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

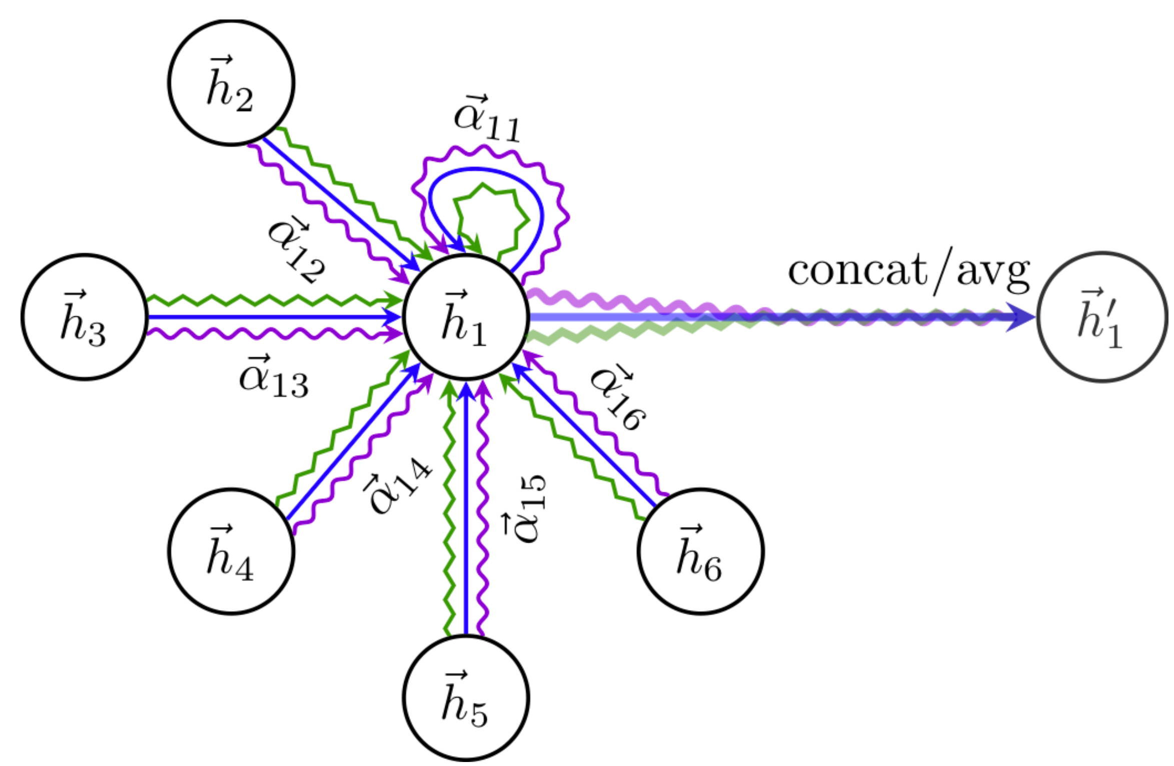
- The GNNs seen so far treat all the nodes in the same way
  - each neighbor contributes equally to update the representation of the central node
- GCN applies weights which however are fixed by the graph structure leading to worse generalization performance
  - DGCNN applies weights 0 and 1 with kNN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

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

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

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

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



P. Veličković et al., ICLR '18

Let's break it down

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

this is the source node i embedding

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

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

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

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

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

this is the source node i embedding

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

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

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

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

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

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

this is the source node i embedding

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

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

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

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

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

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

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

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

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

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

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

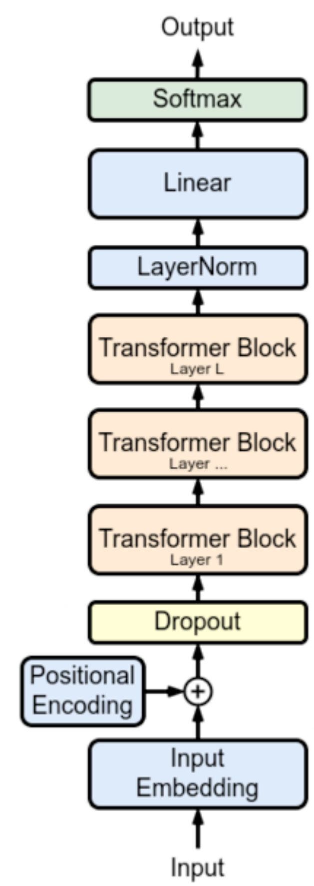
Christopher Hesse Mark Chen Eric Sigler Mateusz Litwin Scott Gray

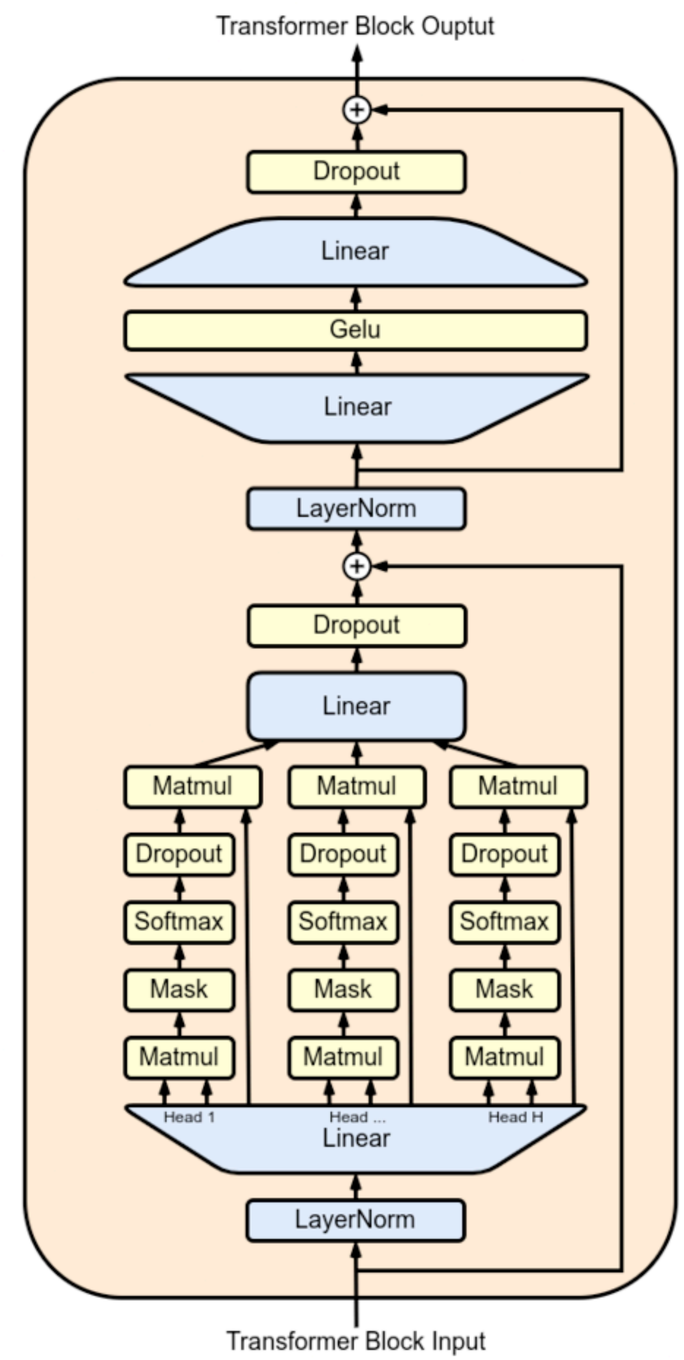
Benjamin Chess Jack Clark Christopher Berner

