COMP 3361 Natural Language Processing

Lecture 9: Neural language models: RNNs and LSTM

Spring 2025

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

Announcements

- Tutorial on Assignment 1 by TA.
- Make sure you check out the recorded tutorial on PyTorch and HuggingFace

Latest AI news

- GPT4.5 is public now.

Lecture plan

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

Neural language models: tokenization

Byte-pair encoding: tokenization/encoding

$$\mathcal{V} = \{1: \text{' '}, 2: \text{'a'}, 3: \text{'e'}, 4: \text{'f'}, 5: \text{'g'}, 6: \text{'h'}, 7: \text{'i'}, 8: \text{'k'}, 9: \text{'m'}, 10: \text{'n'}, 11: \text{'p'}, 12: \text{'s'}, 13: \text{'u'}, 14: \text{'ug'}, 15: \text{' p'}, 16: \text{'hug'}, 17: \text{' pug'}, 18: \text{' pugs'}, 19: \text{'un'}, 20: \text{' hug'}\}$$

Encoding algorithm

Given string $\mathcal{S}$ and (ordered) vocab $\mathcal{V}$,

- Pretokenize $\mathcal{D}$ in same way as before
- Tokenize $\mathcal{D}$ into characters
- Perform merge rules in same order as in training until no more merges may be done

Byte-pair encoding: tokenization/encoding

$$\mathcal{V} = \{1: \text{` '}, 2: \text{`a'}, 3: \text{`e'}, 4: \text{`f'}, 5: \text{`g'}, 6: \text{`h'}, 7: \text{`i'}, 8: \text{`k'}, 9: \text{`m'}, 10: \text{`n'}, 11: \text{`p'}, 12: \text{`s'}, 13: \text{`u'}, 14: \text{`ug'}, 15: \text{` p'}, 16: \text{`hug'}, 17: \text{` pug'}, 18: \text{` pugs'}, 19: \text{`un'}, 20: \text{` hug'}\}$$

Encoding algorithm

Given string $\mathcal{S}$ and (ordered) vocab $\mathcal{V}$,

- Pretokenize $\mathcal{D}$ in same way as before
- Tokenize $\mathcal{D}$ into characters
- Perform merge rules in same order as in training until no more merges may be done

$$\text{Encode}(\text{`` hugs''}) = [20, 12]$$

$$\text{Encode}(\text{``misshapenness''}) = [9, 7, 12, 12, 6, 2, 11, 3, 10, 10, 3, 12, 12]$$

Byte-pair encoding: decoding

$$\mathcal{V} = \{1: \text{` '}, 2: \text{`a'}, 3: \text{`e'}, 4: \text{`f'}, 5: \text{`g'}, 6: \text{`h'}, 7: \text{`i'},$$
$$8: \text{`k'}, 9: \text{`m'}, 10: \text{`n'}, 11: \text{`p'}, 12: \text{`s'}, 13: \text{`u'},$$
$$14: \text{`ug'}, 15: \text{` p'}, 16: \text{`hug'}, 17: \text{` pug'}, 18: \text{` pugs'},$$
$$19: \text{`un'}, 20: \text{` hug'}\}$$

Decoding algorithm

Given list of tokens T:

- Initialize string $s :=$ “”
- Keep popping off tokens from the front of T and appending the corresponding string to s

$$\text{Encode}(\text{`` hugs''}) = [20, 12]$$
$$\text{Encode}(\text{``misshapenness''}) = [9, 7, 12, 12, 6, 2, 11, 3, 10, 10, 3, 12, 12]$$

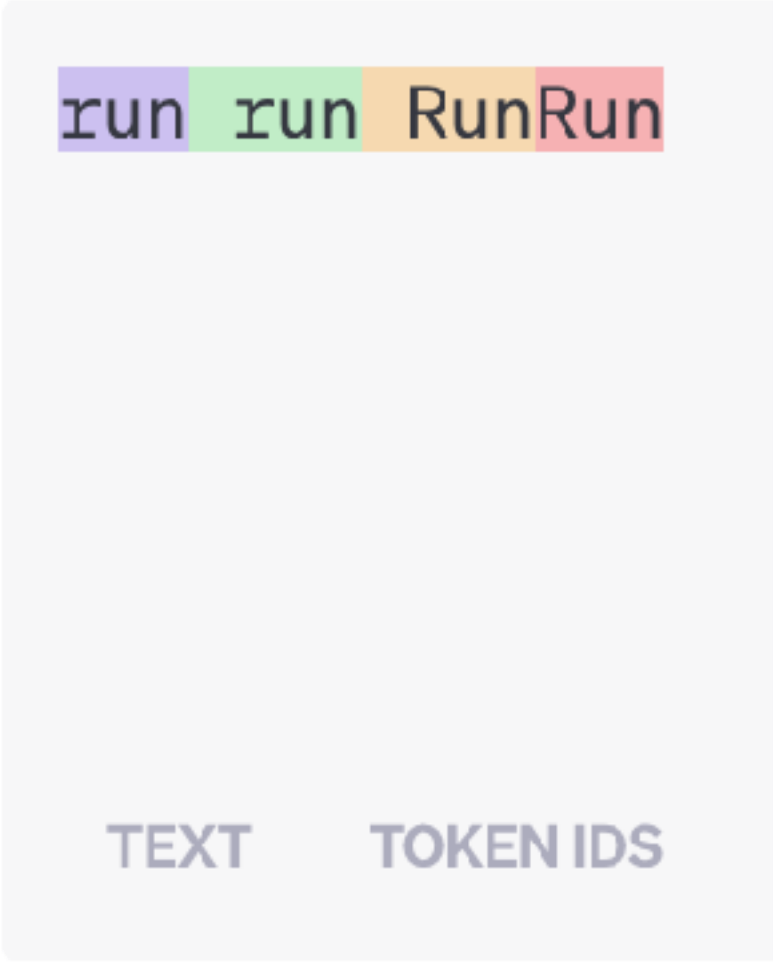
$$\text{Decode}([20, 12]) = \text{`` hugs''}$$
$$\text{Decode}([9, 7, 12, 12, 6, 2, 11, 3, 10, 10, 3, 12, 12]) = \text{``misshapenness''}$$

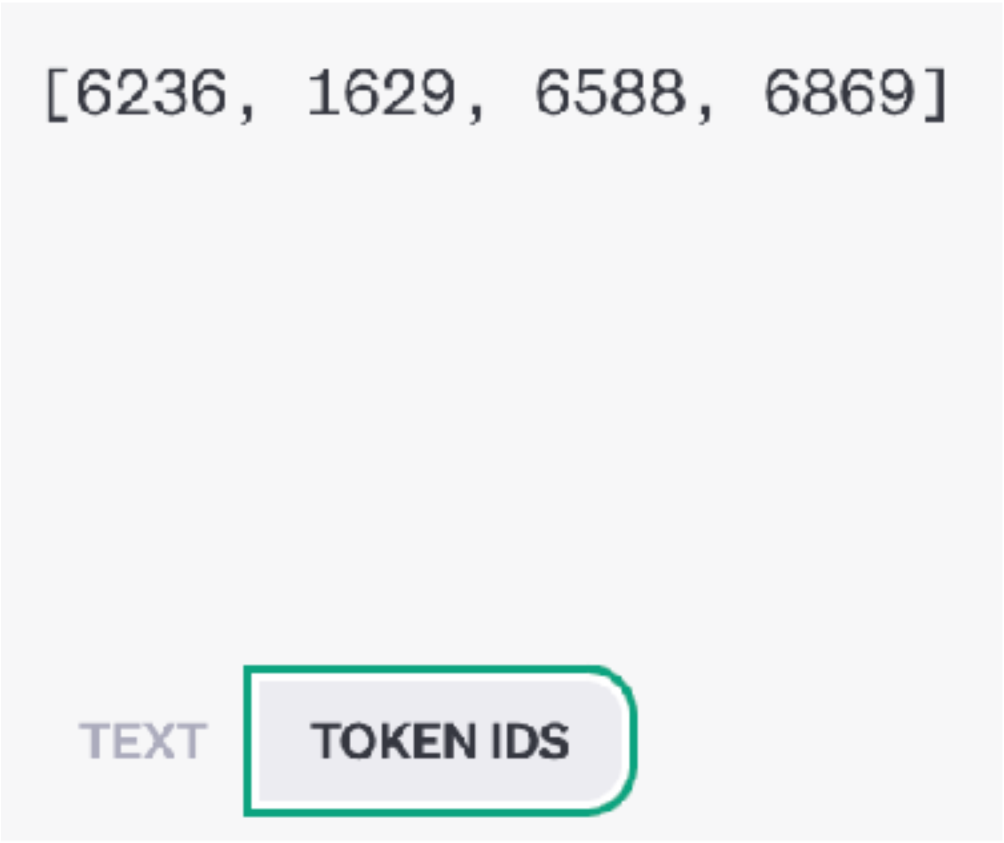
Byte-pair encoding: properties

- Efficient to run (greedy vs. global optimization)
- Lossless compression
- Potentially some shared representations - e.g., the token "hug" could be used both in "hug" and "hugging"

Weird properties of tokenizers

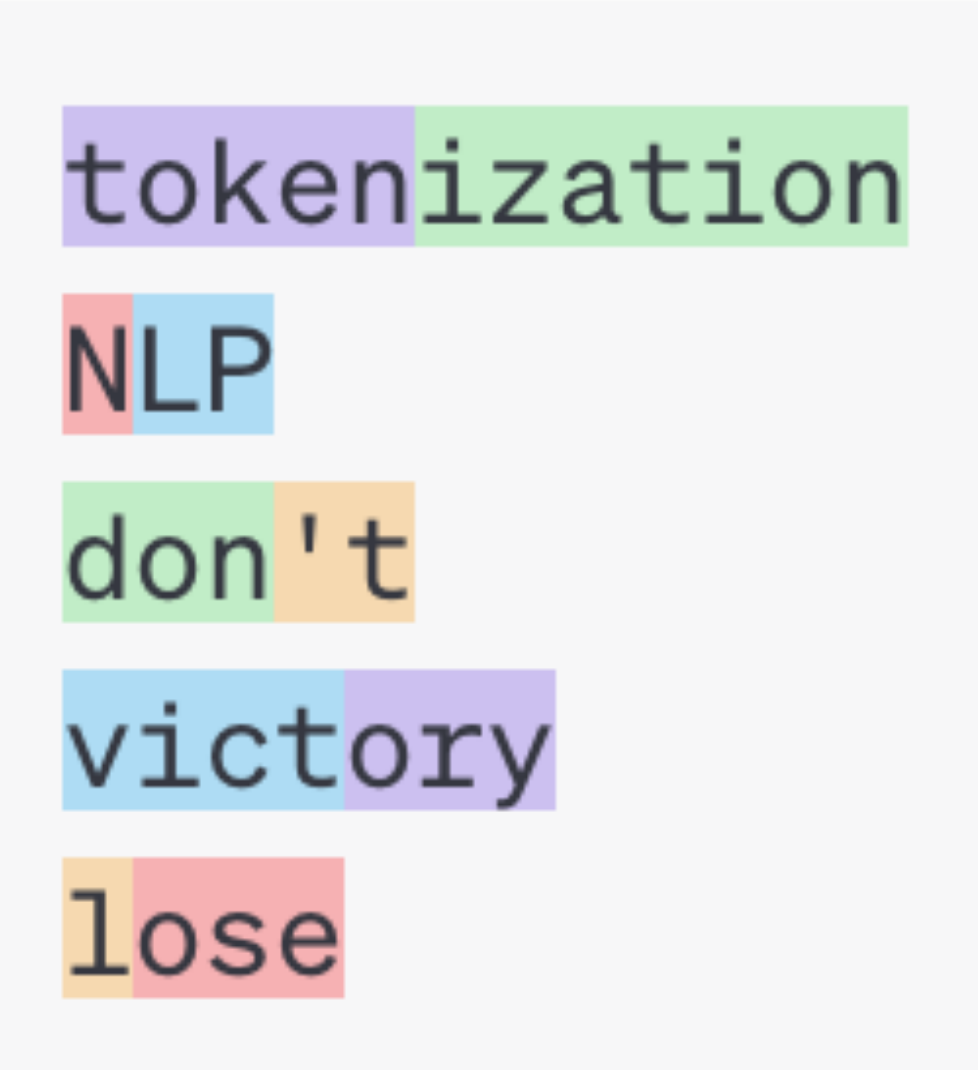
- Token != word
- Spaces are part of token
  - "run" is a different token than " run"
- Not invariant to case changes
  - "Run" is a different token than "run"

