COS 484/584

(Advanced) Natural Language Processing

L18: Self-Attention and Transformers

Spring 2021

(Some slides adapted from Stanford CS224N)

Issues with RNNs?

- Sequential nature $\Longrightarrow$ difficult to parallelize

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \in \mathbb{R}^h$$

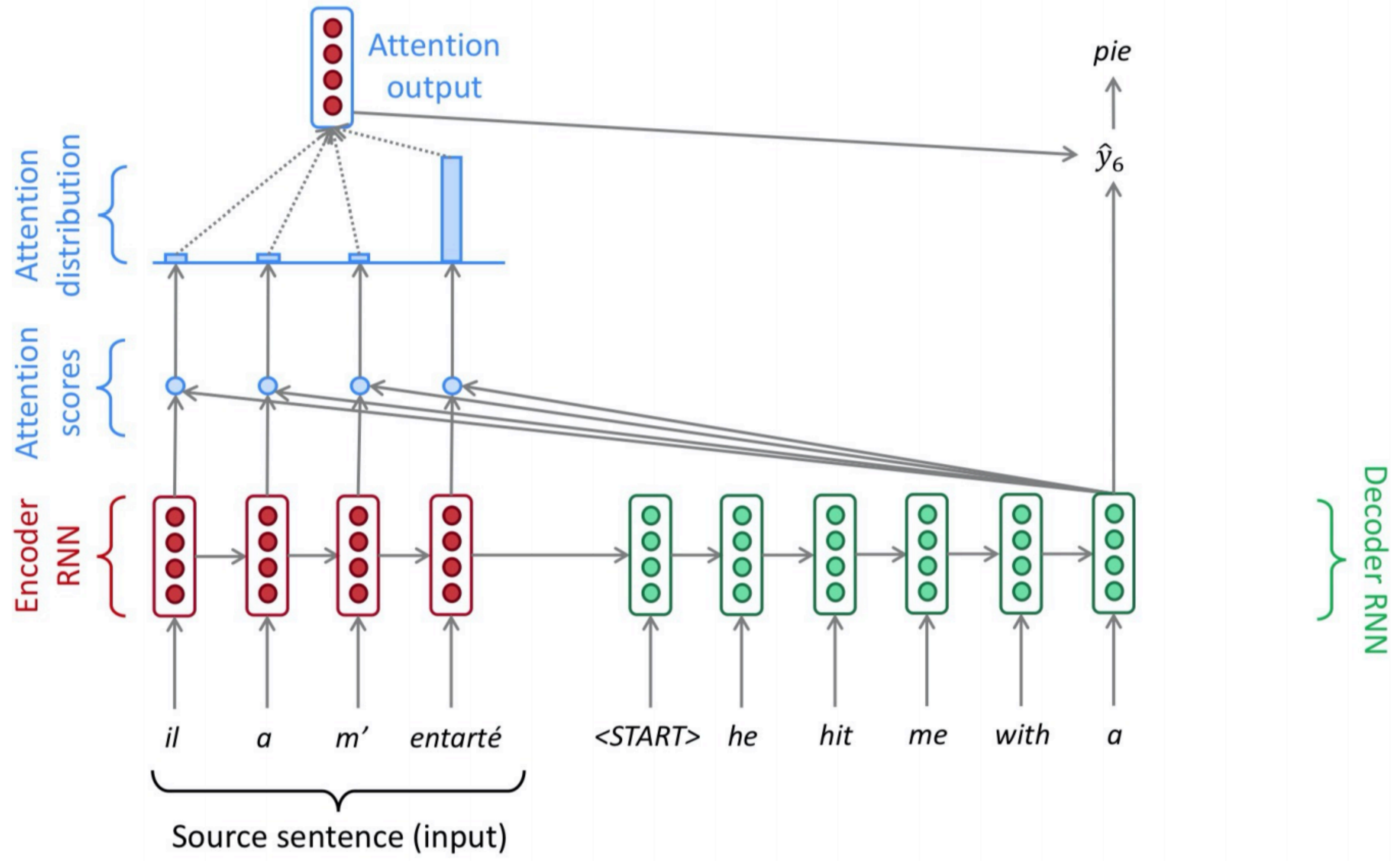
LSTMs

- Input gate **(how much to write)**:
  $\mathbf{i}_t = \sigma(\mathbf{W}^i \mathbf{h}_{t-1} + \mathbf{U}^i \mathbf{x}_t + \mathbf{b}^i) \in \mathbb{R}^h$
- Forget gate **(how much to erase)**:
  $\mathbf{f}_t = \sigma(\mathbf{W}^f \mathbf{h}_{t-1} + \mathbf{U}^f \mathbf{x}_t + \mathbf{b}^f) \in \mathbb{R}^h$
- Output gate **(how much to reveal)**:
  $\mathbf{o}_t = \sigma(\mathbf{W}^o \mathbf{h}_{t-1} + \mathbf{U}^o \mathbf{x}_t + \mathbf{b}^o) \in \mathbb{R}^h$
- New memory cell **(what to write)**:
  $\mathbf{g}_t = \tanh(\mathbf{W}^g \mathbf{h}_{t-1} + \mathbf{U}^g \mathbf{x}_t + \mathbf{b}^g) \in \mathbb{R}^h$
- Final memory cell: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$
- Final hidden cell: $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$

Issues with RNNs?

- Longer sequences can lead to vanishing gradients $\Longrightarrow$ It is hard to capture long-distance information

Attention is the key to solving the problem!



This lecture

- Do we really need RNNs to model the arbitrary context?
- Maybe attention is all you need!

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

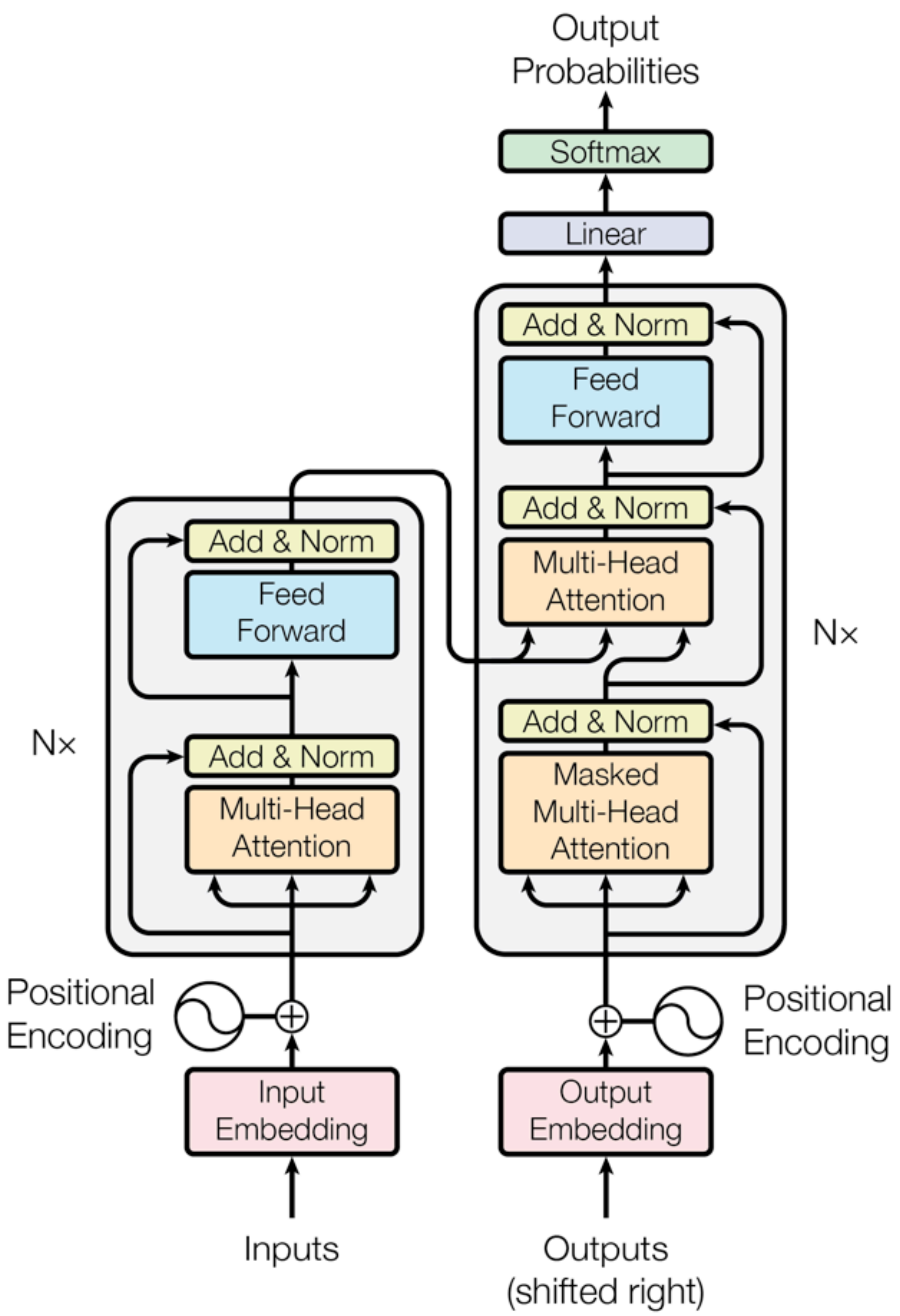
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Vaswani et al., 2017: Attention Is All You Need

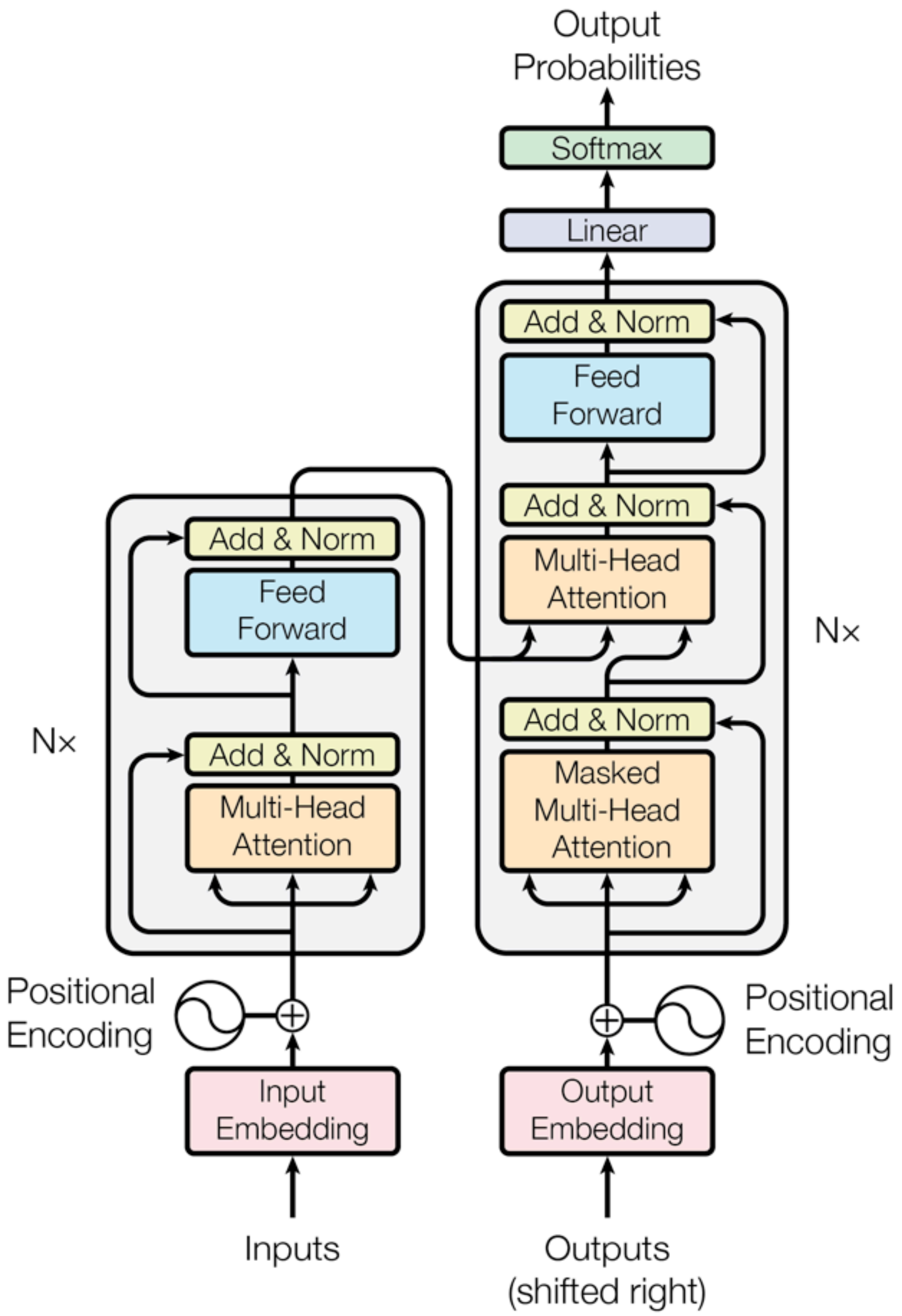
Transformers



- Consists of an encoder and a decoder
- Originally proposed for neural machine translation and later adapted for almost all the NLP tasks
  - For example, BERT only uses the **encoder** of the Transformer architecture (next lecture)
- Both encoder and decoder consist of *N* layers
  - Each encoder layer has two sub-layers
  - Each decoder layer has three sublayers
  - Key innovation: **multi-head self-attention**

