

CSE 493s/599s

Lecture 2. LM basics and
pre-transformer architectures

Sewoong Oh



Lecture notes

- These lecture notes are based on other courses in LLMs, including
 - CSE493S/599S at UW by Ludwig Schmidt: <https://mlfoundations.github.io/advancedml-sp23/>
 - EE-628 at EPFL by Volkan Cevher: <https://www.epfl.ch/labs/lions/teaching/ee-628-training-large-language-models/ee-628-slides-2025/>
 - ECE381V Generative Models at UT Austin by Sujay Sanghavi
 - and various papers and blogs cited at the end of the slide deck

Outline

- Language models
- General LLM framework
 - Token processing
 - Sequence mixing
 - Prediction
- Example architectures

Notations

input data / sample	x
output / label	y
model weight	W
sentence	S
word	w

since we learned a lot about tokenization,
we will treat token=word
unless needed to differentiate them

Language Model (LM) basics

- **Example** language task:

- What is the most *likely* next word $[nw]$ given the following S_{source} ?

S_{source} : “On January 1 people usually say happy new $[nw]$.”

Next Word Prediction: $\mathbb{P}([nw] = w \mid S_{\text{source}})$

Language Model (LM) basics

- **Example** language task:

- What is the most *likely* next word $[nw]$ given the following S_{source} ?

S_{source} : “On January 1 people usually say happy new $[nw]$.”

- **Next Word Prediction** is fundamental in language modeling.

- check spelling & grammar correction

$$\mathbb{P}(\text{year} | S_{\text{source}}) > \mathbb{P}(\text{years} | S_{\text{source}})$$

- machine translation

$$\mathbb{P}(S_{\text{translation 1}} | S_{\text{source}}) > \mathbb{P}(S_{\text{translation 2}} | S_{\text{source}})$$

- sentence classification

$$\mathbb{P}(\text{good} | S_{\text{source}}) > \mathbb{P}(\text{bad} | S_{\text{source}})$$

- speech recognition

$$\mathbb{P}(\text{Wreck a nice beach}) < \mathbb{P}(\text{Recognize speech})$$

- chatbot

$$\mathbb{P}([nw] | S_{\text{source}})$$

- reasoning, planning, coding, math, etc.

- **Definition** (Language Model [Jurafsky and Martin,2023])
 - Models that assign probability to sequences of words are called language models.
- Next word prediction model is sufficient to assign probability to the entire sentence, $S = (w_1, w_2, \dots, w_T)$, of T words with the **chain rule**:

$$\mathbb{P}(S) = \mathbb{P}(w_{1:T}) = \mathbb{P}(w_1)\mathbb{P}(w_2 | w_1)\mathbb{P}(w_3 | w_1, w_2)\cdots\mathbb{P}(w_T | w_{1:T-1})$$

without any assumptions.

- For example, if $S = w_{1:3} = \text{"happy new year"}$, then
 $\mathbb{P}(S) = \mathbb{P}(\text{happy})\mathbb{P}(\text{new} | \text{happy})\mathbb{P}(\text{year} | \text{happy new})$
- **Fundamental question in (autoregressive) LM:**
how can we model and learn $\mathbb{P}(w_t | w_{1:t-1})$?

- how can we model and learn $\mathbb{P}(w_t | w_{1:t-1})$?
- A trivial but memory-inefficient model uses **empirical frequency** on a large corpus:

$$\mathbb{P}(\text{year} | S_{\text{source}}) = \frac{\mathbb{P}(S_{\text{source}} + \text{year})}{\mathbb{P}(S_{\text{source}})} \simeq \frac{\#(\text{On January 1 people usually say happy new year})}{\#(\text{On January 1 people usually say happy new})}$$

- What can go wrong?

- how can we model and learn $\mathbb{P}(w_t | w_{1:t-1})$?
- A trivial but memory-inefficient model uses **empirical frequency** on a large corpus:

$$\mathbb{P}(\text{year} | S_{\text{source}}) = \frac{\mathbb{P}(S_{\text{source}} + \text{year})}{\mathbb{P}(S_{\text{source}})} \simeq \frac{\#(\text{On January 1 people usually say happy new year})}{\#(\text{On January 1 people usually say happy new})}$$

- What can go wrong?
- All semantic relations are lost, e.g., "On January **first** people usually say happy new year."
- A lot of zero counts for rare (combinations of) words.

- how can we model and learn $\mathbb{P}(w_t | w_{1:t-1})$?
- A trivial but memory-inefficient model uses **empirical frequency** on a large corpus:

$$\mathbb{P}(\text{year} | S_{\text{source}}) = \frac{\mathbb{P}(S_{\text{source}} + \text{year})}{\mathbb{P}(S_{\text{source}})} \simeq \frac{\#(\text{On January 1 people usually say happy new year})}{\#(\text{On January 1 people usually say happy new})}$$

- A more efficient alternative is **N -gram LM**, that uses past $(N - 1)$ words to predict the next one:

On January 1 people usually say happy new [nw]

$N = 3$

$$\mathbb{P}(w_t | w_{1:t-1}) = \mathbb{P}(w_t | w_{t-N+1:t-1}) \simeq \frac{\#(w_{t-N+1:t})}{\#(w_{t-N+1:t-1})} \quad \text{e.g., } \frac{\#(\text{say happy new year})}{\#(\text{say happy new})}$$

- Bi-gram model, $N = 2$, uses only $\mathbb{P}(w_t | w_{t-1})$.

- Terminologies

- Language models operate on **tokens** and not words, as we learned in the previous lecture, but we will use them interchangeably, unless we need to be precise.
- **Vocabulary** of tokens is denoted by \mathcal{V} and its size by $|\mathcal{V}|$.
- Special tokens are used to indicate the beginning and end of sentences:
For example, 5 tokens to represent “<**BOS**> Happy new year <**EOS**>”
- **Pre-training** is building an LM from scratch on a large corpus of text.
- **Inference** is using a trained LM to do next word prediction.

- Bi-gram LM **pre-training** requires memory of size $O(|\mathcal{V}|^2)$ and the model is sparse:

- **For** all pair of words in the corpus

- count $\#(w)$ and $\#(w, w')$

- compute and store $\mathbb{P}(w | w') = \frac{\#(w, w')}{\#(w)}$ for all $(w, w') \in \mathcal{V} \times \mathcal{V}$

- Bi-gram LM **inference** generates sequence of words:

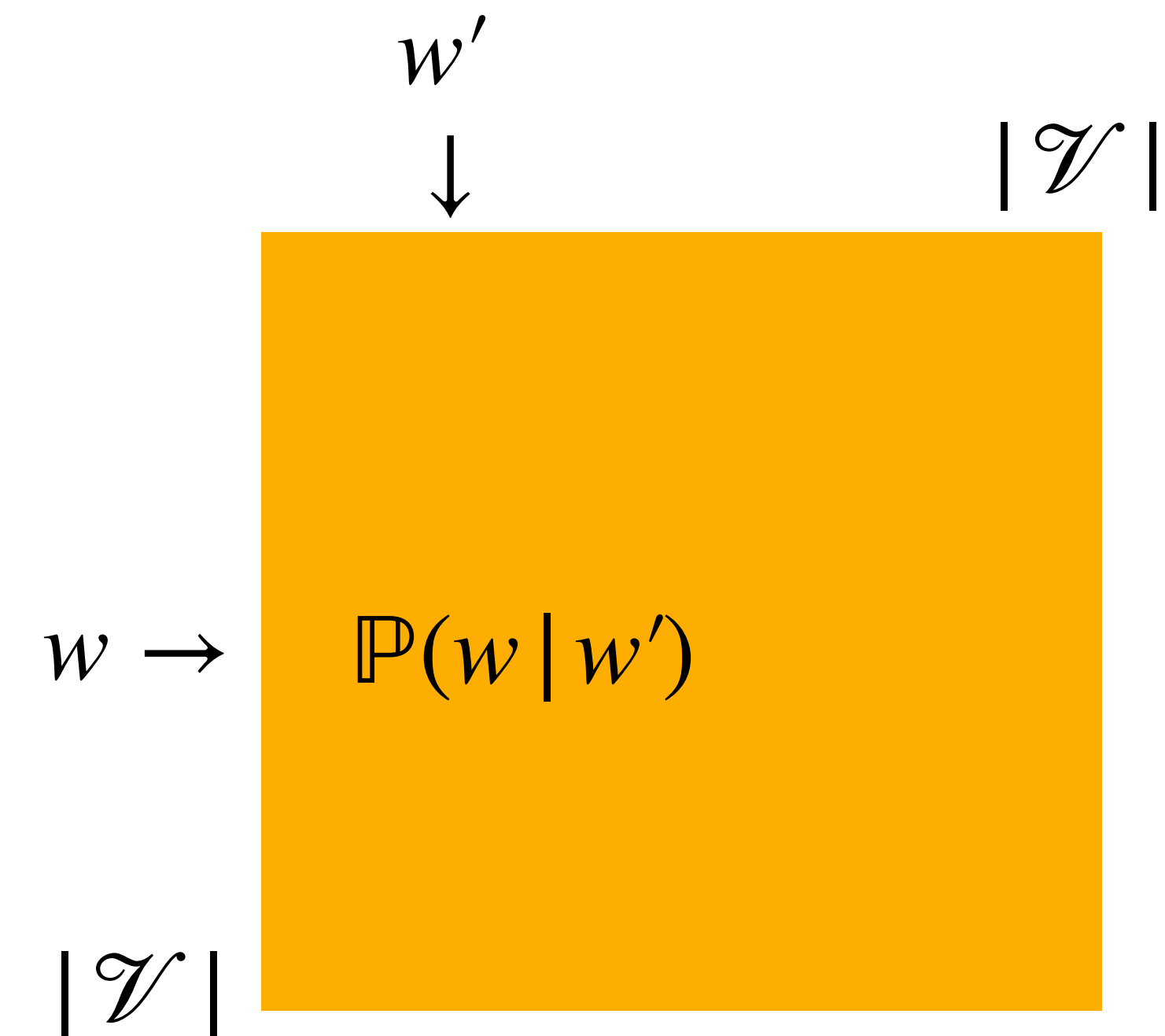
- Initialize: $w_1 \leftarrow \langle \text{BOS} \rangle, t = 1$

- **While True**

- $w_{t+1} \leftarrow \arg \max_{w \in \mathcal{V}} \mathbb{P}(w | w_t)$

- **If** $w_{t+1} = \langle \text{EOS} \rangle$: **Output** $(w_{1:t+1})$; **Break**

- $t \leftarrow t + 1$



- Bi-gram LM **pre-training** requires memory of size $O(|\mathcal{V}|^2)$ and the model is sparse:

- **For** all pair of words in the corpus

- count $\#(w)$ and $\#(w, w')$

- compute and store $\mathbb{P}(w | w') = \frac{\#(w, w')}{\#(w)}$ for all $(w, w') \in \mathcal{V} \times \mathcal{V}$

- Bi-gram LM **inference** generates sequence of words:

- Initialize: $w_1 \leftarrow \langle \text{BOS} \rangle, t = 1$

- **While True**

- $w_{t+1} \leftarrow \arg \max_{w \in \mathcal{V}} \mathbb{P}(w | w_t)$

- **If** $w_{t+1} = \langle \text{EOS} \rangle$: **Output** $(w_{1:t+1})$; **Break**