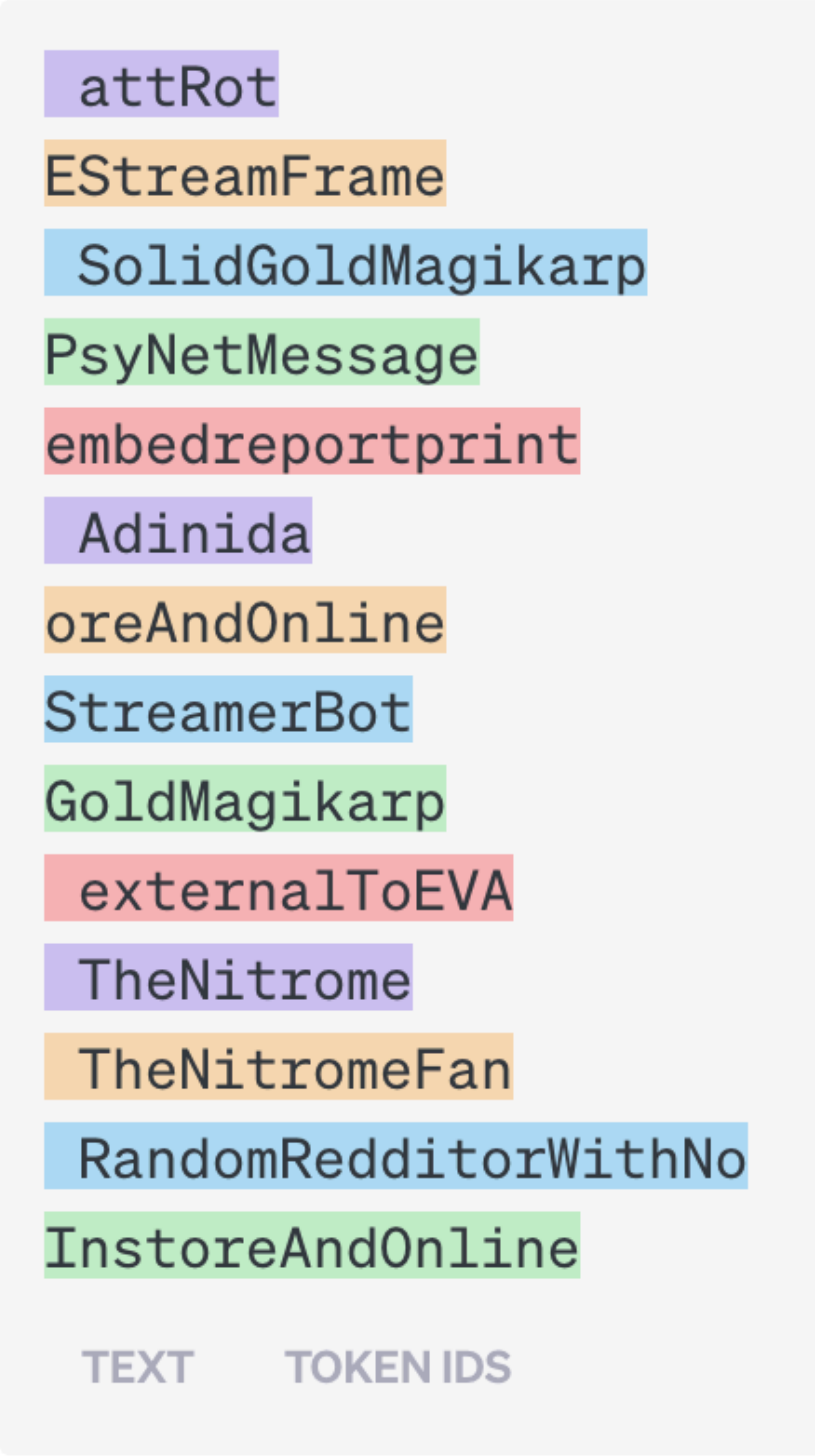




Weird properties of tokenizers

- Token != word
- Spaces are part of token
  - "run" is a different token than " run"
- Not invariant to case changes
  - "Run" is a different token than "run"
- Tokenization fits statistics of your data
  - e.g., while these words are multiple tokens…
  - These words are all 1 token in GPT-3's tokenizer!
  - *Why?*
    - Reddit usernames and certain code attributes appeared enough in the corpus to surface as its own token!





Example from https://www.lesswrong.com/posts/aPeJE8bSo6rAFoLqg/solidgoldmagikarp-plus-prompt-generation

Other tokenization variants

Variants: no spaces in tokens

- The way we presented BPE, we included whitespace with the following word. (E.g., " pug") ← space
  - This is most common in modern LMs
- However, in another BPE variant, you instead strip whitespace (e.g., "pug") and add spaces between words at decoding time ← no space
  - This was the original BPE paper's implementation!
- Example:
  - ["I", "hug", "pugs"] -> "I hug pugs" (w/out whitespace)
  - ["I", " hug", " pugs"] -> "I hug pugs" (w/ whitespace)

Original (w/ whitespace)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (**split before** whitespace/punctuation)

- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$

Updated (w/out whitespace)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

+ Pre-tokenize $\mathcal{D}$ by splitting into words (**removing** whitespace)

- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$

Variants: no spaces in tokens

- For sub-word tokens, need to add “continue word” special character
  - E.g., for the word “Tokenization”, if the subword tokens are “Token” and “ization”,
    - W/out special character: [“Token”, “ization”] -> “Token ization”
    - W/ special character #: [“Token”, “#ization”] -> Tokenization”
  - When decoding, if does not have special character add a space
- Example:
  - [“I”, “li”, “#ke”, “to”, “hug”, “pug”, “#s”] -> “I like to hug pugs”

Variants: no spaces in tokens

- Loses some whitespace information (lossy compression!)
  - E.g., Tokenize("I eat cake.") == Tokenize(" I   eat cake   .")
  - Especially problematic for code (e.g., Python) - why?

```
tokenizer = AutoTokenizer.from_pretrained("openai-gpt")
tokens = tokenizer.encode("i eat cake.")
print(tokens)
print(tokenizer.decode(tokens))

tokens = tokenizer.encode(" i       eat   cake    .")
print(tokens)
print(tokenizer.decode(tokens))
```

✓ 0.4s

```
[249, 2425, 5409, 239]
i eat cake.
[249, 2425, 5409, 239]
i eat cake.
```

(Example using GPT's tokenizer, which does not include spaces in the token)

Variants: no pre-tokenization

- In the variant we proposed, we start by splitting into words
  - This guarantees that each token will be no longer than one word
  - However, this does not work so well for character-based languages. *Why?*

Variants: no pre-tokenization

- Instead, we could *not* pre-tokenize, and treat the entire document or sentence as a single list of tokens
  - Allows for tokens to span multiple words/characters
- Sometimes called SentencePiece tokenization* (Kudo, 2018)

* (not to be confused with the SentencePiece library, which is an implementation of *many* kinds of tokenization)

Paper: https://arxiv.org/abs/1808.06226
Library: https://github.com/google/sentencepiece

Original (w/ pre-tokenization)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- **Pre-tokenize** $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)

- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$

Updated (w/out pre-tokenization)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

\+ **Do not pre-tokenize** $\mathcal{D}$

- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$
- Convert $\mathcal{D}$ into a list of tokens (characters)

Variants: no pre-tokenization

- Allows sequences of <u>words</u>/characters to become tokens

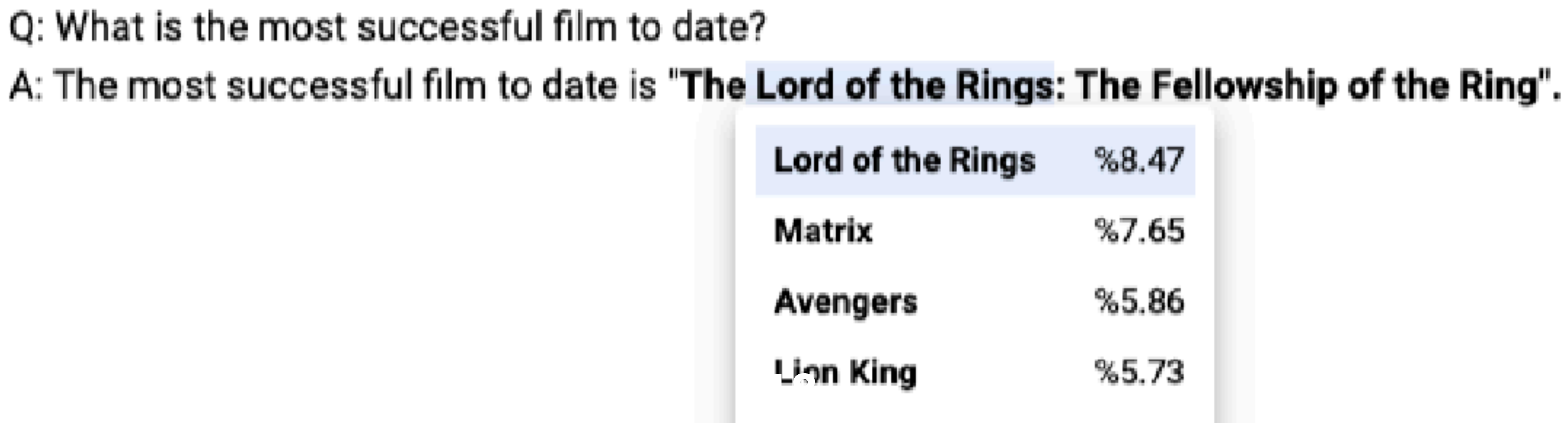
SentencePiece paper example in Japanese:

https://arxiv.org/pdf/1808.06226.pdf

- **Raw text:** [こんにちは世界。] (*Hello world.*)
- **Tokenized:** [こんにちは] [世界] [。]

Jurassic-1 model example in English:

https://uploads-ssl.webflow.com/60fd4503684b466578c0d307/61138924626a6981ee09caf6_jurassic_tech_paper.pdf



Variants: byte-based

- Originally, we presented BPE as dealing with characters as the smallest unit
  - However, there are *many* characters - especially if you want to support:
    - character-based languages (e.g., Chinese has >100k characters!)
    - non-alphanumeric characters like emojis (Unicode 15 has ~150k characters!)
- Instead, can initialize tokens as set of bytes! (e.g., with UTF-8*)

*Only 256 bytes! Each Unicode char is 1-4 bytes

Original (w/ characters)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)

\- Initialize $\mathcal{V}$ as the set of **characters** in $\mathcal{D}$

\- Convert $\mathcal{D}$ into a list of tokens (**characters**)

- While $|\mathcal{V}| < N$:

Modified (w/ bytes)

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$)

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation)

\+ Initialize $\mathcal{V}$ as the set of **bytes** in $\mathcal{D}$

\+ Convert $\mathcal{D}$ into a list of tokens (**bytes**)

- While $|\mathcal{V}| < N$:

Variants: byte-based

While character-based GPT tokenizer fails on emojis and Japanese…

```python
gpt_tokenizer = AutoTokenizer.from_pretrained
tokens = gpt_tokenizer.encode('😂')
print(tokens)
print(gpt_tokenizer.decode(tokens))
tokens = gpt_tokenizer.encode('こんにちは')
print(tokens)
print(gpt_tokenizer.decode(tokens))
```

✓ 0.7s

```
[0]
<unk>
[0, 0, 0, 0, 0]
<unk><unk><unk><unk><unk>
```

The Byte-based GPT-2 tokenizer succeeds!

```python
gpt2_tokenizer = AutoTokenizer.from_pretrained("gpt2")
tokens = gpt2_tokenizer.encode('😂')
print(tokens)
print(gpt2_tokenizer.decode(tokens))
tokens = gpt2_tokenizer.encode('こんにちは')
print(tokens)
print(gpt2_tokenizer.decode(tokens))
```

✓ 0.5s

```
[47249, 224]
😂
[46036, 22174, 28618, 2515, 94, 31676]
こんにちは
```

Variants: WordPiece objective

- To merge, we selected the bigram with highest frequency $p(v_i, v_j)$
  - This is the same as bigram with highest probability!
- Instead, we could choose the bigram which would maximize the likelihood of the data after the merge is made (also called WordPiece!)

