Neural Methods for NLP

Course 5: Beyond words

Master LiTL --- 2021-2022

chloe.braud@irit.fr

Schedule v3

- S47 - C1: 23.11 13h30-15h30 (2H) → ML reminder + TP1
- S48 - C2: 30.11 13h30-15h30 (2H) → Intro DL + TP2
- S50 - C3: 14.12 13h30-16h30 (**3H**) → Embeddings + TP3
- S51 - 21.12: break
- S52 - 28.12: break
- S1 - C4: 06.01 13h-15h (2H) →[Start projects]
- S2 - C5: 14.01 10h-12h (2H) → Training a NN
- S4 - C6: 25.01 13h-15h (2H) → TP 4
- S5 - C7: 01.02 9h30-12h30 (3H) → Projects (fin TP4)
- **S5 - C8: 03.02 13h-15h30 (2h30) → CNN, RNN + TP5 (+ projet)**
- **S4 - C9: 15.02 10h-12h30 (2H30) → NLP applications + TP6 (+ projet)**
- **S6 - C10: 18.02 9h30-12h30 (3H) → Current challenges + project defense**

17.02: Assignments due (code + report)

Content

Problem: encoding sentences?

- Continuous BoW
- Convolutional NN
- Recurrent NN

TP5: Practical session with RNN

Beyond words

- Embeddings are not restricted to words
- Can equally be computed for sentences, paragraphs, documents
- Important trend in current research, with application in e.g. machine translation, information extraction..

Encoding sentences

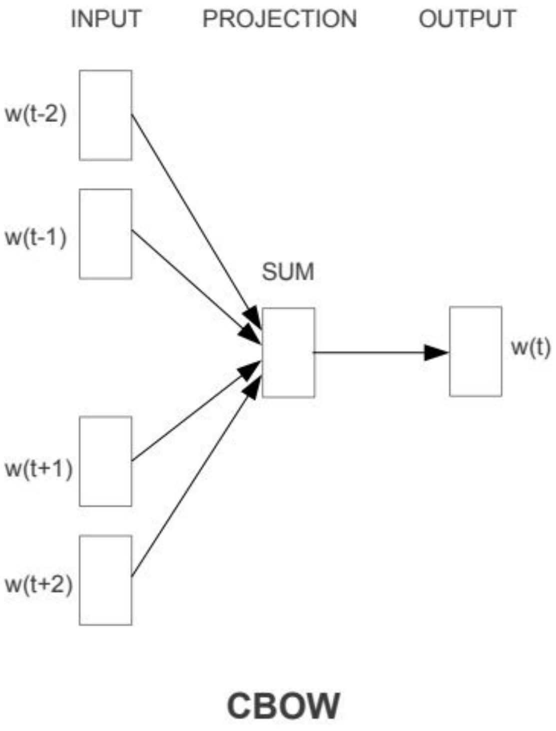
How to represent variable number of features, e.g. words in a sentence?

- Continuous Bag of Words (CBOW): sum embedding vectors of corresponding features
    - no ordering info e.g. ”not good quite bad” = ”not bad quite good”
- Convolutional layer
    - ’Sliding window’ approach that takes local structure into account
    - Combine individual windows to create vector of fixed size
- Recurrent layer
    - Allow to take into account the whole history / sequence

Continuous Bag of words

Variable number of features

- Feed-forward network assumes fixed dimensional input
    - How to represent variable number of features, e.g. words in a sentence, document?
- Continuous Bag of Words (CBOW): sum embedding vectors of corresponding features



Specialized architectures

The Feed-forward neural network are general purpose classification architectures: not tailored specifically for language data or sequences.

Today: 1D CNN and RNN

- CNN: specialized at identifying informative ngrams in a sequence of text, regardless of their position but while taking local ordering patterns into account
- RNN: designed to capture subtle patterns and regularities in sequences

Feature extraction

CNN and RNN architectures are primarily used as feature extractors

- not a standalone component
- rather used to produce a vector (or a sequence of vectors) that are then fed into further parts of the network

The network is trained end-to-end: the convolutional/recurrent part and the predicting part are trained jointly:

- the vectors resulting from the first part capture aspects of the input useful for the task

RNN are more used than CNN for text-based applications

Convolutional Neural Network (CNN)

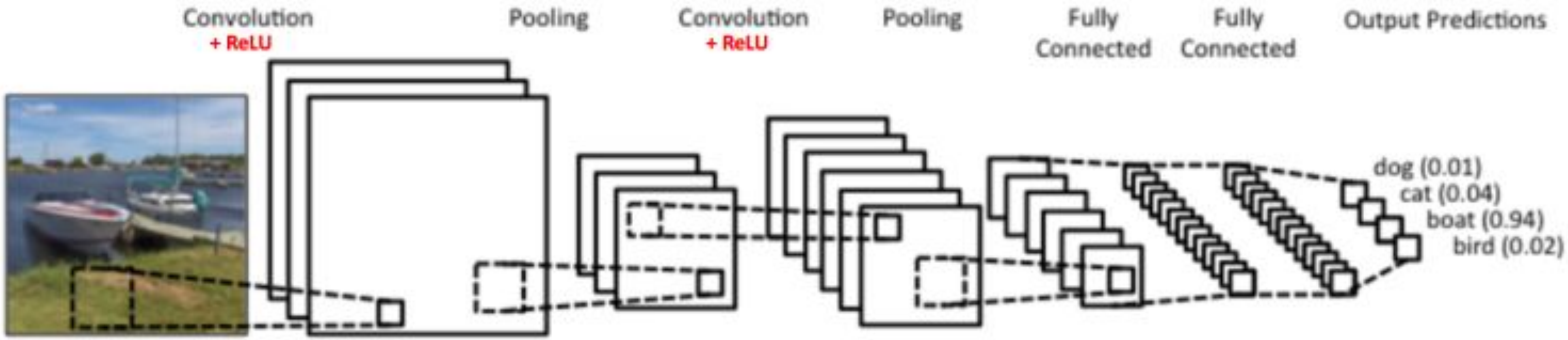
- [LeCun and Bengio, 1995]
- from vision community: great success as object detectors: recognizing an object from a predefined category (e.g. “cat”, “bicycles”) regardless of its position in the image [Krizhevsky et al. 2012]
- images: the architecture is usually 2D (grid) convolutions (or 3D with colors)
- text: 1D (sequence)

→ NLP: [Collobert et al. 2011]: semantic role labelling; [Kalchbrenner et al. 2014; Kim 2014]: sentiment and question-type classification

Convolutional Neural Network (CNN)

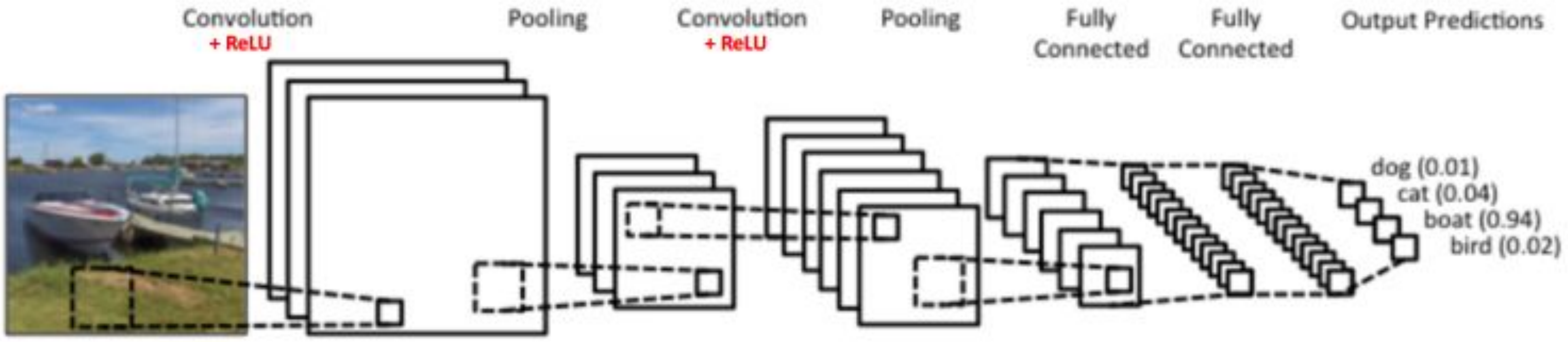
2 parts:

- Convolutions: the goal is to extract features specific to each input by compressing them ; the input goes through filters
- Classification: the output of the convolutional layers is given as input of an MLP



Convolutional Neural Network (CNN)

- Certain layers are not **fully** connected but **locally connected** (convolutional layers, pooling layers)
- same, local cues appear in different places in input (cf. vision)



CNN (intuition)

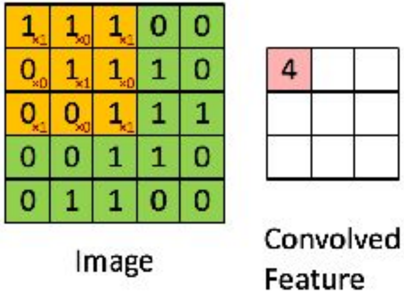
original picture

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

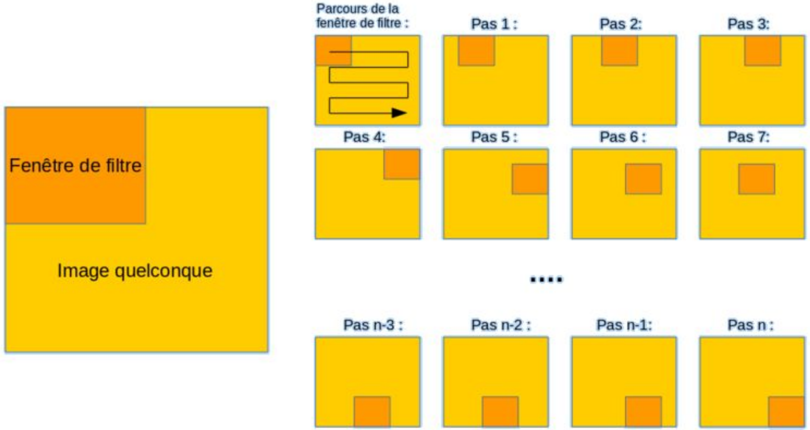
parameter = filter

1	0	1
0	1	0
1	0	1

Convolution computation



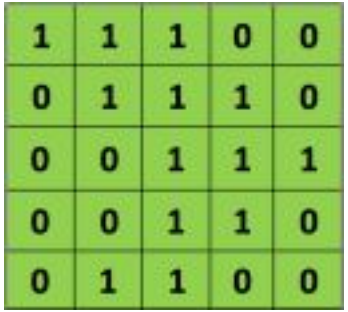
$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0



- define a window size
- go through the picture, using some step
- compute some operation on each filter window: linear function of the pixel values
- give you some value representing the sub-part (“receptive field”)
- at the end: a “feature map”, with a smaller shape

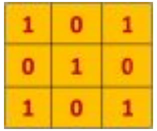
CNN (intuition)

original picture



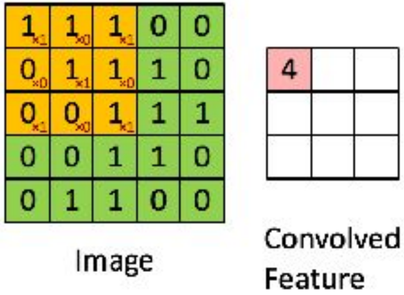
1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

parameter = filter

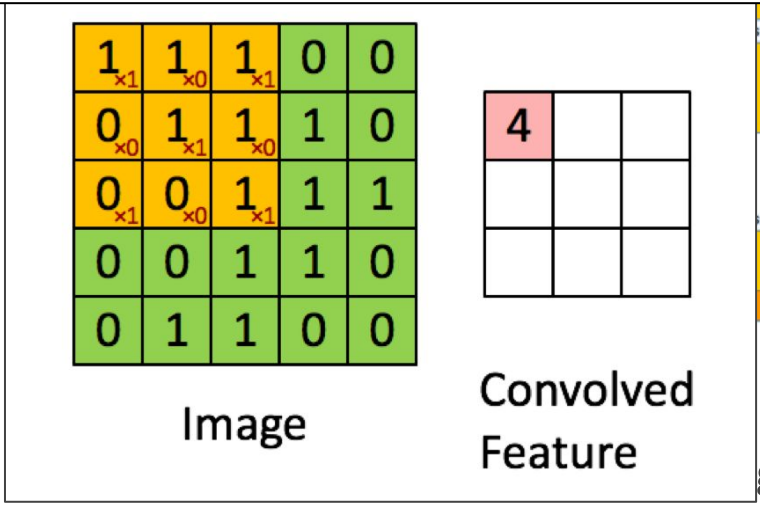


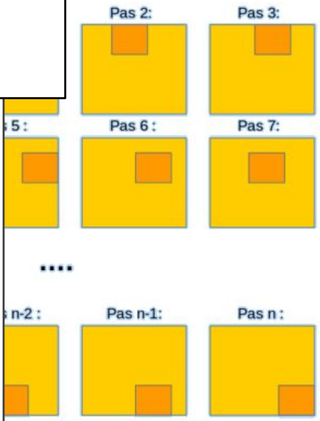
1	0	1
0	1	0
1	0	1

Convolution computation



element-wise multiplication:
(1x1)+(1x0)+(1x1)+(0x0)+(1x1)+(1x0)+(0x1)+(0x0)+(1x1)
=1+0+1+0+1+0+0+0+1=4





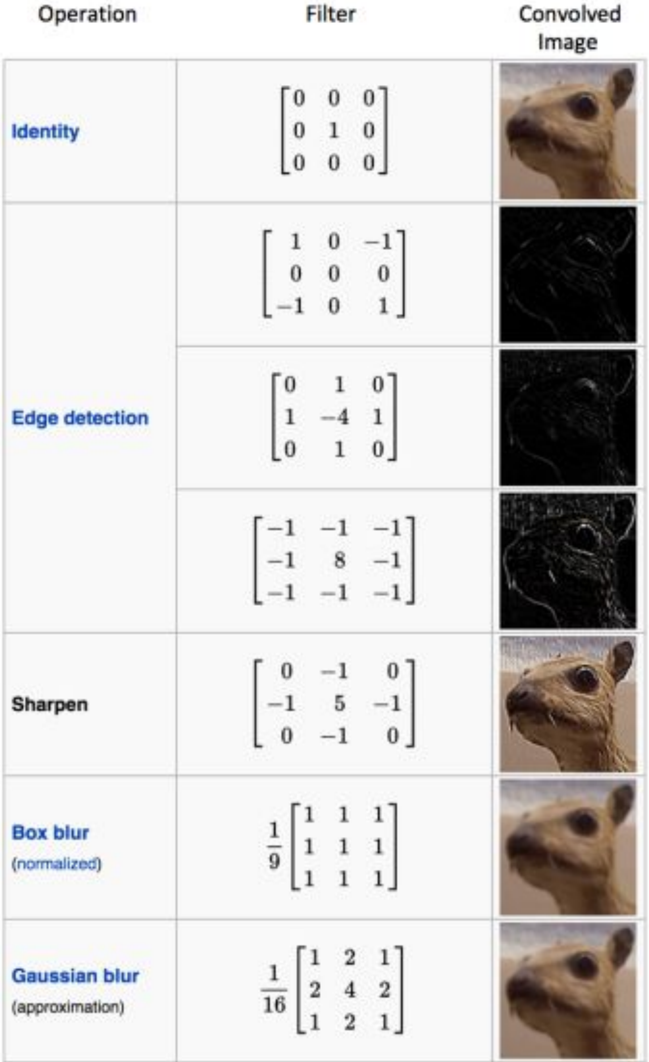
g some step

- compute some operation on each filter window: linear function of the pixel values
- give you some value representing the sub-part (“receptive field”)
- at the end: a “feature map”, with a smaller shape

