Deep Learning for NLP

Lecture 8 – Recurrent Neural Nets

Dr. Steffen Eger
Wei Zhao
Niraj Pandey

Natural Language Learning Group (NLLG)
Technische Universität Darmstadt

Previous lectures:

- Introduction (MLPs, loss functions, batch size, activation functions, etc.)

- Embeddings – continuous representations of words, letters, sentences, etc.

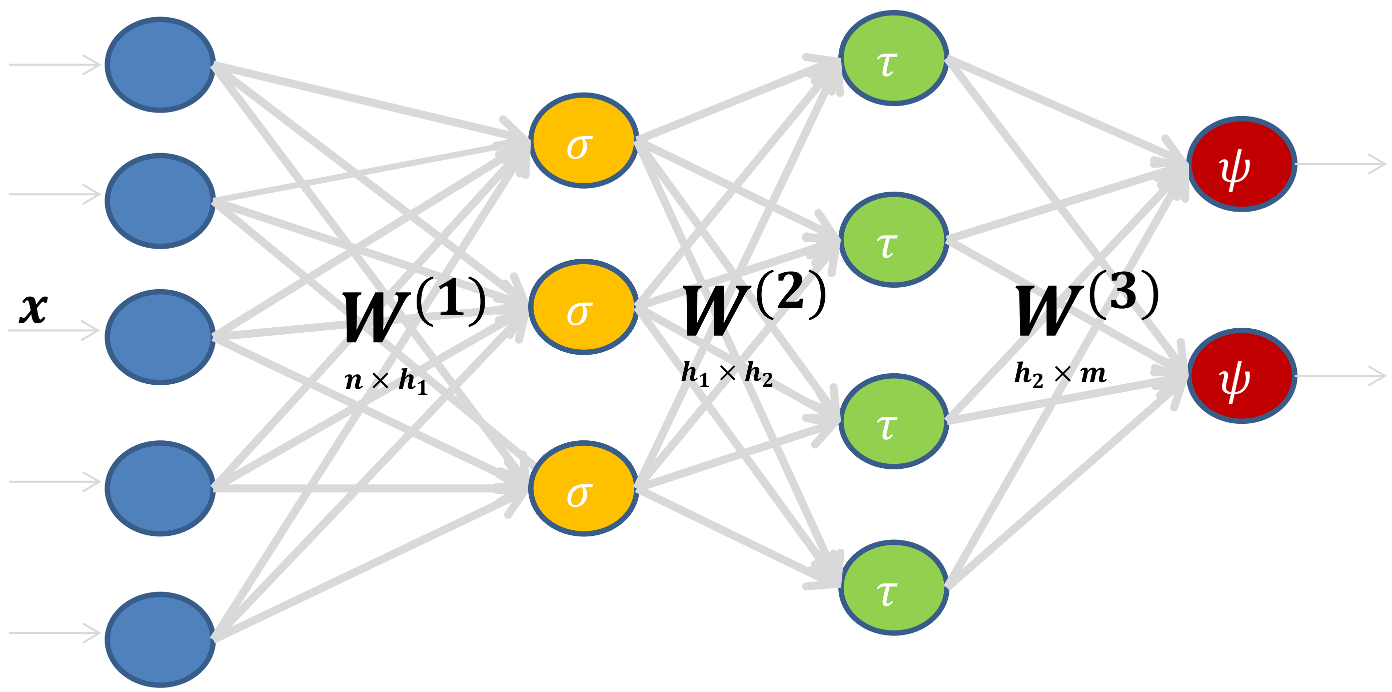
This lecture:

