Natural Language Processing with Deep Learning

Lecture 6 — Word embeddings and RNNs

Prof. Dr. Ivan Habernal

November 20, 2025

www.trusthlt.org
Trustworthy Human Language Technologies Group (TrustHLT)
Ruhr University Bochum & Research Center Trustworthy Data Science and Security

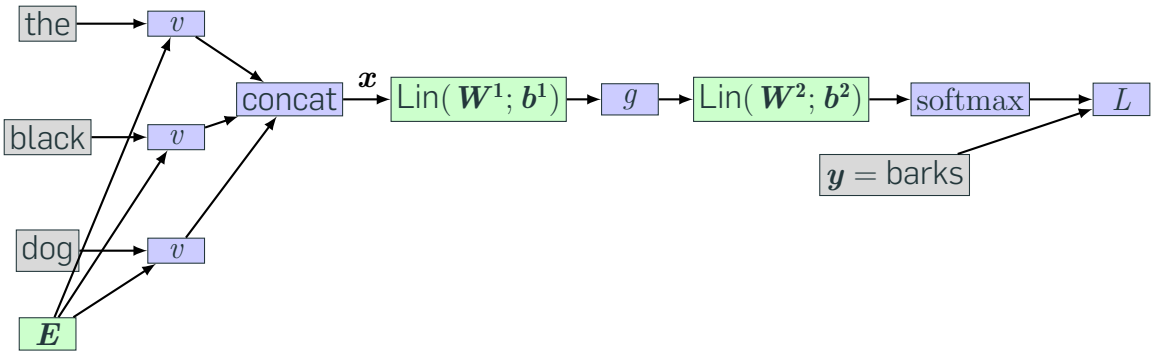
RUHR UNIVERSITÄT BOCHUM RUB

TrustHLT 1001100110101

CENTER FOR TRUSTWORTHY DATA SCIENCE AND SECURITY

Last lecture

Learned word representations as a by-product



Each row of $\boldsymbol{E}$ learns a word representation

Learning word embeddings

Learning word embeddings

Distributional hypothesis

Recall: One-hot encoding of words

Major drawbacks?

- No ‘semantic’ similarity, all words are equally ‘similar’

Example (see Lecture 3 for more)

$$V = \begin{pmatrix} \text{a}_1 & \text{abandon}_2 & \dots & \text{zone}_{2,999} & \text{zoo}_{3,000} \end{pmatrix}$$

$$\text{nice} = \begin{pmatrix} 0_1 & \dots & 1_{1,852} & \dots & 0_{2,999} & 0_{3,000} \end{pmatrix}$$

$$\text{pleasant} = \begin{pmatrix} 0_1 & \dots & 1_{2,012} & \dots & 0_{2,999} & 0_{3,000} \end{pmatrix}$$

$$\text{horrible} = \begin{pmatrix} 0_1 & \dots & 1_{696} & \dots & 0_{2,999} & 0_{3,000} \end{pmatrix}$$

Distributional hypothesis

The distributional hypothesis stating that *words are similar if they appear in similar contexts*

Example

Intuitively, when we encounter a sentence with an unknown word such as the word **wampinuk** in

Distributional hypothesis

The distributional hypothesis stating that *words are similar if they appear in similar contexts*

Example

Intuitively, when we encounter a sentence with an unknown word such as the word **wampinuk** in

Marco saw a hairy little **wampinuk** *crouching behind a tree*

We infer the meaning of the word based on the context in which it occurs

From neural language models to training word embeddings

(Neural) language model's goal: Predict probability distribution over $|V|$ for the next word conditioned on the previous words

- Side product: Can learn useful word embeddings

From neural language models to training word embeddings

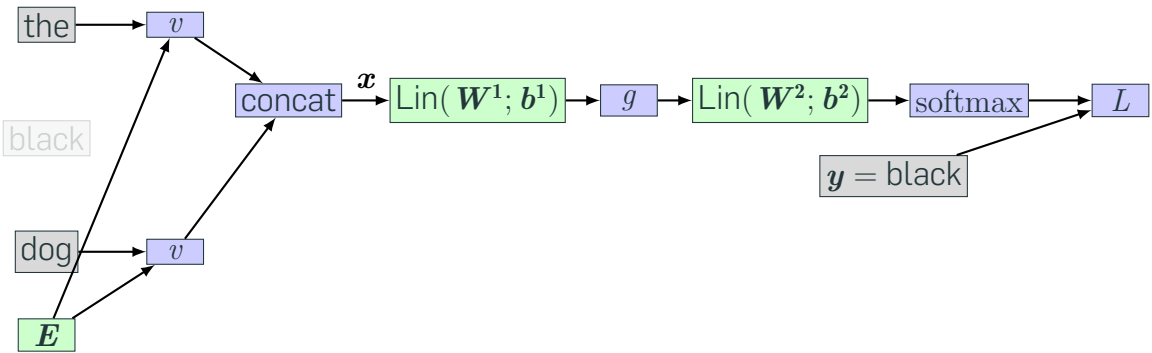
(Neural) language model's goal: Predict probability distribution over $|V|$ for the next word conditioned on the previous words

- Side product: Can learn useful word embeddings

What if we don't need probability distribution but just want to learn word embeddings?

- We can relax our Markov assumption of 'look at k previous words only'
- We can get rid of the costly normalization in softmax

Simplification 1: Ditch the Markov property — look into the future!



For example, instead of modeling $\Pr(w_3|w_1, w_2, \boxdot)$, we model $\Pr(w_2|w_1, \boxdot, w_3)$

Simplification 2: Give up the costly probability distribution

Instead of predicting probability distribution over the entire vocabulary, we just want to predict some score of **context** and **target word**

What could such a score be?

Simplification 2: Give up the costly probability distribution

Instead of predicting probability distribution over the entire vocabulary, we just want to predict some score of **context** and **target word**

What could such a score be?

- Prefer words in their true contexts (high score)

Simplification 2: Give up the costly probability distribution

Instead of predicting probability distribution over the entire vocabulary, we just want to predict some score of **context** and **target word**

What could such a score be?

