

# Deep Learning for NLP

Universität Bielefeld

## Lecture 9 – Recurrent Neural Nets

**Dr. Steffen Eger**

steffen.eger@uni-bielefeld.de



**Natural Language Learning Group (NLLG)**

## Previous lectures:

- Introduction (MLPs, loss functions, batch size, activation functions, etc.)
- Embeddings – continuous representations of words, letters, sentences, etc.
- More elaborate architecture: CNN

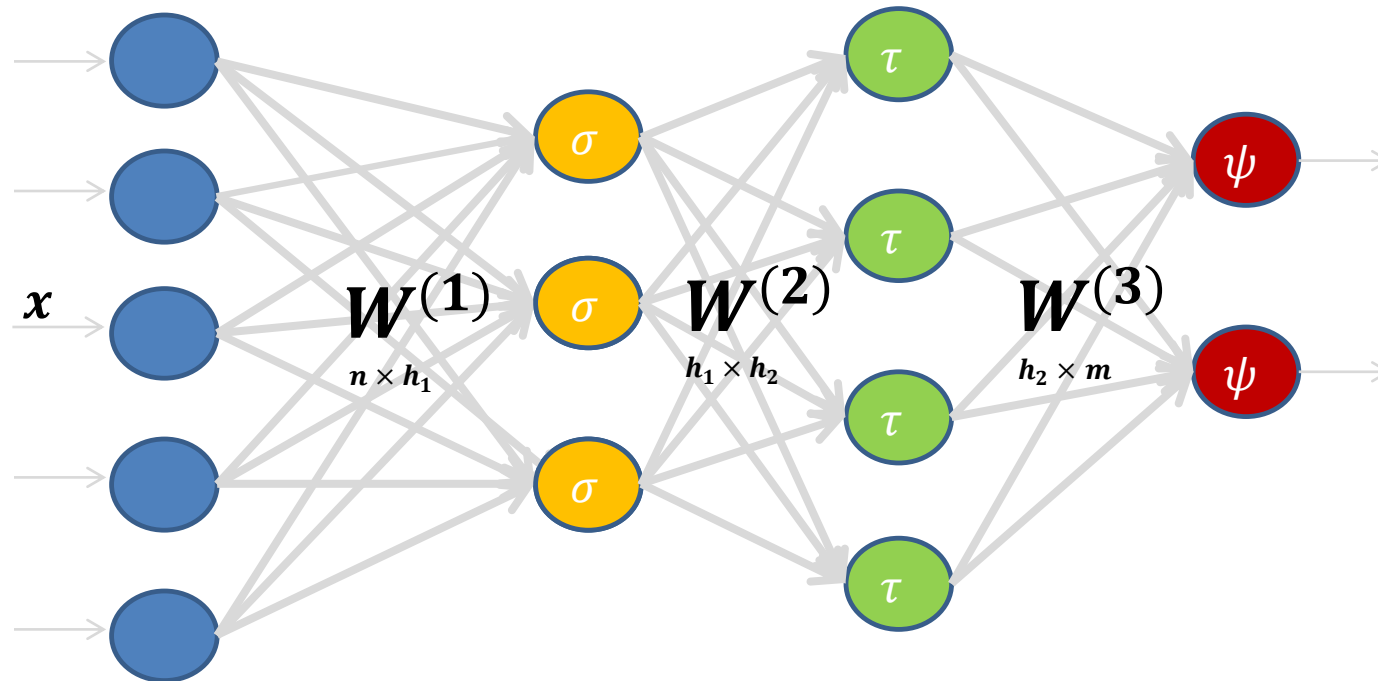
# This lecture:

- Recurrent Neural Nets (RNNs)
  - Basic principles
  - Extensions (Bidirectional, etc.)
  - NLP applications: Sequence tagging & sentence classification
- Vanishing gradients
  - Simple Remedies
  - GRUs & LSTMs

