COMP 3361 Natural Language Processing

Lecture 8: RNNs and LSTM

Spring 2024

Many materials from CSE447@UW (Taylor Sorensen) and CS224n@Stanford with special thanks!

Announcements

- OpenAI’s Sora was out! AI is evolving so quickly!!!
- Assignment 1 due date has just passed.
- Assignment 2 will be out by today.
  - Due: Mar 19
  - Make sure you check out the recorded tutorial on PyTorch and huggingface

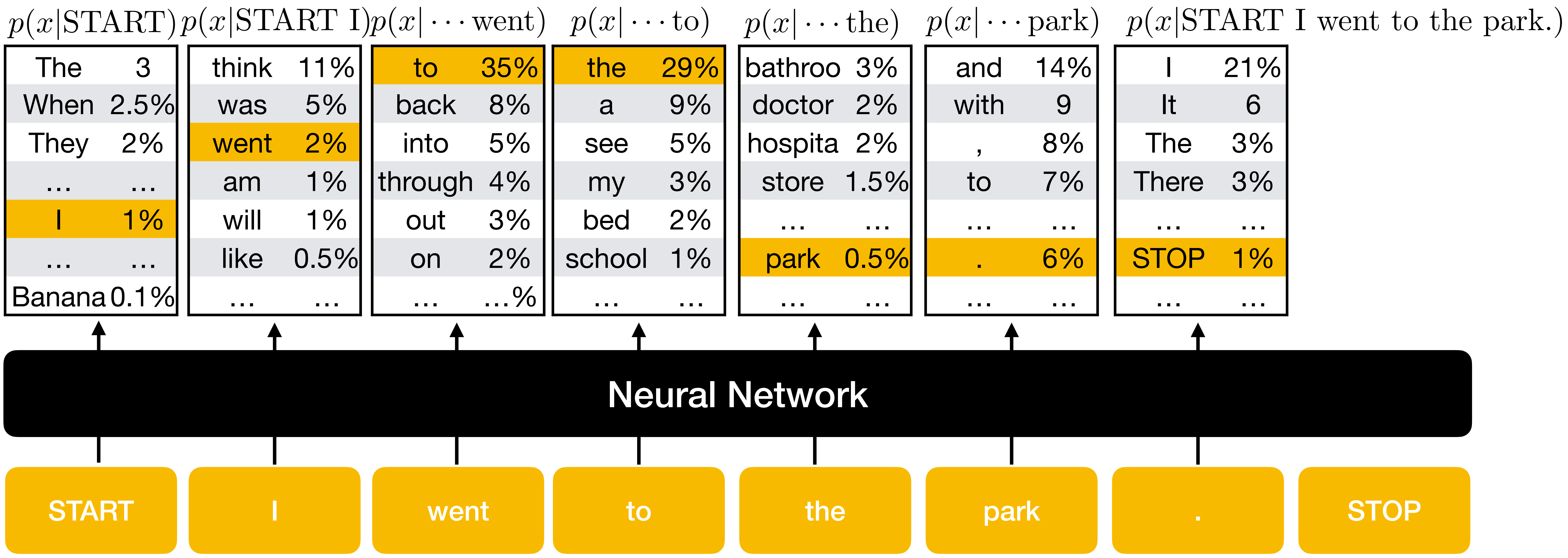
Lecture plan

- Recurrent Neural Networks (RNNs)
- Long Short-Term Memory (LSTM)

Language modeling with neural networks

Inputs/Outputs

- **Input:** sequences of words (or tokens)
- **Output:** probability distribution over the next word (token)



Neural language models

How do neural networks encode text with various lengths?

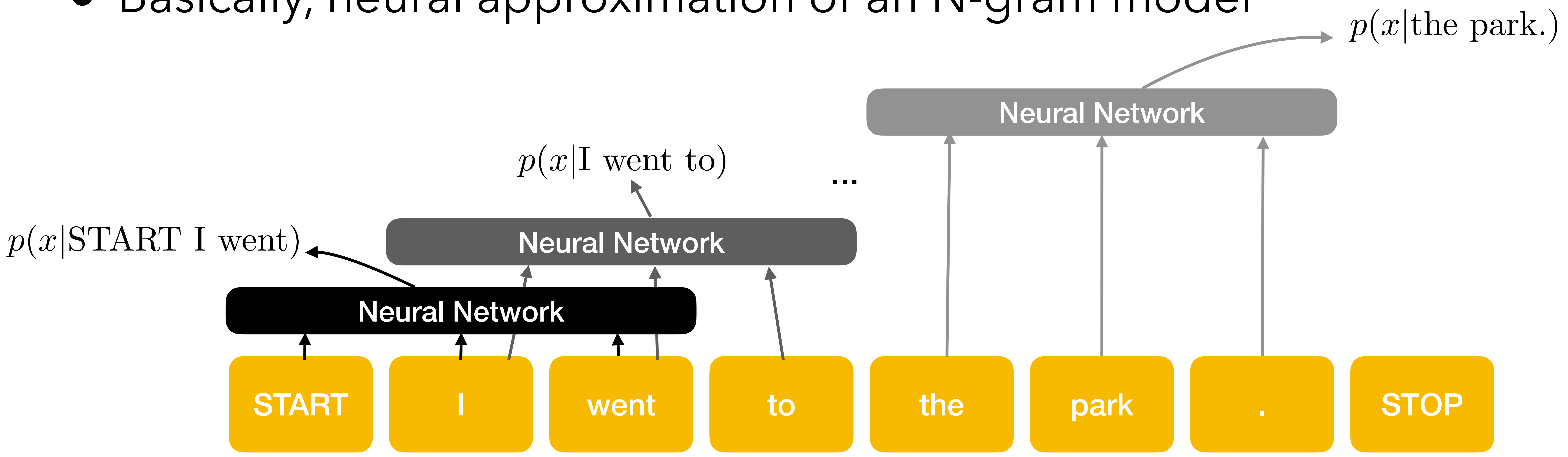
- Don’t neural networks need a fixed-size vector as input? And isn’t text variable length?

Sliding window

Don't neural networks need a fixed-size vector as input? And isn't text variable length?

Idea 1: Sliding window of size N

- Cannot look more than N words back
- Basically, neural approximation of an N-gram model

