

# Transformers Explained Visually — Not Just How, but Why They Work So Well

A Gentle Guide to how the Attention Score calculations capture relationships between words in a sequence, in Plain English.

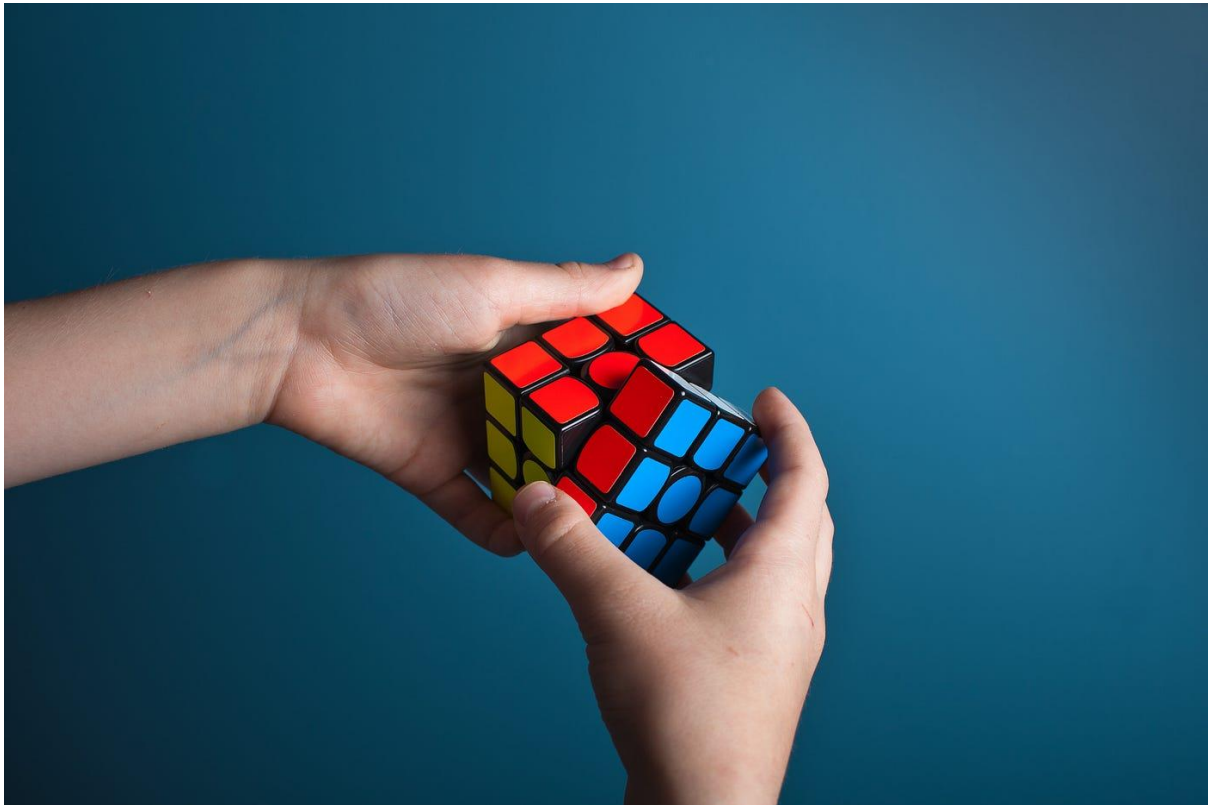


Photo by [Olav Ahrens Røtne](#) on [Unsplash](#)

Transformers have taken the world of NLP by storm in the last few years. Now they are being used with success in applications beyond NLP as well.

The Transformer gets its powers because of the Attention module. And this happens because it captures the relationships between each word in a sequence with every other word.

But the all-important question is *how* exactly does it do that?

In this article, we will attempt to answer that question, and understand *why* it performs the calculations that it does.

I have a few more articles in my series on Transformers. In those articles, we learned about the Transformer architecture and walked through their operation during training and inference, step-by-step. We also explored under the hood and understood exactly how they work in detail.

Our goal is to understand not just how something works but why it works that way.

1. [Overview of functionality](#) (*How Transformers are used, and why they are better than RNNs. Components of the architecture, and behavior during Training and Inference*)
2. [How it works](#) (*Internal operation end-to-end. How data flows and what computations are performed, including matrix representations*)
3. [Multi-head Attention](#) (*Inner workings of the Attention module throughout the Transformer*)

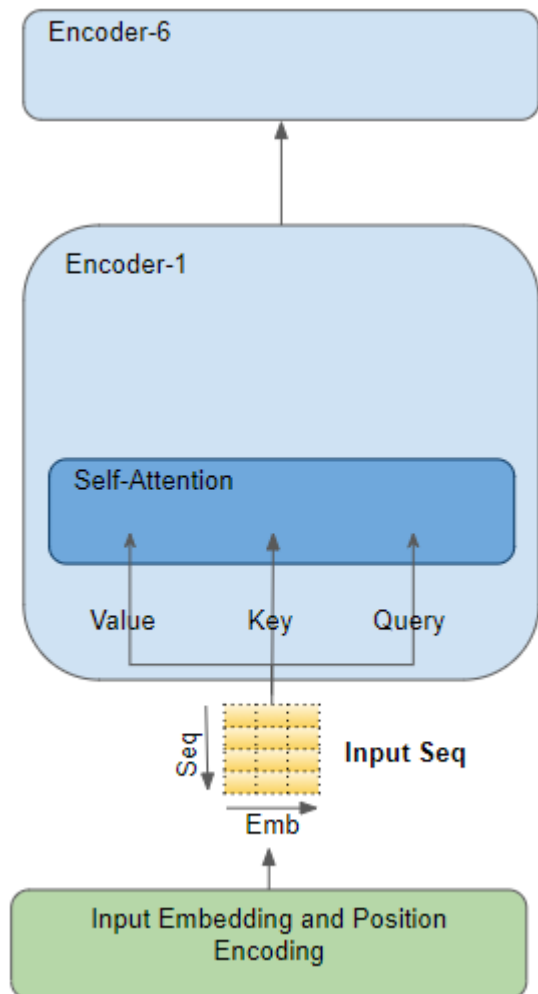
And if you're interested in NLP applications in general, I have some other articles you might like.

1. [Beam Search](#) (*Algorithm commonly used by Speech-to-Text and NLP applications to enhance predictions*)
2. [Bleu Score](#) (*Bleu Score and Word Error Rate are two essential metrics for NLP models*)

To understand what makes the Transformer tick, we must focus on Attention. Let's start with the input that goes into it, and then look at how it processes that input.

## **How does the input sequence reach the Attention module**

The Attention module is present in every Encoder in the Encoder stack, as well as every Decoder in the Decoder stack. We'll zoom in on the Encoder attention first.



Attention in the Encoder (Image by Author)

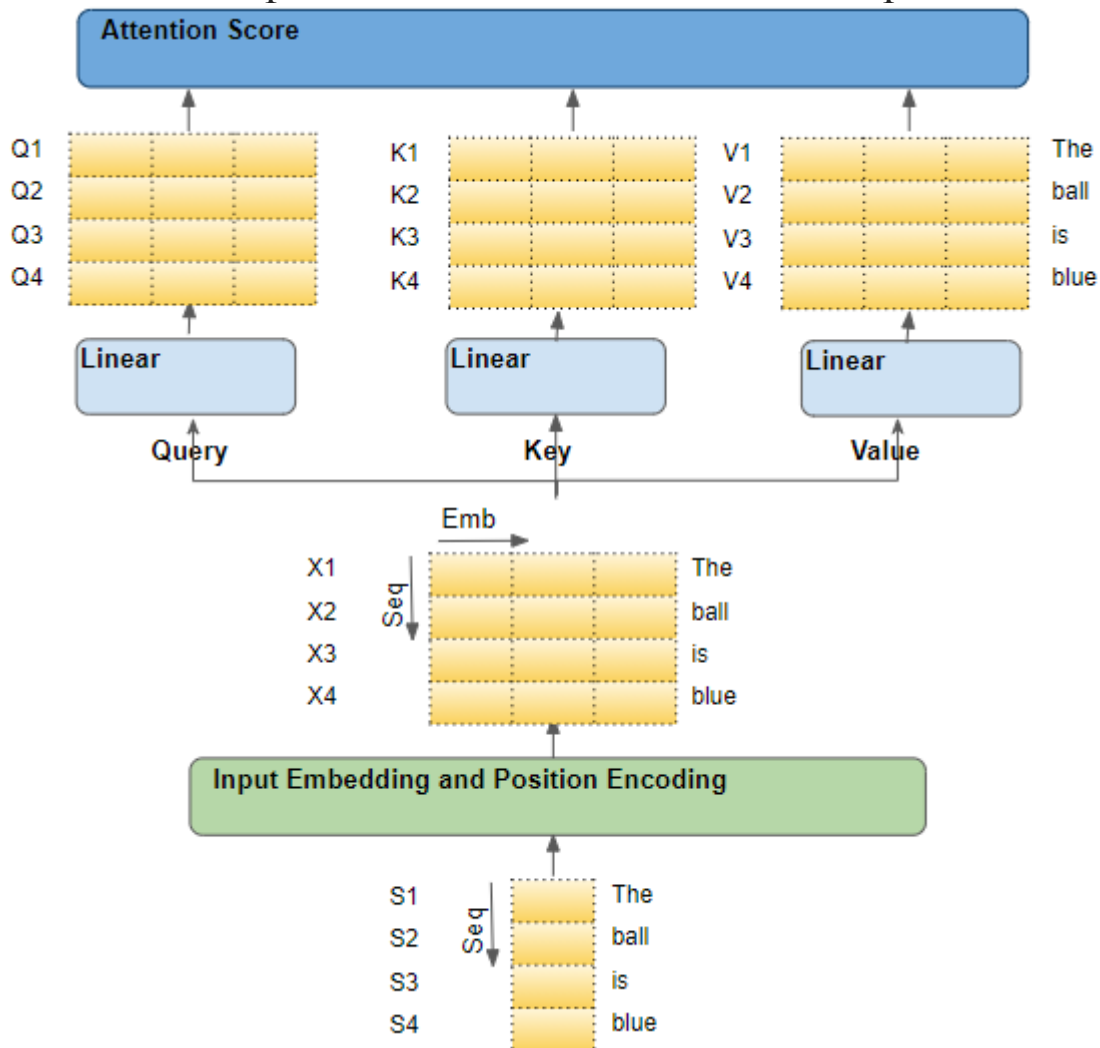
As an example, let's say that we're working on an English-to-Spanish translation problem, where one sample source sequence is "The ball is blue". The target sequence is "La bola es azul".

The source sequence is first passed through the Embedding and Position Encoding layer, which generates embedding vectors for each word in the sequence. The embedding is passed to the Encoder where it first reaches the Attention module.

Within Attention, the embedded sequence is passed through three Linear layers which produce three separate matrices — known as the

Query, Key, and Value. These are the three matrices that are used to compute the Attention Score.

The important thing to keep in mind is that each 'row' of these matrices corresponds to one word in the source sequence.



