COS 484

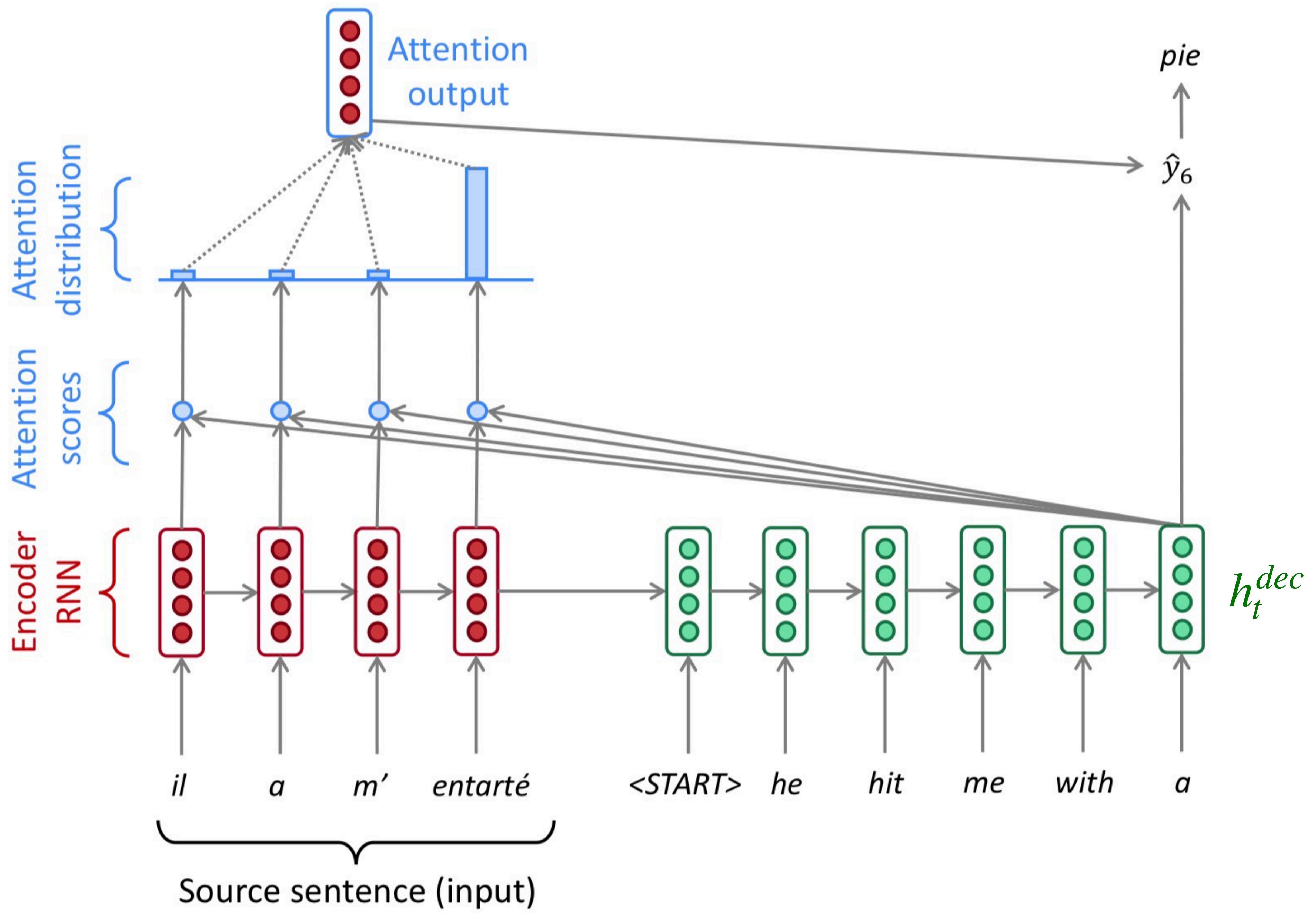
Natural Language Processing

L13: Self-attention and Transformers

Spring 2025

(Some slides adapted from John Hewitt)

Recap: Attention

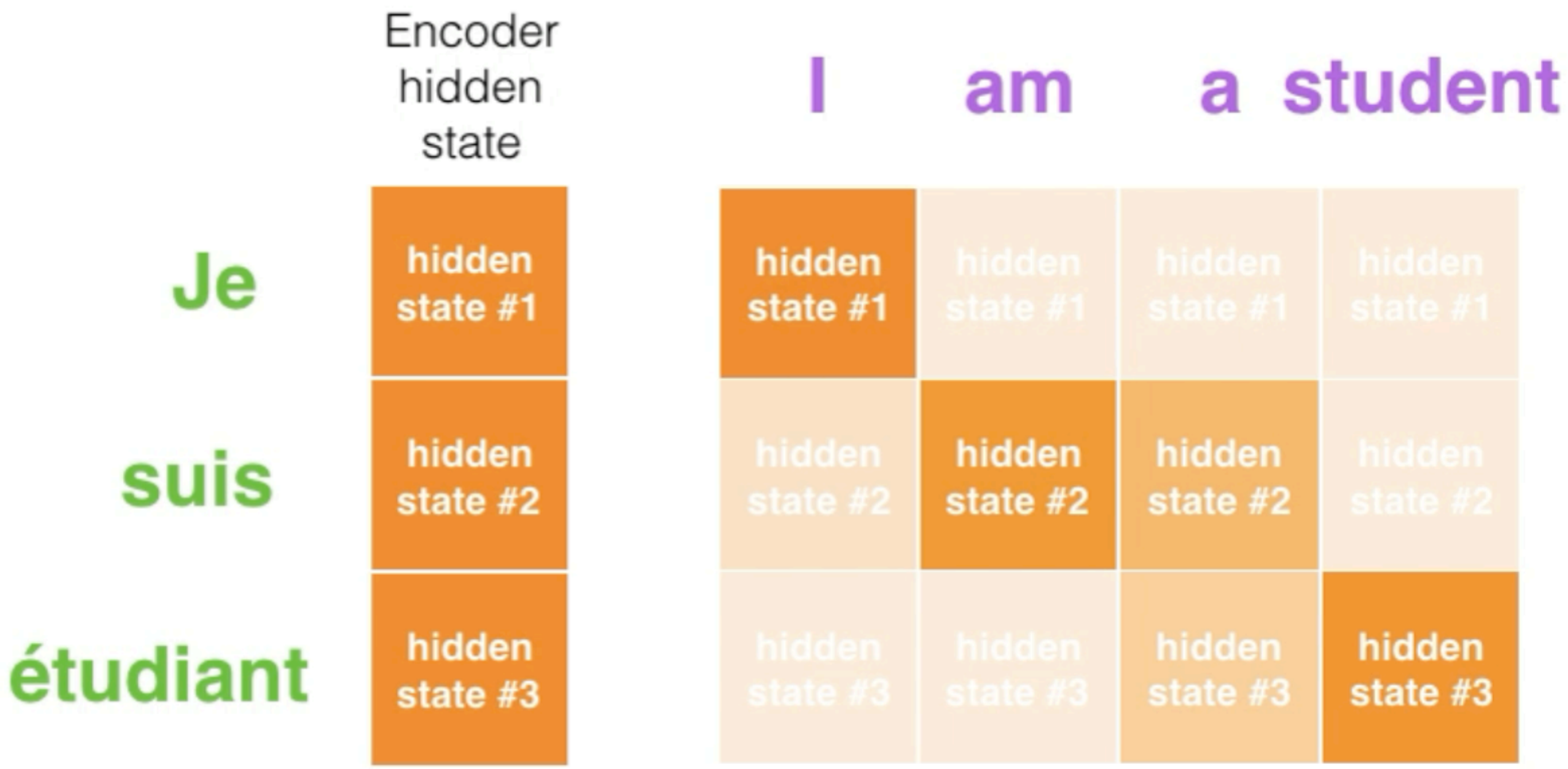


- Encoder hidden states: $h_1^{enc}, \ldots, h_n^{enc}$
  (n: # of words in source sentence)
- Decoder hidden state at time t: h_t^{dec}
- Attention scores:
  $$e^t = [g(h_1^{enc}, h_t^{dec}), \ldots, g(h_n^{enc}, h_t^{dec})] \in \mathbb{R}^n$$
- Attention distribution:
  $$\alpha^t = \text{softmax}(e^t) \in \mathbb{R}^n$$
- Weighted sum of encoder hidden states:
  $$a_t = \sum_{i=1}^{n} \alpha_i^t h_i^{enc} \in \mathbb{R}^h$$

Combine a_t and h_t^{dec} to predict next word

Note that $h_1^{enc}, \ldots, h_n^{enc}$ and h_t^{dec} are hidden states from encoder and decoder RNNs..

Recap: Attention



- Attention addresses the “bottleneck” or fixed representation problem
- Attention learns the notion of **alignment**

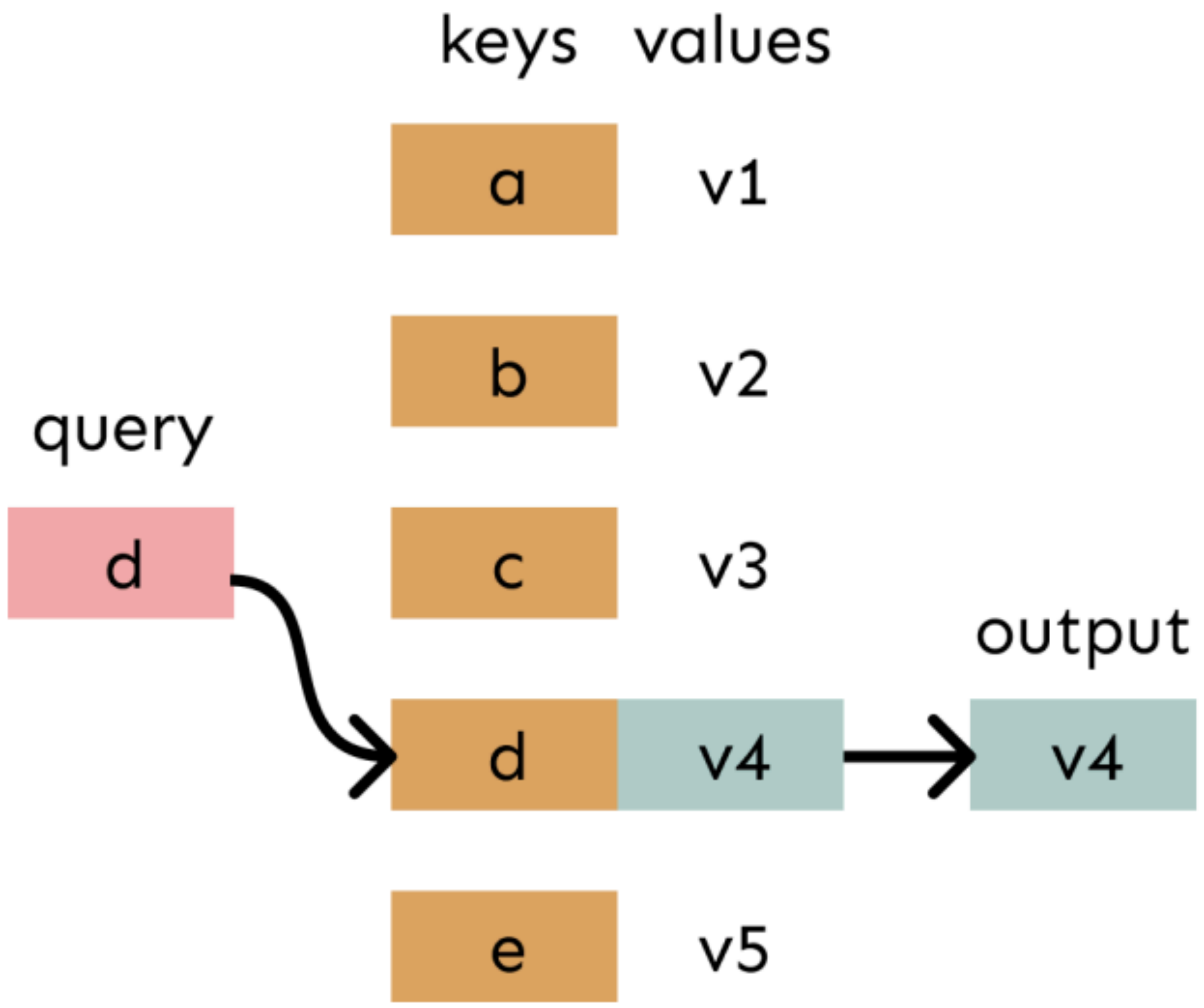
  “Which source words are more relevant to the current target word?”

