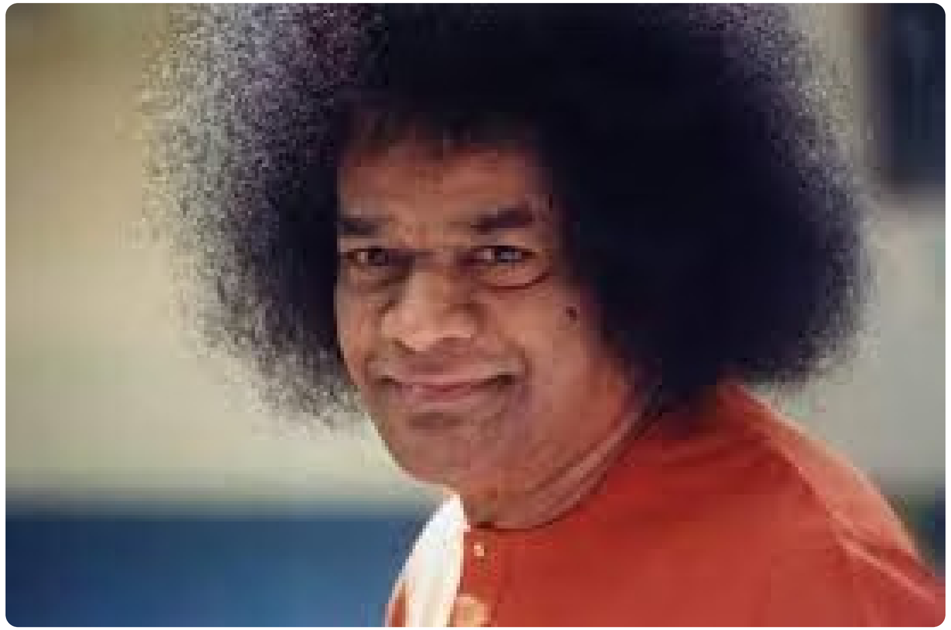
Transformers - Attention is All You need!

by Vishwanath Shetiya

II M.Sc Mathematics with a specialization in Computer Science.



Offering humble Pranams at Swami's Lotus Feet

Table of contents:

What is a Transformer?

- A Transformer is a neural network architecture that takes an input sequence and outputs a sequence.
- It was introduced in 2017.
- Transformers are used for various **natural language processing** (NLP) tasks.
- They are now also being used for image classification, such as in **Vision Transformer**.
- Transformers use a self-attention mechanism.

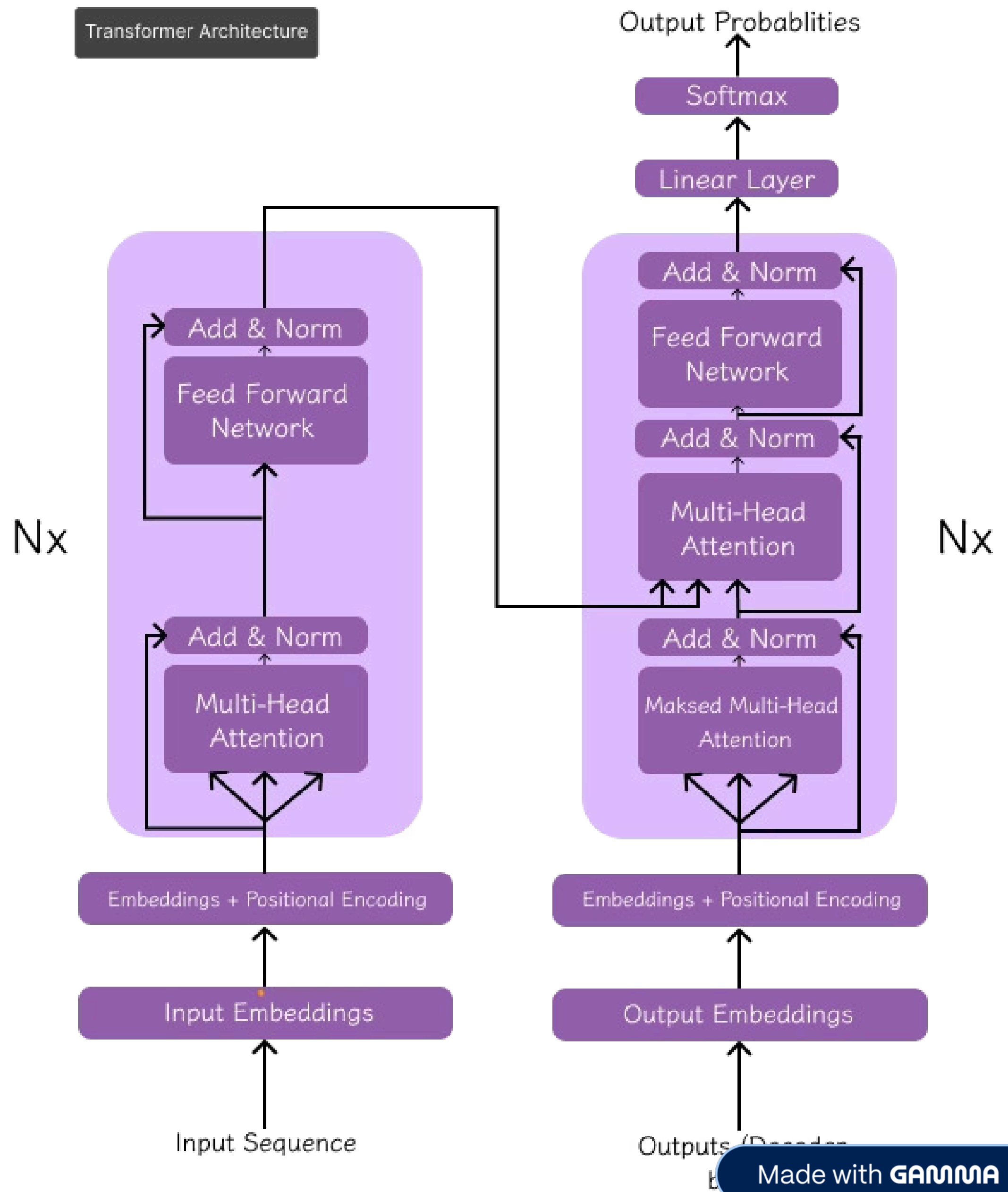
Why Transformers?

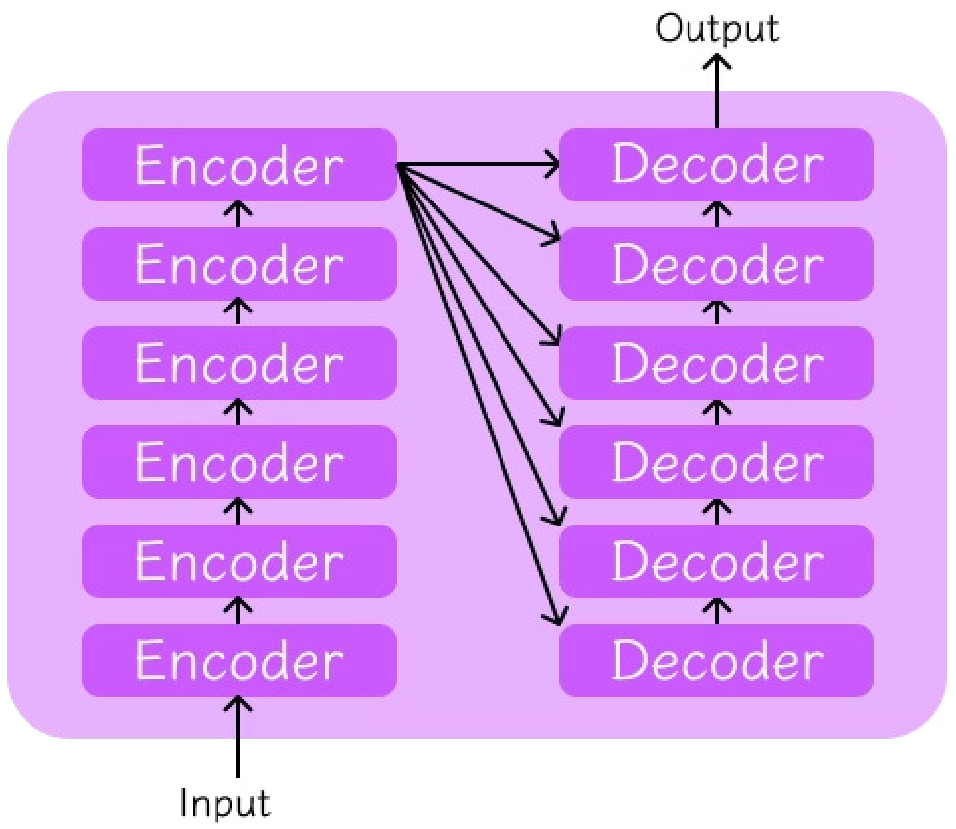
Problems with Recurrent Neural Networks (RNNs):

- Lack of parallelization causes slower training.
- Issues with long-range dependencies:
  - The hidden state is more influenced by the previous time step than earlier time steps.
  - Longer sequences can lead to vanishing gradient.
- Long Short-Term Memory (LSTM) helped to overcome the problem of long-range dependencies with the help of memory cells and gates, but still took the input sequentially.

Transformer Architecture

- It has Encoder block (left) and a Decoder block (right).
- The Encoder is made up of sublayers such as Multi-head Attention, a layer for Normalization, a Feedforward Network, and Residual Connections.
- The Decoder has Masked Multi-head Attention, a Layer Normalization, a Feedforward Network and Residual Connections.
- Other than these blocks, it also has an Embeddings layer, linear layer, softmax and Positional Encoding.

