Natural Language Processing with Deep Learning

Lecture 7 — Text classification 4: Recurrent neural networks

Prof. Dr. Ivan Habernal

November 24, 2023

Natural Language Processing Group
Paderborn University
We focus on Trustworthy Human Language Technologies

PADERBORN UNIVERSITY

www.trusthlt.org

Motivation

Language data – working with sequences (of tokens, characters, etc.)

MLP – fixed input vector size

Motivation

Language data – working with sequences (of tokens, characters, etc.)

MLP – fixed input vector size

How we dealt with it

- Vector concatenation
- Vector addition/averaging (CBOW)
- Limiting context (e.g., Markov property)

What we want to really work with: Sequence of inputs, fixed-size output(s)

Recurrent Neural Networks (RNN) abstraction

- Recurrent Neural Networks (RNN) abstraction
  - RNN as ‘acceptor’ or ‘encoder’
  - RNN as ‘transducer’
- RNN architectures
  - Simple RNN
  - Gated architectures
  - LSTM

RNN abstraction

We have a sequence of n **input** vectors $\boldsymbol{x_{1:n}} = \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$

Each input vector has the same dimension d_{in} : $\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}$

What might $\boldsymbol{x_i}$ contain?

- Typically a word embedding of token i, but could be any arbitrary input, e.g., one-hot encoding of token i

RNN abstraction

We have a sequence of n **input** vectors $\boldsymbol{x_{1:n}} = \boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$

Each input vector has the same dimension d_{in} : $\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}$

What might $\boldsymbol{x_i}$ contain?

- Typically a word embedding of token i, but could be any arbitrary input, e.g., one-hot encoding of token i

We have a single **output** d_{out}-dimensional vector $\boldsymbol{y_n} \in \mathbb{R}^{d_{out}}$

RNN is a function from input to output

$$\boldsymbol{y_n} = \mathrm{RNN}(\boldsymbol{x_{1:n}})$$

RNN in fact returns a sequence of outputs

RNN definition: $\boldsymbol{y_n} = \mathrm{RNN}(\boldsymbol{x_{1:n}})$

Let's have $n = 3$, so our input sequence is $\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}$:

$$\boldsymbol{y_2} = \mathrm{RNN}(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3})$$

RNN in fact returns a sequence of outputs

RNN definition: $\boldsymbol{y_n} = \mathrm{RNN}(\boldsymbol{x_{1:n}})$

Let's have $n = 3$, so our input sequence is $\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}$:

$$\boldsymbol{y_2} = \mathrm{RNN}(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3})$$

But our input sequence also contains $\boldsymbol{x_1}, \boldsymbol{x_2}$, so:

$$\boldsymbol{y_2} = \mathrm{RNN}(\boldsymbol{x_1}, \boldsymbol{x_2})$$

Which makes RNN outputting a vector $\boldsymbol{y_i} \in \mathbb{R}^{d_{out}}$ at each position $i \in (1, \ldots, n)$

Let's call this sequence-outputting function RNN^*:

$$\boldsymbol{y_{1:n}} = \mathrm{RNN}^*(\boldsymbol{x_{1:n}})$$

Recap

For a sequence of input vectors $\boldsymbol{x}_{1:i}$

$$\boldsymbol{y}_i = \mathrm{RNN}(\boldsymbol{x}_{1:i}) \qquad \boldsymbol{y}_{1:n} = \mathrm{RNN}^*(\boldsymbol{x}_{1:n})$$

Recap

For a sequence of input vectors $\boldsymbol{x}_{1:i}$

$$\boldsymbol{y}_i = \text{RNN}(\boldsymbol{x}_{1:i}) \qquad \boldsymbol{y}_{1:n} = \text{RNN}^*(\boldsymbol{x}_{1:n})$$

Without knowing what RNN actually is, what are the advantages?

Recap

For a sequence of input vectors $\boldsymbol{x}_{1:i}$

$$\boldsymbol{y}_i = \text{RNN}(\boldsymbol{x}_{1:i}) \qquad \boldsymbol{y}_{1:n} = \text{RNN}^*(\boldsymbol{x}_{1:n})$$

Without knowing what RNN actually is, what are the advantages?

- Each output $\boldsymbol{y}_i$ takes into account the entire history $\boldsymbol{x}_{1:i}$ without Markov property

What to do with $\boldsymbol{y}_n$ or $\boldsymbol{y}_{1:n}$?

