Machine Learning: Lecture #4

Jennifer Ngadiuba (Fermilab)

University of Pavia, May 8-12 2023

Overview of the lectures

- **Day 1:**
  - Introduction to Machine Learning fundamentals
  - Linear Models
- **Day 2:**
  - Neural Networks
  - Deep Neural Networks
  - Convolutional Neural Networks
- **Day 3:**
  - Recurrent Neural Networks
  - Graph Neural Networks (part 1)
- **Day 4:**
  - Graph Neural Networks (part 2)
  - Transformers
- **Day 5:**
  - Unsupervised learning
  - Autoencoders
  - Generative Adversarial Networks
  - Normalizing Flows

Hands on sessions each day will closely follow the lectures topics

Graph Attention Networks

Graph Attention Networks

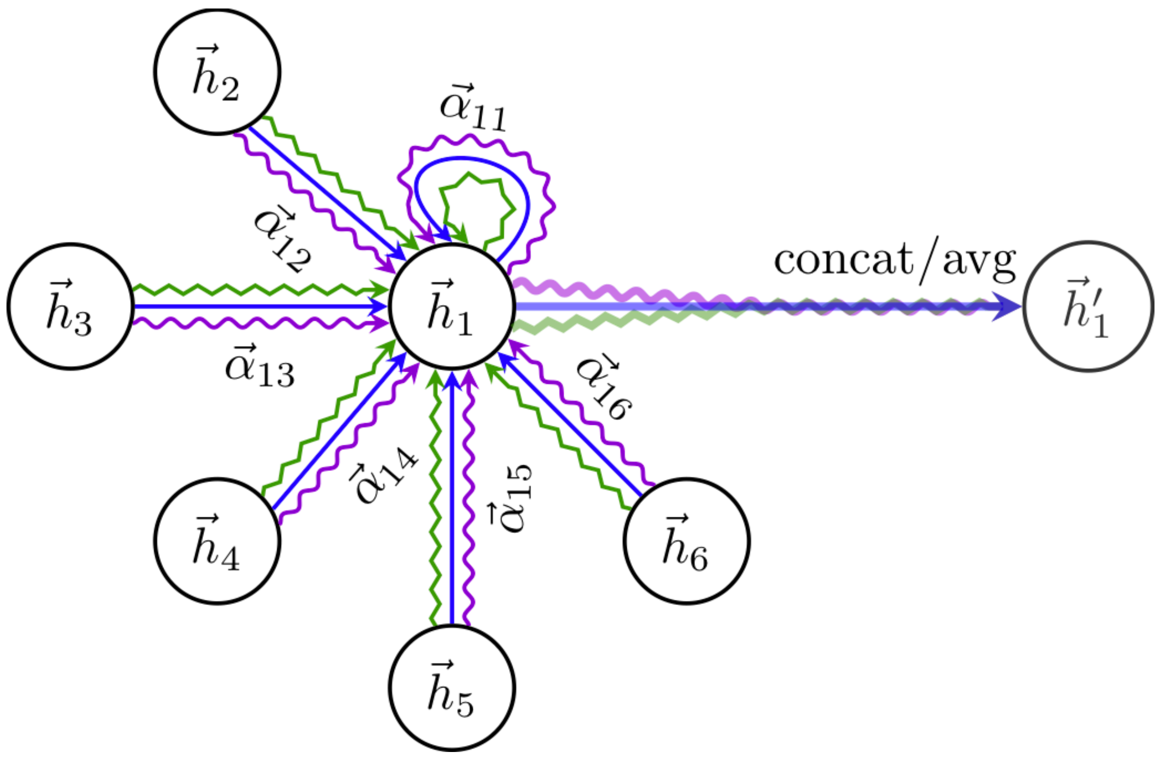
- The GNNs seen so far treat all the nodes in the same way
  - each neighbor contributes equally to update the representation of the central node
- GCN applies weights which however are fixed by the graph structure leading to worse generalization performance
  - DGCNN applies weights 0 and 1 with kNN which however it's not differentiable and so not optimizable
- An improved approach is to learn the weights during training → **attention mechanism**

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \quad (1)$$

$$e_{ij}^{(l)} = \text{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \quad (2)$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik}^{(l)})}, \quad (3)$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \quad (4)$$



P. Veličković et al., ICLR '18

Let's break it down

this is the source node *i* embedding

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

$$e_{ij}^{(l)} = \text{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \tag{4}$$

Let's break it down

this is the source node i embedding

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

$$e_{ij}^{(l)} = \text{LeakyReLU}(\vec{a}^{(l)^T} (z_i^{(l)} || z_j^{(l)})), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$h_i^{(l+1)} = \sigma \left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)} \right), \tag{4}$$

The usual message passing formula for iteration (or layer) l but with learnable weights a_{ij}

z_j are the embeddings of the neighbours nodes

Let's break it down

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

$$e_{ij}^{(l)} = \mathrm{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \tag{4}$$

this is the source node i embedding

here we learn the weights α_{ij} from current node i embedding and neighbour node j embedding (|| is concatenation symbol)

The usual message passing formula for iteration (or layer) l but with learnable weights a_{ij}

z_j are the embeddings of the neighbours nodes

Let's break it down

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

$$e_{ij}^{(l)} = \text{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \tag{4}$$

this is the source node *i* embedding

here we learn the weights α_{ij} from current node *i* embedding and neighbour node *j* embedding (|| is concatenation symbol)

this is the softmax to get normalized values and obtain a probability

The usual message passing formula for iteration (or layer) *l* but with learnable weights α_{ij}

z_j are the embeddings of the neighbours nodes

Let's break it down

$$
\begin{aligned}
z_i^{(l)} &= W^{(l)} h_i^{(l)}, & (1)\\
e_{ij}^{(l)} &= \mathrm{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), & (2)\\
\alpha_{ij}^{(l)} &= \frac{\exp(e_{ij}^{(l)})}{\sum_{k\in\mathcal{N}(i)} \exp(e_{ik}^{(l)})}, & (3)\\
h_i^{(l+1)} &= \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), & (4)
\end{aligned}
$$

this is the source node i embedding

here we learn the weights α_{ij} from current node i embedding and neighbour node j embedding (|| is concatenation symbol)

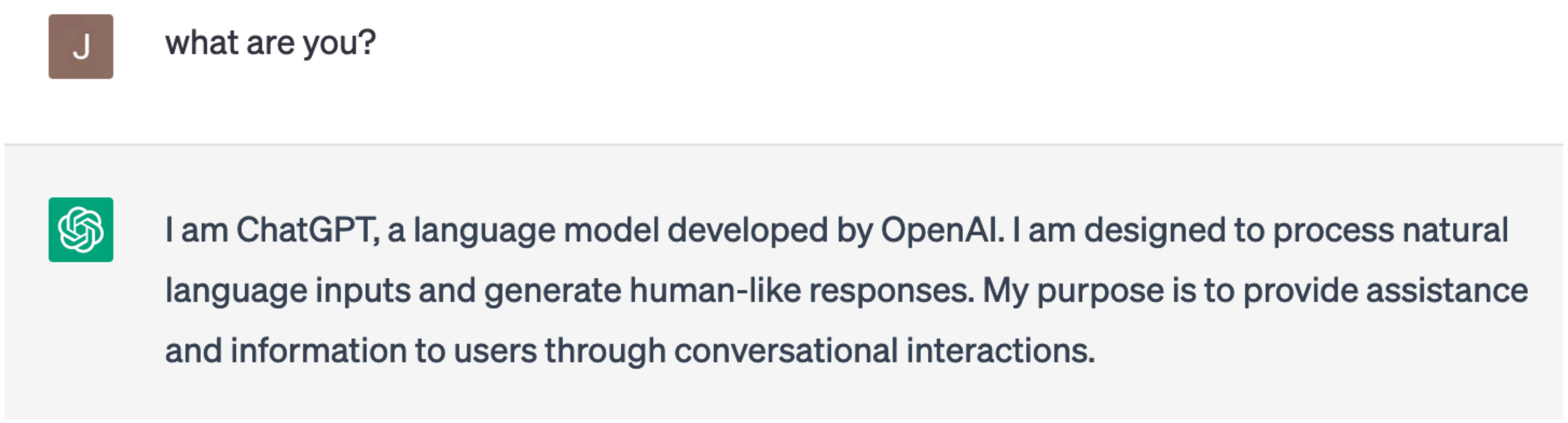
this is the softmax to get normalized values and obtain a probability

The usual message passing formula for iteration (or layer) l but with learnable weights a_{ij}

z_j are the embeddings of the neighbours nodes

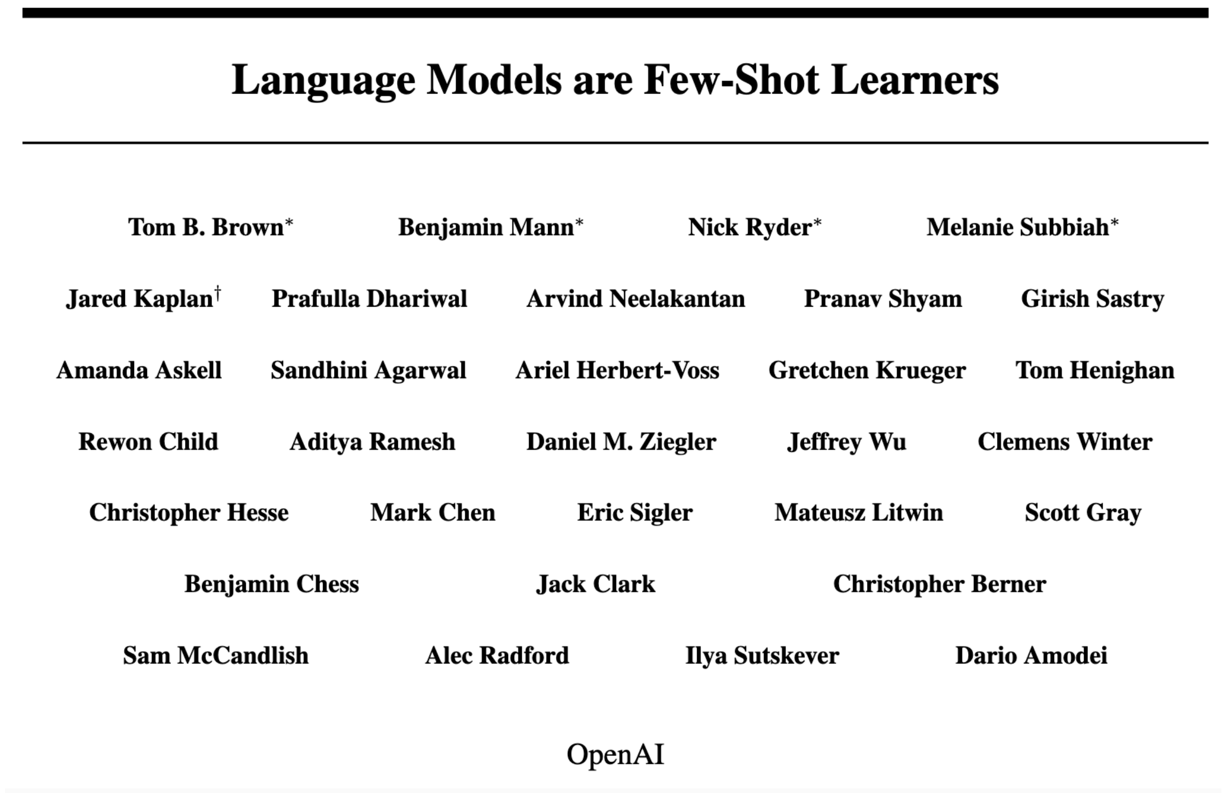
What if there is not a unique function to learn attention weights → multi-head attention!

Transformers



J what are you?

I am ChatGPT, a language model developed by OpenAI. I am designed to process natural language inputs and generate human-like responses. My purpose is to provide assistance and information to users through conversational interactions.

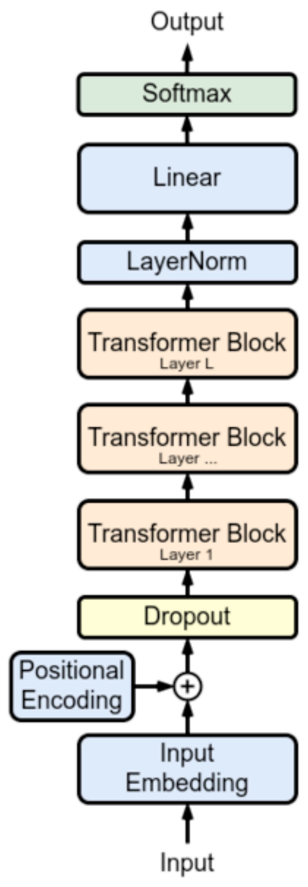


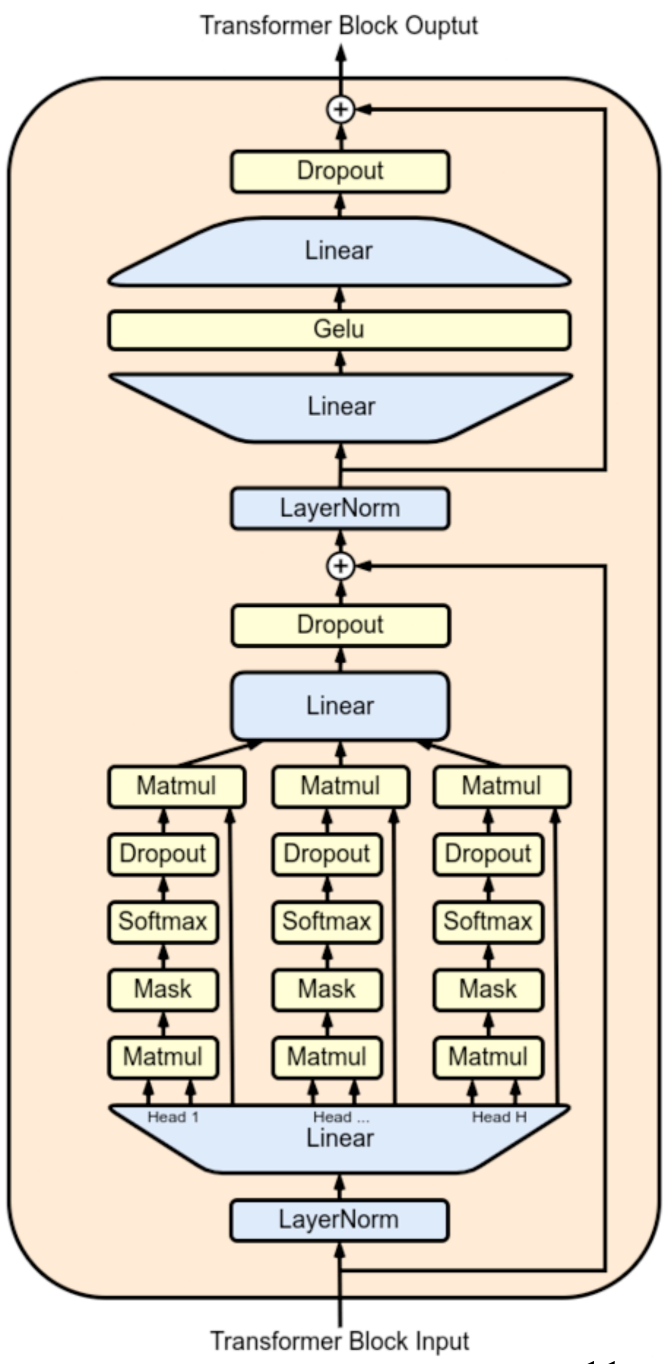
Language Models are Few-Shot Learners

Tom B. Brown* Benjamin Mann* Nick Ryder* Melanie Subbiah*
Jared Kaplan† Prafulla Dhariwal Arvind Neelakantan Pranav Shyam Girish Sastry
Amanda Askell Sandhini Agarwal Ariel Herbert-Voss Gretchen Krueger Tom Henighan
Rewon Child Aditya Ramesh Daniel M. Ziegler Jeffrey Wu Clemens Winter
Christopher Hesse Mark Chen Eric Sigler Mateusz Litwin Scott Gray
Benjamin Chess Jack Clark Christopher Berner
Sam McCandlish Alec Radford Ilya Sutskever Dario Amodei

