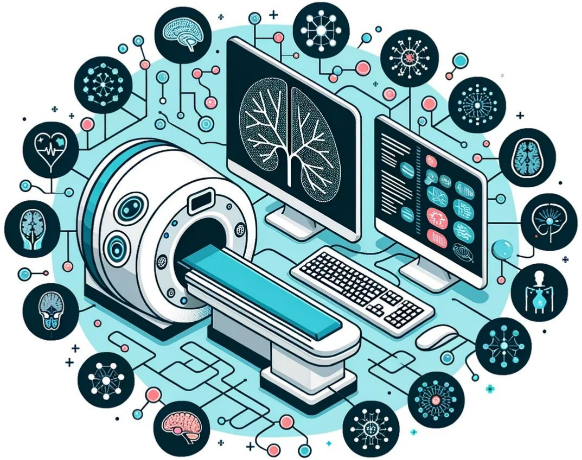
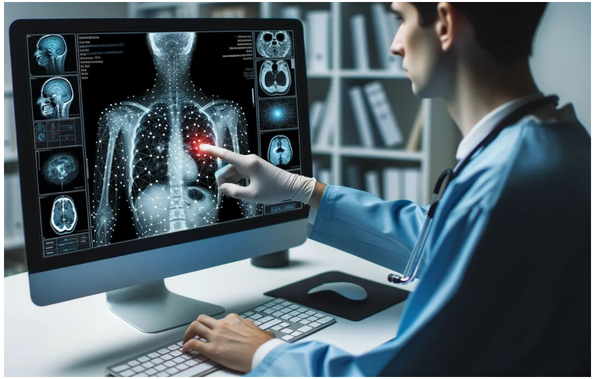
# Applied Deep Learning and Generative Models in Healthcare





**Session 4:** RNNs and Transformers
**Date:** Feb 01 2025

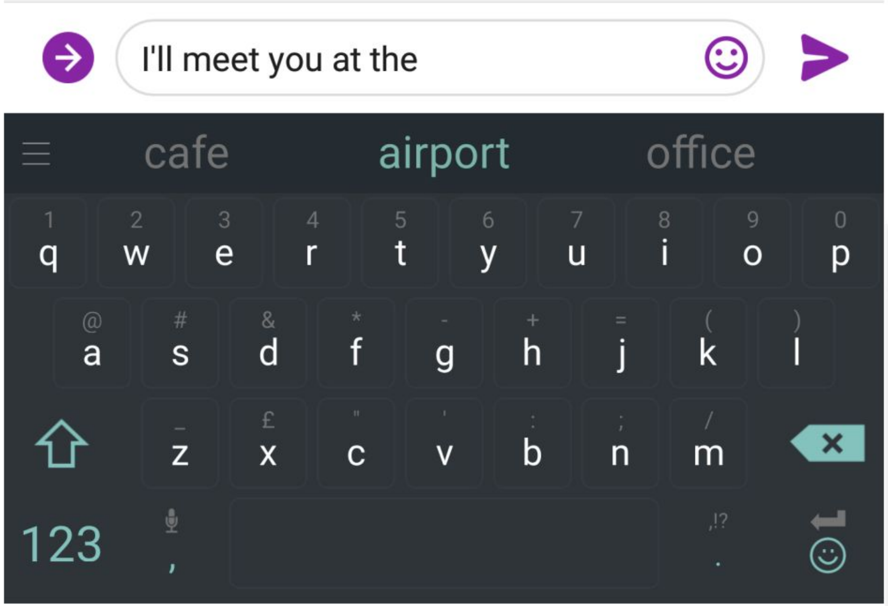
**Instructor:** Mahmoud E. Khani, Ph.D.

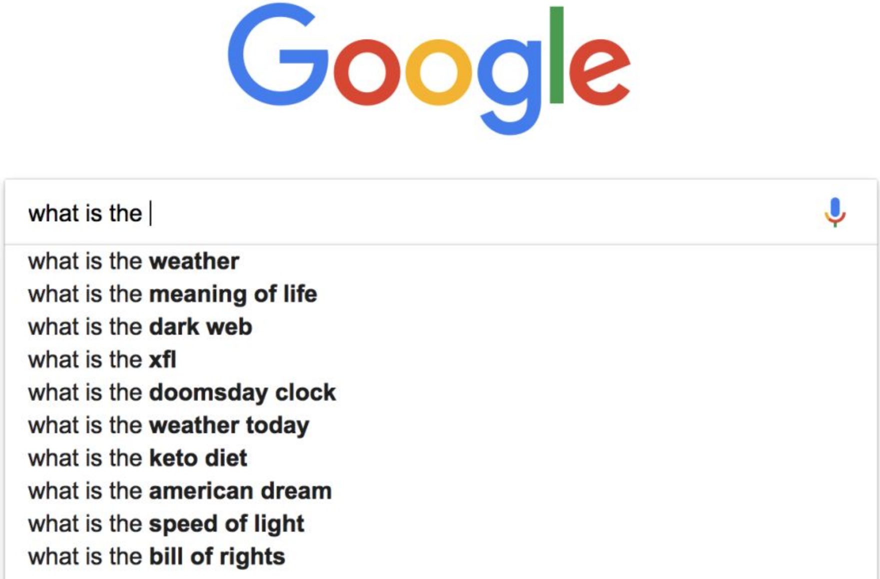
# Learning goals

- Basics first: feed-forward networks, recurrent networks, attention
- Then key methods used in NLP in 2025: transformers, encoder-decoder models, pretraining, post-training (RLHF, SFT), efficient adaptation, model interpretability, language model agents, etc.

# We use language models every day!

- Language modeling is the task of predicting what word comes next!



# Language modeling

- **Given a sequence of words, compute the probability distribution of the next word:**

$$P(\boldsymbol{x}^{(t+1)} \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

where $x^{(t+1)}$ can be any word in the vocabulary $V = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{|V|}\}$

$$\begin{aligned} P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}) &= P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)} \mid \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)} \mid \boldsymbol{x}^{(T-1)}, \ldots, \boldsymbol{x}^{(1)}) \\ &= \underbrace{\prod_{t=1}^{T} P(\boldsymbol{x}^{(t)} \mid \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})}_{\text{This is what the LM provides}} \end{aligned}$$

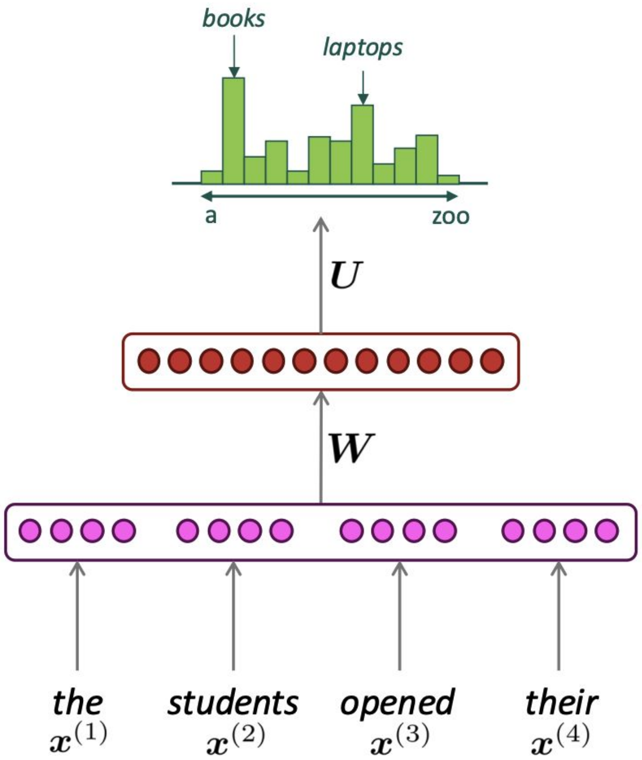
# Language modeling applications

- Speech recognition
- Handwriting recognition
- Machine translation
- Summarization
- Dialogue
- Authorship identification
- Spelling/grammar correction
- ChatGPT is an LM!

# Language modeling with neural networks

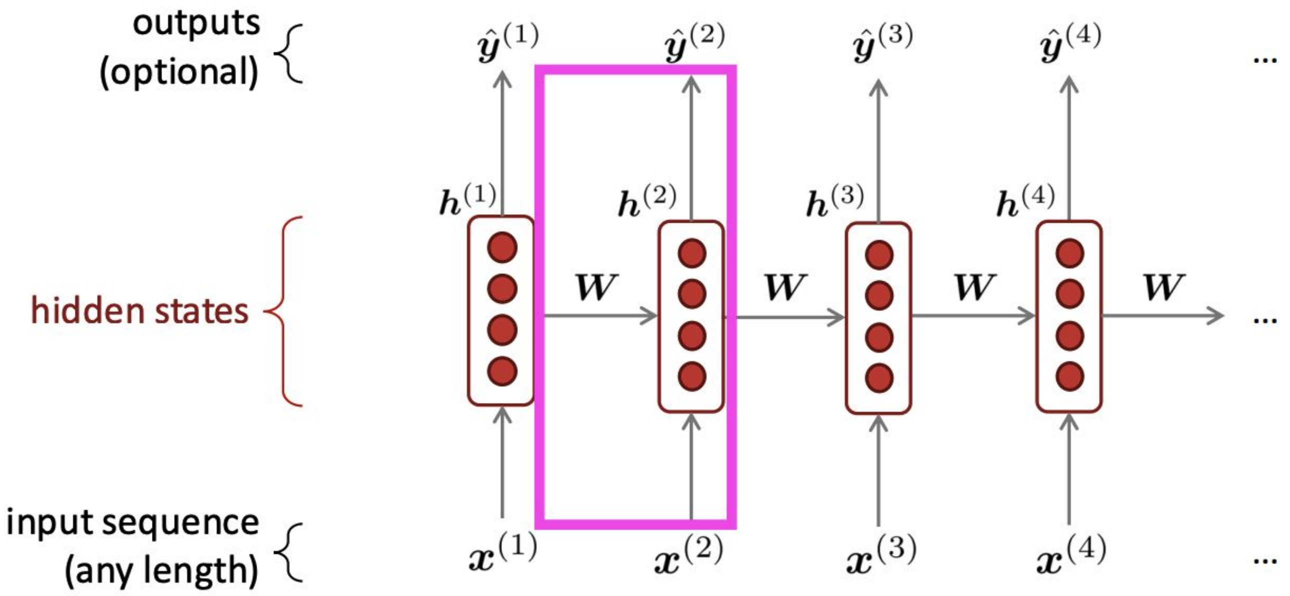
- A neural probabilistic language model (Y. Bengio, et al.)
- Fixed window is small
- No window is large enough

We need a neural architecture that can process *any length input*



# Recurrent neural networks

- Apply the same weights **W** *repeatedly*
- Input can be of any length!