Recap

For a sequence of input vectors $\boldsymbol{x}_{1:i}$

$$\boldsymbol{y}_i = \mathrm{RNN}(\boldsymbol{x}_{1:i}) \qquad \boldsymbol{y}_{1:n} = \mathrm{RNN}^*(\boldsymbol{x}_{1:n})$$

Without knowing what RNN actually is, what are the advantages?

- Each output $\boldsymbol{y}_i$ takes into account the entire history $\boldsymbol{x}_{1:i}$ without Markov property

What to do with $\boldsymbol{y}_n$ or $\boldsymbol{y}_{1:n}$?

- Use for further prediction, e.g., plug into $\mathrm{softmax}$, MLP, etc.

Underlying mechanism of RNNs — states

For "passing information" from one position to the next, i.e. from

$$\boldsymbol{y_i} = \mathrm{RNN}(\boldsymbol{x_{1:i}})$$

to

$$\boldsymbol{y_{i+1}} = \mathrm{RNN}(\boldsymbol{x_{1:i+1}})$$

we use a "state" vector

$$\boldsymbol{s_i} \in \mathbb{R}^{d_{state}}$$

Define RNN recursively — Computing current state

At each step $i \in (1, \ldots, n)$ we have

- Current input vector $\boldsymbol{x_i}$
- Vector of the previous state $\boldsymbol{s_{i-1}}$[1]

and compute

- Current state $\boldsymbol{s_i}$

$$\boldsymbol{s_i} = R(\boldsymbol{s_{i-1}}, \boldsymbol{x_i}) \qquad \text{(we will specify } R \text{ later)}$$

[1] Initial state vector $\boldsymbol{s_0}$ — often omitted, assumed to be zero-filled

Define RNN recursively — Computing current output

At each step $i \in (1, \ldots, n)$ we have

- Current input vector $\boldsymbol{x_i}$
- Vector of the previous state $\boldsymbol{s_{i-1}}$

and compute

- Current state $\boldsymbol{s_i} = R(\boldsymbol{s_{i-1}}, \boldsymbol{x_i})$
- Current output $\boldsymbol{y_i}$

$$\boldsymbol{y_i} = O(\boldsymbol{s_i}) \qquad \text{(we will specify } O \text{ later)}$$

Summary

At each step $i \in (1, \ldots, n)$ we have

- Current input $\boldsymbol{x}_i$ and previous state $\boldsymbol{s}_{i-1}$

and compute

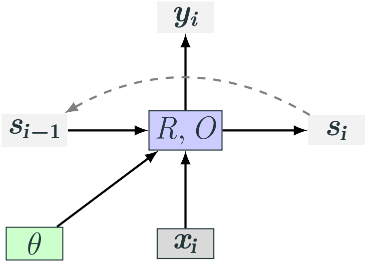
- $\boldsymbol{s}_i = R(\boldsymbol{s}_{i-1}, \boldsymbol{x}_i)$ and $\boldsymbol{y}_i = O(\boldsymbol{s}_i)$

The functions R and O are the same for each position i

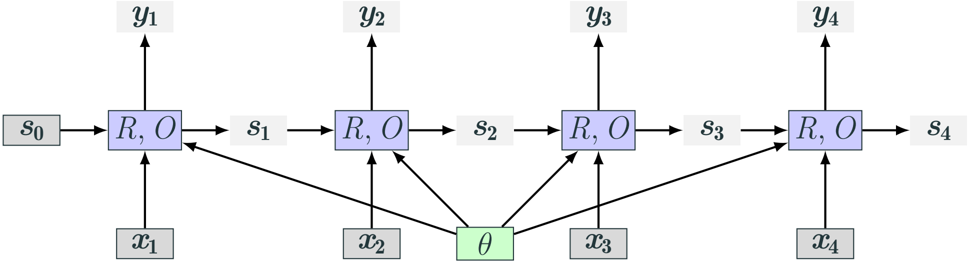
RNN

$\boldsymbol{y}_{1:n} = \mathrm{RNN}^*(\boldsymbol{x}_{1:n}, \boldsymbol{s}_0) \qquad \boldsymbol{s}_i = R(\boldsymbol{s}_{i-1}, \boldsymbol{x}_i) \qquad \boldsymbol{y}_i = O(\boldsymbol{s}_i)$

