

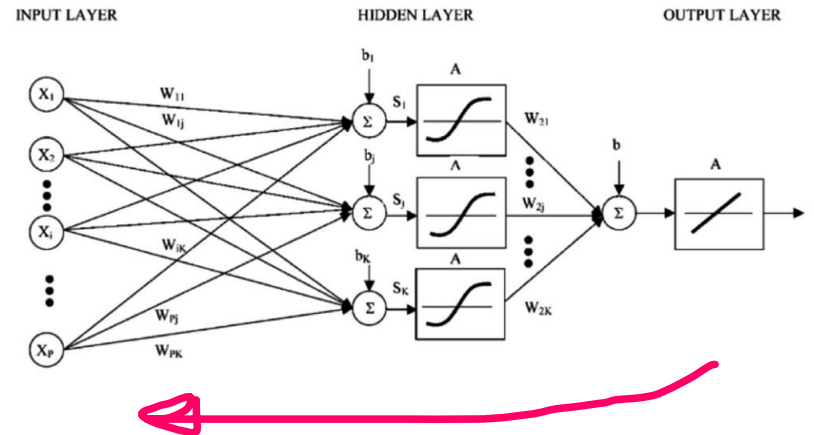
# Recurrent Neural Networks

RNN, LSTM , GRU..

In part from J. Canny, CS294-129 2016 and from Mooney (UTexas, 2015)

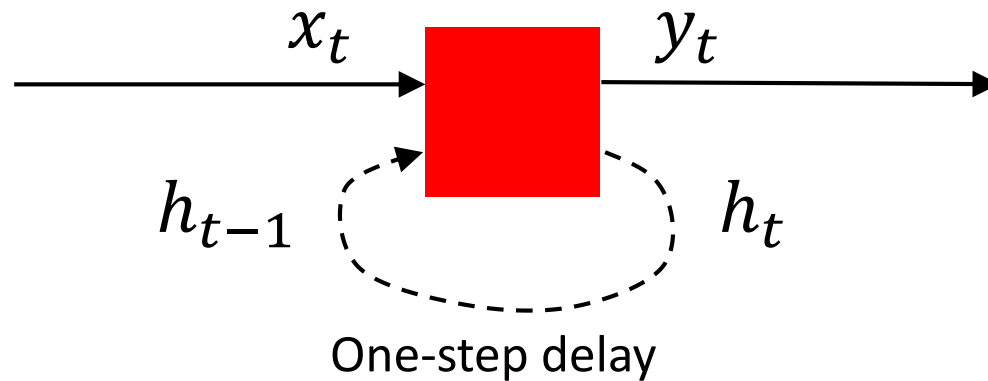
Standard Neural Networks are DAGs (Directed Acyclic Graphs).  
Which means they have a topological ordering.

- The topological ordering is used for activation propagation, and for gradient back-propagation.



- They are implemented as *combinatorial* logic devices.

Recurrent networks introduce **cycles** and a notion of **time**.



- They are designed to process **sequences of data**  $x_0, \dots, x_t, \dots, x_n$  and can produce sequences of outputs  $y_0, \dots, y_m$ .
- They are implemented as **sequential logic devices**, and their behaviour can be described as a sequence of *states* and *transitions*.
- Tuples  $x_t, y_t$  can be single values but in general they are **vectors**

# Sequential data: any kind of data where the order matters

- Examples: sequences of discrete (symbolic) or continuous values
  - Customer purchases history: discrete
  - Patients health records (sequence of medications, diagnoses..): discrete
  - Students actions in and e-learning platforms: discrete
  - Stock market prices : continuous
  - Sensor data where every point is an observation at a given time: continuous
  - Texts (sequence of words): discrete
  - Video (sequence of frames) : continuous
  - Sequence elements can be univariate or multivariate, uninomial or multinomial

# Uni/multi variate, uni/multinomial

## With respect to the number of variables:






- **Univariate:** only one variable, e.g. the sequence of values of a stock market price
- **Multivariate:** many variables, e.g., the sequence of values of different sensors. Every  $\mathbf{x}$  at time  $t$  is a vector of values ( $x_i$  is the value of sensor  $i$  at time  $t$ ).

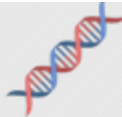
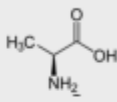







## With respect to the type of variables:

- Discrete (symbolic) variables: they can be uninomial or multinomial distributions. Uninomial have only two values, multinomial have many values.
- Continuous variables may assume any value, and can follow different distributions, e.g., Poisson, Gaussian, Pareto..

**Algorithms may differ according to the type of variables and distribution of values.**

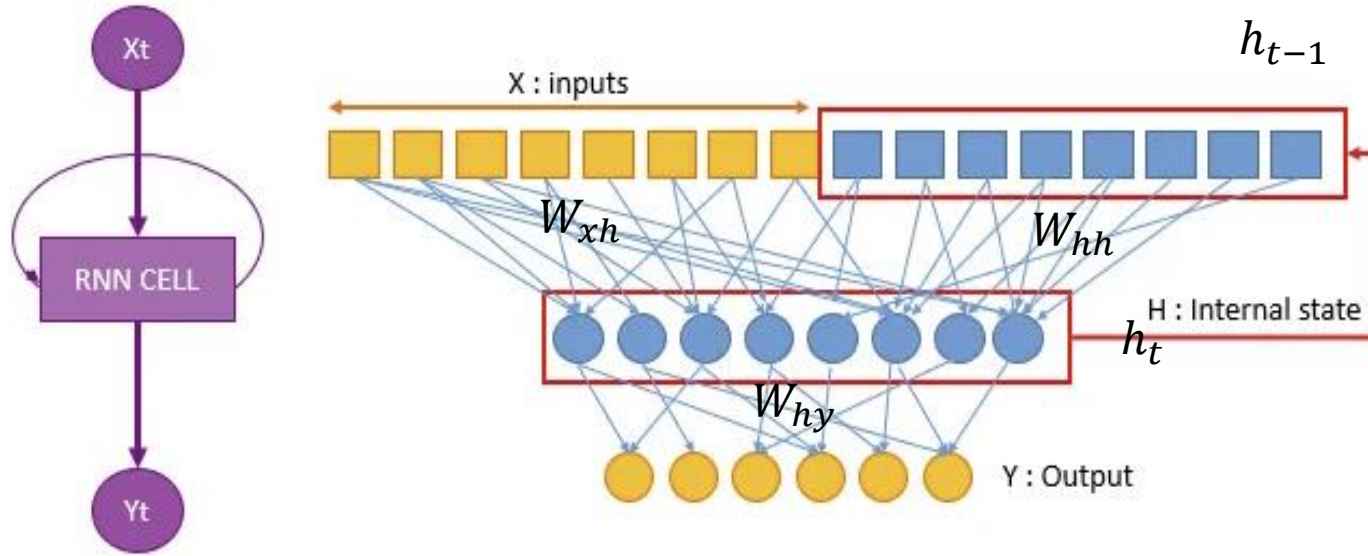
# Overview of common sequential data types

Time Series (Univariate)	0.1	0.4	0.4	0.6	0.5	0.9	
Time Series (Multivariate)	0.1 0.2 0.5	0.4 0.6 -0.1	0.0 -0.2 0.3	-0.4 0.7 -0.7	0.1 0.3 0.5	0.8 -0.6 0.8	
Text (characters)	h	e	l	l	o	-	
Text (words)	the	quick	brown	fox	jumps	over	
Text (Passage)	Chapter 1 The story begins with ...	Chapter 2 When I went to ...	Chapter 3 Every time he said ...	Chapter 4 Finally we arrive at ...			

DNA Sequence	A	T	T	G	C	G	
Chemical String	C	1	C	C	=	1	
Discrete clinical codes (Single Visit)	J9600 Acute respiratory failure	I509 Heart failure	I5020 Systolic heart failure	19301 Partial mastectomy	7052 Morphine		
Discrete clinical codes (Multiple Visits)	Visit 1 J9600 I509 ...	Visit 2 I509 7052 ...	Visit 3 I5020 J9600 ...	Visit 4 7052 1819 ...			
Video							

From: <https://arxiv.org/ftp/arxiv/papers/2004/2004.12524.pdf>

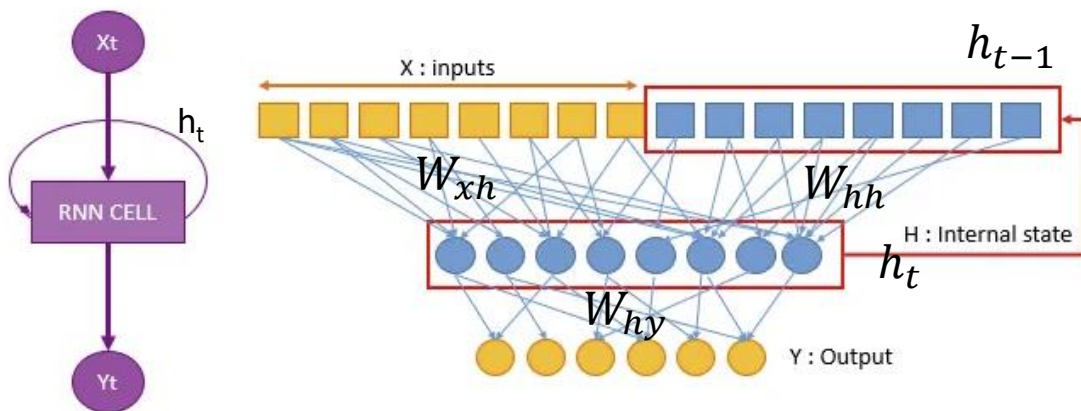
# Synthetic and extended architecture of a RNN cell



Note: we will use from now on matrixes to denote synaptic connections between layers in a synthetic way

$$\begin{aligned} W_{hh} &: h \times h \\ W_{xh} &: x \times h \\ W_{hy} &: h \times y \end{aligned}$$

The “state” of an RNN «cell» at time t is represented by a “hidden” vector  $\mathbf{h}_t$ :

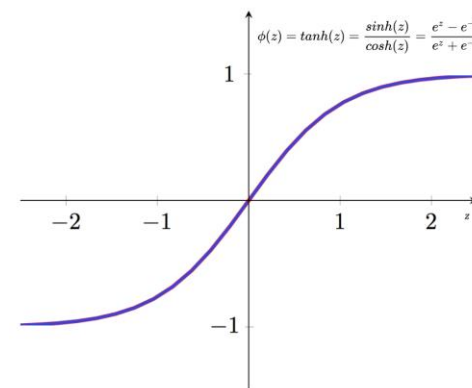


Hyperbolic tangent is used here as activation function

$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$





Note that we can process a *sequence of vectors*  $\mathbf{x}_t$  by applying a *recurrence formula* at every time step:

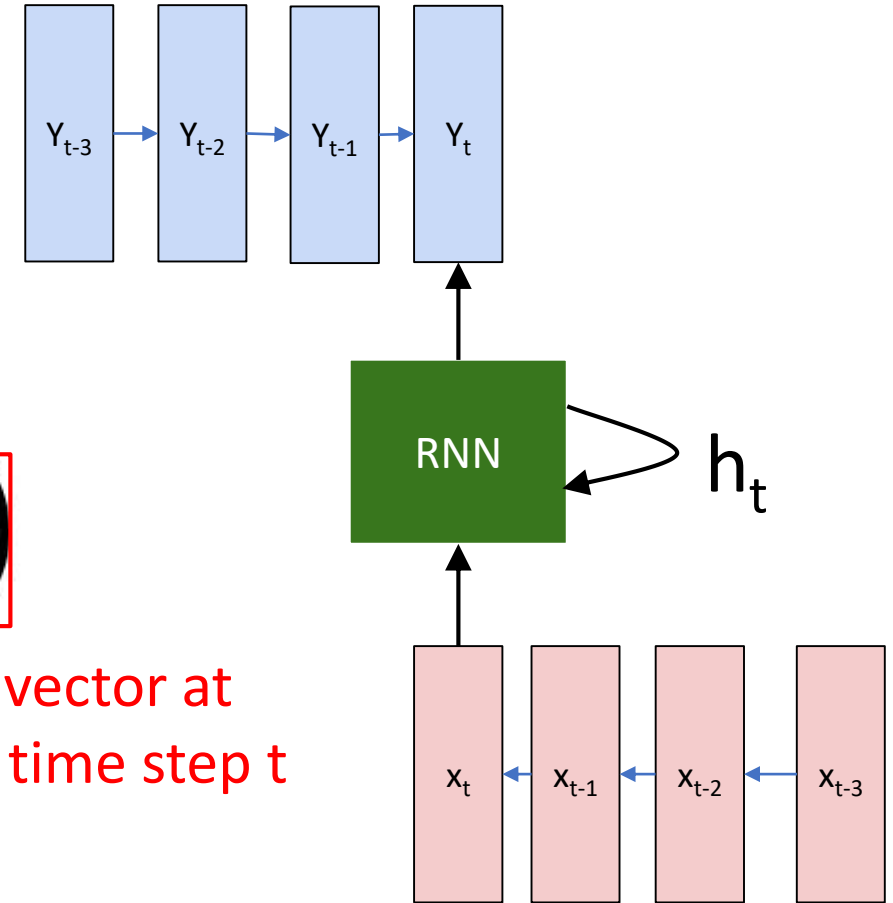
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new hidden state

some function with parameters  $W$  (e.g., tanh)

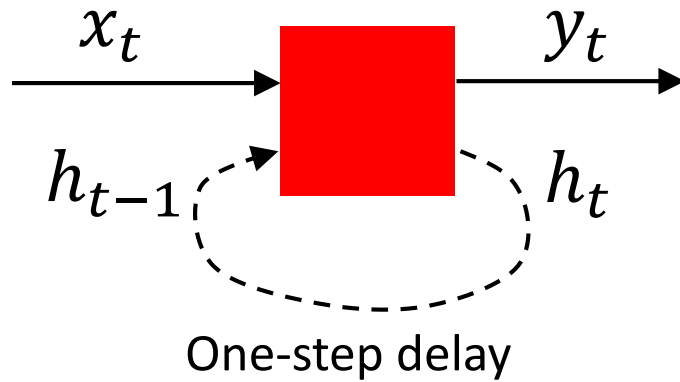
old hidden state

input vector at some time step  $t$



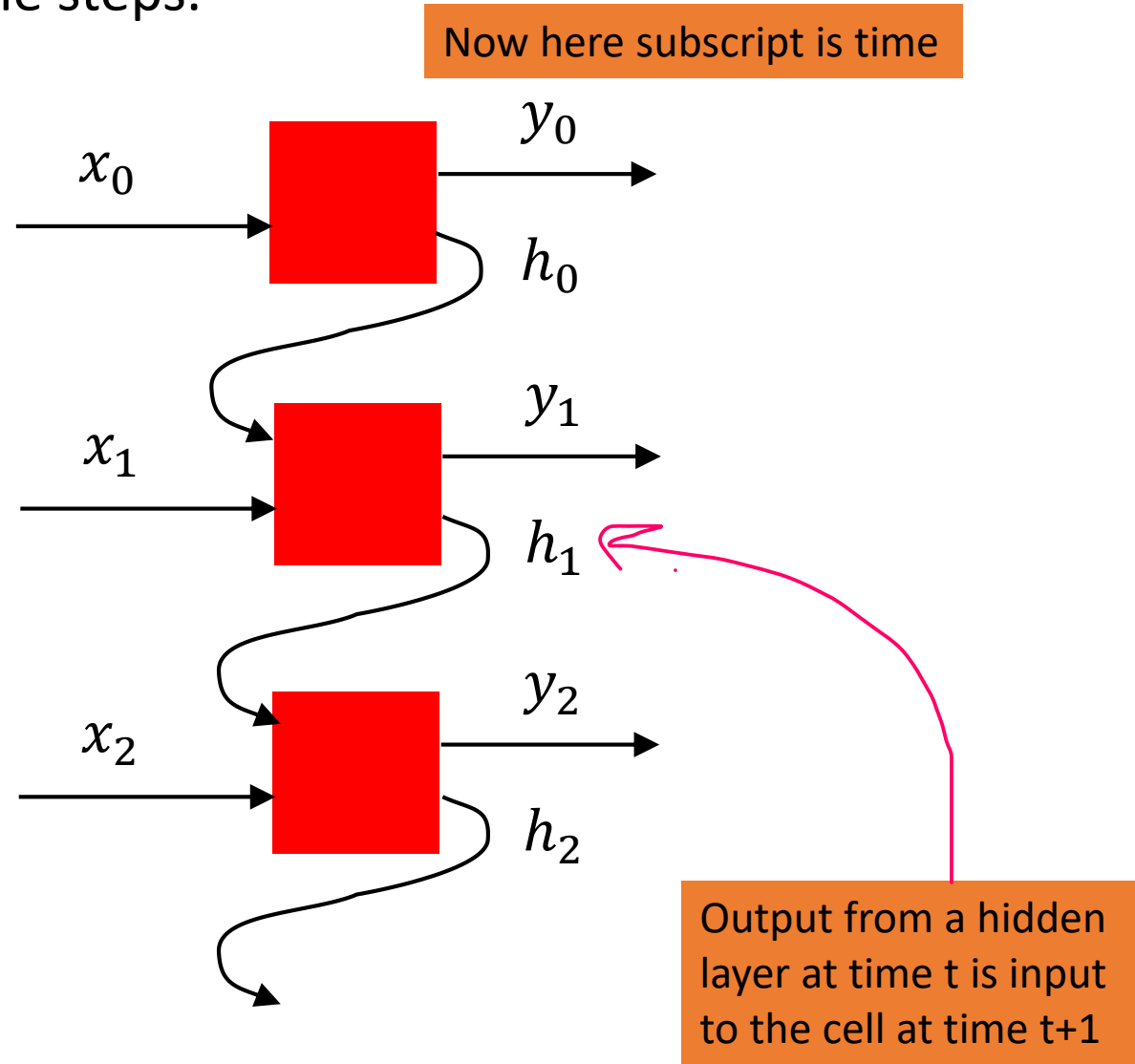
Notice: the same function and the same set of parameters (once learned) are used at every time step!!!!

RNN cells can be **unrolled** across multiple time steps.

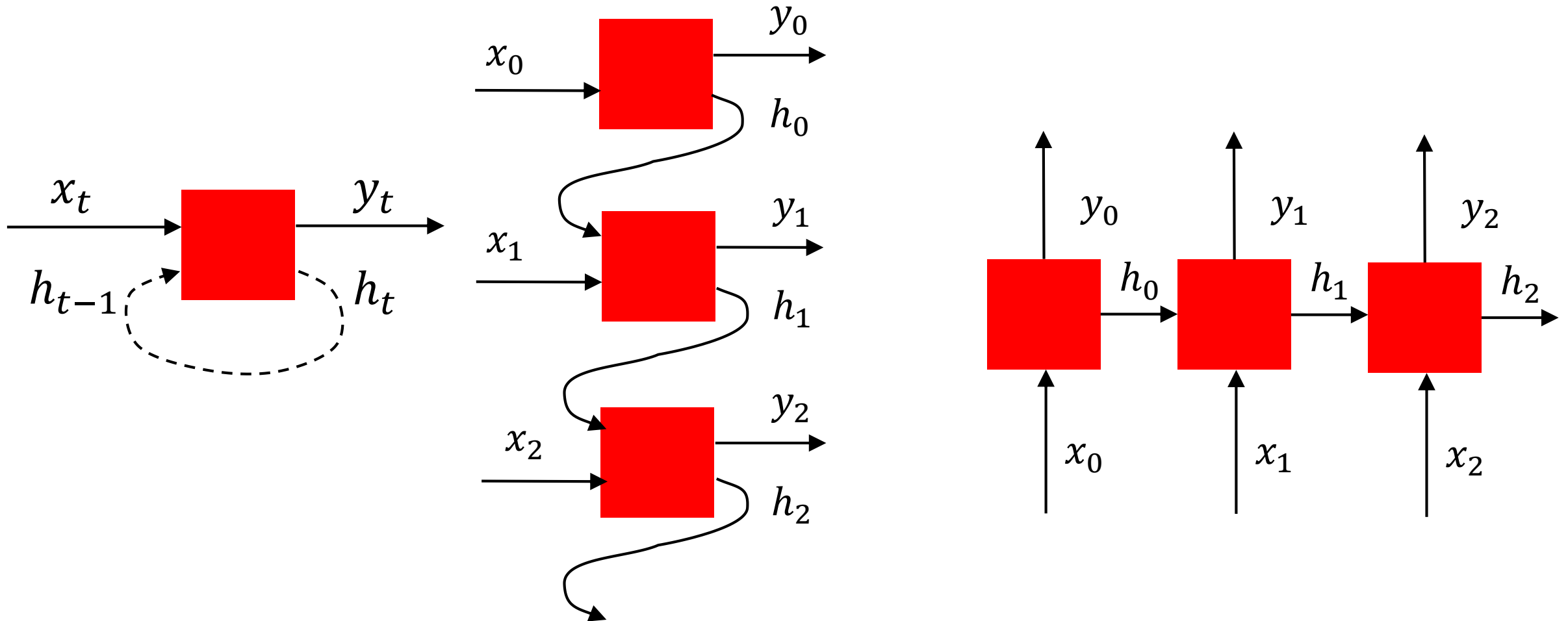


This produces a DAG which supports backpropagation.

