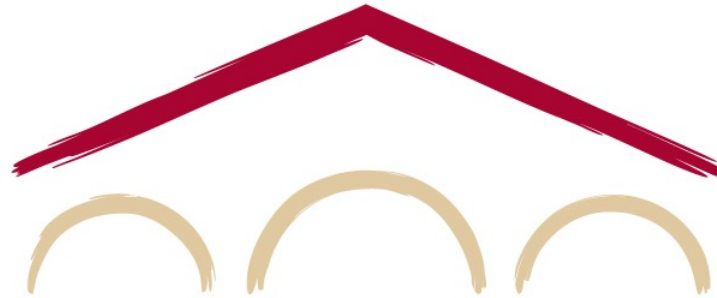


Natural Language Processing with Deep Learning

CS224N/Ling284



Tatsunori Hashimoto

Lecture 8: Self-Attention and Transformers

Lecture Plan

1. From recurrence (RNN) to attention-based NLP models
2. The Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers

Reminders:

See the [2023 lecture notes](#) for some bonus material

Assignment 4 due a week from today! Use Colab for the final training if you don't have a GPU.

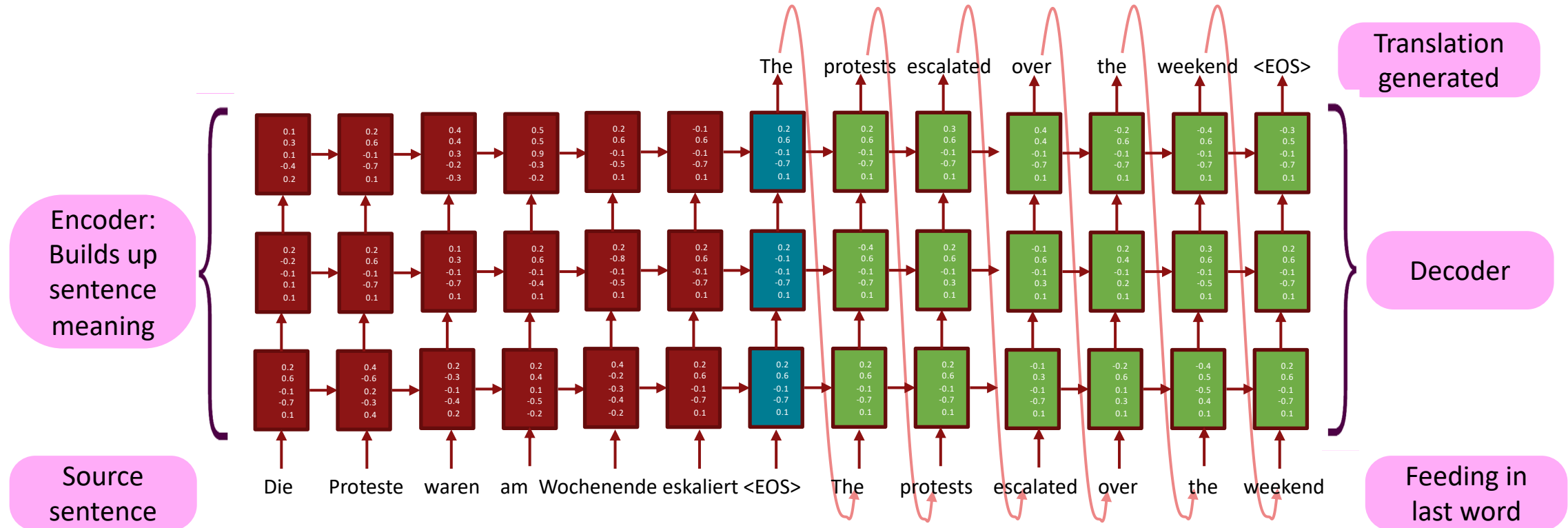
Final project proposal out tonight, due Tuesday, Feb 14 at 4:30PM PST!

Please try to hand in the project proposal on time; we want to get you feedback quickly!

Last last lecture: Multi-layer RNN for machine translation

[Sutskever et al. 2014; Luong et al. 2015]

The hidden states from RNN layer i are the inputs to RNN layer $i + 1$



Conditioning =
Bottleneck

NMT: the first big success story of NLP Deep Learning

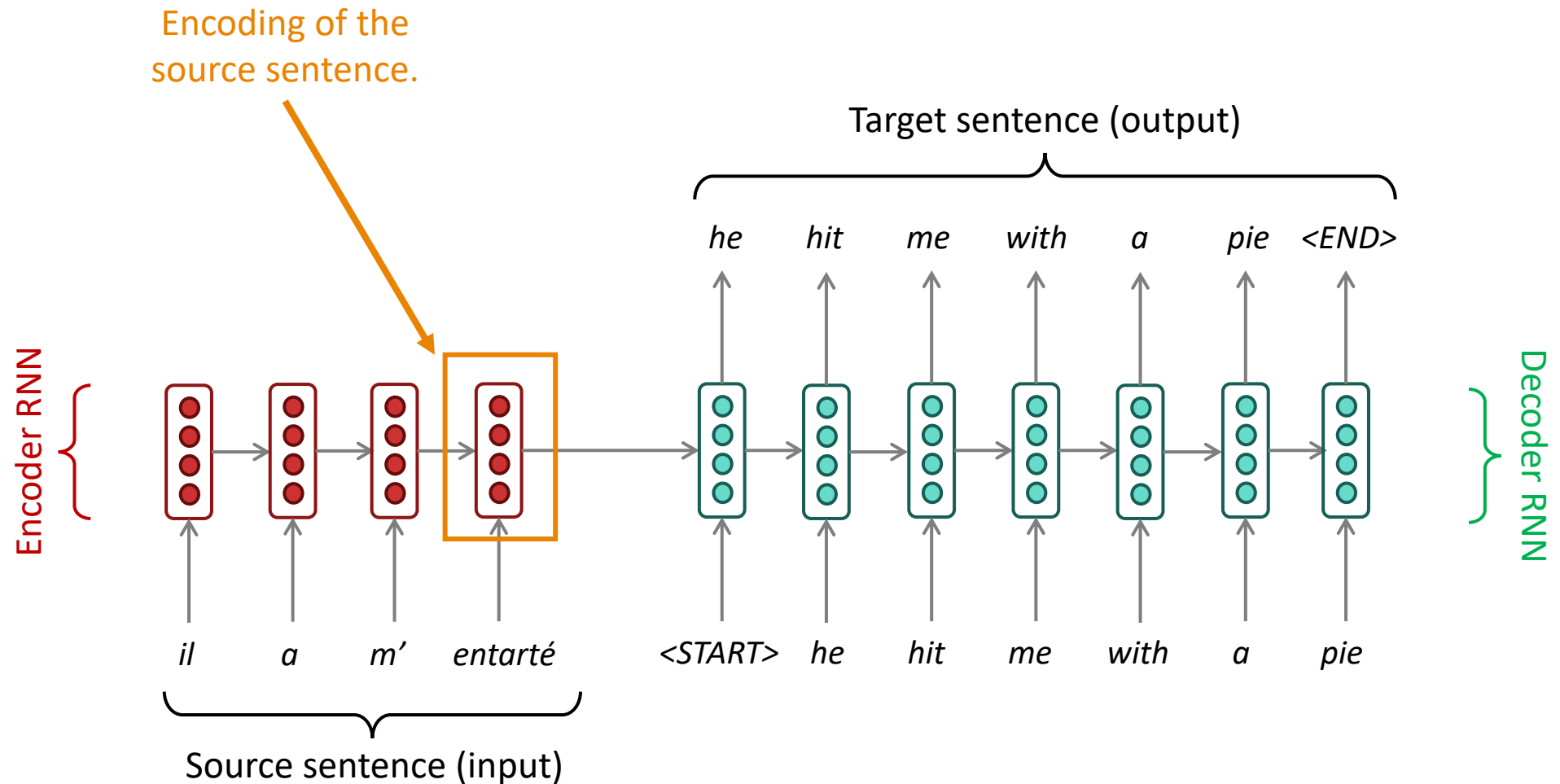
Neural Machine Translation went from a **fringe research attempt** in **2014** to the **leading standard method** in **2016**

- **2014**: First seq2seq paper published [Sutskever et al. 2014]
- **2016**: Google Translate switches from SMT to NMT – and by 2018 everyone has



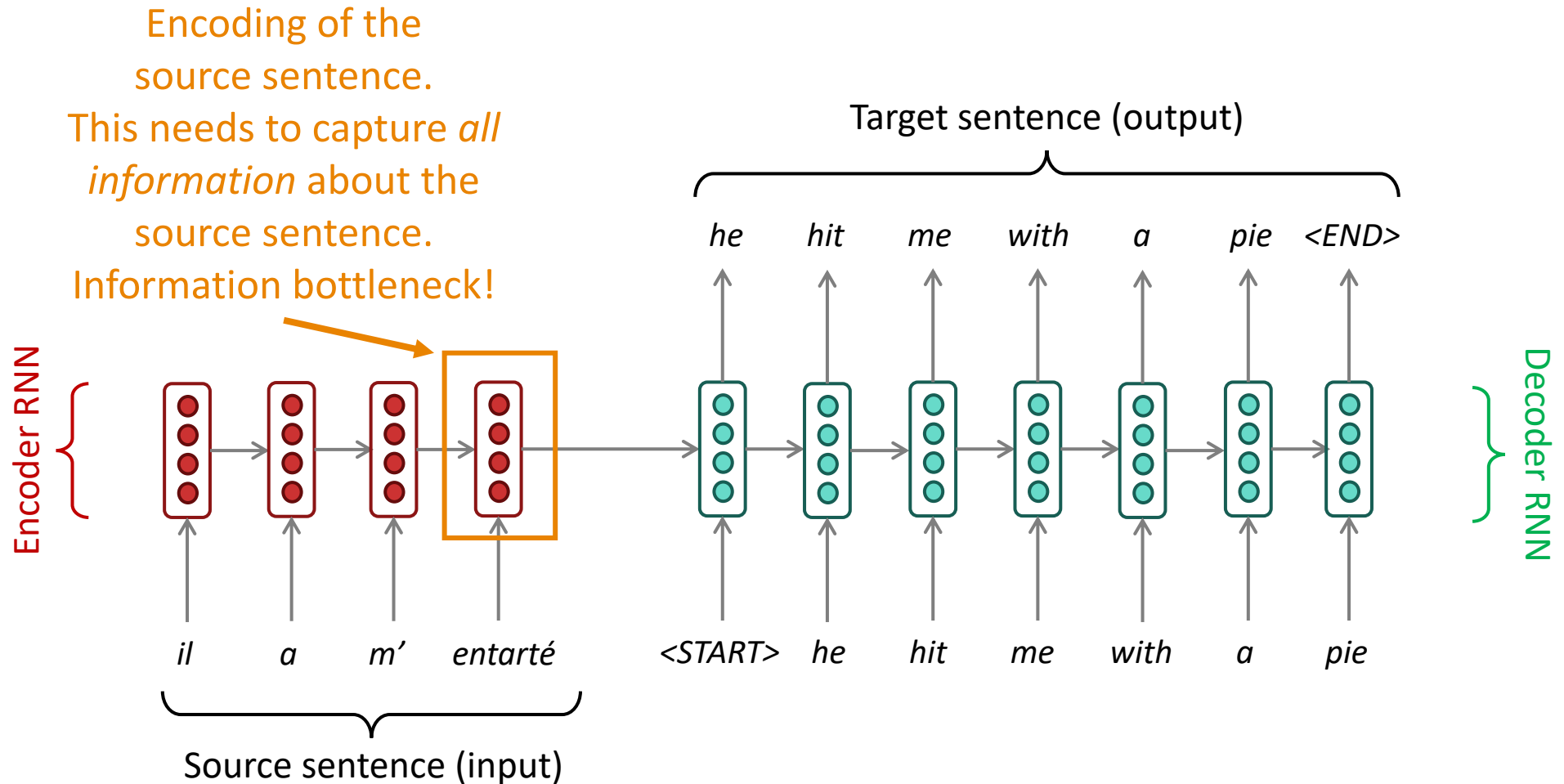
- **This is amazing!**
 - **SMT** systems, built by **hundreds** of engineers over many **years**, outperformed by NMT systems trained by **small groups** of engineers in a few **months**

The final piece: the bottleneck problem in RNNs



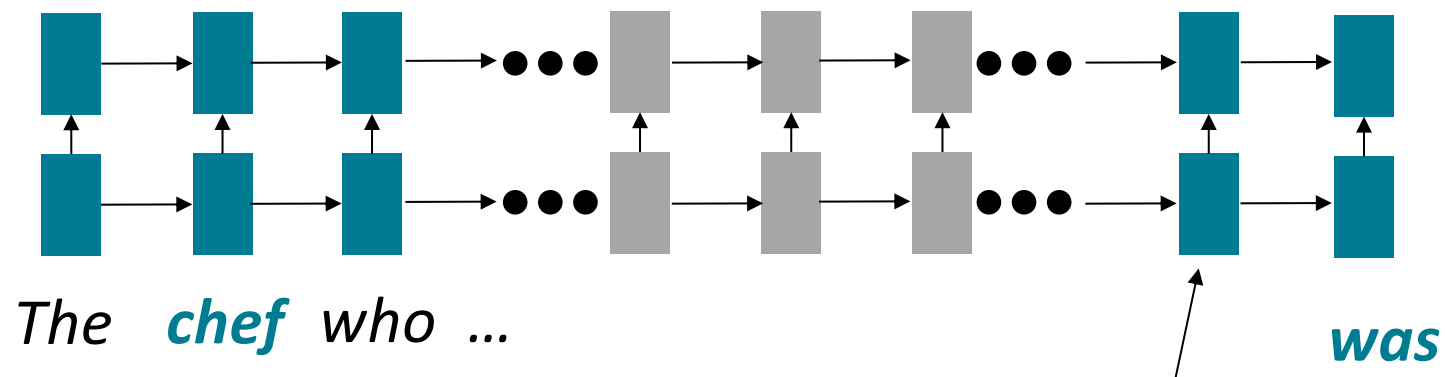
Problems with this architecture?

1. Why attention? Sequence-to-sequence: the bottleneck problem



Issues with recurrent models: Linear interaction distance

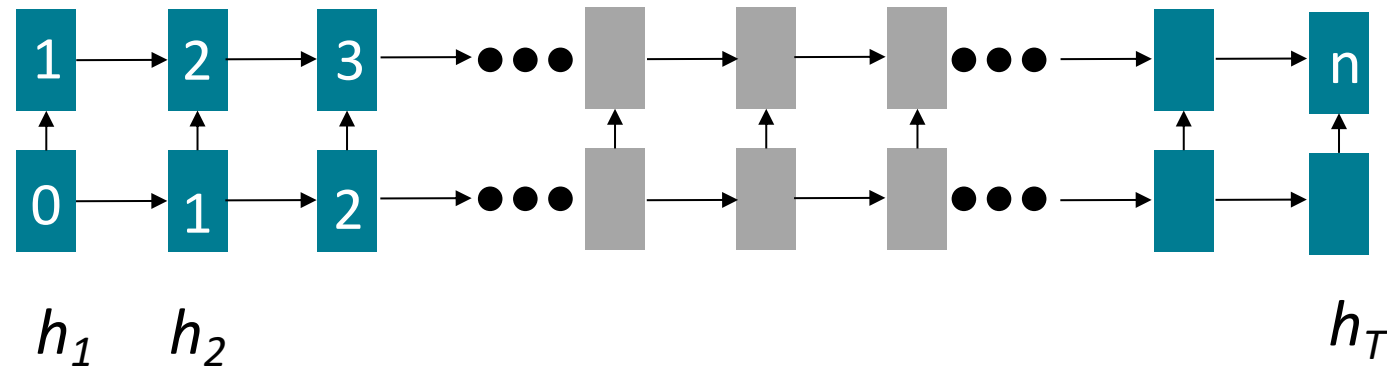
- **O(sequence length)** steps for distant word pairs to interact means:
 - Hard to learn long-distance dependencies (because gradient problems!)
 - Linear order of words is “baked in”; we already know linear order isn’t the right way to think about sentences...



Info of *chef* has gone through
O(sequence length) many layers!

Issues with recurrent models: Lack of parallelizability

- Forward and backward passes have **$O(\text{sequence length})$** unparallelizable operations
 - GPUs can perform a bunch of independent computations at once!
 - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
 - Inhibits training on very large datasets!