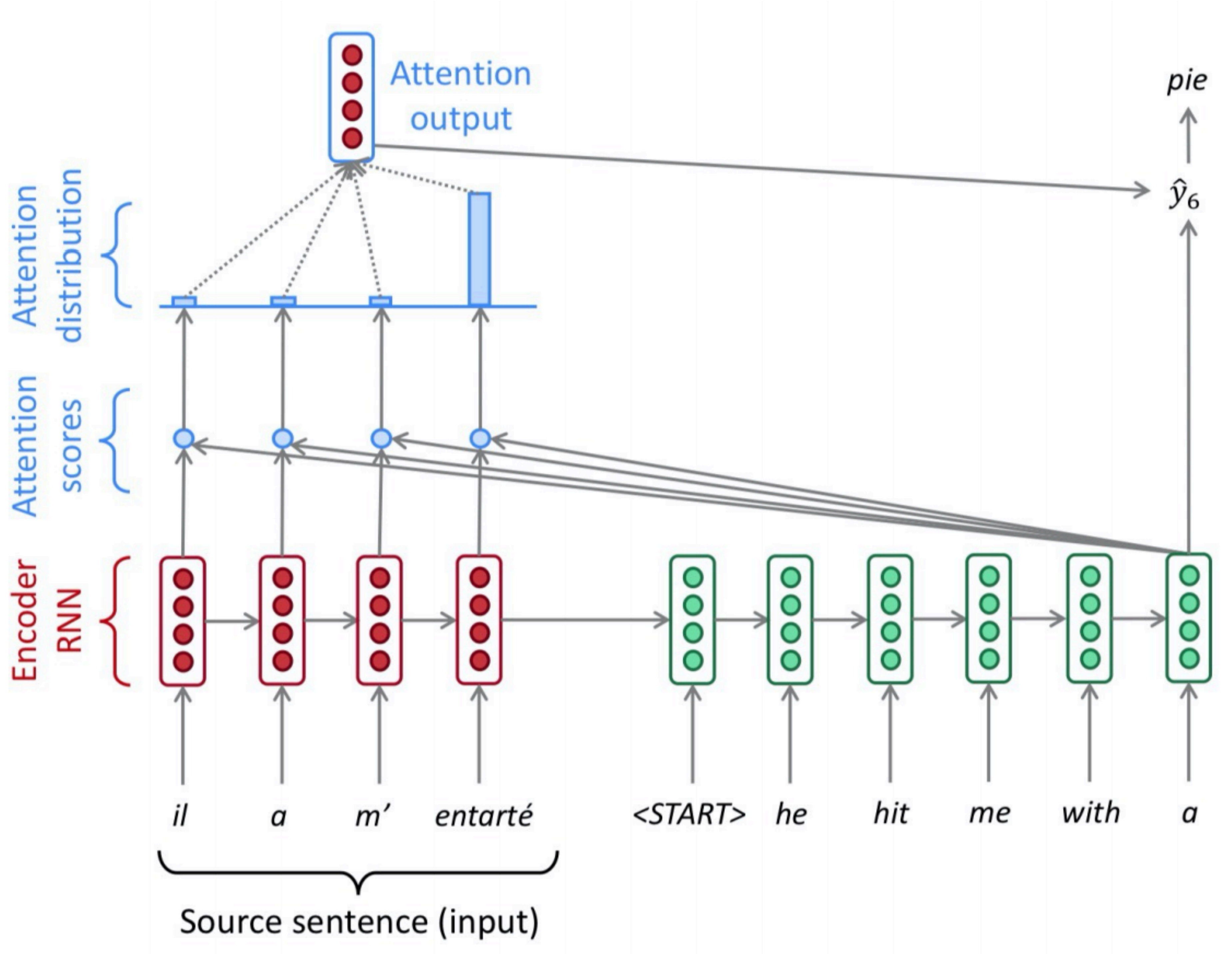
Vaswani et al., 2017: Attention Is All You Need

Transformers: roadmap



- From attention to self-attention
- From self-attention to multi-head self-attention
- Transformer encoder
- Transformer decoder
- Putting the pieces together

Recap: Attention in NMT



- Encoder hidden states: $h_1^{enc}, \ldots, h_n^{enc}$
- Decoder hidden state at time t: h_t^{dec}
- First, get attention scores for this time step of decoder (we'll define g soon):

$$e^t = [g(h_1^{enc}, h_t^{dec}), \ldots, g(h_n^{enc}, h_t^{dec})]$$

g(·) takes dot product in the simplest form!

- Obtain the attention distribution using softmax:

$$\alpha^t = \mathrm{softmax}(e^t) \in \mathbb{R}^n$$

- Compute weighted sum of encoder hidden states:

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

Attention is a *general* deep learning technique

- 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**.
  - In the NMT case, each decoder hidden state (**query**) attends to all the encoder hidden states (**values**).

- 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**).

Attention is a *general* deep learning technique

- Assume that we have a set of **values** $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^{d_v}$ and a **query** vector $\mathbf{q} \in \mathbb{R}^{d_q}$

- Attention always involves the following steps:
  - Computing the **attention scores** $\mathbf{e} = g(\mathbf{v}_i, \mathbf{q}) \in \mathbb{R}^n$
  - Taking softmax to get **attention distribution** α:

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

  - Using attention distribution to take **weighted sum** of values:

$$\mathbf{a} = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i \in \mathbb{R}^{d_v}$$

- A more general form: use a set of **keys** and **values** $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_n, \mathbf{v}_n)$, $\mathbf{k}_i \in \mathbb{R}^{d_k}$, $\mathbf{v}_i \in \mathbb{R}^{d_v}$, **keys** are used to compute the attention scores and **values** are used to compute the output vector

Attention is a *general* deep learning technique

- Assume that we have a set of **key**-**value** pairs $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_n, \mathbf{v}_n), \mathbf{k}_i \in \mathbb{R}^{d_k}, \mathbf{v}_i \in \mathbb{R}^{d_v}$ and a **query** vector $\mathbf{q} \in \mathbb{R}^{d_q}$

- Attention always involves the following steps:
  - Computing the **attention scores** $\mathbf{e} = g(\mathbf{k}_i, \mathbf{q}) \in \mathbb{R}^n$
  - Taking softmax to get **attention distribution** α:

$$\alpha = \mathrm{softmax}(\mathbf{e}) \in \mathbb{R}^n$$

  - Using attention distribution to take **weighted sum** of values:

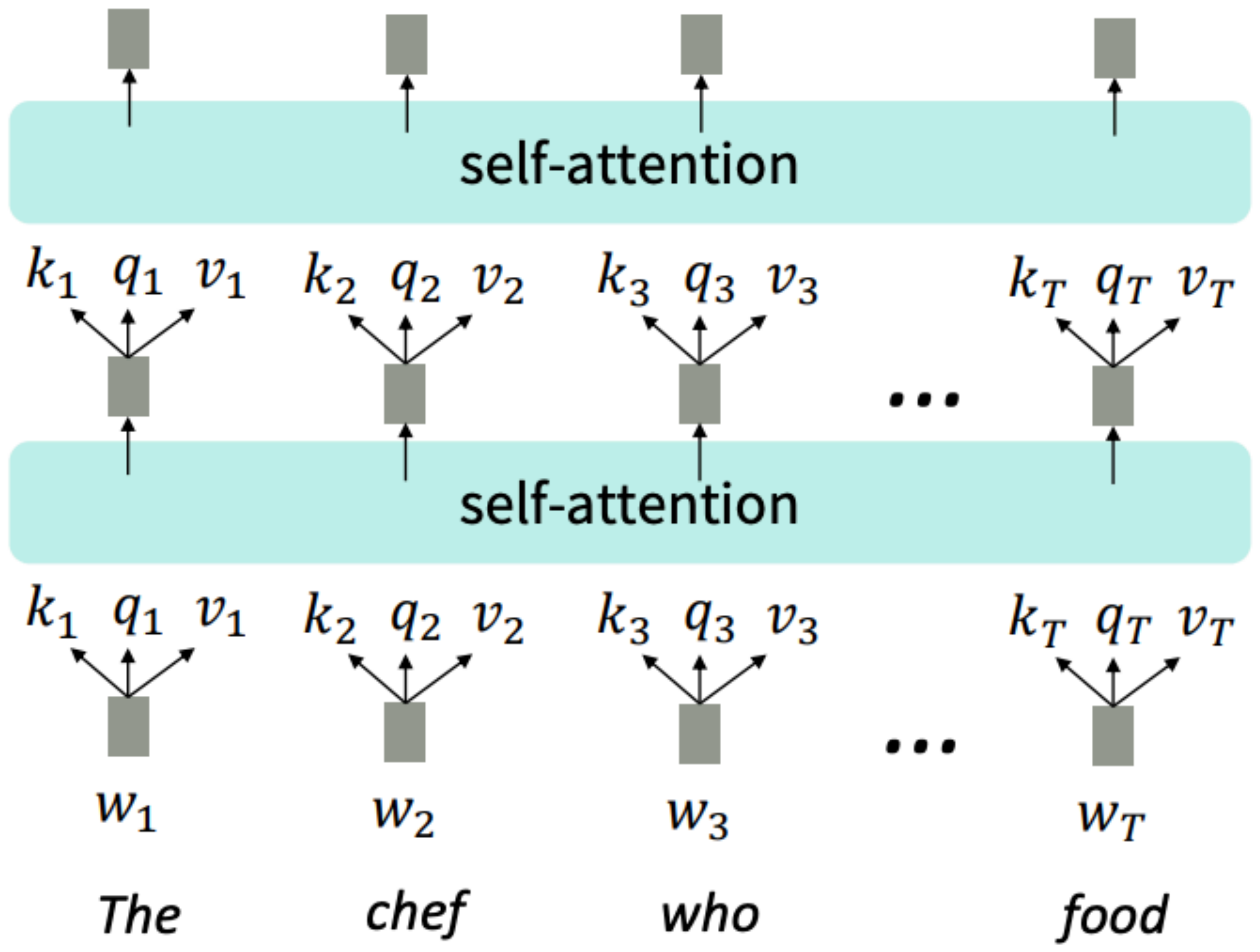
$$\mathbf{a} = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i \in \mathbb{R}^{d_v}$$

Self-attention

- We saw attention from the decoder (query) to the encoder (values), now we think about **attention within one single sequence**.
    - Self-attention = attention from the sequence to itself

- Self-attention: let's use each word in a sequence as the **query**, and all the other words in the sequence as **keys** and **values**.

- The queries, keys and values are drawn from the same source.



Self-attention doesn't know the order of the inputs - we will come back to this later!

Self-attention in equations

- A self-attention layer maps a sequence of input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors: $\mathbf{y}_1, \ldots, \mathbf{y}_n \in \mathbb{R}^{d_2}$
  - The same abstraction as RNNs - can be used as a drop-in replacement for an RNN layer

- First, construct a set of queries, keys and values:

$$\mathbf{q}_i = W^Q \mathbf{x}_i, \mathbf{k}_i = W^K \mathbf{x}_i, \mathbf{v}_i = W^V \mathbf{x}_i$$

$$W^Q \in \mathbb{R}^{d_q \times d_1}, W^K \in \mathbb{R}^{d_k \times d_1}, W^V \in \mathbb{R}^{d_v \times d_1}$$

- Second, for each $\mathbf{q}_i$, compute attention scores and attention distribution:

$$\alpha_{i,j} = \mathrm{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right)$$

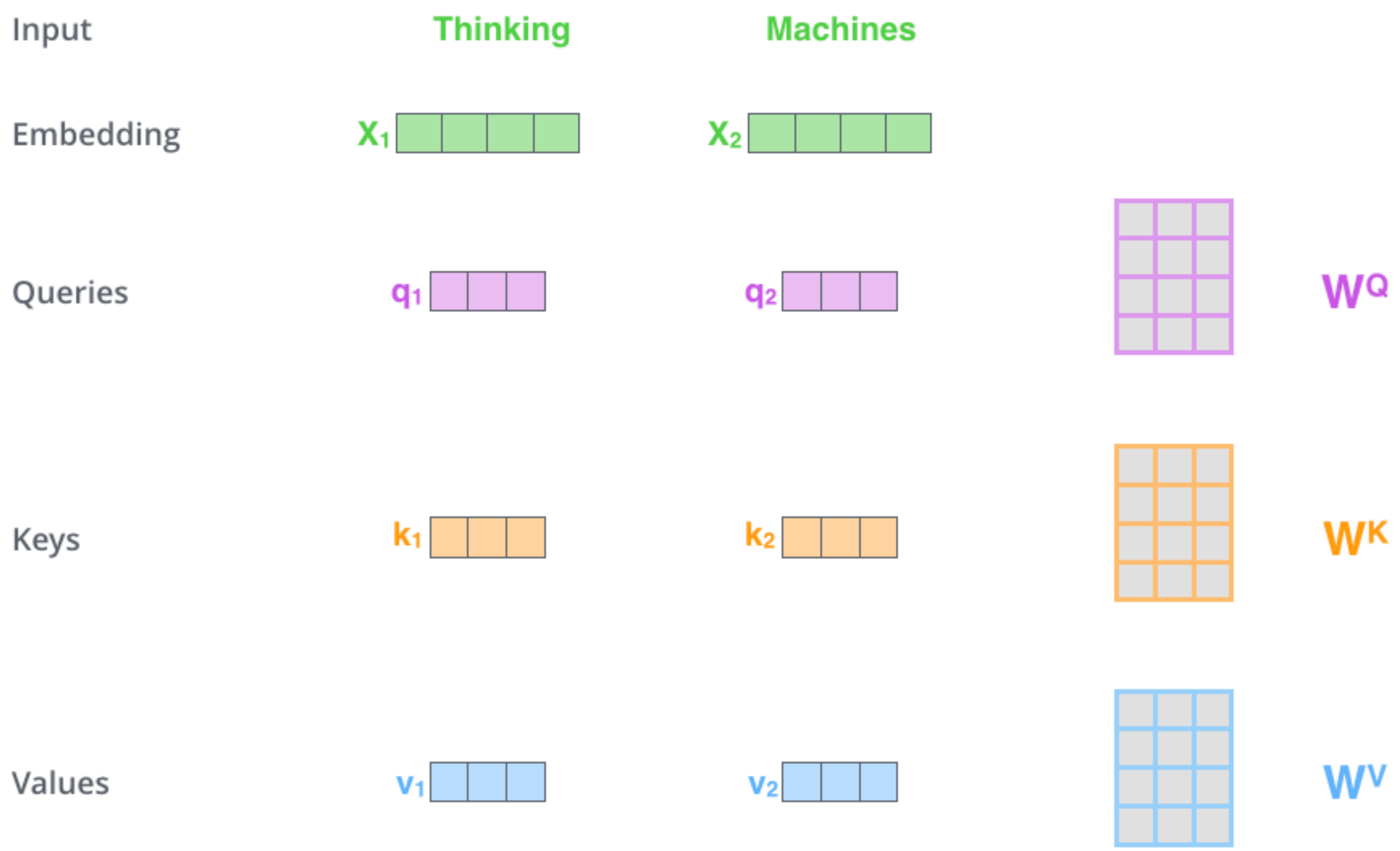
← aka. "scaled dot product"
It must be $d_q = d_k$ in this case

