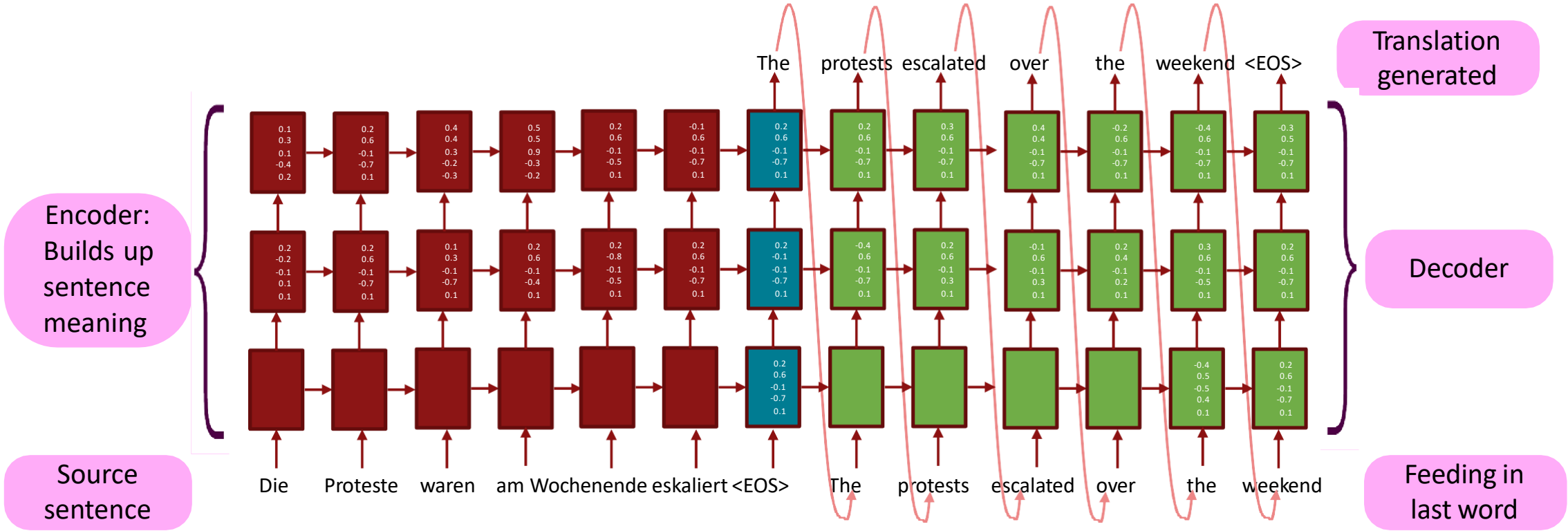
FROM RNN TO TRANSFORMERS

Slide credict: CS224n: Natural Language Processing with Deep Learning, Yen-lung Tsai, and Hung-Yi Lee

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



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**

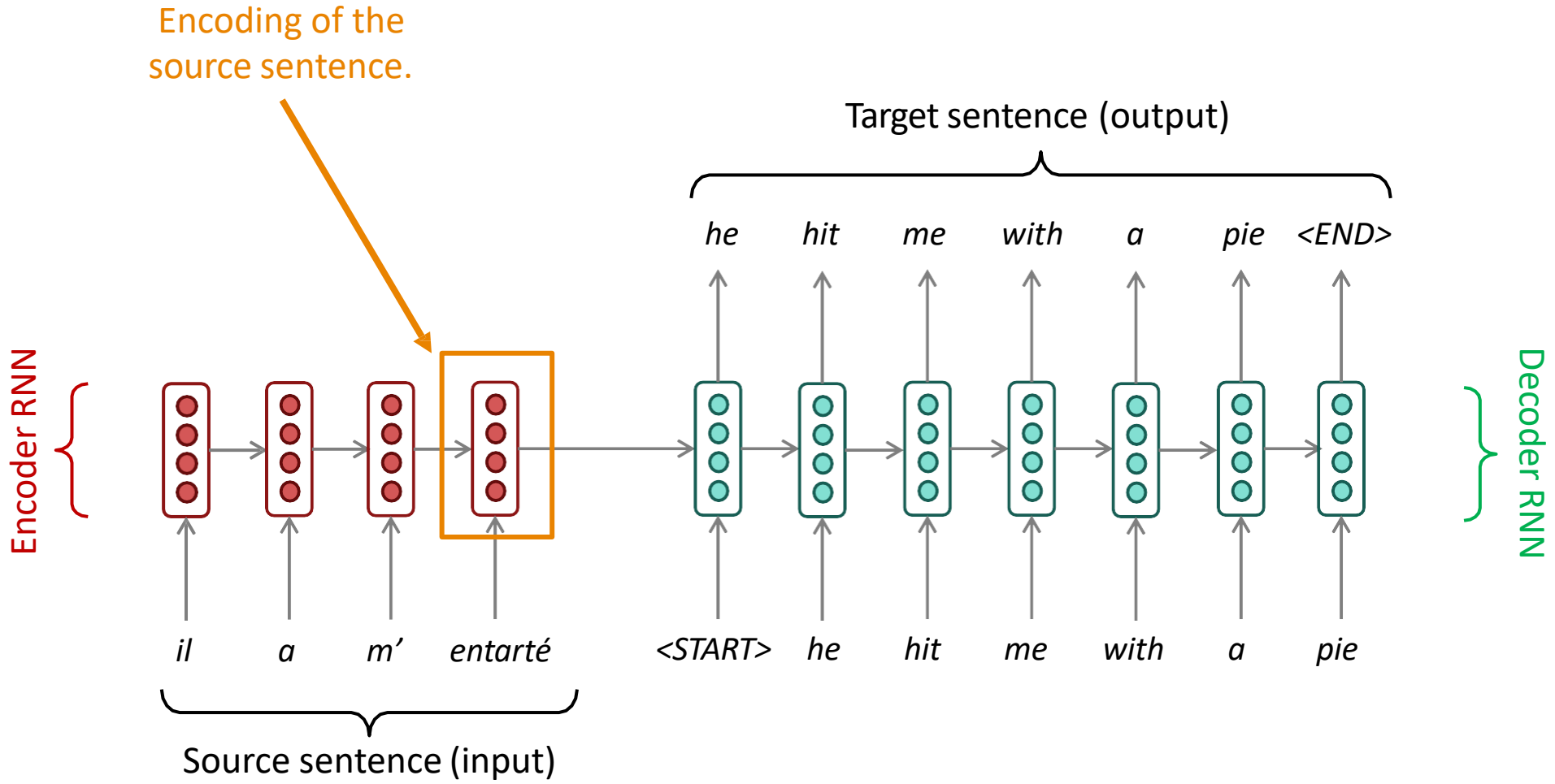
- **2014**: First seq2seq paper published [Sutskever et al. 2014]
- **2016**: Google Translate switches from SMT to NMT – and by 2018 everyone has

Microsoft SYSTRAN beyond language Google facebook

Bai du 百度 網易 NetEase www.163.com Tencent 腾讯 搜狗搜索

- 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

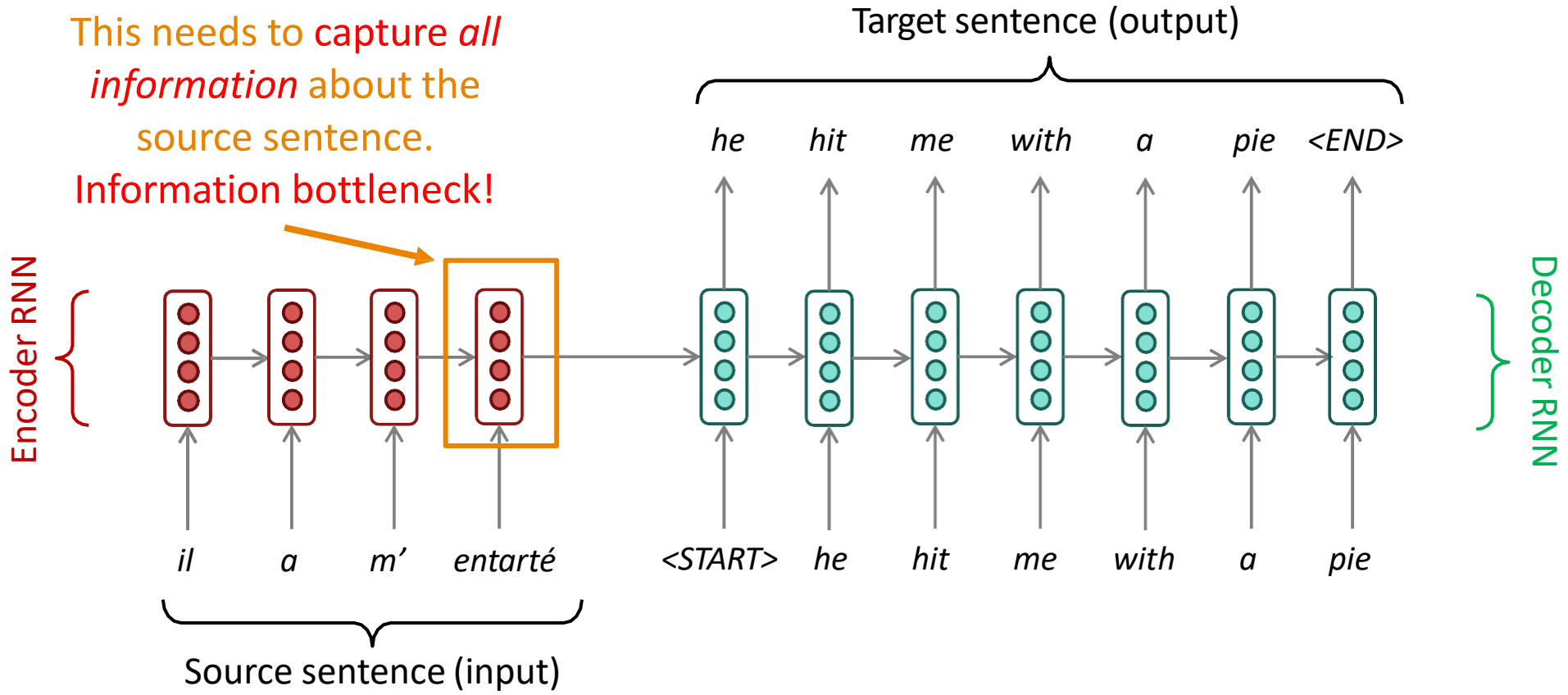
The final piece: the bottleneck problem in RNNs



Problems with this architecture?

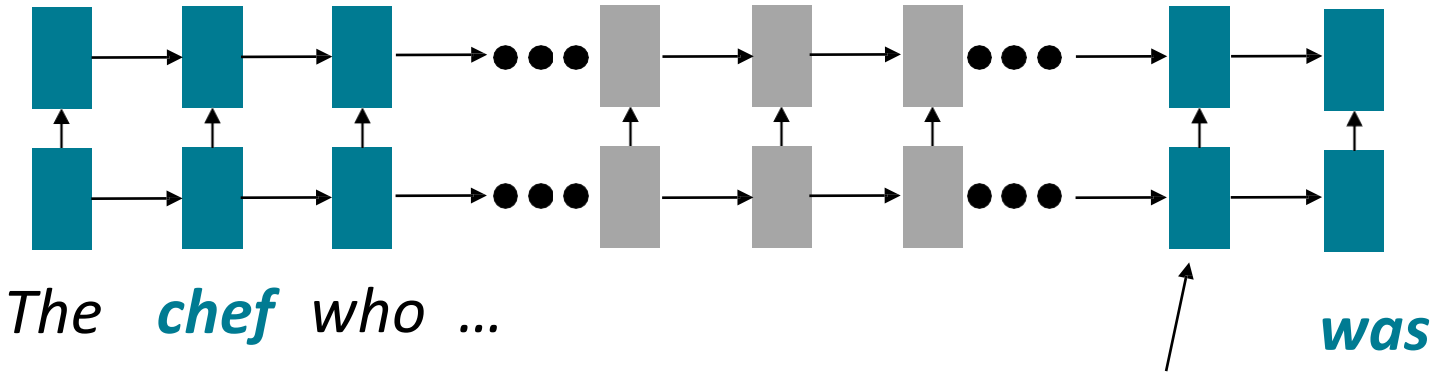
1. Why attention? Sequence-to-sequence: the bottleneck problem

Encoding of the source sentence. This needs to capture *all information* about the source sentence. Information bottleneck!



Issues with recurrent models: Linear interaction distance

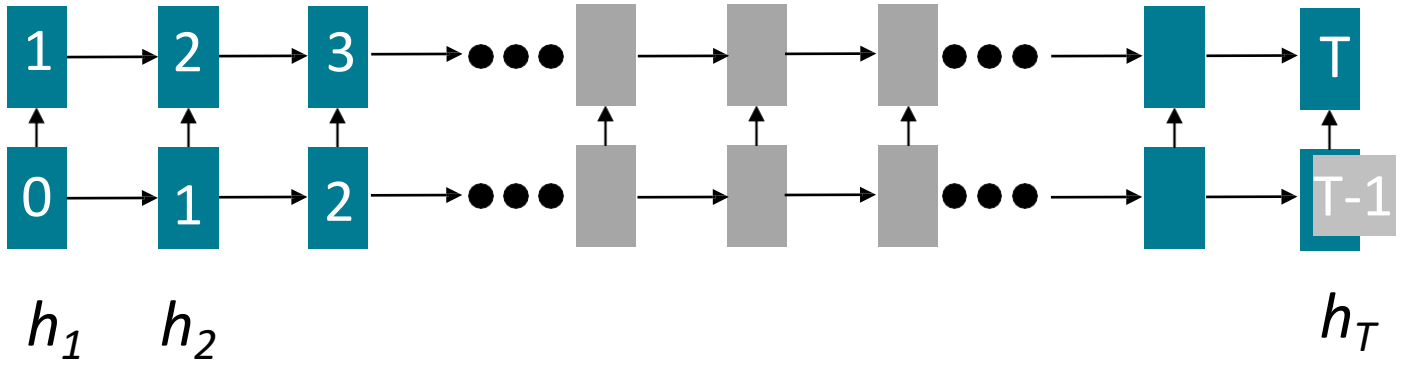
- **O(sequence length)** steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is “baked in”; we already know linear order isn’t the right way to think about sentences…



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

Issues with recurrent models: Lack of parallelizability

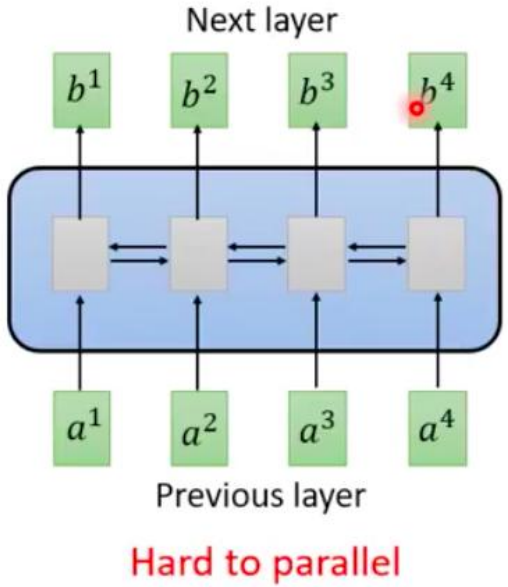
- Forward and backward passes have **O(sequence length)** unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!

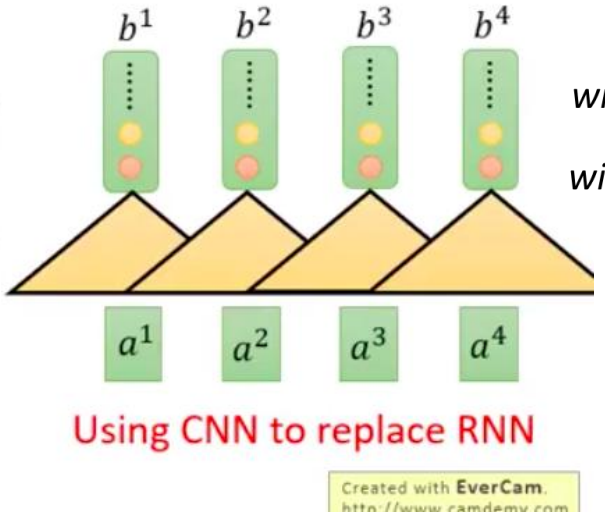


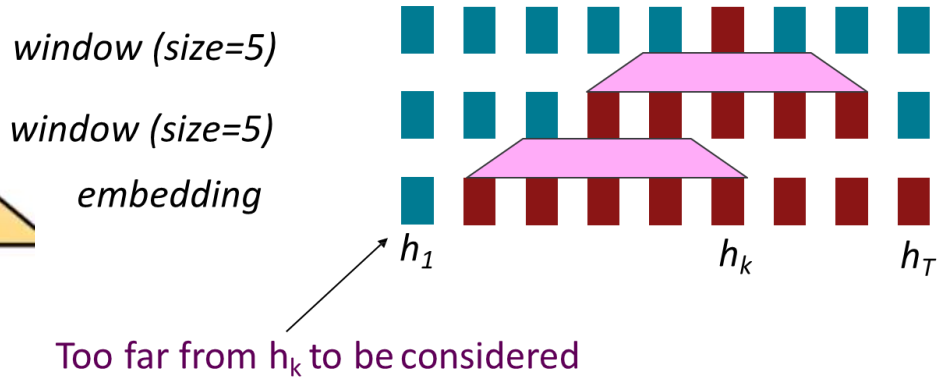
Numbers indicate min # of steps before a state can be computed

If not recurrence, then what? How about word windows?

- Word window models aggregate local contexts
- What about long-distance dependencies?
  - Stacking word window layers allows interaction between farther words
  - Hard to very long distance context







Red states indicate those "visible" to h_k

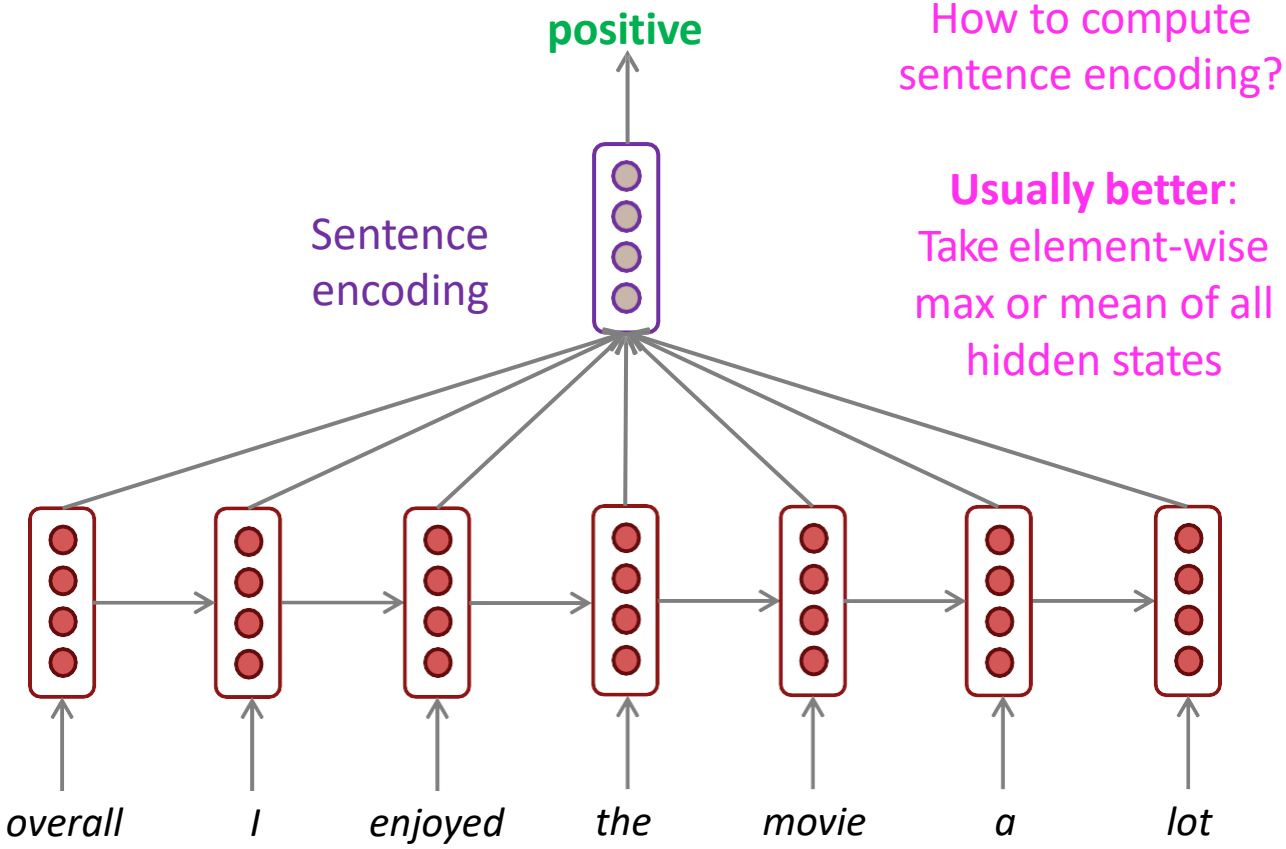
Attention

- **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

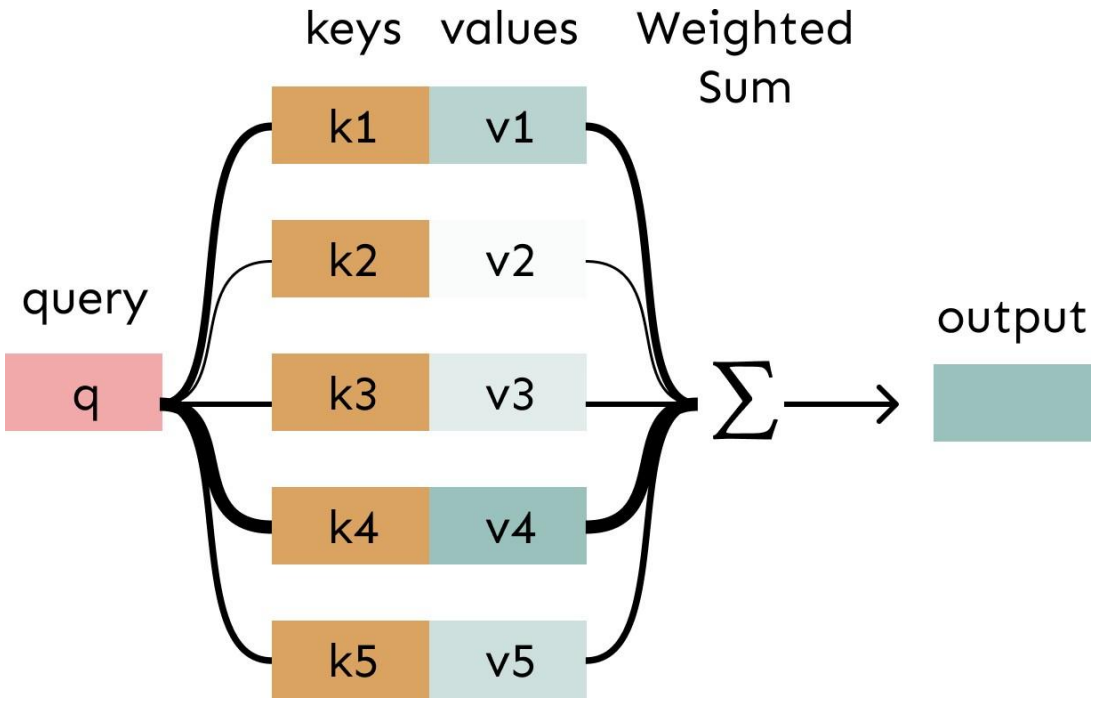


- Starting point: a *very* basic way of 'passing information from the encoder' is to *average*

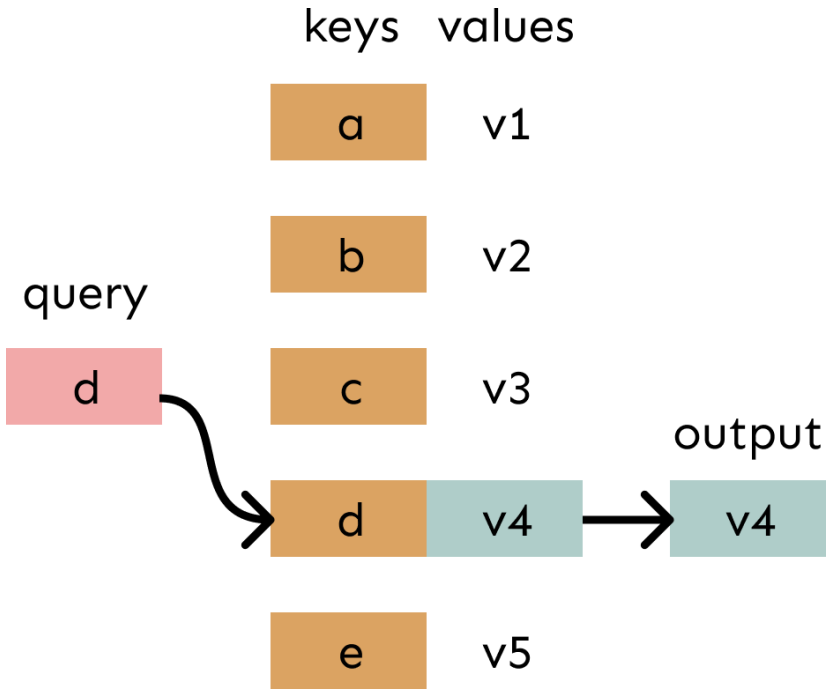
Attention is weighted averaging, which lets you do lookups!

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.

