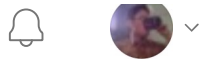


Published in Analytics Vidhya

Alvaro Henriquez Follow

Nov 26, 2020 · 9 min read · Listen

Save

Understanding Attention In Transformers Models

What I wish I knew before my project.

I thought that it would be cool to build a language translator. At first, I thought that I would do so utilizing a `recurrent neural network` (`RNN`), or an `LSTM`. But as I did my research I started to come across articles that mention the `transformer model` as the new state of the art in NLP.

So my interest was peaked, and I thought that it might be cool to build a transformer for my project.

So now I had to get as much information as I could get my hands on to try to learn how to implement one. But more than that, I really wanted to understand how they

work.

I found the research paper that started it all - ***Attention Is All You Need***. it was published in 2017, so it is fairly recent. But as I read it I knew that I would need some help in understanding some of the concepts.

The pieces where I became stuck were the positional encoding and self-attention. So I continued searching, trying to find how other people were explaining these concepts.

The best blog post that I was able to find is Jay Alammar's ***The Illustrated Transformer***. If you are a visual learner like myself you'll find this one invaluable. There I found a link to another post by Jay, ***Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)***.

In this post, I will be tackling the concept of `self-attention`. This will hopefully serve as a resource to anyone else seeking to understand this concept when researching transformer models.

I will not go into transformers in this post, just attention. But here is a picture of the architecture so that you get an idea of where the attention sub-layers are in relation to the model.