Sam McCandlish Alec Radford Ilya Sutskever Dario Amodei

OpenAI

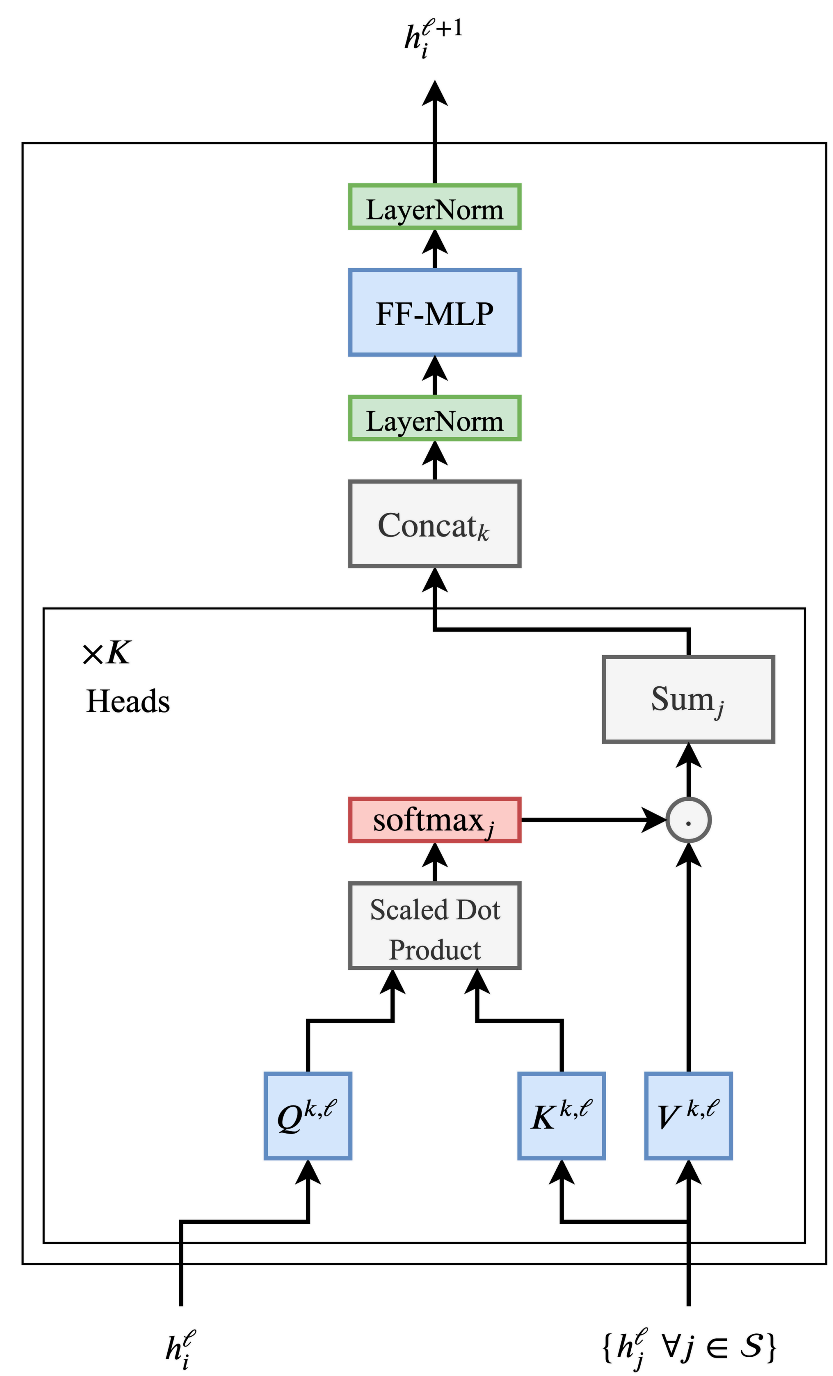
https://arxiv.org/abs/2005.14165





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)**



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

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

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

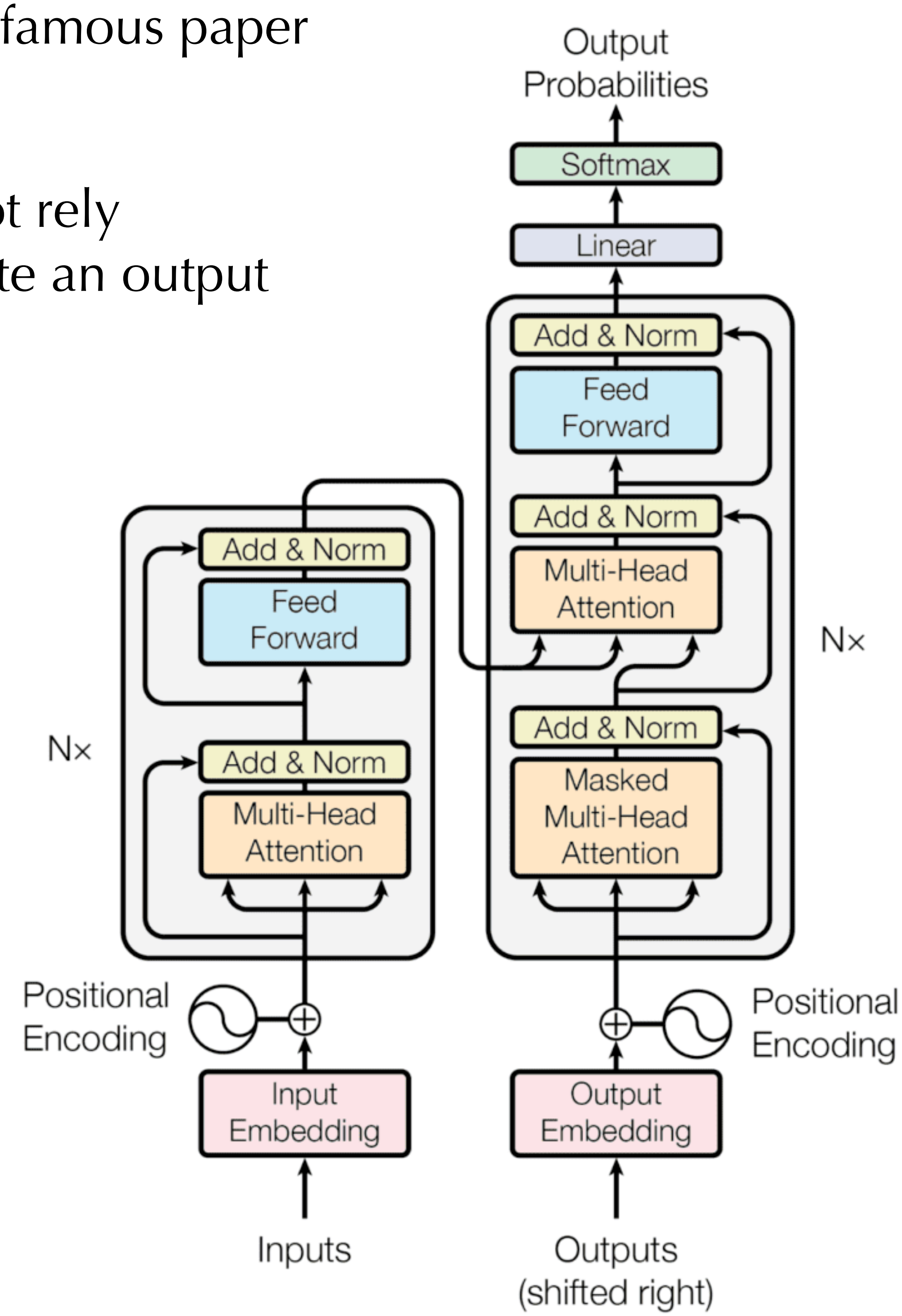
The full transformer model

- 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…



Scaled Dot-Product Attention

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

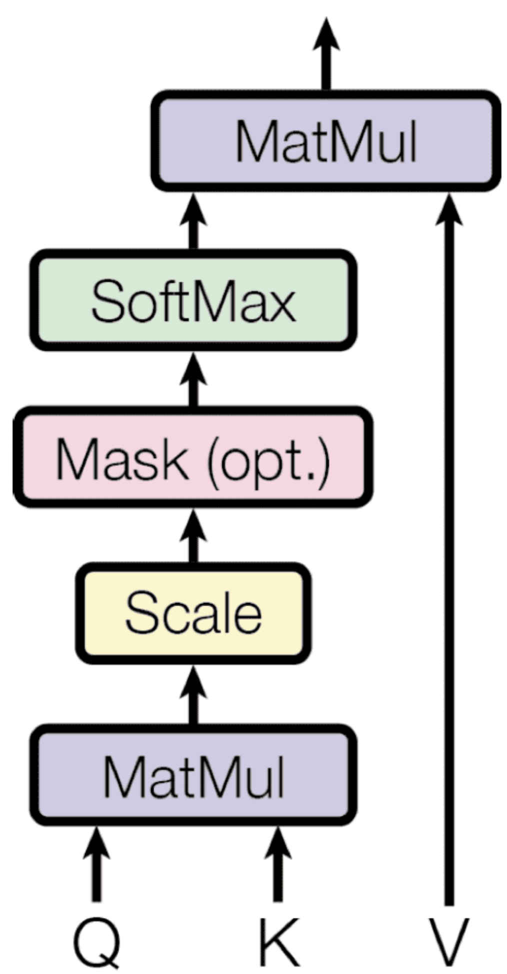
- The scaled dot-product attention first computes a *dot product* for each query, Q, with all of the keys, K.

