Implementing ANNs with TensorFlow

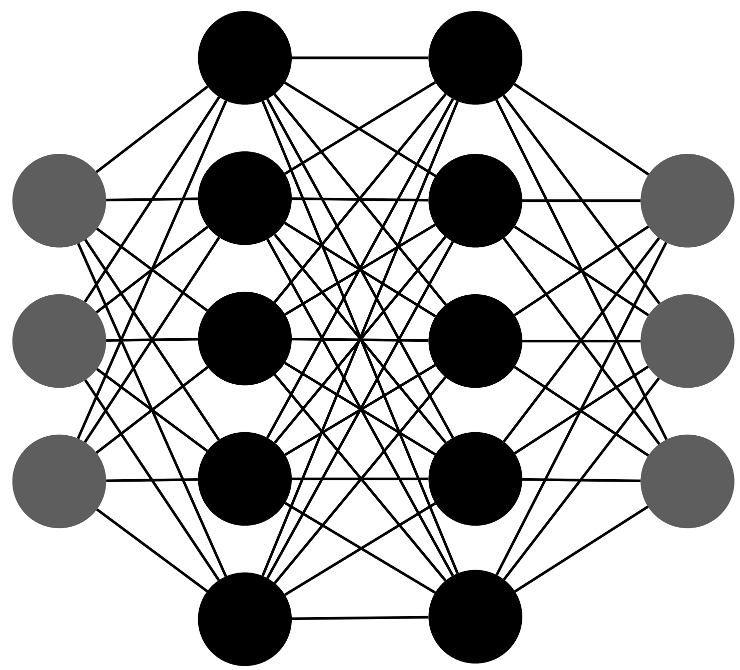
Session 09 - Recurrent Neural Networks

1. Motivation
2. Recurrent Neural Networks
3. Backpropagation through Time
4. Gated Recurrent Units
5. LSTMs

Motivation

Feed Forward Neural Networks

Feed forward neural networks process a static input and return a static label.



“cute dog”

Sequential Data

- Human understanding of the real world is based on processing a stream of data.
- These kinds of data are called <u>sequential data</u>.
- Examples:
  - Text,
  - Sound/Speech,
  - Videos,
  - Temporal Data (e.g. temperatures or stock values).

How to Process Sequential Data?

- Sequential data can have various (and possibly unrestricted) length.
    - Model can’t have fixed input size.
- The “meaning" of an input depends on the inputs that came before.
    - Model needs some form of internal memory.

Task Setup

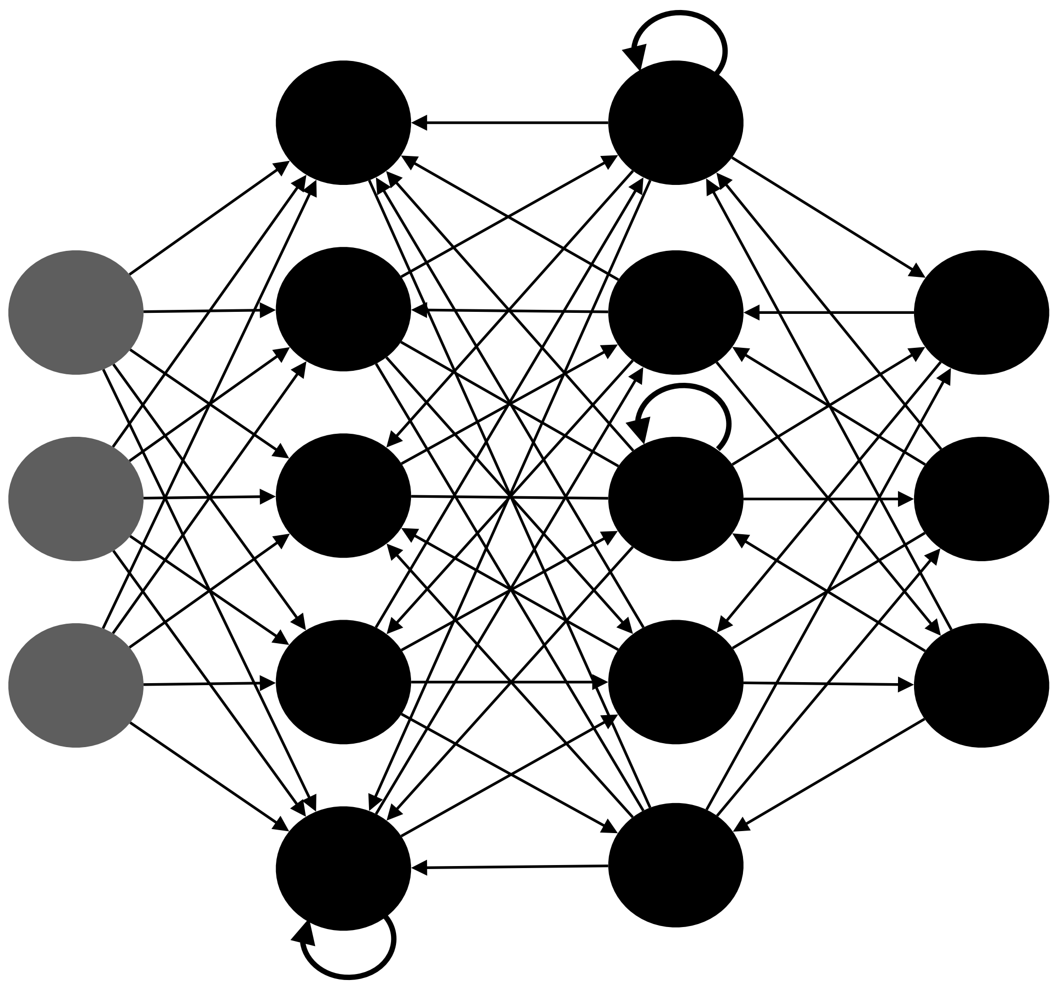
- The input is a sequence of data points: $(\overrightarrow{x}_t)_{t=1}^{T}$.
- The network is fed these datapoints one after the other.
- The labels can be many different things:
    - One label: e.g. sentiment analysis of text: $\vec{t}$
    - Another sequence: e.g. speech-to-text: $(\vec{t}_t)_{t=1}^{T}$
    - The same sequence shifted: e.g. prediction: $(\overrightarrow{x}_t)_{t=2}^{T+1}$

Recurrent Neural Networks

General Definition

- Recurrent neural networks (RNNs) are neural networks which allow self or backward connections.

Input

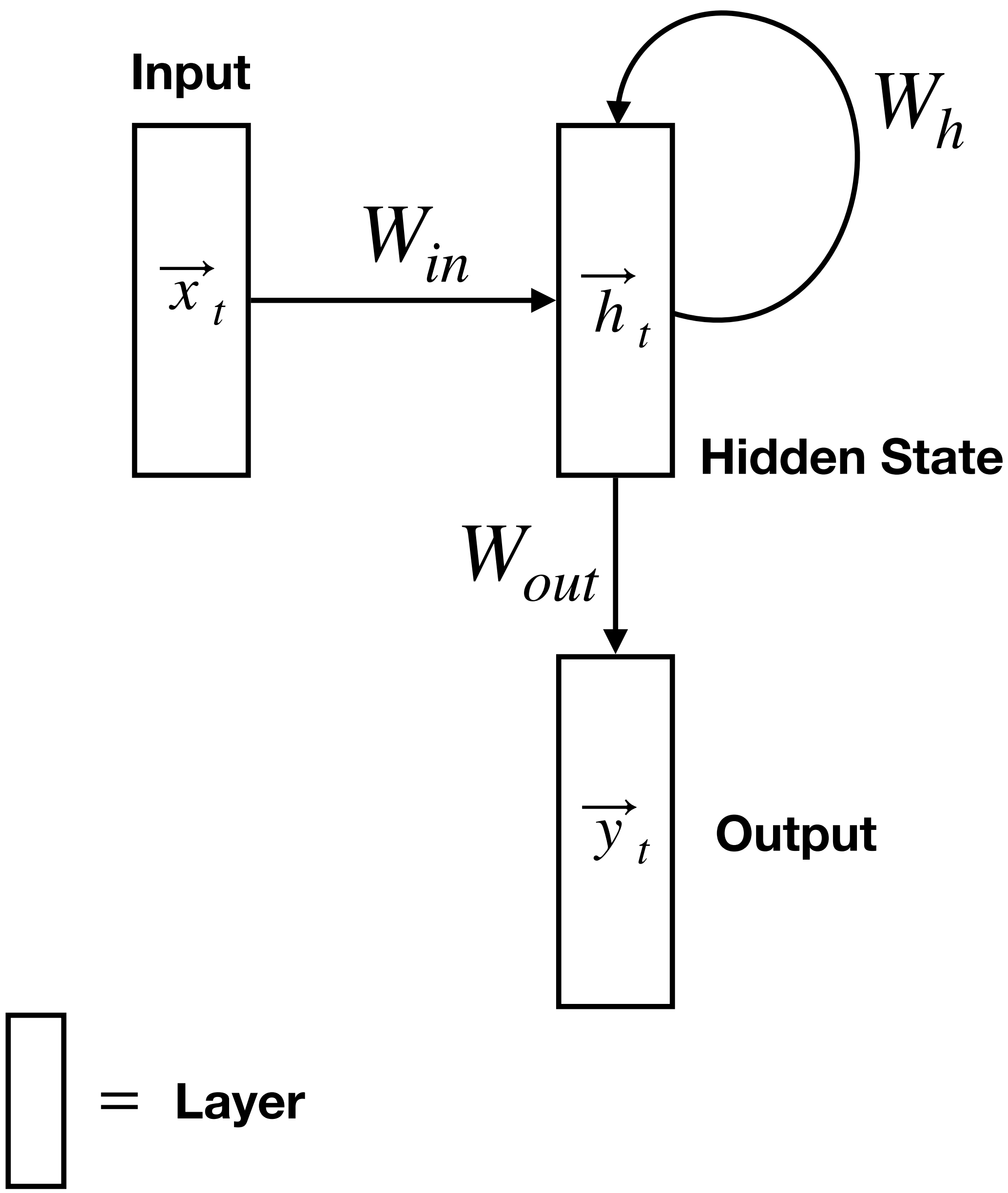


The state of the network always depends on its previous state.

- Such a network can exhibit very complex dynamics. In this form it is unpractical for deep learning.

Vanilla RNN

- The vanilla RNN is the most simple RNN setup there is.



Recursive definition

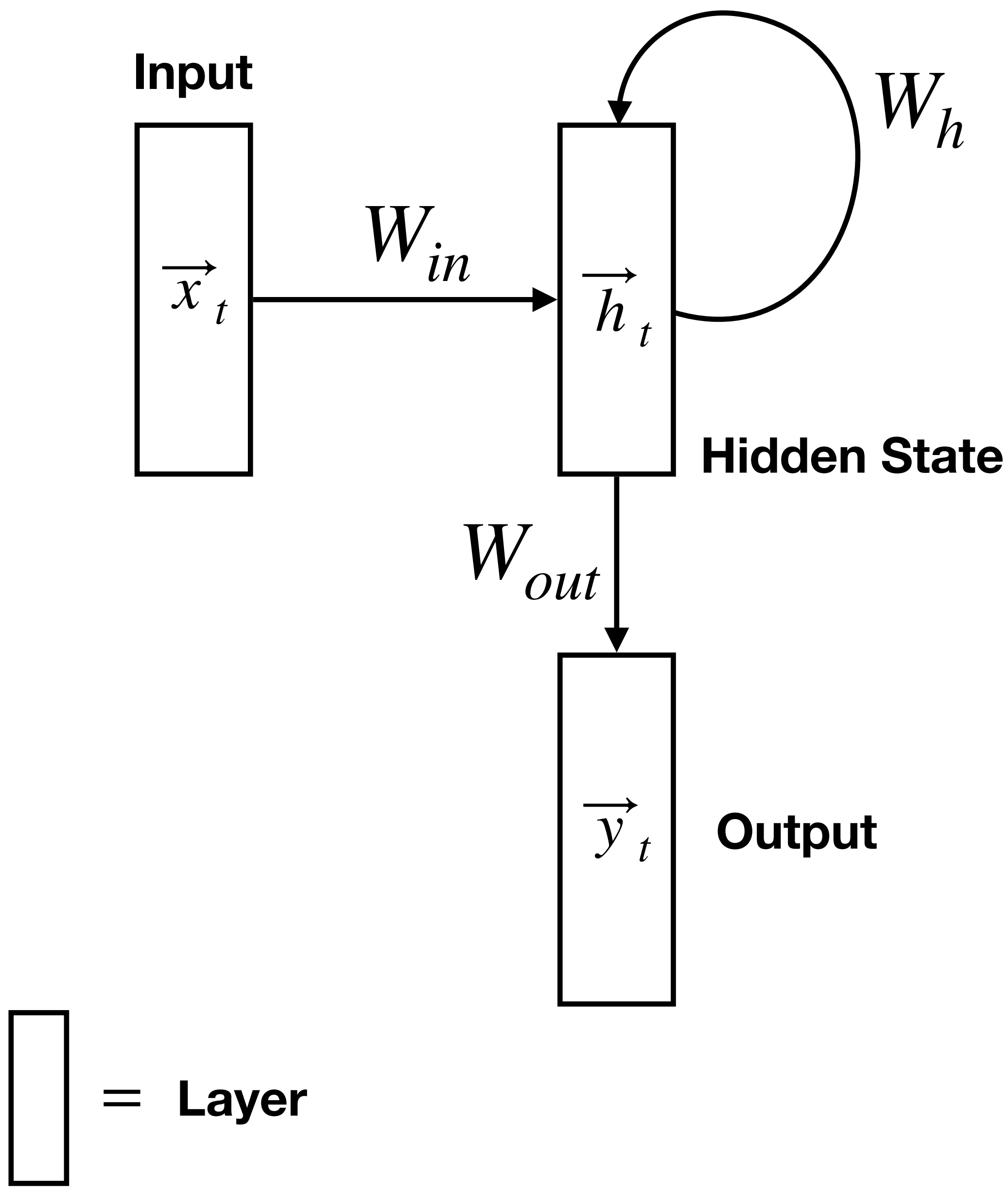
$$\vec{h}_t = \sigma(W_{in}\vec{x}_t + W_h\vec{h}_{t-1} + \vec{b}_h)$$