# A simple RNN language model

output distribution

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

hidden states

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

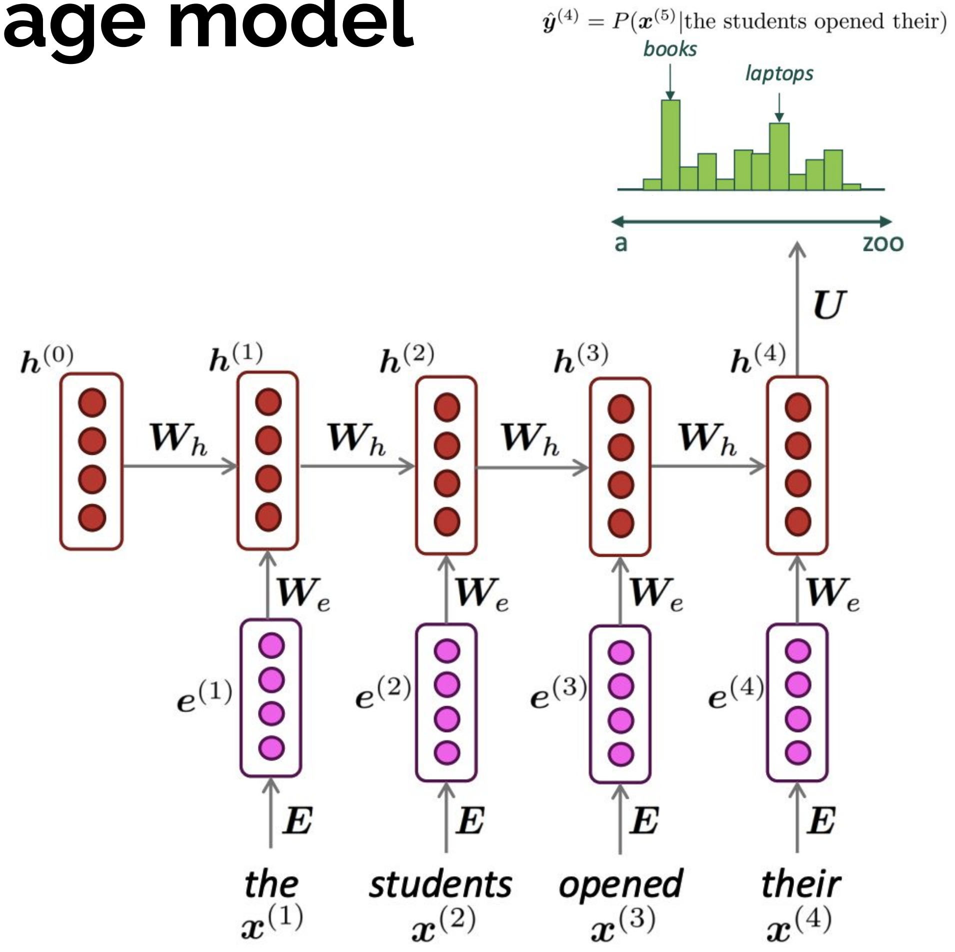
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

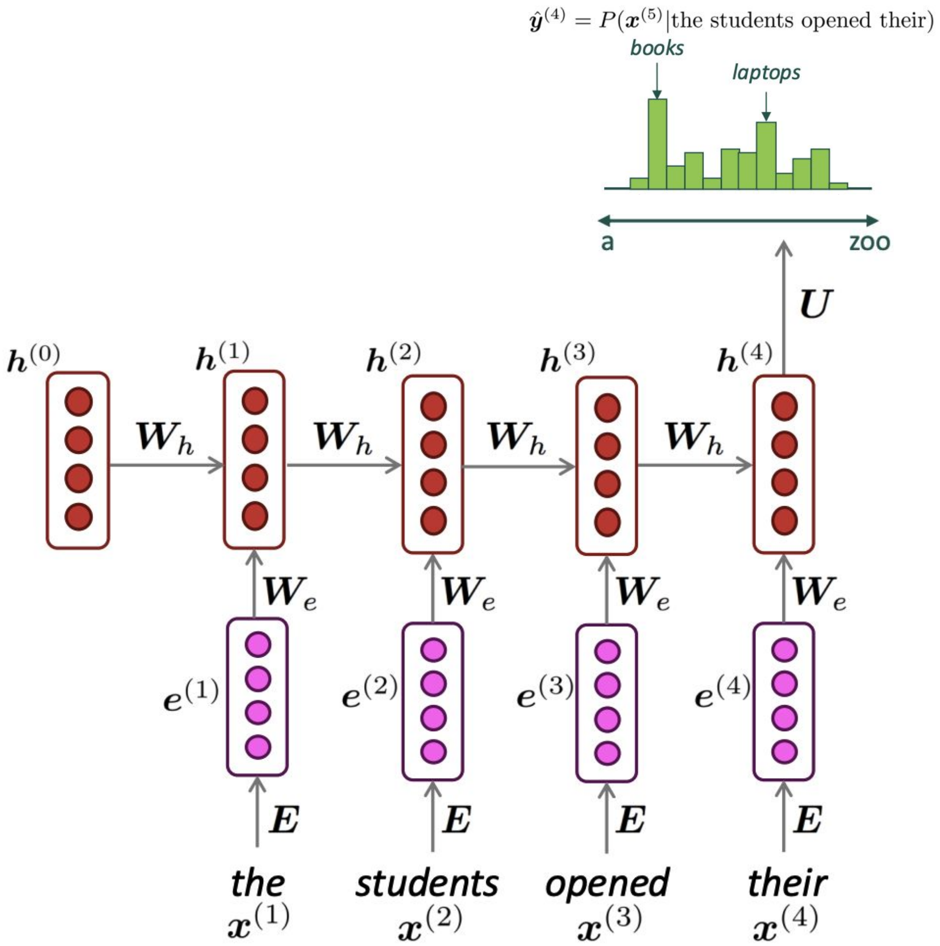
words / one-hot vectors

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



# RNN LMs

- **Advantages**
  - The process any length!
  - Can use information from previous steps
  - Model size does not increase for longer input context
  - Same weights applied on every timestep
- **Disadvantages**
  - Slow
  - Difficult to access information from many steps back



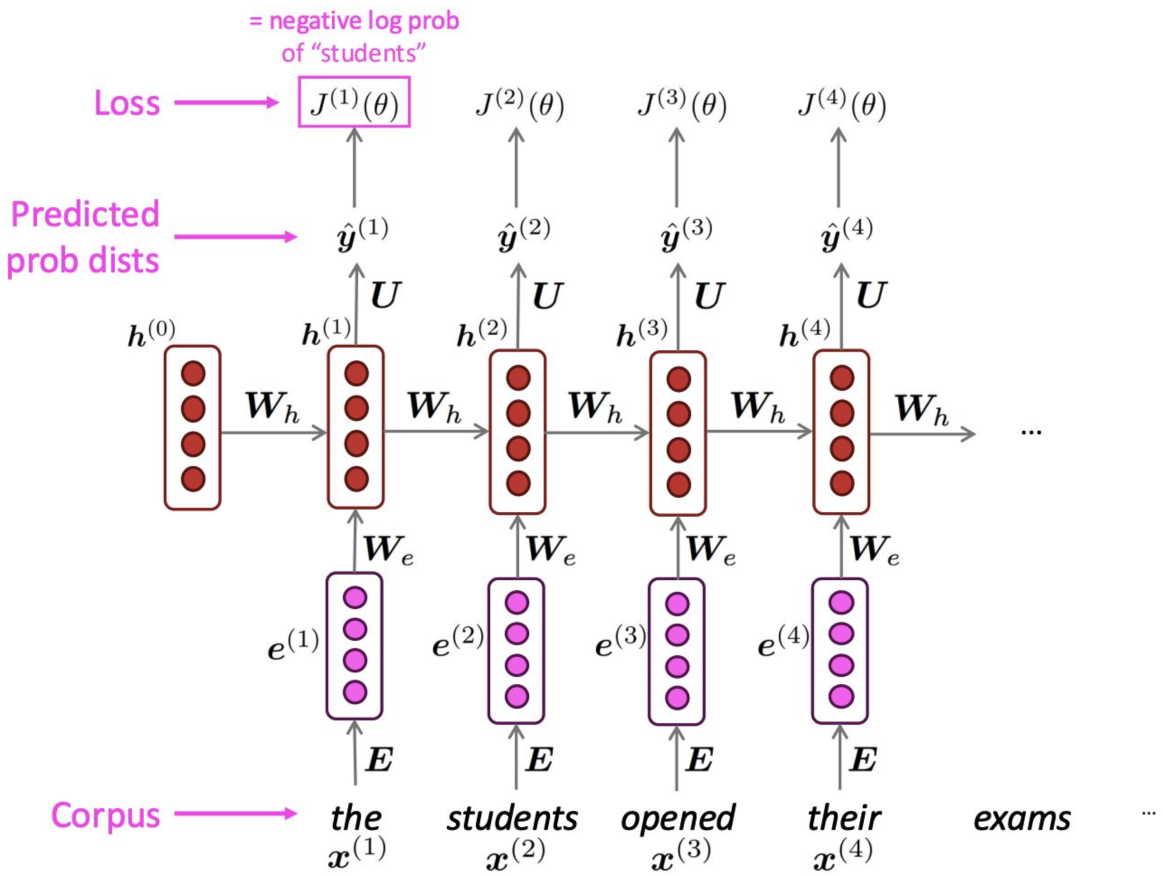
# Training an RNN language model

- **Get a big corpus of text, i.e., sequence of** $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$
- **Feed into RNN, compute output distribution** $\hat{\boldsymbol{y}}^{(t)}$
  - Predict probability dist of every word, given words so far
- **Loss function is cross-entropy between predicted probability** $\hat{\boldsymbol{y}}^{(t)}$ **, and the true next word** $\boldsymbol{y}^{(t)}$

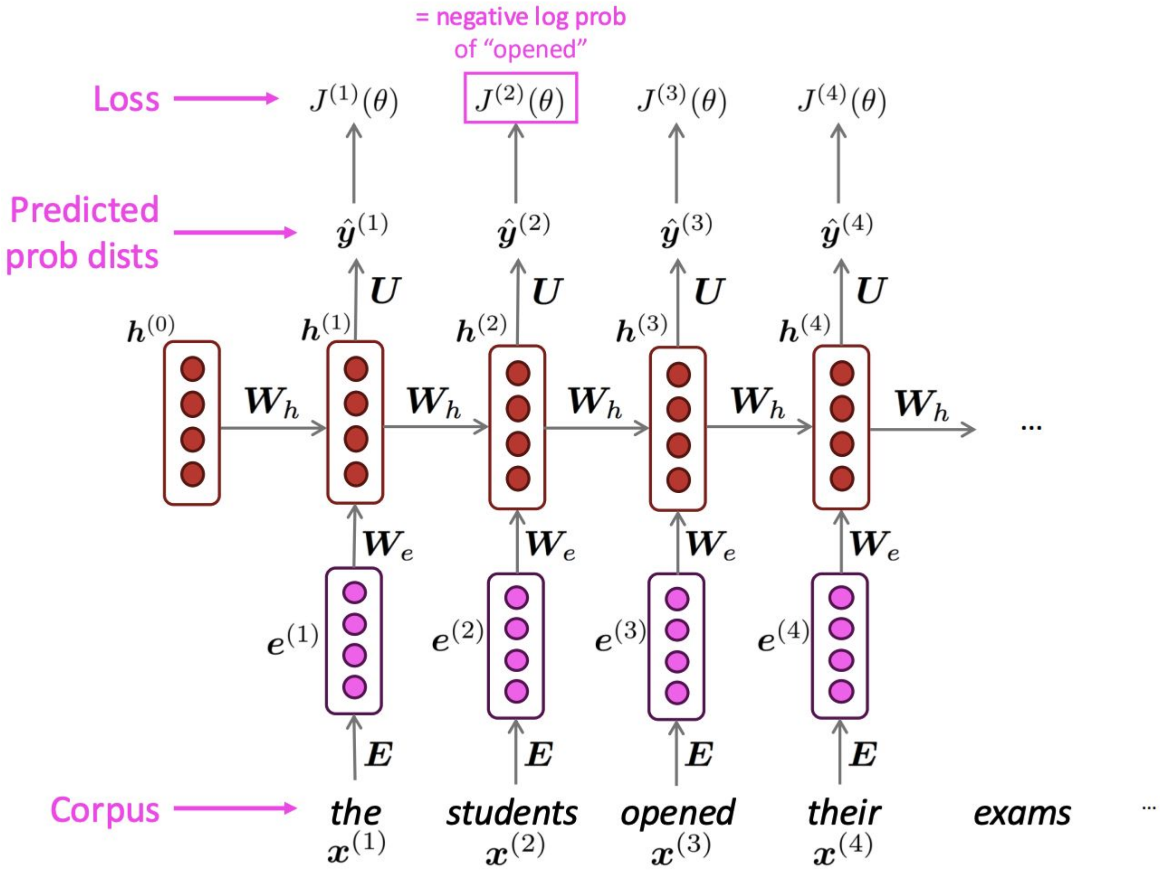
$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