But its size (number of cells) **depends on the input sequence length.**



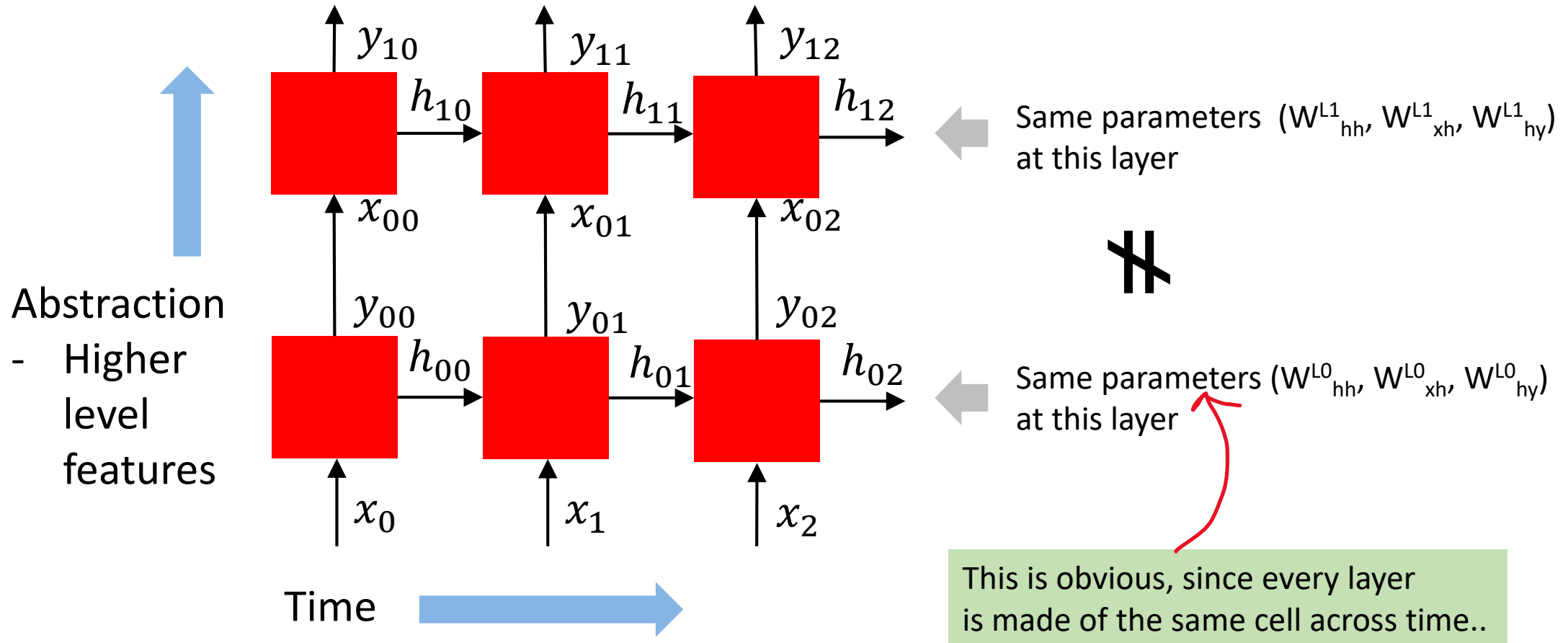
3 equivalent representations (the third is more often used)



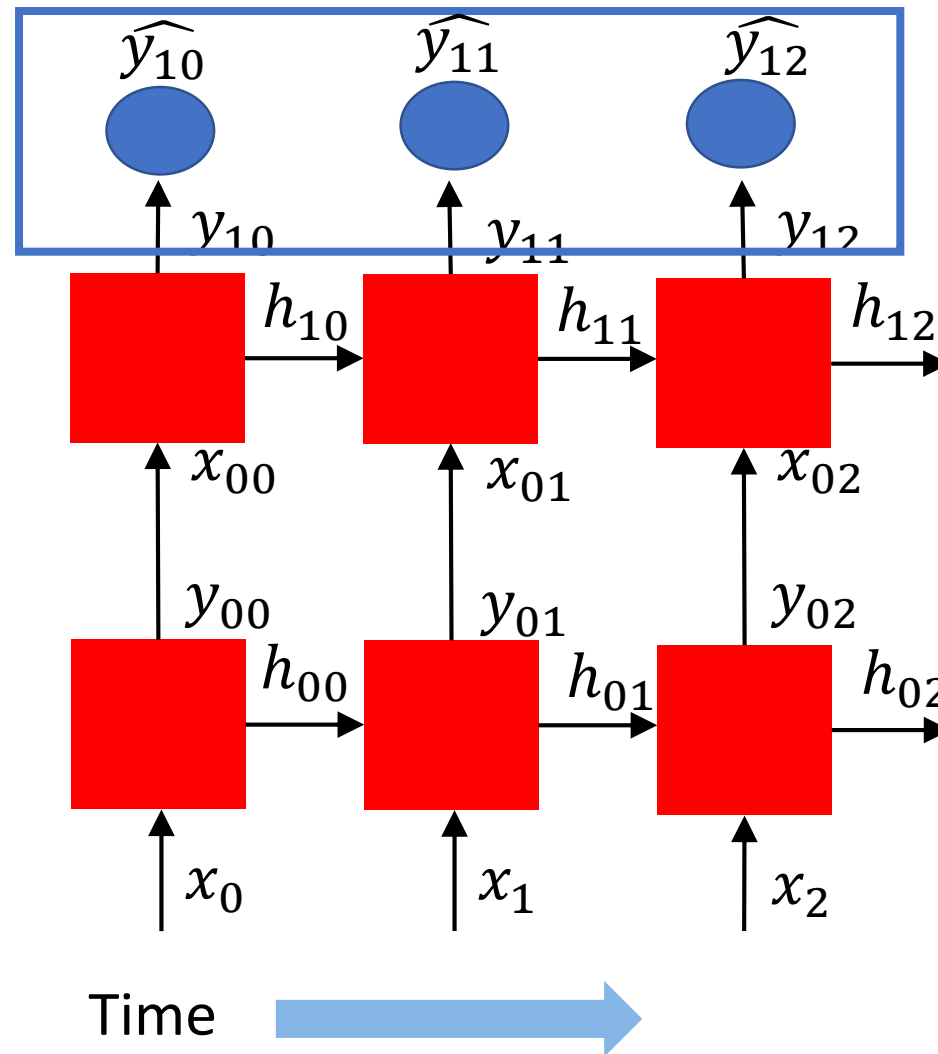
IMPORTANT!! Although we «unroll» the cell along the timeline, these are NOT different cells! There is a unique cell with parameters  $W_{xh}$   $W_{hh}$   $W_{hy}$

# Deep RNN

Often cells are stacked vertically (deep RNNs), and you can have many layers  $L$ :

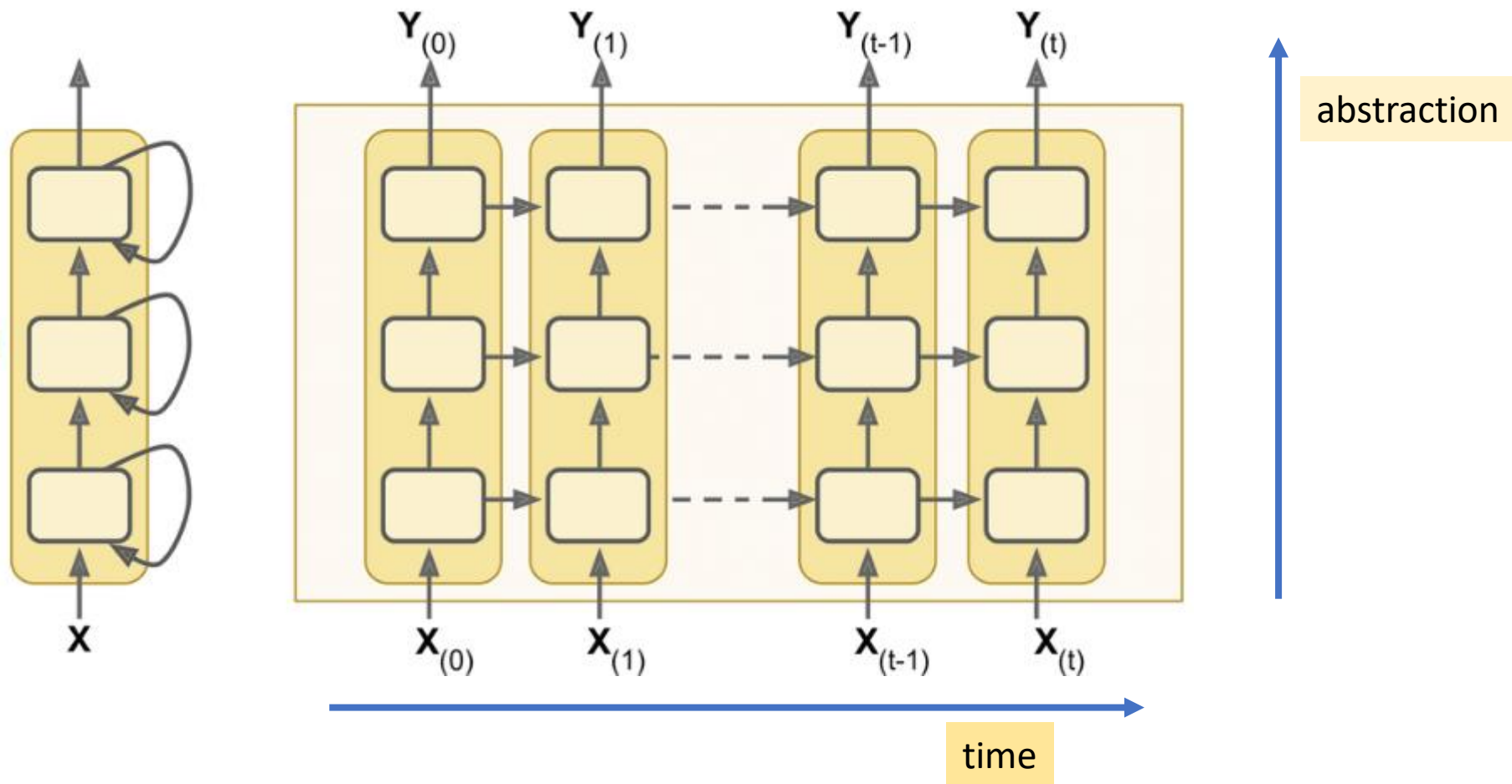


# The last layer computes some output function



Last layer can be a  
Softmax or other functions

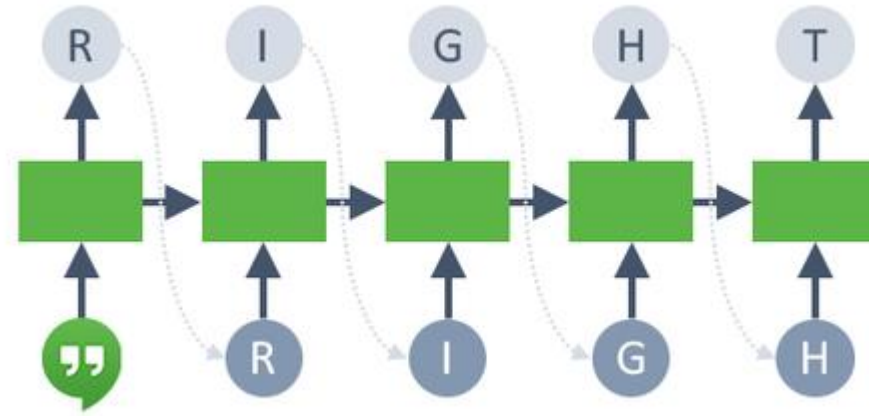
# Deep RNN



Example: learn predicting **next character** of a character string (word precompilation)

Vocabulary (set of character  
[r,i,g,h,d,o,t])

Example training  
sequence:  
**right, rigid, rigor..)**

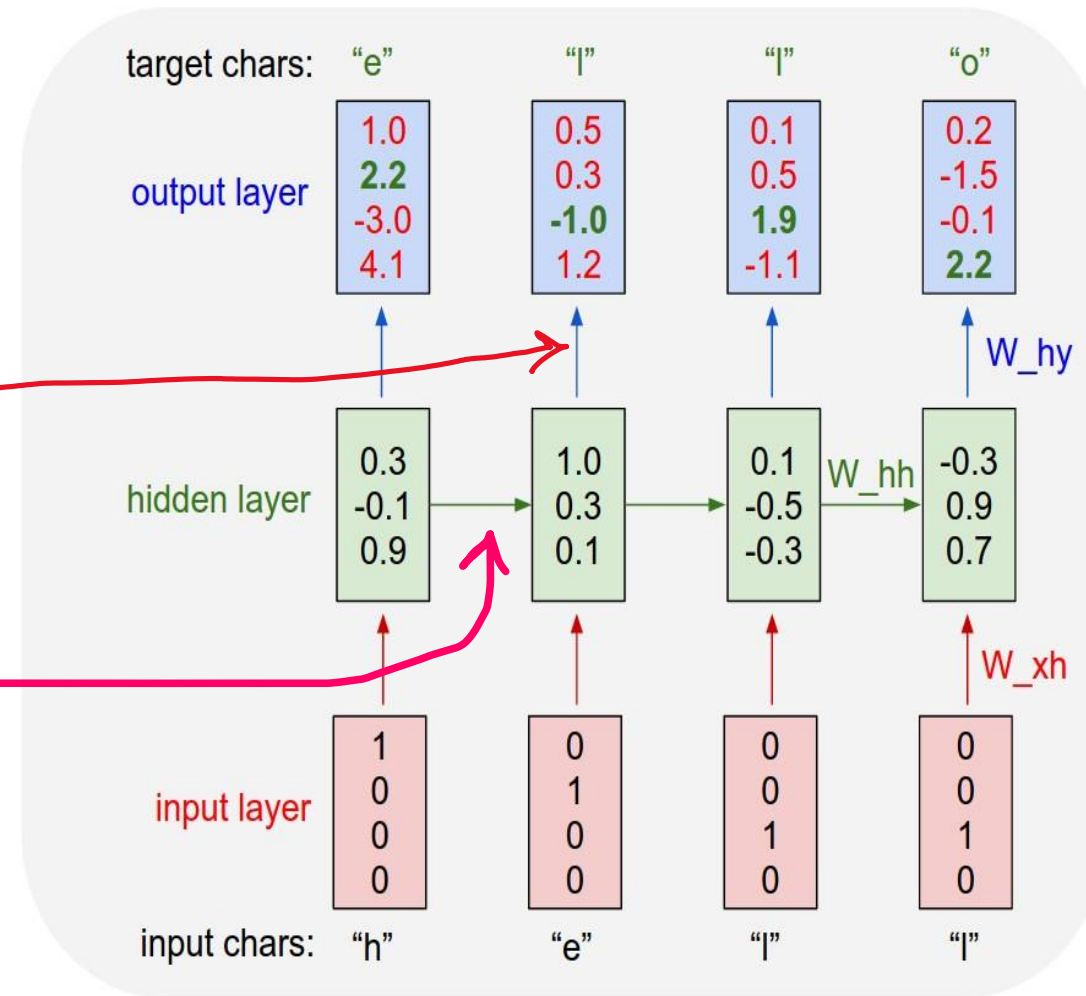


At each time-step  $t$ , Input is a character, output is the subsequent character

Example  
sequence:  
"hello"

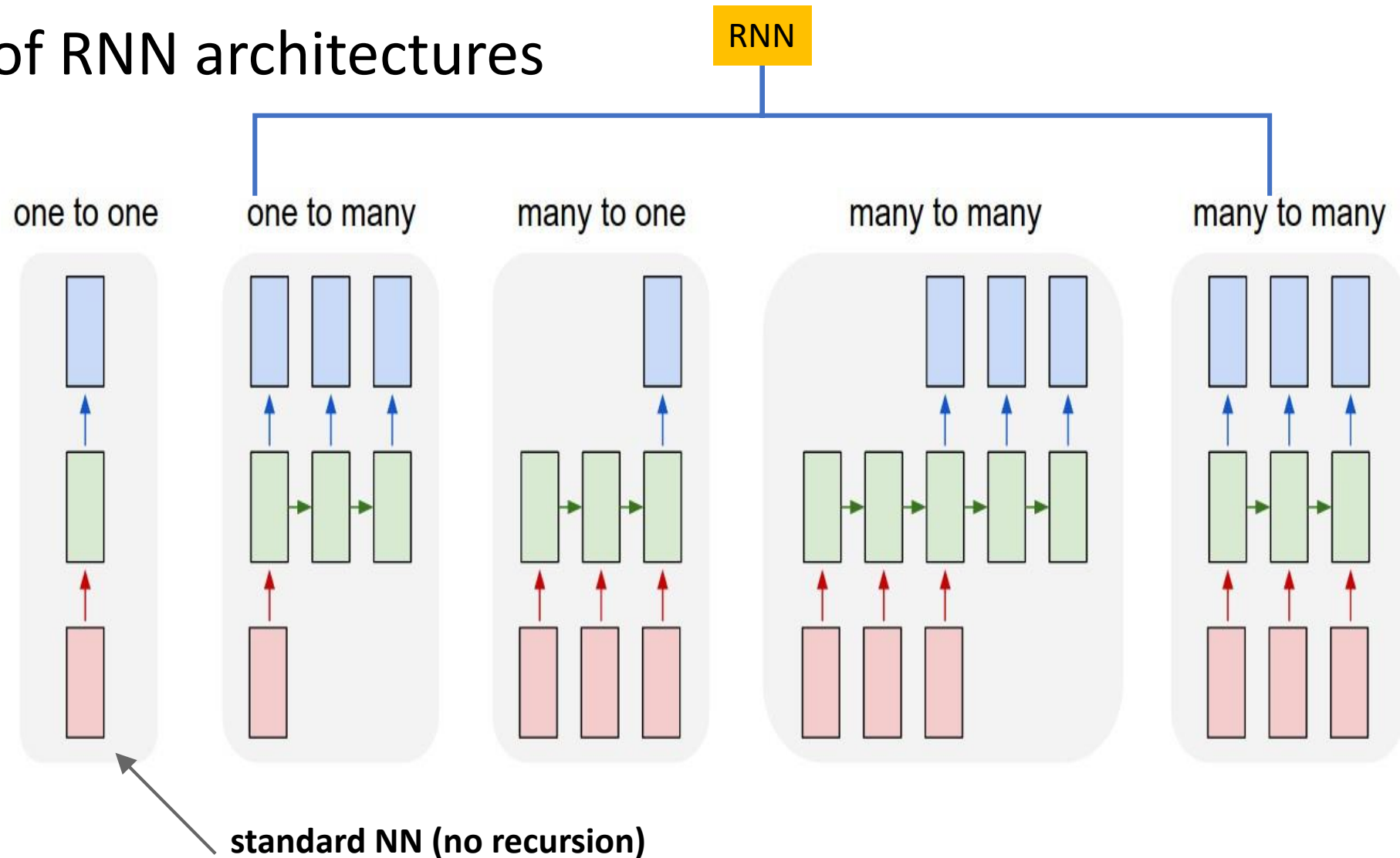
«l» is predicted based on «e»  
and on «h»

