COMP 3361 Natural Language Processing

Lecture 7: Recurrent Neural Networks

Spring 2024

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

Announcements

- The class will not have an in-person/zoom meeting this Friday.
  - We will record a video tutorial on PyTorch and huggingface and upload it to the course website.
- Assignment 1 due in two weeks!
- 新年快樂！ Happy Chinese New Year.

Lecture plan

- Recap of Byte-pair encoding (BPE) tokenization
- Other tokenization variants
- Basics of neural networks
- Recurrent neural networks

Neural language models: tokenization

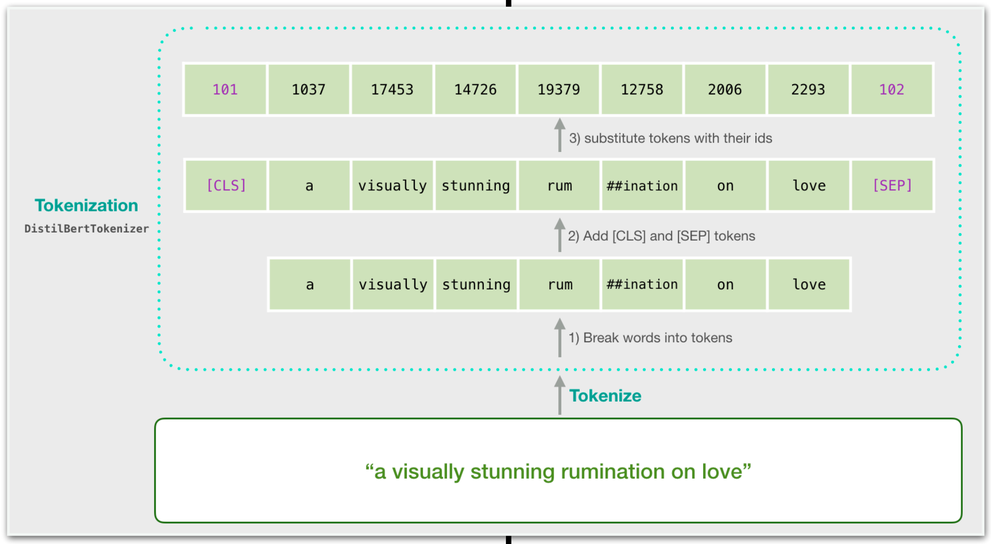
Tokenization to input vectors

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

Neural Network

Mapping each tokenized id into its corresponding embeddings

Tokenization:



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

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: 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

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

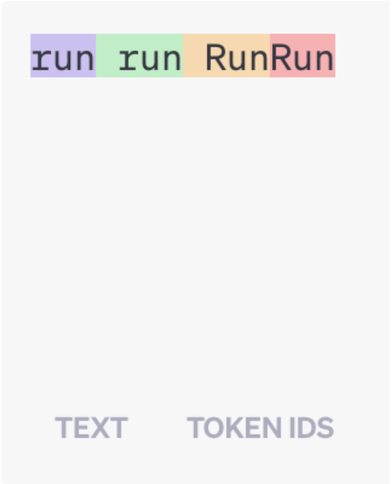
$$\text{Decode}([20, 12]) = \text{“ hugs”}$$
$$\text{Decode}([9, 7, 12, 12, 6, 2, 11, 3, 10, 10, 3, 12, 12]) = \text{“misshapenness”}$$

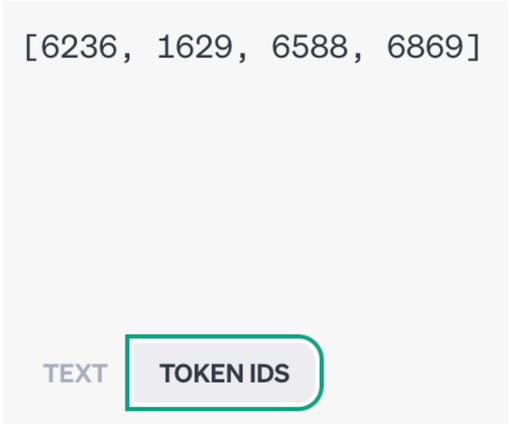
Byte-pair encoding: properties

- 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

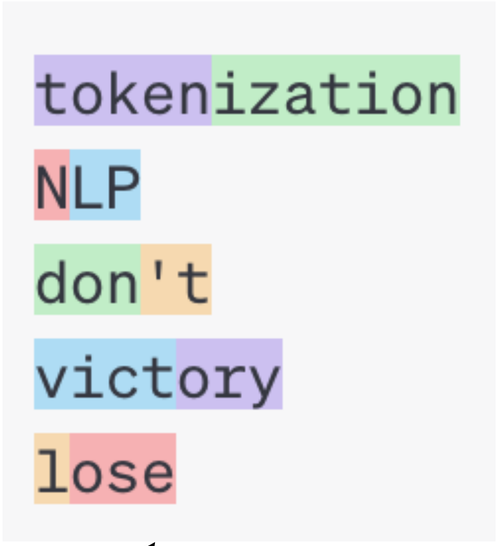
- Token != word
- Spaces are part of token
  - “run” is a different token than “ run”
- Not invariant to case changes
  - “Run” is a different token than “run”

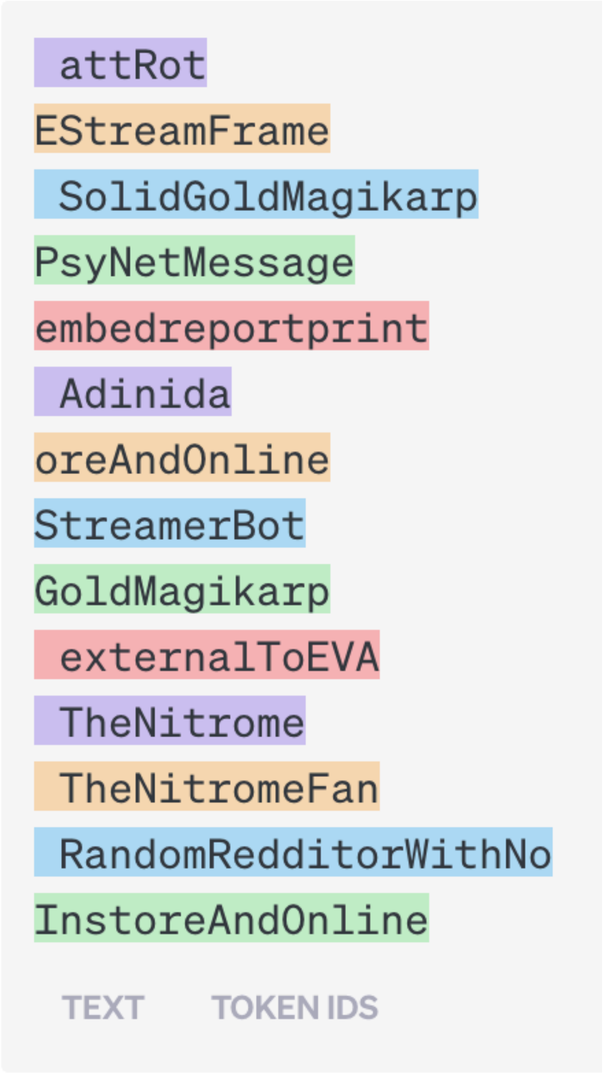




Weird properties of tokenizers

- 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)

