INTUITIVE TRANSFORMERS SERIES NLP

Transformers Explained Visually (Part 1): Overview of Functionality

A Gentle Guide to Transformers, how they are used for NLP, and why they are better than RNNs, in Plain English. How Attention helps improve performance.

Ketan Doshi · Follow
Published in Towards Data Science
10 min read · Dec 13, 2020

Photo by Arseny Togulev on Unsplash

We've been hearing a lot about Transformers and with good reason. They have taken the world of NLP by storm in the last few years. The Transformer is an architecture that uses Attention to significantly improve the performance of deep learning NLP translation models. It was first introduced in the paper Attention is all you need and was quickly established as the leading architecture for most text data applications.

Since then, numerous projects including Google's BERT and OpenAI's GPT series have built on this foundation and published performance results that handily beat existing state-of-the-art benchmarks.

Over a series of articles, I'll go over the basics of Transformers, its architecture, and how it works internally. We will cover the Transformer functionality in a top-down manner. In later articles, we will look under the covers to understand the operation of the system in detail. We will also do a deep dive into the workings of the multi-head attention, which is the heart 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.

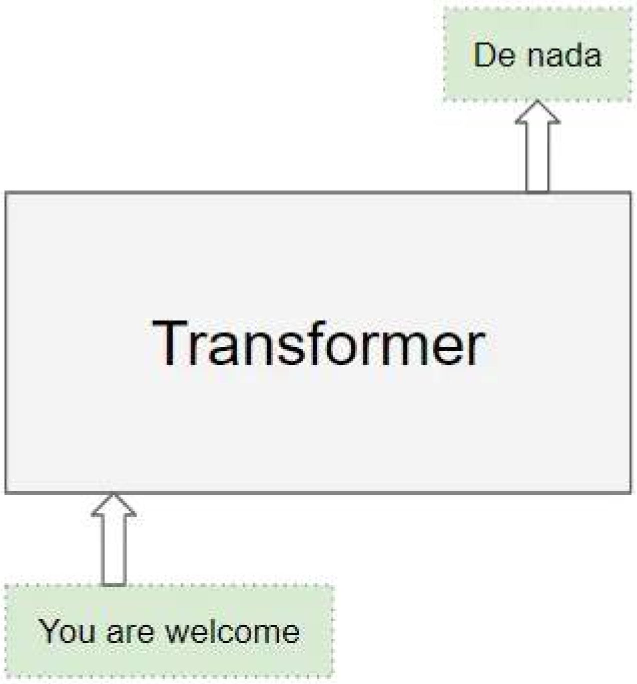
1. **Overview of functionality — this article** *(How Transformers are used, and why they are better than RNNs. Components of the architecture, and behavior during Training and Inference)*
2. How it works *(Internal operation end-to-end. How data flows and what computations are performed, including matrix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

What is a Transformer

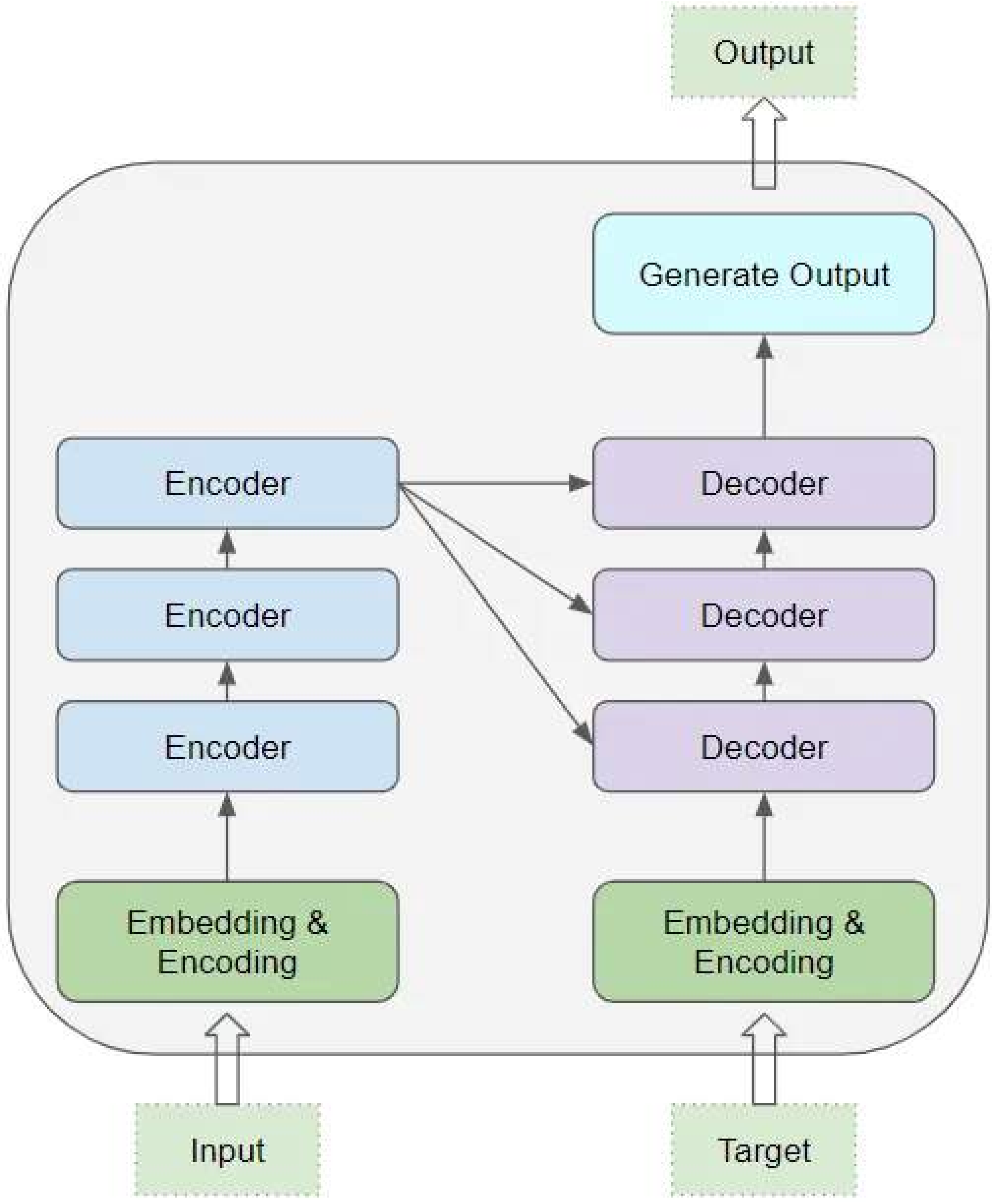
The Transformer architecture excels at handling text data which is inherently sequential. They take a text sequence as input and produce another text sequence as output. eg. to translate an input English sentence to Spanish.



(Image by Author)

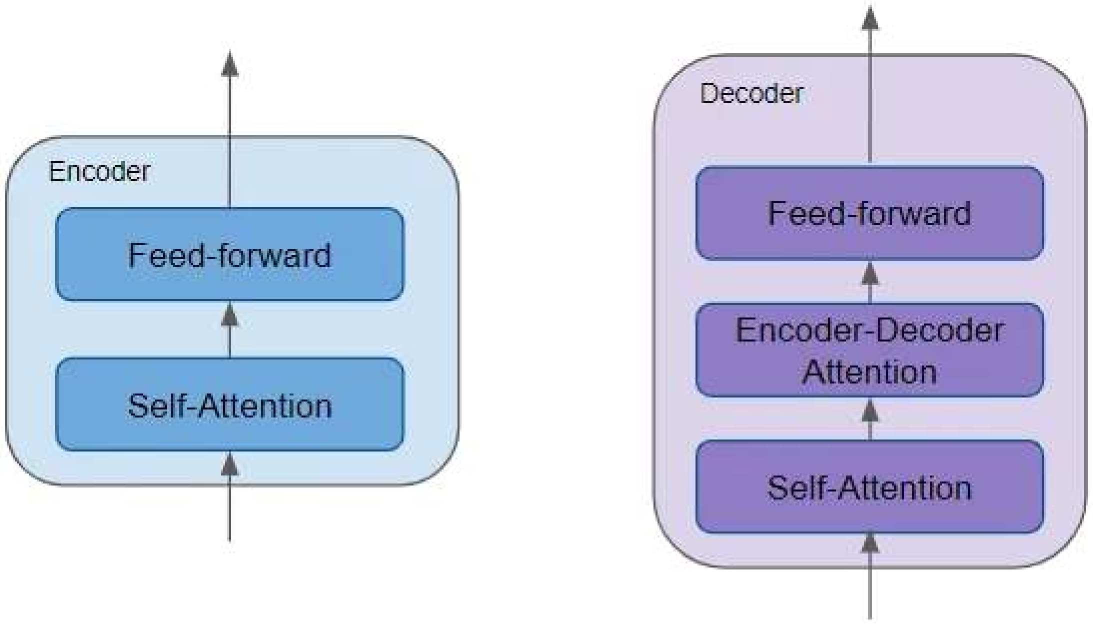
At its core, it contains a stack of Encoder layers and Decoder layers. To avoid confusion we will refer to the individual layer as an Encoder or a Decoder and will use Encoder stack or Decoder stack for a group of Encoder layers.

The Encoder stack and the Decoder stack each have their corresponding Embedding layers for their respective inputs. Finally, there is an Output layer to generate the final output.



(Image by Author)

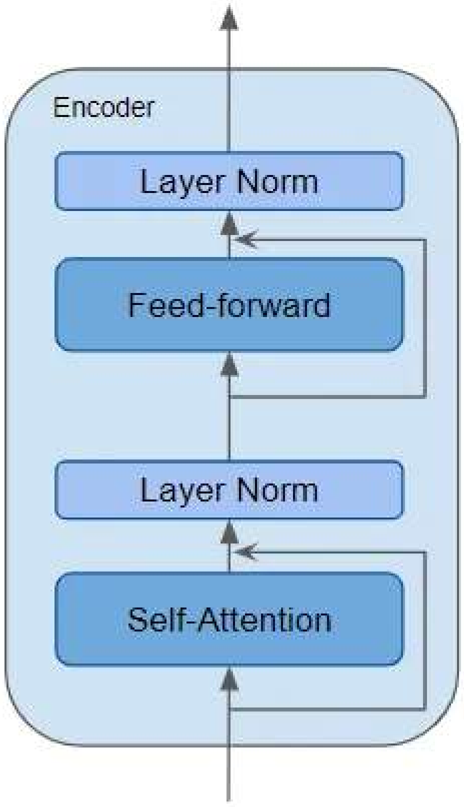
All the Encoders are identical to one another. Similarly, all the Decoders are identical.



(Image by Author)

- The Encoder contains the all-important Self-attention layer that computes the relationship between different words in the sequence, as well as a Feed-forward layer.
- The Decoder contains the Self-attention layer and the Feed-forward layer, as well as a second Encoder-Decoder attention layer.
- Each Encoder and Decoder has its own set of weights.

The Encoder is a reusable module that is the defining component of all Transformer architectures. In addition to the above two layers, it also has Residual skip connections around both layers along with two LayerNorm layers.



(Image by Author)

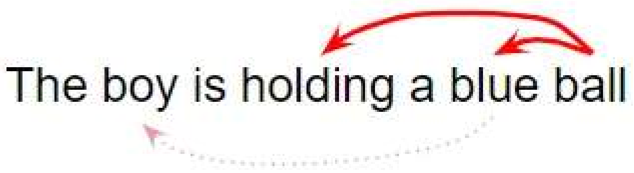
There are many variations of the Transformer architecture. Some Transformer architectures have no Decoder at all and rely only on the Encoder.

What does Attention Do?

The key to the Transformer's ground-breaking performance is its use of Attention.

While processing a word, Attention enables the model to focus on other words in the input that are closely related to that word.

eg. 'Ball' is closely related to 'blue' and 'holding'. On the other hand, 'blue' is not related to 'boy'.

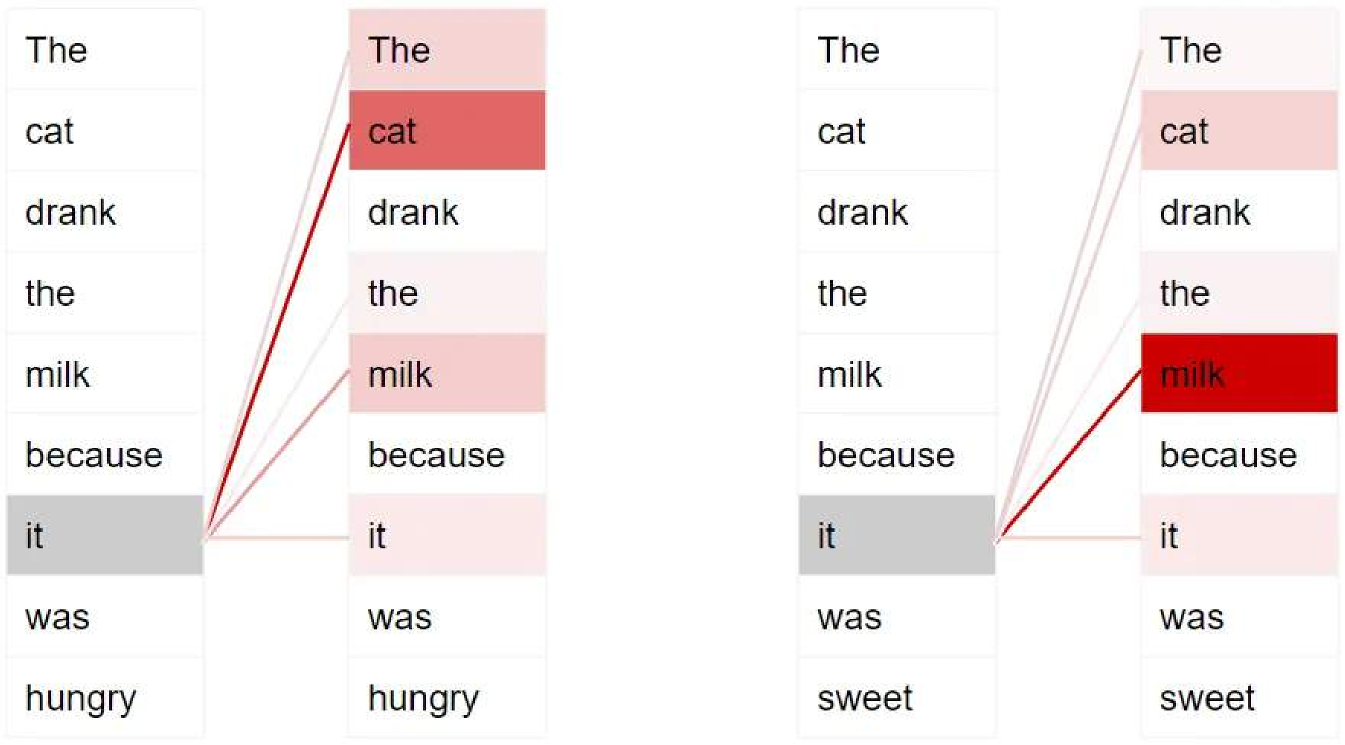


The Transformer architecture uses self-attention by relating every word in the input sequence to every other word.

eg. Consider two sentences:

- The *cat* drank the milk because **it** was hungry.
- The cat drank the *milk* because **it** was sweet.

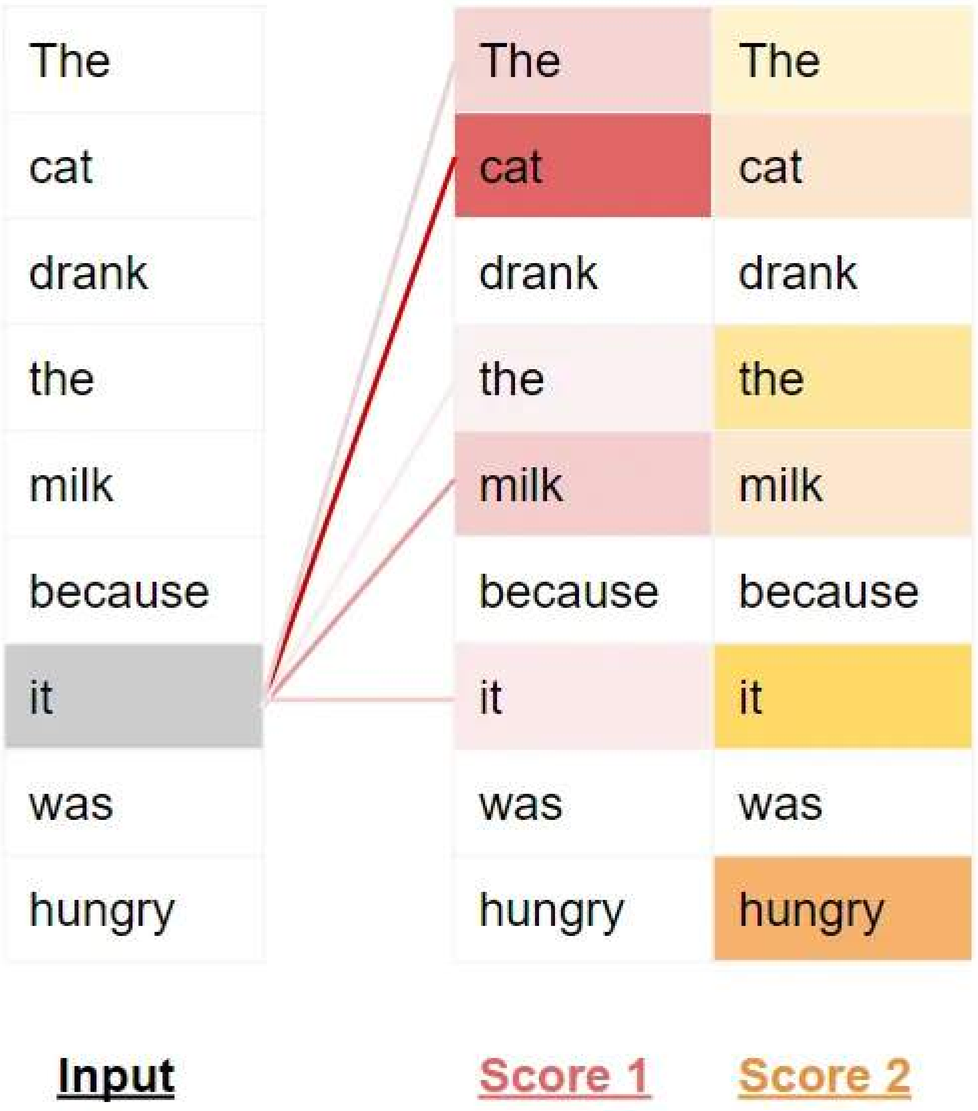
In the first sentence, the word 'it' refers to 'cat', while in the second it refers to 'milk. When the model processes the word 'it', self-attention gives the model more information about its meaning so that it can associate 'it' with the correct word.



Dark colors represent higher attention (Image by Author)

To enable it to handle more nuances about the intent and semantics of the sentence, Transformers include multiple attention scores for each word.

eg. While processing the word 'it', the first score highlights 'cat', while the second score highlights 'hungry'. So when it decodes the word 'it', by translating it into a different language, for instance, it will incorporate some aspect of both 'cat' and 'hungry' into the translated word.



