Recurrent Neural Networks

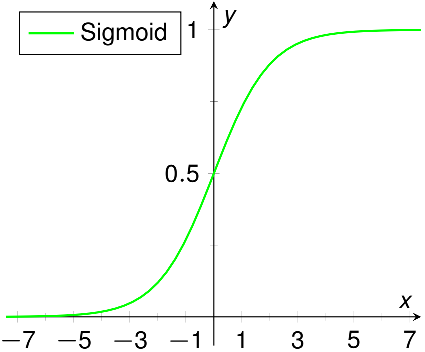
Leonid Mill, Tobias Würfl, Katharina Breininger, Nishant Ravikumar, Sebastian Gündel, Mathis Hoffmann, Florian Thamm, Felix Denzinger
Pattern Recognition Lab, Friedrich-Alexander University of Erlangen-Nürnberg
January 18, 2019

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Activation Functions

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Sigmoid Activation Function



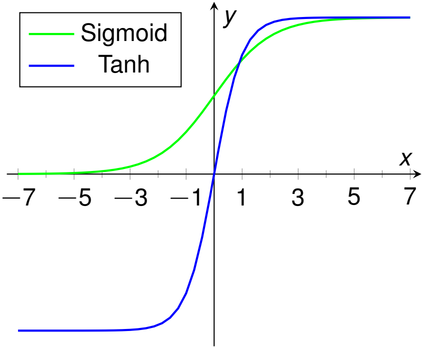
Sigmoid (logistic function)

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

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

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

Tanh Activation Function



Tanh

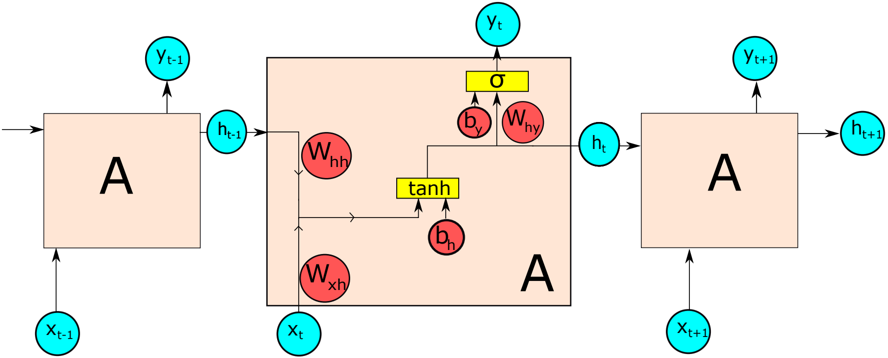
$$f(x) = tanh(x)$$
$$f'(x) = 1 - f(x)^2$$

➔ The derivative is still a function of the activation!

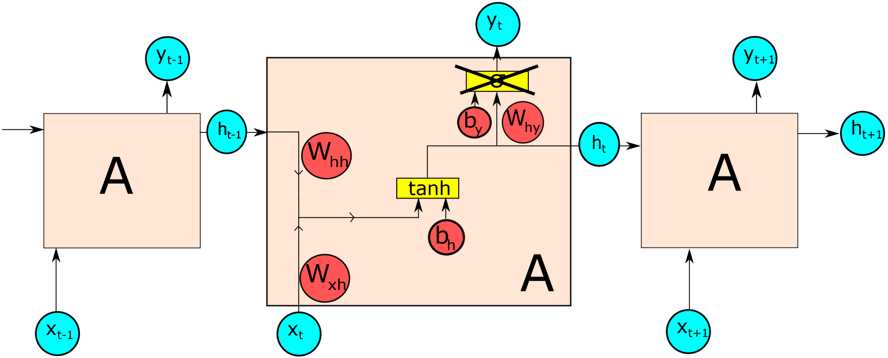
Elman Recurrent Neural Network

General strategy

- We interpret the **batch** dimension as **time** dimension now
- This allows to **reuse** loss functions, optimizers, initializers, activation functions and the Neural Network class
- Side note: Samples are often highly correlated in this dimension

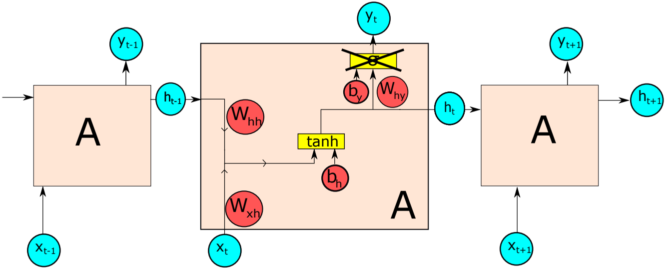


- Standard Elman Cell has sigmoid as “outer” activation function



- Standard Elman Cell has sigmoid as “outer” activation function
- To be more flexible: Outer activation function (sigmoid) will not be part of the unit itself (can be added as its own layer!)

