UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

Introduction to ML
Lecture 8: Modern RNNs

Hamed Tabkhi

Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

UNC CHARLOTTE

Problems with RNNs: Not remembering long-term information

- One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame.
- Sometimes, we only need to look at recent information to perform the present task.

- **In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.**

- But there are also cases where we need more context.
- Consider trying to predict the last word in the text "I grew up in France… I speak fluent *French*." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.
- Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

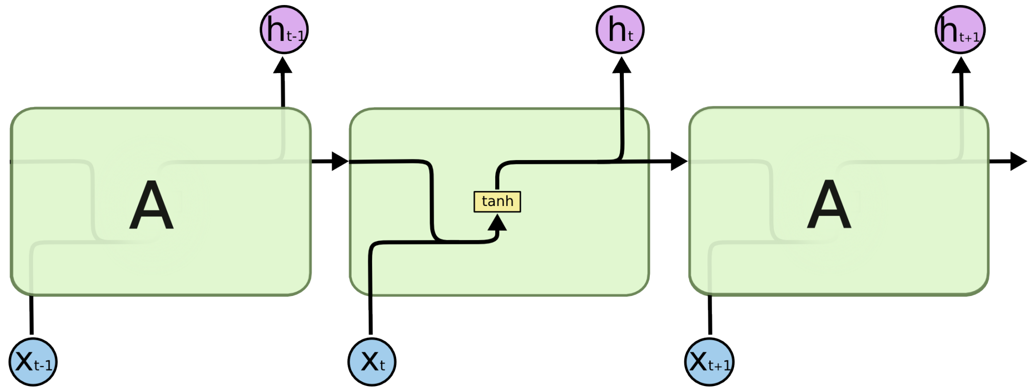
Long Short-Term Memories (LSTMs)

- Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.[1]
- They work tremendously well on a large variety of problems, and are now widely used.
- LSTMs are explicitly designed to avoid the long-term dependency problem.
- Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

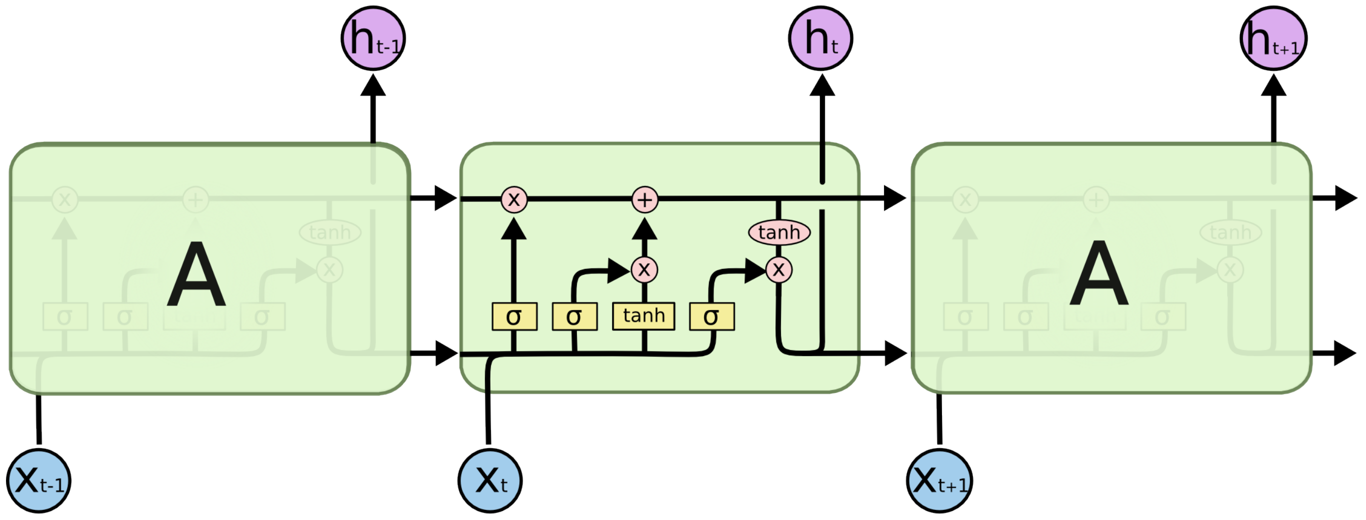
- Similar to classical RNNS, LSTMs have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer, in LSTMS it is replaced with a more complex structure.

UNC CHARLOTTE

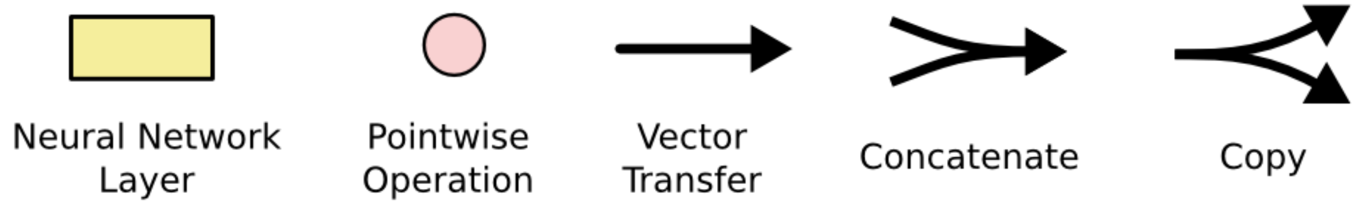
LSTM vs RNN Comparison



Traditional RNN

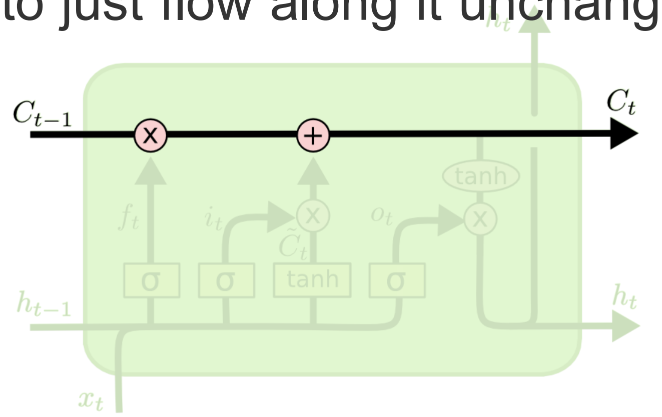


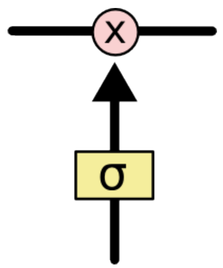
LSTM



The Core Idea Behind LSTMs

- The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
- The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.