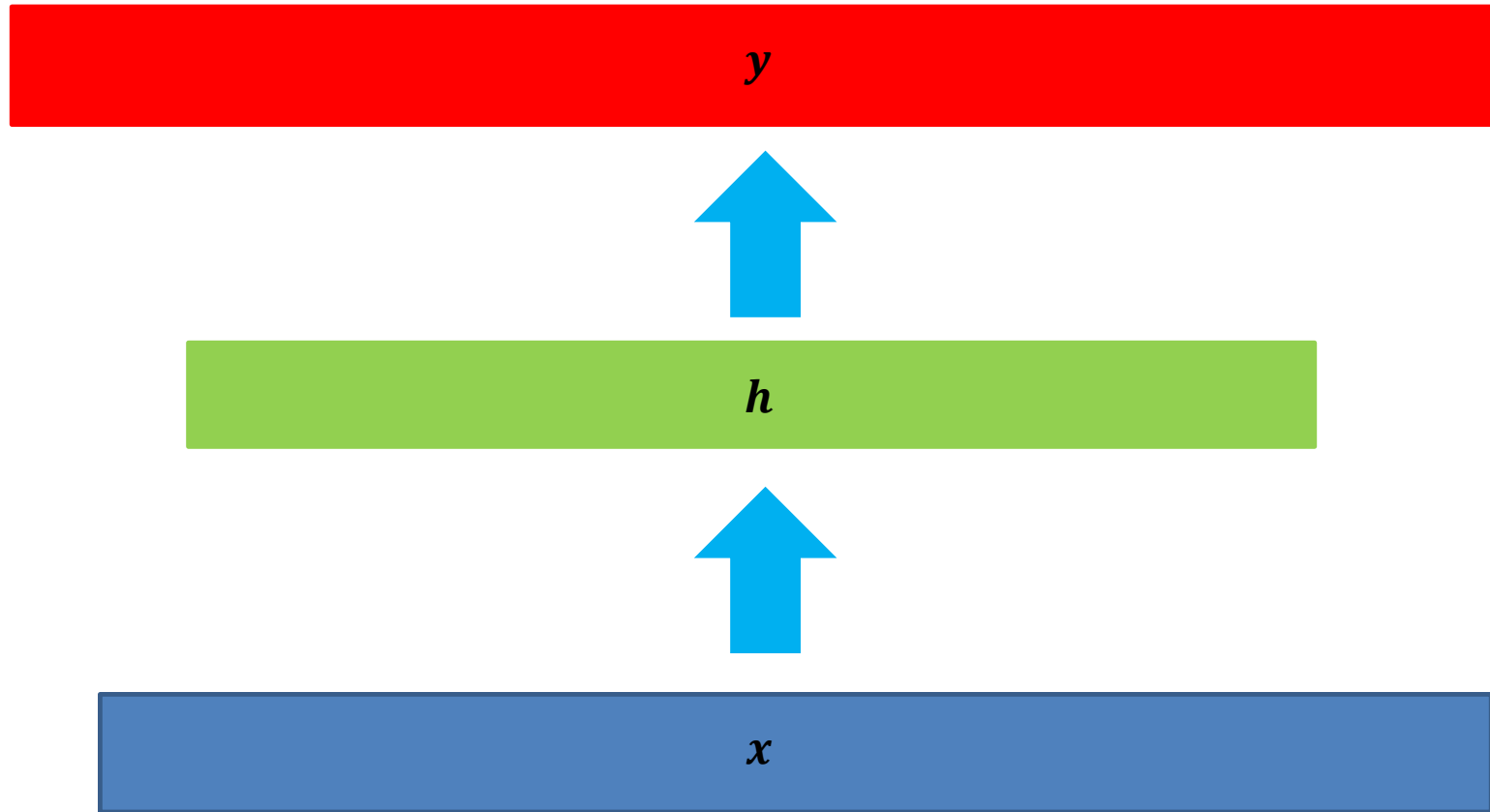
# Recurrent Neural Nets

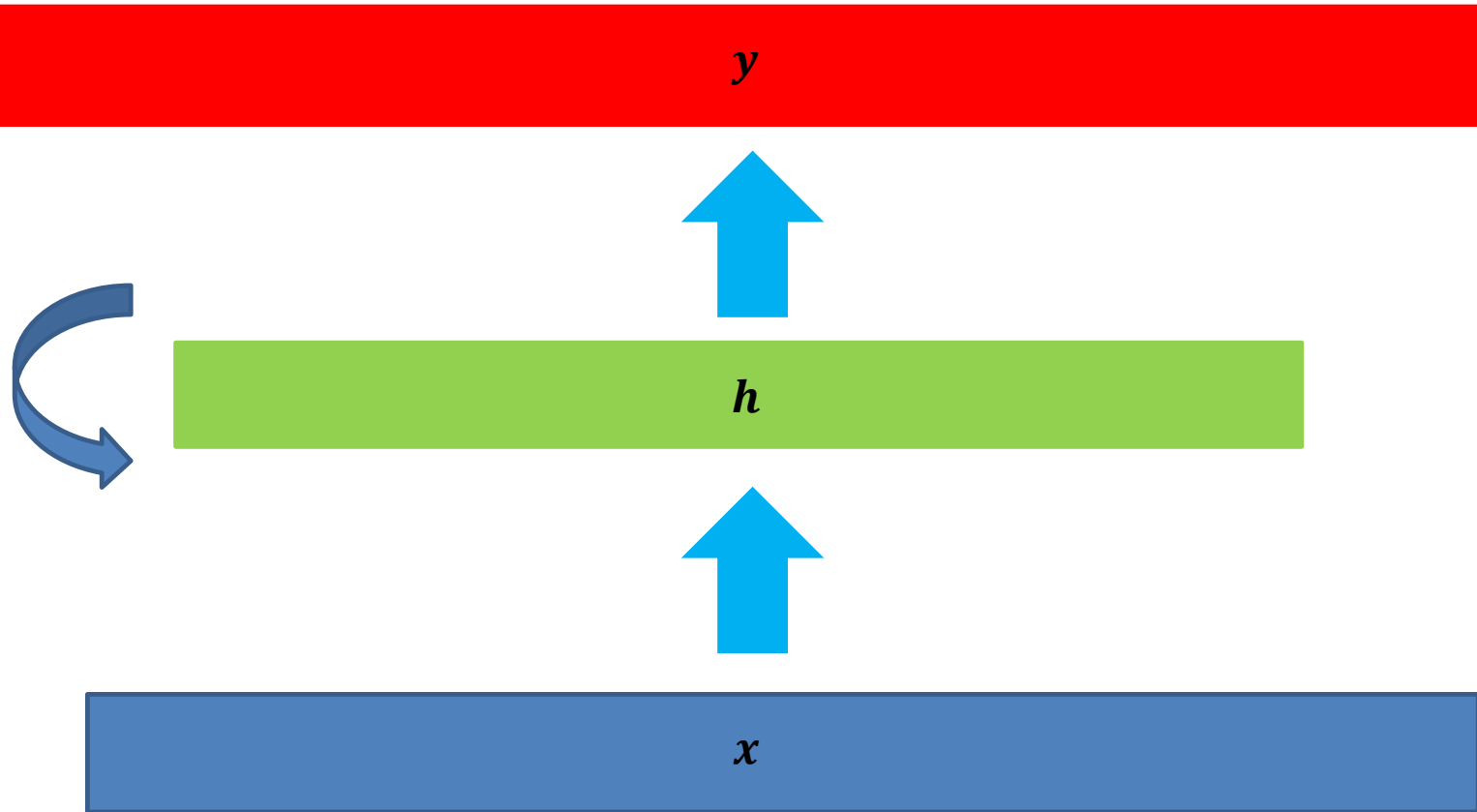
## Basic principles

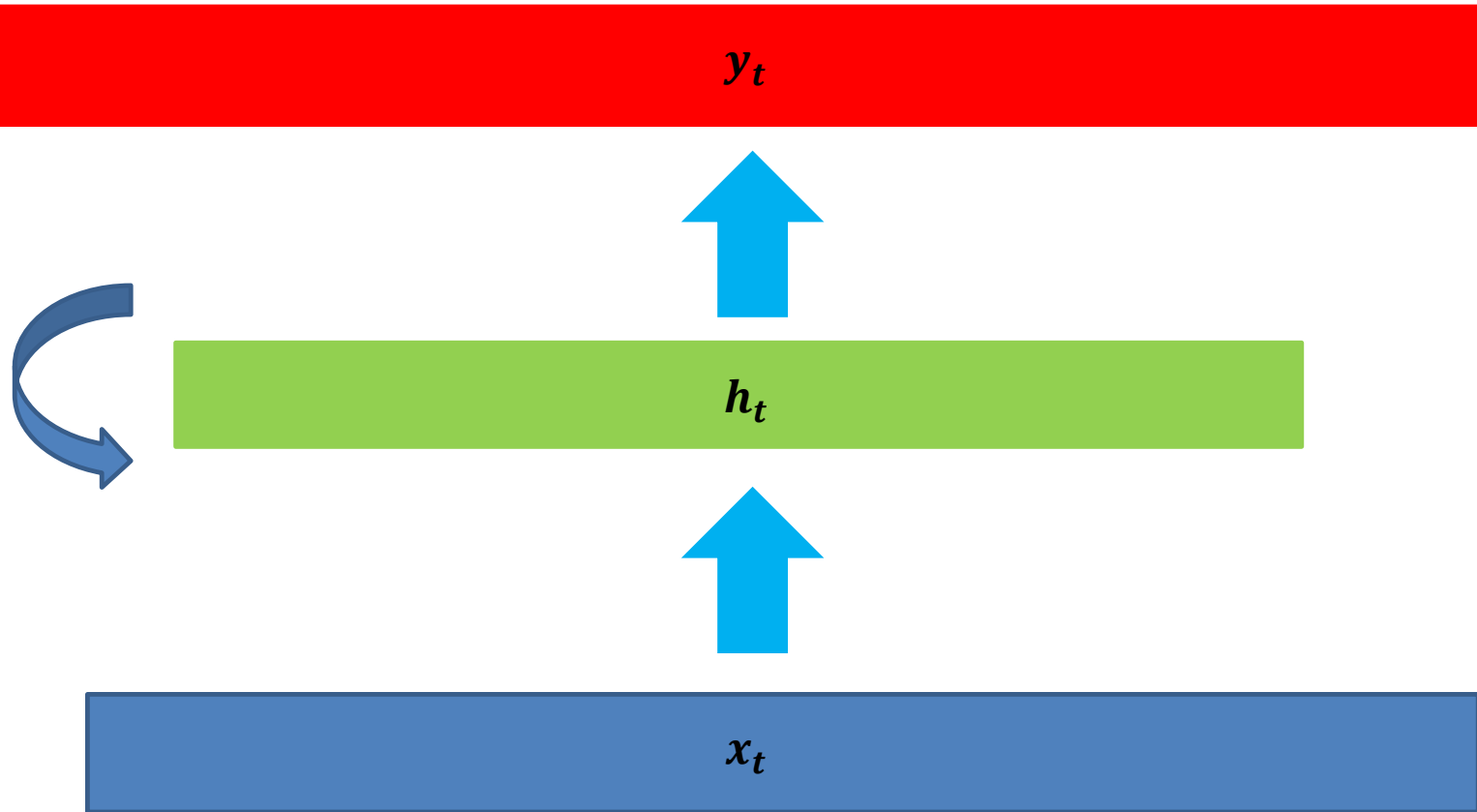
# Remember FF Nets / MLP



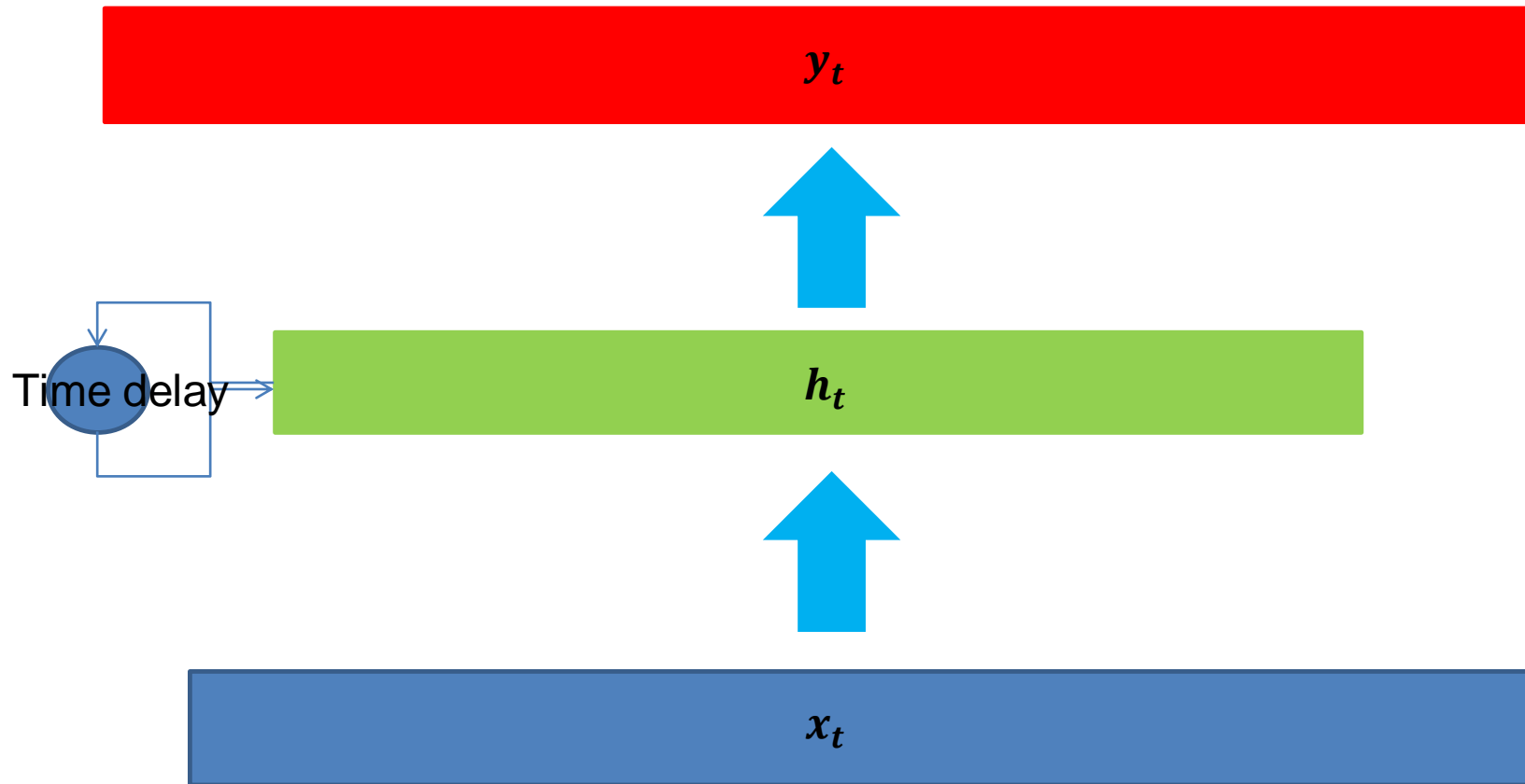
# Feedforward / MLP

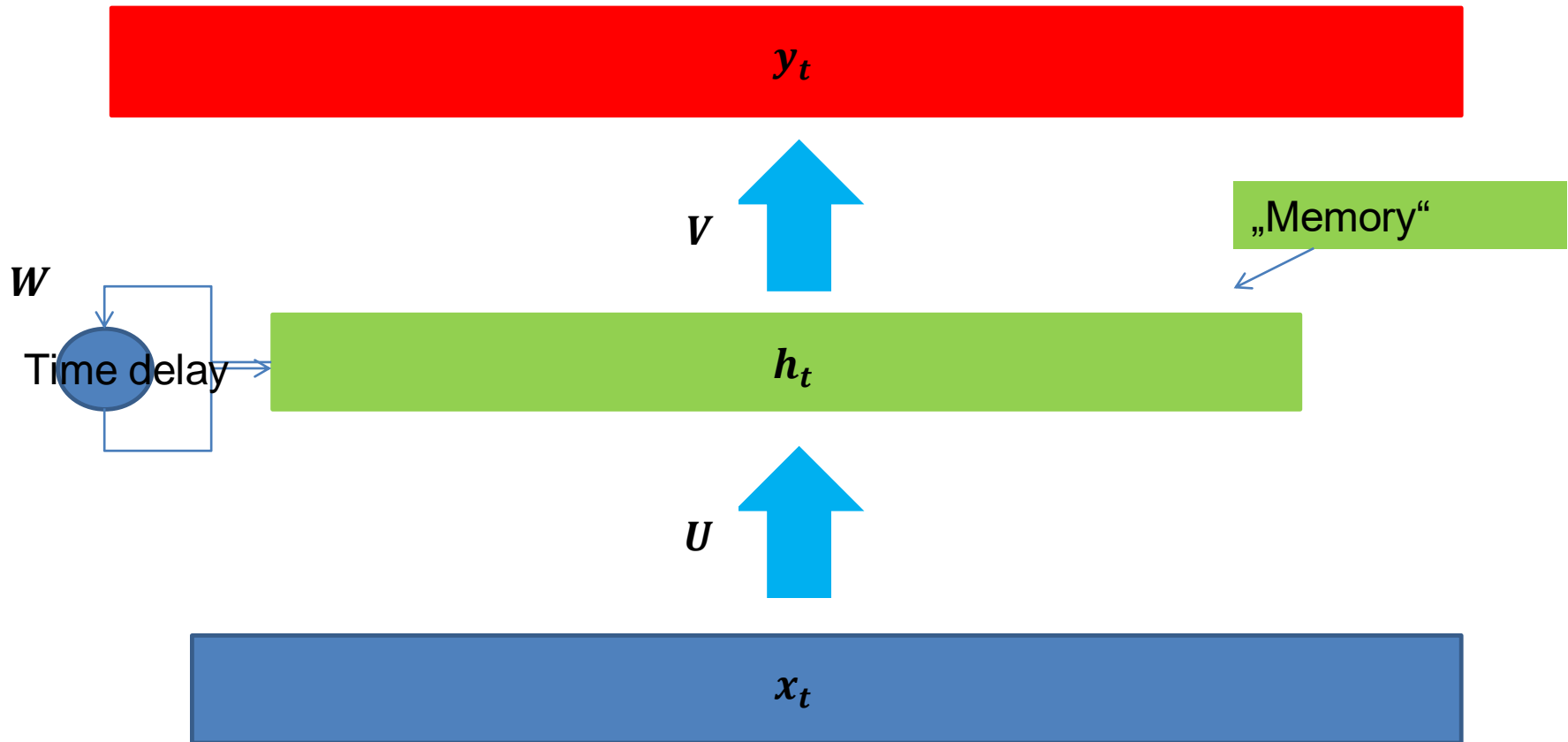












# RNN – Formally

- Input vectors  $\mathbf{x}_t$ ,  $t = 1, 2, 3, \dots$  lie in  $R^{1 \times n}$
- $\mathbf{h}_t = \sigma_H(\mathbf{x}_t \mathbf{U} + \mathbf{h}_{t-1} \mathbf{W} + \mathbf{b})$ 
  - Where  $\mathbf{U} \in R^{n \times d}$ ,  $\mathbf{W} \in R^{d \times d}$ ,  $\mathbf{h}_t \in R^{1 \times d}$ 
    - $d$  is hidden dimensionality
- $\mathbf{y}_t = \sigma_Y(\mathbf{h}_t \mathbf{V} + \mathbf{c})$ 
  - Where  $\mathbf{V} \in R^{d \times m}$

What we want to optimize is

- Average loss  $E$  over individual time losses  $E_t$ 
  - E.g.  $E_t = \text{ce}(\mathbf{y}_t, \mathbf{t}_t) = -\sum_j t_{t,j} \log y_{t,j}$
  - $E = \frac{1}{T} \sum_t E_t$

# RNN – Example

- Input: “A rusty can”
- Embeddings:  $\mathbf{x}_1 = (1,0,0)$ ,  $\mathbf{x}_2 = (1,1,2)$ ,  $\mathbf{x}_3 = (1,-1,1)$
- Truth: DET,ADJ,NOUN, encoded as 1-hot vectors (in a 4-d label space)
- Activations: ReLU for hidden layer, Softmax for output layer

# RNN – Example

- Initialization:

- $U = \begin{pmatrix} 1 & 1 \\ 2 & 0 \\ 0.5 & 1 \end{pmatrix}$

- $W = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

- $V = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & \frac{1}{3} & -1 \end{pmatrix}$

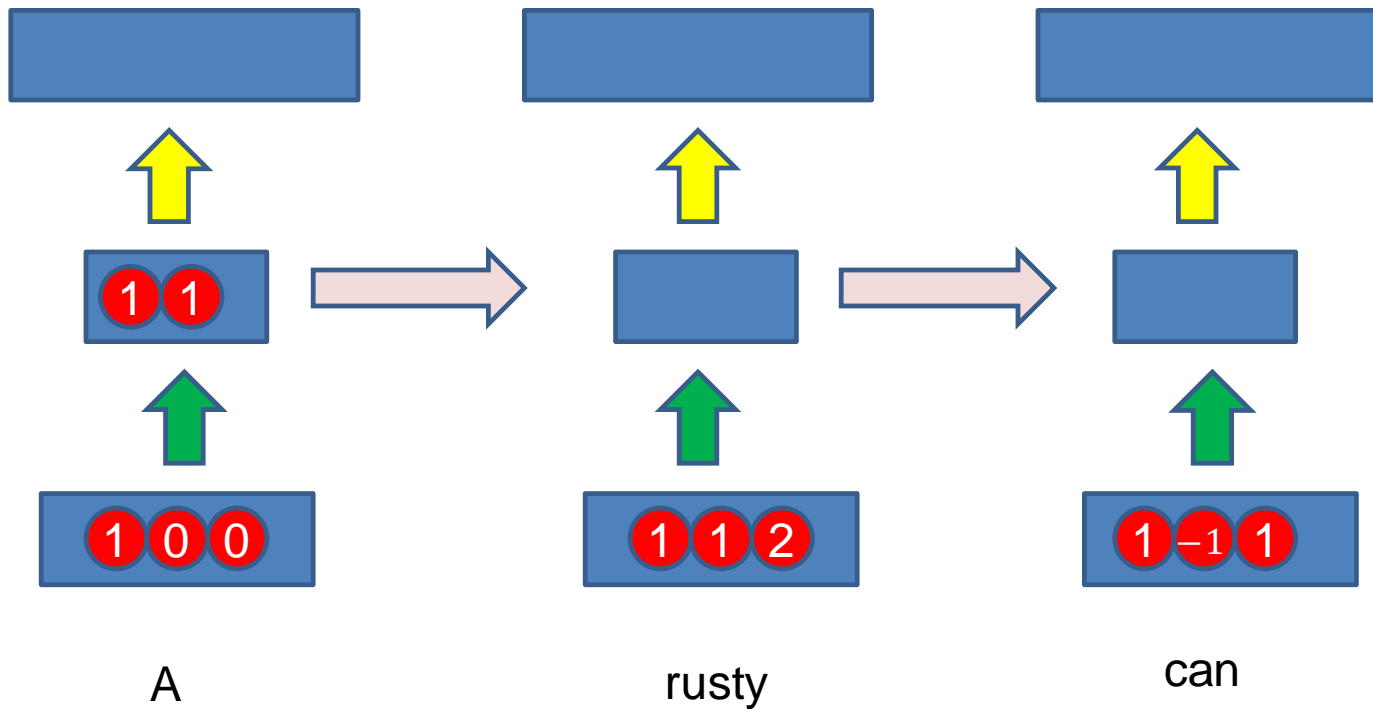
- $\mathbf{b} = \mathbf{c} =$  zero-vectors of appropriate size

