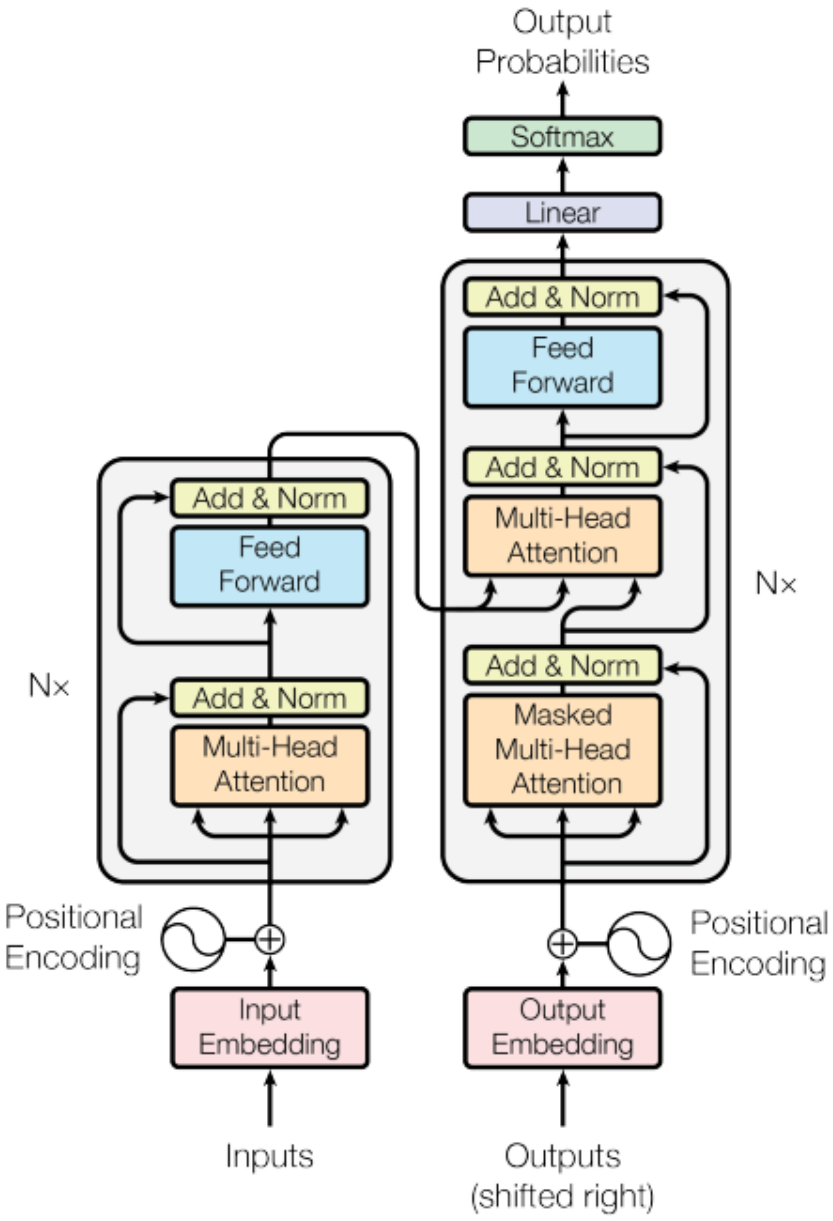
Transformers

This is the basic transformer architecture

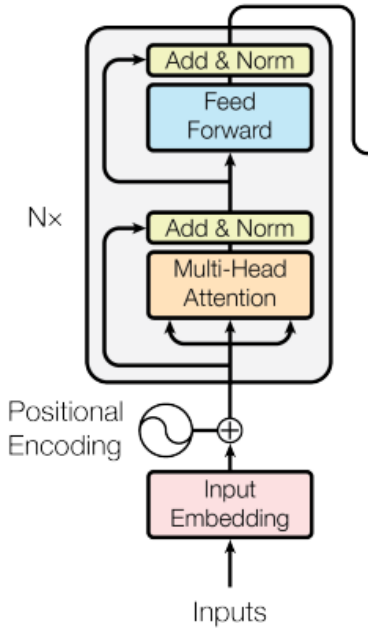


It contains 2 macro-blocks:

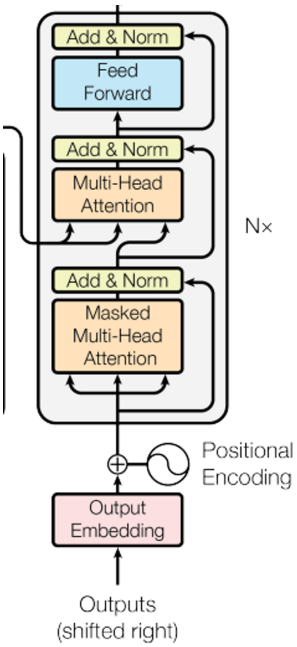
1. Encoder
2. Decoder

and a linear layer

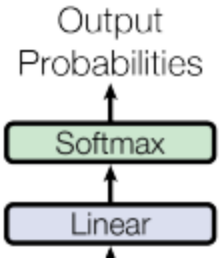
The encoder macro-block:



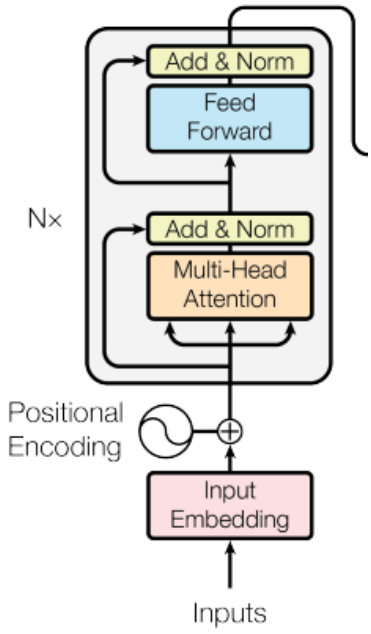
The output of the encoder macro-block is connected to the decoder macro-block:



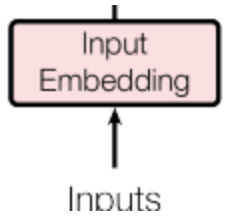
Transformer architecture also contains a linear layer:



Encoder:

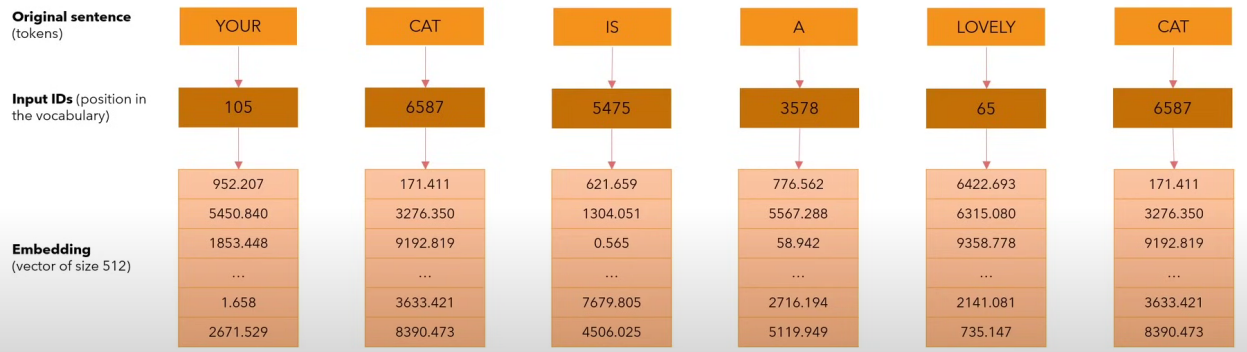


Input Embedding:



- Encoder starts with the input embeddings.

What is an input embedding?



