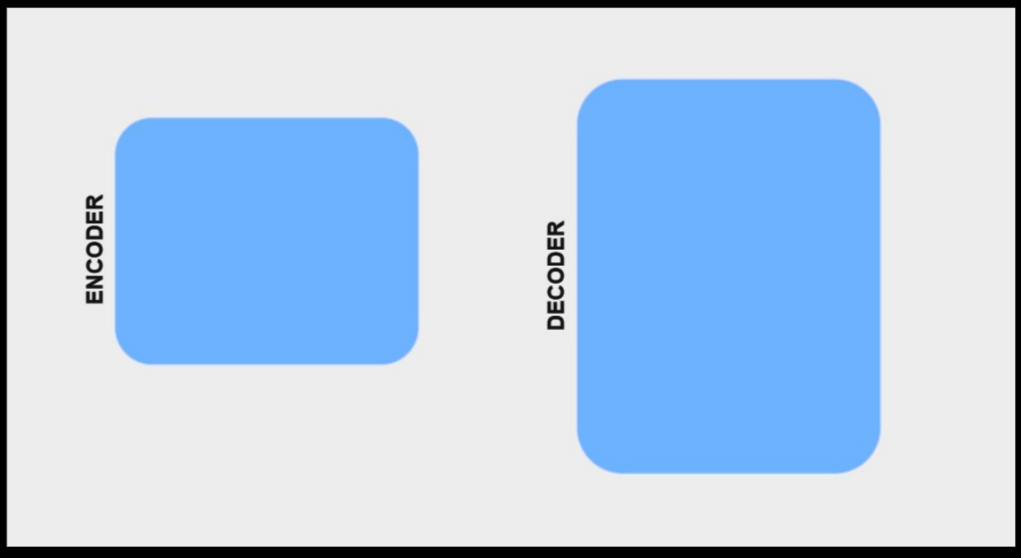
Transformer

Transformers are a type of deep learning model architecture designed to handle sequential data, primarily used in natural language processing (NLP) tasks. They were introduced in the paper "Attention is All You Need" by Vaswani et al. in 2017 by the Google .



It's a encoder-decoder based architecture that's provides multiple advantages like parallel processing, self- attention mechanism etc.

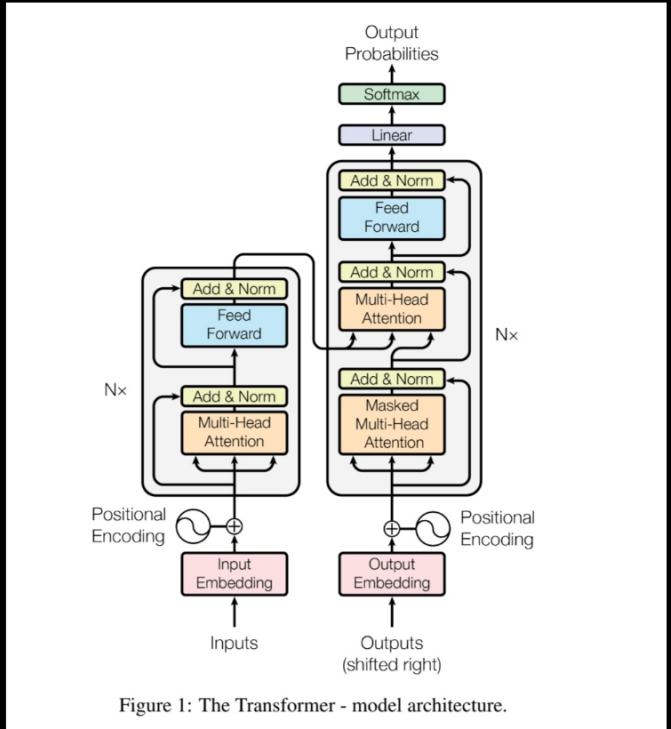


Figure 1: The Transformer - model architecture.

Subword Tokenization

Subword tokenization is a technique used in natural language processing (NLP) to break down text into smaller units called subwords, which can be smaller than words but larger than individual characters. This method helps in handling rare words and unknown words more effectively by splitting them into known subwords.

Example

Consider the word "unhappiness":

Word-level tokenization: The entire word "unhappiness" would be treated as a single token.

Character-level tokenization: Each character 'u', 'n', 'h', 'a', 'p', 'p', 'i', 'n', 'e', 's', 's' would be treated as a separate token.

Subword tokenization: The word can be broken down into meaningful subwords, such as "un", "happiness".

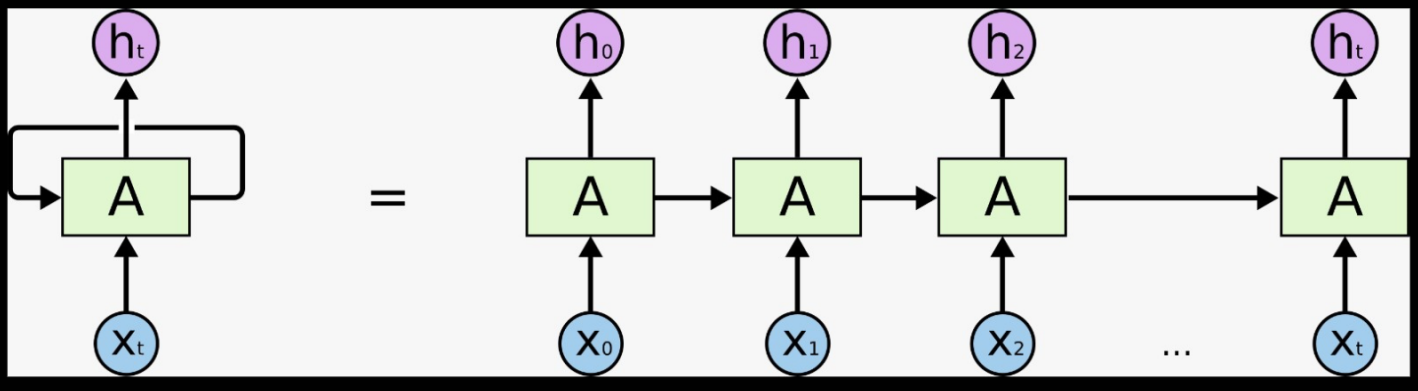
Steps in Subword Tokenization

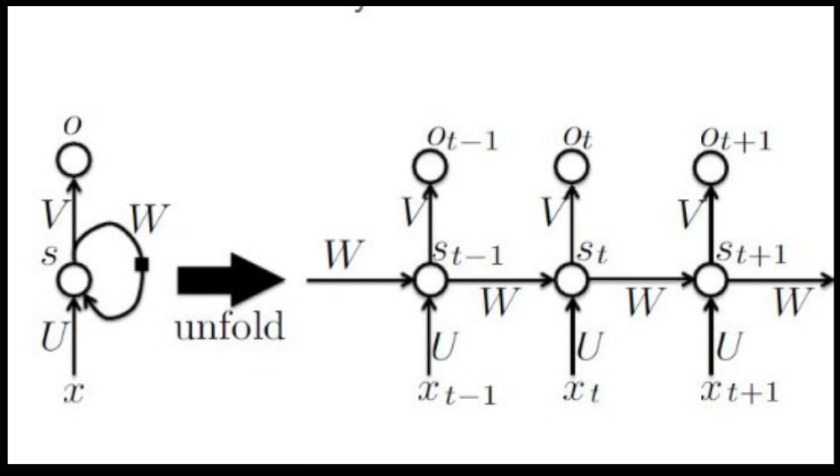
Pretraining: The tokenizer is pretrained on a large corpus to learn the most frequent subwords.

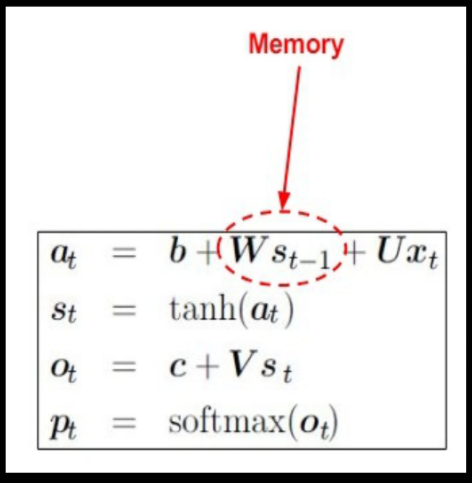
Tokenization: During tokenization, the word is broken down into the learned subwords.

Drawback of RNN's

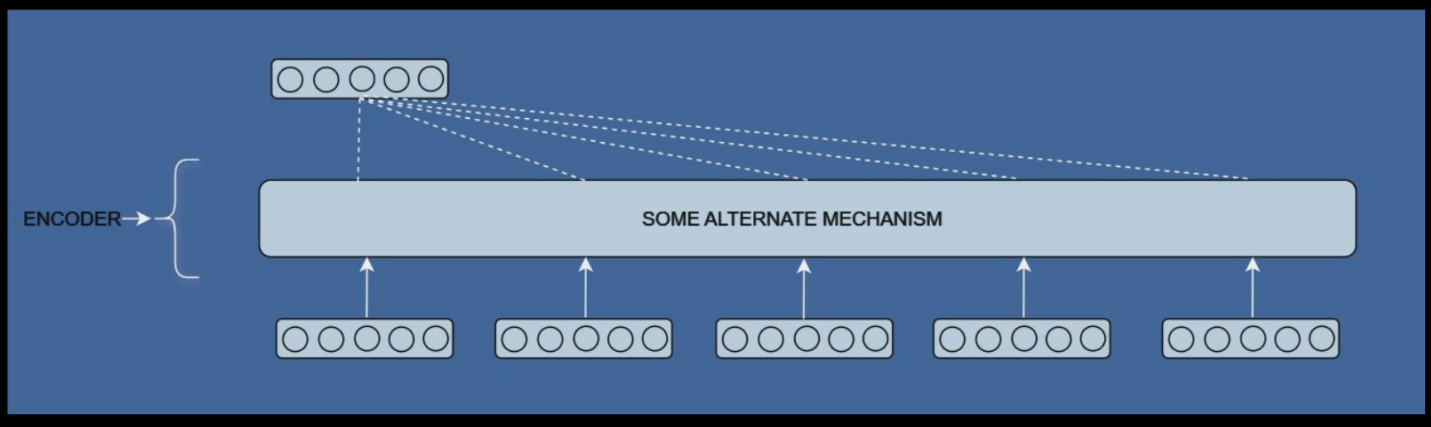
1. Parallel Processing.
2. Information loss overtime.
3. Not able to capture longer contexts.

