Neural Networks in Computer Vision and Natural Language Processing

Stefano Melacci

Department of Information Enginnering

University of Siena

Disclaimer

The contents (and the style) of these slides are taken from Stefano Melacci's slides of the *Machine Learning & Deep Learning* course - Datum Academy (France)

They are used with the sole purpose of supporting Stefano Melacci's teaching activity.

They are not intended to be of public domain in any way, and they must not be published or shared out of the context in which they are presented by Stefano Melacci.

Computer Vision & Natural Language

- This module focuses on applications of deep networks to Computer Vision and Language Modeling, exploiting Convolutional Neural Networks and Recurrent Neural Networks, respectively

- **Convolutional Neural Networks**
  - Neural Networks that implement multi-layer convolutions
  - *Applications to several problems*
  - We will study the case of **Computer Vision (Image Classification)**

- **Recurrent Neural Networks**
  - Neural Networks that can process sequences
  - *Applications to several problems*
  - We will study the case of **Natural Language (Language Modeling)**

Computer Vision & Natural Language (2)

- Overview
  1. **Computer Vision, Image Classification & Neural Networks**
     - Classifying Images with Neural Networks, MLPs
  2. **Convolution & Images**
     - Convolution, Blurring, Receptive Fields, Feature Maps
  3. **Convolutional Neural Networks**
     - From MLPs to Convolutional Networks, Pooling, Stride
  4. **Image Classification with Convolutional Neural Networks**
     - Learning Convolutional Networks in Image Classification Tasks, Properties
  5. **Natural Language Modeling & Neural Networks**
     - Natural Language, Sequences & Neural Networks
  6. **Recurrent Neural Networks**
     - Recurrent Nets, Processing Sequences, Training, Long-term Dependencies
  7. **Language Modeling with Recurrent Neural Networks**
     - Exploiting RNNs in Language Modeling, Representing Words
  8. **Long-Short Term Memories**
     - LSTM Networks

Natural Language Modeling & Neural Networks

Stefano Melacci

Natural Language Processing

- Natural Language Processing (NLP) is a field that includes a large variety of topics which involve processing and understanding **human language**
- There exist a large number of applications in the context of NLP
  - Text Classification
  - Machine Translation
  - Question Answering
  - Sentiment Analysis
  - Text Paraphrasing
  - Summarization
  - Conversational Systems
  - …
- Nowadays it is extremely common to use *Machine Learning* in NLP
  - Neural Networks

Natural Language Processing (2)

- Human language is ambiguous, it involves high-level concepts, it is strongly context dependent
- Machines perform well-defined operations, they are good for implementing algorithms
  - However, we cannot simply write down the rules that define how we (as humans) communicate!
- Programming a machine that **understands** human language is definitely not a simple task
  - Moreover, what is "*understanding*"? What is the right formalization of the concept of "*understanding*" that is well suited for a machine to handle natural language?

Language Modeling

- Understanding whether a given sentence belongs to a target language
  - Well-posed problem in the case of formal languages
  - Example: programming languages
  - For the reasons we mentioned in the previous, it is not so easy to follow rule-based approaches
- (*Probabilistic*) **Language Modeling** (LM): developing a probabilistic model that estimates the probability of observing a certain sequence of words

$$P(w_1, \ldots, w_n) = ?$$

- *Goal: learn a LM from data!*

Language Modeling (2)

- We can model the probability of a certain sequence of words in function of **the probability of observing a word given the words that precede it**

$$P(w_i|w_{i-1}, w_{i-2}, \ldots, w_1)$$

- In particular we can recover the probability of the whole sentence as follows

$$P(w_1, \ldots, w_n) = \prod_{i=1}^{n} P(w_i|w_{i-1}, w_{i-2}, \ldots, w_1)$$

Language Modeling (3)

- We can read the last equation as

$$P(w_1, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | \texttt{context})$$

- If the context is limited to *N − 1* words, we can simply estimate $P(w_i \mid context)$ statistically
  - Given a text corpus, store all the sequences of length *N*, find the ones for which the first *N-1* words correspond to the considered context
  - Then, count the relative frequency of w_i being the N-th word
  - The same sequence can be repeated multiple times in the corpus
  - How do we select *N?*
  - Can we store all the possible sequences of length *N?*
  - Can we handle variable-length contexts?

Language Modeling: Machine Learning

- Machine Learning can be used to learn a LM
- Language has regularities, Machine Learning can be used to learn these regularities from real data, in order generate a more compact and efficient LM
- Once we are given a large collection of text, we aim at building a Neural Network that is able to compute:

$$P(w_i|w_{i-1}, w_{i-2}, \ldots, w_1)$$

- The context that precedes the word we aim at predicting a variable length **sequence**

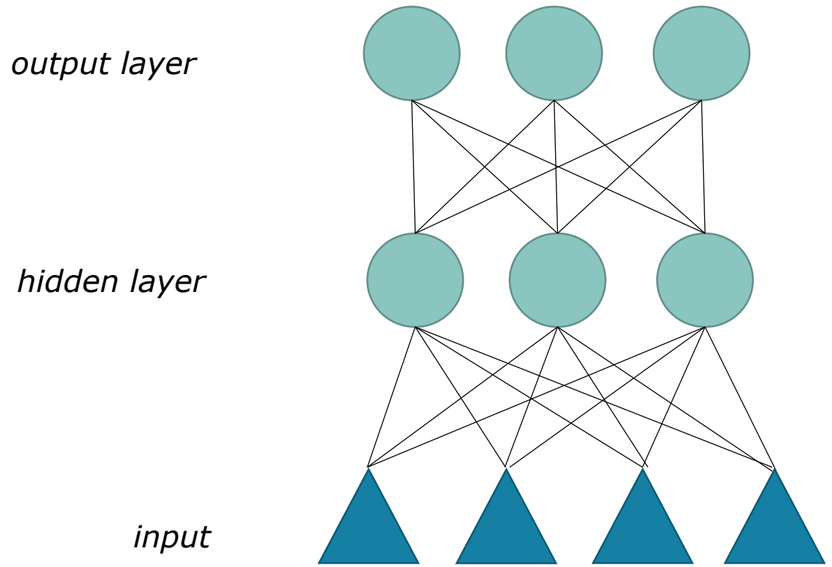
Sequences

- When dealing with natural language, we always have to deal with variable-length sequences
  - Either as *input* or *output* signal
  - *Order* does matter!

- Human-to-human conversation
  - Input: **sequence** of ”sounds”
  - Output: **sequence** of “sounds”
- Reading
  - Input: **sequence** of characters, words
- Writing
  - Output: **sequence** of characters, words
- …and Language Modeling, of course

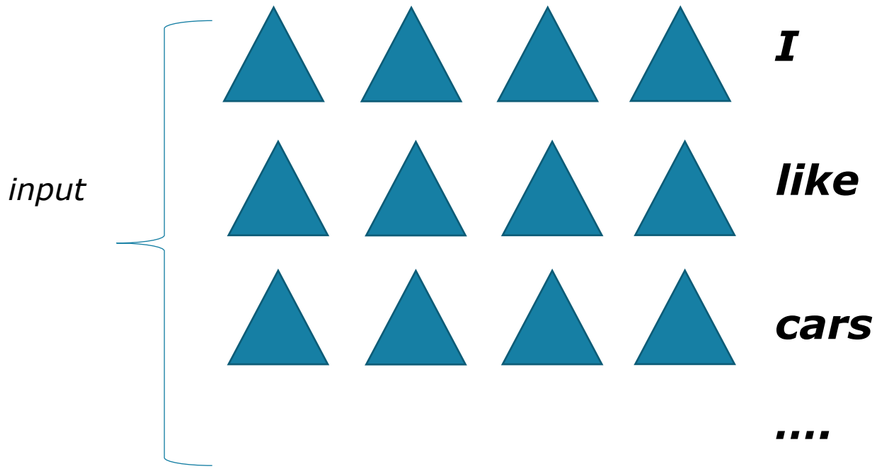
Sequences and Neural Networks

- Multi Layer Perceptrons are designed to process a **fixed-length** input signal
  - How can we adapt them to language?



Sequences and Neural Networks (2)

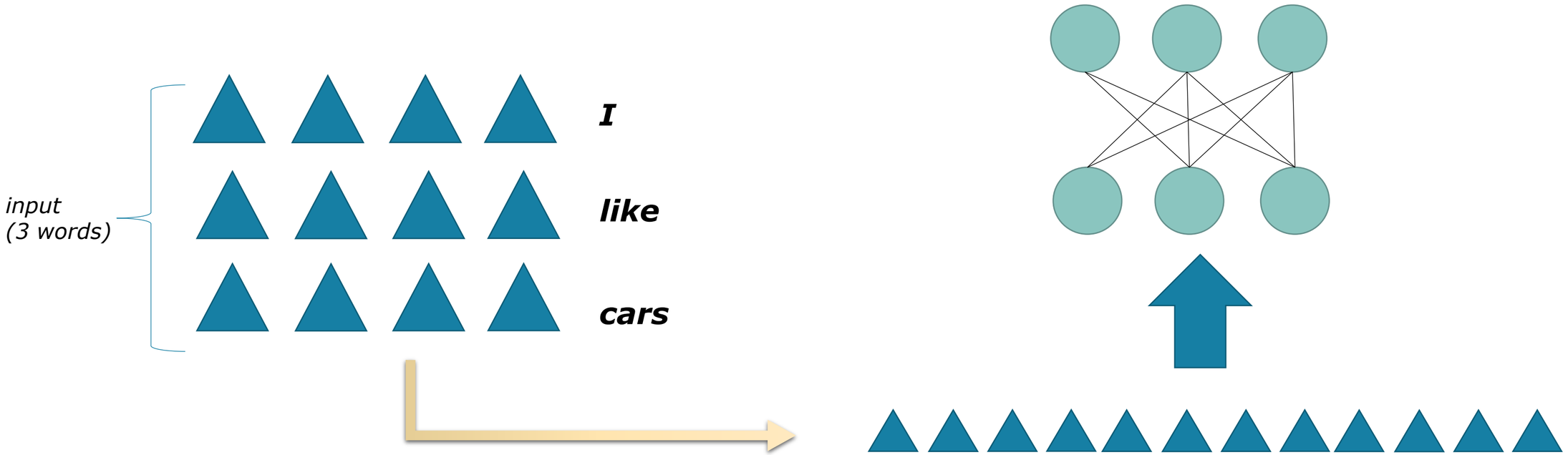
- Let's assume that we are processing language at **word-level**
  - The language is divided into **atomic units** (*tokens*) that correspond to the words
  - Of course this is not the only option, for example, one might also word at character-level
- Each word is an input element, and it is represented with a vector
  - For the moment just assume that for each word you have a vector that represents it
  - Let's postpone the details about the word-vectors



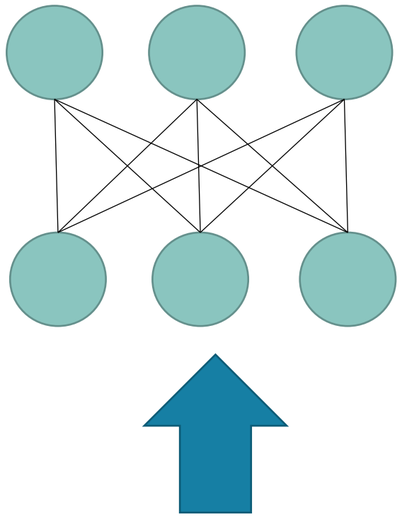
We cannot use an MLP!

