf{y}_t = \sigma(\mathbf{h}_t \cdot \mathbf{W}_{hy} + \mathbf{b}_y)$$

\mathbf{W}_{hy} : Weight matrix for current hidden state \mathbf{h}_t

\mathbf{b}_y : Output bias

A word on software engineering

- In terms of **encapsulation** - how good was the idea to demand exposition of the weights as member?

A word on software engineering

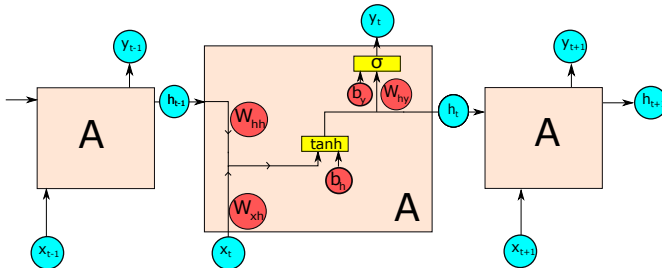
- In terms of **encapsulation** - how good was the idea to demand exposition of the weights as member?
- Suppose we implement the RNN cell as **composite** structure
- **Getters** and **Setters** provide us the flexibility to do so

we need to have access to
intermediate values (setters and
getters)

A word on software engineering

- In terms of **encapsulation** - how good was the idea to demand exposition of the weights as member?
- Suppose we implement the RNN cell as **composite structure**
- **Getters** and **Setters** provide us the flexibility to do so
- Takeaway? Not doing **proper software engineering** most of the time will demand a price at some point.

Elman RNN Cell



hidden layer input

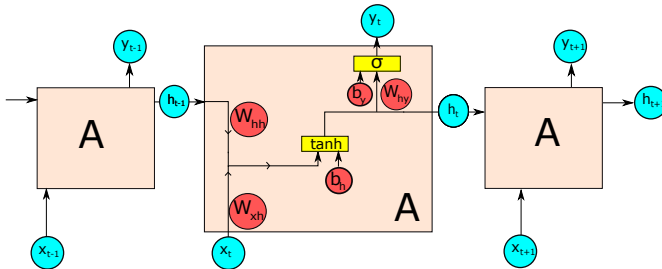
$$\mathbf{h}_t = \tanh(\mathbf{h}_{t-1} \cdot \mathbf{W}_{hh} + \mathbf{x}_t \cdot \mathbf{W}_{xh} + \mathbf{b}_h)$$

\mathbf{W}_{hh} : Weight matrix for previous hidden state \mathbf{h}_{t-1}

\mathbf{W}_{xh} : Weight matrix for current input \mathbf{x}_t

\mathbf{b}_h : Update bias

Elman RNN Cell



$$\mathbf{h}_t = \tanh(\tilde{\mathbf{x}}_t \cdot \mathbf{W}_h)$$

\mathbf{W}_h : Weight matrix of a fully connected layer easier to implement

$\tilde{\mathbf{x}}_t$: Concatenation of \mathbf{x}_t , \mathbf{h}_{t-1} and a 1 1 is bias

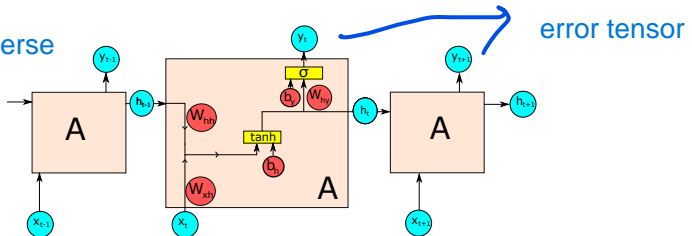
Different from output: Not processed independently!

expect that we don't have output but
error tensor

Backward

how do we compute gradients?

