COS 484

Natural Language Processing

L14: Transformers (cont'd)

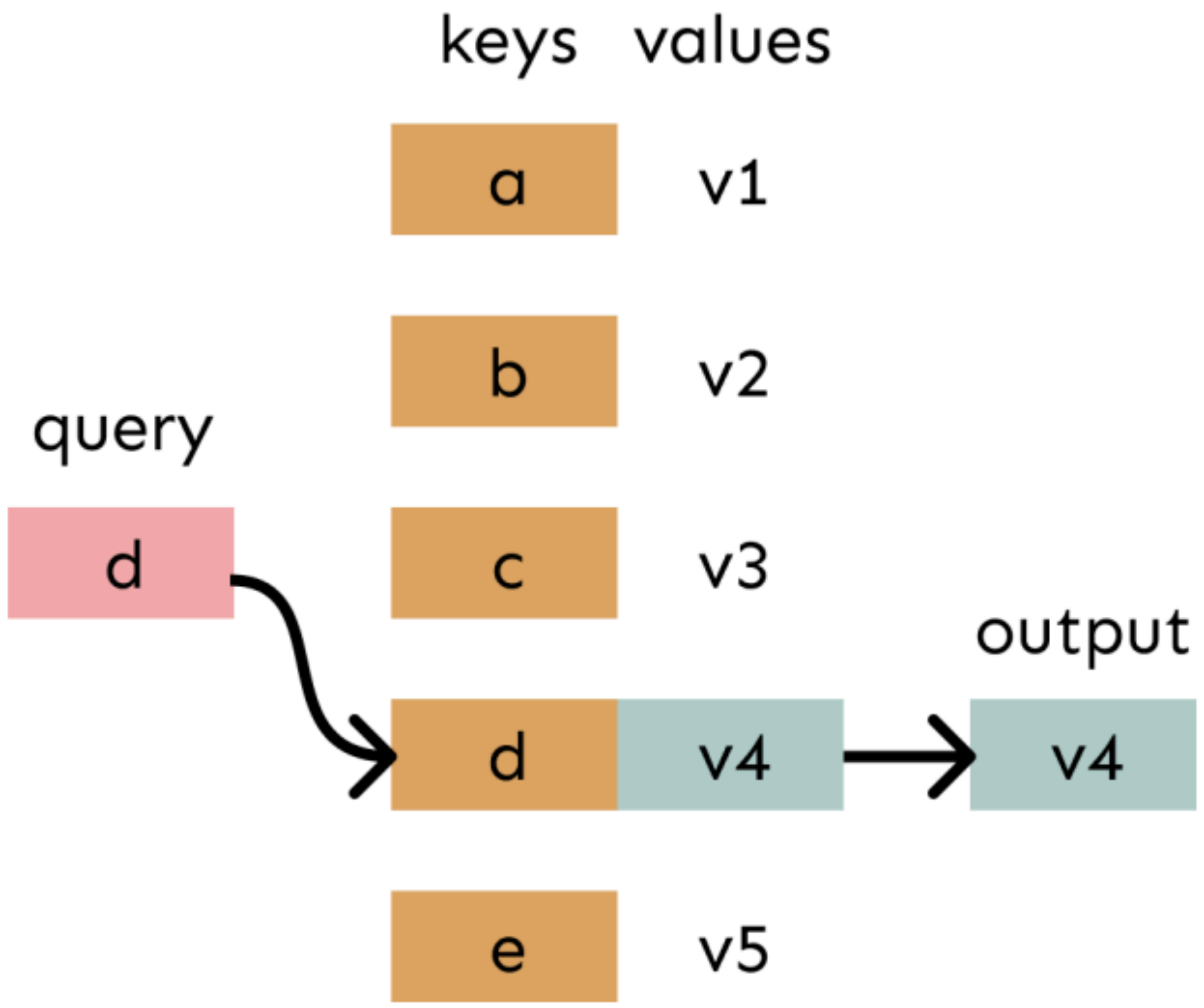
Spring 2024

(Some slides adapted from John Hewitt)

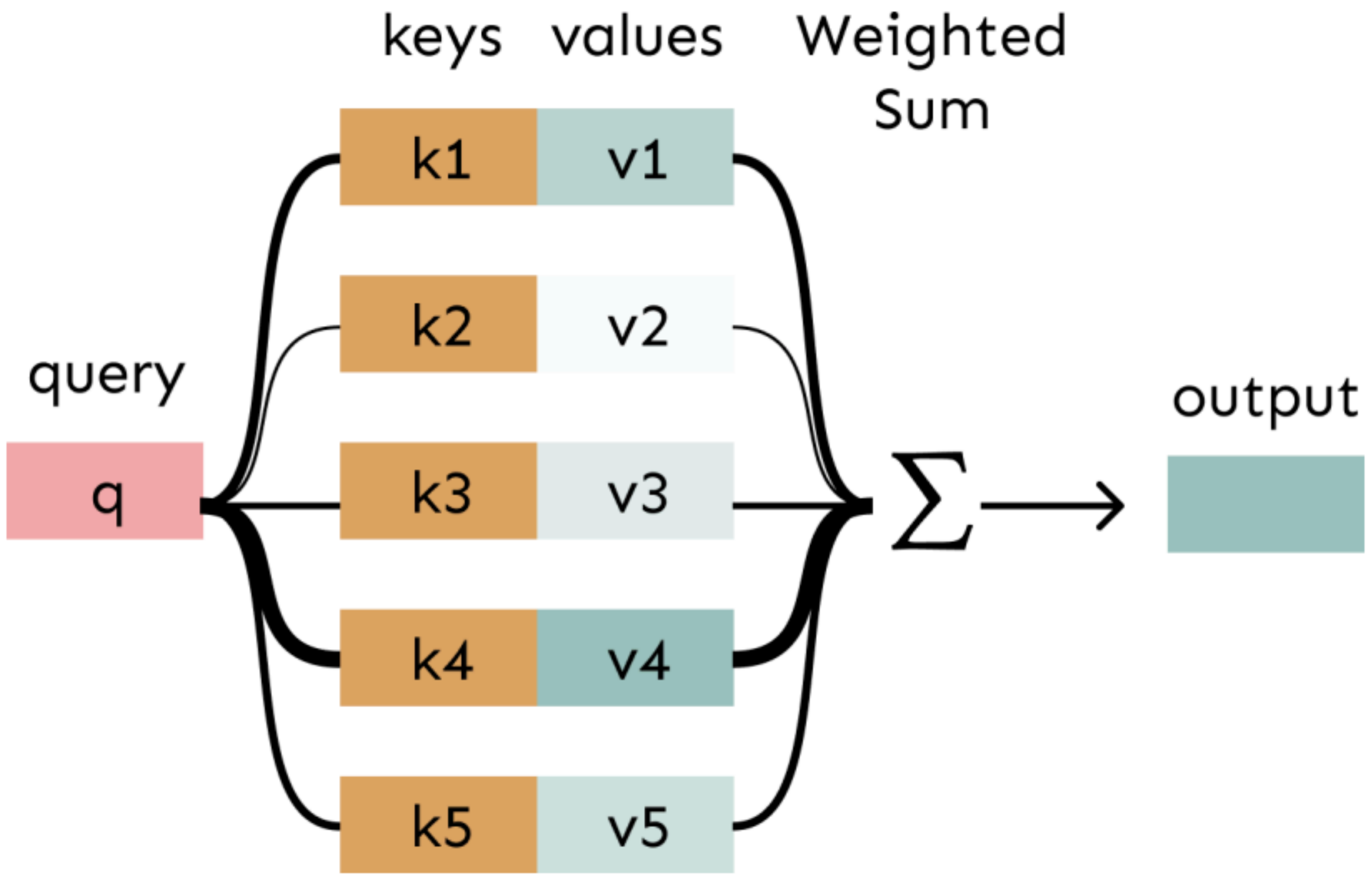
Attention as a soft, averaging lookup table

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 multipled by the weights and summed.



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

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

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

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

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

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

$$e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, n$$

$$\alpha_i = \mathrm{softmax}(\mathbf{e}_i)$$

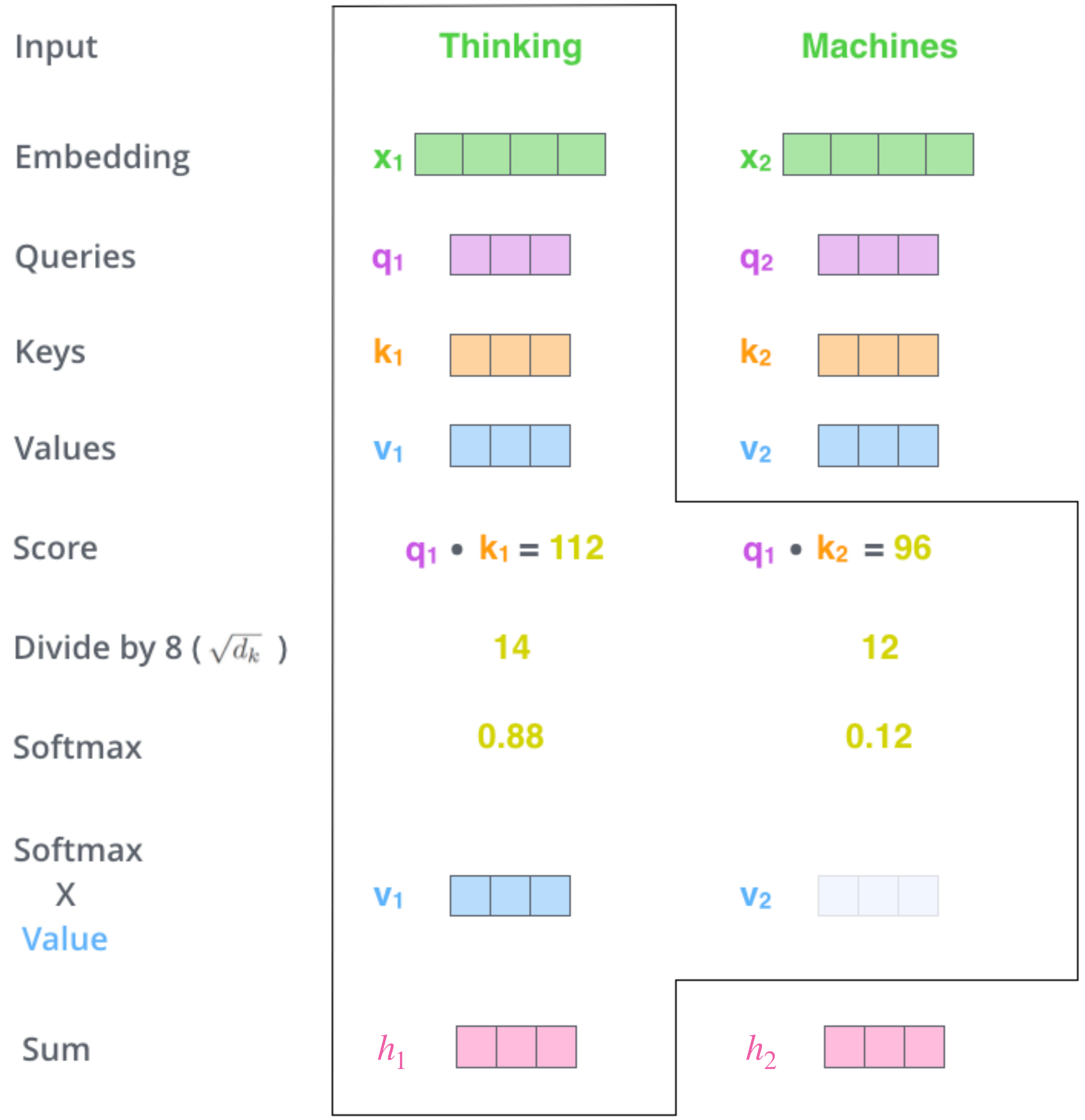
$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_{k=1}^{n} \exp(e_{i,k})}$$