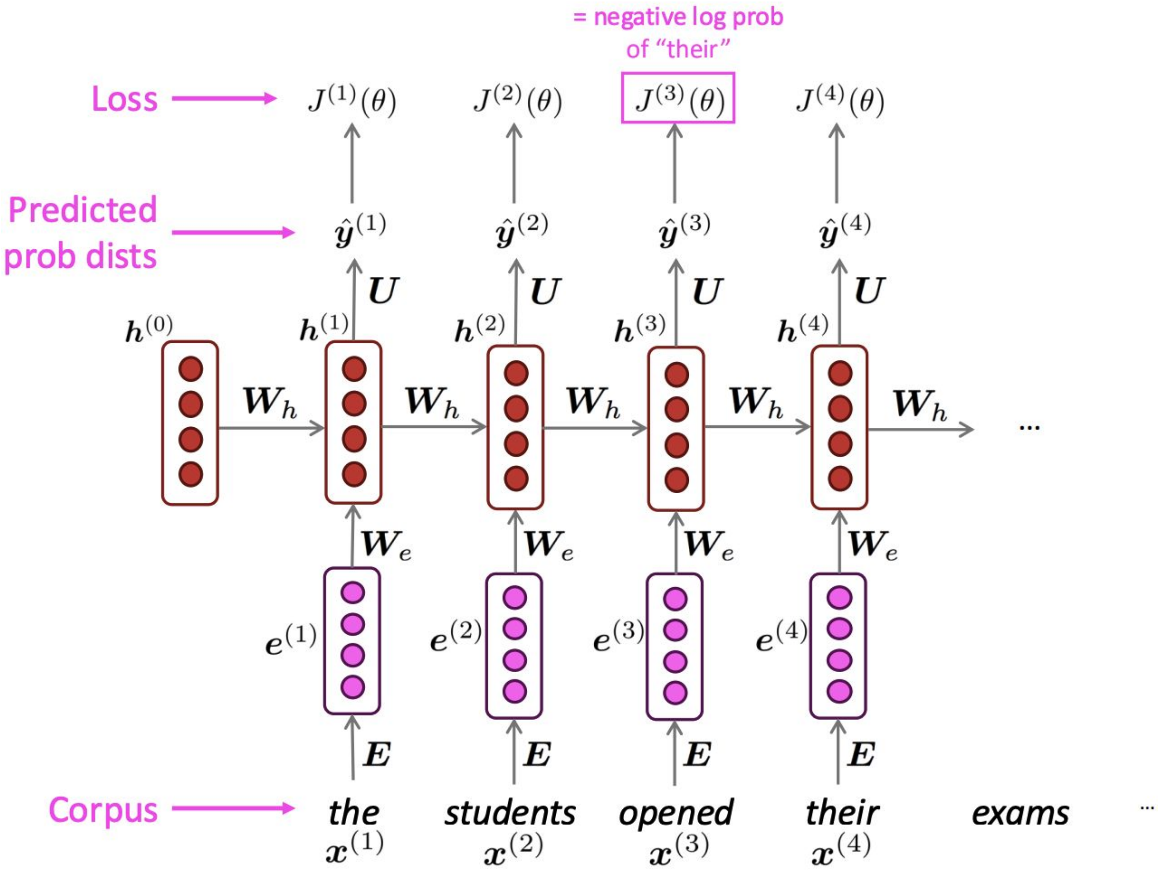
# Training an RNN language model



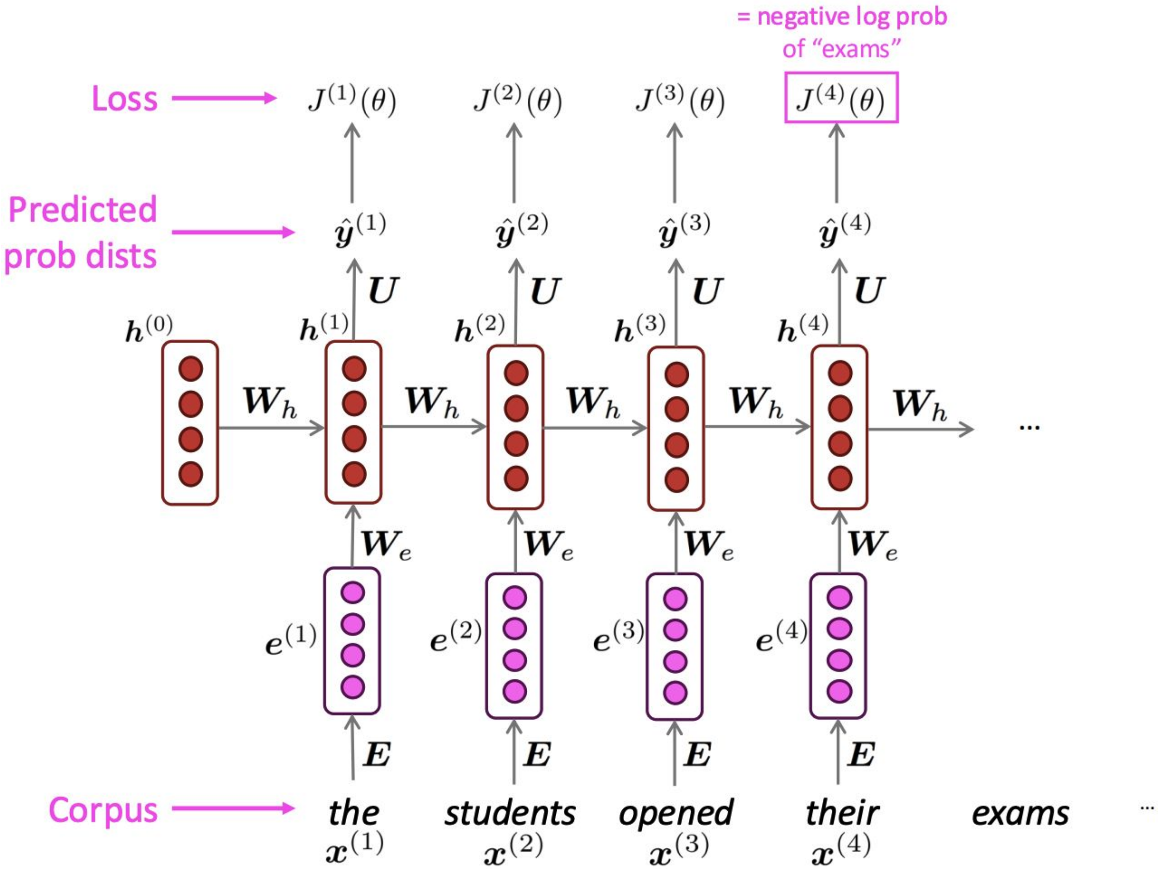
# Training an RNN language model



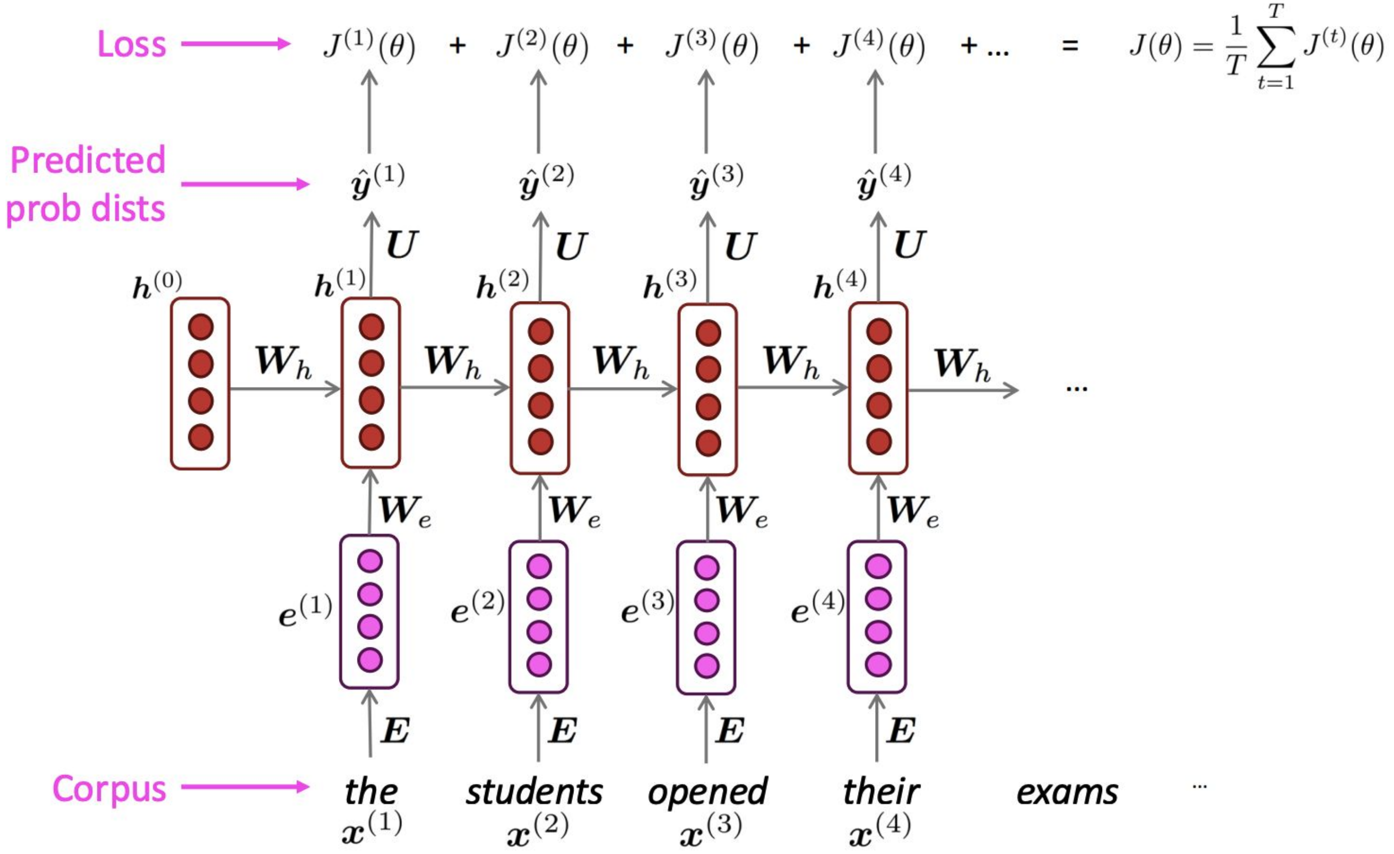
# Training an RNN language model



# Training an RNN language model

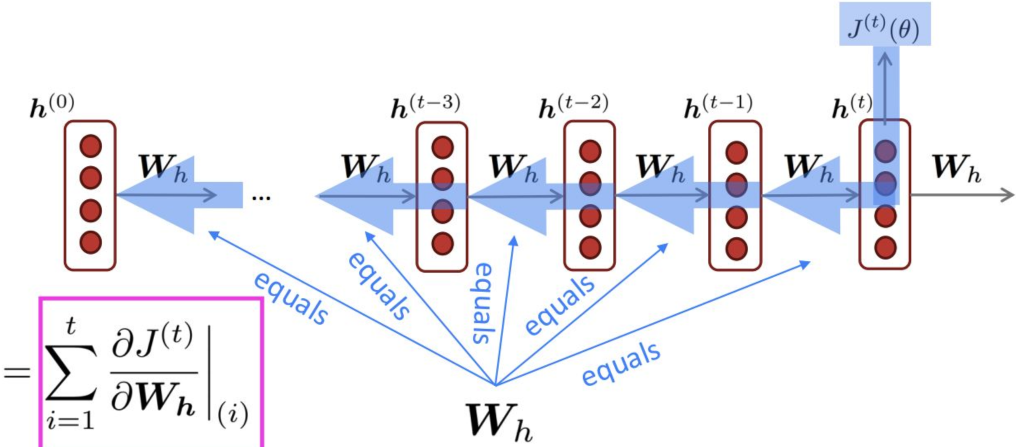


# Training an RNN language model



# Training the parameters of RNNs

- **Backpropagation through time**




$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \boxed{\sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\bigg|_{(i)}}$$

**Question:** How do we calculate this?

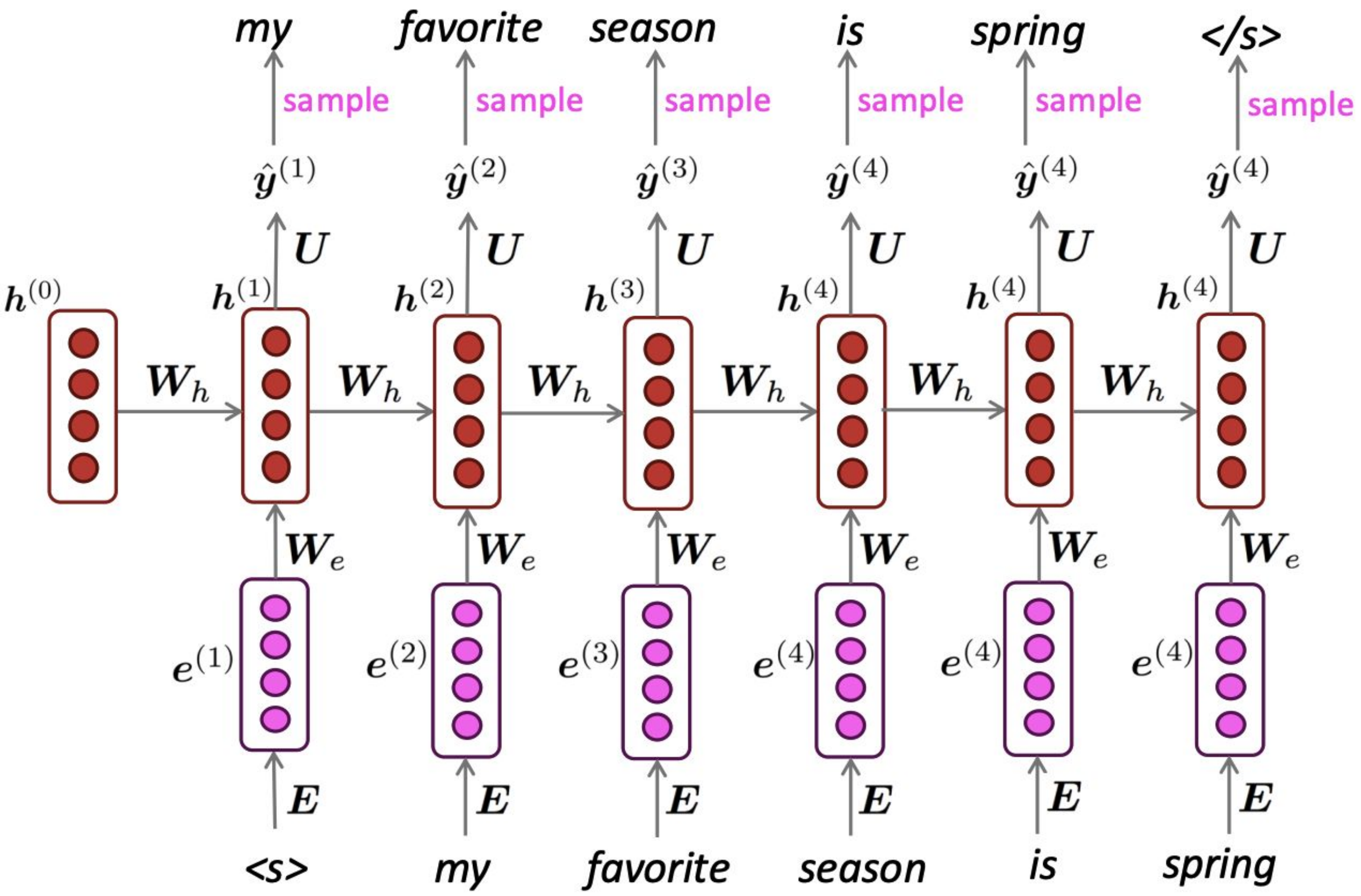