Transformer Model Architecture

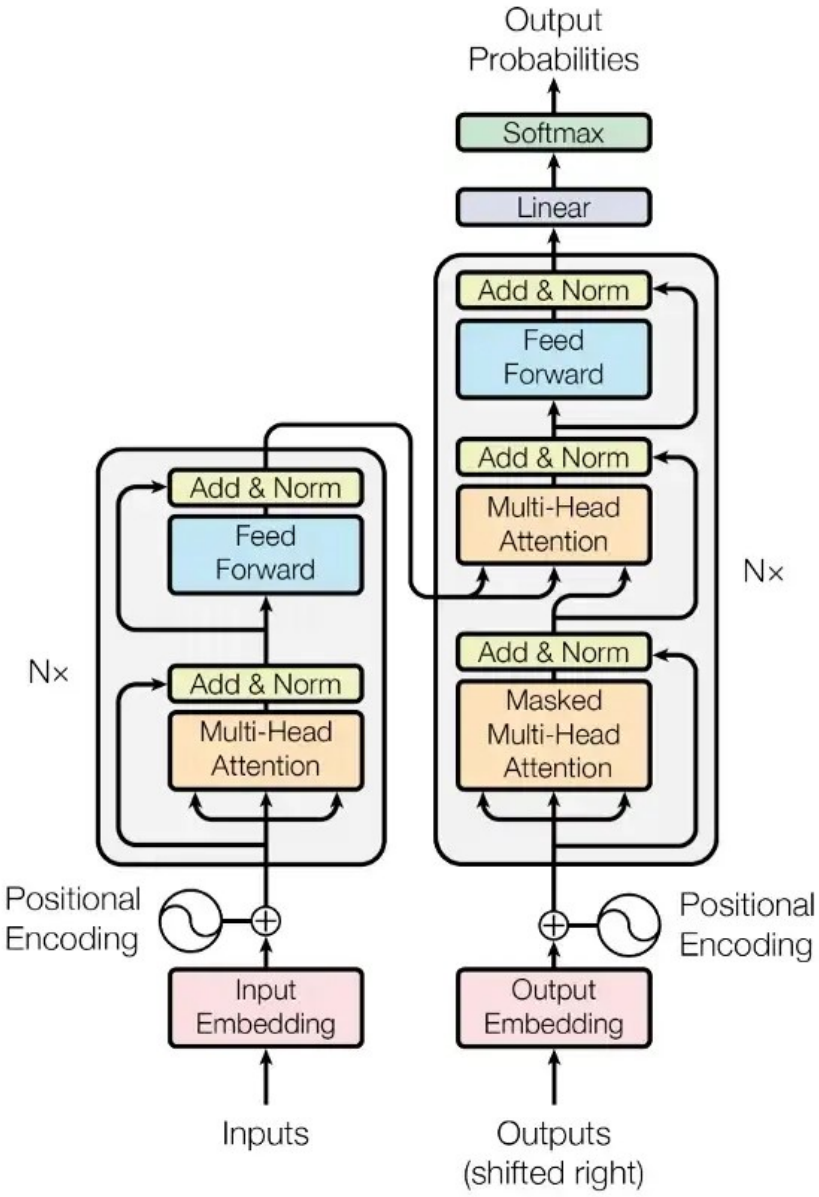


Figure 1: The Transformer - model architecture.

source: Attention Is All You Need

So, why attention?

This is the special sauce in the transformer! It solves the problems of parallelization and loss of contextual information for words that are distant from each other within a sequence. These were issues with `seq2seq` models such as `RNNs` and `LSTMs`.

The main cause of these issues within those models is that they needed to process each word in a sequence one by one. This meant that they took longer to train. What if we could process all of the words in parallel?

The other issue is the way that context and structure are derived. Those models start by inputting the first word (token) to an encoder and calculating a new `state` with a

random starting state. Then in the next time step, use the next word and the previous state to create a new state. This process is repeated until all words are exhausted at which point a fixed-length context vector is generated. The context vector is then passed to the decoder for output generation.

The problem is that by the time the final context vector is computed, contextual information about the earlier states can be lost because the earlier computations do not have access to the later words. In addition, the fixed-sized context vector may not be large enough to retain all of the information.

So how does the transformer solve these problems?

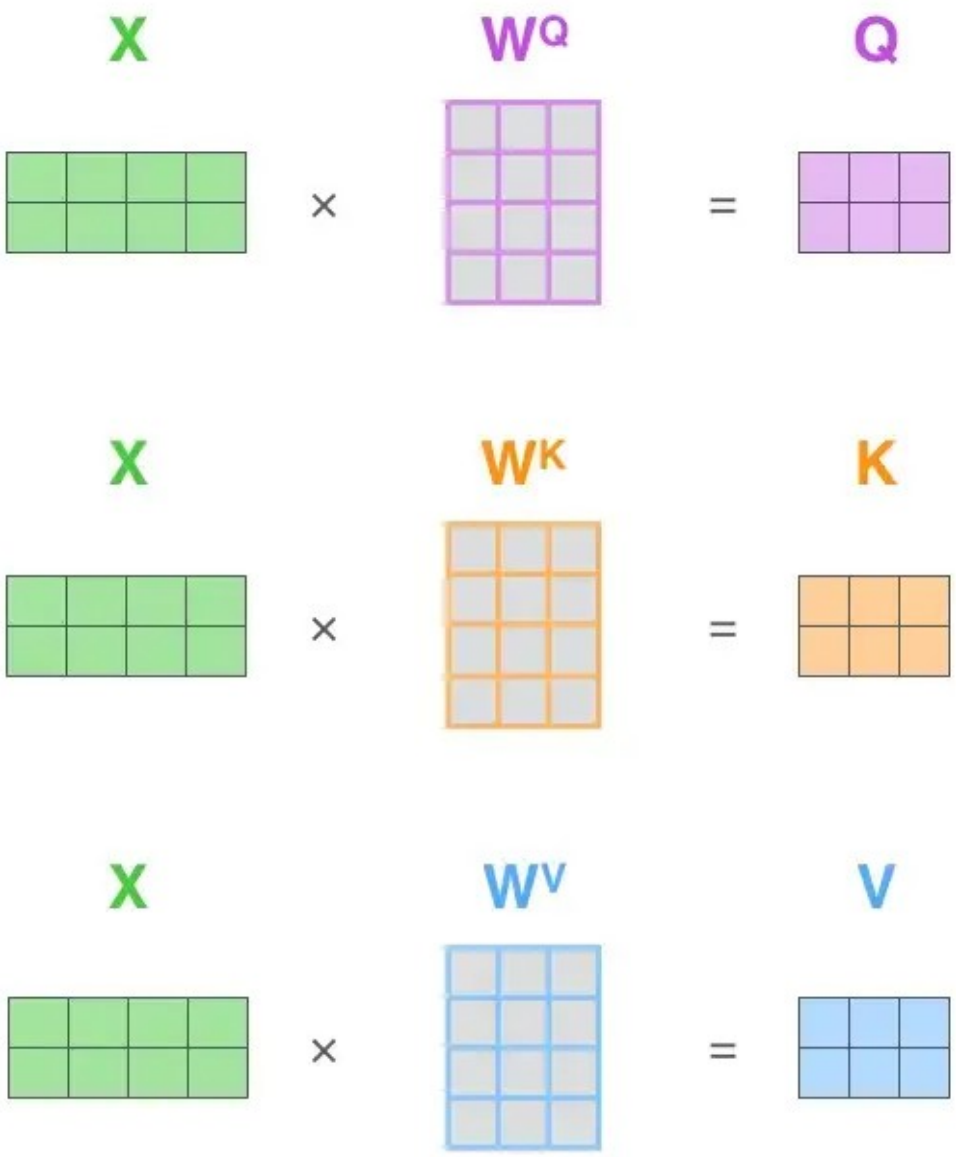
The transformer's encoders and decoders consume the entire sequence and process all words (embedding) in parallel. Because of this parallelization, training time is reduced. Second, this parallel processing means that the context can be computed from all the words together, resulting in a more complete context.