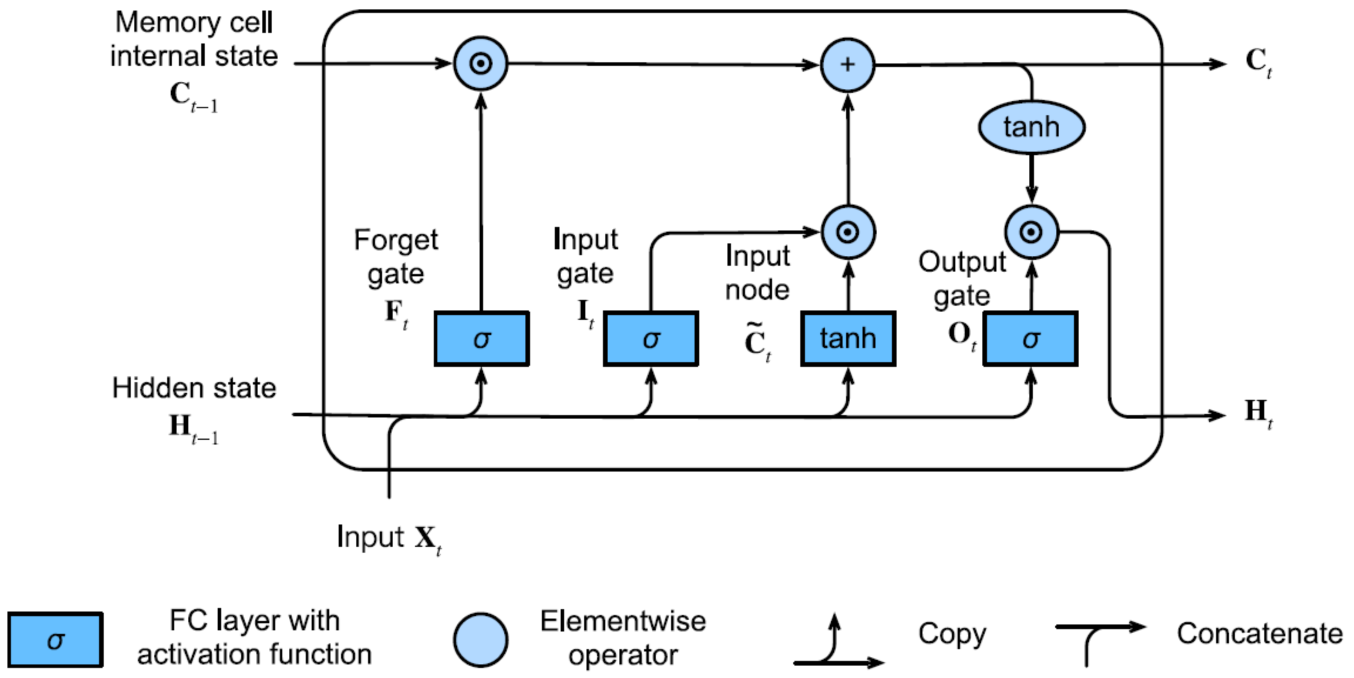
- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.
- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

- The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!"
- An LSTM has three of these gates, to protect and control the cell state.

UNC CHARLOTTE

LSTM Gates

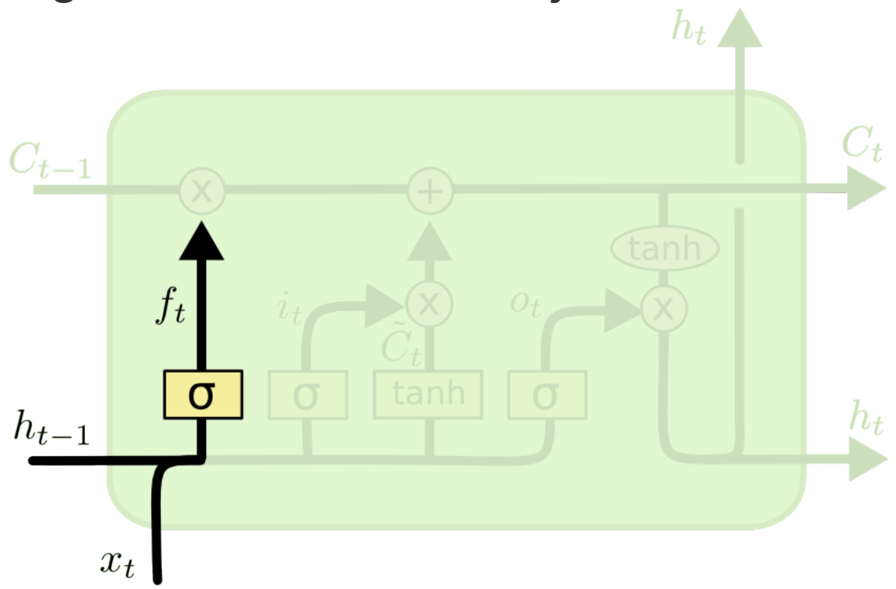
- **Forget Gate**: conditionally decides what information to throw away from the block
- **Input Gate**: conditionally decides which values from the input to update the memory state
- **Output Gate**: conditionally decides what to output based on input and the memory of the block


Memory cell internal state $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
tanh
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
tanh
Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

UNC CHARLOTTE

Step-by-Step LSTM Walk Through (Forget Gate)