https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

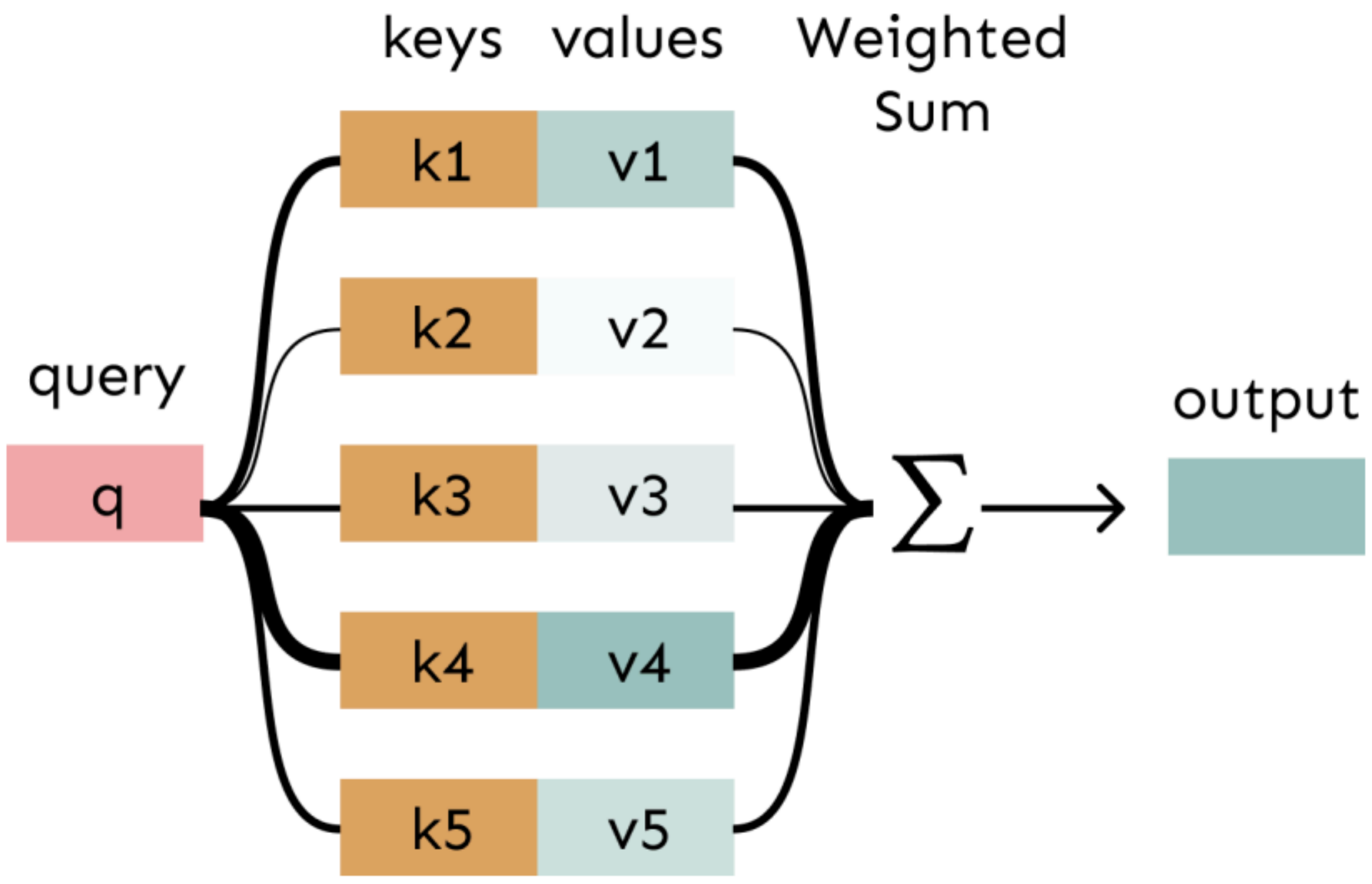
Attention as a soft, averaging lookup table

We can think of **attention** as performing fuzzy lookup a in **key-value store**

Lookup table: a table of keys that map to values. The query matches one of the keys, returning its value.



Attention: The query matches to all keys softly to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



(So far, we assume key = value)

Transformers

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

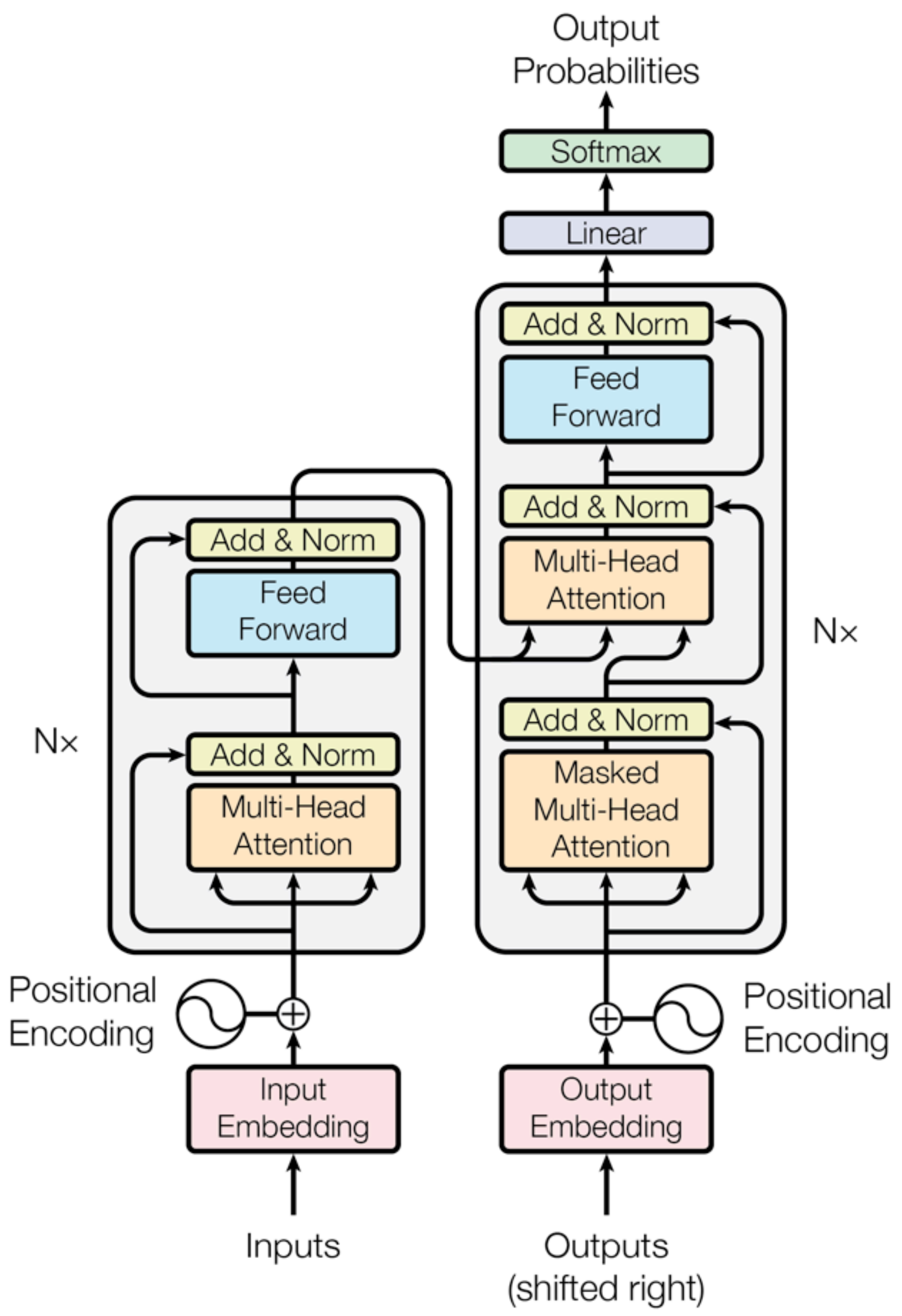
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

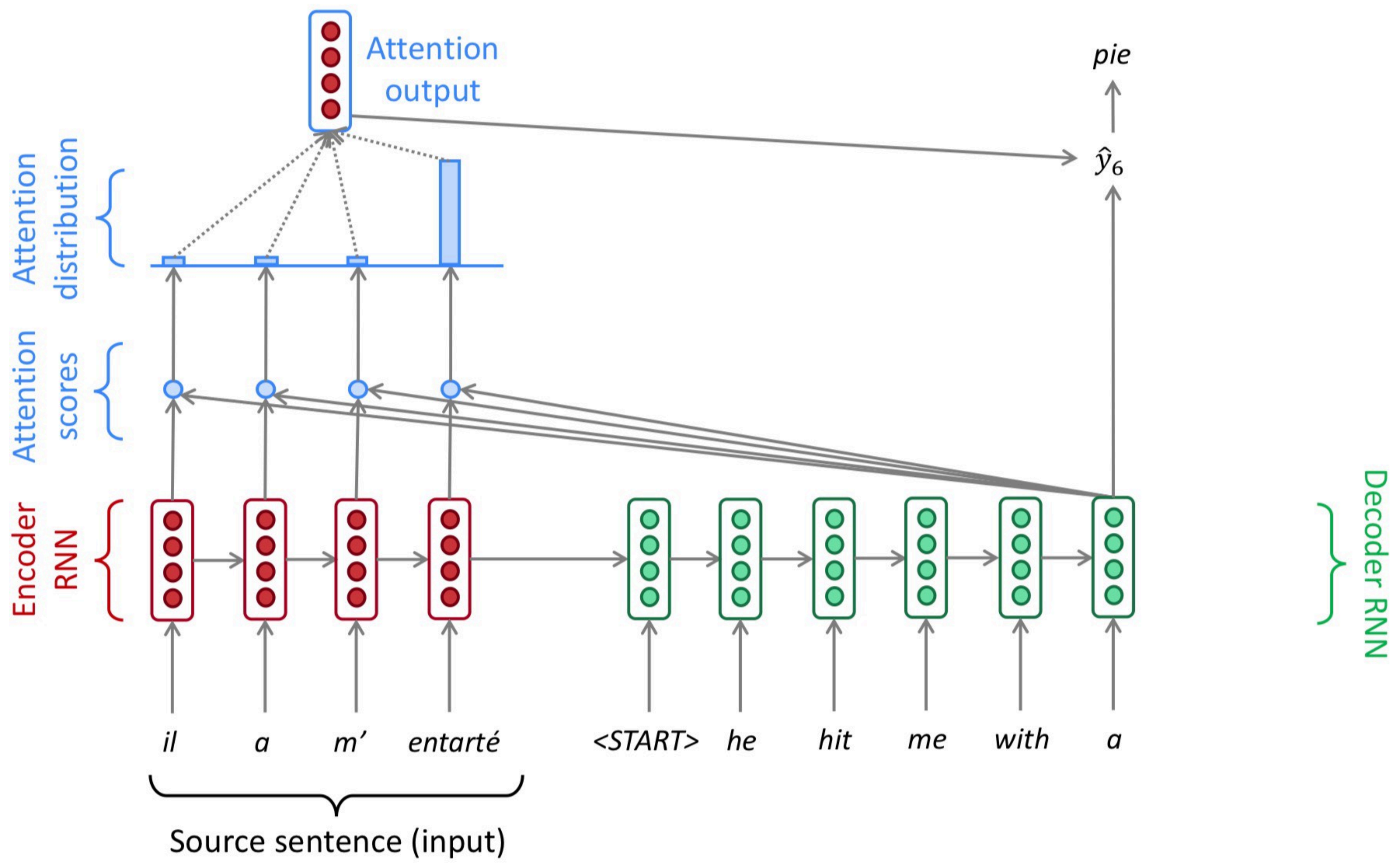
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

(Vaswani et al., 2017)

