

1 E D S M L  
O S O M A 2  
C E M S C O  
2 O 1 C S 2  
O 3 2 1 E 3

# Recurrent Neural Networks and Long Short-Term Memory

Carlos Cueto

1. Recurrent Neural Networks (RNNs)
2. Long Short-Term Memory (LSTM) networks

1. Recurrent Neural Networks (RNNs)
2. Long Short-Term Memory (LSTM) networks

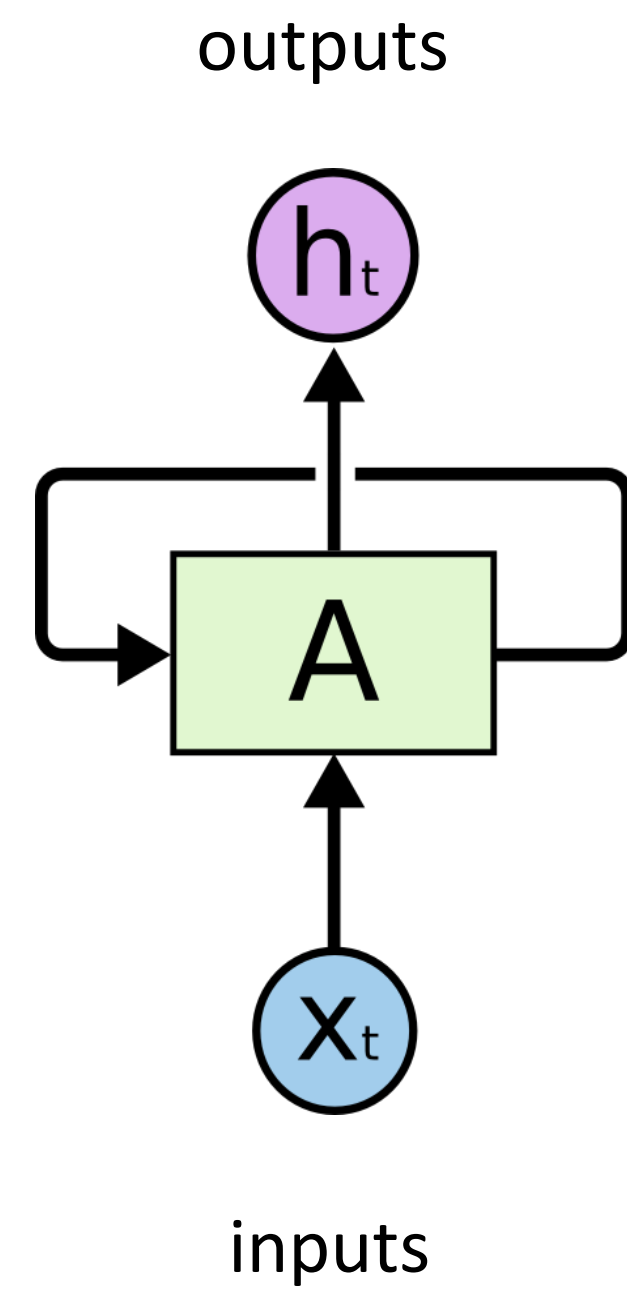
## Examples of sequential data:

- ▶ Text considered as a sequence of words or characters
- ▶ Continuous parameter which is a function of time (e.g. stock price)
- ▶ Sequence of images in a video-clip
- ▶ Sequence of labels in a genome sequence
- ▶ ... plus many others

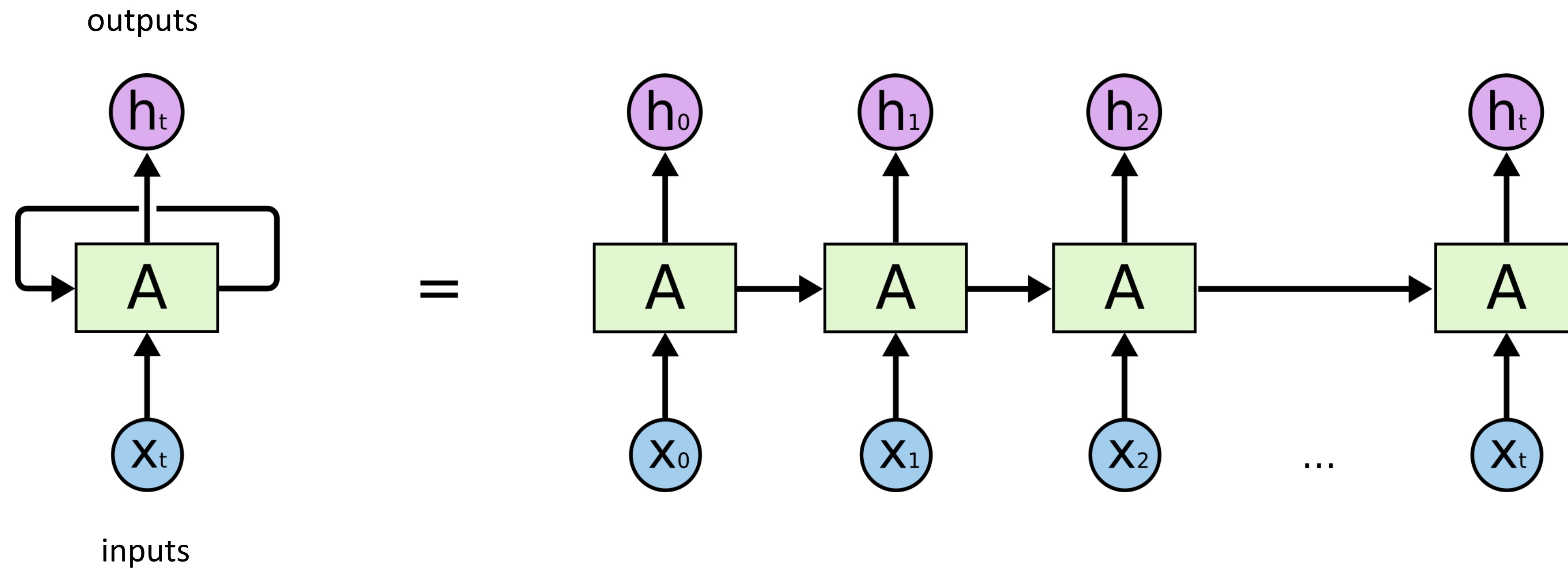
## Applications of RNNs to sequential data:

- ▶ Text generation “in the style of”
- ▶ Provide “sentiment analysis” on a piece of text
- ▶ Automatic “speech-to-text” as in MS Teams
- ▶ Automatic foreign language translation
- ▶ Prediction of stock price on the basis of historical data
- ▶ Label sequence of images in a video-clip

## Basic structure of RNNs:



## Basic structure of RNNs:



## Simple (unrealistic) example:

Example:

Based on a (usually long) text used for training, we wish to generate “similar” text on a character by character basis.

Given a training string: `Hello world!`

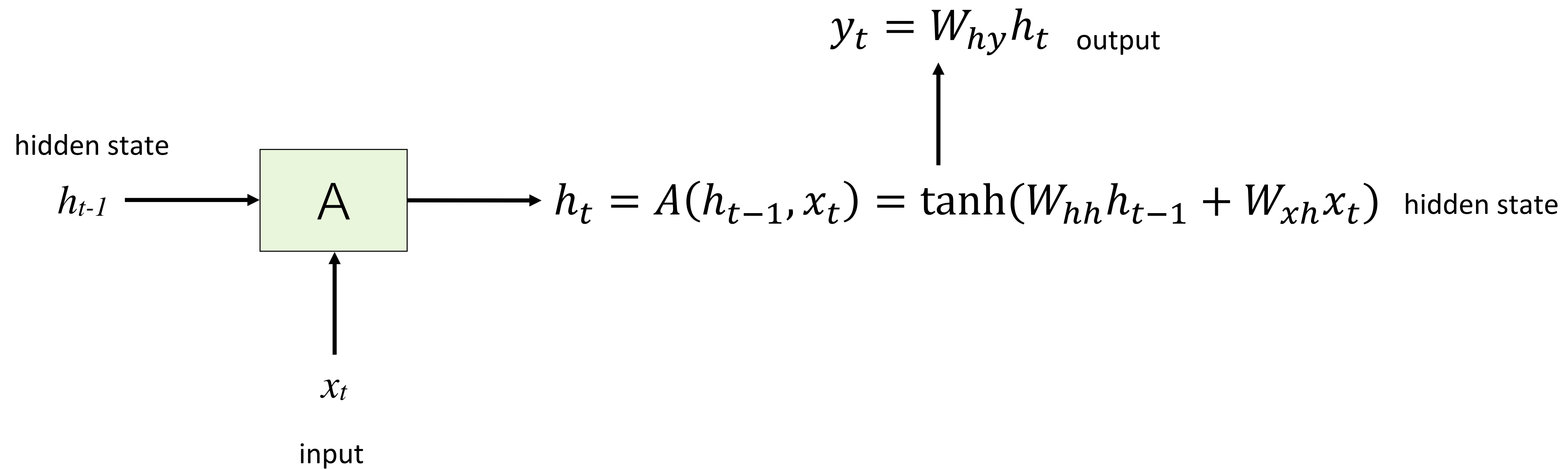
We identify the dictionary associated with the input (ignoring capitalisation) contains 9 characters: (d, e, h, l, o, r, w, , !)

And we hot-encode the dictionary:

$$h = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad ! = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad \dots$$

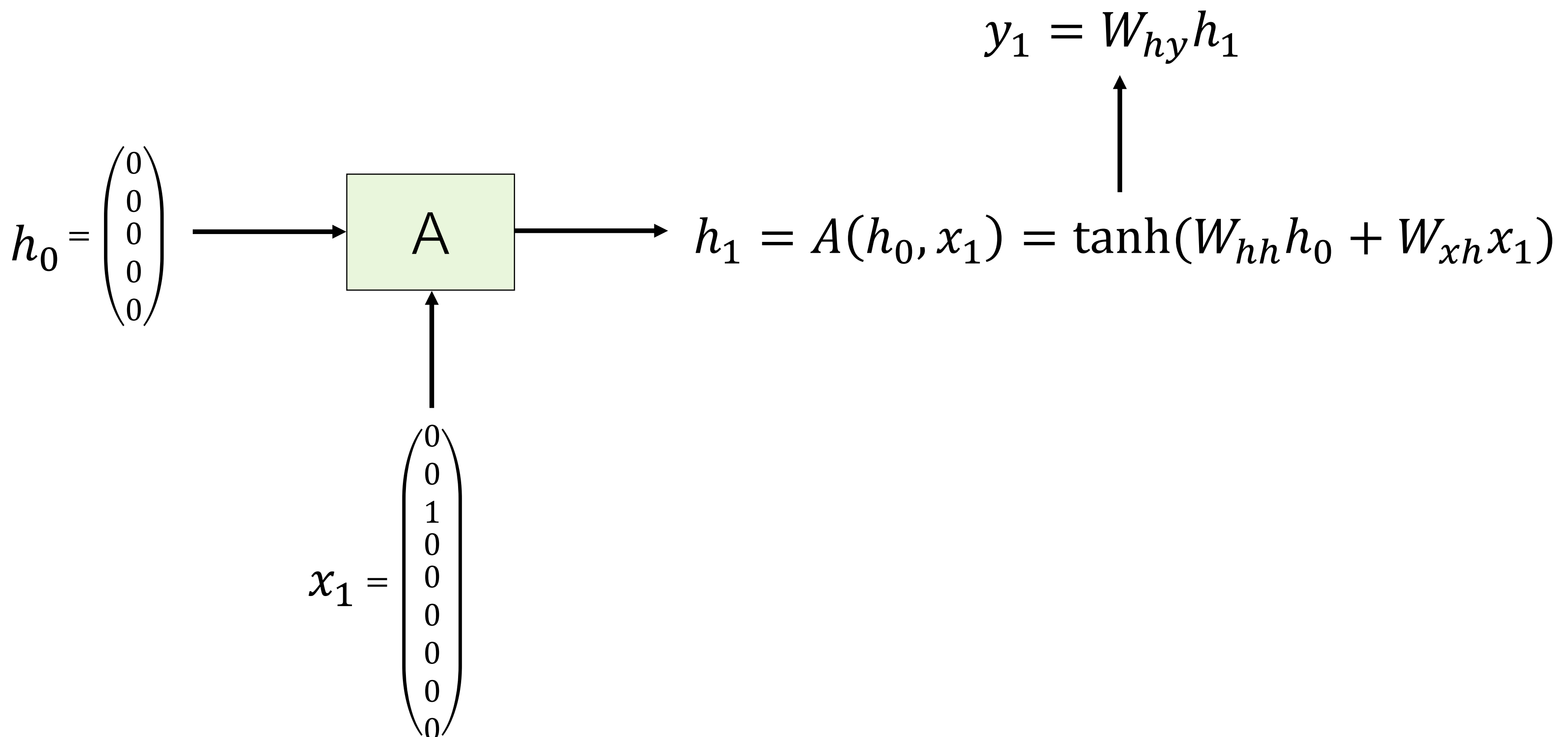


How does a basic RNN cell work?



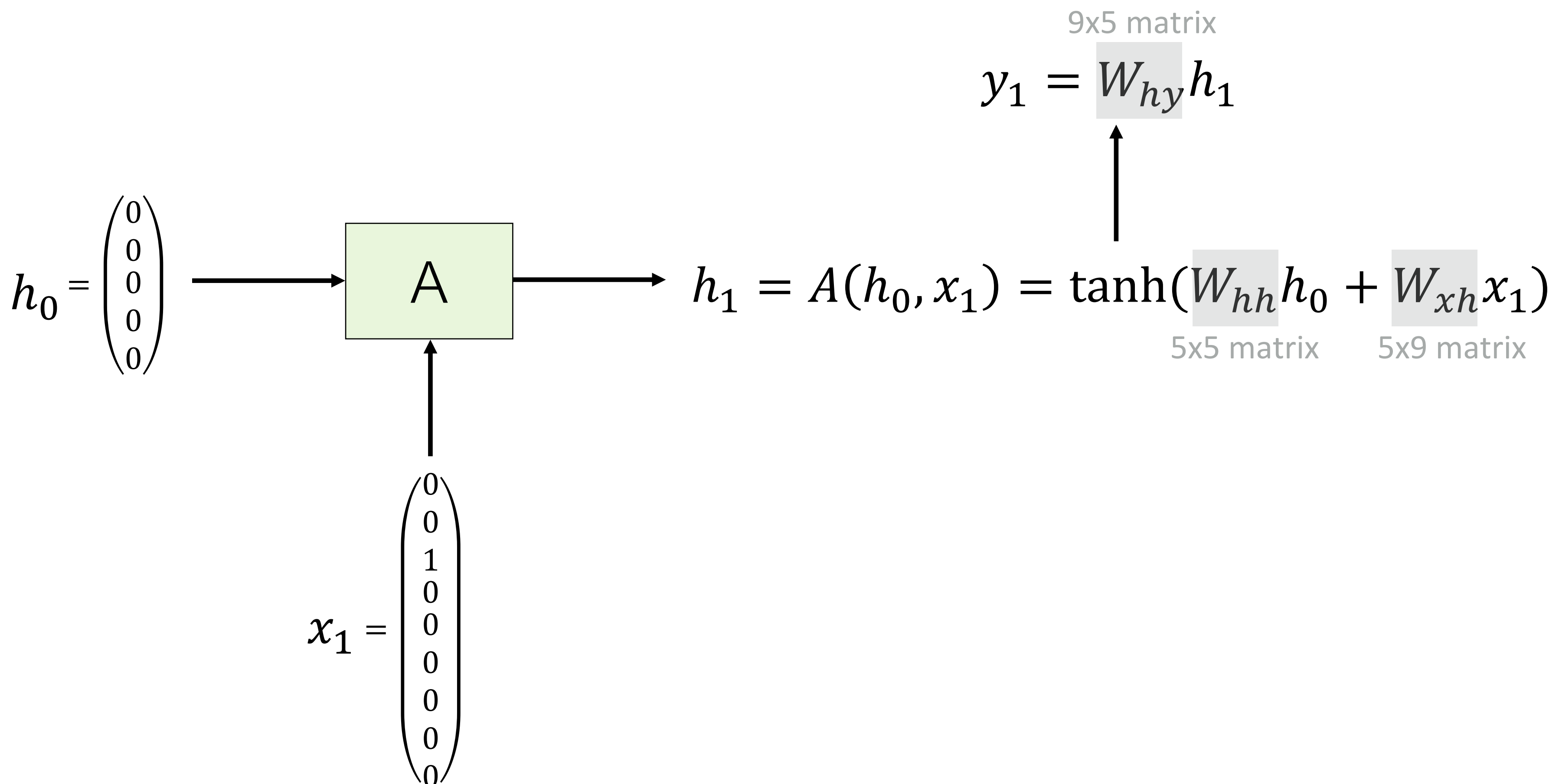
How does training work? Back to our simple example:

We define the size of our hidden state vector  $h$  (hyperparameter) to be 5, and start with our first input character ('h'):



How does training work? Back to our simple example:

We define the size of our hidden state tensor  $h$  (hyperparameter) to be 5, and start with our first input character ('h'):



How does training work? The loss function:

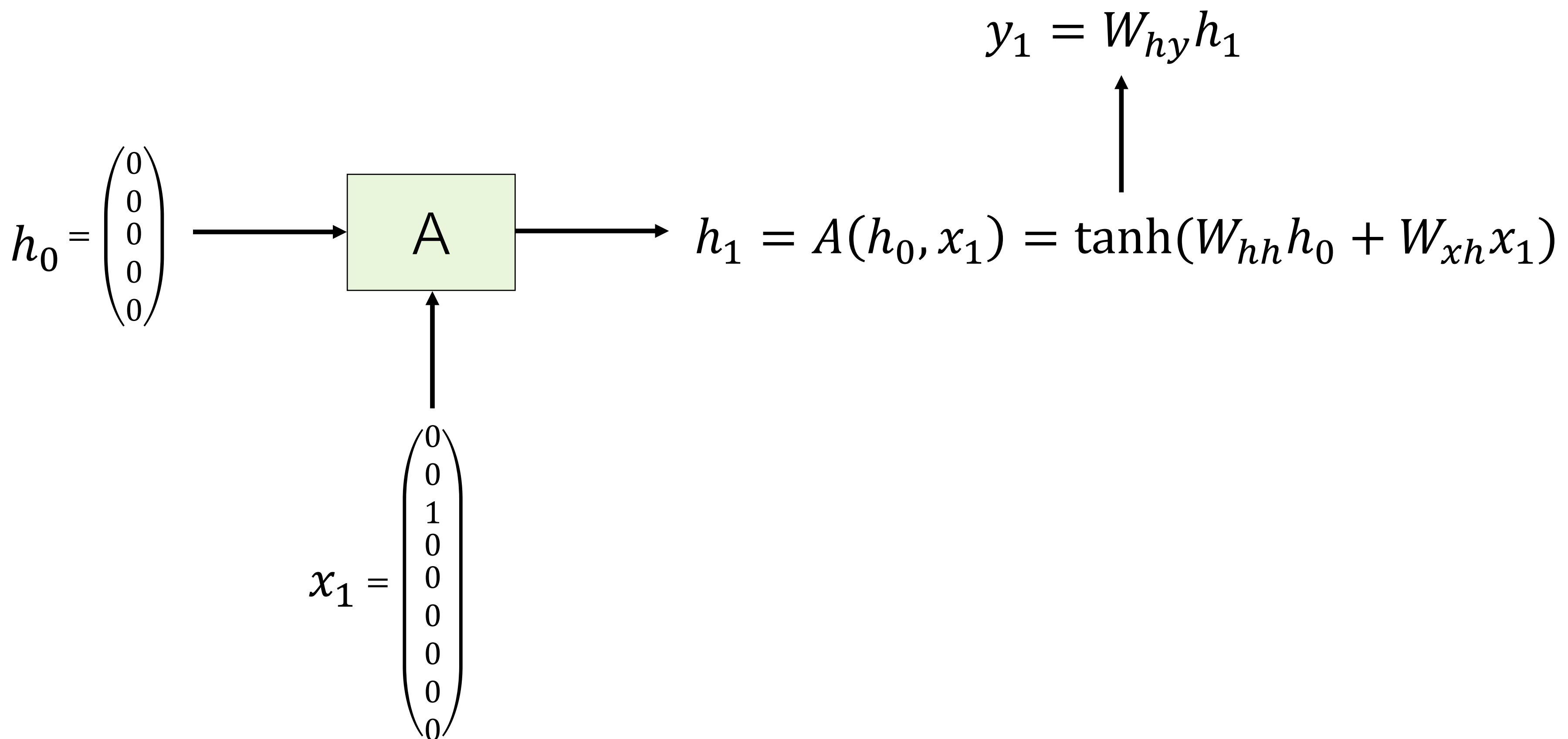
$loss =$

$$y_1 = W_{hy}h_1 \longrightarrow \textit{Cross-entropy} ( \textit{Softmax} (y_1) , x_2 )$$

Loss function corresponding to  
the first predicted letter

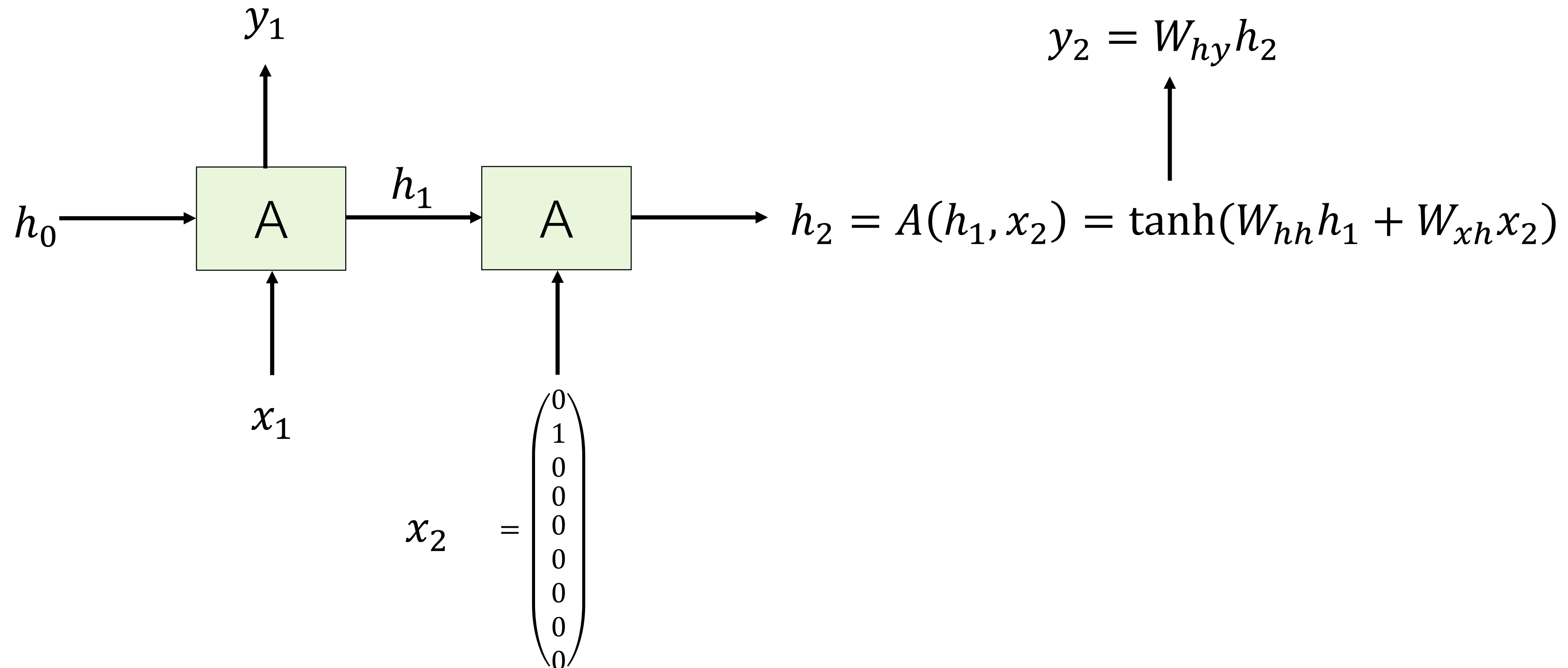
remember: hot-  
encoded

How does training work for the next characters? Back to our previous example:



How does training work for the next characters? Back to our previous example:

We concatenate the **same** cell after the output of our previous one:



How does training work? The loss function:

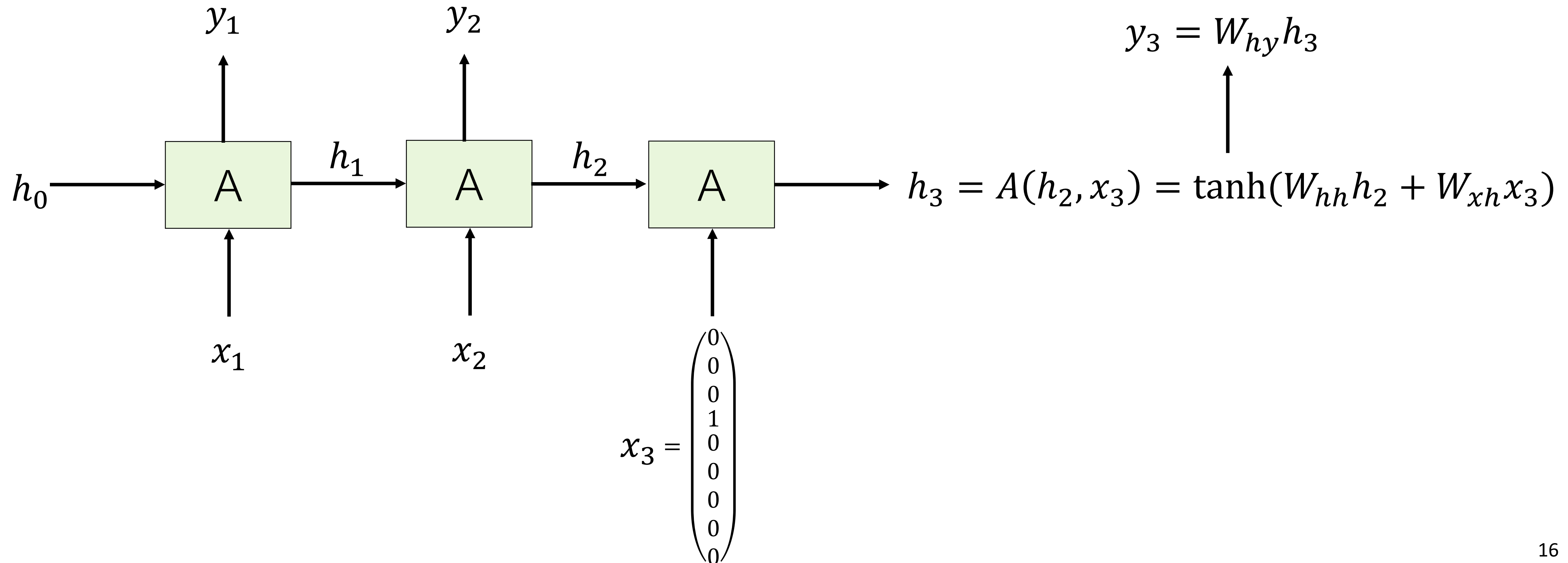
$$\begin{aligned} & \text{loss} = \\ & y_1 = W_{hy}h_1 \longrightarrow \text{Cross-entropy} ( \text{Softmax} (y_1) , x_2 ) \\ & \quad + \\ & \quad \text{Cross-entropy} ( \text{Softmax} (y_2) , x_3 ) \end{aligned}$$

Loss function corresponding to  
the second predicted letter

remember: hot-  
encoded

How does training work for the next characters? Back to our previous example:

We concatenate the **same** cell after the output of our previous one:





How does training work? The loss function:

$$\begin{aligned} & \text{loss} = \\ & y_1 = W_{hy}h_1 \longrightarrow \text{Cross-entropy} ( \text{Softmax} (y_1) , x_2 ) \\ & \quad + \\ & \quad \text{Cross-entropy} ( \text{Softmax} (y_2) , x_3 ) \\ & \quad + \\ & \quad \text{Cross-entropy} ( \text{Softmax} (y_3) , x_4 ) \\ & \quad + \dots \end{aligned}$$

Loss function corresponding to the third predicted letter

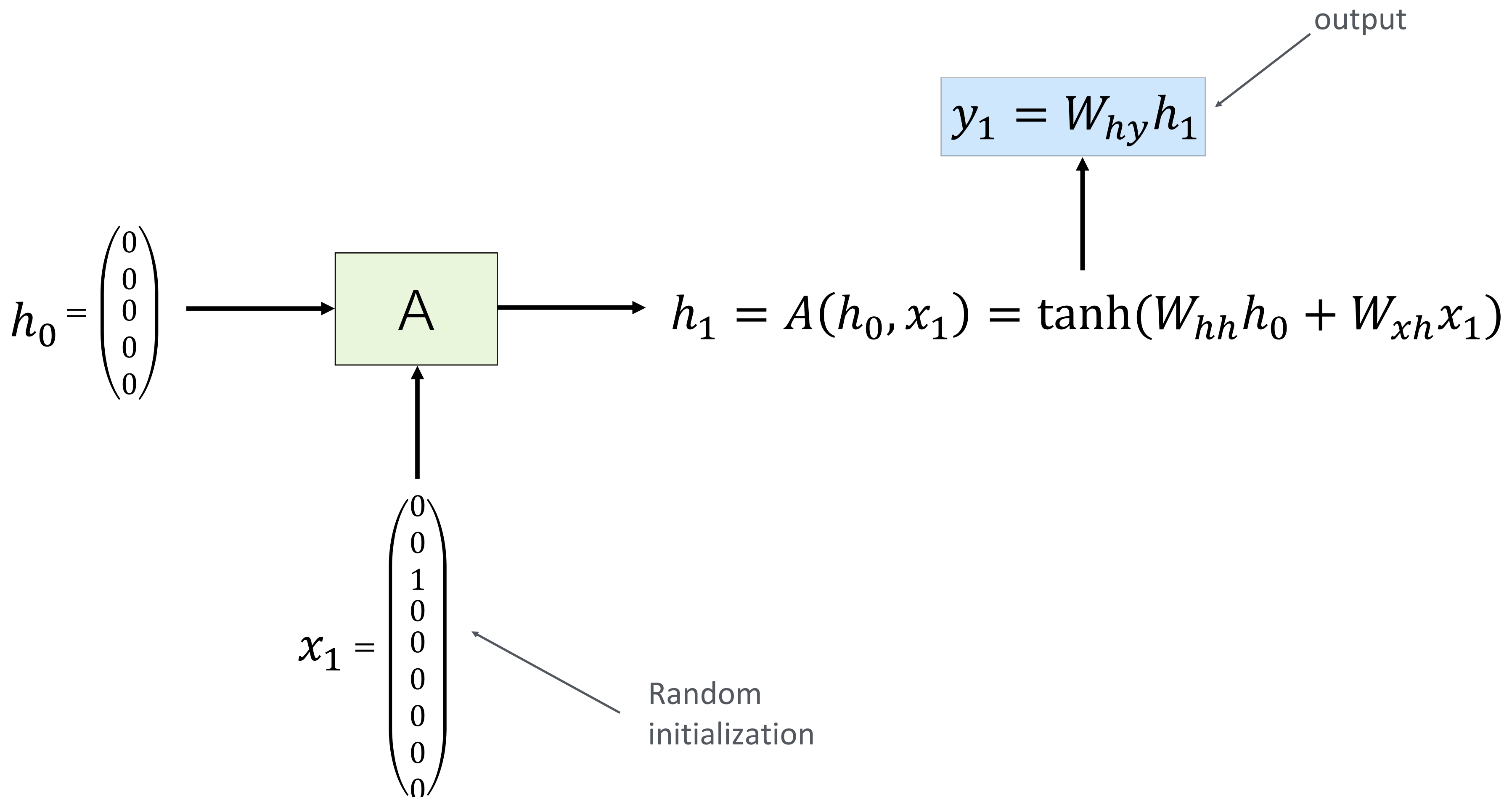
remember: hot-encoded

We repeat the process for all the characters in our input and add their loss contributions