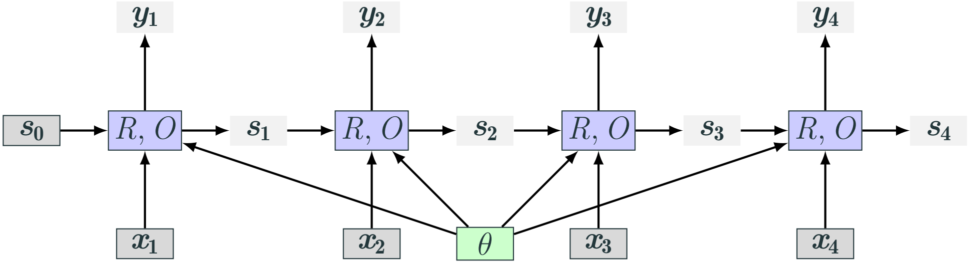
Graphical visualization of abstract RNN (recursive)



Graphical visualization of abstract RNN (unrolled)



Graphical visualization of abstract RNN (unrolled)

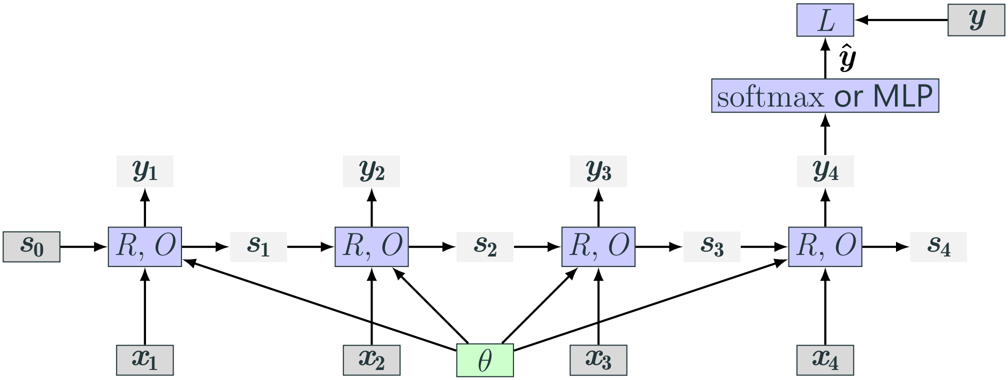


Note that θ (parameters) are “shared” (the same) for all positions

Recurrent Neural Networks (RNN) abstraction

RNN as ‘acceptor’ or ‘encoder’

Supervision on the last output

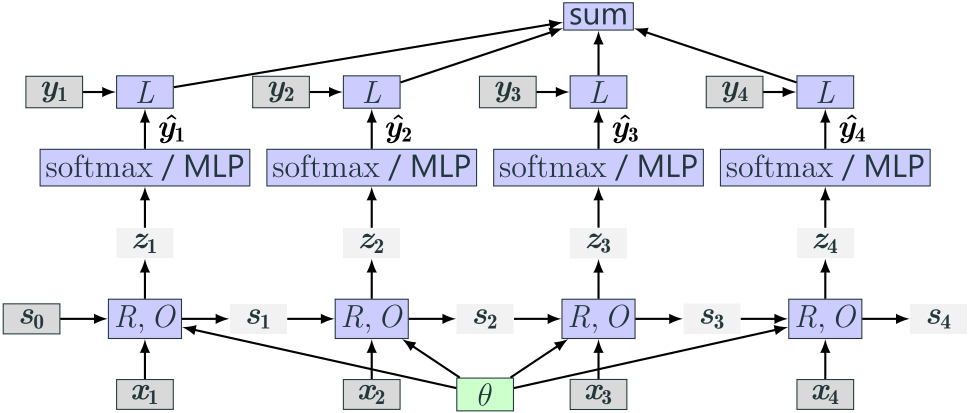


The loss is computed on the final output (e.g., directly on y_n or by putting y_n through MLP)

Recurrent Neural Networks (RNN) abstraction

RNN as ‘transducer’

Supervision on each output



For sequence tagging — loss on each position, overall network's loss simply as a sum of losses

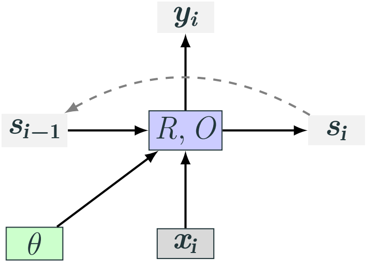
Bi-directional RNNs

Simple idea: Run one RNN from left-to-right (forward, f) and another RNN from right-to-left (backward, b), and concatenate

$$\mathrm{biRNN}(\boldsymbol{x_{1:i}}, i) = \boldsymbol{y_i} = [\underset{f}{\mathrm{RNN}}(\boldsymbol{x_{1:i}}); \underset{b}{\mathrm{RNN}}(\boldsymbol{x_{n:i}})]$$

Both for encoder (concatenate the last outputs) and transducer (concatenate each step's output)

But what is happening ‘inside’ R and O?