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

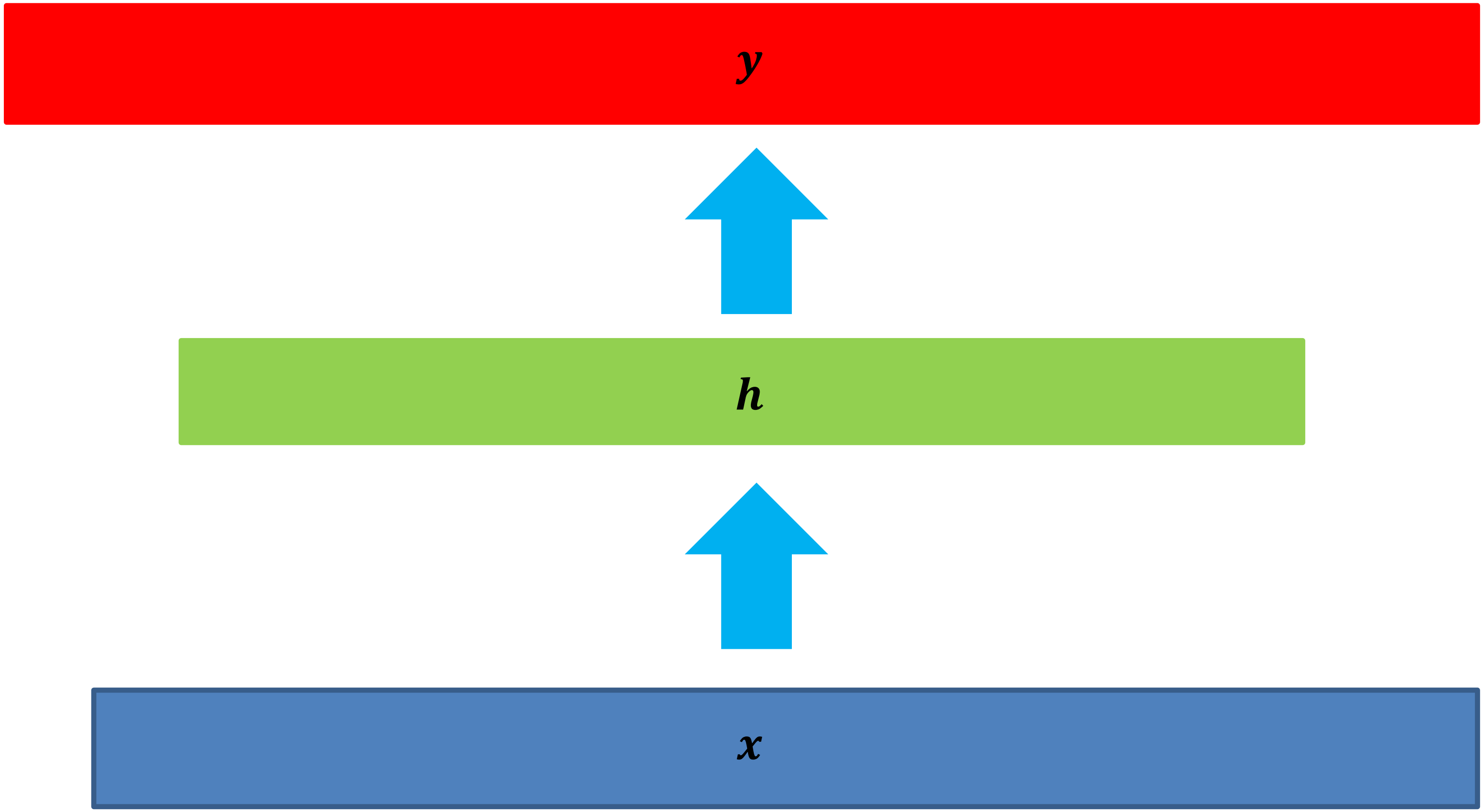
Recurrent Neural Nets

Basic principles

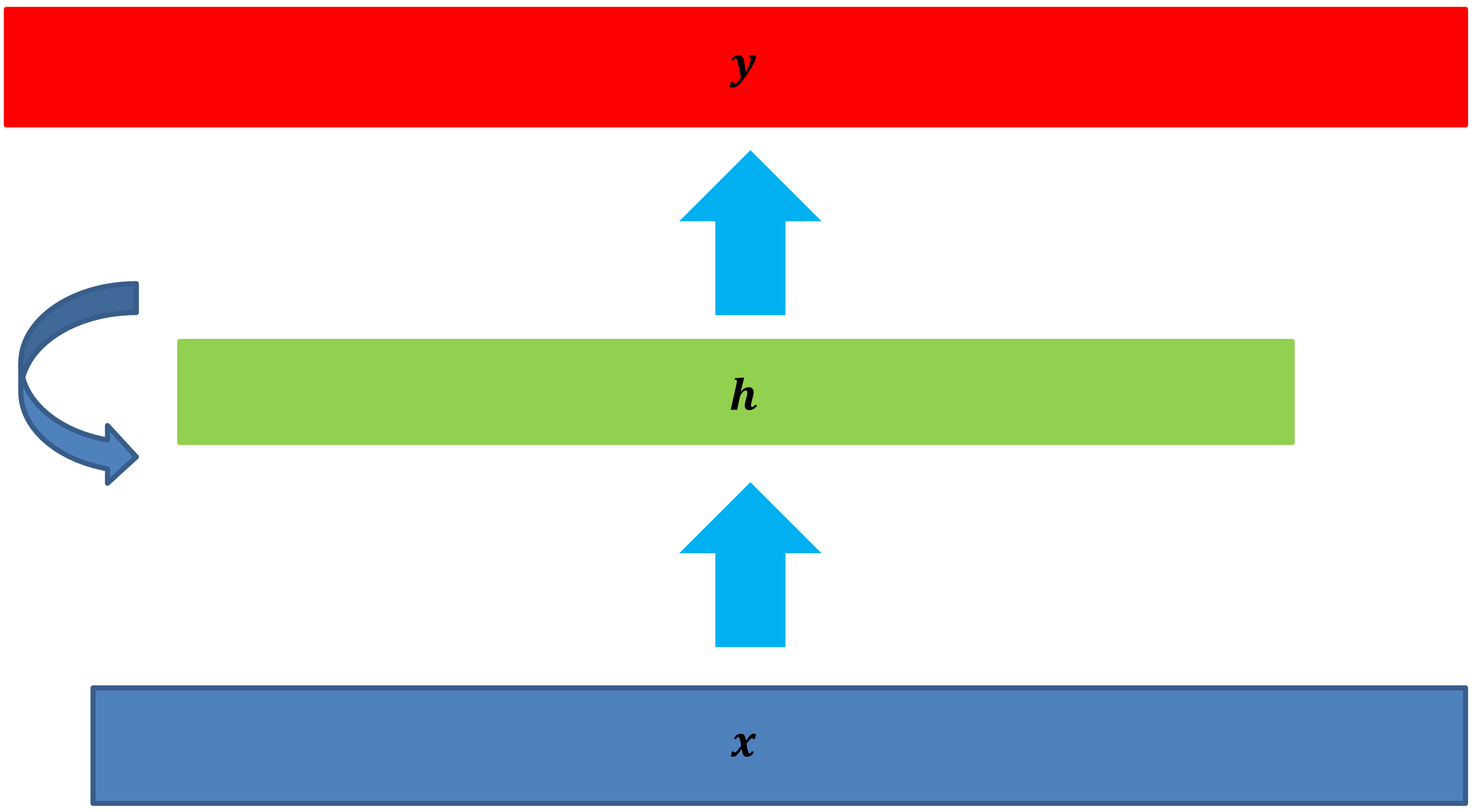
Remember FF Nets / MLP



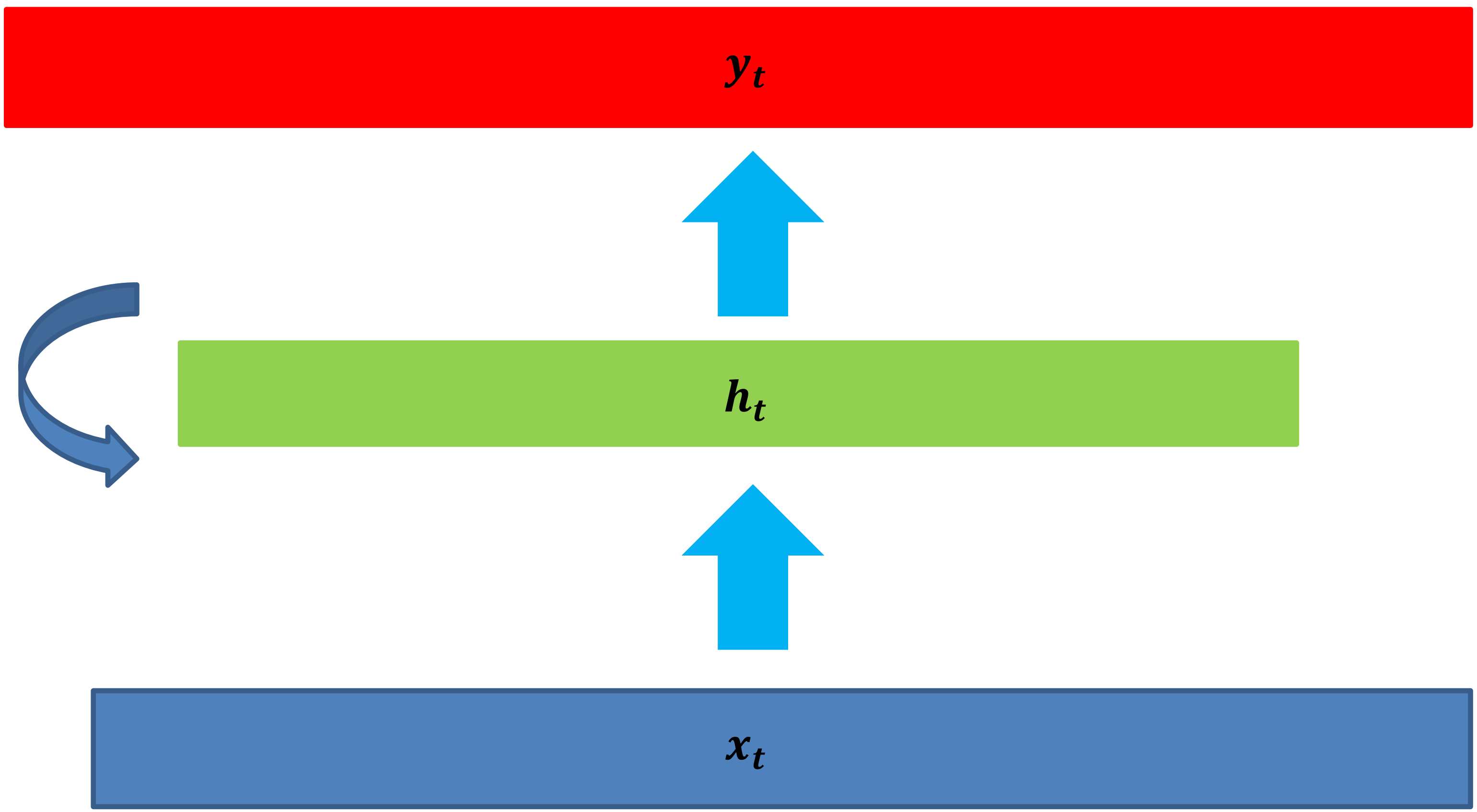
Feedforward / MLP



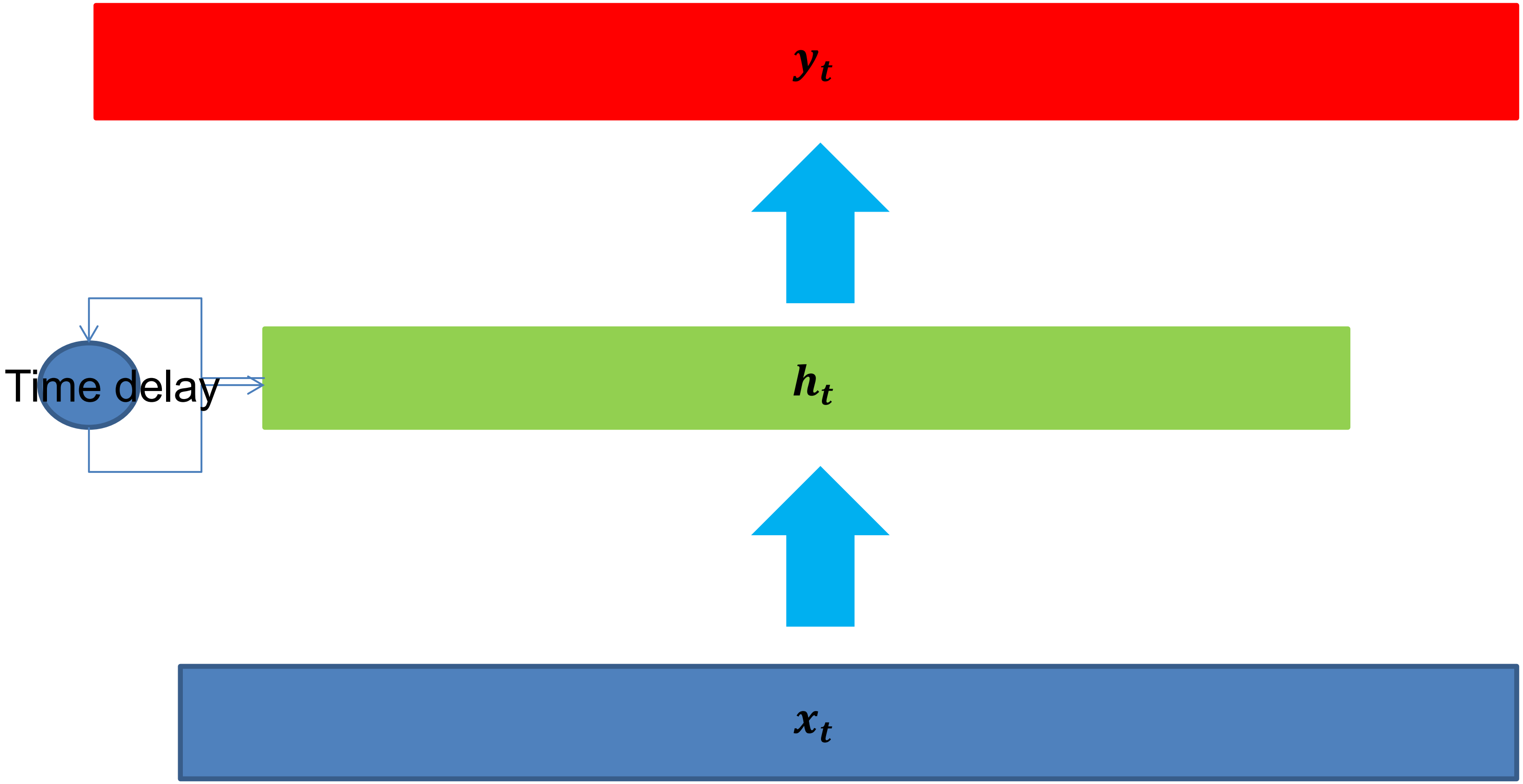
Recurrent



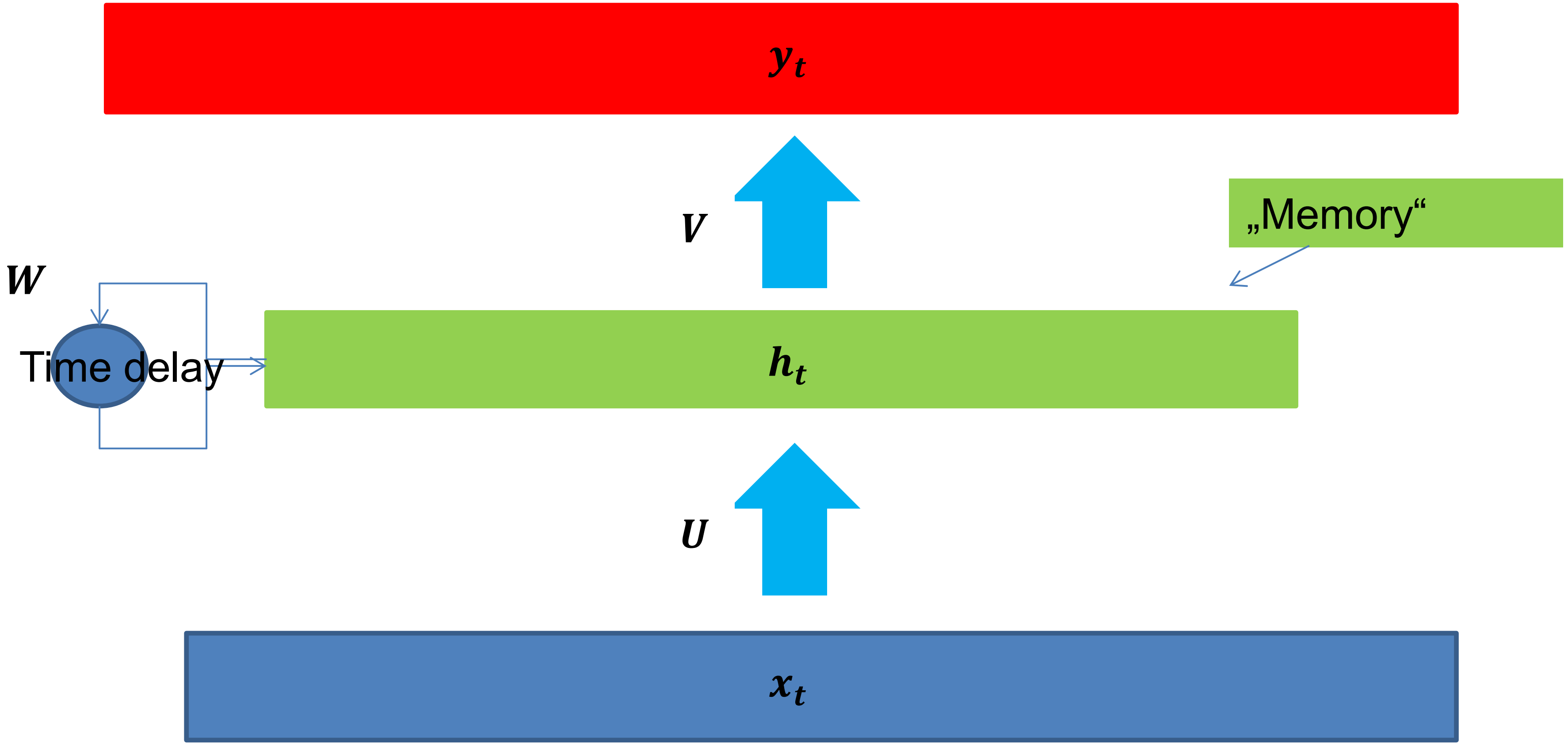
Recurrent



Recurrent



Recurrent



RNN – Formally

- Input vectors $\boldsymbol{x}_t, t = 1,2,3, \ldots$ lie in $R^{1\times n}$

- $\boldsymbol{h}_t = \sigma_H(\boldsymbol{x}_t\boldsymbol{U} + \boldsymbol{h}_{t-1}\boldsymbol{W} + \boldsymbol{b})$
  - Where $\boldsymbol{U} \in R^{n\times d}$, $\boldsymbol{W} \in R^{d\times d}$, $\boldsymbol{h}_t \in R^{1\times d}$
    - d is hidden dimensionality

- $\boldsymbol{y}_t = \sigma_Y(\boldsymbol{h}_t\boldsymbol{V} + \boldsymbol{c})$
  - Where $\boldsymbol{V} \in R^{d\times m}$

RNN – Formally

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

RNN – Example

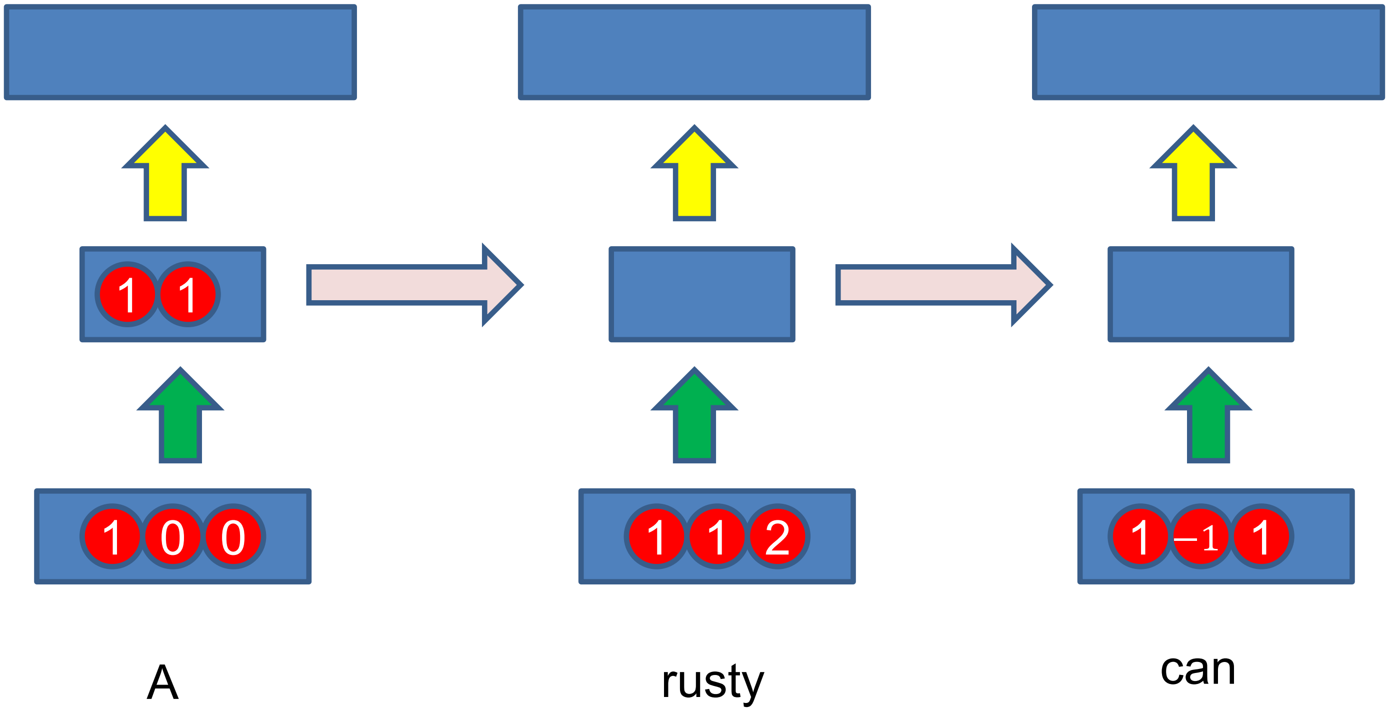
- Input: “A rusty can”
- Embeddings: $\boldsymbol{x}_1 = (1,0,0), \boldsymbol{x}_2 = (1,1,2), \boldsymbol{x}_3 = (1,-1,1)$
- Truth: DET,ADJ,NOUN, encoded as 1-hot vectors (in a 4-d label space)
- Activations: ReLU for hidden layer, Softmax for output layer

RNN – Example

- Initialization:
  - $\boldsymbol{U} = \begin{pmatrix} 1 & 1 \\ 2 & 0 \\ 0.5 & 1 \end{pmatrix}$
  - $\boldsymbol{W} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
  - $\boldsymbol{V} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & \frac{1}{3} & -1 \end{pmatrix}$
  - $\boldsymbol{b} = \boldsymbol{c} =$ zero-vectors of appropriate size
  - $\boldsymbol{h}_0 = (0,0)$

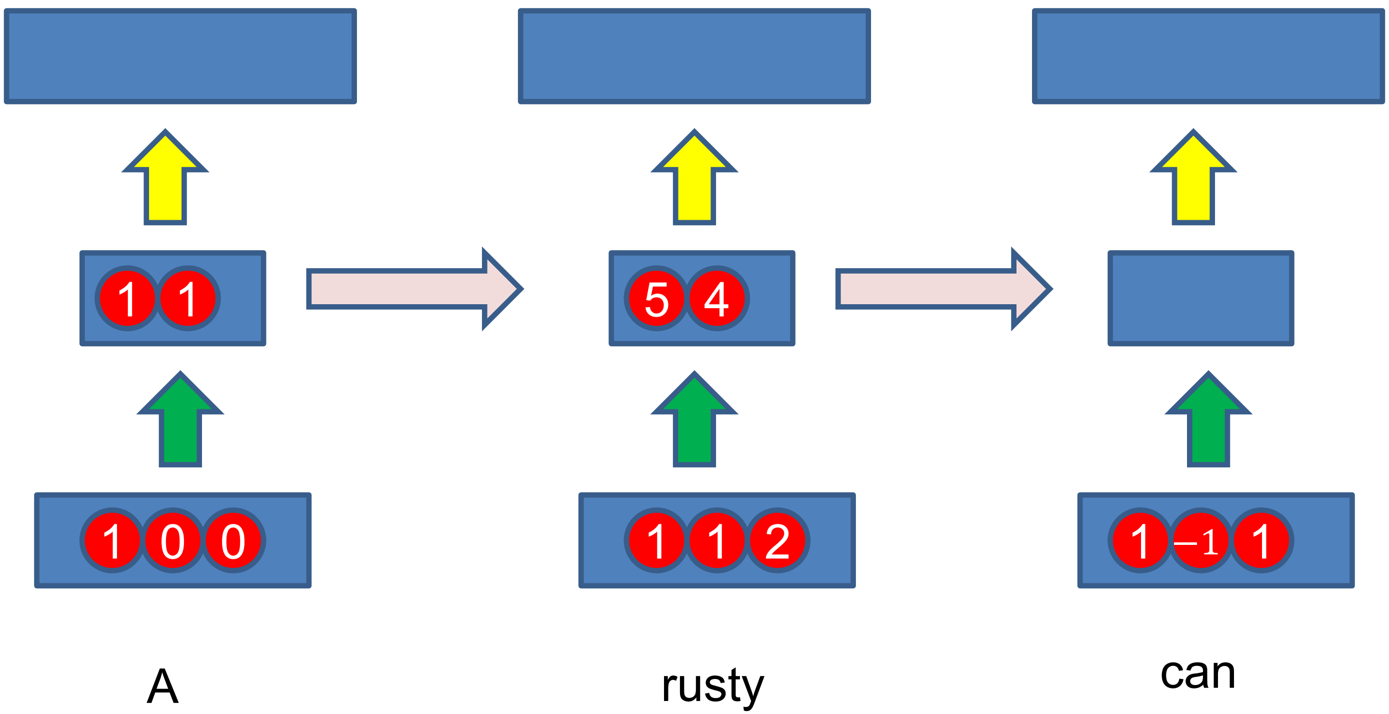
RNN – Example

$$\boldsymbol{h}_1 = \sigma_H(\boldsymbol{x}_1\boldsymbol{U} + \boldsymbol{h}_0\boldsymbol{W} + \boldsymbol{b})$$
$$\boldsymbol{h}_1 = (1,1)$$