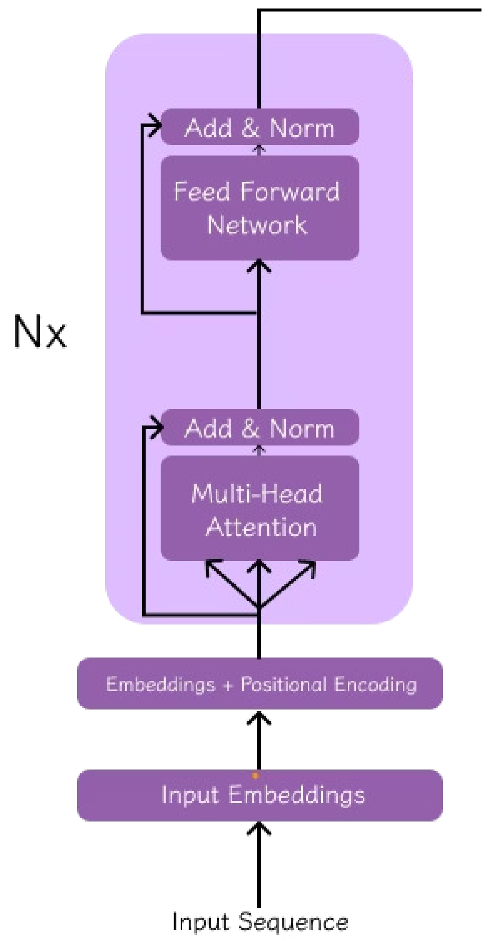




Transformer Architecture

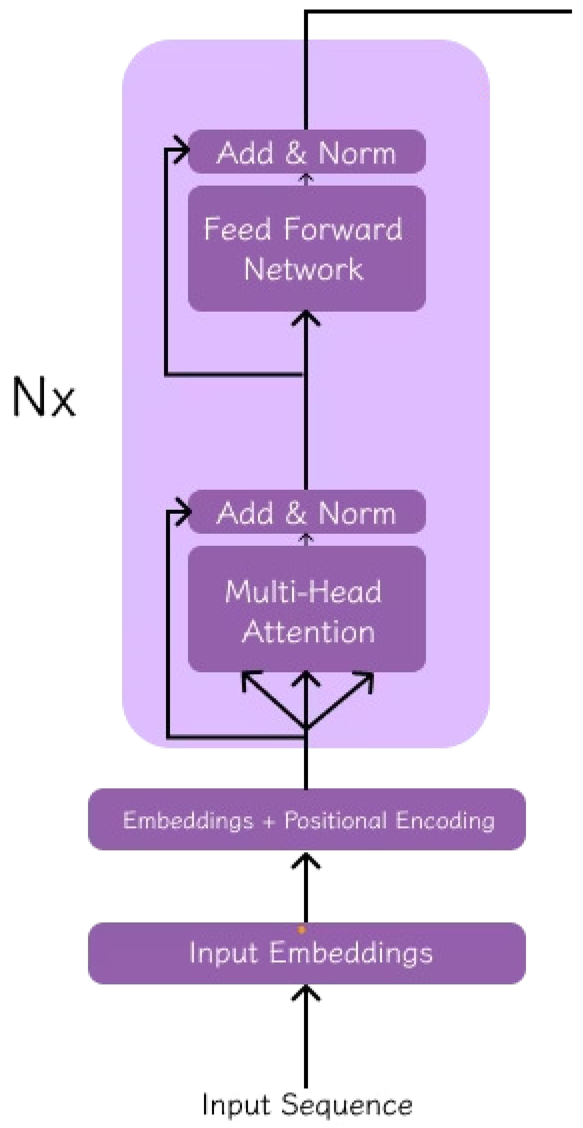
- The model that uses the transformer architecture looks like this.
- The encoder stack is made up of encoder blocks, and the decoder stack is made up of decoder blocks.
- For example, in the diagram, N is 6, which implies the model has an encoder and a decoder stack with 6 blocks each.

Encoder Block



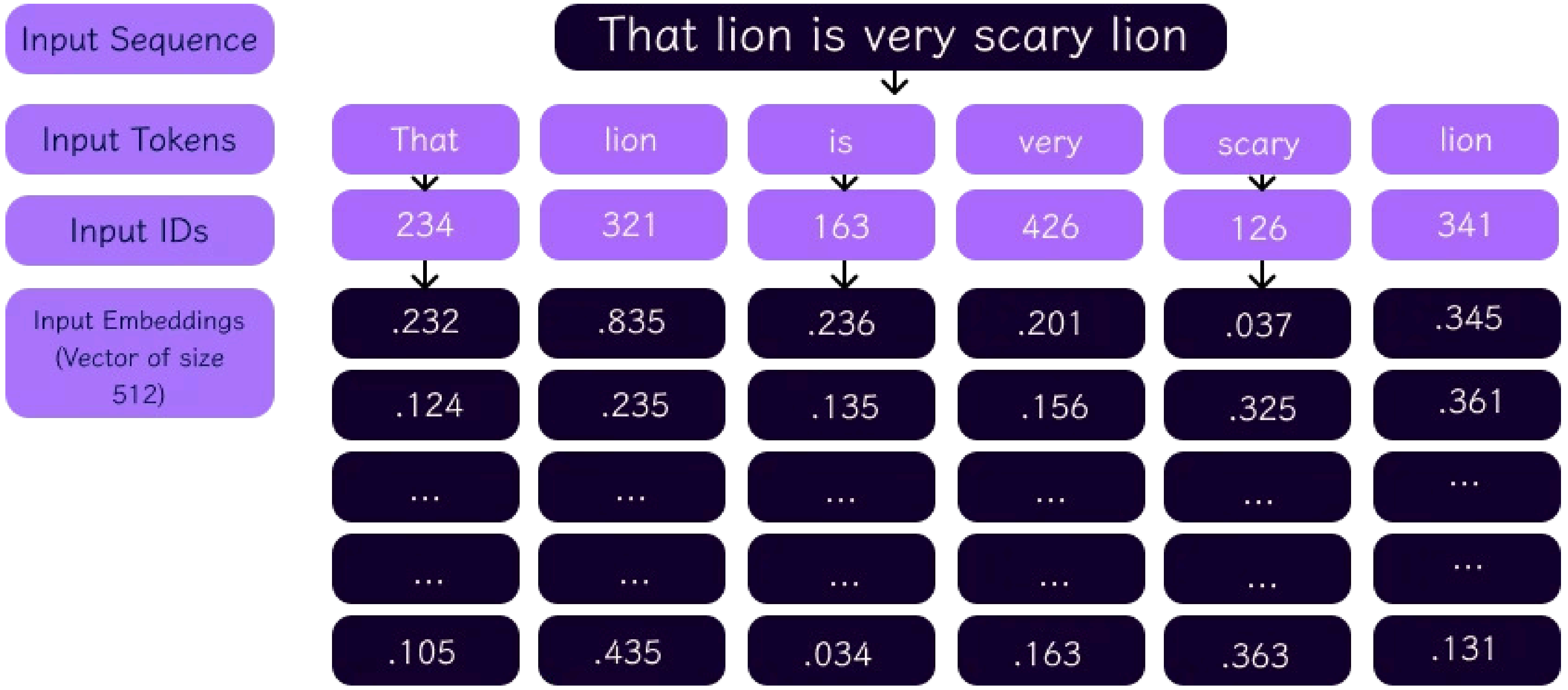
Input Embeddings

- It is the first layer through which the input to the transformer is processed in the manner the architecture understands.
- The models that use this architecture use their own vocabulary that helps in mapping a word or a part of a sequence to a vector (embedding) in the embedding space of that vocabulary.
- A part of a sequence, also known as a token, is some index in the vocabulary, and that ID is mapped to an embedding vector in the embedding space of the vocabulary.



Made with GAMMA

Input Embeddings



Input Sequence	That lion is very scary lion					
Input Tokens	That	lion	is	very	scary	lion
Input IDs	234	321	163	426	126	341
Input Embeddings (Vector of size 512)	.232	.835	.236	.201	.037	.345
	.124	.235	.135	.156	.325	.361
	...	...	...	...	...	...
	...	...	...	...	...	...
	.105	.435	.034	.163	.363	.131

Positional Encoding

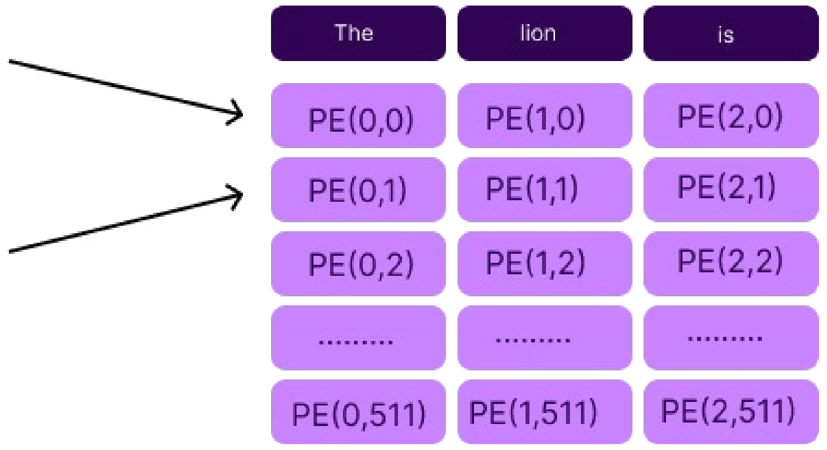
- From the above embeddings, we would have only captured the meaning and numerical representation of the words/tokens, but the positional encoding tells us about the position of a token in the sequence.
- From the previous point, we can accordingly differentiate between the words according to their positions, as the meaning of words depends on the position in which they are placed.
- This helps in getting the contextual information.
- Added to the input embeddings.