Filters

Filter = feature detector

→ different values of the filter will produce different feature maps

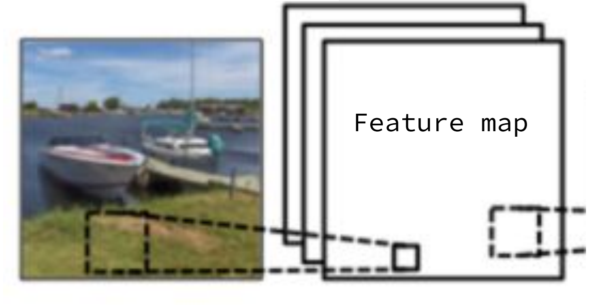


Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Setting

→ In practice, a CNN *learns* the values of these filters during training

- Depth: number of filters

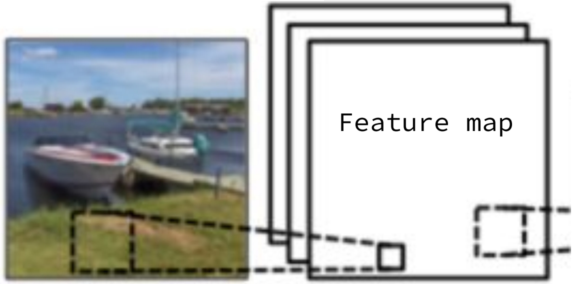


Here: 3 distinct filters producing 3 different feature maps = stacked 2d matrices

Setting

→ In practice, a CNN *learns* the values of these filters during training

- Depth: number of filters
- Stride: number of pixel by which we slide our filter window (larger stride, smaller feature map)

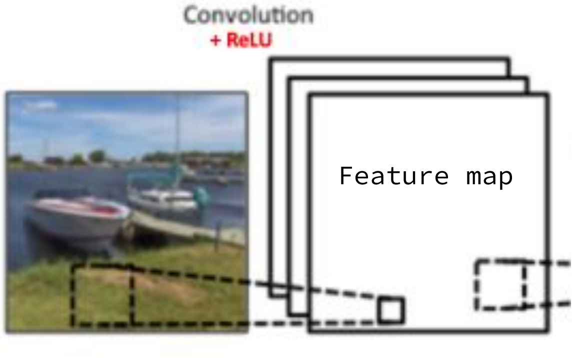


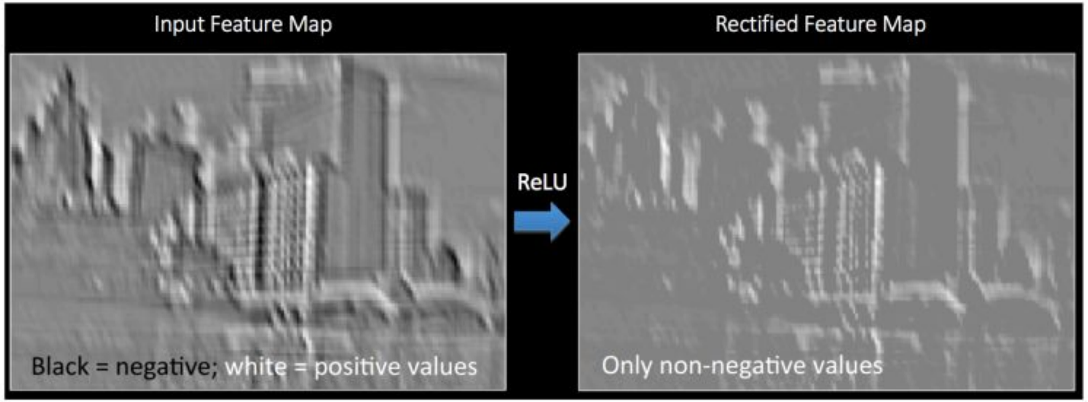
Here: 3 distinct filters producing 3 different feature maps = stacked 2d matrices

Setting

→ In practice, a CNN *learns* the values of these filters during training

- Depth: number of filters
- Stride: number of pixel by which we slide our filter window (larger stride, smaller feature map)
- Introducing non-linearity (ReLU): used after a convolution operation, replace all negative value in the feature map by zero

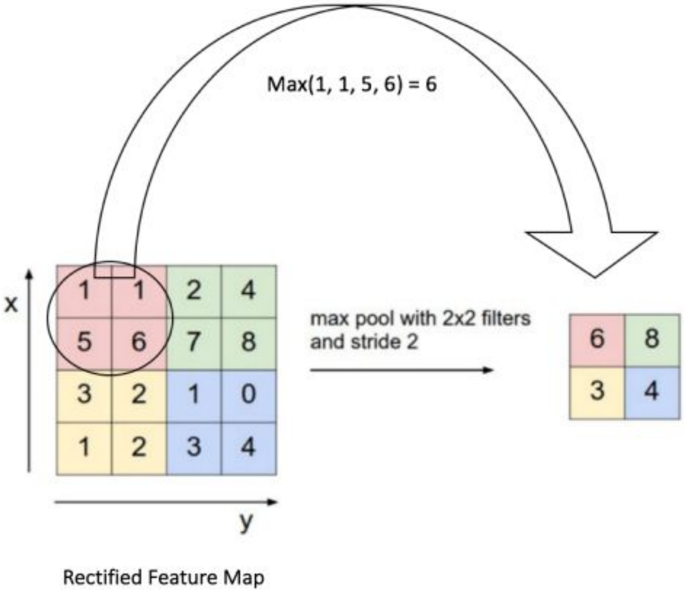




The pooling step

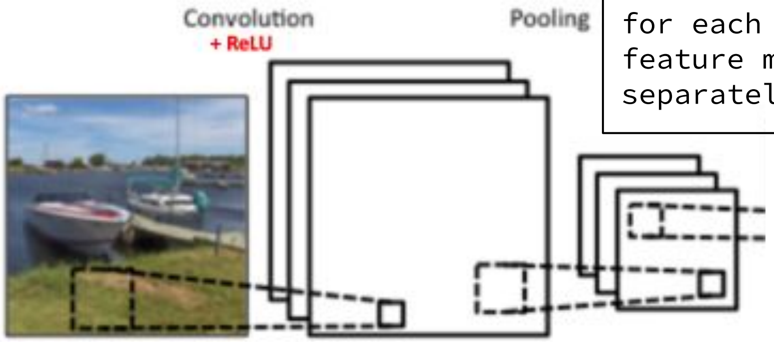
Spatial Pooling (or subsampling or downsampling) reduces the dimensionality of feature map but retains the most important information

- different types: Max, Average, Sum etc.
- Max Pooling: define a spatial neighborhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4



here done for each feature map separately

Pooling

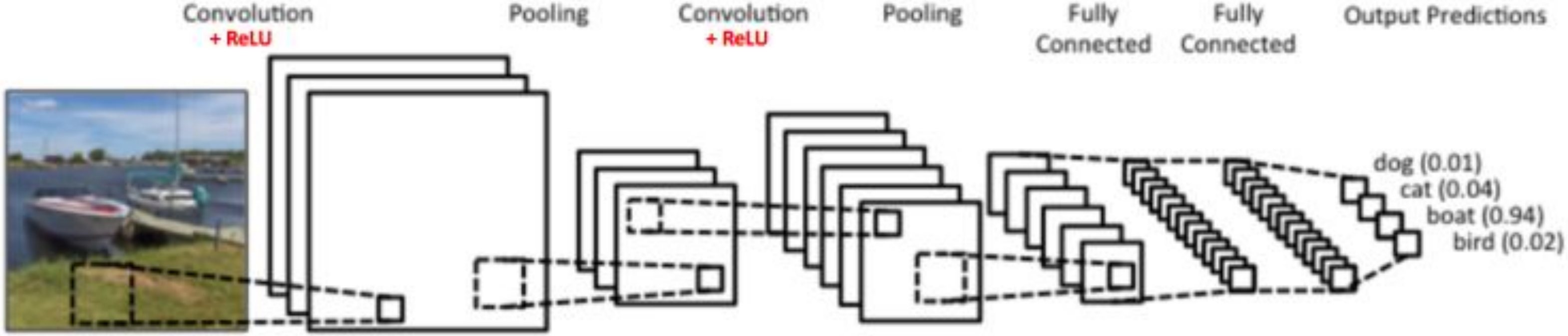
The function of Pooling is to progressively reduce the spatial size of the input representation. In particular, pooling:

- makes the input representations (feature dimension) smaller and more manageable
- reduces the number of parameters and computations in the network, therefore, controlling overfitting
- makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since we take the maximum / average value in a local neighborhood).
- helps us arrive at an almost scale invariant representation of our image (the exact term is “equivariant”). This is very powerful since we can detect objects in an image no matter where they are located.

Full network

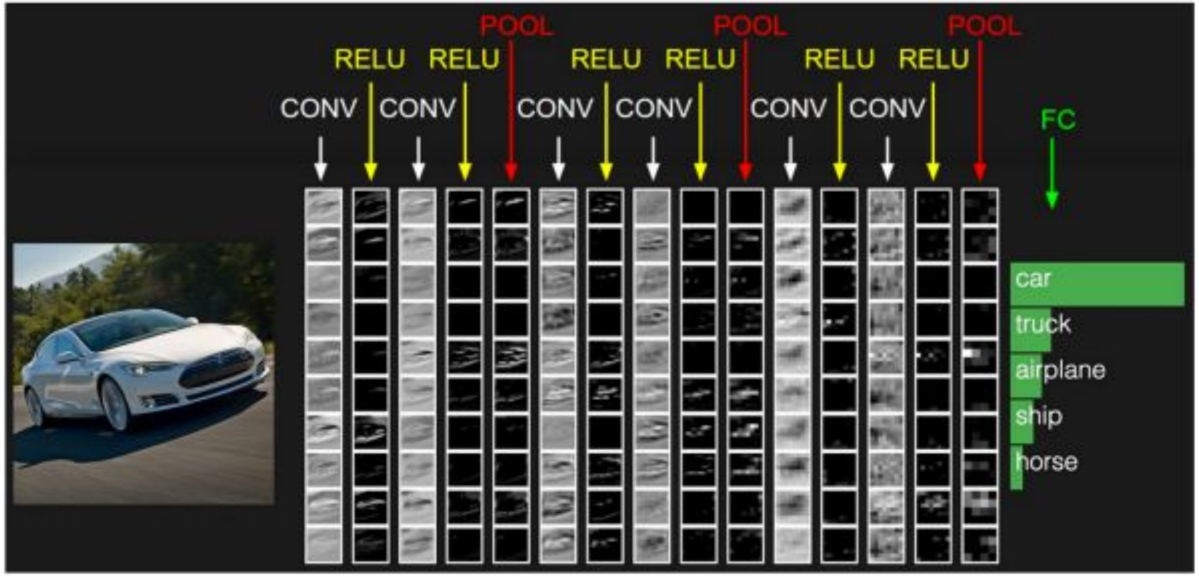
Two convolution steps:

- the 2nd convolution layer performs convolution on the output of the first Pooling Layer using six filters to produce a total of six feature maps
- these layers: extract useful features, introduce non-linearity, reduce feature dimensions
- Fully connected layer (MLP): classification + learn feature combinations



More on CNN

- pooling is not necessary after each conv
- unlimited number of conv
- What about text?

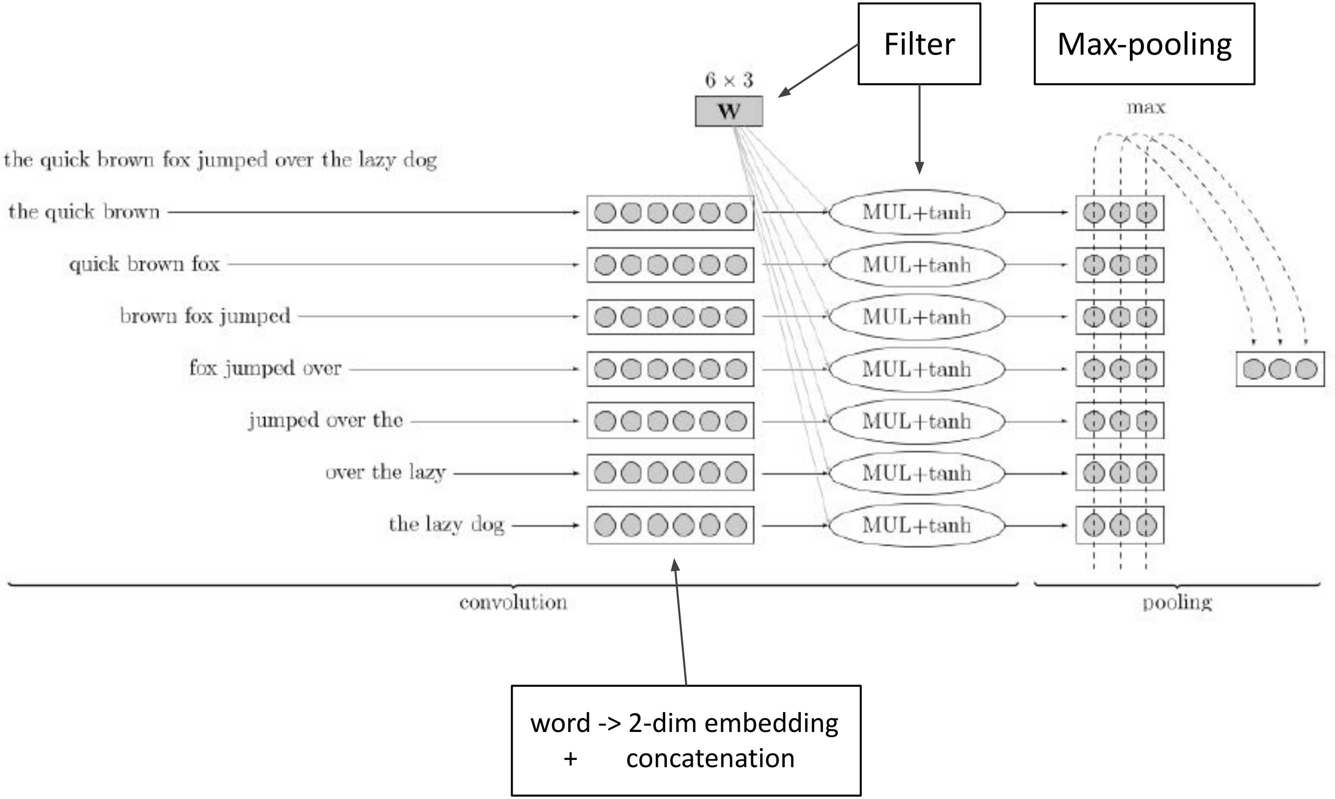


CNN for text data

- Goal: identify indicative local features (n-grams) in large structure, combine them into fixed size vector
- Convolution: apply filter to each window (linear transformation + non-linear activation)
- Pooling: combine by taking maximum

CNN for text data

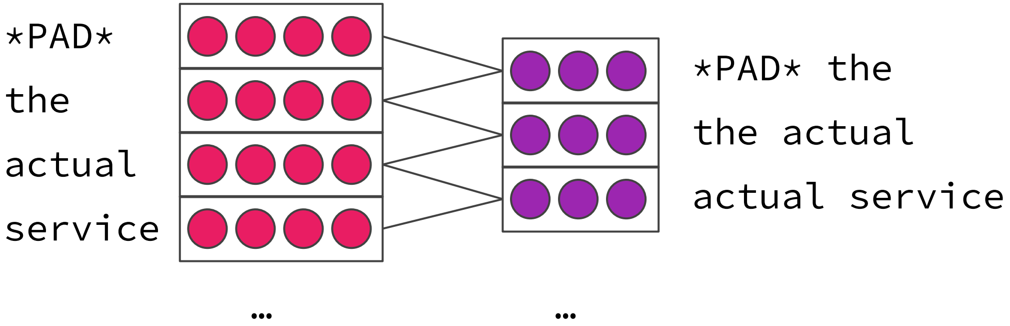
- define a window size
- go through the sentence
- compute some operation on each window
- return a vector for each window
- combine them using pooling