Elman RNN Cell



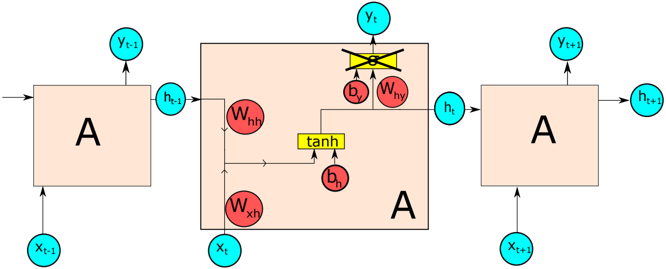
Output formula (without sigmoid):

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y$$

$\mathbf{W}_{hy}$: Weight matrix of a fully connected layer - is multiplied with current hidden state $\mathbf{h}_t$

$\mathbf{b}_h$: Output bias

Elman RNN Cell



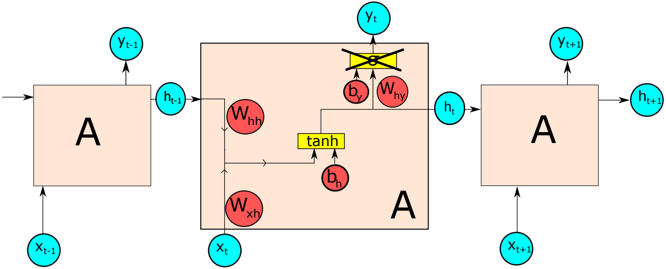
Output formula (without sigmoid):

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y$$

This is a **fully connected layer**!

Note: $\mathbf{y}_t$ only depends on its "own" $\mathbf{h}_t$ - if we have $\mathbf{h}_t$, we can compute $\mathbf{y}_t$ independently

Elman RNN Cell



Output formula (without sigmoid):

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y$$

This is a **fully connected layer**!

Note: $\mathbf{y}_t$ only depends on its "own" $\mathbf{h}_t$ - if we have $\mathbf{h}_t$, we can compute $\mathbf{y}_t$ independent of other inputs

A word on software engineering

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

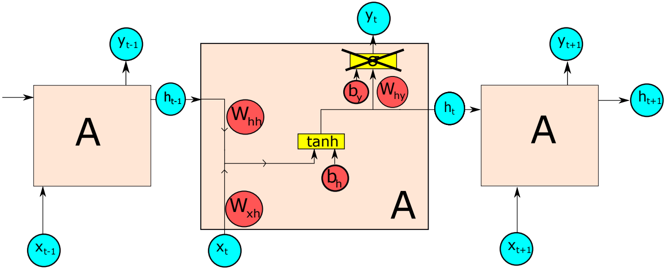
A word on software engineering

- 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

A word on software engineering

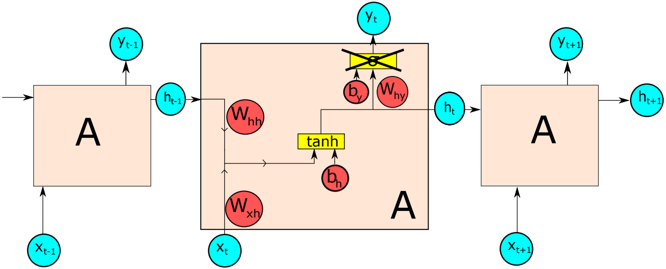
- In terms of **encapsulation** - how good was the idea to demand exposition of the weights as member?
- Suppose we implement the RNN cell as **composite** structure
- **Getters** and **Setters** provide us the flexibility to do so
- Takeaway? Not doing **proper software engineering** most of the time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

Elman RNN Cell



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

Elman RNN Cell



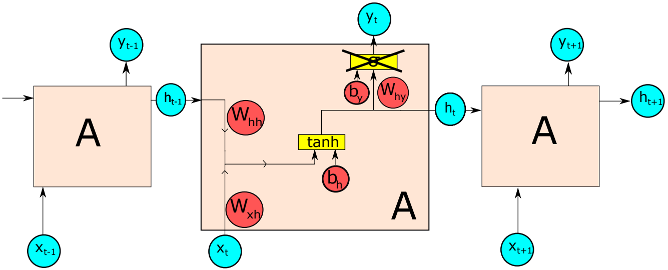
$$\mathbf{h}_t = \tanh\left(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h\right)$$

$\mathbf{W}_{hh}$: 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



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

$\mathbf{W}_h$: Weight matrix of a fully connected layer

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

Note: $\mathbf{h}_t$ cannot be computed independently (in contrast to $\mathbf{y}_t$)! We need $\mathbf{h}_{t-1}$!