RNN architectures

Recurrent Neural Networks (RNN) abstraction

- RNN as ‘acceptor’ or ‘encoder’
- RNN as ‘transducer’

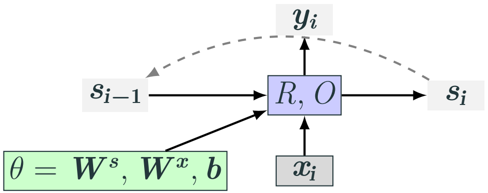
RNN architectures

- Simple RNN
- Gated architectures
- LSTM

RNN architectures

Simple RNN

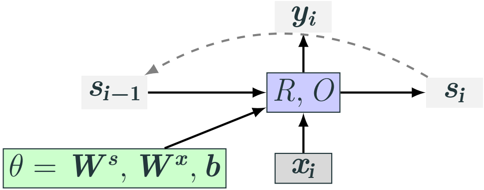
Elman Network or Simple-RNN (S-RNN)



$$\boldsymbol{s}_i = R(\boldsymbol{x}_i, \boldsymbol{s}_{i-1}) = g(\boldsymbol{s}_{i-1}\boldsymbol{W}^s + \boldsymbol{x}_i \boldsymbol{W}^x + \boldsymbol{b})$$

$$\boldsymbol{y}_i = O(\boldsymbol{s}_i) = \boldsymbol{s}_i$$

Elman Network or Simple-RNN (S-RNN)



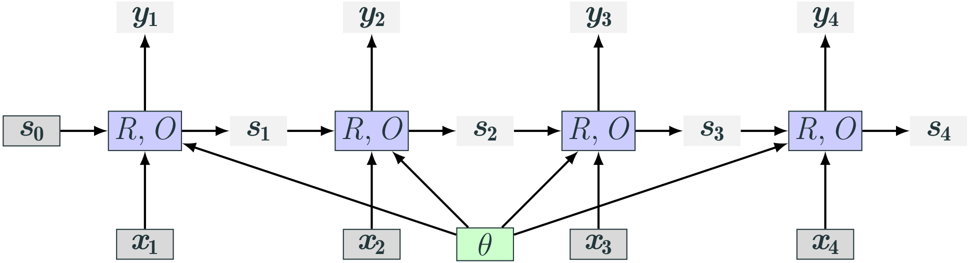
$$\boldsymbol{s_i} = R(\boldsymbol{x_i}, \boldsymbol{s_{i-1}}) = g(\boldsymbol{s_{i-1}} \boldsymbol{W}^s + \boldsymbol{x_i} \boldsymbol{W}^x + \boldsymbol{b})$$

$$\boldsymbol{y_i} = O(\boldsymbol{s_i}) = \boldsymbol{s_i}$$

$\boldsymbol{s_i}, \boldsymbol{y_i} \in \mathbb{R}^d_s \quad \boldsymbol{x_i} \in \mathbb{R}^d_{in} \quad \boldsymbol{W}^x \in \mathbb{R}^{d_{in} \times d_s} \quad \boldsymbol{W}^s \in \mathbb{R}^{d_s \times d_s} \quad \boldsymbol{b} \in \mathbb{R}^{d_s}$

g — commonly tanh or ReLU

Elman Network and vanishing gradient



Gradients might vanish (become exceedingly close to 0) as they propagate back through the computation graph

- Severe in deeper networks, and especially so in recursive and recurrent networks
- Hard for the S-RNN to capture long-range dependencies

RNN architectures

Gated architectures

RNN as a general purpose computing device

State $\boldsymbol{s}_i$ represents a finite memory

Recall: Simple RNN

$$\boldsymbol{s}_i = R(\boldsymbol{x}_i, \boldsymbol{s}_{i-1}) = g(\boldsymbol{s}_{i-1}\boldsymbol{W}^s + \boldsymbol{x}_i\boldsymbol{W}^x + \boldsymbol{b})$$

RNN as a general purpose computing device

State $\boldsymbol{s}_i$ represents a finite memory

Recall: Simple RNN

$$\boldsymbol{s}_i = R(\boldsymbol{x}_i, \boldsymbol{s}_{i-1}) = g(\boldsymbol{s}_{i-1} \boldsymbol{W}^s + \boldsymbol{x}_i \boldsymbol{W}^x + \boldsymbol{b})$$