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

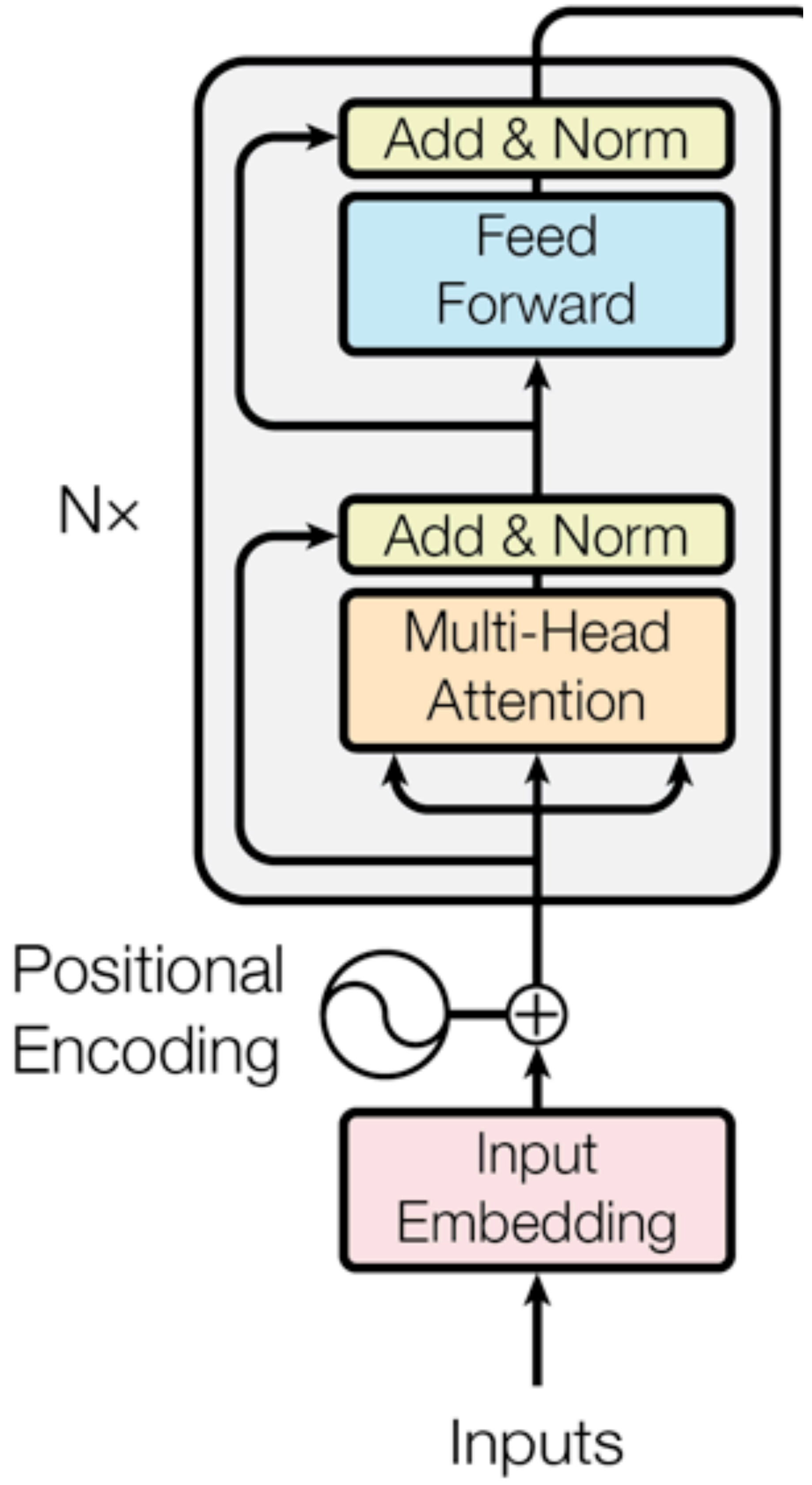
Step #3: Compute output for each input as weighted sum of values

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



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

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

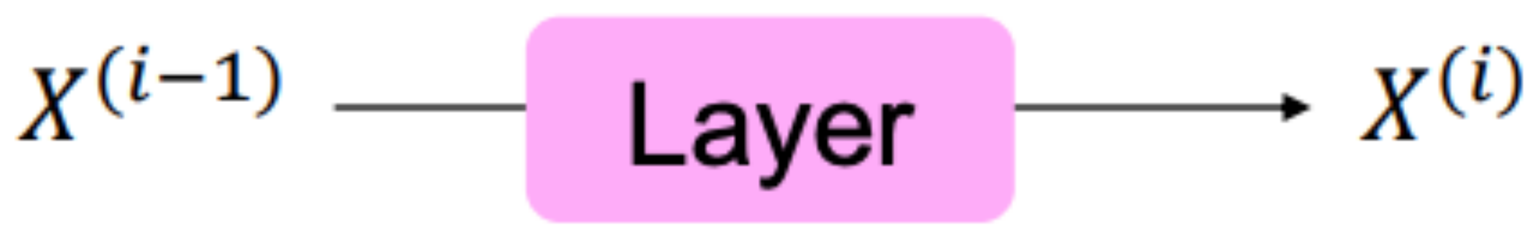
$$\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^{d_1} \longrightarrow \mathbf{h}_1, \ldots, \mathbf{h}_n \in \mathbb{R}^{d_2}$$

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 & layer normalization

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

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

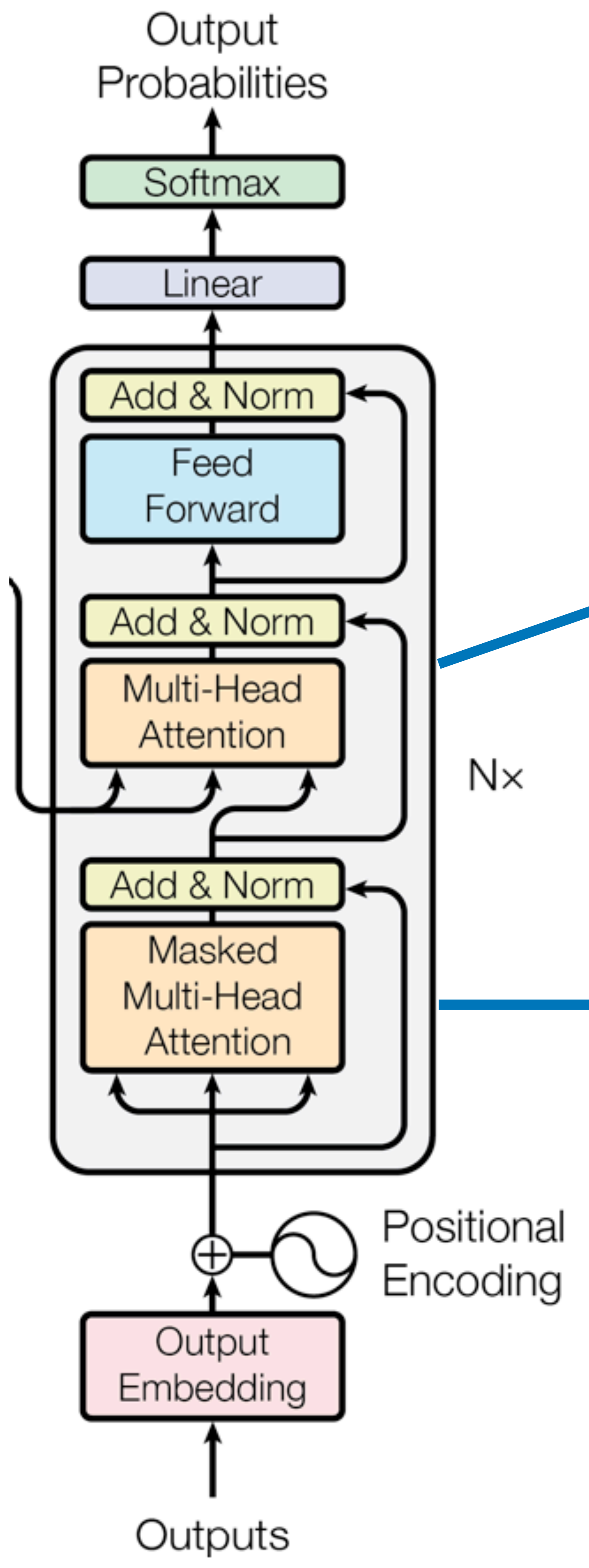
Idea: normalize the hidden vector values to unit mean and stand deviation within each layer

[advanced]

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

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

Transformer decoder



Cross-attention between source and target sequence

Self-attention within target sequence

From the bottom to the top:

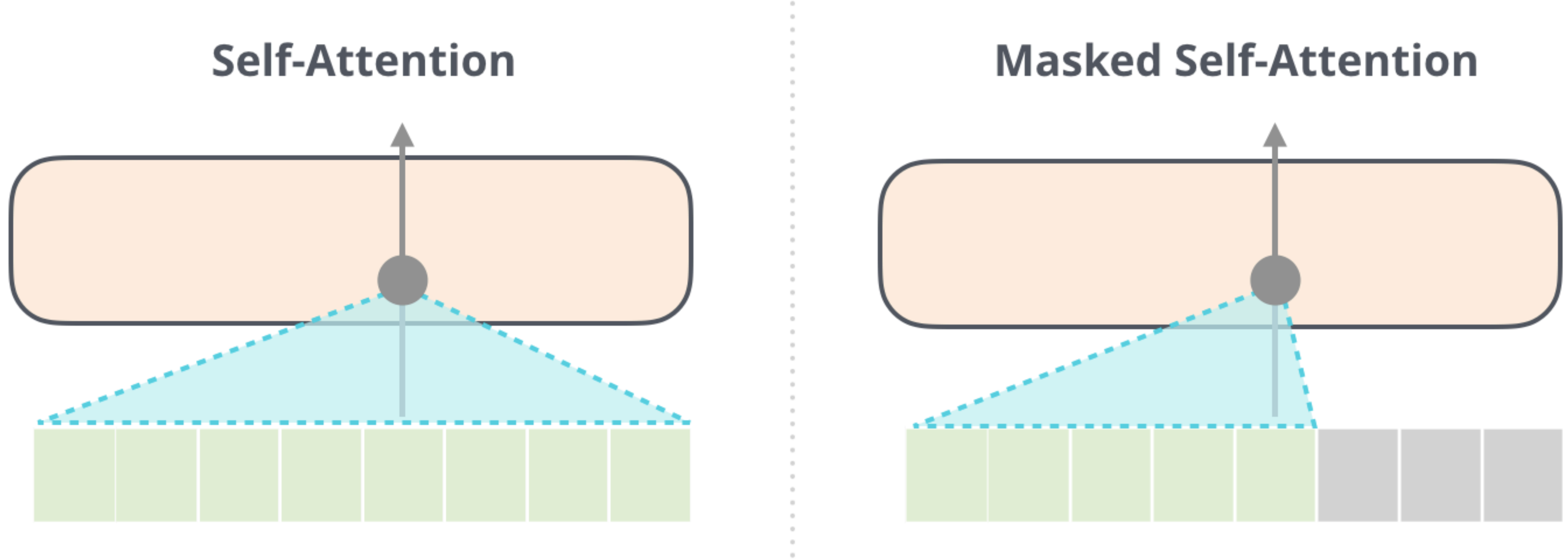
- Output embedding
- Positional encoding
- A stack of Transformer decoder layers
- Linear + softmax

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)

Masked (casual) self-attention

- Key: You can't see the future text for the decoder!



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

How to implement this? Masking!