**Answer:** Backpropagate over timesteps $i = t, \ldots, 0$, summing gradients as you go. This algorithm is called **"backpropagation through time"** [Werbos, P.G., 1988, *Neural Networks* **1**, and others]

Apply the multivariable chain rule:

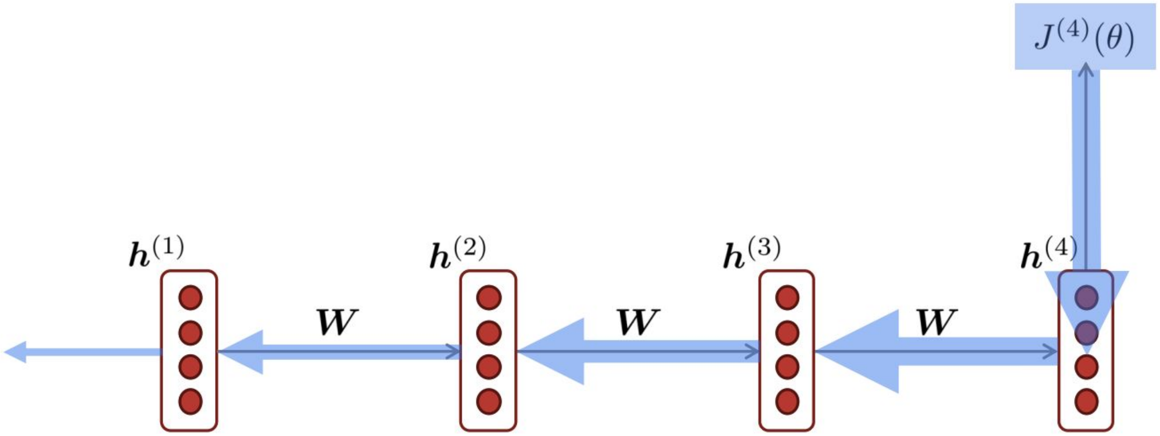
$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\bigg|_{(i)} \underbrace{\boxed{\frac{\partial \boldsymbol{W}_h|_{(i)}}{\partial \boldsymbol{W}_h}}}_{=1}$$

$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\bigg|_{(i)}$$

# Generating with an RNN language model



# Vanishing gradients in RNNs

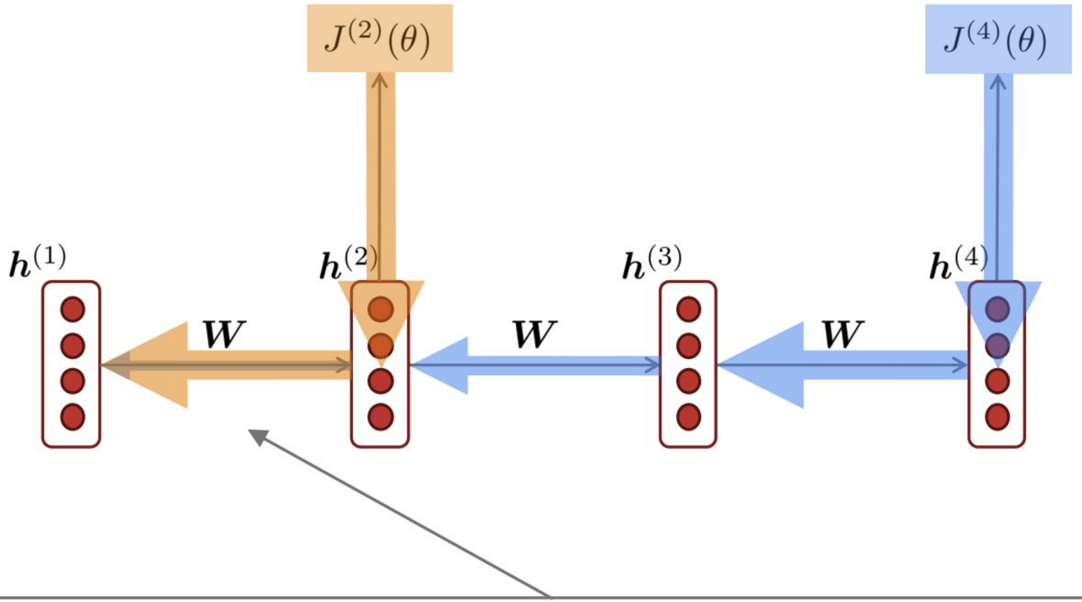



$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \boxed{\frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

What happens if these are small?