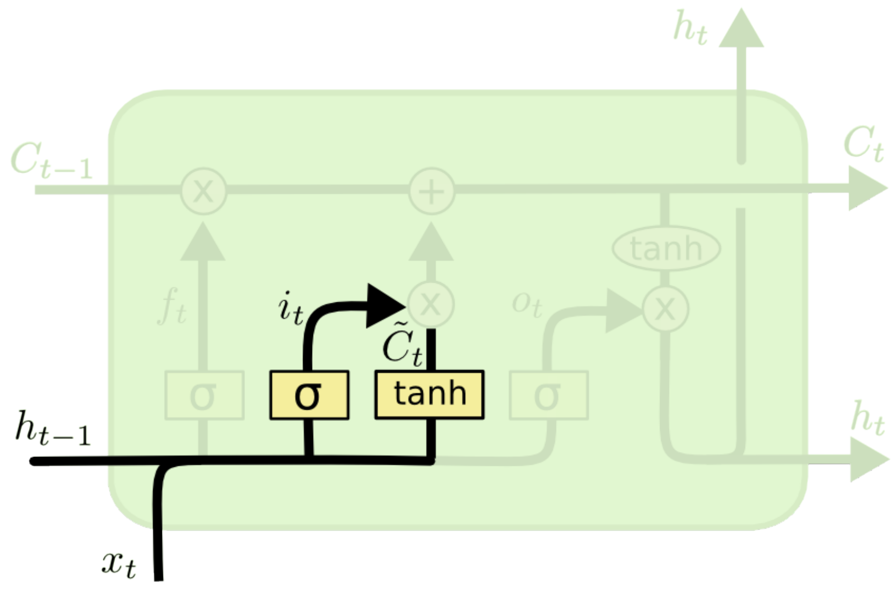
- The first step in our LSTM is to decide what information we're going to throw away from the cell state.
- This decision is made by a sigmoid layer called the "forget gate layer."
- Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



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

Step-by-Step LSTM Walk Through (Input Gate)

- The next step is to decide what new information we're going to store in the cell state.
- This has two parts:
    1. a sigmoid layer called the "input gate layer" decides which values we'll update.
    2. a tanh layer creates a vector of new candidate values, to be added to the state.
- In the next step, we'll combine these two to create an update to the state.
- In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.

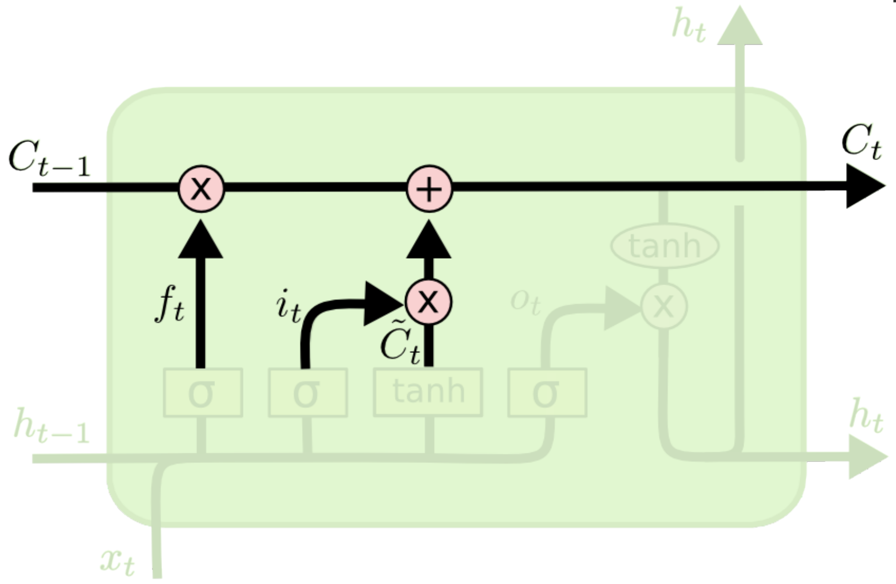


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

UNC CHARLOTTE

Step-by-Step LSTM Walk Through (Input Gate)

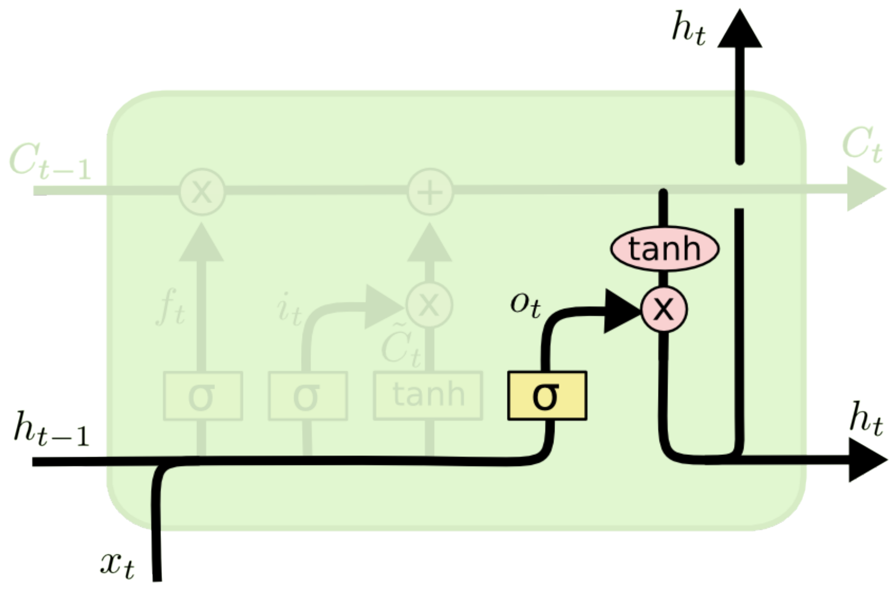
- It's now time to update the old cell state, into the new cell state
- The previous steps already decided what to do, we just need to actually do it.
- We multiply the old state by, forgetting the things we decided to forget earlier.
- In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



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

Step-by-Step LSTM Walk Through (Output Gate)

- Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version.
- First, we run a sigmoid layer which decides what parts of the cell state we're going to output.
- Then, we put the cell state through tanhtanh (to push the values to be between −1−1 and 11) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.
- For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