$$\vec{y}_t = f(W_{out}\vec{h}_t + \vec{b}_{out})$$

$\vec{h}_t$	current hidden state
$\vec{h}_{t-1}$	old hidden state
W_{in}	weight matrix from input to hidden
W_h	weight matrix from hidden to hidden
$\vec{b}_h$	bias for hidden layer
σ	activation function for hidden layer
W_{out}	weight matrix from hidden to output
$\vec{b}_{out}$	bias for output layer
f	activation function for output layer

Vanilla RNN - Dimension Check

- The vanilla RNN is the most simple RNN setup there is.



Recursive definition

$$\vec{h}_t = \sigma(W_{in}\vec{x}_t + W_h\vec{h}_{t-1} + \vec{b}_h)$$

$$\vec{y}_t = f(W_{out}\vec{h}_t + \vec{b}_{out})$$

Layer dimensions: $\vec{x}_t \in \mathbb{R}^m, \vec{h}_t \in \mathbb{R}^n, \vec{y}_t \in \mathbb{R}^p$

$\rightarrow W_{in} \in \mathbb{R}^{n\times m}$

$\rightarrow W_h \in \mathbb{R}^{n\times n}$

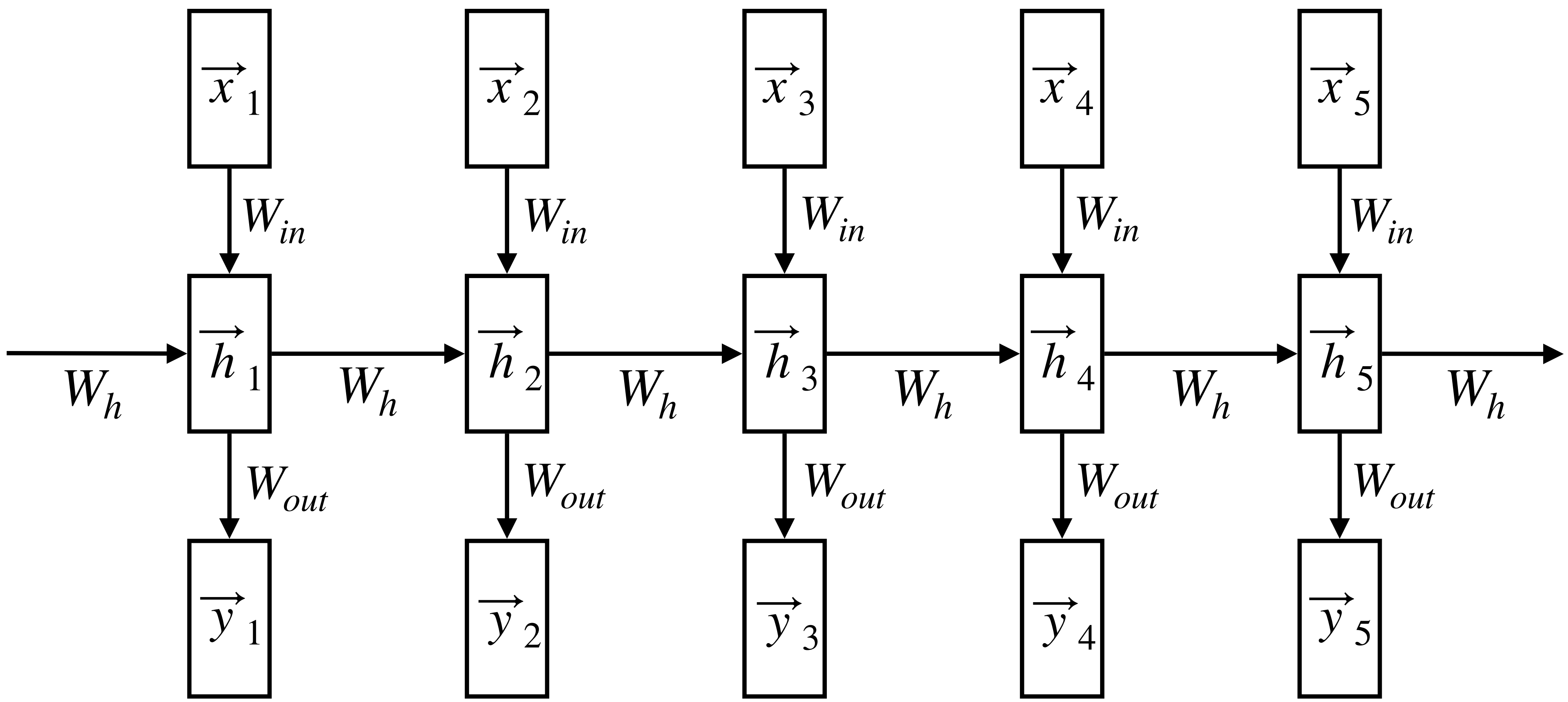
$\rightarrow \vec{b}_h \in \mathbb{R}^n$

$\rightarrow W_{out} \in \mathbb{R}^{p\times n}$

$\rightarrow \vec{b}_{out} \in \mathbb{R}^p$

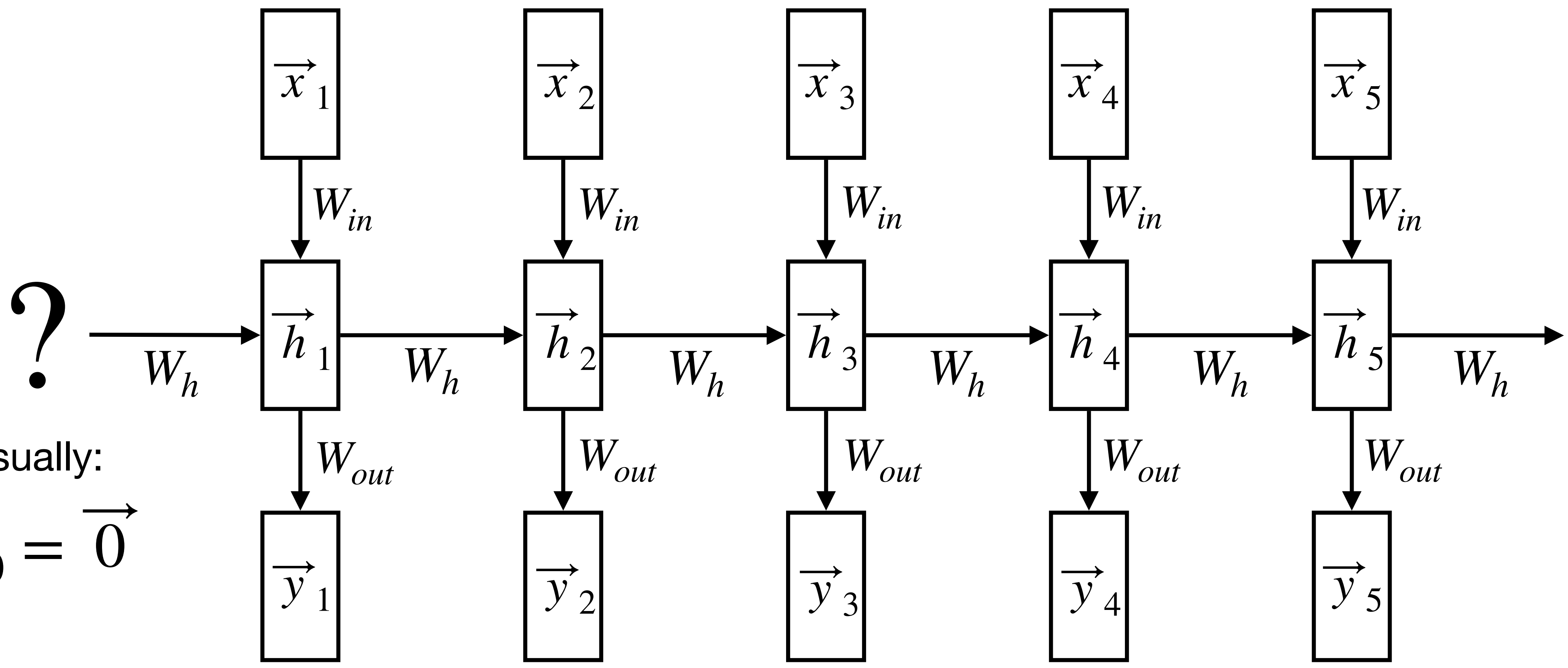
Unfolding RNNs

- <u>Unfolding an RNN</u> is a visualization technique that helps understanding the involved computations:



Hidden State Initialization

- It also reveals that we need to initialize the hidden state, because it is required for computing the first hidden state.



Usually:

$$\vec{h}_0 = \vec{0}$$

Example: Modeling a Text

Simple Example

- A simple example should help in understanding the computational principle.

- We can train an RNN to model a given text, i.e. it should learn to predict the next character given the sequence of previous characters.

- Example Corpus: Bible

Feeding Text to RNN

- But how can we feed the text to the RNN?