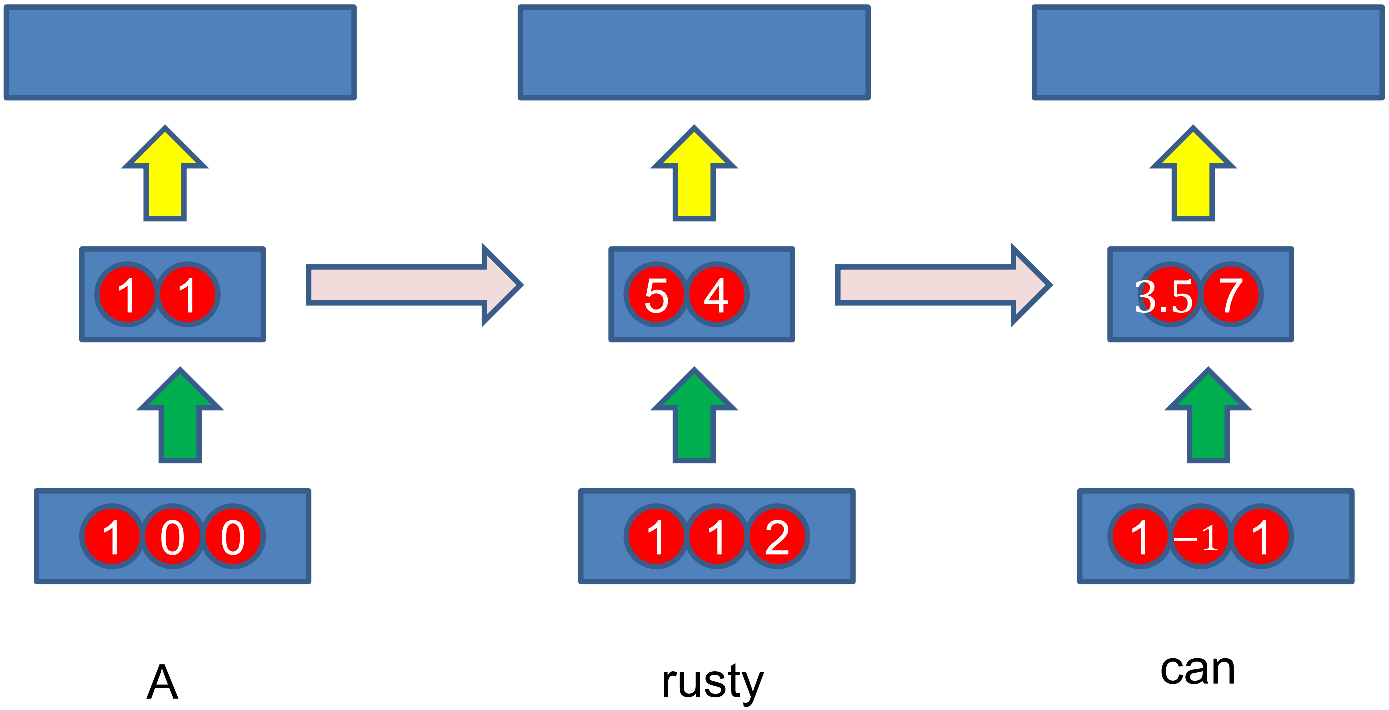
RNN – Example

$$\boldsymbol{h}_2 = \sigma_H(\boldsymbol{x}_2\boldsymbol{U} + \boldsymbol{h}_1\boldsymbol{W} + \boldsymbol{b})$$
$$\boldsymbol{h}_2 = (5{,}4)$$



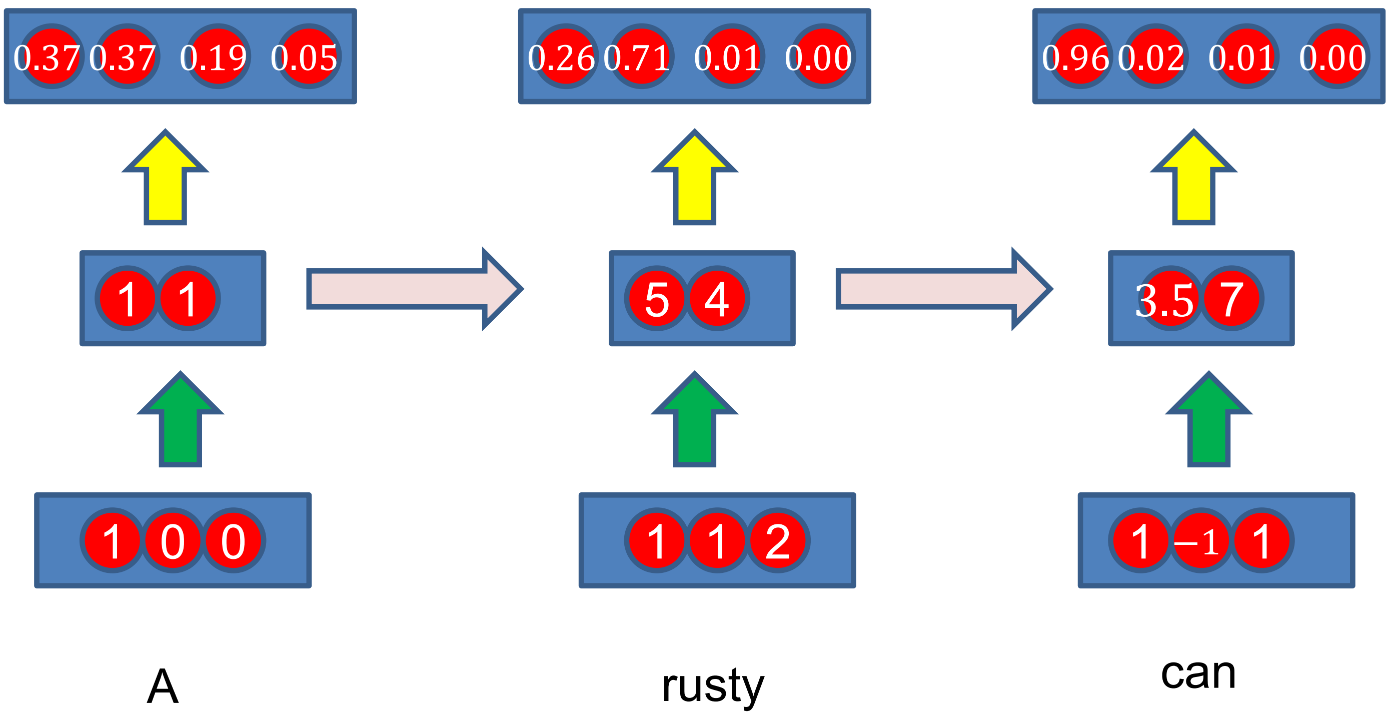
RNN – Example

$$\boldsymbol{h}_3 = \sigma_H(\boldsymbol{x}_3\boldsymbol{U} + \boldsymbol{h}_2\boldsymbol{W} + \boldsymbol{b})$$
$$\boldsymbol{h}_3 = (3.5,7)$$