**Vanishing gradient problem:**
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

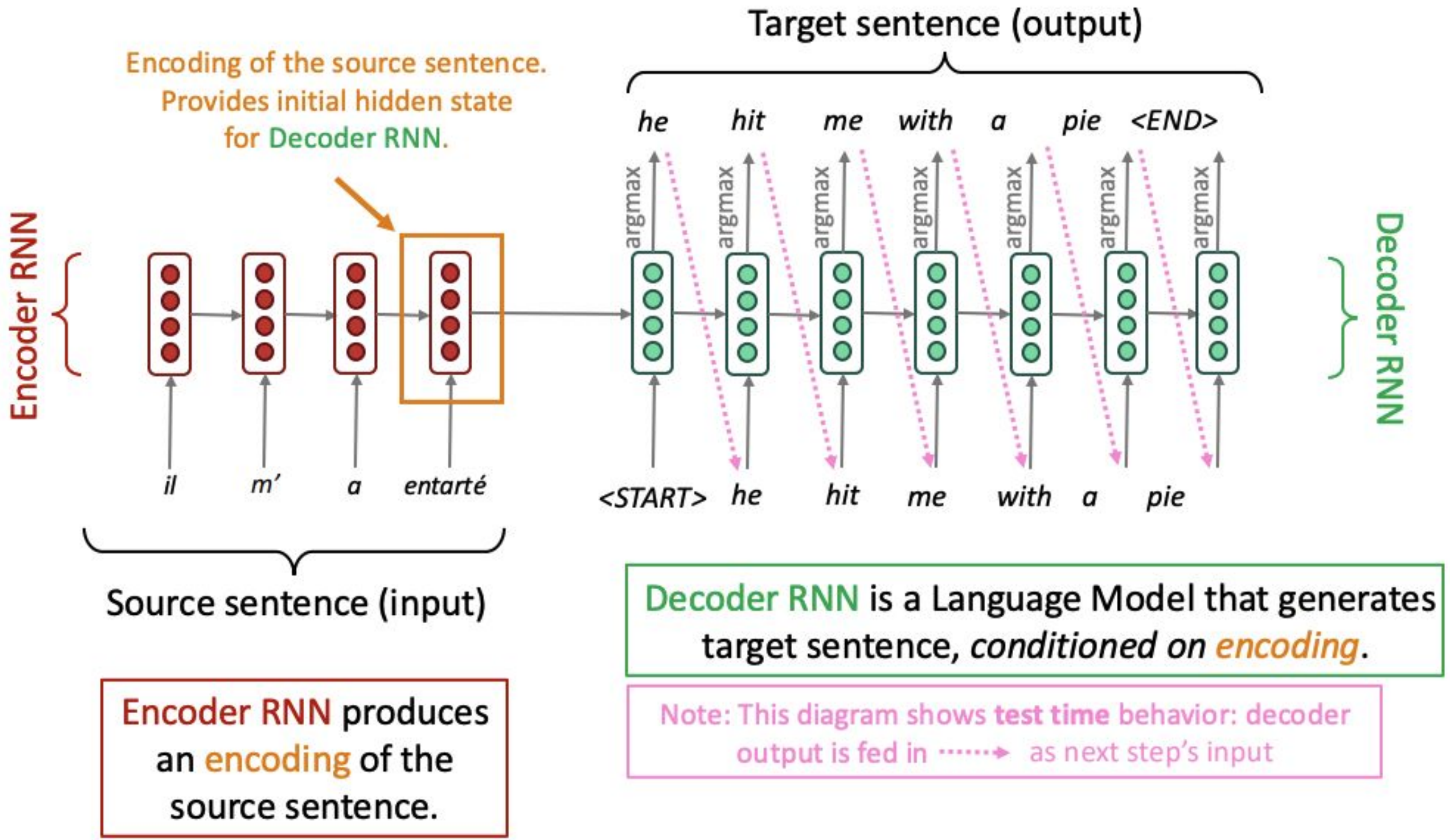
# Why is vanishing gradient a problem?




Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are updated only with respect to near effects, not long-term effects.

# Neural machine translation



# Neural machine translation was an early big success of Neural NLP

BEST DIGITAL NEWS PLATFORM

https://kiswahili.tuko.co.ke/

**Malawi yawapoteza mawaziri 2 kutokana na maafa ya COVID-19**

TUKO.co.ke imefahamishwa kuwa waziri wa serikali ya mitaa Lingson Belekanyama na mwenzake wa uchukuzi Sidik Mia walifariki dunia ndani ya saa mbili tofauti.

Swahili English

Google Translate

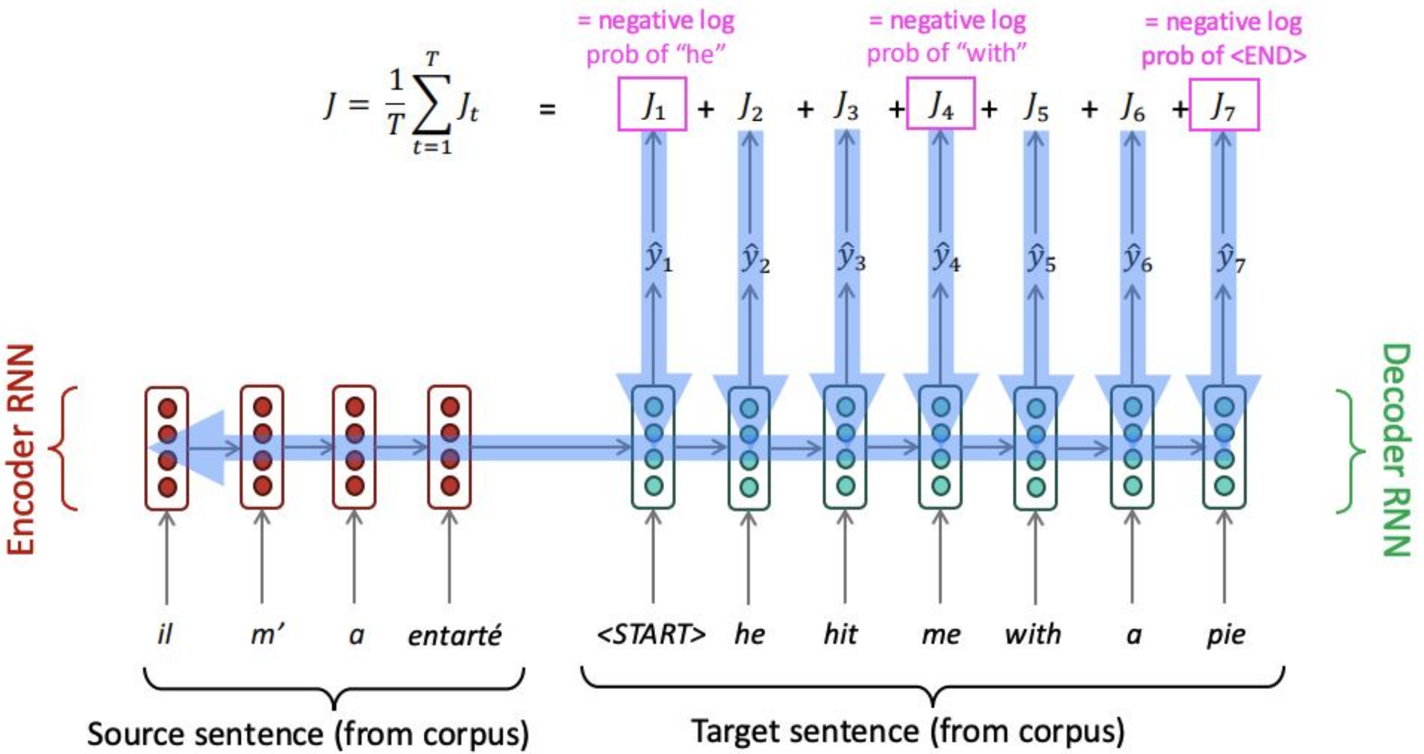
**Malawi loses 2 ministers due to COVID-19 disaster**

TUKO.co.ke has been informed that local government minister Lingson Belekanyama and his transport counterpart Sidik Mia died within two separate hours.

# Sequence to sequence modeling

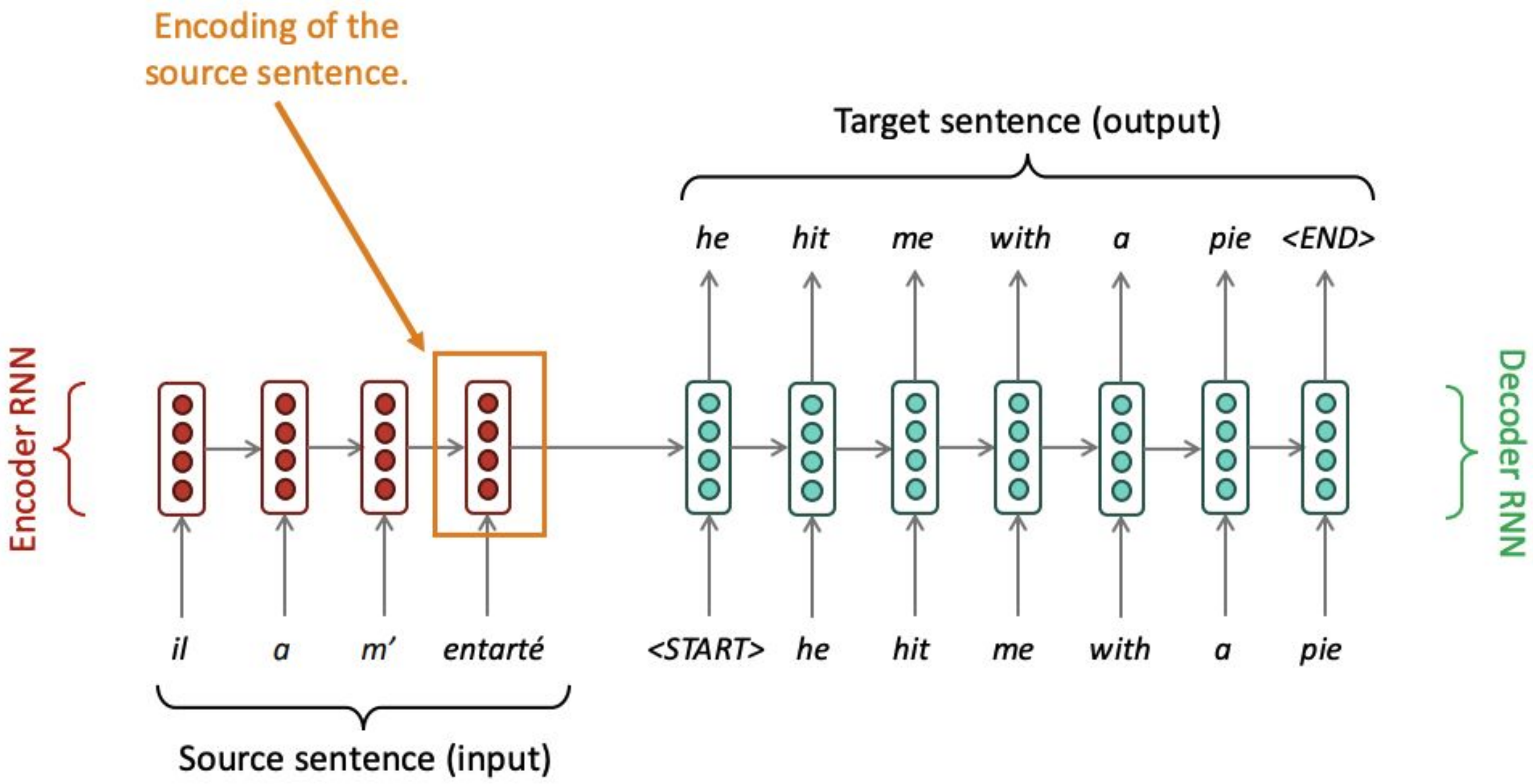
- **The general notion here is an encoder-decoder model**
  - One neural network takes input and produces a neural representation
  - Another network produces output based on that neural representation
  - Many NLP tasks can be phrased as sequence-to-sequence:
    - Summarization
    - Dialogue
    - Code generation

# Training a sequence to sequence model




Seq2seq is optimized as a **single system.** Backpropagation operates "*end-to-end*".