Positional Encoding: The Formula

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

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

- Pos = Position of the Token in the sequence.
  - Starts from 0.
- i - is the index which goes from 0 to 255.
- dmodel - dimension of the model (infeature of the first layer).



The	lion	is
PE(0,0)	PE(1,0)	PE(2,0)
PE(0,1)	PE(1,1)	PE(2,1)
PE(0,2)	PE(1,2)	PE(2,2)
....	....	....
PE(0,511)	PE(1,511)	PE(2,511)

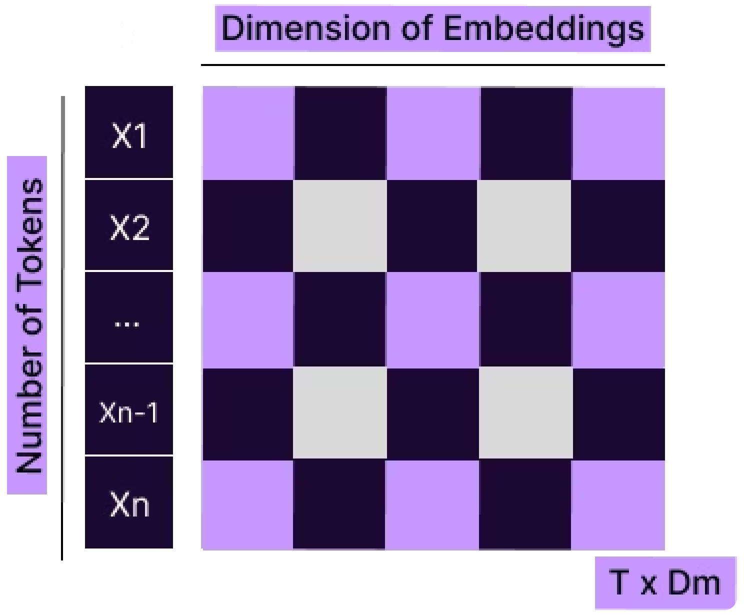
Positional Encoding: Why Sine and Cosine?

- Bounded and continuous, which helps in gradient calculation and backpropagation.
- Sine and cosine are both periodic in nature:
  - Helps with longer sequences.
  - We can find a transformation to move from one position to another (relative position).

Before Self-Attention

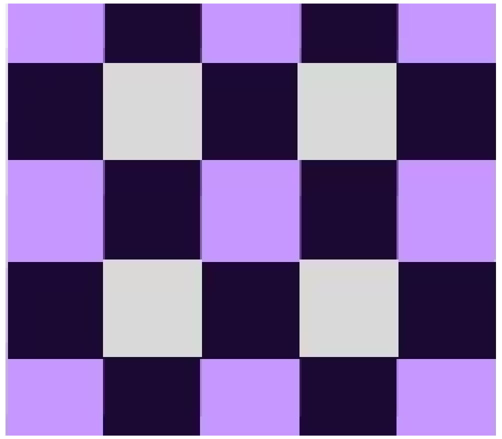
Input Sequence (X):

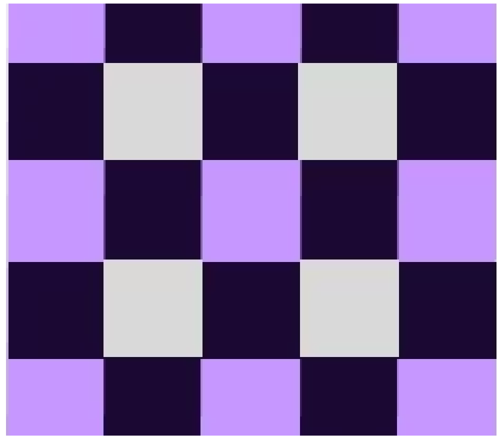
- X is the input matrix representing input sequence to the architecture.
- Each row represents a token, which is made up of the sum of input embeddings and positional encoding done before.
- The number of rows equals the number of tokens in the input sequence.
- The number of columns equals the dimension of the embedding, also known as the dimension of the model.

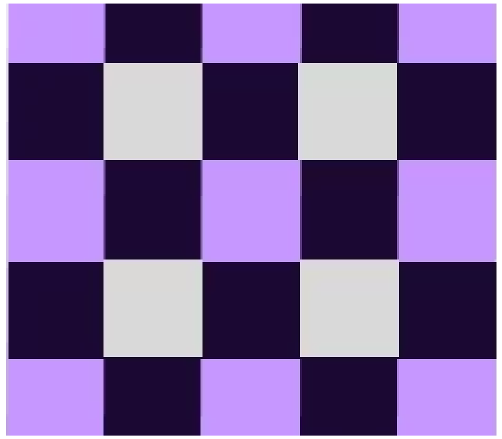


Before Self-Attention

Weight matrices:



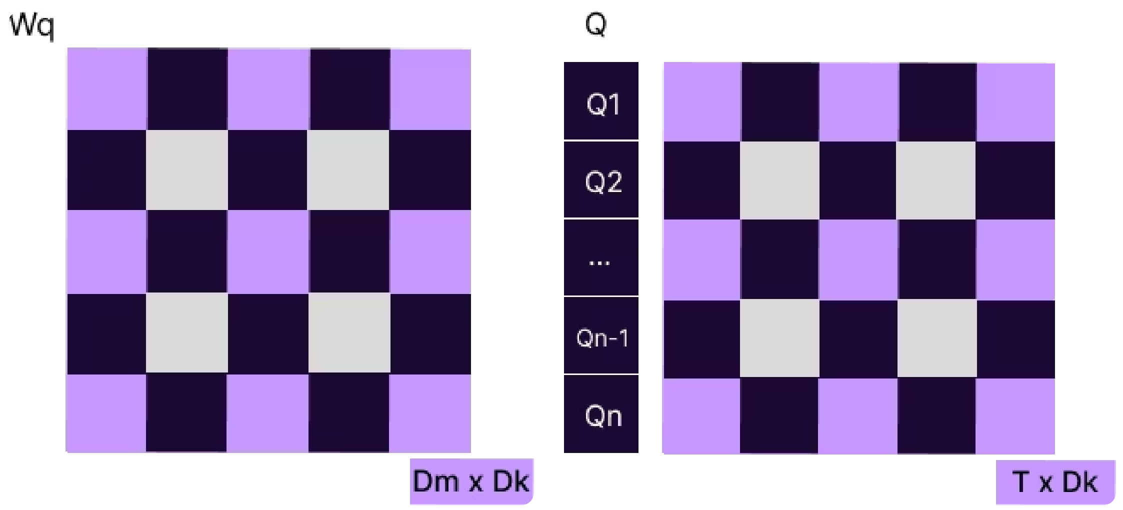




- These are the matrices that transform the input sequence X into Query, Key, and Value matrices.
- These are also called the projection matrices.
- These weights are the parameters that are learned during the training process.

Before Self-Attention

Query Matrix (Q):

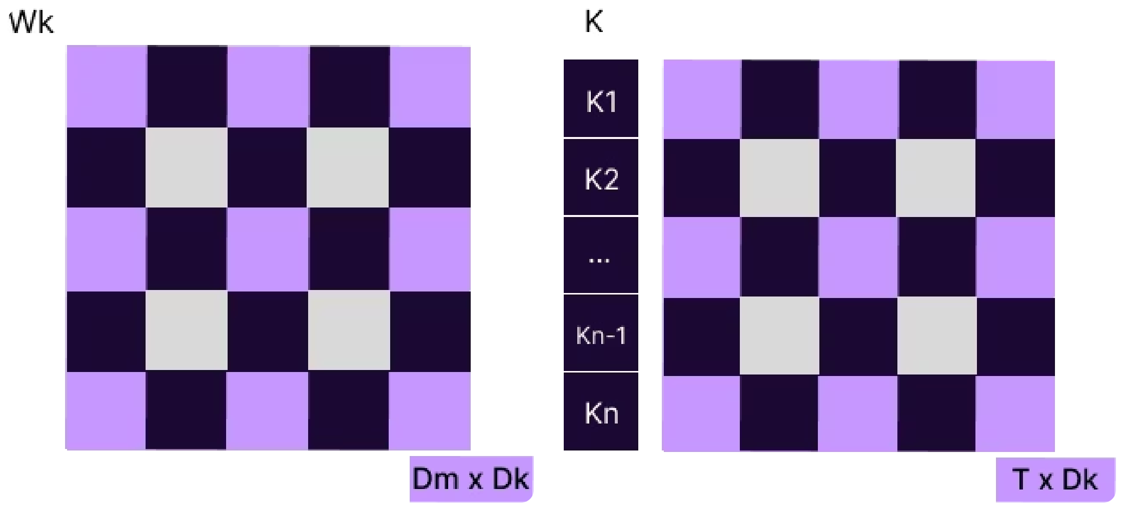


- Projection of X through the Weight matrix Wq.
- Each row in the query matrix is the query vector corresponding to a token in the input sequence.
- The query vector provides the relevant information about the token in the current position.

