# **Transformers AI**

A transformer model is a neural network that learns context and thus meaning by tracking relationships in **sequential data** like the words in this sentence.

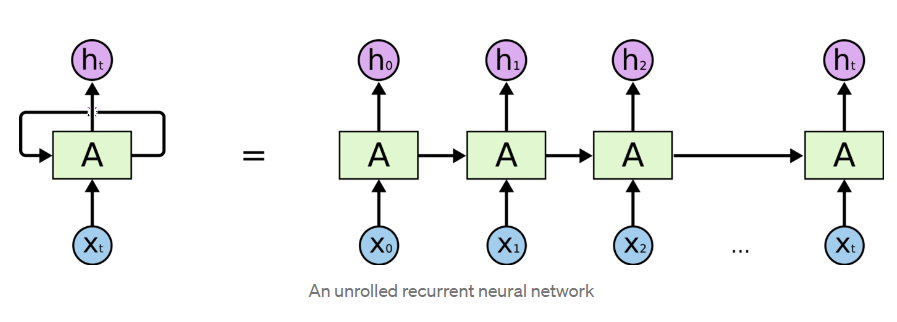
Transformer models apply an evolving set of mathematical techniques, called **attention** or **self-attention**, to detect subtle ways even distant data elements in a series influence and depend on each other.

Transformers were **developed to solve** the problem of ‘**sequence transduction’**, or neural machine translation (transformer neural network is a novel architecture that aims to **solve sequence-to-sequence tasks** while handling **long-range dependencies with ease**). That means any task that transforms an input sequence to an output sequence. This includes speech recognition, text-to-speech transformation, etc. For models to perform sequence transduction, it is necessary to have some sort of memory.

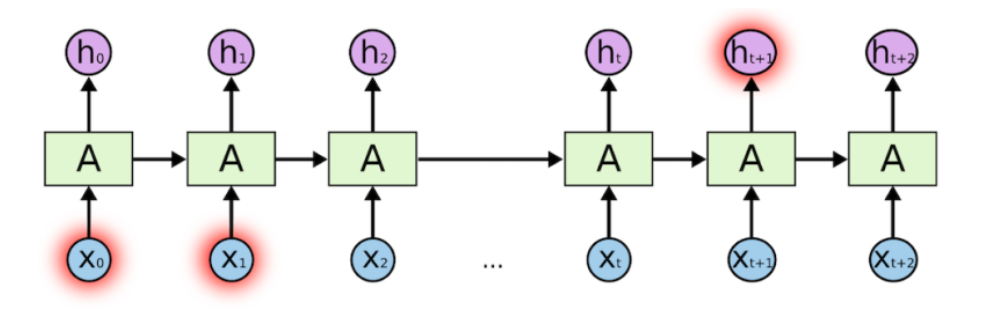
For translating sentences, a model needs to figure out some sort of dependencies and connections. **Recurrent Neural Networks (RNNs)** and **Convolutional Neural Networks (CNNs)** have been used to deal with this problem because of their properties.

## **Recurrent Neural Networks (RNN)**

RNNs are feed-forward neural networks that are rolled out over time. A RNN can be thought of as multiple copies of the same network, each network passing a message to a successor.



This chain-like nature shows that recurrent neural networks are clearly related to **sequences and lists**. In that way, **if we want to translate some text, we can set each input as the word in that text**. The Recurrent Neural Network passes the information of the previous words to the next network that can use and process that information.



Basic feed-forward networks “remember” things too, but they remember the things they learned during training. Although RNNs learn similarly during training, they also remember things learned from prior input(s) while generating output(s).

**RNNs can be used in multiple types of models:**

1. **Vector-Sequence Models —** Take fixed-sized vectors as input and output vectors of any size. For example, in image captioning, the image is the input, and the output describes the image.

2. **Sequence-Vector Model —** Take a vector of any size and output a vector of fixed size.  For example, sentiment analysis of a movie rates the review of any movie, positive or negative, as a fixed size vector.

3. **Sequence-to-Sequence Model —** The most popular and most used variant, this takes a sequence as input and outputs another sequence with variant sizes. An example of this is language translation for time series data for stock market prediction.

**An RNN has two major disadvantages:**

1. It’s slow to train.
2. Long sequences lead to ‘**vanishing gradient’** or the problem of long-term dependencies. In simple terms, **its memory is not that strong when it comes to remembering old connections.**

For example, in the sentence “The clouds are in the \_\_\_\_.” the next word should obviously be sky, as it is linked with the clouds. If the distance between clouds and the predicted word is short, so the RNN can predict it easily.

Consider another example, however: “I grew up in Germany with my parents, I spent many years there and have proper knowledge about their culture. That’s why I speak fluent \_\_\_\_.”

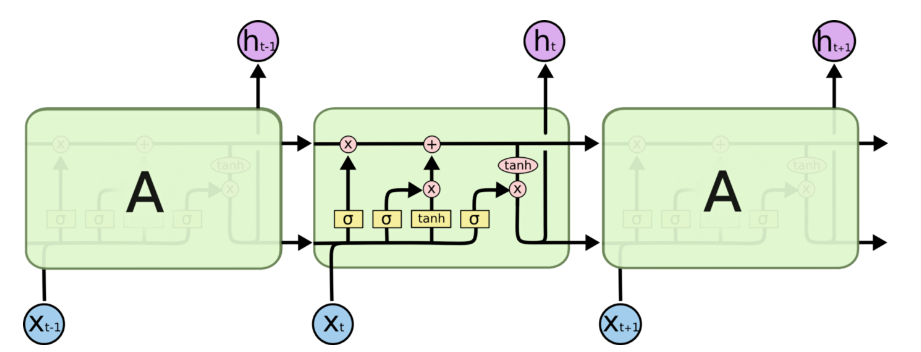
Here the predicted word is German, which is directly connected with Germany. The distance between Germany and the predicted word is longer in this case, however, so it’s difficult for the RNN to predict.

So, unfortunately, **as that gap grows, RNNs become unable to connect as their memory fades with distance**.

## **Long Short-Term Memory (LSTM)**

Long short-term memory (LSTM) is a special kind of RNN, specially made for **solving vanishing gradient problems**. They are **capable of learning long-term dependencies**. In fact, remembering information for long periods of time is practically their default behaviour without struggling.

LSTMs make small modifications to the information by multiplications and additions. With LSTMs, the information flows through a mechanism known as ‘cell states’. In this way**, LSTMs can selectively remember or forget things that are important and not so important**.



LSTM neurons, unlike the normal version, have a branch that allows passing information to skip the long processing of the current cell. This branch allows the network to retain memory for a longer period of time. **It improves the vanishing gradient problem but not terribly well**: It will do fine until 100 words, but around 1,000 words, it starts to lose its grip as the probability of keeping the context from a word that is far away from the current word being processed decreases exponentially with the distance from it. That means that when sentences are long, the model often forgets the content of distant positions in the sequence.

Further, like the simple RNN, it is also **very slow to train**, and perhaps even slower. These systems take input sequentially one by one, which **doesn’t use up GPUs very well, which are designed for parallel computation**. Not only that but there is no model of long and short range dependencies.

**To summarize, LSTMs and RNNs present these problems:**

1. Vanishing gradient
2. Slow training
3. Sequential computation inhibits parallelization
4. No explicit modeling of long and short range dependencies

## **Attention**

To solve some of these problems like vanishing gradient, researchers created a technique for paying attention to specific words.

Suppose someone gave us a book on machine learning and asked us to compile all the information about categorical cross-entropy. There are two ways of doing such a task. First, we could read the whole book and come back with the answer. Second, we could skip to the index, find the chapter on losses, go to the cross-entropy part, and just read the relevant information on categorical cross-entropy.