And then use gradient descent to train our weights in the matrices

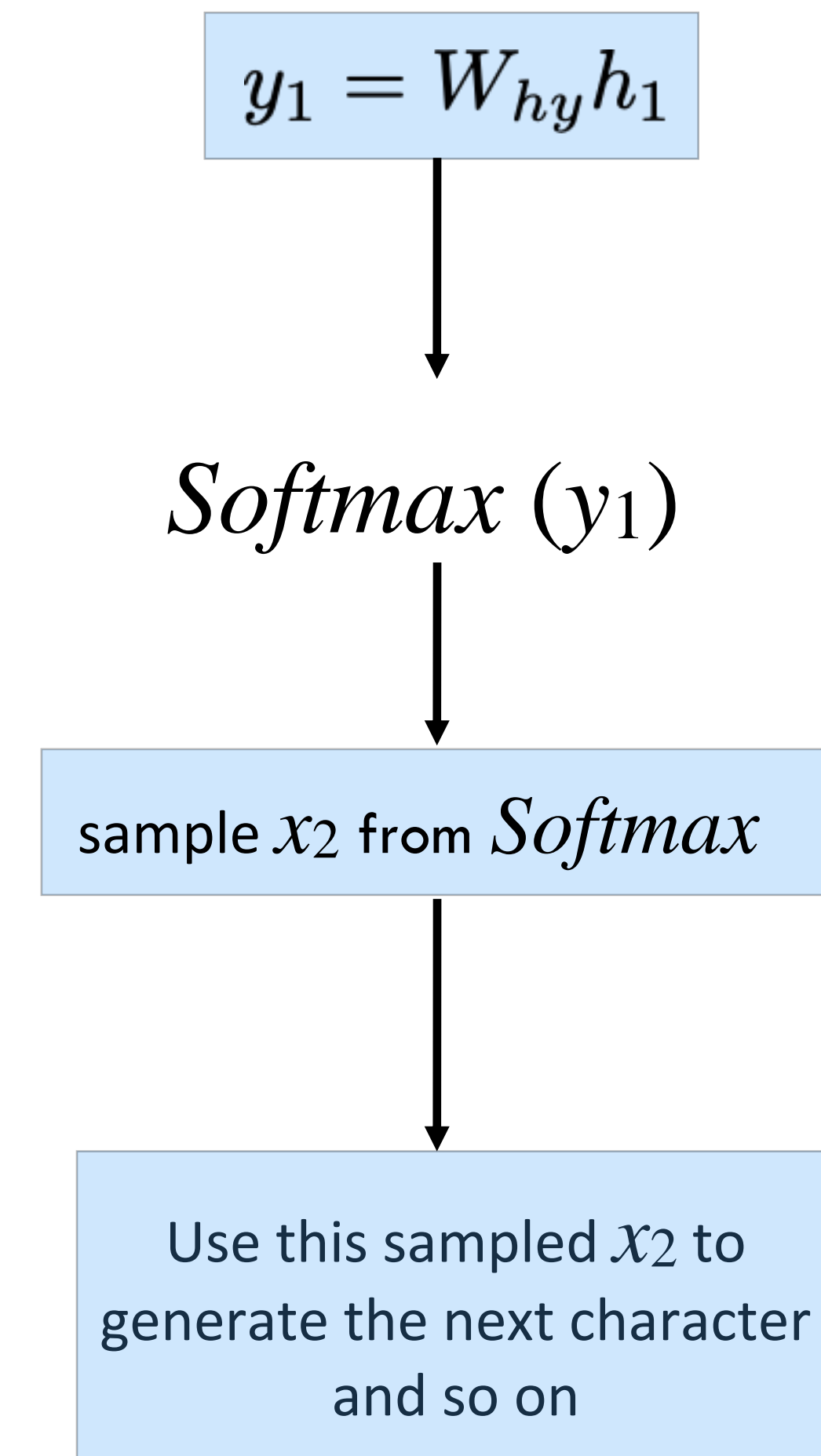
## Text generation using a trained network

Now we are going to use our trained RNN to generate some text:



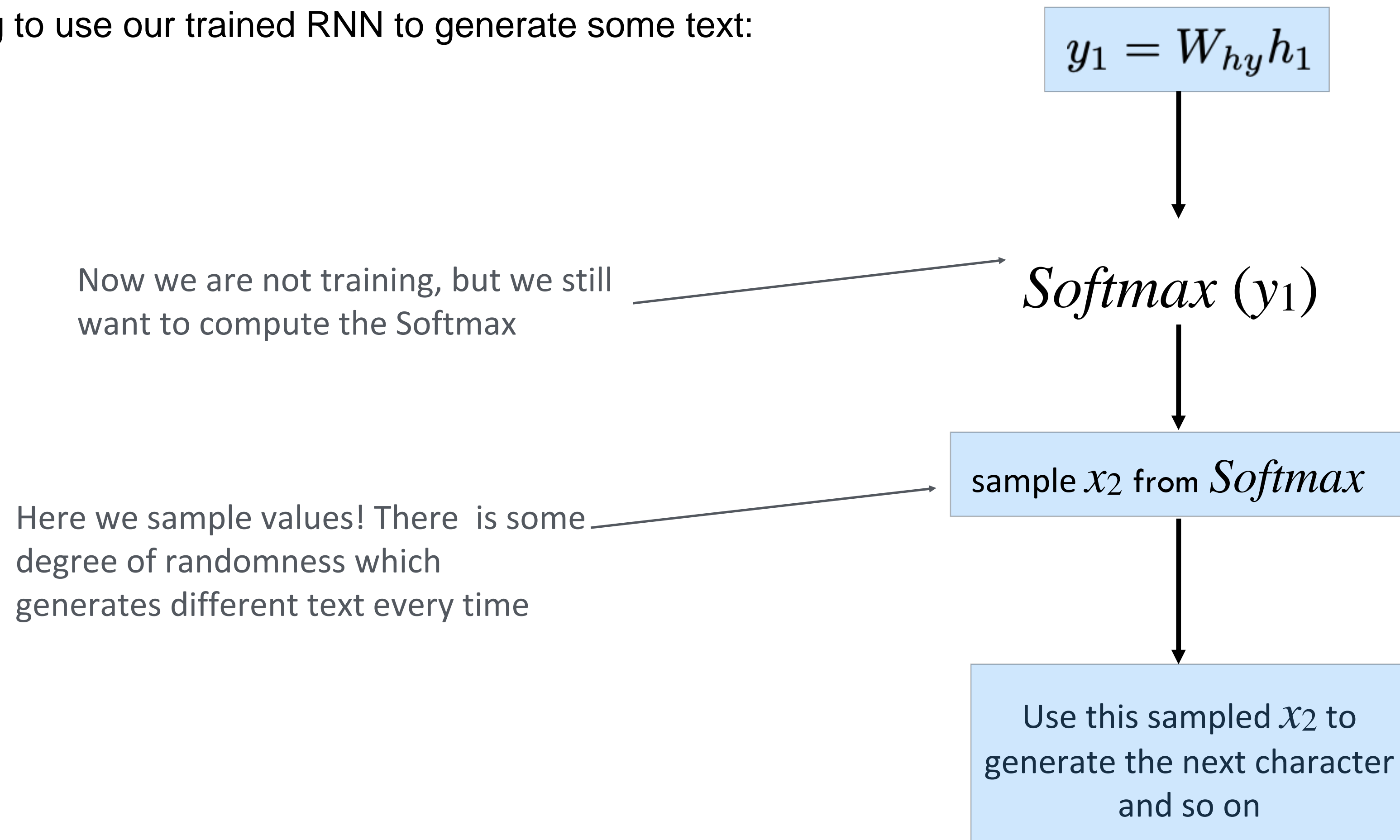
## Text generation using a trained network

Now we are going to use our trained RNN to generate some text:



## Text generation using a trained network

Now we are going to use our trained RNN to generate some text:



## Recap of training an RNN

1. Take first character  $x_1$ , combine with current hidden state  $h_0$  (its dimension is a hyperparameter) to derive  $h_1$ , calculate associated output vector  $y_1$  (of size equal to vocabulary), transform  $y_1$  into Softmax vector, calculate cross-entropy with second character  $x_2$  of the sequence.
2. Take the second character as input to repeat the step 1. and calculate loss function again using the value of the third character of the sequence, and so on until we reach the end of the sequence.
3. Add up all the above loss functions.
4. Do backpropagation to calculate gradients according to each trainable parameter.
5. Modify parameters by gradient descent.
6. Move to next sequence of  $p$  characters, until end of training set is reached. Size of  $p$  is a hyperparameter too!

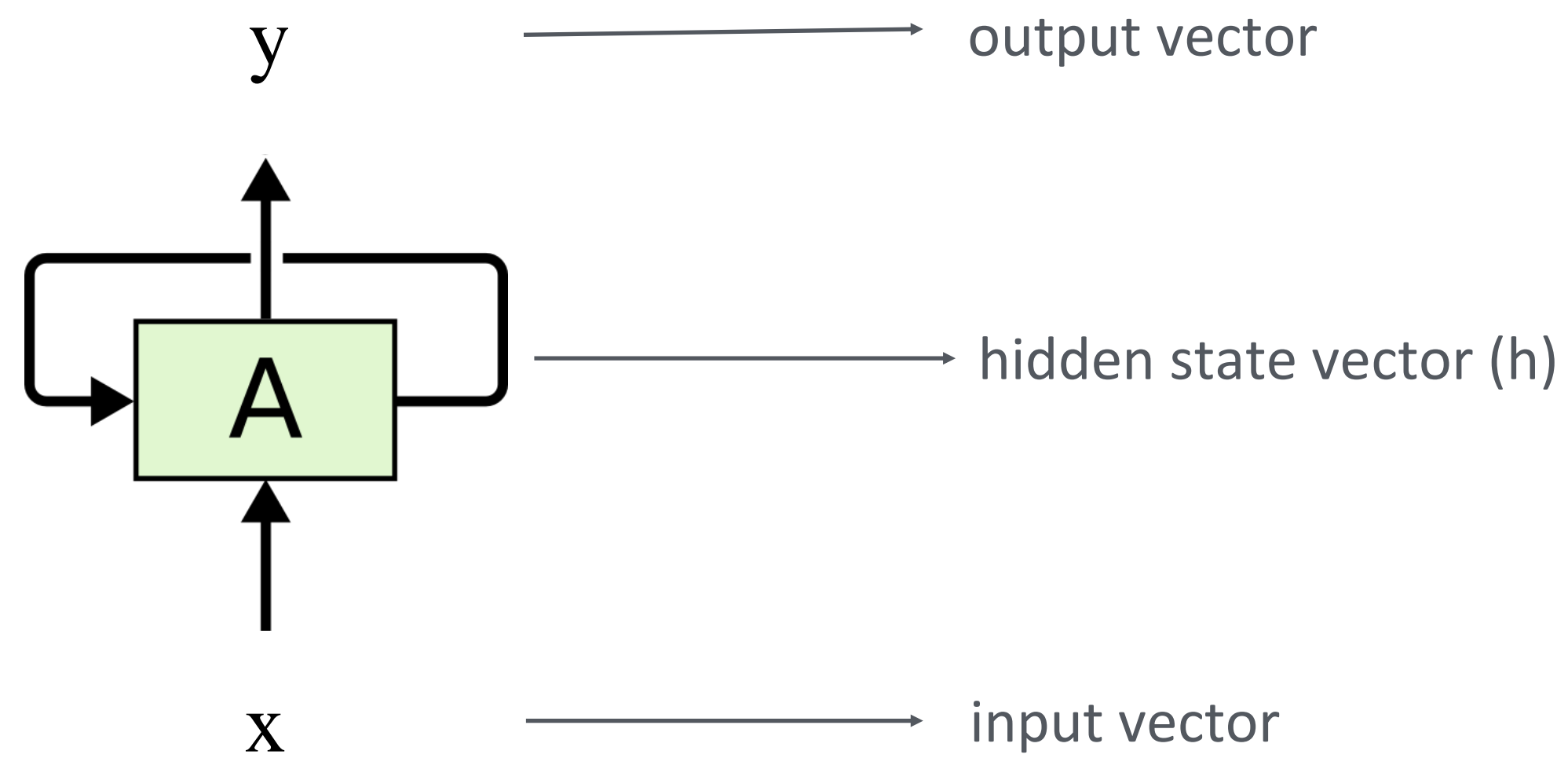
## Recap of generating text with an RNN

Generate a new sequence of  $n$  characters:

1. Randomly sample first character  $x_1$ .
2. Combine with initial (zero) hidden state vector value  $h_0$  to derive new vector  $h_1$ .
3. Calculate associated output vector  $y_1$  of size equal to vocabulary size.
4. Transform  $y_1$  into Softmax vector.
5. Sample new character  $x_2$  based on Softmax probability.
6. Continue until the required number of characters have been generated.

Let's look at some code!

## Synthetic notation for RNNs:





## Simple implementation in python:

<https://gist.github.com/karpathy/d4dee566867f8291f086>

And after a few hours of training, we start to get some decent writing:

```
VIOLA:  
Why, Salisbury must find his flesh and thought  
That which I am not aps, not a man and in fire,  
To show the reining of the raven and the wars  
To grace my hand reproach within, and not a fair are hand,  
That Caesar and my goodly father's world;  
When I was heaven of presence and our fleets,  
We spare with hours, but cut thy council I am great,  
Murdered and by thy master's ready there  
My power to give thee but so much as hell:  
Some service in the noble bondman here,  
Would show him to her wine.
```

```
KING LEAR:  
O, if you were a feeble sight, the courtesy of your law,  
Your sight and several breath, will wear the gods  
With his heads, and my hands are wonder'd at the deeds,  
So drop upon your lordship's head, and your opinion  
Shall be against your honour.
```



Analysis of the role of  $h$ 

Cell sensitive to position in line:

The sole importance of the crossing of the Berezina lies in the fact that it plainly and indubitably proved the fallacy of all the plans for cutting off the enemy's retreat and the soundness of the only possible line of action--the one Kutuzov and the general mass of the army demanded--namely, simply to follow the enemy up. The French crowd fled at a continually increasing speed and all its energy was directed to reaching its goal. It fled like a wounded animal and it was impossible to block its path. This was shown not so much by the arrangements it made for crossing as by what took place at the bridges. When the bridges broke down, unarmed soldiers, people from Moscow and women with children who were with the French transport, all--carried on by vis inertiae--pressed forward into boats and into the ice-covered water and did not, surrender.

Cell that turns on inside quotes:

"You mean to imply that I have nothing to eat out of.... On the contrary, I can supply you with everything even if you want to give dinner parties," warmly replied Chichagov, who tried by every word he spoke to prove his own rectitude and therefore imagined Kutuzov to be animated by the same desire.

Kutuzov, shrugging his shoulders, replied with his subtle penetrating smile: "I meant merely to say what I said."