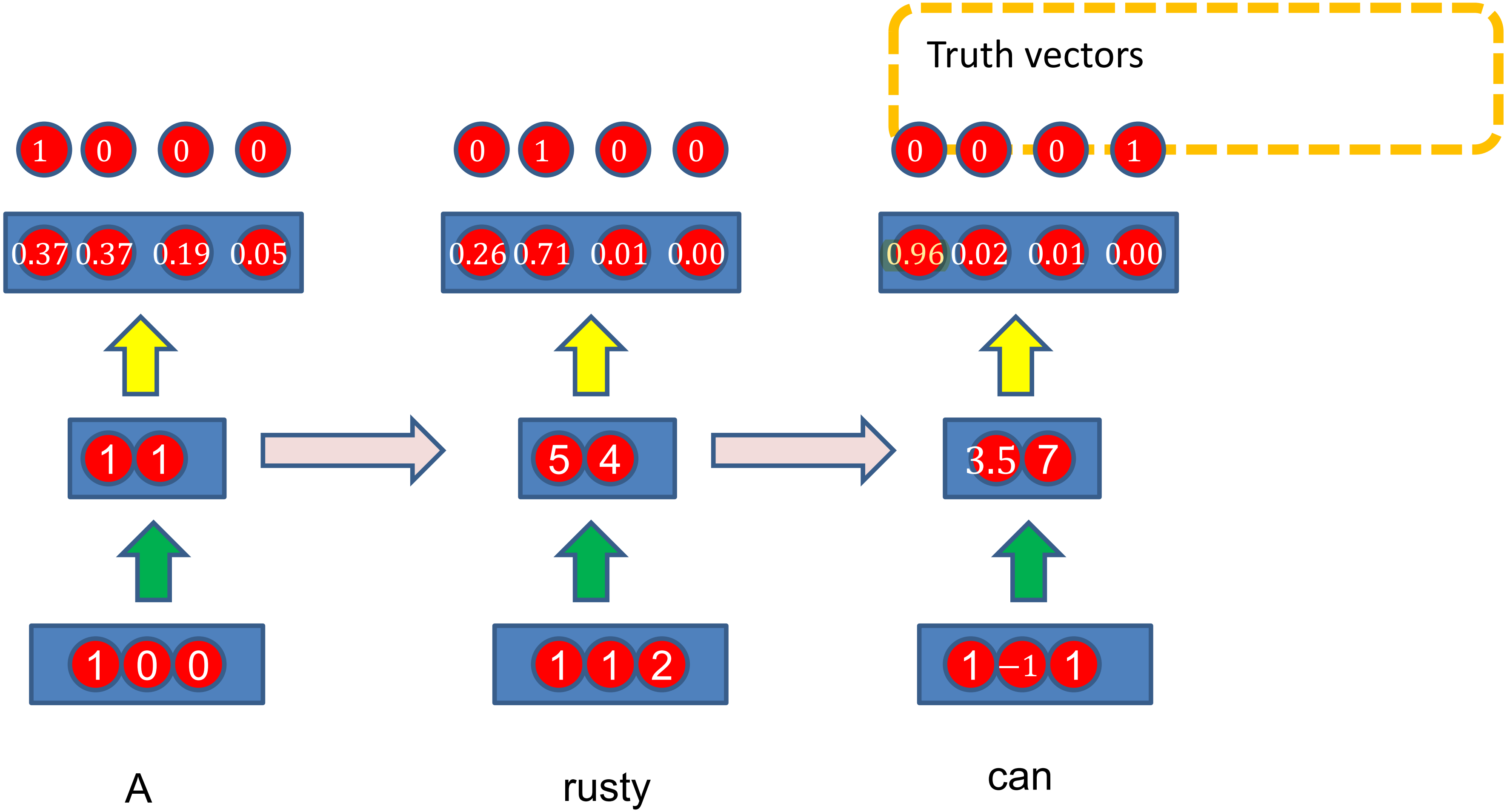


RNN – Example

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

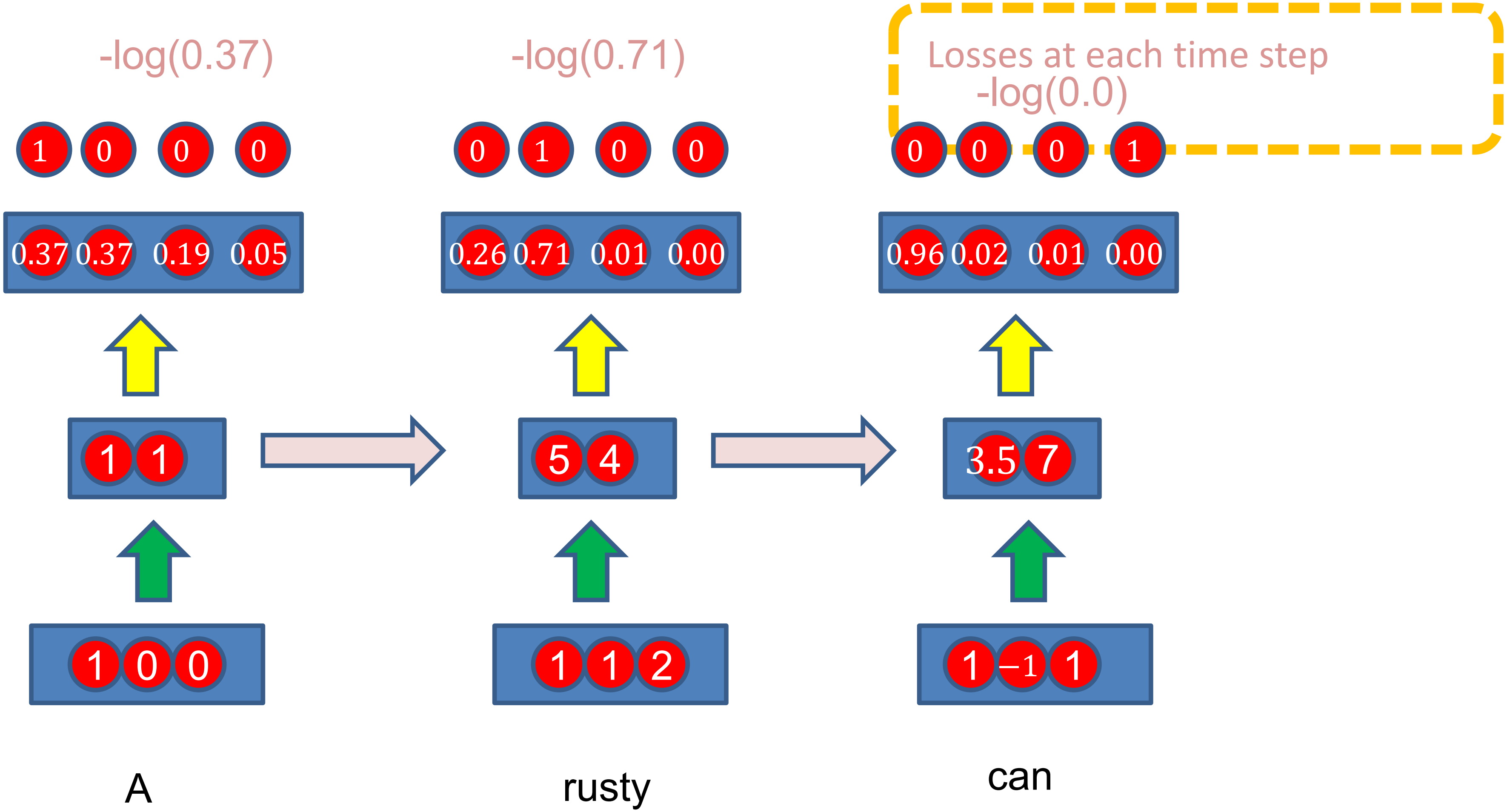


RNN – Example



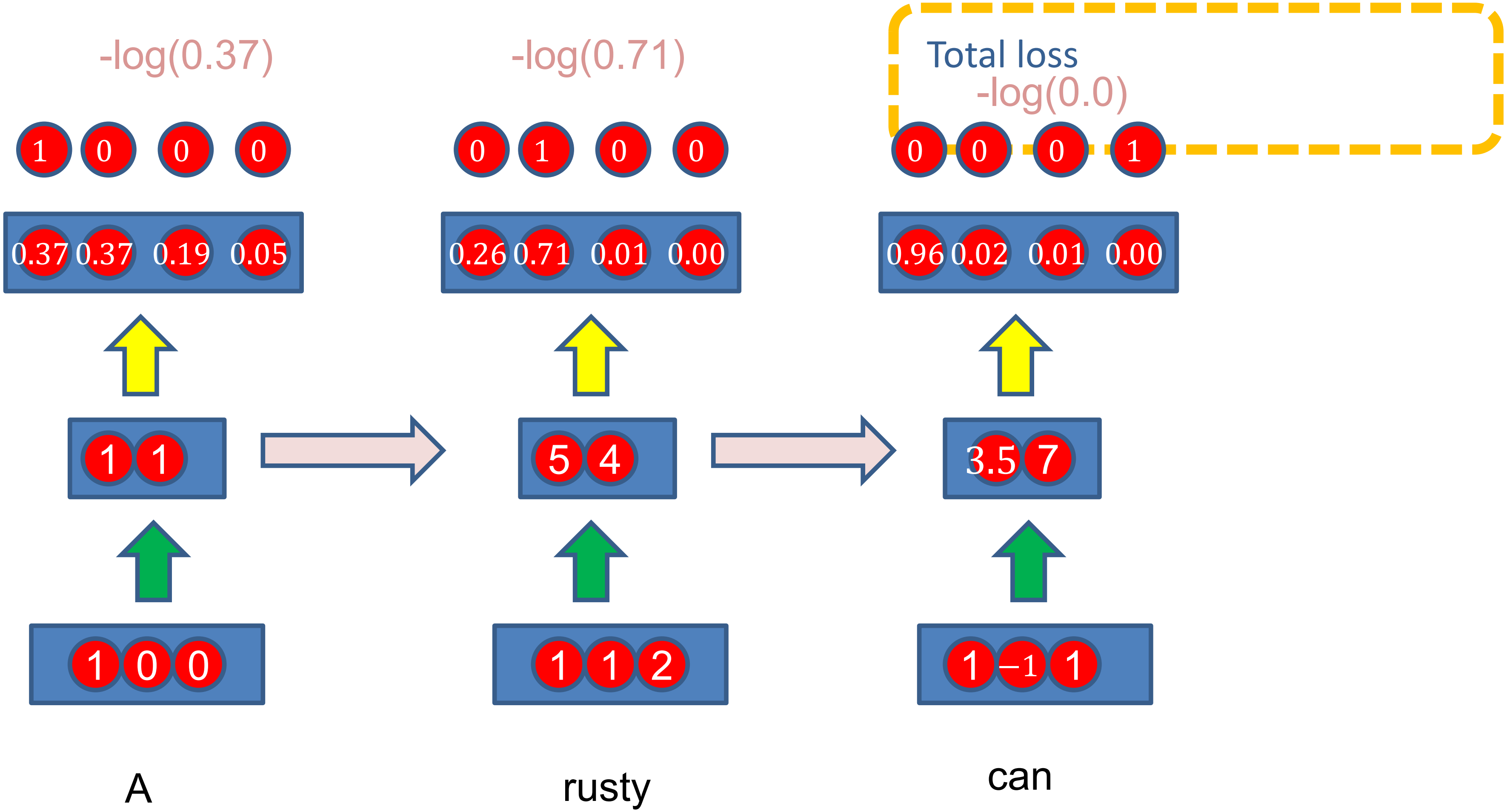
Truth vectors
1 0 0 0
0 1 0 0
0 0 0 1
0.37 0.37 0.19 0.05
0.26 0.71 0.01 0.00
0.96 0.02 0.01 0.00
1 1
5 4
3.5 7
1 0 0
1 1 2
1 –1 1
A
rusty
can

RNN – Example


-log(0.37)
-log(0.71)
Losses at each time step
-log(0.0)
1 0 0 0
0 1 0 0
0 0 0 1
0.37 0.37 0.19 0.05
0.26 0.71 0.01 0.00
0.96 0.02 0.01 0.00
1 1
5 4
3.5 7
1 0 0
1 1 2
1 −1 1
A
rusty
can

RNN – Example

1/3·(-log(0.37)-log(0.71)-log(0.0))

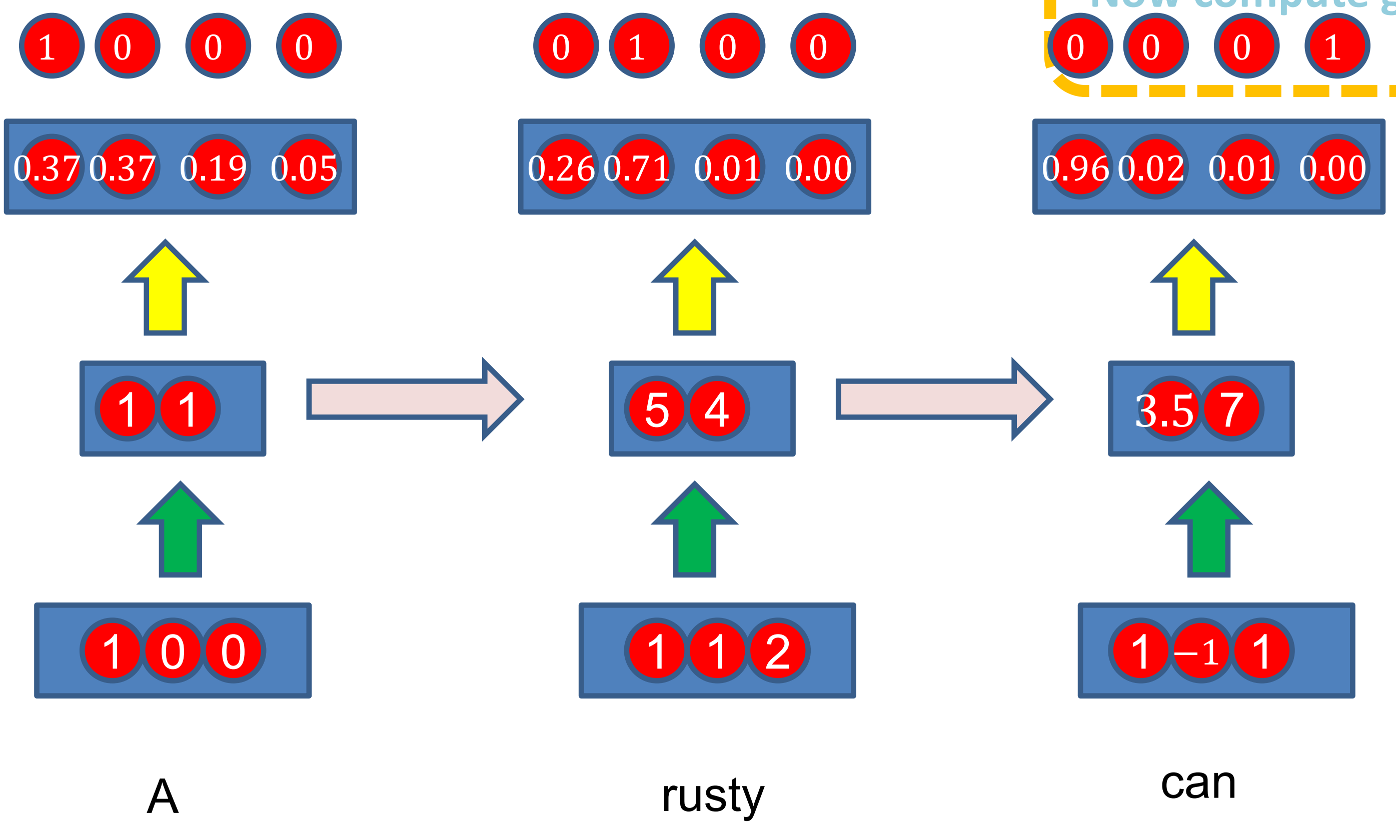


RNN – Example

1/3·(-log(0.37)-log(0.71)-log(0.0))

This was the forward pass

Now compute gradients+update weights



RNN – Weight update / Gradient computation

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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, TF 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 $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha\, \nabla f$ as usual

NLLG

RNN – Properties

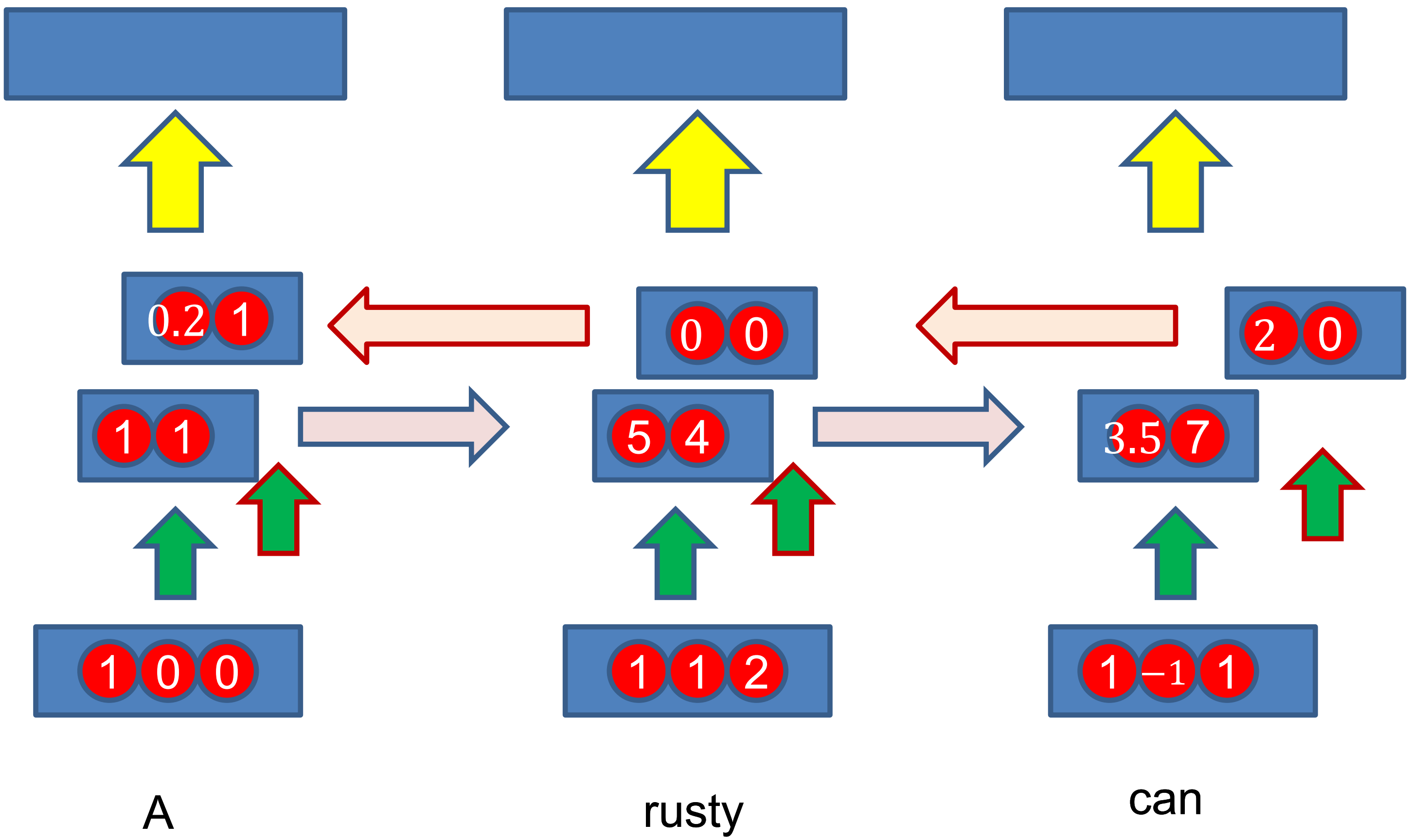
- Infinite window size – “from the left”
  - Memory can (in principle) store everything from the past

- 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
      - $\boldsymbol{h}_t = [\overrightarrow{\boldsymbol{h}}_t; \overleftarrow{\boldsymbol{h}}_t]$
      - Note that $\boldsymbol{V}$ is of dimension $2d \times m$ in this case