Original (BPE)

$\dots$

- For the most frequent bigram v_i, v_j (breaking ties arbitrarily)

(Sam as bigram which maximizes - $p(v_i, v_j)$)

Modified (Word Piece)

$\dots$

+ For the bigram that would maximize likelihood of the training data once the change is made v_i, v_j (breaking ties arbitrarily)

(Same as bigram which maximizes $\frac{p(v_i, v_j)}{p(v_i)p(v_j)}$)

Variants: WordPiece objective

- BPE: the bigram with highest frequency/highest probability $p(v_i, v_j)$
- WordPiece: bigram which maximizes the likelihood of the data after the merge is made $\frac{p(v_i, v_j)}{p(v_i)p(v_j)}$
    - Maximizes the probability of the bigram, normalized by the probability of the unigrams

Variants: WordPiece encoding

At inference time, instead of applying the merge rules in order, tokens are selected left-to-right greedily:

Encoding algorithm

Given string s and (unordered) vocab $\mathcal{V}$,

- Initialize list of tokens $T := []$
- While $len(s) > 0$:
  - Find longest token t_i that matches the beginning of s
  - Let $T := T + [t_i]$
  - Pop corresponding vocab v_i off of front of s
- Return T

Variants: unigram objective

- BPE starts with a small vocabulary (characters) and builds up until the desired vocabulary size N
- The Unigram tokenization algorithm starts with a large vocabulary (all sub-word substrings) and throws away tokens until we reach size N

Examples of LLMs and their tokenizers

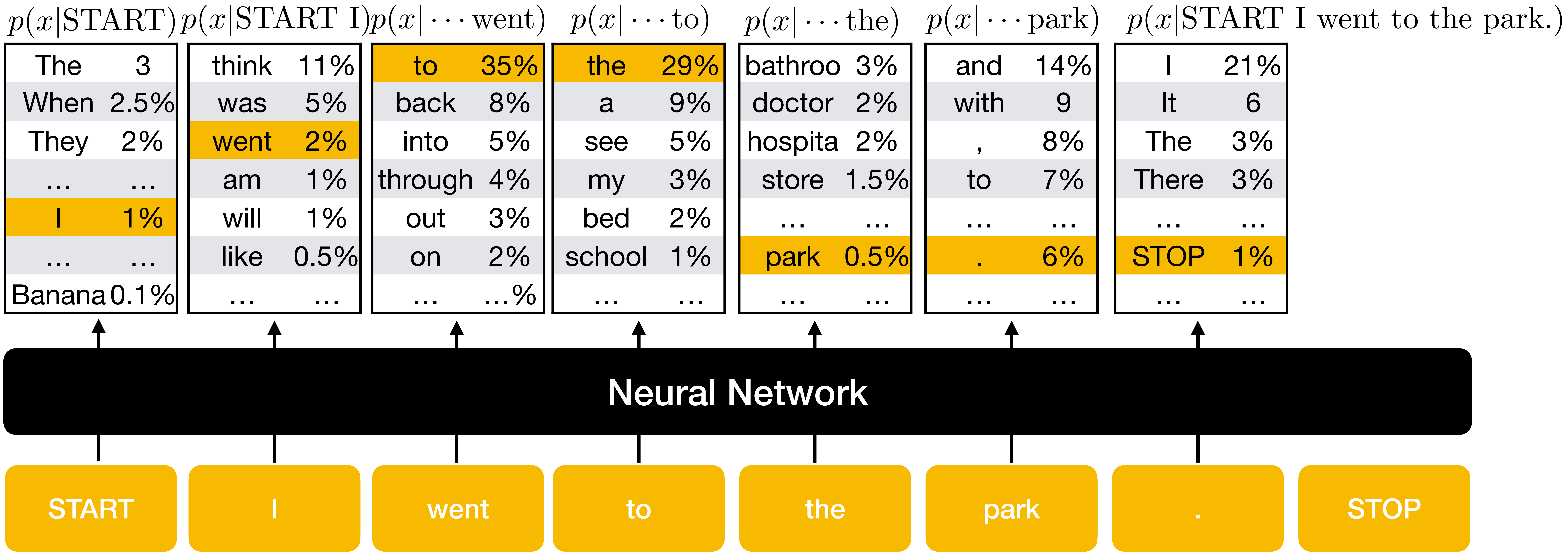
Model/Tokenizer	Objective	Spaces part of token?	Pre-tokenization	Smallest unit
GPT	BPE	No	Yes	Character-level
GPT-2/3/4, ChatGPT, Llama(2), Falcon, …	BPE	Yes	Yes	Byte-level
Jurassic	BPE	Yes	No. "SentencePiece" - treat whitespace like char	Byte-level
Bert, DistilBert, Electra	WordPiece	No	Yes	Character-level
T5, ALBERT, XLNet, Marian	Unigram	Yes	No. "SentencePiece" - treat whitespace like char*	Character-level

*For non-English languages

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?

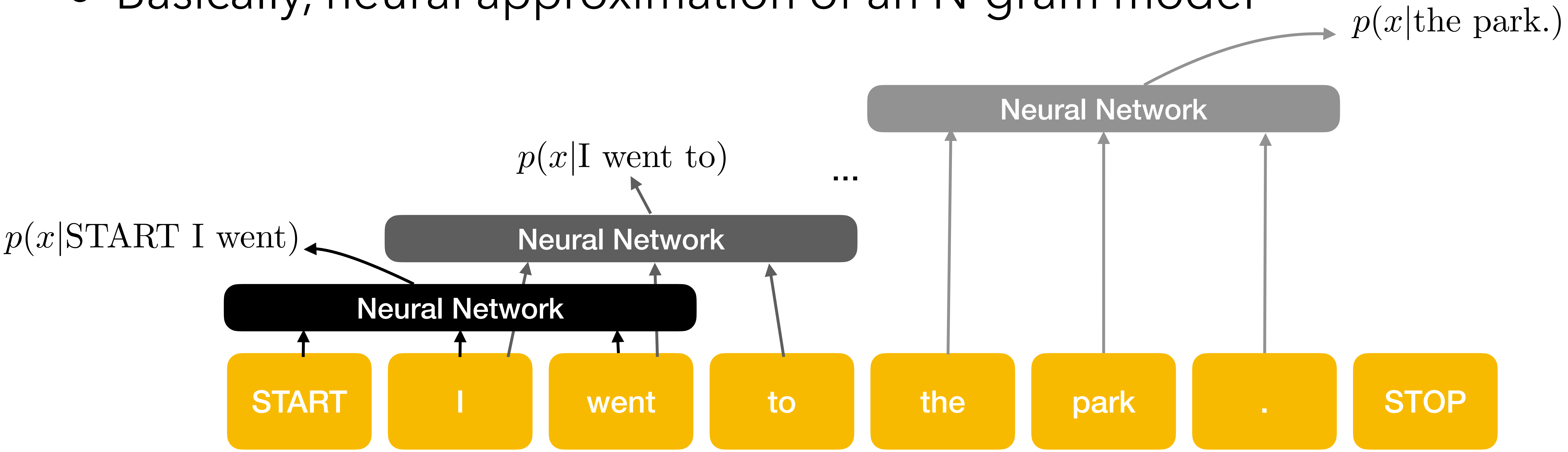
- Don’t neural networks need a fixed-size vector as input? And isn’t text variable length?

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

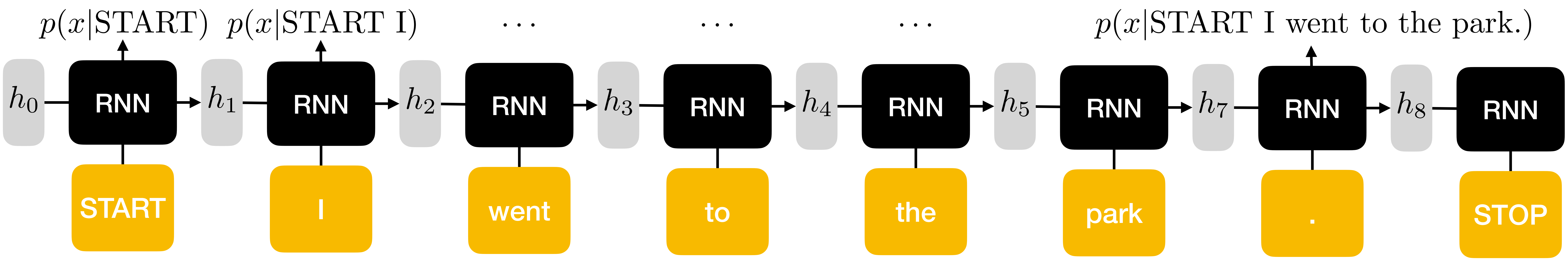


Recurrent neural networks

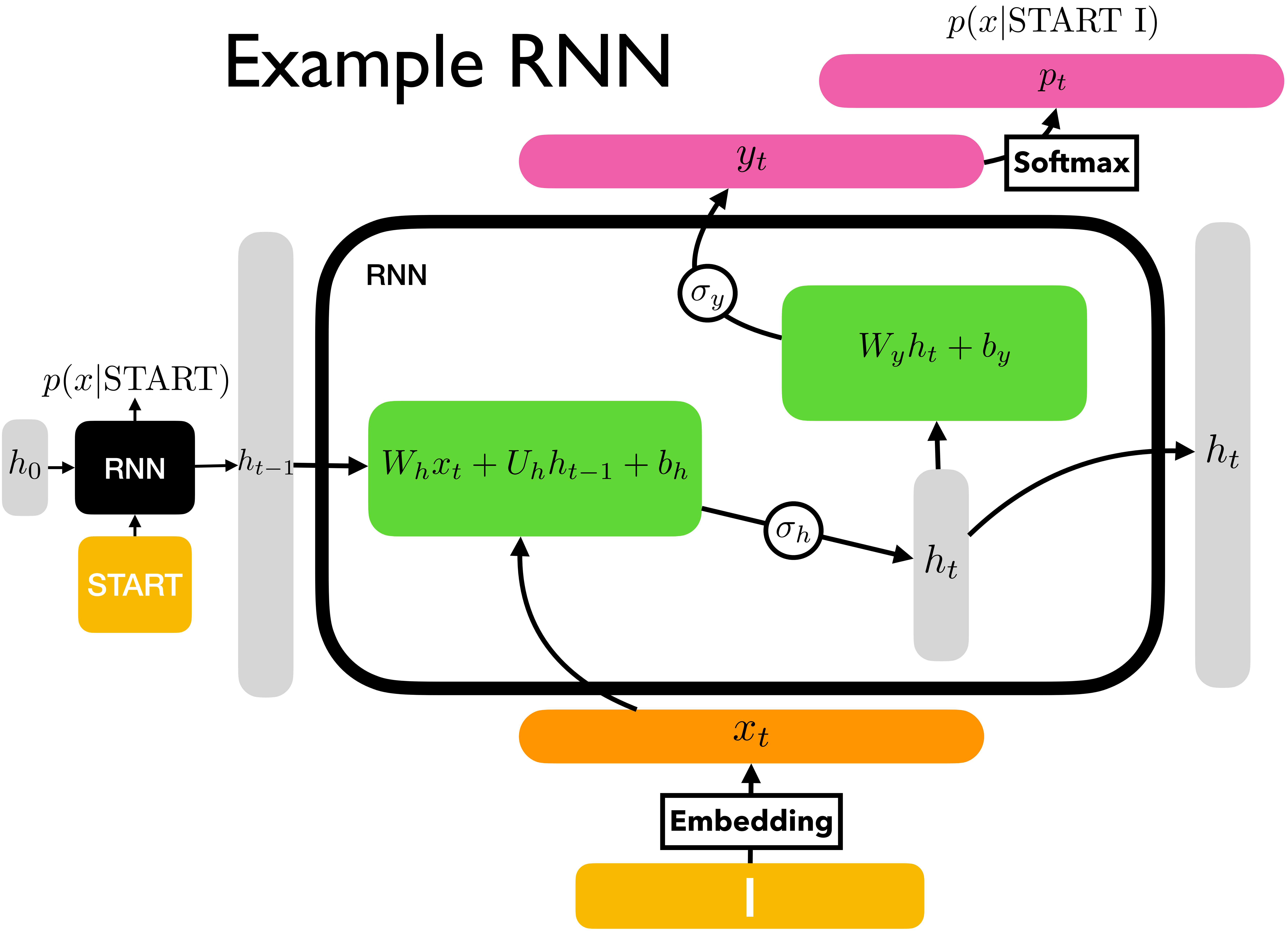
Idea 2: Recurrent Neural Networks (RNNs)