CNN for text data

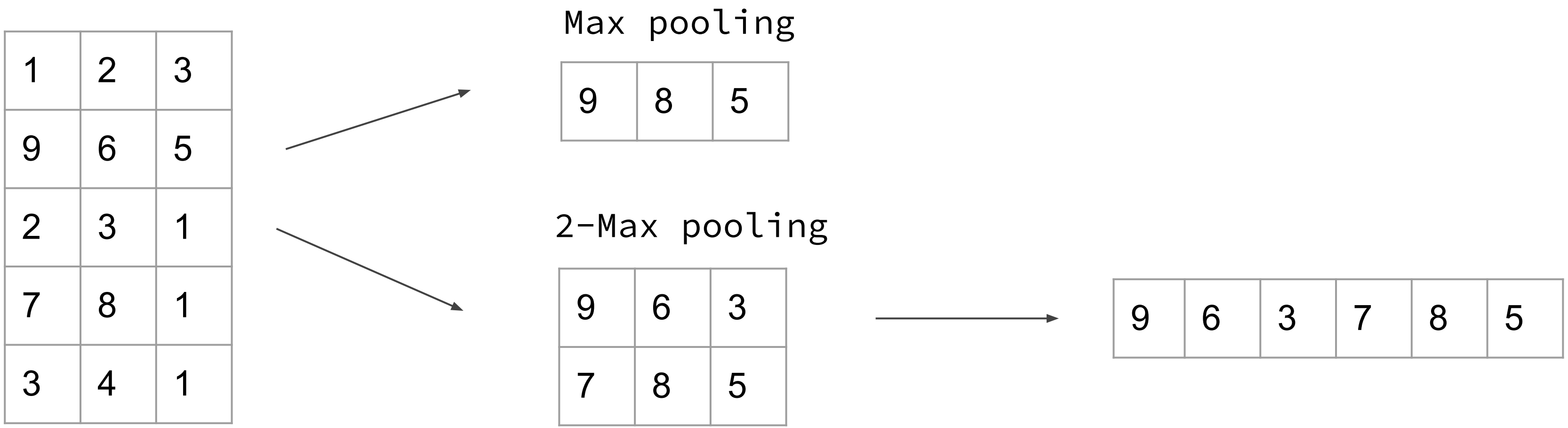
- main idea: apply the same parameterized function over all k-grams in the sequence
- creates a sequence of m vectors, each representing a particular k-gram
- the representation is sensitive to the identity and order of the words within a k-gram
- but the same representation will be used for a k-gram regardless of its position

Padding: add padding-words to each side of the sequence → wide convolution (vs narrow)



Pooling

- Most frequent: max-pooling
- Average pooling: taking the average value of each vector
- k-max pooling : top-k values in each dimension are retained, instead of only the best one = pool the k most active indicators, preserve orders
- (dynamic pooling: different pooling for different subpart of the input)



1	2	3
9	6	5
2	3	1
7	8	1
3	4	1

Max pooling

9	8	5

2-Max pooling

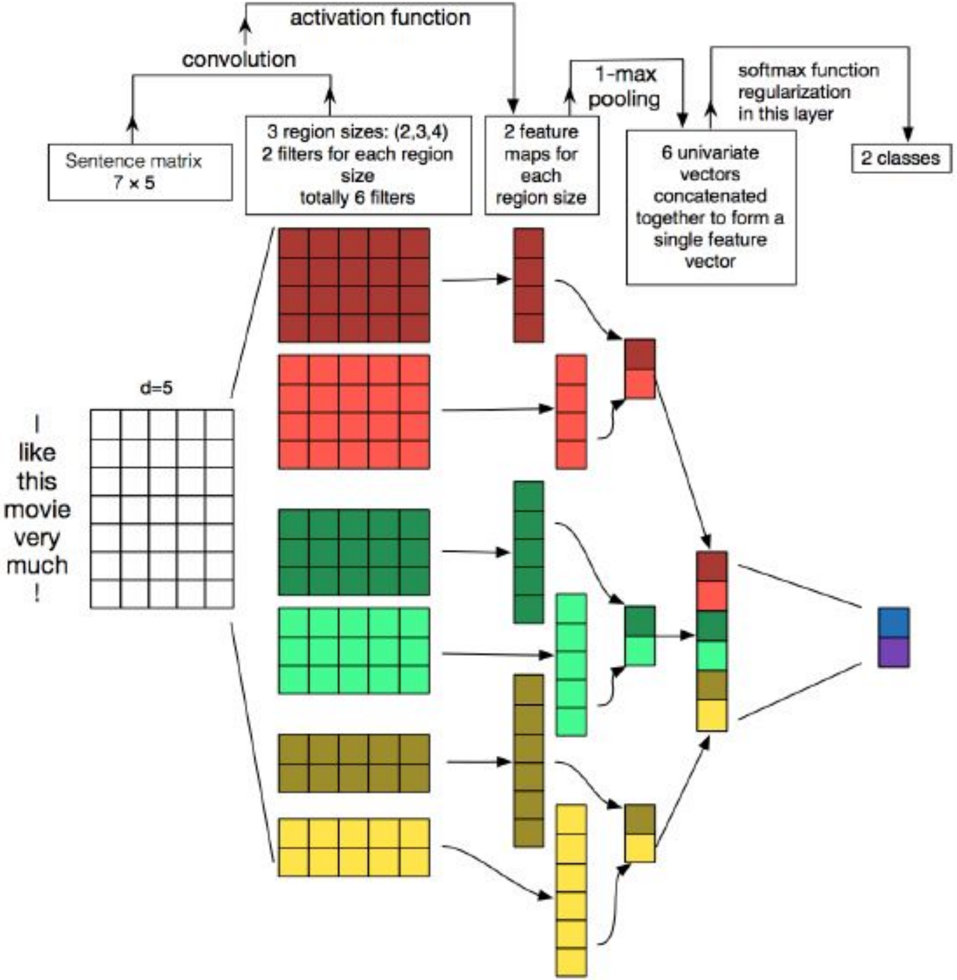
9	6	3
7	8	5

9	6	3	7	8	5

CNN for NLP

Variations:

- several convolutional layers, each with a different window size
- convolution over syntactic trees [Ma et al, 2015]: each window is around a node in the syntactic tree


activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d=5
I
like
this
movie
very
much
!

https://arxiv.org/pdf/1510.03820.pdf

Recurrent Neural Networks (RNN)

- Handle structured data of arbitrary sizes
- Recurrent networks for sequences: a type of *artificial neural network* designed to recognize patterns in sequences of data (e.g. text, genomes, numerical times series data emanating from sensors, stock markets)
- (Recursive networks for trees)
    - parsing can be down with recurrent network: a stack is seen as a sequence for transition-based parsing

RNN

Encoding arbitrary length sequences into a fixed size vector:

- CBOW: no ordering, no structure
- CNN: improvement, but only local patterns
- RNN: represent arbitrarily sized structured input as fixed-size vectors, paying attention to structured properties
    - gated architectures: LSTM, GRU are very powerful at capturing regularities in sequential inputs
    - RNNs condition the next word on the entire sequence history

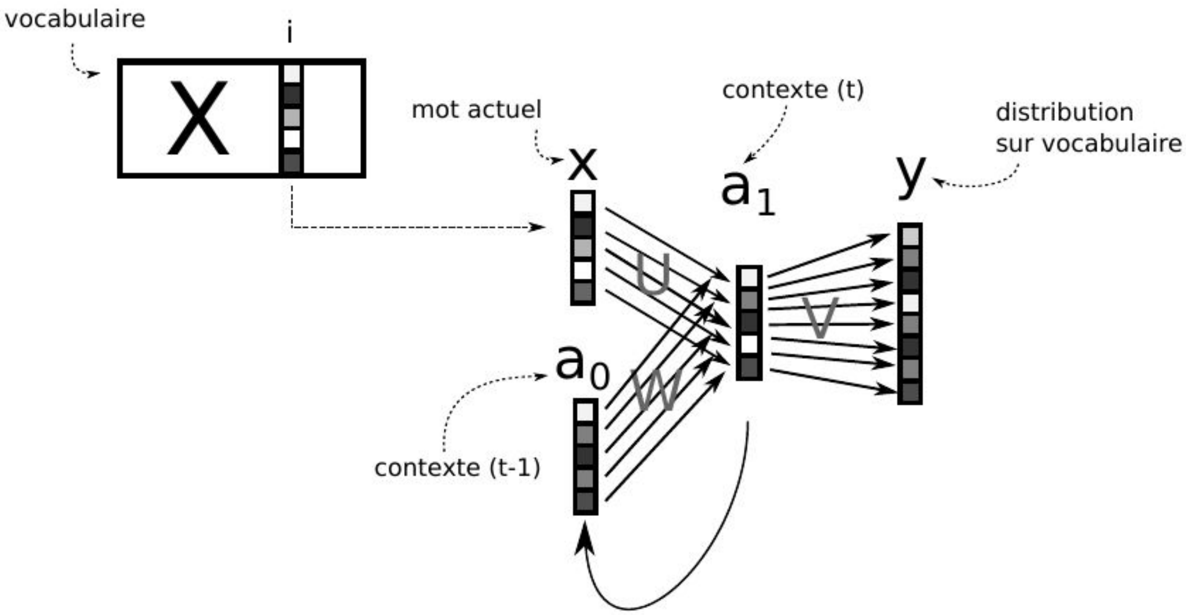
RNN

- Main idea: if we have data in a sequence such that one data point depends upon the previous data point, we need to modify the neural network to incorporate the dependencies between these data points
- RNNs have the concept of ‘memory’ that helps them store the states or information of previous inputs to generate the next output of the sequence.

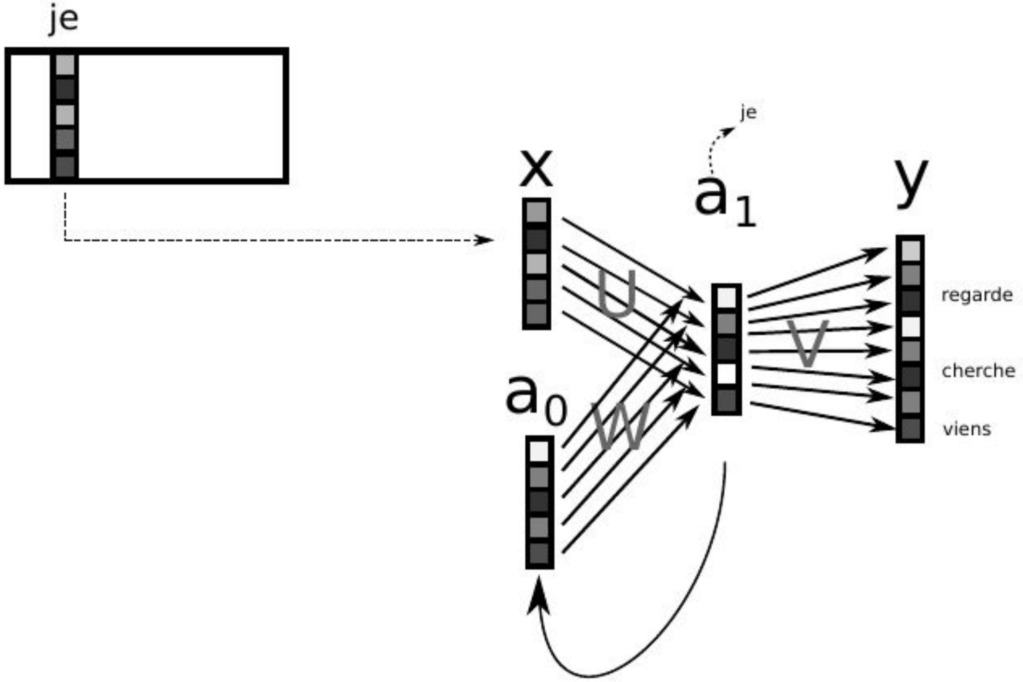
Suppose you want your network to predict the next word in a sentence: you need the previous outputs/words, to predict the next word.

- In Feedforward Networks, every output is independent of the previous output. So basically, the output at time t is independent of output at time t-1 → sequence dependent data problems