(Image by Author)

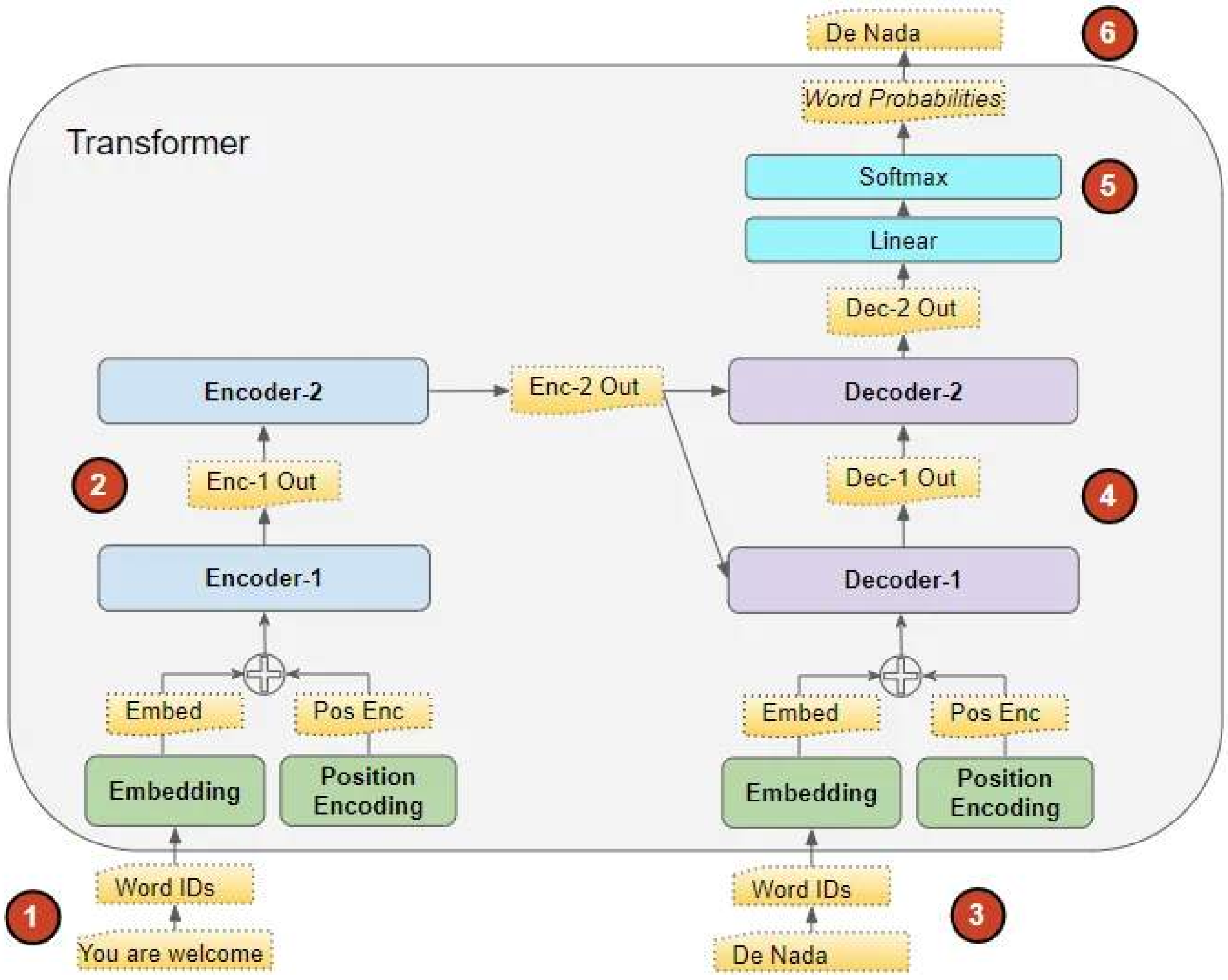
Training the Transformer

The Transformer works slightly differently during Training and while doing Inference.

Let’s first look at the flow of data during Training. Training data consists of two parts:

- The source or input sequence (eg. “You are welcome” in English, for a translation problem)
- The destination or target sequence (eg. “De nada” in Spanish)

The Transformer’s goal is to learn how to output the target sequence, by using both the input and target sequence.



(Image by Author)

The Transformer processes the data like this:

1. The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.

2. The stack of Encoders processes this and produces an encoded representation of the input sequence.

3. The target sequence is prepended with a start-of-sentence token, converted into Embeddings (with Position Encoding), and fed to the Decoder.

4. The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.

5. The Output layer converts it into word probabilities and the final output sequence.

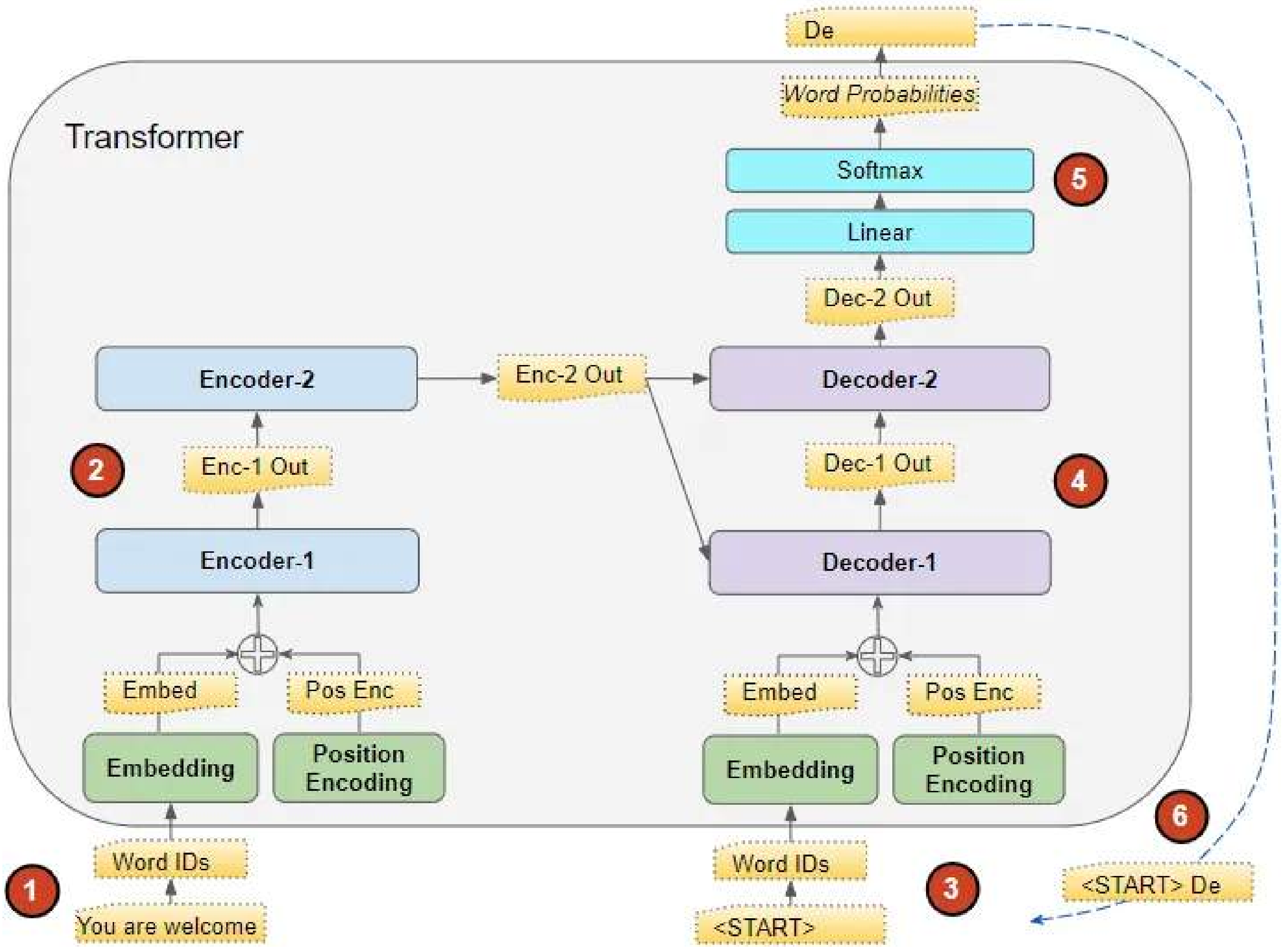
6. The Transformer's Loss function compares this output sequence with the target sequence from the training data. This loss is used to generate gradients to train the Transformer during back-propagation.

Inference

During Inference, we have only the input sequence and don't have the target sequence to pass as input to the Decoder. The goal of the Transformer is to produce the target sequence from the input sequence alone.

So, like in a Seq2Seq model, we generate the output in a loop and feed the output sequence from the previous timestep to the Decoder in the next timestep until we come across an end-of-sentence token.

The difference from the Seq2Seq model is that, at each timestep, we re-feed the entire output sequence generated thus far, rather than just the last word.



Inference flow, after first timestep (Image by Author)

The flow of data during Inference is:

1. The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.

2. The stack of Encoders processes this and produces an encoded representation of the input sequence.

3. Instead of the target sequence, we use an empty sequence with only a start-of-sentence token. This is converted into Embeddings (with Position Encoding) and fed to the Decoder.

4. The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.

5. The Output layer converts it into word probabilities and produces an output sequence.

6. We take the last word of the output sequence as the predicted word. That word is now filled into the second position of our Decoder input sequence, which now contains a start-of-sentence token and the first word.

7. Go back to step #3. As before, feed the new Decoder sequence into the model. Then take the second word of the output and append it to the Decoder sequence. Repeat this until it predicts an end-of-sentence token. Note that since the Encoder sequence does not change for each iteration, we do not have to repeat steps #1 and #2 each time (*Thanks to Michal Kučírka for pointing this out*).

Teacher Forcing

The approach of feeding the target sequence to the Decoder during training is known as Teacher Forcing. Why do we do this and what does that term mean?

During training, we could have used the same approach that is used during inference. In other words, run the Transformer in a loop, take the last word from the output sequence, append it to the Decoder input and feed it to the Decoder for the next iteration. Finally, when the end-of-sentence token is predicted, the Loss function would compare the generated output sequence to the target sequence in order to train the network.

Not only would this looping cause training to take much longer, but it also makes it harder to train the model. The model would have to predict the second word based on a potentially erroneous first predicted word, and so on.

Instead, by feeding the target sequence to the Decoder, we are giving it a hint, so to speak, just like a Teacher would. Even though it predicted an erroneous first word, it can instead use the correct first word to predict the second word so that those errors don't keep compounding.

In addition, the Transformer is able to output all the words in parallel without looping, which greatly speeds up training.

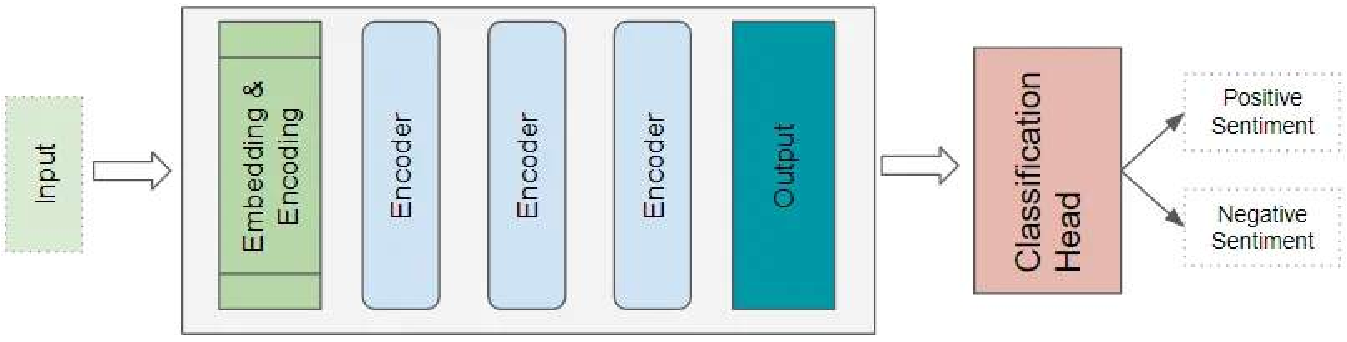
What are Transformers used for?

Transformers are very versatile and are used for most NLP tasks such as language models and text classification. They are frequently used in sequence-to-sequence models for applications such as Machine Translation, Text Summarization, Question-Answering, Named Entity Recognition, and Speech Recognition.

There are different flavors of the Transformer architecture for different problems. The basic Encoder Layer is used as a common building block for these architectures, with different application-specific 'heads' depending on the problem being solved.

Transformer Classification architecture

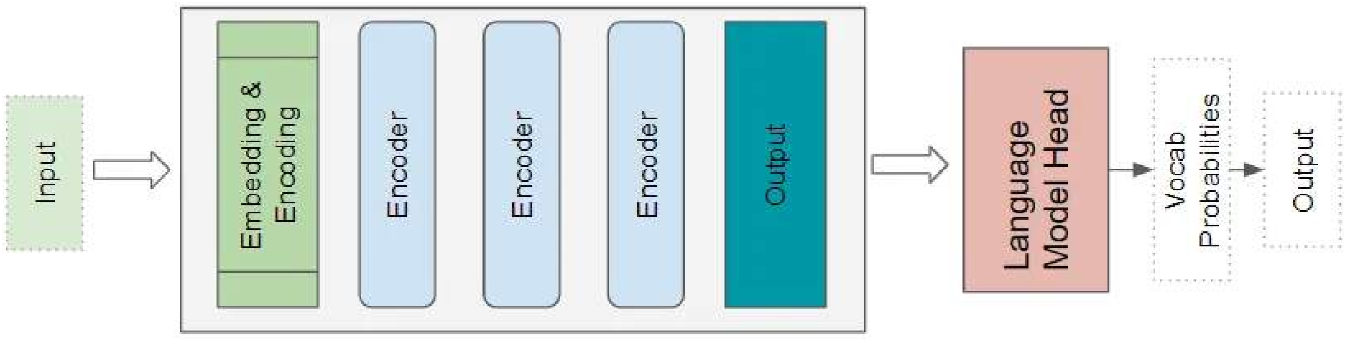
A Sentiment Analysis application, for instance, would take a text document as input. A Classification head takes the Transformer's output and generates predictions of the class labels such as a positive or negative sentiment.



(Image by Author)

Transformer Language Model architecture

A Language Model architecture would take the initial part of an input sequence such as a text sentence as input, and generate new text by predicting sentences that would follow. A Language Model head takes the Transformer's output and generates a probability for every word in the vocabulary. The highest probability word becomes the predicted output for the next word in the sentence.



(Image by Author)

How are they better than RNNs?

RNNs and their cousins, LSTMs and GRUs, were the de facto architecture for all NLP applications until Transformers came along and dethroned them.

RNN-based sequence-to-sequence models performed well, and when the Attention mechanism was first introduced, it was used to enhance their performance.

However, they had two limitations:

- It was challenging to deal with long-range dependencies between words that were spread far apart in a long sentence.
- They process the input sequence sequentially one word at a time, which means that it cannot do the computation for time-step *t* until it has completed the computation for time-step *t — 1*. This slows down training and inference.

As an aside, with CNNs, all of the outputs can be computed in parallel, which makes convolutions much faster. However, they also have limitations in dealing with long-range dependencies:

- In a convolutional layer, only parts of the image (or words if applied to text data) that are close enough to fit within the kernel size can interact with each other. For items that are further apart, you need a much deeper network with many layers.

The Transformer architecture addresses both of these limitations. It got rid of RNNs altogether and relied exclusively on the benefits of Attention.

- They process all the words in the sequence in parallel, thus greatly speeding up computation.

INTUITIVE TRANSFORMERS SERIES NLP

Transformers Explained Visually (Part 2): How it works, step-by-step

A Gentle Guide to the Transformer under the hood, and its end-to-end operation.

Ketan Doshi · Follow
Published in Towards Data Science
11 min read · Jan 2, 2021

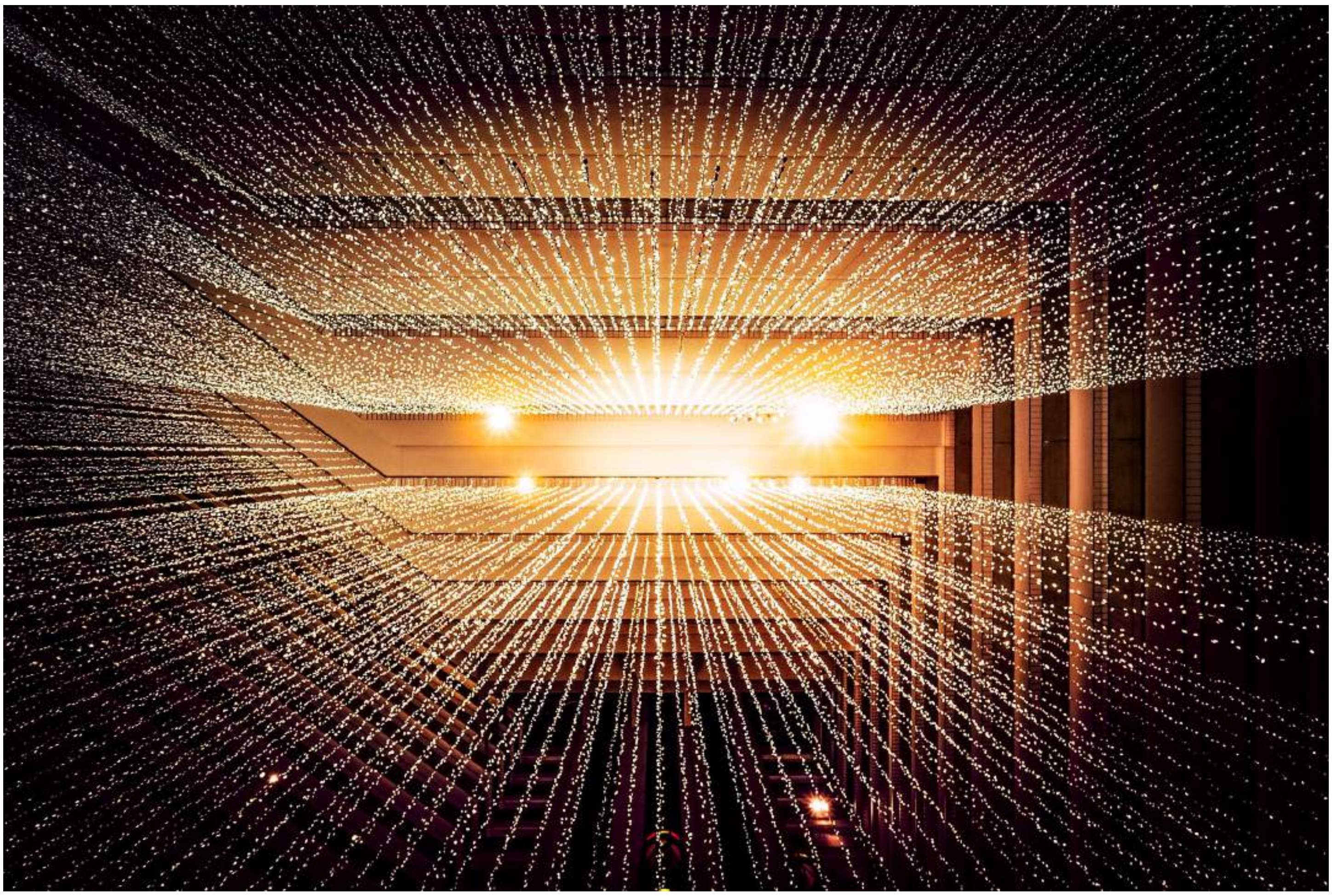


Photo by Joshua Sortino on Unsplash

This is the second article in my series on Transformers. In the first article, we learned about the functionality of Transformers, how they are used, their high-level architecture, and their advantages.

In this article, we can now look under the hood and study exactly how they work in detail. We'll see how data flows through the system with their actual matrix representations and shapes and understand the computations performed at each stage.

