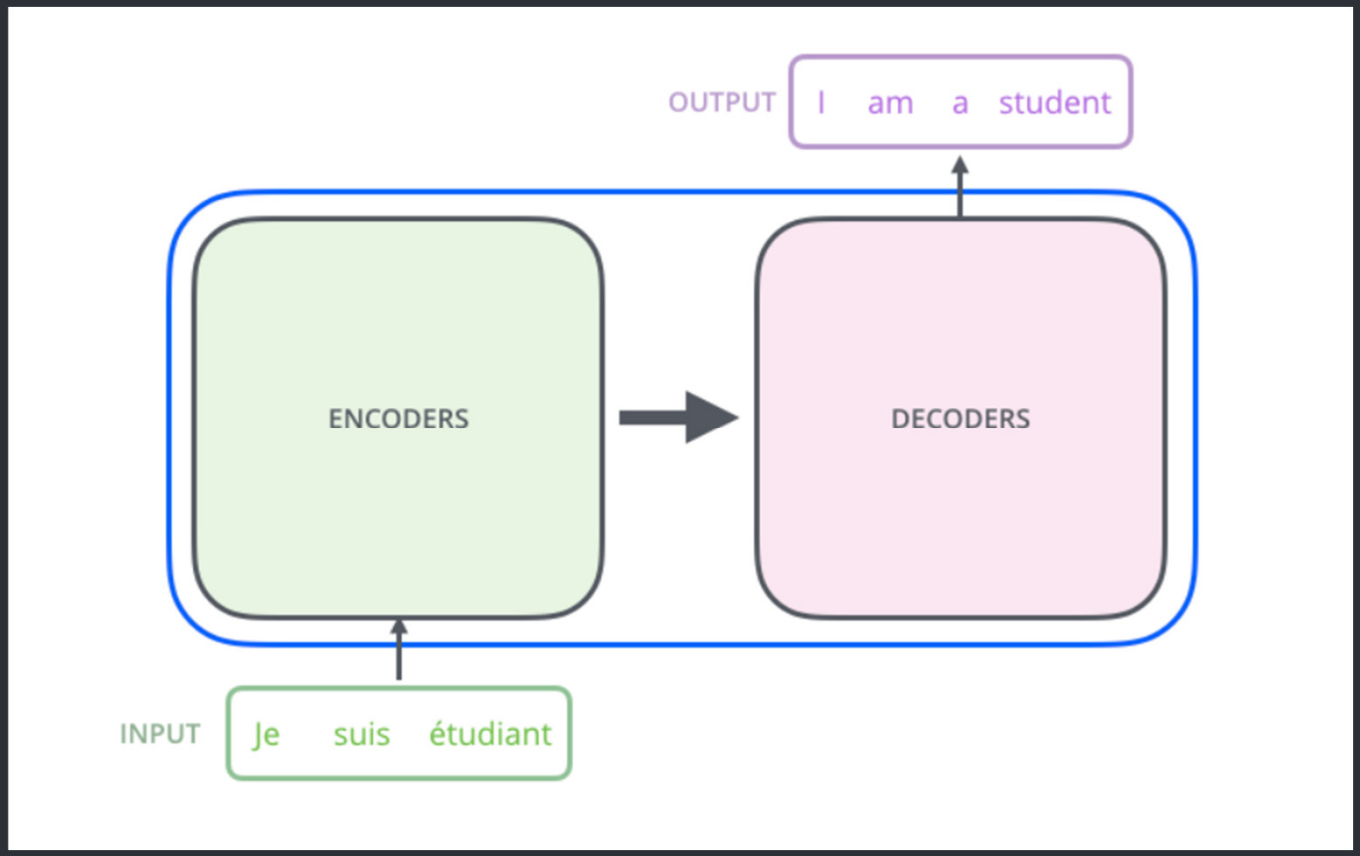
Transformers

Chen Shapira

What is a Transformer?

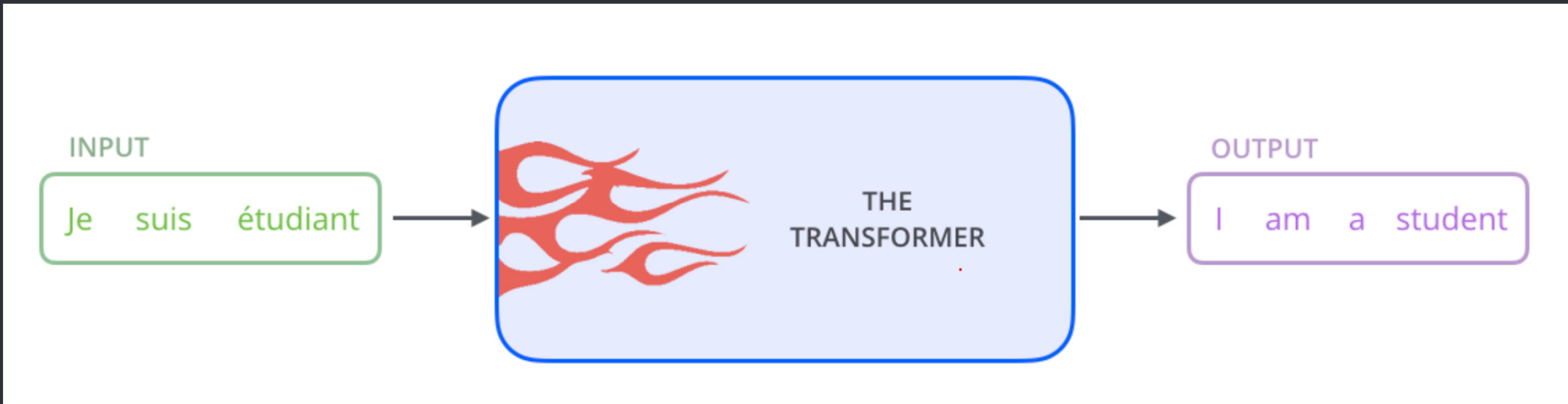
A Transformer is a sequence-to-sequence model, based on an encoder-decoder structure



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

What is a Transformer?

Their original and most common use is Machine Translation. For example, translating a sentence in Portuguese to a sentence in English:



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

What is a Transformer?

Transformers are deep neural networks, they replace CNNs and RNNs with a self-attention mechanism.

What is a Transformer?

Transformers are heavily used today in the domains of natural language processing (NLP) and computer vision (CV).

What is a Transformer?

Transformers were introduced in the paper “Attention is all you need” by Google Brain in 2017.

The Attention mechanism is known from even earlier years.

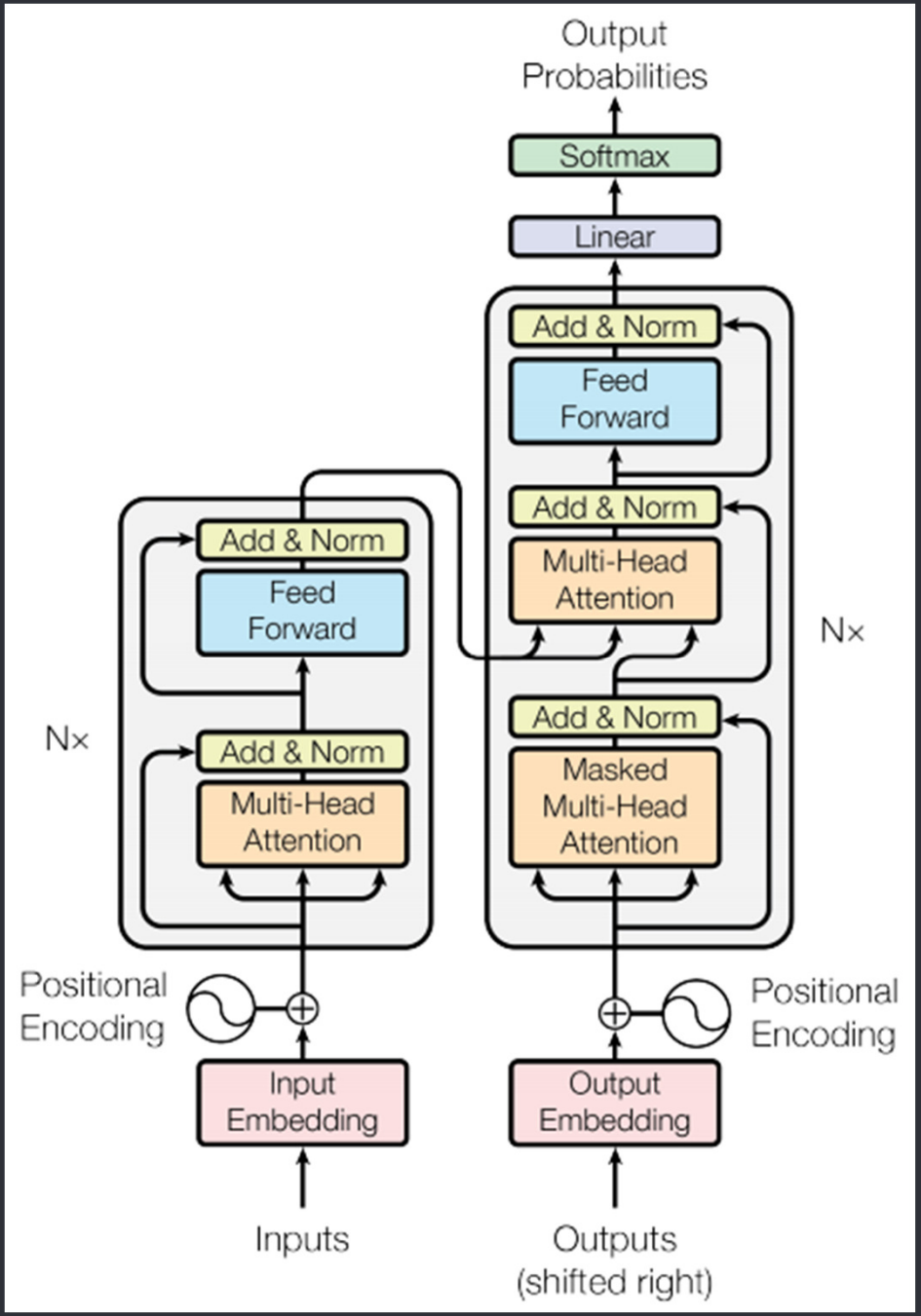
Goals of using a Transformer

Like RNNs, transformers are built *to process sequential data*, such as natural language.

Applications of transformers include language translation and text summarization.

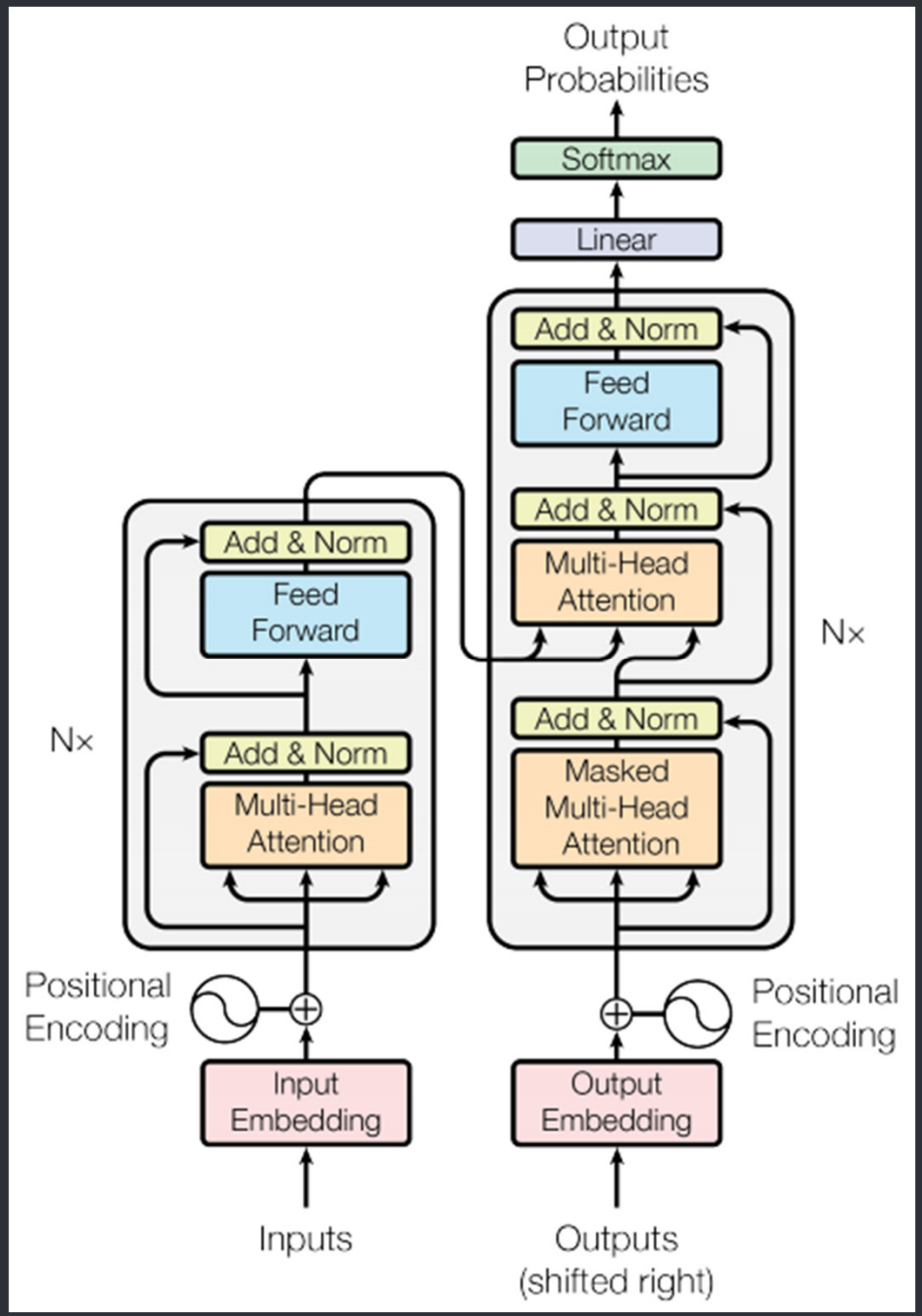
Inputs and Outputs of a Transformer

Inputs: a sequence of elements such as a sentence. For example “This apple is red”



from the “Attention is all you need” paper

Inputs and Outputs of a Transformer



Outputs: a *different* sequence of elements such as a translated sentence. For example

"Dieser Apfel ist rot"

At each inference step, a *single* new word is predicted

Why use a Transformer?

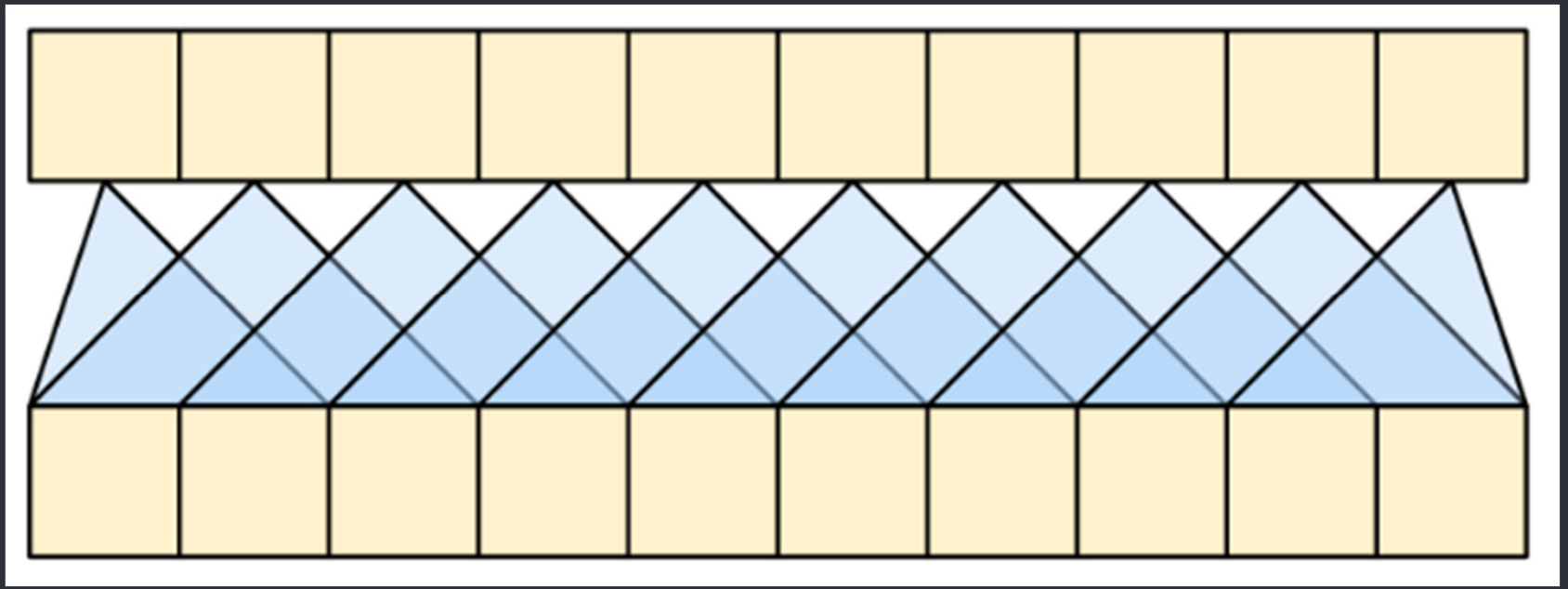
Transformers are very good **at modeling sequential data** such as natural language.

Unlike recurrent neural networks (RNNs), Transformers can be **parallelized**. This makes them much more efficient on GPUs.

Also, unlike RNNs or CNNs, Transformers are able to capture **long-range dependenci**es in the data.

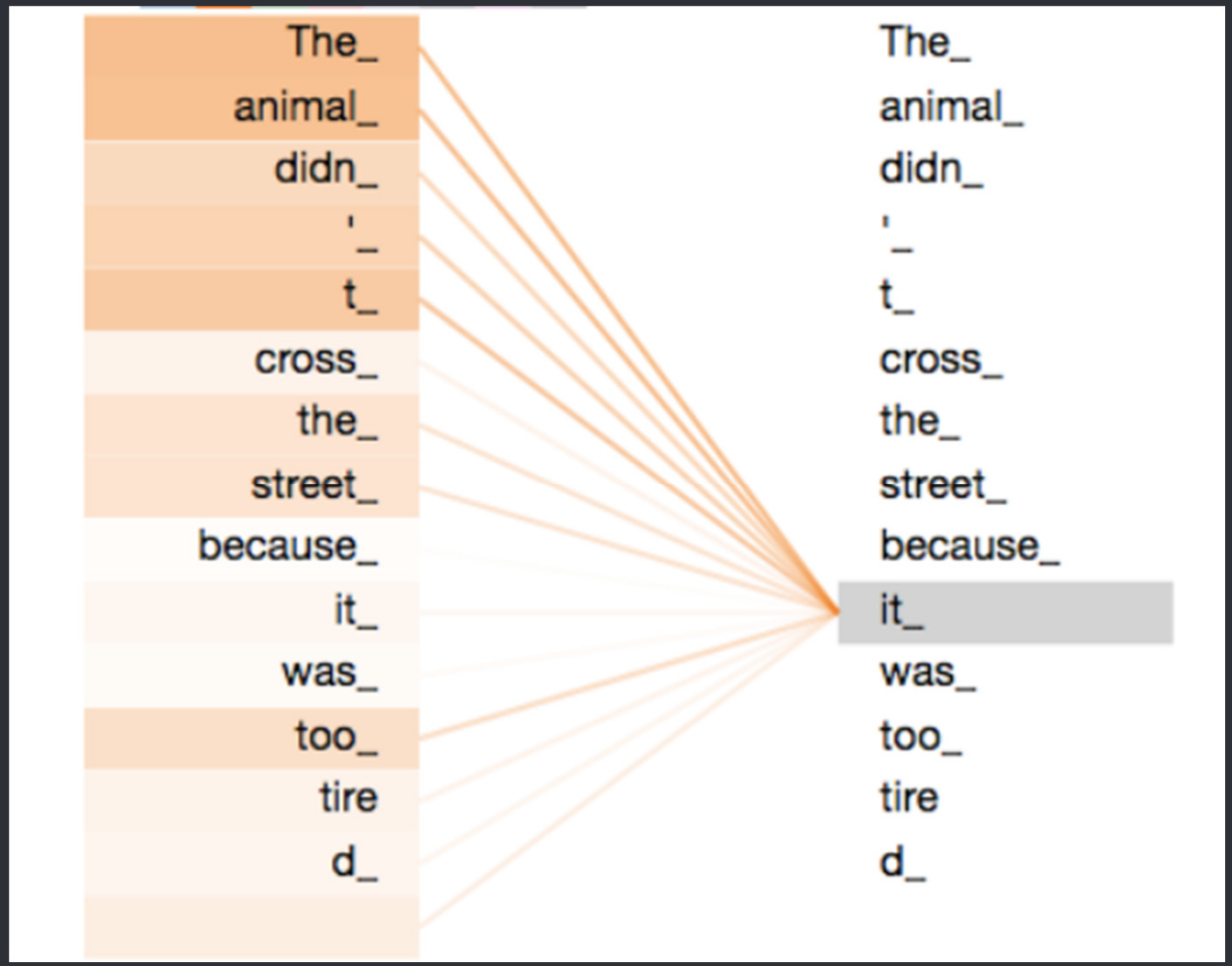
Why use Transformers?

The Transformer also analyzes the relations between *all the input's elements (tokens) to each other.*



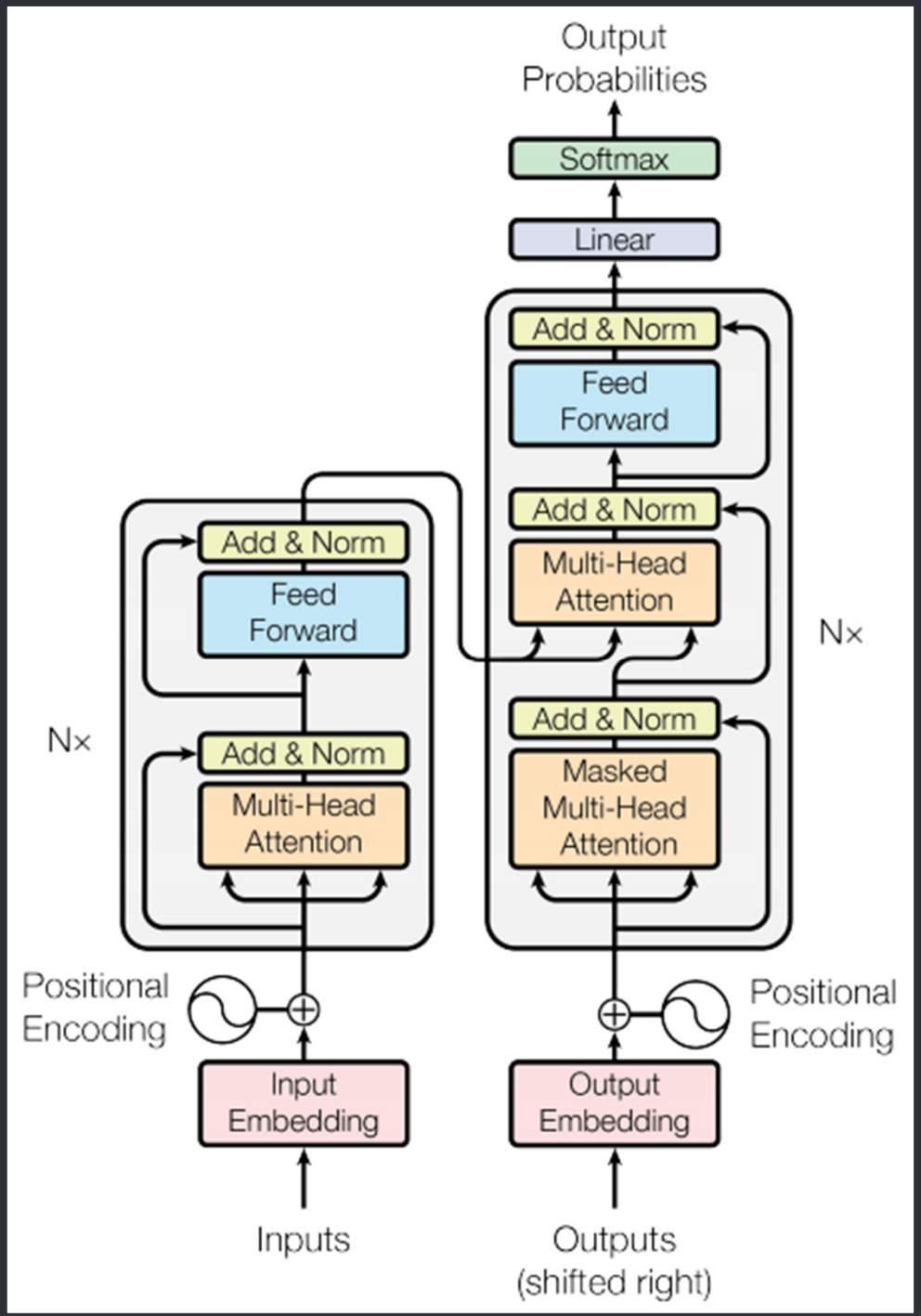
Self-Attention in the Transformer

Self-attention is an important component of Transformers. In high-level, it relates different positions of a single sequence, in order to compute a representation of that sequence:



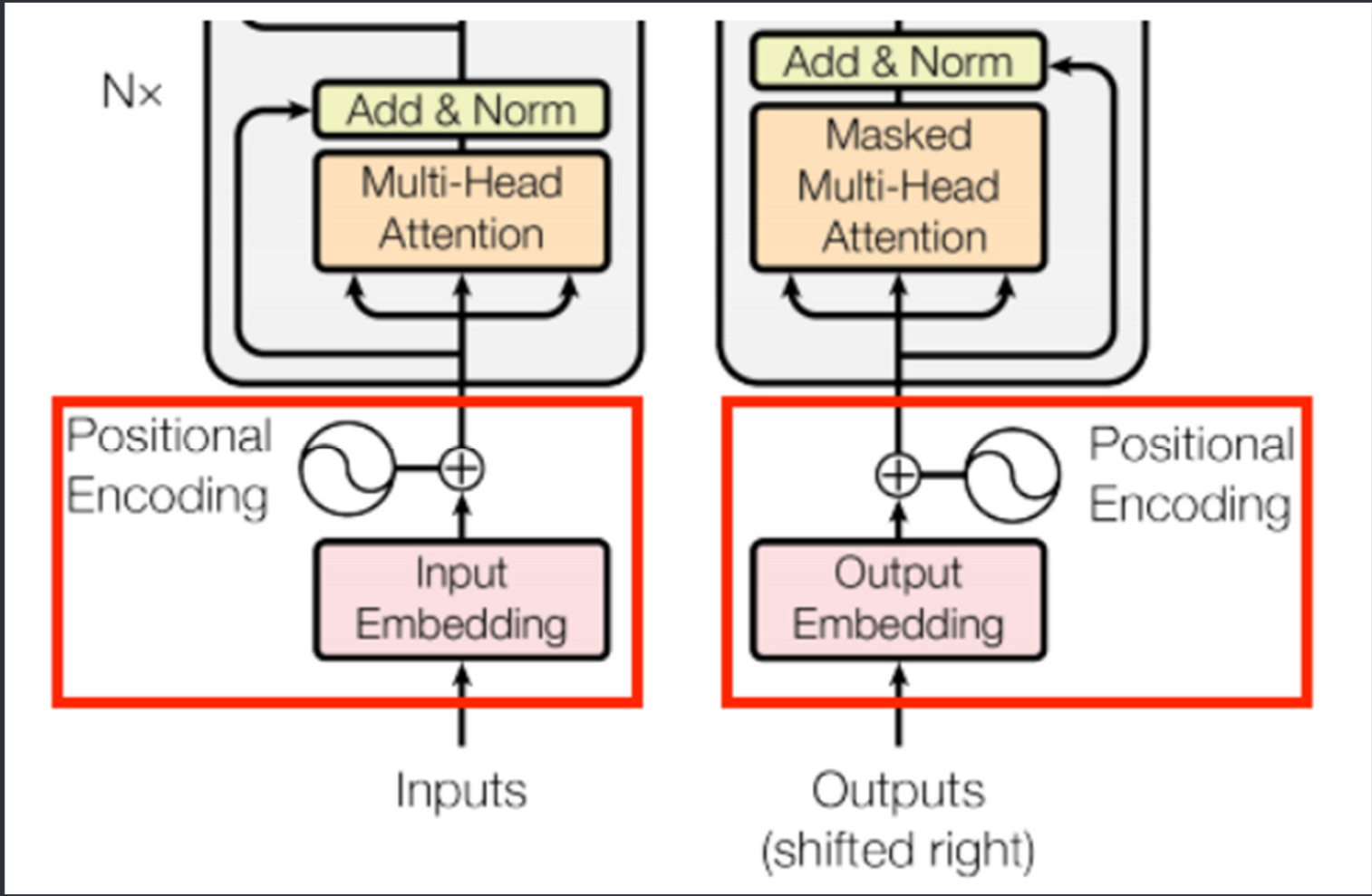
https://jalammar.github.io/illustrated-transformer/

Components of a Transformer



- Embedder
- Positional Encoder
- Encoder:
  - Attention Layer
  - Feed-Forward Layer
  - Add & Norm Layer
- Decoder:
  - Attention Layer
  - Feed-Forward Layer
  - Add & Norm Layer
- Linear & Softmax Output Layer

Embeddings



from www.tensorflow.org

What is an Embedding?

