

UNIVERSITY OF
TORONTO

Applied Deep Learning

Lecture 8. *Recurrent NNs and Gated Architectures*

Ali Bereyhi
ECE Department, University of Toronto
Fall 2026



UNIVERSITY OF
TORONTO

Lecture Overview

UNIVERSITY OF TORONTO

Lecture 8. *Recurrent NNs and Gated Architectures*

Today, we learn how to process sequence data with NNs

1. Recurrent NNs
   - *Building memory component using recursion*
   - *Backpropagation through time*
2. Gated architectures
   - *Principle of gating*
   - *Gated models: GRU and LSTM*



UNIVERSITY OF
TORONTO

Sequence Data and Recurrent NNs

Learning from Sequence Data

In many applications, we have *sequence data*, *e.g.*,

- *speech data which is usually a long time series*
- *text data which is sequence of words and letters*
- *financial data that is typically a time-dependent sequence of values*

Learning from such data can be inefficient via FNNs

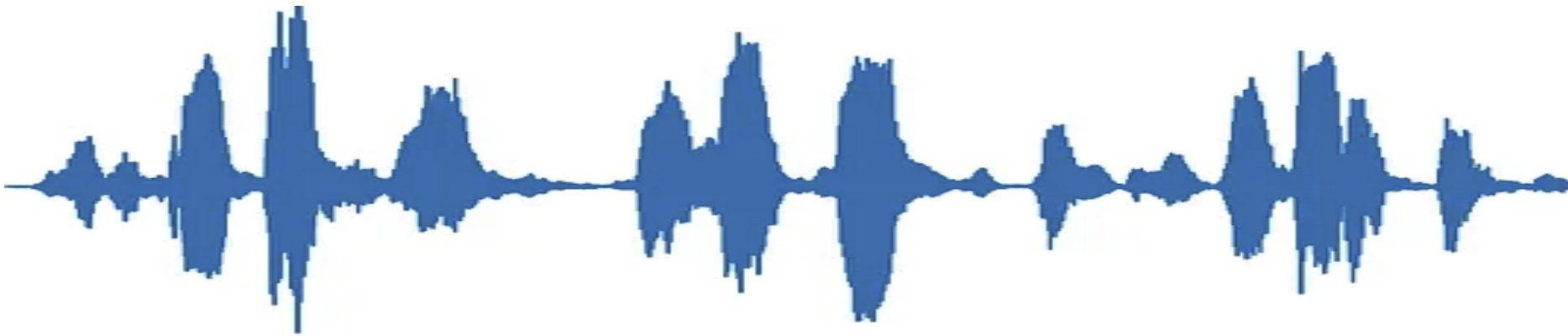
✖ *We have long sequence*

❗ *We do not have so much features*

Learning from Sequence Data: *Example I*

We listen to a 15-minute talk: we want to classify the topic, e.g., sport or science

- *This is a classification problem*
- *What about the data?*
  - ↳ *Sample audio signal at rate 44.1 kHz and quantize*
  - ↳ *Put N successive samples in a frames $\equiv$ vector of samples*



$\mathbf{x}[1]$ $\mathbf{x}[2]$ $\mathbf{x}[3]$ $\cdots$ $\mathbf{x}[t]$ $\mathbf{x}[t+1]$ $\cdots$

With frames of 512 samples, we have

77,587 frames = 39,690,000 samples!

Motivational Example: *Predicting Next Word*

We have a long text and want to learn what is the next word

- *This is again classification*
- *What about the data?*
  - *We parse the text into a sequence of tokens, e.g., letters*
  - *We represent each token with a vector*

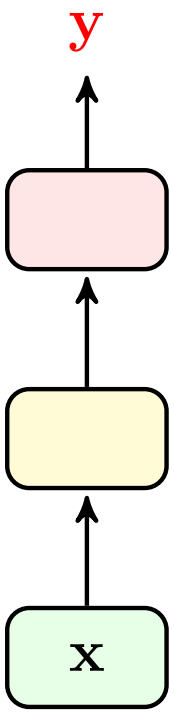
$$\mathbf{x}[t-6] \quad \mathbf{x}[t-5] \quad \mathbf{x}[t-4] \quad \mathbf{x}[t-3] \quad \mathbf{x}[t-2] \quad \mathbf{x}[t-1] \quad \mathbf{x}[t] \quad \mathbf{y} = \mathbf{x}[t+1]$$

```
... Julia has been nominated to receive Alexander von Humboldt Prize for her
```

- *How does the training dataset look like?*
- We are given with several long texts in the same context

Predicting Next Word: *MLP*

Let's try solving this problem with a simple MLP



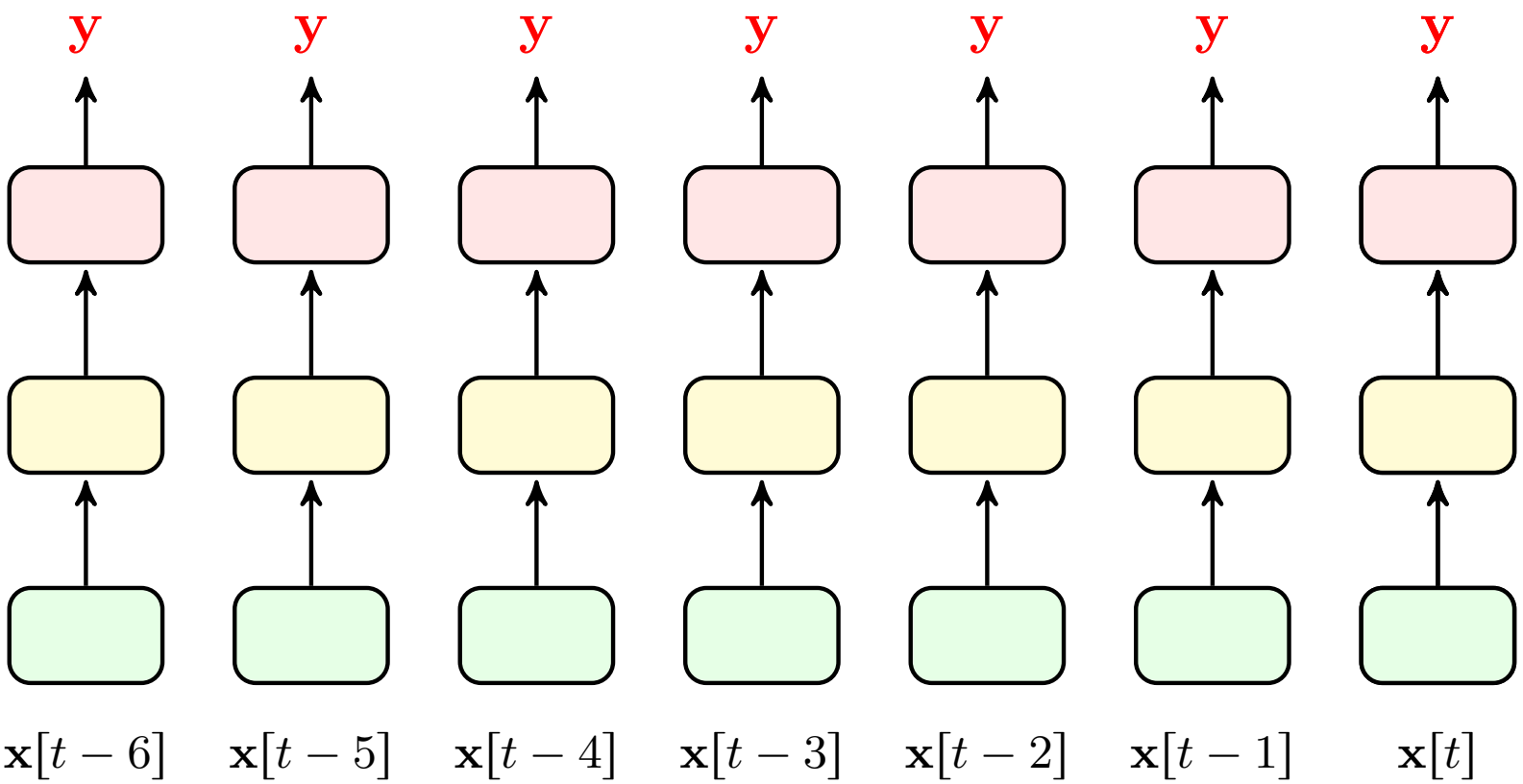
We have an MLP that

- *takes N inputs $\equiv$ a token*
- *it has multiple hidden layers $\equiv$ yellow box*
- *it has an output layer (head) $\equiv$ red box*
- *returns N outputs $\equiv$ predicted next token*

We train this MLP

- *we go over all text*

Predicting Next Word: *MLP*



... Julia has been nominated to receive Alexander von Humboldt Prize for her

Does MLP predict "her"? No! How can it remember we are talking about Julia?!

Predicting Next Word: *MLP*

$$\mathbf{x}[t-6] \quad \mathbf{x}[t-5] \quad \mathbf{x}[t-4] \quad \mathbf{x}[t-3] \quad \mathbf{x}[t-2] \quad \mathbf{x}[t-1] \quad \mathbf{x}[t] \quad \mathbf{x}[t+1]$$

`... Julia has been nominated to receive Alexander von Humboldt Prize for her`

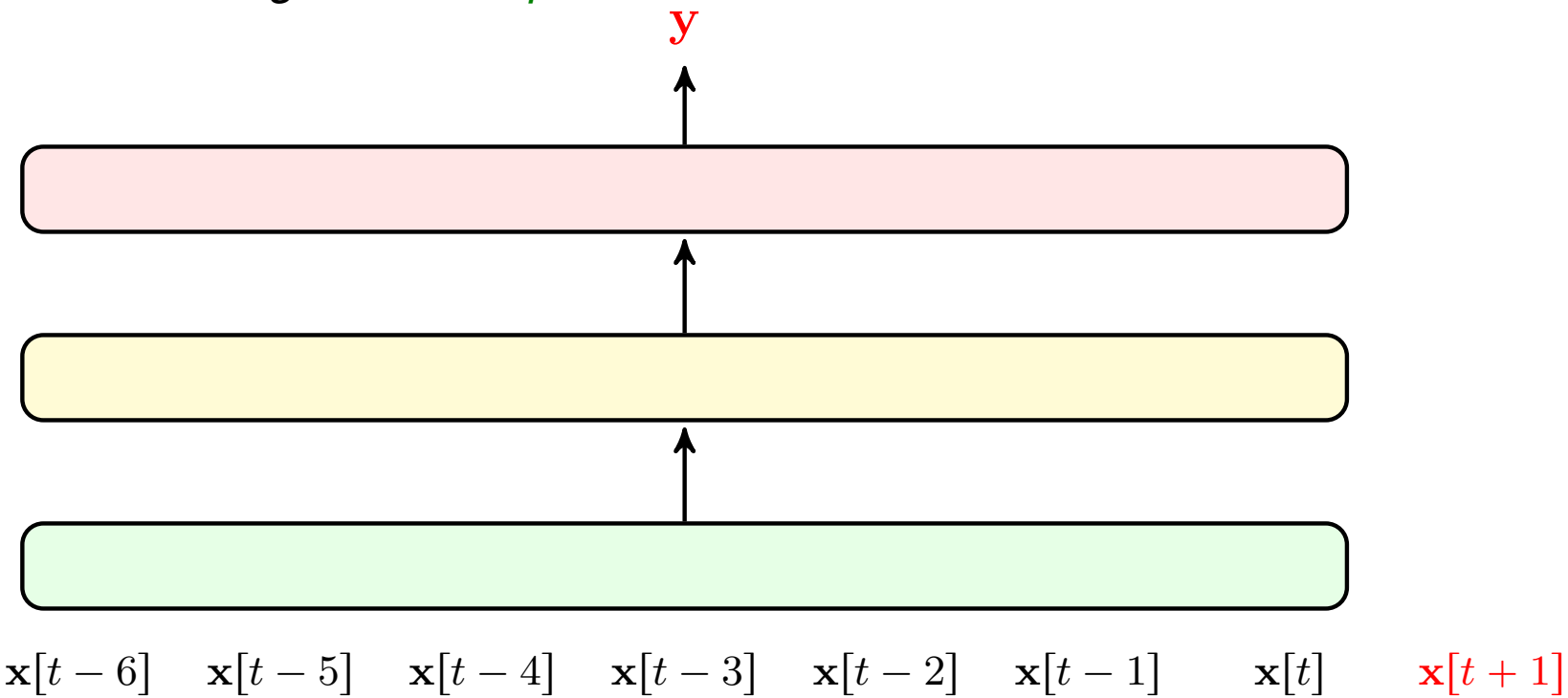
Each time MLP gets trained with a new sample, *it forgets previous text*

- *many of learned predictions are irrelevant, e.g.,*
  - ! "`nominated to`" *is followed by* "`receive`": *has nothing to say about* "`her`"
- *at time t, MLP gets no input that connects it to Julia*

This indicates that we need *a memory component in our model!*

Predicting Next Word: *Large MLP*

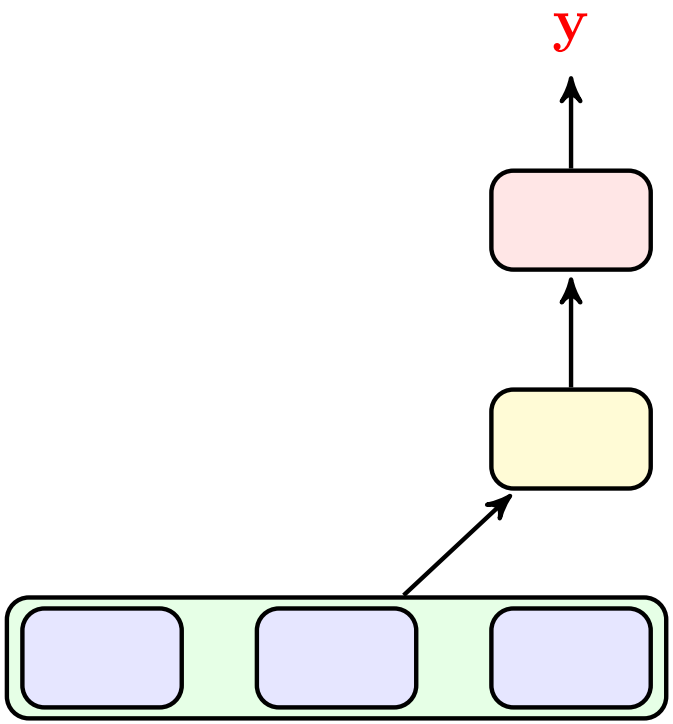
Maybe we could give more inputs to the FNN!



But, what if Julia has been mentioned 10 pages ago? Forget about large MLPs!