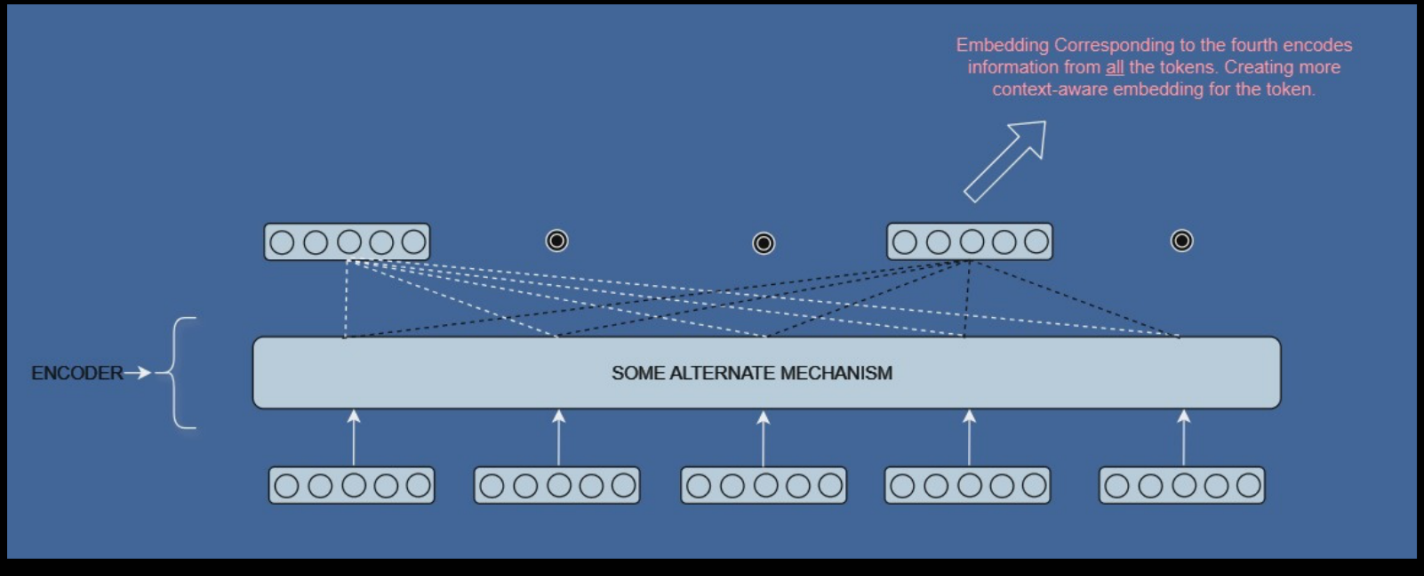






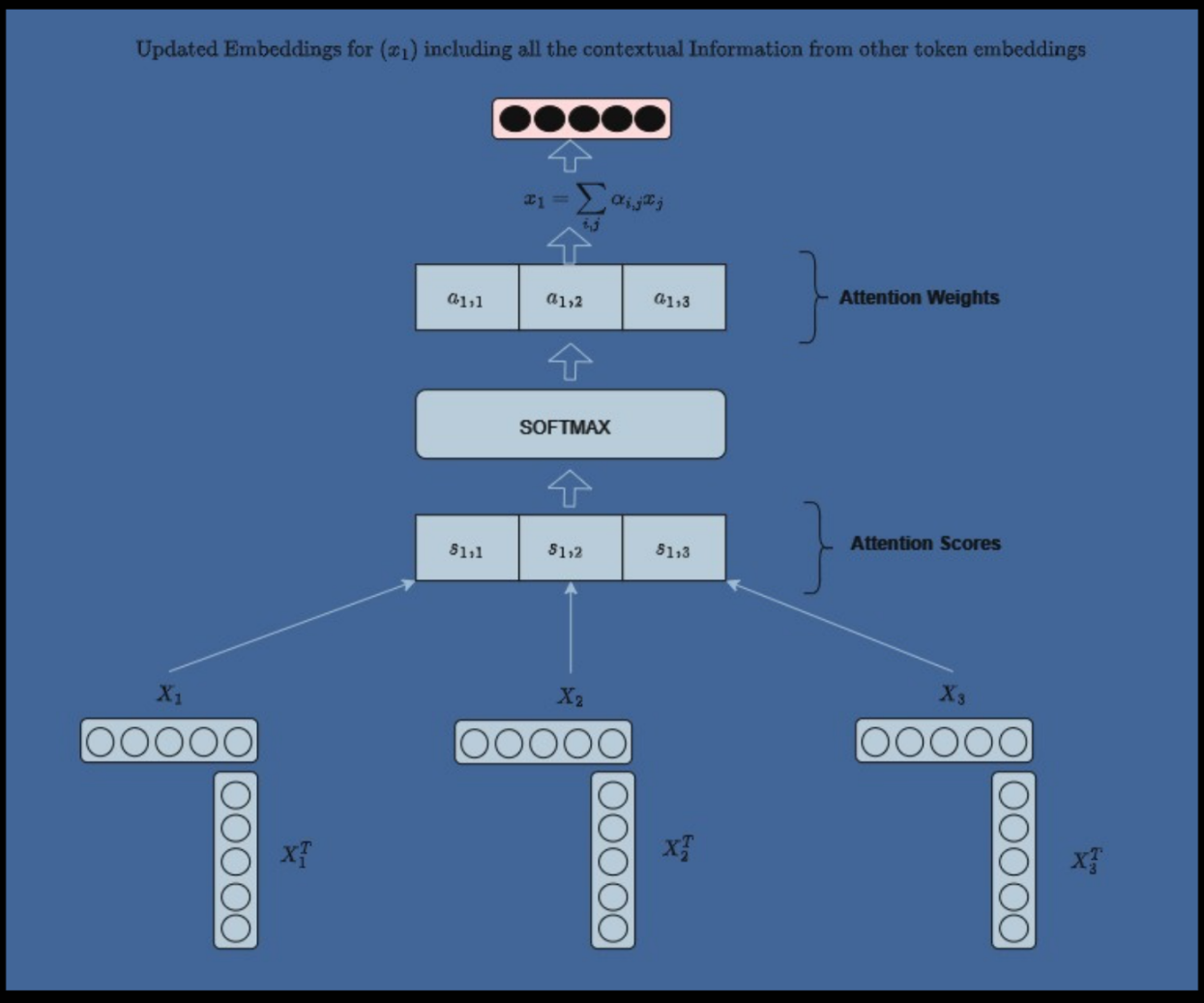
Transformer Approach



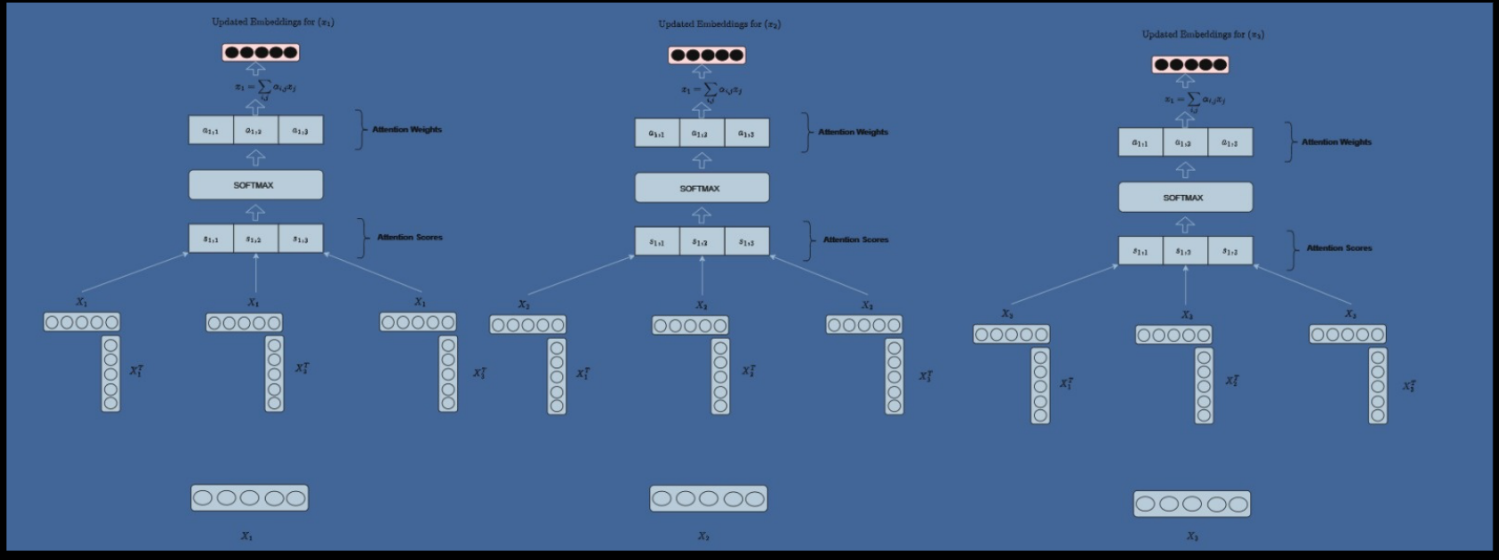


Transformer do this by having the encoder perform attention on itself i.e "Self-Attention"

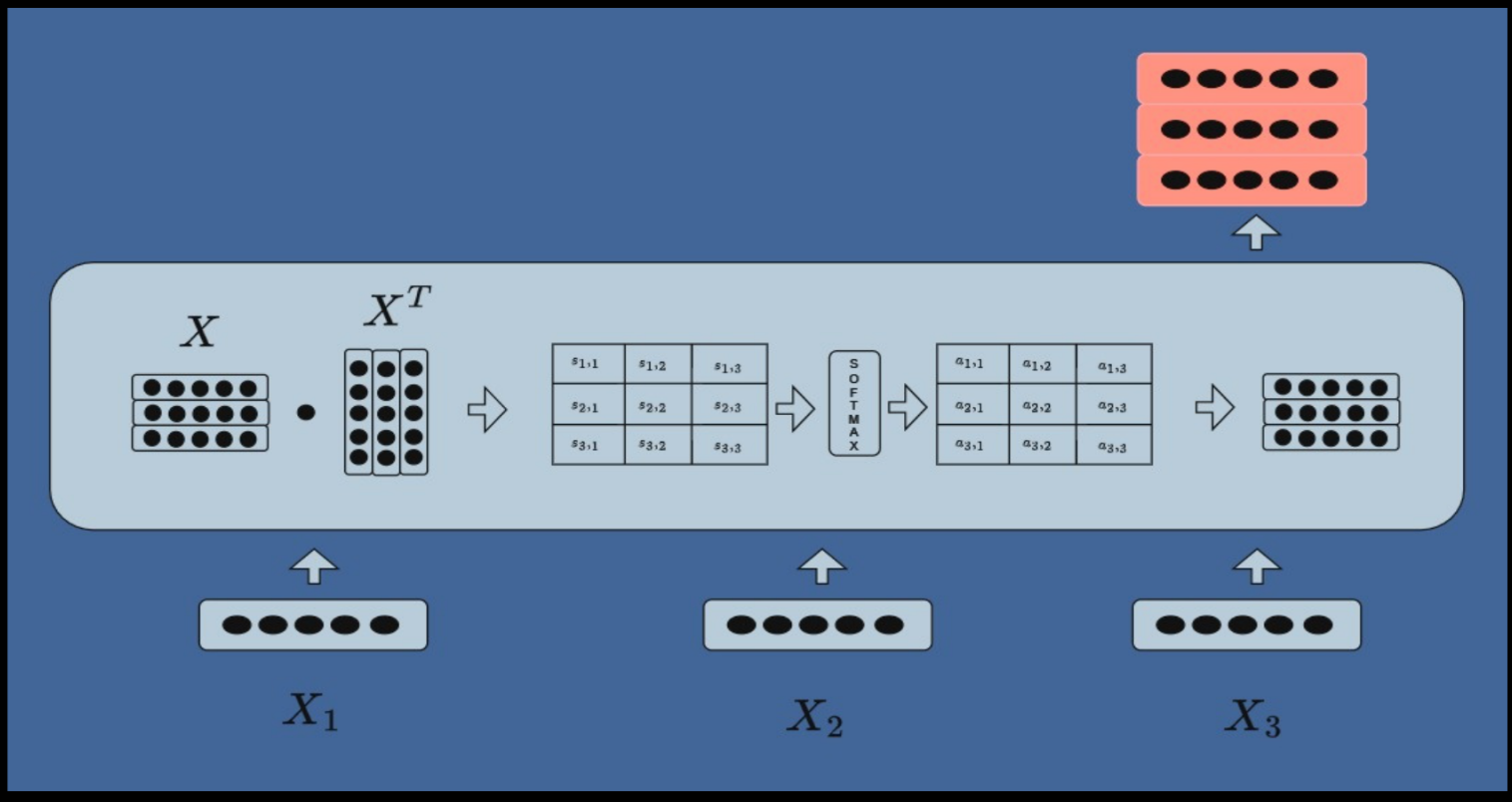
Simple Self-Attention Mechanism



For all other tokens :