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

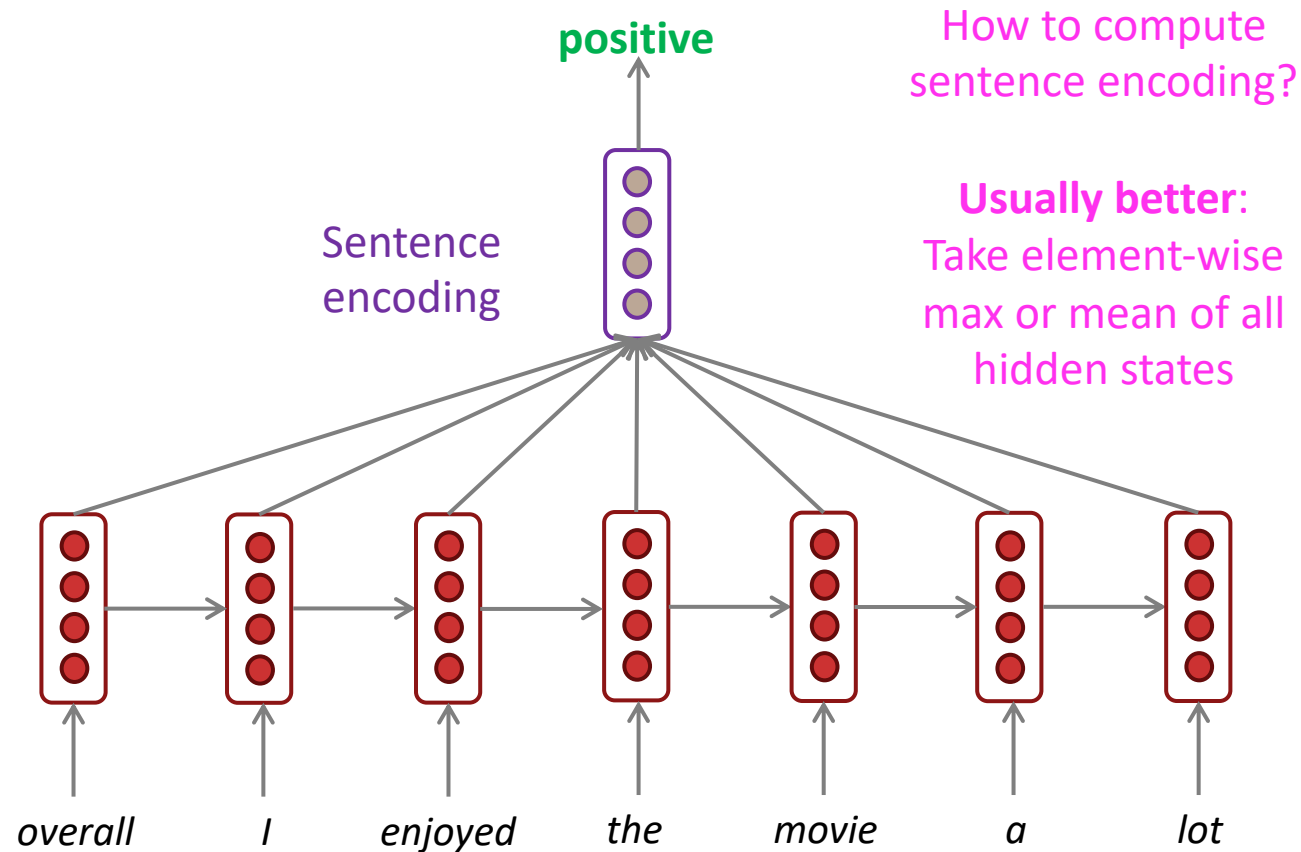
Attention

- **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



- First, we will show via diagram (no equations), then we will show with equations

The starting point: mean-pooling for RNNs

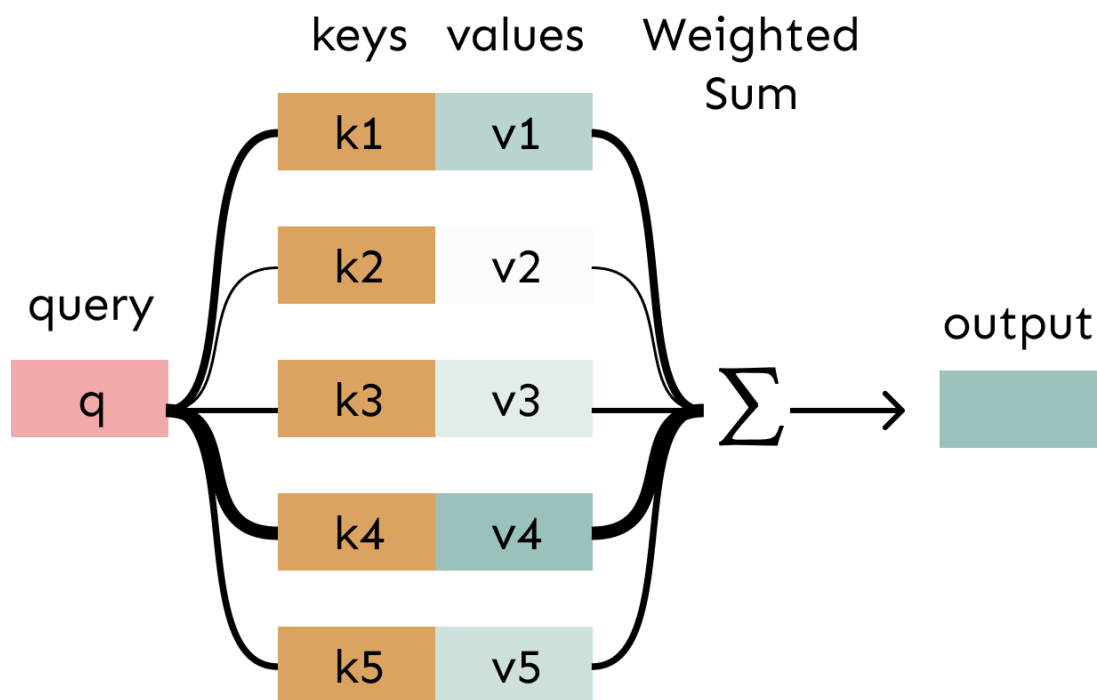


- Starting point: a *very* basic way of 'passing information from the encoder' is to *average*

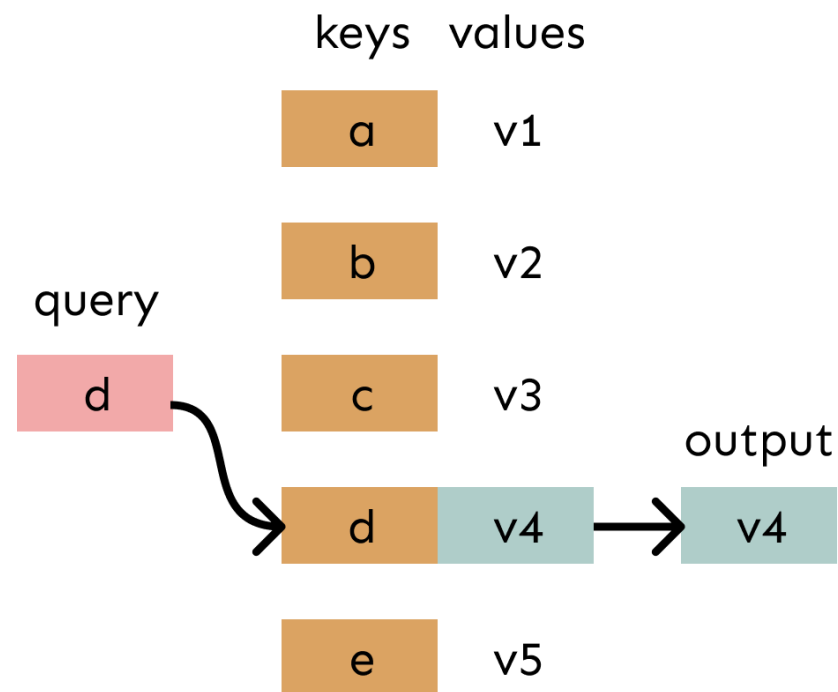
Attention is *weighted* averaging, which lets you do lookups!

Attention is just a **weighted** average – this is very powerful if the weights are learned!

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

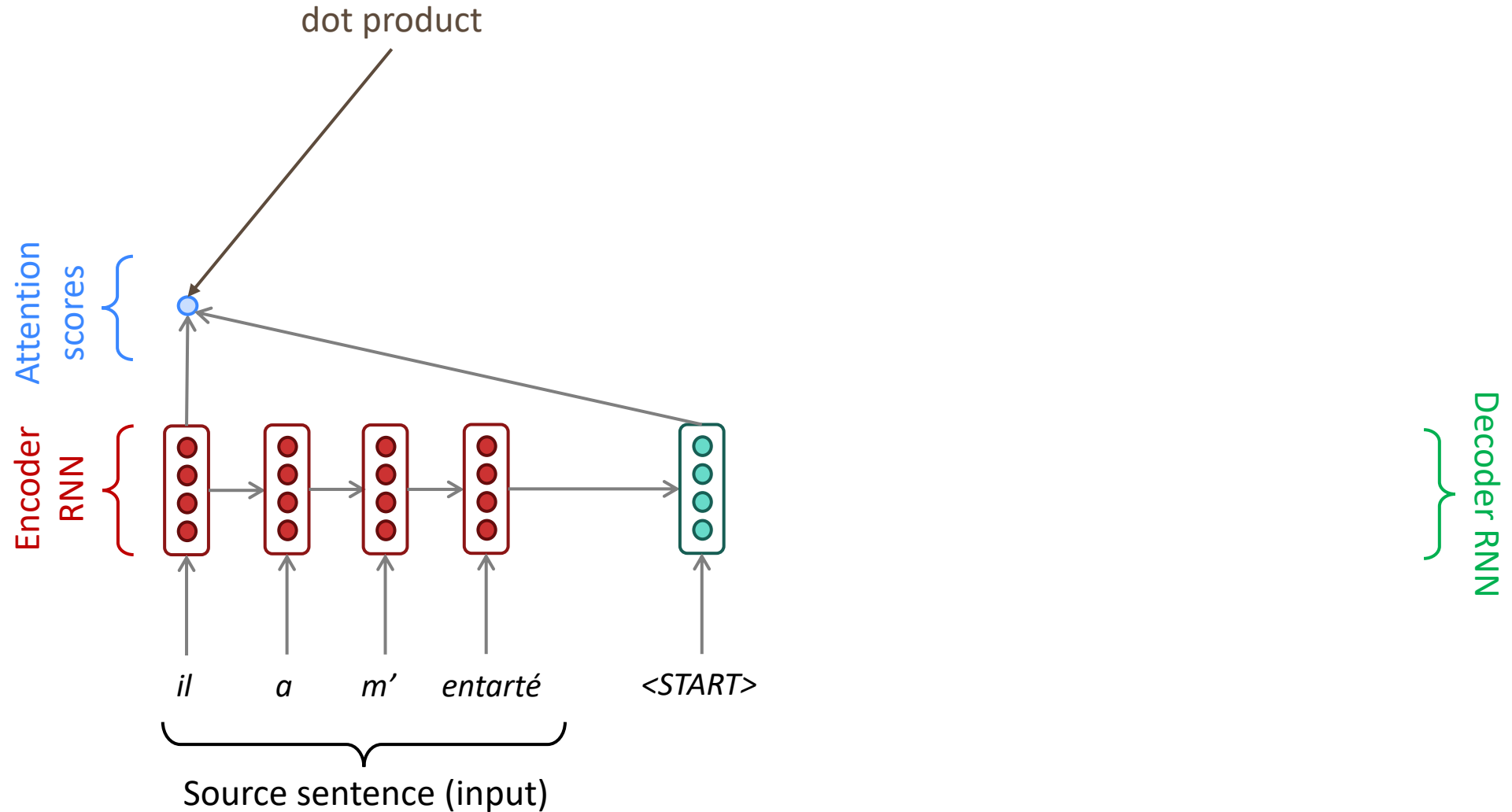


In a **lookup table**, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.

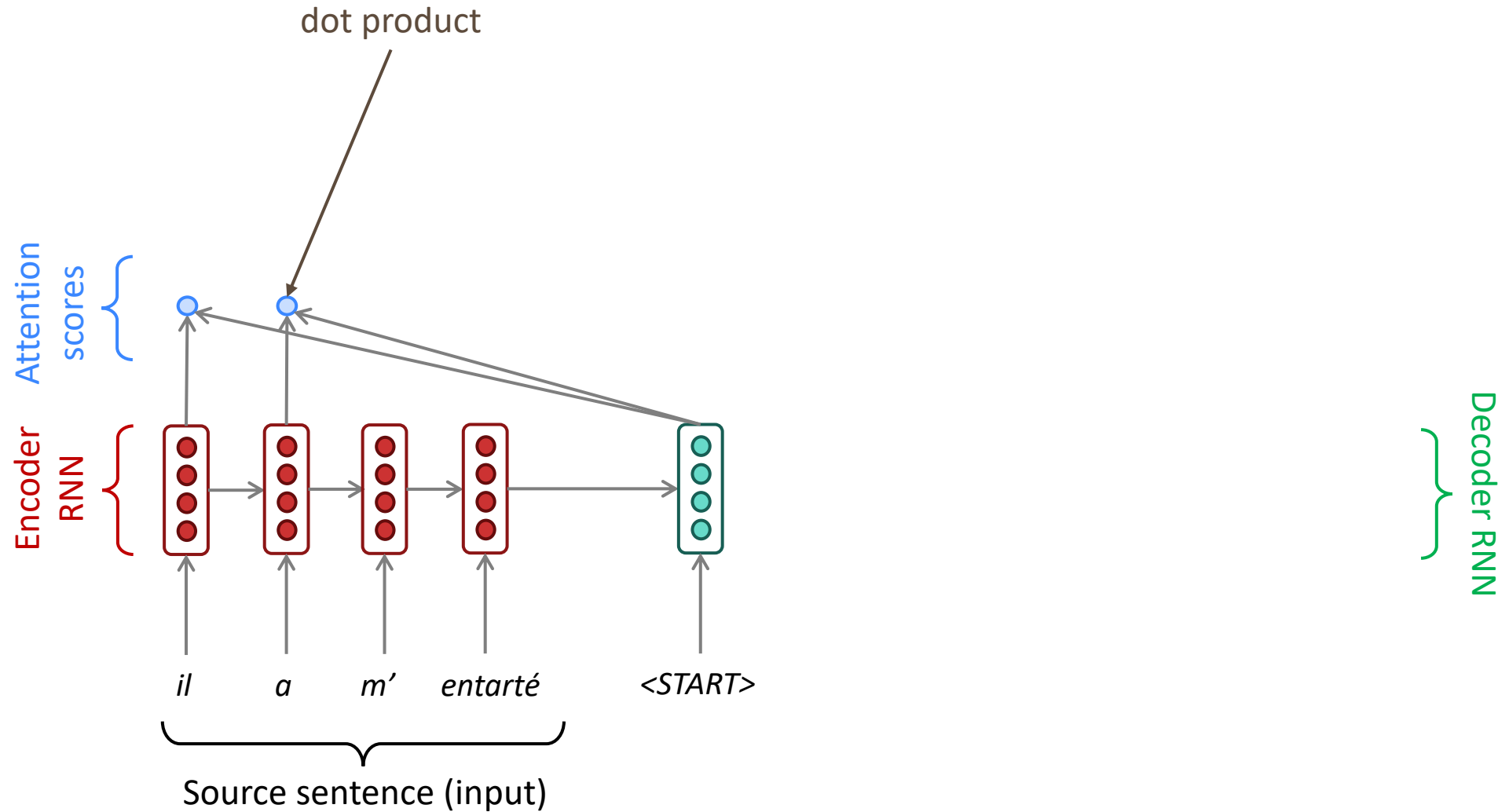


Sequence-to-sequence with attention

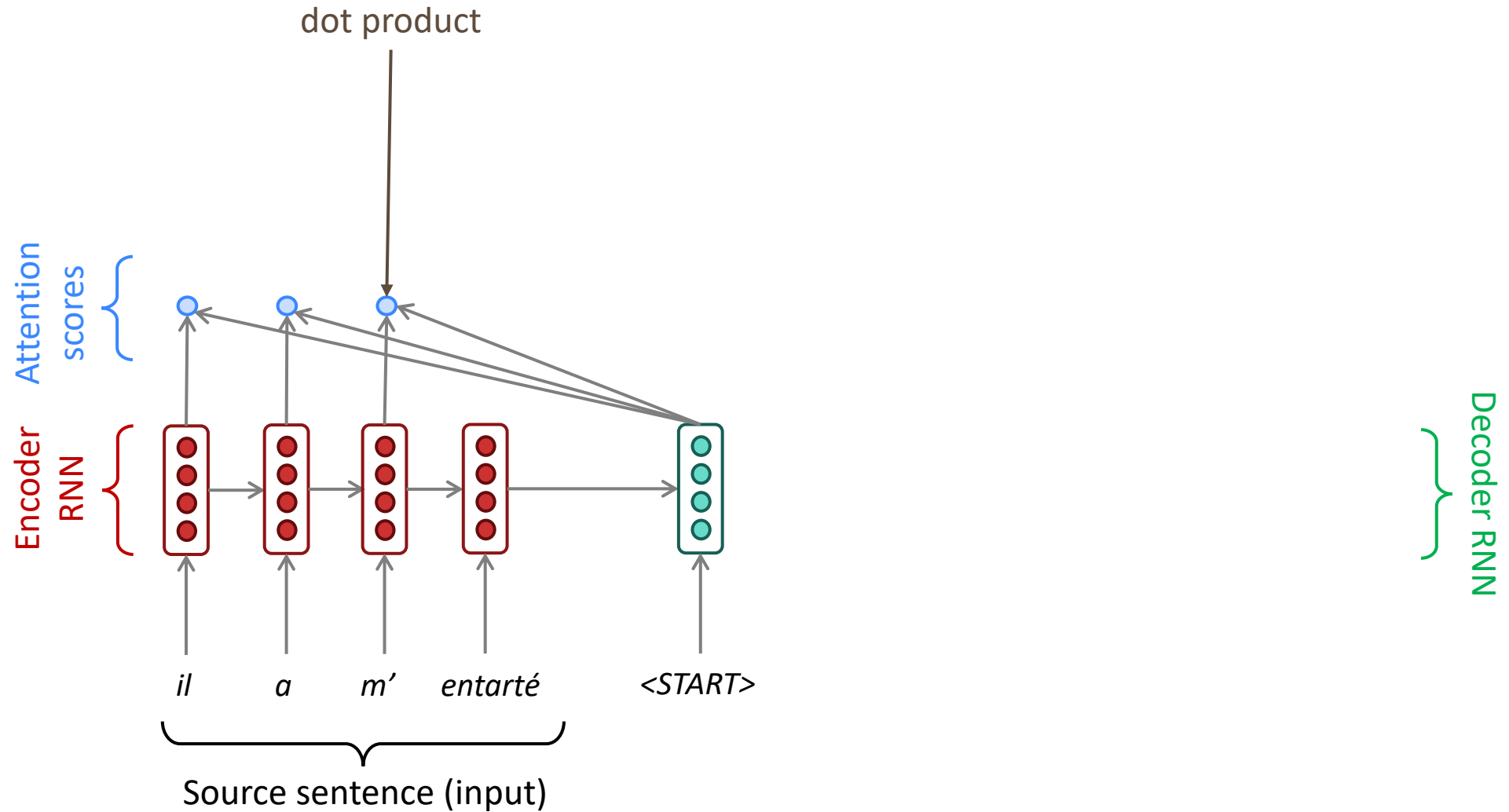
Core idea: on each step of the decoder, *use direct connection to the encoder* to *focus on a particular part* of the source sequence