Sequences and Neural Networks (3)

- At a first glance, in order to use an MLP we could concatenate a **fixed number of word vectors**
  - We would end-up with a vectorial input and we can plug it into an MLP, as usual



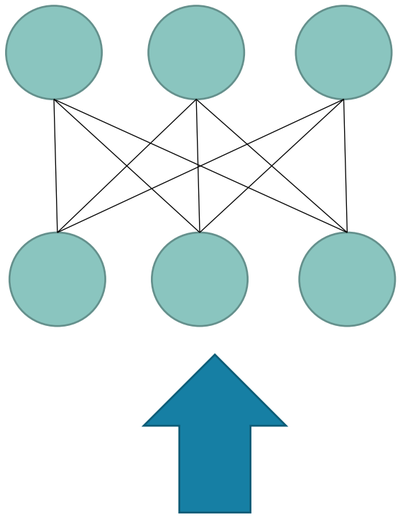
Sequences and Neural Networks (4)



Attempt: divide the sentence into *small* sub-portions of fixed-size, and process them (sliding window)

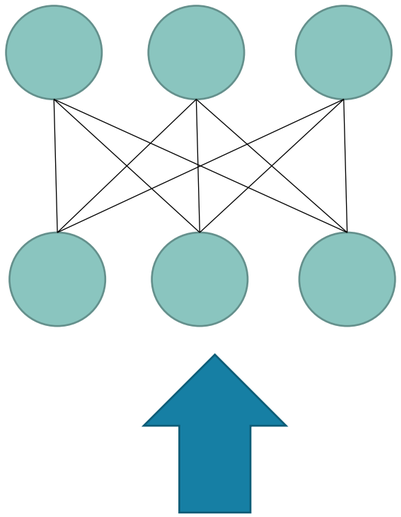
I really would like to eat a pizza during the next week, even if …

Sequences and Neural Networks (4)



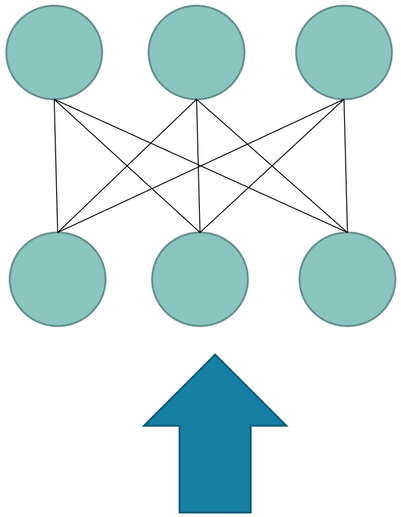
I really would like to eat a pizza during the next week, even if …

Sequences and Neural Networks (4)



I really would like to eat a pizza during the next week, even if …

Sequences and Neural Networks (4)

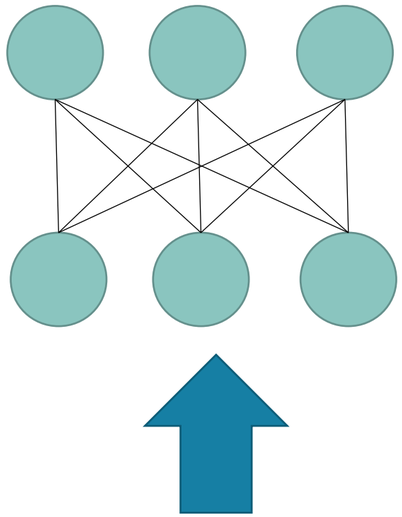


At each time instant, we take a decision that is **independent** from what we processed earlier

The network has never access to a comprehensive view of the whole sentence

I really would like to eat a pizza during the next week, even if …

Sequences and Neural Networks (5)

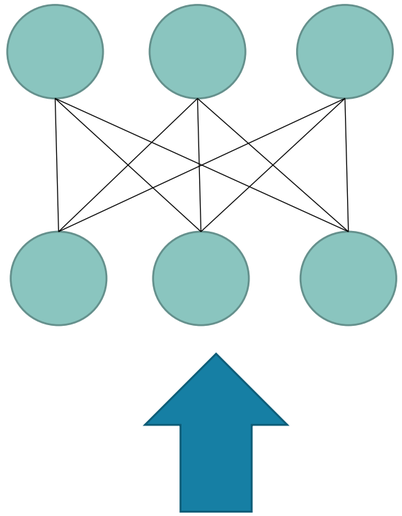


Attempt: use a single, very large window! In other words, we could process sentences of fixed size (*large*), adding dummy words in order to pad the shorter sentences

I really would like to eat a pizza during the next week

12 words

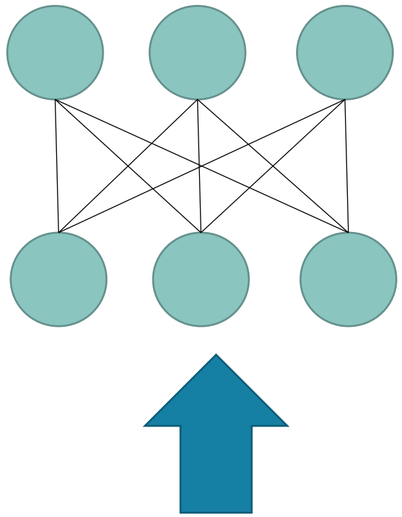
Sequences and Neural Networks (5)



I think that this time I will win the match against Mario

12 words

Sequences and Neural Networks (5)

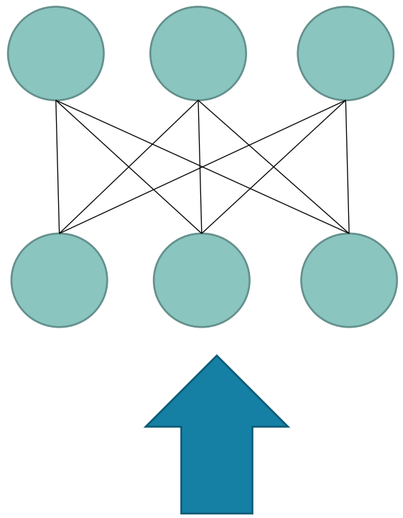


What is the right max length of the input?
Moreover, we are not efficiently exploiting any compositionality of language: if the same sequence of words appear in different positions in different sentences, words will be processed by different connections

Hello world DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY

12 words
(10 dummy words)

Sequences and Neural Networks (6)

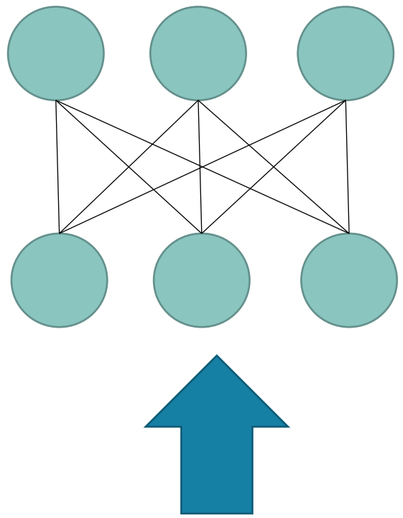


Attempt: only keep word level statistics, the so called bag-of-words
We define a vocabulary of words, fixed-size, and then we count how many times each word appears in the input sentence.
Then we generate a vector with all the counts.

fun:0, really:1, bed:0 , eat:1, random:0 , would:1, …, pizza:**2**

INPUT: I really would like to eat a pizza because I love pizza

Sequences and Neural Networks (6)



This is an OK approach (frequently used in document classification) but **we are loosing all the positional information** (the order in which words appear)

fun:0, really:1, bed:0 , eat:1, random:0 , would:1, …, pizza:**2**

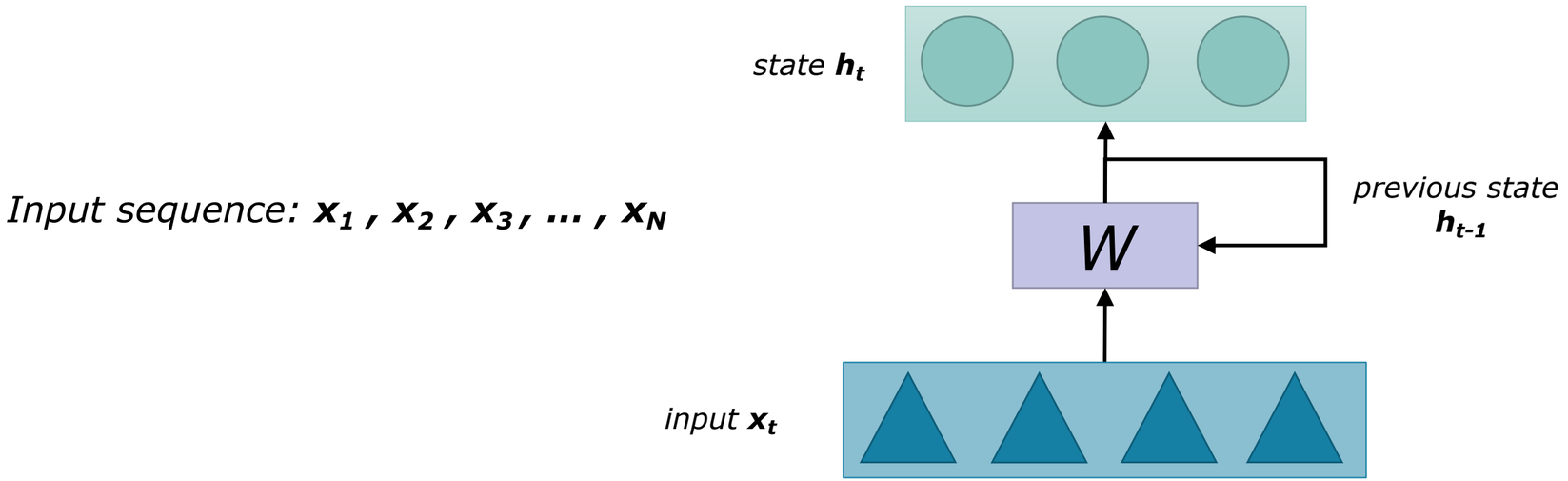
INPUT: I really would like to eat a pizza because I love pizza

Recurrent Neural Networks

Stefano Melacci

Recurrent Neural Networks

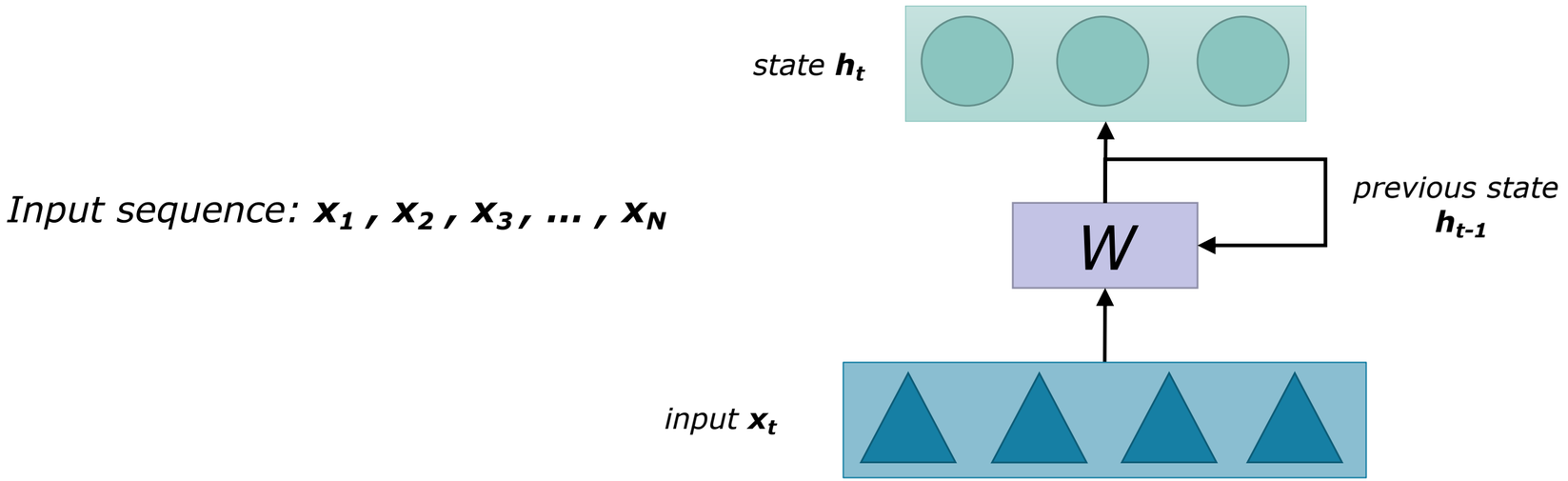
- Recurrent Neural Networks (RNNs) are neural architectures that include a **local feedback**, and that can be used to process variable-length sequences
- RNNs *sequentially* process each element of the input sequence
  - RNNs compute and update a *fixed-length internal representation*, called (hidden) **state**
  - *W* is the set of weights of the net



Recurrent Neural Networks (2)

- Given an input sequence x_1 , x_2 , x_3 , … , x_N the RNN computes the following function *f* at each time instant *t*

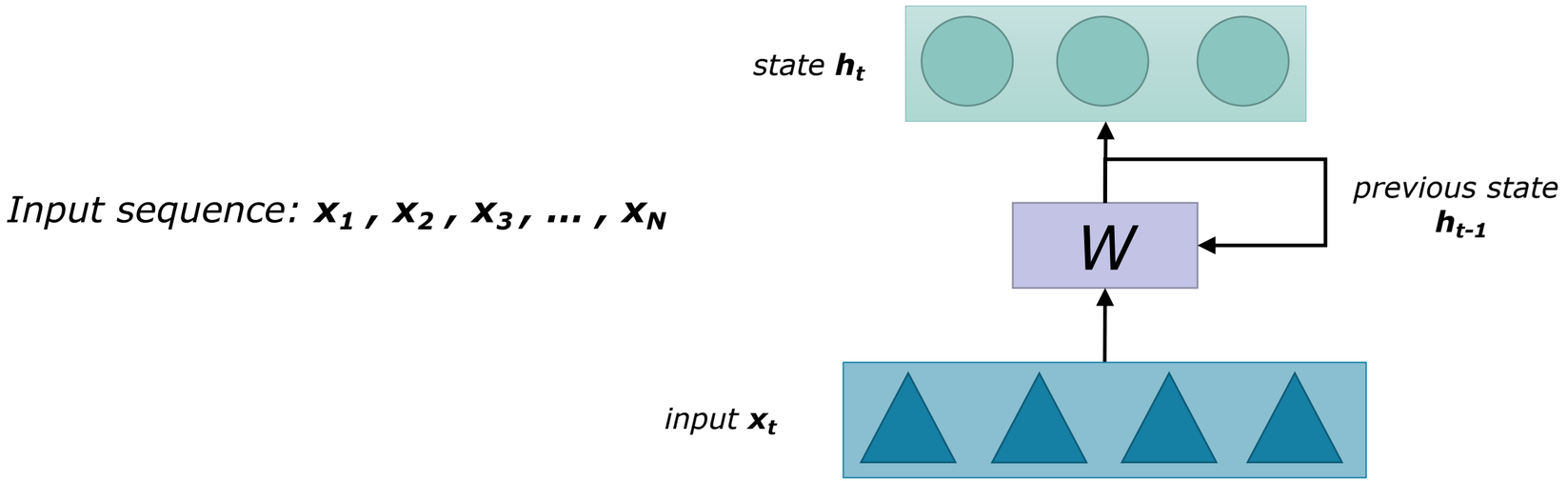
$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$$