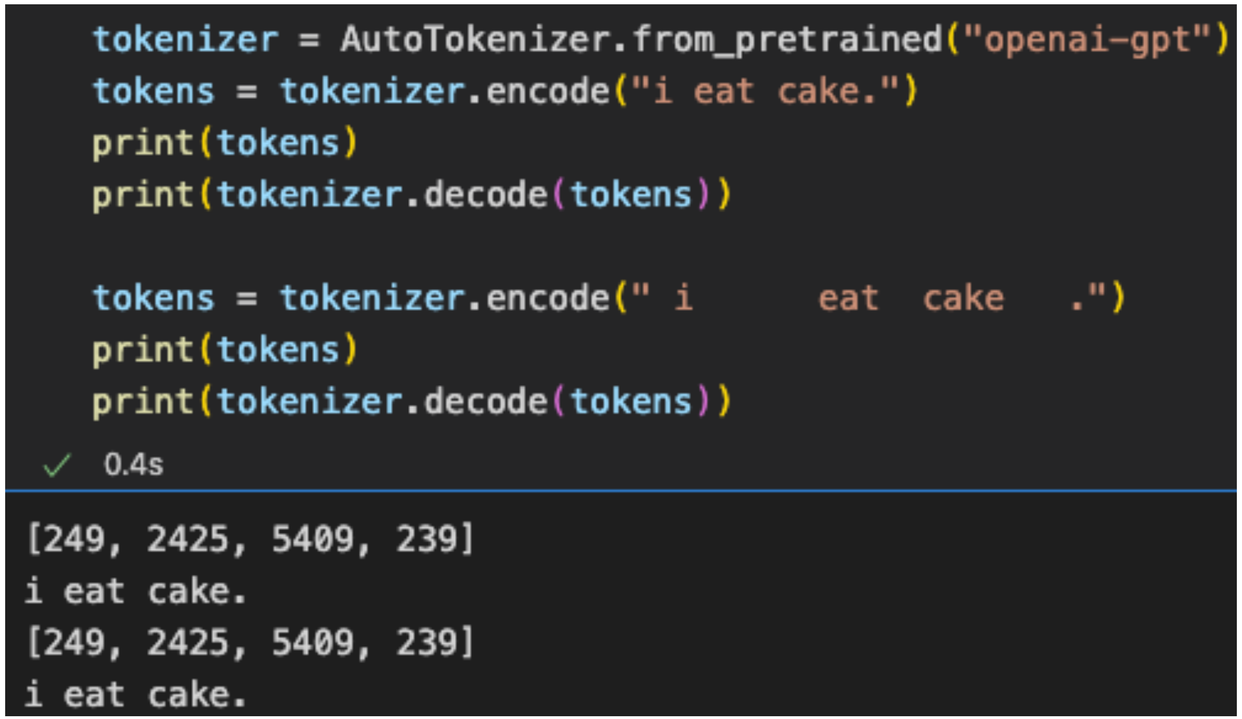
- Initialize $\mathcal{V}$ as the set of characters in $\mathcal{D}$

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

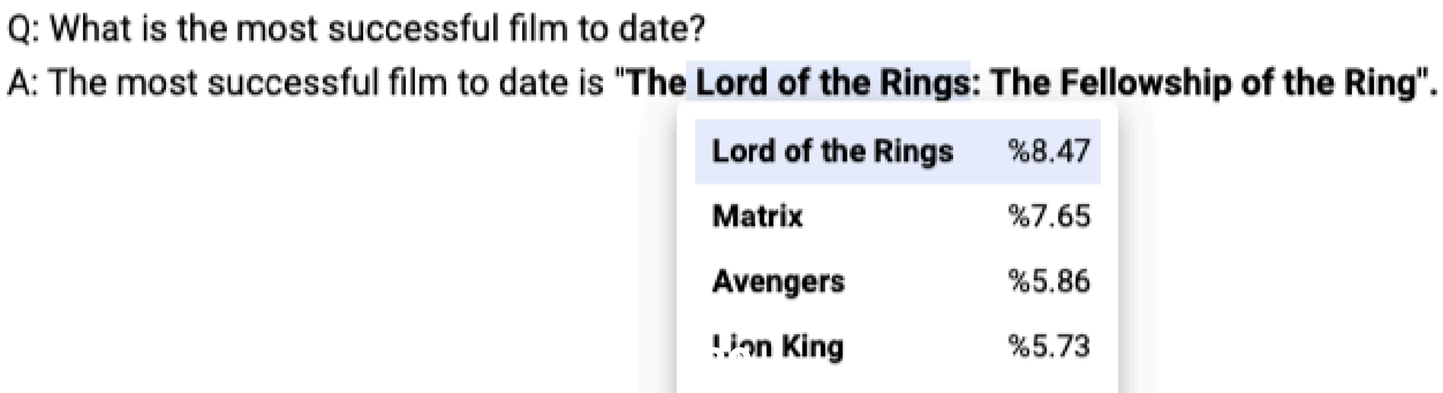
SentencePiece paper example in Japanese:

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 $\mathcal{D}$

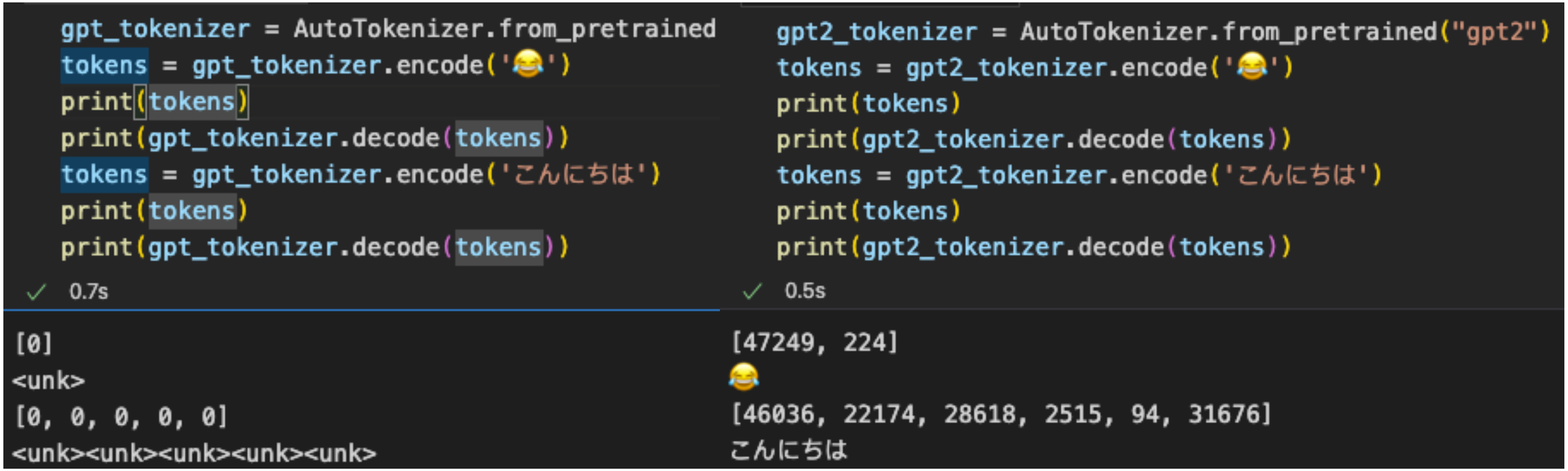
+ Convert $\mathcal{D}$ into a list of tokens (**bytes**)