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.

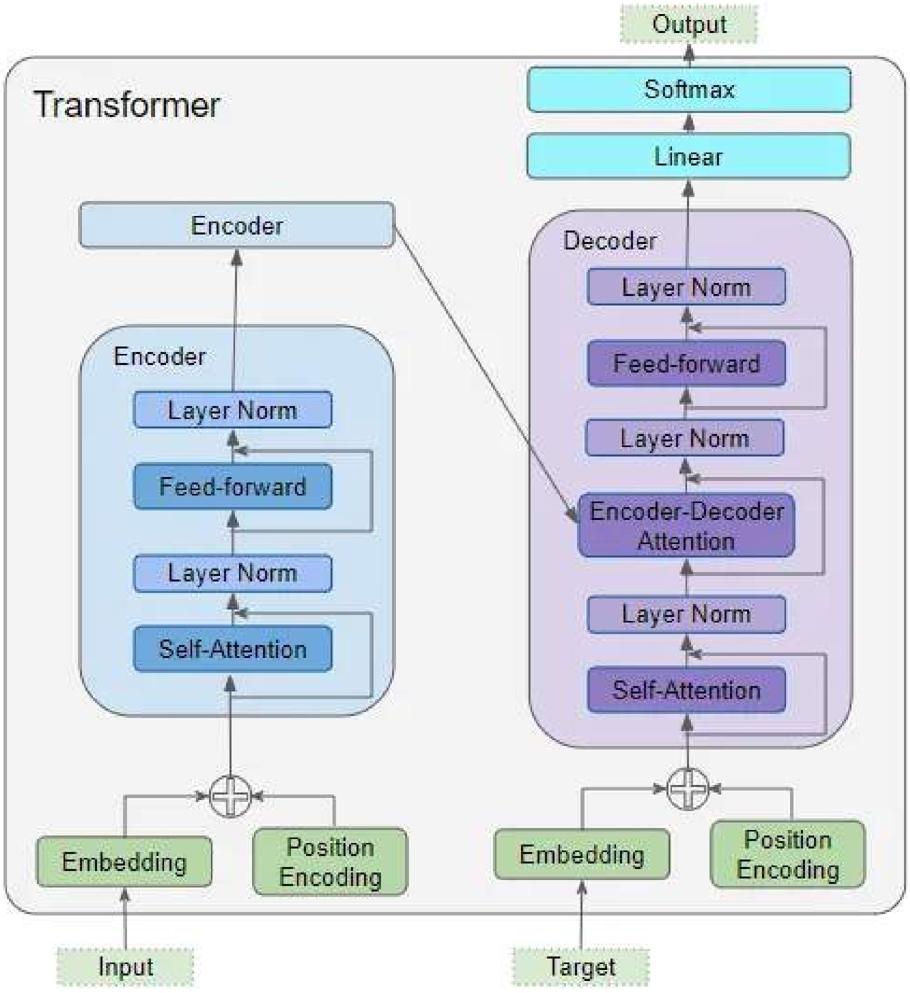
1. Overview of functionality *(How Transformers are used, and why they are better than RNNs. Components of the architecture, and behavior during Training and Inference)*
2. **How it works — this article** *(Internal operation end-to-end. How data flows and what computations are performed, including matrix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

Architecture Overview

As we saw in Part 1, the main components of the architecture are:



(Image by Author)

Data inputs for both the Encoder and Decoder, which contains:

- Embedding layer
- Position Encoding layer

The Encoder stack contains a number of Encoders. Each Encoder contains:

- Multi-Head Attention layer
- Feed-forward layer

The Decoder stack contains a number of Decoders. Each Decoder contains:

- Two Multi-Head Attention layers
- Feed-forward layer

Output (top right) — generates the final output, and contains:

- Linear layer
- Softmax layer.

To understand what each component does, let's walk through the working of the Transformer while we are training it 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).

Embedding and Position Encoding

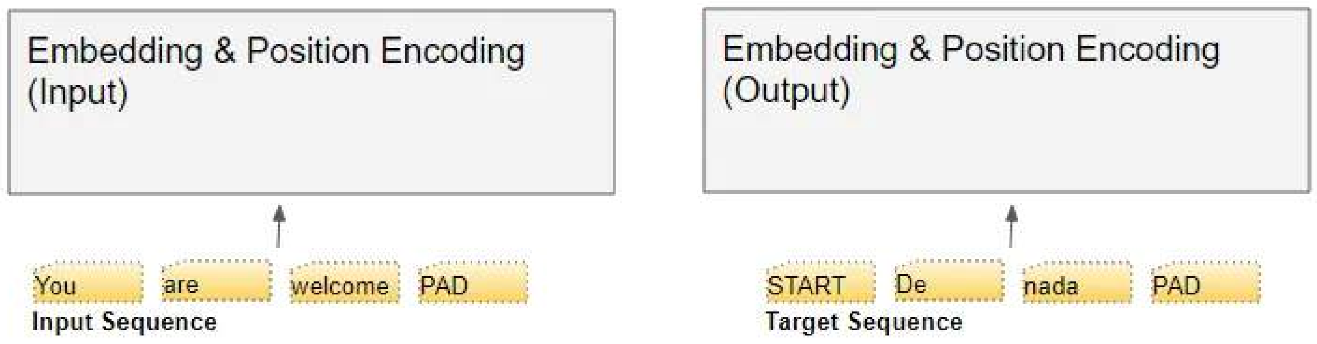
Like any NLP model, the Transformer needs two things about each word — the meaning of the word and its position in the sequence.

- The Embedding layer encodes the meaning of the word.
- The Position Encoding layer represents the position of the word.

The Transformer combines these two encodings by adding them.

Embedding

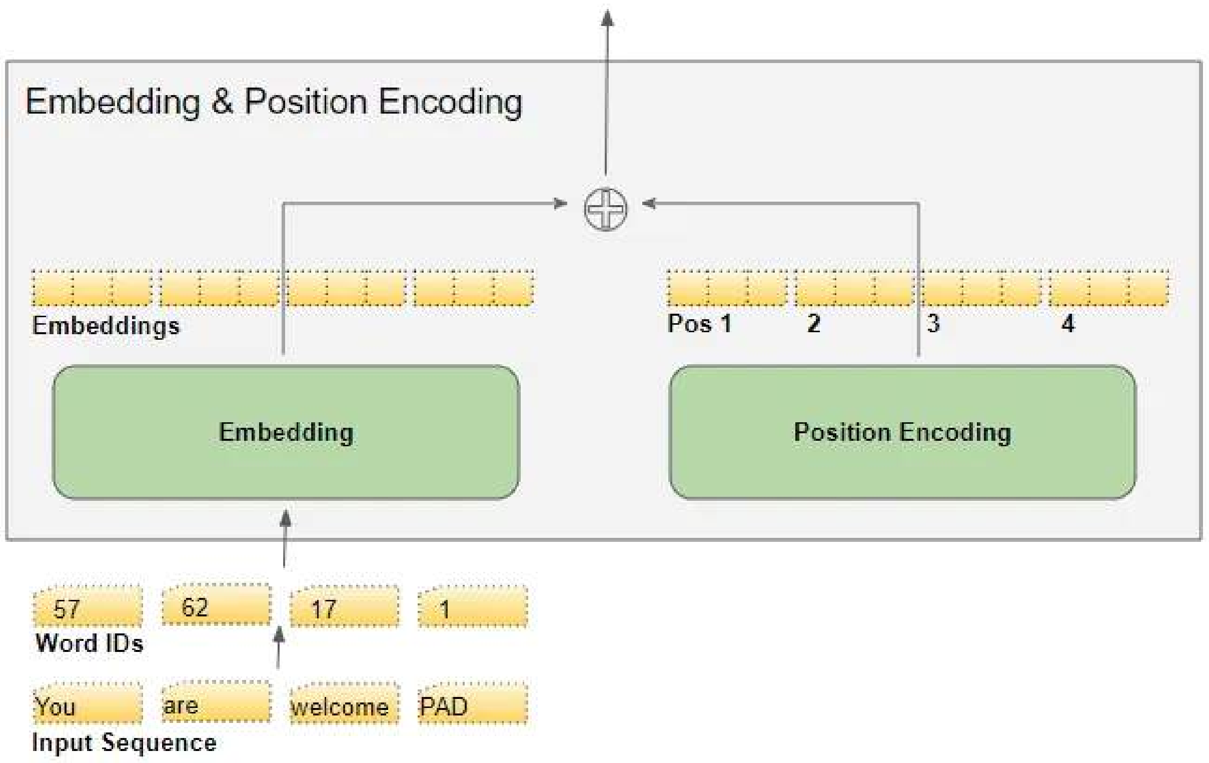
The Transformer has two Embedding layers. The input sequence is fed to the first Embedding layer, known as the Input Embedding.


Embedding & Position Encoding (Input)
You
are
welcome
PAD
Input Sequence
Embedding & Position Encoding (Output)
START
De
nada
PAD
Target Sequence

(Image by Author)

The target sequence is fed to the second Embedding layer after shifting the targets right by one position and inserting a Start token in the first position. Note that, during Inference, we have no target sequence and we feed the output sequence to this second layer in a loop, as we learned in Part 1. That is why it is called the Output Embedding.

The text sequence is mapped to numeric word IDs using our vocabulary. The embedding layer then maps each input word into an embedding vector, which is a richer representation of the meaning of that word.



(Image by Author)

Position Encoding

Since an RNN implements a loop where each word is input sequentially, it implicitly knows the position of each word.

However, Transformers don't use RNNs and all words in a sequence are input in parallel. This is its major advantage over the RNN architecture, but it means that the position information is lost, and has to be added back in separately.

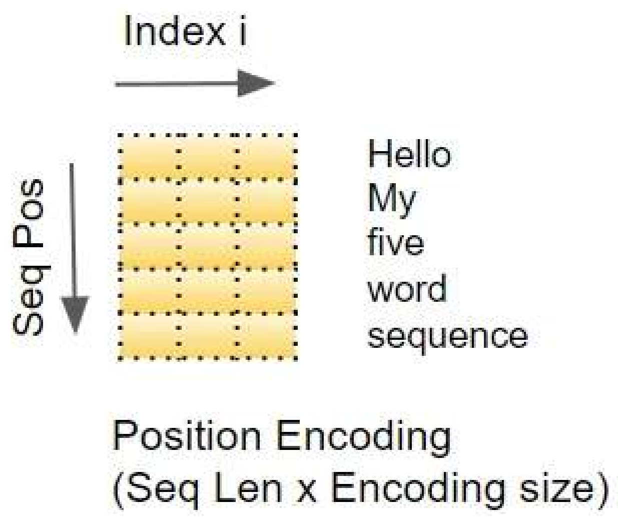
Just like the two Embedding layers, there are two Position Encoding layers. The Position Encoding is computed independently of the input sequence. These are fixed values that depend only on the max length of the sequence. For instance,

- the first item is a constant code that indicates the first position
- the second item is a constant code that indicates the second position,
- and so on.

These constants are computed using the formula below, where

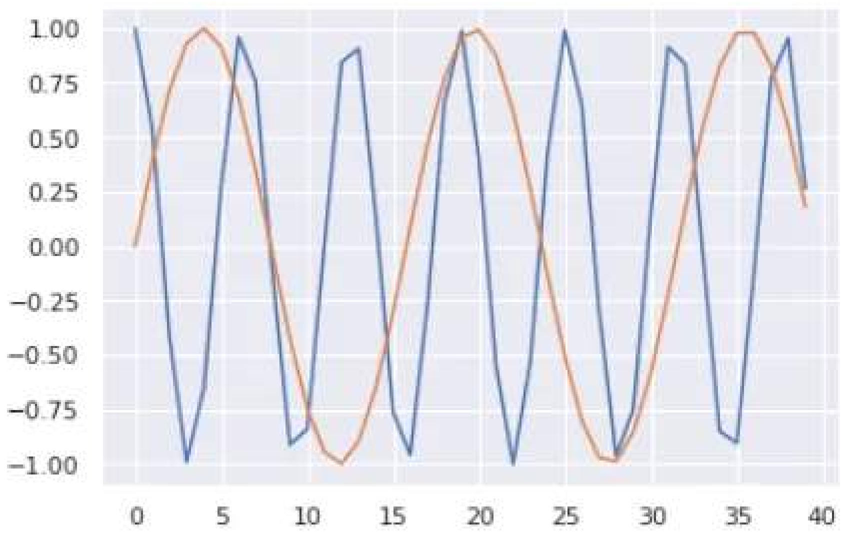
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

- *pos* is the position of the word in the sequence
- *d_model* is the length of the encoding vector (same as the embedding vector) and
- *i* is the index value into this vector.



(Image by Author)

In other words, it interleaves a sine curve and a cos curve, with sine values for all even indexes and cos values for all odd indexes. As an example, if we encode a sequence of 40 words, we can see below the encoding values for a few (word position, encoding_index) combinations.

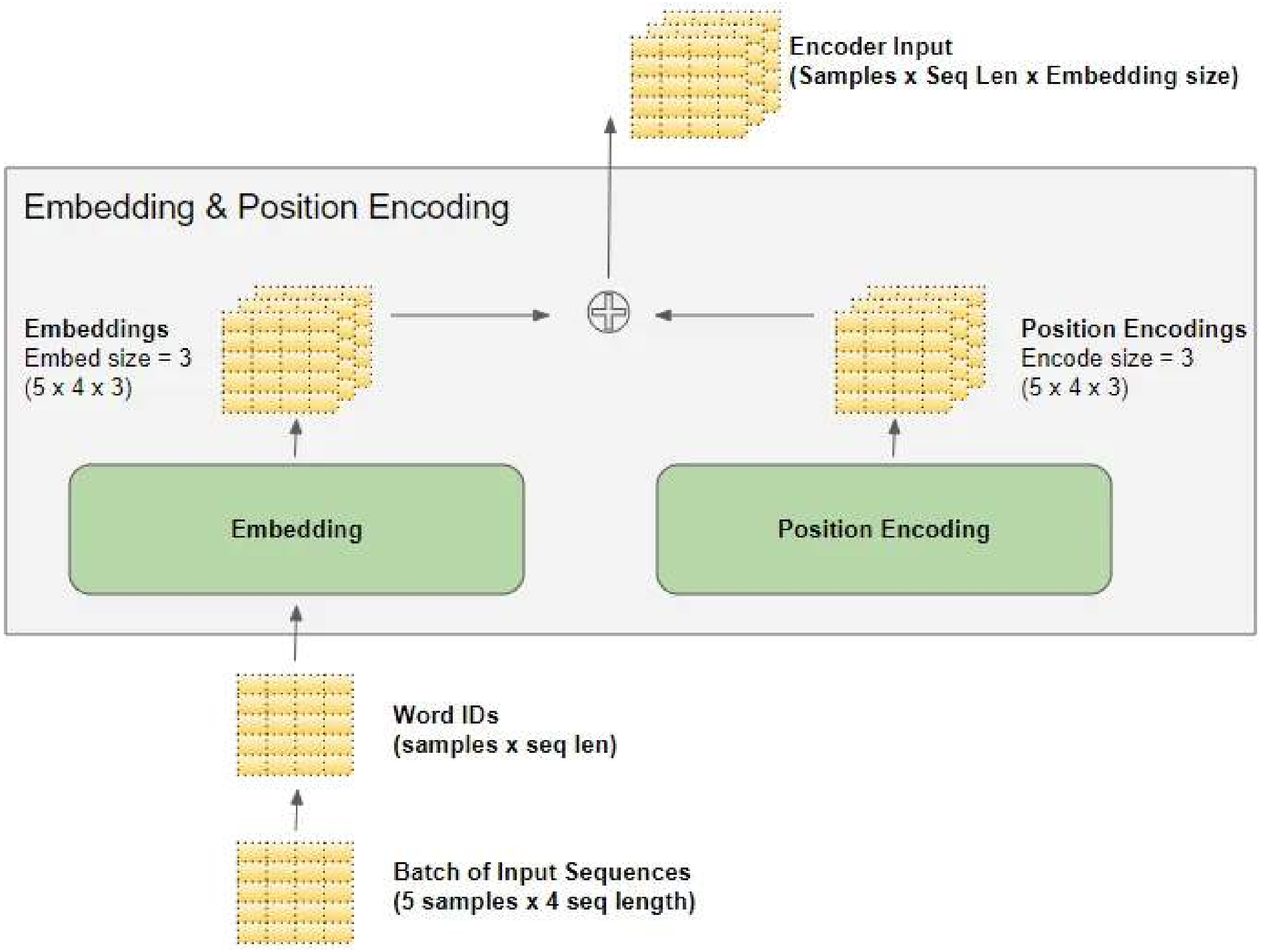


(Image by Author)

The blue curve shows the encoding of the 0th index for all 40 word-positions and the orange curve shows the encoding of the 1st index for all 40 word-positions. There will be similar curves for the remaining index values.

Matrix Dimensions

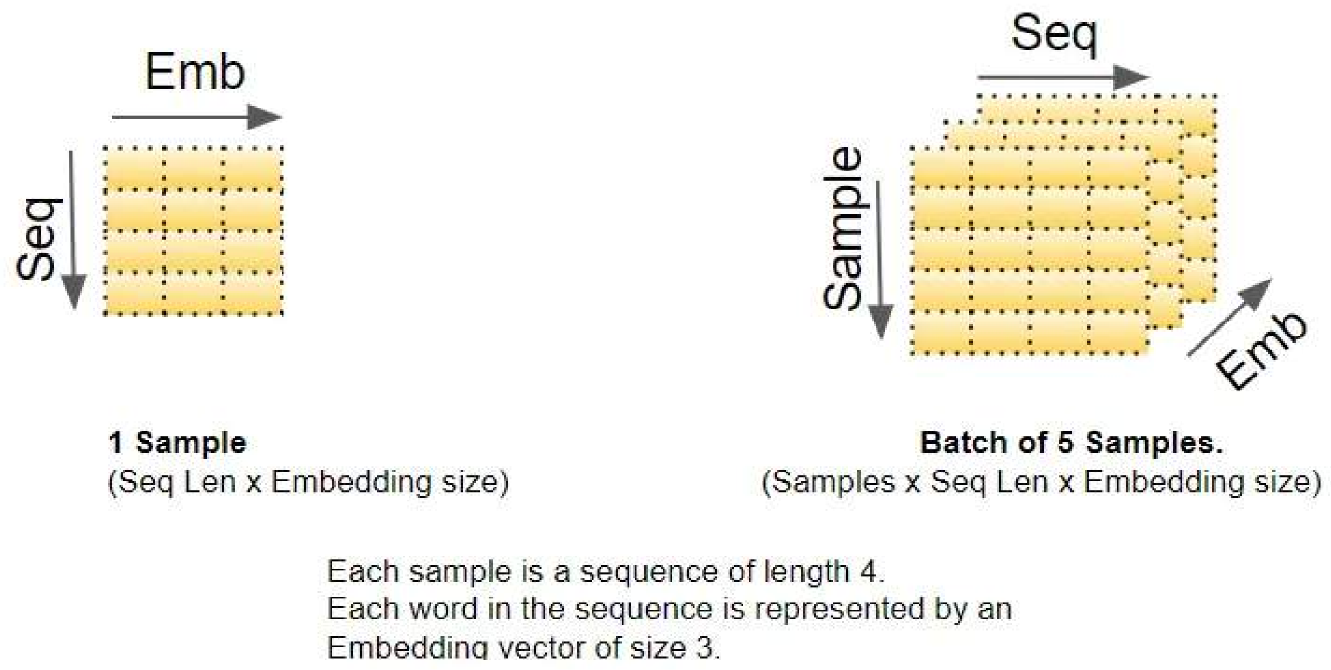
As we know, deep learning models process a batch of training samples at a time. The Embedding and Position Encoding layers operate on matrices representing a batch of sequence samples. The Embedding takes a (samples, sequence length) shaped matrix of word IDs. It encodes each word ID into a word vector whose length is the embedding size, resulting in a (samples, sequence length, embedding size) shaped output matrix. The Position Encoding uses an encoding size that is equal to the embedding size. So it produces a similarly shaped matrix that can be added to the embedding matrix.