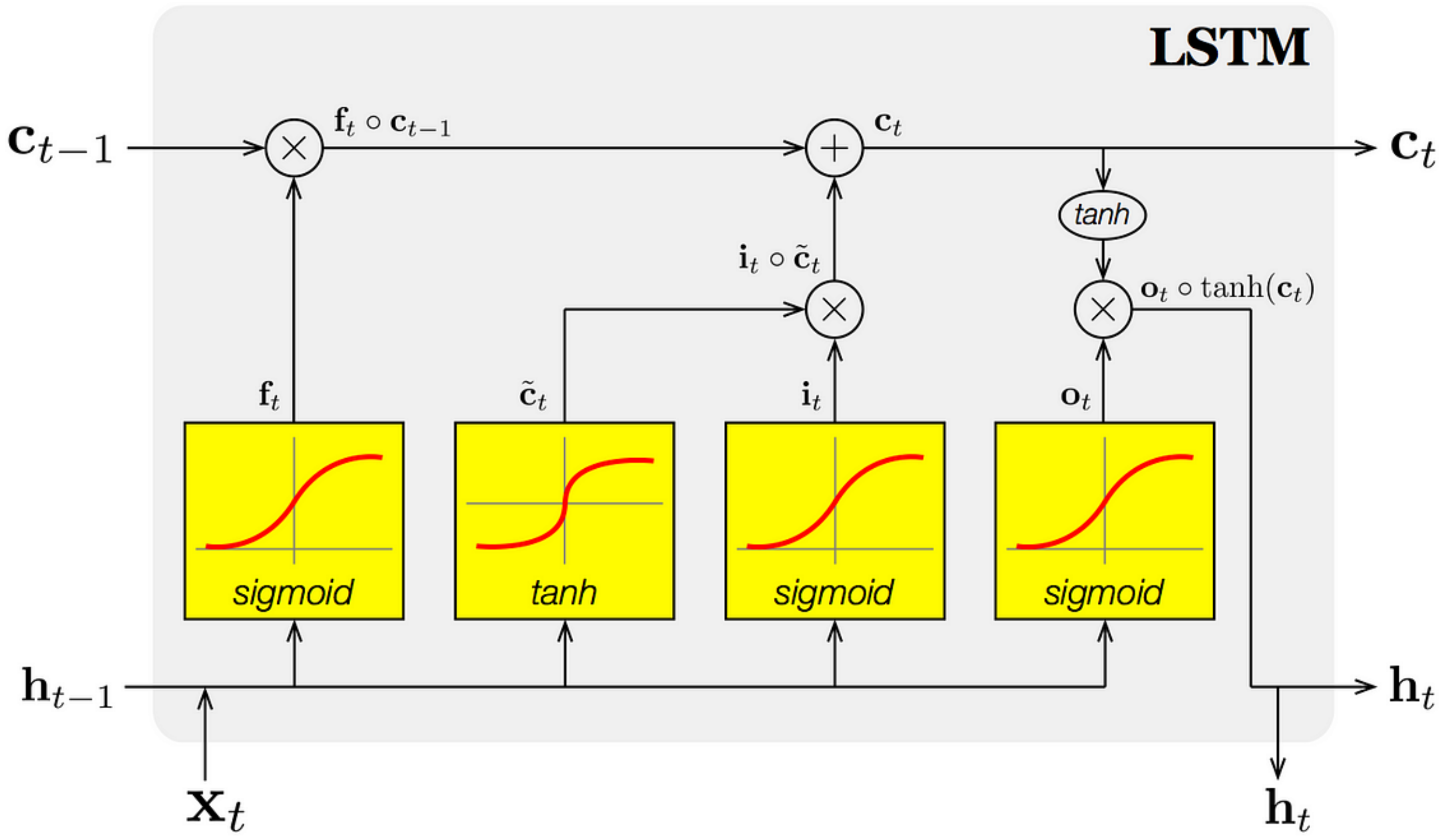


$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

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

UNC CHARLOTTE

LSTM summary



Gating variables

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

Candidate (memory) cell state

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

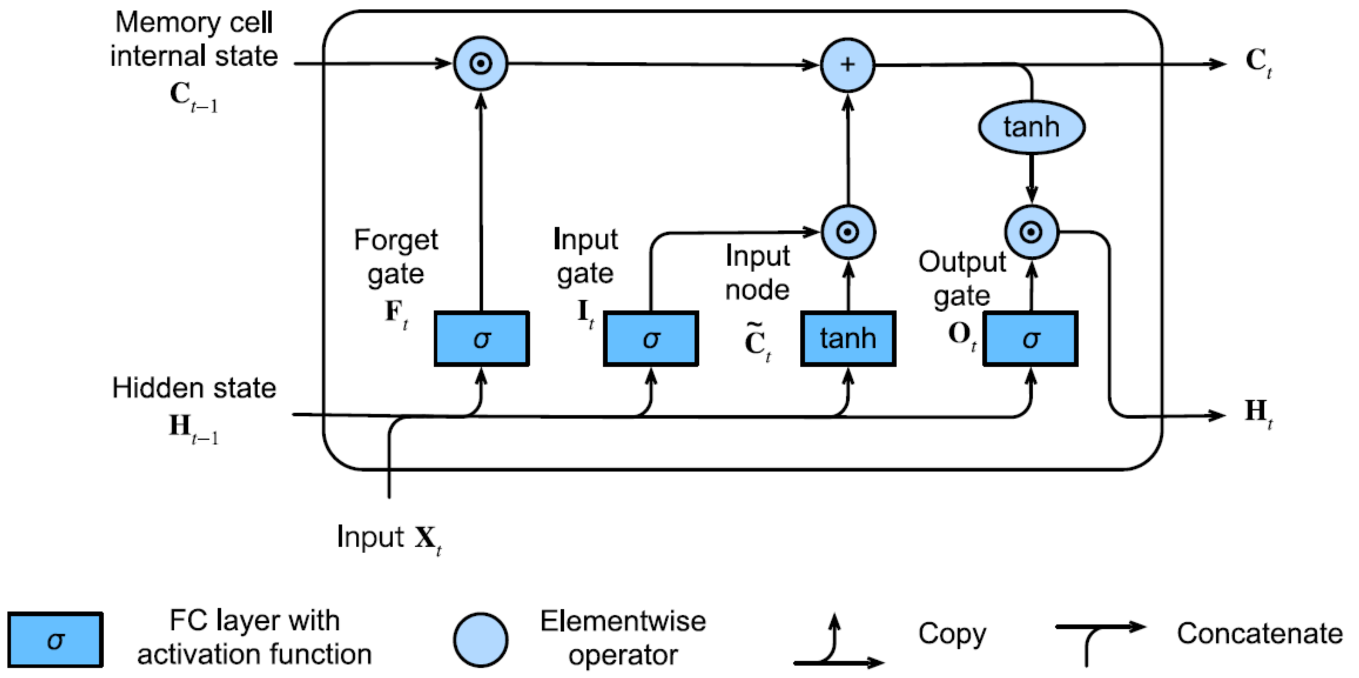
Cell & Hidden state

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tilde{\mathbf{c}}_t$$
$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t)$$