An embedding is a low-dimensional representation which can be used to describe different objects or elements.

For example, if you have 5 words in your vocabulary, we can create an embedding for it in two steps:

“Green” -> [1,0,0,0,0] -> [0.3, 1.5].

What is an Embedding?

“Green” -> [1,0,0,0,0] -> [0.3, 1.5].

The original 5-dimensional representation was used to distinguish each of the 5 words in the vocabulary in a simple way:

[1,0,0,0,0], [0,1,0,0,0], …, [0,0,0,0,1].

The vector [0.3, 1.5] is the Embedding of the first word, “Green”, and it is **2-dimensional** and more compact. We can convert each of the 5 words into a different 2-dimensional vector.

Embedder

An Embedder converts positive **integers** to dense **vectors** of fixed size:

Word strings -> tokenized to integers -> encoded to embeddings.

[“Green”, “Apple”] -> [4, 20] ->
[(0.25,0.35,0.12), (0.22,0.34,0.13)]

Why use Embeddings?

An embedding can capture some of the semantics of the input, by placing *semantically similar* inputs close together in the embedding-space.

An embedding can be learned and reused across models.

Why use Embeddings?

Example of representing semantically-similar words, as ‘similar vectors’:

‘Apple’ -> [1.0, 2.0] , ‘Red’ -> [1.1, 2.3] ,

‘Table’ -> [-1.0, 1.2]

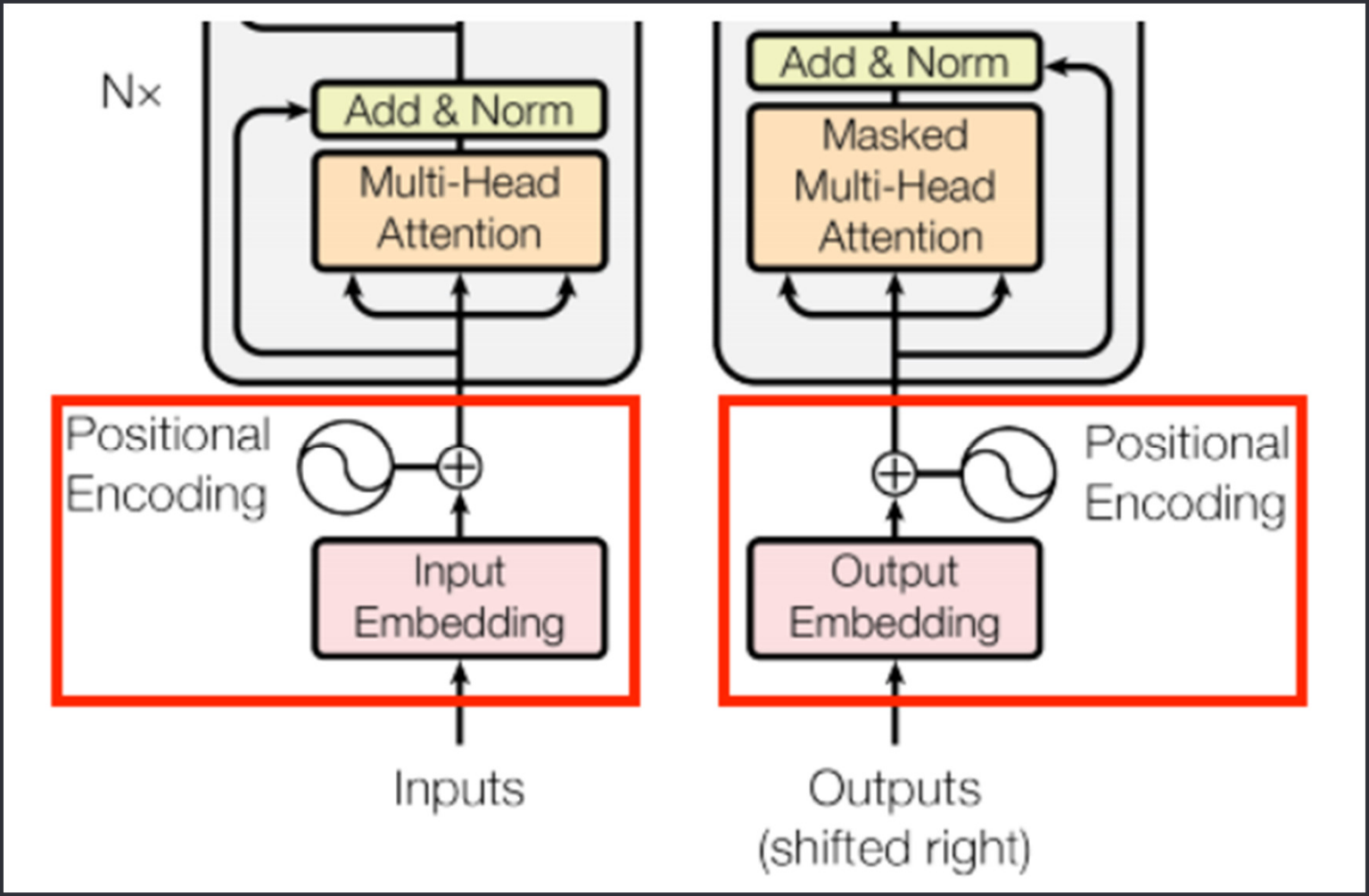
We use the dot-product to measure the similarity between numerical vectors.

The dot-product between the embeddings of ‘Apple’ and ‘Red’ would be larger compared to the dot-product between ‘Apple’ and ‘Table’.

Why use Embeddings?

Instead of specifying the weights of the embedder layer manually, they are trainable and learned by the model during training, in the same way a model learns weights for a dense layer.

Positional Encoding



What is Positional Encoding?

Positional encoding describes the location or position of an element in a sequence, such that *each position is assigned a unique vector representation*.

It is better than "using a single number to represent a position", because even for large position indexes, the sine and cosine functions have values in a **normalized range** [-1, 1].

This **keeps the values of the positional encoding in a limited range**, which in general improves ML training (same as in Batch Normalization).

What is Positional Encoding?

Transformers use a smart positional encoding scheme, where each position/index is mapped to a different vector:

Sentence: [“Green”, “Apple”] ->

Positions Indexes: [1, 2] ->

Positional Encodings: [(0.1, 0.2, 0.3, 0.2, 0.1),
(0.1, 0.4, 0.1, 0.4, 0.1)]

Why use Positional Encoding?

The attention layers used throughout the model see their input as a set of vectors, with *no order*.

the order is *encoded inside* the vector using a *positional-encoding*.

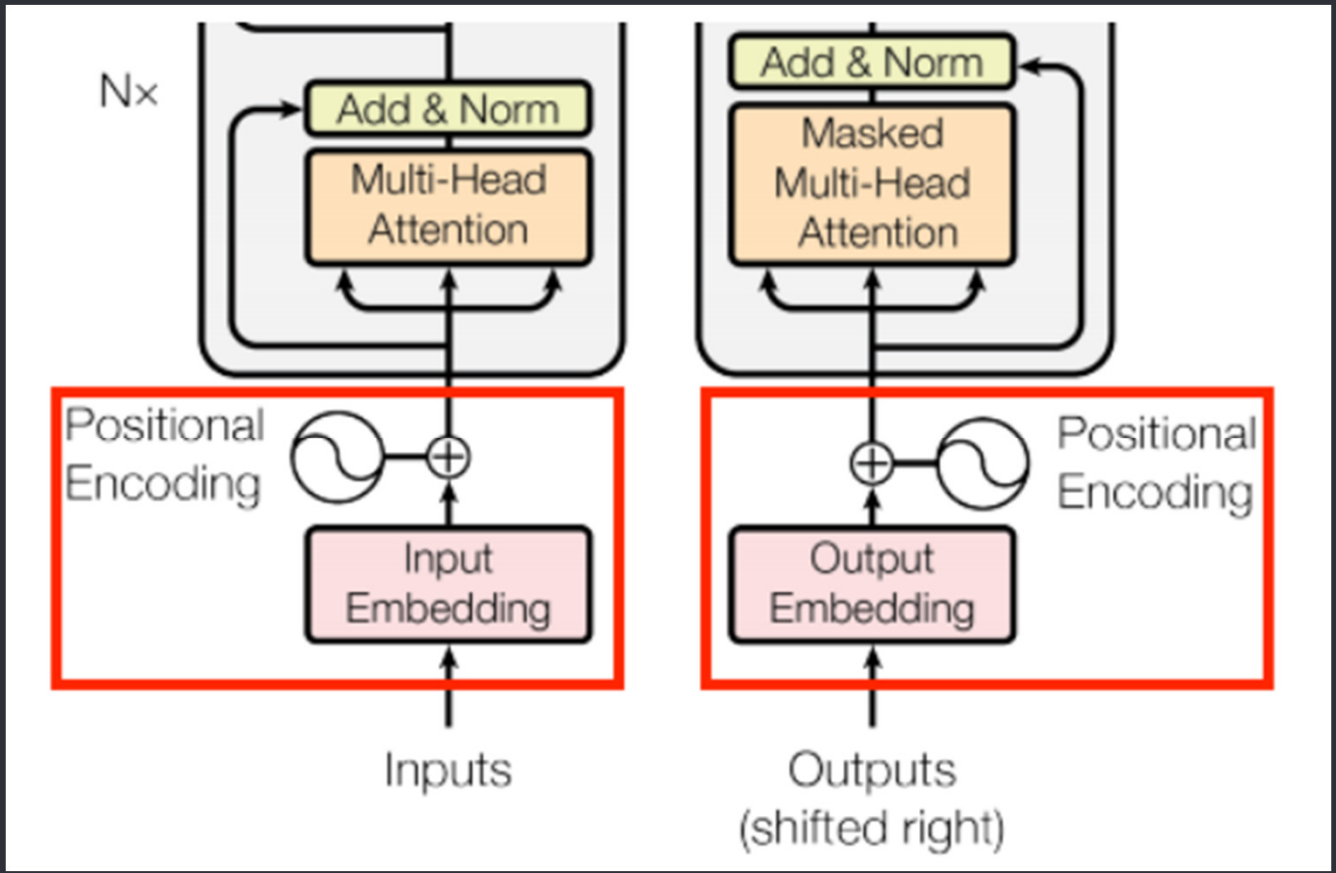
Since the model doesn't contain any recurrent or convolutional layers. It needs some way to identify word order, otherwise it would see the input sequence as a bag of words instance:

"how are you", "how you are", "you how are", are indistinguishable.

Why use Positional Encoding?

The Positional-Encoding has the same dimension as the input-embedding.

It is **summed** together with the embedding and passed to the next layer:



How Positional Encoding Works?

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

- *pos* is the word's position in the sentence.
- *d-model* is a constant representing the number of features in the embedding.
- *i* is the current feature id.

Note that the PE is a **matrix**.

How Positional Encoding Works?

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

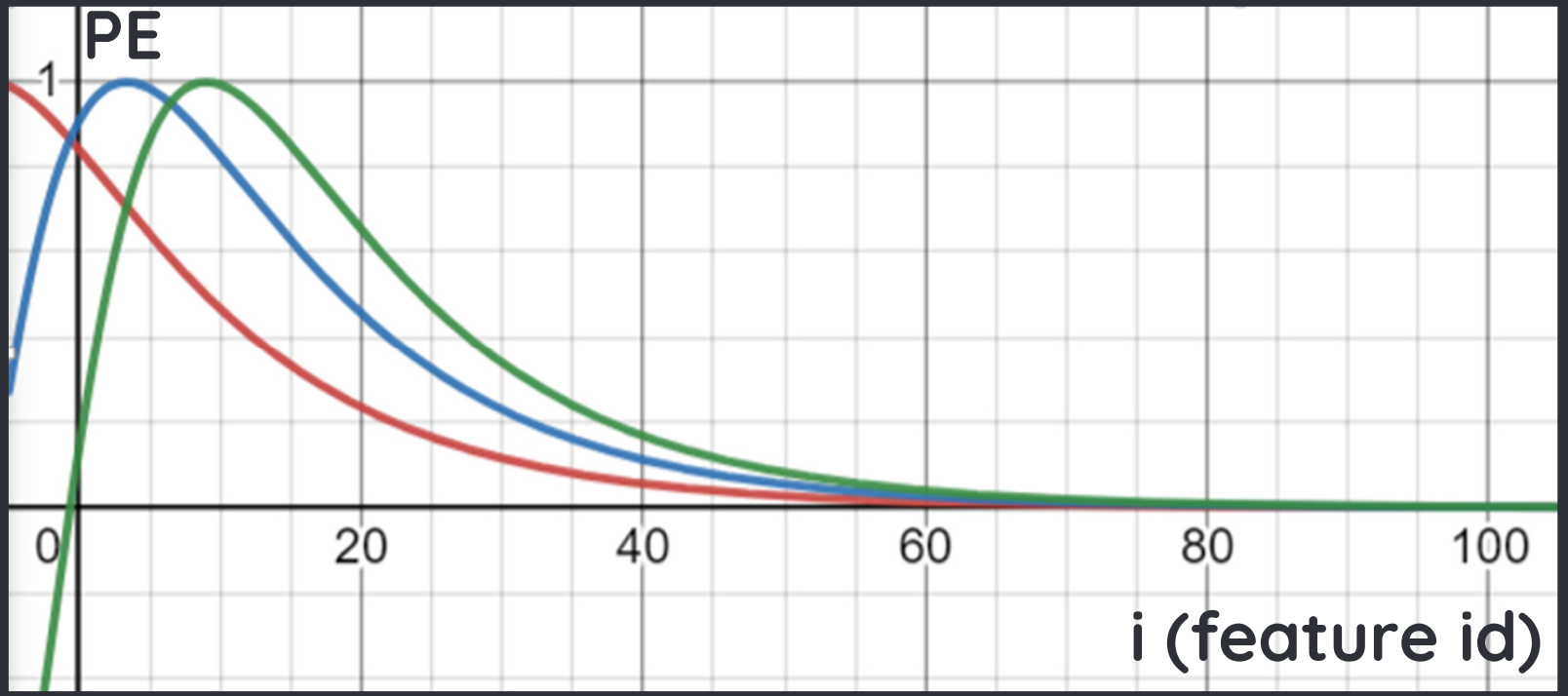
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Examples of positional encodings for **nearby** words:

pos=1

pos=2

pos=3



How Positional Encoding Works?

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

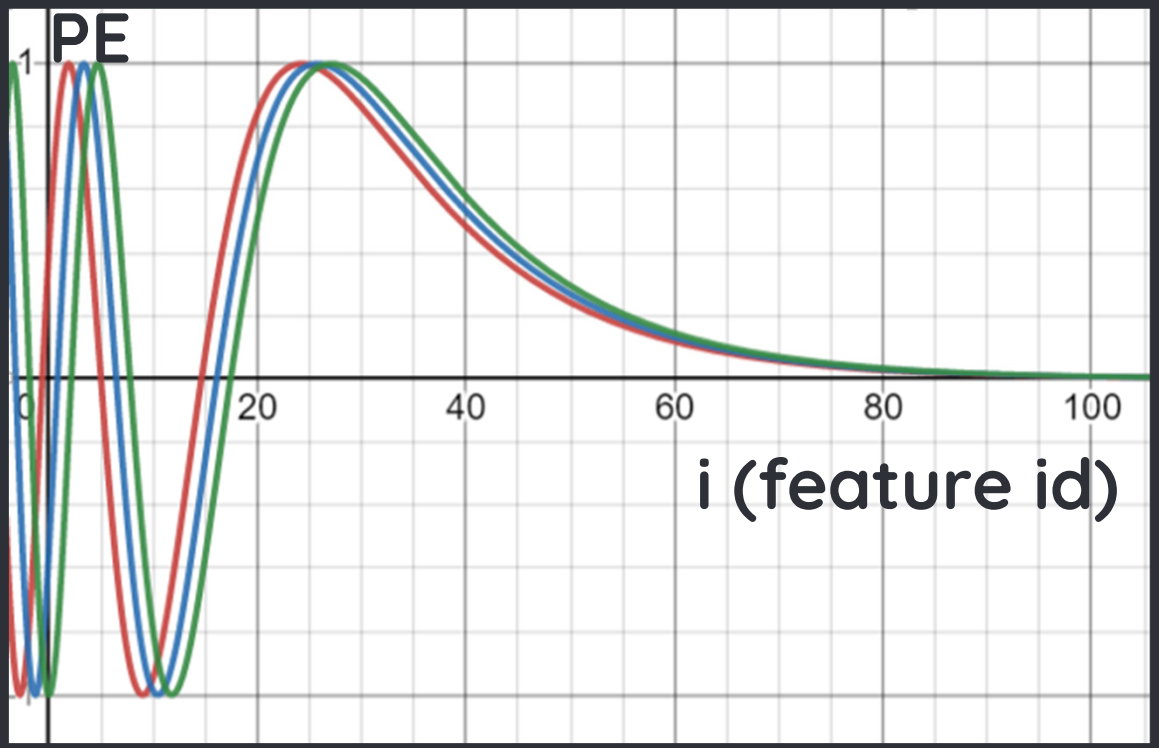
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Examples of positional encodings for **nearby** words:

pos=9

pos=10

pos=11



How Positional Encoding Works?

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

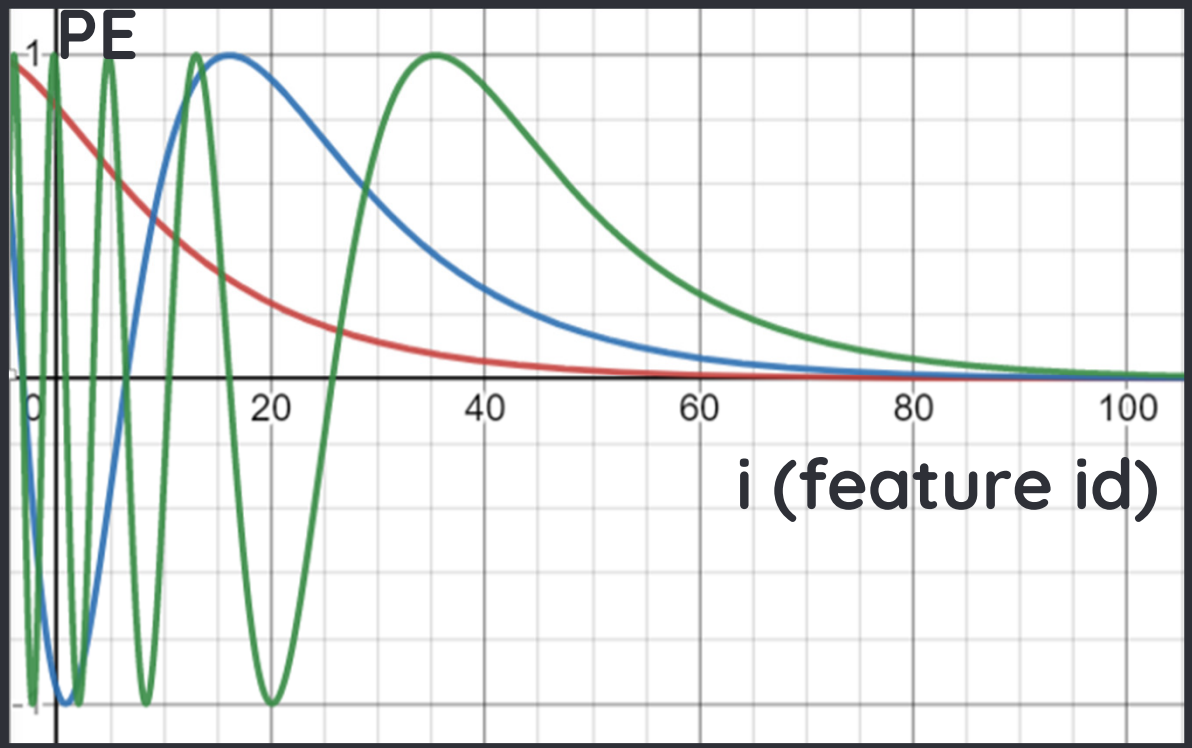
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Examples of positional encodings for **distant** words:

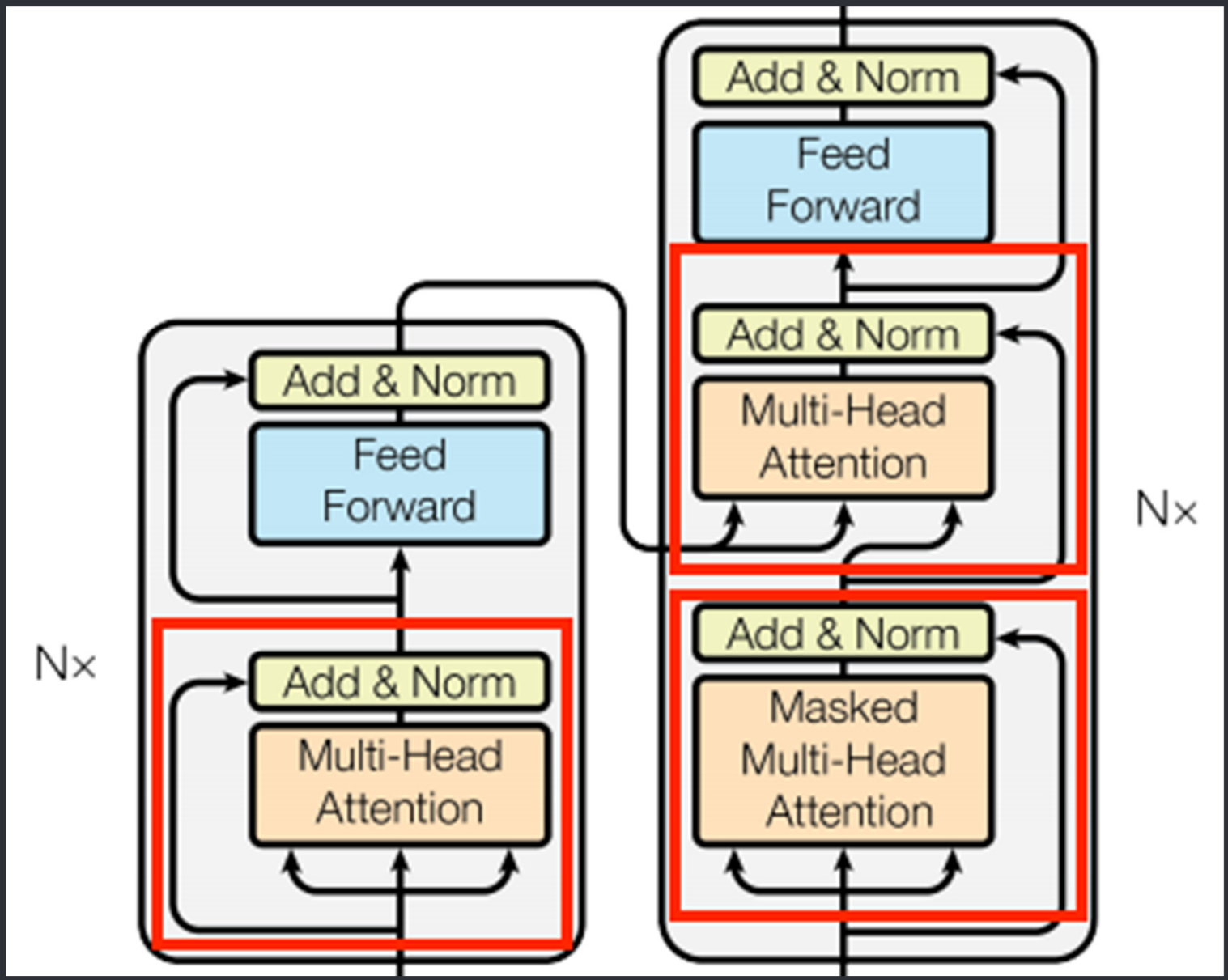
pos=1

pos=5

pos=20



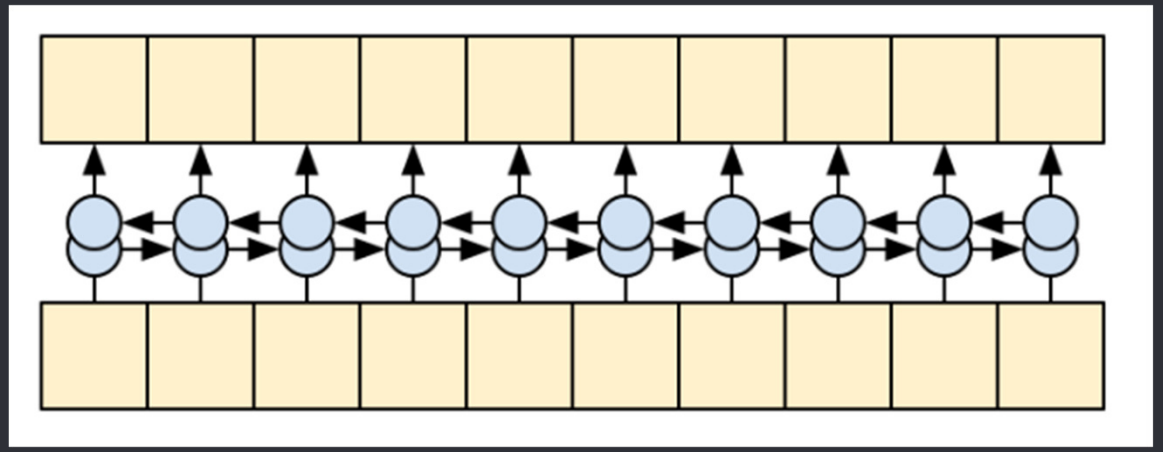
Base Attention Block



Why use Base Attention?

Unlike RNNs or CNNs, Transformers use Base Attention to capture ***distant*** or long-range dependencies in the data, between distant positions in the input sequences.

Thus, longer “connections” can be learned. In contrast, an RNN can more easily “forget” distant elements:



What is Base Attention?

"The Base Attention mechanism is similar to a dictionary lookup - A ***fuzzy***, *differentiable, vectorized* dictionary lookup".

To illustrate this, let's look at a standard dictionary structure:

d = {'color': 'blue', 'age': 22, 'type': 'pickup'}

We'll call the item that we search a "**query**".

What is Base Attention?

d = {'color': ‘red', 'age': 27, 'type': ‘truck'}

When the query is the word: ‘type’, the dictionary will return d[‘type’] = ‘truck’

‘truck’ would be called **value** in this case, and ‘type’ would also be its matching **key.**

What is Base Attention?

d = {'color': ‘red', 'age': 27, 'type': ‘truck'}

d[‘type’] = ‘truck’

The important point is that here, the query either has a matching key or it doesn't.

However, a *fuzzy* dictionary is one where the keys ***don't have to match perfectly***.

What is Base Attention?

Example of a *fuzzy dictionary:*

d = {'color': ‘red', 'age': 27, 'type': ‘truck'}

d[‘***species***’] = ‘truck’

The word *species* is different than the word *type*, but for a **fuzzy** dictionary, their **similarity** is **close enough**, and **it’ll return the matching value** (‘truck’).

What is Base Attention?

A **Base Attention Layer** performs a ***fuzzy*** lookup, but it's not just looking for the *best* key, it **combines** the values based on **how well the query matches each key.**

For example:

d = {'table': 1, 'apple': 20, 'grapes': 22}

d['food'] =

(similarity(food, table) * 1 +

similarity(food, apple) * 20 +

similarity(food, grapes) * 22)

What is Base Attention?

d = {‘table’: 1, ‘apple’: 20, ‘grapes’: 22}

d[‘food’] =
(similarity(food, table) * 1 +
similarity(food, apple) * 20 +
similarity(food, grapes) * 22)
= (0.05 * 1 + 0.45 * 20 + 0.45 * 22) =
18.95

The above operation is called a **weighted-sum**.