*The flow of the source sequence (Image by Author)*

## Each input row is a word from the sequence

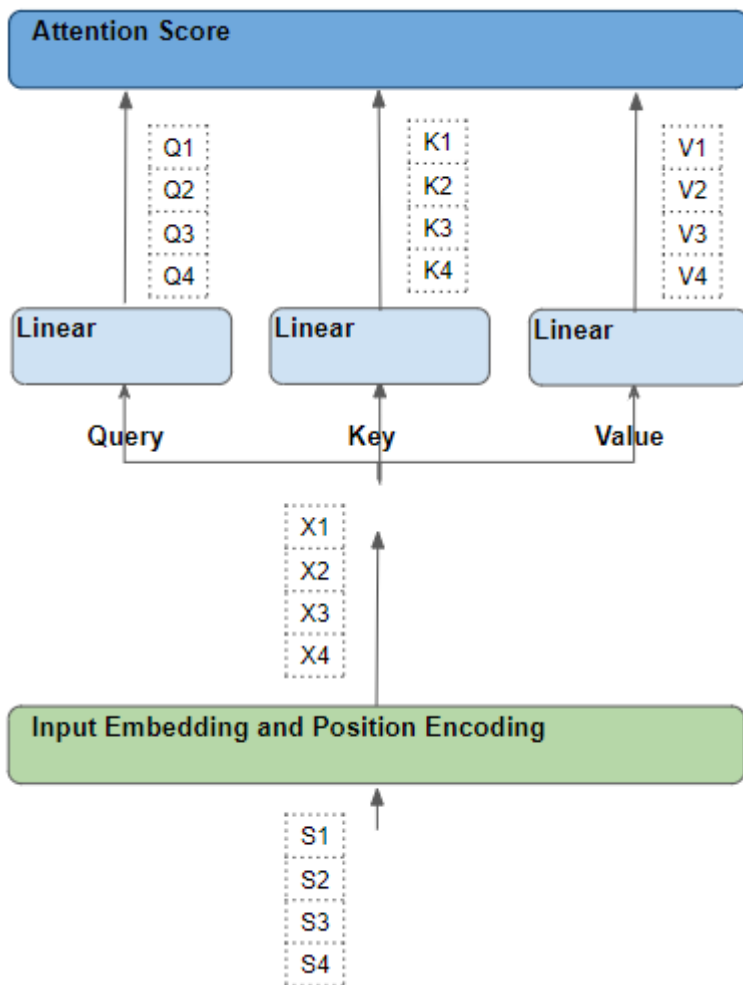
The way we will understand what is going on with Attention, is by starting with the individual words in the source sequence, and then following their path as they make their way through the

Transformer. In particular, we want to focus on what goes on inside the Attention Module.

That will help us clearly see how each word in the source and target sequences interacts with other words in the source and target sequences.

So as we go through this explanation, concentrate on what operations are being performed on each word, and how each vector maps to the original input word. We do not need to worry about many of the other details such as matrix shapes, specifics of the arithmetic calculations, multiple attention heads, and so on if they are not directly relevant to where each word is going.

So to simplify the explanation and the visualization, let's ignore the embedding dimension and track just the rows for each word.

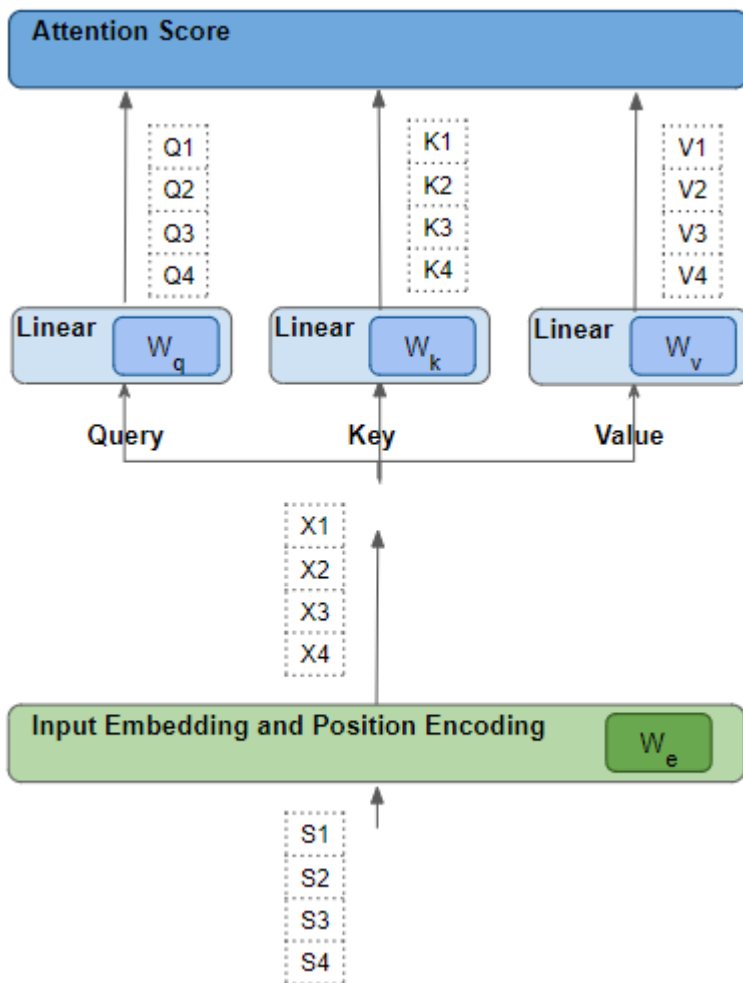


*The flow of each word in the source sequence (Image by Author)*

## Each word goes through a series of learnable transformations

Each such row has been generated from its corresponding source word by a series of transformations — embedding, position encoding, and linear layer.

All of those transformations are trainable operations. This means that the weights used in those operations are not pre-decided but are learned by the model in such a way that they produce the desired output predictions.



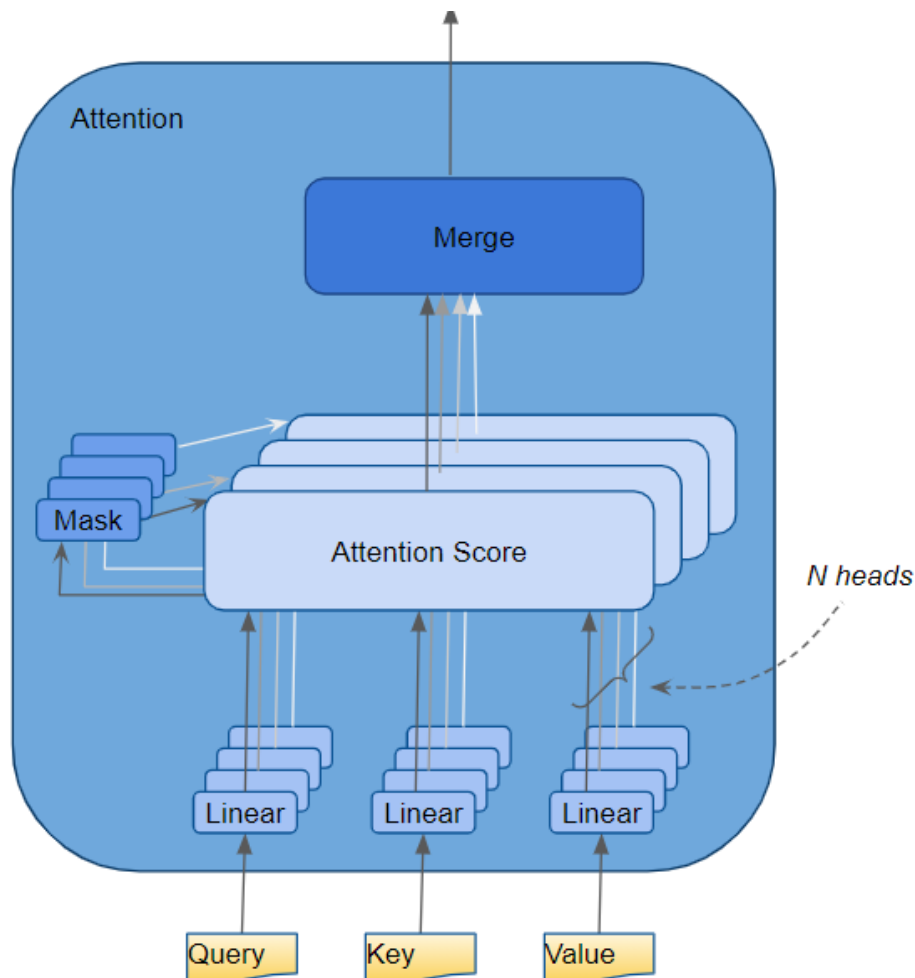
*Linear and Embedding weights are learned (Image by Author)*

The key question is, how does the Transformer figure out what set of weights will give it the best results? Keep this point in the back of your mind as we will come back to it a little later.

## Attention Score — Dot Product between Query and Key words

Attention performs several steps, but here, we will focus only on the Linear layer and the Attention Score.





*Multi-head attention (Image by Author)*

$$Z = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

*Attention Score calculation (Image by Author)*

As we can see from the formula, the first step within Attention is to do a matrix multiply (ie. dot product) between the Query (Q) matrix and a transpose of the Key (K) matrix. Watch what happens to each word.

We produce an intermediate matrix (let's call it a 'factor' matrix) where each cell is a matrix multiplication between two words.

Q1						Q1K1	Q1K2	Q1K3	Q1K4
Q2						Q2K1	Q2K2	Q2K3	Q2K4
Q3						Q3K1	Q3K2	Q3K3	Q3K4
Q4						Q4K1	Q4K2	Q4K3	Q4K4

*Dot Product between Query and Key matrices (Image by Author)*

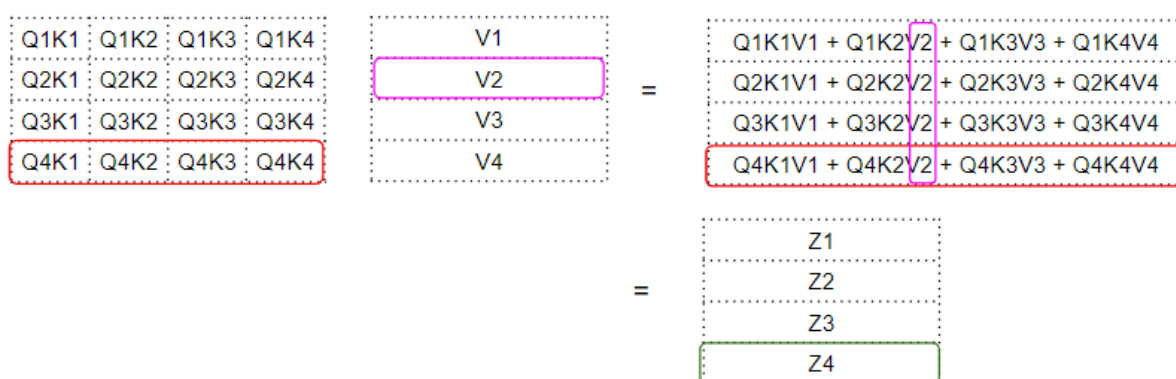
For instance, each column in the fourth row corresponds to a dot product between the fourth Query word with every Key word.

Q1						Q1K1	Q1K2	Q1K3	Q1K4
Q2						Q2K1	Q2K2	Q2K3	Q2K4
Q3						Q3K1	Q3K2	Q3K3	Q3K4
Q4						Q4K1	Q4K2	Q4K3	Q4K4

*Dot Product between Query and Key matrices (Image by Author)*

## Attention Score — Dot Product between Query-Key and Value words

The next step is a matrix multiply between this intermediate ‘factor’ matrix and the Value (V) matrix, to produce the attention score that is output by the attention module. Here we can see that the fourth row corresponds to the fourth Query word matrix multiplied with all other Key and Value words.