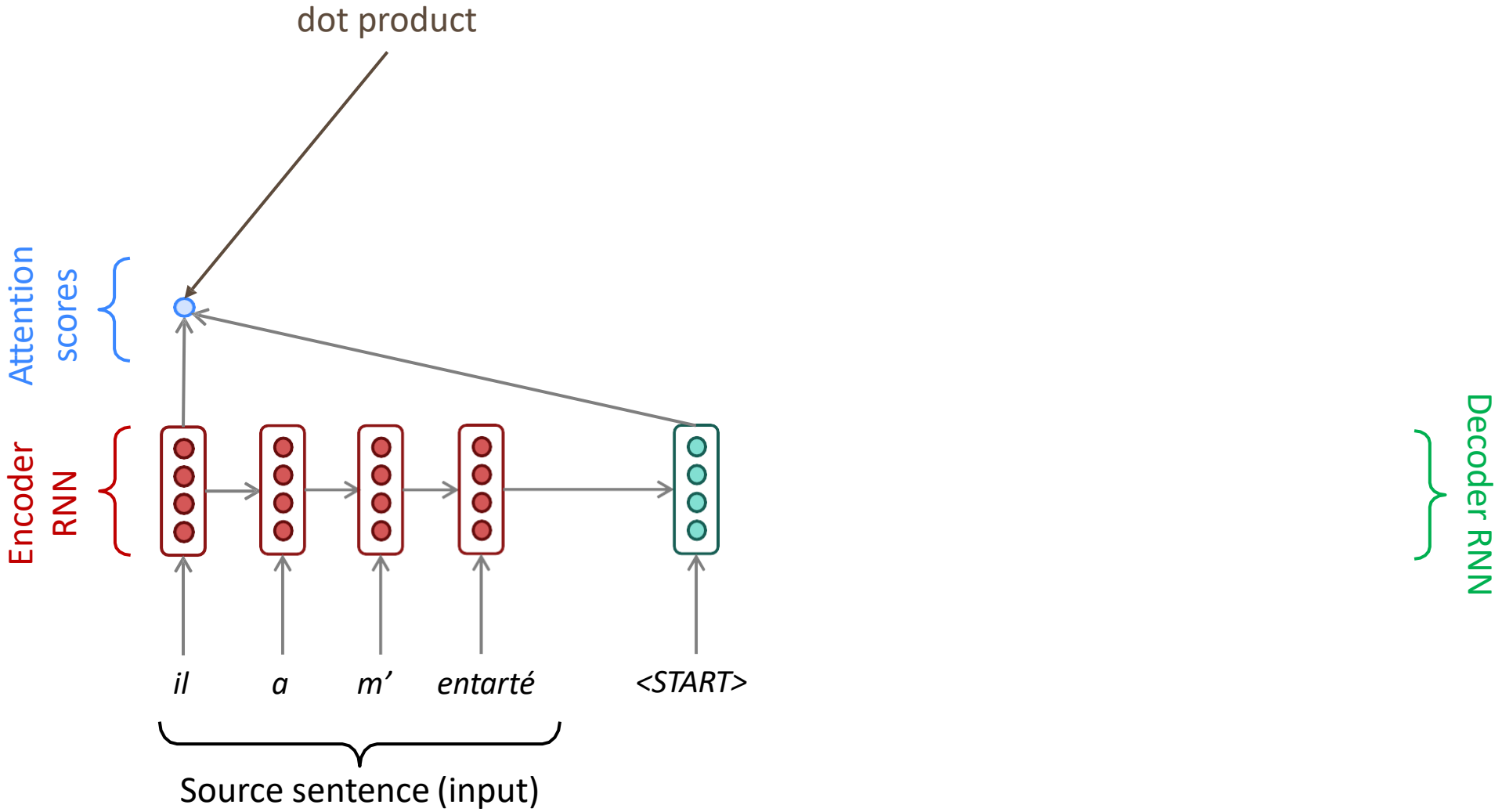


In a **lookup table**, we have a table of keys that map to values. The query matches one of the keys, returning its value.

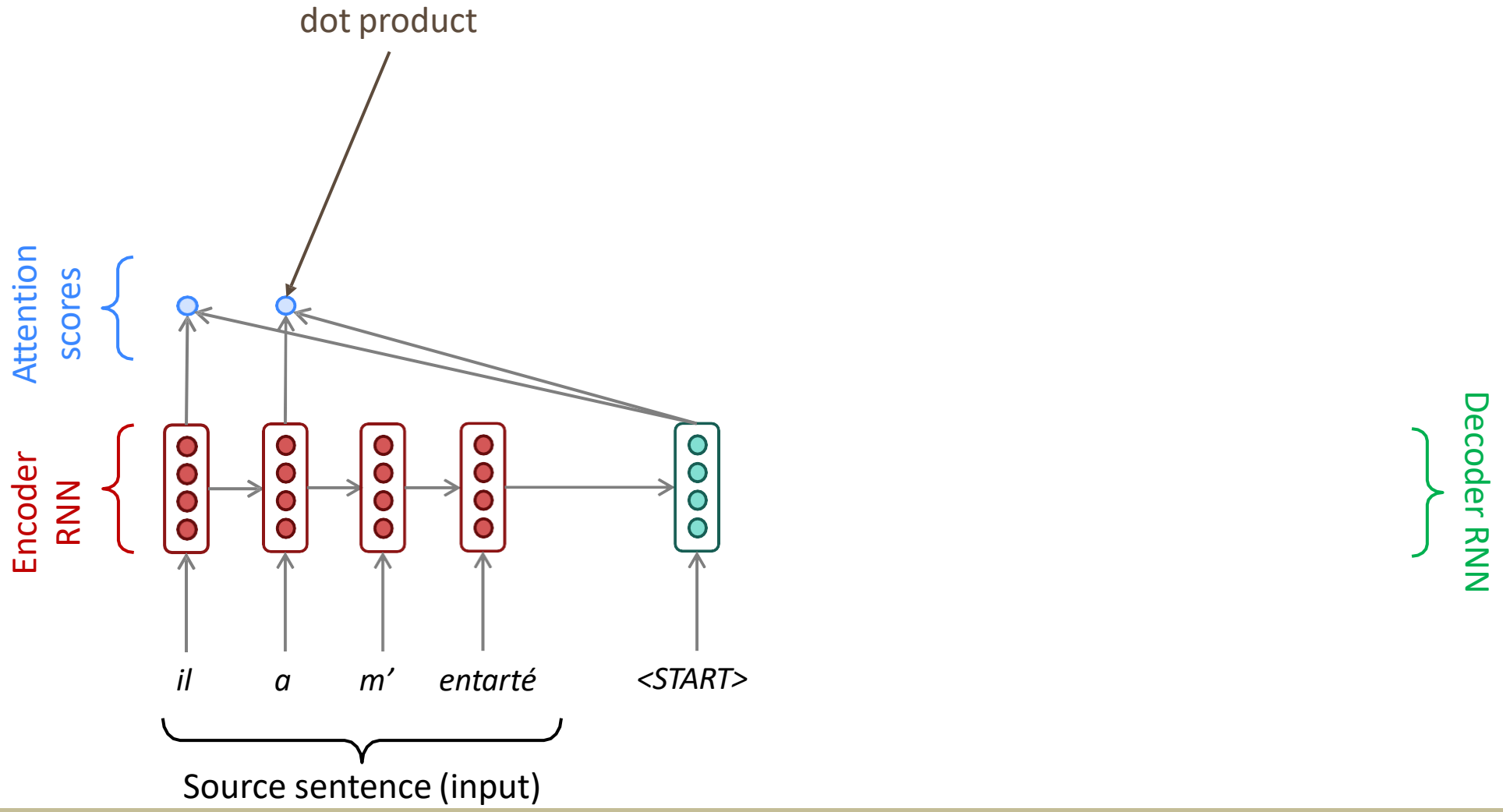


Sequence-to-sequence with attention

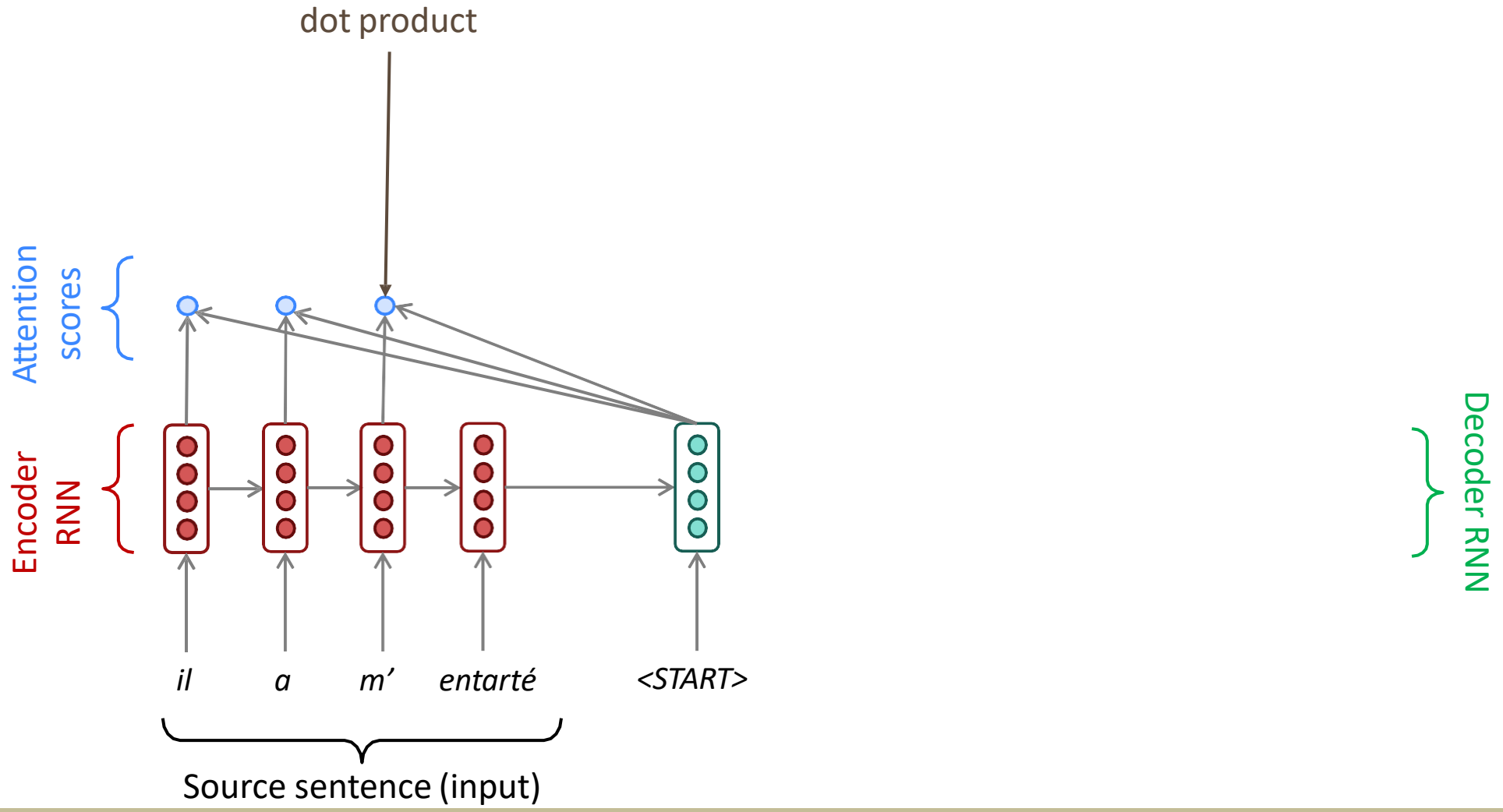
Core idea: on each step of the decoder, use *direct connection to the encoder* to *focus on a particular part* of the source sequence



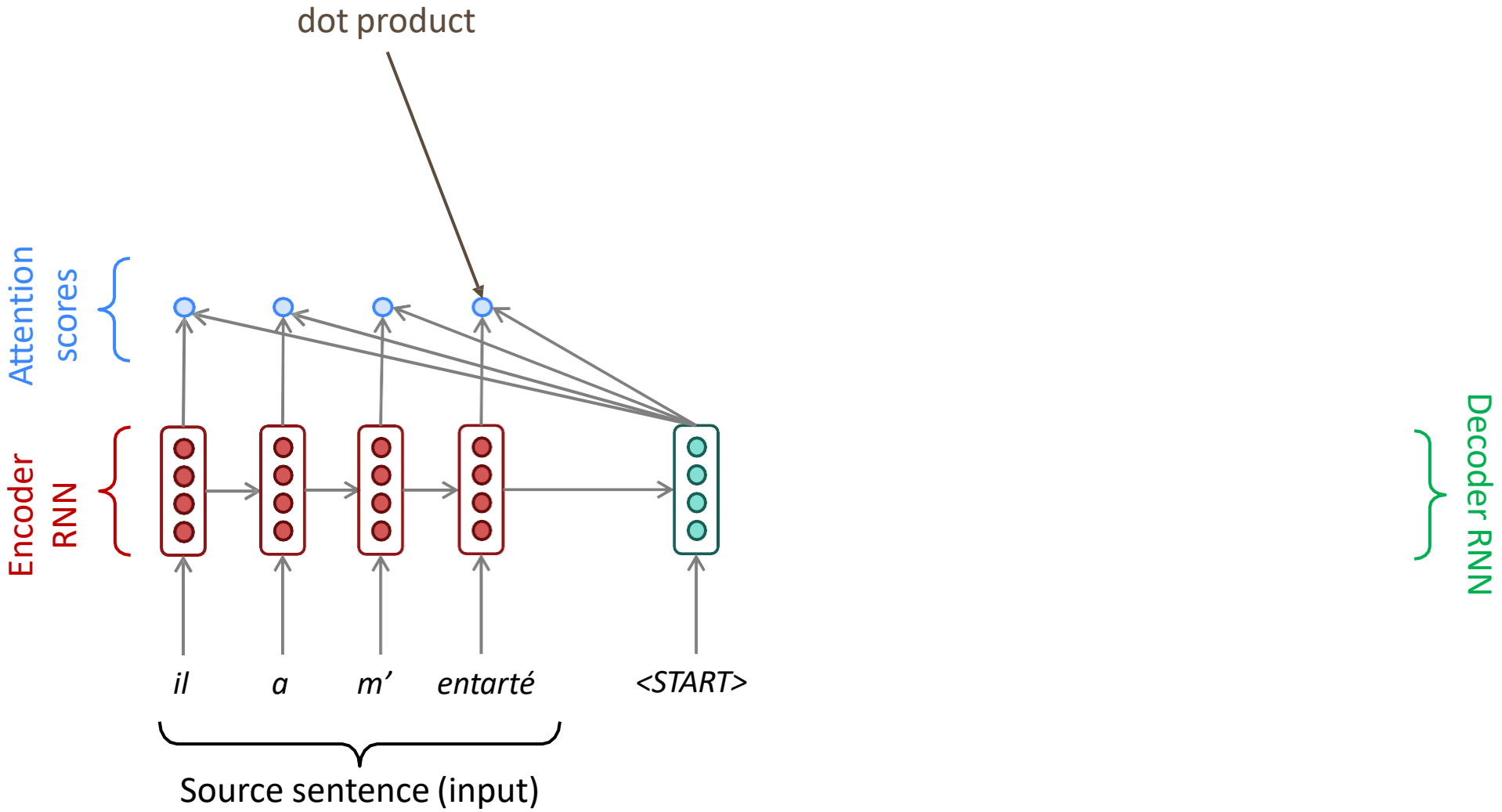
Sequence-to-sequence with attention



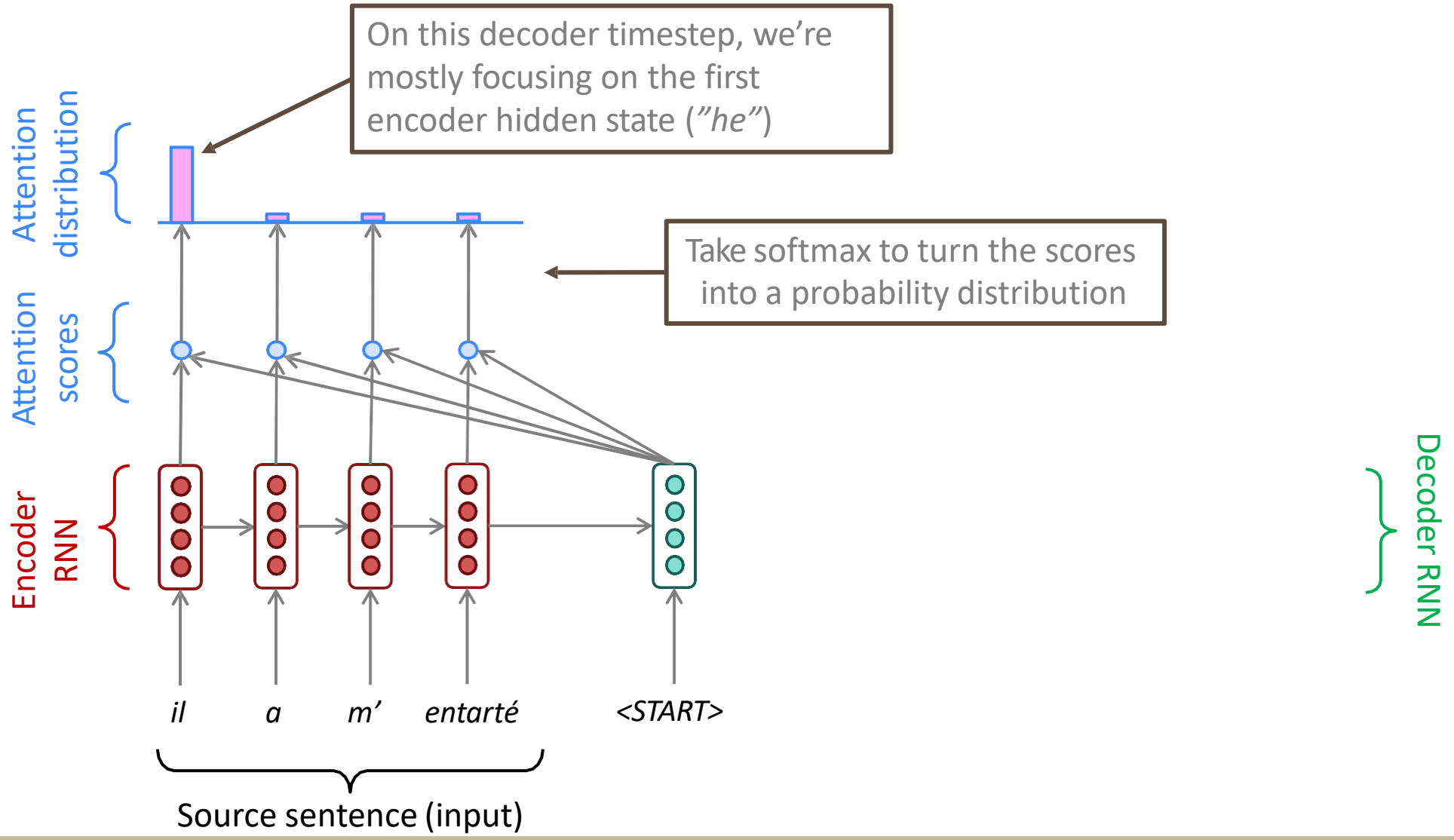
Sequence-to-sequence with attention



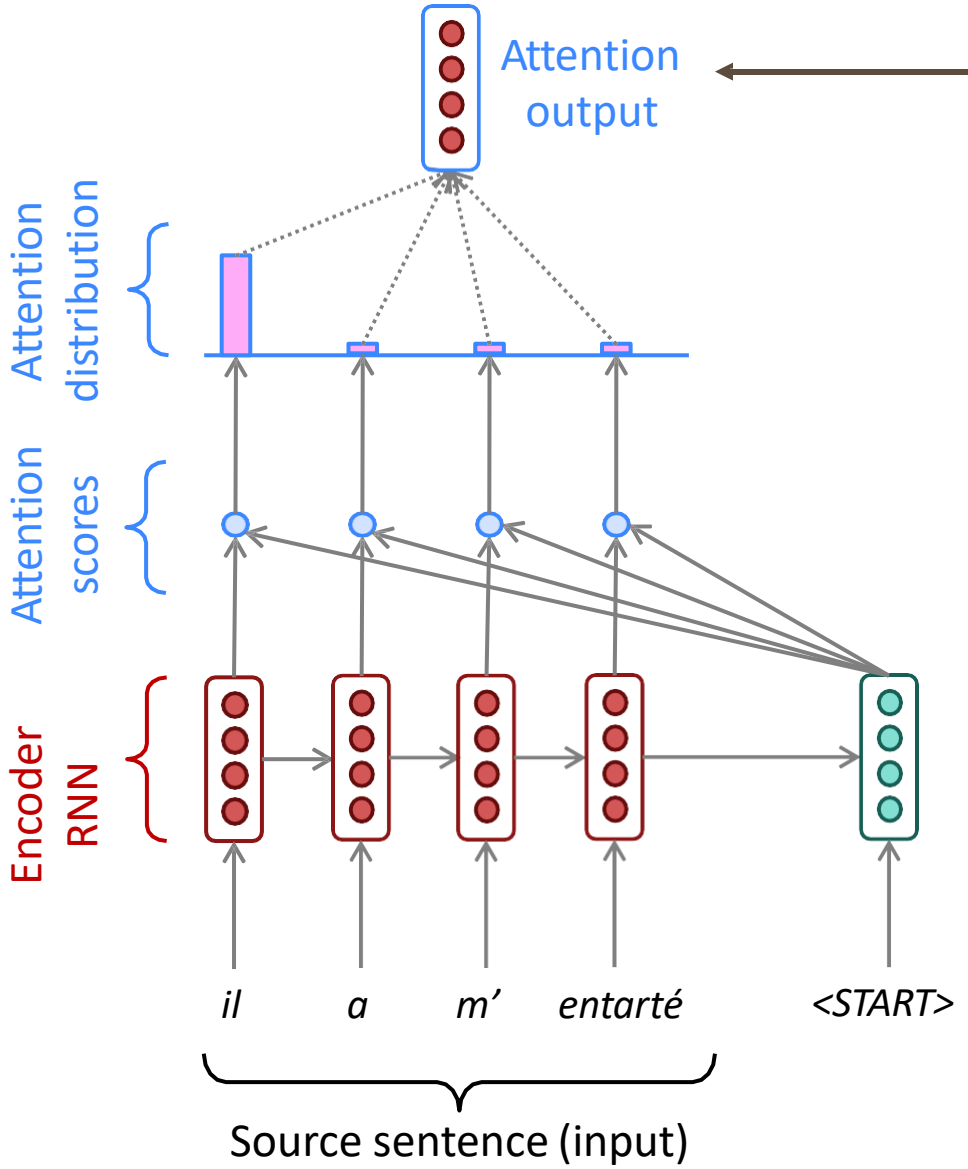
Sequence-to-sequence with attention



Sequence-to-sequence with attention



Sequence-to-sequence with attention

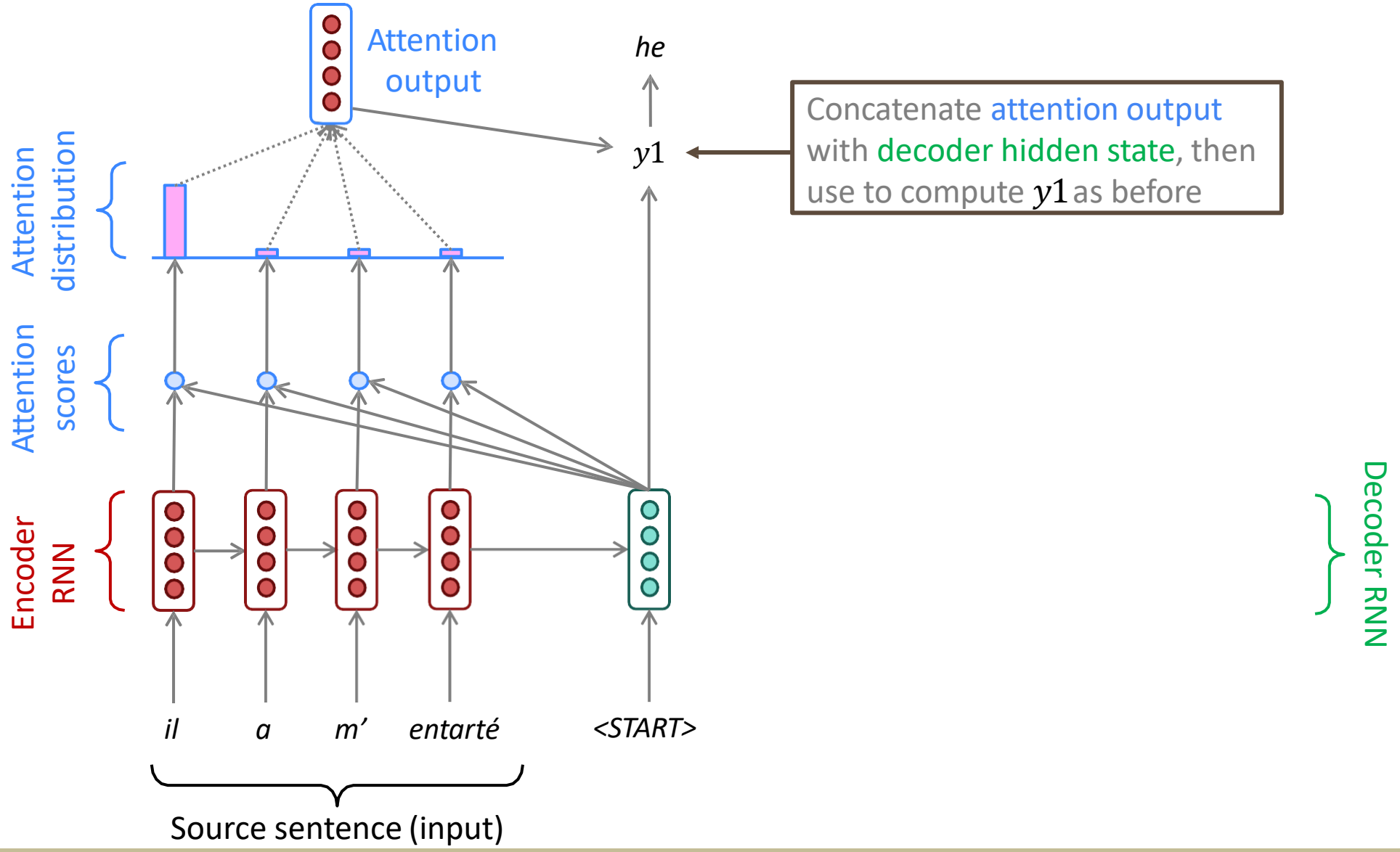


Use the attention distribution to take a **weighted sum** of the encoder hidden states.

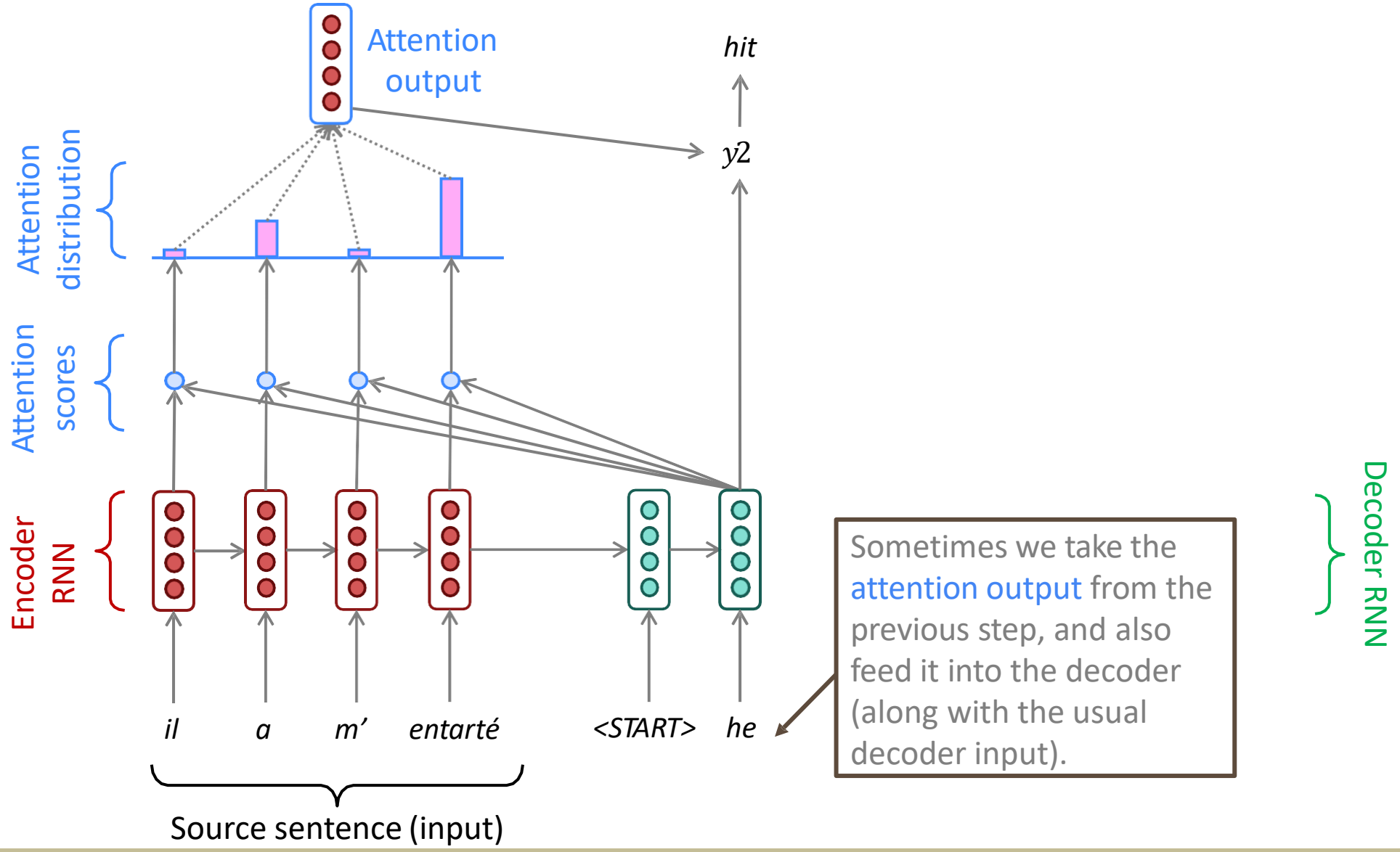
The attention output mostly contains information from the hidden states that received high attention.