What is Base Attention?

d = {‘table’: 1, ‘apple’: 20, ‘grapes’: 22}

d[‘food’] = 18.95

We get that the value of d[‘food’] is 18.95, which is **closer (more similar)** to the values of ‘apple’ (20) and ‘grapes’ (22), than to the value of ‘table’ (1).

This is how *fuzzy* lookup works, and how the *Base Attention works.*

Base Attention Block

The Base Attention Block performs a similar operation to the dictionary we just described.

However here we replace with dictionary with the **Keys and Values matrices** (K and V).

The queries from before, would be replaced by the **Queries matrix** (Q).

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

Base Attention Block

Each line in Q represents a **separate query.**

Each line in K represents a **separate key.**

Each line in v represents a **separate value.**

For to the *key in row "i" of K*, we find the corresponding *value in row "i" of V.*

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

What is Base Attention?

The operations performed by the Attention function are:

1. Finding the correlation between each query and all the keys in K ***using dot-products.***

2. Creating a new representation *for the query,* based on a **weighted-sum** of the *values.*

(the weighted-sum is calculated using the matrix multiplication and will be shown in next slides)

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

Details of Base Attention

The similarity-scores s(i,j) in the third line, are calculated using vector multiplication between Q's rows and V's columns:

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

$$softmax\left(\frac{1}{sqrt(d_k)}\begin{pmatrix} - & q_1 & - \\ & \vdots & \\ - & q_n & - \end{pmatrix}\begin{pmatrix} | & & | \\ k_1 & \cdots & k_m \\ | & & | \end{pmatrix}\right)\begin{pmatrix} - & v_1 & - \\ & \vdots & \\ - & v_m & - \end{pmatrix} =$$

$$= \begin{pmatrix} s_{1,1} & \cdots & s_{1,m} \\ \vdots & & \vdots \\ s_{n,1} & \cdots & s_{n,m} \end{pmatrix}\begin{pmatrix} - & v_1 & - \\ & \vdots & \\ - & v_m & - \end{pmatrix} = \begin{pmatrix} s_1 \cdot v^1 & \cdots & s_1 \cdot v^{d_v} \\ \vdots & & \vdots \\ s_n \cdot v^1 & \cdots & s_n \cdot v^{d_v} \end{pmatrix}$$

Details of Base Attention

I've marked the similarity between query-i and key-j as s(i,j):

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

$$softmax\left(\frac{1}{sqrt(d_k)}\begin{pmatrix} - & q_1 & - \\ & \vdots & \\ - & q_n & - \end{pmatrix}\begin{pmatrix} | & & | \\ k_1 & \cdots & k_m \\ | & & | \end{pmatrix}\right)\begin{pmatrix} - & v_1 & - \\ & \vdots & \\ - & v_m & - \end{pmatrix} =$$

$$= \begin{pmatrix} s_{1,1} & \cdots & s_{1,m} \\ \vdots & & \vdots \\ s_{n,1} & \cdots & s_{n,m} \end{pmatrix}\begin{pmatrix} - & v_1 & - \\ & \vdots & \\ - & v_m & - \end{pmatrix} = \begin{pmatrix} s_1 \cdot v^1 & \cdots & s_1 \cdot v^{d_v} \\ \vdots & & \vdots \\ s_n \cdot v^1 & \cdots & s_n \cdot v^{d_v} \end{pmatrix}$$

Details of Base Attention

I'll use an example to clarify:

$$Assume\ for\ example: \begin{pmatrix} s_{1,1} & \cdots & s_{1,m} \\ \vdots & & \vdots \\ s_{n,1} & \cdots & s_{n,m} \end{pmatrix} \begin{pmatrix} - & v_1 & - \\ & \vdots & \\ - & v_m & - \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix},$$

$$then:$$

$$Attention(Q, K, V) = \begin{pmatrix} 2 & 2 & 2 \\ 1 & 1 & 1 \\ 3 & 3 & 3 \end{pmatrix}$$

We see that every row of the Attention's output, is a weighted-mean of **all the values.** The weighting is done using the **similarity scores** from S.

Details of Base Attention

I'll use an example to clarify:

$$\textit{Assume for example}: \begin{pmatrix} s_{1,1} & \cdots & s_{1,m} \\ \vdots & & \vdots \\ s_{n,1} & \cdots & s_{n,m} \end{pmatrix} \begin{pmatrix} - & v_1 & - \\ & \vdots & \\ - & v_m & - \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix},$$

$$\textit{then}:$$

$$\textit{Attention}(Q, K, V) = \begin{pmatrix} 2 & 2 & 2 \\ 1 & 1 & 1 \\ 3 & 3 & 3 \end{pmatrix}$$

Every row of the Attention's output represents "a combination of values, based on the similarity of the values' keys to the query".

Details of Base Attention

I'll use an example to clarify:

$$\textit{Assume for example}: \begin{pmatrix} s_{1,1} & \cdots & s_{1,m} \\ \vdots & & \vdots \\ s_{n,1} & \cdots & s_{n,m} \end{pmatrix} \begin{pmatrix} - & v_1 & - \\ & \vdots & \\ - & v_m & - \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix},$$

$$\textit{then}:$$

$$\textit{Attention}(Q, K, V) = \begin{pmatrix} 2 & 2 & 2 \\ 1 & 1 & 1 \\ 3 & 3 & 3 \end{pmatrix}$$

The first line in S refers to the first query in Q. We see that the key which is the "most similar" to this query is the second key.

Details of Base Attention

I'll use an example to clarify:

$$\textit{Assume for example}: \begin{pmatrix} s_{1,1} & \cdots & s_{1,m} \\ \vdots & & \vdots \\ s_{n,1} & \cdots & s_{n,m} \end{pmatrix} \begin{pmatrix} - & v_1 & - \\ & \vdots & \\ - & v_m & - \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix},$$

$$\textit{then}:$$

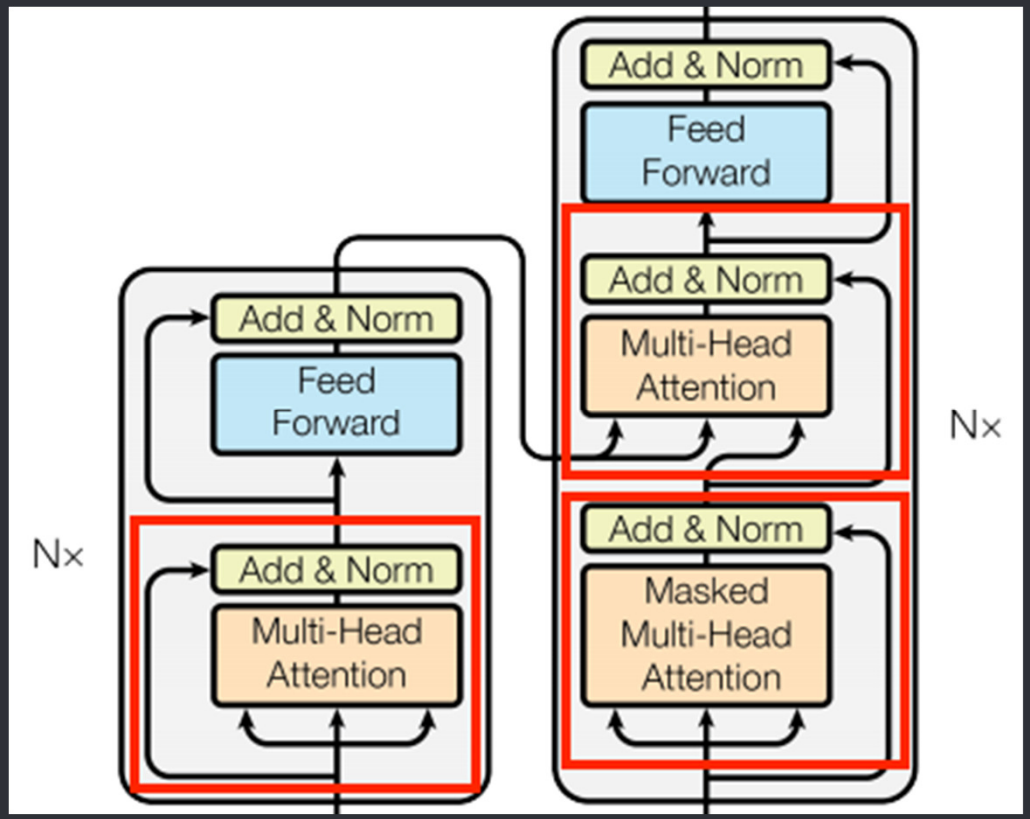
$$\textit{Attention}(Q, K, V) = \begin{pmatrix} 2 & 2 & 2 \\ 1 & 1 & 1 \\ 3 & 3 & 3 \end{pmatrix}$$

Thus, we expect that the first line in the output should be very similar to the second value!

Details of Base Attention

Occurrences of Attention in a Transformer:

Attention is *repeated many times* in the Transformer inside *different components*. The main difference in every instance of this structure is the Queries/Keys/Values which are passed to it.



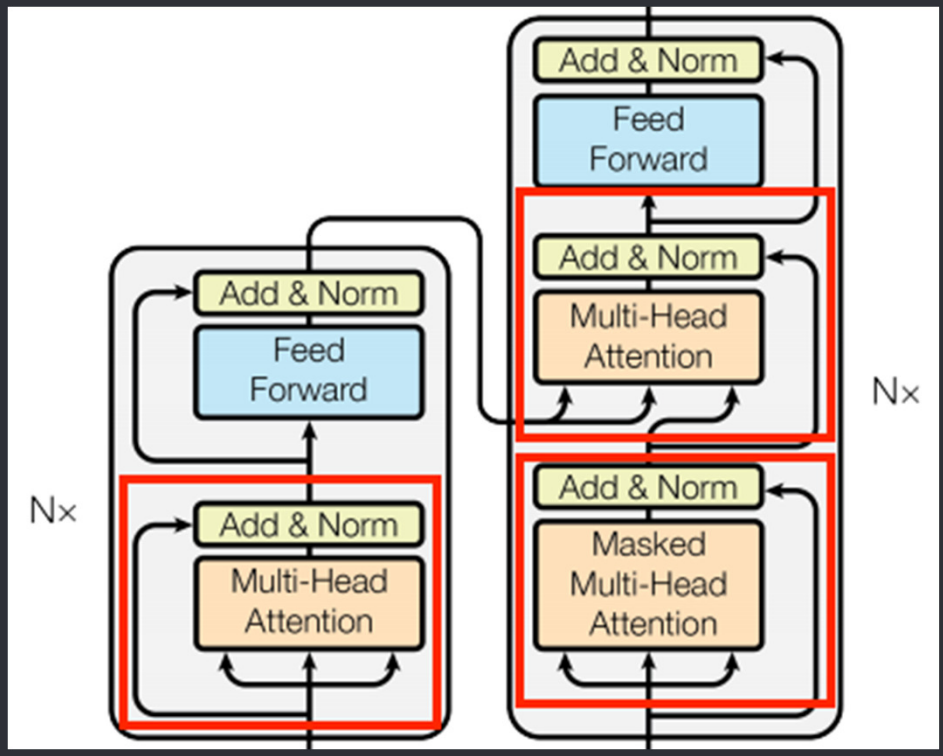
Details of Base Attention

Occurrences of Attention in a Transformer:

For example, in the Encoder layer

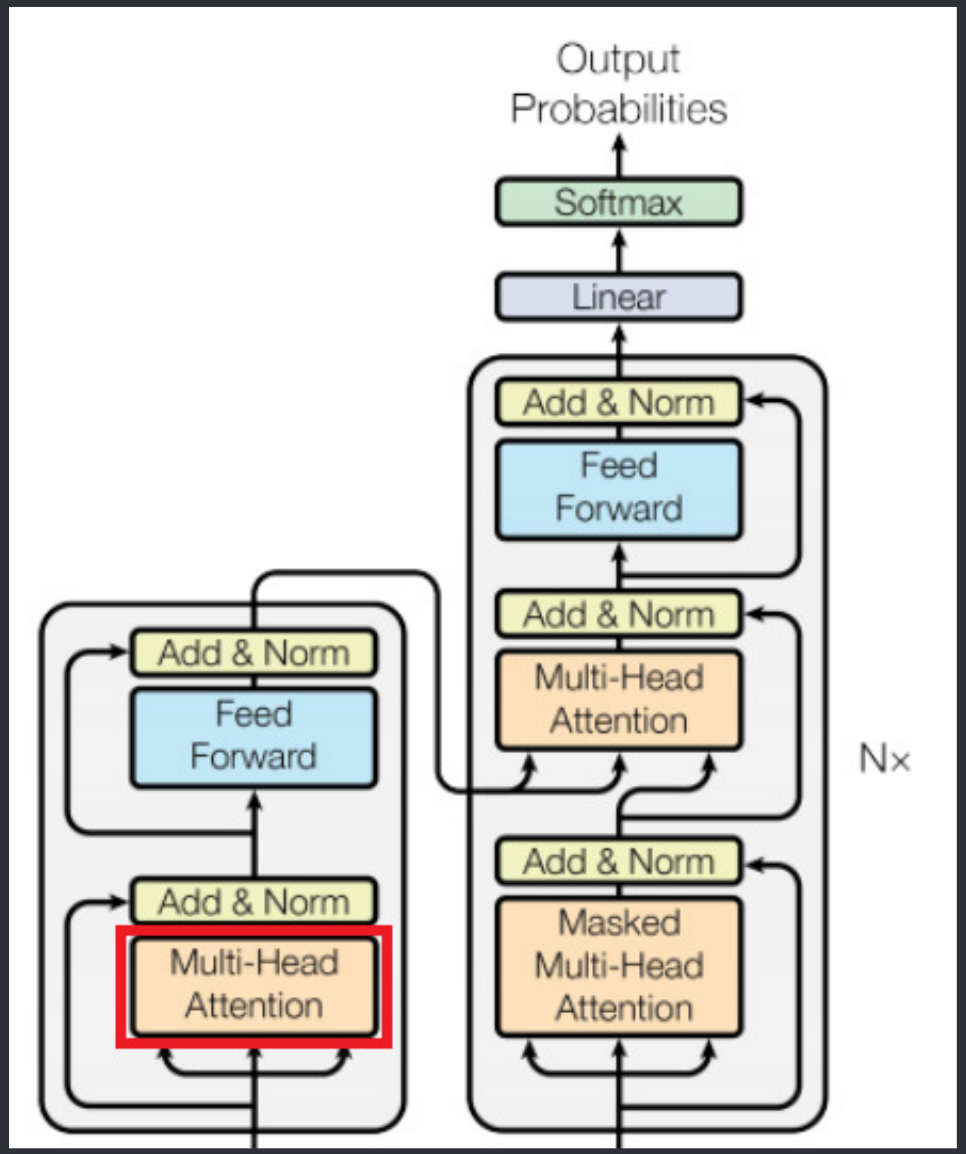
Q=K=V=(input sequence).

However, in the decoder, Q can be different than K and V.



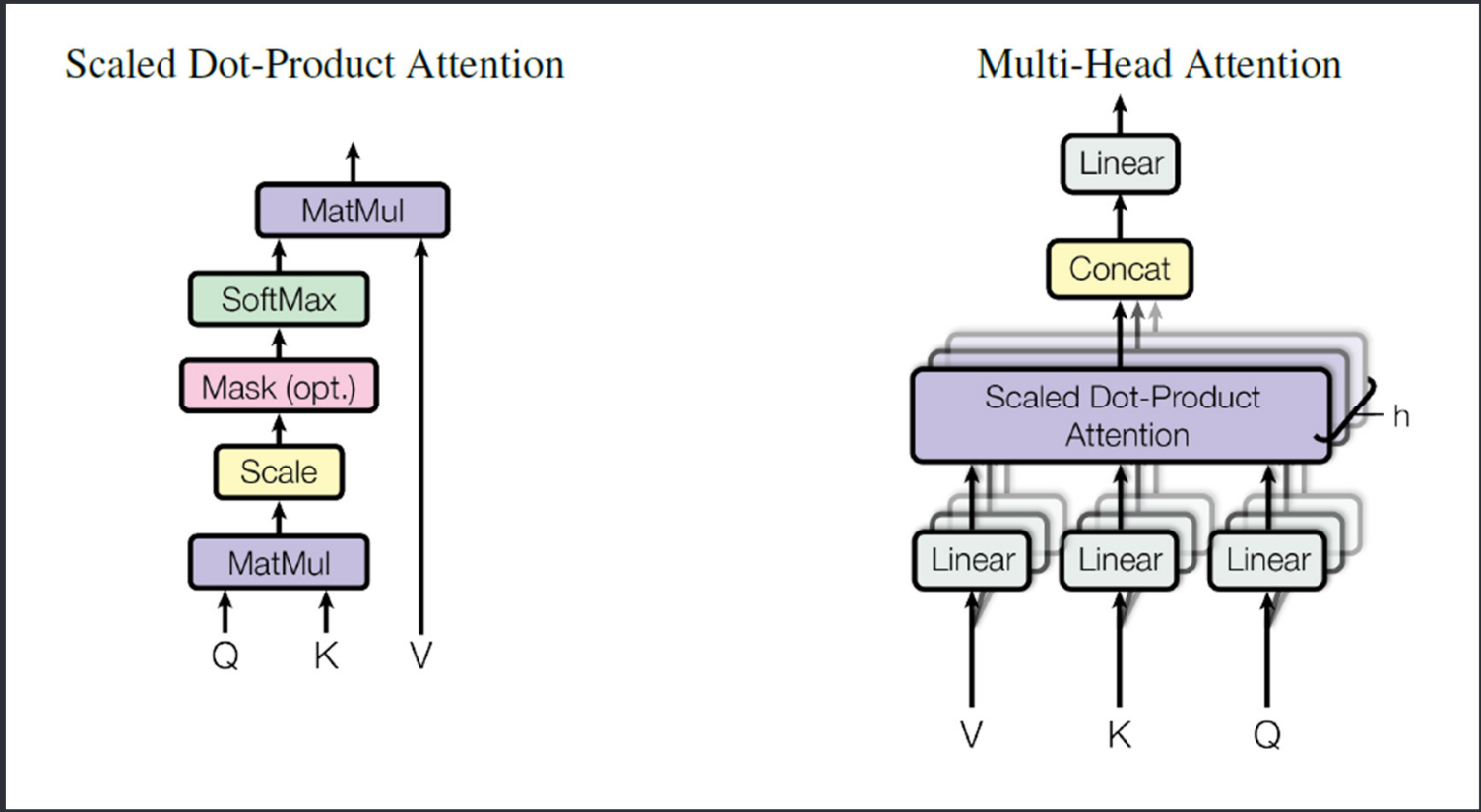
Multi-Head Attention

A multi-head attention combines multiple Base Attention Blocks:



Multi-Head Attention

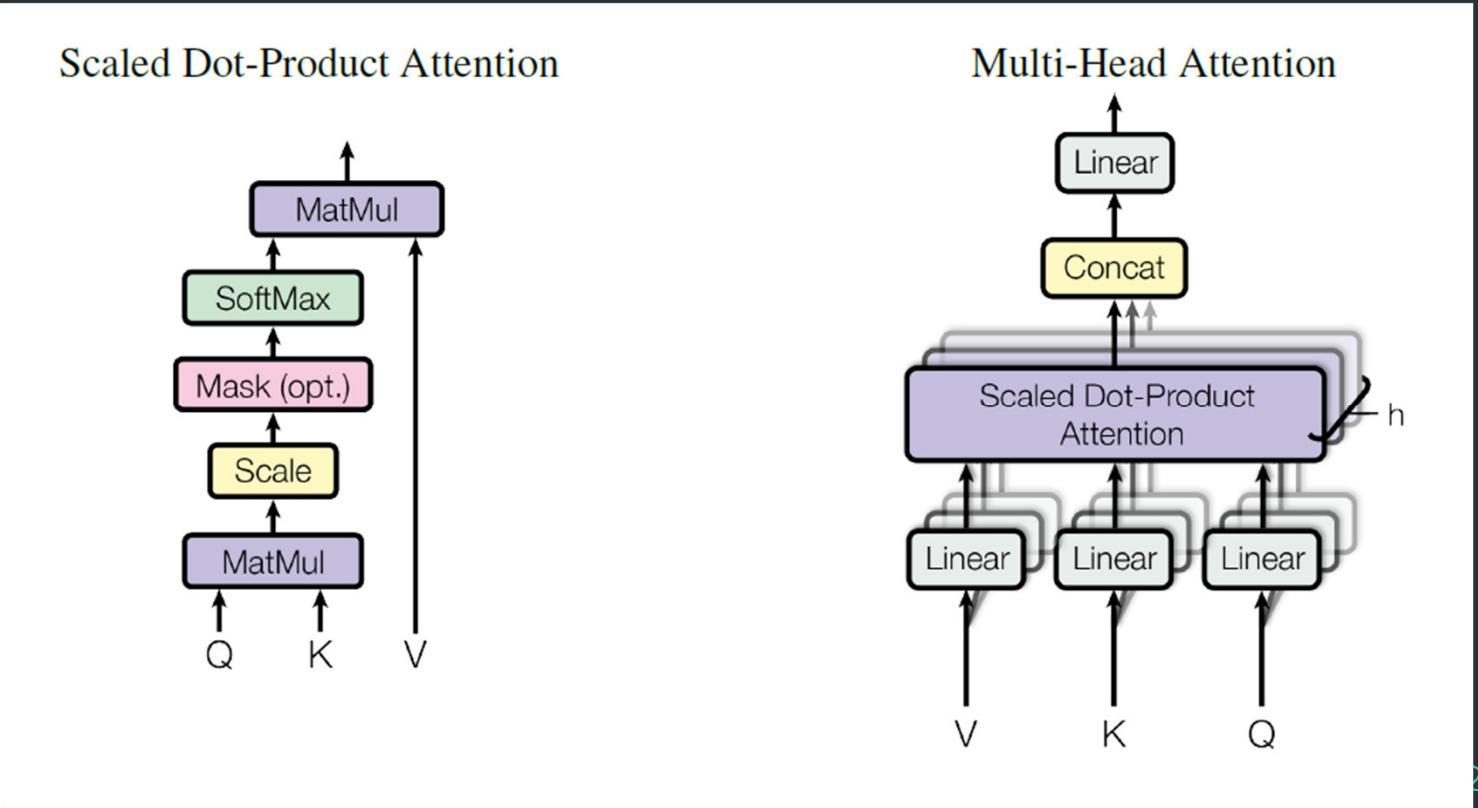
Instead of performing a single attention operation, the paper's authors found it beneficial to linearly project the queries, keys and values 'h' times, with different, learned linear projections:



Multi-Head Attention

Meaning, they created multiple ‘heads’, each head performs an Attention operation, called a "scaled dot product attention“.

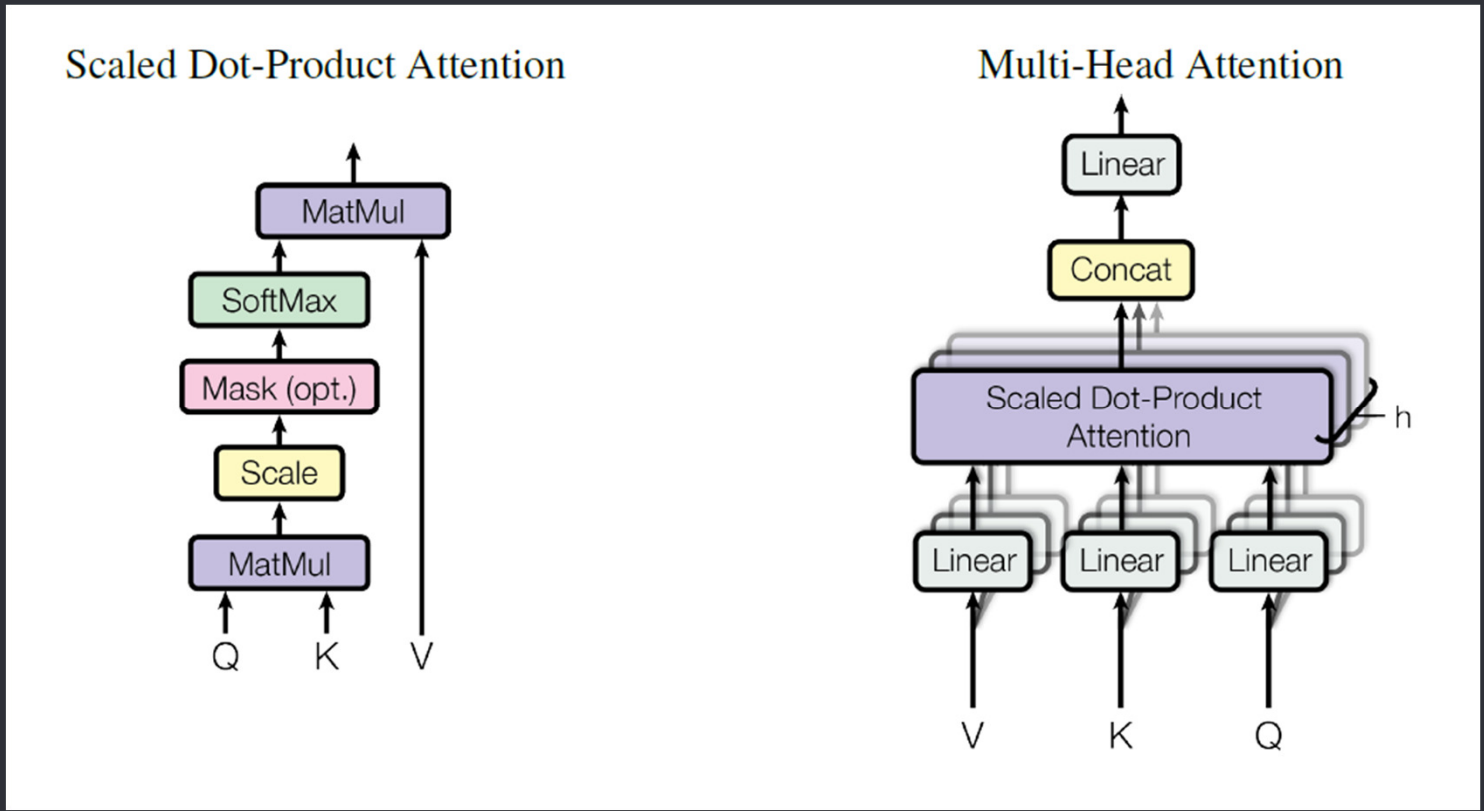
Each head has **different/separate trainable weights**, with different, learned linear projections.



Multi-Head Attention

On each of these projected versions of (Q,K,V), they perform the Attention function **in parallel**, yielding the output values.

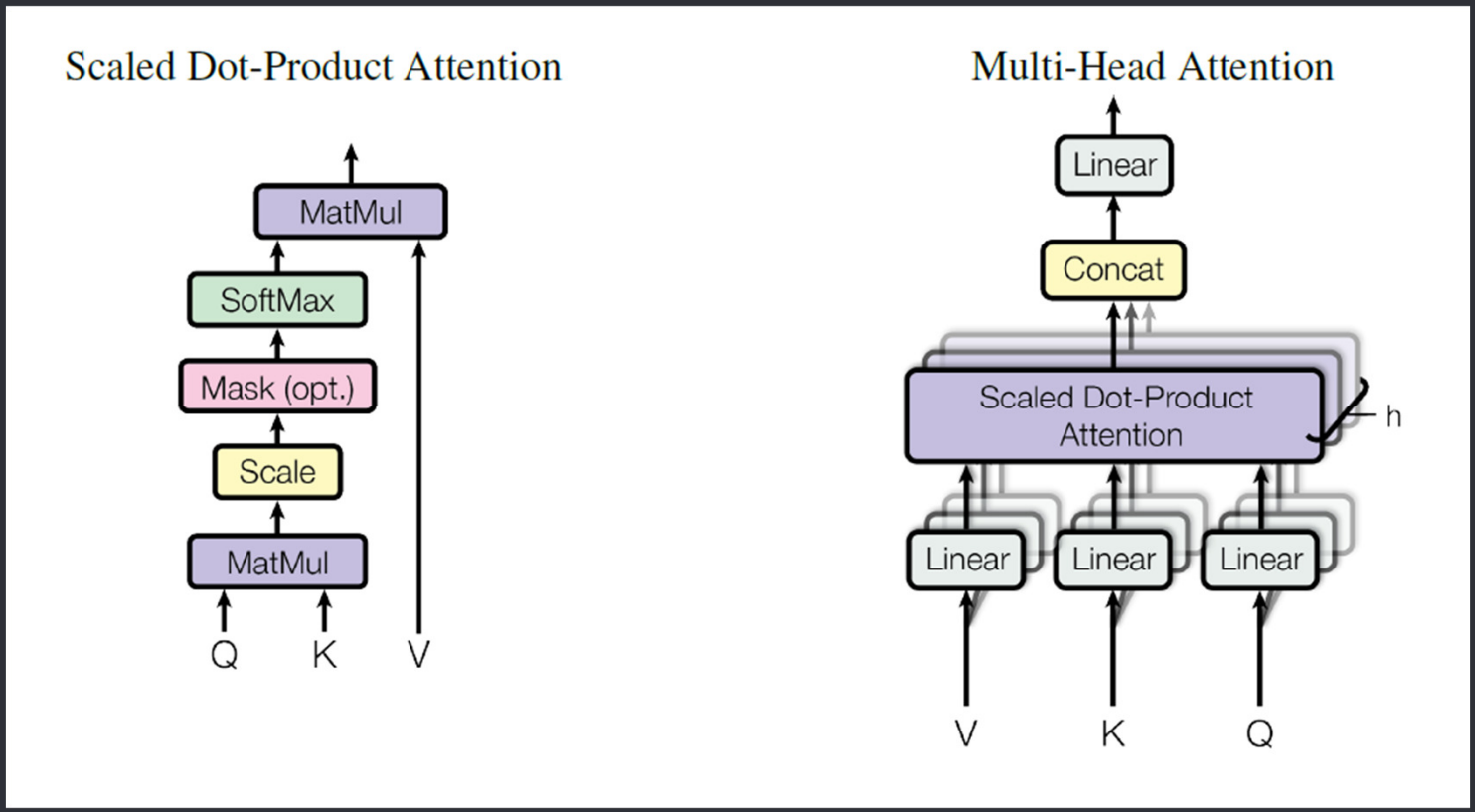
This creates **multiple representations** of the embeddings.