Cell that robustly activates inside if statements:

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
                           siginfo_t *info)
{
    int sig = next_signal(pending, mask);
    if (sig) {
        if (current->notifier) {
            if (sigismember(current->notifier_mask, sig)) {
                if (!(current->notifier)(current->notifier_data)) {
                    clear_thread_flag(TIF_SIGPENDING);
                    return 0;
                }
            }
        }
        collect_signal(sig, pending, info);
    }
    return sig;
}
```



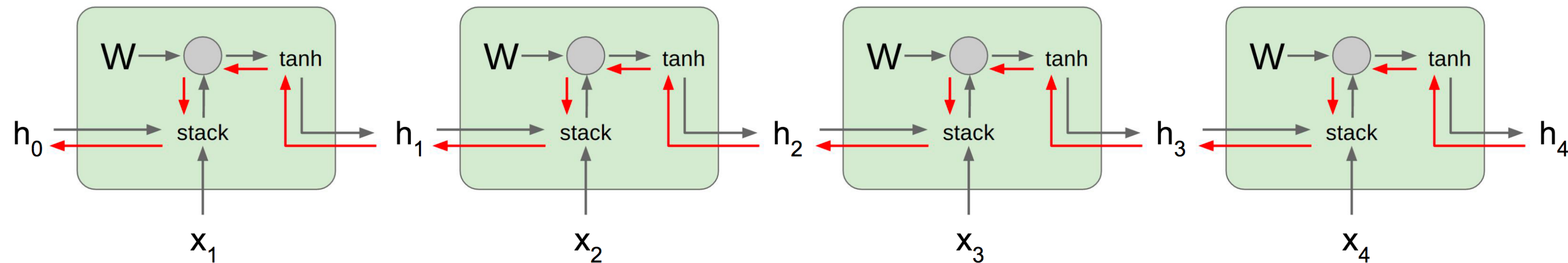
## Analysis of the role of $h$

A large portion of cells are not easily interpretable. Here is a typical example:

```
/* Unpack a filter field's string representation from user-space
 * buffer. */
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)
{
    char *str;
    if (!*bufp || (len == 0) || (len > *remain))
        return ERR_PTR(-EINVAL);
    /* Of the currently implemented string fields, PATH_MAX
     * defines the longest valid length.
     */
}
```

In their study, Karpathy et al found that only ~5% of the  $h$  positions were interpretable

## Backpropagation through RNNs



Computing gradients involves many factors of  $W$  and repeated  $\tanh$  activations and that creates problems:

- ▶ Exploding gradients: can be solved by gradient clipping
- ▶ Vanishing gradients: requires changes in the RNN architecture

the derivative of  $\tanh$  is in the range  $(0, 1)$  and we apply it multiple times using the chain rule in backprop

## Issues with RNNs

- Exploding and vanishing gradients
- Lack of long term memory

## Issues with RNNs

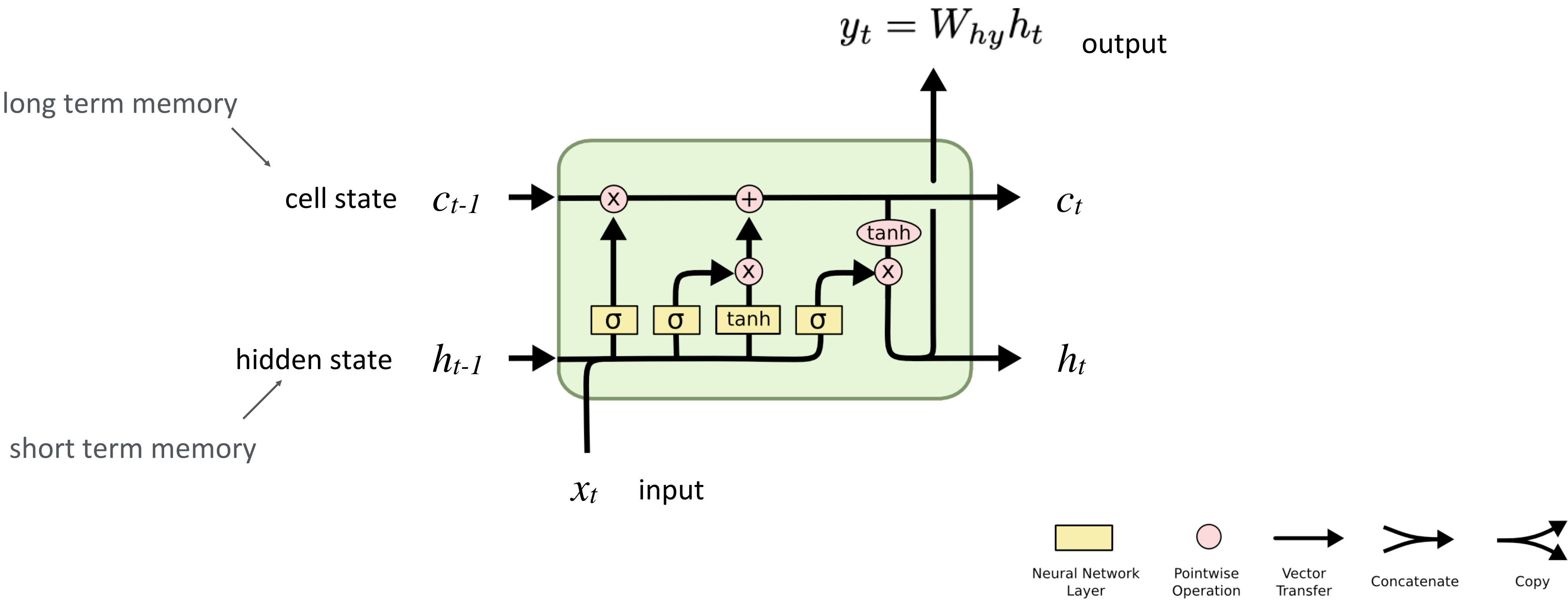
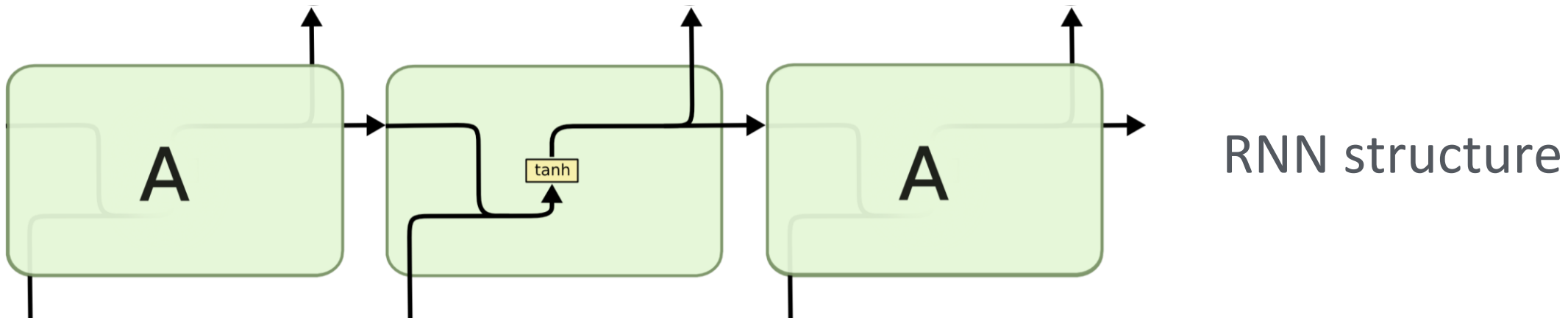
- Exploding and vanishing gradients
- Lack of long term memory



LSTMs to the rescue!

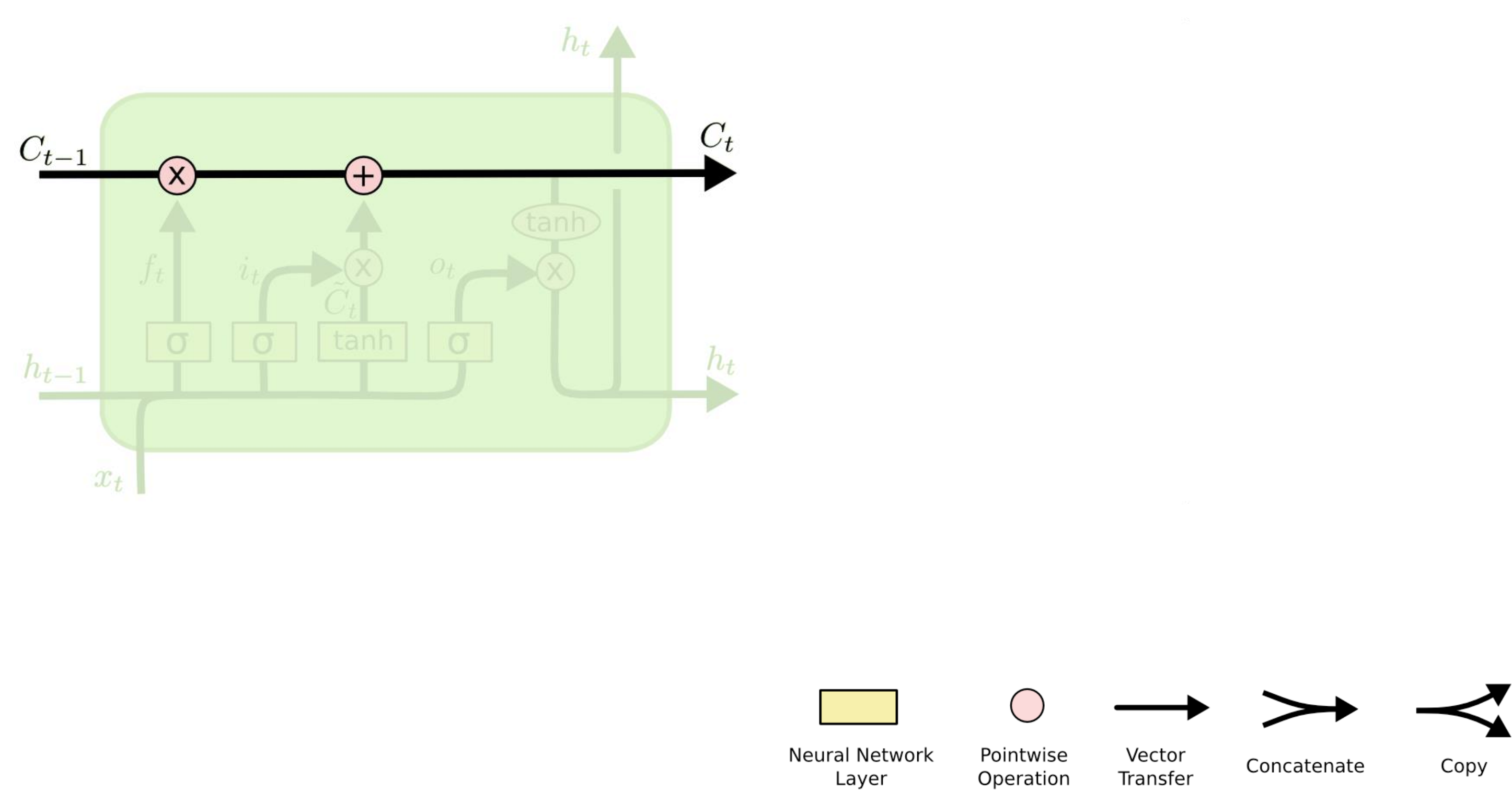
1. Recurrent Neural Networks (RNNs)
2. Long Short-Term Memory (LSTM) networks

Basic structure of an LSTM:

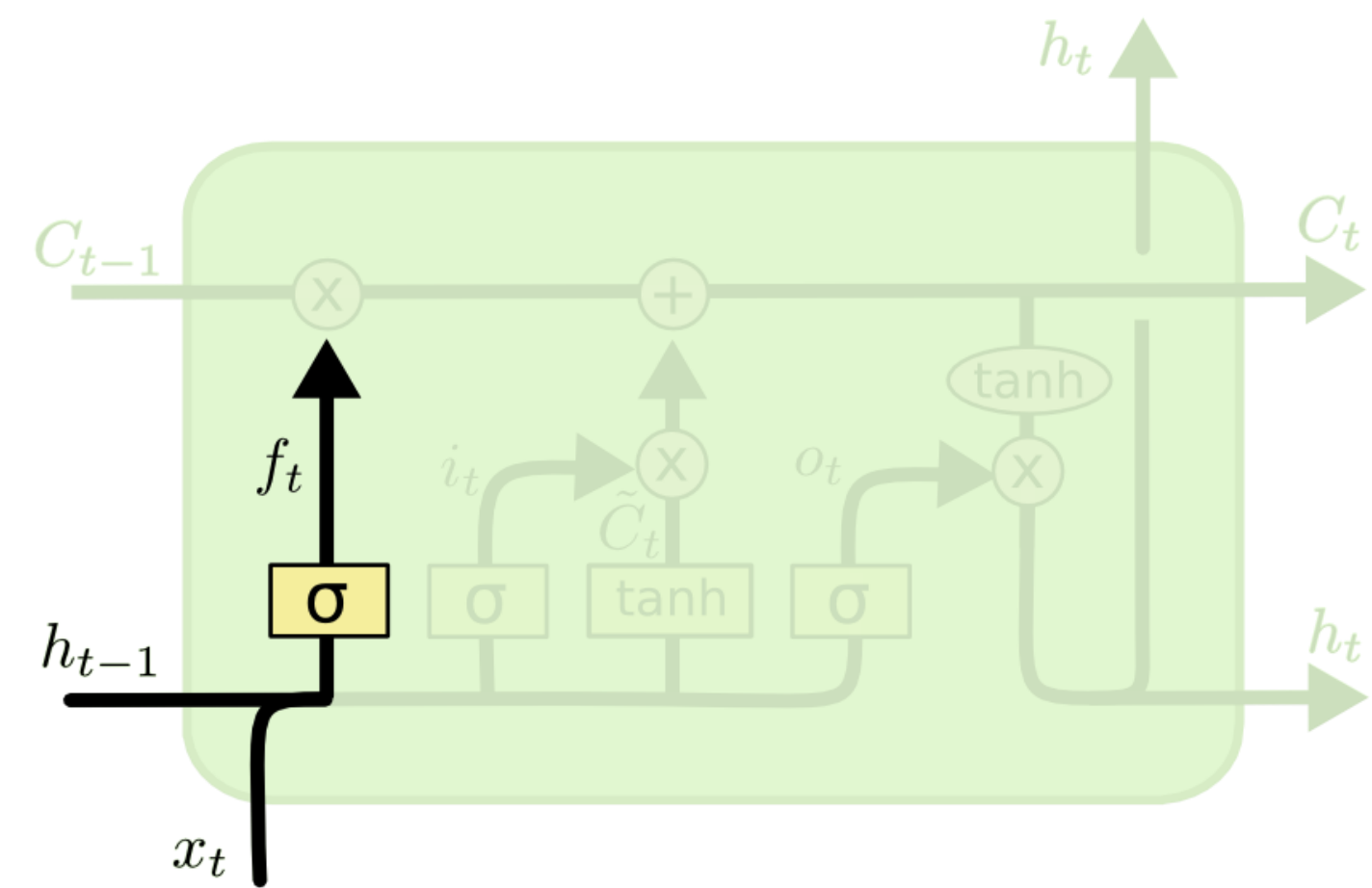




The core idea behind LSTMs: the **cell state** directly connects the whole chain of cells.

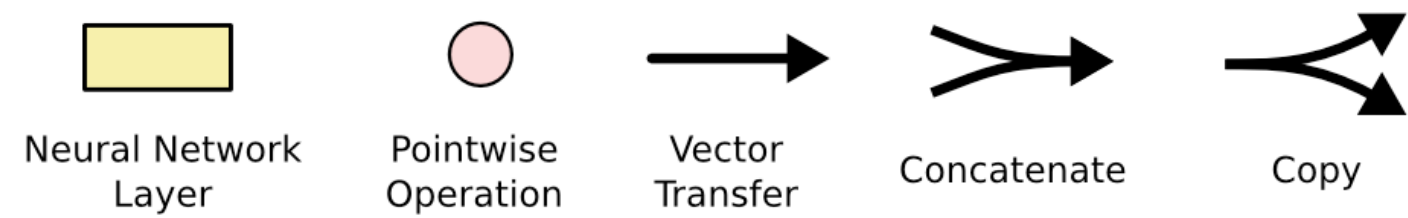


The **forget gate**: is there any information in the cell state that we no longer need?

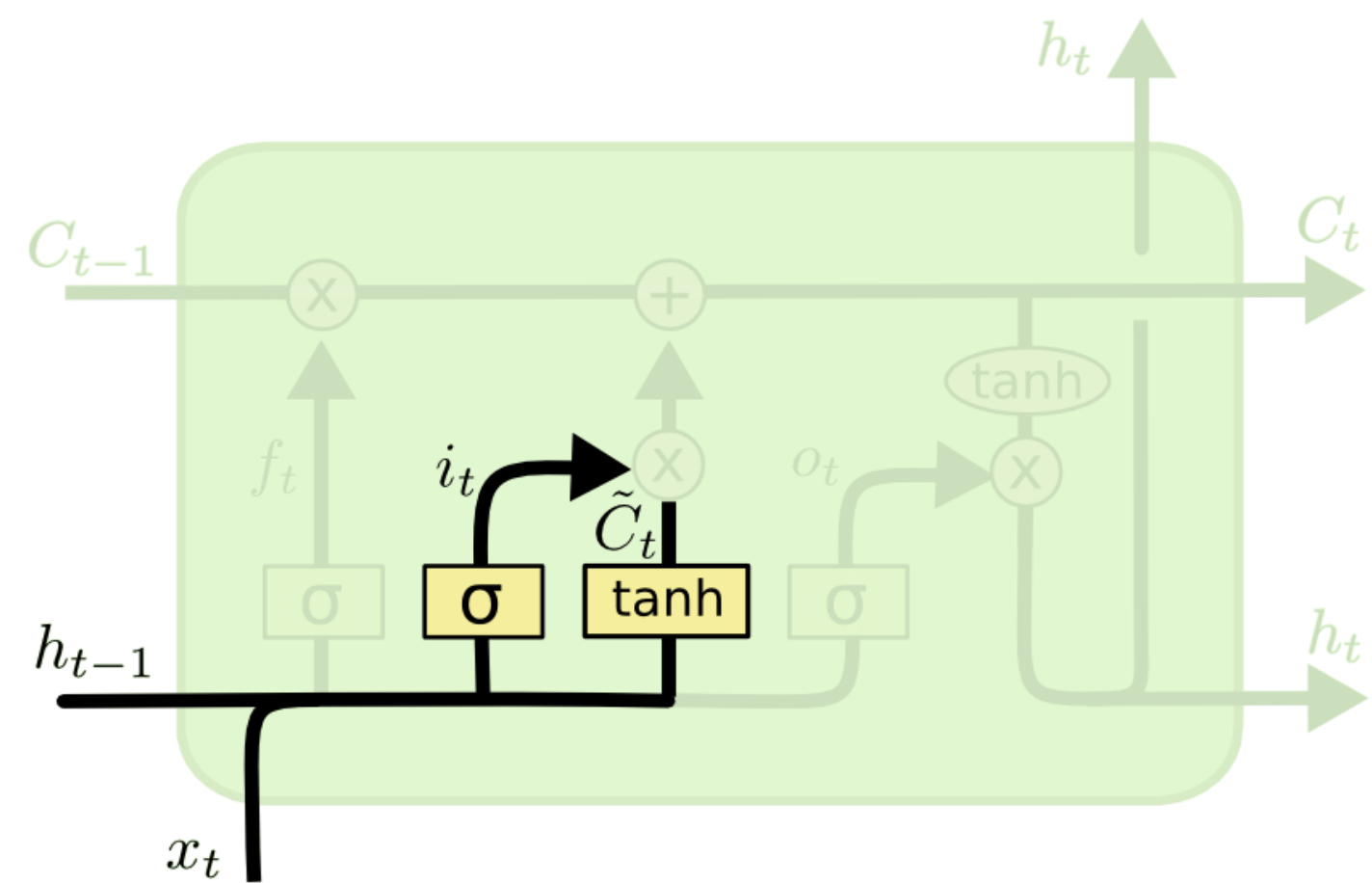


sigmoid normalises between 0 and 1

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

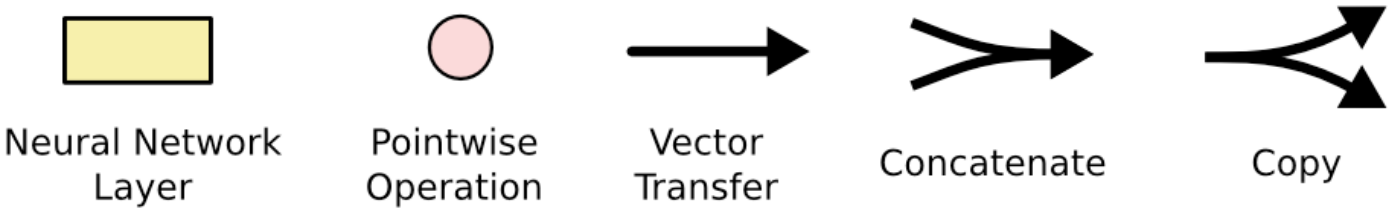