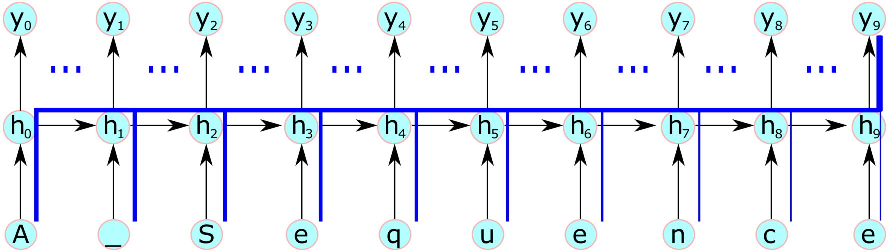
Backward pass

- Since we use **existing** fully connected layers, we can let them handle most of the backward pass!
- BUT: because of **multiple calls to their forward method**, the "internal state" (e.g. the saved input tensor) may be wrong

➔ We need to store and feed the necessary values for backpropagation externally and provide them to the embedded layers

- We also have to defer the weight updates until all outputs and gradients (for all current time steps) have been computed

Reminder: Backpropagation through time



- Implemented by passing the whole sequence as a **batch**
- Problem: Memory (we have to have **all** hidden states for the backward pass!)

Reminder: Truncated backpropagation through time

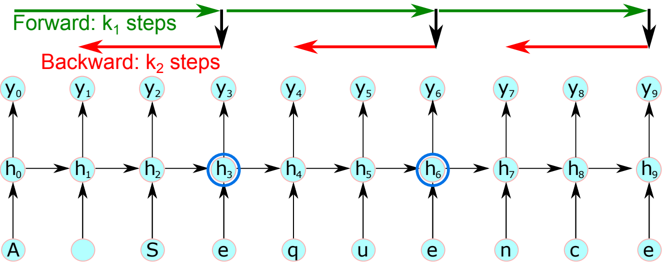
- Main idea: Keep processing sequence as a whole
- Adapt frequency and depth of update:
  - Every k_1 time steps, run BPTT for k_2 time steps
  - ➔ Parameter update cheap if k_2 small
- Hidden states are still exposed to many time steps
- Typically $k_2 \leq k_1$

Reminder: Truncated backpropagation through time

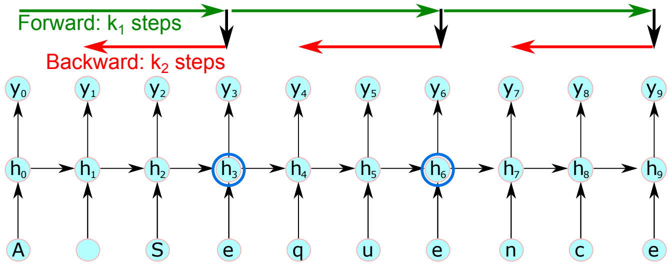
- Main idea: Keep processing sequence as a whole
- Adapt frequency and depth of update:
  - Every k_1 time steps, run BPTT for k_2 time steps
  - ➔ Parameter update cheap if k_2 small
- Hidden states are still exposed to many time steps
- Typically $k_2 \leq k_1$

Algorithm:

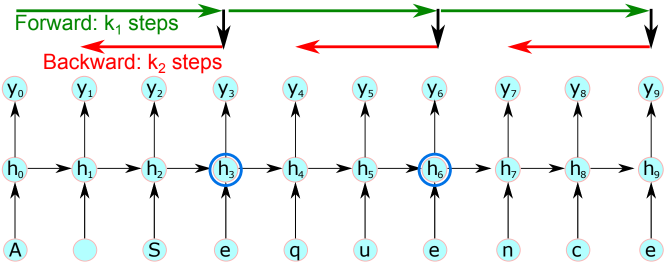
for t **from** 1 **to** T **do**:
    Run RNN for one step, computing h_t and y_t
    if $t \bmod k_1 == 0$:
        Run BPTT from t down to $t - k_2$



- Implemented by passing parts of the sequence as **batch**: k_1 = batch size
- k_2 given as additional parameter (will always be $\leq k_1$)

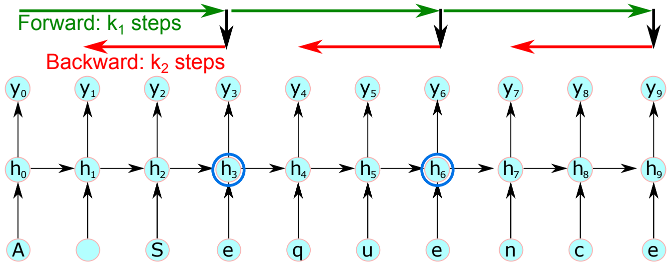


- Implemented by passing parts of the sequence as **batch**: k_1 = batch size
- k_2 given as additional parameter (will always be $\leq k_1$)
- BUT: If we **continue** a sequence with the next batch, we need to know the previous hidden state! (e.g. $\mathbf{h_3}$ for second batch)



