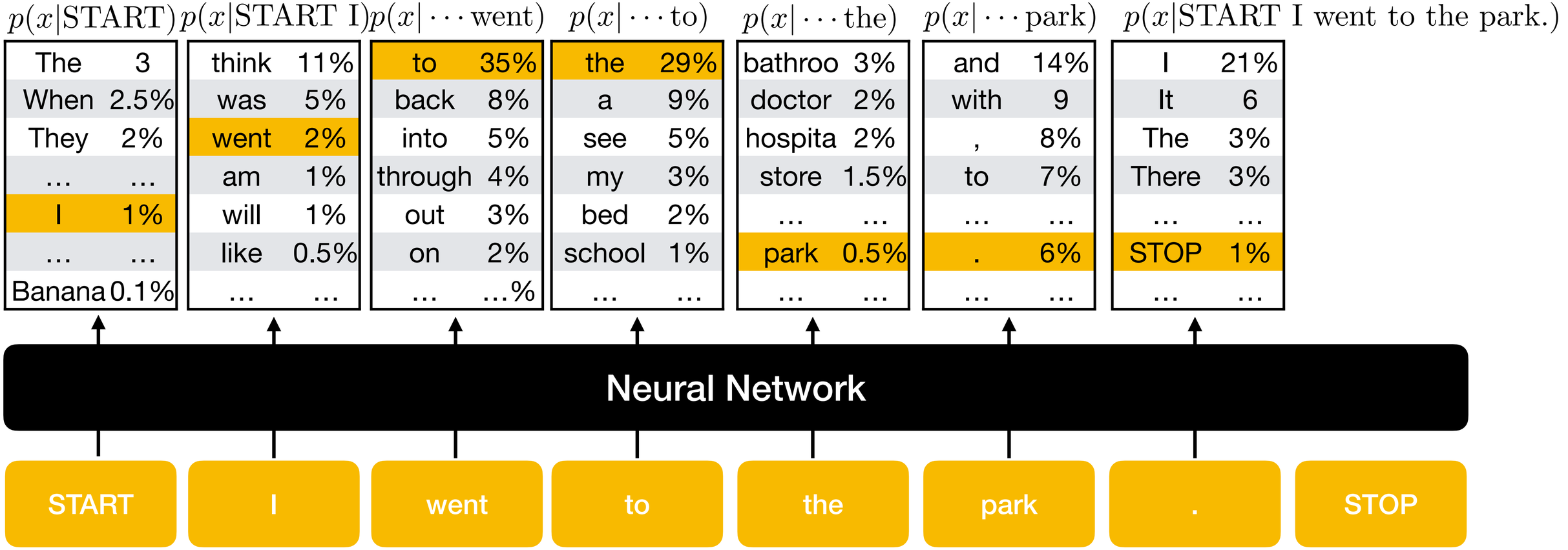
Neural Language Models

Lingpeng Kong

Department of Computer Science, The University of Hong Kong
Some materials from Stanford University CS224n with special thanks!

Neural language models: inputs/outputs

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



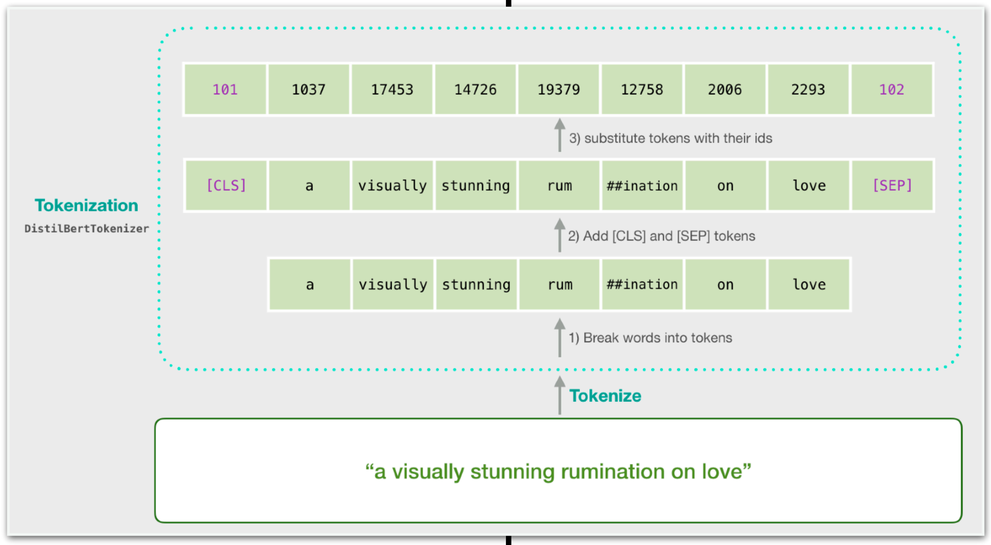
Tokenization to input vectors

$p(x|\text{START})\ p(x|\text{START I})\ p(x|\cdots\text{went})\quad p(x|\cdots\text{to})\quad p(x|\cdots\text{the})\quad p(x|\cdots\text{park})\quad p(x|\text{START I went to the park.})$

Neural Network

Mapping each tokenized id into its corresponding embeddings

Tokenization:



START | I | went | to | the | park | . | STOP

ChatGPT tokenization example

Call me Ishmael. Some years ago—never mind how long precisely—having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people’s hats off—then, I account it high time tozz get to sea as soon as I can. This is my substitute for pistol and ball. With a philosophical flourish Cato throws himself upon his sword; I quietly take to the ship. There is nothing surprising in this. If they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me.

TEXT TOKEN IDS

Tokens 239

Characters 1109

[7368, 757, 57704, 1764, 301, 13, 4427, 1667, 4227, 2345, 37593, 4059, 1268, 1317, 24559, 2345, 69666, 2697, 477, 912, 3300, 304, 856, 53101, 11, 323, 4400, 4040, 311, 2802, 757, 389, 31284, 11, 358, 3463, 358, 1053, 30503, 922, 264, 2697, 323, 1518, 279, 30125, 727, 961, 315, 279, 1917, 13, 1102, 374, 264, 1648, 358, 617, 315, 10043, 1022, 279, 87450, 268, 323, 58499, 279, 35855, 13, 43633, 358, 1505, 7182, 7982, 44517, 922, 279, 11013, 26, 15716, 433, 374, 264, 41369, 11, 1377, 73825, 6841, 304, 856, 13836, 26, 15716, 358, 1505, 7182, 4457, 3935, 6751, 7251, 985, 1603, 78766, 83273, 11, 323, 12967, 709, 279, 14981, 315, 1475, 32079, 358, 3449, 26, 323, 5423, 15716, 856, 6409, 981, 636, 1778, 459, 8582, 1450, 315, 757, 11, 430, 433, 7612, 264, 3831, 16033, 17966, 311, 5471, 757, 505, 36192, 36567, 1139, 279, 8761, 11, 323, 1749, 2740, 50244, 1274, 753, 45526, 1022, 2345, 3473, 11, 358, 2759, 433, 1579, 892, 311, 10616, 636, 311, 9581, 439, 5246, 439, 358, 649, 13, 1115, 374, 856, 28779, 369, 40536, 323, 5041, 13, 3161, 264, 41903, 67784, 356, 4428, 3872, 5678, 5304, 813, 20827, 26, 358, 30666, 1935, 311, 279, 8448, 13, 2684, 374, 4400, 15206, 304, 420, 13, 1442, 814, 719, 7020, 433, 11, 4661, 682, 3026, 304, 872, 8547, 11, 1063, 892, 477, 1023, 11, 87785, 1890, 16024, 7119, 279, 18435, 449, 757, 13]

TEXT TOKEN IDS

https://platform.openai.com/tokenizer

Vocabulary: word-level

- For the n-gram model, our vocabulary $\mathcal{V}$ was comprised of all of the words in a language
- Some problems with this:
  - $|\mathcal{V}|$ **can be quite large** - ~470,000 words Webster's English Dictionary (3rd edition)
  - **Language is changing all of the time** - 690 words were added to Merriam Webster's in September 2023 ("rizz", "goated", "mid")
  - **Long tail of infrequent words**. Many words just occur a few times
  - **Some words may not appear** in a training set of documents
  - **No modeled relationship between words** - e.g., "run", "ran", "runs", "runner" are all separate entries despite being linked in meaning

Character-level?

What about representing text with characters?

- $V = \{a, b, c, \ldots, z\}$
  - (Maybe add capital letters, punctuation, spaces, …)
- Pros:
  - Small vocabulary size ($|V| = 26$ for English)
  - Complete coverage (unseen words are represented by letters)
- Cons:
  - Encoding becomes very long - # chars instead of # words
  - Poor inductive bias for learning

~~Word~~ ~~Character~~ Subword tokenization!

How can we combine the high coverage of character-level representation with the efficiency of word-level representation?

Subword tokenization! (e.g., Byte-Pair Encoding)

- Start with character-level representations
- Build up representations from there

Original BPE Paper (Sennrich et al., 2016)

https://arxiv.org/abs/1508.07909

Byte-pair encoding: ChatGPT example

Call me Ishmael. Some years ago—never mind how long precisely—having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off—then, I account it high time tozz get to sea as soon as I can. This is my substitute for pistol and ball. With a philosophical flourish Cato throws himself upon his sword; I quietly take to the ship. There is nothing surprising in this. If they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me.