*Dot Product between Query-Key and Value matrices (Image by Author)*

This produces the Attention Score vector (Z) that is output by the Attention Module.

The way to think about the output score is that, for each word, it is the encoded value of every word from the “Value” matrix, weighted by the “factor” matrix. The factor matrix is the dot product of the Query value for that specific word with the Key value of all words.

Fourth word Score      Fourth Query word \* first Key word

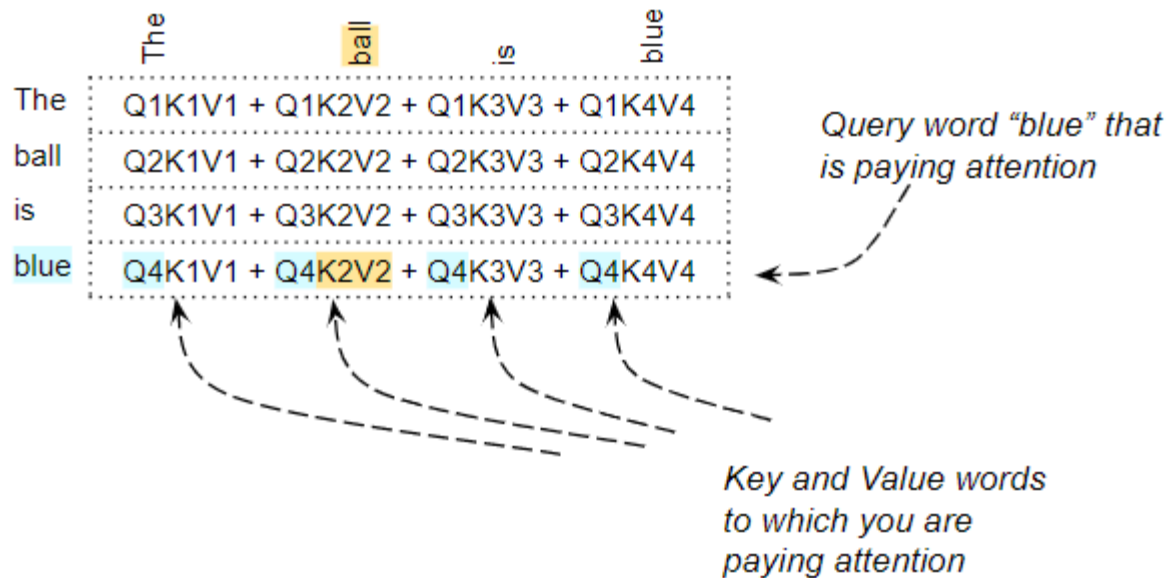
$$Z_4 = (Q_4K_1) V_1 + (Q_4K_2) V_2 + (Q_4K_3) V_3 + (Q_4K_4) V_4$$

Fourth Query word \* second Key word

*Attention Score is a weighted sum of the Value words (Image by Author)*

## What is the role of the Query, Key, and Value words?

The Query word can be interpreted as the word *for which* we are calculating Attention. The Key and Value word is the word *to which* we are paying attention ie. how relevant is that word to the Query word.



*Attention Score for the word "blue" pays attention to every other word (Image by Author)*

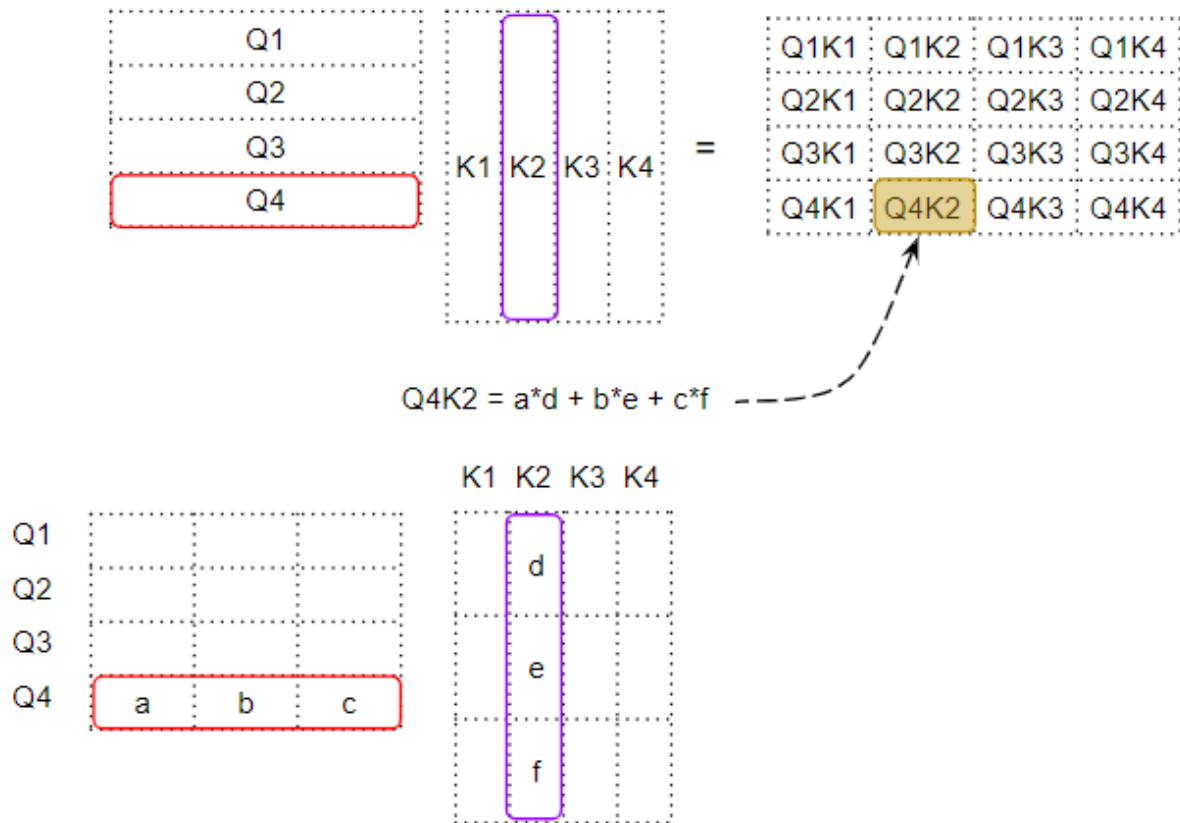
For example, for the sentence, "The ball is blue", the row for the word "blue" will contain the attention scores for "blue" with every other word. Here, "blue" is the Query word, and the other words are the "Key/Value".

There are other operations being performed such as a division and a softmax, but we can ignore them in this article. They just change the numeric values in the matrices but don't affect the position of each word row in the matrix. Nor do they involve any inter-word interactions.

## Dot Product tells us the similarity between words

So we have seen that the Attention Score is capturing some interaction between a particular word, and every other word in the sentence, by doing a dot product, and then adding them up. But how does the matrix multiply help the Transformer determine the relevance between two words?

To understand this, remember that the Query, Key, and Value rows are actually vectors with an Embedding dimension. Let's zoom in on how the matrix multiplication between those vectors is calculated.



*Each cell is a dot product between two word vectors (Image by Author)*

When we do a dot product between two vectors, we multiply pairs of numbers and then sum them up.

- If the two paired numbers (eg. 'a' and 'd' above) are both positive or both negative, then the product will be positive. The product will increase the final summation.
- If one number is positive and the other negative, then the product will be negative. The product will reduce the final summation.

- If the product is positive, the larger the two numbers, the more they contribute to the final summation.

This means that if the signs of the corresponding numbers in the two vectors are aligned, the final sum will be larger.

## **How does the Transformer learn the relevance between words?**

This notion of the Dot Product applies to the attention score as well. If the vectors for two words are more aligned, the attention score will be higher.

So what is the behavior we want for the Transformer?

We want the attention score to be high for two words that are relevant to each other in the sentence. And we want the score to be low for two words that are unrelated to one another.

For example, for the sentence, “The black cat drank the milk”, the word “milk” is very relevant to “drank”, perhaps slightly less relevant to “cat”, and irrelevant to “black”. We want “milk” and “drank” to produce a high attention score, for “milk” and “cat” to produce a slightly lower score, and for “milk” and “black”, to produce a negligible score.

This is the output we want the model to learn to produce.

For this to happen, the word vectors for “milk” and “drank” must be aligned. The vectors for “milk” and “cat” will diverge somewhat. And they will be quite different for “milk” and “black”.

Let’s go back to the point we had kept at the back of our minds — how does the Transformer figure out what set of weights will give it the best results?

The word vectors are generated based on the word embeddings and the weights of the Linear layers. Therefore the Transformer can learn those embeddings, Linear weights, and so on to produce the word vectors as required above.

In other words, it will learn those embeddings and weights in such a way that if two words in a sentence are relevant to each other, then their word vectors will be aligned. And hence produce a higher attention score. For words that are not relevant to each other, the word vectors will not be aligned and will produce a lower attention score.

Therefore the embeddings for “milk” and “drank” will be very aligned and produce a high attention score. They will diverge somewhat for “milk” and “cat” to produce a slightly lower score and will be quite different for “milk” and “black”, to produce a low score.

This then is the principle behind the Attention module.

## **Summarizing — What makes the Transformer tick?**

The dot product between the Query and Key computes the relevance between each pair of words. This relevance is then used as a “factor” to compute a weighted sum of all the Value words. That weighted sum is output as the Attention Score.

The Transformer learns embeddings etc, in such a way that words that are relevant to one another are more aligned.