- Prefer words in their true contexts (high score)
- Penalize words in their 'untrue' contexts (low score)

Negative sampling

Instead of predicting probability distribution for the target word, we **create an artificial binary task** by choosing either the correct or a random incorrect target word

Negative sampling

Instead of predicting probability distribution for the target word, we **create an artificial binary task** by choosing either the correct or a random incorrect target word

$$y = \begin{cases} 1 & \text{if } (w, c_{1:k}) \text{ is a positive example from the corpus} \\ 0 & \text{if } (w', c_{1:k}) \text{ is a negative example from the corpus} \end{cases}$$

Negative sampling

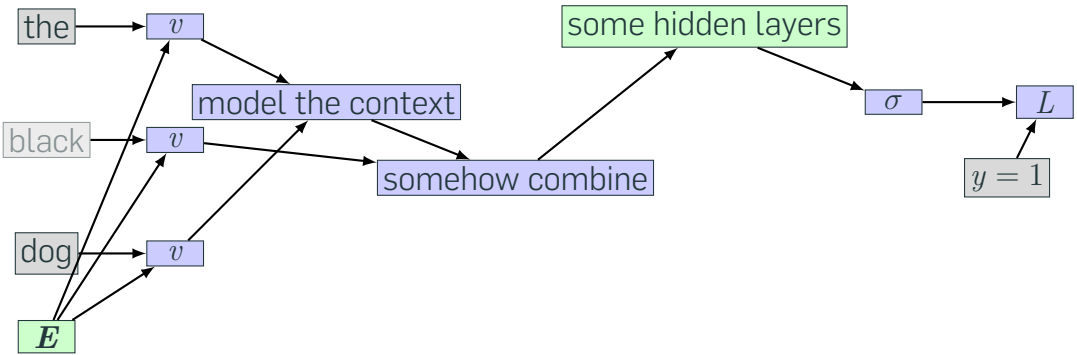
Instead of predicting probability distribution for the target word, we **create an artificial binary task** by choosing either the correct or a random incorrect target word

$$y = \begin{cases} 1 & \text{if } (w, c_{1:k}) \text{ is a positive example from the corpus} \\ 0 & \text{if } (w', c_{1:k}) \text{ is a negative example from the corpus} \end{cases}$$

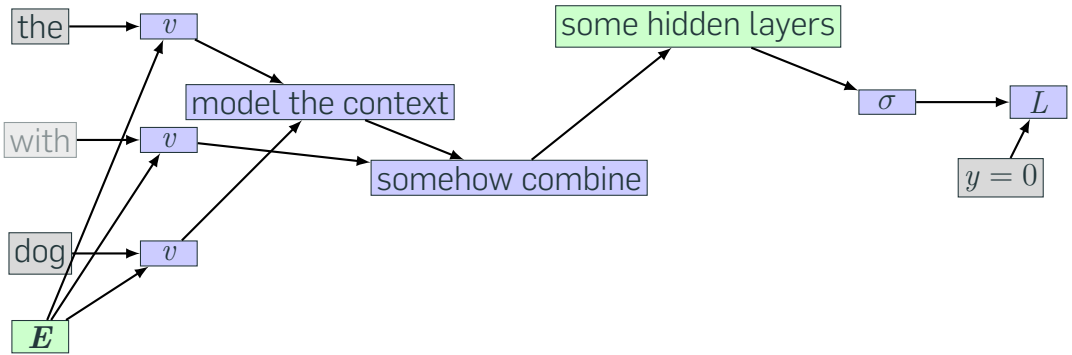
Which distribution for sampling w' from V?

- Corpus-based frequency: $\frac{\#(v)}{\sum_{v' \in V} \#(v')}$ (pr. of each word v)
- Re-weighted: $\frac{\#(v)^{0.75}}{\sum_{v' \in V} \#(v')^{0.75}}$ (more weight on less frequent words done in word2vec)

Turn the problem into binary classification (positive example)



Turn the problem into binary classification (negative example)

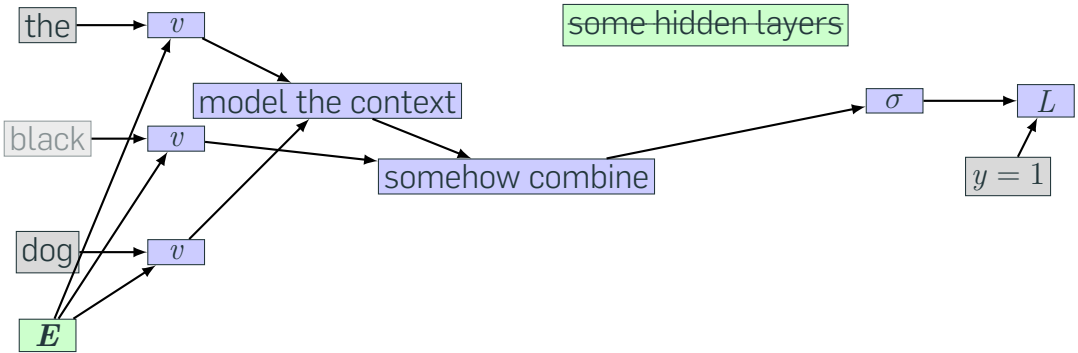


word2vec

1. Last lecture
2. Learning word embeddings
3. word2vec
4. Advantages and limitations of words embeddings
5. Recurrent neural networks
6. RNN architectures

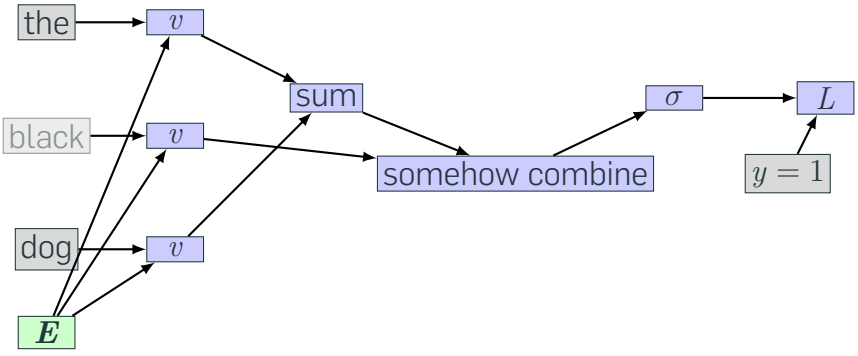
word2vec

word2vec simplifies the neural LM by removing the hidden layer (so turning it into a log-linear model!)



word2vec — how to model the context?