Multi-Head Attention

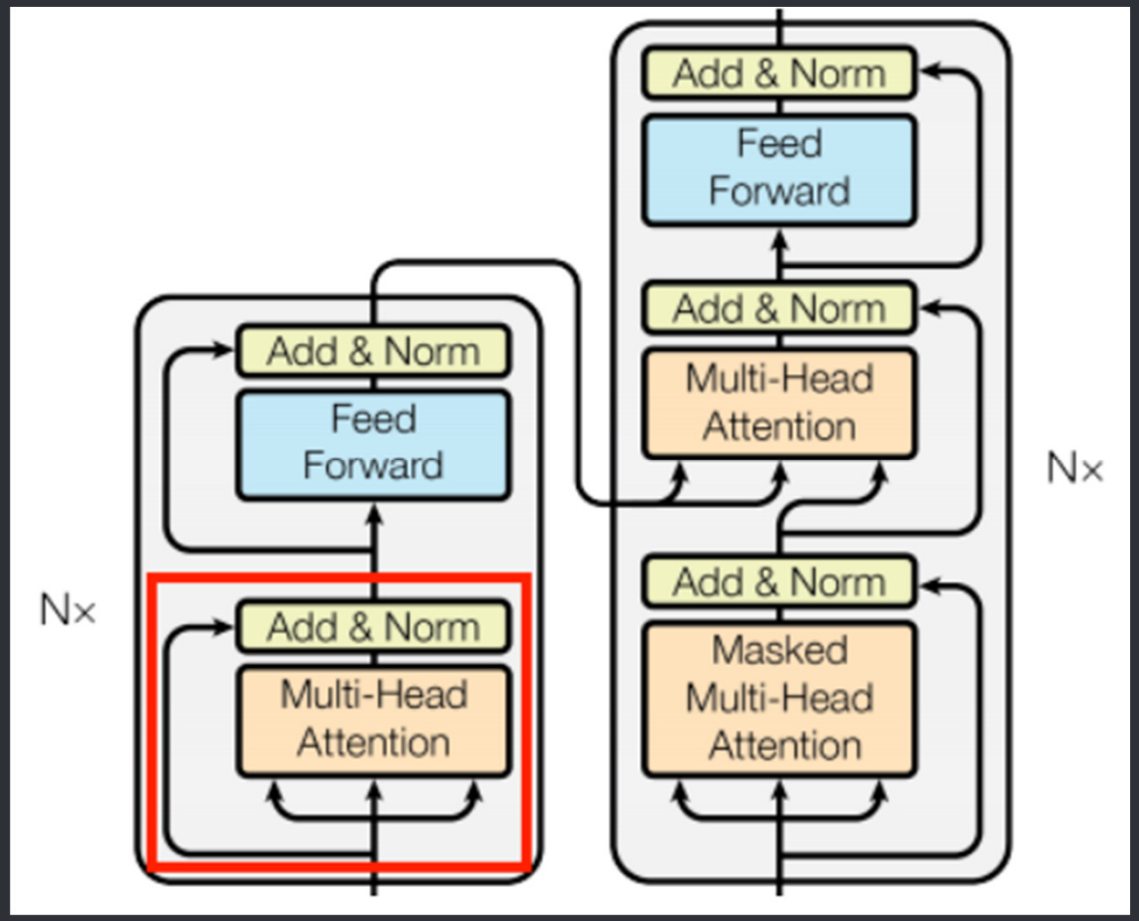
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.



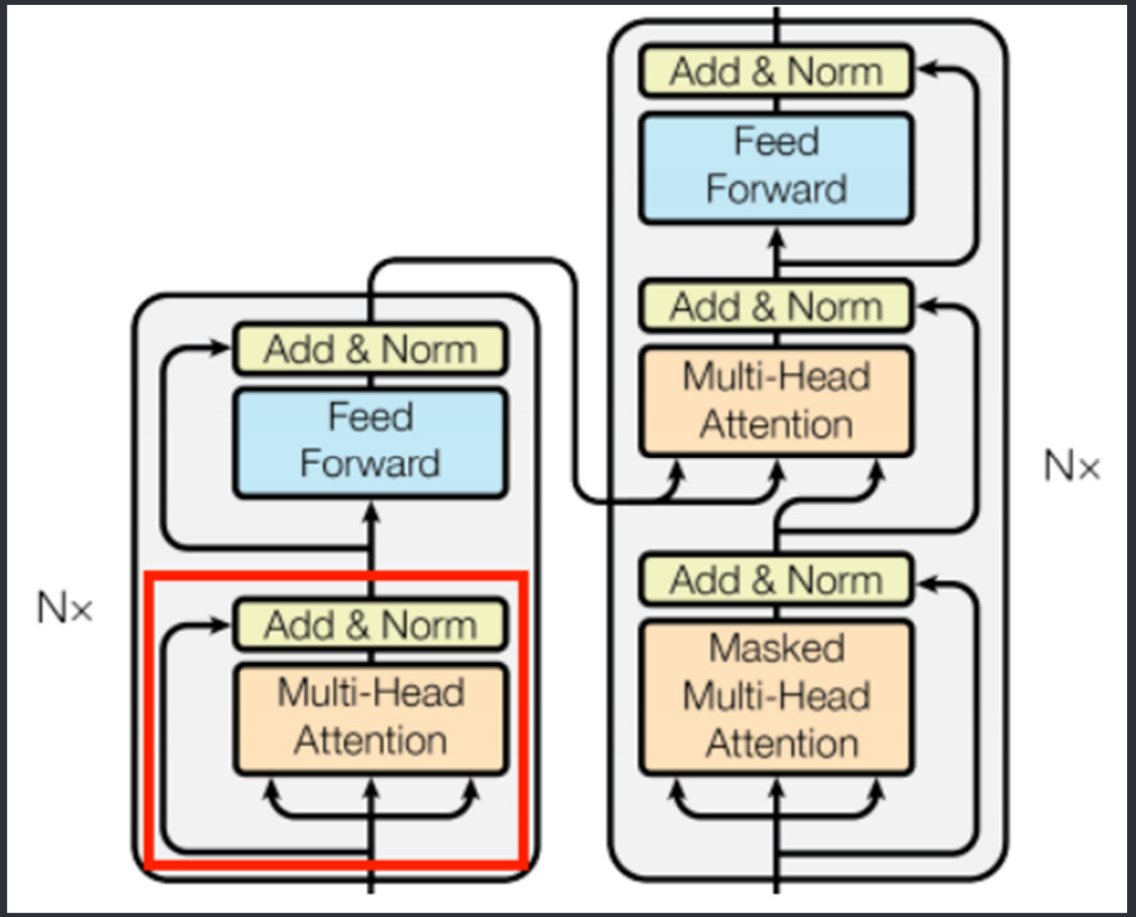
Global Attention Layer

This layer is responsible for processing the context sequence, and propagating information along its length.



Global Attention Layer

Since the context sequence is known while the translation is being generated, information is allowe to flow in both directions of this sequence.



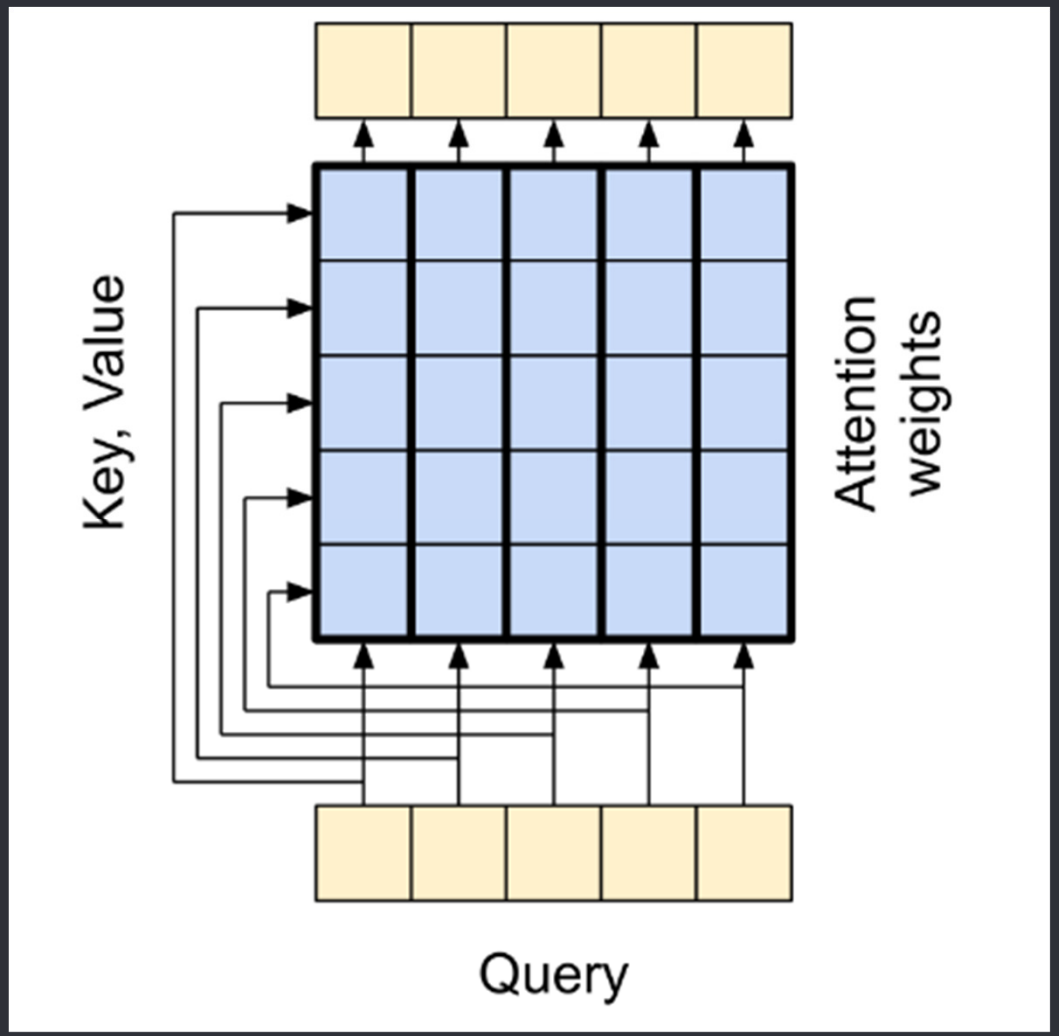
Global Attention Layer

The input to this layer are the initial representations/embeddings of all words in the original sentence.

Then, using self-attention it aggregates information from all of the other words, generating a new representation-per-word, **informed by the entire context.**

Global Attention Layer

This layer lets every sequence element directly access every other sequence element, with only a few operations:



Global Attention Layer

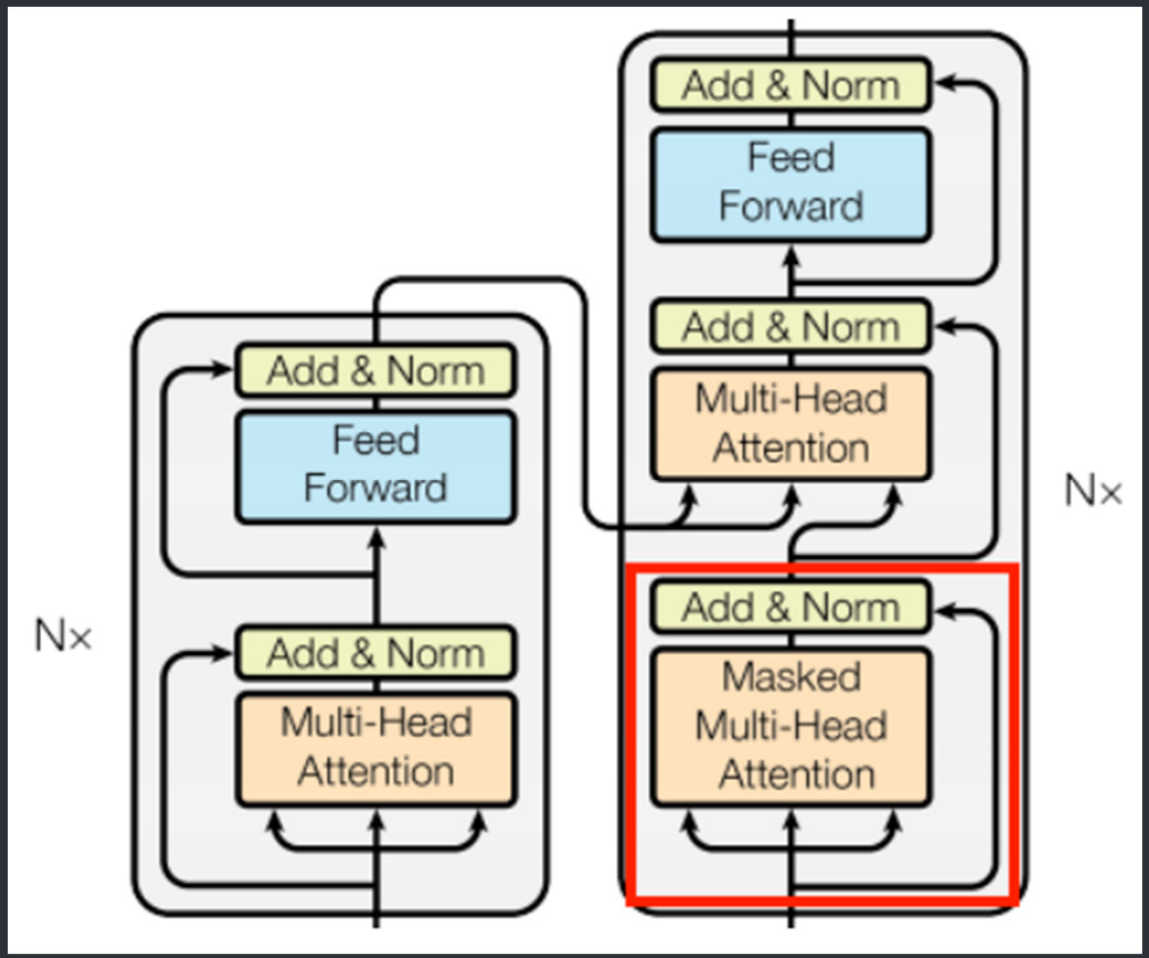
This layer has weights matrices W_i^Q, W_i^K, W_i^V which are updated by the training process.

The more we train, *the better our representations will be*:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

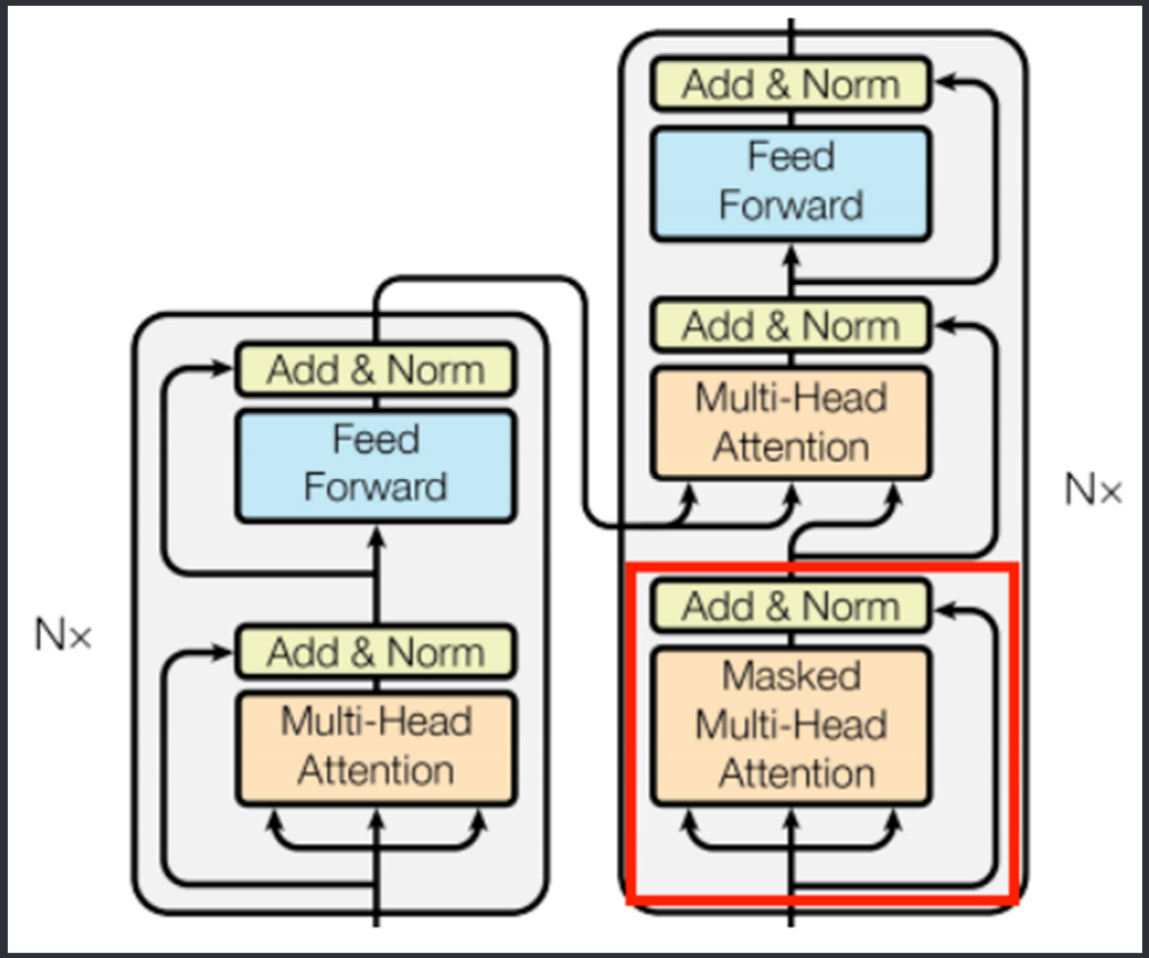
Causal Attention Layer

Transformers are an "autoregressive" model - They generate the text one token at a time and feed that output back to the input.



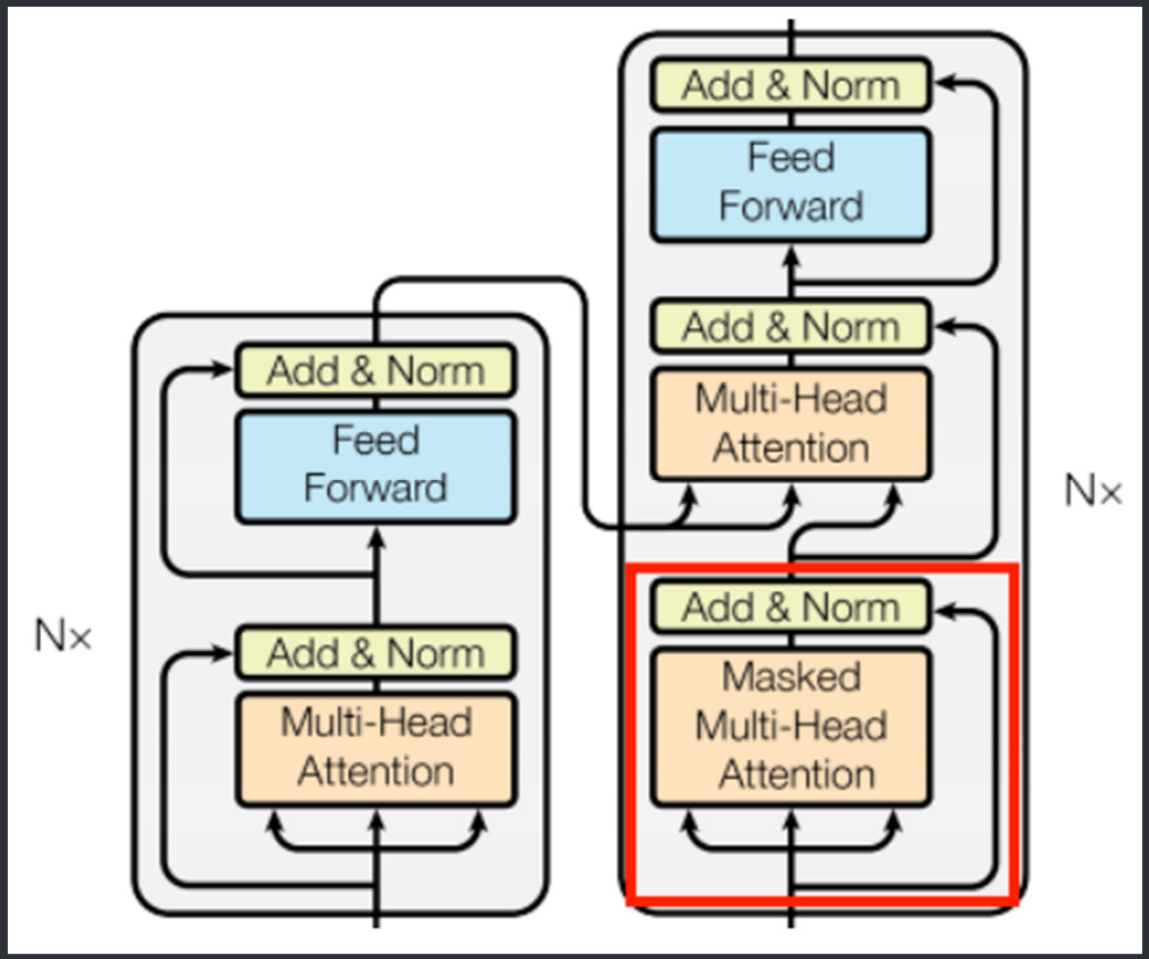
Causal Attention Layer

To make this efficient, this layer ensures that the output for each sequence element only depends on the *previous sequence elements*, i.e the models are *causal.*



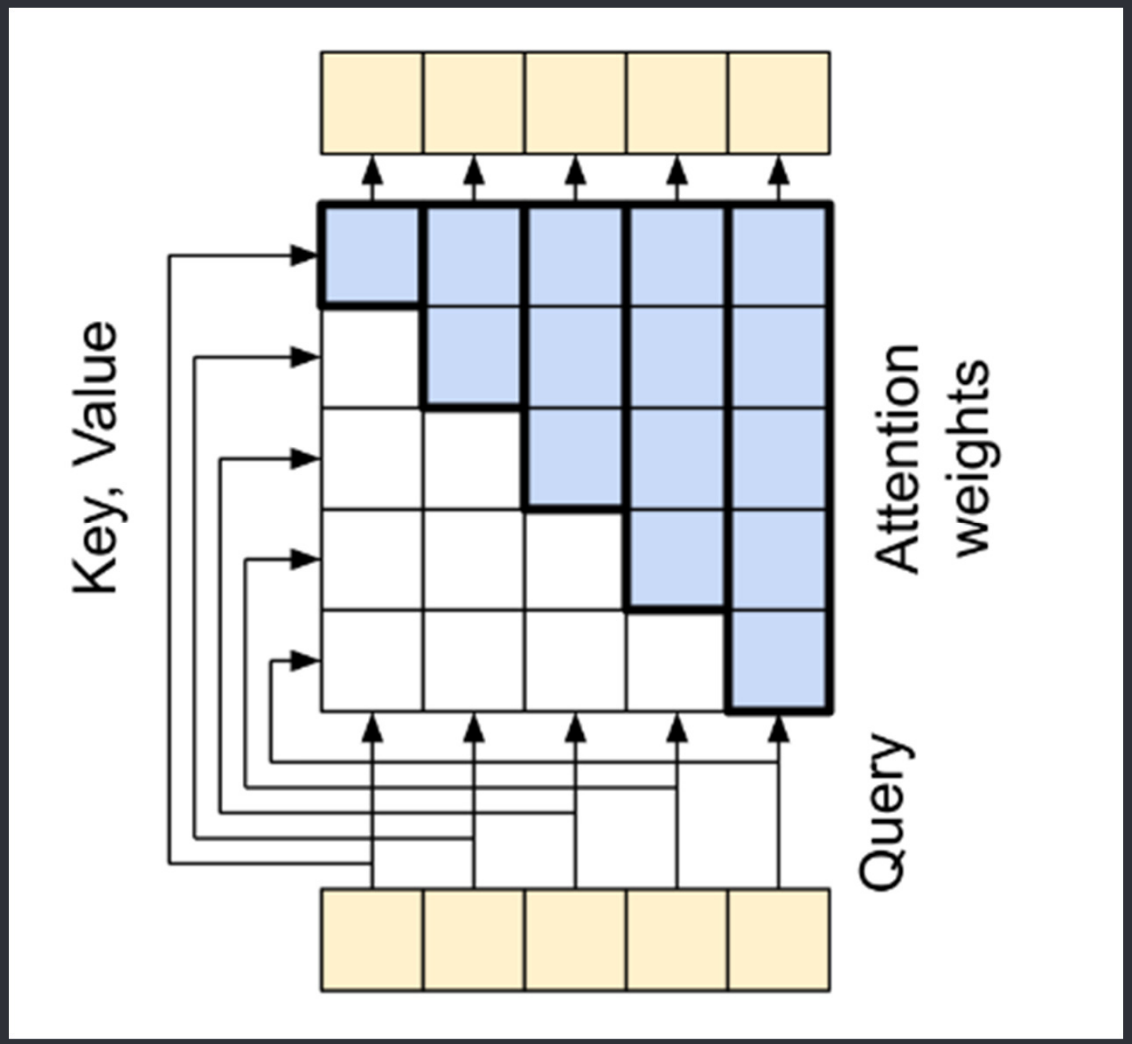
Causal Attention Layer

The inputs to this layer is the translated sentence (in our example, English), passed to Q,K,V.