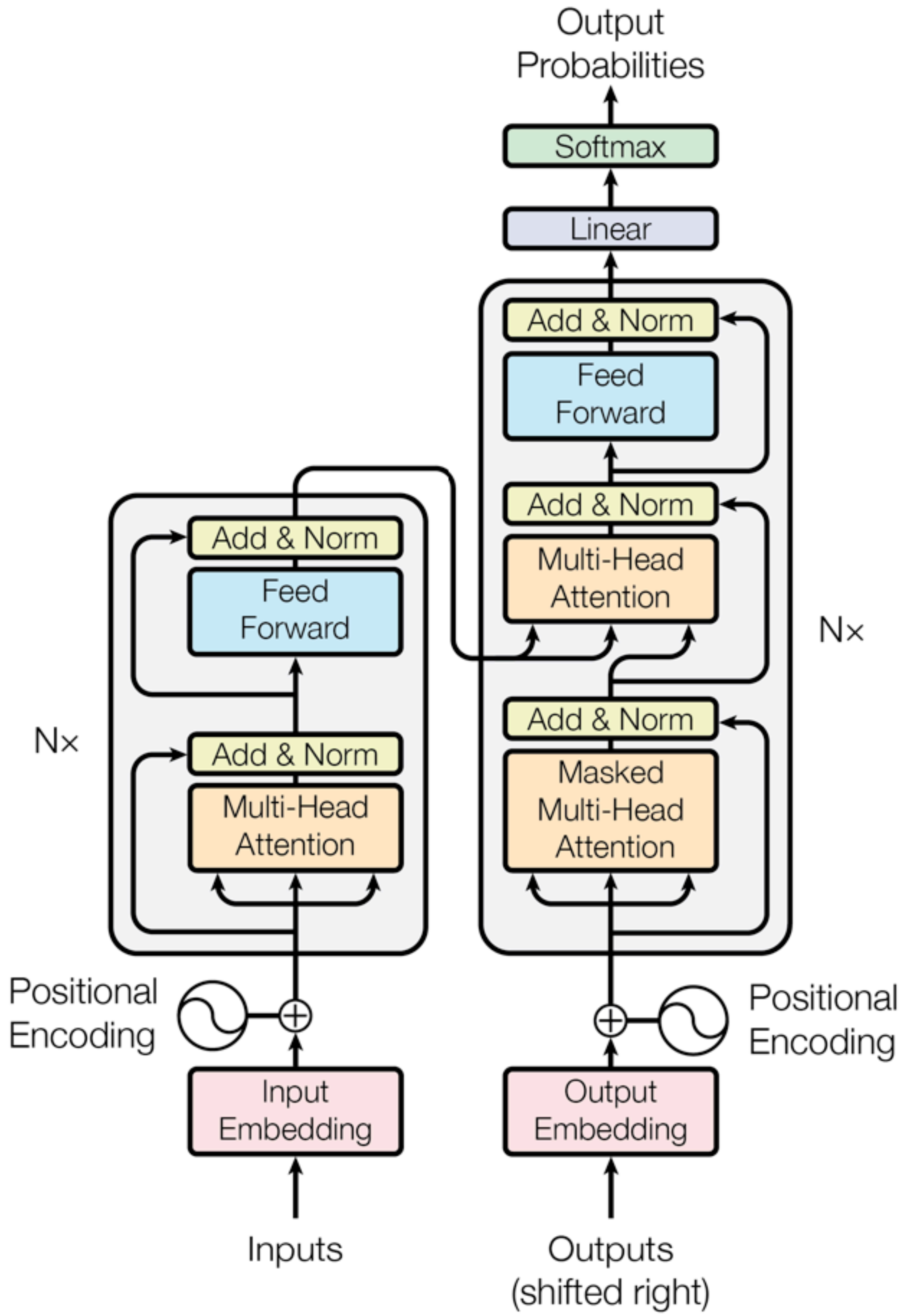
Transformer encoder-decoder



- Transformer encoder + Transformer decoder
- First designed and experimented on NMT
- Can be viewed as a replacement for seq2seq + attention based on RNNs



Transformer encoder-decoder



- Transformer encoder = a stack of **encoder layers**
- Transformer decoder = a stack of **decoder layers**

Transformer encoder: BERT, RoBERTa, ELECTRA

Transformer decoder: GPT-3, ChatGPT, Palm

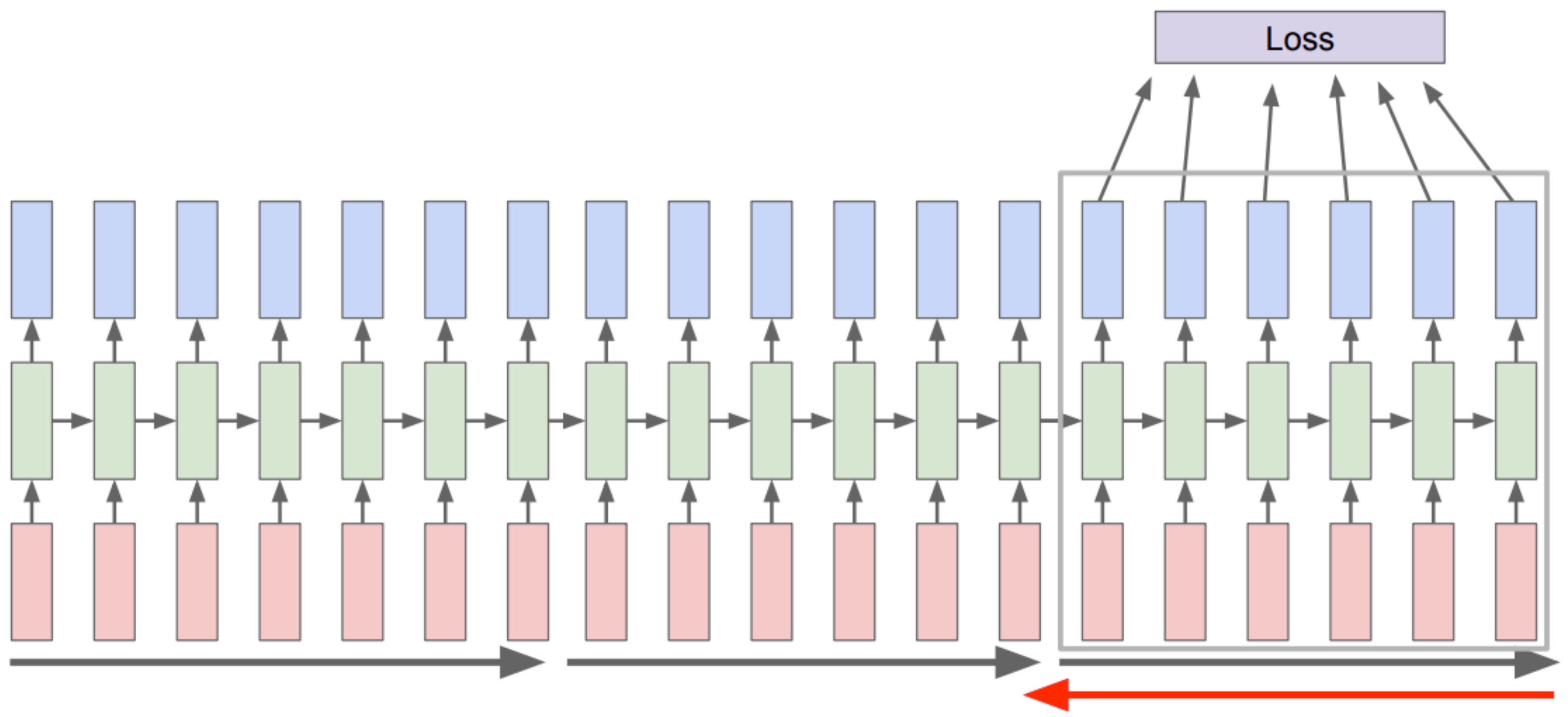
Transformer encoder-decoder: T5, BART

- Key innovation: **multi-head, self-attention**
- Clean & effective architecture design
- Transformers don't have any recurrence structures!

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

Issues with recurrent NNs

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

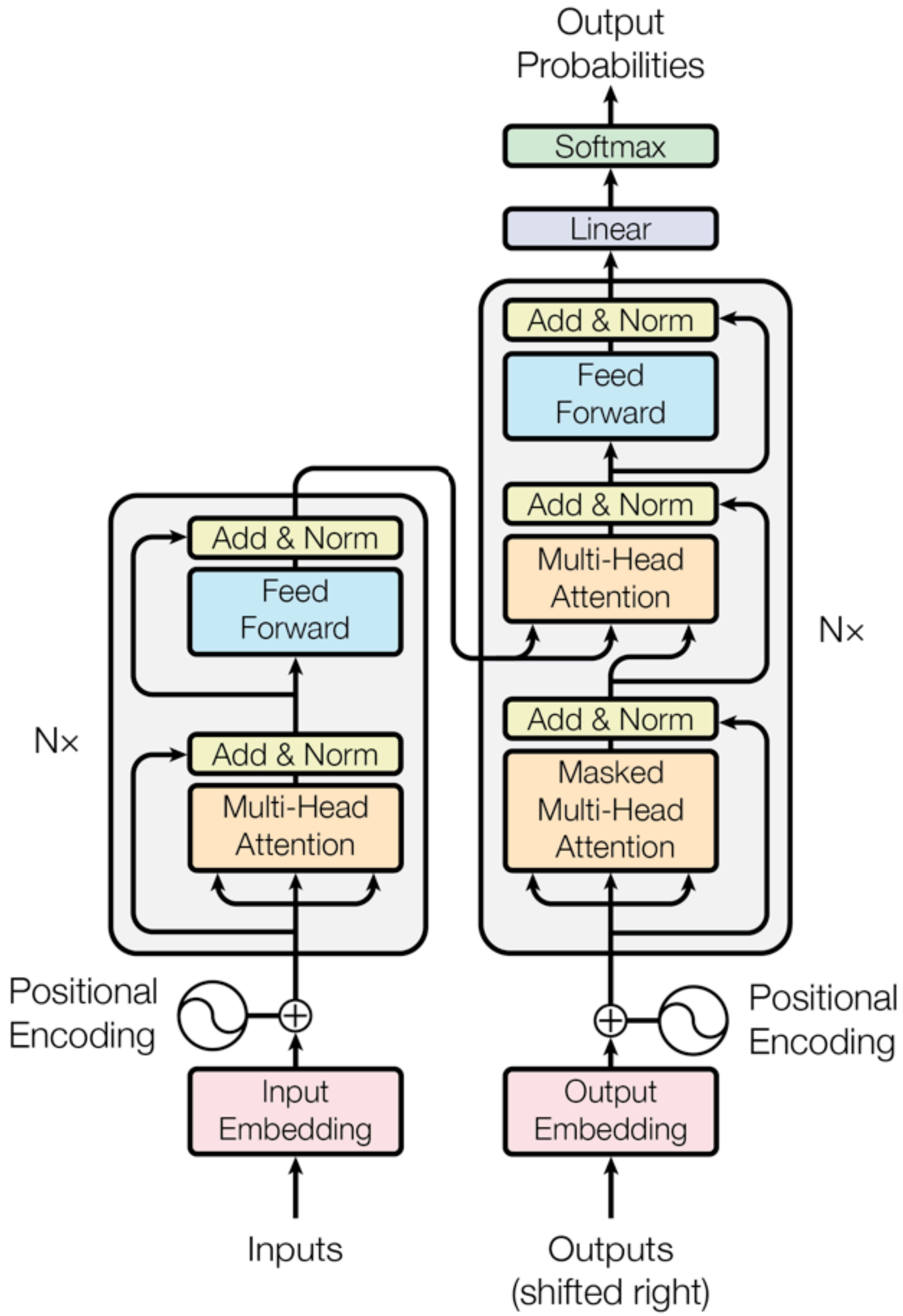


- RNNs **lack parallelizability**
  - Forward and backward passes have O(sequence length) unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - Inhibits training on very large datasets

RNNs / LSTMs → seq2seq → seq2seq + attention → attention only = Transformers!