- 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 $m \times n$

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

  2. Scale each of the alignment scores:

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

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

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

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

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

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**

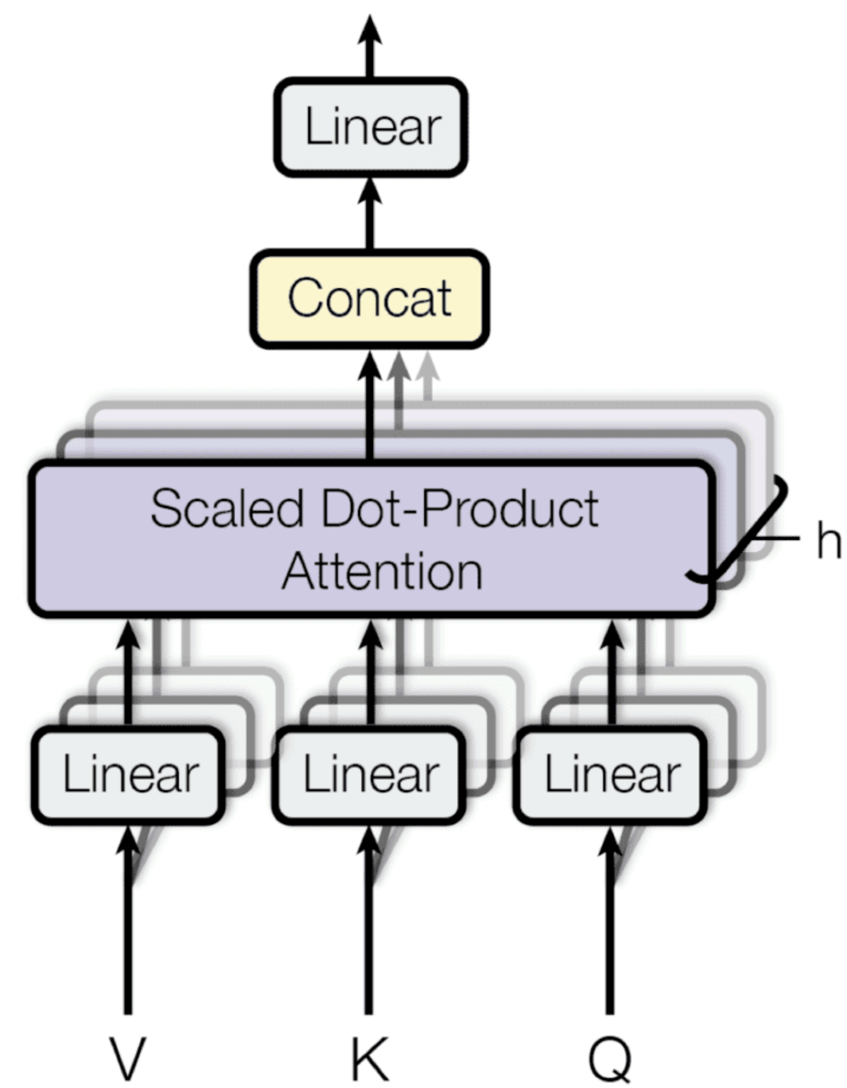
- Their multi-head attention mechanism linearly projects the queries, keys, and values h times, using a different learned projection each time