Causal Attention Layer

To build a causal self attention layer, you need to use an appropriate mask:



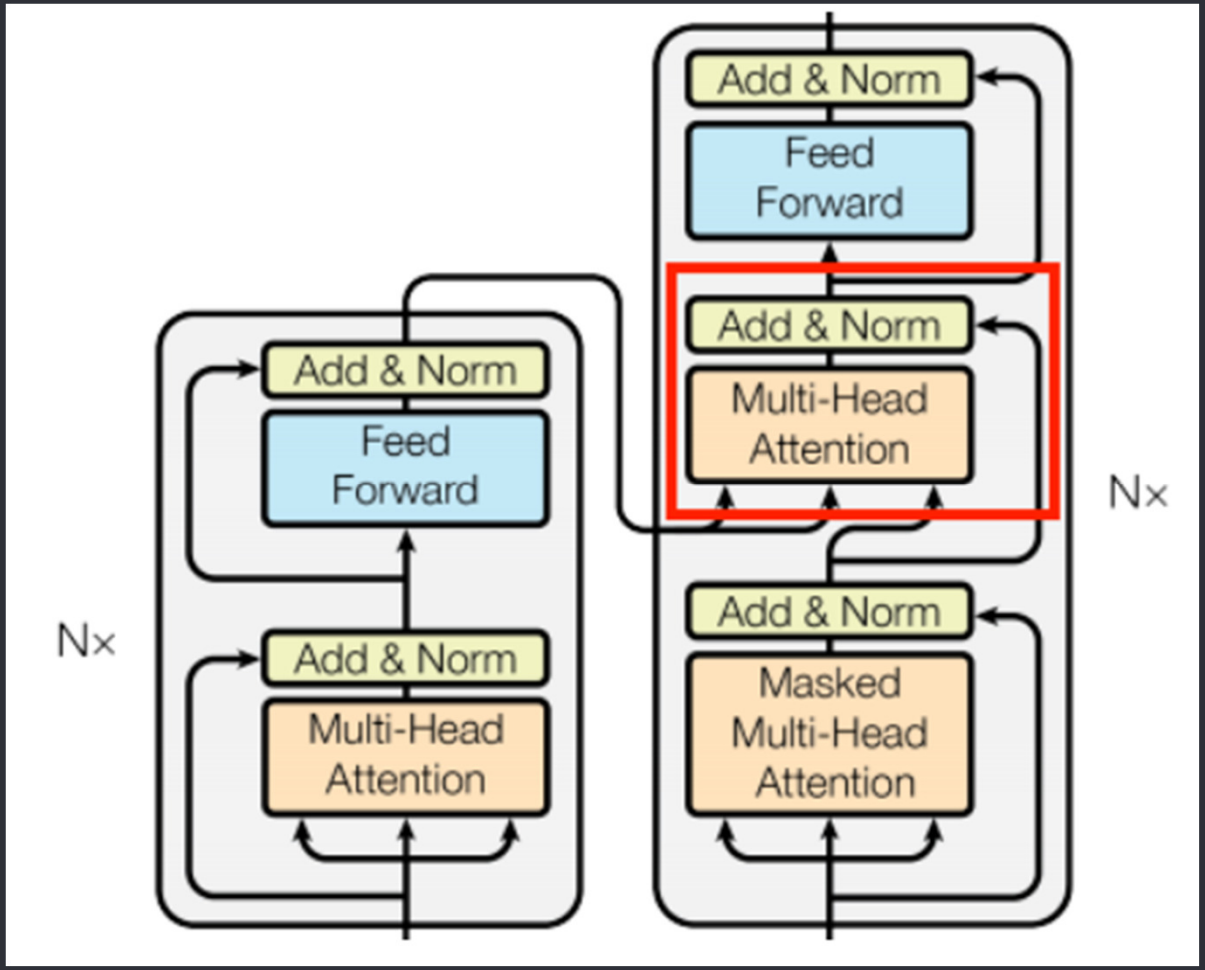
Causal Attention Layer

This layer's weights matrices W_i^Q, W_i^K, W_i^V are updated by the training procedure, so that they'll transform the English sequence encoded in Q,K,V into a *better representation of itself:*

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

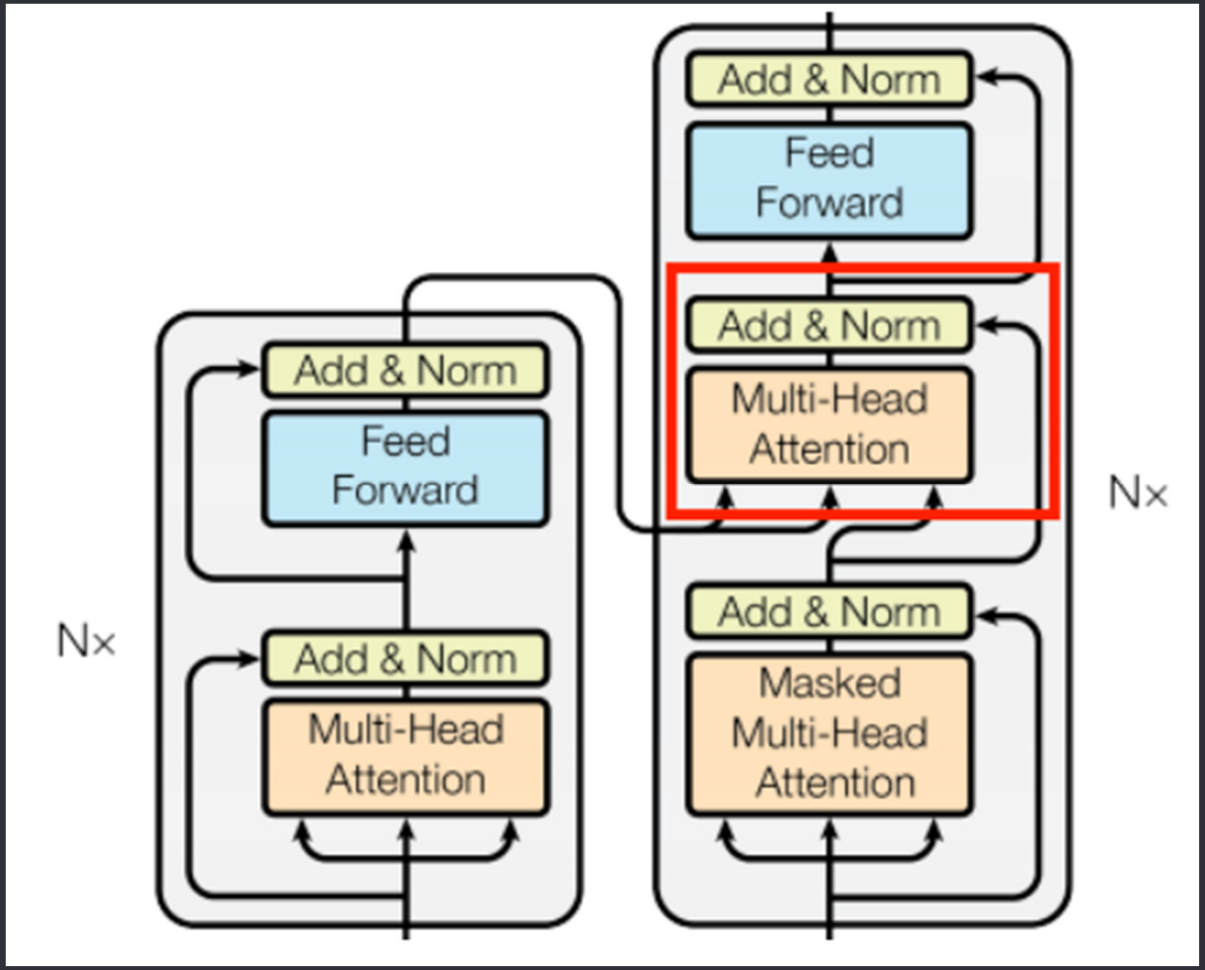
Cross Attention Layer

This layer connects the encoder and decoder. This layer is the most straight-forward use of attention in this model.



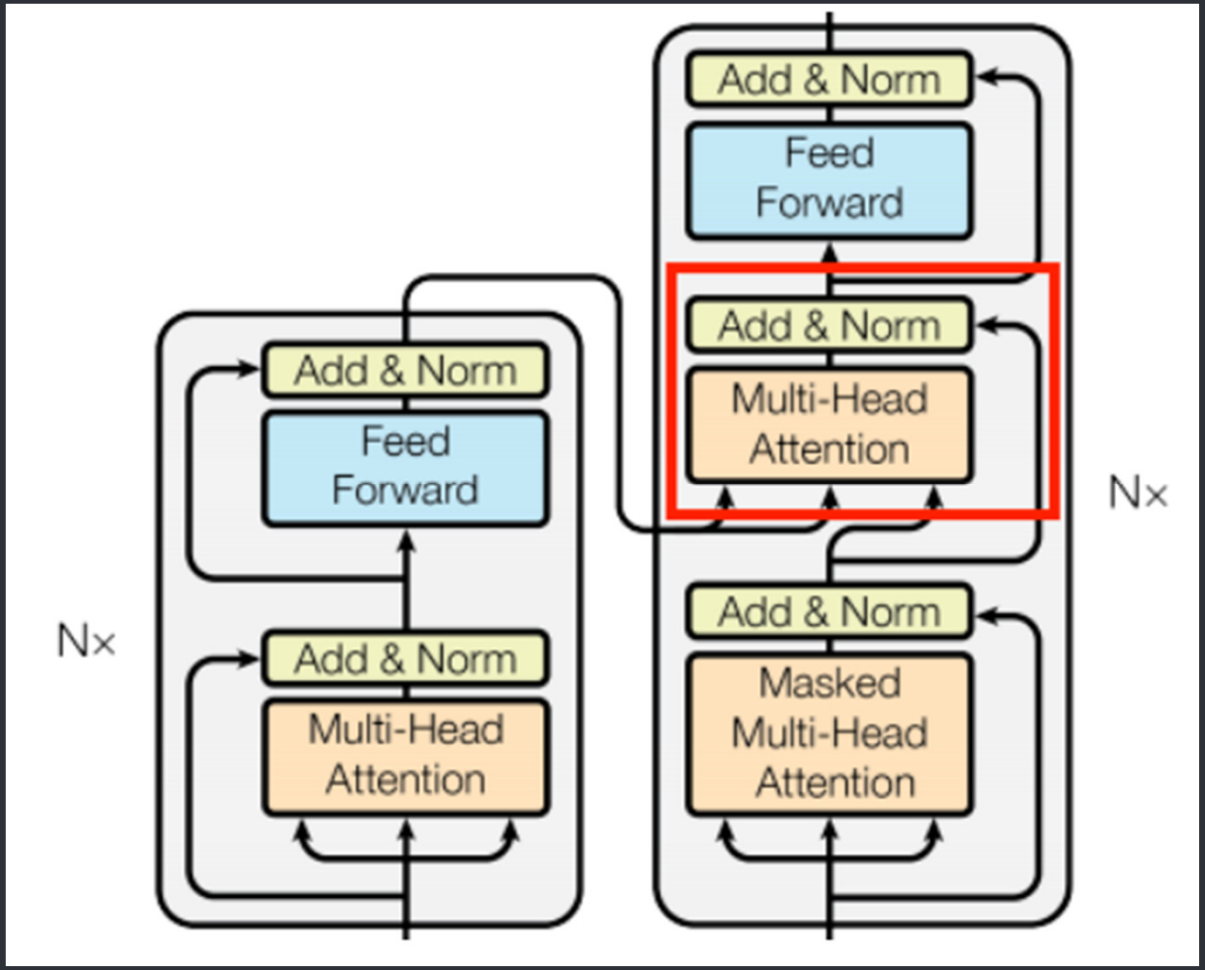
Cross Attention Layer

The output of the Causal Attention layer below, is passed as the Queries to this layer.



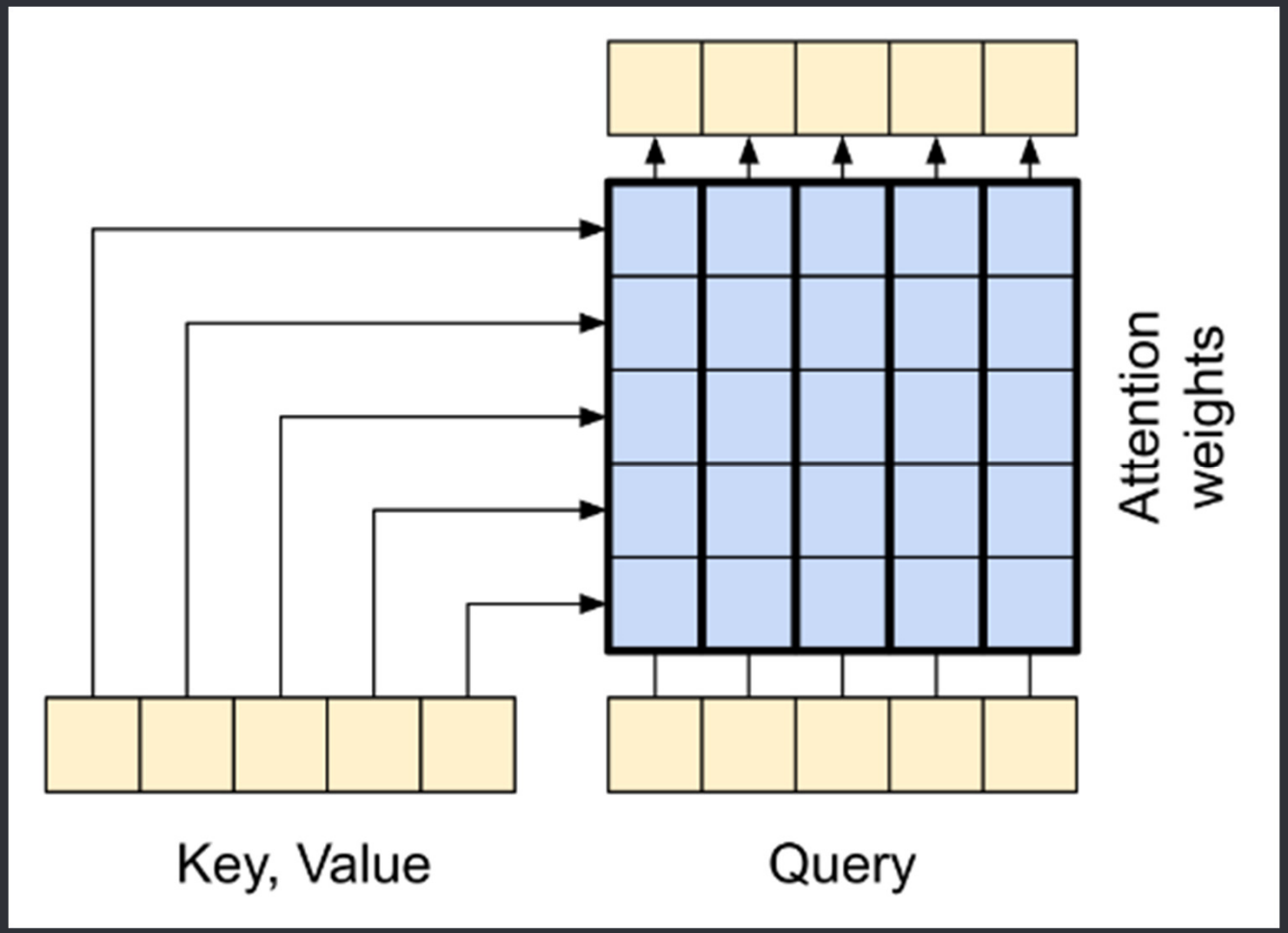
Cross Attention Layer

The output of the encoder layer to the left, is passed as the Keys and Values to this layer.



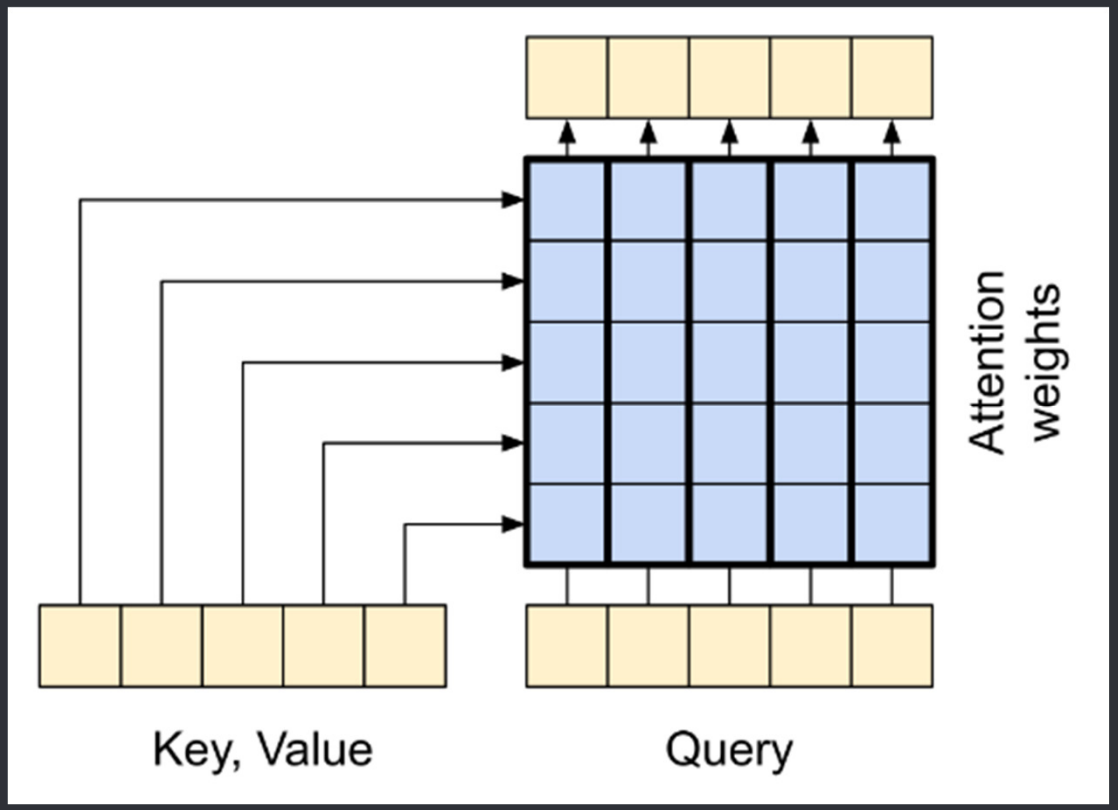
Cross Attention Layer

Each query can see all the key/value pairs in the context.



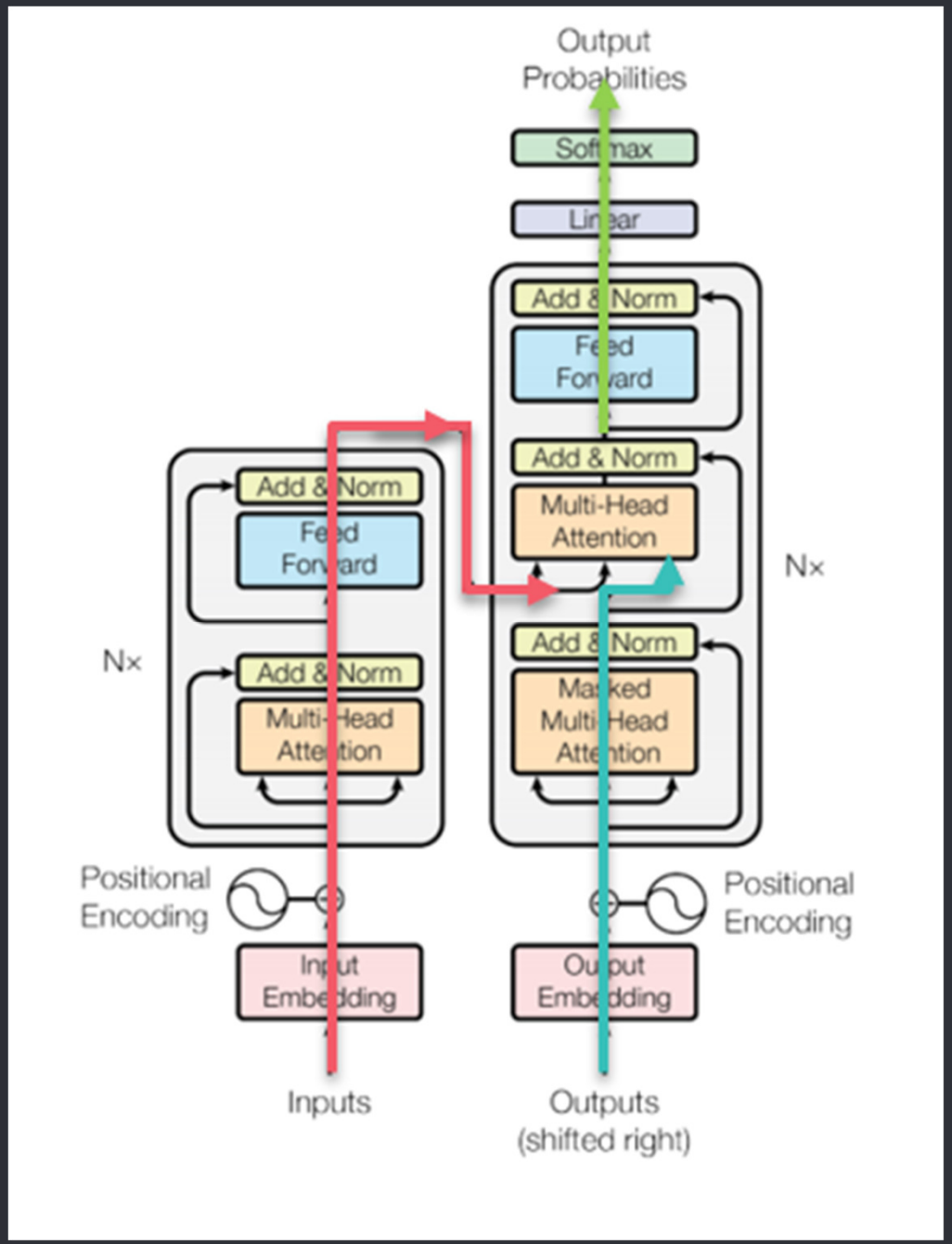
Cross Attention Layer

The W_i^Q, W_i^K, W_i^V matrices in this layer are trained to transform the Portuguese context sentence (encoded in K and V), into a representation which is more *useful* to the queries in Q.



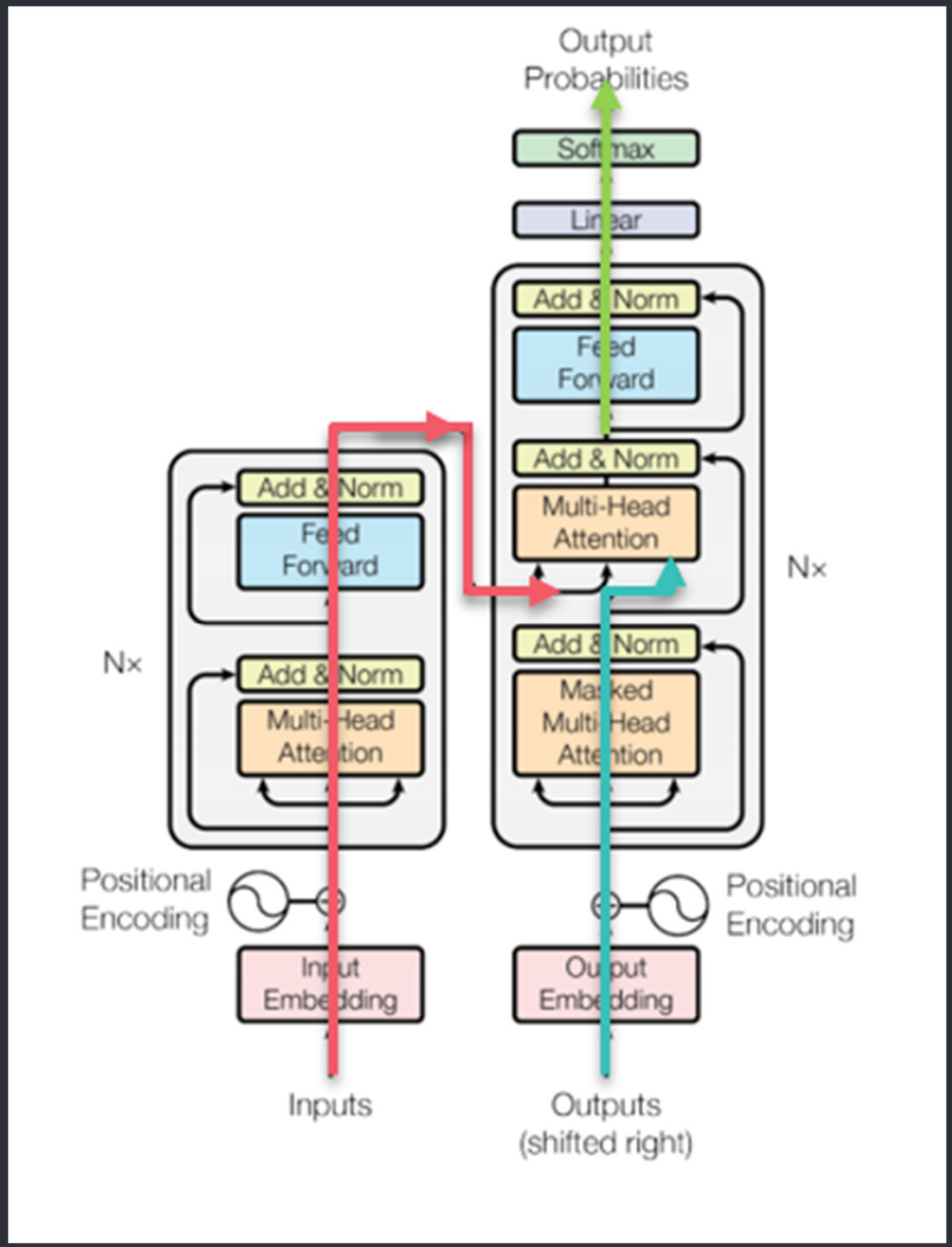
Holistic View

The flow of word embeddings from the original sentence in **Portuguese**, is combined with the word-embeddings of the translated sentence in **English**.



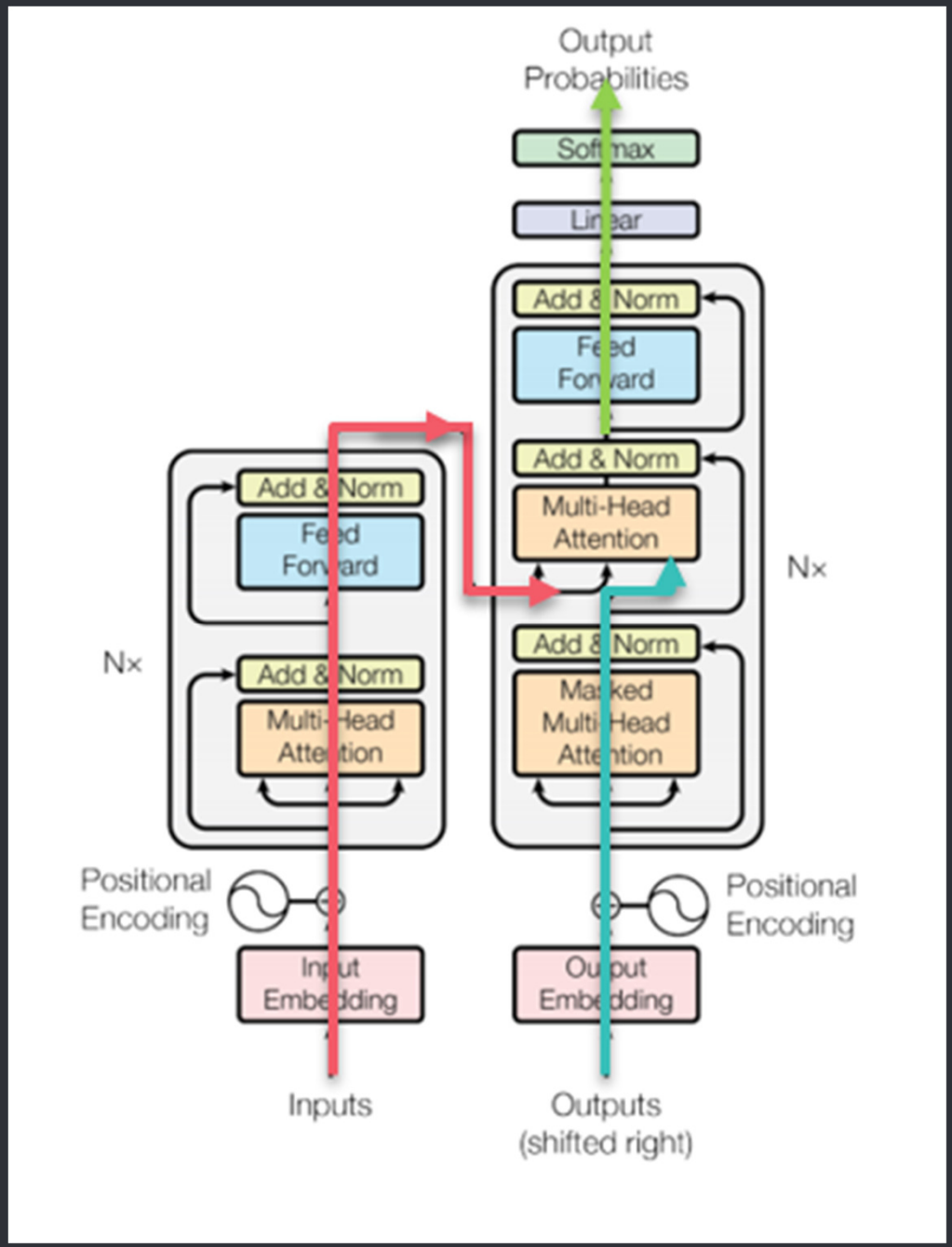
Holistic View

The Portuguese embeddings provide the **context** to the English words *translated so far*, and allow them to "see the future" by looking at the entire sentence's context.



