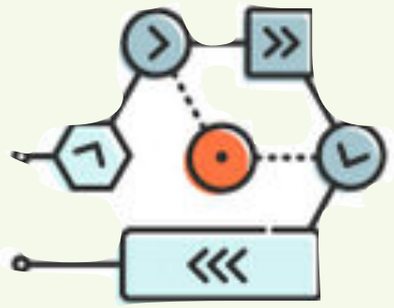
DEEP LEARNING



ALGORITHM

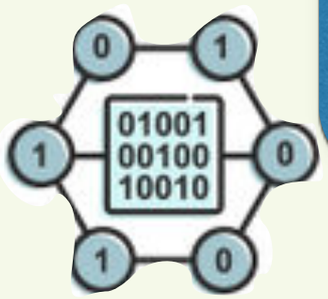
LEARNING

AI

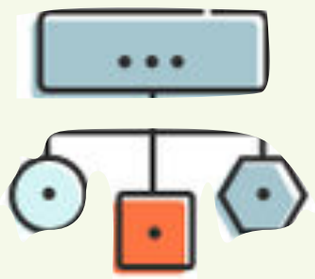
COURSE
"APLICAÇÕES DE INTELIGÊNCIA ARTIFICIAL"
University of Aveiro

Mestrado em Eng. Computational

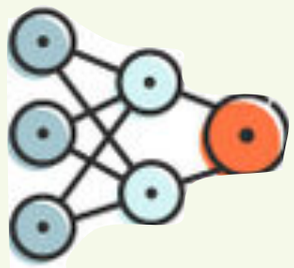
IMPROVES



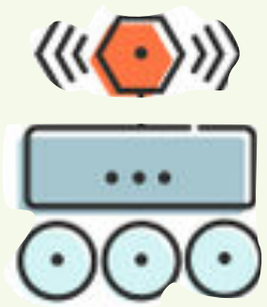
DATA MINING



CLASSIFICATION



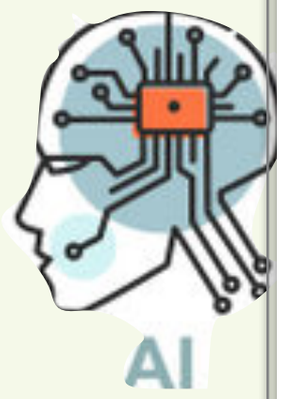
NEURAL NETWORKS

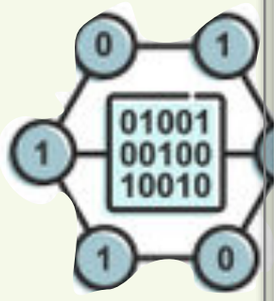


AUTONOMUS

ANALYZE

EXTRA INFO - this label will appear in slides with complementary information



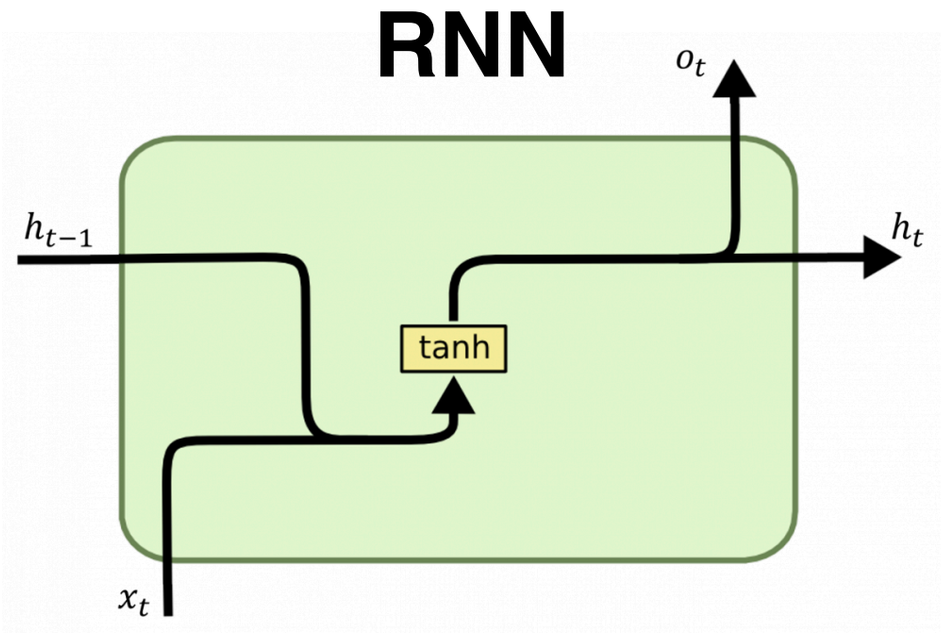


SUMMARY: Recurrent Neural Networks: LSTM.

[T] the LSTM model;

[P] Practical application of a RNN and a LSTM example..

EXTRA INFO - this label will appear in slides with complementary information

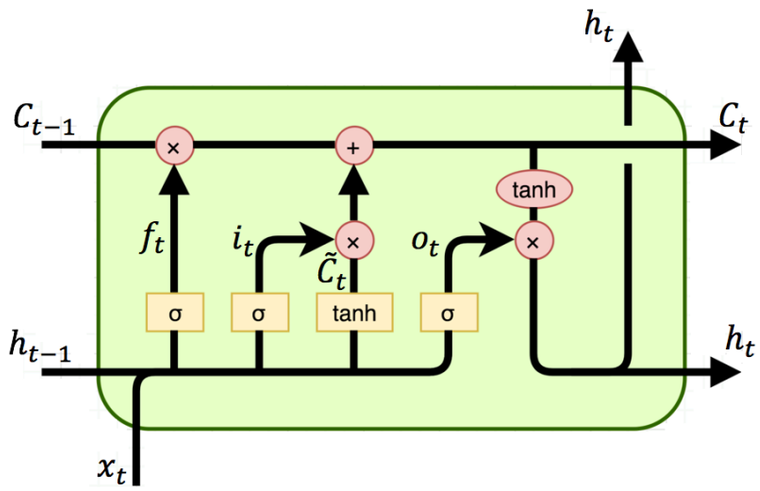


Feed-Forward

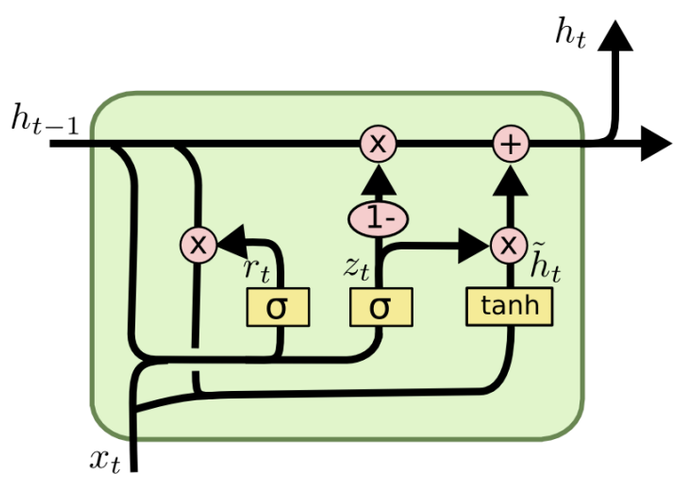
$$h_t = \sigma_h(i_t) = \sigma_h(U_h x_t + V_h h_{t-1} + b_h)$$