- 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

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

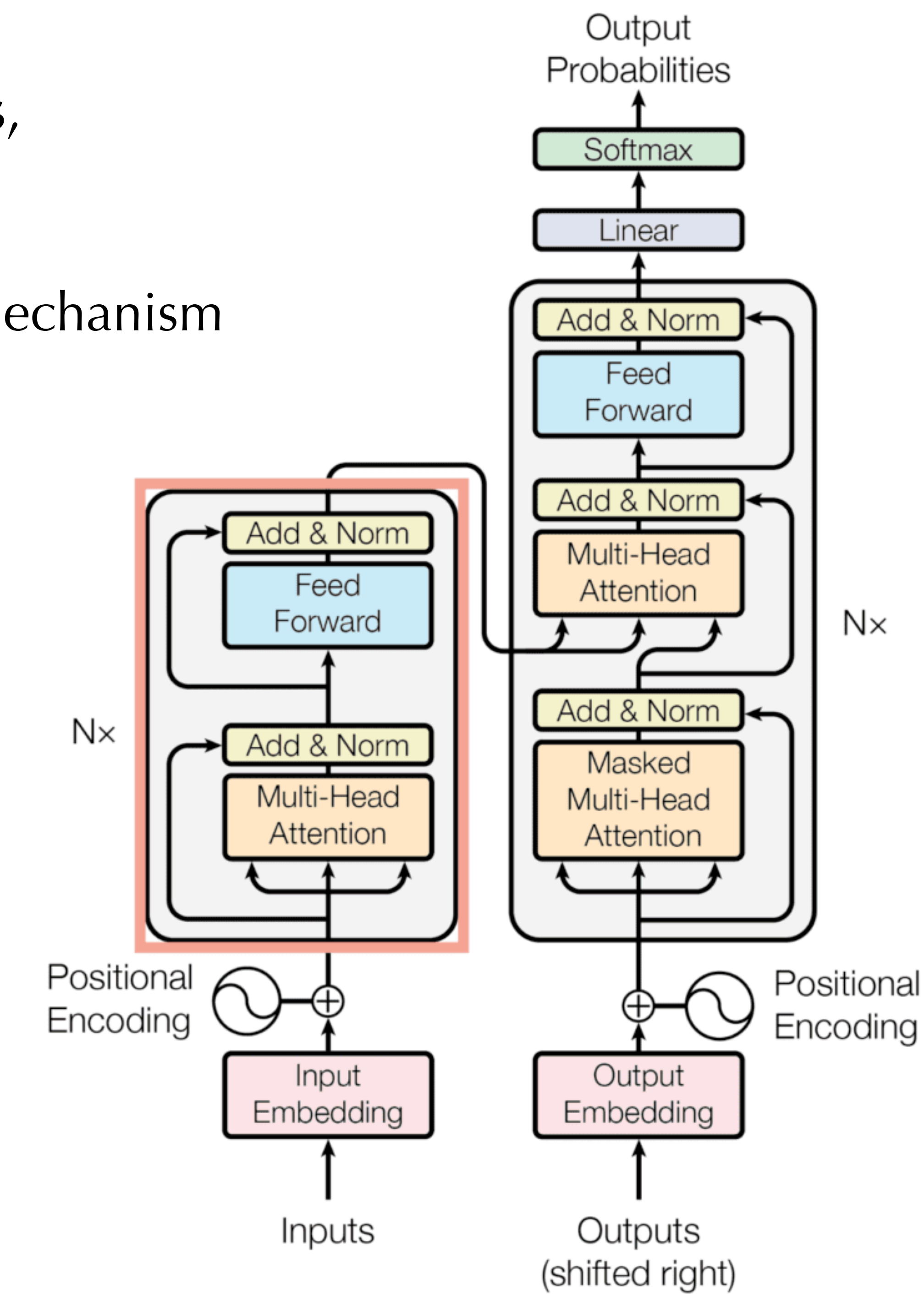
- Each *head* implements a single attention function characterized by its own learned weight matrices