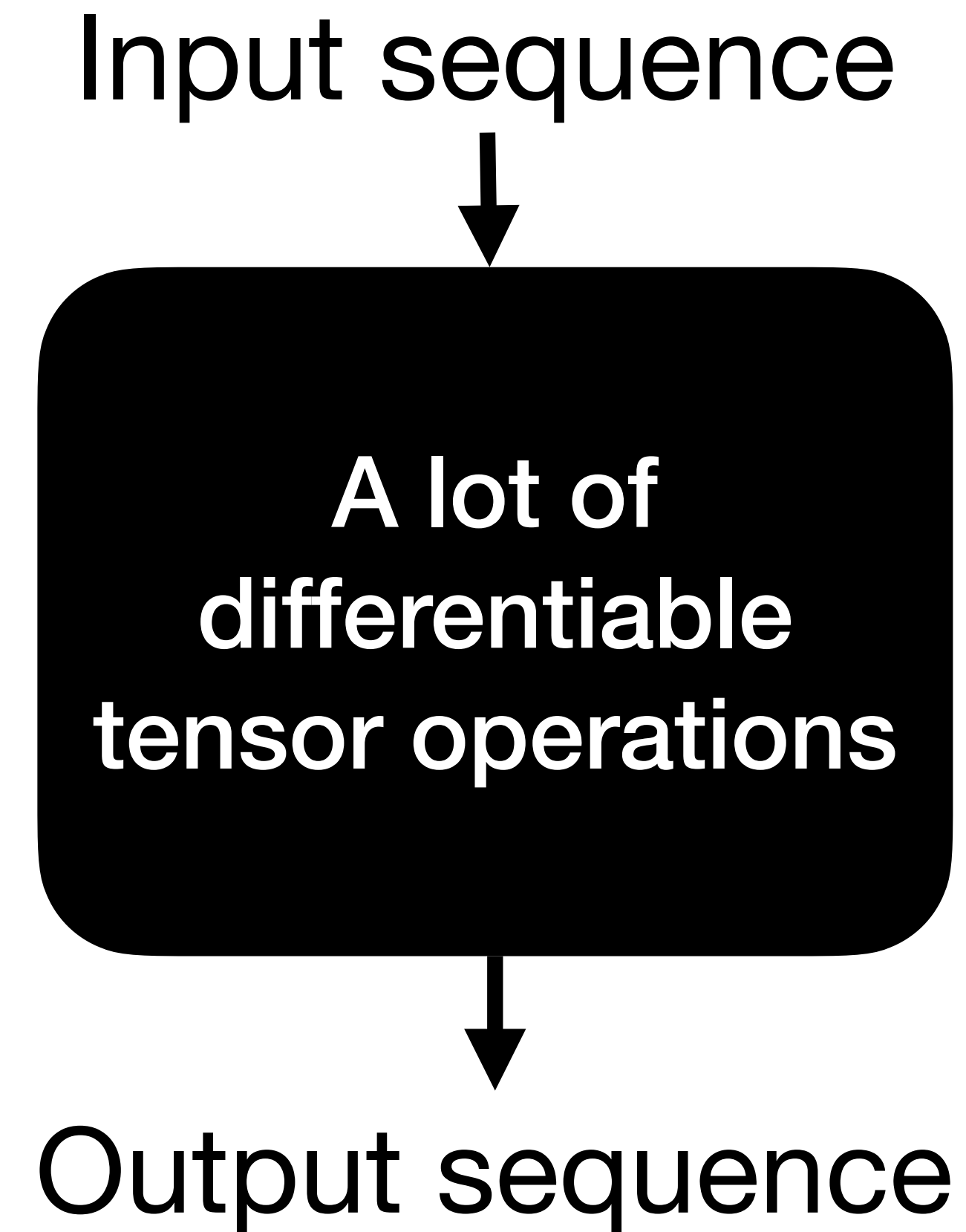
- $t \leftarrow t + 1$

- **With no prompt**, this toy model generates the same **deterministic** sequence all the time.
- We can instead start with a **given prompt**, and continue completing the given prompt using the same techniques, that we will learn more about later.
- Later, we will learn random sampling that makes the output better.

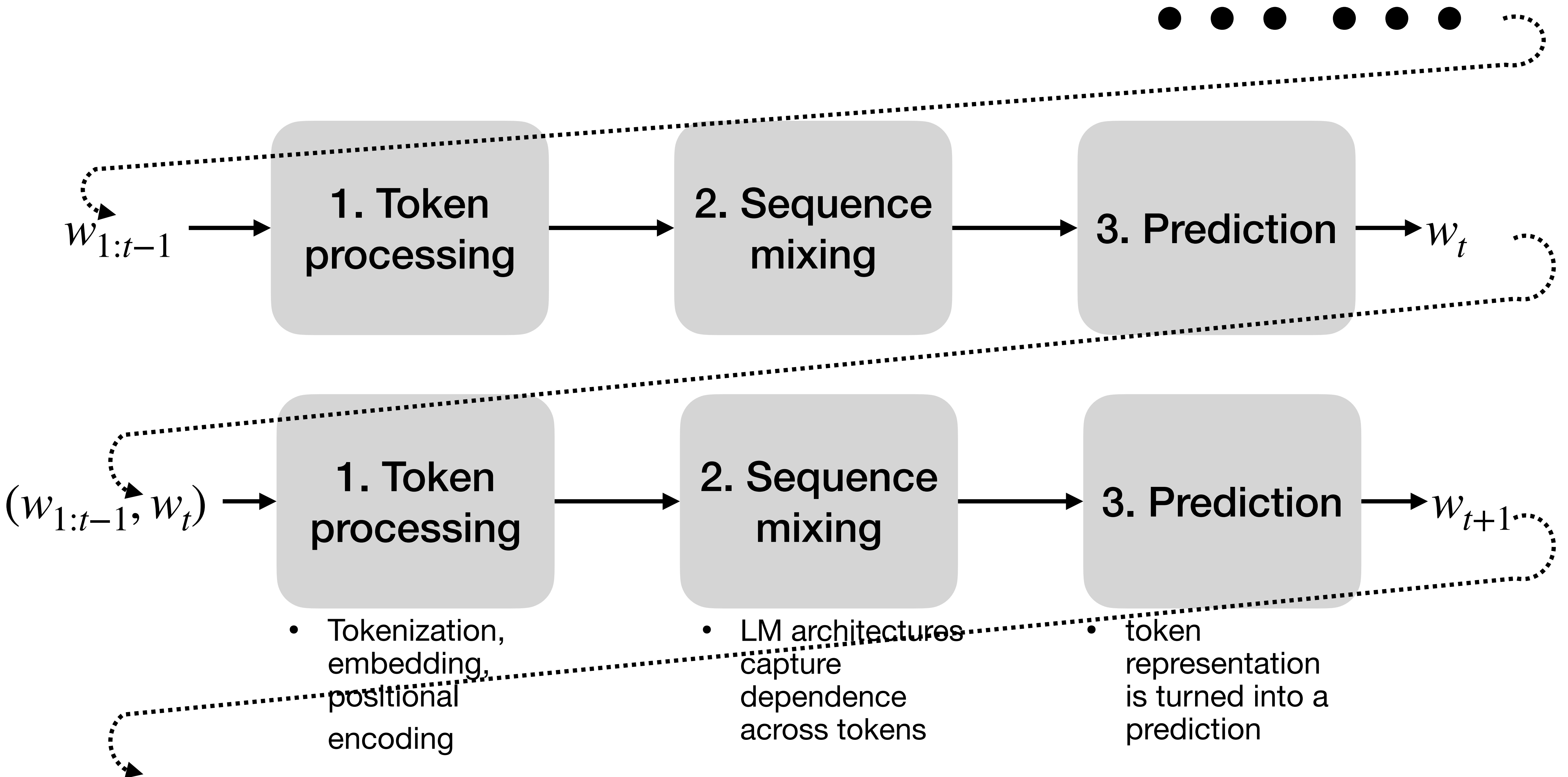
Modern language models



<https://xkcd.com/1838/>



General LLM's autoregressive framework

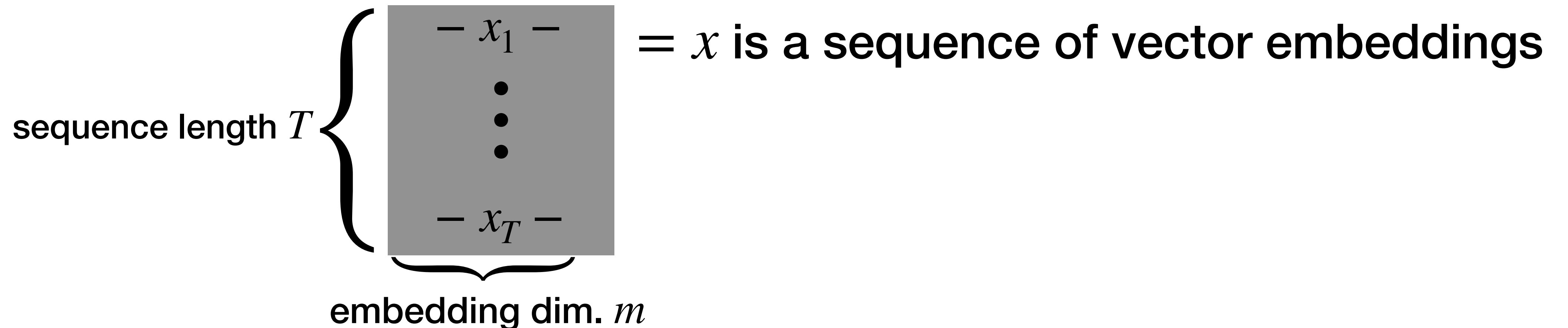
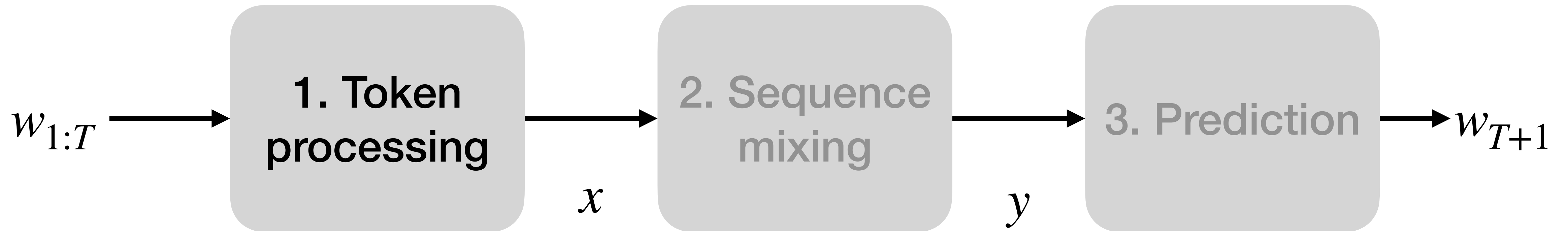


Outline

- Language models
- General LLM framework
 - Token processing
 - Sequence mixing
 - Prediction
- Example architectures

Token processing

- Token processing turns a word or a token into a combination of a **word embedding** and a **positional embedding**



- Fundamental question:
what numerical representation of a word/token is more useful?
- **Word (token) embeddings** are the vectors representing words (tokens).
- **One-hot representation** uses standard basis vectors in $\mathbb{R}^{|\mathcal{V}|}$.
 - What can go wrong?