TEXT TOKEN IDS

Tokens 239

Characters 1109

[7368, 757, 57704, 1764, 301, 13, 4427, 1667, 4227, 2345, 37593, 4059, 1268, 1317, 24559, 2345, 69666, 2697, 477, 912, 3300, 304, 856, 53101, 11, 323, 4400, 4040, 311, 2802, 757, 389, 31284, 11, 358, 3463, 358, 1053, 30503, 922, 264, 2697, 323, 1518, 279, 30125, 727, 961, 315, 279, 1917, 13, 1102, 374, 264, 1648, 358, 617, 315, 10043, 1022, 279, 87450, 268, 323, 58499, 279, 35855, 13, 43633, 358, 1505, 7182, 7982, 44517, 922, 279, 11013, 26, 15716, 433, 374, 264, 41369, 11, 1377, 73825, 6841, 304, 856, 13836, 26, 15716, 358, 1505, 7182, 4457, 3935, 6751, 7251, 985, 1603, 78766, 83273, 11, 323, 12967, 709, 279, 14981, 315, 1475, 32079, 358, 3449, 26, 323, 5423, 15716, 856, 6409, 981, 636, 1778, 459, 8582, 1450, 315, 757, 11, 430, 433, 7612, 264, 3831, 16033, 17966, 311, 5471, 757, 505, 36192, 36567, 1139, 279, 8761, 11, 323, 1749, 2740, 50244, 1274, 753, 45526, 1022, 2345, 3473, 11, 358, 2759, 433, 1579, 892, 311, 10616, 636, 311, 9581, 439, 5246, 439, 358, 649, 13, 1115, 374, 856, 28779, 369, 40536, 323, 5041, 13, 3161, 264, 41903, 67784, 356, 4428, 3872, 5678, 5304, 813, 20827, 26, 358, 30666, 1935, 311, 279, 8448, 13, 2684, 374, 4400, 15206, 304, 420, 13, 1442, 814, 719, 7020, 433, 11, 4661, 682, 3026, 304, 872, 8547, 11, 1063, 892, 477, 1023, 11, 87785, 1890, 16024, 7119, 279, 18435, 449, 757, 13]

TEXT TOKEN IDS

https://platform.openai.com/tokenizer

Byte-pair encoding: usage

- Basically state of the art in tokenization
- Used in all modern left-to-right large language models (LLMs), including ChatGPT

Model/Tokenizer	Vocabulary Size
GPT-3.5/GPT-4/ChatGPT	100k
GPT-2/GPT-3	50k
Llama2	32k
Falcon	65k

Byte-pair encoding (BPE): algorithm

Required:

- Documents $\mathcal{D}$
- Desired vocabulary size N (greater than characters 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 $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

Byte-pair encoding: example

Required:

- Documents $\mathcal{D}$ → $\mathcal{D} = \{\text{“i hug pugs”}, \text{“hugging pugs is fun”}, \text{“i make puns”}\}$
- Desired vocabulary size N (greater than chars in $\mathcal{D}$) → $N = 20$

Algorithm:

- Pre-tokenize $\mathcal{D}$ by splitting into words (split before whitespace/punctuation) → $\mathcal{D} = \{\text{“i”}, \text{“ hug”}, \text{“ pugs”}, \text{“hugging”}, \text{“ pugs”}, \text{“ is”}, \text{“ fun”}, \text{“i”}, \text{“ make”}, \text{“ puns”}\}$
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$ → $\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}\}, |\mathcal{V}| = 13$
- Convert $\mathcal{D}$ into a list of tokens (characters) → $\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$
- While $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

Example inspired by: https://huggingface.co/docs/transformers/tokenizer_summary

Byte-pair encoding: example

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 $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{V} = \{1 : \text{`\ '}, 2 : \text{`a'}, 3 : \text{`e'}, 4 : \text{`f'}, 5 : \text{`g'}, 6 : \text{`h'}, 7 : \text{`i'}, 8 : \text{`k'}, 9 : \text{`m'}, 10 : \text{`n'}, 11 : \text{`p'}, 12 : \text{`s'}, 13 : \text{`u'}\}$$

Implementation aside: We normally store $\mathcal{D}$ with the token indices instead of the text itself!

$$\mathcal{D} = \{ [7], [1, 6, 13, 5], [1, 11, 13, 5, 12], [6, 13, 5, 5, 7, 10, 5], [1, 11, 13, 5, 12], [1, 7, 12], [1, 4, 13, 10], [7], [1, 9, 2, 8, 3], [1, 11, 13, 10, 12]\}$$

For legibility of the example, we will show the text corresponding to each token

Byte-pair encoding: example

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 $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}] \}$$

Bigram	Count
‘u’,’g’	4
‘p’, ‘u’	3
‘ ‘, ‘p’	3
‘h’, ‘u’	2
…	…

$$v_{14} := \text{concat}(\text{‘u’}, \text{‘g’}) = \text{‘ug’}$$

Byte-pair encoding: example

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 $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}],$$
$$[\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

$$v_{14} := \text{concat}(\text{‘u’}, \text{‘g’}) = \text{‘ug’}$$

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’},$$
$$\text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}\}, |\mathcal{V}| = 14$$

Byte-pair encoding: example

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 $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}] \}$$

Bigram	Count
‘ ‘, ‘p’	3
‘p’, ‘ug’	2
‘ug’, ’s'	2
‘u’, ’n'	2
…	…

$$v_{15} := \text{concat}(\text{‘ ’}, \text{‘p’}) = \text{‘ p’}$$

Byte-pair encoding: example

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 $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

$$v_{15} := \text{concat}(\text{‘ ’}, \text{‘p’}) = \text{‘ p’}$$

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ p’}, \text{‘ug’}, \text{‘s’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’},$$
$$\text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}, \text{‘ p’}\}, |\mathcal{V}| = 15$$

Byte-pair encoding: example

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 $|\mathcal{V}| < N$:
  - Let $n := |\mathcal{V}| + 1$
  - Get counts of all bigrams in $\mathcal{D}$
  - For the most frequent bigram v_i, v_j (breaking ties arbitrarily)
    - Let $v_n := \text{concat}(v_i, v_j)$
    - Change all instances in $\mathcal{D}$ of v_i, v_j to v_n and add v_n to $\mathcal{V}$

Repeat until $|\mathcal{V}| = N$...

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ hug’}], [\text{‘ pugs’}], [\text{‘hug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ pugs’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘un’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ p’}, \text{‘un’}, \text{‘s’}] \}$$

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}, \text{‘ p’}, \text{‘hug’}, \text{‘ pug’}, \text{‘ pugs’}, \text{‘un’}, \text{‘ hug’}\},$$

$$|\mathcal{V}| = 20$$

CHANGES FROM START

Byte-pair encoding: example

CHANGES FROM START

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ hug’}], [\text{‘ pugs’}],$$
$$[\text{‘hug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ pugs’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘un’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ p’}, \text{‘un’}, \text{‘s’}]\}$$

Questions to think about:

- Is every token we made used in the corpus? Why or why not?
- How much memory (#tokens) have we saved for each document?
- What would happen if you kept adding vocabulary until you couldn’t anymore?

$$\mathcal{D} = \{ [7], [20], [18],$$
$$[16, 5, 7, 10, 5], [18],$$
$$[1, 7, 12], [1, 4, 19], [7],$$
$$[1, 9, 2, 8, 3], [15, 19, 12]\}$$

(as tokens indices)

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

Byte-pair encoding: tokenization/encoding

With this vocabulary, can you represent (or, tokenize/encode):

- “apple”?
  - No, there is no ‘l’ in the vocabulary
- “huge”?
  - Yes - [16, 3]
- “ huge”?
  - Yes - [20, 4]
- “ hugest”?
  - No, there is no ‘t’ in the vocabulary
- “unassumingness”?
  - Yes - [19, 2, 12, 12, 13, 9, 7, 10, 5, 10, 3, 12, 12]

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

Byte-pair encoding: tokenization/encoding

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

- Sometimes, there may be more than one way to represent a word with the vocabulary…
  - E.g., “ hugs” = [20, 12] = [1, 16, 12] = [1, 6, 14, 12] = [1, 6, 13, 5, 13]
    - Which is the best representation? Why?

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

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

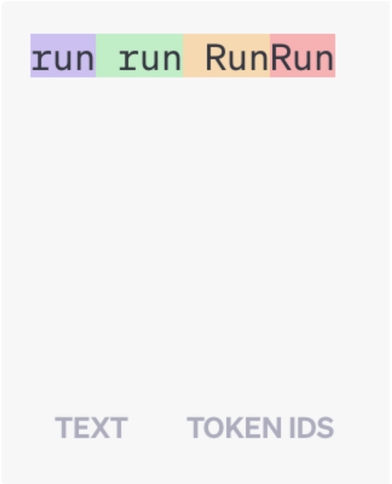
$$\text{Decode}([20, 12]) = \text{“ hugs”}$$
$$\text{Decode}([9, 7, 12, 12, 6, 2, 11, 3, 10, 10, 3, 12, 12]) = \text{“misshapenness”}$$

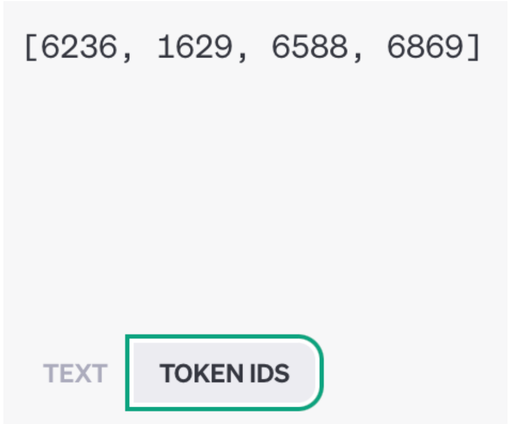
Byte-pair encoding: properties

- 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

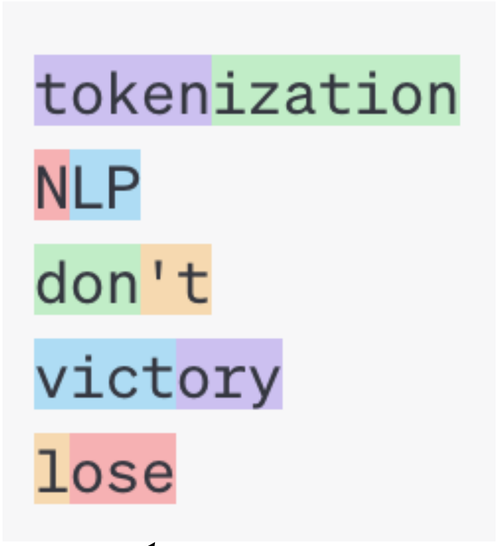
- Token != word
- Spaces are part of token
  - “run” is a different token than “ run”
- Not invariant to case changes
  - “Run” is a different token than “run”