The **input gate** and the **candidate update**: what part of the input should we use to update the cell state?

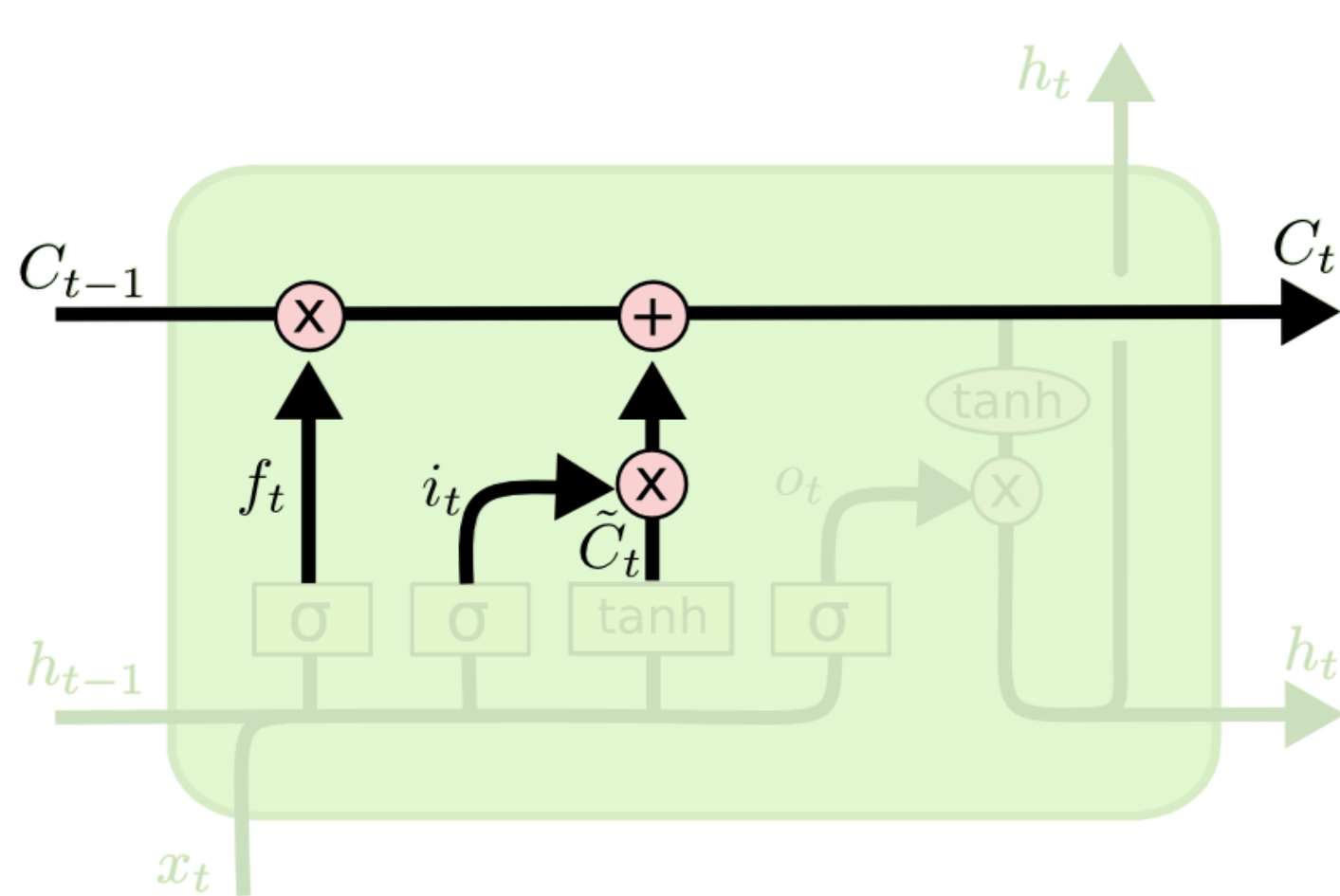


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

tanh normalises between -1 and 1,  
like cell state



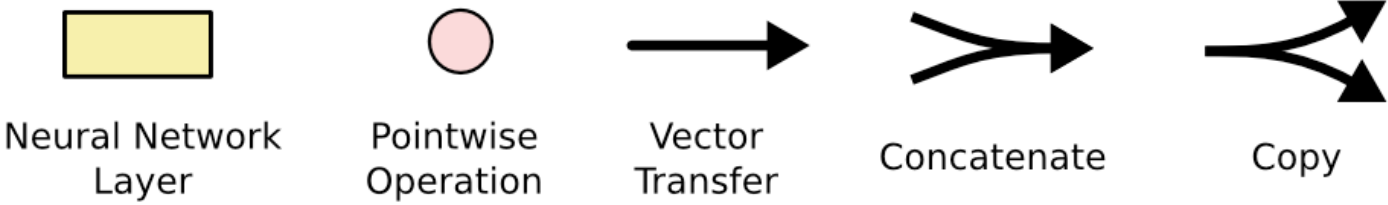
Updating the cell state:



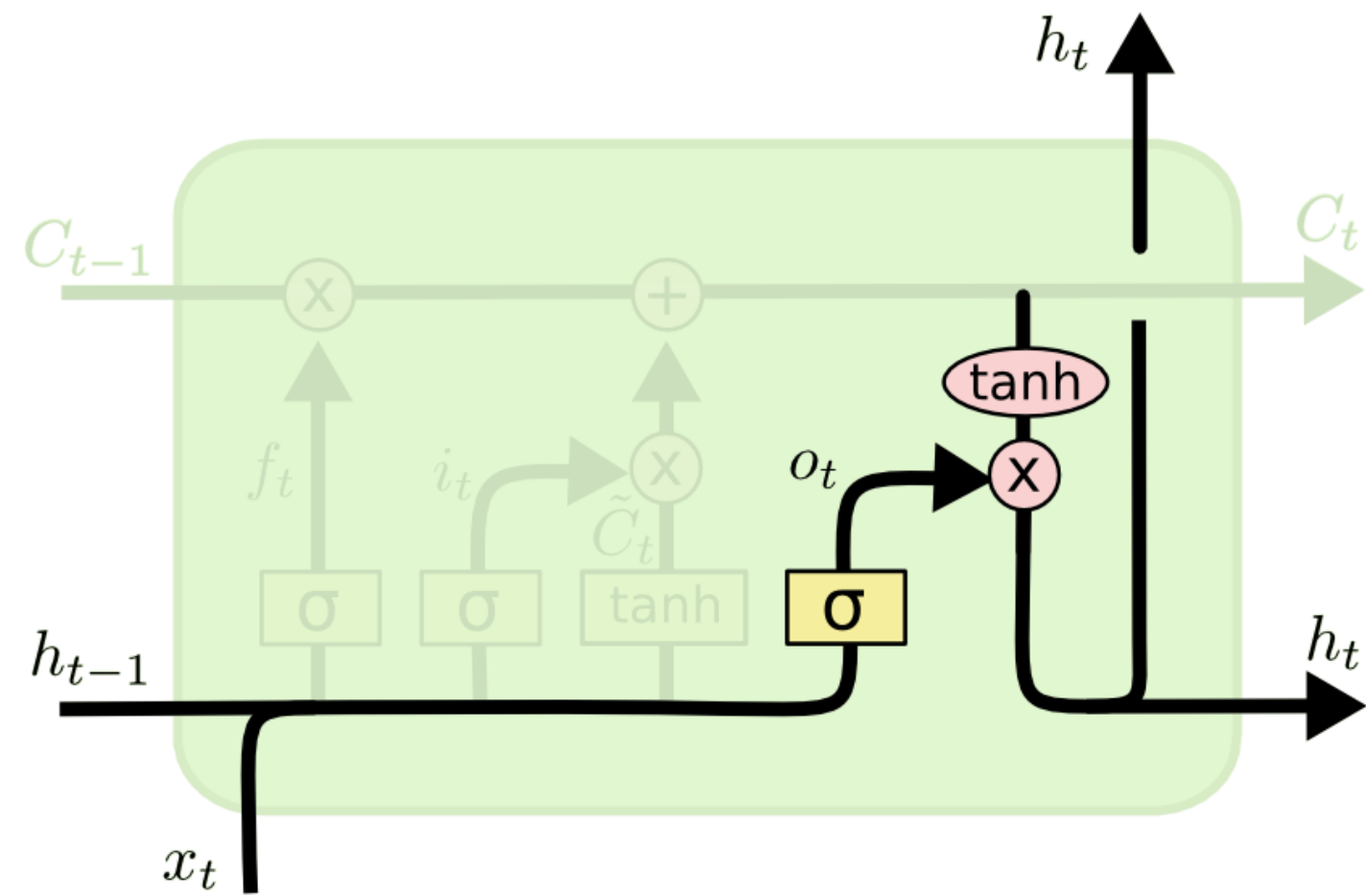
add the gated candidate values

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

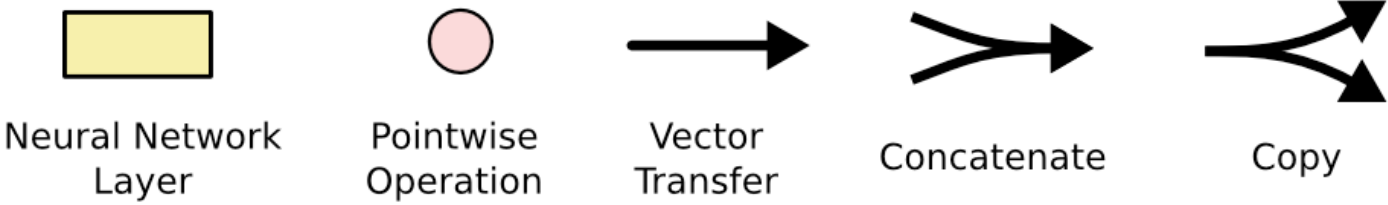
forget part of the previous cell state



The **output gate**: what part of the cell state should influence the next output?



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



## Basic structure of an LSTM:

vanilla RNN:

$$h_t = \tanh \left( W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

LSTM:

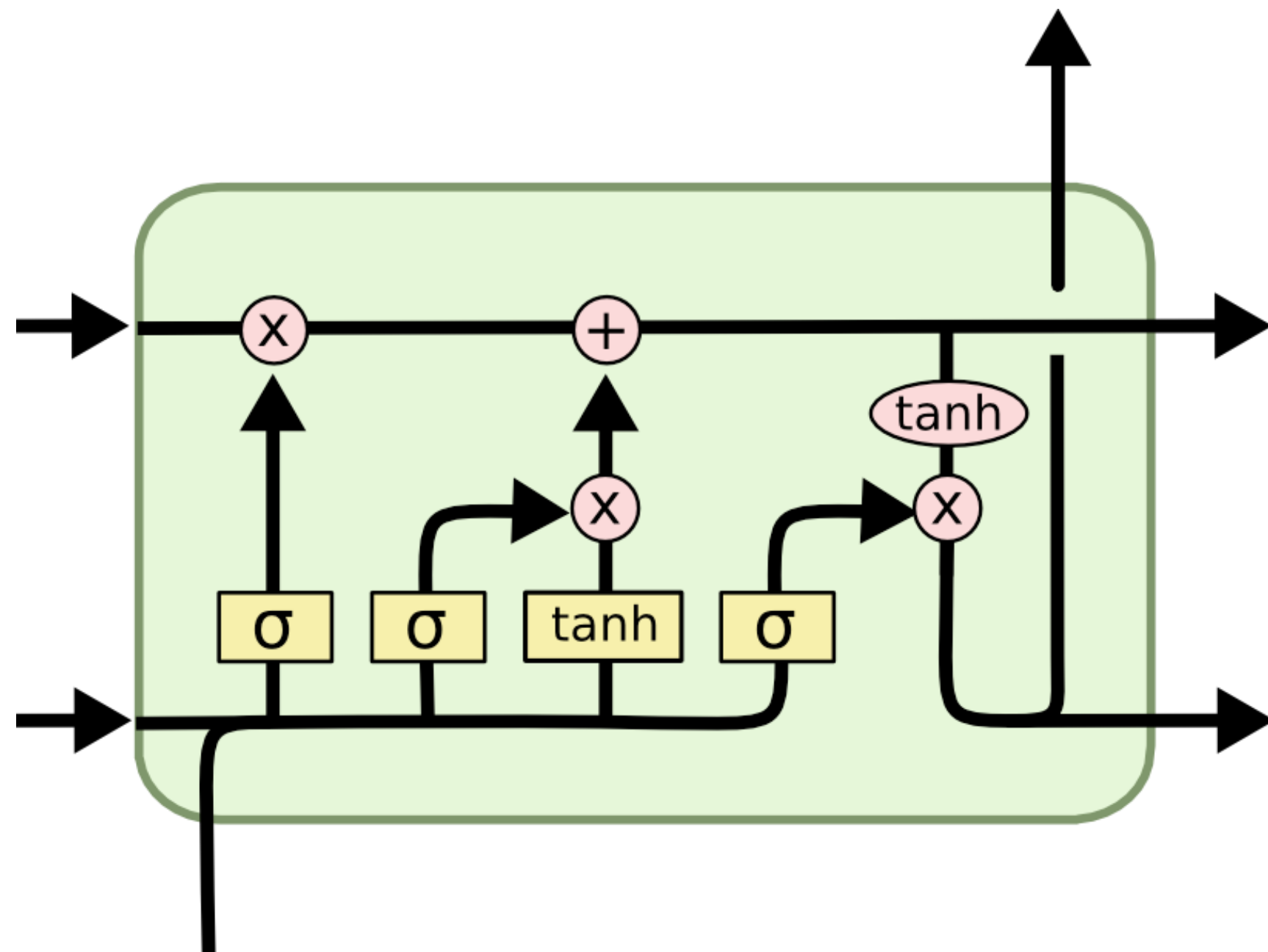
$$\begin{matrix} \text{input gate} & \nearrow & & \\ \text{forget gate} & \nearrow & & \\ \text{output gate} & \nearrow & & \\ \text{candidate update} & \nearrow & & \end{matrix} \begin{pmatrix} i \\ f \\ o \\ \tilde{c} \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot \tilde{c}$$

$$h_t = o \odot \tanh(c_t)$$

↑  
Hadamard product (element-wise multiplication)

Basic structure of an LSTM:

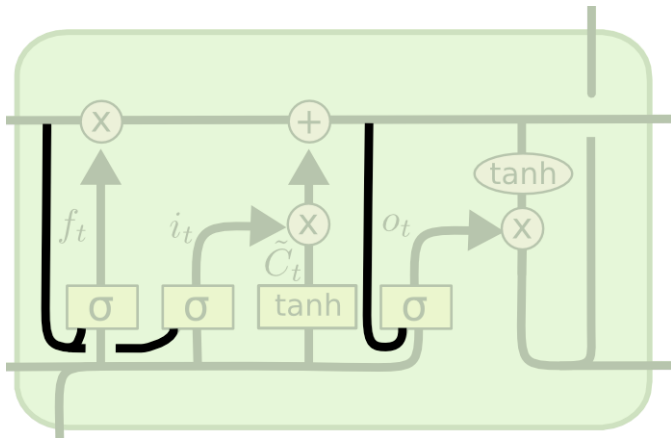


$$\begin{pmatrix} i \\ f \\ o \\ \tilde{c} \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \left( \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

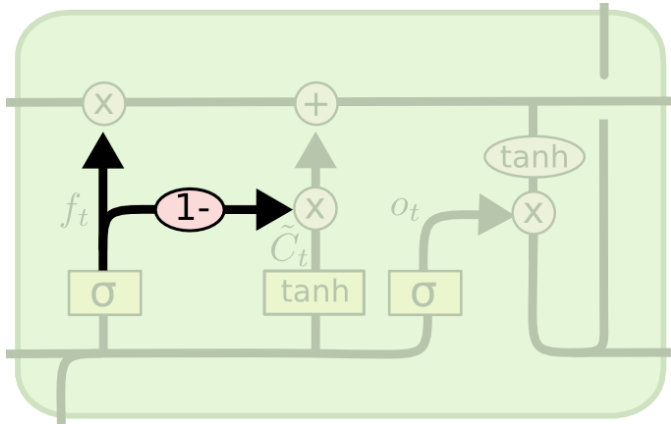
$$c_t = f \odot c_{t-1} + i \odot \tilde{c}$$

$$h_t = o \odot \tanh(c_t)$$

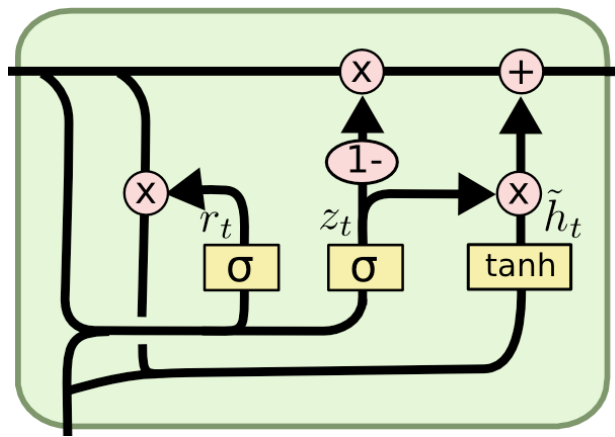
Variations on LSTMs:



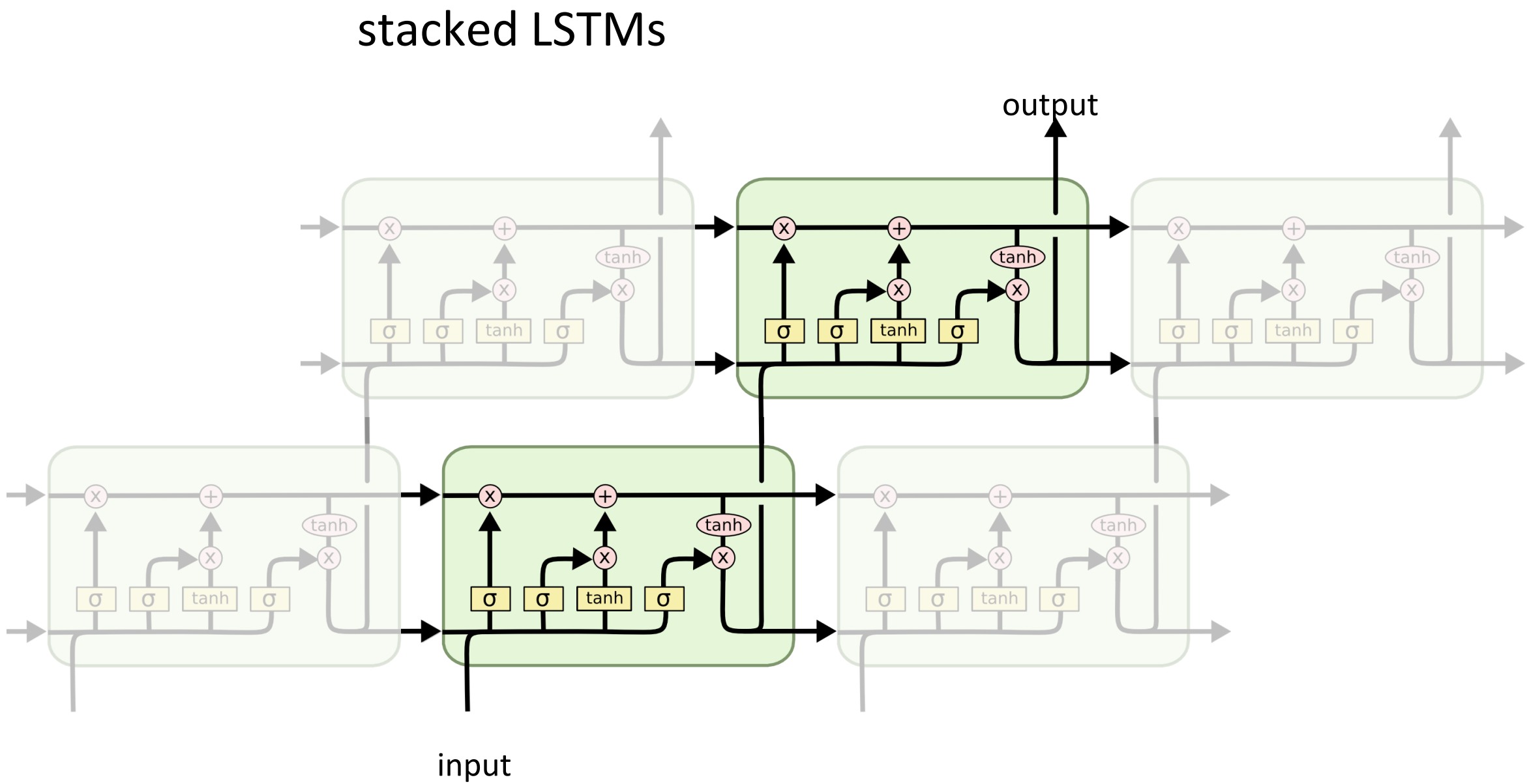
peep-hole connections



coupled forget and input gates