- $\mathbf{h}_0 = (0,0)$

# RNN – Example

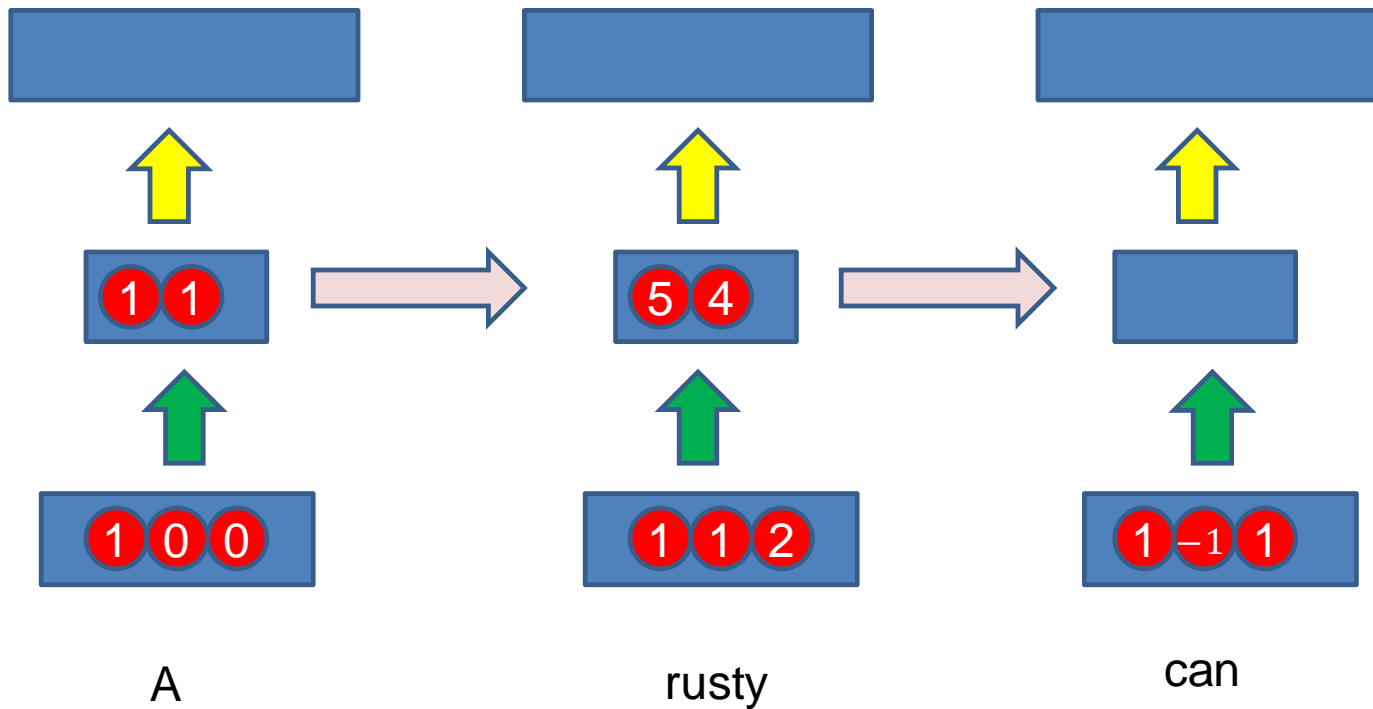
$$h_1 = \sigma_H(x_1 U + h_0 W + b)$$

$$h_1 = (1, 1)$$



# RNN – Example

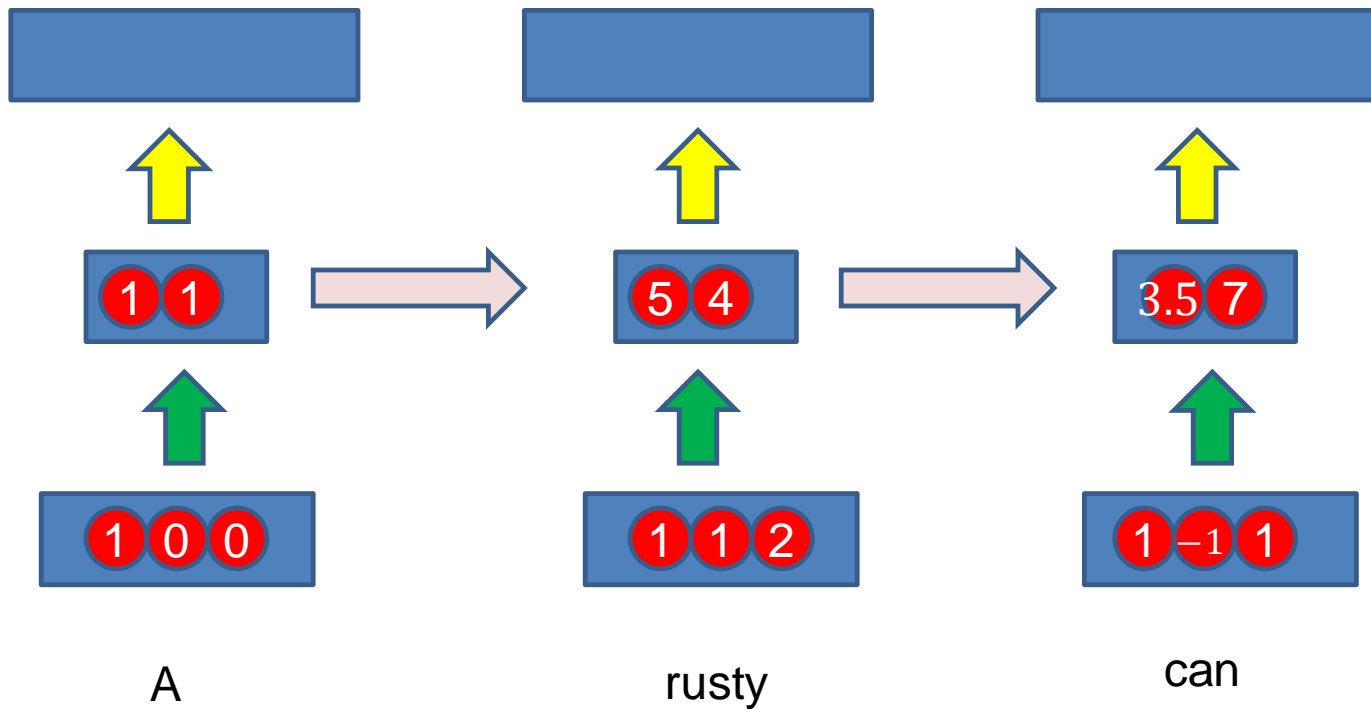
$$h_2 = \sigma_H(x_2 U + h_1 W + b)$$
$$h_2 = (5, 4)$$