$$y_t = \sigma_y(a_t) = \sigma_y(W_y h_t + b_h)$$

LSTM – LONG-SHORT TERM MEMORY



GRU – GATED RECURRENT UNIT

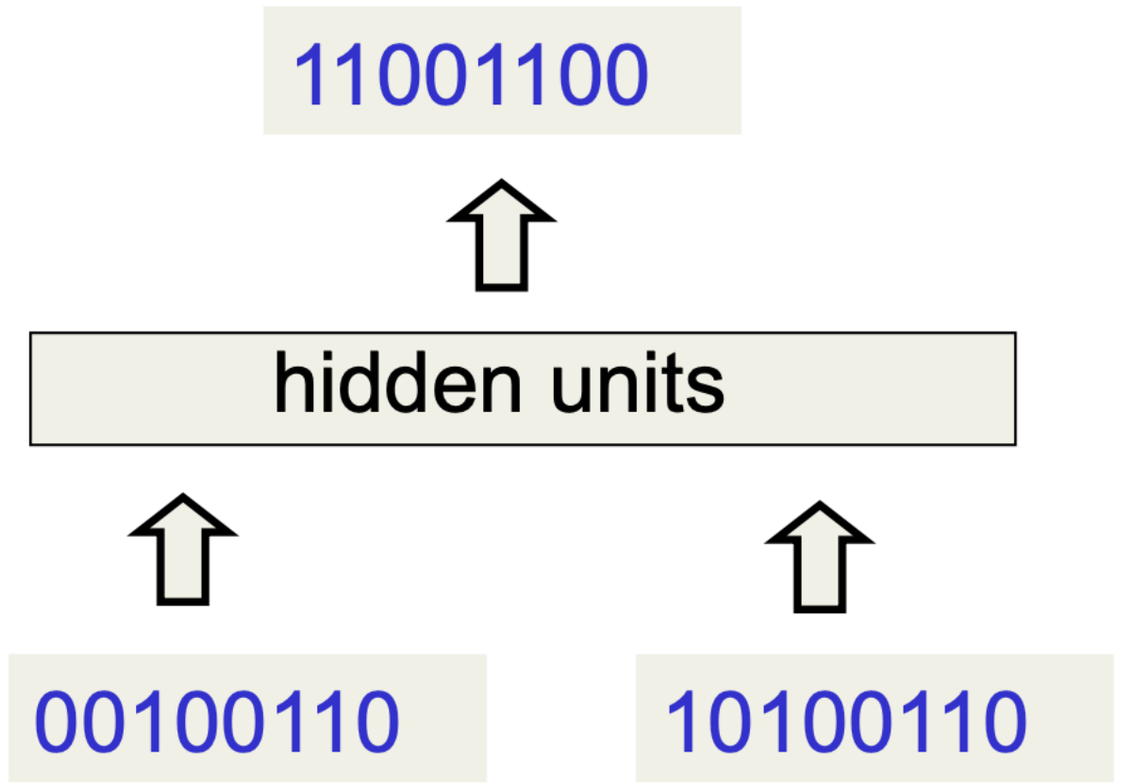


PRATICAL TASK

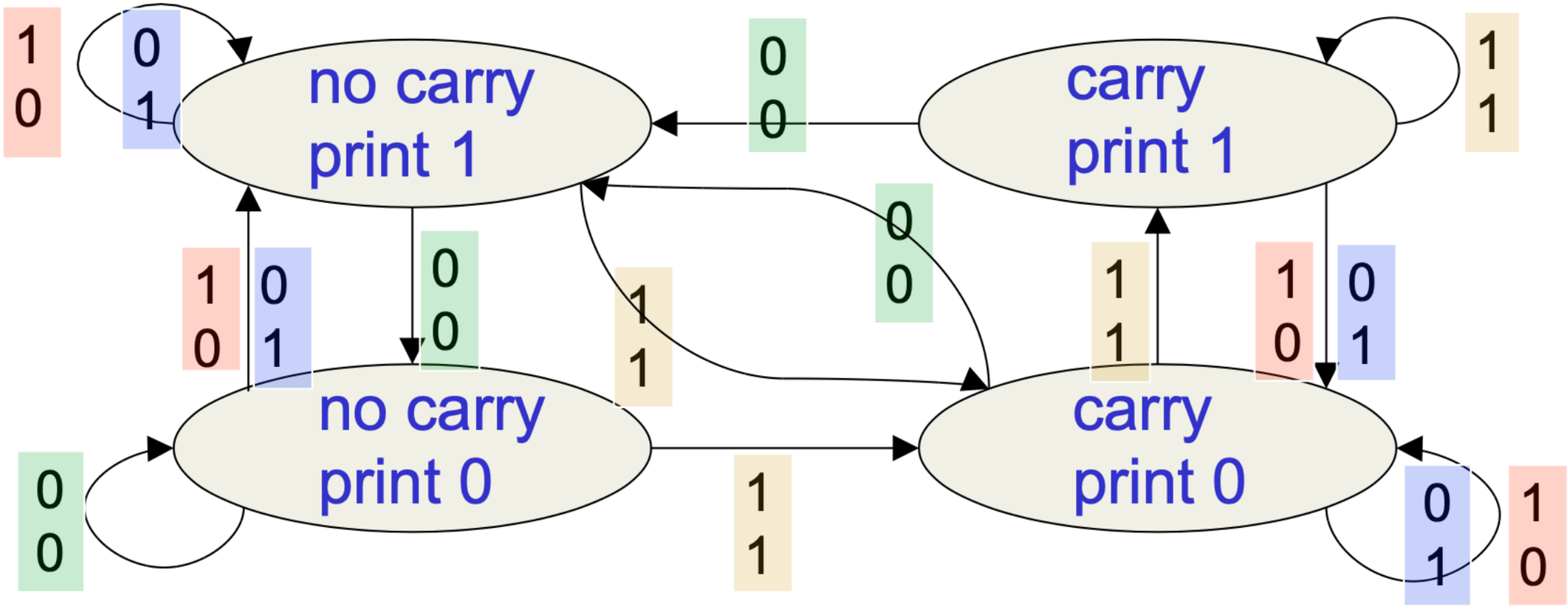
1. Explore the notebook RNN.ipynb
2. **Change the “input sequence” to something useful…**

A good toy problem for a recurrent network

- We can train a feedforward net to do binary addition, but there are obvious regularities that it cannot capture efficiently.
  - We must decide in advance the maximum number of digits in each number.
  - The processing applied to the beginning of a long number does not generalize to the end of the long number because it uses different weights.
- As a result, feedforward nets do not generalize well on the binary addition task.