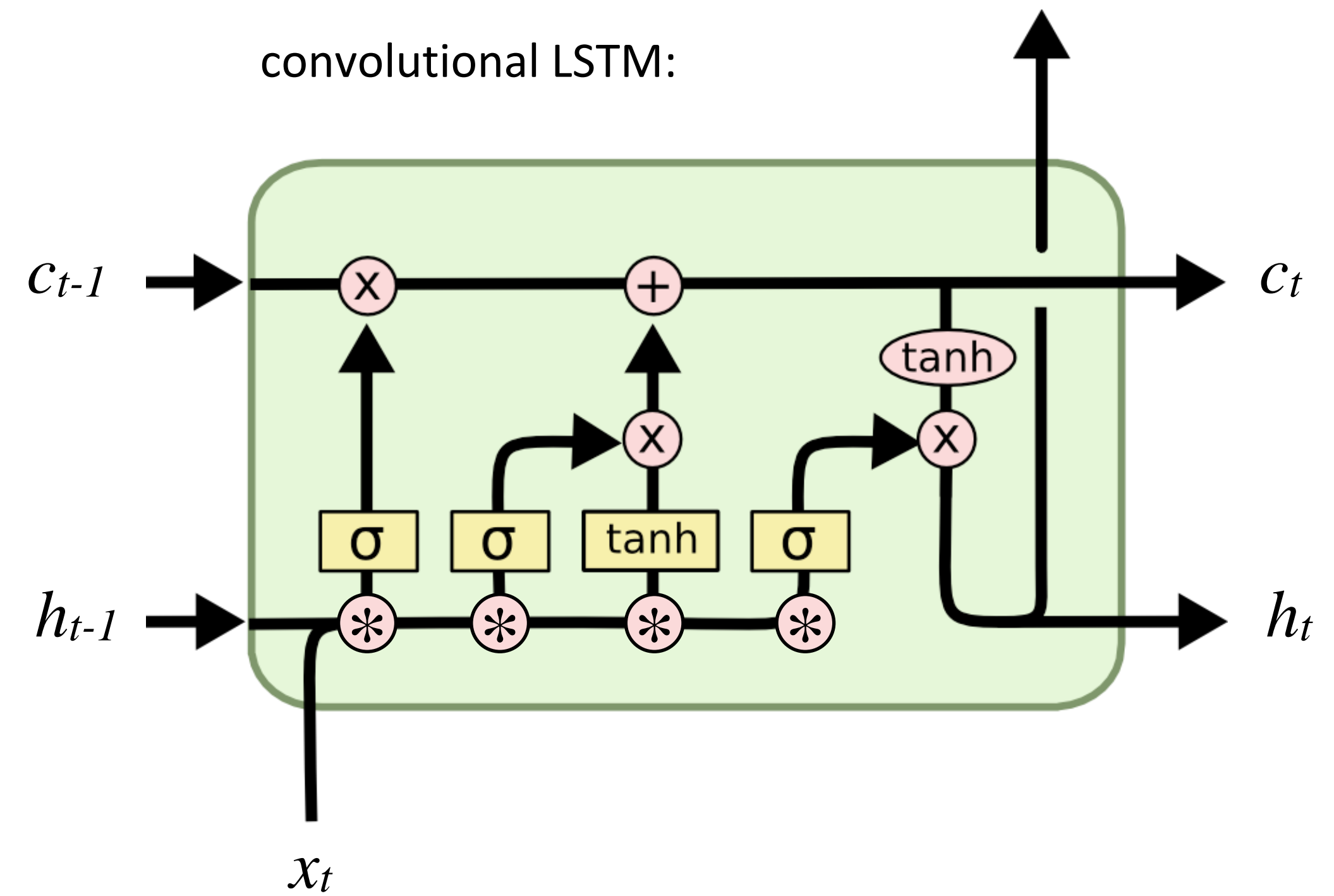
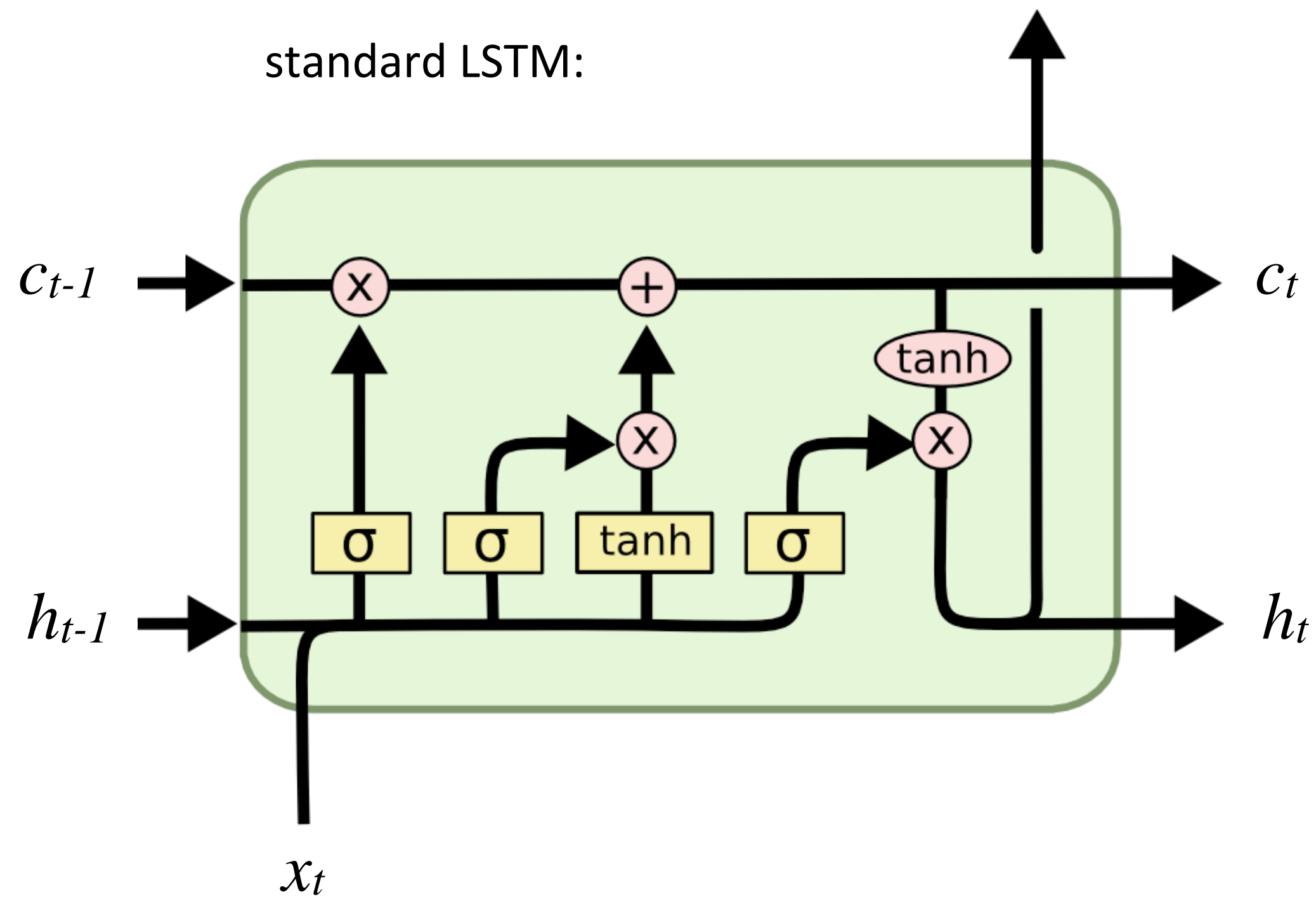
GRU (Gated Recurrent Unit), combines forget and input gates into a single update gate



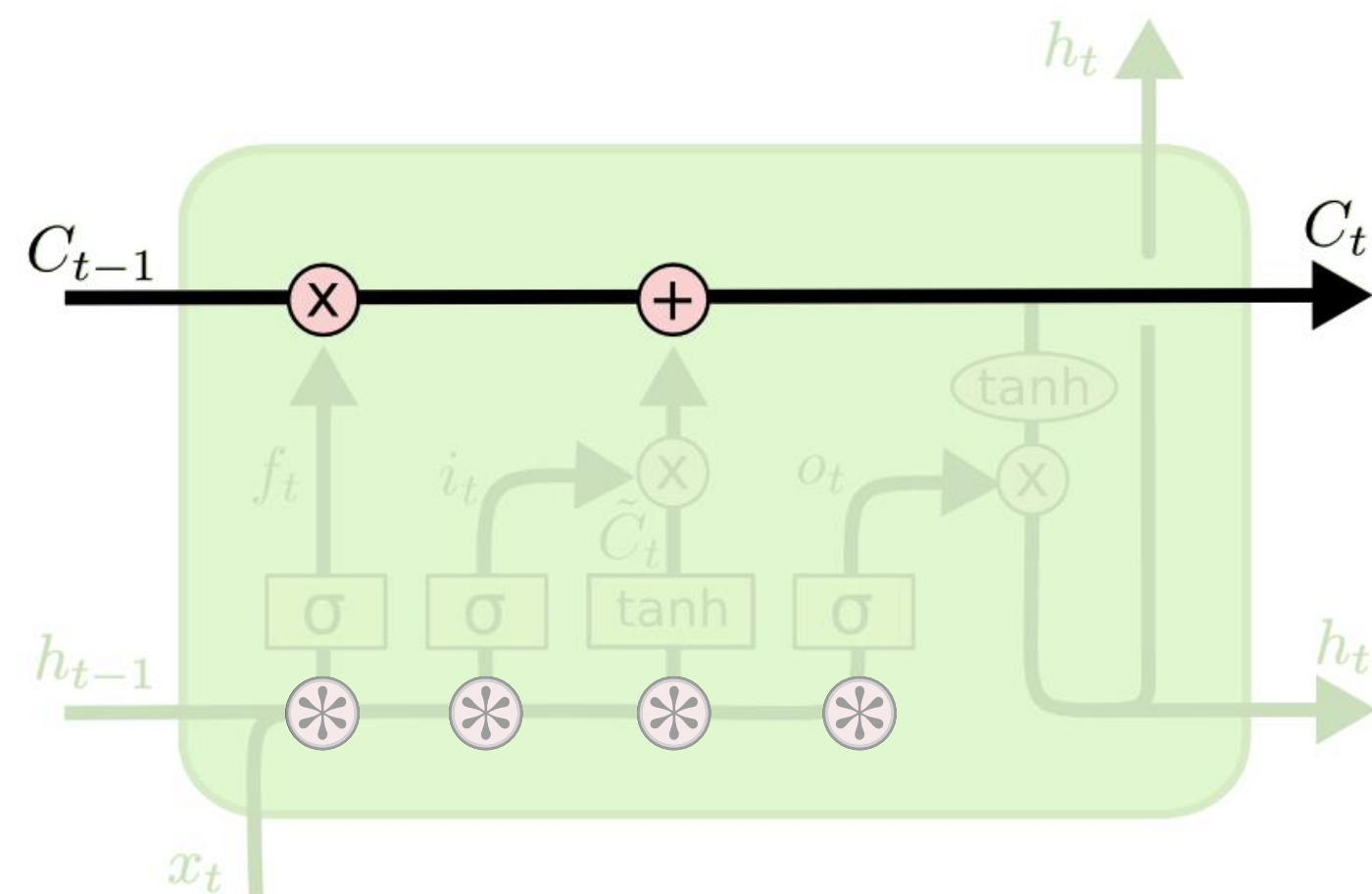


Let's look at some code!

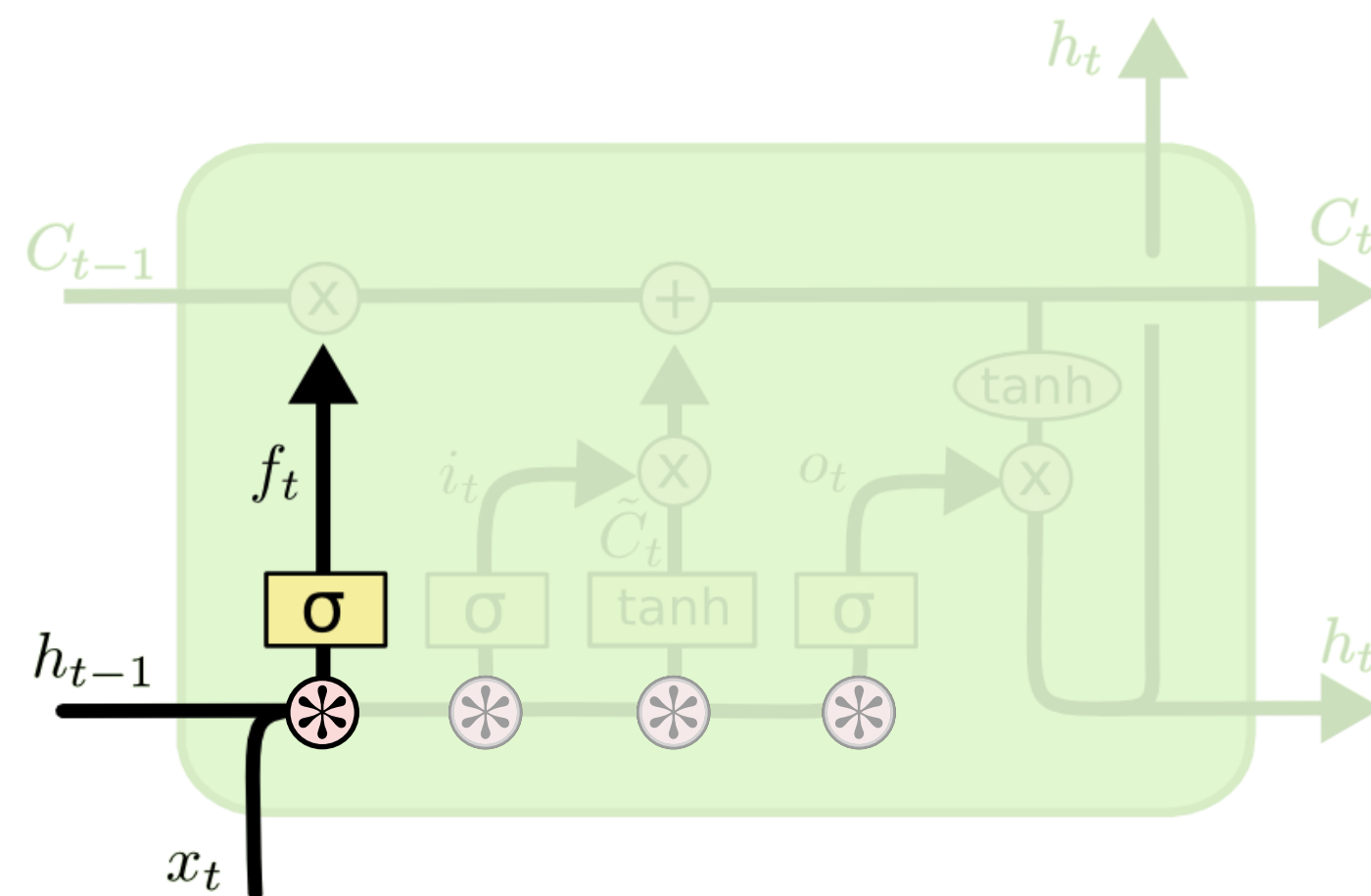
## Basic structure of a convolutional LSTM:



The core idea behind convolutional LSTMs is the same:



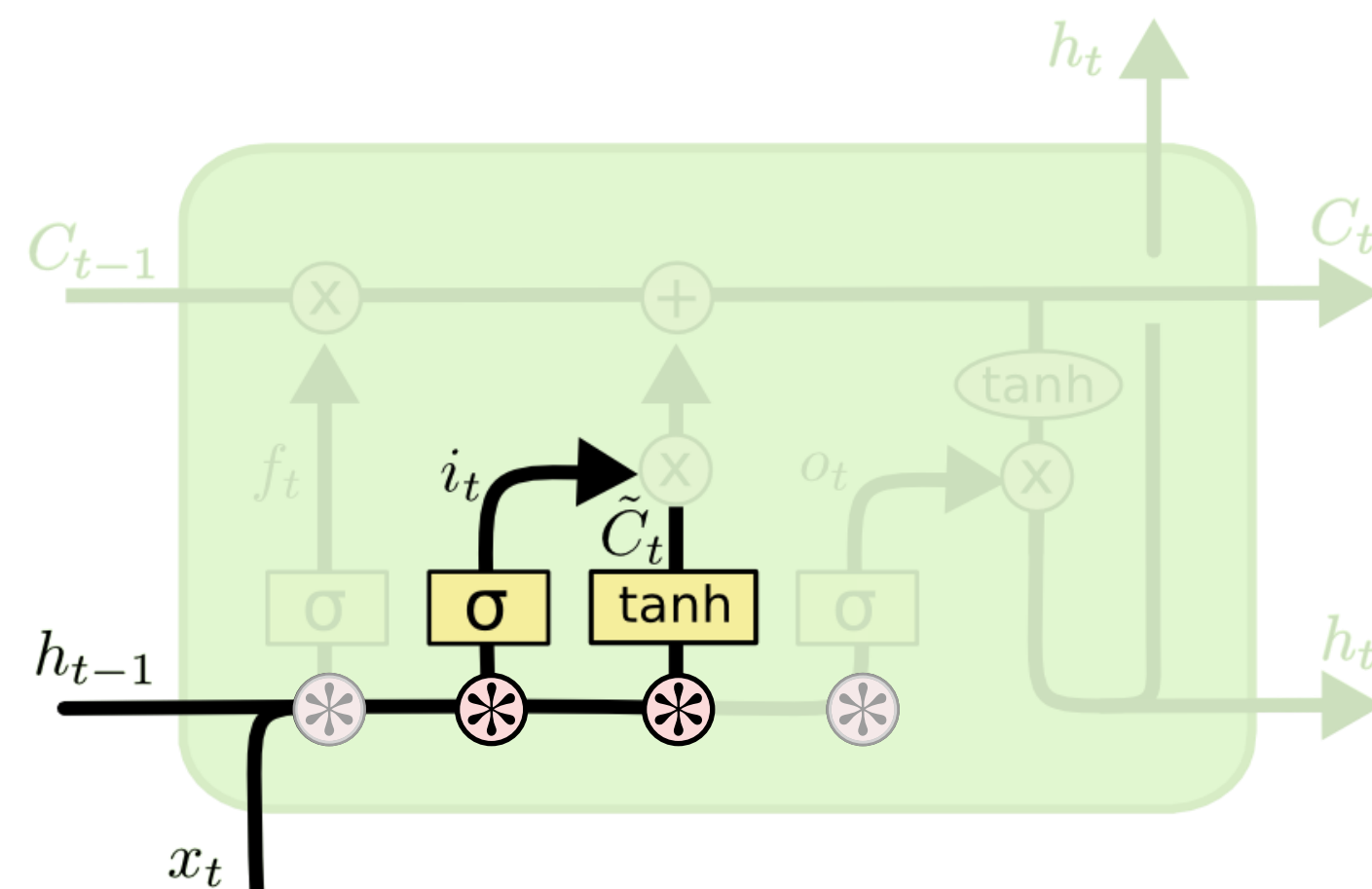
The **forget gate**: is there any information in the cell state that we no longer need?



convolution operation

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

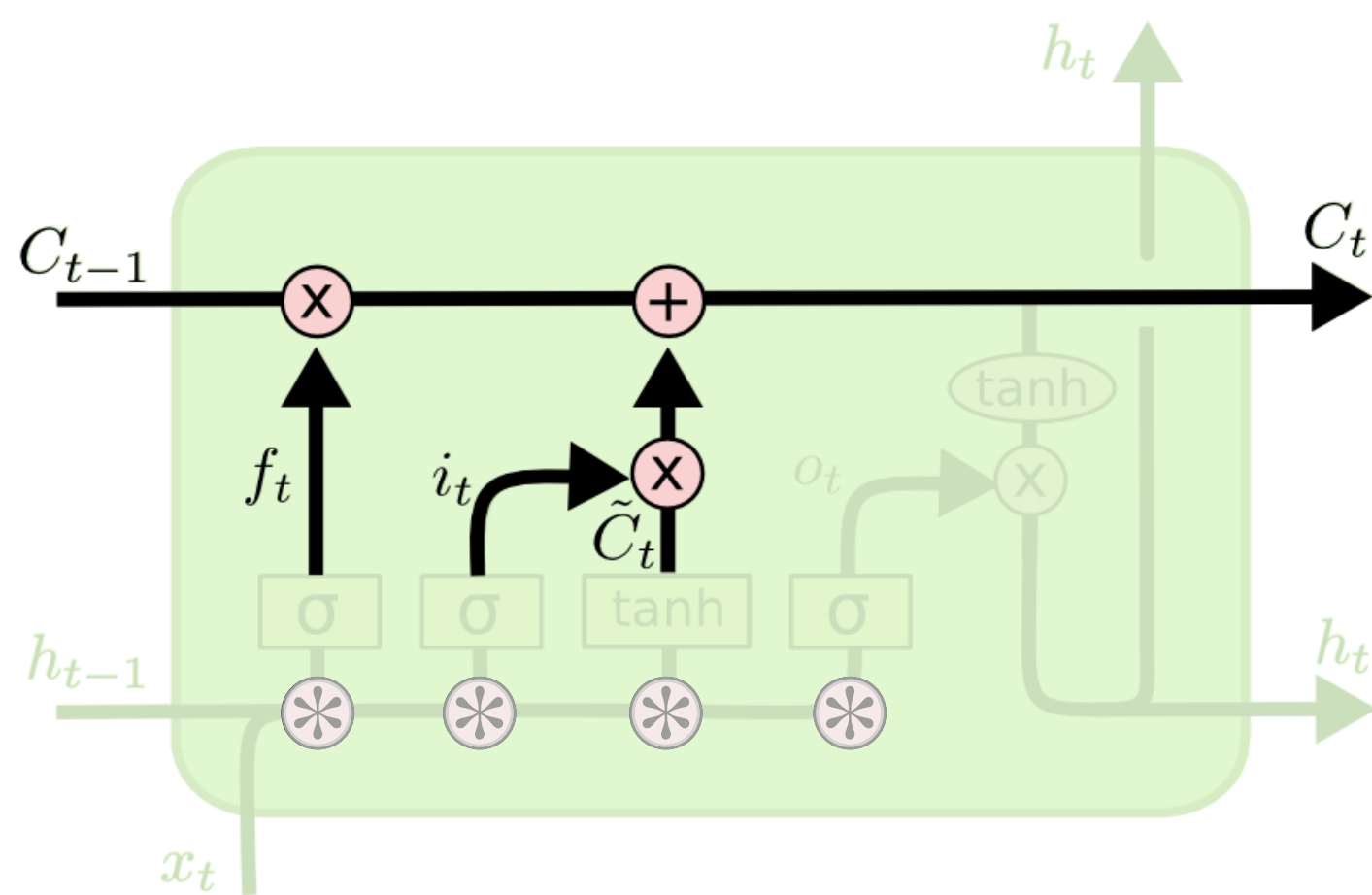
The **input gate** and the **candidate update**: what part of the input should we use to update the cell state?



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

$$\tilde{C}_t = \tanh(W_c * [h_{t-1}, x_t] + b_c)$$

Updating the cell state:

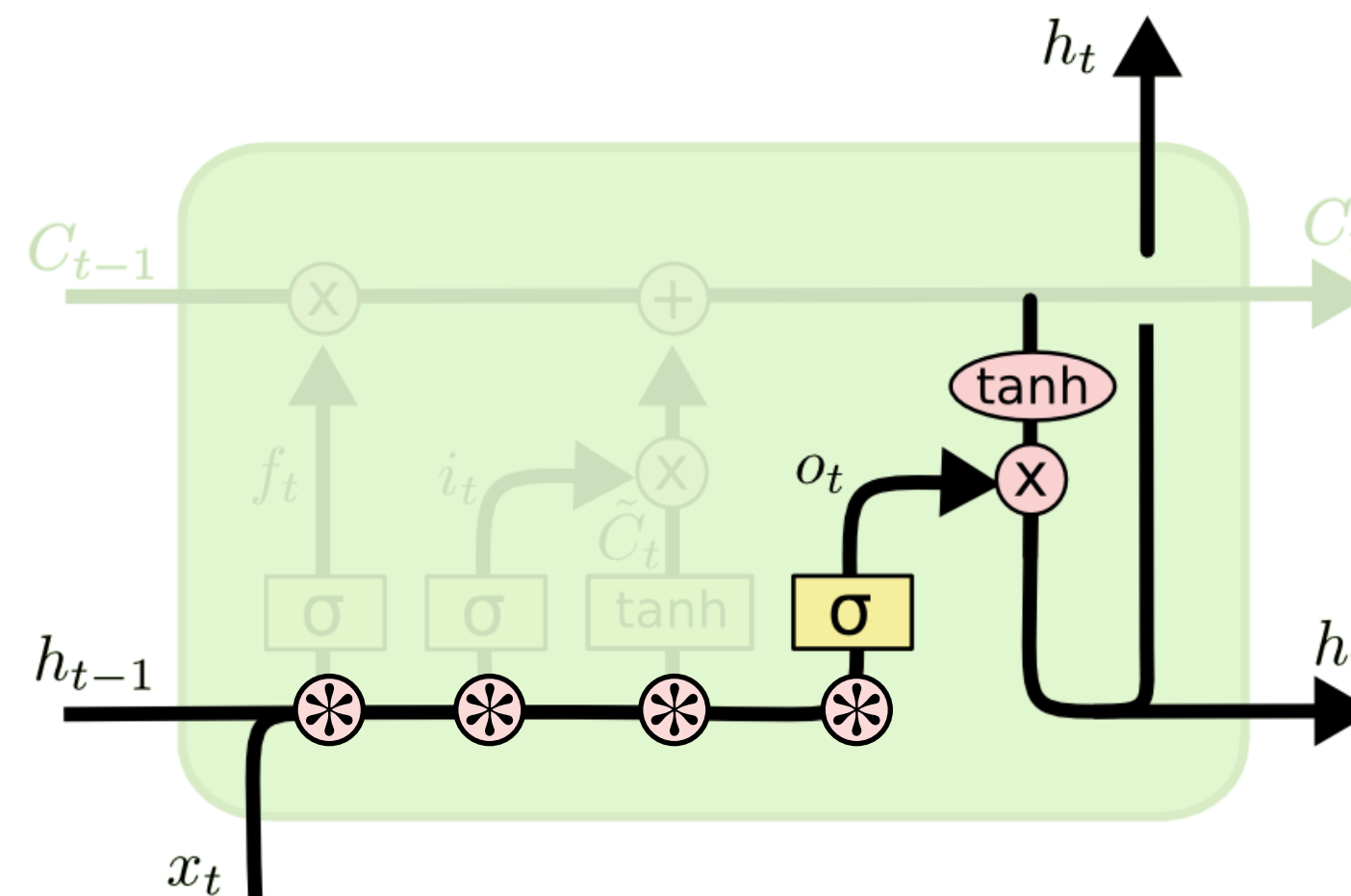


add the gated candidate values

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

forget part of the previous cell state

The **output gate**: what part of the cell state should influence the next output?



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

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

Basic structure of a convolutional LSTM:

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

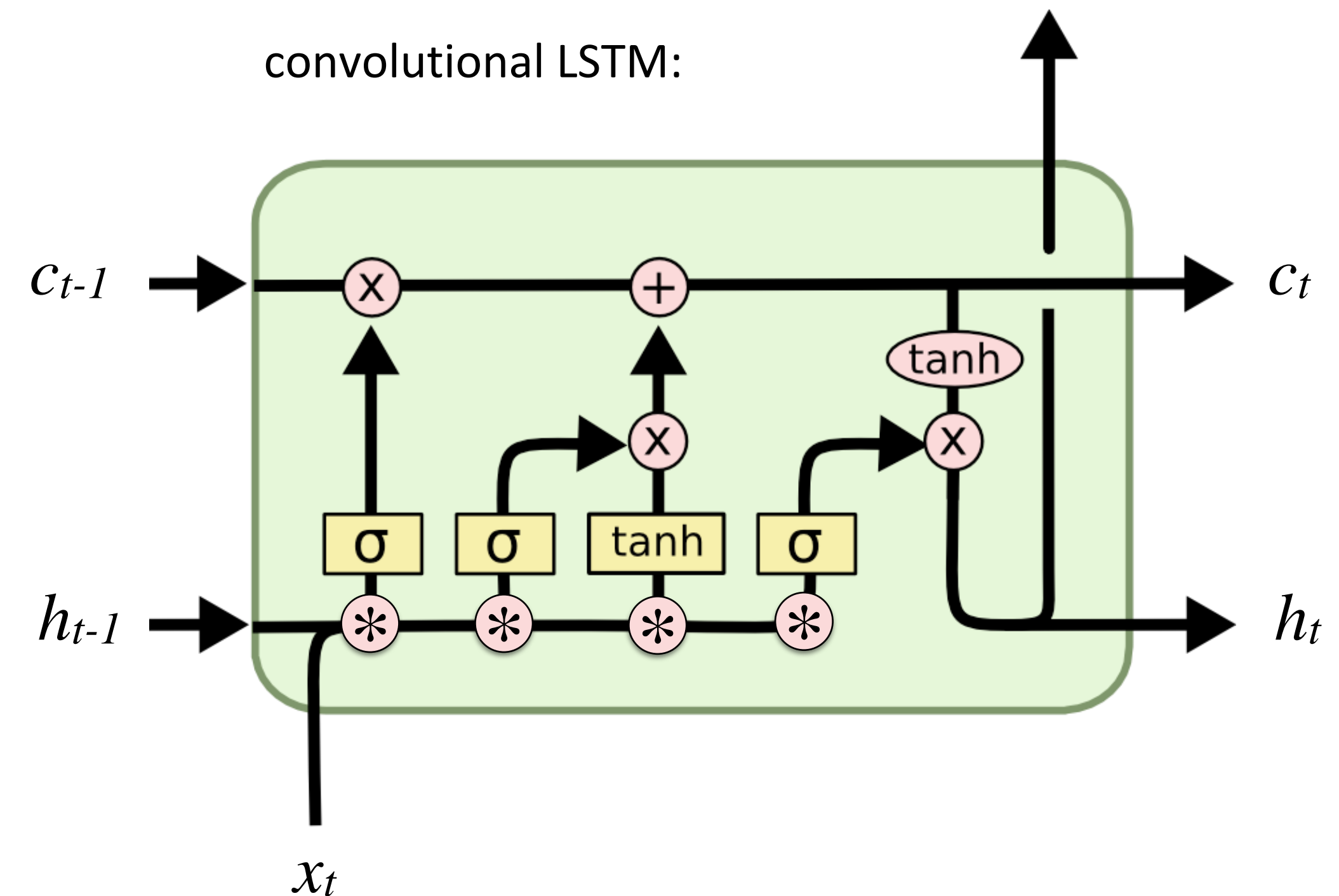
$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_c * [h_{t-1}, x_t] + b_c)$$

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

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

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





Basic structure of a convolutional LSTM (with peep-hole connections):