Transformers have become a new building block to replace RNNs

Transformers: roadmap



- From attention to self-attention
- From self-attention to multi-head self-attention
- Feedforward layers
- Positional encoding
- Residual connections + layer normalization
- Transformer encoder vs Transformer decoder

Attention in a general form

- 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** $\mathbf{e} = g(\mathbf{q}, \mathbf{v}_i) \in \mathbb{R}^n$
  - 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:

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

Attention in a general form

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

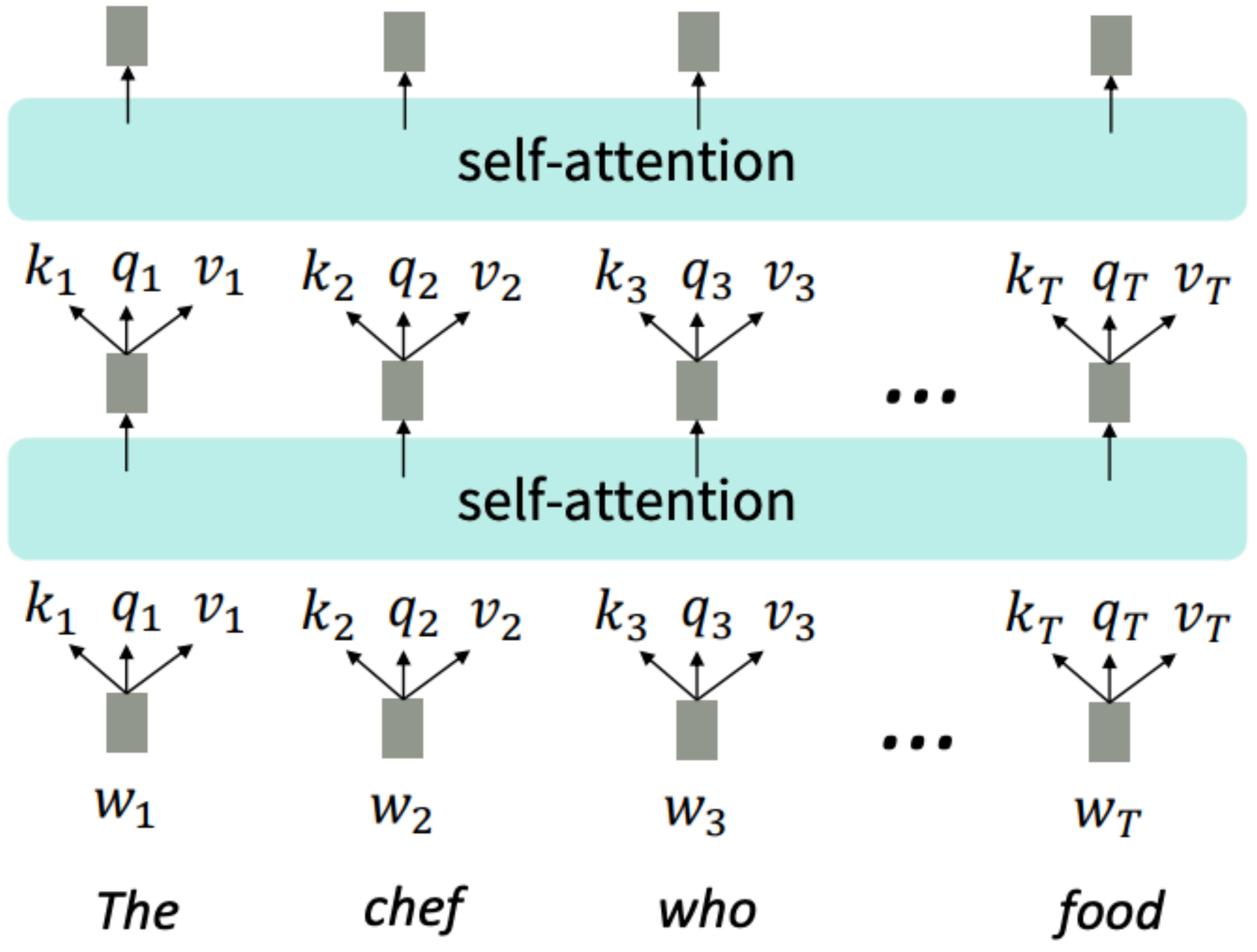
$$\alpha = \mathrm{softmax}(\mathbf{e}) \in \mathbb{R}^n$$

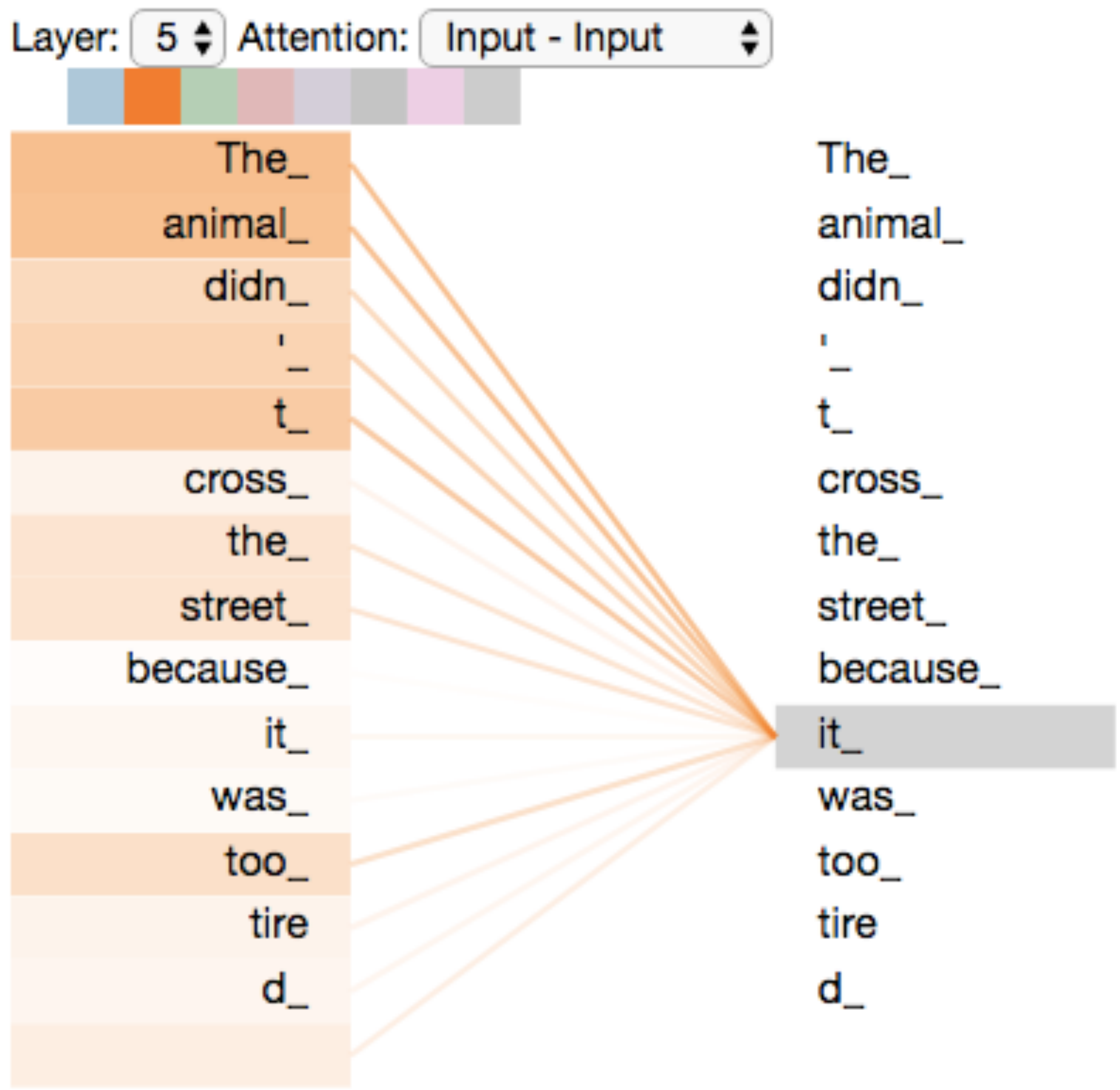
  - Using attention distribution to take **weighted sum** of values:

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

Self-attention

- In NMT, query = decoder hidden state, keys = values = encoder hidden states
- 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.





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

Self-attention

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: $\mathbf{h}_1, \ldots, \mathbf{h}_n \in \mathbb{R}^{d_2}$

- The same abstraction as RNNs - used as a drop-in replacement for an RNN layer

$$\mathbf{h}_t = g(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}) \in \mathbb{R}^h$$

Self-attention:

$$\mathbf{q}_i = \mathbf{W}^{(q)}\mathbf{x}_i, \quad \mathbf{k}_i = \mathbf{W}^{(k)}\mathbf{x}_i, \quad \mathbf{v}_i = \mathbf{W}^{(v)}\mathbf{x}_i,$$

$$\mathbf{h}_i = \mathbf{W}^{(o)} \sum_{j=1}^{n} \left(\frac{\exp(\mathbf{q}_i \cdot \mathbf{k}_j / \sqrt{d})}{\sum_{j'=1}^{n} \exp(\mathbf{q}_i \cdot \mathbf{k}_{j'} / \sqrt{d})} \mathbf{v}_j \right)$$

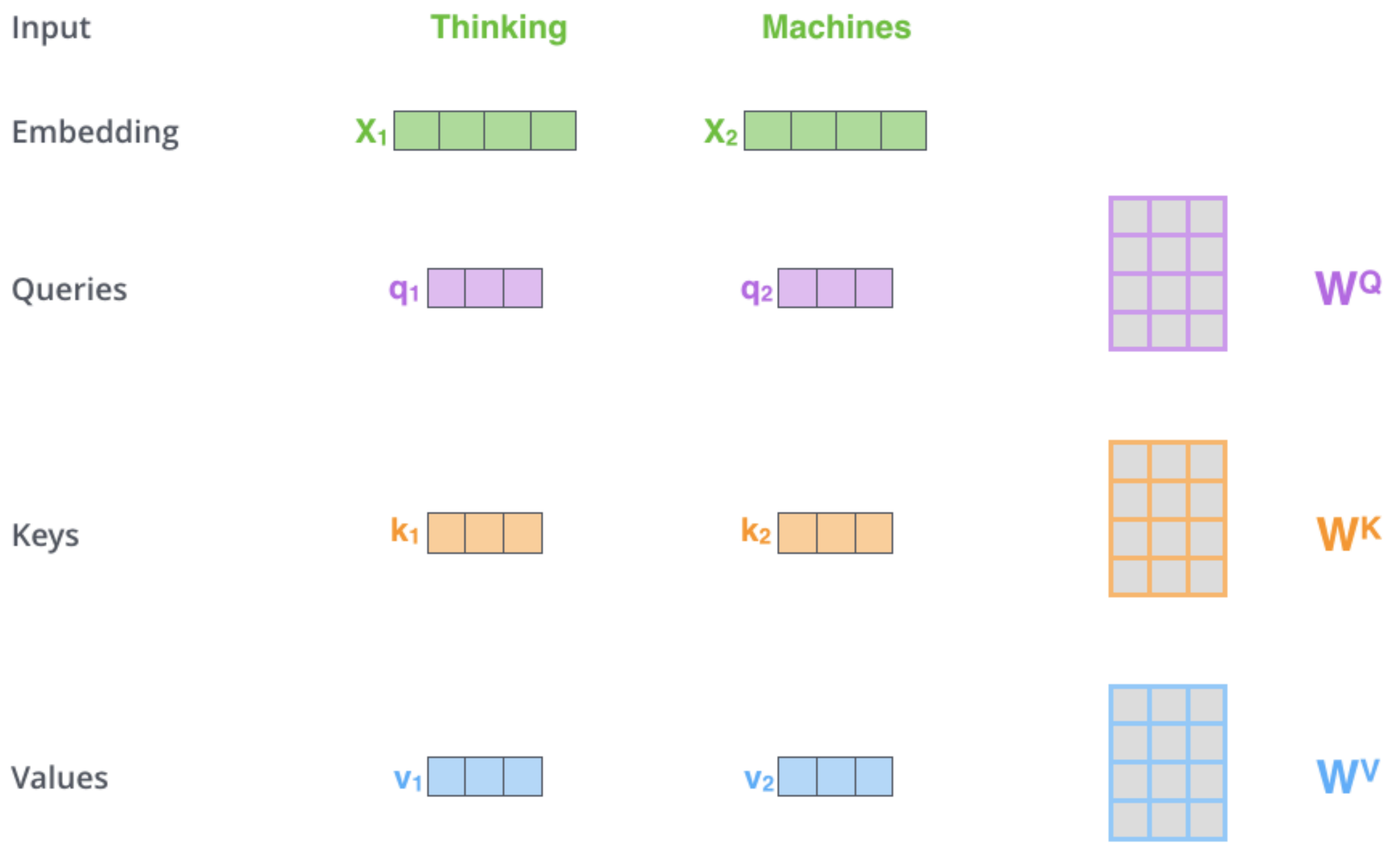
where $\mathbf{W}^{(q)}, \mathbf{W}^{(k)}, \mathbf{W}^{(v)}, \mathbf{W}^{(o)} \in \mathbb{R}^{d \times d}$.

