Global vs. Local Attention

So far, we have discussed the most basic Attention mechanism where all the inputs have been given some importance. Let's take things a bit deeper now.

The term "global" Attention is appropriate because all the inputs are given importance. Originally, the Global Attention (defined by Luong et al 2015) had a few subtle differences with the Attention concept we discussed previously.

The differentiation is that it considers all the hidden states of both the encoder LSTM and decoder LSTM to calculate a "variable-length context vector **ct**, whereas Bahdanau et al. used the previous hidden state of the unidirectional decoder LSTM and all the hidden states of the encoder LSTM to calculate the context vector.

In encoder-decoder architectures, the score generally is a function of the encoder and the decoder hidden states. Any function is valid as long as it captures the relative importance of the input words with respect to the output word.

When a "global" Attention layer is applied, a lot of computation is incurred. This is because all the hidden states must be taken into consideration, concatenated into a matrix, and multiplied with a weight matrix of correct dimensions to get the final layer of the feedforward connection.

So, as the input size increases, the matrix size also increases. In simple terms, the number of nodes in the feedforward connection increases and in effect it increases computation.

Can we reduce this in any way? Yes! Local Attention is the answer.

Intuitively, when we try to infer something from any given information, our mind tends to intelligently reduce the search space further and further by taking only the most relevant inputs.

The idea of Global and Local Attention was inspired by the concepts of Soft and Hard Attention used mainly in computer vision tasks.

Soft Attention is the global Attention where all image patches are given some weight; but in hard Attention, only one image patch is considered at a time.

But local Attention is not the same as the hard Attention used in the image captioning task. On the contrary, it is a blend of both the concepts, where instead of considering all the encoded inputs, only a part is considered for the context vector generation. This not only avoids expensive computation incurred in soft Attention but is also easier to train than hard Attention.

How can this be achieved in the first place? Here, the model tries to predict a position **pt** in the sequence of the embeddings of the input words. Around the position **pt**, it considers a window of size, say, **2D**. Therefore, the context vector is generated as a weighted average of the inputs in a position **[pt – D,pt + D]** where **D** is empirically chosen.

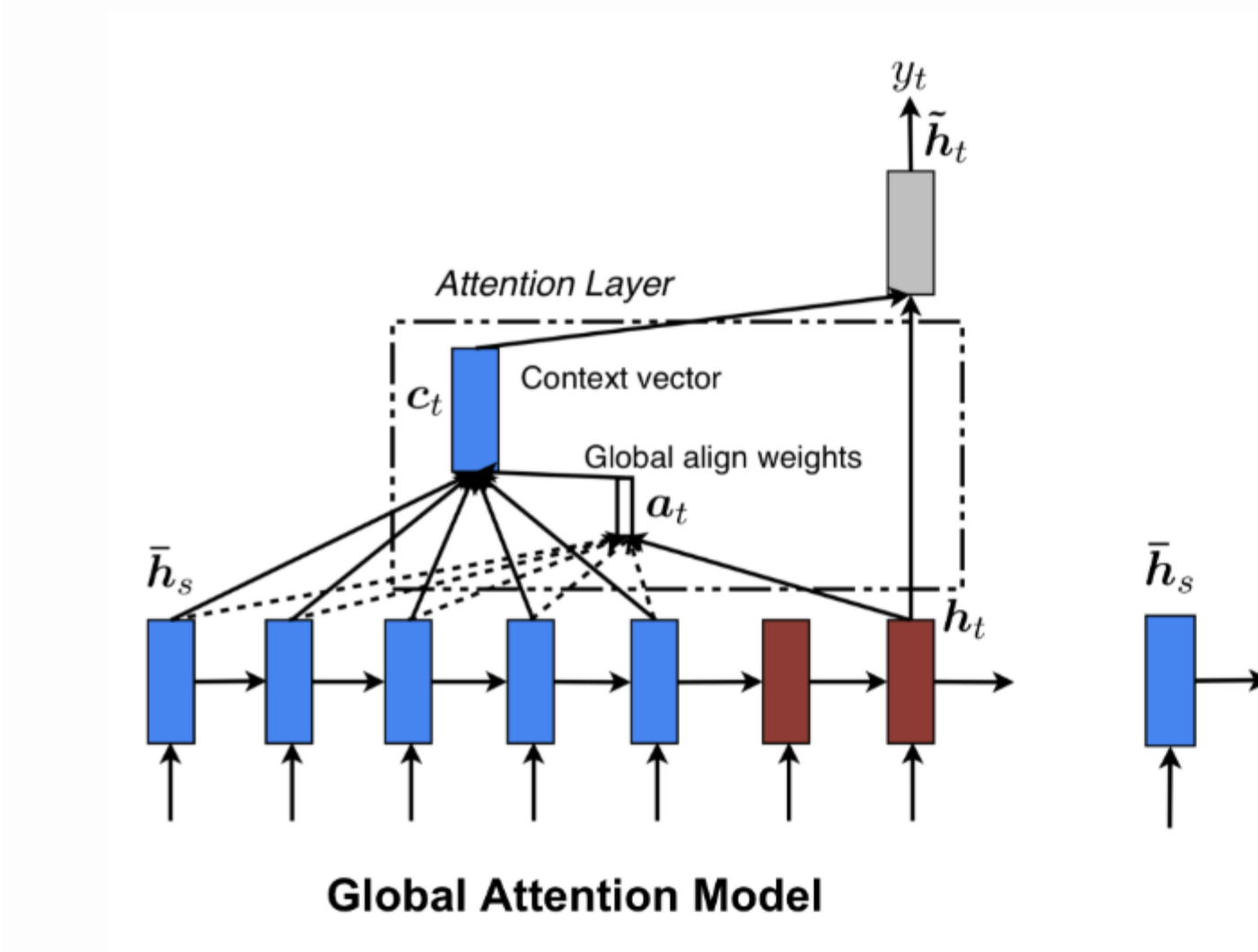
Furthermore, there can be two types of alignments:

