All you need to know about ‘Attention’ and ‘Transformers’ — In-depth Understanding — Part 1

Attention, Self-Attention, Multi-head Attention, and Transformers

This is a long article that talks about almost everything one needs to know about the Attention mechanism including Self-Attention, Query, Keys, Values, Multi-Head Attention, Masked-Multi Head Attention, and Transformers including some details on BERT and GPT. I have hence divided the article into two parts. In this article, I cover all the Attention blocks, and in the next story, I will dive into the Transformer Network architecture.

Contents:

Introduction

The attention mechanism was first used in 2014 in computer vision, to try and understand what a neural network is looking at while making a prediction. This was one of the first steps to try and understand the outputs of Convolutional Neural Networks (CNNs). In 2015, attention was used first in Natural Language Processing (NLP) in Aligned Machine Translation. Finally, in 2017, the attention mechanism was used in Transformer networks for language modeling. Transformers have since surpassed the prediction accuracies of Recurrent Neural Networks (RNNs), to become state-of-the-art for NLP tasks.

1. Challenges with RNNs and how Transformer models can help overcome those challenges

1.1 *RNN problem 1* — Suffers issues with long-range dependencies. RNNs do not work well with long text documents.

Transformer Solution —Transformer networks almost exclusively use attention blocks. Attention helps to draw connections between any parts of the sequence, so long-range dependencies are not a problem anymore. With transformers, long-range dependencies have the same likelihood of being taken into account as any other short-range dependencies.

1.2. *RNN problem 2* — Suffers from gradient vanishing and gradient explosion.

Transformer Solution — There is little to no gradient vanishing or explosion problem. In Transformer networks, the entire sequence is trained simultaneously, and to build on that only a few more layers are added. So gradient vanishing or explosion is rarely an issue.

1.3. *RNN problem 3* — RNNs need larger training steps to reach a local/global minima. RNNs can be visualized as an unrolled network that is very deep. The size of the network depends on the length of the sequence. This gives rise to many parameters, and most of these parameters are interlinked with one another. As a result, the optimization requires a longer time to train and a lot of steps.

Transformer Solution — Requires fewer steps to train than an RNN.

1.4. *RNN problem 4* — RNNs do not allow parallel computation. GPUs help to achieve parallel computation. But RNNs work as sequence models, that is, all the computation in the network occurs sequentially and can not be parallelized.

Transformer Solution — No recurrence in the transformer networks allows parallel computation. So computation can be done in parallel for every step.

2. The Attention Mechanism

2.1 Self-Attention

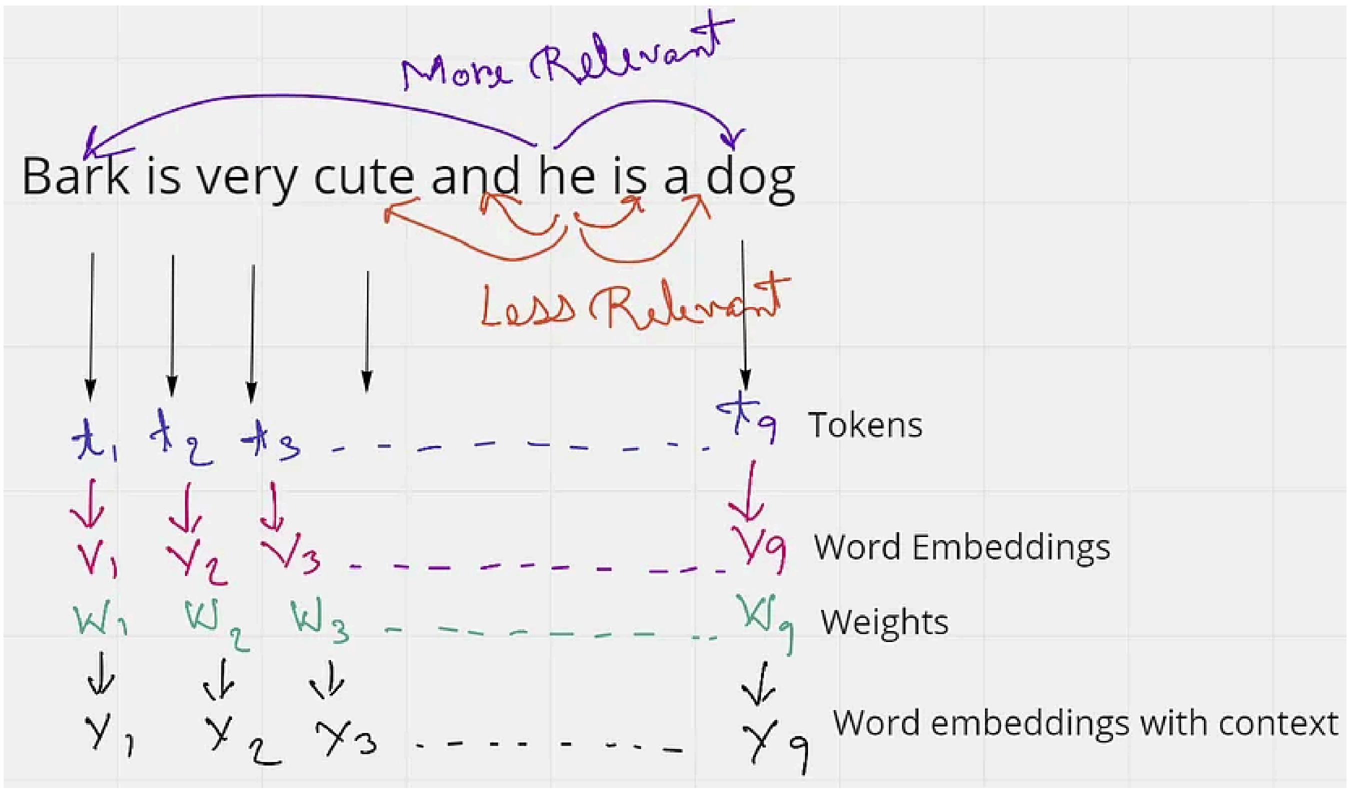
More Relevant
Bark is very cute and he is a dog
Less Relevant
t_1 t_2 t_3 ... t_9 Tokens
V_1 V_2 V_3 ... V_9 Word Embeddings
W_1 W_2 W_3 ... W_9 Weights
Y_1 Y_2 Y_3 ... Y_9 Word embeddings with context

Figure 2. Example to explain self-attention (Source: Image created by author)

Consider the sentence — " *Bark is very cute and he is a dog*". This sentence has 9 words or tokens. If we just consider the word 'he' in the sentence, we see that 'and' and 'is' are the two words in close proximity to it. But these words do not give the word 'he' any context. Rather the words 'Bark' and 'dog' are much more related to 'he' in the sentence. From this, we understand that proximity is not always relevant but context is more relevant in a sentence.