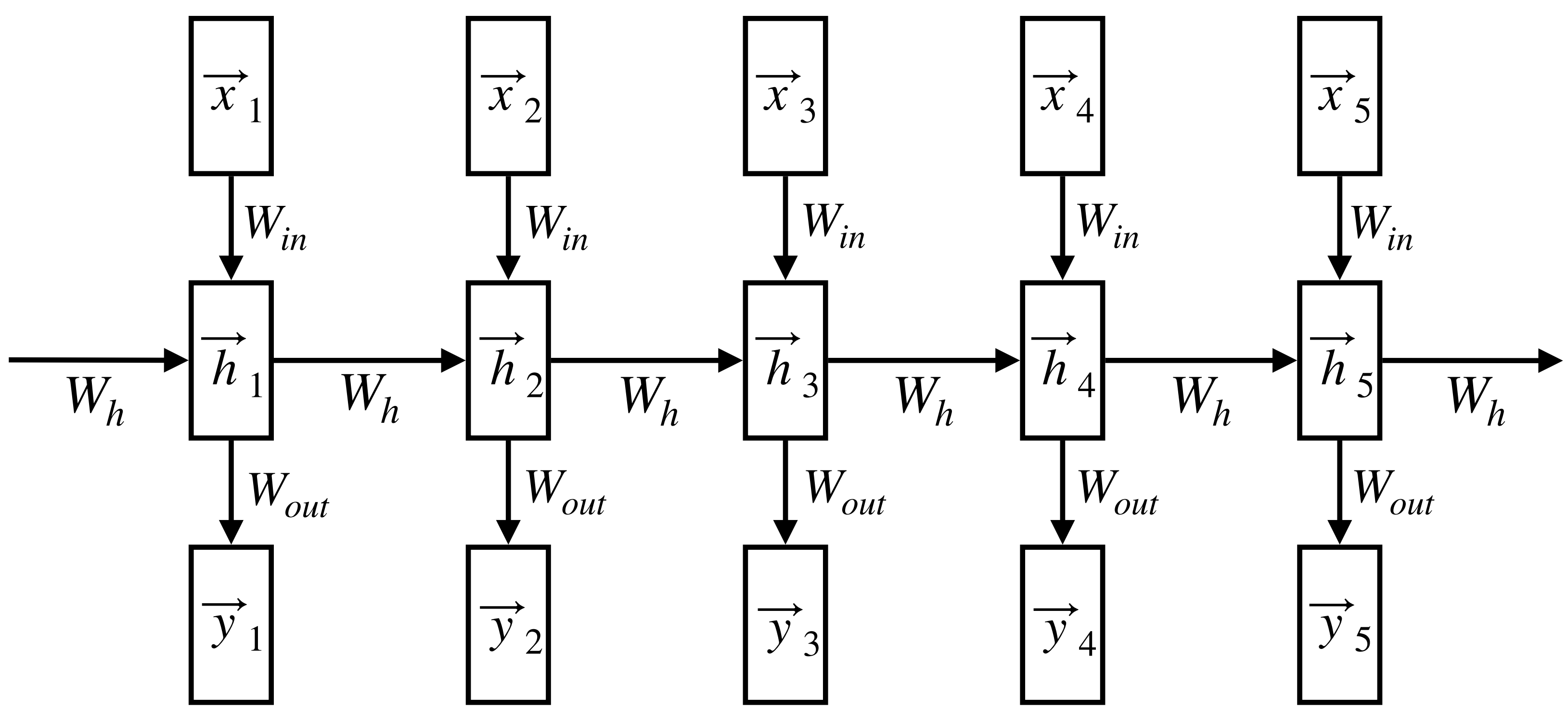
Line in script → Sequence of characters → One-hot repr. of characters

In the beginning God created …

```
[I,n, ,t,h,e, ,b,e,g
,i,n,n,i,n,g, ,G,o,d
, ,c,r,e,a,t,e,d, …]
```

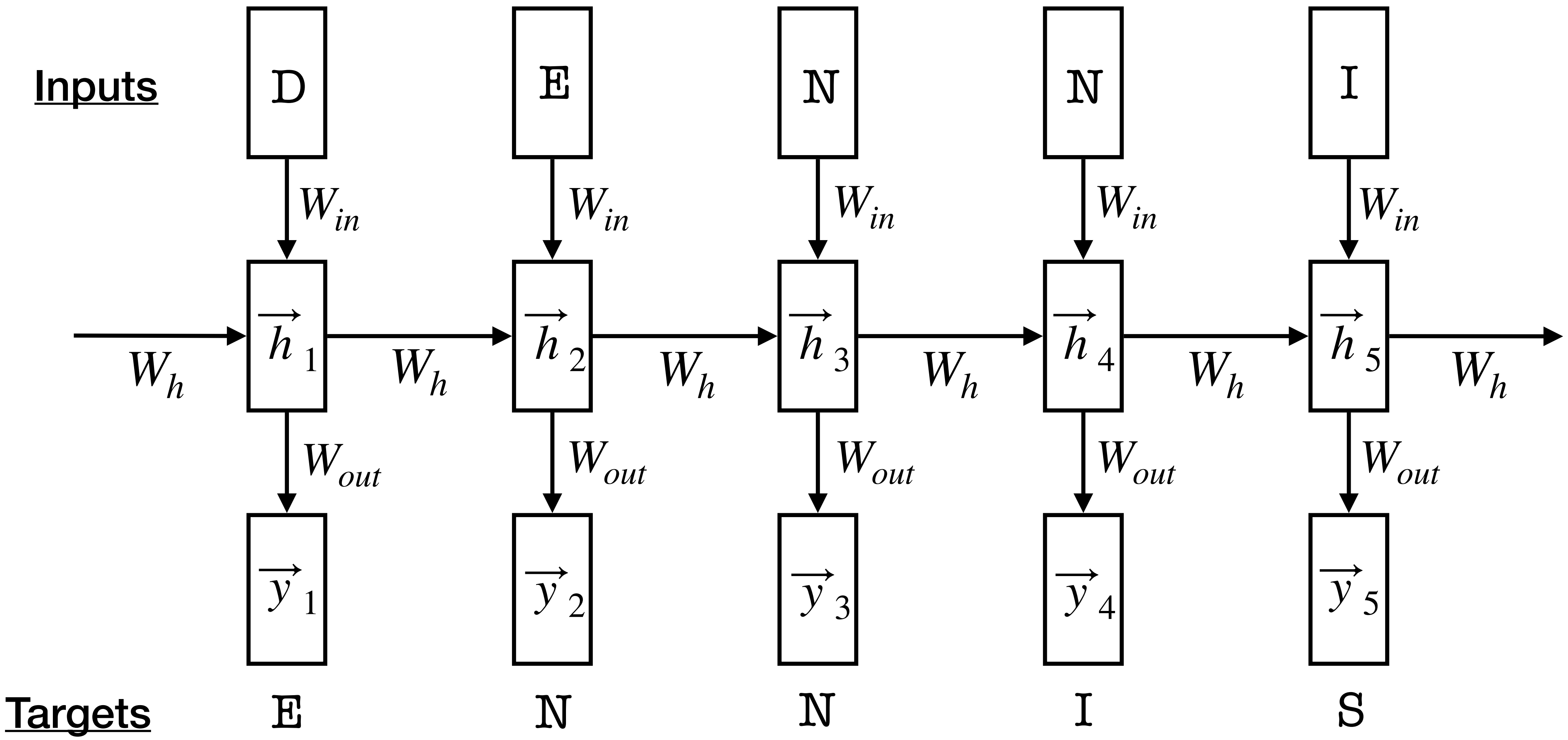
$$\left[\begin{pmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}, \ldots \right]$$

number of indices= size of vocabulary



Simple Example

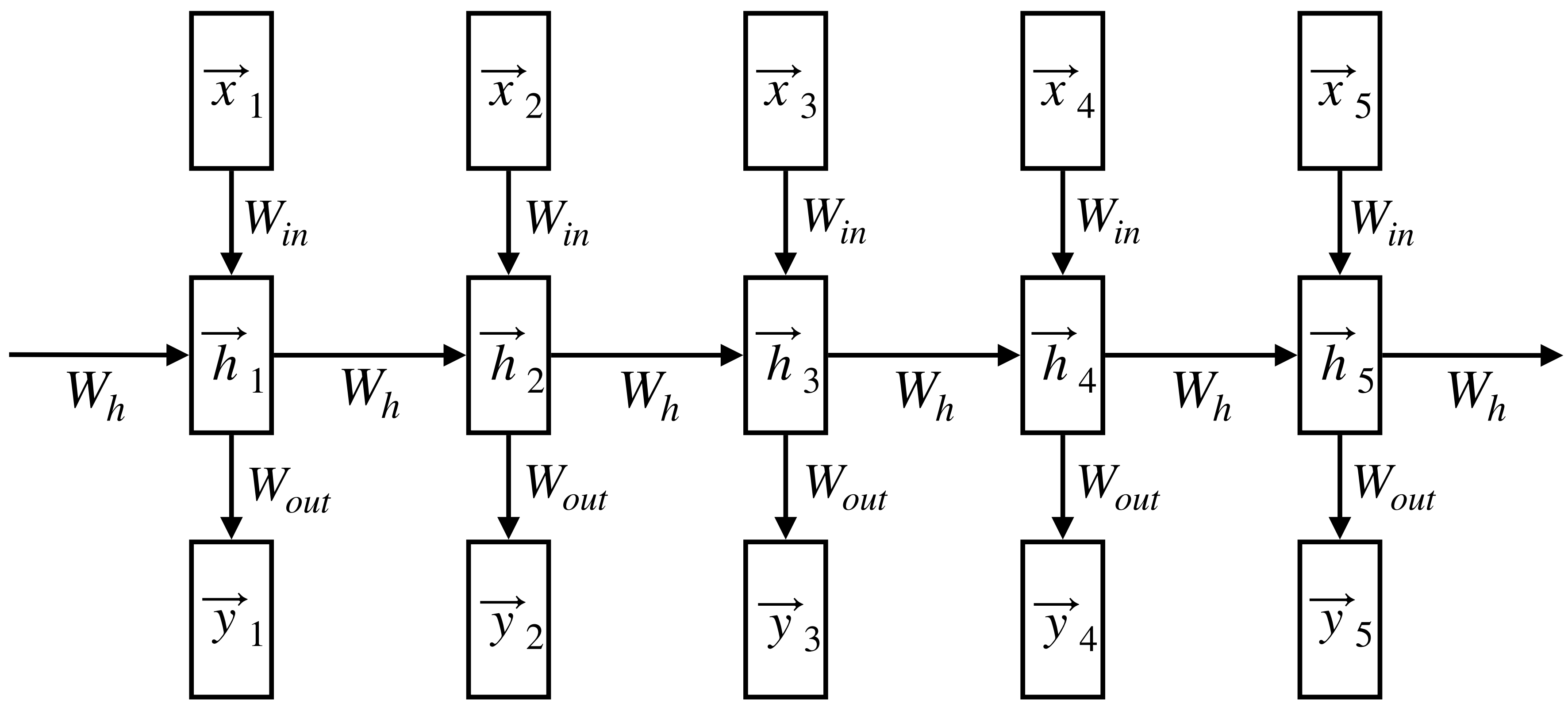
- In each step the network is given one more character and it tries to predict the next one



Training RNNs

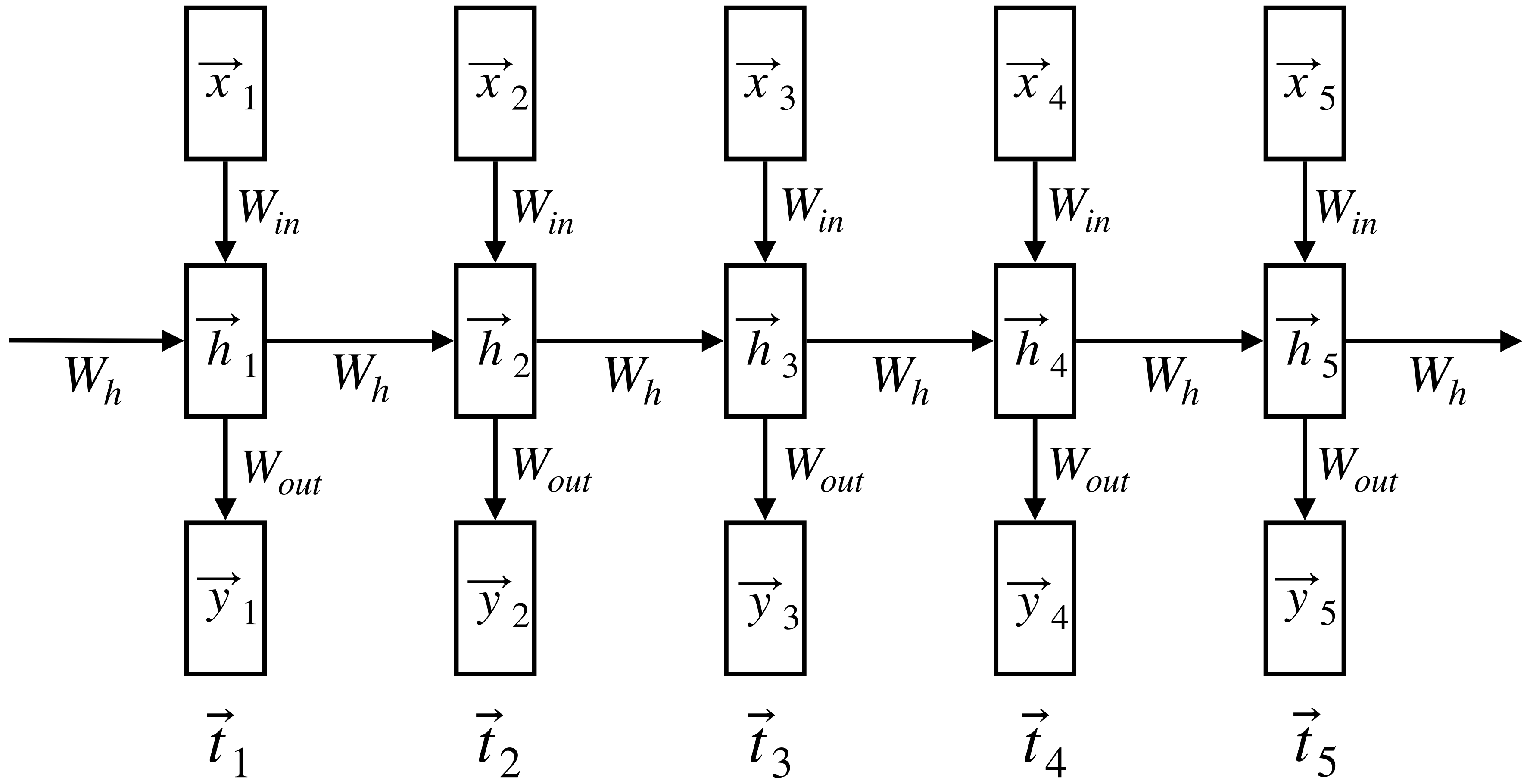
Backpropagation Through Time

- <u>Backpropagation Through Time</u> (BPTT) describes the algorithm used to train RNNs.
- Although at first sight it could seem to be quiet complex it is actually not.



