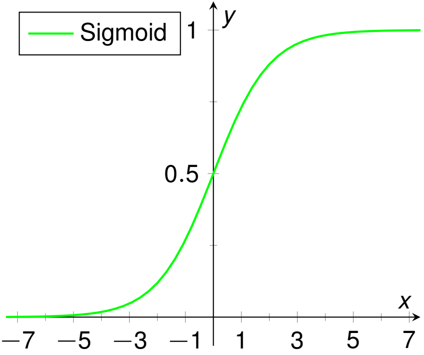
Recurrent Neural Networks

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-Nï¿½rnberg
April 25, 2023

Activation Functions

Sigmoid Activation Function



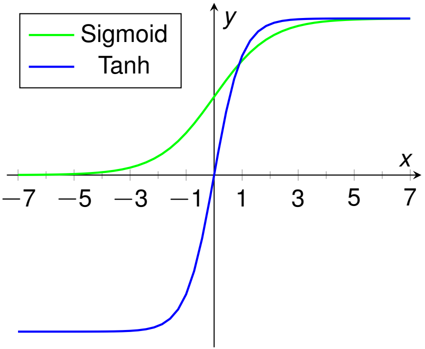
Sigmoid (logistic function)

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

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

➔ 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

General strategy

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

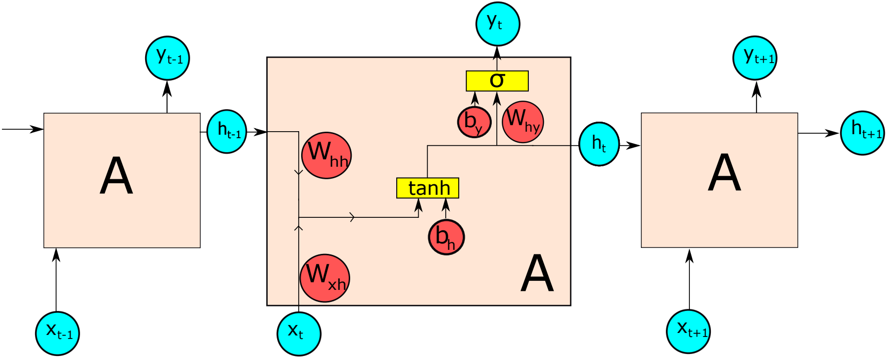
➔ Samples are correlated in this dimension

General strategy

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

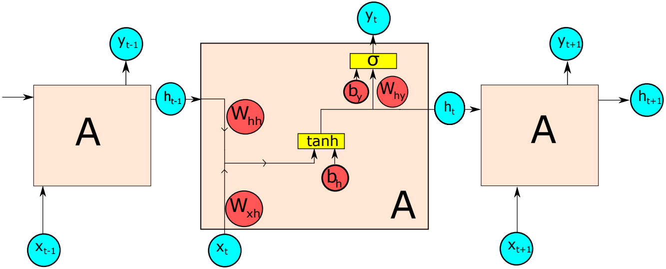
➔ Samples are correlated in this dimension

- This allows to **reuse** loss functions, optimizers, initializers, activation functions and the Neural Network class



y_t
y_{t-1}
y_{t+1}
σ
b_y
W_{hy}
h_{t-1}
h_t
h_{t+1}
W_{hh}
tanh
b_h
W_{xh}
A
A
A
x_{t-1}
x_t
x_{t+1}

Elman RNN Cell



Output formula:

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

$\mathbf{W}_{hy}$: 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?

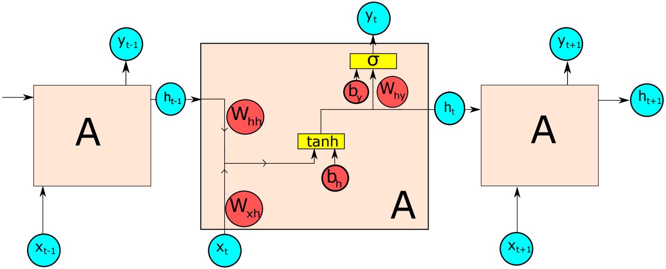
A word on software engineering

- 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

A word on software engineering

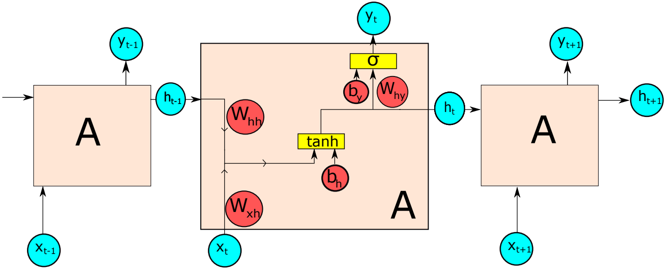
- In terms of **encapsulation** - how good was the idea to demand exposition of the weights as member?
- Suppose we implement the RNN cell as **composite** structure
- **Getters** and **Setters** provide us the flexibility to do so
- Takeaway? Not doing **proper software engineering** most of the time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