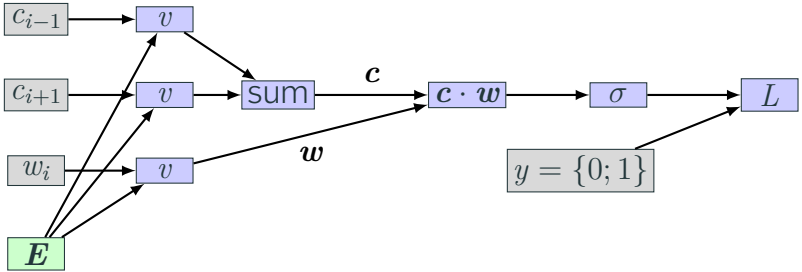
word2vec — how to model the context?



T. Mikolov, K. Chen, G. Corrado, and J. Dean (2013). **"Efficient estimation of word representations in vector space".** In: *1st International Conference on Learning Representations ICLR, Workshop Track Proceedings*. Ed. by Y. Bengio and Y. LeCun, pp. 1–12

Variant 1: Continuous bag of words (CBOW) $\boldsymbol{c} = \sum_{i=1}^{k} v(c_i)$

Final CBOW word2vec — similarity score is the dot product



w_i — target word, c_{i-1}, c_{i+1} — context words (one-hot)

$y = 1$ for correct word-context pairs, $y = 0$ for random w_i

The only learnable parameter is the embedding matrix $\boldsymbol{E}$

What is $\sigma(\boldsymbol{c} \cdot \boldsymbol{w})$ doing?

word2vec: Learning useful word embeddings

Train the network to distinguish ‘good’ word-context pairs from ‘bad’ ones

Create a set D of correct word-context pairs and set $\bar{D}$ of incorrect word-context pairs

The goal of the algorithm is to estimate the probability $\Pr(D = 1 \mid w, c)$ that the word-context pair w, c comes from the correct set D

This should be high ($\rightarrow 1$) for pairs from D and low ($\rightarrow 0$) for pairs from $\bar{D}$

Advantages and limitations of words embeddings

Using word embeddings

Pre-trained embeddings: 'Semantic' input to any neural network instead of one-hot word encoding

- Instance of **transfer learning** — pre-trained (self-trained) on an auxiliary task, plugged into a more complex model as pre-trained weights

Example: Represent a document as an average of its words' embeddings (average bag-of-words through embeddings) for text classification

Side note: word2vec and word embeddings → part of the deep-learning revolution in NLP around 2015

Semantic similarity, short document similarity, query expansion

S. Kuzi, A. Shtok, and O. Kurland (2016). **"Query Expansion Using Word Embeddings".** In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, pp. 1929–1932

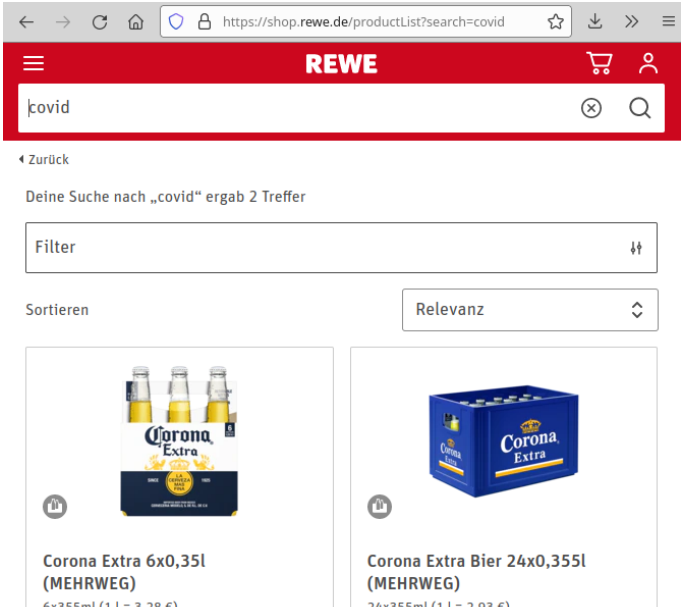
"Using Word2Vec's CBOW embedding approach, applied over the entire corpus on which search is performed, we select terms that are semantically related to the query."

Semantic similarity, short document similarity, query expansion

S. Kuzi, A. Shtok, and O. Kurland (2016). **"Query Expansion Using Word Embeddings".** In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, pp. 1929–1932

"Using Word2Vec's CBOW embedding approach, applied over the entire corpus on which search is performed, we select terms that are semantically related to the query."

What can possibly go wrong?



Searched for **covid** (test), returned the closest items with **corona** in the title (because their embeddings learned that covid ≈ corona).

Query expansion with word embeddings might be tricky

Mining word analogies with word2vec

'Germany to Berlin is France to ?'

Solved by $v(\text{Berlin}) - v(\text{Germany}) + v(\text{France})$, outputs vector $\boldsymbol{x}$ which is closest to Paris in the embeddings space (the closest row in $\boldsymbol{E}$)

Find the queen

$v(\text{king}) - v(\text{man}) + v(\text{woman}) \approx v(\text{queen})$

Limitations of word embeddings (1)

Definition of similarity

Completely operational: words are similar if used in similar contexts

Antonyms

Words opposite of each other (buy—sell, hot—cold) tend to appear in similar contexts (things that can be hot can also be cold, things that are bought are often sold)

Models might tend to judge antonyms as very similar to each other

Limitations of word embeddings (2)

Biases

Distributional methods reflect the usage patterns in the corpora on which they are based

The corpora reflect human biases in the real world (cultural or otherwise)

"Word embeddings encode not only stereotyped biases but also other knowledge [..] are problematic as toward race or gender, or even simply veridical, reflecting the status quo distribution of gender with respect to careers or first names."