Self-attention

Step #1: Transform each input vector into three vectors: query, key, and value vectors

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \in \mathbb{R}^{d_q} \qquad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \in \mathbb{R}^{d_k} \qquad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^{d_v}$$

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



Note that we use row vectors here;
It is also common to write

$$\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i \in \mathbb{R}^{d_q}$$

for $\mathbf{x}_i$ = a column vector

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

Self-attention

Step #2: Compute pairwise similarities between keys and queries; normalize with softmax

For each $\mathbf{q}_i$, compute attention scores and attention distribution:

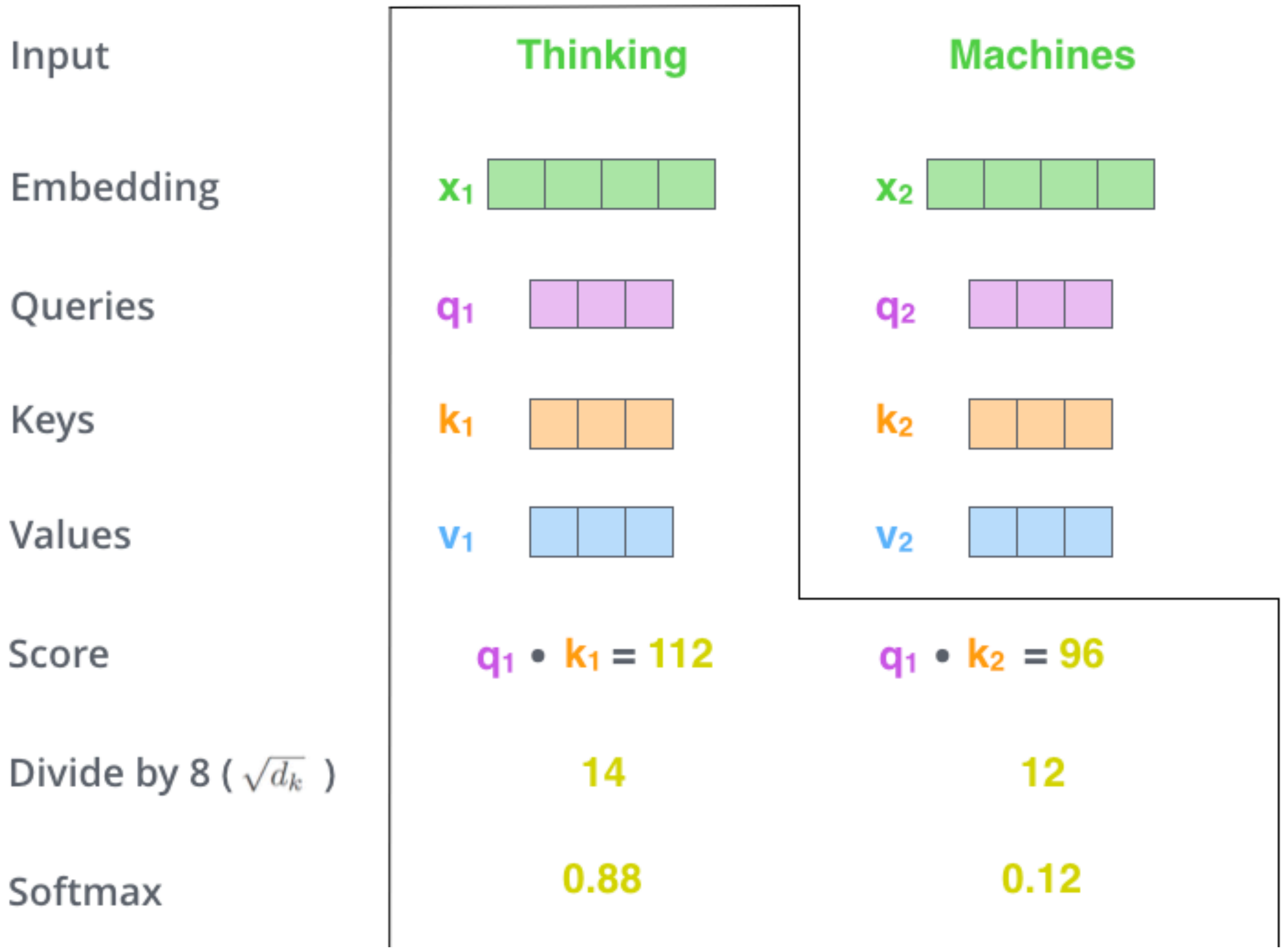
$$\alpha_{i,j} = \mathrm{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right)$$

aka. "scaled dot product"

It must be $d_q = d_k$ in this case

Q. Why scaled dot product?

Intuition: Assuming q_i and k_j are normally distributed, the dot product might be too large for larger d_k; scaling the dot product by $\frac{1}{\sqrt{d_k}}$ is easier for optimization



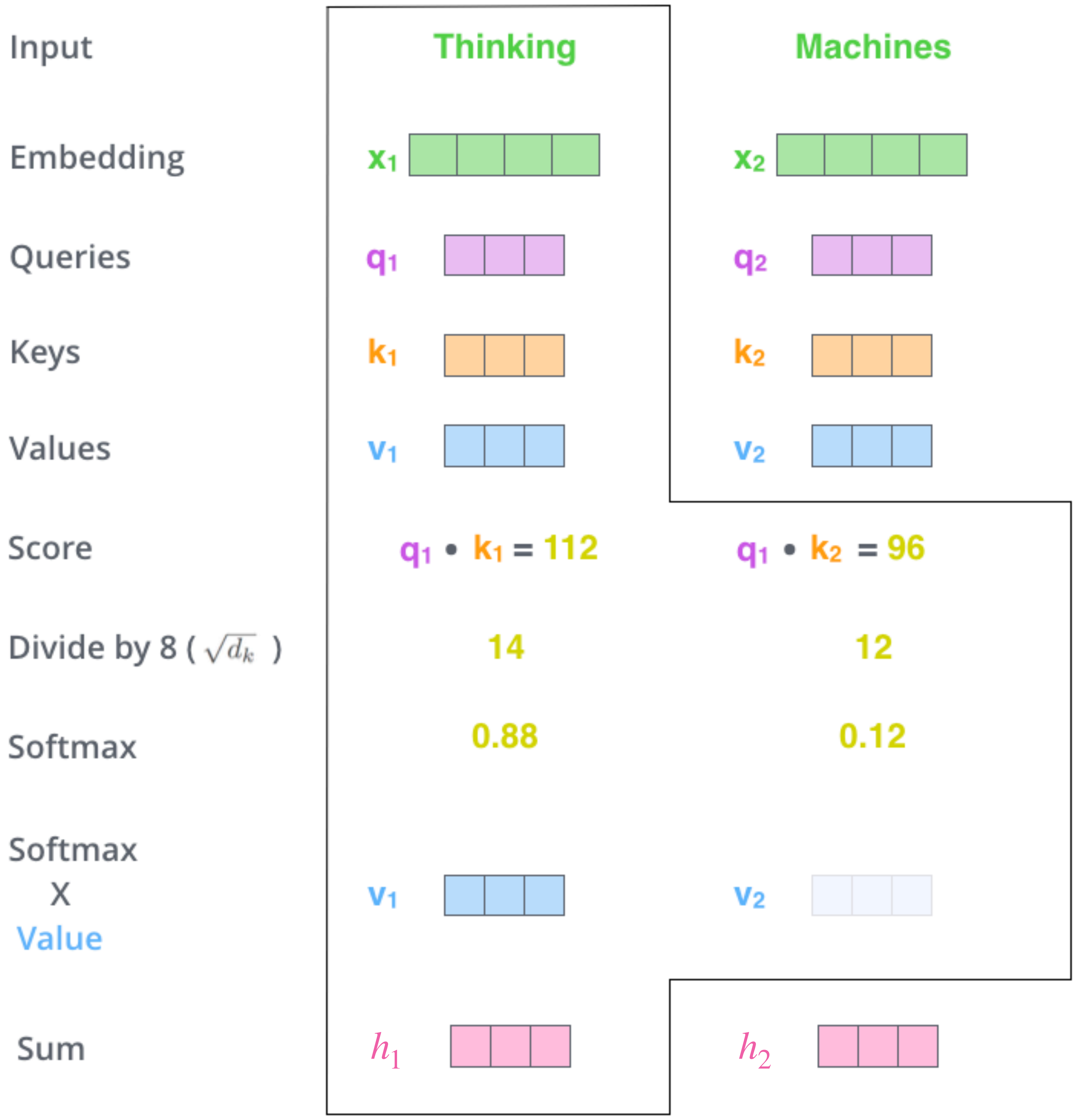
https://jalammar.github.io/illustrated-transformer/

Self-attention

Step #3: Compute output for each input as weighted sum of values

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

$(d_v = d_2)$



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

Self-attention

What would be the output vector for the word "Thinking" approximately?