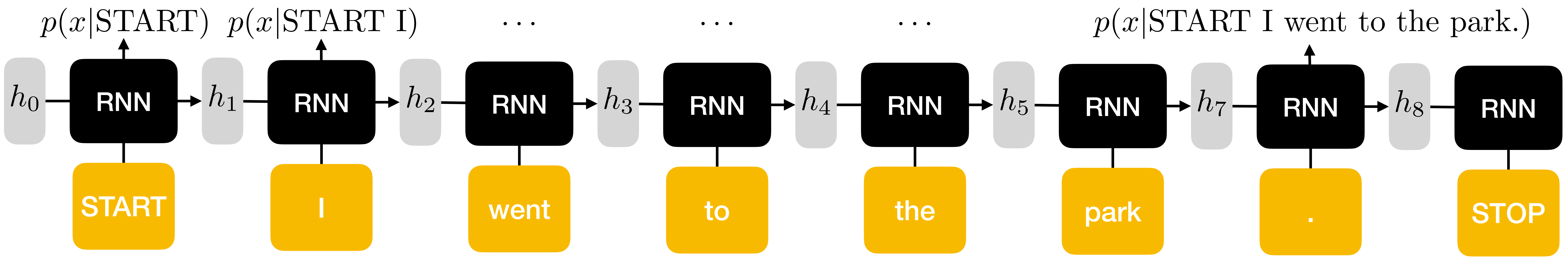


Recurrent neural networks

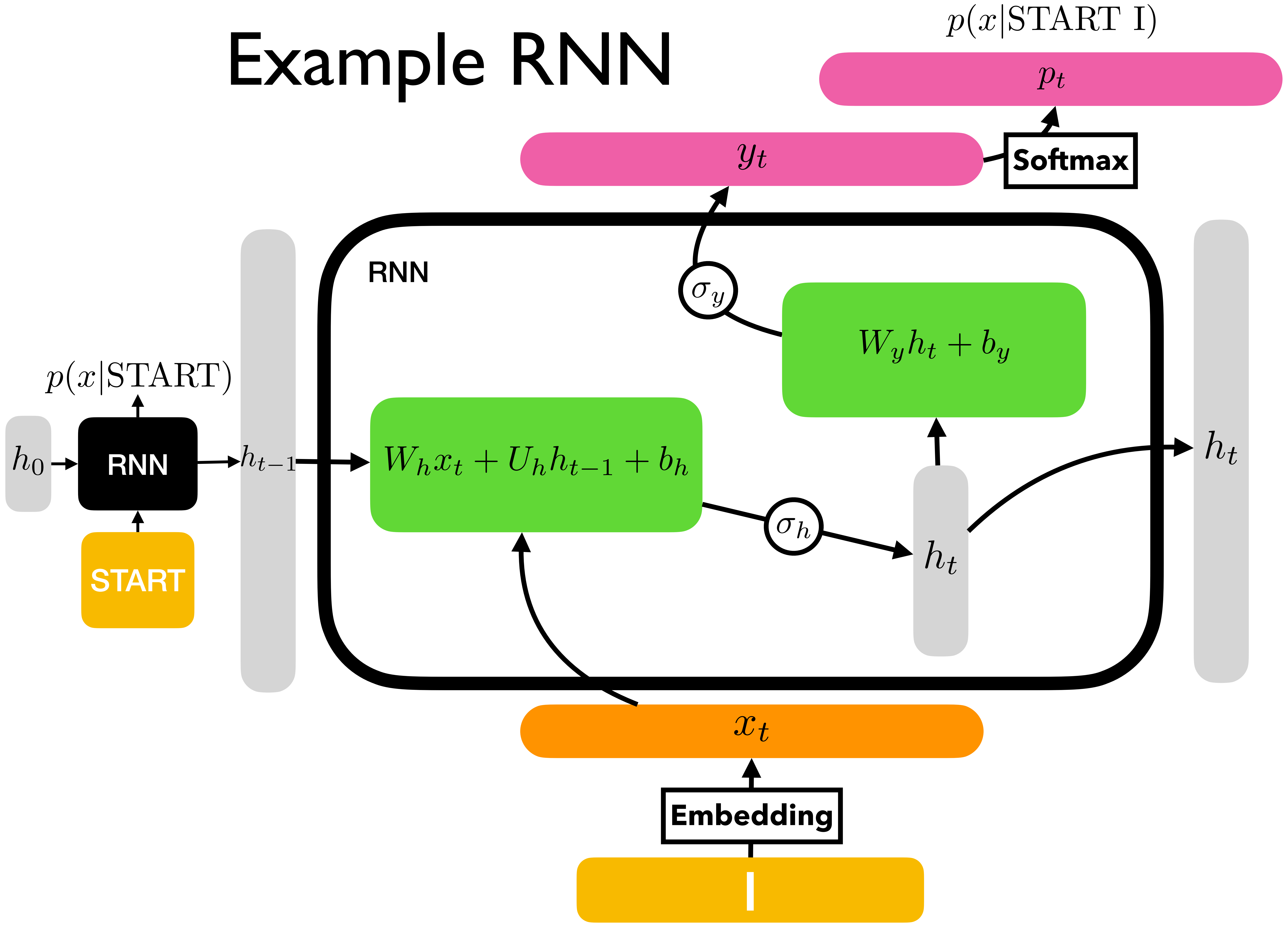
Idea 2: Recurrent Neural Networks (RNNs)

Essential components:

- One network is applied recursively to the sequence
- *Inputs:* previous hidden state h_{t-1}, observation x_t
- *Outputs:* next hidden state h_t, (optionally) output y_t
- Memory about history is passed through hidden states



Example RNN



Variables:

x_t: input (embedding) vector

y_t: output vector (logits)

p_t: probability over tokens

h_{t-1}: previous hidden vector

h_t: next hidden vector

σ_h: activation function for hidden state

σ_y: output activation function

Equations:

$$h_t := \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$

$$y_t := \sigma_y(W_y h_t + b_y)$$

$$p_{t_i} = \frac{\exp(y_{t_i})}{\sum_{i=j}^{d} \exp(y_{t_j})}$$

Example RNN

What are trainable parameters θ?

output distribution

$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$

hidden states

$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$

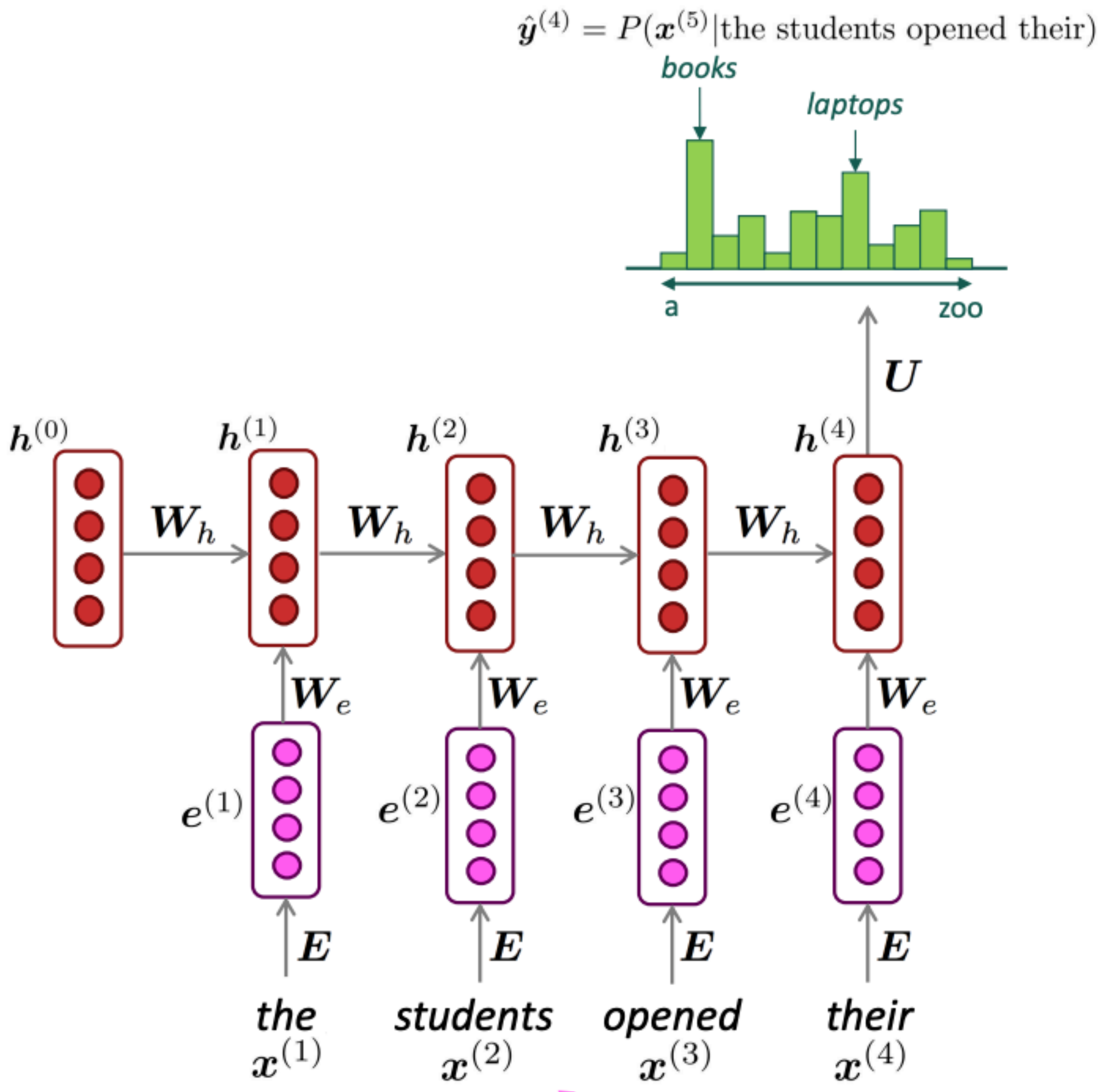
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$

words / one-hot vectors

$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$



***Note**: this input sequence could be much longer now!*

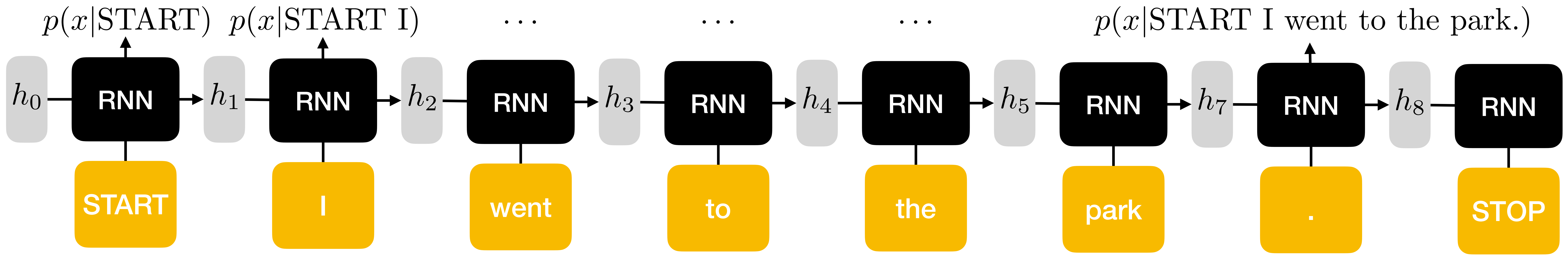
Recurrent neural networks

- How can information from time an earlier state (e.g., time 0) pass to a later state (time t?)
  - Through the hidden states!
  - Even though they are continuous vectors, can represent very rich information (up to the entire history from the beginning)

$$P(w_1, w_2, \ldots, w_n) = P(w_1) \times P(w_2 \mid w_1) \times P(w_3 \mid w_1, w_2) \times \ldots \times P(w_n \mid w_1, w_2, \ldots, w_{n-1})$$

$$= P(w_1 \mid \mathbf{h}_0) \times P(w_2 \mid \mathbf{h}_1) \times P(w_3 \mid \mathbf{h}_2) \times \ldots \times P(w_n \mid \mathbf{h}_{n-1})$$

No Markov assumption here!