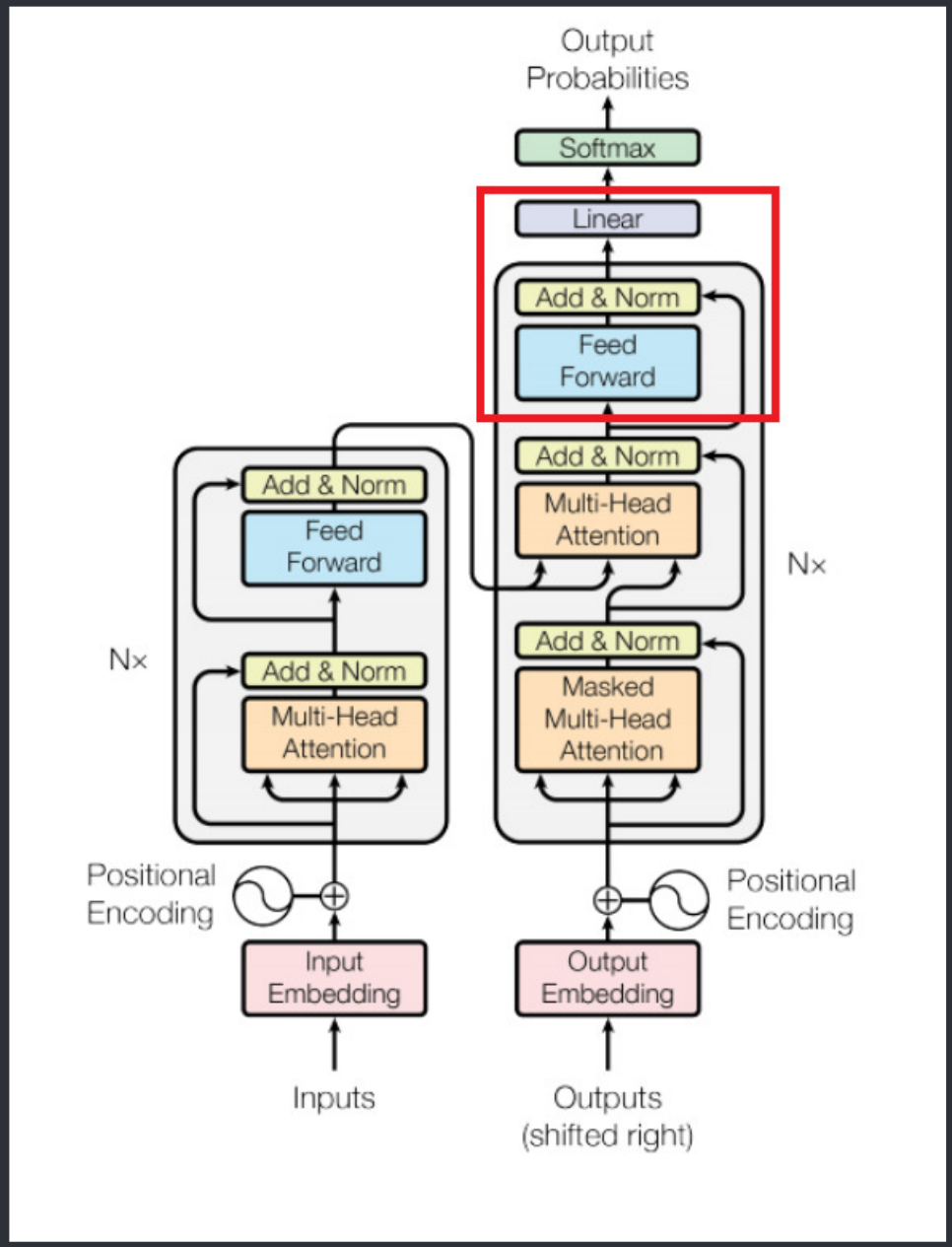
Holistic View

Now we have enough "data" to predict the **next English word** in the sentence.



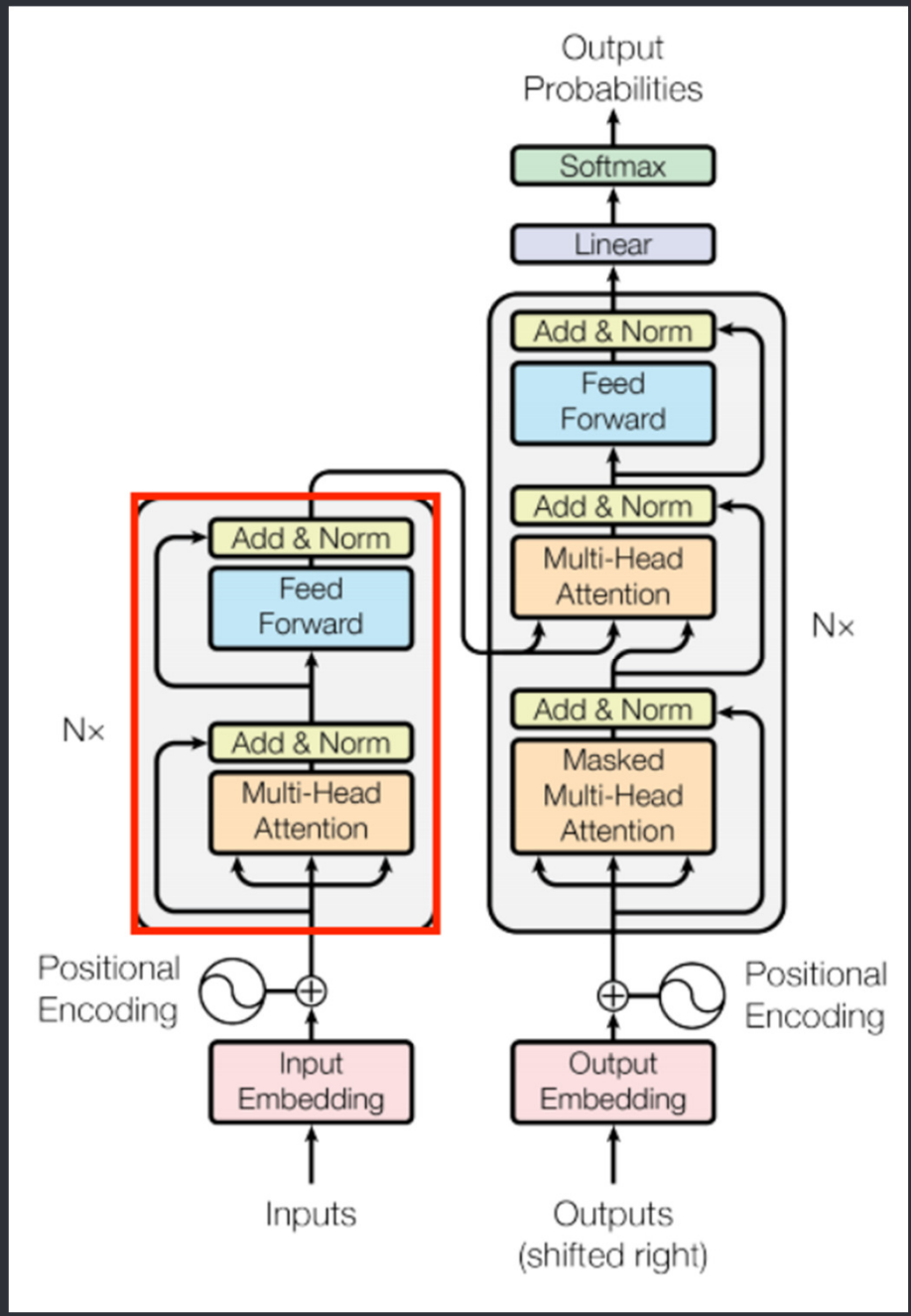
Holistic View

In the next stage, the Feed Forward layer and the Linear layer, transform and extract from the output of the Decoder, the next word predicted in English.



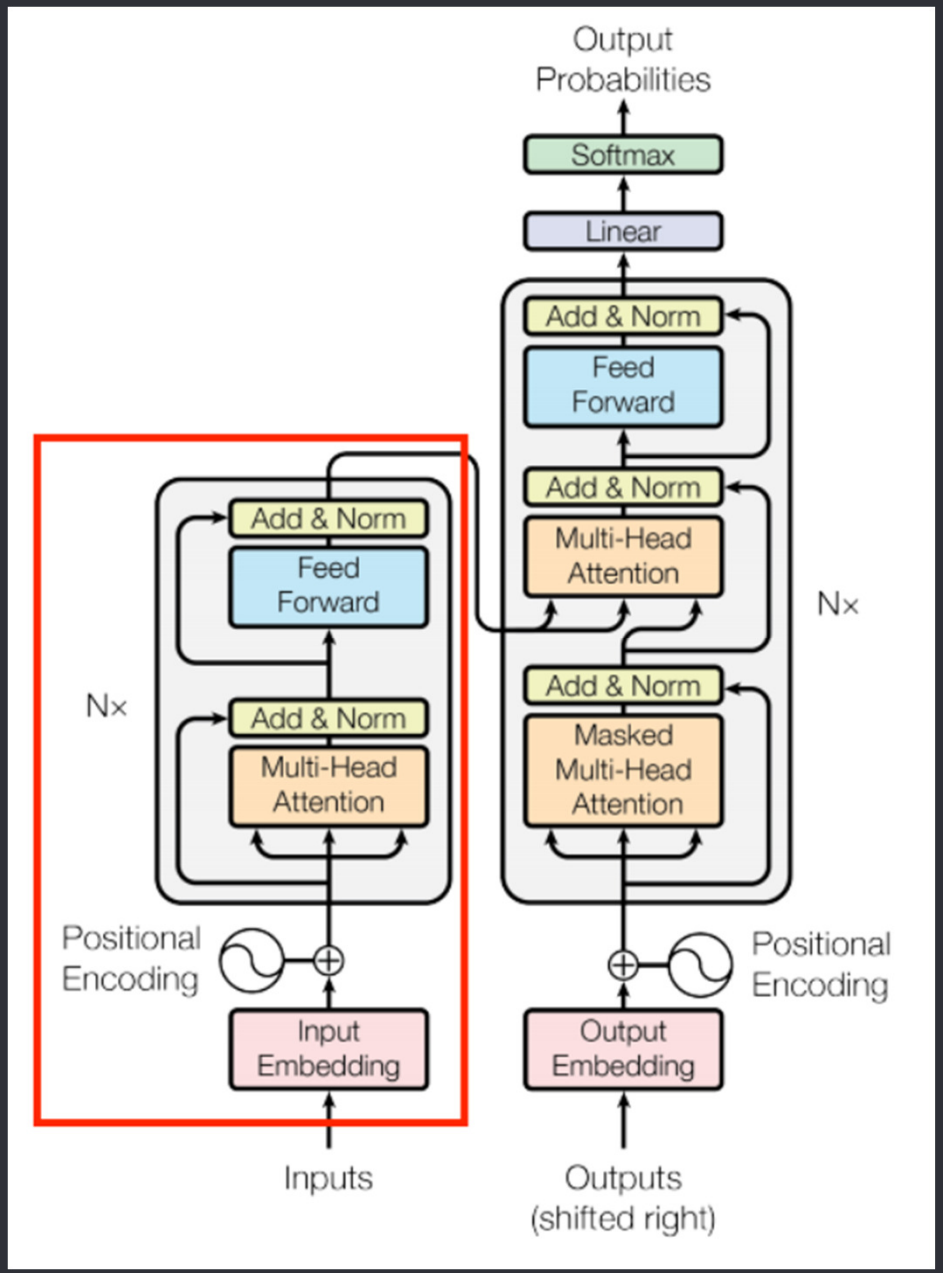
The Encoder Layer

Each Encoder-Layer contains a Global Self Attention and Feed Forward layer.



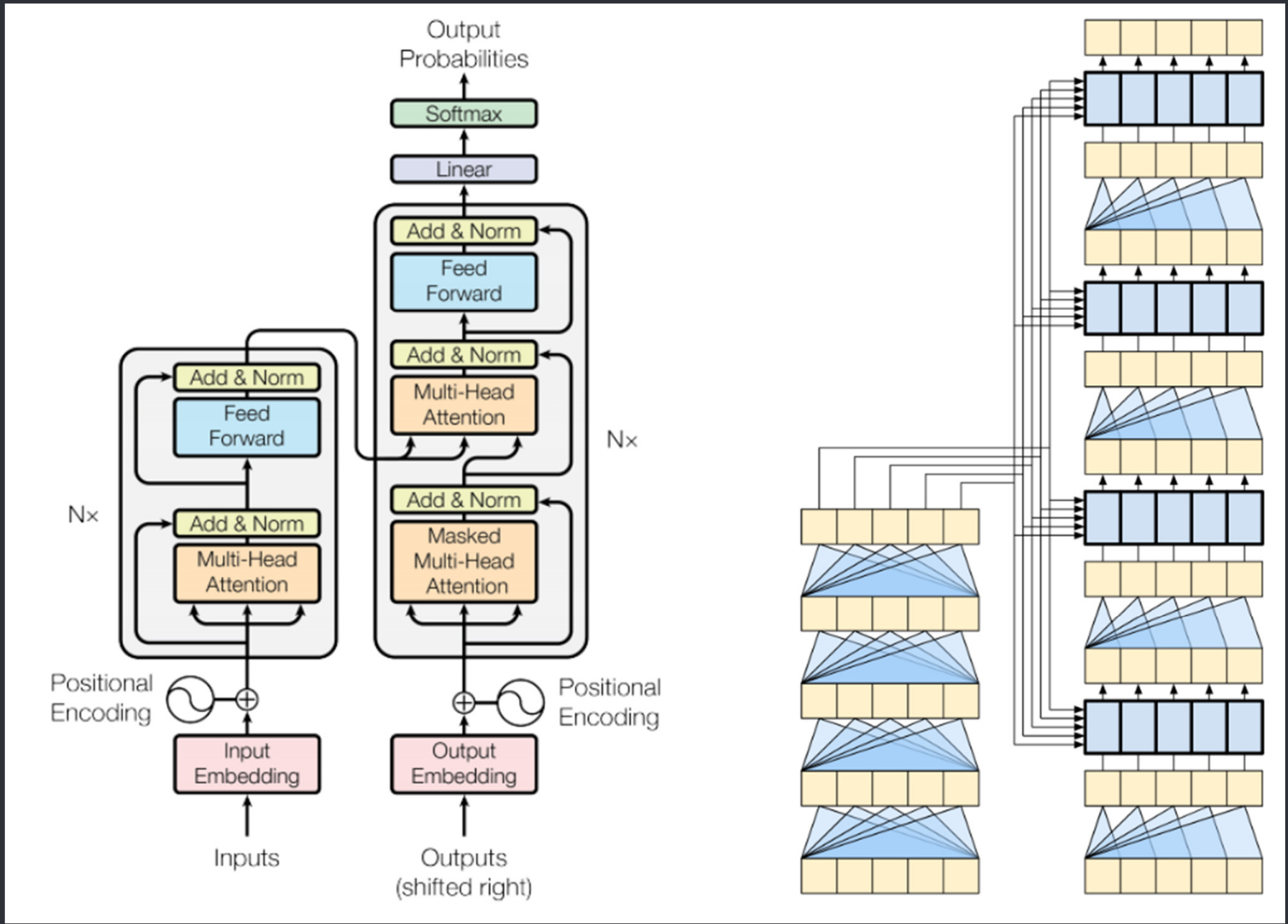
The Encoder

The encoder consists of a Positional-Embedding layer at the input and A stack of Encoder-Layers.



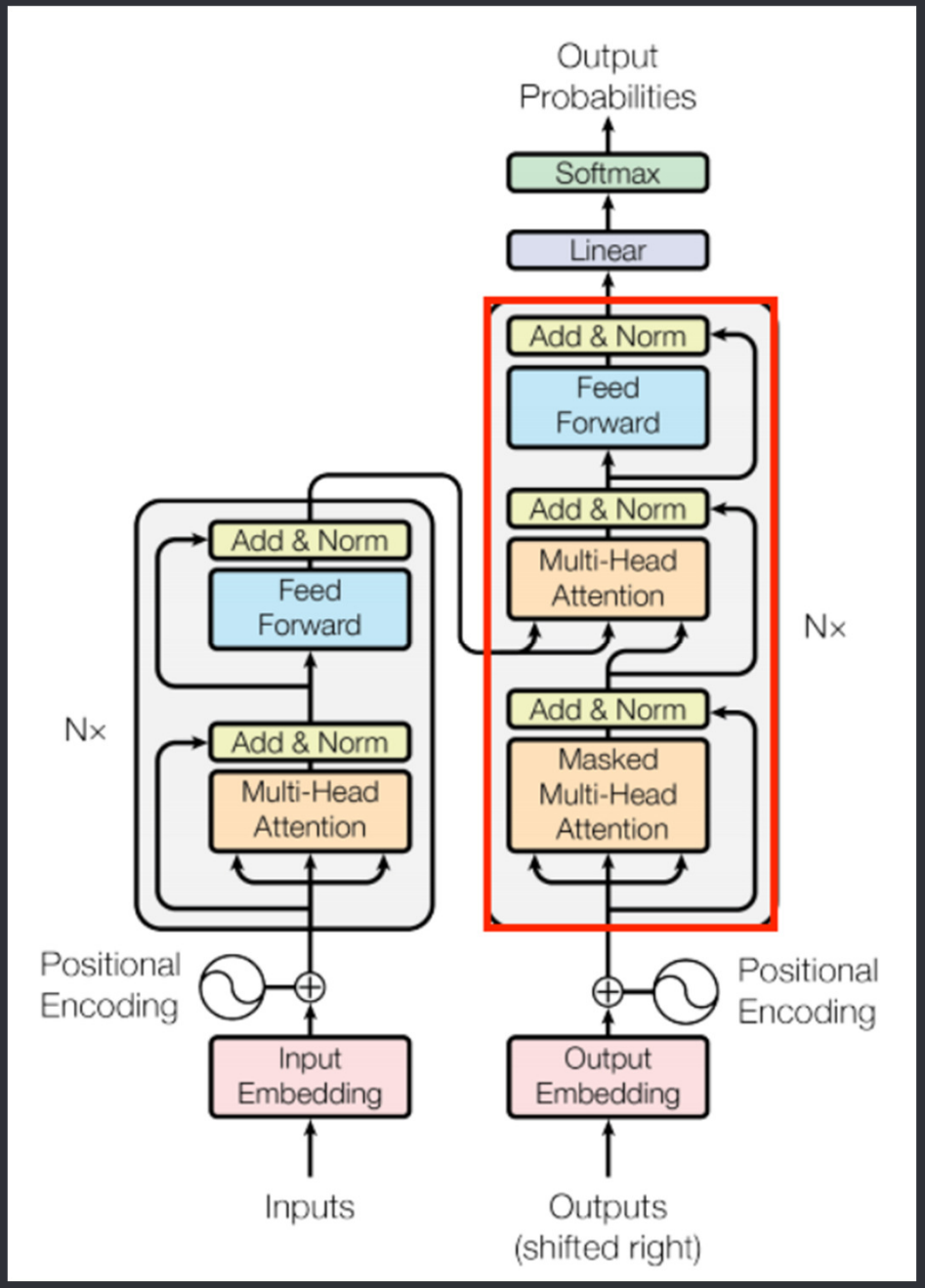
The Encoder

The Encoder contains multiple Encoder-Layers:



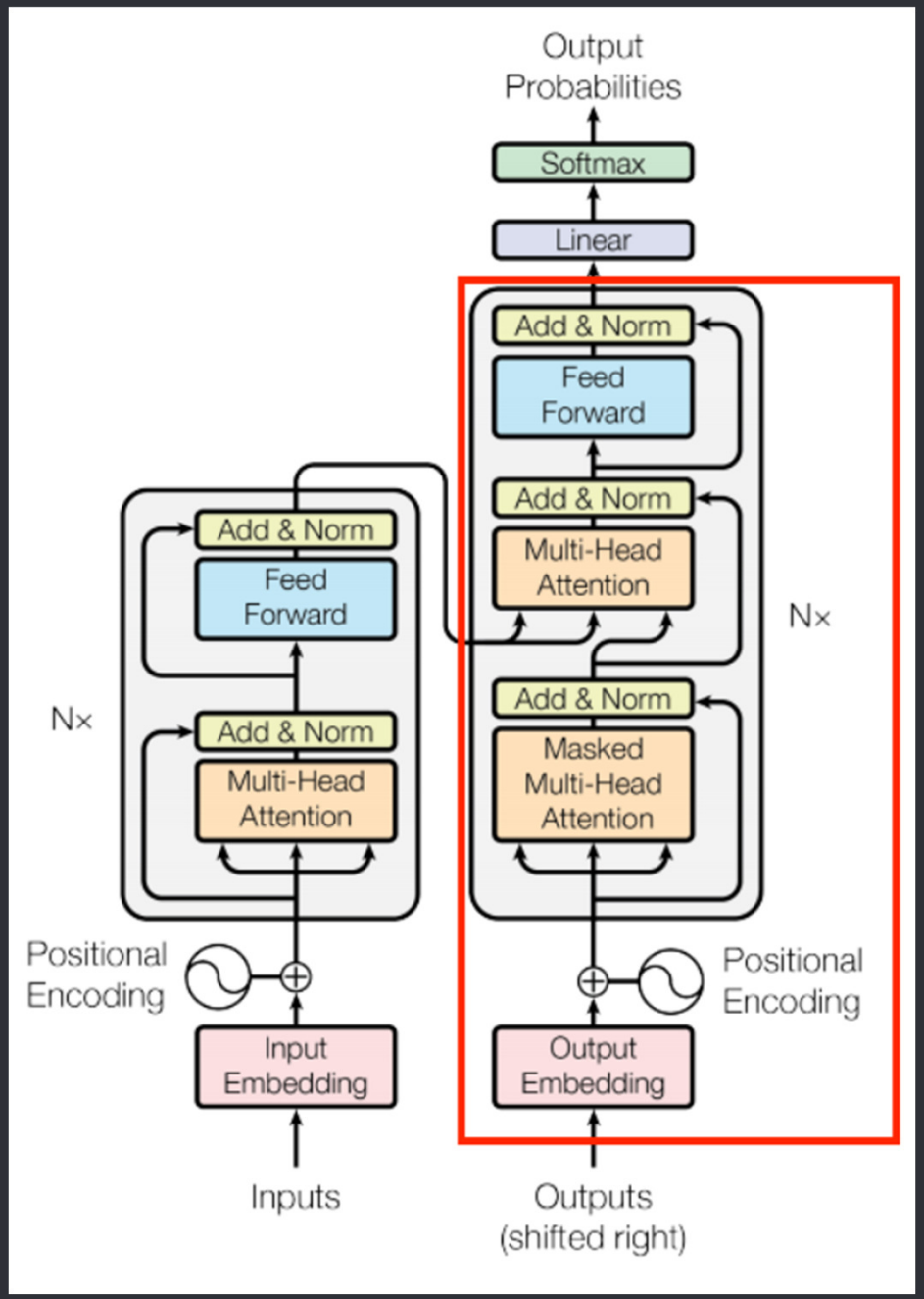
The Decoder Layer

The Decoder-Layer has more components, it contains a Causal Self Attention, a Cross Attention, and a Feed Forward layer.



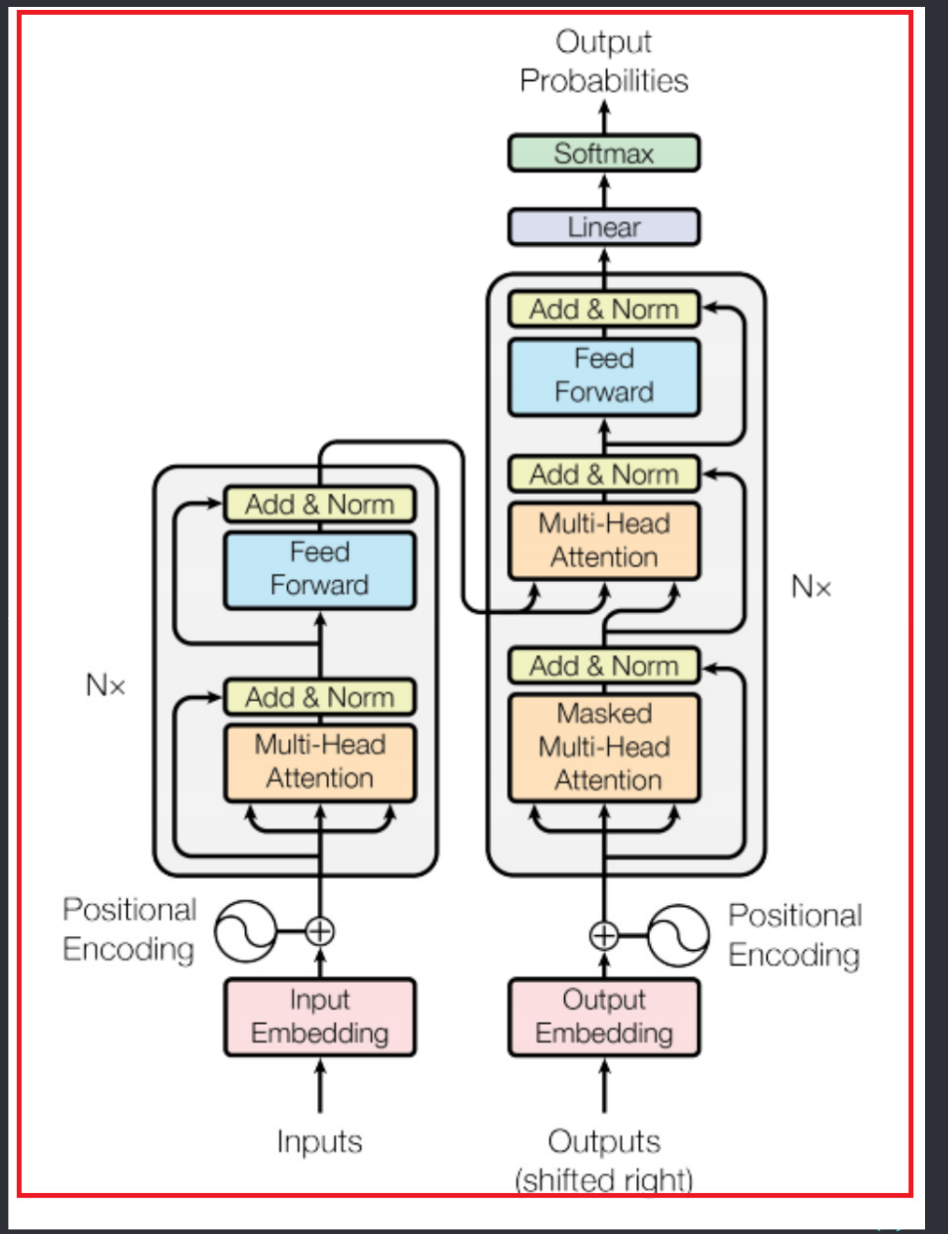
The Decoder

The Decoder contains a Positional-Embedding and a stack of Decoder-Layers.