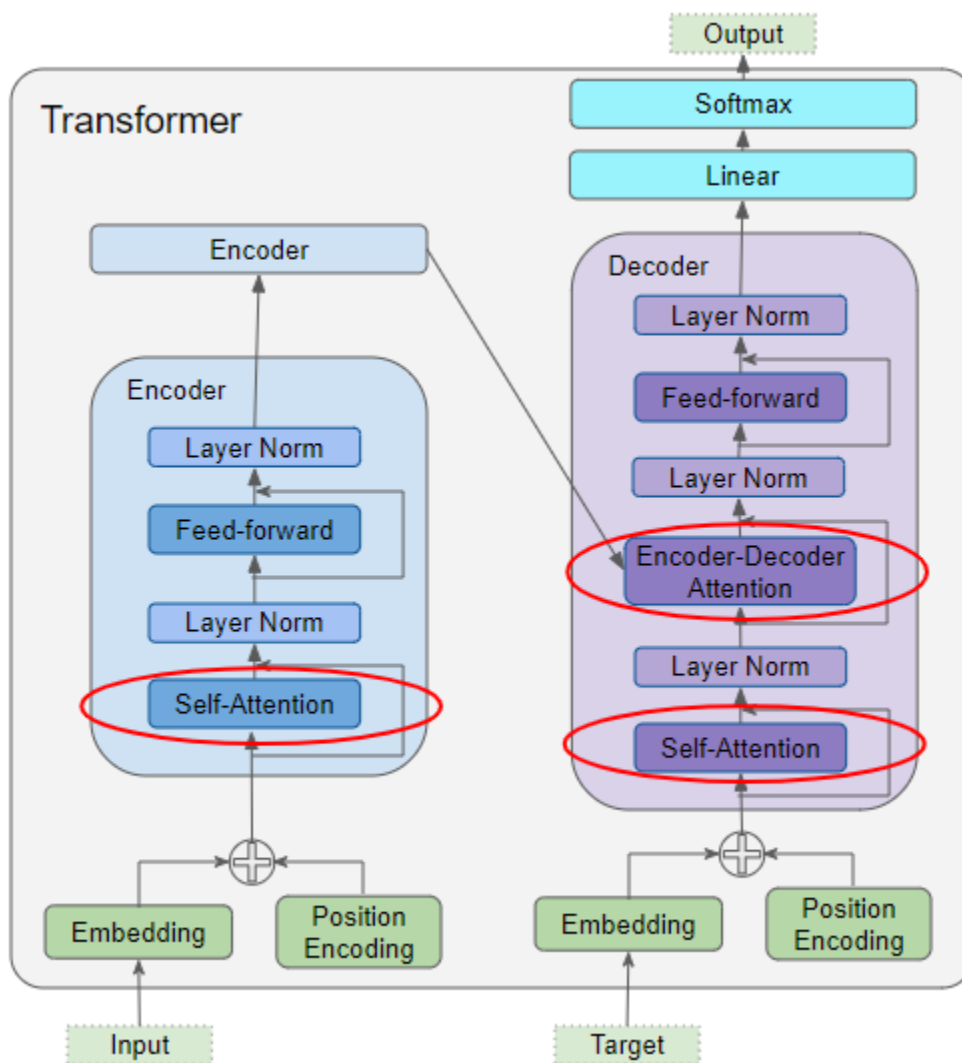
This is one reason for introducing the three Linear layers and making three versions of the input sequence, for the Query, Key, and Value. That gives the Attention module some more parameters that it is able to learn to tune the creation of the word vectors.

## **Encoder Self-Attention in the Transformer**

Attention is used in the Transformer in three places:

- Self-attention in the Encoder — the source sequence pays attention to itself
- Self-attention in the Decoder — the target sequence pays attention to itself
- Encoder-Decoder-attention in the Decoder — the target sequence pays attention to the source sequence



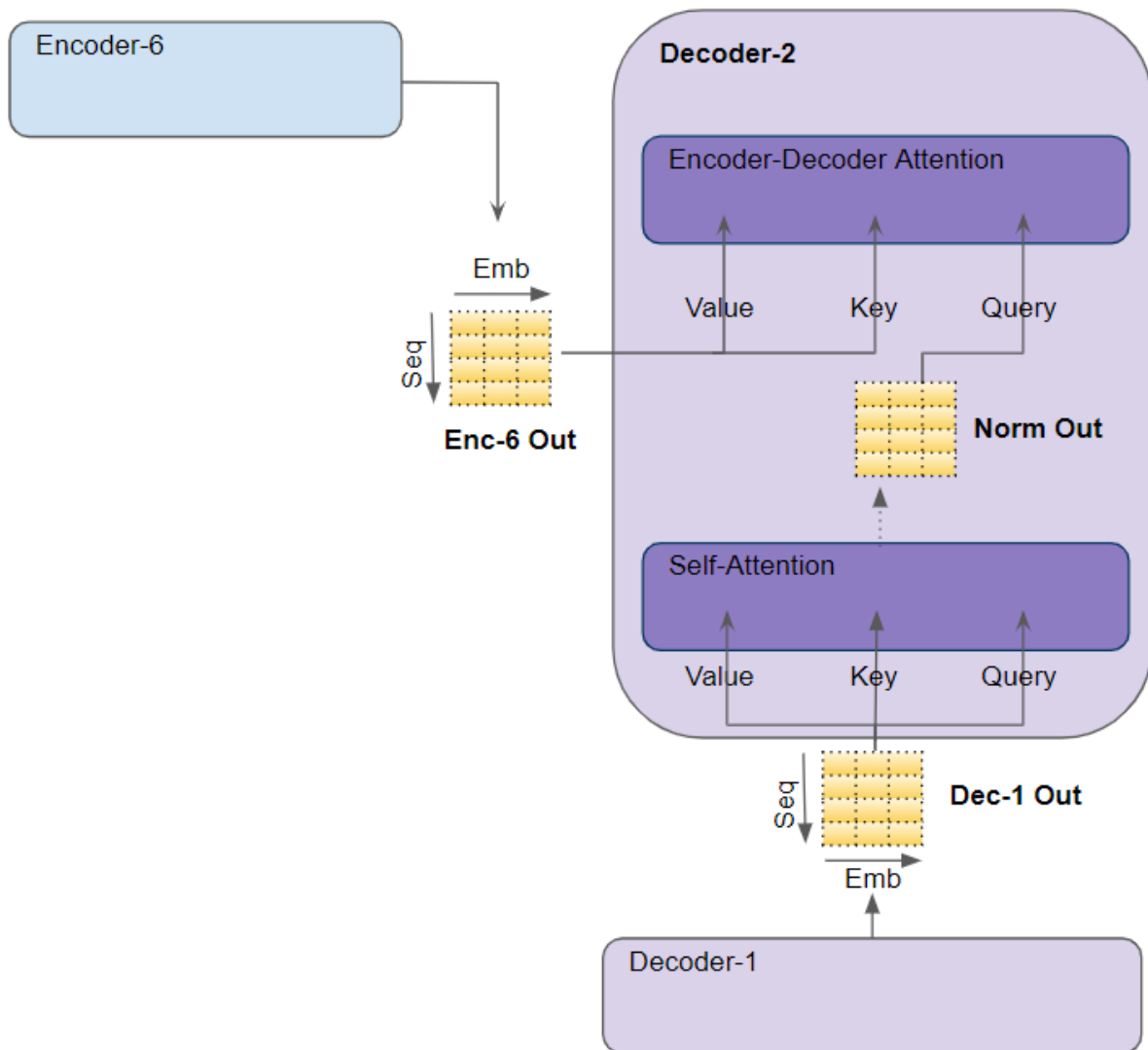


*Attention in the Transformer (Image by Author)*

In the Encoder Self Attention, we compute the relevance of each word in the source sentence to each other word in the source sentence. This happens in all the Encoders in the stack.

## Decoder Self-Attention in the Transformer

Most of what we've just seen in the Encoder Self Attention applies to Attention in the Decoder as well, with a few small but significant differences.



Attention in the Decoder (Image by Author)

In the Decoder Self Attention, we compute the relevance of each word in the target sentence to each other word in the target sentence.

	La	bola	es	azul
La	$Q1K1V1 + Q1K2V2 + Q1K3V3 + Q1K4V4$			
bola	$Q2K1V1 + Q2K2V2 + Q2K3V3 + Q2K4V4$			
es	$Q3K1V1 + Q3K2V2 + Q3K3V3 + Q3K4V4$			
azul	$Q4K1V1 + Q4K2V2 + Q4K3V3 + Q4K4V4$			

### Decoder Self Attention

Target sentence paying attention to itself

Decoder Self Attention (Image by Author)

## Encoder-Decoder Attention in the Transformer

In the Encoder-Decoder Attention, the Query is obtained from the target sentence and the Key/Value from the source sentence. Thus it computes the relevance of each word in the target sentence to each word in the source sentence.

	The	ball	is	blue
La	$Q1K1V1 + Q1K2V2 + Q1K3V3 + Q1K4V4$			
bola	$Q2K1V1 + Q2K2V2 + Q2K3V3 + Q2K4V4$			
es	$Q3K1V1 + Q3K2V2 + Q3K3V3 + Q3K4V4$			
azul	$Q4K1V1 + Q4K2V2 + Q4K3V3 + Q4K4V4$			

Query word "azul" that is paying attention

### Encoder-Decoder Attention

Target sentence paying attention to source sentence

Encoder-Decoder Attention (Image by Author)

# Transformers Explained Visually (Part 3): Multi-head Attention, deep dive

A Gentle Guide to the inner workings of Self-Attention, Encoder-Decoder Attention, Attention Score and Masking, in Plain English.



Photo by [Scott Tobin](#) on [Unsplash](#)

This is the third article in my series on Transformers. We are covering its functionality in a top-down manner. In the previous

articles, we learned what a Transformer is, its architecture, and how it works.

In this article, we will go a step further and dive deeper into Multi-head Attention, which is the brains of the Transformer.

Here's a quick summary of the previous and following articles in the series. My goal throughout will be to understand not just how something works but why it works that way.

1. [Overview of functionality](#) (*How Transformers are used, and why they are better than RNNs. Components of the architecture, and behavior during Training and Inference*)
2. [How it works](#) (*Internal operation end-to-end. How data flows and what computations are performed, including matrix representations*)
3. **Multi-head Attention — this article** (*Inner workings of the Attention module throughout the Transformer*)
4. [Why Attention Boosts Performance](#) (*Not just what Attention does but why it works so well. How does Attention capture the relationships between words in a sentence*)

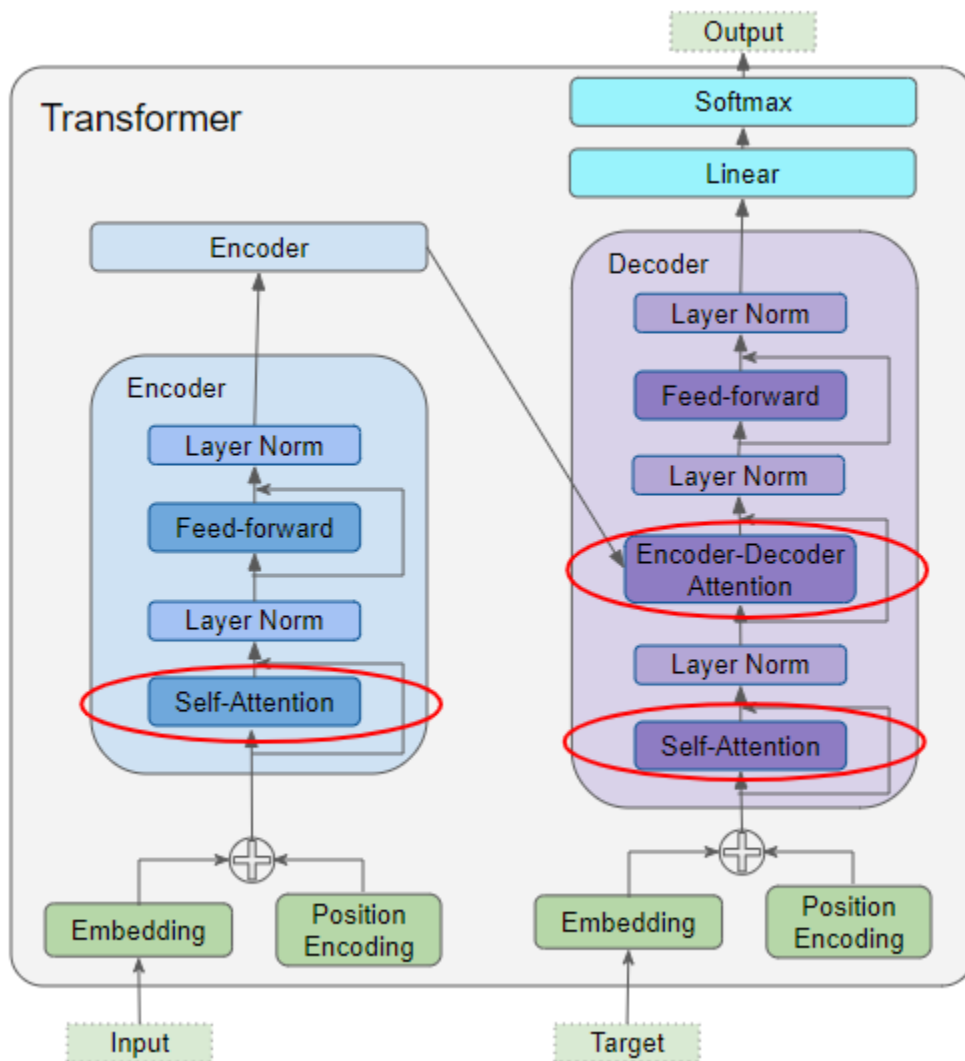
And if you're interested in NLP applications in general, I have some other articles you might like.