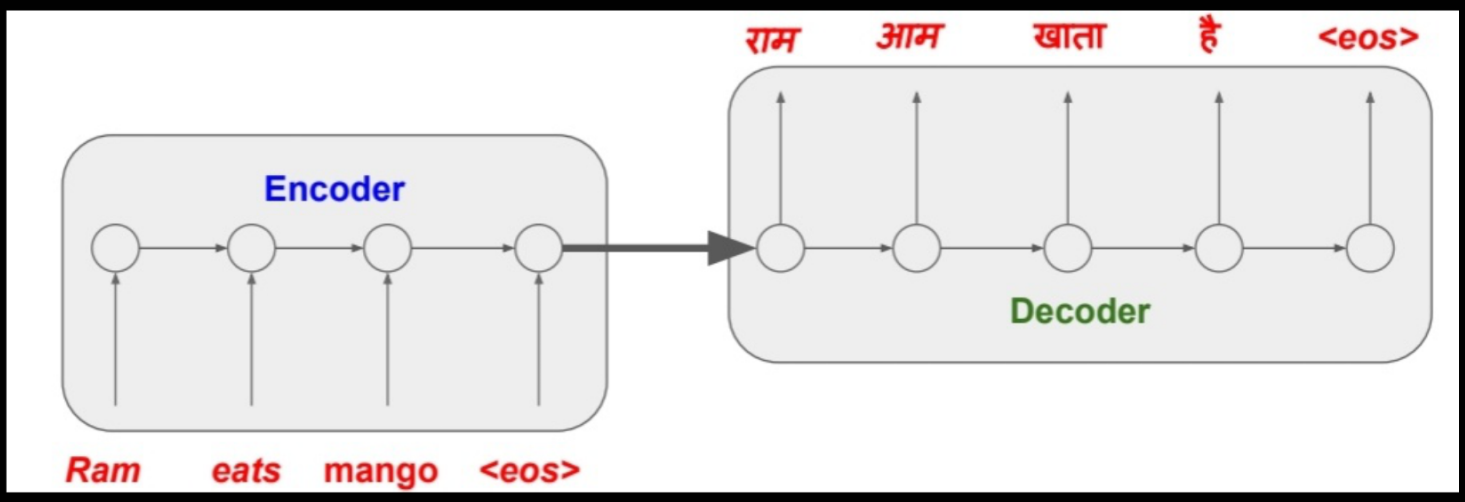
Since every output is based on all inputs , there is no need of the sequential processing like that in RNN's and all the operation's can be parallelize and all outputs can be obtained in single pass using matrices.



Since the output of the attention mechanism is same dimensions as that of inputs so it can be stacked to get better abstract representation.

Main drawback of this Approach is that, there is no learn able parameters anywhere beyond the embeddings layers.

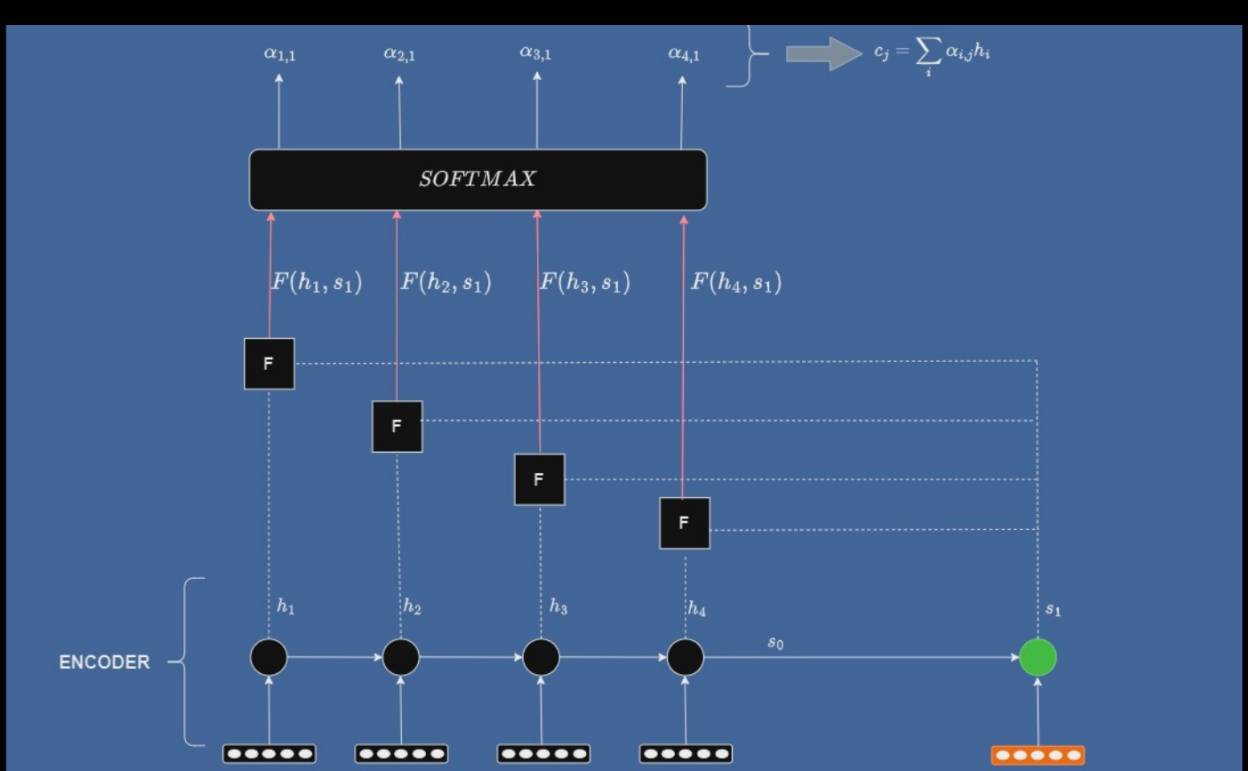
Recap of Badnaue Attention



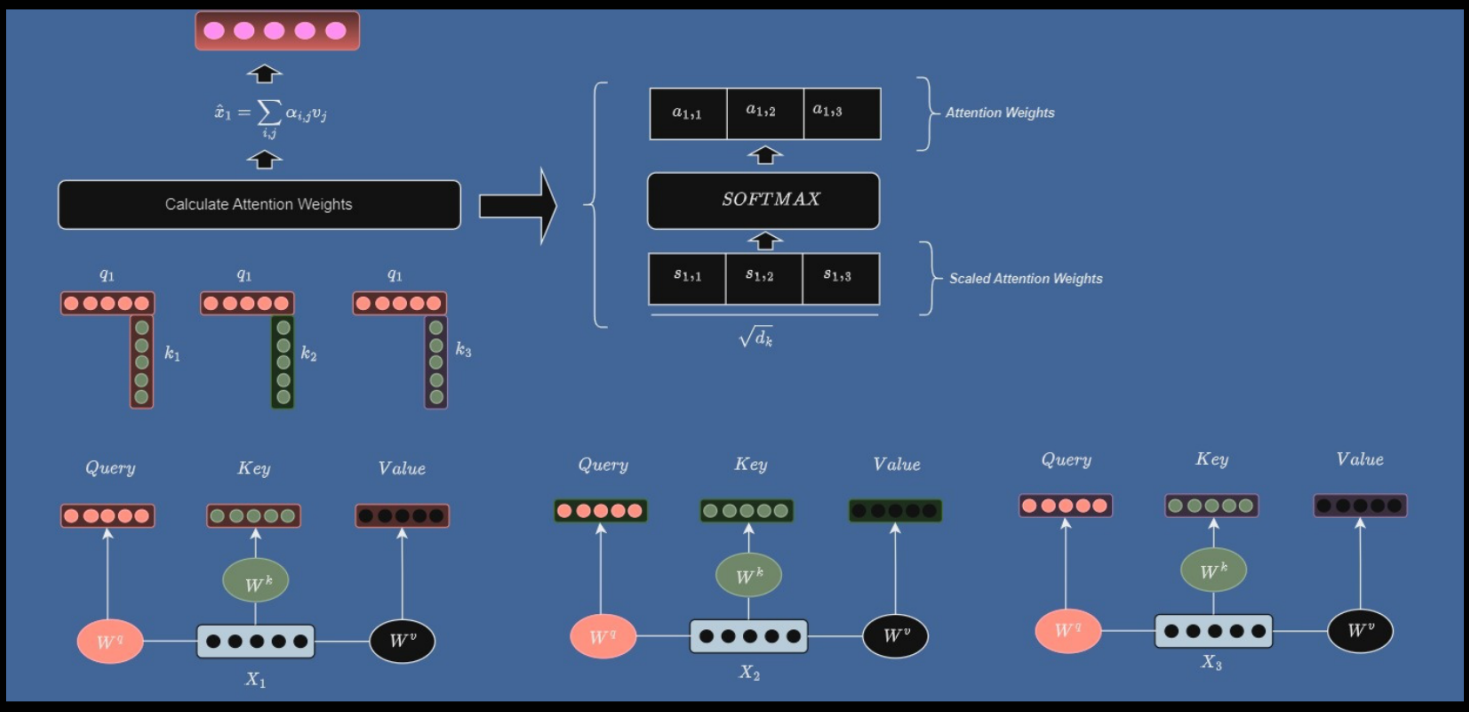
Attention as Information Retrival

Each Decoder state makes a query to the encoder which part of the sequence to focus on to generate Output at the current state.

The encoder hidden states serve as keys and values.



Same thinking leads to scaled dot product Self-Attention.



we have a sequence "a pink giant elephant gracefully wandering through the dense jungle".

Suppose that the only type of update to the embeddings that we care about is having the adjectives adjust the meanings of their corresponding nouns.

So the goal is to have a series of computation and produces a new refined set of embeddings where ,for example those corresponding to the nouns have ingested the meaning from their corresponding adjectives.