we should inverse
everything

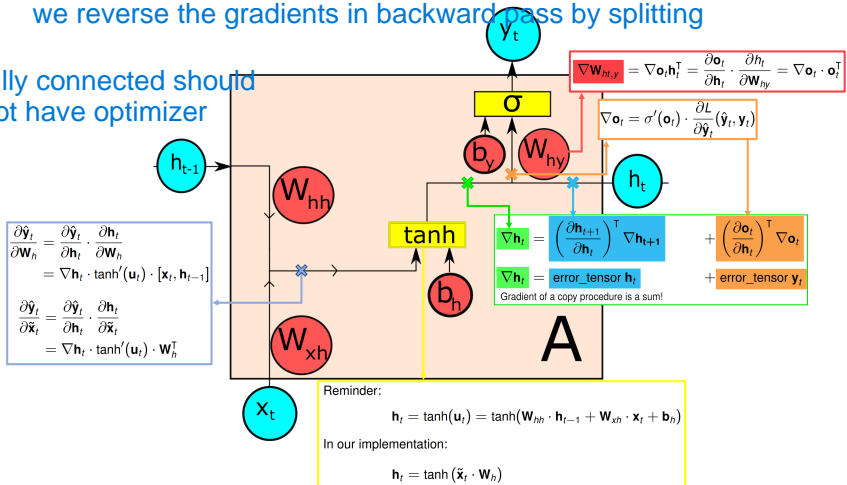


- Most gradients are handled by the **embedded layers**
- **Store and feed the values for backprop (input tensors, activations) externally** to the embedded layers because of **multiple forward calls**
- We need gradients through **summation, multiplication and copying**

Elman RNN Cell Backward

we reverse the gradients in backward pass by splitting

fully connected should
not have optimizer



when i forwad in the elman we should save the intermediate results

Backward how to backpropagate the copy procedure

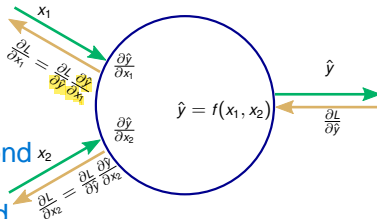
chain rule and
back propagation

because the second
part is one
we should forward
only the first part

Sum

Multiply

Copy



$$f(x_1, x_2) = x_1 + x_2$$

$$\frac{\partial \hat{y}}{\partial x_1} = 1$$

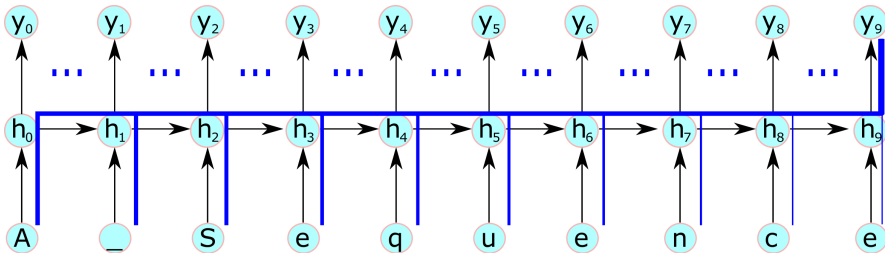
Gradient is **copying** $\frac{\partial L}{\partial \hat{y}}$

$$f(x_1, x_2) = x_1 \cdot x_2$$

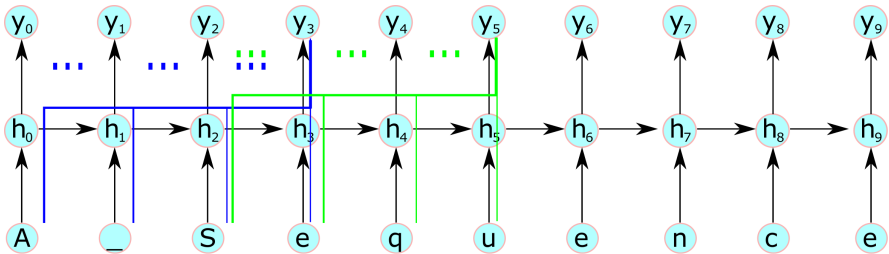
$$\frac{\partial \hat{y}}{\partial x_1} = x_2$$

Gradient is \cdot with **switched**
inputs

Backward pass of sum
So the gradient is a sum!