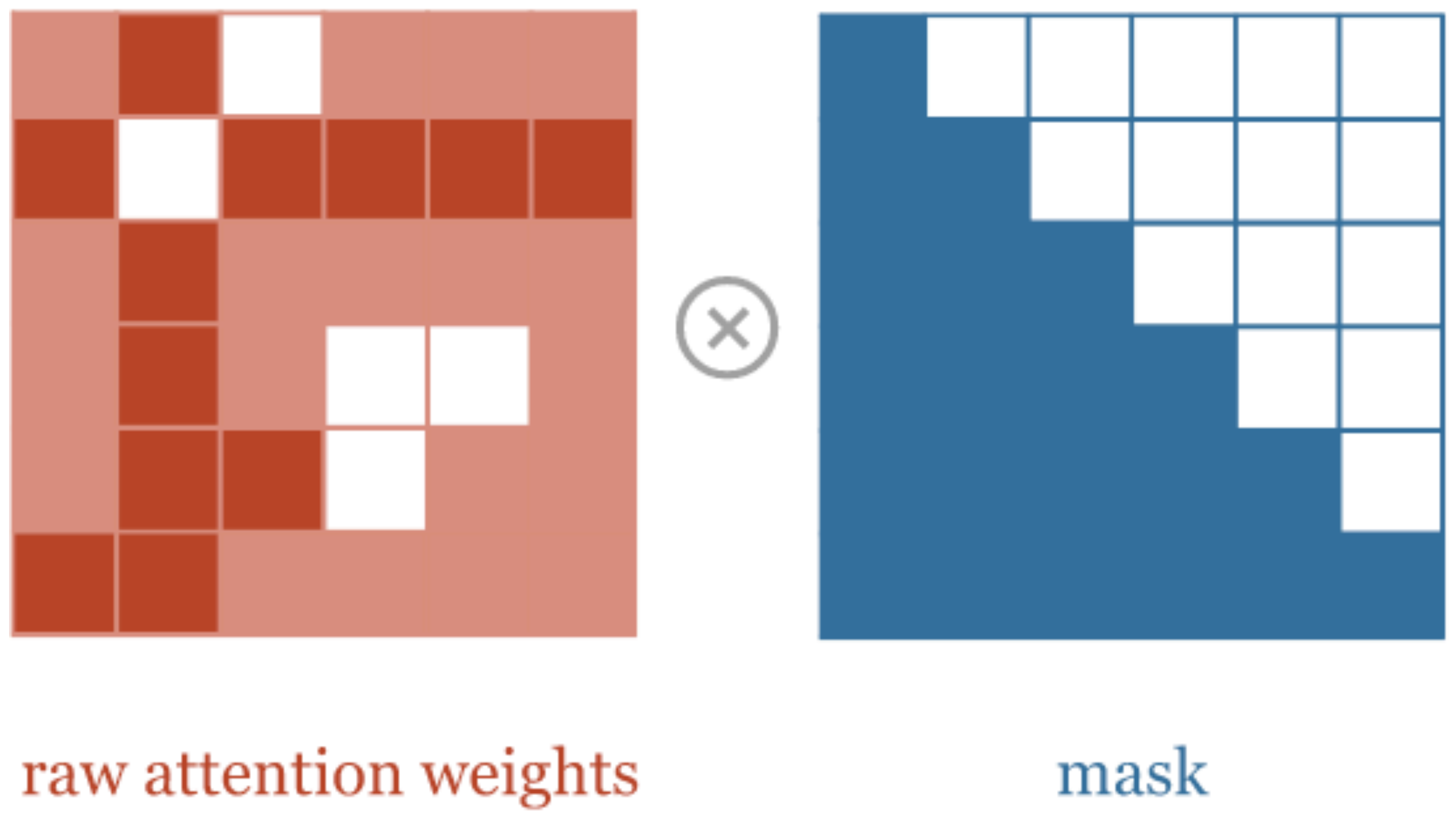
https://jalammar.github.io/illustrated-gpt2/

Masked multi-head attention

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

$$e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, n$$

$$\alpha_i = \mathrm{softmax}(\mathbf{e}_i)$$



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

Masked (multi-head) attention

The following matrix denotes the values of $\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$ for $1 \le i \le n, 1 \le j \le n \ (n = 4)$

1	0	-1	-1
1	1	-1	0
0	1	1	-1
-1	-1	2	1

What should be the value of $\alpha_{2,2}$ in masked attention?

(A) 0

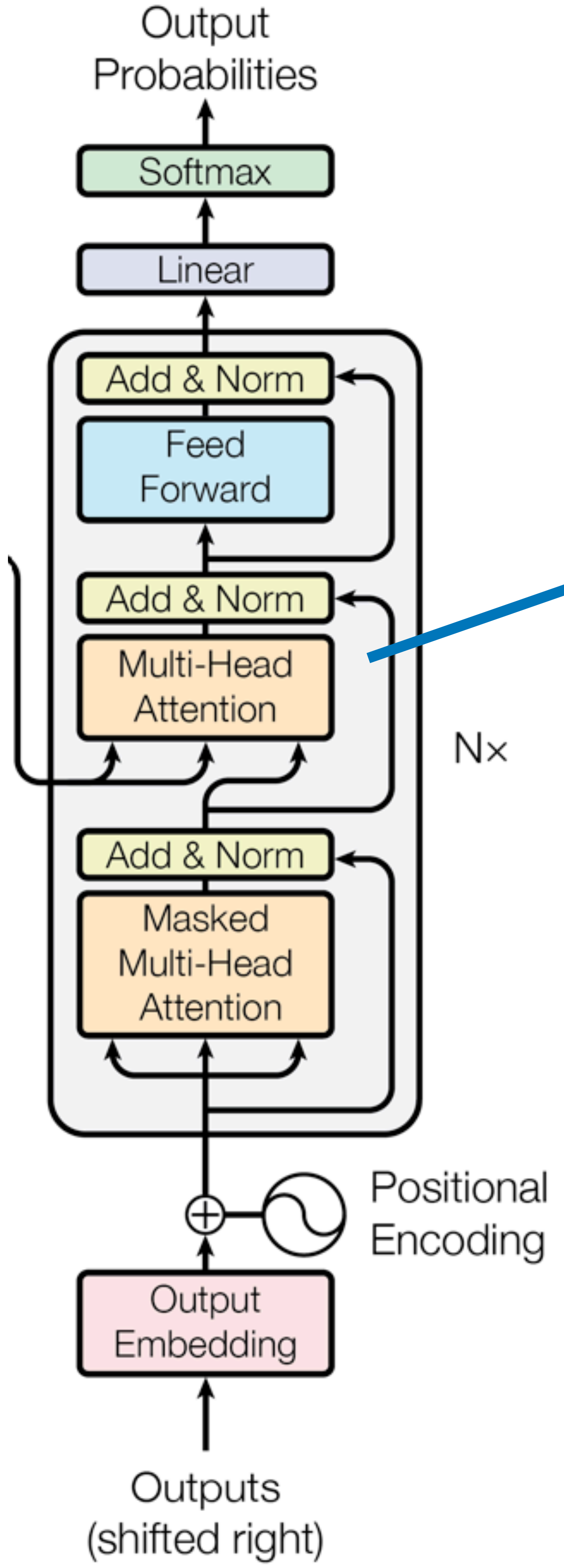
(B) 0.5

(C) $\frac{e}{2e + e^{-1} + 1}$

(D) 1

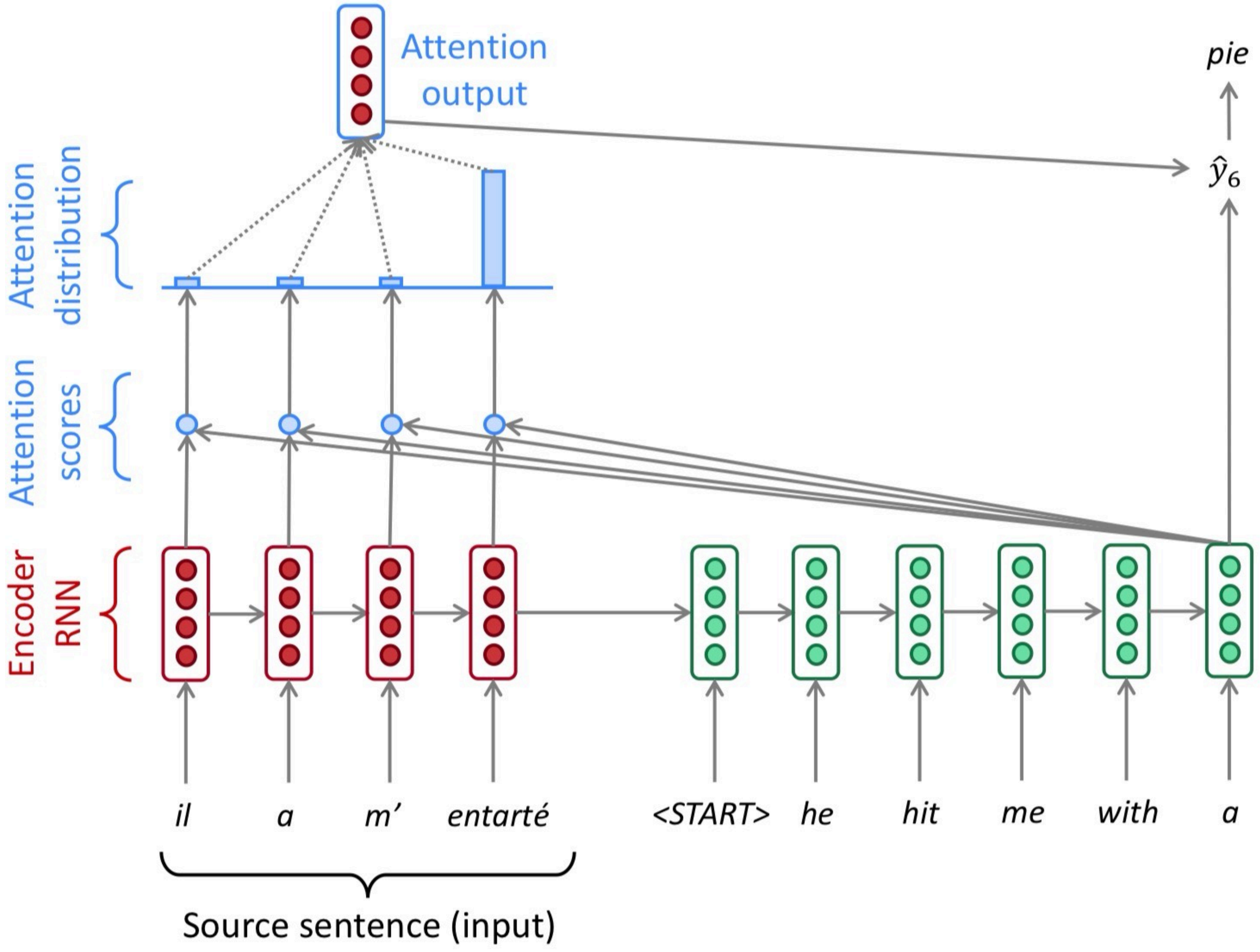
The correct answer is (B)

Multi-head cross-attention



Cross-attention between source and target sequence

Similar as the attention we learned in the previous lecture



Multi-head cross-attention

Self-attention:

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

$$e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, n$$

$$\alpha_i = \mathrm{softmax}(\mathbf{e}_i)$$

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

Cross-attention:

(always from the top layer)

$\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_m$: hidden states from encoder