Made with GAMMA

Before Self-Attention

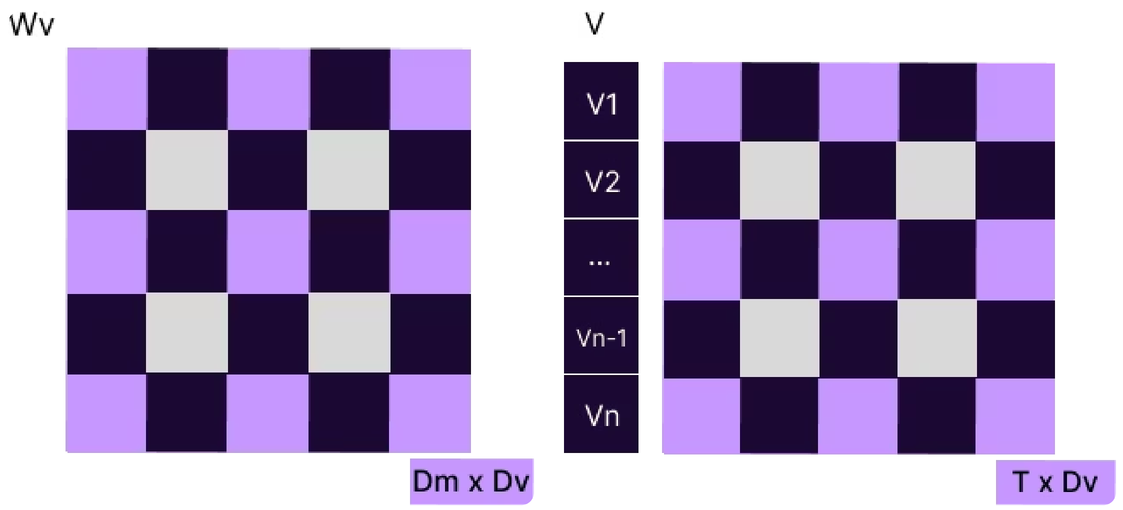
Key Matrix (K):



- Projection of X through the Weight matrix Wk.
- Each row in the key matrix is the key vector corresponding to a token in the input sequence.
- The key vector acts as a reference point.
- The key vector provides information about a token that is relevant to all other tokens in the sequence.
- This helps clarify the meaning of the query vector corresponding to a particular key vector.

Before Self-Attention

Value Matrix (V):



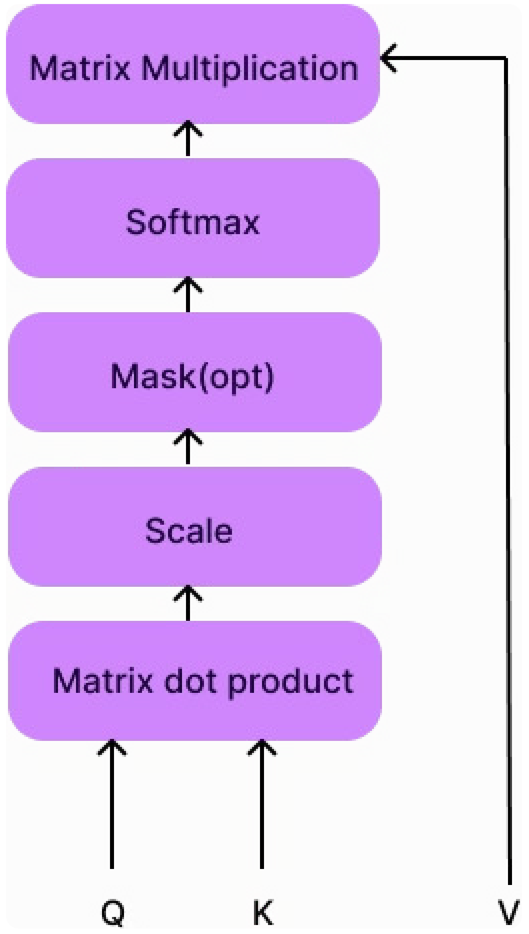
- Projection of X through the Weight matrix Wv.
- Each row in the value matrix represents the value vector corresponding to a token in the input sequence.
- The value vector holds the meaning of the corresponding input token or sometimes holds the token itself.

Self-Attention

Matrix Dot Product

- We get the projections of X (input sequence) through Wq, Wk, and Wv as the Query, Key, and Value matrices.
- The Query (Q) and Key (K) matrices are first sent through a layer in self-attention, where the dot product is performed.

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



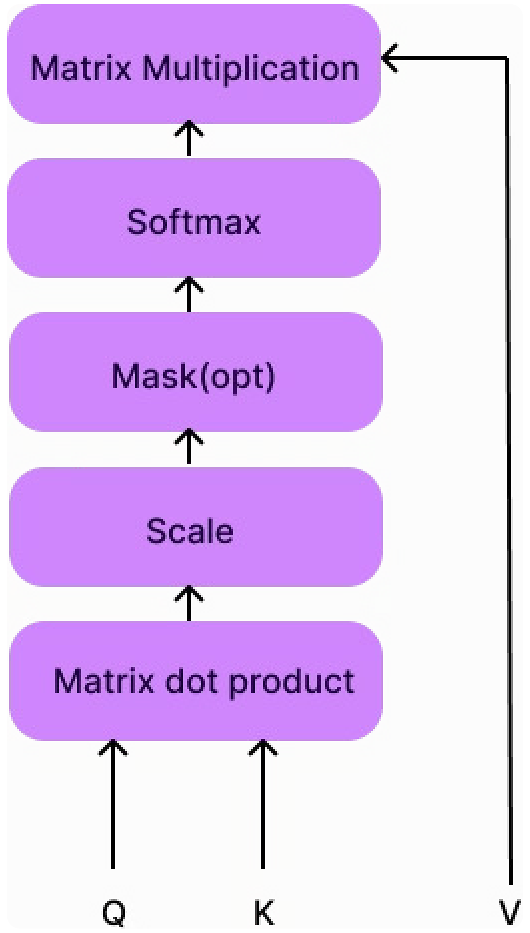
Self-Attention

Scale

- We scale it with

$1/\sqrt{Dk}$

- Dk is assigned the dimension of the embedding, otherwise it is assigned (dimension of embedding) / (number of heads).
- Reason to scale:
  - So that we have a stable gradient and learning process.



Self-Attention

Softmax

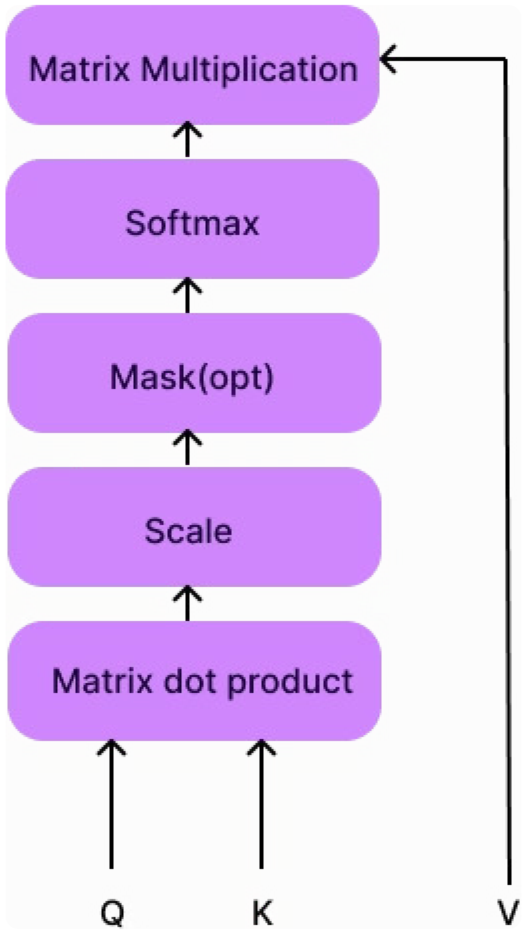
$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

The matrix that we got from the scale sublayer of self-attention would now be sent to the softmax layer, which applies this formula on each row.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

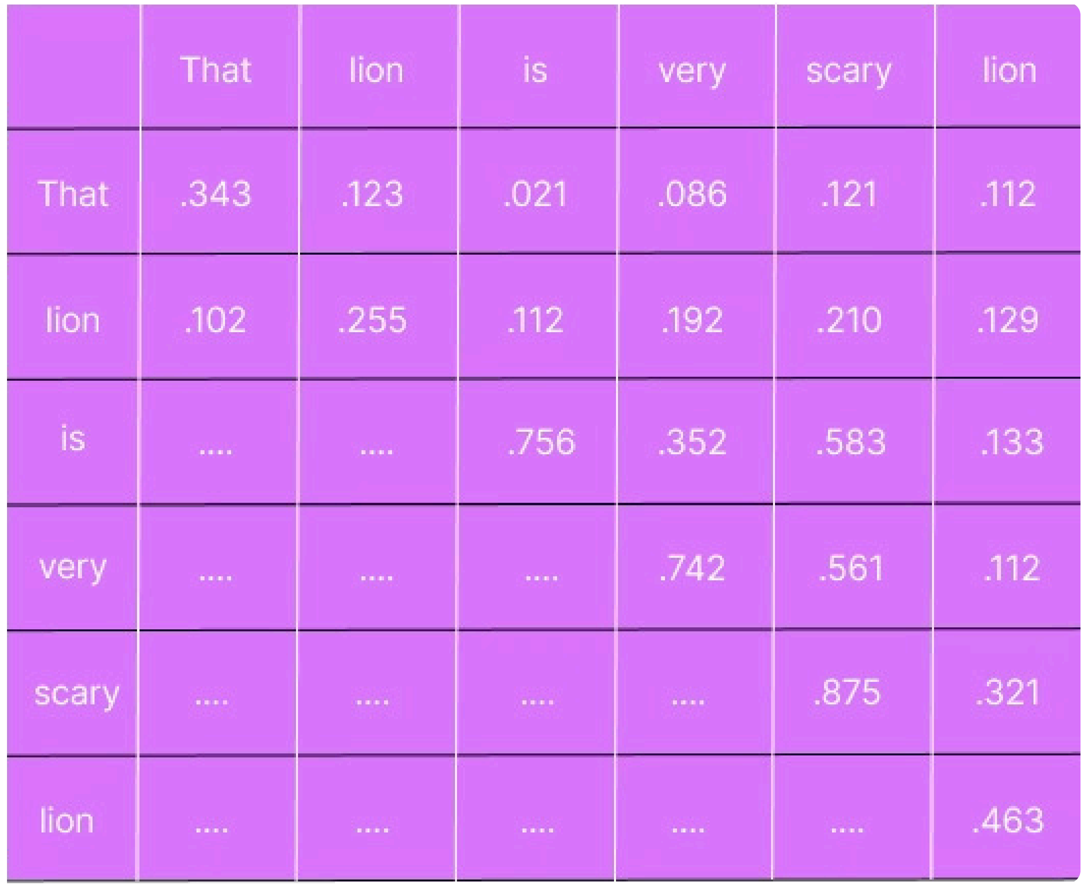
Why Softmax?