Vocab

one-hot representation

a	$e_1 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
ability	$e_2 = [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
able	$e_3 = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
about	$e_4 = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
above	$e_5 = [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
acarus	$e_6 = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, \dots]$

⋮

⏟
vocab size = $|\mathcal{V}|$

- Fundamental question:
what numerical representation of a word/token is more useful?
- **Word (token) embeddings** are the vectors representing words (tokens).
- **One-hot representation** uses standard basis vectors in $\mathbb{R}^{|\mathcal{V}|}$.
 - **Relations** between words are lost and similarity cannot be measured.
 - **Dimension** of the vector is too large, e.g., for prediction

Vocab

one-hot representation

a	$e_1 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
ability	$e_2 = [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
able	$e_3 = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
about	$e_4 = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
above	$e_5 = [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, \dots]$
acarus	$e_6 = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, \dots]$

⋮

⏟
vocab size = $|\mathcal{V}|$

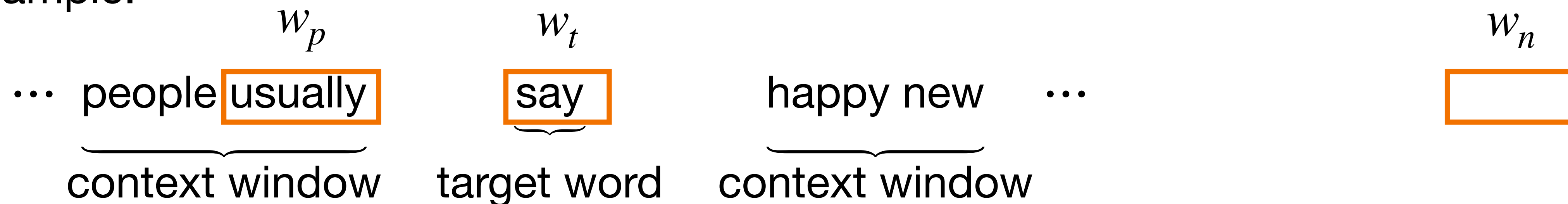
- Breakthrough in word embedding: **Word2Vec** [Mikolov et al. 2013] and **Glove** [Pennington et al. 2014]
 - capture **similarities and relations** between words via training on a text corpus
 - gave **significant gain** when used together with older model architectures like RNN and LSTM
- Word2vec is trained to predict a missing word given a context window, for example, of size 2, before and after the masked word:

\dots people usually say happy new \dots

 context window target word context window

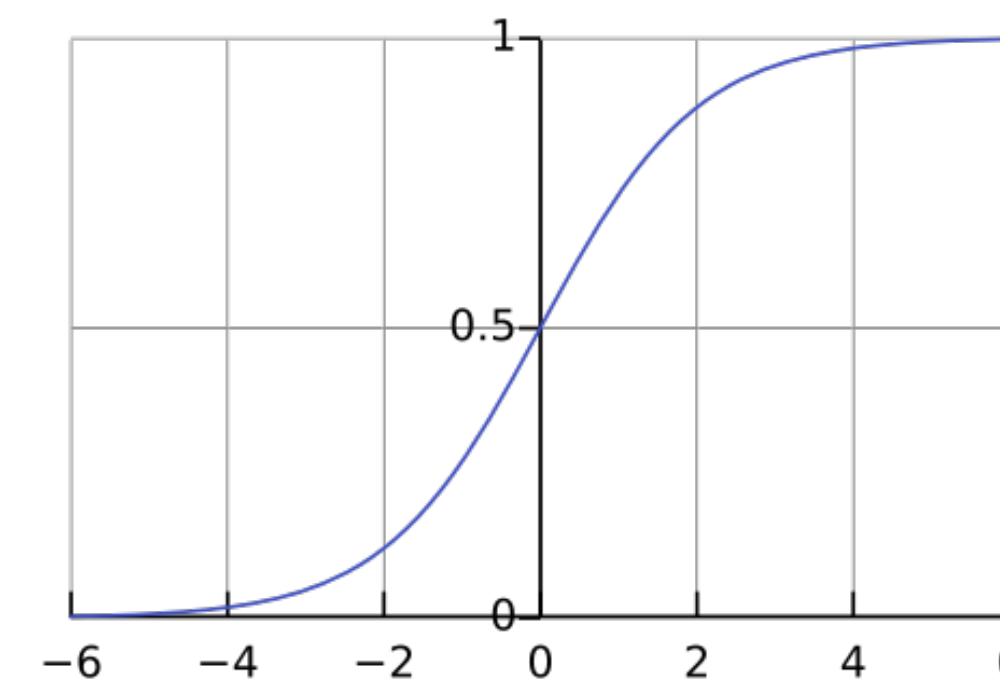
- The i -th word in the vocabulary is associated with two vector embeddings in m -dimensions,
 - $t_i \in \mathbb{R}^m$ is the embedding when it is a target word, and
 - $c_i \in \mathbb{R}^m$ is the embedding when it is used as a context.
- After these embeddings are learned, the word is represented by either the summation or the concatenation of these two vectors, when used in an LM.
- m is about 200~300 in practice.

- example sample:



- Word2vec embeddings are trained on a corpus of text, where for each context window the loss is defined over a tuple (w_t, w_c, w_n) , where

- w_t is the **target word** in the sample with embedding t_{w_t}
- w_p is one of the **context words** (called positive sample) with c_{w_p}
- w_n is **not a context word** (called negative sample) with c_{w_n}
- Word2vec is learned from text corpus to minimize a loss function:



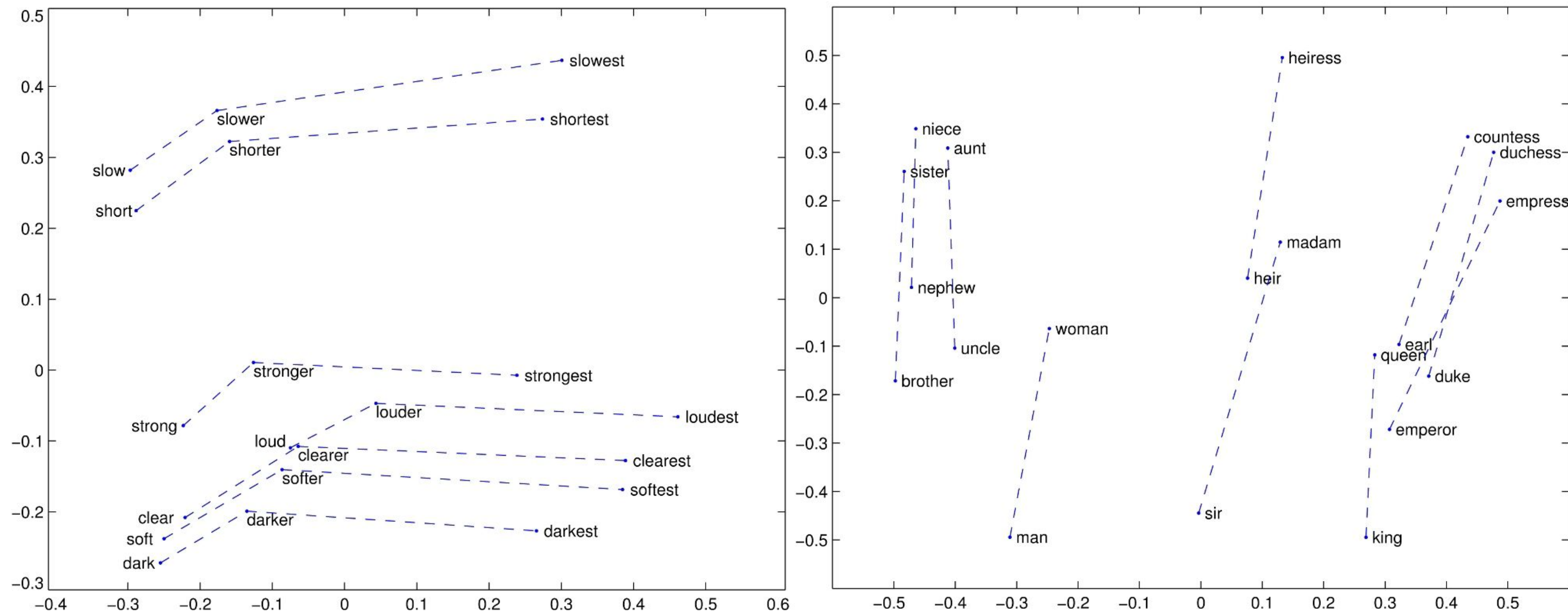
$$\begin{aligned}
 L &= -\log\left(\mathbb{P}(w_t \text{ is a true target of } w_p) \cdot \mathbb{P}(w_t \text{ is not a true target of } w_n)\right) \\
 &= -\log(\sigma(\langle t_{w_t}, c_{w_p} \rangle)) - \log(1 - \sigma(\langle t_{w_t}, c_{w_n} \rangle)) \\
 &= -\log\left(\frac{1}{1 + \exp(-\langle t_{w_t}, c_{w_p} \rangle)}\right) - \log\left(1 - \frac{1}{1 + \exp(-\langle t_{w_t}, c_{w_n} \rangle)}\right)
 \end{aligned}$$

Inner product \uparrow for positive ex.

Inner product \downarrow for negative ex.

- Word2vec was a breakthrough due to its simplicity, efficient training, semantic quality, and vector arithmetic (which was not intended at all!)
 - for example, other than the missing word prediction that it was trained to solve, word2vec was amazing at analogy tasks:
king : man = ? : woman,
can be solved by **vector arithmetic**, outputting the embedding that is the **nearest neighbor**

$$x_{\text{king}} + x_{\text{woman}} - x_{\text{man}} \simeq x_{\text{queen}}$$



- It is quite surprising that **local rules** of co-occurrences, i.e., how often do two words appear together in a context window, are enough to learn such **rich semantic relations**.

- Building upon the success of Word2vec, for a while, the typical approach has been to use these word vectors, and train specific model for each task: translation, sentiment analysis, etc.
- Typical embedding dimension is 300, which is not too large, but the number of words can be huge, perhaps in the order of 1,000,000.
- No longer used, as subword tokenization is more popular than words, and embeddings learned together with model weights perform better than word2vec.
- Modern embeddings for subword tokens are **learned** in pretraining stage, with an exception of a part of the embedding called the **positional embedding**, which contains information about the word's position in the sentence.
- token vocab ~ 200k, and token embedding vectors of dimension 700~12000 are trained.

- **Positional embeddings** are designed to capture the positions of the input sequence of words to a transformer.
- Without positional encoding, the original transformer treats the input as a set, in which case the following two inputs are treated the same (more to follow when we learn architectures):

“I am happy” vs. “Am I happy”

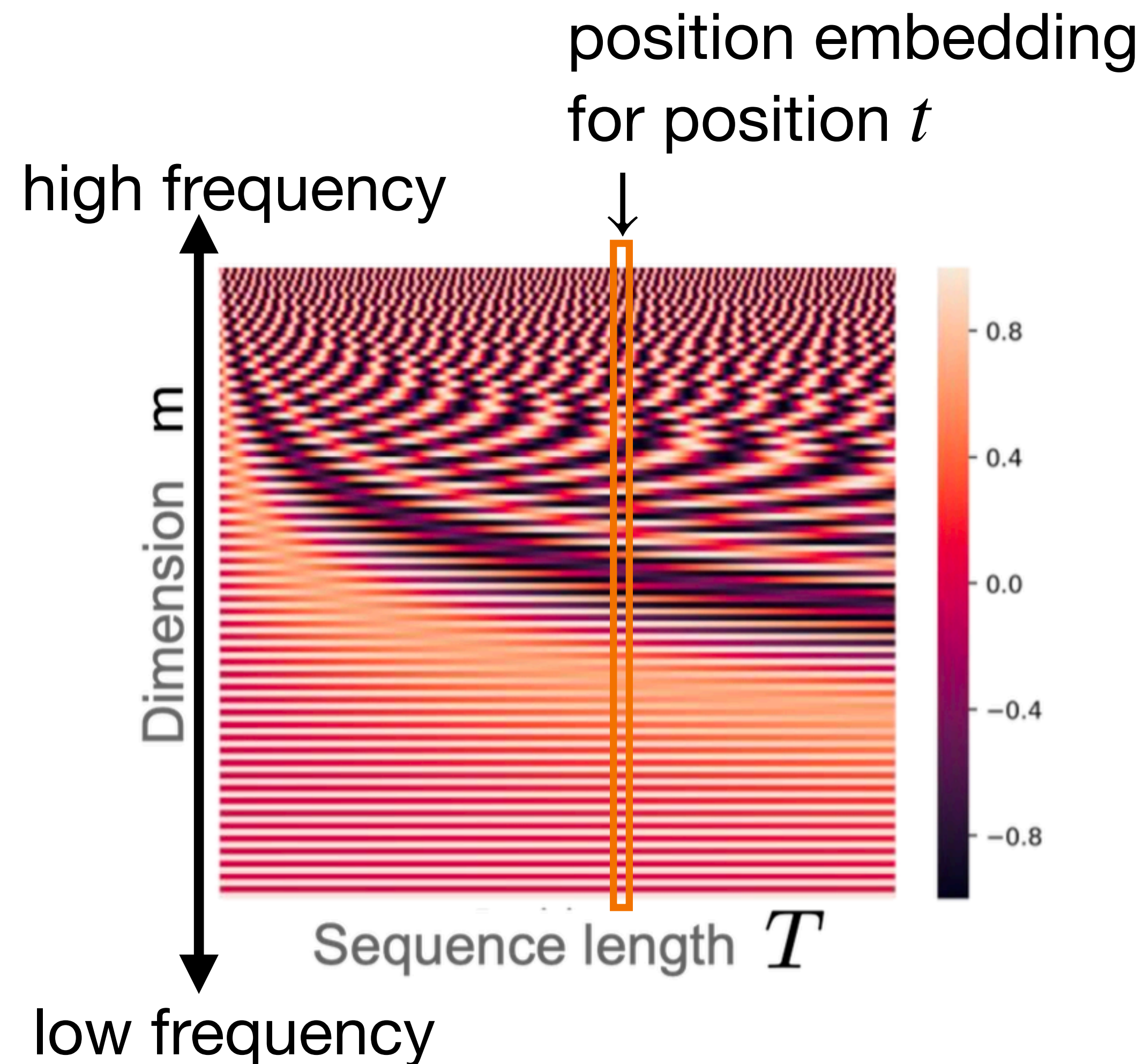
- To take into account the order of the input, **absolute positional embedding** represents a word by concatenating its learned semantic embedding with an absolute position of the word, for example,
- **Attempt 1**: its position index t can be used as absolute position embedding:

$$x_t = (\text{WordEmb}(w_t), t)$$

but this is challenging to generalize to unseen sequence lengths, and the range of the value of the embedding increases with t .

- Absolute positional embedding **solution 1**: original transformer paper [Vaswani et al. 2017] proposes using alternating $\sin(\cdot)$ and $\cos(\cdot)$ functions of decreasing frequencies at position index t , **added** to the vector word embedding:

$$x_t = [\text{WordEmb}(w_t) ; \underbrace{\begin{bmatrix} \sin(t/10000^{0/m}) \\ \cos(t/10000^{0/m}) \\ \sin(t/10000^{2/m}) \\ \cos(t/10000^{2/m}) \\ \vdots \\ \sin(t/10000^{1/2}) \\ \cos(t/10000^{1/2}) \end{bmatrix}}_{\text{position embedding}}]$$



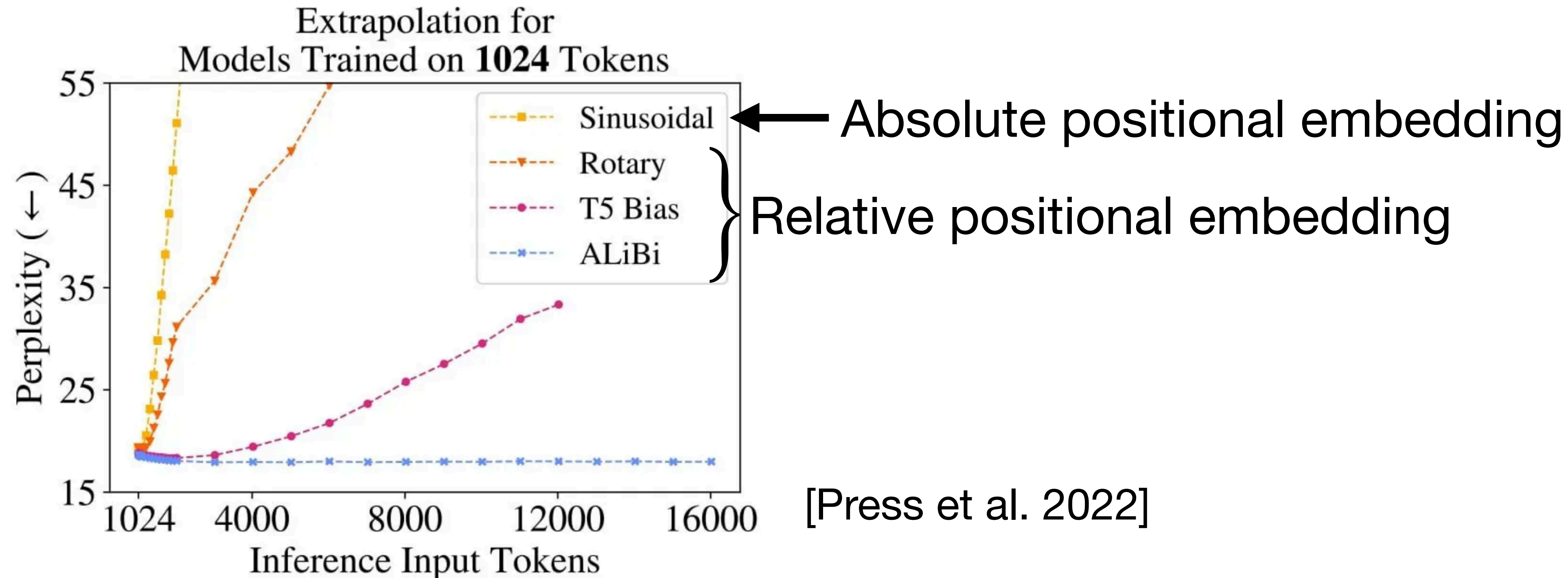
- Absolute positional embedding **solution 2**: learned positional embedding.

$$x_t = [\text{WordEmb}(w_t) ; \text{PosEmb}(t)]$$

- Empirical performance is similar for the two absolute positional embeddings: sinusoidal and learned
- Learned positional embedding is popular in vision transformers

- **Absolute positional embeddings** encode the absolute position of the word in the sequence, which has two problems:

- it is hard to **extrapolate** to sequence lengths unseen during training,



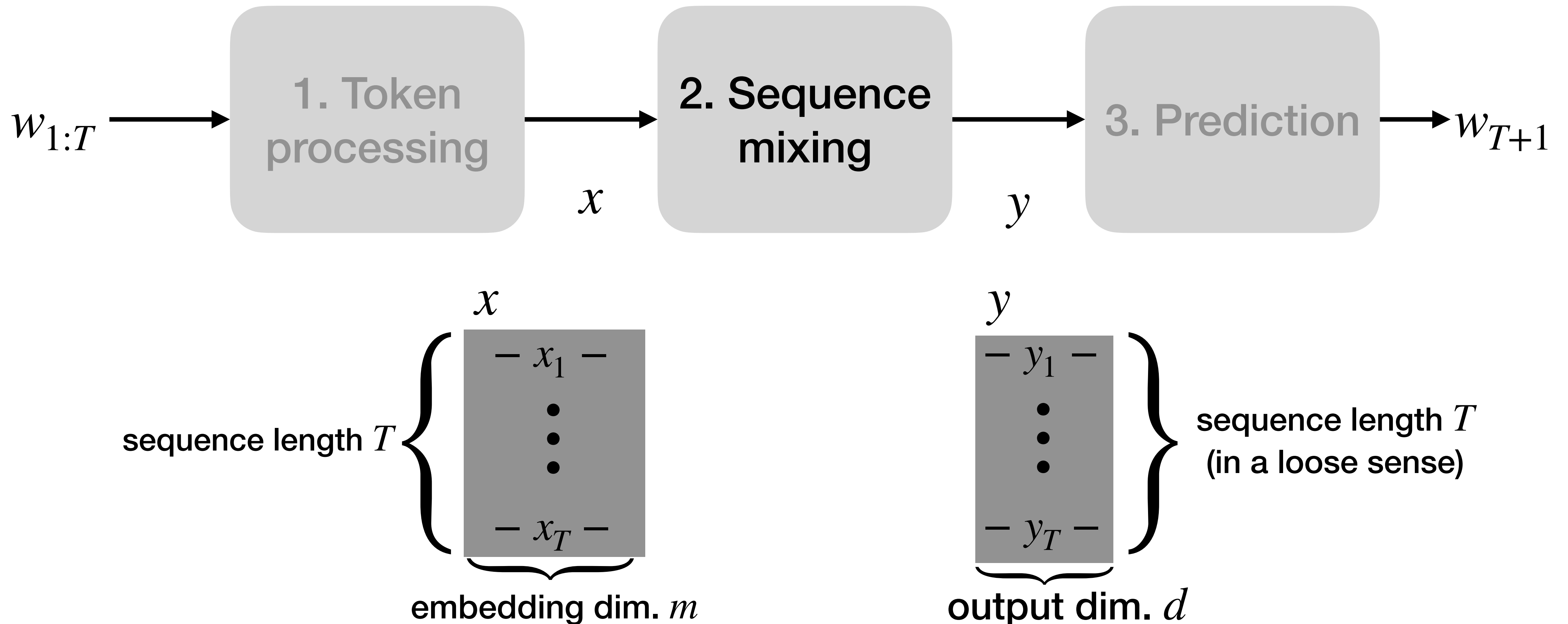
- and **relative position** is as important as absolute position, for example, “happy new ?” appearing in positions (1,2,3) have similar meaning as appearing in positions (500,501,502).
- Relative positional encoding addresses both: generalize to sequences of unseen lengths by relying on the pairwise distances between two words. We will skip the details of relative positional encoding and refer to the cited references.

Outline

- Language models
- General LLM framework
 - Token processing
 - Sequence mixing
 - Prediction
- Example architectures

Sequence mixing

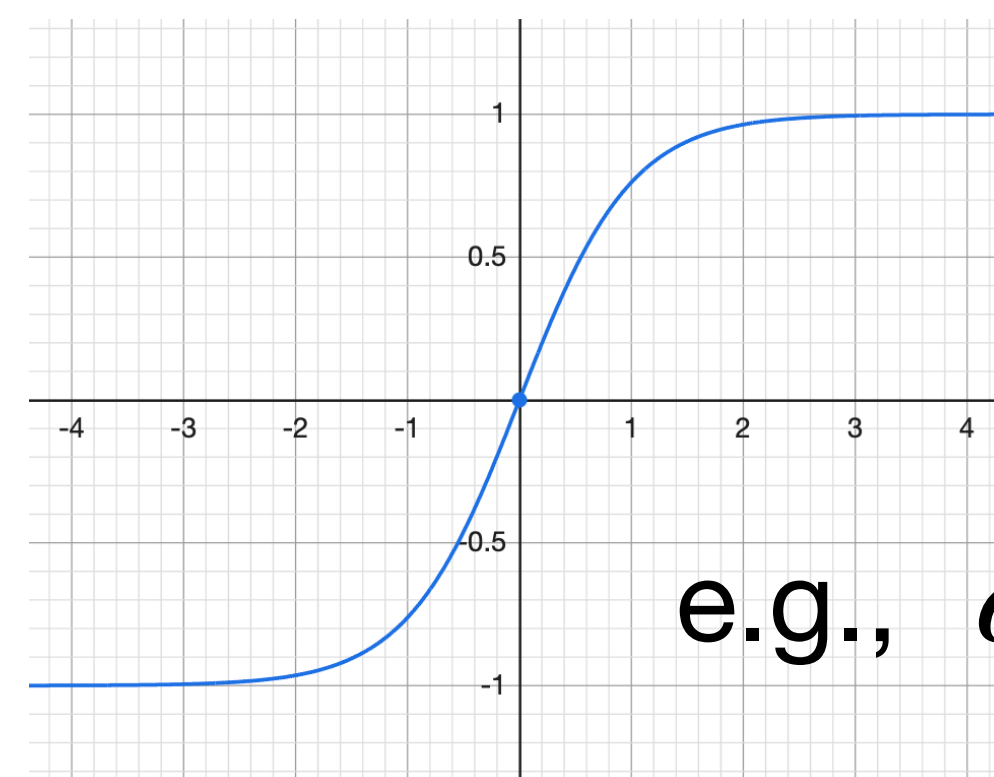
- Sequence mixing is one of the most well studied part of an LM that captures the dependencies across tokens.