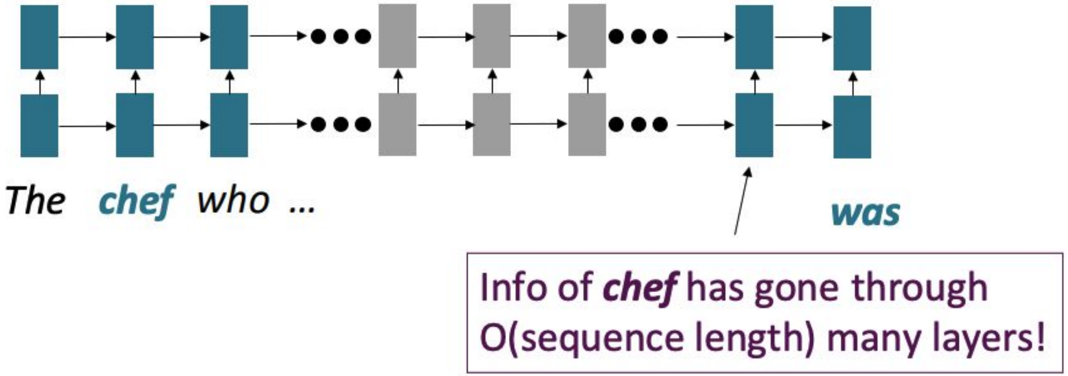
# The bottleneck problem in RNNs




Problems with this architecture?

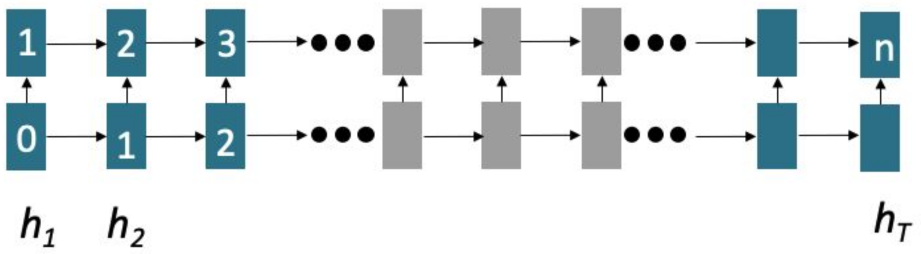
# Linear interaction distance

- **O(sequence length) steps for distant word pairs to interact means:**
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is "baked in"; we already know linear order isn't the right way to think about sentences…



# Issue: Lack of parallelizability

- **Forward and backward passes have O(sequence length) unparallelizable operations**
  - GPUs can perform a bunch of independent operations at once!
  - **BUT!** future RNN hidden states can't be computed in full before past RNN hidden states have been computed



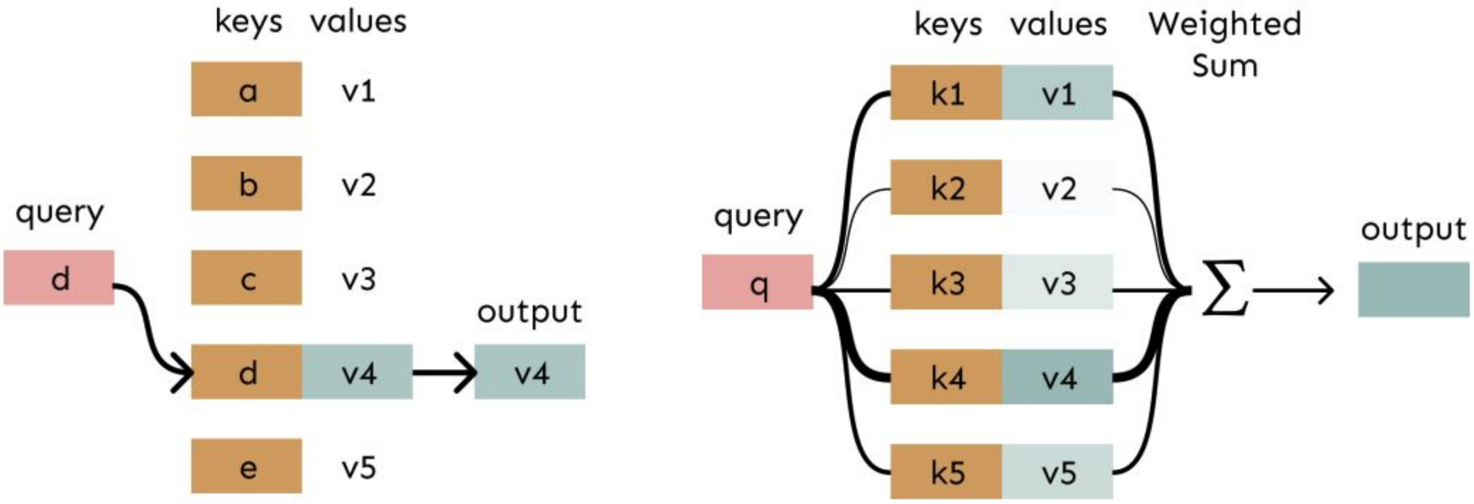

Numbers indicate min # of steps before a state can be computed

# Attention

- Attention provides a solution to the bottleneck problem!
- Core idea: on each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence!

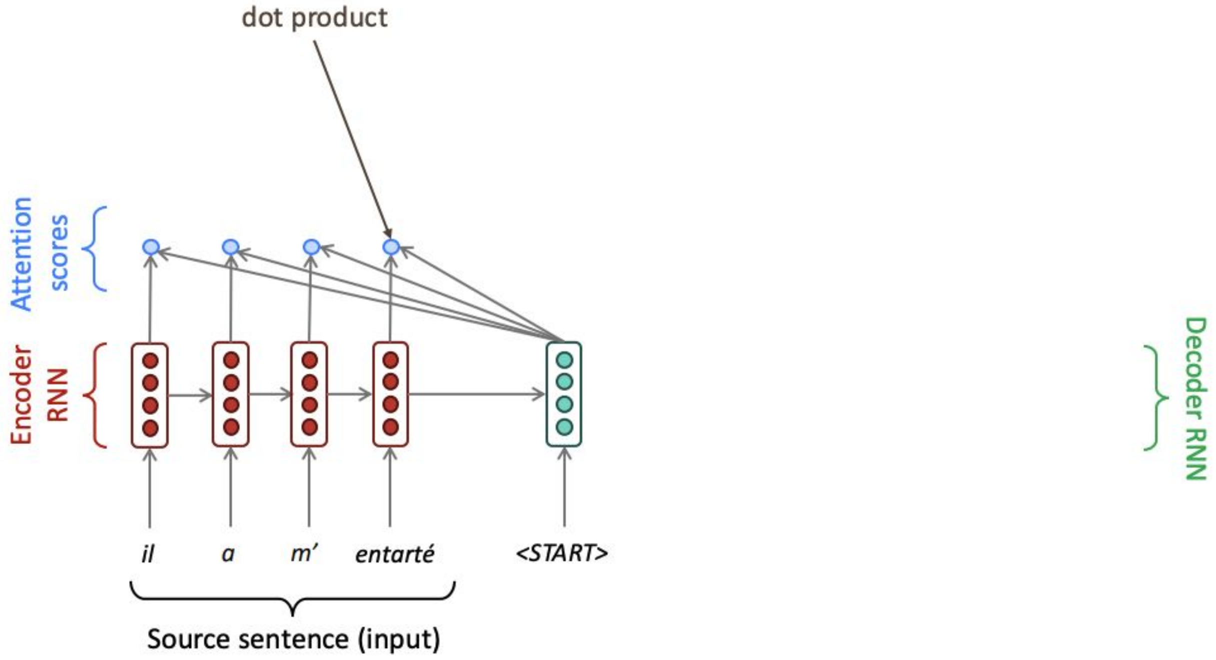
# Attention is weighted averaging

- In attention, the query matches all keys softly, to a weight between 0 and 1. The key's values are multiplied by the weights and summed!