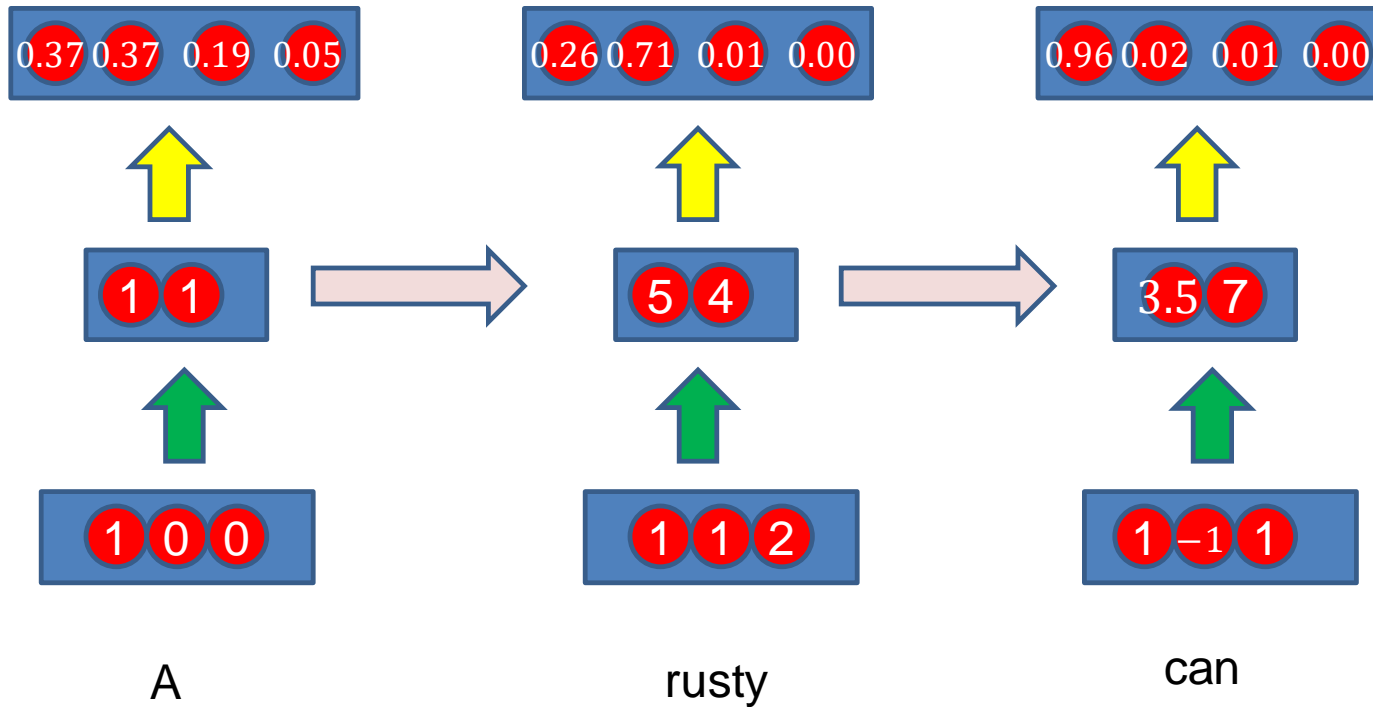
# RNN – Example

$$h_3 = \sigma_H(x_3 U + h_2 W + b)$$
$$h_3 = (3.5, 7)$$



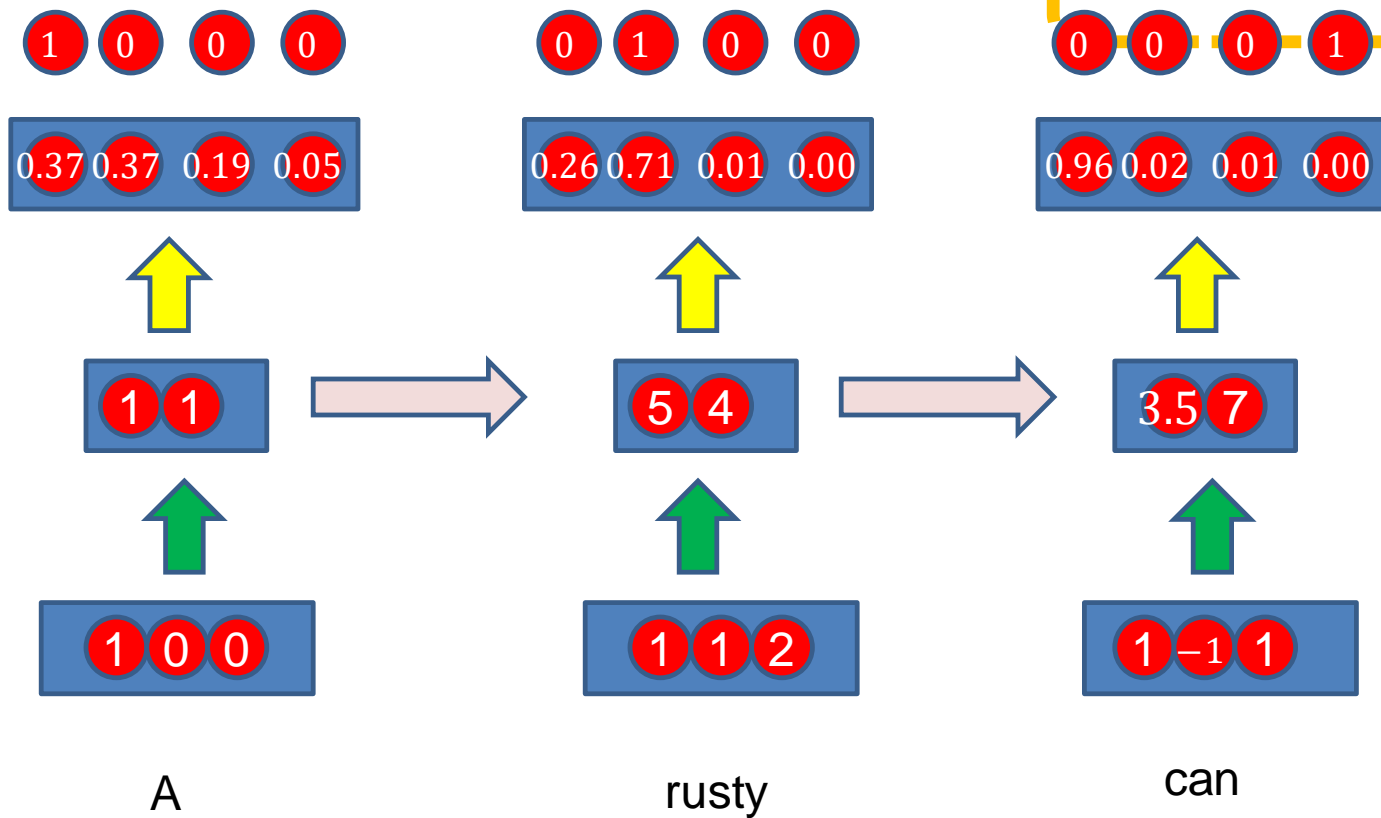
# RNN – Example

$$y_t = \sigma_Y(h_t V + c)$$

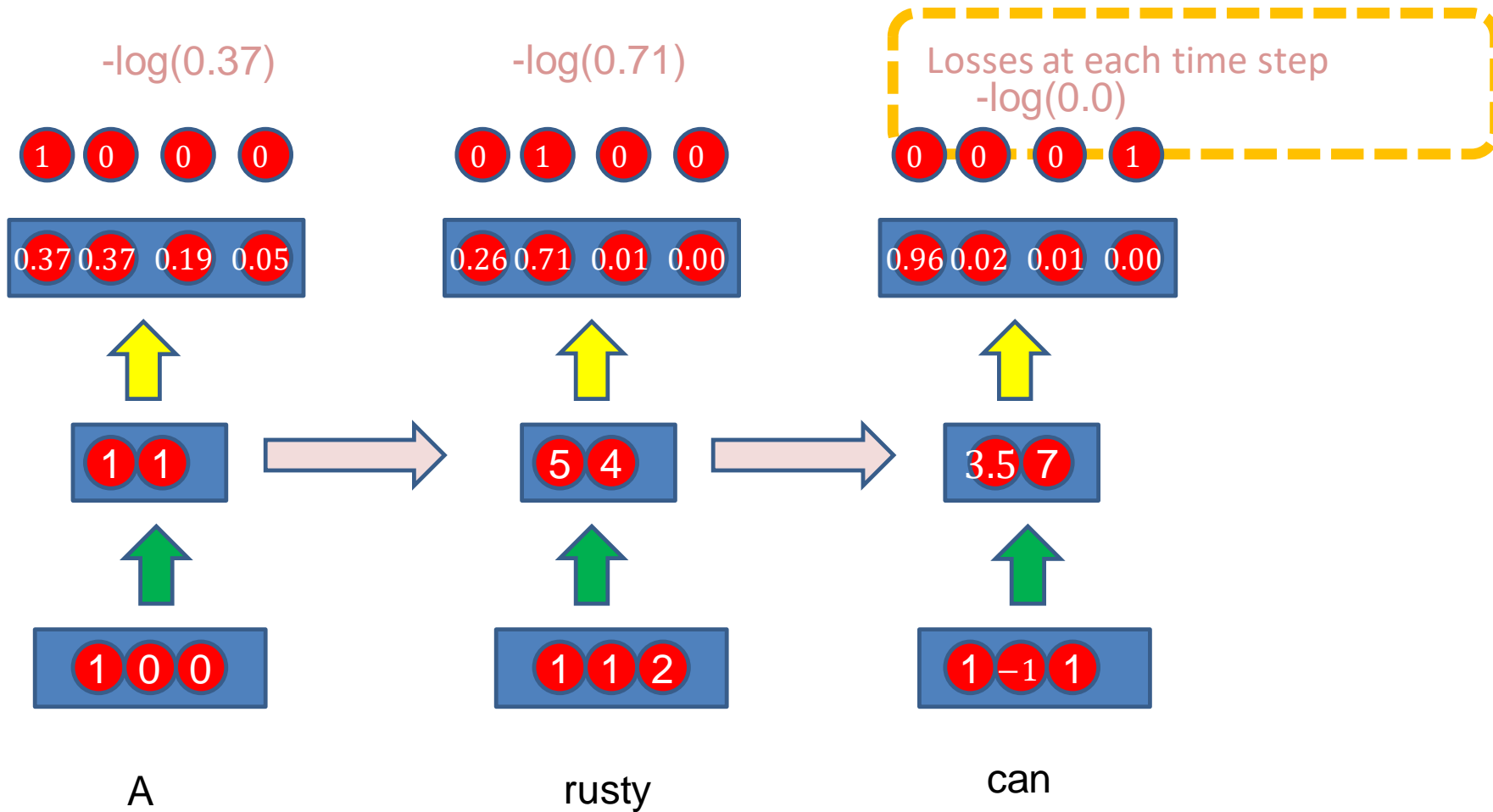


# RNN – Example

Truth vectors



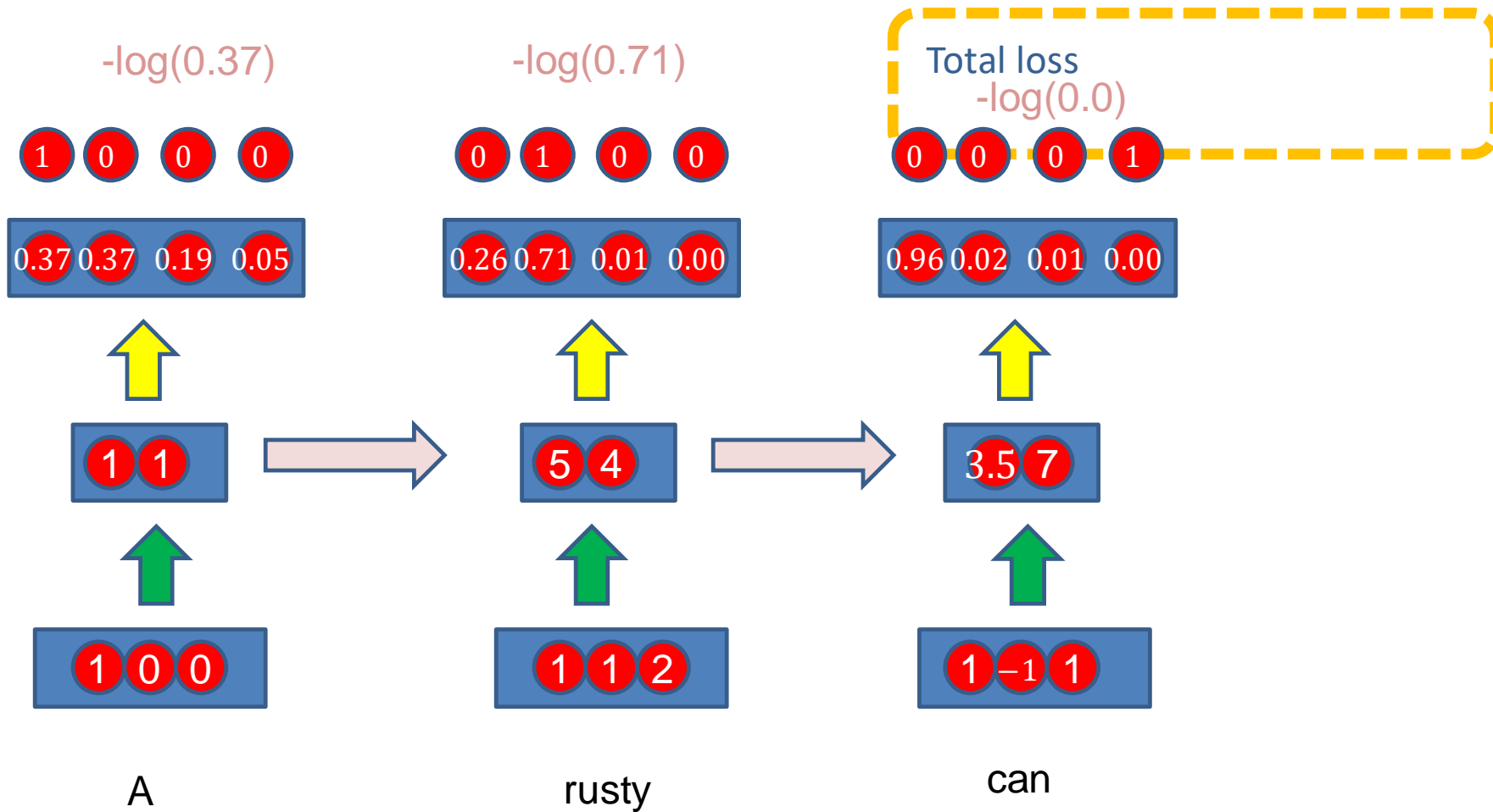
# RNN – Example



# RNN – Example

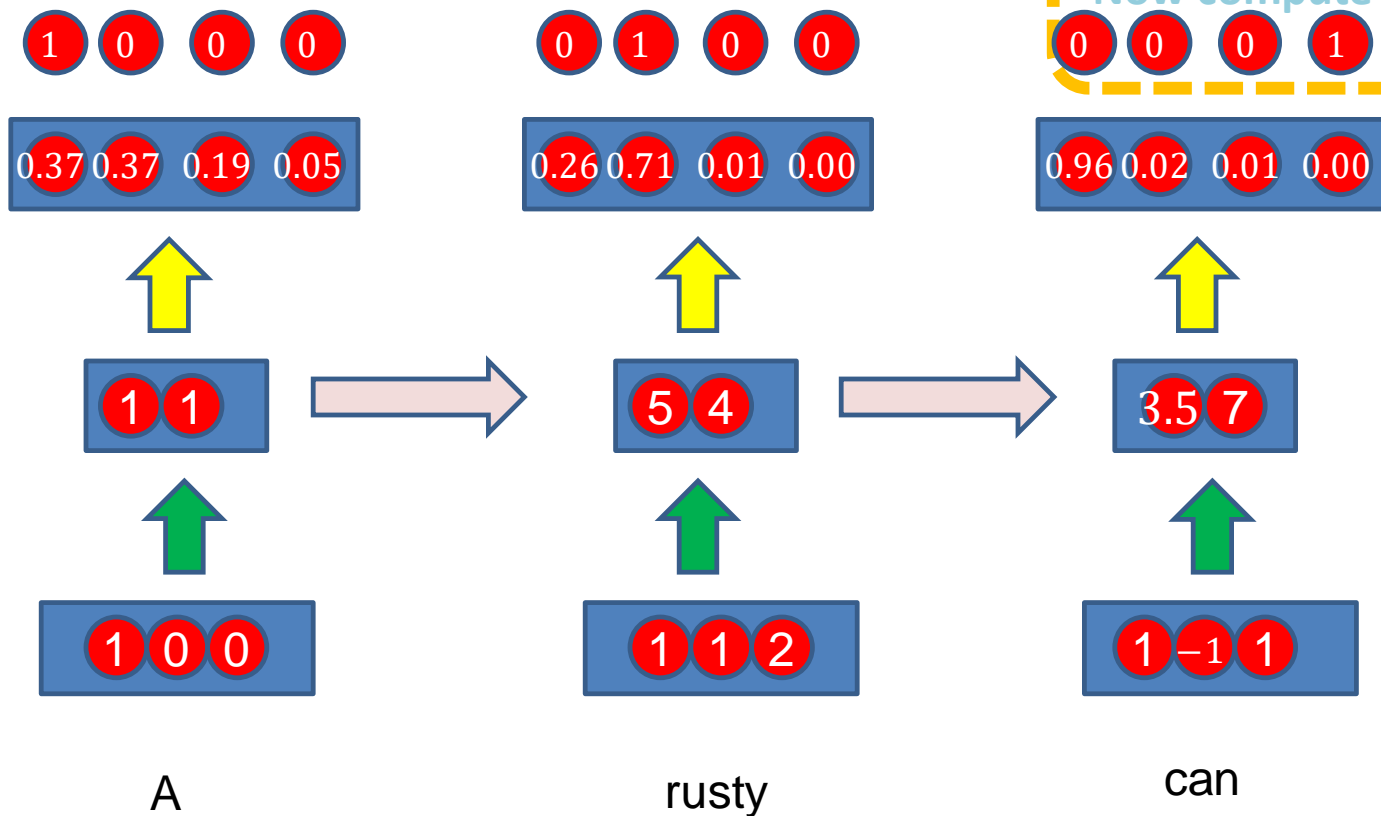
$$\frac{1}{3} \cdot (-\log(0.37) - \log(0.71) - \log(0.0))$$

Universität Bielefeld



# RNN – Example

$\frac{1}{3} \cdot (-\log(0.37) - \log(0.71) - \log(0.0))$  Universität Bielefeld



