Νευρο-Ασαφής Υπολογιστική
Neuro-Fuzzy Computing

Διδάσκων –
Δημήτριος Κατσαρός

@ Τμ. ΗΜΜΥ
Πανεπιστήμιο Θεσσαλίας

Διάλεξη 23η

Modern RNN:
Gated Recurrent Units (GRU)
Long-Short Term Memory (LSTM)

Gated Recurrent Units

- We have found that long products of matrices can lead to vanishing or divergent gradients
- Let us briefly think about what such gradient anomalies mean:
  - We might encounter a situation where an early observation is highly significant for predicting all future observations
    - Consider the somewhat contrived case where the first observation contains a checksum and the goal is to discern whether the checksum is correct at the end of the sequence
  - We might encounter situations where some symbols carry no pertinent observation
    - For instance, when parsing a web page there might be auxiliary HTML code that is irrelevant for the purpose of assessing the sentiment conveyed on the page. We would like to have some mechanism for *skipping* such symbols in the latent state representation
  - We might encounter situations where there is a logical break between parts of a sequence
    - For instance, there might be a transition between chapters in a book, or a transition between a bear and a bull market for securities. In this case it would be nice to have a means of *resetting* our internal state representation.

Gating the hidden state

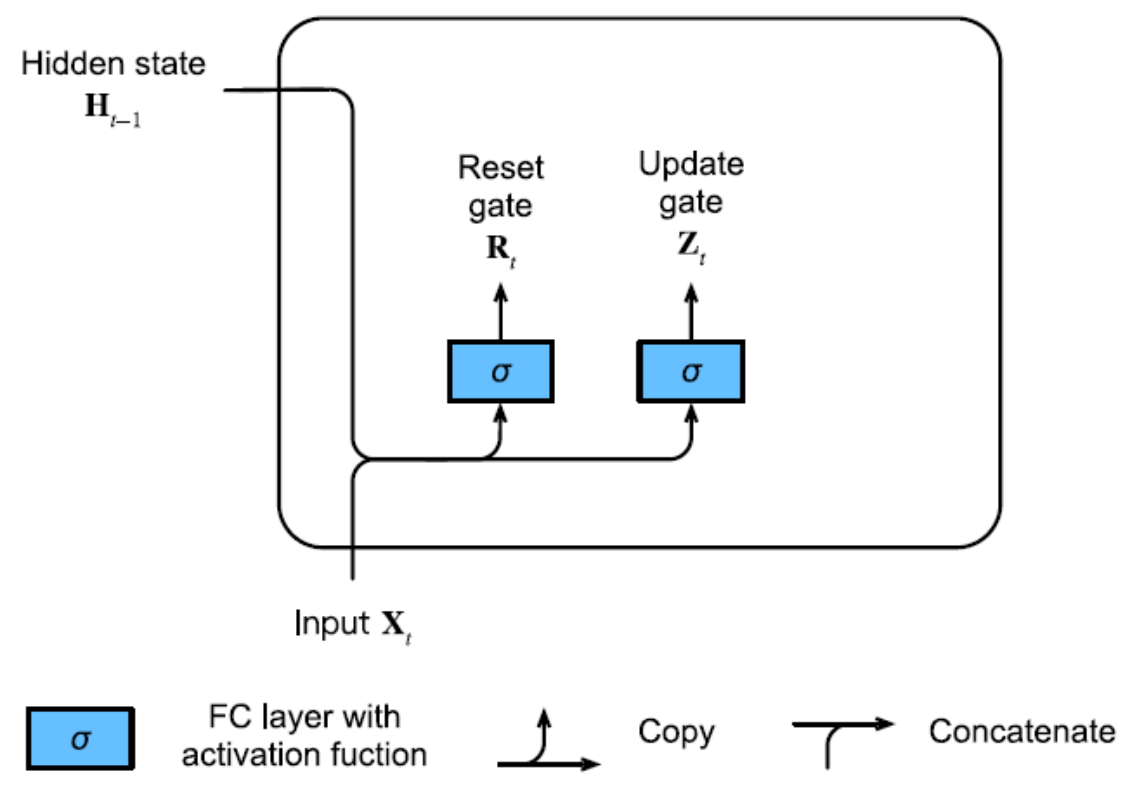
- The key distinction between regular RNNs and GRUs is that the latter support gating of the hidden state
- This means that we have dedicated mechanisms for when a hidden state should be updated and also when it should be reset
- These mechanisms are learned and they address the concerns listed earlier
  - For instance, if the first symbol is of great importance we will learn not to update the hidden state after the first observation
  - Likewise, we will learn to skip irrelevant temporary observations
  - Last, we will learn to reset the latent state whenever needed

Reset gates and Update gates

- The first thing we need to introduce are reset and update gates
  - We engineer them to be vectors with entries in (0; 1) such that we can perform convex combinations
    - For instance, a reset variable would allow us to control how much of the previous state we might still want to remember
    - Likewise, an update variable would allow us to control how much of the new state is just a copy of the old state
- We begin by engineering gates to generate these variables

Reset gates and Update gates

The inputs for both reset and update gates in a GRU, given the current time step input $\mathbf{X}_t$ and the hidden state of the previous time step $\mathbf{H}_{t-1}$. The output is given by a fully connected layer with a sigmoid as its activation function



The reset gate $\mathbf{R}_t$ and update gate $\mathbf{Z}_t$ are computed as follows:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r)$$

$$\mathbf{Z}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z)$$

Reset gates in action

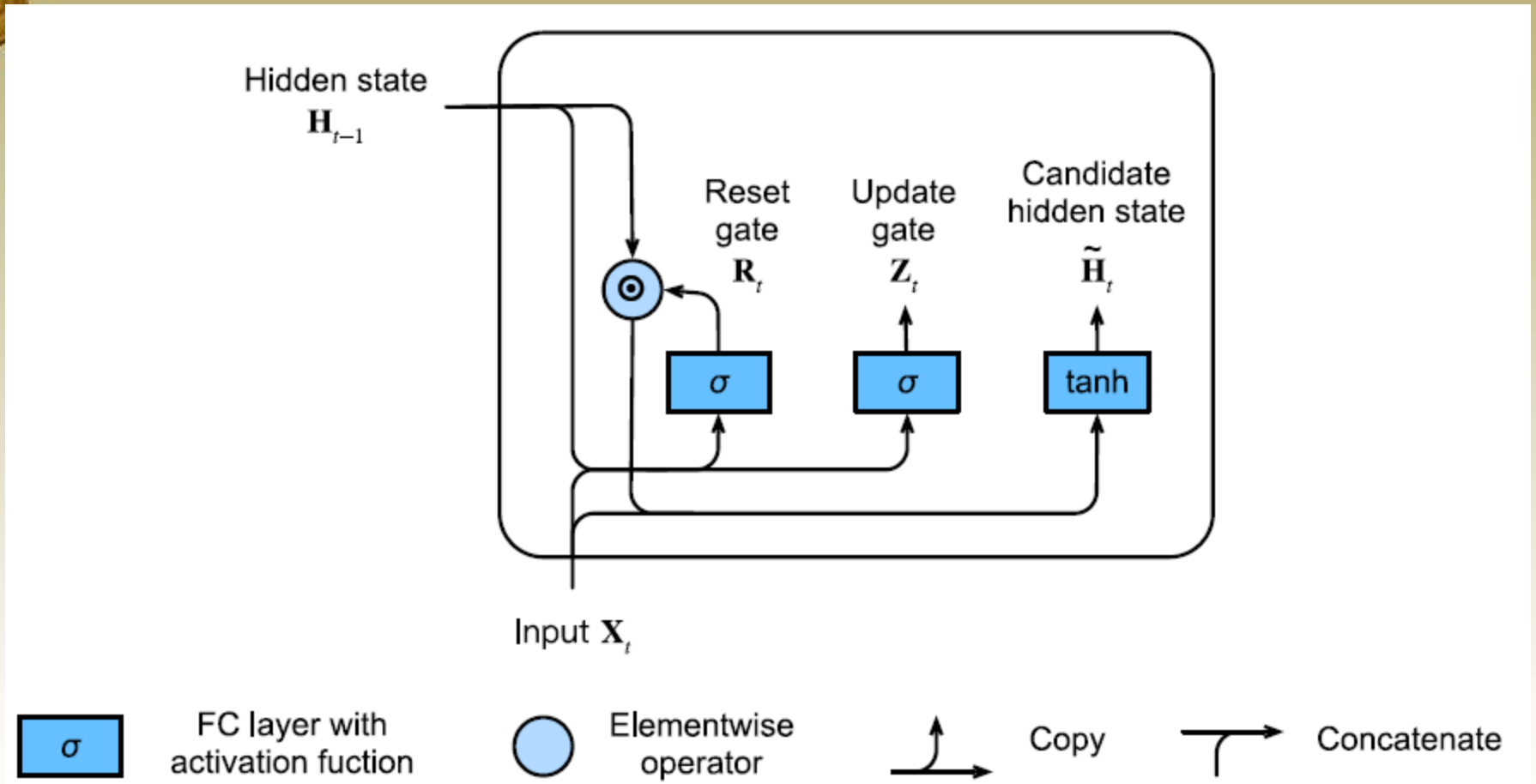
- We begin by integrating the reset gate with a regular latent state updating mechanism. In a conventional RNN, we would have an hidden state update of the form:

$$\mathbf{H}_t = tanh(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h)$$

- This is essentially identical to the previous discussion, albeit with a nonlinearity in the form of tanh to ensure that the values of the hidden states remain in the interval (-1, 1). If we want to be able to reduce the influence of the previous states we can multiply $\mathbf{H}_{t-1}$ with $\mathbf{R}_t$ elementwise. Whenever the entries in the reset gate $\mathbf{R}_t$ are close to 1, we recover a conventional RNN. For all entries of the reset gate $\mathbf{R}_t$ that are close to 0, the hidden state is the result of an MLP with $\mathbf{X}_t$ as input. Any pre-existing hidden state is thus reset to defaults. This leads to the following *candidate hidden state* (it is a *candidate* since we still need to incorporate the action of the update gate)

$$\tilde{\mathbf{H}}_t = tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h)$$

Reset gates in action



Candidate hidden state computation in a GRU. The multiplication is carried out elementwise.

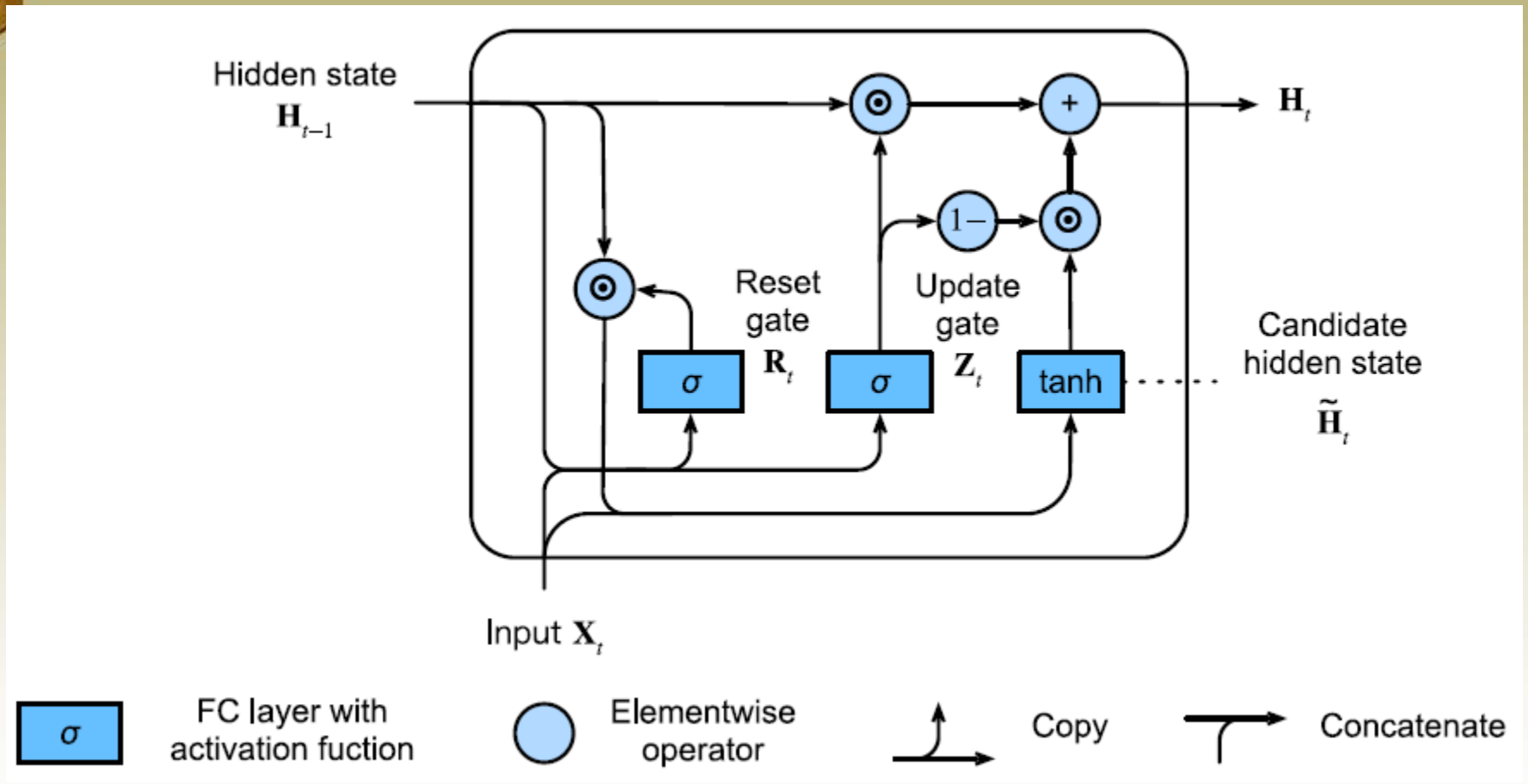
Update gates in action

- Next we need to incorporate the effect of the update gate $\mathbf{Z}_t$, as shown in next slide's figure. This determines the extent to which the new state $\mathbf{H}_t$ is just the old state $\mathbf{H}_{t-1}$ and by how much the new candidate state $\tilde{\mathbf{H}}_t$ is used. The gating variable $\mathbf{Z}_t$ can be used for this purpose, simply by taking elementwise convex combinations between both candidates. This leads to the final update equation for the GRU

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$$

- Whenever the update gate Zt is close to 1, we simply retain the old state. In this case the information from $\mathbf{X}_t$ is essentially ignored, effectively skipping time step *t* in the dependency chain. In contrast, whenever $\mathbf{Z}_t$ is close to 0, the new latent state $\mathbf{H}_t$ approaches the candidate latent state $\tilde{\mathbf{H}}_t$. These designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for time series with large time step distances. In summary, GRUs have the following two distinguishing features:
  - **Reset gates help capture short-term dependencies in time series**
  - **Update gates help capture long-term dependencies in time series**