Linear Transformations

The first thing that happens when a sequence of embeddings is passed to the transformer's inputs (encoder and decoder) is that each embedding experiences three separate linear transformations resulting in three vectors — `query`, `key`, and `value`. These transformations occur when the input vectors (embedding) are multiplied by 3 weight matrices. The proper weights are learned through training. The image below represents a vector of `sequence length` of 2 and `embedding size` of 4.

Why three linear transformations?

Since each `weight matrix` is initialized with random weights, the resultant vectors each learn some different information about the embedding (word) being processed. This is important when calculating the attention score since we don't want to just derive the dot product of the vector with itself.



source: The Illustrated Transformer

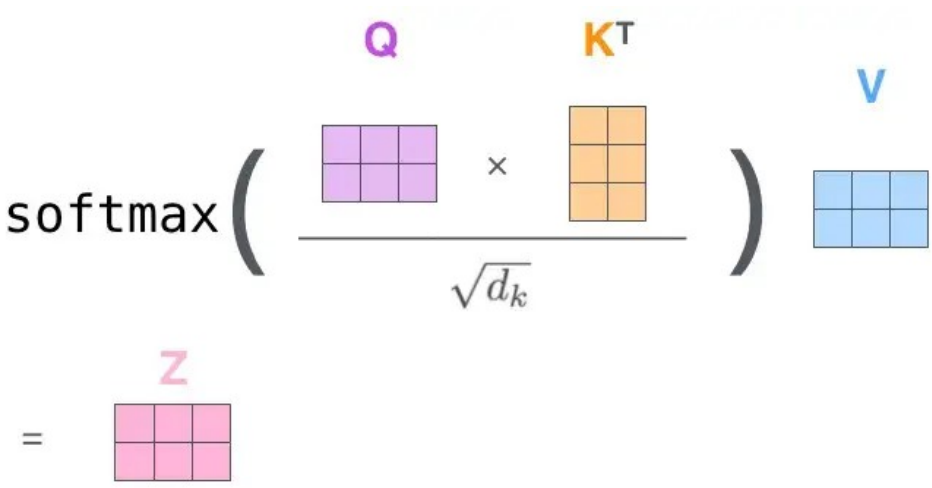
Now that we have these 3 vectors for each embedding in the sequence we can calculate the **`attention score`**. The attention score measures the strength of the relationship between a word in the sequence with all the other words.

Steps to calculating Attention

1. Take the query vector for a word and calculate it's **`dot product`** with the transpose of the **`key vector`** of each word in the sequence — including itself. This is the **`attention score`** or **`attention weight`**.
2. Then divide each of the results by the square root of the dimension of the key vector. This is the **`scaled attention score`**.
3. Pass them through a **`softmax`** function, so that values are contained between 0 and

1.
4. Take each of the `value vectors` and calculate the `dot product` with the output of the `sofmax` function.
5. Add all the `wighted value vectors` together.