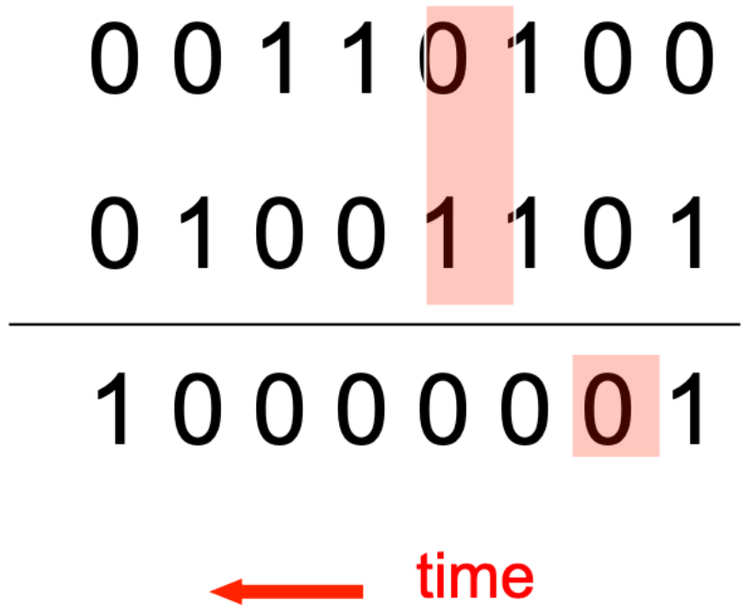
The algorithm for binary addition



This is a finite state automaton. It decides what transition to make by looking at the next column. It prints after making the transition. It moves from right to left over the two input numbers.

A recurrent net for binary addition

- The network has two input units and one output unit.
- It is given two input digits at each time step.
- The desired output at each time step is the output for the column that was provided as input two time steps ago.
  - It takes one time step to update the hidden units based on the two input digits.
  - It takes another time step for the hidden units to cause the output.

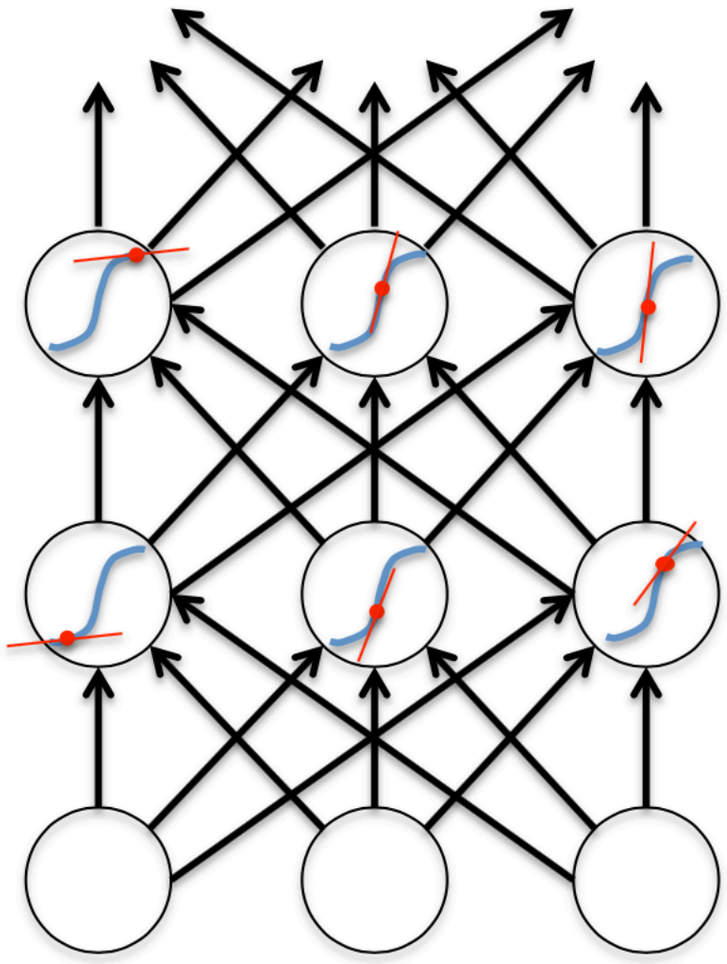


What the network learns

- It learns four distinct patterns of activity for the 3 hidden units. These patterns correspond to the nodes in the finite state automaton.
  - Do not confuse units in a neural network with nodes in a finite state automaton. Nodes are like activity vectors.
  - The automaton is restricted to be in exactly one state at each time. The hidden units are restricted to have exactly one vector of activity at each time.
- A recurrent network can emulate a finite state automaton, but it is exponentially more powerful. With N hidden neurons it has 2^N possible binary activity vectors (but only N^2 weights)
  - This is important when the input stream has two separate things going on at once.
  - A finite state automaton needs to square its number of states.
  - An RNN needs to double its number of units.