This was the forward pass  
Now compute gradients+update weights

# RNN – Weight update / Gradient computation

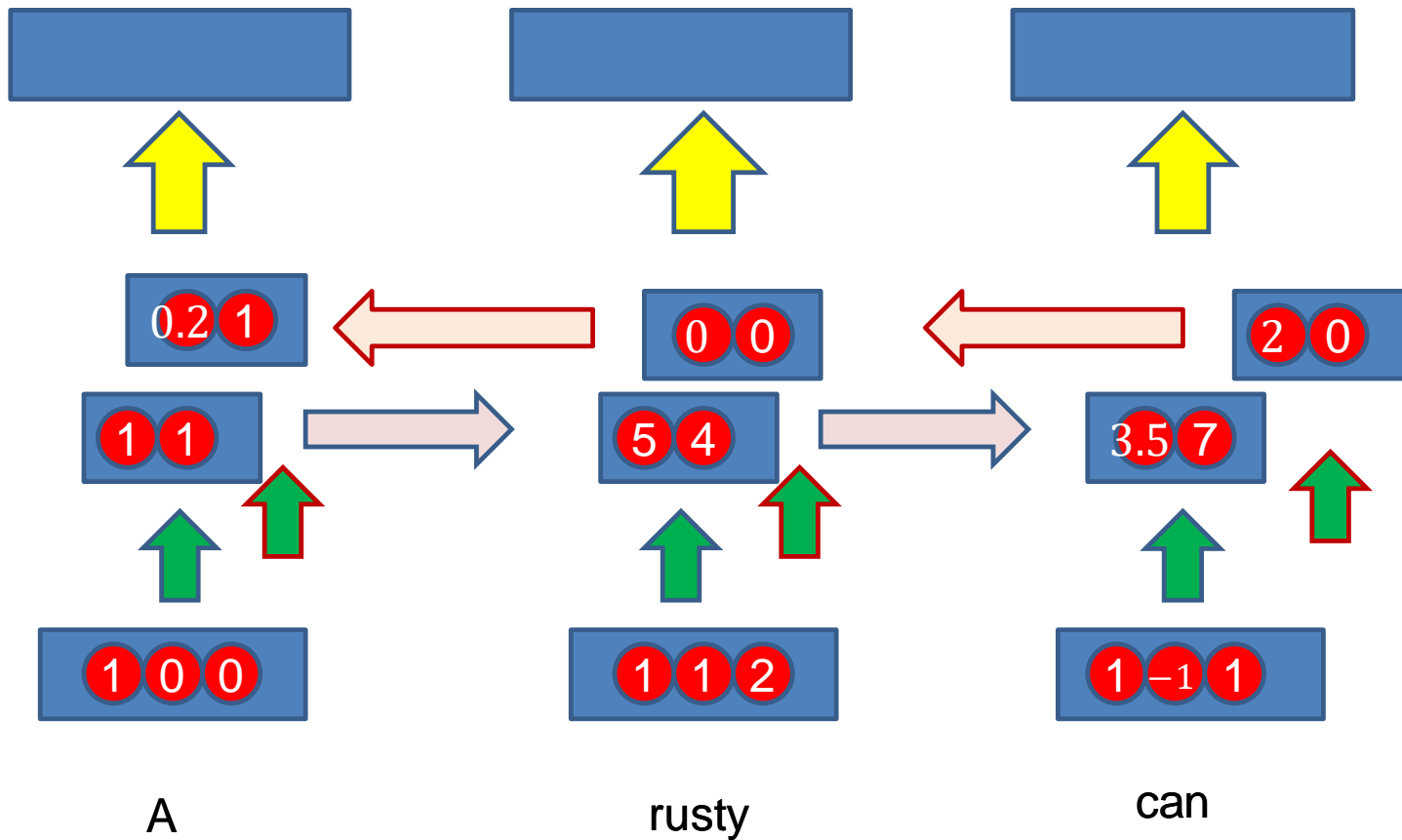
- Computation of gradient is similar as in standard MLP
  - But: Need to keep in mind that several parameters are shared
  - Some people call backprop for RNNs “backpropagation through time” (BPTT)
  - No need to go through, TensorFlow does it for you
- If you want to do it brute-force, can also do it numerically
  - i.e., for each individual weight  $w$ , compute  $\frac{f(w+h)-f(w)}{h}$
  - Where  $f$  is the loss function
- Weight update after gradient computation is  $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla f$  as usual

- Infinite window size – “from the left”
  - Memory can (in principle) store everything from the past
- That’s good, but we also want to base our decision on future words/tokens
  - Bidirectional RNN:
    - run a second RNN from “right to left”
    - With independent weights
    - Concatenate the forward and backward hidden states
      - $\mathbf{h}_t = [\vec{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t]$
      - Note that  $\mathbf{V}$  is of dimension  $2d \times m$  in this case



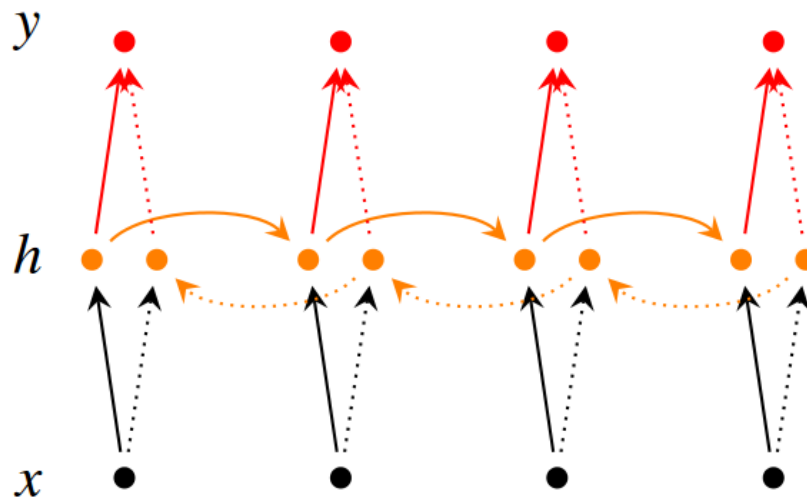
# Recurrent Neural Nets Extensions

# Bidirectional RNN – Illustration



- **Bidirectional RNNs**

Problem: For classification you want to incorporate information from words both preceding and following



$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

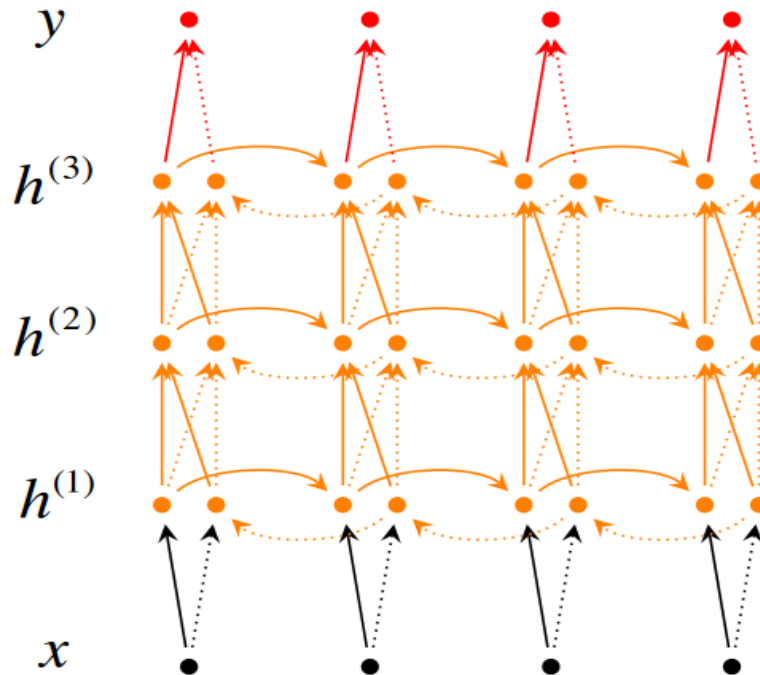
$$y_t = g(U[\vec{h}_t; \overleftarrow{h}_t] + c)$$

$h = [\vec{h}; \overleftarrow{h}]$  now represents (summarizes) the past and future around a single token.

From: <https://cs224d.stanford.edu/lectures/CS224d-Lecture8.pdf>

# Extensions of simple RNNs

- Deep Bidirectional RNNs



$$\vec{h}_t^{(i)} = f(\vec{W} h_t^{(i-1)} + \vec{V} \vec{h}_{t-1}^{(i)} + \vec{b}^{(i)})$$

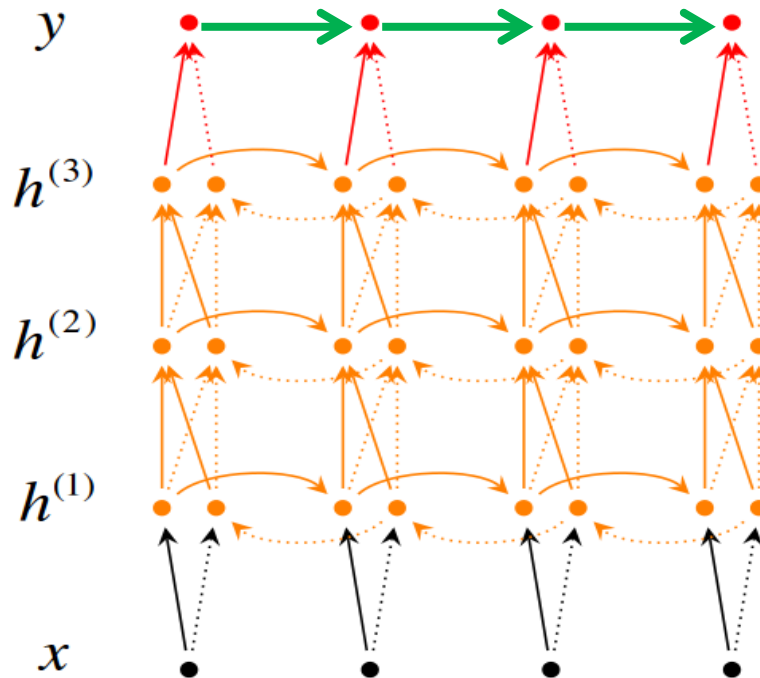
$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W} h_t^{(i-1)} + \overleftarrow{V} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$y_t = g(U[\vec{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

Each memory layer passes an intermediate sequential representation to the next.

# Extensions of simple RNNs

- RNNs with output connections



$$\vec{h}_t^{(i)} = f(\vec{W}^{(i)} h_t^{(i-1)} + \vec{V}^{(i)} \vec{h}_{t-1}^{(i)} + \vec{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

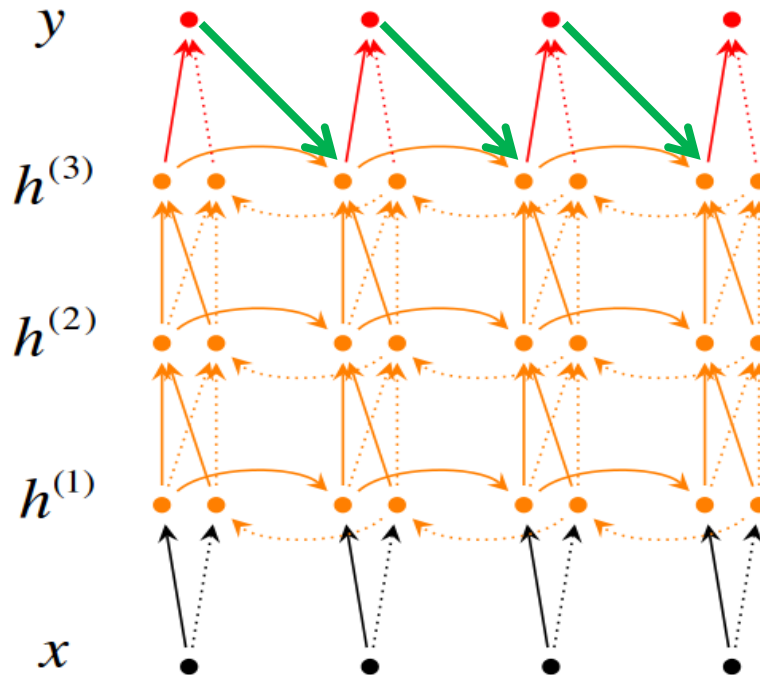
$$y_t = g(U[\vec{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

Equations?

Each memory layer passes an intermediate sequential representation to the next.

# Extensions of simple RNNs

- RNNs with output connections



$$\vec{h}_t^{(i)} = f(\vec{W}^{(i)} h_t^{(i-1)} + \vec{V}^{(i)} \vec{h}_{t-1}^{(i)} + \vec{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

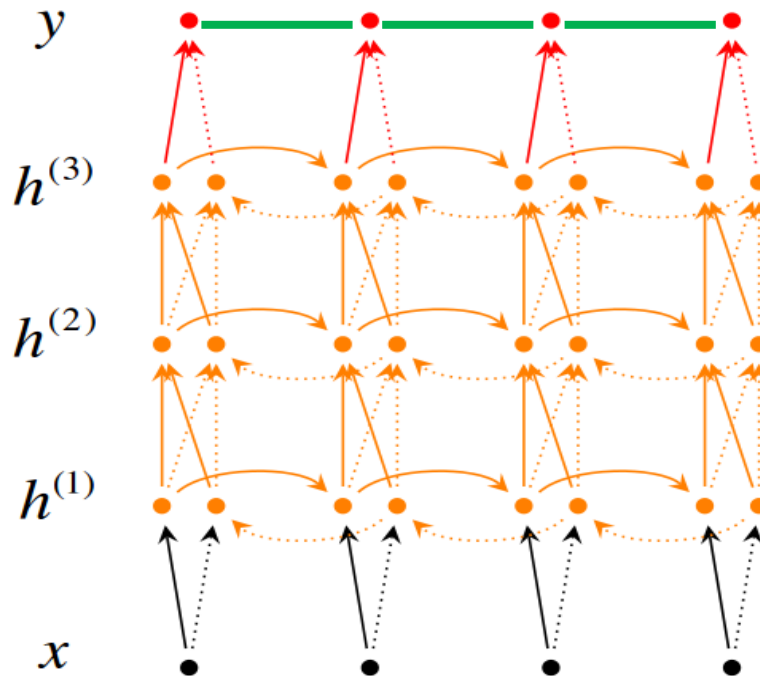
$$y_t = g(U[\vec{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

Equations?

Each memory layer passes an intermediate sequential representation to the next.