- 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

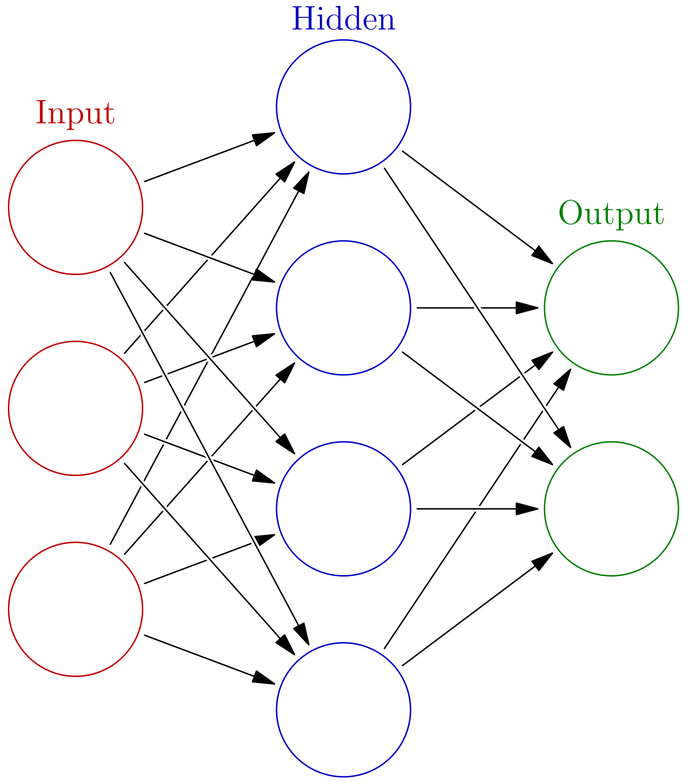
Examples of LLMs and their tokenizers

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

(Very quick) Deep learning review

Neural networks



Goal: Approximate some function $f : \mathbb{R}^k \rightarrow \mathbb{R}^d$

Essential elements:

- Input: Vector $x \in \mathbb{R}^k$, Output: $y \in \mathbb{R}^d$
- Hidden representation layers $h_i \in \mathbb{R}^{d_i}$
- Non-linear, differentiable (almost everywhere) activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ (applied element-wise)
- Weights connecting layers: $W \in \mathbb{R}^{d_{i+1} \times d_i}$ and bias term $b \in \mathbb{R}^{d_{i+1}}$
  - Set of all parameters is often referred to as θ

$$\hat{y} = W_2\sigma(W_1 x + b_1) + b_2$$

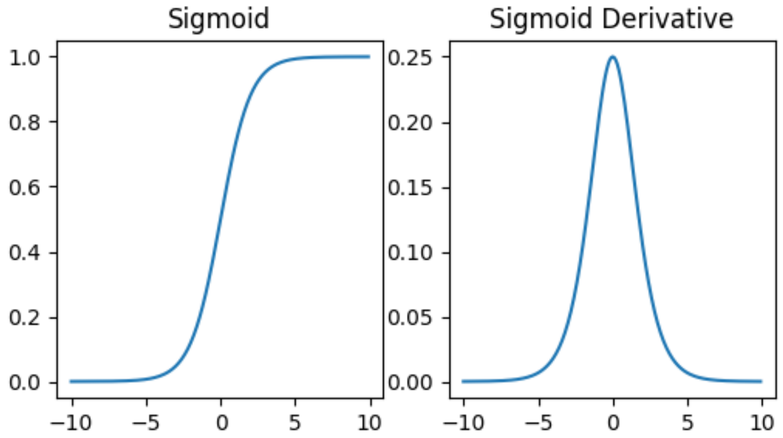
where $x \in \mathbb{R}^3, \hat{y} \in \mathbb{R}^2, W_1 \in \mathbb{R}^{4\times 3}$, $W_2 \in \mathbb{R}^{2\times 4}, b_1 \in \mathbb{R}^4,$ and $b_2 \in \mathbb{R}^2$