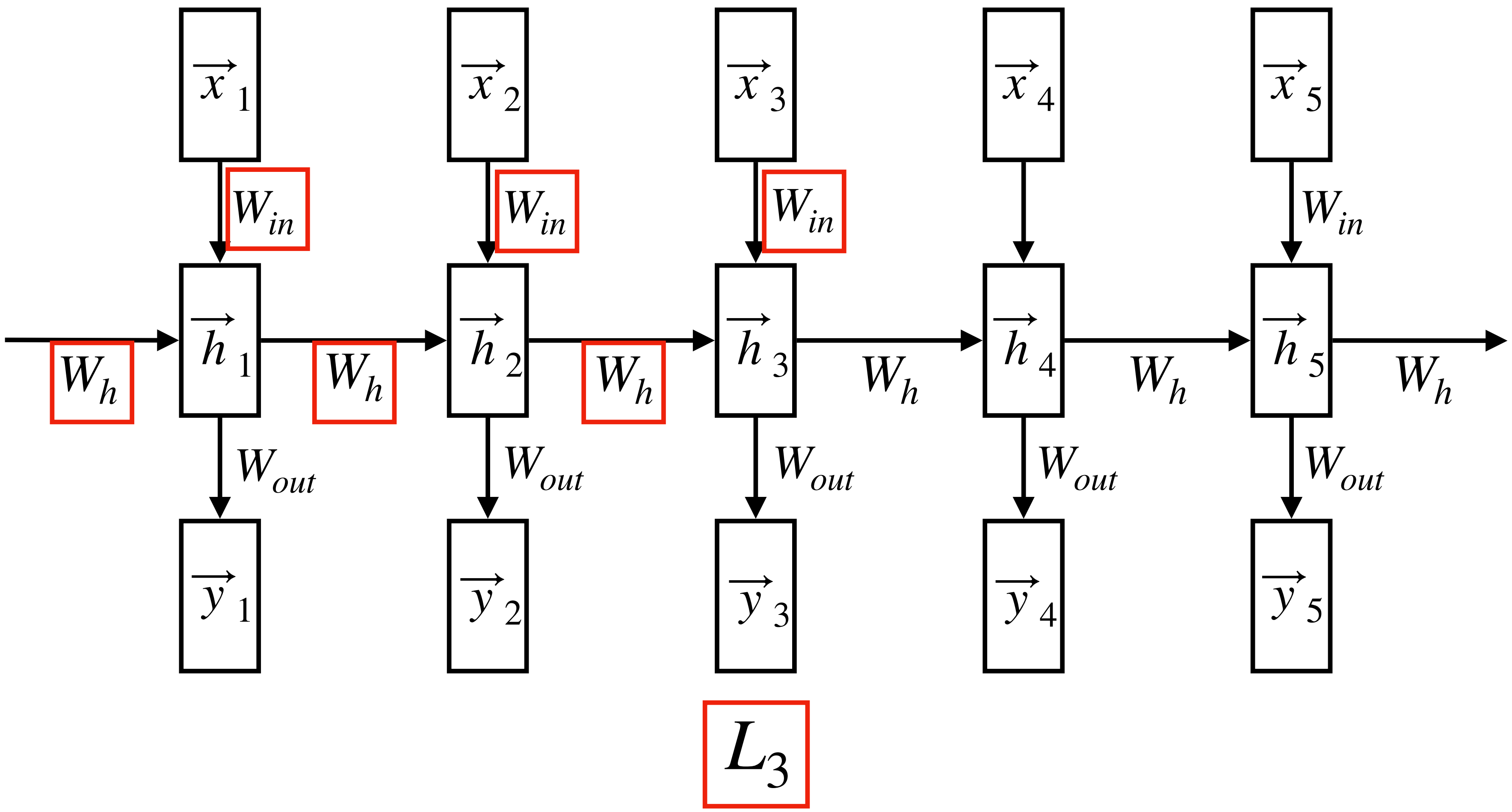
Backpropagation Through Time

- First we can see that at each time step we get a loss term of how well the output matched our target: $L_t = L(\vec{t}_t, \vec{y}_t)$
- The gradient of the loss in respect to a parameter θ is therefore the average of all L_t: $\nabla_\theta L = \frac{1}{N} \sum_t \nabla_\theta L_t$.



Backpropagation Through Time

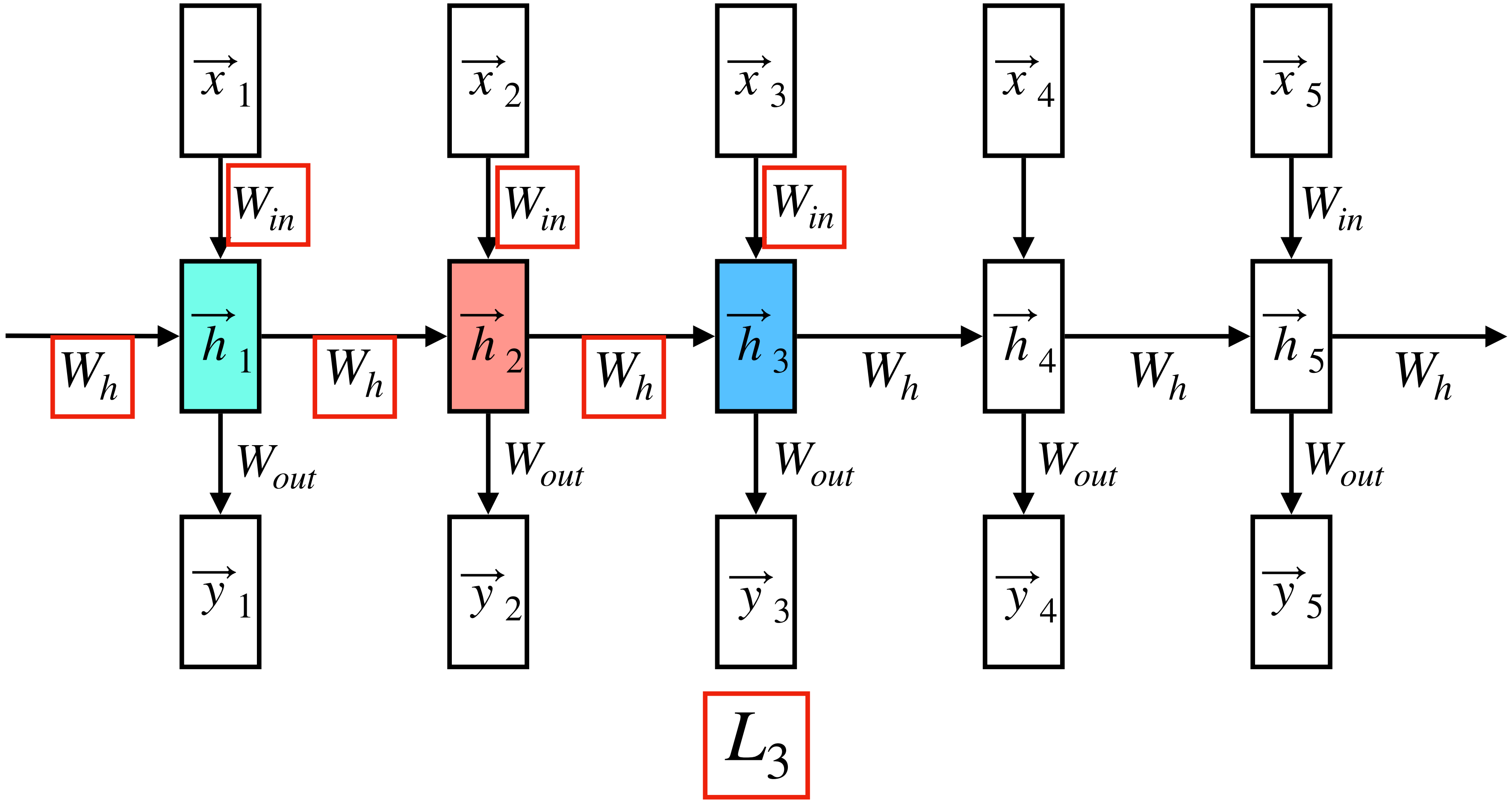
- The question then is how do we compute the gradient of one of these losses in respect to a parameter: $\nabla_\theta L_t$.
- E.g. the gradient of loss L_3 in respect to weights in W_{in} and W_h.



Backpropagation Through Time

- The problem here is that L_3 is dependent on $\vec{y}_4$, which is a function in which W_{in} and W_h appear several times.
- Calculating this derivative would result in a quite complex term.

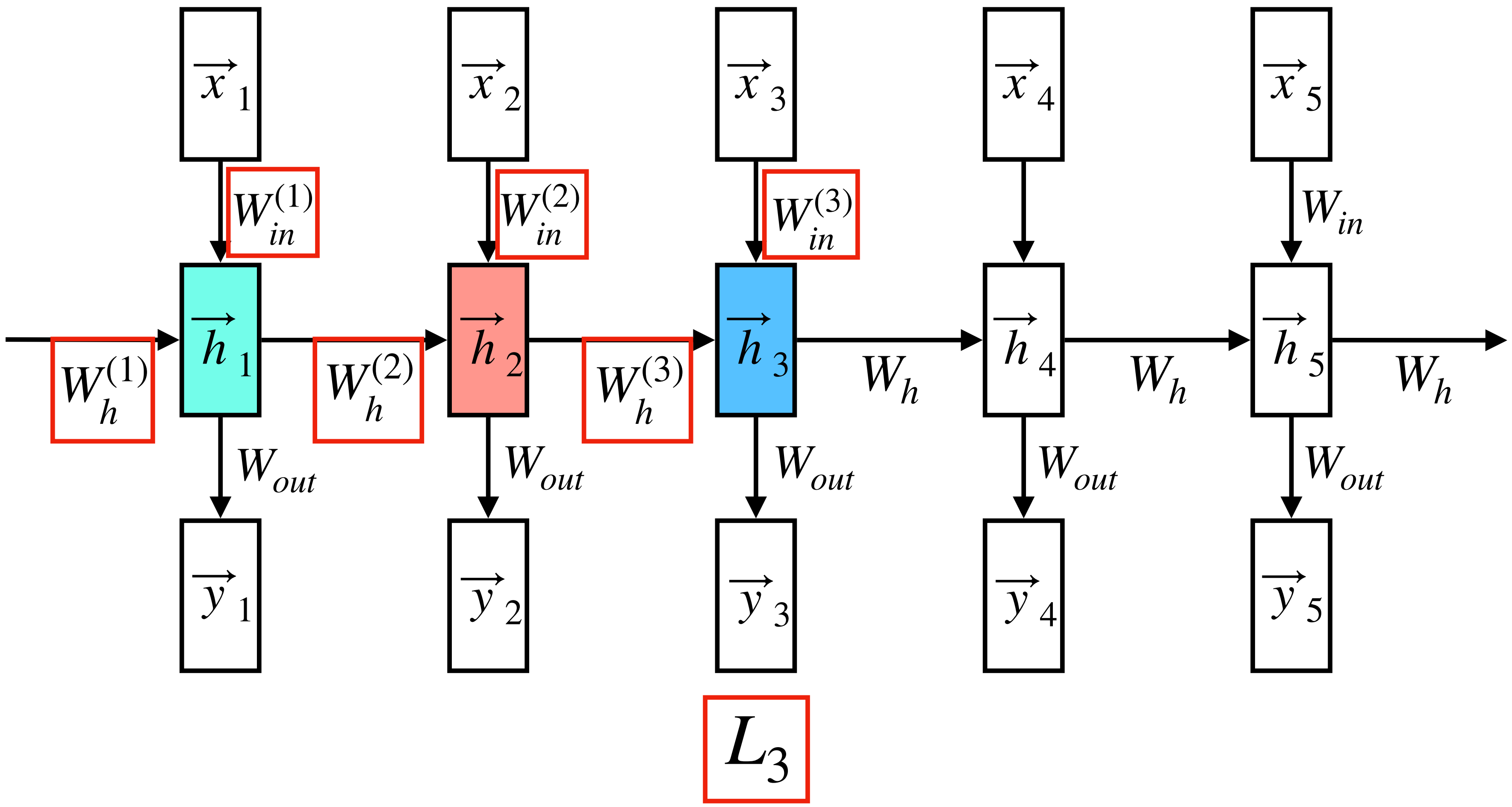
$$\vec{y}_3 = f(W_{out}\sigma(W_{in}\vec{x}_3 + W_h\sigma(W_{in}\vec{x}_2 + W_h\sigma(W_{in}\vec{x}_1 + W_h\vec{h}_0 + \vec{b}_h) + \vec{b}_h) + \vec{b}_h) + \vec{b}_{out})$$