- Implemented by passing the whole sequence as a **batch**



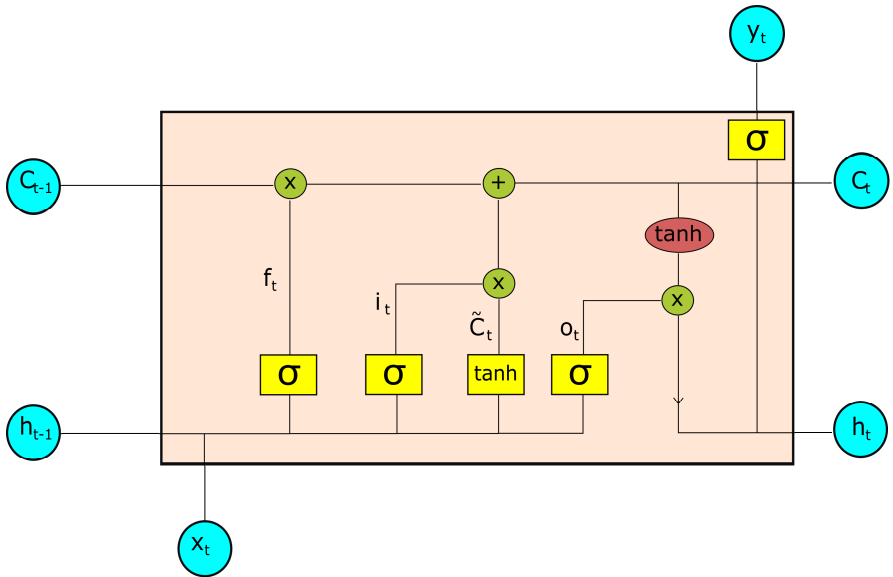
- Implemented by passing **overlapping** parts as a **batch**
- We need to implement memory **between states**
- Simply store the **last hidden state** and implement a **method** switching whether this state is reused in subsequent forward passes.
- Data has to be fed in **accordingly!**
- Referencing the TBPPT Algorithm presented in the lecture: k_1 is always the sequence length and k_2 is always the TBPTT length.



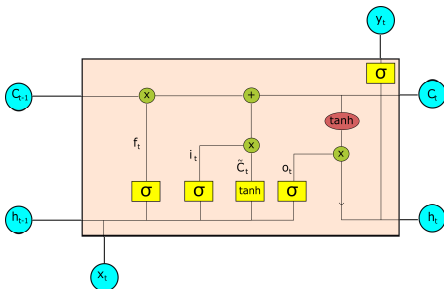
FRIEDRICH-ALEXANDER-
UNIVERSITÄT
ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Long Short-Term Memory (optional)



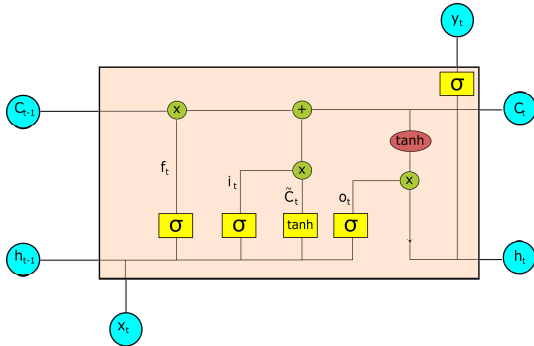


Forward



- We can reuse a **fully connected** layer again for the **output**
- Unlike in the lecture, the output gate is a trainable fully connected layer to allow different output sizes
- The **concatenation** is also **analogous** to the RNN
- The gates σ and the yellow tanh can be a single **fully connected** layer with an output size of $4 \cdot \text{dim}(\text{hidden state})$

Backward



- Remember that we have to pass the vectors of the input tensor **sequentially**
- Most gradients are again handled by the **embedded layers**
- Again **store and feed the values for backprop externally** to the embedded layers

Thanks for listening.
Any questions?