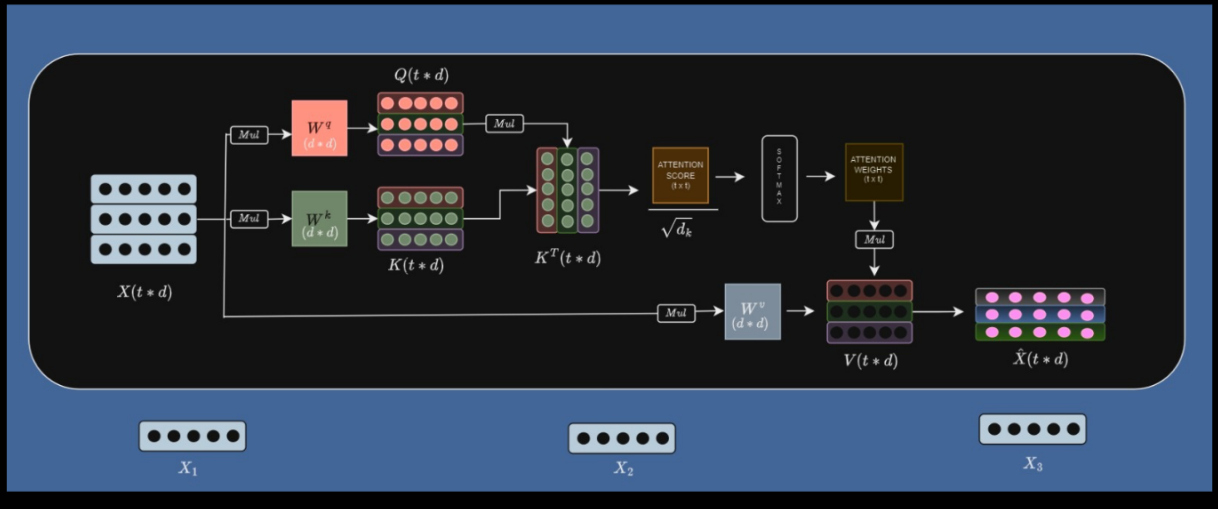
For the query step we can imagine like the elephant is asking question, is there any adjectives associated with me. This question is encoded as the query vector. By Transforming the

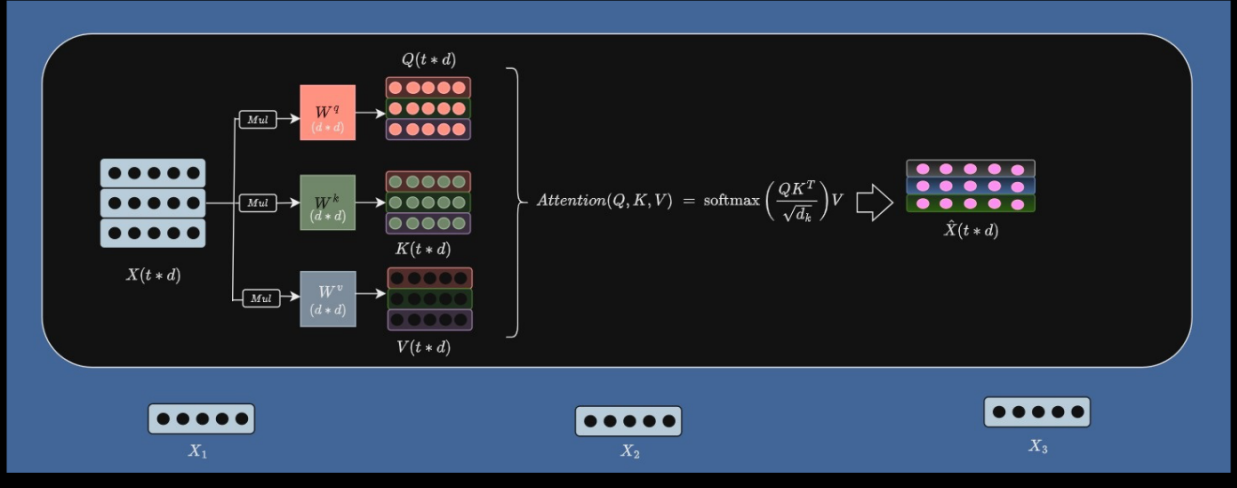
embeddings using Wq.

Similarly to get the query we generate the key vector for each token. So if the key produced by the "pink" align closely with the query produced by "elephant", then dot product of the two would be a large value.

It's called embeddings of pink attend to the embeddings of the elephant.

Instead of using the embeddings they transform the embeddings to value vector and then form the embedding of the token as a weighted linear combination of each token obtained by the dot product of query and keys vector.





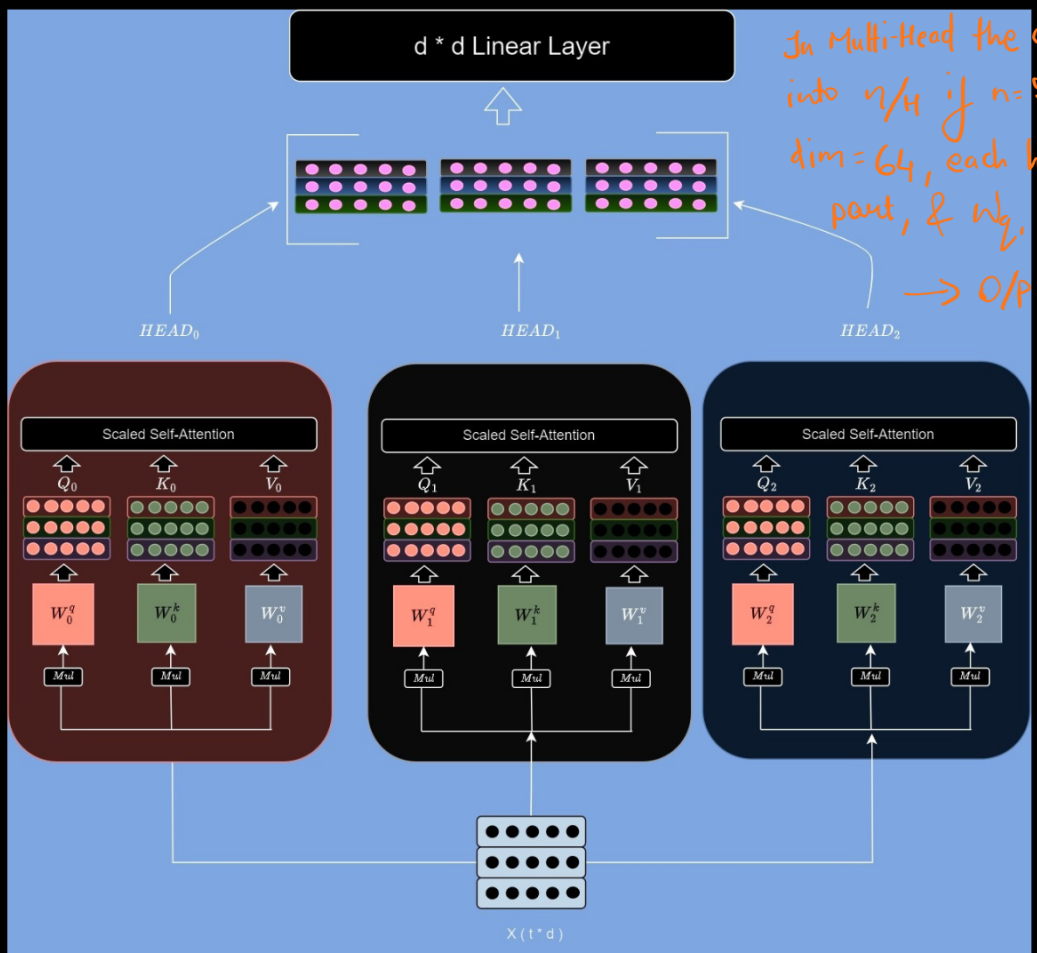
Multi-Head Attention Mechanism

Consider the sentence

" sarah went to a restaurant to meet her friend that night"

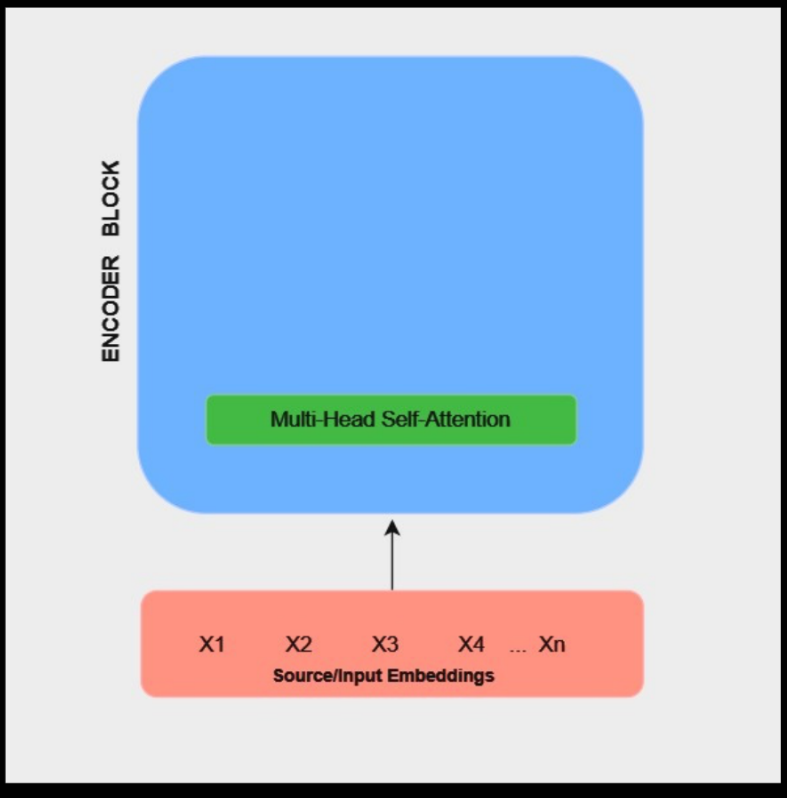
So the word "sarah" relates to other words in different ways , for example :
"what did sarah do ?" , to meet
"Where did the sarah go ?" , to a restaurant
"Who did sarah meet ?", her friend
"When did the sarah go ?" , at night. and so each

So Self-Attention focuses on only one aspect of the sentence, but there many other therefore we need to have separate Self-Attention for each aspect, that lead to Multi-Head Self Attention mechanism each head focus on one such aspect.



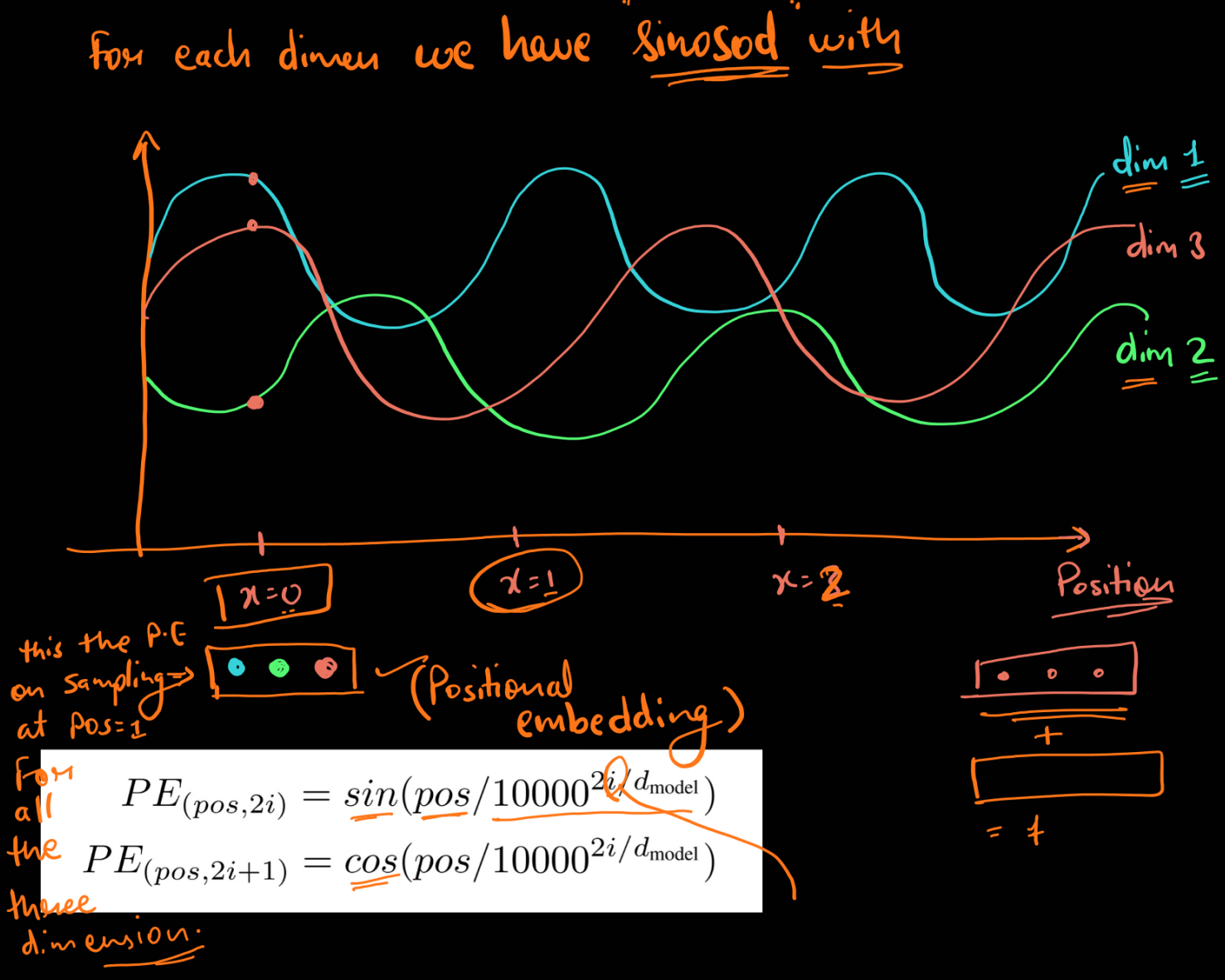
Since each head operates in a smaller dimension of d/h, the overall computation across multiple heads isn't higher than. Single-head attention.