Recurrent Neural Networks (3)

- The internal state is updated due to the current input element x_t, and the previous state of the network h_{t-1}

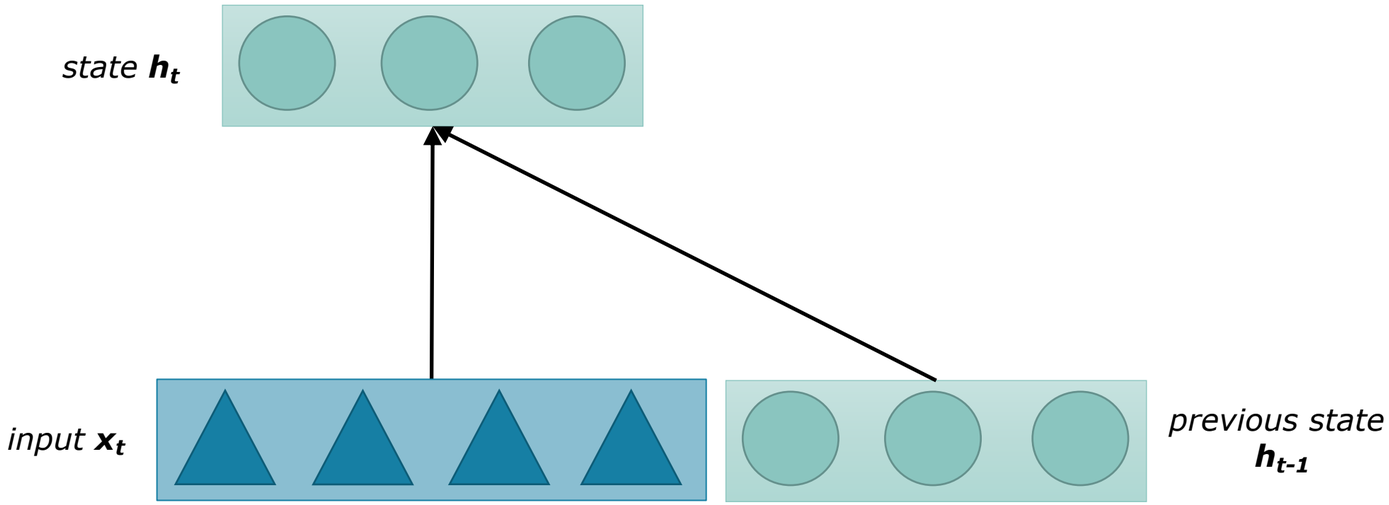
$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$$



Recurrent Neural Networks – Projection

- In detail, the network computes a linear projection of the concatenated representation $[x_t , h_{t-1}]$, followed by a nonlinear activation (tanh, sigmoid, ReLu, ...)

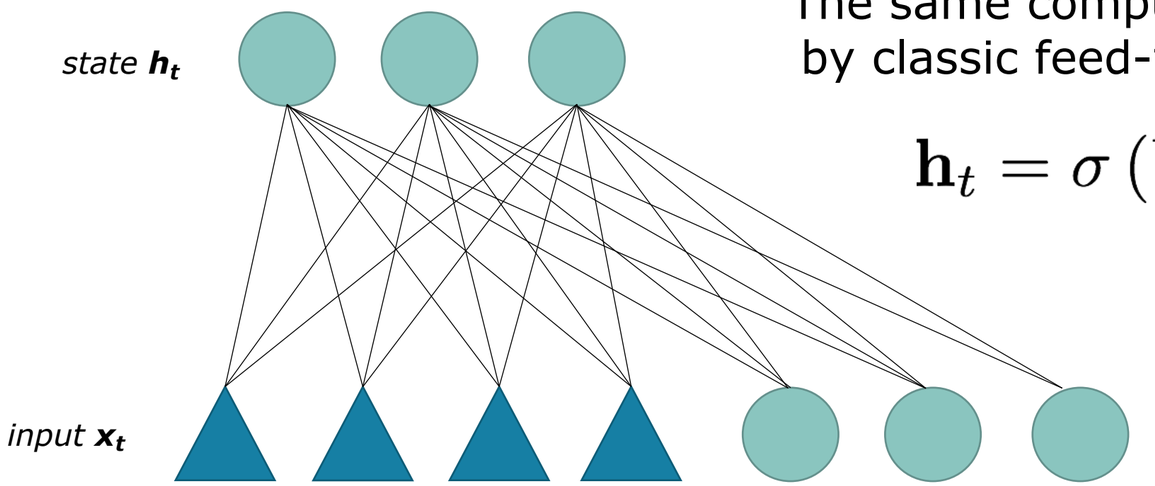
$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$$



Recurrent Neural Networks – Projection (2)

- In detail, the network computes a linear projection of the concatenated representation *[x_t , h_{t-1}]*, followed by a nonlinear activation (tanh, sigmoid, ReLu, ...)

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$$



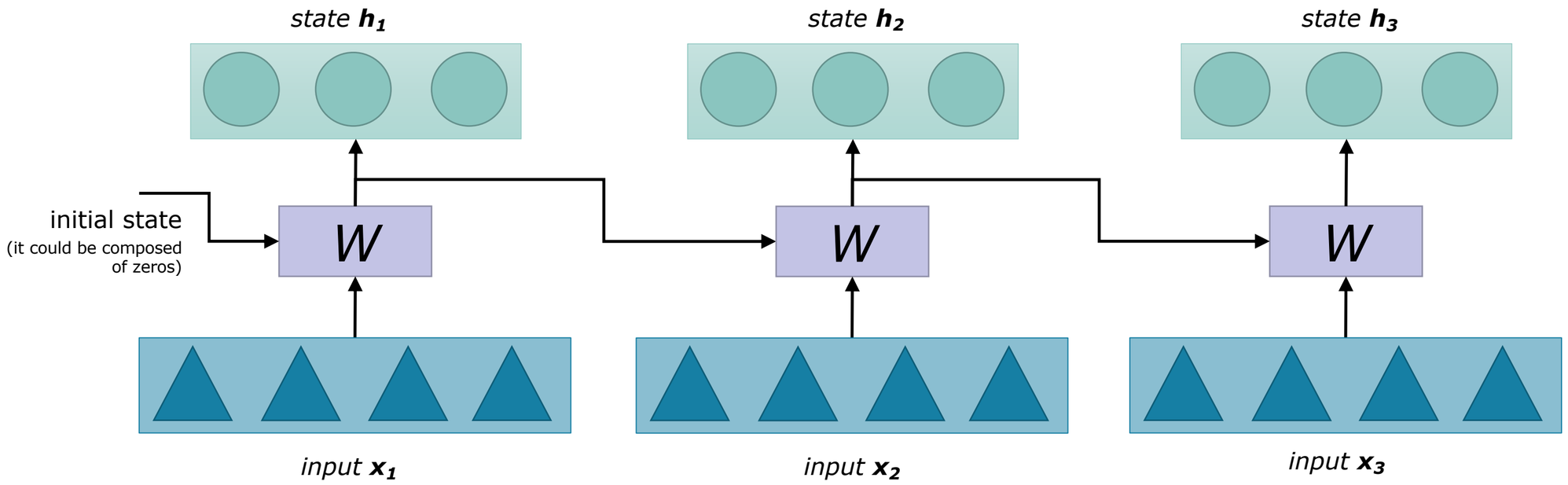
The same computation that is performed by classic feed-forward neural networks!

$$\mathbf{h}_t = \sigma\left(W\left[\mathbf{h}_{t-1}, \mathbf{x}_t\right] + \mathbf{b}\right)$$

*[**a**,**b**] = concatenation of the two vectors **a** and **b***

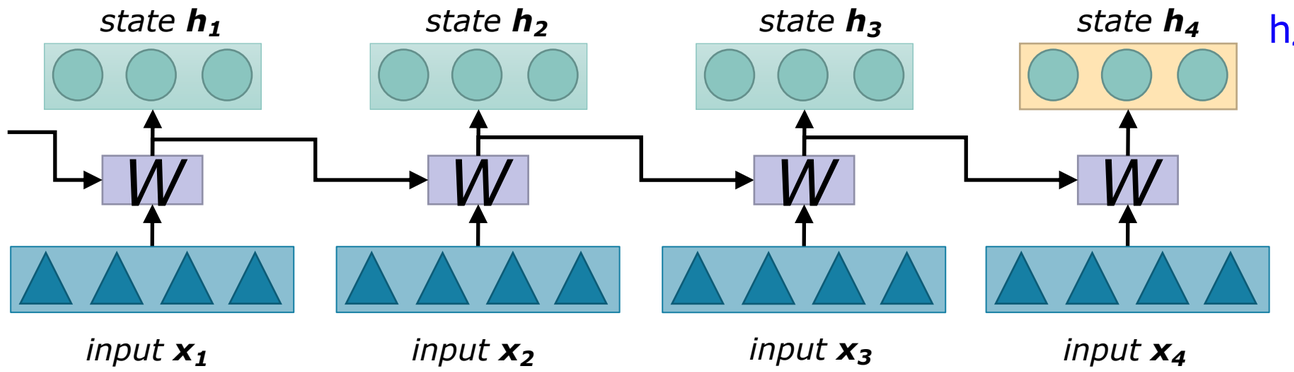
Unfolding

- In order to better visualize the computation that is performed by an RNN, we can *unfold* the network *through "time"* *(in the following example, a sequence of length 3 is considered)*
  - The **same weights W** are used in all the time instants



The Final State: Fixed-Length Representation

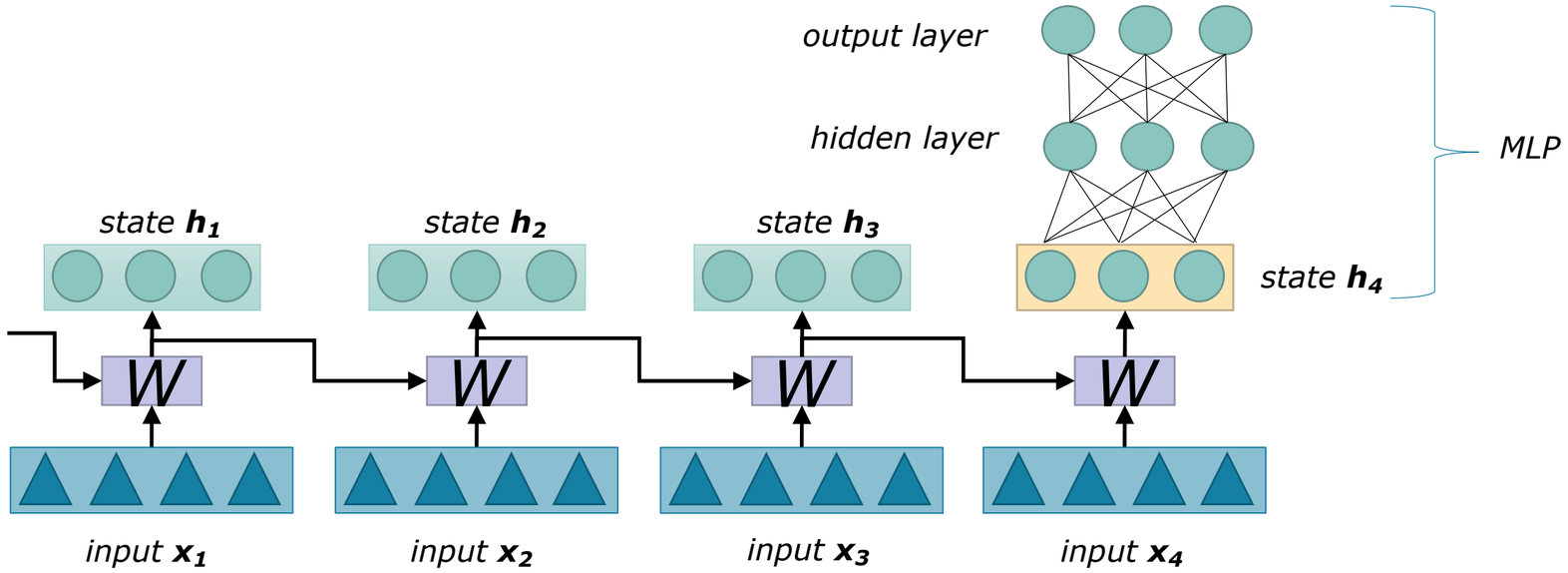
- Once the whole sequence has been processed by the RNN, we have the use of the last updated **state** vector
- Such vector is a form of "**memory**" of the network, and it is a representation of the *whole* input sequence
  - As a matter of fact, the state is updated after every time step
- It is a **fixed-length representation** of a sequence
  - Sequences of different lengths are mapped into the same fixed-length representation