<u>Essential components:</u>

- 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

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

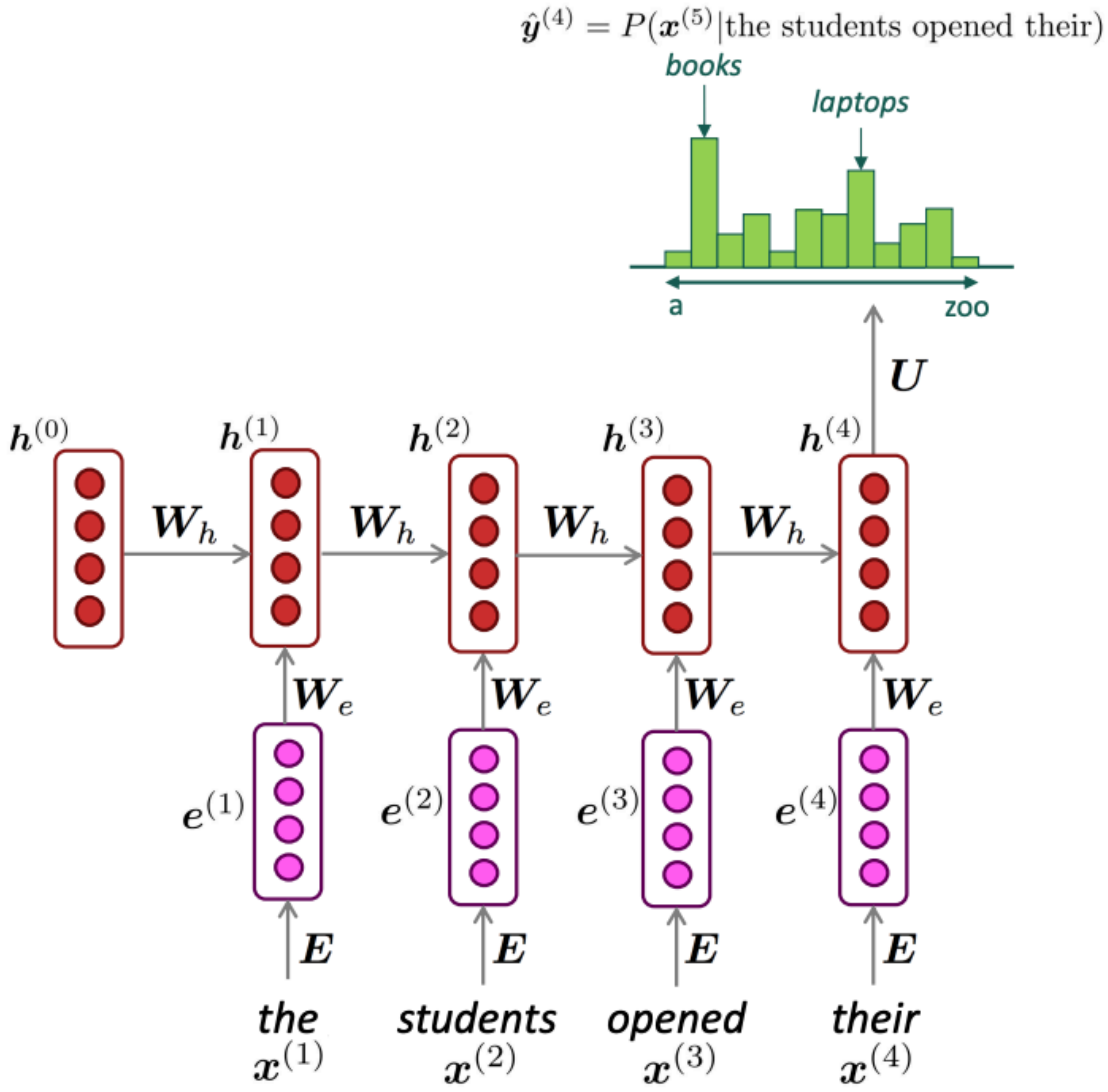
$\boldsymbol{h}^{(0)}$ is the initial hidden state

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!*

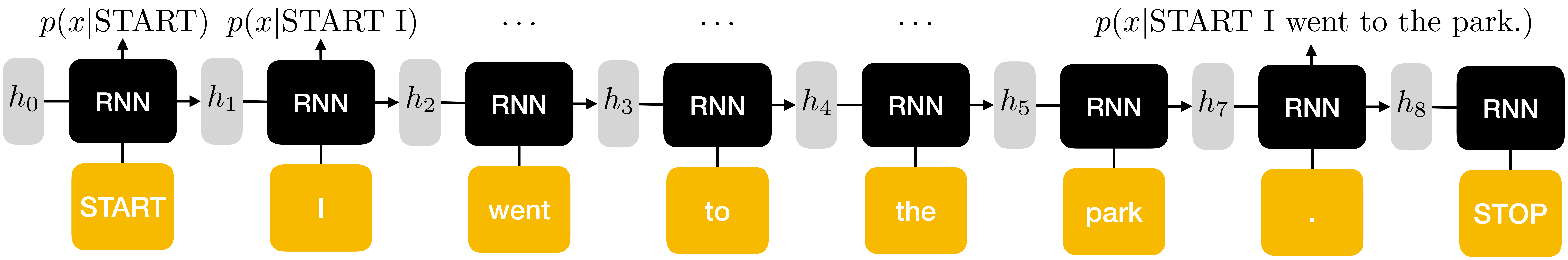
Recurrent neural networks

- 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

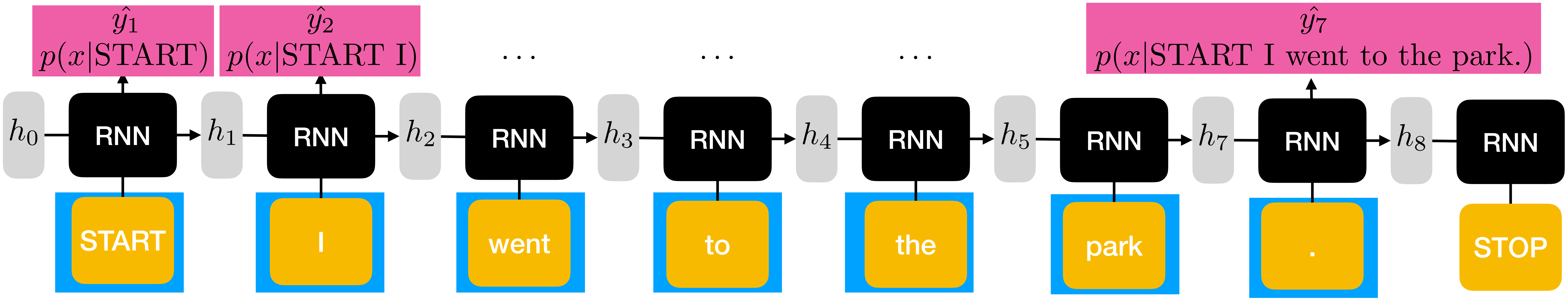
Training procedure

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

$$\hat{y}_1 = \begin{matrix} p(\text{STOP}|\text{START}) \\ p(\text{The}|\text{START}) \\ p(\text{I}|\text{START}) \\ \\ \\ p(\text{apple}|\text{START}) \end{matrix} \begin{bmatrix} .01 \\ .03 \\ .1 \\ .001 \\ \vdots \\ .002 \end{bmatrix} \qquad y_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

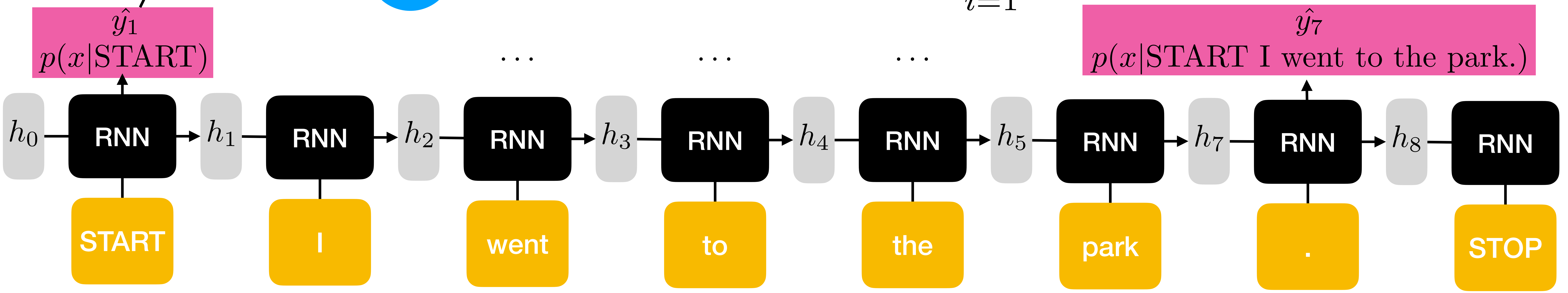
2. Run model on (batch of) x's from data $\mathcal{D}$ to get probability distributions $\hat{y}$

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

$$\hat{y}_7 = \begin{bmatrix} .2 \\ .03 \\ .12 \\ .01 \\ \vdots \\ .001 \end{bmatrix} \qquad y_7 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