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



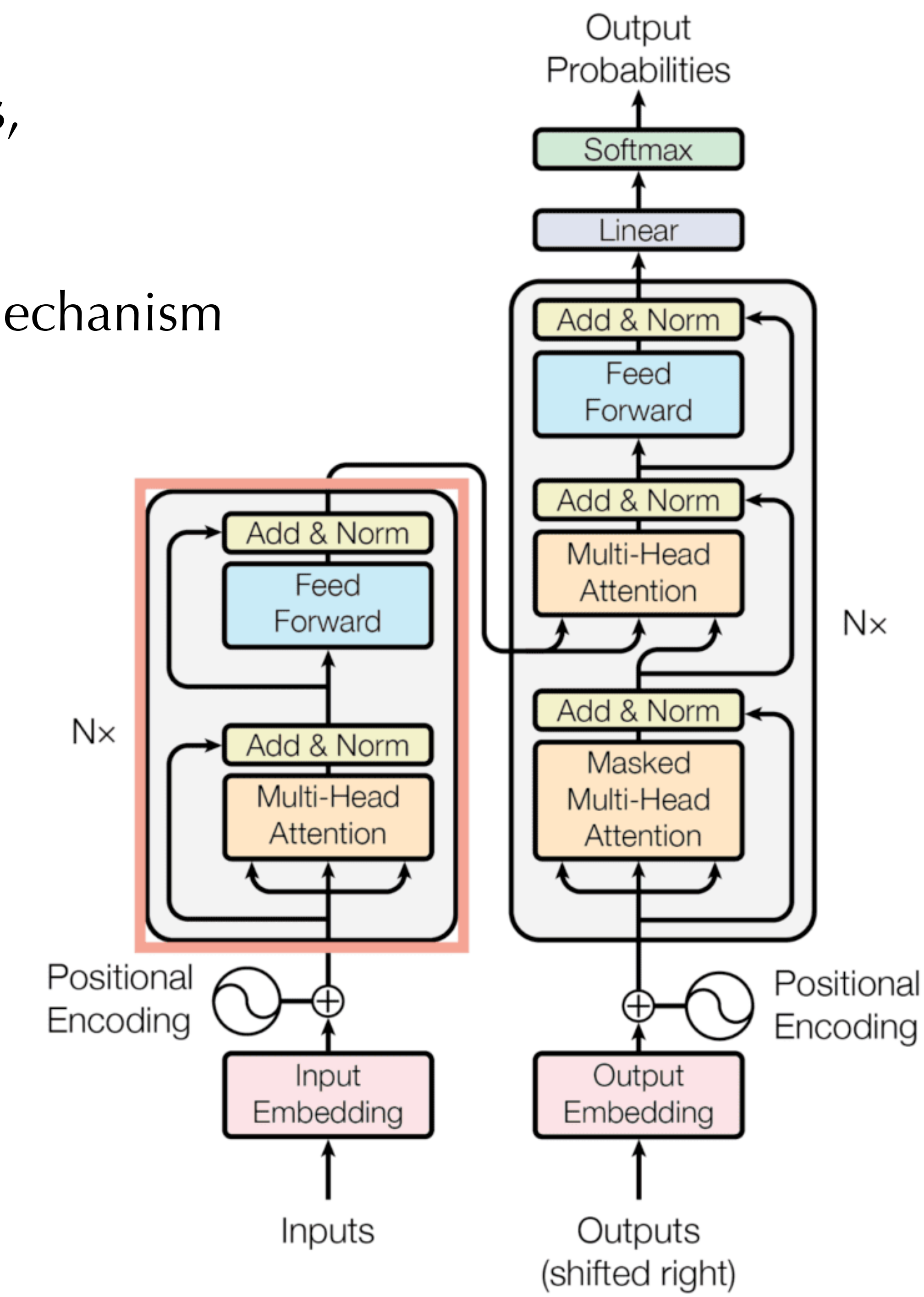
The encoder

- The encoder consists of a stack of $N = 6$ 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



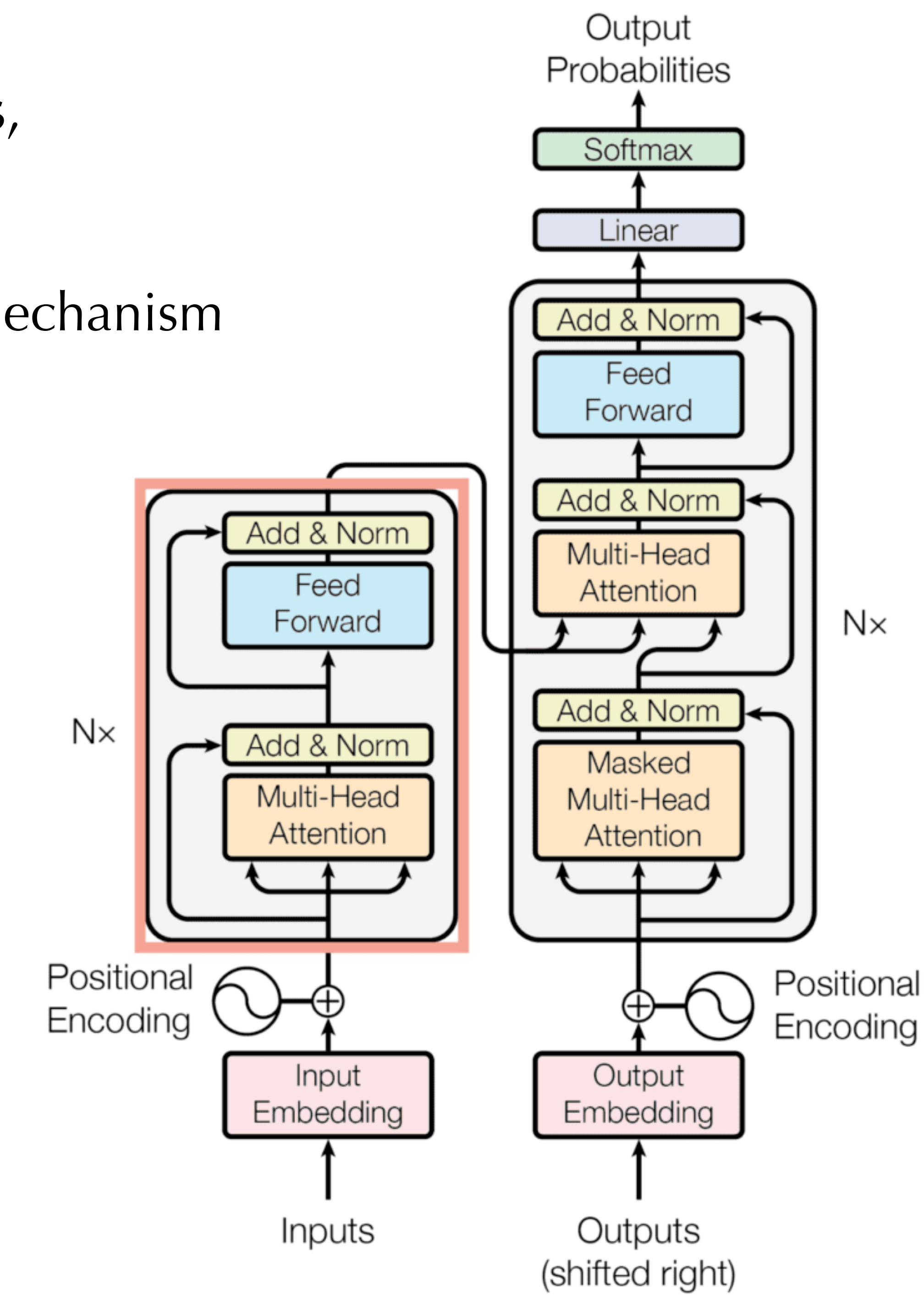
The encoder

- The encoder consists of a stack of $N = 6$ 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