***Which do you think is the faster method?***

The first approach may take a whole week, whereas the second should just take a few minutes. Furthermore, our results from the first method will be vaguer and full of too much information. The second approach will more accurately meet the requirement.

***What did we do differently here?***

In the former case, we didn’t zero in on any one part of the book. In the latter method, however, we focused our attention on the losses chapter and more specifically on the part where the concept of categorical cross-entropy is explained. This second version is the way most of us humans would actually do this task.

Attention in neural networks is somewhat similar to what we find in humans. ‘**It means they focus on certain parts of the inputs while the rest gets less emphasis**’. Attention highly improved the quality of machine translation as it allows the model to focus on the relevant part of the input sequence as necessary.

This attention model is different from the classic ‘seq-to-seq model’ in two ways:

* As compared to a simple seq-to-seq model, here, the **encoder passes a lot more data to the decoder**. Previously, only the final, hidden state of the encoding part was sent to the decoder, but **now the encoder passes all the hidden states, even the intermediate ones.**
* The decoder part also does an extra step before producing its output. This step proceeds like this:

1. It checks each hidden state that it received as every hidden state of the encoder is mostly associated with a particular word of the input sentence.
2. It give each hidden state a score.
3. Each score is multiplied by its respective softmax score, thus amplifying hidden states with high scores, and drowning out hidden states with low scores.

But some of the problems that we discussed, still are not solved with RNNs using **attention.**For example, **processing inputs (words) in parallel is not possible**. **For a large corpus of text, this increases the time spent translating the text.**

## **Convolutional Neural Networks (CNN)**

Convolutional Neural Networks help solve these problems. With them we can

* **Trivial to parallelize (per layer)**
* **Exploits local dependencies**
* **Distance between positions is logarithmic**

Some of the most popular neural networks for sequence transduction, ‘Wavenet’ and ‘Bytenet’, are Convolutional Neural Networks.

The reason why **Convolutional Neural Networks can work in parallel**, is that each word on the input can be processed at the same time and does not necessarily depend on the previous words to be translated. Not only that, but the “distance” between the output word and any input for a CNN is in the order of **log(N).**

The problem is that Convolutional Neural Networks do not necessarily help with the problem of figuring out the problem of dependencies when translating sentences. That’s why **Transformers**were created, **they are a combination of both CNNs with attention.**

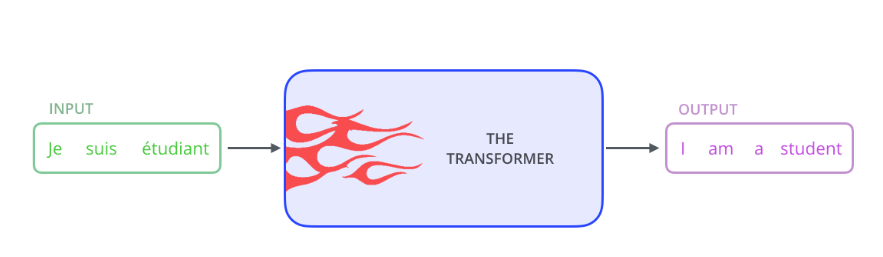
## **Transformers**

A paper called “[Attention Is All You Need](https://arxiv.org/abs/1508.04025),” published in 2017, introduced an encoder-decoder architecture based on attention layers, which the authors called the transformer.

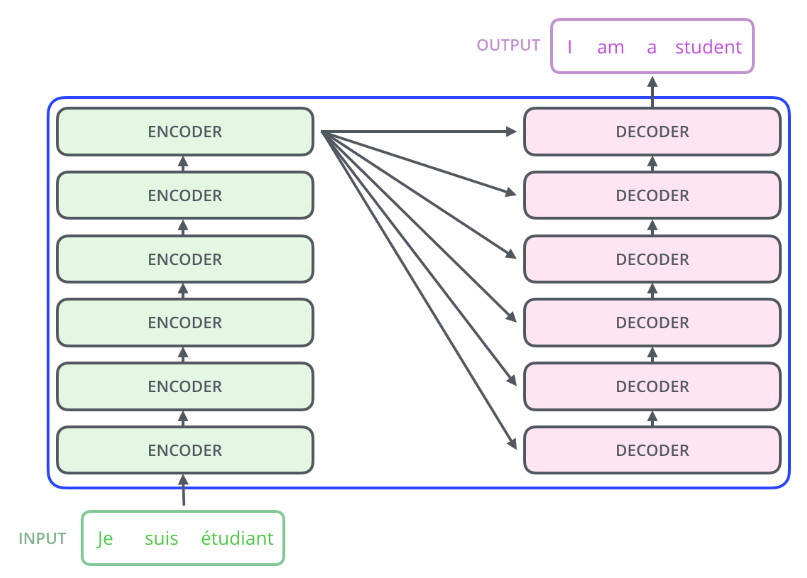
To solve the problem of parallelization, Transformers try to solve the problem by using Convolutional Neural Networks (CNN) together with **attention models,** so that input sequence can be passed parallelly so that GPU can be used effectively, and the speed of training can also be increased**.**It is also based on the ‘**multi-headed attention’ layer**, so it **easily overcomes the vanishing gradient issue**.Attention boosts the speed of how fast the model can translate from one sequence to another.

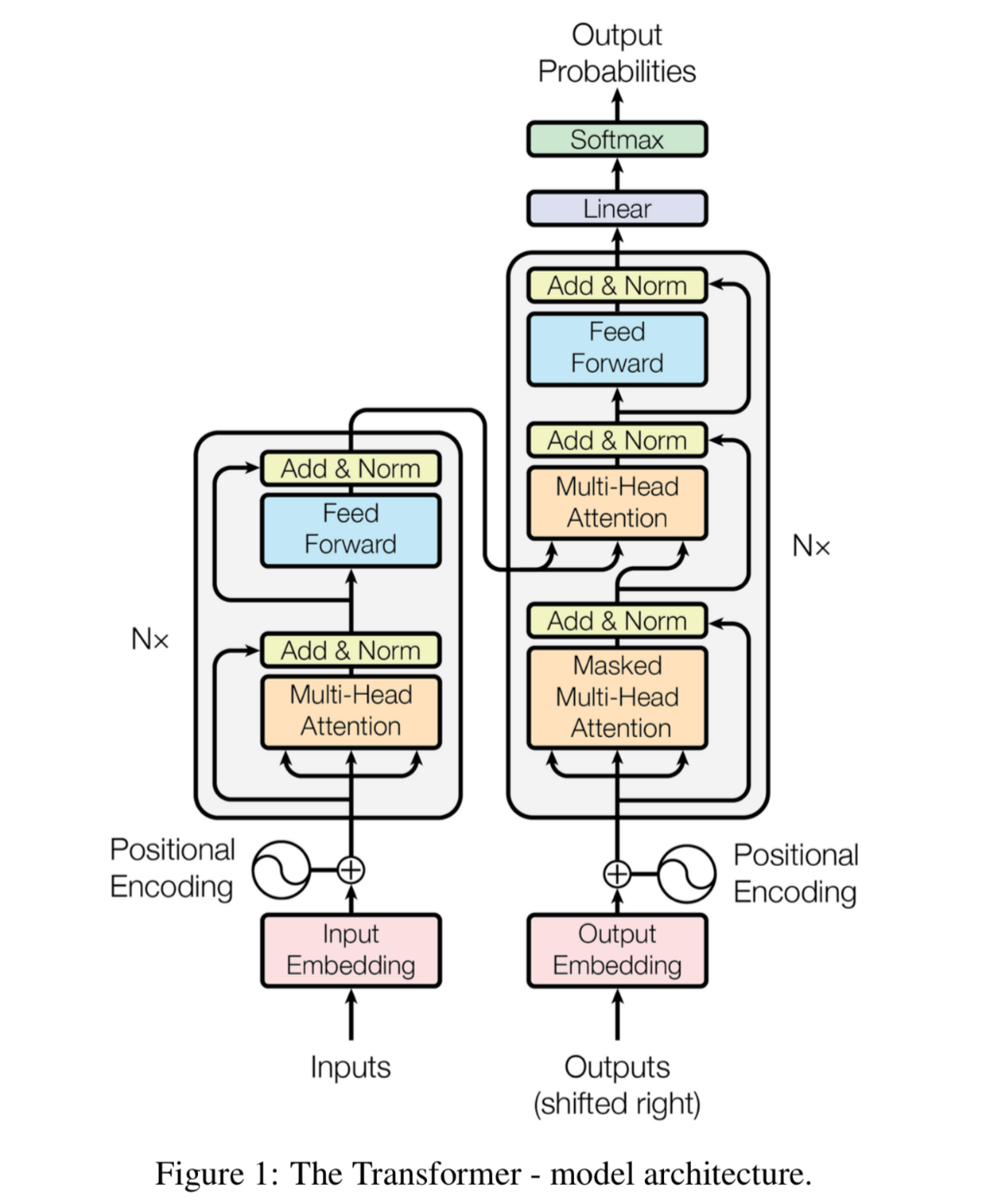
For example, in a translator made up of a simple RNN, we input our sequence or the sentence in a continuous manner, one word at a time, to generate word embeddings. As every word depends on the previous word, its hidden state acts accordingly, so we have to feed it in one step at a time. **In a transformer, however, we can pass all the words of a sentence and determine the word embedding simultaneously.**