$\mathbf{x}_1, \ldots, \mathbf{x}_n$: hidden states from decoder

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \quad i = 1,2,\ldots,n$$

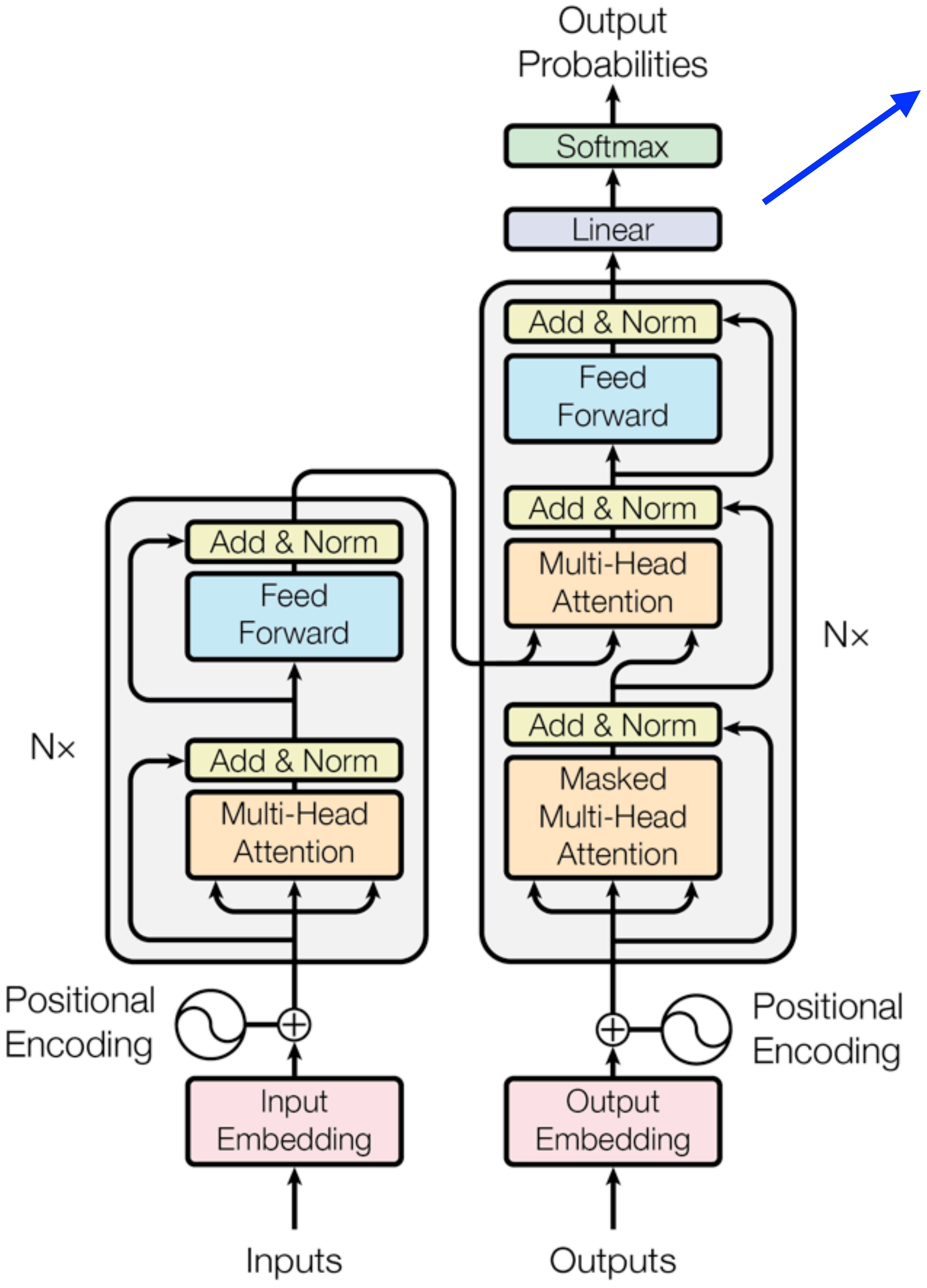
$$\mathbf{k}_j = \tilde{\mathbf{x}}_j \mathbf{W}^K, \mathbf{v}_j = \tilde{\mathbf{x}}_j \mathbf{W}^V \quad \forall j = 1,2,\ldots,m$$

$$e_{i,j} = \frac{\mathbf{q}_\mathbf{i} \cdot \mathbf{k}_j}{\sqrt{d_k}}, \forall j = 1, \ldots, m$$

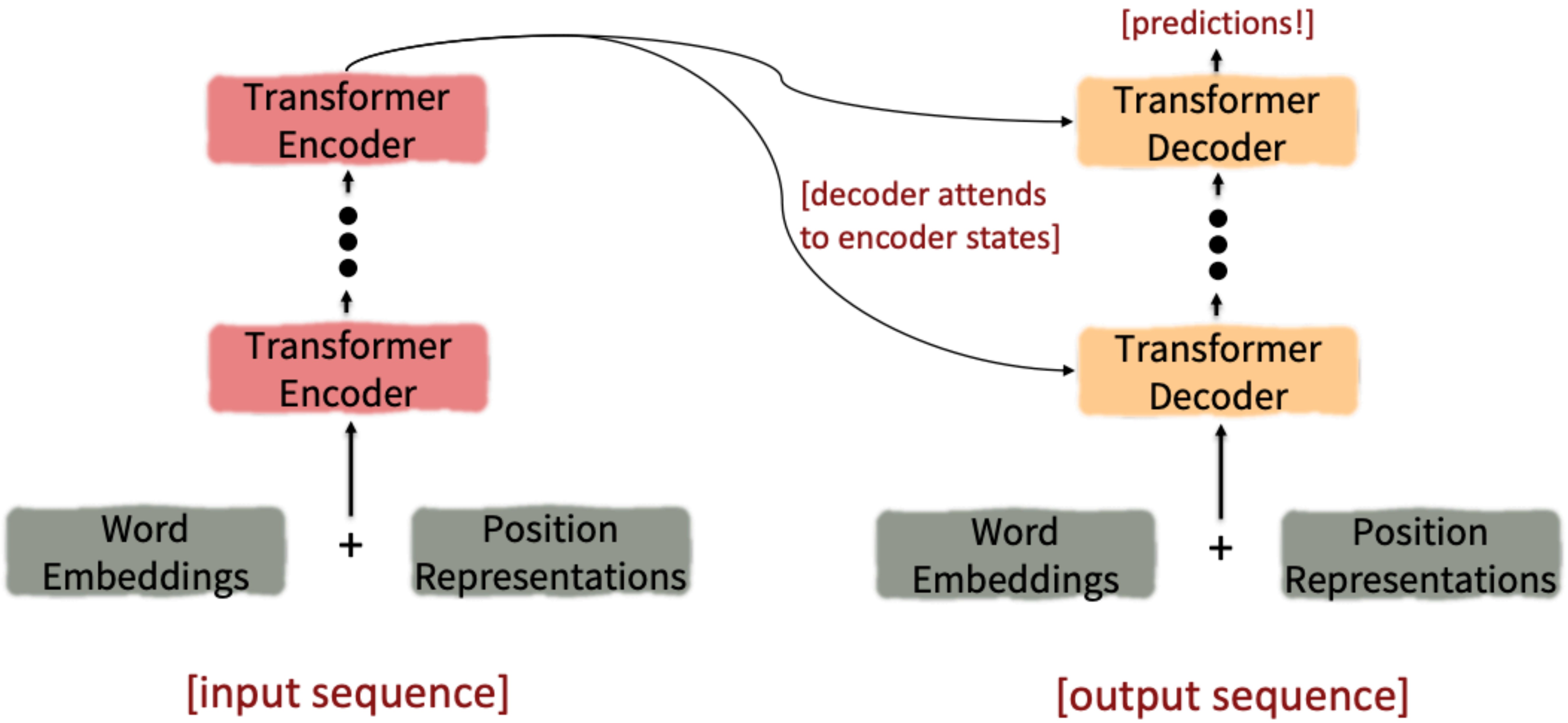
$$\alpha_i = \mathrm{softmax}(\mathbf{e}_i)$$