Backpropagation Through Time

- As a trick we introduce dummy copies for each variable: $\theta \rightarrow \theta^{(1)}, \theta^{(2)}, \theta^{(3)}, \ldots$
- Now the gradient for θ becomes the sum of the gradients of all its copies: $\boxed{\nabla_\theta L_t = \sum_{i=1}^{t} \nabla_{\theta^{(i)}} L_t}$

$$\vec{y}_3 = f(W_{out}\sigma(W_{in}^{(3)}\vec{x}_3 + W_h^{(3)}\sigma(W_{in}^{(2)}\vec{x}_2 + W_h^{(2)}\sigma(W_{in}^{(1)}\vec{x}_1 + W_h^{(1)}\vec{h}_0 + \vec{b}_h^{(1)}) + \vec{b}_h^{(2)}) + \vec{b}_h^{(3)}) + \vec{b}_{out})$$



Backpropagation Through Time

- Luckily we don't have to implement any of that, because of TensorFlow.
- But using this simple approach is usually not feasible.

Unstable Gradients

- Consider a sequence of 10.000 datapoints.
- Unrolling the corresponding RNN gives you essentially a network with 10.000 layers.
- We already know that training such deep networks does not work because of phenomena as vanishing/exploding gradients.

Truncated BPTT

- The solution is called Truncated Backpropagation Through Time (TBPTT).

- It is not necessary to do the updates for the whole sequence at once.

- Instead we can only compute for a certain bounded past.

- Also we don't have to update every step.

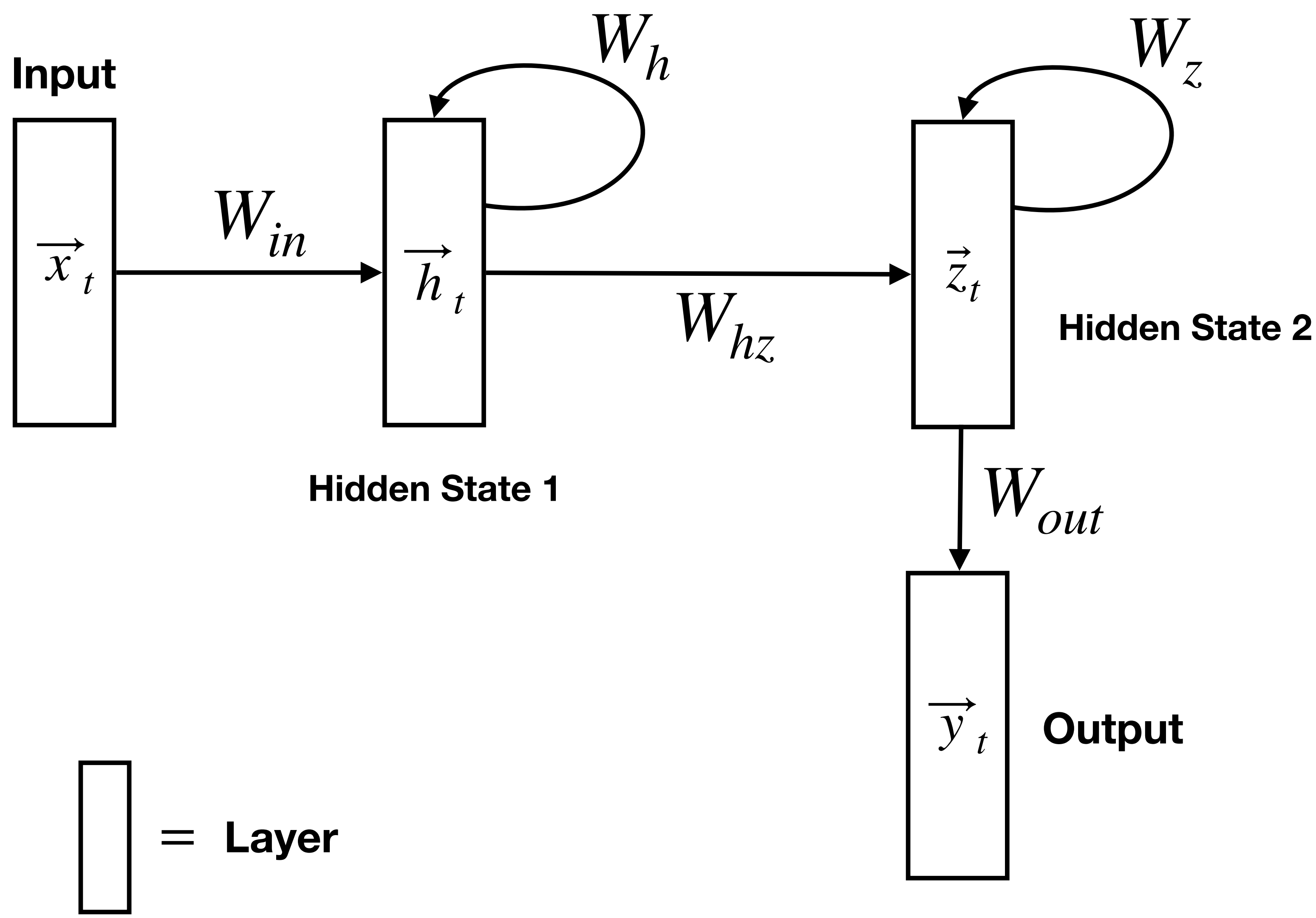
- This gives us the algorithm TBPTT(k_1, k_2) , with k_1 defining after how many steps we apply BPTT and k_2 defining for how many steps in the past we apply it.

Truncated BPTT

- TBPTT($1,n$): classical BPTT - applied each step for all time steps seen so far
- TBPTT(n, n): classical BPTT in the case that there is only on label for the whole sequence
- TBPTT(k_1, k_2), $k_1 = k_2 < n$: common version of TBPTT in which the sequence is basically chunked in chunks that are treated independently (except for the hidden state init)
- TBPTT(k_1, k_2), $k_1 < k_2 < n$: each timestep is involved in multiple updates, can be more efficient

Stacked RNNs

- Stacked RNNs allow to predict more complex behavior:



Example

Not available due to copyright reasons.

The Unreasonable Effectiveness of Recurrent Neural Networks (Andrej Karpathy)

Long-Term Dependencies

- The problem with the solution of TBPTT is that the RNN can't learn long-term dependencies.

Example

[…] From age 3 until the age of 9 I lived in France. My mother's parents are from there and therefore we moved there in 1999. I have three brothers. I am the youngest child. […] Lastly I am multilingual, I am fluent in English, German and …???

Long-Term Dependencies

- The problem with the solution of TBPTT is that the RNN can't learn long-term dependencies.

Example

Which one depends on a word several sentences before.

[…] From age 3 until the age of 9 I lived in France. My mother's parents are from there and therefore we moved there in 1999. I have three brothers. I am the youngest child. […] Lastly I am multilingual, I am fluent in English, German and …???

Local context suggests a language!

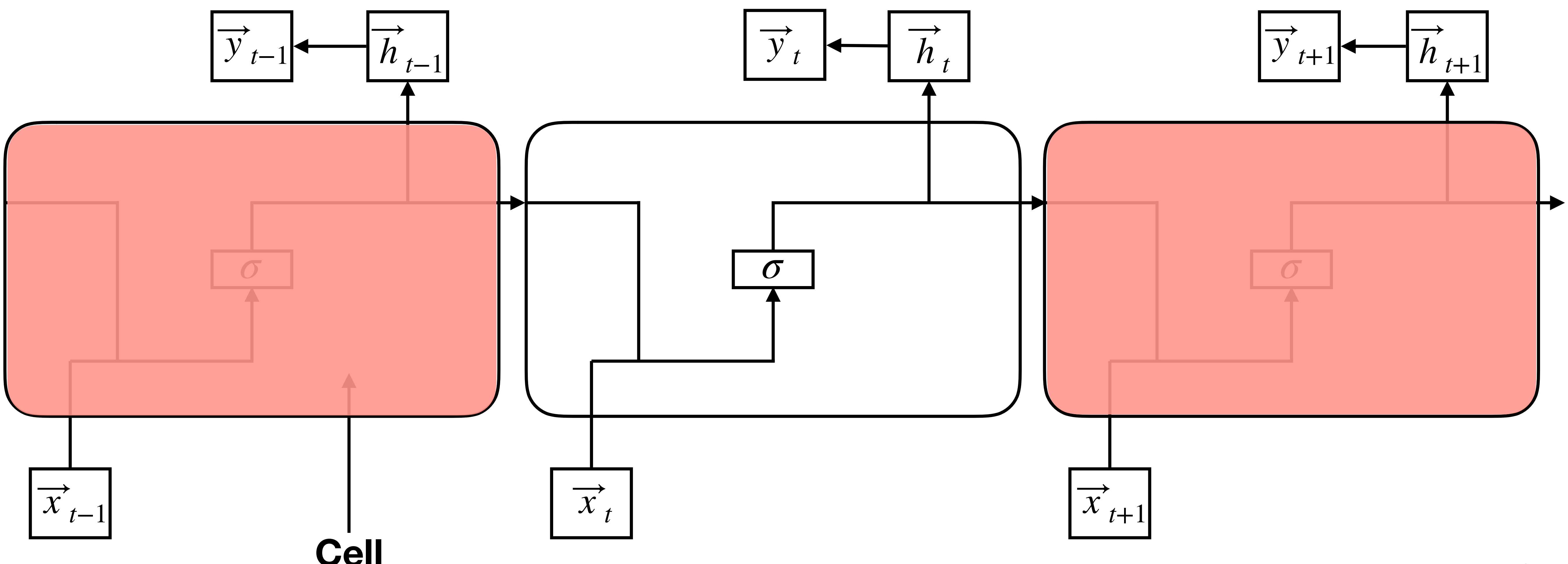
LSTMs

Long Short-Term Memory

- The solution to this problem is called Long Short-Term Memory (LSTM, german: Langes Kurzzeitgedächtnis).
- Developed by Hochreiter & Schmidhuber (1997).
- LSTMs have the power to remember information over long periods of steps.
- The content of the following pages is strongly inspired by the great blogpost Understanding LSTM Networks (Chris Olah)