Predicting Next Word: *CNN*

Let's now think about CNNs



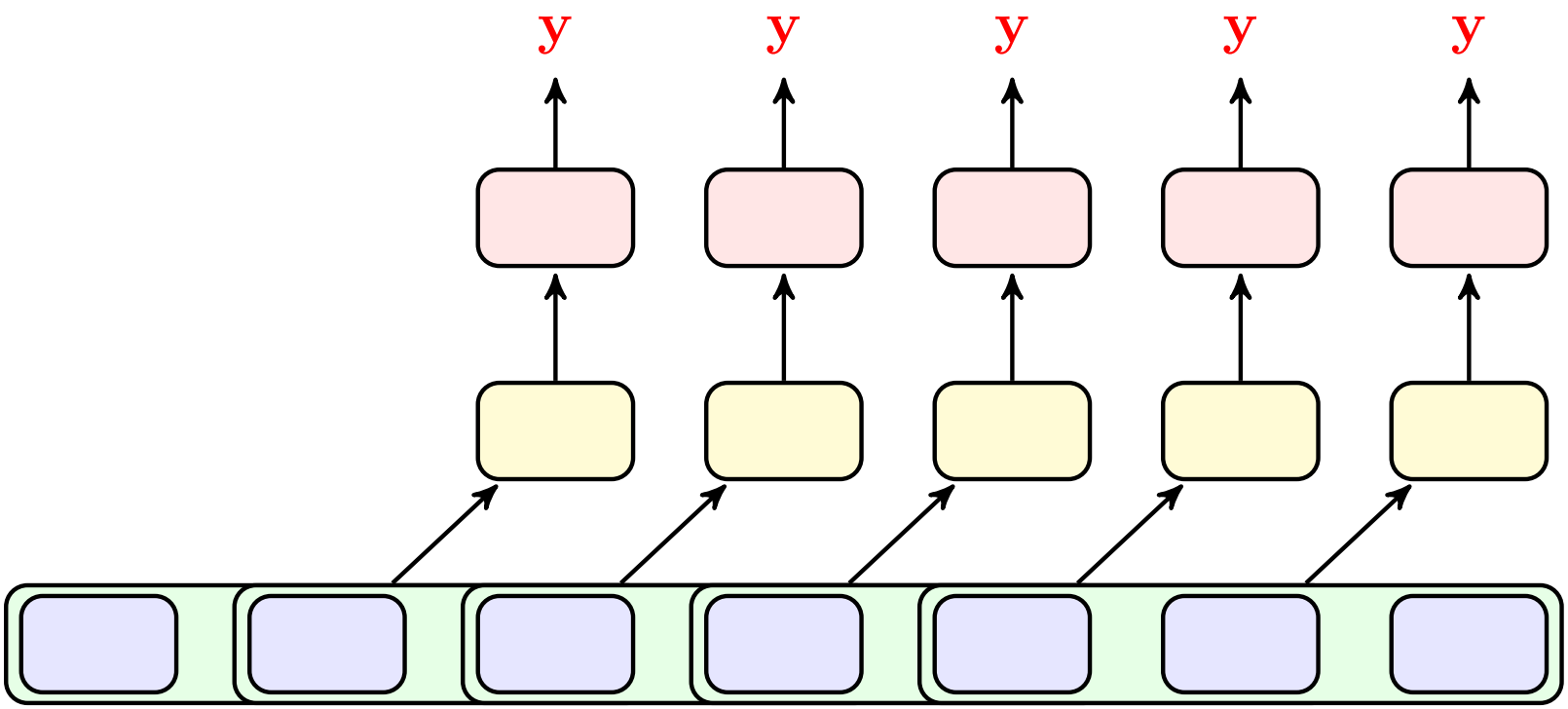
We add a convolutional layer at input

- *it has filter with L channels*
- *it does 1D convolution over last L tokens*

We use convolution to

- *extract features from a larger part of text*
- *pass it to hidden layers*

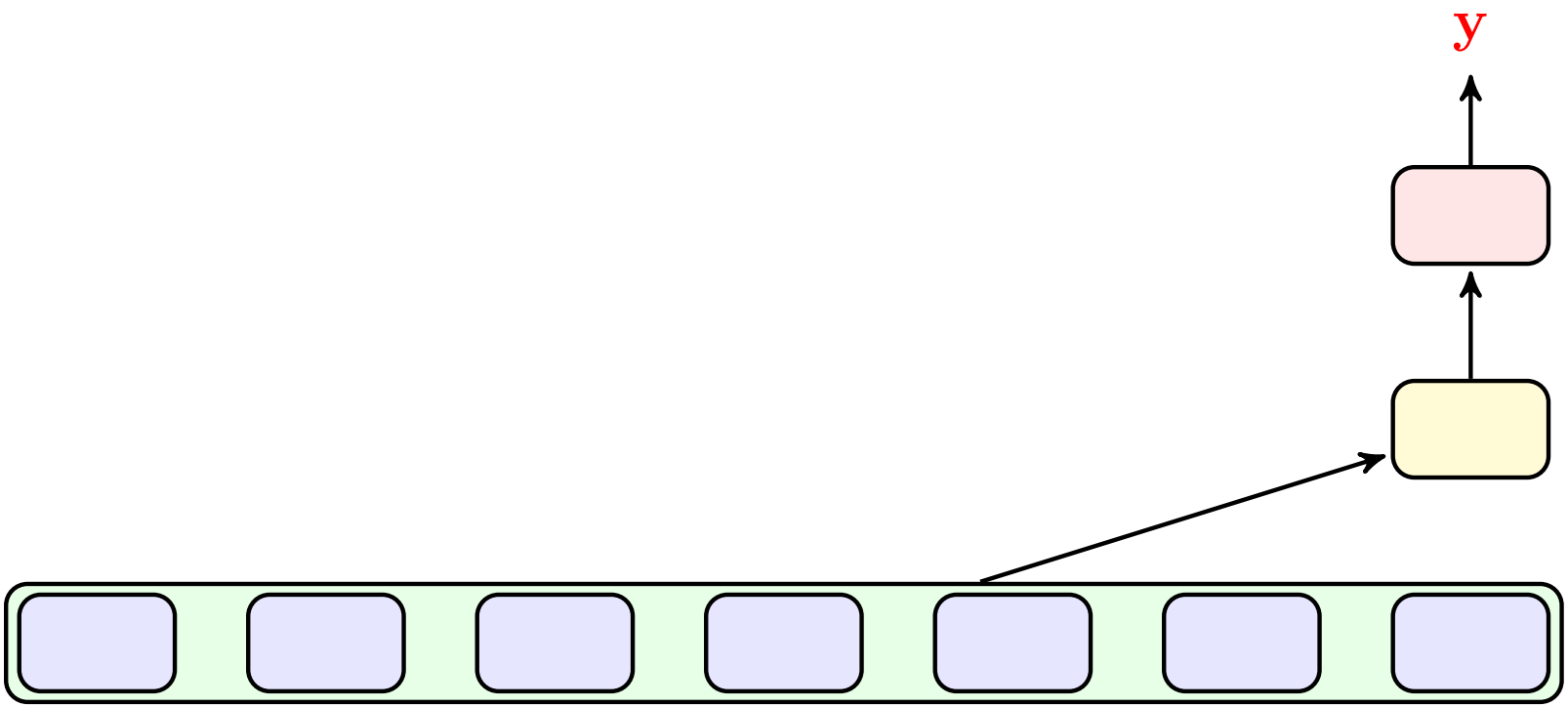
Predicting Next Word: *CNN*




```
... Julia has been nominated to receive Alexander von Humboldt Prize for her
```

Yet, it doesn't seem to remember Julia!

Predicting Next Word: *Large CNN*



Unless we slide over the whole text! Forget about it!

Finite Memory: *Root Problem*

$$\mathbf{x}[t-6] \quad \mathbf{x}[t-5] \quad \mathbf{x}[t-4] \quad \mathbf{x}[t-3] \quad \mathbf{x}[t-2] \quad \mathbf{x}[t-1] \quad \mathbf{x}[t] \quad \mathbf{x}[t+1]$$

```
... Julia has been nominated to receive Alexander von Humboldt Prize for her
```

The problem with *all architectures we know* is that *they have finite memory*

- ✖ *They can only remember from their input*
- ! *If we need to remember for long time we have to give them huge inputs*
- 💡 *But the memory component does not seem to be so huge*
  - ↳ *We may only remember that Julia is a "single person" and "female"*
  - ↳ *If text now switches to Theodore we should refresh our memory*

Finite Memory Component *with Infinite Response*

Component We Miss

We need to extract a memory component from our data that is finite in size but influenced infinitely (at least theoretically)

State-space model can help us building such memory component!

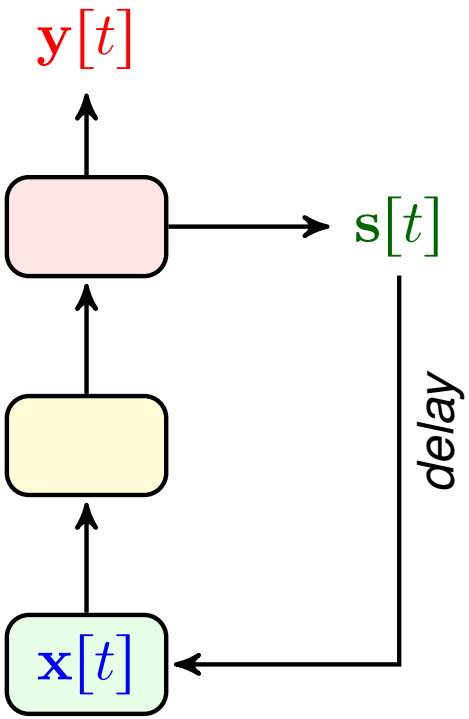
State-space Model

Assume $\mathbf{x}[t]$ *is an input to a system at time* t*: the system returns an output* $\mathbf{y}[t]$ *to this input and a state variable* $\mathbf{s}[t]$*. The output in the next time, i.e.,* $t+1$*, depends on the new input and current state, i.e.,*

$$\mathbf{y}[t+1], \mathbf{s}[t+1] = f\left(\mathbf{x}[t+1], \mathbf{s}[t]\right)$$

State-Space Model with *NNs*

We can make a state-space model via NNs



In this architecture, the NN returns

- $\mathbf{y}[t]$ *and state* $\mathbf{s}[t]$

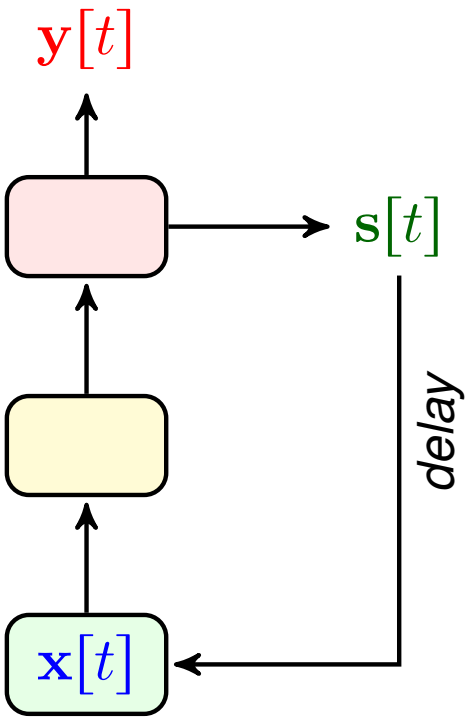
Next time, it takes

- $\mathbf{x}[t]$ *and previous state* $\mathbf{s}[t-1]$

This NN captures temporal behavior of data!

State-Space Model for *NNs*

Theoretically, this NN has infinite time response



Say it takes only sample at $t = 1$

- $\mathbf{y}[t]$ *depends on* $\mathbf{s}[t-1]$
- $\mathbf{s}[t-1]$ *depends on* $\mathbf{s}[t-2]$

  $\ldots$
- $\mathbf{s}[1]$ *depends on* $\mathbf{x}[1]$

This means that $\mathbf{y}[t]$ *still remembers* $\mathbf{x}[1]$

Early State-Dependent NNs

Several attempts have been done: *we look briefly into two of them*

- *Jordan Network*
  - ↳ *proposed by Michael Jordan*[1] *in 1986*
  - ↳ *it computes the state variable to be a simple moving average*
- *Elman Network*
  - ↳ *proposed by Jeffrey Elman*[2] *in 1990*
  - ↳ *it learns the state variable but does not track back completely over time*

These models are simple forms of what we nowadays know as

$$\textit{Recurrent NNs} \equiv \textit{RNNs}$$

RNN

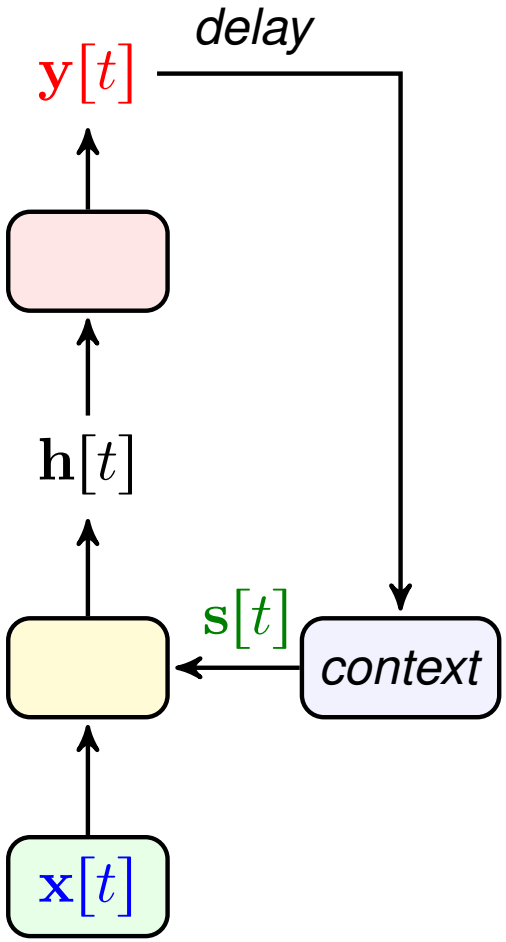
RNN is an NN with state-variable: the term recurrent refers to the connection between former state and new output

[1] Professor at CS Department of University of Berkeley; check his page

[2] Late Professor at UCSD ('48 – '18); check his page

Jordan Model

Jordan Network uses a simple moving average of output as the state



The proposal uses a shallow NN

- *It starts with initial state* $\mathbf{s}[1]$
- *It updates the state with fixed* μ *as*

$$\mathbf{s}[t] = \mu \mathbf{s}[t-1] + \mathbf{y}[t-1]$$

- *The hidden layer computes*

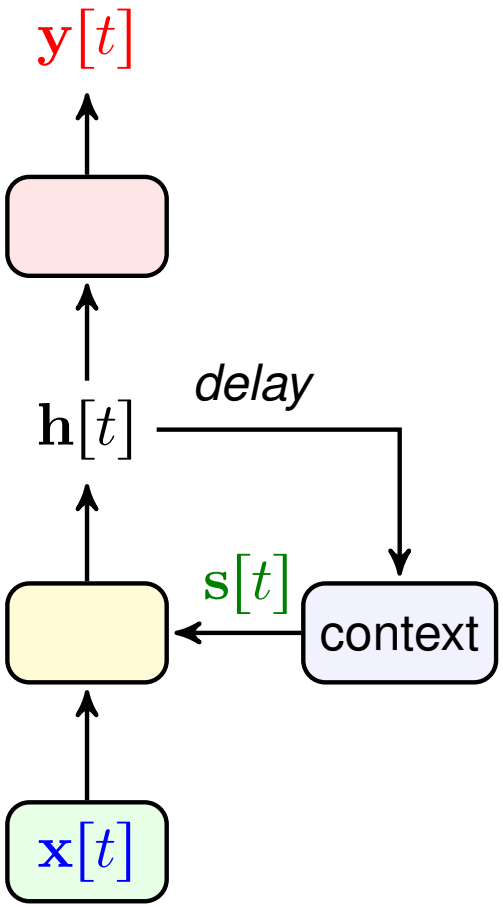
$$\mathbf{h}[t] = f\left(\mathbf{W}_1 \mathbf{x}[t] + \mathbf{W}_\mathrm{m} \mathbf{s}[t]\right)$$

- *The output is*

$$\mathbf{y}[t] = f\left(\mathbf{W}_2 \mathbf{h}[t]\right)$$

Elman Network

Elman Network uses output of hidden layer as state: also called hidden state



The proposal uses the same shallow NN

- *It starts with initial hidden state* $\mathbf{h}[0]$
- *It updates the state as*

$$\mathbf{s}[t] = \mathbf{h}[t-1]$$

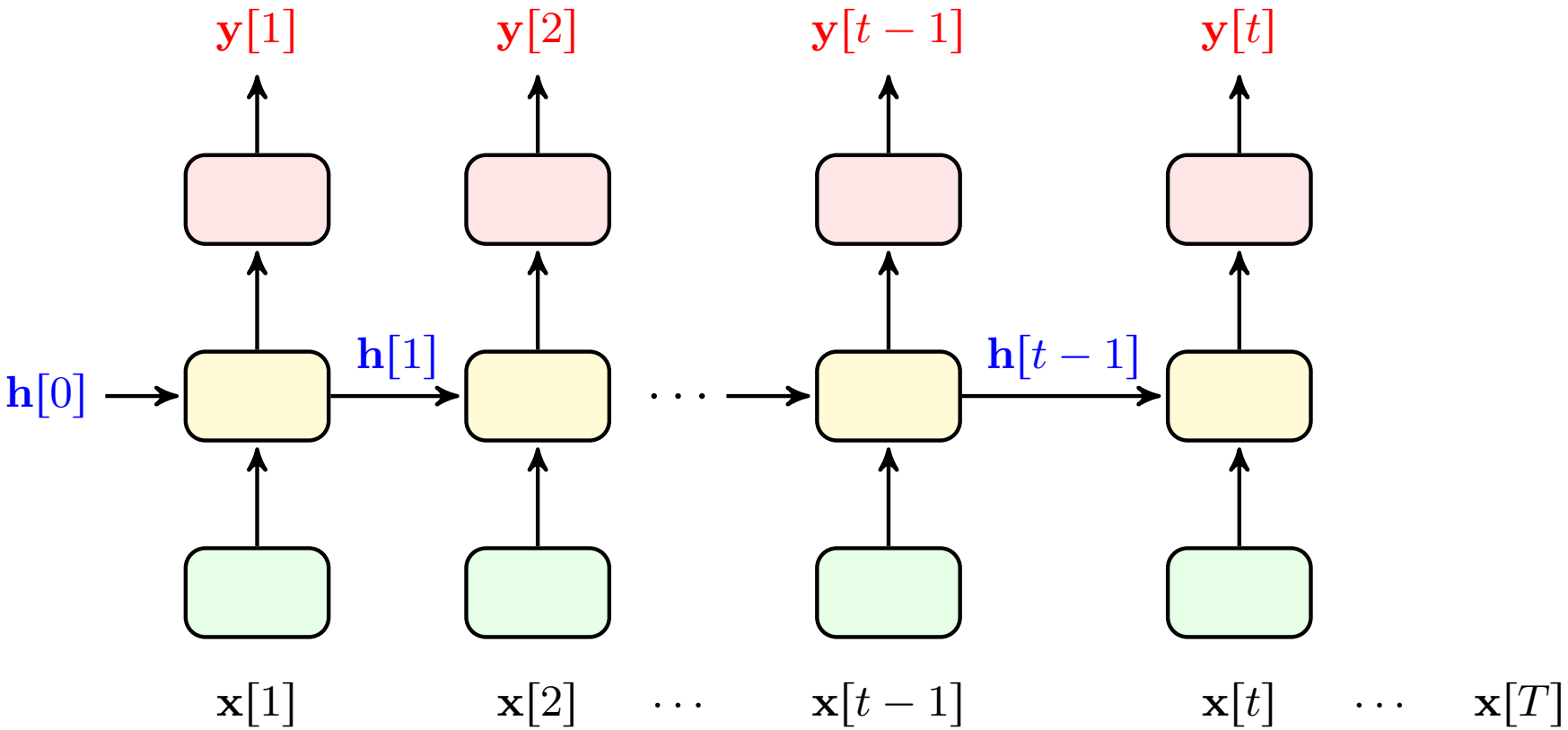
- *The hidden layer computes*

$$\mathbf{h}[t] = f\left(\mathbf{W}_1\mathbf{x}[t] + \mathbf{W}_\mathrm{m}\mathbf{s}[t]\right)$$

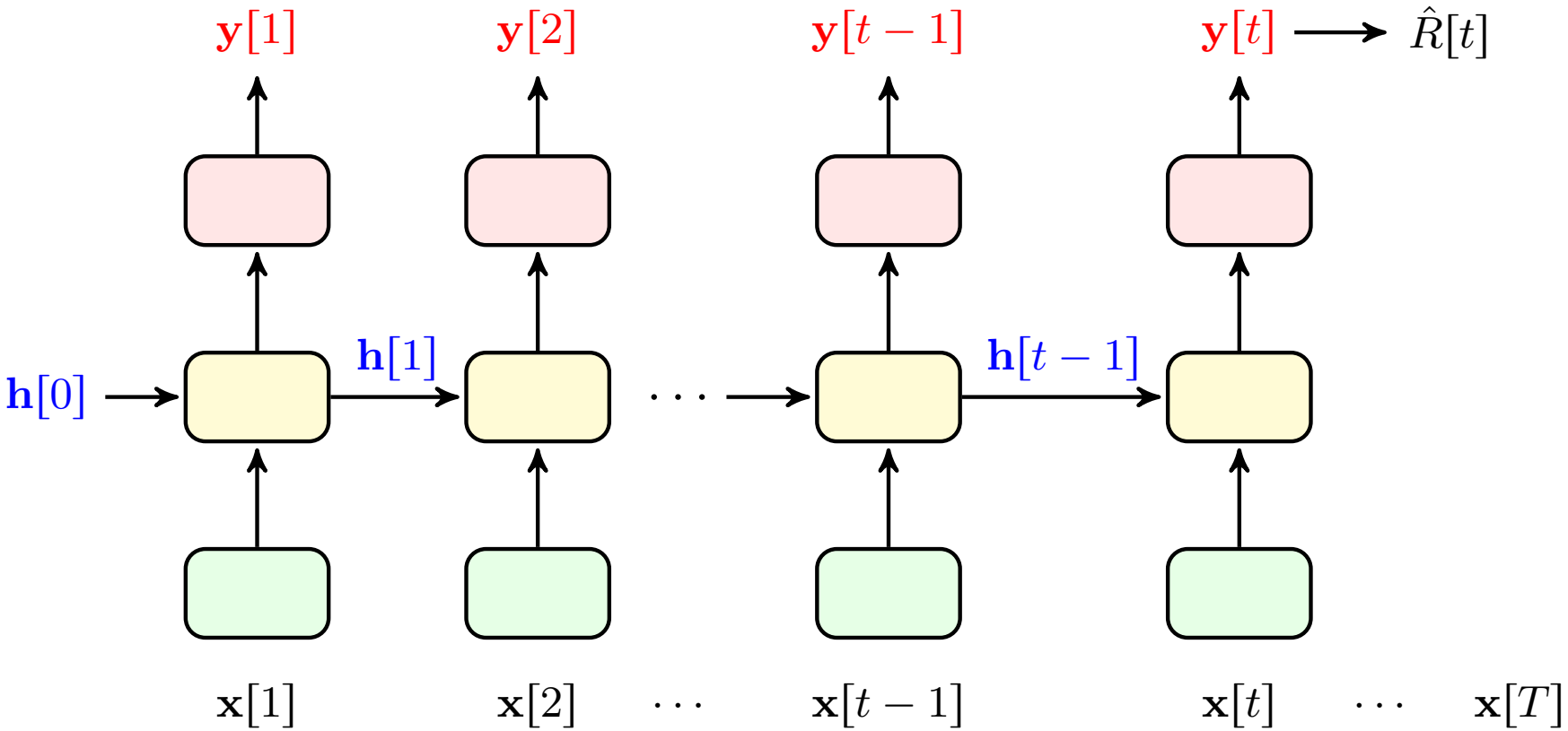
- *The output is*

$$\mathbf{y}[t] = f\left(\mathbf{W}_2\mathbf{h}[t]\right)$$

Inferring Through Time: *Elman Network*



Learning Through Time: *Elman Network*



$$\hat{R} = \sum_{t=1}^{T} \underbrace{\mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[t]\right)}_{\hat{R}[t]} = \sum_{t=1}^{T} \hat{R}[t]$$