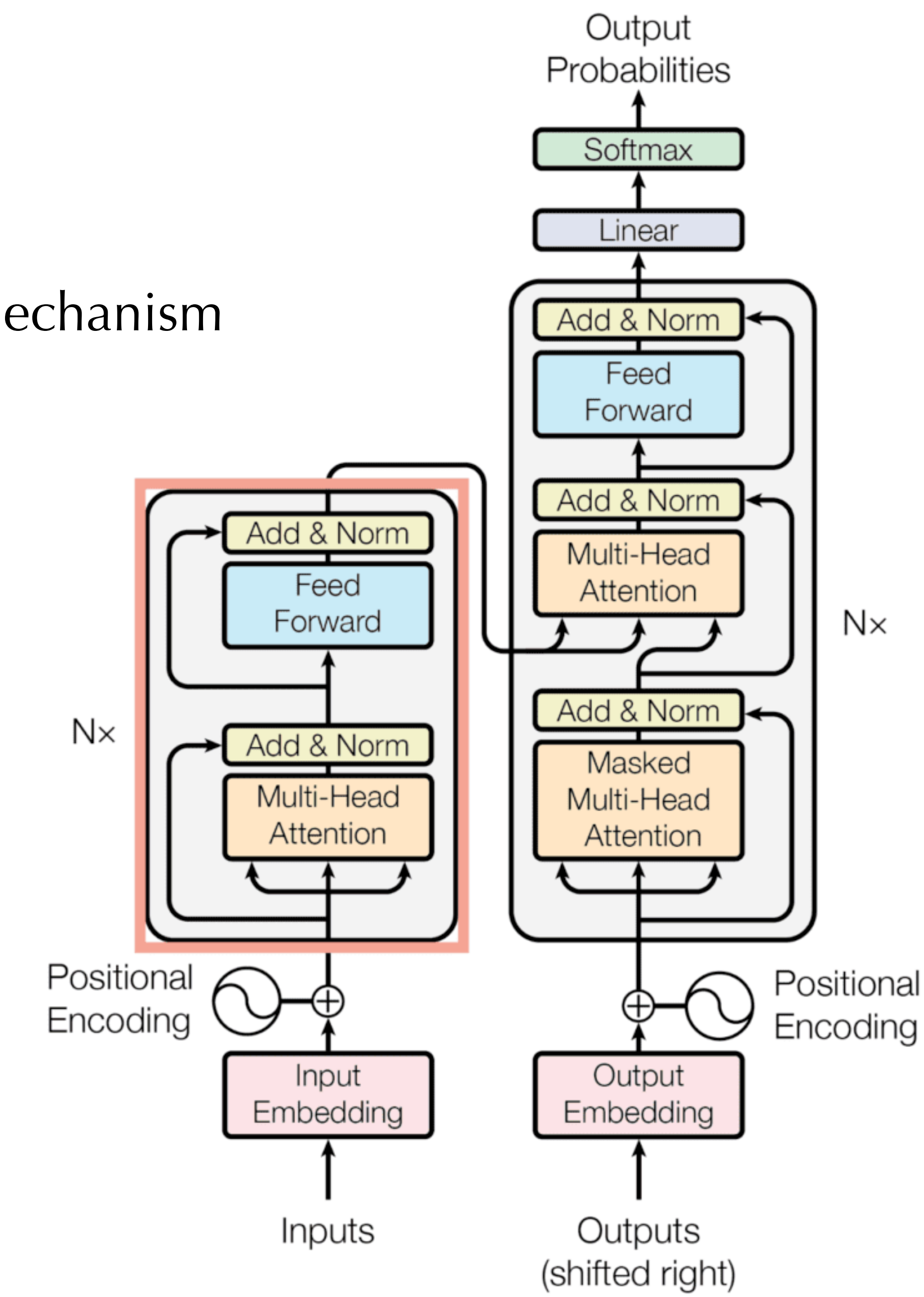
The encoder

- The encoder consists of a stack of $N = 6$ 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