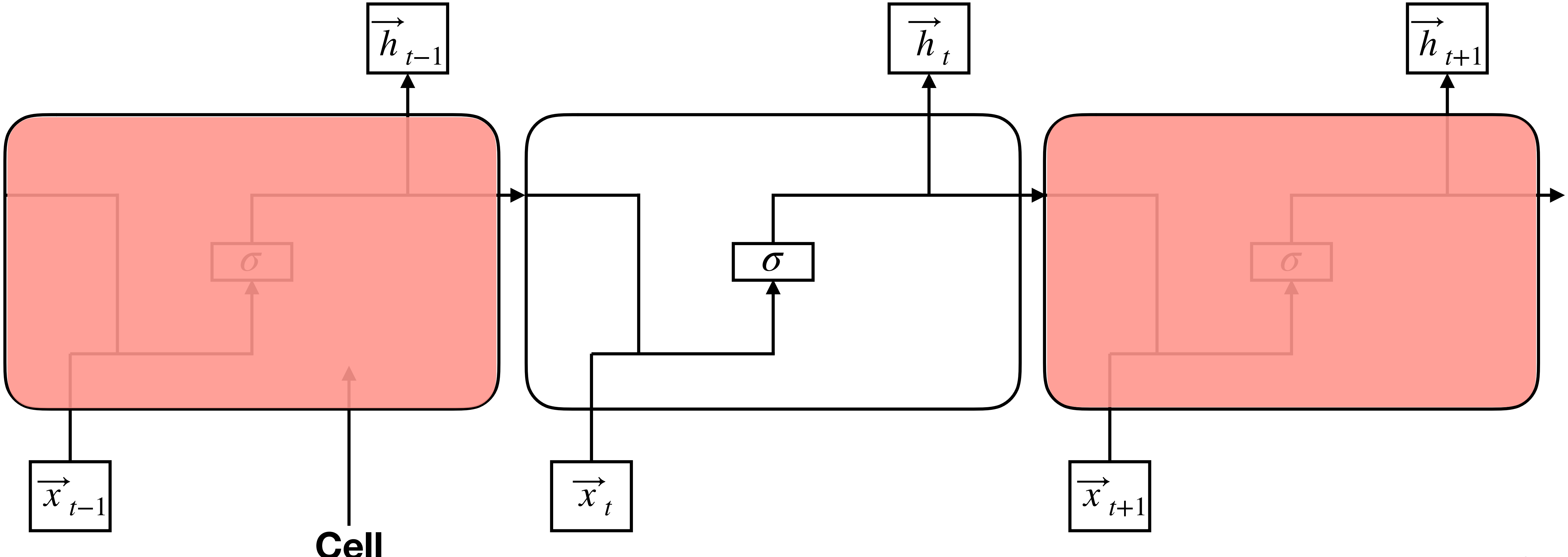
RNN Cell

- An unfolded recurrent network can be visualized as a repeating cell in which certain computations happen.
- This is the vanilla RNN.
- For visualization we can also leave the outputs, as they are independent from the recurrent system.

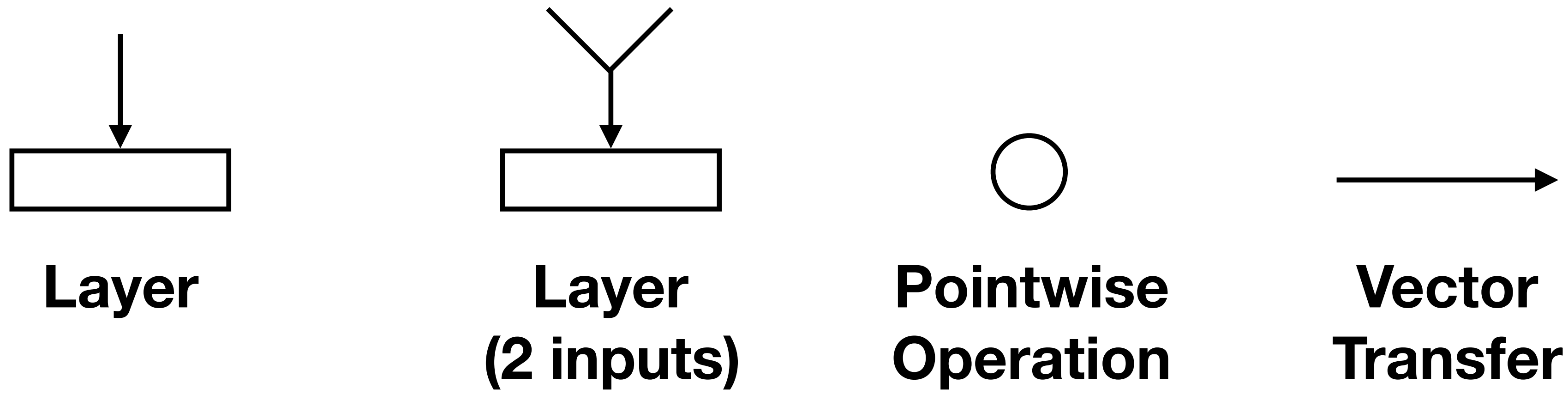


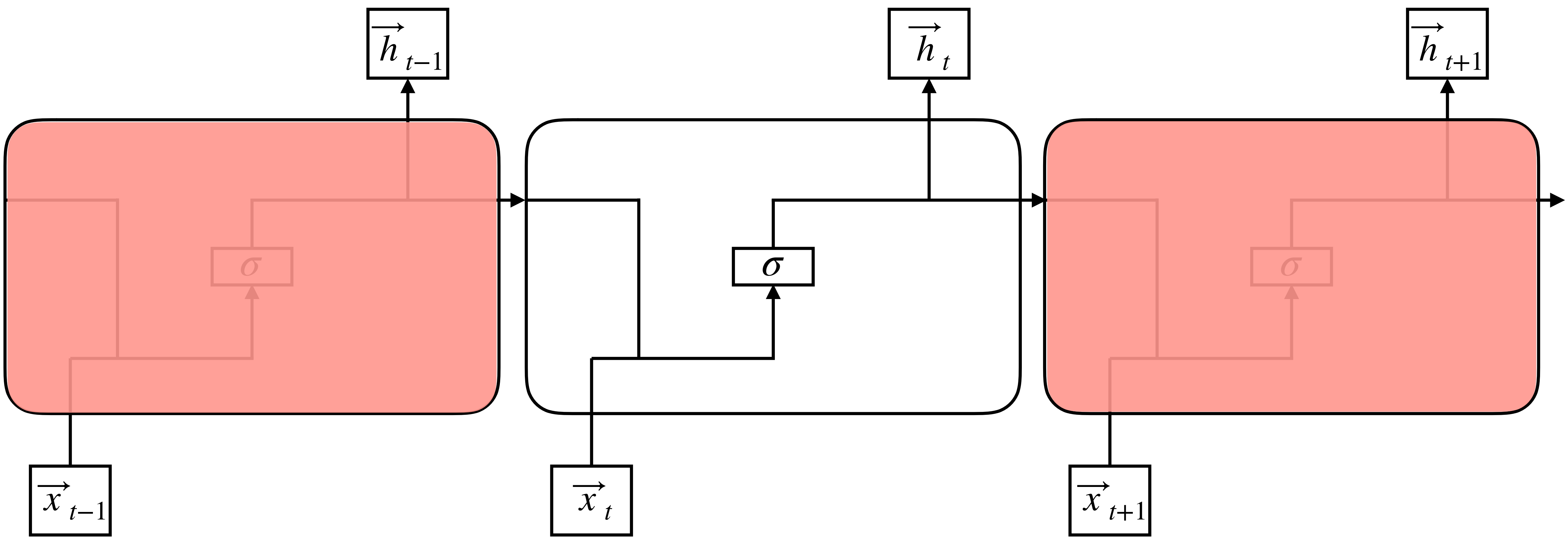
RNN Cell

- An unfolded recurrent network can be visualized as a repeating cell in which certain computations happen.
- This is the vanilla RNN.
- For visualization we can also leave the outputs, as they are independent from the recurrent system.



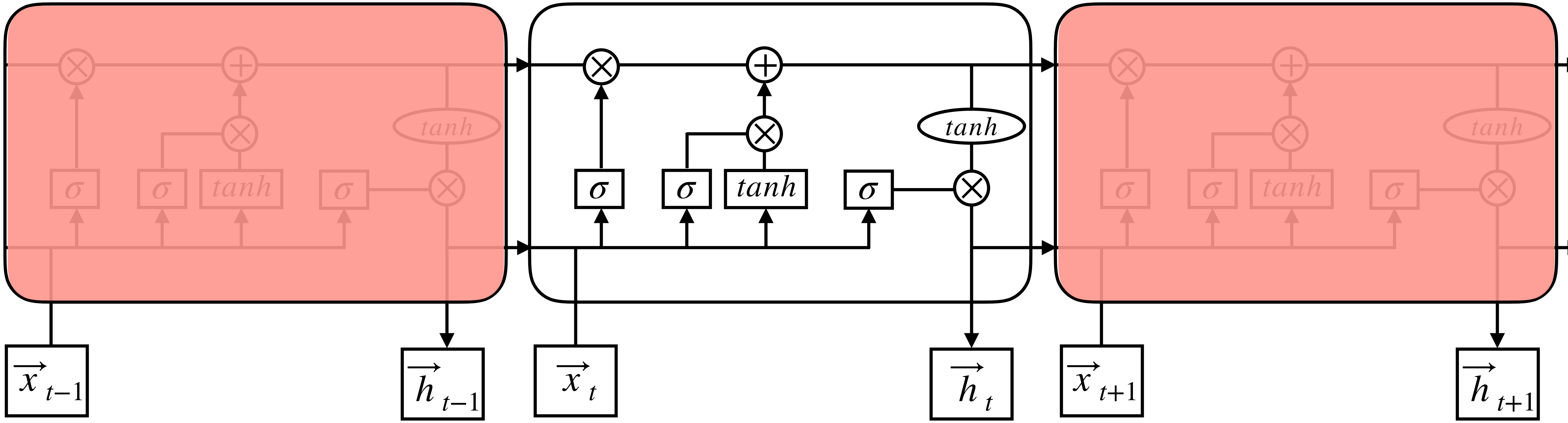
Notation





$$\vec{h}_t = \sigma(W_{in}\vec{x}_t + W_h\vec{h}_{t-1} + \vec{b}_h)$$

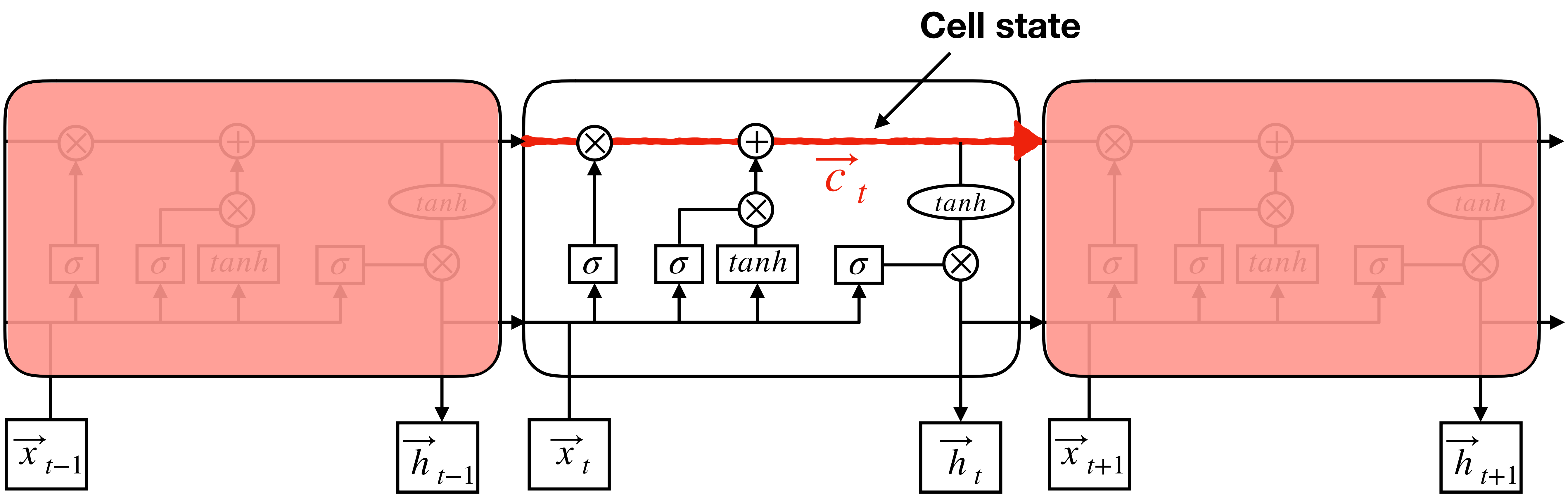
LSTM Cell



Don't worry! We will go through it step for step!