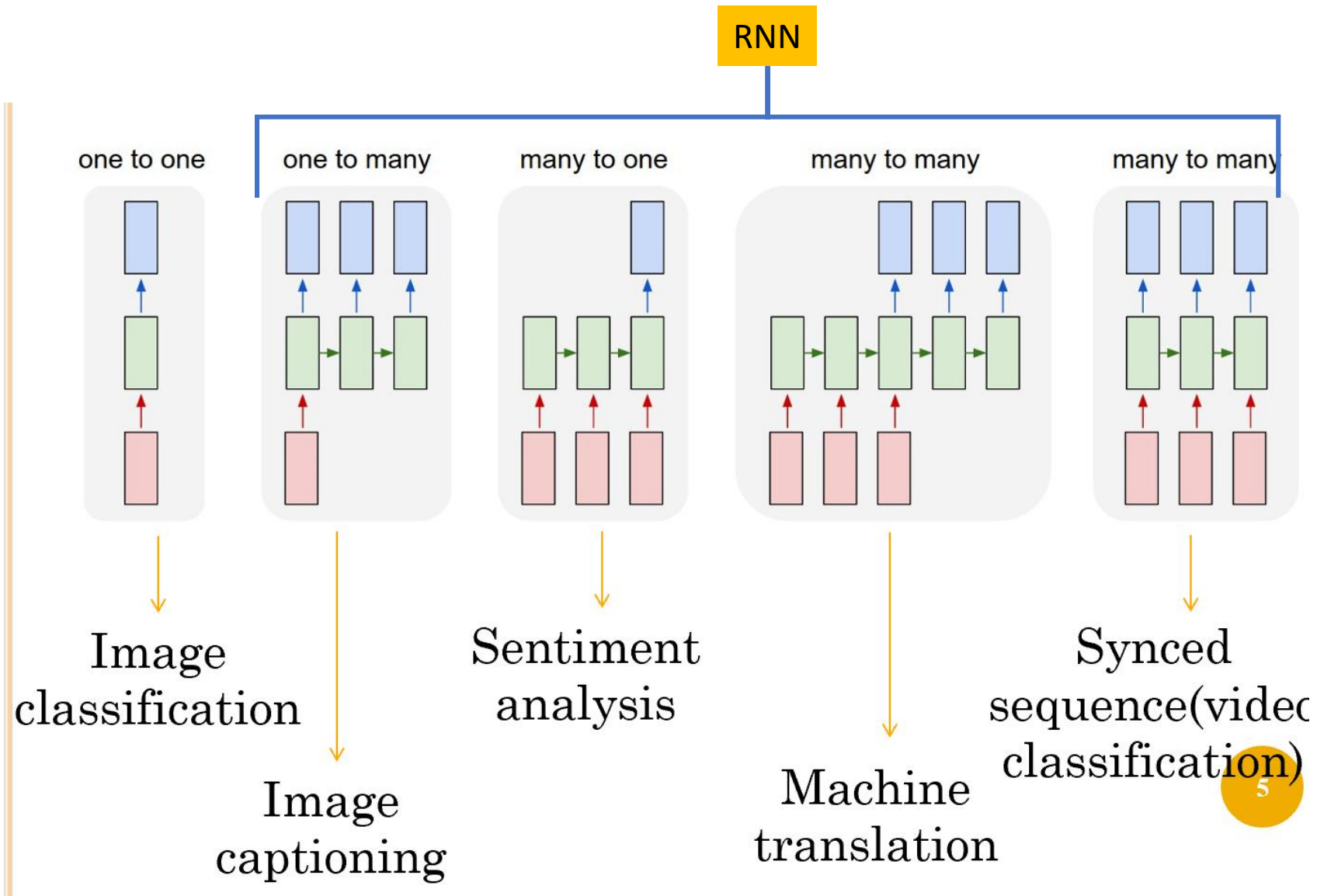
$h_{t-1}$  represents some  
«compressed» representation  
of previous character



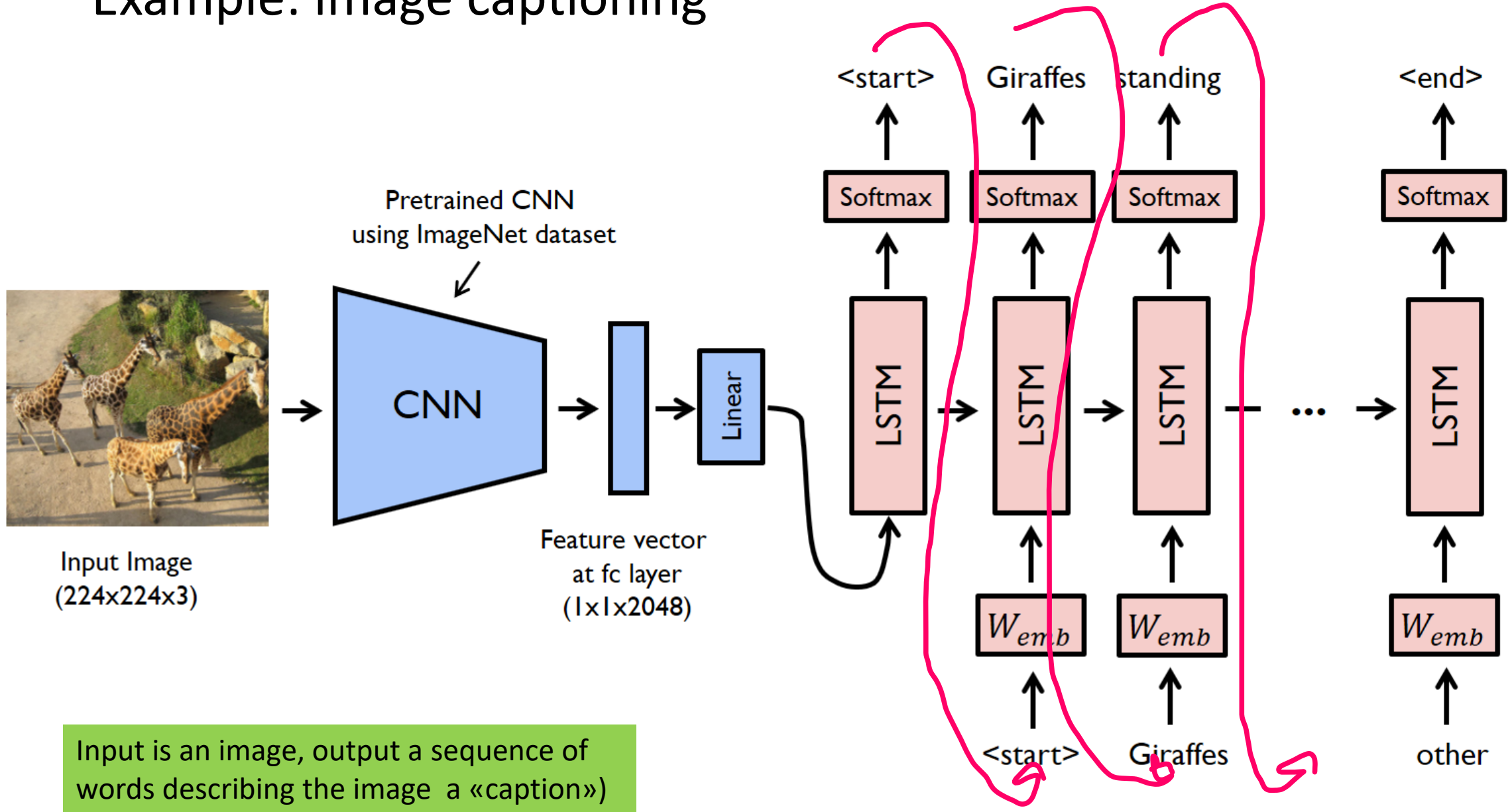


# Types of RNN architectures





# Example: image captioning



# Or text to image..

---

*«an astronaut riding a horse  
in a photorealistic style»*  
(<https://openai.com/dall-e-2/>  
)

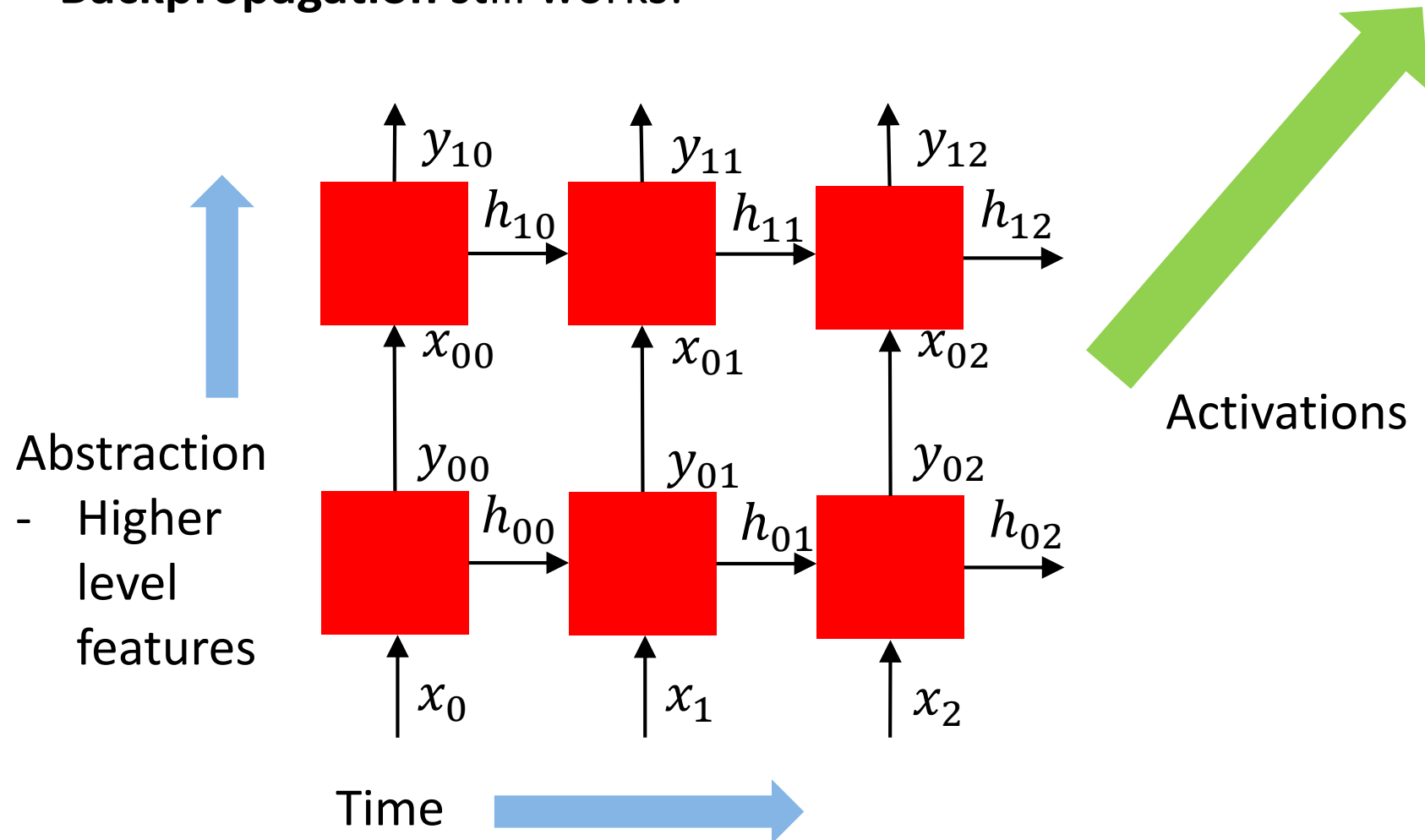


# What is the learning task?

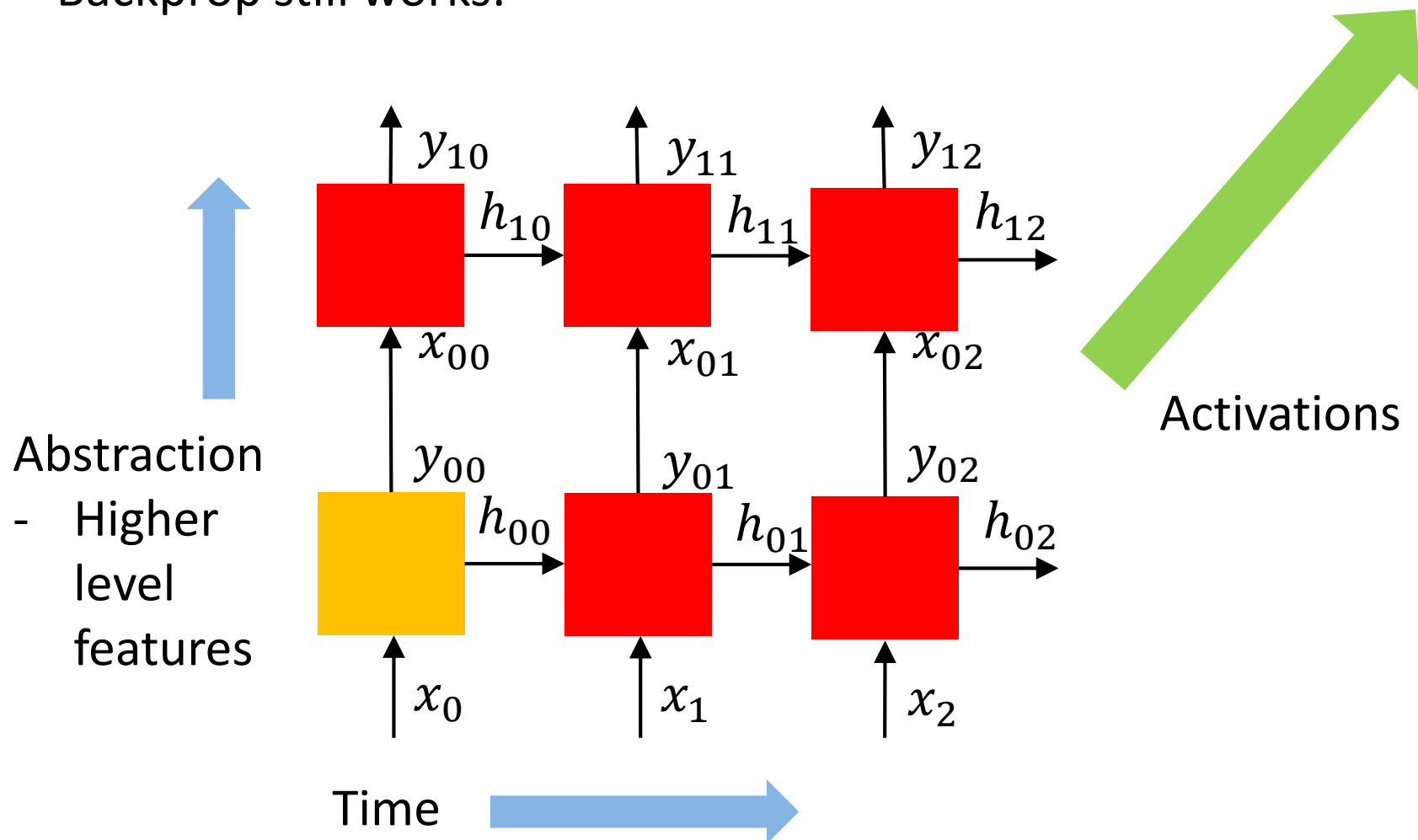
---

As usual, learning the model parameters (the 3 matrixes  $W$  in any layer), like for «standard» CNN

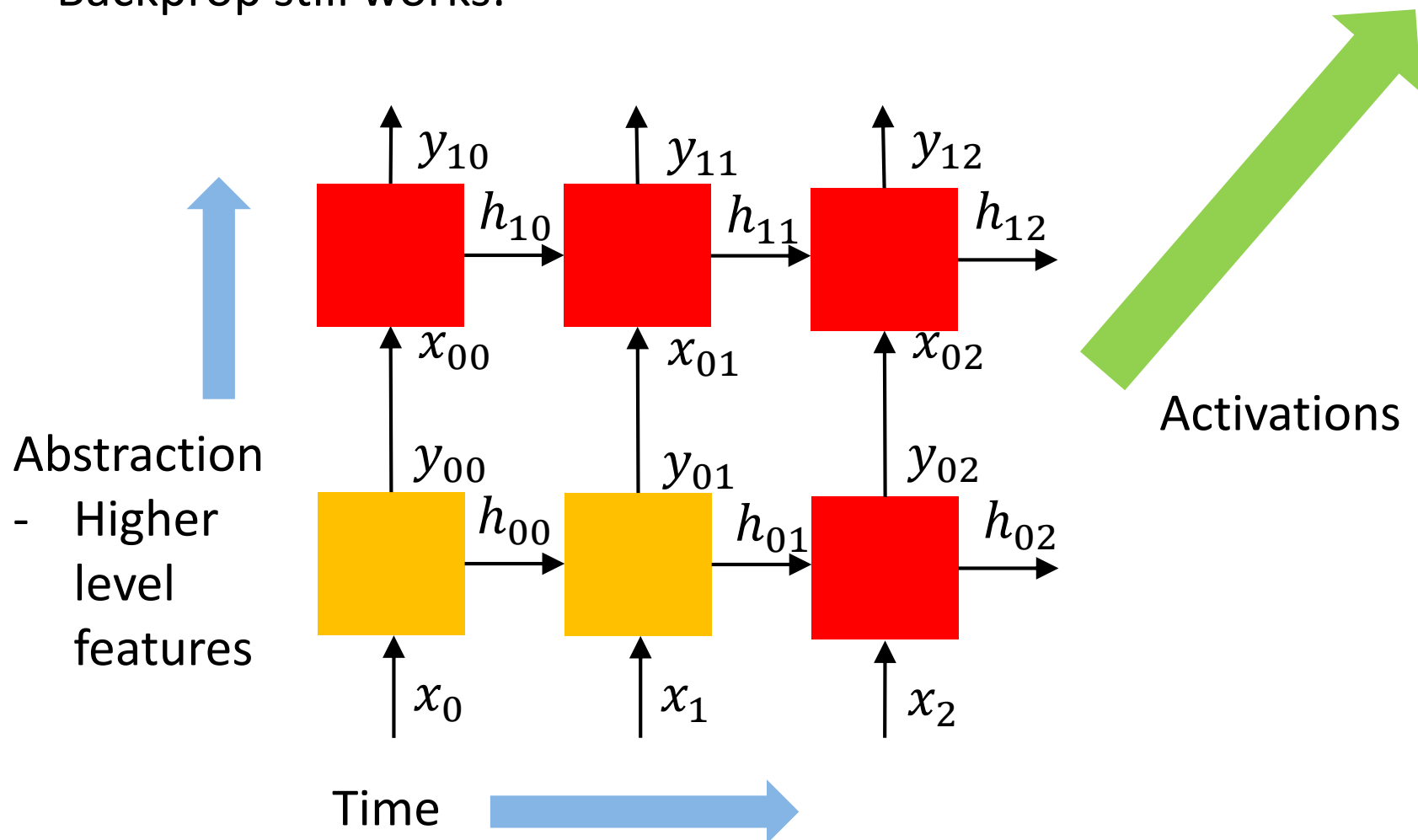
**Backpropagation** still works:



Backprop still works:

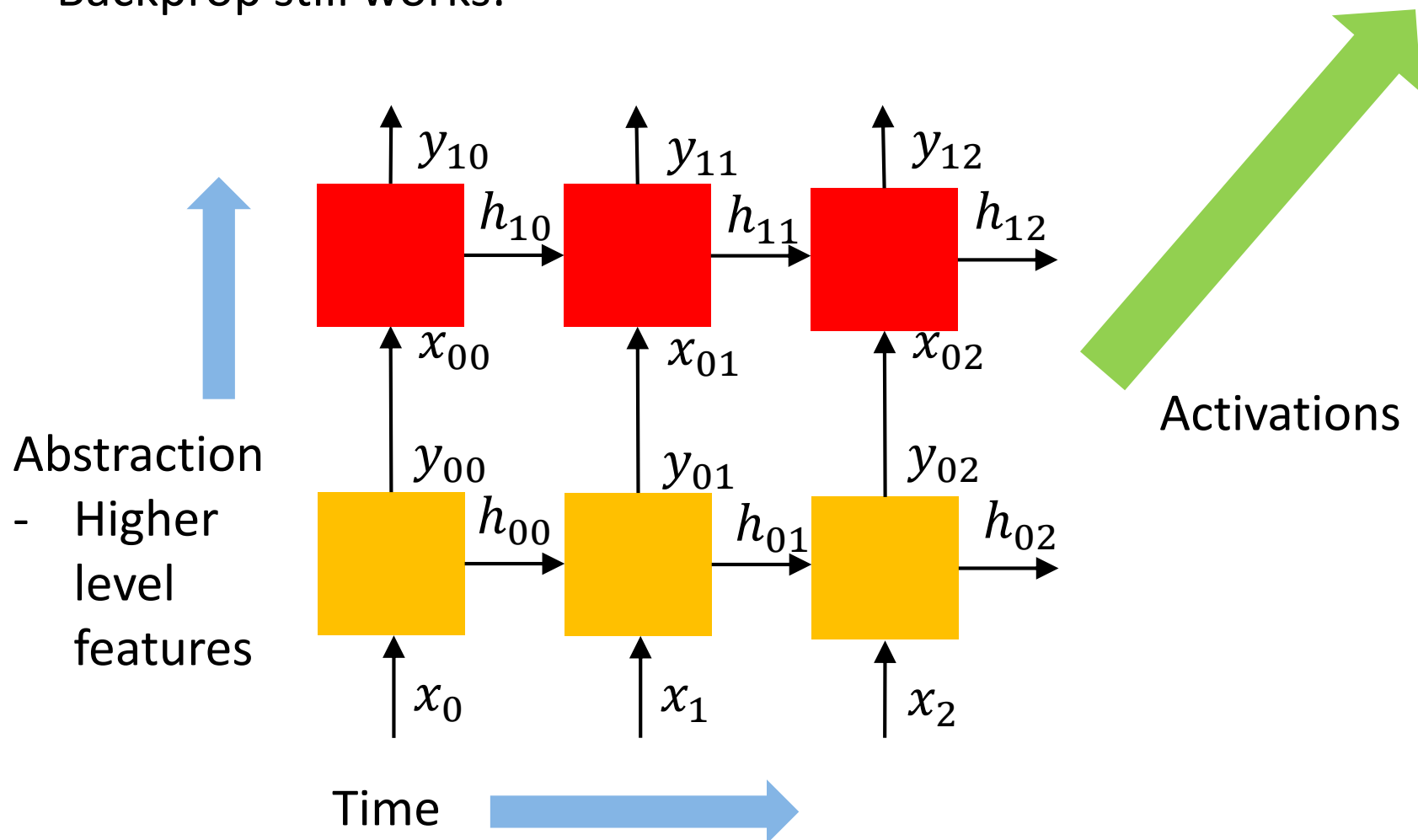


Backprop still works:

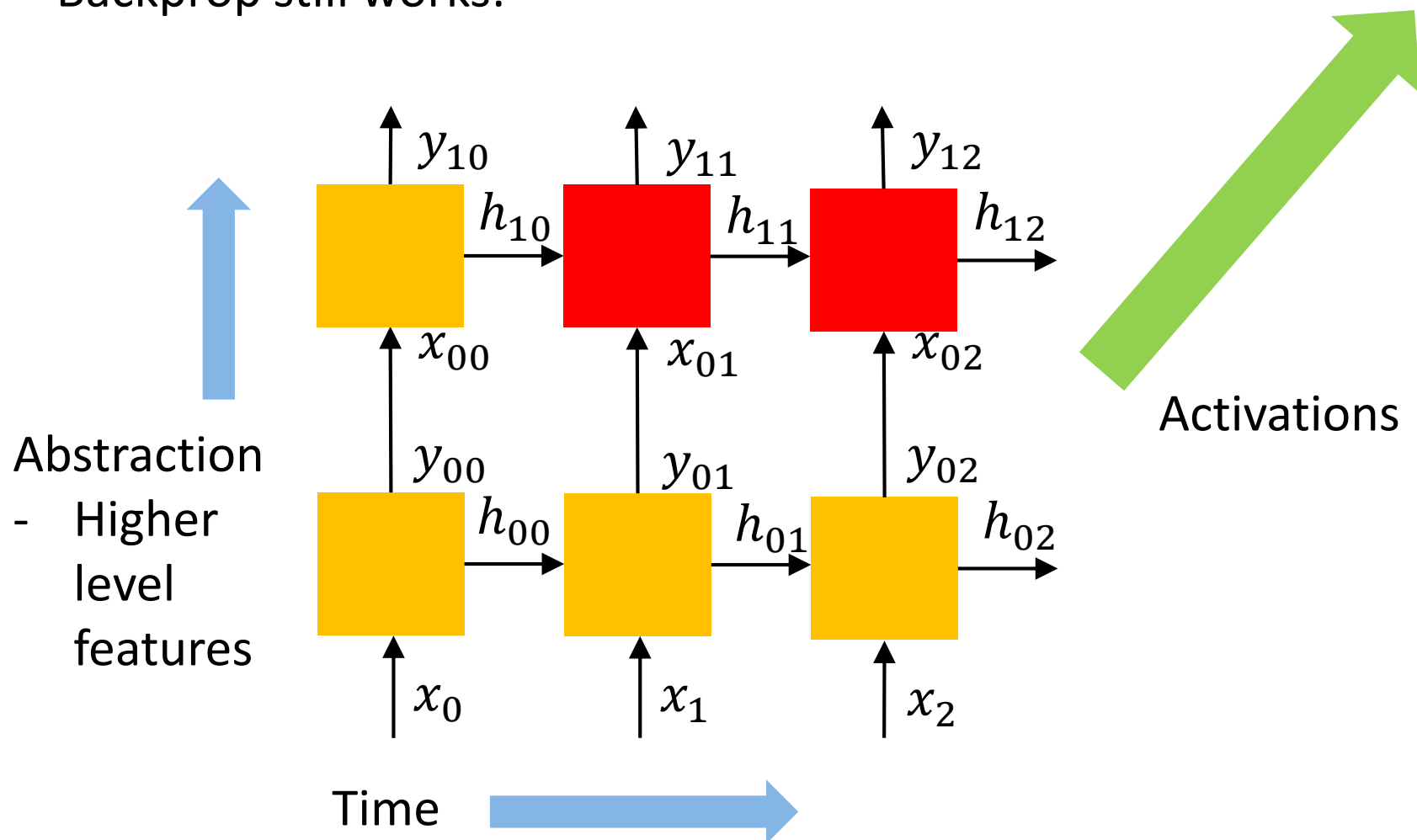




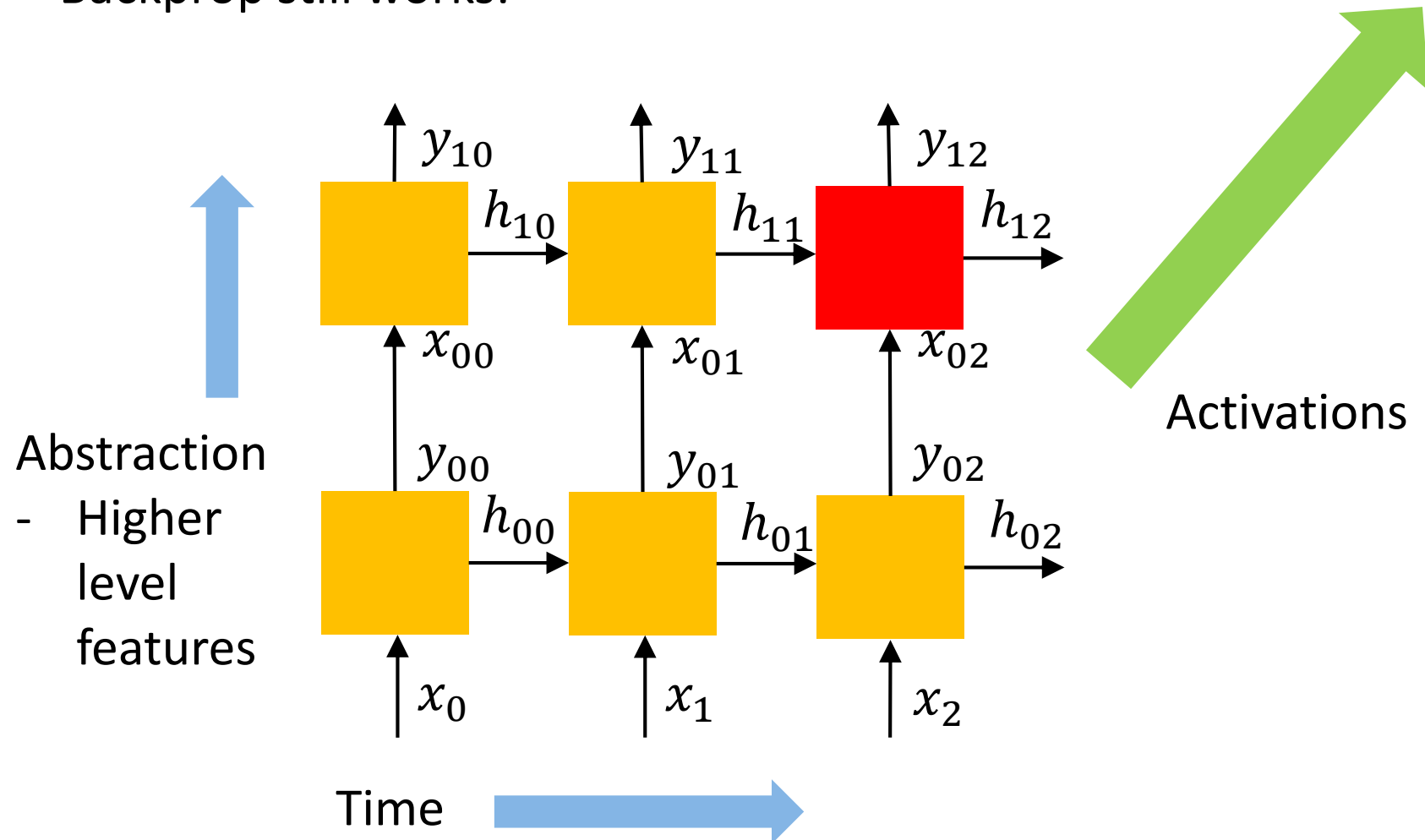
Backprop still works:



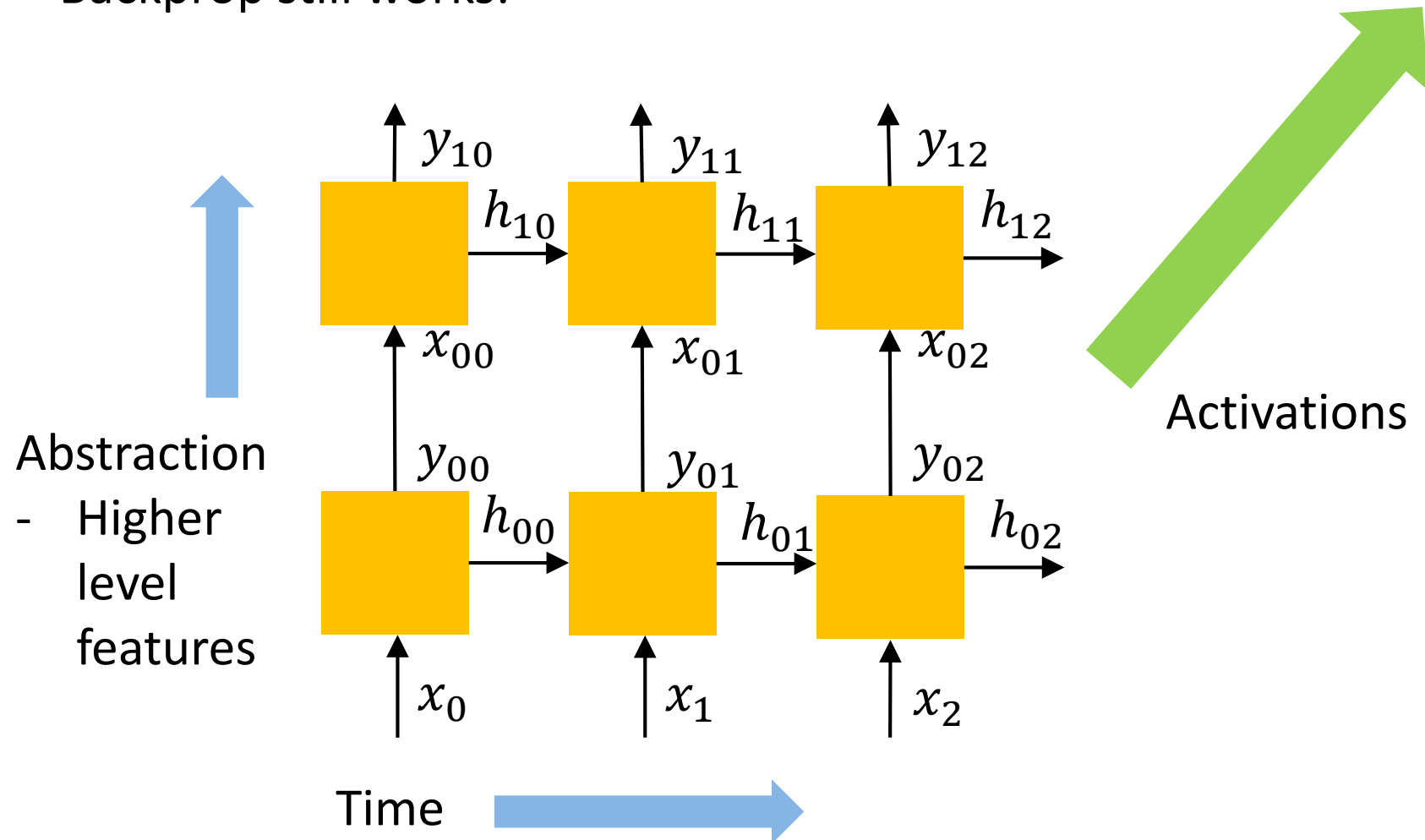
Backprop still works:



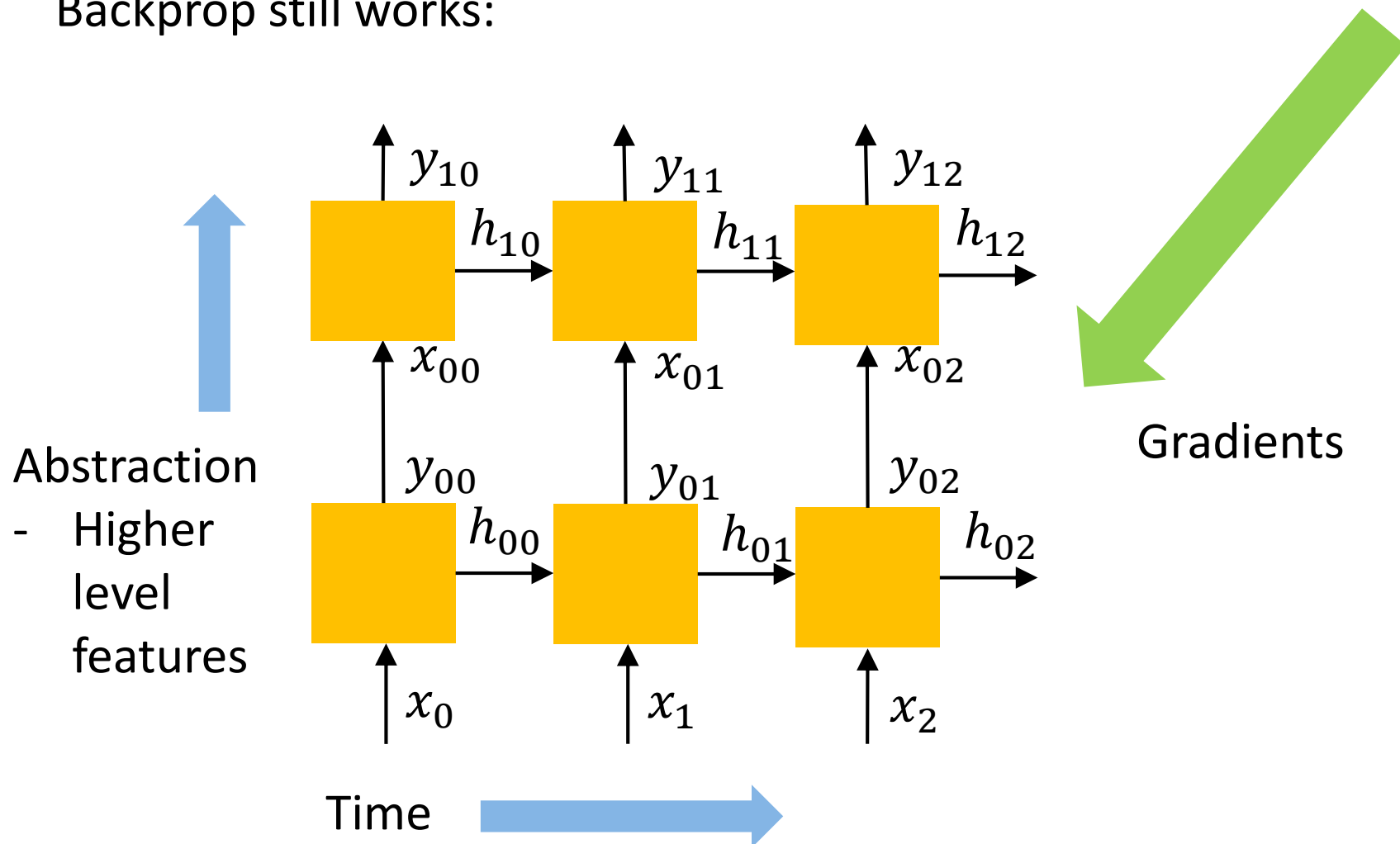
Backprop still works:



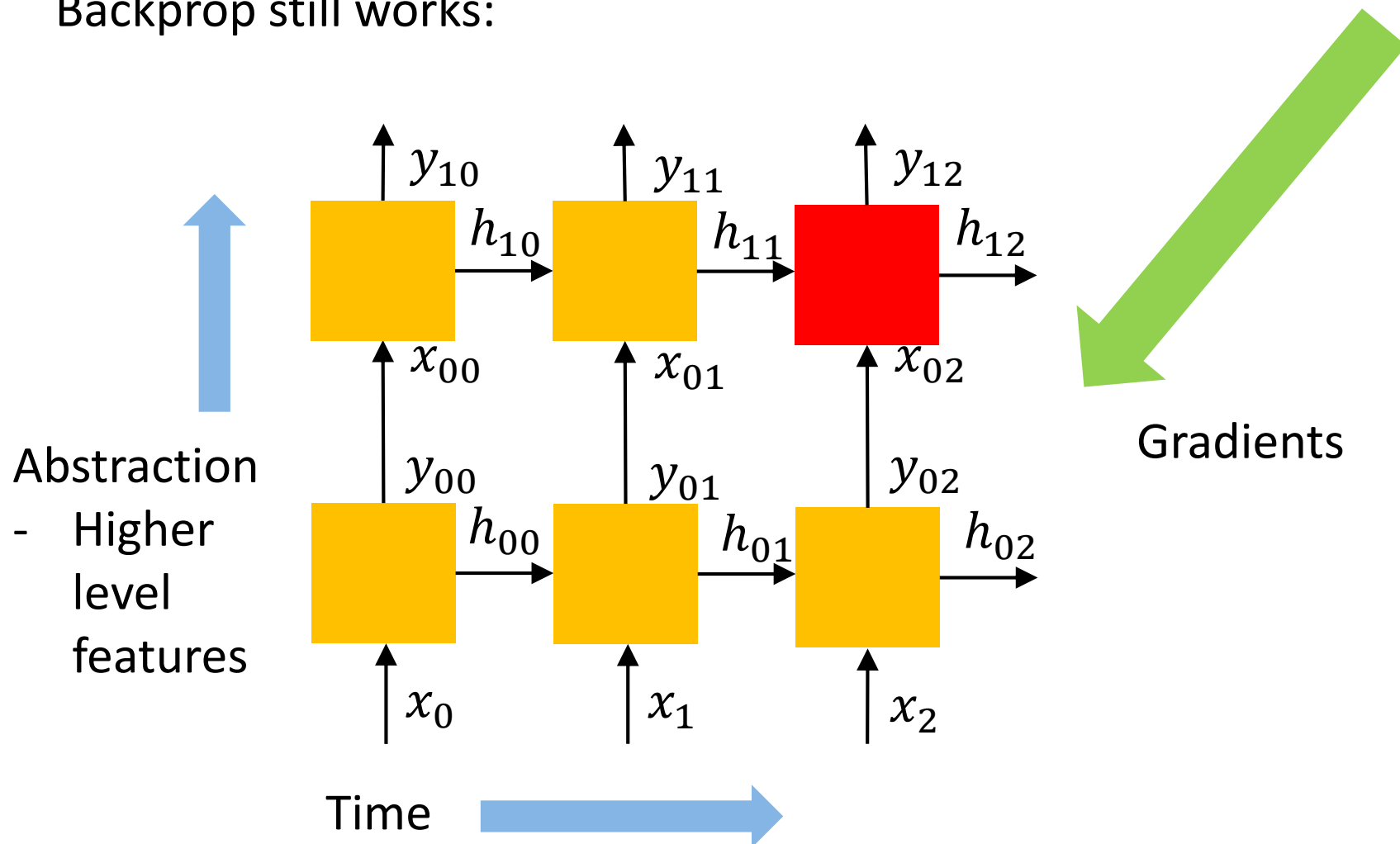
Backprop still works:



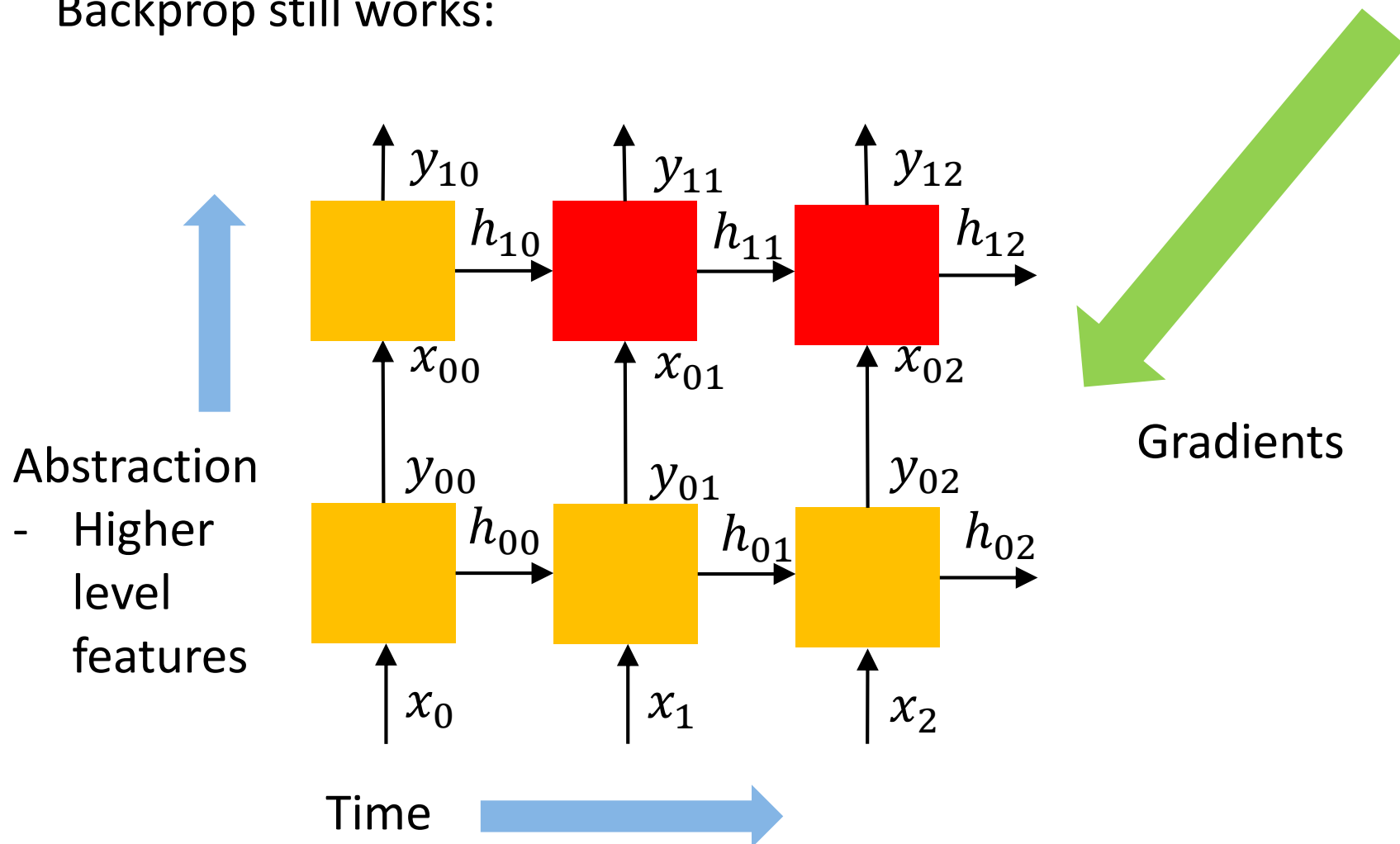
Backprop still works:



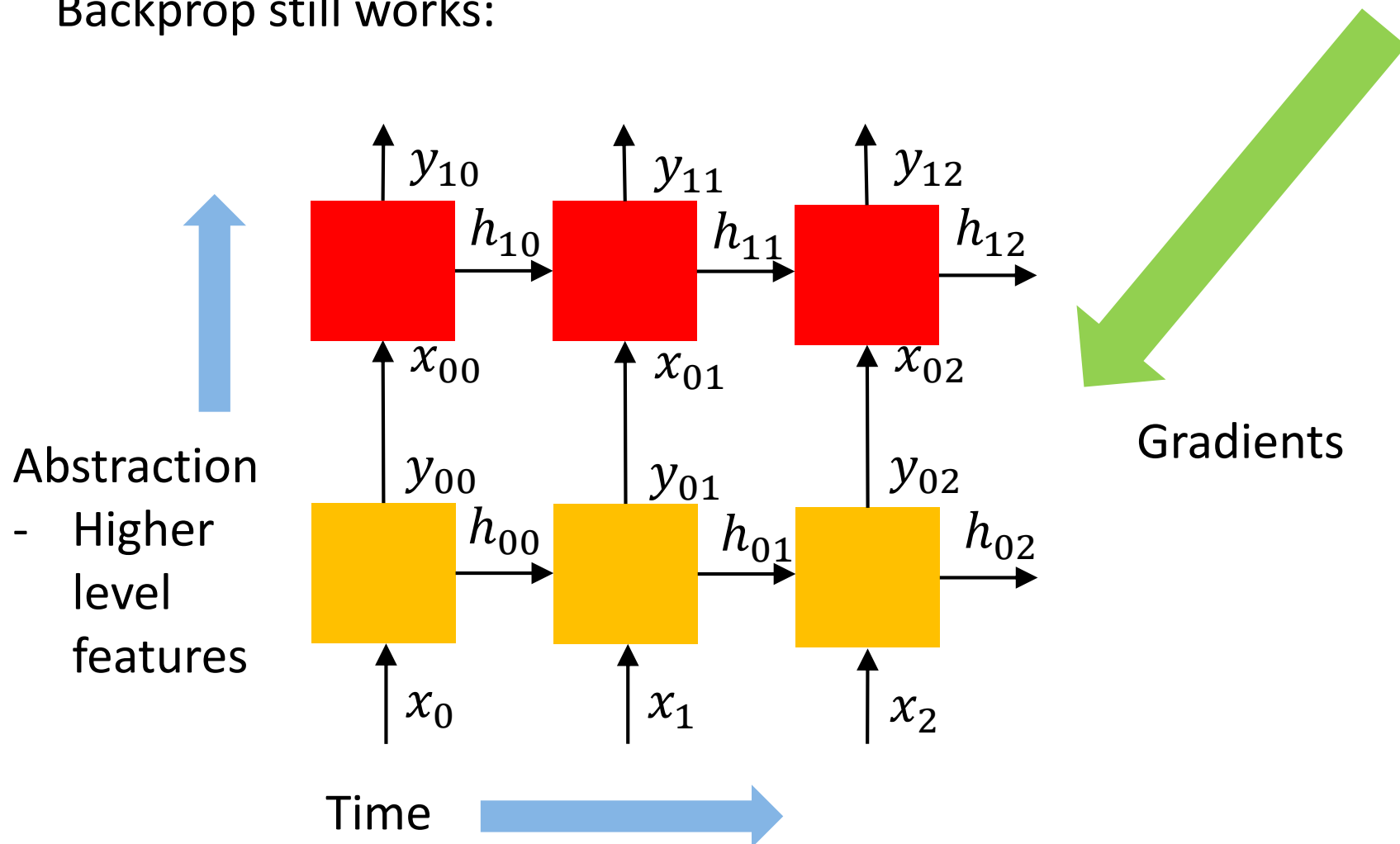
Backprop still works:



Backprop still works:

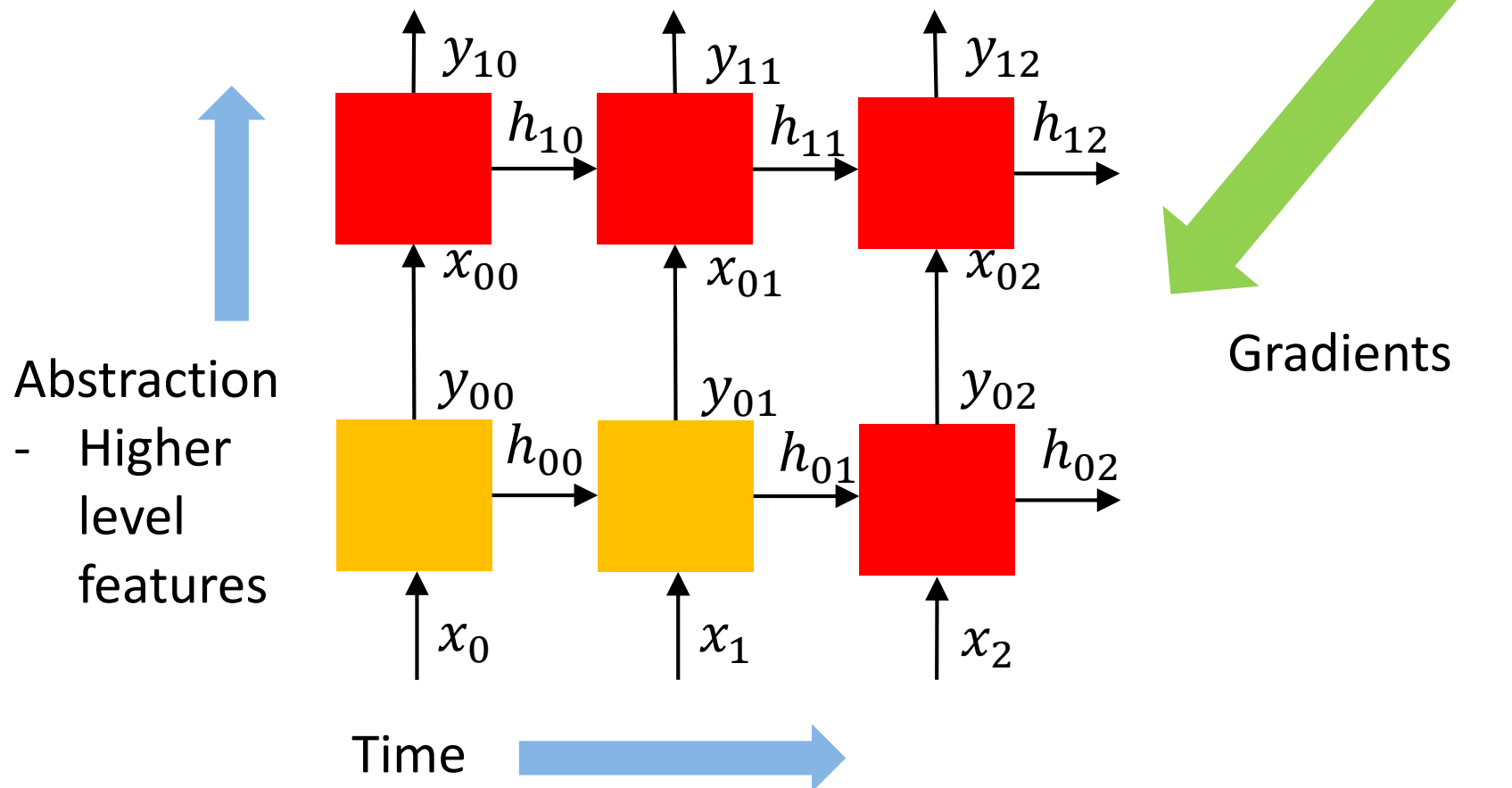


Backprop still works:

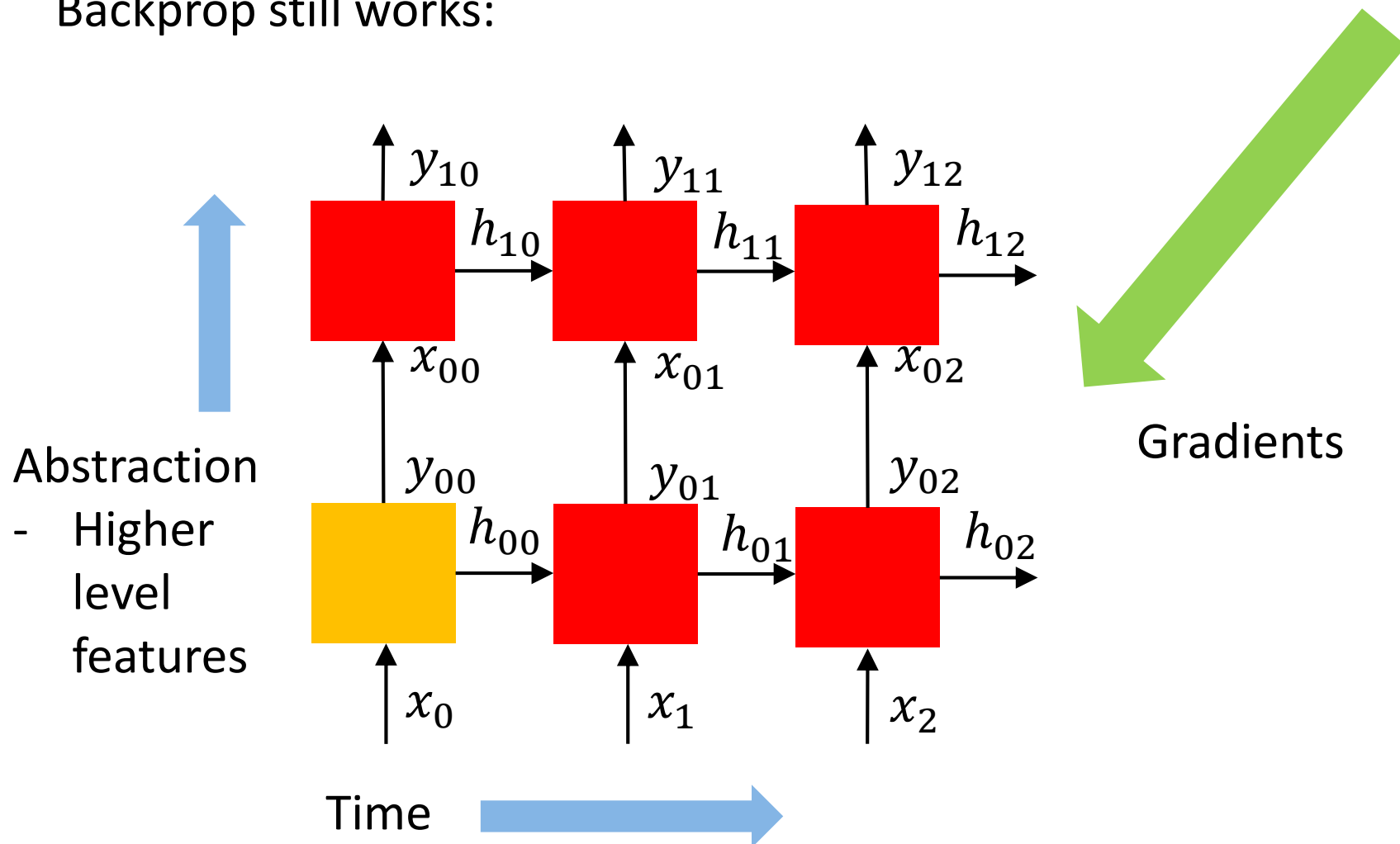




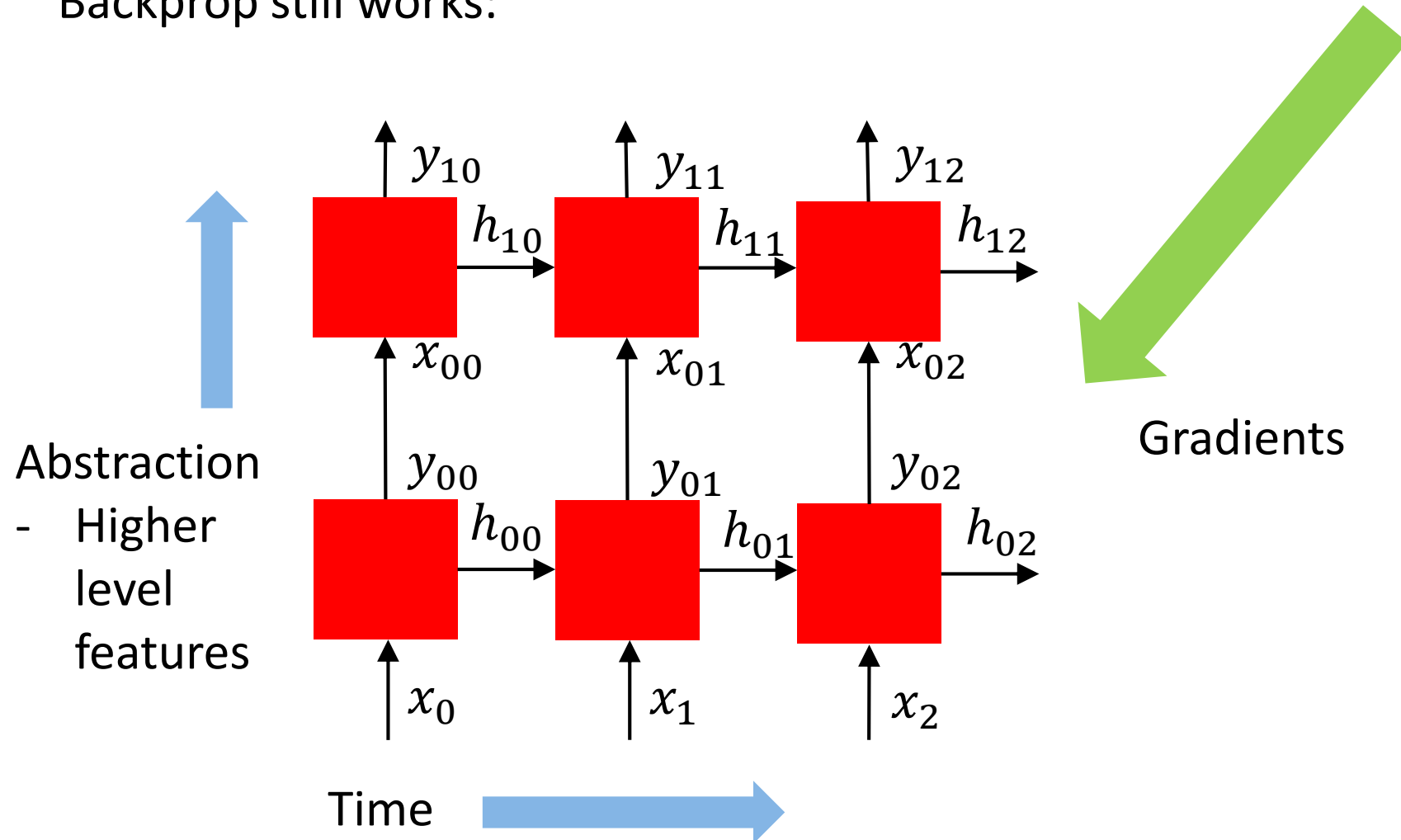
Backprop still works:



Backprop still works:



Backprop still works:



# Learning parameters in a RNN

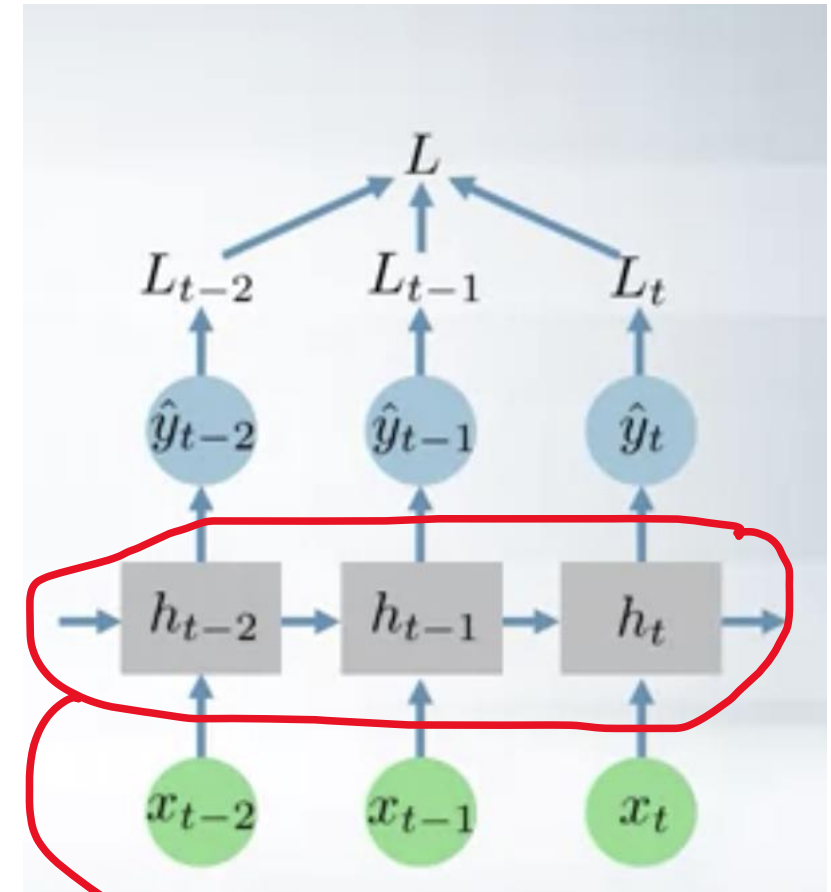
- What is the total loss for this network?
- How do we update the weights  $W_{hh}$ ,  $W_{xh}$  and  $W_{hy}$ ?