- Implemented by passing parts of the sequence as **batch**: k_1 = batch size
- k_2 given as additional parameter (will always be $\leq k_1$)
- BUT: If we **continue** a sequence with the next batch, we need to know the previous hidden state! (e.g. $\mathbf{h_3}$ for second batch)

➔ We need to know whether a new batch is the **start** of a sequence or it **continues** from the previous batch



- Implemented by passing parts of the sequence as **batch**: k_1 = batch size
- k_2 given as additional parameter (will always be $\leq k_1$)
- BUT: If we **continue** a sequence with the next batch, we need to know the previous hidden state! (e.g. $\mathbf{h_3}$ for second batch)

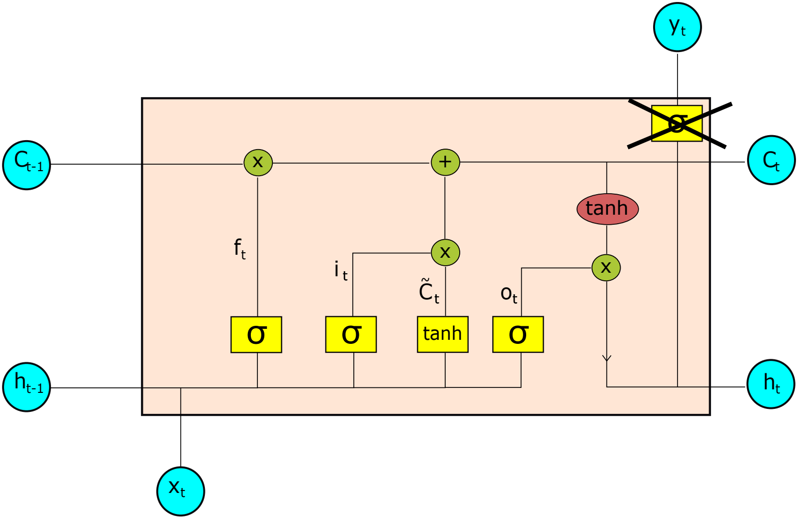
➔ We need to know whether a new batch is the **start** of a sequence or it **continues** from the previous batch

- Simply store the **last hidden state** of a batch and implement a **method** switching whether this state is reused in subsequent forward passes

Additional details in lecture slides

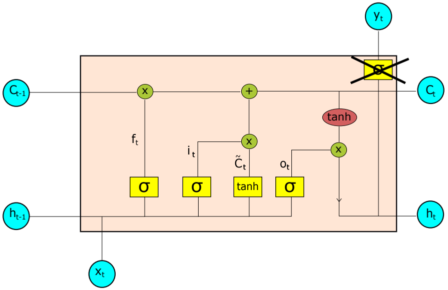
Please also refer to the lecture slides (7-Recurrent Neural Networks)!
Forward and Backward pass as well as BPTT for the Elman Unit are descriped on slides 7-9 and 15-23! (just keep in mind that we don't have the sigmoid function!)

Long Short-Term Memory



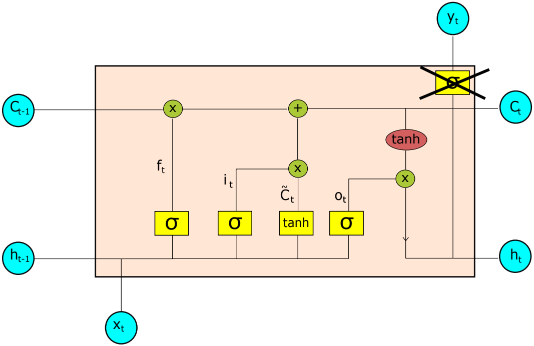
Note: Also refer to lecture slides 29-36!

Forward



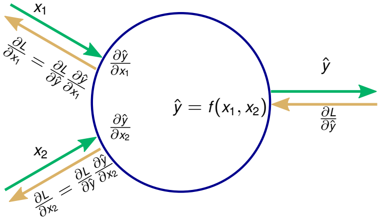
- We can again reuse **fully connected** layers
- The **concatenation** of input and hidden state is also **analogous** to the RNN
- The σ-gates and the yellow $\mathtt{tanh}$ can be a single **fully connected** layer with an output size of 4 · **dim(hidden state)**
- Remember that we have to pass the vectors of the input tensor to the embedded layers **sequentially**

Backward



- Most gradients are again handled by the **embedded layers**
- Again **store and feed the values for backprop externally** to the embedded layers because of **multiple calls to forward**
- We need gradients through **summation, multiplication and copying**

Backward



Sum

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

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

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

Multiply

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

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

Gradient is $\cdot$ with **switched inputs**

Copy

Backward pass of sum

So the gradient is a sum!

Thanks for listening.

Any questions?