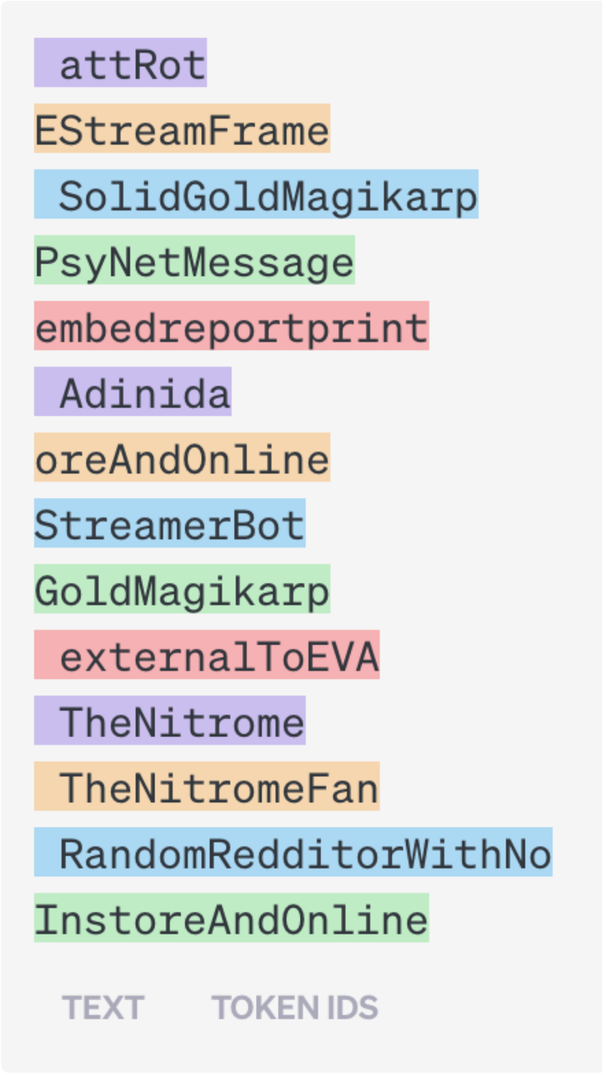




Weird properties of tokenizers

- 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”)
  - 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
  - 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)

space

no space

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)

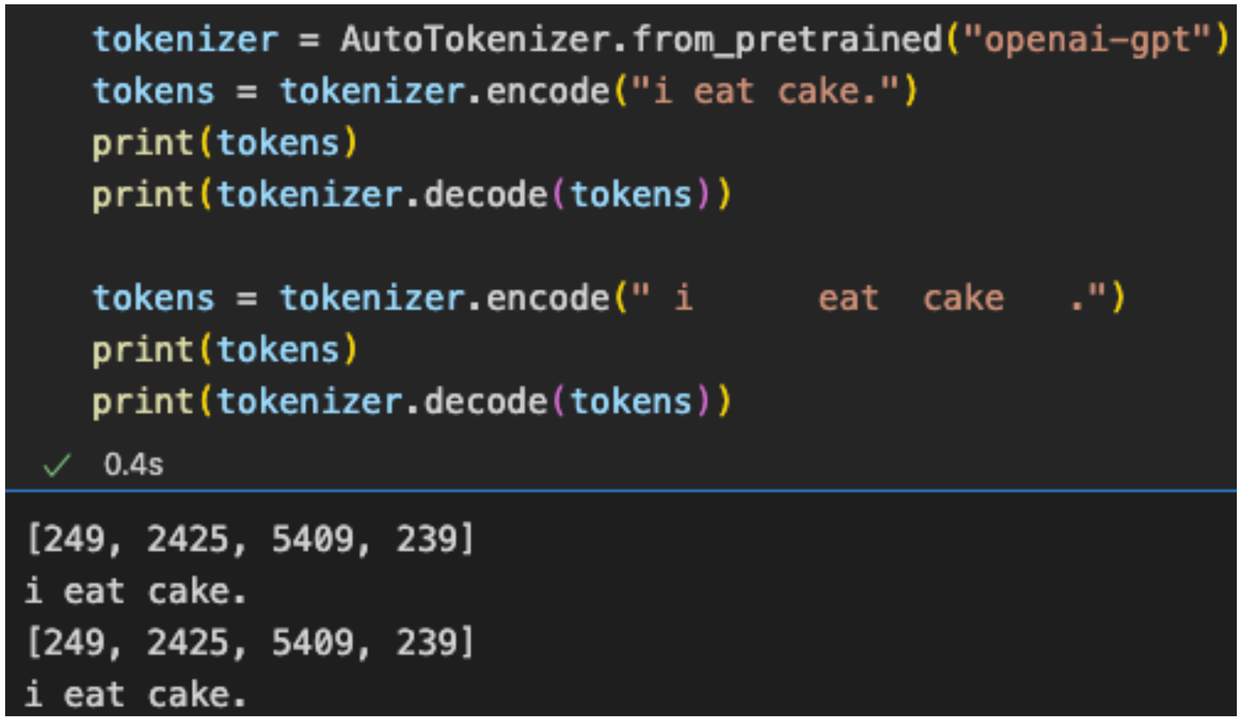
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$

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?



(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 words/characters to become tokens

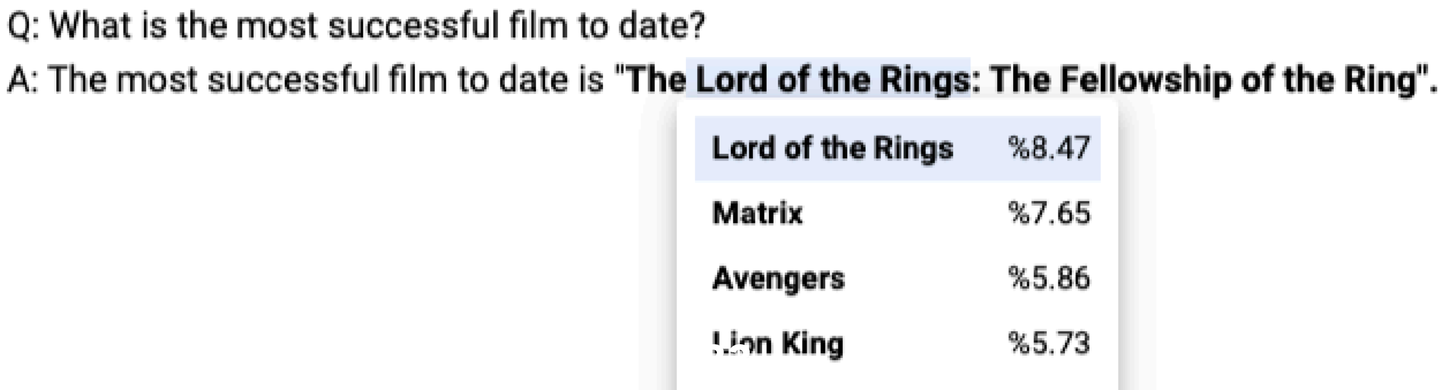
SentencePiece paper example in Japanese:

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 $\mathcal{D}$

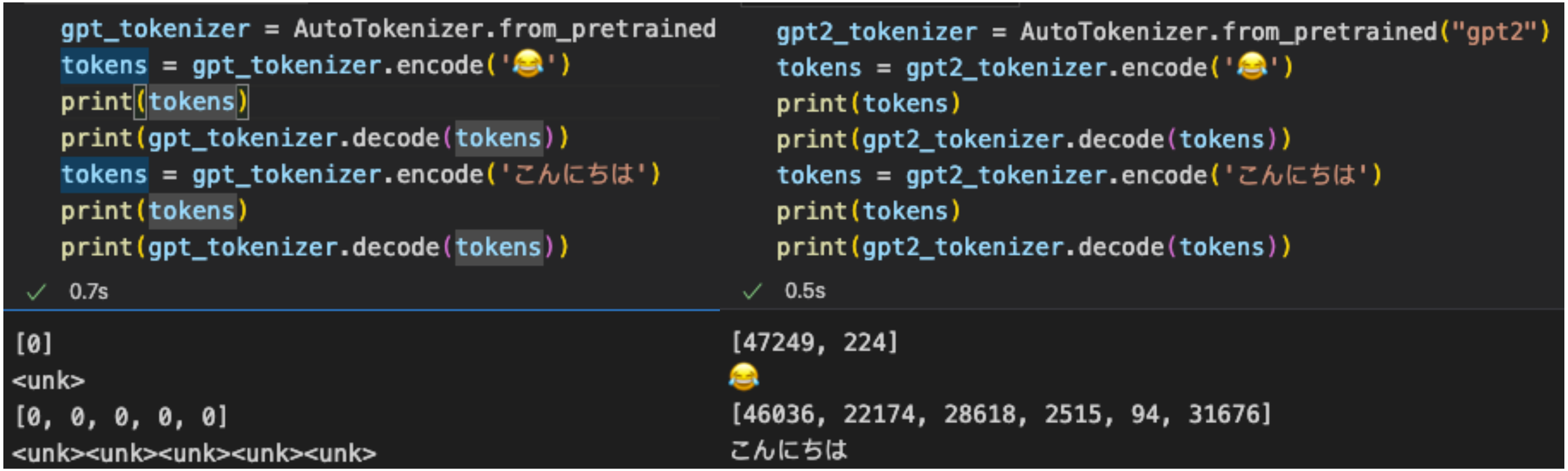
+ Convert $\mathcal{D}$ into a list of tokens (**bytes**)

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

Variants: byte-based

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

The Byte-based GPT-2 tokenizer succeeds!