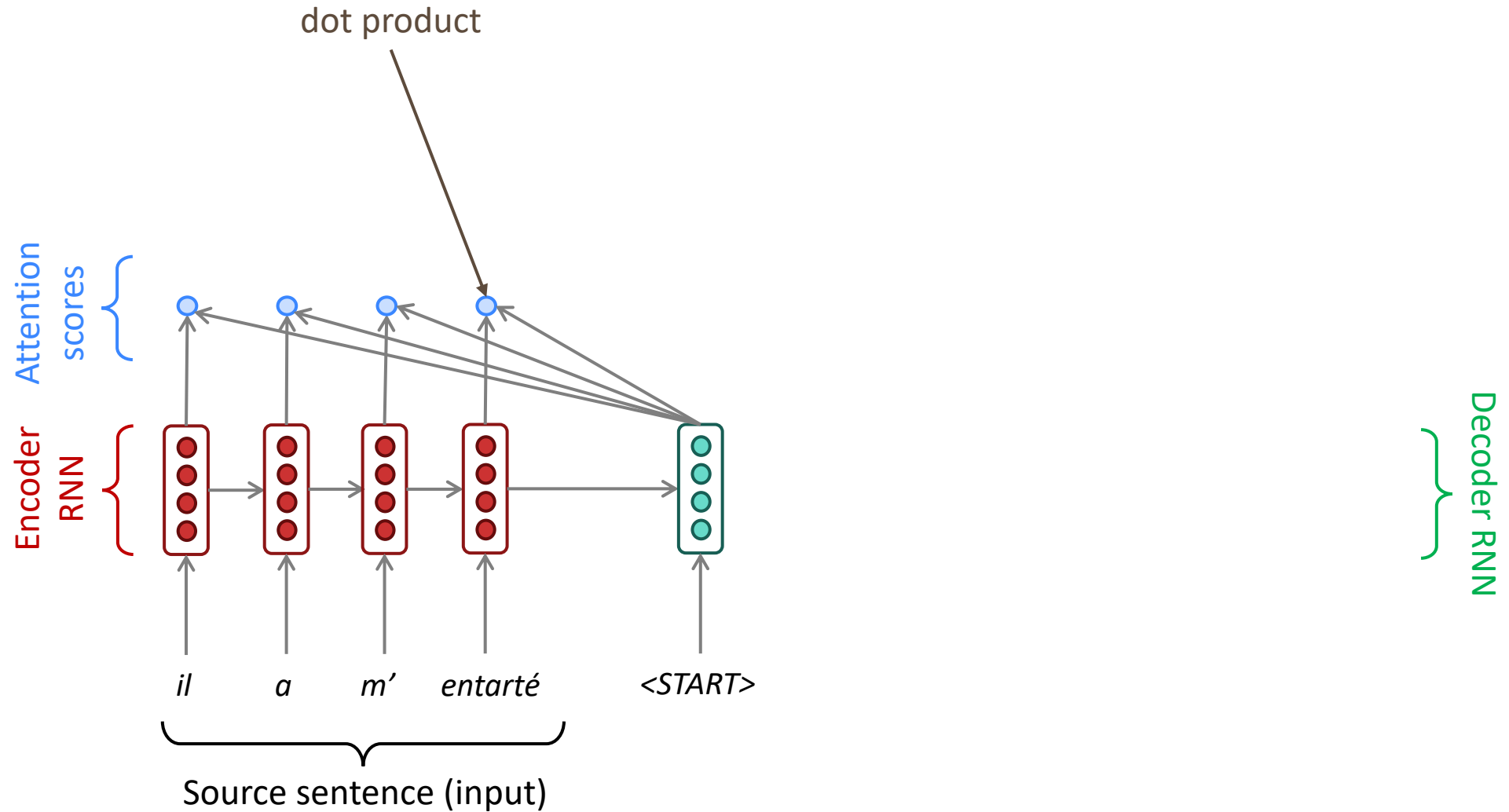
Sequence-to-sequence with attention



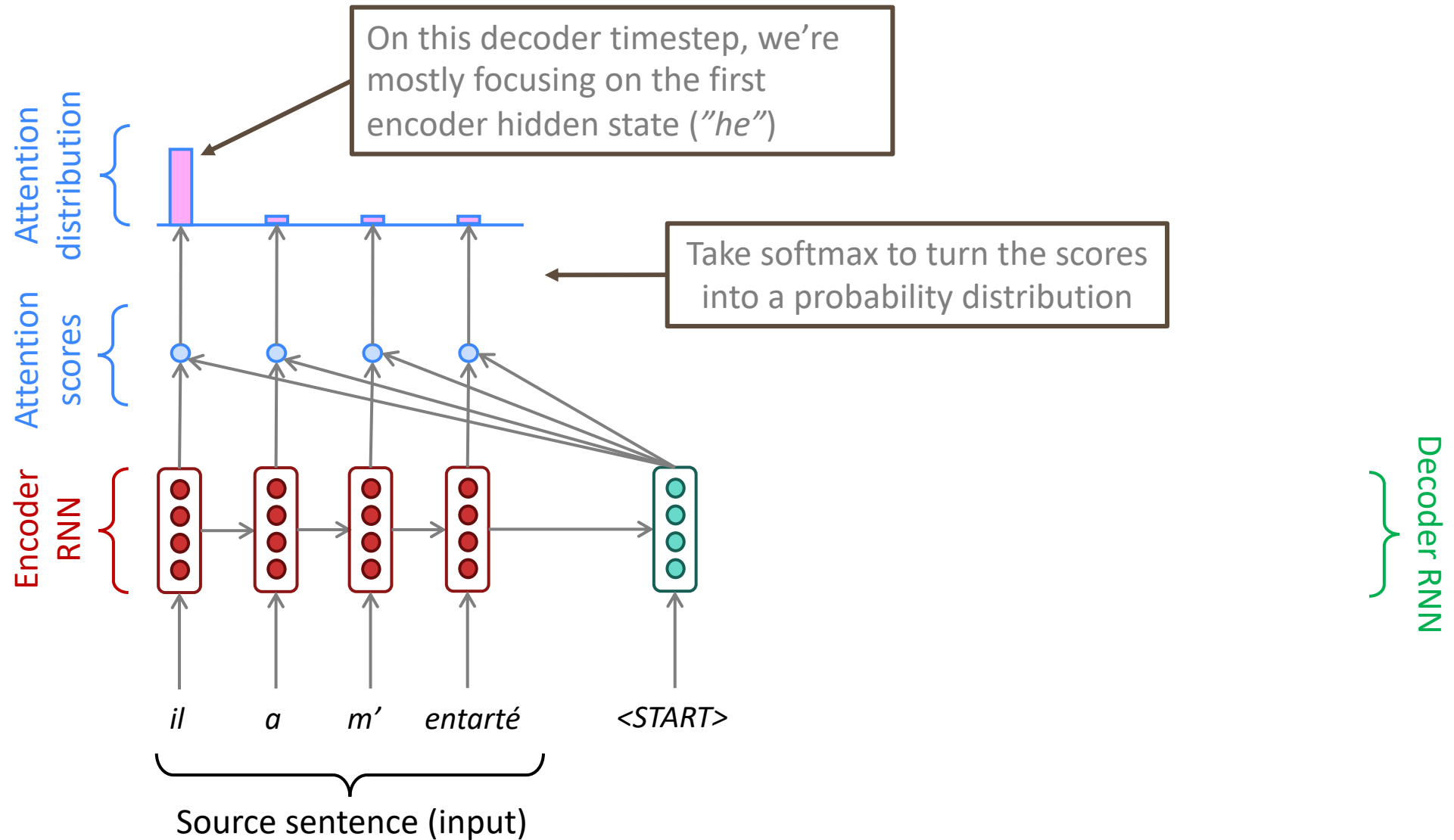
Sequence-to-sequence with attention



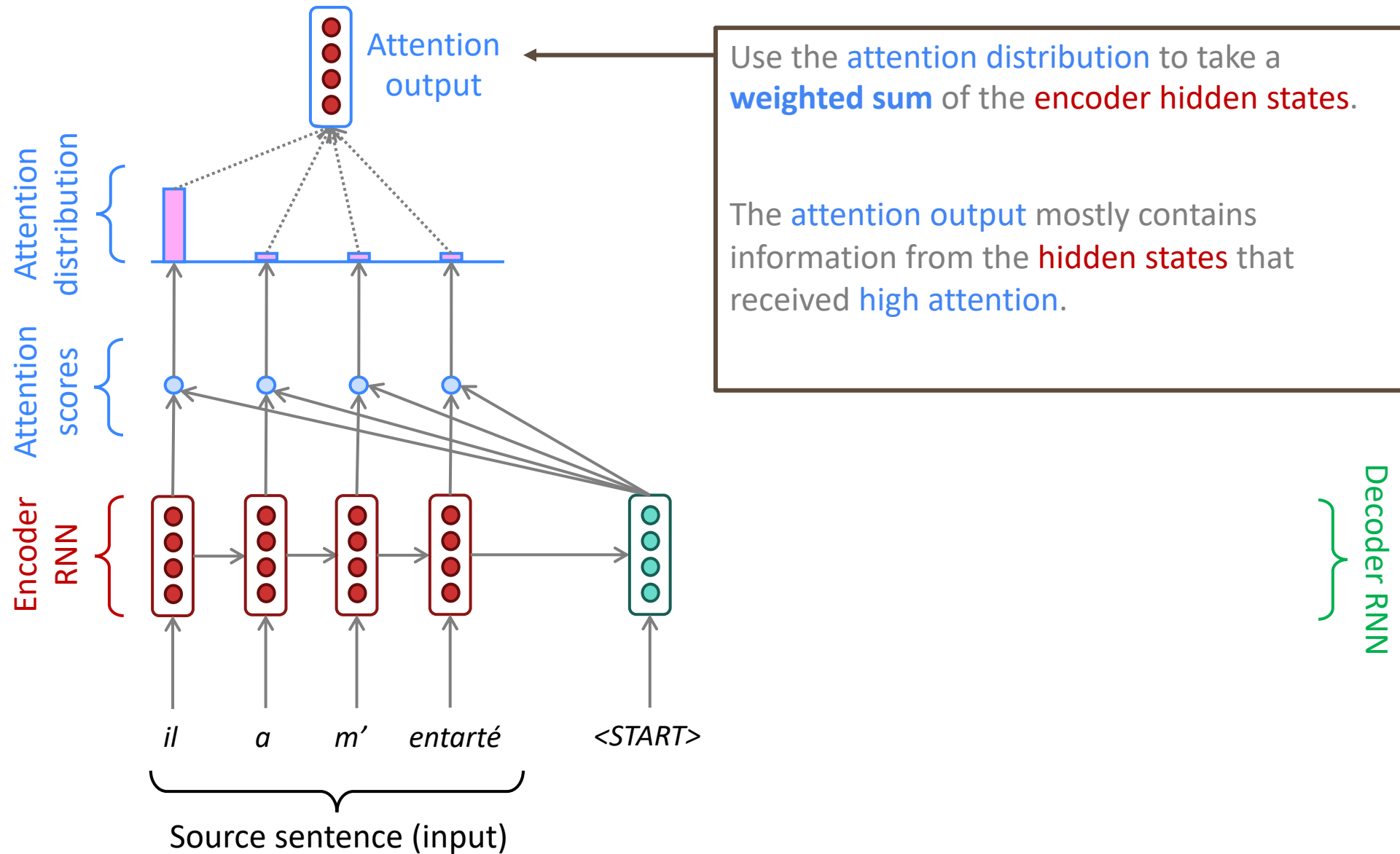
Sequence-to-sequence with attention



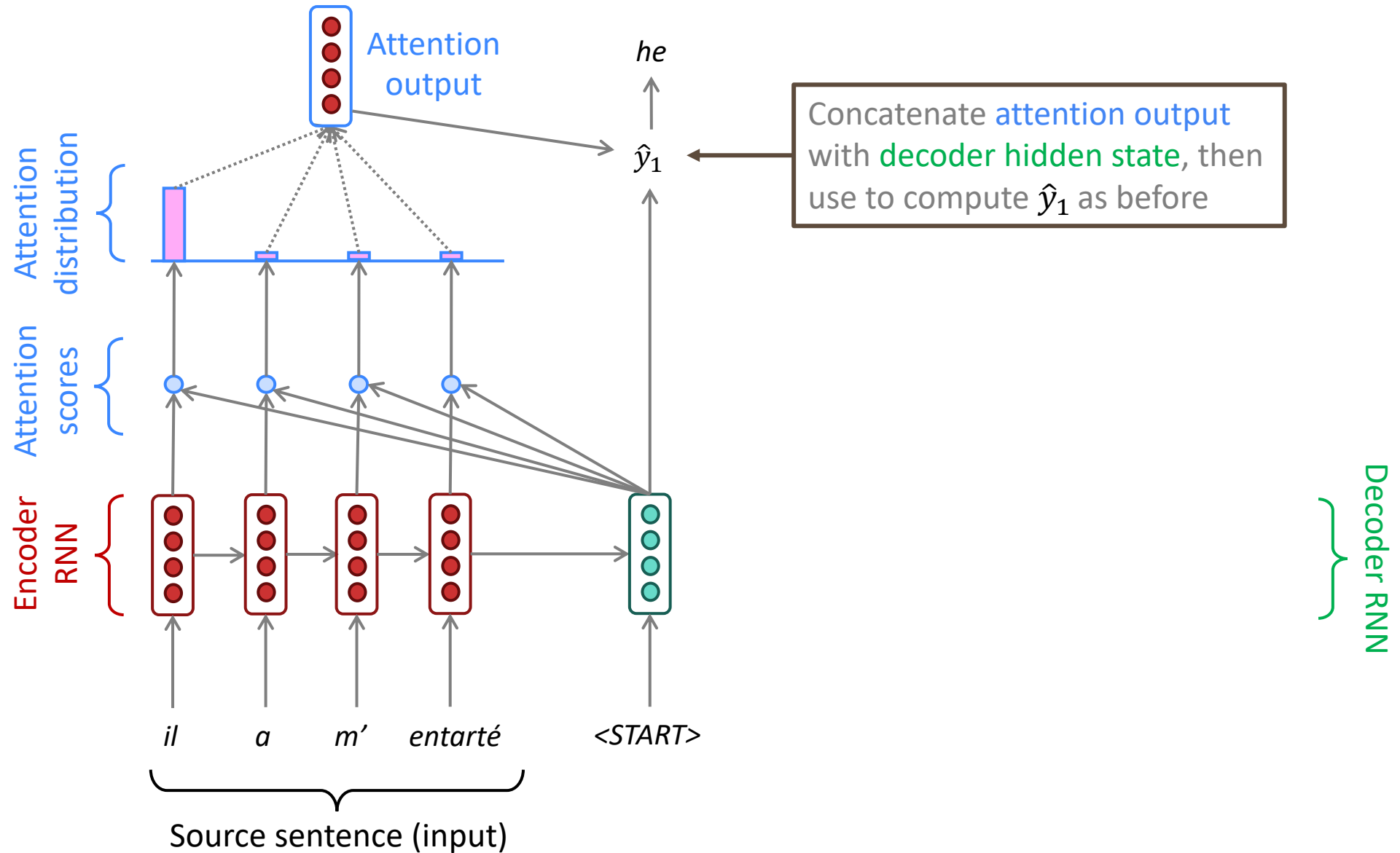
Sequence-to-sequence with attention



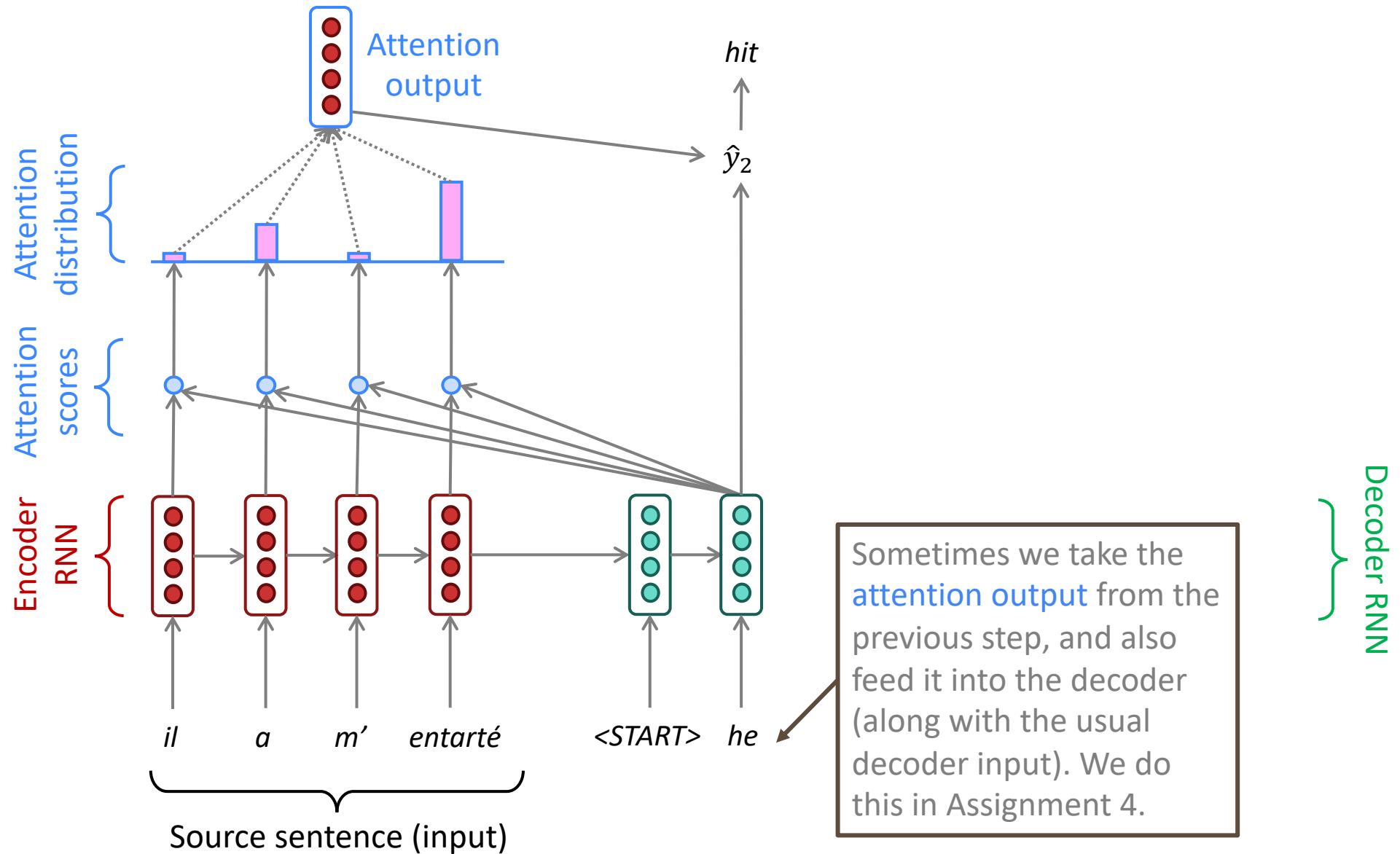
Sequence-to-sequence with attention



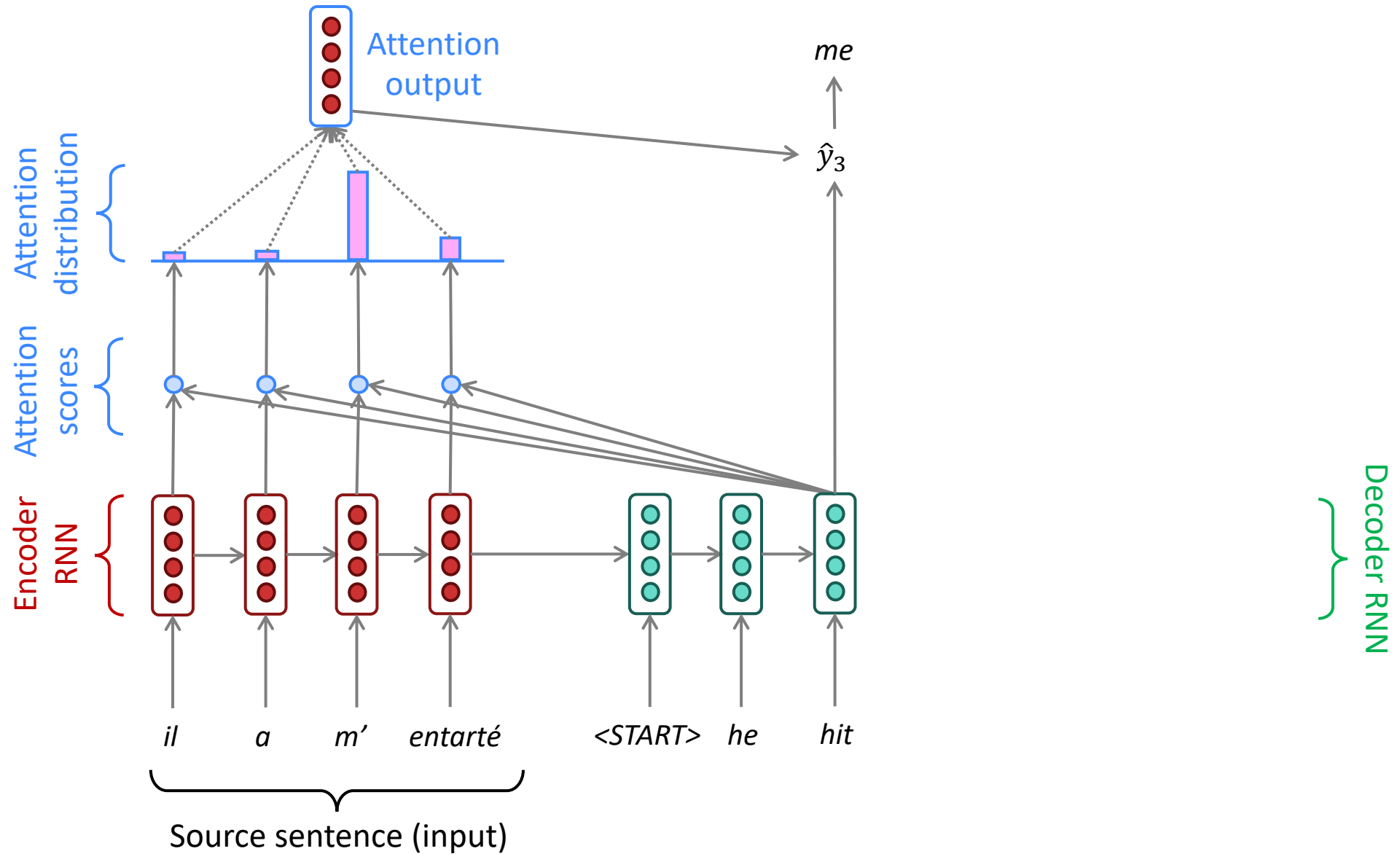
Sequence-to-sequence with attention



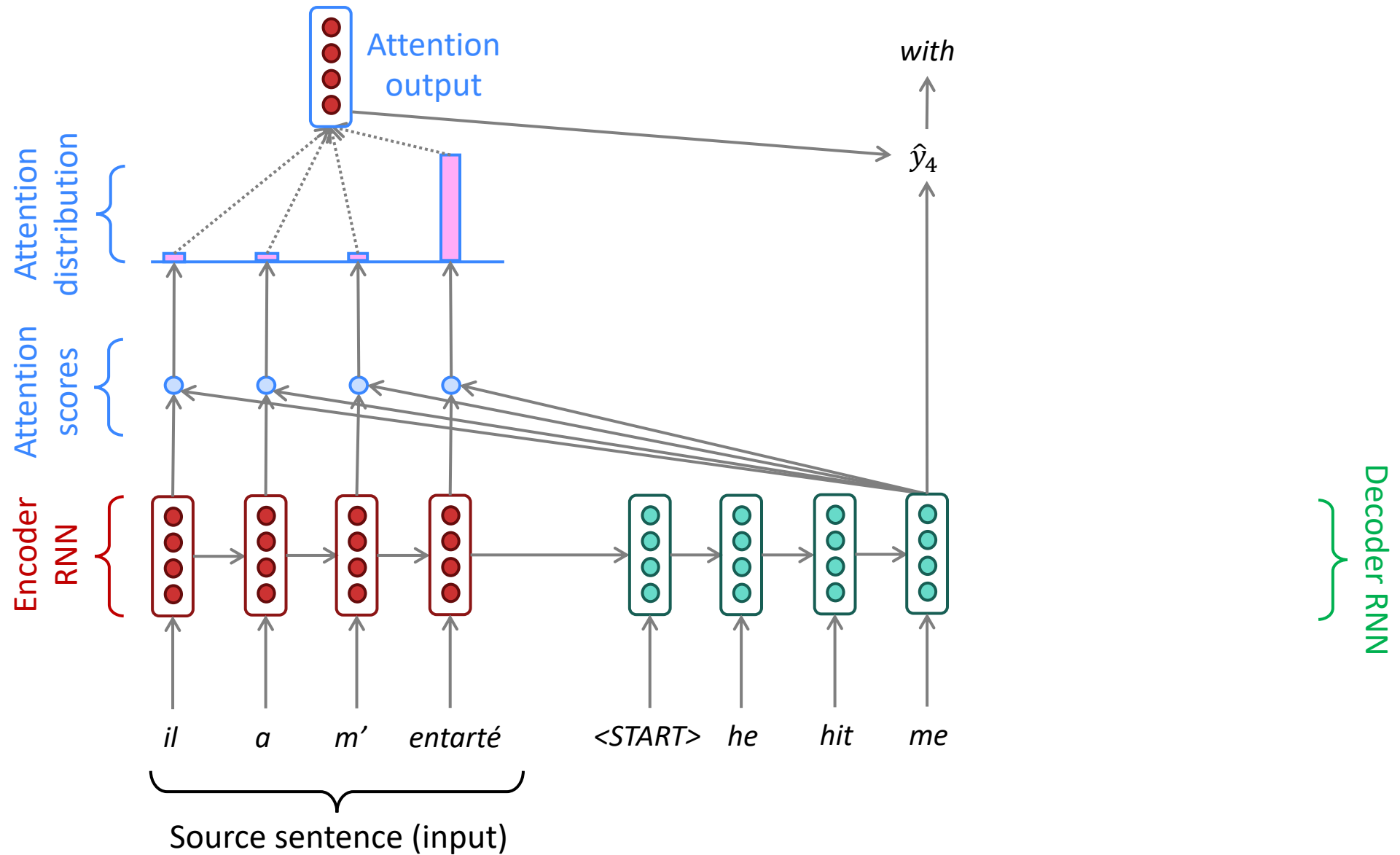
Sequence-to-sequence with attention



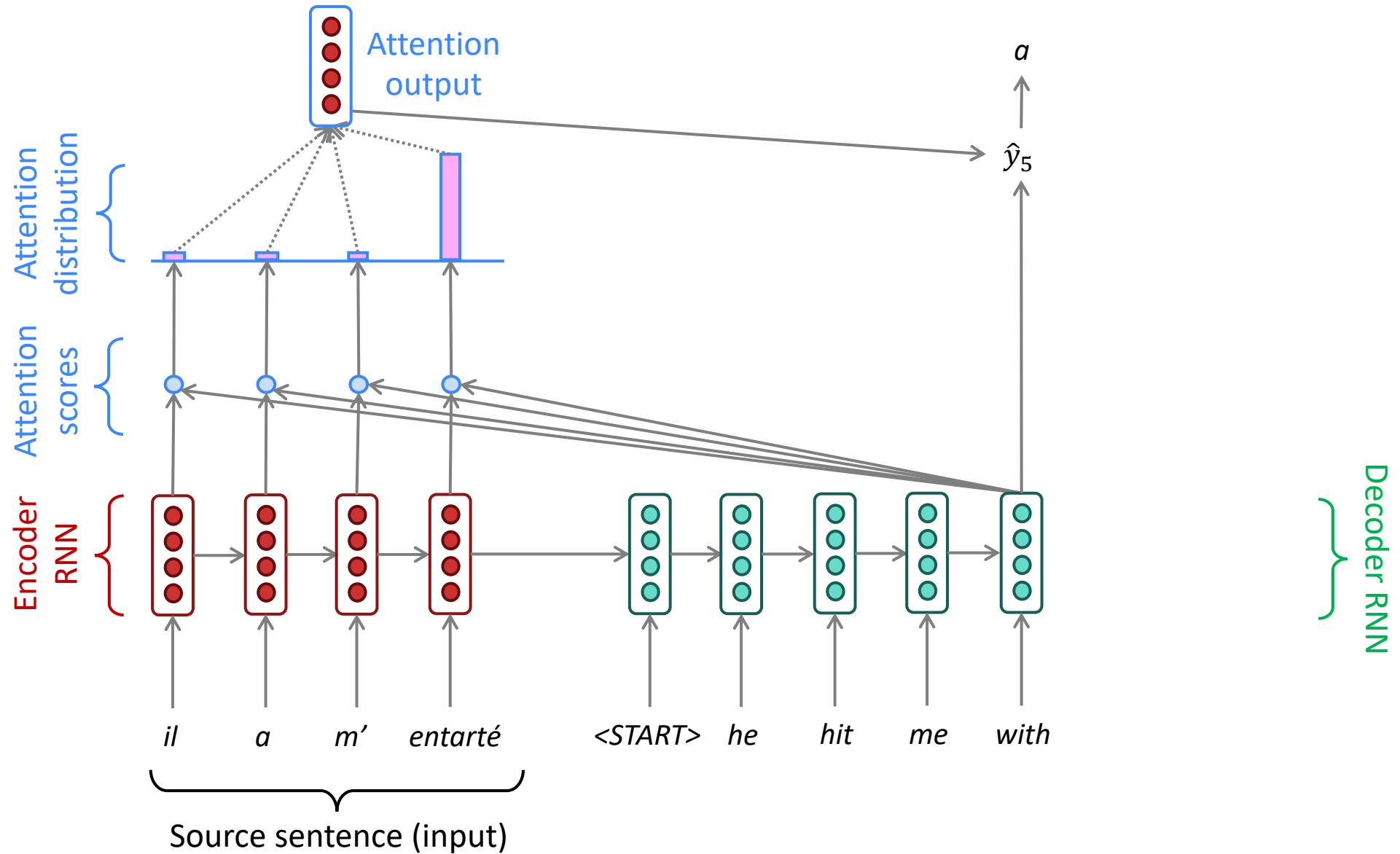
Sequence-to-sequence with attention



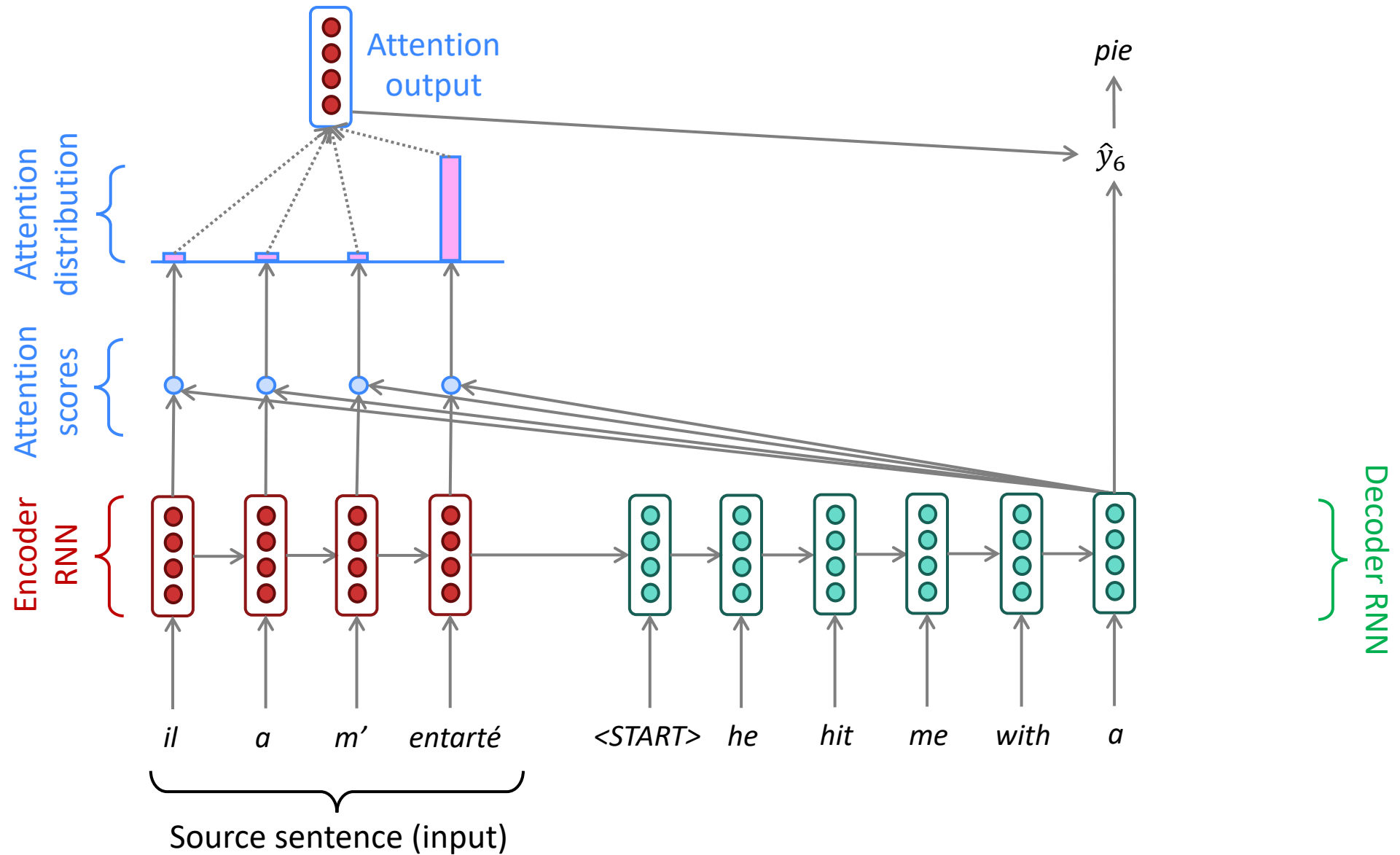
Sequence-to-sequence with attention



Sequence-to-sequence with attention



Sequence-to-sequence with attention



Attention: in equations

- We have encoder hidden states $h_1, \dots, h_N \in \mathbb{R}^h$
- On timestep t , we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores e^t for this step:

$$e^t = [s_t^T h_1, \dots, s_t^T h_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \text{softmax}(e^t) \in \mathbb{R}^N$$

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output a_t

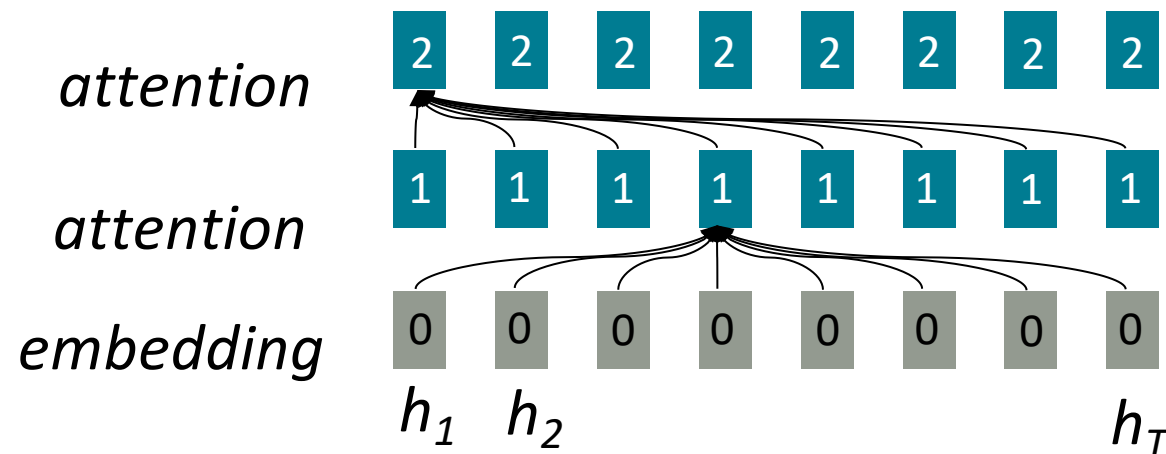
$$a_t = \sum_{i=1}^N \alpha_i^t h_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output a_t with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[a_t; s_t] \in \mathbb{R}^{2h}$$

Attention is parallelizable, and solves bottleneck issues.

- **Attention** treats each word's representation as a **query** to access and incorporate information from **a set of values**.
 - We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.
- Number of unparallelizable operations does not increase with sequence length.
- Maximum interaction distance: $O(1)$, since all words interact at every layer!



All words attend to all words in previous layer; most arrows here are omitted

Attention is great!



- Attention significantly **improves NMT performance**
 - It's very useful to allow decoder to focus on certain parts of the source
- Attention provides a **more “human-like” model** of the MT process
 - You can look back at the source sentence while translating, rather than needing to remember it all
- Attention **solves the bottleneck problem**
 - Attention allows decoder to look directly at source; bypass bottleneck
- Attention **helps with the vanishing gradient problem**
 - Provides shortcut to faraway states