- Embeddings are an array of floating point number. They can be used to represent different modalities (text, image, video, etc.).
- Note that the same object (word here) will always have the same embeddings. E.g. CAT in the example given above.
- Here, we convert each word into an embedding of size 512 (contains 512 floating point numbers).
- Therefore, we can define our $d_{model} = 512$

Positional Encoding:

- We want each word to carry some information about it's position in the sentence.
- We want the model to treat words that appear close to each other as 'close' and words that are distant as 'distant'.
- E.g. in the sentence:

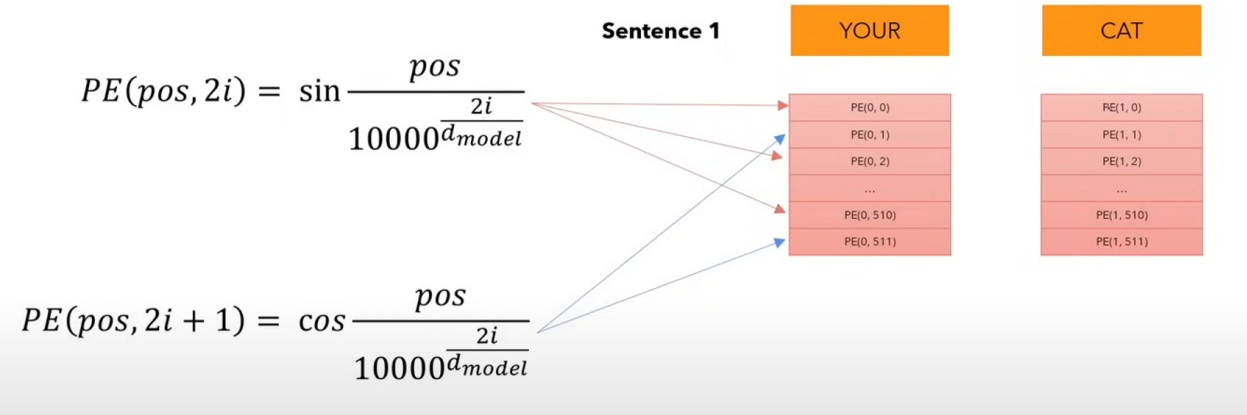
> Hi, I'm Vansh Kharidia, and I'm into tech.

- We know that *Vansh* and *Kharidia* are close to each other by seeing the sentence, but the model doesn't have this information.
- We want the model to have this information about the position of words/tokens. Positional encoding is used to give this information to the model.
- We want the positional encoding to represent a pattern (e.g. Vansh is followed by Kharidia) that can be learned by the model.

What is a positional encoding?

- We add a position embedding vector of size 512 to our original embedding.
- The values in the position encoding vector are calculated only once and reused for every sentence during training and inference.
- The sum of the embedding and position embedding gives us the encoder input.
- For the same word, the vector embedding is the same but the position embedding is different.
- *Encoder input = Embedding + Position Embedding*

How are position embeddings calculated?



- For even positions in the position embedding (count starts from 0), we use the 1st formula, and for odd positions in the position embeddings, we use the 2nd formula.
- We do this for each of the 512 values of a position embedding, for each word/token in the sentence.
- So, the position embedding for every position in the sentence is the same, regardless of the sentence. It is the encoder input (sum of embedding and position embedding) that is unique.
- Therefore, we need to compute the positional encodings only once and then we can reuse them during training & inference.
- **Why do we use trigonometric functions here?**
  - Trigonometric functions like *sin* and *cos* naturally represent a pattern that the model can recognize as continuous.
  - So, it is easier for the model to see the relative positions of a word when using trigonometric functions.

Multi-head Attention:

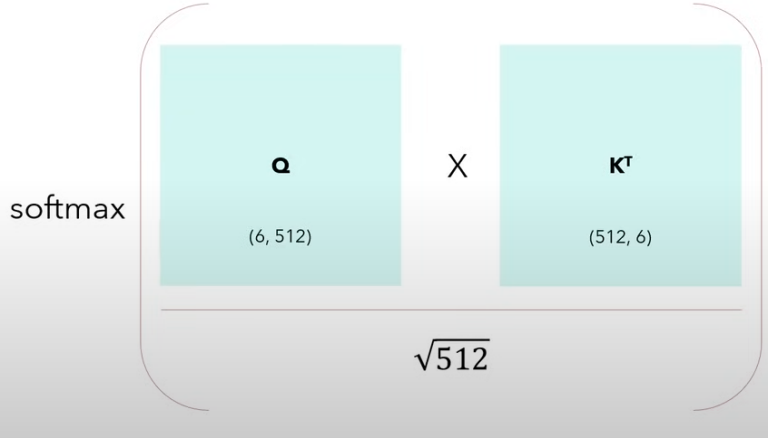
- Before diving into multi-head attention, let's first understand self-attention with a single head.

What is Self-Attention?

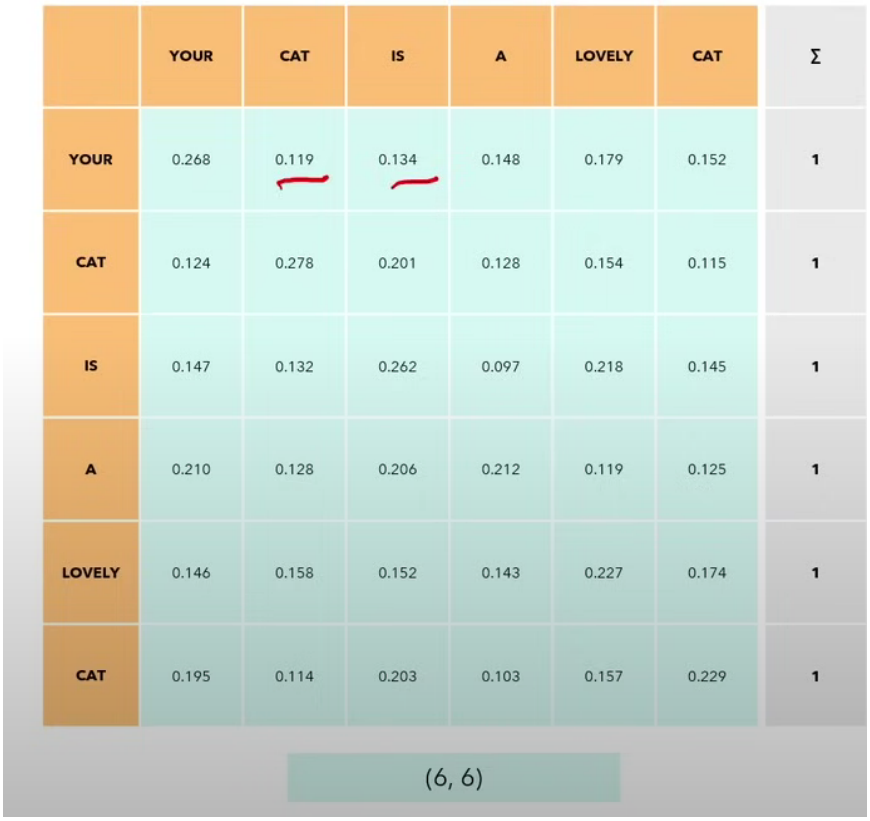
- It is a mechanism that existed before transformers. The introduction of transformers changed it to multi-head attention.
- Self-attention allowed models to relate words to each other.
- We calculate self attention using the formula

$$SingleHeadAttention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

- Where, sequence length *seq* = 6, and d_{model} = d_k = 512.
- The matrices Q (query), K (key) and V (value) are the input sentence of dimension 6×512 (6 rows and 512 columns).