Training procedure

E.g., if you wanted to train on “<START>I went to the park.<STOP>”…

1. Input/Output Pairs

$\mathcal{D}$

x (input)	y (output)
START	I
START I	went
START I went	to
START I went to	the
START I went to the	park
START I went to the park	.
START I went to the park.	STOP

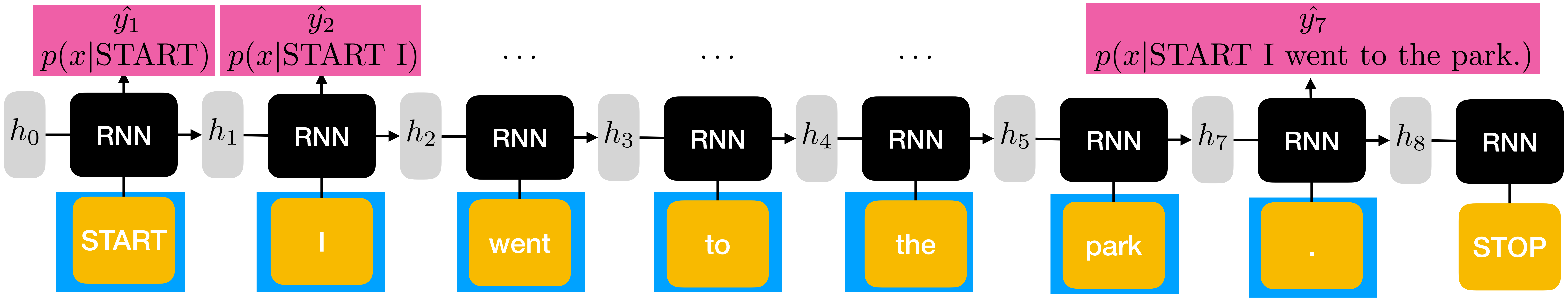
Training procedure

1. Input/Output Pairs

$\mathcal{D}$

x (input)	y (output)
START	I
START I	went
START I went	to
START I went to	the
START I went to the	park
START I went to the park	.
START I went to the park.	STOP

2. Run model on (batch of) x's from data $\mathcal{D}$ to get probability distributions $\hat{y}$ (running softmax at end to ensure valid probability distribution)



Training procedure

2. Run model on (batch of) x's from data $\mathcal{D}$ to get probability distributions $\hat{y}$

3. Calculate loss compared to true y's (Cross Entropy Loss)

$$L_{\text{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

$\hat{y}_1 = \begin{array}{r} p(\text{STOP}|\text{START}) \\ p(\text{The}|\text{START}) \\ p(\text{I}|\text{START}) \\ \\ \\ p(\text{apple}|\text{START}) \end{array} \begin{bmatrix} .01 \\ .03 \\ .1 \\ .001 \\ \vdots \\ .002 \end{bmatrix}$
$y_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$
L_{CE}
$\hat{y}_7 = \begin{bmatrix} .2 \\ .03 \\ .12 \\ .01 \\ \vdots \\ .001 \end{bmatrix}$
$y_7 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$
$\hat{y}_1$
$p(x|\text{START})$
$\hat{y}_7$
$p(x|\text{START I went to the park.})$
h_0 h_1 h_2 h_3 h_4 h_5 h_7 h_8
RNN
START
I
went
to
the
park
.
STOP

Training procedure $L_{\text{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$

3. Calculate loss compared to true y's (Cross Entropy Loss)

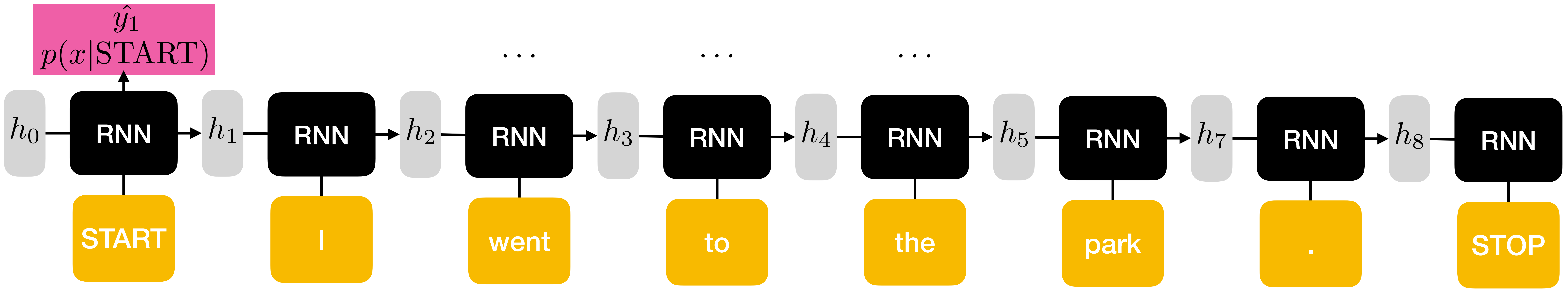
$\hat{y}_1 = \begin{array}{r} p(\text{STOP}|\text{START}) \\ p(\text{The}|\text{START}) \\ p(\text{I}|\text{START}) \\ \\ \\ p(\text{apple}|\text{START}) \end{array} \begin{bmatrix} .01 \\ .03 \\ .1 \\ .001 \\ \vdots \\ .002 \end{bmatrix}$
$y_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$
(Actual observed word)
L_{CE}
$\hat{y}_1$
$p(x|\text{START})$
$\dots$
h_0
RNN
h_1
h_2
h_3
h_4
h_5
h_7
h_8
START
I
went
to
the
park
.
STOP

$$L_{\text{CE}}(y_1, \hat{y}_1) = -0 * \log(.01) - 0 * \log(.03) - 1 * -\log(.1) - \cdots - 0 * \log(.002)$$
$$= -\log(.1) = -\log(p(\text{I}|\text{START}))$$

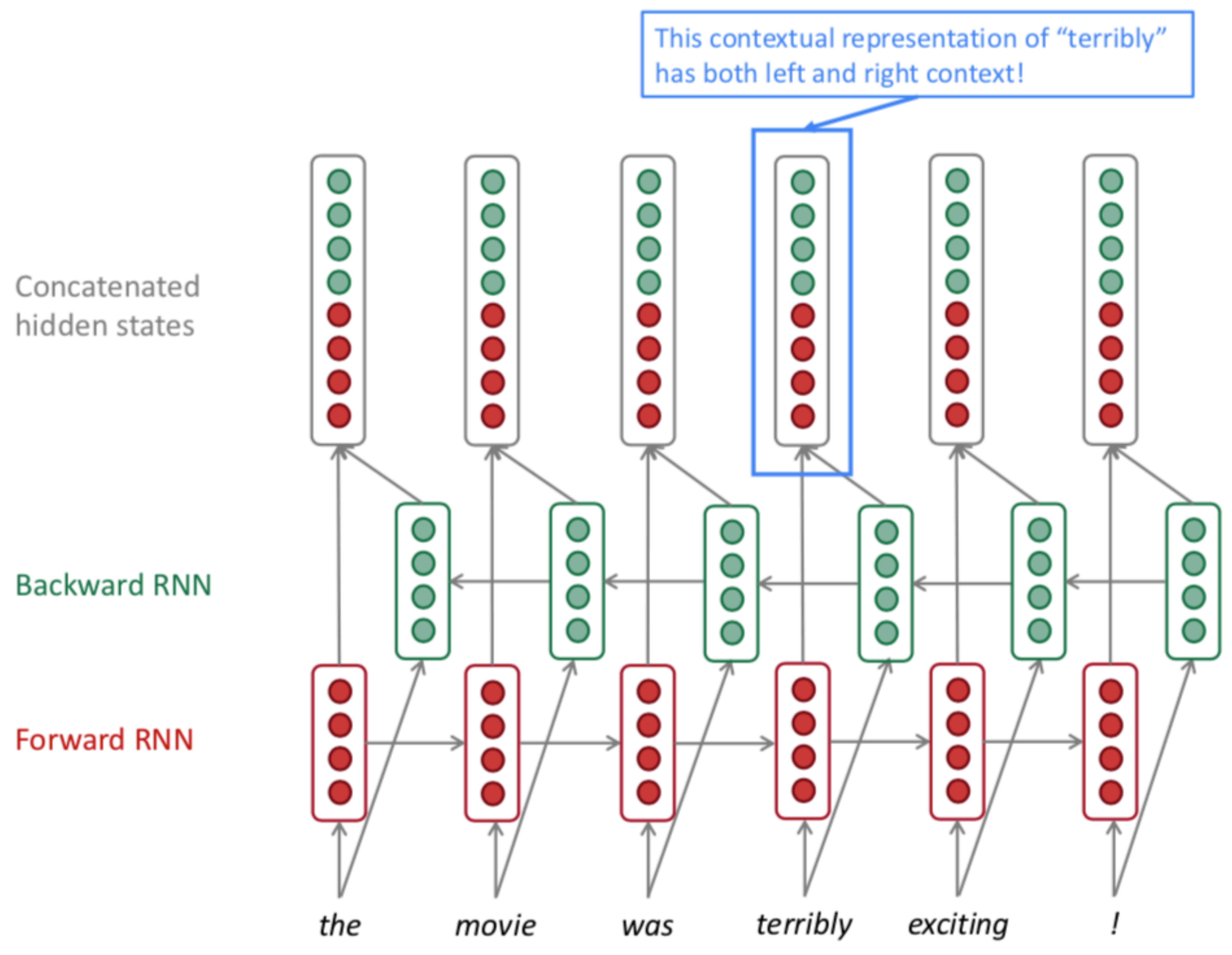
Training procedure - gradient descent step