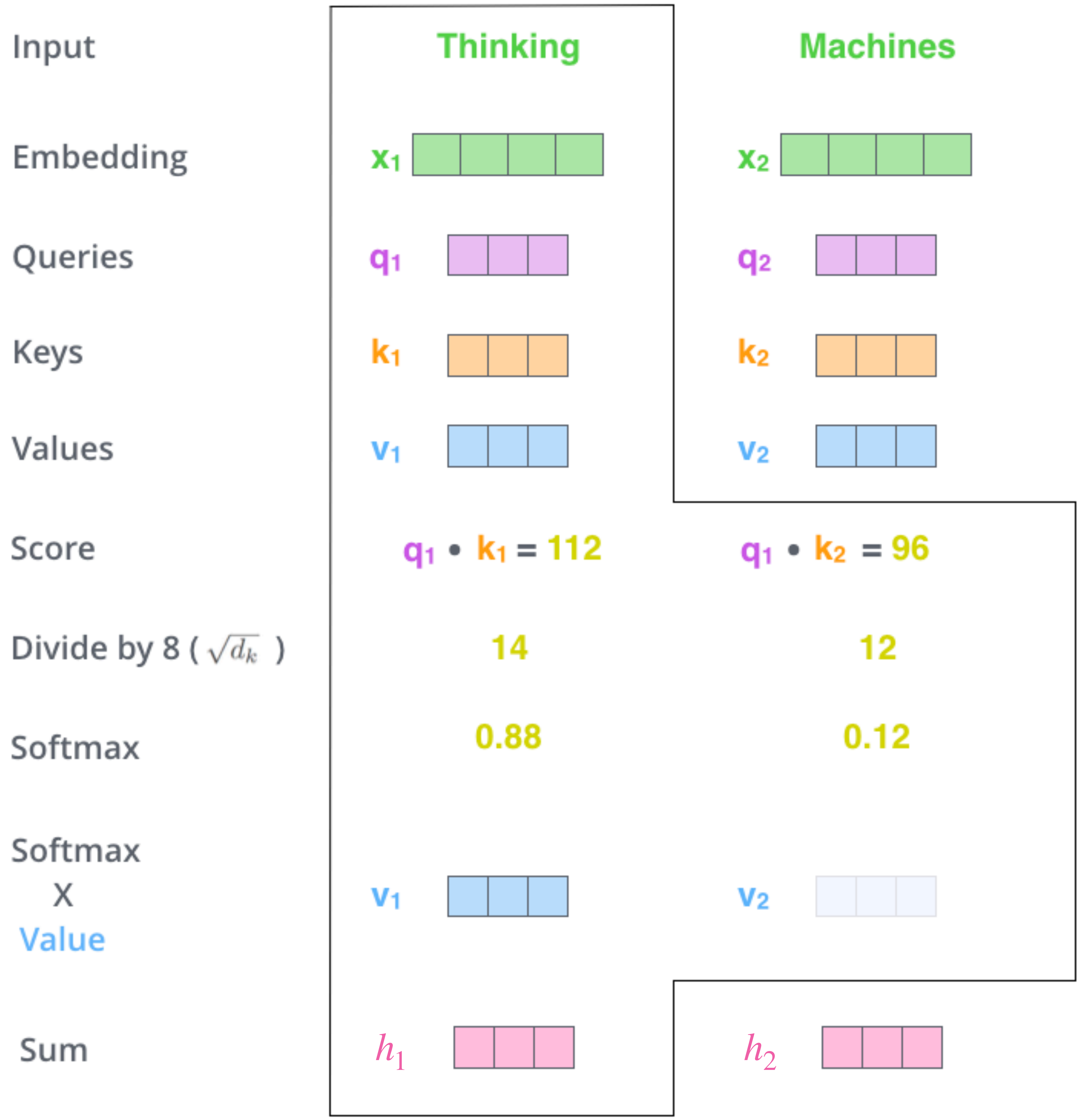
(a) $0.5\mathbf{v}_1 + 0.5\mathbf{v}_2$

(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: matrix notations

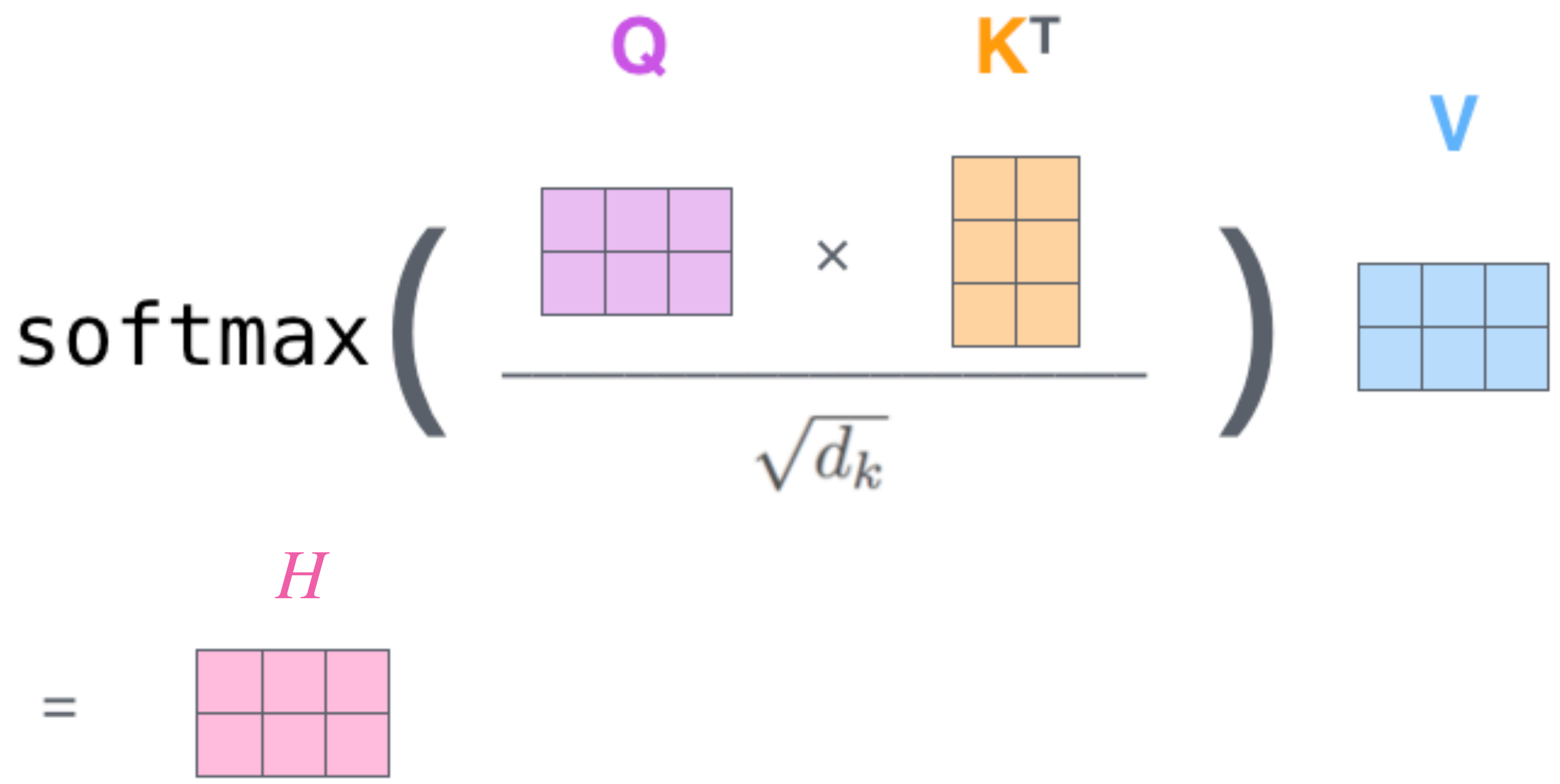
$X \in \mathbb{R}^{n \times d_1}$ (n = input length)

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

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

$n \times d_q$ $\quad d_k \times n$ $\quad n \times d_v$

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

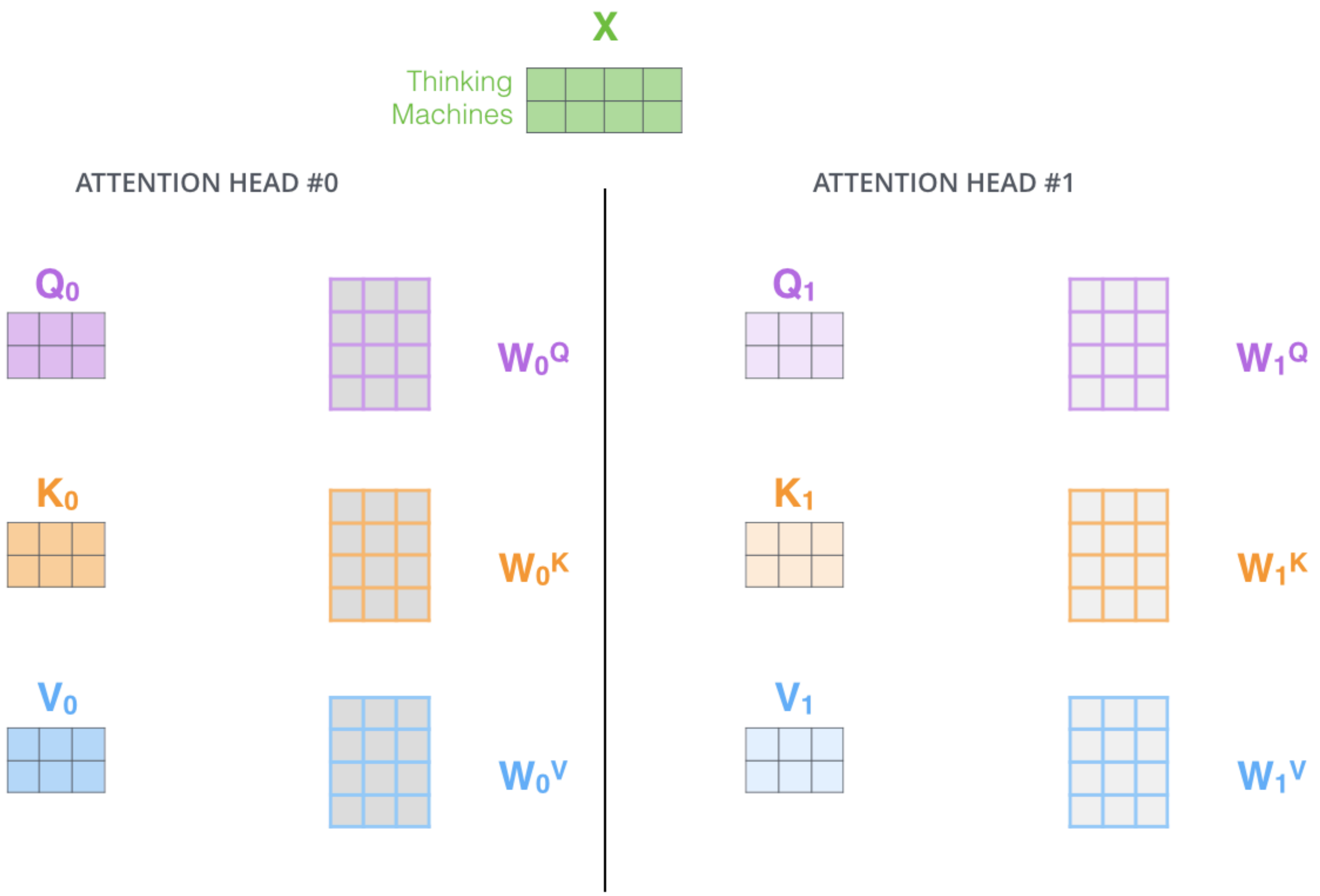


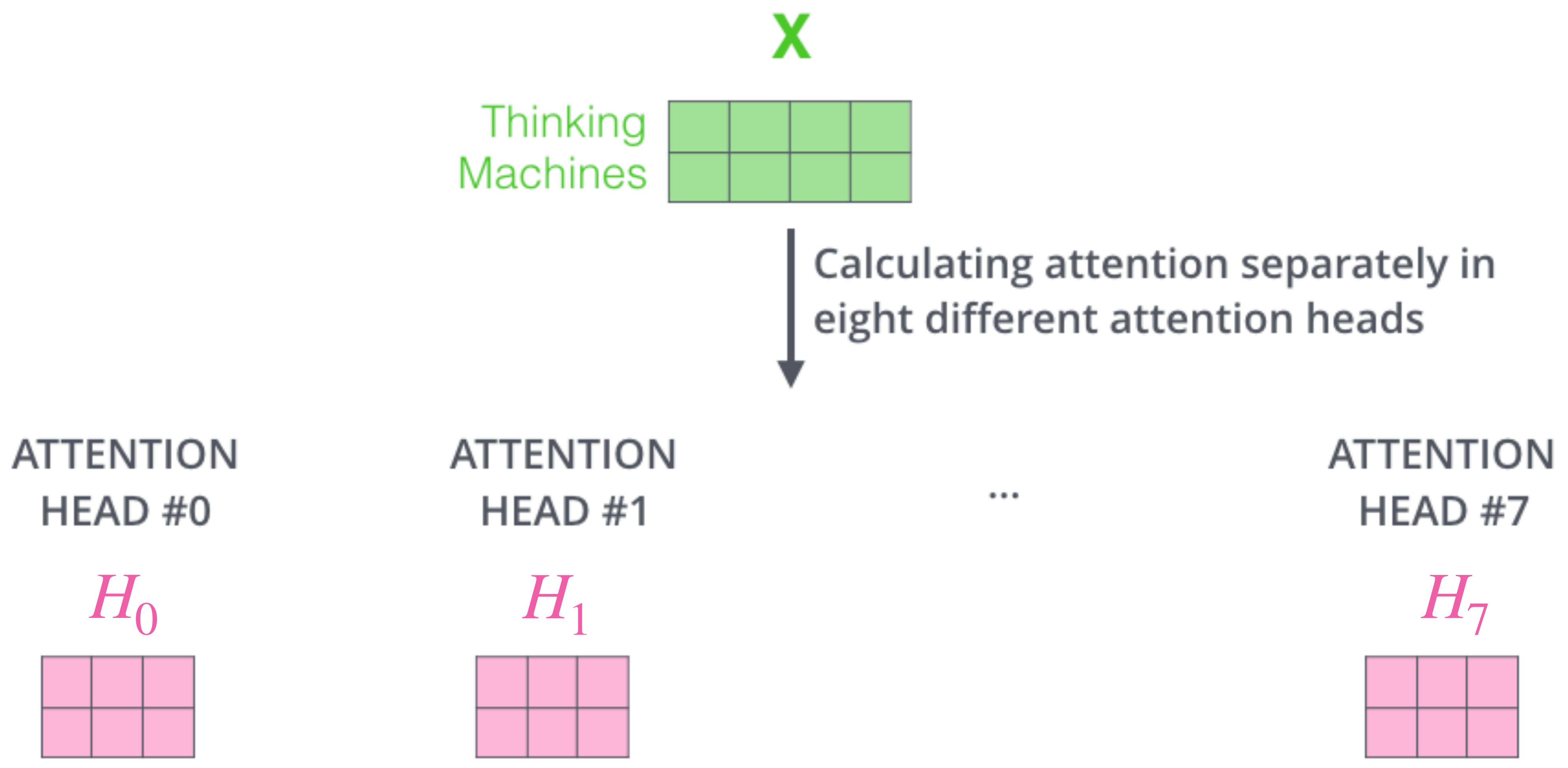
https://jalammar.github.io/illustrated-transformer/

Multi-head attention

"The Beast with Many Heads"

- It is better to use multiple attention functions instead of one!
  - Each attention function ("head") can focus on different key positions / content.