- Attention provides **some interpretability**
 - By inspecting attention distribution, we see what the decoder was focusing on
 - We get (soft) **alignment for free!**
 - This is cool because we never explicitly trained an alignment system
 - The network just learned alignment by itself

	he	hit	me	with	a	pie
il	■					
a		■				
m'			■			
entarté		■	■	■	■	■

There are *several* attention variants

- We have some *values* $\mathbf{h}_1, \dots, \mathbf{h}_N \in \mathbb{R}^{d_1}$ and a *query* $\mathbf{s} \in \mathbb{R}^{d_2}$

- Attention always involves:

1. Computing the *attention scores* $\mathbf{e} \in \mathbb{R}^N$
2. Taking softmax to get *attention distribution* α :

There are multiple ways to do this

$$\alpha = \text{softmax}(\mathbf{e}) \in \mathbb{R}^N$$

3. Using attention distribution to take weighted sum of values:

$$\mathbf{a} = \sum_{i=1}^N \alpha_i \mathbf{h}_i \in \mathbb{R}^{d_1}$$

thus obtaining the *attention output* \mathbf{a} (sometimes called the *context vector*)

Attention variants

You'll think about the relative advantages/disadvantages of these in Assignment 4!

There are **several ways** you can compute $e \in \mathbb{R}^N$ from $\mathbf{h}_1, \dots, \mathbf{h}_N \in \mathbb{R}^{d_1}$ and $\mathbf{s} \in \mathbb{R}^{d_2}$:

- Basic dot-product attention: $e_i = \mathbf{s}^T \mathbf{h}_i \in \mathbb{R}$
 - Note: this assumes $d_1 = d_2$. This is the version we saw earlier.
- Multiplicative attention: $e_i = \mathbf{s}^T \mathbf{W} \mathbf{h}_i \in \mathbb{R}$ [Luong, Pham, and Manning 2015]
 - Where $\mathbf{W} \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix. Perhaps better called “bilinear attention”
- Reduced-rank multiplicative attention: $e_i = \mathbf{s}^T (\mathbf{U}^T \mathbf{V}) \mathbf{h}_i = (\mathbf{U} \mathbf{s})^T (\mathbf{V} \mathbf{h}_i)$ ← Remember this when we look at Transformers next week!
 - For low rank matrices $\mathbf{U} \in \mathbb{R}^{k \times d_2}$, $\mathbf{V} \in \mathbb{R}^{k \times d_1}$, $k \ll d_1, d_2$
- Additive attention: $e_i = \mathbf{v}^T \tanh(\mathbf{W}_1 \mathbf{h}_i + \mathbf{W}_2 \mathbf{s}) \in \mathbb{R}$ [Bahdanau, Cho, and Bengio 2014]
 - Where $\mathbf{W}_1 \in \mathbb{R}^{d_3 \times d_1}$, $\mathbf{W}_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $\mathbf{v} \in \mathbb{R}^{d_3}$ is a weight vector.
 - d_3 (the attention dimensionality) is a hyperparameter
 - “Additive” is a weird/bad name. It’s really using a feed-forward neural net layer.