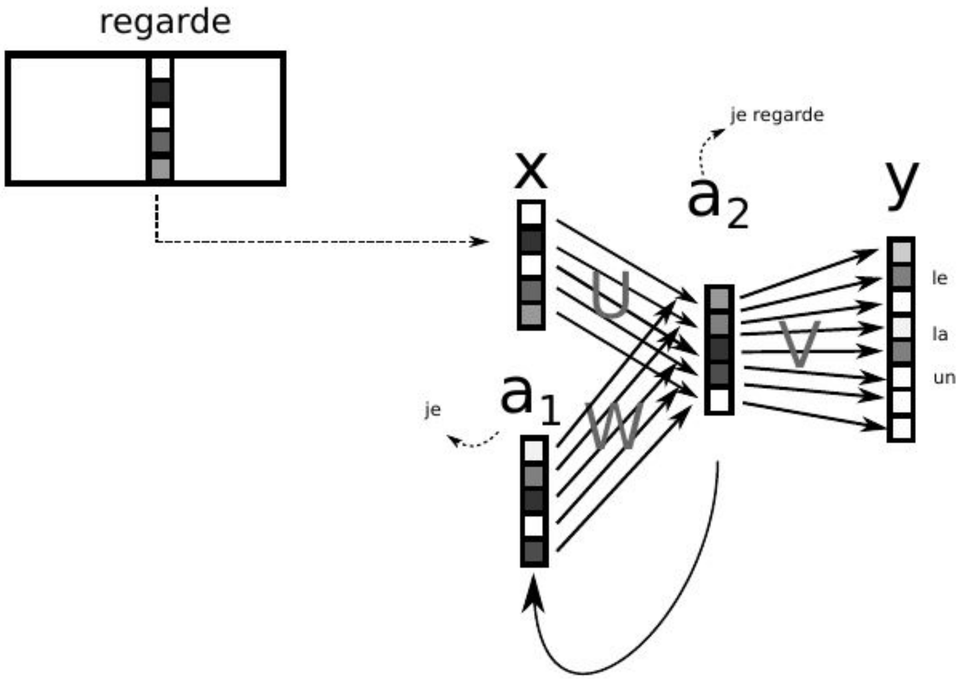
RNN: modèle de langue



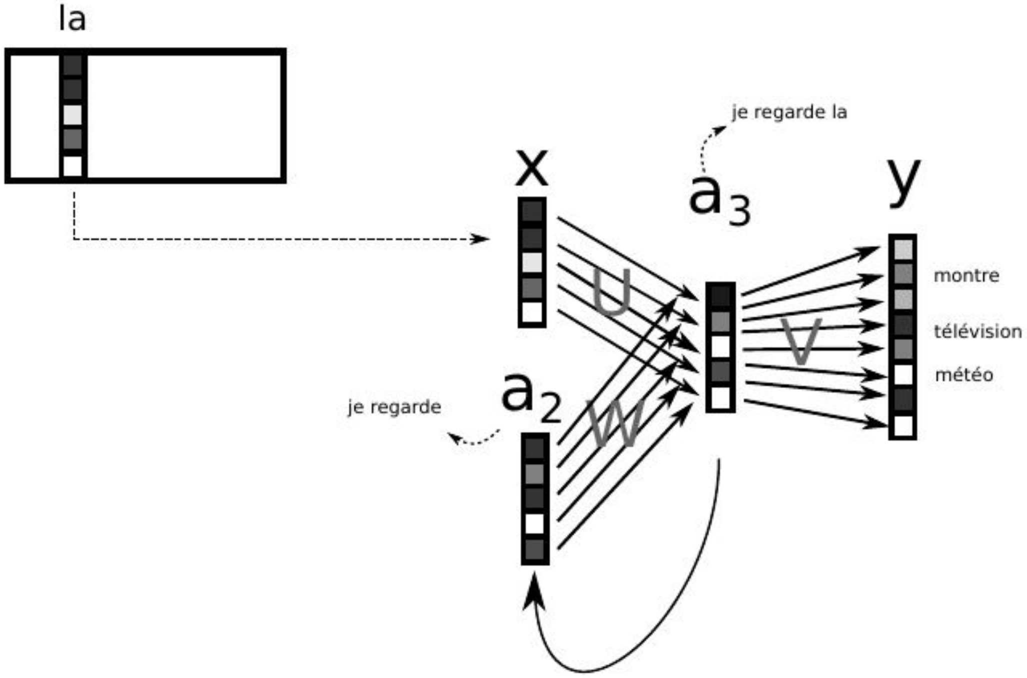
RNN: modèle de langue



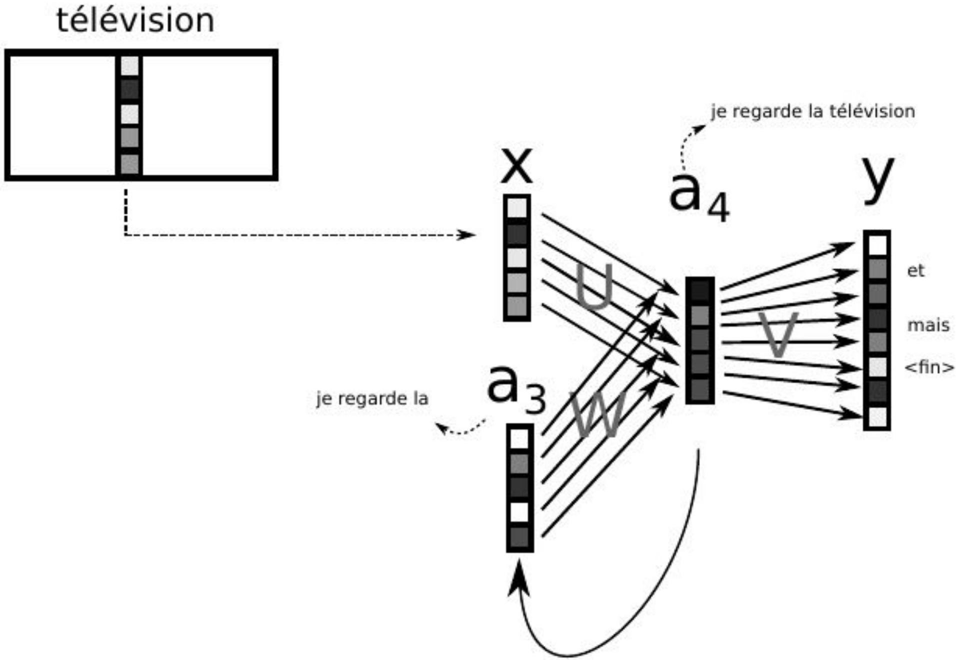
RNN: modèle de langue



RNN: modèle de langue

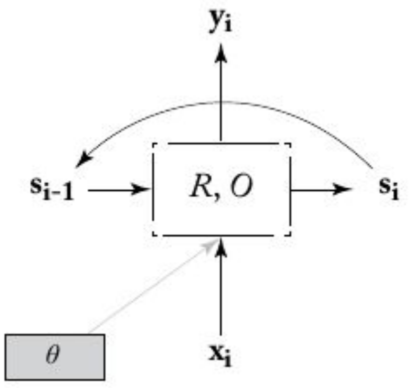


RNN: modèle de langue



RNN abstraction

- a sequence of input: $x_1, x_2, \ldots., x_n$
- a sequence of output: $y_1, y_2, \ldots, y_n$
- idea: y_i depends on x_i but also $x_1, \ldots, x_{i-1}$
- we have a recursive function R:
  - takes the current input vector x_i
  - and a vector representing the current state of m
  - output: a new state vector s_i
- the final representation y_i is based on this s
  - in practice: $y_i = s_i$ or y_i = a subpart of s_i



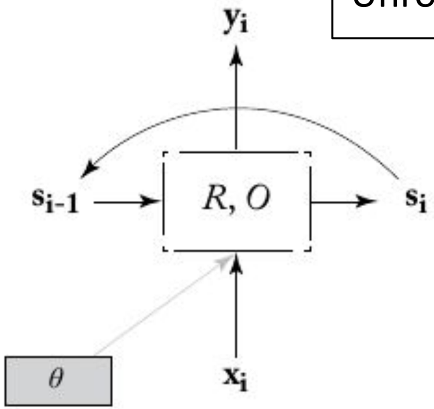
$$\mathrm{RNN}^{\star}(\boldsymbol{x}_{1:n}; \boldsymbol{s}_0) = \boldsymbol{y}_{1:n}$$

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

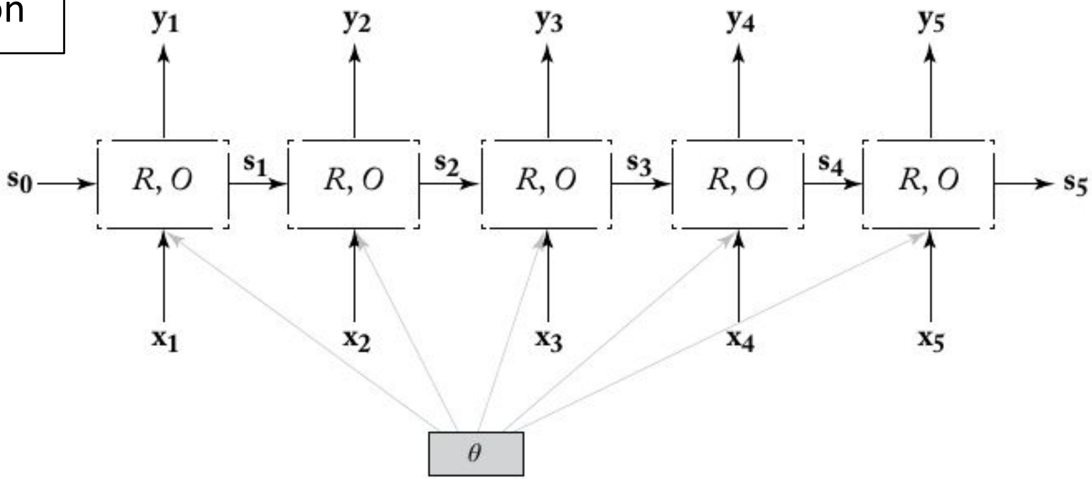
$$\boldsymbol{s}_i = R(\boldsymbol{s}_{i-1}, \boldsymbol{x}_i)$$

$$\boldsymbol{x}_i \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{y}_i \in \mathbb{R}^{d_{out}}, \quad \boldsymbol{s}_i \in \mathbb{R}^{f(d_{out})}.$$

RNN abstraction

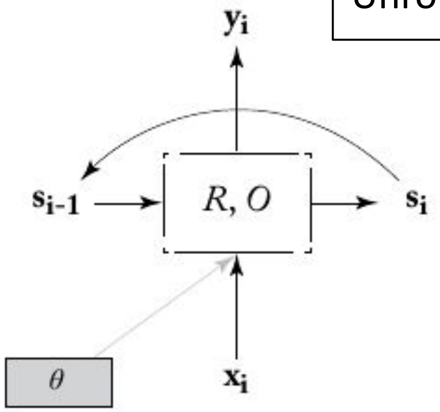


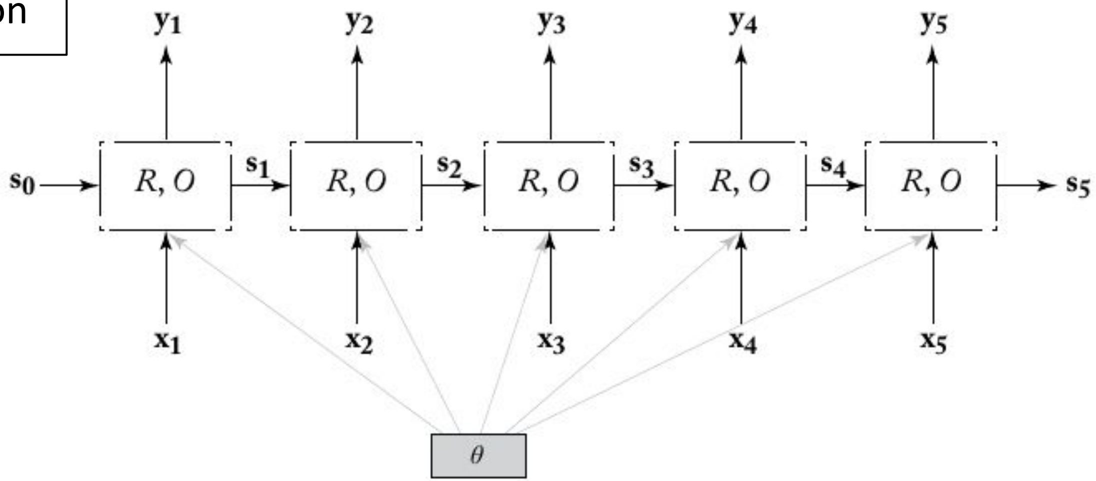
Unroll the recursion



RNN abstraction

Unroll the recursion



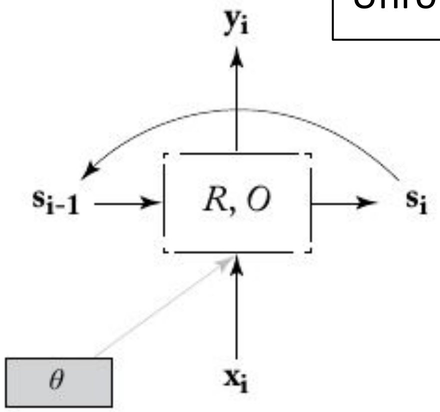


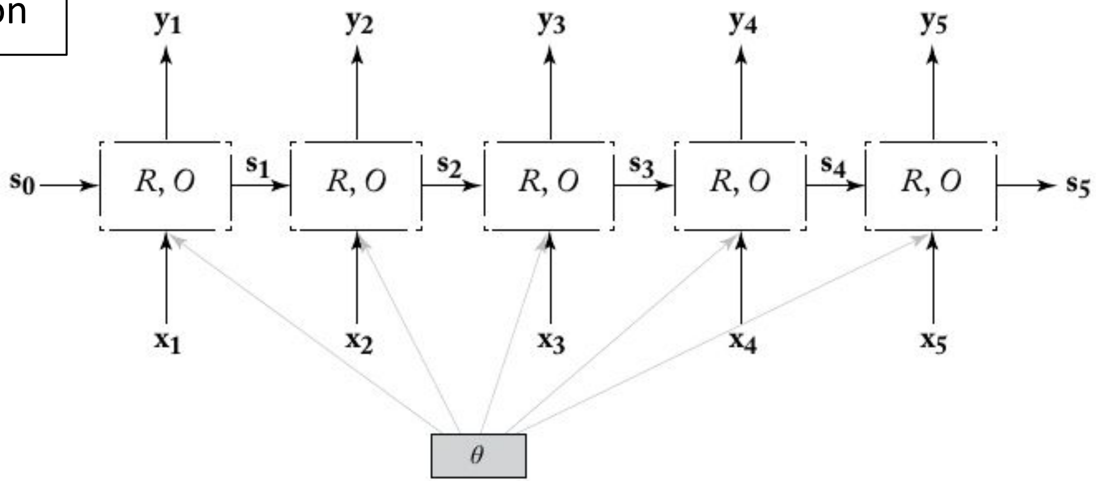
The same parameters are shared across all time steps

RNN abstraction

$$
\begin{aligned}
s_4 &= R(s_3, x_4)\\
&= R(\overbrace{R(s_2, x_3)}^{s_3}, x_4)\\
&= R(R(\overbrace{R(s_1, x_2)}^{s_2}, x_3), x_4)\\
&= R(R(R(\overbrace{R(s_0, x_1)}^{s_1}, x_2), x_3), x_4).
\end{aligned}
$$

Unroll the recursion





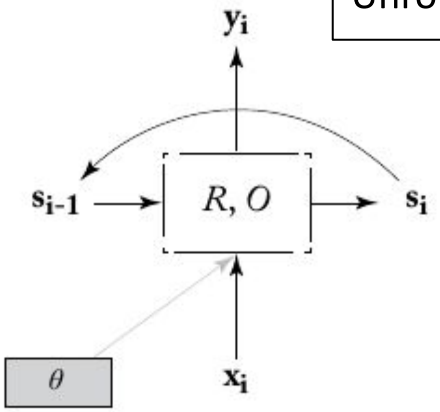
The same parameters are shared across all time steps

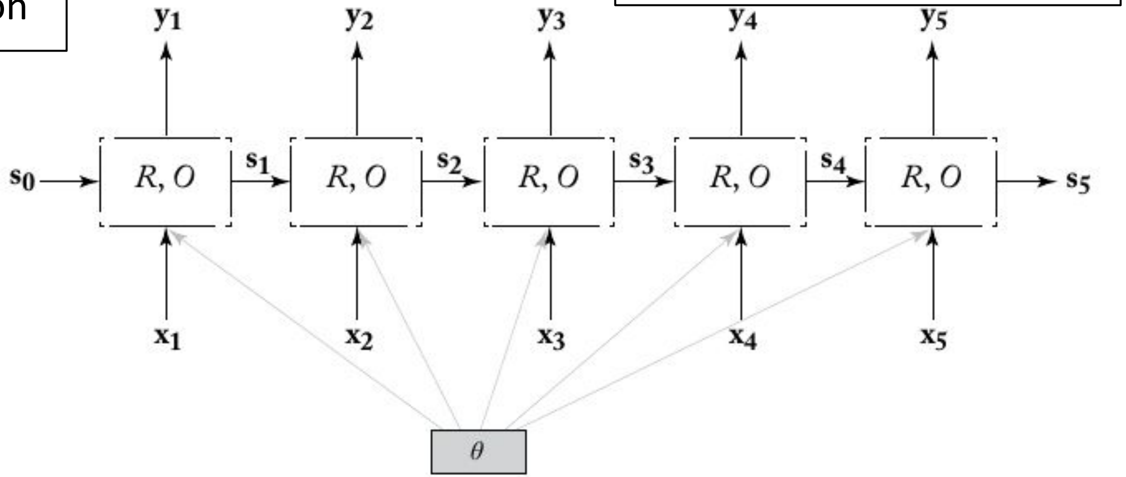
RNN abstraction

$$\begin{aligned}
s_4 &= R(s_3, x_4) \\
&= R(\overbrace{R(s_2, x_3)}^{s_3}, x_4) \\
&= R(R(\overbrace{R(s_1, x_2)}^{s_2}, x_3), x_4) \\
&= R(R(R(\overbrace{R(s_0, x_1)}^{s_1}, x_2), x_3), x_4).
\end{aligned}$$