Learning Through Time: *Elman Network*

Let's again try to learn $\mathbf{W}_1$: *we start with the output*

$$\nabla_{\mathbf{w}_1} \hat{R}[t] = \nabla_{\mathbf{y}[t]} \hat{R}[t] \circ \nabla_{\mathbf{w}_1} \mathbf{y}[t]$$

We know that $\mathbf{y}[t] = f\left(\mathbf{W}_2 \mathbf{h}[t]\right)$, *so we can write*

$$\nabla_{\mathbf{w}_1} \mathbf{y}[t] = \nabla_{\mathbf{h}[t]} \mathbf{y}[t] \circ \nabla_{\mathbf{w}_1} \mathbf{h}[t]$$

We note that $\mathbf{h}[t] = f\left(\mathbf{W}_1 \mathbf{x}[t] + \mathbf{W}_\mathrm{m} \mathbf{h}[t-1]\right)$

$$\begin{aligned}\nabla_{\mathbf{w}_1} \mathbf{h}[t] = \nabla_{\mathbf{w}_1} f\left(\mathbf{W}_1 \mathbf{x}[t] + \mathbf{W}_\mathrm{m} \mathbf{h}[t-1]\right) \\ + \nabla_{\mathbf{x}[t]} \mathbf{h}[t] \circ \underbrace{\nabla_{\mathbf{w}_1} \mathbf{x}[t]}_{\mathbf{0}} \\ + \nabla_{\mathbf{h}[t-1]} \mathbf{h}[t] \circ \underbrace{\nabla_{\mathbf{w}_1} \mathbf{h}[t-1]}_{?}\end{aligned}$$

Learning Through Time: *Elman Network*

Well! We know that $\mathbf{h}[t-1] = f\left(\mathbf{W}_1\mathbf{x}[t-1] + \mathbf{W}_\mathrm{m}\mathbf{h}[t-2]\right)$

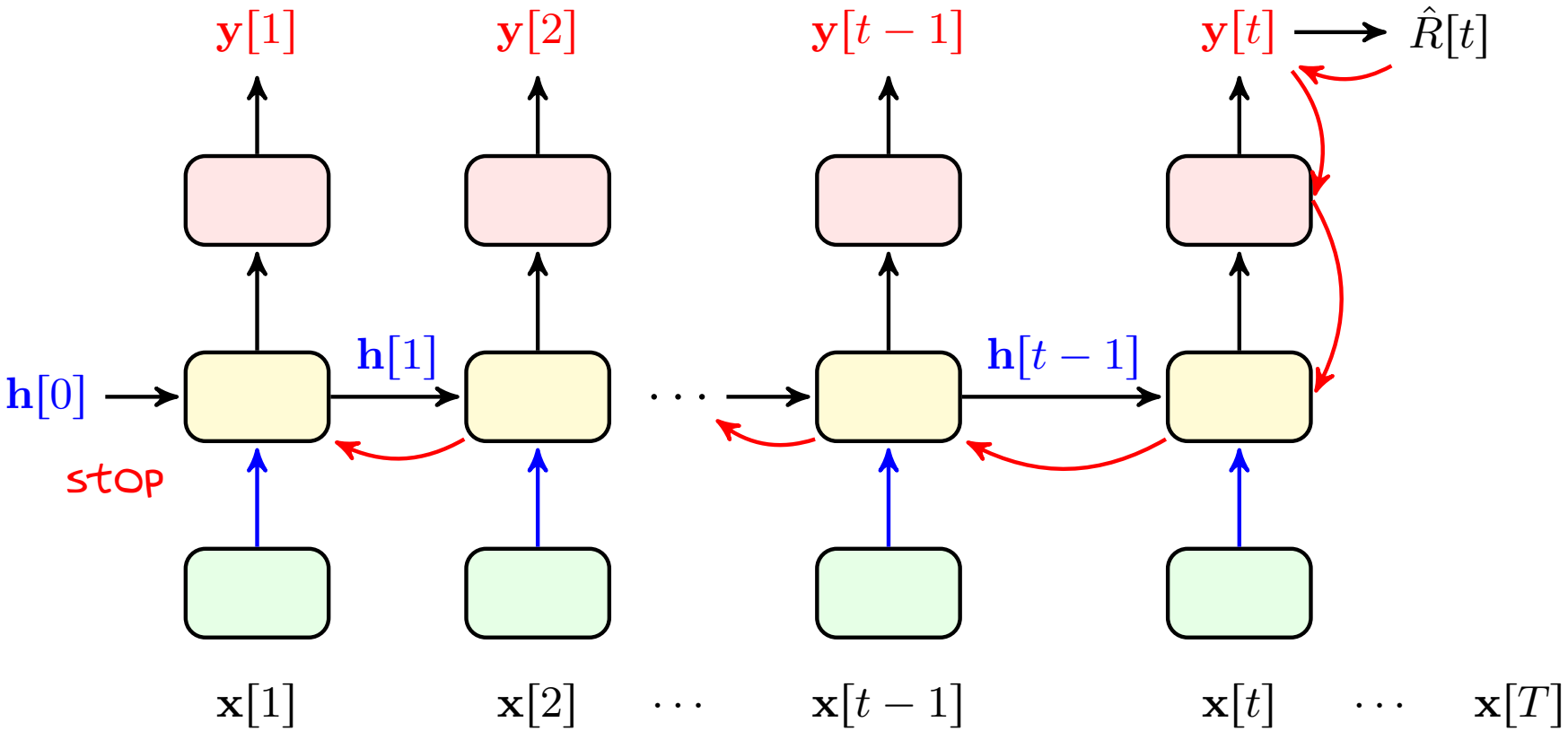
$$\begin{aligned}\nabla_{\mathbf{W}_1}\mathbf{h}[t-1] = \nabla_{\mathbf{W}_1} f\left(\mathbf{W}_1\mathbf{x}[t-1] + \mathbf{W}_\mathrm{m}\mathbf{h}[t-2]\right) \\ + \nabla_{\mathbf{x}[t-1]}\mathbf{h}[t-1] \circ \underbrace{\nabla_{\mathbf{W}_1}\mathbf{x}[t-1]}_{\mathbf{0}} \\ + \nabla_{\mathbf{h}[t-2]}\mathbf{h}[t-1] \circ \underbrace{\nabla_{\mathbf{W}_1}\mathbf{h}[t-2]}_{?}\end{aligned}$$

We are not still done! We know $\mathbf{h}[t-2] = f\left(\mathbf{W}_1\mathbf{x}[t-2] + \mathbf{W}_\mathrm{m}\mathbf{h}[t-3]\right)$

$$\begin{aligned}\nabla_{\mathbf{W}_1}\mathbf{h}[t-2] = \nabla_{\mathbf{W}_1} f\left(\mathbf{W}_1\mathbf{x}[t-2] + \mathbf{W}_\mathrm{m}\mathbf{h}[t-3]\right) \\ + \nabla_{\mathbf{x}[t-2]}\mathbf{h}[t-2] \circ \underbrace{\nabla_{\mathbf{W}_1}\mathbf{x}[t-2]}_{\mathbf{0}} \\ + \nabla_{\mathbf{h}[t-3]}\mathbf{h}[t-2] \circ \underbrace{\nabla_{\mathbf{W}_1}\mathbf{h}[t-3]}_{?}\end{aligned}$$

Learning Through Time: *Elman Network*

We should in fact pass all the way back to the initial time interval time!



Note that all blue edges are representing $\mathbf{W}_1$

Learning Through Time

Moral of Story

To learn how to remember, we need to train our RNN through time: at each time interval, we should backpropagate all the way back to origin!

+ *Did Elman did so?*
– Not really!

Elman treated the hidden state $\mathbf{h}[t-1]$ as a fixed variable, i.e., he assumed

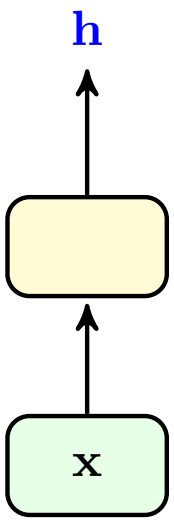
$$\nabla_{\mathbf{w}_1}\mathbf{h}[t-1] \approx \nabla_{\mathbf{w}_\mathrm{m}}\mathbf{h}[t-1] = \mathbf{0}$$

So, he did not need to move backward in time!

This means that Elman did not really learn true memory!

Basic RNN

Let's break it down a bit: *say the yellow box is a fully-connected layer*

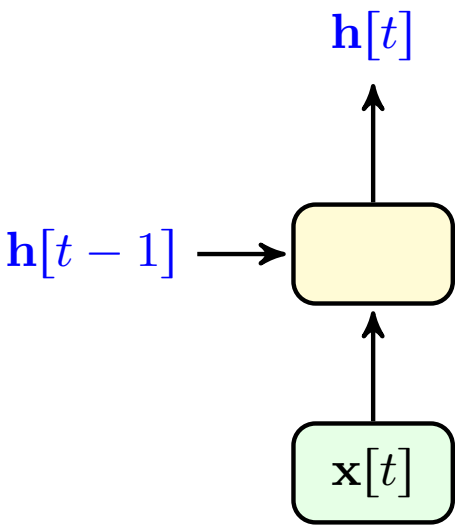


It computes activated feature $\mathbf{h} \in \mathbb{R}^M$

- *We replace its activation by* $\tanh(\cdot)$
- *We modify it to get a new input in* $\mathbb{R}^{N+M}$
  - ↳ N *entries for data inputs* $\mathbf{x}$ *in time* t
  - ↳ M *entries for features* $\mathbf{h}$ *in time* $t-1$

Basic RNN

Let's break it down a bit: *say the yellow box is a fully-connected layer*

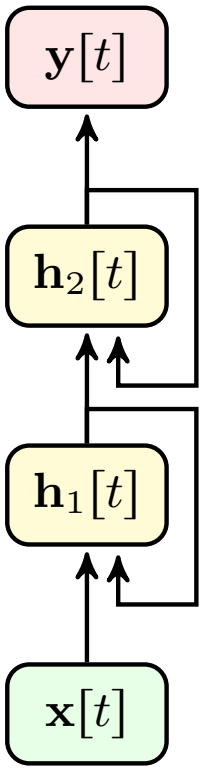


We use the following terminology

- *We call* $\mathbf{h}[t]$ *the hidden state*
- *We pass hidden state to output layer*

Deep RNN

We can add more layers to make a deep RNN

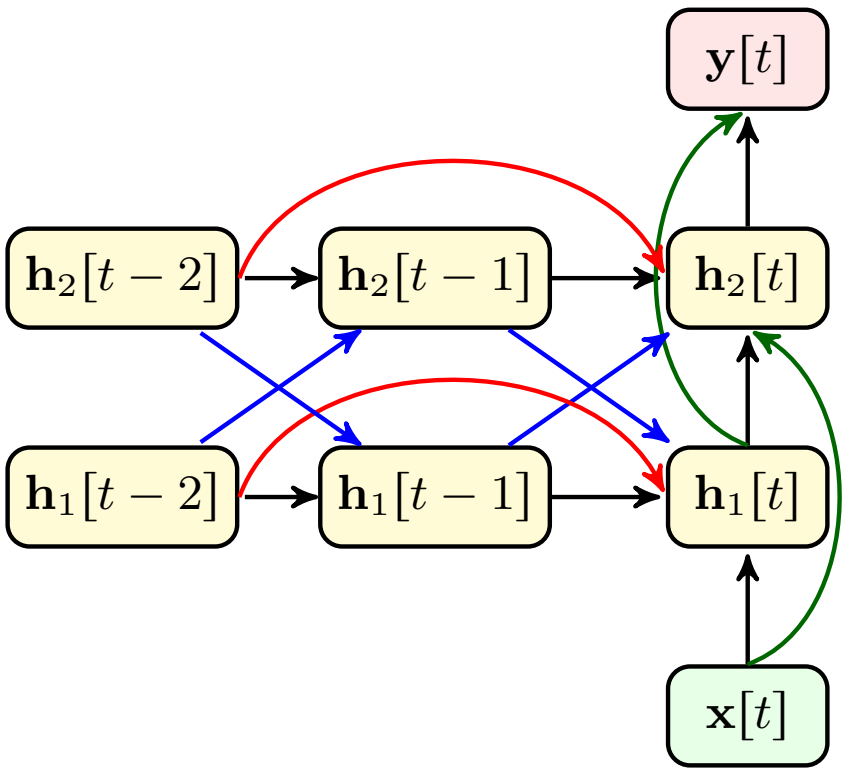


In this diagram

- *Each edge is a set of weights*
- *The return edges make delay*

Deep RNN

It might be easier to think of it as the following diagram

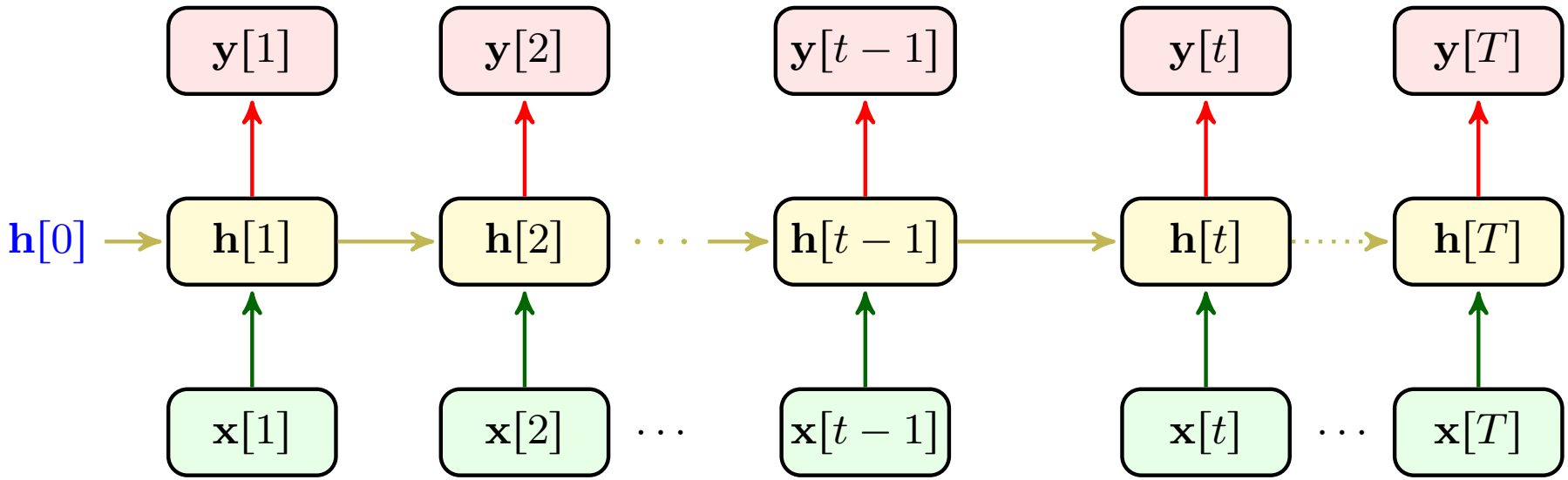


We can use any module that we like

- *We may add skip connection*
- *We could make dense connections*
- *We may skip over time*
- *We may replace hidden layers with convolutions or anything else*

Basic RNN: *Forward Propagation*

Let's specify the learnable parameters with some colors

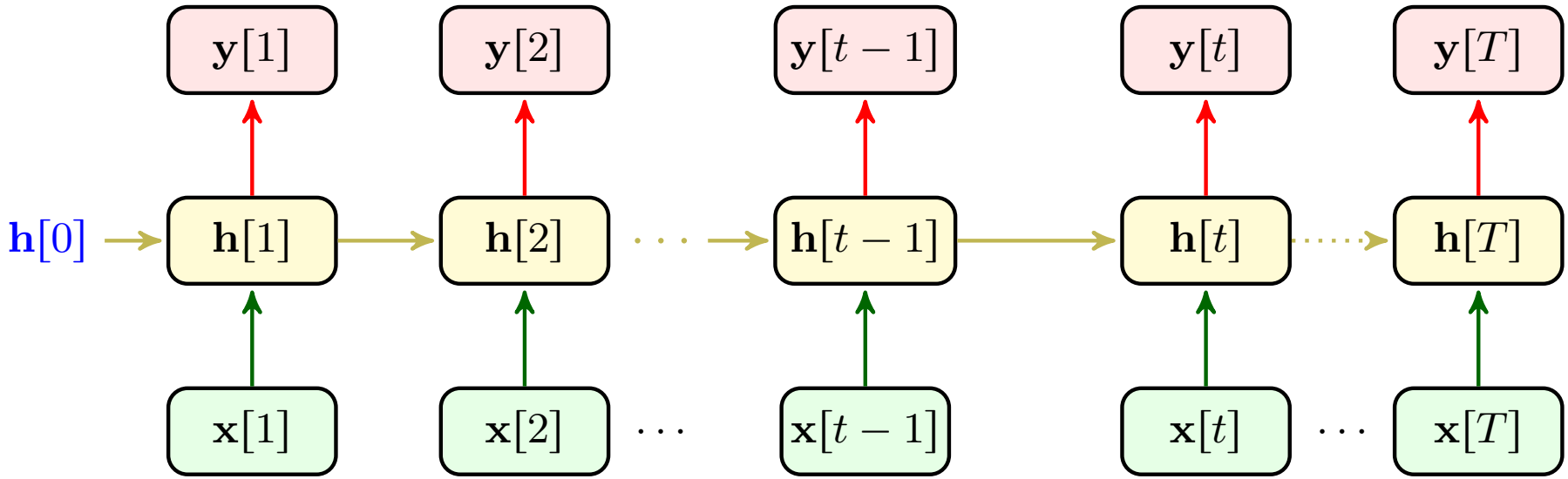


Say we set the activation to $f(\cdot)$

1. *We have* $\mathbf{y}[t] = f(\mathbf{W}_2\mathbf{h}[t])$*: we can learn* $\mathbf{W}_2$
2. *We have* $\mathbf{h}[t] = f(\mathbf{W}_1\mathbf{x}[t] + \mathbf{W}_\mathrm{m}\mathbf{h}[t-1])$*: we can learn* $\mathbf{W}_1$ *and* $\mathbf{W}_\mathrm{m}$
3. *We can start with any hidden state: this means we can learn* $\mathbf{h}[0]$ *too*