Decoder RNN

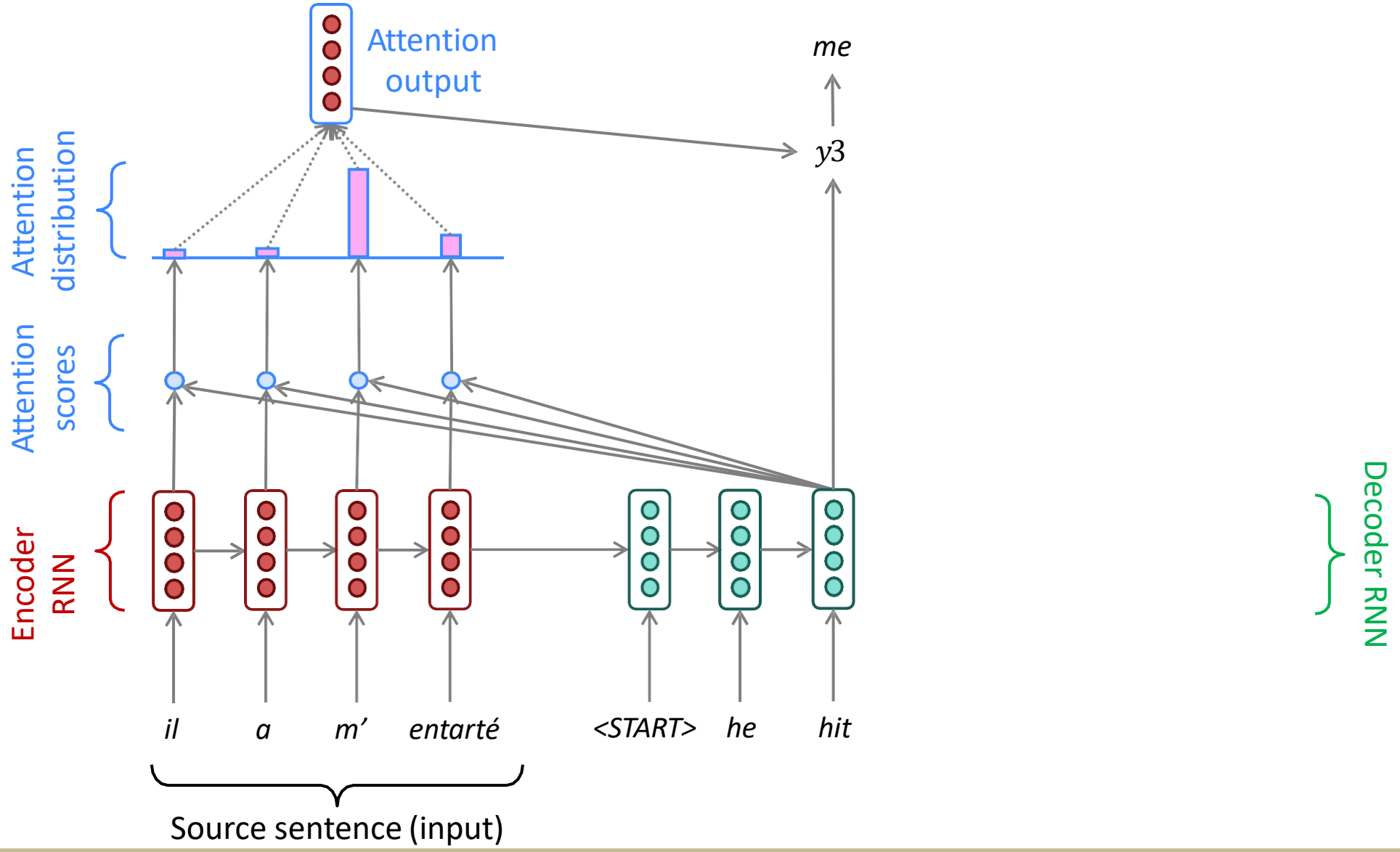
Sequence-to-sequence with attention



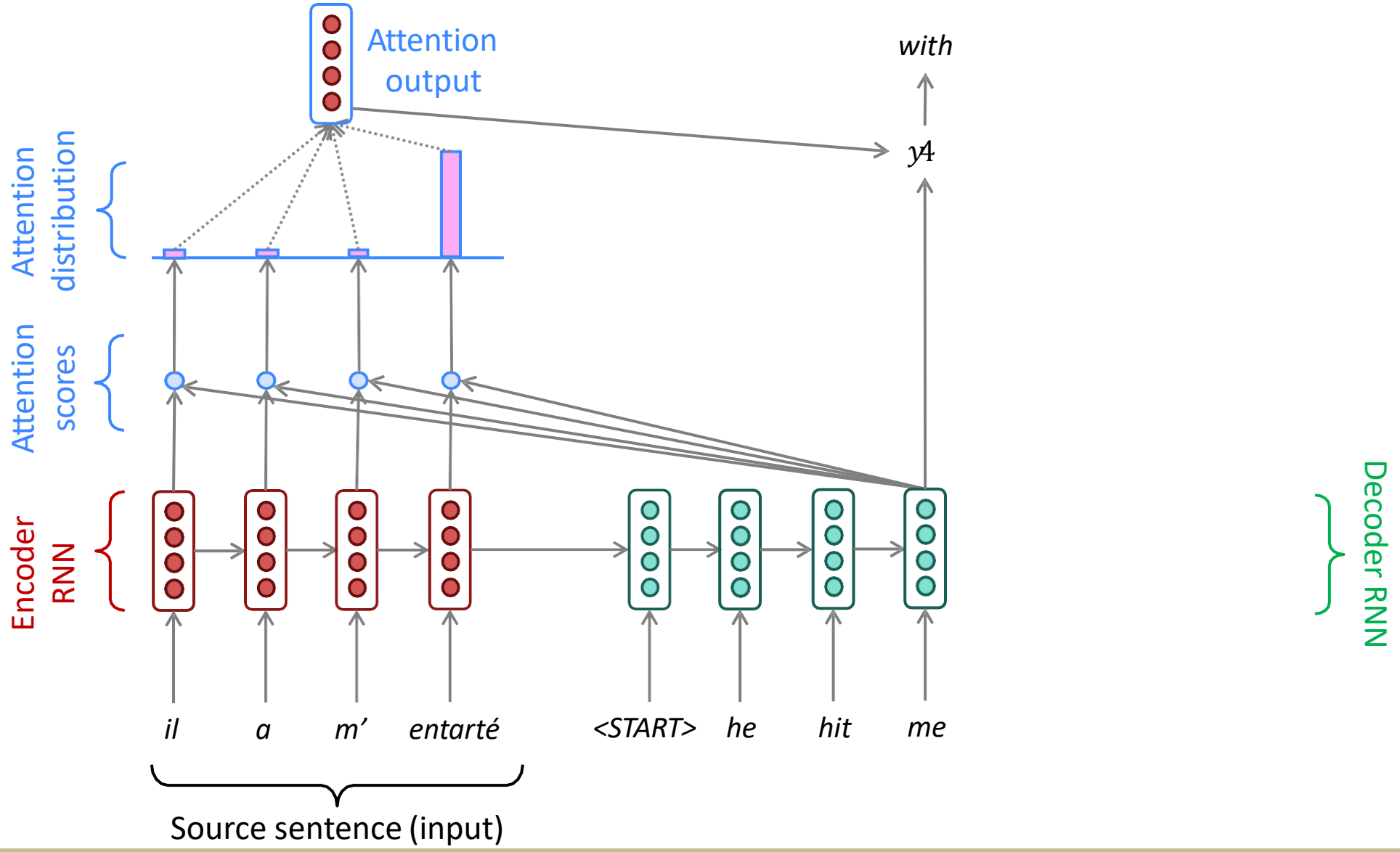
Sequence-to-sequence with attention



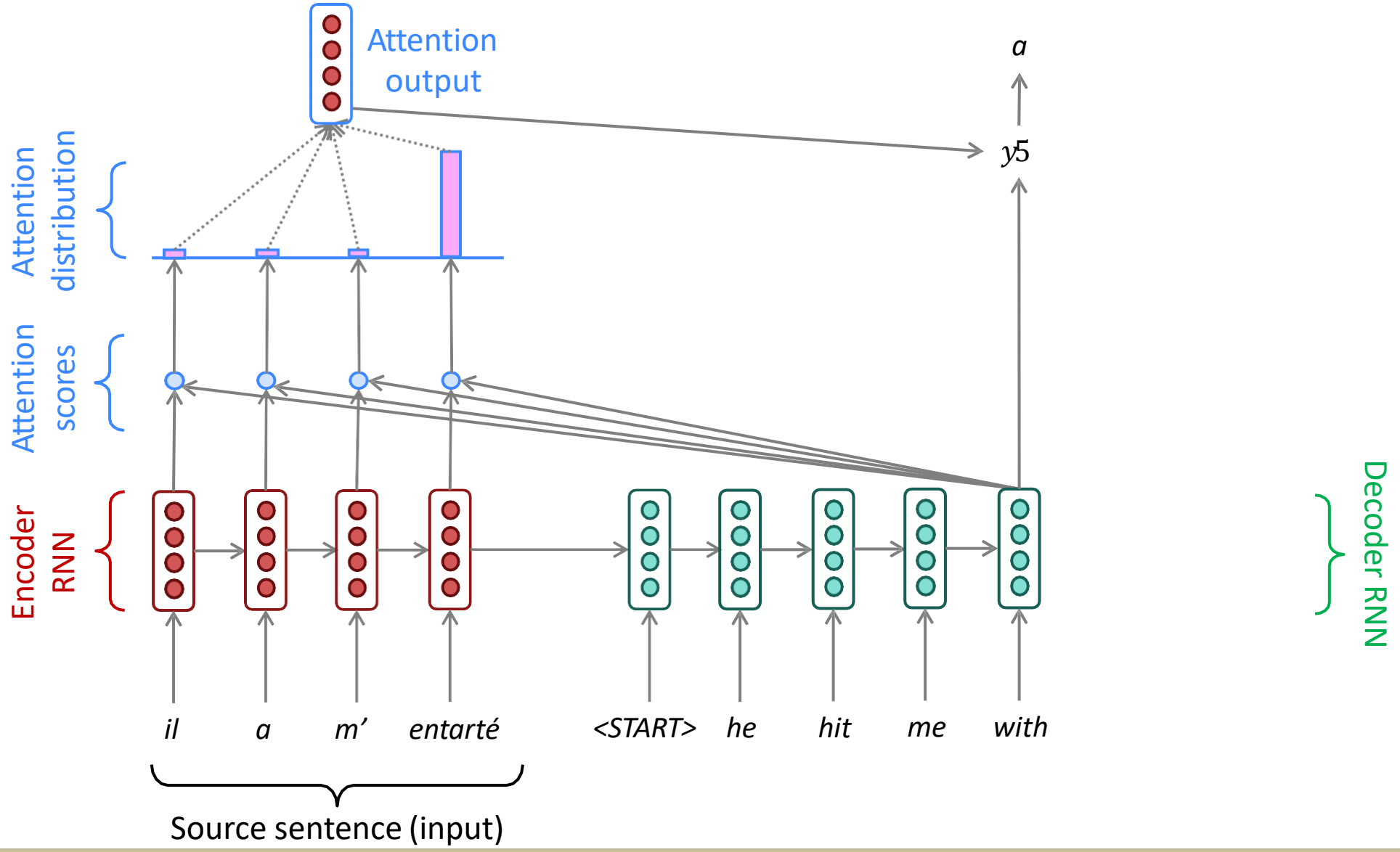
Sequence-to-sequence with attention



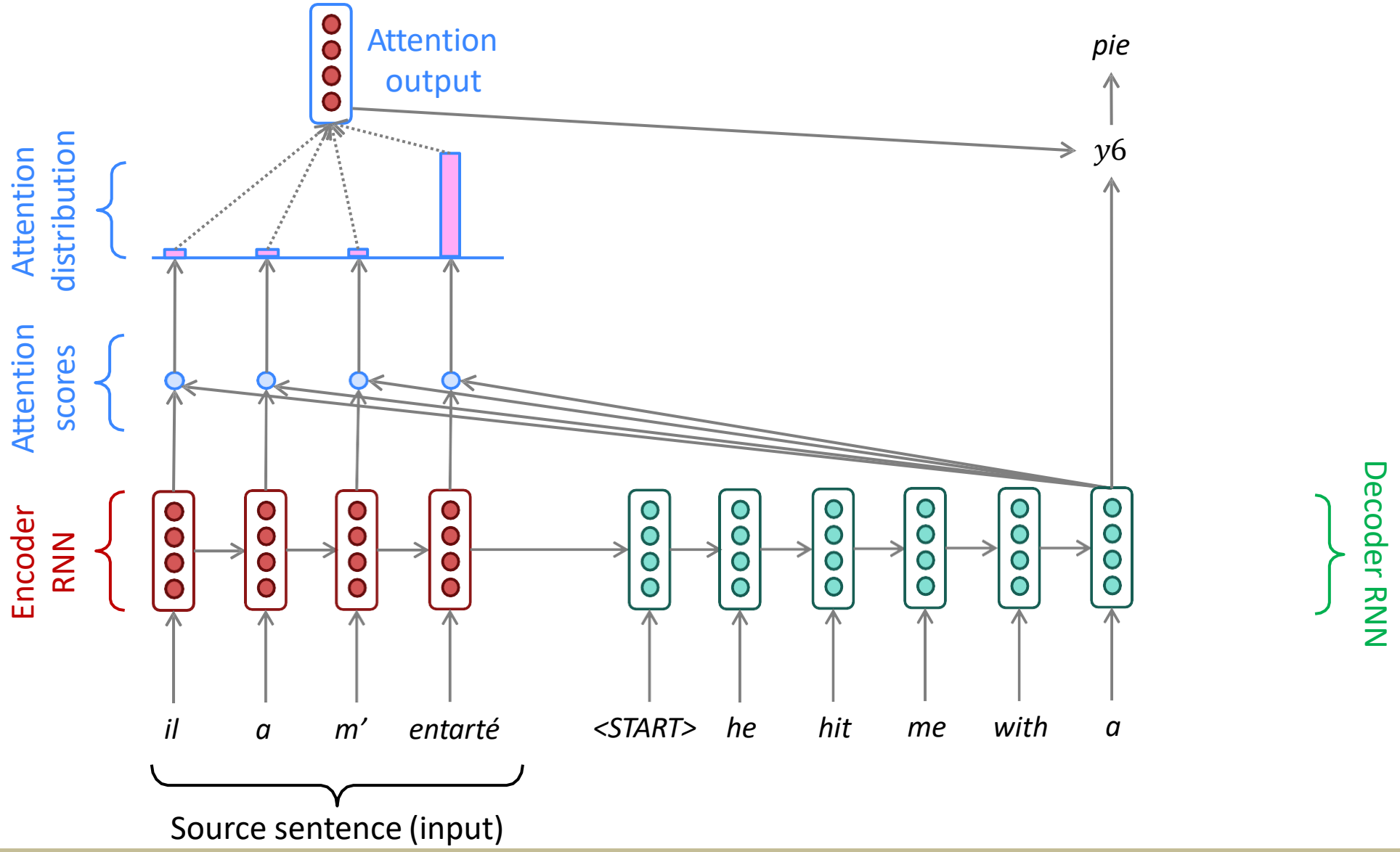
Sequence-to-sequence with attention



Sequence-to-sequence with attention



Sequence-to-sequence with attention



Attention: in equations

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

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

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

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

- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

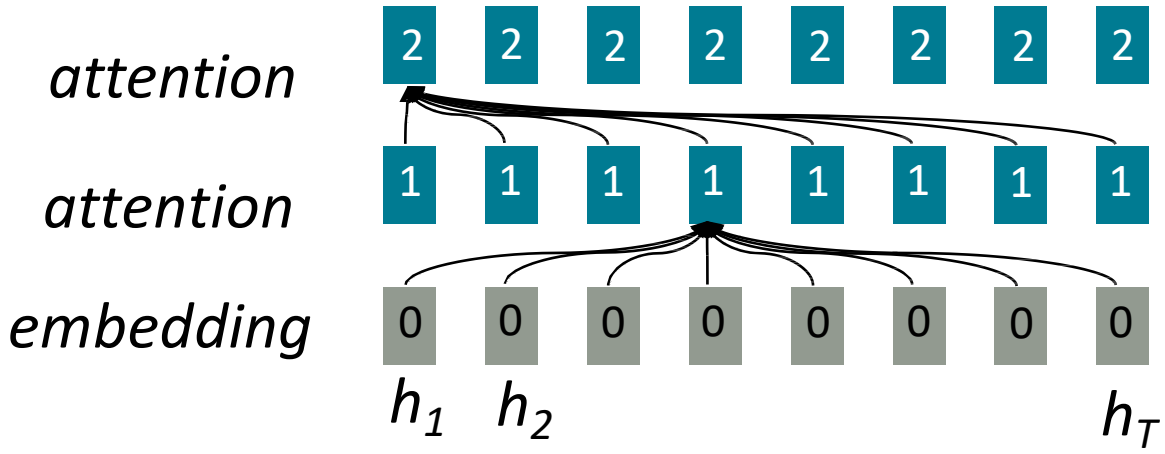
$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

Attention is Parallelizable and Solves Bottleneck Issues

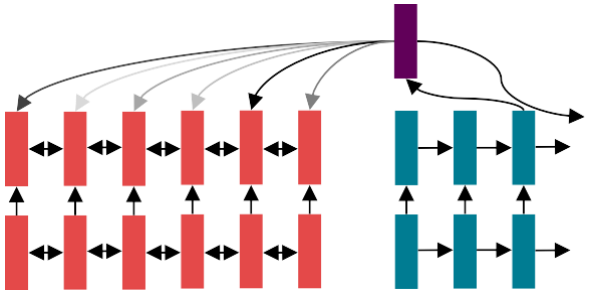
- **Attention** treats each word's representation as a **query** to access and incorporate information from **a set of values.**
  - We saw attention from the decoder to the encoder; today we'll think about attention within a single sentence
- Number of unparallelizable operations does not increase with 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

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!



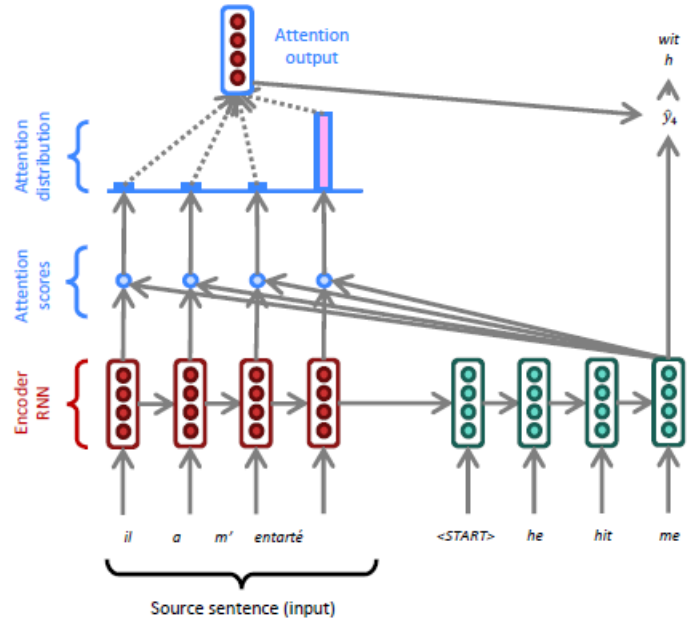
2014-2017ish
Recurrence

Lots of trial
and error

2021
??????

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

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

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

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

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

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

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

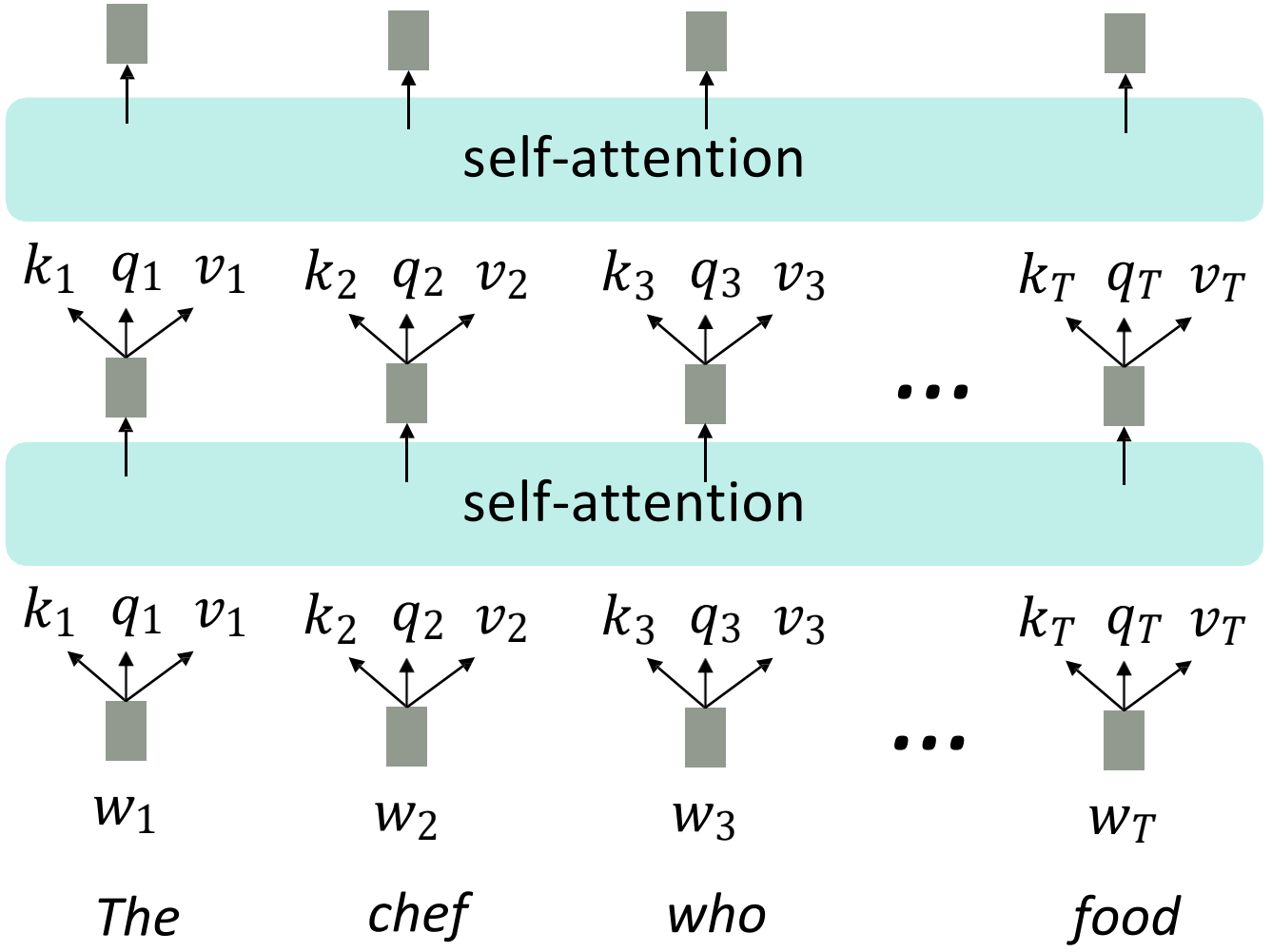
3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \boldsymbol{v}_i$$

FROM ATTENTION TO SELF ATTENTION

Self-attention as an NLP building block

- In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.
- Can self-attention be a drop-in replacement for recurrence?
- No. It has a few issues, which we'll go through.
- First, self-attention is an operation on sets. It has no inherent notion of order.



Self-attention doesn't know the order of its inputs.

Barriers and solutions for Self-Attention as a building block

Barriers **Solutions**

- Doesn't have an inherent notion of order!

- need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$$p_i \in \mathbb{R}^d, \text{ for } i \in \{1,2, \dots, T\} \text{ are position vectors}$$

- 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 p_i to our inputs!

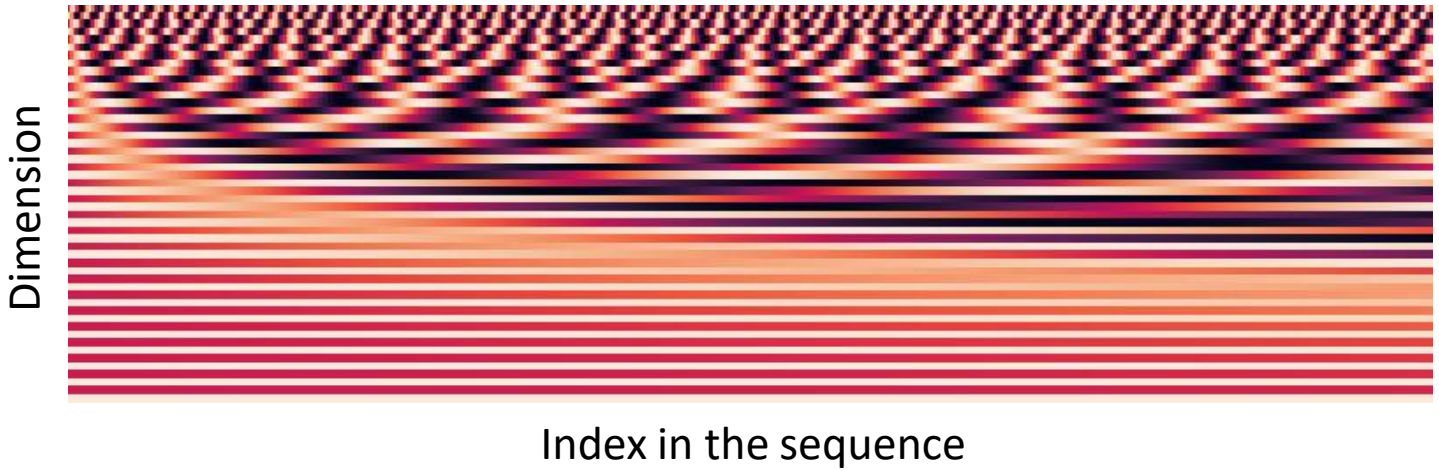
$$v_i = \tilde{v}_i + p_i$$
$$q_i = \tilde{q}_i + p_i$$
$$k_i = \tilde{k}_i + p_i$$

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

Fixing the first self-attention problem: sequence order

- **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!

Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Position representation vectors learned from scratch

- **Learned absolute position representations:** Let all p_i be learnable parameters!
  Learn a matrix $\boldsymbol{p} \in \mathbb{R}^{\&\times\#}$, *and let each* $\boldsymbol{p}_i$ *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 $1, \ldots, n$.
- 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

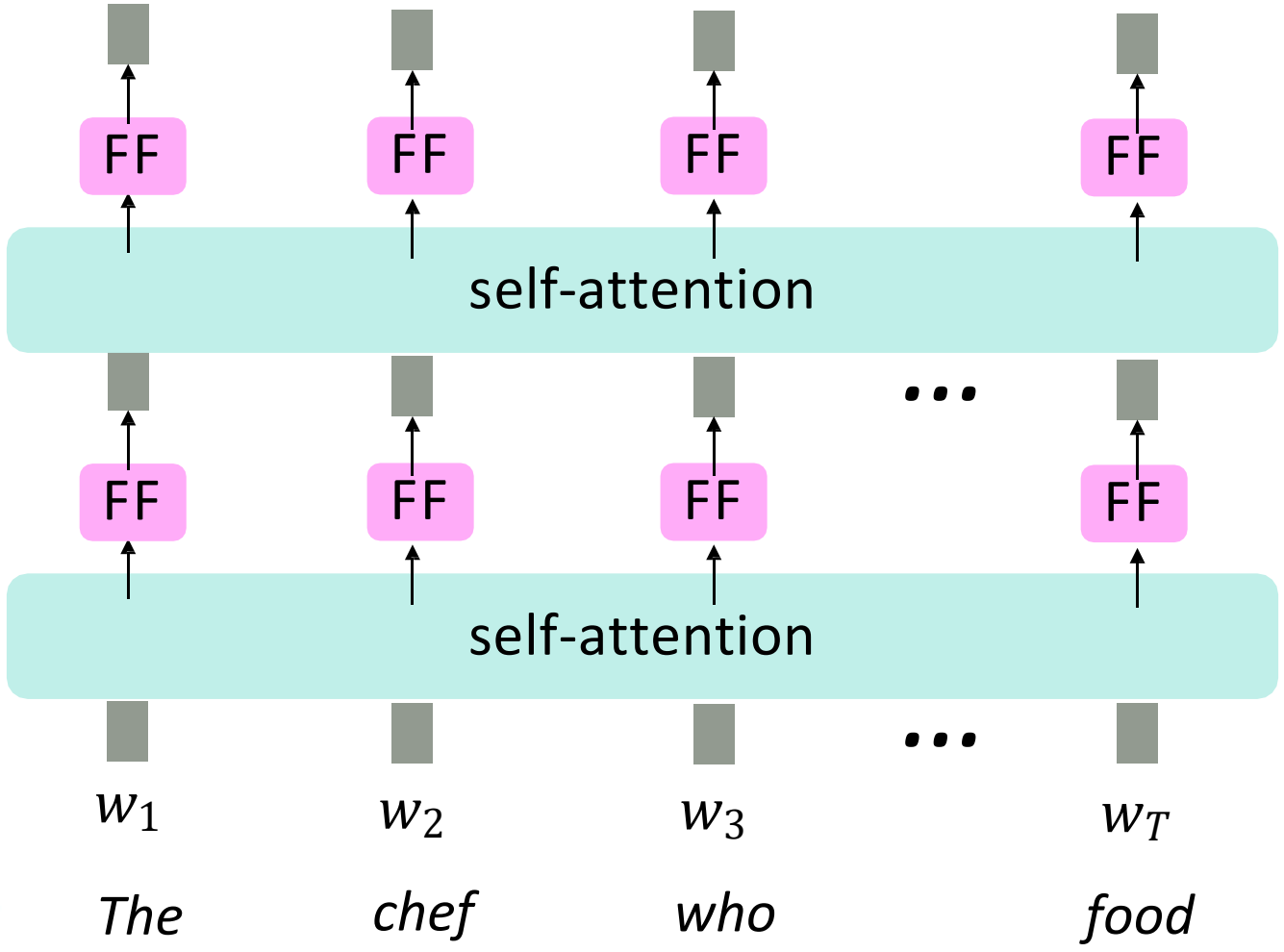
Barriers	Solutions
• Doesn't have an inherent notion of order!	• Add position representations to the inputs
• No nonlinearities for deep learning! It's all just weighted averages	

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

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

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



Intuition: the FF network processes the result of attention

Barriers and solutions for Self-Attention as a building block

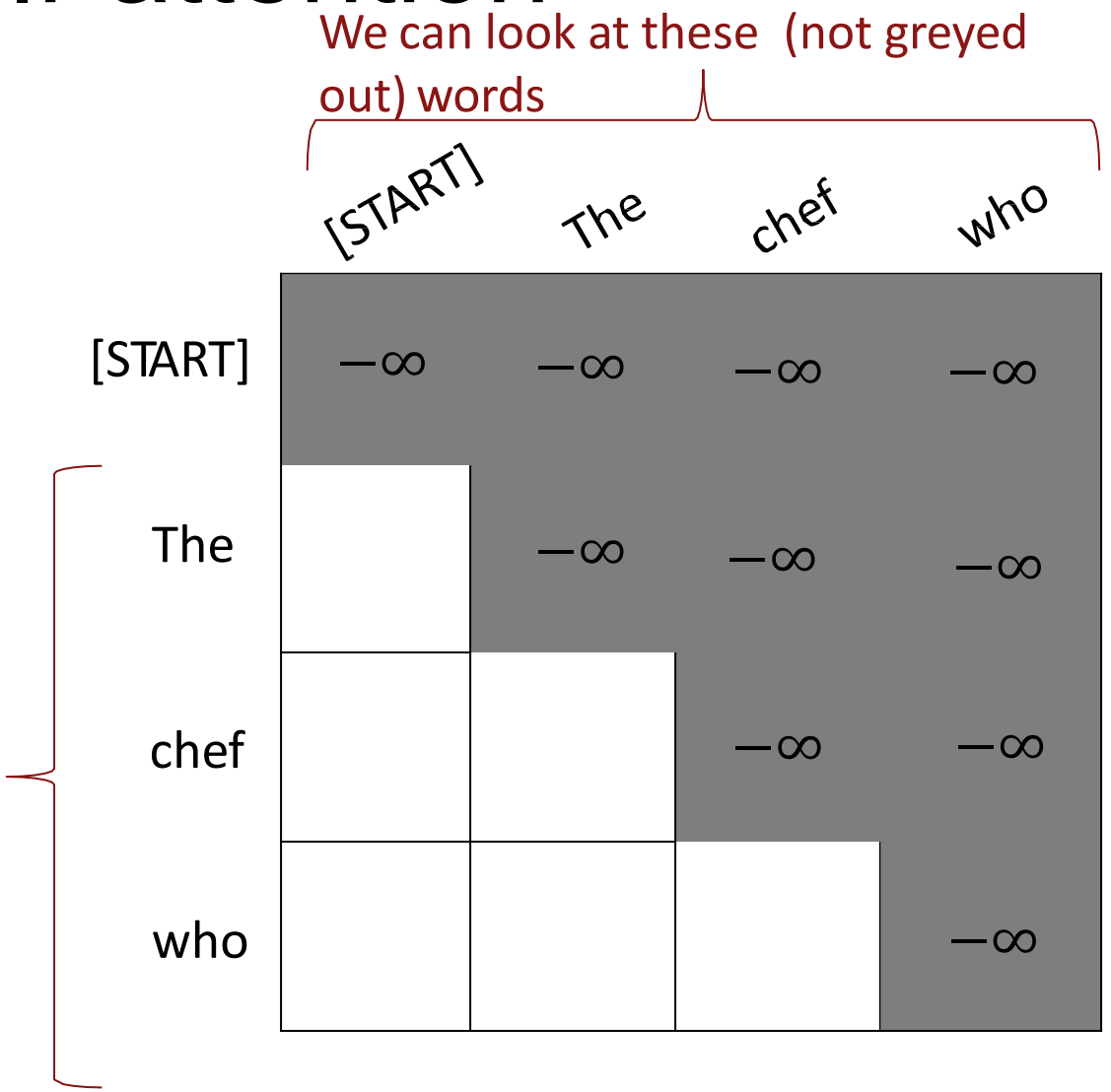
Barriers	Solutions
• Doesn't have an inherent notion of order!	• Add position representations to the inputs
• No nonlinearities for deep learning magic! It's all just weighted averages	• Easy fix: apply the same feedforward network to each self-attention output.

• Need to ensure we don't "look at the future" when predicting a sequence
  • Like in machine translation
  • Or language modeling	

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 $-\infty$.

$$e_{ij} = \begin{cases} q_i^\top k_j, j < i \\ -\infty, j \geq i \end{cases}$$



Barriers and solutions for Self-Attention as a building block

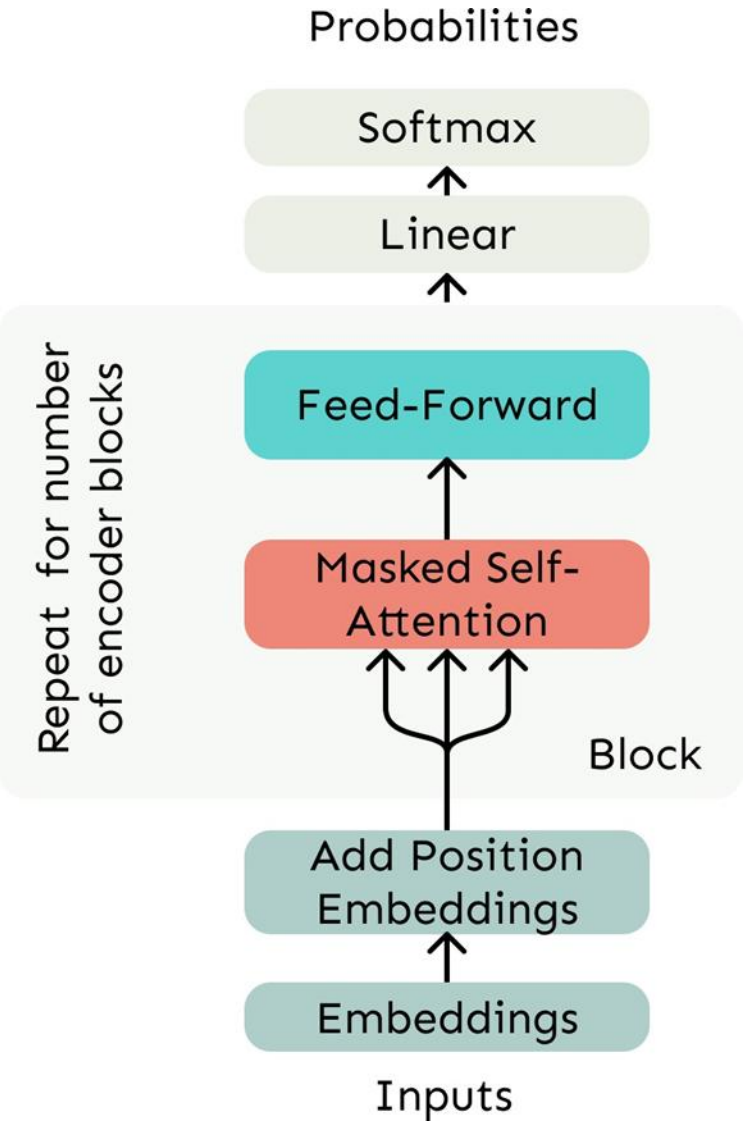
Barriers	Solutions
• Doesn't have an inherent notion of order!	• Add position representations to the inputs
• No nonlinearities for deep learning magic! It's all just weighted averages	• Easy fix: apply the same feedforward network to each self-attention output.

• Need to ensure we don't "look at the future" when predicting a sequence
  • Like in machine translation
  • Or language modeling
	• 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.

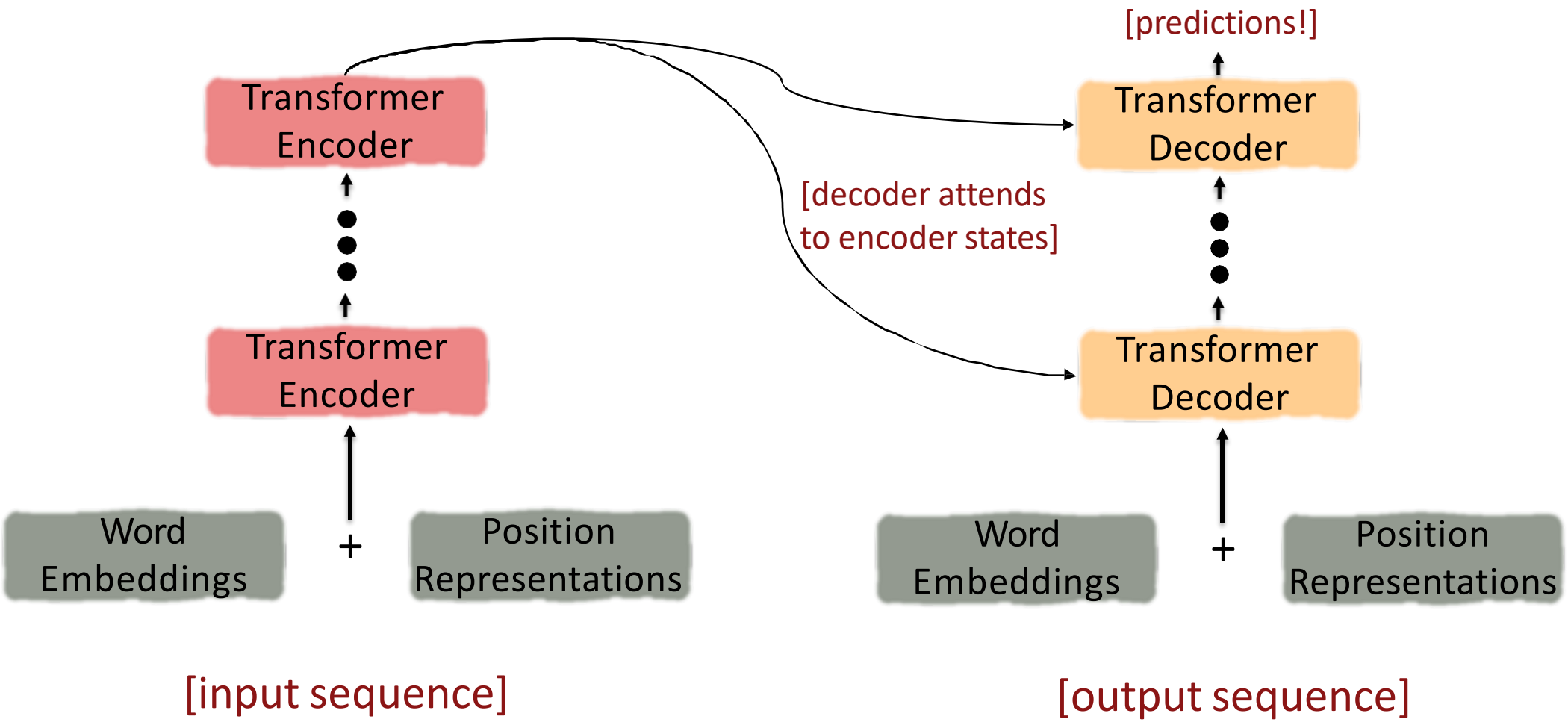
- That’s it! But this is not the Transformer model we’ve been hearing about.



FROM SELF ATTENTION TO TRANSFORMER

The Transformer Encoder-Decoder [Vaswani et al., 2017]

First, let's look at the Transformer Encoder and Decoder Blocks at a high level



The Transformer Encoder-Decoder [Vaswani et al., 2017]

Next, let's look at the Transformer Encoder and Decoder Blocks

What's left in a Transformer Encoder Block that we haven't covered?

1. **Key-query-value attention:** How do we get the k, q, v vectors from a single word embedding?
2. **Multi-headed attention**: Attend to multiple places in a single layer!
3. **Tricks to help with training!**
    1. Residual connections
    2. Layer normalization
    3. Scaling the dot product
    4. These tricks **don't improve** what the model is able to do; they help improve the training process. Both of these types of modeling improvements are very important!

The Transformer Encoder: Key-Query-Value Attention

- We saw that self-attention is when keys, queries, and values come from the same source. The Transformer does this in a particular way:
  - Let $x_1, \ldots, x_T$ be input vectors to the Transformer encoder; $x_i \in \mathbb{R}^d$

- Then keys, queries, values are:
  - $k_i = Kx_i$, where $K \in \mathbb{R}^{d \times d}$ is the key matrix.
  - $q_i = Qx_i$, where $\mathrm{Q} \in \mathbb{R}^{d \times d}$ is the query matrix.
  - $v_i = Vx_i$, where $\mathrm{V} \in \mathbb{R}^{d \times d}$ is the value matrix.

- These matrices allow *different aspects* of the x vectors to be used/emphasized in each of the three roles.

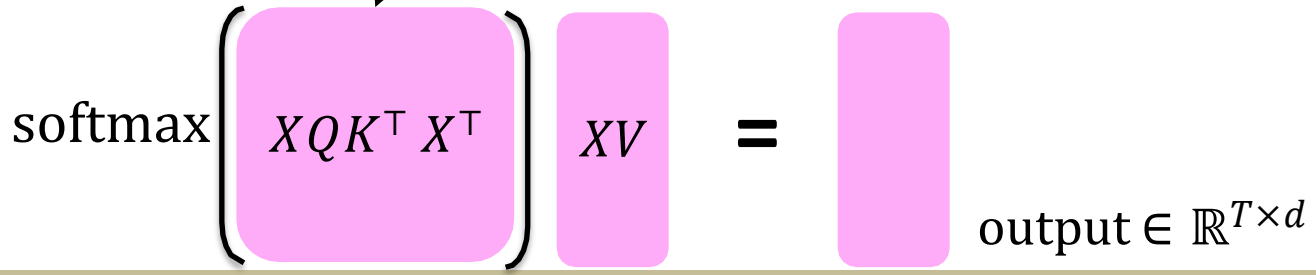
The Transformer Encoder: Key-Query-Value Attention

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

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

All pairs of attention scores!

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



figure

The Transformer Encoder: Multi-headed attention

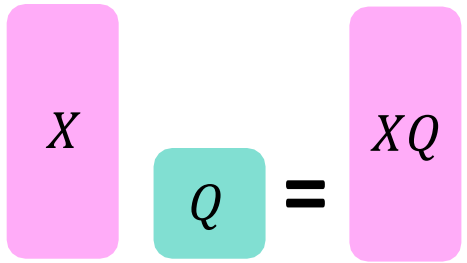
- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $x^\top Q^\top K x_j$ is high, but maybe we want to focus on different j for different reasons?
- We'll define **multiple attention "heads"** through multiple Q,K,V matrices
- Let, $Q_\ell,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

- Each head gets to "look" at different things, and construct value vectors differently.

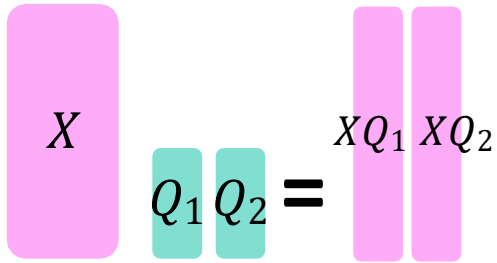
The Transformer Encoder: Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $x^\top Q^\top K x_j$ 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.

Single-head attention
(just the query matrix)

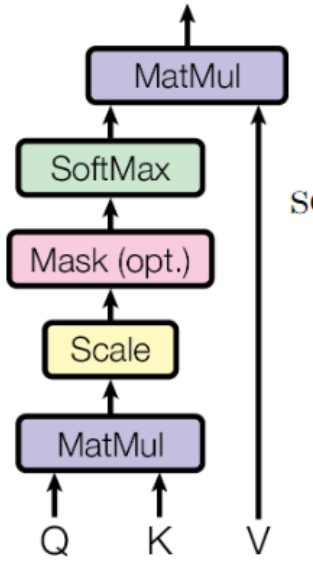


Multi-head attention
(just two heads here)



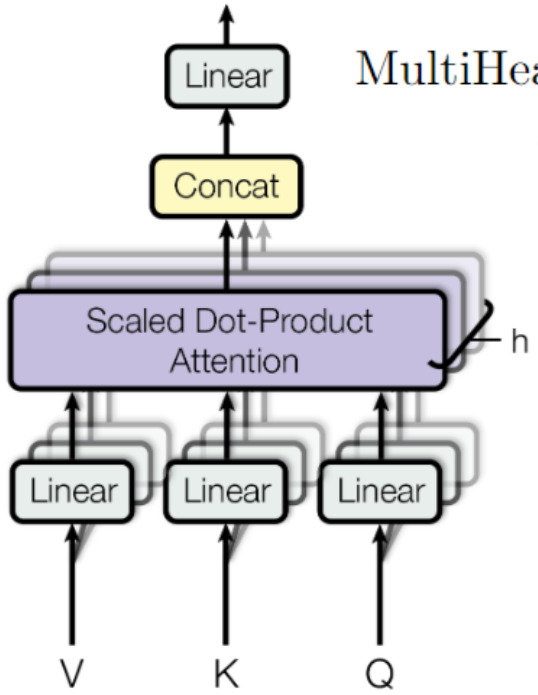
Same amount of computation as single-head self-attention!

Scaled Dot-Product Attention



$$\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Multi-Head Attention



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

In “Attention is all you need” paper, they use

d_model=512, h =8,
dk = dv = dmodel/h = 64

3.2.2 Multi-Head Attention

Instead of performing a single attention function with d_{model}-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

Training Tricks

- Residual connections are a trick to help models train better
- 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
- **Scaled Dot Product**
  - 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

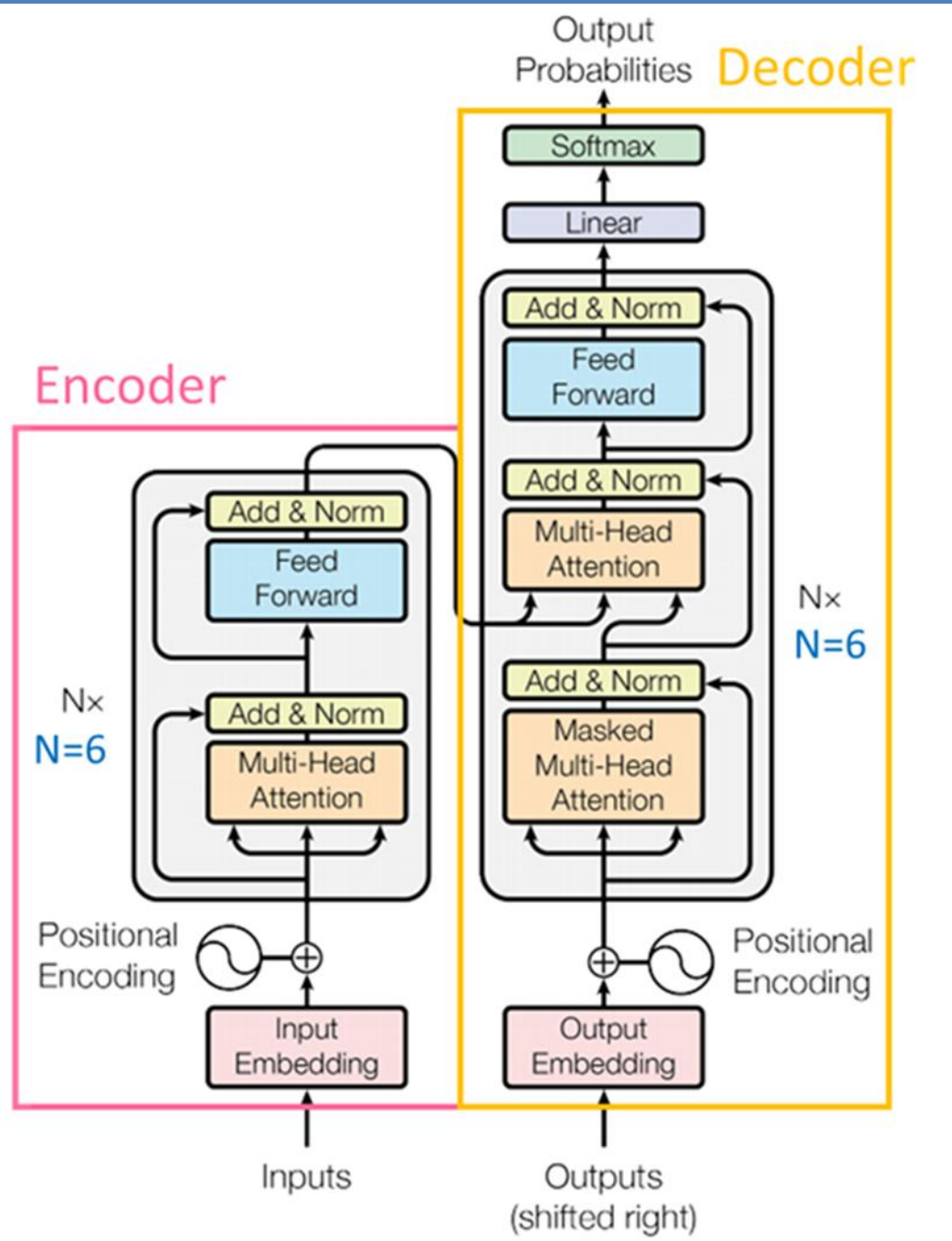


Figure 1: The Transformer - model architecture.