When this sentence is fed to a computer, it considers each word as a token *t,* and each token has a word embedding *V*. But these word embeddings have no context. So the idea is to apply some kind of weighing or similarity to obtain final word embedding *Y*, which has more context than the initial embedding *V*.

In an embedding space, similar words appear closer together or have similar embeddings. Such as the word 'king' will be more related to the word 'queen' and 'royalty', than with the word 'zebra'. Similarly, 'zebra' will be more related to 'horse' and 'stripes', than with the word 'emotion'. To know more about embedding space, please visit this video by Andrew Ng (NLP and Word Embeddings).

So, intuitively, if the word 'king' appears at the beginning of the sentence, and the word 'queen' appears at the end of the sentence, they should provide each other with better context. We use this idea to find the weight vectors *W,* by multiplying (dot product) the word embeddings together to gain more context. So, in the sentence

Bark is very cute and he is a dog, instead of using the word embeddings as it is, we multiply the embedding of each word with one another. Figure 3 should illustrate this better.

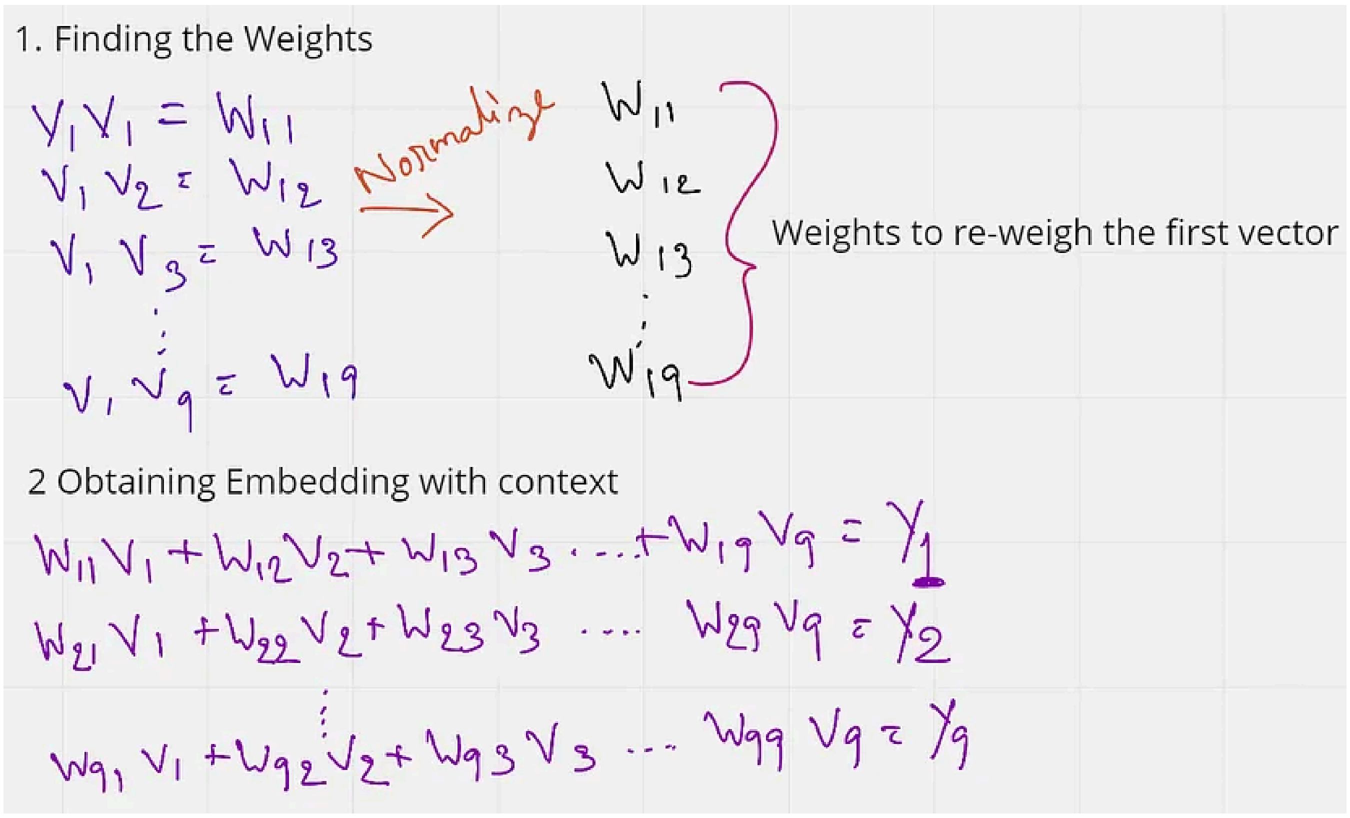
1. Finding the Weights
$V_1 V_1 = W_{11}$
$V_1 V_2 = W_{12}$
$V_1 V_3 = W_{13}$
$V_1 V_9 = W_{19}$
Normalize
W_{11}
W_{12}
W_{13}
W_{19}
Weights to re-weigh the first vector
2 Obtaining Embedding with context
$W_{11}V_1 + W_{12}V_2 + W_{13}V_3 \ldots + W_{19}V_9 = Y_1$
$W_{21}V_1 + W_{22}V_2 + W_{23}V_3 \ldots W_{29}V_9 = Y_2$
$W_{91}V_1 + W_{92}V_2 + W_{93}V_3 \ldots W_{99}V_9 = Y_9$

Figure 3. Finding weights and Obtaining final embedding (Source: Image created by author)

As we see in figure 3, we first find the weights by multiplying (dot product) the initial embedding of the first word with the embedding of all other words in the sentence. These weights (W11 to W19) are also normalized to have a sum of 1. Next, these weights are multiplied with the initial embeddings of all the words in the sentence.