Training Basic RNN

We now want to train this basic RNN

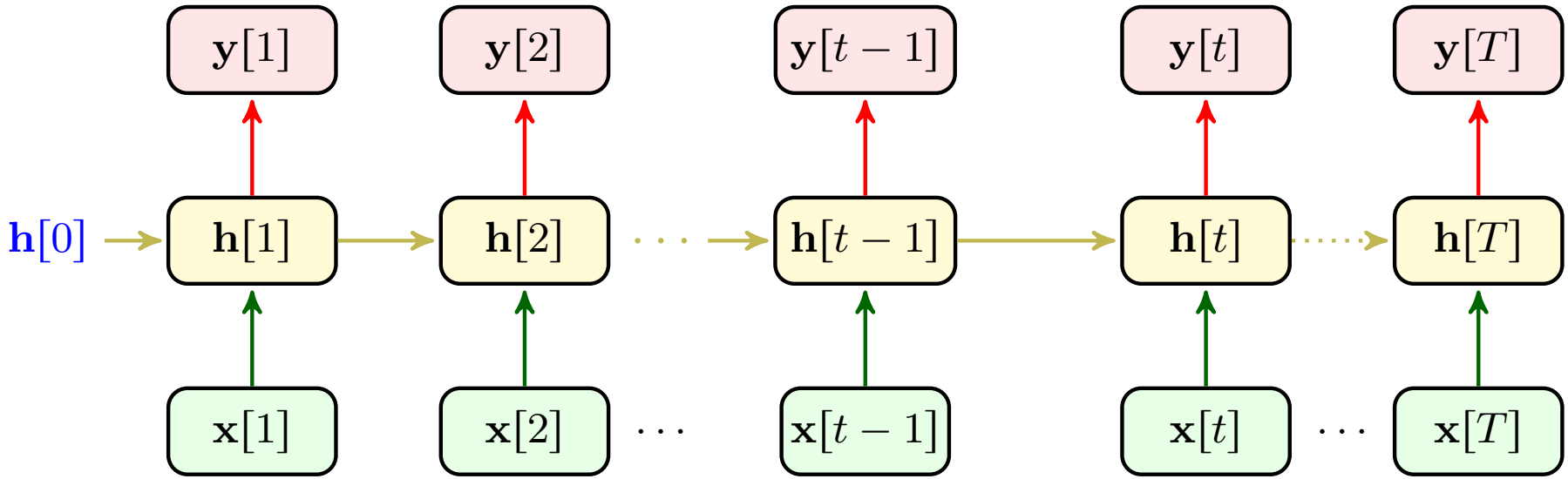


We assume that we have a sequence of labels $\mathbf{v}[1], \ldots, \mathbf{v}[T]$

1. *We know some $\mathbf{v}[t]$ could be empty*
2. *So could be some of $\mathbf{x}[t]$'s*

Training Basic RNN

We now want to train this basic RNN

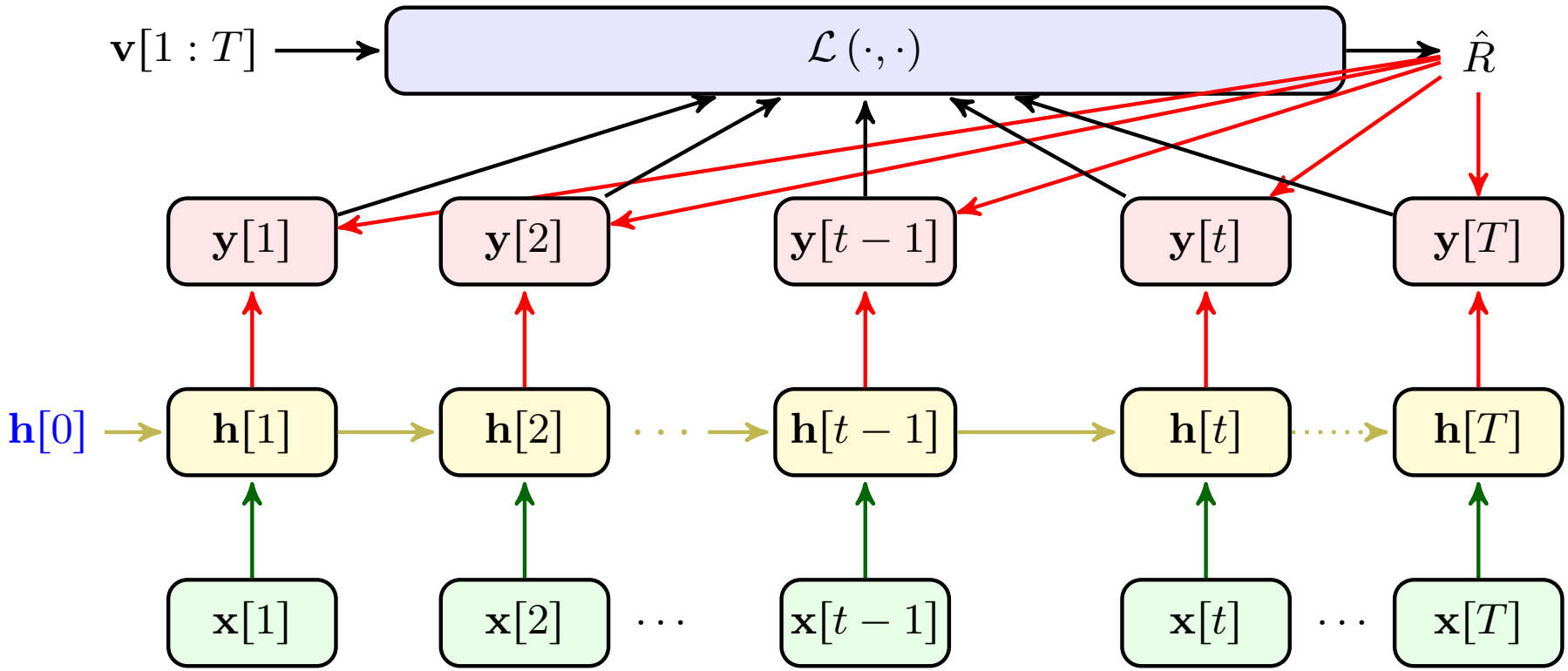


The loss in general can be written as

$$\hat{R} = \mathcal{L}\left(\mathbf{y}[1:T], \mathbf{v}[1:T]\right)$$

where we use shorten notation $\mathbf{y}[1:T] = \mathbf{y}[1], \ldots, \mathbf{y}[T]$

Training Basic RNN



We compute $\nabla_{\mathbf{y}[t]}\hat{R}$ for all $t \equiv$ move backward from $\hat{R}$ to any $\mathbf{y}[t]$

Backward Pass Through Time

Starting from $\hat{R}$, *say we want to find* $\nabla_{\mathbf{W}_{\mathrm{m}}} \hat{R}$

- *Since* $\hat{R}$ *is a function of* $\mathbf{y}[1:T]$, *we should write a vectorized chain rule*

$$\nabla_{\mathbf{W}_{\mathrm{m}}} \hat{R} = \nabla_{\mathbf{y}[1]} \hat{R} \circ \nabla_{\mathbf{W}_{\mathrm{m}}} \mathbf{y}[1] + \ldots + \nabla_{\mathbf{y}[T]} \hat{R} \circ \nabla_{\mathbf{W}_{\mathrm{m}}} \mathbf{y}[T]$$

$$= \sum_{t=1}^{T} \nabla_{\mathbf{y}[t]} \hat{R} \circ \nabla_{\mathbf{W}_{\mathrm{m}}} \mathbf{y}[t]$$

- *Since we assumed that we have* $\nabla_{\mathbf{y}[t]} \hat{R}$, *our main task is to find*

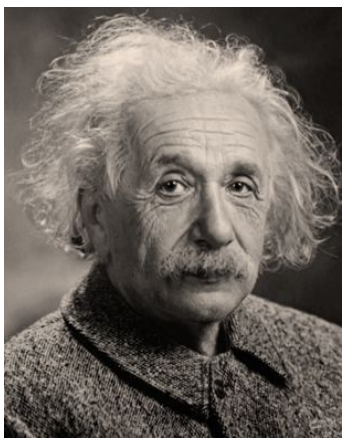
$$\nabla_{\mathbf{W}_{\mathrm{m}}} \mathbf{y}[t]$$

for all t*: we could hence compute it for a general* t

- *Apparently, we should apply chain rule for several times!*

Backward Pass Through Time

+ *But, this is going to be exhausting?*
– Well, we could again follow Albert Einstein advice!



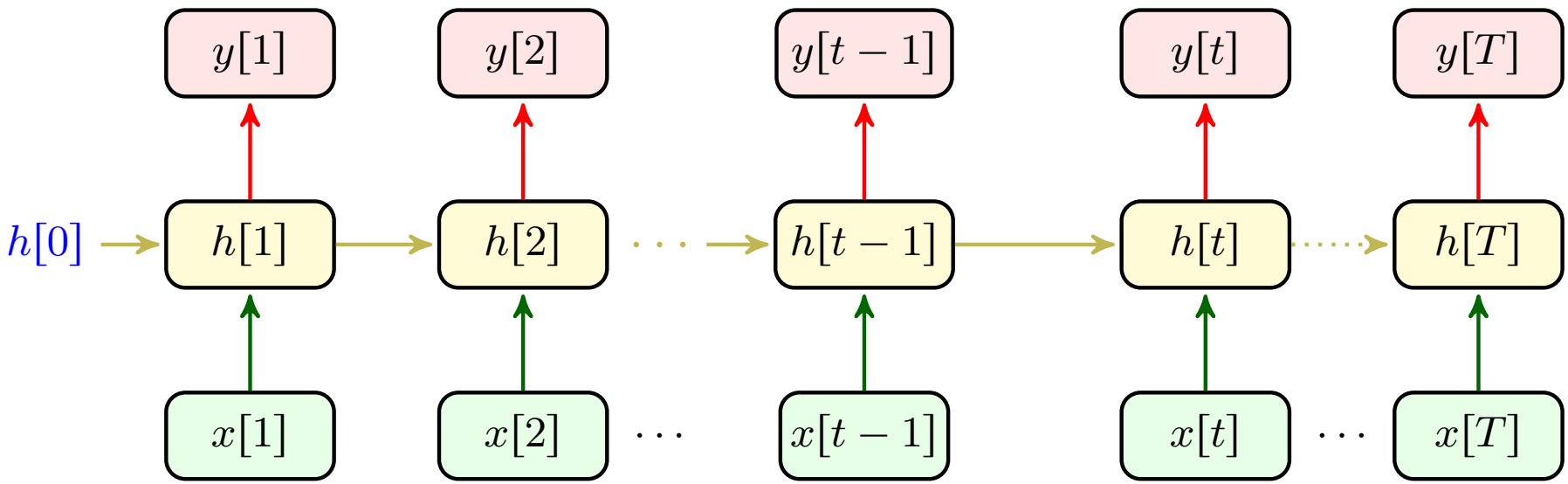
"Everything should be made as simple as possible, but not simpler!"

Backward Pass Through Time: *Simple Example*

Let's consider a dummy RNN with all variables being scalar

1. *We have* $y[t] = f(w_2 h[t])$
2. *We have* $h[t] = f(w_1 x[t] + w_{\mathrm{m}} h[t-1])$
3. *We start with hidden state* $h[0]$

So the diagram gets simplified as below



Backward Pass Through Time: *Simple Example*

Starting from $\hat{R}$, say we want to find $\nabla_{w_{\mathrm{m}}} \hat{R}$: *our main task is to find*

$$\frac{\partial y[t]}{\partial w_{\mathrm{m}}}$$

Let's start the computation: say we have passed forward through the RNN

- *we now have* $y[t]$, $h[t]$ *and* $x[t]$ *for all* t

To go backward, we note that $y[t] = f(w_2 h[t])$

- $y[t]$ *is a function of* $h[t]$*: so we write the chain rule as*

$$\frac{\partial y[t]}{\partial w_{\mathrm{m}}} = \frac{\partial y[t]}{\partial h[t]} \frac{\partial h[t]}{\partial w_{\mathrm{m}}}$$

↳ *we can compute the first term as* $\partial y[t]/\partial h[t] = w_2 \dot{f}(w_2 h[t])$
↳ *we should compute the second term by further chain rule*

Backward Pass Through Time: *Simple Example*

$$\frac{\partial y[t]}{\partial w_{\mathrm{m}}} = w_2 \dot{f}(w_2 h[t]) \frac{\partial h[t]}{\partial w_{\mathrm{m}}}$$

We keep going backward, by noting that $h[t] = f(w_1 x[t] + w_{\mathrm{m}} h[t-1])$

- $h[t]$ *is a function of* w_{m} *and* $h[t-1]$.[3]
  - ↳ *let's write* $h[t] = g\left(w_{\mathrm{m}}, h[t-1]\right)$
- *So we write the chain rule as*

$$\frac{\partial h[t]}{\partial w_{\mathrm{m}}} = \frac{\partial g}{\partial w_{\mathrm{m}}} \underbrace{\frac{\partial w_{\mathrm{m}}}{\partial w_{\mathrm{m}}}}_{1} + \frac{\partial g}{\partial h[t-1]} \frac{\partial h[t-1]}{\partial w_{\mathrm{m}}}$$

$$= \frac{\partial g}{\partial w_{\mathrm{m}}} + \frac{\partial g}{\partial h[t-1]} \frac{\partial h[t-1]}{\partial w_{\mathrm{m}}}$$

[3] *We ignore* w_1 *and* $x[t]$*, as they are obviously not functions of* w_{m}

Backward Pass Through Time: *Simple Example*

$$\frac{\partial y[t]}{\partial w_{\mathrm{m}}} = w_2 \dot{f}(w_2 h[t]) \frac{\partial h[t]}{\partial w_{\mathrm{m}}}$$

We keep going backward, by noting that $h[t] = f(z[t])$

- *Let's define* $z[t] = w_1 x[t] + w_{\mathrm{m}} h[t-1]$
- *We can simplify the chain rule as*

$$\begin{aligned}\frac{\partial h[t]}{\partial w_{\mathrm{m}}} &= h[t-1]\dot{f}\left(z[t]\right) + w_{\mathrm{m}}\dot{f}\left(z[t]\right)\frac{\partial h[t-1]}{\partial w_{\mathrm{m}}} \\ &= \dot{f}\left(z[t]\right)\left(h[t-1] + w_{\mathrm{m}}\frac{\partial h[t-1]}{\partial w_{\mathrm{m}}}\right)\end{aligned}$$

Backward Pass Through Time: *Simple Example*

$$\frac{\partial y[t]}{\partial w_{\mathrm{m}}} = w_2 \dot{f}(w_2 h[t]) \dot{f}\left(z[t]\right) \left(h[t-1] + w_{\mathrm{m}} \frac{\partial h[t-1]}{\partial w_{\mathrm{m}}} \right)$$

We keep going backward: $h[t-1] = f(z[t-1])$

↳ *Recall that* $z[t-1] = w_1 x[t-1] + w_{\mathrm{m}} h[t-2]$

- *We just need to replace* t *with* $t-1$ *in the last derivation*

$$\frac{\partial h[t-1]}{\partial w_{\mathrm{m}}} = \dot{f}\left(z[t-1]\right) \left(h[t-2] + w_{\mathrm{m}} \frac{\partial h[t-2]}{\partial w_{\mathrm{m}}} \right)$$

Backward Pass Through Time: *Simple Example*

Backpropagation at time t is hence described by a pair of recursive equations

- *At time t, we compute*

$$\frac{\partial y[t]}{\partial w_{\mathrm{m}}} = w_2 \dot{f}(w_2 h[t]) \frac{\partial h[t]}{\partial w_{\mathrm{m}}}$$

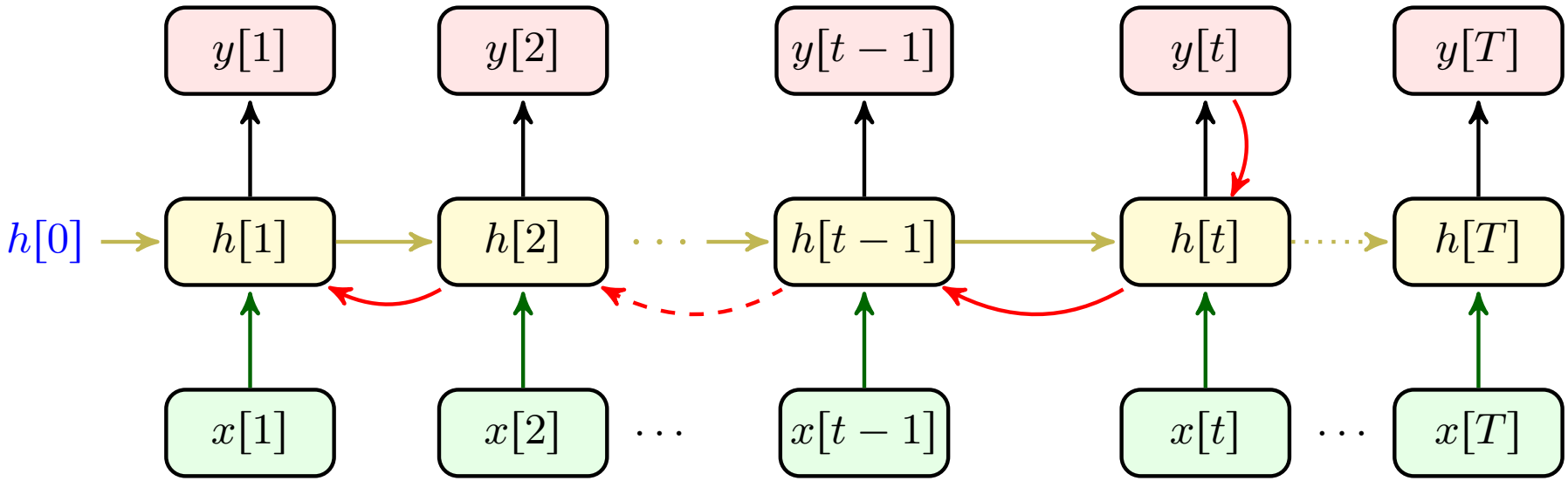
- *For each $i = t, t-1, \ldots, 1$, we use*

$$\frac{\partial h[i]}{\partial w_{\mathrm{m}}} = \dot{f}\left(z[i]\right) \left(h[i-1] + w_{\mathrm{m}} \frac{\partial h[i-1]}{\partial w_{\mathrm{m}}} \right)$$