- **Sequence mixer** captures the hidden correlation across the tokens in a sequence.
- Markov in 1913 [Markov 1913] used **Markov chains** to predict whether an upcoming letter would be a vowel or a consonant.
 - **Markov assumption:** the probability of a next word only depends on the past $N - 1$ words, i.e., $\mathbb{P}(w_t | w_{1:t-1}) = \mathbb{P}(w_t | w_{t-N+1:t-1})$.

- Markov chain with a window size of N can be captured by the N -gram models we just studied a few slides ago. But the learning of the Markov chain $\mathbb{P}(w_t | w_{t-N+1:t-1})$ can be made efficient using neural networks, for example **Feed Forward Neural networks** that computes

$$\text{Emb}(w_t) = \sigma \left(W \begin{bmatrix} \text{Emb}(w_{t-N+1}) \\ \vdots \\ \text{Emb}(w_{t-1}) \end{bmatrix} \right)$$



e.g., $\sigma(x) = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$

for an input of past $N - 1$ word embeddings, weight matrix $W \in \mathbb{R}^{m \times m(N-1)}$, entry-wise scalar activation function $\sigma(\cdot)$ of choice, and output of a word embedding.

- **Feed Forward Neural networks (FFNs)** as a sequence mixer uses N recent tokens to predict next token similar to N -gram models (differentiable and parametric version of N -gram models).
 - Model weight $W \in \mathbb{R}^{d \times Nm}$ takes as input N token embeddings, each of dimension m , and outputs a d -dimensional representation.
- **Forward pass in pre-training**, the pseudocode is for a single sentence (using only two recent tokens, i.e., $N = 2$), and training proceeds by taking gradient of this loss and using stochastic gradient descent (SGD)

- Initialize the input $x_0 \leftarrow 0$ and loss $L \leftarrow 0$
- **For** $t = 1, \dots, T$

$$y_t \leftarrow \sigma\left(W \begin{bmatrix} x_{t-1} \\ x_t \end{bmatrix}\right),$$

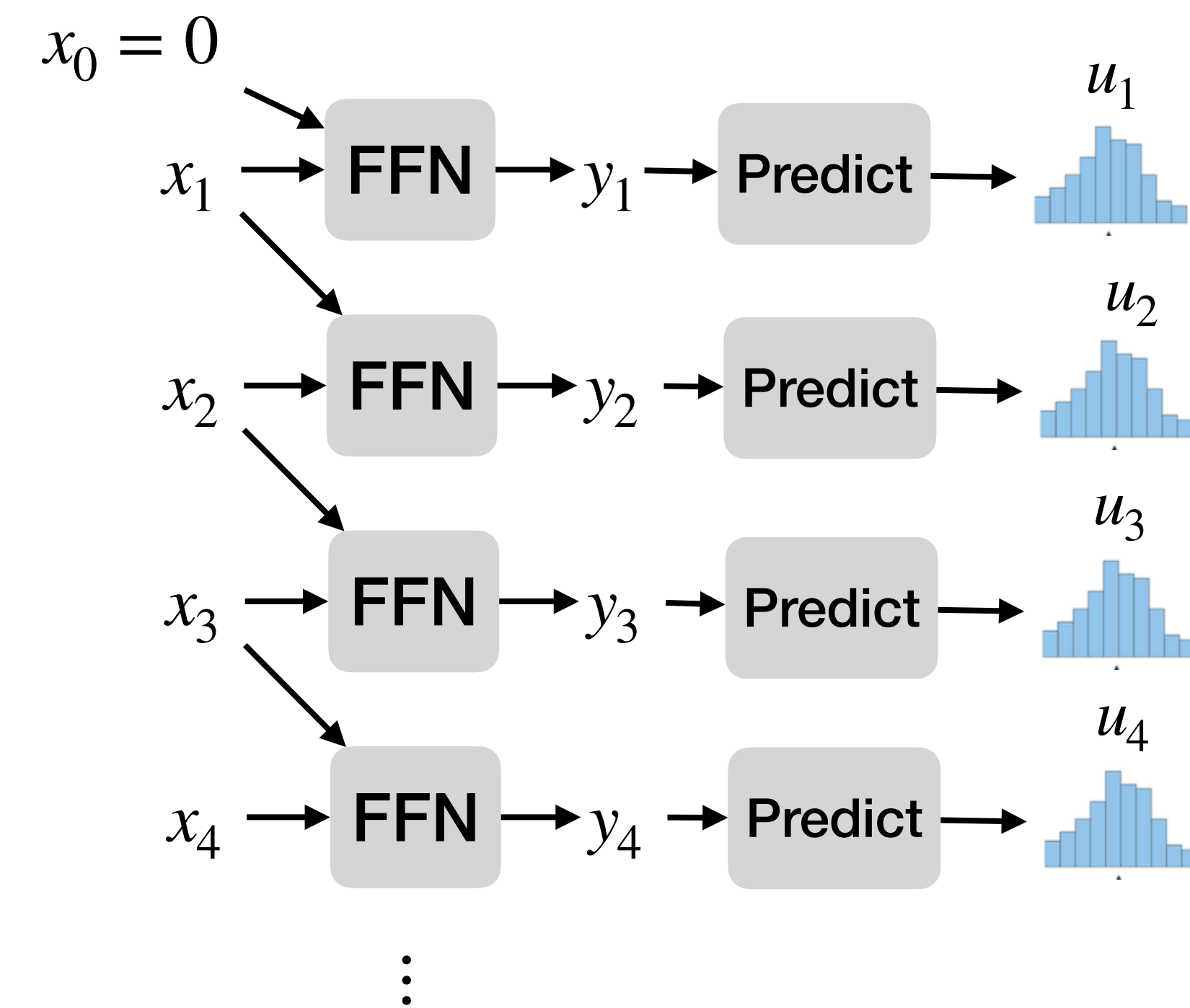
FFN

$$u_t \leftarrow \text{Predict}(y_t),$$

probability

$$L += \left(\sum_{i \in \mathcal{V}} - \mathbf{I}_{(i=w_{t+1})} \log(u_t^{[i]}) \right)$$

cross-entropy loss



- A longer window with larger N increases the model size, and short N cannot capture longer dependencies.

- **Recurrent Neural Networks (RNNs)** as a sequence mixer tracks a **hidden state** $h_t \in \mathbb{R}^{d_h}$ to capture the temporal dependencies, and uses it to compute the output $y_t \in \mathbb{R}^d$ as

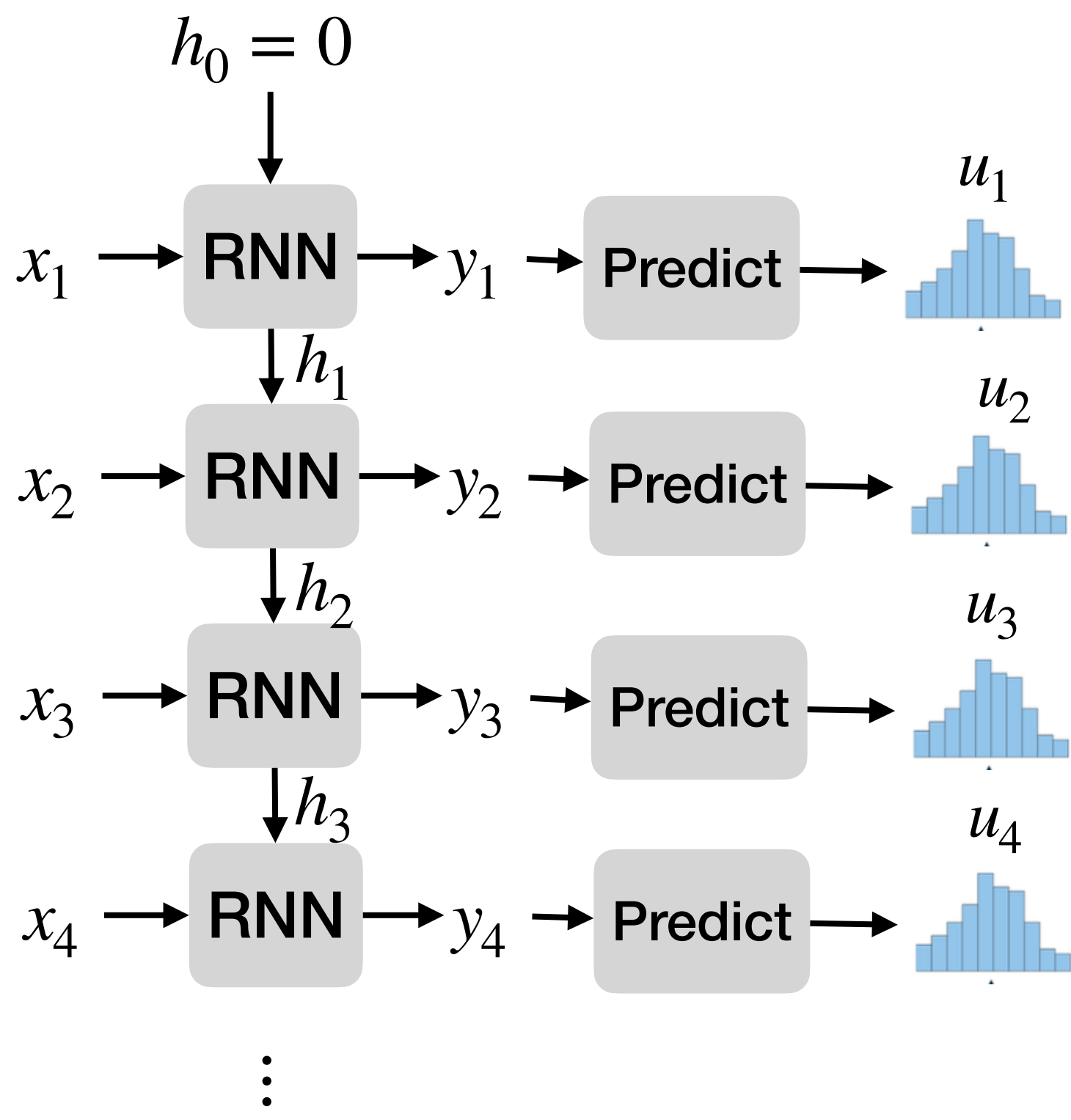
$$h_t = g(x_t, h_{t-1}), \text{ and}$$

$$y_t = f(h_t)$$

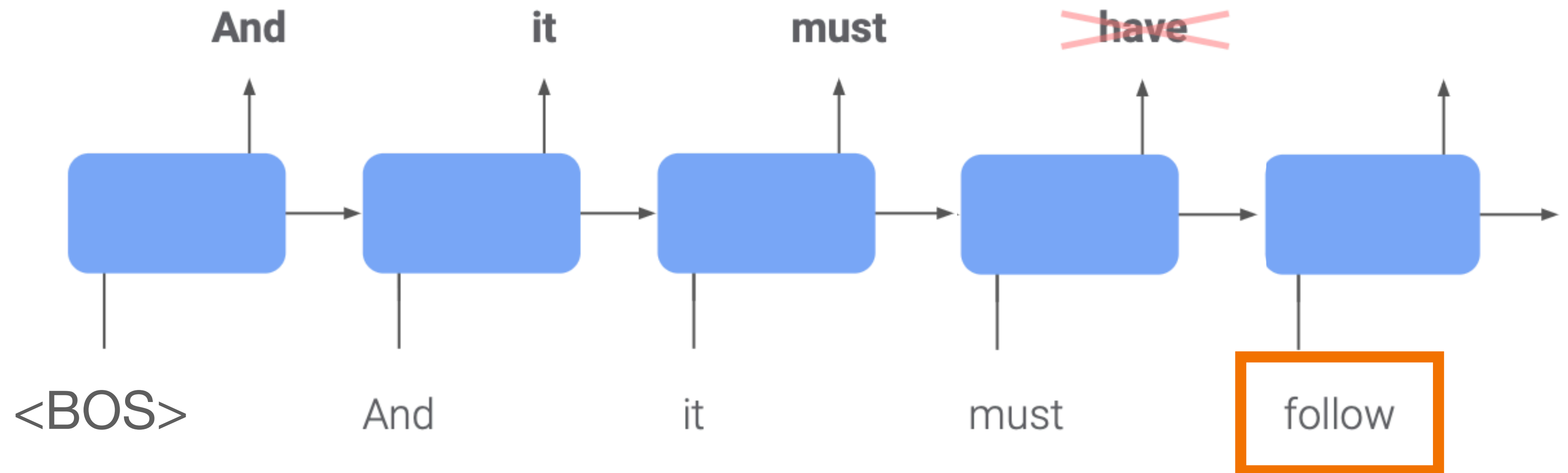
where $g(\cdot)$ and $f(\cdot)$ are parametric learnable functions, e.g., multiple layers of FFNs, that are *the same across all positions in the sequence*, motivated by **hidden Markov models**.