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

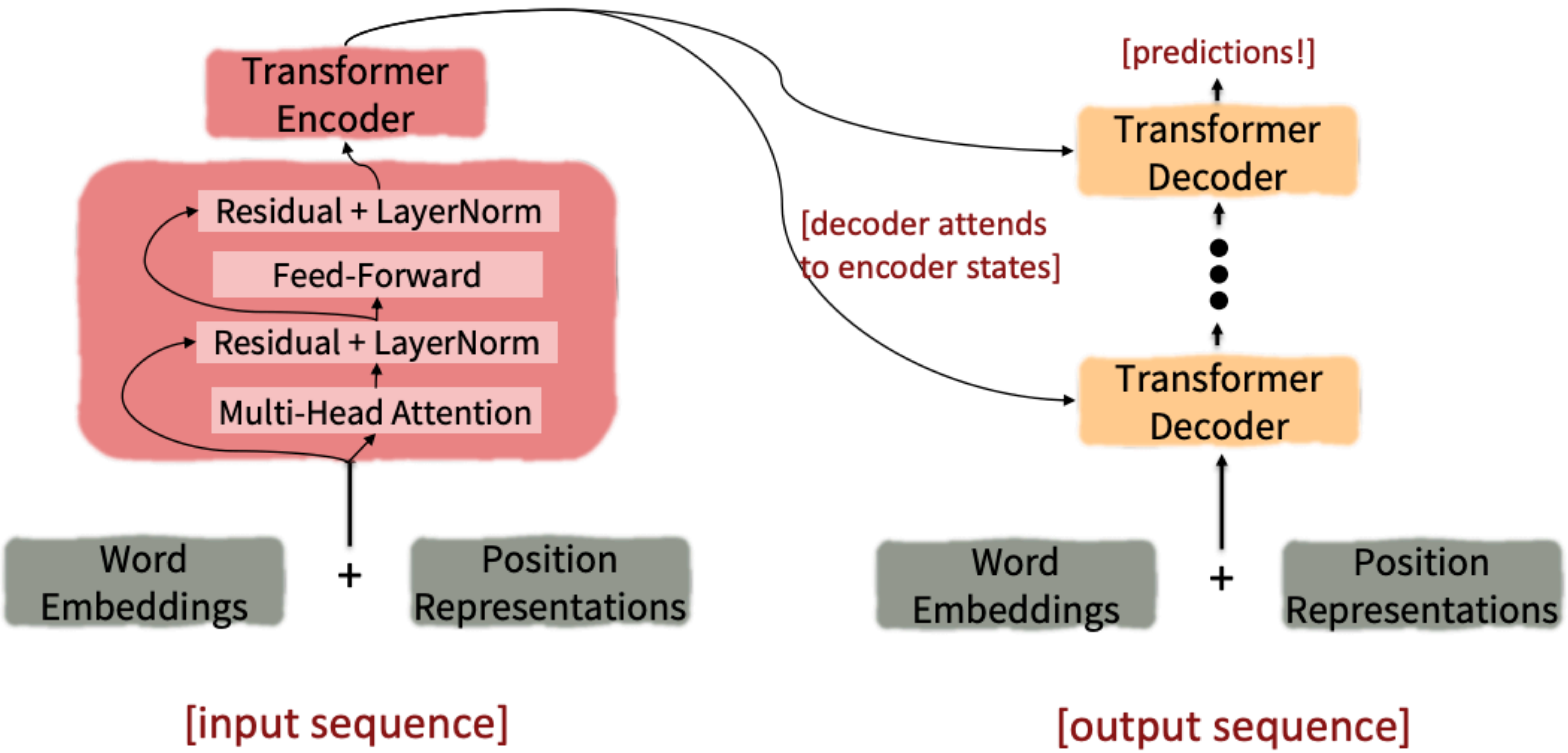
Transformer encoder-decoder



$\text{softmax}(\mathbf{W}_o \mathbf{h}_i)$

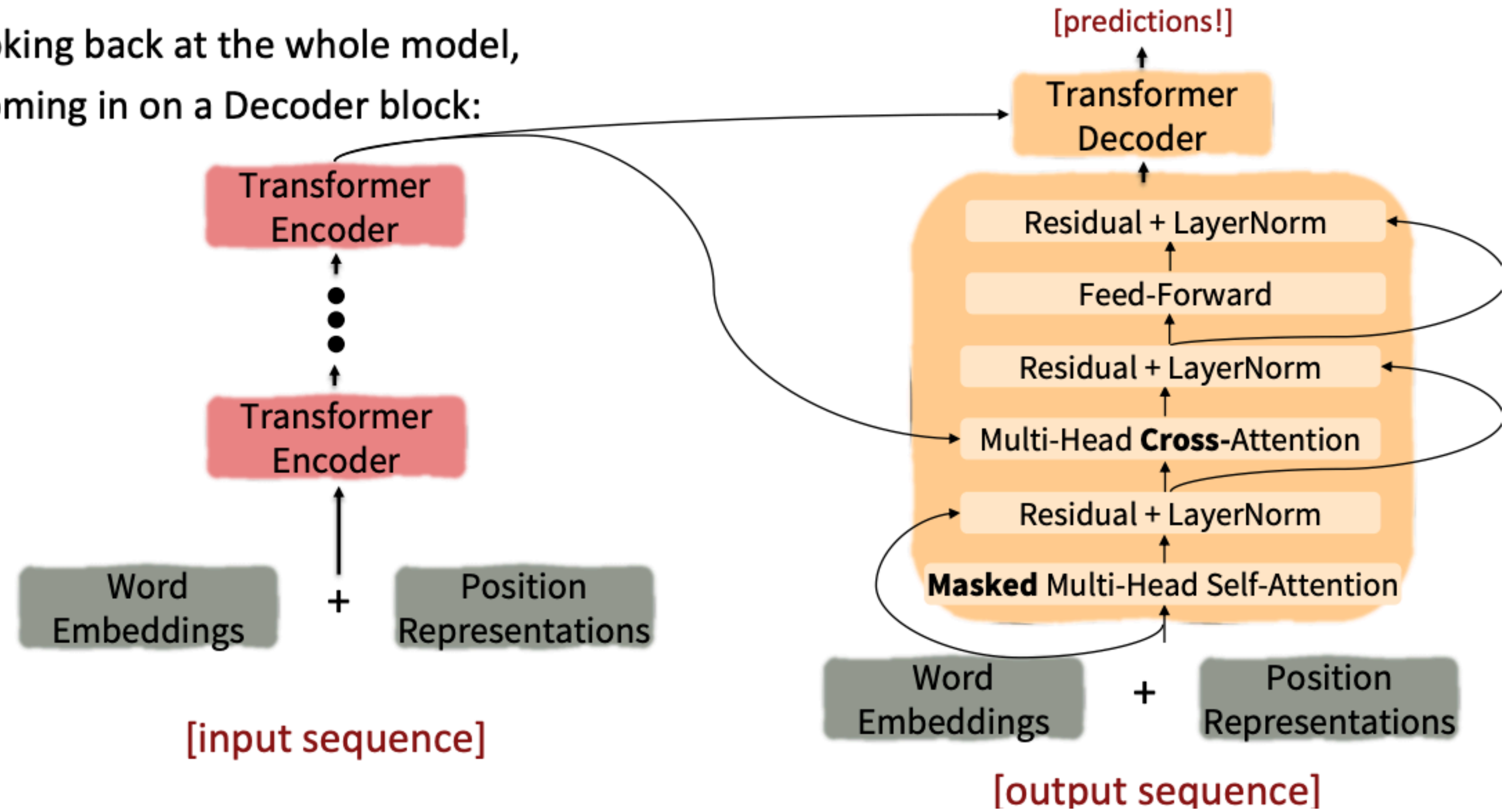


Transformer encoder-decoder



Transformer encoder-decoder

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



Training Transformer encoder-decoder models

The same as the way that we train seq2seq models before!

- Training data: parallel corpus $\{(\mathbf{w}_i^{(s)}, \mathbf{w}_i^{(t)})\}$
- Minimize cross-entropy loss:

$$\sum_{t=1}^{T} -\log P(y_t \mid y_1, \ldots, y_{t-1}, \mathbf{w}^{(s)})$$

(denote $\mathbf{w}^{(t)} = y_1, \ldots, y_T$)

- Back-propagate gradients through both encoder and decoder

12M sentence pairs

French: bonjour le monde .

English: hello world .

Masked self-attention is the key!

This can enable parallelizable operations while NOT looking at the future

Empirical results with Transformers

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)

Test sets: WMT 2014 English-German and English-French

Empirical results with Transformers

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

ED: encoder-decoder, D: decoder

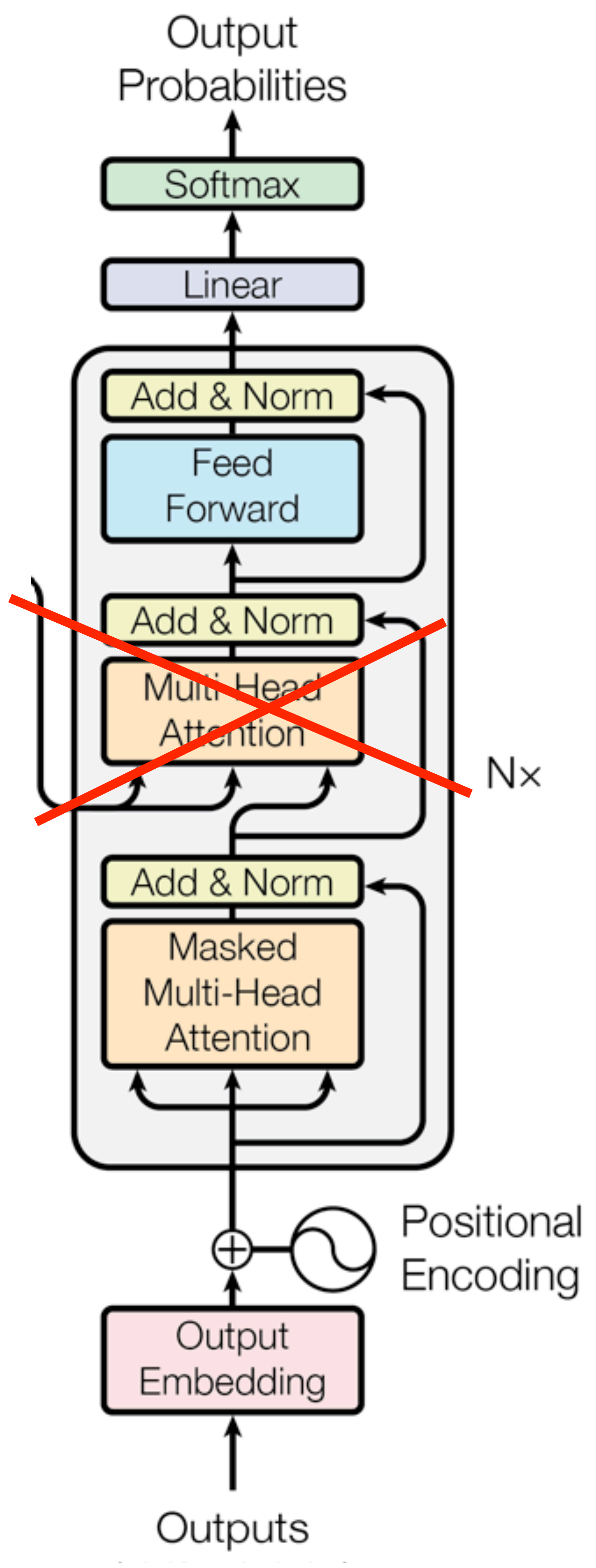
DMCA: decoder with memory-compressed attention