(Image by Author)

The (samples, sequence length, embedding size) shape produced by the Embedding and Position Encoding layers is preserved all through the Transformer, as the data flows through the Encoder and Decoder Stacks until it is reshaped by the final Output layers.

This gives a sense of the 3D matrix dimensions in the Transformer. However, to simplify the visualization, from here on we will drop the first dimension (for the samples) and use the 2D representation for a single sample.

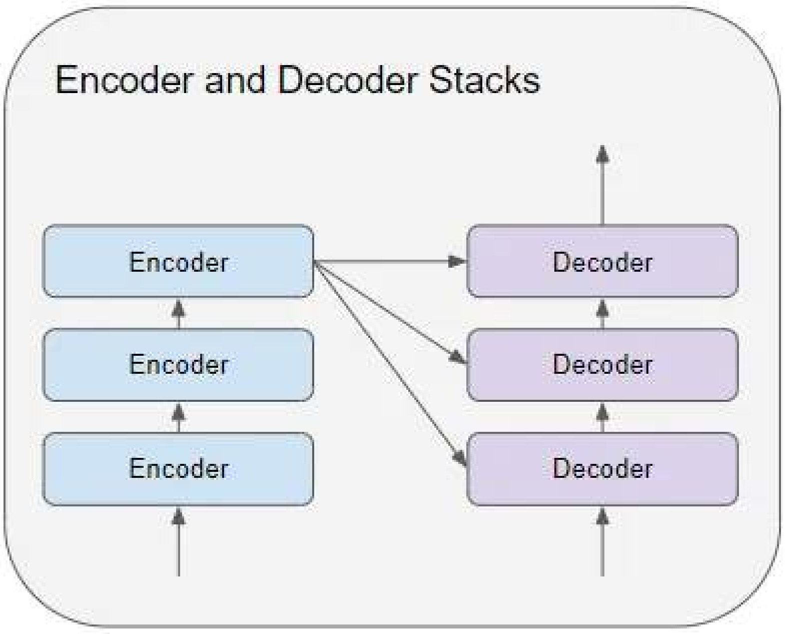

Emb
Seq
Seq
Sample
Emb
1 Sample
(Seq Len x Embedding size)
Batch of 5 Samples.
(Samples x Seq Len x Embedding size)
Each sample is a sequence of length 4.
Each word in the sequence is represented by an Embedding vector of size 3.

(Image by Author)

The Input Embedding sends its outputs into the Encoder. Similarly, the Output Embedding feeds into the Decoder.

Encoder

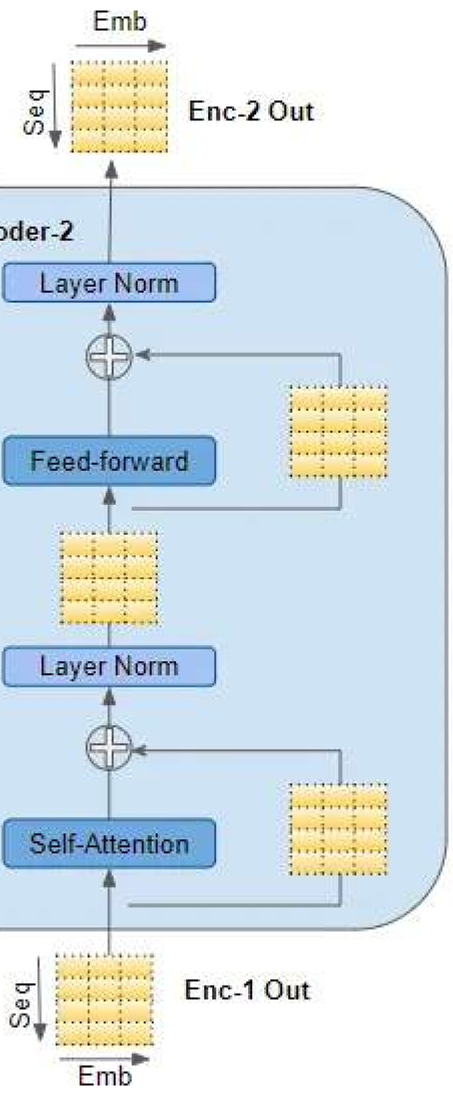
The Encoder and Decoder Stacks consists of several (usually six) Encoders and Decoders respectively, connected sequentially.



(Image by Author)

The first Encoder in the stack receives its input from the Embedding and Position Encoding. The other Encoders in the stack receive their input from the previous Encoder.

The Encoder passes its input into a Multi-head Self-attention layer. The Self-attention output is passed into a Feed-forward layer, which then sends its output upwards to the next Encoder.



(Image by Author)

Both the Self-attention and Feed-forward sub-layers, have a residual skip-connection around them, followed by a Layer-Normalization.

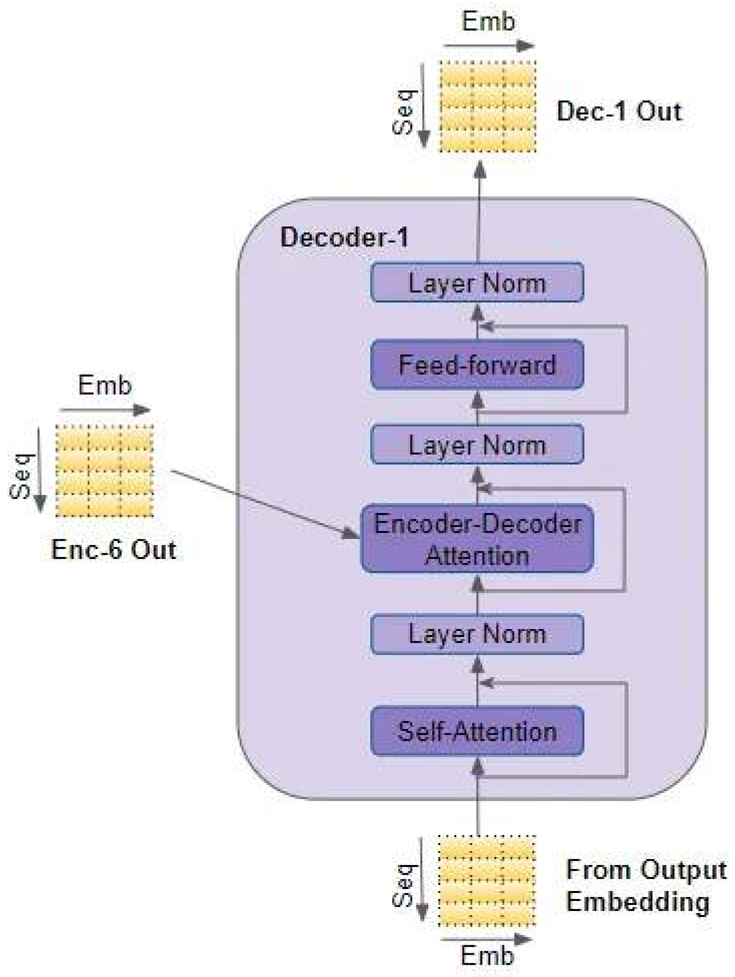
The output of the last Encoder is fed into each Decoder in the Decoder Stack as explained below.

Decoder

The Decoder's structure is very similar to the Encoder's but with a couple of differences.

Like the Encoder, the first Decoder in the stack receives its input from the Output Embedding and Position Encoding. The other Decoders in the stack receive their input from the previous Decoder.

The Decoder passes its input into a Multi-head Self-attention layer. This operates in a slightly different way than the one in the Encoder. It is only allowed to attend to earlier positions in the sequence. This is done by masking future positions, which we'll talk about shortly.



(Image by Author)

Unlike the Encoder, the Decoder has a second Multi-head attention layer, known as the Encoder-Decoder attention layer. The Encoder-Decoder attention layer works like Self-attention, except that it combines two sources of inputs — the Self-attention layer below it as well as the output of the Encoder stack.

The Self-attention output is passed into a Feed-forward layer, which then sends its output upwards to the next Decoder.

Each of these sub-layers, Self-attention, Encoder-Decoder attention, and Feed-forward, have a residual skip-connection around them, followed by a Layer-Normalization.

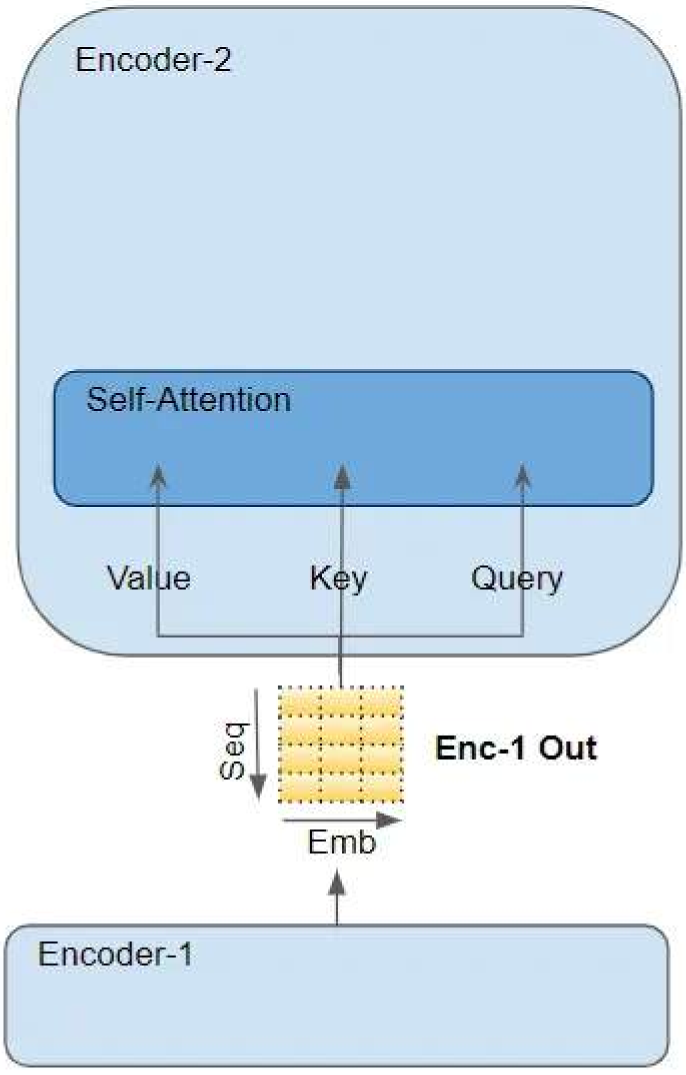
Attention

In Part 1, we talked about why Attention is so important while processing sequences. In the Transformer, Attention is used 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

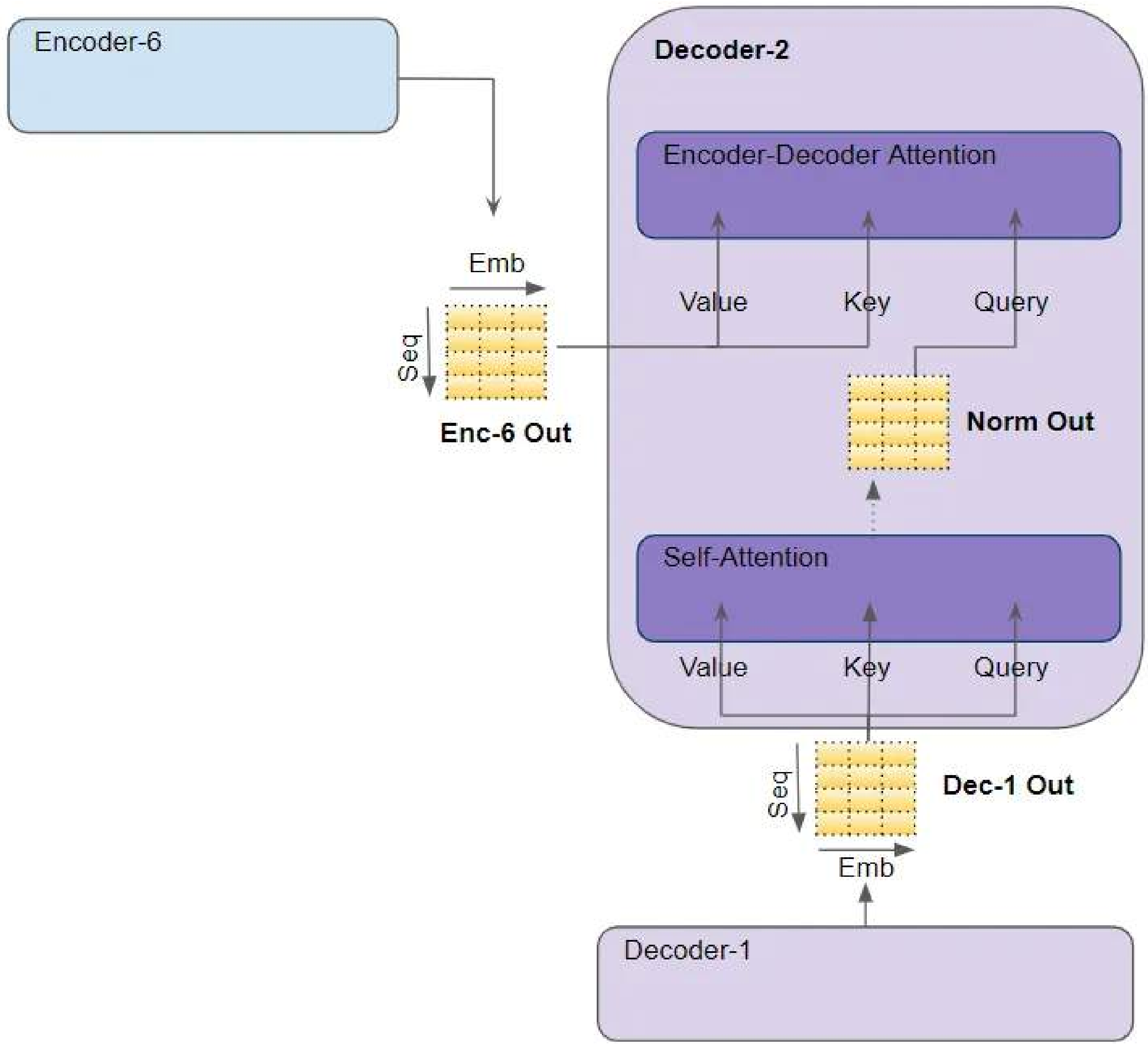
The Attention layer takes its input in the form of three parameters, known as the Query, Key, and Value.

- In the Encoder's Self-attention, the Encoder's input is passed to all three parameters, Query, Key, and Value.



(Image by Author)

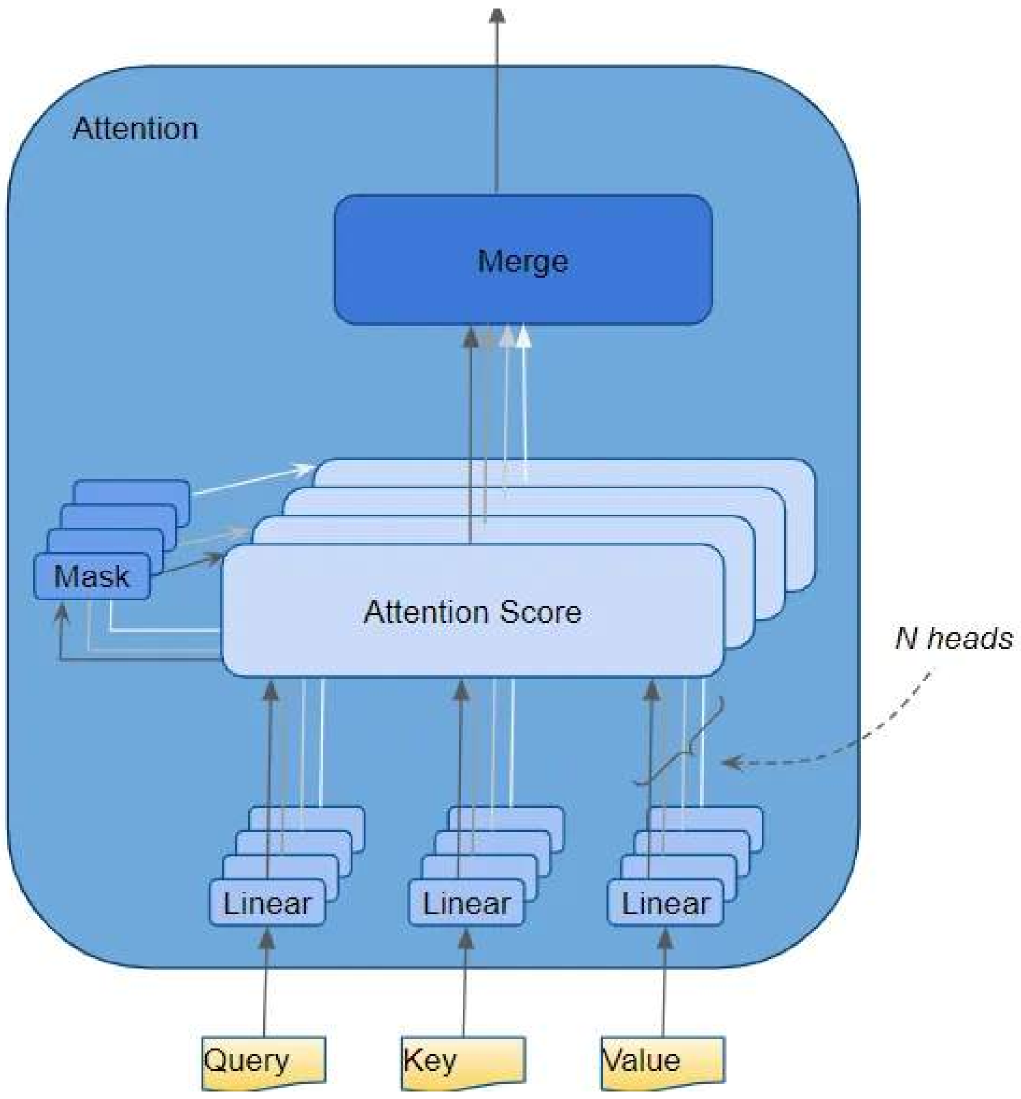
- In the Decoder's Self-attention, the Decoder's input is passed to all three parameters, Query, Key, and Value.
- In the Decoder's Encoder-Decoder attention, the output of the final Encoder in the stack is passed to the Value and Key parameters. The output of the Self-attention (and Layer Norm) module below it is passed to the Query parameter.