- Forward pass in **pre-training** for RNN, the pseudocode is for a single sentence

- Initialize the state $h_0 \leftarrow 0$ and loss $L \leftarrow 0$
- **For** $t = 1, \dots, T$
 - $h_t \leftarrow g(x_t, h_{t-1}),$
 - $y_t \leftarrow f(h_t),$
 - $u_t \leftarrow \text{Predict}(y_t),$
 - $L += \left(\sum_{i \in \mathcal{V}} - \mathbf{I}_{(i=w_{t+1})} \log(u_t^{[i]}) \right)$ cross-entropy loss



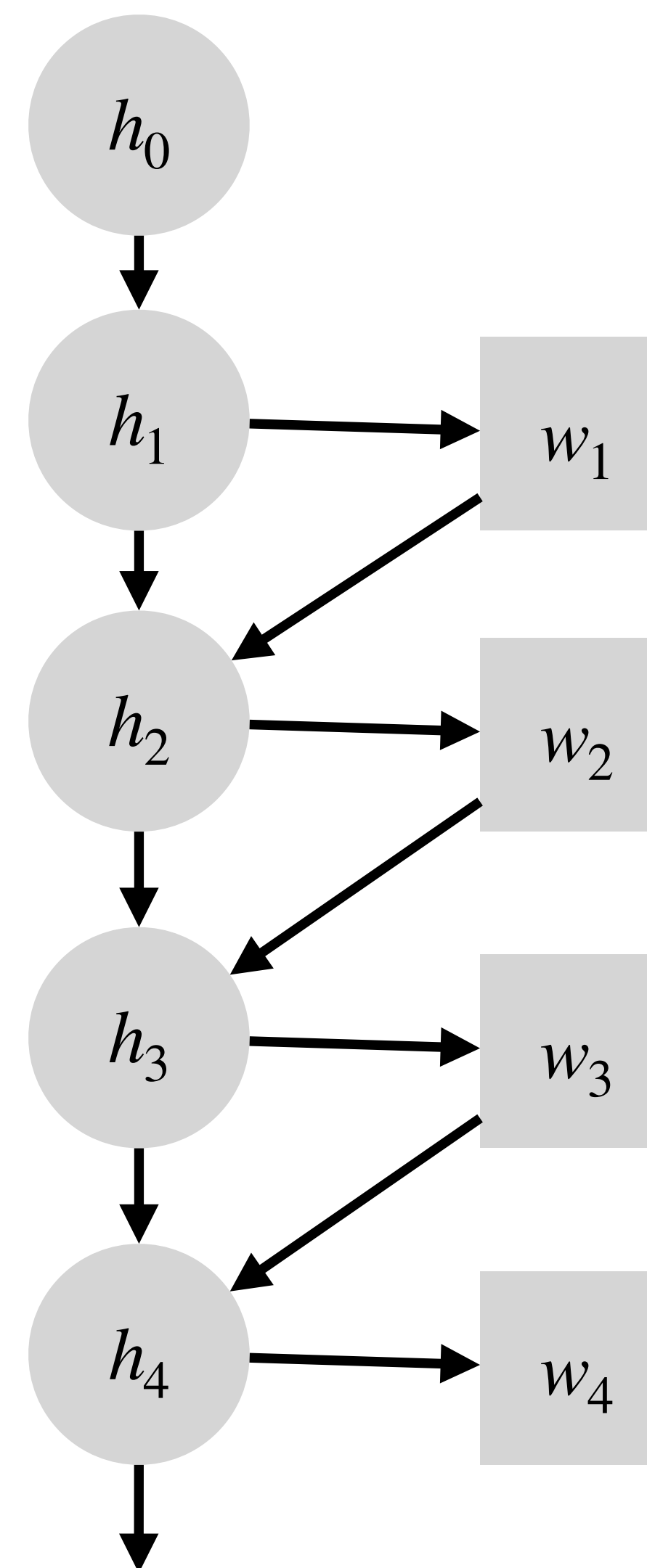
- RNNs are trained via **teacher forcing** on next word prediction:
 - Given a training sequence of words, [And, it, must, follow, ...], each RNN step predicts the next word, which is compared to the ground truths and the loss is computed.
 - The model output is not fed back into the model during training. The next RNN step is fed the ground truth word as an input, and not the prediction from the previous step.



- RNN-based LM is motivated by **Hidden Markov Models (HMMs)**, which can capture temporal dependencies in a hidden state, h_t , that is not observed.
 - The probability of a next word only depends on the hidden state as

$$\mathbb{P}(w_t | w_{1:t-1}, h_{1:t-1}) = \mathbb{P}(w_t | h_{t-1}) \text{ and}$$

$$\mathbb{P}(h_t | w_{1:t}, h_{1:t-1}) = \mathbb{P}(h_t | w_t, h_{t-1}).$$



graphical model representation of a HMM
see CSE515 if you want to learn a lot more.

- At **inference-time**, RNN models run auto-regressive inference over forward passes

- Initialize the state $h_0 \leftarrow 0$, $t \leftarrow 1$,
and $x_1 \leftarrow \text{Emb}(\langle \text{BOS} \rangle)$

- While True:**

$$h_t \leftarrow g(x_t, h_{t-1}),$$

$$y_t \leftarrow f(h_t), \quad \text{RNN}$$

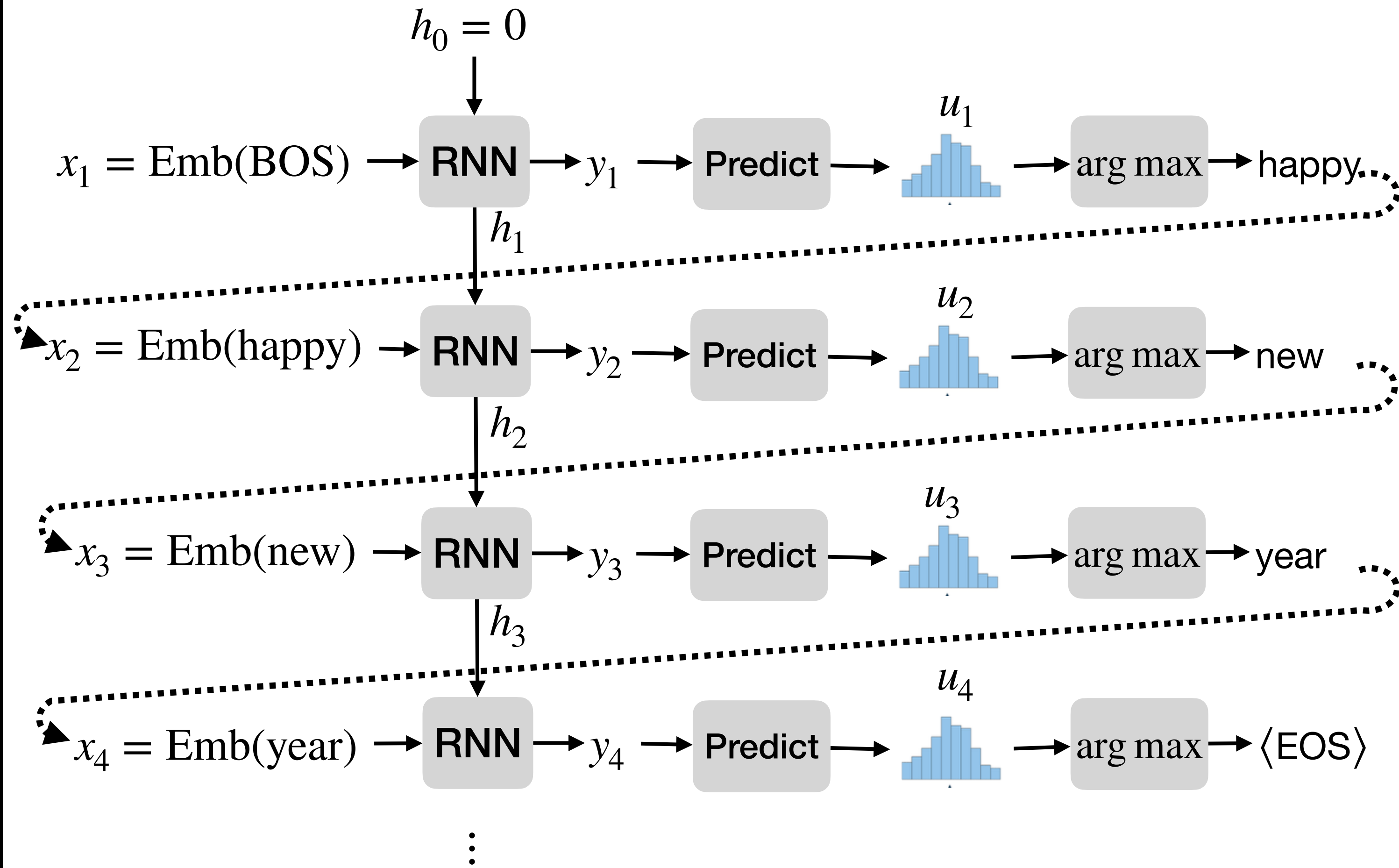
$$u_t \leftarrow \text{Predict}(y_t), \quad \text{probability}$$

Set x_{t+1} as the embedding of the token
for $\arg \max_i u_t^{[i]} x_{t+1} = \text{Emb}(\text{new})$