OpenAI

https://arxiv.org/abs/2005.14165





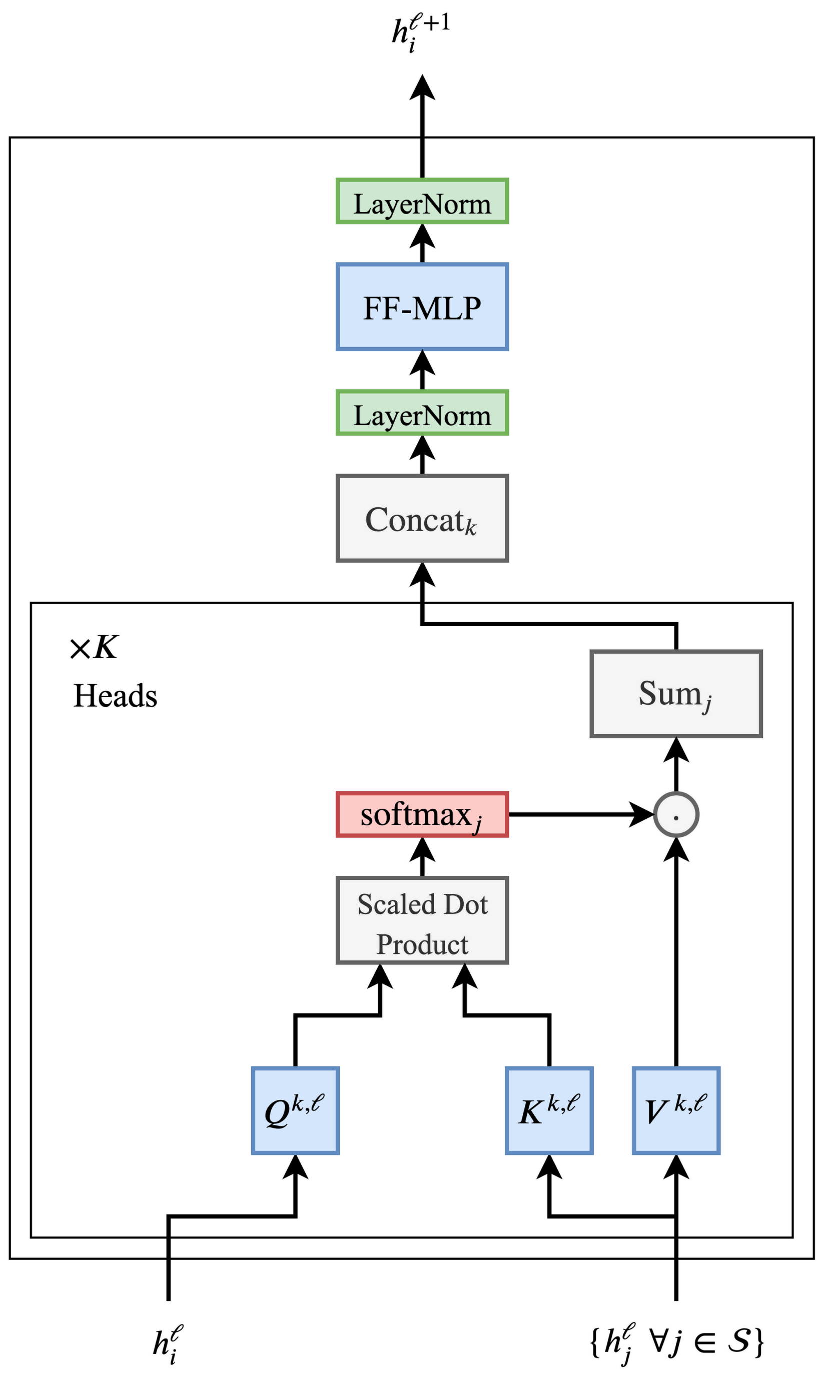
Transformers

- Based on the same concept of attention but multiple (different) functions K are used to learn the attention weights → **multi-head attention**
    - allow the attention function to extract information from different representation subspaces (like filters in CNNs)
- It was a breakthrough in Natural Processing Language where attention is needed to different parts of the text
- A dedicated formalism is used where weights are separately called **Query (Q), Key (K), and Value (V)**

$$m_i^{(k)} = \text{Concat}\Big(\sum_j h_j^{(k)} V^{(l)} a_{ij}^{(k,l)}\Big)$$

$$a_{ij}^{(k,l)} = \text{softmax}(Q^{(k,l)} h_i^{(k)} \cdot K^{(k,l)} h_j^{(k)})$$

where $a_{ij}^{(k,l)}$ is the lth attention in the kth layer (or iteration)



Scaled Dot-Product Attention

- The Transformer implements a **scaled dot-product attention**, which follows the procedure of the general attention mechanism of previous slides

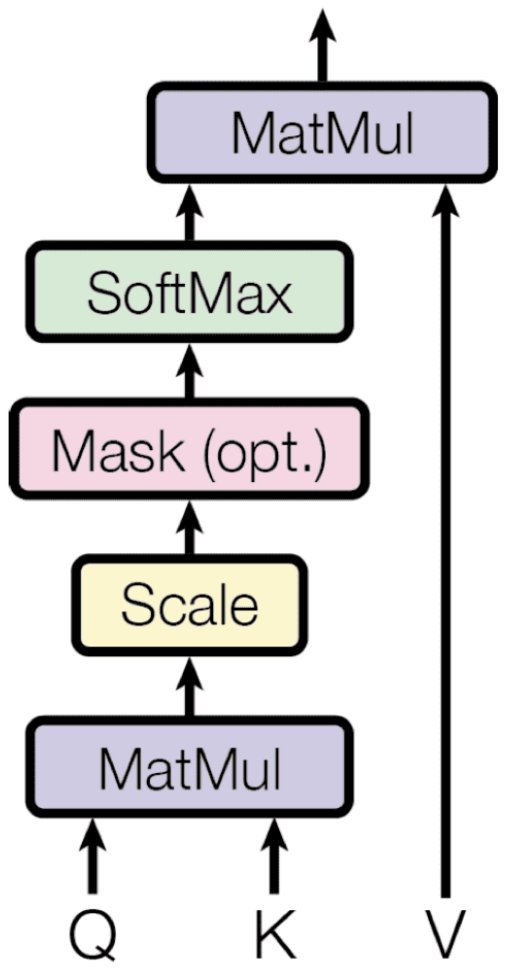
- The scaled dot-product attention first computes a *dot product* for each query, Q, with all of the keys, K.

- It subsequently divides each result by $\sqrt{d_k}$ and proceeds to apply a softmax function

- In doing so, it obtains the weights that are used to *scale* the values, V.

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

where d_k is the dimension of the keys



Scaled Dot-Product Attention

- The step-by-step procedure for computing the scaled-dot product attention is the following:

1. multiplying the set of queries in the matrix $\mathbf{Q}$ with the keys in the matrix $\mathbf{K}$. If the matrix $\mathbf{Q}$ is of size $m \times d_k$ and the matrix $\mathbf{K}$ is of size $n \times d_k$ then the resulting matrix is of size $m \times n$

$$\mathbf{QK}^{\mathrm{T}} = \begin{bmatrix} e_{11} & e_{12} & \dots & e_{1n} \\ e_{21} & e_{22} & \dots & e_{2n} \\ \vdots & \vdots & \ddots & \dots \\ e_{m1} & e_{m2} & \dots & e_{mn} \end{bmatrix}$$

2. Scale each of the alignment scores:

$$\frac{\mathbf{QK}^{\mathrm{T}}}{\sqrt{d_k}} = \begin{bmatrix} \frac{e_{11}}{\sqrt{d_k}} & \frac{e_{12}}{\sqrt{d_k}} & \dots & \frac{e_{1n}}{\sqrt{d_k}} \\ \frac{e_{21}}{\sqrt{d_k}} & \frac{e_{22}}{\sqrt{d_k}} & \dots & \frac{e_{2n}}{\sqrt{d_k}} \\ \vdots & \vdots & \ddots & \dots \\ \frac{e_{m1}}{\sqrt{d_k}} & \frac{e_{m2}}{\sqrt{d_k}} & \dots & \frac{e_{mn}}{\sqrt{d_k}} \end{bmatrix}$$

3. Apply a softmax operation σ in order to obtain a set of weights:

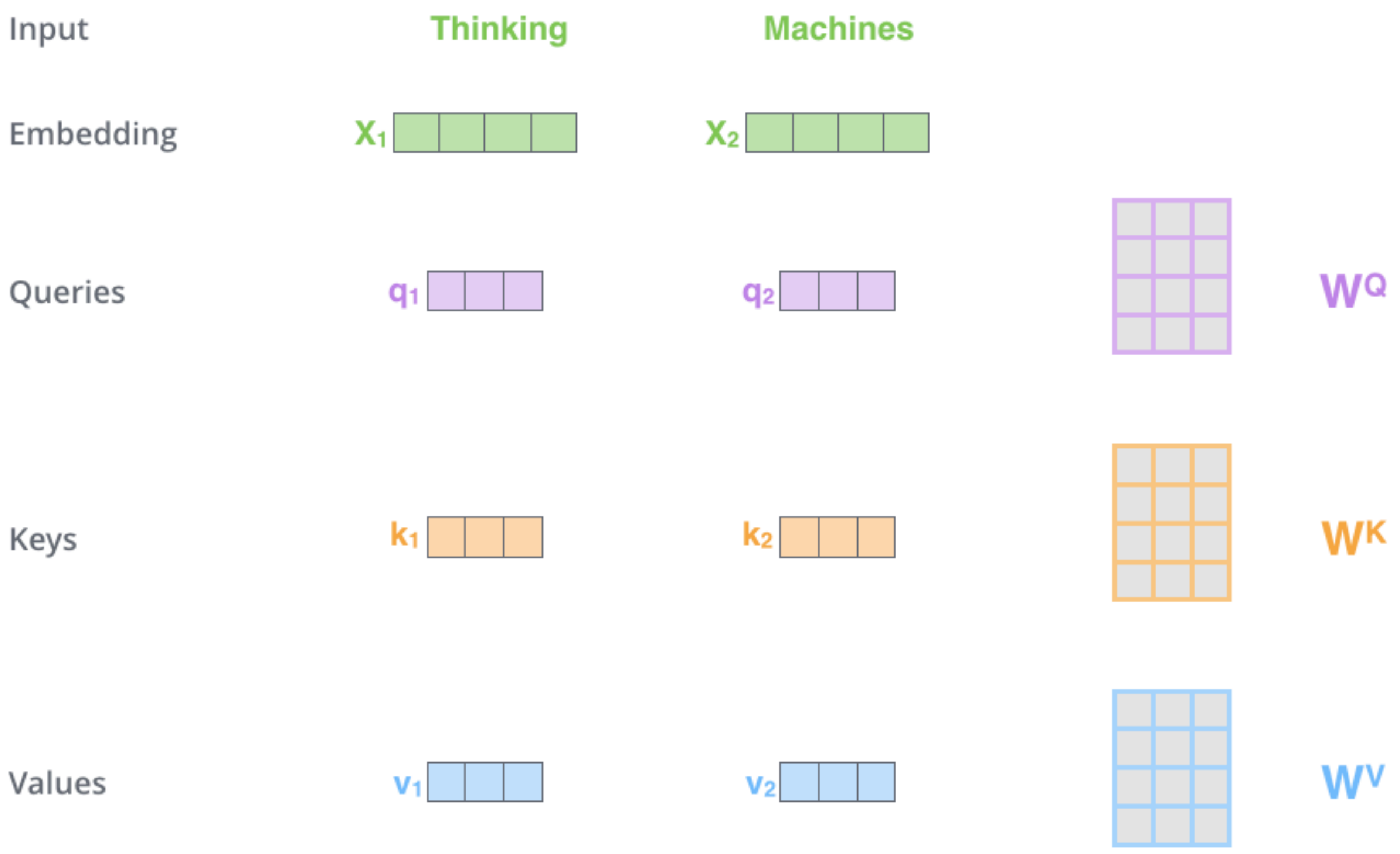
$$\sigma\left(\frac{\mathbf{QK}^{\mathrm{T}}}{\sqrt{d_k}}\right) = \begin{bmatrix} \sigma\left(\frac{e_{11}}{\sqrt{d_k}}\right) & \sigma\left(\frac{e_{12}}{\sqrt{d_k}}\right) & \dots & \sigma\left(\frac{e_{1n}}{\sqrt{d_k}}\right) \\ \sigma\left(\frac{e_{21}}{\sqrt{d_k}}\right) & \sigma\left(\frac{e_{22}}{\sqrt{d_k}}\right) & \dots & \sigma\left(\frac{e_{2n}}{\sqrt{d_k}}\right) \\ \vdots & \vdots & \ddots & \dots \\ \sigma\left(\frac{e_{m1}}{\sqrt{d_k}}\right) & \sigma\left(\frac{e_{m2}}{\sqrt{d_k}}\right) & \dots & \sigma\left(\frac{e_{mn}}{\sqrt{d_k}}\right) \end{bmatrix}$$

4. Finally, apply the resulting weights to the values in the matrix $\mathbf{V}$ of size $n \times d_v$

$$\sigma\left(\frac{\mathbf{QK}^{\mathrm{T}}}{\sqrt{d_k}}\right)\mathbf{V} = \begin{bmatrix} \sigma\left(\frac{e_{11}}{\sqrt{d_k}}\right) & \sigma\left(\frac{e_{12}}{\sqrt{d_k}}\right) & \dots & \sigma\left(\frac{e_{1n}}{\sqrt{d_k}}\right) \\ \sigma\left(\frac{e_{21}}{\sqrt{d_k}}\right) & \sigma\left(\frac{e_{22}}{\sqrt{d_k}}\right) & \dots & \sigma\left(\frac{e_{2n}}{\sqrt{d_k}}\right) \\ \vdots & \vdots & \ddots & \dots \\ \sigma\left(\frac{e_{m1}}{\sqrt{d_k}}\right) & \sigma\left(\frac{e_{m2}}{\sqrt{d_k}}\right) & \dots & \sigma\left(\frac{e_{mn}}{\sqrt{d_k}}\right) \end{bmatrix} \begin{bmatrix} v_{11} & v_{12} & \dots & v_{1d_v} \\ v_{21} & v_{22} & \dots & v_{2d_v} \\ \vdots & \vdots & \ddots & \dots \\ v_{n1} & v_{1n} & \dots & v_{nd_v} \end{bmatrix}$$

Self-attention visualized

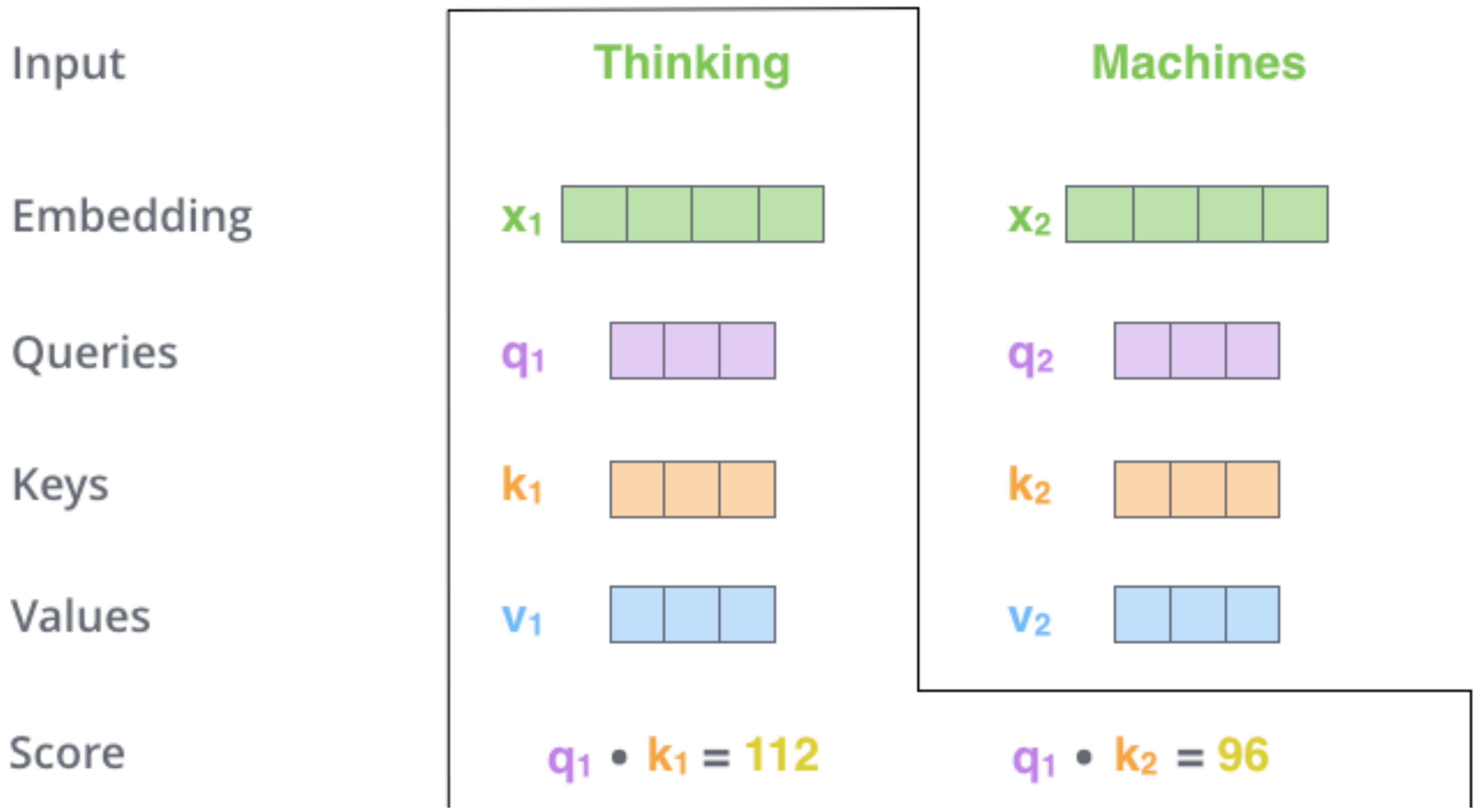
- **Step 1:** create three vectors from each of the encoder's input vectors (e.g., the embedding of each word)
  - for each word we create the Query, the Key, and the Value vectors by multiplying the embedding by three matrices that we trained during the training process



[source]

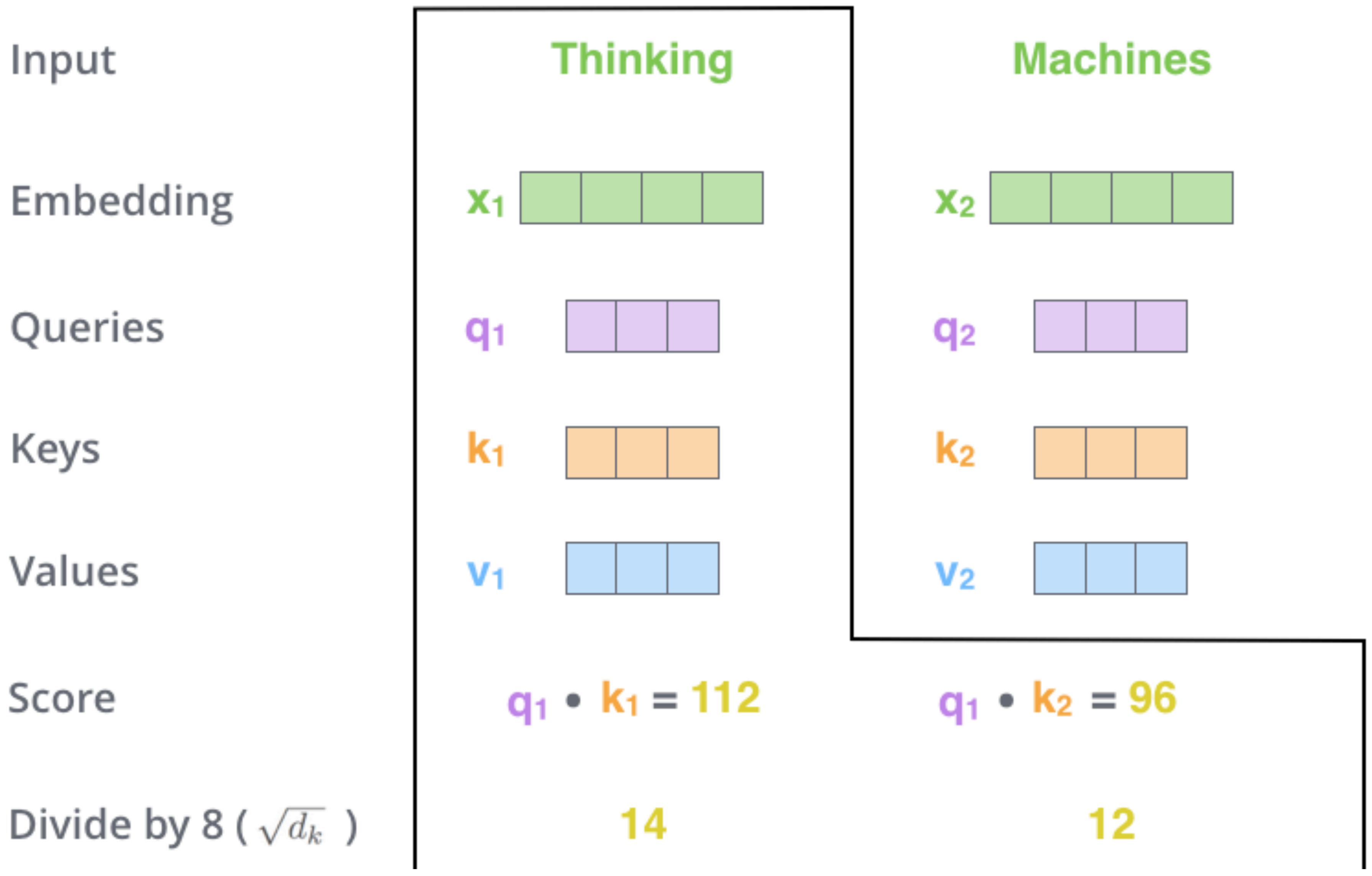
Self-attention visualized

- **Step 2:** calculate a score, i.e. how much focus to place on other parts of the input sentence as we encode a word at a certain position
    - score = dot product of the query vector with the key vector of the respective word we're scoring