- Brings the values between 0 and 1.
- The sum of all the elements in the row would be equal to 1.
- Non-negative.



Until Softmax

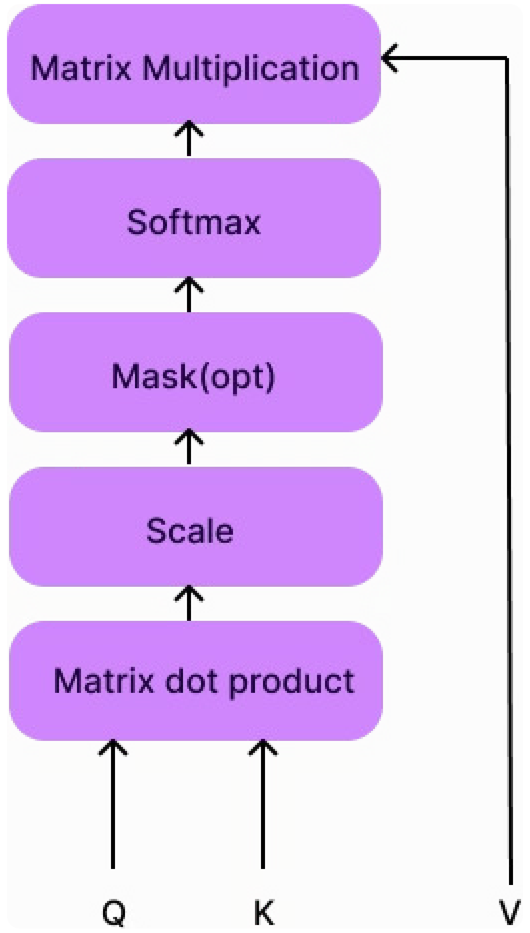
Q 6×512 X K^T 512×6 =



	That	lion	is	very	scary	lion
That	.343	.123	.021	.086	.121	.112
lion	.102	.255	.112	.192	.210	.129
is	....	....	.756	.352	.583	.133
very	....	....	....	.742	.561	.112
scary	....	....	....	....	.875	.321
lion	....	....	....	....	....	.463

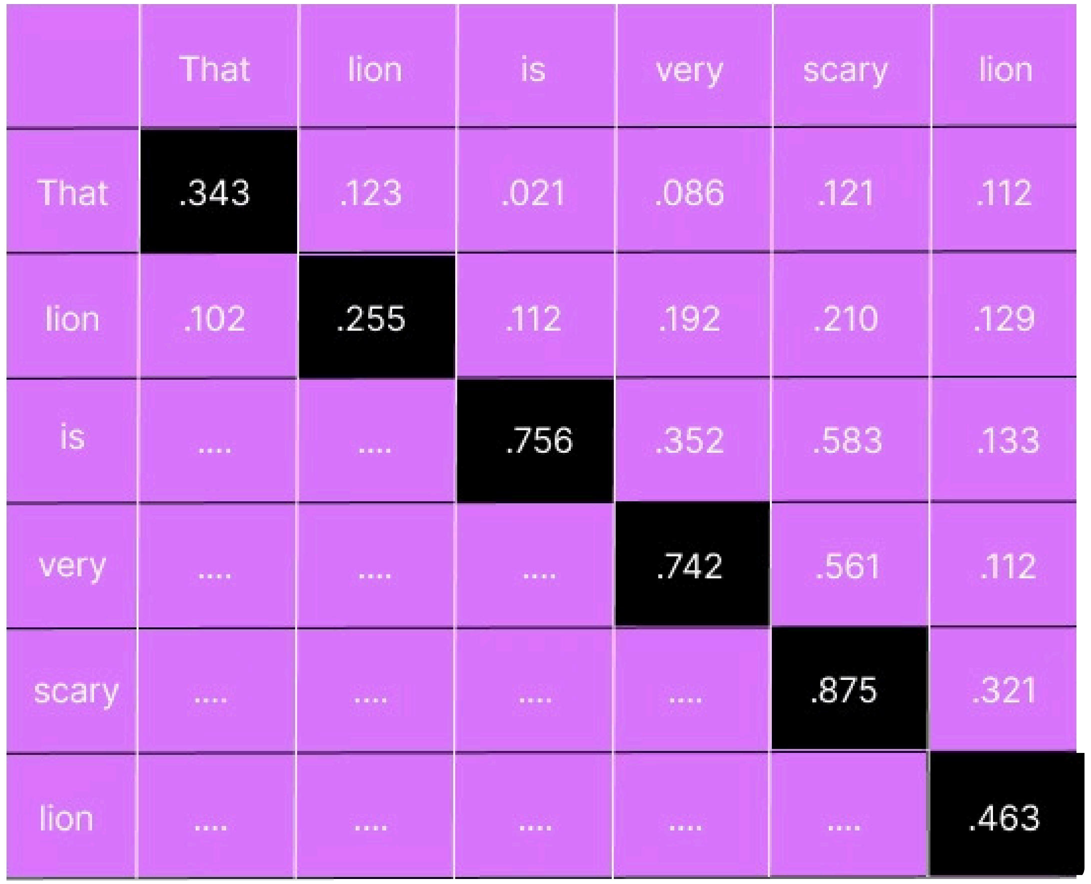
Self-Attention

- After the sublayer softmax, we get the Attention scores.
- The scores represent the relationship of one query vector with each of the key vectors.
- They also speak about how each key vector (token) is related to a particular query vector (token).
  - Higher scores represent more relevance, and lower scores represent less relevance between the words.
- Essentially, we get the meaning of a token with respect to the whole sequence.



Attention Scores

$$\underset{6\times512}{Q} \; X \; \underset{512\times6}{K^T} =$$

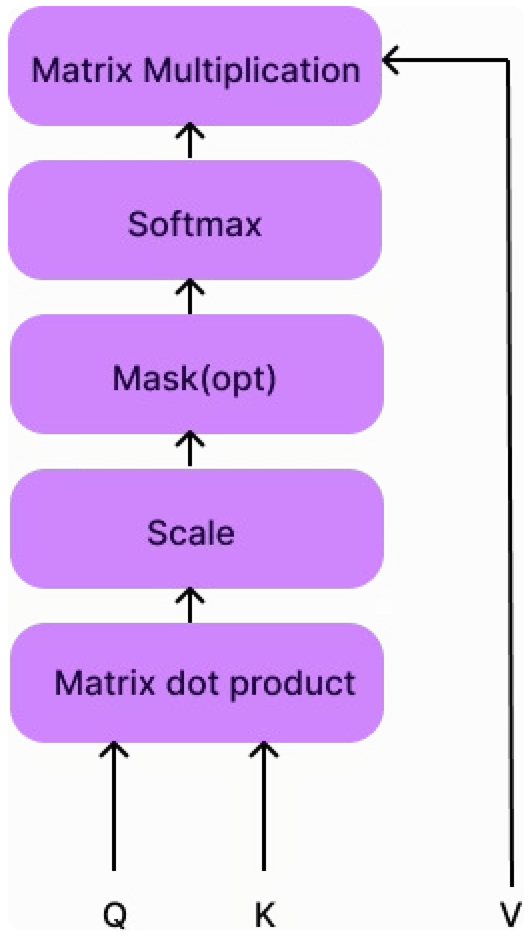


	That	lion	is	very	scary	lion
That	.343	.123	.021	.086	.121	.112
lion	.102	.255	.112	.192	.210	.129
is	....	....	.756	.352	.583	.133
very	....	....	....	.742	.561	.112
scary	....	....	....	....	.875	.321
lion	....	....	....	....	....	.463

Self-Attention

Matrix Multiplication and Weighted Sum:

- After we get the attention scores that indicate how relevant each key vector is to a particular query vector (done for all query vectors).
- We take all those scores from one row (with respect to one query vector) and multiply them with the corresponding value vectors in the value matrix.
- Then we do a weighted sum of all these, where the weights are the attention scores.
- The vector that we get is the context vector of that query vector.
    - It is influenced more by those key vectors (tokens) for which the attention score is higher.
    - Essentially, it keeps the relevant information of the query vector with respect to the whole sequence, as the less relevant key vectors are eliminated due to lower weights, giving us the meaning or information about the query vector with respect to the input sequence.