UNC CHARLOTTE

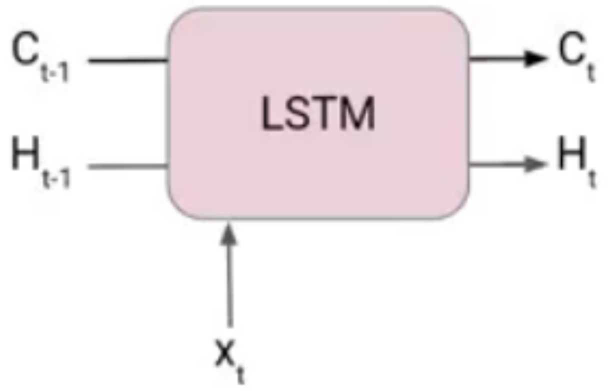
LSTM summary

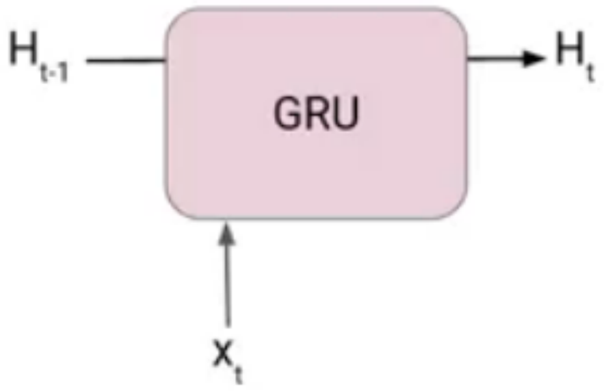
Forget Gate: conditionally decides what information to throw away from the block
•**Input Gate**: conditionally decides which values from the input to update the memory state
•**Output Gate**: conditionally decides what to output based on input and the memory of the block


Memory cell internal state $\mathbf{C}_{t-1}$
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
tanh
+
$\mathbf{C}_t$
$\mathbf{H}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

UNC CHARLOTTE

GRU vs LSTM





Gated Recurrent Units (GRU)

- The gated recurrent unit (GRU) (Cho *et al.*, 2014) offered a streamlined version of the LSTM memory cell that often achieves comparable performance but with the advantage of being faster to compute (Chung *et al.*, 2014).
- LSTM Gates:
  1. **Forget Gate:**
  2. **Input Gate:**
  3. **Output Gate:**

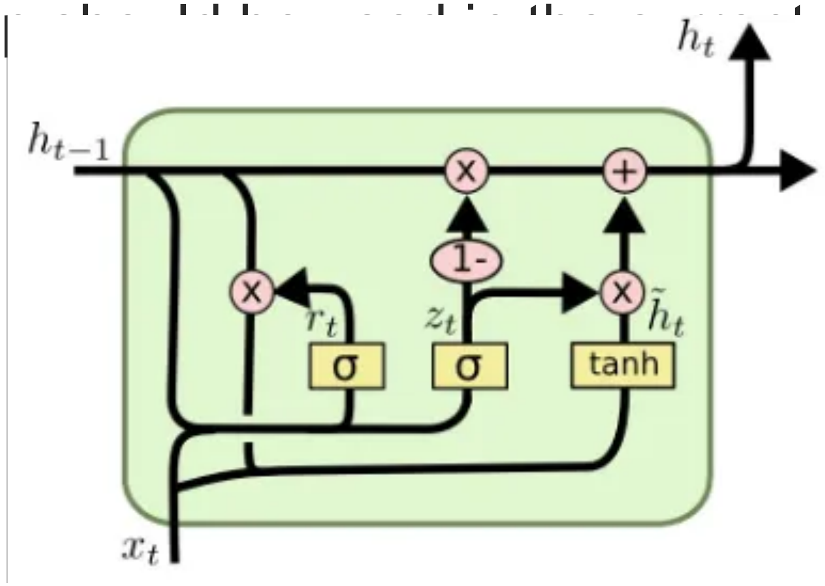
- In GRU, the LSTM's three gates are replaced by two:
  1. ***reset gate***
  2. ***update gate***

- As with LSTMs, GRU gates are given sigmoid activations, forcing their values to lie in the interval (0*;* 1).
- **Reset gate** controls how much of the previous state we might still want to remember.
- **Update gate** would allow us to control how much of the new state is just a copy of the old state. `

GRU Explained

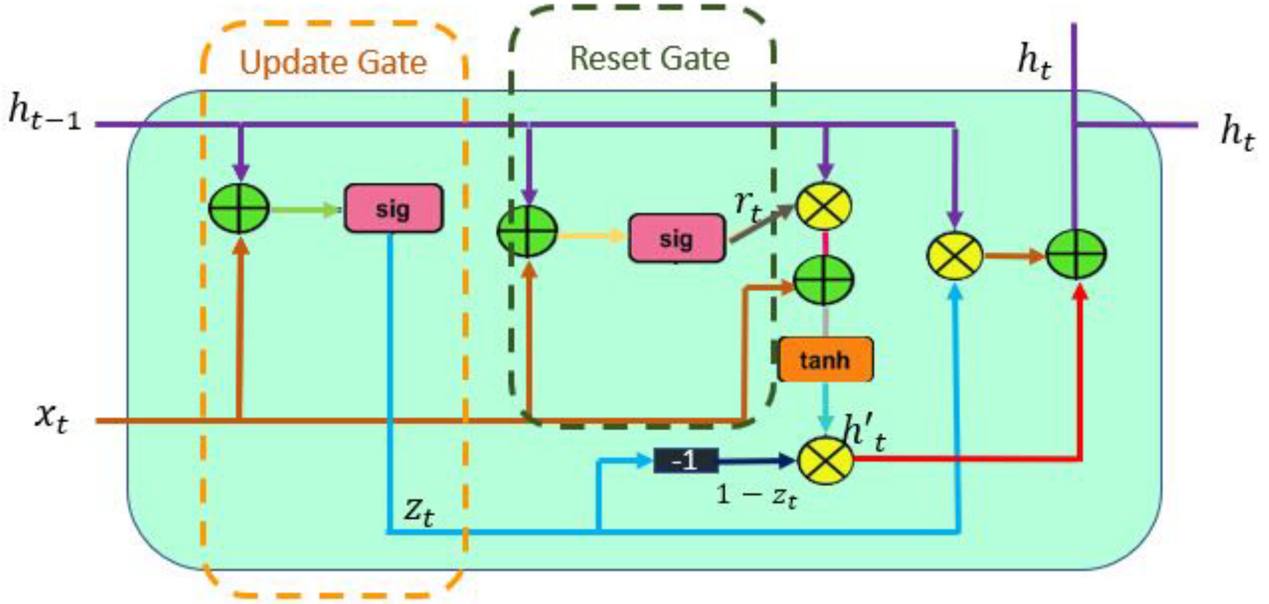
- GRU only passes along its important internal state at each time step.