Also each head can be run in parallel.



So right now our encoder looks like above. But we have an issue here. Changing the position of the input words would results in the same corresponding output. In RNN's we have the word-order information as the sequence is processed one by one, but here we don't. Therefore we need to infuse this information in the embeddings.

Positional Embeddings



For all the three dimension:

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

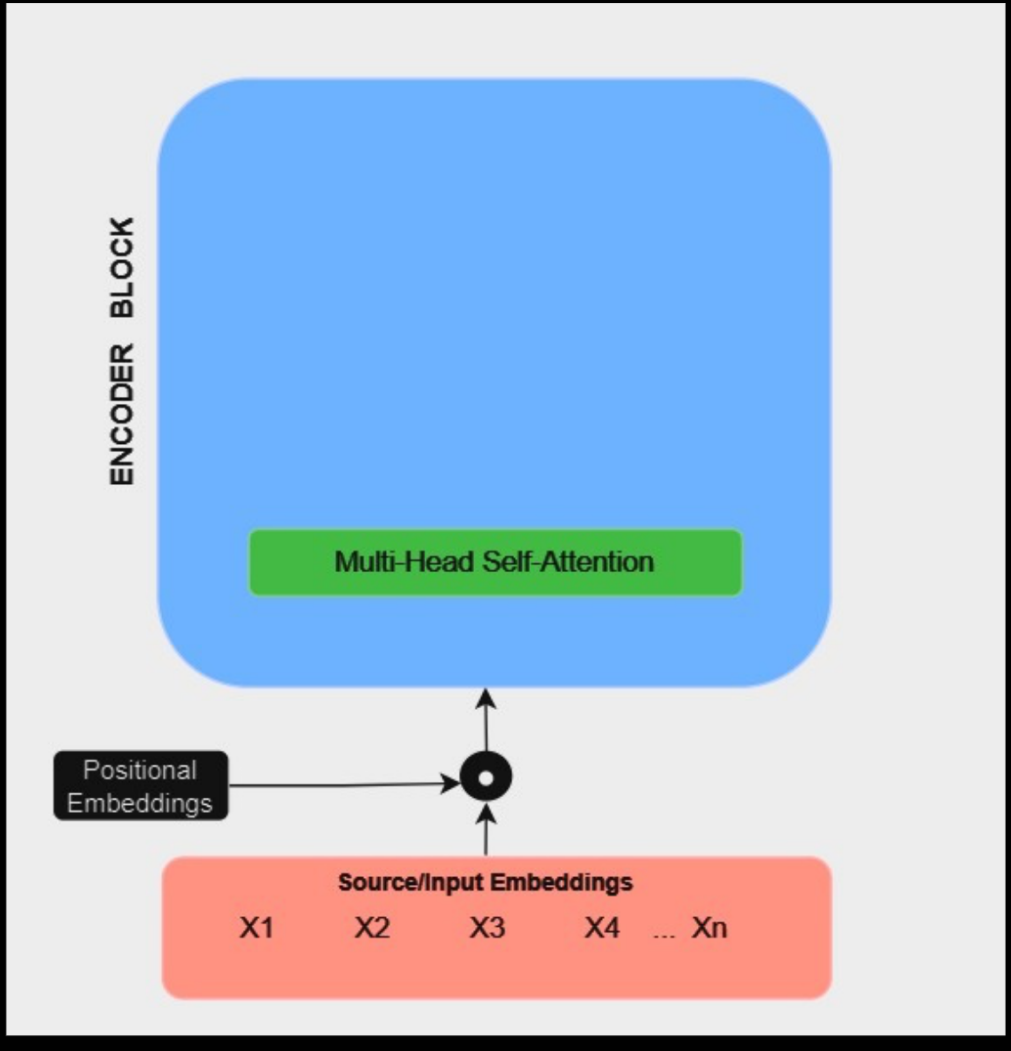
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Where pos is the position of the word in the sequence , i is the dimensions index, d_model is the dimensions of the model.(embedding size)

The formula uses alternating sine and cosine functions for each dimension of the positional encoding. For even indices, the sine function is used, and for odd indices, the cosine function is used.

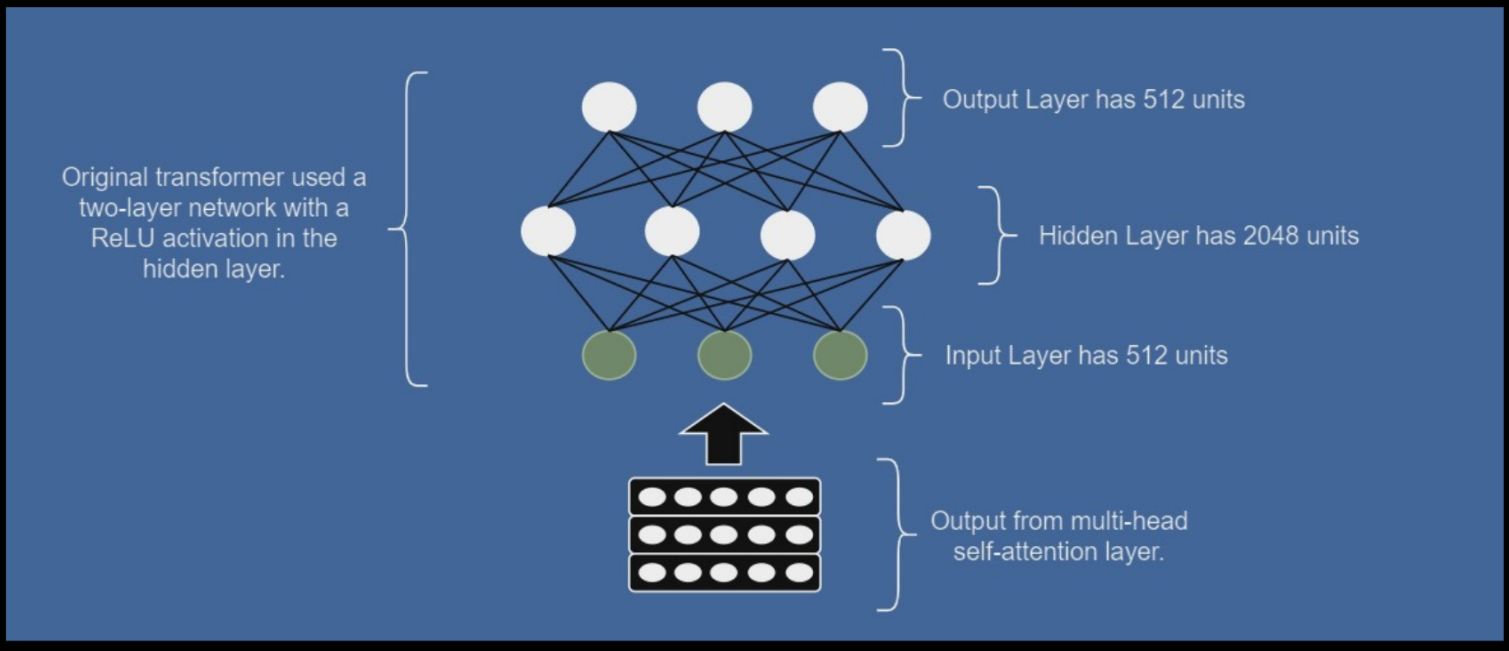
Why This Formula?

1 Distinct Values: Using different sine and cosine functions ensures that the positional encodings for different positions are distinct.
2 Scale Invariance: The division by
ensures that the frequencies of the sine and cosine functions are different for each dimension, allowing the model to learn to attend to different positions in the sequence.
3 Smooth Variation: The use of the sine and cosine functions ensures that positional encodings vary smoothly, which helps the model to generalize better over different sequence lengths.

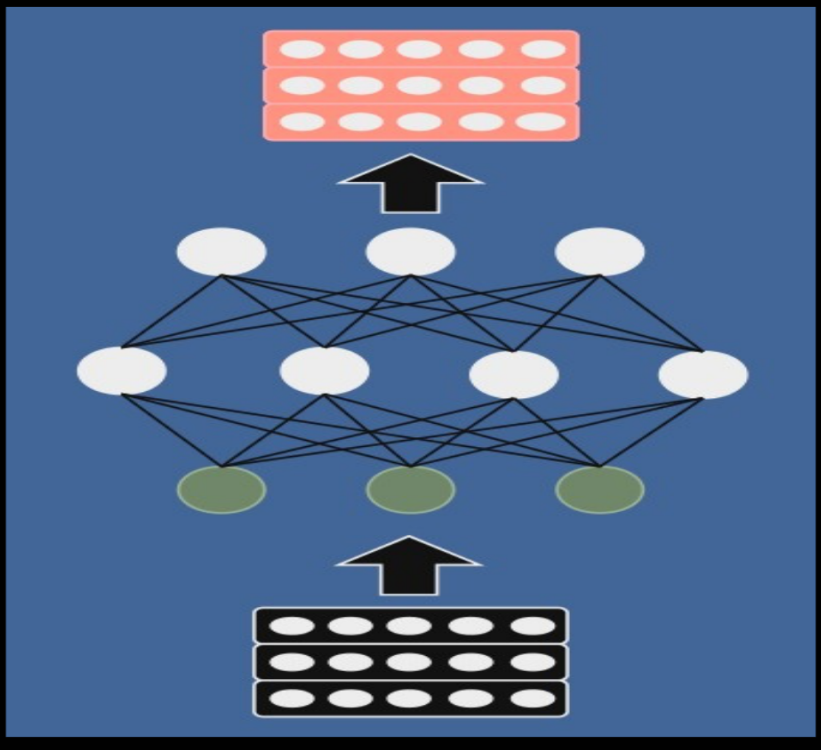


So after adding positional Embeddings our encoder looks like above. But still we have a problem.Till now we haven't introduced the non-linearity, all transformation in the Self-Attention, Multi-Head Attention are linear transformation.