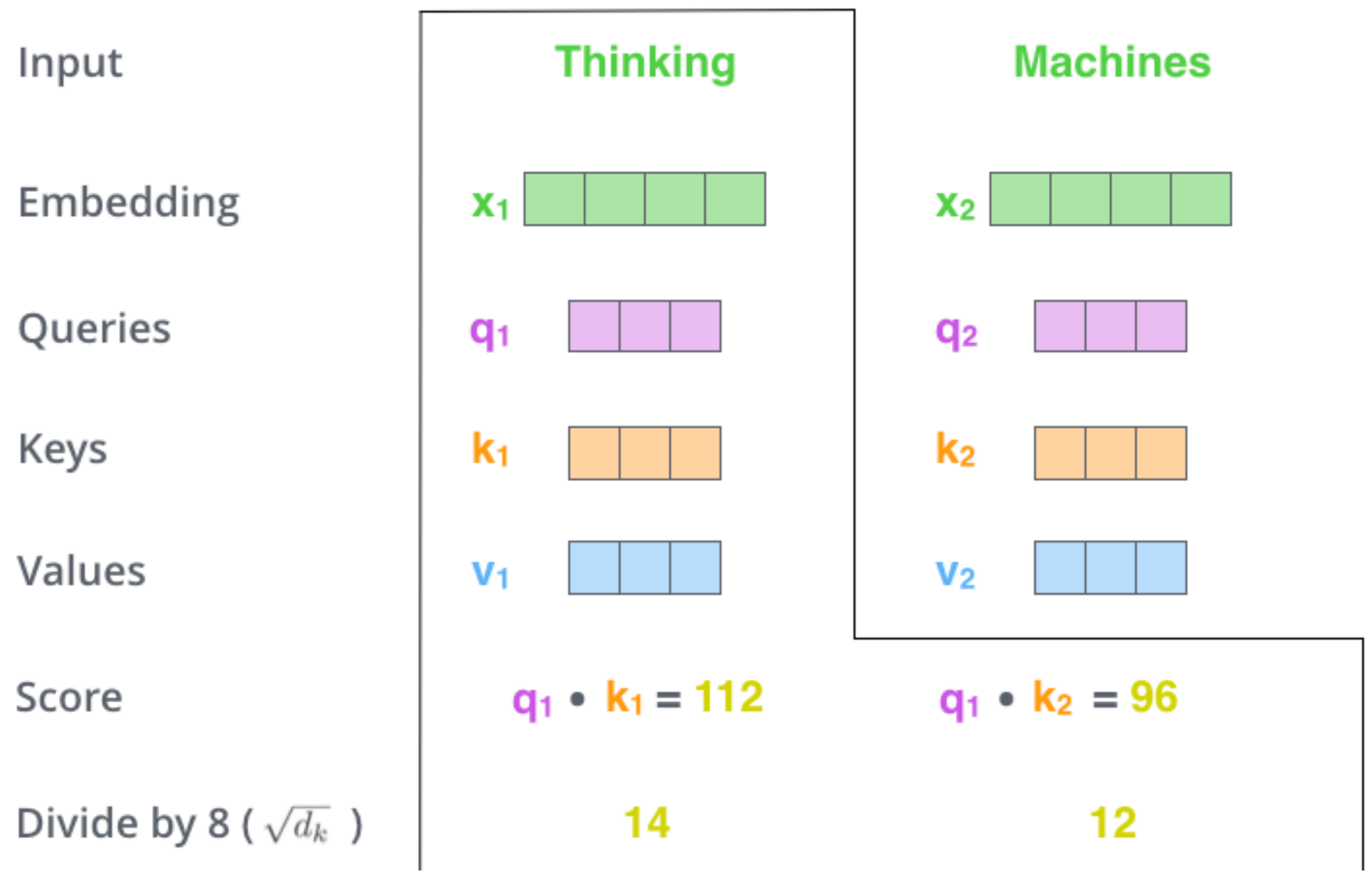
- Finally, compute the weighted sum:

$$\mathbf{y}_i = \sum_{j=1}^{n} \alpha_{i,j} \mathbf{v}_j \in \mathbb{R}^{d_v}$$

$(d_v = d_2)$

Self-attention: illustration





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

Zoom poll

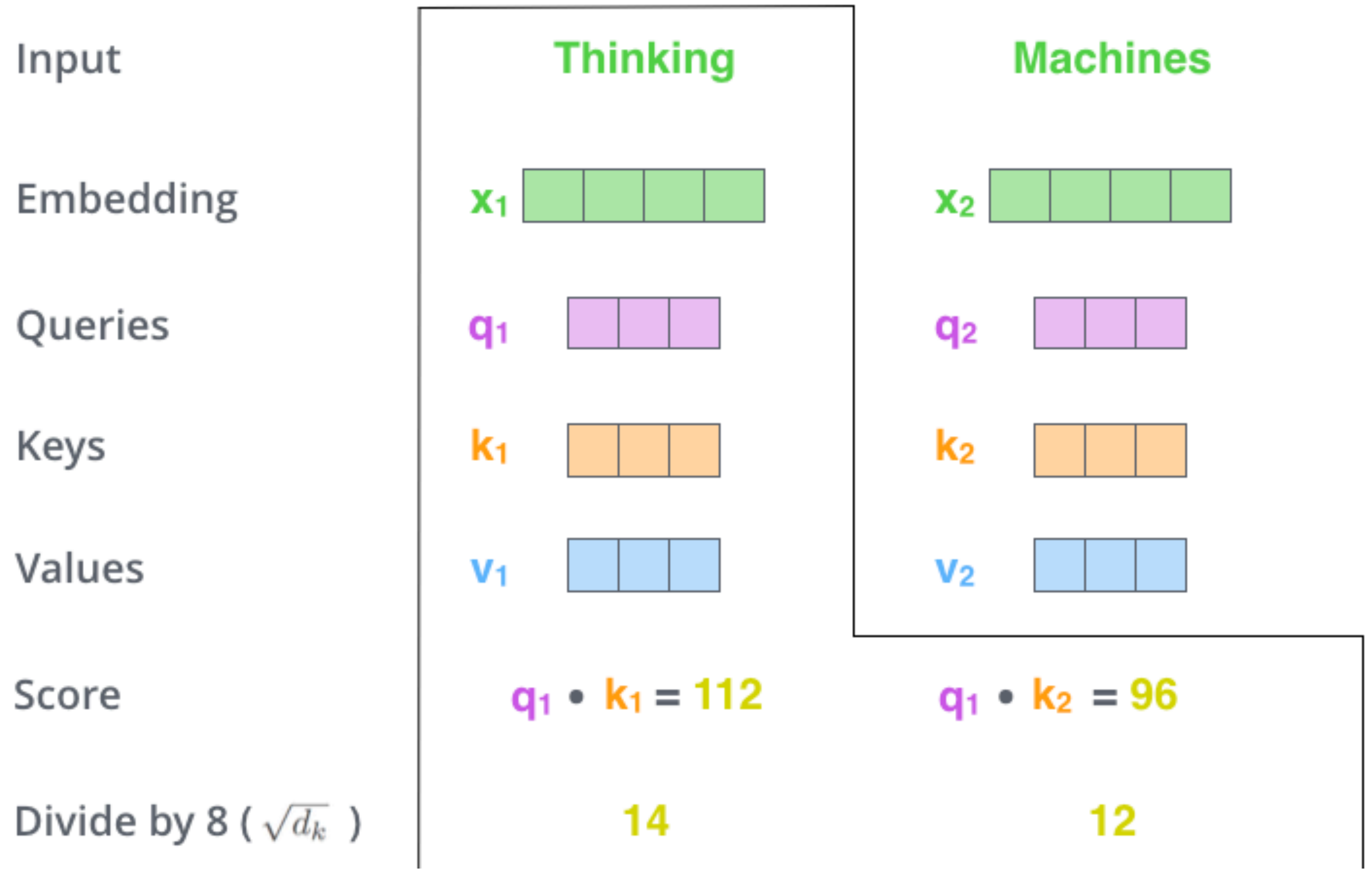
What would be the output vector for the word “Thinking” approximately?

(a) $0.5\mathbf{v}_1 + 0.5\mathbf{v}_2$

(b) $0.54\mathbf{v}_1 + 0.46\mathbf{v}_2$

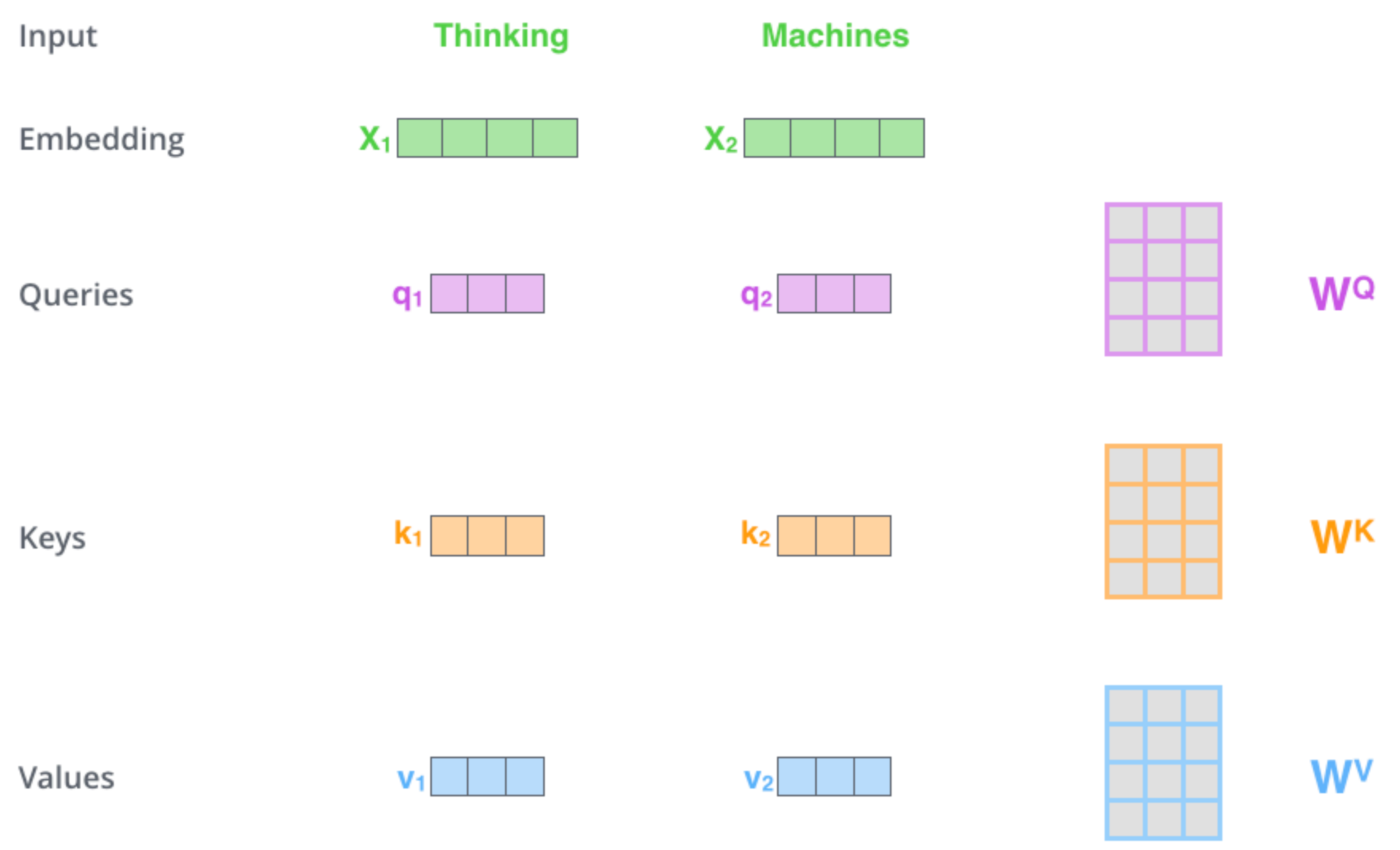
(c) $0.88\mathbf{v}_1 + 0.12\mathbf{v}_2$