W11 V1 + W12 V2 + …. W19 V9 = Y1

W11 to W19 are all weights that have the context of the first word V1. So when we are multiplying these weights to each word, we are essentially reweighing all the other words towards the first word. So in a sense, the word '*Bark*' is now tending more towards the words '*dog*' and '*cute*', rather than the word that comes right after it. And this, in a way gives some context.

This is repeated for all words so that each word gets some context from every other word in the sentence.

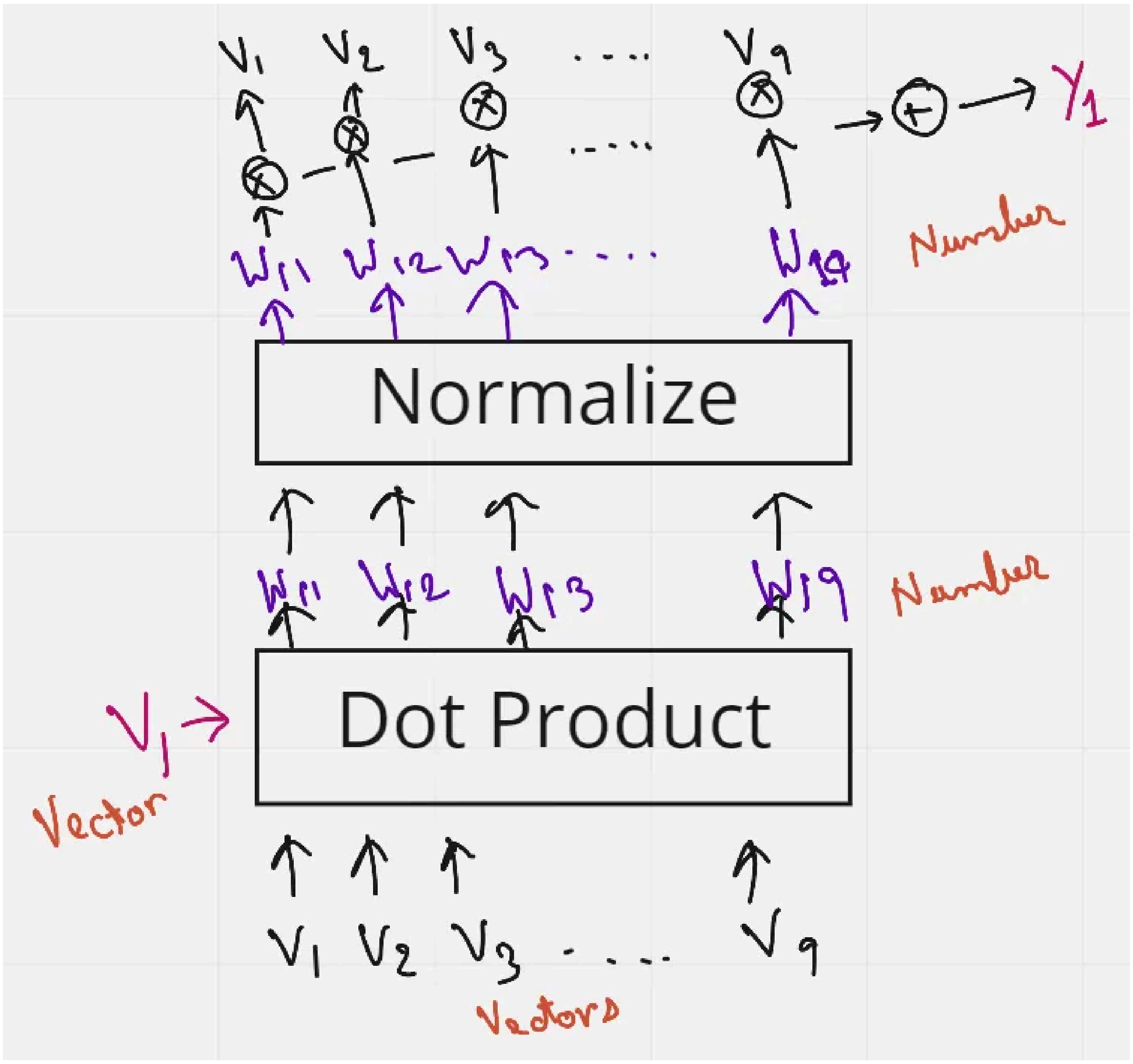

Figure 4. Pictorial representation of the above steps (Source: image created by author)

Figure 4 gives a better understanding of the above steps to obtain Y1, using a pictorial diagram.

What is interesting here is that no weights are trained, the order or proximity of the words have no influence on each other. Also, the process has no dependency on the length of the sentence, that is, more or fewer words in a sentence do not matter. This approach of adding some context to the words in a sentence is known as **Self-Attention**.

. . .

2.2 Query, Key, and Values

The issue with Self-Attention is that nothing is being trained. But maybe if we add some trainable parameters, the network can then learn some patterns which give much better context. This trainable parameter can be a matrix whose values are trained. So the idea of Query, Key, and Values was introduced.

Let's again consider the previous sentence — ” *Bark is very cute and he is a dog*”. In Figure 4 in self-attention, we see that the initial word embeddings (*V*) are used 3 times. 1st as a dot product between the first word embedding and all other words (including itself, 2nd) in the sentence to obtain the weights, and then multiplying them again (3rd time) to the weights, to obtain the final embedding with context. These 3 occurrences of the V’s can be replaced by the three terms **Query**, **Keys**, and **Values**.

Let's say we want to make all the words similar with respect to the first word V1. We then send V1 as the Query word. This query word will then do a dot product with all the words in the sentence (V1 to V9) — and these are the Keys. So the combination of the Query and the Keys give us the weights. These weights are then multiplied with all the words again (V1 to V9) which act as Values. There we have it, the Query, Keys, and the Values. If you still have some doubts, figure 5 should be able to clear them.