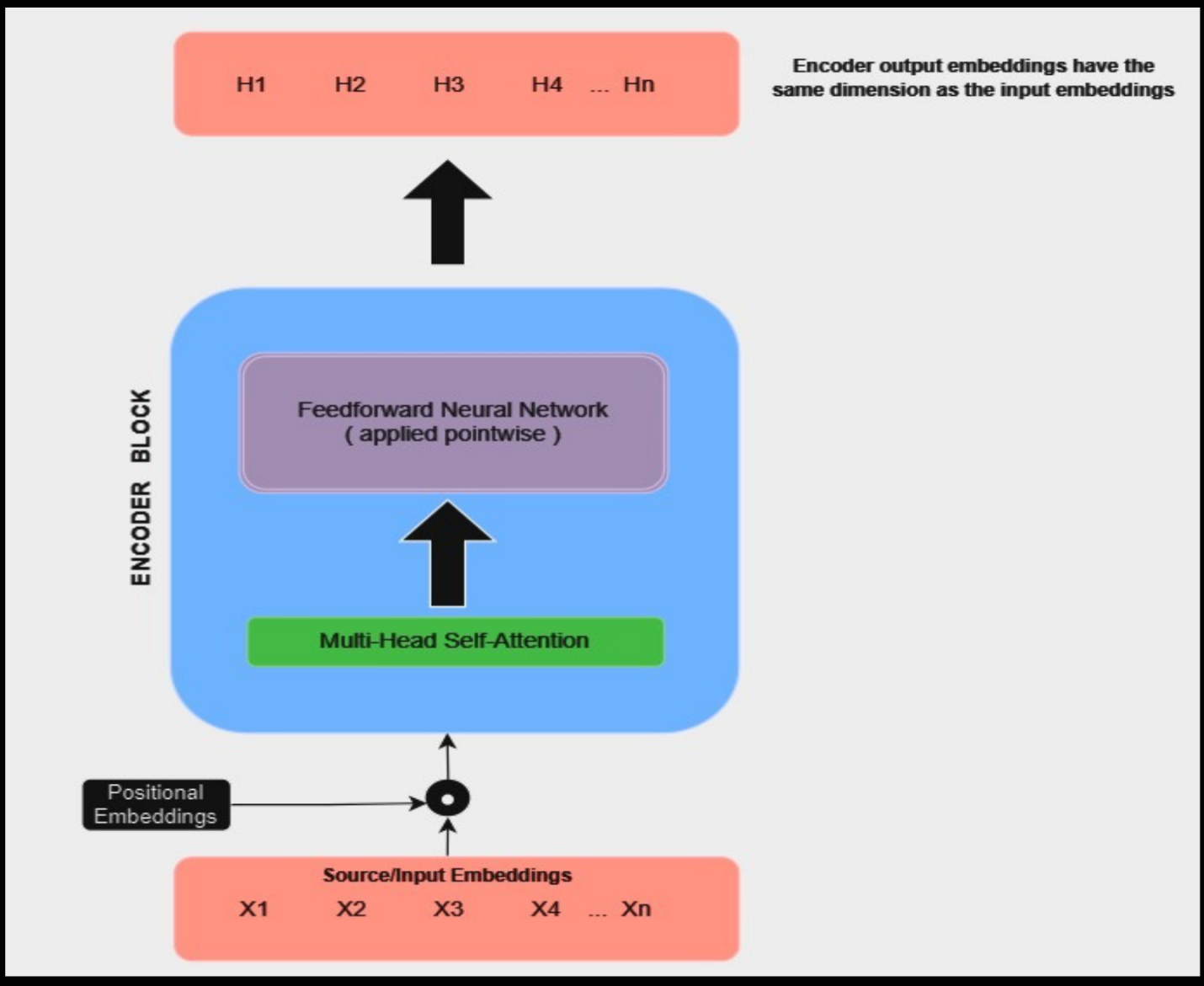
To do so the author introduced a linear layer with hidden layer having ReLU activation function.



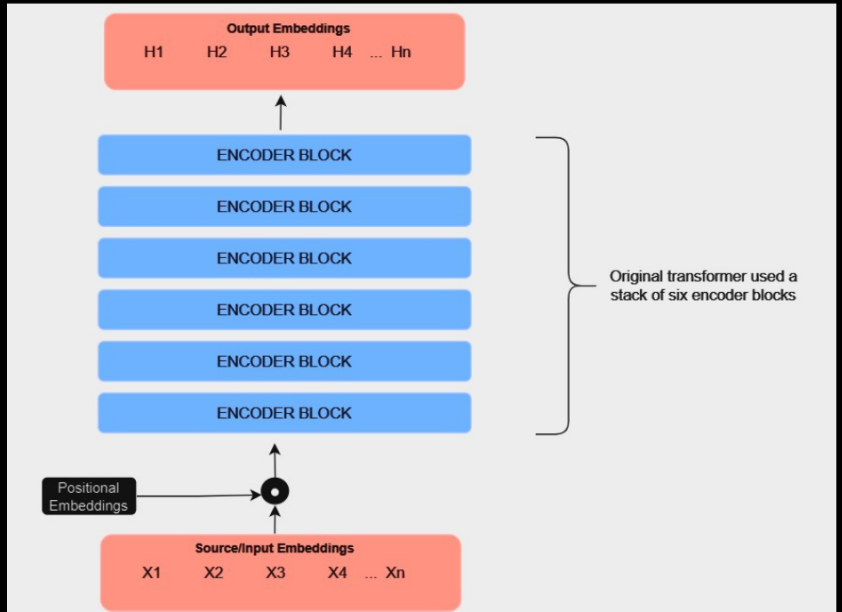
Network is applied point-wise.



So after introducing linear layer our encoder looks like



FF layer will produce the output dimensions of 512. Since the dimension is maintained after all the operation's we can stack encoders to get a more refined representation of



inputs.

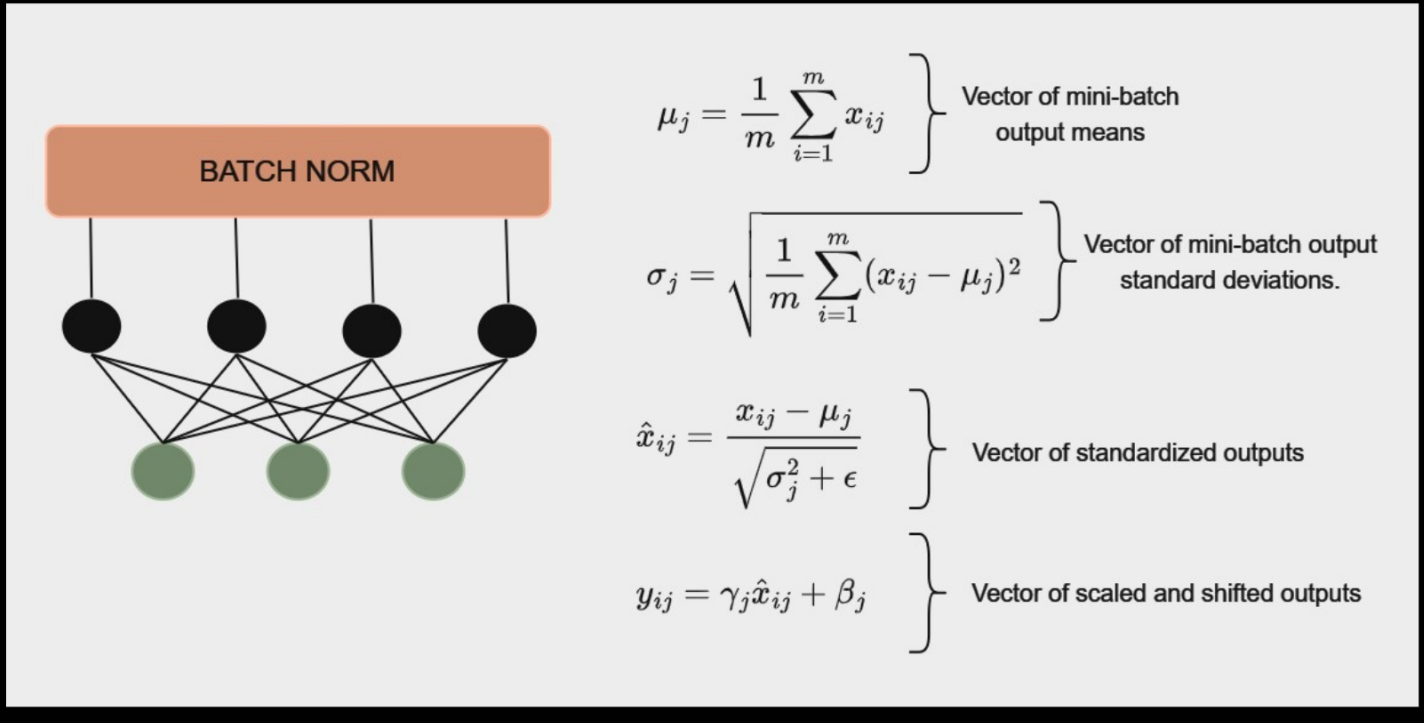
In Paper they have used a stack of 6 encoder blocks.

Due to such an arrangements we face two problems. First is the shifting inputs from earlier encoder blocks add noise and as the network becomes deeper vanishing gradients problem is faced.

So they solved the above problem using Normalization and skip connection inspired by Resnet.

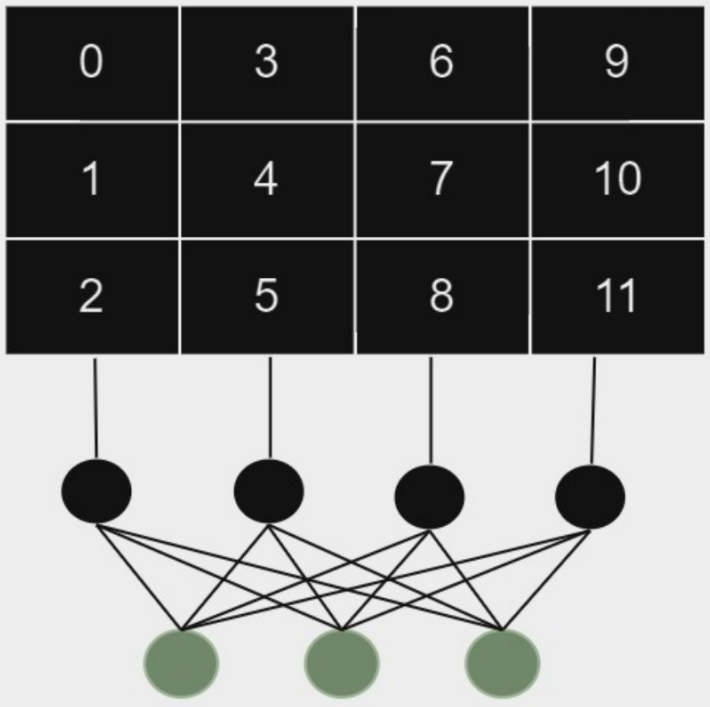
Normalization

First we look into our old Normalization technique that is Batch Normalization.



But there is an issue with batch normalization, it's not suitable for variable length sequence, dependent on batch size and can't be parallelize.