- s1 depends on x1
- s2 depends on x1, x2
- s2 depends on x1,x2,x3

Unroll the recursion





The same parameters are shared across all time steps

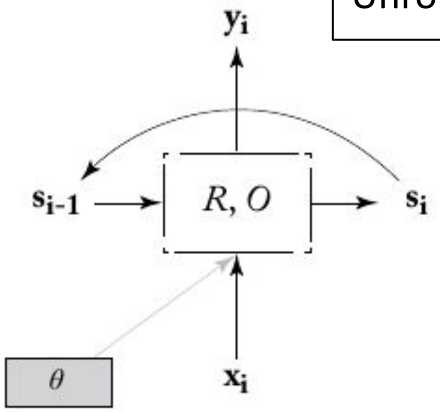
RNN abstraction

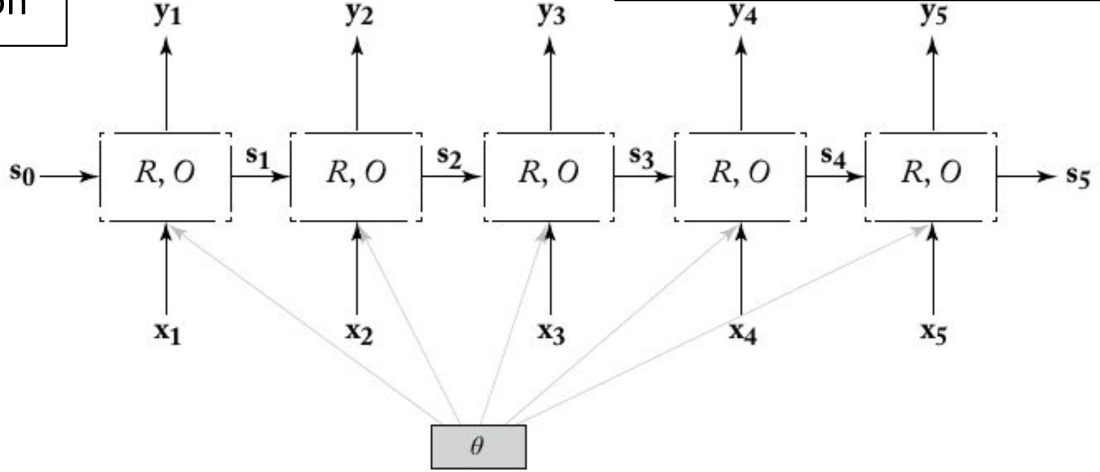
$$
\begin{aligned}
s_4 &= R(s_3, x_4) \\
&= R(\overbrace{R(s_2, x_3)}^{s_3}, x_4) \\
&= R(R(\overbrace{R(s_1, x_2)}^{s_2}, x_3), x_4) \\
&= R(R(R(\overbrace{R(s_0, x_1)}^{s_1}, x_2), x_3), x_4).
\end{aligned}
$$

- s1 depends on x1
- s2 depends on x1, x2
- s2 depends on x1,x2,x3

→ sn (and yn) depends on the entire input = **encoding of the input sequence**

Unroll the recursion





The same parameters are shared across all time steps

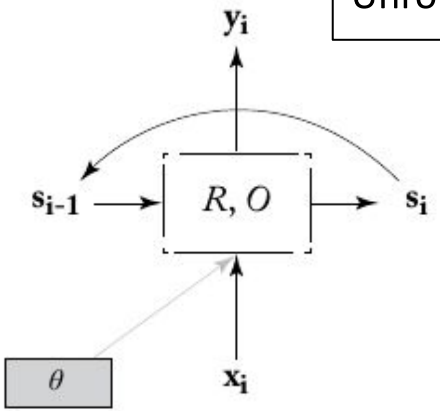
RNN abstraction

$$\begin{aligned} s_4 &= R(s_3, x_4) \\ &= R(\overbrace{R(s_2, x_3)}^{s_3}, x_4) \\ &= R(R(\overbrace{R(s_1, x_2)}^{s_2}, x_3), x_4) \\ &= R(R(R(\overbrace{R(s_0, x_1)}^{s_1}, x_2), x_3), x_4). \end{aligned}$$

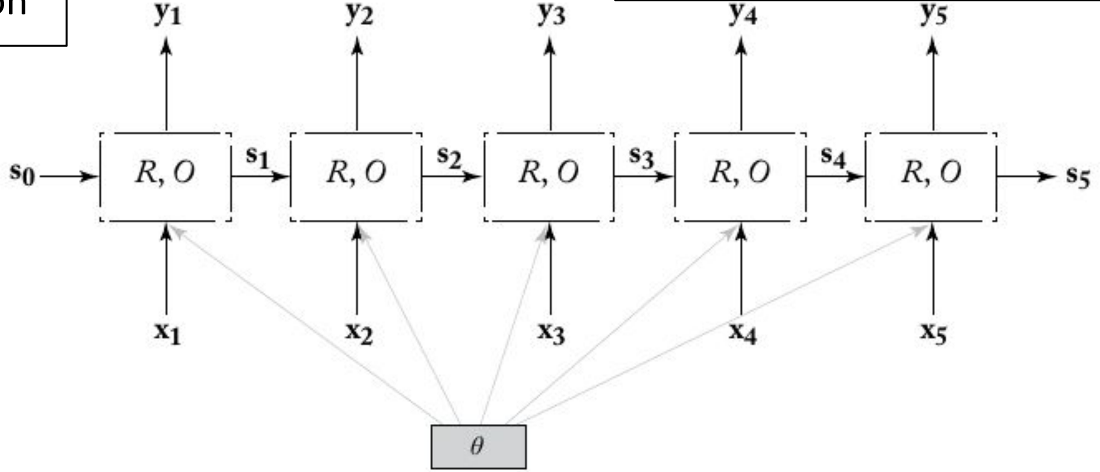
- s1 depends on x1
- s2 depends on x1, x2
- s2 depends on x1,x2,x3

→ sn (and yn) depends on the entire input = **encoding of the input sequence**

→ training must set parameters in order to have a state that conveys useful info



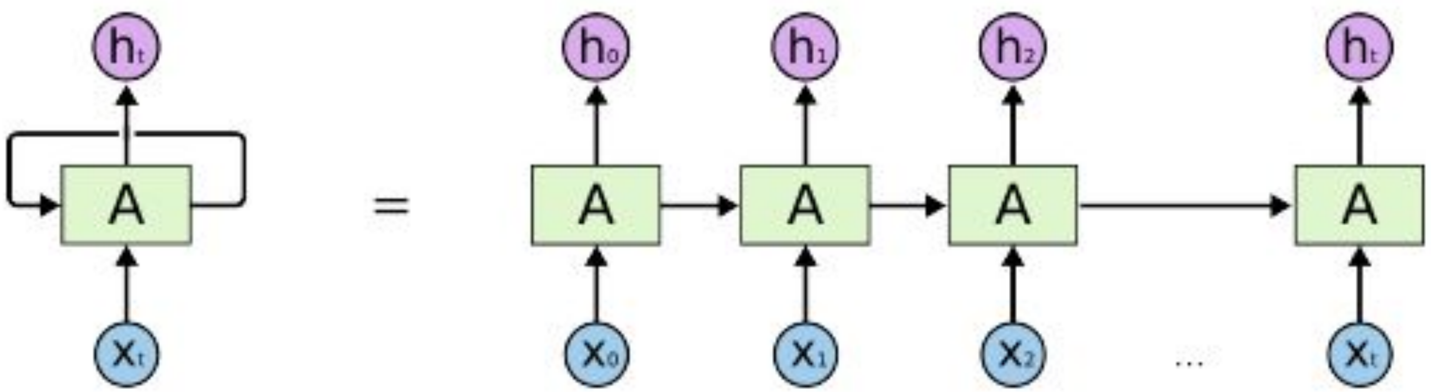
Unroll the recursion



The same parameters are shared across all time steps

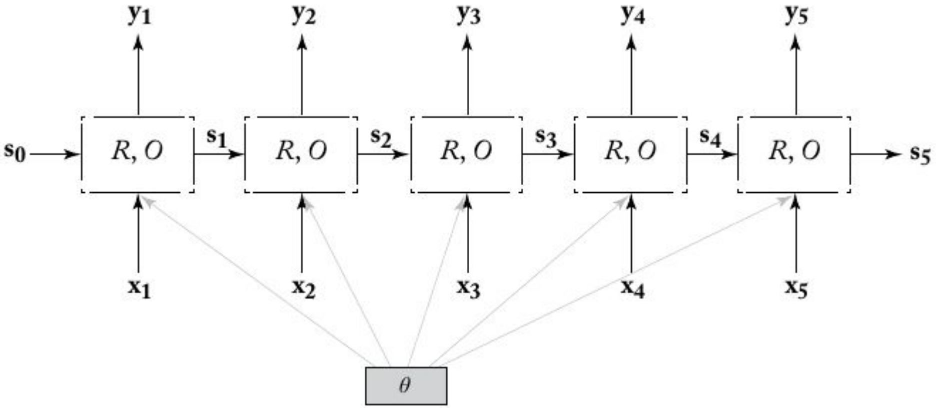
RNN

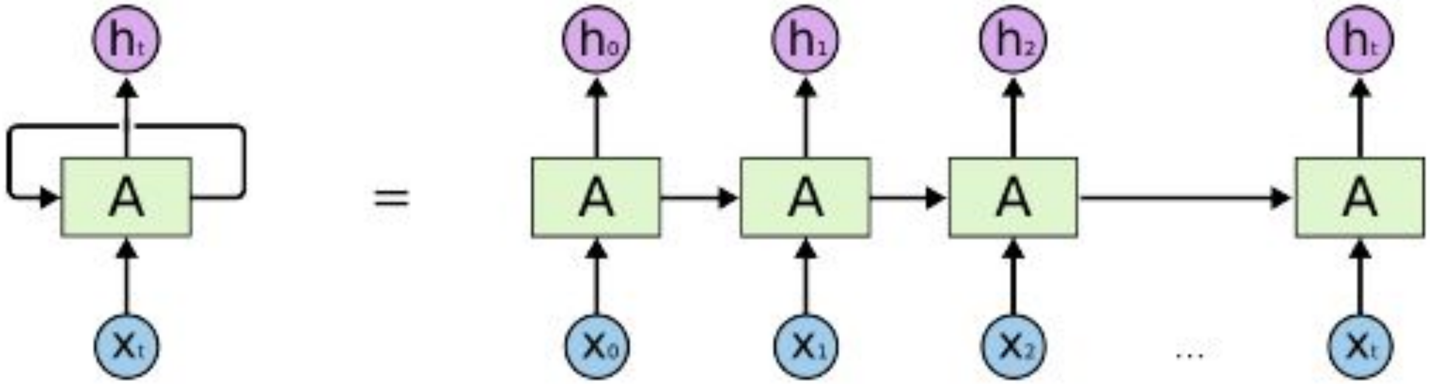
- Consider recurrent neural network as very deep neural network with shared parameters across computation
- Backpropagation through time



RNN abstraction

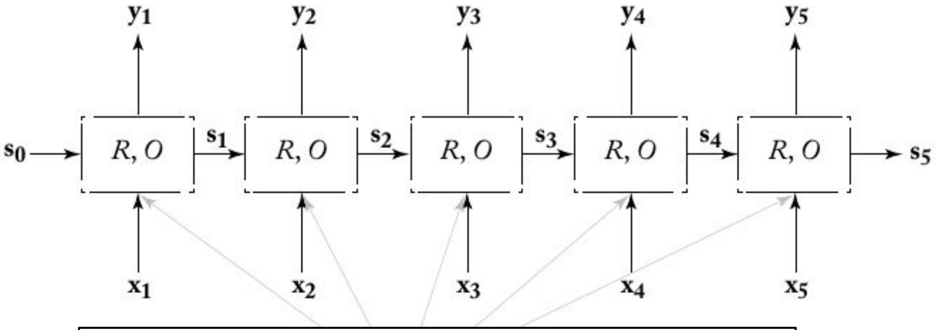
- Consider recurrent neural network as very deep neural network with shared parameters across computation
- Backpropagation through time





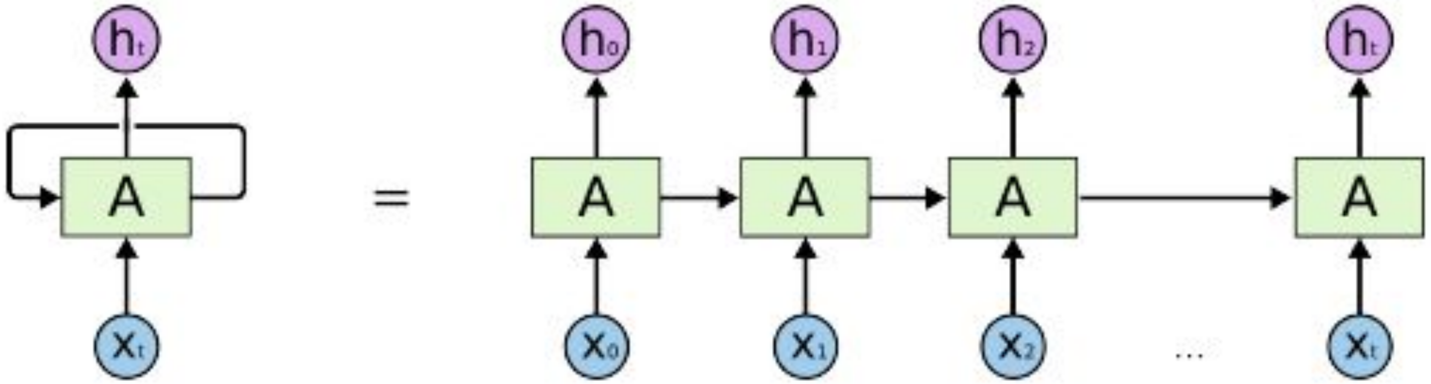
RNN abstraction

- Consider recurrent neural network as very deep neural network with shared parameters across computation
- Backpropagation through time



A note on notations:

- states: noted ***s*** or ***h*** (but both for LSTM)
- ***h*** and ***y*** sometimes merged



RNN training

What kind of supervision? RNN does not do much on its own, it’s a trainable component

⟶ the RNN learns to encode properties of the input sequences that are useful for the further prediction task

Some common architectures:

- Acceptor / Encoder: based on the final output (e.g. text classification)
- Transducer: an output for each input (e.g. language modeling)
    - Encoder-decoder: one RNN to encode sequence into vector representation, another RNN to decode into sequence (e.g. machine translation)

Acceptor

The supervision signal is based on the final state / output vector y_n

→ Acceptor: we observe the final state, and decide on an outcome, e.g.:

- Read characters of a word one by one and use the final state to predict the POS [Ling et al. 2015]
- Read a sentence and based on the final state decide if it conveys positive or negative sentiment [Wang et al 2015]
- Read a sequence of words and decide whether it's a valid noun-phrase

The RNN's output vector is fed into a fully connected layer or an MLP which make the prediction (same loss as previously)

Encoder

Similar: uses only the final output vector y_n

- but here, the prediction is not solely on the basis of this vector
- the final vector is treated as an encoding of the information in the sequence
- and is used as additional information together with other signals

e.g. Extractive document summarization:

- run over the document with a RNN: y_n summarizes the document
- then y_n is used with other features to select the sentences to be kept

Transducer

Producing an output $\mathbf{y_i}$ for each input read

e.g. Sequence tagger:

- $\mathbf{x_{1:n}}$ are feature representations for the n words of a sentence
- $\mathbf{y_i}$ is the input for predicting the tag assigned to word i (based on words $w_{1:i}$)