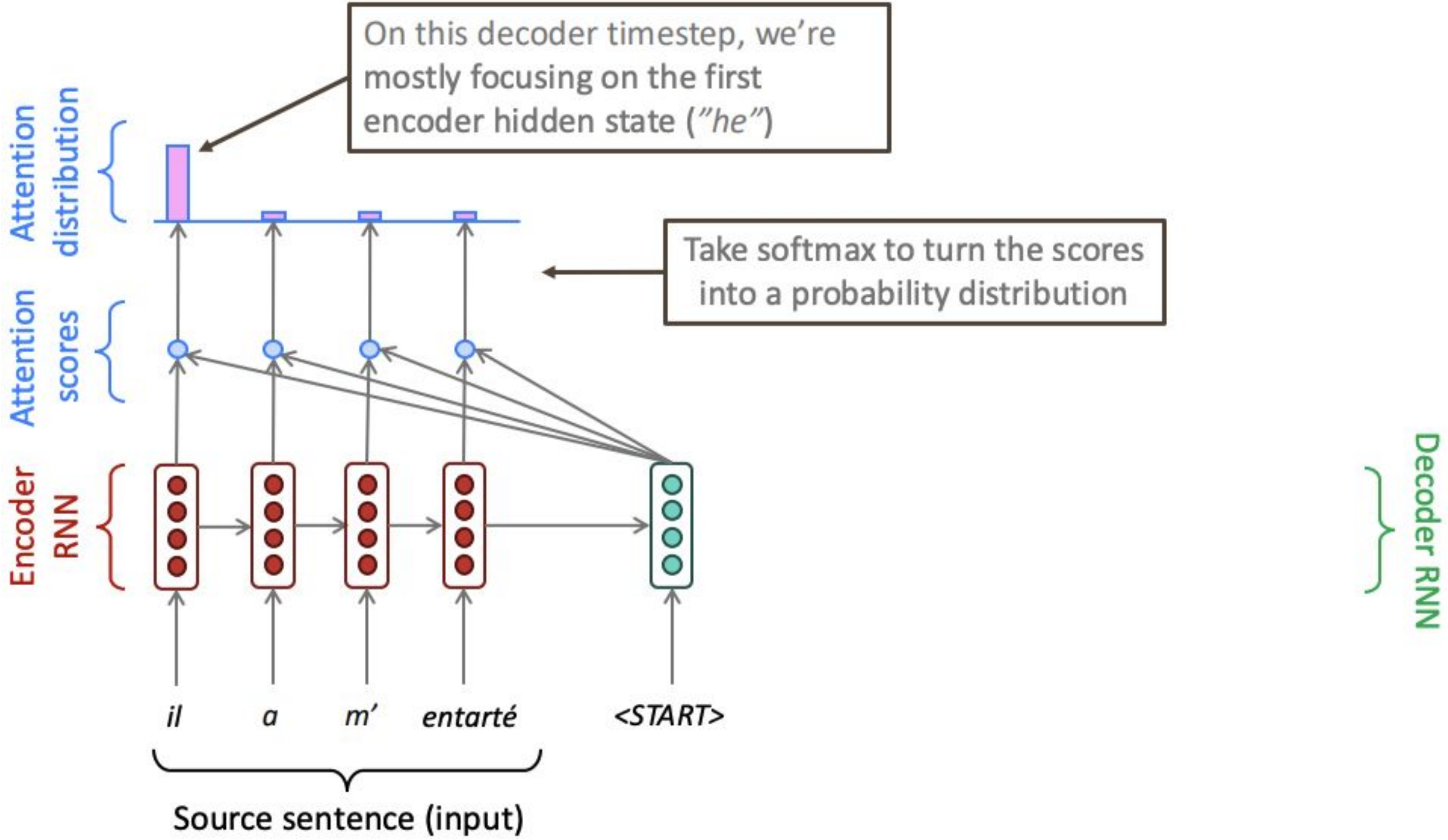


# RNNs with attention

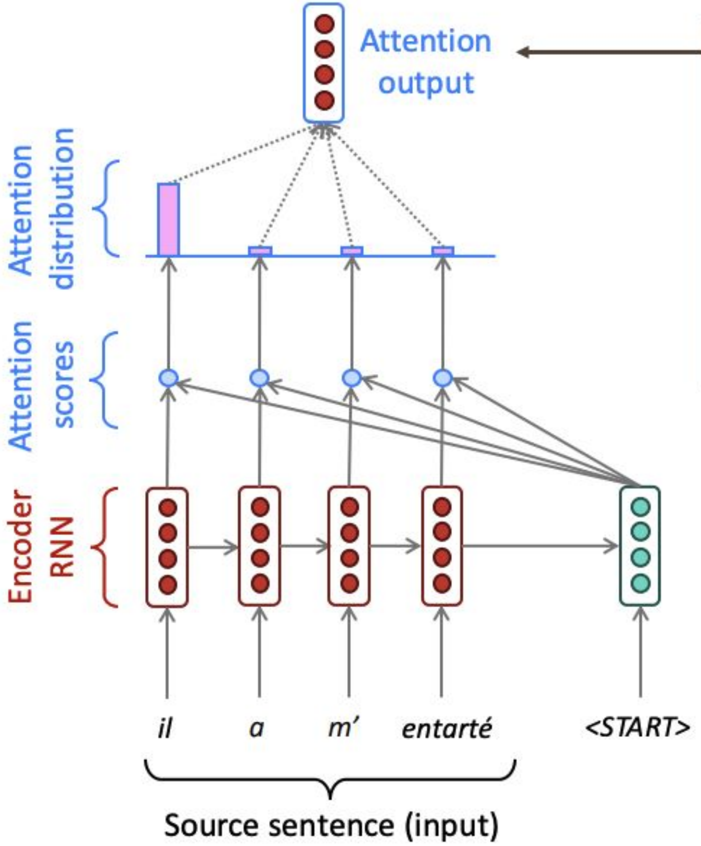
- On each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence.



# RNN with attention



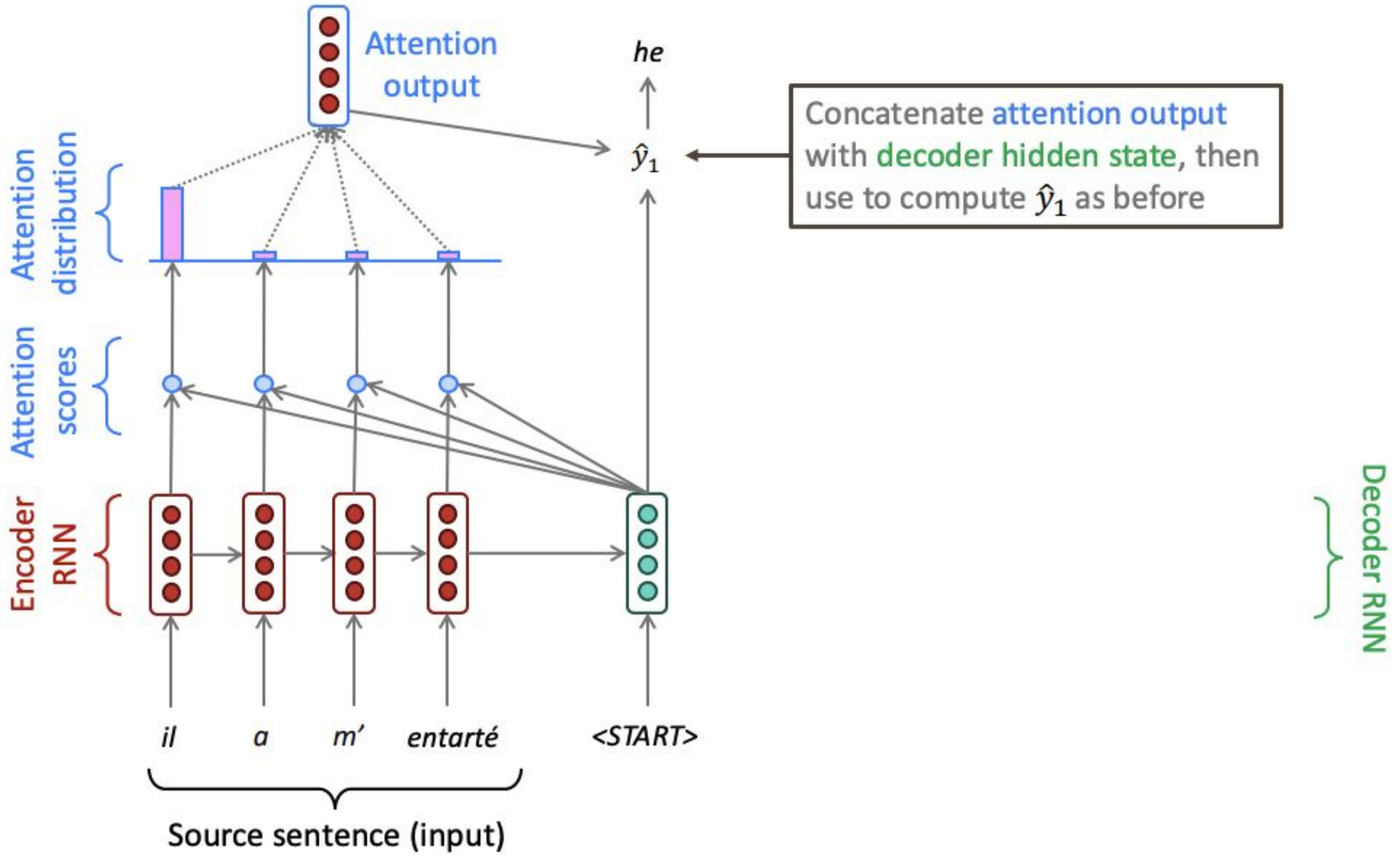
# RNN with attention



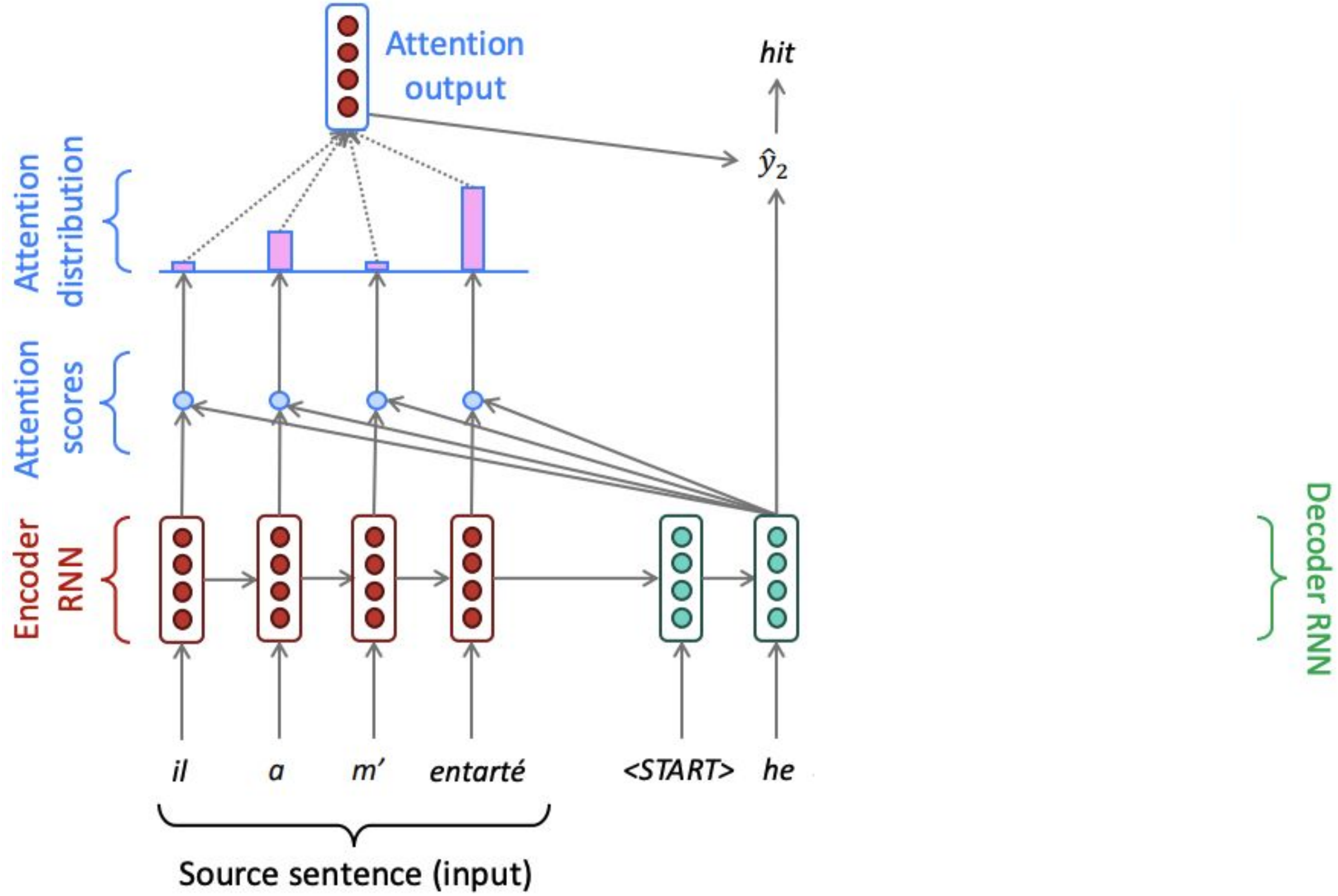

Use the attention distribution to take a **weighted sum** of the encoder hidden states.