Gated Recurrent Units



Hidden state computation in a GRU. The multiplication is carried out elementwise

Long-Short Term Memory

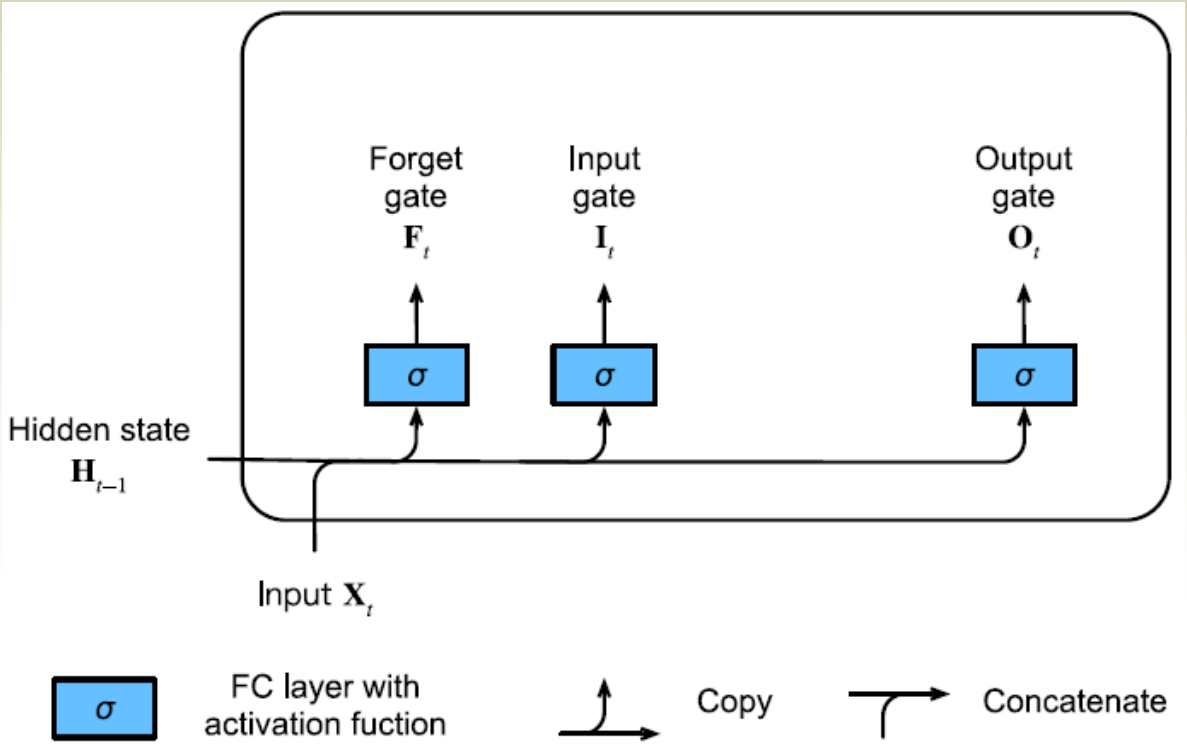
- LSTM shares many of the properties of the Gated Recurrent Unit (GRU). Interestingly, LSTM's design is slightly more complex than GRU, but predates GRU by almost two decades. Arguably it is inspired by logic gates of a computer
- To control a memory cell we need a number of gates
  - One gate is needed to read out the entries from the cell (as opposed to reading any other cell). We will refer to this as the *output* gate
  - A second gate is needed to decide when to read data into the cell. We refer to this as the *input* gate
  - Last, we need a mechanism to reset the contents of the cell, governed by a *forget* gate. The motivation for such a design is the same as before, namely to be able to decide when to remember and when to ignore inputs in the latent state via a dedicated mechanism

Long-Short Term Memory

- Three gates are introduced in LSTMs: the input gate, the forget gate, and the output gate. In addition to that we will introduce the memory cell that has the same shape as the hidden state
  - Strictly speaking this is just a fancy version of a hidden state, engineered to record additional information.

LSTM: Input, forget and output gates

- Just like with GRUs, the data feeding into the LSTM gates is the input at the current time step $\mathbf{X}_t$ and the hidden state of the previous time step $\mathbf{H}_{t-1}$. These inputs are processed by a fully connected layer and a sigmoid activation function to compute the values of input, forget and output gates. As a result, the three gates' all output values are in the range of [0, 1]



LSTM: Input, forget and output gates

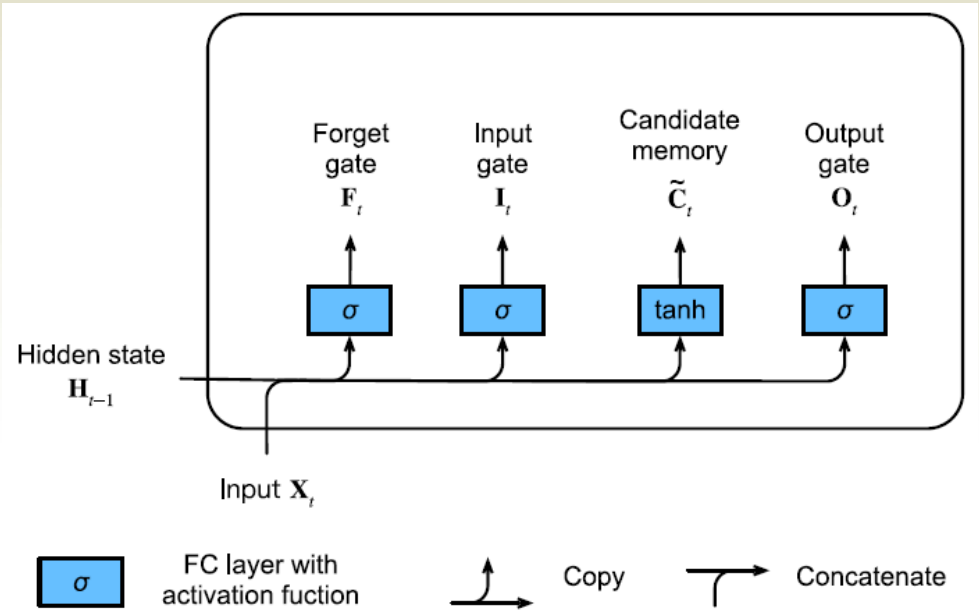
- We assume that there are h hidden units. The input is $\mathbf{X}_t$ and the hidden state of the last time step is $\mathbf{H}_{t-1}$. Correspondingly, the gates are defined as follows: the input gate is $\mathbf{I}_t$, the forget gate is $\mathbf{F}_t$, and the output gate is $\mathbf{O}_t$. They are calculated as follows:

$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i)$$
$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f)$$
$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o)$$

LSTM: Candidate memory cell

- Next we design the memory cell. Since we have not specified the action of the various gates yet, we first introduce the candidate memory cell \widetilde{\mathbf{C}}_t. Its computation is similar to the three gates described above, but using a tanh function with a value range for [-1, 1] as the activation function. This leads to the following equation at time step t

$$\widetilde{\mathbf{C}}_t = tanh(\mathbf{X}_t \mathbf{W}_{xc} + (\mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c)$$