Transformer is a model that **uses attention to boost the speed**. More specifically, it uses **‘self-attention’**.

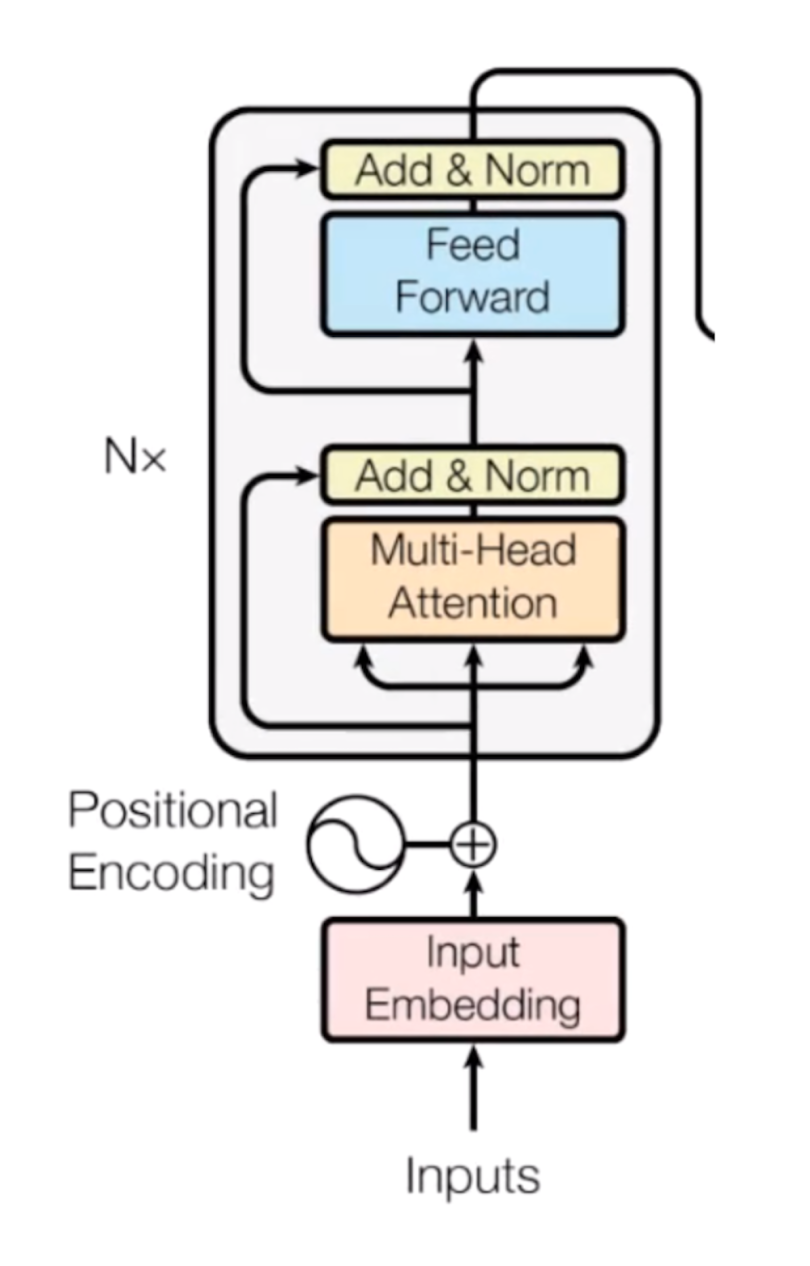


**Transformer consists of six encoders and six decoders:**

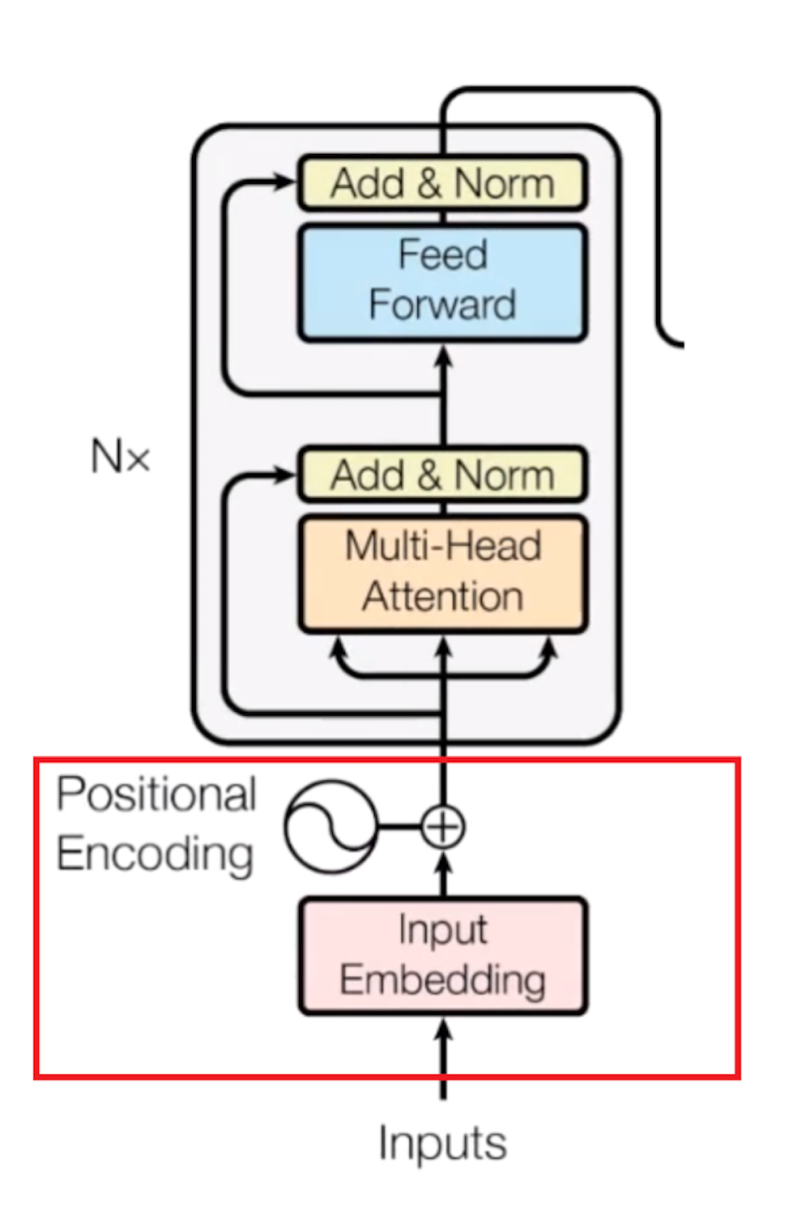




## **Encoder Block**



Computers don’t understand words. Instead, they work on numbers, **vectors or matrices**. So, we need to **convert our words to a vector**. But how is this possible? Here’s where the concept of embedding space comes into play. It’s like an **open space or dictionary where words of similar meanings are grouped together**. This is called an **embedding space**, and here every word, according to its meaning, is mapped and assigned with a particular value. Thus, we convert our words into vectors.



The inputs and outputs (target sentences) are first embedded into an n-dimensional space since we cannot use strings directly.

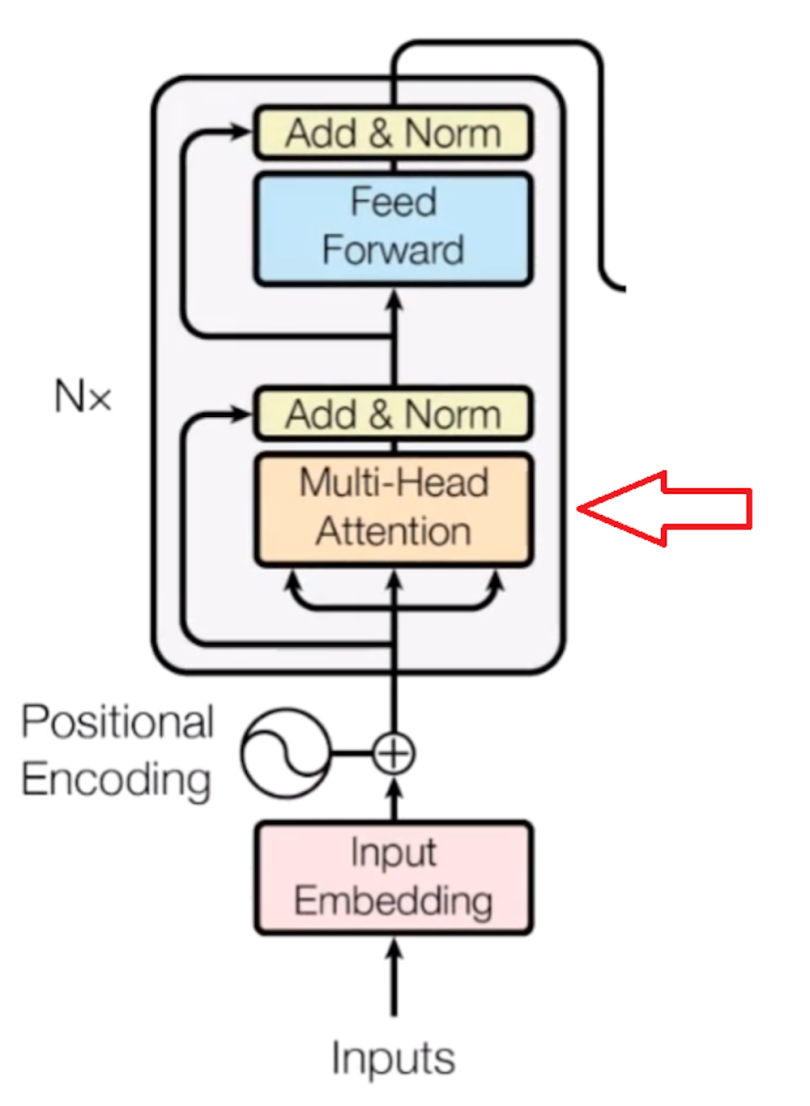