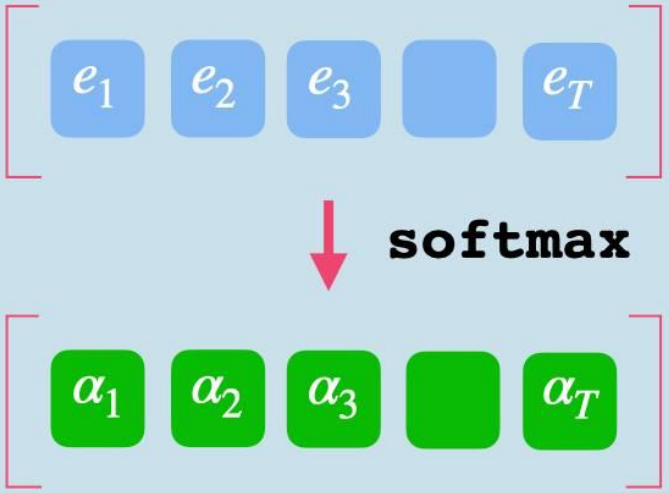
Q-K-V 矩陣

所有的 q_i, 所有的 attention 一次寫出來是這樣:

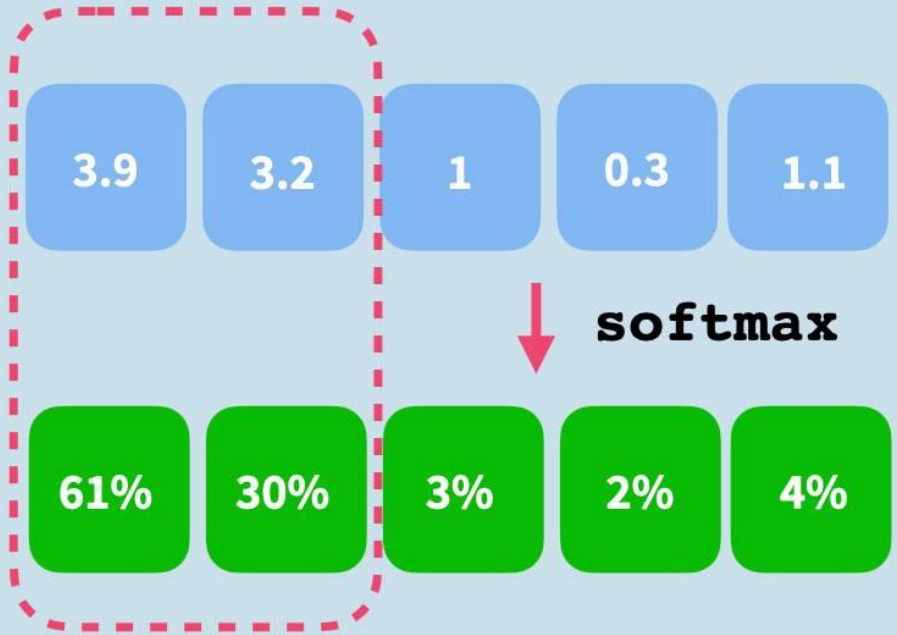
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

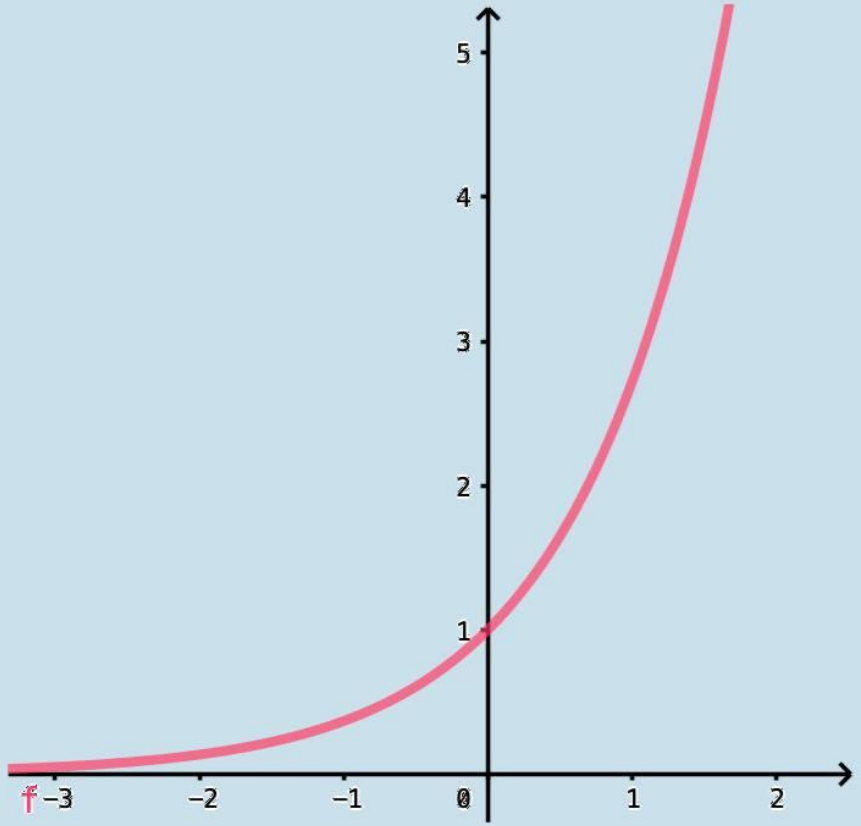
d_k 是一個 key 向量的維度, 為什麼要除以這神秘數字呢?

Q-K-V 矩陣



Q-K-V 矩陣





本來感覺只差一點啊...

Q-K-V 矩陣

3.9	3.2	1	0.3	1.1

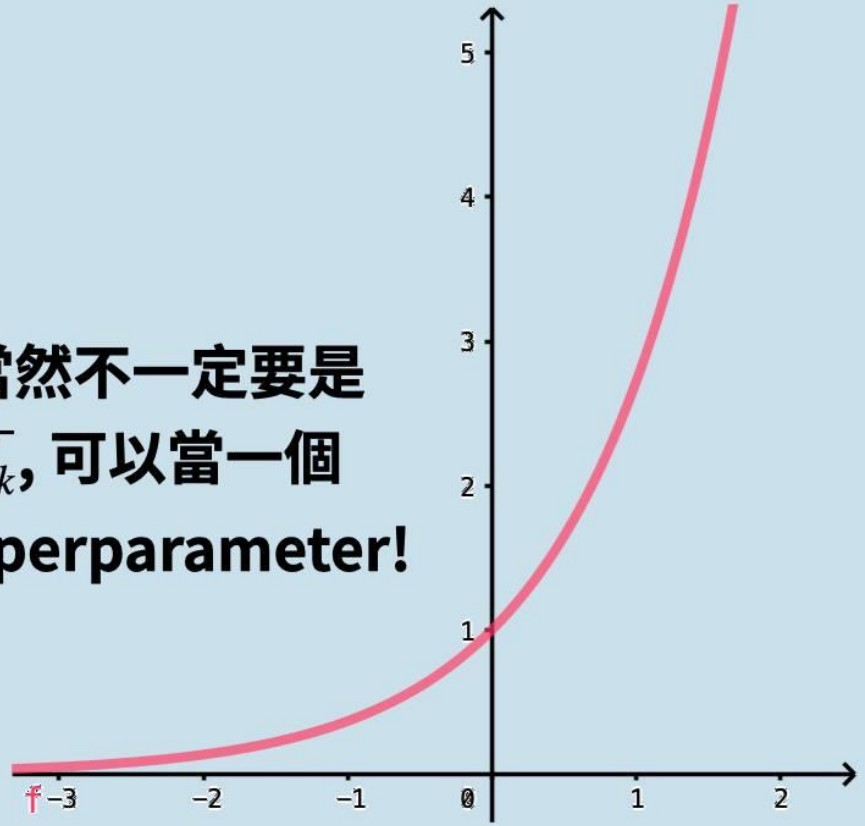
↓ 同除以 $\tau = \sqrt{d_k}$

1.74	1.43	0.45	0.13	0.49

↓ softmax

40%	29%	11%	8%	11%

τ 當然不一定要是 $\sqrt{d_k}$, 可以當一個 hyperparameter!

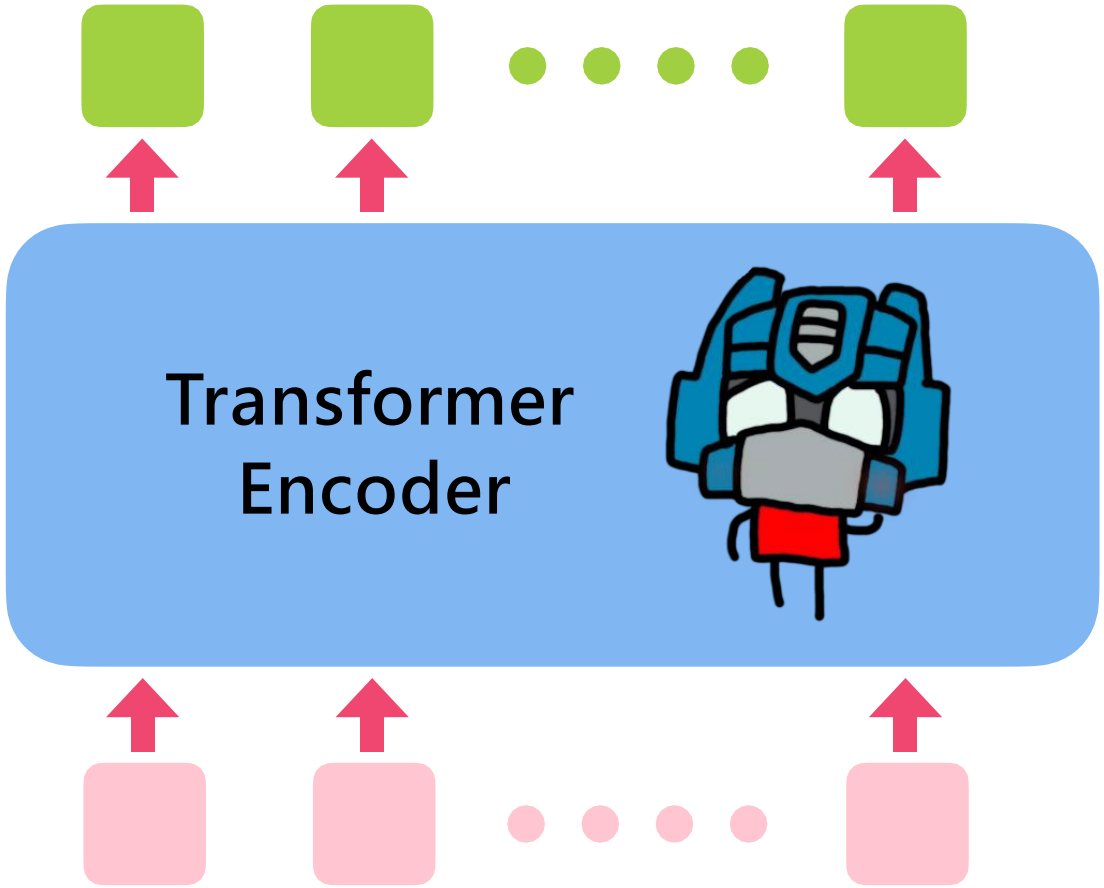


Q-K-V 矩陣

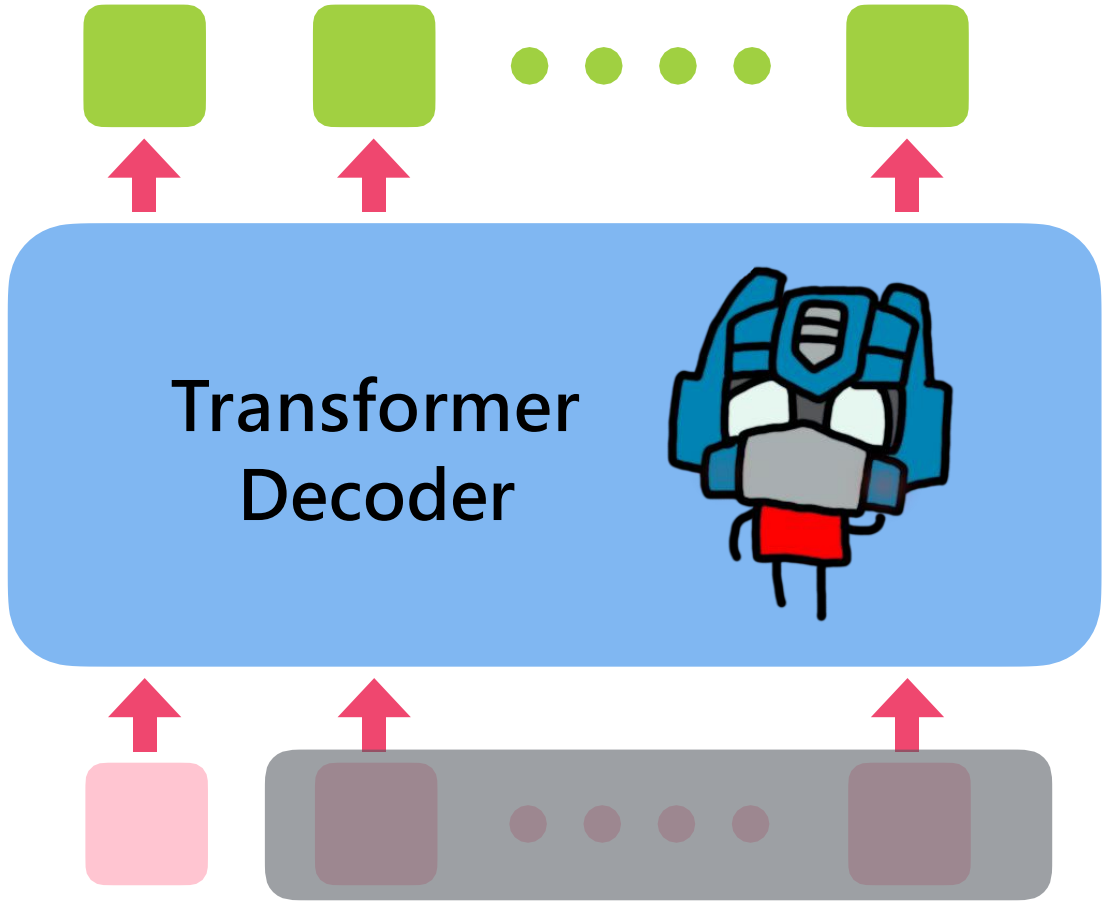
再看一次 transformer 的 attention 公式, 是不是就很清楚了呢?

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

要注意的是，transformer 輸入有幾個 (字)，
輸出就有幾個。

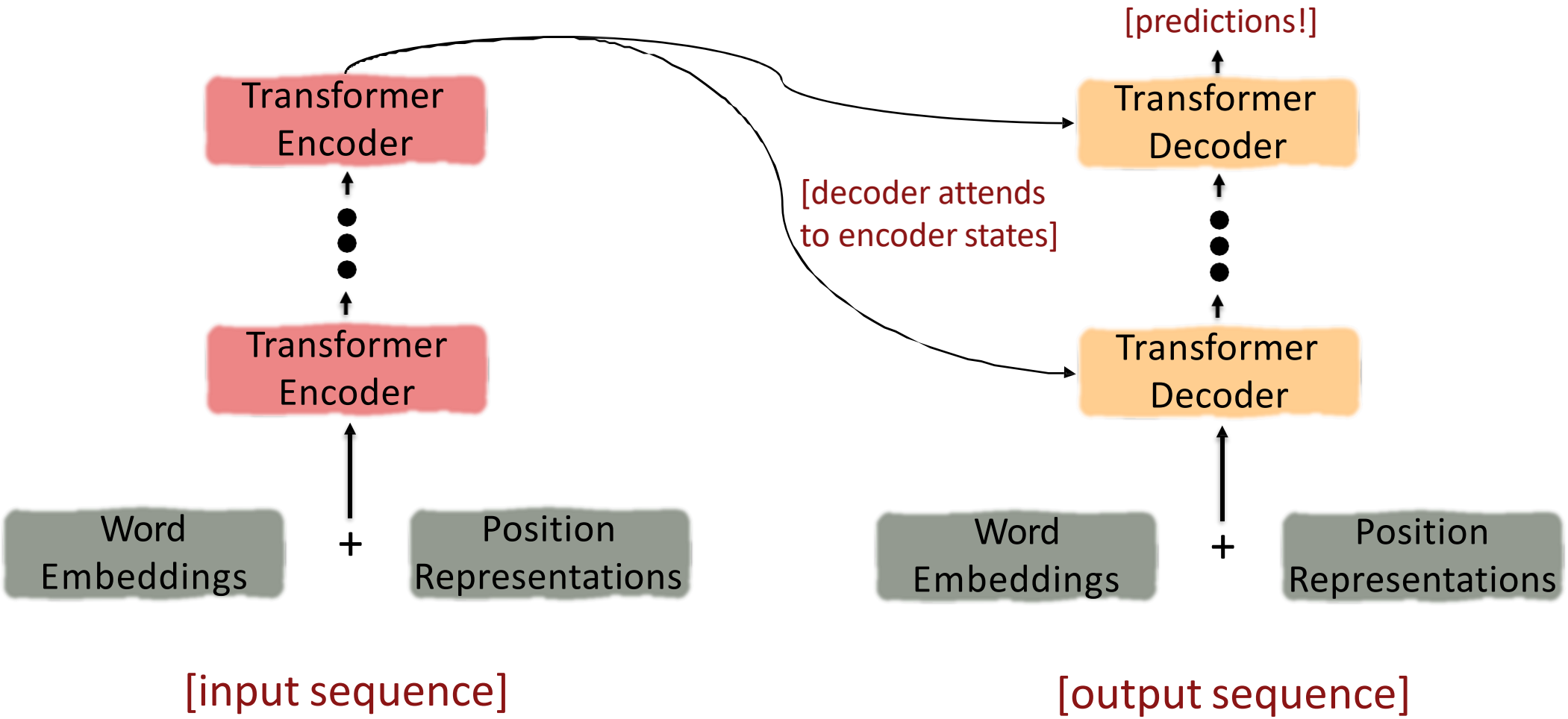


包括 Decoder 也是這樣，只是開始還沒有的輸入會被 mask 住。

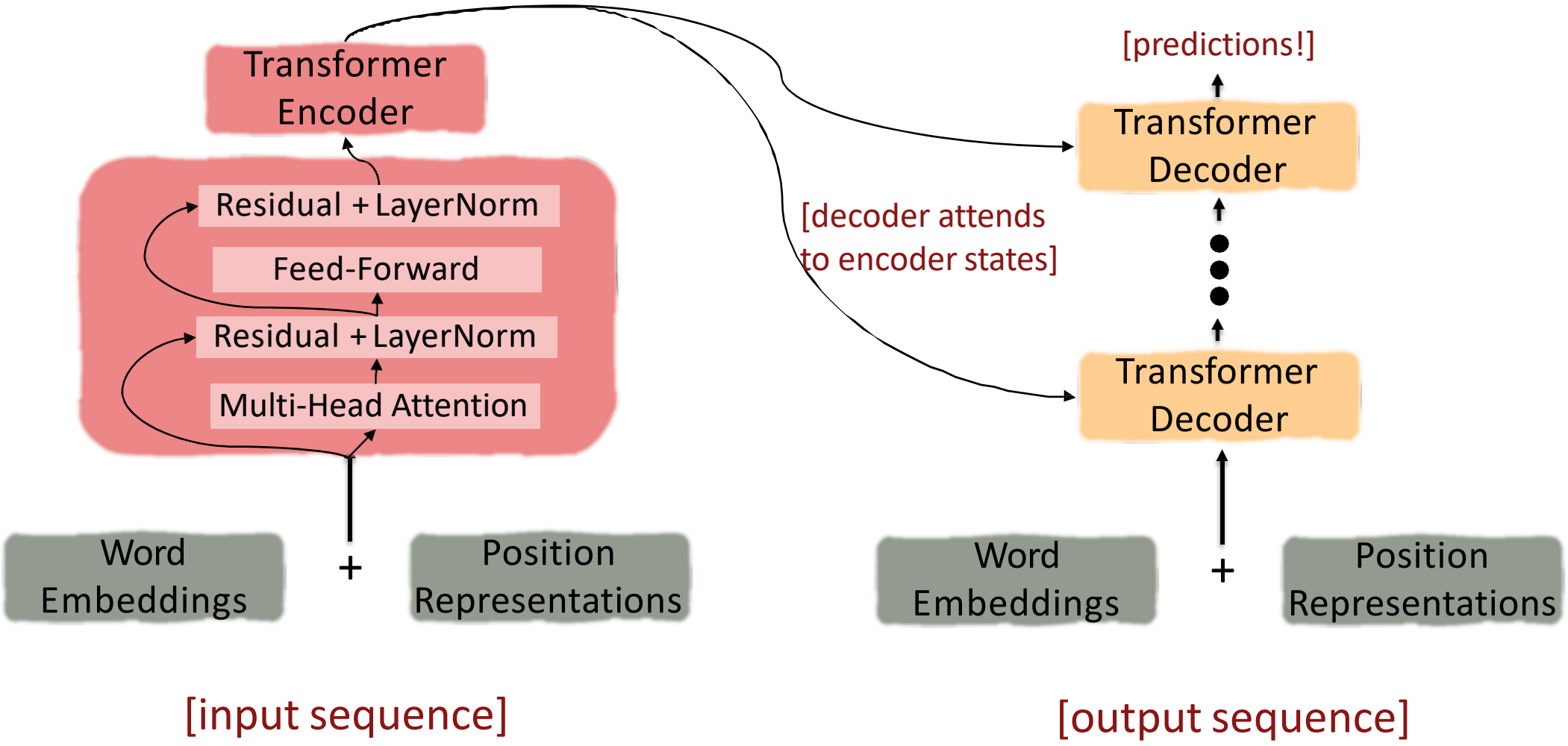


The Transformer Encoder-Decoder [Vaswani et al., 2017]

Looking back at the whole model, zooming in on an Encoder block:

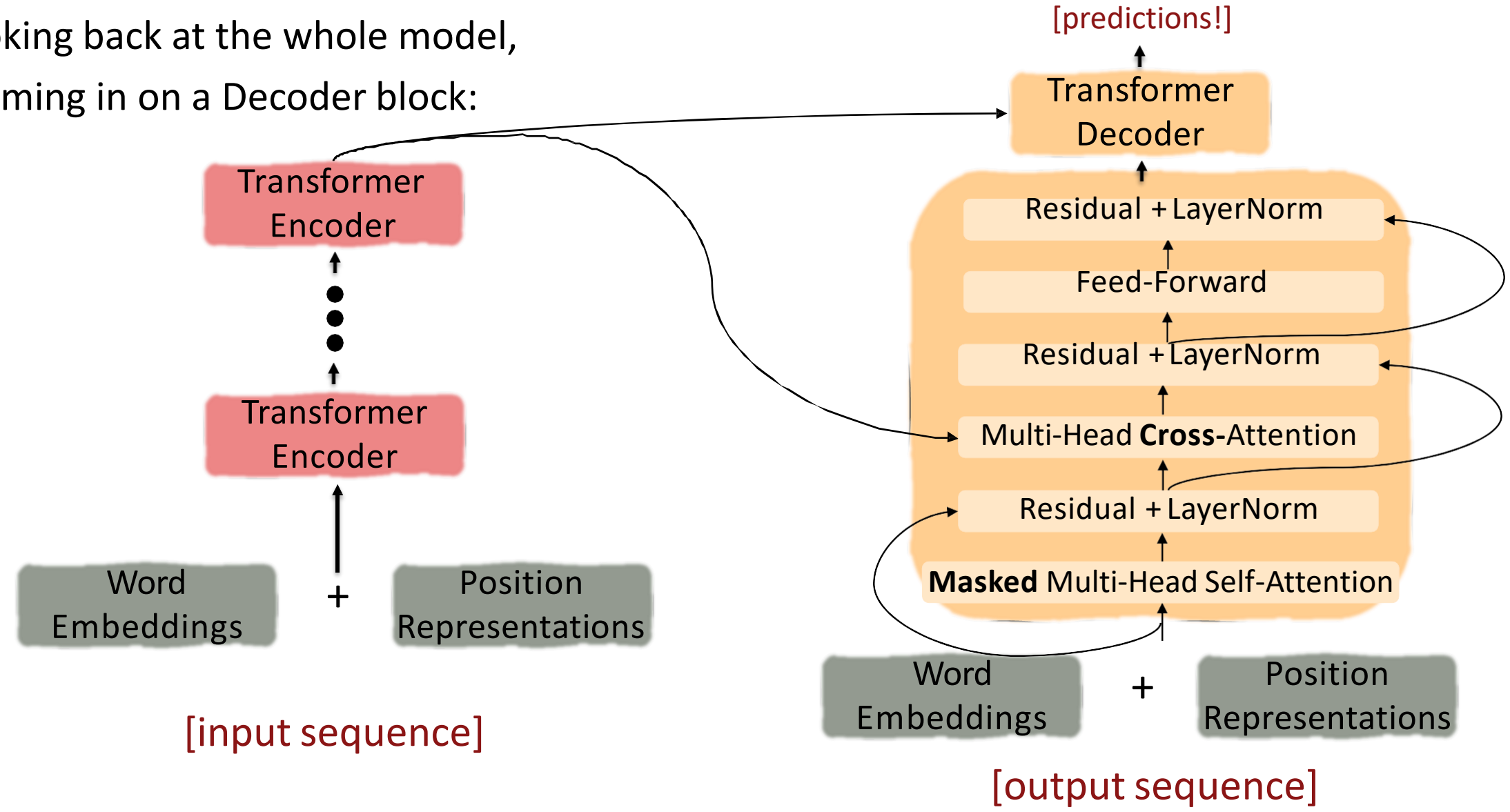


Looking back at the whole model, zooming in on an Encoder block:

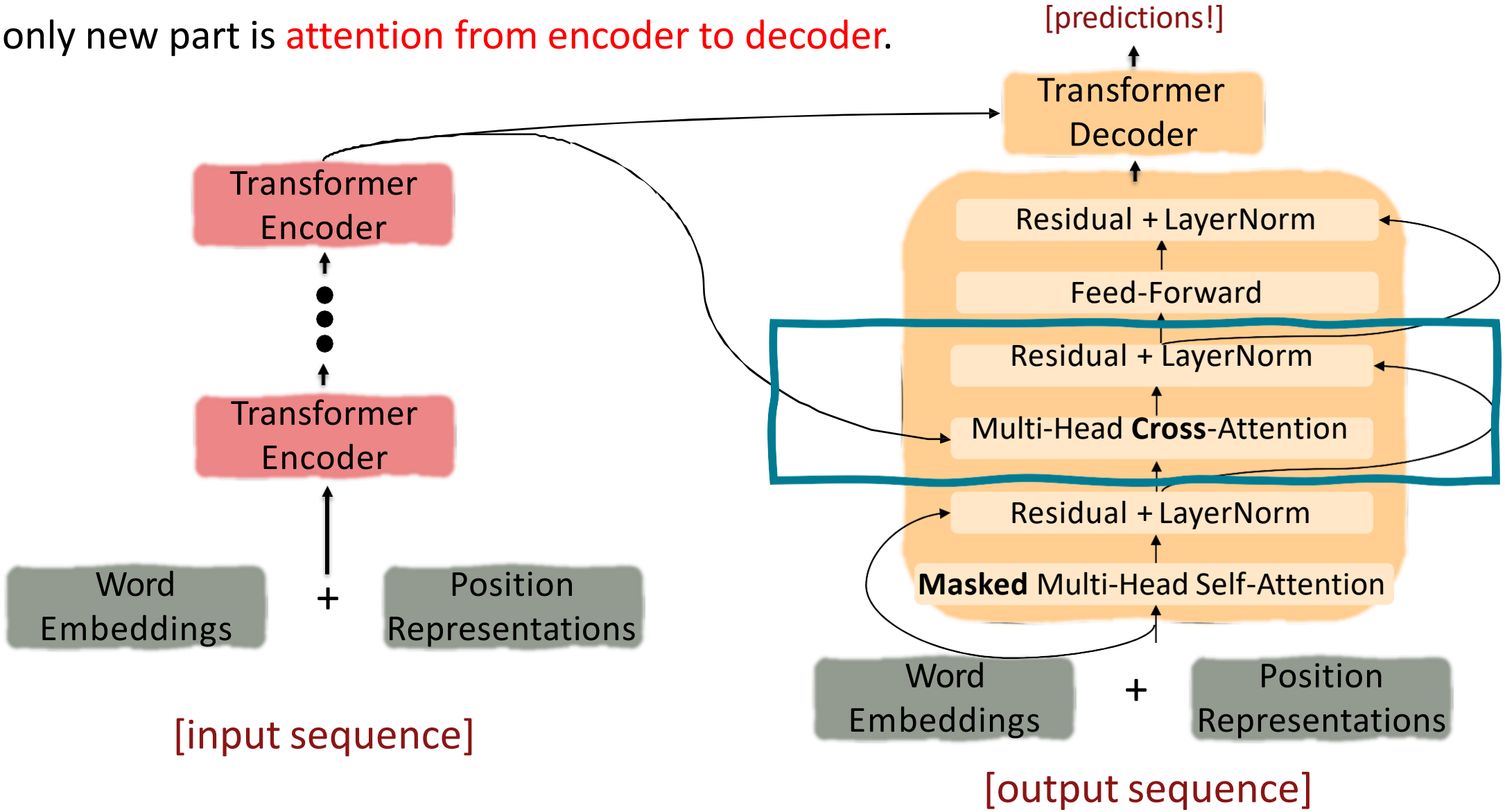


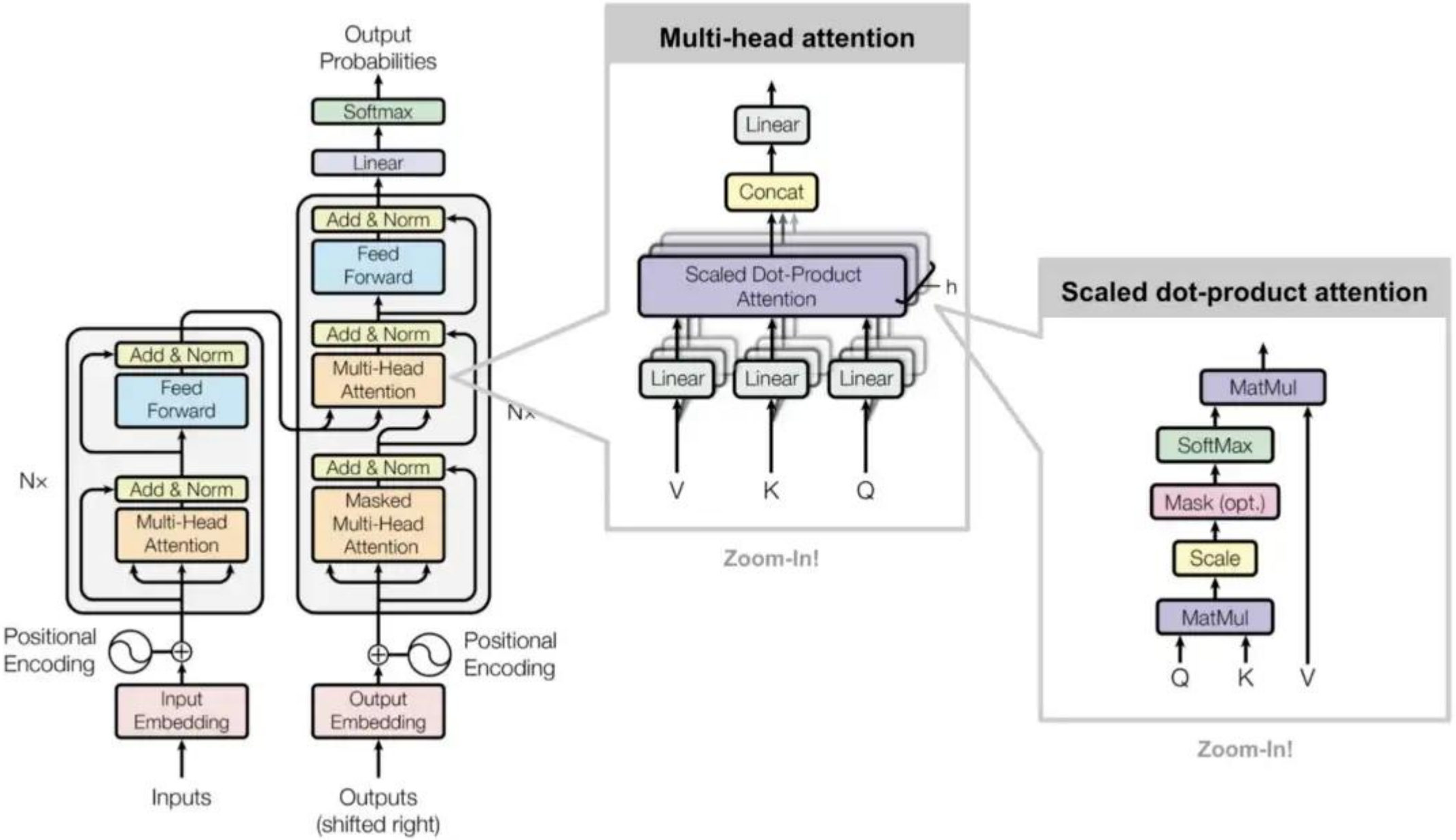
Looking back at the whole model,

zooming in on a Decoder block:



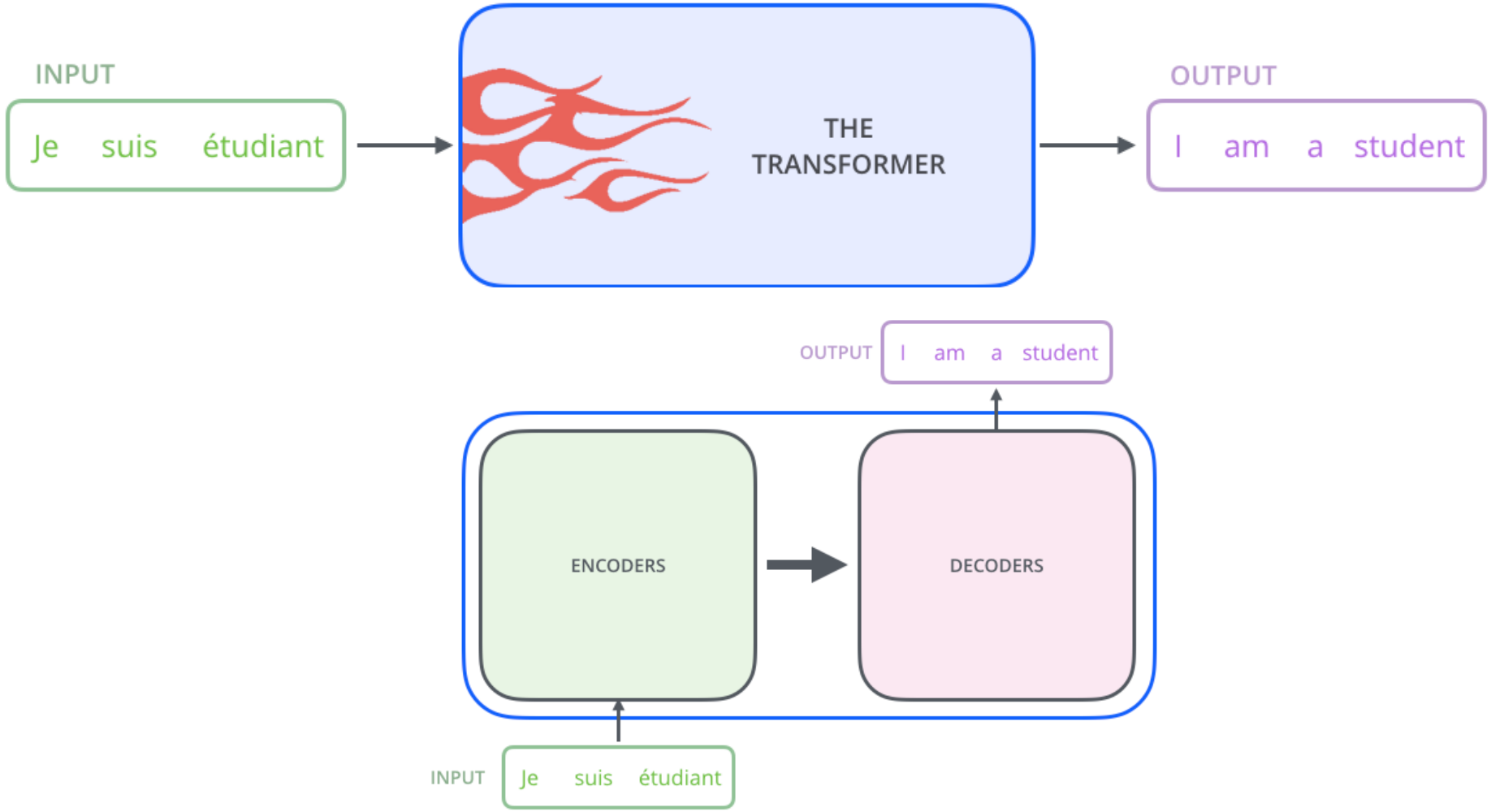
The only new part is attention from encoder to decoder.

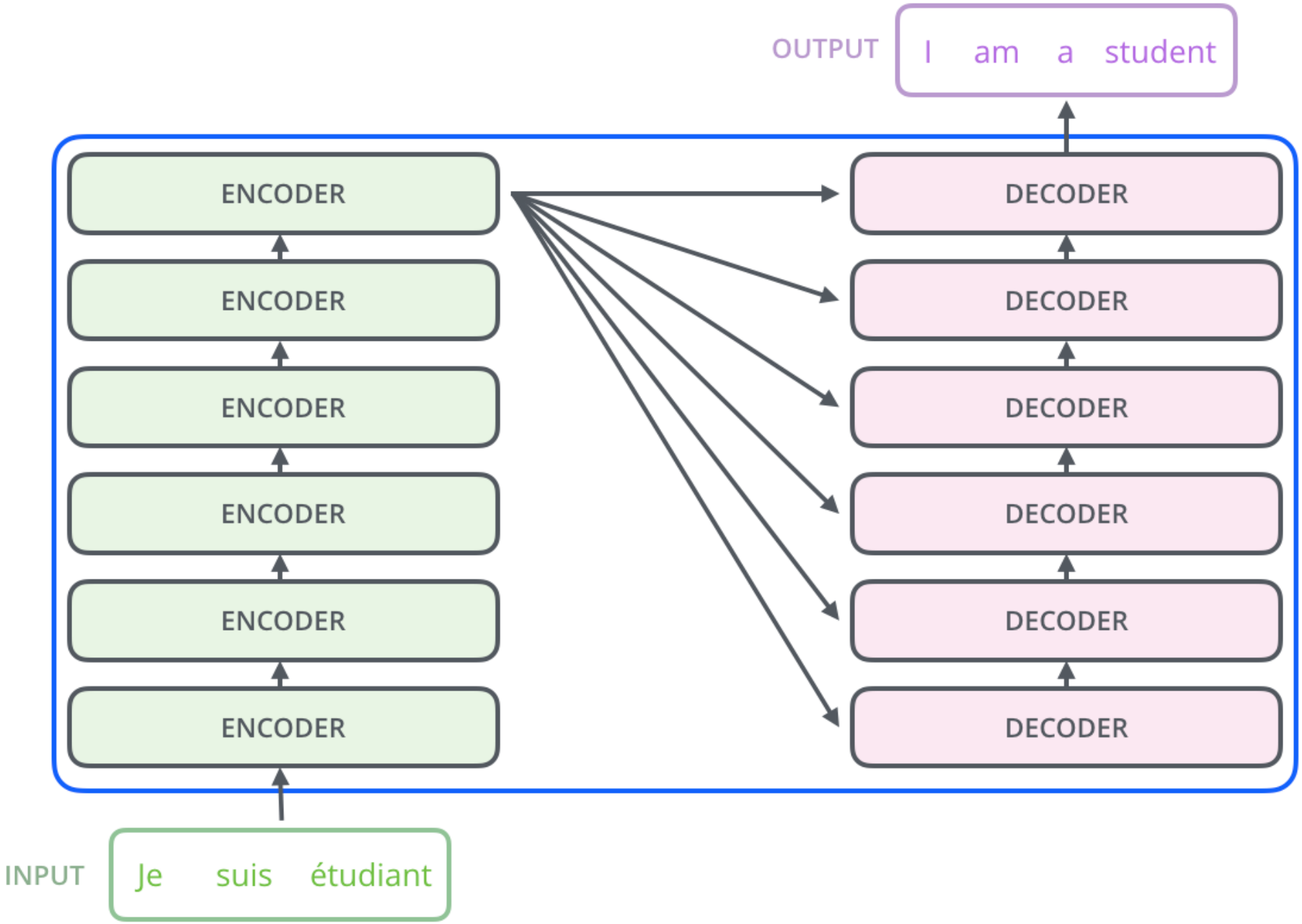




https://tamoghnasaha-22.medium.com/transformers-illustrated-5c9205a6c70f

Transformer



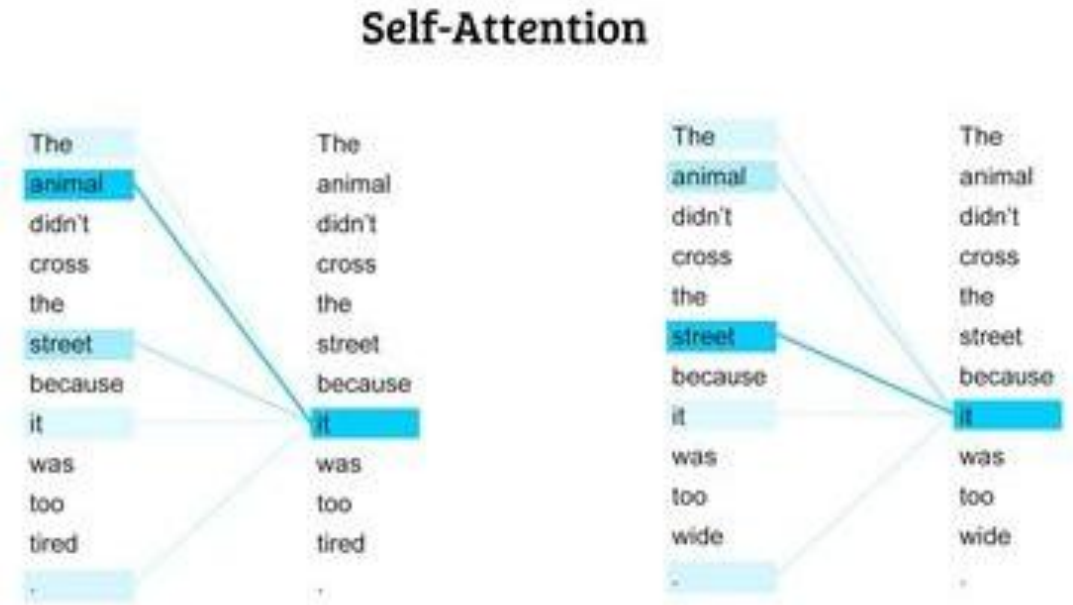

OUTPUT
I am a student
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
DECODER
DECODER
DECODER
DECODER
DECODER
DECODER
INPUT
Je suis étudiant

- 不同的Attention Head到底學習到了什麼樣的語義。

1. *"The animal didn't cross the street because it was too tired"*

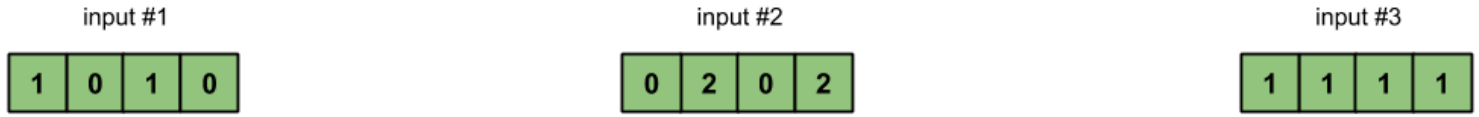
2. *"The animal didn't cross the street because it was too wide"*

- All tokens can attend to each other equally well, no matter how far they are from each other
- Transformer captures long-range semantic dependencies
  - Creates global context into the output embeddings

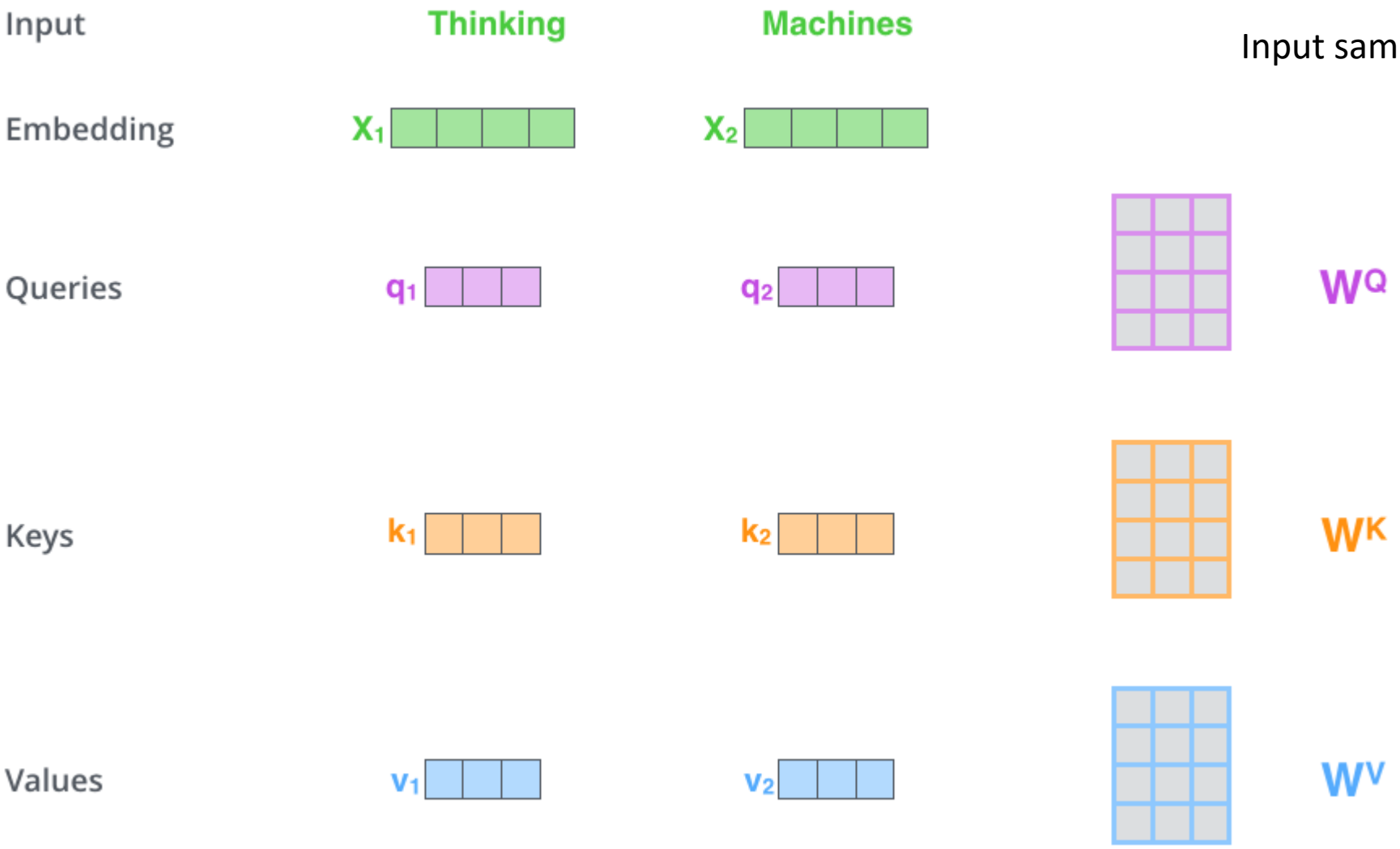


ILLUSTRATION

Self-attention



1. Create Q, K, V form inputs (embedding of input vectors)

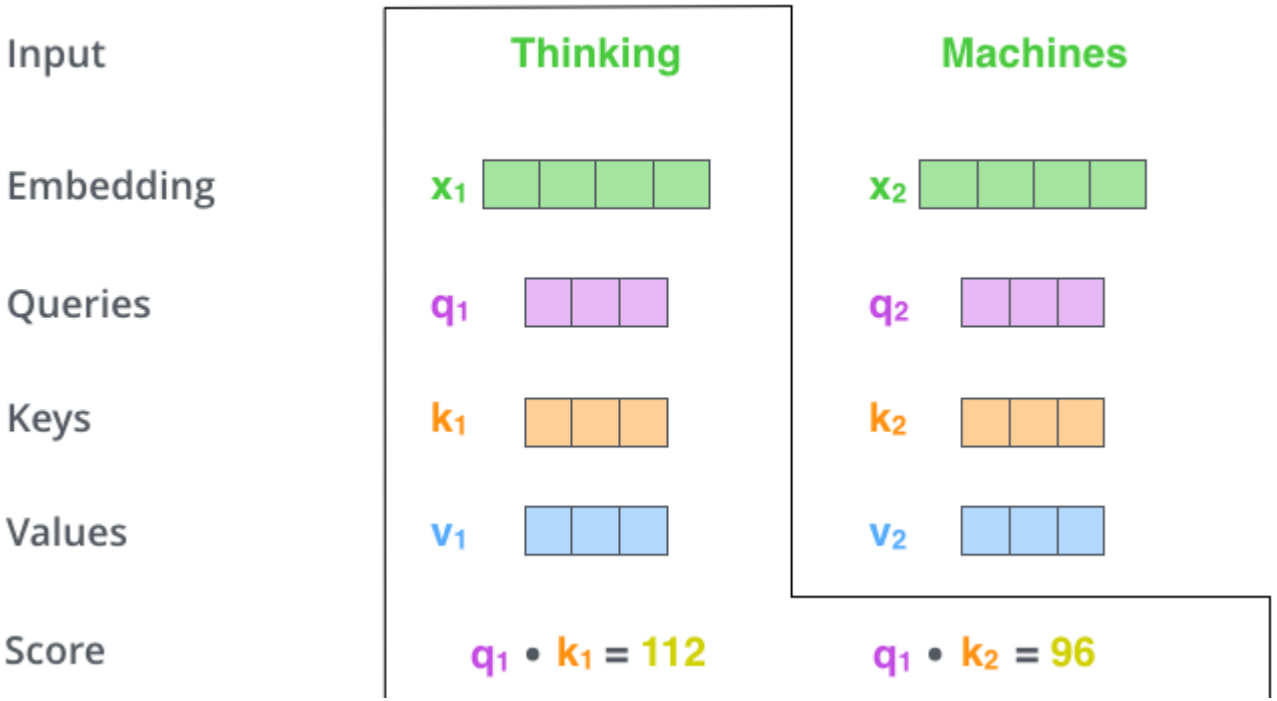


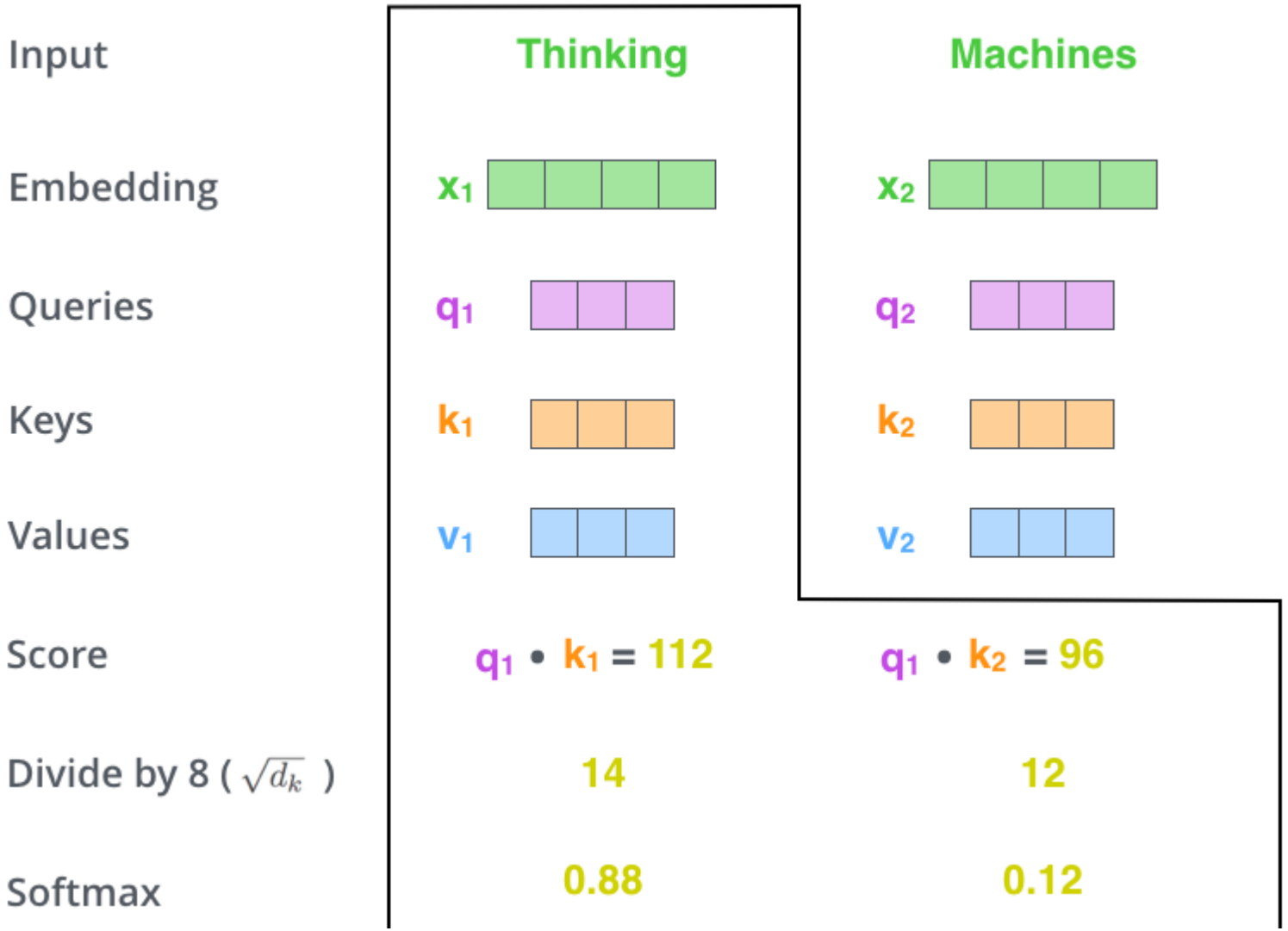
Input sample * Wq/Wk/Wv

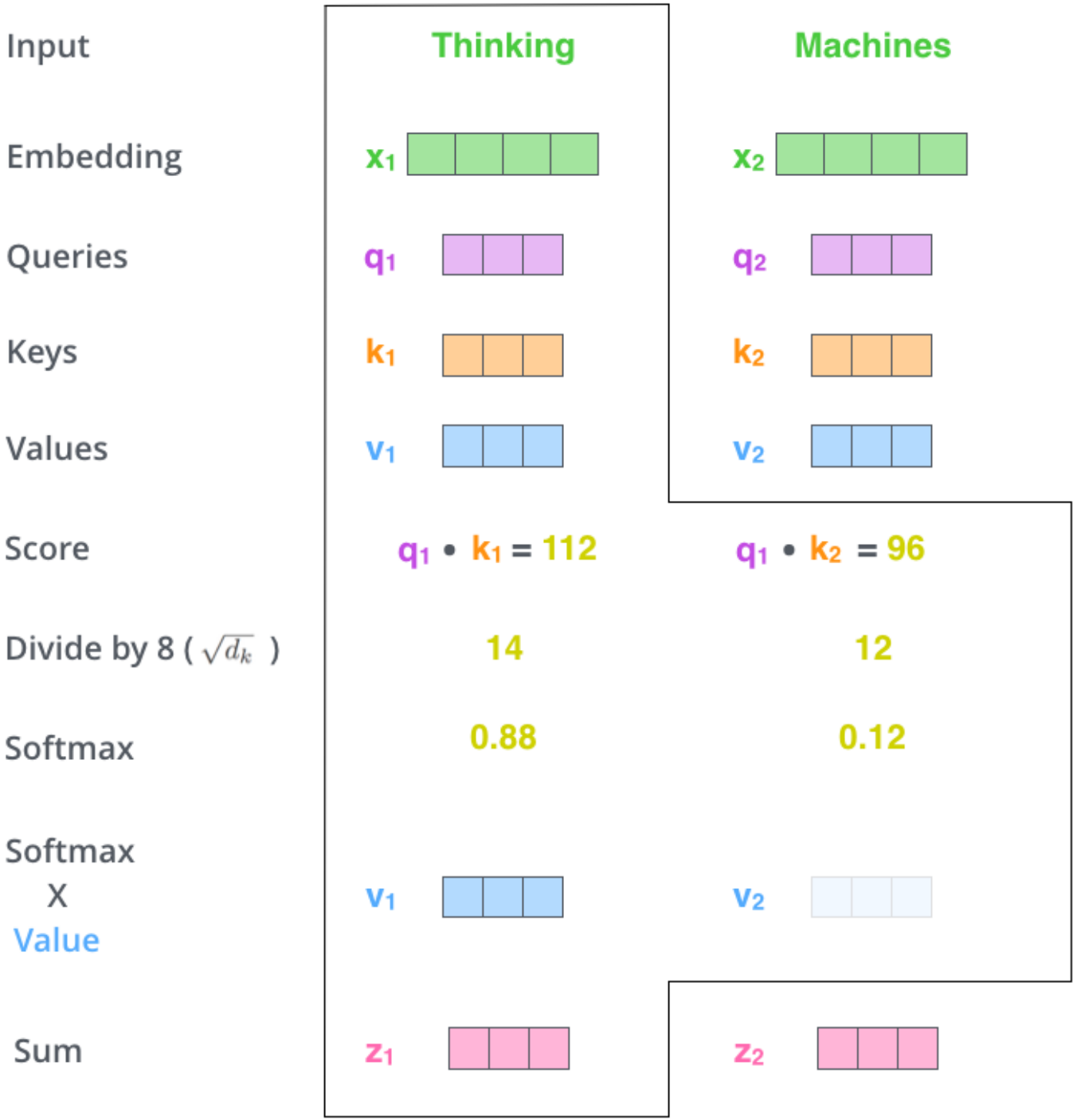
Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word.
We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