A. Caliskan, J. J. Bryson, and A. Narayanan (Apr. 2017). **"Semantics derived automatically from language corpora contain human-like biases".** In: *Science* 356 (6334), pp. 183–186

Limitations of word embeddings (3)

Polysemy, context independent representation

Some words have obvious multiple senses

A *bank* may refer to a financial institution or to the side of a river, a *star* may be an abstract shape, a celebrity, an astronomical entity

Using a single vector for all forms is problematic

Recurrent neural networks

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.)

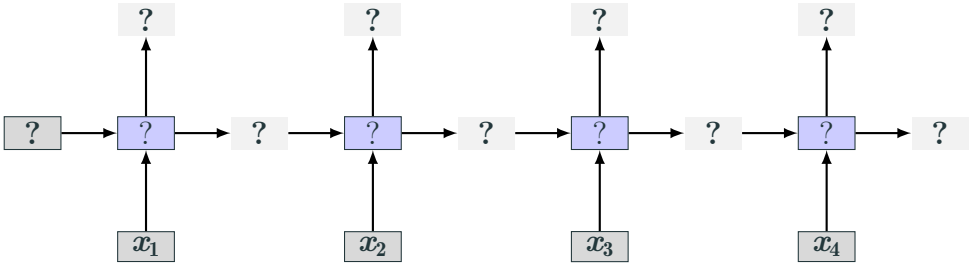
MLP – fixed input vector size

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)

Our goal would be to build something like this



Example for 4 input tokens

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?

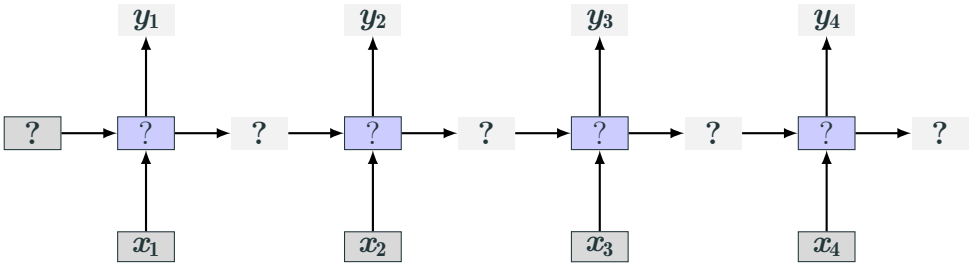
- Typically a word embedding of token i, but could be any arbitrary input, e.g., one-hot encoding of token i

Output: 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^*:

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

Adding outputs to our sketch



Recap

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

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

Recap

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

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

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

Recap

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

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

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 $\boldsymbol{y_{1:n}}$?

Recap

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

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

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

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

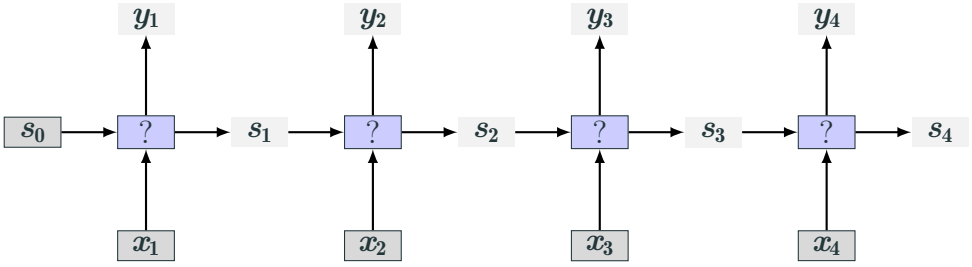
to

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

we use a “state” vector

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

Adding state vectors



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

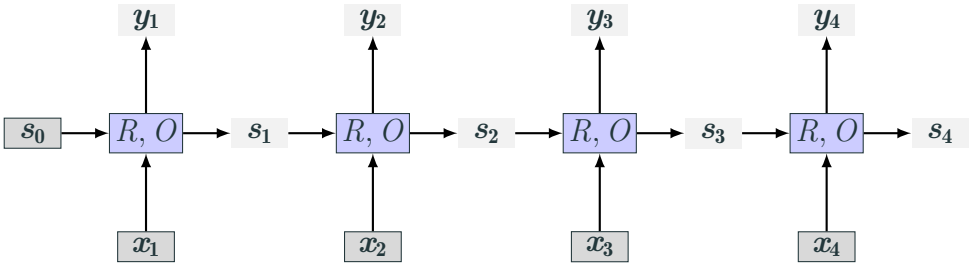
- Current input vector $\boldsymbol{x_i}$
- Vector of the previous state $\boldsymbol{s_{i-1}}$

and compute

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

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

Adding R and O



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

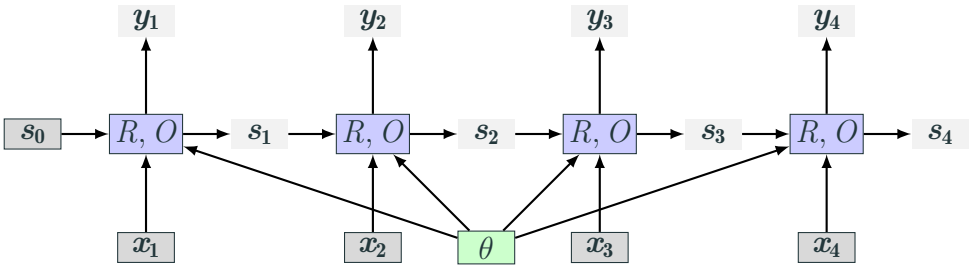
- $\boldsymbol{s_i} = R(\boldsymbol{s_{i-1}}, \boldsymbol{x_i})$ and $\boldsymbol{y_i} = O(\boldsymbol{s_i})$