One other issue we will face is that, **in different sentences, each word may take on  different meanings**. So, to solve this issue, we use ‘**positional encoders**’. These are **vectors that give context** according to the position of the word in a sentence.

One slight but important part of the model is the **positional encoding of the different words**. Since we have no recurrent networks that can remember how sequences are fed into a model, we need to somehow give every word/part in our sequence **a relative position since a sequence depends on the order of its elements**. These positions are added to the embedded representation (n-dimensional vector) of each word.

**Word → Embedding → Positional Embedding → Final Vector, framed as Context.**

So, now that our input is ready, it goes to the encoder block.

**MULTI-HEAD ATTENTION PART**

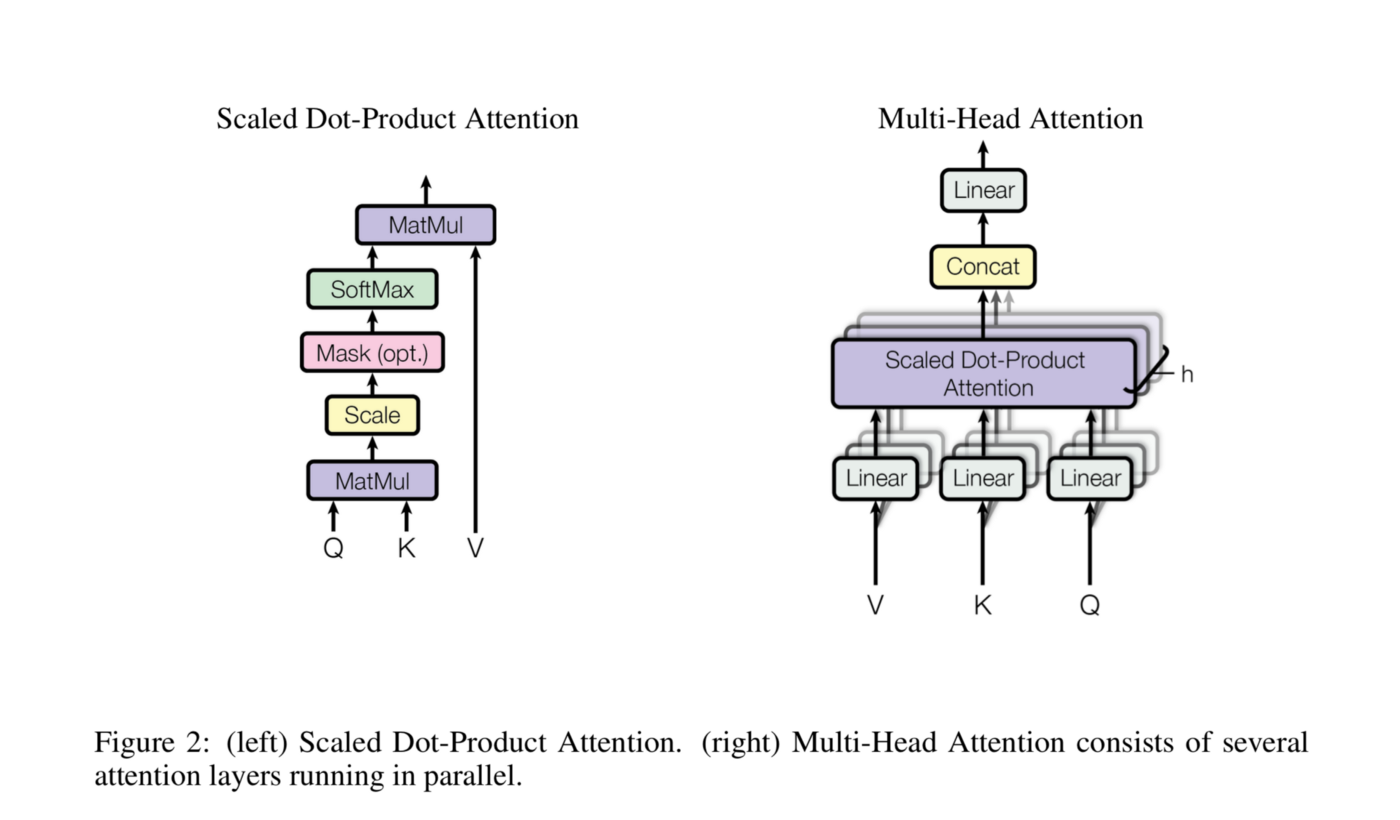


Now comes the main essence of the transformer: **self-attention**.