Elman RNN Cell



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

Elman RNN Cell



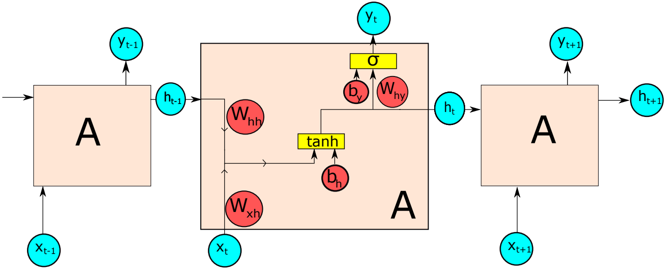
$$\mathbf{h}_t = \tanh\left(\mathbf{h}_{t-1} \cdot \mathbf{W}_{hh} + \mathbf{x}_t \cdot \mathbf{W}_{xh} + \mathbf{b}_h\right)$$

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



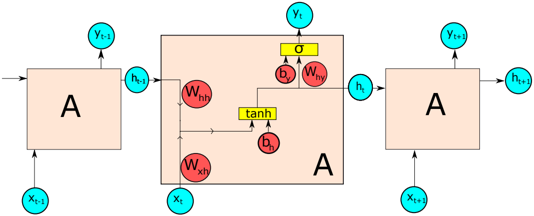
$$\mathbf{h}_t = \tanh\left(\tilde{\mathbf{x}}_t \cdot \mathbf{W}_h\right)$$

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

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

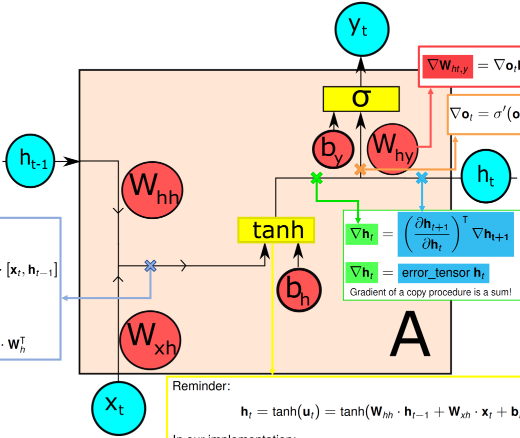
Different from output: Not processed independently!

Backward



- 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



$$\nabla \mathbf{W}_{ht,y} = \nabla \mathbf{o}_t \mathbf{h}_t^T = \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial h_t}{\partial \mathbf{W}_{hy}} = \nabla \mathbf{o}_t \cdot \mathbf{o}_t^T$$

$$\nabla \mathbf{o}_t = \sigma'(\mathbf{o}_t) \cdot \frac{\partial L}{\partial \hat{\mathbf{y}}_t}(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

$$\nabla \mathbf{h}_t = \left(\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right)^T \nabla \mathbf{h}_{t+1} + \left(\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\right)^T \nabla \mathbf{o}_t$$

$\nabla \mathbf{h}_t =$ error_tensor $\mathbf{h}_t$ + error_tensor $\mathbf{y}_t$

Gradient of a copy procedure is a sum!

$$\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{W}_h} = \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h} = \nabla \mathbf{h}_t \cdot \tanh'(\mathbf{u}_t) \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}]$$

$$\frac{\partial \hat{\mathbf{y}}_t}{\partial \tilde{\mathbf{x}}_t} = \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \tilde{\mathbf{x}}_t} = \nabla \mathbf{h}_t \cdot \tanh'(\mathbf{u}_t) \cdot \mathbf{W}_h^T$$

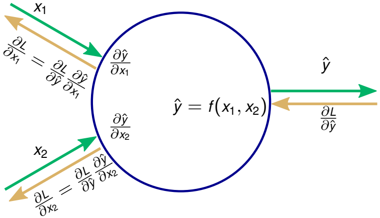
Reminder:

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

In our implementation:

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

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

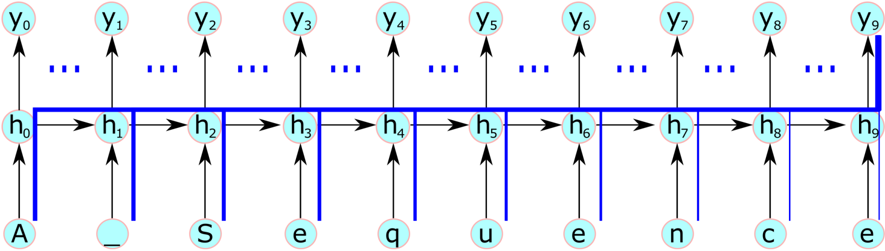
$$f(x_1, x_2) = x_1 \cdot x_2$$
$$\frac{\partial \hat{y}}{\partial x_1} = x_2$$