L_{CE}

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



Training procedure

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

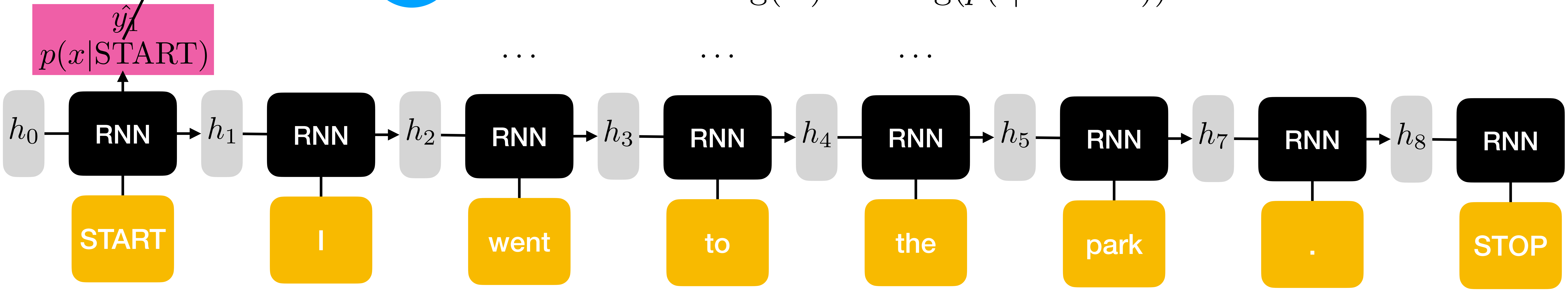
3. Calculate loss compared to true y's (Cross Entropy Loss)

$$\begin{matrix} p(\text{STOP}|\text{START}) \\ p(\text{The}|\text{START}) \\ p(\text{I}|\text{START}) \\ \hat{y}_1 = \\ \\ p(\text{apple}|\text{START}) \end{matrix} \begin{bmatrix} .01 \\ .03 \\ .1 \\ .001 \\ \vdots \\ .002 \end{bmatrix} \qquad y_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

(Actual observed word)

L_{CE}

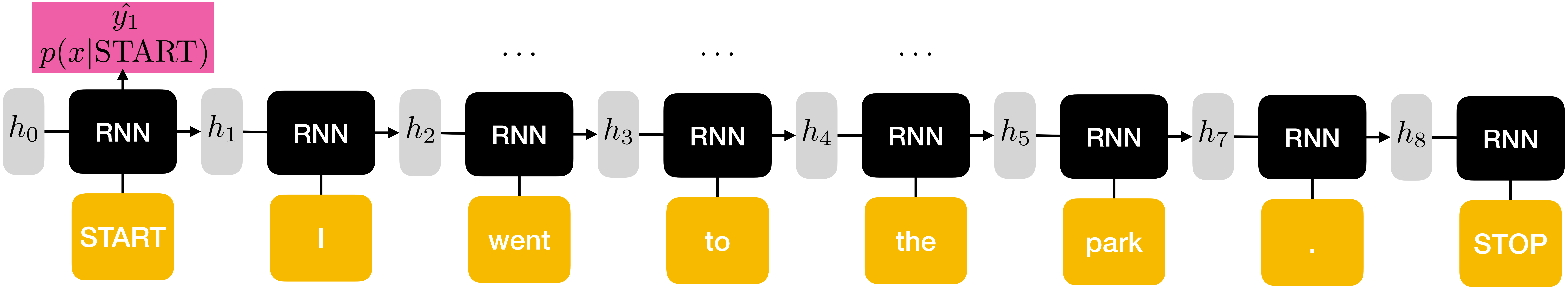
$$L_{\text{CE}}(y_1, \hat{y}_1) = -0 * \log(.01) - 0 * \log(.03) - 1 * -\log(.1) - \cdots - 0 * \log(.002)$$
$$= -\log(.1) = -\log(p(\text{I}|\text{START}))$$



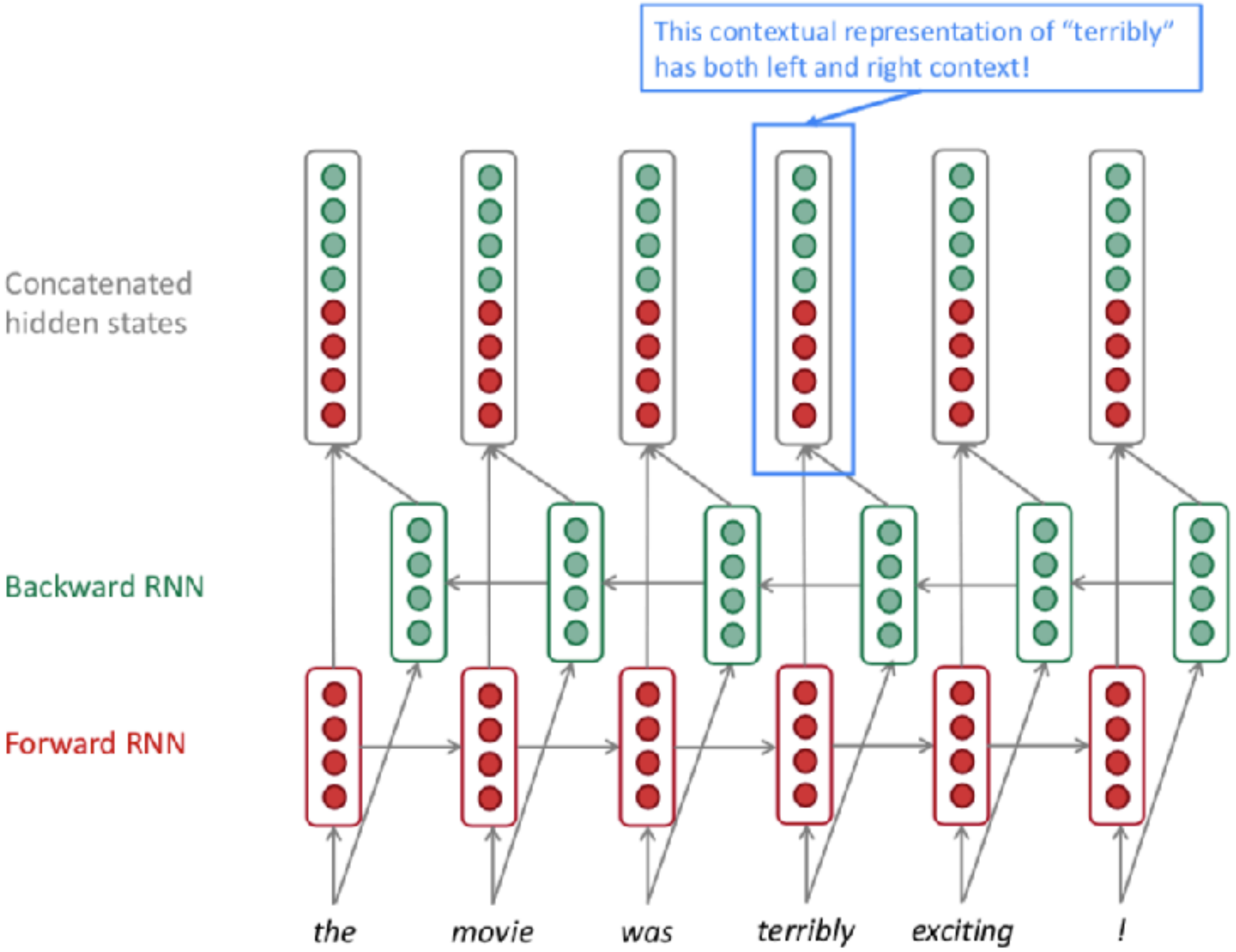
Training procedure - gradient descent step

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



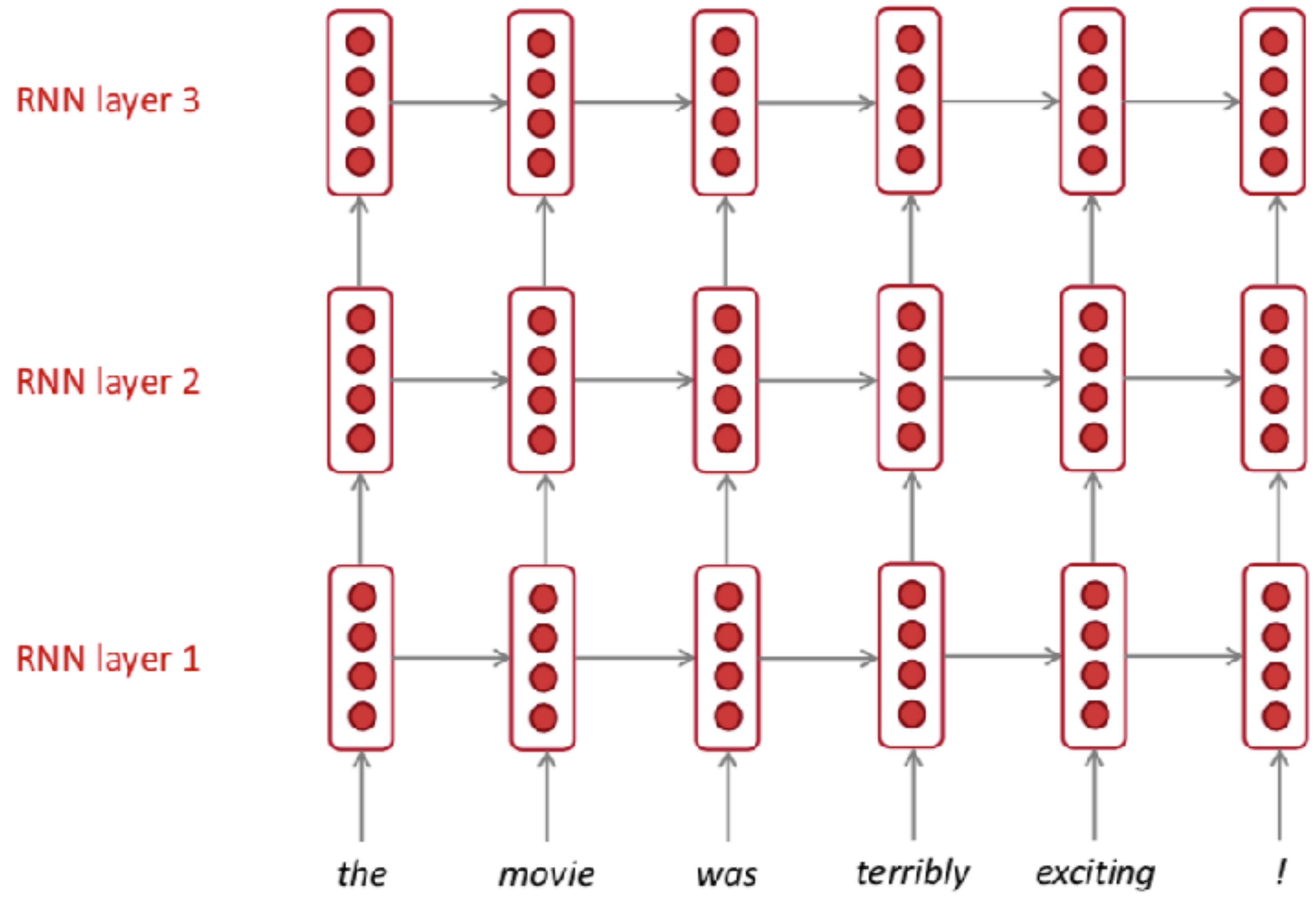
$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^h$$