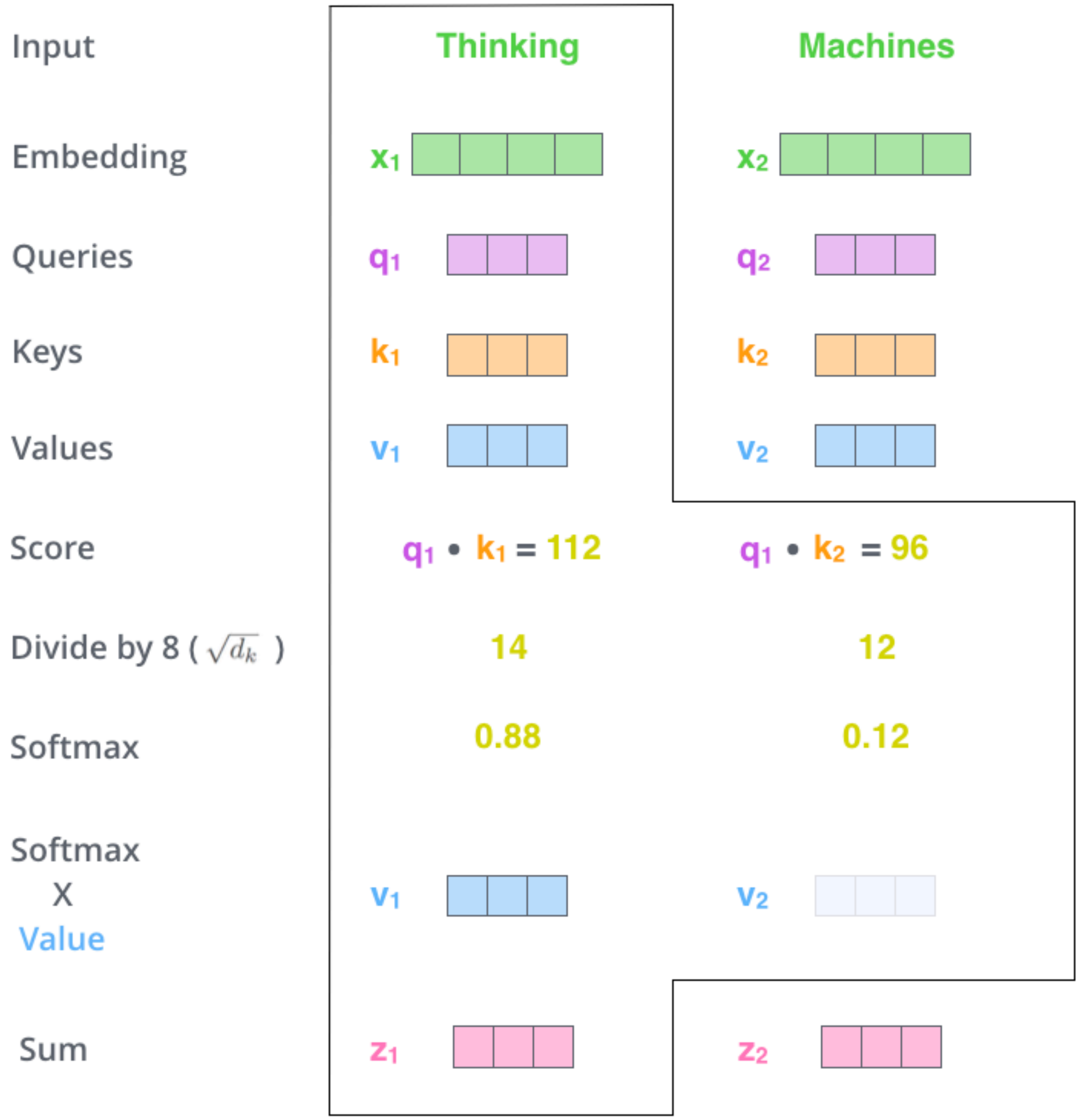
(d) $0.12\mathbf{v}_1 + 0.88\mathbf{v}_2$



(c) is correct.

Self-attention: illustration





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

Self-attention: matrix notations

$X \in \mathbb{R}^{n \times d_1}$

Note: the notations we use here are following the original paper (= the transpose of the matrices in previous notations)

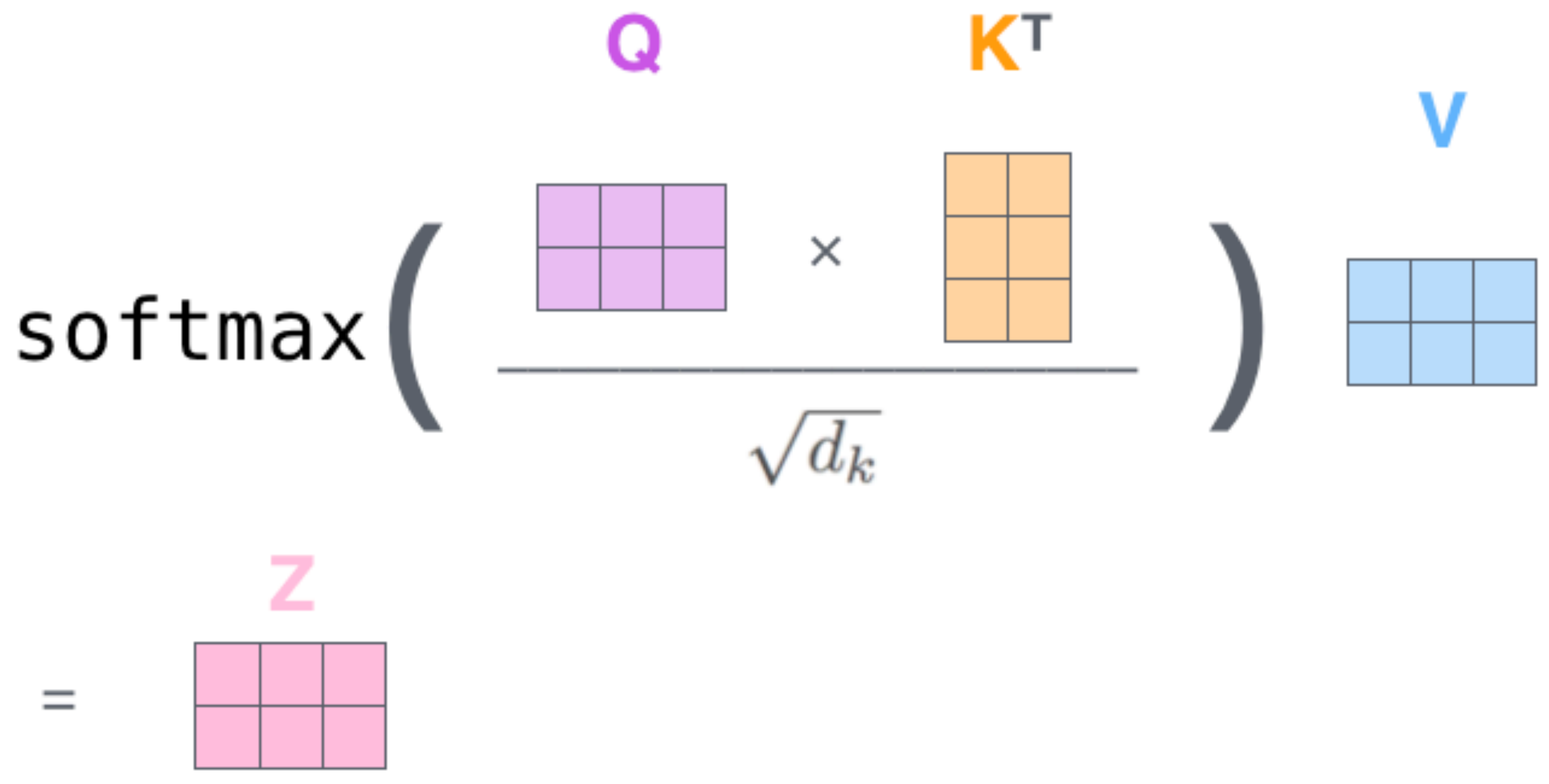
$Q = XW^Q \quad K = XW^K \quad V = XW^V$

$W^Q \in \mathbb{R}^{d_1 \times d_q}, W^K \in \mathbb{R}^{d_1 \times d_k}, W^V \in \mathbb{R}^{d_1 \times d_v}$

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

Q: $n \times d_q$; K^T: $d_k \times n$; V: $n \times d_v$

Q: What is this softmax operation?



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

hardmaru
@hardmaru

The most important formula in deep learning after 2018

Self-Attention

What is self-attention? Self-attention calculates a weighted average of feature representations with the weight proportional to a similarity score between pairs of representations. Formally, an input sequence of n tokens of dimensions d, $X \in \mathbf{R}^{n \times d}$, is projected using three matrices $W_Q \in \mathbf{R}^{d \times d_q}$, $W_K \in \mathbf{R}^{d \times d_k}$, and $W_V \in \mathbf{R}^{d \times d_v}$ to extract feature representations Q, K, and V, referred to as query, key, and value respectively with $d_k = d_q$. The outputs Q, K, V are computed as

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V. \tag{1}$$

So, self-attention can be written as,

$$S = D(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_q}}\right)V, \tag{2}$$

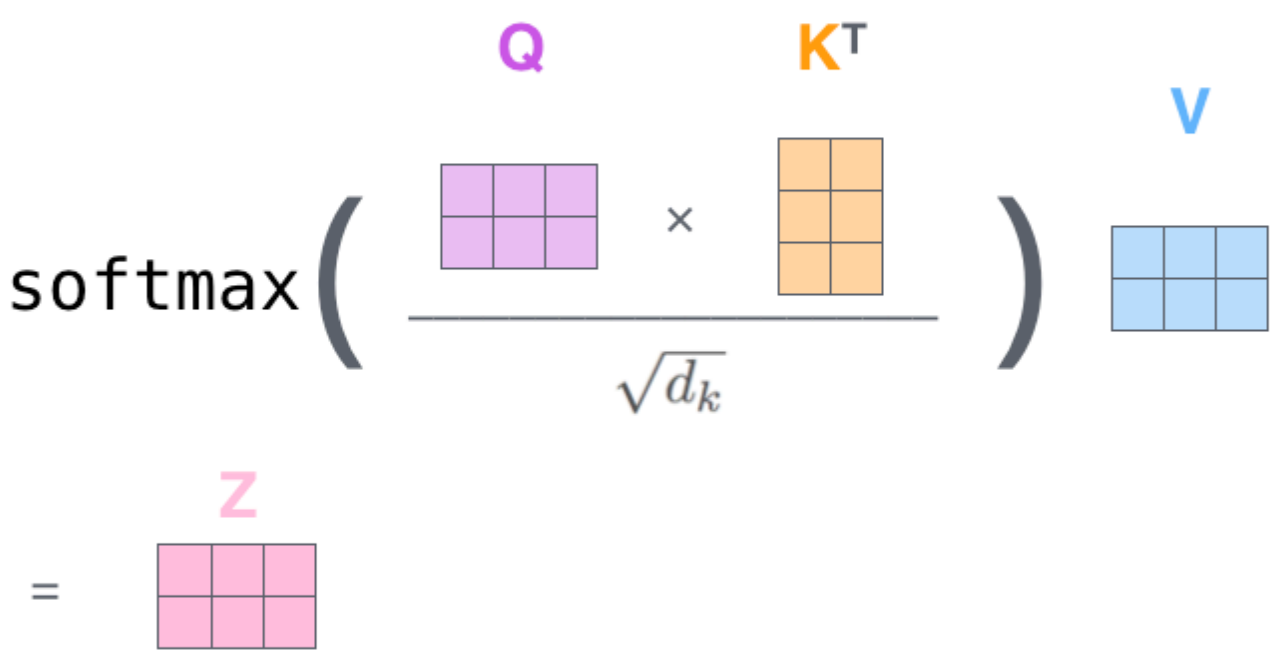
where softmax denotes a *row-wise* softmax normalization function. Thus, each element in S depends on all other elements in the same row.