- *We stop at $i = 1$, where we get*

$$\frac{\partial h[1]}{\partial w_{\mathrm{m}}} = \dot{f}\left(z[1]\right) \left(h[0] + w_{\mathrm{m}} \underbrace{\frac{\partial h[0]}{\partial w_{\mathrm{m}}}}_{0} \right) = \dot{f}\left(z[1]\right) h[0]$$

Backward Pass Through Time: *Simple Example*



Key Point

Propagating back in time is described via a recursive equation

Backward Pass Through Time: *Simple Example*

Recursive equation has an interesting feature

- *Say we have backpropagated from time* t

$$\frac{\partial y[t]}{\partial w_{\mathrm{m}}} = w_2 \dot{f}(w_2 h[t]) \frac{\partial h[t]}{\partial w_{\mathrm{m}}}$$

  - ↳ *We hence have* $\partial h[t]/\partial w_{\mathrm{m}}, \partial h[t-1]/\partial w_{\mathrm{m}}, \ldots, \partial h[1]/\partial w_{\mathrm{m}}$
- *Now, if we want to backpropagate from* $t-1$
  - ↳ *We already have* $\partial h[t-1]/\partial w_{\mathrm{m}}, \partial h[t-2]/\partial w_{\mathrm{m}}, \ldots, \partial h[1]/\partial w_{\mathrm{m}}$
  - ↳ *We do not need to backpropagate through time anymore*

Moral of Story

Pass forward till end of sequence and backpropagate to the beginning just once

Backpropagation Through Time (BPTT)

BPTT():

1. Start at t with $\nabla_{\mathbf{w}_\mathrm{m}} \mathbf{y}[t] = \mathbf{W}_2 \circ \dot{f}(\mathbf{W}_2 \mathbf{h}[t]) \circ \nabla_{\mathbf{w}_\mathrm{m}} \mathbf{h}[t]$
2. Go back in time as $\nabla_{\mathbf{w}_\mathrm{m}} \mathbf{h}[i] = \dot{f}\left(\mathbf{z}[i]\right) \circ (\mathbf{h}[i-1] + \mathbf{W}_\mathrm{m} \circ \nabla_{\mathbf{w}_\mathrm{m}} \mathbf{h}[i-1])$
3. Stop at $i = 1$ with $\nabla_{\mathbf{w}_\mathrm{m}} \mathbf{h}[1] = \dot{f}\left(\mathbf{z}[1]\right) \circ \mathbf{h}[0]$

\+ *It looks very similar to backpropagation in deep NNs!*

– Exactly! *Even simple RNN is very deep through time*

\+ *Don't we experience vanishing or exploding gradient then!*

– Yes! Let's check it out

BPTT: *Vanishing Gradient*

Let's expand the gradient in our example

$$\frac{\partial y[t]}{\partial w_{\mathrm{m}}} = \underbrace{w_2 \dot{f}(w_2 h[t]) \dot{f}\left(z[t]\right)}_{\alpha_1} h[t-1]$$
$$+ \underbrace{w_2 \dot{f}(w_2 h[t]) w_{\mathrm{m}} \dot{f}\left(z[t]\right) \dot{f}\left(z[t-1]\right)}_{\alpha_2} h[t-2] + \cdots$$
$$+ \underbrace{w_2 \dot{f}(w_2 h[t]) w_{\mathrm{m}}^{i-1} \left(\prod_{j=0}^{i-1} \dot{f}\left(z[t-j]\right) \right)}_{\alpha_i} h[t-i] + \cdots$$
$$+ \underbrace{w_2 \dot{f}(w_2 h[t]) w_{\mathrm{m}}^{t-1} \left(\prod_{j=0}^{t-1} \dot{f}\left(z[t-j]\right) \right)}_{\alpha_t} h[0]$$

BPTT: *Vanishing Gradient*

$$\frac{\partial y[t]}{\partial w_{\mathrm{m}}} = \sum_{i=1}^{t} \alpha_i h[t-i]$$

This says that gradient of hidden state in i steps back in time is multiplied by

$$\alpha_i = w_2 w_{\mathrm{m}}^{i-1} \dot{f}(w_2 h[t]) \left(\prod_{j=0}^{i-1} \dot{f}\left(z[t-j]\right) \right)$$

Recall deep NNs: we could see two cases

- *If $\dot{f}(\cdot) > 1$ most of the time*
  - ↳ *very old hidden states can explode the gradient*
- *If $\dot{f}(\cdot) < 1$ most of the time*
  - ↳ *very old hidden states have pretty much no impact on the gradient*

Handling *Vanishing Gradient Through Time*

+ *Cant we use the same remedy as in deep NNs?*
– Yes and No!

Nature of Vanishing in BPTT

It is important to note that *vanishing gradient through time is a bit different: we are still updating weights with large enough gradients, but these gradients have no information of long-term memory*

Solutions to vanishing gradient through time are mainly

- *Use* $\tanh(\cdot)$ *activation*
  - ↳ $\tanh(\cdot)$ *is able to keep memory for a longer period*
- *Use truncated BPTT*
  - ↳ *Repeat short-term BPTTs every couple of time intervals*
- *Invoke gating approach*
  - ↳ *This is the most sophisticated approach*

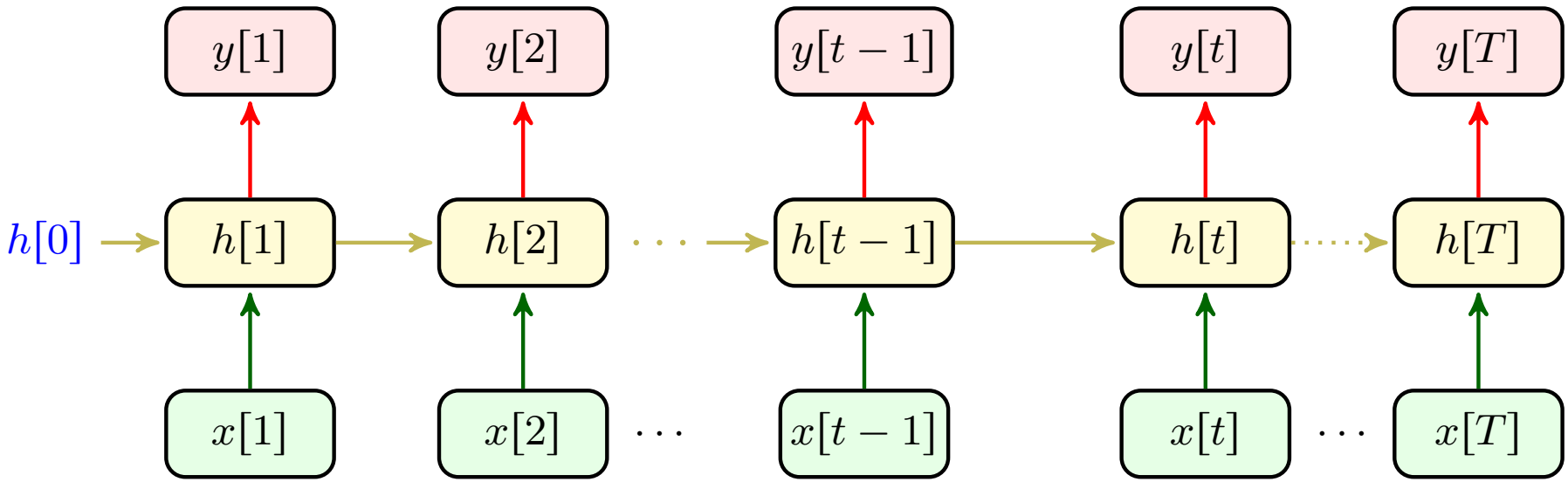


UNIVERSITY OF
TORONTO

Gated Architectures

Principle of Gating

Let's get back to our basic RNN



Looking at $h[t]$ as memory, we can say we are always updating the memory

Principle of Gating

Recall our motivating example

$$\mathbf{x}[t-6] \quad \mathbf{x}[t-5] \quad \mathbf{x}[t-4] \quad \mathbf{x}[t-3] \quad \mathbf{x}[t-2] \quad \mathbf{x}[t-1] \quad \mathbf{x}[t]$$

```
... Julia has been nominated to receive Alexander von Humboldt Prize for her
```

Should we update the memory all the way from Julia?

- ✖ *Obviously No!* $\rightsquigarrow$ *we should stop updating at Julia*
- ? *How can we do it? Let's try a thought experiment*

Principle of Gating

We build $u[t] \in [0, 1]$: say we get to *"Julia"* at time t_0, i.e., $x[t_0] \propto$ *"Julia"*

- ↳ At time t_0, we set $u[t_0] = 1$
- ↳ From $t_0 + 1$ to t, we have $u[t_0 + 1] = \cdots = u[t] = 0$

Now, we update our memory like this

1. *We start with hidden state* $h[0]$
2. *We compute* $\tilde{h}[t] = f(w_1 x[t] + w_m h[t-1])$
3. *We update* $h[t] = u[t]\tilde{h}[t] + (1 - u[t])\, h[t-1]$
4. *At each time, we have* $y[t] = f(w_2 h[t])$

Let's see what happens to the memory

Principle of Gating

At time t_0*:* $x[t_0] \propto$ *"Julia"*

- We compute $\tilde{h}[t_0] = f(w_1 x[t_0] + w_{\mathrm{m}} h[t_0 - 1])$
  - ↳ $\tilde{h}[t_0]$ has fresh memory about *"Julia"*
  - ↳ $u[t_0] = 1 \rightsquigarrow$ we update $h[t_0] = 1 \times \tilde{h}[t_0] + 0 \times h[t_0 - 1] = \tilde{h}[t_0]$
  - ↳ RNN has a fresh memory about *"Julia"*

At time $t_0 + 1$

- No matter what $\tilde{h}[t_0 + 1]$ is we have $u[t_0 + 1] = 0$
  - ↳ Memory is $h[t_0 + 1] = 0 \times \tilde{h}[t_0 + 1] + 1 \times h[t_0] = h[t_0] = \tilde{h}[t_0]$
  - ↳ We still have a fresh memory about *"Julia"*

This repeats from $t_0 + 1$ *till* t*, so at time* t

- No matter what $\tilde{h}[t]$, we have $u[t] = 0 \rightsquigarrow h[t] = 0$
  - ↳ We still have a fresh memory about *"Julia"*

Principle of Gating: *Updating via Gates*

$u[t]$ *gates the memory: it decides how much memory we should pass and forget*

- *We update* $h[t] = u[t]\tilde{h}[t] + (1 - u[t])\, h[t-1]$

? *How does it help with vanishing gradient?*

Well! *It is implicitly making skip connections through time*

Recall that *with standard BPTT, we have for* $i = t, t-1, \ldots, t_0$

$$\frac{\partial h[i]}{\partial w_{\mathrm{m}}} = \dot{f}\left(z[i]\right)\left(h[i-1] + w_{\mathrm{m}}\frac{\partial h[i-1]}{\partial w_{\mathrm{m}}}\right)$$

But, now we skip multiple time slots, as we have for $i = t, t-1, \ldots, t_0$

$$\frac{\partial h[i]}{\partial w_{\mathrm{m}}} = \frac{\partial h[i-1]}{\partial w_{\mathrm{m}}}$$

Principle of Gating: *A Generic Gate*

+ *Sounds inspiring! But, how could you get* $u[t]$*?*
– Well! *Like what we did the whole time: we learn it!*

Gate

Let $\mathbf{x}[t]$ *be input and* $\mathbf{h}[t-1]$ *be last hidden state: a gate* $\mathbf{\Gamma}[t]$ *is computed as*

$$\mathbf{\Gamma}[t] = \sigma\left(\mathbf{W}_{\Gamma,\text{in}}\mathbf{x}[t] + \mathbf{W}_{\Gamma,\text{m}}\mathbf{h}[t-1] + \mathbf{b}_{\Gamma}\right)$$

where $\sigma\left(\cdot\right)$ *is sigmoid function, and* $\mathbf{W}_{\Gamma,\text{in}}$, $\mathbf{W}_{\Gamma,\text{m}}$ *and* $\mathbf{b}_{\Gamma}$ *are learnable*

+ *Why do we use sigmoid function?*
– Simply because *it is between 0 and 1*
+ *What should we set the dimension of* $\mathbf{\Gamma}[t]$*?*
- Same as the *variable (memory component) that we want to gate*

Practical Gated Architectures

There are various *gated* architectures: *we look into two of them*

1. *Gated Recurrent Unit (GRU)*
2. *Long Short-Term Memory (LSTM)*

Before we start, let's recall their basic RNN counterpart

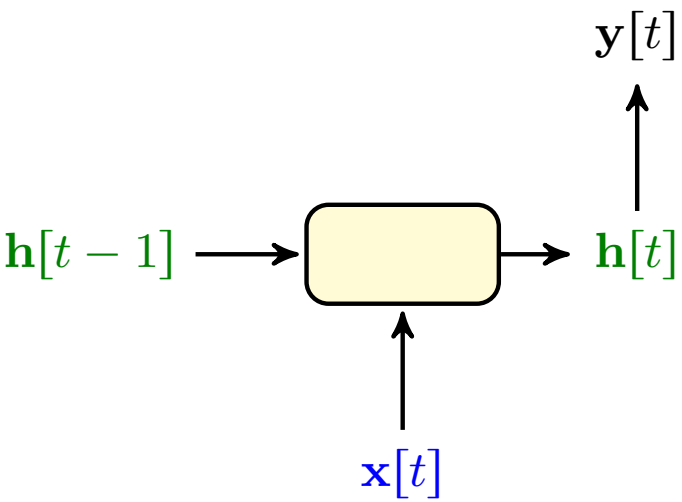
Basic RNN Counterpart

Say we set activation to $f(\cdot)$ ⇝ we usually set it to $\tanh(\cdot)$

1. *Start with an initial hidden state*
   ↳ *we can learn* $\mathbf{h}[0]$
2. *Compute memory as* $\mathbf{h}[t] = f(\mathbf{W}_1\mathbf{x}[t] + \mathbf{W}_\mathrm{m}\mathbf{h}[t-1])$
   ↳ *we can learn* $\mathbf{W}_1$ *and* $\mathbf{W}_\mathrm{m}$
3. *Compute output* $\mathbf{y}[t] = f_\mathrm{out}(\mathbf{W}_2\mathbf{h}[t])$ ⇝ f_out and f could be different
   ↳ *we can learn* $\mathbf{W}_2$

Classical Diagram: *Hidden Layer as Unit*

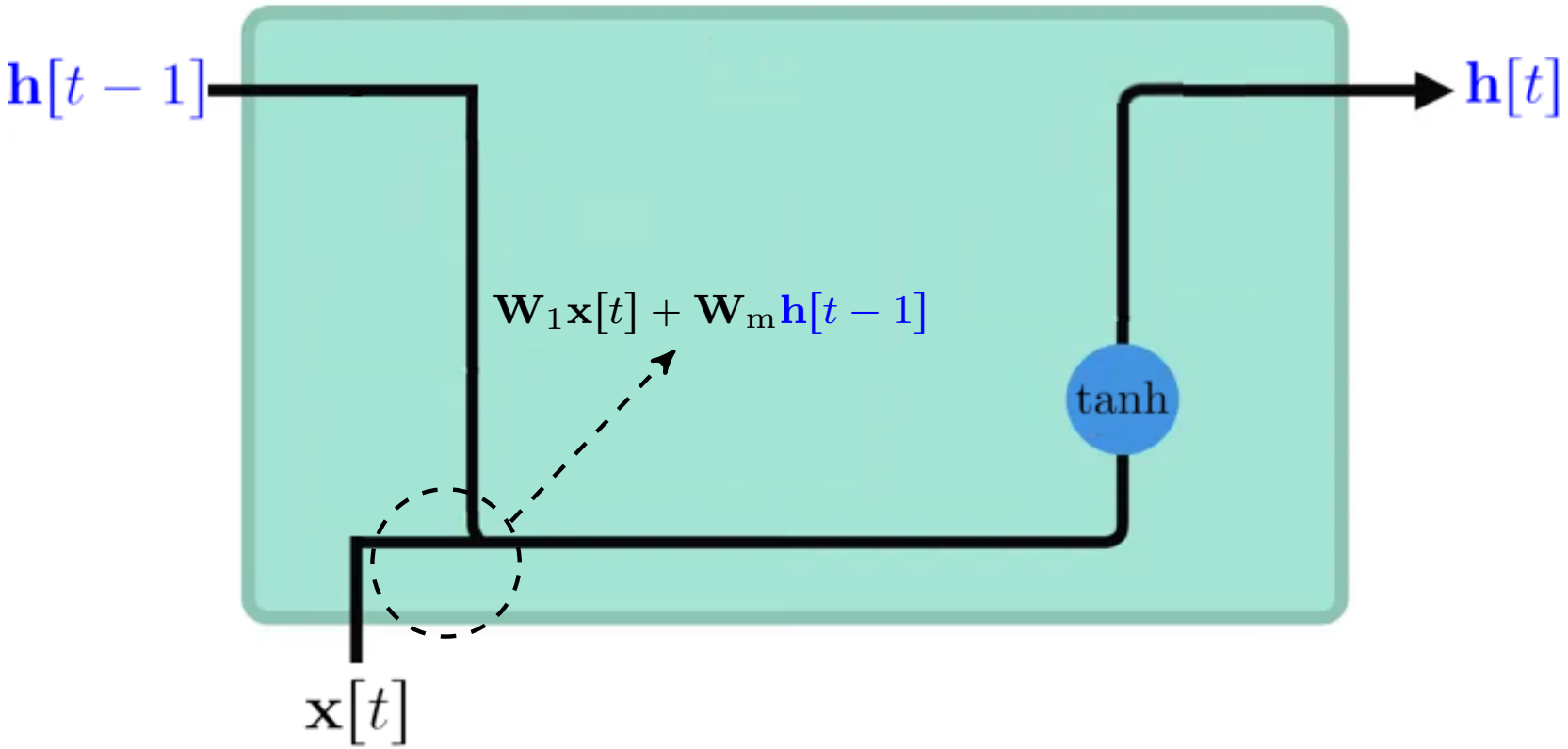
When we study a gated architecture: *it is common to look at the hidden layer as a unit which takes some inputs and returns some outputs*



We are mainly interested on this block: we want to know that given last state and new input

1. *How does this unit update hidden state?*
2. *What components are passed to the next time interval?*
   - ↳ *Here, we have only* $\mathbf{h}[t]$
   - ↳ *But, we may have other components*
     - ↳ *We will see it in LSTM*