# Backpropagation in RNN

As usual we define «some» Loss function  $L$ , for example, cross entropy loss (as for CNN), or

$$L(y^{TRUE}, \hat{y}) = \sum_t L_t(y_t^{TRUE}, \hat{y}_t)$$

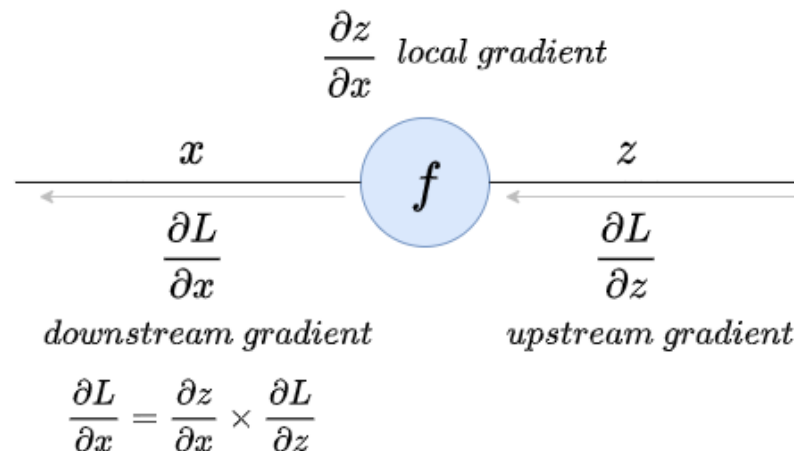
Note that here the **error is defined over the entire sequence**, so that the total error is the sum of losses at each time step



Note here we can have multiple layers

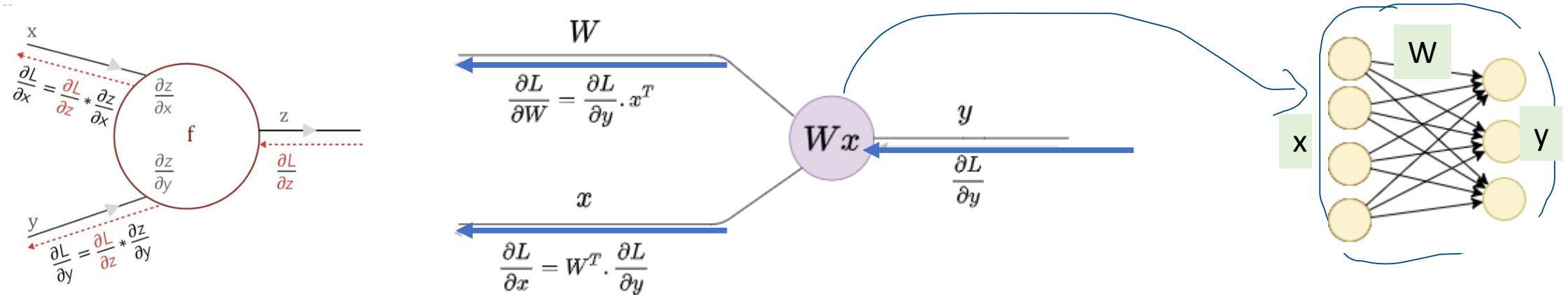
# Always with computational graphs!

- We can consider separately every node (or layer of nodes) of the graph, and analyze the gradient flow at the node level
- Remember: on each node we have 3 gradients: the *upstream* (backpropagating from pervious layer), the *local* gradient, computed on the current node, and the *downstream* gradient, that propagates backwards
- The upstream is known, the local is computed applying tthe derivative on  $f(x)$  (whatever function  $f(x)$  is), and the downstream gradient is computed applying the chain rule.



# A more compact notation: Matrix multiplication gradient

We usually deal with high dimensional inputs and outputs ( $\mathbf{x}$ ) which are represented as vectors ( $\mathbf{x}$ ) and matrices ( $W$ ). The derivative of a scalar (the matrix) wrt a vector ( $\mathbf{x}$ ) is a vector that represents how the scalar is affected by a change in each element of the vector; the derivative of a vector wrt another vector is a matrix that represents how each element of the vector is affected by a change in each element of the other vector.



Here  $W$  is the weight matrix,  $\mathbf{x}$  is the input vector, and  $\mathbf{y}$  is the output product vector,  $T$  is the transpose (of  $W$  or  $\mathbf{x}$ ). The purple node  $W\mathbf{x}$  represents an entire layer of neurons that receive the input vector  $\mathbf{x}$  and compute the convolution.

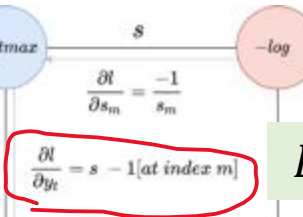
# Now, let's consider the full RNN graph

Forward pass

$$s = \text{softmax}(y_t) = \frac{e^{y_t}}{\sum y^k}$$

The softmax backprop is same as for CNN

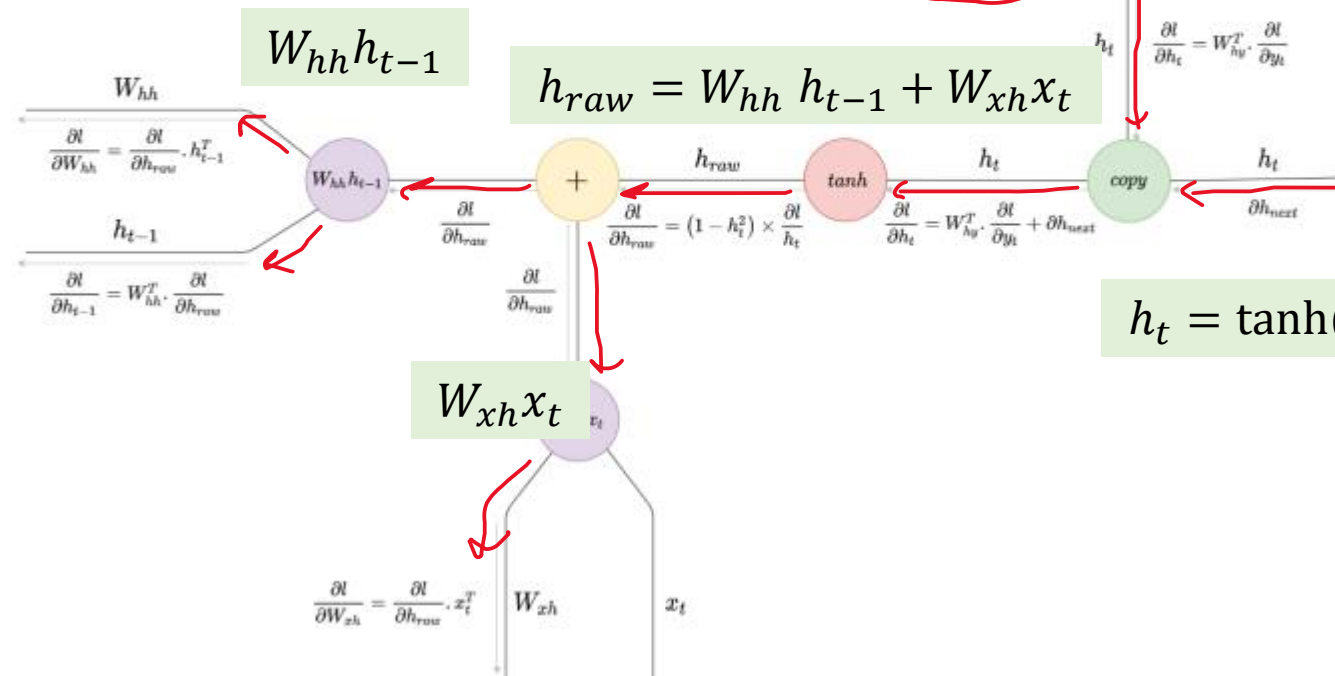
$$y_t = W_{hy} h_t$$



$$L = -\log(s)$$

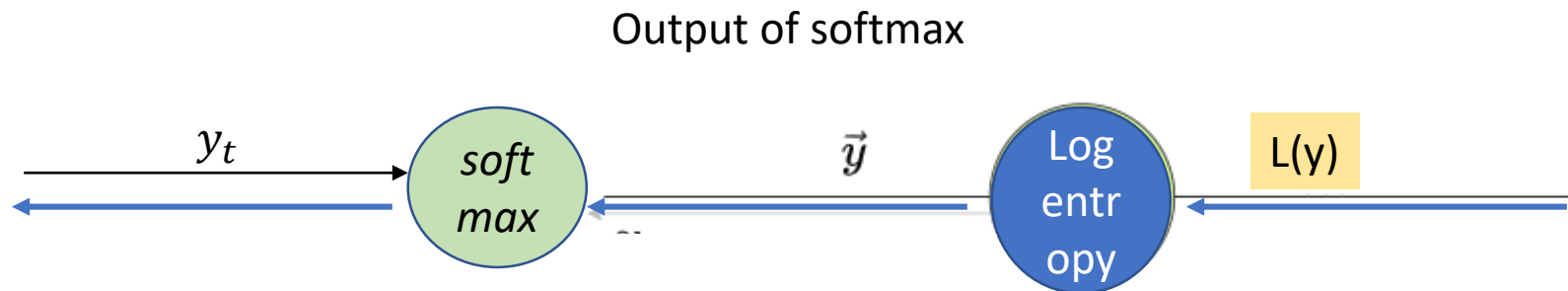
We use the Logloss

Same as for CNN, we use matrix&vector form



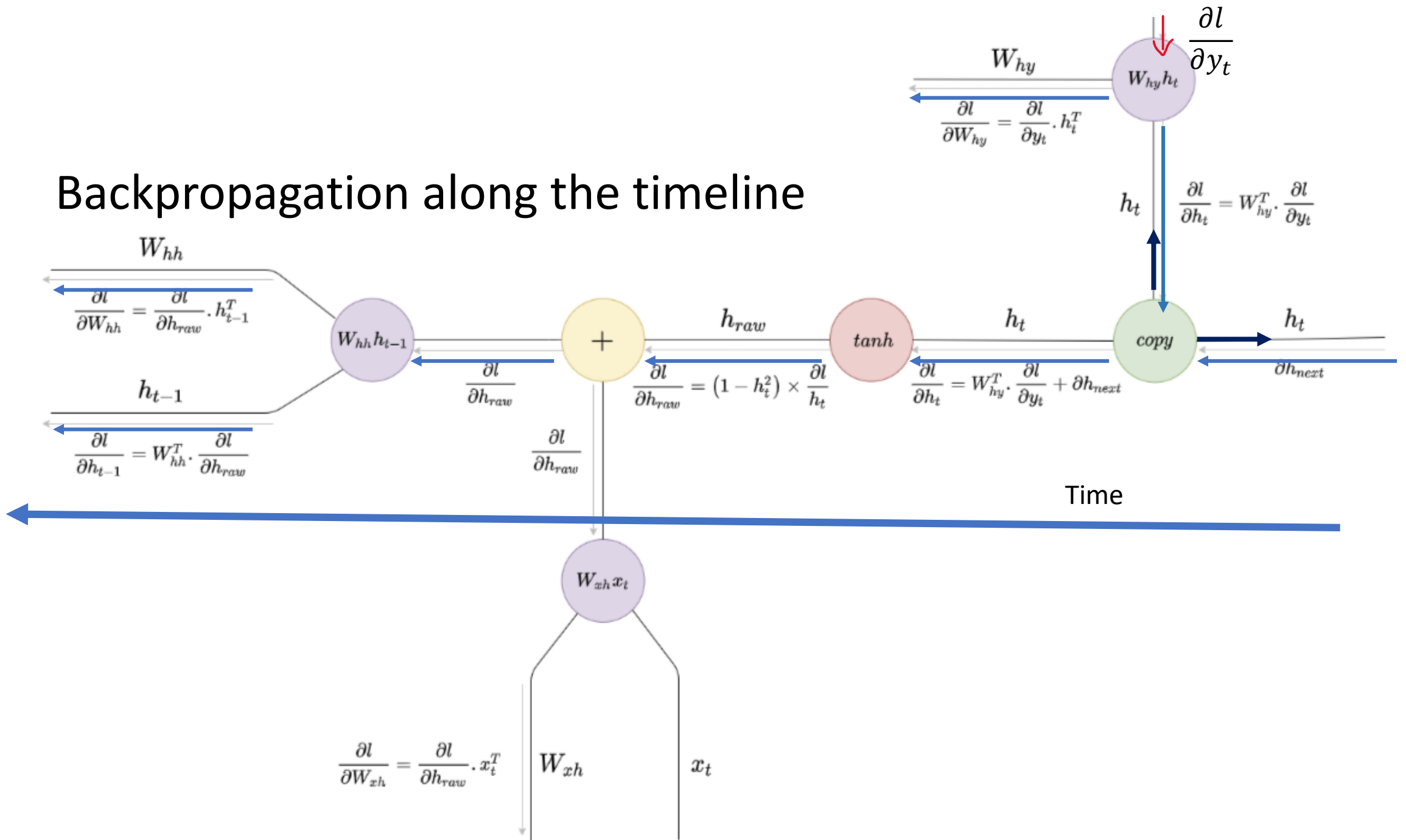


# Step by step: softmax backpropagation

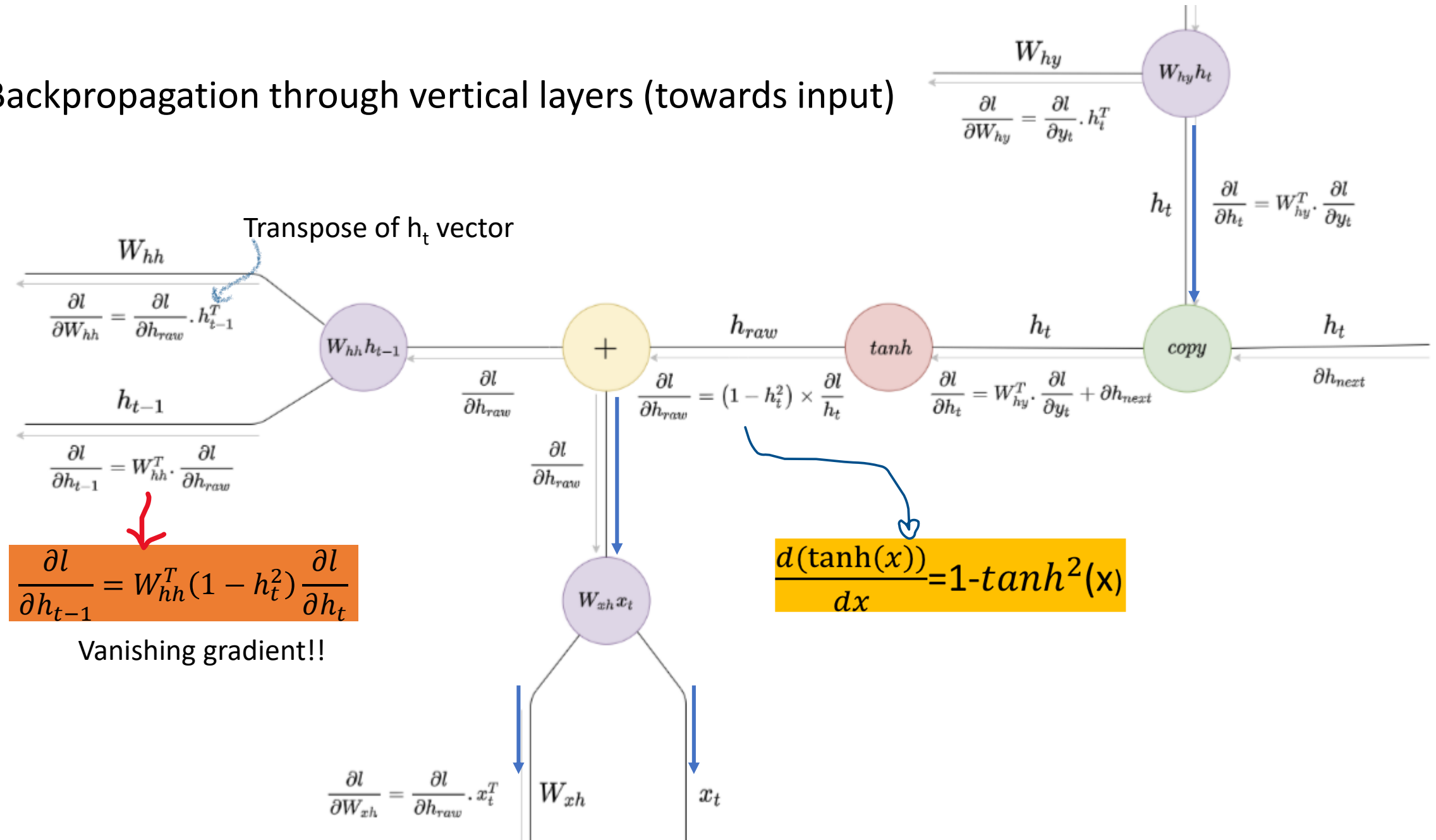


$$\frac{\partial L}{\partial y_t} = \{S_m - 1 \text{ if } t = m, \text{ else } S_t\} \text{ where } m \text{ is the index of the ground truth vector } [0, 0, \textcolor{red}{1}, 0, 0]$$

# Backpropagation along the timeline



# Backpropagation through vertical layers (towards input)



# Vanishing gradient with RNN

Each partial derivative of hidden states  $\frac{\partial L}{\partial h_t}$  is a function of the activation function used to produce that hidden state — often the tanh and sigmoid functions.

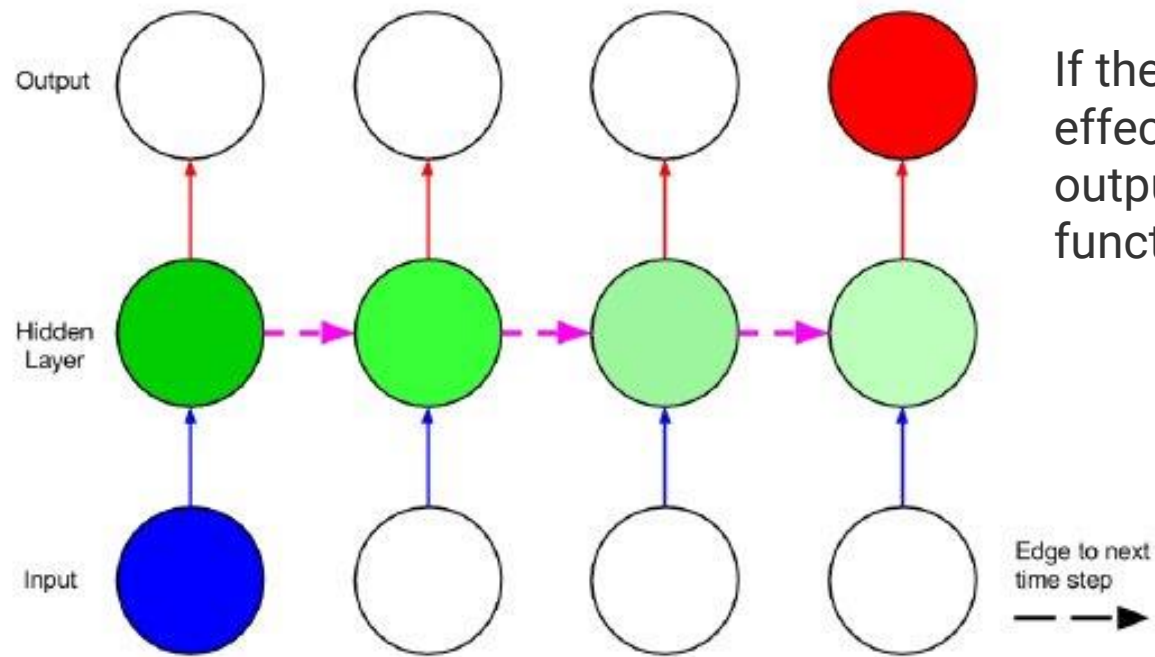
An important property of these functions is that they map input to values between -1 and 1 for tanh and 0 and 1 for sigmoid. **Thus, the derivative of the hidden states is generally bounded by 1.**

Because the gradient **is calculated using the product of these derivatives all the way through the time line** (which might be quite longer than the vertical line, e.g., the depth of the network), and the magnitude of each derivative of hidden states is less than 1, the result is that the entire gradient approaches 0 as the length of the sequence increases.

With a gradient that approaches 0, each update of the weight vectors becomes smaller and smaller, leading to a neural net that does not improve after a few samples of training, with stagnant weight vectors.