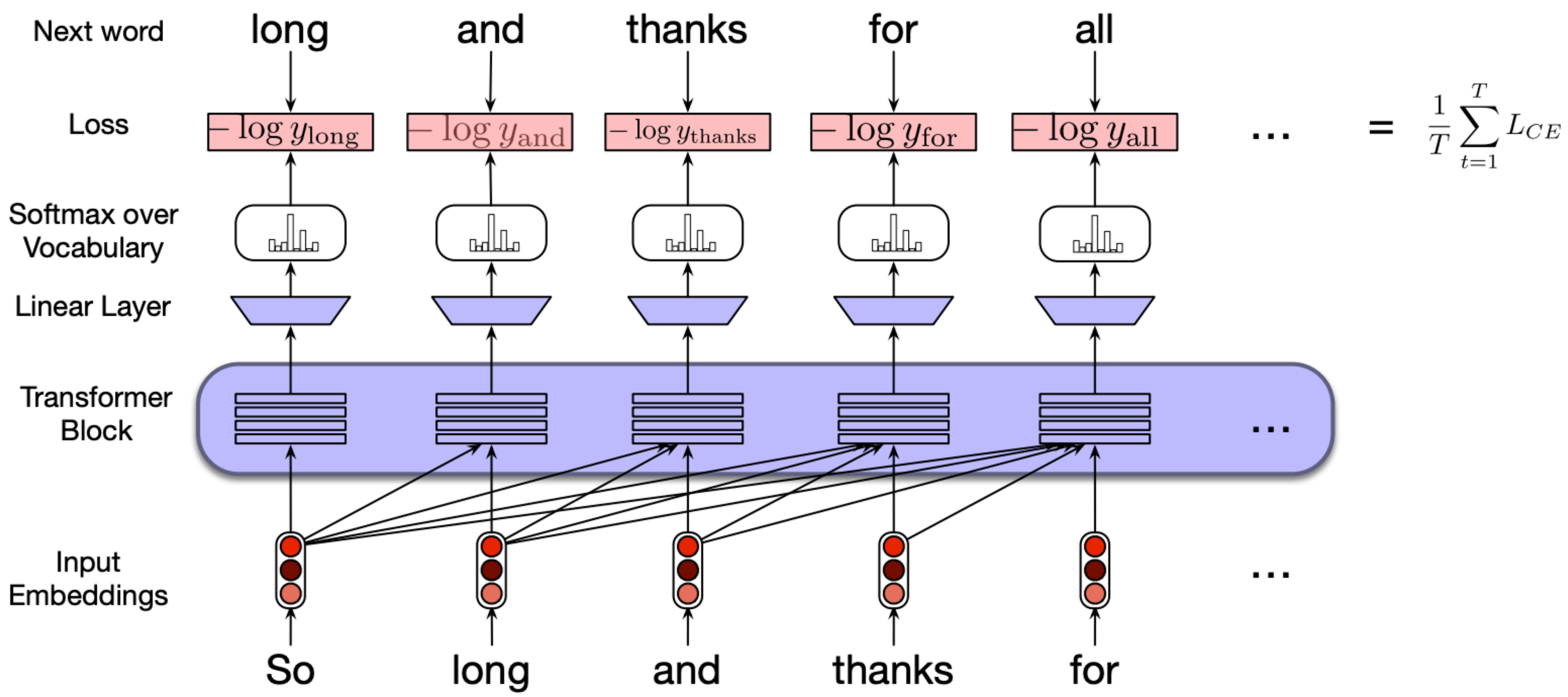
MoE: mixture of experts

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

Transformer-based language models

- The model architecture of GPT-3, ChatGPT, …





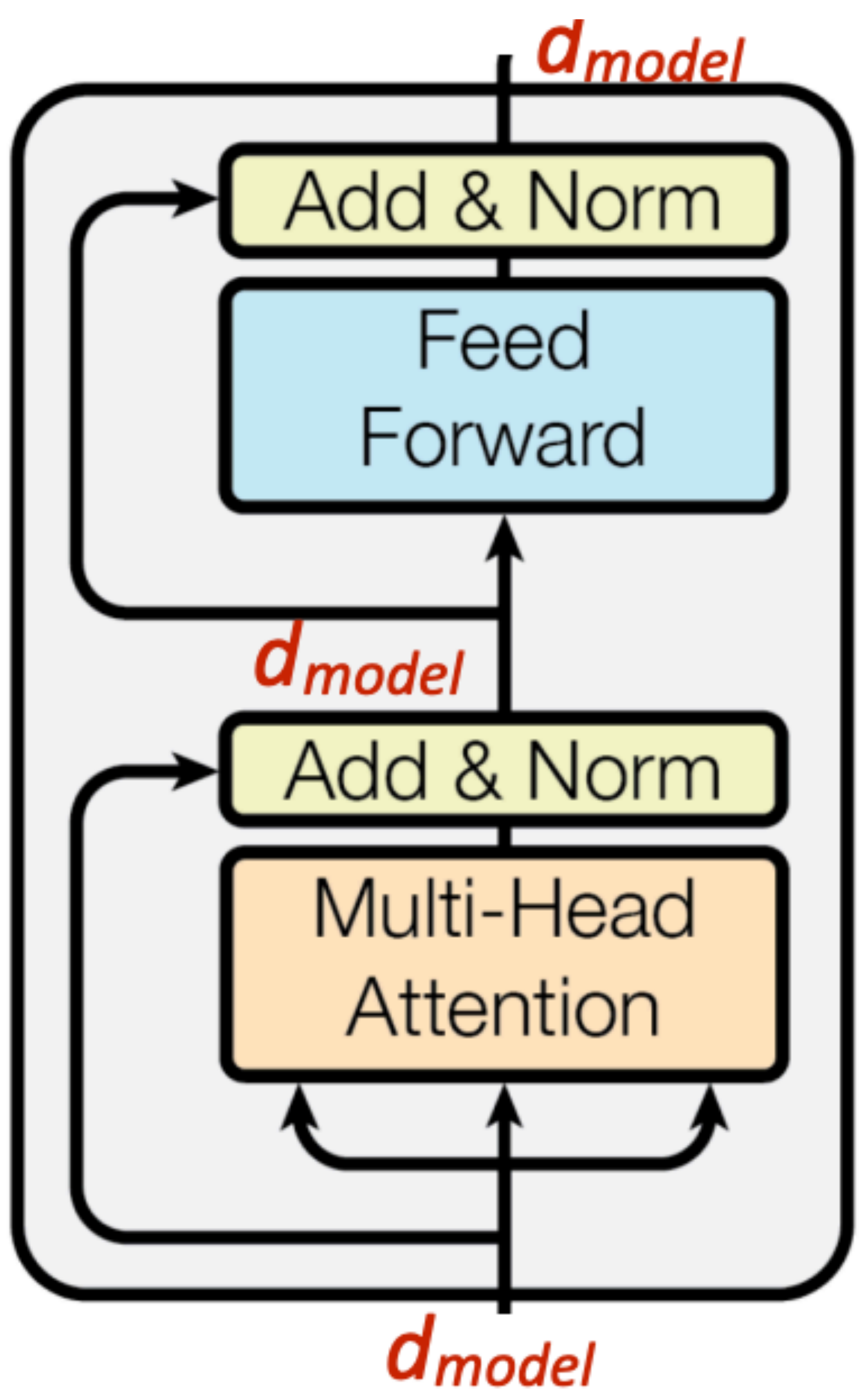
Transformer architecture specifications

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

- From Vaswani et al.

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}
GPT-3 Small	125M	12	768	12	64
GPT-3 Medium	350M	24	1024	16	64
GPT-3 Large	760M	24	1536	16	96
GPT-3 XL	1.3B	24	2048	24	128
GPT-3 2.7B	2.7B	32	2560	32	80
GPT-3 6.7B	6.7B	32	4096	32	128
GPT-3 13B	13.0B	40	5140	40	128
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128

- From GPT-3; d_{head} is our d_k



The Annotated Transformer

The Annotated Transformer

Attention is All You Need

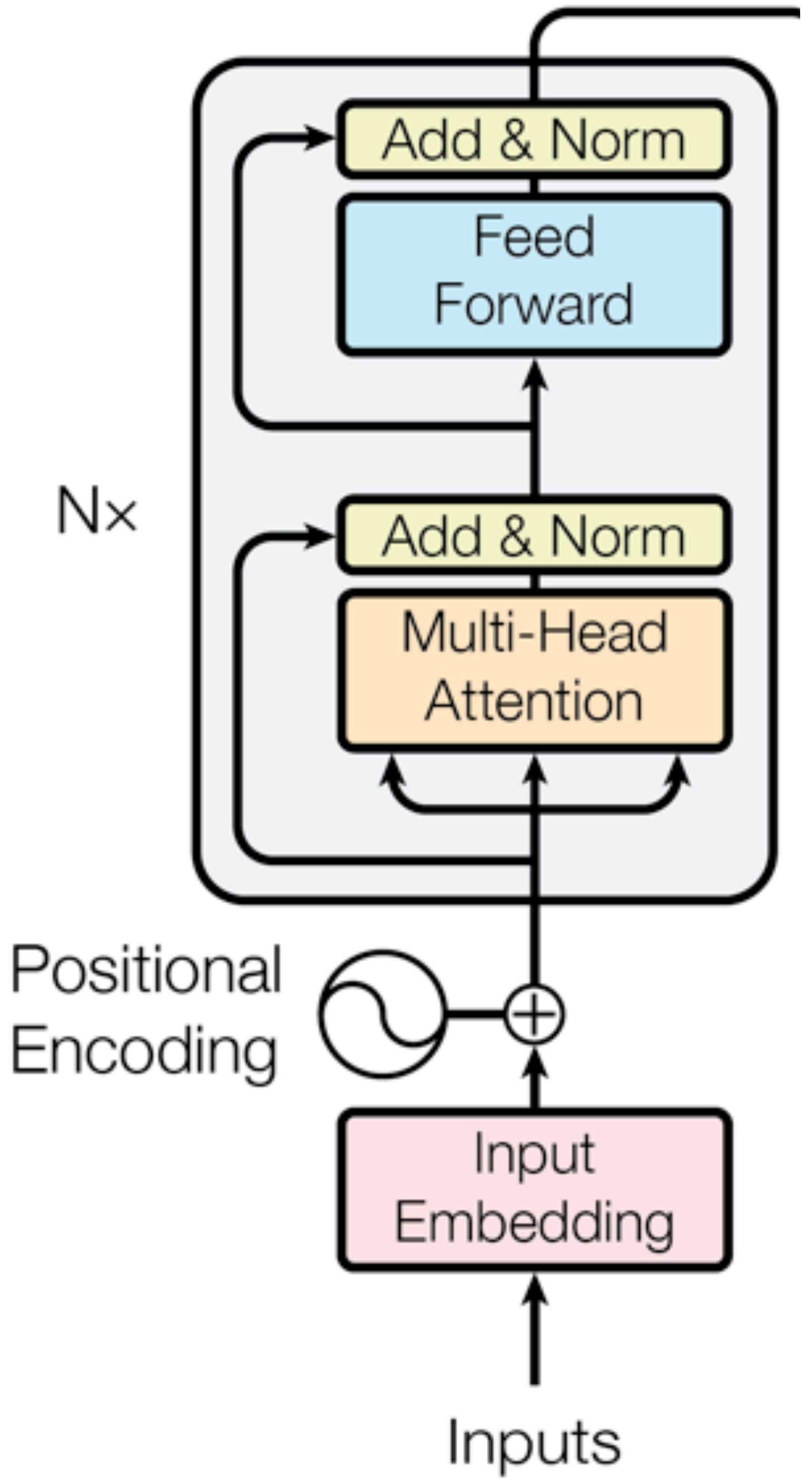
Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

- *v2022: Austin Huang, Suraj Subramanian, Jonathan Sum, Khalid Almubarak, and Stella Biderman.*
- *Original: Sasha Rush.*

Table of Contents

http://nlp.seas.harvard.edu/annotated-transformer/

Understanding Transformers



Which of the following is CORRECT?

(A) Multi-head attention is more computationally expensive than feedforward layers

(B) Multi-head attention is more computationally expensive than single-head attention

(C) It is hard to apply Transformers to sequences that are longer than the pre-defined max_seq_length L

(D) We can easily scale Transformers to long sequences

The correct answer is (C)

Transformers: pros and cons

- **Easier to capture long-range dependencies**: we draw attention between every pair of words!
- **Easier to parallelize:**

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

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

- **Are positional encodings enough to capture positional information?**

Otherwise self-attention is an unordered function of its input

- **Quadratic computation in self-attention**

Can become very slow when the sequence length is large

Quadratic computation as a function of sequence length

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

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

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

$n \times d_v$

Need to compute n^2 pairs of scores (= dot product) $\quad$ $O(n^2 d)$

RNNs only require $O(nd^2)$ running time:

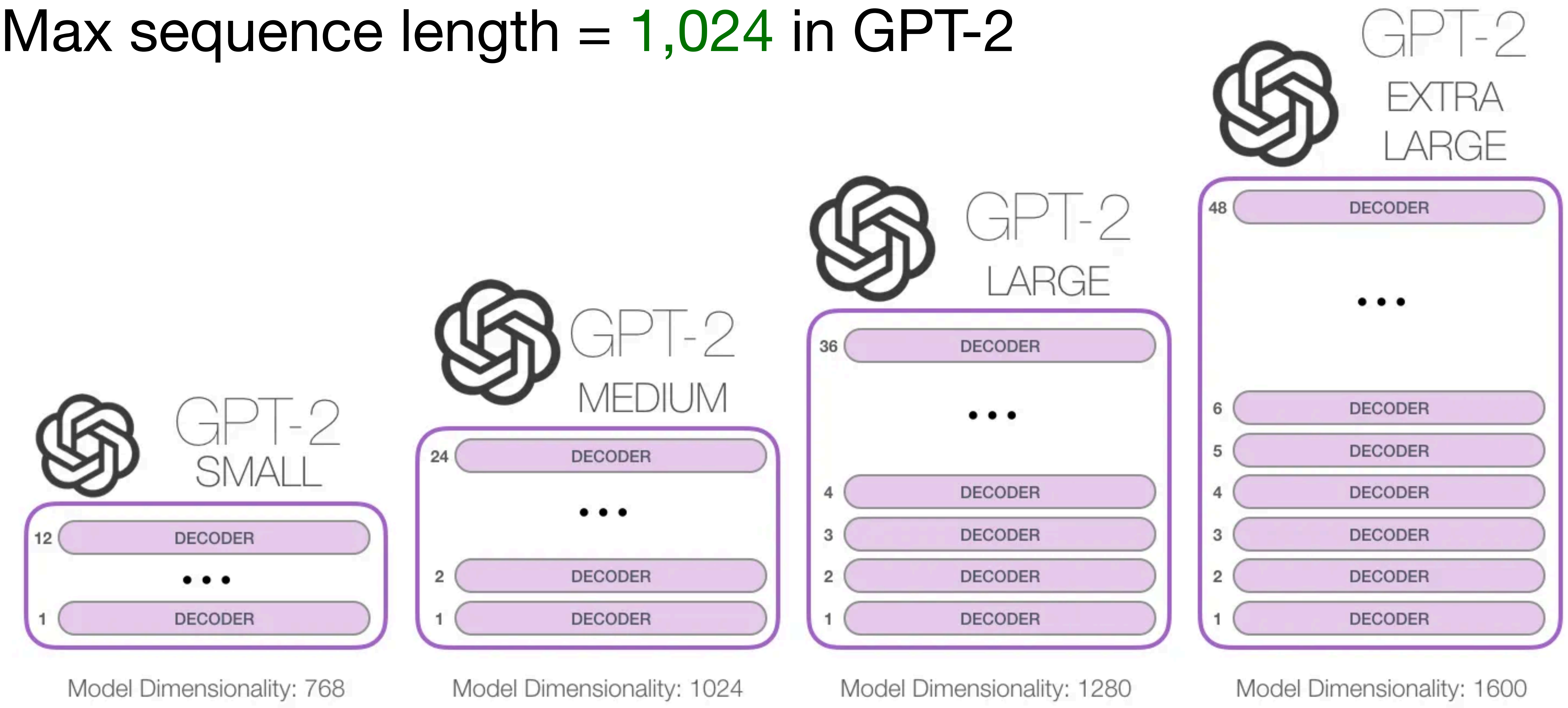
$$\mathbf{h}_t = g(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