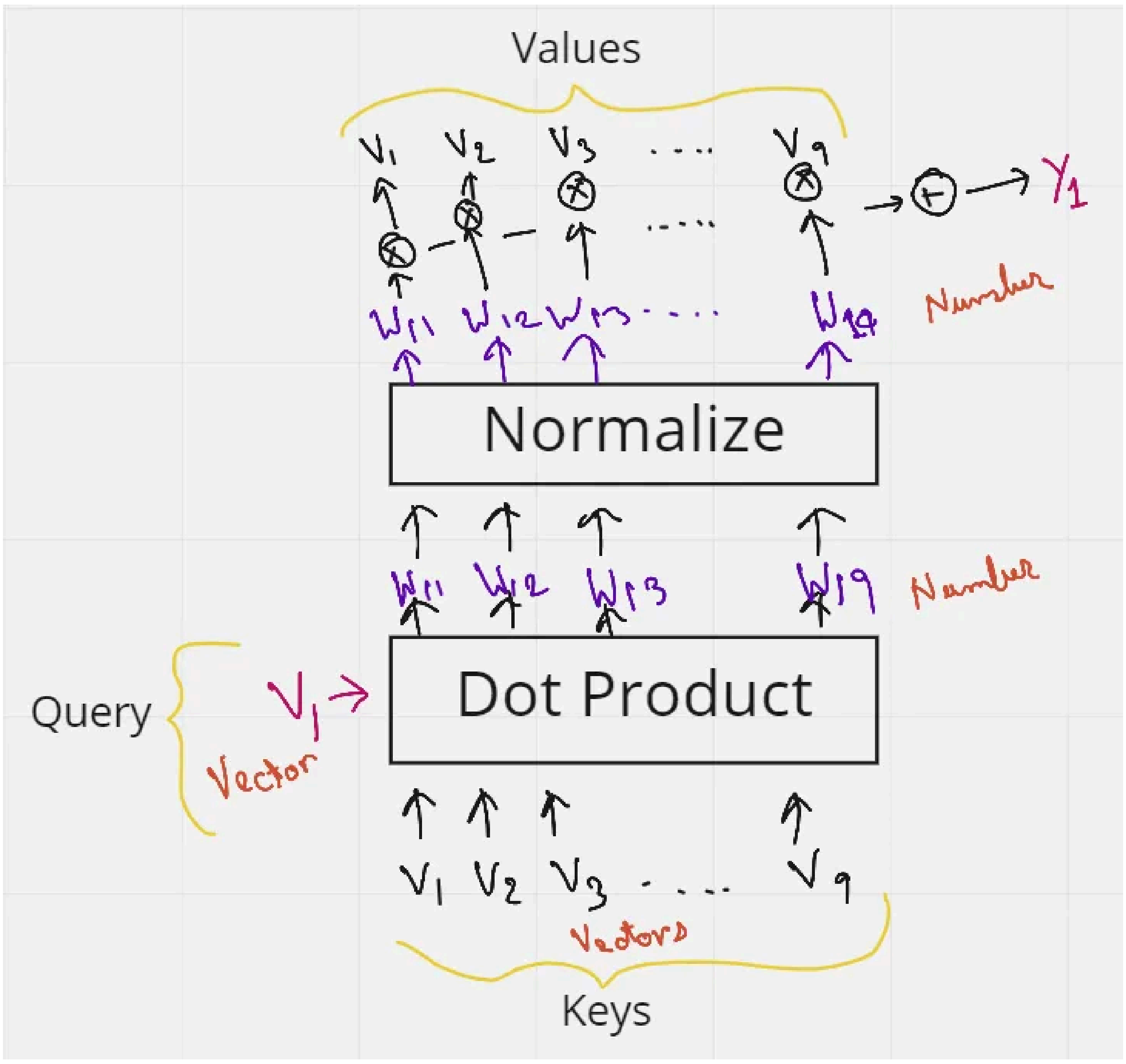

Figure 5. Denoting the Query, Keys, and the Values (Source: image created by author)

But wait, we have not added any matrix that can be trained yet. That's pretty simple. We know, if a 1 x k shaped vector is multiplied with a k x k shaped matrix, we get a 1 x k shaped vector as output. Keeping this in mind let's just multiply each key from V1 to V10 (each of shape 1 x k), with a matrix Mk (Key matrix) of shape k x k. Similarly, the query vector is multiplied with a matrix Mq (Query matrix)., and the Values vectors are multiplied with Values matrix Mv. All the values in these matrices Mk, Mq, and Mv can now be trained by the neural network, and give much better context than just using self-attention. Again for a better understanding, Figure 6 shows a pictorial representation of what I just explained.