(Image by Author)

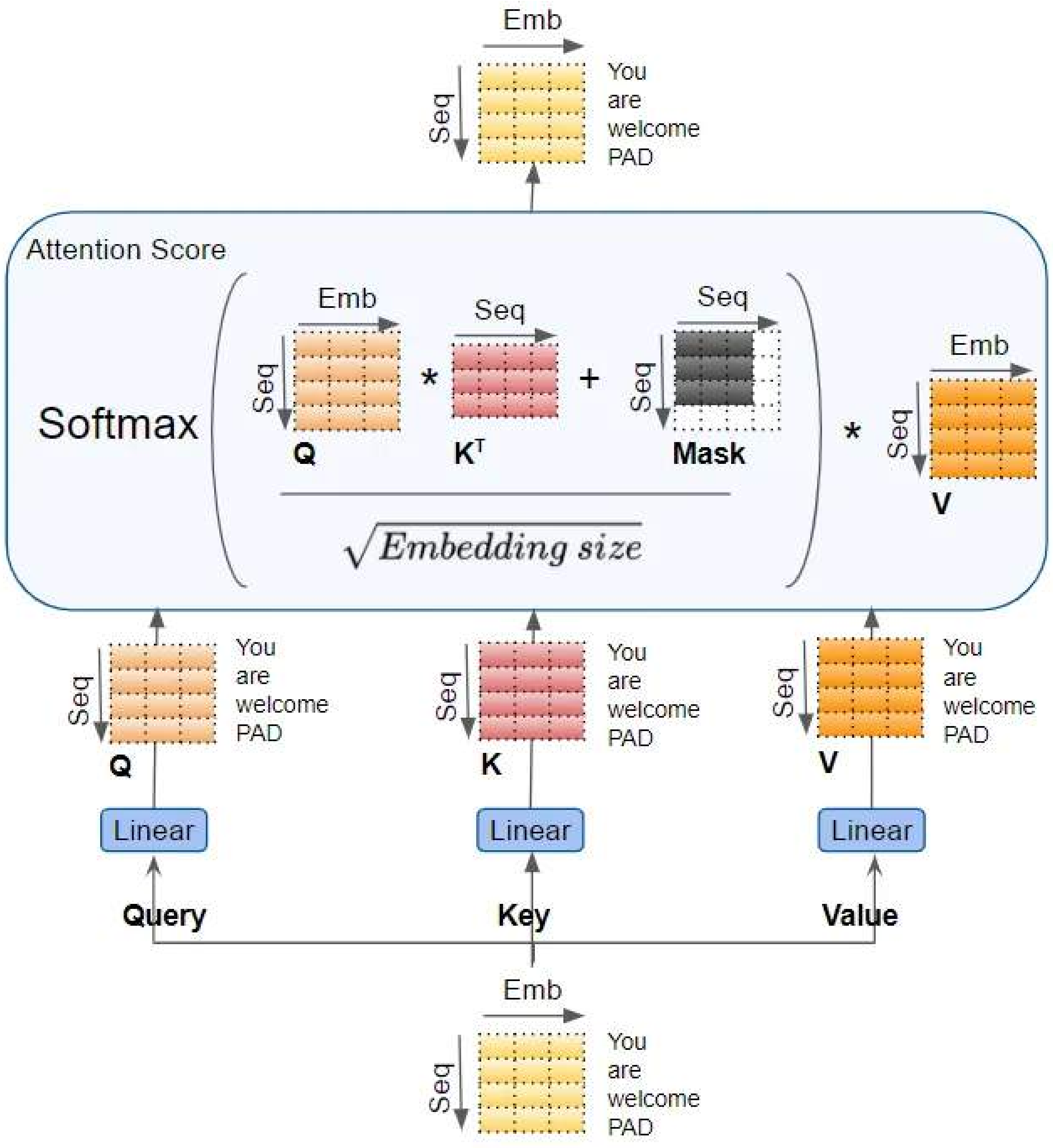
Multi-head Attention

The Transformer calls each Attention processor an Attention Head and repeats it several times in parallel. This is known as Multi-head attention. It gives its Attention greater power of discrimination, by combining several similar Attention calculations.



(Image by Author)

The Query, Key, and Value are each passed through separate Linear layers, each with their own weights, producing three results called Q, K, and V respectively. These are then combined together using the Attention formula as shown below, to produce the Attention Score.



(Image by Author)

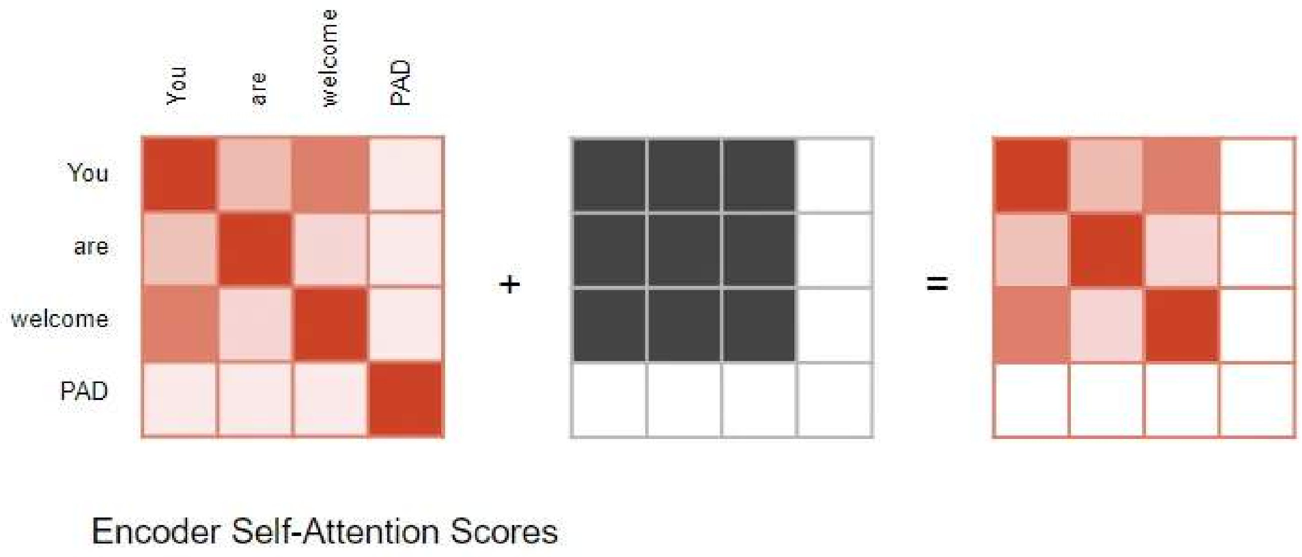
The important thing to realize here is that the Q, K, and V values carry an encoded representation of each word in the sequence. The Attention calculations then combine each word with every other word in the sequence, so that the Attention Score encodes a score for each word in the sequence.

When discussing the Decoder a little while back, we briefly mentioned masking. The Mask is also shown in the Attention diagrams above. Let's see how it works.

Attention Masks

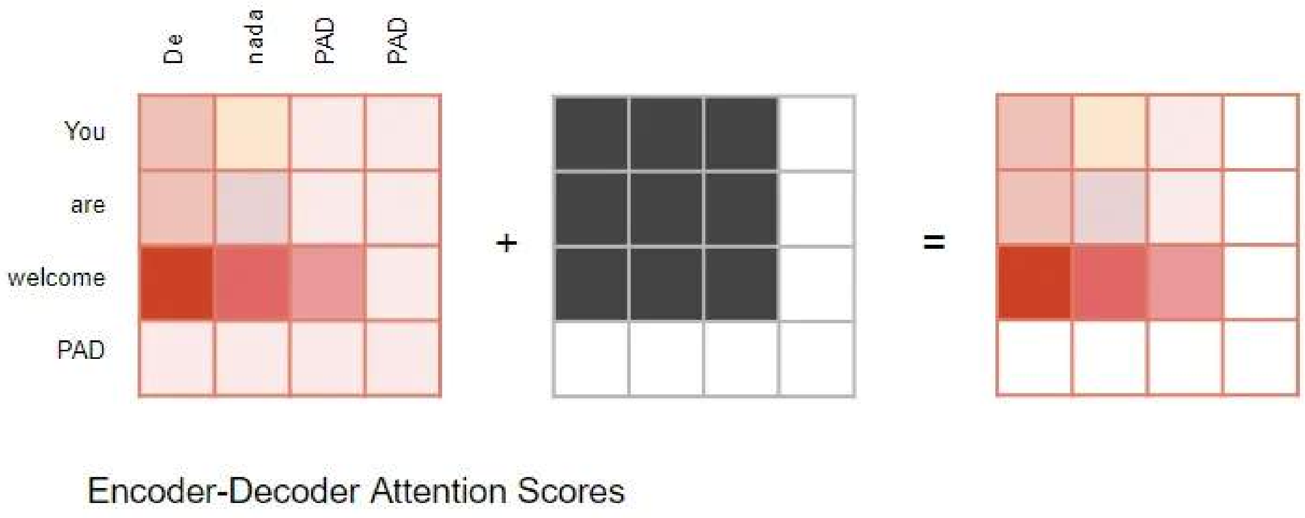
While computing the Attention Score, the Attention module implements a masking step. Masking serves two purposes:

In the Encoder Self-attention and in the Encoder-Decoder-attention: masking serves to zero attention outputs where there is padding in the input sentences, to ensure that padding doesn't contribute to the self-attention. (Note: since input sequences could be of different lengths they are extended with padding tokens like in most NLP applications so that fixed-length vectors can be input to the Transformer.)



(Image by Author)

Similarly for the Encoder-Decoder attention.

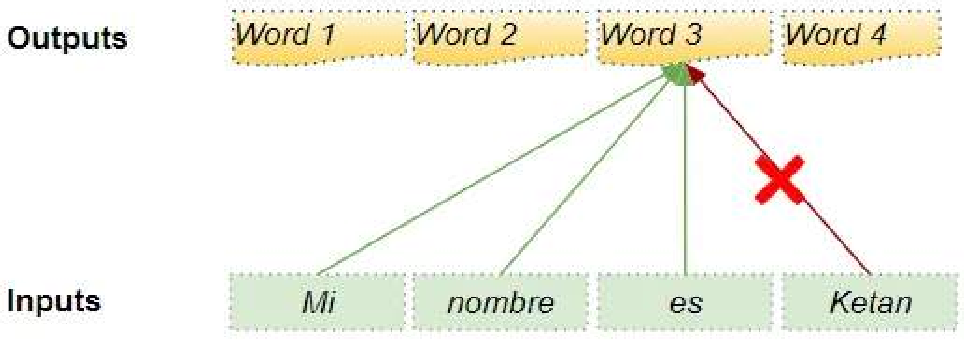


(Image by Author)

In the Decoder Self-attention: masking serves to prevent the decoder from 'peeking' ahead at the rest of the target sentence when predicting the next word.

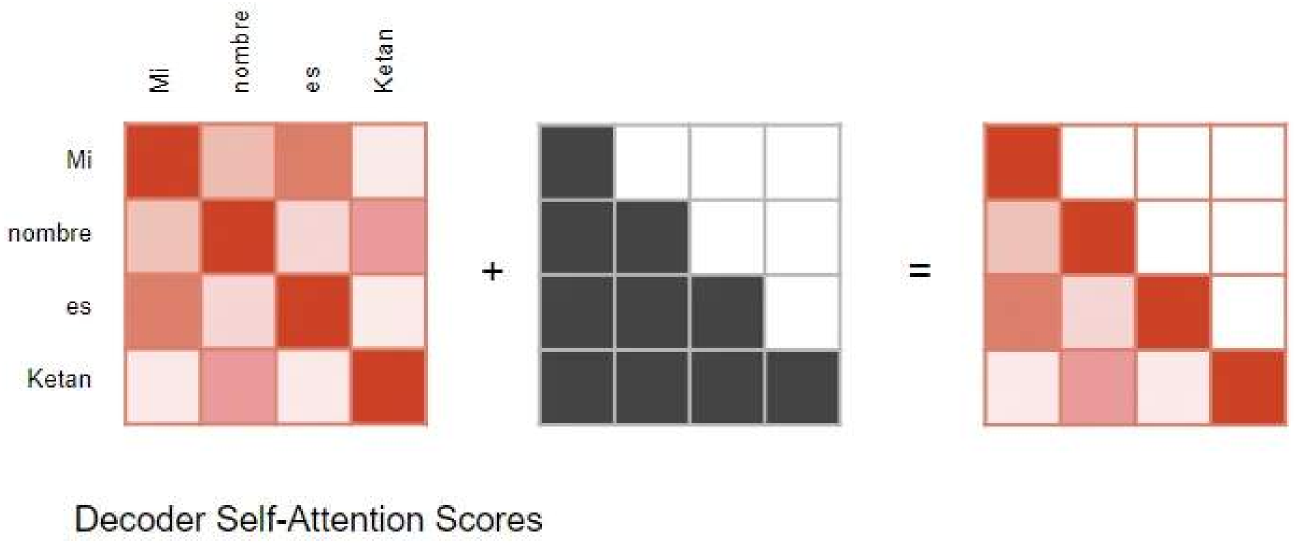
The Decoder processes words in the source sequence and uses them to predict the words in the destination sequence. During training, this is done via Teacher Forcing, where the complete target sequence is fed as Decoder inputs. Therefore, while predicting a word at a certain position, the Decoder has available to it the target words preceding that word as well as the target words following that word. This allows the Decoder to 'cheat' by using target words from future 'time steps'.

For instance, when predicting '*Word 3*', the Decoder should refer only to the first 3 input words from the target but not the fourth word '*Ketan*'.



(Image by Author)

Therefore, the Decoder masks out input words that appear later in the sequence.



(Image by Author)

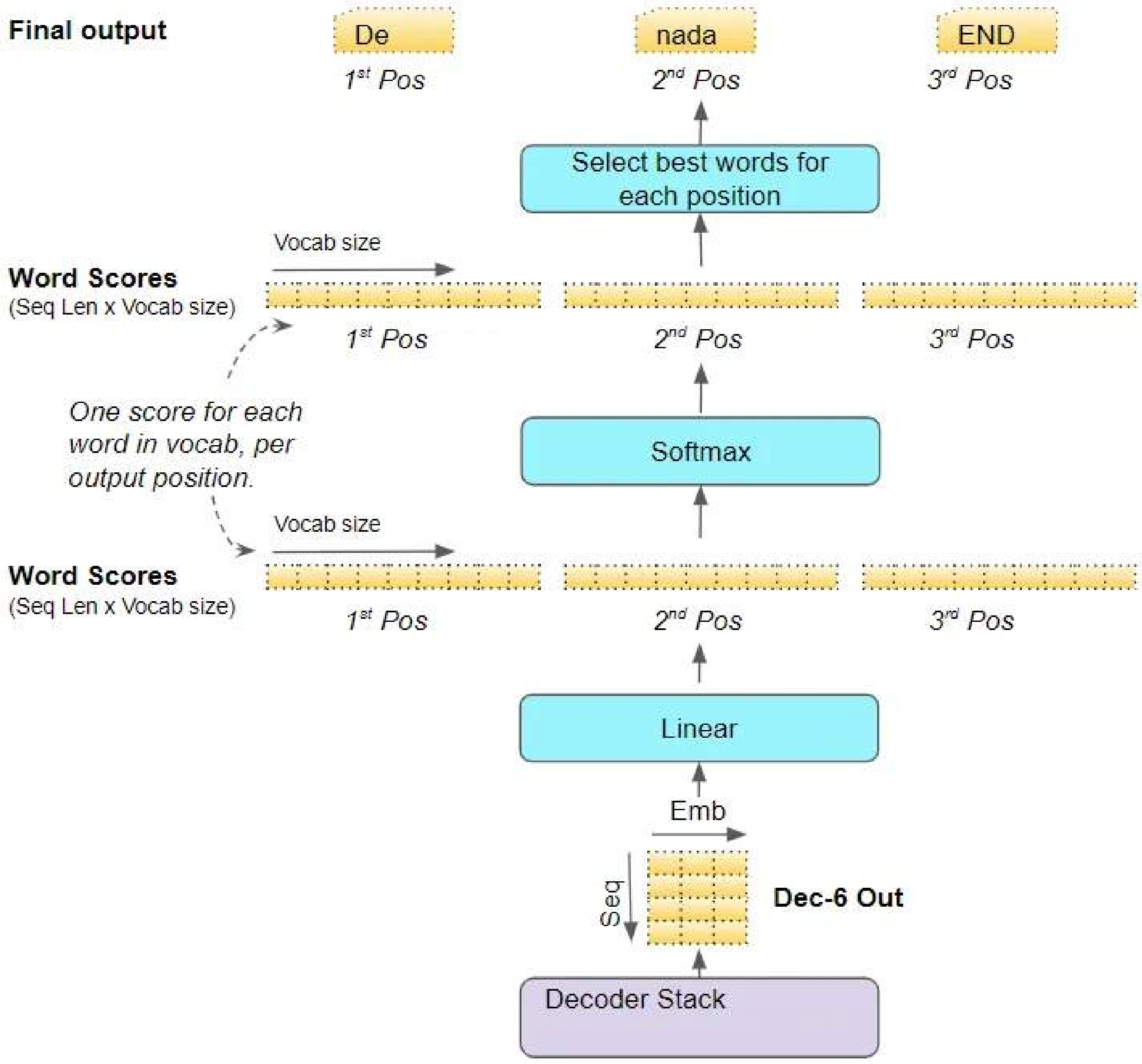
When calculating the Attention Score (refer to the picture earlier showing the calculations) masking is applied to the numerator just before the Softmax. The masked out elements (white squares) are set to negative infinity, so that Softmax turns those values to zero.

Generate Output

The last Decoder in the stack passes its output to the Output component which converts it into the final output sentence.

The Linear layer projects the Decoder vector into Word Scores, with a score value for each unique word in the target vocabulary, at each position in the sentence. For instance, if our final output sentence has 7 words and the target Spanish vocabulary has 10000 unique words, we generate 10000 score values for each of those 7 words. The score values indicate the likelihood of occurrence for each word in the vocabulary in that position of the sentence.

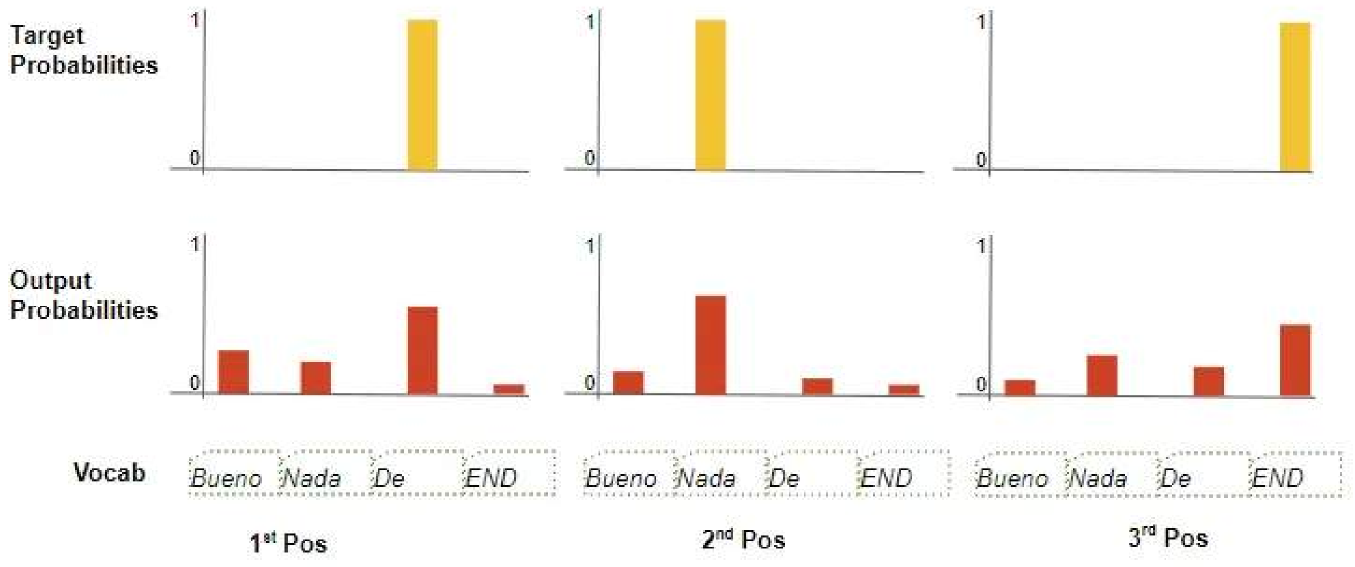
The Softmax layer then turns those scores into probabilities (which add up to 1.0). In each position, we find the index for the word with the highest probability, and then map that index to the corresponding word in the vocabulary. Those words then form the output sequence of the Transformer.