1. **Monotonic alignment**, where pt is set to t, assuming that at time t, only the information in the neighborhood of t matters
2. **Predictive alignment** where the model itself predicts the alignment position as follows:

$$p_t = S \cdot \text{sigmoid}(\boldsymbol{v}_p^\top \tanh(\boldsymbol{W}_p \boldsymbol{h}_t))$$

where 'Vp' and 'Wp' are the model parameters that are learned during training and 'S' is the source sentence length. Clearly, pt ε [0,S].

The figures below demonstrate the difference between the Global and Local Attention mechanism. Global Attention considers all hidden states (blue) whereas local Attention considers only a subset:



Global Attention Model

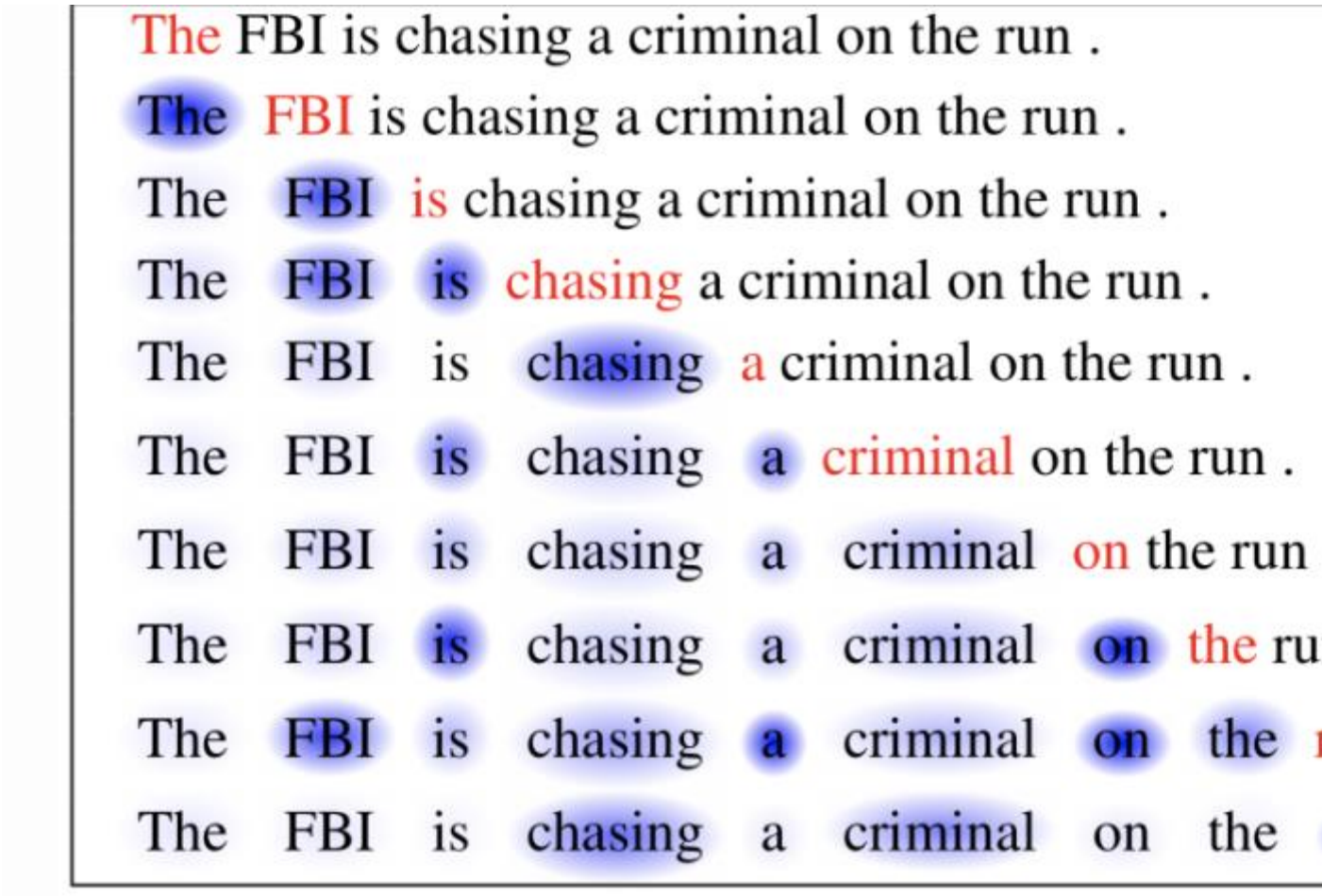
Transformers – Attention is All You Need

The paper named **"Attention is All You Need"** by **Vaswani et al** is one of the most important contributions to Attention so far. They have redefined Attention by providing a very generic and broad definition of Attention based on **key**, **query,** and **values**. They have referenced another concept called **multi-headed Attention.** Let's discuss this briefly.

First, let's define what **"self-Attention"** is. Cheng et al, in their paper named **"Long Short-Term Memory-Networks for Machine Reading"**, defined self-Attention as the mechanism of relating different positions of a single sequence or sentence in order to gain a more vivid representation.

Machine reader is an algorithm that can automatically understand the text given to it. We have taken the below picture from the paper. The red words are read or processed at the current instant, and the blue words are the memories. The different shades represent the degree of memory activation.

When we are reading or processing the sentence word by word, where previously seen words are also emphasized on, is inferred from the shades, and this is exactly what self-Attention in a machine reader does.



Previously, to calculate the Attention for a word in the sentence, the mechanism of score calculation was to either use a dot product or some other function of the word with the hidden state representations of the previously seen words. In this paper, a fundamentally same but a more generic concept altogether has been proposed.

Let's say we want to calculate the Attention for the word "chasing". The mechanism would be to take a dot product of the embedding of "chasing" with the embedding of each of the previously seen words like "The", "FBI", and "is".

Now, according to the generalized definition, each embedding of the word should have three different vectors corresponding to it, namely **Key**, **Query,** and **Value**. We can easily derive these vectors using matrix multiplications.

Whenever we are required to calculate the Attention of a target word with respect to the input embeddings, we should use the **Query** of the target and the **Key** of the input to calculate a matching score, and these matching scores then act as the weights of the **Value** vectors during summation.