1. Get training x-y pairs from batch
2. Run model to get probability distributions over $\hat{y}$
3. Calculate loss compared to true y
4. Backpropagate to get the gradient
5. Take a step of gradient descent

$$\theta^{(i+1)} = \theta^{(i)} - \alpha * \frac{\partial L}{\partial \theta}(\theta^{(i)})$$



Bidirectional RNNs


This contextual representation of "terribly" has both left and right context!
Concatenated hidden states
Backward RNN
Forward RNN
the
movie
was
terribly
exciting
!

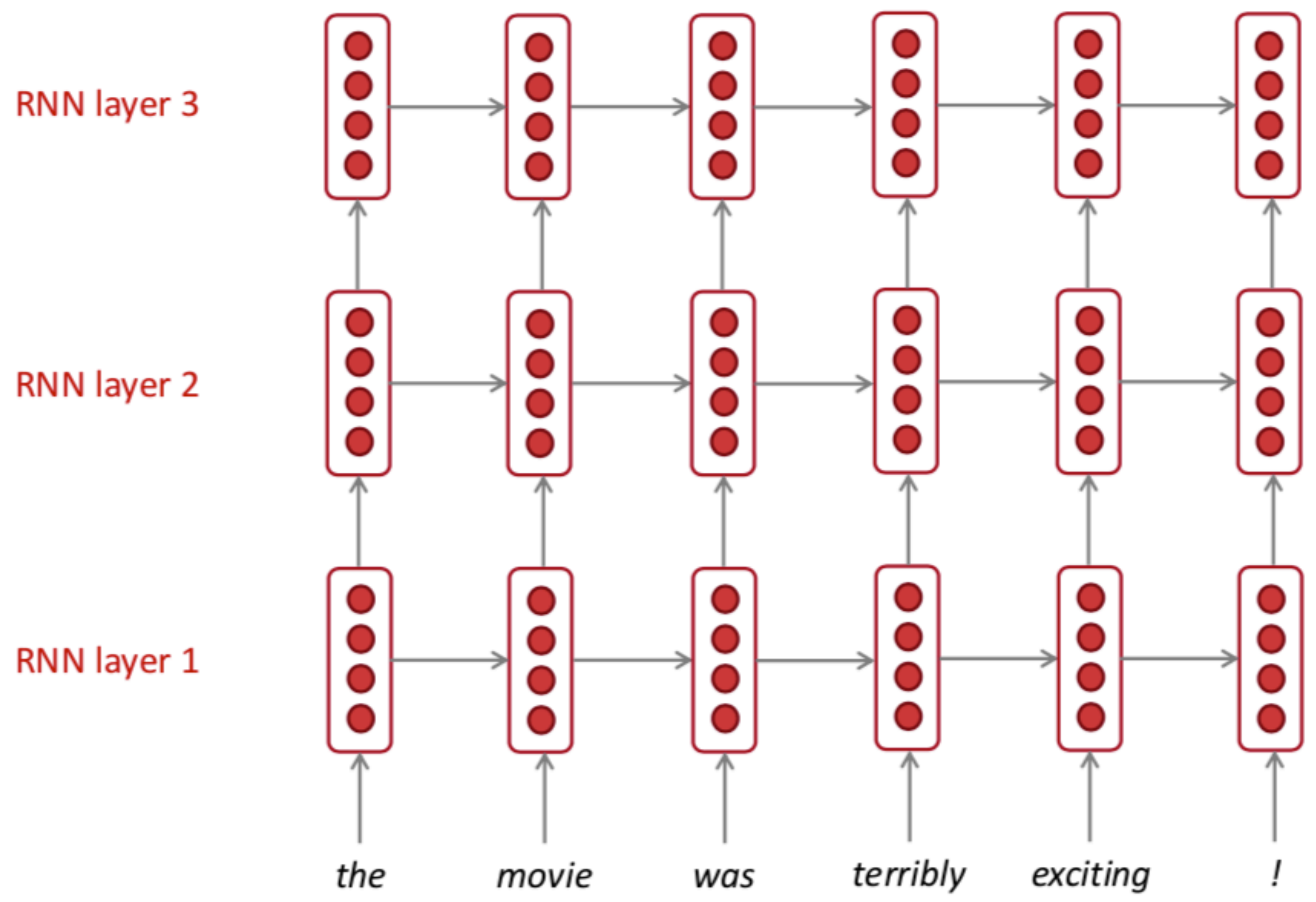
$$\boxed{\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^h}$$

$$\overrightarrow{\mathbf{h}}_t = f_1(\overrightarrow{\mathbf{h}}_{t-1}, \mathbf{x}_t), t = 1,2,\ldots n$$

$$\overleftarrow{\mathbf{h}}_t = f_2(\overleftarrow{\mathbf{h}}_{t+1}, \mathbf{x}_t), t = n, n-1,\ldots 1$$

$$\mathbf{h}_t = [\overleftarrow{\mathbf{h}}_t, \overrightarrow{\mathbf{h}}_t] \in \mathbb{R}^{2h}$$

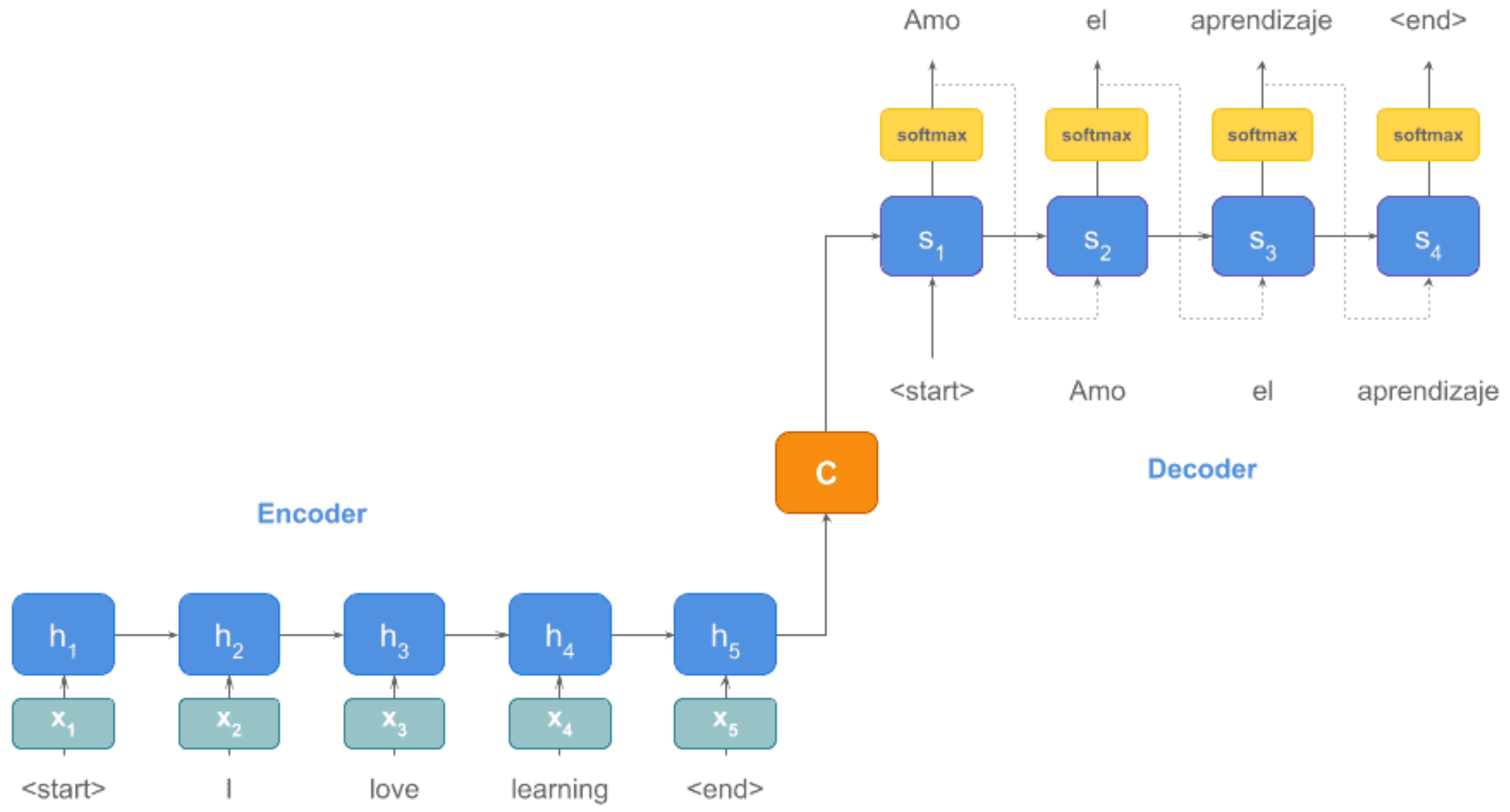
Multi-layer RNNs



The hidden states from RNN layer i are the inputs to RNN layer $i+1$

- In practice, using 2 to 4 layers is common (usually better than 1 layer)
- Transformer networks can be up to 24 layers with lots of skip-connections