The attention output mostly contains information from the hidden states that received high attention.

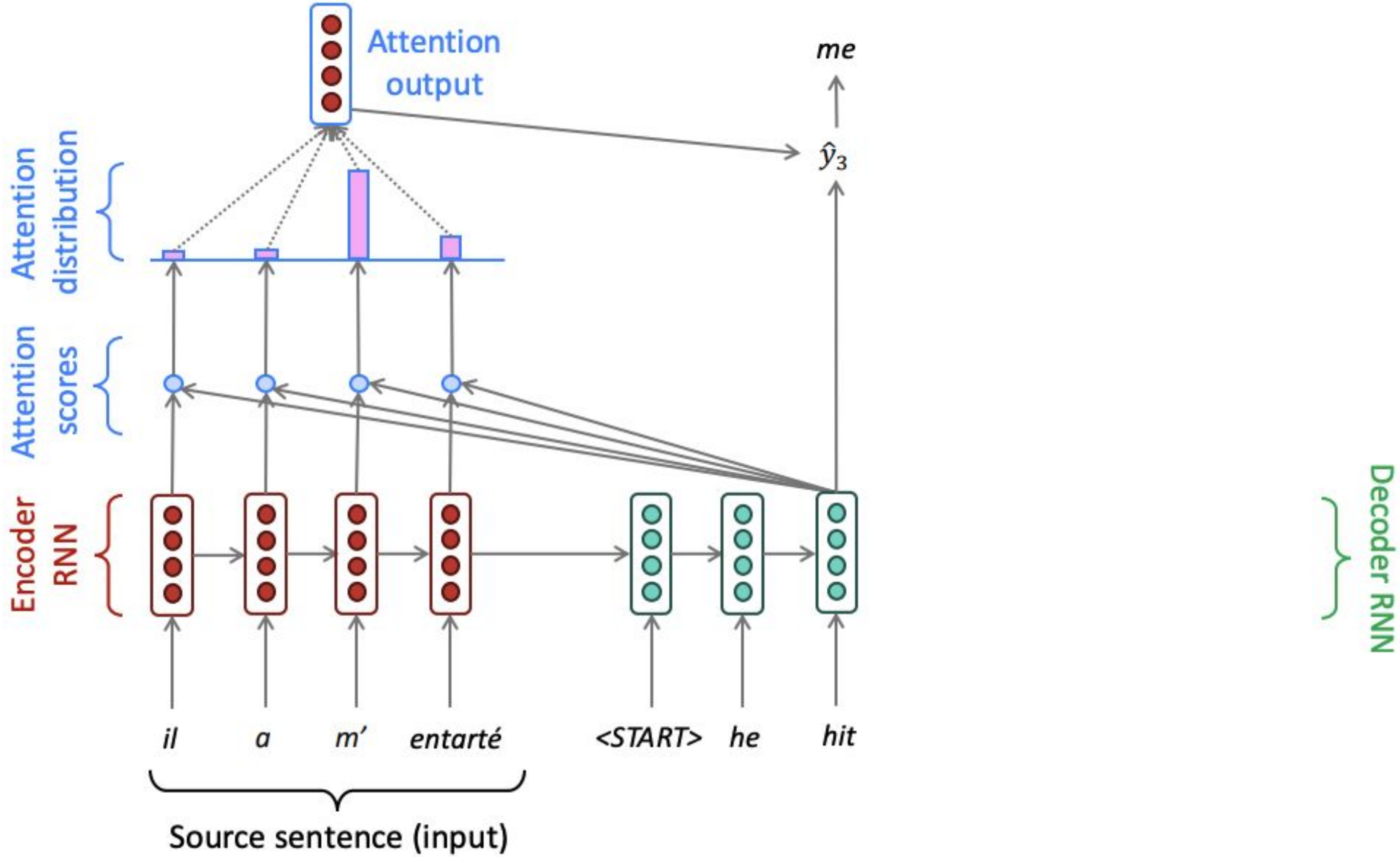
# RNN with attention



# RNN with attention



# RNN with attention



# Do we need recurrence at all?

- **Abstractly: Attention is a way to pass information from a sequence ($x$) to a neural network input. ($h_t$)**
  - This is also exactly what RNNs are used for – to pass information!
  - Can we just get rid of the RNN entirely? Maybe attention is just a better way to pass information!
  - The building block we need is **Self Attention!**
  - **So far we saw cross-attention!**

# Self-attention: Keys, Queries, Values!

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary $V$, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$ , let $\boldsymbol{x}_i = E\boldsymbol{w}_{\boldsymbol{i}}$, where $E \in \mathbb{R}^{d\times|V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices Q, K, V , each in $\mathbb{R}^{d\times d}$

$\boldsymbol{q}_i = Q\boldsymbol{x}_{\boldsymbol{i}}$ (queries) $\boldsymbol{k}_i = K\boldsymbol{x}_{\boldsymbol{i}}$ (keys) $\boldsymbol{v}_i = V\boldsymbol{x}_{\boldsymbol{i}}$ (values)

2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \boldsymbol{q}_{\boldsymbol{i}}^{\top}\boldsymbol{k}_{\boldsymbol{j}} \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'}\exp(\boldsymbol{e}_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_{\boldsymbol{j}} \boldsymbol{\alpha}_{ij}\,\boldsymbol{v}_i$$

# But there is no inherent order in SA!

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values
- Consider representing each sequence index as a vector

$\boldsymbol{p}_i \in \mathbb{R}^d$, for $i \in \{1, 2, \ldots, n\}$ are position vectors

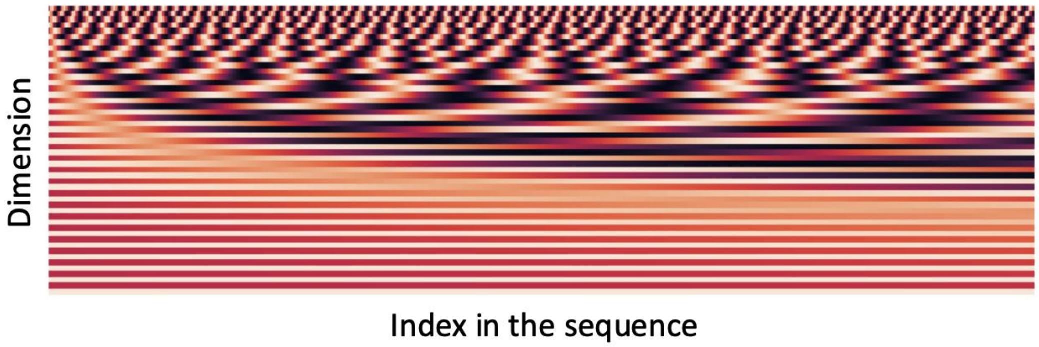
$$\widetilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add...

# Sinusoidal position representation to add order

- **Sinusoidal position representations: concatenate sinusoidal functions of varying periods**
  - Periodicity indicates that maybe "absolute position" isn't as important
  - can extrapolate to longer sequences as periods restart!

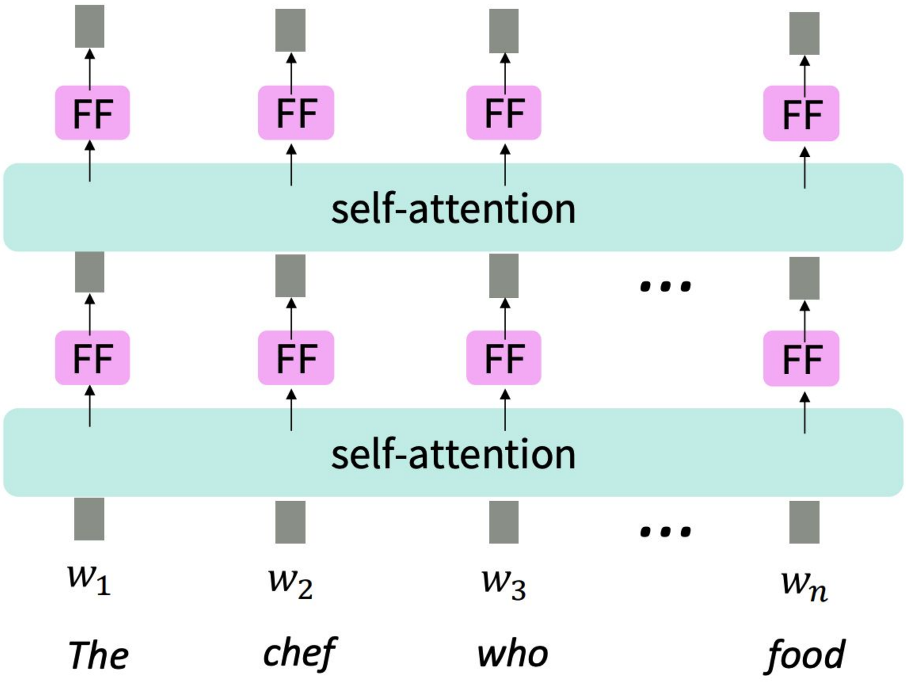
$$\boldsymbol{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



# But there are not non-linearities in SA!

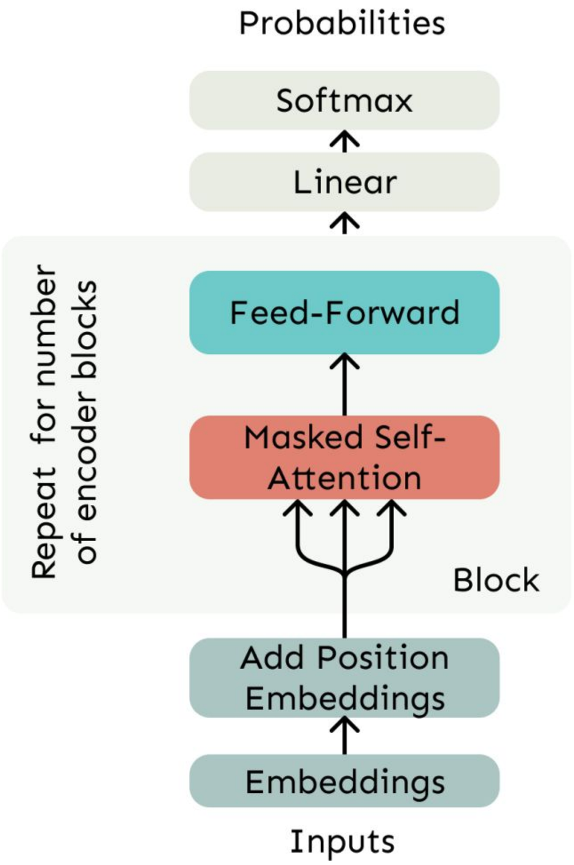
- Easy fix: add a **feed-forward network** to post-process each output vector.

$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1 \,\text{output}_i + b_1) + b_2$$



# Put everything together

- **Position representation:**
  - Specify the sequence order, since self-attentiis an unordered function of its inputs.
- **Nonlinearities**
  - Frequently implemented as a simple feedforward network.



# More to come next session!

- Encoder-decoder transformer!