https://en.wikipedia.org/wiki/Artificial_neural_network

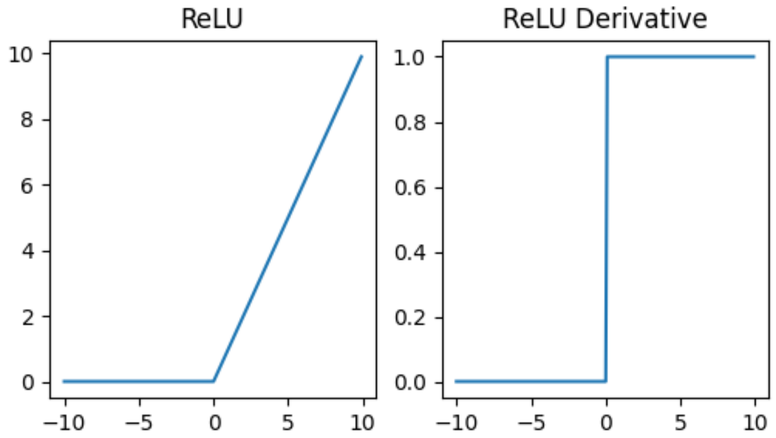
Common activation functions

Sigmoid

$$\frac{1}{1+e^{-x}}$$



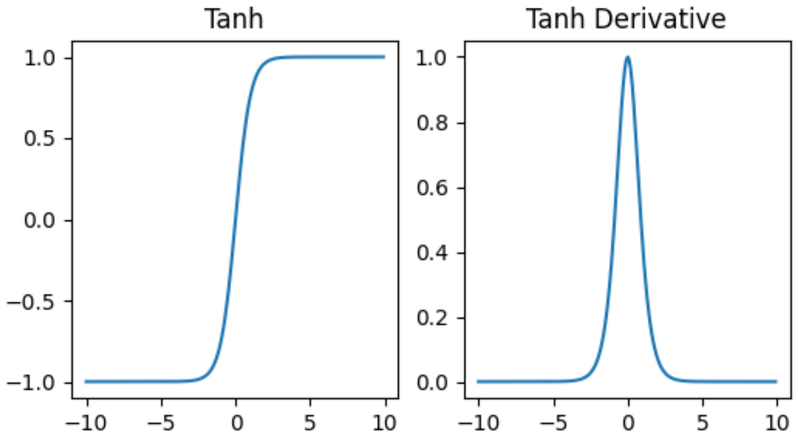
ReLU



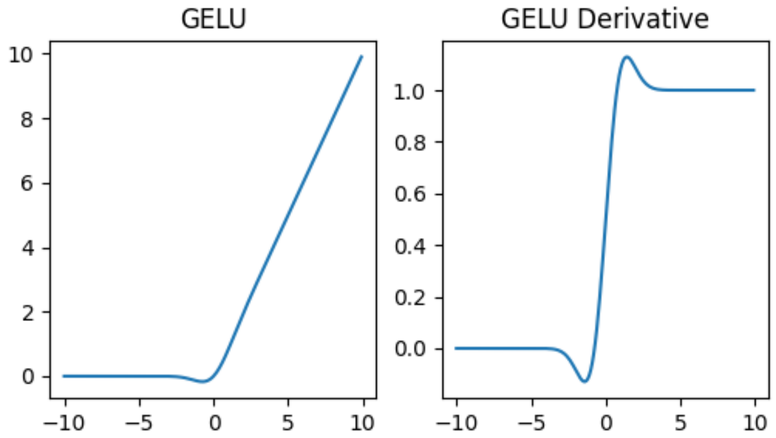
$$\max(0, x)$$

Tanh

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$



GeLU



$$\frac{1}{2}x\left(1+\operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

Learning

Required:

- Training data $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$
- Model family: some specified function (e.g., $\hat{y} = W_2\sigma(W_1 x + b_1) + b_2$)
  - Number/size of hidden layers, activation function, etc. are FIXED here
- (Differentiable) Loss function $L(y, \hat{y}) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$

Learning Problem:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, \hat{y}^{(i)} = f_\theta(x^{(i)}))$$

Common loss functions

- **Regression problems:**
  - Euclidean Distance/Mean Squared Error/L2 loss:

    $$L_2(y, \hat{y}) = ||y - \hat{y}||_2^2 = \frac{1}{2}\sum_{i=1}^{k}(y_i - \hat{y}_i)^2$$

  - Mean Absolute Error/L1 loss: $L_1(y, \hat{y}) = ||y - \hat{y}||_1 = \sum_{i=1}^{k}|y_i - \hat{y}_i|$
- **2-way classification:**
  - Binary Cross Entropy Loss: $L_{\text{BCE}}(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$
- **Multi-class classification:** *(for example, words…)*
  - Cross Entropy Loss**:**

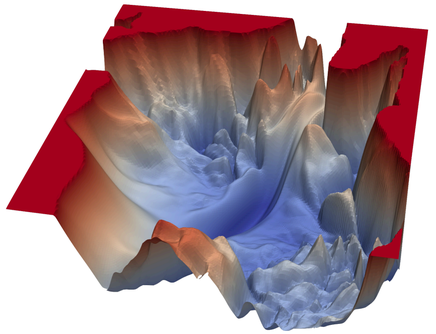
    (Very related to perplexity!)

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

Gradient Descent

Learning Problem:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, \hat{y}^{(i)} = f_\theta(x^{(i)}))$$



"Loss landscape" - loss w.r.t θ

https://www.cs.umd.edu/~tomg/projects/landscapes/

- However, finding the global minimum is often impossible in practice (need to search over all of $\mathbb{R}^{\dim(\theta)}$!)
- Instead, get a local minimum with *gradient descent*

Gradient Descent

- Learning rate $\alpha \in \mathbb{R}, \alpha > 0$ (often quite small e.g., 3e-4)
- Randomly initialize $\theta^{(0)}$
- Iteratively get better estimate with:

Next estimate — Learning rate (step size)

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

Previous Estimate

Gradient is:

- the vector of partial derivatives of the parameters with respect to the loss function
- A linear approximation of the loss function at $\theta^{(i)}$

$$\frac{\partial L}{\partial \theta}(\theta^{(i)}) = \begin{bmatrix} \frac{\partial L}{\partial \theta_1^{(i)}} \\ \frac{\partial L}{\partial \theta_2^{(i)}} \\ \vdots \\ \frac{\partial L}{\partial \theta_n^{(i)}} \end{bmatrix}$$

Stochastic gradient descent

Gradient Descent:

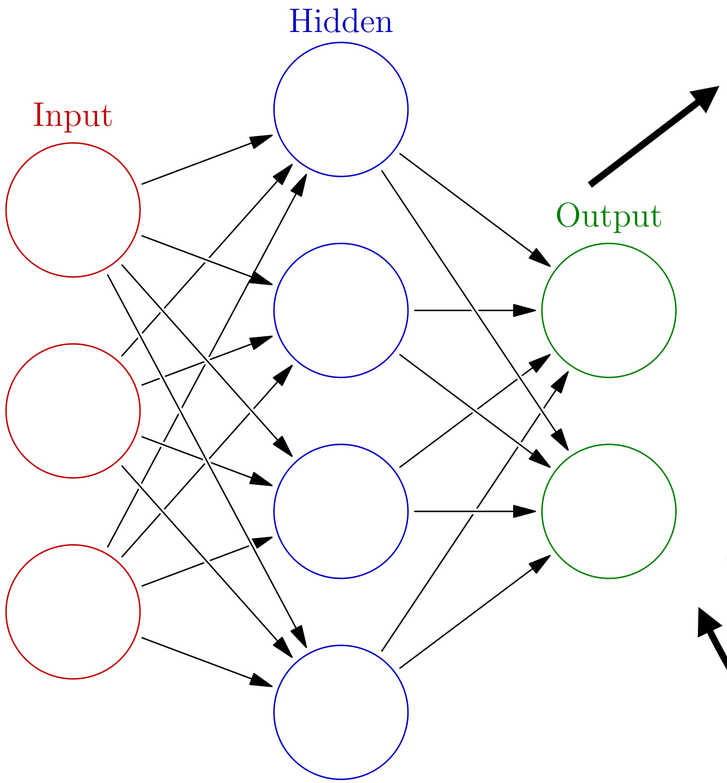
$$\theta^{(i+1)} = \theta^{(i)} - \alpha * \frac{\partial L}{\partial \theta}(\theta^{(i)})$$

- Problem: calculating the true gradient can be very expensive (requires running model on entire dataset!)
- Solution: **Stochastic Gradient Descent**
  - Sample a subset of the data of fixed size (batch size)
  - Take the gradient with respect to that subset
  - Take a step in that direction; repeat
- Not only is it more computationally efficient, but it often finds better minima than vanilla gradient descent
  - Why? Possibly because it does a better job skipping past plateaus in loss landscape

Backpropagation

One efficient way to calculate the gradient is with **backpropagation**.