RNN encoder-decoder for machine translation



https://www.baeldung.com/cs/nlp-encoder-decoder-models

RNNs - vanishing gradient problem

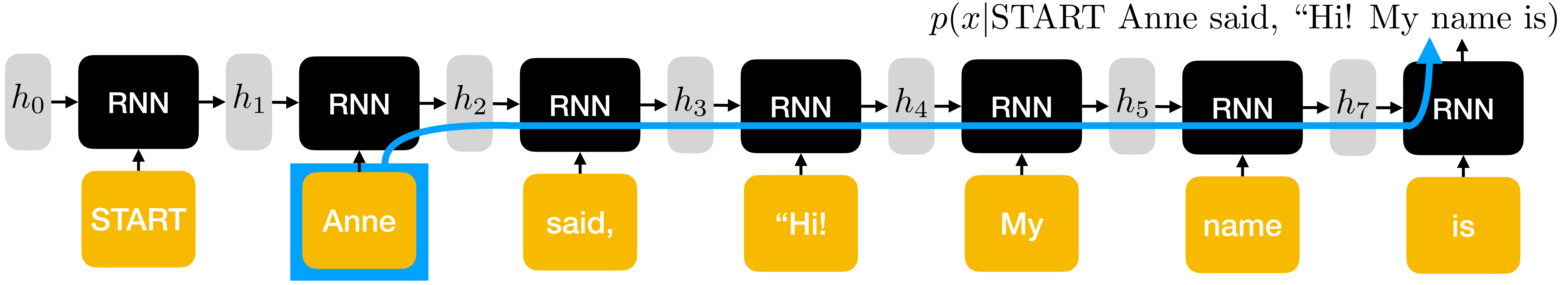
What word is likely to come next for this sequence?

Anne said, "Hi! My name is

RNNs - vanishing gradient problem

What word is likely to come next for this sequence?

Anne said, "Hi! My name is



- Need **relevant information** to flow across many time steps
- When we backpropagate, we want to allow the relevant information to flow

RNNs - vanishing gradient problem

h_0
RNN
h_1
RNN
h_2
RNN
h_3
RNN
h_4
RNN
h_5
RNN
h_7
RNN
START
Anne
said,
“Hi!
My
name
is
Backprop steps

$p(x|\text{START Anne said, “Hi! My name is})$

$$\delta_{\hat{y}} = \frac{\partial L}{\partial \hat{y}}$$

$$\delta_{h_8} = \delta_{\hat{y}} W_y^T \odot \sigma_y'(W_y h_8 + b_y)$$

$$\delta_{h_t} = \delta_{h_{t+1}} U_h^T \odot \sigma_h'(W_h x_{t+1} + U_h h_t + b_h) \quad \cdots \quad \delta_{h_7} = \delta_{h_8} U_h^T \odot \sigma_h'(W_h x_8 + U_h h_7 + b_h)$$

However, when we backprop, it involves multiplying a chain of computations from time t_7 to time t_1...

If any of the terms are close to zero, the whole gradient goes to zero (vanishes!)

The **vanishing gradient problem**

RNNs - vanishing gradient problem

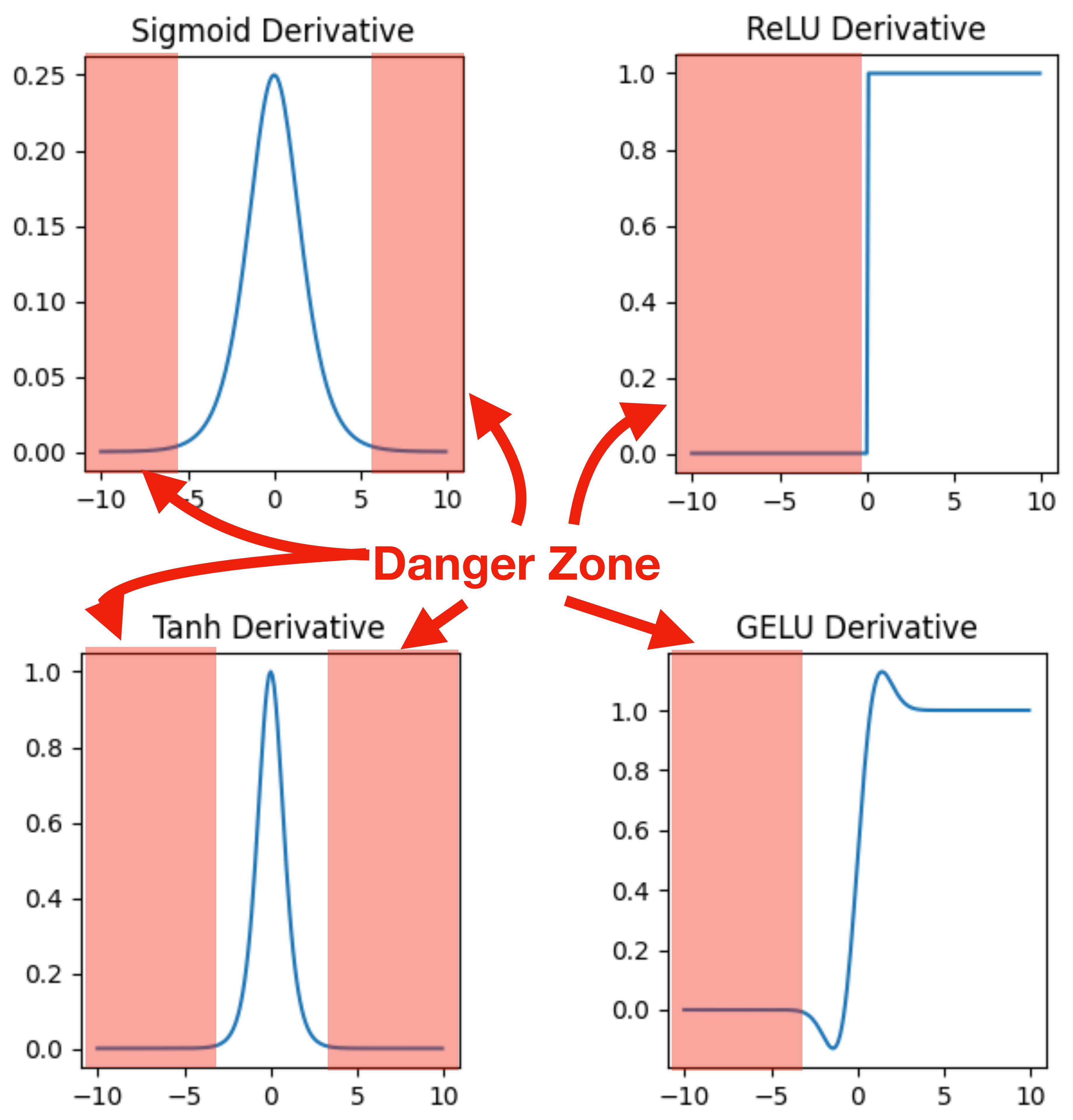
$$\delta_{h_t} = \delta_{h_{t+1}} U_h^T \odot \sigma_h'(W_h x_{t+1} + U_h h_t + b_h)$$

If any of the terms are close to zero, the whole gradient goes to zero (vanishes!)