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

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

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

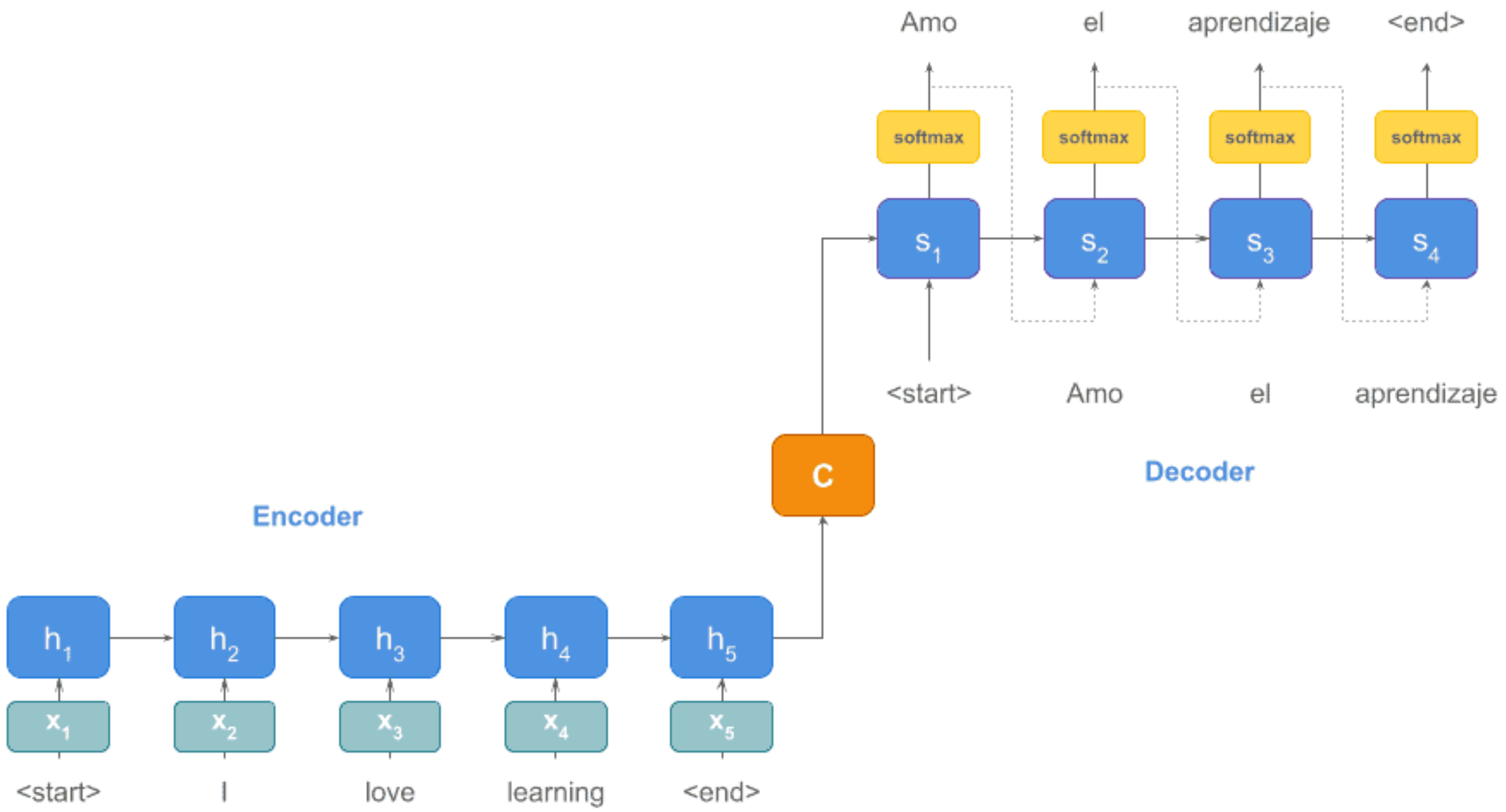
Multi-layer RNNs



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

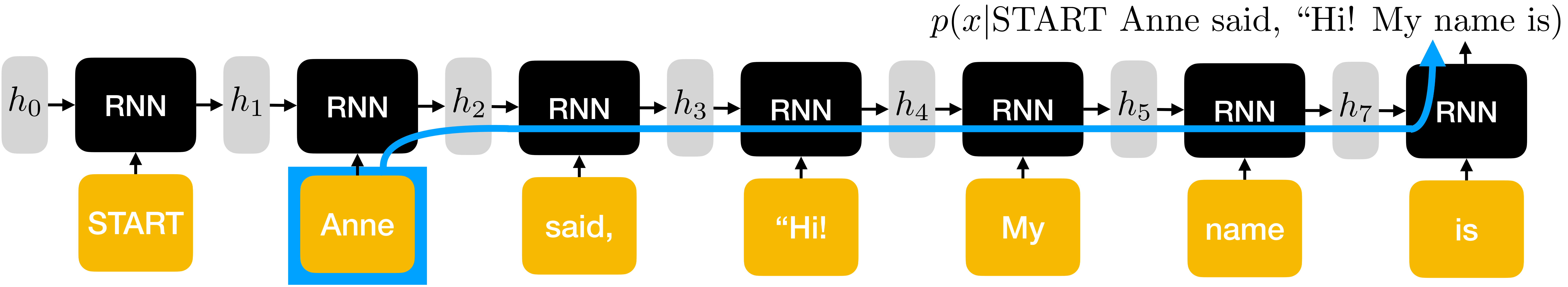
What word is likely to come next for this sequence?

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

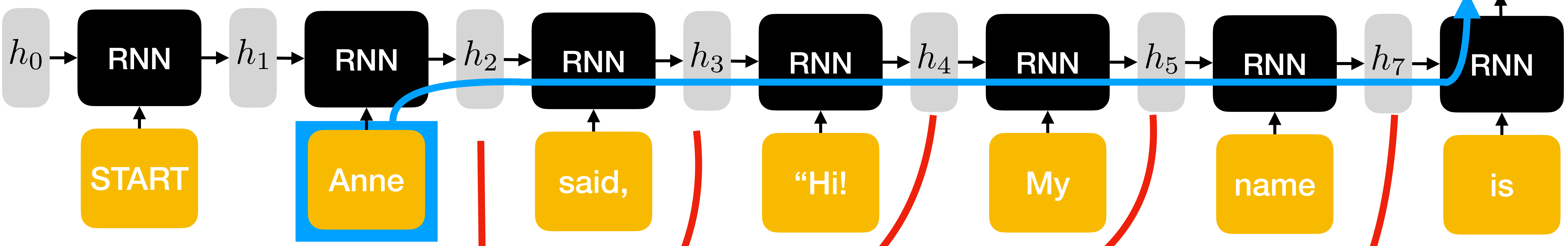


- Need **relevant information** to flow across many time steps
- When we backpropagate, we want to allow the relevant information to flow

RNNs - vanishing gradient problem

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

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



Backprop steps

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

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

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

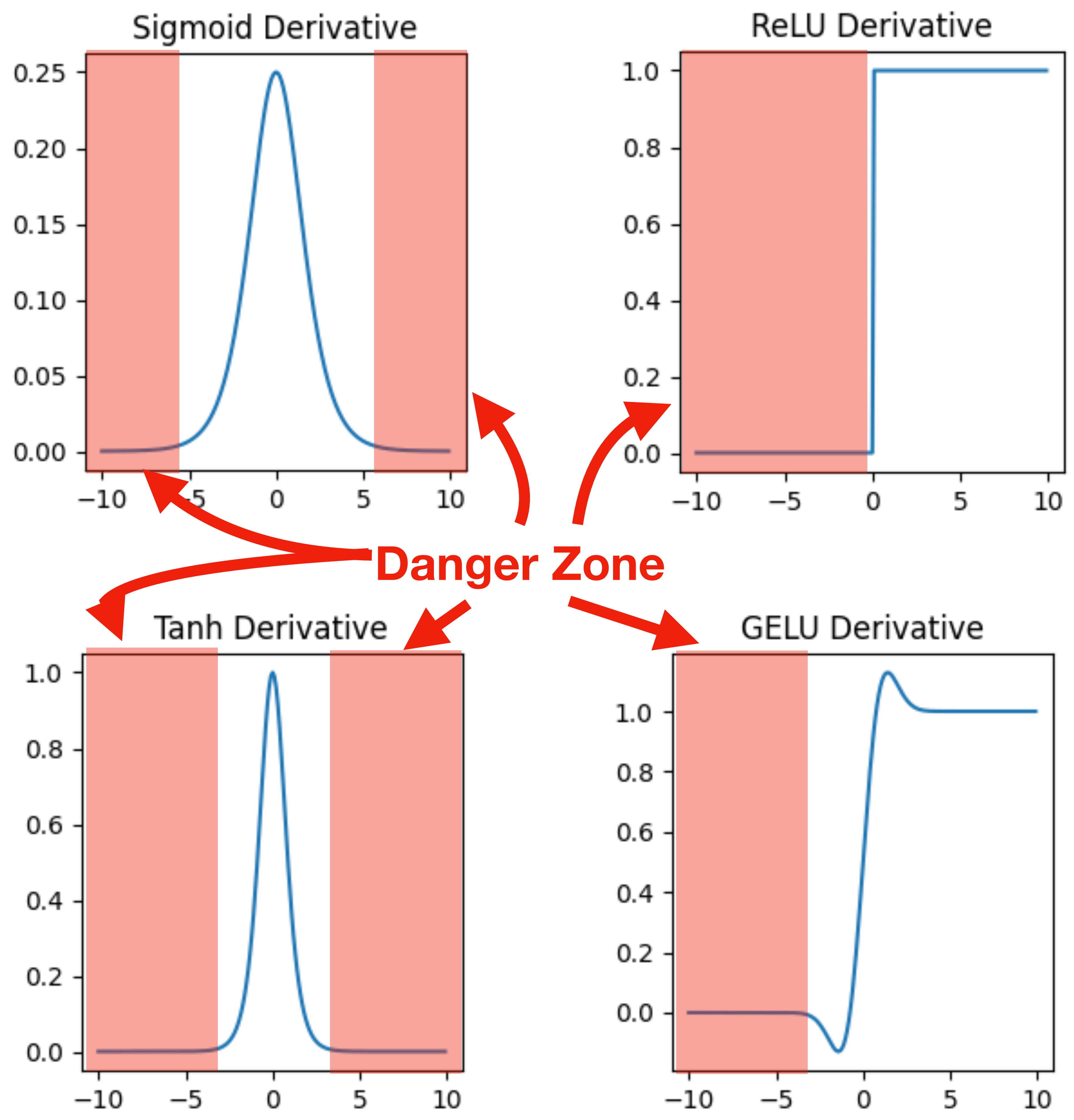
$$\delta_{h_t} = \delta_{h_{t+1}} U_h^T \odot \sigma_h'(W_h x_{t+1} + U_h h_t + b_h)$$

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

Essential components:

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

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

$x_t \in \mathbb{R}^d$: 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

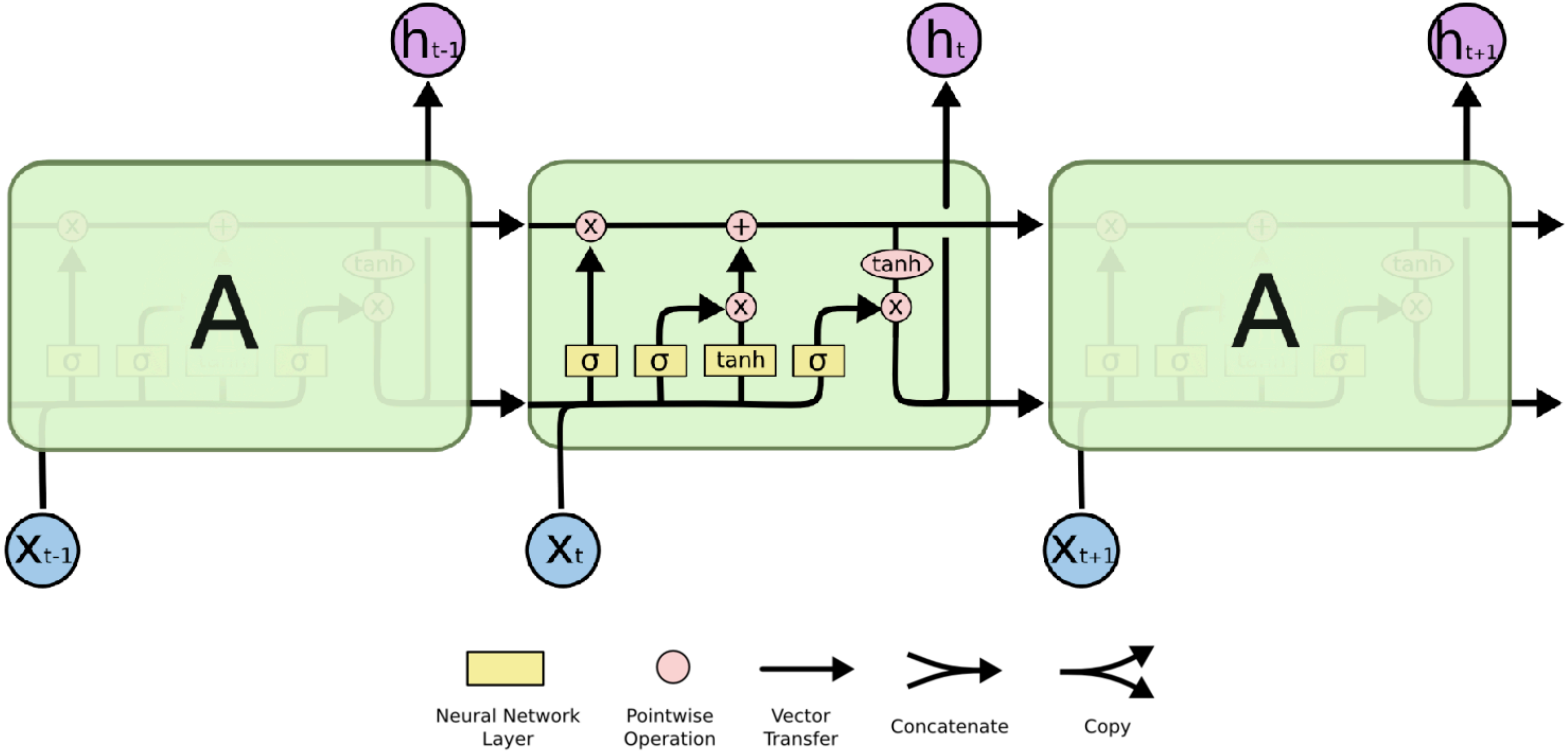
$h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit