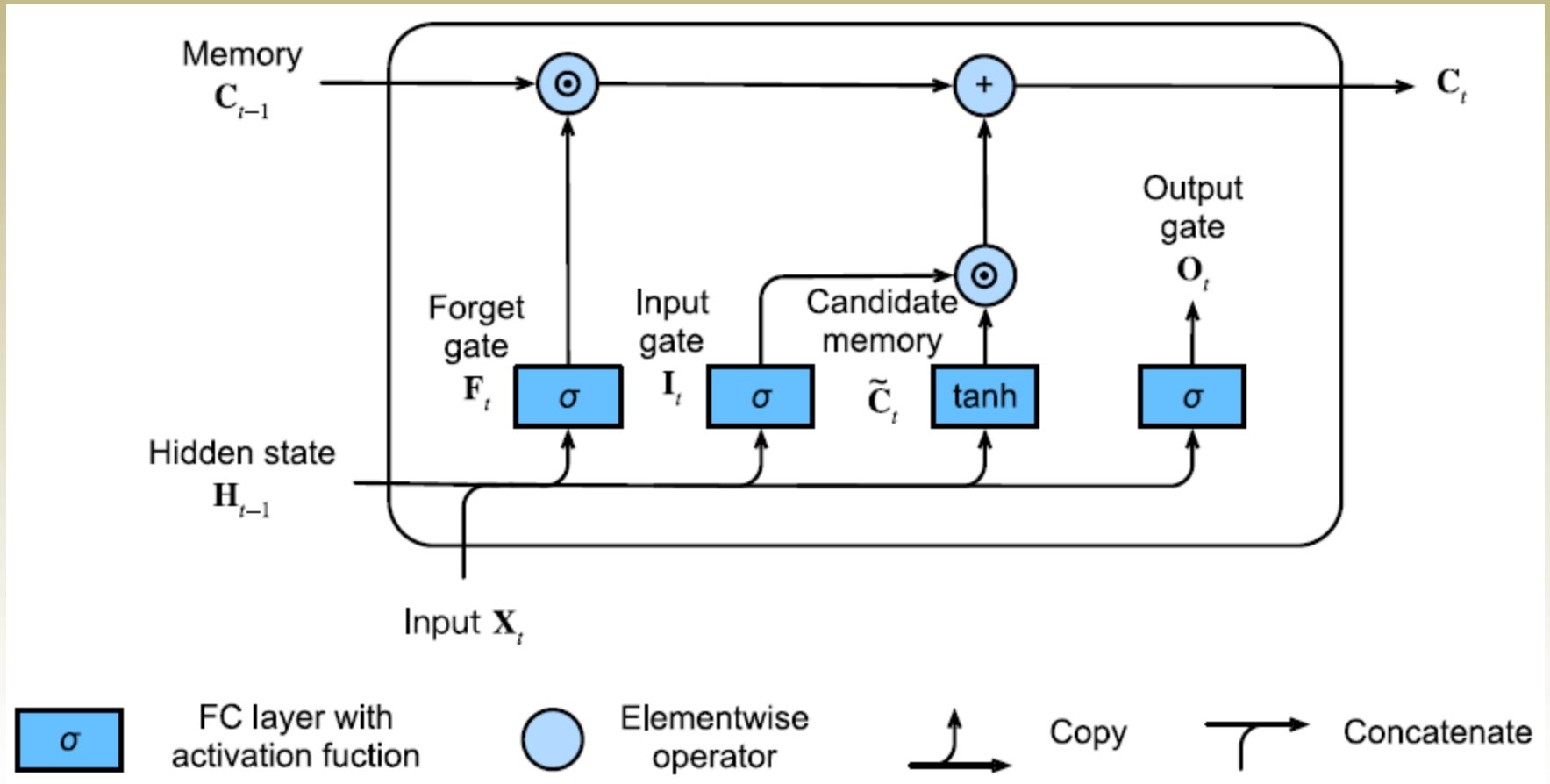
LSTM: Memory cell

- In LSTMs we have two parameters to govern input and forgetting: $\mathbf{I}_t$ which governs how much we take new data into account via \widetilde{\mathbf{C}}_t and the forget parameter $\mathbf{F}_t$ which addresses how much of the old memory cell content $\mathbf{C}_{t-1}$ we retain. Using the same pointwise multiplication trick as before, we arrive at the following update equation:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \widetilde{\mathbf{C}}_t$$

- If the forget gate is always approximately 1 and the input gate is always approximately 0, the past memory cells $\mathbf{C}_{t-1}$ will be saved over time and passed to the current time step. This design was introduced to alleviate the vanishing gradient problem and to better capture dependencies for time series with long range dependencies

LSTM: Memory cell

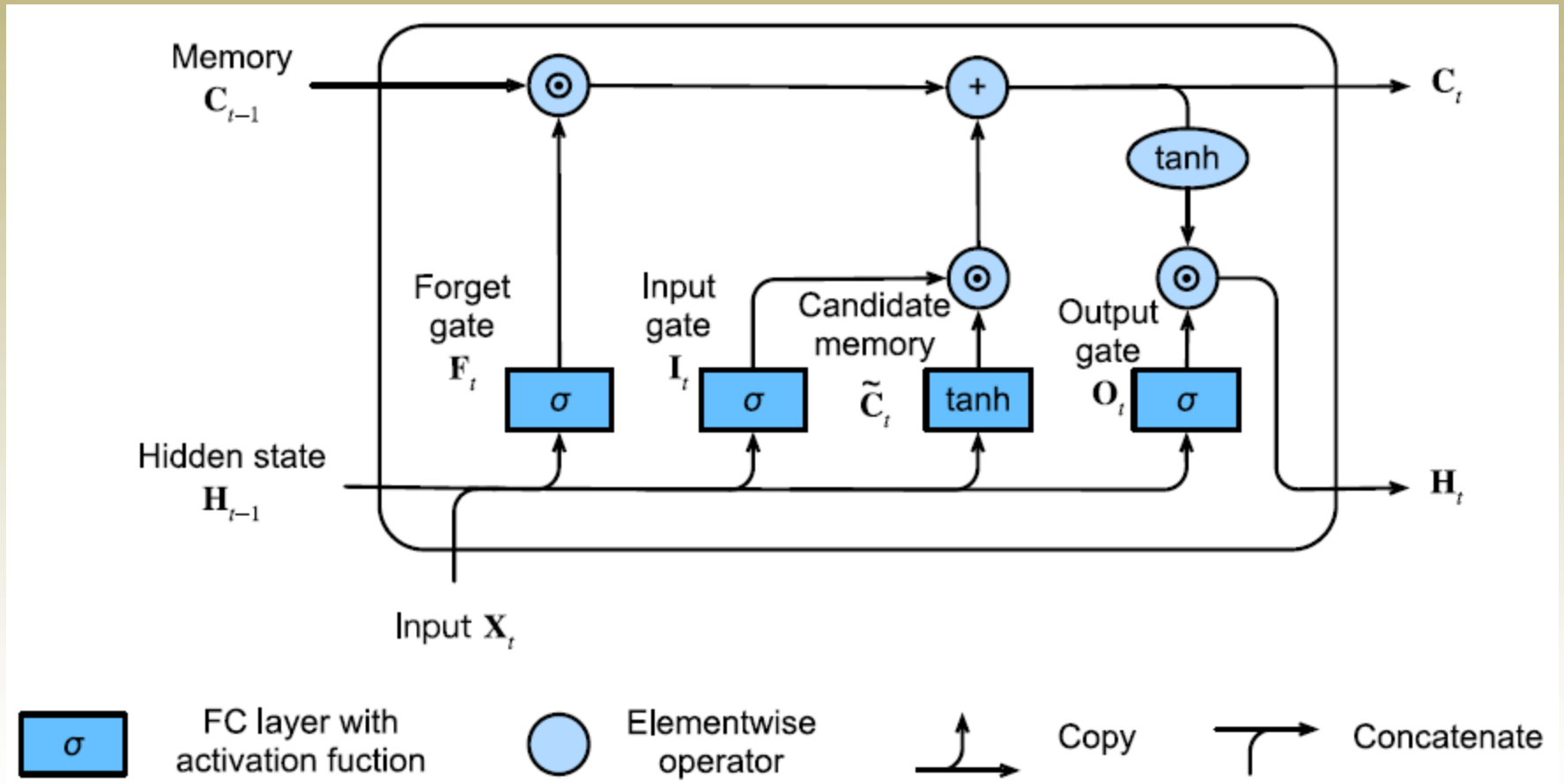


LSTM: Hidden states

- Last, we need to define how to compute the hidden state $\mathbf{H}_t$. This is where the output gate comes into play. In LSTM it is simply a gated version of the tanh of the memory cell. This ensures that the values of $\mathbf{H}_t$ are always in the interval (-1, 1)
  - Whenever the output gate is 1 we effectively pass all memory information through to the predictor, whereas for output 0 we retain all the information only within the memory cell and perform no further processing