Leverages the **Chain Rule**: $\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$



1. Forward Pass

$$h_1 = W_1 x + b_2$$

$$h_2 = \sigma(h_1)$$

$$\hat{y} = W_2 h_2 + b_2$$

2. Calculate Loss

$$L(y, \hat{y})$$

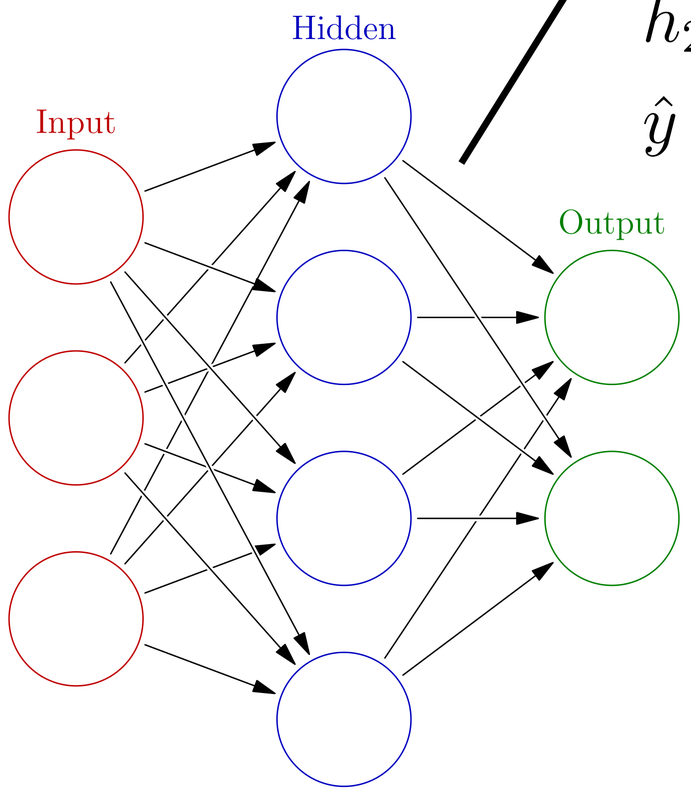
3. Backwards Pass

Calculate the gradient of the loss w.r.t. each parameter using the chain rule and intermediate outputs

Backpropagation

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

(Long, messy exact derivation below)



1. Forward Pass

$$h_1 = W_1 x + b_2$$
$$h_2 = \sigma(h_1)$$
$$\hat{y} = W_2 h_2 + b_2$$

2. Calculate Loss

$$L(y, \hat{y})$$

3. Backwards Pass

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

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial (W_2 h_2 + b_2)}{\partial W_2} = \delta_{\hat{y}} \cdot h_2^T$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial (W_2 h_2 + b_2)}{\partial b_2} = \delta_{\hat{y}}$$

$$\delta_{h_2} := \frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial (W_2 h_2 + b_2)}{\partial h_2} = \delta_{\hat{y}} \cdot W_2^T$$

$$\delta_{h_1} := \frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} = \delta_{h_2} \cdot \frac{\partial \sigma(h_1)}{\partial h_1}$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial W_1} = \delta_{h_1} \cdot \frac{\partial (W_1 x + b)}{\partial W_1} = \delta_{h_1} \cdot x^T$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial b_1} = \delta_{h_1} \cdot \frac{\partial (W_1 x + b)}{\partial b_1} = \delta_{h_1}$$

Classification with deep learning

- For classification problems (like next word-prediction…) we want to predict a **probability distribution** over the label space
- However, neural networks' output $y \in \mathbb{R}^d$ is not guaranteed (or likely) to be a probability distribution
- To force the output to be a probability distribution, we apply the **softmax function**

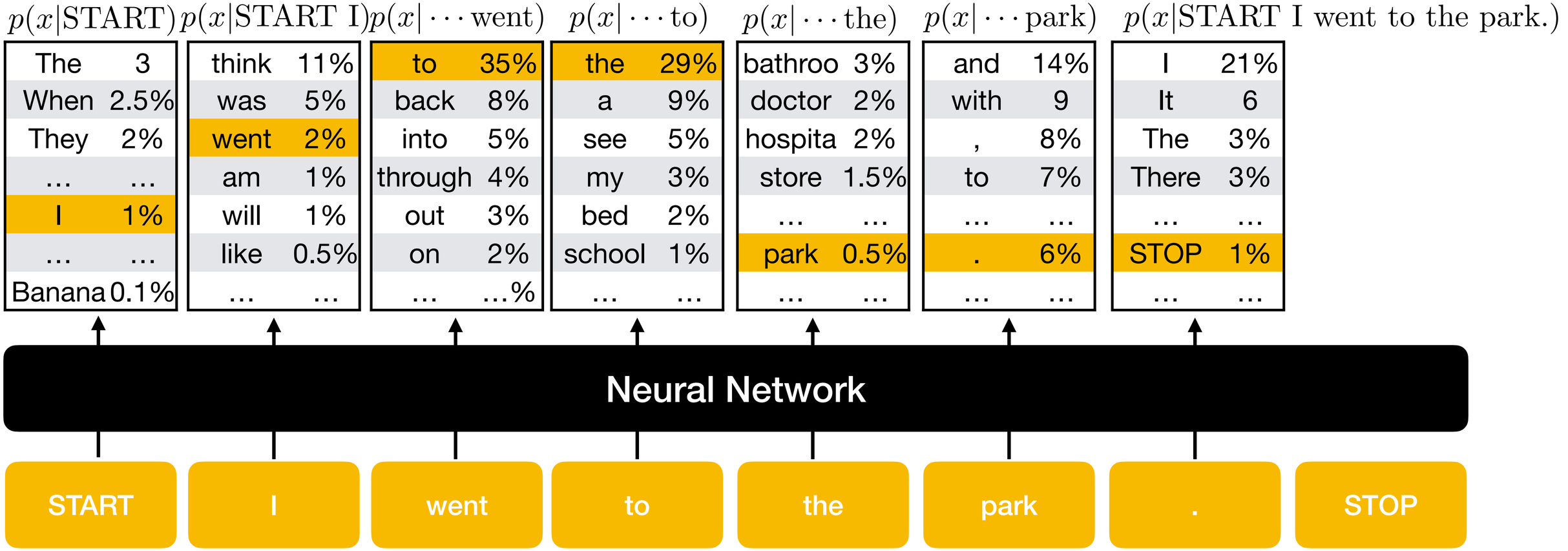
$$\mathrm{softmax}(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{d} \exp(y_j)}$$

  - The values y before applying the softmax are often called "logits"

Language modeling with neural networks

Inputs/Outputs

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



Neural language models

But neural networks take in real-valued vectors, not words…

- Use one-hot or learned embeddings to map from words to vectors!
  - Learned embeddings become part of parameters θ

Neural networks output vectors, not probability distributions…

- Apply the softmax to the outputs!
- What should the size of our output distribution be?
  - Same size as our vocabulary $|\mathcal{V}|$