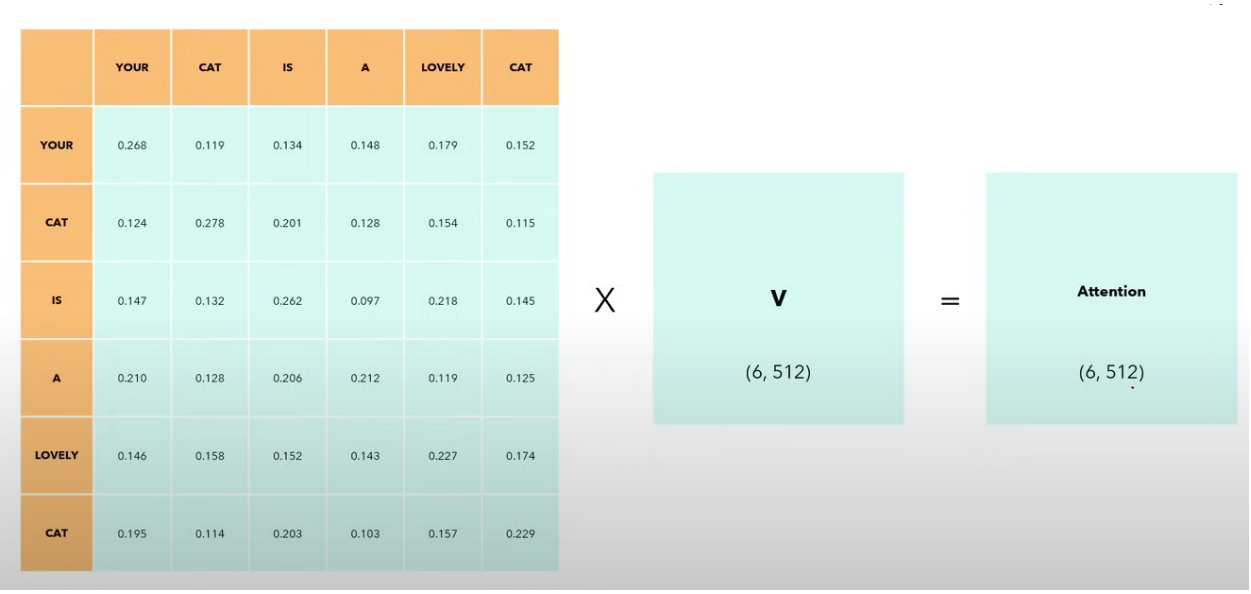
- As a result of this, we get a 6×6 matE.g. 0.119 shows



	YOUR	CAT	IS	A	LOVELY	CAT	Σ
YOUR	0.268	0.119	0.134	0.148	0.179	0.152	1
CAT	0.124	0.278	0.201	0.128	0.154	0.115	1
IS	0.147	0.132	0.262	0.097	0.218	0.145	1
A	0.210	0.128	0.206	0.212	0.119	0.125	1
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174	1
CAT	0.195	0.114	0.203	0.103	0.157	0.229	1

(6, 6)

- The dot product shows the relationship between each combination of words.
- E.g. 0.119 shows how intense the relationship is between the words *YOUR* and *CAT*. It gives a score.
- NOTE: in the image, random values are inserted.



- Now, we multiply this matrix by the *V* matrix to get the attention.
- The dimension of the resultant (attention) matrix is the same as the original matrix: 6×512.
- The attention matrix contains information about 6 words (rows), and for each word, it contains information not only about the meaning of the word (given by the embedding) or it's position in the sentence (as given by the position encodings), but also **each word's interaction with other words**.

Self-Attention in Detail:

- Self-attention is permutation invariant. This means that the values of the content (e.g. a word) do not change if we switch it's row with another row. It is a desirable property.
- Self-attention requires no parameters except the embeddings of the words. This will change with multi-head attention.
- We expect values along the diagonal to be the highest as they are the same word (and so they will be highly related).

	YOUR	CAT	IS	A	LOVELY	CAT
YOUR .	0.268	0.119	0.134	0.148	0.179	0.152
CAT	0.124	0.278	0.201	0.128	0.154	0.115
IS	0.147	0.132	0.262	0.097	0.218	0.145
A	0.210	0.128	0.206	0.212	0.119	0.125
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174
CAT	0.195	0.114	0.203	0.103	0.157	0.229

- If we don't want some positions to interact, we can always set their values in the matrix to $-\infty$ before applying softmax (which will turn $-\infty$ to 0). The model will not learn these interactions. This property will be used in the decoder.

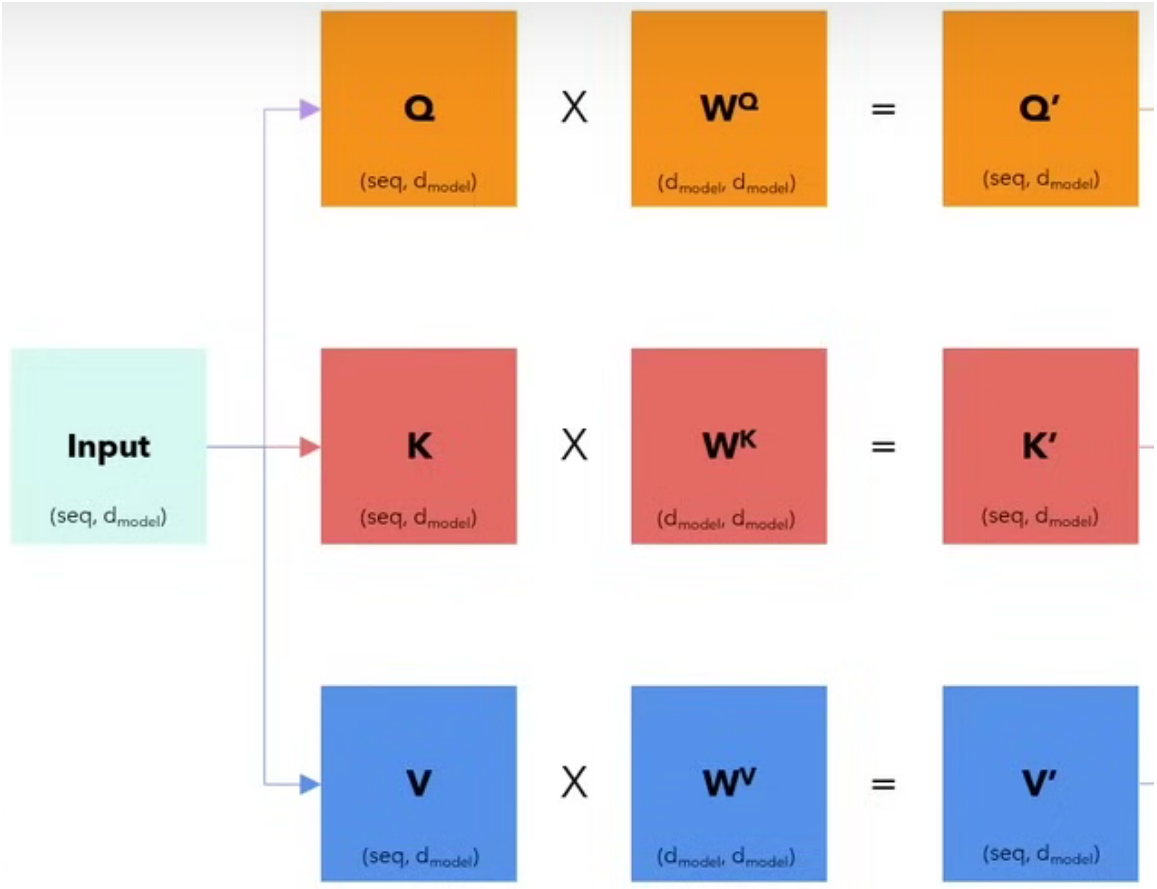
Multi-head Attention:

- The encoder input embedding is passed 3 times (Q, K, V) into multi-head attention and once into Add & Norm, so we have 4 heads.