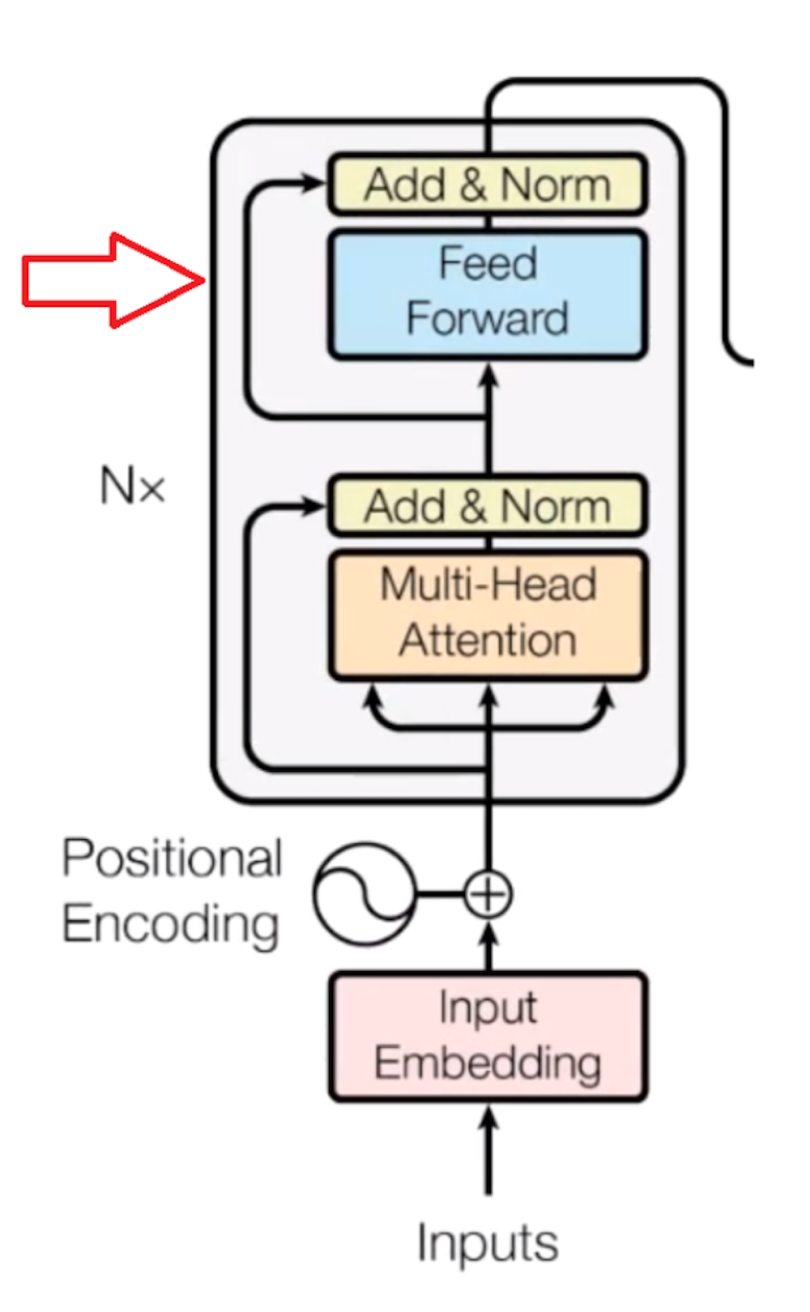
**This focuses on how relevant a particular word is with respect to other words in the sentence**. It is represented as an attention vector. For every word, we can generate an attention vector generated that captures the contextual relationship between words in that sentence.

The only problem now is that, for every word, it weighs its value much higher on itself in the sentence, but we want to know its interaction with other words of that sentence. So, we determine multiple attention vectors per word and take a weighted average to compute the final attention vector of every word.

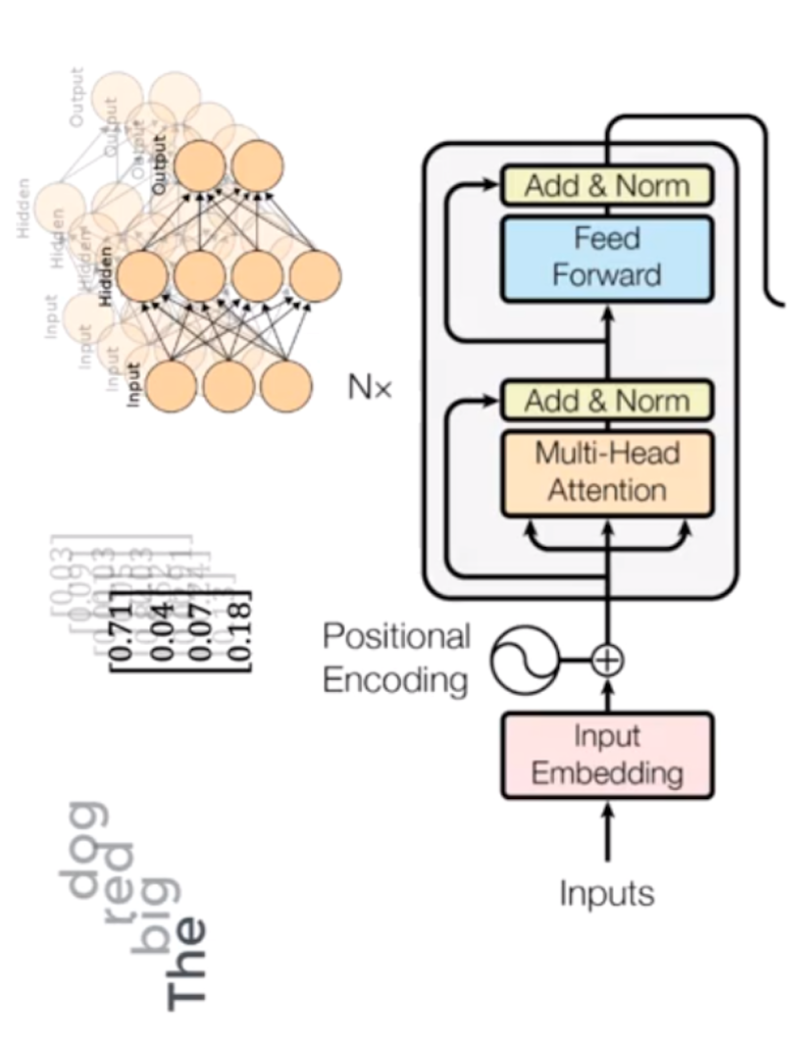
As **we are using multiple attention vectors**, this process is called the ‘**multi-head attention block’**.