Now, you might ask what these Key, Query and Value vectors are. These are basically abstractions of the embedding vectors in different subspaces. Think of it in this way: you raise a query; the query hits the key of the input vector. The Key can be compared with the memory location read from, and the value is the value to be read from the memory location. Simple, right?

If the dimension of the embeddings is **(D, 1)** and we want a **Key** vector of dimension (**D/3, 1),** we must multiply the embedding by a matrix **Wk** of dimension (**D/3, D).** So, the key vector becomes **K=Wk*E.** Similarly, for **Query** and **Value** vectors, the equations will be **Q=Wq*E, V=Wv*E** (**E** is the embedding vector of any word).

Now, to calculate the Attention for the word **"chasing"**, we need to take the dot product of the **query** vector of the embedding of **"chasing"** to the **key** vector of each of the previous words, i.e., the key vectors corresponding to the words **"The"**, **"FBI"** and **"is"**. Then these values are divided by D (the dimension of the embeddings) followed by a **softmax** operation. So, the operations are respectively:

- **softmax(Q"chasing"** . K"The" / D)
- **softmax(Q"chasing"** .K"FBI" / D)
- **softmax(Q"chasing"** . K"is" / D)

Basically, this is a function **f(Qtarget, Kinput)** of the query vector of the target word and the key vector of the input embeddings. It doesn't necessarily have to be a dot product of **Q** and **K.** Anyone can choose a function of his/her own choice.

Next, let's say the vector thus obtained is **[0.2, 0.5, 0.3]**. These values are the **"alignment scores"** for the calculation of Attention. These alignment scores are multiplied with the **value vector** of each of the input embeddings and these weighted value vectors are added to get the **context vector**:

C"chasing"= 0.2 * VThe + 0.5* V"FBI" + 0.3 * V"is"

Practically, all the embedded input vectors are combined in a single matrix **X,** which is multiplied with common weight matrices **Wk**, **Wq, Wv** to get **K, Q and V** matrices respectively. Now the compact equation becomes:

$$Z=\text{Softmax}(Q*K^T/D)V$$

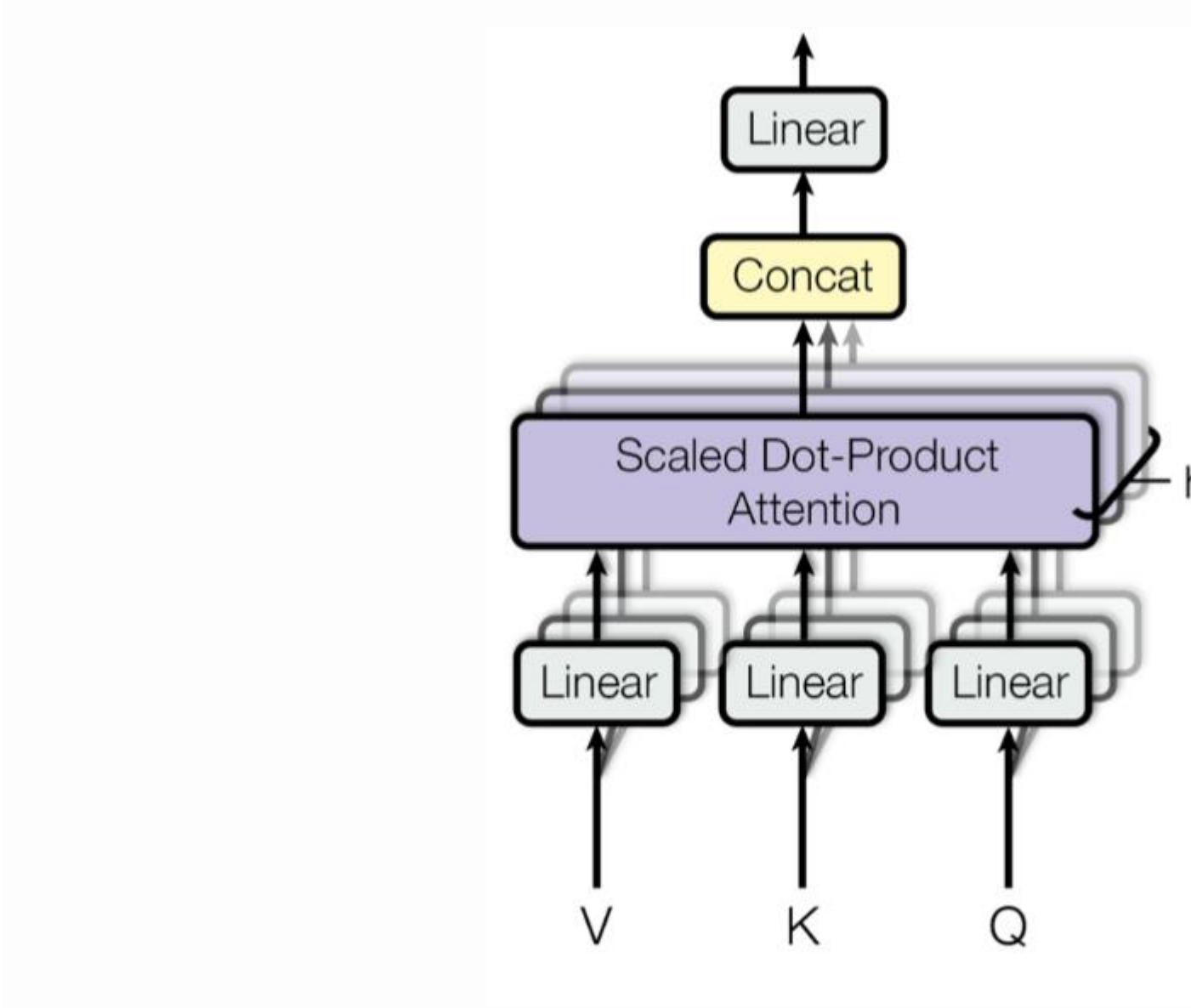
Therefore, the context vector is a function of Key, Query and Value F(K, Q, V).

The Bahdanau Attention or all other previous works related to Attention are the special cases of the Attention Mechanisms described in this work. The salient feature/key highlight is that the single embedded vector is used to work as **Key, Query and Value** vectors simultaneously.