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

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

https://en.wikipedia.org/wiki/Long_short-term_memory

LSTM architecture

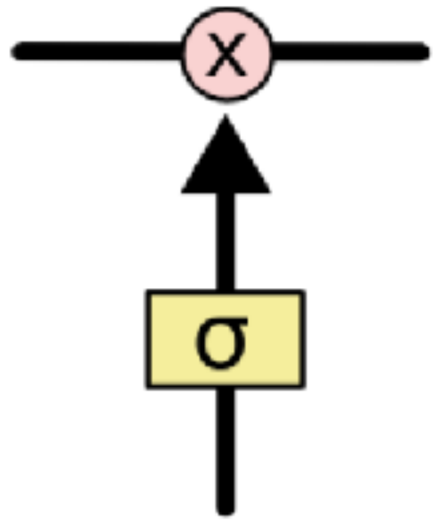


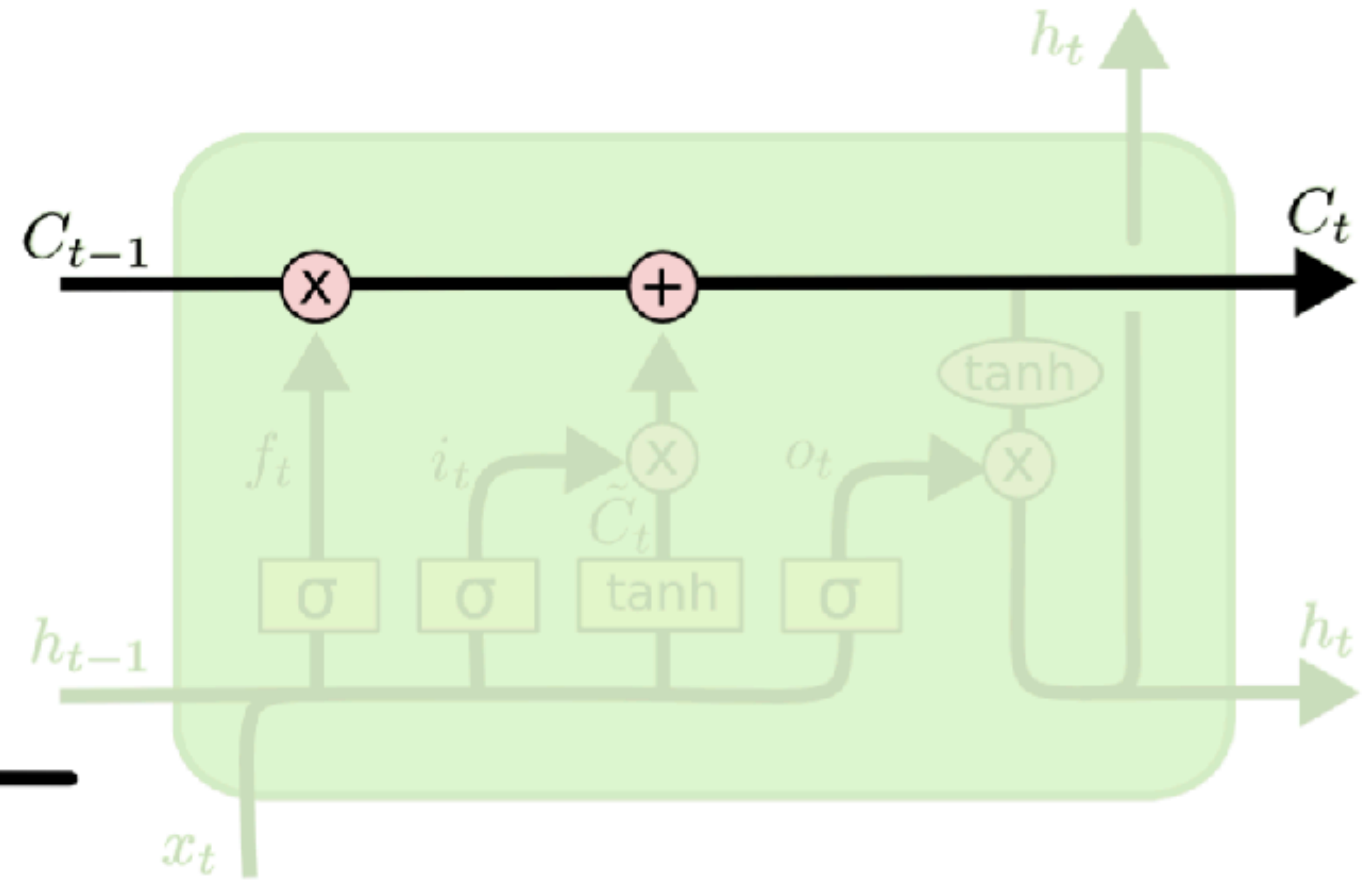
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

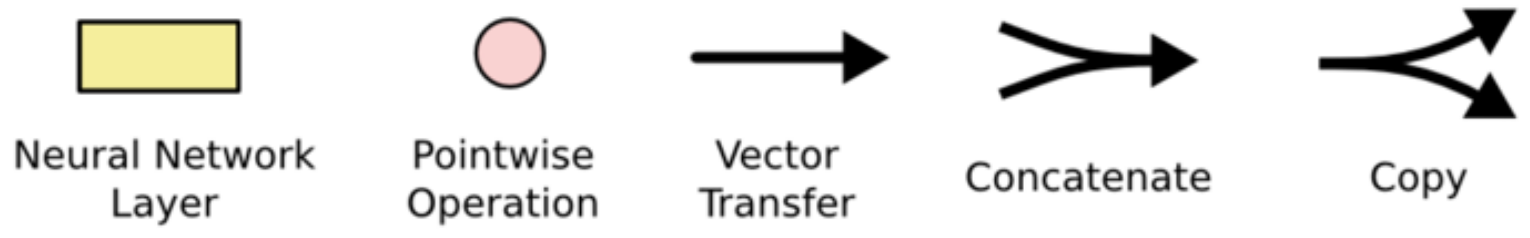
LSTM architecture

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

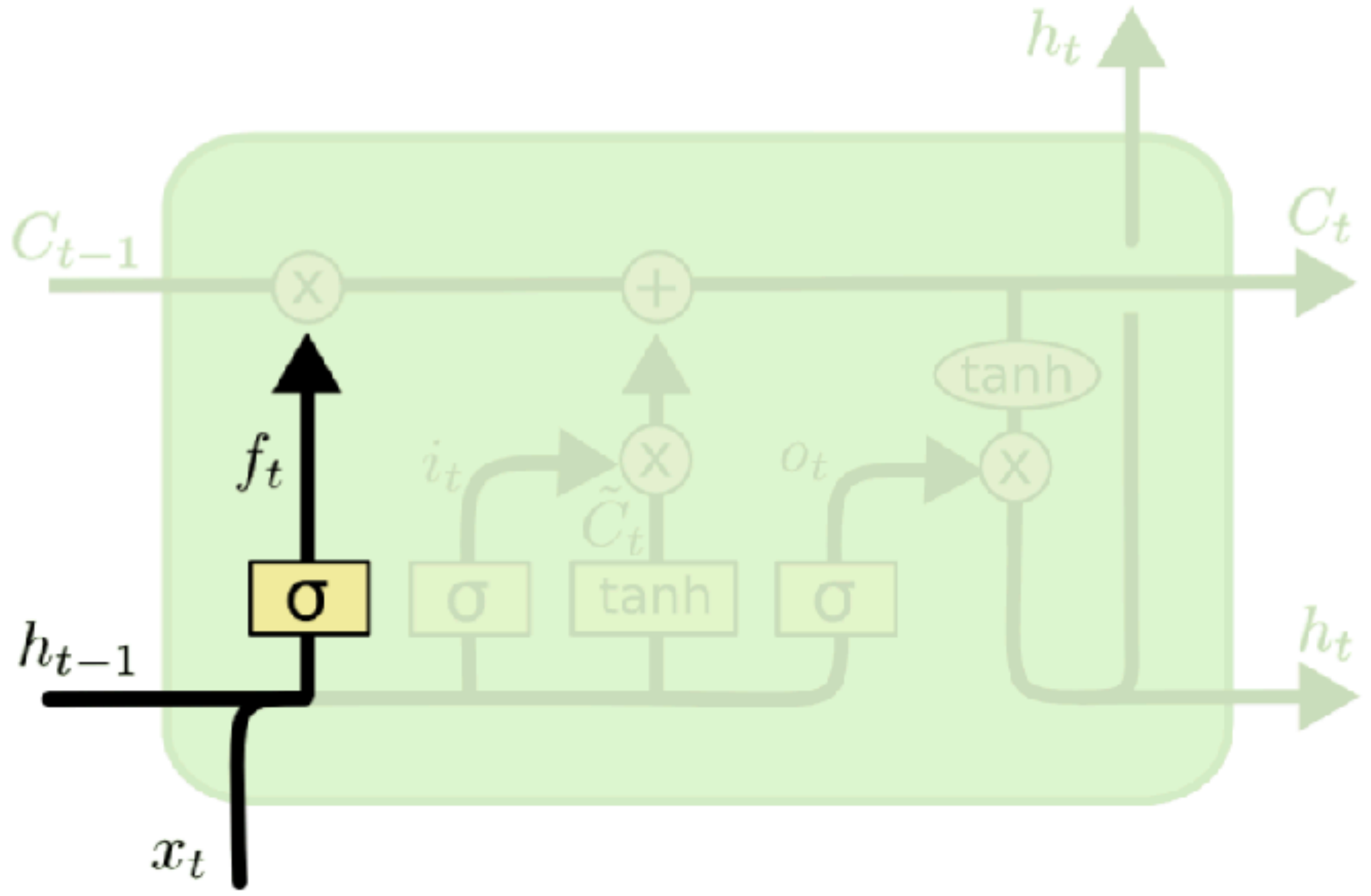
$\tilde{c}_t \in (-1,1)^h$: cell input activation vecto