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

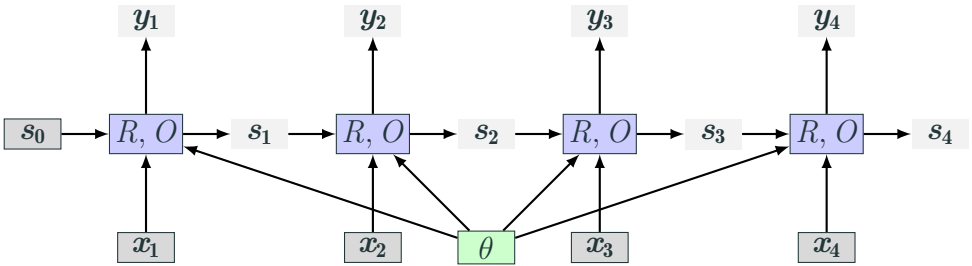
RNN

$$\boldsymbol{y_{1:n}} = \mathrm{RNN}^*(\boldsymbol{x_{1:n}}, \boldsymbol{s_0}) \qquad \boldsymbol{s_i} = R(\boldsymbol{s_{i-1}}, \boldsymbol{x_i}) \qquad \boldsymbol{y_i} = O(\boldsymbol{s_i})$$

Graphical visualization of abstract RNN (unrolled)

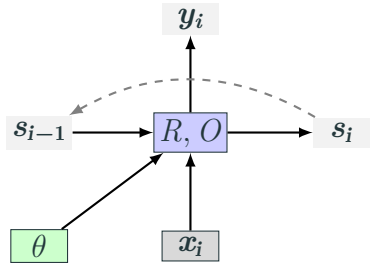


Graphical visualization of abstract RNN (unrolled)



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

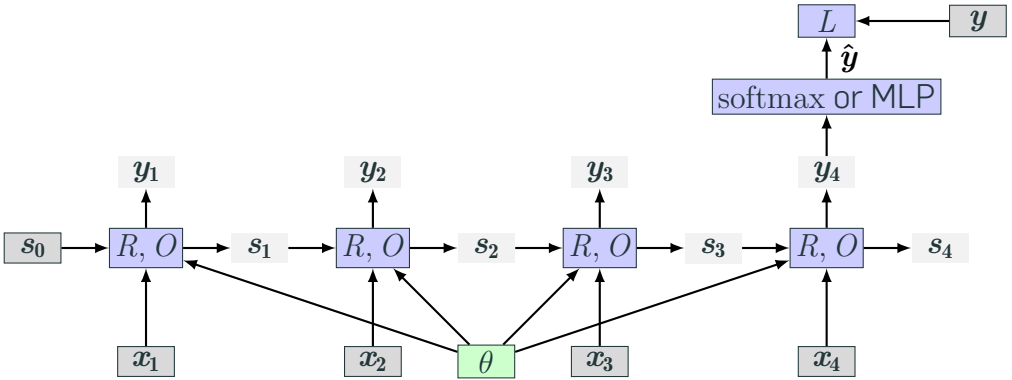
Graphical visualization of abstract RNN (recursive)



Recurrent neural networks

RNN as ‘acceptor’ or ‘encoder’

Supervision on the last output

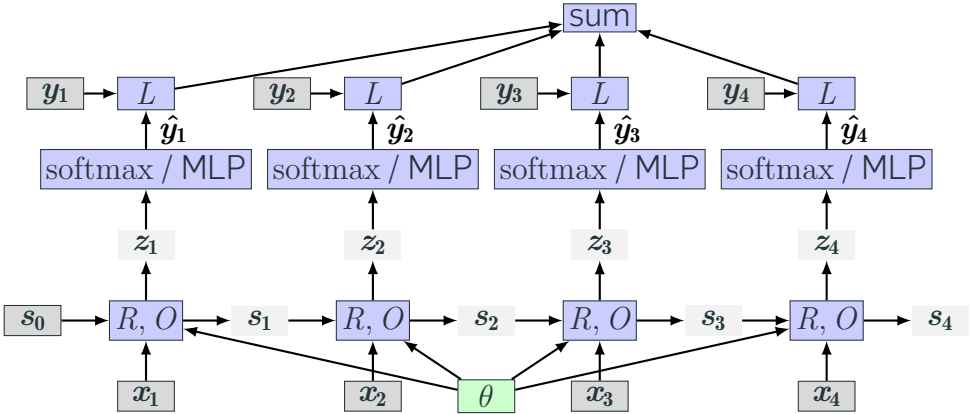


The loss is computed on the final output (e.g., directly on y_n or by putting y_n through MLP)

Recurrent neural networks

RNN as ‘transducer’

Supervision on each output



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

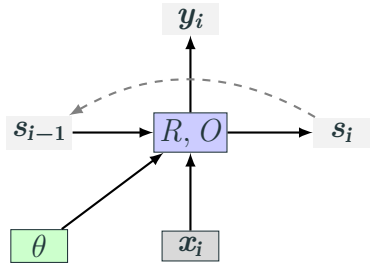
Bi-directional RNNs

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

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

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

But what is happening 'inside' R and O?



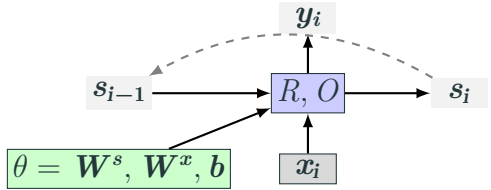
RNN architectures

1 Last lecture
2 Learning word embeddings
3 word2vec
4 Advantages and limitations of words embeddings
5 Recurrent neural networks
6 RNN architectures

RNN architectures

Simple RNN

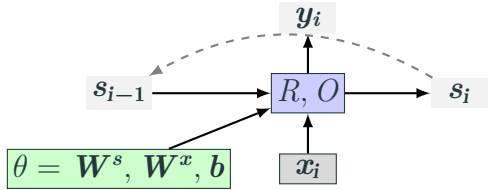
Elman Network or Simple-RNN (S-RNN)



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

$$y_i = O(s_i) = s_i$$

Elman Network or Simple-RNN (S-RNN)