The **vanishing gradient problem**

- This happens often for many activation functions… the gradient is close to zero when outputs get very large or small
- The more time steps back, the more chances for a vanishing gradient

Solution: **LSTMs!**



LSTMs

Idea 3: Long short-term memory network

<u>Essential components:</u>

- It is a recurrent neural network (RNN)
- Has modules to learn when to “remember”/“forget” information
- Allows gradients to flow more easily

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot \sigma_h(c_t)
\end{aligned}
$$

$x_t \in \mathbb{R}^d$: input vector to the LSTM unit

$f_t \in (0,1)^h$: forget gate’s activation vector

$i_t \in (0,1)^h$: input/update gate’s activation vector

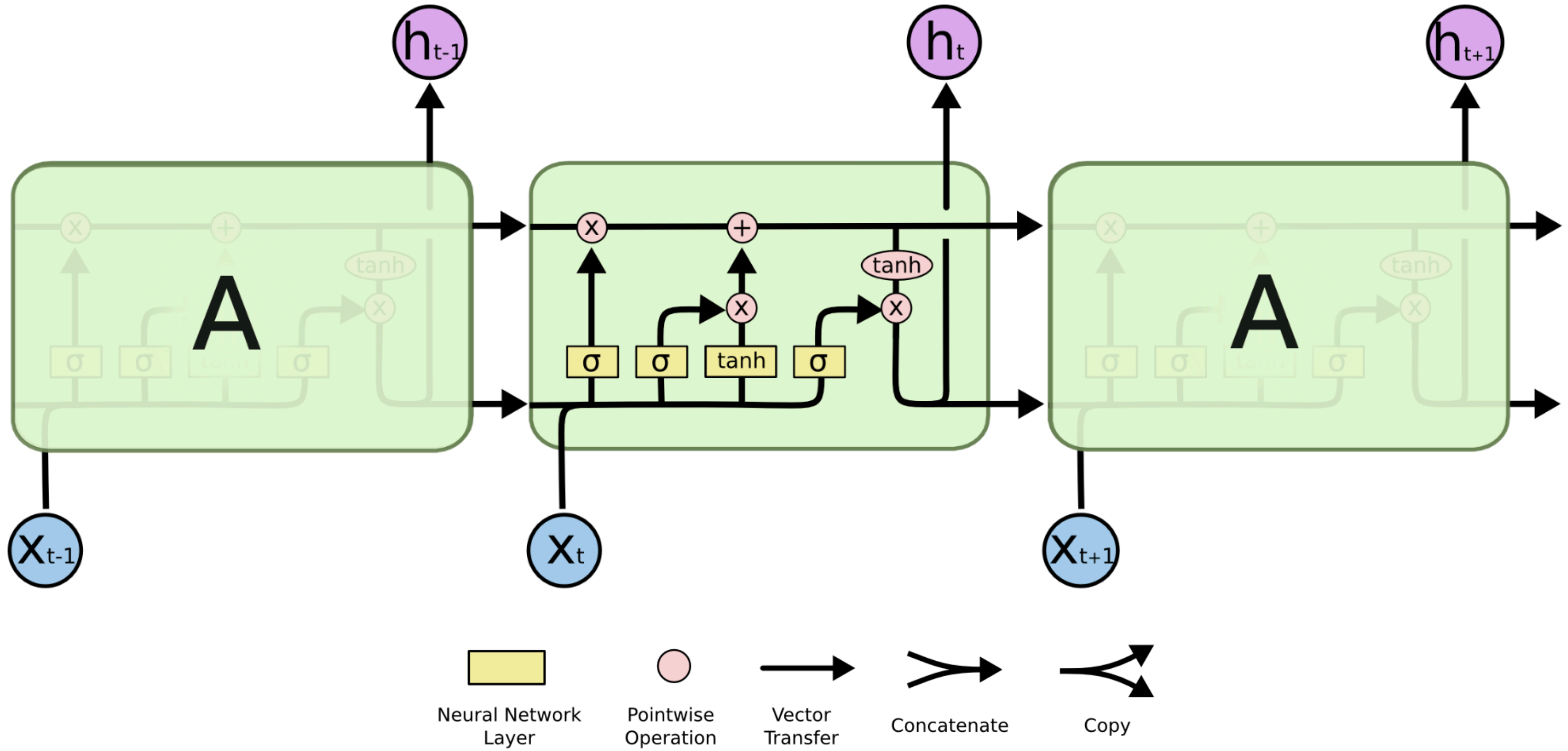
$o_t \in (0,1)^h$: output gate’s activation vector

$h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit

$\tilde{c}_t \in (-1,1)^h$: cell input activation vector

$c_t \in \mathbb{R}^h$: cell state vector

LSTM architecture

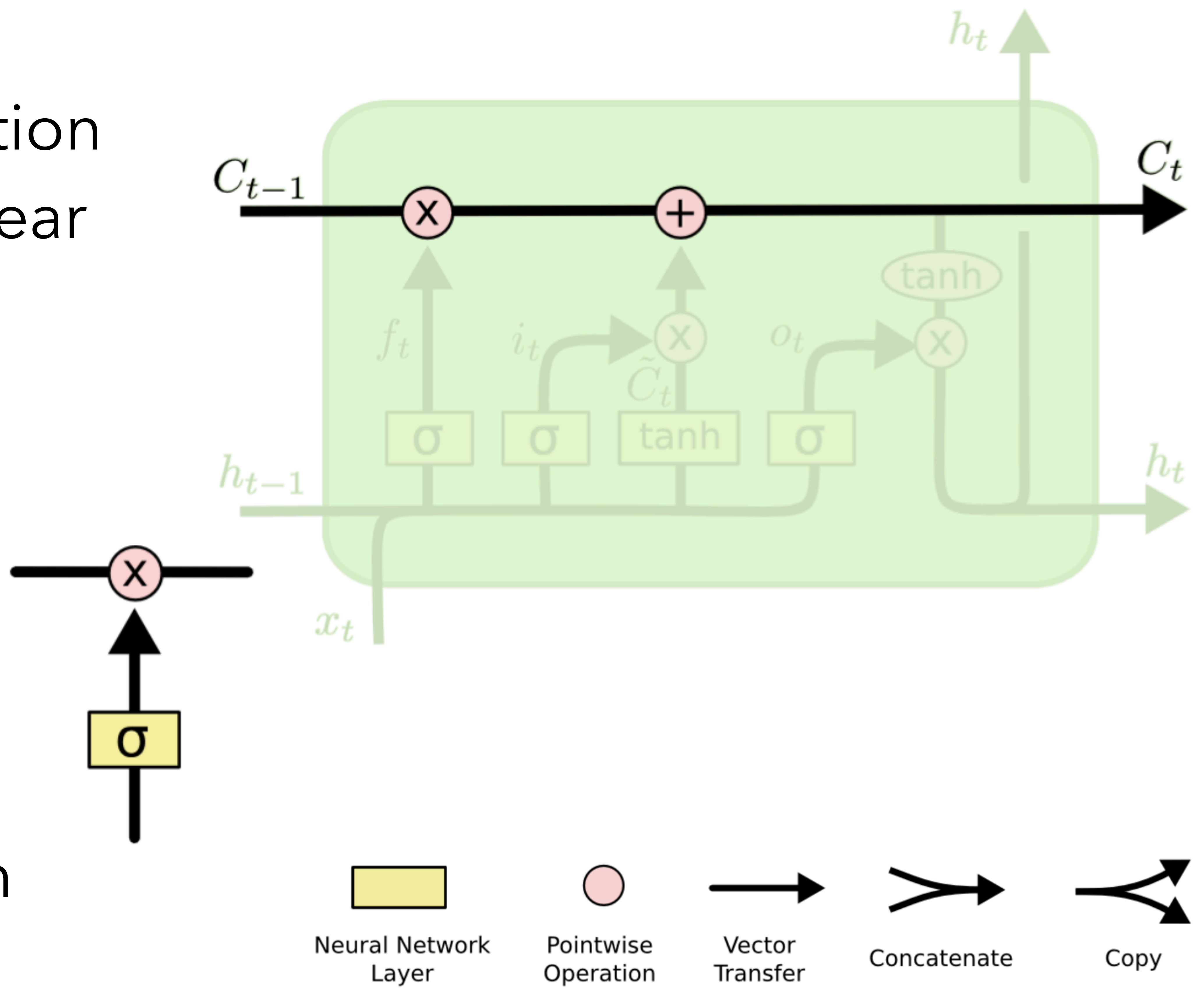


https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM architecture

Cell state (long term memory): allows information to flow with only small, linear interactions (good for gradients!)