Notice in the figure below that we are doing matrix operations on `seq_length x embedding_size` matrices. This shows a toy example of two words with an embedding size of 3.



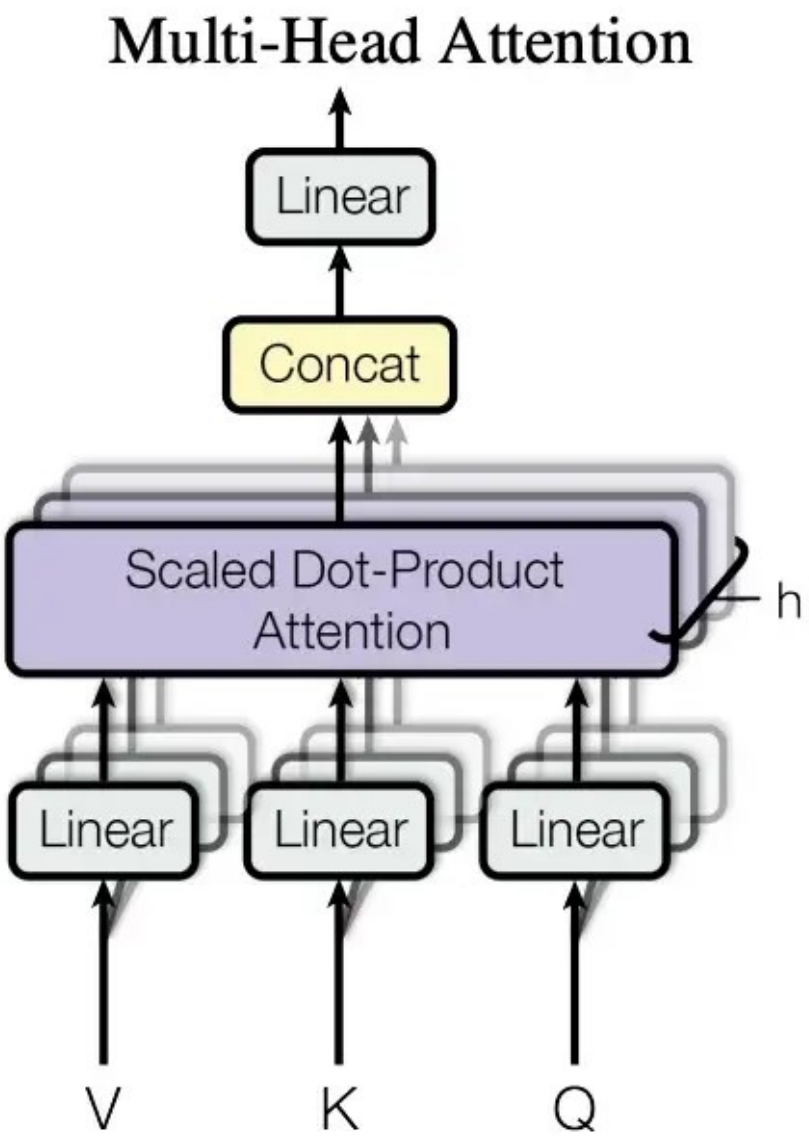
source: The Illustrated Transformer

The `dot product` results in a `seq_length x seq_length` matrix. Think `correlation matrix` where the relationships between any two members can be found by their intersections. In this case, the members are words and their intersections are `attention scores`.

Multiplying the `value matrix` by the `attention matrix` results once again in a `seq_length x embedding_size` matrix. This matrix holds the `contextual information` for each embedding.

Multi-head Self-Attention

What I described above is `single-head self-attention`. In practice we use `multi-head self-attention`. With `multi-head self-attention`, each word is processed by several attention heads. In the original paper, they used eight, which is what I use here.



source: Attention Is All You Need

The **`query`**, **`key`**, and **`value`** vectors are divided by the number of heads, and each segment is passed through a different head. This results in eight vectors which are then concatenated together and transformed by multiplying with another weight matrix so that the resultant vector is the size of the input embedding vector.