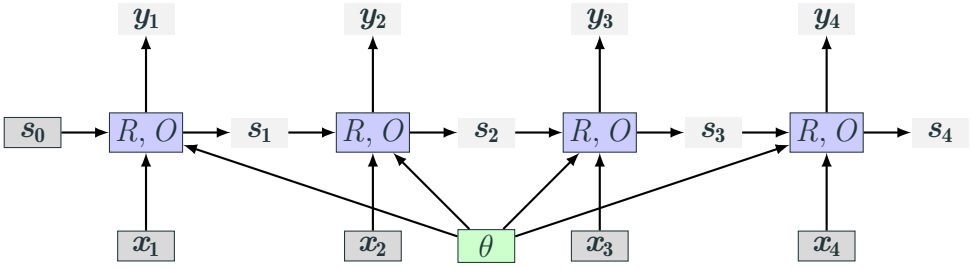


$$s_i = R(x_i, s_{i-1}) = g(s_{i-1} W^s + x_i W^x + b)$$
$$y_i = O(s_i) = s_i$$

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

g — commonly tanh or ReLU

Elman Network and vanishing gradient



Gradients might vanish ($\rightarrow 0$) as they propagate back through the computation graph

- Severe in deeper nets, especially in 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 $\boldsymbol{z}_{[i]} = \boldsymbol{u}_{[i]} \cdot \boldsymbol{v}_{[i]}$

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

- 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
- Note that the operation $+$ here is modulo 2

Gate example

Updating memory position 2

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

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

Gate example

Updating memory position 2

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

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

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 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 memory cell should be forgotten