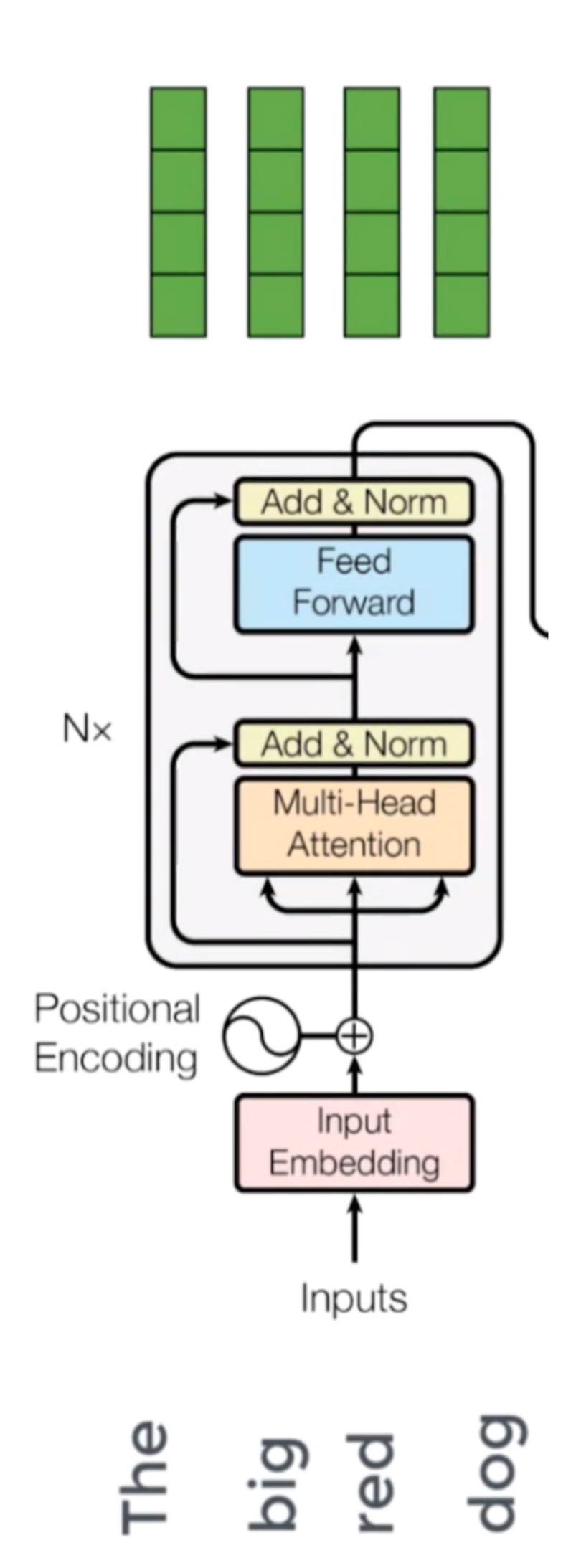
**FEED-FORWARD NETWORK**



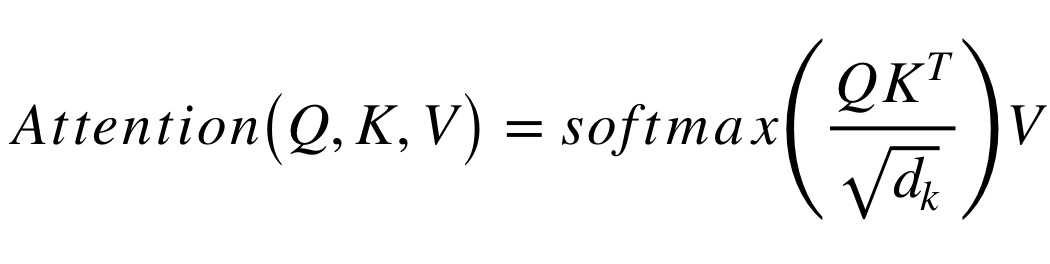
Now, the second step is the ‘**feed-forward neural network’**. A simple feed-forward neural network is applied to every attention vector **to transform the attention vectors into a form that is acceptable to the next encoder or decoder layer**.



The feed-forward network **accepts attention vectors one at a time**. And the best thing here is, unlike the case of the RNN, **each of these attention vectors is independent of one another**. So, we can apply ‘**parallelization**’ here, and that makes all the difference.

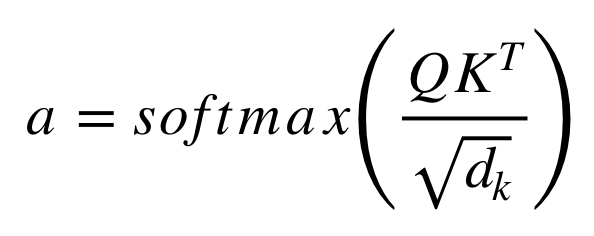


Now we can pass all the words at the same time into the encoder block and get the set of encoded vectors for every word simultaneously.



Q is a matrix that contains the query (vector representation of one word in the sequence), K are all the keys (vector representations of all the words in the sequence) and V are the values, which are again the vector representations of all the words in the sequence. For the encoder and decoder, multi-head attention modules, V consists of the same word sequence than Q. However, for the attention module that is taking into account the encoder and the decoder sequences, V is different from the sequence represented by Q.

To simplify this a little bit, we could say that the values in V are multiplied and summed with some attention-weights *a,*where our weights are defined by:



This means that the weights *a* are defined by how each word of the sequence (represented by Q) is influenced by all the other words in the sequence (represented by K). Additionally, the SoftMax function is applied to the weights *a*to have a distribution between 0 and 1.

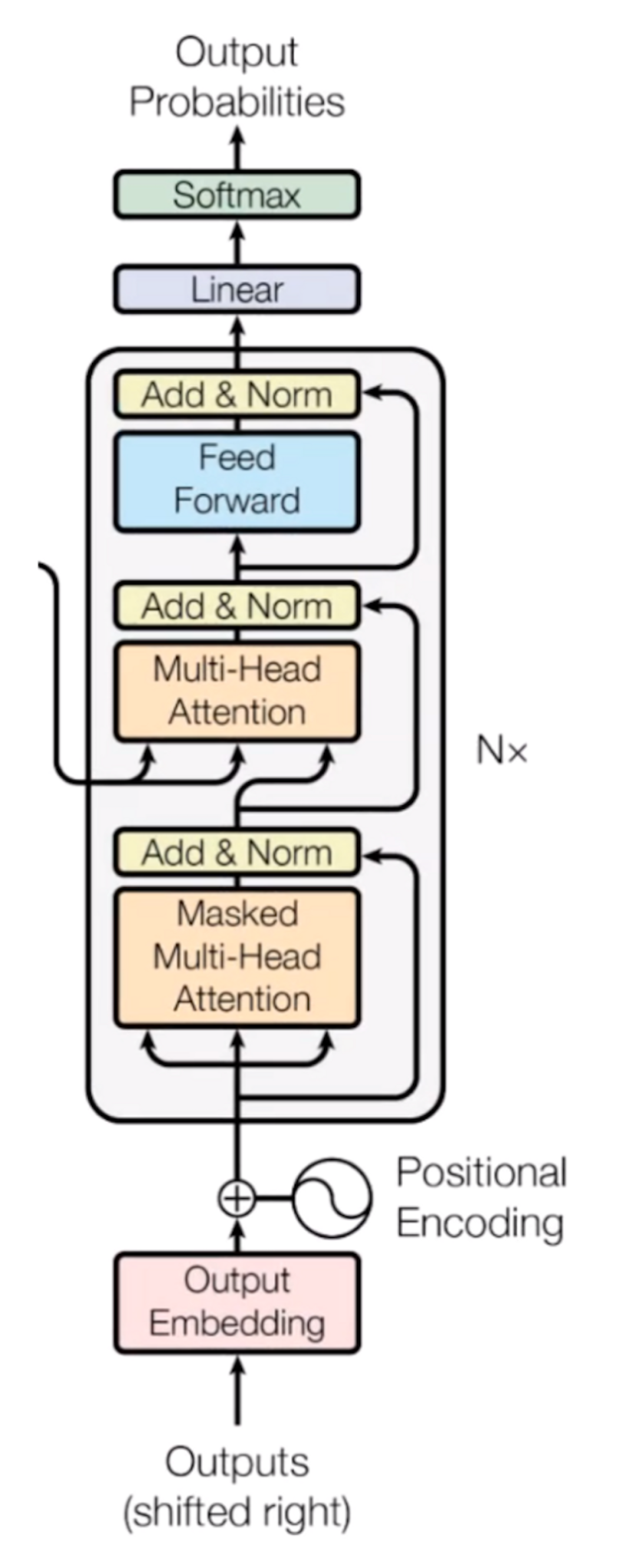
The attention mechanism is repeated multiple times with linear projections of Q, K and V. This allows the system to learn from different representations of Q, K and V, which is beneficial to the model. These linear representations are done by multiplying Q, K and V by weight matrices W that are learned during the training.