h_4 is a representation of length 3 of the given input sequence

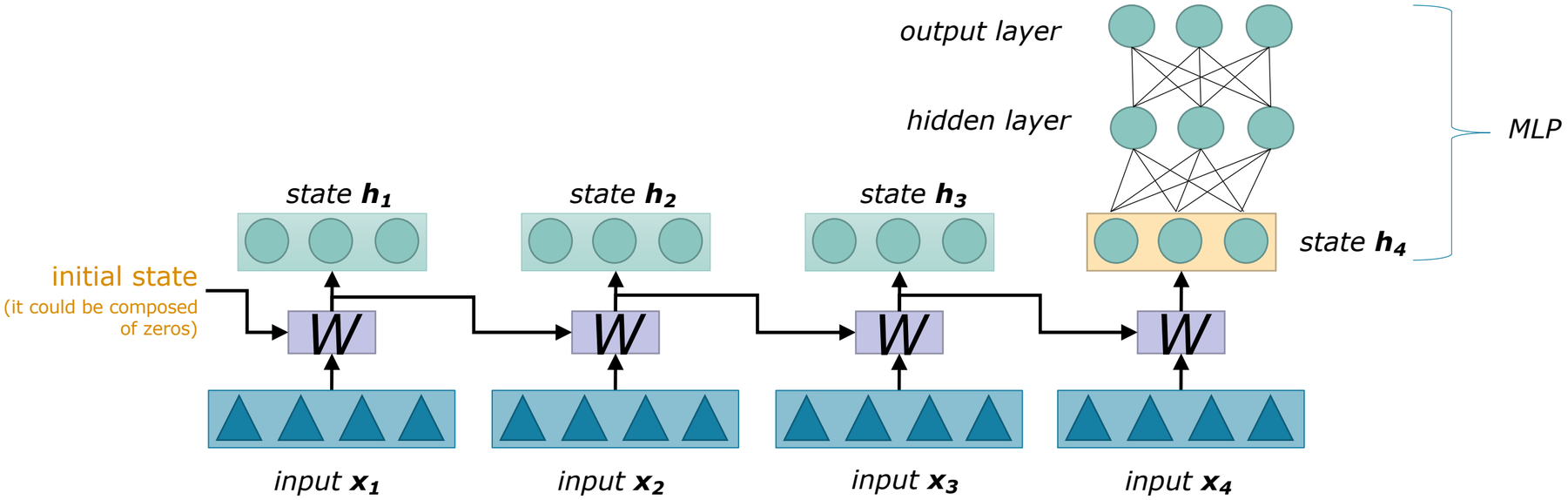
Making Predictions over Sequences

- Once we are given the final state of the network, we can provide it to an MLP to classify it, or to make other kind of predictions on the whole sequence
  - Note: *we could also make predictions at each time step, or evaluate differently organized architecture, but it goes beyond the scope of this lecture*



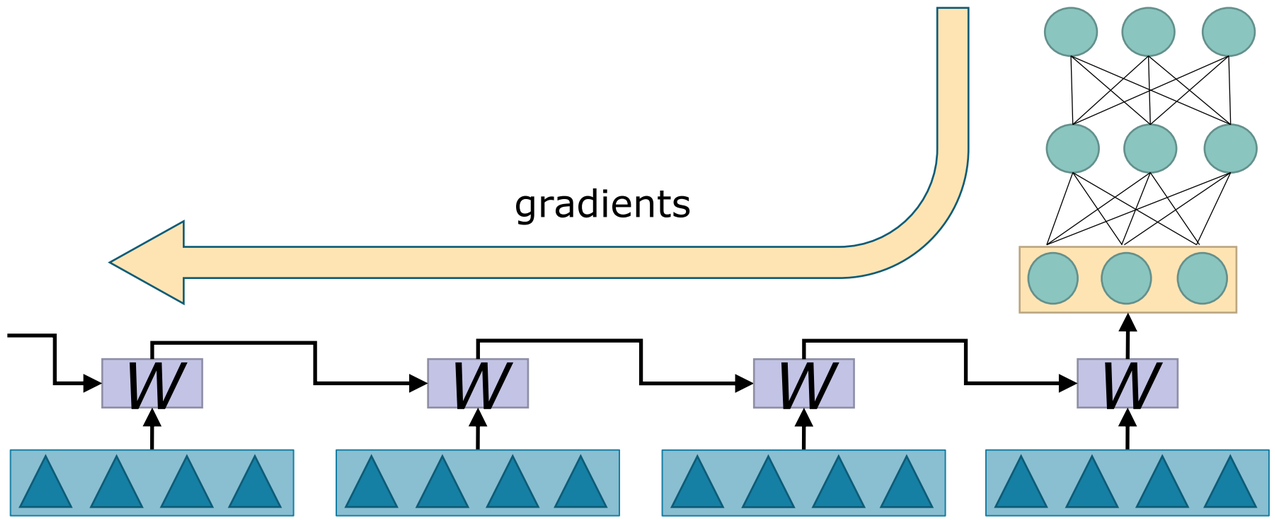
Making Predictions over Sequences (2)

- Before processing an input sentence, we have to take care of **resetting** the initial state of the network to avoid the network to be influenced by previously processed sequences
  - Resetting is frequently implemented by zeroing the initial state vector



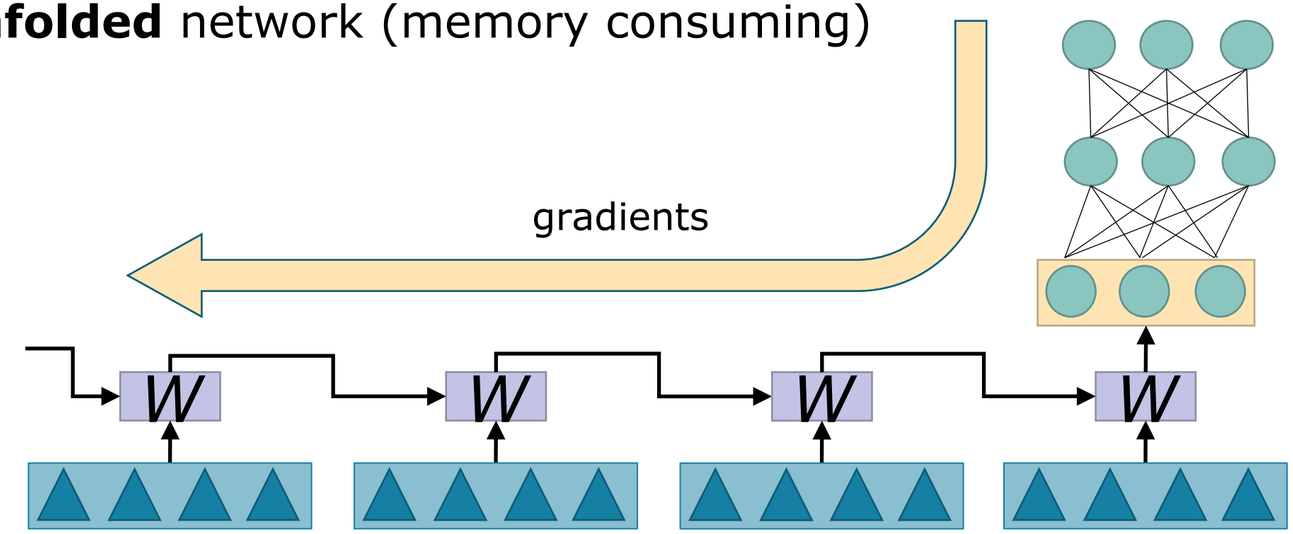
Training Recurrent Neural Networks

- Recurrent Networks are trained by Backpropagation
- However, we have to take care of unfolding the network in order to correctly compute the gradients along the whole sequence
  - **Backpropagation Through Time (BPTT)**



Training Recurrent Neural Networks (2)

- Since the weights in *W* are the same at each time step (**shared**), the gradients of the loss function with respect to *W* are accumulated (summed up) during Backprop
  - BPTT needs to memorize the hidden states that are generated by the network during the forward stage, for all the time instants
  - It also needs further memory structures to keep track of the gradient contributes for the whole **unfolded** network (memory consuming)



Long-term Dependencies

- If we process a **long sequence**, the temporal unfolding generates a huge structure

- The classic problem of vanishing (or exploding) gradients affects the capabilities of RNN to learn from long sequences (long-term memories)
  - For example, suppose we have a long sequence that we want to classify, and suppose that the first portion of the sequence is the one that is crucial in the classification process, while the rest is only noise
  - Due to the vanishing gradients, the network might be not able to compute meaningful gradients and to correctly update *W*

Long Short Term Memory RNNs (LSTMs)

- Long Short Term Memories (**LSTMs**) are a variant of the classic RNN model, designed to increase the memorization capabilities over time
  - Hochreiter and Schmidhuber, Long Short-Term Memory, *Neural Computation* 1997

- LSTMs include multiple computational blocks, out of which we mention nonlinear gating units to regulate the information flow
  - Basically, the network can learn to discard some elements of the sequence
  - It learns to "forget"

- We will describe them in detail at the end of this module

Language Modeling with Recurrent Neural Networks

Stefano Melacci

Language Modeling and RNNs

- RNNs can be used to learn a Language Model from data
- We can use RNN to build a neural architecture that computes the probability of emitting a certain word, given the sequence of words that precede it

$$P(w_i|w_{i-1}, w_{i-2}, \ldots, w_1)$$

- The network will be able to “read” text and learn the properties of the considered language
- *In order to setup the appropriate neural architecture, we have to consider some aspects that are related to the way **words** are represented*

Representing Words

- Basic problem in every Natural Language Processing-related application
- Common approach: encode each word with the index of its **position in the considered vocabulary *V***

- First, a vocabulary *V* must de defined (list of words)
  - For example, by parsing a large corpus and extracting the list of words that belong to it
  - This process can include also misspelled words and other "artifacts" that might not be of interest (URLs, …)
  - It is better to keep the list of the most frequent *k* words
  - The size of the vocabulary is task dependent: *k* is frequently of the order of *thousands* or *millions*
  - A special vocabulary element is always introduce as placeholder for **out-of-vocabulary** words

Representing Words (2)

- Then, each word is represented with a **1-hot vector of size |*V*|** (the size of the vocabulary)
  - A vector of all-zeros with the exception of an element, that is set to 1
- The 1-hot vector represent the index of the word in *V*
- For example:

Vocabulary *V*

1	is
2	house
3	cat
4	must
5	nice
6	bat
7	fun
8	are
9	goes
10	my

Word	**→ Representation**
my	→ [0 0 0 0 0 0 0 0 0 **1**]
cat	→ [0 0 **1** 0 0 0 0 0 0 0]
is	→ [**1** 0 0 0 0 0 0 0 0 0]
nice	→ [0 0 0 0 **1** 0 0 0 0 0]

Representing Words (3)

- This representation has the main drawback of not modeling any semantics
- Words with similar meanings (*house, home*) or words that belong to the same category of concepts (*cat, dog*) will have orthogonal representations
  - The distance between each pair of words is the same, independently on the word themselves

Vocabulary *V*		→ Representation
1	is	→ [**1** 0 0 0 0 0 0 0 0 0]
2	house	→ [0 **1** 0 0 0 0 0 0 0 0]
3	cat	→ [0 0 **1** 0 0 0 0 0 0 0]
4	must	→ [0 0 0 **1** 0 0 0 0 0 0]
5	nice	→ [0 0 0 0 **1** 0 0 0 0 0]
6	home	→ [0 0 0 0 0 **1** 0 0 0 0]
7	fun	→ [0 0 0 0 0 0 **1** 0 0 0]
8	dog	→ [0 0 0 0 0 0 0 **1** 0 0]
9	goes	→ [0 0 0 0 0 0 0 0 **1** 0]
10	my	→ [0 0 0 0 0 0 0 0 0 **1**]

Representing Words (4)

- Nowadays Machine Learning-based NLP applications exploit **learnable representation of words (word embeddings)**
- Each word is represented by a vector of length *z* (design choice), and the elements of the vectors are parameters that must be learnt (they are treated as weights of the network)

Vocabulary *V*		**→ Representation**
1	is	→ $[w_{01}\ w_{02}\ w_{03}\ w_{04}]$
2	house	→ $[w_{05}\ w_{06}\ w_{07}\ w_{08}]$
3	cat	→ $[w_{09}\ w_{10}\ w_{11}\ w_{12}]$
4	must	→ $[w_{13}\ w_{14}\ w_{15}\ w_{16}]$
5	nice	→ $[w_{17}\ w_{18}\ w_{19}\ w_{20}]$
6	home	→ $[w_{21}\ w_{22}\ w_{23}\ w_{24}]$
7	fun	→ $[w_{25}\ w_{26}\ w_{27}\ w_{28}]$
8	dog	→ $[w_{29}\ w_{30}\ w_{31}\ w_{32}]$
9	goes	→ $[w_{33}\ w_{34}\ w_{35}\ w_{36}]$
10	my	→ $[w_{37}\ w_{38}\ w_{39}\ w_{40}]$