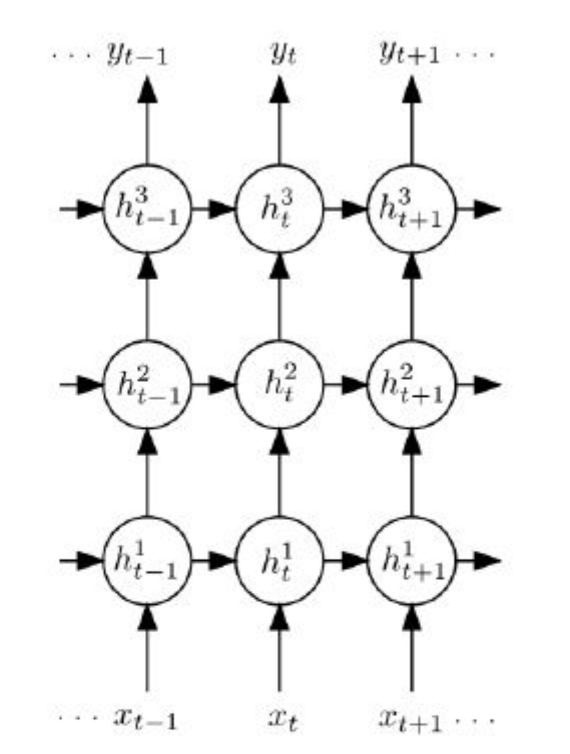
Used for CCG super-tagging [Xu et al, 2015]

Natural use case = language modeling: the sequence of words $x_{1:i}$ is used to predict a distribution over the (i+1)th word [Mikolov 2012, Mikolov et al 2010, Sundermeyer et al. 2016]

- special cases: RNN generator, encoder-decoder, conditioned-generation with attention

Variations: Multi-layer RNN

- multiple layers of RNNs, deep RNN [Hihi and Be
- input of next layer is output of RNN layer below
- Empirically shown to work better



Variation: Bi-directional RNN

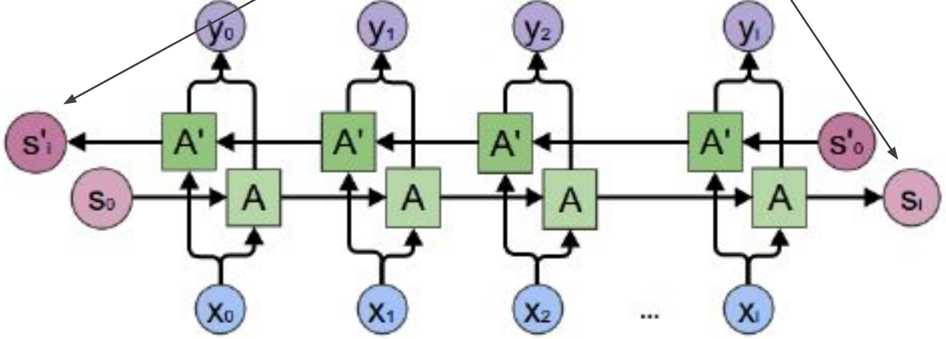
- Input sequence both forward and backward to different RNNs
- Representation is concatenation of forward and backward state
- Represent both history and future
- [Graves, 2008; Schuster and Paliwal, 1997]

2 separate states

- s_i^f (forward, based on x1,x2,...,xi)
- s_i^b (backward, based on xn, xn-1,...,xi)

idea: for sequence tagging, using the history $x_{1:i}$ is useful, but following words $x_{i:n}$ could also be useful

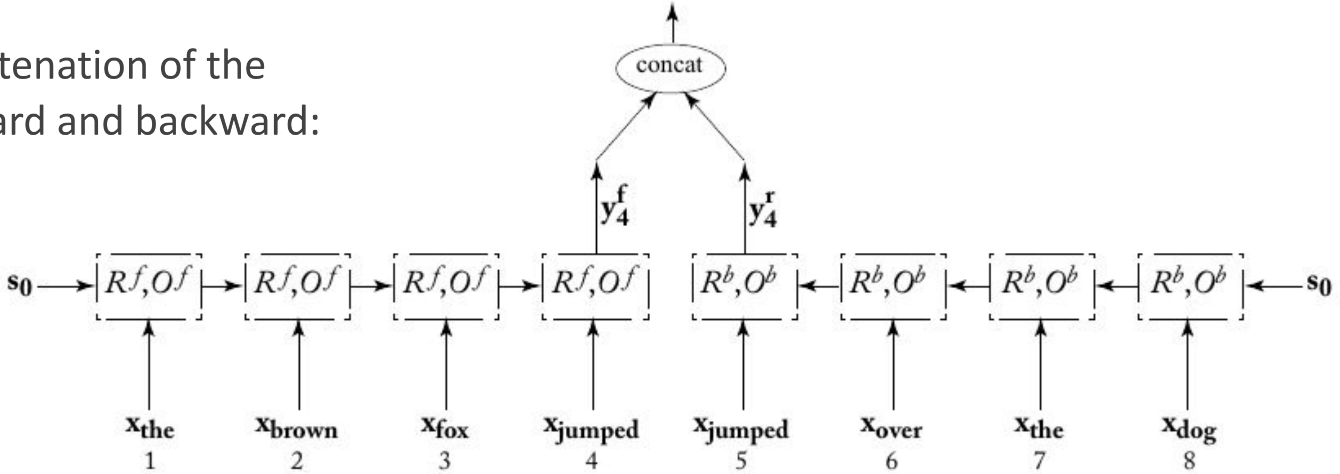
→ similar to a window representing the surrounding words



Bi-RNN

The output y_i is the concatenation of the output of each RNN forward and backward:

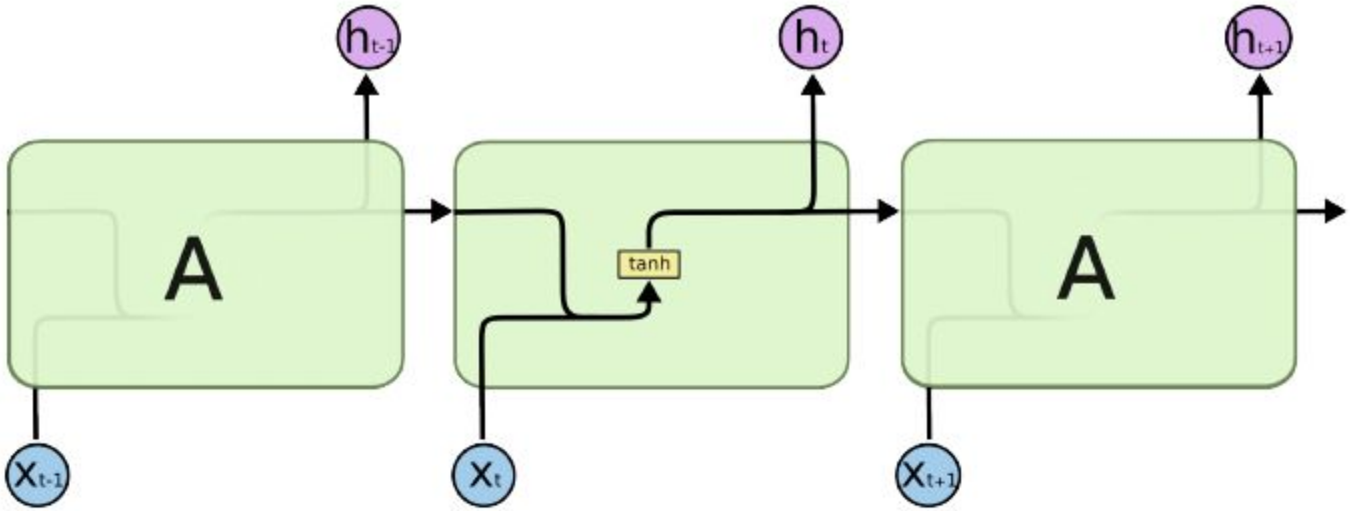
$y_i = [RNN^f(x_{1:i});RNN^b(x_{n:i})]$


concat
$\mathbf{y}_4^\mathbf{f}$
$\mathbf{y}_4^\mathbf{r}$
$\mathbf{s}_0$
R^f,O^f
R^f,O^f
R^f,O^f
R^f,O^f
R^b,O^b
R^b,O^b
R^b,O^b
R^b,O^b
$\mathbf{s}_0$
$\mathbf{x}_{\mathbf{the}}$ 1
$\mathbf{x}_{\mathbf{brown}}$ 2
$\mathbf{x}_{\mathbf{fox}}$ 3
$\mathbf{x}_{\mathbf{jumped}}$ 4
$\mathbf{x}_{\mathbf{jumped}}$ 5
$\mathbf{x}_{\mathbf{over}}$ 6
$\mathbf{x}_{\mathbf{the}}$ 7
$\mathbf{x}_{\mathbf{dog}}$ 8

- very effective for tagging tasks: one output for each input [Irsoy and Cardie 2018]
- useful as a general-purpose trainable feature-extracting component whenever a window around a word is required

Concrete RNN architectures

Concrete architecture: a function $s_i = R(x_i, s_{i-1})$

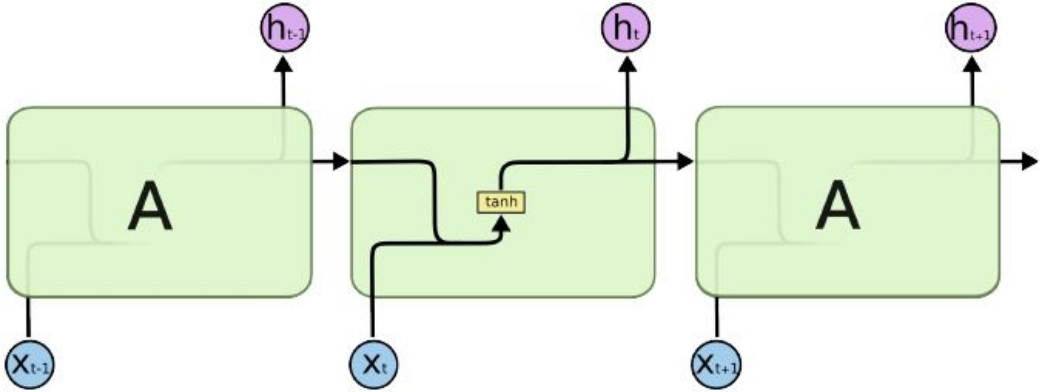


Concrete RNN architectures

Simple-RNN or Elman Network [Elman, 1990; Mikolov, 2012]

$$\begin{aligned} \boldsymbol{s_i} &= R_{\text{SRNN}}(\boldsymbol{x_i}, \boldsymbol{s_{i-1}}) = g(\boldsymbol{s_{i-1}} \boldsymbol{W^s} + \boldsymbol{x_i} \boldsymbol{W^x} + \boldsymbol{b}) \\ \boldsymbol{y_i} &= O_{\text{SRNN}}(\boldsymbol{s_i}) = \boldsymbol{s_i} \end{aligned}$$

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



Concrete RNN architectures

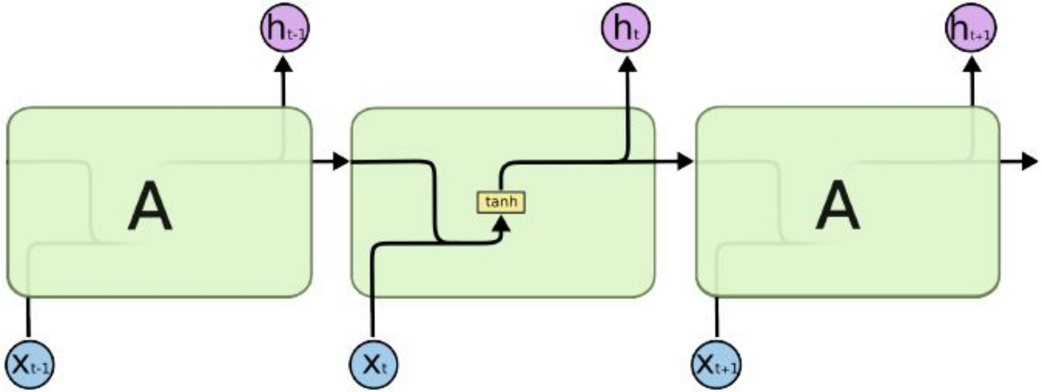
Simple-RNN or Elman Network [Elman, 1990; Mikolov, 2012]

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

$$y_i = O_{\text{SRNN}}(s_i) = s_i$$

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

- a linear transformation of input and prev state
- a non-linear activation (tanh or ReLU)



Concrete RNN architectures

Simple-RNN or Elman Network [Elman, 1990; Mikolov, 2012]

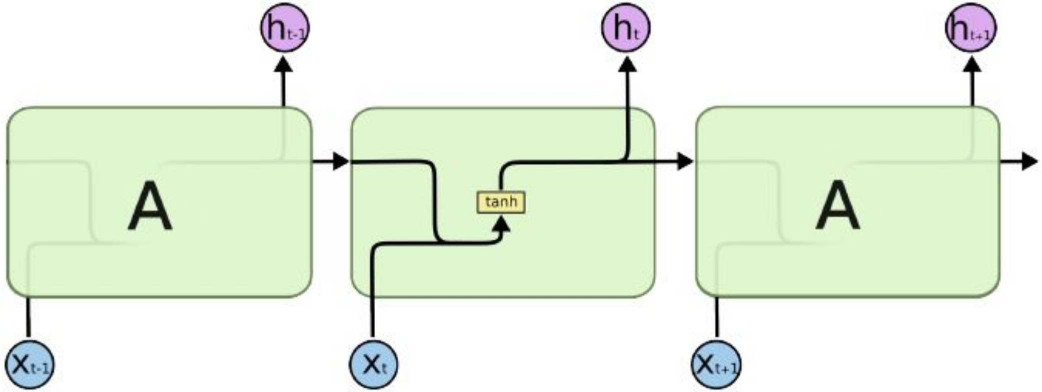
$$\boldsymbol{s_i} = R_{\text{SRNN}}(\boldsymbol{x_i}, \boldsymbol{s_{i-1}}) = g(\boldsymbol{s_{i-1}} \boldsymbol{W^s} + \boldsymbol{x_i} \boldsymbol{W^x} + \boldsymbol{b})$$

$$\boldsymbol{y_i} = O_{\text{SRNN}}(\boldsymbol{s_i}) = \boldsymbol{s_i}$$

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

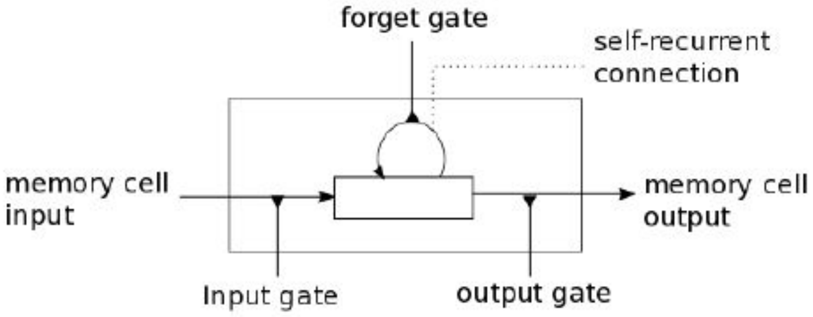
- a linear transformation of input and prev state
- a non-linear activation (tanh or ReLU)

- effective for sequence tagging [Xu et al, 2015] and language modeling
- hard to train: vanishing gradients = hard to capture long-range dependencies



Gating architectures

- Long short term memory networks (LSTM) and Gated Recurrent Unit (GRU)
- In practice, simple RNNs only able to remember narrow context (vanishing gradient)
- LSTM: complex architecture able to capture long-term dependencies through gating mechanisms