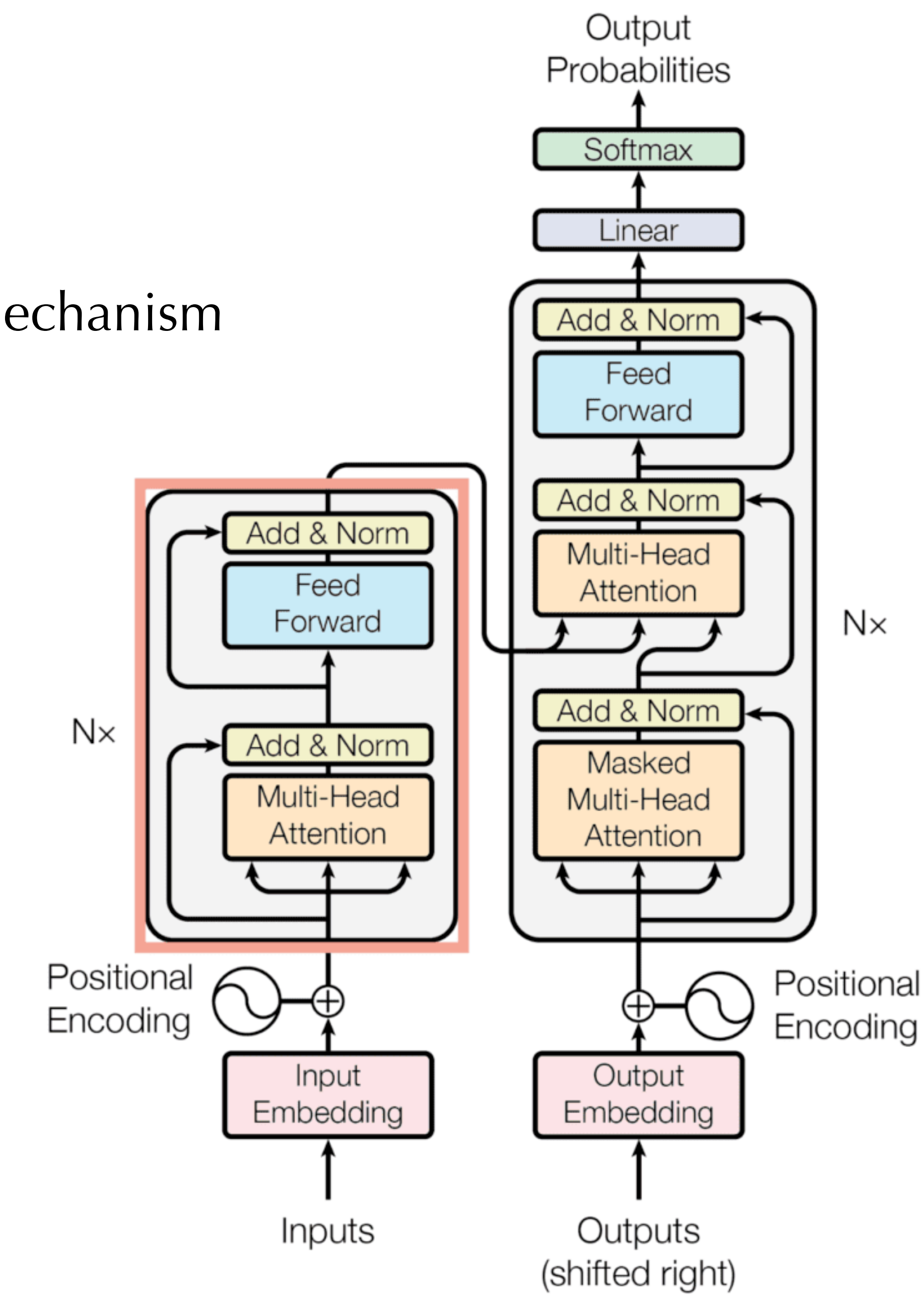
The encoder

- 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

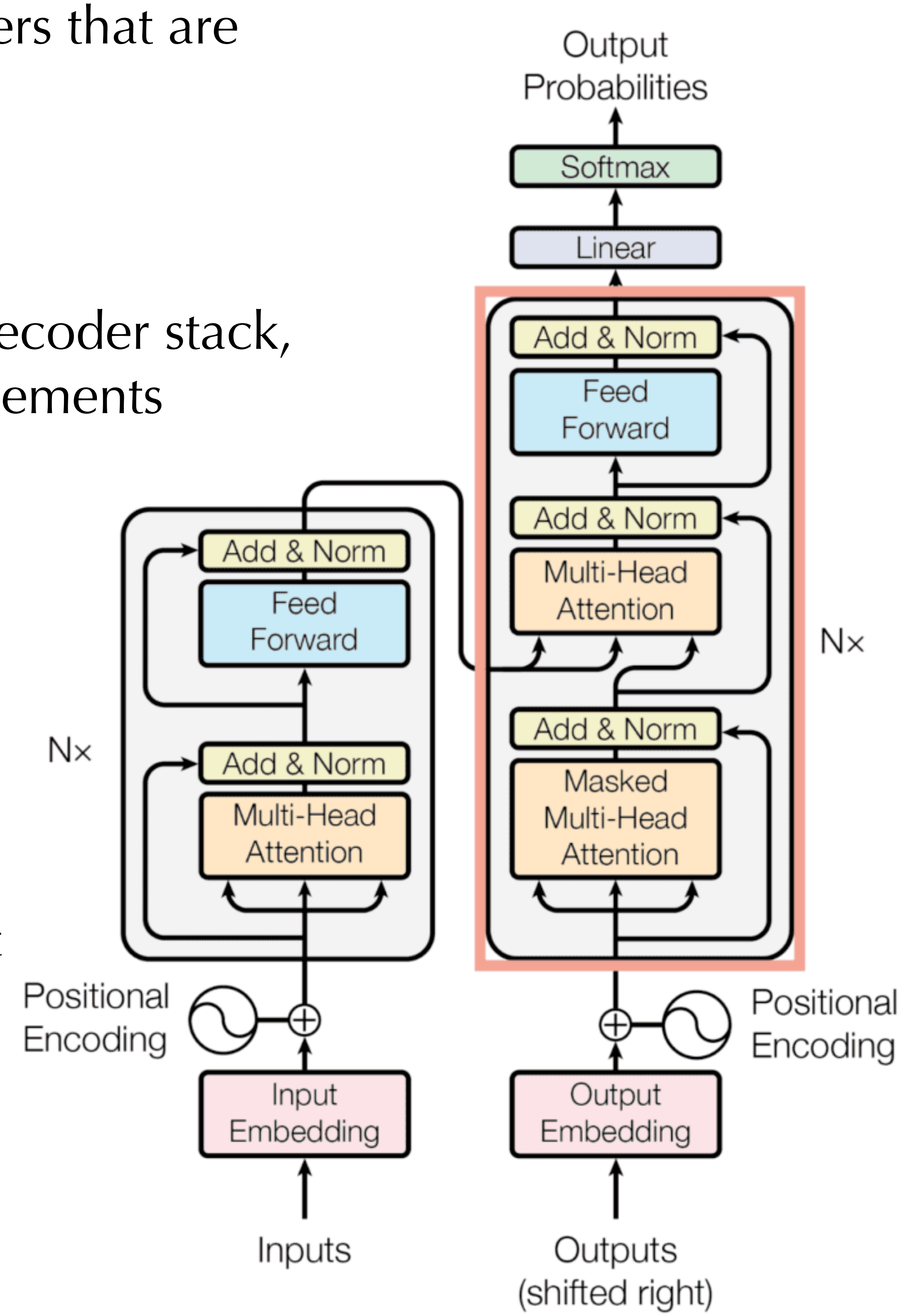
- The decoder also consists of a stack of N identical layers that are each composed of **three sublayers**:

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 → **decoder is sequential**