(assuming input dimension = hidden dimension = d)

Quadratic computation as a function of sequence length

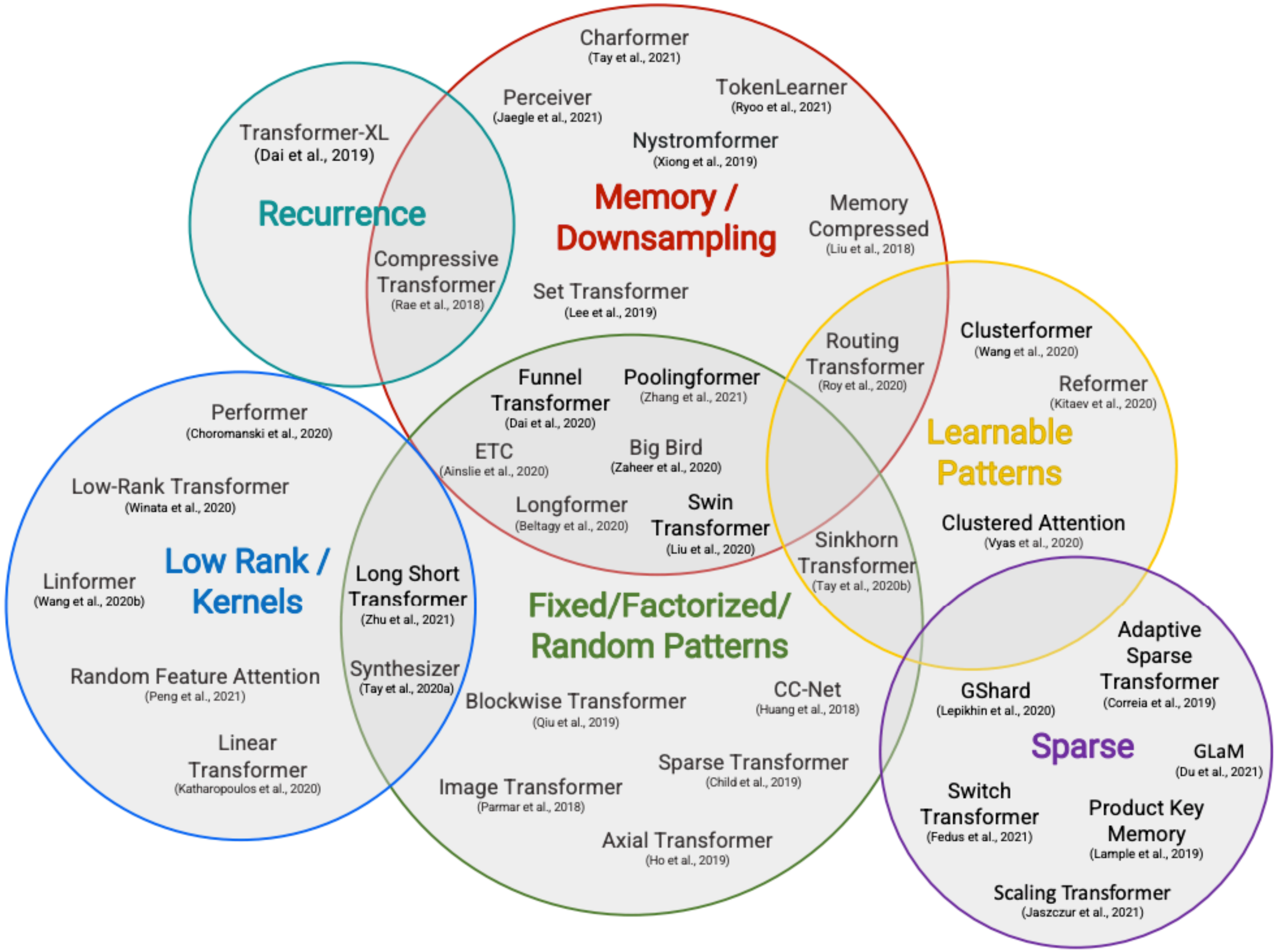
Need to compute n^2 pairs of scores (= dot product) $O(n^2 d)$

Max sequence length = 1,024 in GPT-2



What if we want to scale $n \geq 50{,}000$? For example, to work on long documents?

Efficient Transformers



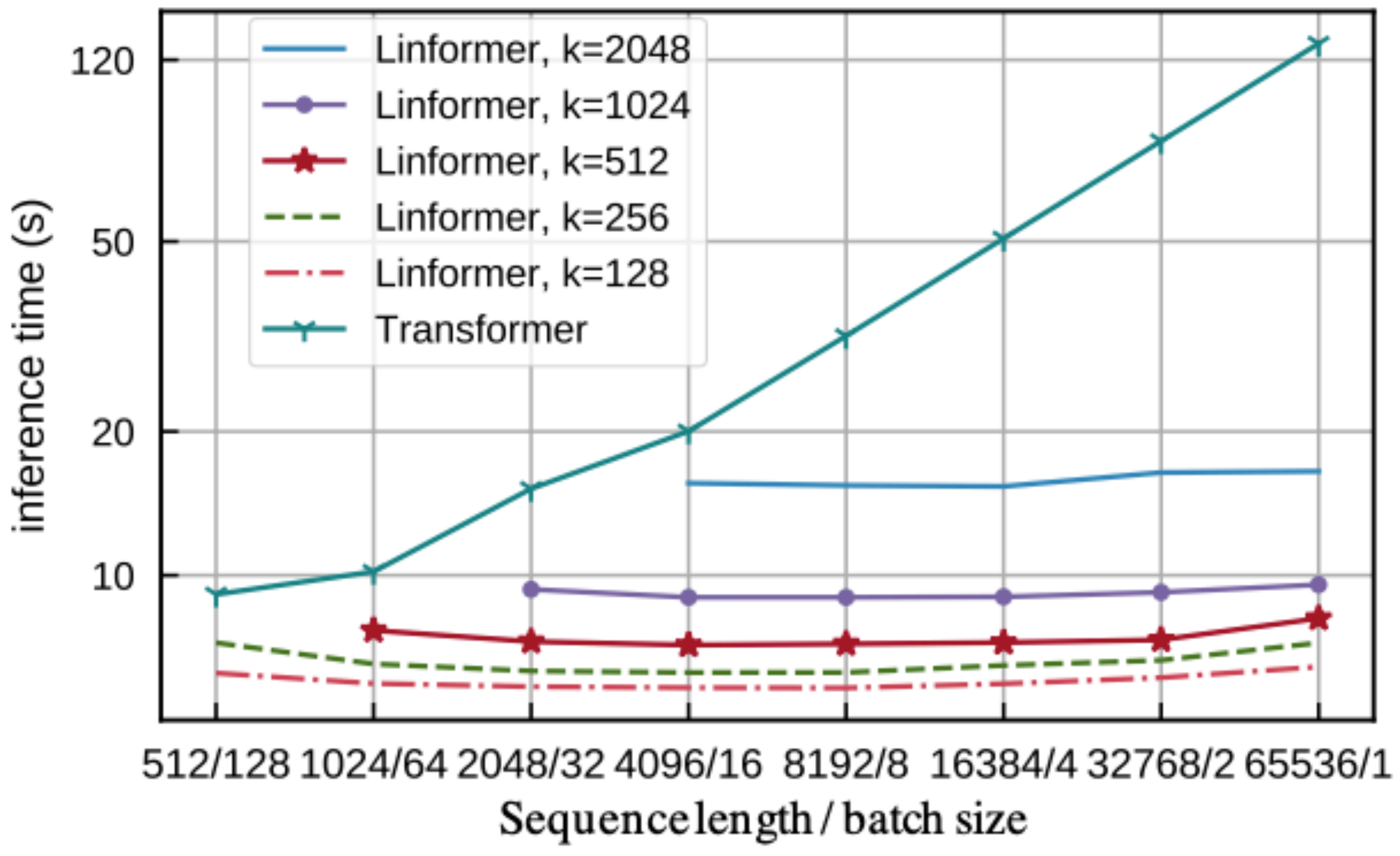
(Tay et al., 2020): Efficient Transformers: A Survey

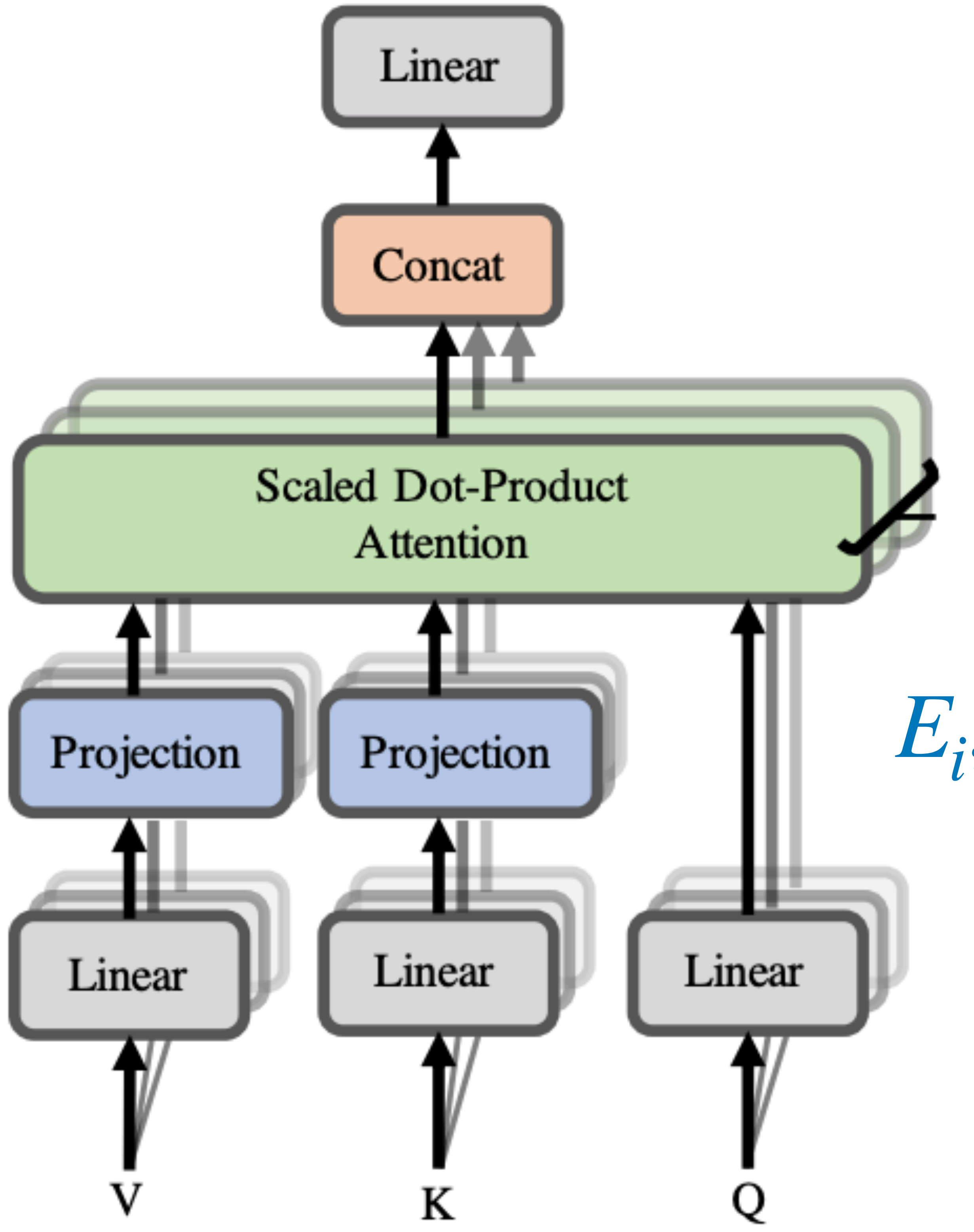
Example: Linformer

Key idea: The attention matrix $e_{i,j}$ can be approximated by a low-rank matrix