Getting the Context Vector of one query vector

Q = (Query Vector)

That	.343	.123	.021	.086	.121	.112

V =

	That	lion	is	very	scary	lion
That	1	0	1	0	0	0
lion	0	1	0	0	1	0
is	1	0	1	0	0	0
very	0	1	0	1	0	1
scary	0	0	1	0	1	0
lion	0	0	0	1	0	1

Context Vector =

[.343 * [1,0,1,0,0,0]
+.123*[0,1,0,0,1,0]+.021*[1,0,1,0,0,0]+.086*[0,1,0,1,0,1]+.121*[0,
0,1,0,1,0]+.112*[0,0,0,1,0,1]]

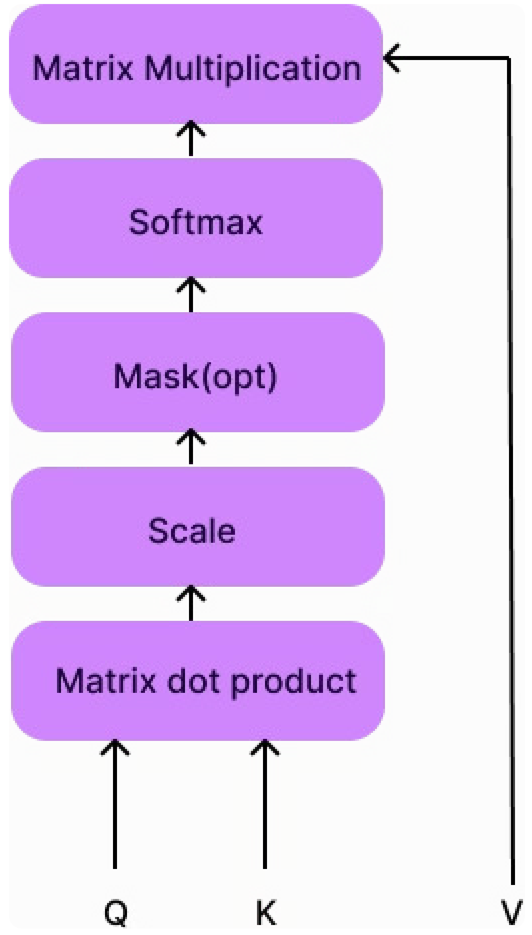
[.364,.209,.485,.198,.244,.198]

Made with GAMMA

Self-Attention

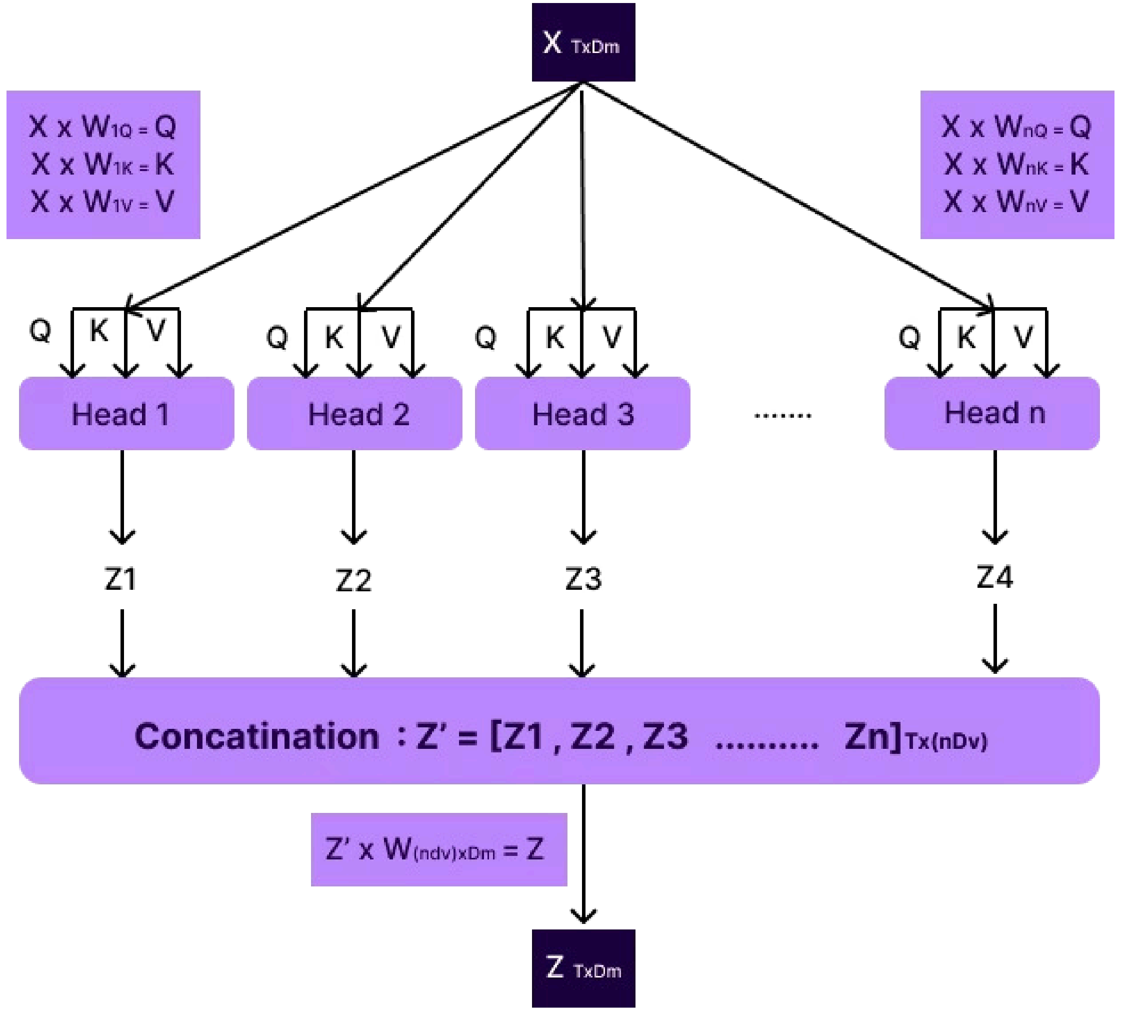
Mask

- After the sublayer scale and before the softmax sublayer.
- Used in the decoder block of the transformer.
- All the elements above the diagonal are set to -infinity (masked), and the model will not learn or ignore those.



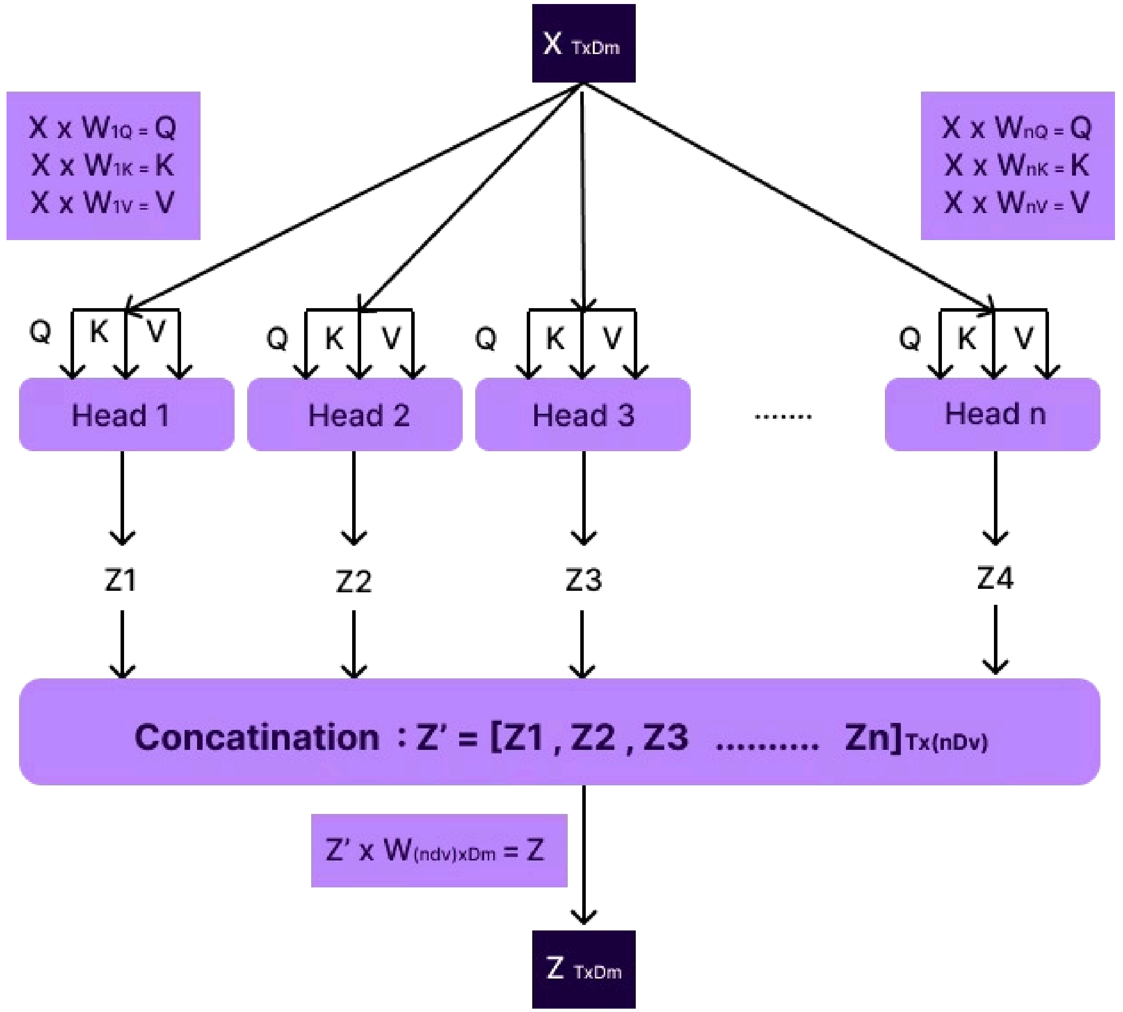
Multi-head Attention

- Works the same as self-attention (one head).
- But here we have multiple heads doing the same job of self-attention.
- Hence, we will have the dk and dv dimensions of the weight/projection matrices equal to dm/h, where h is the number of heads.
- Then from each head, we get Zi's, which are then sent to the sublayer where they are concatenated.
- Finally, using one more weight matrix, we project the output of the concatenation sublayer to the input dimension.