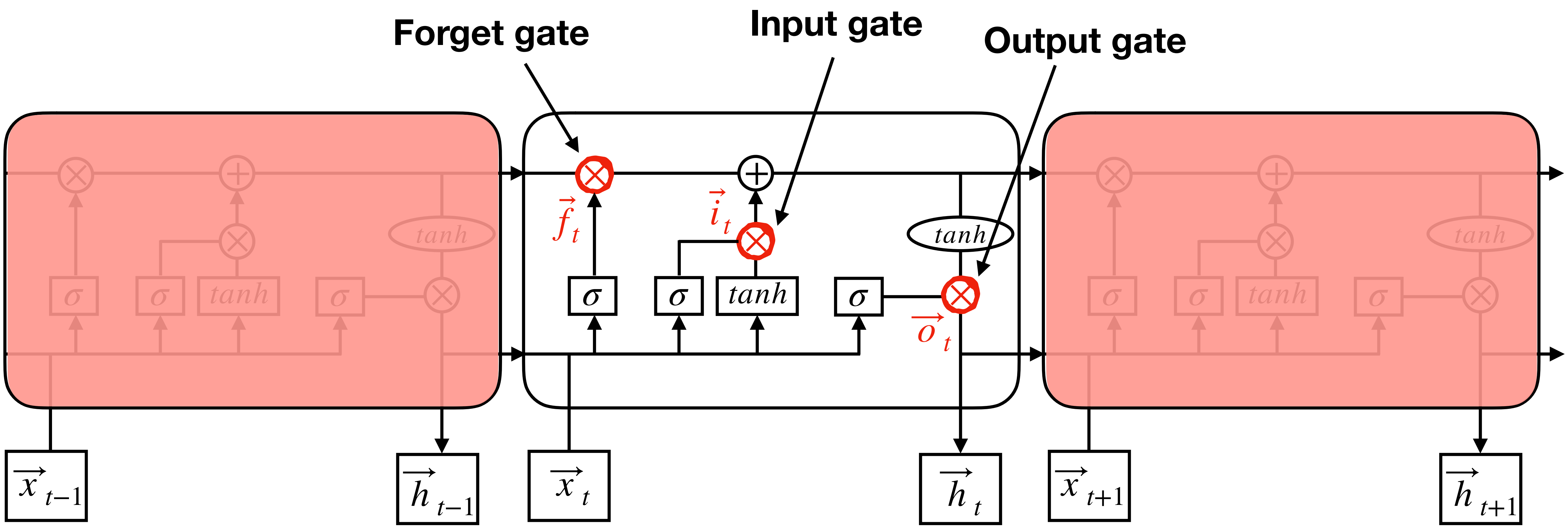
Cell State

- The <u>cell state</u> is the major enhancement of the LSTM. Similar to the <u>hidden state</u> it is passed on with every time step.
- The cell state is only modulated through simples operations, thus information can flow easily.



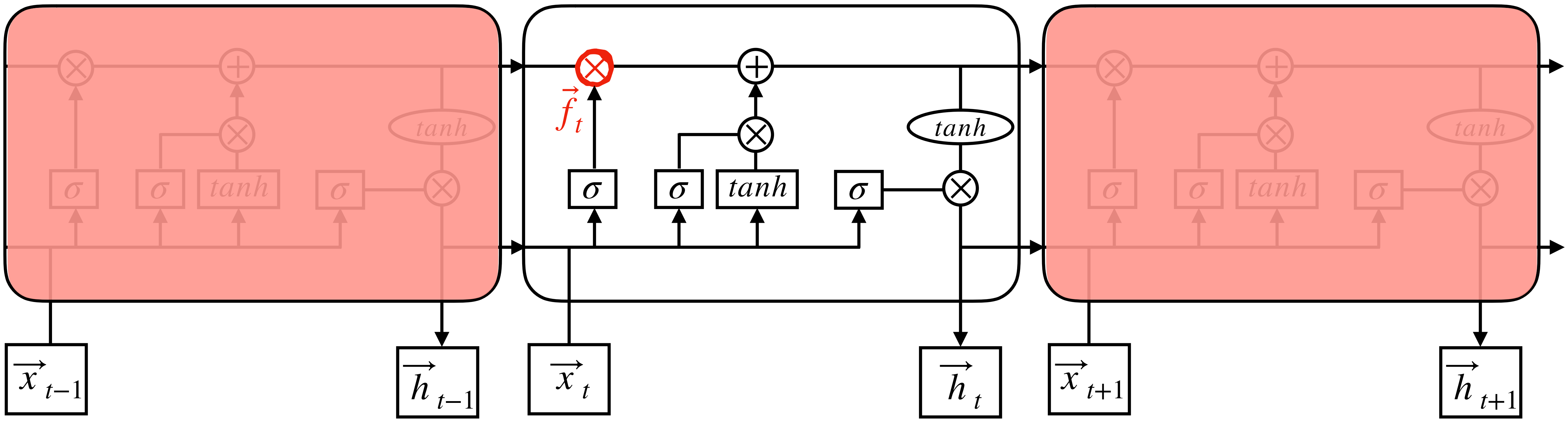
Gates

- The LSTM has 3 gates: the forget gate, the input gate and the output gate.
- First each gate has a sigmoidal layer taking the current input and the last hidden state as an input.
- The output of this layer is then componentwise multiplied.
- Because of sigmoid activation function each component is between (0=stop) and (1=go).



- The forget gate regulates, which information of the of the old cell state should be kept.

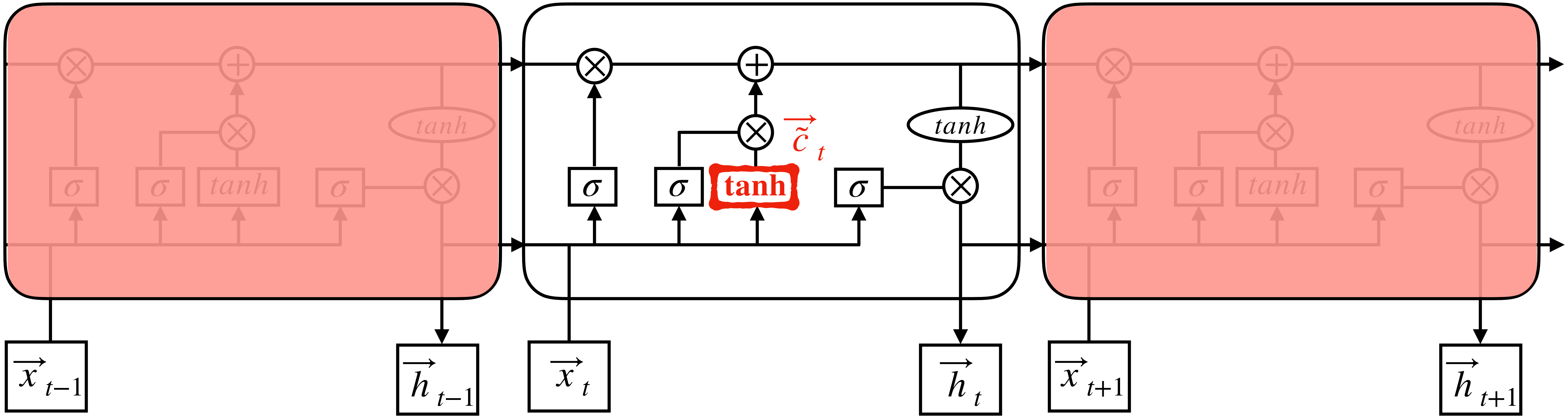
$$\vec{f}_t = \sigma(W_{fx}\vec{x}_t + W_{fh}\vec{h}_{t-1} + \vec{b}_f)$$



New Candidate for Cell State

- After forgetting we need new information for the cell state.
- First a new candidate is generated.

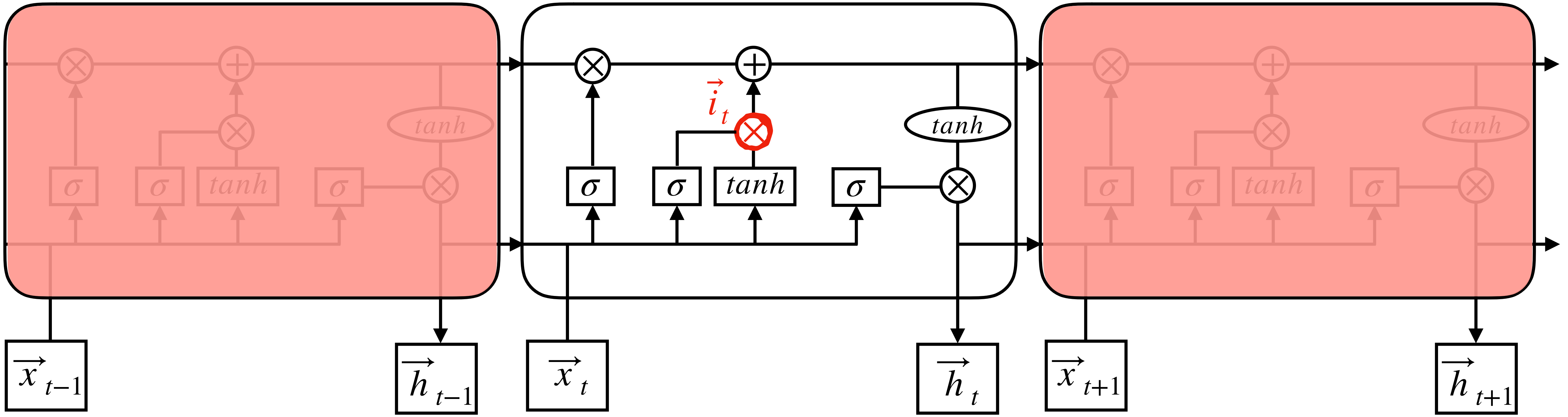
$$\vec{\tilde{c}}_t = tanh(W_{cx}\vec{x}_t + W_{ch}\vec{h}_{t-1} + \vec{b}_c)$$



Input Gate

- But before the new candidate is included into the cell state the input gate processes the new candidate.

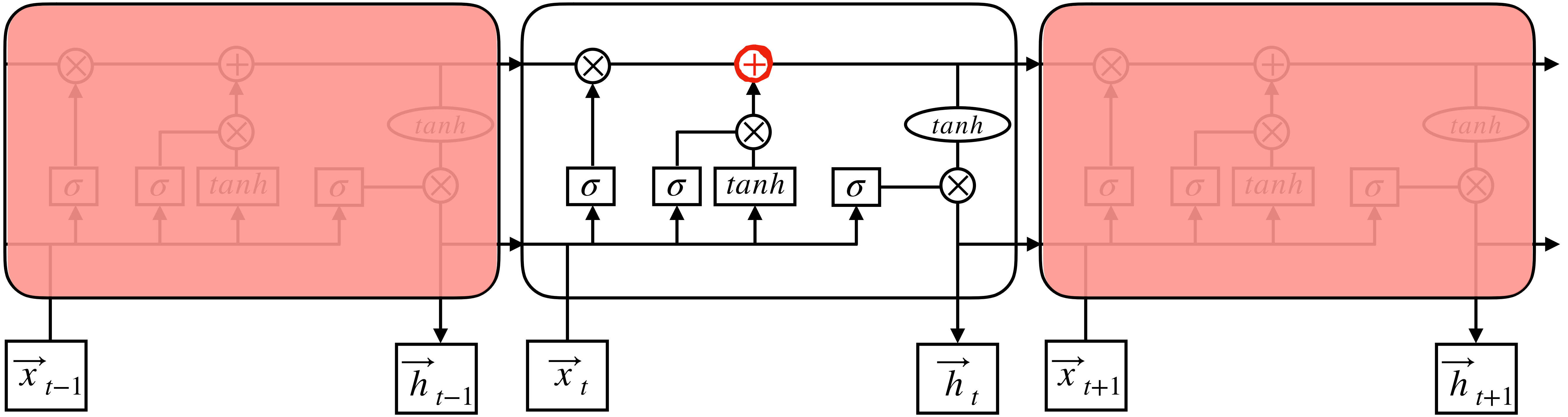
$$\vec{i}_t = \sigma(W_{ix}\vec{x}_t + W_{ih}\vec{h}_{t-1} + \vec{b}_i)$$