Recurrent Neural Nets Extensions

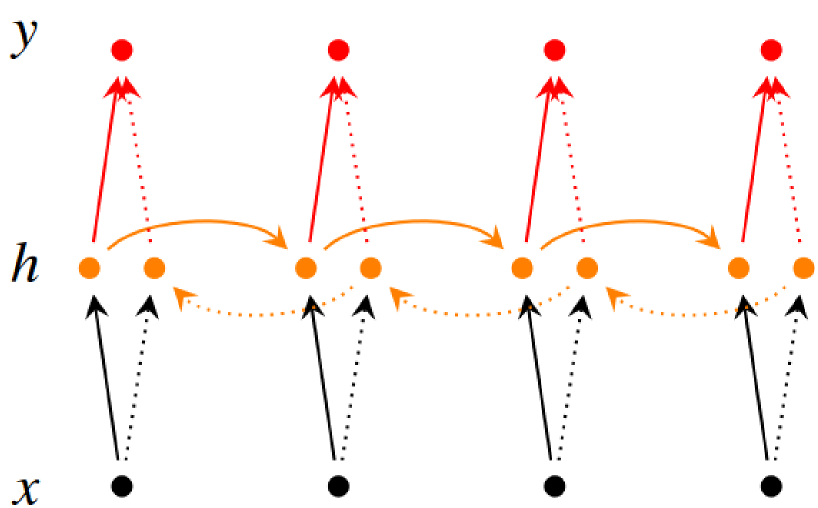
Bidirectional RNN – Illustration



Extensions of simple RNNs

- **Bidirectional RNNs**

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



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

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

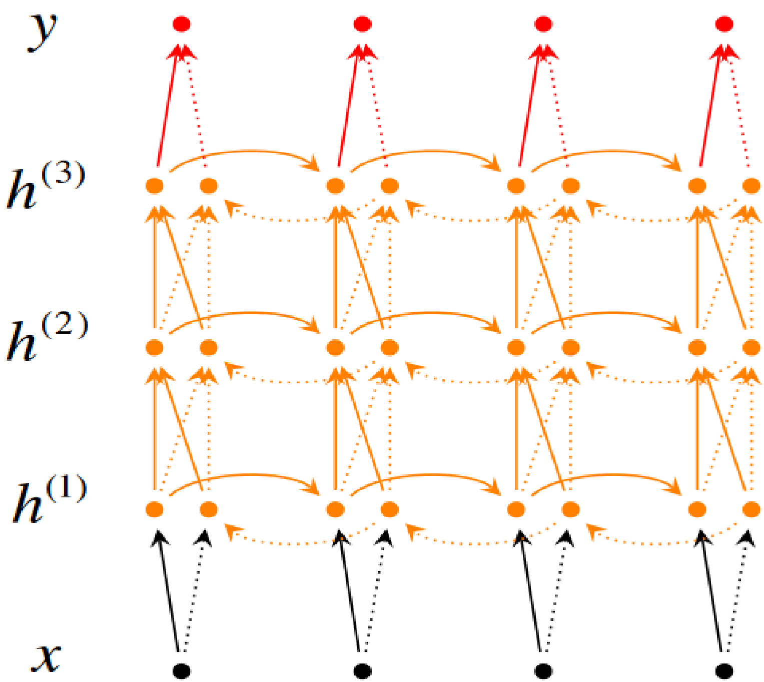
$$y_t = g(U[\overrightarrow{h}_t;\overleftarrow{h}_t] + c)$$

$h = [\overrightarrow{h};\overleftarrow{h}]$ 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**



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

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

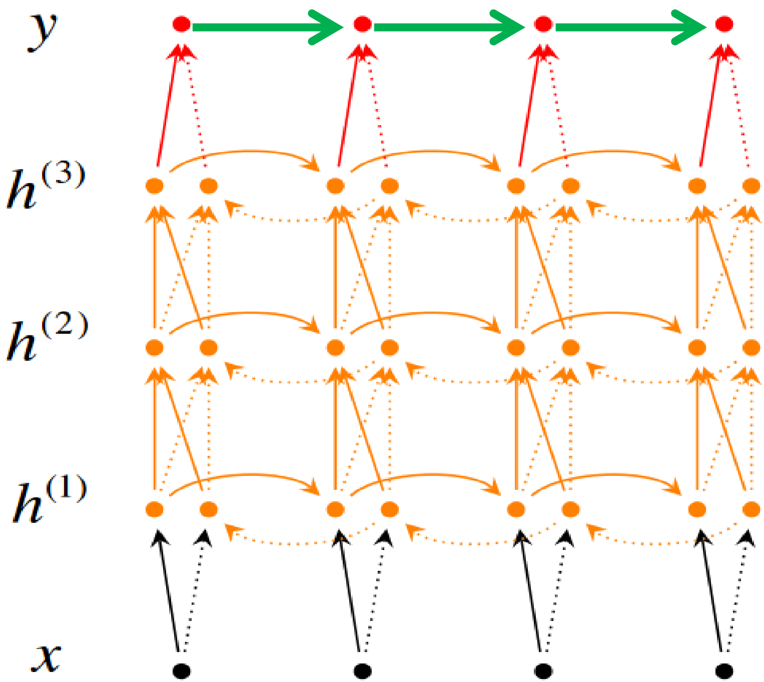
$$y_t = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

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

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

Extensions of simple RNNs

- **RNNs with output connections**



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

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

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

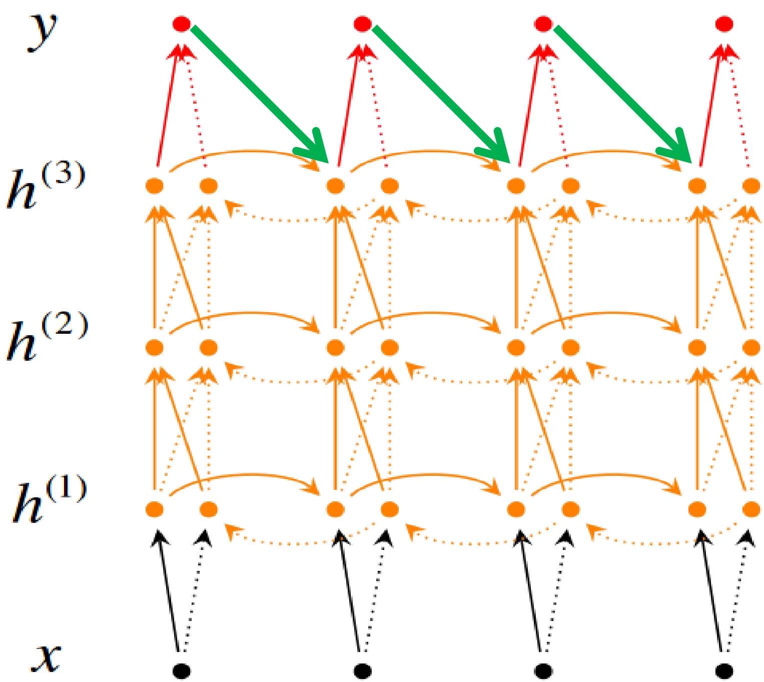
Equations?

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

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

Extensions of simple RNNs

- **RNNs with output connections**



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

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

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

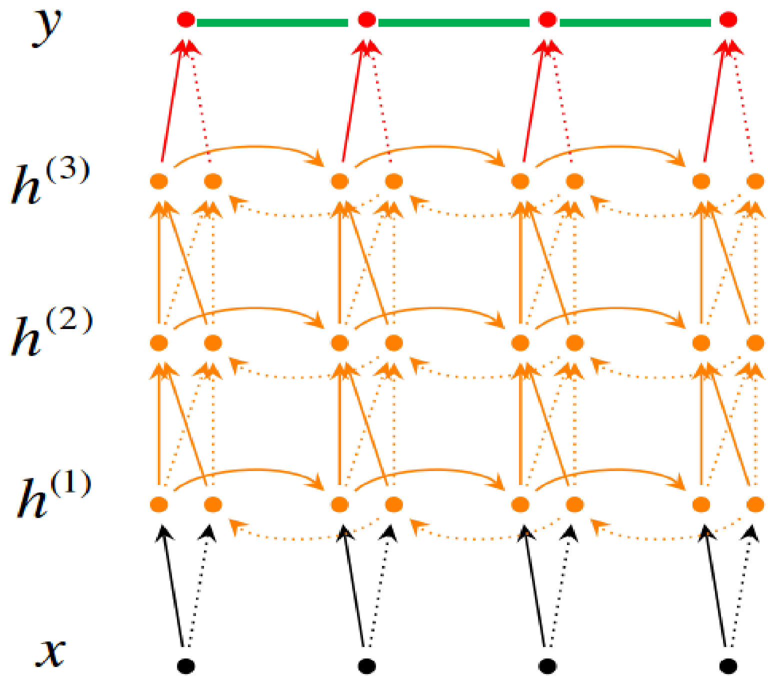
Equations?

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

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

Extensions of simple RNNs

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



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

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

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

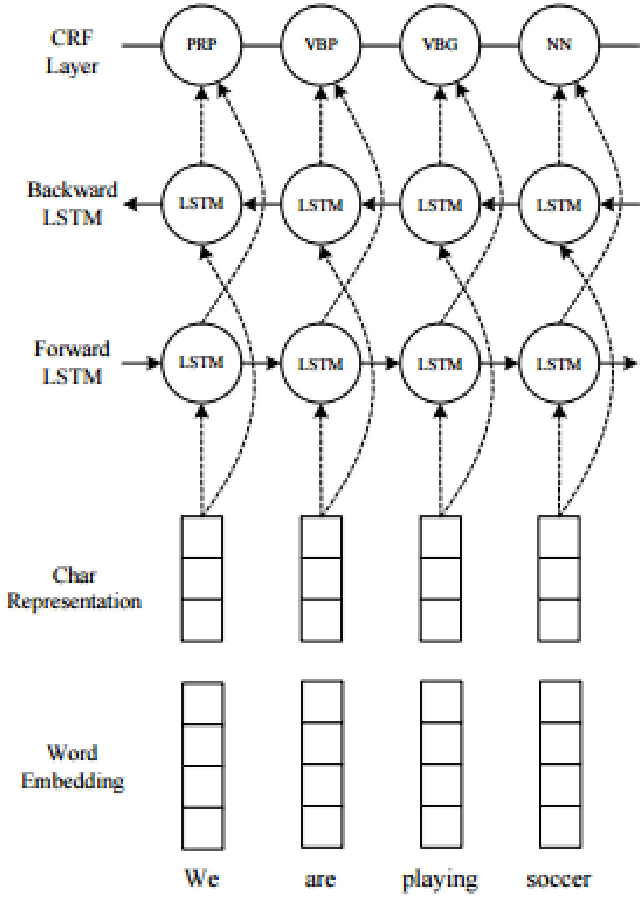
Equations?

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

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

Extensions of simple RNNs

- RNNs with output connections and character information



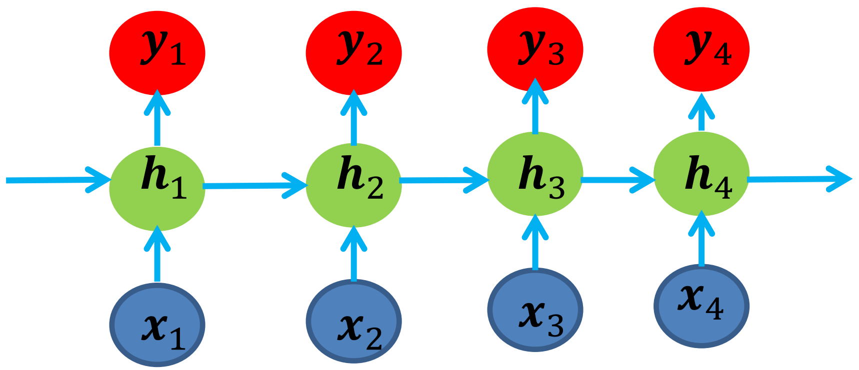
Why character information?
Ma and Hovy (2016)
Lample et al. (2016)

TECHNISCHE UNIVERSITÄT DARMSTADT

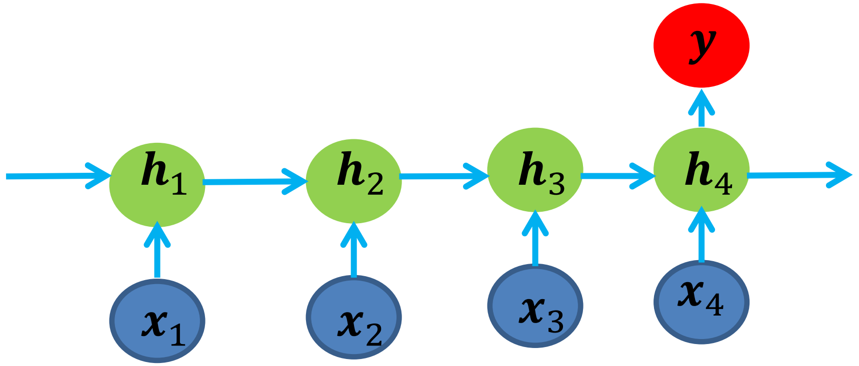
Recurrent Neural Nets
For sequence tagging & for classification

NLLG

RNNs for sequence tagging (aka sequence labeling)



RNNs for sentence classification



Code for RNNs

- Many implementations out there
- Lample et al (2016), Ma and Hovy (2016), and also newer stuff
- 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

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

x y	We	love	cold	beer
	PRON	V	ADJ	Noun

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

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

x	Angela	Merkel	loves	Vladimir	Putin
y	B-PER	I-PER	O	B-PER	I-PER

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

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

x
y

s	c	h	u	h
S	Ø	Ø	U:	Ø

Label space = y = {S,a,a:,Ø,…} encoded as 1-hot vectors
Input space = x = chars = {a,b,c,…} encoded as char embeddings or 1-hot

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

x	g	e	l	i	e	b	t
y	Ø	Ø	l	i	e	b	en

Label space = y = {a,b,c,st,en,…}+{Ø} encoded as 1-hot vectors
Input space = x = chars = {a,b,c,…} encoded as char embeddings or 1-hot

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

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

Label space = y = {words}+padding encoded as 1-hot vectors
Input space = x = {words}+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: $f\big(g(x)\big)' = f'\big(g(x)\big)g'(x)$
- Leibniz notation: $\frac{df}{dx} = \frac{df}{dy} \cdot \frac{dy}{dx}$

- In higher dimensions, multiplication becomes scalar product or matrix multiplication

  - And $\frac{\partial f}{\partial \boldsymbol{x}}$ is a vector (=gradient) when $\boldsymbol{x}$ is a vector:
    - $\frac{\partial f}{\partial \boldsymbol{x}} = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})$
  - And $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is a matrix (=Jacobian) when $\boldsymbol{y}, \boldsymbol{x}$ are vectors
    - $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = (\frac{\partial y_1}{\partial \boldsymbol{x}}, \dots, \frac{\partial y_m}{\partial \boldsymbol{x}})$

vector

Back to RNNs

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

- RNN formulation:
  - $\boldsymbol{h}_t = \sigma_H(\boldsymbol{x}_t \boldsymbol{U} + \boldsymbol{h}_{t-1} \boldsymbol{W})$
  - $\boldsymbol{y}_t = \sigma_Y(\boldsymbol{h}_t \boldsymbol{V})$
  - $E_t = -\log \, \boldsymbol{y}_{t,j}$
  - $E = \sum_{t=1}^{T} E_t$

(Ignoring biases for simplicity)

Index j=j(t) is the true class at time index t

- Total error/loss is the sum of each individual error at time steps t
  - $\frac{\partial E}{\partial \boldsymbol{W}} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial \boldsymbol{W}}$