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

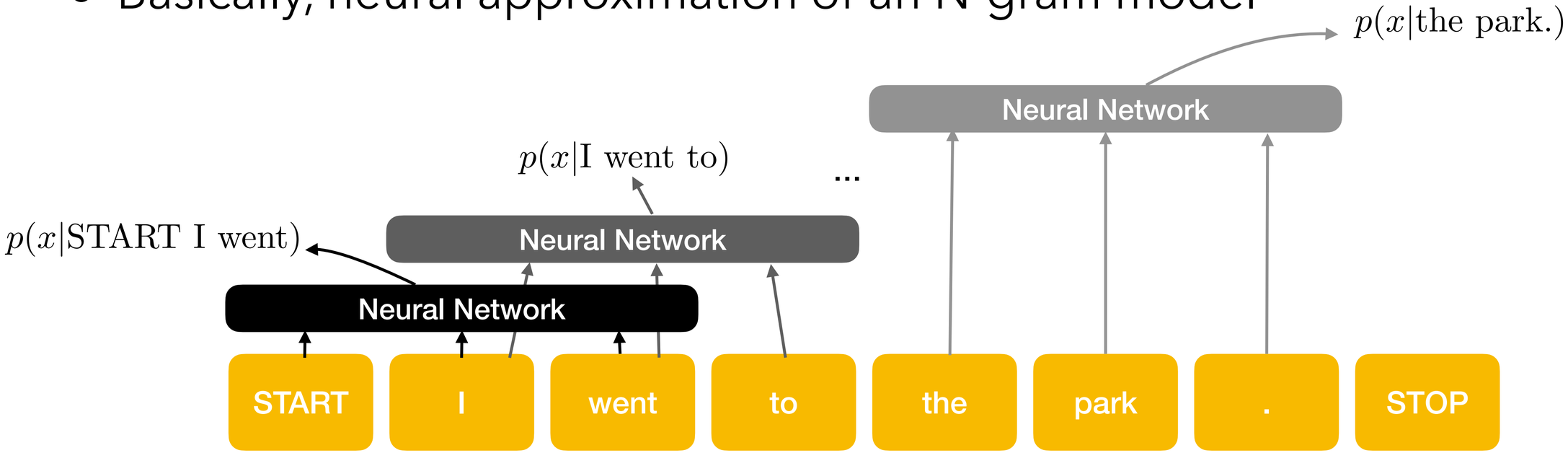
- Ideas?

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

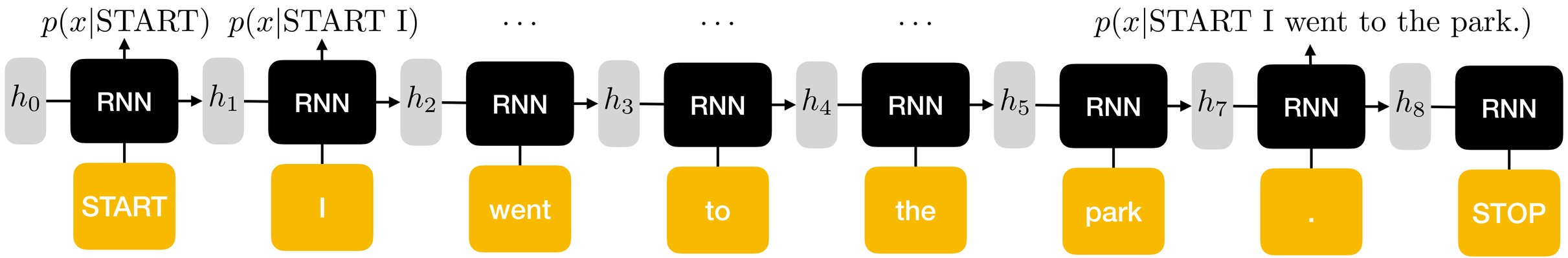


Recurrent neural networks

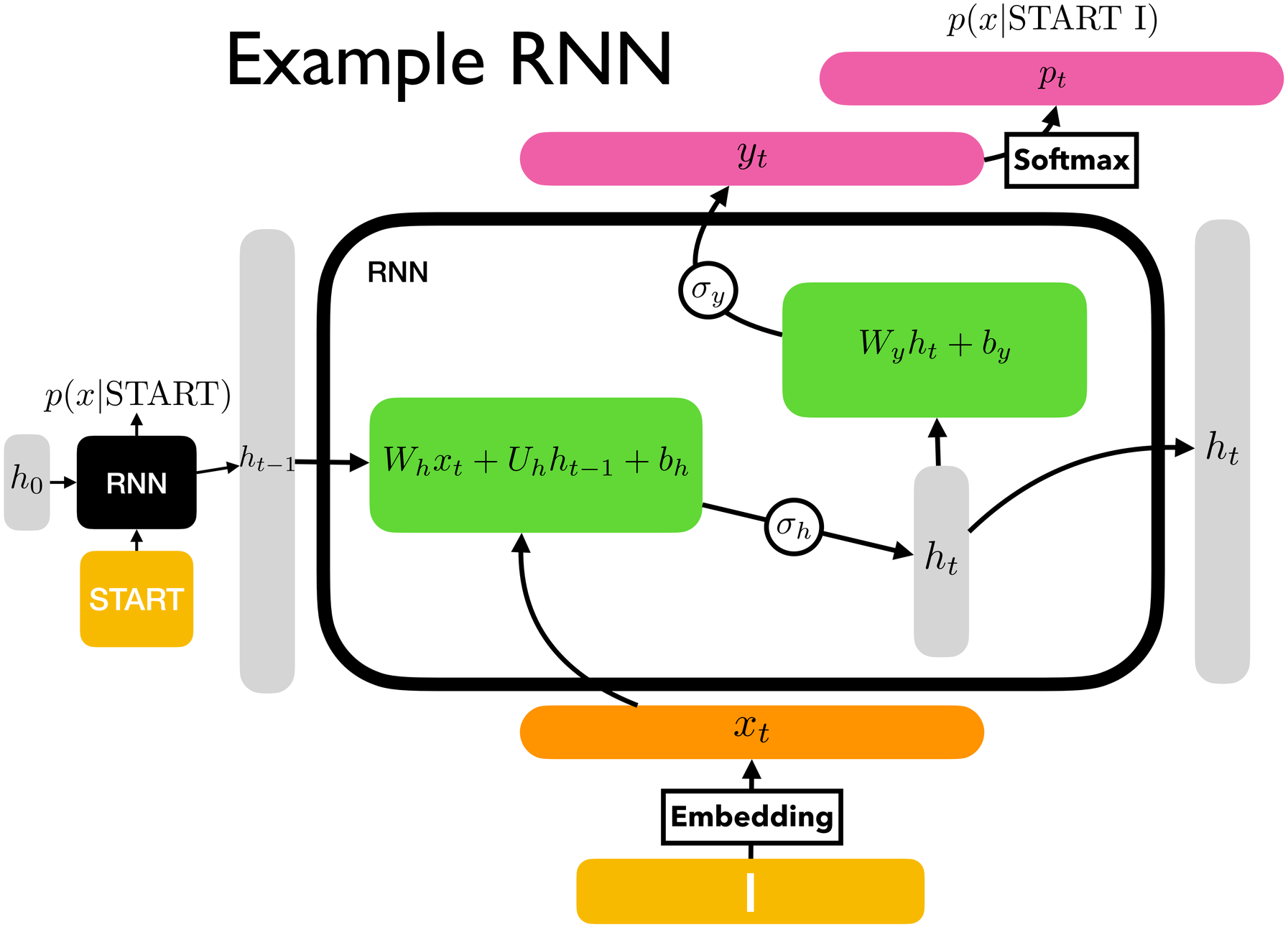
Idea 2: Recurrent Neural Networks (RNNs)