- "Gates" optionally let information through
  - 1 - retain information ("remember")
  - 0 - forget information ("forget")



$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$h_t = o_t \odot \sigma_h(c_t)$$

$x_t \in \mathbb{R}^d$: input vector to the LSTM unit

$f_t \in (0,1)^h$: forget gate's activation vec

$i_t \in (0,1)^h$: input/update gate's activat

$o_t \in (0,1)^h$: output gate's activation ve

$h_t \in (-1,1)^h$: hidden state vector also k vector of the LSTM unit

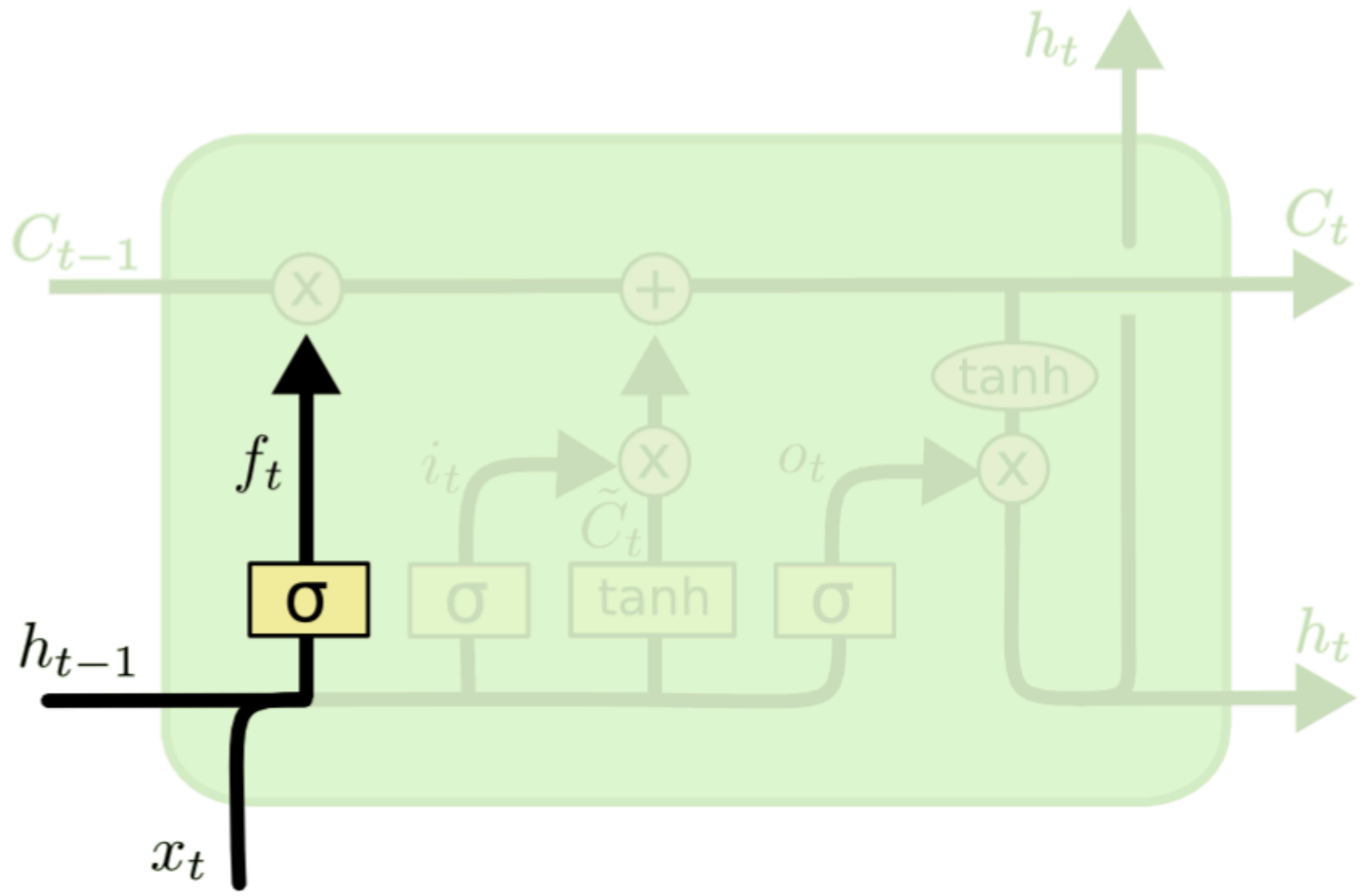
$\tilde{c}_t \in (-1,1)^h$: cell input activation vecto

$c_t \in \mathbb{R}^h$: cell state vector

LSTM architecture

Input Gate Layer: Decide what information to "forget"

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$h_t = o_t \odot \sigma_h(c_t)$$



$x_t \in \mathbb{R}^d$: input vector to the LSTM unit

$f_t \in (0,1)^h$: forget gate's activation vector

$i_t \in (0,1)^h$: input/update gate's activation vector

$o_t \in (0,1)^h$: output gate's activation vector

$h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit

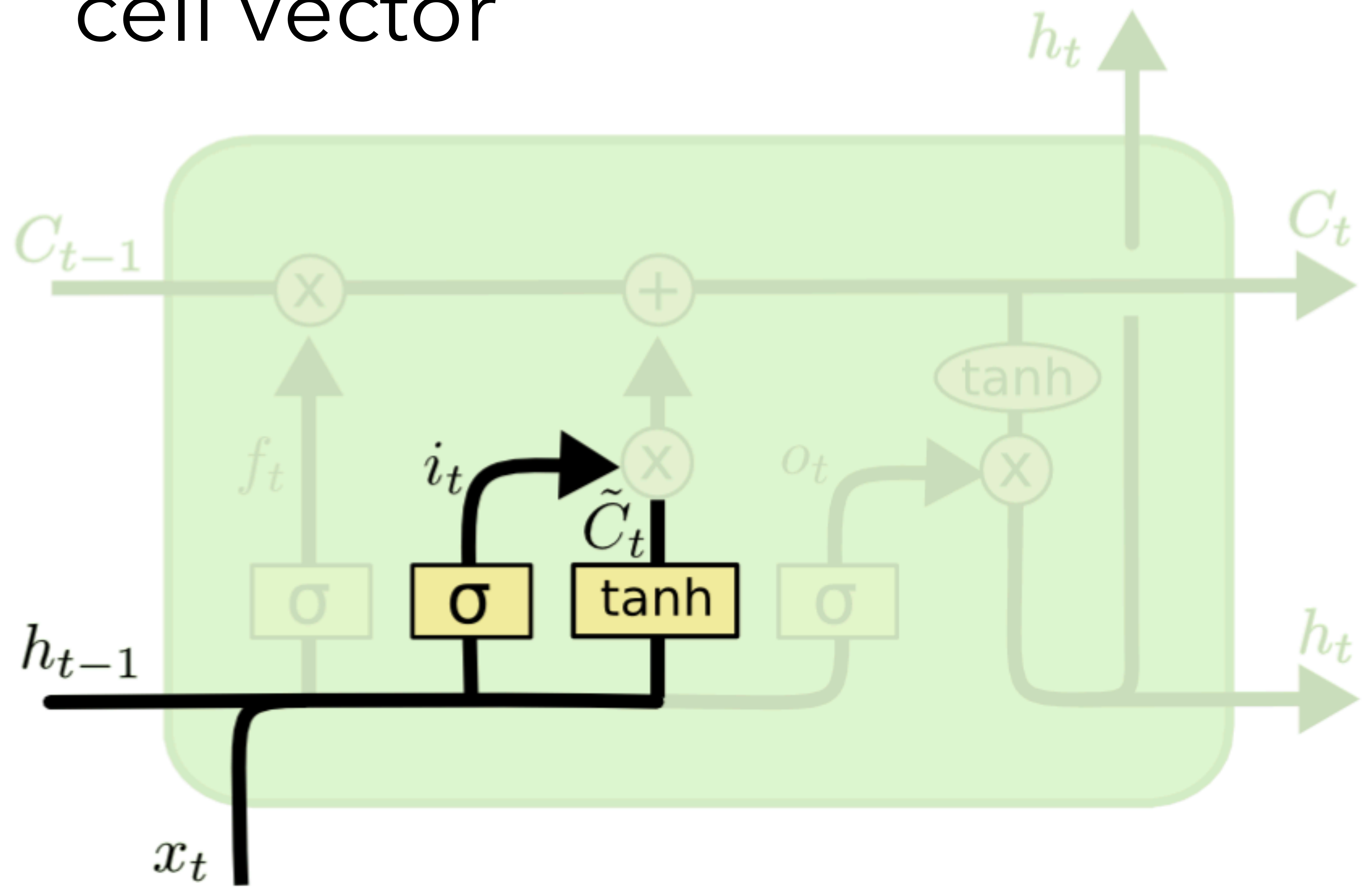
$\tilde{c}_t \in (-1,1)^h$: cell input activation vector

$c_t \in \mathbb{R}^h$: cell state vector

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM architecture

Candidate state values:
Extract candidate information to put into the cell vector



$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$h_t = o_t \odot \sigma_h(c_t)$$

$x_t \in \mathbb{R}^d$: input vector to the LSTM unit

$f_t \in (0,1)^h$: forget gate's activation vector

$i_t \in (0,1)^h$: input/update gate's activation vector

$o_t \in (0,1)^h$: output gate's activation vector

$h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit

$\tilde{c}_t \in (-1,1)^h$: cell input activation vector

$c_t \in \mathbb{R}^h$: cell state vector

LSTM architecture

Update cell: "Forget" the information we decided to forget and update with new candidate information