More information: “Deep Learning for NLP Best Practices”, Ruder, 2017. <http://ruder.io/deep-learning-nlp-best-practices/index.html#attention>
“Massive Exploration of Neural Machine Translation Architectures”, Britz et al, 2017, <https://arxiv.org/pdf/1703.03906.pdf>

Attention is a *general* Deep Learning technique

- We've seen that attention is a great way to improve the sequence-to-sequence model for Machine Translation.
- However: You can use attention in **many architectures** (not just seq2seq) and **many tasks** (not just MT)

- More general definition of attention:
 - Given a set of vector *values*, and a vector *query*, attention is a technique to compute a weighted sum of the values, dependent on the query.

- We sometimes say that the *query attends to the values*.
- For example, in the seq2seq + attention model, each decoder hidden state (query) *attends to* all the encoder hidden states (values).

Attention is a *general* Deep Learning technique

- More general definition of attention:
 - Given a set of vector *values*, and a vector *query*, attention is a technique to compute a weighted sum of the values, dependent on the query.

Intuition:

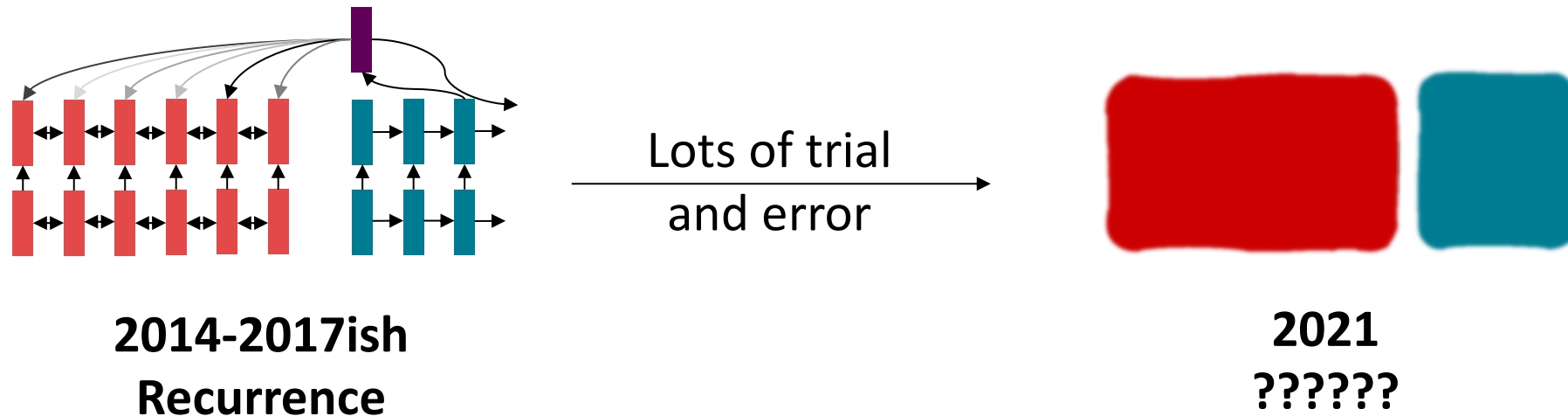
- The weighted sum is a *selective summary* of the information contained in the values, where the query determines which values to focus on.
- Attention is a way to obtain a *fixed-size representation of an arbitrary set of representations* (the values), dependent on some other representation (the query).

Upshot:

- Attention has become the powerful, flexible, general way pointer and memory manipulation in all deep learning models. A new idea from after 2010! From NMT!

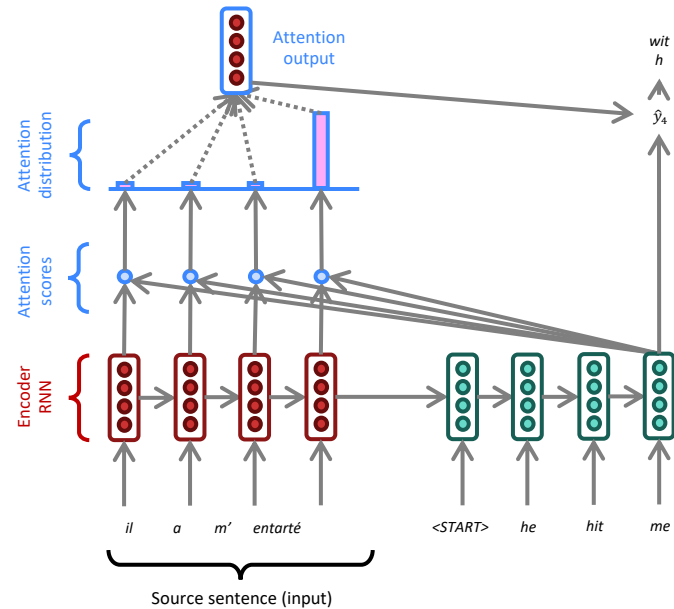
Do we even need recurrence at all?

- 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!



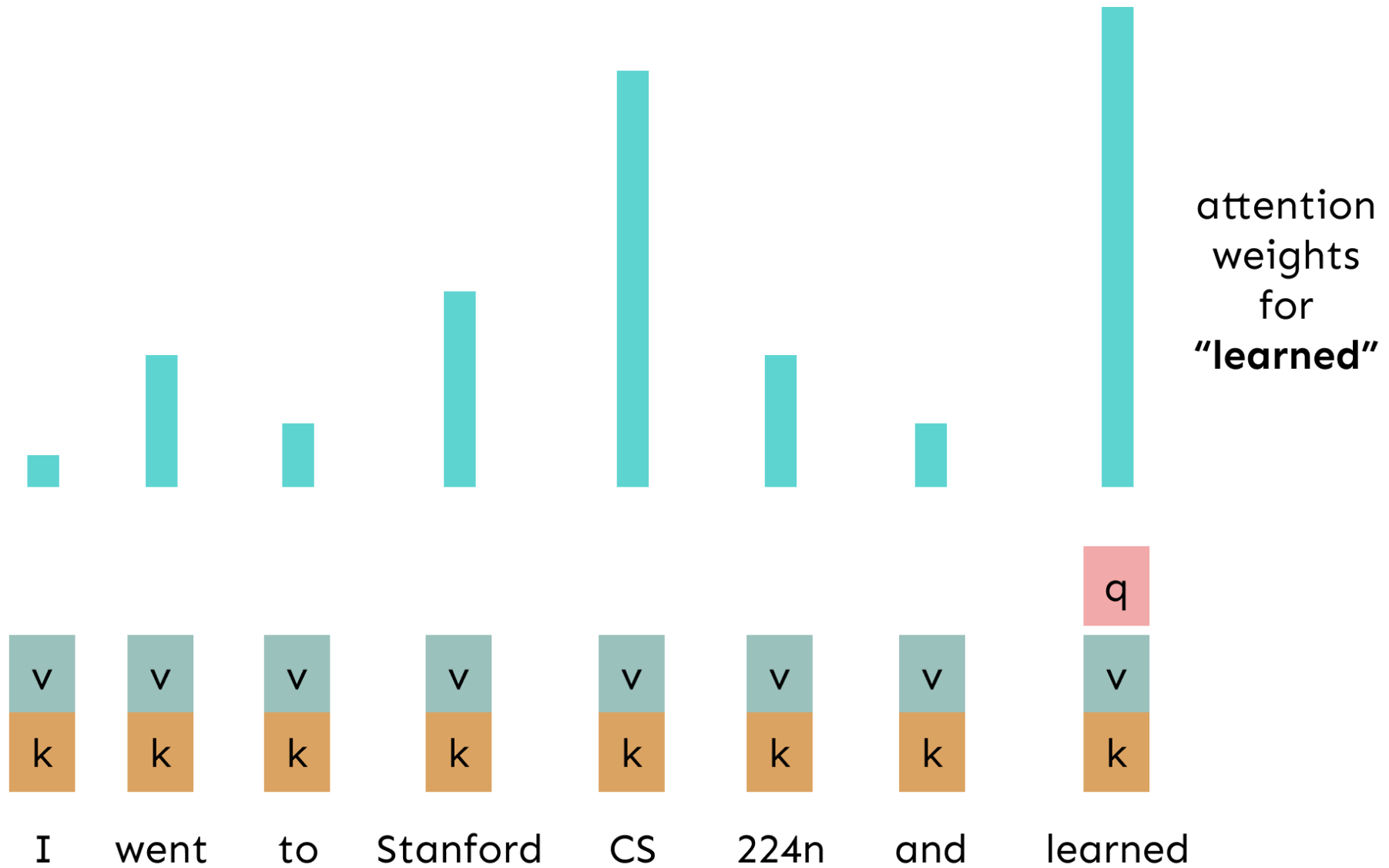
The building block we need: *self* attention

- What we talked about – **Cross** attention: paying attention to the input x to generate y_t



- What we need – **Self** attention: to generate y_t , we need to pay attention to $y_{<t}$

Self-Attention Hypothetical Example



Self-Attention: keys, queries, values from the same sequence

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

For each \mathbf{w}_i , let $\mathbf{x}_i = E\mathbf{w}_i$, 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 $\mathbb{R}^{d \times d}$

$$\mathbf{q}_i = Q\mathbf{x}_i \text{ (queries)} \quad \mathbf{k}_i = K\mathbf{x}_i \text{ (keys)} \quad \mathbf{v}_i = V\mathbf{x}_i \text{ (values)}$$

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

$$\mathbf{e}_{ij} = \mathbf{q}_i^\top \mathbf{k}_j \quad \alpha_{ij} = \frac{\exp(\mathbf{e}_{ij})}{\sum_{j'} \exp(\mathbf{e}_{ij'})}$$

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

$$\mathbf{o}_i = \sum_j \alpha_{ij} \mathbf{v}_i$$

$$\mathbf{o}_i = \sum_j \alpha_{ij} \mathbf{v}_i$$

Barriers and solutions for Self-Attention as a building block

Barriers

- Doesn't have an inherent notion of order!



Solutions

Fixing the first self-attention problem: **sequence order**

- 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**

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

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the \mathbf{p}_i to our inputs!
- Recall that \mathbf{x}_i is the embedding of the word at index i . The positioned embedding is:

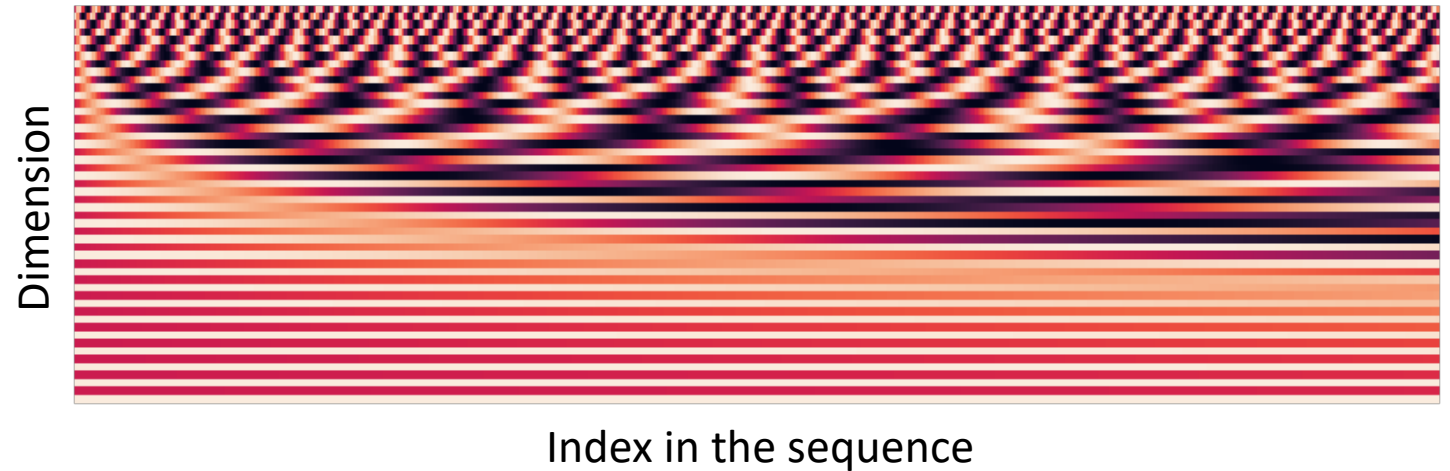
$$\tilde{\mathbf{x}}_i = \mathbf{x}_i + \mathbf{p}_i$$

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

Position representation vectors through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

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



- Pros:
 - Periodicity indicates that maybe “absolute position” isn’t as important
 - Maybe can extrapolate to longer sequences as periods restart!
- Cons:
 - Not learnable; also the extrapolation doesn’t really work!

Position representation vectors learned from scratch

- **Learned absolute position representations:** Let all p_i be learnable parameters!
Learn a matrix $\mathbf{p} \in \mathbb{R}^{d \times n}$, and let each \mathbf{p}_i be a column of that matrix!
- Pros:
 - Flexibility: each position gets to be learned to fit the data
- Cons:
 - Definitely can't extrapolate to indices outside $1, \dots, n$.
- Most systems use this!
- Sometimes people try more flexible representations of position:
 - Relative linear position attention [\[Shaw et al., 2018\]](#)
 - Dependency syntax-based position [\[Wang et al., 2019\]](#)

Barriers and solutions for Self-Attention as a building block

Barriers

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning! It's all just weighted averages



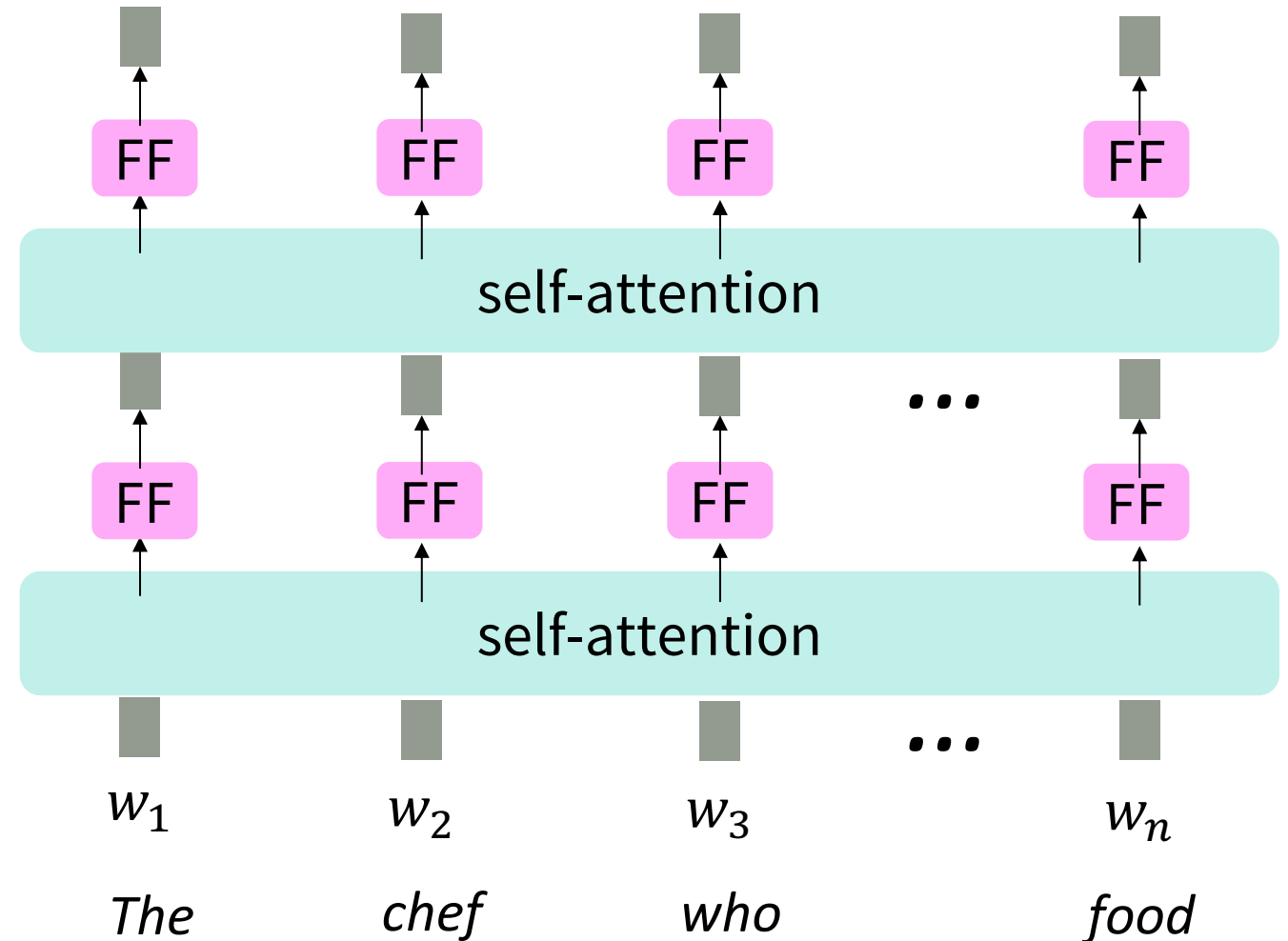
Solutions

- Add position representations to the inputs

Adding nonlinearities in self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages **value** vectors (Why? Look at the notes!)
- Easy fix: add a **feed-forward network** to post-process each output vector.

$$\begin{aligned} m_i &= MLP(\text{output}_i) \\ &= W_2 * \text{ReLU}(W_1 \text{output}_i + b_1) + b_2 \end{aligned}$$



Intuition: the FF network processes the result of attention

Barriers and solutions for Self-Attention as a building block

Barriers

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning magic! It's all just weighted averages
- Need to ensure we don't "look at the future" when predicting a sequence
 - Like in machine translation
 - Or language modeling



Solutions

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output.