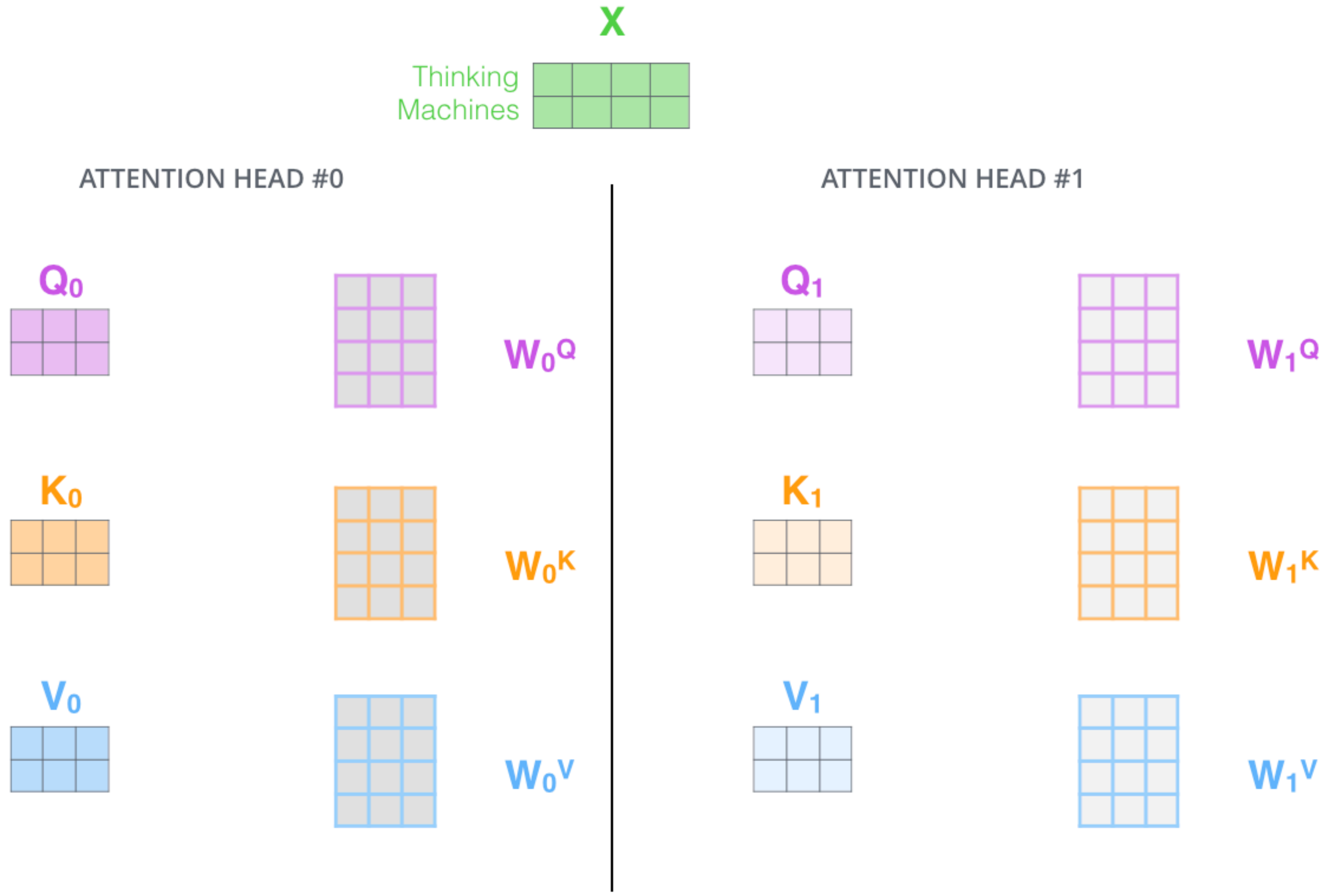
9:08 PM · Feb 9, 2021 · Twitter Web App

580 Retweets **38** Quote Tweets **3,407** Likes

Multi-head attention

- It is better to use multiple attention functions instead of one!
  - Each attention function ("head") can focus on different positions.
- How to do this? Use different sets of query, key and value matrices!

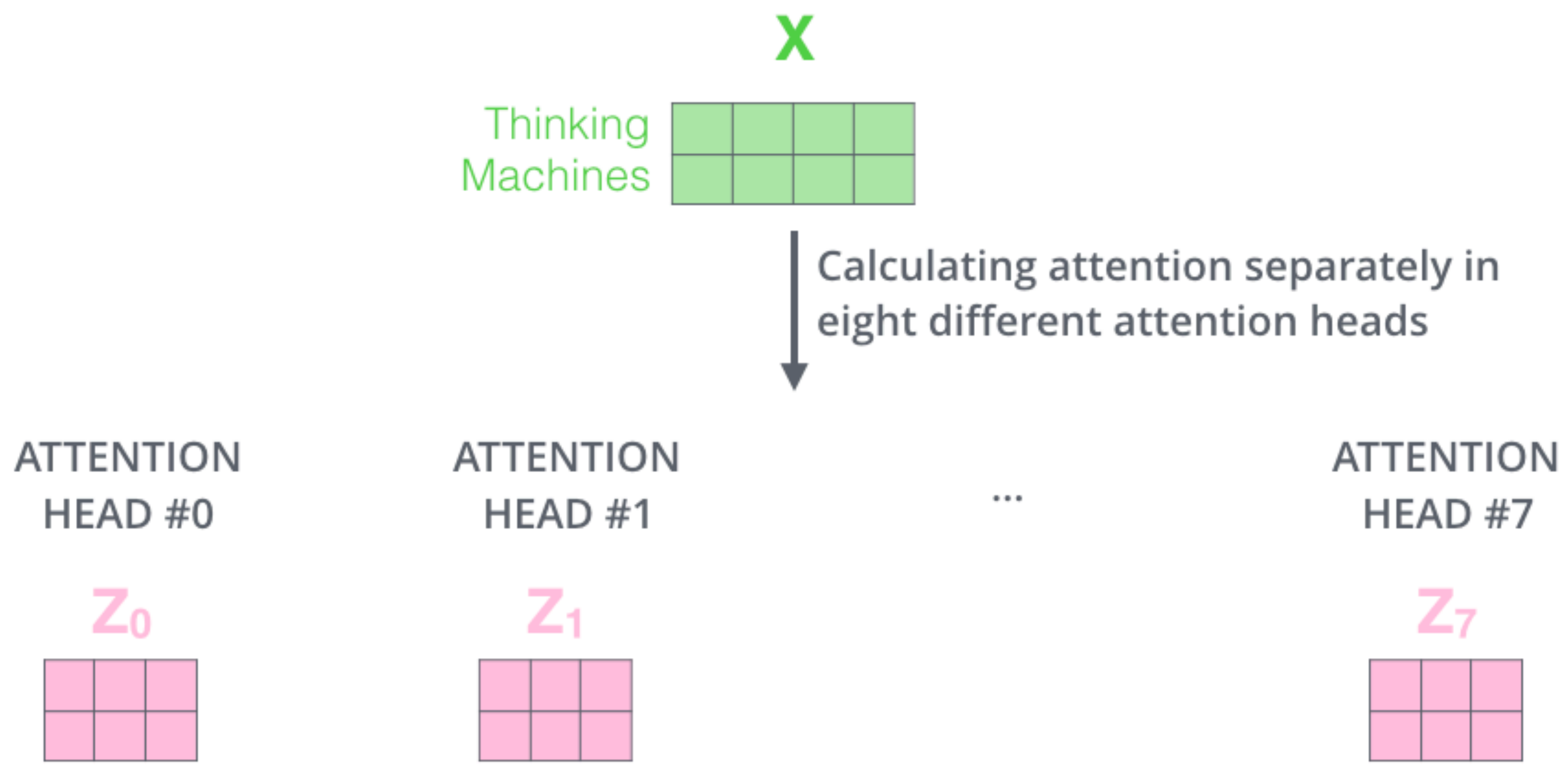




http://jalammar.github.io/illustrated-transformer/

Multi-head attention

- It is better to use multiple attention functions instead of one!



- Finally, we just concatenate all the heads and apply an output projection matrix.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

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

Multi-head attention

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

- In practice, we use a *reduced* dimension for each head.

$$W_i^Q \in \mathbb{R}^{d_1 \times d_q}, W_i^K \in \mathbb{R}^{d_1 \times d_k}, W_i^V \in \mathbb{R}^{d_1 \times d_v}$$

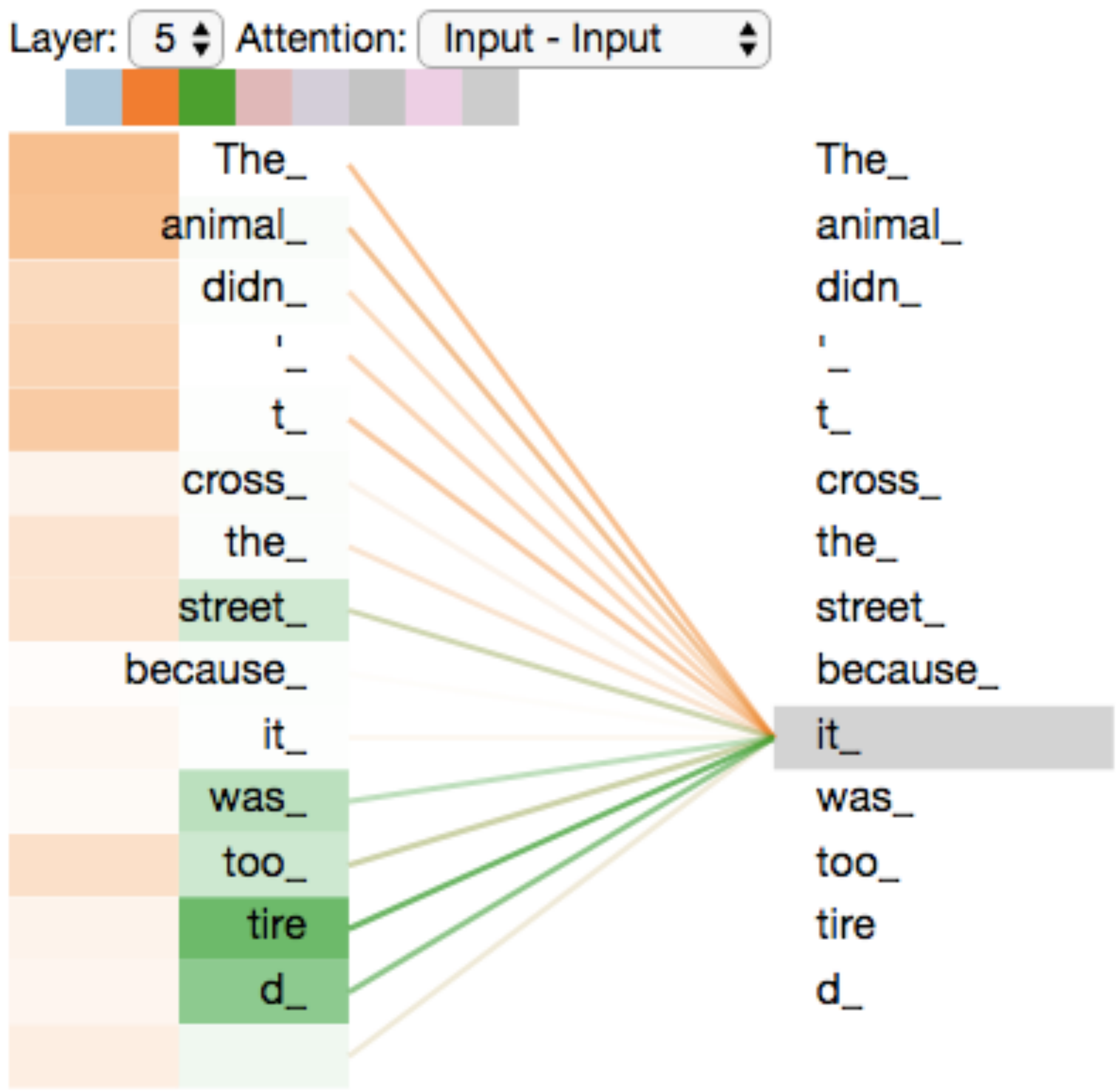
$$d_q = d_k = d_v = d/h$$ d = hidden size, h = # of heads

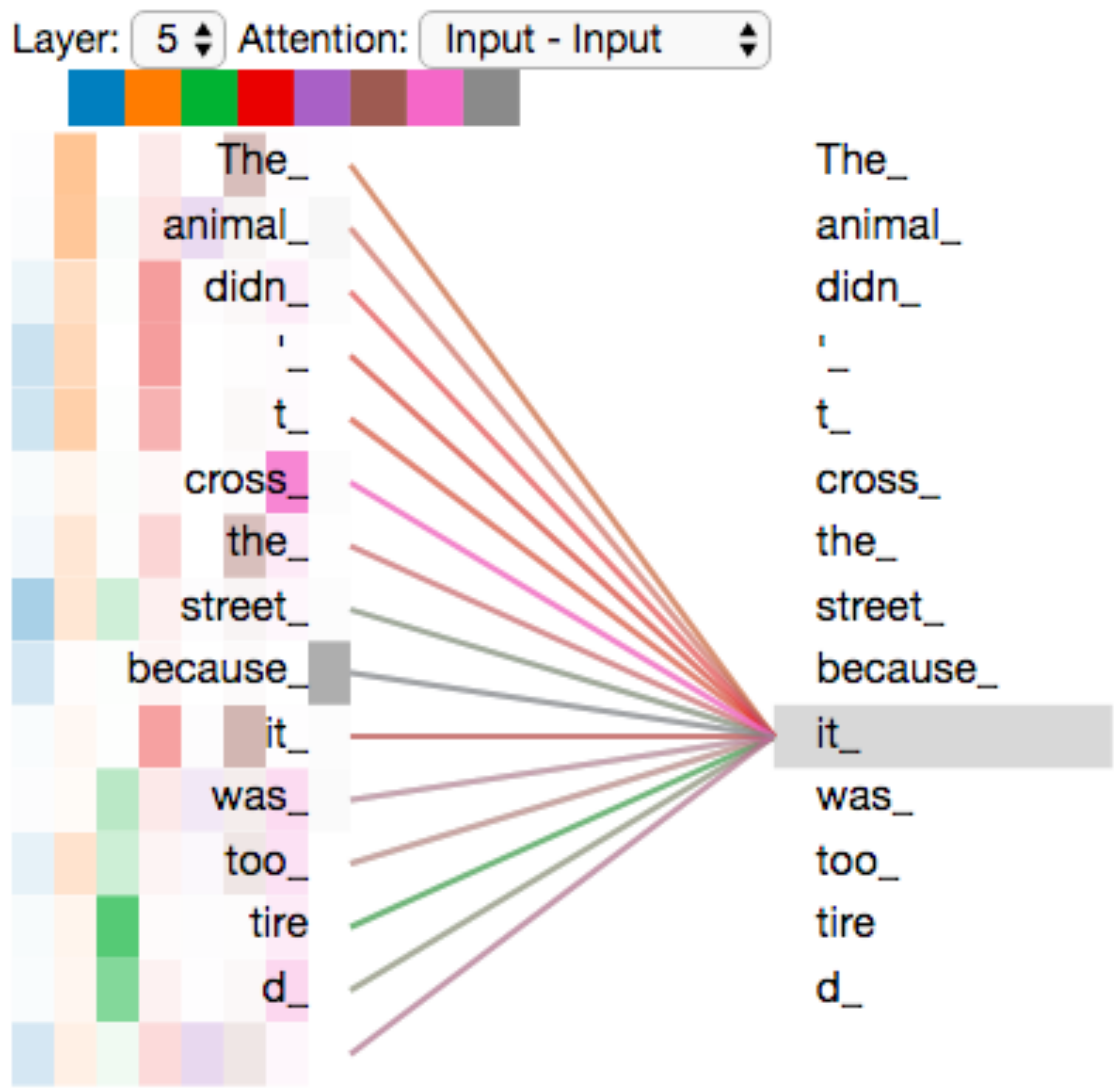
$$W^O \in \mathbb{R}^{d \times d_2}$$ If we stack multiple layers, usually $d_1 = d_2 = d$

- The total computational cost is similar to that of single-head attention with full dimensionality.