1. [Beam Search](#) (*Algorithm commonly used by Speech-to-Text and NLP applications to enhance predictions*)
2. [Bleu Score](#) (*Bleu Score and Word Error Rate are two essential metrics for NLP models*)

## **How Attention is used in the Transformer**

As we discussed in [Part 2](#), Attention is used in the Transformer in three places:

- Self-attention in the Encoder — the input sequence pays attention to itself
- Self-attention in the Decoder — the target sequence pays attention to itself
- Encoder-Decoder-attention in the Decoder — the target sequence pays attention to the input sequence



(Image by Author)

## Attention Input Parameters — Query, Key, and Value

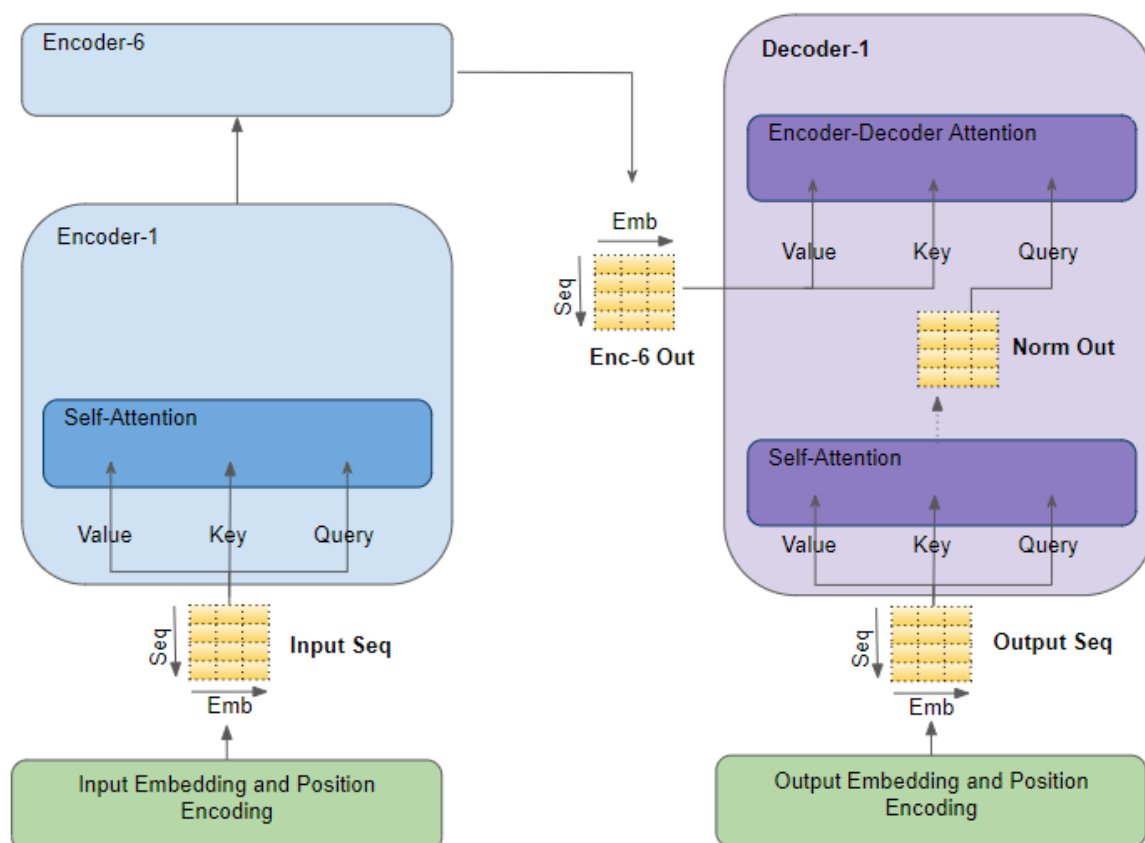
The Attention layer takes its input in the form of three parameters, known as the Query, Key, and Value.

All three parameters are similar in structure, with each word in the sequence represented by a vector.



## Encoder Self-Attention

The input sequence is fed to the Input Embedding and Position Encoding, which produces an encoded representation for each word in the input sequence that captures the meaning and position of each word. This is fed to all three parameters, Query, Key, and Value in the Self-Attention in the first Encoder which then also produces an encoded representation for each word in the input sequence, that now incorporates the attention scores for each word as well. As this passes through all the Encoders in the stack, each Self-Attention module also adds its own attention scores into each word's representation.



(Image by Author)



## **Decoder Self-Attention**

Coming to the Decoder stack, the target sequence is fed to the Output Embedding and Position Encoding, which produces an encoded representation for each word in the target sequence that captures the meaning and position of each word. This is fed to all three parameters, Query, Key, and Value in the Self-Attention in the first Decoder which then also produces an encoded representation for each word in the target sequence, which now incorporates the attention scores for each word as well.

After passing through the Layer Norm, this is fed to the Query parameter in the Encoder-Decoder Attention in the first Decoder

## **Encoder-Decoder Attention**

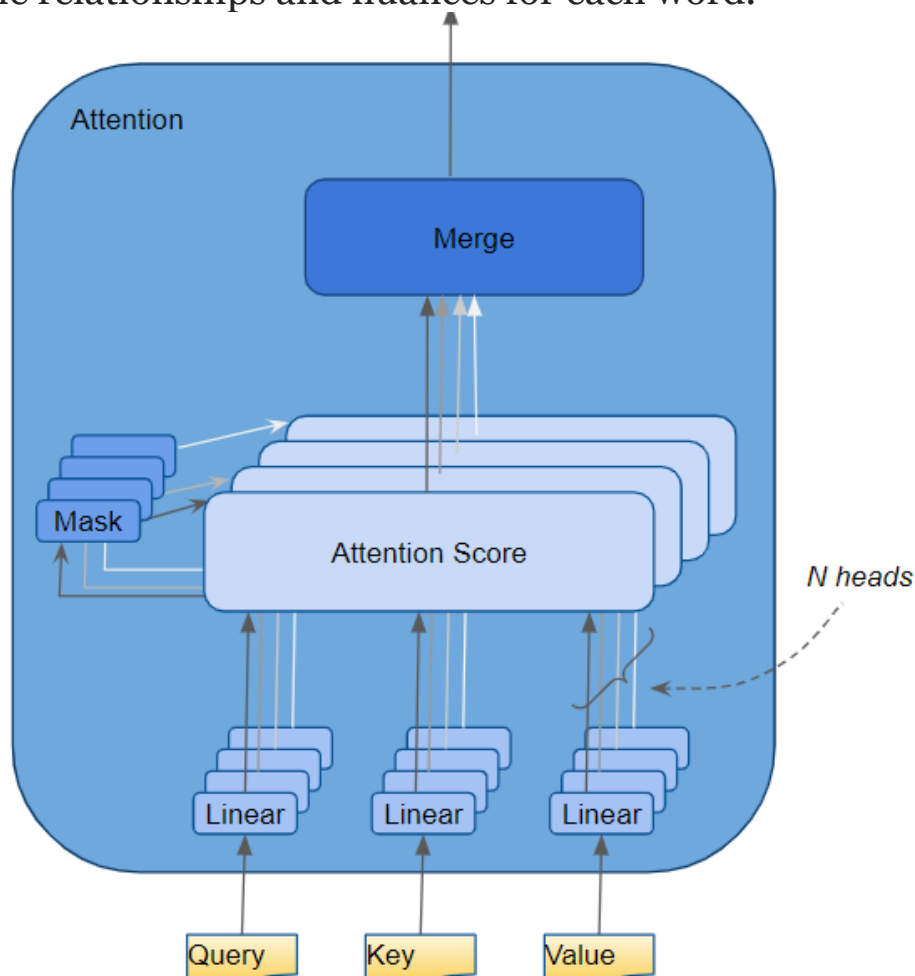
Along with that, the output of the final Encoder in the stack is passed to the Value and Key parameters in the Encoder-Decoder Attention.

The Encoder-Decoder Attention is therefore getting a representation of both the target sequence (from the Decoder Self-Attention) and a representation of the input sequence (from the Encoder stack). It, therefore, produces a representation with the attention scores for each target sequence word that captures the influence of the attention scores from the input sequence as well.

As this passes through all the Decoders in the stack, each Self-Attention and each Encoder-Decoder Attention also add their own attention scores into each word's representation.

## Multiple Attention Heads

In the Transformer, the Attention module repeats its computations multiple times in parallel. Each of these is called an Attention Head. The Attention module splits its Query, Key, and Value parameters N-ways and passes each split independently through a separate Head. All of these similar Attention calculations are then combined together to produce a final Attention score. This is called Multi-head attention and gives the Transformer greater power to encode multiple relationships and nuances for each word.



(Image by Author)

To understand exactly how the data is processed internally, let's walk through the working of the Attention module while we are training the Transformer to solve a translation problem. We'll use one sample of our training data which consists of an input sequence ('You are welcome' in English) and a target sequence ('De nada' in Spanish).