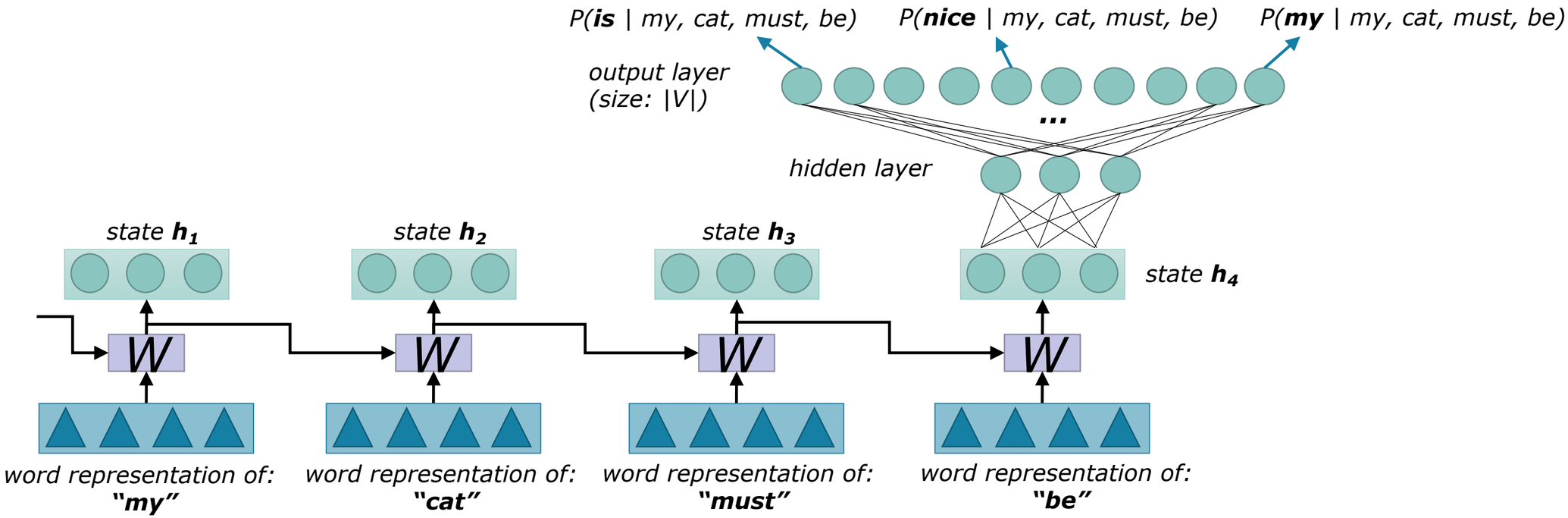
Representing Words (5)

- When this representations are used as inputs for the Language Modeling task (or other NLP tasks), the machine will also learn the values of the word representations
  - For example, if two words are **synonyms**, the machine could learn that they must be represented with similar vectors, thus facilitating the learning process of the whole Language Model

	Vocabulary *V*	**→ Representation**
1	is	→ […]
2	house	→ [-1.1 -5.7 +0.9 +0.6]
3	cat	→ [+0.4 +1.3 -1.0 +5.2]
4	must	→ […]
5	nice	→ […]
6	home	→ [-1.1 -5.7 +0.8 +0.5]
7	fun	→ […]
8	dog	→ [+0.3 +1.4 -0.9 +5.1]
9	goes	→ […]
10	my	→ […]

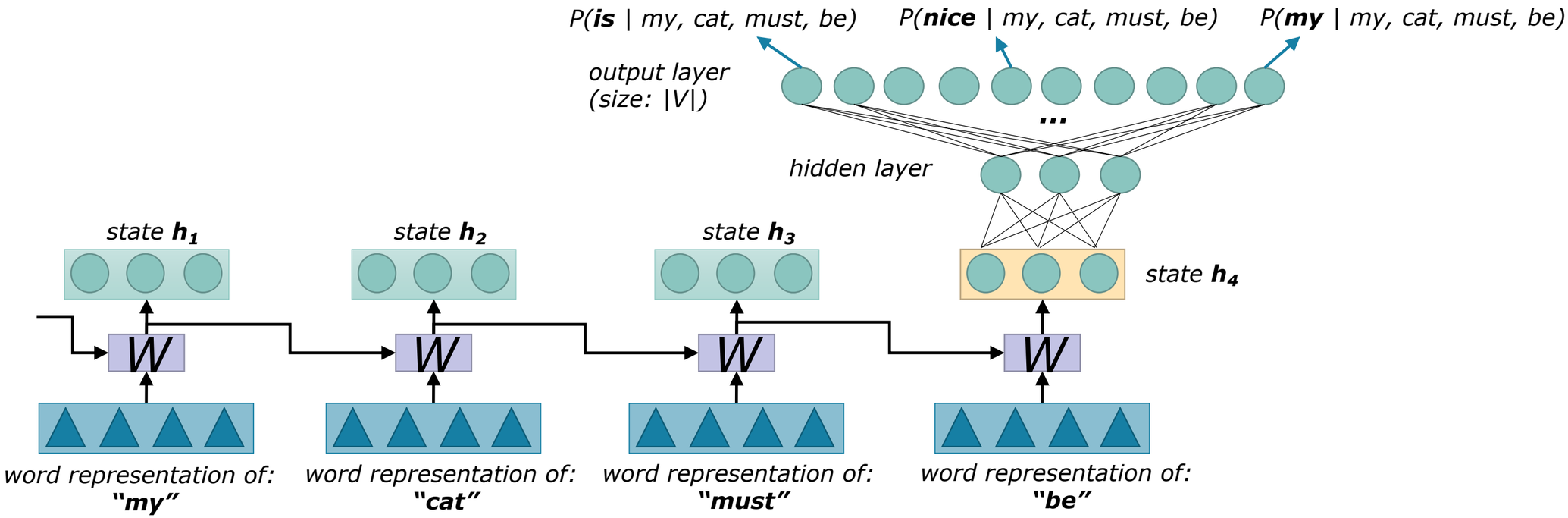
RNN-based Neural Network for LM

- An RNN-based neural architecture for Language Modeling is represented in the following scheme (unfolded)
  - The sample sentence that is provided to the network is: *my cat must be*



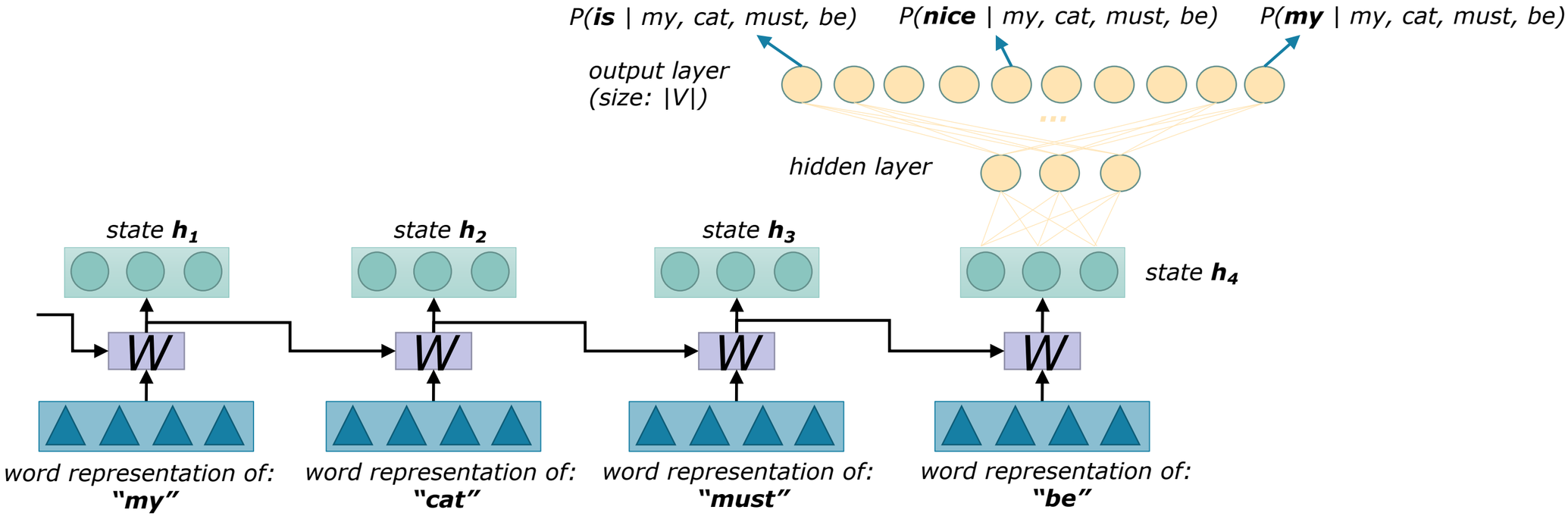
RNN-based Neural Network for LM (2)

- The RNN progressively computes a representation (**state**) of the whole context (the words that precede the one we want to guess)



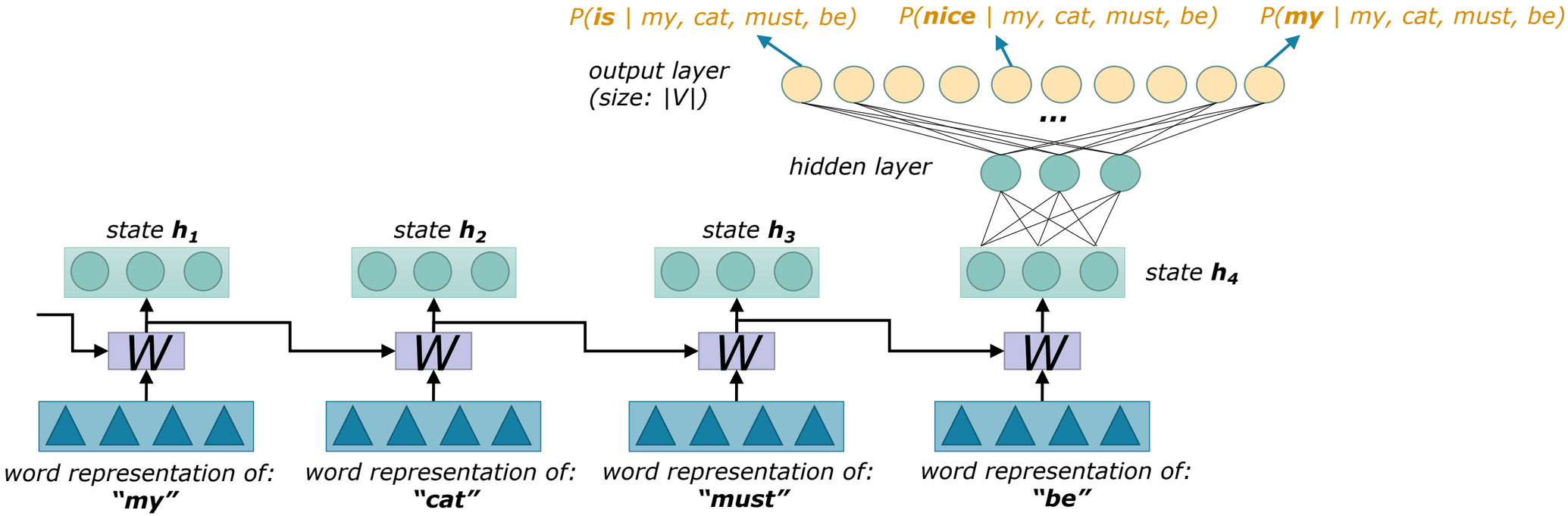
RNN-based Neural Network for LM (3)

- An MLP processes the representation of the context
- The output layer of the MLP is composed by a number of neurons that is the same as the size of the vocabulary *V*



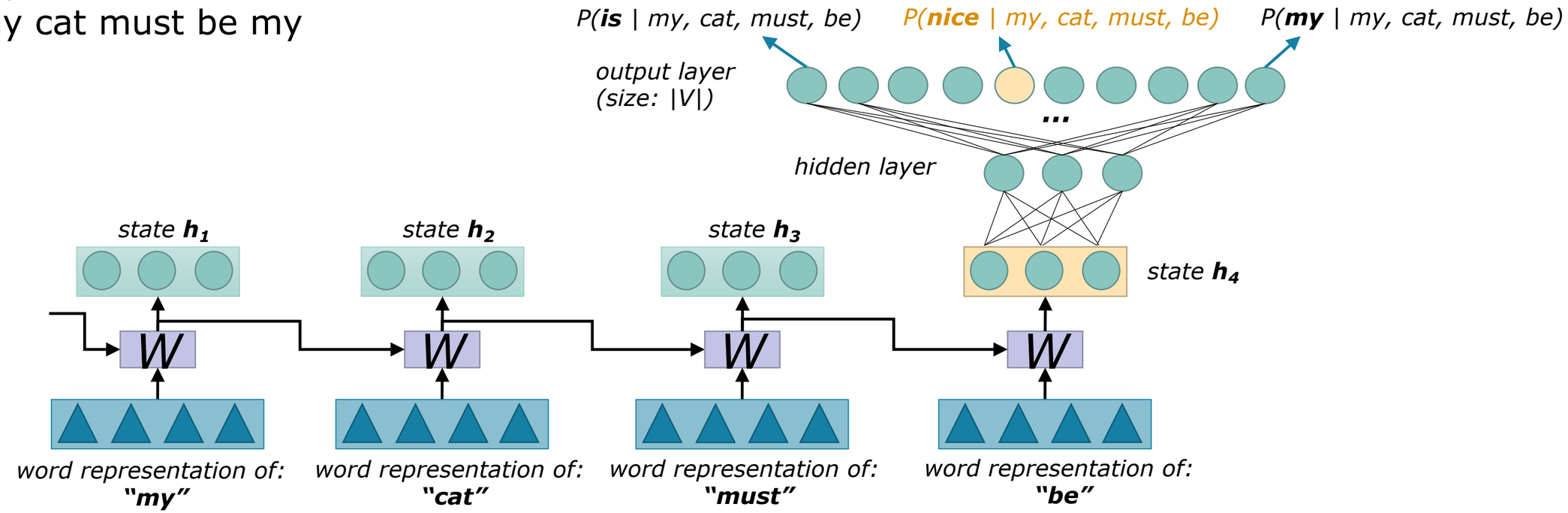
RNN-based Neural Network for LM (4)