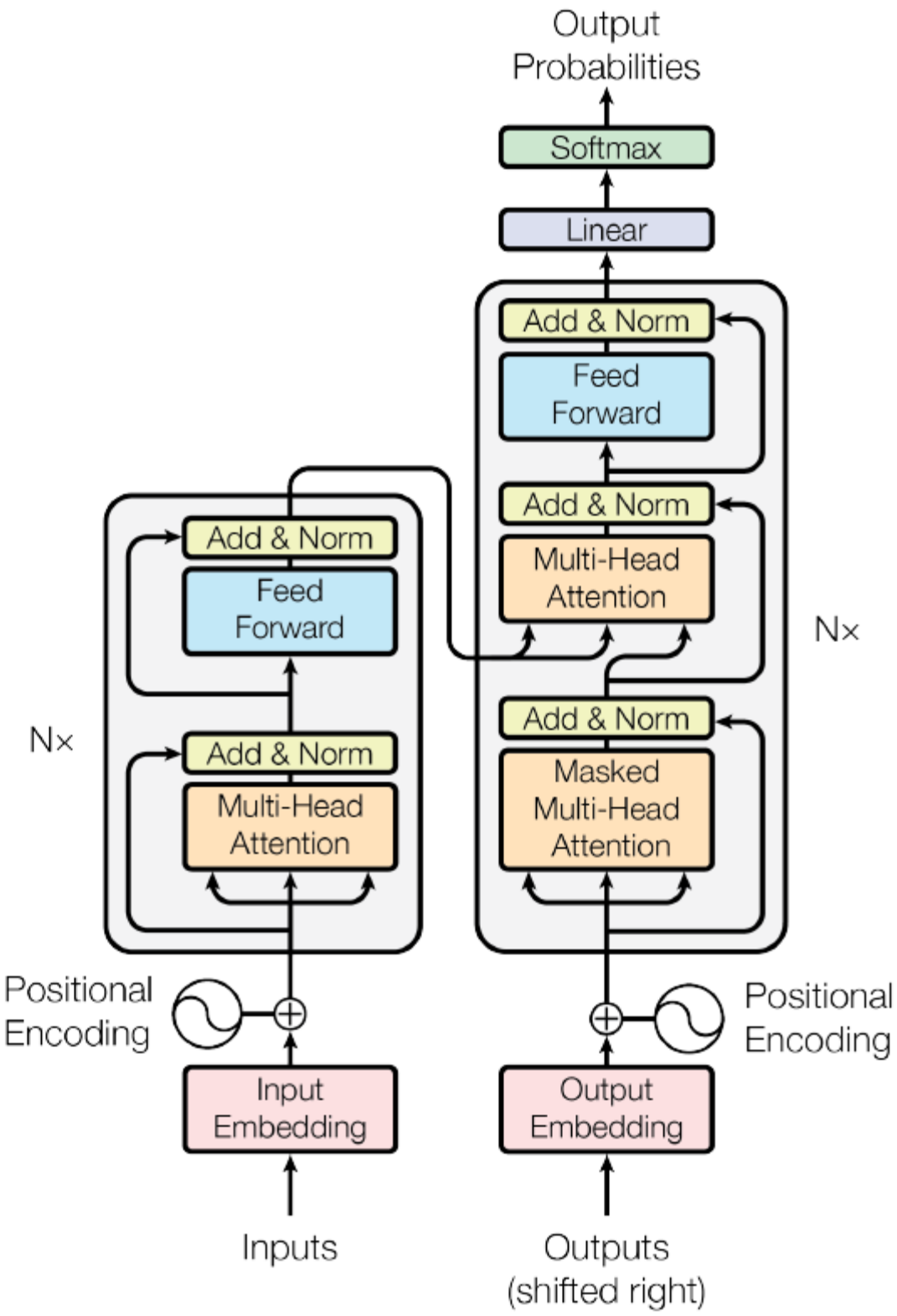
In multi-headed Attention, matrix **X** is multiplied by different **Wk**, **Wq** and **Wv** matrices to get different **K, Q and V** matrices respectively. And we end up with different **Z** matrices, i.e., embedding of each input word is projected into different "representation subspaces".

In, say, 3-headed self-Attention, corresponding to the “chasing” word, there will be 3 different **Z** matrices also called **“Attention Heads”.** These Attention heads are concatenated and multiplied with a single weight matrix to get a single Attention head that will capture the information from all the Attention heads.

The picture below depicts the multi-head Attention. You can see that there are multiple Attention heads arising from different V, K, Q vectors, and they are concatenated:



The actual transformer architecture is a bit more complicated. You can read it in much more detail here.



This image above is the transformer architecture. We see that something called 'positional encoding' has been used and added with the embedding of the inputs in both the encoder and decoder.

The models that we have described so far had no way to account for the order of the input words. They have tried to capture this through positional encoding. **This mechanism adds a vector to each input embedding, and all these vectors follow a pattern that helps to determine the position of each word, or the distances between different words in the input.**

As shown in the figure, on top of this positional encoding + input embedding layer, there are two sublayers:

1. In the first sublayer, there is a multi-head self-attention layer. There is an additive residual connection from the output of the positional encoding to the output of the multi-head self-attention, on top of which they have applied a layer normalization layer. The **layer normalization** is a technique **(Hinton, 2016)** similar to batch normalization where instead of considering the whole minibatch of data for calculating the normalization statistics, all the hidden units in the same layer of the network have been considered in the calculations. This overcomes the drawback of estimating the statistics for the summed input to any neuron over a minibatch of the training samples. Thus, it is convenient to use in RNN/LSTM
2. In the second sublayer, instead of the multi-head self-attention, there is a feedforward layer (as shown), and all other connections are the same

On the decoder side, apart from the two layers described above, there is another layer that applies multi-head Attention on top of the encoder stack. Then, after a sublayer followed by one linear and one softmax layer, we get the output probabilities from the decoder.