Gating architectures

Idea: controlling the memory

e.g. using a binary vector as a gate to decide what to keep or forget $s' \leftarrow g \odot x + (1 - g) \odot (s)$

$$\underset{\mathbf{s'}}{\begin{bmatrix} 8 \\ 11 \\ 3 \\ 7 \\ 5 \\ 15 \end{bmatrix}} \leftarrow \underset{\mathbf{g}}{\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}} \odot \underset{\mathbf{x}}{\begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{bmatrix}} + \underset{\mathbf{(1-g)}}{\begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}} \odot \underset{\mathbf{s}}{\begin{bmatrix} 8 \\ 9 \\ 3 \\ 7 \\ 5 \\ 8 \end{bmatrix}}$$

→ gate vectors are used to control access to the memory state

Gating architectures

We do not use these binary vectors as gates in practice, because we need gates to be:

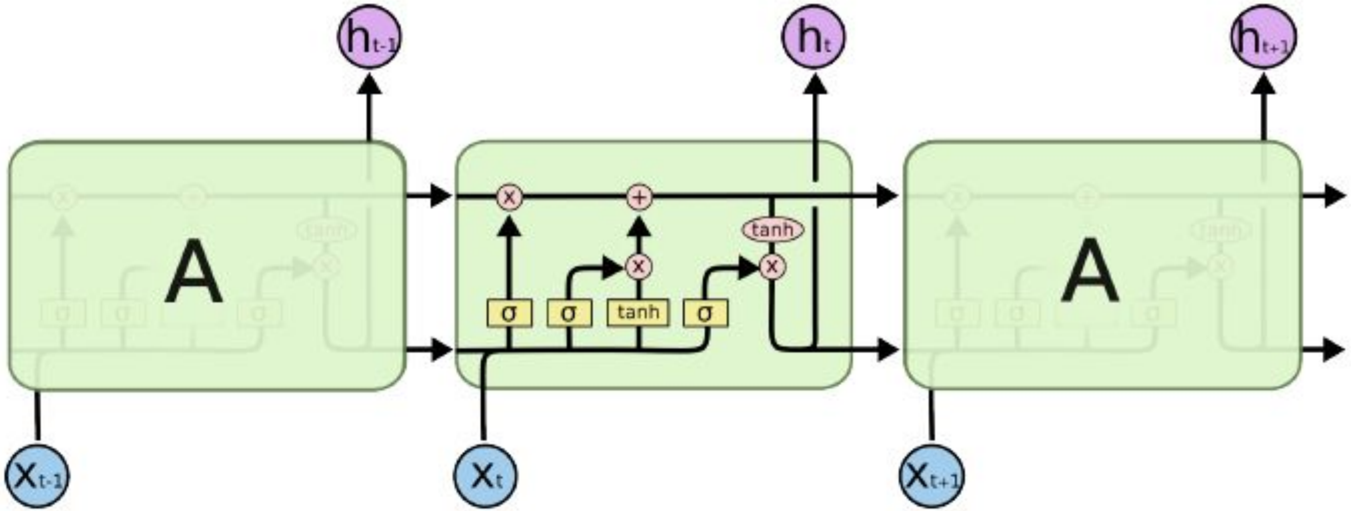
- not static (conditioned on the input and state)
- learned during training
- thus not binary (differentiable)

→ we use vectors of real numbers, then pass through a sigmoid function [0,1]

- values in x corresponding to near one: pass / written to memory
- values in x corresponding to near zero: block / forgotten

LSTM

[Hochreiter and Schmidhuber, 1997]



LSTM

- splits state into 2 parts:
  - one is treated as memory cells: preserve the memory, controlled through gates
  - the other is working memory

$$\boldsymbol{s_j} = R_{\text{LSTM}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) = [\boldsymbol{c_j}; \boldsymbol{h_j}]$$

$$\begin{aligned}
\boldsymbol{c_j} &= \boldsymbol{f} \odot \boldsymbol{c_{j-1}} + \boldsymbol{i} \odot \boldsymbol{z} \\
\boldsymbol{h_j} &= \boldsymbol{o} \odot \tanh(\boldsymbol{c_j}) \\
\boldsymbol{i} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xi}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hi}}) \\
\boldsymbol{f} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xf}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hf}}) \\
\boldsymbol{o} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xo}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{ho}}) \\
\boldsymbol{z} &= \tanh(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hz}})
\end{aligned}$$

$$\boldsymbol{y_j} = O_{\text{LSTM}}(\boldsymbol{s_j}) = \boldsymbol{h_j}$$

$$\boldsymbol{s_j} \in \mathbb{R}^{2 \cdot d_h}, \;\; \boldsymbol{x_i} \in \mathbb{R}^{d_x}, \;\; \boldsymbol{c_j}, \boldsymbol{h_j}, \boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{z} \in \mathbb{R}^{d_h}, \;\; \boldsymbol{W^{x\circ}} \in \mathbb{R}^{d_x \times d_h}, \;\; \boldsymbol{W^{h\circ}} \in \mathbb{R}^{d_h \times d_h}.$$

LSTM

- splits state into 2 parts:
  - one is treated as memory cells: preserve the memory, controlled through gates
  - the other is working memory

$$\boldsymbol{s_j} = R_{\text{LSTM}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) = [\boldsymbol{c_j}; \boldsymbol{h_j}]$$

← s_j split into c_j (memory) and h_j (hidden state)

$$\begin{aligned}
\boldsymbol{c_j} &= \boldsymbol{f} \odot \boldsymbol{c_{j-1}} + \boldsymbol{i} \odot \boldsymbol{z} \\
\boldsymbol{h_j} &= \boldsymbol{o} \odot \tanh(\boldsymbol{c_j}) \\
\boldsymbol{i} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xi}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hi}}) \\
\boldsymbol{f} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xf}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hf}}) \\
\boldsymbol{o} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xo}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{ho}}) \\
\boldsymbol{z} &= \tanh(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hz}})
\end{aligned}$$

$$\boldsymbol{y_j} = O_{\text{LSTM}}(\boldsymbol{s_j}) = \boldsymbol{h_j}$$

$$\boldsymbol{s_j} \in \mathbb{R}^{2 \cdot d_h}, \ \boldsymbol{x_i} \in \mathbb{R}^{d_x}, \ \boldsymbol{c_j}, \boldsymbol{h_j}, \boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{z} \in \mathbb{R}^{d_h}, \ \boldsymbol{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \ \boldsymbol{W^{ho}} \in \mathbb{R}^{d_h \times d_h}.$$

LSTM

- splits state into 2 parts:
  - one is treated as memory cells: preserve the memory, controlled through gates
  - the other is working memory

$$s_j = R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j ; h_j]$$

← s_j split into c_j (memory) and h_j (hidden state)

$$c_j = f \odot c_{j-1} + i \odot z$$

$$h_j = o \odot \tanh(c_j)$$

3 gates:
input
forget
output

$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi})$$

$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf})$$

$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho})$$

$$z = \tanh(x_j W^{xz} + h_{j-1} W^{hz})$$

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

$$s_j \in \mathbb{R}^{2 \cdot d_h}, \quad x_i \in \mathbb{R}^{d_x}, \quad c_j, h_j, i, f, o, z \in \mathbb{R}^{d_h}, \quad W^{xo} \in \mathbb{R}^{d_x \times d_h}, \quad W^{ho} \in \mathbb{R}^{d_h \times d_h}.$$

LSTM

- splits state into 2 parts:
    - one is treated as memory cells: preserve the memory, controlled through gates
    - the other is working memory

$$\boldsymbol{s_j} = R_{\text{LSTM}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) = [\boldsymbol{c_j}; \boldsymbol{h_j}]$$

$$\begin{aligned}
\boldsymbol{c_j} &= \boldsymbol{f} \odot \boldsymbol{c_{j-1}} + \boldsymbol{i} \odot \boldsymbol{z} \\
\boldsymbol{h_j} &= \boldsymbol{o} \odot \tanh(\boldsymbol{c_j}) \\
\boldsymbol{i} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xi}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hi}}) \\
\boldsymbol{f} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xf}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hf}}) \\
\boldsymbol{o} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xo}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{ho}}) \\
\boldsymbol{z} &= \tanh(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hz}})
\end{aligned}$$

Values of the gates:

- linear combination of the input $\mathbf{x}_j$ and the previous state $\mathbf{h}_{j-1}$
- + tanh activation fct

$$\boldsymbol{y_j} = O_{\text{LSTM}}(\boldsymbol{s_j}) = \boldsymbol{h_j}$$

$$\boldsymbol{s_j} \in \mathbb{R}^{2 \cdot d_h}, \ \boldsymbol{x_i} \in \mathbb{R}^{d_x}, \ \boldsymbol{c_j}, \boldsymbol{h_j}, \boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{z} \in \mathbb{R}^{d_h}, \ \boldsymbol{W^{x\circ}} \in \mathbb{R}^{d_x \times d_h}, \ \boldsymbol{W^{h\circ}} \in \mathbb{R}^{d_h \times d_h}.$$

LSTM

- splits state into 2 parts:
  - one is treated as memory cells: preserve the memory, controlled through gates
  - the other is working memory

$$\boldsymbol{s_j} = R_{\text{LSTM}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) = [\boldsymbol{c_j}; \boldsymbol{h_j}]$$

$$\begin{aligned}
\boldsymbol{c_j} &= \boldsymbol{f} \odot \boldsymbol{c_{j-1}} + \boldsymbol{i} \odot \boldsymbol{z} \\
\boldsymbol{h_j} &= \boldsymbol{o} \odot \tanh(\boldsymbol{c_j}) \\
\boldsymbol{i} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xi}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hi}}) \\
\boldsymbol{f} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xf}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hf}}) \\
\boldsymbol{o} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xo}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{ho}}) \\
\boldsymbol{z} &= \tanh(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hz}})
\end{aligned}$$

Values of the gates:
- linear combination of the input **x$_j$** and the previous state **h$_{j-1}$**
- + tanh activation fct

z: update candidate
(linear + non-linear)

$$\boldsymbol{y_j} = O_{\text{LSTM}}(\boldsymbol{s_j}) = \boldsymbol{h_j}$$

$$\boldsymbol{s_j} \in \mathbb{R}^{2 \cdot d_h}, \ \boldsymbol{x_i} \in \mathbb{R}^{d_x}, \ \boldsymbol{c_j}, \boldsymbol{h_j}, \boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{z} \in \mathbb{R}^{d_h}, \ \boldsymbol{W^{x\circ}} \in \mathbb{R}^{d_x \times d_h}, \ \boldsymbol{W^{h\circ}} \in \mathbb{R}^{d_h \times d_h}.$$

LSTM

- splits state into 2 parts:
  - one is treated as memory cells: preserve the memory, controlled through gates
  - the other is working memory

$$\mathbf{s}_j = R_{\text{LSTM}}(\mathbf{s}_{j-1}, \mathbf{x}_j) = [\mathbf{c}_j ; \mathbf{h}_j]$$

1. Update **c**$_{\mathbf{j}}$

$$\mathbf{c}_j = \mathbf{f} \odot \mathbf{c}_{j-1} + \mathbf{i} \odot \mathbf{z}$$

$$\mathbf{h}_j = \mathbf{o} \odot \tanh(\mathbf{c}_j)$$

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

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

$$\mathbf{o} = \sigma(\mathbf{x}_j \mathbf{W}^{xo} + \mathbf{h}_{j-1} \mathbf{W}^{ho})$$

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

Values of the gates:
- linear combination of the input **x**$_{\mathbf{j}}$ and the previous state **h**$_{\mathbf{j-1}}$
- + tanh activation fct

z: update candidate (linear + non-linear)

$$\mathbf{y}_j = O_{\text{LSTM}}(\mathbf{s}_j) = \mathbf{h}_j$$

$$\mathbf{s}_j \in \mathbb{R}^{2 \cdot d_h},\ \mathbf{x}_i \in \mathbb{R}^{d_x},\ \mathbf{c}_j, \mathbf{h}_j, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{z} \in \mathbb{R}^{d_h},\ \mathbf{W}^{x\circ} \in \mathbb{R}^{d_x \times d_h},\ \mathbf{W}^{h\circ} \in \mathbb{R}^{d_h \times d_h}.$$

LSTM

- splits state into 2 parts:
  - one is treated as memory cells: preserve the memory, controlled through gates
  - the other is working memory

$$s_j = R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j ; h_j]$$

what to keep from previous memory

1. Update **c**

$$\begin{aligned}
c_j &= f \odot c_{j-1} + i \odot z \\
h_j &= o \odot \tanh(c_j) \\
i &= \sigma(x_j W^{xi} + h_{j-1} W^{hi}) \\
f &= \sigma(x_j W^{xf} + h_{j-1} W^{hf}) \\
o &= \sigma(x_j W^{xo} + h_{j-1} W^{ho}) \\
z &= \tanh(x_j W^{xz} + h_{j-1} W^{hz})
\end{aligned}$$

Values of the gates:

- linear combination of the input $\mathbf{x}_j$ and the previous state $\mathbf{h}_{j-1}$
- + tanh activation fct

z: update candidate (linear + non-linear)

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

$$s_j \in \mathbb{R}^{2 \cdot d_h}, \ x_i \in \mathbb{R}^{d_x}, \ c_j, h_j, i, f, o, z \in \mathbb{R}^{d_h}, \ W^{xo} \in \mathbb{R}^{d_x \times d_h}, \ W^{ho} \in \mathbb{R}^{d_h \times d_h}.$$

LSTM

- splits state into 2 parts:
  - one is treated as memory cells: preserve the memory, controlled through gates
  - the other is working memory

$$s_j = R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j; h_j]$$

1. Update $\mathbf{c_j}$

$$c_j = f \odot c_{j-1} + i \odot z$$

what to keep from previous memory

what to keep from the proposed update

$$h_j = o \odot \tanh(c_j)$$

$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi})$$

$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf})$$

$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho})$$

$$z = \tanh(x_j W^{xz} + h_{j-1} W^{hz})$$

Values of the gates:
- linear combination of the input $\mathbf{x_j}$ and the previous state $\mathbf{h_{j-1}}$
- + tanh activation fct

z: update candidate (linear + non-linear)

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

$$s_j \in \mathbb{R}^{2 \cdot d_h},\ x_i \in \mathbb{R}^{d_x},\ c_j, h_j, i, f, o, z \in \mathbb{R}^{d_h},\ W^{xo} \in \mathbb{R}^{d_x \times d_h},\ W^{ho} \in \mathbb{R}^{d_h \times d_h}.$$

LSTM

- splits state into 2 parts:
  - one is treated as memory cells: preserve the memory, controlled through gates
  - the other is working memory

$$s_j = R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j ; h_j]$$

1. Update **c$_j$**
2. Compute **h$_j$**

$$\begin{aligned}
c_j &= f \odot c_{j-1} + i \odot z \\
h_j &= o \odot \tanh(c_j) \\
i &= \sigma(x_j W^{xi} + h_{j-1} W^{hi}) \\
f &= \sigma(x_j W^{xf} + h_{j-1} W^{hf}) \\
o &= \sigma(x_j W^{xo} + h_{j-1} W^{ho}) \\
z &= \tanh(x_j W^{xz} + h_{j-1} W^{hz})
\end{aligned}$$

Based on the memory **cj** (+ non-linearity) and output gate

Values of the gates:
- linear combination of the input **x$_j$** and the previous state **h$_{j-1}$**
- + tanh activation fct

z: update candidate (linear + non-linear)

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

$$s_j \in \mathbb{R}^{2\cdot d_h},\ x_i \in \mathbb{R}^{d_x},\ c_j, h_j, i, f, o, z \in \mathbb{R}^{d_h},\ W^{xo} \in \mathbb{R}^{d_x \times d_h},\ W^{ho} \in \mathbb{R}^{d_h \times d_h}.$$