Update Cell State

- Now the input gate and the forget gate can update the cell state

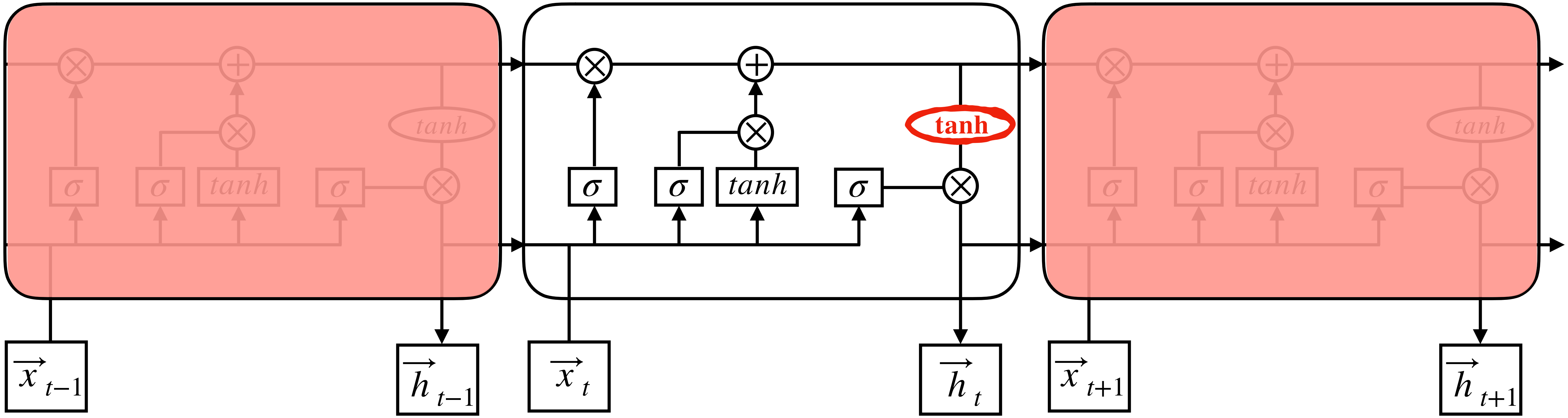
$$\overrightarrow{c}_t = \vec{f}_t * \overrightarrow{c}_{t-1} + \vec{i}_t * \overrightarrow{\tilde{c}}_t$$



New Candidate for Hidden State

- The new cell state is then used to generate a new candidate for the hidden state.

$$\vec{\tilde{h}}_t = tanh(\vec{c}_t)$$

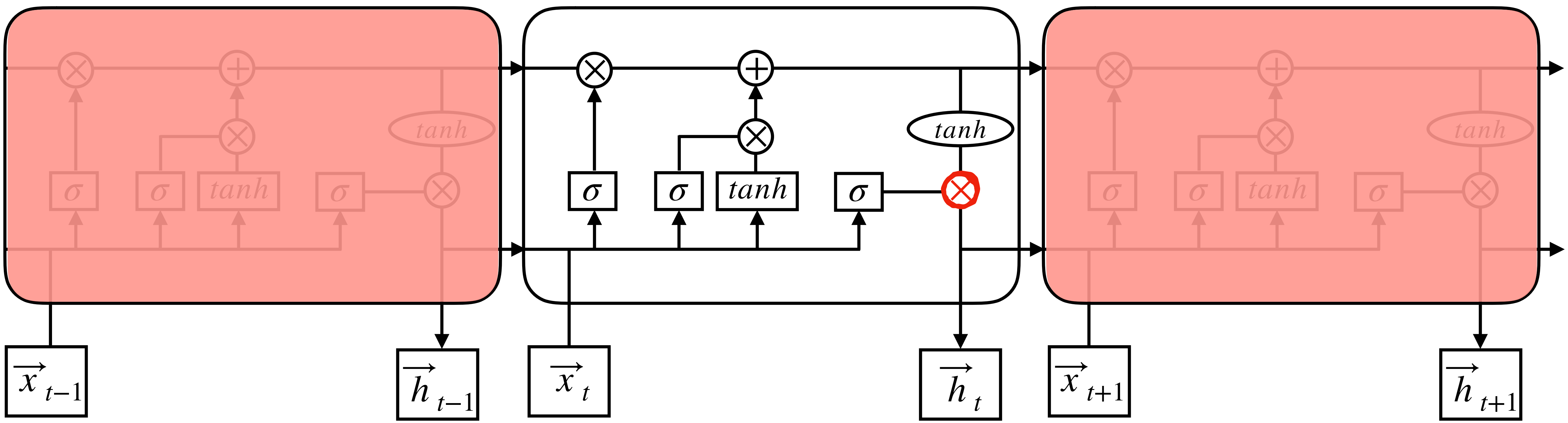


Output Gate

- **Lastly the output gate modulates this candidate for the new hidden state.**

$$\vec{o}_t = \sigma(W_{ox}\vec{x}_t + W_{oh}\vec{h}_{t-1} + \vec{b}_o)$$

$$\vec{h}_t = \vec{o}_t * \vec{\tilde{h}}_t$$



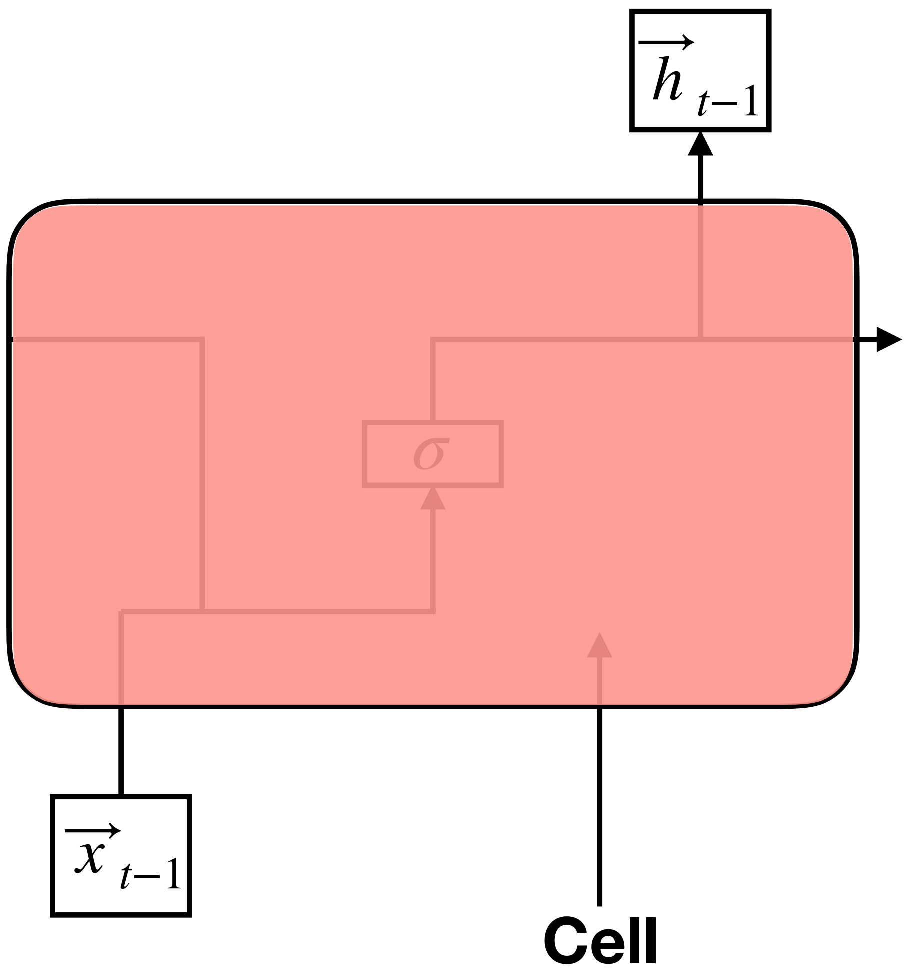
RNNs in TensorFlow

RNNs in TensorFlow

- Defining an RNN in TensorFlow happens in two steps:
  - First you define the cell (e.g. vanilla, LSTM or whatever you want).
  - Then you define the encapsulating RNN, i.e. the model that actually runs through a sequence.

Cell

The cell defines what happens in one time step.



Example in TF

- An RNN cell is like a normal layer, but there are some things required!

```
class VanillaRNNCell(tf.keras.layers.Layer):

    def __init__(self, input_dim, units):
        super(VanillaRNNCell, self).__init__()
        self.input_dim = input_dim
        self.units = units
        # TF needs this.
        self.state_size = units

    def build(self, input_shape):
        self.w_in = self.add_weight(
                                    shape=(self.input_dim, self.units),
                                    initializer='uniform'
                                    )
        self.w_h = self.add_weight(
                                    shape=(self.units, self.units),
                                    initializer='uniform'
                                    )
        self.b_h = self.add_weight(
                                    shape=(self.units,),
                                    initializer='zeros'
                                    )

    def call(self, inputs, hidden_states):
        h_prev = hidden_states[0]
        h_new = tf.nn.sigmoid(tf.matmul(inputs, self.w_in) + tf.matmul(h_prev, self.w_h) + self.b_h)
        return h_new, [h_new]
```

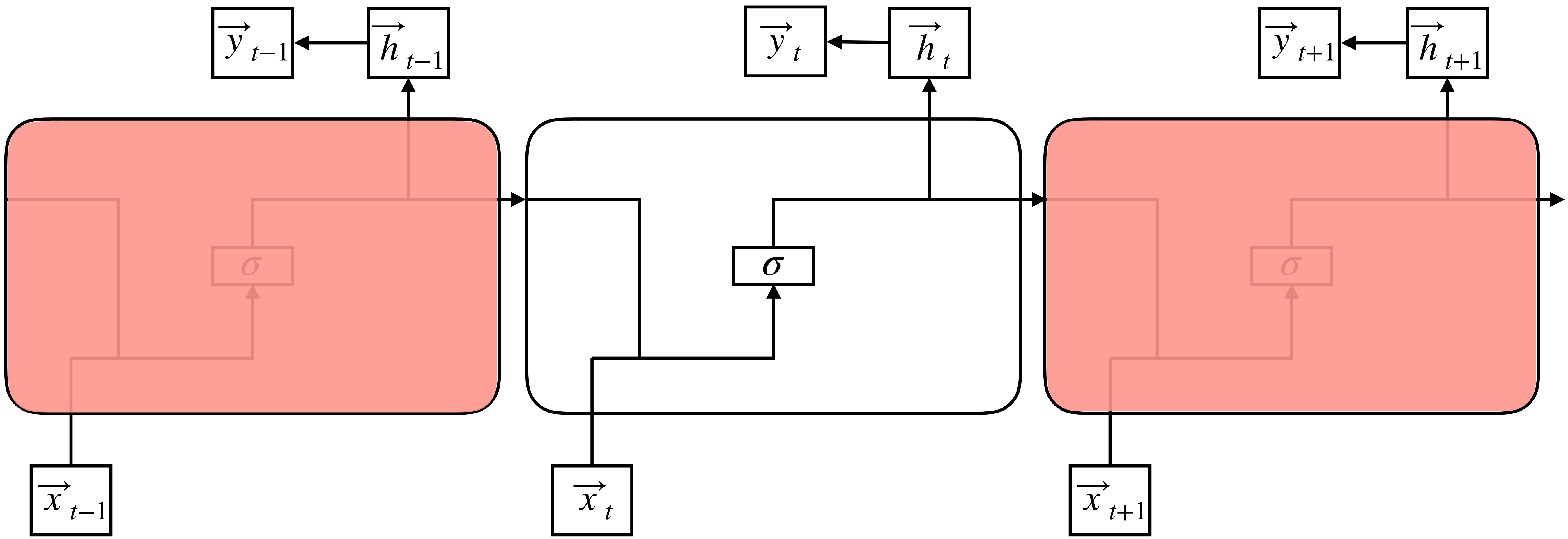
needs this parameter self.state_size

call method takes in current input and previous hidden state

returns hidden state + hidden state in a list

RNN

The RNN defines how the sequential application of the previously defined cell.



Example in TF

- You can define a second class that encapsulates the RNN including the output layer.

```
class RNN(tf.keras.layers.Layer):

    def __init__(self):
        super(RNN, self).__init__()
        self.cell = VanillaRNNCell(input_dim=80, units=256)
        self.rnn = tf.keras.layers.RNN(self.cell, return_sequences=True)
        self.output_layer = tf.keras.layers.Dense(units=10, activation=tf.nn.softmax)

    def call(self,x):
        seqs = self.rnn(x)
        output = self.output_layer(seqs)
        return output
```

This layer is the actual RNN model.

If this is true the RNN returns all hidden states of the whole sequence.

Defining the output computations (either for all sequence steps or just for last output).

Docs

- There are different pre-defined cells in TensorFlow:
    - SimpleRNNCell, GRUCell, LSTMCell
    - RNN

Applications

Applications

- Image to Caption (https://www.captionbot.ai)
- Caption to Image (https://arxiv.org/abs/1511.02793)
- Translation
- Speech to Text (speech processing)
- Text to Speech (speech synthesis)

Although there are other models (e.g. WaveNet or Transformers), which are better for certain tasks.

Questions?

See you next week!