Each application of function R

- Reads the current memory $\boldsymbol{s}_{i-1}$
- Reads the current input $\boldsymbol{x}_i$
- Operates on them in some way
- Writes the result to the memory $\boldsymbol{s}_i$

RNN as a general purpose computing device

State $\boldsymbol{s}_i$ represents a finite memory

Recall: Simple RNN

$$\boldsymbol{s}_i = R(\boldsymbol{x}_i, \boldsymbol{s}_{i-1}) = g(\boldsymbol{s}_{i-1} \boldsymbol{W}^s + \boldsymbol{x}_i \boldsymbol{W}^x + \boldsymbol{b})$$

Each application of function R

- Reads the current memory $\boldsymbol{s}_{i-1}$
- Reads the current input $\boldsymbol{x}_i$
- Operates on them in some way
- Writes the result to the memory $\boldsymbol{s}_i$

Memory access not controlled: At each step, entire memory state is read, and entire memory state is written

How to provide more controlled memory access?

Memory vector $\boldsymbol{s} \in \mathbb{R}^d$ and input vector $\boldsymbol{x} \in \mathbb{R}^d$

How to provide more controlled memory access?

Memory vector $\boldsymbol{s} \in \mathbb{R}^d$ and input vector $\boldsymbol{x} \in \mathbb{R}^d$

Let's have a binary vector ("gate") $\boldsymbol{g} \in \{0, 1\}^d$

How to provide more controlled memory access?

Memory vector $\boldsymbol{s} \in \mathbb{R}^d$ and input vector $\boldsymbol{x} \in \mathbb{R}^d$

Let's have a binary vector ("gate") $\boldsymbol{g} \in \{0, 1\}^d$

Hadamard-product $\boldsymbol{z} = \boldsymbol{u} \odot \boldsymbol{v}$

Fancy name for element-wise multiplication $\boldsymbol{z}_{[i]} = \boldsymbol{u}_{[i]} \cdot \boldsymbol{v}_{[i]}$

$$\boldsymbol{s}' \leftarrow \boldsymbol{g} \odot \boldsymbol{x} + (\boldsymbol{1} + \boldsymbol{g}) \odot \boldsymbol{s}$$

How to provide more controlled memory access?

Memory vector $\boldsymbol{s} \in \mathbb{R}^d$ and input vector $\boldsymbol{x} \in \mathbb{R}^d$

Let's have a binary vector ("gate") $\boldsymbol{g} \in \{0, 1\}^d$

Hadamard-product $\boldsymbol{z} = \boldsymbol{u} \odot \boldsymbol{v}$

Fancy name for element-wise multiplication $\boldsymbol{z}_{[i]} = \boldsymbol{u}_{[i]} \cdot \boldsymbol{v}_{[i]}$

$\boldsymbol{s}' \leftarrow \boldsymbol{g} \odot \boldsymbol{x} + (\mathbf{1} + \boldsymbol{g}) \odot \boldsymbol{s}$

- Reads the entries in $\boldsymbol{x}$ corresponding to ones in the gate, writes them to the memory

How to provide more controlled memory access?

Memory vector $\boldsymbol{s} \in \mathbb{R}^d$ and input vector $\boldsymbol{x} \in \mathbb{R}^d$

Let's have a binary vector ("gate") $\boldsymbol{g} \in \{0, 1\}^d$

Hadamard-product $\boldsymbol{z} = \boldsymbol{u} \odot \boldsymbol{v}$

Fancy name for element-wise multiplication $\boldsymbol{z}_{[i]} = \boldsymbol{u}_{[i]} \cdot \boldsymbol{v}_{[i]}$

$$\boldsymbol{s}' \leftarrow \boldsymbol{g} \odot \boldsymbol{x} + (\mathbf{1} + \boldsymbol{g}) \odot \boldsymbol{s}$$

- Reads the entries in $\boldsymbol{x}$ corresponding to ones in the gate, writes them to the memory
- Remaining locations are copied from the memory

Gate example

Updating memory position 2

$$\begin{pmatrix} 8 \\ 11 \\ 3 \end{pmatrix} \leftarrow \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \odot \begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \odot \begin{pmatrix} 8 \\ 9 \\ 3 \end{pmatrix}$$

$$\boldsymbol{s}' \leftarrow \boldsymbol{g} \odot \boldsymbol{x} + (\boldsymbol{1} + \boldsymbol{g}) \odot \boldsymbol{s}$$