- Chain rule
  - $\frac{\partial E_t}{\partial \boldsymbol{W}} = \frac{\partial E_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}}$

- We‘ll look at
  - $\frac{\partial E_t}{\partial \boldsymbol{W}} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \boxed{\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k}} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}}$

- We‘ll look at
  - $\frac{\partial E_t}{\partial \boldsymbol{W}} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \boxed{\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k}} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}}$
- Remember:
  - $\boldsymbol{h}_t = \sigma_H(\boldsymbol{x}_t \boldsymbol{U} + \boldsymbol{h}_{t-1} \boldsymbol{W})$

- We'll look at
  - $\frac{\partial E_t}{\partial \boldsymbol{W}} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \boxed{\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k}} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}}$

- Remember:
  - $\boldsymbol{h}_t = \sigma_H(\boldsymbol{x}_t \boldsymbol{U} + \boldsymbol{h}_{t-1} \boldsymbol{W})$

- More chain rule
  - $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{h}_{t-2}} \cdots \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k}$

- We‘ll look at
  - $\frac{\partial E_t}{\partial \boldsymbol{W}} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \boxed{\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k}} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}}$

- Remember:
  - $\boldsymbol{h}_t = \sigma_H(\boldsymbol{x}_t \boldsymbol{U} + \boldsymbol{h}_{t-1} \boldsymbol{W})$

Each $\frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}}$ is a matrix (called Jacobian)

- More chain rule
  - $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{h}_{t-2}} \cdots \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k}$

- From previous slide
  - $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{h}_{t-2}} \cdots \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k}$

Each $\frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}}$ is a matrix (called Jacobian)

- Remember:
  - $\boldsymbol{h}_t = \sigma_H(\boldsymbol{x}_t \boldsymbol{U} + \boldsymbol{h}_{t-1} \boldsymbol{W})$

- Hence,
  - $\frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}} = \mathrm{diag}\big(\sigma_H'(\boldsymbol{x}_s \boldsymbol{U} + \boldsymbol{h}_{s-1} \boldsymbol{W})\big) \cdot \boldsymbol{W}$

- From previous slide
  - $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{h}_{t-2}} \cdots \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k}$

Each $\frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}}$ is a matrix (called Jacobian)

- Remember:
  - $\boldsymbol{h}_t = \sigma_H(\boldsymbol{x}_t \boldsymbol{U} + \boldsymbol{h}_{t-1} \boldsymbol{W})$
- Hence,
  - $\frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}} = \mathrm{diag}\big(\sigma_H'(\boldsymbol{x}_s \boldsymbol{U} + \boldsymbol{h}_{s-1} \boldsymbol{W})\big) \cdot \boldsymbol{W}$
  - Let $\boldsymbol{z} = \boldsymbol{x}_s \boldsymbol{U} + \boldsymbol{h}_{s-1} \boldsymbol{W}$

http://www.atmos.washington.edu/~dennis/MatrixCalculus.pdf

TECHNISCHE UNIVERSITÄT DARMSTADT

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

NLLG

- Analyzing the norms of the Jacobians yields:

- $$\left\|\frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}}\right\| = \left\|\operatorname{diag}\left(\sigma_H'(\boldsymbol{z})\right) \cdot \boldsymbol{W}\right\| \\ \leq \left\|\operatorname{diag}\left(\sigma_H'(\boldsymbol{z})\right)\right\| \cdot \|\boldsymbol{W}\|$$

- Assume β_H is an upper bound for the norm of diag and β_W is an upper bound for the norm of $\boldsymbol{W}$
- Similarly, assume that the norm of $\boldsymbol{Q} = \operatorname{diag}\left(\sigma_H'(\boldsymbol{z})\right) \cdot \boldsymbol{W}$ is bounded from below by α

- Then:
- $\alpha \leq \left\| \frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}} \right\| \leq \left\| \mathrm{diag}\left(\sigma_H'(\boldsymbol{z})\right) \right\| \cdot \left\| \boldsymbol{W} \right\| \leq \beta_H \beta_W$

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

- This can become very large **(exploding gradients)** or very small (**vanishing gradients**) quickly (Bengio et al. 1994)

  - If very large:
    - $\frac{\partial E_t}{\partial \boldsymbol{W}} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{W}} = \infty$
  - If very small:
    - $\boldsymbol{h}_k$ (and all that goes into it) has no effect on $\boldsymbol{h}_t$

- Vanishing gradient problem for language models/sequence labeling, etc.

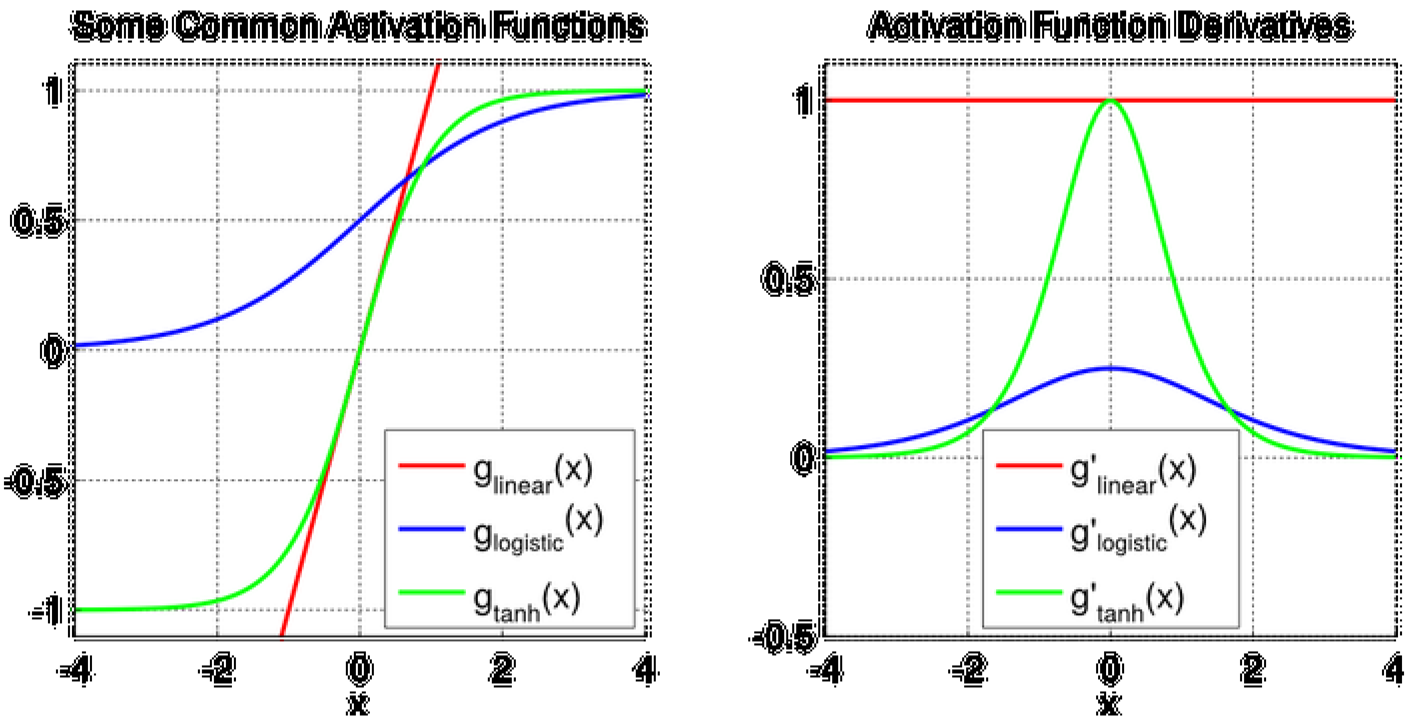
  - Time steps far away are not taken into consideration

- „Jane walked into the room. John walked in too. It was late in the day. Jane said hi to XX“

- „B e r l i n _ (_ 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 β_H:
  - $\left\|\operatorname{diag}\left(\sigma_H'(\boldsymbol{x}_s \boldsymbol{U} + \boldsymbol{h}_{s-1} \boldsymbol{W})\right)\right\| \leq \beta_H$

Rule of thumb:
$\sigma' > 1$ exploding gradient
$\sigma' < 1$ vanishing gradient
$\sigma' = 1$ good region

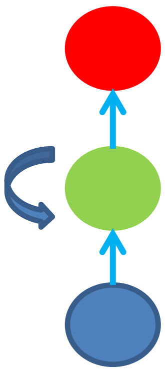


Vanishing gradients in MLPs

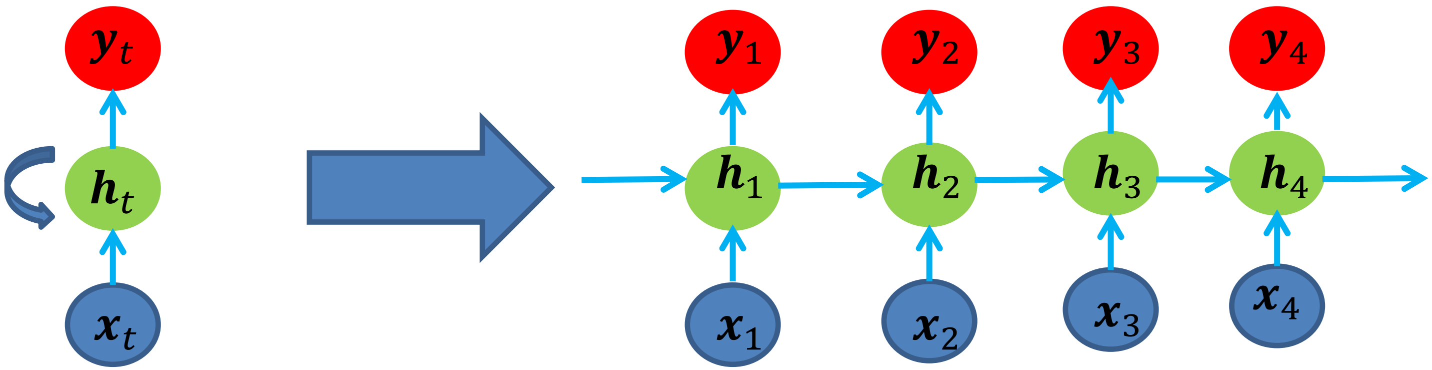
- Note that the vanishing gradient problem is not specific to RNNs
- It occurs in all deep networks, also in deep MLPs

- Also behold that RNNs are a form of deep neural nets:

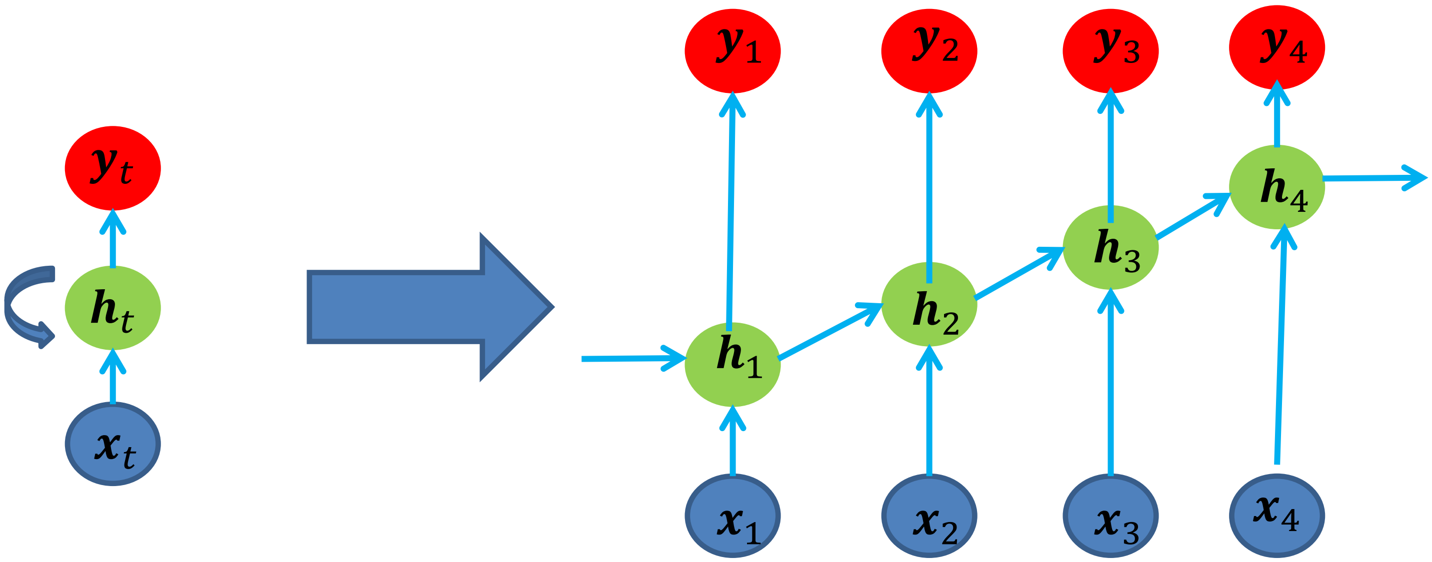
RNNs as deep nets



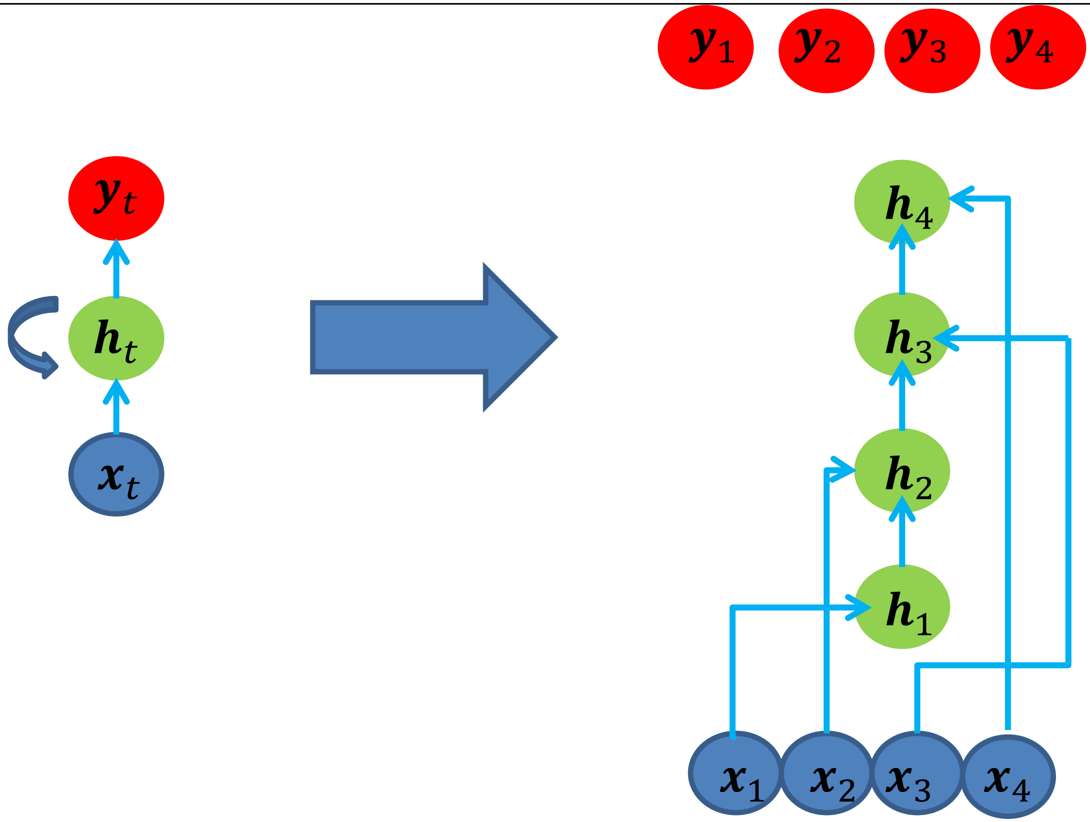
RNNs as deep nets



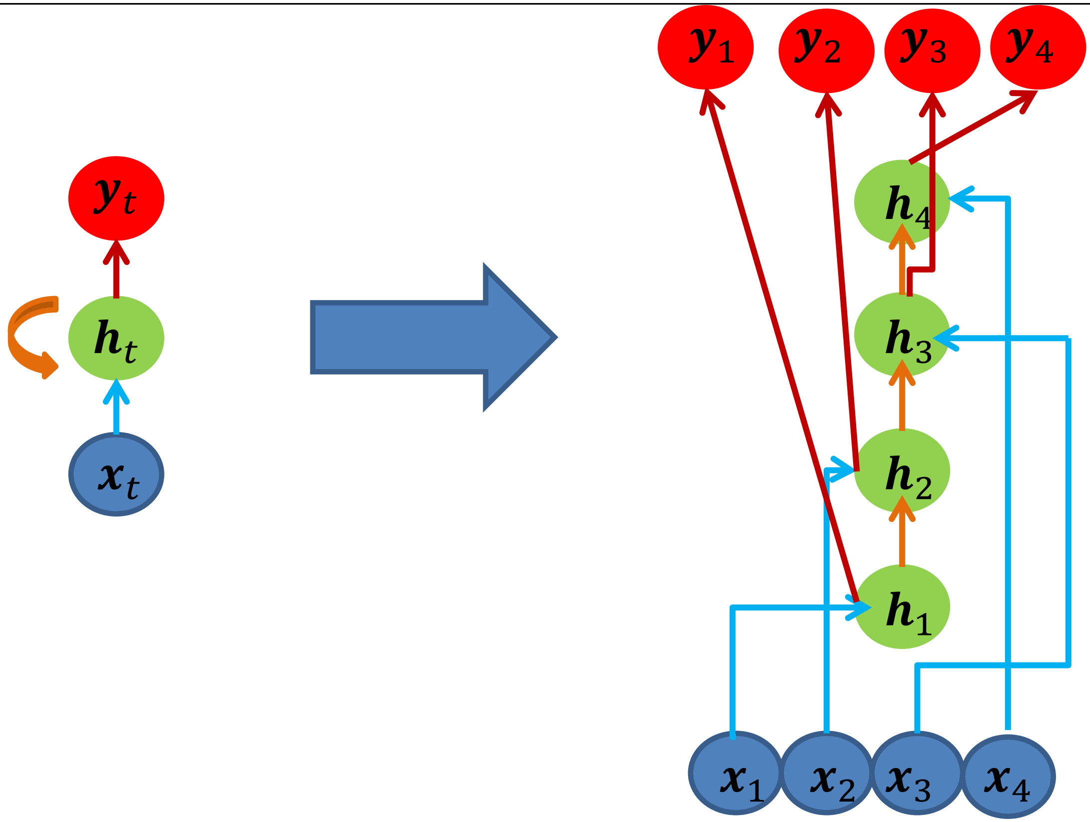
RNNs as deep nets



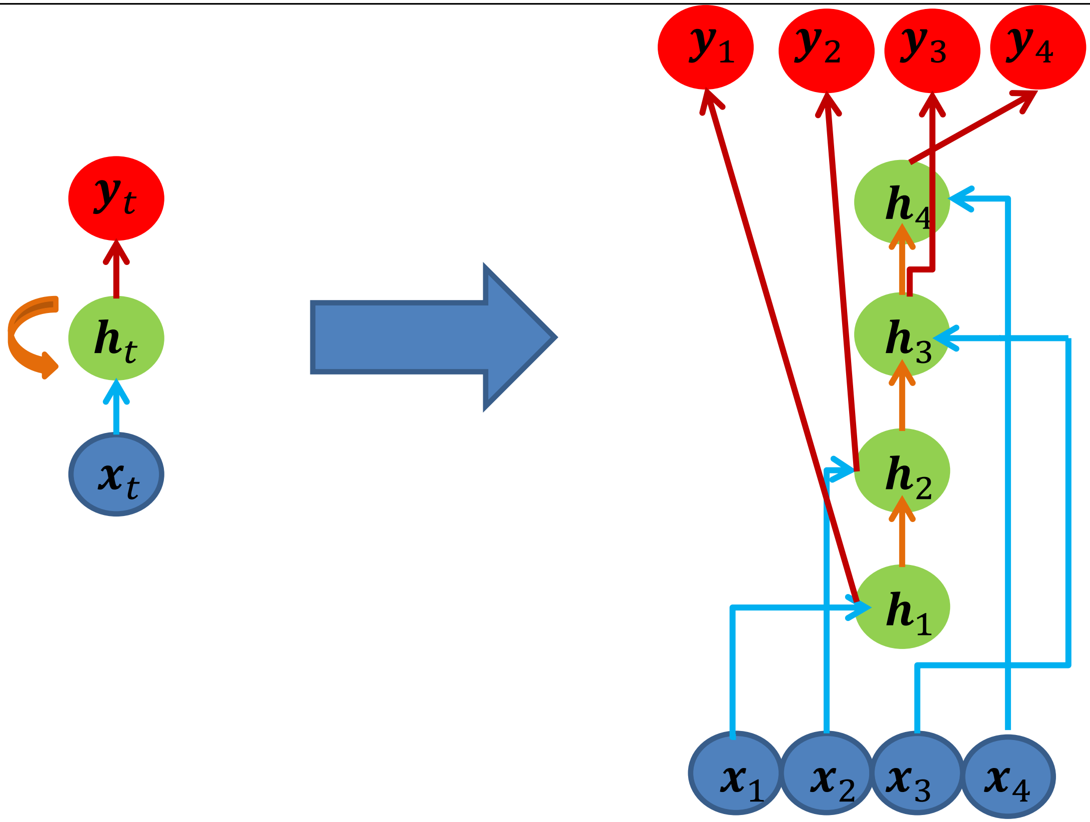
RNNs as deep nets



RNNs as deep nets



RNNs as deep nets



- RNNs are deep MLPs
- With weight sharing
- And sparse connectivity
- And skip connections

Vanishing gradients

Simple Remedies

Regularization & Norm clipping

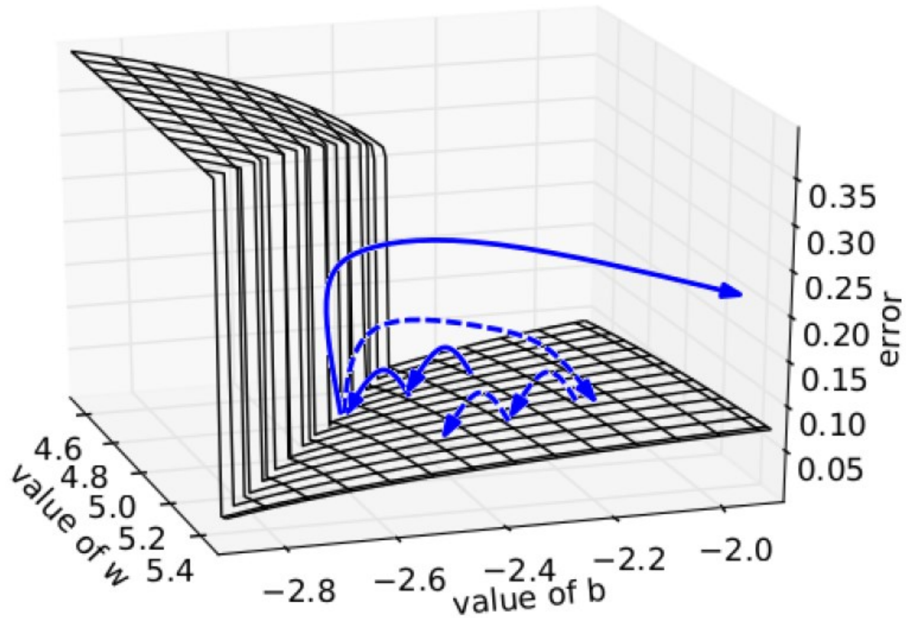
- **Exploding gradients**:

  - L1 or L2 regularization on recurrent weights → keeps $\boldsymbol{W}$ 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

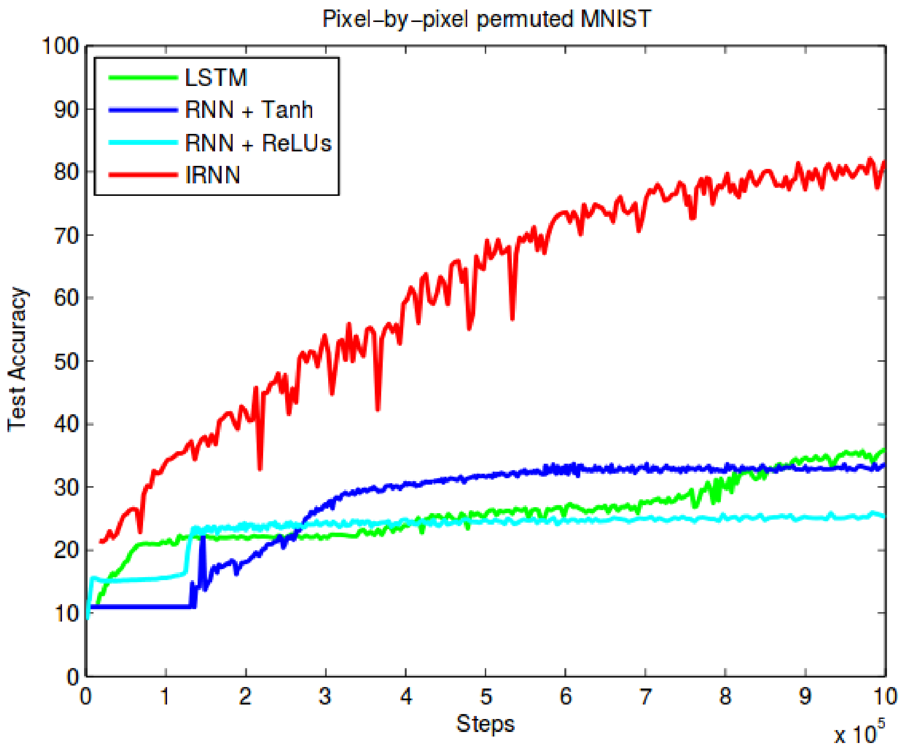
IRNNs

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

IRNNs

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

    (I = identity matrix)



Vanishing gradients

GRUs & LSTMs

Illustration

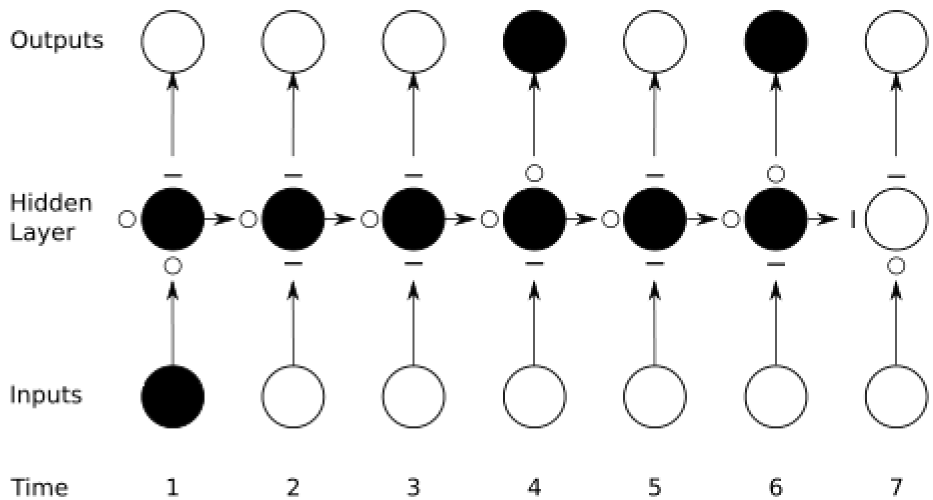


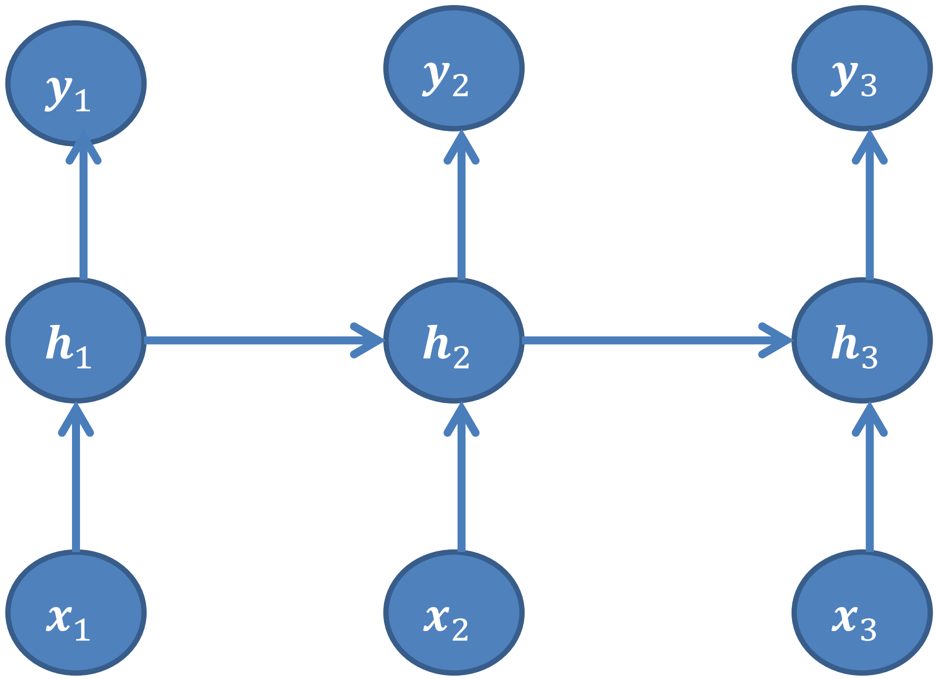
Figure 4.4: **Preservation of gradient information by LSTM.** As in Figure 4.1 the shading of the nodes indicates their sensitivity to the inputs at time one; in this case the black nodes are maximally sensitive and the white nodes are entirely insensitive. The state of the input, forget, and output 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.