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

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

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

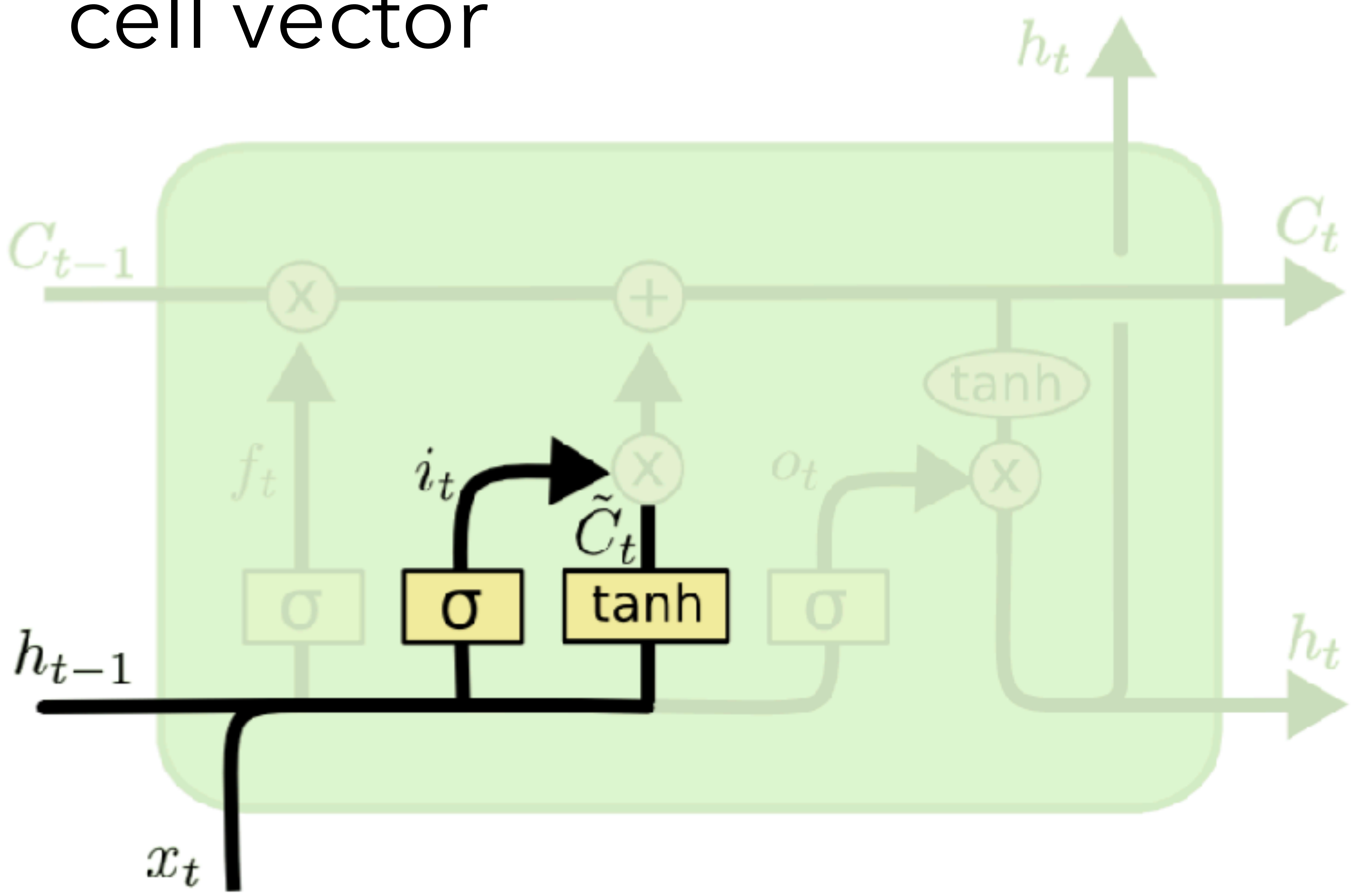
$\tilde{c}_t \in (-1,1)^h$: cell input activation vector

$c_t \in \mathbb{R}^h$: 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

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

LSTM architecture

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

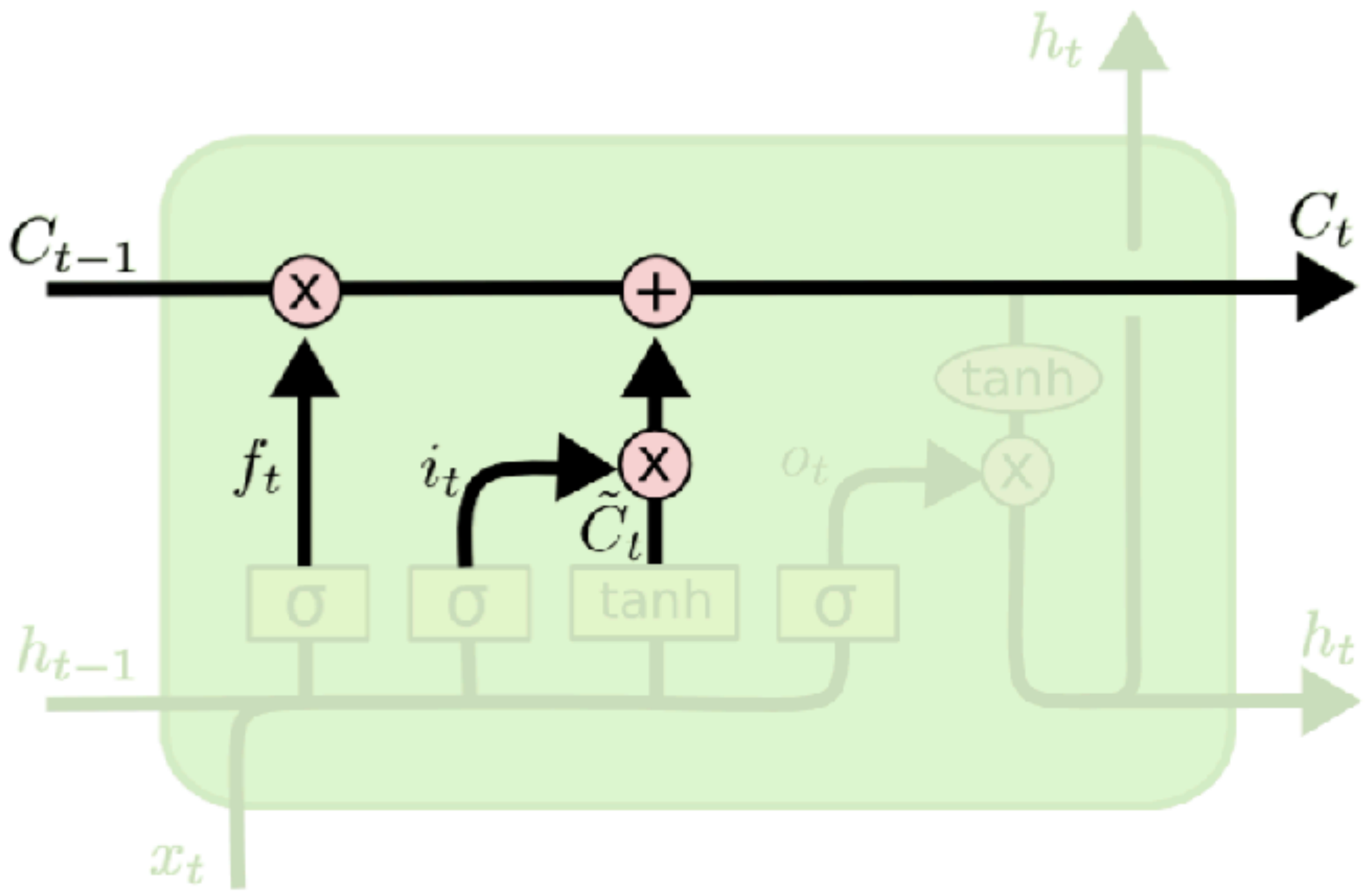
If f_t is

- 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

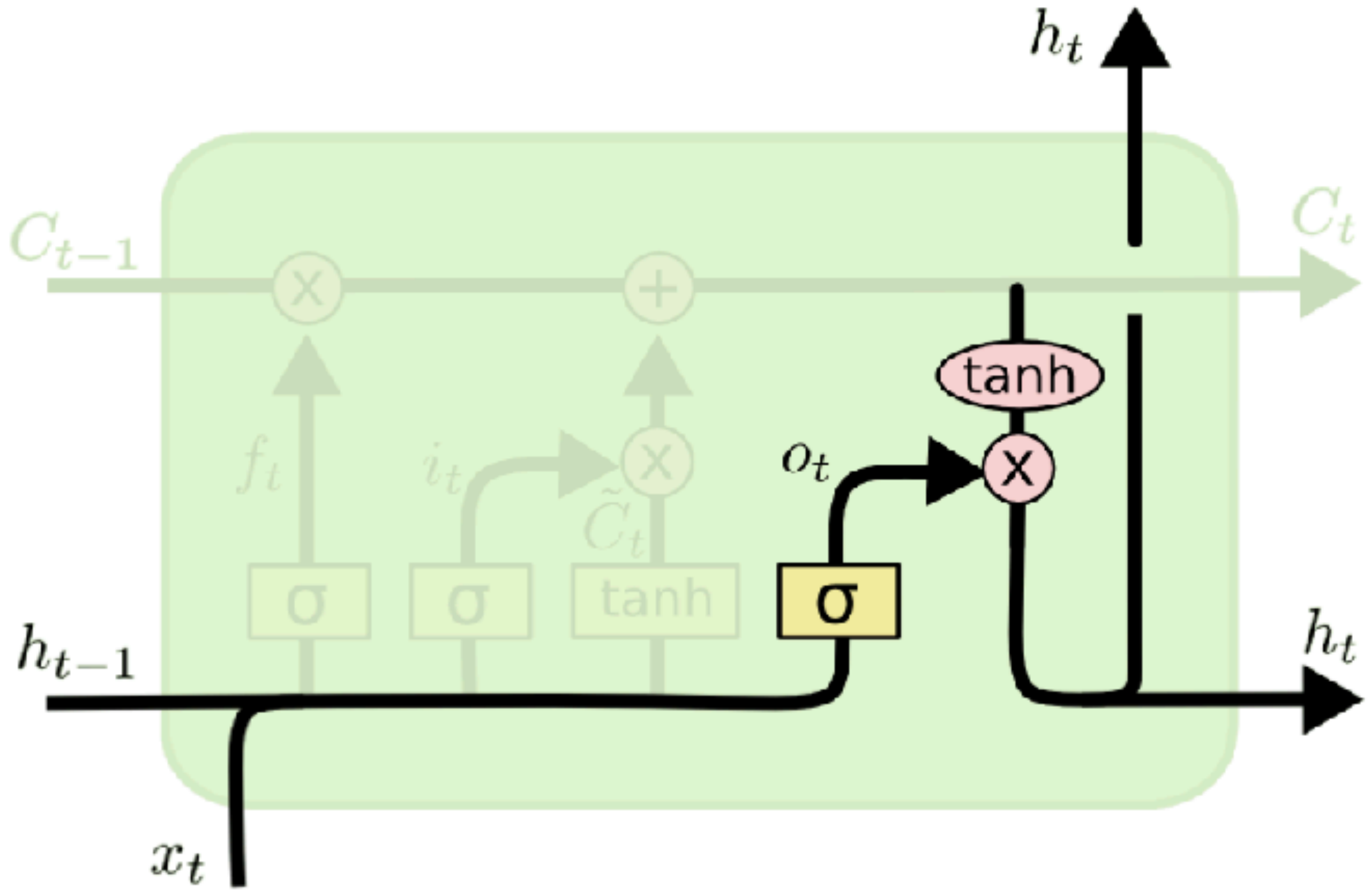
$c_t \in \mathbb{R}^h$: cell state vector

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)