From Attention Is All You Need:

> *Multi − head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.*

$$MultiHead(Q, K, V) = Concat(head1, ..., headh)W^O \quad (1)$$

$$where, head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \quad (2)$$

In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = {}^{d_{model}}/_{h} = 64$. Due to the reduced dimension of each head, the total computational cost is similar to that of single − head attention with full dimensionality.

Prior to the **`softmax`** function, we need to apply a mask to the **`attention score`**. The mask will be either a **`padding mask`** or a combination of the **`padding mask`** and a **`look-ahead mask`**.

The sequences that are the input to the **`encoder`** and the **`decoder`** must be of the same length. Because sentences are of varying length, one must either truncate the sentence or pad the sentence so that they are all of a specified length. As a result, the padding might be interpreted as meaningful by the **`attention`** mechanism leading to unreliable learning.

To mitigate this effect, a **`padding mask`** is applied so that when added to the **`attention scores,`** the values in the padded positions become so small as to be ignored. This is achieved by setting the positions in the mask to a value close to **`negative infinity`**.

The padding mask is considered optional, but I believe that most implementations choose to use it.

Transformers are **`autoregressive models`**. They look at all of the previous input in a sequence in order to predict the next output (**`token`**). But because unlike **`RNNs`**, they receive the entire sequence at once there needs to be a way to limit attention for any position so that only the portion of the sequence that comes before it is attended to. This is because it is desired that the model infers the next position without having a peak. If it were able to look at the next position, then it would just copy it.

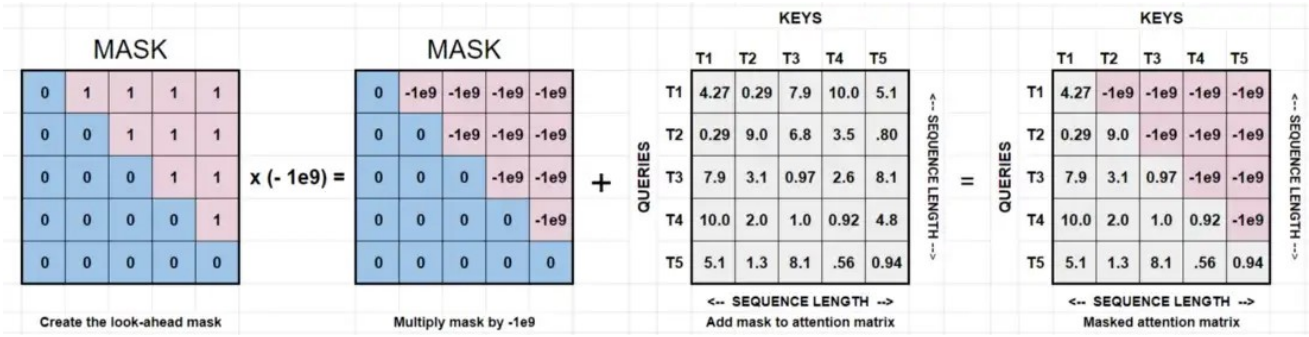
That's what the **`look-ahead mask`** accomplishes. All of the **`attention scores`** for words that occur to the right of a **`query`** word in the sequence are masked. This limits the **`query`** word to attend to itself and all words to the left in the sequence.

The **`look-ahead mask`** is only applied to the first Attention sub-layer of each decoder layer. That's because the decoder is where the inferencing occurs. Therefore, that is where the model should not be able to peek ahead.

How does it work?