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

What does multi-head attention learn?





https://github.com/jessevig/bertviz

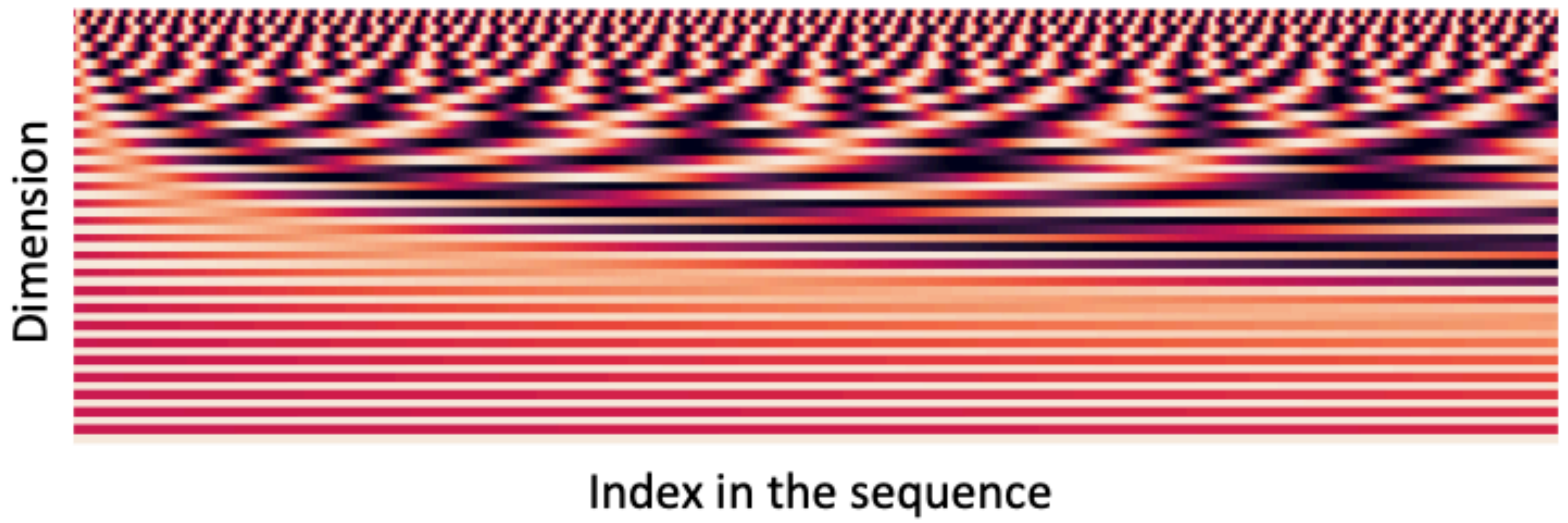
Missing piece: positional information!

- Unlike RNNs, self-attention doesn't build in order information, we need to encode the order of the sentence.
- Solution: Add **"positional encoding"** to the input embeddings

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{p}_i$$

- Use sine and cosine functions of different frequencies (not learnable):

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



- Later, people just use a learnable embedding $\mathbf{p}_i \in \mathbb{R}^{d_1}$ for every unique position.

Adding nonlinearities

- There is no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors

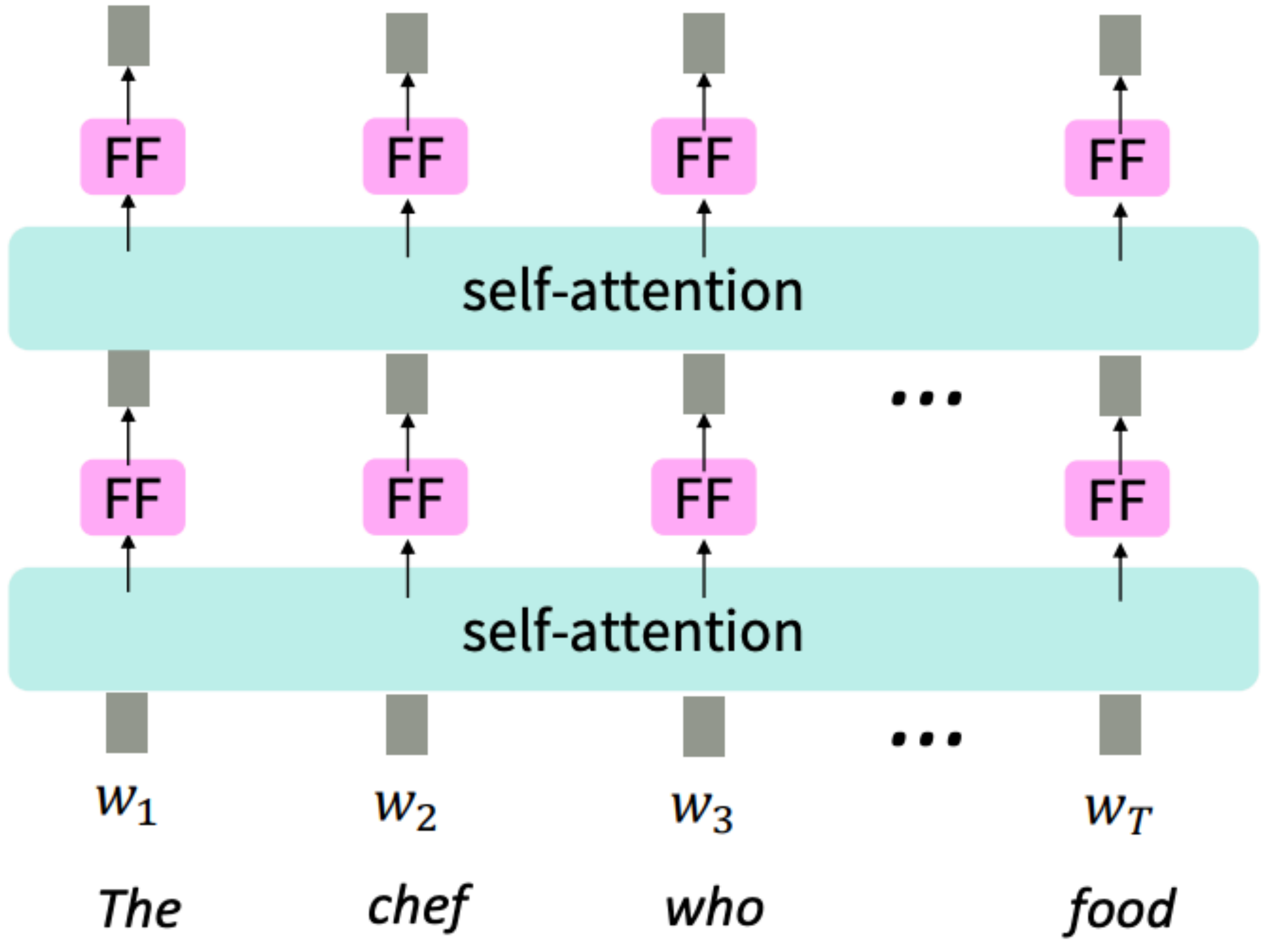
- Simple fix: add a feed-forward network to post-process each output vector

$$\mathrm{FFN}(\mathbf{x}_i) = W_2 \mathrm{ReLU}(W_1 \mathbf{x}_i + \mathbf{b}_1) + \mathbf{b}_2$$

$$W_1 \in \mathbb{R}^{d_{ff} \times d}, \mathbf{b}_1 \in \mathbb{R}^{d_{ff}}$$

$$W_2 \in \mathbb{R}^{d \times d_{ff}}, \mathbf{b}_2 \in \mathbb{R}^{d}$$

In practice, they use $d_{ff} = 4d$



Zoom poll

Which of the following statements is correct?

(a) Transformers run faster than LSTMs

(b) Transformers are easier to parallelize compared to LSTMs

(c) Transformers have less parameters compared to LSTMs

(d) Transformers are better at capturing positional information than LSTMs

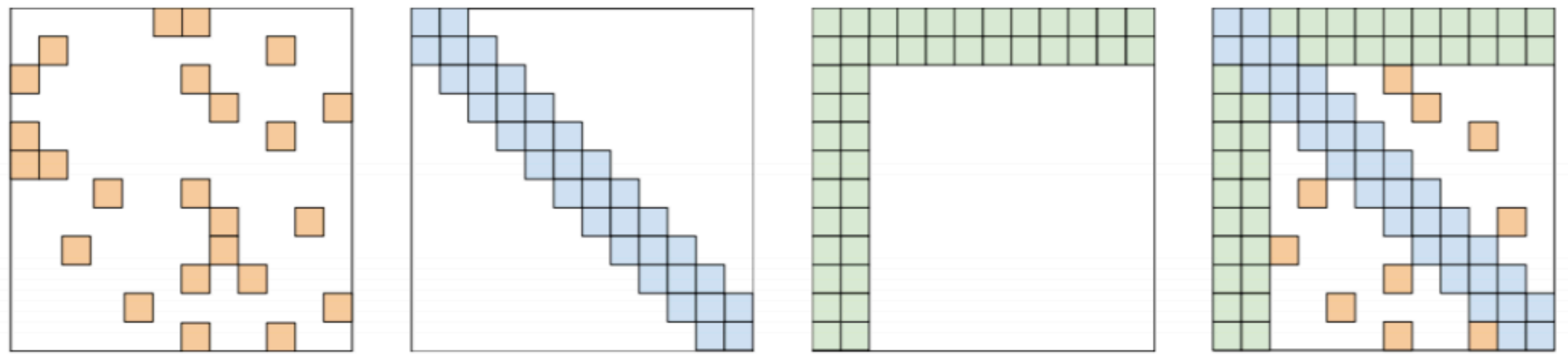
(b) is correct.

Transformers: pros and cons

- Easier to capture dependencies: we draw attention between every pair of words!
- Easier to parallelize: $Q = XW^Q \quad K = XW^K \quad V = XW^V$

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

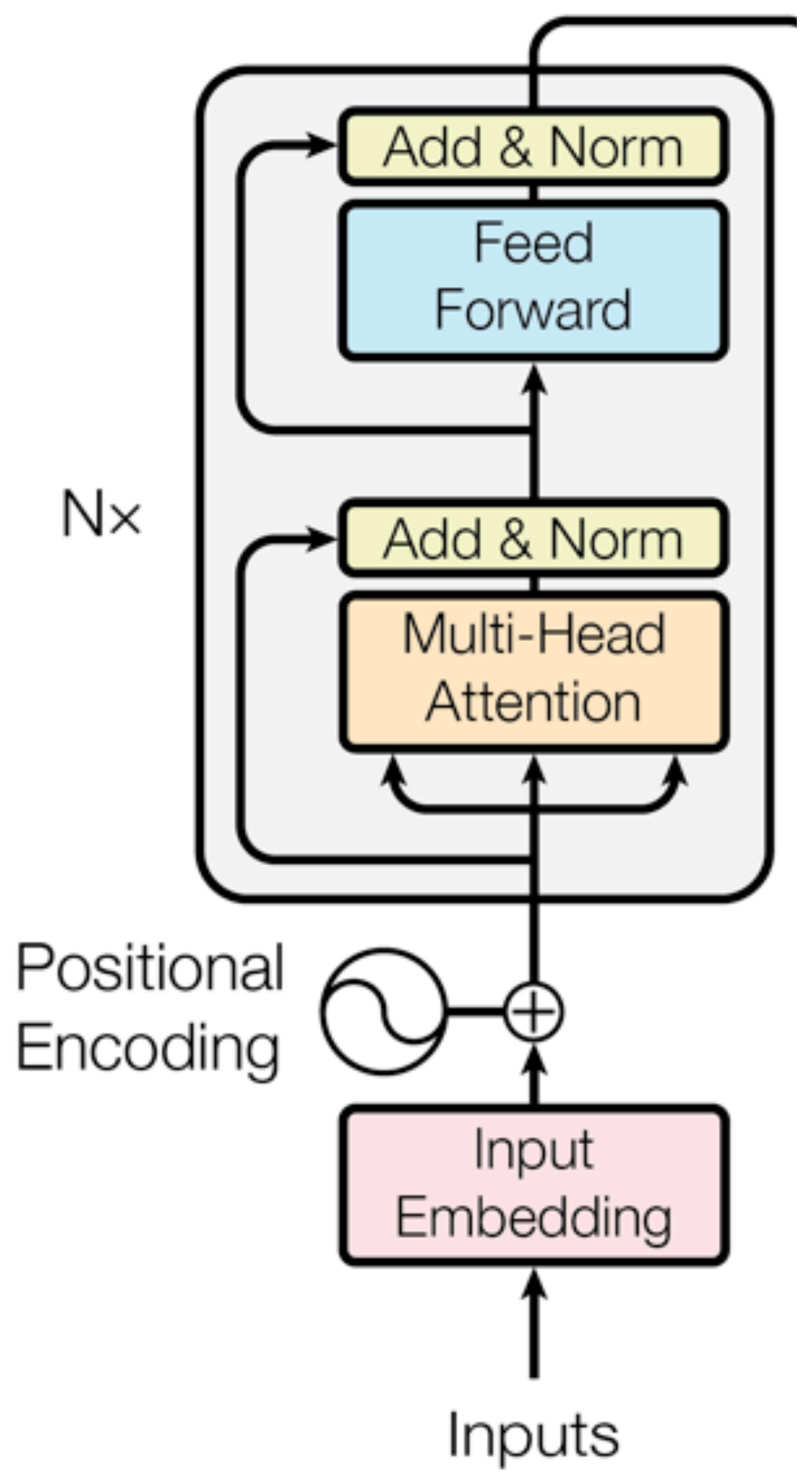
- Quadratic computation in self-attention:
  - Can become very slow when the sequence length is large



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

- Are these positional representations enough to capture positional information?