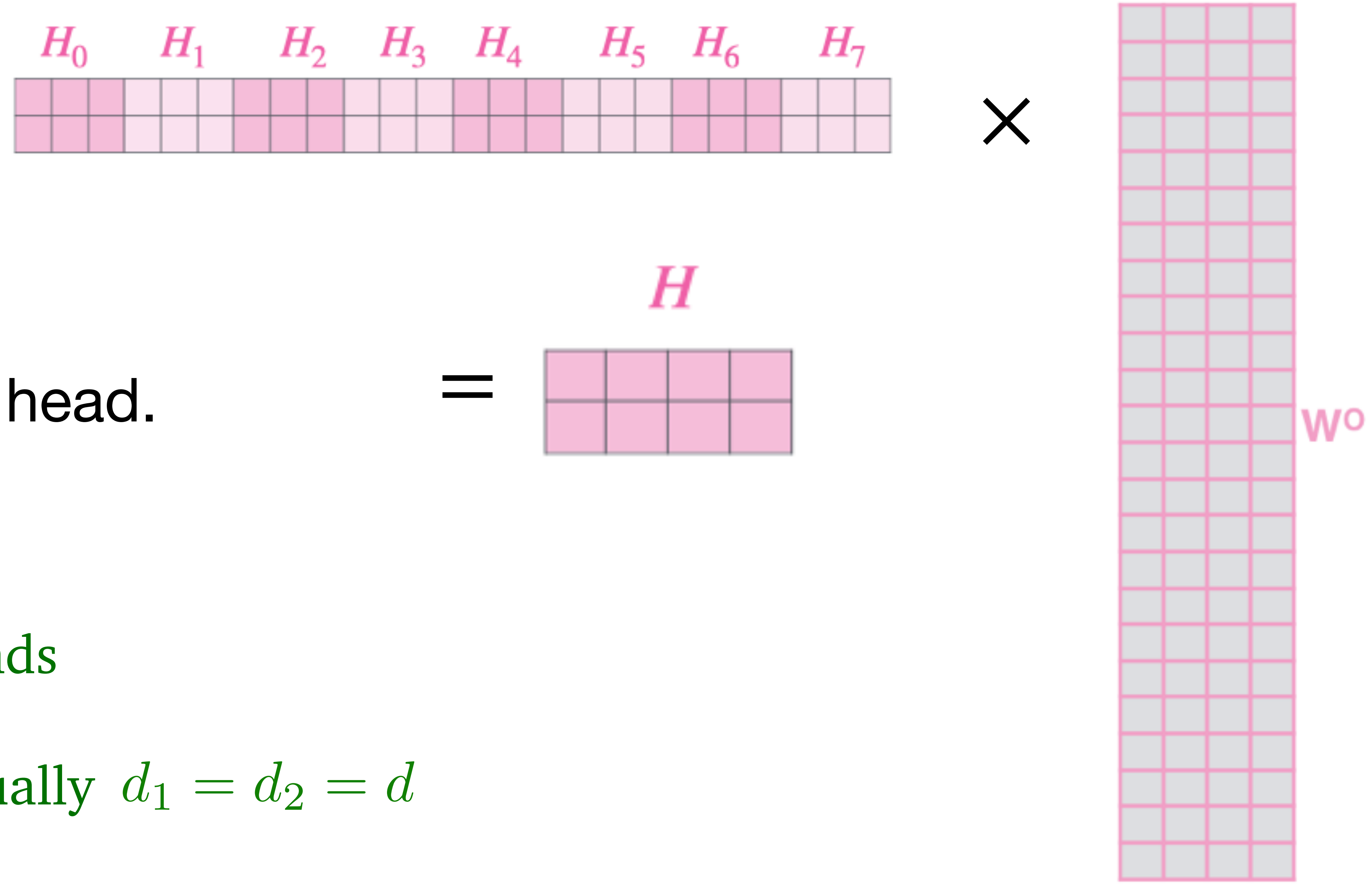
https://jalammar.github.io/illustrated-transformer/

Multi-head attention

"The Beast with Many Heads"

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

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



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

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

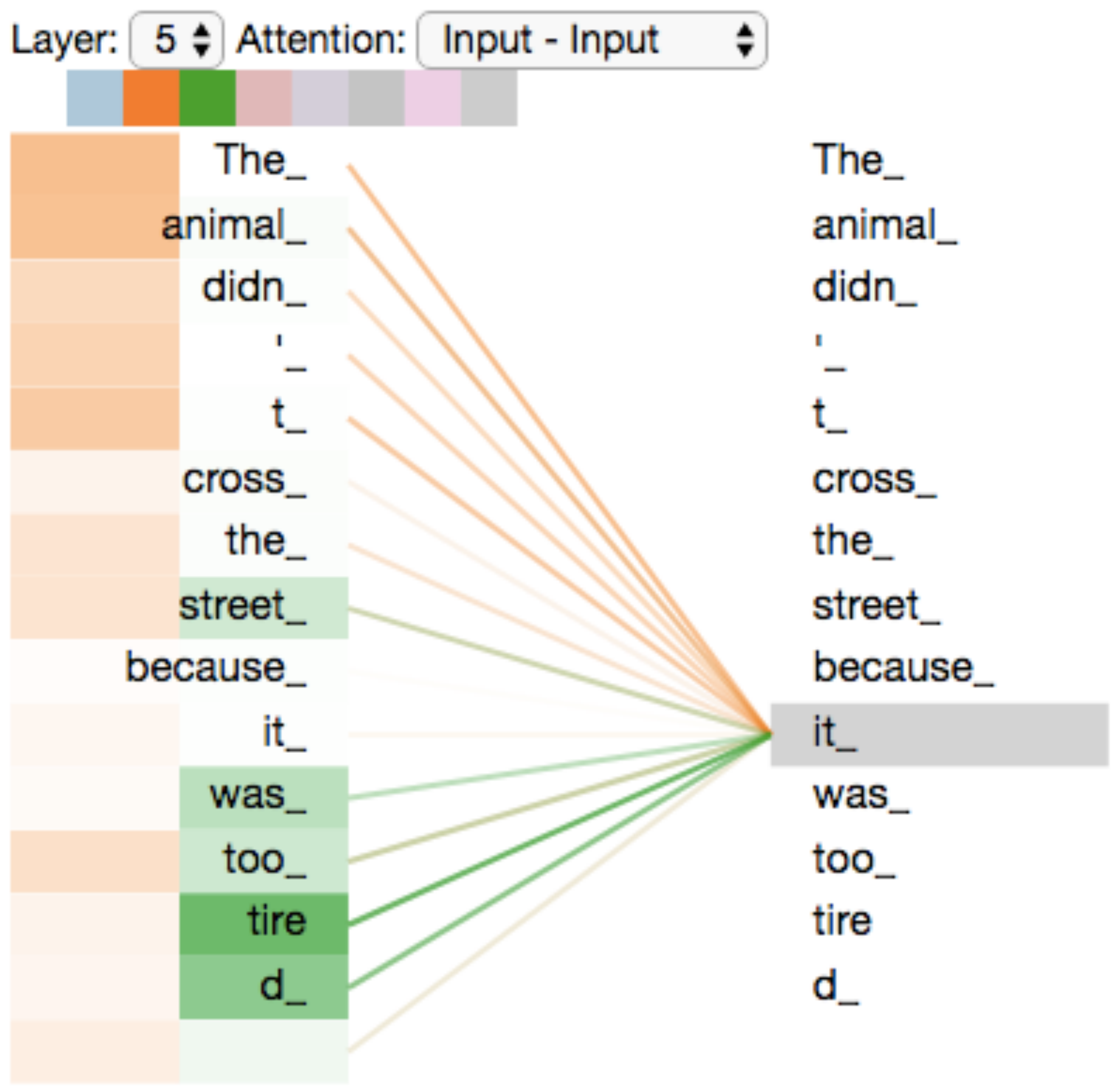
$$d_q = d_k = d_v = d/m$$ d = hidden size, m = # of heads

$$W^O \in \mathbb{R}^{d \times d_2}$$ 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.

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

What does multi-head attention learn?



https://github.com/jessevig/bertviz

Missing piece: positional encoding

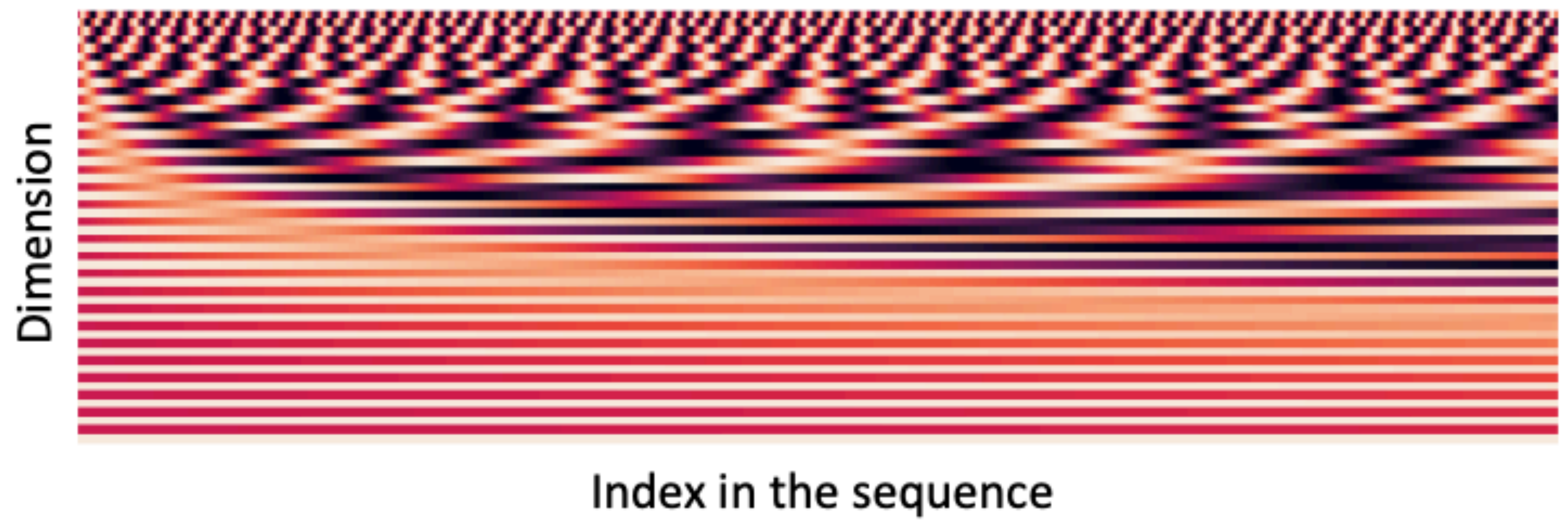
- Unlike RNNs, self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values

- Solution: Add **"positional encoding"** to the input embeddings: $\mathbf{p}_i \in \mathbb{R}^d$ for $i = 1,2,\ldots,n$

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

- **Sinusoidal position encoding**: sine and cosine functions of different frequencies:

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



    - **Pros**: Periodicity + can extrapolate to longer sequences
    - **Cons**: Not learnable

Missing piece: positional encoding

- **Learned absolute position encoding:** let all $\mathbf{p}_i$ be learnable parameters
  Similar to word embeddings
    - $P \in \mathbb{R}^{d \times L}$ for L = max sequence length
  - **Pros**: each position gets to be learned to fit the data
  - **Cons**: can't extrapolate to indices outside of max sequence length *L*
  - Most systems use this!