$$SingleHeadAttention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

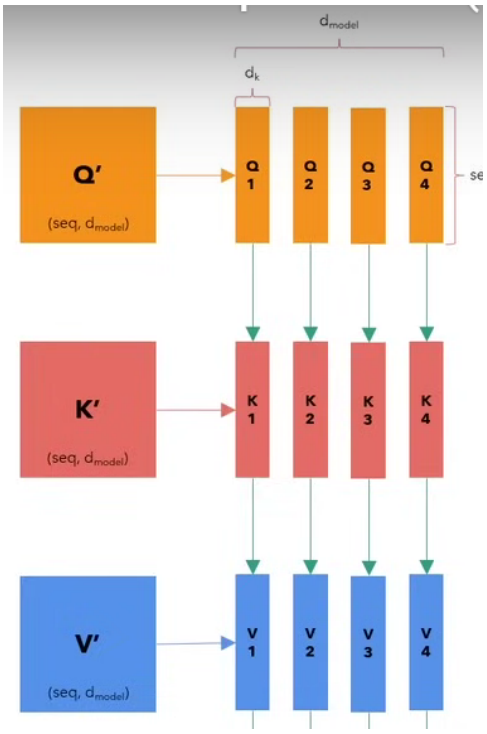
$$MultiHeadAttention(Q,K,V) = Concat(head_1,...head_h)W^0$$

$$head_i = SingleHeadAttention(QW_i^Q, KW_i^k, VW_i^V)$$

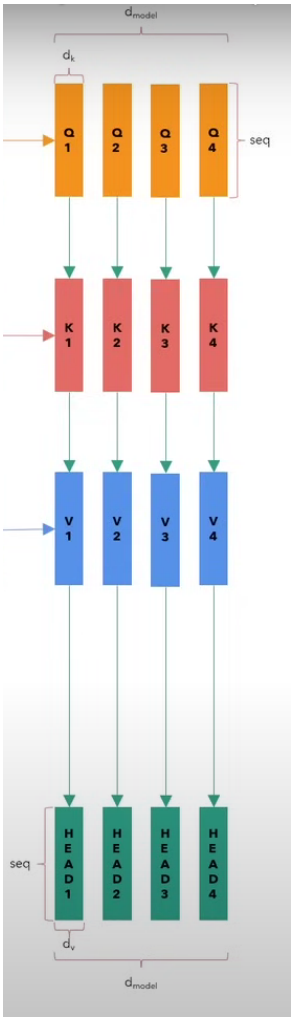


$$seq = sequence\ length$$
$$d_{model} = size\ of\ the\ embedding\ vector$$
$$h = number\ of\ heads$$
$$d_k = d_v = \frac{d_{model}}{h}$$

- We perform the same computation that we did for single-head attention for Q, K and V.



- The resultant Q′, K′ and V′ matrices are divided into 4 matrices each (as there are 4 heads).
- They are divided by d_{model}, so each submatrix the entire sentence but only a subsection of the embeddings
- Each submatrix contains d_k embeddings (columns) for each word.
- In our case, $d_k = \frac{d_{model}}{h} = \frac{512}{4} = 128$

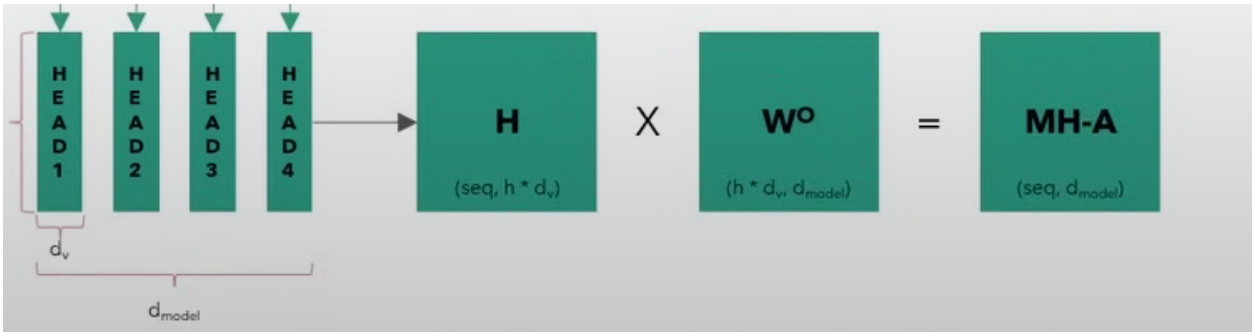


- We can calculate the attention between submatrices using the formulae:

$$SingleHeadAttention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

$$head_i = SingleHeadAttention(QW_i^Q, KW_i^k, VW_i^V)$$

- Which gives us *HEAD1, HEAD2, HEAD3* and *HEAD4*.



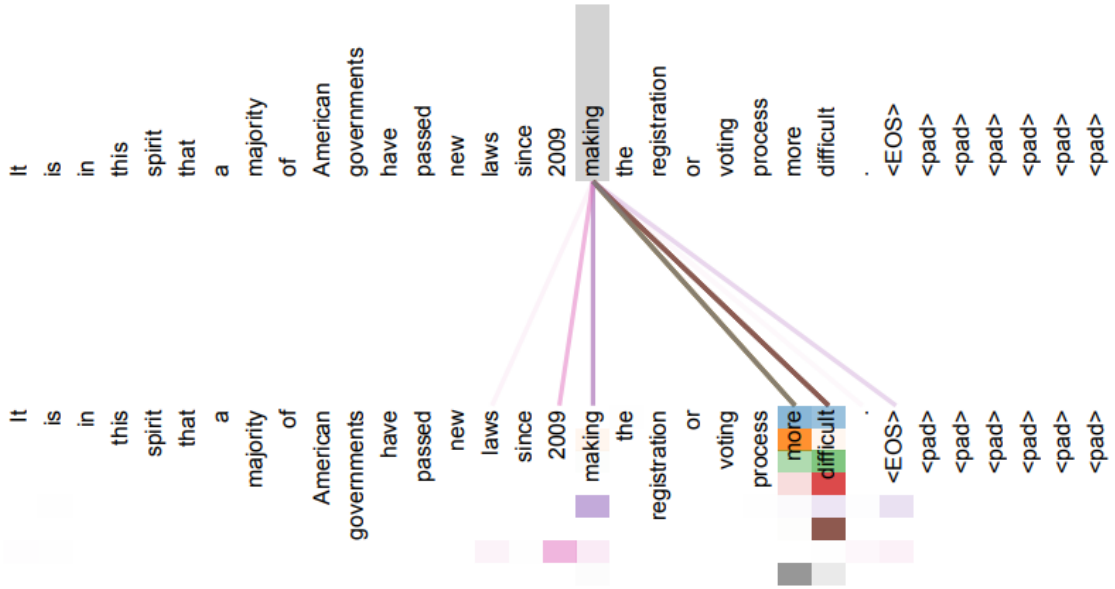
- Then we concatenate all the 4 heads using the formula:

$$MultiHeadAttention(Q, K, V) = Concat(head_1, ...head_h)W^0$$

- We get the *H* matrix which retains the original dimensions $seq * d_{model}$.
- We multiply *H* by W^0 matrix which is of the dimensions $d_{model} * d_{model}$
- The result is a multi-head attention matrix which is also of the original dimensions $seq * d_{model}$

Why multiple heads?

- As each head contains the entire sentence but only a section of the embeddings.
- As the same word can be used a noun in a context, adjective in another context, adverb in another context, etc., same word has different meanings, connotations and position in the sentence depending on the context.
- We can leverage the multi-head architecture as different heads can learn to relate the same word in different contexts (e.g. as noun, adjective, etc.).



- Different lines represent different heads and how they relate the word *making* with other words.
- The various colour blocks show the distribution of attention weights for each head, with different colours representing different heads.
- In the visualization, *making* is related to *more difficult*.
- Multiple heads relate *making* to *difficult*, meaning that multiple Q, K and V submatrices contained context that related *making* to *difficult*.
- The head denoted by purple relates *making* with *2009*, while other heads don't make this relation. This indicates that the purple head saw the portion of the embedding that made the interaction between *making* and *2009* possible, but other heads did not see this interaction.