Source: Alex Graves, PhD thesis

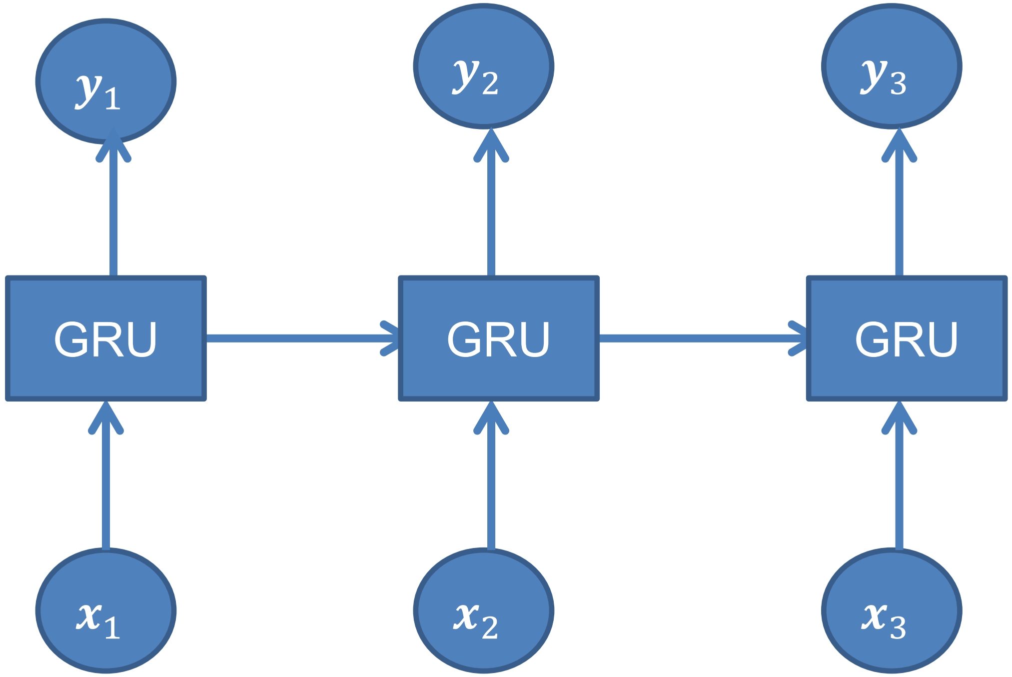
GRUs

- More complex hidden unit computation in recurrence
- Gated Recurrent Units (GRU) introduced by Cho et al. (2014)
- Main idea:
  - Keep around memories to capture long distance dependencies

GRUs



GRUs



Some notation

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

GRU memory unit

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

GRU memory unit - Analysis

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

GRU memory unit - Analysis

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

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

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

GRU memory unit - Analysis

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

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

  - $\boldsymbol{z}_t = 1$:
    - $\boldsymbol{h}_t = \widetilde{\boldsymbol{h}}_t$

GRU memory unit - Analysis

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

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

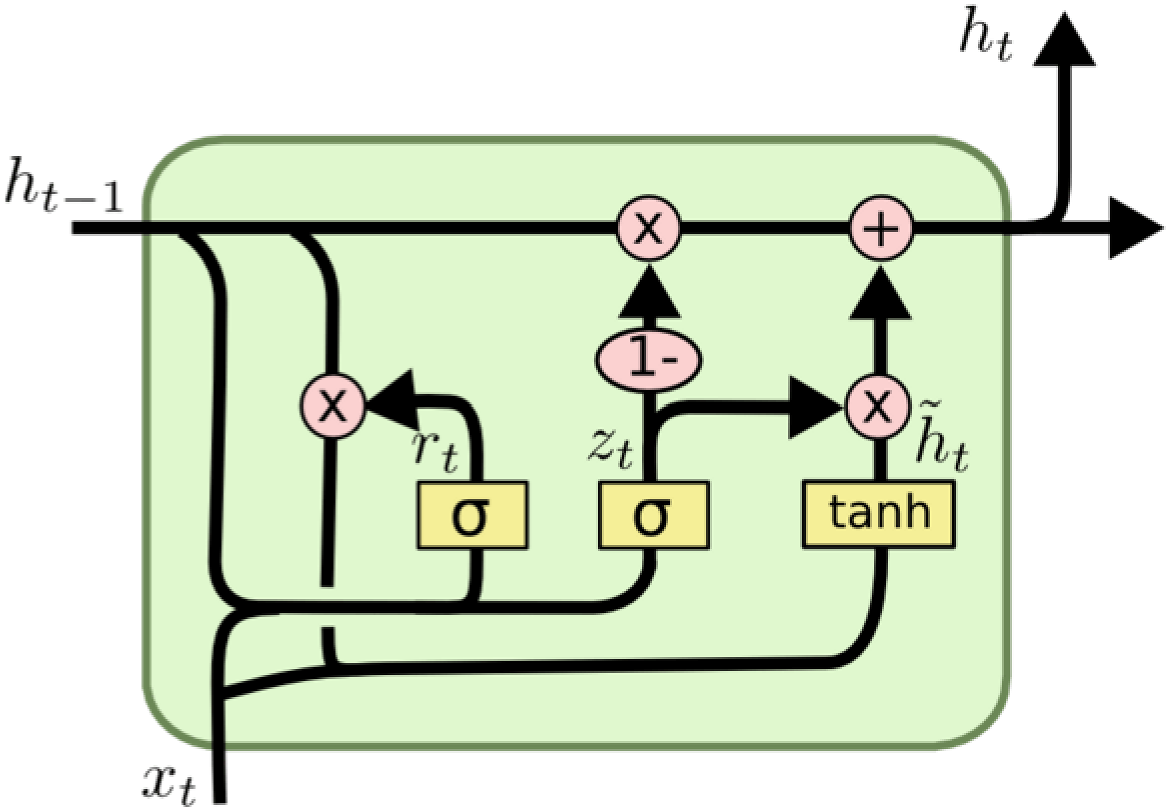
  - $\boldsymbol{z}_t = 1$:
    - $\boldsymbol{h}_t = \widetilde{\boldsymbol{h}}_t$
    - $\boldsymbol{r}_t = 0$:
      - $\boldsymbol{h}_t = \tanh(\boldsymbol{x}_t \boldsymbol{U})$ → Forget past
    - $\boldsymbol{r}_t = 1$:
      - $\boldsymbol{h}_t = \tanh(\boldsymbol{x}_t \boldsymbol{U} + \boldsymbol{h}_{t-1}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

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

GRU memory unit - Analysis

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

GRU illustration



LSTM

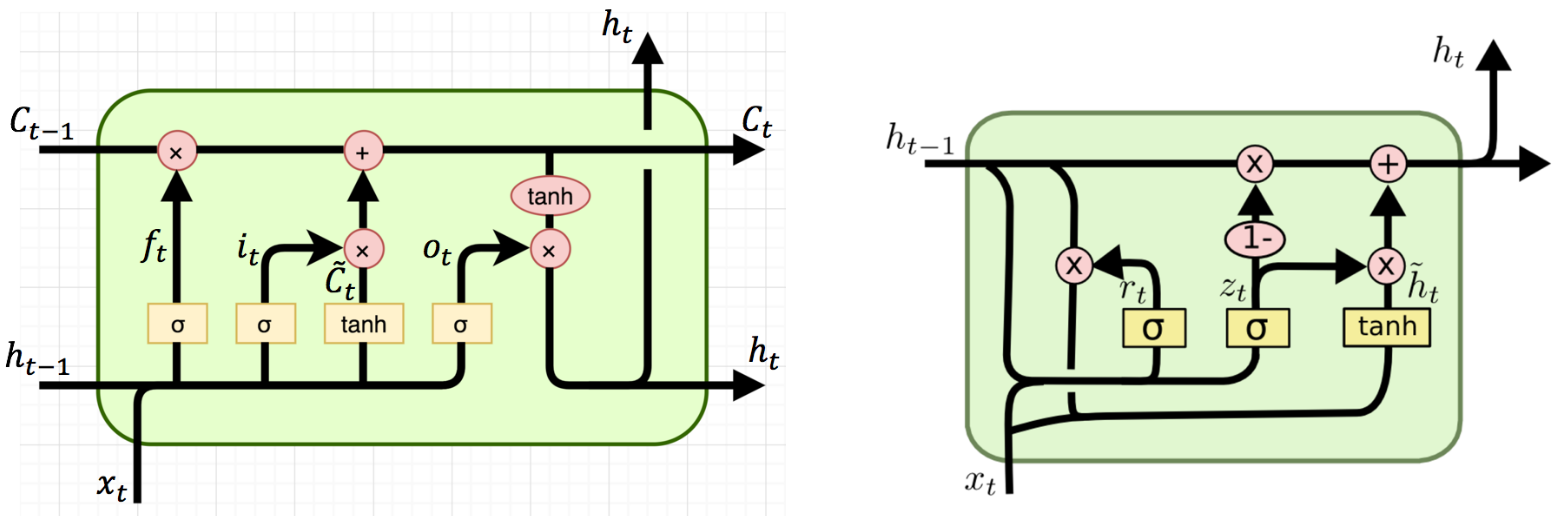
- Can make units even more complex

LSTM (Hochreiter & Schmidhuber, 1997)

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

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

LSTM (Hochreiter & Schmidhuber, 1997)



(a) Long Short-Term Memory (b) Gated Recurrent Unit

LSTM (Hochreiter & Schmidhuber, 1997)

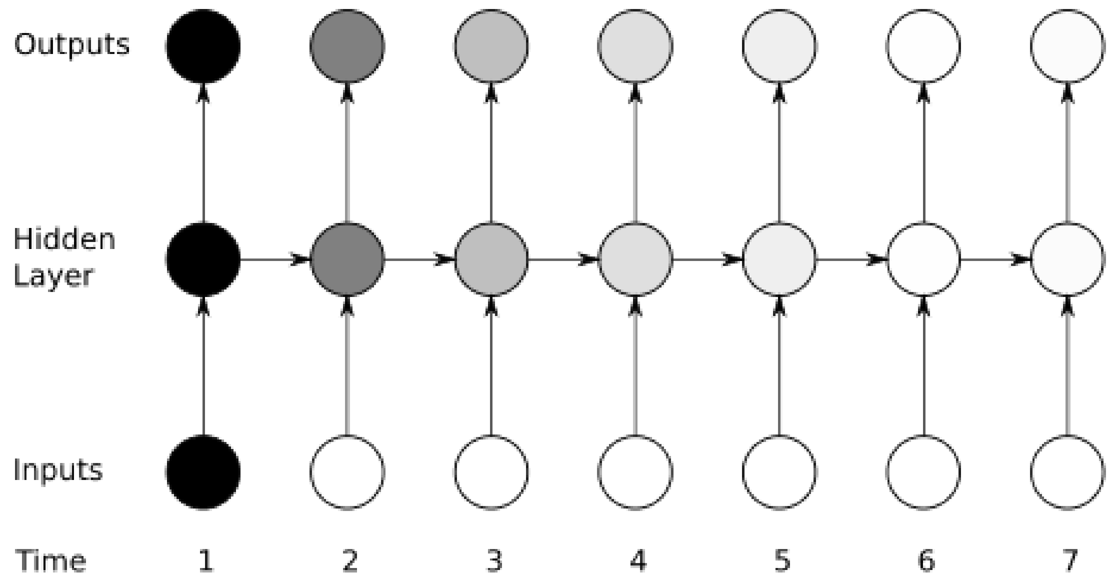


Figure 4.1: **The vanishing gradient problem for RNNs.** The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity). The sensitivity decays over time as new inputs overwrite the activations of the hidden layer,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

Source: Alex Graves, PhD thesis

LSTM (Hochreiter & Schmidhuber, 1997)

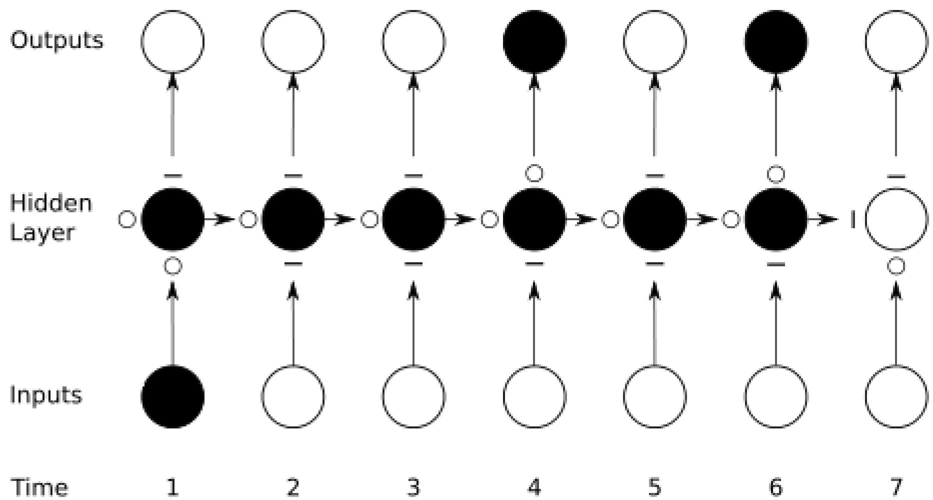


Figure 4.4: **Preservation of gradient information by LSTM.** As in Figure 4.1 the shading of the nodes indicates their sensitivity to the inputs at time one; in this case the black nodes are maximally sensitive and the white nodes are entirely insensitive. The state of the input, forget, and output 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.

Source: Alex Graves, PhD thesis

GRU vs. LSTM

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

Source: Alex Graves, PhD thesis

Summary

TECHNISCHE UNIVERSITÄT DARMSTADT

- Recurrent Neural Networks are powerful
- A lot of ongoing work right now
- Gated Recurrent Units even better
- LSTMs maybe even better

- Next lectures:
  - Lec09: CNNs
  - Lec10: Encoder-Decoder

Mandatory Reading

Reimers and Gurevych (2017), Reporting Score Distributions Makes a Difference: Performance Study of LSTM-networks for Sequence Tagging

References

Pascanu, R., Mikolov, T., & Bengio, Y.: On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning*, 2013

Martens, J. (2010), Deep kearning 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