Variants: WordPiece objective

- To merge, we selected the bigram with highest frequency
  - 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)

…

- 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)

…

+ 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

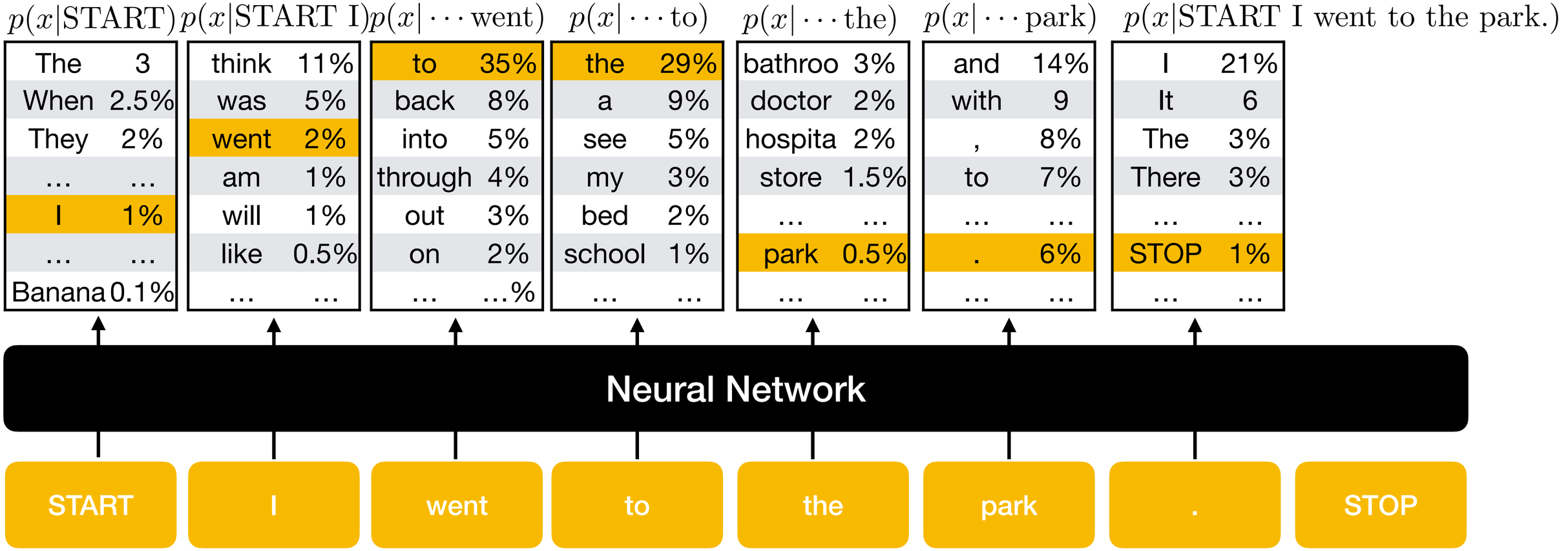
Examples of LLMs and their tokenizers

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

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?

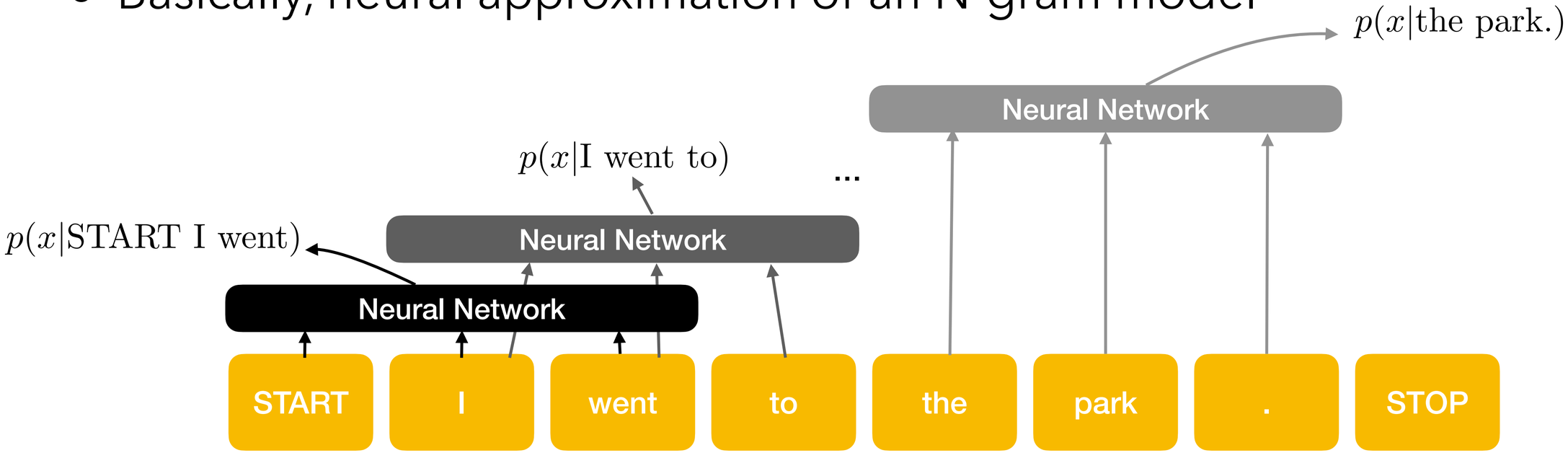
- Don’t neural networks need a fixed-size vector as input? And isn’t text variable length?

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

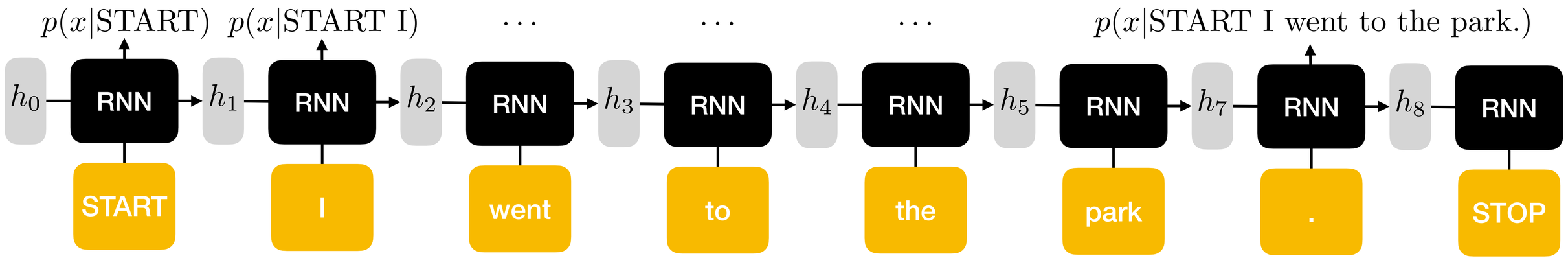


Recurrent neural networks

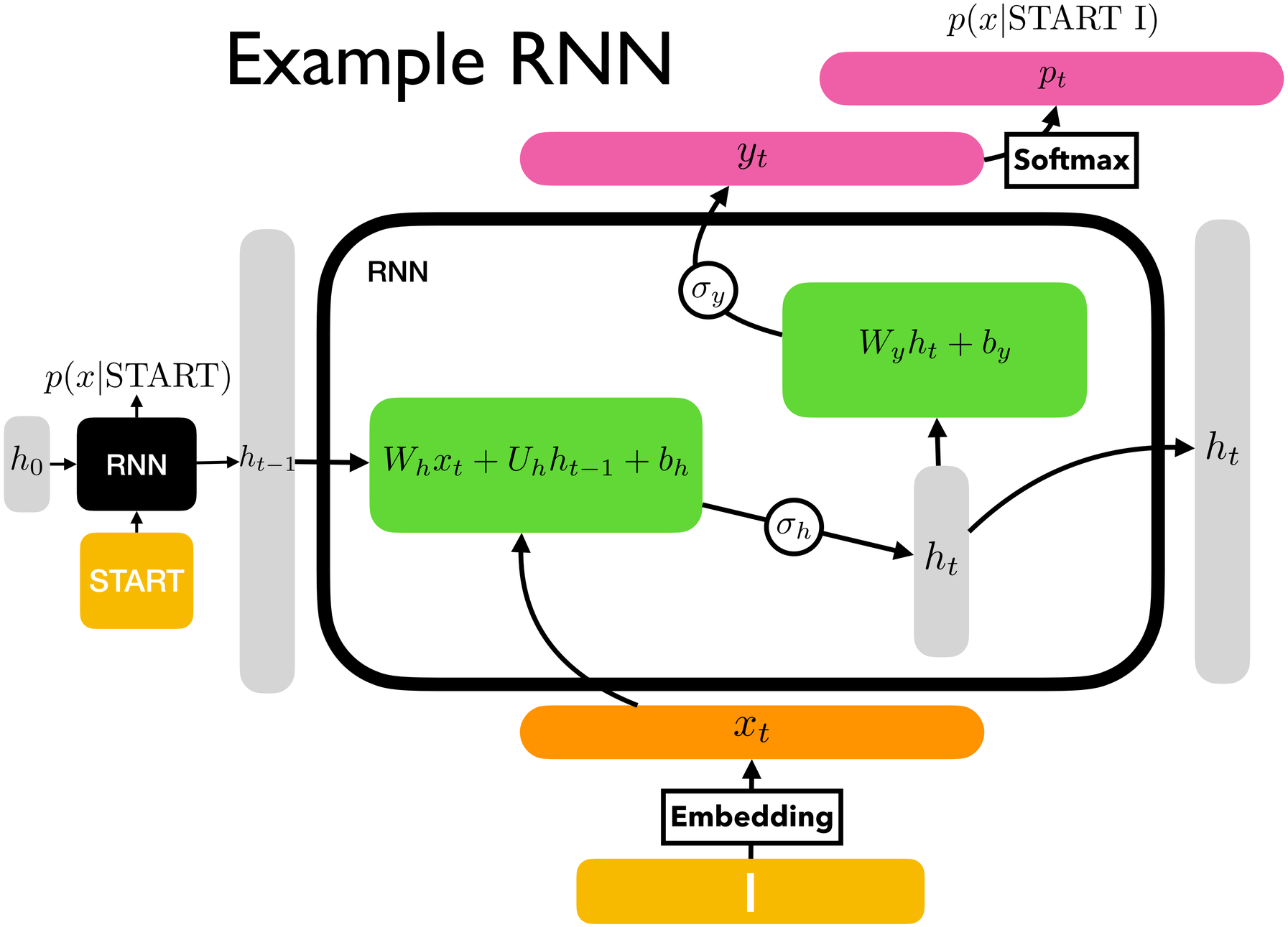
Idea 2: Recurrent Neural Networks (RNNs)