1. Reset gate to determine what should be removed from the cell's internal state before passing itself along to the next time step,
2. Update gate to determine how much of the state from the previous time ste ime step.

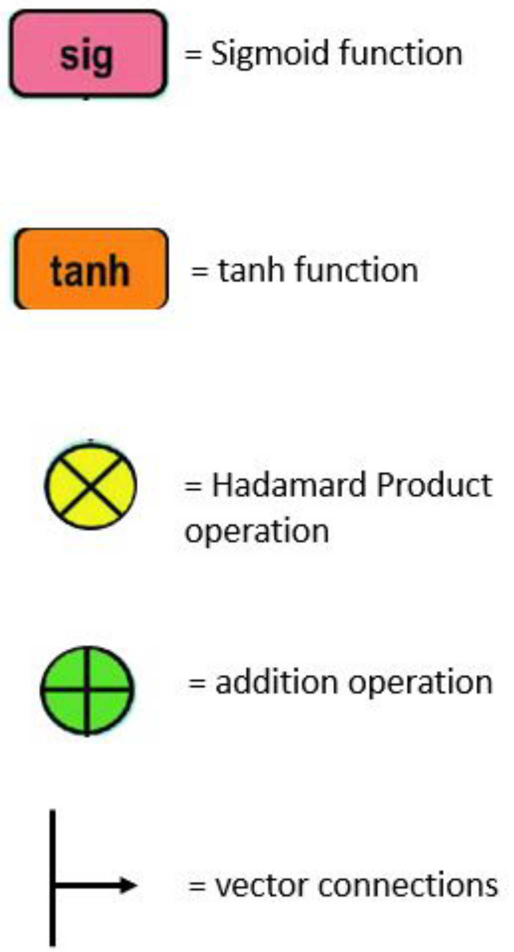


- In practice, GRUs tend to have a slight advantage over LSTMs in many use cases, especially when GRU cells are a bit simpler than LSTM cells
- The best thing to do is to build a model with each and see which one does better

GRU Formal Description

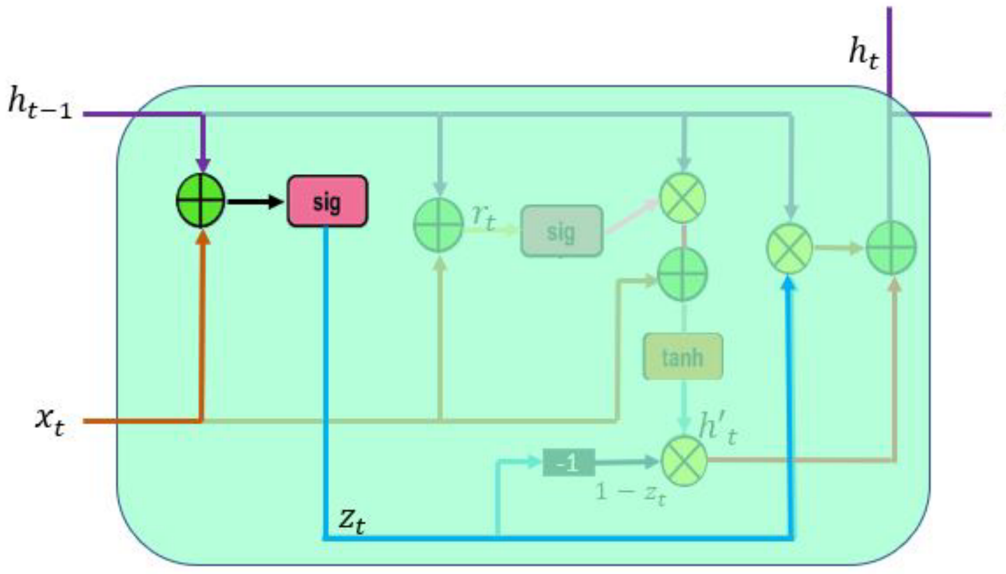


Gated Recurrent Network (GRU)



GRU – Update Gate

The update gate (z_t) is responsible for determining the amount of previous information (prior time steps) that needs to be passed along the next state. It is an important unit. The below schema shows the arrangement of the update gate.



Update Gate Operation

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

$t = timestep$

$z_t = Update\ gate\ at\ t$

$x_t = input\ vector$

$h_{t-1} = Previous\ hidden\ state$

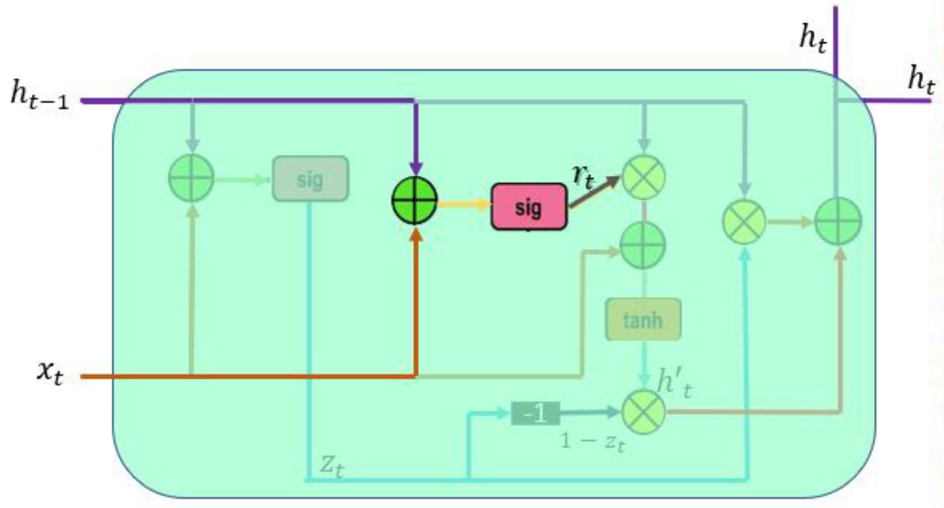
$W_z = Update\ gate\ weight\ matrix$

$b_z = connection\ bias\ at\ t$

UNC CHARLOTTE

GRU Reset Gate

The reset gate (r_t) is used from the model to decide how much of the past information is needed to neglect. The formula is the same as the update gate. There is a difference in their weights and gate usage.