Layer Normalization



$$\mu = \frac{1}{d_H}\sum_{i=1}^{d_H} x_i$$

$$\sigma = \sqrt{\frac{1}{d_H}\sum_{i=1}^{d_H}(x_i - \mu)^2}$$

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

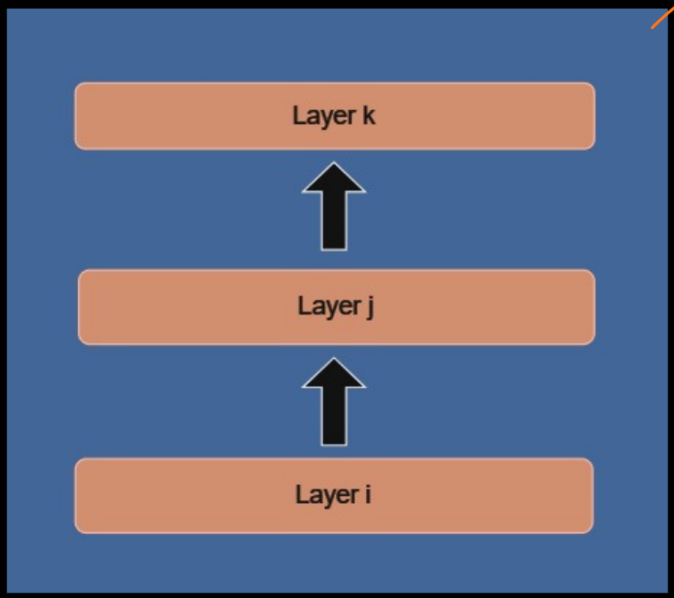
$$z = \gamma\hat{x} + \beta$$

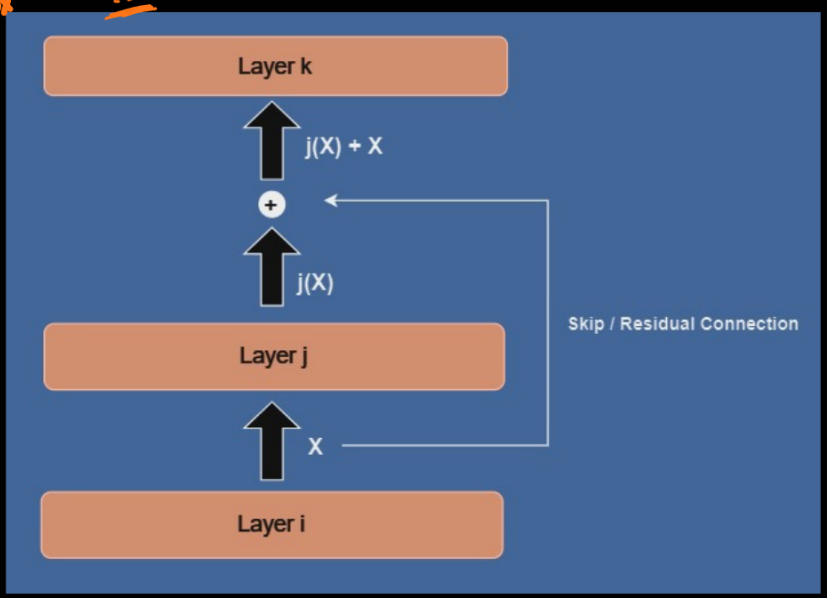
Skip - Connection

$L = j(x) + x$

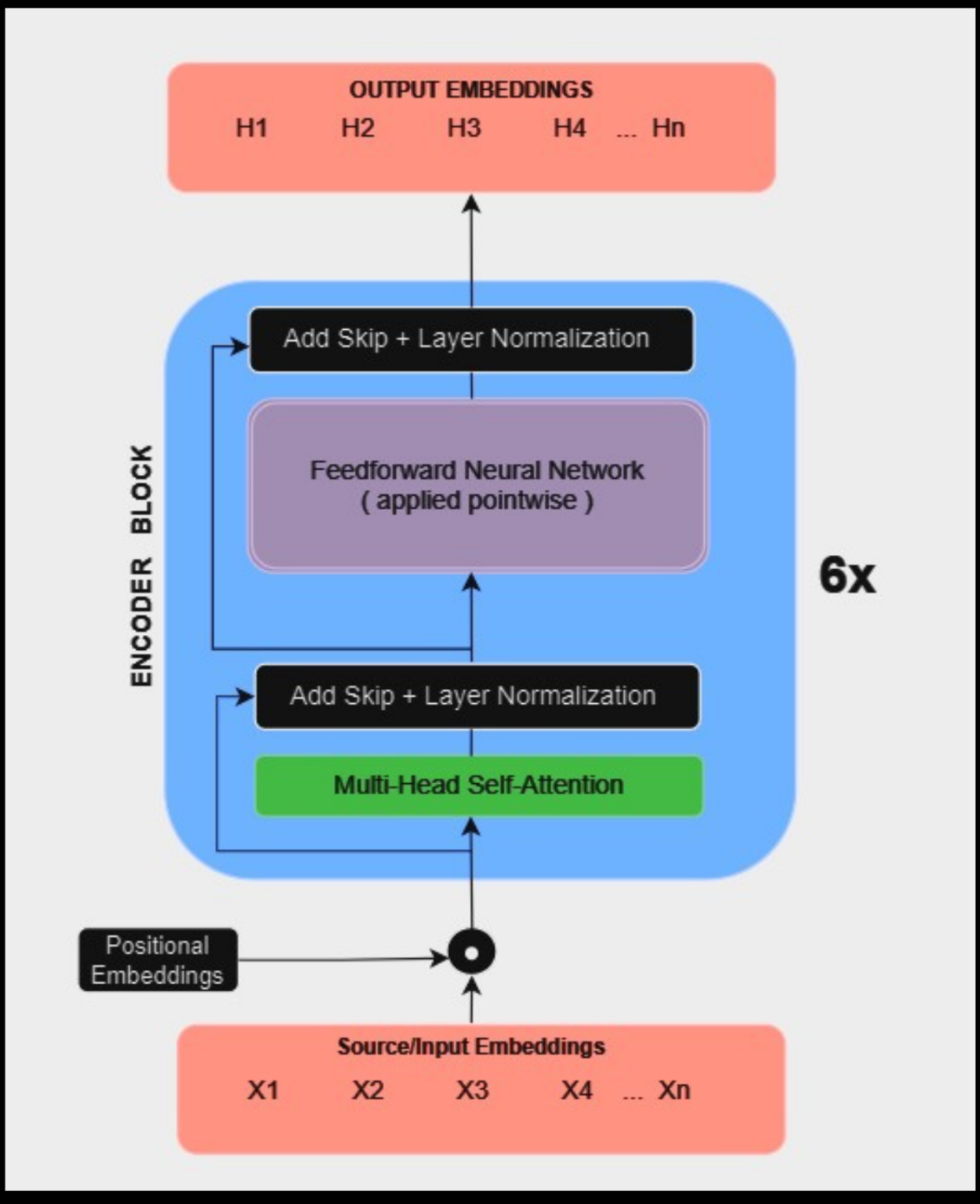
$\frac{\partial L}{\partial x} = \frac{\partial (j(x) + x)}{\partial x}$

$= \frac{\partial j(x)}{\partial x} + 1$



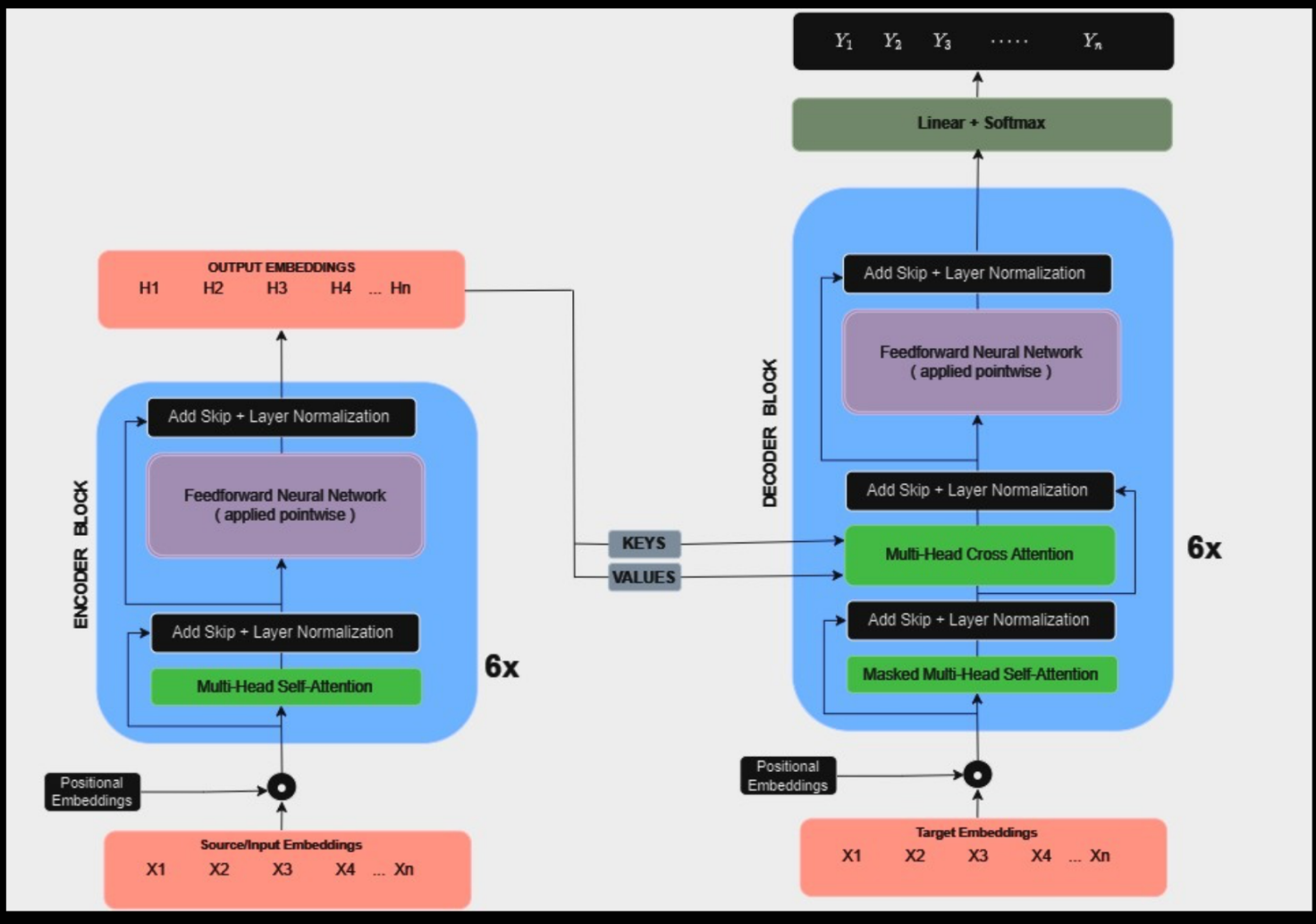


After all this our encoder looks like this.



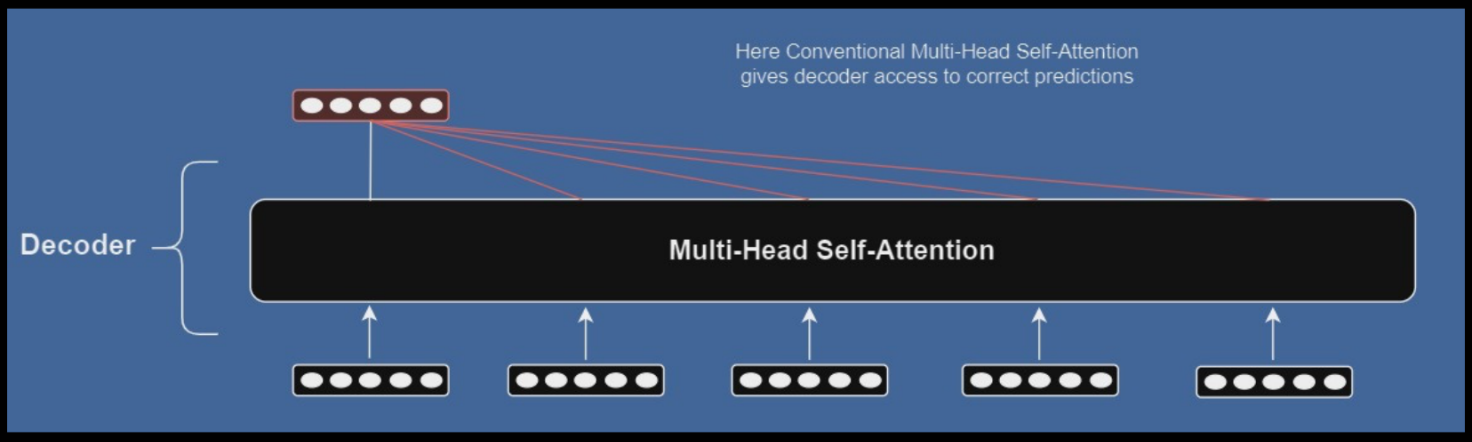
The output embeddings are contextualized. So the word "rock" will be different embedding depending on whether input was "rock concert" or "rock climbing".