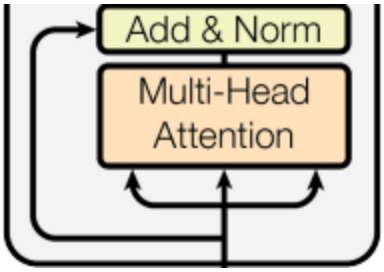
Why is there a line between *making* and *making when they are the same word?*

- This phenomenon is called self-attention, i.e. the model is paying attention to the word itself.
- When a word attends to itself, the model is considering it's own importance in it's context. This is often necessary for understanding and retaining the word's

specific meaning or role in the sentence, especially in complex phrases or when disambiguation is required.

- As the line between *making* and *making* is thick, it indicates that the head is strongly considering *making* while determining the context.
- It is important for context preservation, e.g. the phrase "making voter registration and the voting process more difficult," the word "making" is a verb that connects to multiple objects and actions. Attending to itself helps the model keep the verb's action central while processing other parts of the sentence.

Layer Normalization (Add and Norm):



Layer Normalization:

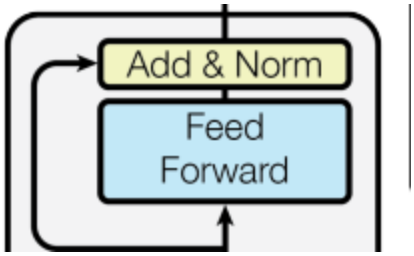
- We normalize the values so that they are in the range of 0 and 1.
- We also introduce 2 parameters, usually *beta* and *gamma*.
- Gamma is multiplicative, we multiply it with the normalized value.
- Beta is additive, we add beta to the product of gamma and the normalized value.
- Beta and gamma introduce some fluctuations in the data as having all the values between 0 and 1 may be too restrictive for the network.
- The network will learn to tune beta and gamma to introduce fluctuations when necessary.

- So, beta and gamma control which values are amplified and by how much.

Batch vs Layer Normalization:

- In batch normalization, we consider the same feature for the entire batch (of words/tokens).
- In layer normalization, we consider all the features of an item in the batch.

Feed Forward & Add and Norm:



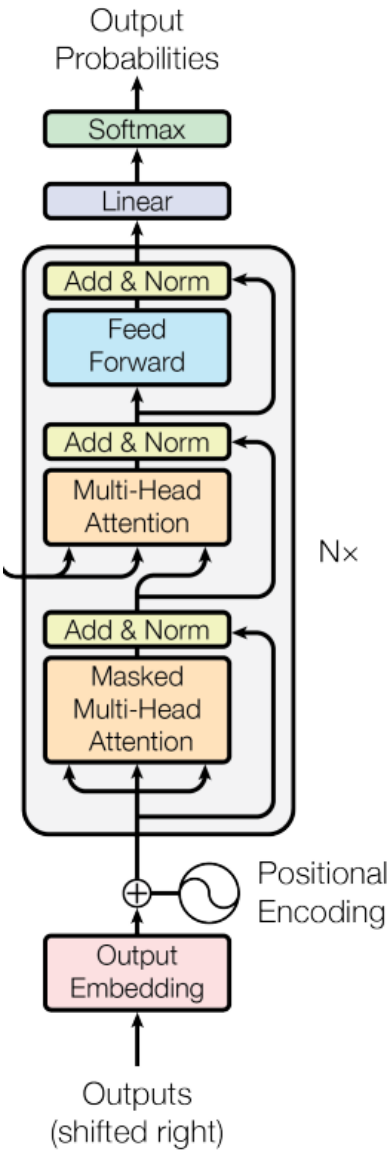
Feed Forward:

- It processes each position in the sequence independently and helps the model to learn complex representations by applying non-linear transformations to the input.
- It is a simple, fully connected feed-forward network that is applied to each position separately and identically. It consists of two linear transformations with a ReLU activation in between.
- **First Linear Transformation**: This layer projects the input into a higher-dimensional space.
- **ReLU Activation**: A non-linear activation function applied to introduce non-linearity into the model.
- **Second Linear Transformation**: This layer projects the higher-dimensional representation back to the original dimension.

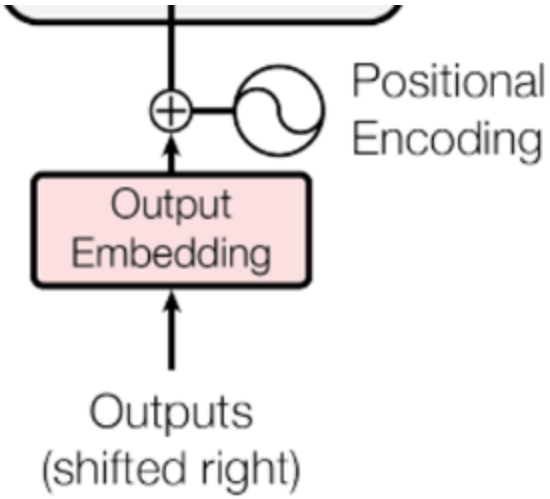
Add & Norm:

- Performed after feed forward.
- **Residual Connection (Add)**: The input to the FFN is added to its output. This is known as a skip connection or residual connection and helps in addressing the vanishing gradient problem, facilitating better gradient flow through the network.
- **Layer Normalization (Norm)**: Layer normalization is applied to the result of the addition. Layer normalization normalizes the summed vectors to have zero mean and unit variance, which helps in stabilizing and accelerating the training process.

Decoder:

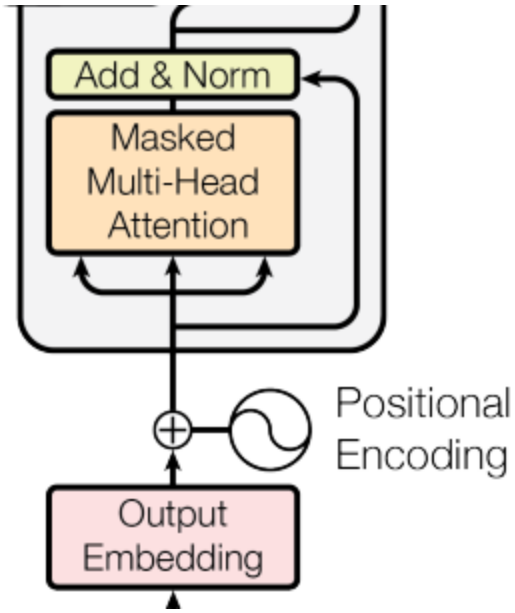


Output Embedding (& Positional Encoding):



- This part is similar to the encoder.
- During training, the target sequence (i.e., the correct output sequence) is used as input to the decoder. However, it is shifted to the right by one position.I
- Shifting the target sequence allows the model to predict the next token based on the previous tokens.
- If the target sequence is [y1, y2, y3, ..., yn], it is transformed to [<start>, y1, y2, y3, ..., yn-1] before being fed into the decoder.
- The model learns to predict y1 based on <start>, y2 based on <start>, y1, and so on.
- The embeddings and positional encoding is combined to form a decoder input vector.

Masked Multi-Head Attention & Add and Norm:

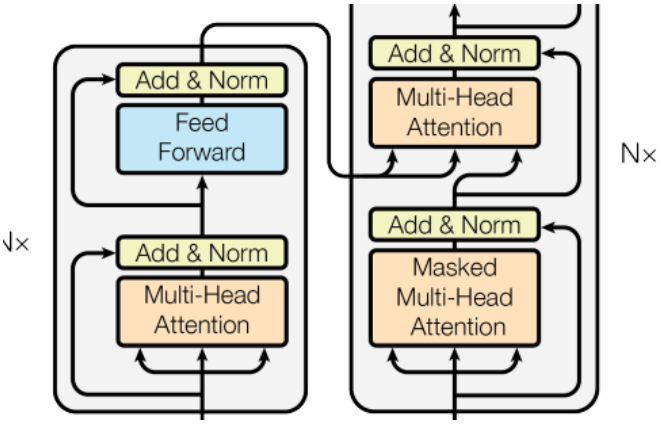


What is masked multi-head attention?