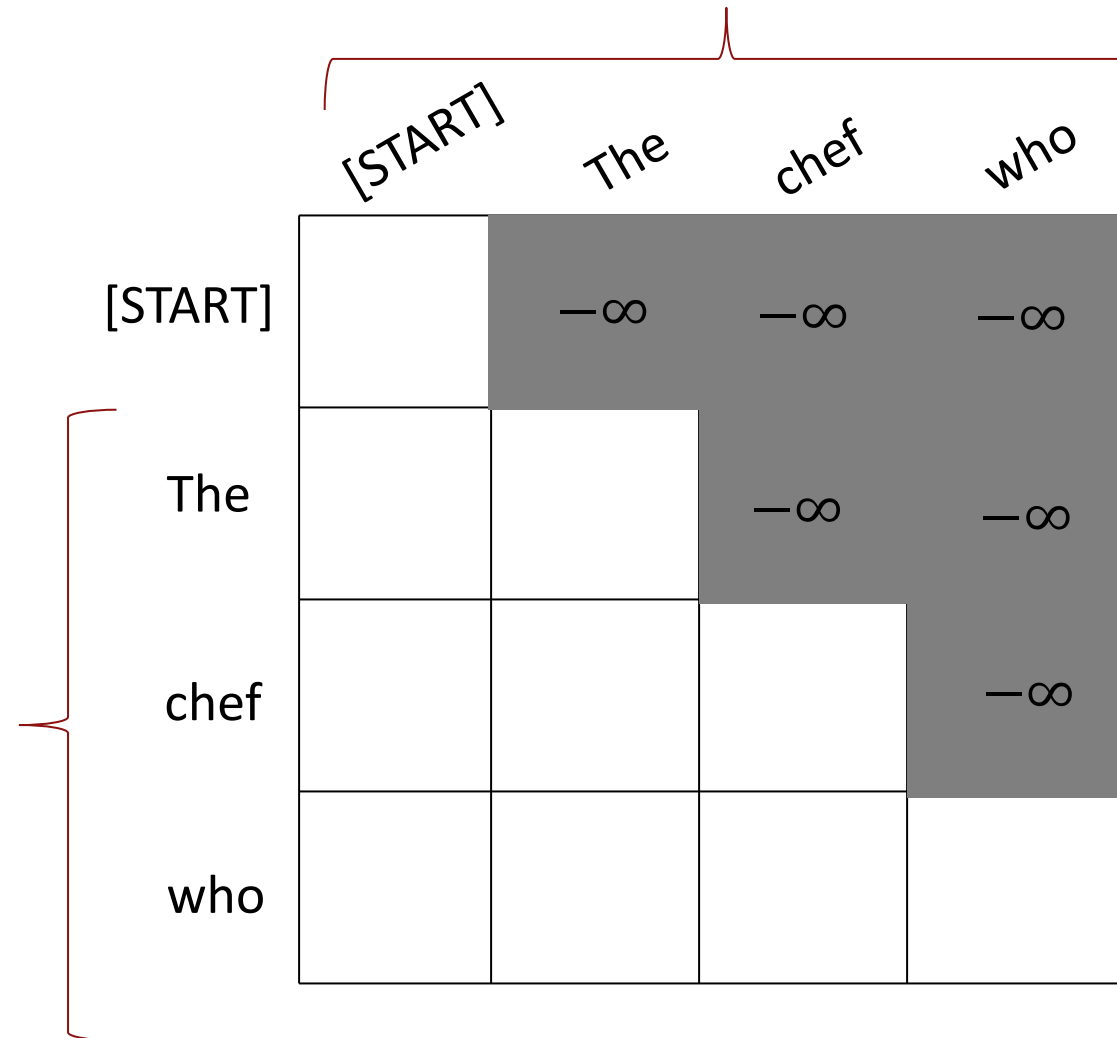
Masking the future in self-attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^\top k_j, & j \leq i \\ -\infty, & j > i \end{cases}$$

For encoding these words

We can look at these (not greyed out) words



Barriers and solutions for Self-Attention as a building block

Barriers

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning magic! It's all just weighted averages
- Need to ensure we don't "look at the future" when predicting a sequence
 - Like in machine translation
 - Or language modeling

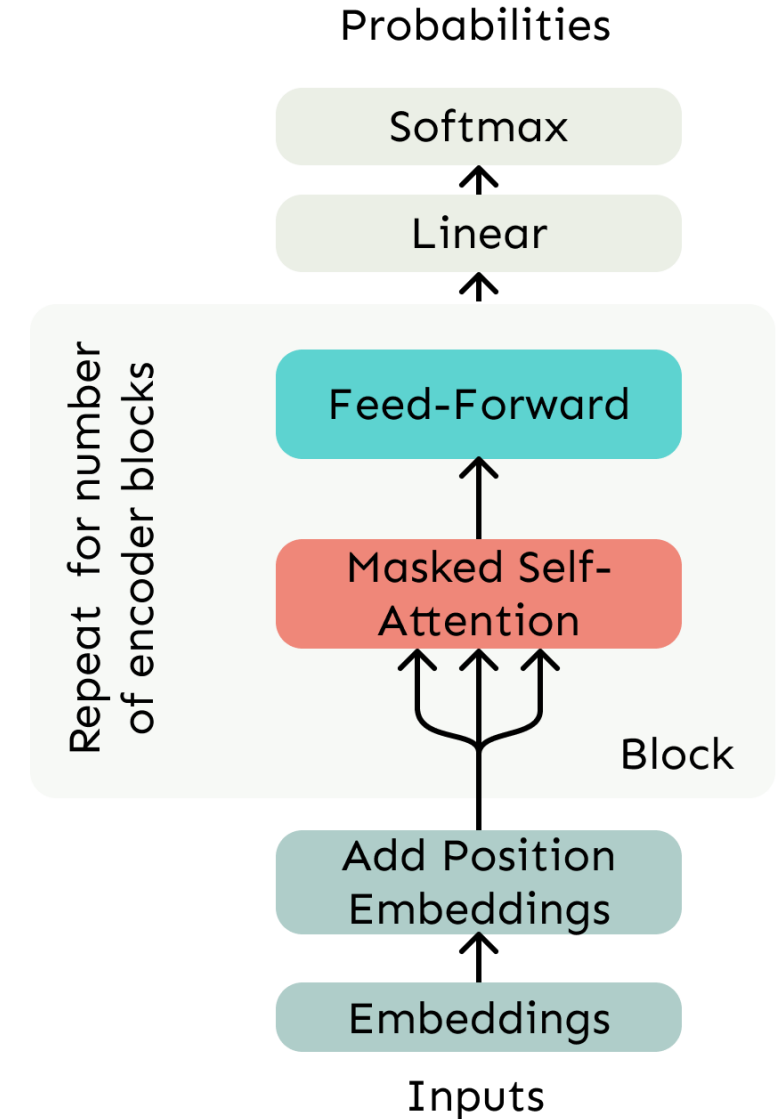


Solutions

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output.
- Mask out the future by artificially setting attention weights to 0!

Necessities for a self-attention building block:

- **Self-attention:**
 - the basis of the method.
- **Position representations:**
 - Specify the sequence order, since self-attention is an unordered function of its inputs.
- **Nonlinearities:**
 - At the output of the self-attention block
 - Frequently implemented as a simple feed-forward network.
- **Masking:**
 - In order to parallelize operations while not looking at the future.
 - Keeps information about the future from “leaking” to the past.

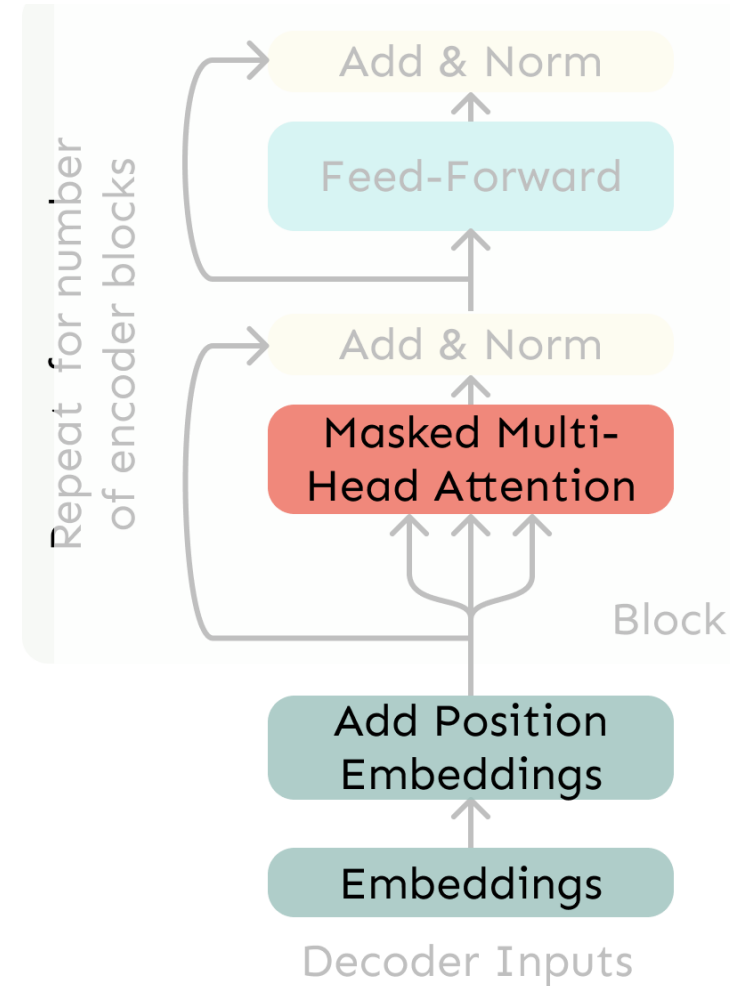


Outline

1. From recurrence (RNN) to attention-based NLP models
2. The Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers

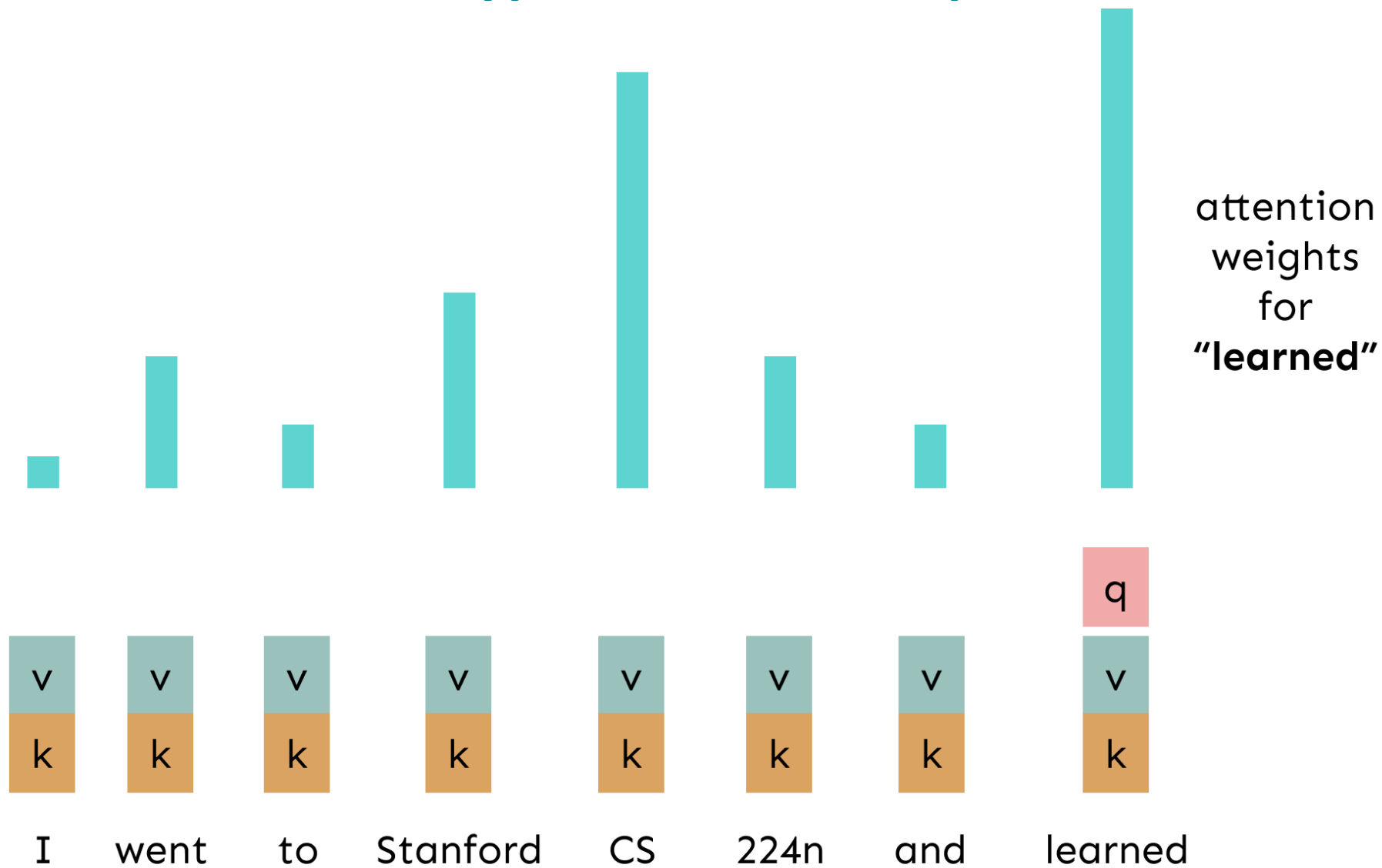
The Transformer Decoder

- A Transformer decoder is how we'll build systems like **language models**.
- It's a lot like our minimal self-attention architecture, but with a few more components.
- The embeddings and position embeddings are identical.
- We'll next replace our self-attention with **multi-head self-attention**.

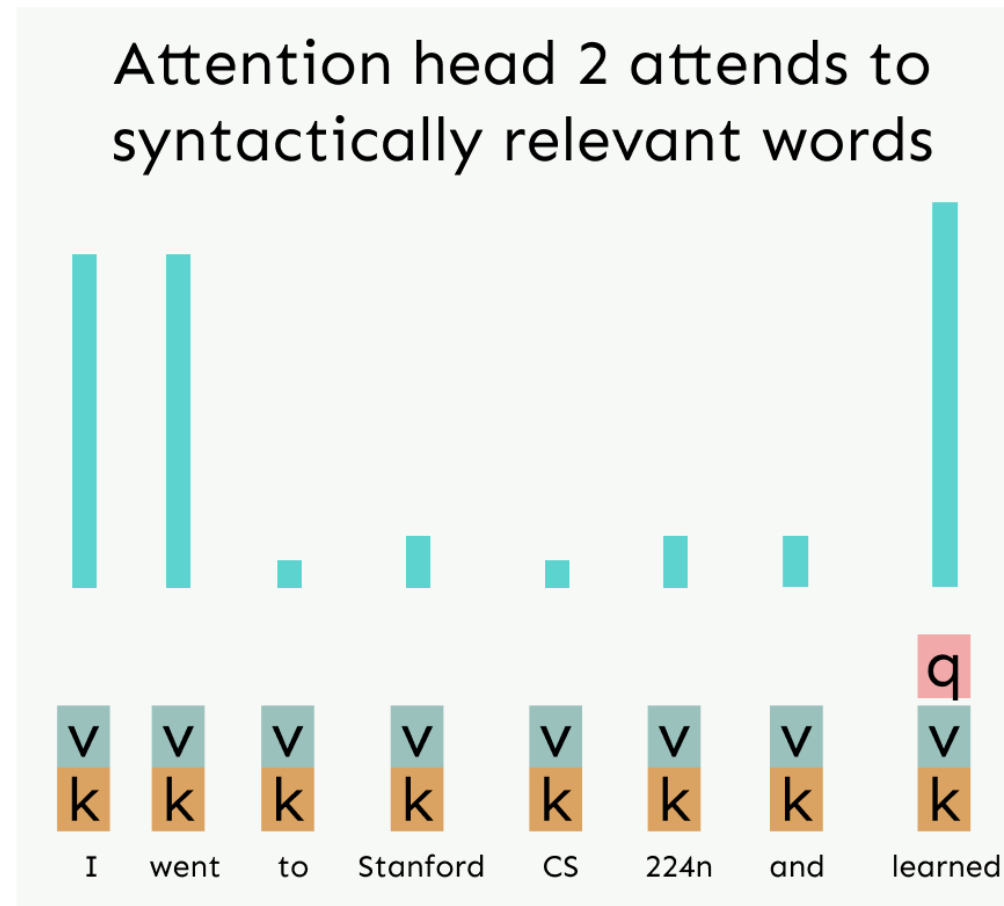
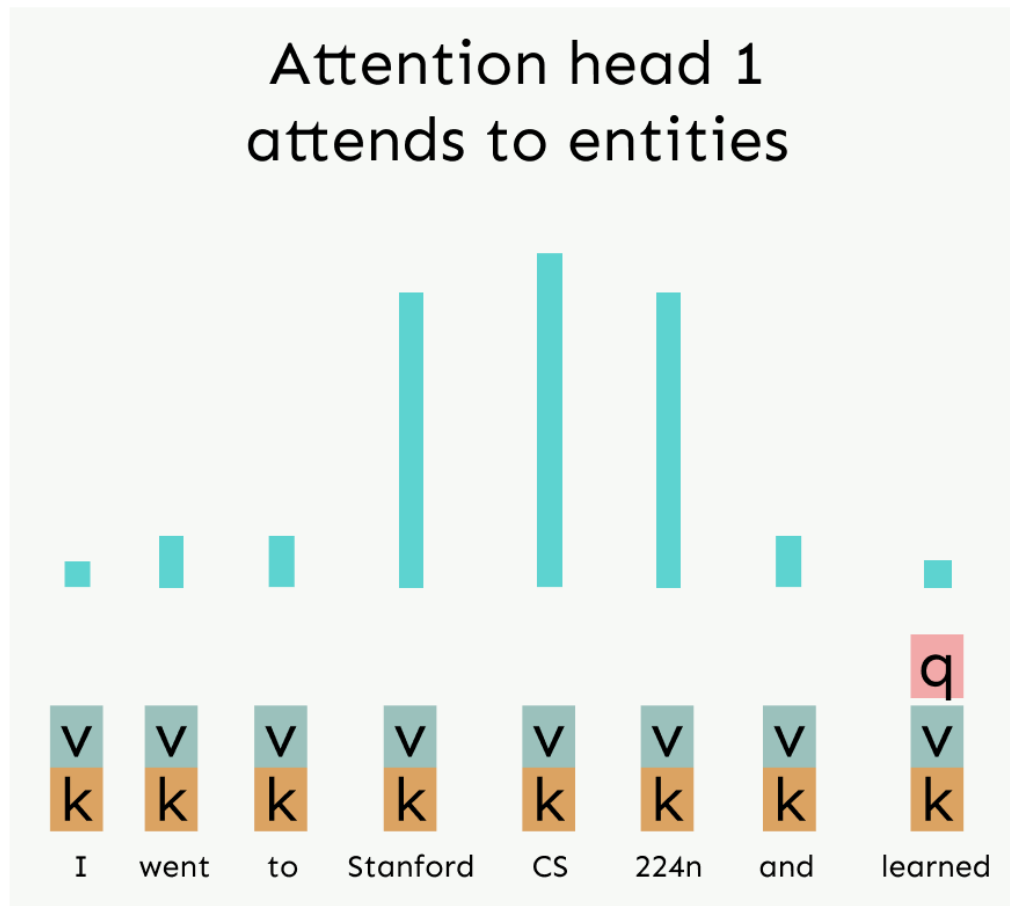


Transformer Decoder

Recall the Self-Attention Hypothetical Example



Hypothetical Example of Multi-Head Attention



I went to Stanford

CS 224n and learned

Sequence-Stacked form of Attention

- Let's look at how key-query-value attention is computed, in matrices.
 - Let $X = [x_1; \dots; x_n] \in \mathbb{R}^{n \times d}$ be the concatenation of input vectors.
 - First, note that $XK \in \mathbb{R}^{n \times d}$, $XQ \in \mathbb{R}^{n \times d}$, $XV \in \mathbb{R}^{n \times d}$.
 - The output is defined as $\text{output} = \text{softmax}(XQ(XK)^T)XV \in \mathbb{R}^{n \times d}$.

First, take the query-key dot products in one matrix multiplication: $XQ(XK)^T$

A diagram illustrating the first step of attention computation. On the left, a pink vertical rectangle labeled XQ is multiplied by an orange horizontal rectangle labeled $K^T X^T$. An equals sign follows, leading to a grey rounded rectangle labeled $XQK^T X^T$. To the right of this rectangle is the text $\in \mathbb{R}^{n \times n}$. Further to the right, the text "All pairs of attention scores!" is written in blue.