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

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

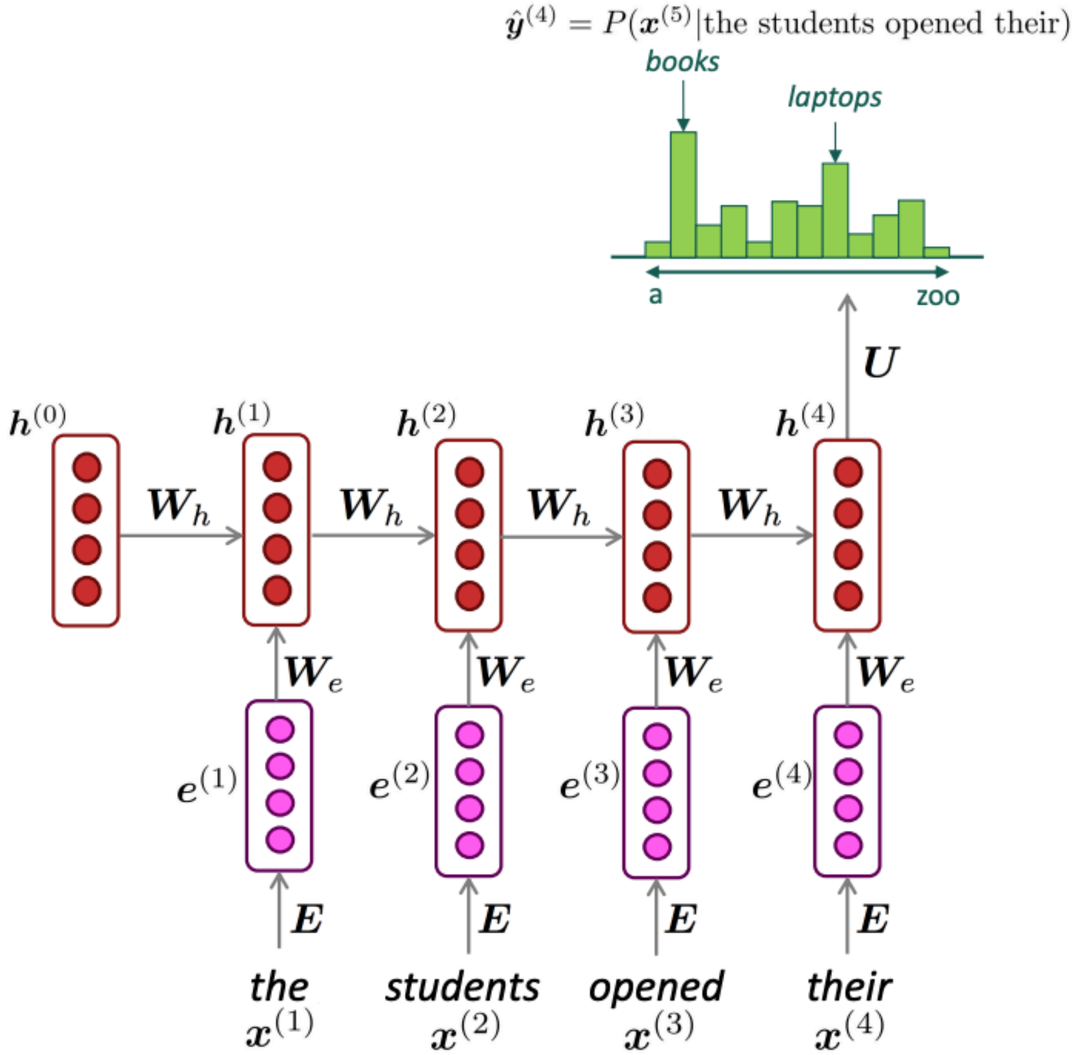
$\boldsymbol{h}^{(0)}$ is the initial hidden state

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!*

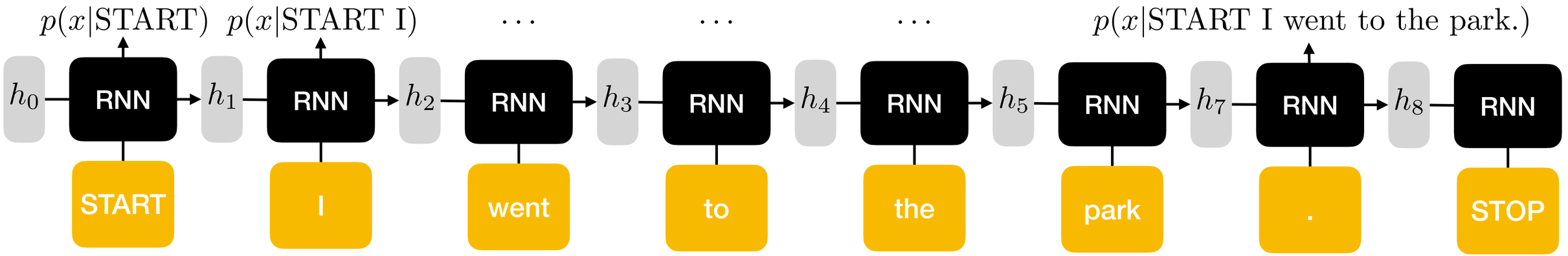
Recurrent neural networks

- 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

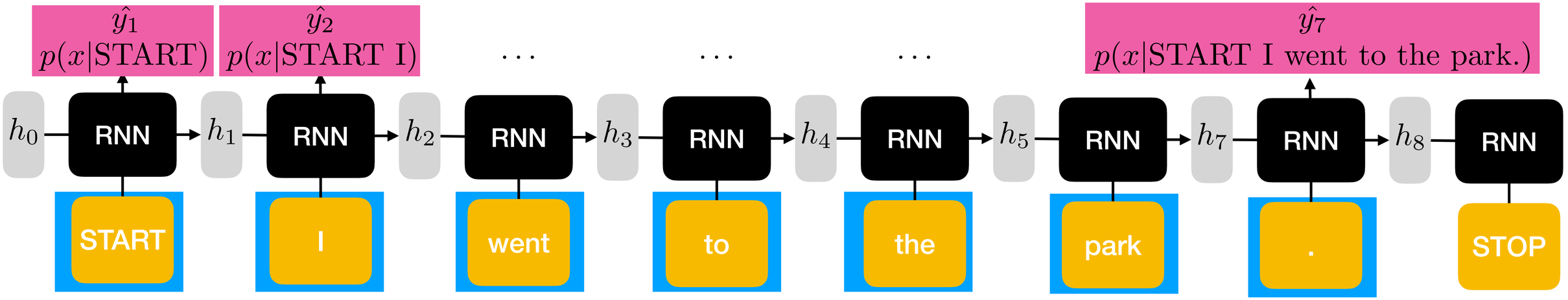
Training procedure

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)

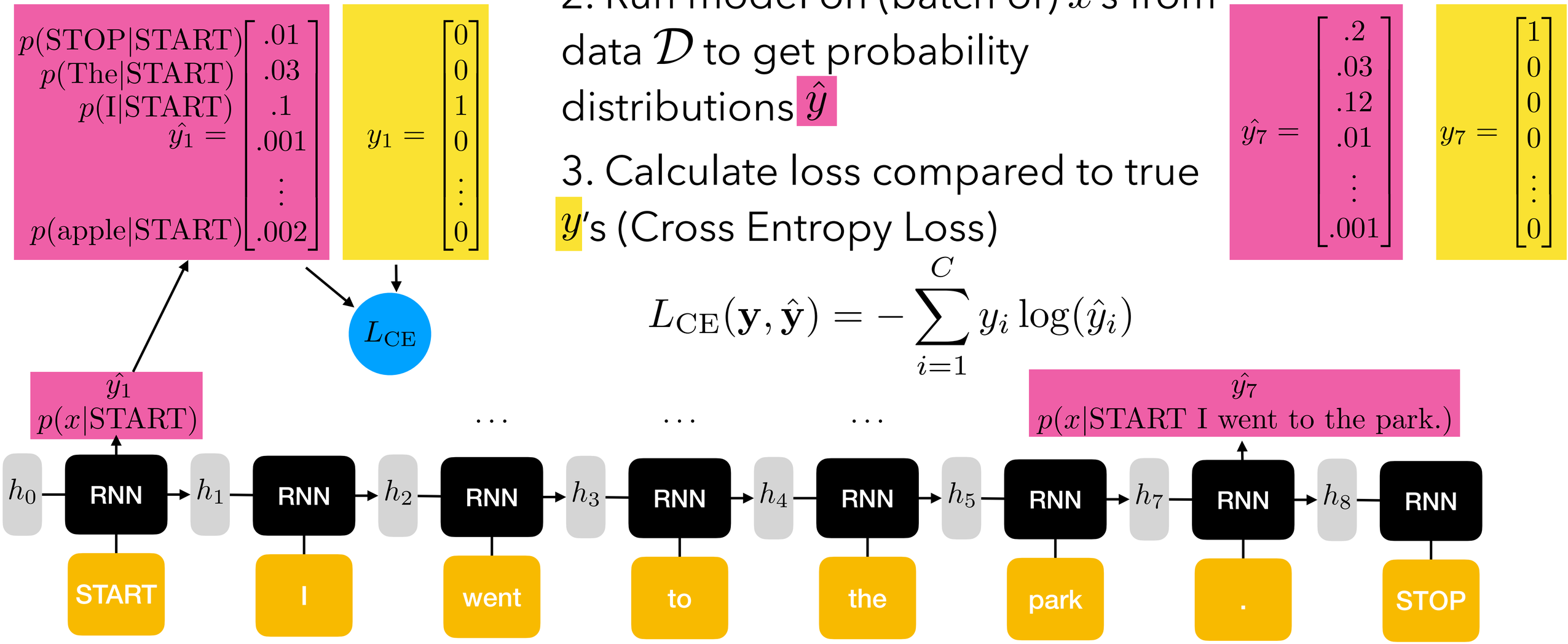


Training procedure

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

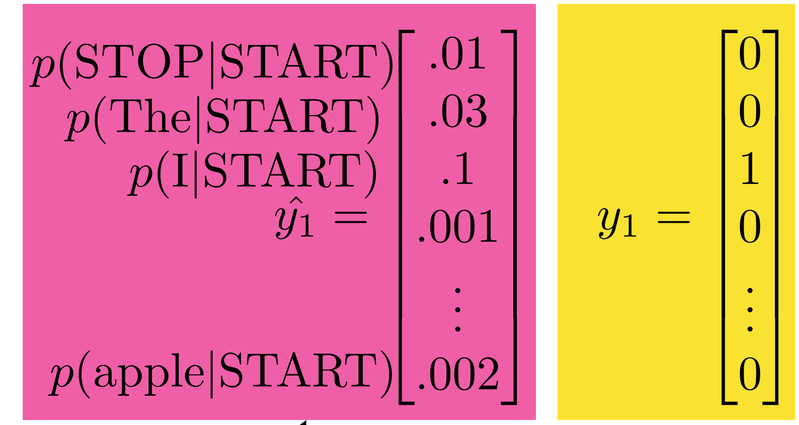
3. Calculate loss compared to true y's (Cross Entropy Loss)