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

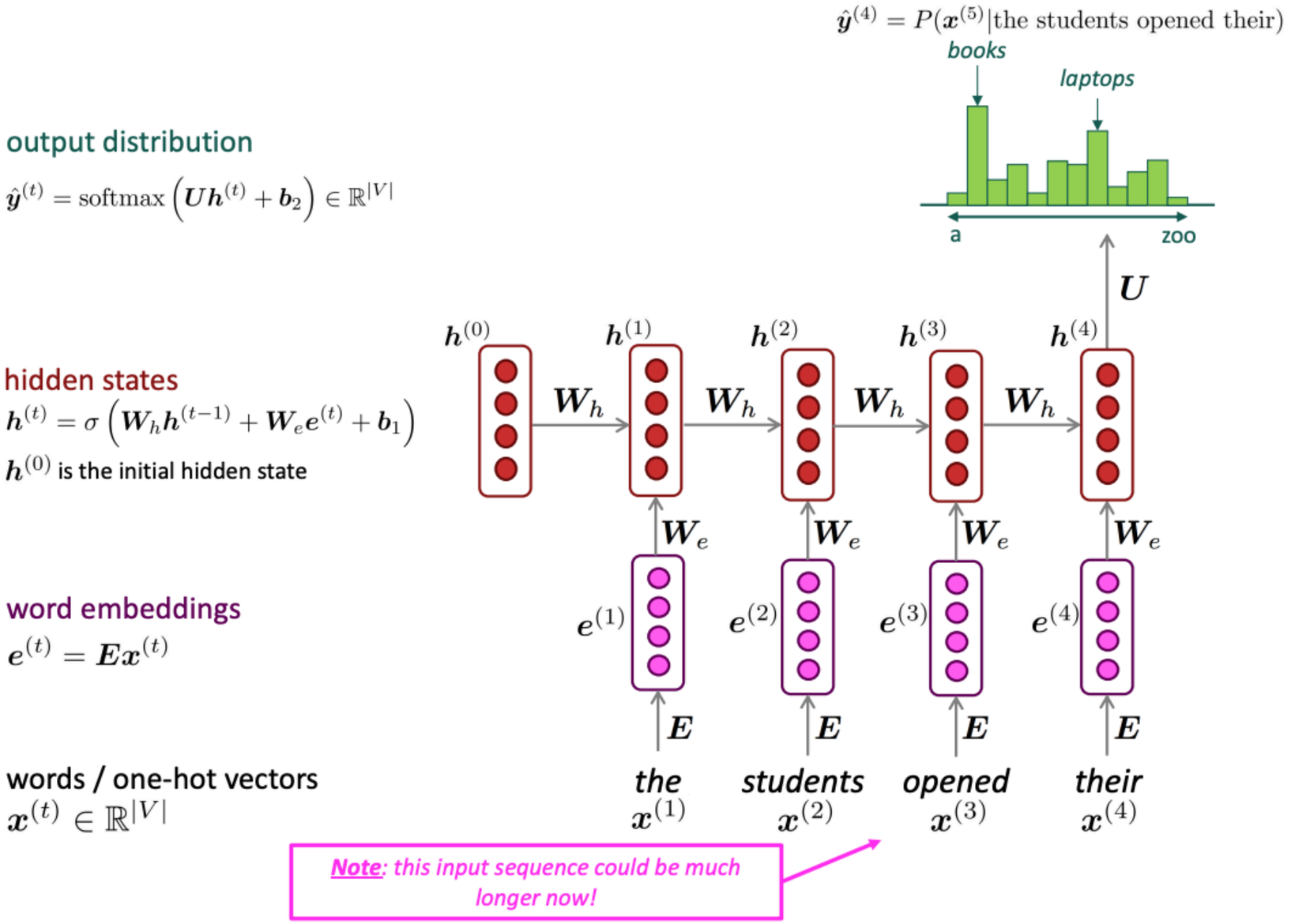
Equations:

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

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

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

Example RNN


output distribution
$\hat{y}^{(t)} = \text{softmax}\left(Uh^{(t)} + b_2\right) \in \mathbb{R}^{|V|}$
hidden states
$h^{(t)} = \sigma\left(W_h h^{(t-1)} + W_e e^{(t)} + b_1\right)$
$h^{(0)}$ is the initial hidden state
word embeddings
$e^{(t)} = Ex^{(t)}$
words / one-hot vectors
$x^{(t)} \in \mathbb{R}^{|V|}$
$\hat{y}^{(4)} = P(x^{(5)}|\text{the students opened their})$
books
laptops
a
zoo
U
W_h
W_e
E
$h^{(0)}$
$h^{(1)}$
$h^{(2)}$
$h^{(3)}$
$h^{(4)}$
$e^{(1)}$
$e^{(2)}$
$e^{(3)}$
$e^{(4)}$
the
students
opened
their
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$
$x^{(4)}$
Note: this input sequence could be much longer now!