Map the sequence length dimension to a lower-dimensional space for values, keys

$$\begin{aligned}\overline{\text{head}_i} &= \text{Attention}(QW_i^Q, E_iKW_i^K, F_iVW_i^V) \\ &= \underbrace{\text{softmax}\left(\frac{QW_i^Q(E_iKW_i^K)^T}{\sqrt{d_k}}\right)}_{\bar{P}:n\times k} \cdot \underbrace{F_iVW_i^V}_{k\times d},\end{aligned}$$



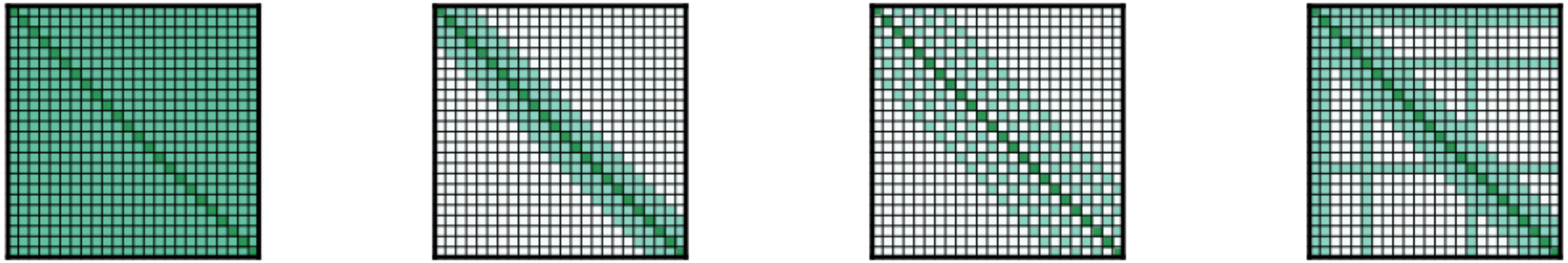


$E_i, F_i \in \mathbb{R}^{k\times n}$

(Wang et al., 2020): Linformer: Self-Attention with Linear Complexity

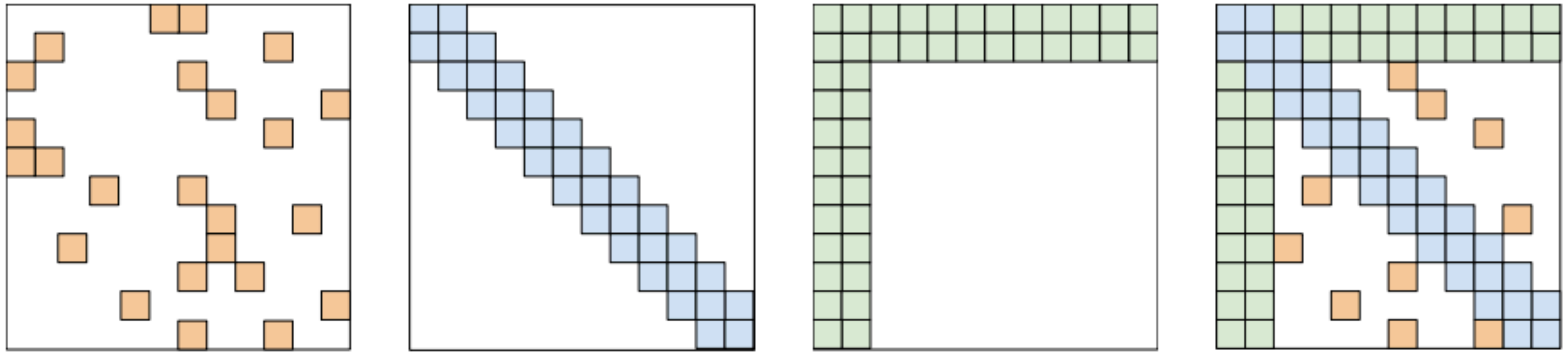
Example: Longformer / Big Bird

Key idea: use sparse attention patterns!



(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window

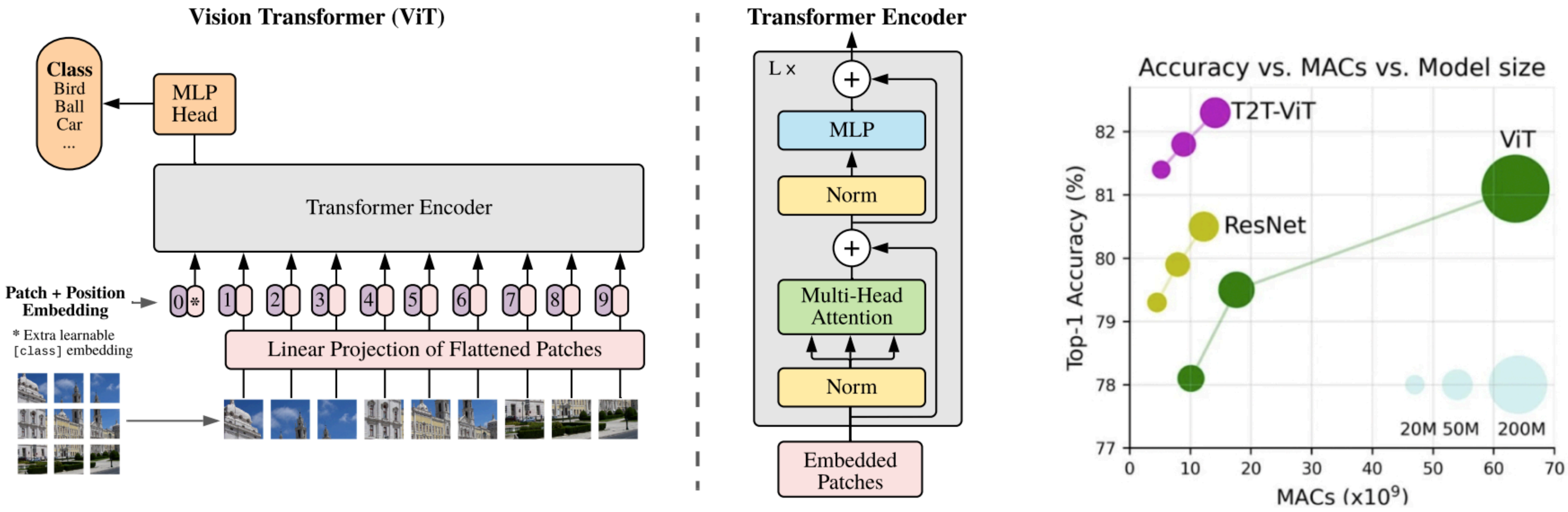
(Beltagy et al., 2020): Longformer: The Long-Document Transformer



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

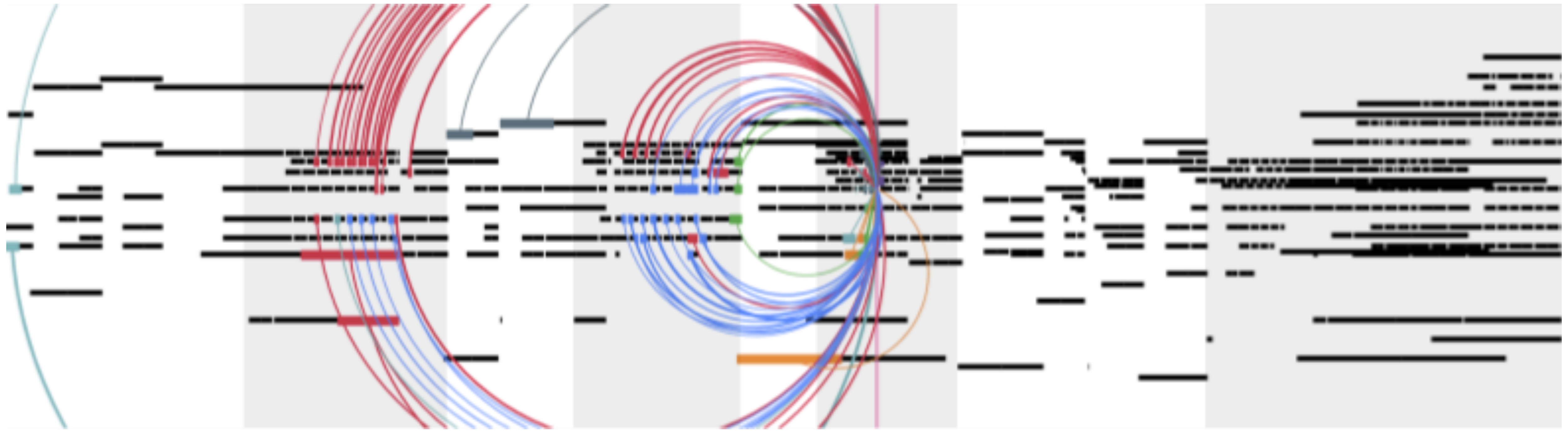
(Zaheer et al., 2021): Big Bird: Transformers for Longer Sequences

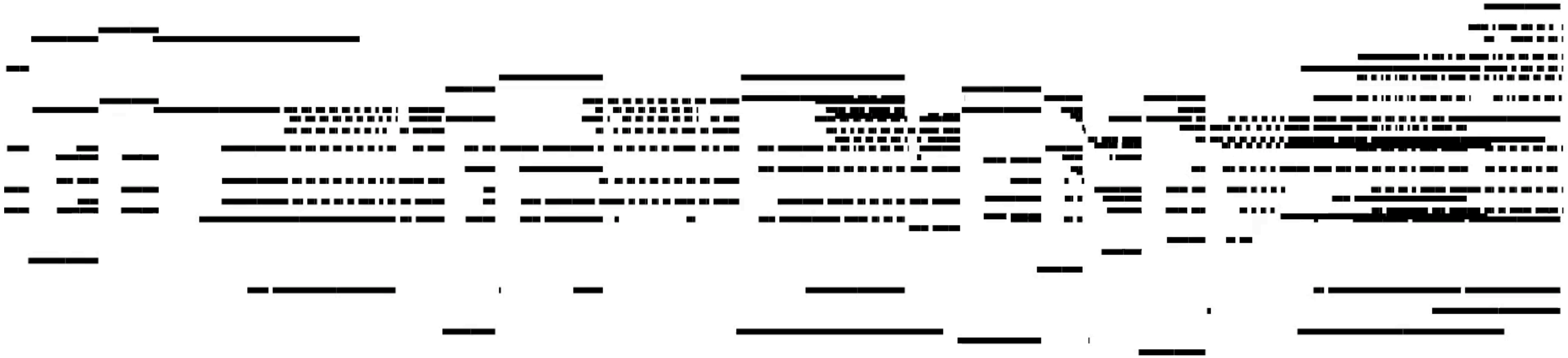
Vision Transformer (ViT)



(Dosovitskiy et al., 2021): An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

Music Transformer





https://magenta.tensorflow.org/music-transformer

(Huang et al., 2018): Music Transformer: Generating Music with Long-Term Structure