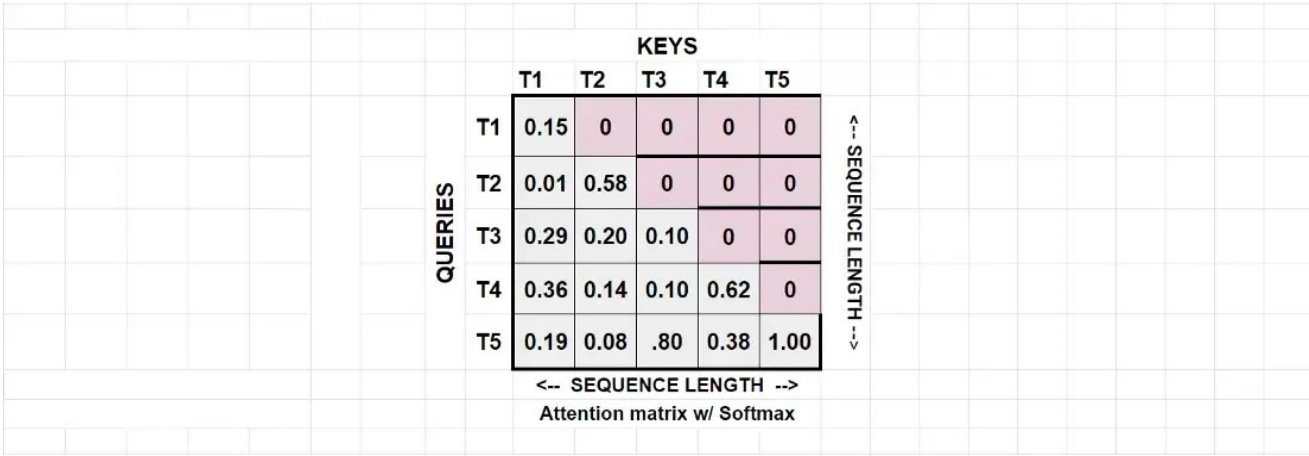
1. Create a mask where the values for the upper right triangle above the diagonal are all ones and the rest zeros.

2. Multiply the mask by `-1e9`.
3. Add the mask to the `attention matrix`.



Attention Mask + Attention Matrix

Next, apply the softmax along the `key` columns, converting those values to `probability distributions`. After applying `softmax` to the `attention matrix`, all of those extremely small values will become zero. So how does that matter?



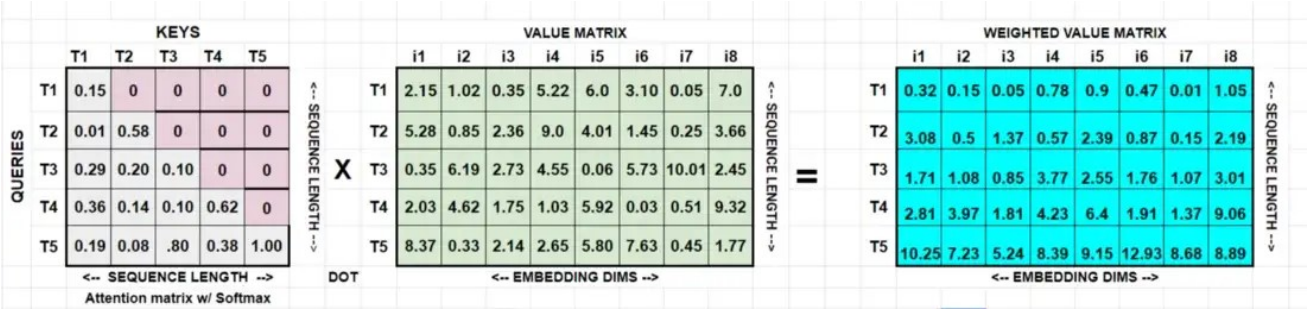
Attention Matrix with softmax

In the toy `attention matrix` in the figure, I've labeled both axes [*T1,…,Tn*]. The columns represent the token (word) `keys` and the rows are the token `queries`. The attention score for any, `query` *Ti*, are the values in its row with respect to the `key` columns [*T1,…,Tn*]. Notice that the topmost `query`, *T1* can only get the score for the leftmost `key` `T1`. There is no meaningful information for `T2,...,T5` (values too low). The `query` `T2` has access to the set of `keys` `[T2, $T1]`. And so on, until the last word, `T5`, which has access to the entire sequence.