If x_{t+1} is $\text{Emb}(\langle \text{EOS} \rangle)$: **break**

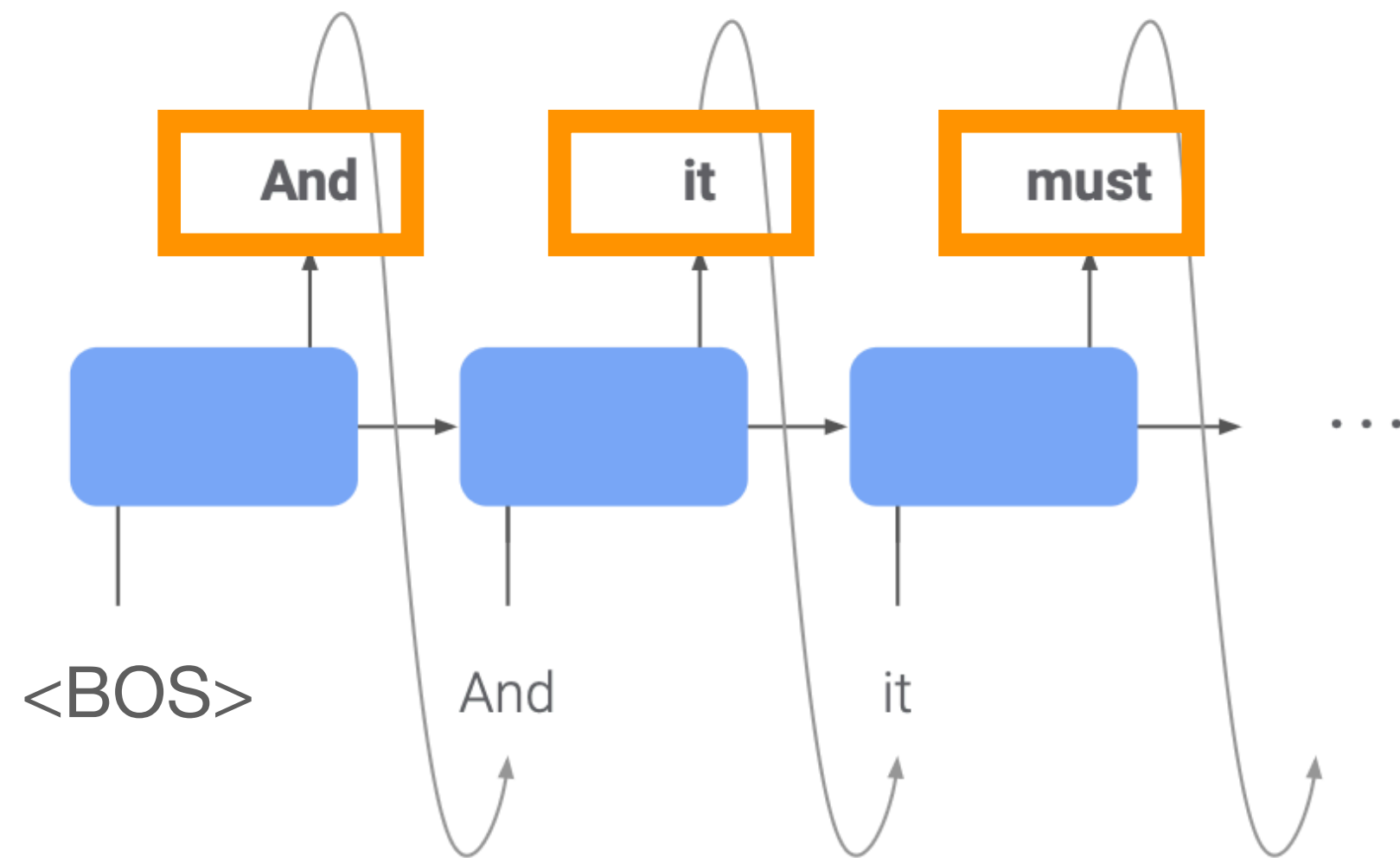
$$t \leftarrow t + 1$$

- Output:** $x = [x_1, \dots, x_t], y = [y_1, \dots, y_{t+1}]$.

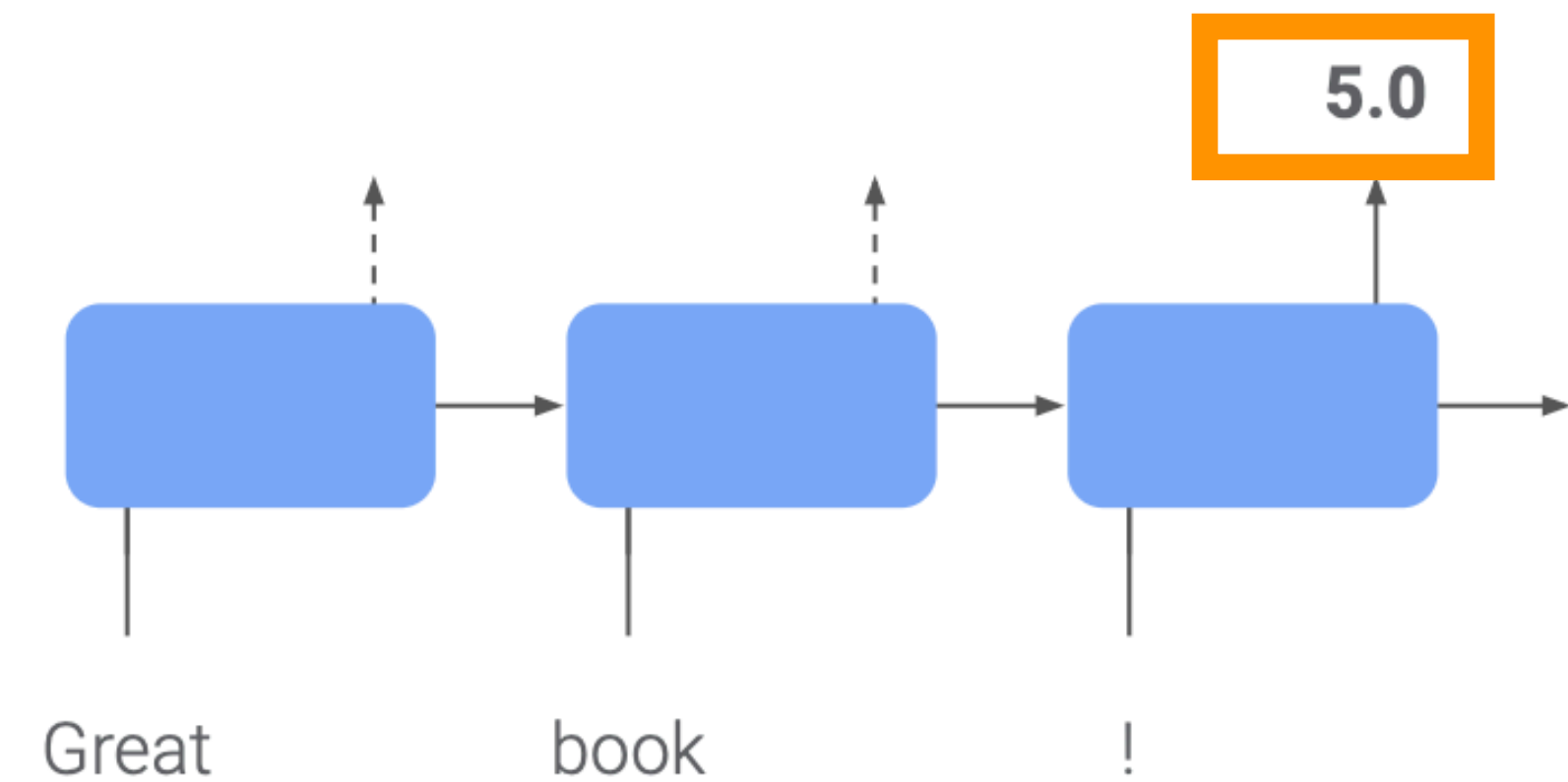


- Next word prediction is a fundamental task central to various other linguistic tasks, but how do we use RNN-based LMs to solve them?