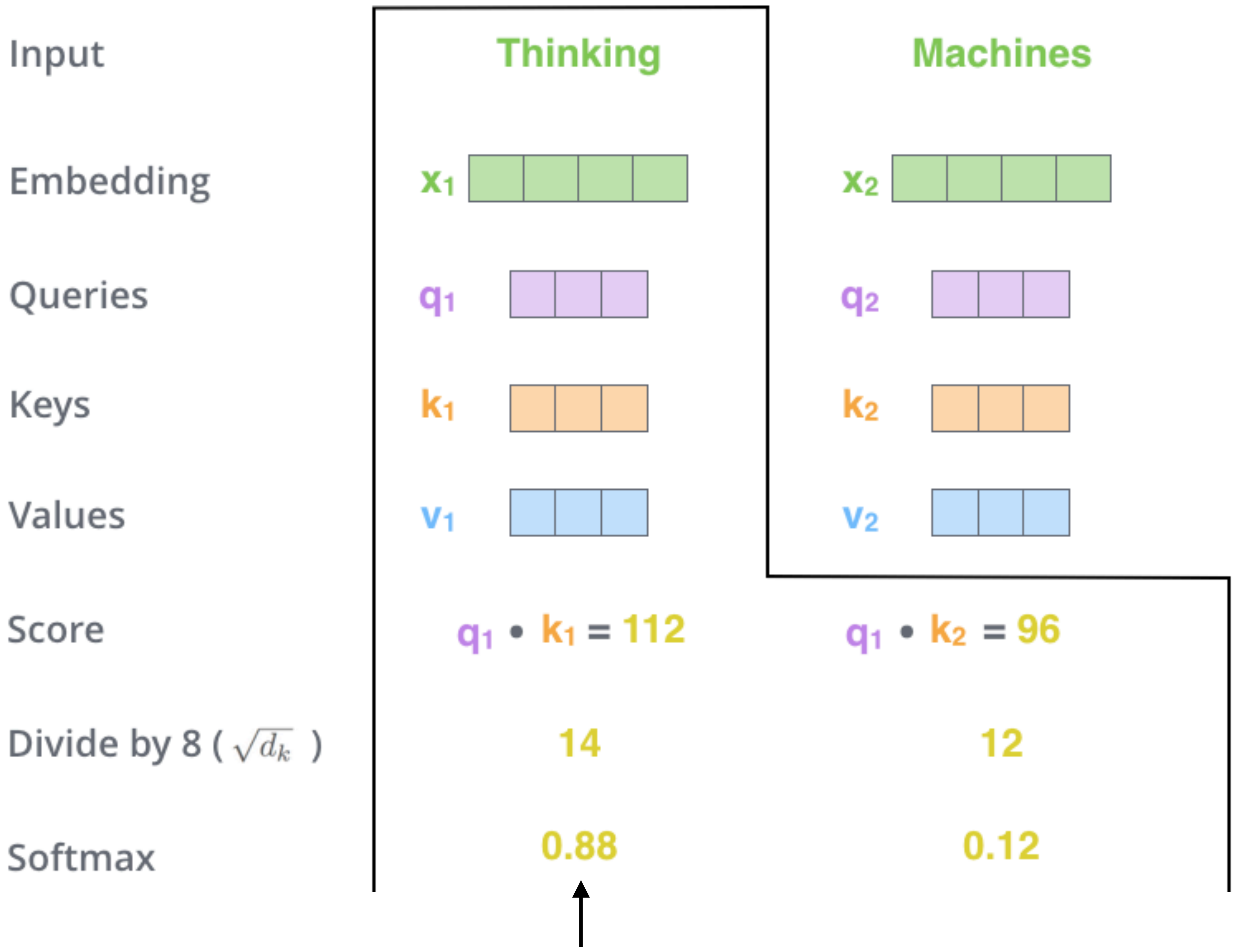
Self-attention visualized

- **Step 3:** divide the score by the square root of the dimension of the key vector



Self-attention visualized

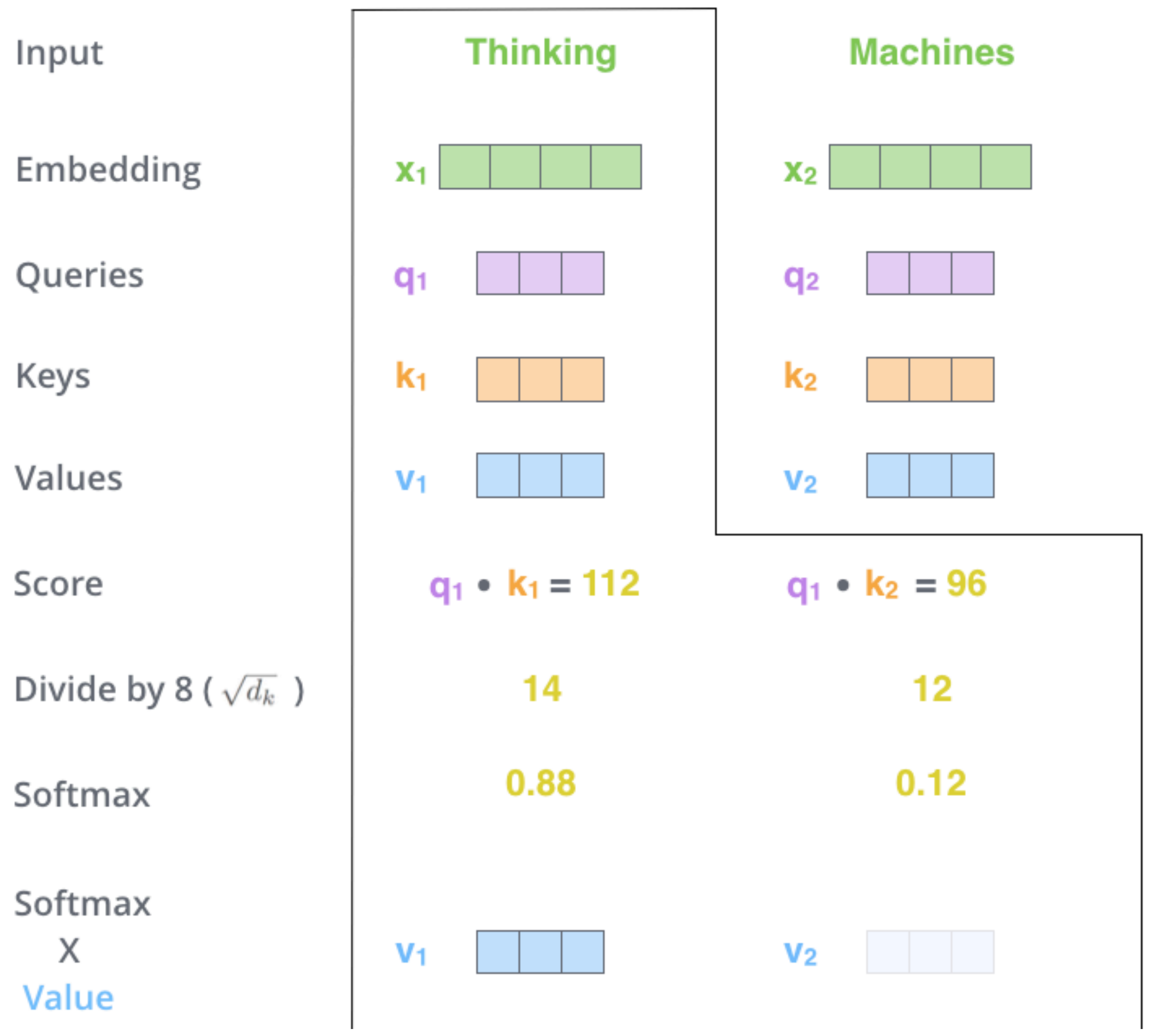
- **Step 4:** normalize the score with the Sofmax so they're all positive and add up to 1



highest score for same word

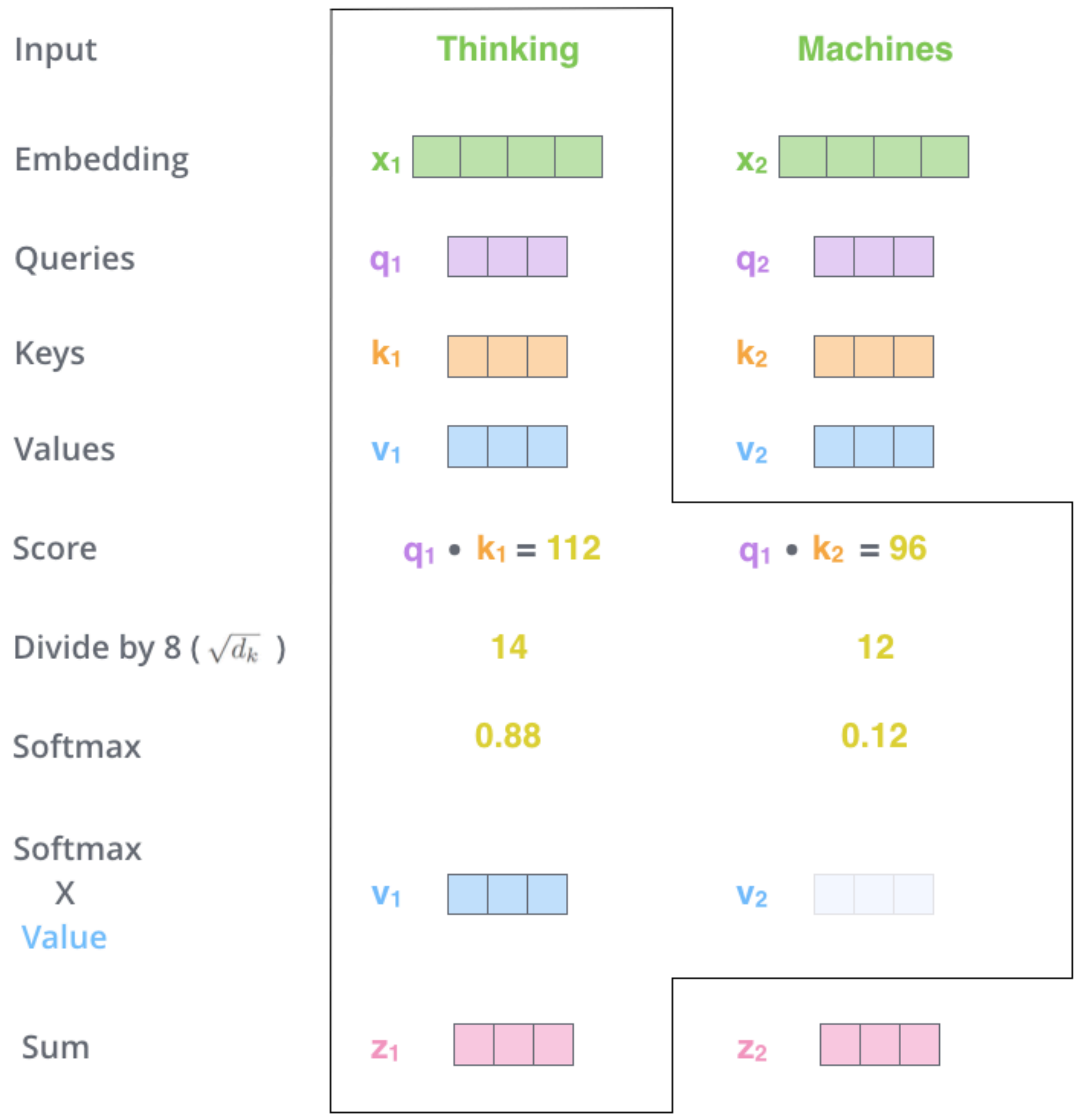
Self-attention visualized

- **Step 5:** multiply each value vector by the softmax score — keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words by multiplying by small softmax score



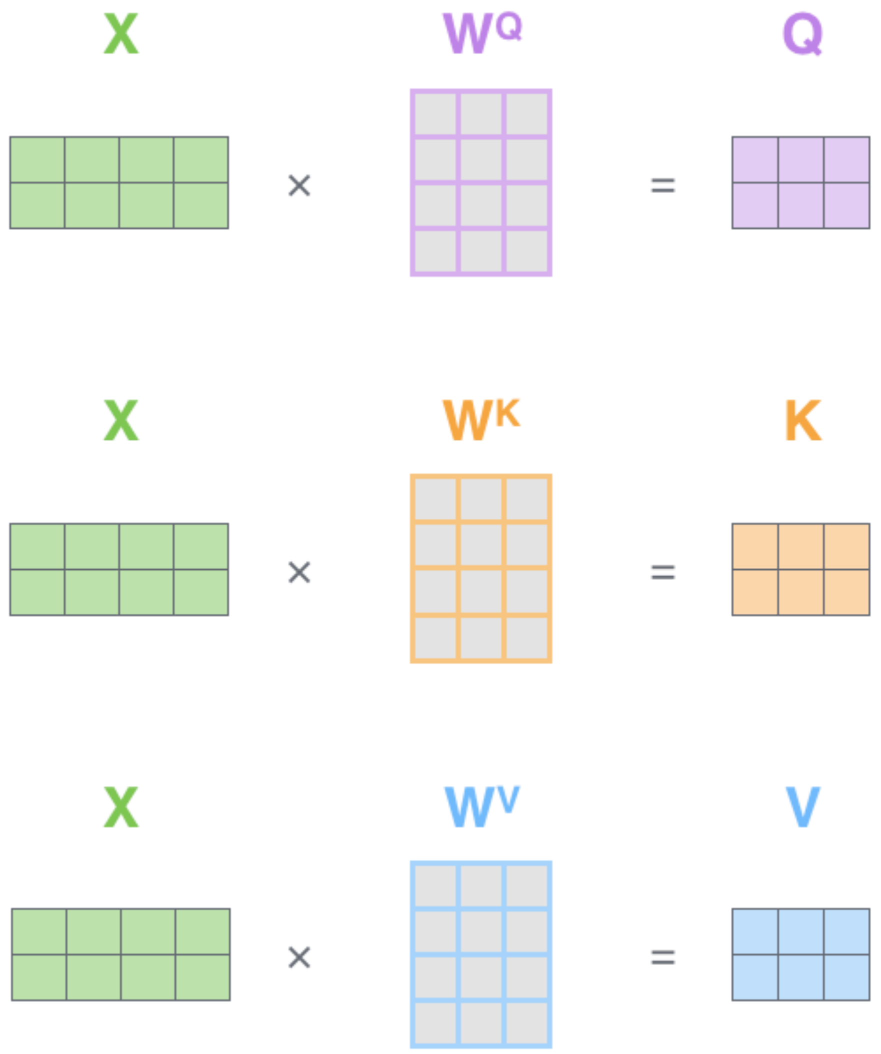
Self-attention visualized

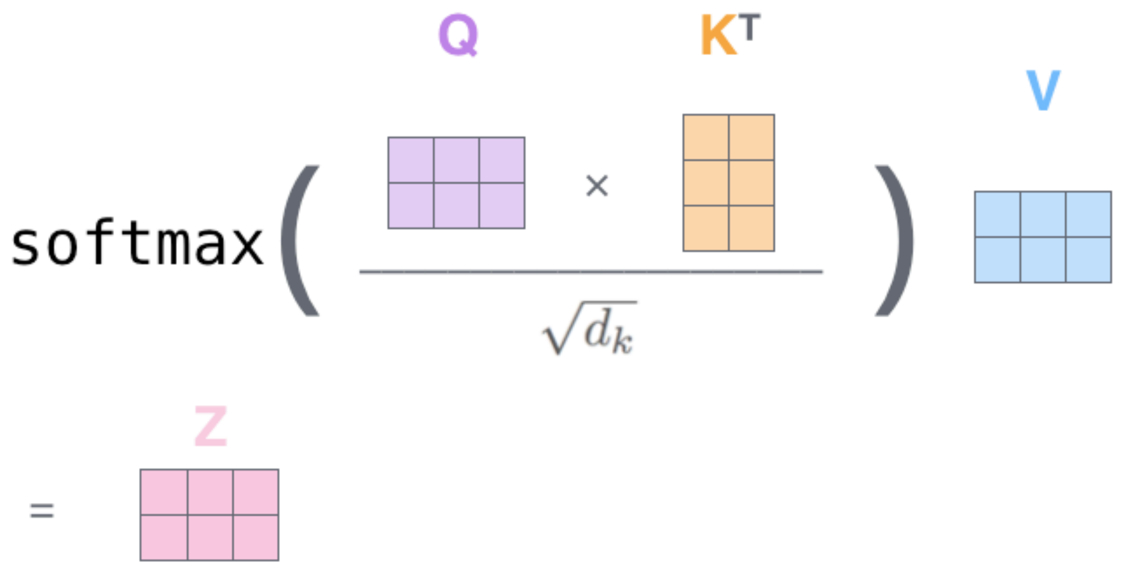
- **Step 6:** sum up the weighted value vectors. This produces the output of the self-attention layer for the first word