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



Training procedure

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



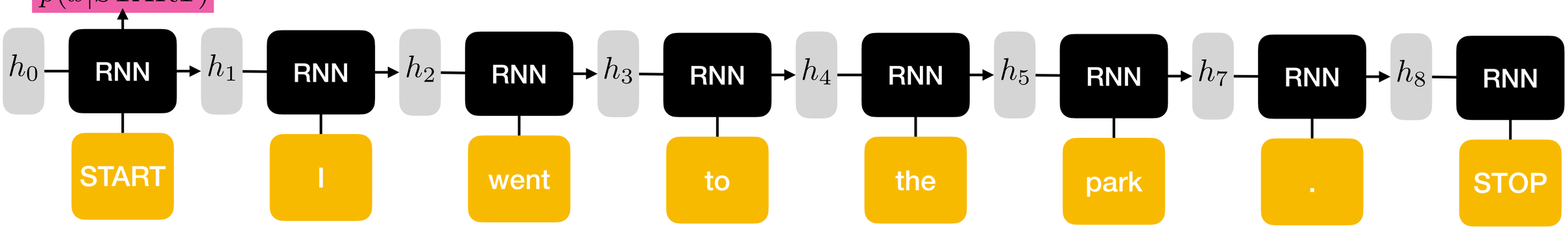
(Actual observed word)

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

L_{CE}

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

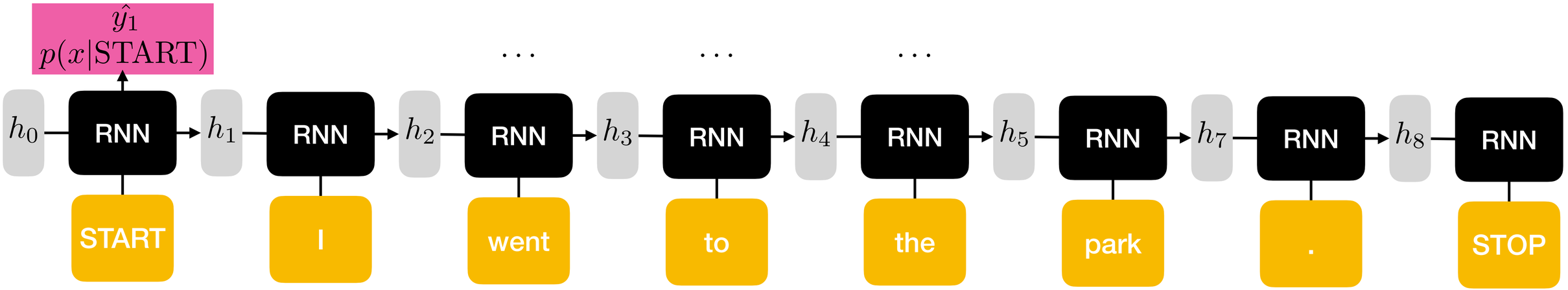
$\hat{y}_1$
$p(x|\text{START})$



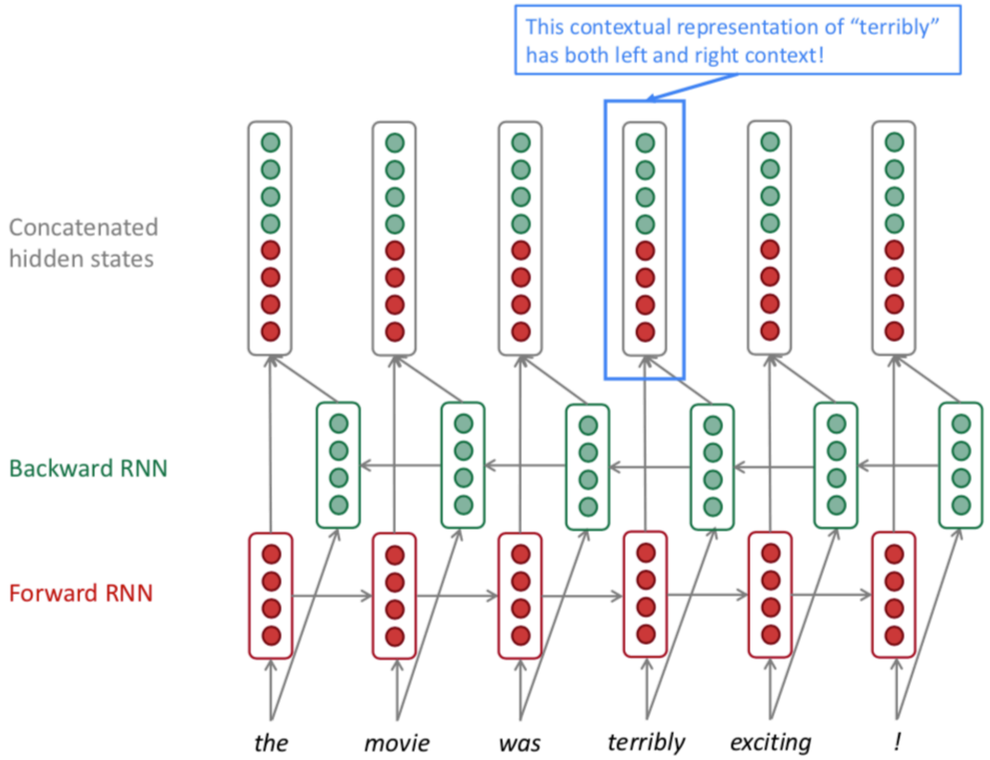
Training procedure - gradient descent step

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



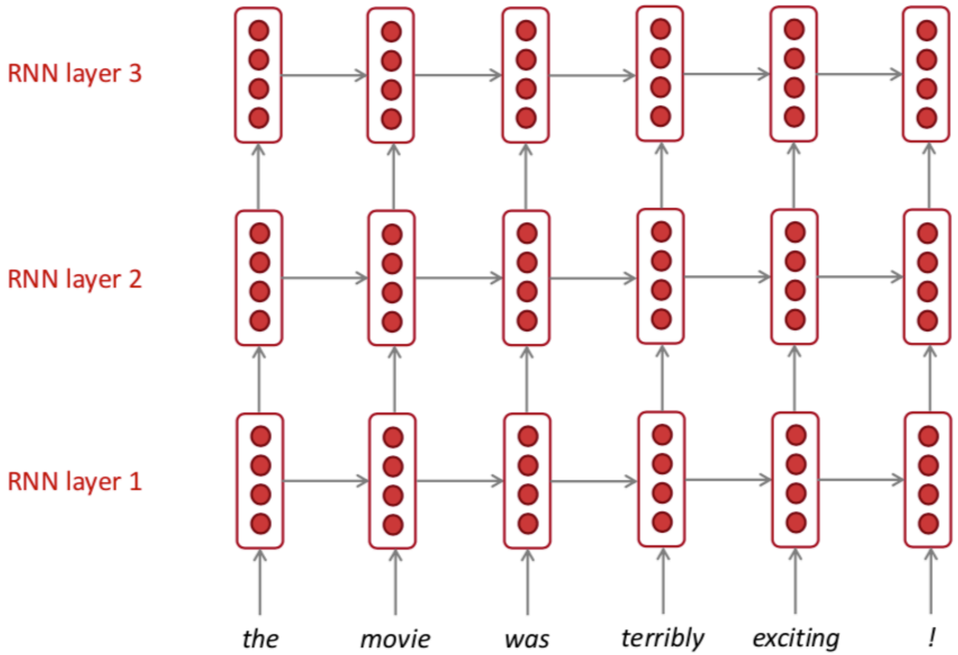
$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^h$$

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

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

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

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

RNNs - vanishing gradient problem

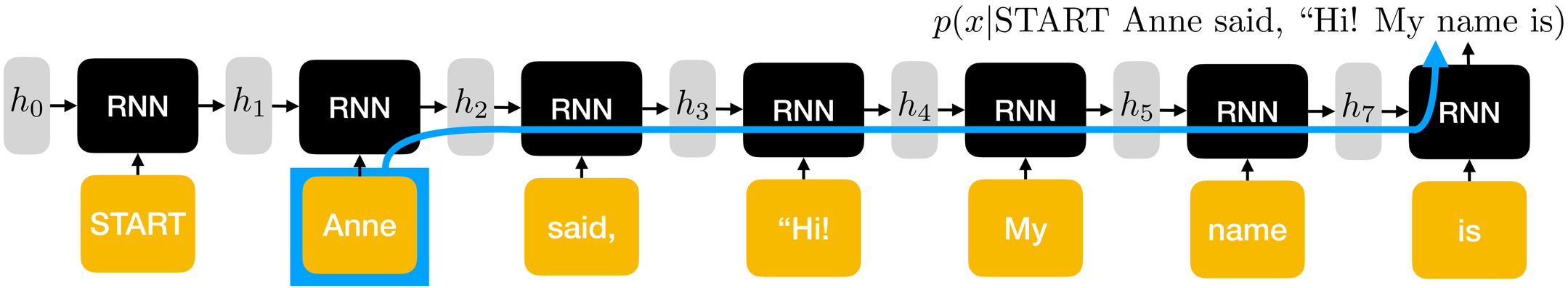
What word is likely to come next for this sequence?