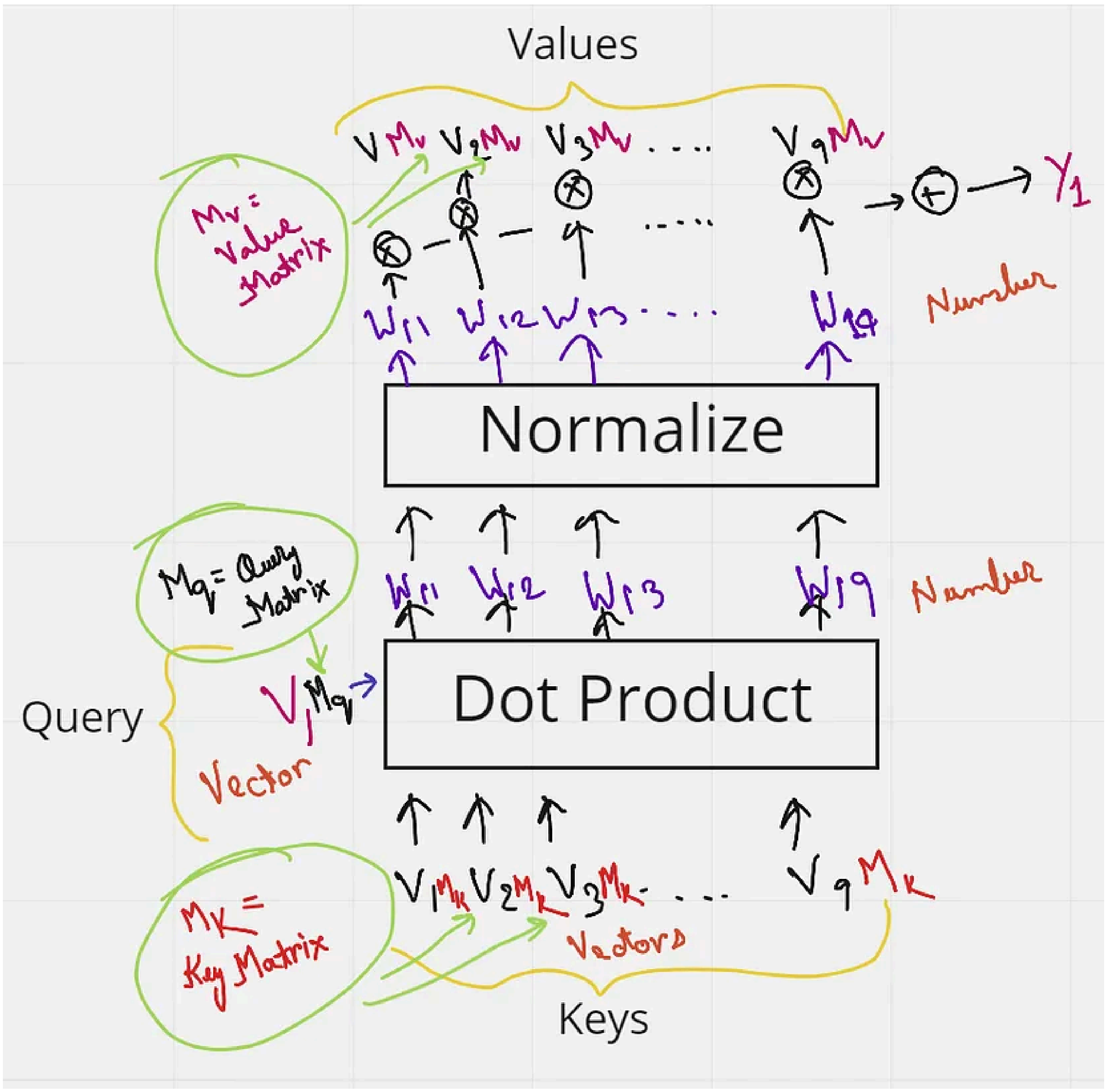

Figure 6. Key Matrix, Query matrix, and Value Matrix (Source: image created by author)

. . .

Now that we know the intuition of Keys, Query, and Values, let's look at a database analysis and the official steps and formulas behind Attention.

Let's try and understand the Attention mechanism, by looking into an example of a database. So, in a database, if we want to retrieve some **value** v*i* based on a **query** q and **key** k*i,* some operations can be done where we can use a query to identify a key that corresponds to a certain value. Attention can be thought to be a similar process to this database technique but in a more probabilistic manner. This is demonstrated in the figure below.

Figure 7, shows the steps of data retrieval in a database. Suppose we send a query into the database, some operations will find out which key in the database is the most similar to the query. Once the key is located, it will send out the value corresponding to that key as an output. In the figure, the operation finds that the Query is most similar to Key 5, and hence gives us the value 5 as output.

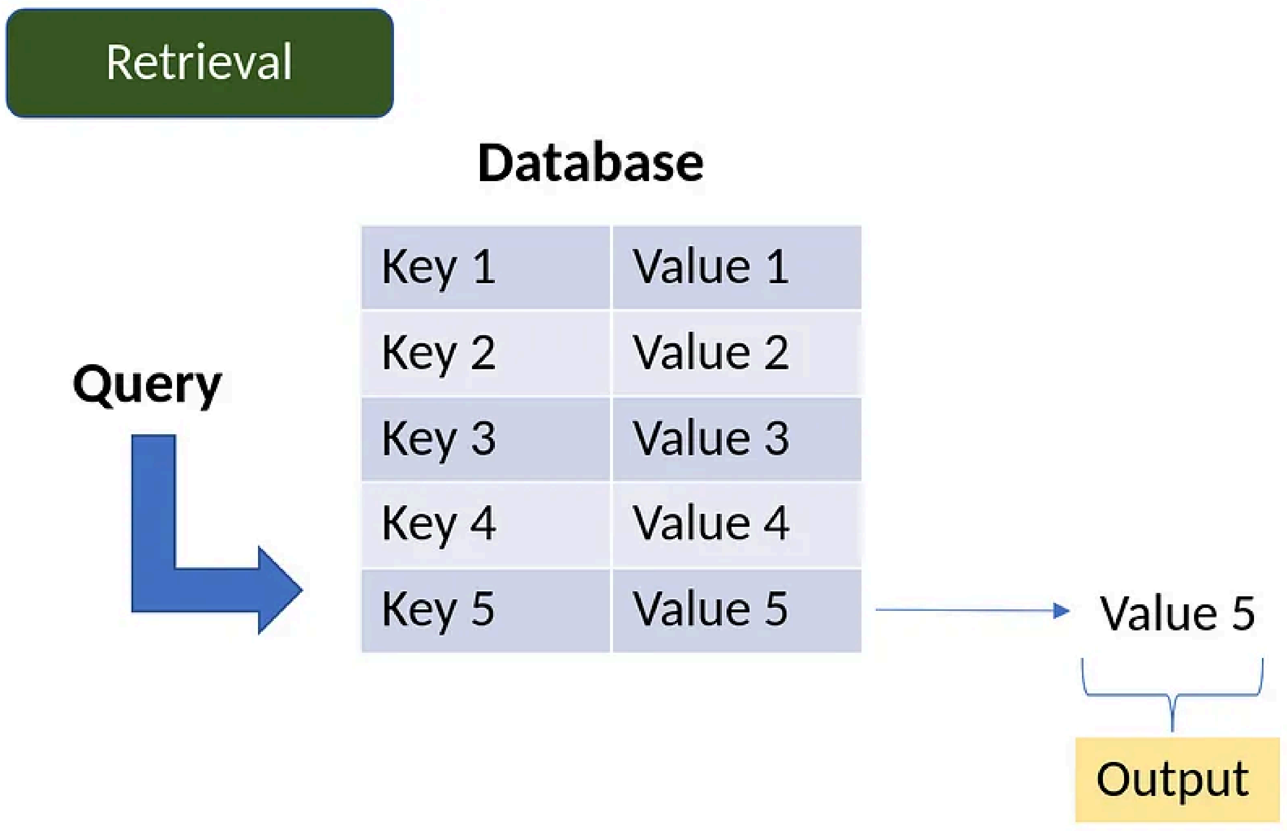

Figure 7. Value retrieval process in a database (Source: image created by Author)

The Attention mechanism is a neural architecture that mimics this process of retrieval.

$$attention(q, k, v) = \sum_i similarity(q, k_i) * v_i$$

1. The attention mechanism measures the similarity between the query q and each key k*i*.
2. This similarity returns a weight for each key's value.
3. Finally, it produces an output that is the weighted combination of all the values in our database.