So next, the `dot product` between the `attention weight matrix` and the `value matrix` is calculated. This has the effect of applying the weights to the values. If you are following along, you'll notice that the values in the *T1* embeddings are only affected by the *T1* `attention weights` in row 1. The *T2* embeddings only affected the

`attention weights` in row 2, *T1* and *T2*. And so it goes until you reach the final embedding (last word in sequence) whose embeddings are affected by all the weights in the final row. That's how autoregression is achieved.

This figure shows that which was just described.



Calculating the weighted value matrix

These steps occur in each of the attention heads. The next step is to concatenate all of the weighted matrices and pass them through a linear transformation to restore the original input dimensions.

And that's all there is to attention.

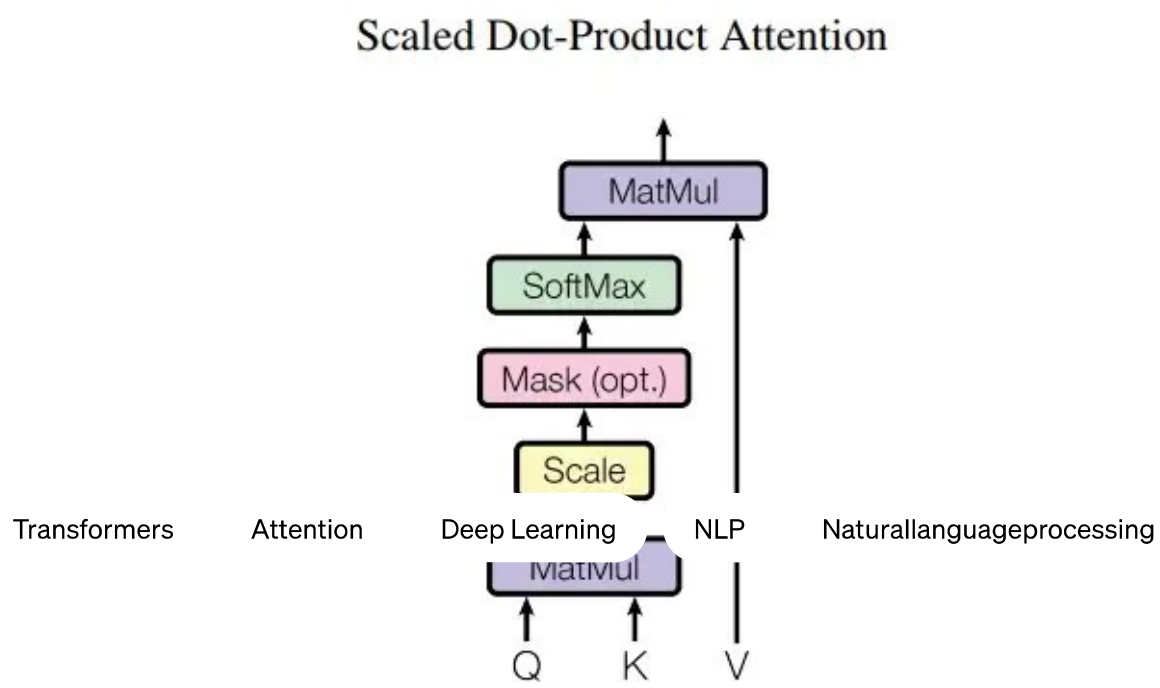
There is one more thing. The attention mask is not always applied alone. Remember that there is also padding to account for. Therefore, the `look-ahead mask` is applied along with the padding mask if we elect to use the padding mask.

Self-Attention Formula:

This is the self-attention formula with the mask included.

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

Representation of the Attention calculations:



source: Attention Is All You Need

Conclusion

In this article, we described the **`self-attention mechanism`** that is the heart and soul of the **`transformer model`**. We talked about

1. The reason for attention
2. The steps for calculating attention
3. The need for the padding mask
4. How the look-ahead mask works in the decoder to solve autoregression
5. Visually stepped through how the look-ahead mask works.

Thank you for sticking this far. I hope that you enjoyed the read and that it has been helpful in understanding attention.

Resources

- Attention Is All You Need by Ashish Vaswani, et al

- The Illustrated Transformer by Jay Alammar
- A Deep Dive Into Transformer Architecture — The Development of Transformer Models