The backward pass is linear

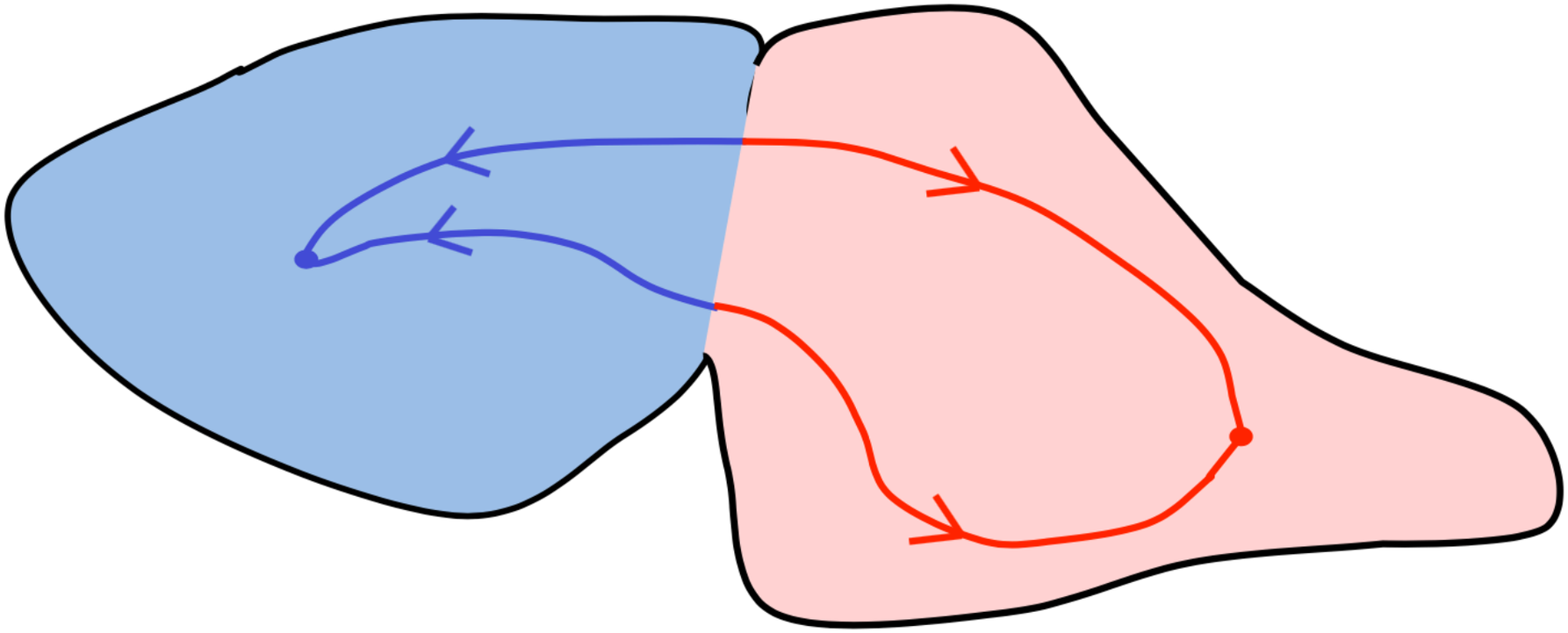
- There is a big difference between the forward and backward passes.
- In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding.
- The backward pass, is completely linear. If you double the error derivatives at the final layer, all the error derivatives will double.
  - The forward pass determines the slope of the linear function used for backpropagating through each neuron.



The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
  - If the weights are small, the gradients shrink exponentially.
  - If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.
  - We can avoid this by initializing the weights very carefully.
- Even with good initial weights, its very hard to detect that the current target output depends on an input from many time-steps ago.
  - So RNNs have difficulty dealing with long-range dependencies.

Why the back-propagated gradient blows up



- If we start a trajectory within an attractor, small changes in where we start make no difference to where we end up.
- But if we start almost exactly on the boundary, tiny changes can make a huge difference.

Four effective ways to learn an RNN

- Long Short Term Memory
  Make the RNN out of little modules that are designed to remember values for a long time.
- Hessian Free Optimization: Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature.
  - The HF optimizer (Martens & Sutskever, 2011) is good at this.
- Echo State Networks: Initialize the input→hidden and hidden→hidden and output→hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input.
  - ESNs only need to learn the hidden→output connections.
- Good initialization with momentum
  Initialize like in Echo State Networks, but then learn all of the connections using momentum.

Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
- They designed a memory cell using logistic and linear units with multiplicative interactions.
- Information gets into the cell whenever its "write" gate is on.
- The information stays in the cell so long as its "keep" gate is on.
- Information can be read from the cell by turning on its "read" gate.

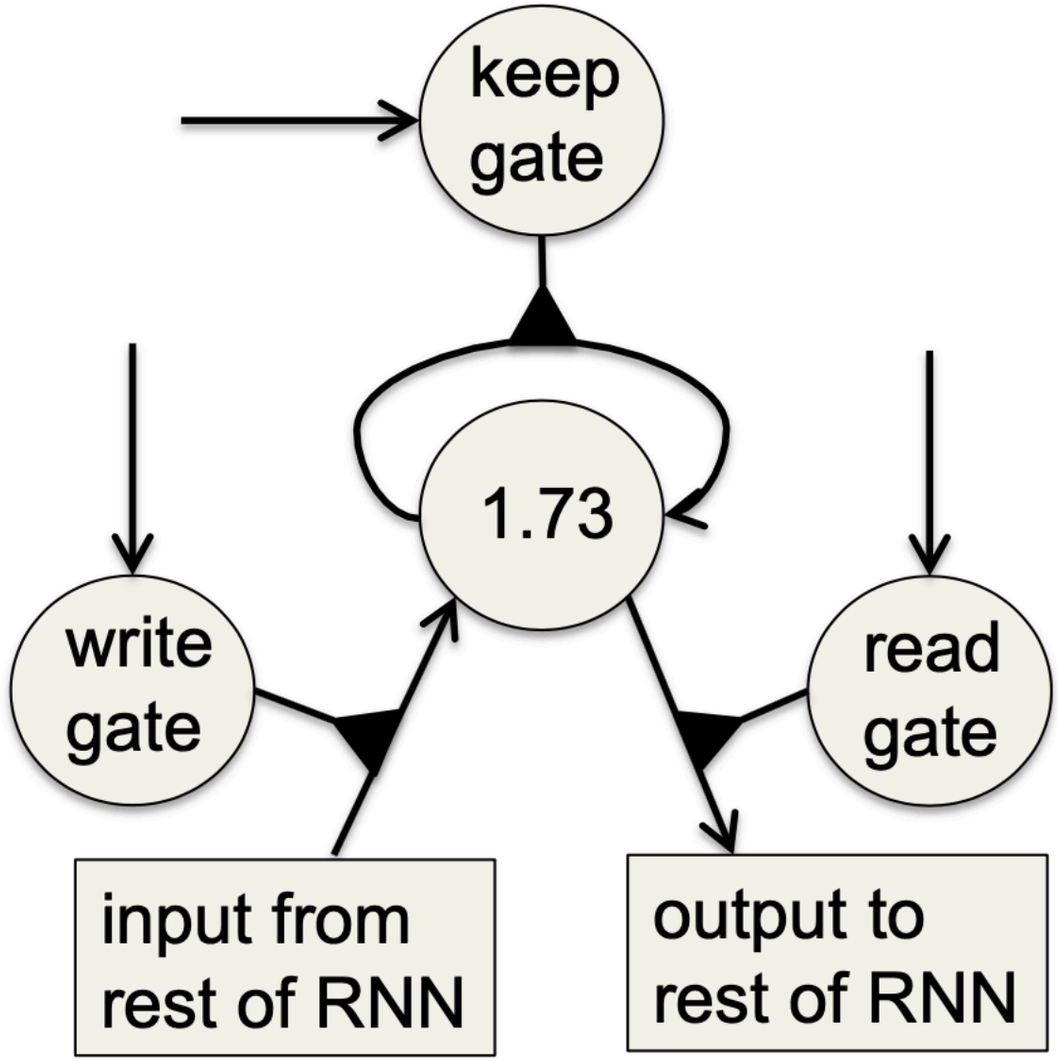
A great reference for dissecting the details of their paper is the blog post by Christopher Olah:

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

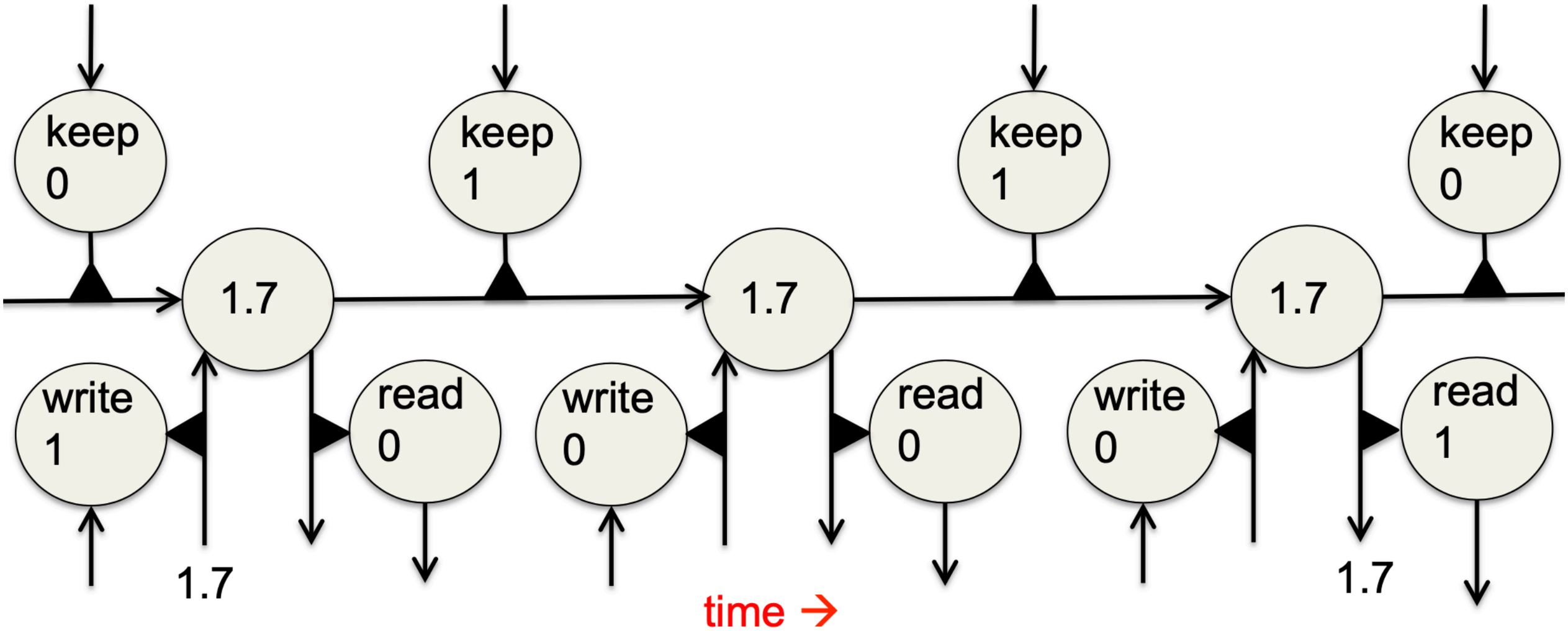
Implementing a memory cell in a neural network

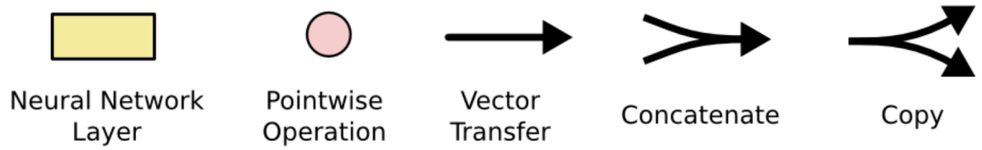
To preserve information for a long time in the activities of an RNN, we use a circuit that implements an analog memory cell.

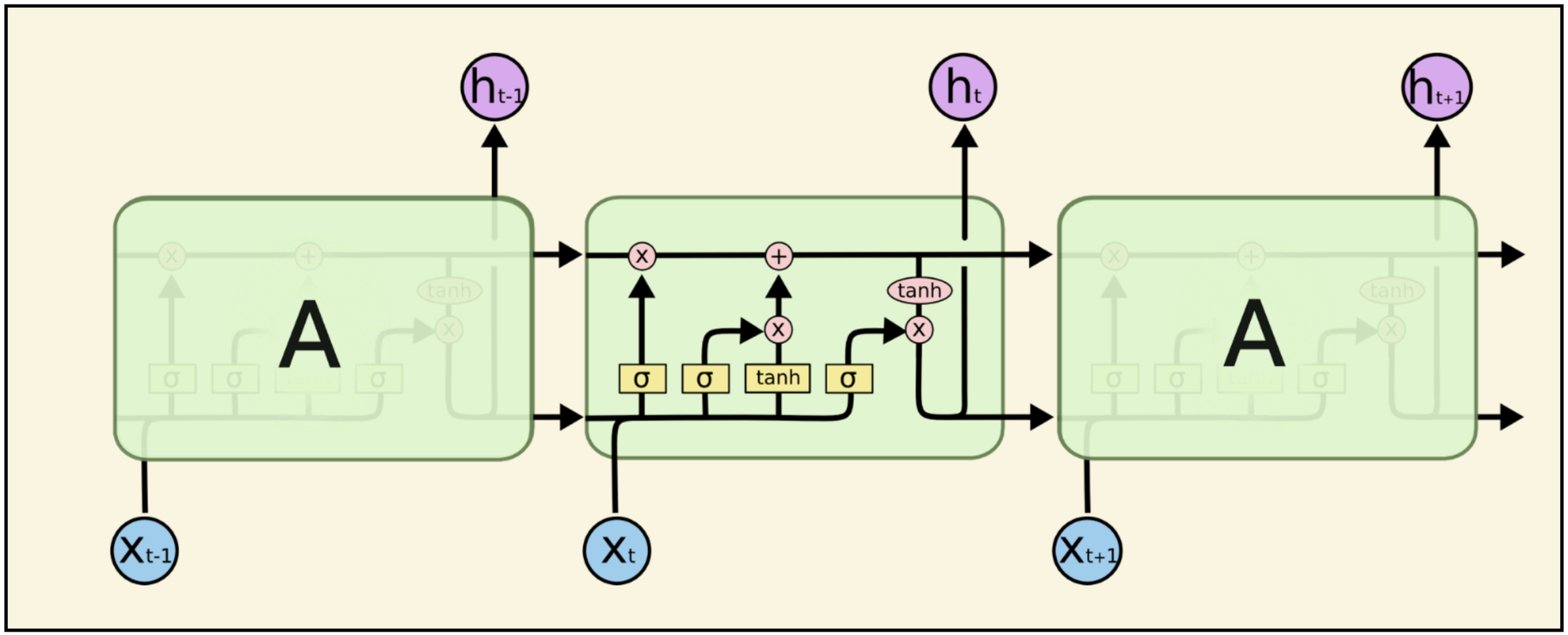
- A linear unit that has a self-link with a weight of 1 will maintain its state.
- Information is stored in the cell by activating its write gate.
- Information is retrieved by activating the read gate.
- We can backpropagate through this circuit because logistics are have nice derivatives.