(Image by Author)

Training and Loss Function

During training, we use a loss function such as cross-entropy loss to compare the generated output probability distribution to the target sequence. The probability distribution gives the probability of each word occurring in that position.



(Image by Author)

Let's assume our target vocabulary contains just four words. Our goal is to produce a probability distribution that matches our expected target sequence "De nada END".

This means that the probability distribution for the first word-position should have a probability of 1 for "De" with probabilities for all other words in the vocabulary being 0. Similarly, "nada" and "END" should have a probability of 1 for the second and third word-positions respectively.

As usual, the loss is used to compute gradients to train the Transformer via backpropagation.

Conclusion

Hopefully, this gives you a feel for what goes on inside the Transformer during Training. As we discussed in the previous article, it runs in a loop during Inference but most of the processing remains the same.

The Multi-head Attention module is what gives the Transformer its power. In the next article, we will continue our journey and go one step deeper to really understand the details of how Attention is computed.

And finally, if you liked this article, you might also enjoy my other series on Audio Deep Learning, Geolocation Machine Learning, and Image Caption architectures.

Audio Deep Learning Made Simple (Part 1): State-of-the-Art Techniques

11.1. Queries, Keys, and Values

COLAB [PYTORCH]

esearch.google.com/github/d2l-

ster/chapter_attention-
nd-transformers/queries-keys-

SAGEMAKER STUDIO LAB

(https://studiolab.sagemaker.aw
ai/d2l-pytorch-sagemaker-
studio-
lab/blob/main/GettingStarted-
D2L.ipynb)

So far all the networks we have reviewed crucially relied on the input being of a well-defined size. For instance, the images in ImageNet are of size 224×224 pixels and CNNs are specifically tuned to this size. Even in natural language processing the input size for RNNs is well defined and fixed. Variable size is addressed by sequentially processing one token at a time, or by specially designed convolution kernels (Kalchbrenner *et al.*, 2014 (../chapter_references/zreferences.html#id141)). This approach can lead to significant problems when the input is truly of varying size with varying information content, such as in Section 10.7 (../chapter_recurrent-modern/seq2seq.html#sec-seq2seq) in the transformation of text (Sutskever *et al.*, 2014 (../chapter_references/zreferences.html#id273)). In particular, for long sequences it becomes quite difficult to keep track of everything that has already been generated or even viewed by the network. Even explicit tracking heuristics such as proposed by Yang *et al.* (2016 (../chapter_references/zreferences.html#id272)) only offer limited benefit.

Compare this to databases. In their simplest form they are collections of keys (k) and values (v). For instance, our database $\mathcal{D}$ might consist of tuples {("Zhang", "Aston"), ("Lipton", "Zachary"), ("Li", "Mu"), ("Smola", "Alex"), ("Hu", "Rachel"), ("Werness", "Brent")} with the last name being the key and the first name being the value. We can operate on $\mathcal{D}$, for instance with the exact query (q) for "Li" which would return the value "Mu". If ("Li", "Mu") was not a record in $\mathcal{D}$, there would be no valid answer. If we also allowed for approximate matches, we would retrieve ("Lipton", "Zachary") instead. This quite simple and trivial example nonetheless teaches us a number of useful things:

- We can design queries q that operate on (k,v) pairs in such a manner as to be valid regardless of the database size.
- The same query can receive different answers, according to the contents of the database.
- The "code" being executed for operating on a large state space (the database) can be quite simple (e.g., exact match, approximate match, top-k).
- There is no need to compress or simplify the database to make the operations effective.

Clearly we would not have introduced a simple database here if it wasn't for the purpose of explaining deep learning. Indeed, this leads to one of the most exciting concepts introduced in deep learning in the past decade: the *attention mechanism* (Bahdanau *et al.*, 2014 (../chapter_references/zreferences.html#id10)). We will cover the specifics of its application to machine translation later. For now, simply consider the following: denote by $\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \ldots (\mathbf{k}_m, \mathbf{v}_m)\}$ a database of m tuples of *keys* and *values*. Moreover, denote by $\mathbf{q}$ a *query*. Then we can define the *attention* over $\mathcal{D}$ as

$$\text{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i, \tag{11.1.1}$$

where $\alpha(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}$ ($i = 1, \ldots, m$) are scalar attention weights. The operation itself is typically referred to as *attention pooling*. The name *attention* derives from the fact that the operation pays particular attention to the terms for which the weight α is significant (i.e., large). As such, the attention over $\mathcal{D}$ generates a linear combination of values contained in the database. In fact, this contains the above example as a special case where all but one weight is zero. We have a number of special cases:

- The weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ are nonnegative. In this case the output of the attention mechanism is contained in the convex cone spanned by the values $\mathbf{v}_i$.
- The weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ form a convex combination, i.e., $\sum_i \alpha(\mathbf{q}, \mathbf{k}_i) = 1$ and $\alpha(\mathbf{q}, \mathbf{k}_i) \geq 0$ for all i. This is the most common setting in deep learning.
- Exactly one of the weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ is 1, while all others are 0. This is akin to a traditional database query.
- All weights are equal, i.e., $\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{1}{m}$ for all i. This amounts to averaging across the entire database, also called average pooling in deep learning.

A common strategy for ensuring that the weights sum up to 1 is to normalize them via

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\alpha(\mathbf{q}, \mathbf{k}_i)}{\sum_j \alpha(\mathbf{q}, \mathbf{k}_j)}. \tag{11.1.2}$$

In particular, to ensure that the weights are also nonnegative, one can resort to exponentiation. This means that we can now pick *any* function $a(\mathbf{q}, \mathbf{k})$ and then apply the softmax operation used for multinomial models to it via

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(a(\mathbf{q}, \mathbf{k}_j))}. \tag{11.1.3}$$

This operation is readily available in all deep learning frameworks. It is differentiable and its gradient never vanishes, all of which are desirable properties in a model. Note though, the attention mechanism introduced above is not the only option. For instance, we can design a non-differentiable attention model that can be trained using reinforcement learning methods (Mnih *et al.*, 2014 (../chapter_references/zreferences.html#id195)). As one would expect, training such a model is quite complex. Consequently the bulk of modern attention research follows the framework outlined in Fig. 11.1.1. We thus focus our exposition on this family of differentiable mechanisms.

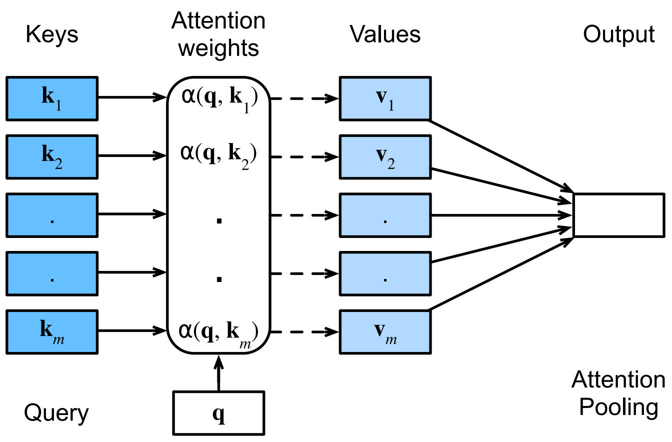


Fig. 11.1.1 The attention mechanism computes a linear combination over values $\mathbf{v}_i$ via attention pooling, where weights are derived according to the compatibility between a query $\mathbf{q}$ and keys $\mathbf{k}_i$.

What is quite remarkable is that the actual "code" for executing on the set of keys and values, namely the query, can be quite concise, even though the space to operate on is significant. This is a desirable property for a network layer as it does not require too many parameters to learn. Just as convenient is the fact that attention can operate on arbitrarily large databases without the need to change the way the attention pooling operation is performed.

PYTORCH MXNET JAX TENSORFLOW

```
import torch
from d2l import torch as d2l
```

```
from mxnet import np, npx
from d2l import mxnet as d2l

npx.set_np()
```

INTUITIVE TRANSFORMERS SERIES NLP

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.

Ketan Doshi · Follow

Published in Towards Data Science

11 min read · Jan 17, 2021

Listen Share

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)*

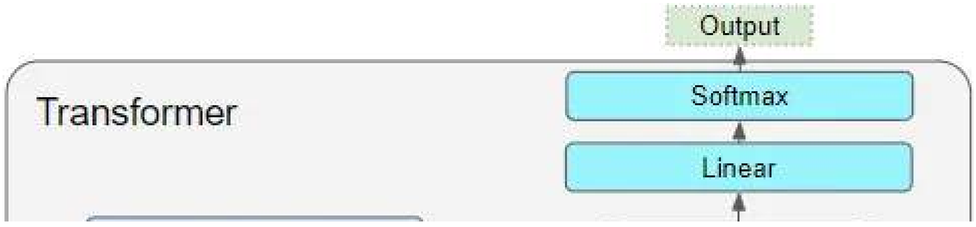
And if you're interested in NLP applications in general, I have some other articles you might like.

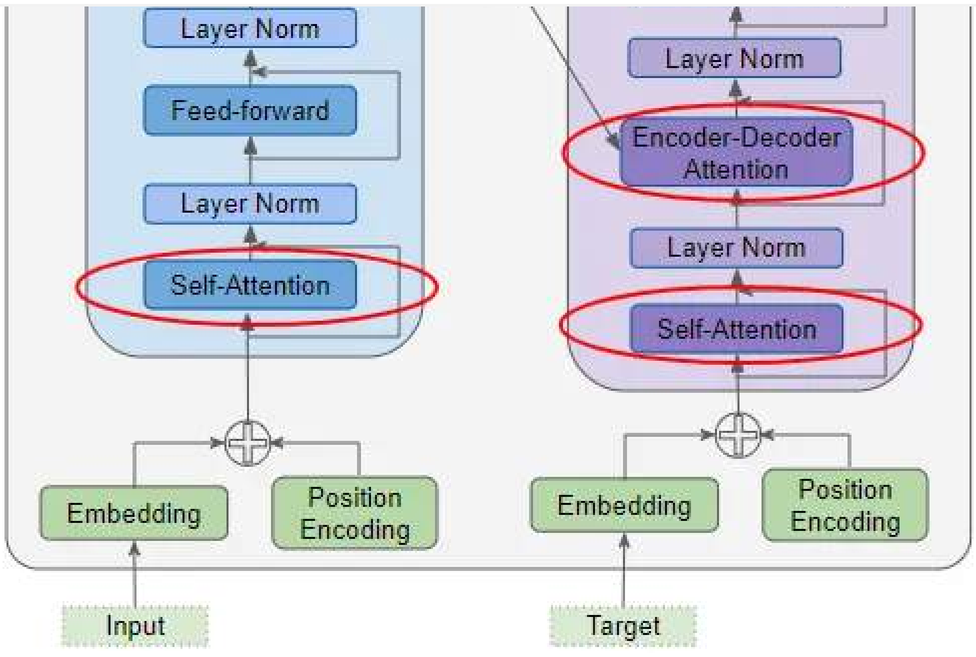
1. Beam Search *(Algorithm commonly used by Speech-to-Text and NLP applications to enhance predictions)*
2. Bleu Score (*Bleu Score and Word Error Rate are two essential metrics for NLP models*)

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)

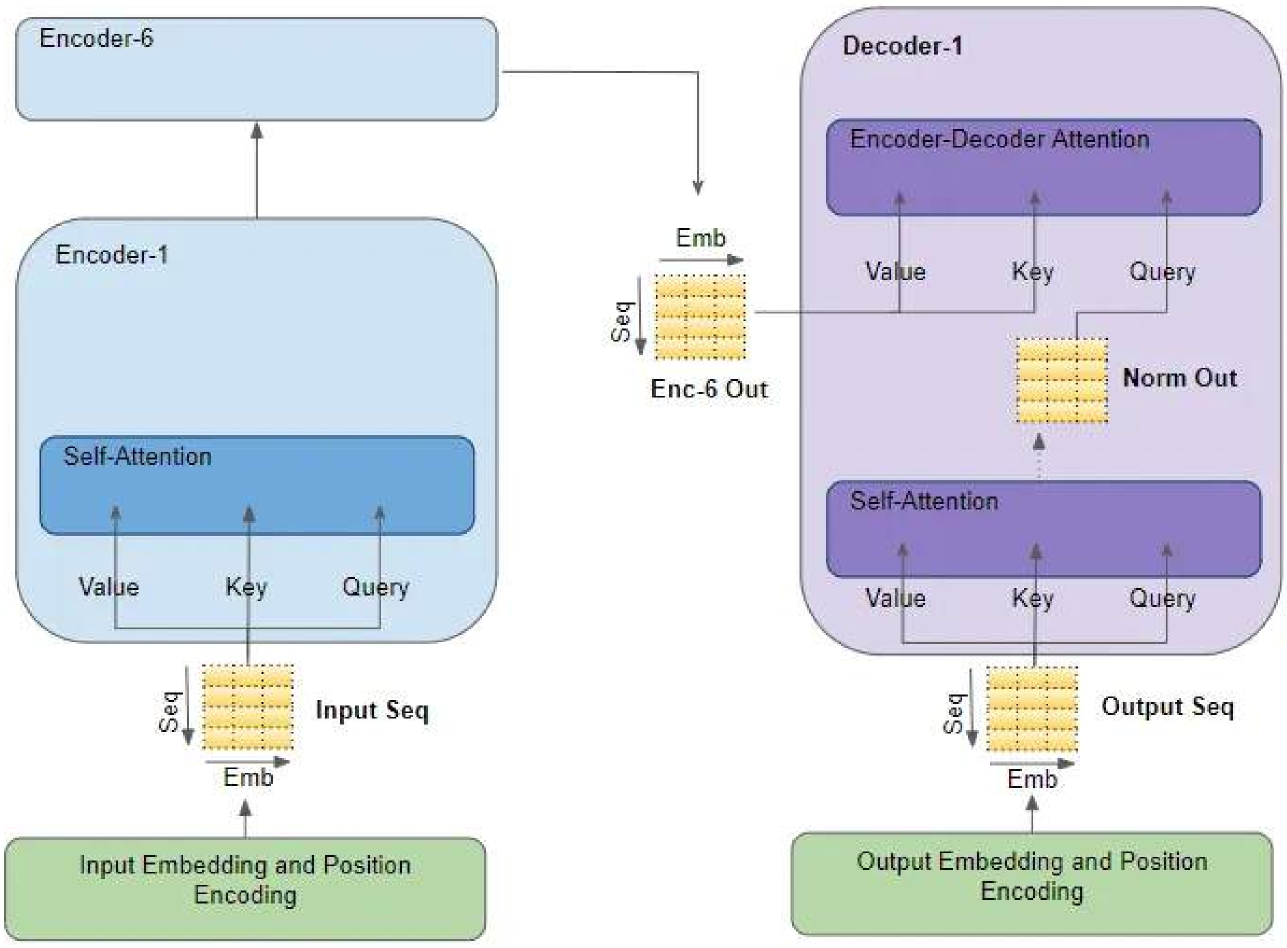
Attention Input Parameters — Query, Key, and Value

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

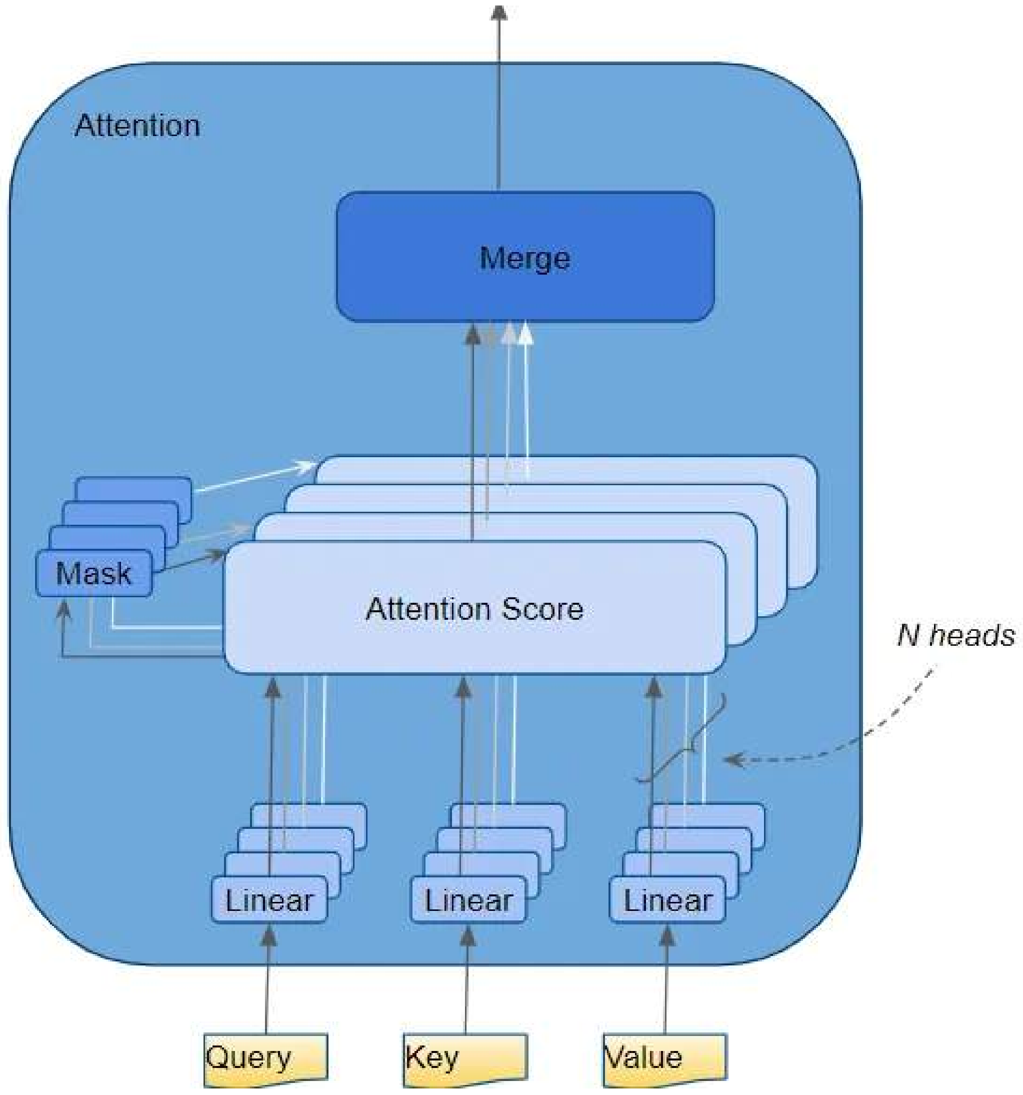
Along with that, the output of the final Encoder in the stack is passed to the Value and Key parameters in the Encoder-Decoder Attention.