The only difference between database retrieval and attention in a sense is that in database retrieval we only get one value as output, but here we get a weighted combination of values. In the attention mechanism, if a query is most similar to say, key 1 and key 4, then both these keys will get the most weights, and the output will be a combination of value 1 and value 4.

Figure 8 shows the steps required to get to the final attention value from the query, keys, and values. Each step is explained in detail below.

(The Keys *k* are vectors, the similarities *s* are scalars, the weights (softmax) *a* are scalars, and the Values *V* are vectors)

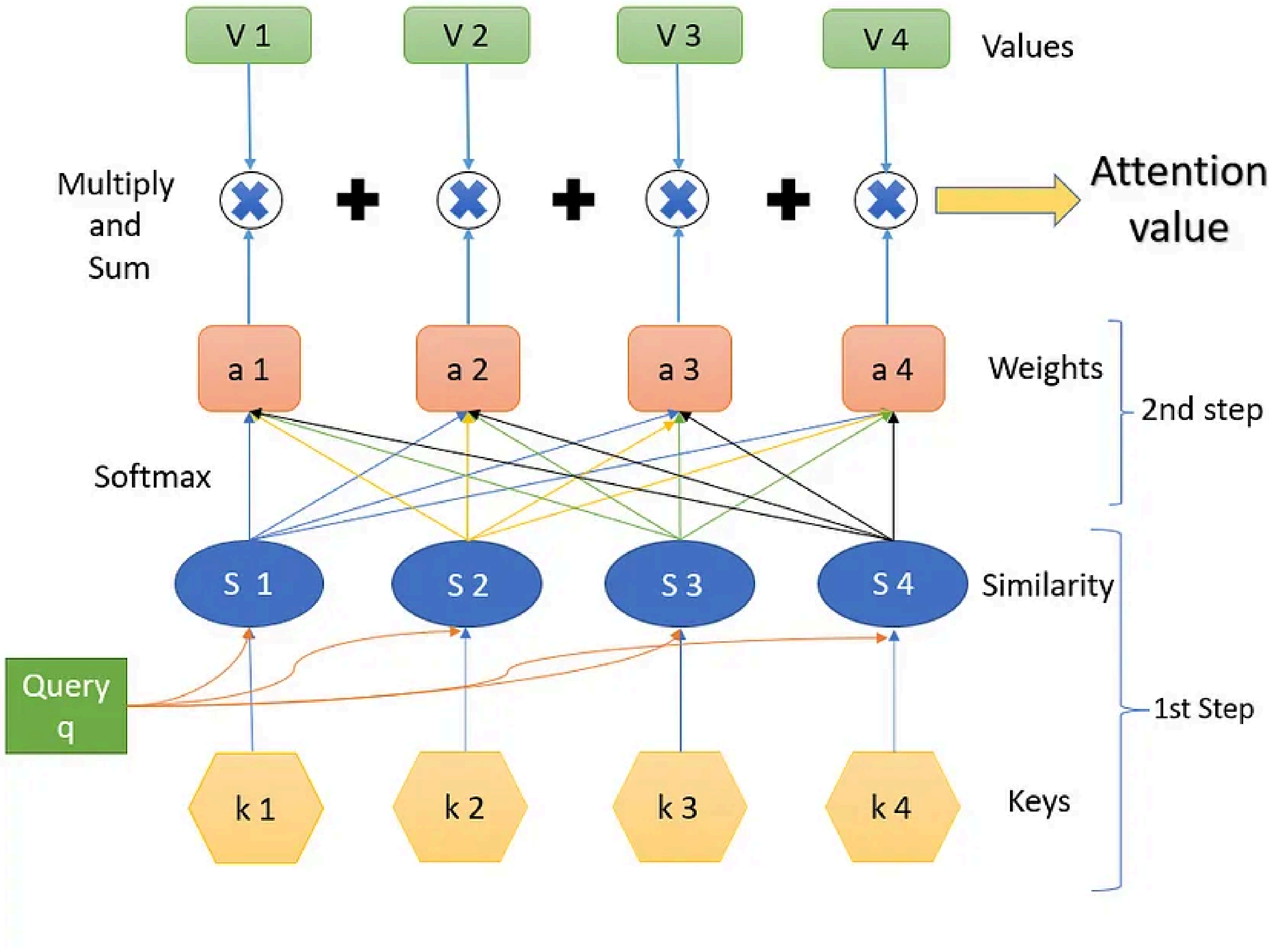

Figure 8. Steps to attain the Attention Value (Source: Image created by author)

Step 1.

Step 1 contains the keys and the query and the respective similarity measures. The query *q* influences the similarity. What we have are the query and the keys, and we calculate the similarity. The similarity is some function of the query *q* and the keys *k*. Both the query and the keys are some embedding vectors. Similarity *S* can be calculated using various methods as shown in figure 9.

Similarity Calculation

$$
\begin{aligned}
S_i = f(q,k_i) &= q^T k_i \;\ldots\ldots\ldots\ldots\; \textit{dot product} \\
&\qquad\qquad (T \textit{ is transpose}) \\
&= q^T k_i / \sqrt{d} \;\ldots\ldots\; \textit{scaled dot product} \\
&\qquad\qquad (d \textit{ is dimensionality of each key}) \\
&= q^T W k_i \;\ldots\ldots\ldots\; \textit{general dot product} \\
&\qquad\qquad (\textit{query projected into a new space using } W, \\
&\qquad\qquad W \textit{ is weight matrix}) \\
&= \text{Kernel Methods (Mapping the two vectors q and} \\
&\qquad\qquad \text{k in a new space using a non}-\text{linear function)}
\end{aligned}
$$

Figure 9. Ways to calculate the Similarity (Souce: image created by author)

Similarity can be a simple dot product of the query and the key. It can be scaled dot product, where the dot product of *q* and *k,* is divided by the square root of the dimensionality of each key, *d.* These are the most commonly used two techniques to find the similarity.