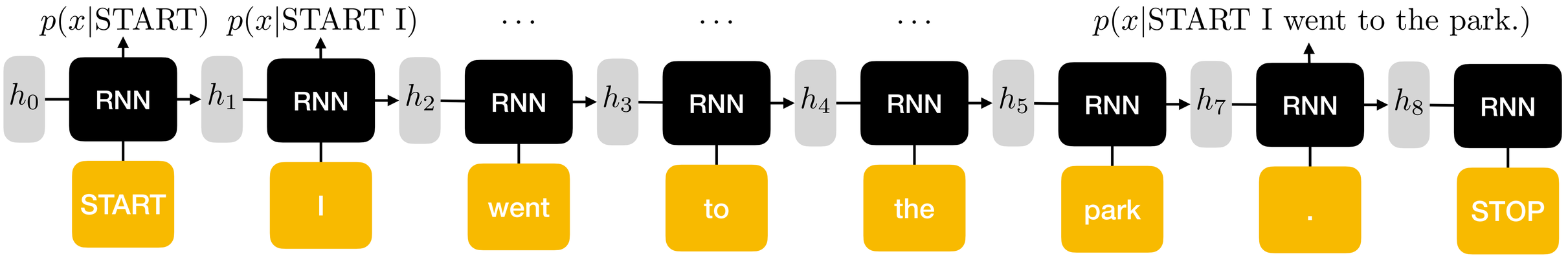
Recurrent neural networks

- 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

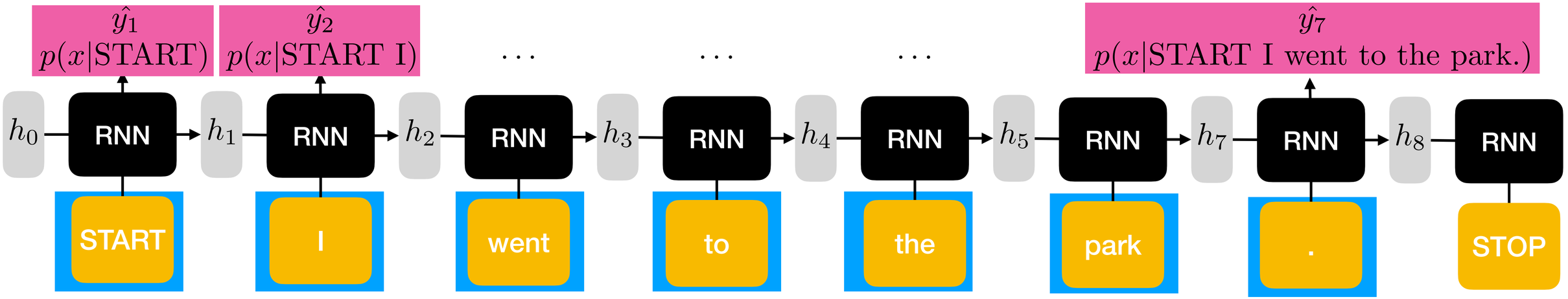
Training procedure

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)

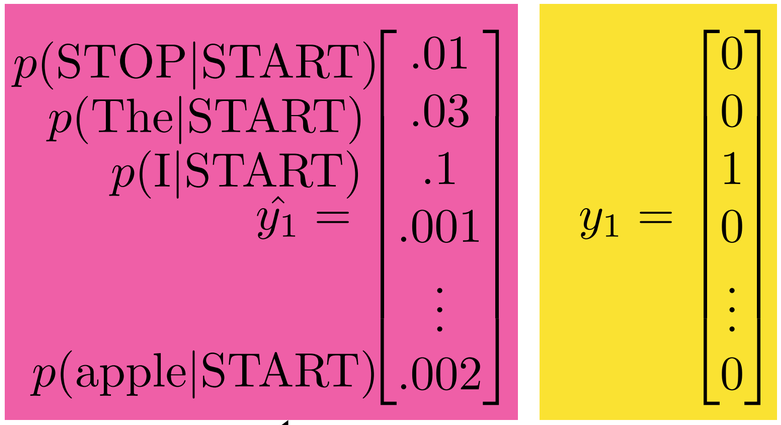


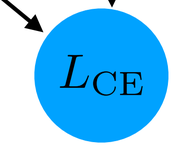
Training procedure

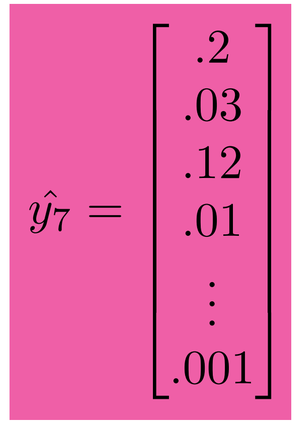
2. Run model on (batch of) x's from data $\mathcal{D}$ to get probability distributions $\hat{y}$

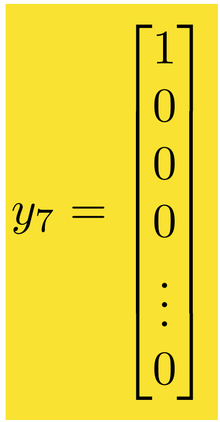
3. Calculate loss compared to true y's (Cross Entropy Loss)

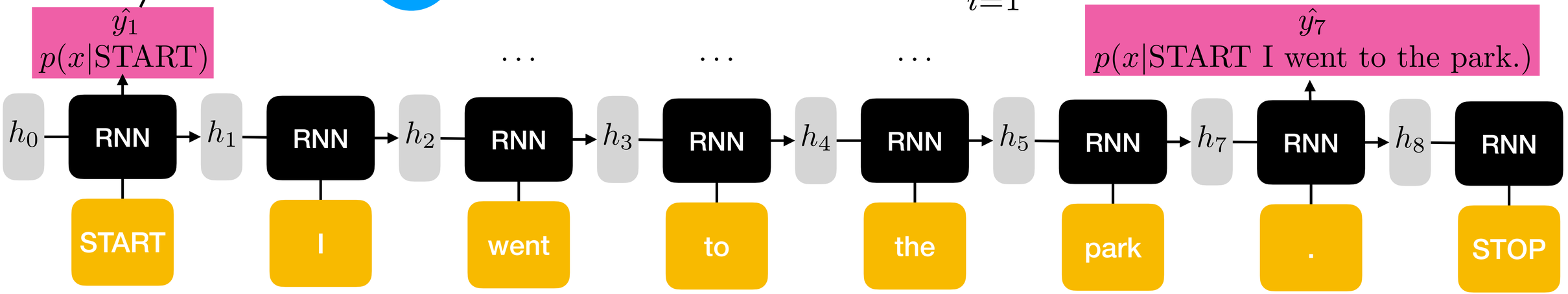
$$L_{\text{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$











Training procedure

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

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

$$y_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

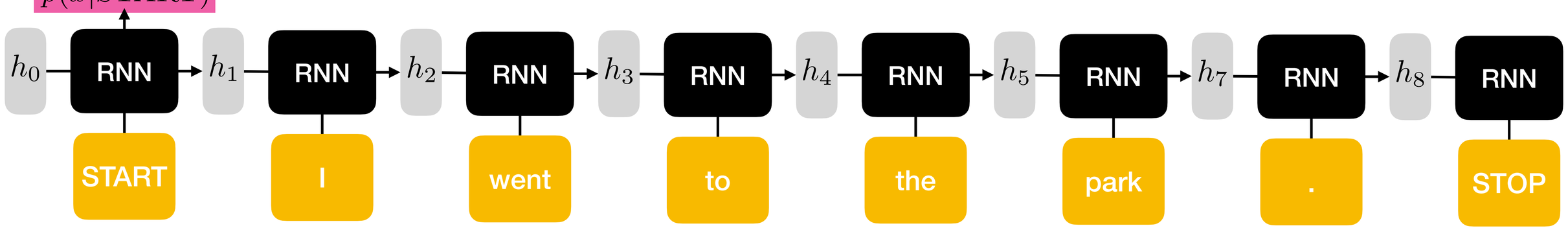
(Actual observed word)

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

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