Next, softmax, and compute the weighted average with another matrix multiplication.

A diagram illustrating the second step of attention computation. A grey rounded rectangle labeled $XQK^T X^T$ is enclosed in large square brackets. To the left of the brackets is the word "softmax". To the right of the brackets is a teal vertical rectangle labeled XV . An equals sign follows, leading to a grey vertical rectangle representing the output. To the right of this rectangle is the text "output $\in \mathbb{R}^{n \times d}$ ". A curved arrow points from the $XQK^T X^T$ rectangle in the upper diagram to the $XQK^T X^T$ rectangle in this diagram.

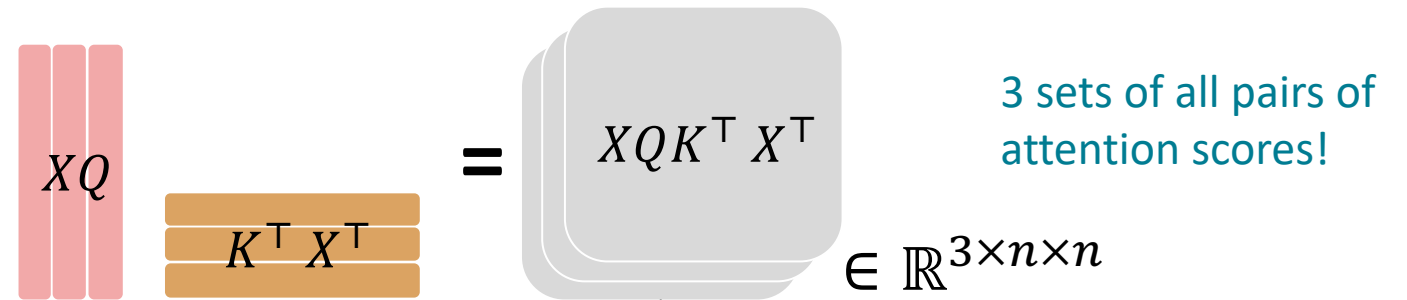
Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
 - For word i , self-attention “looks” where $x_i^\top Q^\top K x_j$ is high, but maybe we want to focus on different j for different reasons?
- We’ll define **multiple attention “heads”** through multiple Q,K,V matrices
- Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and ℓ ranges from 1 to h .
- Each attention head performs attention independently:
 - $\text{output}_\ell = \text{softmax}(X Q_\ell K_\ell^\top X^\top) * X V_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
 - $\text{output} = [\text{output}_1; \dots; \text{output}_h] Y$, where $Y \in \mathbb{R}^{d \times d}$
- Each head gets to “look” at different things, and construct value vectors differently.

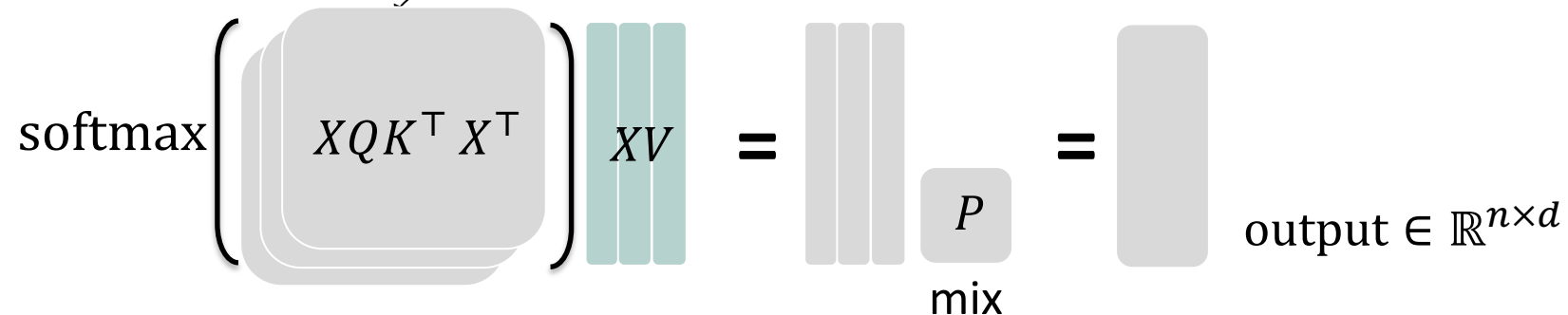
Multi-head self-attention is computationally efficient

- Even though we compute h many attention heads, it's not really more costly.
 - We compute $XQ \in \mathbb{R}^{n \times d}$, and then reshape to $\mathbb{R}^{n \times h \times d/h}$. (Likewise for XK , XV .)
 - Then we transpose to $\mathbb{R}^{h \times n \times d/h}$; now the head axis is like a batch axis.
 - Almost everything else is identical, and the **matrices are the same sizes**.

First, take the query-key dot products in one matrix multiplication: $XQ(XK)^\top$



Next, softmax, and compute the weighted average with another matrix multiplication.



Scaled Dot Product [Vaswani et al., 2017]

- “**Scaled Dot Product**” attention aids in training.
- When dimensionality d becomes large, dot products between vectors tend to become large.
 - Because of this, inputs to the softmax function can be large, making the gradients small.

- Instead of the self-attention function we’ve seen:

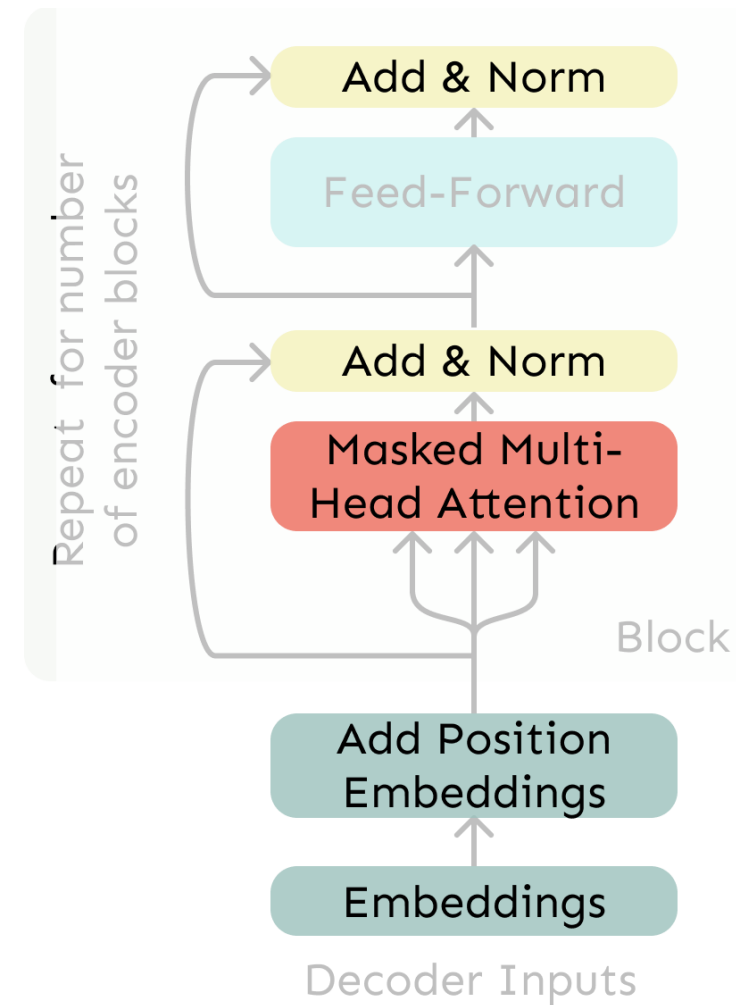
$$\text{output}_\ell = \text{softmax}(XQ_\ell K_\ell^\top X^\top) * XV_\ell$$

- We divide the attention scores by $\sqrt{d/h}$, to stop the scores from becoming large just as a function of d/h (The dimensionality divided by the number of heads.)

$$\text{output}_\ell = \text{softmax}\left(\frac{XQ_\ell K_\ell^\top X^\top}{\sqrt{d/h}}\right) * XV_\ell$$

The Transformer Decoder

- Now that we've replaced self-attention with multi-head self-attention, we'll go through two **optimization tricks** that end up being :
 - **Residual Connections**
 - **Layer Normalization**
- In most Transformer diagrams, these are often written together as "Add & Norm"



Transformer Decoder

The Transformer Encoder: Residual connections [He et al., 2016]

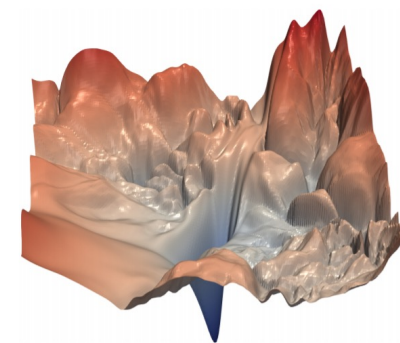
- **Residual connections** are a trick to help models train better.
 - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where i represents the layer)



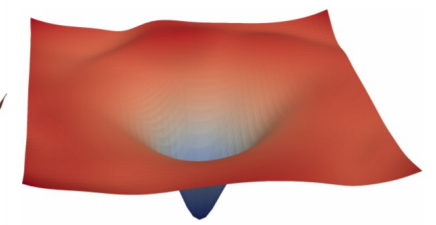
- We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$ (so we only have to learn “the residual” from the previous layer)



- Gradient is **great** through the residual connection; it's 1!
- Bias towards the identity function!



[no residuals]



[residuals]

[Loss landscape visualization,
[Li et al., 2018](#), on a ResNet]

The Transformer Encoder: **Layer normalization** [[Ba et al., 2016](#)]

- **Layer normalization** is a trick to help models train faster.
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**.
 - LayerNorm's success may be due to its normalizing gradients [[Xu et al., 2019](#)]
- Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.
- Let $\mu = \sum_{j=1}^d x_j$; this is the mean; $\mu \in \mathbb{R}$.
- Let $\sigma = \sqrt{\frac{1}{d} \sum_{j=1}^d (x_j - \mu)^2}$; this is the standard deviation; $\sigma \in \mathbb{R}$.
- Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned “gain” and “bias” parameters. (Can omit!)
- Then layer normalization computes:

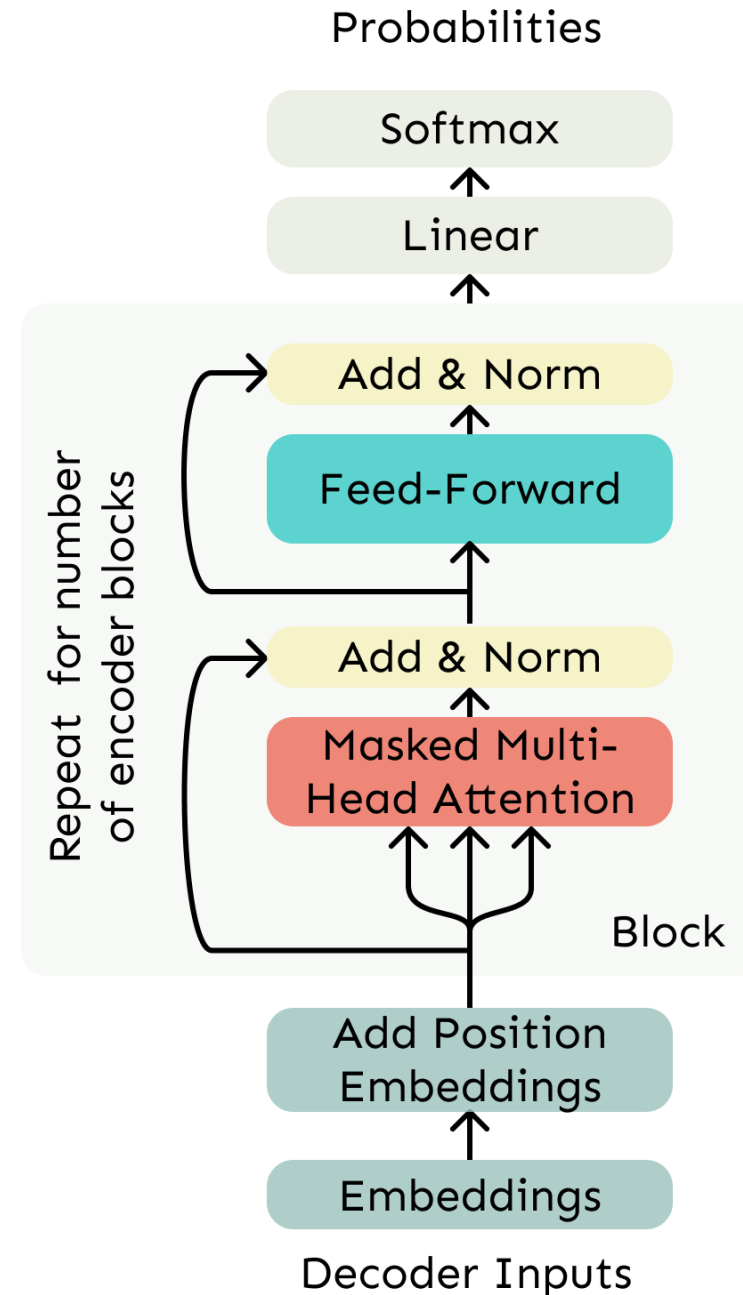
$$\text{output} = \frac{x - \mu}{\sqrt{\sigma + \epsilon}} * \gamma + \beta$$

Normalize by scalar mean and variance \rightarrow

\leftarrow Modulate by learned elementwise gain and bias

The Transformer Decoder

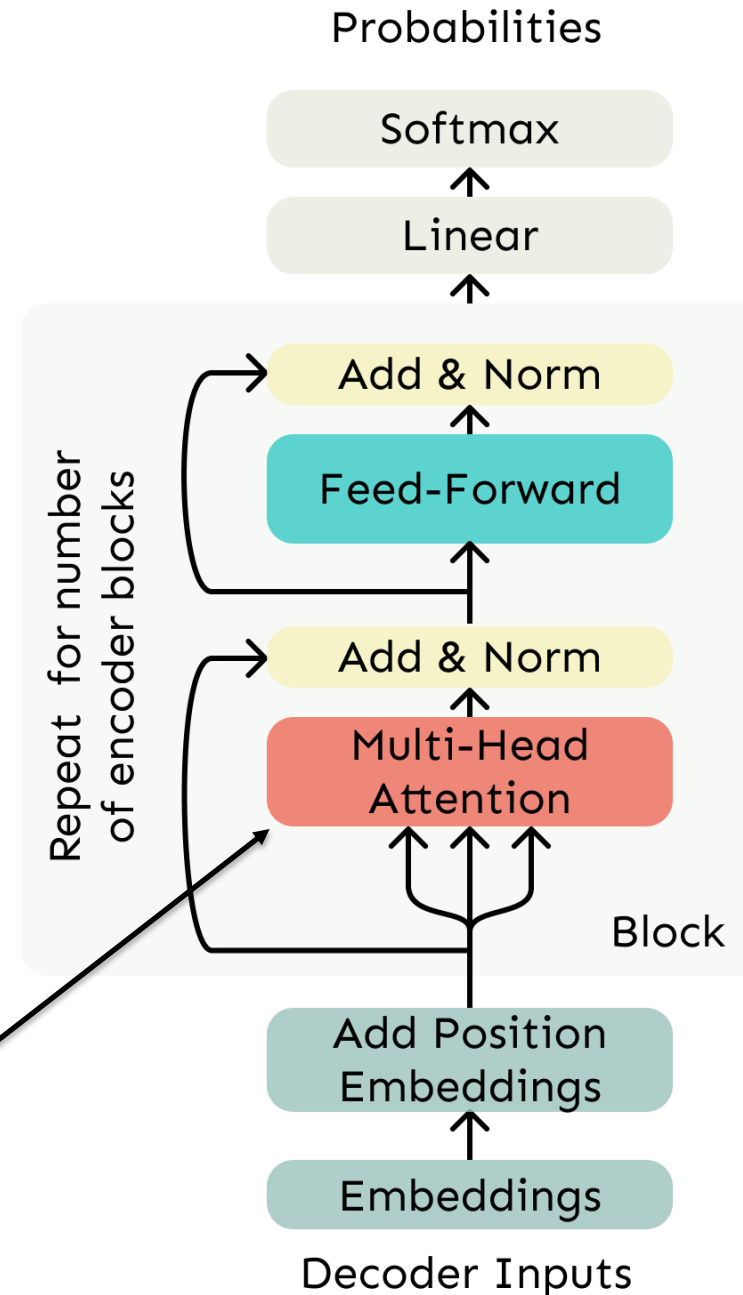
- The Transformer Decoder is a stack of Transformer Decoder **Blocks**.
- Each Block consists of:
 - Self-attention
 - Add & Norm
 - Feed-Forward
 - Add & Norm
- That's it! We've gone through the Transformer Decoder.



The Transformer Encoder

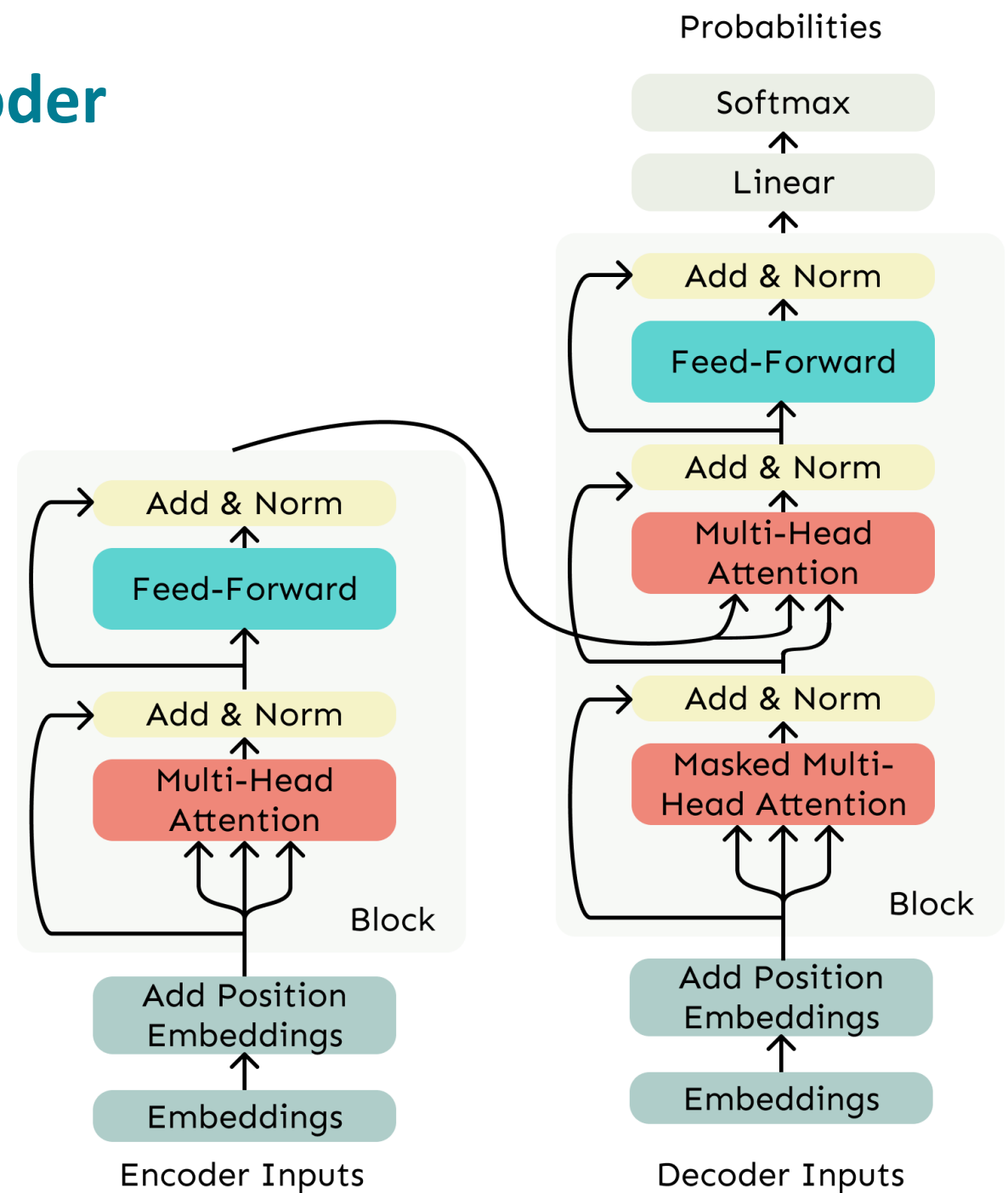
- The Transformer Decoder constrains to **unidirectional context**, as for **language models**.
- What if we want **bidirectional context**, like in a bidirectional RNN?
- This is the Transformer Encoder. The only difference is that we **remove the masking** in the self-attention.