Self-attention: matrix version

- Calculate the Query, Key, and Value matrices by packing the embeddings into a matrix X, and multiplying it by the weight matrices we trained (WQ, WK, WV)
- And finally we can condense steps two through six in one formula to calculate the outputs of the self-attention layer





The Multi-Head Attention

- Building on their single attention function that takes matrices, Q, K , and V, as input, Vaswani et al. also propose a **multi-head attention mechanism**

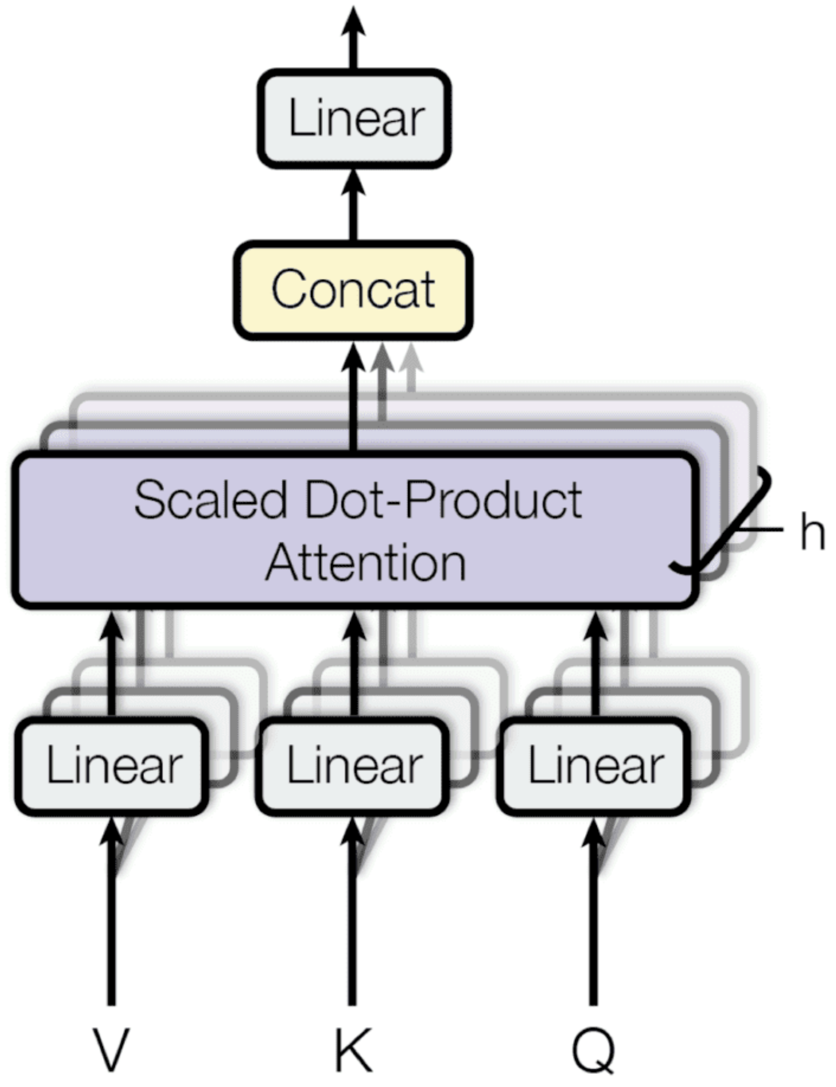
- Their multi-head attention mechanism linearly projects the queries, keys, and values h times, using a different learned projection each time

- The single attention mechanism is then applied to each of these h projections in parallel to produce h outputs, which, in turn, are concatenated and projected again to produce a final result

$$\text{multihead}(Q, K, V) = \text{concat}(\text{head}_1, \ldots, \text{head}_h)W^o$$

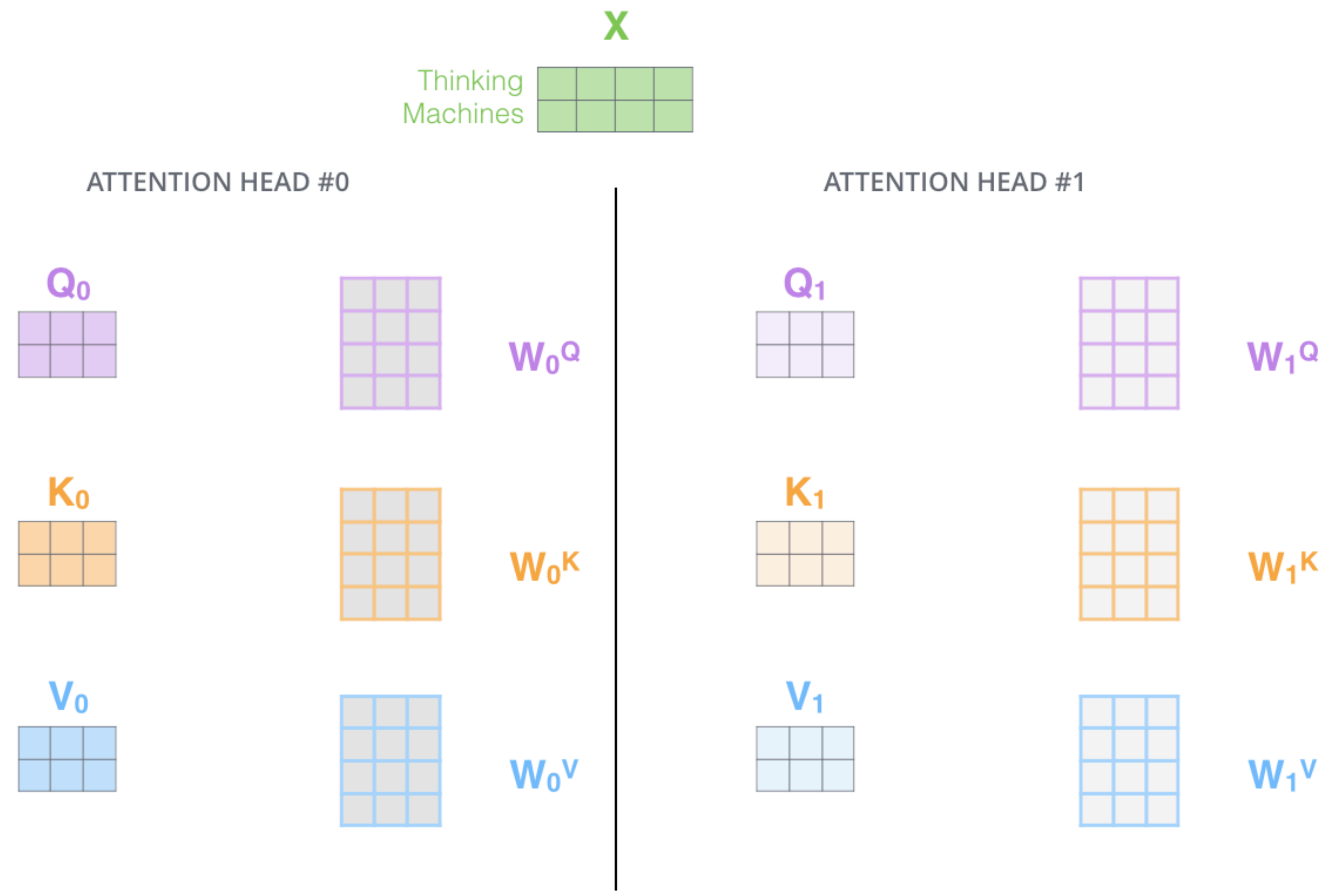
- Each *head* implements a single attention function characterized by its own learned weight matrices

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



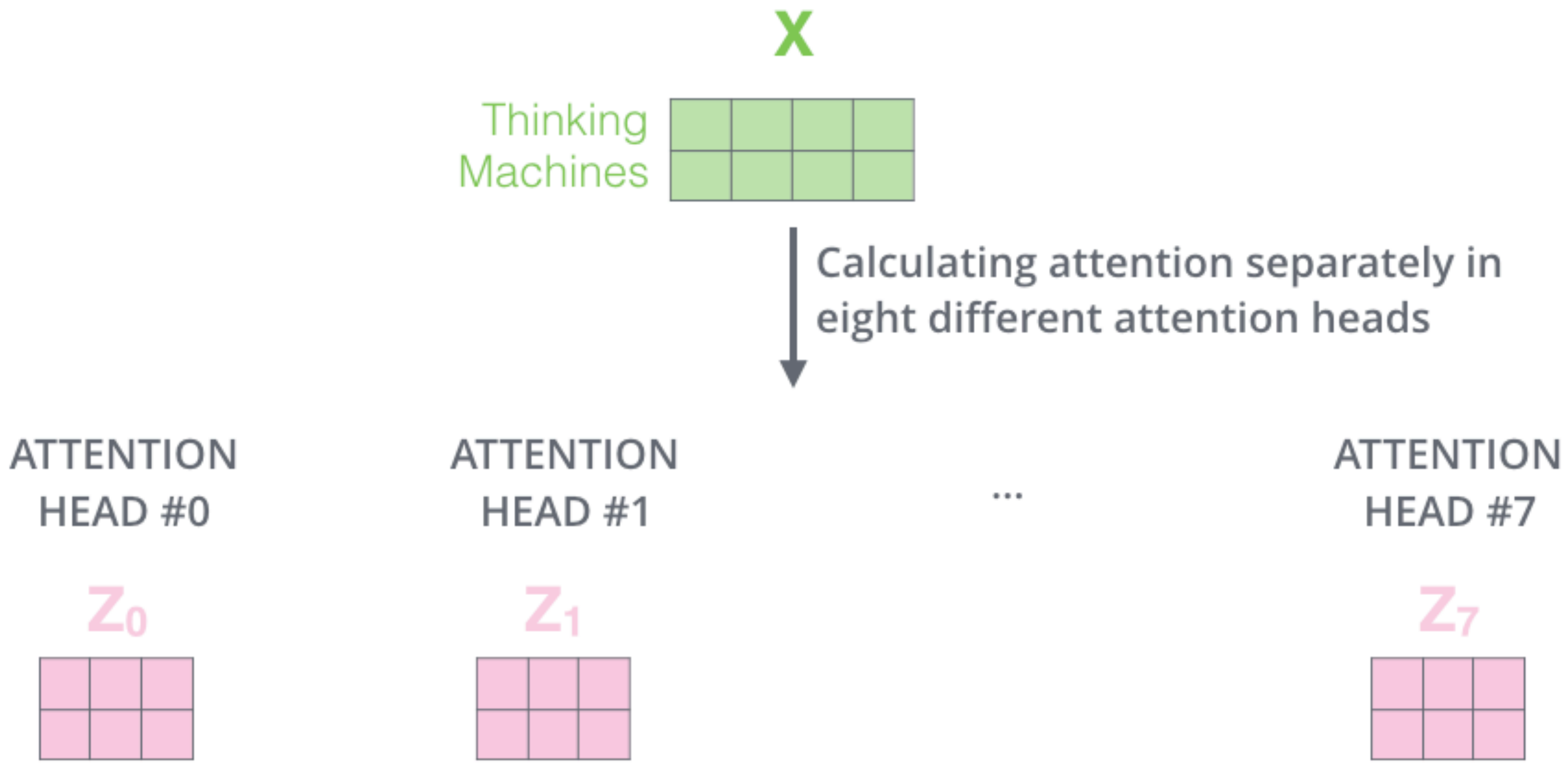
MHA visualized

- **Multiple sets of Query/Key/Value weight matrices** (one per head) — each of these is randomly initialized and after training used to project the input embeddings into a different representation subspace



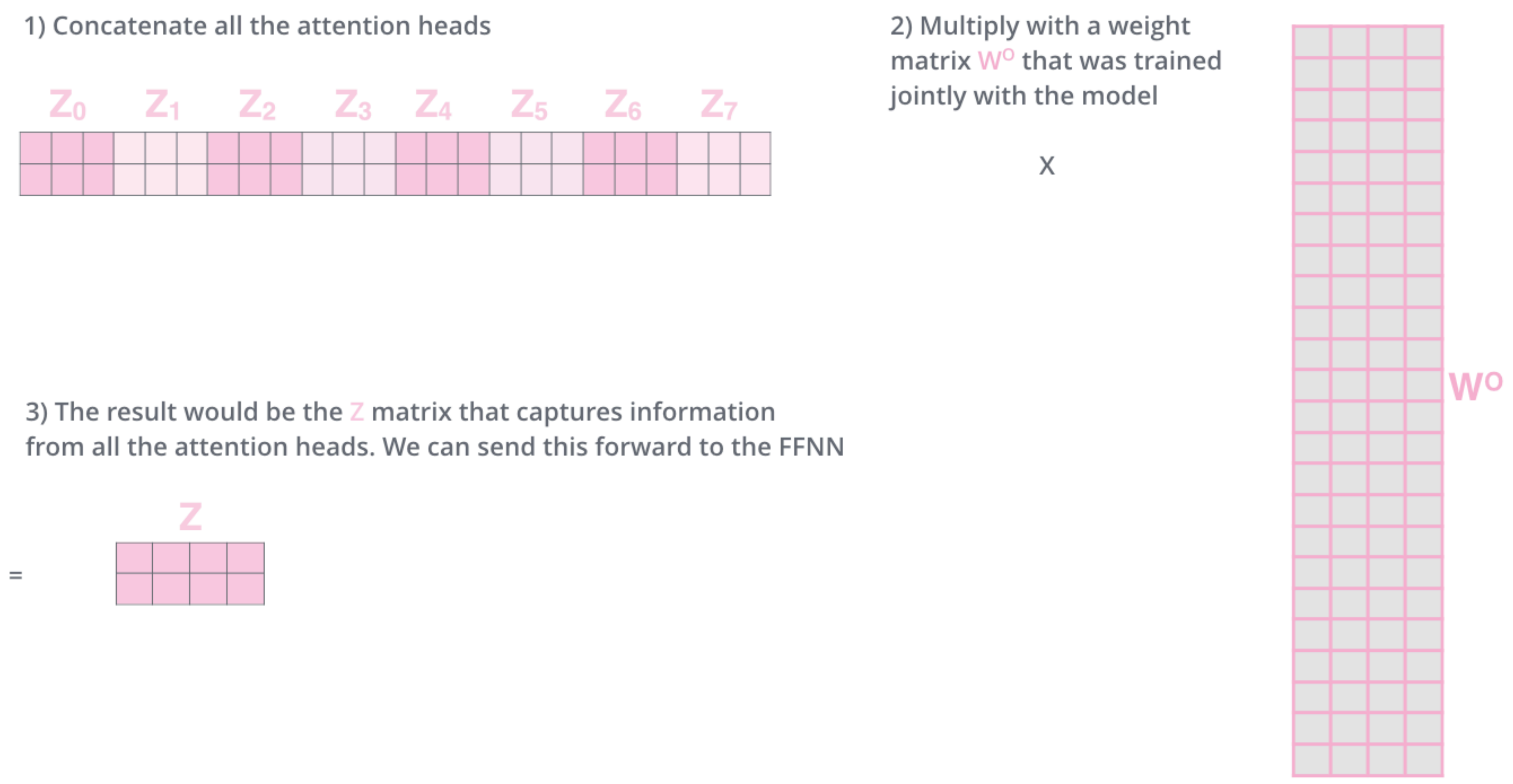
MHA visualized

- If we do the same self-attention calculation explained in previous slides just 8 (= number of heads) different times with different weight matrices, we end up with 8 different Z matrices

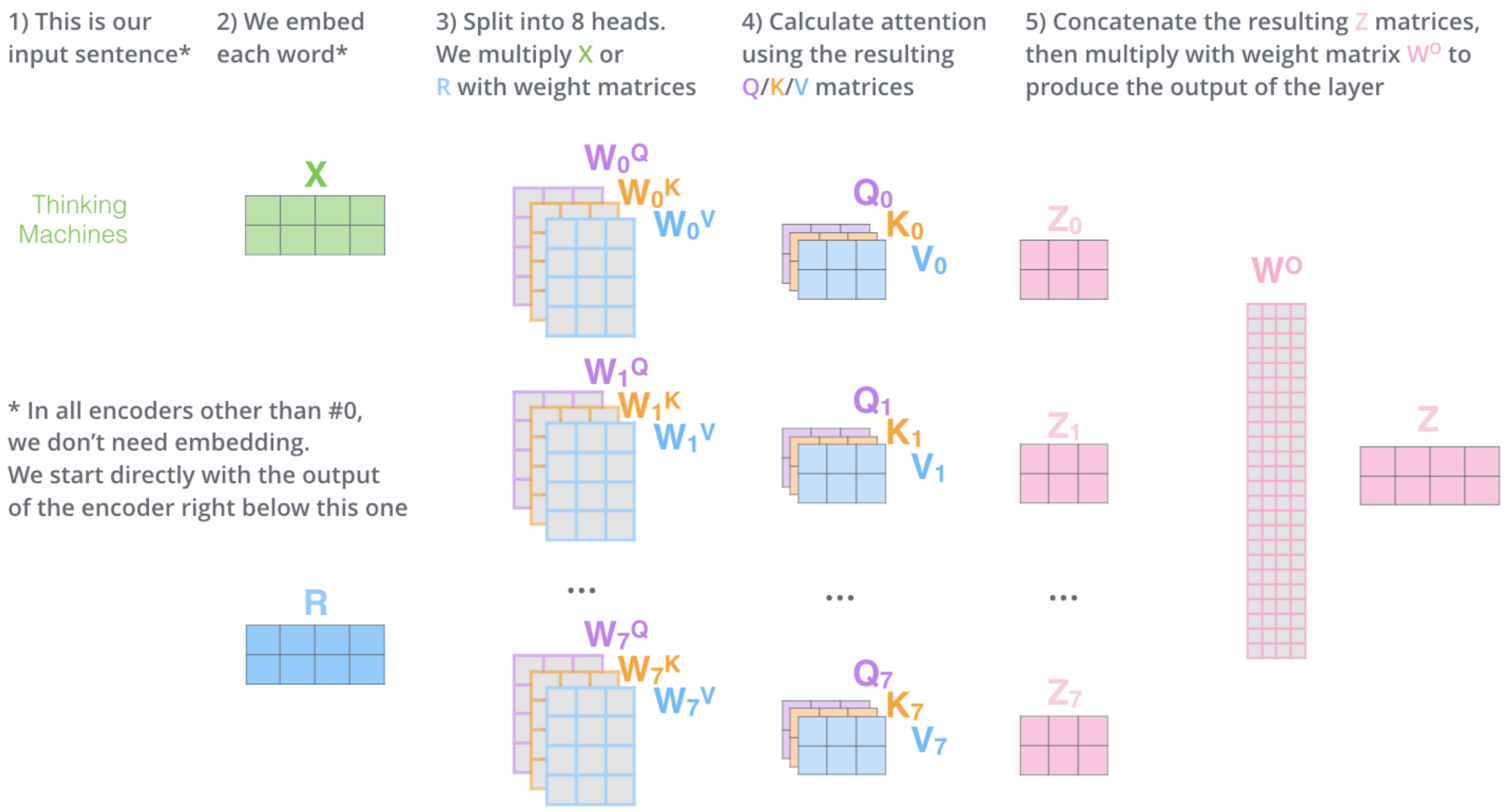


MHA visualized

- **Problem:** the downstream MLP layers do not expect N matrices but rather a single matrix (a vector for each word)
- **Fix:** concatenate the matrices then multiply them by an additional weights matrix WO