Those **matrices Q, K and V are different for each position of the attention modules in the structure** depending on whether they are in the encoder, decoder or in-between encoder and decoder.

After the multi-attention heads in both the encoder and decoder, we have a **pointwise feed-forward layer**. This little feed-forward network has identical parameters for each position, which can be described as a **separate, identical linear transformation of each element from the given sequence.**

## **Decoder Block**

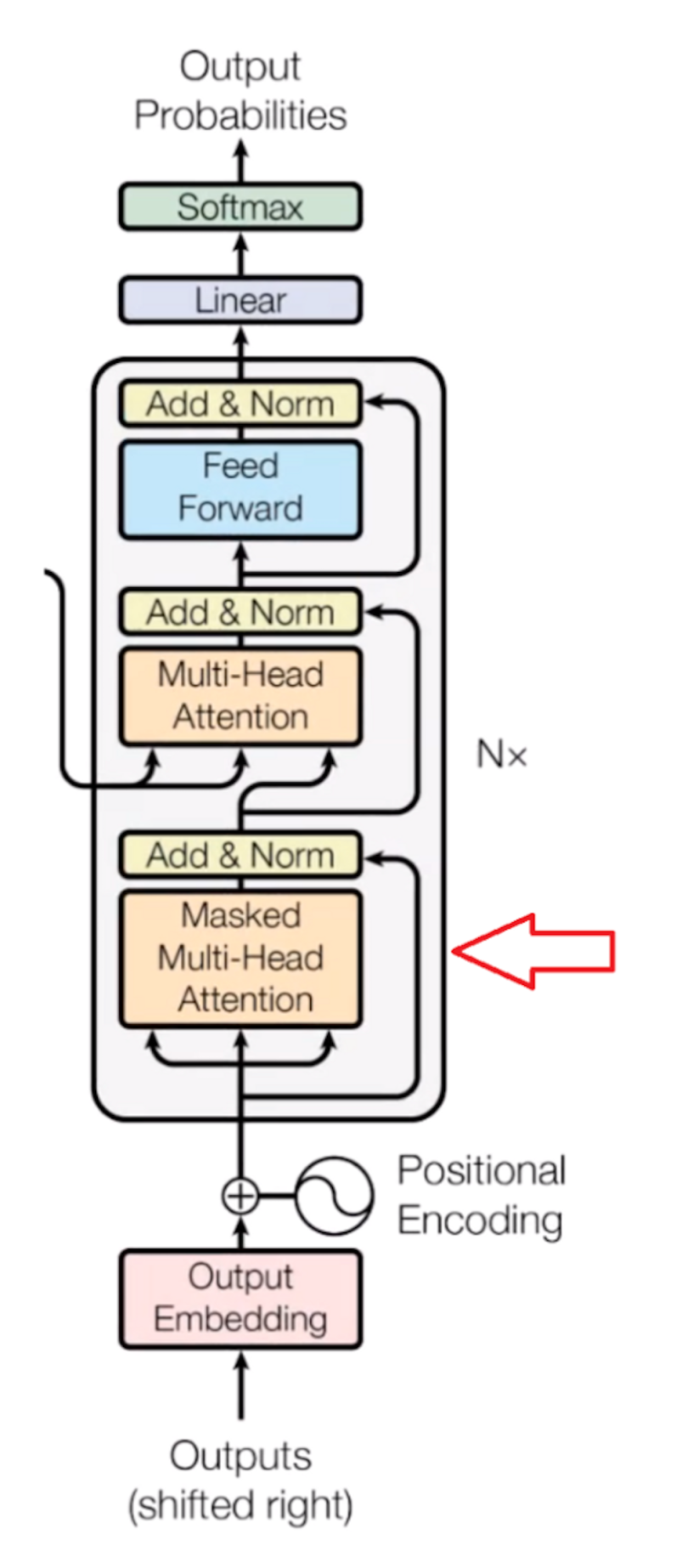
To train a model for translation tasks **we need two sentences in different languages that are translations of each other**. Once we have a lot of sentence pairs, we can start training our model. Now, if we’re training a translator for English to French, for training, **we need to give an English sentence along with its translated French version for the model to learn**. So, our **English sentences pass through encoder block**, and **French sentences pass through the decoder block**.



At first, we have the embedding layer and positional encoder part, which changes the words into respective vectors. This is similar to what we saw in the encoder part.