Self-Attention with Relative Position Representations

Peter Shaw
Google
petershaw@google.com

Jakob Uszkoreit
Google Brain
usz@google.com

Ashish Vaswani
Google Brain
avaswani@google.com

ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING

Jianlin Su
Zhuiyi Technology Co., Ltd.
Shenzhen
bojonesu@wezhuiyi.com

Yu Lu
Zhuiyi Technology Co., Ltd.
Shenzhen
julianlu@wezhuiyi.com

Shengfeng Pan
Zhuiyi Technology Co., Ltd.
Shenzhen
nickpan@wezhuiyi.com

Ahmed Murtadha
Zhuiyi Technology Co., Ltd.
Shenzhen
mengjiayi@wezhuiyi.com

Bo Wen
Zhuiyi Technology Co., Ltd.
Shenzhen
brucewen@wezhuiyi.com

Yunfeng Liu
Zhuiyi Technology Co., Ltd.
Shenzhen
glenliu@wezhuiyi.com

Feed-forward Network (MLP)

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

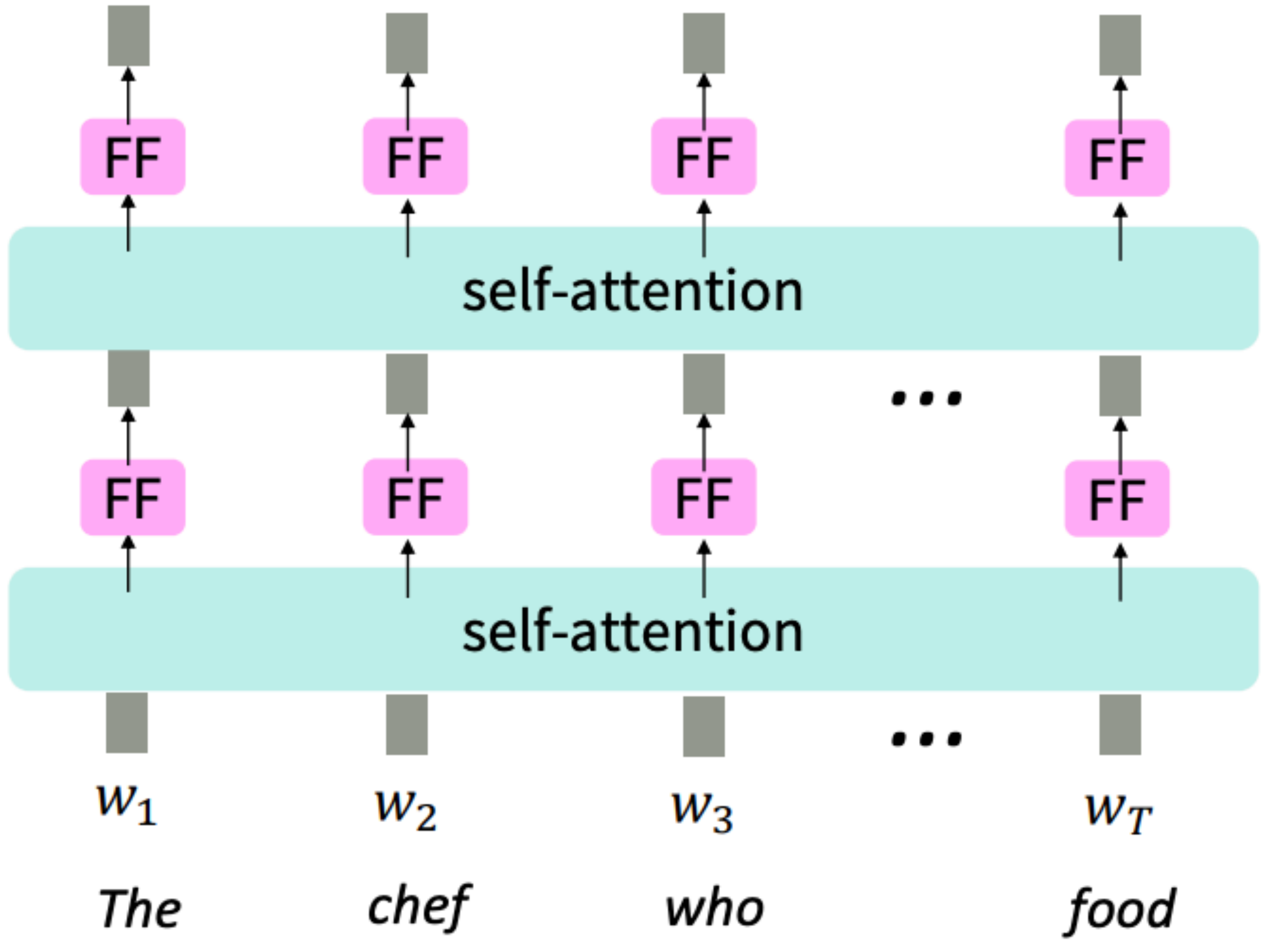
- Simple fix: add a feed-forward network to post-process each output vector

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

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

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

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



This is actually where the majority of the compute and parameters go!

Transformers vs LSTMs

Which of the following statements is correct?

(a) Transformers have less operations compared to LSTMs

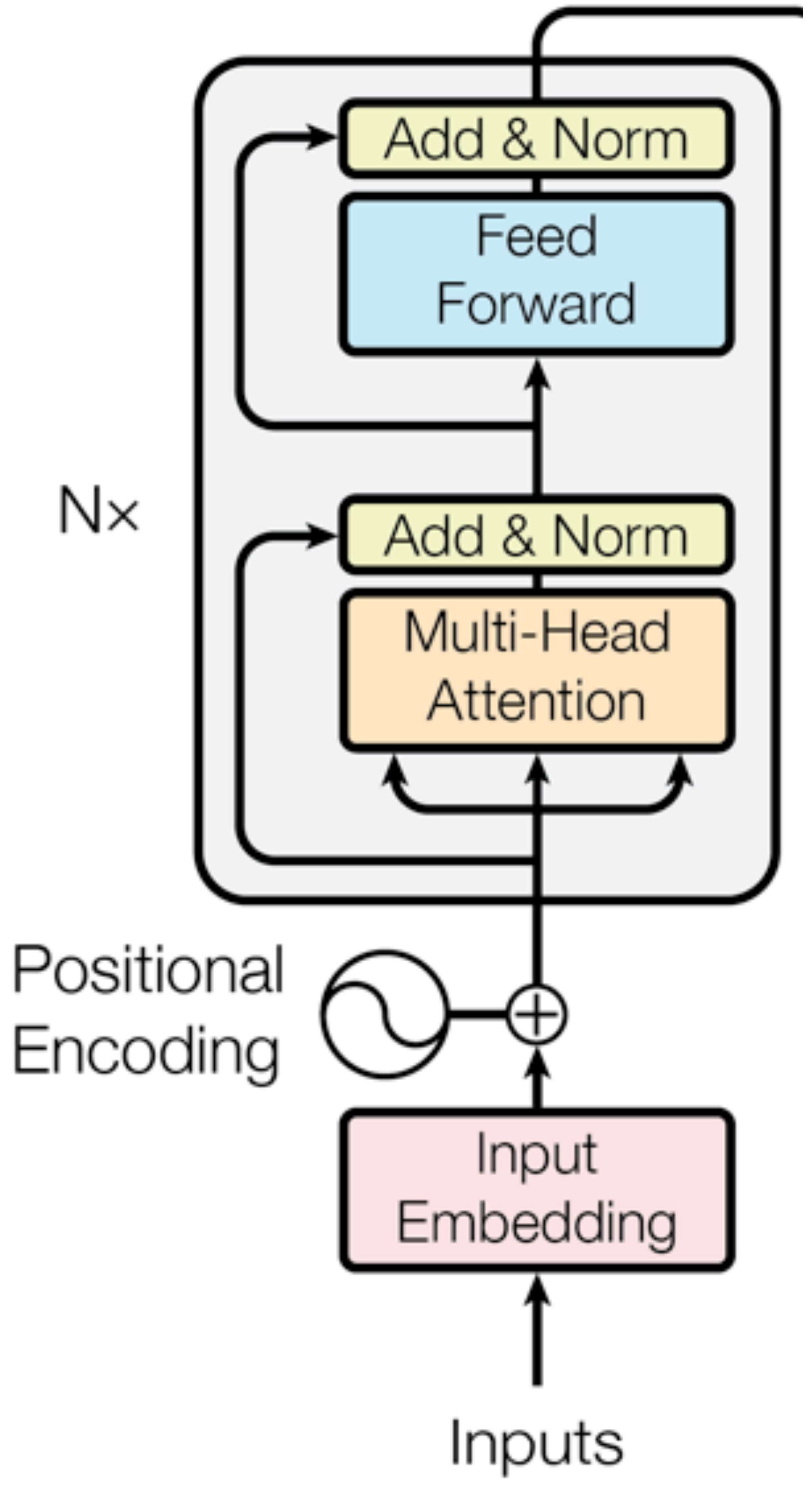
(b) Transformers are easier to parallelize compared to LSTMs

(c) Transformers have less parameters compared to LSTMs

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

(b) is correct.

Transformer encoder: let's put things together



From the bottom to the top:

- Input embedding
- Positional encoding
- A stack of Transformer encoder layers

Transformer encoder is a stack of N layers, which consists of two sub-layers:

- Multi-head attention layer
- Feed-forward layer

$$\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^{d_1} \longrightarrow \mathbf{h}_1, \ldots, \mathbf{h}_n \in \mathbb{R}^{d_2}$$

Residual connection & layer normalization

Add & Norm: $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$

Residual connections (He et al., 2016)

Instead of $X^{(i)} = \mathrm{Layer}(X^{(i-1)})$ (i represents the layer)

We let $X^{(i)} = X^{(i-1)} + \mathrm{Layer}(X^{(i-1)})$, so we only need to learn "the residual" from the previous layer

Gradient through the residual connection is 1 - good for propagating information through layers

Residual connection unlocks deep networks!

Residual connection & layer normalization

Add & Norm: $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$

Layer normalization (Ba et al., 2016) helps train model faster

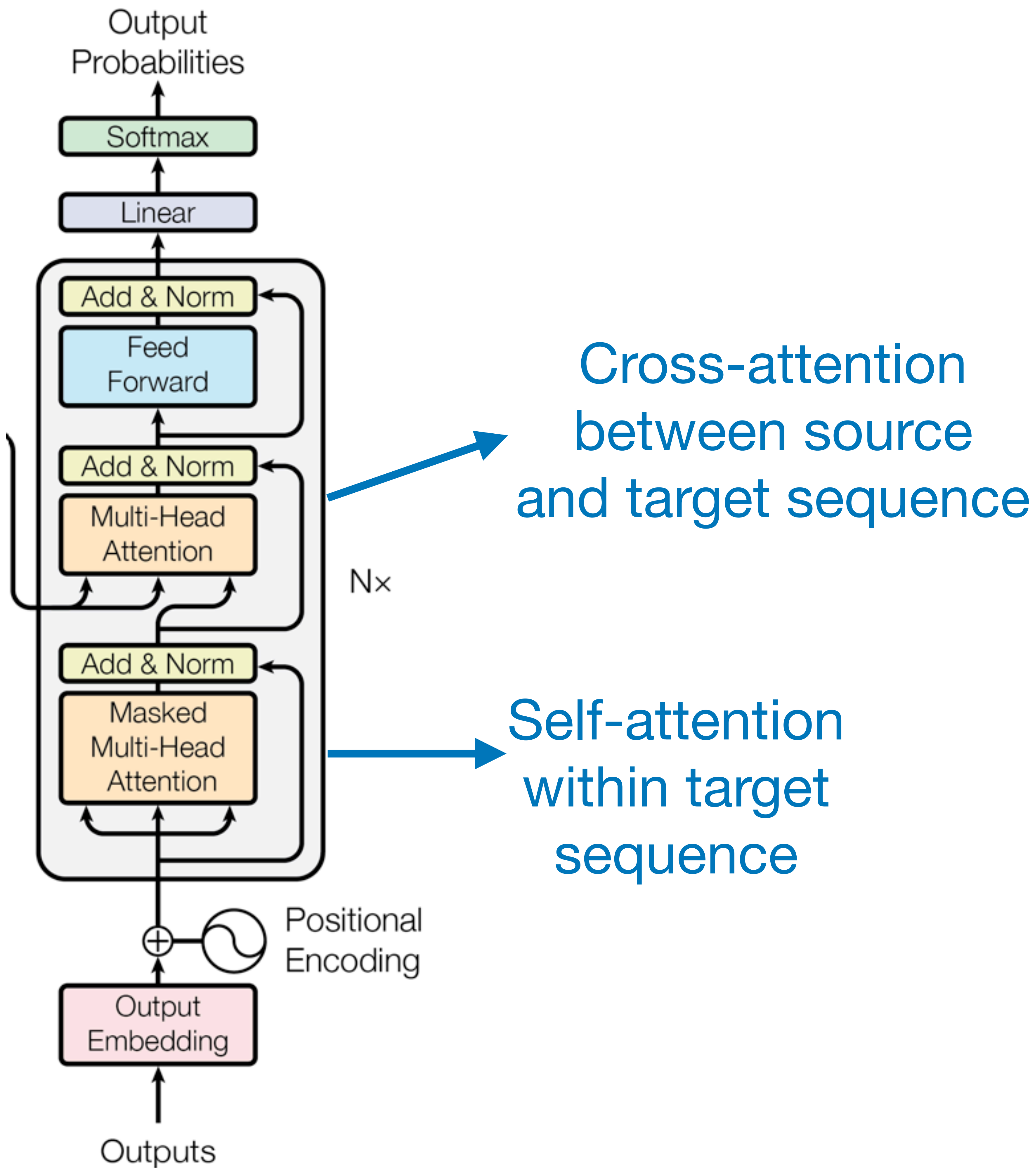
Idea: normalize the hidden vector values to unit mean and stand deviation within each layer

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

$\gamma, \beta \in \mathbb{R}^d$ are learnable parameters

LayerNorm is crucial for stable training with deep networks

Transformer decoder



From the bottom to the top:

- Output embedding
- Positional encoding
- A stack of Transformer decoder layers
- Linear + softmax

Transformer decoder is a stack of N layers, which consists of three sub-layers:

- Masked multi-head attention
- Multi-head cross-attention
- Feed-forward layer
- (W/ Add & Norm between sub-layers)