Classical Diagram: *Basic RNN*



Practical Gated Architectures: *Gated Recurrent Unit*

Gated Recurrent Unit (GRU)

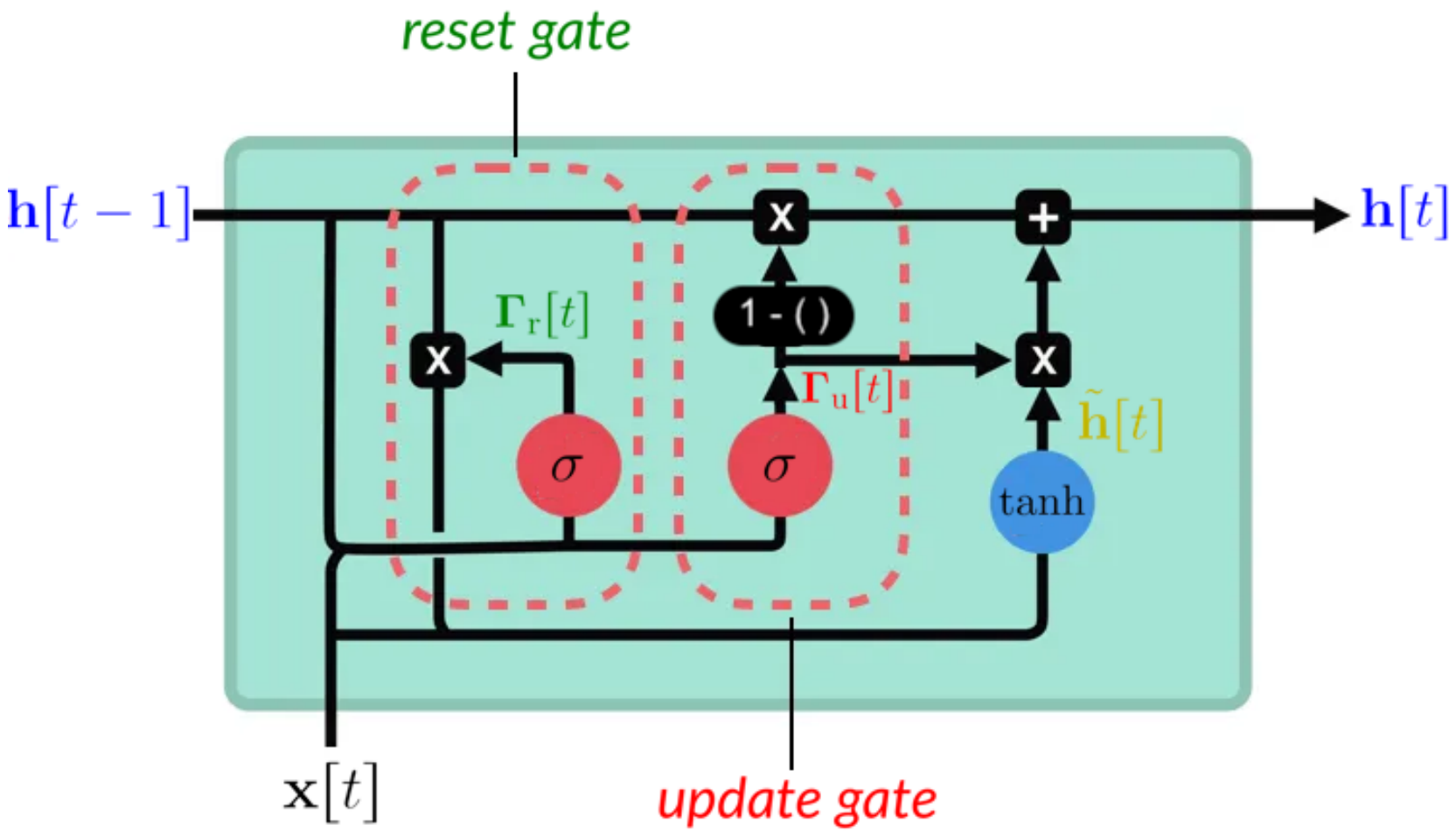
Say we set activation to $f(\cdot)$ ⇝ we usually set it to $\tanh(\cdot)$

1. *Start with an initial hidden state*
2. *Compute update gate* $\mathbf{\Gamma}_{\mathrm{u}}[t] = \sigma\left(\mathbf{W}_{\mathrm{u,in}}\mathbf{x}[t] + \mathbf{W}_{\mathrm{u,m}}\mathbf{h}[t-1]\right)$
3. *Compute reset gate* $\mathbf{\Gamma}_{\mathrm{r}}[t] = \sigma\left(\mathbf{W}_{\mathrm{r,in}}\mathbf{x}[t] + \mathbf{W}_{\mathrm{r,m}}\mathbf{h}[t-1]\right)$
4. *Compute actual memory* $\tilde{\mathbf{h}}[t] = f(\mathbf{W}_{1}\mathbf{x}[t] + \mathbf{W}_{\mathrm{m}}\mathbf{\Gamma}_{\mathrm{r}}[t] \odot \mathbf{h}[t-1])$
5. *Update hidden state as* $\mathbf{h}[t] = (1 - \mathbf{\Gamma}_{\mathrm{u}}[t]) \odot \mathbf{h}[t-1] + \mathbf{\Gamma}_{\mathrm{u}}[t] \odot \tilde{\mathbf{h}}[t]$
6. *Compute output* $\mathbf{y}[t] = f_{\mathrm{out}}(\mathbf{W}_{2}\mathbf{h}[t])$ ⇝ f_{out} and f could be different

Or we could give $\mathbf{h}[t]$ *to a new layer: for instance a new GRU whose input is* $\mathbf{h}[t]$ *and has its own state*

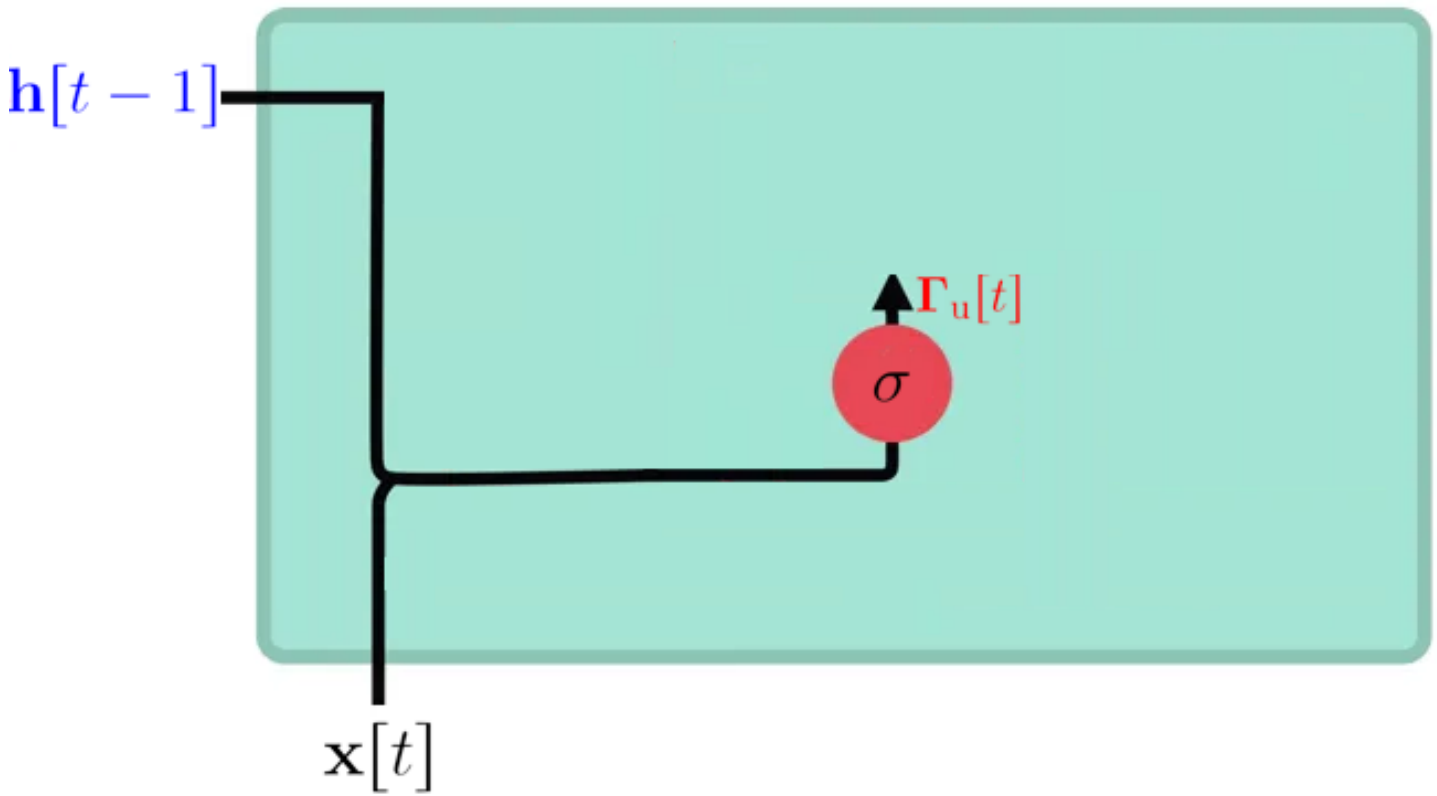
Practical Gated Architectures: *GRU*

This is what's going on in a GRU cell



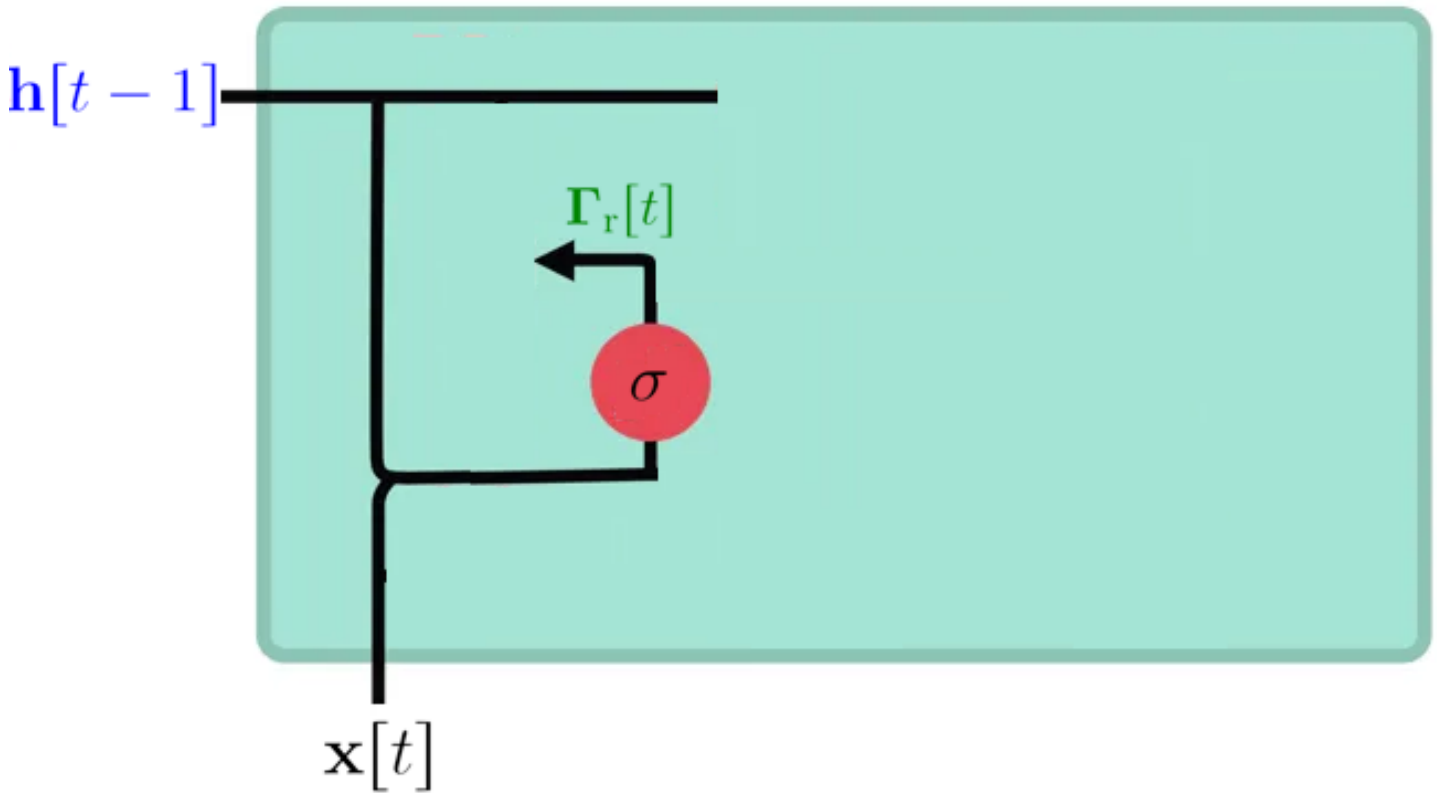
Practical Gated Architectures: *GRU*

Compute update gate $\mathbf{\Gamma}_{\mathrm{u}}[t] = \sigma\left(\mathbf{W}_{\mathrm{u,in}}\mathbf{x}[t] + \mathbf{W}_{\mathrm{u,m}}\mathbf{h}[t-1]\right)$



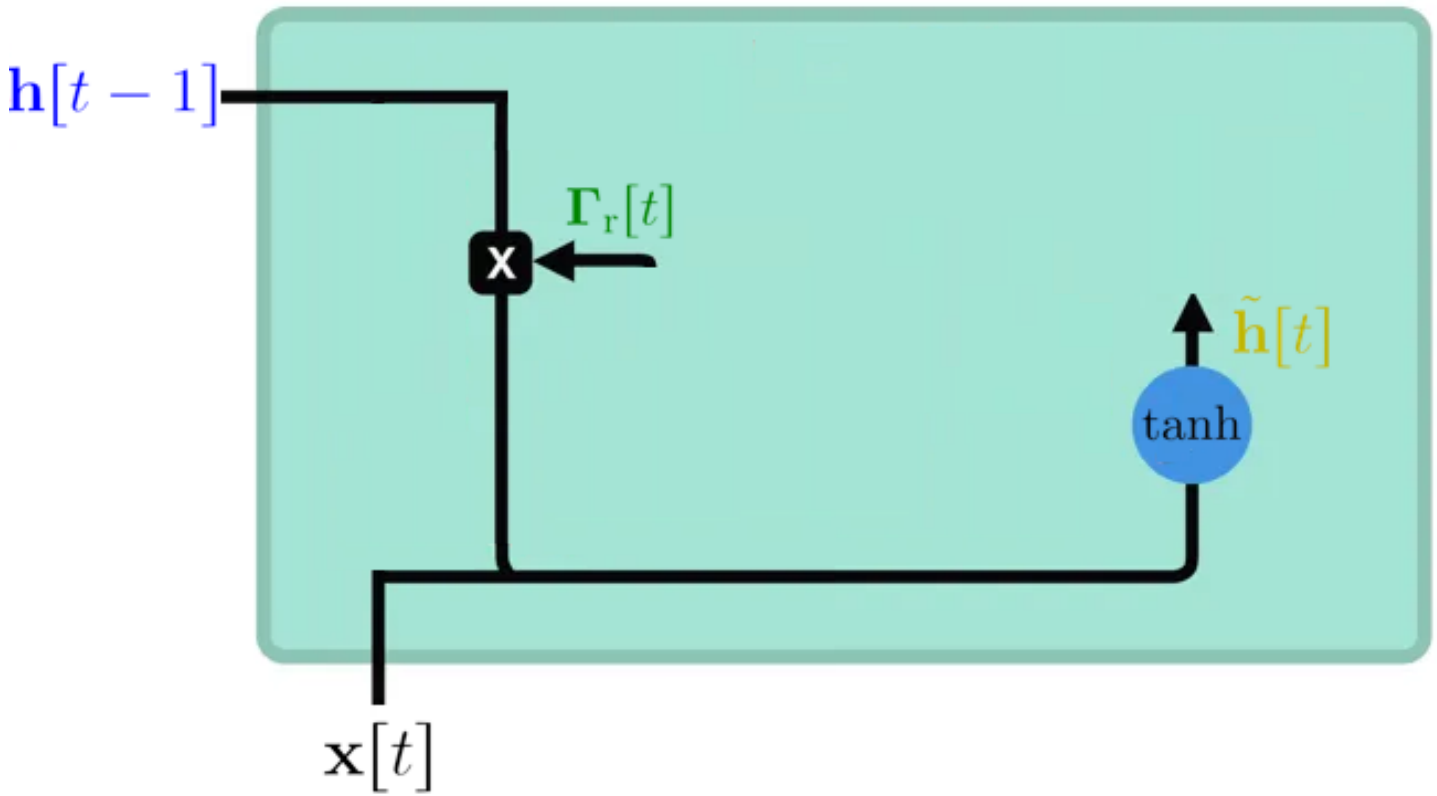
Practical Gated Architectures: *GRU*

Compute reset gate $\mathbf{\Gamma}_{\mathrm{r}}[t] = \sigma\left(\mathbf{W}_{\mathrm{r,in}}\mathbf{x}[t] + \mathbf{W}_{\mathrm{r,m}}\mathbf{h}[t-1]\right)$