Gate example

Updating memory position 2

$$\begin{pmatrix} 8 \\ 11 \\ 3 \end{pmatrix} \leftarrow \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \odot \begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \odot \begin{pmatrix} 8 \\ 9 \\ 3 \end{pmatrix}$$

$$\boldsymbol{s}' \leftarrow \boldsymbol{g} \odot \boldsymbol{x} + (\mathbf{1} + \boldsymbol{g}) \odot \boldsymbol{s}$$

Could be used for gates in RNNs! But:

- Our gates are not learnable
- Our hard-gates are not differentiable

Solution: Replace with 'soft' gates

RNN architectures

LSTM

Long Short-Term Memory (LSTM)

Designed to solve the vanishing gradients problem, first to introduce the gating mechanism

Long Short-Term Memory (LSTM)

Designed to solve the vanishing gradients problem, first to introduce the gating mechanism

LSTM splits the state vector s_i exactly in two halves

- One half is treated as ‘memory cells’
- The other half is ‘working memory’

Long Short-Term Memory (LSTM)

Designed to solve the vanishing gradients problem, first to introduce the gating mechanism

LSTM splits the state vector s_i exactly in two halves

- One half is treated as 'memory cells'
- The other half is 'working memory'

Memory cells

- Designed to preserve the memory, and also the error gradients, across time
- Controlled through *differentiable gating components* — smooth functions that simulate logical gates

Long Short-Term Memory (LSTM)

The state at time j is composed of two vectors:

- $\boldsymbol{c_j}$ — the memory component
- $\boldsymbol{h_j}$ — the hidden state component

Long Short-Term Memory (LSTM)

The state at time j is composed of two vectors:

- $\boldsymbol{c_j}$ — the memory component
- $\boldsymbol{h_j}$ — the hidden state component

At each input state j, a gate decides how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten

Long Short-Term Memory (LSTM)

The state at time j is composed of two vectors:

- $\boldsymbol{c_j}$ — the memory component
- $\boldsymbol{h_j}$ — the hidden state component

At each input state j, a gate decides how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten

There are three gates

- $\boldsymbol{i}$ — input gate
- $\boldsymbol{f}$ — forget gate
- $\boldsymbol{o}$ — output gate

LSTM architecture

c_{j-1}

h_{j-1}

x_j

LSTM architecture

$\boldsymbol{c}_{j-1}$

$\boldsymbol{h}_{j-1}$

$$\boldsymbol{z} = \tanh\left(\boldsymbol{x}_j \boldsymbol{W}^{\boldsymbol{xz}} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{\boldsymbol{hz}}\right)$$

$\boldsymbol{x}_j$

$\boldsymbol{z}$ — update candidate

LSTM architecture

$\boldsymbol{c}_{j-1}$

$\boldsymbol{h}_{j-1}$

$$\boldsymbol{f} = \sigma(\boldsymbol{x}_j \boldsymbol{W}^{\mathbf{xf}} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{\mathbf{hf}})$$

$$\boldsymbol{z} = \tanh(\boldsymbol{x}_j \boldsymbol{W}^{\mathbf{xz}} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{\mathbf{hz}})$$

$\boldsymbol{x}_j$

$\boldsymbol{z}$ — update candidate

LSTM architecture

$\boldsymbol{c}_{j-1}$

$$\boldsymbol{i} = \sigma(\boldsymbol{x}_j \boldsymbol{W}^{xi} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{hi})$$

$\boldsymbol{h}_{j-1}$

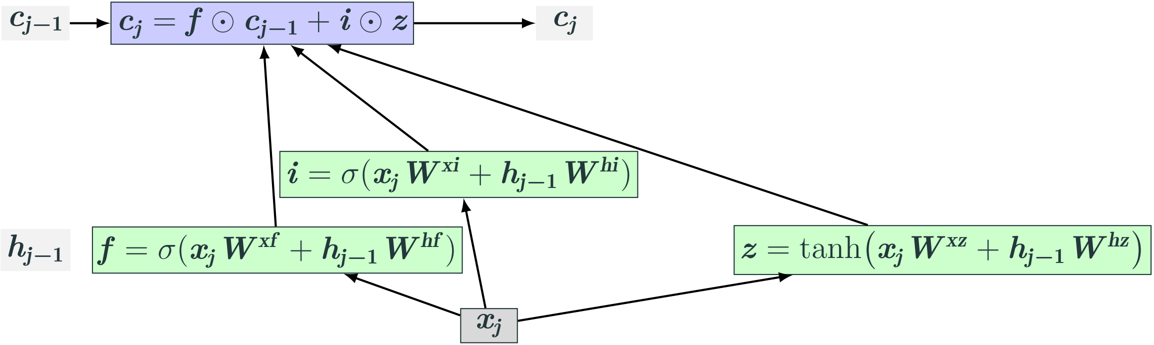
$$\boldsymbol{f} = \sigma(\boldsymbol{x}_j \boldsymbol{W}^{xf} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{hf})$$

$$\boldsymbol{z} = \tanh(\boldsymbol{x}_j \boldsymbol{W}^{xz} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{hz})$$