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:

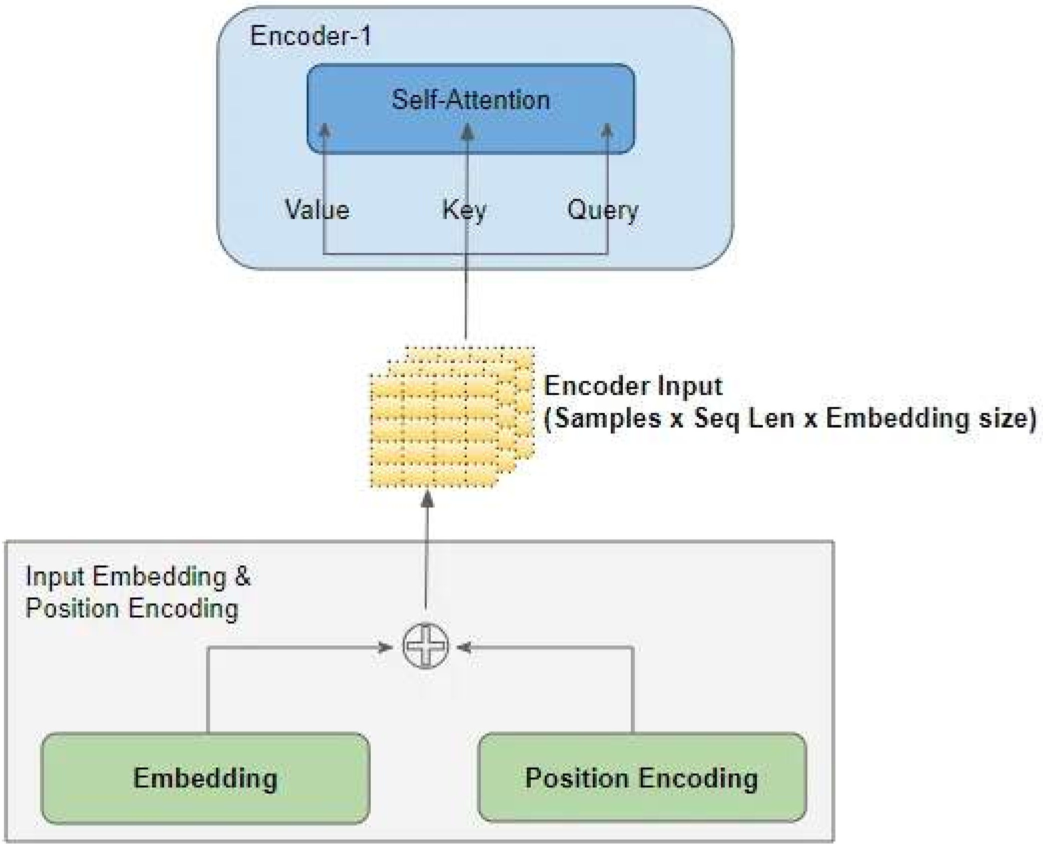
- Embedding Size — width of the embedding vector (we use a width of 6 in our example). This dimension is carried forward throughout the Transformer model and hence is sometimes referred to by other names like 'model size' etc.
- Query Size (equal to Key and Value size)— the size of the weights used by 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.

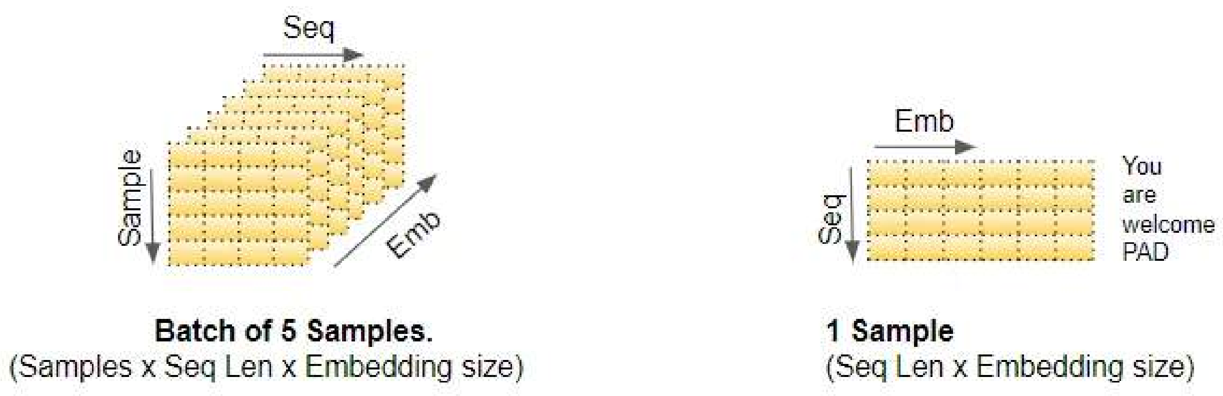
Input Layers

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)

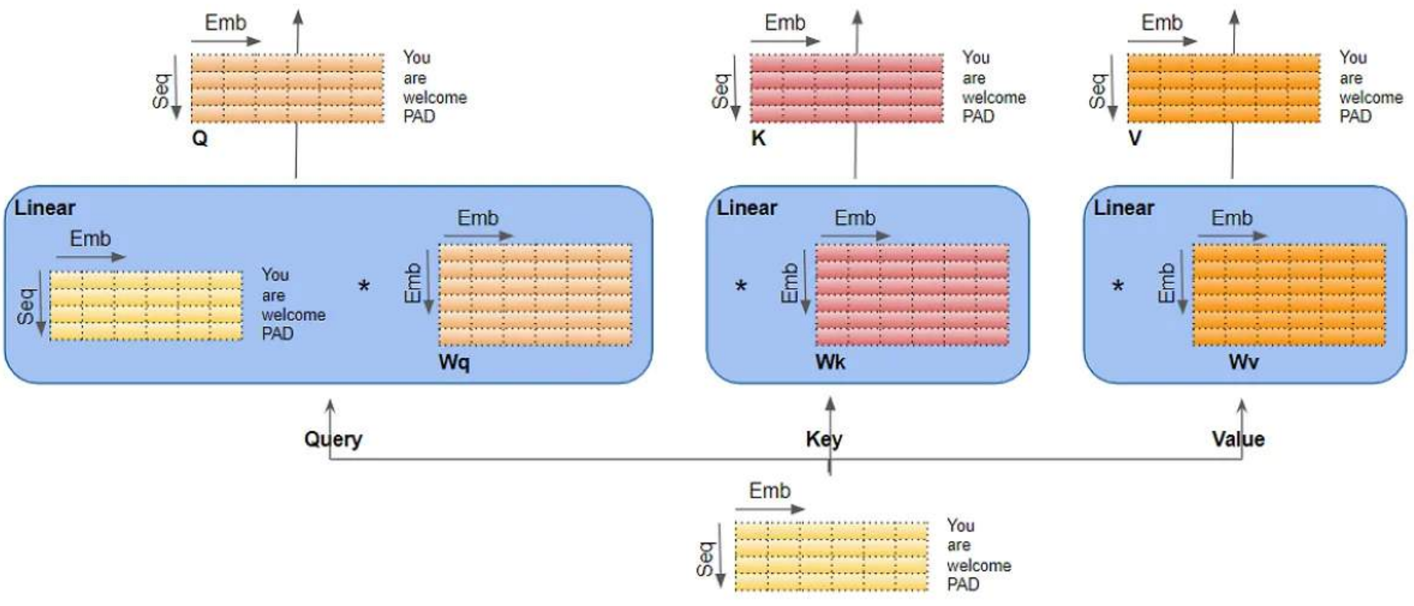
To make it simple to visualize, we will drop the Batch dimension in our pictures and focus on the remaining dimensions.


Seq
Sample
Emb
Batch of 5 Samples.
(Samples x Seq Len x Embedding size)
Emb
Seq
You
are
welcome
PAD
1 Sample
(Seq Len x Embedding size)

(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.

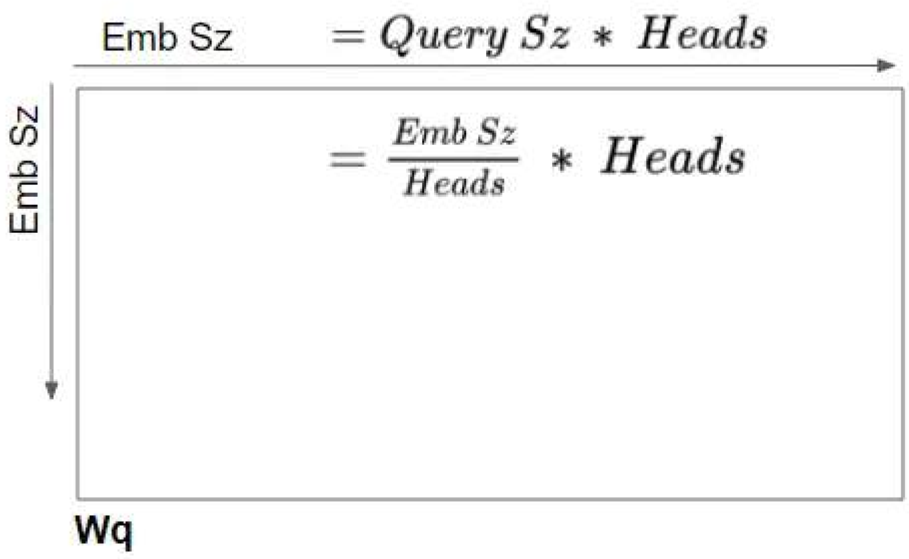
However, the important thing to understand is that this is a logical split only. The Query, Key, and Value are not physically split into separate matrices, one for each Attention head. A single data matrix is used for the Query, Key, and Value, respectively, with logically separate sections of the matrix 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:

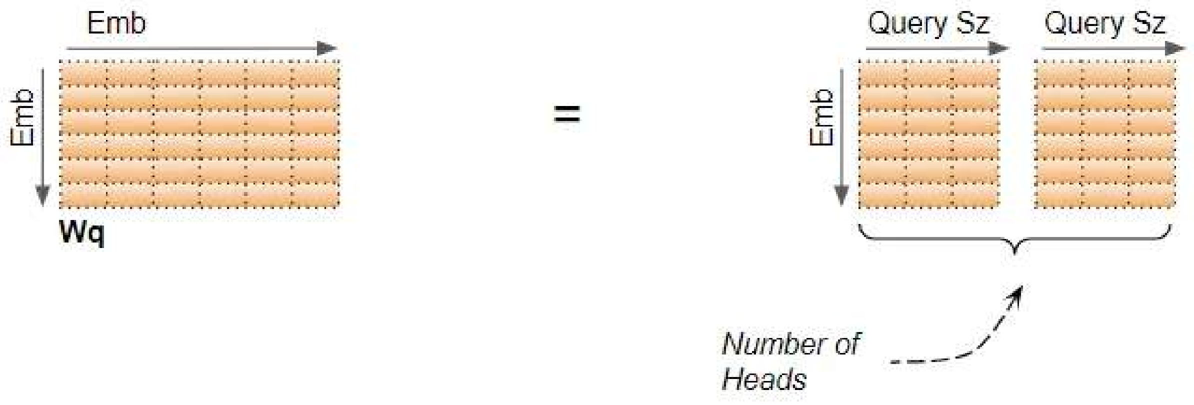
Query Size = Embedding Size / Number of heads



Emb Sz $= Query\ Sz * Heads$
$= \frac{Emb\ Sz}{Heads} * Heads$
Emb Sz
Wq

(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.



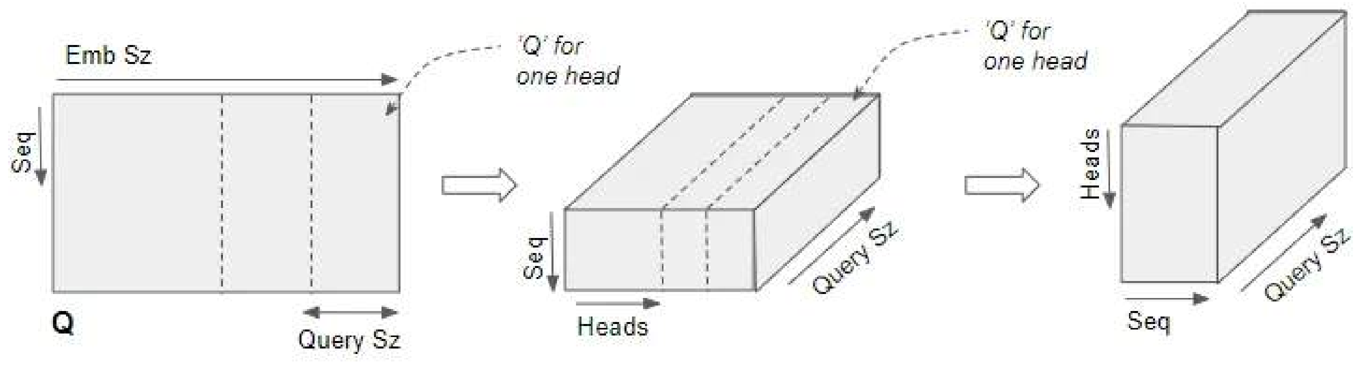
(Image by Author)

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.

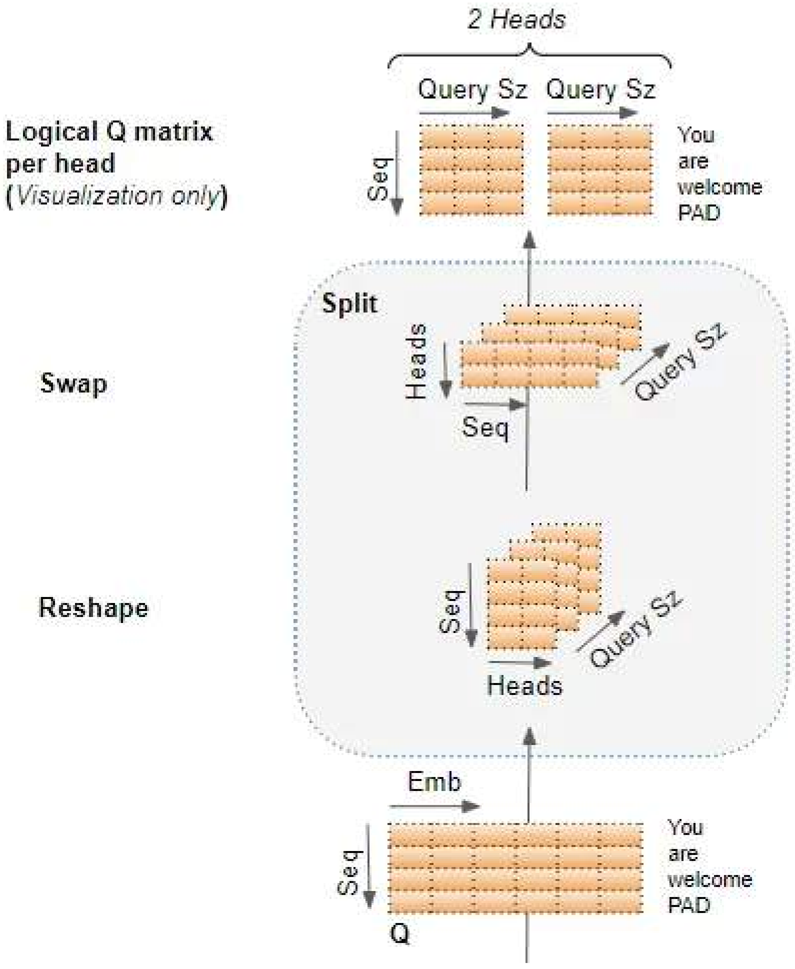
This matrix is reshaped again by swapping the Head and Sequence dimensions. Although the Batch dimension is not drawn, the dimensions of Q are now (Batch, Head, Sequence, Query size).



The Q matrix is reshaped to include a Head dimension and then reshaped again by swapping the Head and Sequencd 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)

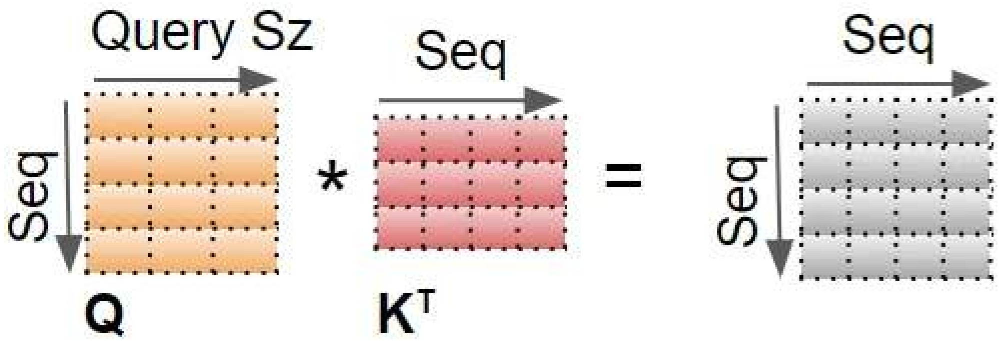
We are ready to compute the Attention Score.

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).

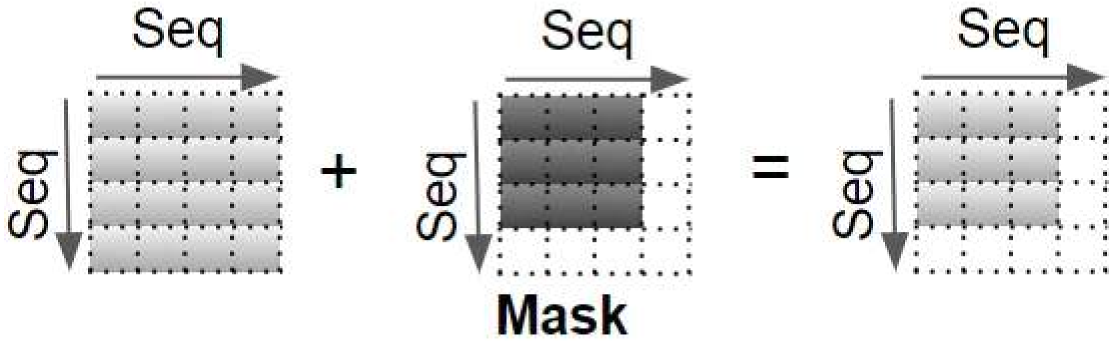
The first step is to do a matrix multiplication between Q and K.



(Image by Author)

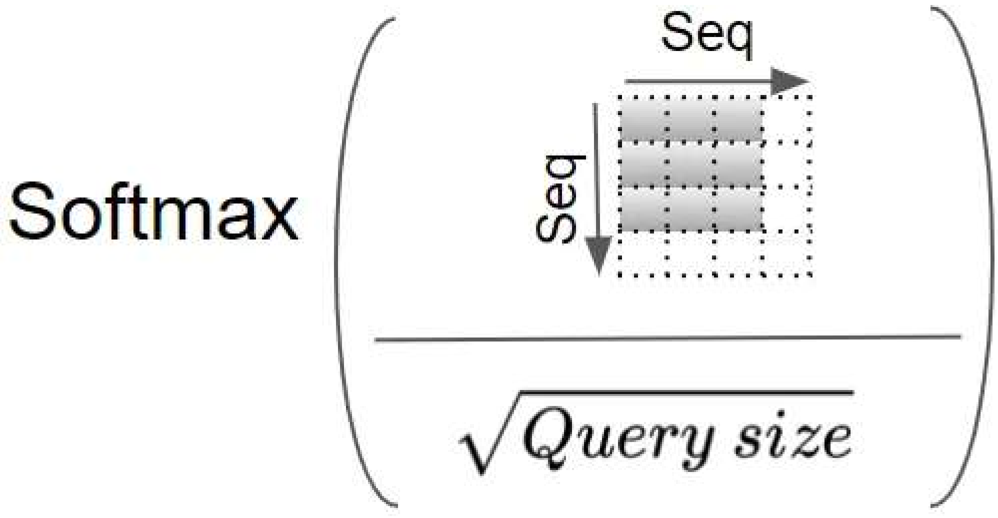
A Mask value is now added to the result. In the Encoder Self-attention, the mask is used to mask out the Padding values so that they don't participate in the Attention Score.