Transformer encoder



Each encoder layer has two sub-layers:

- A multi-head self-attention layer
- A feedforward layer

Add & Norm:

$$\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$$

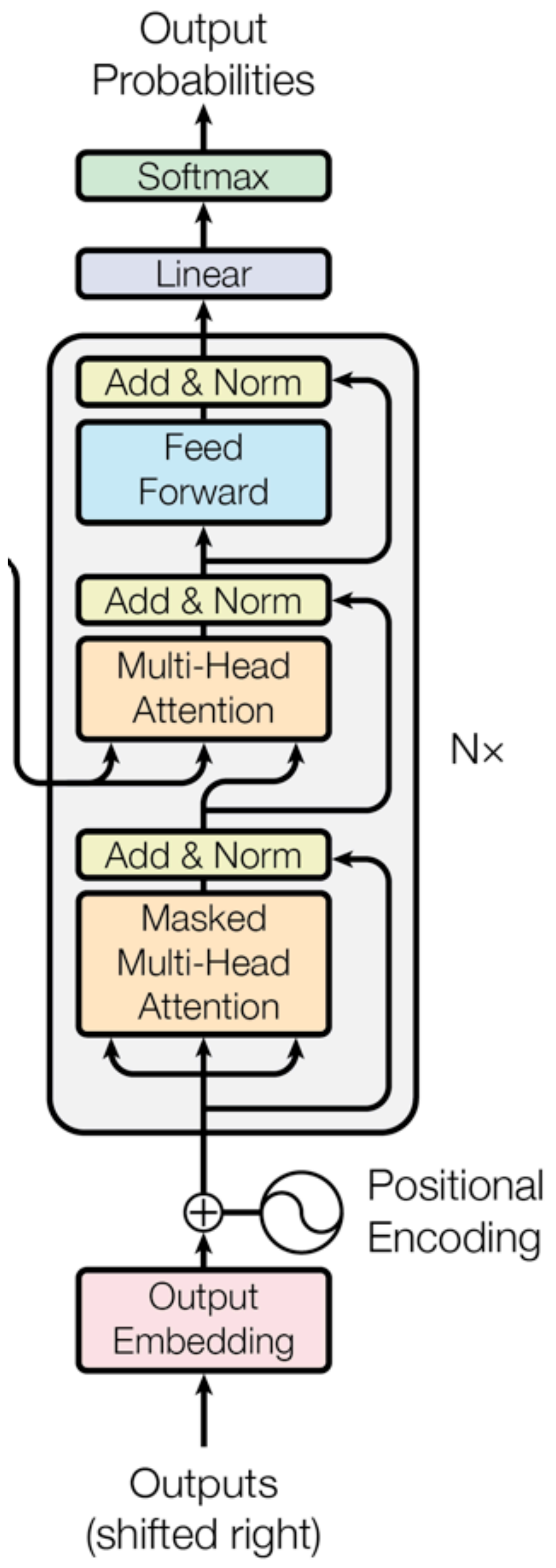
- Residual connection (He et al., 2016)
- Layer normalization (Ba et al., 2016)

[advanced]

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

In (Vaswani et al., 2017), N = 6

Transformer decoder



Each decoder layer has three sub-layers:

- A **masked multi-head attention** layer
- A multi-head **cross-attention** layer
- A feedforward layer

Masked multi-head attention:

self-attention on the decoder states

However, you can't see the future!

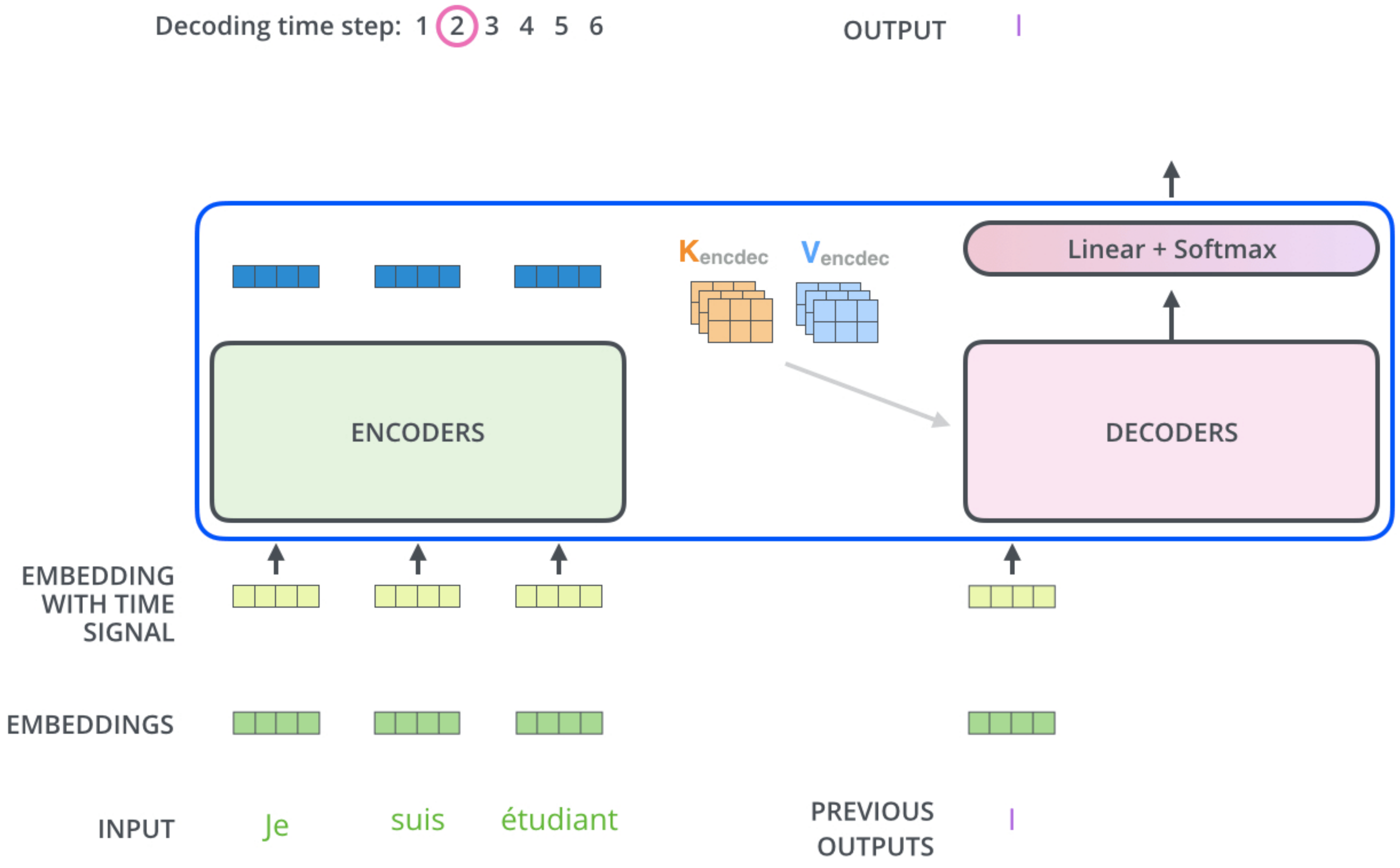
Multi-head cross-attention:

Decoder attends to encoder states

encoder: keys/values, decoder: queries

In (Vaswani et al., 2017), N = 6

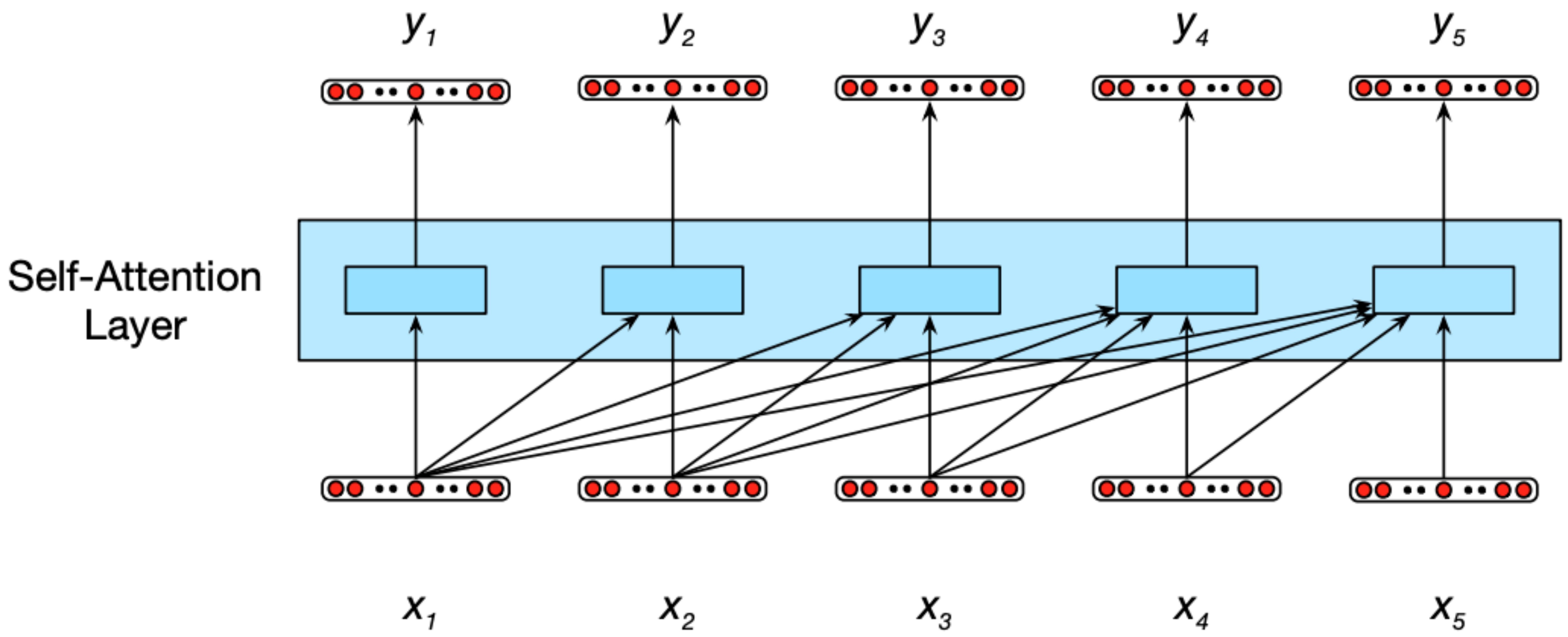
Transformer decoder



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

Masked multi-head attention

- Key point: you can't see the future words for the decoder!

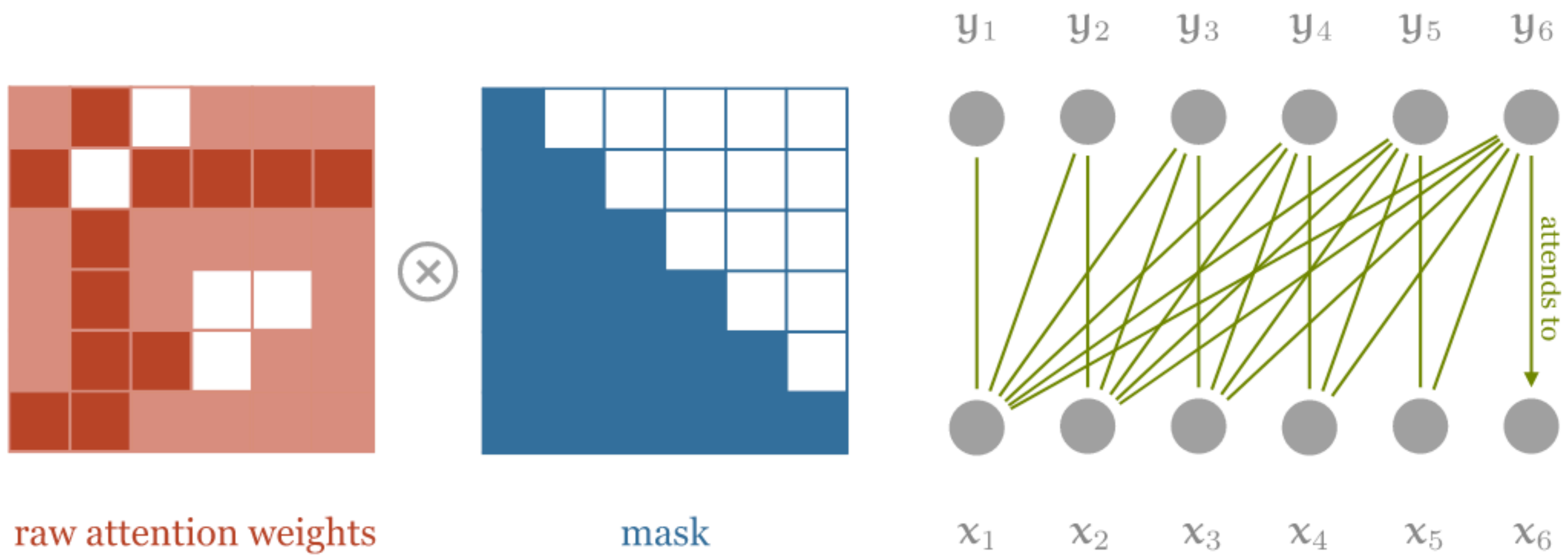


- Solution: for every $\mathbf{q}_i$, only attend to $\{(\mathbf{k}_j, \mathbf{v}_j)\}, j \leq i$