**RNN then tend to “forget” the effect of earlier signals !!!!**

# Vanishing gradient along the timeline



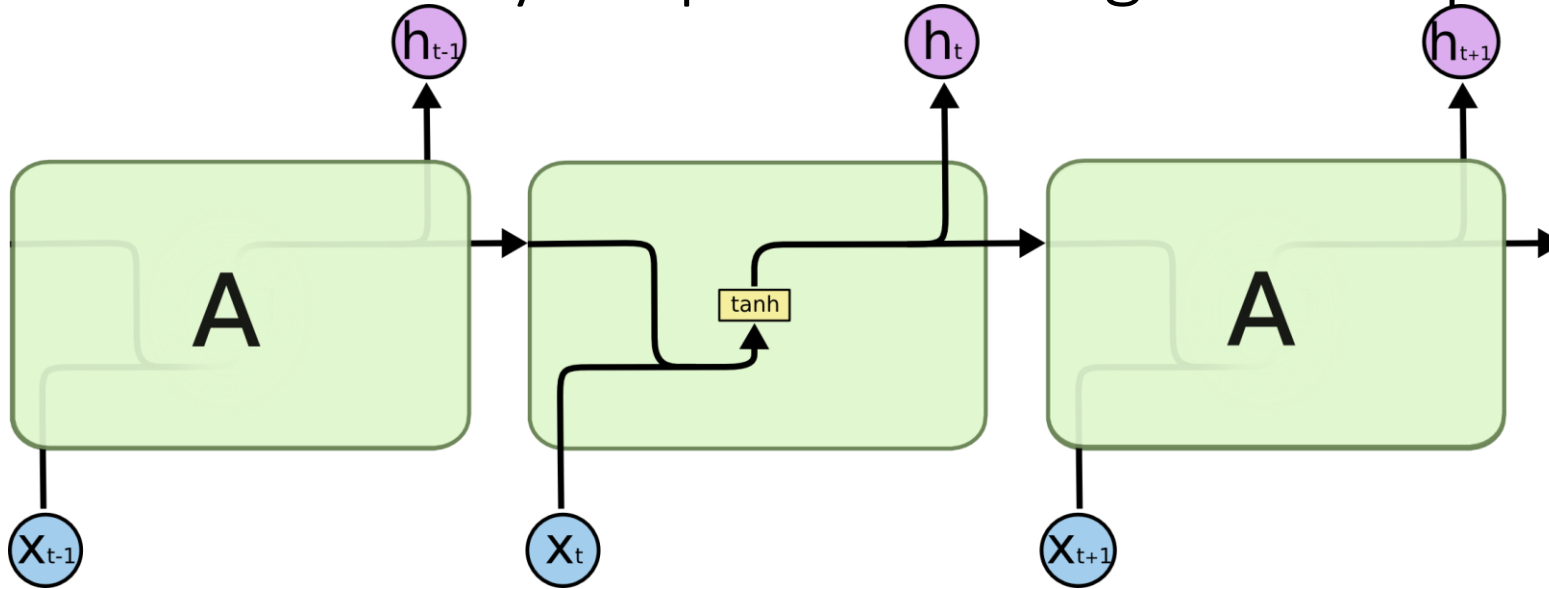
If the weights along the purple edge are less than one, the effect of the input at the first time step (blue) on the output at the final time step (red) will rapidly diminish as a function of the size of the interval in between.

RNN suffer from vanishing gradient!

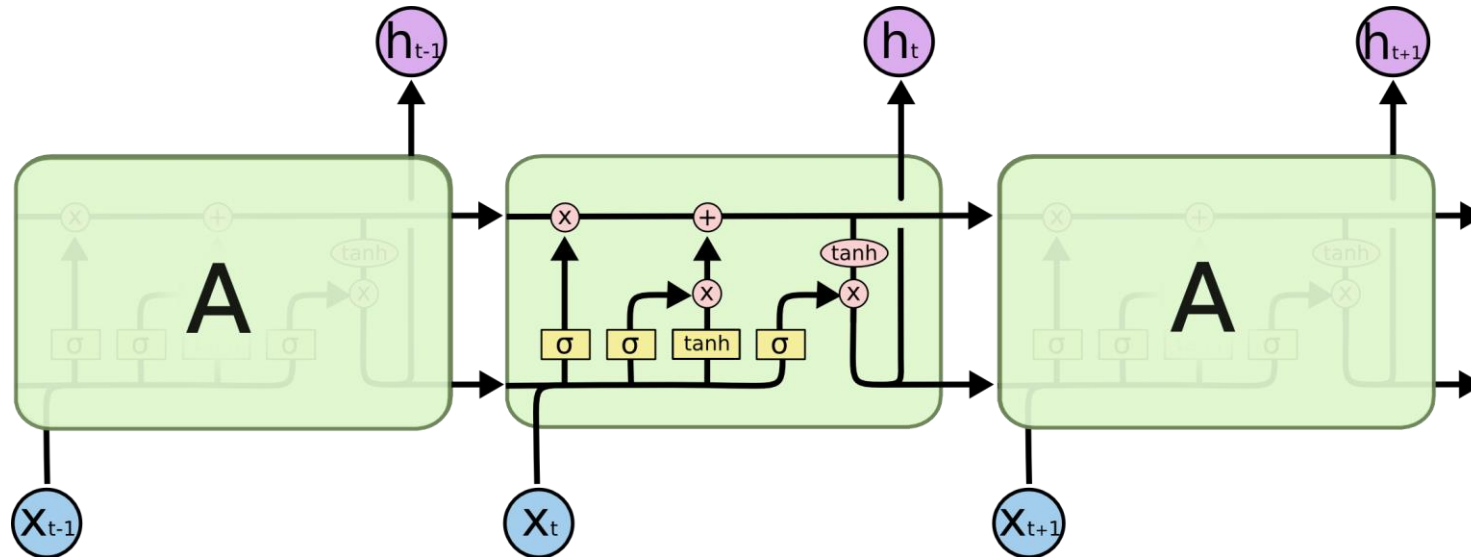
- ➔ Long-term dependencies are lost!
- Better solution: LSTM (long-short term memory)

Long Short-Term Memory: to preserve long-term dependencies

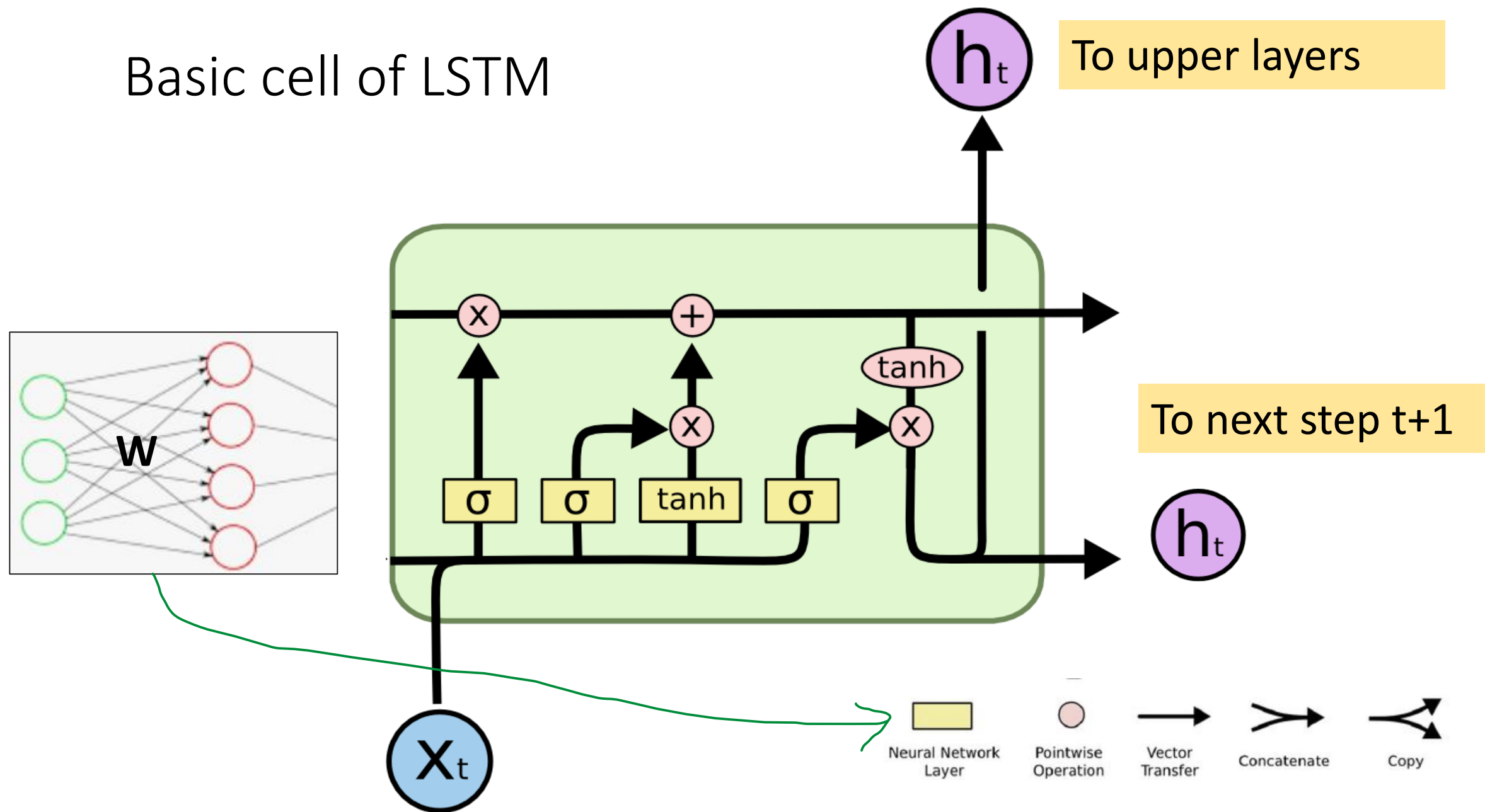
RNN



LSTM



# Basic cell of LSTM

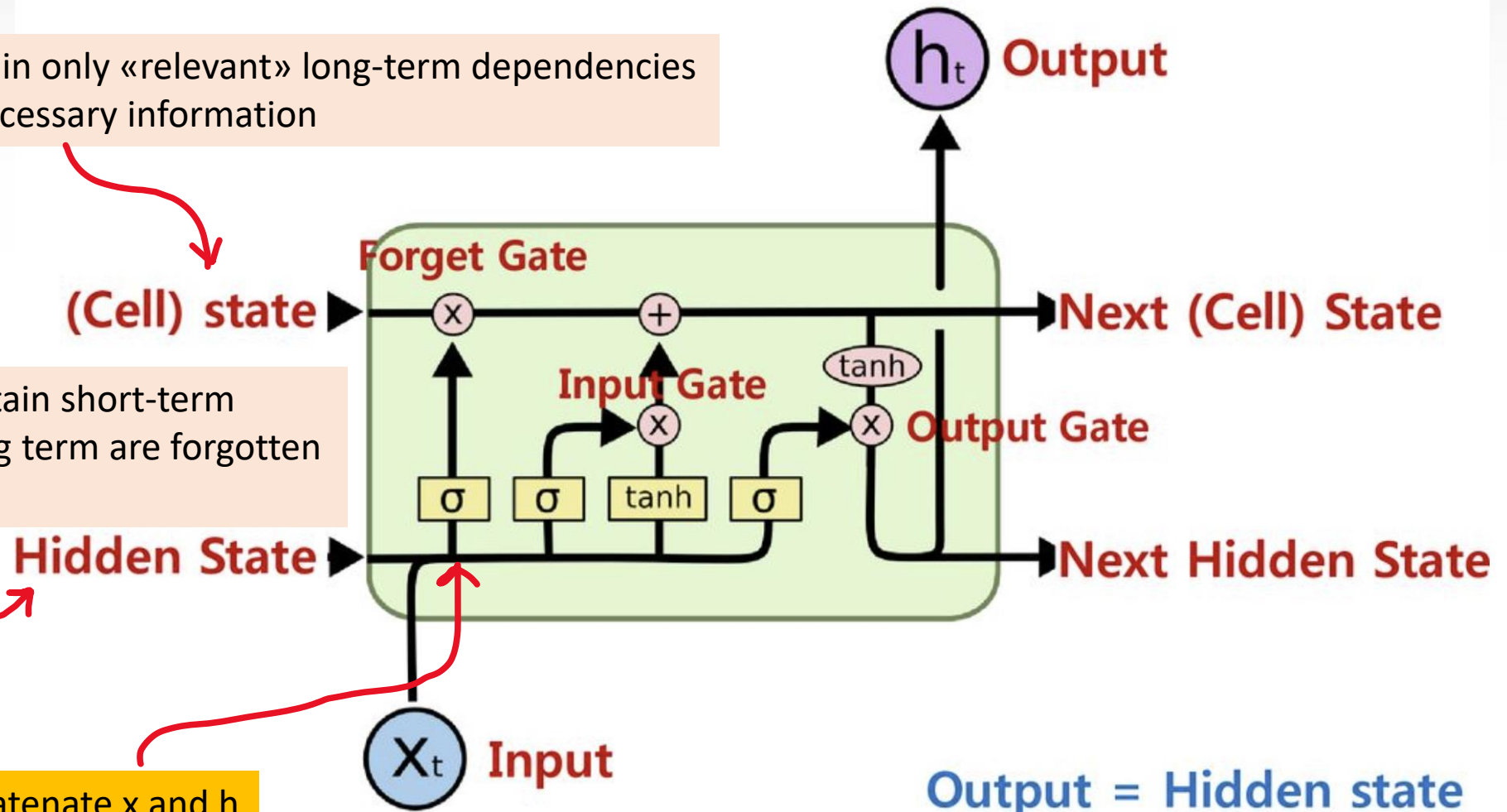




**Cell state** to retain only «relevant» long-term dependencies and cancel unnecessary information

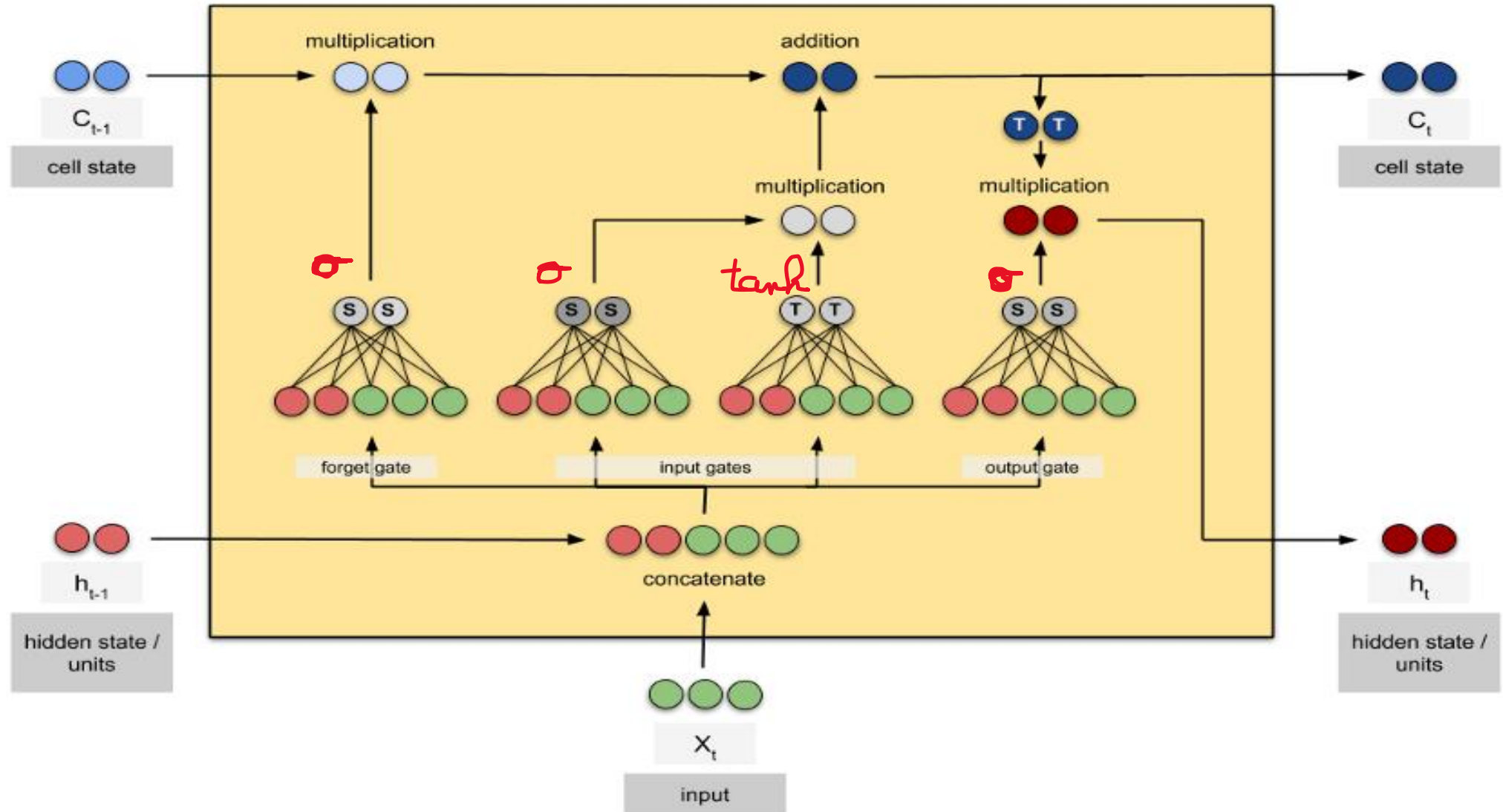
**Hidden state** to retain short-term dependencies (long term are forgotten due to VG)

Concatenate  $x$  and  $h$



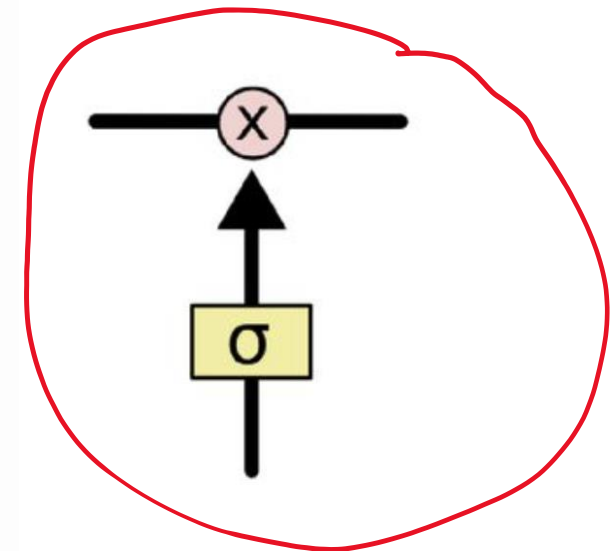
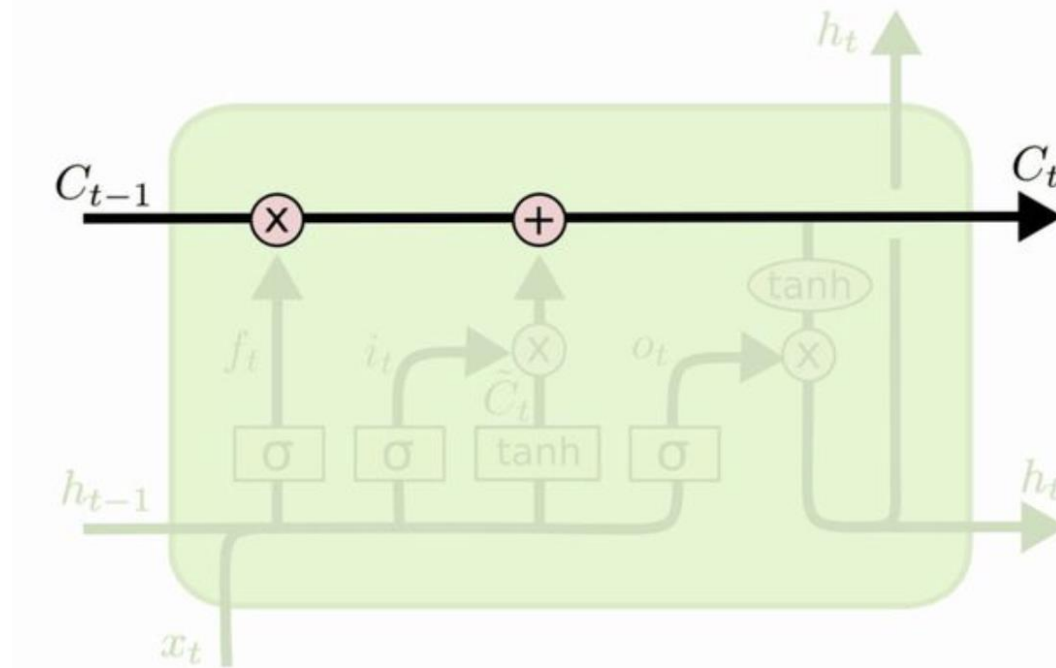
Output = Hidden state