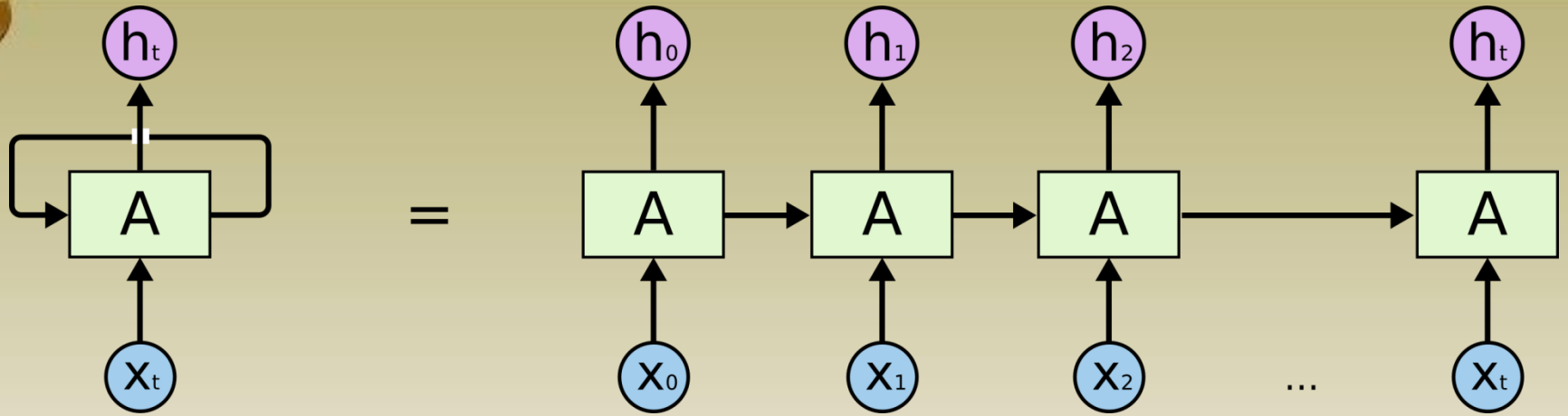
$$\mathbf{H}_t = \mathbf{O}_t \odot tanh(\mathbf{C}_t)$$