- The softmax activation function is exploited in the output layer, so that the MLP actually outputs probabilities
  - The *i*-th output neuron models the *probability* that the *i*-th word of the vocabulary will be the *next word of the sentence*



RNN-based Neural Network for LM (5)

- We expect the model to learn that the probability of "nice" is larger than the ones of "is" and "my"
  - **my cat must be nice**
  - my cat must be is
  - my cat must be my



Training the RNN-based LM

- Given a large corpus of text in the target language, we can train the neural LM in a straightforward way

1. Define the vocabulary *V*
2. Segment the text into sentences, to avoid the network to have to process long sequences
3. For each word of the sentence
    1. **Reset the initial state of the network**
    2. Take the sequence on the left-side of the word, and provide it as input to network
    3. Let the network predict the probability distribution over the words of *V*
    4. Compute the mismatch between the predicted probability on the target word and 1.0
4. Compute the gradients by Backprop (BPTT) and update the weights of the whole net

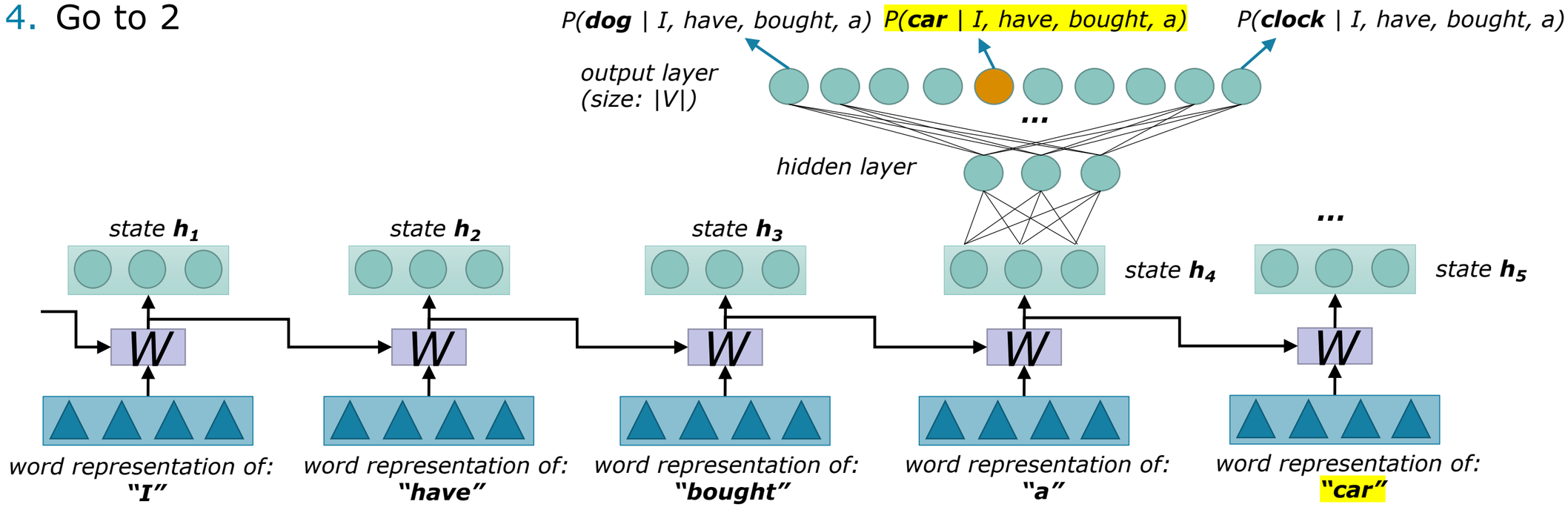
Training the RNN-based LM (2)

- Due to the update scheme based on a local feedback, the network is forced to learn regularities of the target language
- Once the training is complete, the network is a compact LM that can be used to estimate the probability of a sentence by

$$P(w_1, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_{i-1}, w_{i-2}, \ldots, w_1)$$

Generating Language

- Once we are given a *Language Model*, we can use it to **generate language**
  1. Provide an initial context
  2. Generate the next word as the most likely word accordingly to the *Language Model*
  3. Augment the initial context
  4. Go to 2



Example

- Given the following training data

  yesterday he read a book
  I really like playing with my cat
  he really wants to sign a song
  my kids are playing in the garden

- Once the RNN has been trained, we provide the following input

  he wants to read (The word with the highest predicted probability is "**a**")

- Then, we provide the augmented context as input to the network

  he wants to read **a** (The word with the highest predicted probability is "**book**")

- The network was able to complete the partial sentence using the learned language model

  he wants to read **a book**

Long-Short Term Memories

Stefano Melacci

Long-Short Term Memory RNNs

- Long Short Term Memories (**LSTMs**) are a variant of the classic RNN model, designed to increase the memorization capabilities over time
  - Hochreiter and Schmidhuber, Long Short-Term Memory, *Neural Computation* 1997

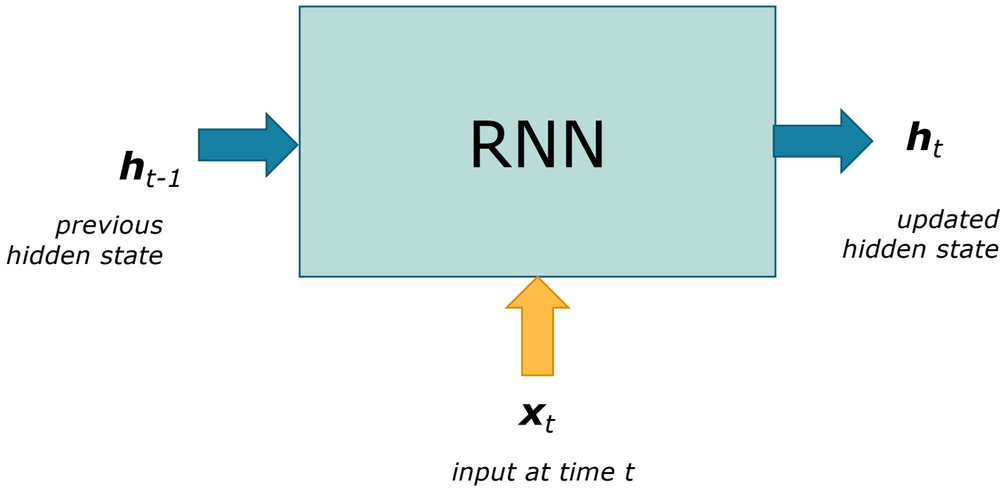
- They gained a huge popularity in the last decade, following the success of Deep Learning
  - They have more parameters than "vanilla" RNNs, and their structure is more complicated…
  - …modern machines have no problems in handling LSTMs
  - Moreover, LSTMs achieved strong results in Speech Recognition and other language-related tasks, and several major companies started to use them in their products
  - There exists a number of variants of RNNs that are inspired by LSTMs

RNN → LSTM

RNNs vs. LSTMs

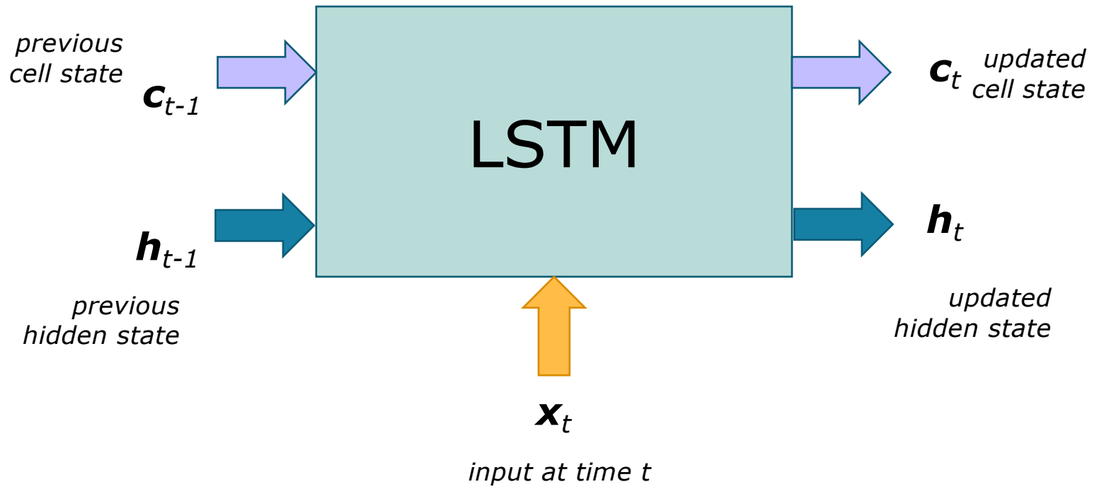
- From the external point of view, RNNs and LSTMs have a very similar structure
- However LSTMs include a new input/output signal, that is the **cell state (*c*)**

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$



Vector composed of the concatenation of $\mathbf{h}_t$ and $\mathbf{c}_t$

$$[\mathbf{h}_t, \mathbf{c}_t] = f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1})$$



RNNs vs. LSTMs

- RNNs update their internal state (hidden representation) at each time instant
  - Differently, what makes LSTMs more robust to long-term term dependencies is the fact that they **can learn** *if they should update their internal parameters or not*

- LSTMS are rooted around the idea of **gating function**
  - It is a function in [0,1] that modulates *a given signal*
    - If the gate is 0, the given signal is **zeroed**
    - If the gate is 1, the given signal is left **unchanged**

- The gating function is a *single-layer neural network* with *sigmoidal* activation units, that output a vector of values in [0,1]
  - LSTMS learn the weights of the gating functions
    - They learn if they should zero the gate signal or not

Cell State

- In vanilla RNNs, the hidden state is updated by the current input and the previous hidden state

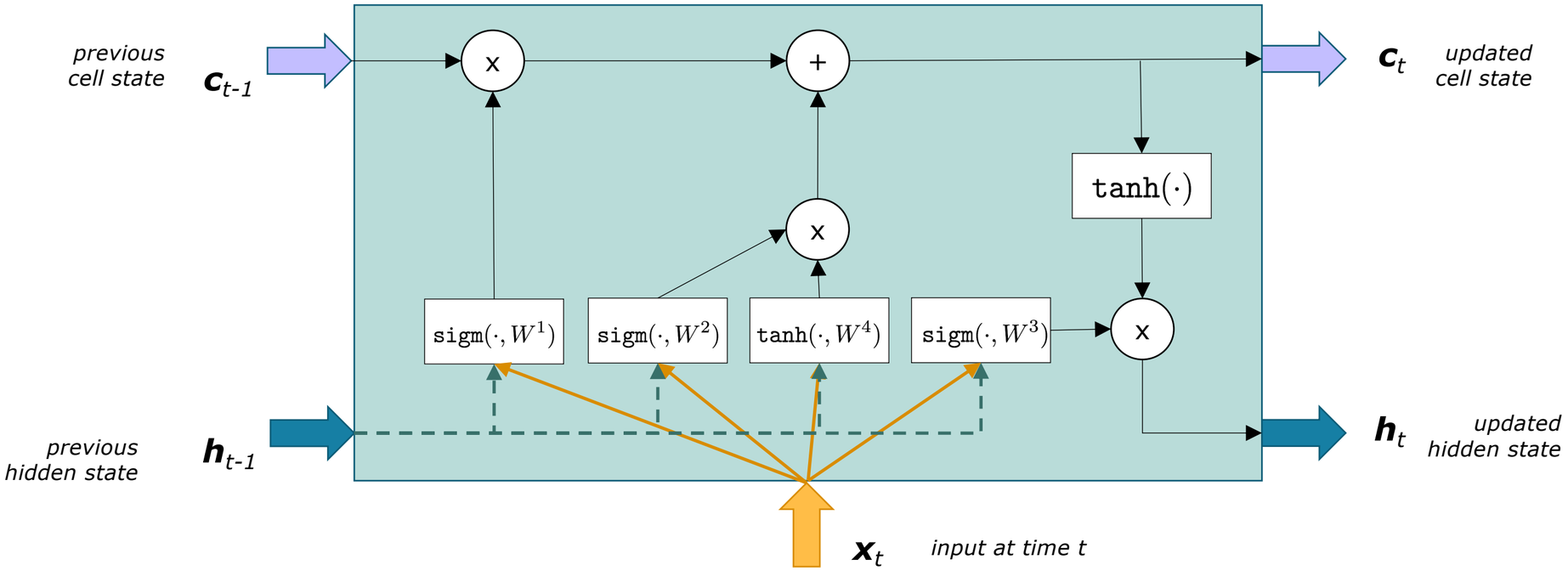
$$\mathbf{h}_t = \tanh(W[\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b})$$

- In LSTMS, the cell state is modeled by a vector of a given size (design choice) and it introduces an *indirection* in the way the hidden state is updated
  1. The **cell state** is updated considering the current input and the ***previous hidden state***
  2. The ***hidden state*** is updated by means of the **cell state**