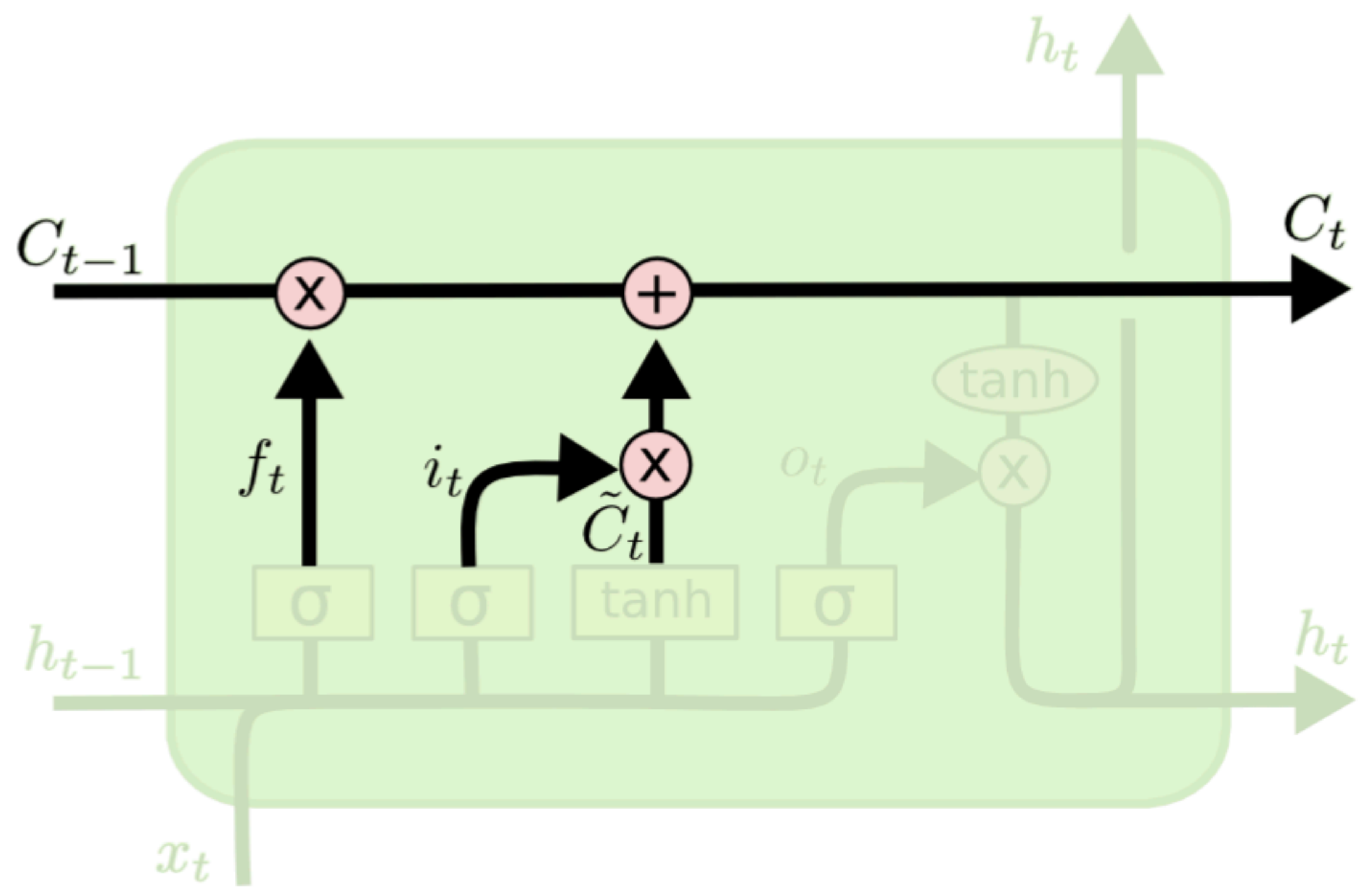
If f_t is

- High: we "remember" more previous info
- Low: we "forget" more info

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$h_t = o_t \odot \sigma_h(c_t)$$

If i_t is

- High: we add more new info
- Low: we add less new info



$x_t \in \mathbb{R}^d$: input vector to the LSTM unit

$f_t \in (0,1)^h$: forget gate's activation vector

$i_t \in (0,1)^h$: input/update gate's activation vector

$o_t \in (0,1)^h$: output gate's activation vector

$h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit

$\tilde{c}_t \in (-1,1)^h$: cell input activation vector

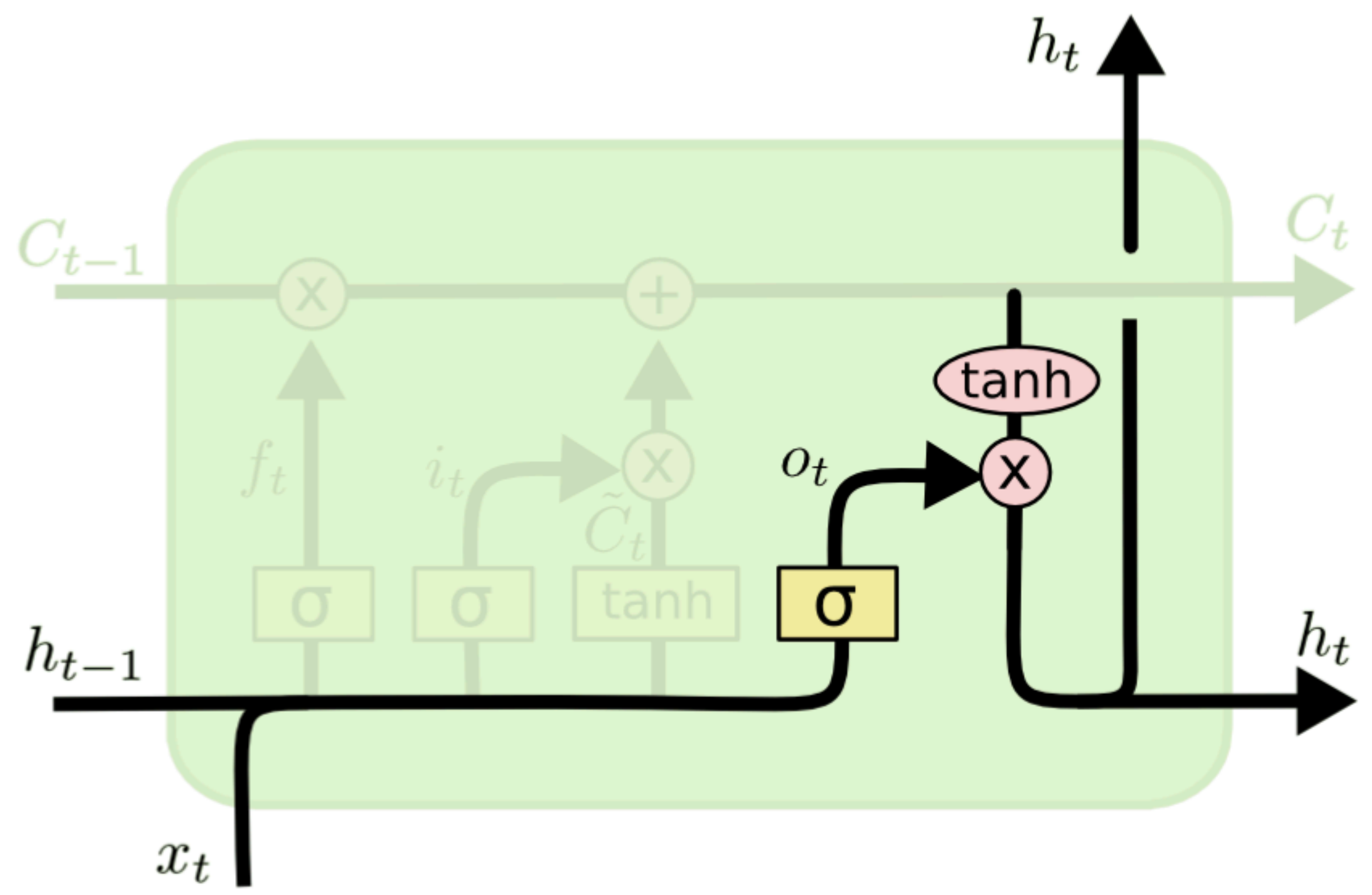
$c_t \in \mathbb{R}^h$: cell state vector

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM architecture

Output/Short-term Memory (as in RNN):

Pass information onto the next state/for use in output (e.g., probabilities)



Pass on different information than in the long-term memory vector

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$h_t = o_t \odot \sigma_h(c_t)$$

$x_t \in \mathbb{R}^d$: input vector to the LSTM unit

$f_t \in (0,1)^h$: forget gate's activation vector

$i_t \in (0,1)^h$: input/update gate's activation vector

$o_t \in (0,1)^h$: output gate's activation vector

$h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit

$\tilde{c}_t \in (-1,1)^h$: cell input activation vector

$c_t \in \mathbb{R}^h$: cell state vector

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTMs (summary)

Pros:

- Works for arbitrary sequence lengths (as RNNs)
- Address the vanishing gradient problems via long- and short-term memory units with gates

Cons:

- Calculations are sequential - computation at time t depends entirely on the calculations done at time t-1
  - As a result, hard to parallelize and train

Enter transformers… (next time)