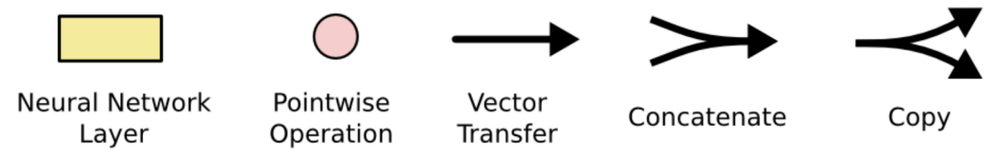


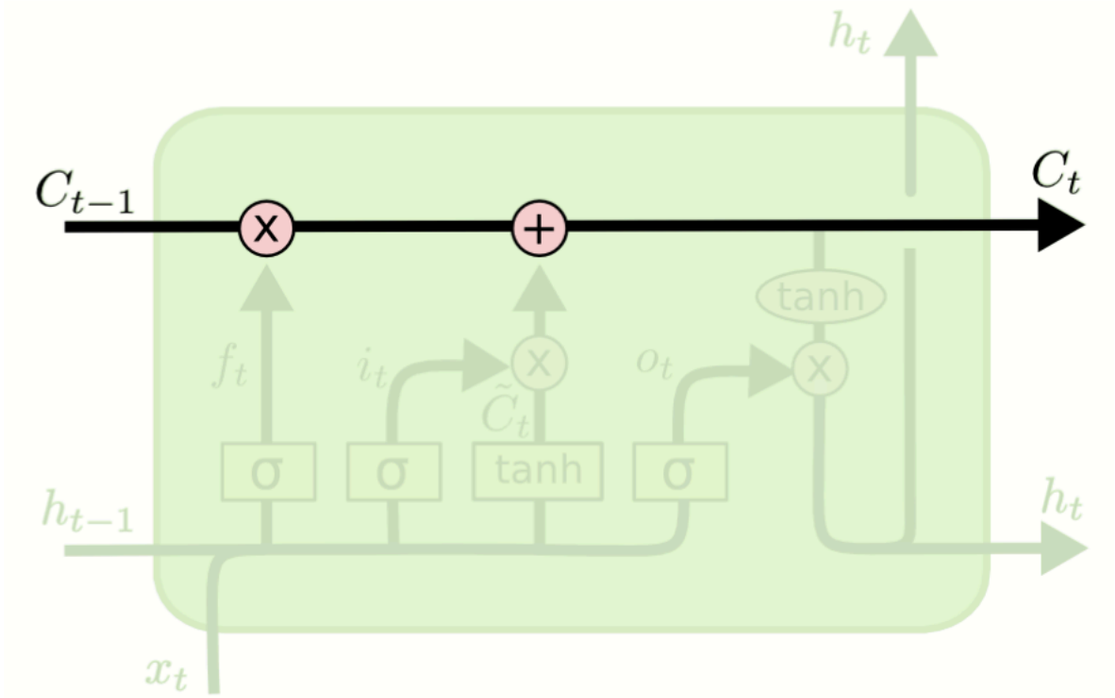
Backpropagation through a memory cell



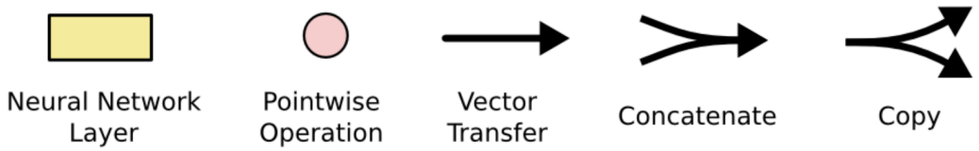

Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

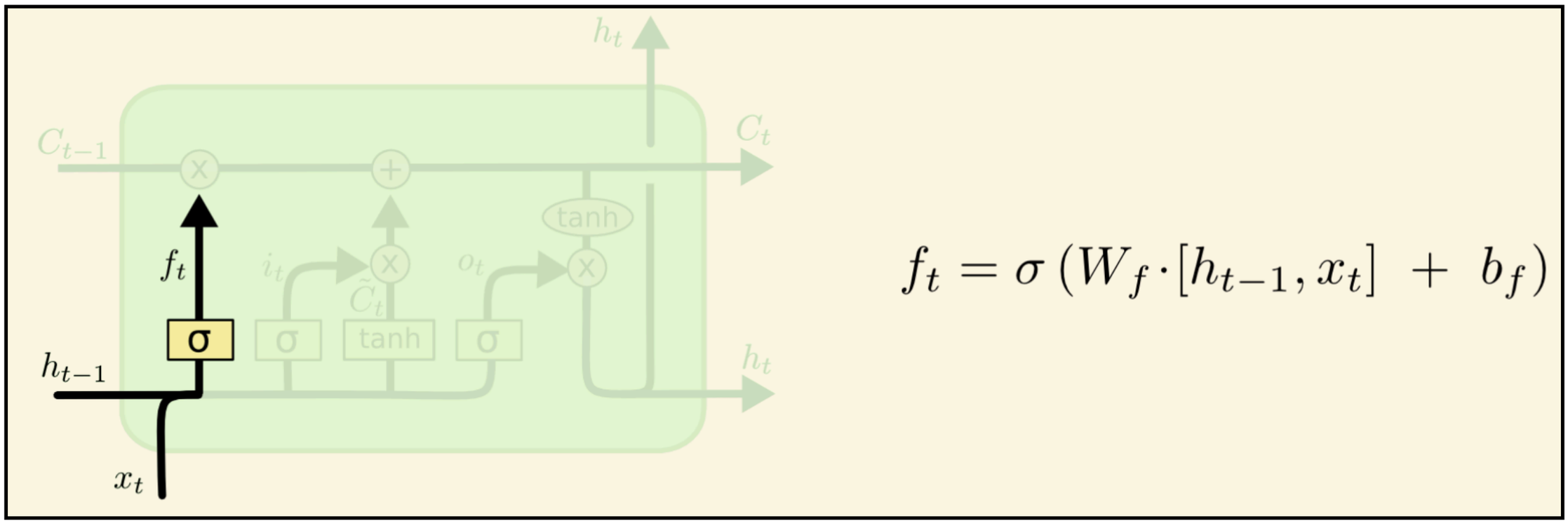

h_{t-1}
h_t
h_{t+1}
A
A
tanh
σ
σ
tanh
σ
X_{t-1}
X_t
X_{t+1}





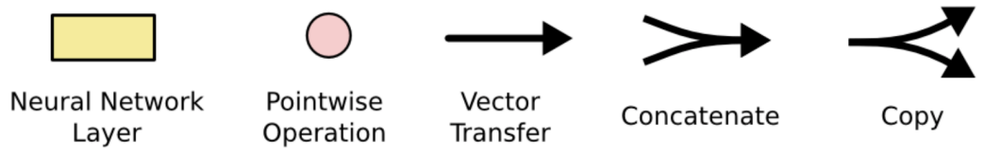
The Cell state need not be the same in each LSTM component, that propagates in the hidden layers of the RNN.