Often a query is projected into a new space by using a weight matrix *W,* and then a dot product is made with the key *k*. Kernel methods can also be used as a similarity.

Step 2.

Step 2 is finding the weights *a*. This is done using '*SoftMax*'. The formula is shown below. (exp is exponential)

$$a_i = \frac{exp(S_i)}{\sum_j \exp(S_j)}$$

The similarities are connected to the weights like a fully connected layer.

Step 3.

Step 3 is a weighted combination of the results of the softmax (*a*) with the corresponding values (*V*). The 1st value of *a* is multiplied with the first value of *V* and is then summed with the product of the 2nd value of *a* with the 2nd value of Values *V*, and so on. The final output that we obtain is the resulting attention value that is desired.

$$attention\ value\ =\ \sum_i a_i V_i$$

Summary of the three steps:

With the help of the query *q* and the keys *k*, we obtain the attention value, which is a weighted sum/linear combination of the Values *V*, and the weights come from some sort of similarity between the query and the keys.

. . .

2.3 Neural network representation of Attention

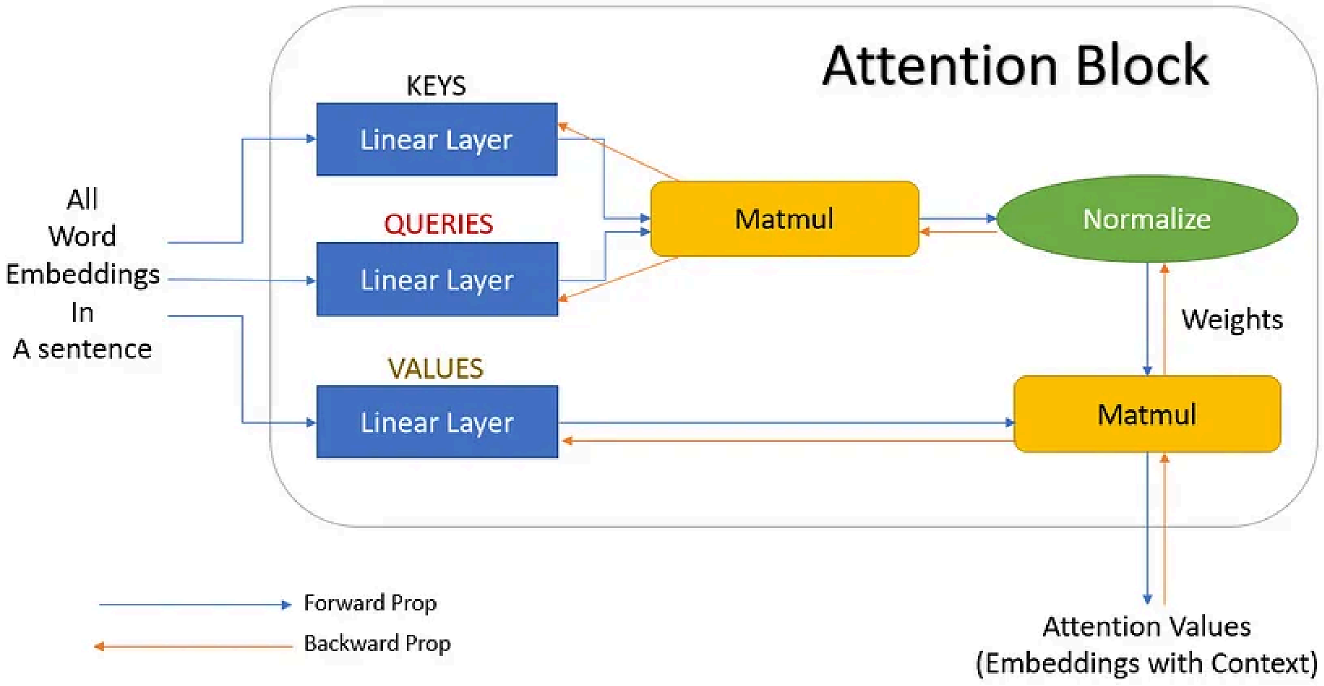

Figure 10. Neural network representation of Attention block (Source: image created by author)

Figure 10 shows the neural network representation of an attention block. The word embeddings are first passed into some linear layers. These linear layers do not have a ‘bias’ term, and hence are nothing but matrix multiplications. One of these layers is denoted as ‘keys’, the other as ‘queries’, and the last one as ‘values’. If a matrix multiplication is performed between the keys and the queries and are then normalized, we get the weights. These weights are then multiplied by the values, and summed up, to get the final attention vector. This block can now be used in a neural network and is known as the Attention block. Multiple such attention blocks can be added to provide more context. And the best part is, we can get a gradient backpropagating to update the attention block (weights of keys, queries, values). **再看看前面的黑体加粗的那一段**

. . .

2.4 Multi-Head Attention

To overcome some of the pitfalls of using single attention, multi-head attention is used. Let's go back to the sentence — ” *Bark is very cute and he is a dog*”. Here, if we take the word ‘dog’, grammatically we understand that the words ‘Bark’, ‘cute’, and ‘he’ should have some significance or relevance with the word ‘dog’. These words say that the dog’s name is Bark, it is a male dog, and that he is a cute dog. Just one attention mechanism may not be able to correctly identify these three words as relevant to ‘dog’, and we can say that three attentions are better here to signify the three words with the word ‘dog’. This reduces the load on one attention to find all significant words and also increases the chances of finding more relevant words easily.

可以直观理解为为1个attention捕获一种类型的上下文，3个attention分别捕获性别相关、姓名相关、性格相关的上下文。

So let's add more linear layers as the keys, queries, and values. These linear layers are training in parallel, and have independent weights to one another. So now, each of the values, keys, and queries gives us three outputs instead of one. These 3 keys and queries now give three different weights. These three weights then with matrix multiplication with the three values, to give three multiple outputs. These three attention blocks are finally concatenated to give one final attention output. This representation is shown in figure 11.