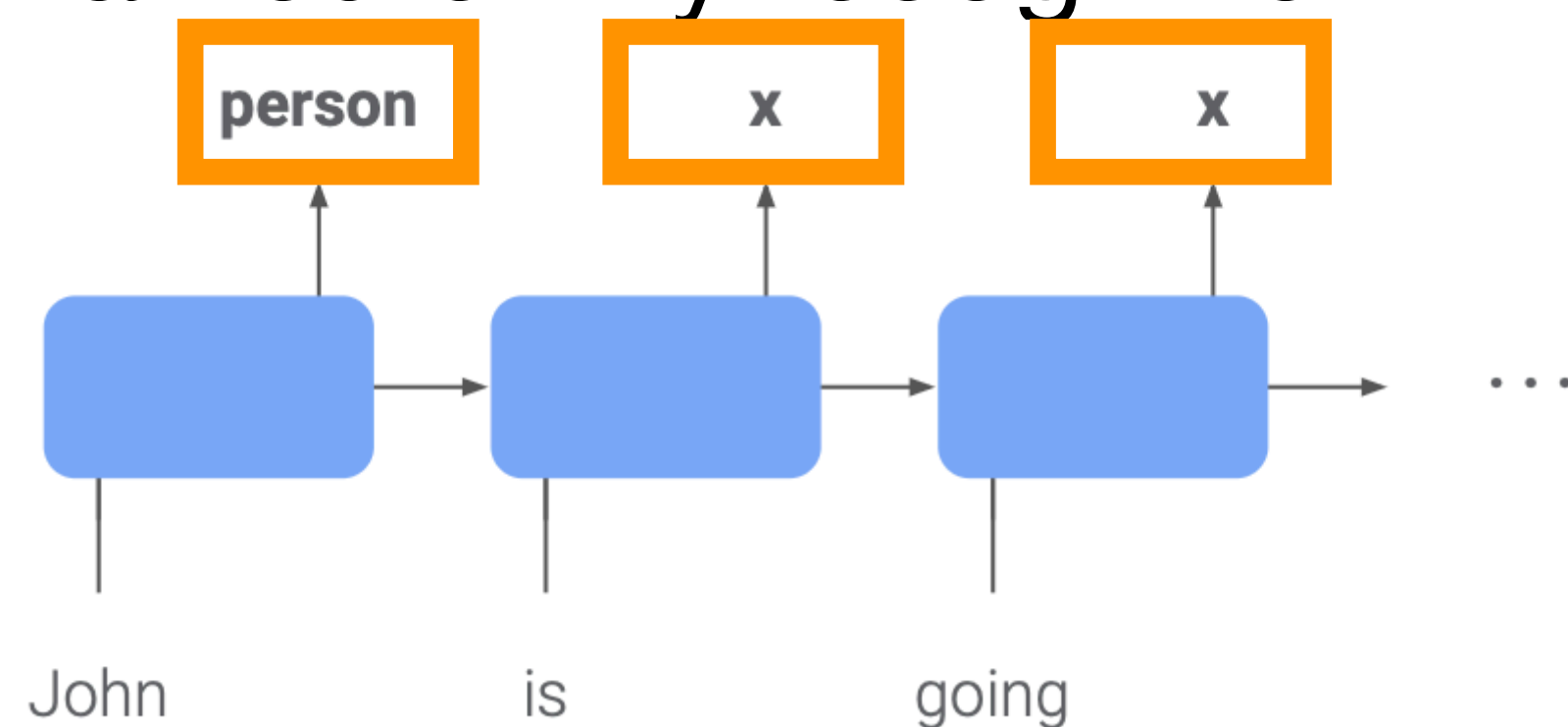
Next word prediction



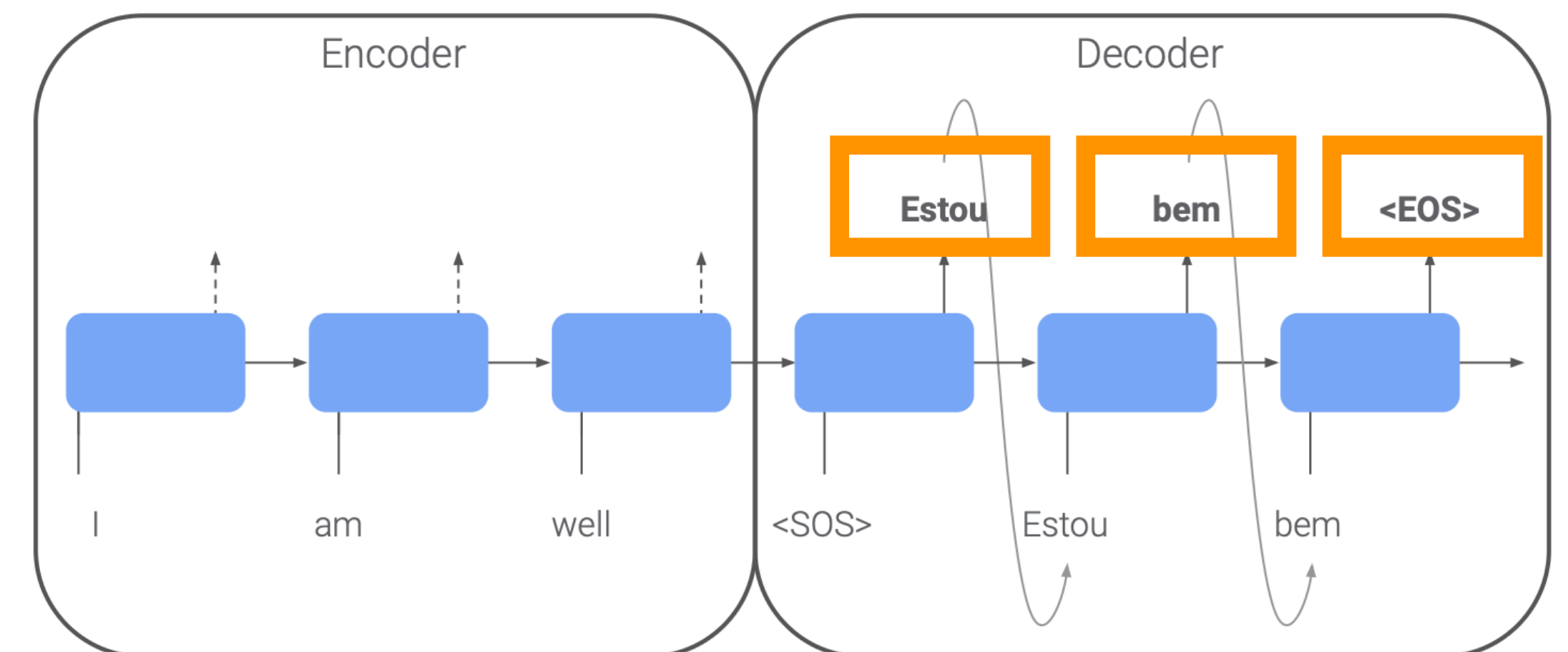
Sentiment analysis in 5 star rating



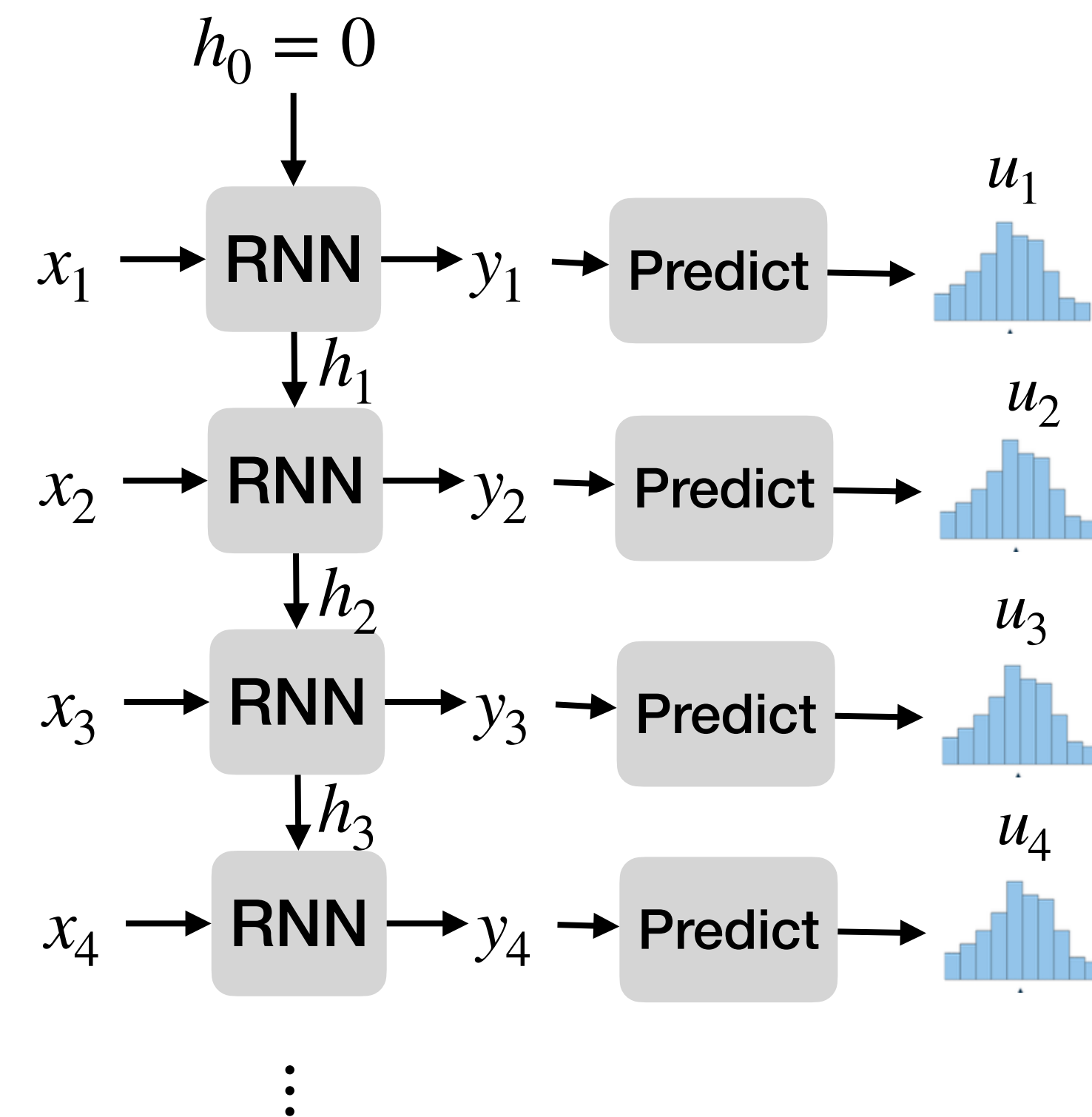
Named entity recognition



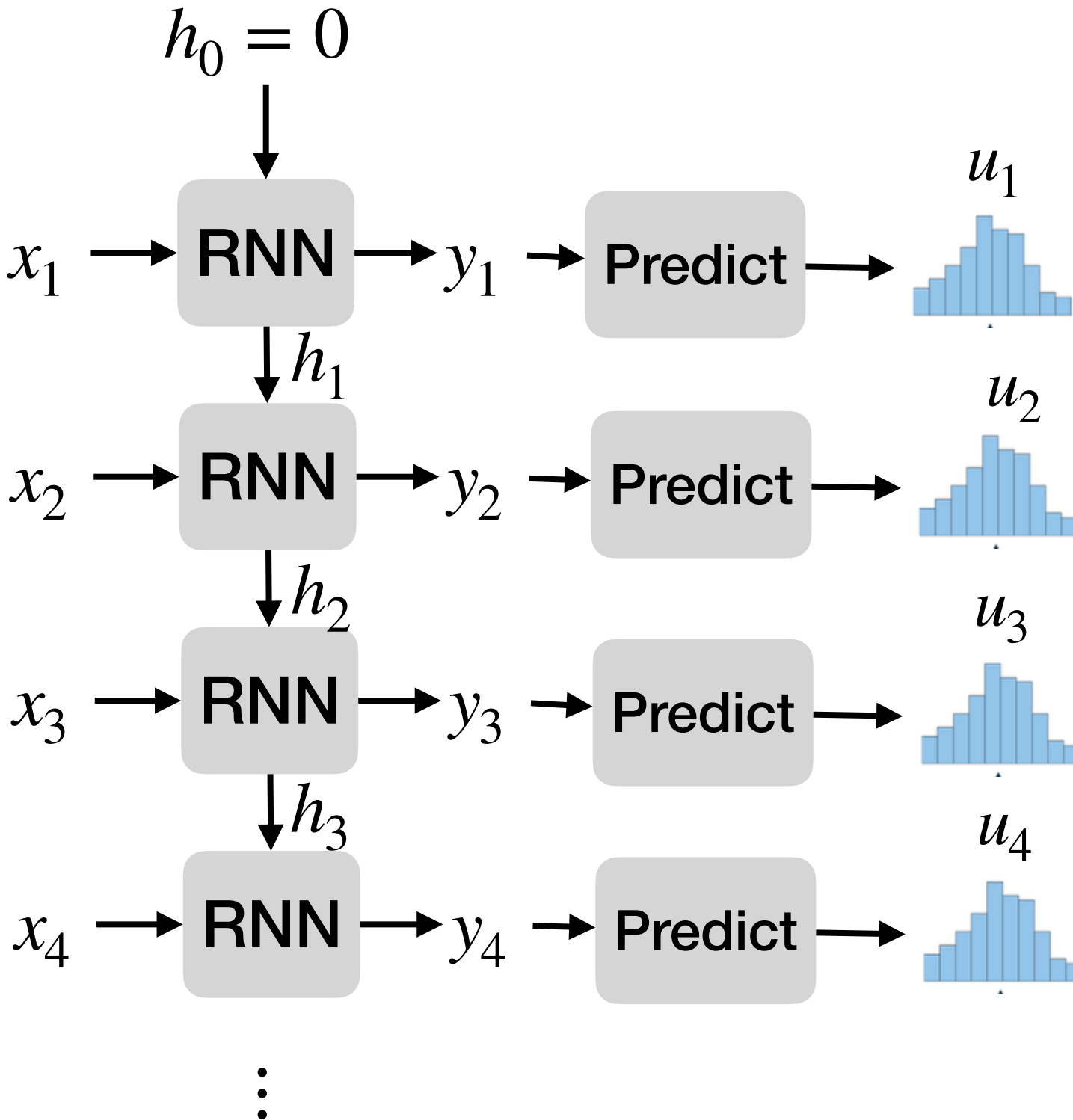
Machine translation



- RNN architectures only partially address the long-range dependency problem, since information in the hidden state gets diluted over time.
- Following problems persist:
 - long sequences have vanishing or exploding gradients [Pascanu et al. 2013, Hochreiter et al. 1997],
 - mode collapse (i.e., generating repetitive outputs),
 - struggle with highly variable input sizes due to limited memory in the hidden state, acting as a bottleneck.
- Resource considerations:
 - Inference memory: $O(d)$.
 - Training complexity: $O(Td)$
 - Training time: no parallelization $O(T)$
- Many attempts to tackle these problems: LSTM [Hochreiter et al. 1997], GRUs [Cho et al. 2014]



- RNN architectures only partially address the long-range dependency problem, since information in the hidden state gets diluted over time.
- What could go wrong?



Sources

- Other courses in LLMs that the lecture slides are based on
 - CSE493S/599S at UW by Ludwig Schmidt: <https://mlfoundations.github.io/advancedml-sp23/>
 - EE-628 at EPFL by Volkan Cevher: <https://www.epfl.ch/labs/lions/teaching/ee-628-training-large-language-models/ee-628-slides-2025/>
- Useful blog posts
 - <https://azizbelaweid.substack.com/p/complete-summary-of-absolute-relative>
- Dan Jurafsky and James H. Martin. Speech and Language Processing (3rd ed. draft). draft, third edition, 2023.
- Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. “Efficient estimation of word representations in vector space”, In International Conference on Learning Representations, 2013.
- Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "Glove: Global vectors for word representation", Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.
- Ofir Press, Noah A. Smith^{1,3} Mike Lewis² , “Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation”, In International Conference on Learning Representations, 2022
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, “Attention Is All You Need”, In Neural Information Processing Systems, 2017
- Beitong Zhou, Cheng Cheng, Guijun Ma, and Yong Zhang. “Remaining useful life prediction of lithium-ion battery based on attention mechanism with positional encoding”, In IOP Conference Series: Materials Science and Engineering, 2020.
- Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks.” In International Conference on Machine Learning, 2013

- Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory.” In *Neural Computation*, 9(8):1735–1780, 11 1997.
- Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. “Learning phrase representations using rnn encoder-decoder for statistical machine translation”, In *ACL 2014*
- Andrey Andreyevich Markov. “Essai d’une recherche statistique sur le texte du roman. ‘Eugene Onegin’ illustrant la liaison des epreuve en chain”. In: *Izvestia Imperatorskoi Akademii Nauk (Bulletin de l’Académie Impériale des Sciences de St.-Pétersbourg)*. 6th ser, 7:153–162, 1913.
-