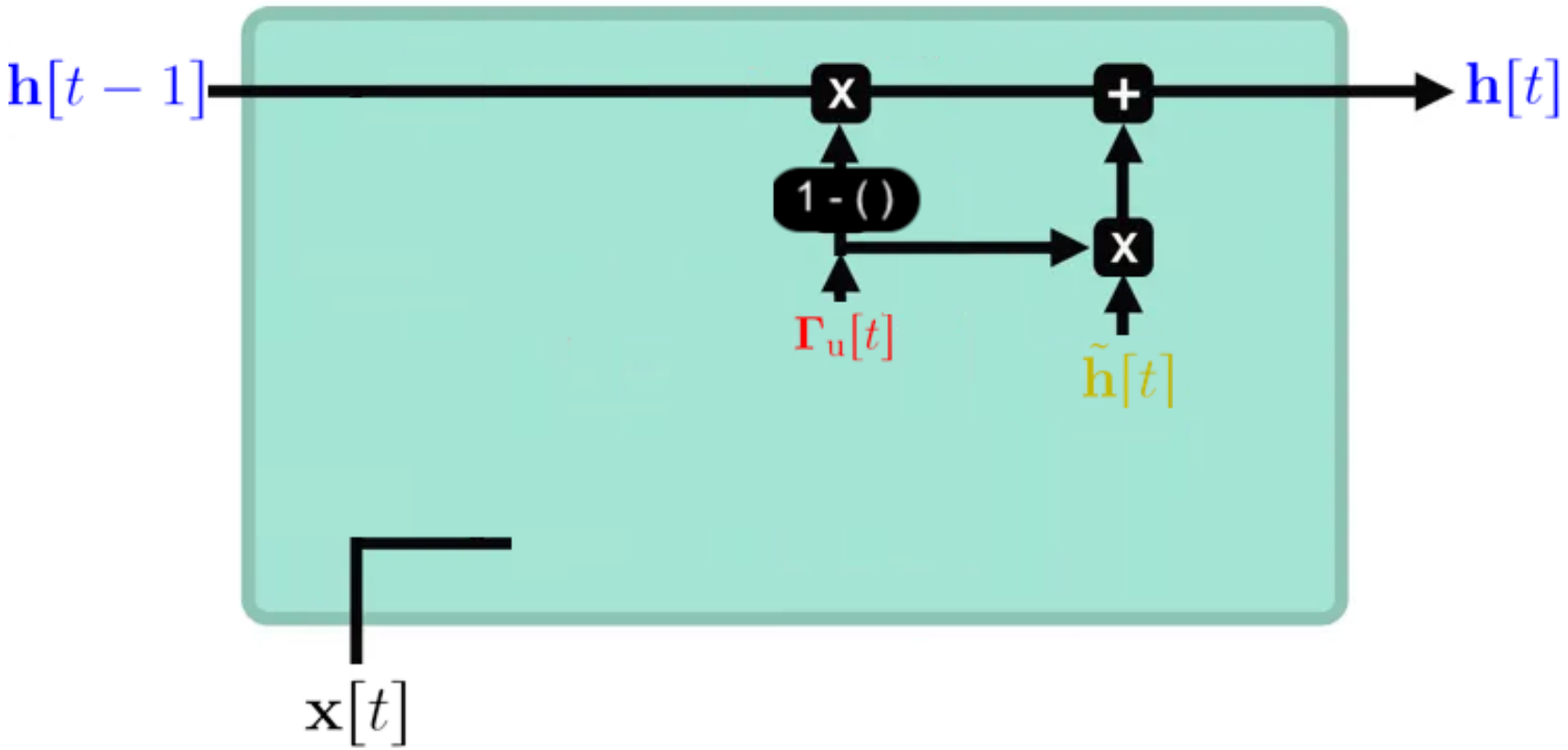
Practical Gated Architectures: *GRU*

Compute actual memory $\tilde{\mathbf{h}}[t] = f(\mathbf{W}_1\mathbf{x}[t] + \mathbf{W}_\mathrm{m}\boldsymbol{\Gamma}_\mathrm{r}[t] \odot \mathbf{h}[t-1])$



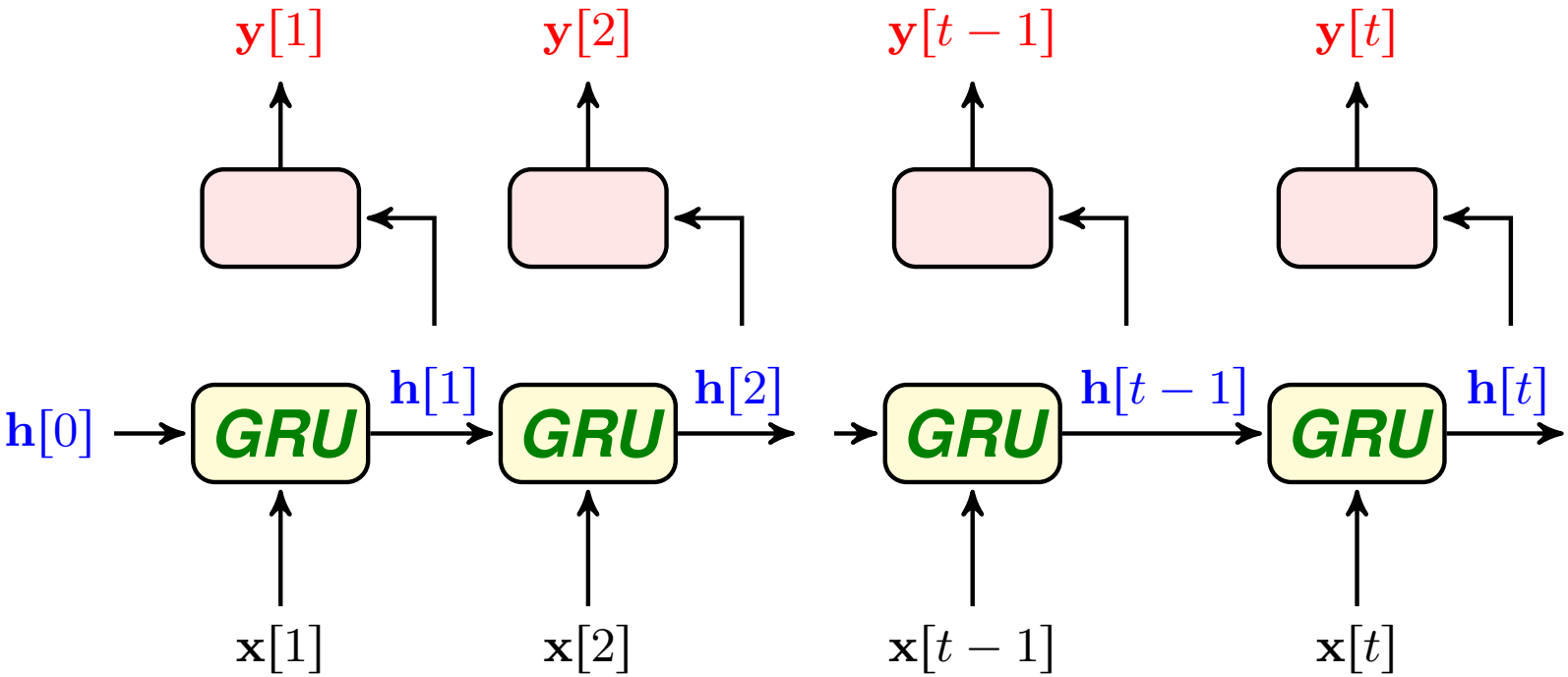
Practical Gated Architectures: *GRU*

Update hidden state as $\mathbf{h}[t] = (1 - \mathbf{\Gamma}_{\mathrm{u}}[t]) \odot \mathbf{h}[t-1] + \mathbf{\Gamma}_{\mathrm{u}}[t] \odot \tilde{\mathbf{h}}[t]$



GRU: *Forward Pass*

Starting from an initial state: GRU applies the first 5 steps each time



GRU: *Backward Pass*

Say we finished forward pass at time t. We now want to find $\nabla_{\mathbf{W}} \hat{R}$ for some $\mathbf{W}$ that is inside GRU, e.g., $\mathbf{W}_{\mathrm{u,m}}$: we start backpropagating from $\nabla_{\mathbf{y}[t]} \hat{R}$

$$\nabla_{\mathbf{W}} \hat{R} = \nabla_{\mathbf{y}[t]} \hat{R} \circ \nabla_{\mathbf{W}} \mathbf{y}[t]$$

1. *We know* $\mathbf{y}[t] = f_{\mathrm{out}}(\mathbf{W}_2 \mathbf{h}[t])$

$$\nabla_{\mathbf{W}} \mathbf{y}[t] = \nabla_{\mathbf{h}[t]} \mathbf{y}[t] \circ \nabla_{\mathbf{W}} \mathbf{h}[t]$$

2. *We know that* $\mathbf{h}[t] = (1 - \mathbf{\Gamma}_{\mathrm{u}}[t]) \odot \mathbf{h}[t-1] + \mathbf{\Gamma}_{\mathrm{u}}[t] \odot \tilde{\mathbf{h}}[t]$

$$\begin{aligned}\nabla_{\mathbf{W}} \mathbf{h}[t] =& \nabla_{\mathbf{\Gamma}_{\mathrm{u}}[t]} \mathbf{h}[t] \circ \nabla_{\mathbf{W}} \mathbf{\Gamma}_{\mathrm{u}}[t] + \nabla_{\mathbf{h}[t-1]} \mathbf{h}[t] \circ \nabla_{\mathbf{W}} \mathbf{h}[t-1] \\ &+ \nabla_{\tilde{\mathbf{h}}[t]} \mathbf{h}[t] \circ \nabla_{\mathbf{W}} \tilde{\mathbf{h}}[t]\end{aligned}$$

3. ...

Practical Gated Architectures: *Long Short-Term Memory*

Long Short-Term Memory (LSTM)

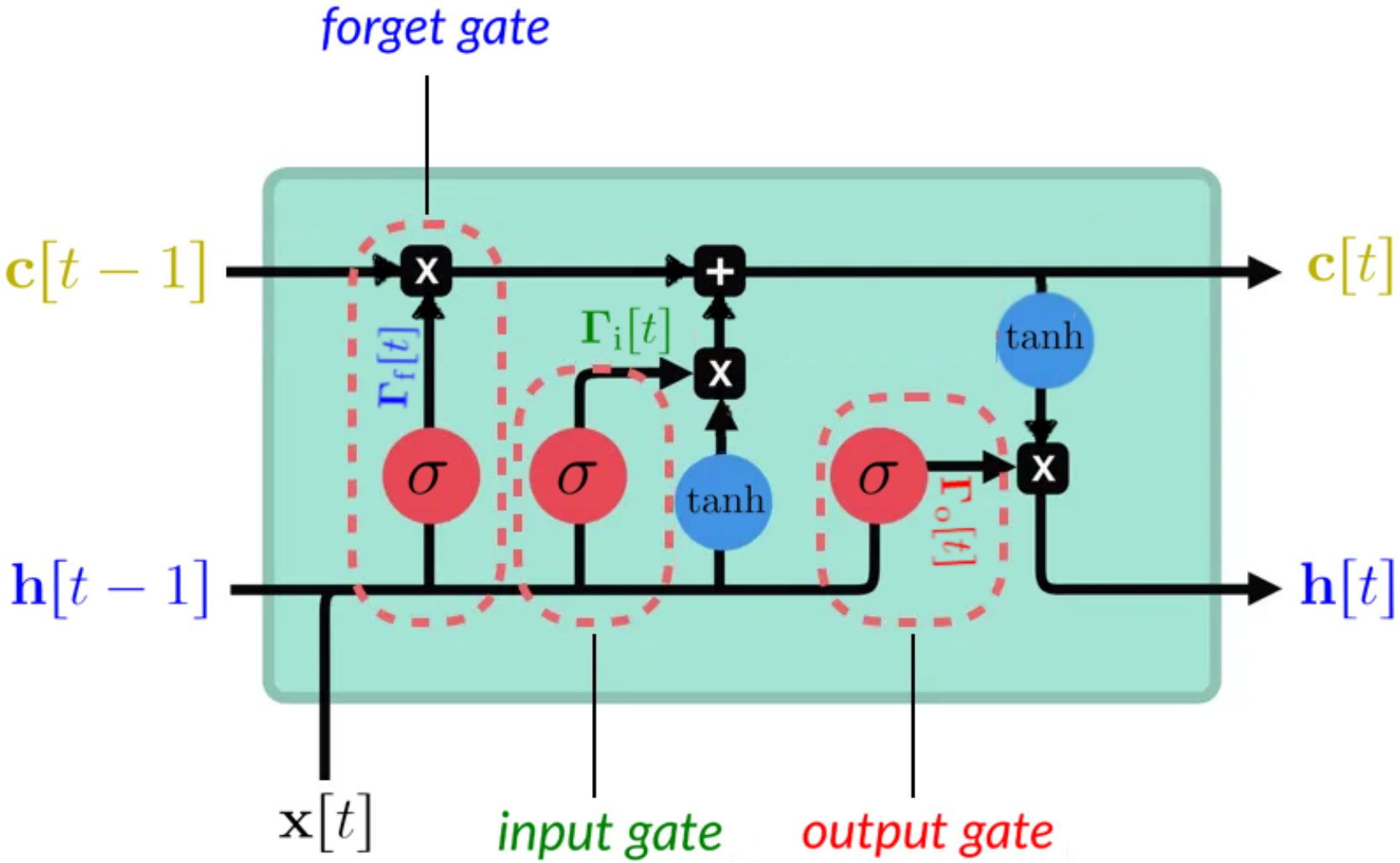
Say we set activation to $f(\cdot)$ ⇝ we usually set it to $\tanh(\cdot)$

1. *Start with initial hidden state and cell state*
2. *Compute forget gate* $\mathbf{\Gamma}_{\mathrm{f}}[t] = \sigma\left(\mathbf{W}_{\mathrm{f,in}}\mathbf{x}[t] + \mathbf{W}_{\mathrm{f,m}}\mathbf{h}[t-1]\right)$
3. *Compute input gate* $\mathbf{\Gamma}_{\mathrm{i}}[t] = \sigma\left(\mathbf{W}_{\mathrm{i,in}}\mathbf{x}[t] + \mathbf{W}_{\mathrm{i,m}}\mathbf{h}[t-1]\right)$
4. *Compute output gate* $\mathbf{\Gamma}_{\mathrm{o}}[t] = \sigma\left(\mathbf{W}_{\mathrm{o,in}}\mathbf{x}[t] + \mathbf{W}_{\mathrm{o,m}}\mathbf{h}[t-1]\right)$
5. *Compute actual cell state* $\tilde{\mathbf{c}}[t] = f(\mathbf{W}_{1}\mathbf{x}[t] + \mathbf{W}_{\mathrm{m}}\mathbf{h}[t-1])$
6. *Update cell state as* $\mathbf{c}[t] = \mathbf{\Gamma}_{\mathrm{f}}[t]\mathbf{c}[t-1] + \mathbf{\Gamma}_{\mathrm{i}}[t] \odot \tilde{\mathbf{c}}[t]$
7. *Update hidden state as* $\mathbf{h}[t] = \mathbf{\Gamma}_{\mathrm{o}}[t] \odot f\left(\mathbf{c}[t]\right)$
8. *Compute output* $\mathbf{y}[t] = f_{\mathrm{out}}(\mathbf{W}_{2}\mathbf{h}[t])$ ⇝ f_{out} and f could be different

Or we could give $\mathbf{h}[t]$ *to a new layer: for instance a new LSTM whose input is* $\mathbf{h}[t]$ *and has its own hidden and cell states*