## **Attention Hyperparameters**

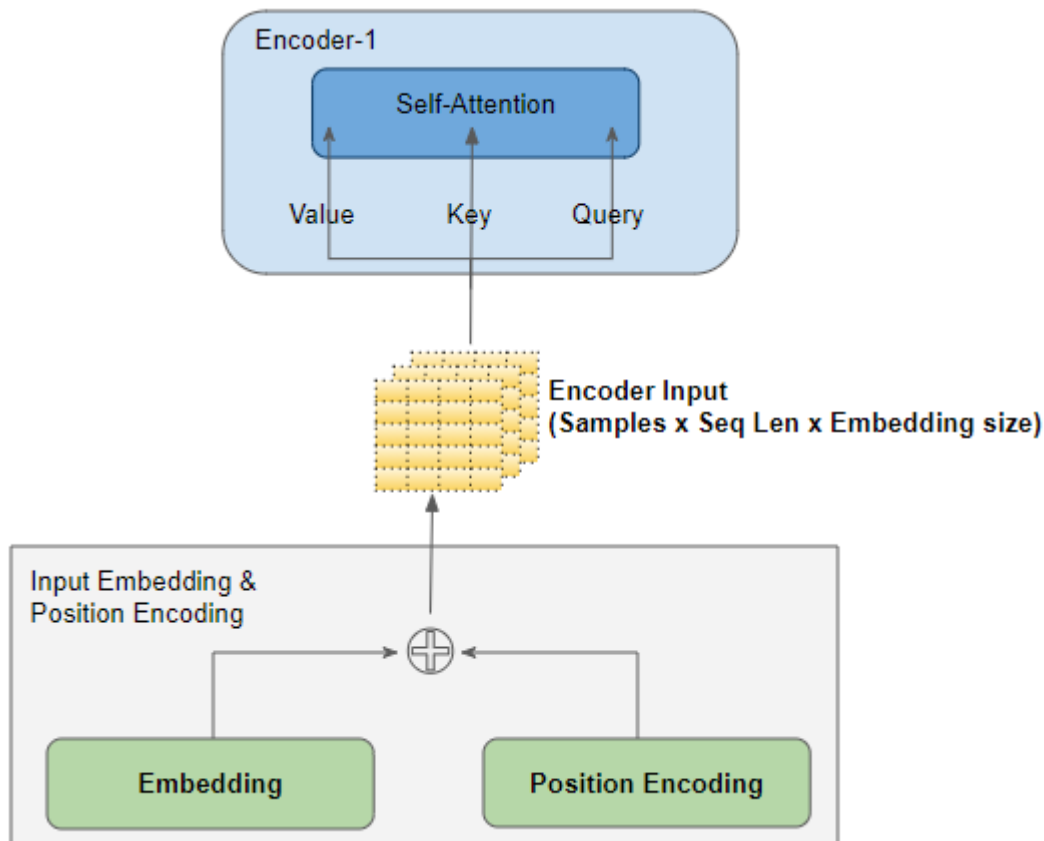
There are three hyperparameters that determine the data dimensions:

- Embedding Size — width of the embedding vector (we use a width of 6 in our example). This dimension is carried forward throughout the Transformer model and hence is sometimes referred to by other names like 'model size' etc.
- Query Size (equal to Key and Value size)— the size of the weights used by three Linear layers to produce the Query, Key, and Value matrices respectively (we use a Query size of 3 in our example)
- Number of Attention heads (we use 2 heads in our example)

In addition, we also have the Batch size, giving us one dimension for the number of samples.

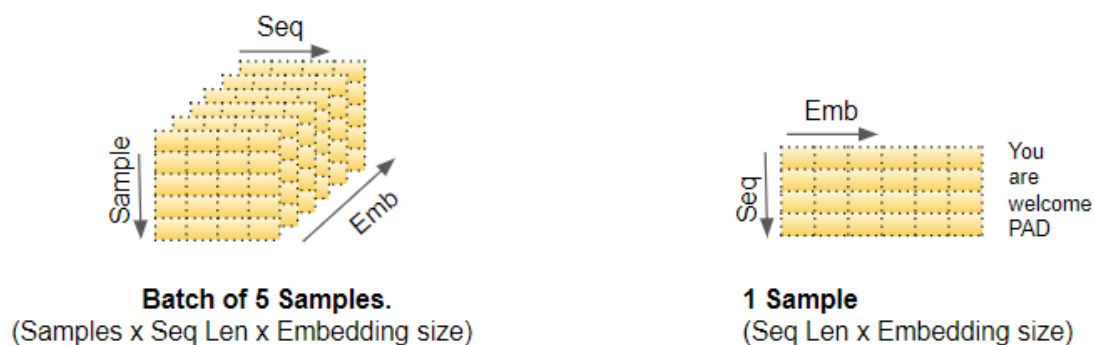
## Input Layers

The Input Embedding and Position Encoding layers produce a matrix of shape (Number of Samples, Sequence Length, Embedding Size) which is fed to the Query, Key, and Value of the first Encoder in the stack.



(Image by Author)

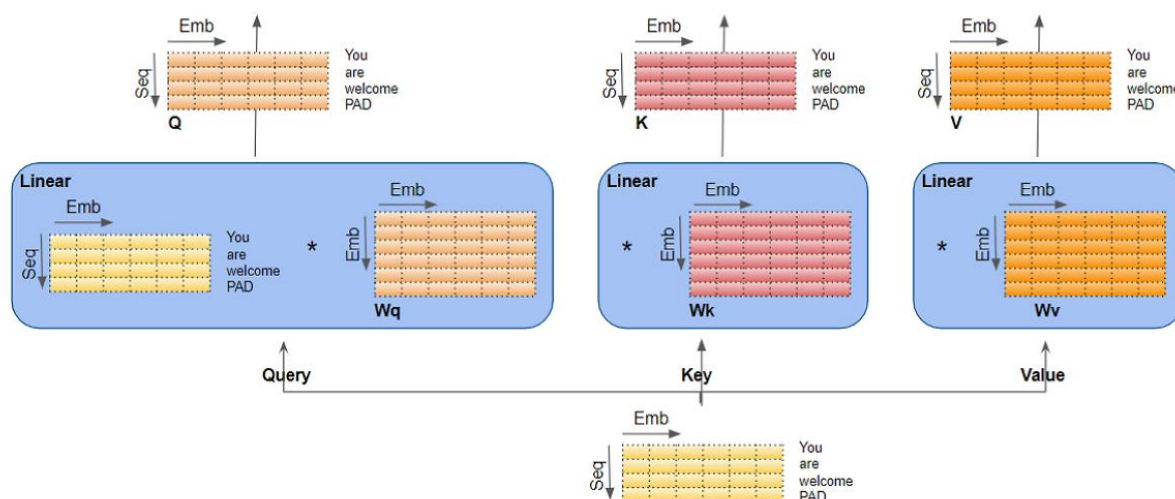
To make it simple to visualize, we will drop the Batch dimension in our pictures and focus on the remaining dimensions.



(Image by Author)

## Linear Layers

There are three separate Linear layers for the Query, Key, and Value. Each Linear layer has its own weights. The input is passed through these Linear layers to produce the Q, K, and V matrices.



(Image by Author)

## Splitting data across Attention heads

Now the data gets split across the multiple Attention heads so that each can process it independently.

However, the important thing to understand is that this is a logical split only. The Query, Key, and Value are not physically split into separate matrices, one for each Attention head. A single data matrix is used for the Query, Key, and Value, respectively, with logically separate sections of the matrix for each Attention head. Similarly, there are not separate Linear layers, one for each Attention head. All the Attention heads share the same Linear layer but simply operate on their 'own' logical section of the data matrix.

## Linear layer weights are logically partitioned per head

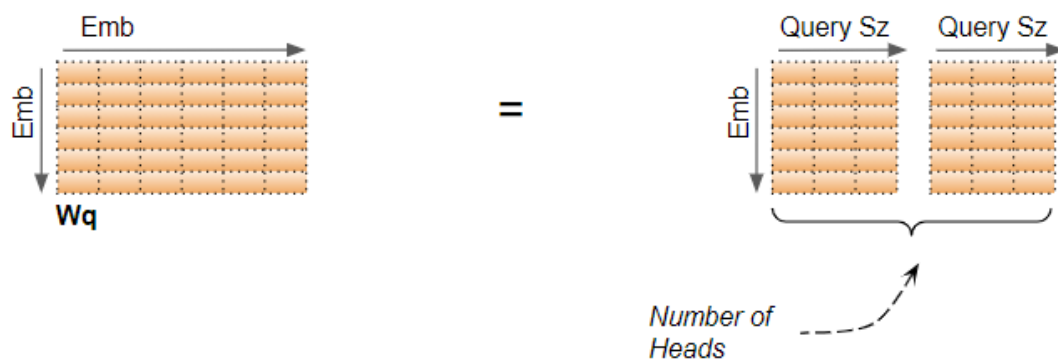
This logical split is done by partitioning the input data as well as the Linear layer weights uniformly across the Attention heads. We can achieve this by choosing the Query Size as below:

$$\text{Query Size} = \text{Embedding Size} / \text{Number of heads}$$

A diagram showing a large rectangular matrix labeled  $W_q$  at the bottom. A vertical arrow on the left is labeled "Emb Sz" and a horizontal arrow at the top is labeled "Emb Sz". Inside the matrix, the equation  $= \frac{\text{Emb Sz}}{\text{Heads}} * \text{Heads}$  is written. Above the matrix, the equation  $= \text{Query Sz} * \text{Heads}$  is written with an arrow pointing to the right.

(Image by Author)

In our example, that is why the Query Size =  $6/2 = 3$ . Even though the layer weight (and input data) is a single matrix we can think of it as 'stacking together' the separate layer weights for each head.



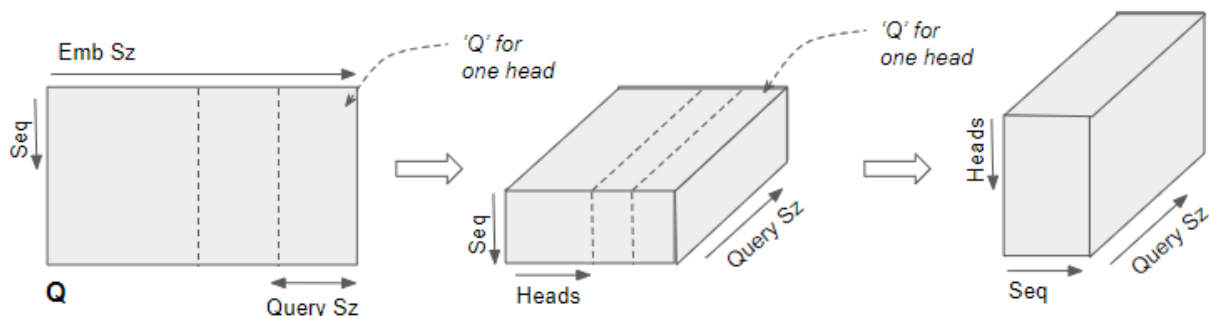
(Image by Author)

The computations for all Heads can be therefore be achieved via a single matrix operation rather than requiring N separate operations. This makes the computations more efficient and keeps the model simple because fewer Linear layers are required, while still achieving the power of the independent Attention heads.

## Reshaping the Q, K, and V matrices

The Q, K, and V matrices output by the Linear layers are reshaped to include an explicit Head dimension. Now each 'slice' corresponds to a matrix per head.

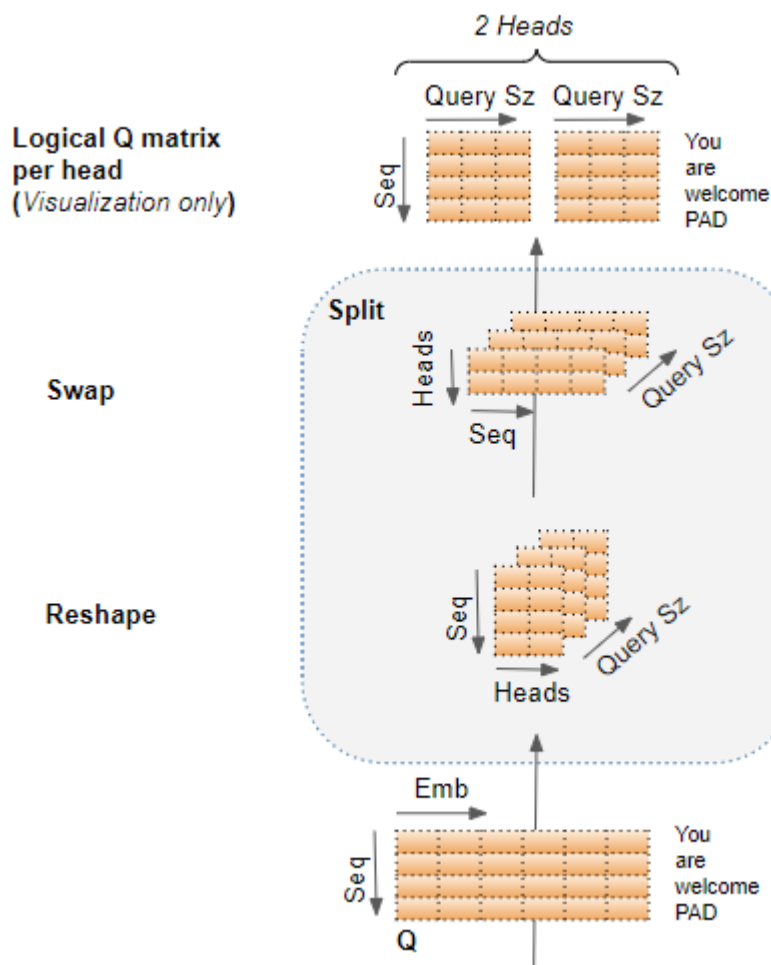
This matrix is reshaped again by swapping the Head and Sequence dimensions. Although the Batch dimension is not drawn, the dimensions of Q are now (Batch, Head, Sequence, Query size).