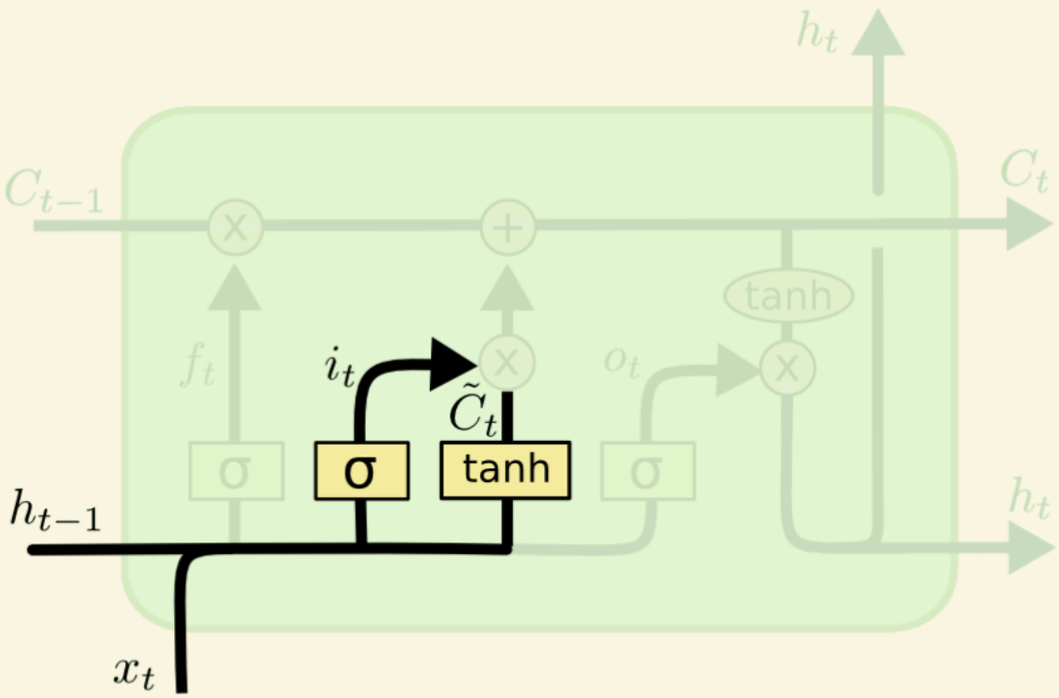




$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

FORGET GATE: uses the previous output and current input to derive the a new cell state.

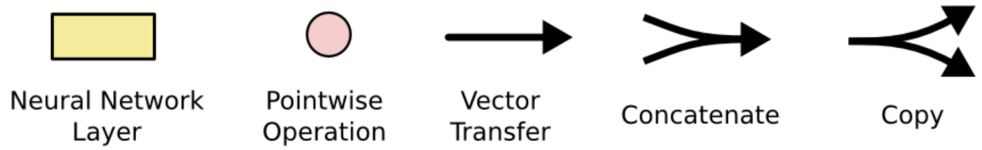


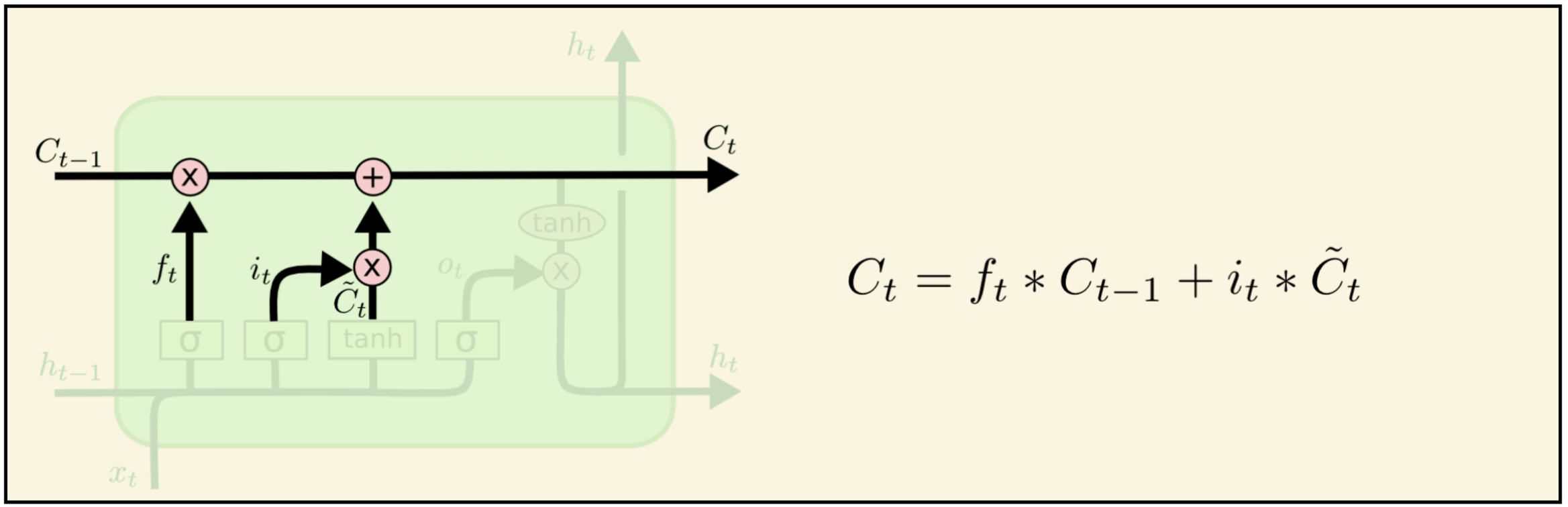


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

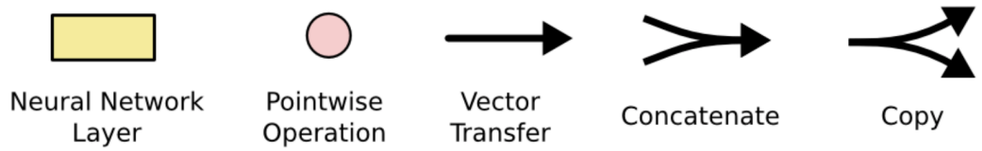
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