https://jalammar.github.io/illustrated-transformer/

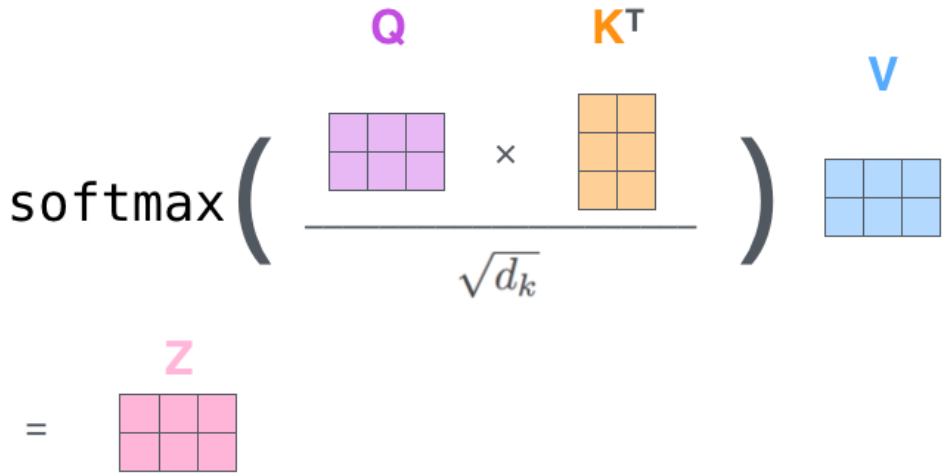
2. compute the similarity (or attention score)




Input
Thinking
Machines
Embedding
x1
x2
Queries
q1
q2
Keys
k1
k2
Values
v1
v2
Score
q1 • k1 = 112
q1 • k2 = 96
Divide by 8 (√dk)
14
12
Softmax
0.88
0.12


Input
Thinking
Machines
Embedding
x1
x2
Queries
q1
q2
Keys
k1
k2
Values
v1
v2
Score
q1 • k1 = 112
q1 • k2 = 96
Divide by 8 (√dk)
14
12
Softmax
0.88
0.12
Softmax X Value
v1
v2
Sum
z1
z2

Matrix Calculation of Self-Attention



https://jalammar.github.io/illustrated-transformer/

1) This is our input sentence*

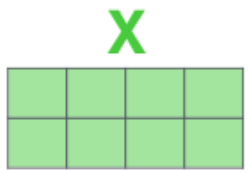
2) We embed each word*

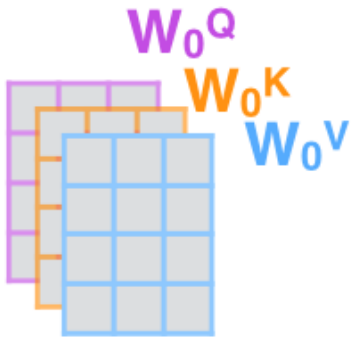
3) Split into 8 heads. We multiply X or R with weight matrices

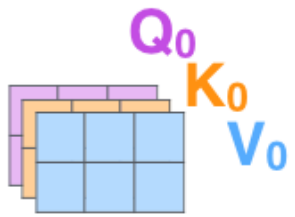
4) Calculate attention using the resulting Q/K/V matrices

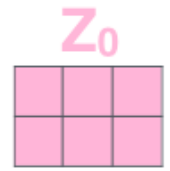
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

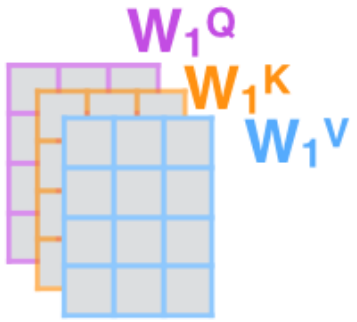
Thinking Machines

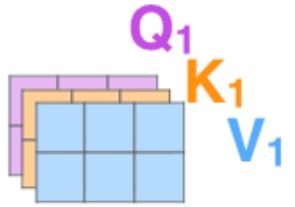


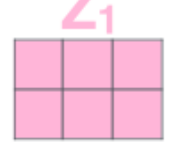


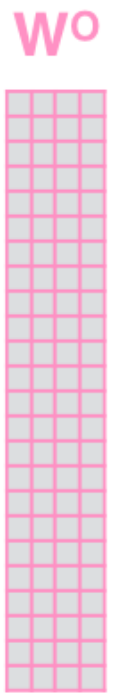


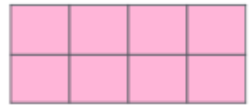




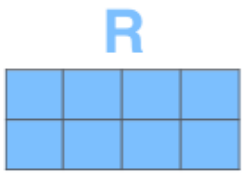


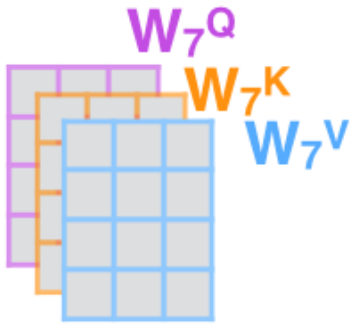


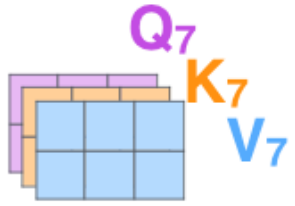


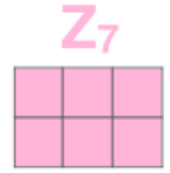


* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one









Self-attention

https://arxiv.org/abs/1706.03762

q: query (to match others)

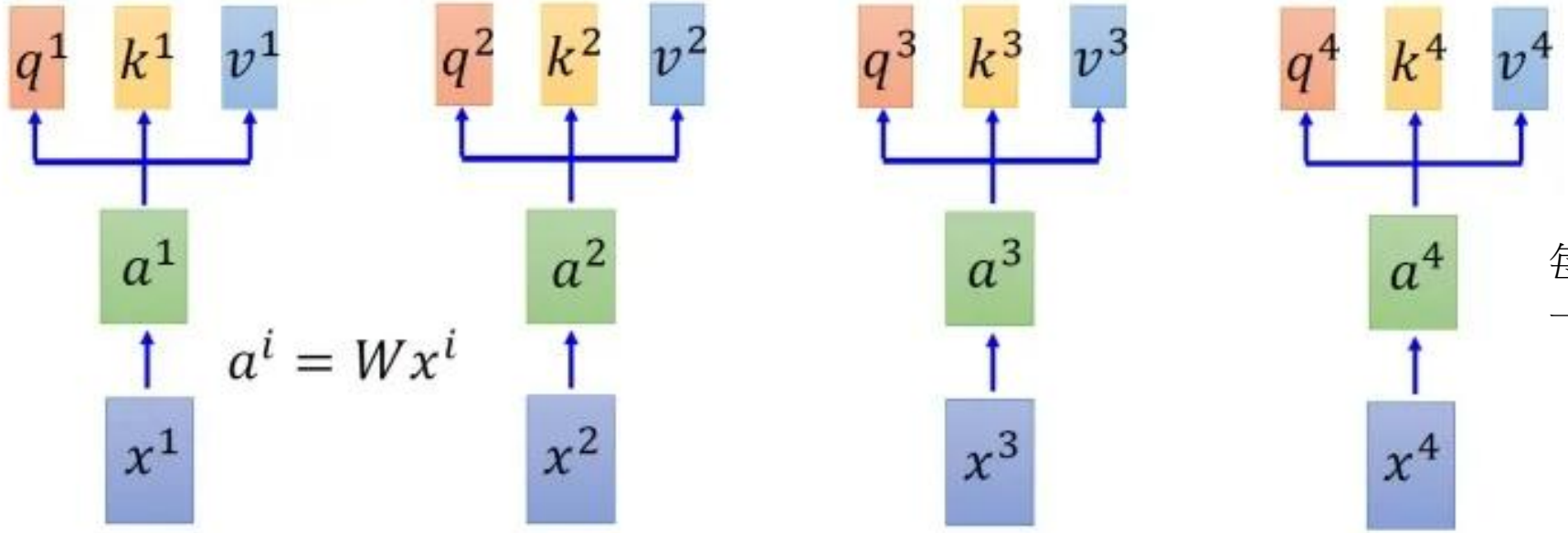
$$q^i = W^q a^i$$

k: key (to be matched)

$$k^i = W^k a^i$$

v: information to be extracted

$$v^i = W^v a^i$$



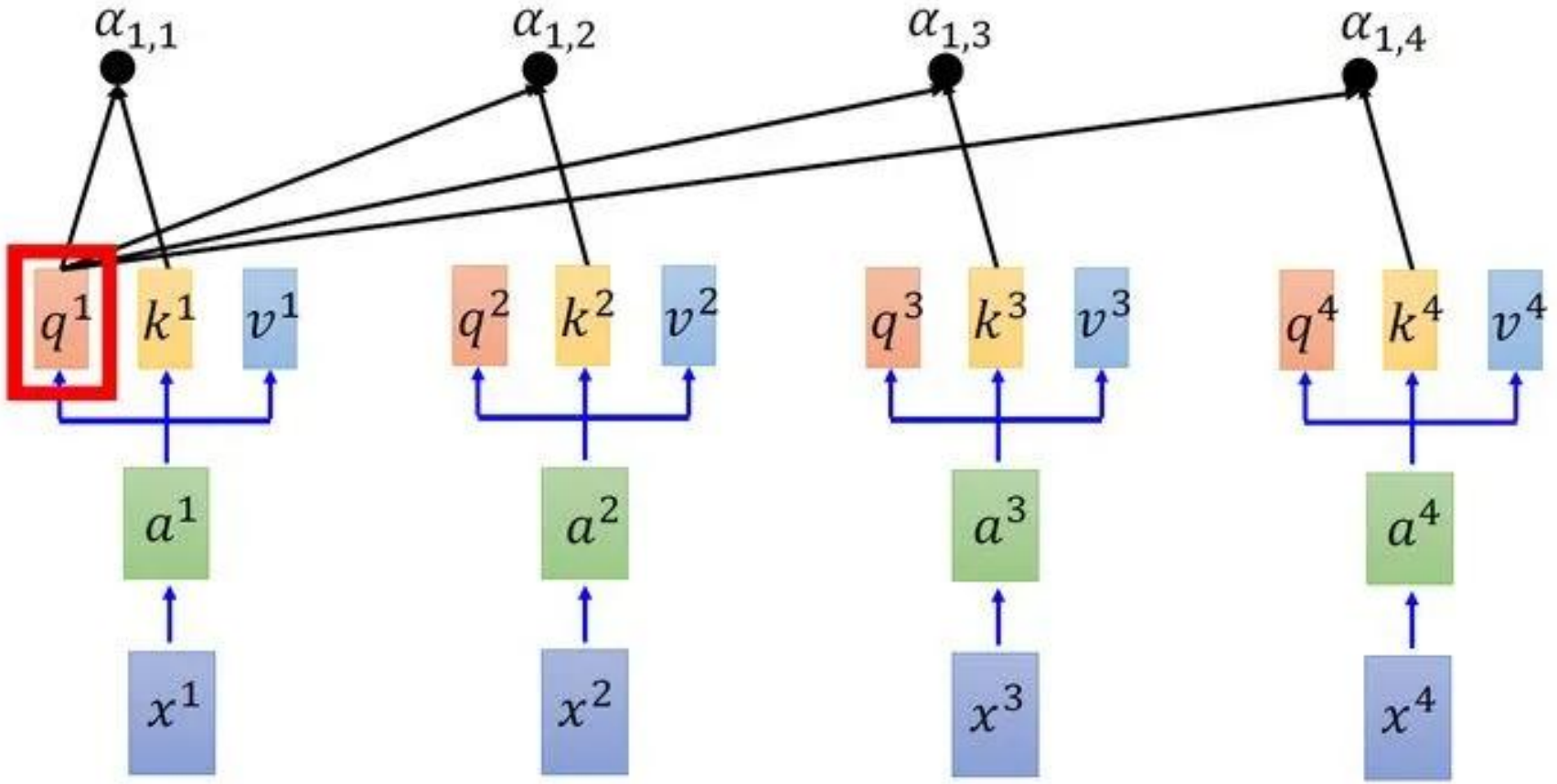
每一個input (vector) X，先乘上一個矩陣W 得到embedding a

Self-attention

拿每個 query q 去對每個 key k 做 attention

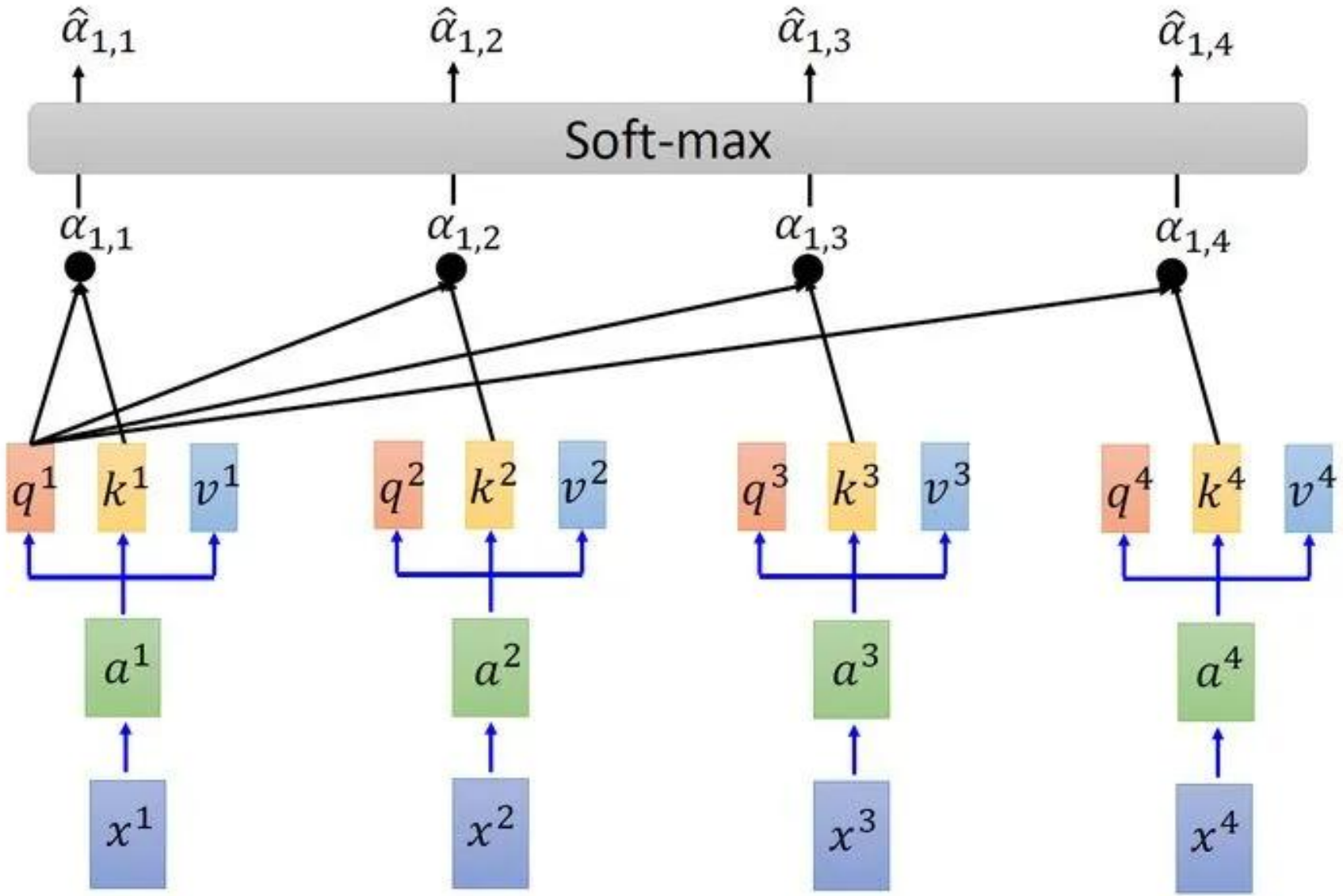
d is the dim of q and k

Scaled Dot-Product Attention: $\alpha_{1,i} = \underbrace{q^1 \cdot k^i}_{\text{dot product}} / \sqrt{d}$



Self-attention

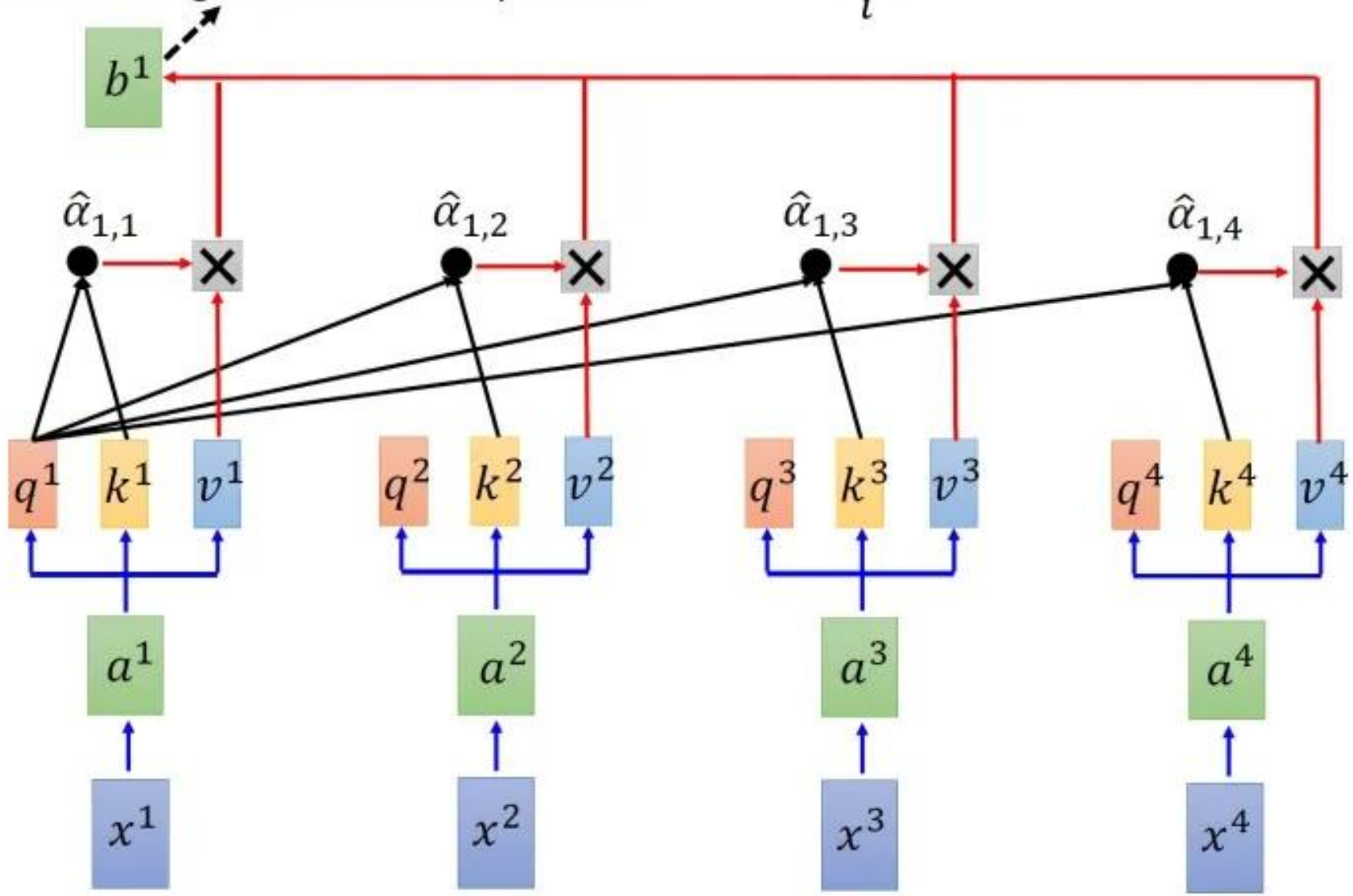
$$\hat{\alpha}_{1,i} = exp(\alpha_{1,i}) / \sum_j exp(\alpha_{1,j})$$