- Our goal is to make the model causal, meaning that the output at a certain position can only depend on the words on the previous positions.
- The model must not be able to see future words.
- We achieve this by replacing all the future words by $-\infty$ in the *seq * seq* matrices. After the softmax function is applied, all the $-\infty$ will be replaced by 0.

·	YOUR	CAT	IS	A	LOVELY	CAT
YOUR	0.268	~~0.114~~	~~0.134~~	~~0.148~~	~~0.129~~	~~0.112~~
CAT	0.124	0.278	~~0.231~~	~~0.158~~	~~0.154~~	~~0.115~~
IS	0.147	0.132	0.262	~~0.097~~	~~0.218~~	~~0.145~~
A	0.210	0.128	0.206	0.212	~~0.179~~	~~0.125~~
LOVELY	0.146	0.158	0.152	0.143	0.227	~~0.114~~
CAT	0.195	0.114	0.203	0.103	0.157	0.229

Multi-Head Attention & Add and Norm:



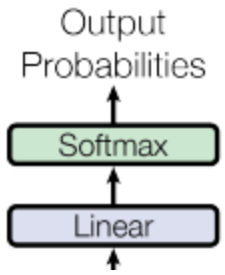
- The multi-head attention layer gets keys and values matrices from the encoder's output and the query from the output of the masked multi-head attention.
- As the keys and values are from output of the encoder and the query is from output of the masked multi-head attention, it is **cross attention** (earlier it was

self attention).

Feed Forward & Add and Norm:

- Just like in the encoder block: it consists of the same structure and serves the same purpose (introducing non-linearity to make the model learn complex representations).

Linear layer and Softmax:



Linear Layer:

- The linear layer transforms the output of the previous layer to a different dimensionality, to match the number of classes in the output vocabulary.
- It performs a matrix multiplication between the input and a weight matrix, followed by the addition of a bias term. The formula is:

$$z = xW + b$$
$$x = input\ vector$$
$$W = weight\ matrix$$
$$b = bias\ vector$$
$$z = output\ of\ the\ linear\ layer$$

- It transforms the final hidden state outputs from the decoder into logits (output of a neural network before activation function is applied).

Softmax:

- Converts the logits from linear layer into probabilities by applying the softmax function.

Training a Transformer:

- Using an example of translation using transformers to illustrate how transformers are trained.

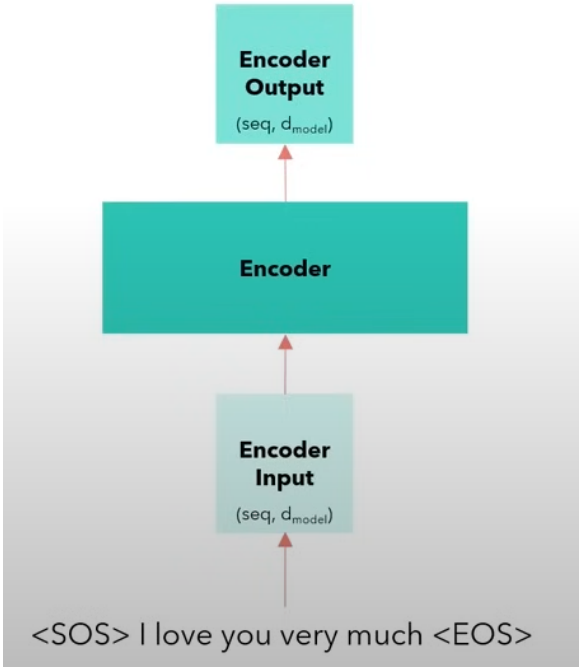
Task at Hand:

Convert English to Italian:

English: I love you very much

Italian: Ti amo molto

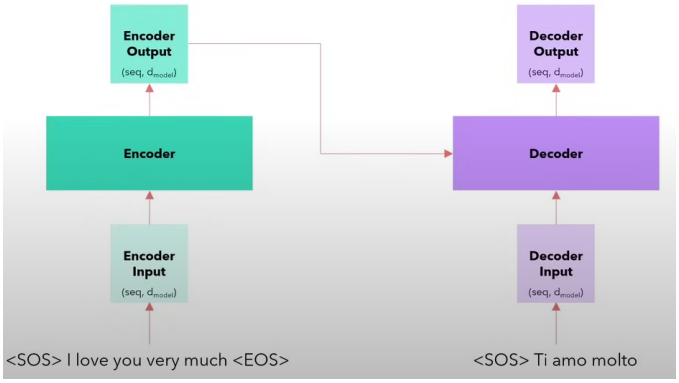
Encoder:



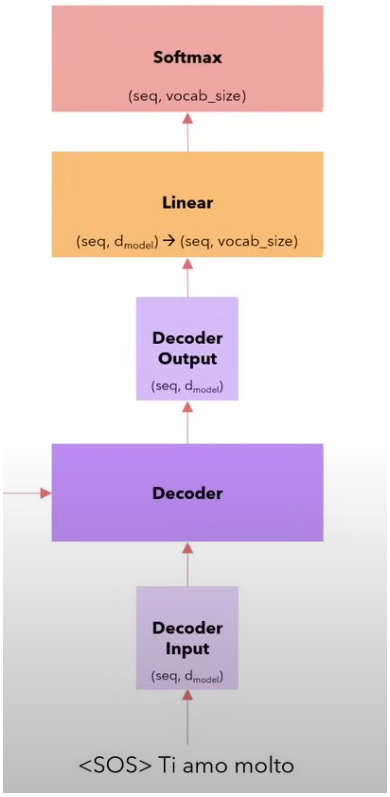
- We append 2 special tokens to our English input sentence:
  - **<SOS>**: Start of sentence (tells the model the start position of a sentence)
  - **<EOS>**: End of sentence (tells the model the end position of a sentence)
- The sentence becomes:
  - <SOS>I love you very much<EOS>
- We convert it into embeddings, add the positional encoding to form the encoder input of dimension $seq * d_{model}$.
- We pass the encoder input to the encoder to get the encoder output.
- The encoder output for each word is a vector that not only captures it's meaning or the position, but also its interaction with other words, with the help of multi-head attention
- The encoder output is of the same dimensions as the original encoder input: $seq * d_{model}$.

Decoder:

- We add a <SOS> token to the expected Italian translation:
  - <SOS>Ti amo molto
- As we add the <SOS>, the output is shifted right, as expected by the model.
- The transformer expects each decoder input sequence to be of the same length.
- Our input is of 4 tokens (including <SOS>), so we add 996 padding (<PAD>) tokens.
- We pass this input now as the decoder input.
- It is converted to output embeddings and we add positional encoding to it to form the input to the masked multi-head attention.



- We pass the keys and values from the encoder output and the query from the output of the masked multi-head attention layer to the decoder (multi-head attention + feed forward layer). We get the decoder output.



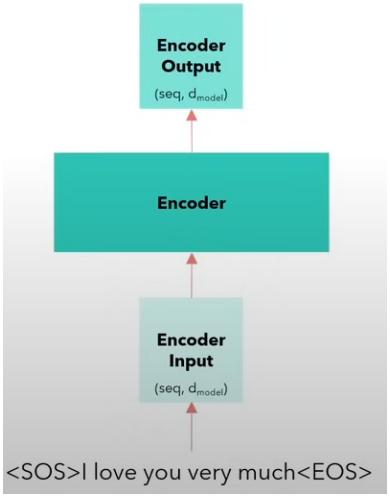
- The decoder output is of the dimension $seq * d_{block}$, it is still an embedding. We don't know the actual words/tokens from our vocabulary as of yet.
- The linear layer will now convert the $seq * d_{block}$ matrix into a $seq * vocab\ size$ matrix and we apply softmax to it.
- Now, we have the output sequence with the matrix giving the corresponding position of that word in our vocabulary.
- Now we have the output. The expected output is:
  - Ti amo molto <EOS?
- This expected output is called the label or target.
- The model may or may not output the expected output. If the model doesn't predict this, we need to know how wrong our model is to improve.
- We use cross-entropy loss as the loss function for transformers.