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.



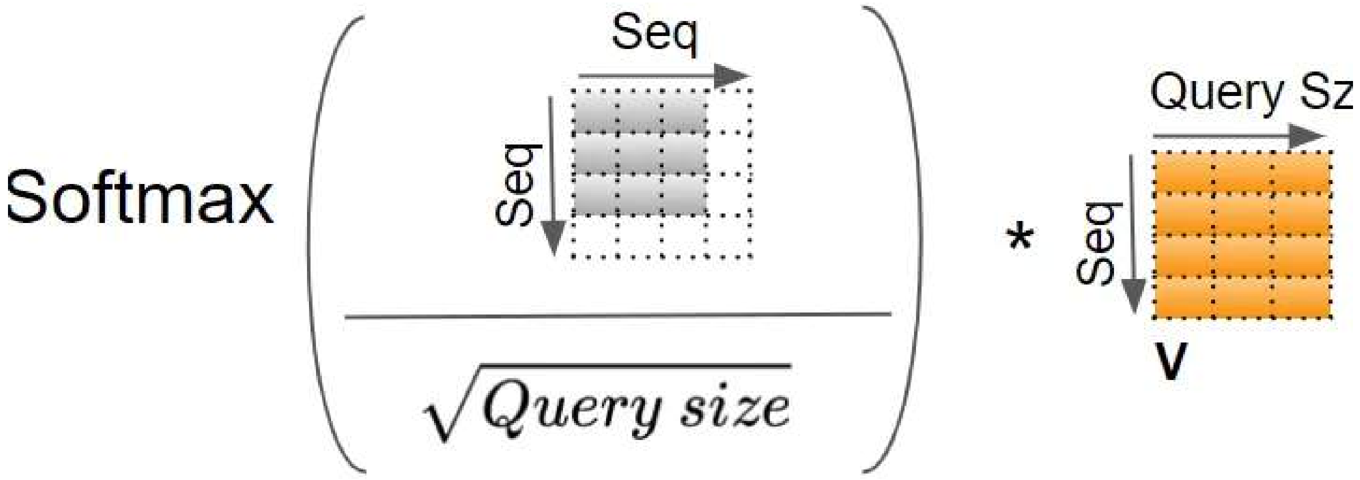
(Image by Author)

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



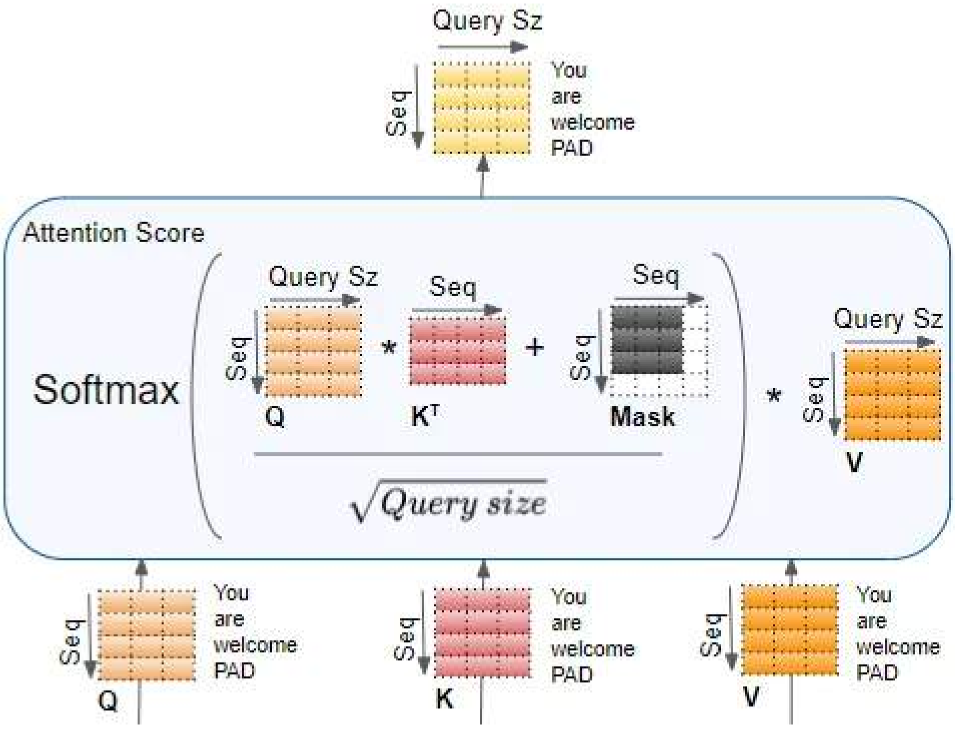
(Image by Author)

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



(Image by Author)

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



(Image by Author)

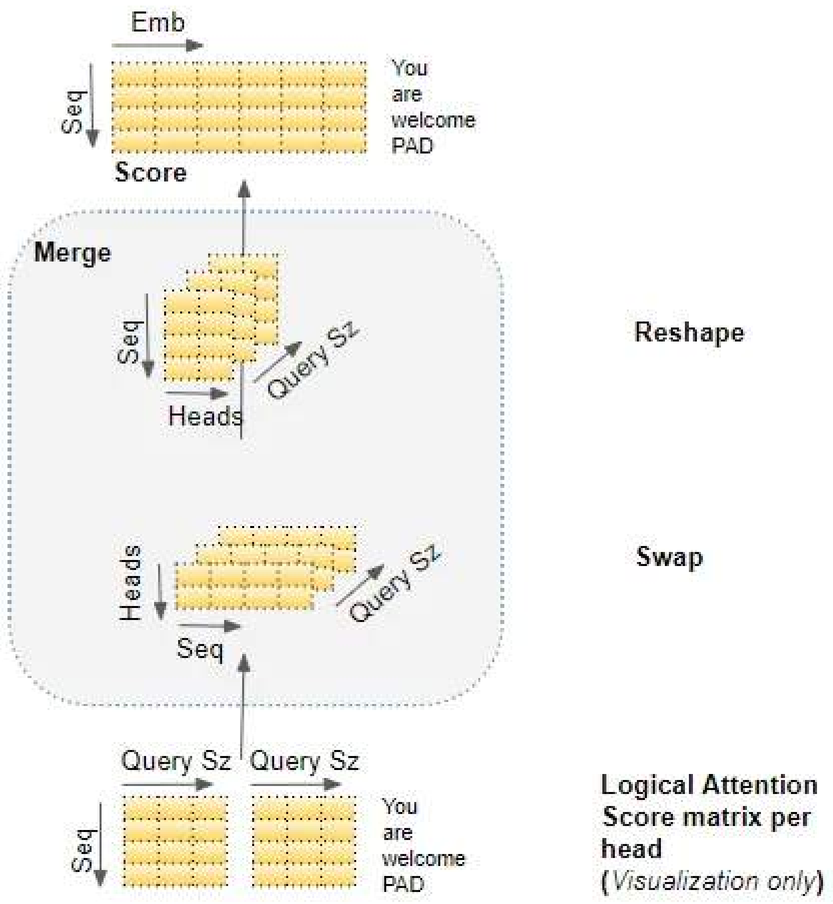
Merge each Head's Attention Scores together

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.

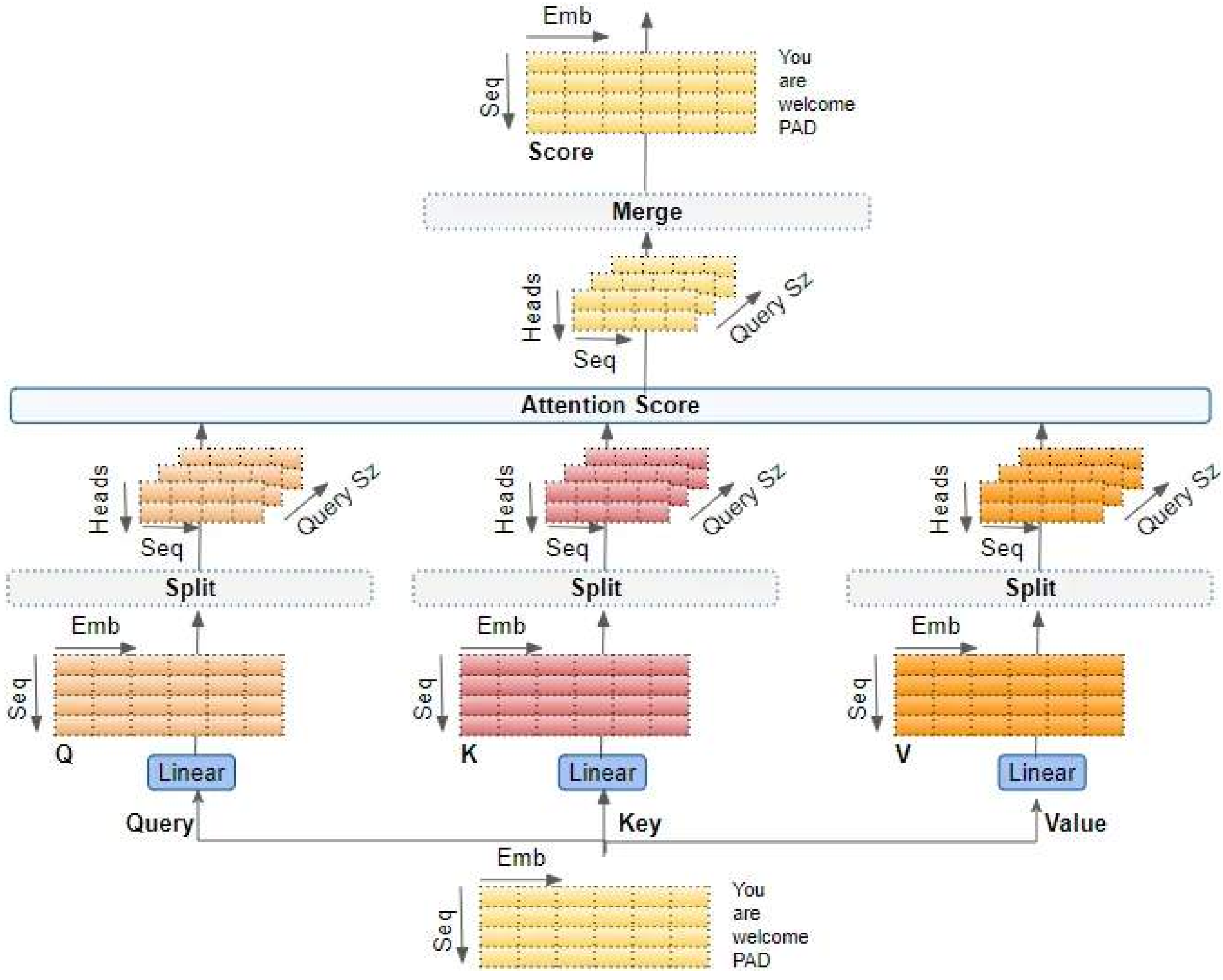
Since Embedding size =Head * Query size, the merged Score is (Batch, Sequence, Embedding size). In the picture below, we can see the complete process of merging for the example Score matrix.



(Image by Author)

End-to-end Multi-head Attention

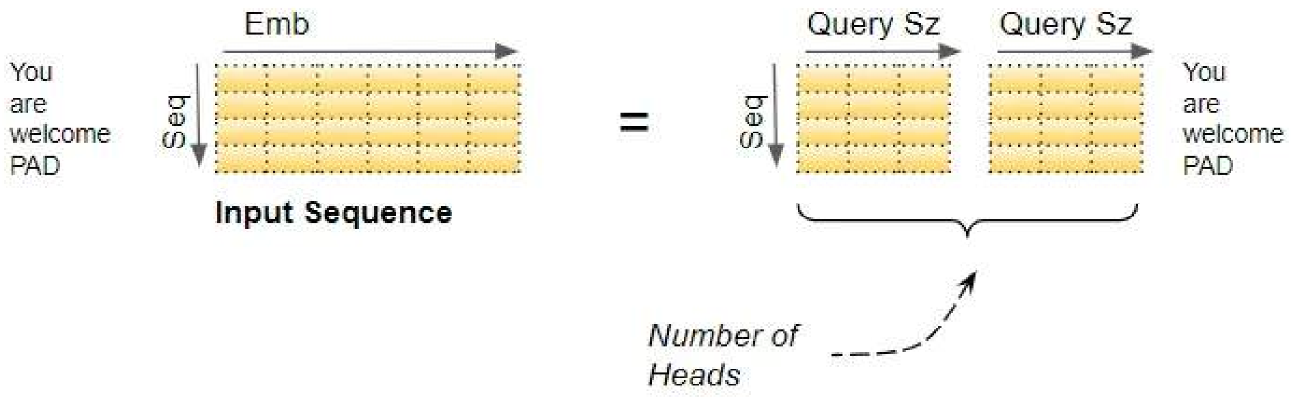
Putting it all together, this is the end-to-end flow of the Multi-head Attention.



(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?



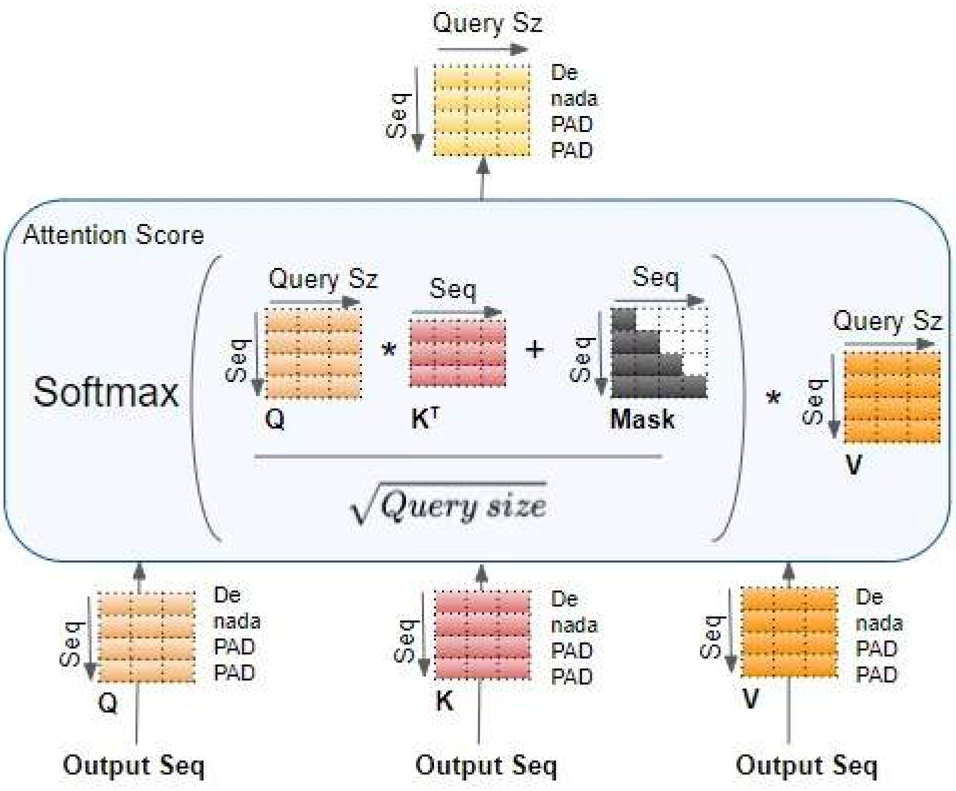
(Image by Author)

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.

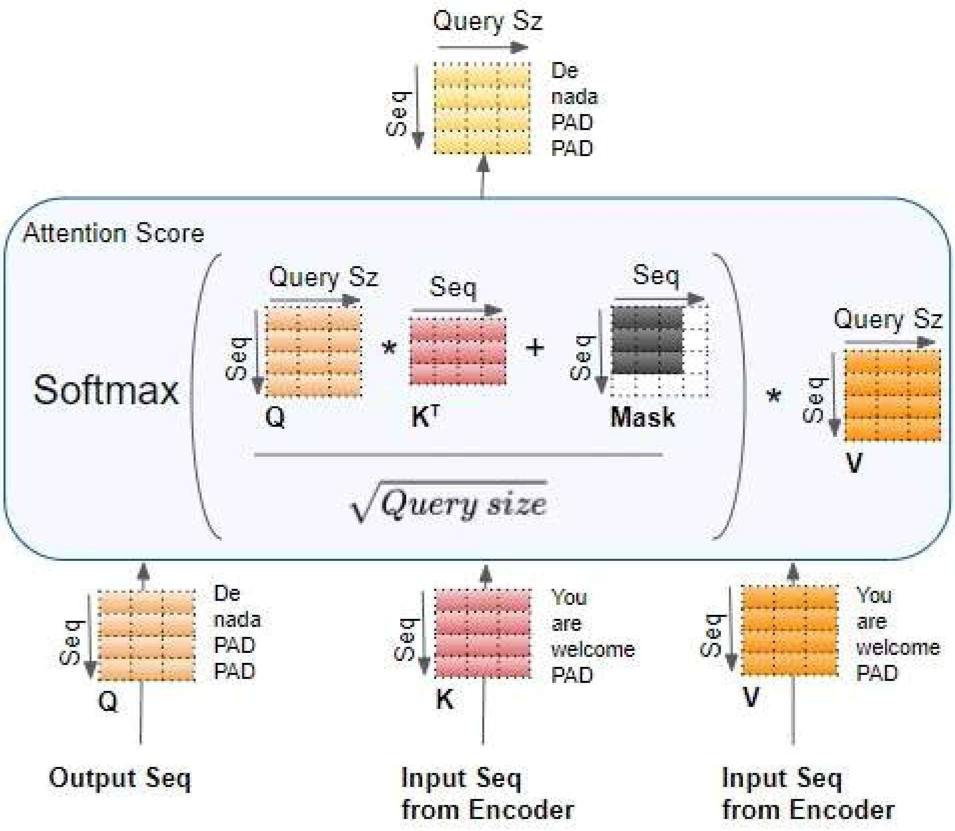


(Image by Author)

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.

Conclusion

Hopefully, this gives you a good sense of what the Attention modules in the Transformer do. When put together with the end-to-end flow of the Transformer as a whole that we went over in the second article, we have now covered the detailed operation of the entire Transformer architecture.

We now understand exactly *what* the Transformer does. But we haven't fully answered the question of *why* the Transformer's Attention performs the calculations that it does. Why does it use the notions of Query, Key, and Value, and why does it perform the matrix multiplications that we just saw?

We have a vague intuitive idea that it 'captures the relationship between each word with each other word', but what exactly does that mean? How exactly does that give the Transformer's Attention the capability to understand the nuances of each word in the sequence?

That is an interesting question and is the subject of the final article of this series. Once we learn that, we will truly understand the elegance of the Transformer architecture.

And finally, if you liked this article, you might also enjoy my other series on Audio Deep Learning, Geolocation Machine Learning, and Image Caption architectures.

Audio Deep Learning Made Simple (Part 1): State-of-the-Art Techniques

A Gentle Guide to the world of disruptive deep learning audio applications and architectures. And why we all need to...

towardsdatascience.com

Leveraging Geolocation Data for Machine Learning: Essential Techniques

A Gentle Guide to Feature Engineering and Visualization with Geospatial data, in Plain English

towardsdatascience.com

Image Captions with Deep Learning: State-of-the-Art Architectures

A Gentle Guide to Image Feature Encoders, Sequence Decoders, Attention, and Multi-modal Architectures, in Plain English

towardsdatascience.com

Let's keep learning!

Deep Learning · Data Science · Machine Learning · Artificial Intelligence · NLP