Reset Gate Operation

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

$t = timestep$

$r_t = Reset\ gate\ at\ t$

$x_t = input\ vector$

$h_{t-1} = Previous\ hidden\ state$

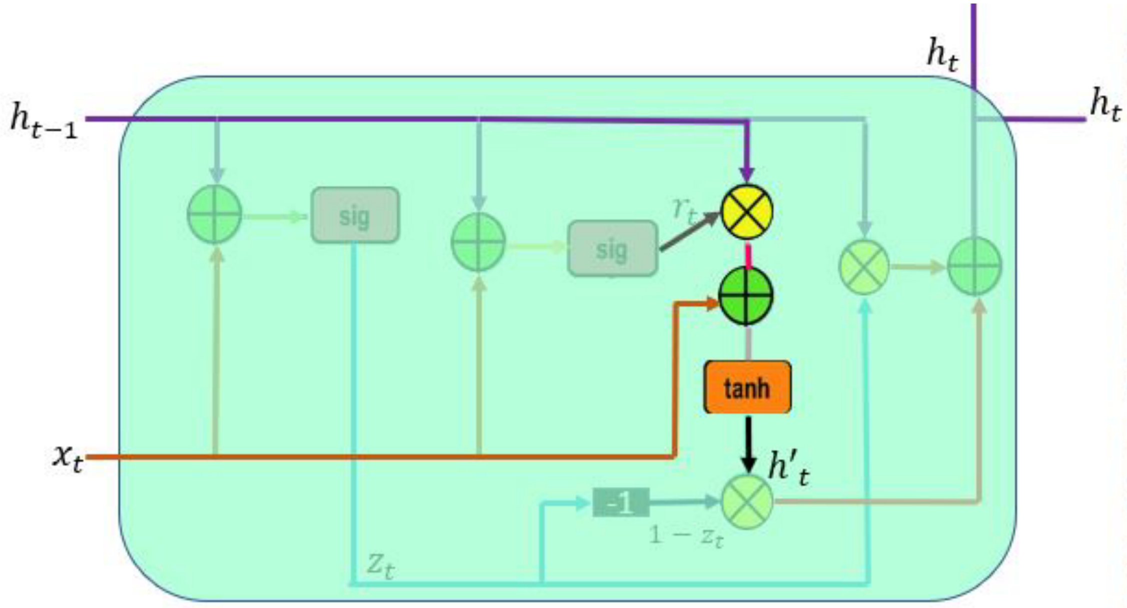
$W_r = Reset\ gate\ weight\ matrix$

$b_r = connection\ bias\ at\ t$

UNC CHARLOTTE

GRU Gates in Action

- It calculates element-wise multiplication (Hadamard) between the reset gate and previously hidden state multiple.
- After summing up, the above steps non-linear activation function is applied to results, and it produces h'_t.



Current Memory Content

$$\boxed{h'_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h)}$$

$t = timestep$

$h'_t = Current\ Memory\ function$

$r_t = Reset\ gate\ at\ t$

$x_t = input\ vector$

$h_{t-1} = Previous\ hidden\ state$

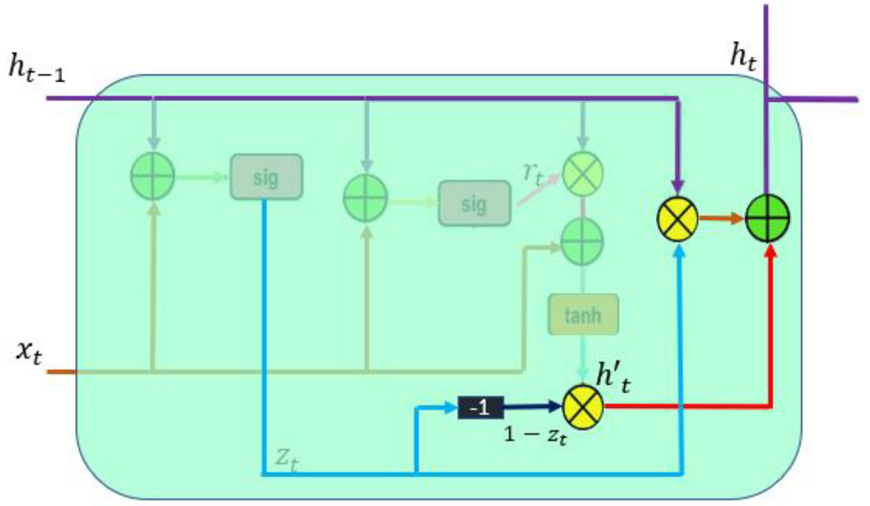
$W_h = Reset\ gate\ weight\ matrix$

$b_h = connection\ bias\ at\ t$

UNC CHARLOTTE

GRU Gates in Action

- The last step at the current time, the network needs to calculate h_t.
- Here, the update gate will play a vital role. This vector value will hold information for the current unit and pass it down to the network. It will determine which information to collect from current memory content (h't) and previous timesteps h(t-1).
- Element-wise multiplication (Hadamard) is applied to the update gate and h(t -1), and summing it with the Hadamard product operation between (1-z_t) and h'(t).