The Transformer

The Transformer contains an Encoder, Decoder and add a final linear dense layer which converts the Decoder's output into English *token-probabilities*



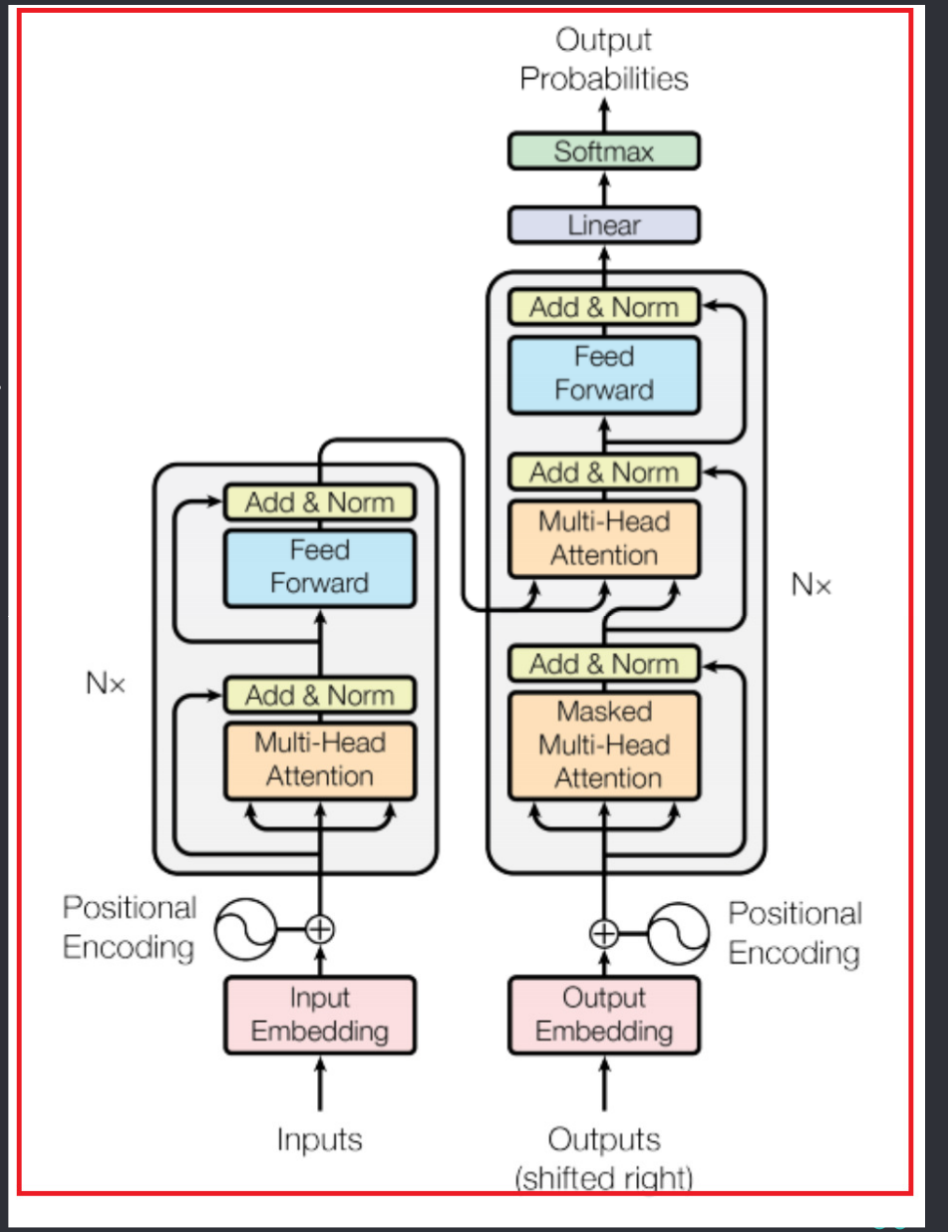
The Transformer

Example for using a “token probabilities” vector:

T=[0.01,0.01,0.01,0.96,0.01] ->

Argmax(T) = 4 ->

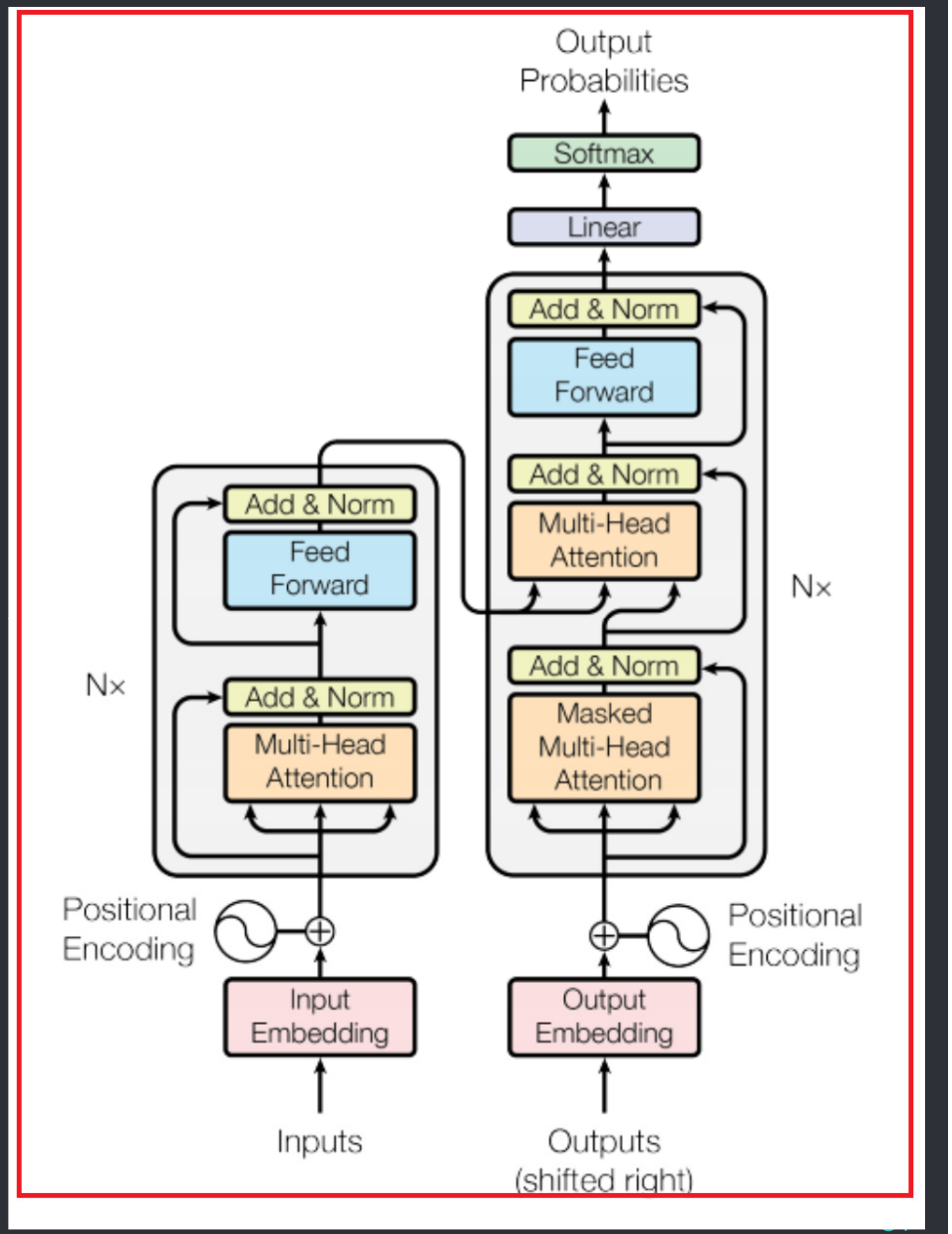
(Meaning the 4th word in our English vocabulary is the next predicted word)



The Transformer

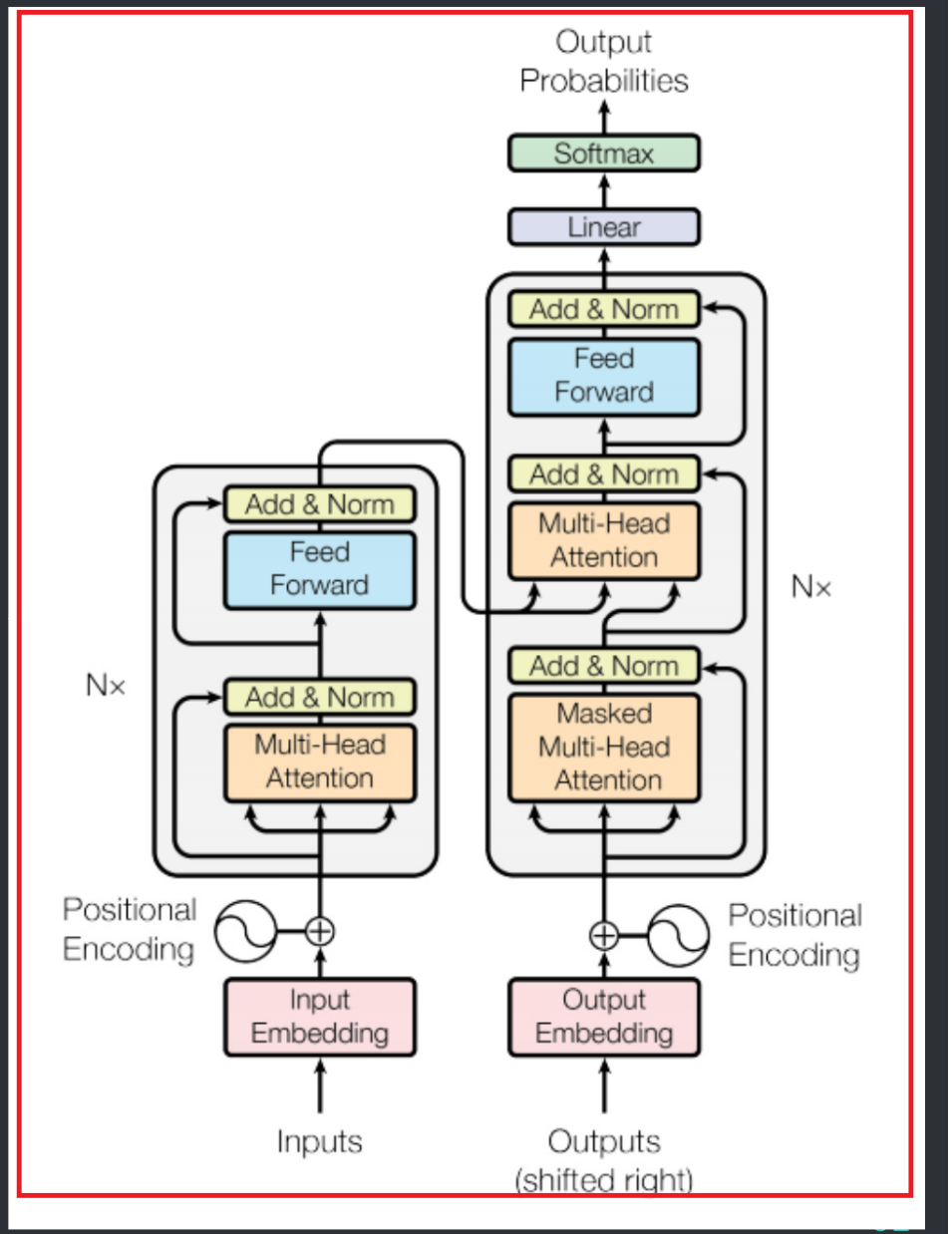
The output of the entire Transformer is the part of the English sentence which was translated so far.

Only the last word in the sentence is new and was added in the last step.



The Transformer

Each word in a sentence is represented by a *vector of probabilities* of all possible words in the *English vocabulary*.

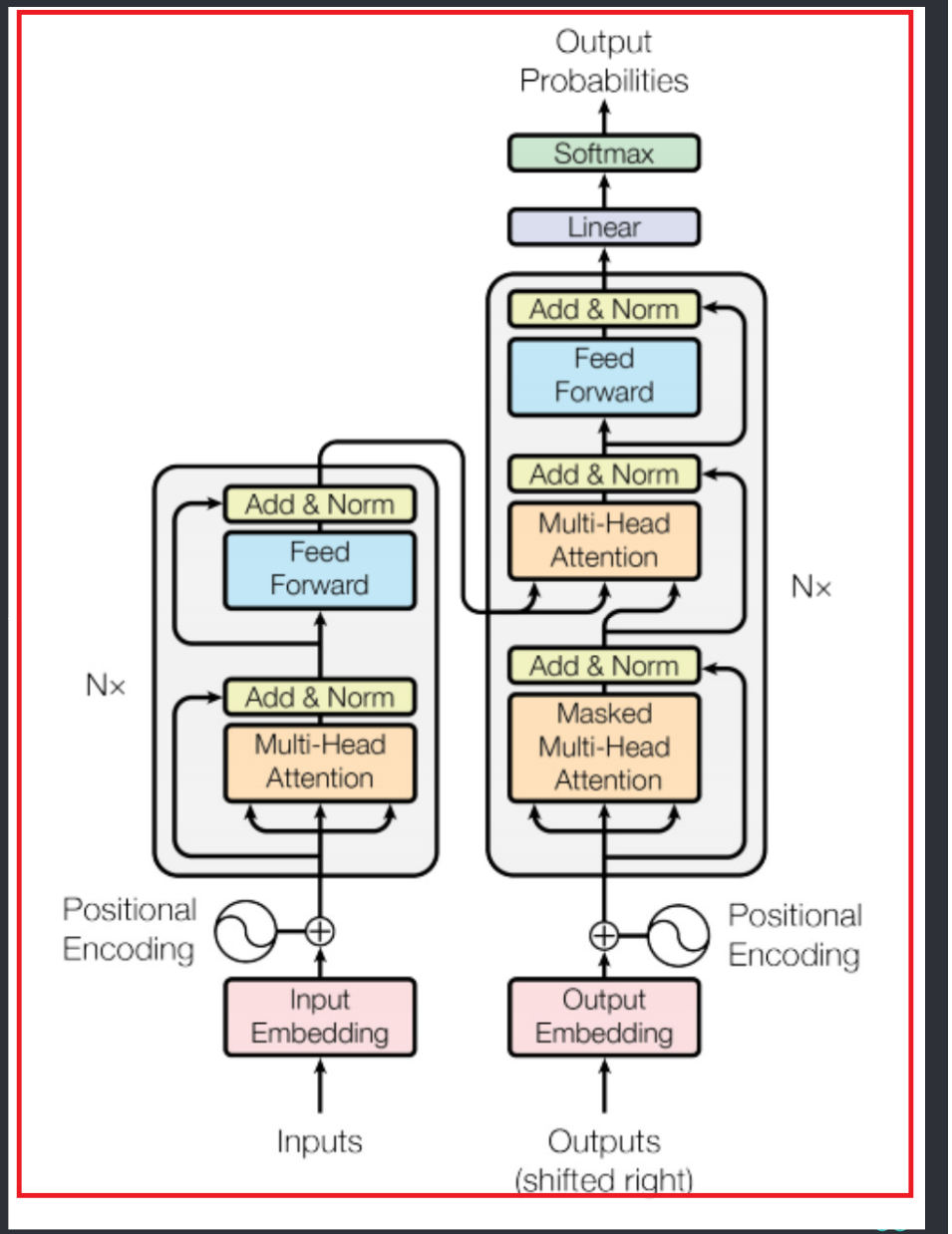


The Transformer

The total output tensor's shape is thus:

[Length (tokens number) of the English sentence,

Number of words in the English vocabulary]



The Loss

In the basic Transformer implementation, they used the cross-entropy loss function:

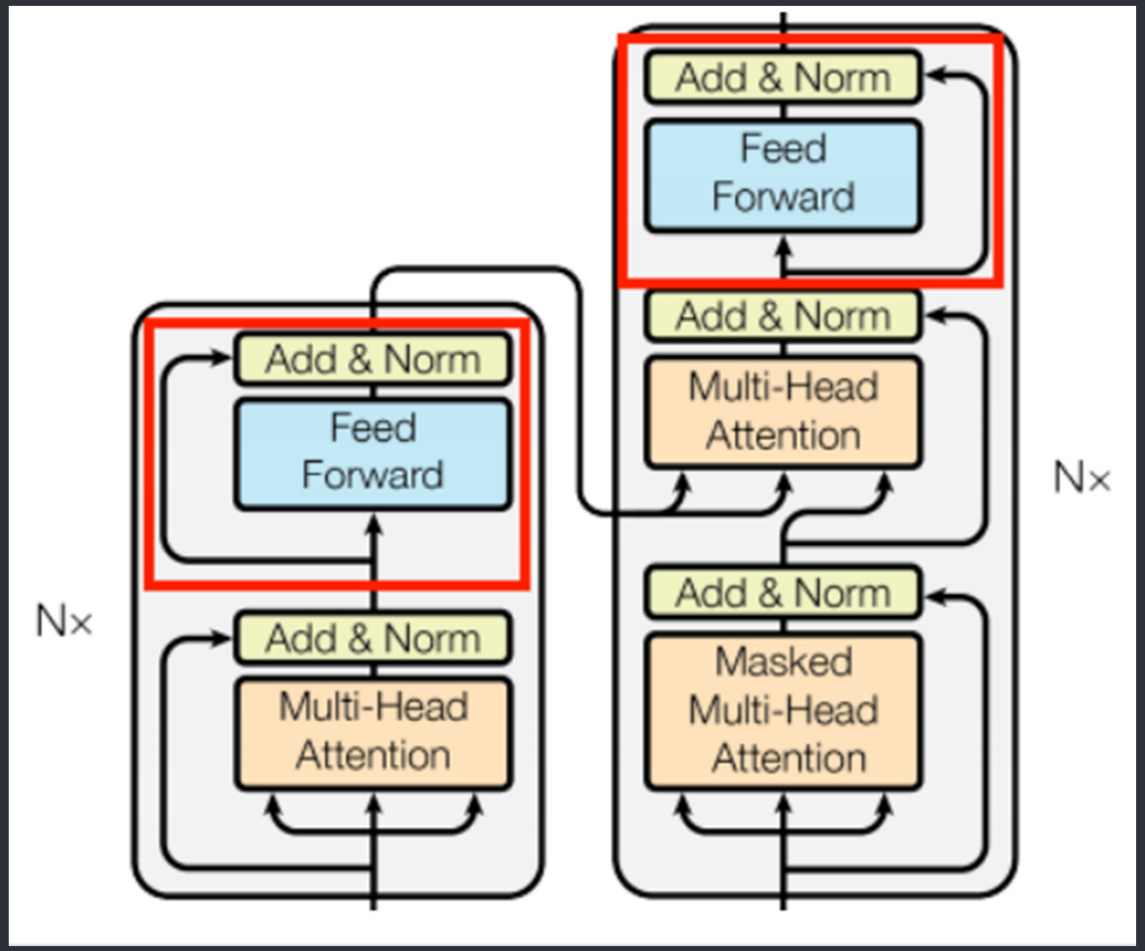
```
y_true = [1, 2]
y_pred = [[0.1, 0.9, 0], [0.05, 0.8, 0.15]]
loss = tf.keras.losses.SparseCategoricalCrossentropy()
loss(y_true,y_pred)
```

The cross-entropy loss between ‘y’ and ‘y-hat’ is:

$$-\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

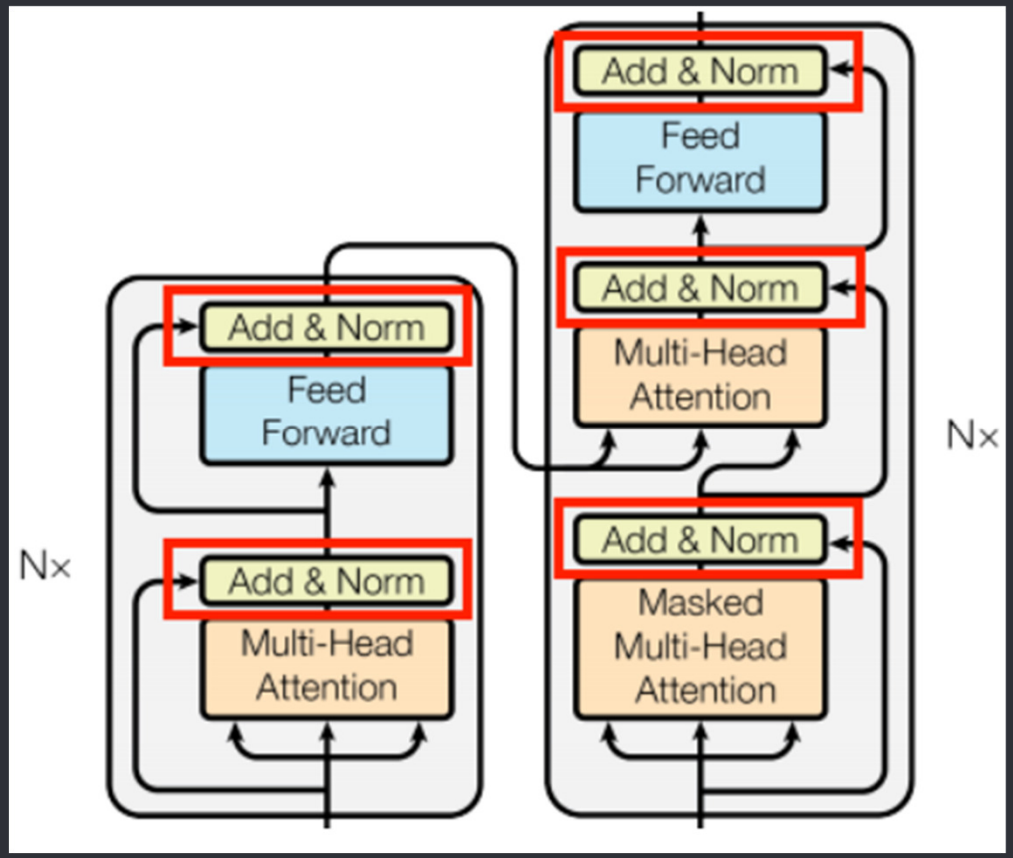
The Feed-Forward Layer

The network contains two linear dense layers with a ReLU activation in-between, as well as a dropout layer.



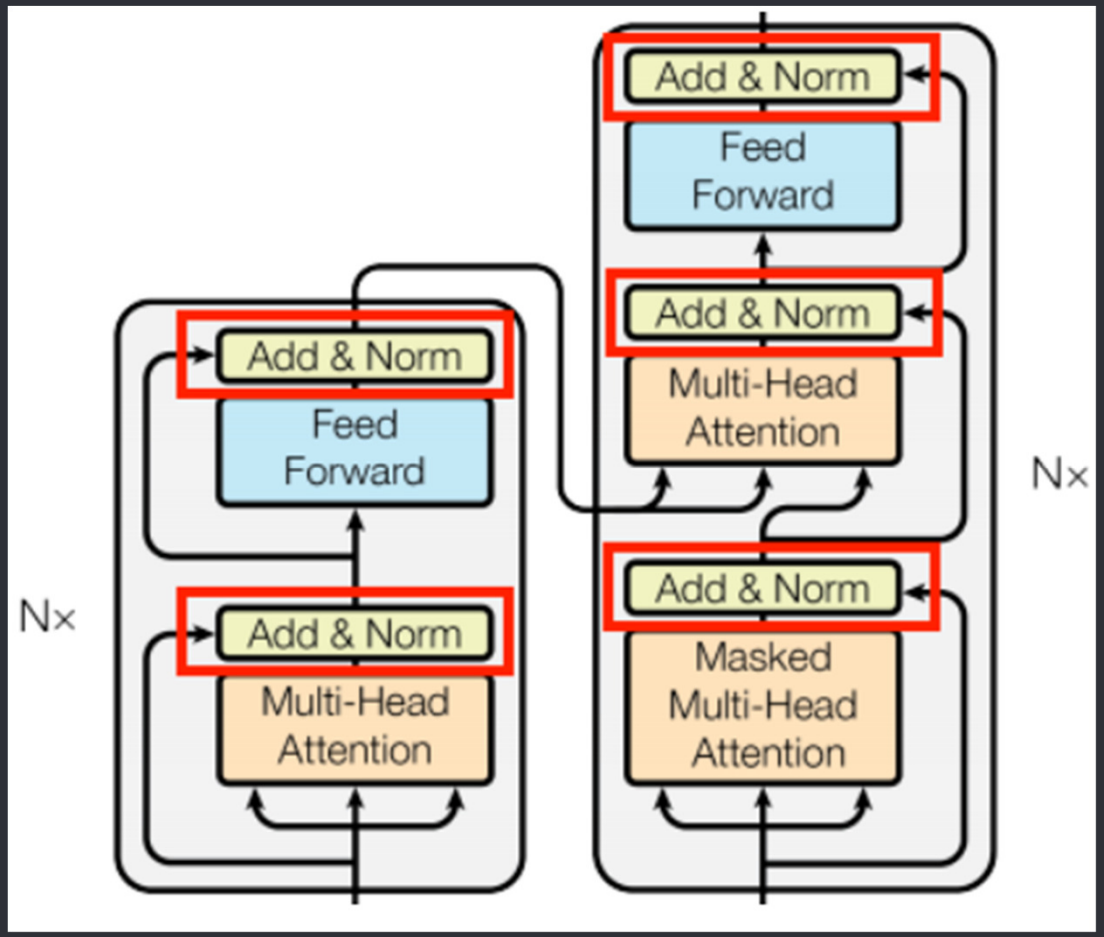
The Add & Norm Layer

The Add & Norm Layers are included to improve the training's efficiency. The residual connection provides a direct path for the gradient. It also ensures that vectors are updated by the attention layers, instead of *replaced*.



The Add & Norm Layer

The normalization maintains a reasonable size and scale for the outputs of each layer.



Hyperparameters

The base model described in the original Transformer paper used:

num_layers=6, d_{model}=512, d_{ff}=2048, num_heads=8.

In the code I've used, most hyperparameters were reduced to increase the speed:

num_layers=4, d_{model}=128, d_{ff}=512, num_heads=8.