Transformer Training Advantage:

- All of this happens in 1 time stamp for the entire sequence (unlike previous architectures like RNNs which required *n* time steps).
- So transformers made it very easy and fast to train extremely long sequences with great performance.

Inferencing a Transformer:

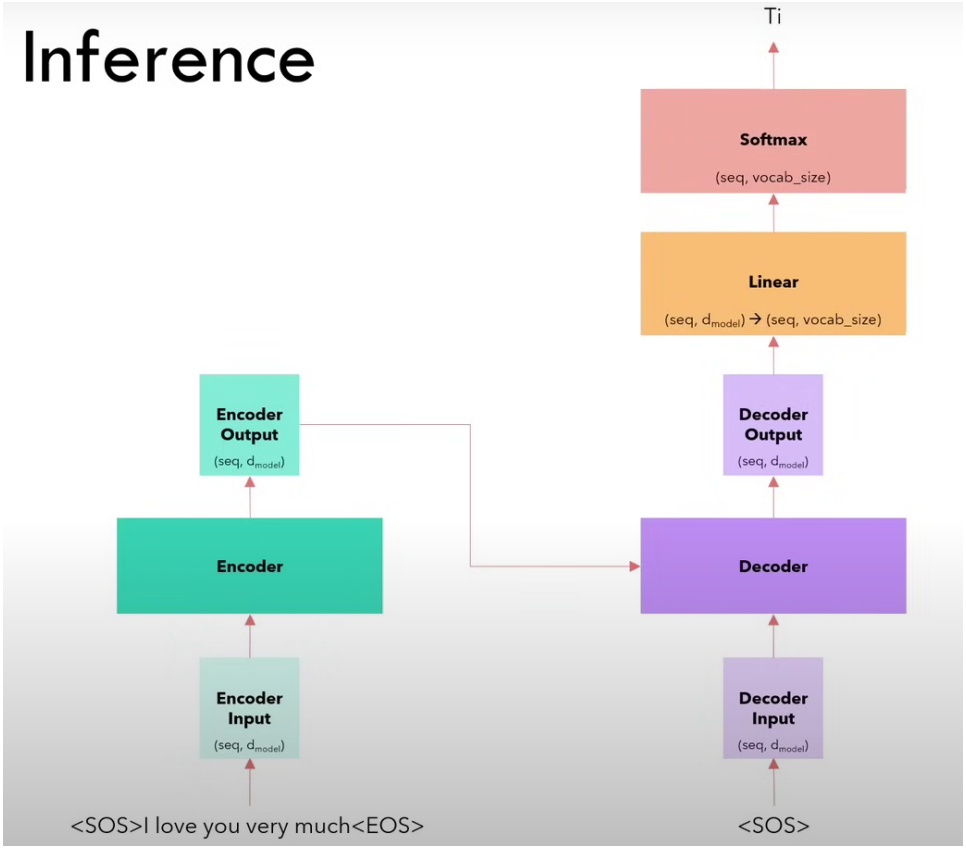
Encoder:



- The encoder part stays the same as training, we provide the input: <SOS>I love you very much<EOS>

Decoder:

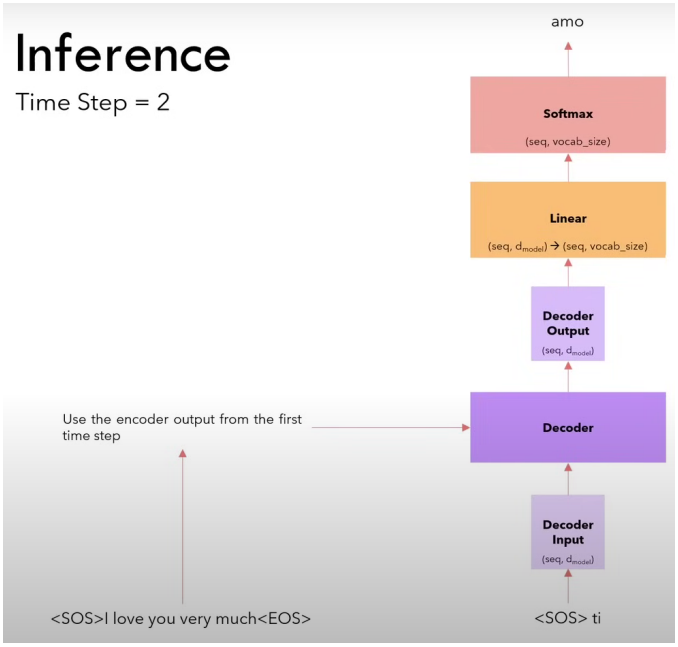
Time Step 1:



- Instead of providing the entire translation preceded by <SOS> and followed by padding <PAD> tokens as during training, we provide only <SOS> as the decoder input, **followed by no padding tokens** during inferencing.
- As before, we get the key and value matrix from the encoder output and the query matrix from the masked multi-head attention layer's output for the input of the decoder.
- Then, we get the decoder layer, which is passed through the linear layer and softmax.
- The output of the linear layer is known as logits. The softmax function selects a token from our vocabulary corresponding to the position of the token with the maximum value, i.e. the token with the highest probability is selected as the model's predicted output *ti*.
- We get the first token (which follows <SOS>) in the 1st time step. This first token that we generated is t*i*.

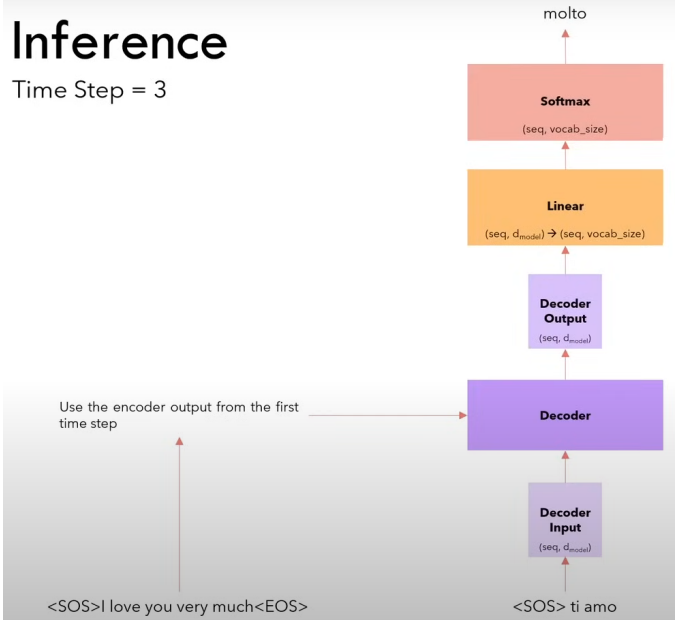
- While the training happens in 1 time step for each sequence, while inferencing, we need *n* time steps.