# Extensions of simple RNNs

- RNNs with output connections: CRF instead of forward conn.



$$\vec{h}_t^{(i)} = f(\vec{W}^{(i)} h_t^{(i-1)} + \vec{V}^{(i)} \vec{h}_{t-1}^{(i)} + \vec{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

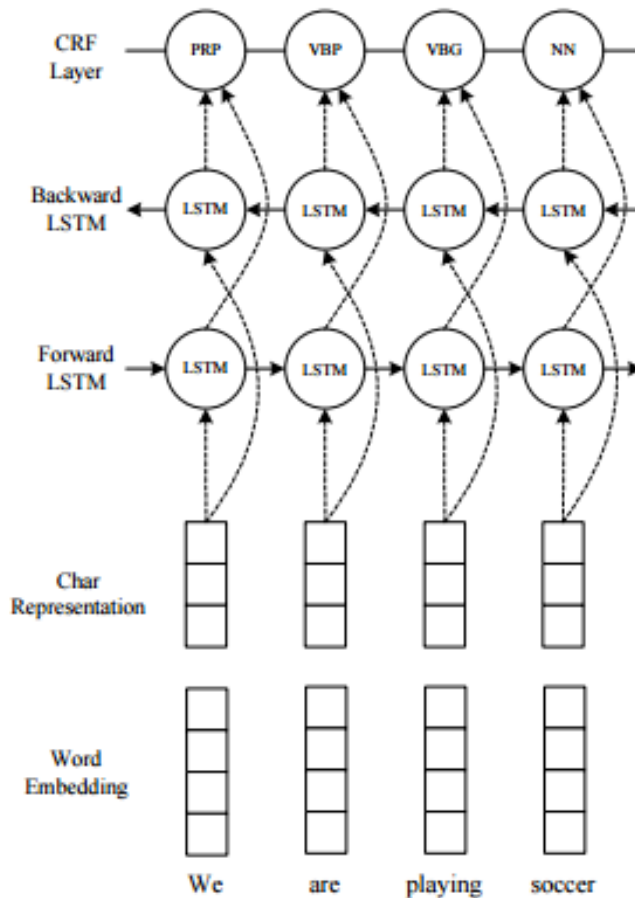
$$y_t = g(U[\vec{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

Equations?

Each memory layer passes an intermediate sequential representation to the next.

# Extensions of simple RNNs

- RNNs with output connections and character information



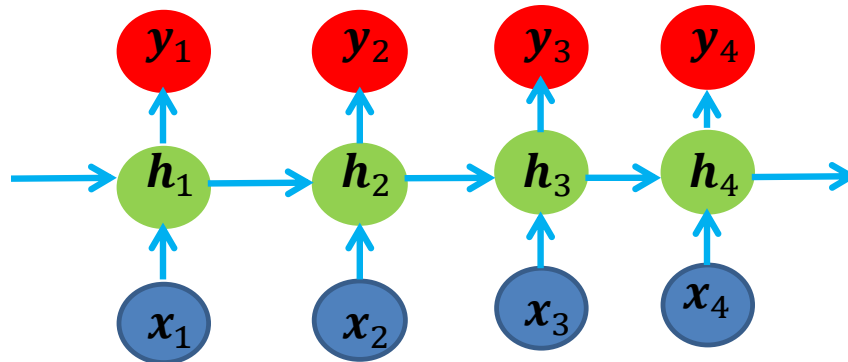
Why character information?  
Ma and Hovy (2016)  
Lample et al. (2016)



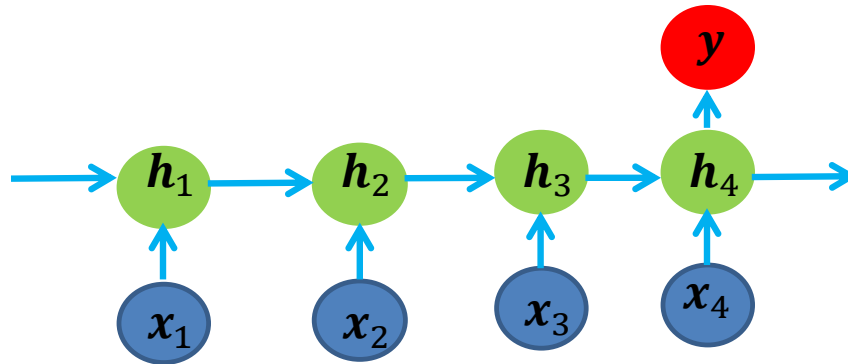
# Recurrent Neural Nets

## For sequence tagging & for classification

# RNNs for sequence tagging (aka sequence labeling)



# RNNs for sentence classification



- Many implementations out there
- Nils Reimers has a nice Keras implementation
  - See: <https://github.com/UKPLab/emnlp2017-bilstm-cnn-crf>
  - He also has one for ELMo embeddings
- We also have a TensorFlow implementation (using Multi-Task Learning, etc.)
  - See: [https://github.com/UKPLab/thesis2018-tk\\_mtl\\_sequence\\_tagging](https://github.com/UKPLab/thesis2018-tk_mtl_sequence_tagging)

# Recurrent Neural Nets

## NLP applications

- RNNs are „natural“ forms for sequence labeling tasks
  - POS tagging

- RNNs are „natural“ forms for sequence labeling tasks
  - POS tagging

<b>x</b> <b>y</b>	<b>We</b>	<b>love</b>	<b>cold</b>	<b>beer</b>
	PRON	V	ADJ	Noun

Label space =  $y = \{\text{PRON}, \text{V}, \text{DET}, \text{ADVERB}, \dots\}$  encoded as 1-hot vectors  
Input space =  $x = \text{natural language words} = \{\text{I}, \text{you}, \text{he}, \text{she}, \text{run}, \dots\}$  encoded as embeddings

- RNNs are „natural“ forms for sequence labeling tasks
  - NER

x y	Angela	Merkel	loves	New	York
	B-PER	I-PER	O	B-LOC	I-LOC

Label space =  $y = \{B\text{-PER}, I\text{-PER}, O, B\text{-LOC}, I\text{-LOC}, \dots\}$  encoded as 1-hot vectors  
Input space =  $x = \text{natural language words} = \{I, \text{you}, \text{he}, \text{she}, \text{run}, \dots\}$  encoded as embeddings



- RNNs are „natural“ forms for sequence labeling tasks
  - Grapheme-to-Phoneme Conversion (s c h u h  $\rightarrow$  S U:)

x y	s	c	h	u	h
	S	∅	∅	U:	∅

Label space =  $y = \{S, a, a:, \emptyset, \dots\}$  encoded as 1-hot vectors  
 Input space =  $x = \text{chars} = \{a, b, c, \dots\}$  encoded as char embeddings or 1-hot

- RNNs are „natural“ forms for sequence labeling tasks
  - Lemmatization (g e l i e b t  $\rightarrow$  l i e b e n)

<div style="background-color: red; color: white; padding: 5px; text-align: center;">                     x y                 </div>	g	e	l	i	e	b	t
	Ø	Ø	l	i	e	b	en

Label space =  $y = \{a,b,c,st,en,\dots\} + \{\emptyset\}$  encoded as 1-hot vectors  
 Input space =  $x = \text{chars} = \{a,b,c,\dots\}$  encoded as char embeddings or 1-hot

- RNNs are „natural“ forms for sequence labeling tasks
  - Language Modeling


x y	<SOS>	Here	comes	a	new	year
	Here	comes	a	new	year	<EOS>

Label space =  $y = \{\text{words}\} + \text{padding}$  encoded as 1-hot vectors  
Input space =  $x = \{\text{words}\} + \text{padding}$  encoded as embeddings

# Vanishing Gradients Introduction

- (following mostly the lecture slides of Richard Socher, <https://cs224d.stanford.edu/lectures/CS224d-Lecture8.pdf>)
- See also de Freitas' video: <https://www.youtube.com/watch?v=56TYLaQN4N8>

# The chain rule

- Newton notation:  $f(g(x))' = f'(g(x))g'(x)$
- Leibniz notation:  $\frac{df}{dx} = \frac{df}{dy} \cdot \frac{dy}{dx}$
- In higher dimensions, multiplication becomes scalar product or matrix multiplication
  - And  $\frac{\partial f}{\partial \mathbf{x}}$  is a vector (=gradient) when  $\mathbf{x}$  is a vector:
    - $\frac{\partial f}{\partial \mathbf{x}} = \left( \frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$
  - And  $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$  is a matrix (=Jacobian) when  $\mathbf{y}, \mathbf{x}$  are vectors
    - $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \left( \frac{\partial y_1}{\partial \mathbf{x}}, \dots, \frac{\partial y_m}{\partial \mathbf{x}} \right)$   
  
vector

# Back to RNNs

- RNN formulation:
  - $\mathbf{h}_t = \sigma_H(\mathbf{x}_t \mathbf{U} + \mathbf{h}_{t-1} \mathbf{W})$
  - $\mathbf{y}_t = \sigma_Y(\mathbf{h}_t \mathbf{V})$

- Interested in:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$$

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}} \dots \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k}$$

- Remember:
    - $\mathbf{h}_t = \sigma_H(\mathbf{x}_t \mathbf{U} + \mathbf{h}_{t-1} \mathbf{W})$
  - Hence,
    - $\frac{\partial \mathbf{h}_s}{\partial \mathbf{h}_{s-1}} = \text{diag}(\sigma'_H(\mathbf{x}_s \mathbf{U} + \mathbf{h}_{s-1} \mathbf{W})) \cdot \mathbf{W}$
- Each  $\frac{\partial \mathbf{h}_s}{\partial \mathbf{h}_{s-1}}$  is a matrix (called Jacobian)

- Definition for diag:

- Let  $\mathbf{z} = \mathbf{x}_s \mathbf{U} + \mathbf{h}_{s-1} \mathbf{W}$

- $\text{diag}([z_1 \cdots z_n]) = \begin{bmatrix} z_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & z_n \end{bmatrix}$



- Analyzing the norms of the Jacobians yields:
- $$\left\| \frac{\partial \mathbf{h}_s}{\partial \mathbf{h}_{s-1}} \right\| = \left\| \text{diag}(\sigma'_H(\mathbf{z})) \cdot \mathbf{W} \right\|$$
$$\leq \left\| \text{diag}(\sigma'_H(\mathbf{z})) \right\| \cdot \left\| \mathbf{W} \right\|$$
- Assume  $\beta_H$  is an upper bound for the norm of  $\text{diag}$  and  $\beta_W$  is an upper bound for the norm of  $\mathbf{W}$
- Similarly, assume that the norm of  $\mathbf{Q} = \text{diag}(\sigma'_H(\mathbf{z})) \cdot \mathbf{W}$  is bounded from below by  $\alpha$

- Then:

$$\alpha \leq \left\| \frac{\partial \mathbf{h}_s}{\partial \mathbf{h}_{s-1}} \right\| \leq \left\| \text{diag}(\sigma'_H(\mathbf{z})) \right\| \cdot \left\| \mathbf{W} \right\| \leq \beta_H \beta_W$$

- Thus

$$\alpha^{t-k} \leq \left\| \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \right\| = \left\| \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}} \dots \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \right\| \leq (\beta_H \beta_W)^{(t-k)}$$

- This can become very large (**exploding gradients**) or very small (**vanishing gradients**) quickly (Bengio et al. 1994)
  - If very large:
    - $\frac{\partial E_t}{\partial W} = \sum_{k=1}^t \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial W}$  explodes
  - If very small:
    - $\mathbf{h}_k$  (and all that goes into it) has no effect on  $\mathbf{h}_t$

- Vanishing gradient problem for language models/sequence labeling, etc.
  - Time steps far away are not taken into consideration
- „Jane walked into the room. John walked in too. It was late in the day. John said hi to **XX**“
- „Berlin \_ ( \_ t h e \_ v e r y \_ b e a u t i f u l \_ . . . \_ c a p i t a l \_ o f \_ **XX**“

- A note on the term  $\beta_H$ :
  - $\left\| \text{diag}(\sigma'_H(\mathbf{x}_s \mathbf{U} + \mathbf{h}_{s-1} \mathbf{W})) \right\| \leq \beta_H$

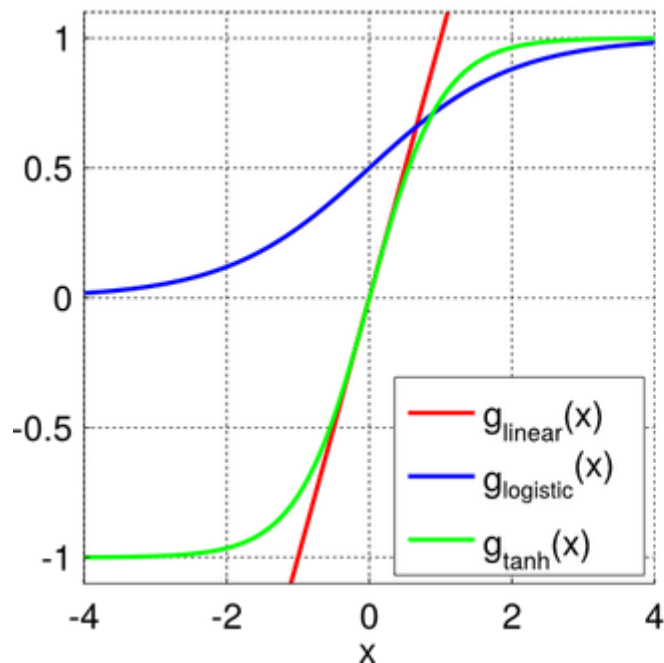
Rule of thumb:

$\sigma' > 1$  exploding gradient

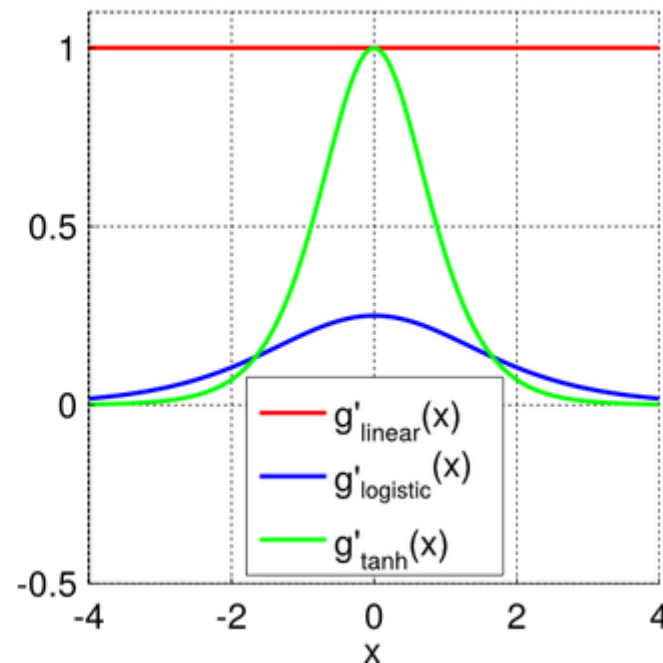
$\sigma' < 1$  vanishing gradient

$\sigma' = 1$  good region

Some Common Activation Functions



Activation Function Derivatives

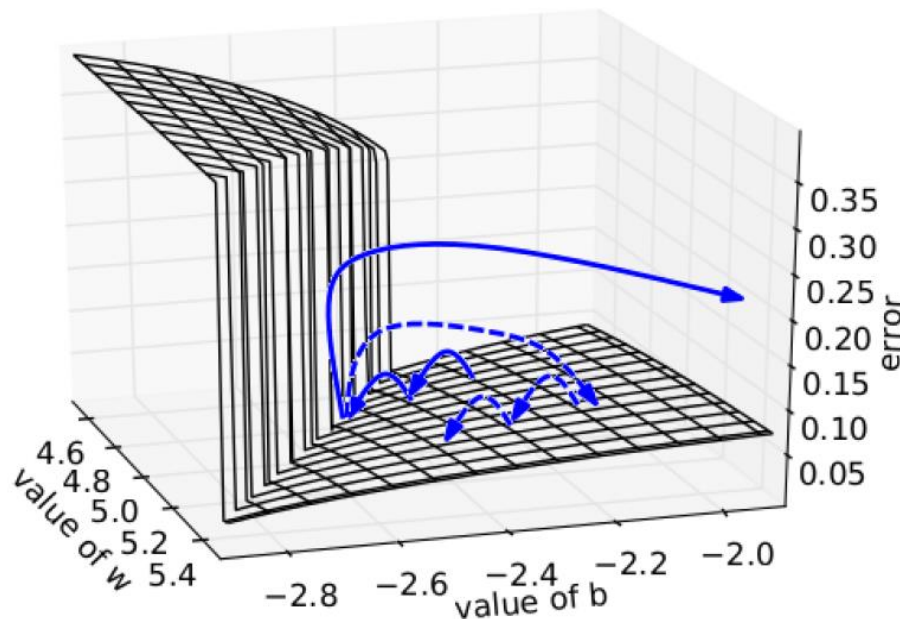


# Vanishing gradients Simple Remedies

- **Exploding gradients:**
  - L1 or L2 regularization on recurrent weights  $\rightarrow$  keeps  $\mathbf{W}$  small
  - Gradient clipping (first introduced by Mikolov)
    - If error derivative  $\frac{\partial E}{\partial w_{ik}}$  is too large, set it to some fixed constant

# Norm clipping

- Gradient clipping intuition:



From: On the difficulty of training RNNs, Pascanu et al. 2013

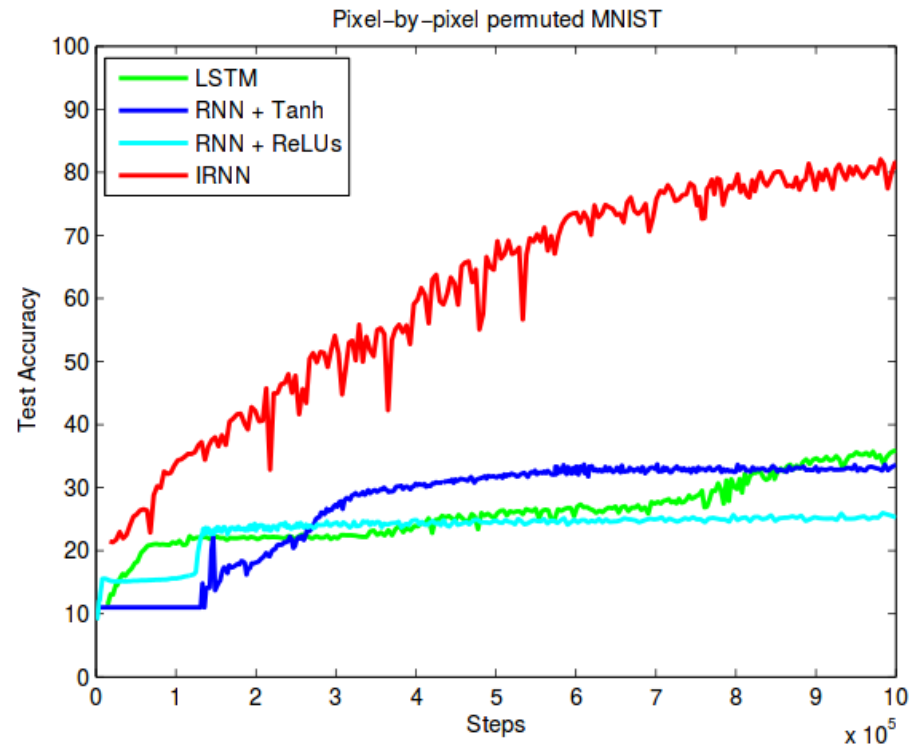
- Solid lines: standard gradient descent trajectories
- Dashed lines: gradients rescaled to fixed size



- **Vanishing (/exploding) gradients**
  - ReLU and initialization, Le et al., 2015
    - Initialize  $\mathbf{W}$ 's to identity matrix  $\mathbf{I}$
    - $\sigma_H(z) = \max(0, z)$

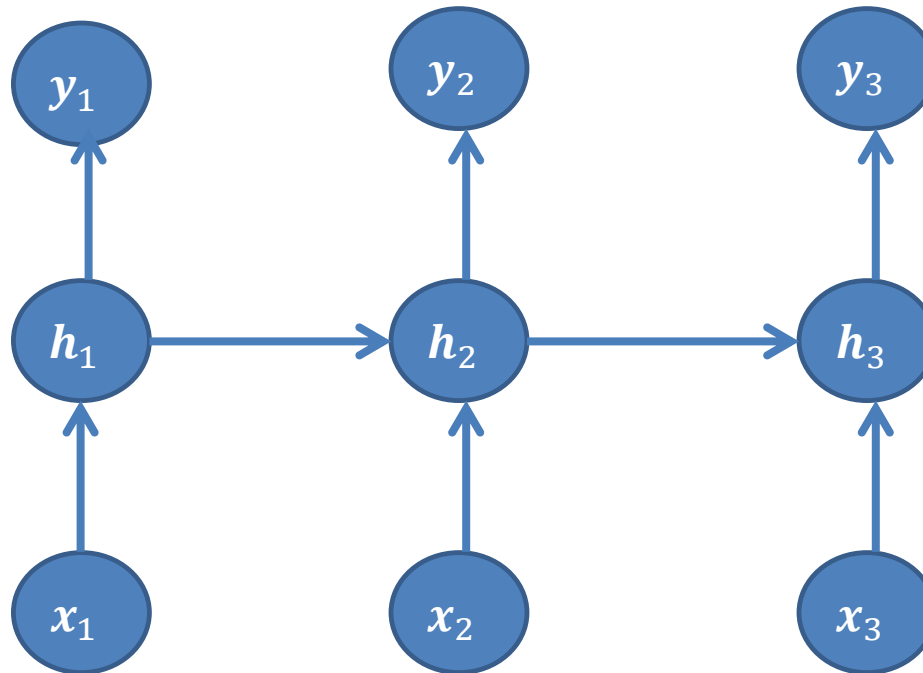
- **Vanishing (/exploding) gradients**

- ReLU and initialization, Le et al., 2015
  - Initialize  $W$ 's to identity matrix  $I$
  - $\sigma_H(z) = \max(0, z)$
  - They call this IRNNs ( $I$  = identity matrix)

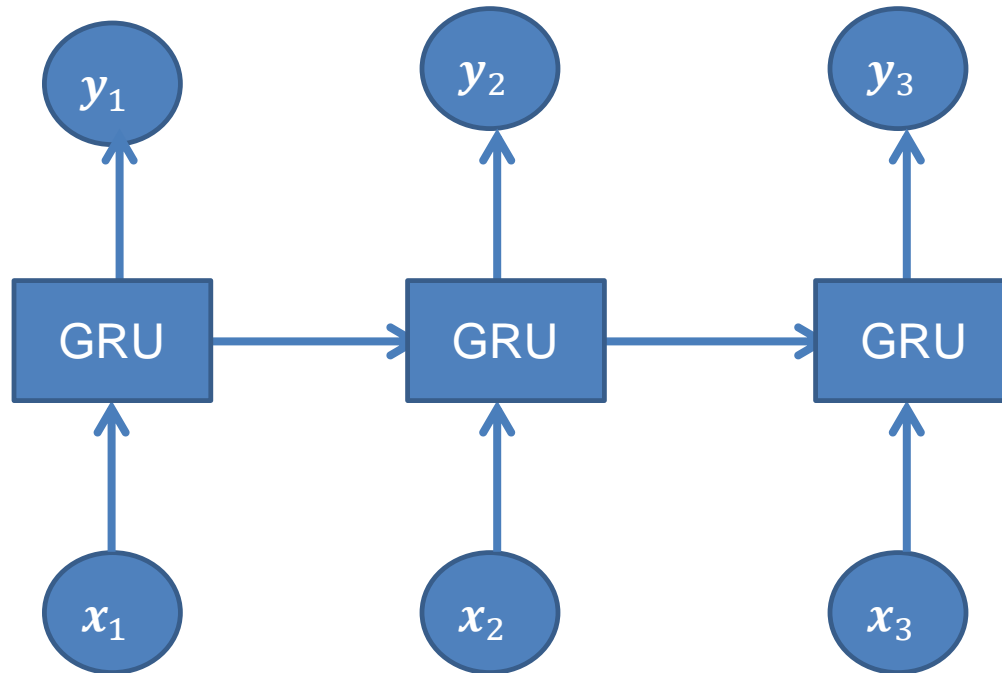


# Vanishing gradients GRUs & LSTMs

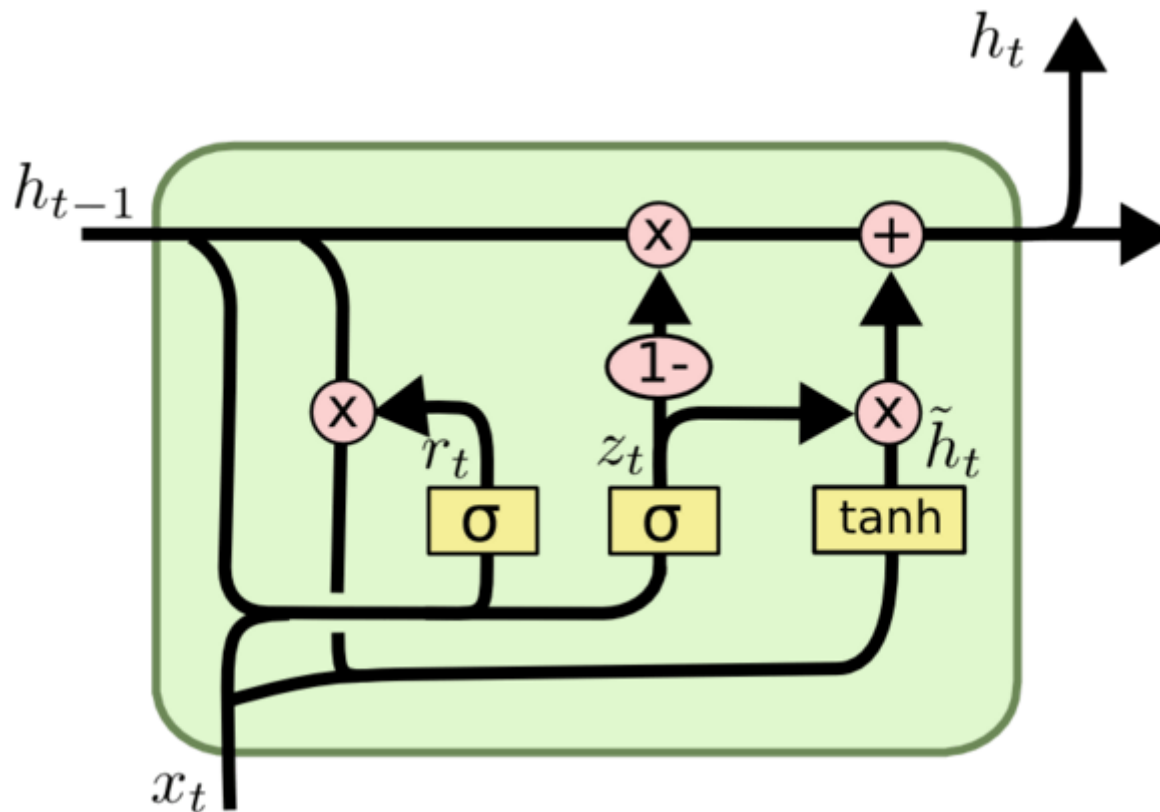
- More complex hidden unit computation in recurrence
- Gated Recurrent Units (GRU) introduced by Cho et al. (2014)
- Main idea:
  - Gates to control the flow of information