Multi-head Attention

- Advantages of the multi-head attention
  - Better representation of the input.
  - Due to multiple projection matrices projecting the input to Q, K, and V, we can focus on different aspects.
  - Improved adaptability to various datasets.



Made with GAMMA

Multi-head Attention

Transformer uses multi-head attention in three ways:

1. **Encoder Attention**
   - Q, K, V come from the output of the previous encoder block or, for the first encoder block, from the original input sequence.
   - Each token will be able to attend to all other tokens.
   - The output of the encoder stack will be a rich representation of the input sequence, containing diverse information.
   - This gives us information regarding which word is relevant to which word in terms of meaning and context.
2. **Decoder Attention**
   - Q, K, V come from the output of the previous decoder block.
   - Each token will be able to attend only to the previous token and itself.
   - This allows us to get the information relating to a particular token only based on the previous tokens.
3. **Encoder-Decoder Attention**
   - K, V come from the output of the encoder stack, providing contextual and relevant information regarding each input token.
   - Q comes from the previous sublayer.
   - Here, each token will be able to attend to every other token of the encoded input sequence.
   - The attention scores calculated for a particular query vector (token being generated) indicate how each key vector from the encoded key matrices is relevant to that query vector.
   - These attention scores are then multiplied with value vectors from the encoded value matrix, giving us the context vector for that query vector.
   - This context vector would have the relevant information about the input sequence and also get the information from the previous tokens being generated.
   - With this information from both the encoded input sequence and the information of previous tokens, we will be able to genarate current token being processed in the target sequence.

Add & Norm

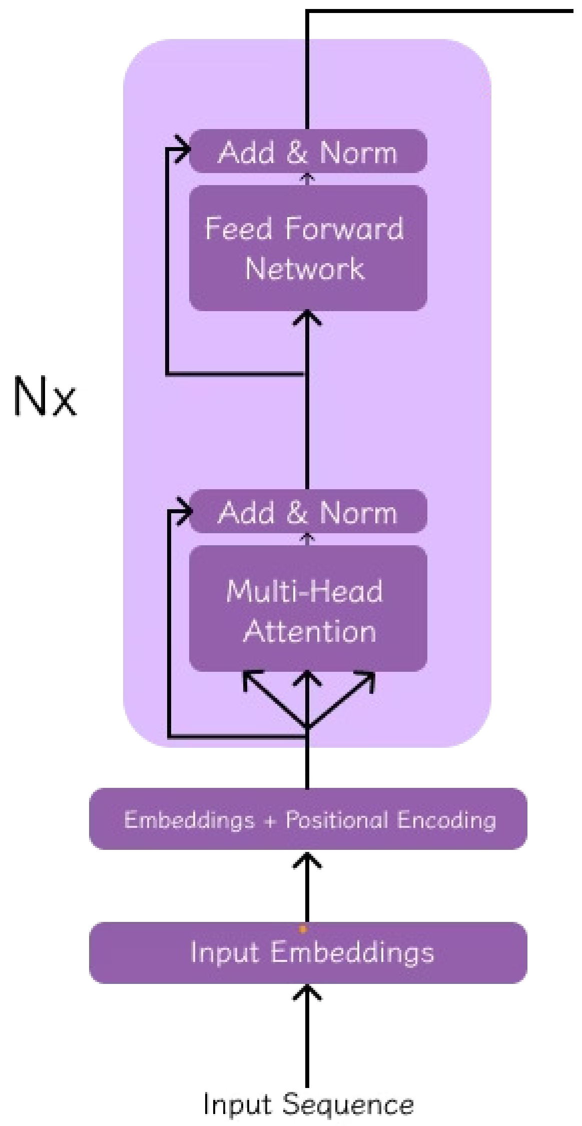
- After the Multi-Head attention sublayer.

Add

- **Residual connection:** We add the input sequence matrix to the output of the multi-head attention layer.
- Used during training mainly.
- Why we use residual connection:
    - So that we won't face the issue of vanishing gradient.
    - Richer representation, so better for training.

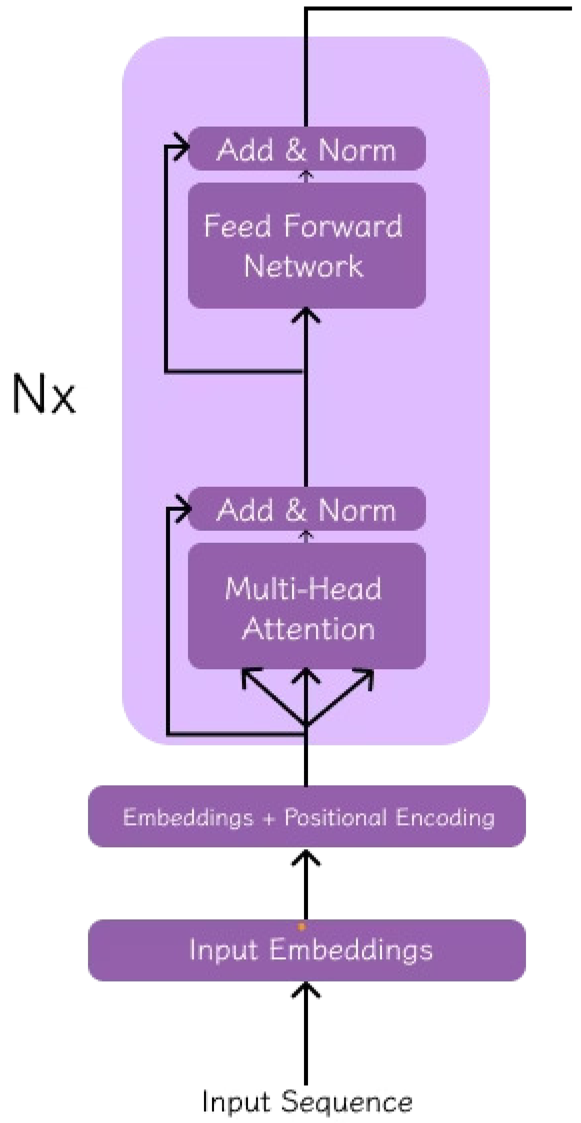
Layer Normalization

- To bring them to the same scale and distribution.
- To stabilize the learning process.
- Done for each token independently within the features.



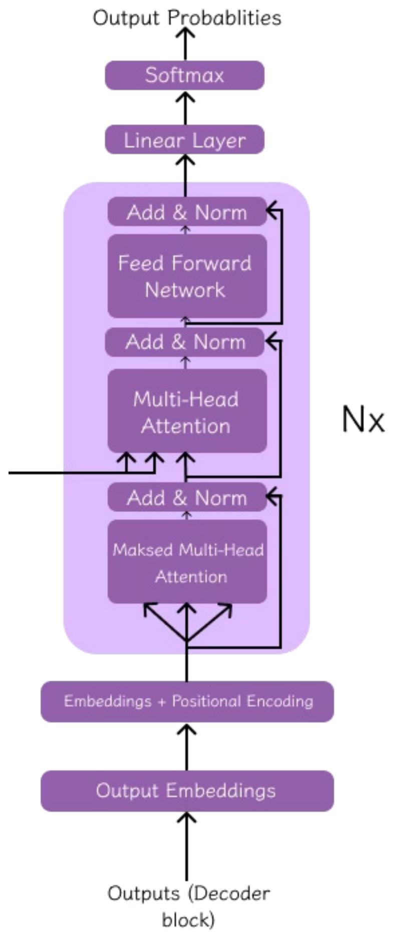
Feed Forward Network

- Performed for each position in the input sequence.
- **Network consists of:**
  - Two linear layers.
  - An activation function.
- The linear layers are used to project from a lower dimension to a higher one(by 4 times), and after the activation function, we use another linear layer to project it back to the dimensions of the input sequence.
- We use the ReLU activation function, which is ReLU(x) = max(0,x).
  - Helps the model learn complex patterns.
- **The summary of the network:** FFN(x) = max(0, xW1 + b1)W2 + b2.
- The final sublayer is **add & norm**, after which we will have the input for the next encoder block or the encoded output of the input sequence.