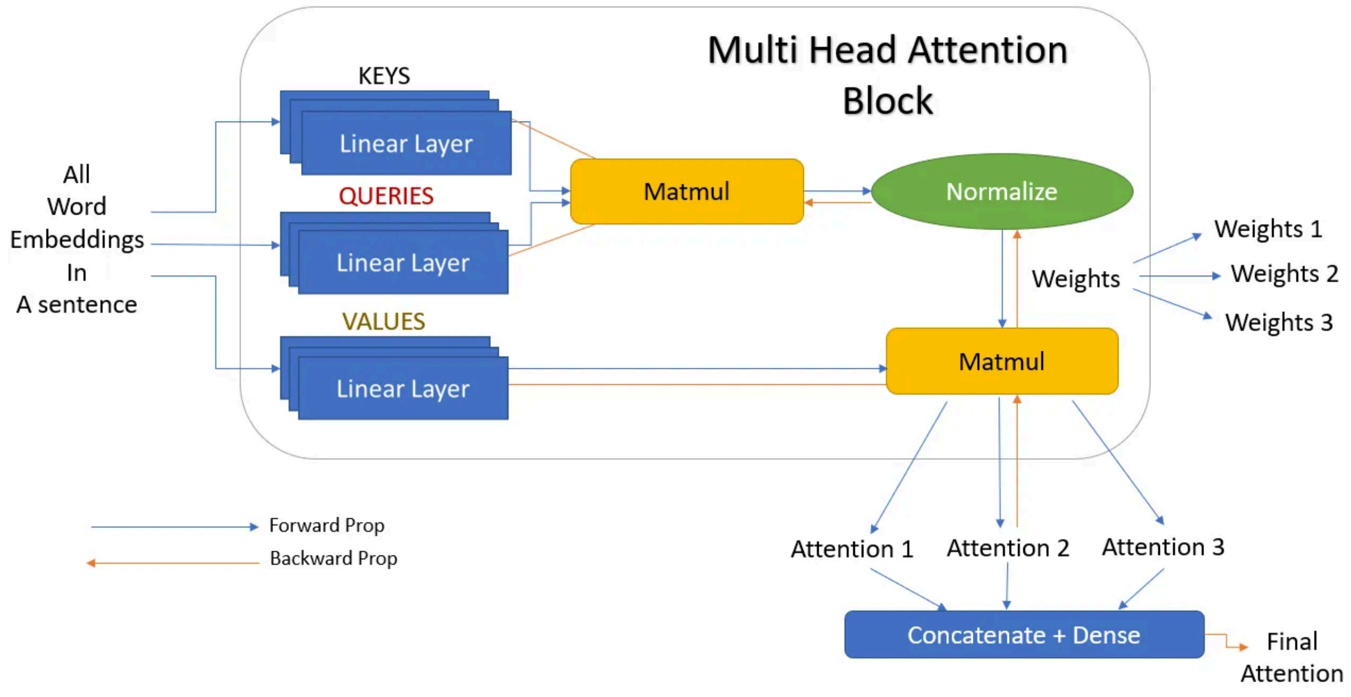

Figure 11. Multi-head attention with 3 linear layers (Source: image created by author)

But 3 is just a random number we chose. In the actual scenario, these can be any number of linear layers, and these are called heads (h). That is there can be h number of linear layers giving h attention outputs which are then concatenated together. And this is exactly why it is called multi-head attention (multiple heads). The simpler version of figure 11, but with h number of heads is shown in figure 12.

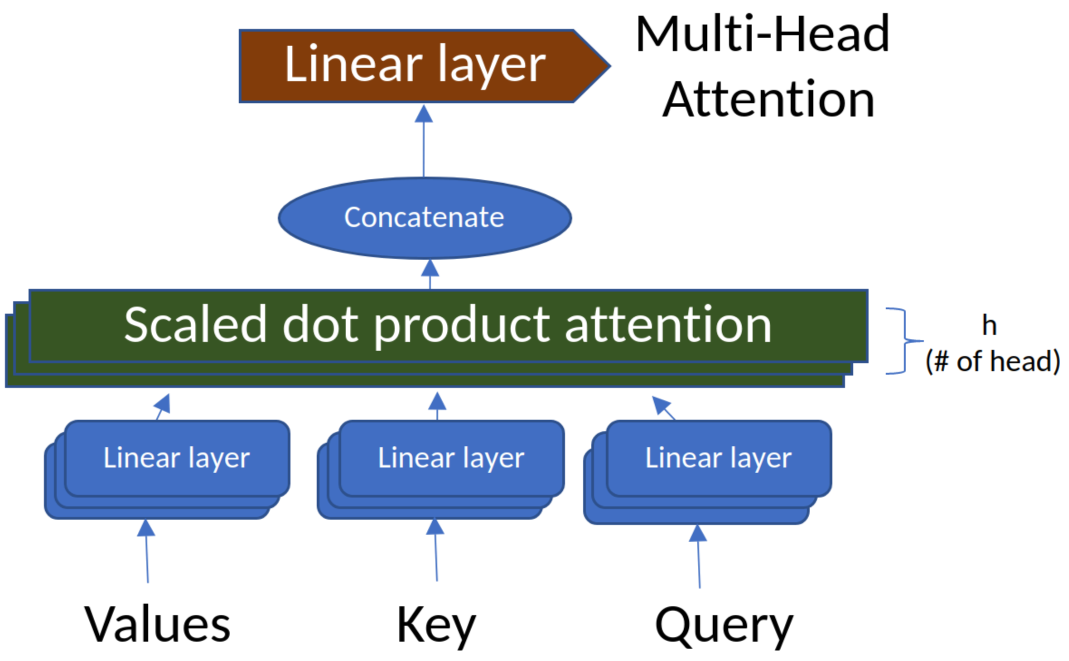

Figure 12. Multi-Head attention with 'h' layers (Source: Image created by author)

. . .

总结一下：Attention模块的输入是每个时间步的特征向量，如，词向量，输出是每个时间步的考虑上下文的特征向量，Multi-Head表示有多个Attention，从多个角度捕获上下文相关性。

Now that we understand the mechanism and idea behind Attention, Query, Keys, Values, and Multi-Head attention, we have covered all the important building blocks of a Transformer network. In the next story, I will talk about how all these blocks are stacked together to form the Transformer Network, and also talk about some networks based on Transformers such as BERT and GPT.

. . .

References:

1. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17). Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

2. https://towardsdatascience.com/all-you-need-to-know-about-attention-and-transformers-in-depth-understanding-part-1-552f0b41d021#8607