$\boldsymbol{x}_j$

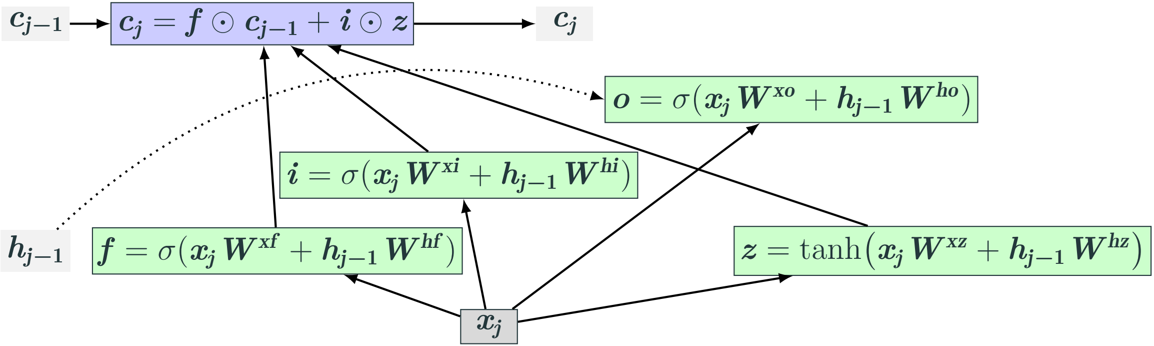
$\boldsymbol{z}$ — update candidate

LSTM architecture



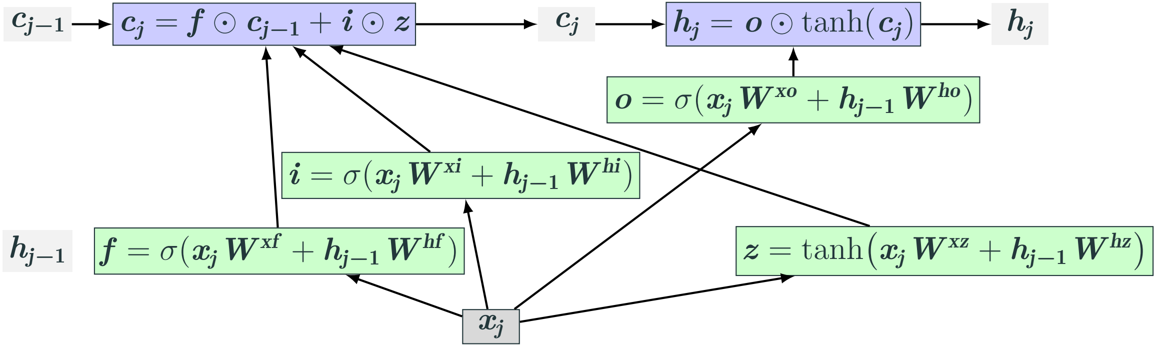
z — update candidate

LSTM architecture



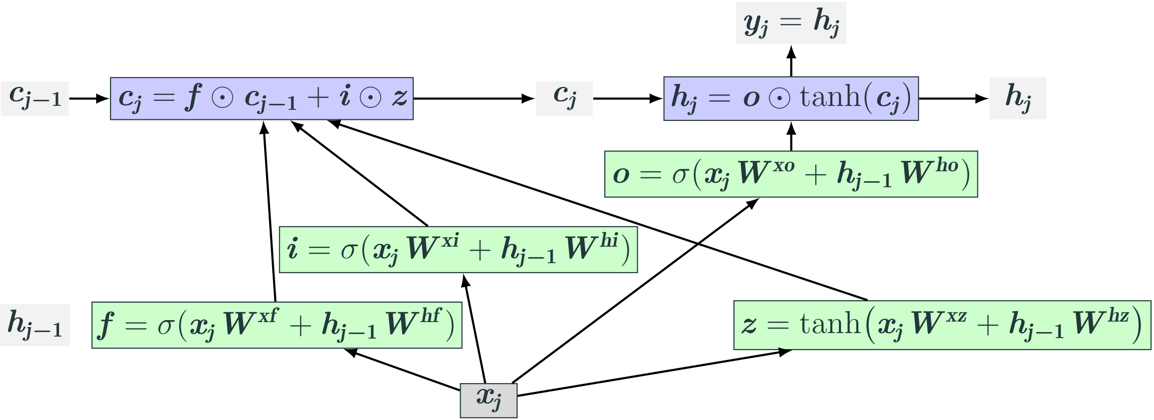
z — update candidate

LSTM architecture



z — update candidate

LSTM architecture

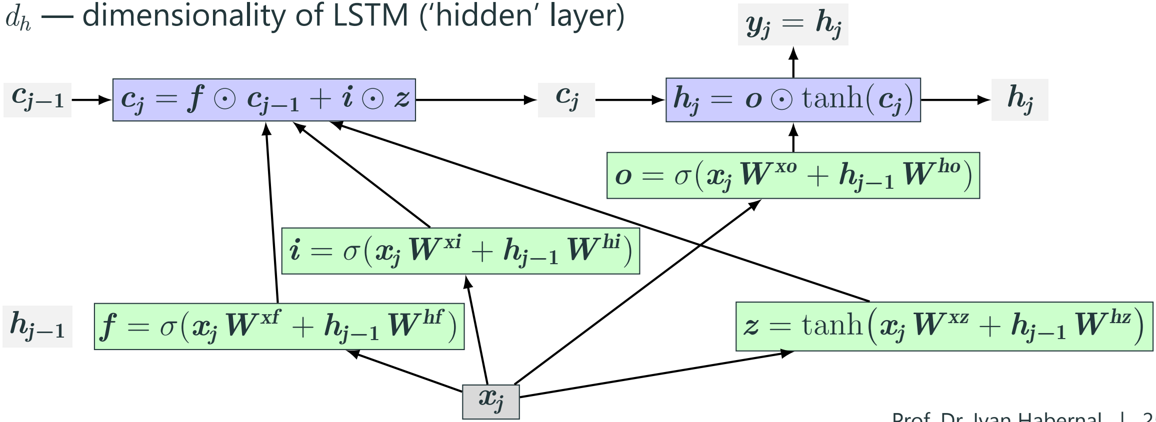


z — update candidate

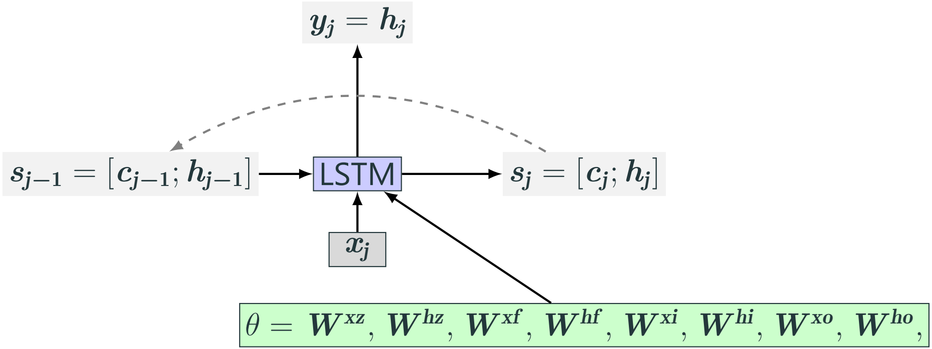
LSTM parameters and dimensions

$\boldsymbol{x_j} \in \mathbb{R}^{d_{in}}$ $\quad \boldsymbol{c_j}, \boldsymbol{h_j}, \boldsymbol{y_j}, \boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{z} \in \mathbb{R}^{d_h}$ $\quad \boldsymbol{W}^{\mathbf{x}\star} \in \mathbb{R}^{d_{in} \times d_h}$ $\quad \boldsymbol{W}^{\boldsymbol{h}\star} \in \mathbb{R}^{d_h \times d_h}$

d_h — dimensionality of LSTM (‘hidden’ layer)



LSTM as a ‘layer’



We also ignored bias terms for each gate

Recap

- Recurrent Neural Networks (RNN) abstraction
  - RNN as ‘acceptor’ or ‘encoder’
  - RNN as ‘transducer’
- RNN architectures
  - Simple RNN
  - Gated architectures
  - LSTM

Take aways

- RNNs for arbitrary long input
- Encoding the entire sequence and/or each step
- Modeling freedom with bi-directional RNNs
- Vanishing gradients in deep nets — gating mechanism, memory cells
- LSTM a particularly powerful RNN

License and credits

Credits

Ivan Habernal