No Masking!



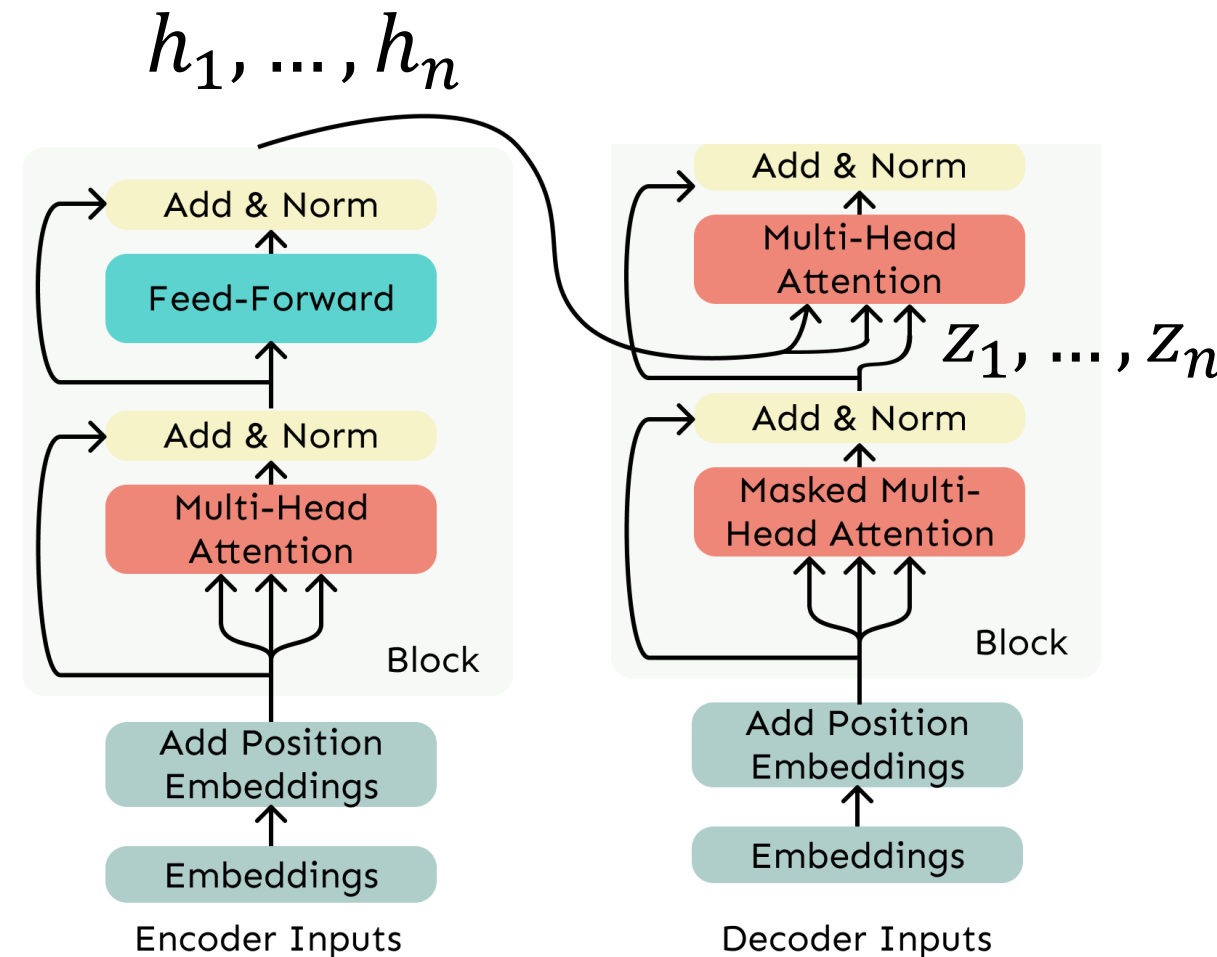
The Transformer Encoder-Decoder

- Recall that in machine translation, we processed the source sentence with a **bidirectional** model and generated the target with a **unidirectional model**.
- For this kind of seq2seq format, we often use a Transformer Encoder-Decoder.
- We use a normal Transformer Encoder.
- Our Transformer Decoder is modified to perform **cross-attention** to the output of the Encoder.



Cross-attention (details)

- We saw that self-attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let h_1, \dots, h_n be **output** vectors from the Transformer **encoder**; $x_i \in \mathbb{R}^d$
- Let z_1, \dots, z_n be input vectors from the Transformer **decoder**, $z_i \in \mathbb{R}^d$
- Then keys and values are drawn from the **encoder** (like a memory):
 - $k_i = Kh_i, v_i = Vh_i$.
- And the queries are drawn from the **decoder**, $q_i = Qz_i$.



Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers

Great Results with Transformers

First, Machine Translation from the original Transformers paper!

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$

Great Results with Transformers

Next, document generation!

Model	Test perplexity	ROUGE-L
<i>seq2seq-attention, L = 500</i>	5.04952	12.7
<i>Transformer-ED, L = 500</i>	2.46645	34.2
<i>Transformer-D, L = 4000</i>	2.22216	33.6
<i>Transformer-DMCA, no MoE-layer, L = 11000</i>	2.05159	36.2
<i>Transformer-DMCA, MoE-128, L = 11000</i>	1.92871	37.9
<i>Transformer-DMCA, MoE-256, L = 7500</i>	1.90325	38.8

The old standard



Transformers all the way down.



Great Results with Transformers

Before too long, most Transformers results also included **pretraining**, a method we'll go over on Thursday.

Transformers' parallelizability allows for efficient pretraining, and have made them the de-facto standard.

On this popular aggregate benchmark, for example:



All top models are Transformer (and pretraining)-based.

Rank	Name	Model	URL	Score
1	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4		90.8
2	HFL iFLYTEK	MacALBERT + DKM		90.7
+	Alibaba DAMO NLP	StructBERT + TAPT		90.6
+	PING-AN Omni-Sinitic	ALBERT + DAAF + NAS		90.6
5	ERNIE Team - Baidu	ERNIE		90.4
6	T5 Team - Google	T5		90.3

More results Thursday when we discuss pretraining.

[[Liu et al., 2018](#)]

Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers

What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today):**
 - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
 - For recurrent models, it only grew linearly!
- **Position representations:**
 - Are simple absolute indices the best we can do to represent position?
 - Relative linear position attention [\[Shaw et al., 2018\]](#)
 - Dependency syntax-based position [\[Wang et al., 2019\]](#)

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(n^2d)$, where n is the sequence length, and d is the dimensionality.

XQ $K^T X^T$ = $XQK^T X^T \in \mathbb{R}^{n \times n}$

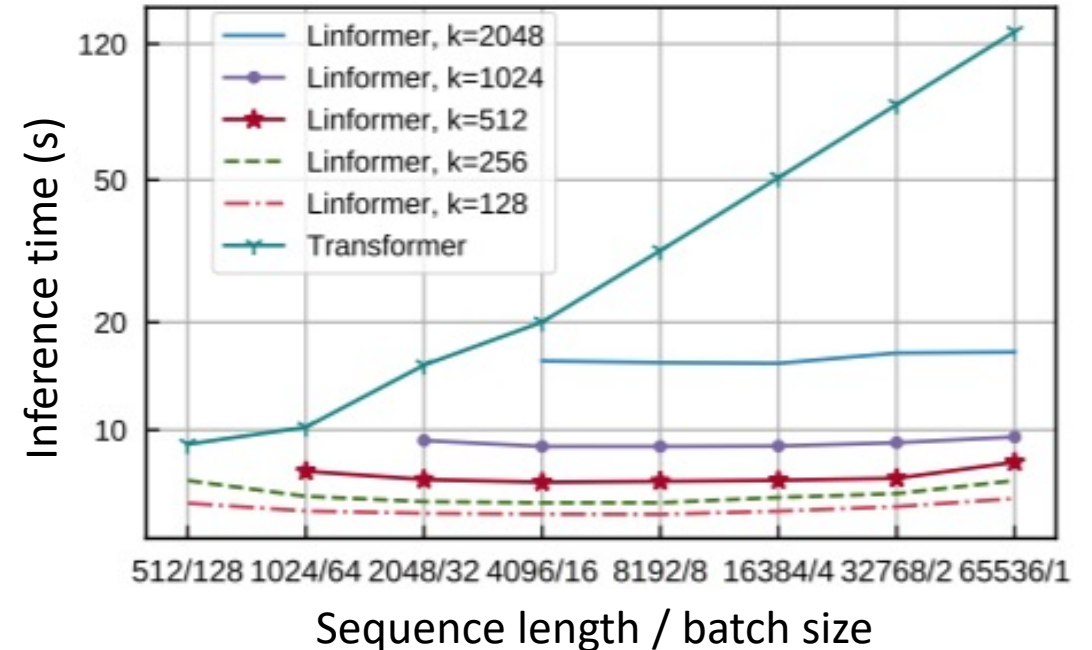
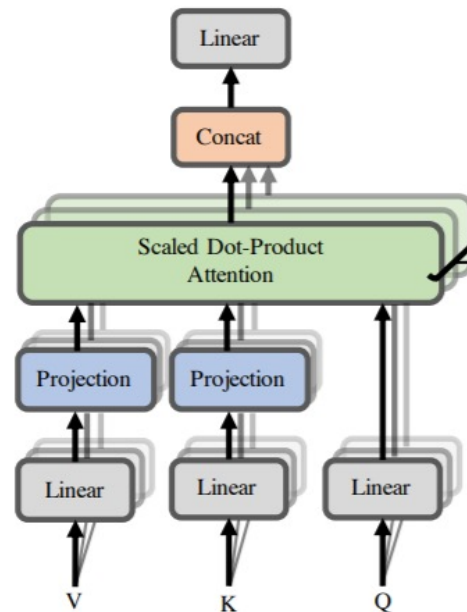
Need to compute all pairs of interactions!
 $O(n^2d)$

- Think of d as around **1,000** (though for large language models it's much larger!).
 - So, for a single (shortish) sentence, $n \leq 30$; $n^2 \leq \mathbf{900}$.
 - In practice, we set a bound like $n = 512$.
 - **But what if we'd like $n \geq 50,000$?** For example, to work on long documents?

Work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*
- For example, **Linformer** [\[Wang et al., 2020\]](#)

Key idea: map the sequence length dimension to a lower-dimensional space for values, keys



Do we even need to remove the quadratic cost of attention?

- As Transformers grow larger, a larger and larger percent of compute is **outside** the self-attention portion, despite the quadratic cost.
- In practice, **almost no large Transformer language models use anything but the quadratic cost attention we've presented here.**
 - The cheaper methods tend not to work as well at scale.
- So, is there no point in trying to design cheaper alternatives to self-attention?
- Or would we unlock much better models with much longer contexts (>100k tokens?) if we were to do it right?

Do Transformer Modifications Transfer?

- "Surprisingly, we find that most modifications do not meaningfully improve performance."

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
GeLU	223M	11.1T	3.58	2.179 ± 0.003	1.838	75.79	17.86	25.13	26.47
Swish	223M	11.1T	3.62	2.186 ± 0.003	1.847	73.77	17.74	24.34	26.75
ELU	223M	11.1T	3.56	2.270 ± 0.007	1.932	67.83	16.73	23.02	26.08
GLU	223M	11.1T	3.59	2.174 ± 0.003	1.814	74.20	17.42	24.34	27.12
GeGLU	223M	11.1T	3.55	2.130 ± 0.006	1.792	75.96	18.27	24.87	26.87
ReLU	223M	11.1T	3.57	2.145 ± 0.004	1.803	76.17	18.36	24.87	27.02
Selu	223M	11.1T	3.55	2.315 ± 0.004	1.948	69.76	16.76	22.75	25.99
SwiGLU	223M	11.1T	3.53	2.127 ± 0.003	1.789	76.00	18.20	24.34	27.02
LiGLU	223M	11.1T	3.59	2.149 ± 0.005	1.798	75.34	17.97	24.34	26.53
Sigmoid	223M	11.1T	3.63	2.291 ± 0.019	1.887	74.31	17.51	23.02	26.30
Softplus	223M	11.1T	3.47	2.207 ± 0.011	1.850	72.45	17.65	24.34	26.89
RMS Norm	223M	11.1T	3.68	2.167 ± 0.008	1.821	75.45	17.94	24.07	27.14
Resero	223M	11.1T	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
Resero + LayerNorm	223M	11.1T	3.26	2.223 ± 0.006	1.858	70.42	17.58	23.02	26.29
Resero + RMS Norm	223M	11.1T	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fluip	223M	11.1T	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31
24 layers, $d_v = 1536, H = 6$	224M	11.1T	3.33	2.200 ± 0.007	1.843	74.89	17.75	25.13	26.89
18 layers, $d_v = 2048, H = 8$	223M	11.1T	3.38	2.185 ± 0.005	1.831	76.45	16.83	24.34	27.10
8 layers, $d_v = 4096, H = 18$	223M	11.1T	3.69	2.190 ± 0.005	1.847	74.58	17.69	23.28	26.85
6 layers, $d_v = 6144, H = 24$	223M	11.1T	3.70	2.291 ± 0.010	1.857	73.55	17.59	24.60	26.66
Block sharing	65M	11.1T	3.91	2.497 ± 0.037	2.164	64.50	14.53	21.96	25.48
+ Factorized embeddings	45M	9.4T	4.21	2.631 ± 0.305	2.183	60.84	14.00	19.84	25.27
+ Factorized & shared embeddings	20M	9.1T	4.37	2.907 ± 0.313	2.385	53.95	11.37	19.84	25.19
Encoder only block sharing	170M	11.1T	3.68	2.298 ± 0.023	1.929	69.60	16.23	23.02	26.23
Decoder only block sharing	144M	11.1T	3.70	2.352 ± 0.029	2.082	67.93	16.13	23.81	26.08
Factorized Embedding	227M	9.4T	3.80	2.208 ± 0.006	1.855	70.41	15.92	22.75	26.50
Factorized & shared embeddings	202M	9.1T	3.92	2.320 ± 0.010	1.952	68.69	16.33	22.22	26.44
Tied encoder/decoder input embeddings	248M	11.1T	3.55	2.192 ± 0.002	1.840	71.70	17.72	24.34	26.49
Tied decoder input and output embeddings	248M	11.1T	3.57	2.187 ± 0.007	1.827	74.86	17.74	24.87	26.67
Unified embeddings	273M	11.1T	3.53	2.195 ± 0.005	1.834	72.99	17.58	23.28	26.48
Adaptive input embeddings	204M	9.2T	3.55	2.250 ± 0.002	1.899	66.57	16.21	24.07	26.66
Adaptive softmax	204M	9.2T	3.60	2.364 ± 0.005	1.982	72.91	16.67	21.16	25.56
Adaptive softmax without projection	223M	10.8T	3.43	2.229 ± 0.009	1.914	71.82	17.10	23.02	25.72
Mixture of softmaxes	232M	16.3T	2.24	2.227 ± 0.017	1.821	76.77	17.62	22.75	26.82
Transparent attention	223M	11.1T	3.33	2.181 ± 0.014	1.874	54.31	10.40	21.16	26.80
Dynamic convolution	257M	11.8T	2.65	2.403 ± 0.009	2.047	58.30	12.67	21.16	17.03
Lightweight convolution	224M	10.4T	4.07	2.370 ± 0.010	1.989	63.07	14.86	23.02	24.73
Evolved Transformer	217M	9.9T	3.09	2.220 ± 0.003	1.863	73.67	10.76	24.07	26.58
Synthesizer (dense)	224M	11.4T	3.47	2.334 ± 0.021	1.962	61.03	14.27	16.14	26.63
Synthesizer (dense plus)	243M	12.6T	3.22	2.191 ± 0.010	1.840	73.98	16.96	23.81	26.71
Synthesizer (dense plus alpha)	243M	12.6T	3.01	2.180 ± 0.007	1.828	74.25	17.02	23.28	26.61
Synthesizer (factorized)	207M	10.1T	3.94	2.341 ± 0.017	1.968	62.78	15.39	23.55	26.42
Synthesizer (random)	254M	10.1T	4.08	2.326 ± 0.012	2.009	54.27	10.35	19.56	26.44
Synthesizer (random plus)	292M	12.0T	3.63	2.189 ± 0.004	1.842	73.32	17.04	24.87	26.43
Synthesizer (random plus alpha)	292M	12.0T	3.42	2.186 ± 0.007	1.828	75.24	17.08	24.08	26.39
Universal Transformer	84M	40.0T	0.88	2.406 ± 0.036	2.053	70.13	14.09	19.05	23.91
Mixture of experts	648M	11.7T	3.20	2.148 ± 0.006	1.785	74.55	18.13	24.08	26.94
Switch Transformer	1100M	11.7T	3.18	2.135 ± 0.007	1.788	75.38	18.02	26.19	26.81
Funnel Transformer	223M	1.9T	4.30	2.288 ± 0.008	1.918	67.34	16.26	22.75	23.20
Weighted Transformer	280M	71.0T	0.59	2.378 ± 0.021	1.989	69.04	16.98	23.02	26.30
Product key memory	421M	386.6T	0.25	2.155 ± 0.003	1.798	75.16	17.04	23.55	26.73

Do Transformer Modifications Transfer Across Implementations and Applications?

Sharan Narang* Hyung Won Chung Yi Tay William Fedus
 Thibault Fevry† Michael Matena† Karishma Malkan† Noah Fiedel
 Noam Shazeer Zhenzhong Lan† Yanqi Zhou Wei Li
 Nan Ding Jake Marcus Adam Roberts Colin Raffel†

Parting remarks

- Pretraining on Tuesday!
- Good luck on assignment 4!
- Remember to work on your project proposal!