Masked multi-head attention

$$\mathbf{q}_i = W^Q \mathbf{x}_i, \mathbf{k}_i = W^K \mathbf{x}_i, \mathbf{v}_i = W^V \mathbf{x}_i$$

$$\alpha_{i,j} = \text{softmax}(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}})$$



Efficient implementation: compute attention as we normally do, mask out attention to future words by setting attention scores to $-\infty$

```
dot = torch.bmm(queries, keys.transpose(1, 2))

indices = torch.triu_indices(t, t, offset=1)
dot[:, indices[0], indices[1]] = float('-inf')

dot = F.softmax(dot, dim=2)
```

http://peterbloem.nl/blog/transformers

Multi-head cross-attention

$$\mathbf{q}_i = W^Q \mathbf{x}_i, \mathbf{k}_i = W^K \mathbf{x}_i, \mathbf{v}_i = W^V \mathbf{x}_i$$

$$\alpha_{i,j} = \text{softmax}(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}})$$

↓

$$\mathbf{q}_i = W^Q \mathbf{x}_i \quad \mathbf{k}_j = W^K \mathbf{h}_j, \mathbf{v}_j = W^V \mathbf{h}_j$$

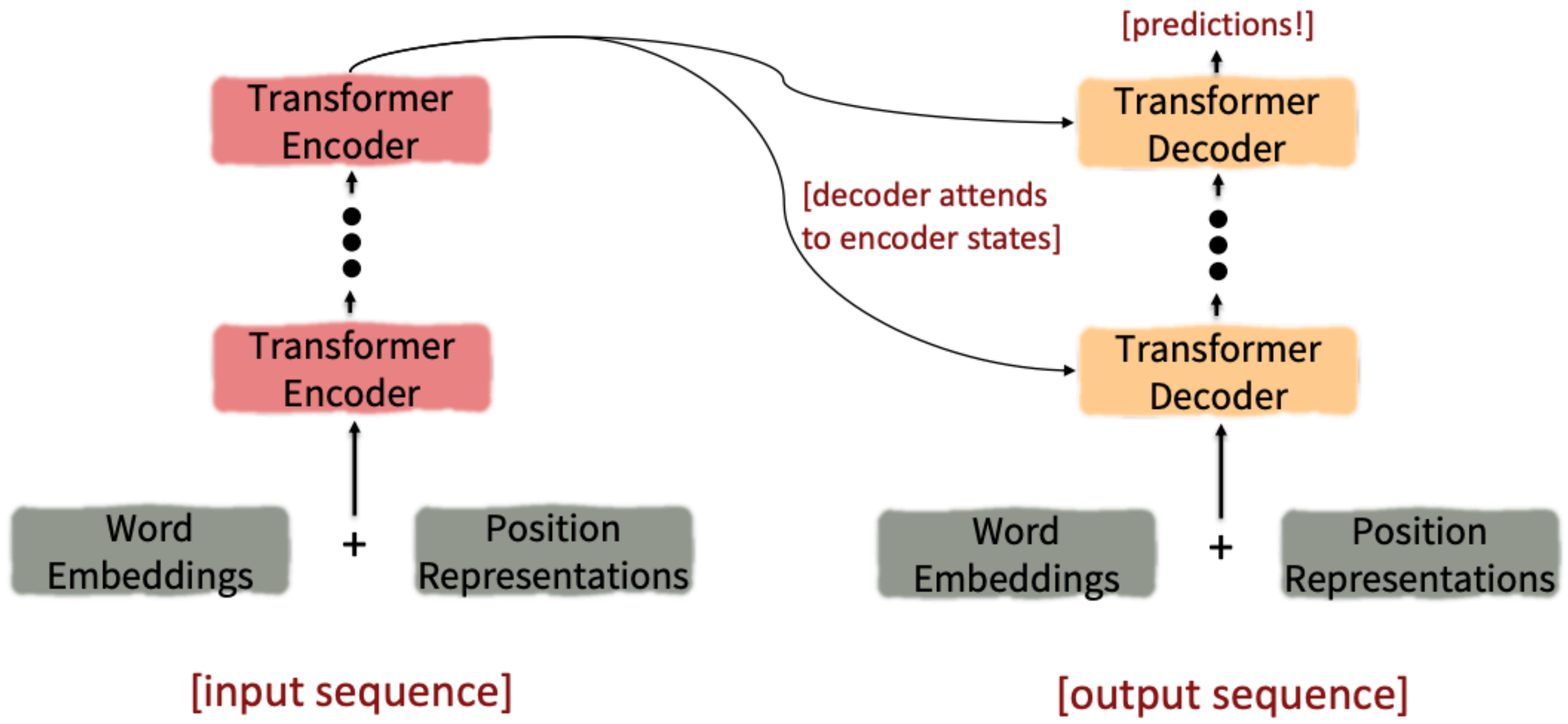
$$\alpha_{i,j} = \text{softmax}(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}})$$

Q: What is the size of α?

$$\mathbf{y}_i = \sum_{j=1}^{m} \alpha_{i,j} \mathbf{v}_j$$

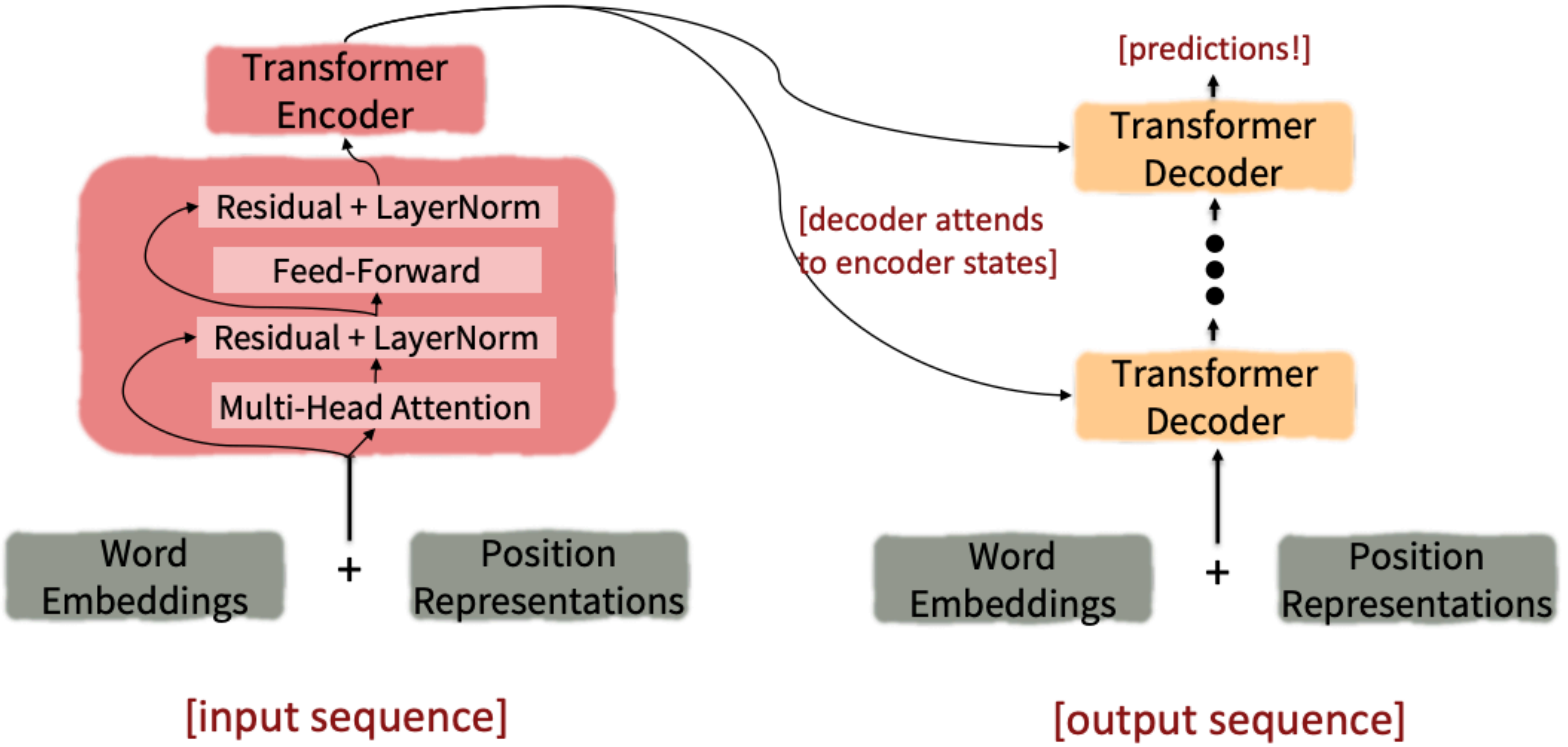
- $\mathbf{h}_1, \ldots, \mathbf{h}_m$: hidden states from encoder
- $\mathbf{x}_1, \ldots, \mathbf{x}_n$: hidden states from decoder

Putting the pieces together



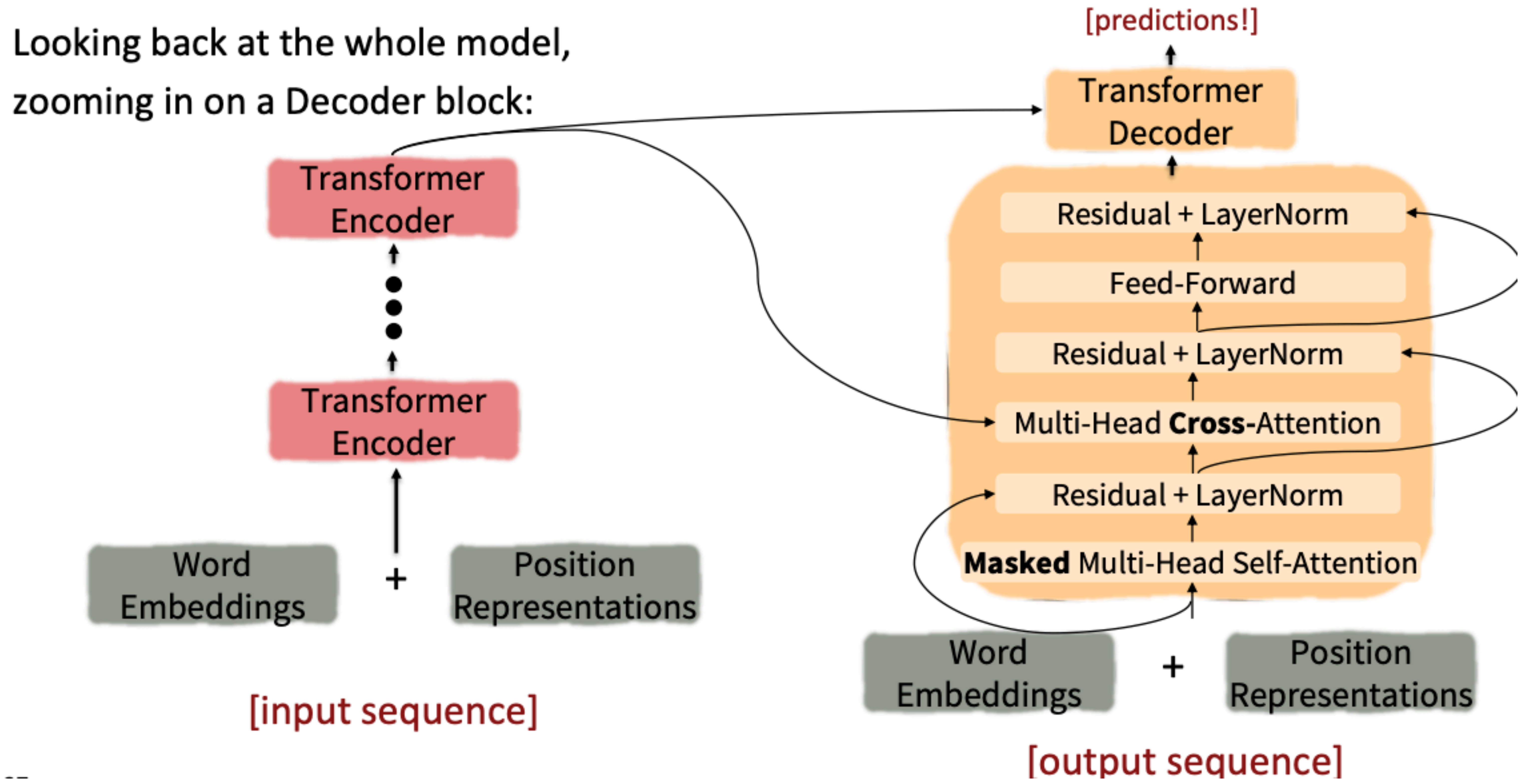
Putting the pieces together

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



Putting the pieces together

Looking back at the whole model,
zooming in on a Decoder block:



Transformers: machine translation

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

Vaswani et al., 2017: Attention Is All You Need

Transformers: document generation

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

Very large gains compared to seq2seq-attention with LSTMs!

Liu et al., 2018: Generating Wikipedia by Summarizing Long Sequences