- This choice introduces more flexibility in the hidden state update operation that, as we will see in the next slides, is intermixed with several learnable **gates**
  - More freedom

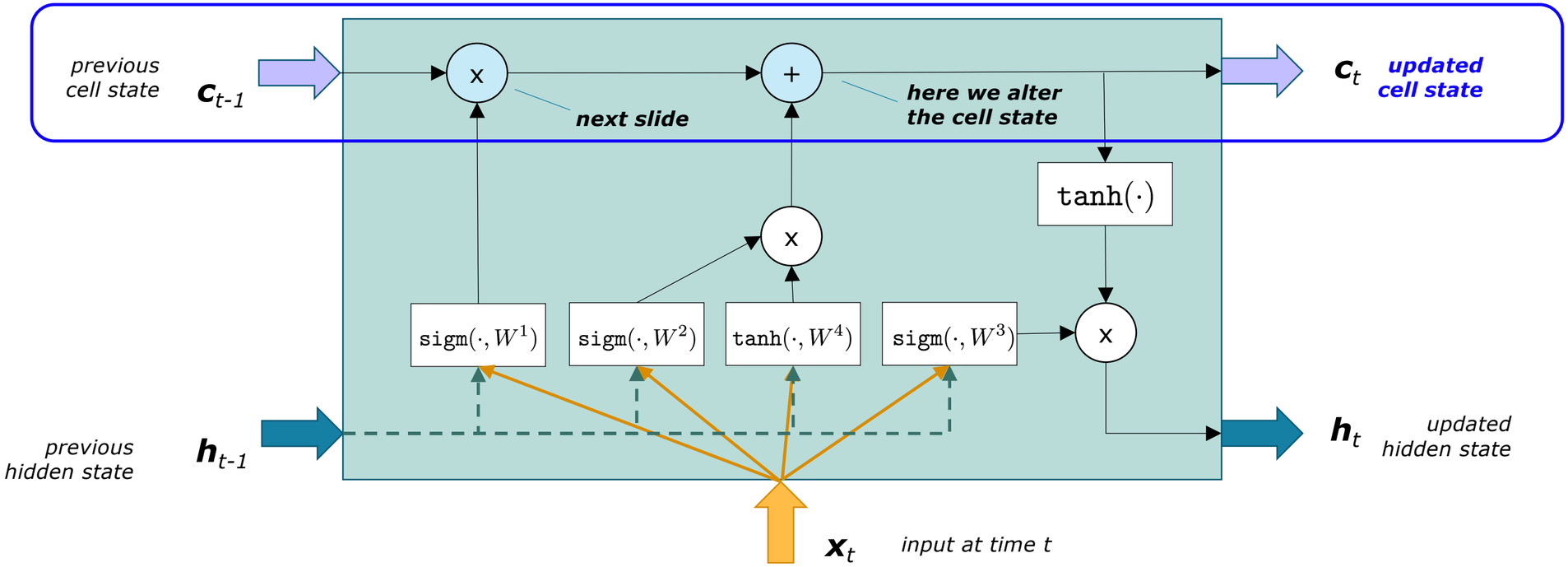
LSTM Cell

- An LSTM "cell" is composed by several operations that are equivalent to the ones computed by a neural network layer
  - Linear projection W^i + activation function, either **sigm**oid or hyperbolic tangent (**tanh**)
  - Then we also have element-wise product **x** and sum **+**



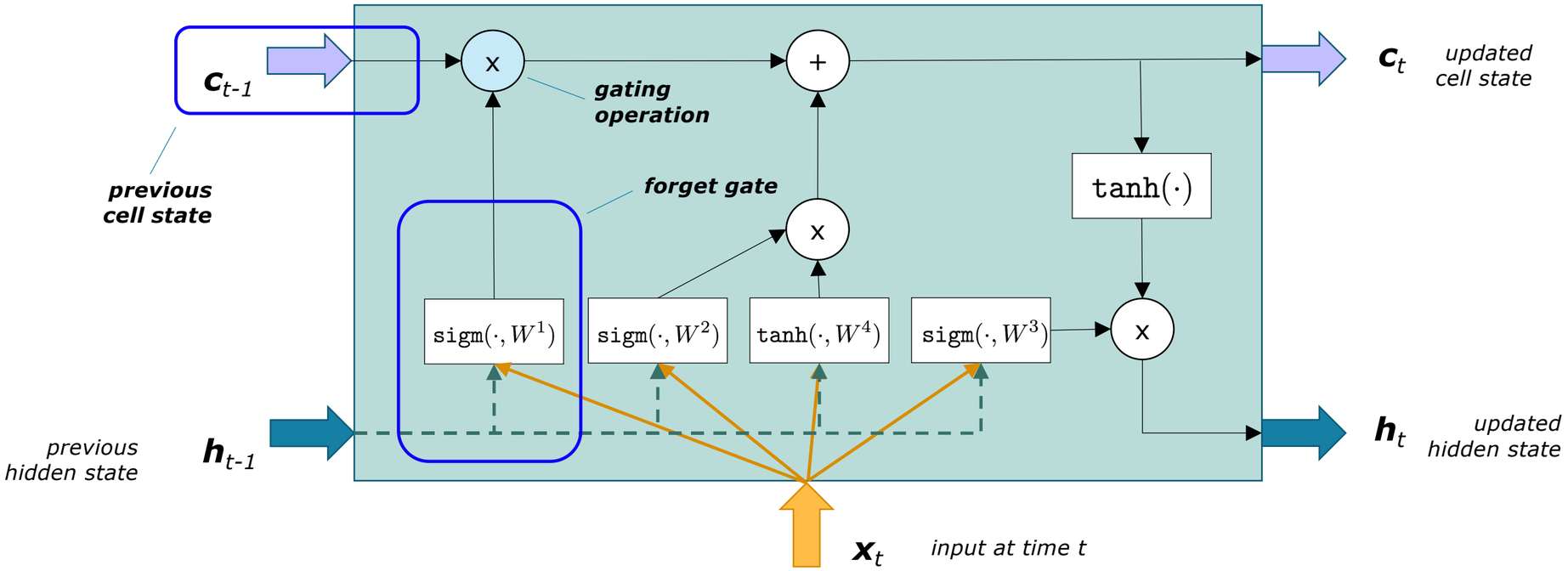
LSTM Cell (2)

- **Cell state**: the main signal of the cell
  - LSTMs alter the information of the cell state c_{t-1}
  - The way the hidden state h_{t-1} will be updated depends on the cell state!



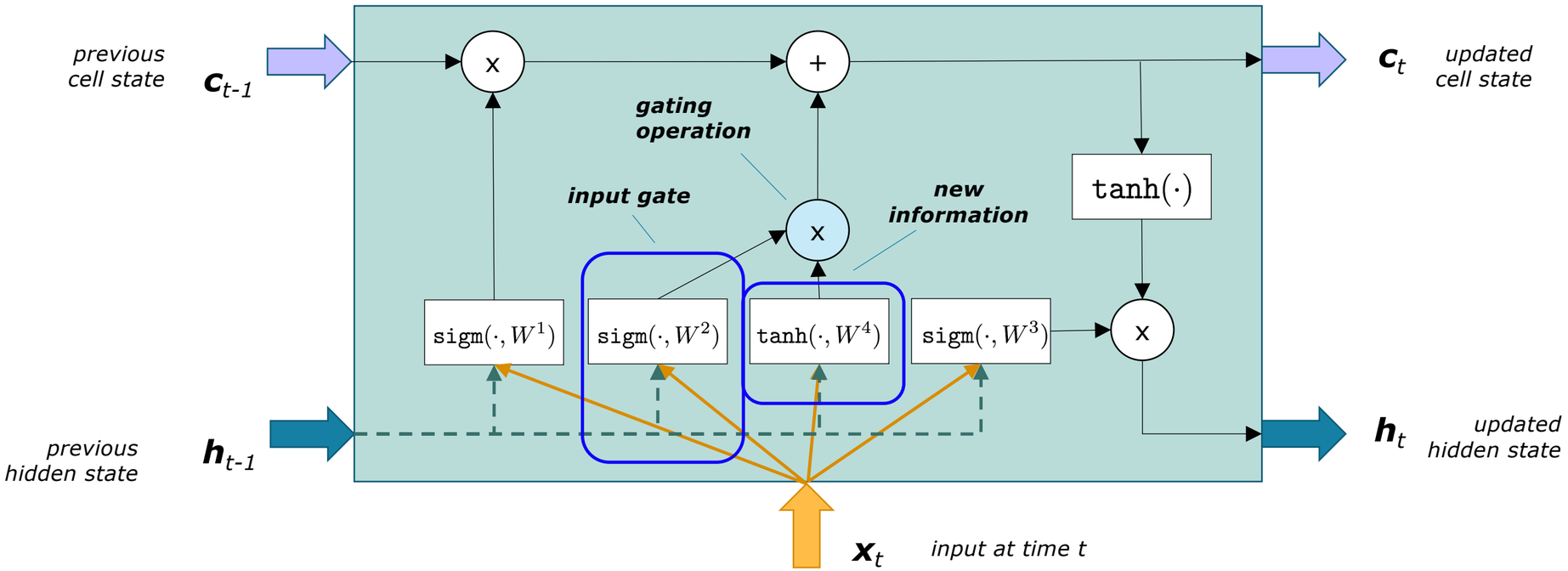
LSTM Cell (3)

- **Forget gate**: it decides to what extent we want to keep or forget the previous cell state
  - If the gate is zero, the previous cell state is forgotten!



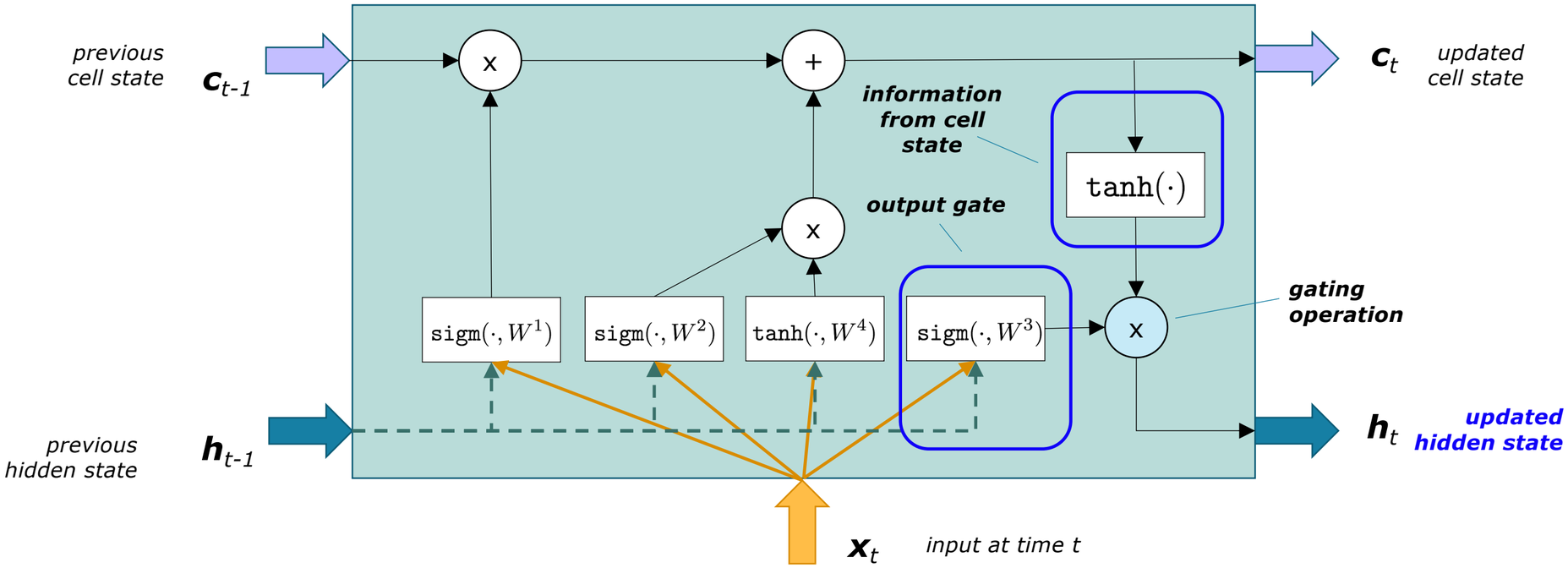
LSTM Cell (4)

- **Input gate**: it decides to what extent we want to inject "new information" into the cell state