Final Memory Content

$$\boxed{h_t = (1 - z_t) \odot h_{t-1} + z_t \odot h'_t}$$

$t = timestep$

$h_t = hidden\ layer\ vectors$

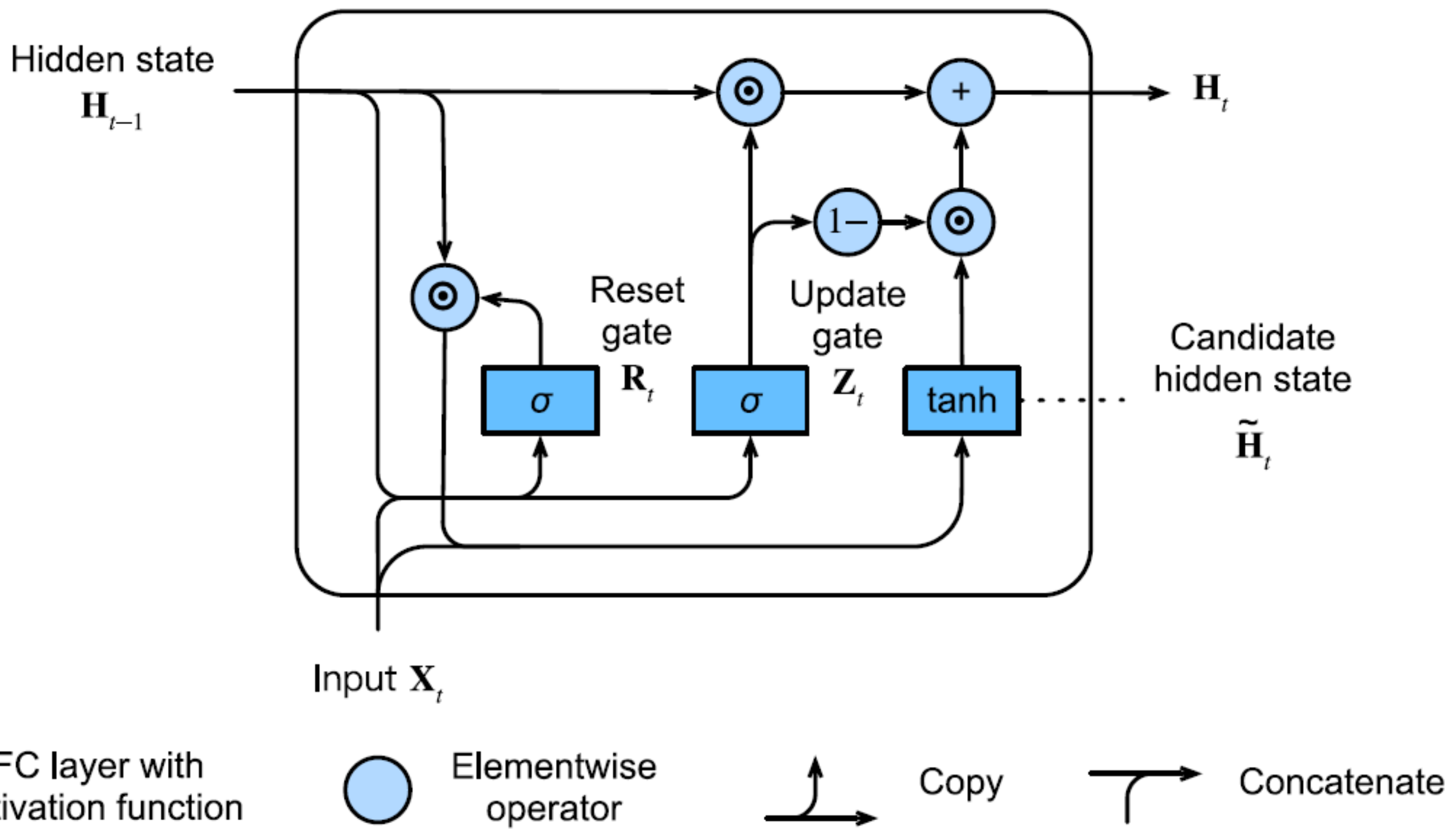
$z_t = Update\ gate\ at\ t$

$h'_t = Current\ Memory\ function$

$h_{t-1} = Previous\ hidden\ state$

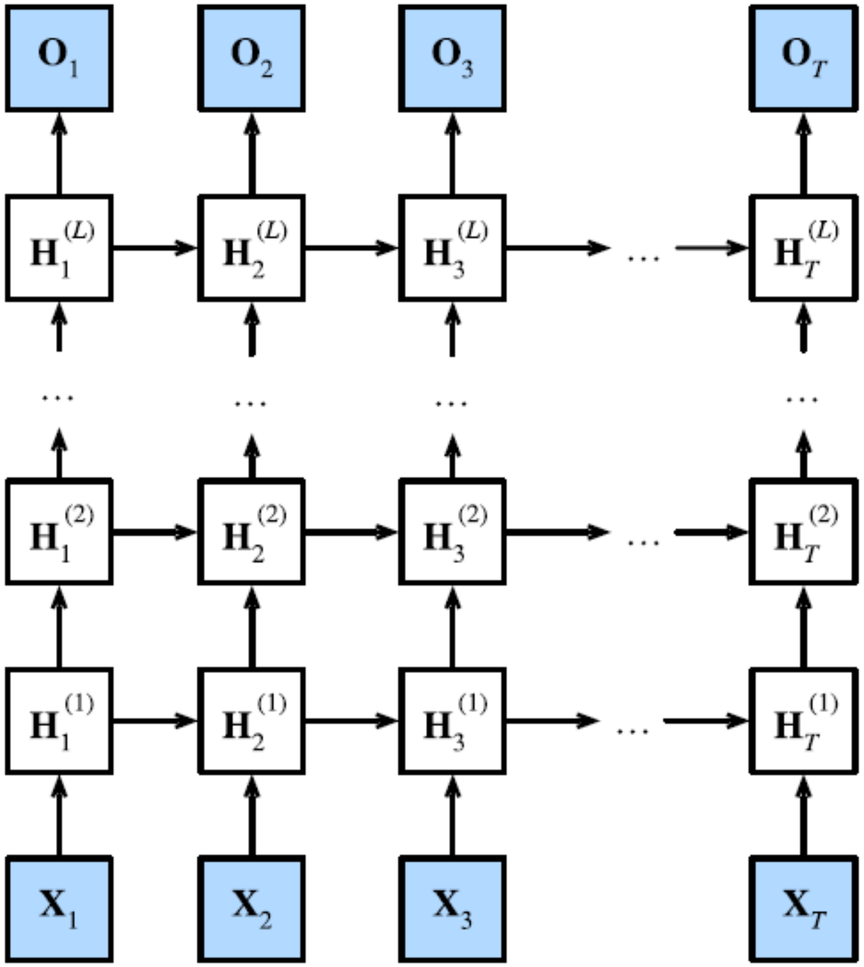
UNC CHARLOTTE

GRU Summary



Hidden state
$\mathbf{H}_{t-1}$
$\mathbf{H}_t$
1−
+
Reset gate
$\mathbf{R}_t$
Update gate
$\mathbf{Z}_t$
σ
σ
tanh
Candidate hidden state
$\tilde{\mathbf{H}}_t$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

Deep RNNs



- Deep RNN with *L* hidden layers.
- Each hidden state operates on a sequential input and produces a sequential output.
- Moreover, any RNN cell at each time step depends on both the same layer's value at the previous time step and the previous layer's value at the same time step.