The Q matrix is reshaped to include a Head dimension and then reshaped again by swapping the Head and Sequence dimensions. (Image by Author)

In the picture below, we can see the complete process of splitting our example Q matrix, after coming out of the Linear layer.

The final stage is for visualization only — although the Q matrix is a single matrix, we can think of it as a logically separate Q matrix per head.



Q matrix split across the Attention Heads (Image by Author)

We are ready to compute the Attention Score.

## Compute the Attention Score for each head

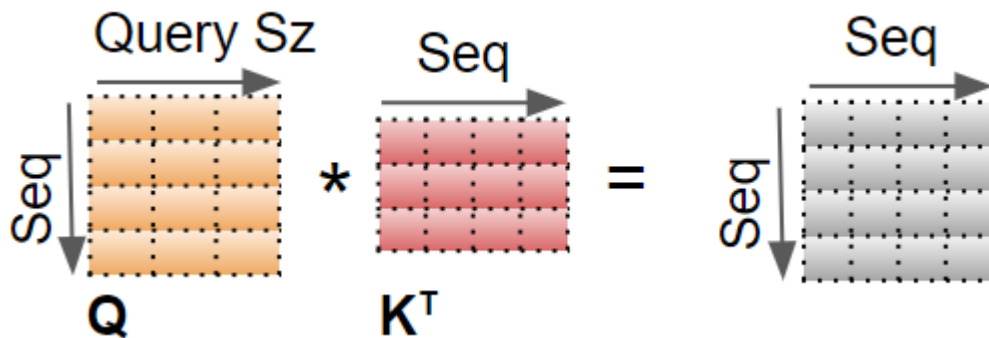
We now have the 3 matrices, Q, K, and V, split across the heads. These are used to compute the Attention Score.

We will show the computations for a single head using just the last two dimensions (Sequence and Query size) and skip the first two dimensions (Batch and Head). Essentially, we can imagine that the computations we're looking at are getting 'repeated' for each head



and for each sample in the batch (although, obviously, they are happening as a single matrix operation, and not as a loop).

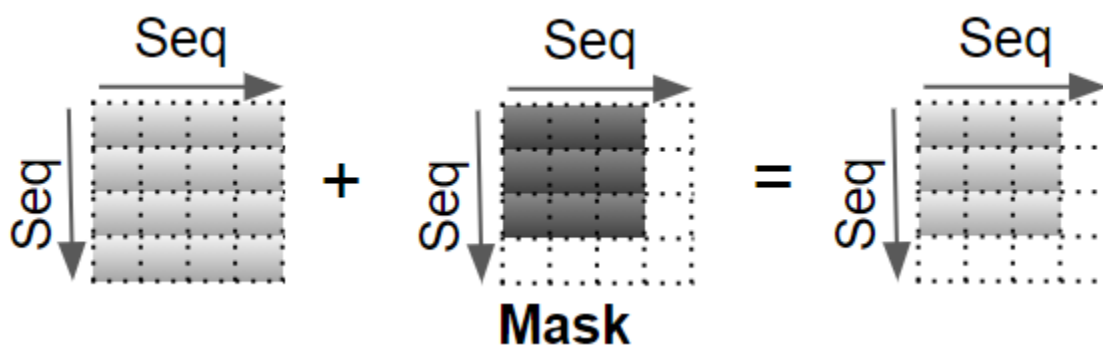
The first step is to do a matrix multiplication between Q and K.



(Image by Author)

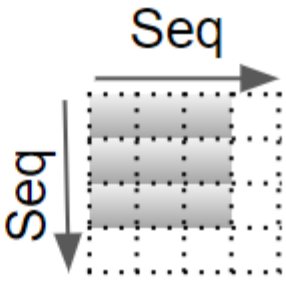
A Mask value is now added to the result. In the Encoder Self-attention, the mask is used to mask out the Padding values so that they don't participate in the Attention Score.

Different masks are applied in the Decoder Self-attention and in the Decoder Encoder-Attention which we'll come to a little later in the flow.



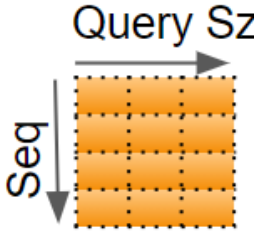
(Image by Author)

The result is now scaled by dividing by the square root of the Query size, and then a Softmax is applied to it.

$$\text{Softmax} \left( \frac{\text{Seq} \times \text{Seq}}{\sqrt{\text{Query size}}} \right)$$


(Image by Author)

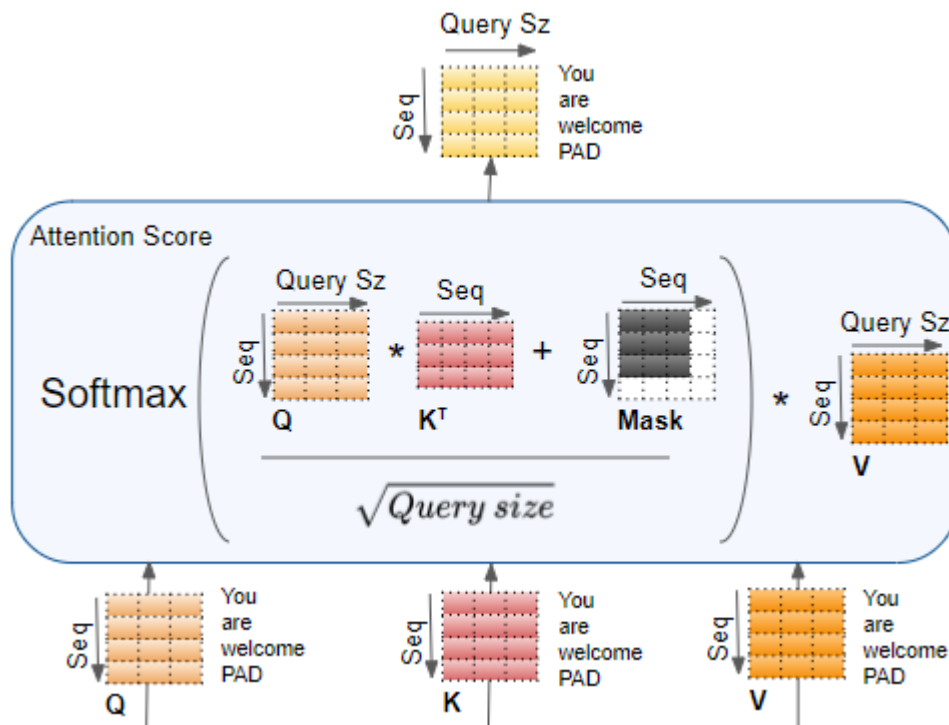
Another matrix multiplication is performed between the output of the Softmax and the V matrix.

$$\text{Softmax} \left( \frac{\text{Seq} \times \text{Seq}}{\sqrt{\text{Query size}}} \right) * \text{Seq} \times \text{Query Sz}$$


**V**

(Image by Author)

The complete Attention Score calculation in the Encoder Self-attention is as below:



(Image by Author)

## Merge each Head's Attention Scores together

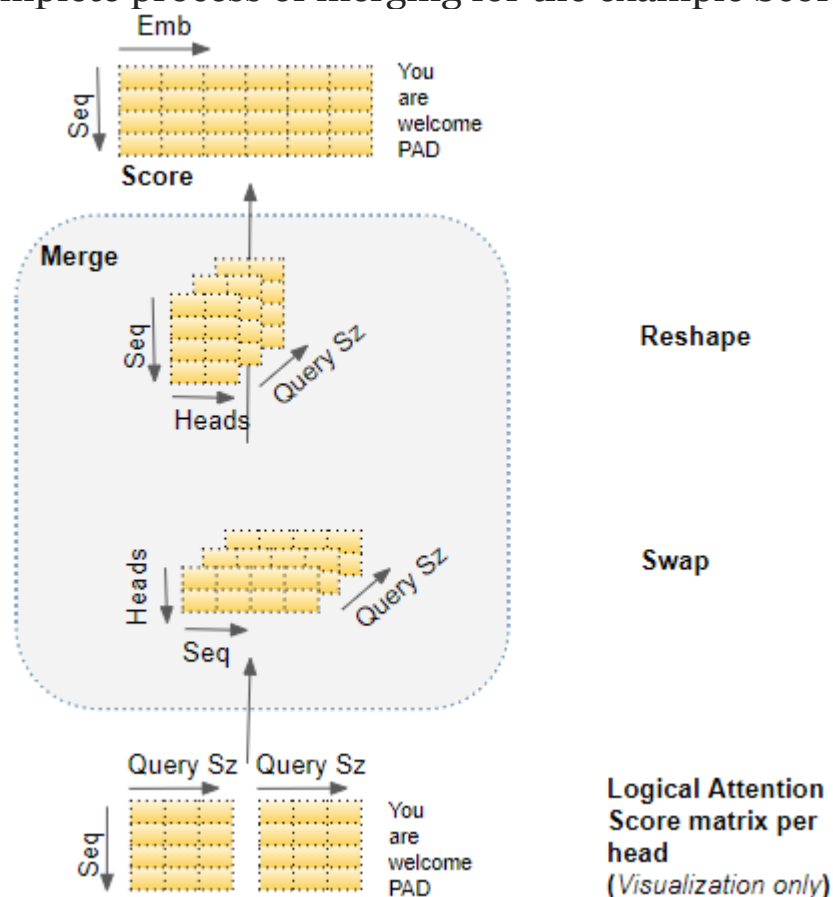
We now have separate Attention Scores for each head, which need to be combined together into a single score. This Merge operation is essentially the reverse of the Split operation.

It is done by simply reshaping the result matrix to eliminate the Head dimension. The steps are:

- Reshape the Attention Score matrix by swapping the Head and Sequence dimensions. In other words, the matrix shape goes from (Batch, Head, Sequence, Query size) to (Batch, Sequence, Head, Query size).
- Collapse the Head dimension by reshaping to (Batch, Sequence, Head \* Query size). This effectively

concatenates the Attention Score vectors for each head into a single merged Attention Score.