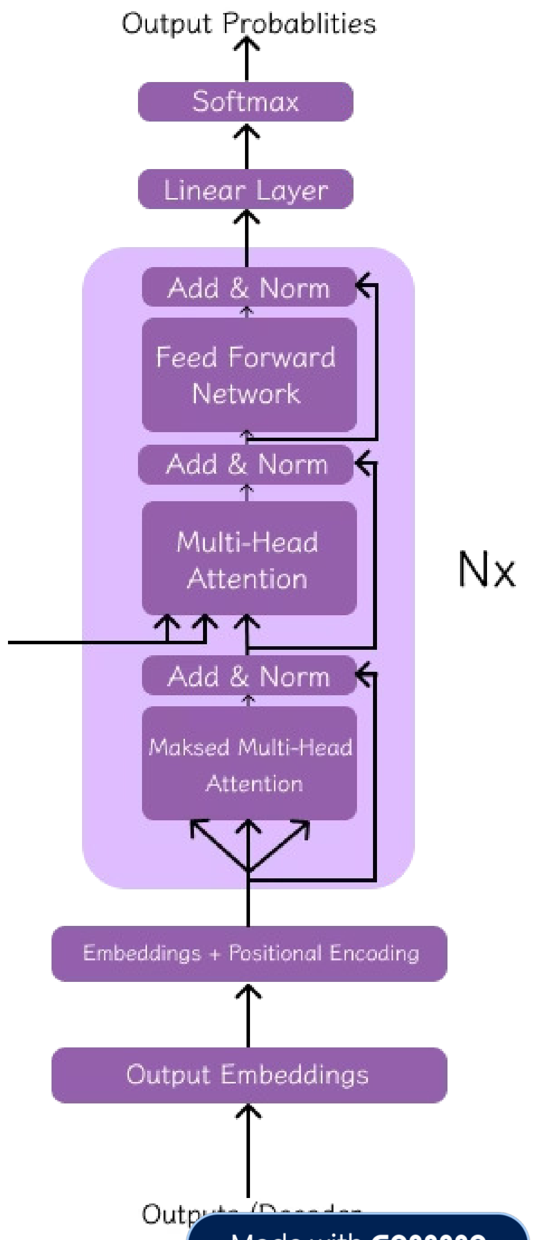
Made with GAMMA

Decoder Block



Decoder Block

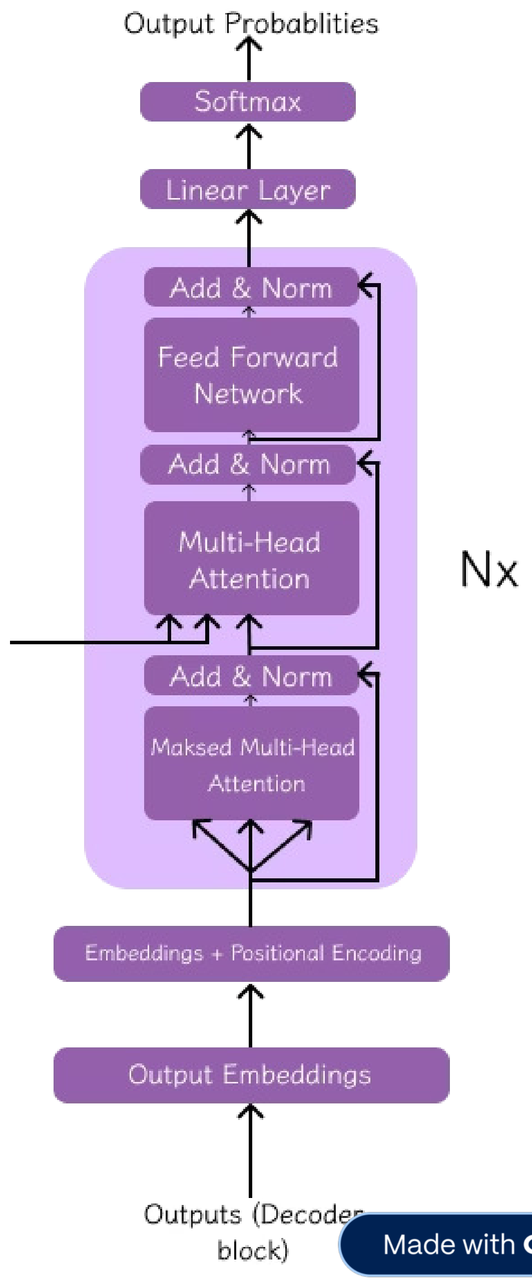
- Similar sublayers as the encoder block.
- **Autoregressive:**
  - Begins with the start token.
  - After which, it predicts one token at each time step.
  - The token being generated is dependent only on preceding tokens.
  - It takes information from the encoded input sequence.
  - The outputs of a decoder stack are given as list of input to bottom decoder layer to predict or generate the next token.



Decoder Block

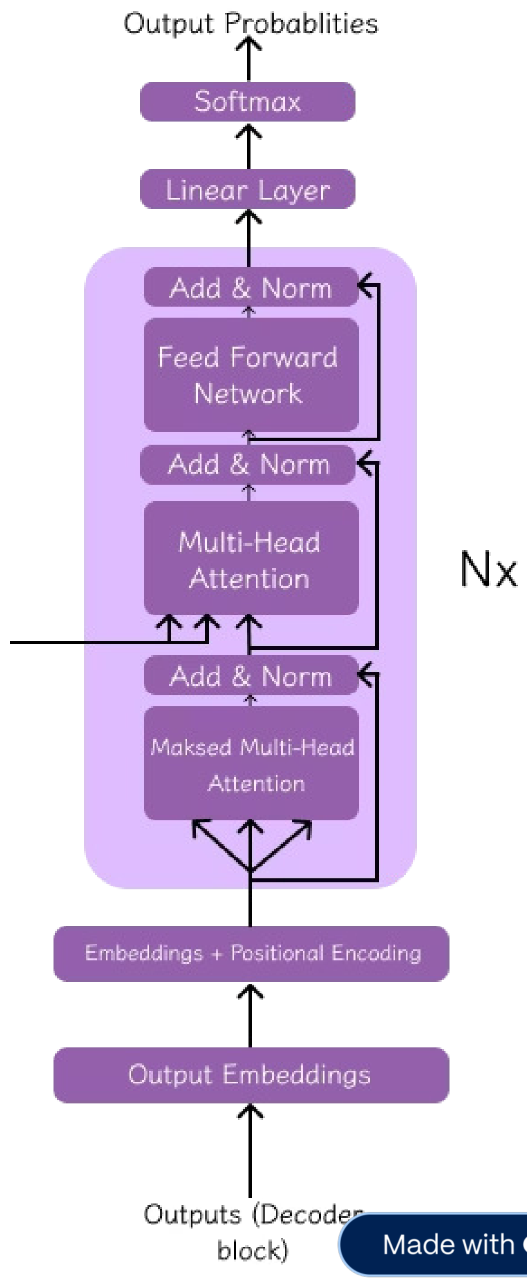
Sublayers

- Embedding is done with positional encoding.
- Decoder attention(Masked).
- Add & norm.
- Encoder-decoder attention.
- Feed forward network.



Linear Layer - Classifier

- The Decoder stack gives us a vector that provides information regarding the token at a particular position in relation to the whole input sequence.
- This would help us in generating the token in the target sequence.
- Now, this vector is projected over the vocabulary, and we get a vector of the size of the vocabulary.
- This output is called **logits.**
- These logits are not normalized and give us the likelihood or the prediction being that token from the vocabulary.

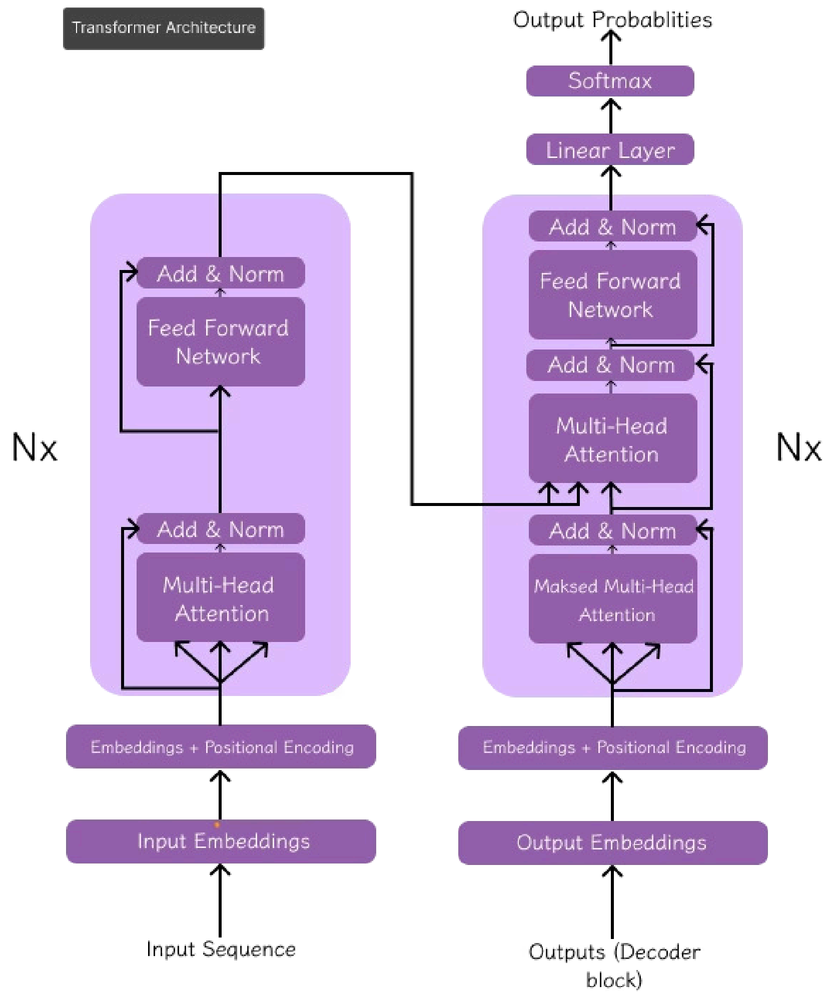


Softmax

- Process of getting the word or token from numerical representation.
- Converts the logits into probabilities.
- The cell or element with the highest probability wins and will be added to the sequence and given to the bottom decoder block for the next prediction.

Conclusion: Transformer

- Examine the architecture and flow of the Transformer.
- The Transformer is faster and more efficient than traditional RNN architectures.
- The Transformer has been extended to handle other datas like Images, Video, and Audio.



References

- https://jalammar.github.io/illustrated-transformer/
- https://arxiv.org/abs/1706.03762
- https://www.geeksforgeeks.org/getting-started-with-transformers/