## «Extended» view of the cell







# Cell state

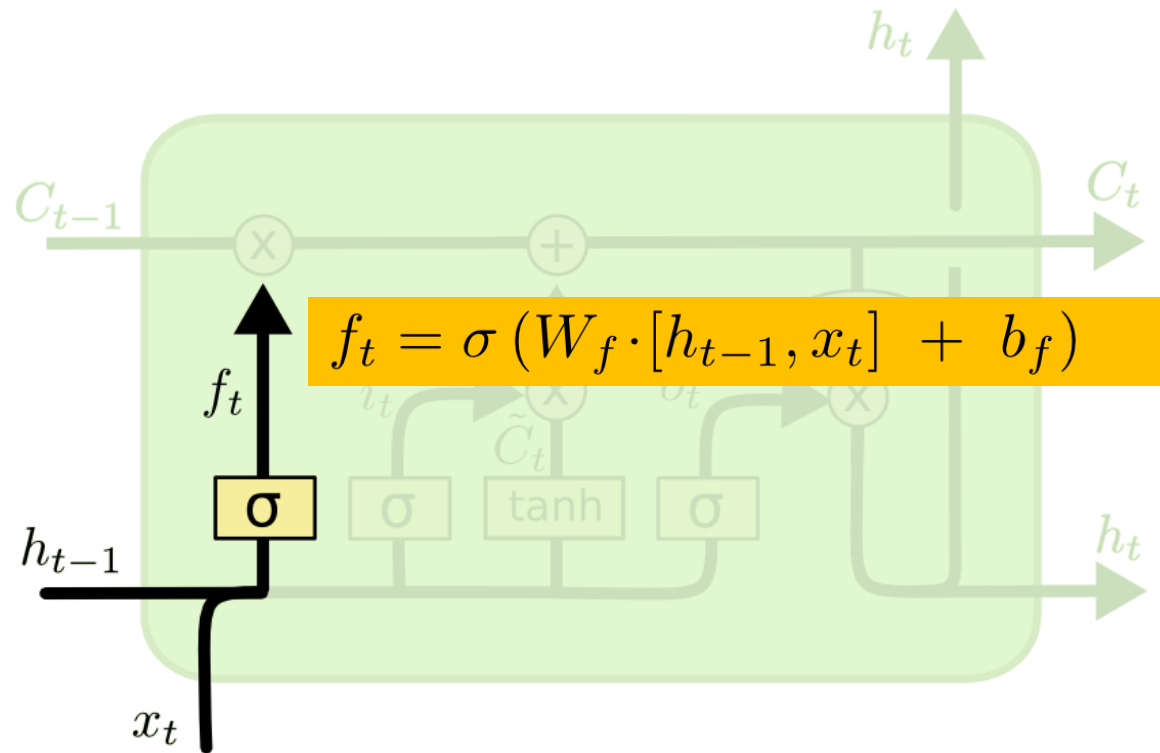
- Maintains a vector  $C_t$  that is the **same dimensionality** as the hidden state,  $h_t$
- Information can be deleted (forgotten) from, or added to, this state vector  $C$  via the Forget ( $\times$ ) and Input ( $+$ ) operators.
- The “X” operator multiplies  $C_t$  vector by the output of sigmoid vector, whose values are in the range  $[0,1]$ . This causes *some of the elements in  $C_t$  to be forgotten* (deleted).
- The “+” operator adds to  $C_t$  some of the information computed in the current cell
- Help remember long-term dependencies (sort of “summary” of previous history, updated in each cell)



# Cell State Example: forget and add

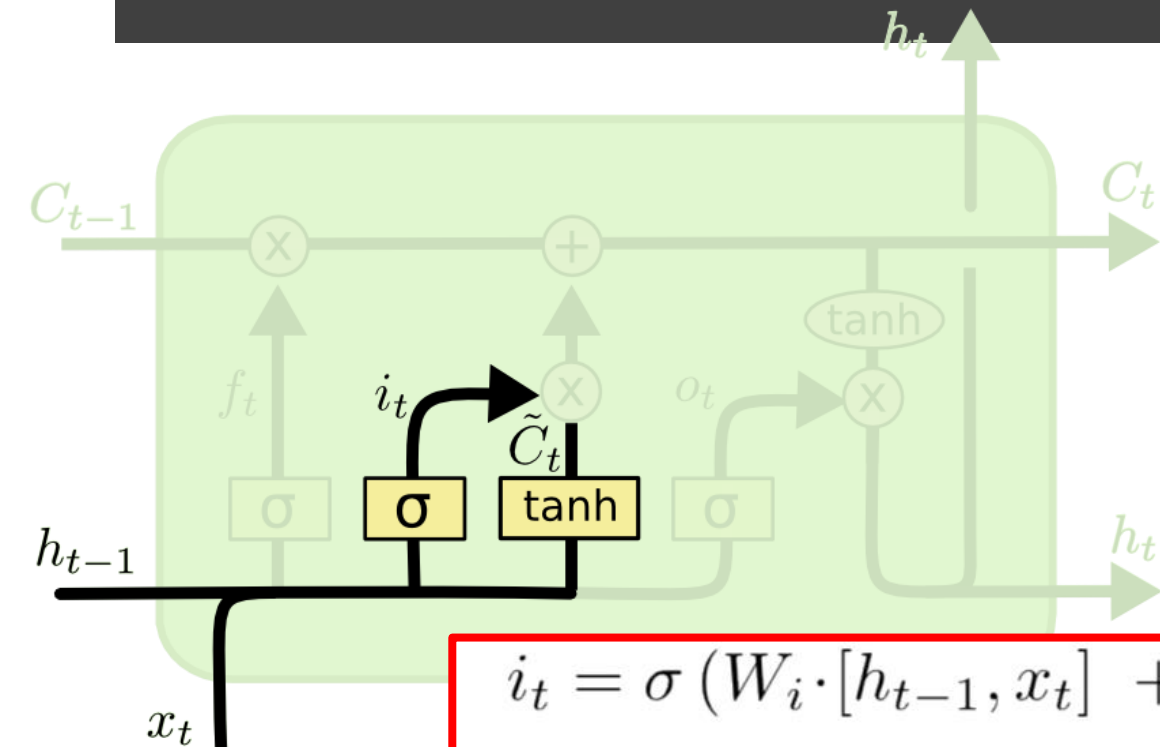
- Automated anaphora resolution: Want to remember person & number of a subject noun so that it can be checked to agree with the person & number of pronoun when it is eventually encountered.
- Example: John is a student, **he** likes machine learning.  

- **Forget**  gate will remove (multiply by zero or small value) existing information of a prior subject when a new one is encountered.  

- Example: ~~John~~ is a student, George is a professor, **he** hates machine learning.
- **Input**  gate "adds" in the information for the new subject.

# Forget Gate (what should be forgotten about previous states)



- Forget gate computes a 0-1 value using a sigmoid output function from the input,  $x_t$ , and the current hidden state,  $h_t$ :
- Multiplicatively combined with cell state C, "forgetting" information where the gate outputs something close to 0.

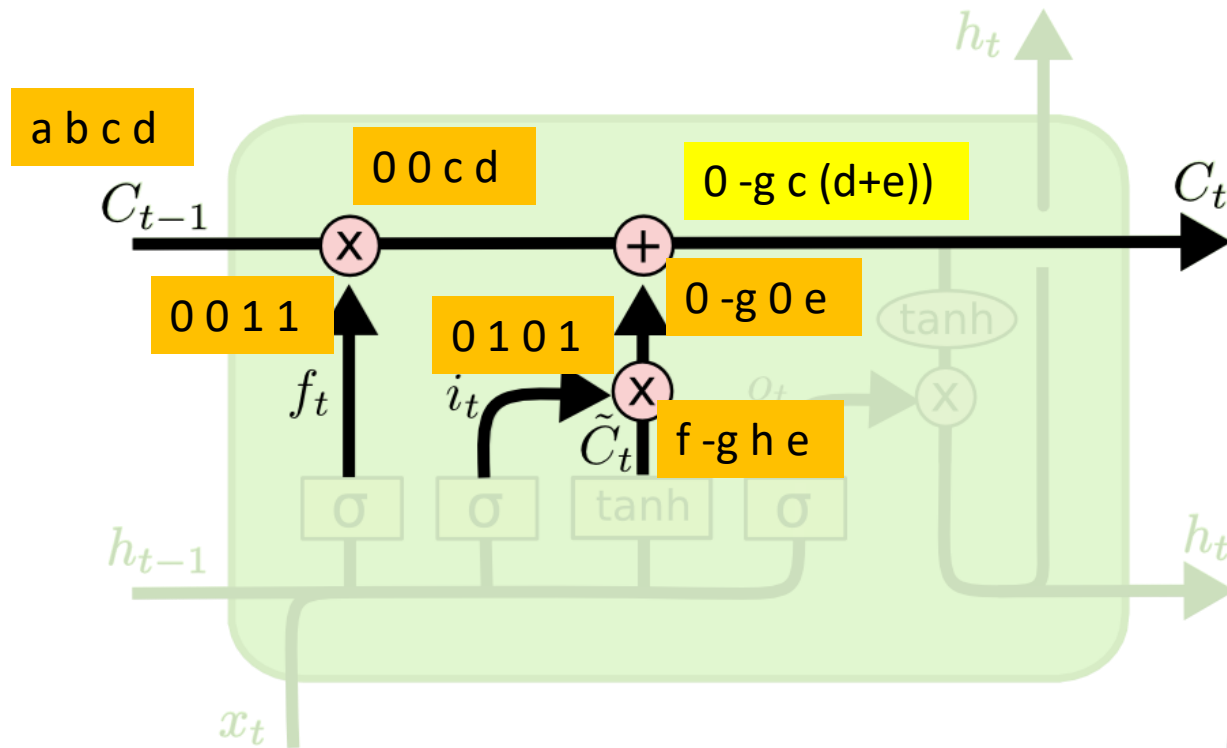
# Input Gate (what is added by this state)



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

- First, determine **which entries in the cell state to update** by computing 0-1 sigmoid output (when zero the correspondent value  $c_{jt}$  in  $C_t$  is not updated,  $c_{jt} + 0 = c_{jt}$ )
- Then determine the sign (whether to add or subtract) of these entries by computing a tanh output (valued  $-1$  to  $1$ ) function of the input and hidden state.

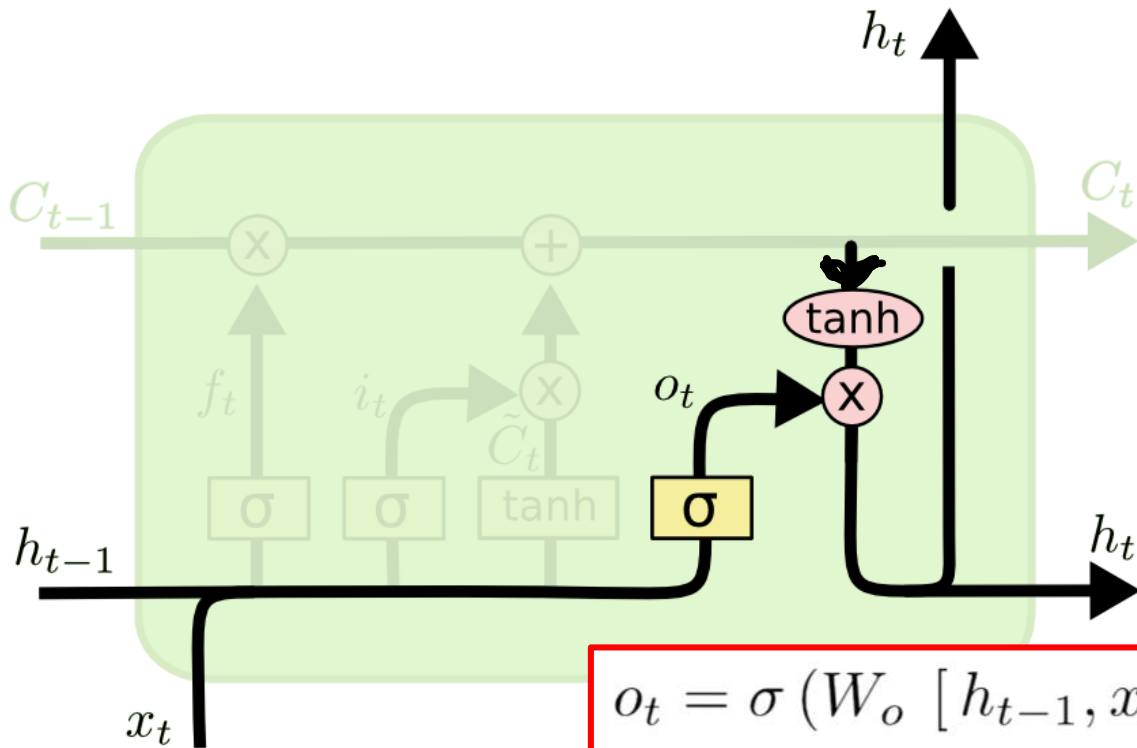
# Updating the Cell State



- Cell state is updated by using component-wise vector multiply to "forget" and vector addition to "input" new information.
- Note that actually the values (a b c d) are SCALED in  $[-1 \ 1,]$  rather than multiplied by  $-1$   $+1$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

# Output Gate (what goes to next layer and next state)

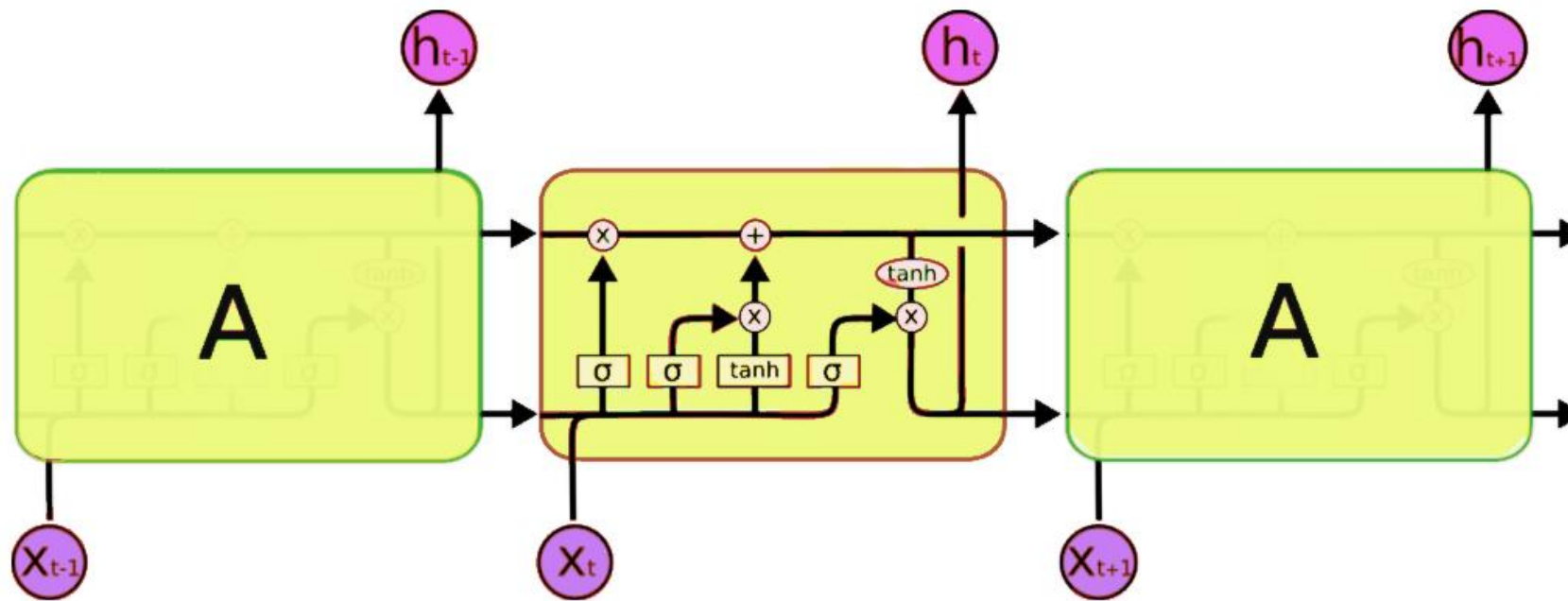


$$o_t = \sigma (W_o [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh (C_t)$$

- Hidden state is updated based on a "filtered" version of the cell state, scaled to  $[-1, 1]$  using tanh.
- Output gate computes a sigmoid function of the input and current hidden state to determine which elements of the cell state to "output".





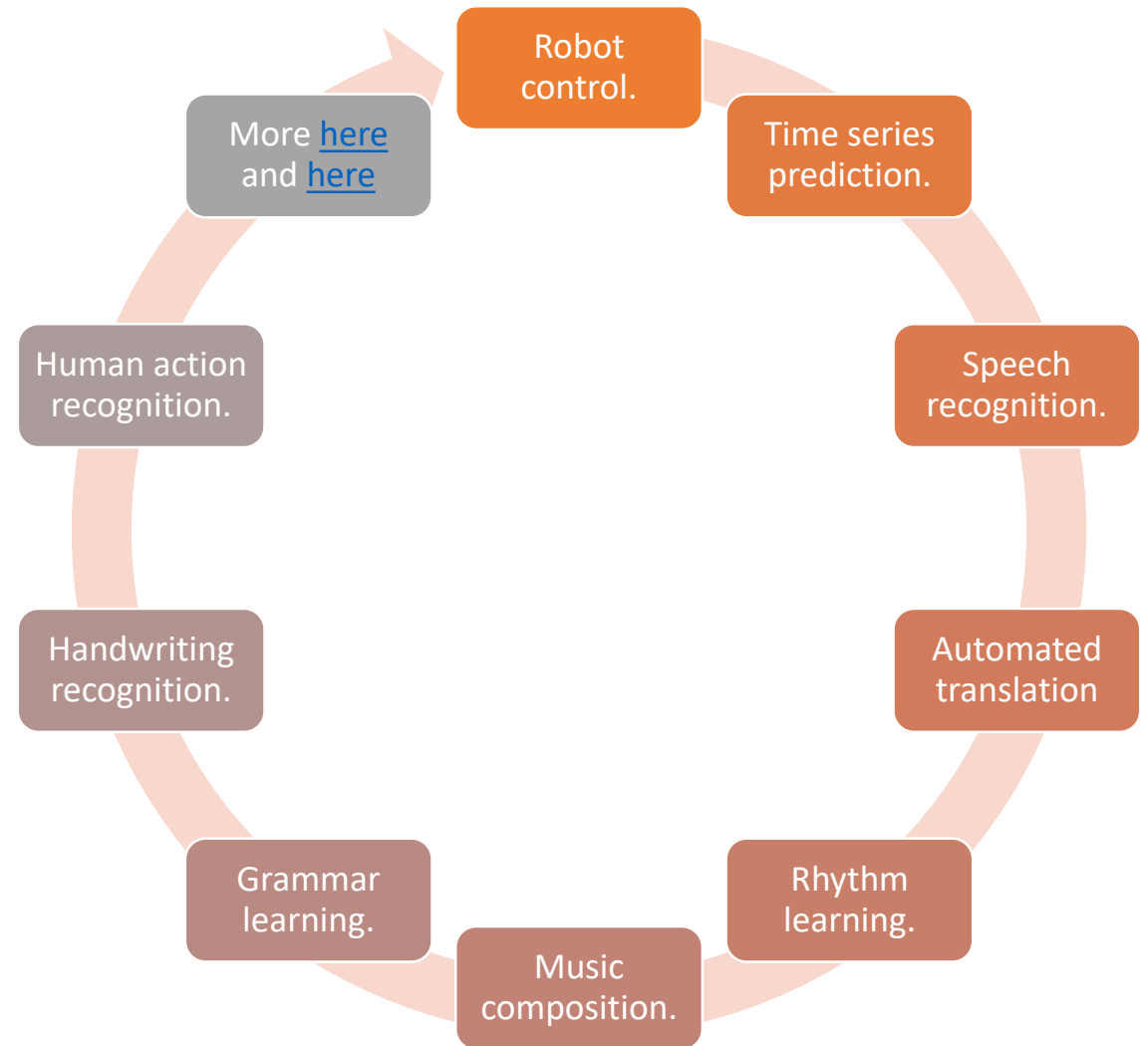
# Overall Network Architecture

---

# LSTM Training

- Trainable with backpropagation derivatives - Stochastic gradient descent with momentum
- Each cell has many parameters ( $W_f$ ,  $W_i$ ,  $W_C$ ,  $W_o$ )
  - Generally requires lots of training data.
  - Requires lots of computation time that exploits GPU clusters.

# Common applications of LSTMs



# Advanced methods in sequence learning

- [Attention networks](#) to identify those elements in a sequence with the highest relevance
- [Transformers](#)