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

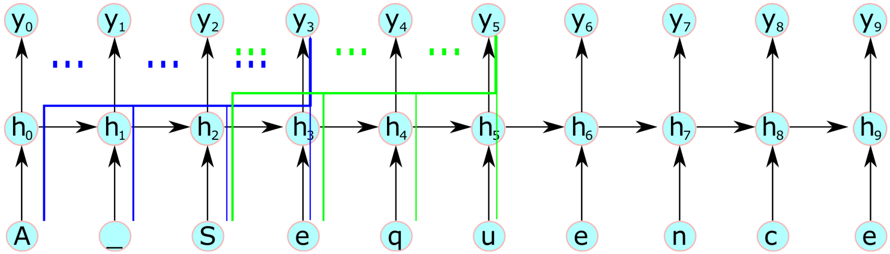
Copy

Backward pass of sum

So the gradient is a sum!

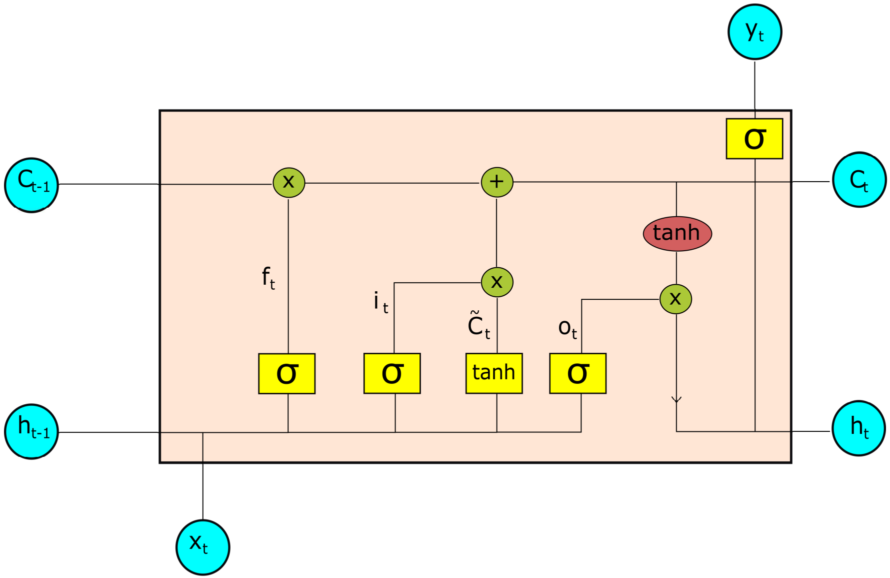


- Implemented by passing the whole sequence as a **batch**

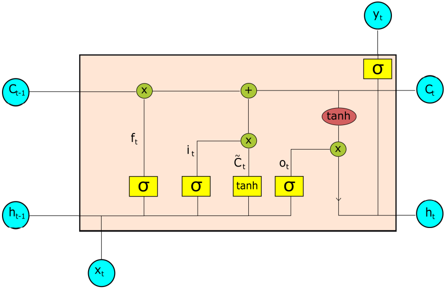


- Implemented by passing **overlapping** parts as a **batch**
- We need to implement memory **between states**
- Simply store the **last hidden state** and implement a **method** switching whether this state is reused in subsequent forward passes.
- Data has to be fed in **accordingly**!
- Referencing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

Long Short-Term Memory (optional)

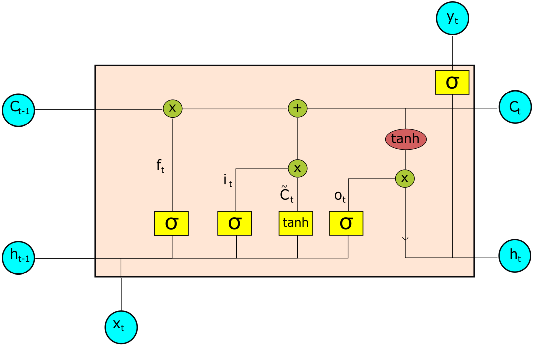

yt
σ
Ct-1
x
+
Ct
tanh
ft
x
it
C̃t
x
ot
σ
σ
tanh
σ
ht-1
ht
xt

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 $\mathtt{tanh}$ can be a single **fully connected** layer with an output size of $4 \cdot$ **dim(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?