LSTM

- splits state into 2 parts:
  - one is treated as memory cells: preserve the memory, controlled through gates
  - the other is working memory

$$s_j = R_{\text{LSTM}}(s_{j-1}, x_j) = [c_j ; h_j]$$

1. Update **c_j**
2. Compute **h_j**
3. **y_j = h_j**

$$c_j = f \odot c_{j-1} + i \odot z$$

$$h_j = o \odot \tanh(c_j)$$

$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi})$$

$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf})$$

$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho})$$

$$z = \tanh(x_j W^{xz} + h_{j-1} W^{hz})$$

Based on the memory **cj** (+ non-linearity) and output gate

Values of the gates:
- linear combination of the input **x_j** and the previous state **h_{j-1}**
- + tanh activation fct

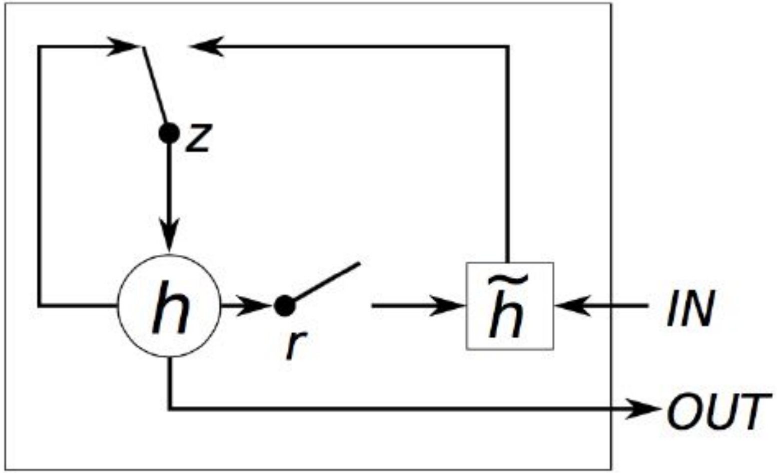
z: update candidate (linear + non-linear)

$$y_j = O_{\text{LSTM}}(s_j) = h_j$$

$$s_j \in \mathbb{R}^{2 \cdot d_h}, \; x_i \in \mathbb{R}^{d_x}, \; c_j, h_j, i, f, o, z \in \mathbb{R}^{d_h}, \; W^{xo} \in \mathbb{R}^{d_x \times d_h}, \; W^{ho} \in \mathbb{R}^{d_h \times d_h}.$$

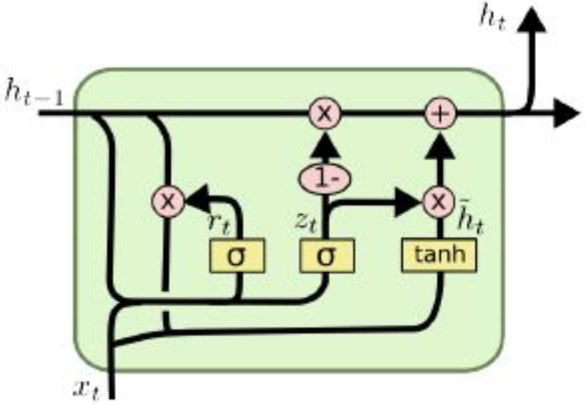
GRU

- LSTM: effective, but complex, computationally expensive
- GRU: cheaper alternative that works well in practice [Cho et al. 2014; Chung et al. 2014]



GRU

- reset gate (**r**): how much information from previous hidden state needs to be included (reset with current information?)
- upgate gate (**z**): controls updates to hidden state (how much does hidden state need to be updated with current information?)
- **y_j = s_j**



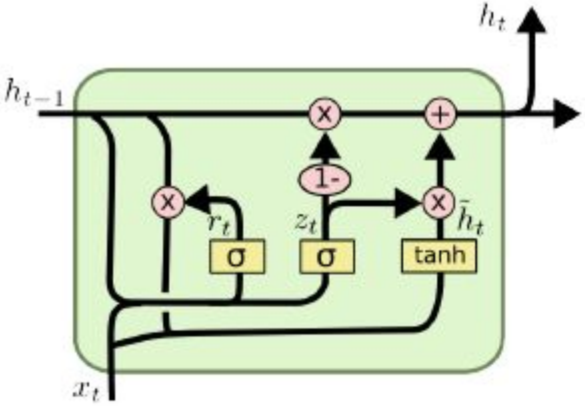
$$
\begin{aligned}
\boldsymbol{s_j} = R_{\mathrm{GRU}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) &= (\mathbf{1} - \boldsymbol{z}) \odot \boldsymbol{s_{j-1}} + \boldsymbol{z} \odot \tilde{\boldsymbol{s_j}} \\
\boldsymbol{z} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{s_{j-1}} \boldsymbol{W^{sz}}) \\
\boldsymbol{r} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xr}} + \boldsymbol{s_{j-1}} \boldsymbol{W^{sr}}) \\
\tilde{\boldsymbol{s_j}} &= \tanh(\boldsymbol{x_j} \boldsymbol{W^{xs}} + (\boldsymbol{r} \odot \boldsymbol{s_{j-1}}) \boldsymbol{W^{sg}})
\end{aligned}
$$

$$\boldsymbol{y_j} = O_{\mathrm{GRU}}(\boldsymbol{s_j}) = \boldsymbol{s_j}$$

$$\boldsymbol{s_j}, \tilde{\boldsymbol{s_j}} \in \mathbb{R}^{d_s},\ \boldsymbol{x_i} \in \mathbb{R}^{d_x},\ \boldsymbol{z}, \boldsymbol{r} \in \mathbb{R}^{d_s},\ \boldsymbol{W^{x\circ}} \in \mathbb{R}^{d_x \times d_s},\ \boldsymbol{W^{s\circ}} \in \mathbb{R}^{d_s \times d_s}.$$

GRU

- reset gate (**r**): how much information from previous hidden state needs to be included (reset with current information?)
- upgate gate (**z**): controls updates to hidden state (how much does hidden state need to be updated with current information?)
- **$y_j = s_j$**



state is not split

$$s_j = R_{\text{GRU}}(s_{j-1}, x_j) = (1 - z) \odot s_{j-1} + z \odot \tilde{s_j}$$

2 gates

$$z = \sigma(x_j W^{xz} + s_{j-1} W^{sz})$$

$$r = \sigma(x_j W^{xr} + s_{j-1} W^{sr})$$

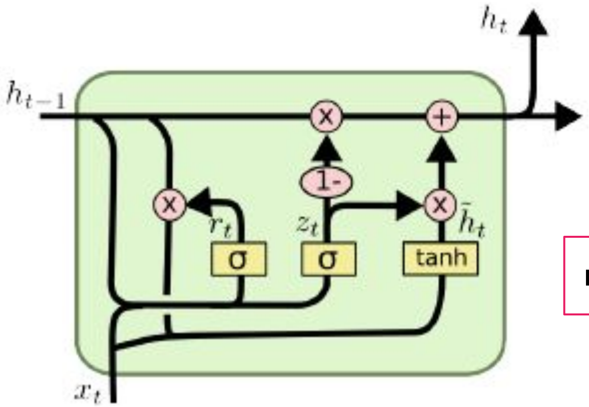
$$\tilde{s_j} = \tanh(x_j W^{xs} + (r \odot s_{j-1}) W^{sg})$$

$$y_j = O_{\text{GRU}}(s_j) = s_j$$

$$s_j, \tilde{s_j} \in \mathbb{R}^{d_s}, \ x_i \in \mathbb{R}^{d_x}, \ z, r \in \mathbb{R}^{d_s}, \ W^{x\circ} \in \mathbb{R}^{d_x \times d_s}, \ W^{s\circ} \in \mathbb{R}^{d_s \times d_s}.$$

GRU

- reset gate (r): how much information from previous hidden state needs to be included (reset with current information?)
- upgate gate (z): controls updates to hidden state (how much does hidden state need to be updated with current information?)
- **$y_j = s_j$**



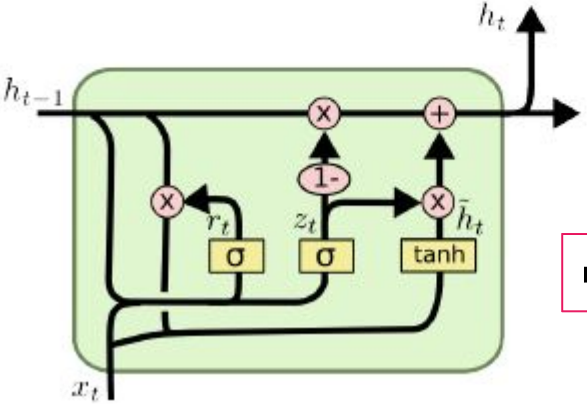
$$
\begin{aligned}
\boldsymbol{s_j} = R_{\mathrm{GRU}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) &= (\boldsymbol{1} - \boldsymbol{z}) \odot \boldsymbol{s_{j-1}} + \boldsymbol{z} \odot \tilde{\boldsymbol{s_j}} \\
\boldsymbol{z} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{s_{j-1}} \boldsymbol{W^{sz}}) \\
\boldsymbol{r} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xr}} + \boldsymbol{s_{j-1}} \boldsymbol{W^{sr}}) \\
\tilde{\boldsymbol{s_j}} &= \tanh(\boldsymbol{x_j} \boldsymbol{W^{xs}} + (\boldsymbol{r} \odot \boldsymbol{s_{j-1}}) \boldsymbol{W^{sg}})
\end{aligned}
$$

`r control access to the previous state and compute update`

$$\boldsymbol{s_j}, \tilde{\boldsymbol{s_j}} \in \mathbb{R}^{d_s}, \ \boldsymbol{x_i} \in \mathbb{R}^{d_x}, \ \boldsymbol{z}, \boldsymbol{r} \in \mathbb{R}^{d_s}, \ \boldsymbol{W^{x\circ}} \in \mathbb{R}^{d_x \times d_s}, \ \boldsymbol{W^{s\circ}} \in \mathbb{R}^{d_s \times d_s}.$$

GRU

- reset gate (r): how much information from previous hidden state needs to be included (reset with current information?)
- upgate gate (z): controls updates to hidden state (how much does hidden state need to be updated with current information?)
- **$y_j = s_j$**



```
new state = interpolation of
previous state and update,
controlled by z
```

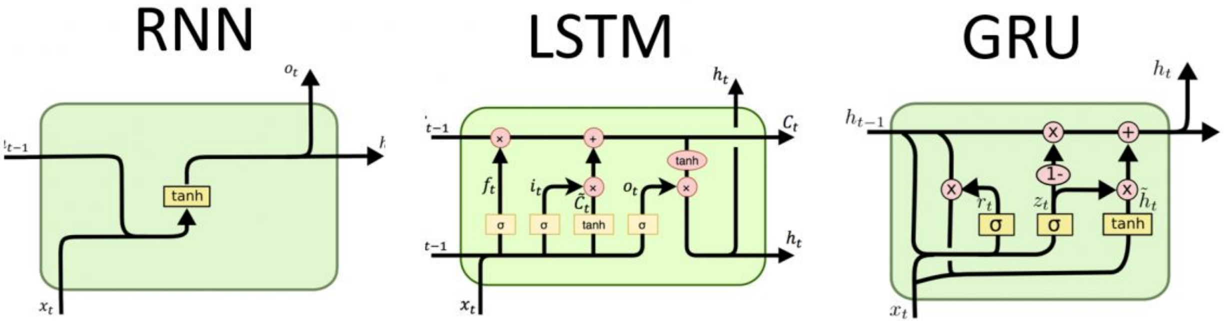
$$
\begin{aligned}
\boldsymbol{s_j} &= R_{\text{GRU}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) = (\boldsymbol{1} - \boldsymbol{z}) \odot \boldsymbol{s_{j-1}} + \boldsymbol{z} \odot \tilde{\boldsymbol{s_j}} \\
\boldsymbol{z} &= \sigma(\boldsymbol{x_j} \boldsymbol{W}^{\boldsymbol{xz}} + \boldsymbol{s_{j-1}} \boldsymbol{W}^{\boldsymbol{sz}}) \\
\boldsymbol{r} &= \sigma(\boldsymbol{x_j} \boldsymbol{W}^{\boldsymbol{xr}} + \boldsymbol{s_{j-1}} \boldsymbol{W}^{\boldsymbol{sr}}) \\
\tilde{\boldsymbol{s_j}} &= \tanh(\boldsymbol{x_j} \boldsymbol{W}^{\boldsymbol{xs}} + (\boldsymbol{r} \odot \boldsymbol{s_{j-1}}) \boldsymbol{W}^{\boldsymbol{sg}})
\end{aligned}
$$

```
r control access to the previous state and compute update
```

$$\boldsymbol{s_j}, \tilde{\boldsymbol{s_j}} \in \mathbb{R}^{d_s}, \ \boldsymbol{x_i} \in \mathbb{R}^{d_x}, \ \boldsymbol{z}, \boldsymbol{r} \in \mathbb{R}^{d_s}, \ \boldsymbol{W}^{\boldsymbol{x\circ}} \in \mathbb{R}^{d_x \times d_s}, \ \boldsymbol{W}^{\boldsymbol{s\circ}} \in \mathbb{R}^{d_s \times d_s}.$$

RNNs

- GRU effective for language modeling and machine translation, but still no consensus on which architecture is the best [Jozefowicz et al. 2015]
- Variants with non gated architecture [Mikolov et al 2014; Le et al, 2015]
- Initialisation: bias and forget gate close to 1 for LSTM [Jozefowicz et al 2015]
- Drop-out: only on the non-recurrent connection (in between layers in deep-RNNs) or variational RNN dropout method = same dropout across time steps [Gal, 2015]

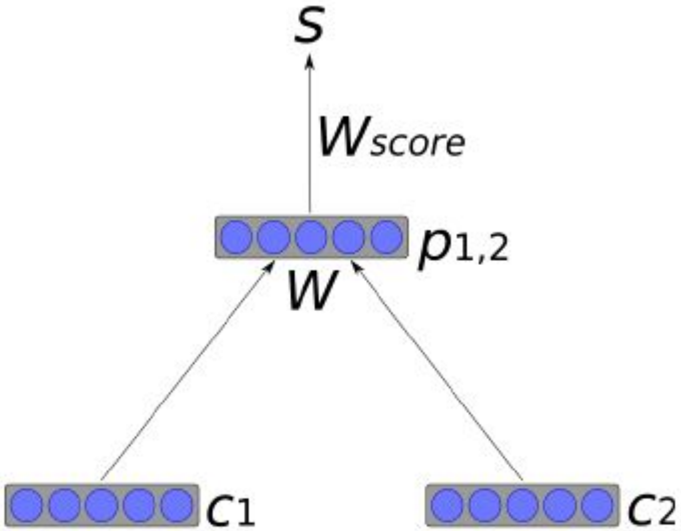


Recursive Neural Networks

- Generalization of RNNs from sequences to (binary) trees

- Linear transformation + non-linear activation function applied recursively throughout a tree

- Useful for parsing



- https://www.wikiwand.com/fr/R%C3%A9seau_neuronal_convolutif
- Nice intro: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/
- More details: https://cs231n.github.io/convolutional-networks/
- (picture, bad explanation) https://datascientest.com/convolutional-neural-network
- https://www.quora.com/What-are-recurrent-neural-network-algorithms
- https://medium.com/analytics-vidhya/recurrent-neural-network-and-its-variants-de75f9ee063
- https://machinelearningmastery.com/an-introduction-to-recurrent-neural-networks-and-the-math-that-powers-them/
- https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks
- https://karpathy.github.io/2015/05/21/rnn-effectiveness/
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html
- https://github.com/pytorch/examples/tree/master/word_language_model
- https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html
- https://www.pluralsight.com/guides/natural-language-processing-with-pytorch
- https://stats.stackexchange.com/questions/335321/in-a-convolutional-neural-network-cnn-when-convolving-the-image-is-the-opera
-