There are three gates

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

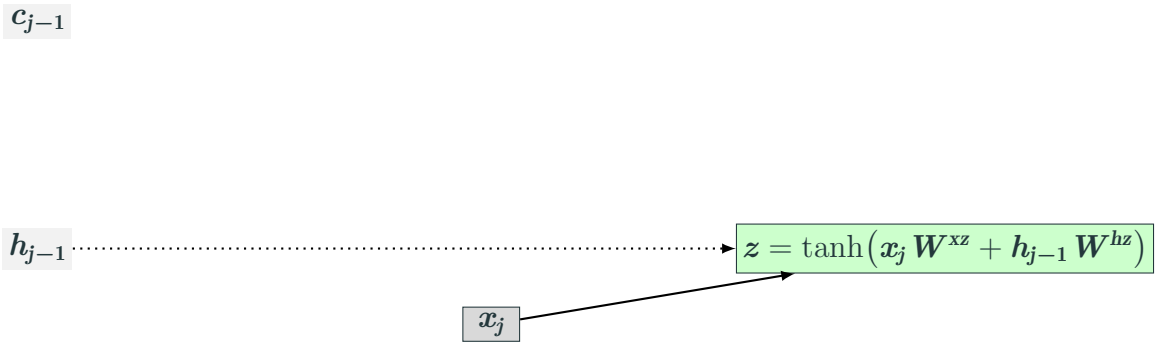
LSTM architecture

c_{j-1}

h_{j-1}

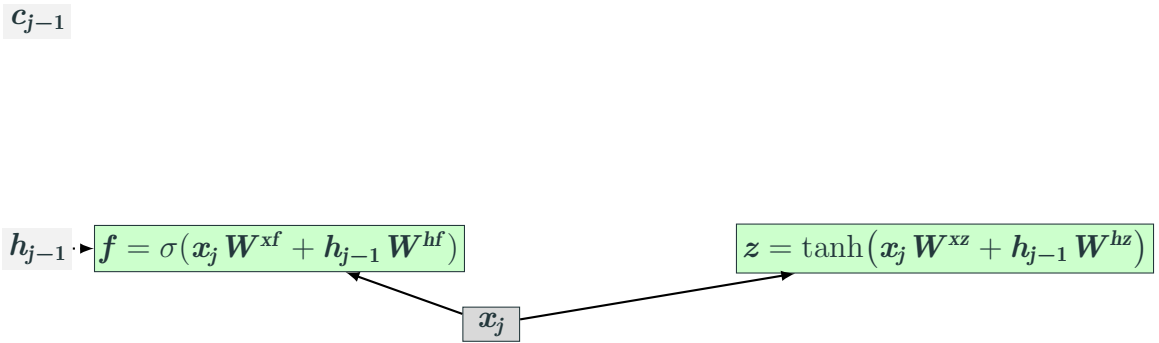
x_j

LSTM architecture



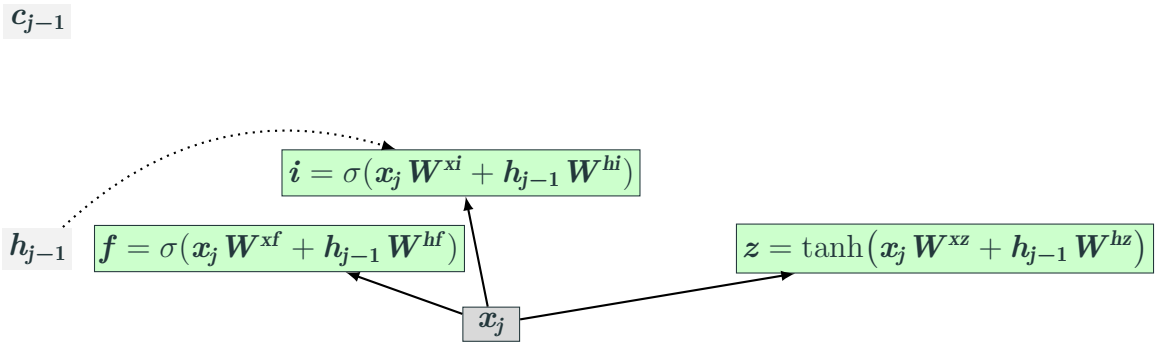
z — update candidate

LSTM architecture



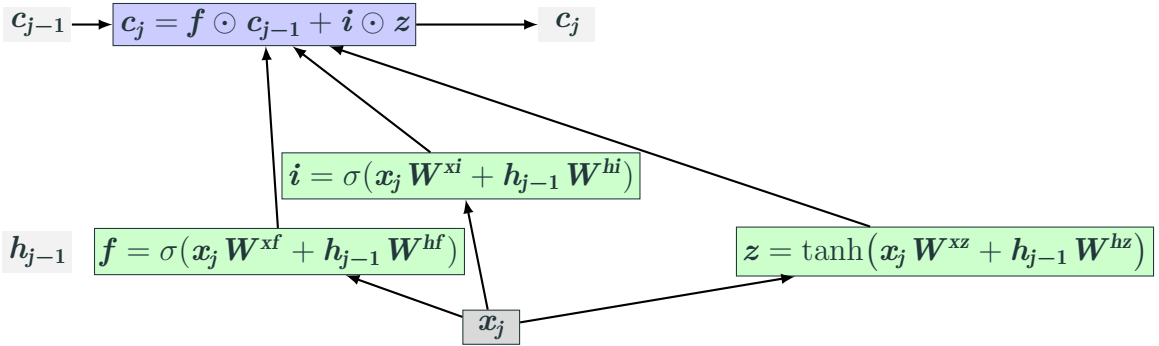
z — update candidate

LSTM architecture



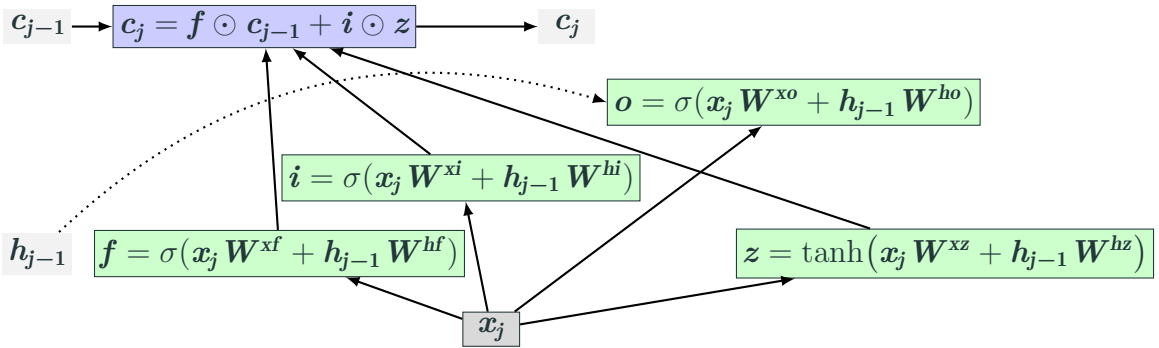
z — update candidate

LSTM architecture



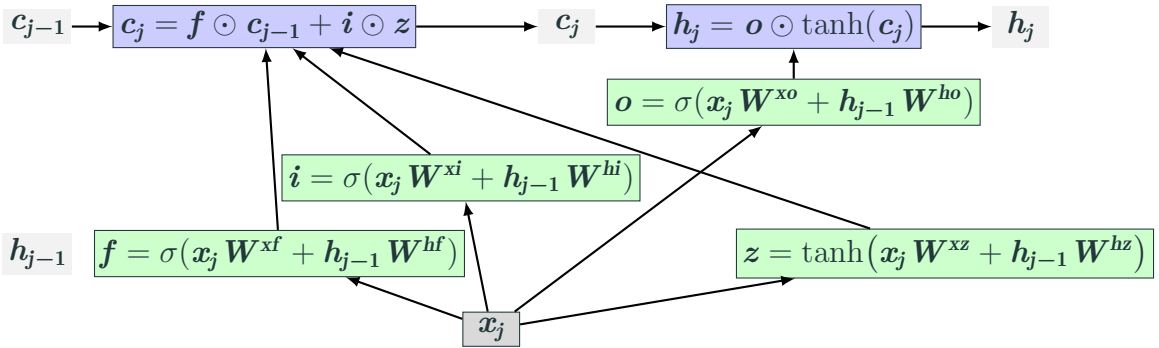
z — update candidate

LSTM architecture



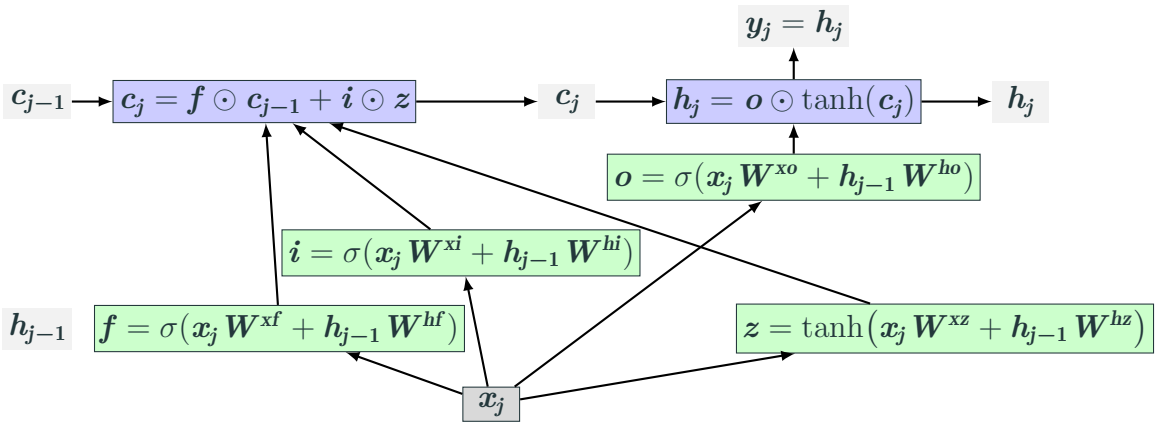
z — update candidate

LSTM architecture



z — update candidate

LSTM architecture

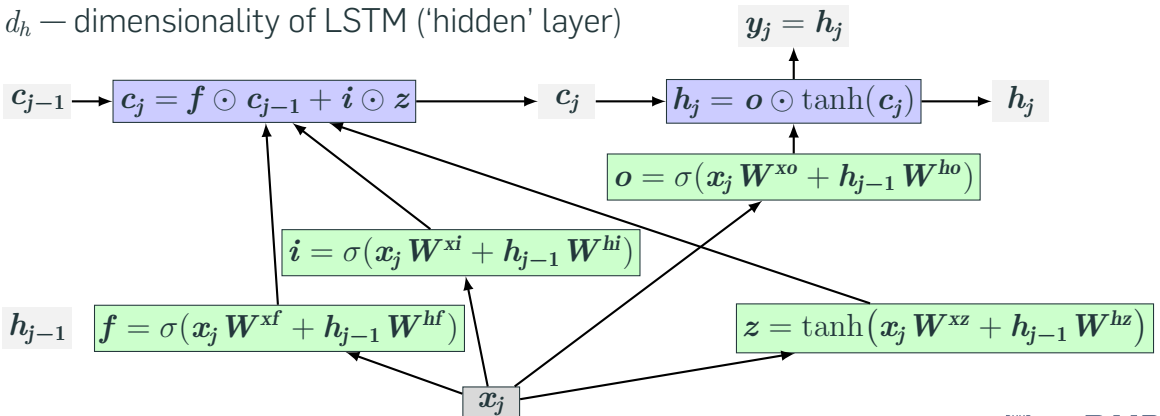


z — update candidate

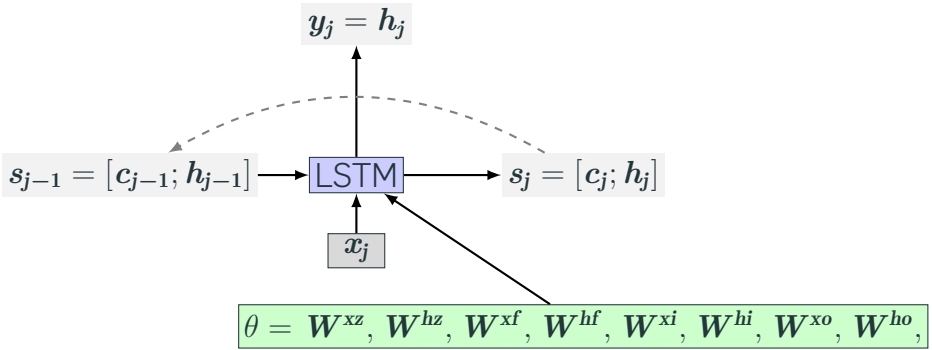
LSTM parameters and dimensions

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

d_h — dimensionality of LSTM ('hidden' layer)



LSTM as a ‘layer’



We also ignored bias terms for each gate

LSTM relevance in the Transformer era?

"How far do we get in language modeling when scaling LSTM to billions of parameters? So far, we can answer: At least as far as current technologies like Transformers or State Space Models. We have enhanced LSTM to xLSTM by exponential gating with memory mixing and a new memory structure."

M. Beck, K. Pöppel, M. Spanring, A. Auer, O. Prudnikova, M. Kopp, G. Klambauer, J. Brandstetter, and S. Hochreiter (2024). **"xLSTM: Extended Long Short-Term Memory".** In: *Advances in Neural Information Processing Systems 37*. Vancouver, BC, Canada: Neural Information Processing Systems Foundation, Inc. (NeurIPS), pp. 107547–107603

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

Licensed under Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

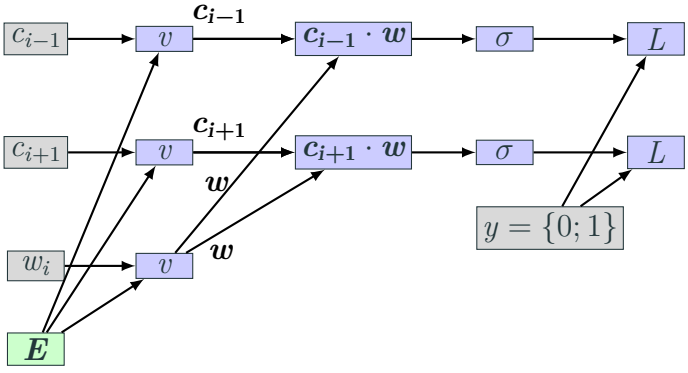
Credits

Ivan Habernal

Content from ACL Anthology papers licensed under CC-BY
https://www.aclweb.org/anthology