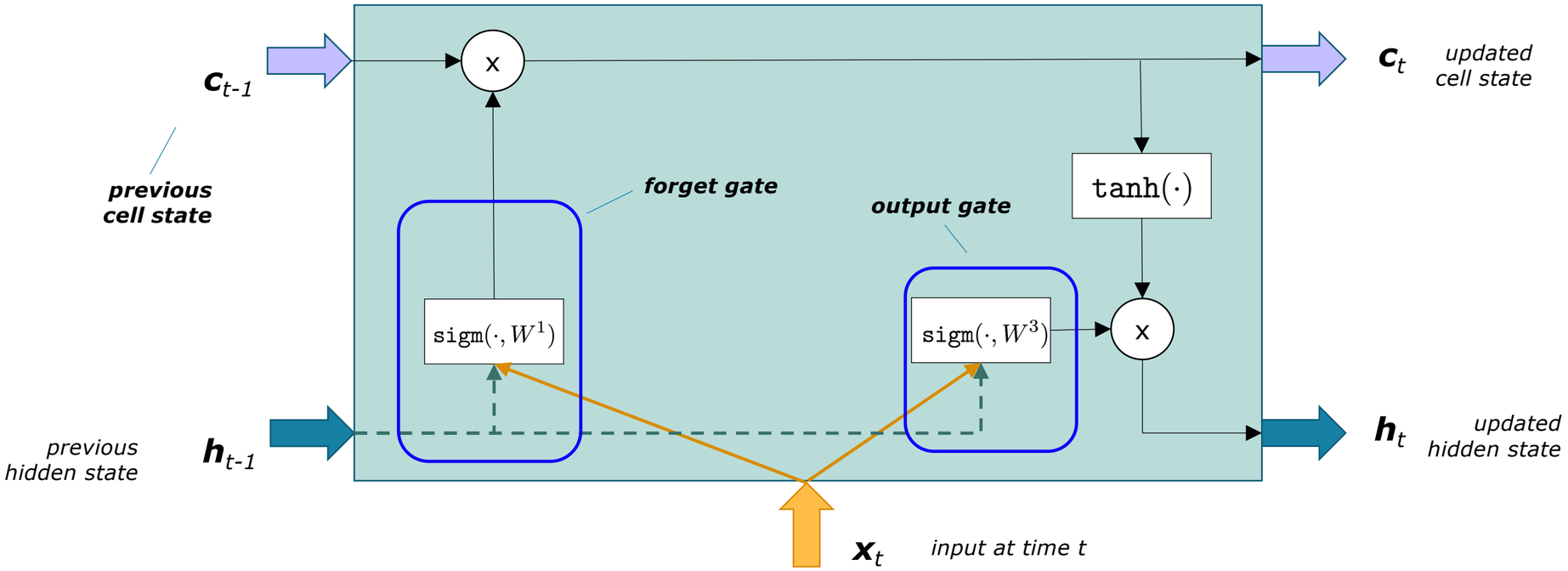
LSTM Cell (5)

- **Output gate**: it decides to what extent we want to propagate the new cell state c_t to generate the new hidden state h_t
  - The cell state is first squashed with a tanh function (no weights to learn here)
    - It is just a normalization operation, that enforces the cell state to stay in [-1,1]



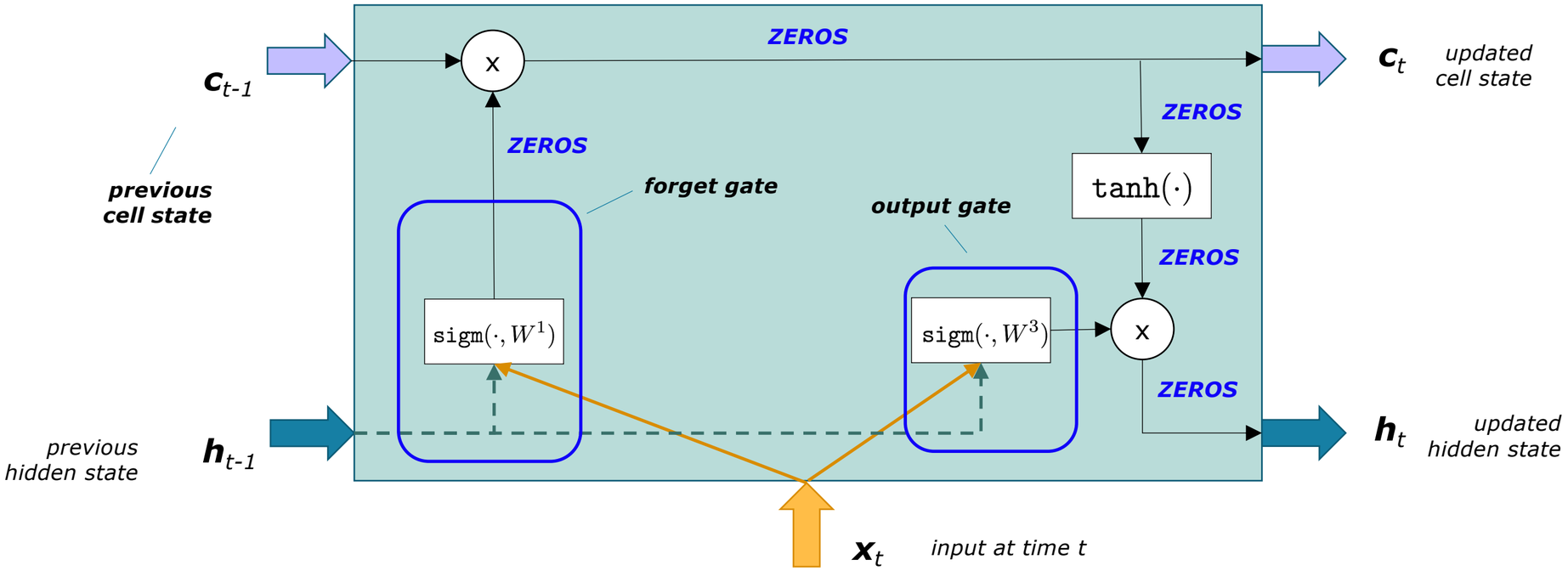
LSTM Cell - Example

- Example: when the **input** gate yield zeros, the new hidden state will only depend on the previous cell state (not on the current input $\boldsymbol{x}_t$)
  - Of course, it also depends on what the **forget** and **output** gates will decide!



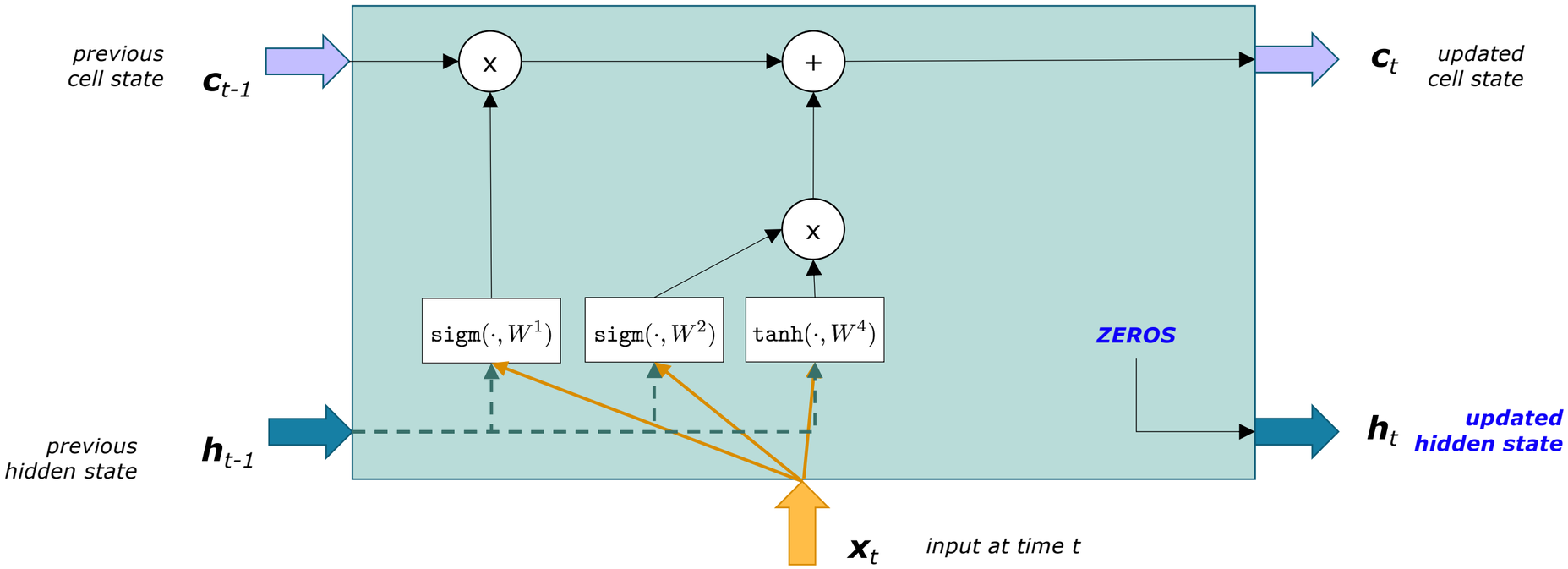
LSTM Cell - Example (2)

- Still on the same example: if the **forget** gate is zero too, then, independently on what the **output** gate will decide, the <u>hidden</u> state will be reset (zeroed)



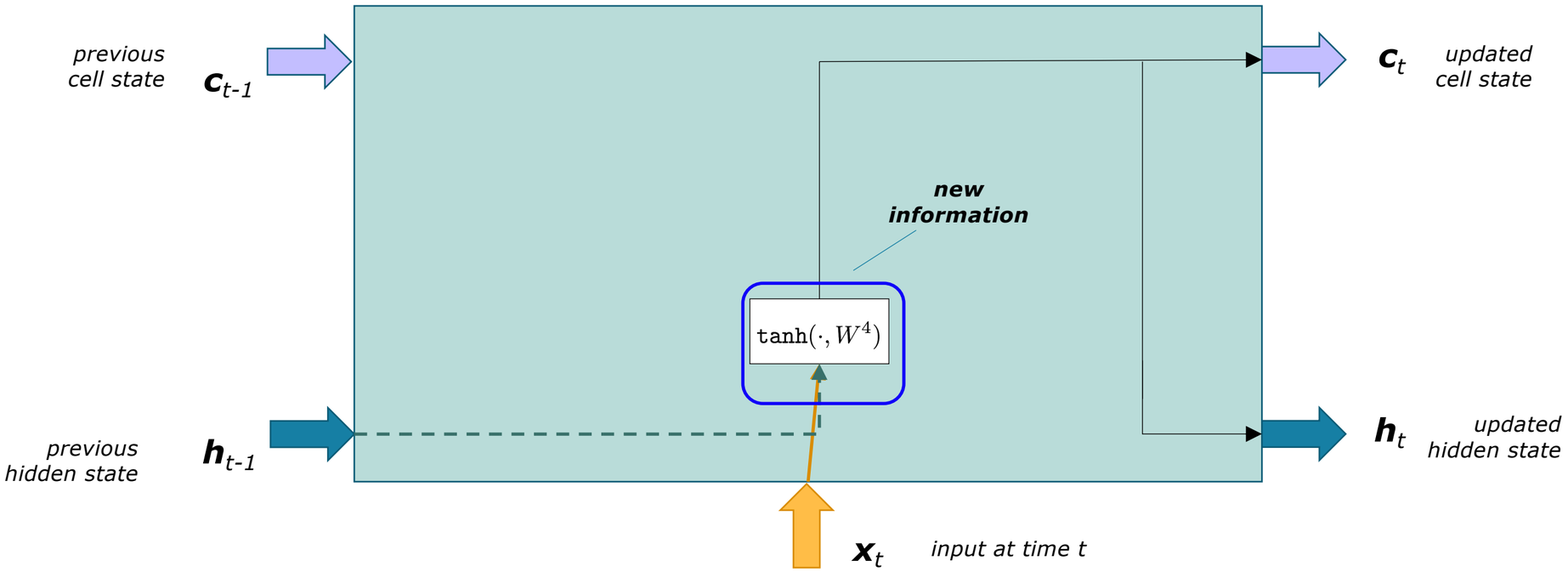
LSTM Cell - Example (3)

- Another example: if the **output** gate is zero, then the new hidden state will be composed of zeros, independently on the rest of the LSTM cell
  - Notice that the cell state will be still updated (if the other gates are not blocking the signal)



LSTM Cell - Example (4)

- Yet another example: if the **forget gate** is zero, the **input** and **output gates** yield ones, we get an instance of the hidden state update rule in vanilla RNNs



LSTM Cell (6)

- We can formalize the previously described operations as follows
  - **Superscripts are indices**, not exponents
  - Activation functions operate *element-wise* on the components of their arguments
  - The "dot" symbol is an *element-wise* product
  - We also show the bias vectors $\boldsymbol{b}^j$

*forget **gate***

$$\mathbf{f}_t = \mathtt{sigm}(W^1[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^1)$$

*input **gate***

$$\mathbf{i}_t = \mathtt{sigm}(W^2[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^2)$$

*output **gate***

$$\mathbf{o}_t = \mathtt{sigm}(W^3[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^3)$$

updating the cell state

$$\mathbf{c}_t = \mathbf{f}_t \cdot \mathbf{c}_{t-1} + \mathbf{i}_t \cdot \mathtt{tanh}(W^4[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^4)$$

updating the hidden state

$$\mathbf{h}_t = \mathbf{o}_t \cdot \mathtt{tanh}(\mathbf{c}_t)$$

Vanishing Gradients (Intuition)

- LSTMs still stuffer from the problem of exploding/vanishing gradients
  - As every "deep" architecture
  - LSTMs are structured in a way that make them *a bit more robust* than vanilla RNNs

- LSTMs can make a proficient use of the **forget** gate: if the gate is open (1), then the cell state is updated by an **addition** operation
  - *The network only learns the difference between consecutive cell states*
  - It is easier to preserve information

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \texttt{vanilla\_RNN\_update\_rule}$$

- Differently, RNNs directly estimate the new hidden state
  - "Weight matrix"-times-"previous hidden state"
  - Harder to *always* preserve info from the previous hidden state, especially when weights are small