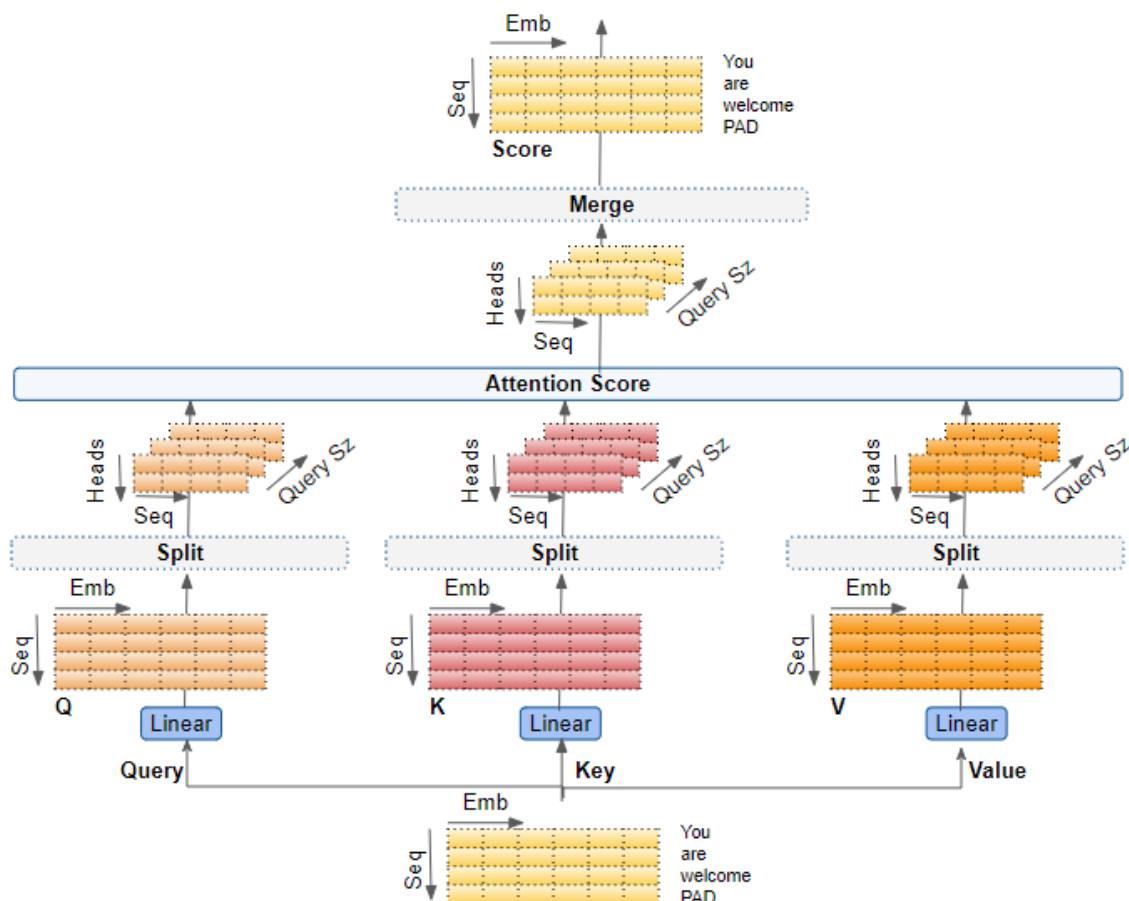
Since Embedding size = Head \* Query size, the merged Score is (Batch, Sequence, Embedding size). In the picture below, we can see the complete process of merging for the example Score matrix.



(Image by Author)

## End-to-end Multi-head Attention

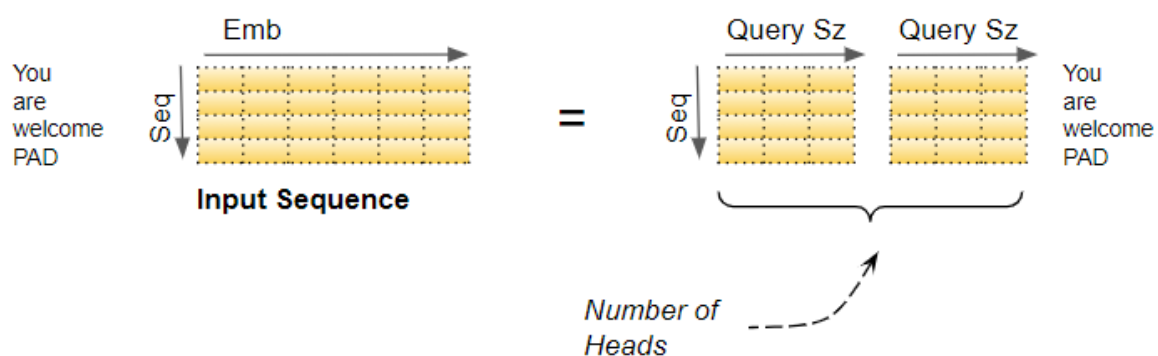
Putting it all together, this is the end-to-end flow of the Multi-head Attention.



(Image by Author)

## Multi-head split captures richer interpretations

An Embedding vector captures the meaning of a word. In the case of Multi-head Attention, as we have seen, the Embedding vectors for the input (and target) sequence gets logically split across multiple heads. What is the significance of this?



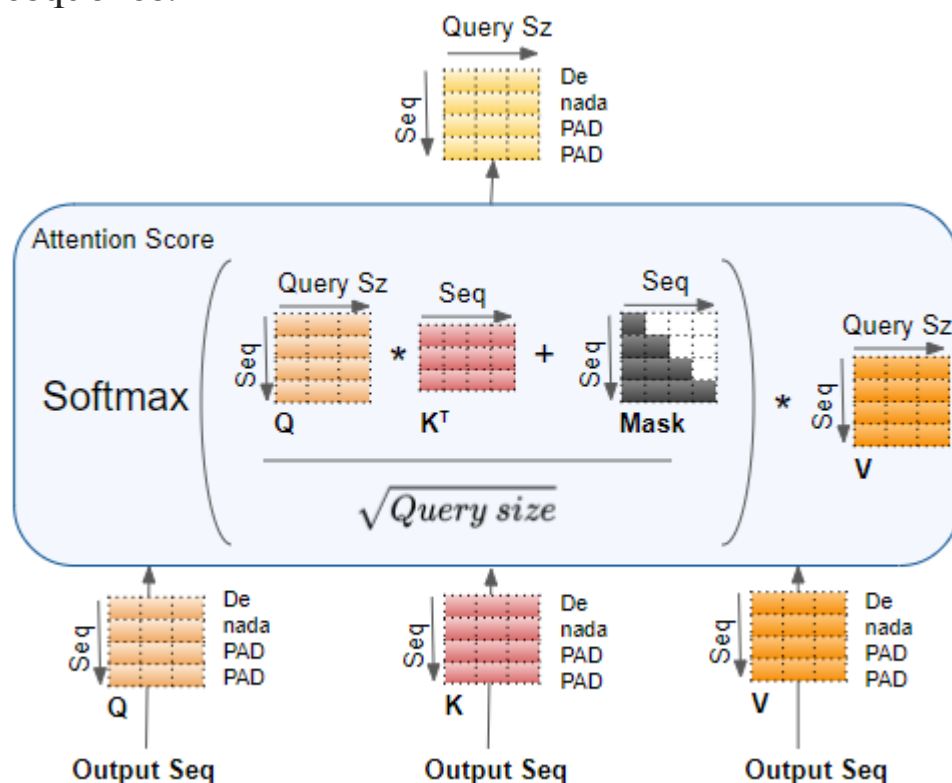
(Image by Author)

This means that separate sections of the Embedding can learn different aspects of the meanings of each word, as it relates to other words in the sequence. This allows the Transformer to capture richer interpretations of the sequence.

This may not be a realistic example, but it might help to build intuition. For instance, one section might capture the ‘gender-ness’ (male, female, neuter) of a noun while another might capture the ‘cardinality’ (singular vs plural) of a noun. This might be important during translation because, in many languages, the verb that needs to be used depends on these factors.

## Decoder Self-Attention and Masking

The Decoder Self-Attention works just like the Encoder Self-Attention, except that it operates on each word of the target sequence.

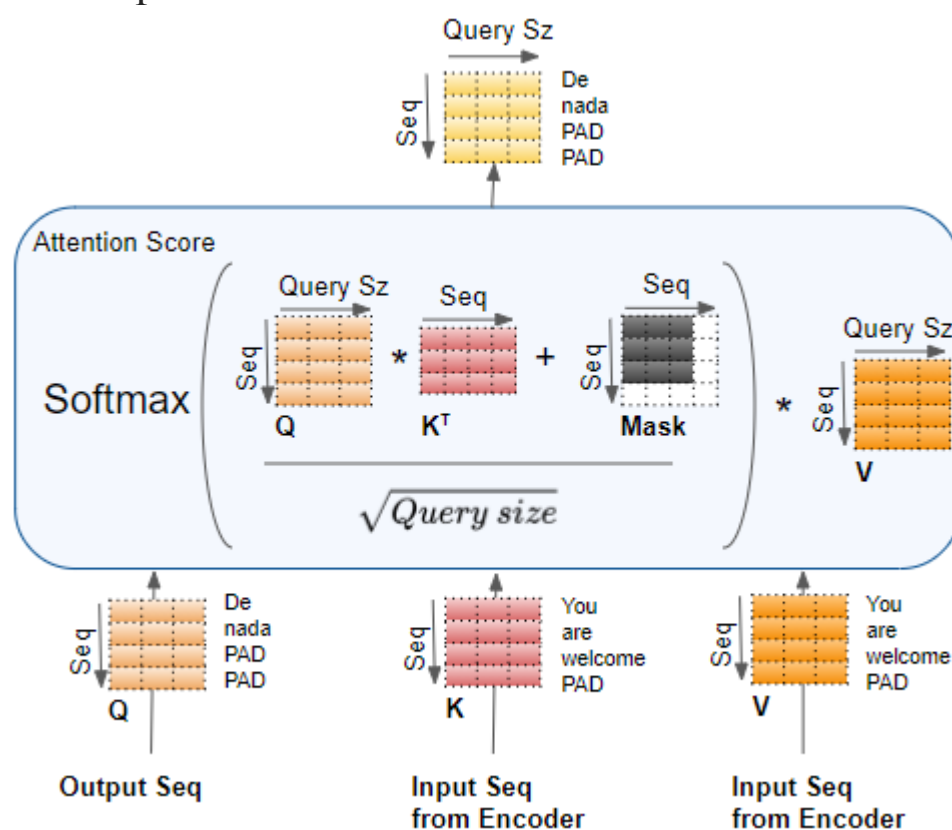


(Image by Author)

Similarly, the Masking masks out the Padding words in the target sequence.

## Decoder Encoder-Decoder Attention and Masking

The Encoder-Decoder Attention takes its input from two sources. Therefore, unlike the Encoder Self-Attention, which computes the interaction between each input word with other input words, and Decoder Self-Attention which computes the interaction between each target word with other target words, the Encoder-Decoder Attention computes the interaction between each target word with each input word.



(Image by Author)

Therefore each cell in the resulting Attention Score corresponds to the interaction between one Q (ie. target sequence word) with all other K (ie. input sequence) words and all V (ie. input sequence) words.

Similarly, the Masking masks out the later words in the target output, as was explained in detail in the [second article](#) of the series.