Appendix: Skip-Gram in word2vec

Skip-Gram — even more relaxed context notion



For k-element context $c_{1:k}$, treat as k different independent contexts (w_i, c_i)

Choosing the context

Sliding window approach — CBOW

Auxiliary tasks are created by taking a sequence of $2m + 1$ words

- The middle word is the target (focus) word
- The m words to each side is the context

Sliding window approach — Skip-Gram

$2m$ distinct tasks are created, each pairing the focus word with a different context word

Skip-gram-based approaches shown to be robust and efficient to train

FastText embeddings: Sub-word embeddings

FastText embeddings

Popular word embedding models ignore the morphology of words, by assigning a distinct vector to each word.

- Limitation, especially for languages with large vocabularies and many rare words

FastText embeddings

Popular word embedding models ignore the morphology of words, by assigning a distinct vector to each word.

- Limitation, especially for languages with large vocabularies and many rare words

$\rightarrow$ Model each word as a bag of character n-grams

- Each character n-gram has own embedding
- Word is represented as a sum of n-gram embeddings

P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov (2017). **"Enriching Word Vectors with Subword Information".** In: *Transactions of the ACL* 5, pp. 135–146

FastText embeddings example

Extract all the character n-grams for $3 \leq n \leq 6$

Example

`eating` $\rightarrow G_w =$ { `<ea`, `eat`, `ati`, `tin`, `ing`, `ng>`, `<eat`, `eati`, `atin`, `ting`, `ing>`, `<eati`, `eatin`, `ating`, `ting>`, `<eatin`, `eating`, `ating>` }

$$v(\text{eating}) = \sum_{g \in G_w} v(G_w)$$

Train with skip-gram and negative sampling (same as word2vec)

P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov (2017). **"Enriching Word Vectors with Subword Information".** In: *Transactions of the ACL* 5, pp. 135–146