1) Concatenate all the attention heads
Z_0 Z_1 Z_2 Z_3 Z_4 Z_5 Z_6 Z_7
2) Multiply with a weight matrix W^O that was trained jointly with the model
X
W^O
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN
=
Z

Summary



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
X
R
W_0^Q W_0^K W_0^V
W_1^Q W_1^K W_1^V
W_7^Q W_7^K W_7^V
Q_0 K_0 V_0
Q_1 K_1 V_1
Q_7 K_7 V_7
Z_0
Z_1
Z_7
...
W^O
Z
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

QKV intuition

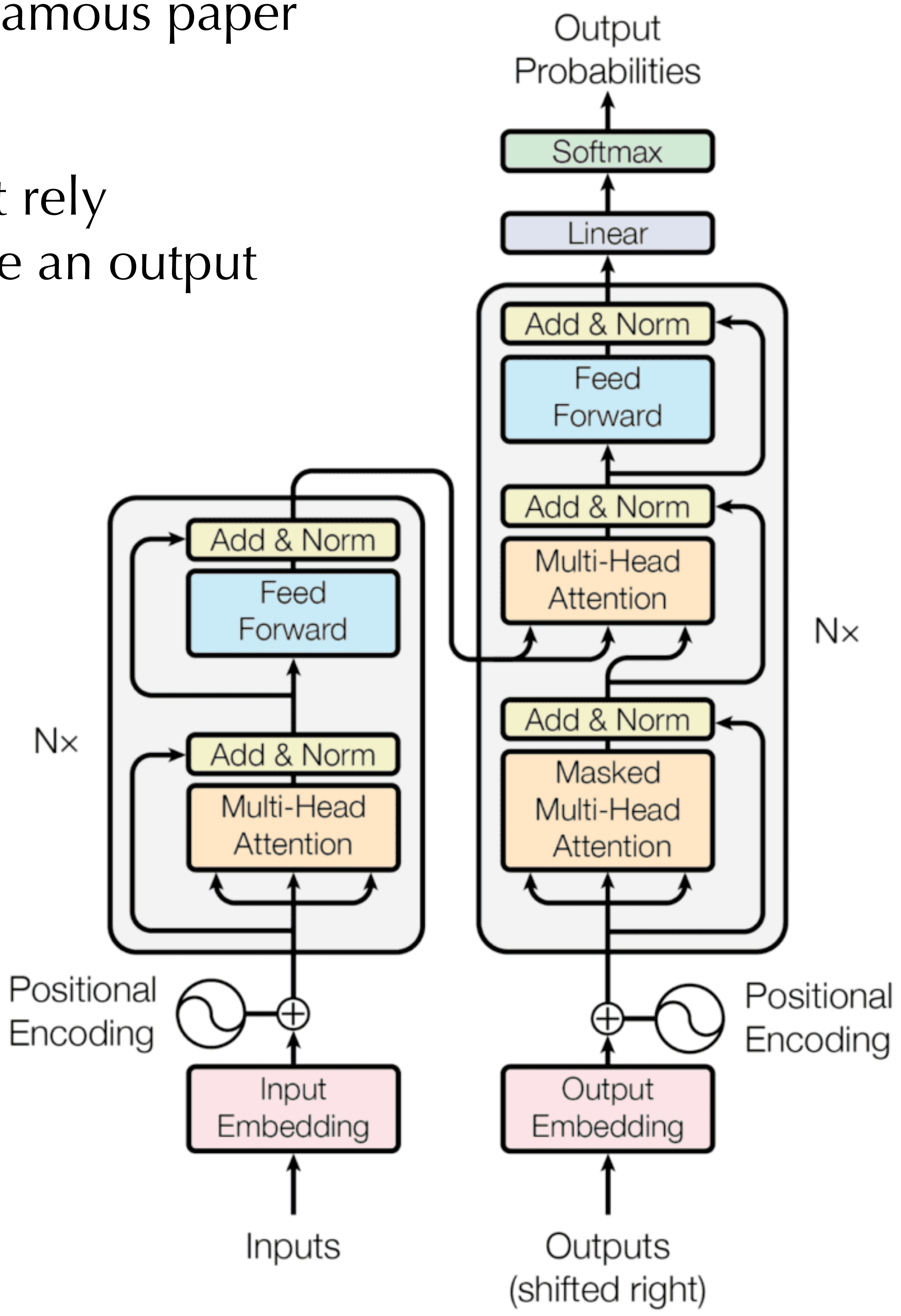
- **A sentence: “The dog chased the cat accross the street”**

- **The query:** “Dog” → the input that is being searched for
- **The key:** every word in the sentence
- **The value:** the general meaning of each word in the sentence (e.g. semantic)

- K + V = the database that is being searched
- **The model compares the query and the keys to identify relevant information and then uses the associated values to generate the output**

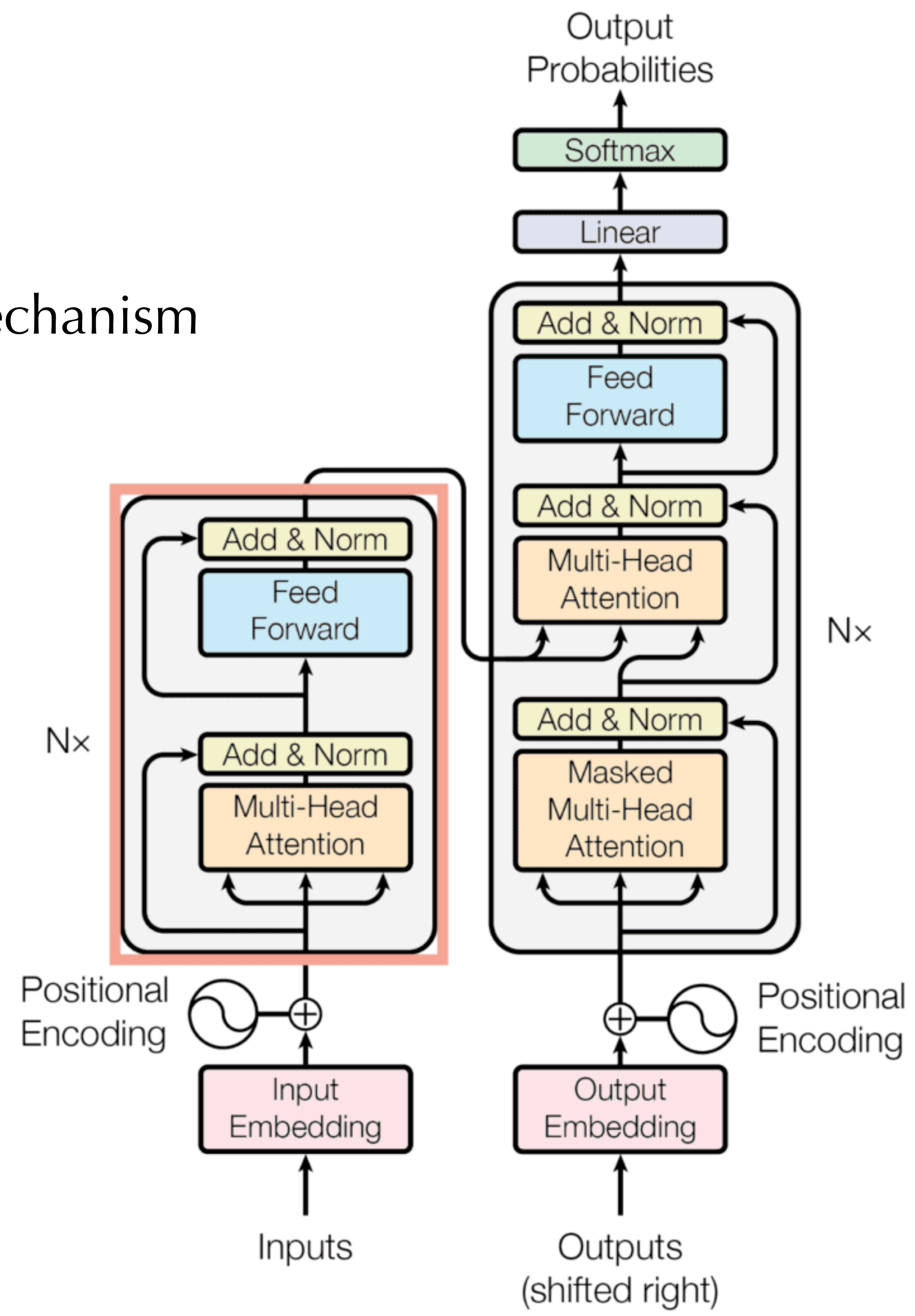
The full transformer model

- Was in introduced by Vaswani et al in 2017 in the famous paper "Attention is All You Need" (73K citations)
- Follows an **encoder-decoder structure** but does not rely on recurrence and convolutions in order to generate an output
- It improved state-of-the-art performance on natural language processing tasks
- Let's break it down…



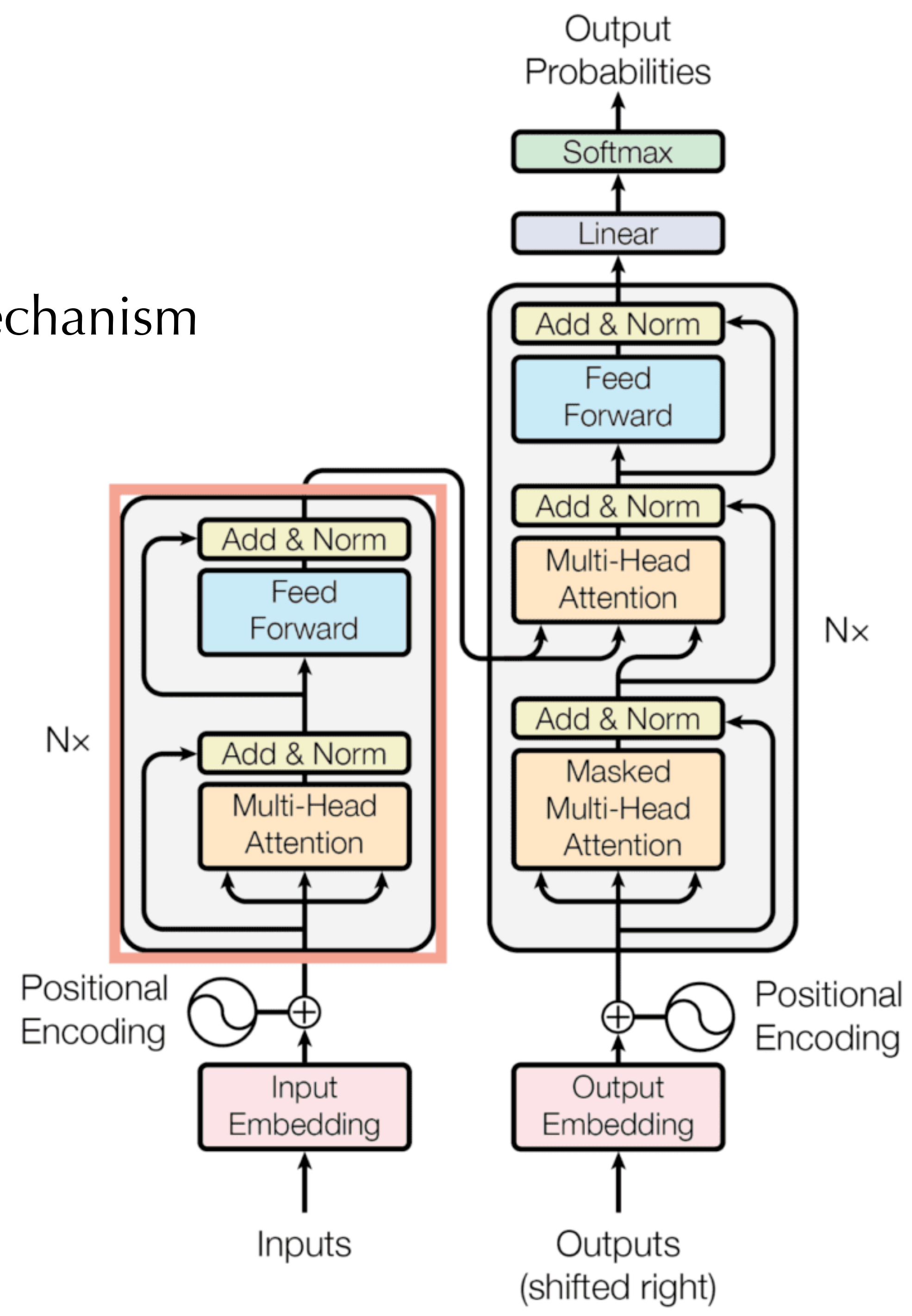
The encoder

- The encoder consists of a stack of N identical layers, where each layer is composed of two sublayers:
  - the 1st sublayer implements a multi-head self-attention mechanism
  - the 2nd sublayer consists of two MLPs



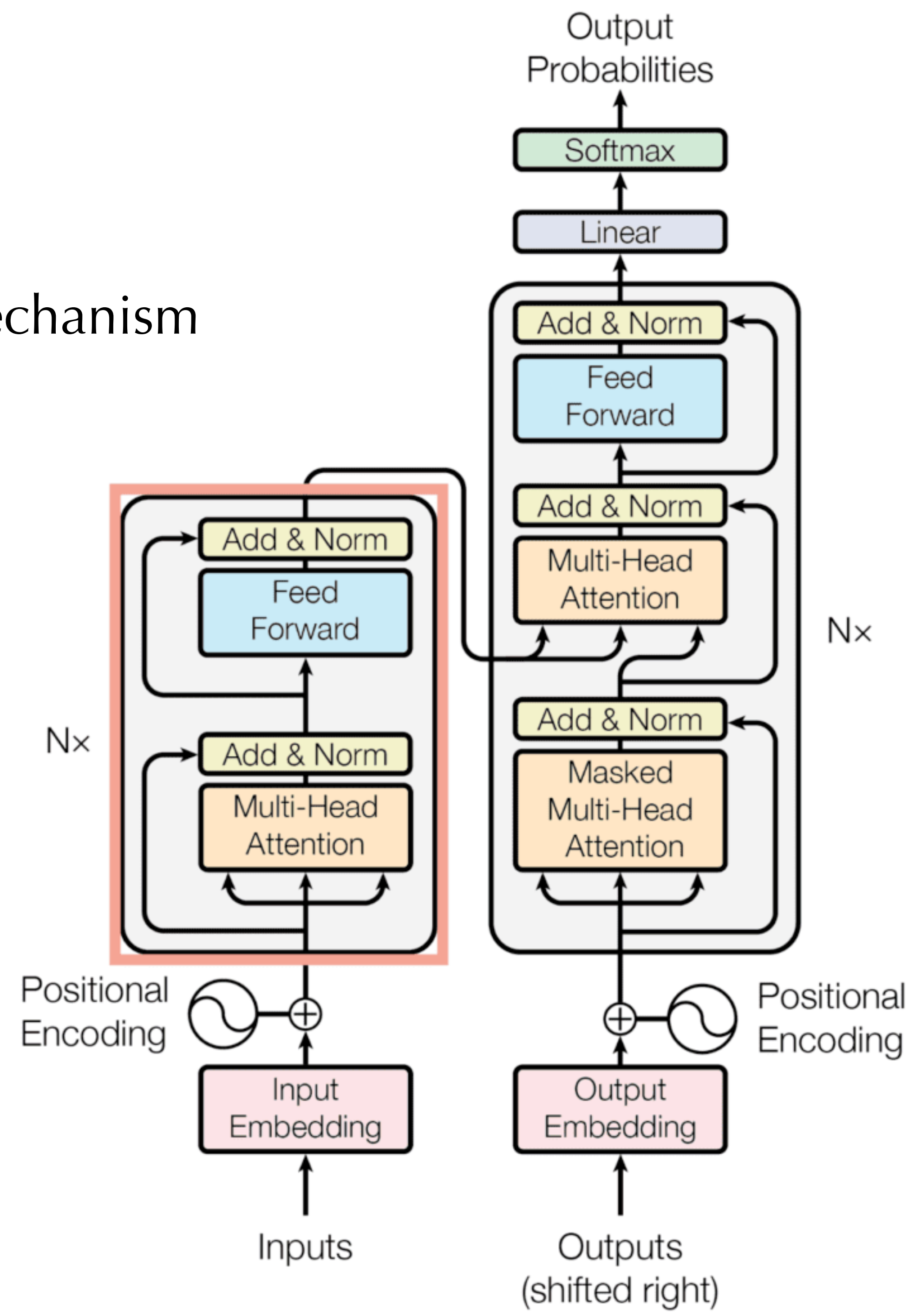
The encoder

- The encoder consists of a stack of N identical layers, where each layer is composed of two sublayers:
  - the 1st sublayer implements a multi-head self-attention mechanism
  - the 2nd sublayer consists of two MLPs
- The six layers of the encoder apply the same linear transformations to all the words in the input sequence (or nodes in the graph for graph data) but *each* layer employs different weights