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



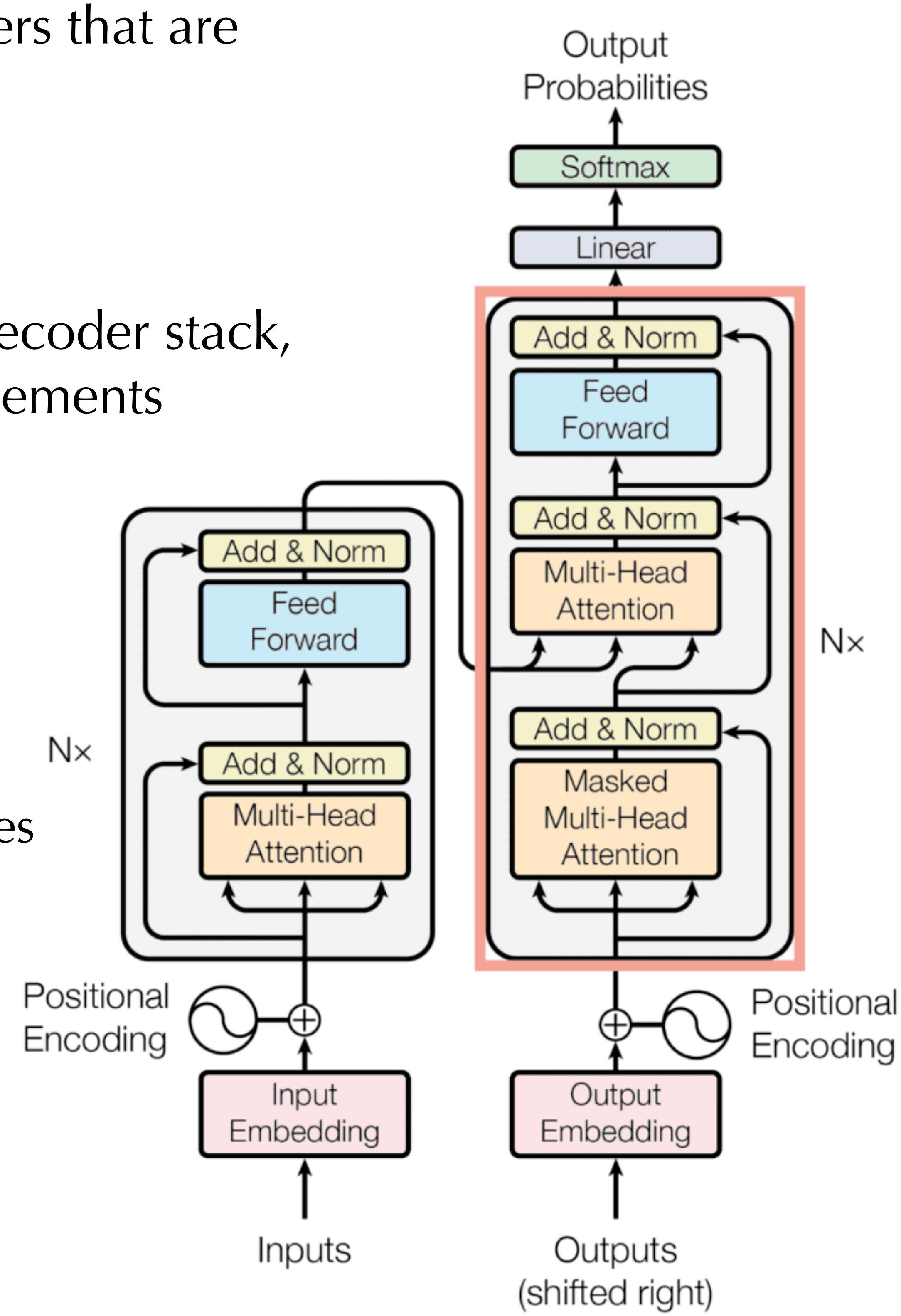
The decoder

- 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

   - 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

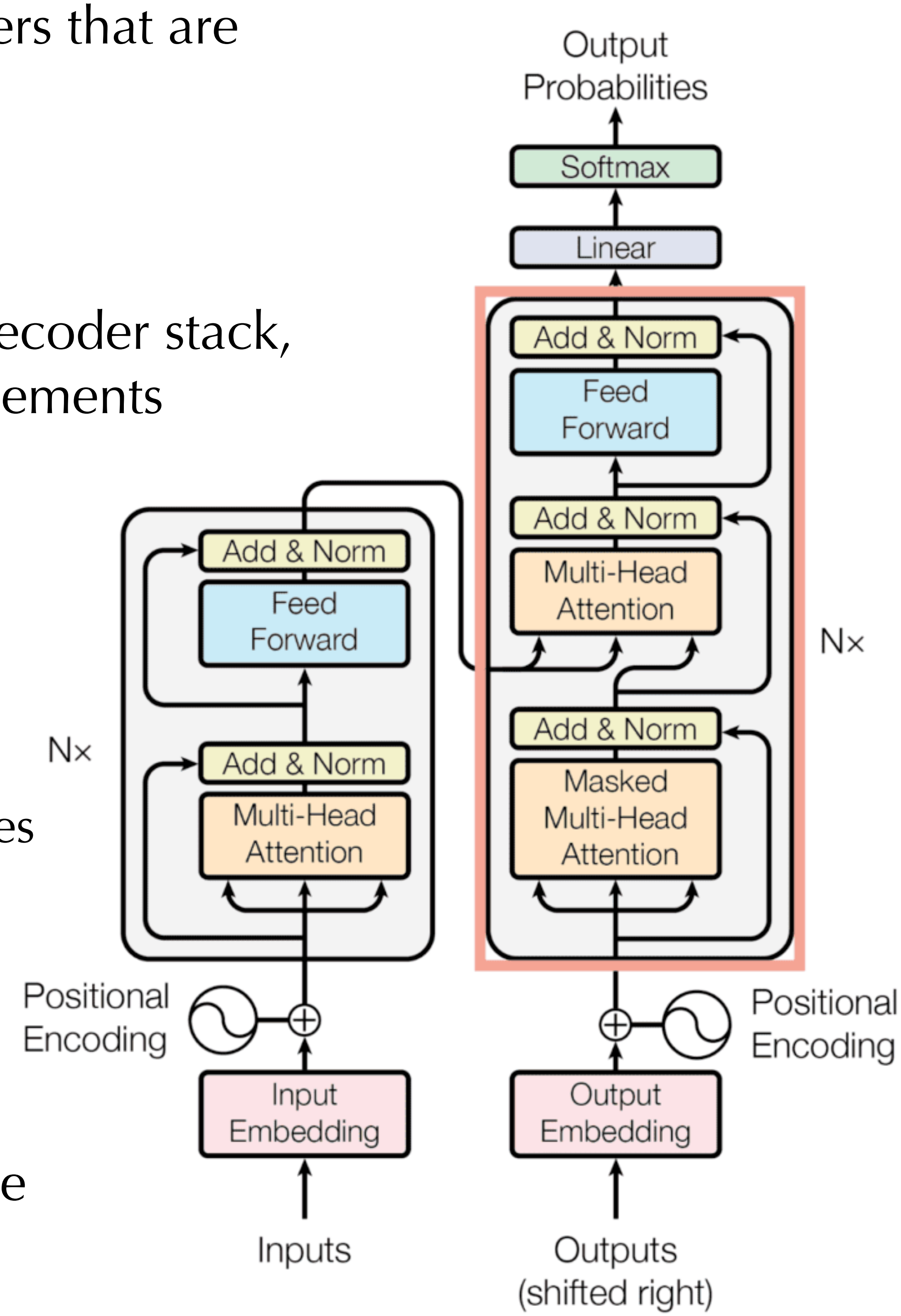
- The decoder also consists of a stack of N identical layers that are each composed of **three sublayers**

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.



The decoder

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

- the three sublayers on the decoder side also have residual connections around them and are

- positional encodings are also added to the input embeddings of the decoder in the same manner as previously explained for the encoder