Anne said, “Hi! My name is

RNNs - vanishing gradient problem

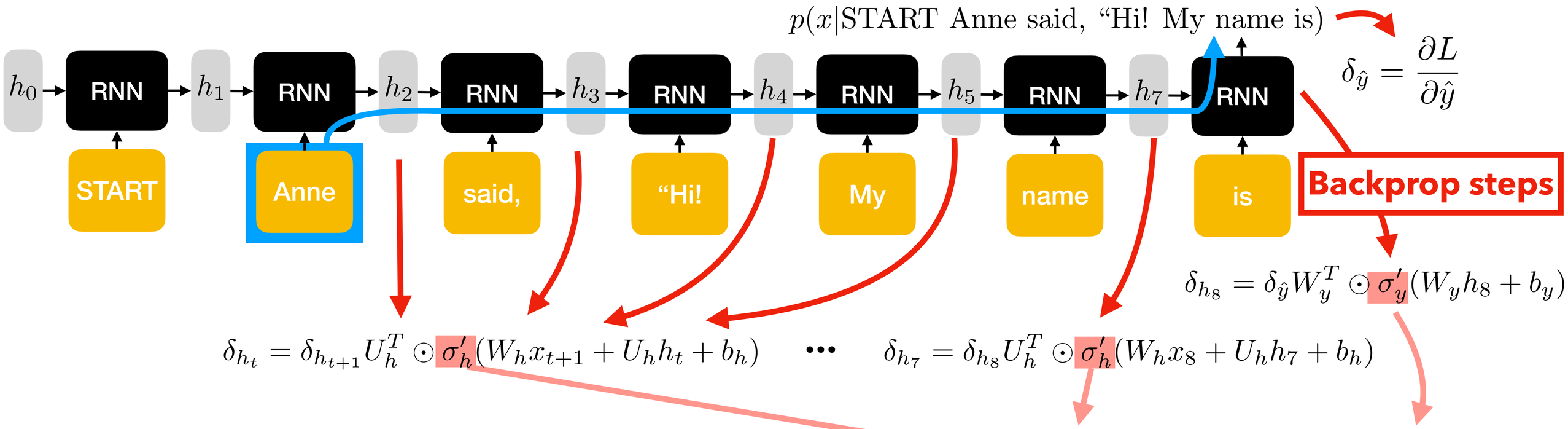
What word is likely to come next for this sequence?

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



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

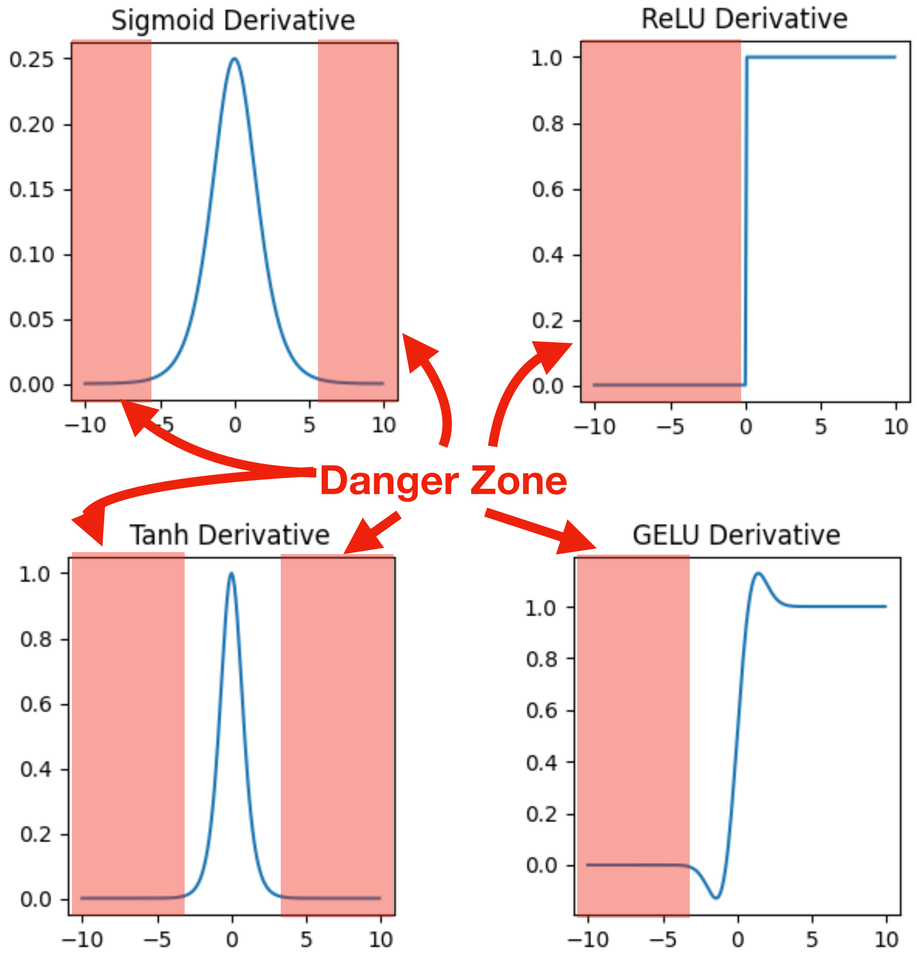
$$\delta_{h_t} = \delta_{h_{t+1}} U_h^T \odot \sigma_h'(W_h x_{t+1} + U_h h_t + b_h)$$

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

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

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

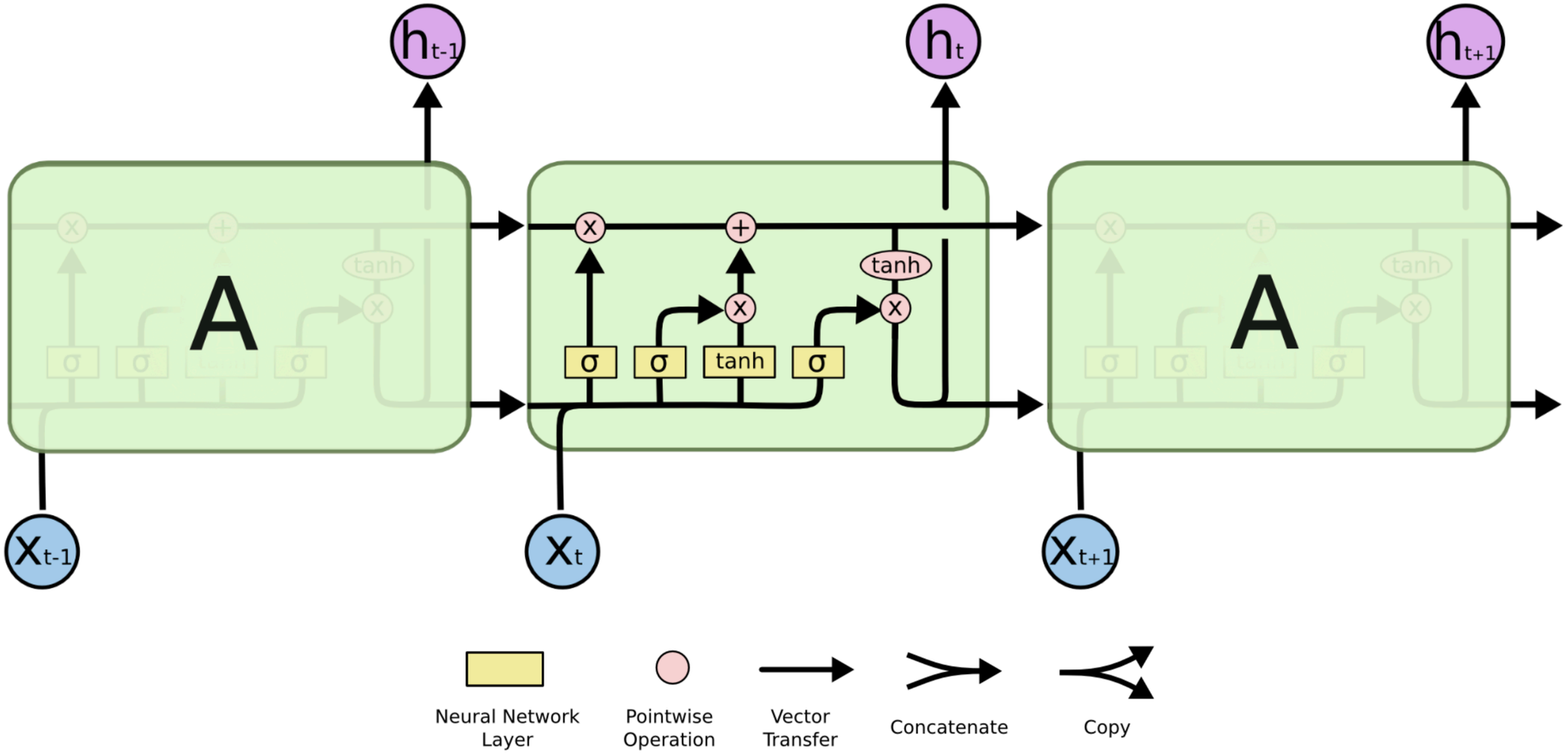
$h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit

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

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

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

LSTM architecture

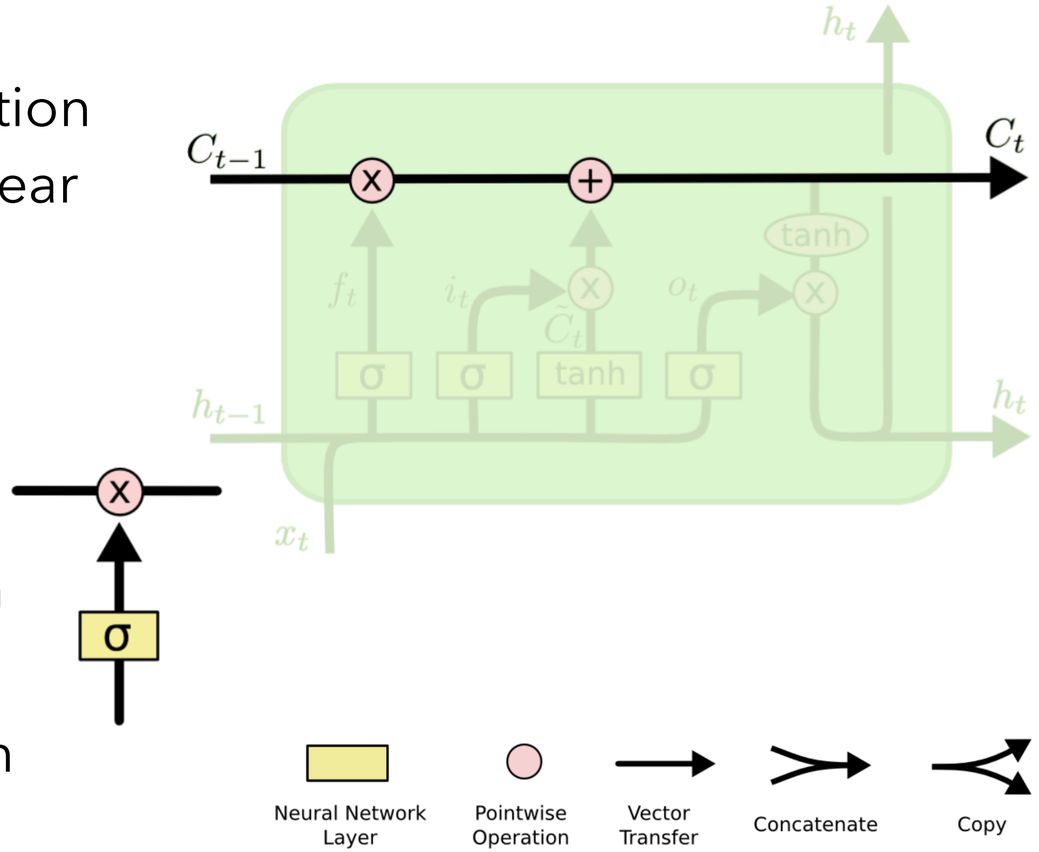


https://colah.github.io/posts/2015-08-Understanding-LSTMs/

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 vect

$i_t \in (0,1)^h$: 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

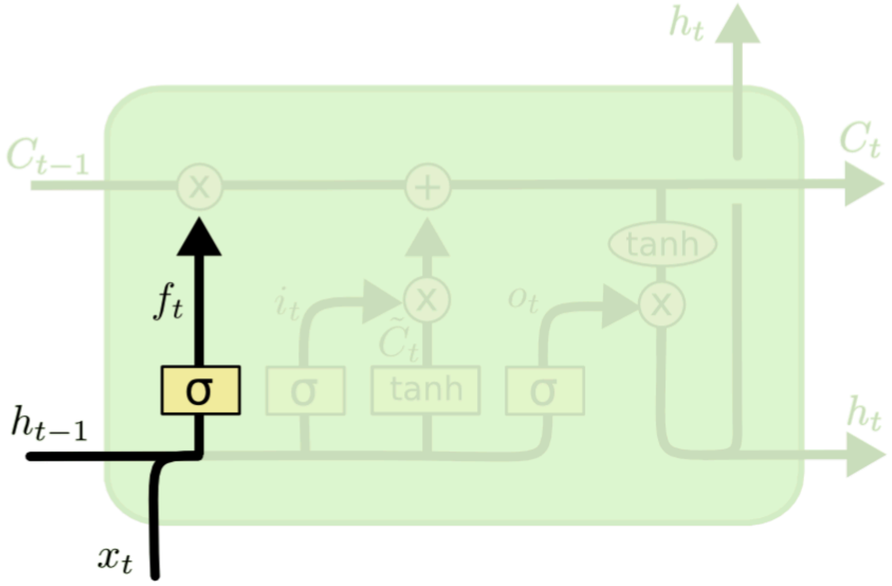
$\tilde{c}_t \in (-1,1)^h$: cell input activation vecto

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

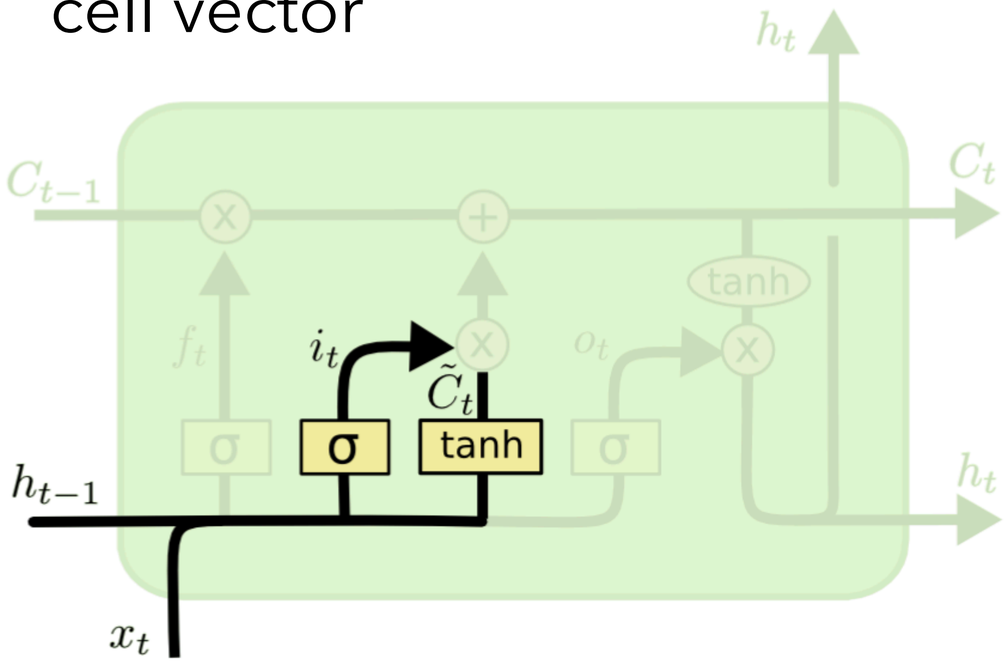
$\tilde{c}_t \in (-1,1)^h$: cell input activation vector

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

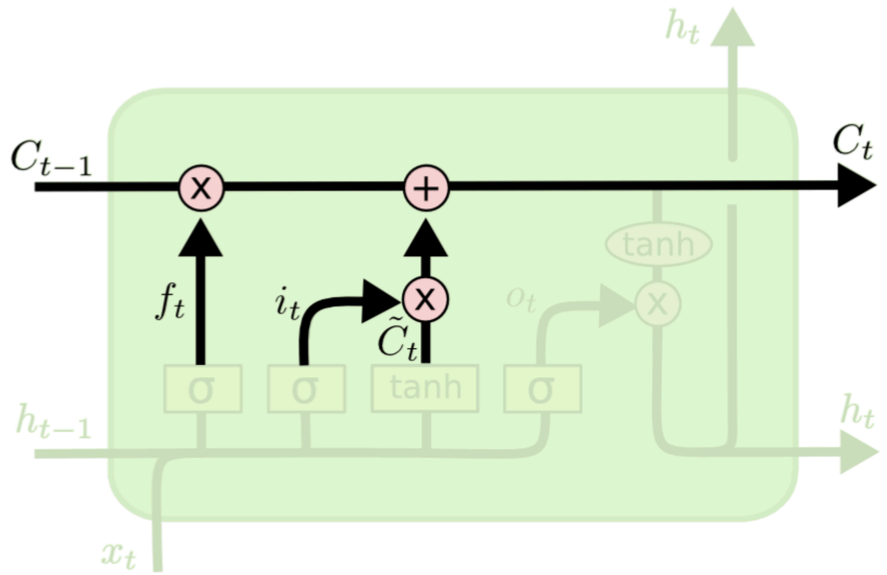
$\tilde{c}_t \in (-1,1)^h$: cell input activation vector

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

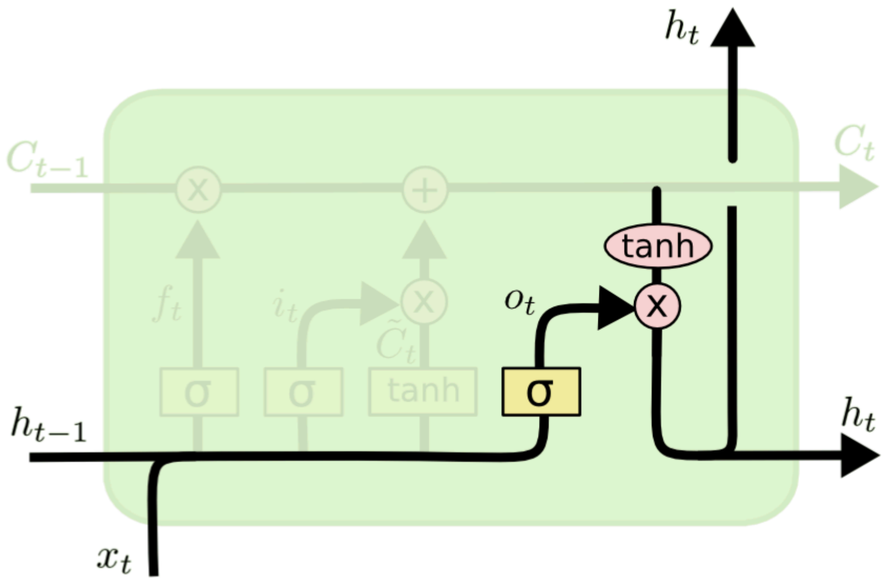
$c_t \in \mathbb{R}^h$: cell state vector

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/