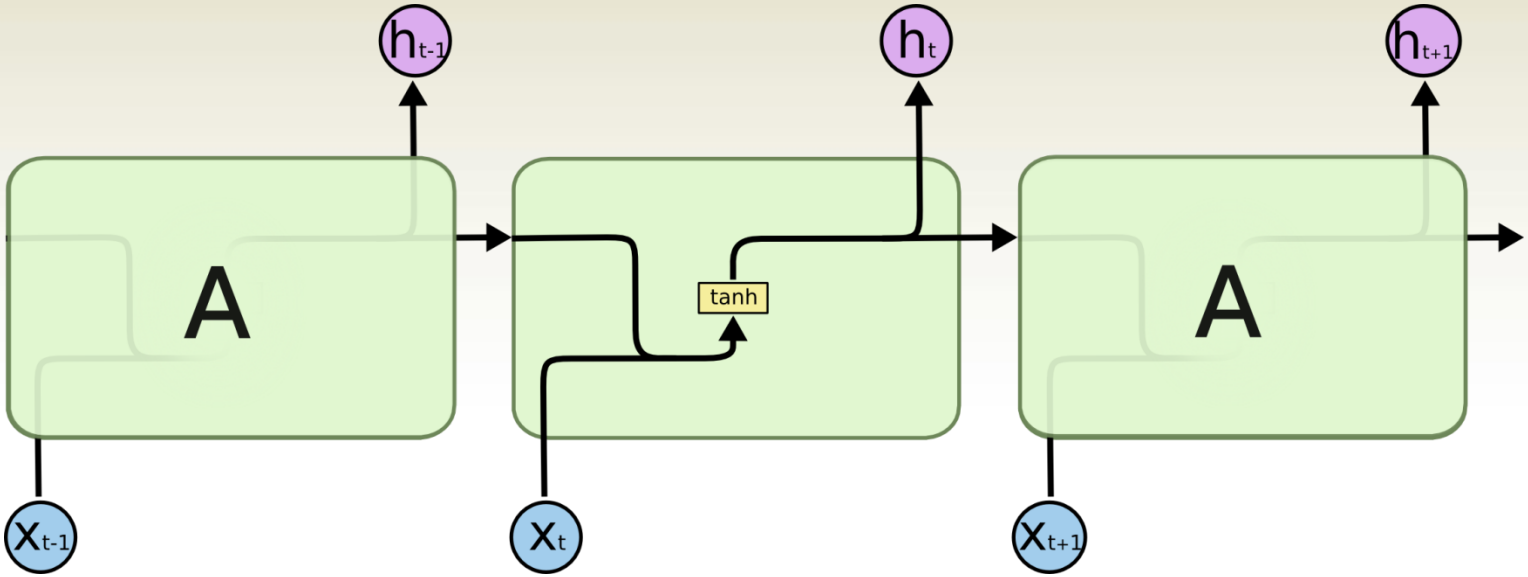
LSTM: Hidden states



Simplifying LSTM

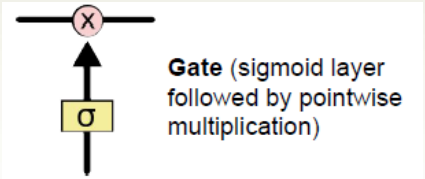
Unrolled RNN

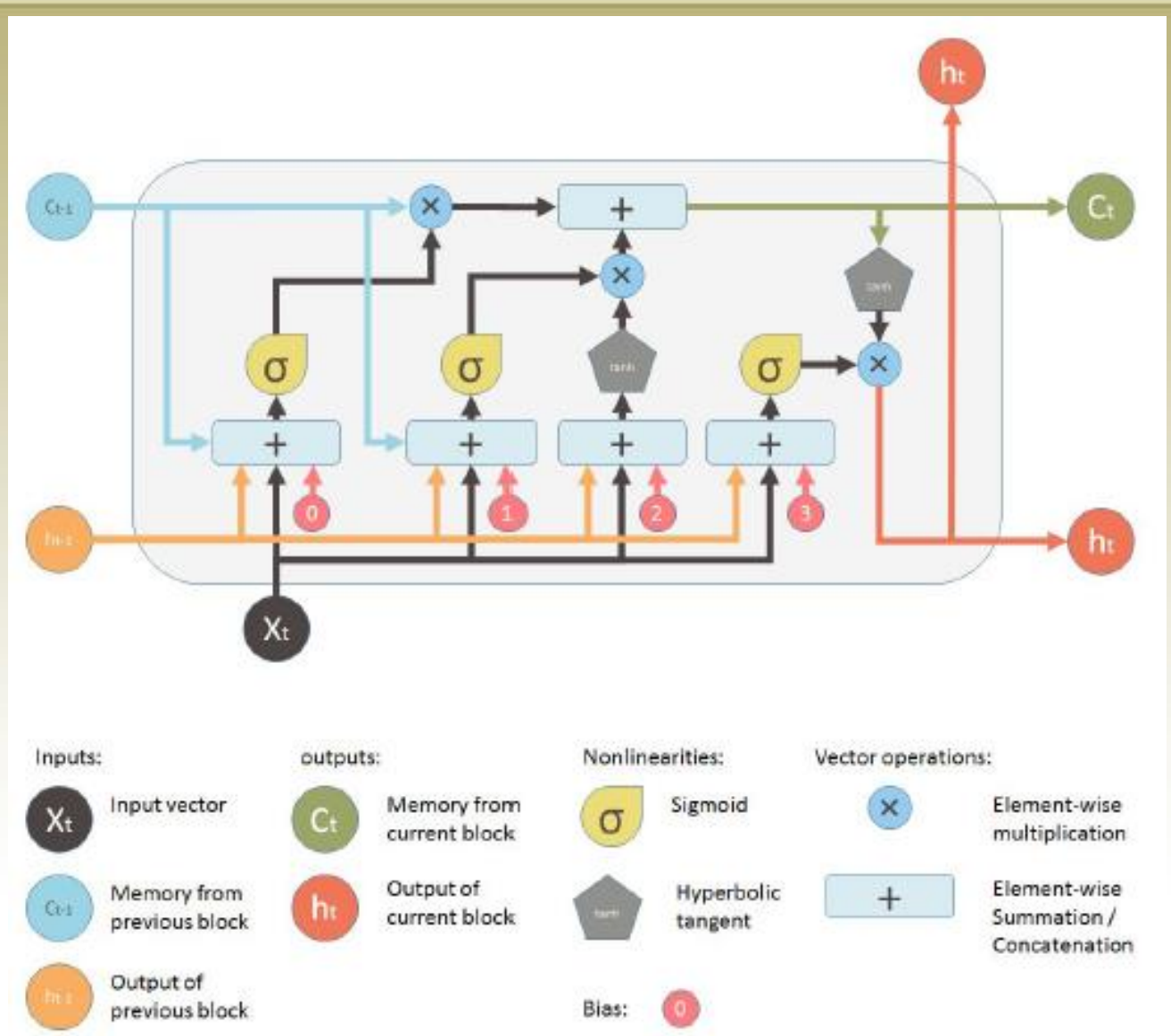




LSTM memory cell

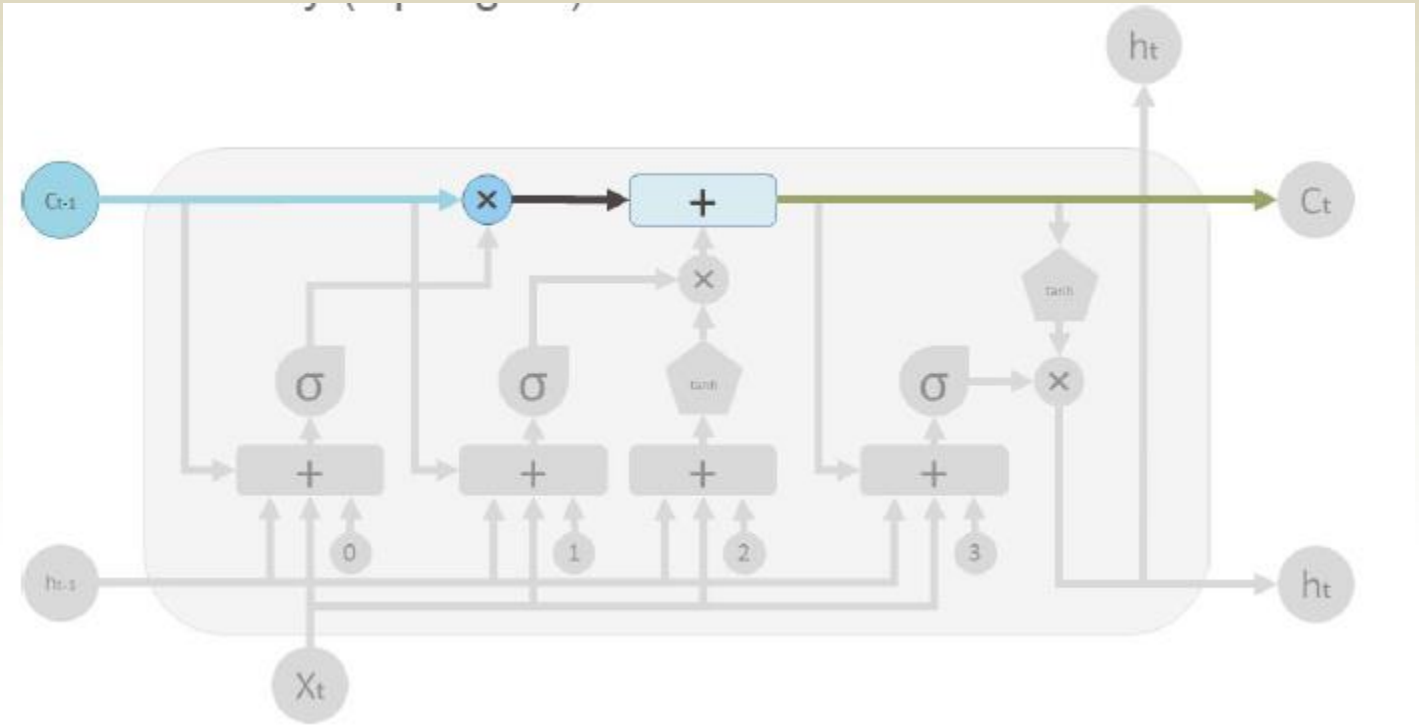
- LSTMs also have this chain like structure, but the repeating module has a different structure;
  Instead of having a single neural network layer, there are four.
- Terminology:





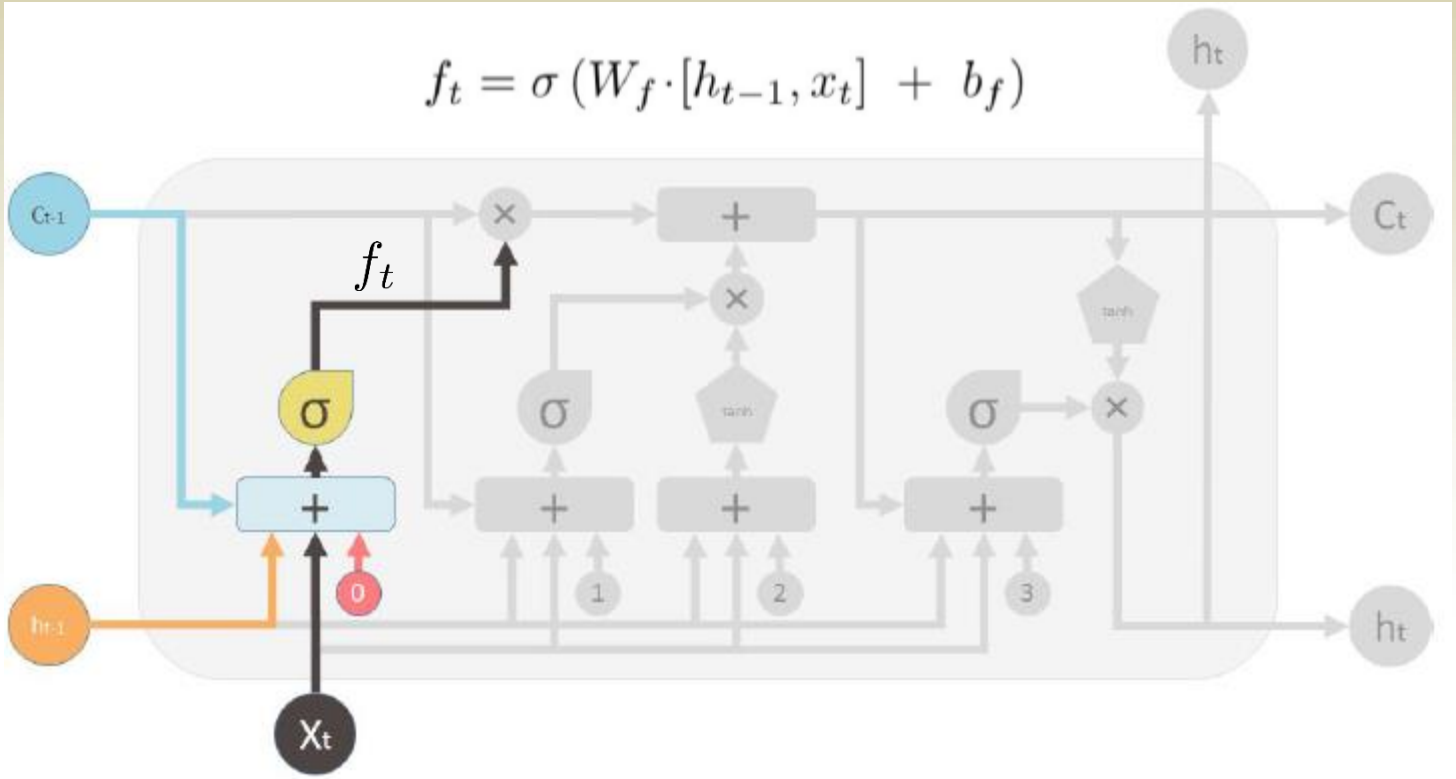
LSTM: Cell state vector

- The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state 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!"



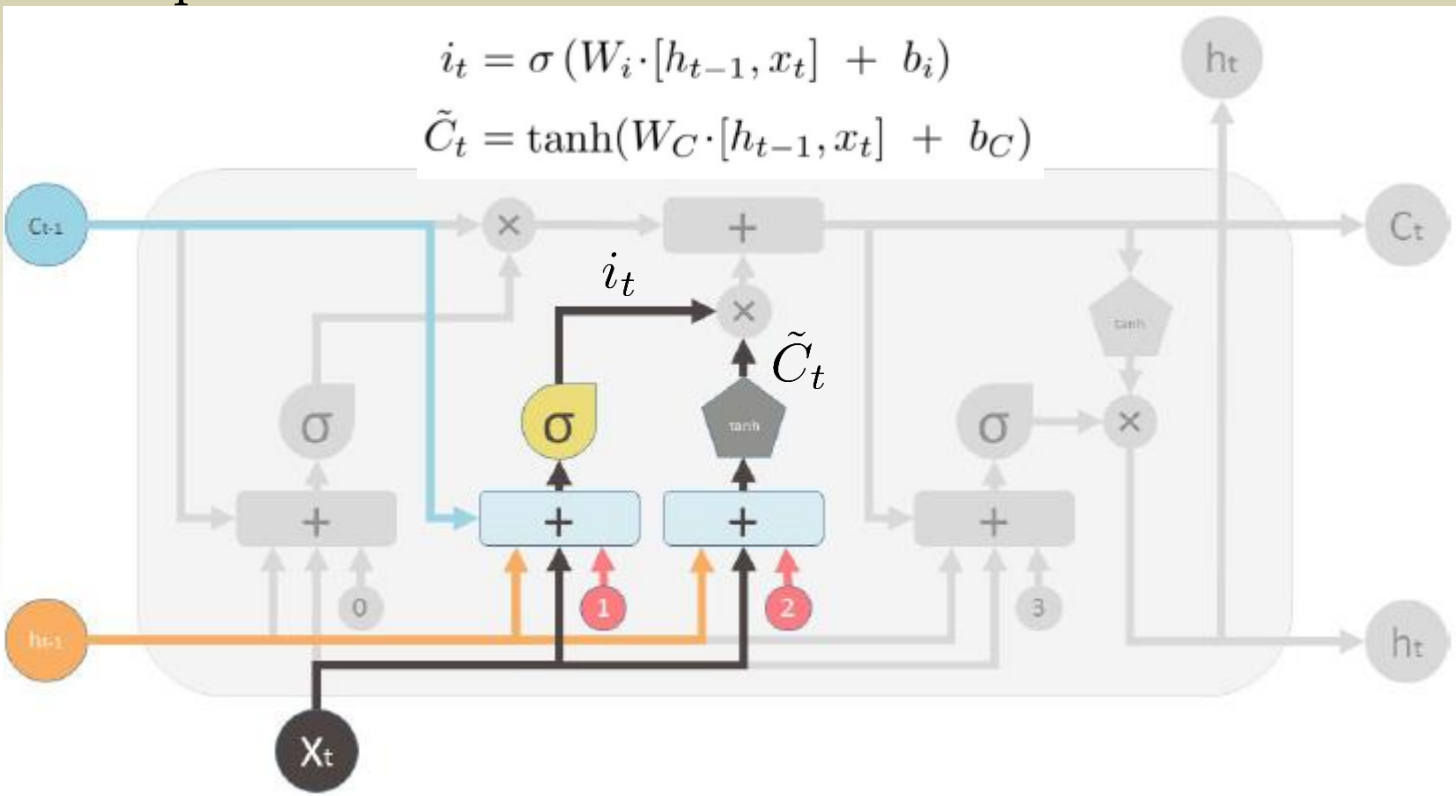
LSTM: Forget gate

- The first step in 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*". It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this"



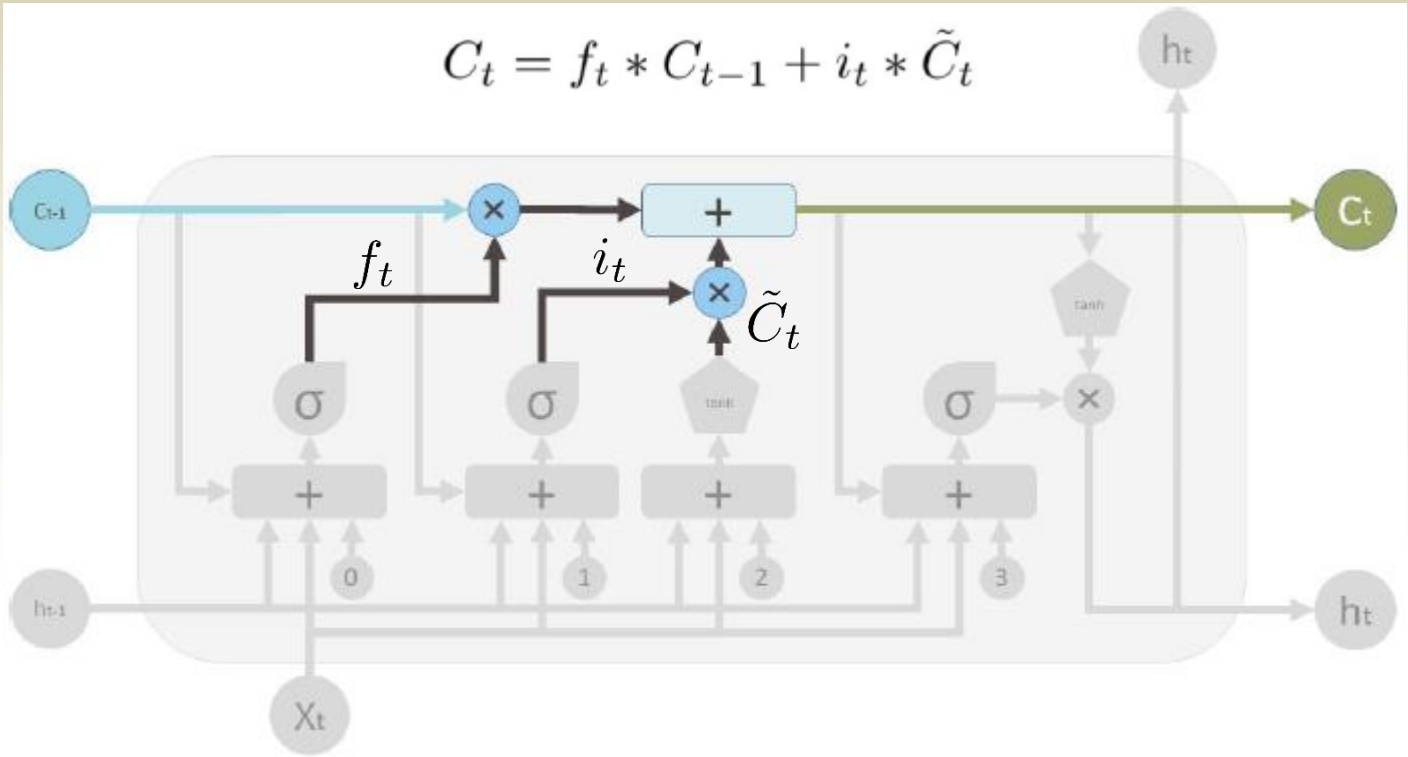
LSTM: Input gate

- The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "*input gate layer*" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, \tilde{C}$_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state