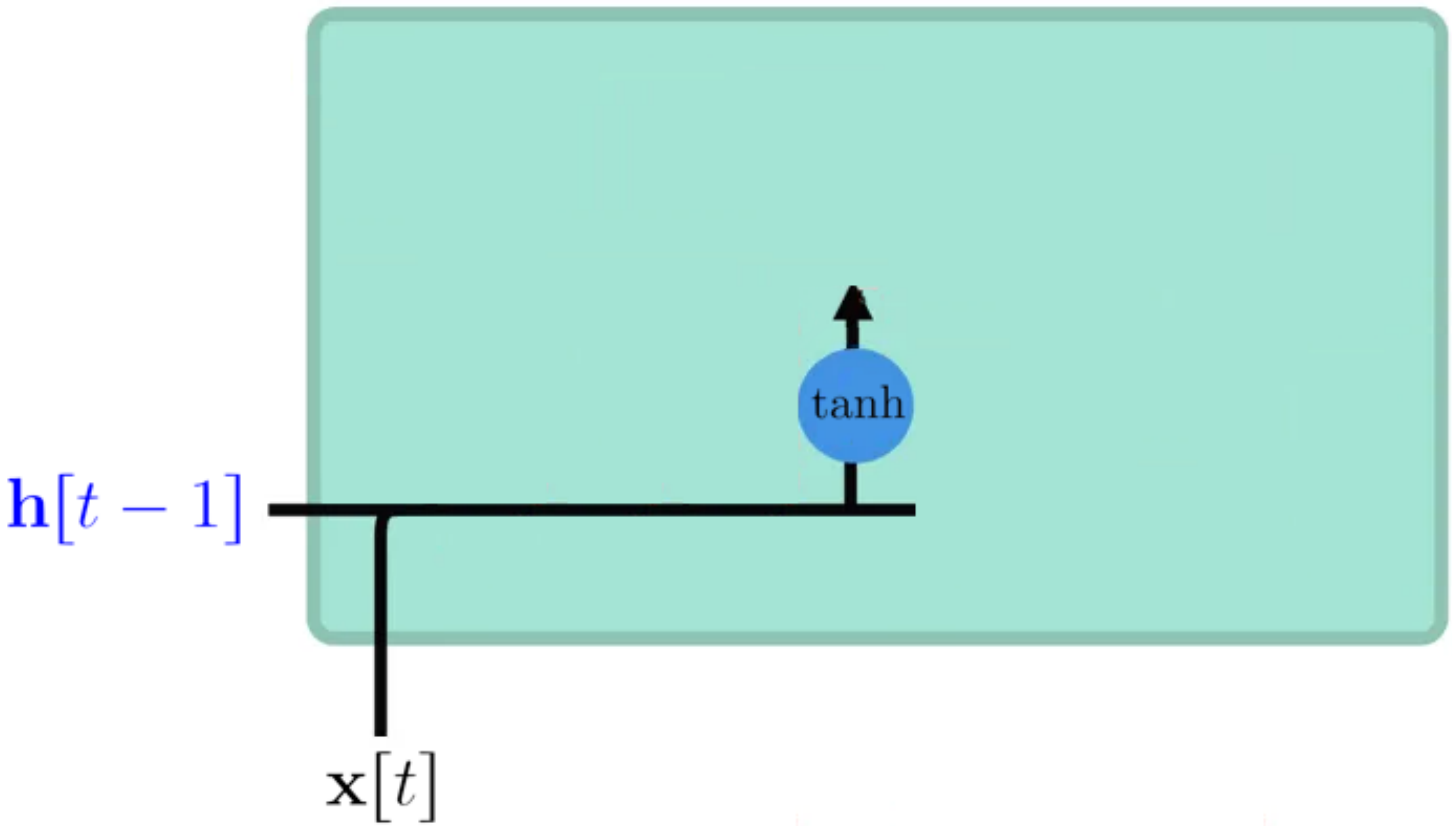
Practical Gated Architectures: *LSTM*

This is how inside an LSTM unit looks like



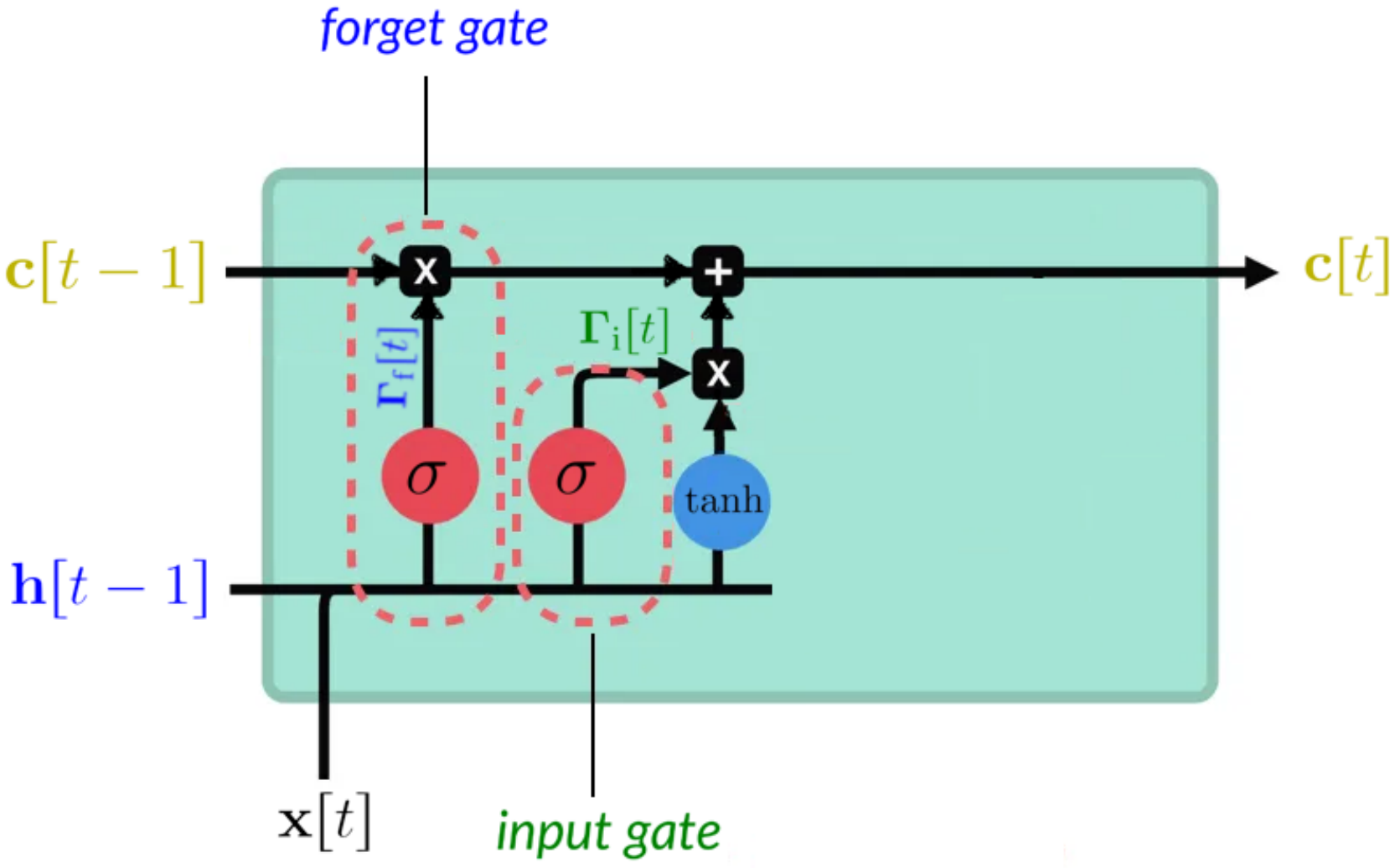
Practical Gated Architectures: *LSTM*

Actual cell state $\tilde{\mathbf{c}}[t] = f(\mathbf{W}_1\mathbf{x}[t] + \mathbf{W}_m\mathbf{h}[t-1])$



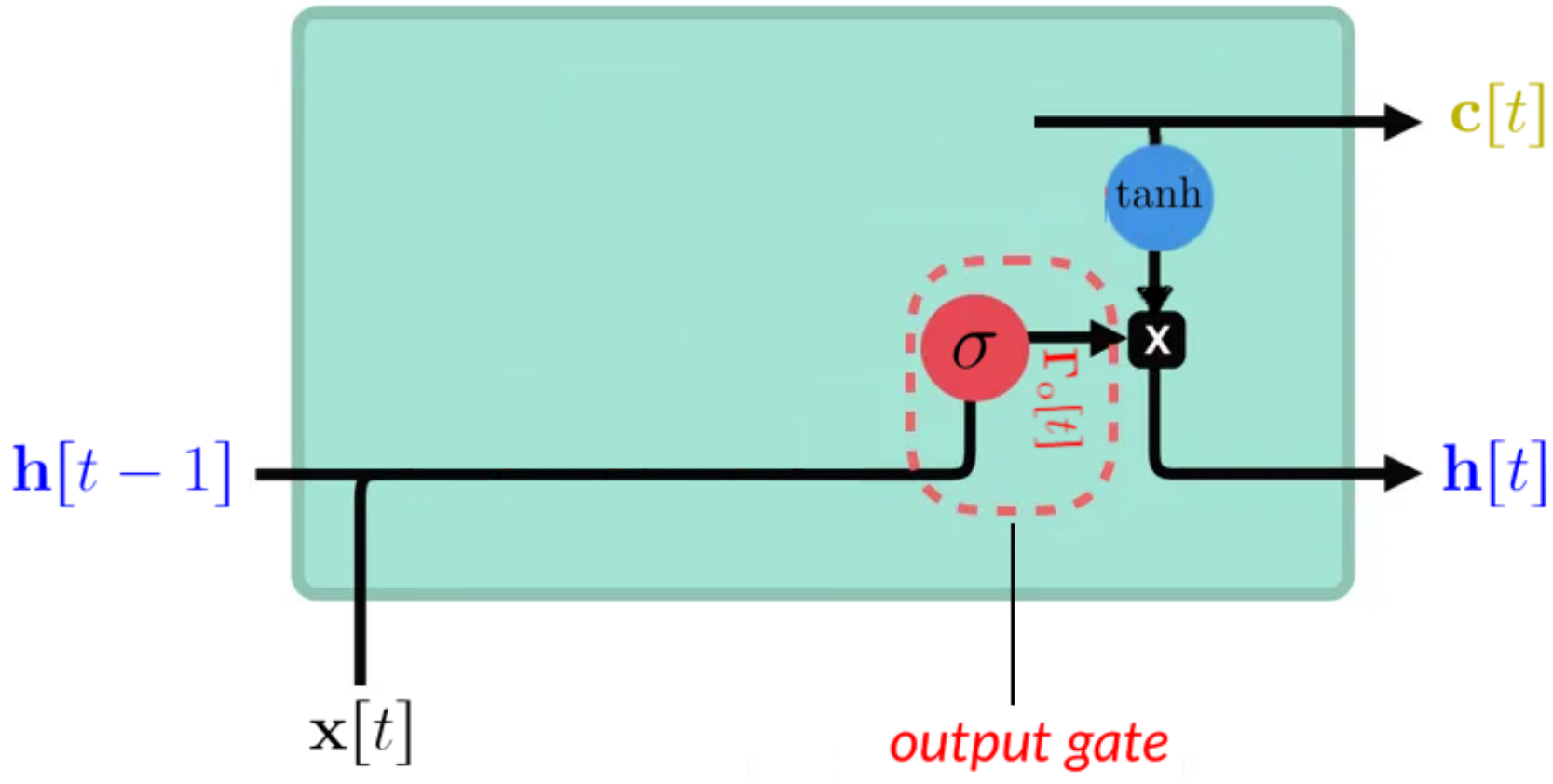
Practical Gated Architectures: *LSTM*

We use forget gate and update gate to update cell state



Practical Gated Architectures: *LSTM*

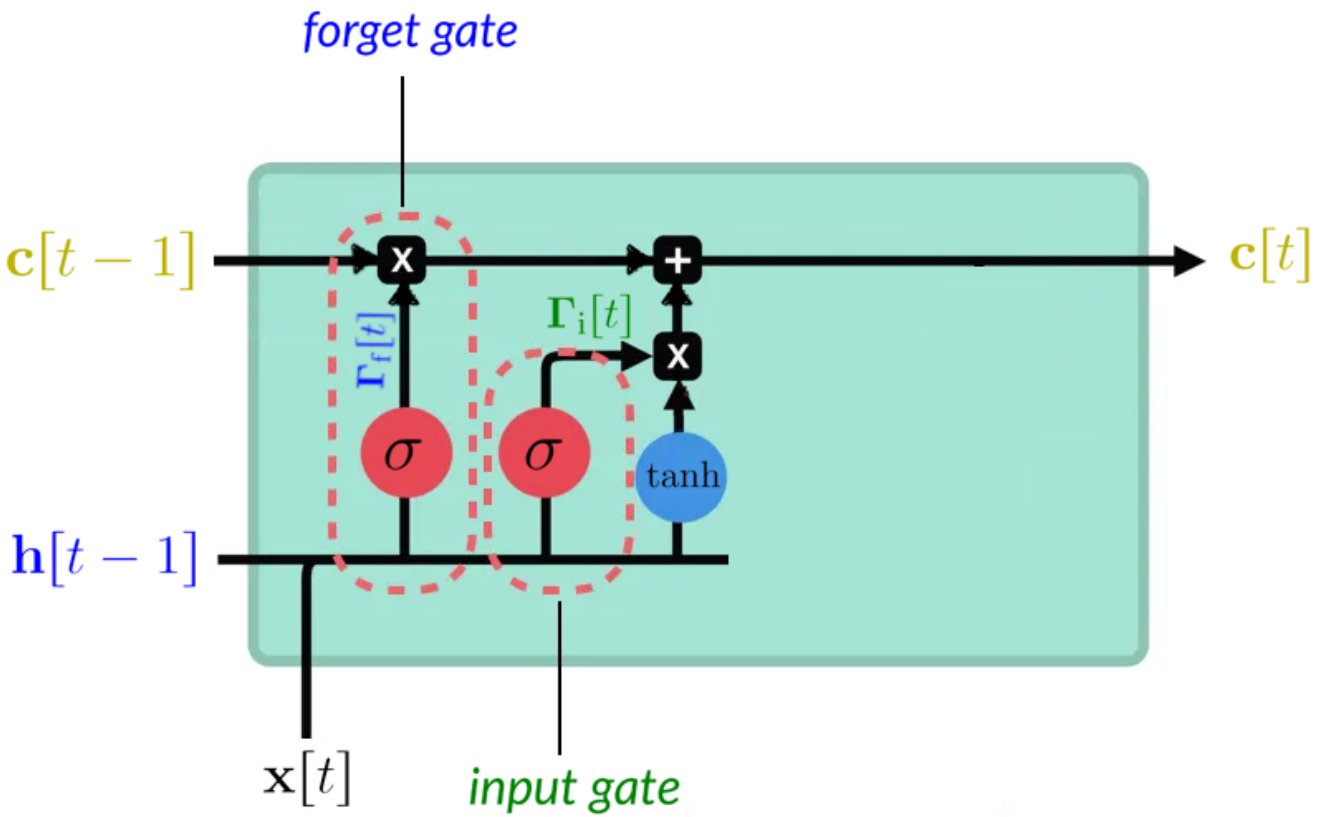
We use output gate to control fellow of memory to the hidden state



Practical Gated Architectures: *LSTM*

Intuitively, the gates in LSTM impact *the flow of information as follows*

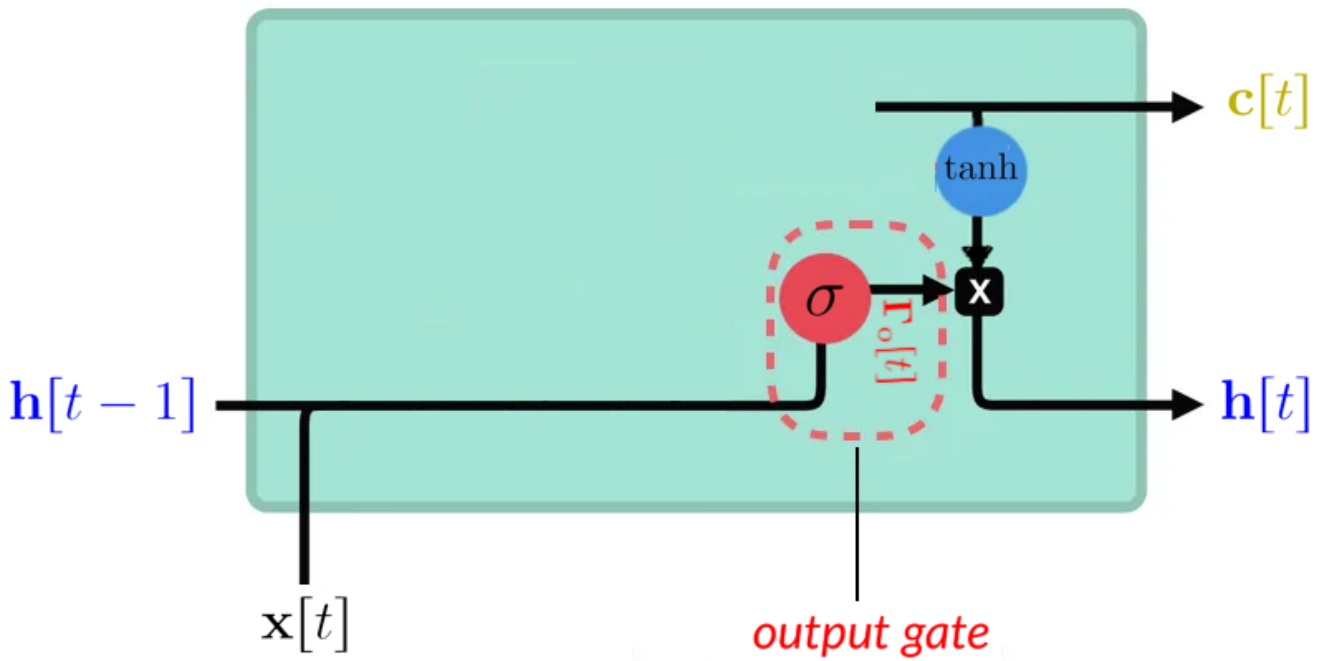
- *Forget gate controls how much we forget from last state*
- *Input gate controls how much we remember from new cell state*



Practical Gated Architectures: *LSTM*

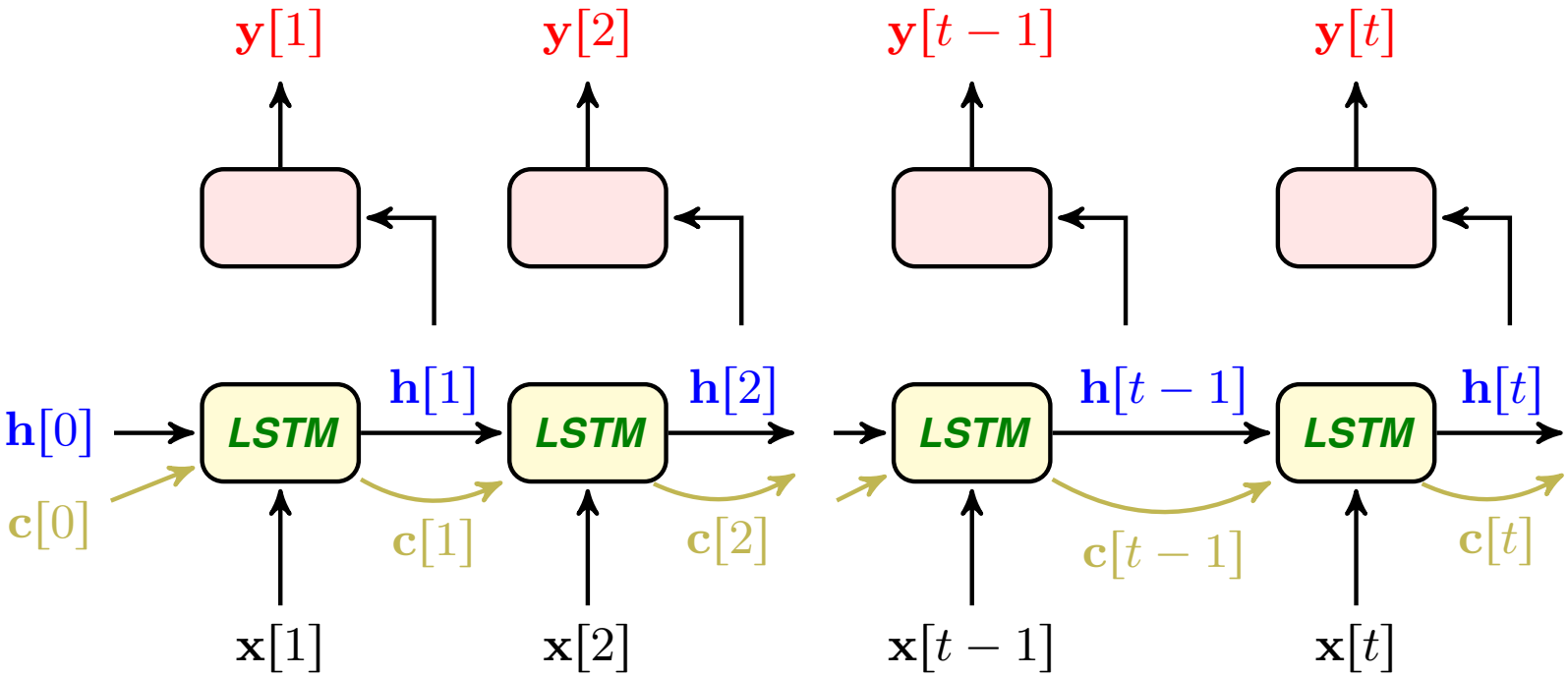
Intuitively, the gates in LSTM impact *the flow of information as follows*

- *Output gate controls how much we let from updated state to go out*



LSTM: *Forward Pass*

Starting from initial hidden and cell state: LSTM passes forward as



Pay Attention

Note that unlike other architectures, LSTM does not keep all memory inside hidden state but it carries it also in cell state. This state is only for memory and is not directly used by higher layers, e.g., output layer of the NN

LSTM: *Backward Pass*

Say we finished forward pass at time t. We now want to find $\nabla_{\mathbf{W}} \hat{R}$ for some $\mathbf{W}$ that is inside LSTM, e.g., $\mathbf{W}_{\mathrm{i,m}}$: we start backpropagating from $\nabla_{\mathbf{y}[t]} \hat{R}$

$$\nabla_{\mathbf{W}} \hat{R} = \nabla_{\mathbf{y}[t]} \hat{R} \circ \nabla_{\mathbf{W}} \mathbf{y}[t]$$

1. *We know* $\mathbf{y}[t] = f_{\mathrm{out}}(\mathbf{W}_2 \mathbf{h}[t])$

$$\nabla_{\mathbf{W}} \mathbf{y}[t] = \nabla_{\mathbf{h}[t]} \mathbf{y}[t] \circ \nabla_{\mathbf{W}} \mathbf{h}[t]$$

2. ...

Suggestion

Try writing it once to see the impact of gates!



UNIVERSITY OF
TORONTO

Further Read

Further Read

Read More on These Topics

- Goodfellow, et al., *Deep Learning*
  - ↳ Chapter 10
- Bishop & Bishop, *Deep Learning: Foundations and Concepts*
  - ↳ Chapter 11

Nice to Check

1. *Backpropagation through time: What it does and how to do it*
   - ↳ P. Werbos (2002)
2. *Long short-term memory*
   - ↳ Hochreiter and J. Schmidhuber (1997)