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + W_{cf} \odot C_{t-1} + b_f)$$

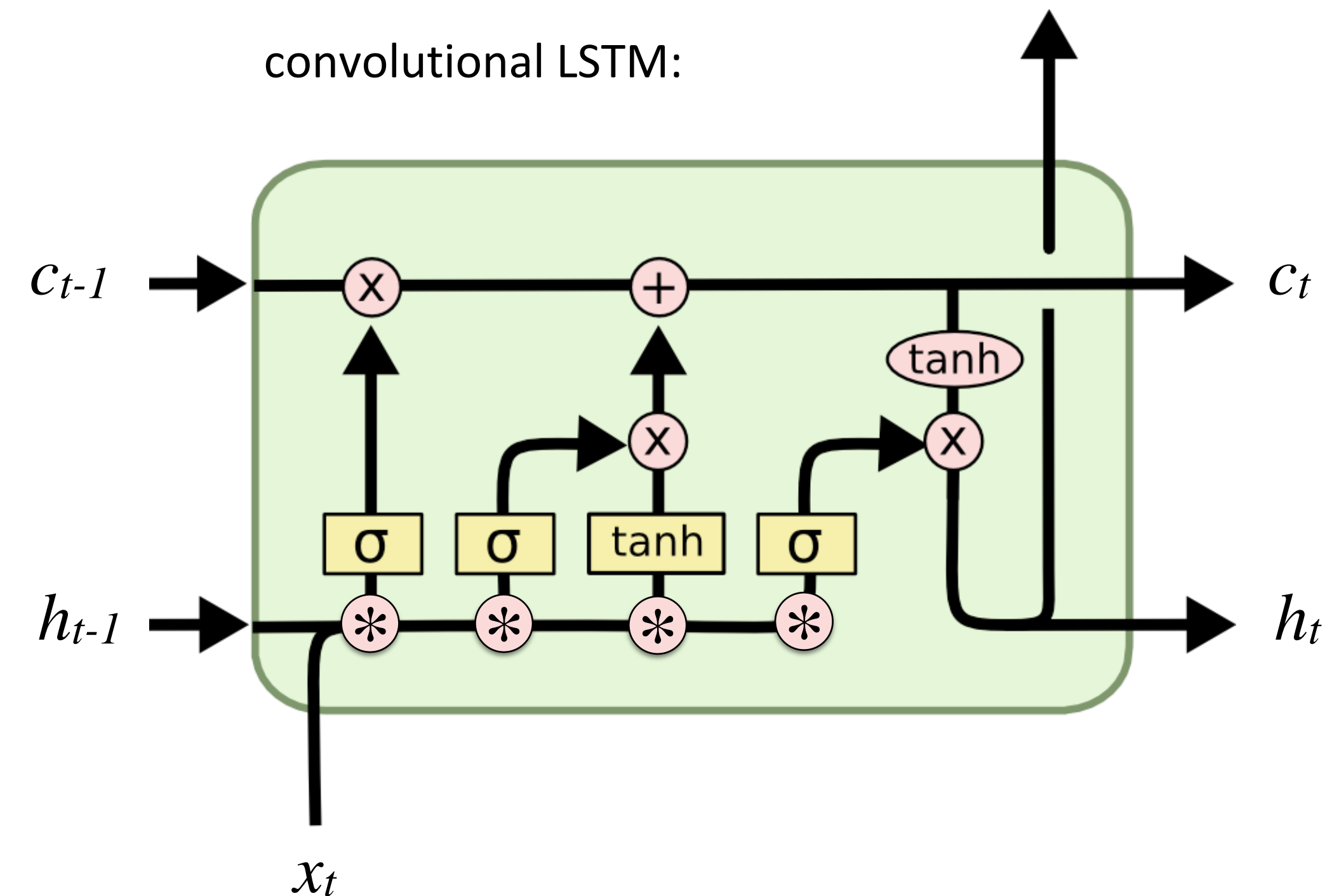
$$i_t = \sigma(W_i * [h_{t-1}, x_t] + W_{ci} \odot C_{t-1} + b_i)$$

$$\tilde{C}_t = \tanh(W_c * [h_{t-1}, x_t] + b_c)$$

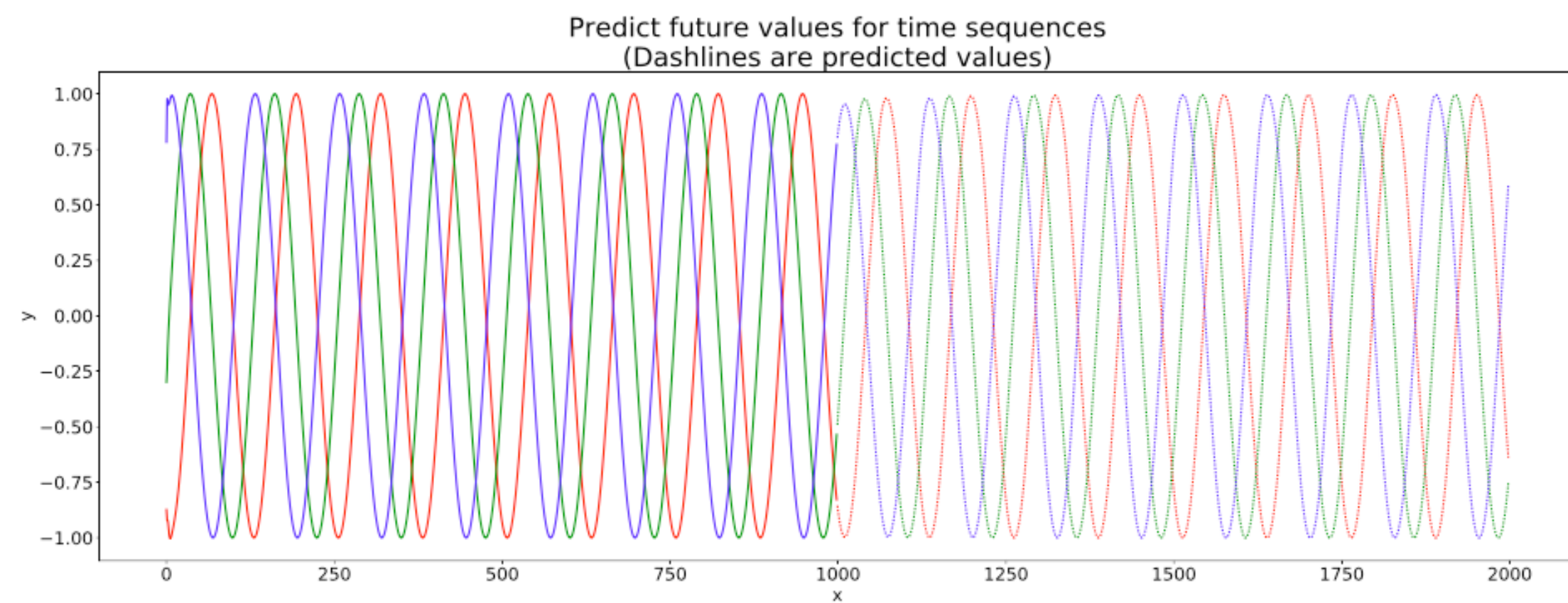
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

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

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



## Examples of application:



[https://github.com/pytorch/examples/tree/master/time\\_sequence\\_prediction](https://github.com/pytorch/examples/tree/master/time_sequence_prediction)

VIOLA:

Why, Salisbury must find his flesh and thought  
That which I am not aps, not a man and in fire,  
To show the reining of the raven and the wars  
To grace my hand reproach within, and not a fair are hand,  
That Caesar and my goodly father's world;  
When I was heaven of presence and our fleets,  
We spare with hours, but cut thy council I am great,  
Murdered and by thy master's ready there  
My power to give thee but so much as hell:  
Some service in the noble bondman here,  
Would show him to her wine.

KING LEAR:

O, if you were a feeble sight, the courtesy of your law,  
Your sight and several breath, will wear the gods  
With his heads, and my hands are wonder'd at the deeds,  
So drop upon your lordship's head, and your opinion  
Shall be against your honour.

<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

- ▶ Recurrent Neural Networks (RNNs) are the basic approach for sequential data.
- ▶ RNNs suffer from exploding and vanishing gradients and only preserve short-term memory.
- ▶ Long Short-Term Memory (LSTM) networks address these two issues.