Self-attention

Considering the whole sequence

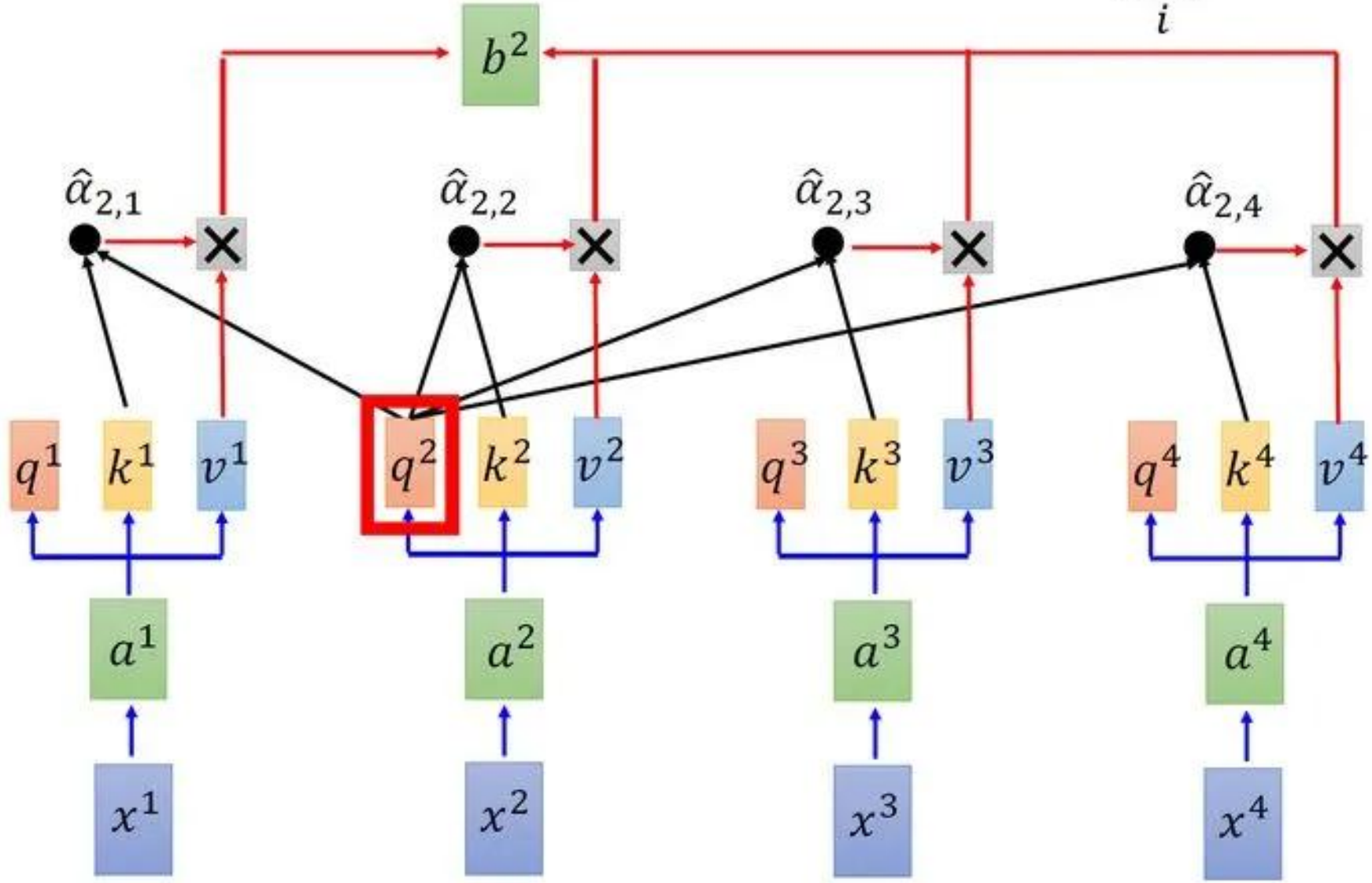
$$b^1 = \sum_i \hat{\alpha}_{1,i} v^i$$



Self-attention

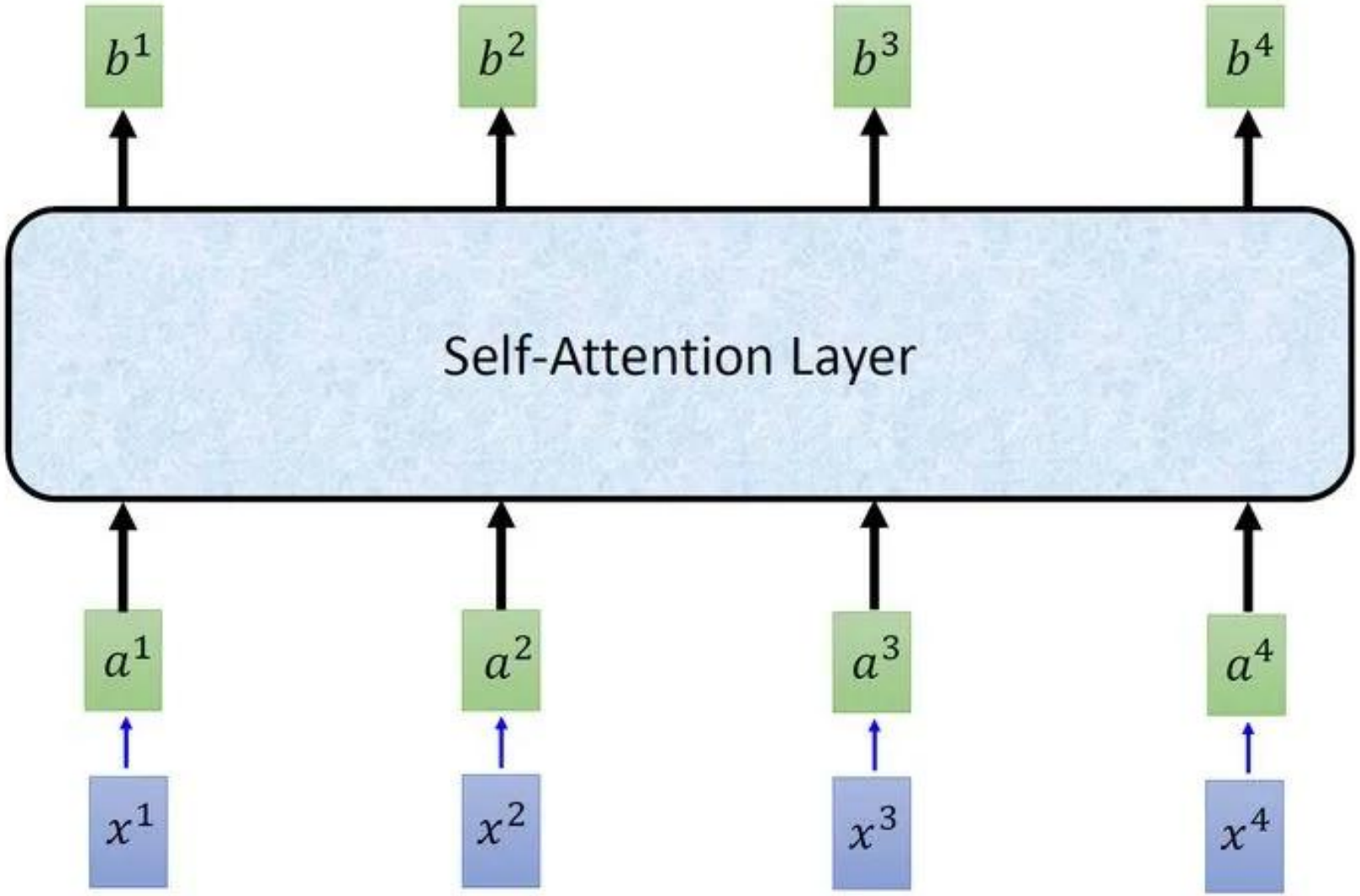
拿每個 query q 去對每個 key k 做 attention

$$b^2 = \sum_i \hat{\alpha}_{2,i} v^i$$



Self-attention

b^1, b^2, b^3, b^4 can be parallelly computed.



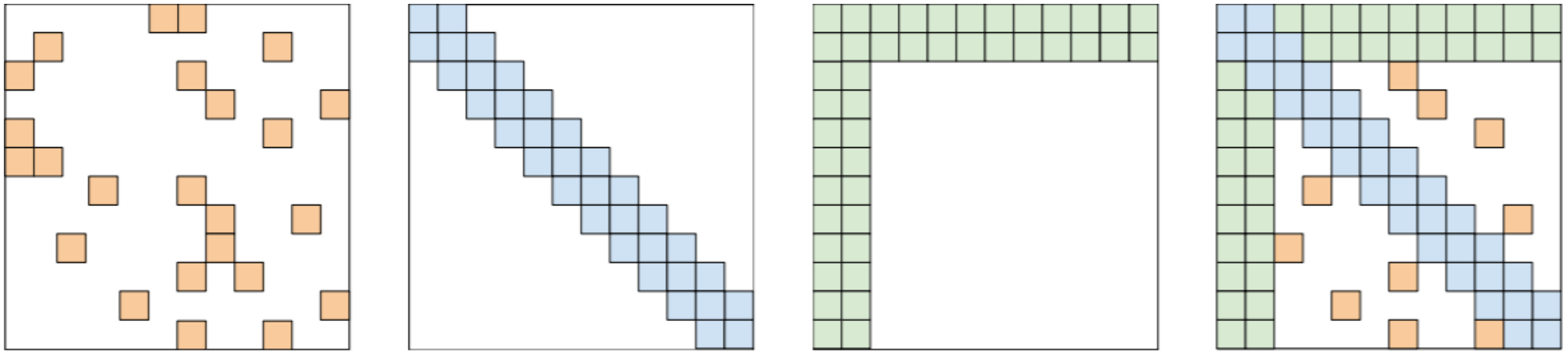
Drawbacks of Transformers

- **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]

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, **BigBird** [Zaheer et al., 2021]

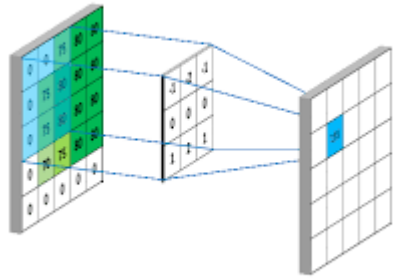
Key idea: replace all-pairs interactions with a family of other interactions, **like local windows**, **looking at everything**, and **random interactions**.



(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

Self-attention vs. Convolution in Vision

- Each pixel is an input “token”
- Traditionally, convolution is used to integrate pixels
  - Recognize patterns within a small window of pixels
  - Difficult to integrate non-local pixels
  - Have to make network very deep to “see the big picture”

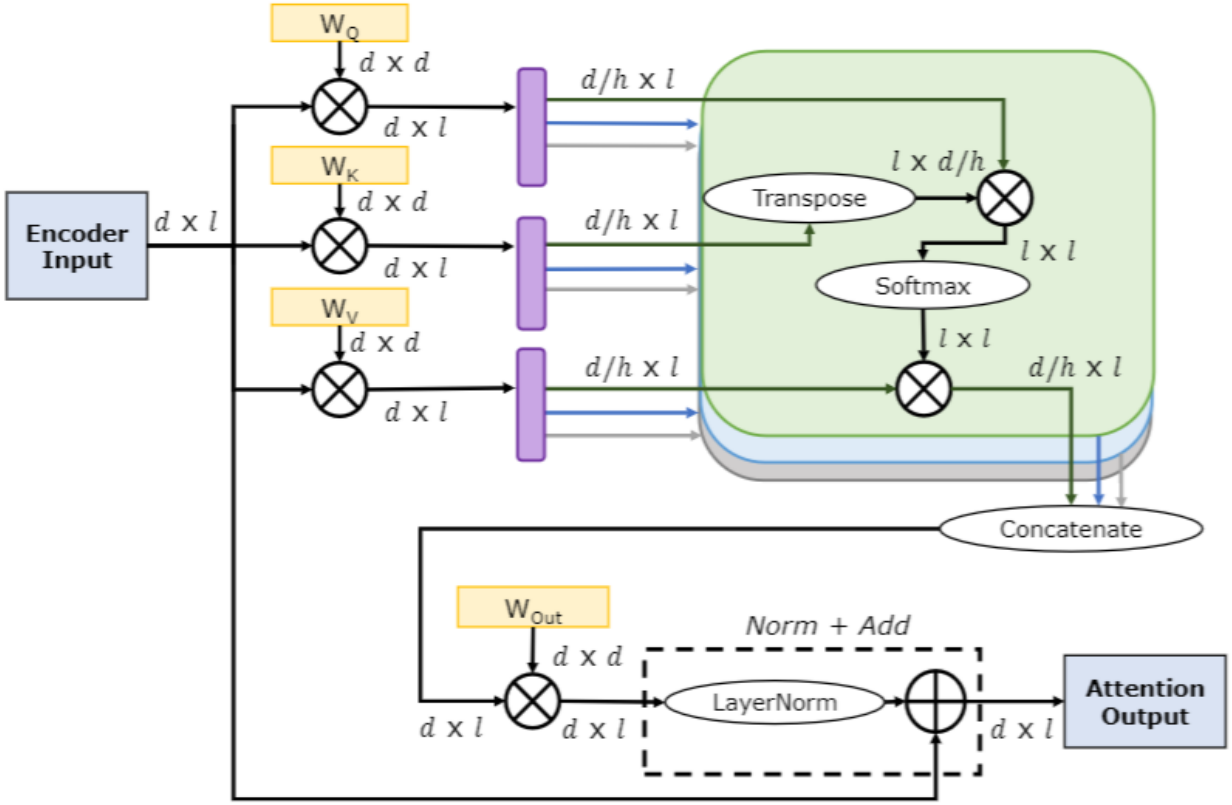


- Self-attention (transformer) also integrates multiple pixels
  - Works when the correlated pixels are non-local
  - E.g. Trunk, tail, legs of an elephant to a whole elephant

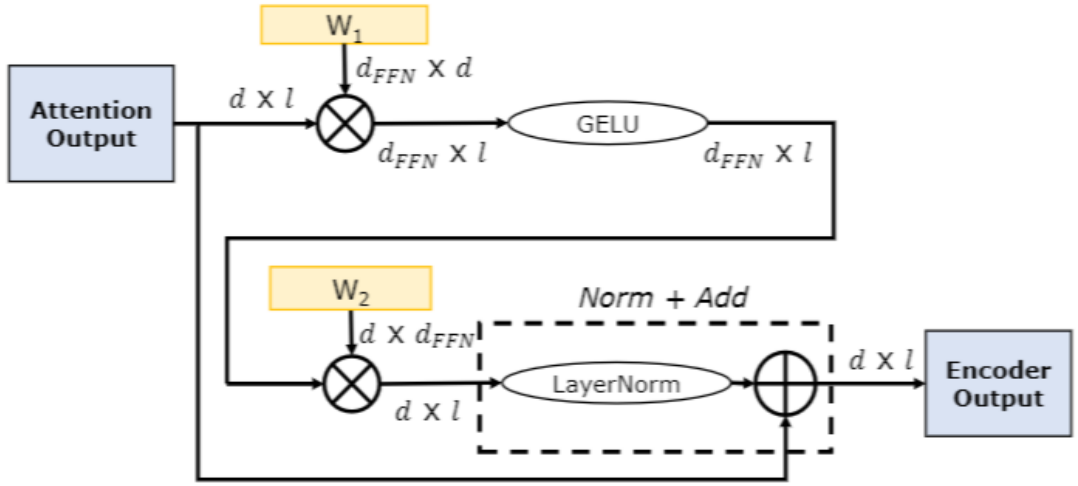
DIMENSION OF TRANSFORMER

https://towardsdatascience.com/into-the-transformer-5ad892e0cee

N	d_{model}	d_{ff}	h	d_k	d_v
6	512	2048	8	64	64



Muti-Head Attention (MHA) Module



Feed-Forward Network (FFN) Module

Figure 1: *Map of the computations performed in (Left) the multi-head attention (MHA) module and (Right) the feed-forward network (FFN) module in the Transformer encoder block.*

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 1: *Configuration parameters for Transformer architectures. Parameters for BERT-Base, BERT-Large, and GPT-2 (smallest) are given as examples. Note that GPT-2 has the same parameters as BERT-Base. Sequence length can be any number, as long as it doesn't exceed the maximum possible sequence length.*

Symbol	Parameter	BERT-Base	BERT-Large	GPT-2
N	# Layers	12	24	12
d	Model dimension	768	1024	768
h	# Attention Heads	12	16	12
d_{FFN}	FFN dimension	3072	4096	3072
l	Sequence length	-	-	-

Table 2: *Linear operations in Transformer models. The last column is the matrix multiplication dimensions, i.e., $m \times n \times k$ means the input dimensions of $m \times n$ and $n \times k$, and the output dimension of $m \times k$. Note that act-to-act matmuls are both repeated h times in the multi-headed scheme. The entire computation graphs of MHA and FFN are illustrated in detail in Fig. 1.*

Module	operation	matmul dim
MHA	W_Q projection	$d \times d \times l$
	W_K projection	$d \times d \times l$
	W_V projection	$d \times d \times l$
	query × key	$l \times d/h \times l$
	attn. score × value	$d/h \times l \times l$
	W_{out} projection	$d \times d \times l$
FFN	W_1 projection	$d_{\text{FFN}} \times d \times l$
	W_2 projection	$d \times d_{\text{FFN}} \times l$

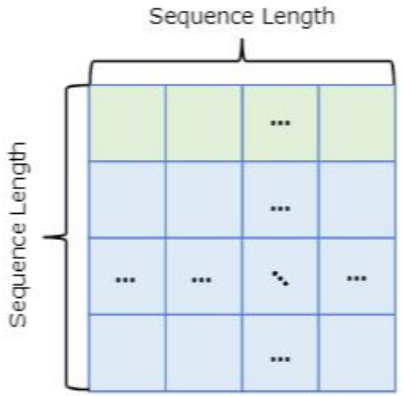
d: also called embedding dimension, or d_model
表示每個token 的embedding vector長度

Nonlinear Operations

(a) Softmax

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Sum of exponential terms is computed along the sequence length

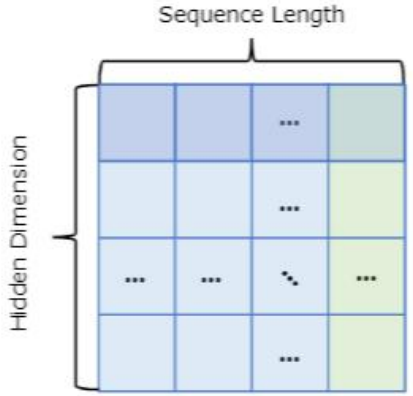


(b) Layer Normalization

$$p_{out} = \frac{p_{in} - \mu_t}{\sigma_t}\gamma_{e+\beta_e}$$

γ, β learned during training and applied along the sequence length

μ, σ computed during inference across the hidden dimension



(c) Batch Normalization

$$p_{out} = \frac{p_{in} - \mu_c}{\sigma_c}\gamma_{c+\beta_c}$$

γ, β learned during training per channel

μ, σ learned during training per channel

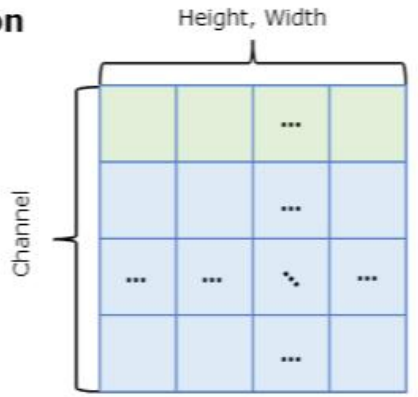
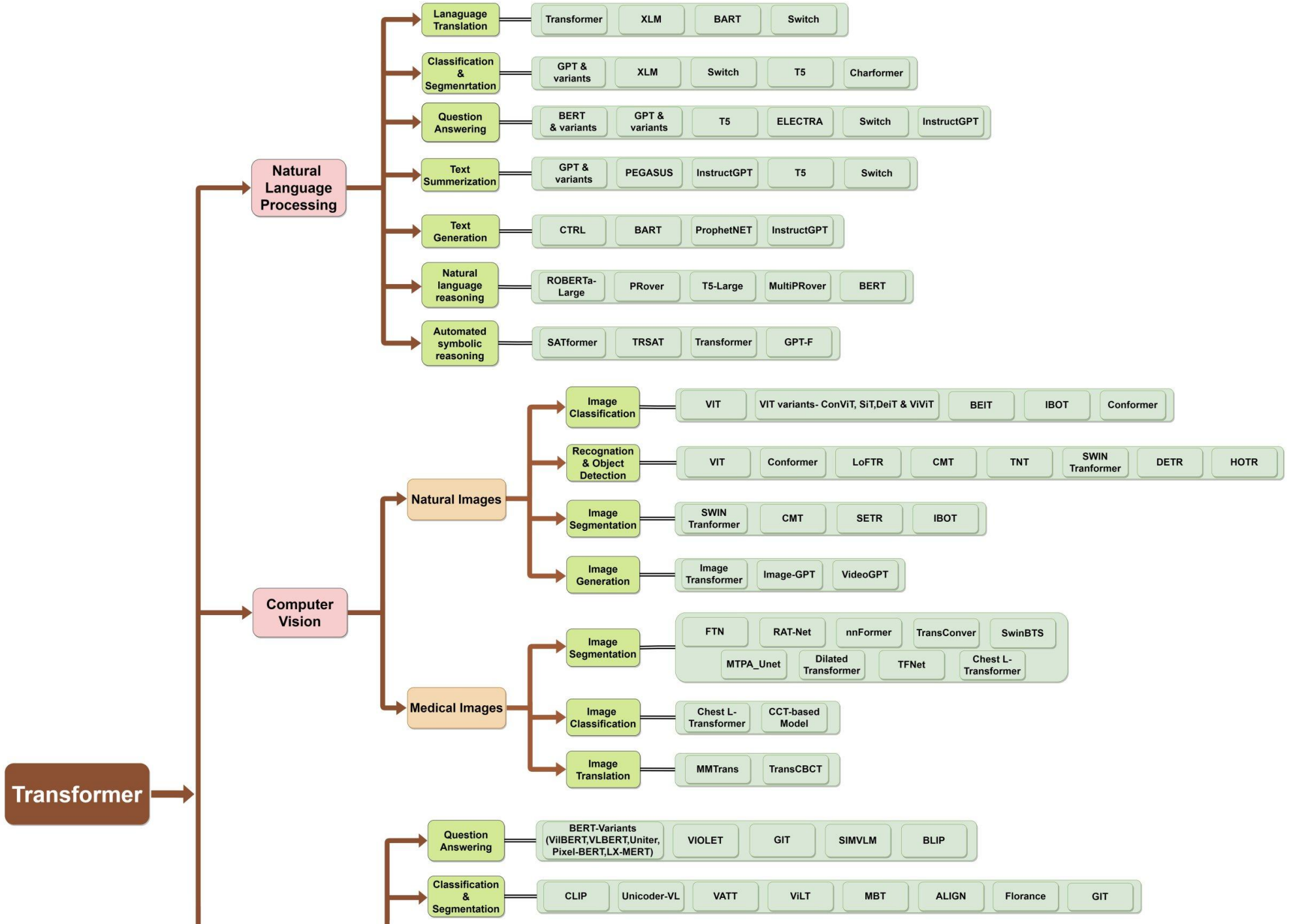
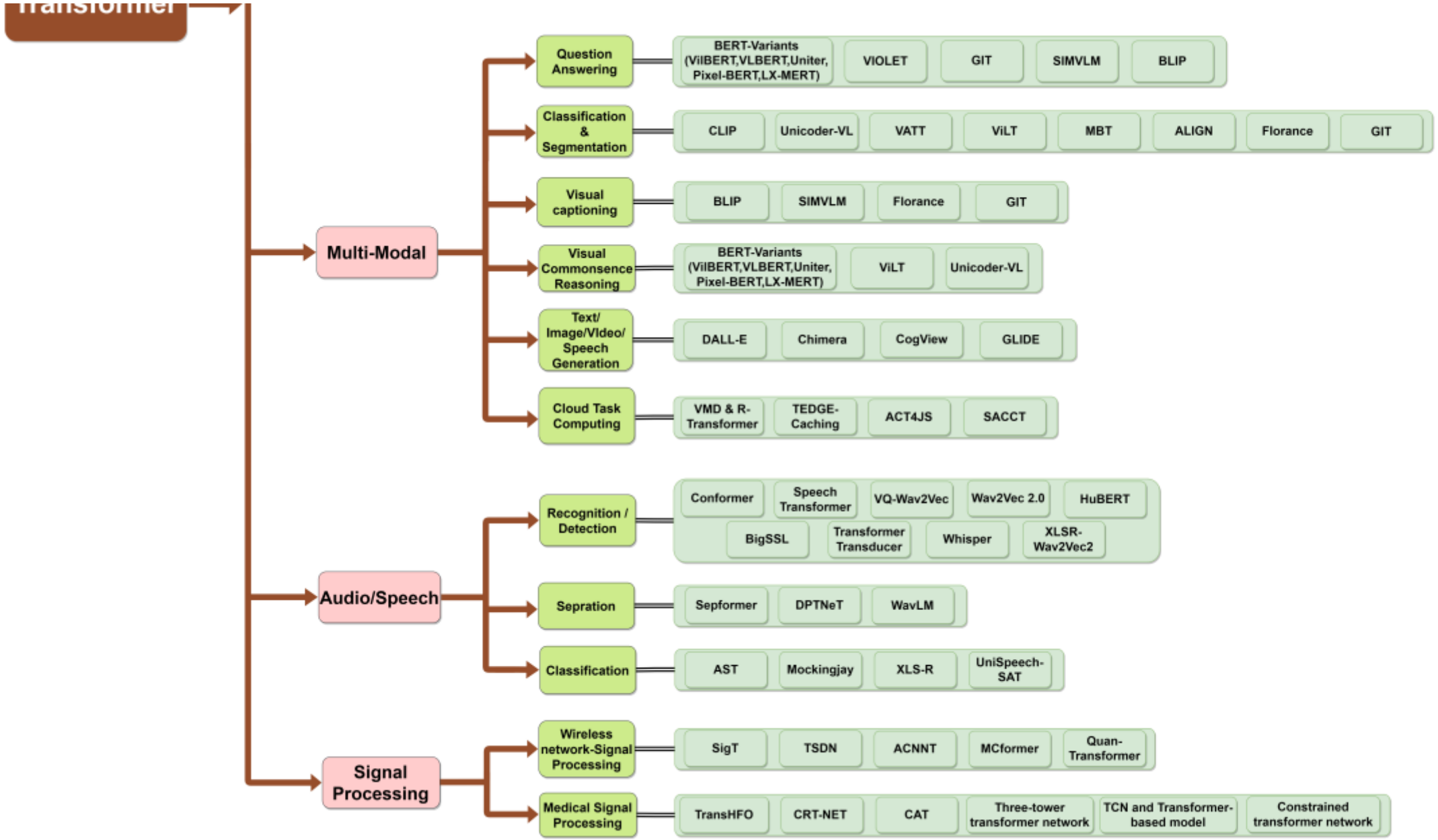


Figure 2: *Diagrams outlining the Softmax, LayerNorm, and BatchNorm operations. Since they rely on runtime statistics, LayerNorm and Softmax both require multiple passes over the input in order to compute the nonlinear operation. In the case of Softmax, a first pass over the inputs is required to compute the denominator. For LayerNorm, three passes are required over the inputs: one to compute the mean; one to compute the standard deviation; and one to apply the normalization. Unlike LayerNorm and Softmax, BatchNorm only uses statistics which are learned during training, and therefore it only requires one pass over the inputs.*


Transformer
Natural Language Processing
Lanaguage Translation
Transformer
XLM
BART
Switch
Classification & Segmenrtation
GPT & variants
XLM
Switch
T5
Charformer
Question Answering
BERT & variants
GPT & variants
T5
ELECTRA
Switch
InstructGPT
Text Summerization
GPT & variants
PEGASUS
InstructGPT
T5
Switch
Text Generation
CTRL
BART
ProphetNET
InstructGPT
Natural language reasoning
ROBERTa-Large
PRover
T5-Large
MultiPRover
BERT
Automated symbolic reasoning
SATformer
TRSAT
Transformer
GPT-F
Computer Vision
Natural Images
Image Classification
VIT
VIT variants- ConViT, SiT,DeiT & ViViT
BEIT
IBOT
Conformer
Recognation & Object Detection
VIT
Conformer
LoFTR
CMT
TNT
SWIN Tranformer
DETR
HOTR
Image Segmentation
SWIN Tranformer
CMT
SETR
IBOT
Image Generation
Image Transformer
Image-GPT
VideoGPT
Medical Images
Image Segmentation
FTN
RAT-Net
nnFormer
TransConver
SwinBTS
MTPA_Unet
Dilated Transformer
TFNet
Chest L-Transformer
Image Classification
Chest L-Transformer
CCT-based Model
Image Translation
MMTrans
TransCBCT
Question Answering
BERT-Variants (VilBERT,VLBERT,Uniter, Pixel-BERT,LX-MERT)
VIOLET
GIT
SIMVLM
BLIP
Classification & Segmentation
CLIP
Unicoder-VL
VATT
ViLT
MBT
ALIGN
Florance
GIT


Transformer
Multi-Modal
Question Answering
BERT-Variants (VilBERT,VLBERT,Uniter, Pixel-BERT,LX-MERT)
VIOLET
GIT
SIMVLM
BLIP
Classification & Segmentation
CLIP
Unicoder-VL
VATT
ViLT
MBT
ALIGN
Florance
GIT
Visual captioning
BLIP
SIMVLM
Florance
GIT
Visual Commonsence Reasoning
BERT-Variants (VilBERT,VLBERT,Uniter, Pixel-BERT,LX-MERT)
ViLT
Unicoder-VL
Text/ Image/Video/ Speech Generation
DALL-E
Chimera
CogView
GLIDE
Cloud Task Computing
VMD & R-Transformer
TEDGE-Caching
ACT4JS
SACCT
Audio/Speech
Recognition / Detection
Conformer
Speech Transformer
VQ-Wav2Vec
Wav2Vec 2.0
HuBERT
BigSSL
Transformer Transducer
Whisper
XLSR-Wav2Vec2
Sepration
Sepformer
DPTNeT
WavLM
Classification
AST
Mockingjay
XLS-R
UniSpeech-SAT
Signal Processing
Wireless network-Signal Processing
SigT
TSDN
ACNNT
MCformer
Quan-Transformer
Medical Signal Processing
TransHFO
CRT-NET
CAT
Three-tower transformer network
TCN and Transformer-based model
Constrained transformer network

- https://github.com/jadore801120/attention-is-all-you-need-pytorch