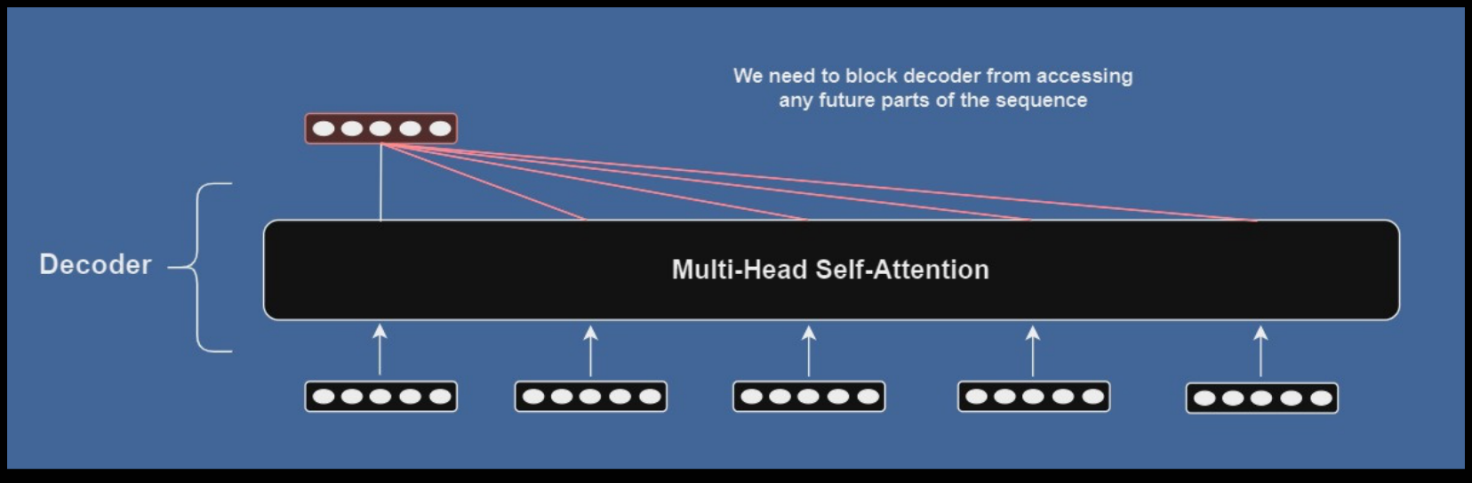
Now let's focus on the decoder part of the transformer.



Here in Decoder we uses Masked Multi-Head Attention which is one of the differences between encoder and decoder.

Masked Multi-Head Attention





Algorithm

1. **Compute Query, Key, and Value Matrices:**

   Given an input sequence X, compute the query, key, and value matrices:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

   where W_Q, W_K, W_V are learnable weight matrices.

2. **Compute Scaled Dot-Product Attention:**

   Compute the attention scores:

$$\text{scores} = \frac{QK^T}{\sqrt{d_k}}$$

   where d_k is the dimension of the key vectors.

3. **Apply Mask to the Scores:**

   Create a mask matrix M such that positions that should not be attended to (future positions) are set to $-\infty$. This is typically done using an upper triangular matrix:

$$M_{ij} = \begin{cases} 0, & \text{if } j \leq i \\ -\infty, & \text{if } j > i \end{cases}$$

   Add the mask to the attention scores:

$$\text{masked\_scores} = \text{scores} + M$$

4. **Apply Softmax to Get Attention Weights:**

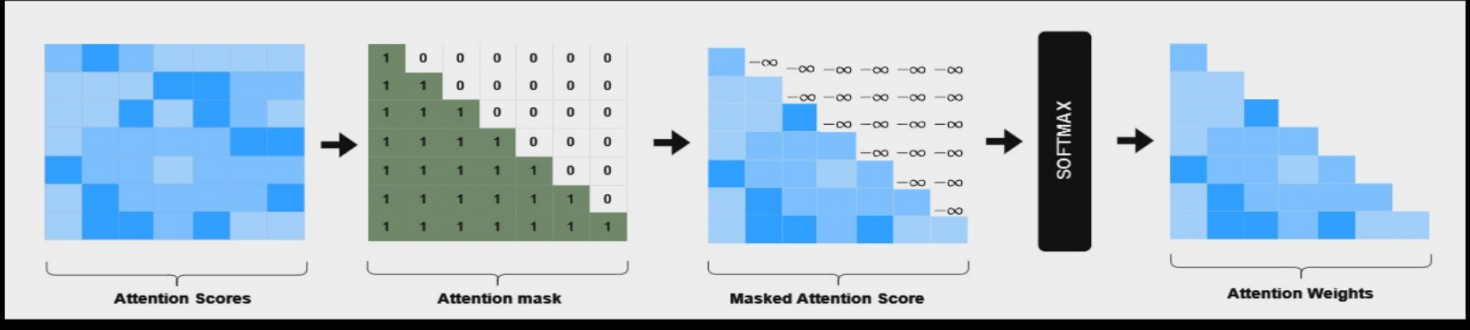
   Apply the softmax function to the masked scores to obtain the attention weights:

$$\text{weights} = \text{softmax}(\text{masked\_scores})$$

5. **Compute the Output:**

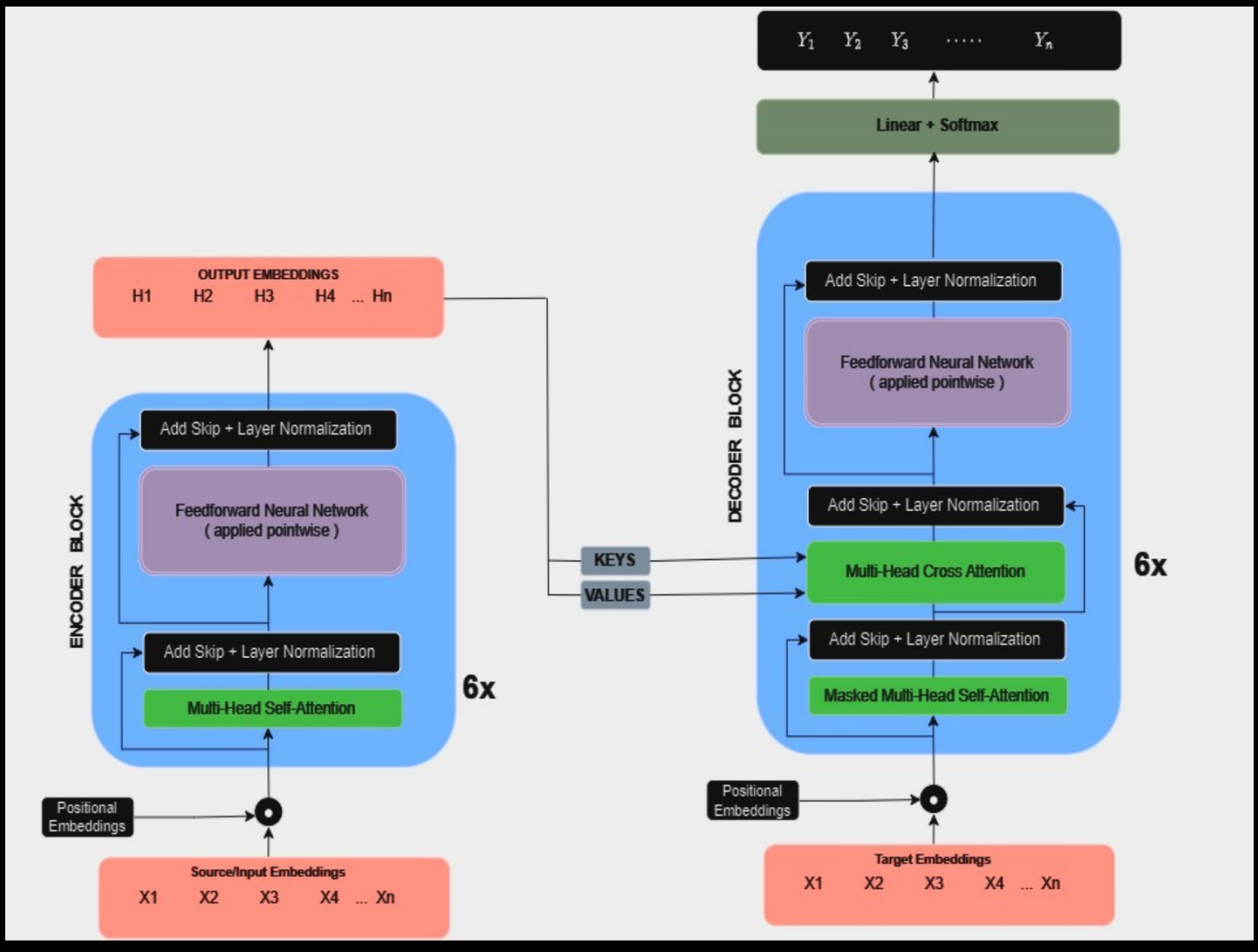
   Compute the weighted sum of the value vectors:

$$\text{output} = \text{weights} \cdot V$$



Cross-Attention

Refer to the attention mechanism between the decoder and the encoder in the model.



Algorithm

Input to the Decoder: The decoder receives the encoded representations (output of the encoder) of the input sequence. Let's denote these encoded representations as $H^{\text{enc}} = \{h_1^{\text{enc}}, h_2^{\text{enc}}, ..., h_n^{\text{enc}}\}$, where n is the length of the input sequence.

Compute Queries, Keys, and Values:

- **Query (from Decoder):** For each position t in the output sequence, compute a query vector q_t using a learned weight matrix W^Q:

$$q_t = h_t^{\text{dec}} W^Q$$

  Here, h_t^{dec} is the hidden state of the decoder at position t.

- **Keys and Values (from Encoder):** Compute key vectors k_i^{enc} and value vectors v_i^{enc} for each position i in the input sequence using learned weight matrices W^K and W^V:

$$k_i^{\text{enc}} = h_i^{\text{enc}} W^K, \quad v_i^{\text{enc}} = h_i^{\text{enc}} W^V$$

  Here, h_i^{enc} is the encoded representation of the input sequence at position i.

Compute Attention Scores: For each query q_t, compute attention scores with respect to all keys $\{k_i^{\text{enc}}\}$:

$$\text{score}_t(i) = \frac{q_t k_i^{\text{enc}T}}{\sqrt{d_k}}$$

where d_k is the dimension of the key vectors.

Apply Softmax: Apply the softmax function to obtain attention weights $\alpha_t(i)$:

$$\alpha_t(i) = \text{softmax}(\text{score}_t(i))$$

Compute Weighted Sum: Compute the weighted sum of the value vectors $\{v_i^{\text{enc}}\}$ using the attention weights $\{\alpha_t(i)\}$:

$$c_t^{\text{enc}} = \sum_i \alpha_t(i) v_i^{\text{enc}}$$

This weighted sum c_t^{enc} represents the context vector for the decoder at position t, which captures the relevant information from the input sequence.

Cross-attention in Transformer models enables the decoder to selectively focus on different parts of the input sequence (encoded representations) when generating the output sequence. This mechanism allows the model to effectively learn dependencies between input and output sequences in

tasks such as machine translation,

Similar to encoder we have 6 stacked decoder also and the output of the final decoder goes to a linear layer and the to the softmax to generate the probability distribution over the vocabulary. We select the max one as our prediction.