# GRUs



# GRU illustration



# Some notation

- Conventions for the following slides
  - $\sigma$  is the sigmoid (=logistic) non-linearity
  - $\odot$  is the *Hadamard* (=point-wise) product
    - $\mathbf{a} \odot \mathbf{b} = (a_1 \cdot b_1, \dots, a_n \cdot b_n)$



- Update gate
  - $\mathbf{z}_t = \sigma(\mathbf{x}_t \mathbf{U}^{(z)} + \mathbf{h}_{t-1} \mathbf{W}^{(z)})$
- Reset gate
  - $\mathbf{r}_t = \sigma(\mathbf{x}_t \mathbf{U}^{(r)} + \mathbf{h}_{t-1} \mathbf{W}^{(r)})$
- New memory content
  - $\tilde{\mathbf{h}}_t = \tanh(\mathbf{x}_t \mathbf{U} + \mathbf{h}_{t-1} \mathbf{W} \odot \mathbf{r}_t)$
- Final memory at time step combines current and previous time steps
  - $\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$

# GRU memory unit - Analysis

- Extreme cases:  $\mathbf{z}_t \in \{0,1\}, \mathbf{r}_t \in \{0,1\}$ 
  - **Note:**  $\mathbf{z}_t$  and  $\mathbf{r}_t$  are vectors, but we look at individual components here

# GRU memory unit - Analysis

- Extreme cases:  $\mathbf{z}_t \in \{0,1\}, \mathbf{r}_t \in \{0,1\}$

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

- $\mathbf{z}_t = 0$ :
  - $\mathbf{h}_t = \mathbf{h}_{t-1} \rightarrow$  no update  $\rightarrow$  can keep memory from previous time step  $\rightarrow$  no vanishing gradient

# GRU memory unit - Analysis

- Extreme cases:  $\mathbf{z}_t \in \{0,1\}, \mathbf{r}_t \in \{0,1\}$

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

- $\mathbf{z}_t = 1$ :
  - $\mathbf{h}_t = \tilde{\mathbf{h}}_t$

# GRU memory unit - Analysis

- Extreme cases:  $\mathbf{z}_t \in \{0,1\}, \mathbf{r}_t \in \{0,1\}$

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

- $\mathbf{z}_t = 1$ :
  - $\mathbf{h}_t = \tilde{\mathbf{h}}_t$
  - $\mathbf{r}_t = 0$ :
    - $\mathbf{h}_t = \tanh(\mathbf{x}_t \mathbf{U}) \rightarrow$  Forget past
  - $\mathbf{r}_t = 1$ :
    - $\mathbf{h}_t = \tanh(\mathbf{x}_t \mathbf{U} + \mathbf{h}_{t-1} \mathbf{W}) \rightarrow$  Standard RNN

$$\begin{aligned} \tilde{\mathbf{h}}_t &= \tanh(\mathbf{x}_t \mathbf{U} + \mathbf{h}_{t-1} \mathbf{W} \odot \mathbf{r}_t) \end{aligned}$$

- Summary:
  - Can store memory at a cell indefinitely
  - Can also forget past memory, and reset everything
  - Can also go back to standard RNN mode, where memory is continuously updated based on past memory and current input

- Can make units even more complex

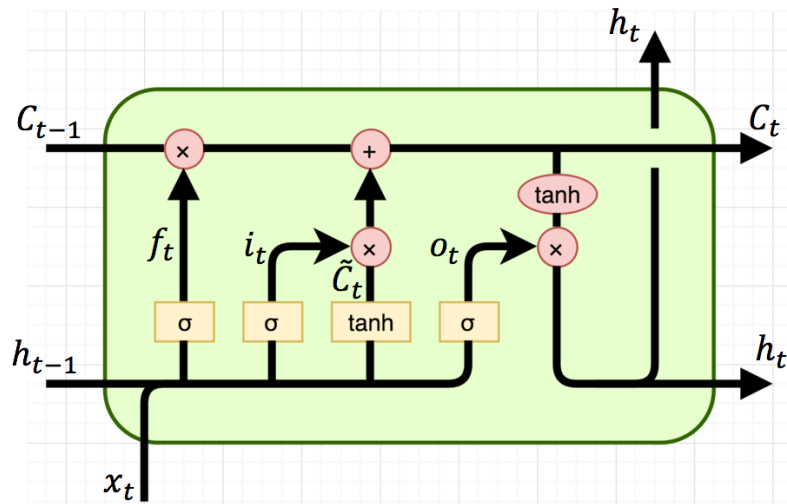
# LSTM (Hochreiter & Schmidhuber, 1997)

- Input gate (= write gate)  $\mathbf{i}_t = F(\mathbf{x}_t, \mathbf{h}_{t-1}; \boldsymbol{\theta}_i)$
- Forget gate (= reset gate)  $\mathbf{f}_t = F(\mathbf{x}_t, \mathbf{h}_{t-1}; \boldsymbol{\theta}_f)$
- Output gate (= read gate)  $\mathbf{o}_t = F(\mathbf{x}_t, \mathbf{h}_{t-1}; \boldsymbol{\theta}_o)$
- New memory cell
  - $\tilde{\mathbf{c}}_t = \tanh(\mathbf{x}_t \mathbf{U} + \mathbf{h}_{t-1} \mathbf{W})$
- Final memory cell
  - $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$
- Final hidden state
  - $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$

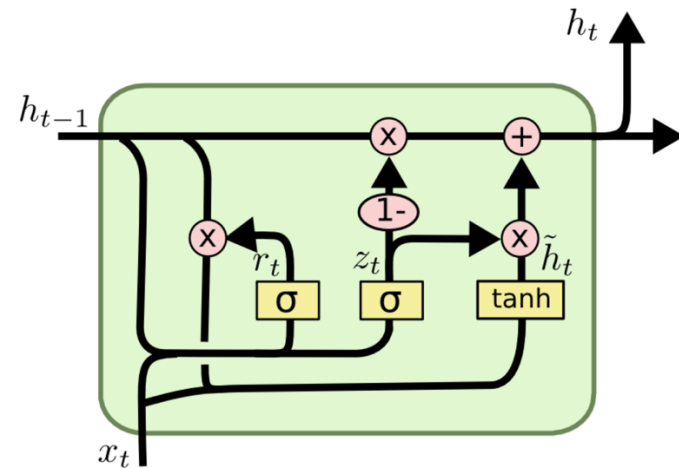
$$F(\mathbf{x}, \mathbf{h}; \boldsymbol{\theta} = [\mathbf{W}, \mathbf{U}]) \\ = \sigma(\mathbf{x}\mathbf{U} + \mathbf{h}\mathbf{W})$$



# LSTM (Hochreiter & Schmidhuber, 1997)



(a) Long Short-Term Memory



(b) Gated Recurrent Unit

# LSTM (Hochreiter & Schmidhuber, 1997)

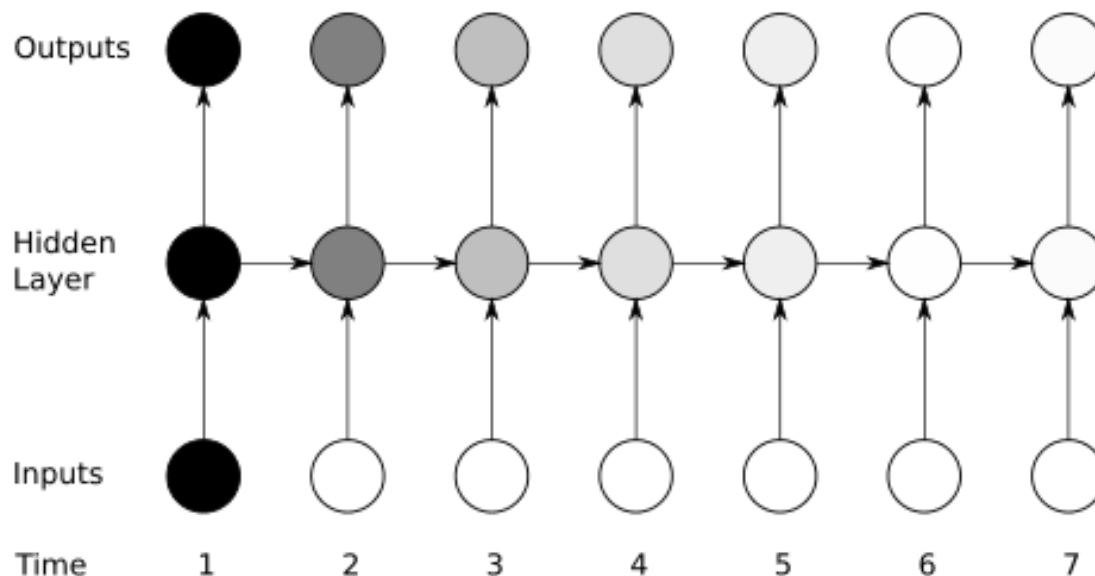


Figure 4.1: **The vanishing gradient problem for RNNs.** The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity). The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network ‘forgets’ the first inputs.

# LSTM (Hochreiter & Schmidhuber, 1997)

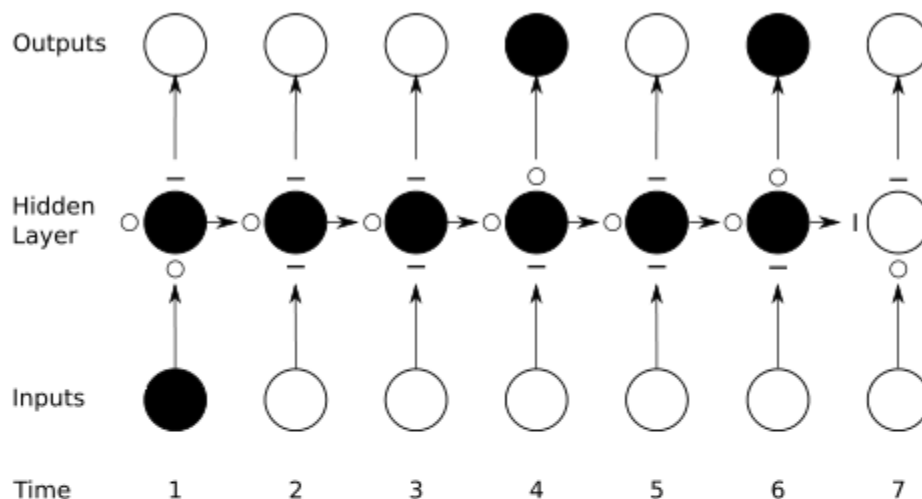


Figure 4.4: **Preservation of gradient information by LSTM.** As in Figure 4.1 the shading of the nodes indicates their sensitivity to the inputs at time one; in this case the black nodes are maximally sensitive and the white nodes are entirely insensitive. The state of the input, forget, and output gates are displayed below, to the left and above the hidden layer respectively. For simplicity, all gates are either entirely open ('O') or closed ('—'). The memory cell 'remembers' the first input as long as the forget gate is open and the input gate is closed. The sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.

# GRU vs. LSTM

- LSTM much more popular (a lot has to do with bias)
- But follows same principles of controlling flow of information via gates

# Summary

- Recurrent Neural Networks are powerful
- Gated Recurrent Units even better
- LSTMs maybe even better
- LSTMs in heavy use until 2018
- Problem: parallelization is difficult
  - Search for more efficient architectures (Transformers)

# References

- Pascanu, R., Mikolov, T., & Bengio, Y.: On the difficulty of training recurrent neural networks. In *Proceedings of the 30<sup>th</sup> International Conference on Machine Learning*, 2013
- Martens, J. (2010), Deep learning via Hessian-free optimization.
- Martens, J., & Sutskever, I.: Learning recurrent neural networks with Hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning*, 2011
- Le, Q. V., Jaitly, N., & Hinton, G. E.: A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015
- Cho, K., van Merriënboer, B., Bahdanau, D., & Bengio, Y.: On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *Syntax, Semantics and Structure in Statistical Translation*, 2014
- Hochreiter, S., & Schmidhuber, J.: Long short-term memory. In *Neural computation*, 1997
- Ma and Hovy (2016), End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF
- Lample et al. (2016), Neural Architectures for Named Entity Recognition
- Sutskever et al. (2013), On the importance of initialization and momentum in deep learning

# **Presentations and Term Papers**

- How to make good presentations
  - Please see the videos on the course github
  - In general:
    - Don't make crowded slides
    - Unfolding is a good idea
    - Adding (many) pictures is a good idea



- Further:
  - Know your audience
  - Give illustrations and examples (most important)
  - Explain the setup, give necessary background
  - Detail your experiments and experiment plans
  - Stick to the time limits

- How to make good term papers
  - Check the Latex template, it gives good guidelines
  - Check „Best term papers“ of previous courses of mine
    - E.g. [github.com/steffeneger/dldh/](https://github.com/steffeneger/dldh/) → 2021
  - Add indication **who did what in the experiments and writing**
    - In the term paper itself
  - Make an illustrating example
  - Make your own tables and figures
  - Write proper English