**MASKED MULTI-HEAD ATTENTION PART**

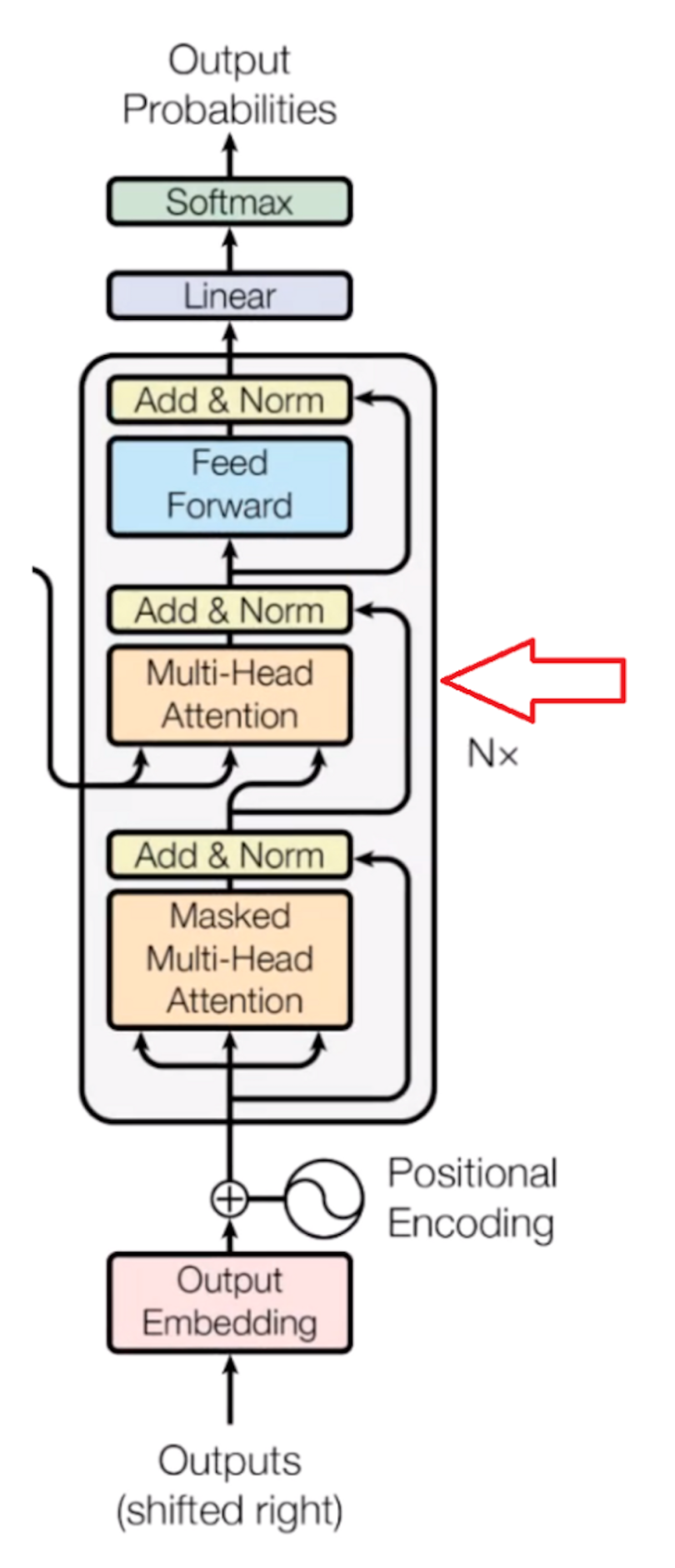
Now it will **pass through the ‘self-attention’ block**, where attention vectors are generated for every word in the French sentences to represent how much each word is related to every word in the same sentence, just like we saw in the encoder part.



But this block is called the ‘**masked multi-head attention block**’. First, we need to know how the learning mechanism works. When we provide an English word, it will be translated into its French version using previous results. It will then match and compare with the actual French translation that we fed into the decoder block. After comparing both, it will update its matrix value. This is how it will learn after several iterations.

What we observe is that **we need to hide the next French word** so that, at first**, it will predict the next word itself using previous results without knowing the real translated word**. For learning to take place, it will make no sense if it already knows the next French word. Therefore, **we need to hide (or mask)** it.

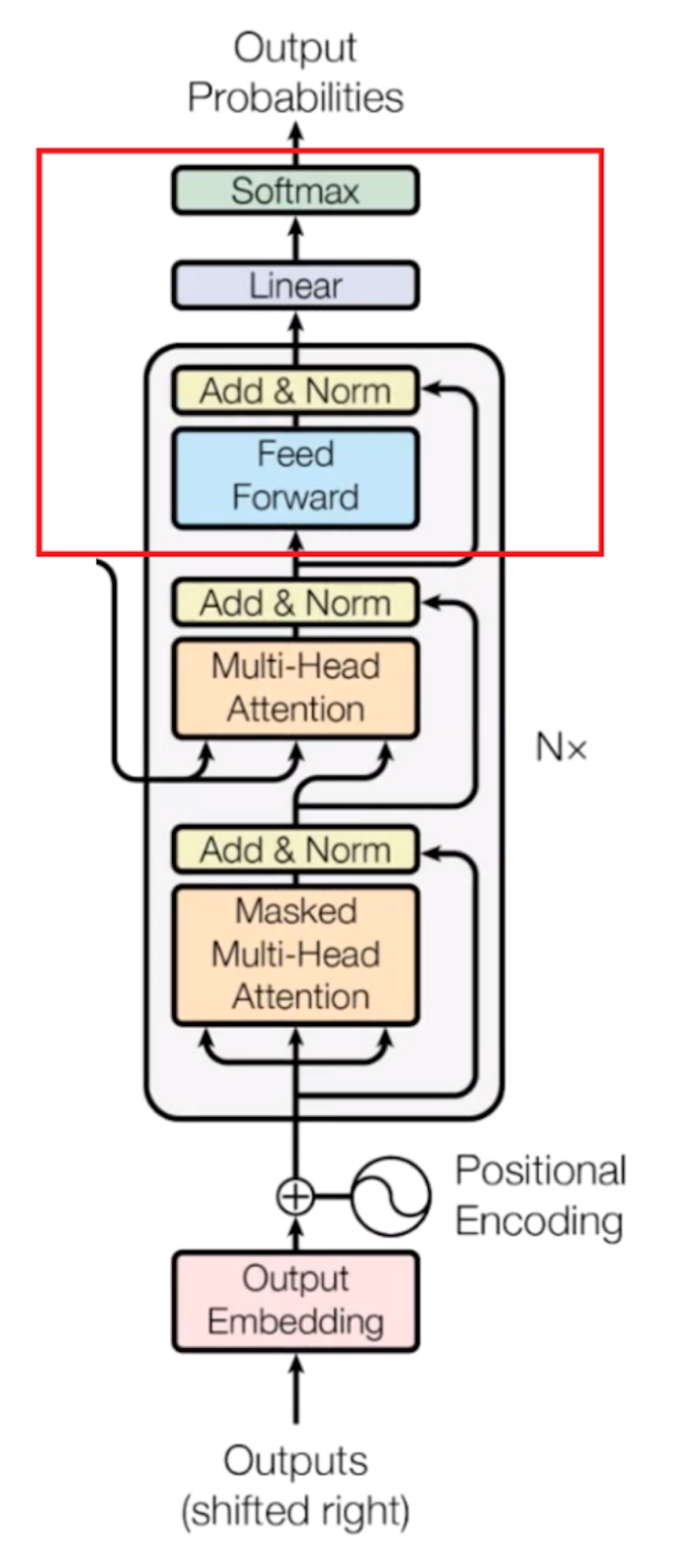
We can take any word from the English sentence, **but we can only take the previous word of the French sentence for learning purposes**. So, **while performing parallelization with the matrix operation**, we need to make sure that the **matrix will mask the words appearing later** by transforming them into zeroes so that the attention network can’t use them.



Now, the resulting attention vectors from the previous layer and the vectors from the encoder block are **passed into another multi-head attention block**. This is where the results from the encoder block also come into the picture. In the diagram, the results from the encoder block also clearly come here. That’s why it is called the **encoder-decoder attention block**.

Since we have one vector of every word for each English and French sentence, **this block actually does the mapping of English and French words and finds out the relation between them**. So, this is the part where the main **English to French word mapping** happens.

The **output of this block is attention vectors for every word in the English and French sentences**. Each vector represents the relationship with other words in both languages.



Now, if we **pass each attention vector into a feed-forward unit**, it will make the **output vectors into a form that is easily acceptable** by another **decoder block or a linear layer**. A linear layer is another feed-forward layer that expands the dimensions into numbers of words in the French language after translation.

Now it is **passed through a softmax layer** that **transforms the input into a probability distribution**, which is human interpretable, and the resulting word is produced with the highest probability after translation.

**Comments**

1. In addition to the right-shifting, the Transformer **applies a mask to the input** in the first multi-head attention module **to avoid seeing potential ‘future’ sequence elements**. This is **specific to the Transformer architecture** because we do not have RNNs where we can input our sequence sequentially. Here, we input everything together and if there were no mask, the **multi-head attention would consider the whole decoder input sequence at each position**.
2. The **process of feeding the correct shifted input into the decoder** is also called ‘**Teacher-Forcing**’.
3. The **decoder** operates similarly, but **generates one word at a time**, from left to right. It attends not only to the other previously generated words but also to the final representations generated by the encoder.