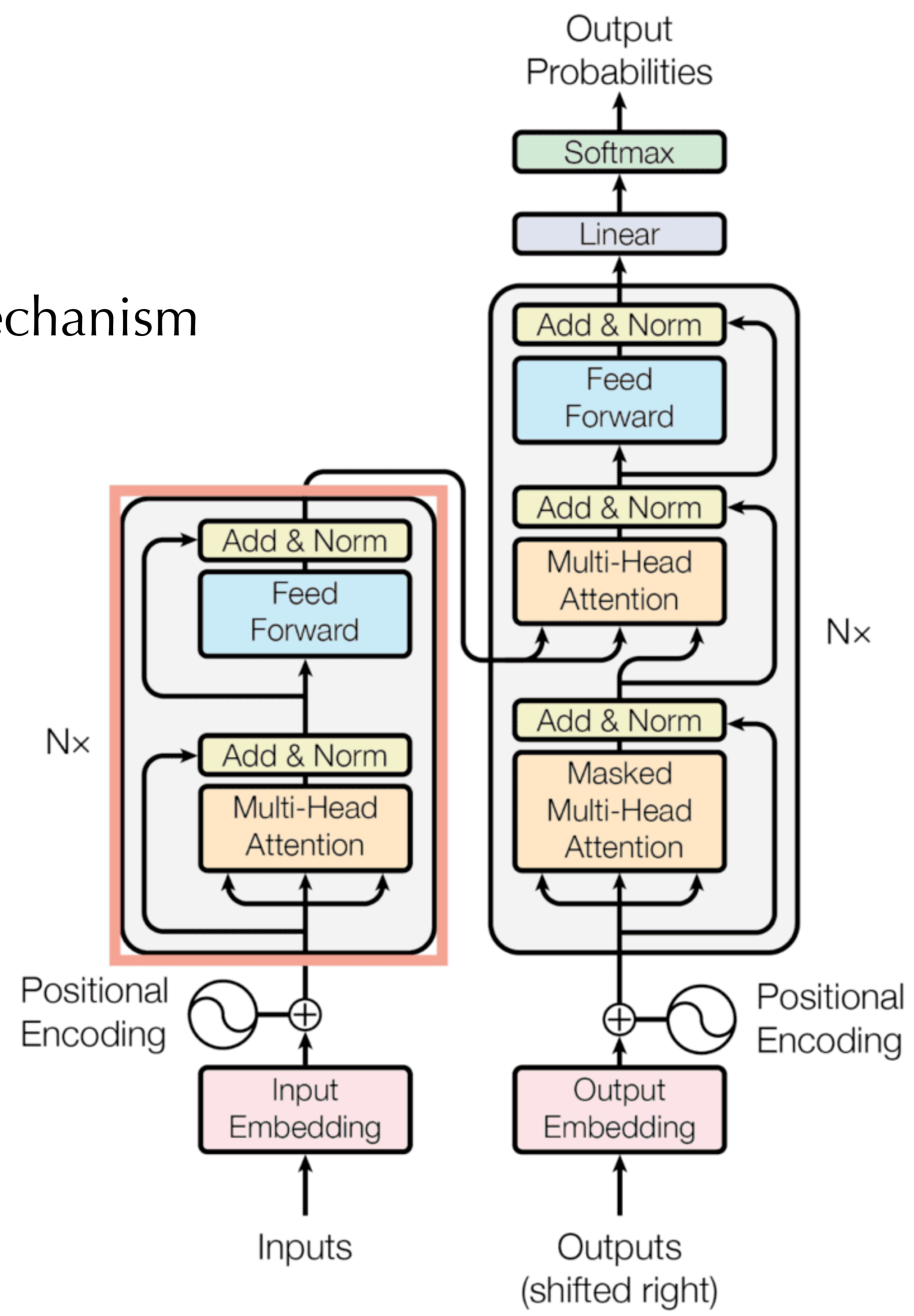
The encoder

- The encoder consists of a stack of N identical layers, where each layer is composed of two sublayers:
  - the 1st sublayer implements a multi-head self-attention mechanism
  - the 2nd sublayer consists of two MLPs
- The six layers of the encoder apply the same linear transformations to all the words in the input sequence (or nodes in the graph for graph data) but *each* layer employs different weights
- Each of these two sublayers has a **residual connection** around it



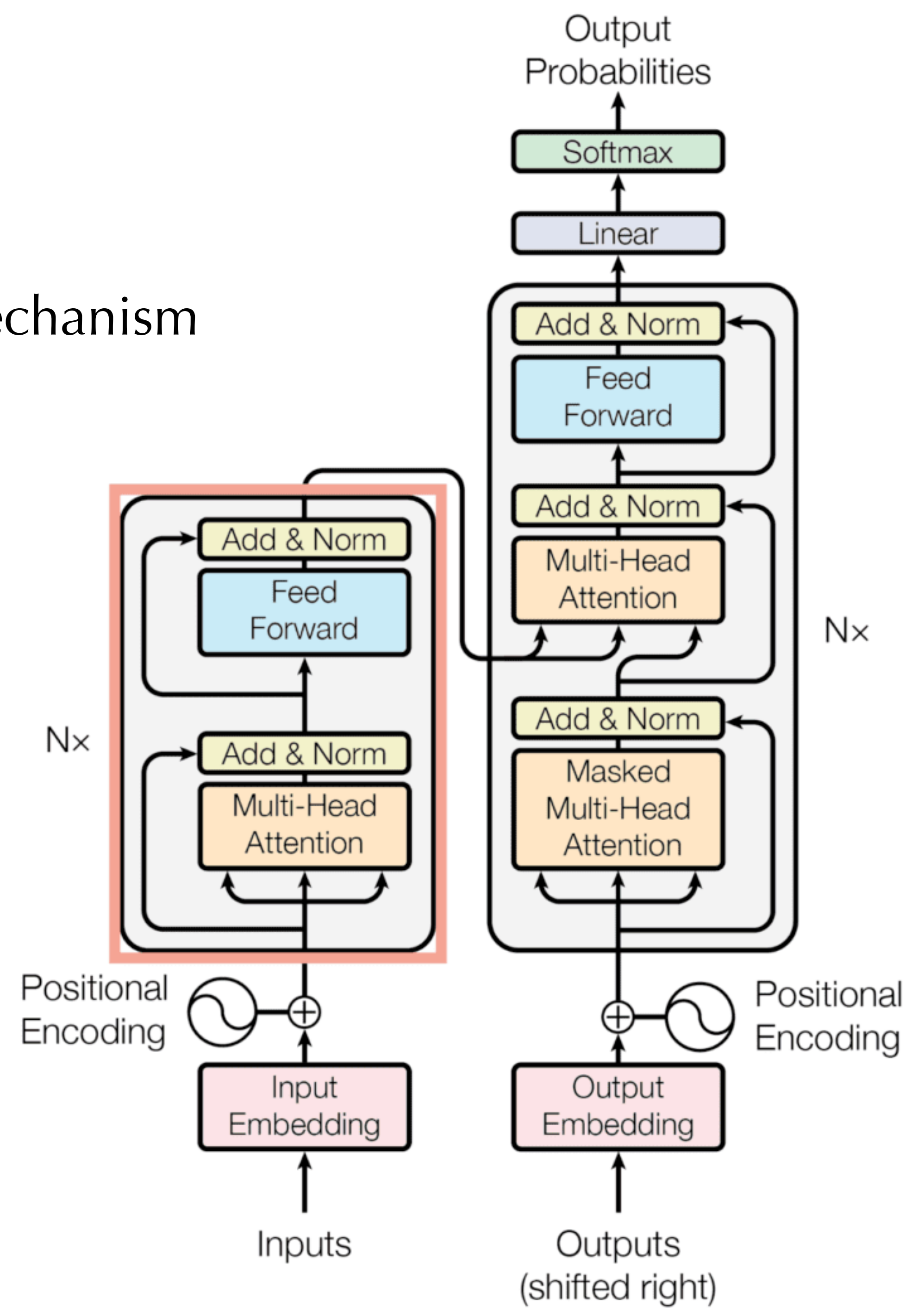
The encoder

- The encoder consists of a stack of N identical layers, where each layer is composed of two sublayers:
  - the 1st sublayer implements a multi-head self-attention mechanism
  - the 2nd sublayer consists of two MLPs
- The six layers of the encoder apply the same linear transformations to all the words in the input sequence (or nodes in the graph for graph data) but *each* layer employs different weights
- Each of these two sublayers has a **residual connection** around it
- Each sublayer is also succeeded by a normalization layer which normalizes the sum between the sublayer input and the output generated by the sublayer itself



The encoder

- The encoder consists of a stack of N identical layers, where each layer is composed of two sublayers:
  - the 1st sublayer implements a multi-head self-attention mechanism
  - the 2nd sublayer consists of two MLPs
- The six layers of the encoder apply the same linear transformations to all the words in the input sequence (or nodes in the graph for graph data) but *each* layer employs different weights
- Each of these two sublayers has a **residual connection** around it
- Each sublayer is also succeeded by a normalization layer which normalizes the sum between the sublayer input and the output generated by the sublayer itself



- nb, the architecture cannot inherently capture any information about the relative positions of the words in the sequence (no recurrence) → **positional encoding** needed

The decoder

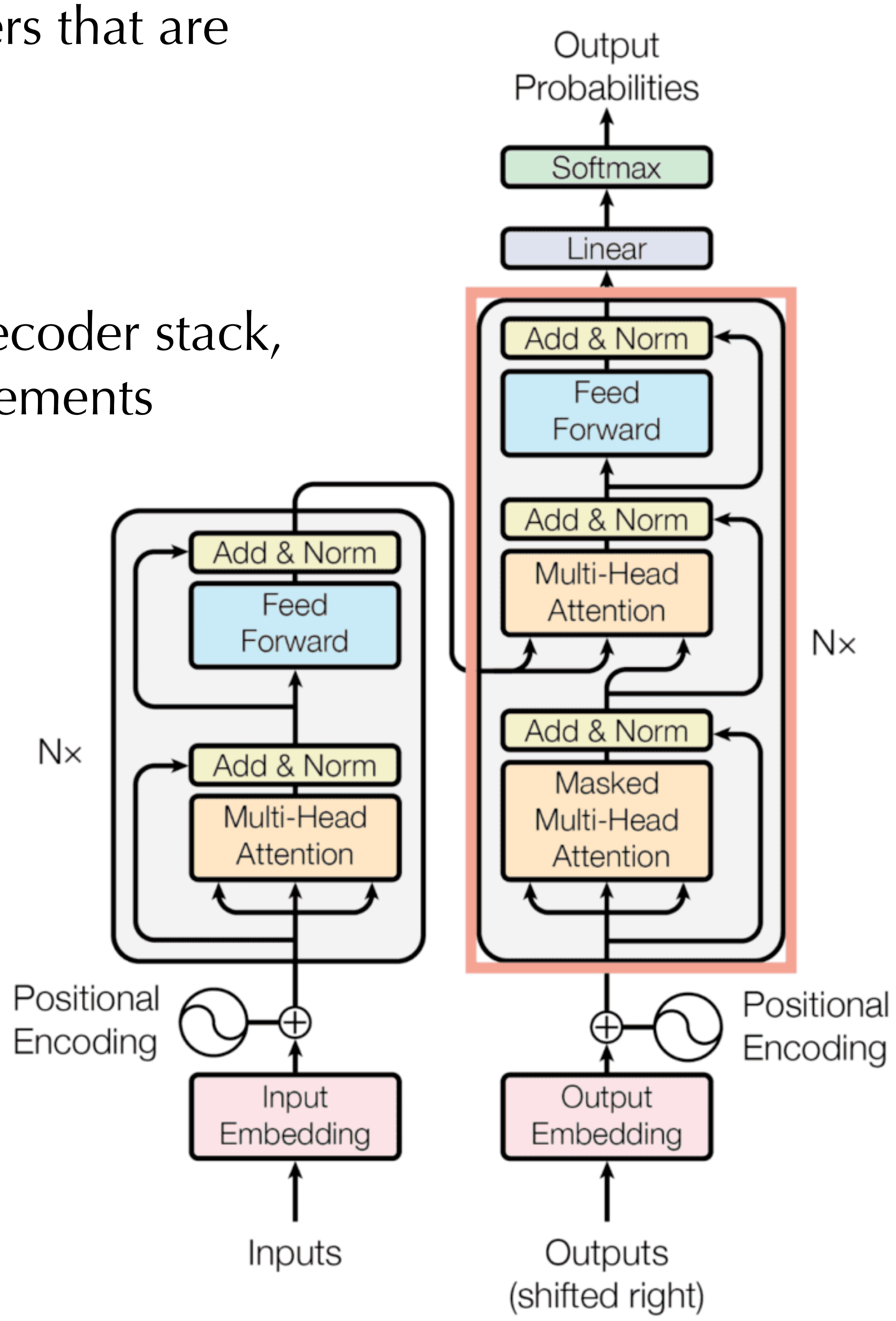
- The decoder also consists of a stack of N identical layers that are each composed of **three sublayers**:

1. The **1st sublayer** receives the previous output of the decoder stack, augments it with positional and information, and implements multi-head self-attention over it

    - while the encoder is designed to attend to all words in the input sequence *regardless* of their position in the sequence, the decoder is modified to attend *only* to the preceding encoded sequence

    - the prediction for a word at position i can only depend on the known outputs for the words that come before it in the sequence

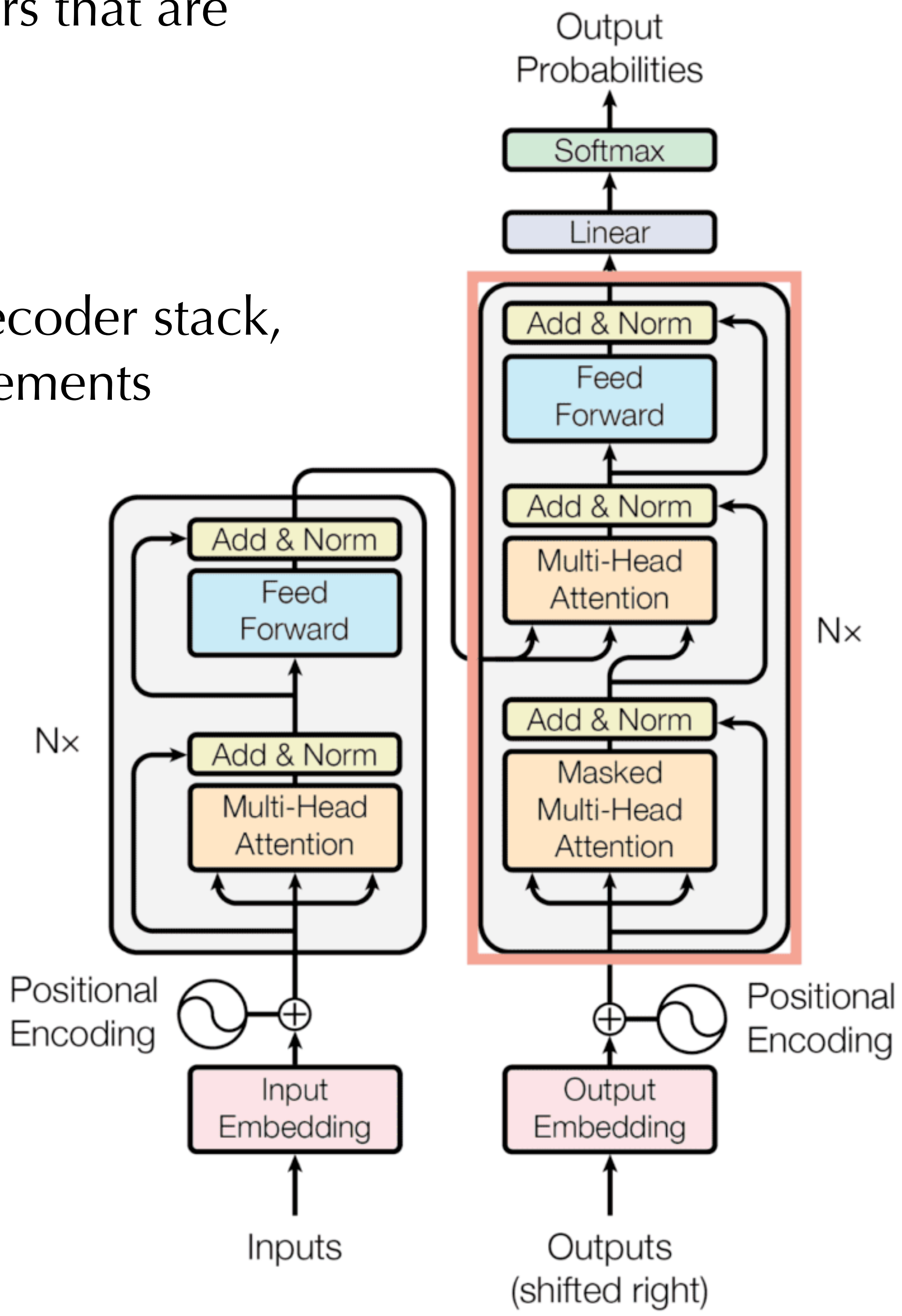
    - the word positioning can be obtained with a matrix that masks illegal connections in the sequence



The decoder

- The decoder also consists of a stack of *N* identical layers that are each composed of **three sublayers**:

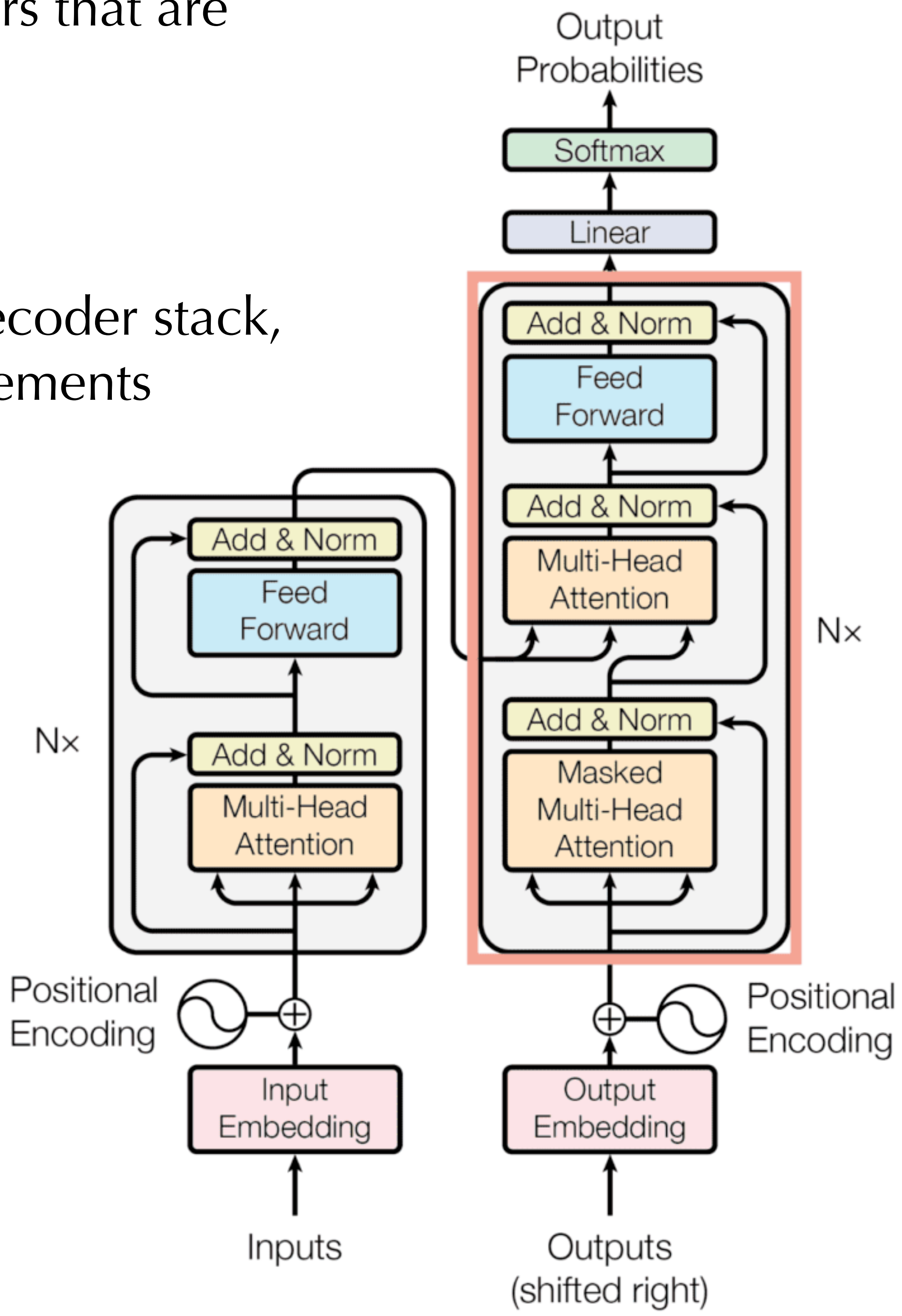
1. The **1st sublayer** receives the previous output of the decoder stack, augments it with positional and information, and implements multi-head self-attention over it



The decoder

- The decoder also consists of a stack of N identical layers that are each composed of **three sublayers**:

1. The **1st sublayer** receives the previous output of the decoder stack, augments it with positional and information, and implements multi-head self-attention over it
    - while the encoder is designed to attend to all words in the input sequence *regardless* of their position in the sequence, the decoder is modified to attend *only* to the preceding encoded sequence



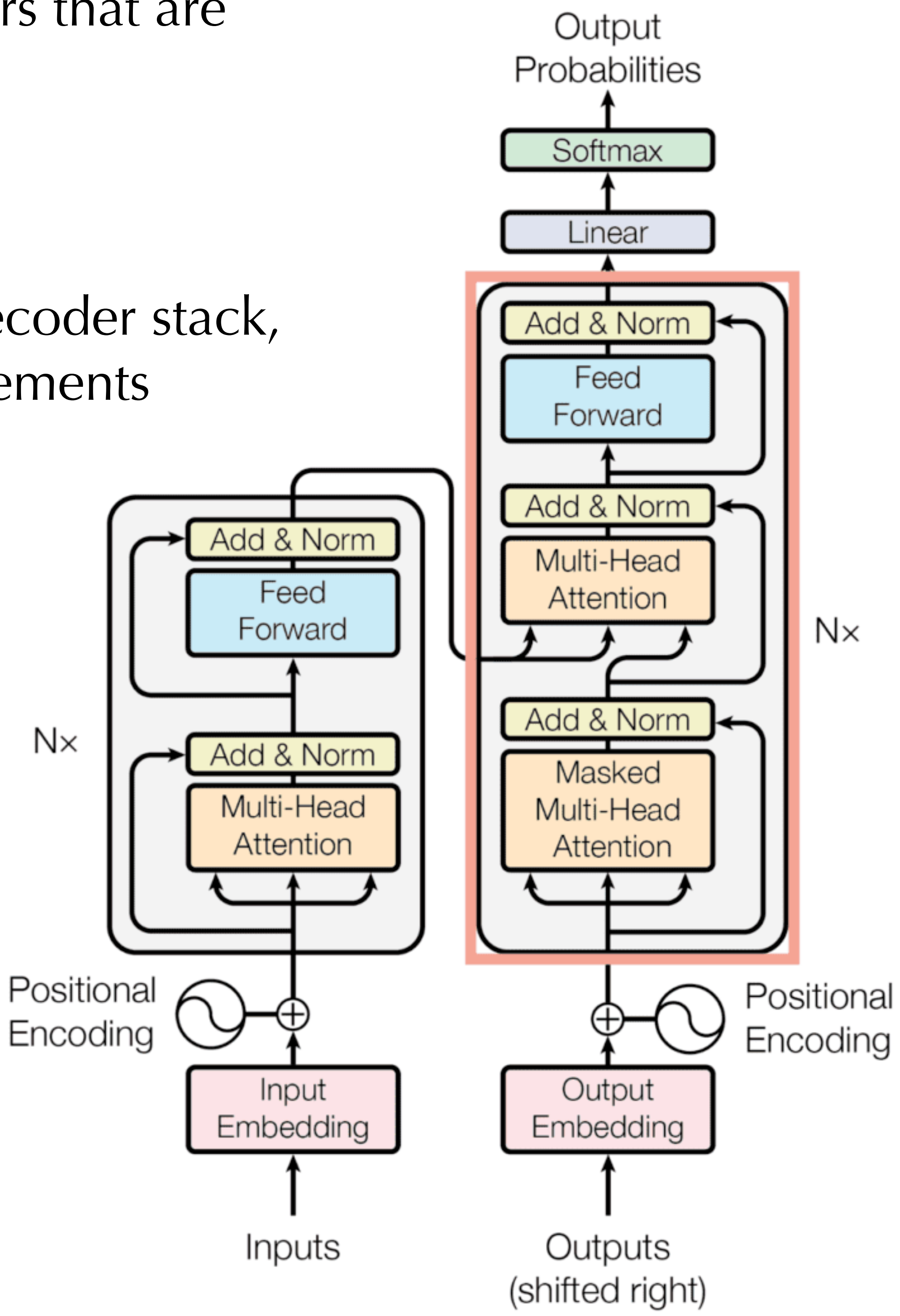
The decoder

- The decoder also consists of a stack of N identical layers that are each composed of **three sublayers**:

1. The **1st sublayer** receives the previous output of the decoder stack, augments it with positional and information, and implements multi-head self-attention over it

   - while the encoder is designed to attend to all words in the input sequence *regardless* of their position in the sequence, the decoder is modified to attend *only* to the preceding encoded sequence

   - the prediction for a word at position i can only depend on the known outputs for the words that come before it in the sequence → **decoder is sequential**



The decoder

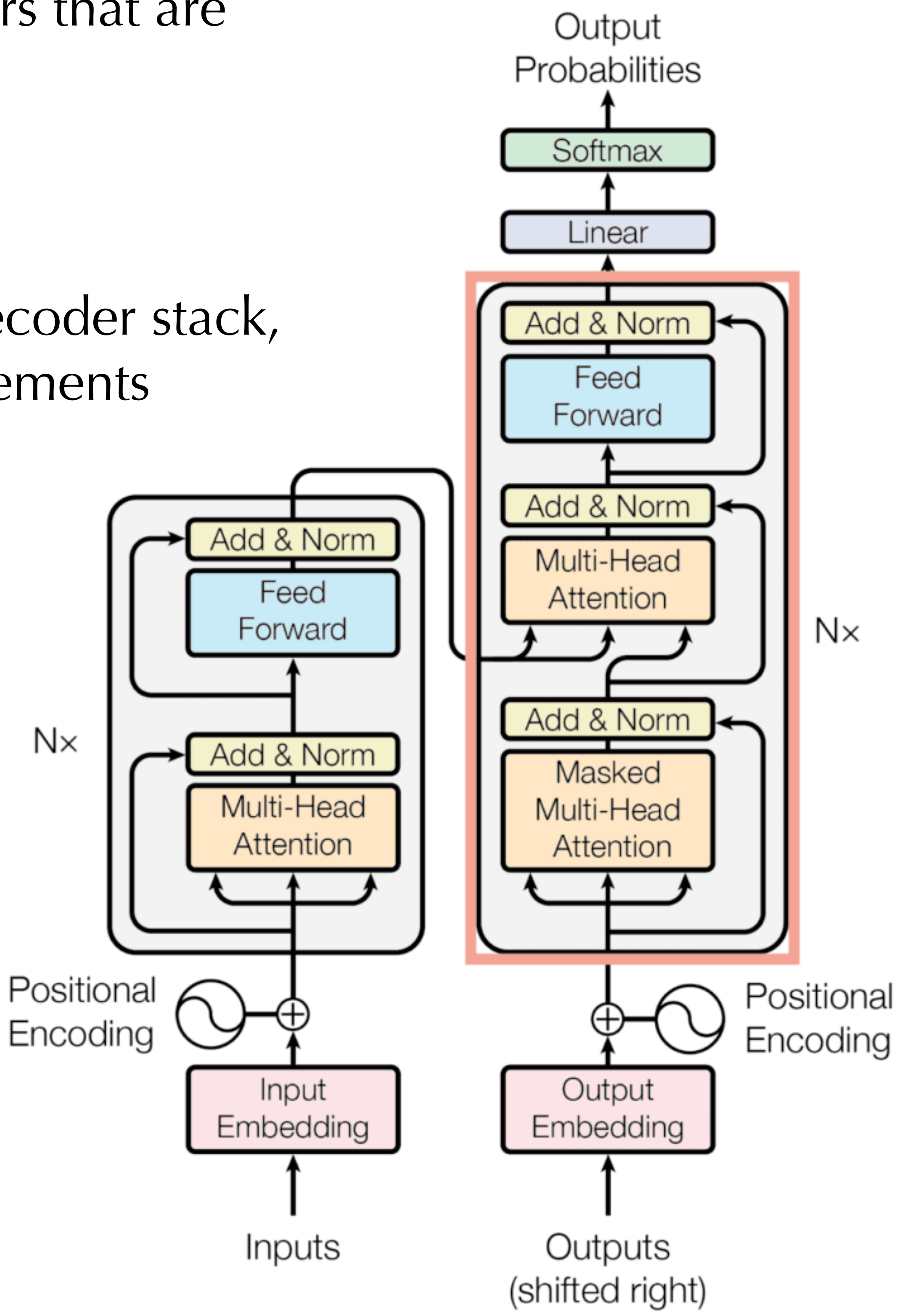
- The decoder also consists of a stack of N identical layers that are each composed of **three sublayers**:

1. The **1st sublayer** receives the previous output of the decoder stack, augments it with positional and information, and implements multi-head self-attention over it

   - while the encoder is designed to attend to all words in the input sequence *regardless* of their position in the sequence, the decoder is modified to attend *only* to the preceding encoded sequence

   - the prediction for a word at position i can only depend on the known outputs for the words that come before it in the sequence → **decoder is sequential**

   - the word positioning can be obtained with a matrix that masks illegal connections in the sequence

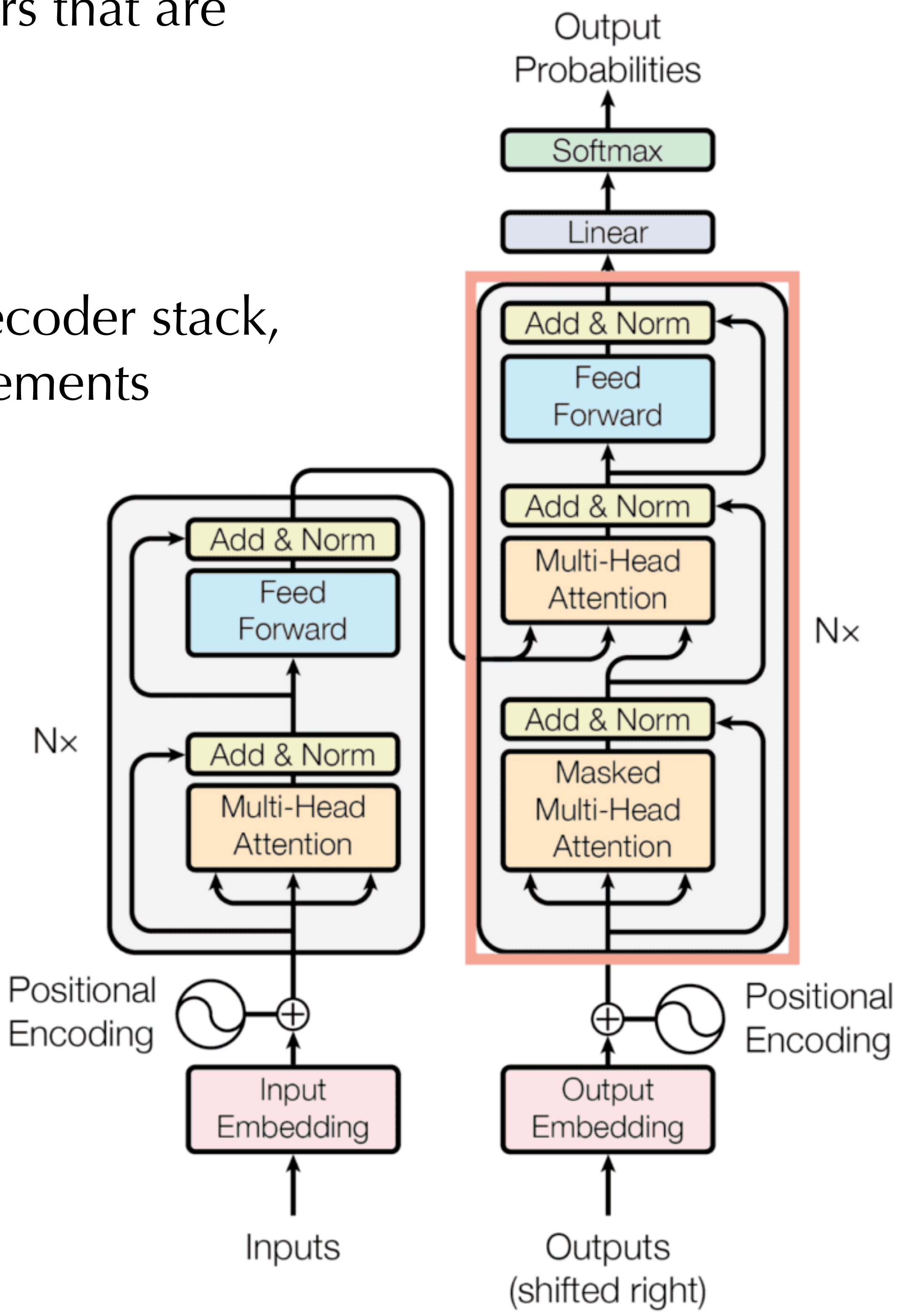


The decoder

- The decoder also consists of a stack of N identical layers that are each composed of **three sublayers**:

1. The **1st sublayer** receives the previous output of the decoder stack, augments it with positional and information, and implements multi-head self-attention over it

2. The **2nd layer** implements the multi-head attention mechanism similar to the one implemented in the first sublayer of the encoder



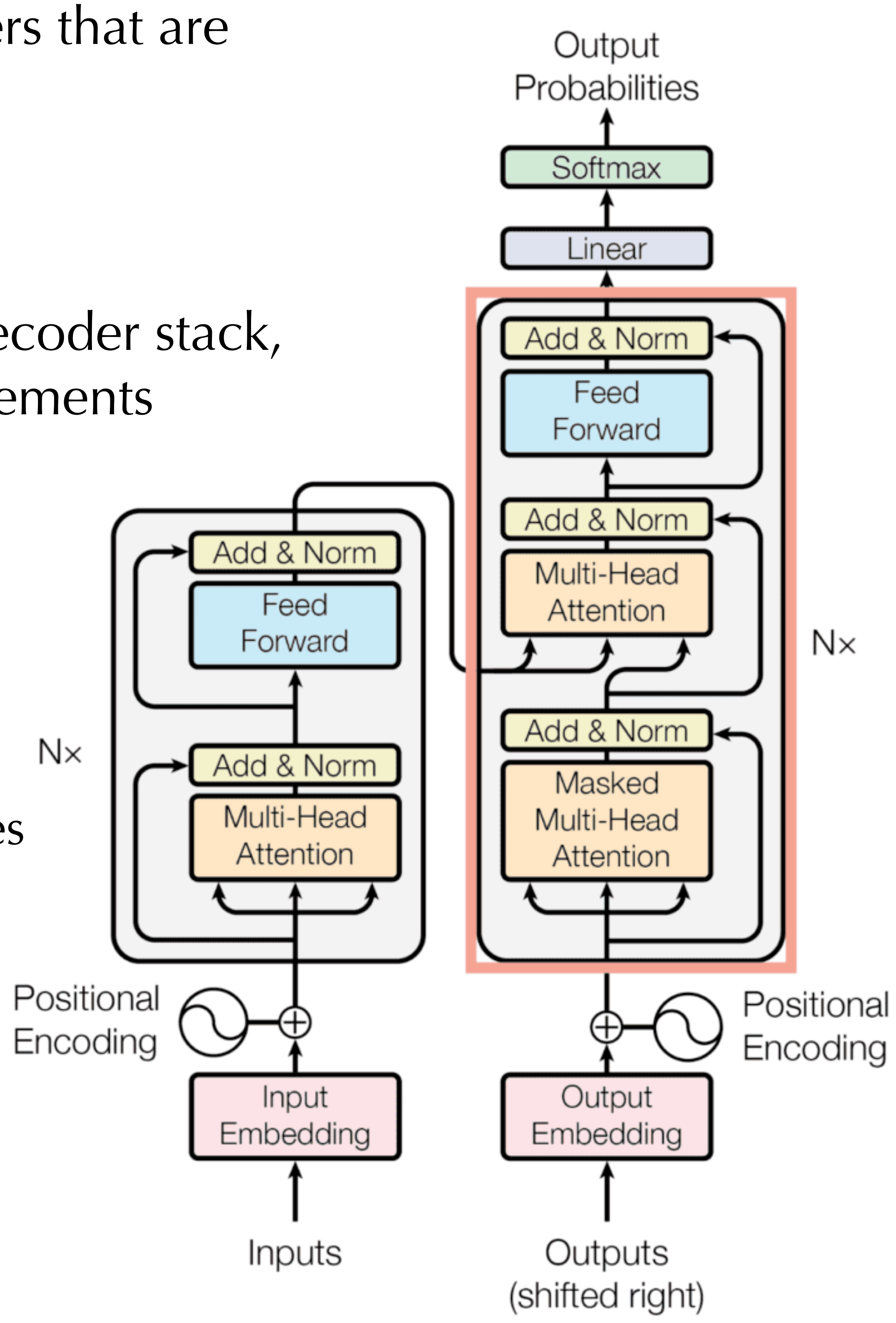
The decoder

- The decoder also consists of a stack of N identical layers that are each composed of **three sublayers**:

1. The **1st sublayer** receives the previous output of the decoder stack, augments it with positional and information, and implements multi-head self-attention over it

2. The **2nd layer** implements the multi-head attention mechanism similar to the one implemented in the first sublayer of the encoder

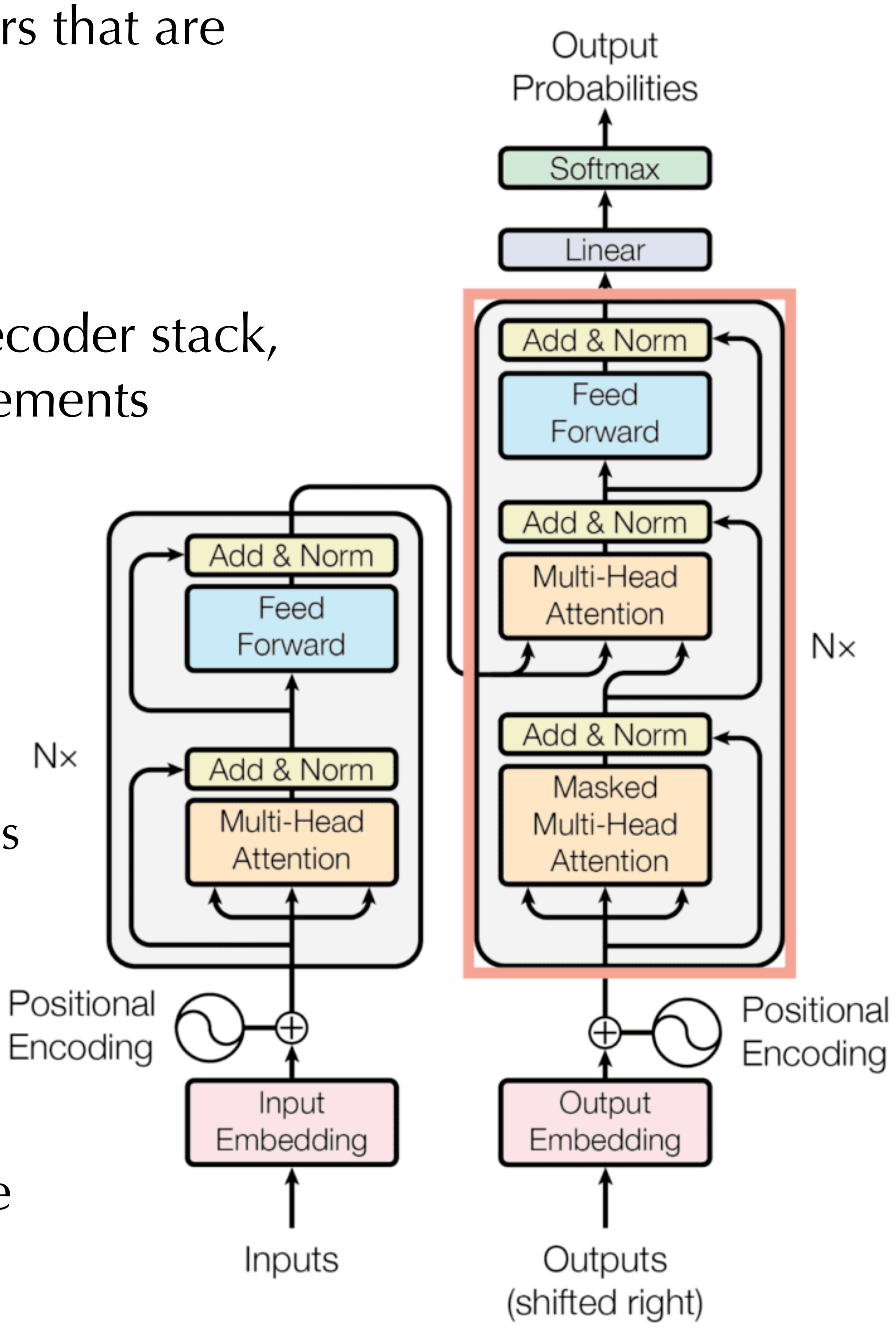
   - on the decoder side, this multi-head mechanism receives the queries from the previous decoder sublayer and the keys and values from the output of the encoder. This allows the decoder to attend to all the words in the input sequence



The decoder

- The decoder also consists of a stack of N identical layers that are each composed of **three sublayers**

1. The **1st sublayer** receives the previous output of the decoder stack, augments it with positional and information, and implements multi-head self-attention over it
2. The **2nd layer** implements the multi-head attention mechanism similar to the one implemented in the first sublayer of the encoder
    - on the decoder side, this multi-head mechanism receives the queries from the previous decoder sublayer and the keys and values from the output of the encoder. This allows the decoder to attend to all the words in the input sequence
3. The **3rd layer** implements two MLPs, similar to the one implemented in the second sublayer of the encoder.

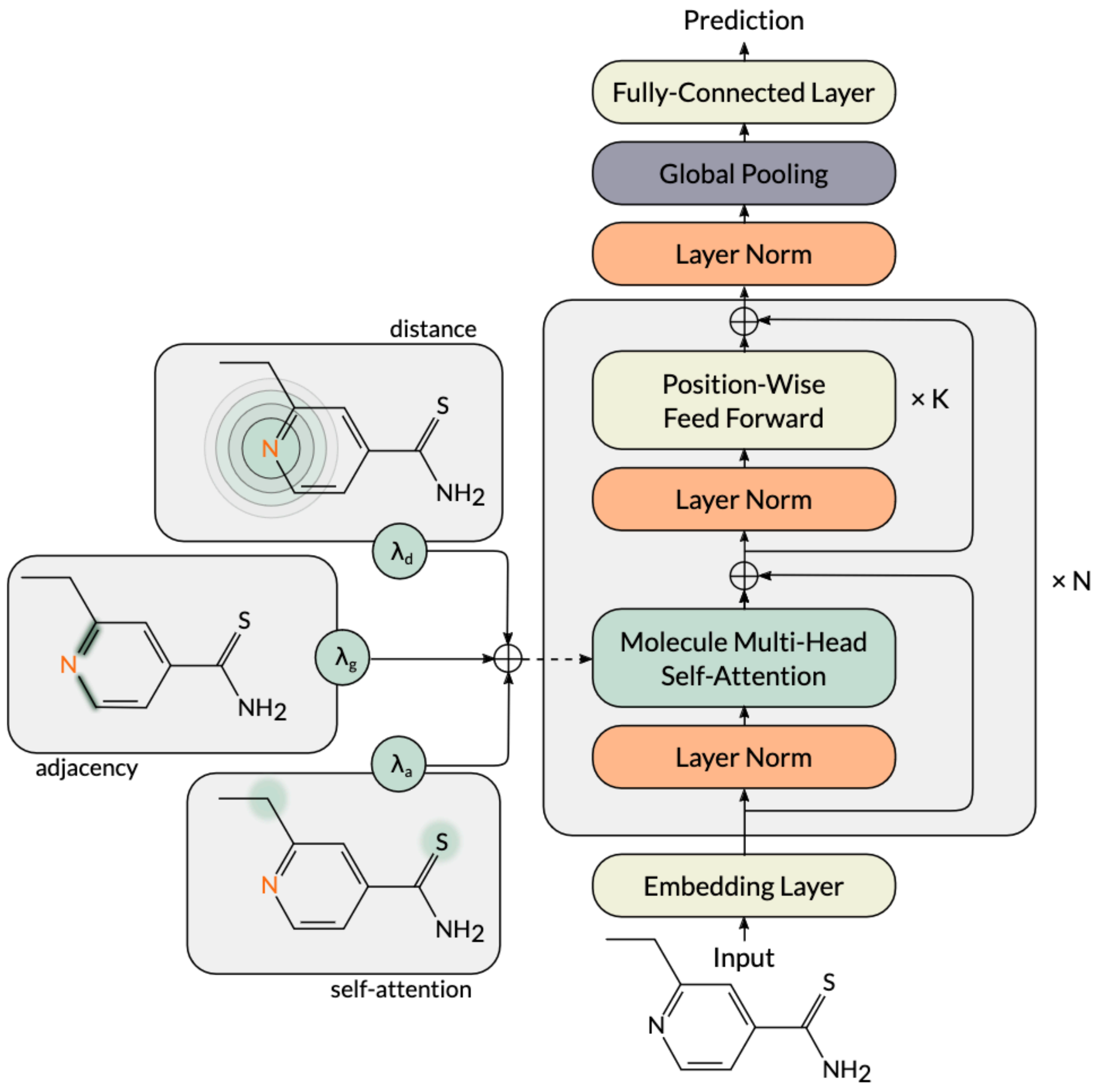


Why are transformers powerful?

- **Attention mechanism:** allows the model to focus on the most relevant parts of the input at each step of processing
- **Parallel processing:** can process all inputs in parallel, unlike RNNs — this makes transformers faster and more efficient for processing large datasets
- **Contextual understanding:** can capture long-term dependencies and context in the input data
- **Transfer learning:** can be pre-trained on large amounts of data and tasks and then fine-tuned for other specific tasks → quickly adapt to new domains with minimal additional training

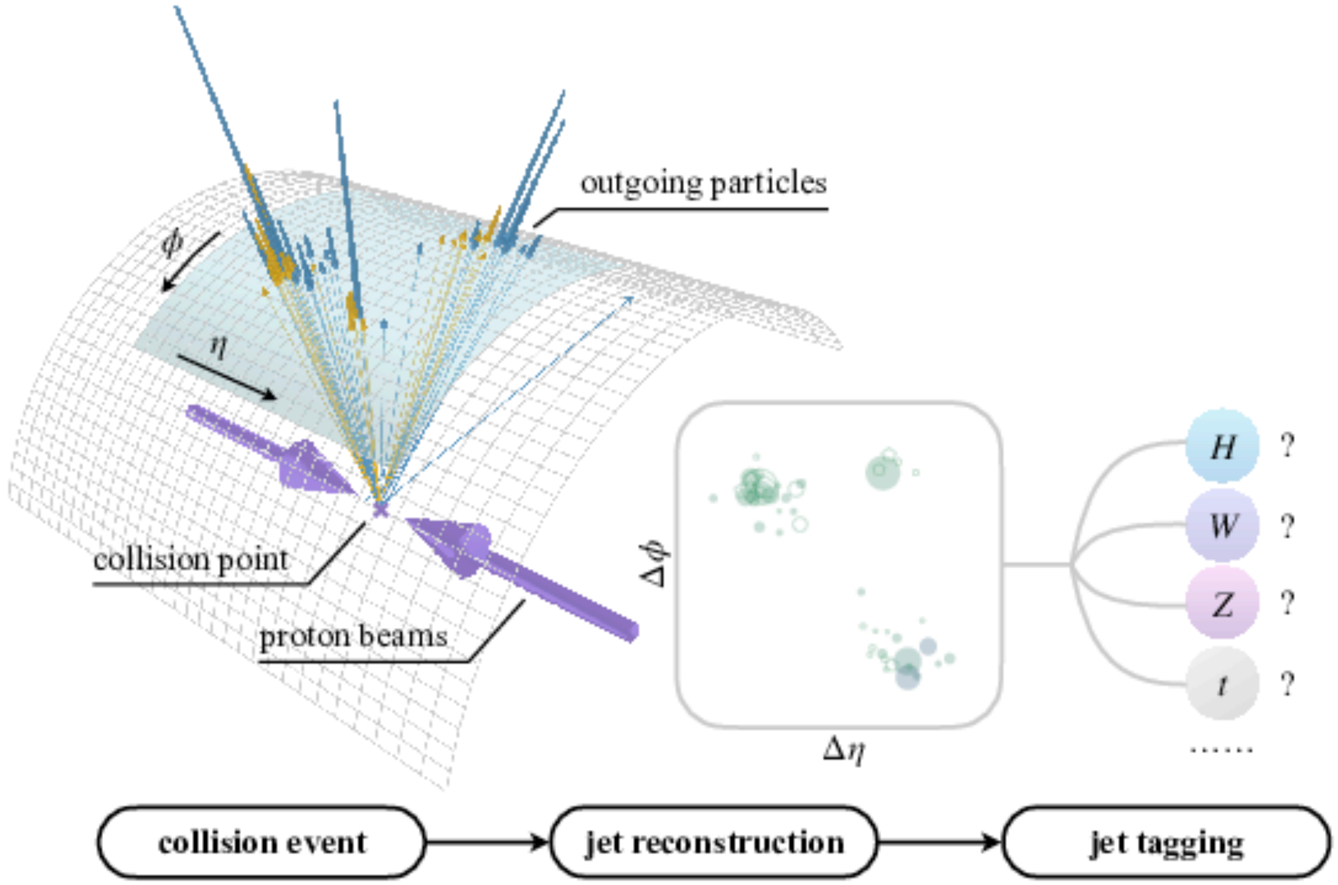
Applications of transformers in science

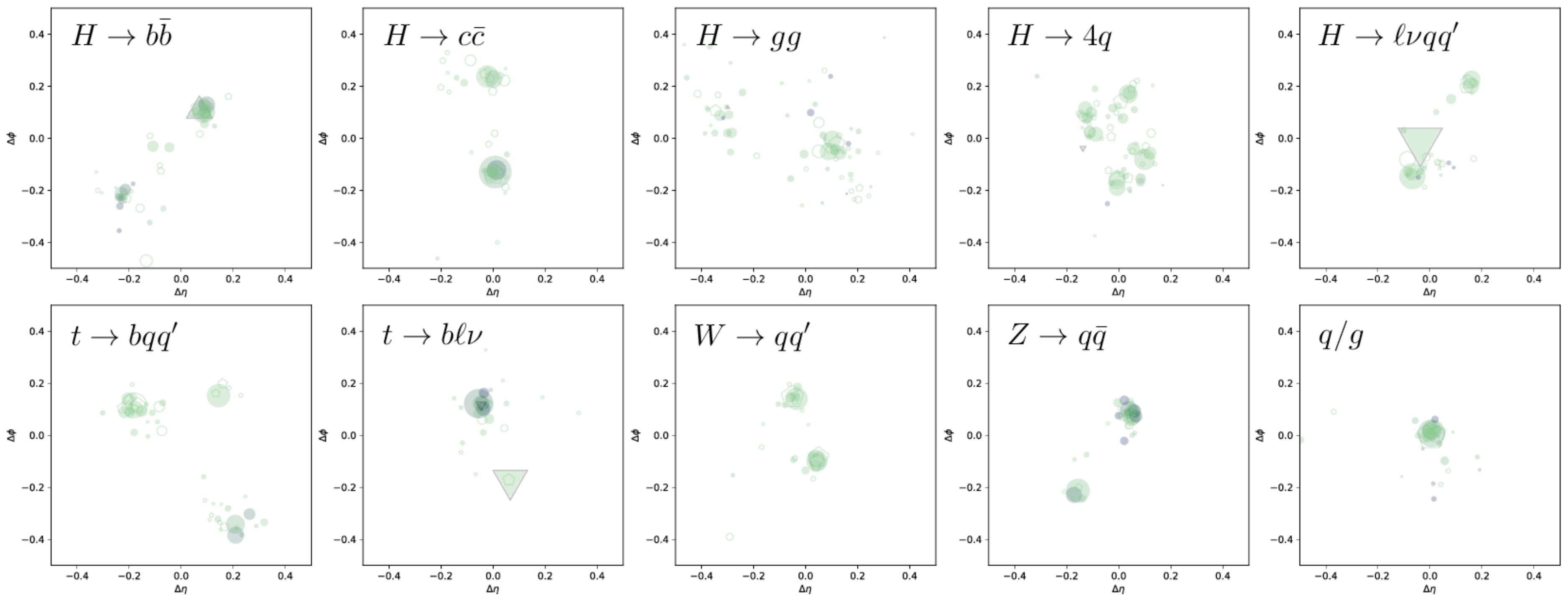
- “Molecule Attention Transformer” by L. Maziarka et al. (NeurIPS ‘19)
- Challenge: find a single NN that performs competitively across a range of molecule property prediction tasks
- Key innovation: augment the attention mechanism in Transformer using inter-atomic distances and the molecular graph structure
- **Accelerate drug discovery!**



Applications of transformers in science

- [“Particle Transformer for Jet Tagging” by Huilin Qu et al. (ICML ’22)]
- one model for many tasks
- key innovation: incorporating pairwise particle interactions in the attention mechanism
- also demonstrate fine-tuning for other jet tagging datasets


outgoing particles
ϕ
η
collision point
proton beams
$\Delta\phi$
$\Delta\eta$
H ?
W ?
Z ?
t ?
……
collision event
jet reconstruction
jet tagging


$H \to b\bar{b}$
$H \to c\bar{c}$
$H \to gg$
$H \to 4q$
$H \to \ell\nu qq'$
$t \to bqq'$
$t \to b\ell\nu$
$W \to qq'$
$Z \to q\bar{q}$
q/g
$\Delta\eta$
$\Delta\phi$