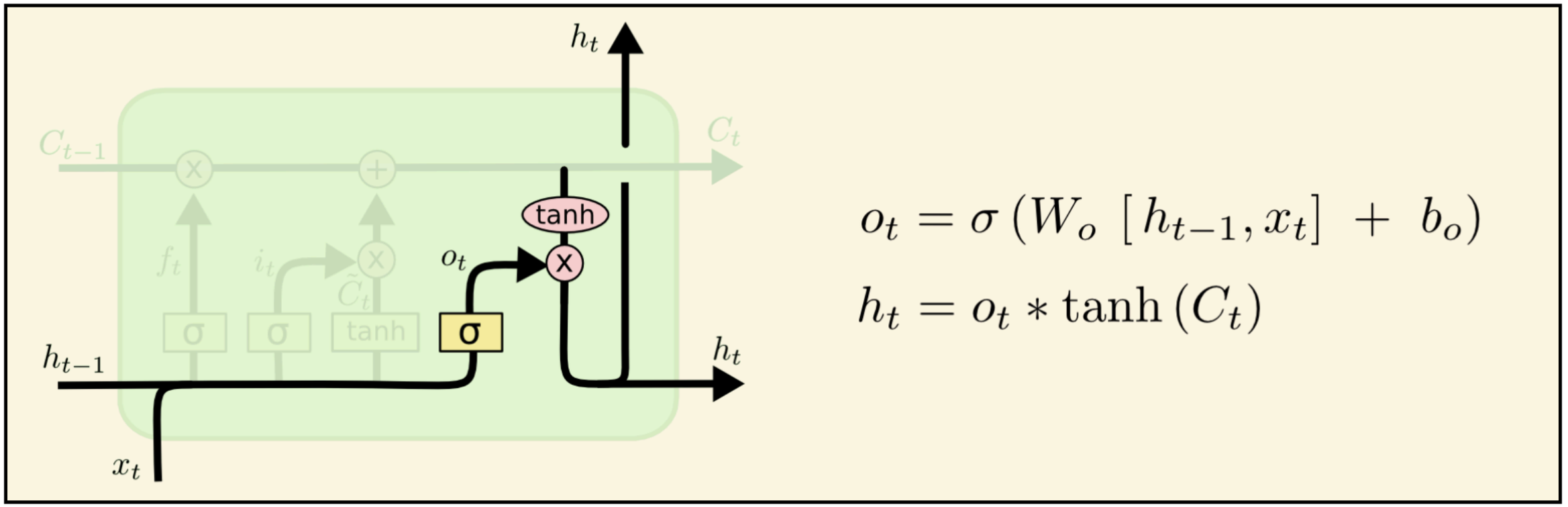
INPUT/WRITE GATE: uses the previous output and current input to determine if there is an update of the value.





STATE GATE: Can now be completely defined from the forget gate and teh update/write gate.

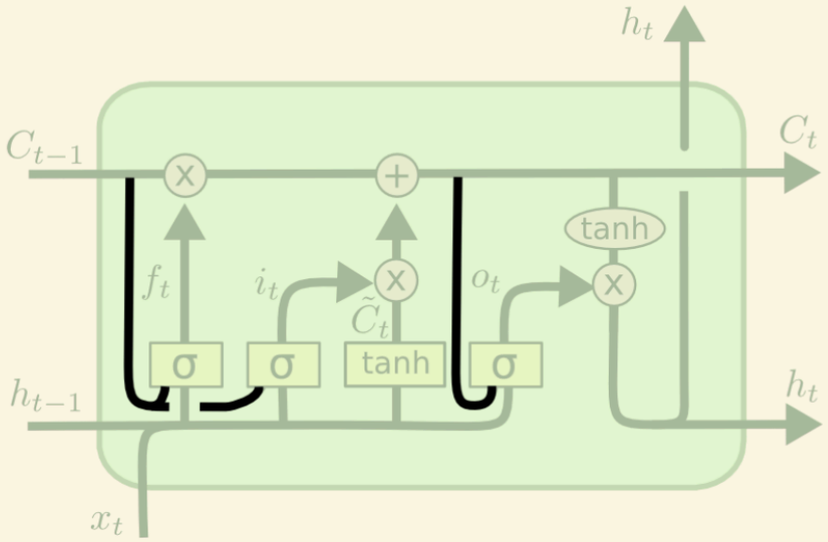




OUTPUT/READ GATE: It a weighted version of the cell state.

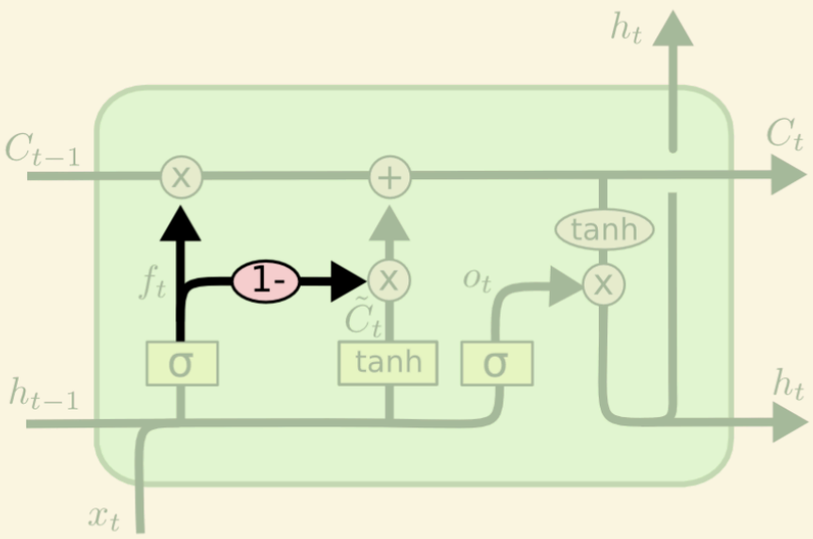
There are many LSTM variants in the Literature, e.g.:

(a) Introduction of “peepholes” where are gates also consider the state



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

(b) Making the forget gate the negation of the update gate



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

PRATICAL TASK

1. Explore the notebook LSTM.ipynb (univariable)
2. Change the “data.csv” file to something useful…