LSTM: Memory update

- It's now time to update the old cell state, C*t−1*, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it
- We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t*\tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value

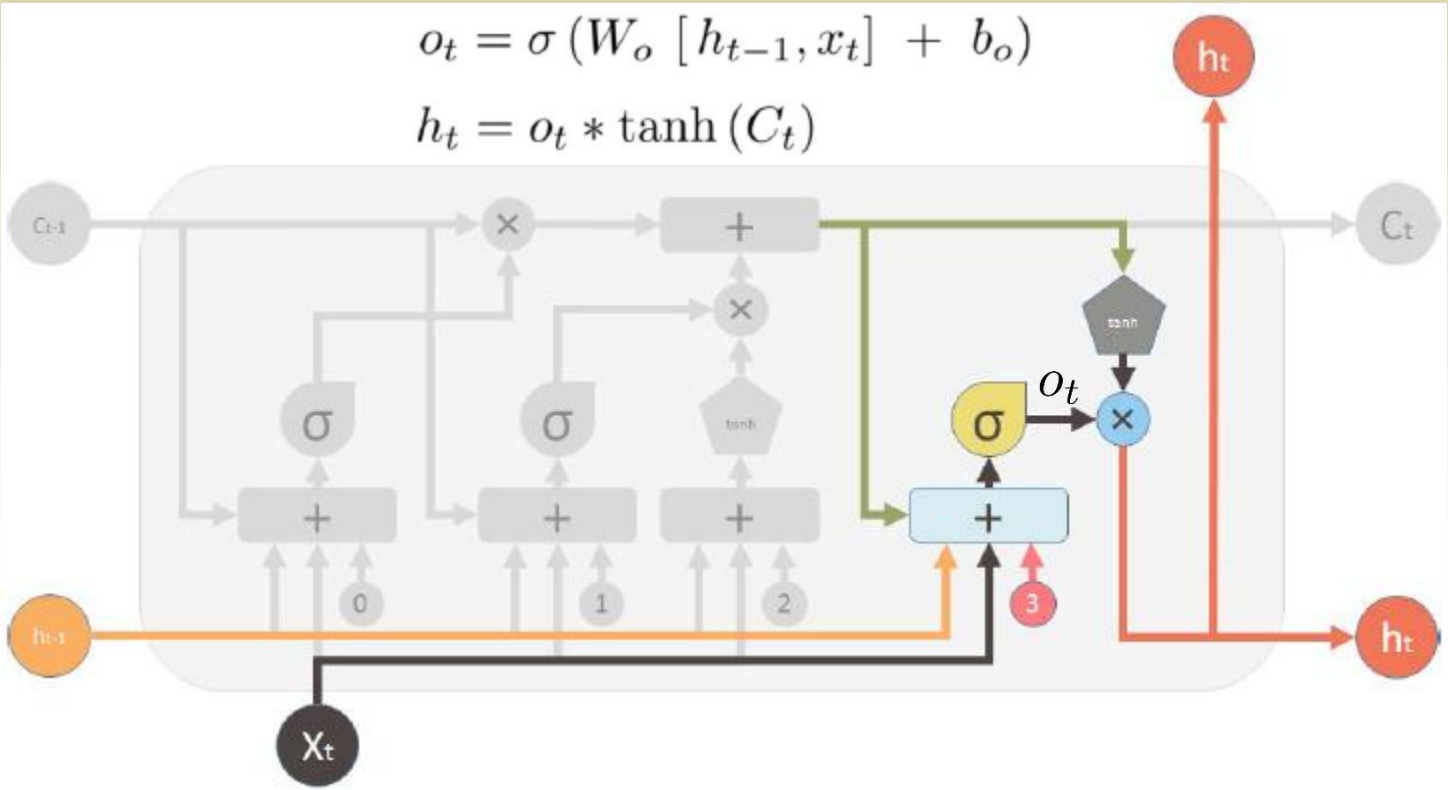


LSTM: Output gate

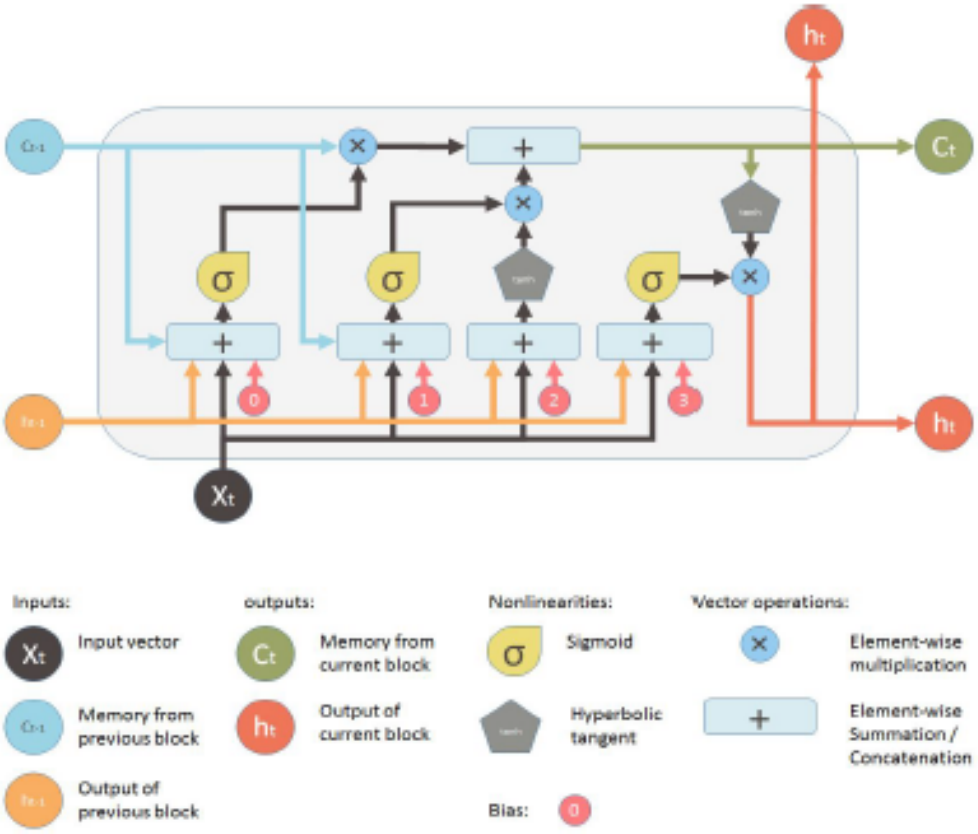
- Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to

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

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



LSTM Memory Cell Summary



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

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

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

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

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

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

Sequence of LSTM Memory Cells