Training

It took me **65 minutes** to train this Transformer model using **1 GPU**, over a dataset of **53,000 sentences** in Portuguese and English:

Training

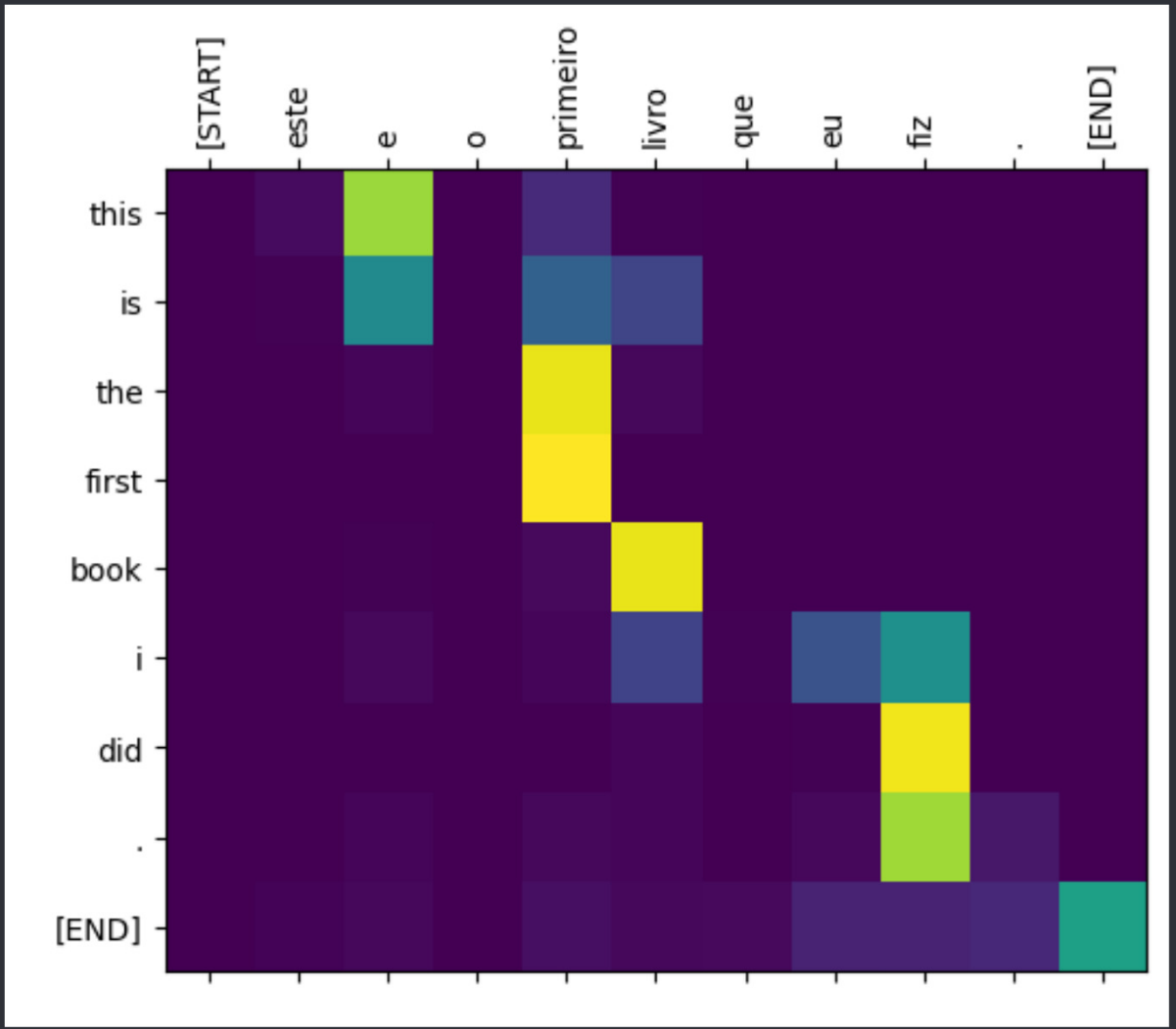
```
Epoch 1/20
810/810 [==============================] - 203s 232ms/step - loss: 6.6010 - masked_accuracy: 0.1393 - val_loss: 5.0210 - val_masked_accuracy: 0.2545
Epoch 2/20
810/810 [==============================] - 189s 233ms/step - loss: 4.5564 - masked_accuracy: 0.3010 - val_loss: 4.0271 - val_masked_accuracy: 0.3654
Epoch 3/20
810/810 [==============================] - 190s 234ms/step - loss: 3.8082 - masked_accuracy: 0.3817 - val_loss: 3.5695 - val_masked_accuracy: 0.4134
Epoch 4/20
810/810 [==============================] - 189s 233ms/step - loss: 3.2636 - masked_accuracy: 0.4422 - val_loss: 3.0256 - val_masked_accuracy: 0.4848
Epoch 5/20
810/810 [==============================] - 191s 236ms/step - loss: 2.8650 - masked_accuracy: 0.4873 - val_loss: 2.7071 - val_masked_accuracy: 0.5226
Epoch 6/20
810/810 [==============================] - 190s 234ms/step - loss: 2.5523 - masked_accuracy: 0.5256 - val_loss: 2.4814 - val_masked_accuracy: 0.5528
Epoch 7/20
810/810 [==============================] - 192s 237ms/step - loss: 2.2835 - masked_accuracy: 0.5599 - val_loss: 2.3753 - val_masked_accuracy: 0.5673
Epoch 8/20
810/810 [==============================] - 189s 233ms/step - loss: 2.0935 - masked_accuracy: 0.5857 - val_loss: 2.2713 - val_masked_accuracy: 0.5803
Epoch 9/20
810/810 [==============================] - 191s 235ms/step - loss: 1.9477 - masked_accuracy: 0.6055 - val_loss: 2.1711 - val_masked_accuracy: 0.5979
Epoch 10/20
810/810 [==============================] - 192s 236ms/step - loss: 1.8308 - masked_accuracy: 0.6228 - val_loss: 2.1240 - val_masked_accuracy: 0.6040
Epoch 11/20
810/810 [==============================] - 191s 235ms/step - loss: 1.7333 - masked_accuracy: 0.6363 - val_loss: 2.1020 - val_masked_accuracy: 0.6109
Epoch 12/20
810/810 [==============================] - 190s 234ms/step - loss: 1.6509 - masked_accuracy: 0.6492 - val_loss: 2.0854 - val_masked_accuracy: 0.6138
Epoch 13/20
810/810 [==============================] - 190s 234ms/step - loss: 1.5809 - masked_accuracy: 0.6595 - val_loss: 2.0538 - val_masked_accuracy: 0.6170
Epoch 14/20
810/810 [==============================] - 191s 236ms/step - loss: 1.5180 - masked_accuracy: 0.6692 - val_loss: 2.0520 - val_masked_accuracy: 0.6231
Epoch 15/20
810/810 [==============================] - 190s 234ms/step - loss: 1.4605 - masked_accuracy: 0.6782 - val_loss: 2.0484 - val_masked_accuracy: 0.6268
Epoch 16/20
810/810 [==============================] - 192s 236ms/step - loss: 1.4125 - masked_accuracy: 0.6855 - val_loss: 2.0523 - val_masked_accuracy: 0.6253
Epoch 17/20
810/810 [==============================] - 191s 235ms/step - loss: 1.3659 - masked_accuracy: 0.6932 - val_loss: 2.0508 - val_masked_accuracy: 0.6292
Epoch 18/20
810/810 [==============================] - 189s 233ms/step - loss: 1.3249 - masked_accuracy: 0.7000 - val_loss: 2.0557 - val_masked_accuracy: 0.6268
Epoch 19/20
810/810 [==============================] - 189s 233ms/step - loss: 1.2864 - masked_accuracy: 0.7059 - val_loss: 2.0551 - val_masked_accuracy: 0.6310
Epoch 20/20
810/810 [==============================] - 190s 235ms/step - loss: 1.2491 - masked_accuracy: 0.7123 - val_loss: 2.0563 - val_masked_accuracy: 0.6317
```

Disadvantages of Transformers

For a dataset of time-series sequences, the output for each step is calculated from the *entire history,* although it could be much more efficient to use the use a sliding window together with the hidden-state.

Results

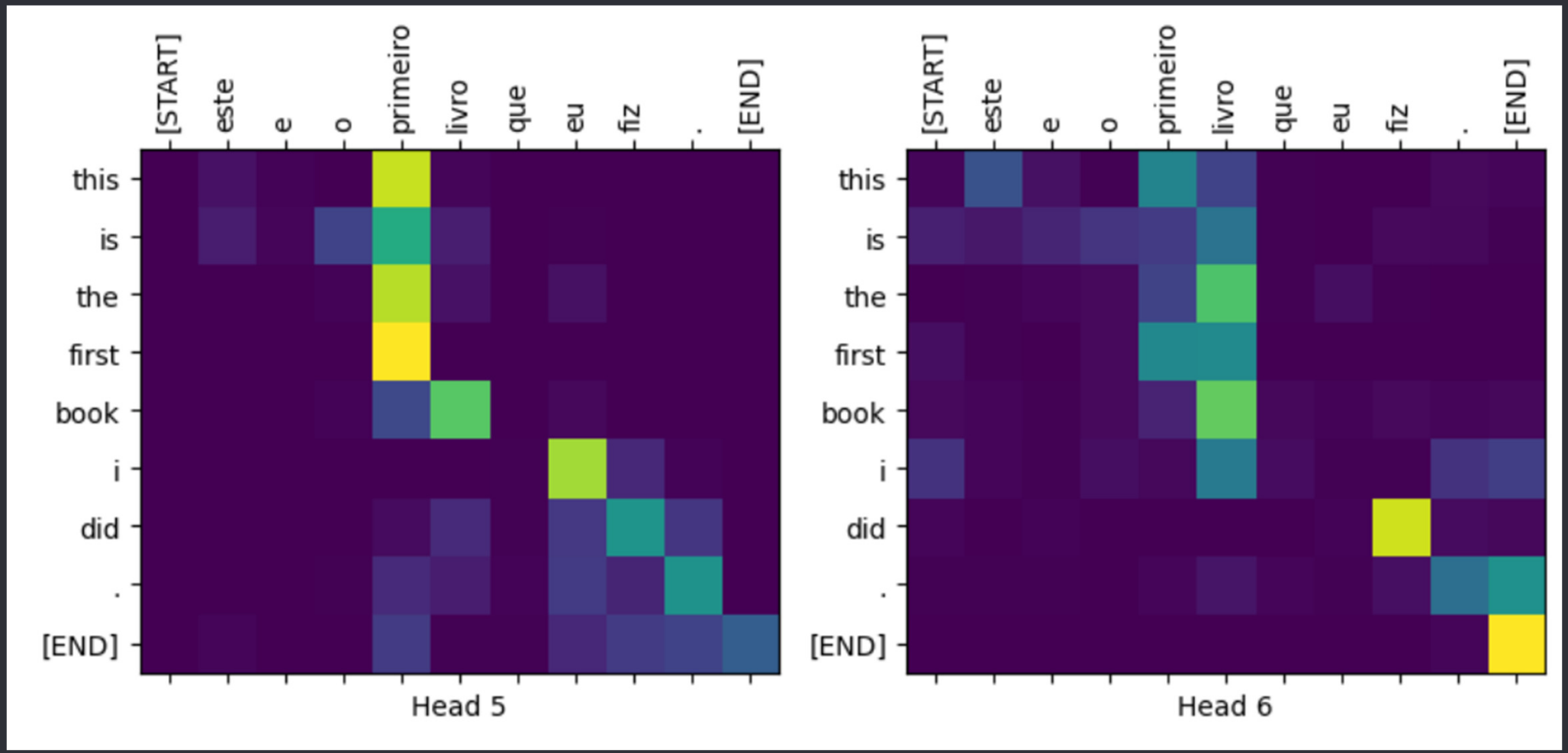
After training, the attention-weights of the Cross Attention layer were plotted:



www.tensorflow.org

Results

Attention-weights of multiple ‘heads’ from the Multi-Head Attention:



www.tensorflow.org

Dataset

In the code I've used, the dataset was derived from a set of TED talks transcripts, for comparing similar language-pairs.

Under the Open Translation project these TED talks transcripts are available for more than 2400 talks in 109 languages.

This dataset has 53K examples of sentences in multiple languages.

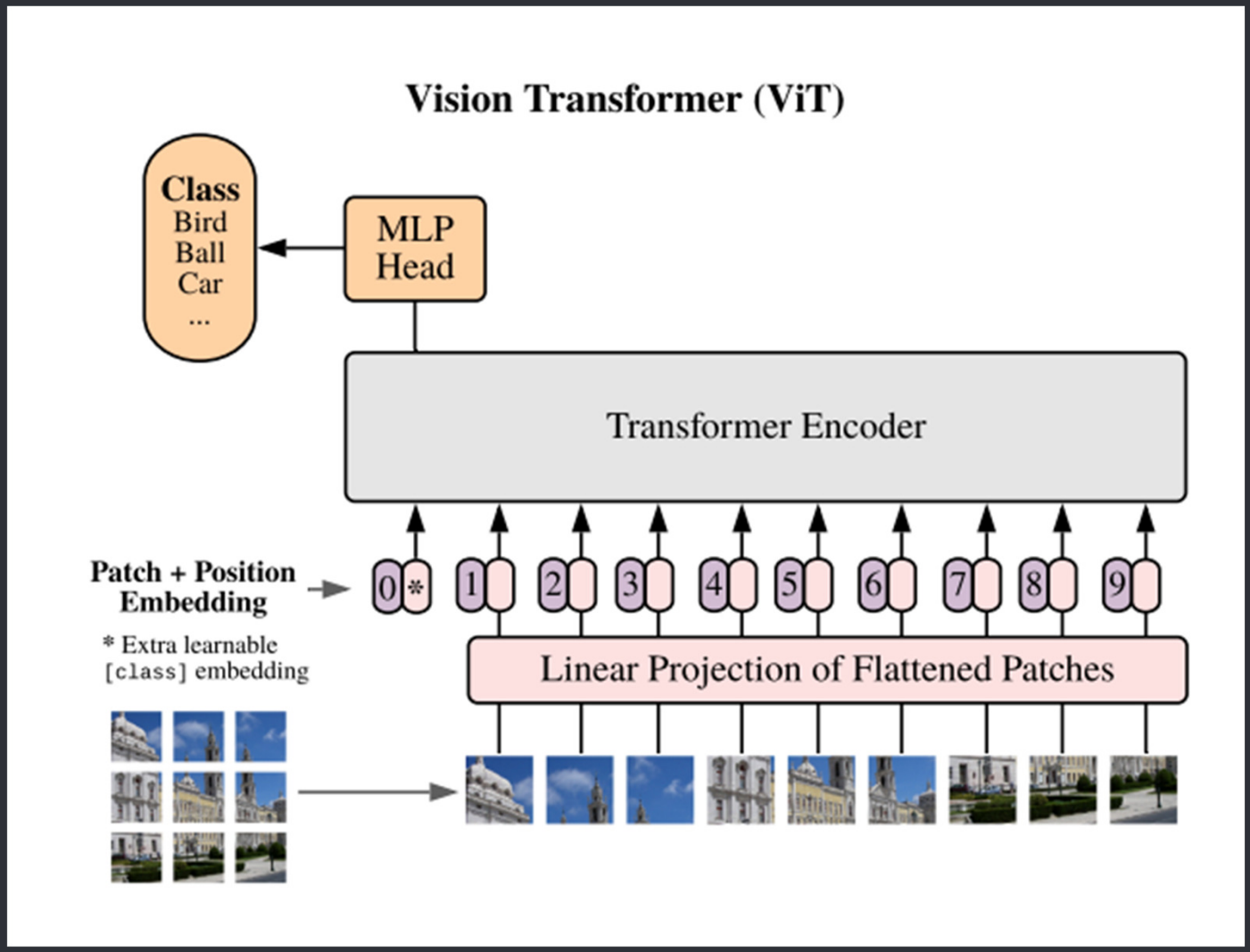
Dataset

In the original paper they trained with the standard “WMT 2014 English-German” dataset, consisting of about *4.5 million sentence* pairs.

Sentences were encoded into a vocabulary of about 37000 tokens.

Applications in Computer Vision

A major application of Transformers in Computer Vision is **ViT - Vision Transformer**:



https://paperswithcode.com/method/vision-transformer

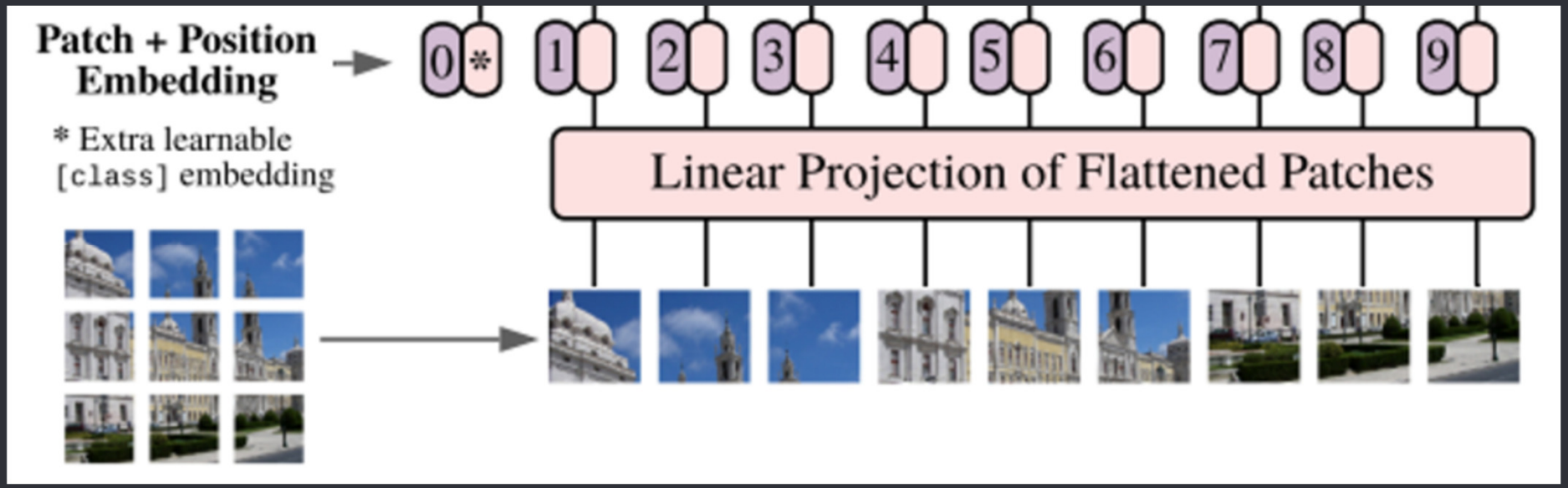
What is a Vision Transformer?

The ViT (Vision Transformer) is a model for image classification, that is based on a Transformer-like architecture, but using patches of the image instead of words.

It is targeted at various other vision-processing tasks such as Object Detection and Image Segmentation.

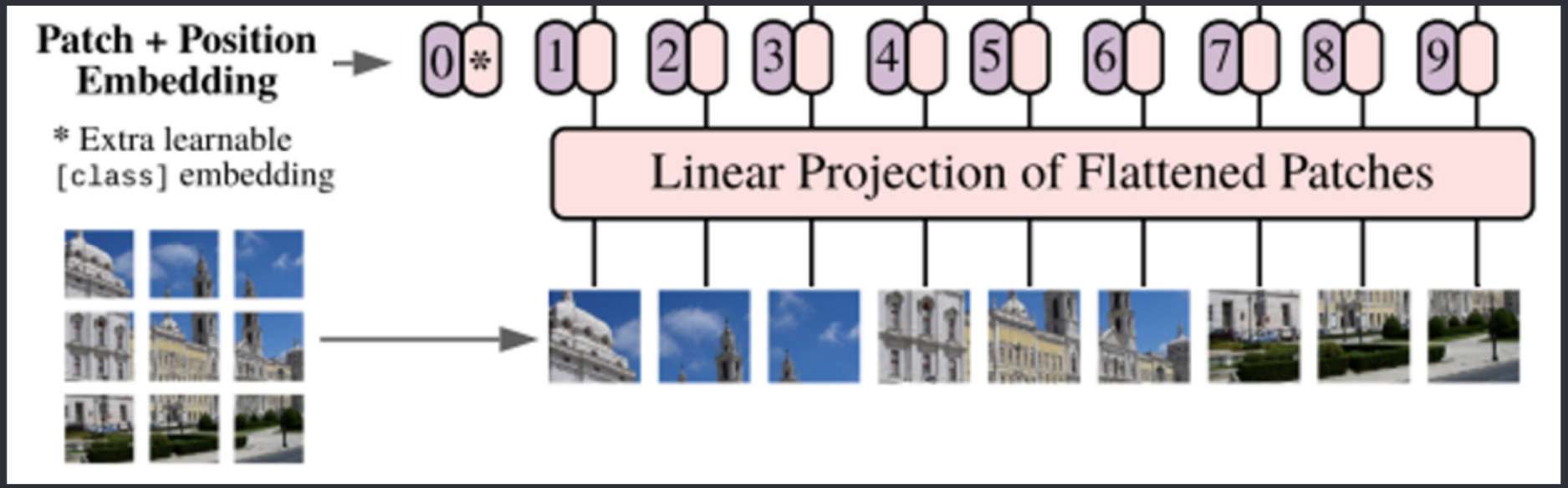
What is a Vision Transformer?

ViT represents an input image as a sequence of image patches, similar to the sequence of word used in a Transformer:



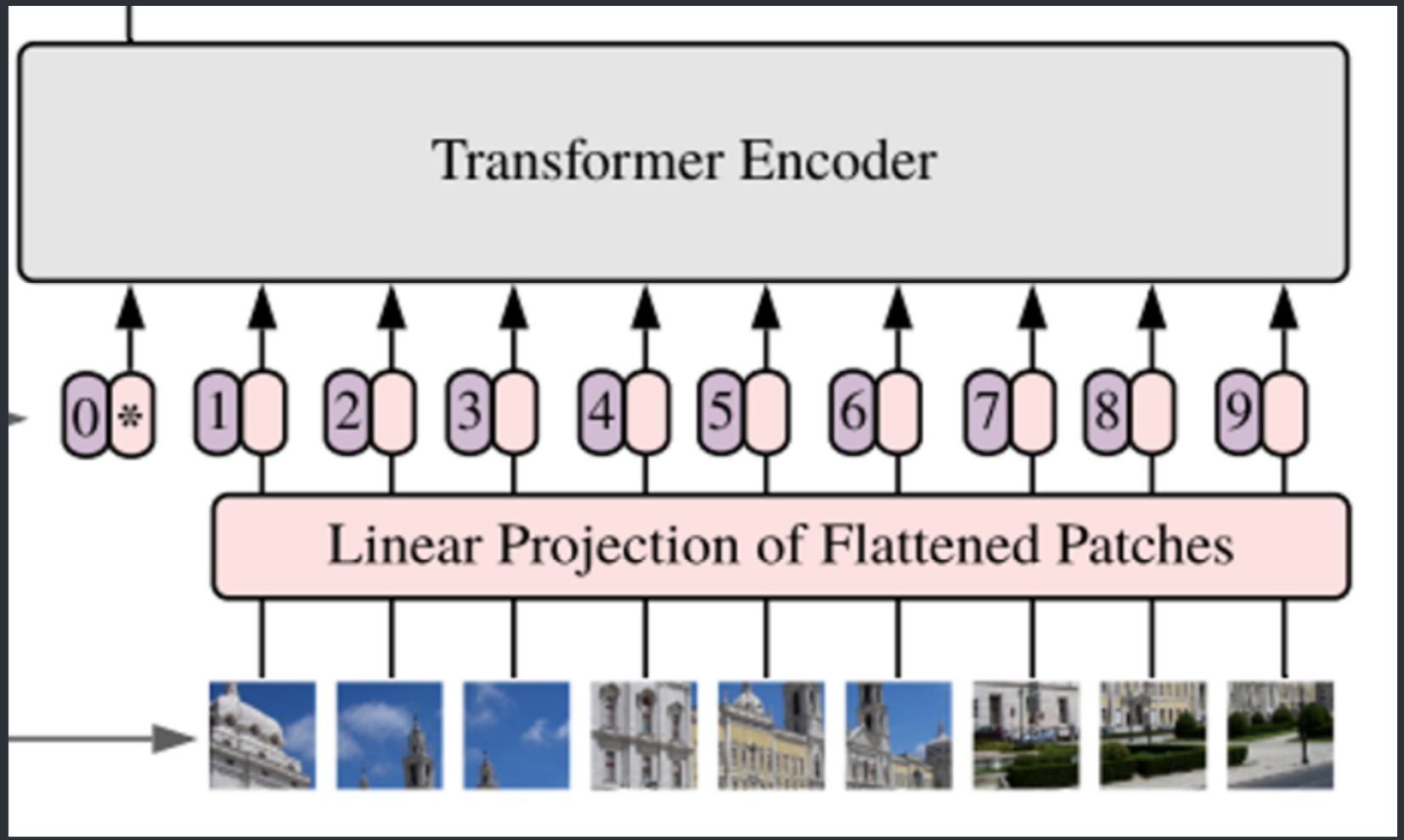
What is a Vision Transformer?

The patches are being flattened from a 2D matrix to a 1D vector, similar to the embeddings shown in the original Transformer:



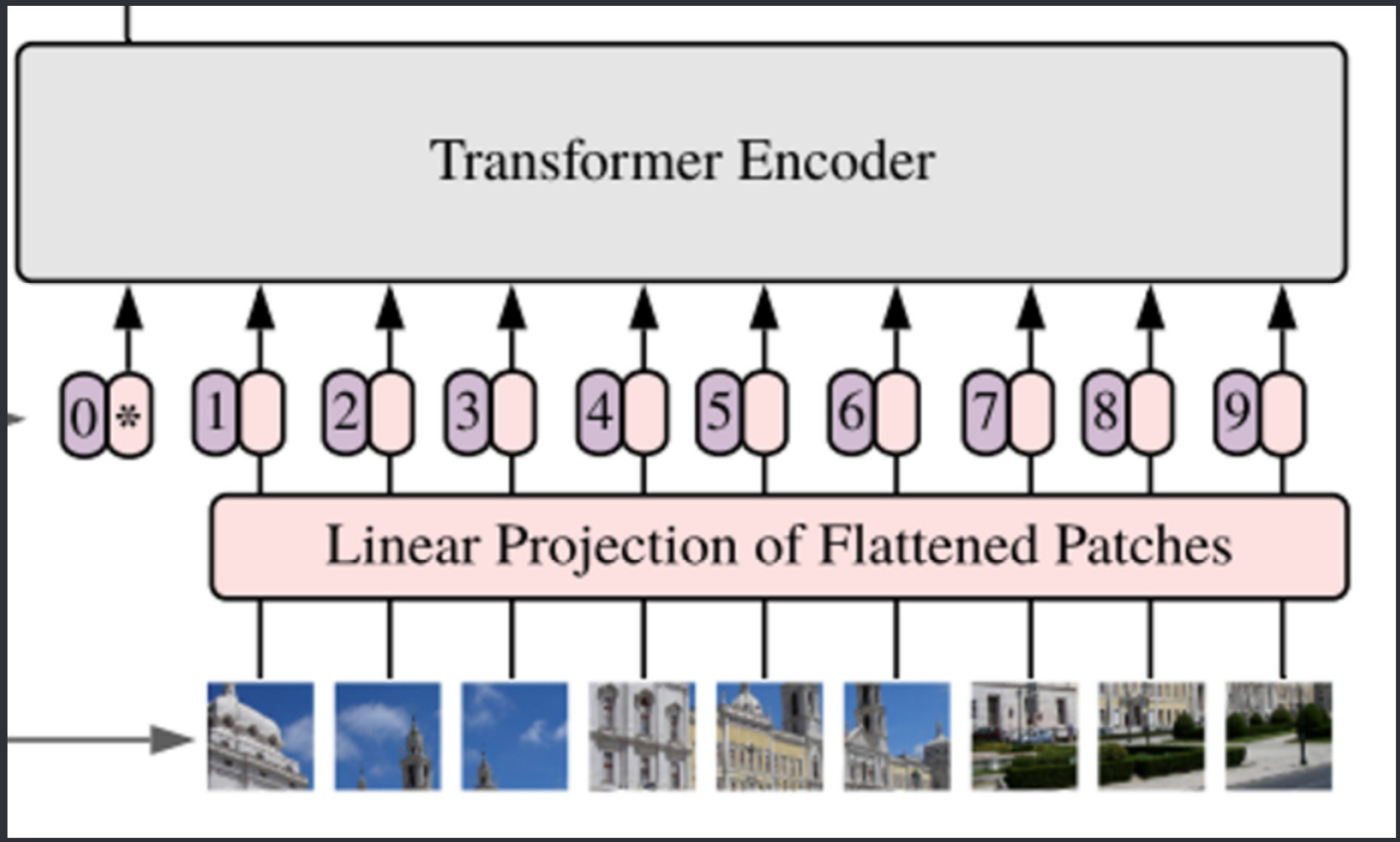
What is a Vision Transformer?

Then, these 1D vectors (the flattened patches) are being fed to a *Transformer's Encoder*:



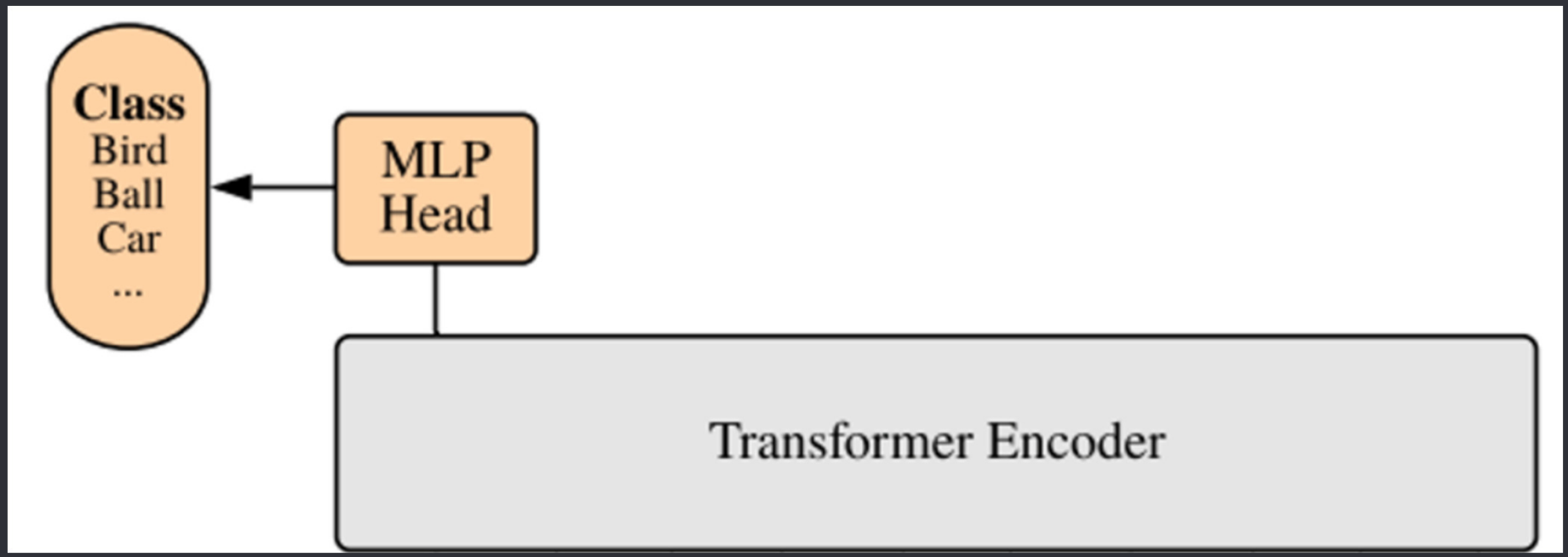
What is a Vision Transformer?

The Encoder “processes” these flattened patches using its **Attention** mechanism, and extracts “interesting correlations” between the patches:



What is a Vision Transformer?

Finally, the Encoder outputs a new representation of the image, which is passed to a standard fully-connected layer (MLP head) and outputted as a "class probabilities" vector:



Why use a Vision Transformer?

ViT demonstrates excellent performance when trained on enough data. It outperforms a state-of-the-art CNN of the same size, with ***four times fewer computational resources.***

ViT was found to be more efficient than CNNs in terms of utilizing any number of GPU resources.

However, ViT appears to only outperform CNNs in terms of accuracy, when the dataset is huge (>100M images).

Thank You