Time Step 2:



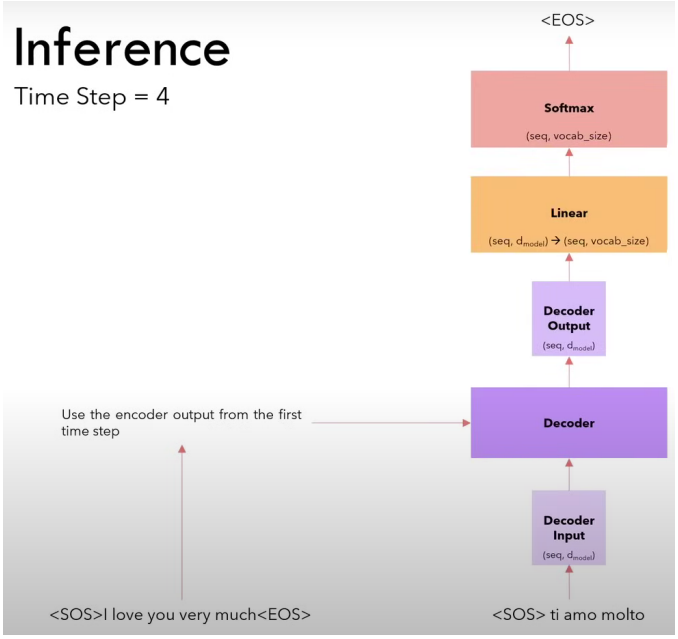
- In the 2nd time step, we don't need to recompute the encoder output as our input (English sentence) didn't change, so the encoder output will not change.
- We append the output of the previous step (ti) to the decoder input sequence (<SOS> ti) and feed this as the input to the decoder layer.
- Now, we repeat the same process as the 1st time step, converting it to form decoder input through masked multi-head attention, pass it to the decoder, convert it's output by passing it through the linear and softmax layer to get the next token.
- The next token generated is amo.

Time Step 3:



- We get the next token molto when we pass <SOS> + decoder output until now (ti amo).

Time Step 4:



- Now, we get the <EOS> token.

- As we get the <EOS> we stop and no more tokens are generated.

Inferencing Strategy:

- At every step we selected the word with the maximum softmax value, this is called a greedy strategy and usually doesn't perform really well.

Beam Search:

A better strategy is called **beam search**:

- At each step, instead of selecting the word with the maximum softmax value, we choose the top *B* words and evaluate all the possible next words for each of them.
- We keep the *B* most probable sequences for each prediction.
- Beam search example:
- Beam size B = 2:
    1. Start with <SOS>.
    2. Generate the top 2 tokens: A and B.
    3. For each sequence <SOS> A and <SOS> B, generate the next top 2 tokens:
        - <SOS> A might generate sequences <SOS> A C and <SOS> A D.
        - <SOS> B might generate sequences <SOS> B E and <SOS> B F.
    4. Evaluate and keep the top 2 sequences among <SOS> A C, <SOS> A D, <SOS> B E, and <SOS> B F.
    5. Repeat this process until the sequences are complete.
- While being computationally more intensive than greedy search, the performance is better as it explores more possibilities.
- Choosing the optimal beam size becomes optimal as a small *B* might miss the optimal solution but a larger *B* might be too compute intensive.

- Modern LLMs like ChatGPT uses beam search.

How LLMs like ChatGPT are trained using Transformers:

- LLMs are trained in 2 steps:
    1. **Pre-training:** Training the LLM on a large corpus of text data to learn general language patterns and representations.
    2. **Fine-tuning (Instruction-tuning):** To adapt the pre-trained model for a specific like question answering. We achieve this by training the model on a smaller task specific dataset.

Pre-training and Transformers:

Input to Encoder:

- **Input**: A sequence of tokens representing a segment of text from the training corpus.
- **Example**: "The quick brown fox jumps over the lazy dog."
- **Tokenization**: Each word is tokenized, and special tokens such as <SOS> (start-of-sequence) and <EOS> (end-of-sequence) may be added as needed.
- **Tokenized Input**: <SOS> The quick brown fox jumps over the lazy dog.

Output from Decoder:

- **Output**: The decoder does not produce output during pre-training because the pre-training tasks are usually self-supervised.
- **Objective**: The model is trained to predict masked or randomly replaced tokens in the input sequence. For instance, a certain percentage of tokens

in the input sequence are replaced with a special <MASK> token, and the model is trained to predict the original tokens.

- **Example (Masked Language Modeling)**:
  - Input: "The quick brown fox jumps over the lazy dog."
  - Masked Input: "The quick brown [MASK] jumps over the [MASK] dog."
  - Expected Output: "The quick brown fox jumps over the lazy dog."

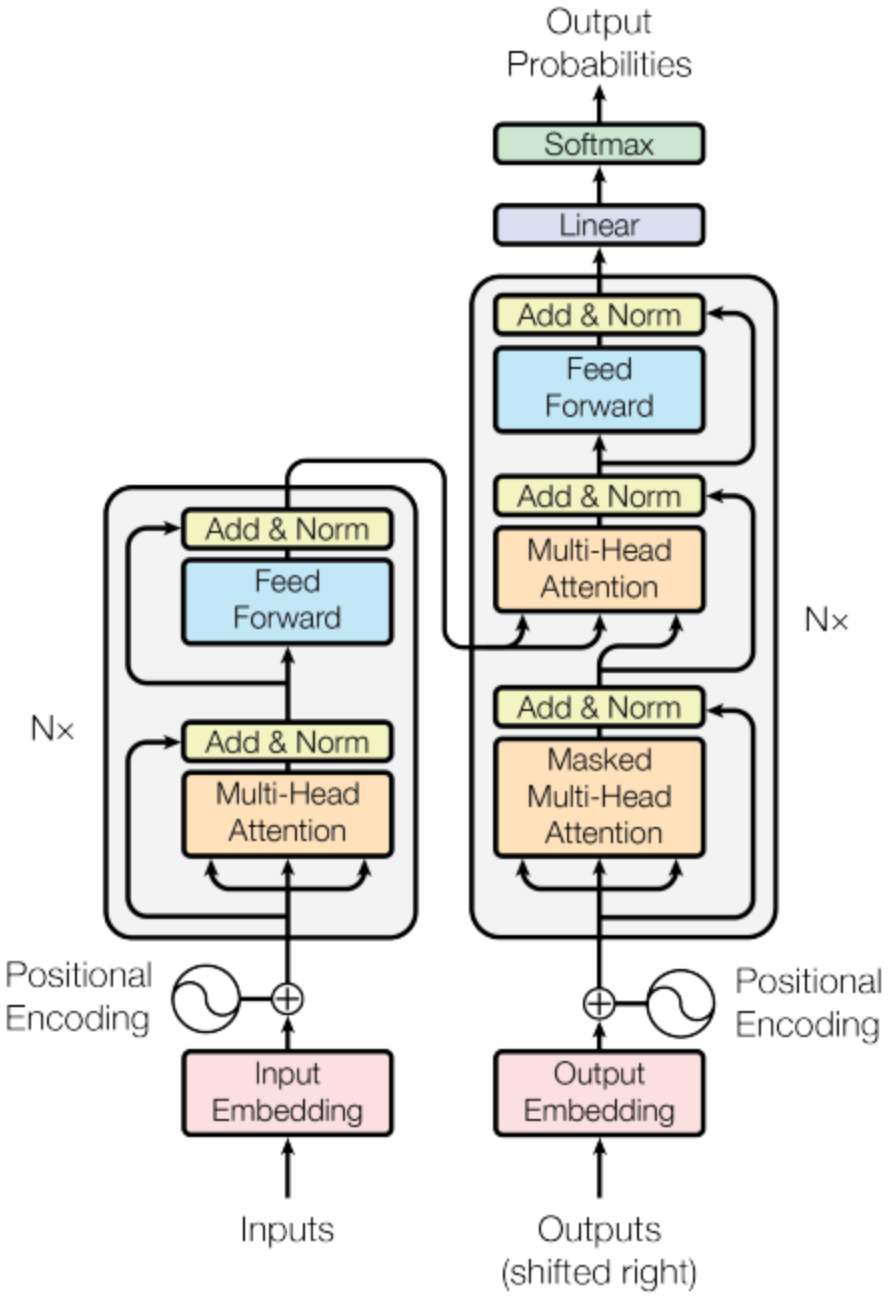
Fine-tuning (Instruction-tuning) and Transformers:

Input to Encoder:

- **Input**: A sequence of tokens representing the user input or conversation history.
- **Example**: "User: How are you?"
- **Tokenization**: Tokenize the input sequence and add special tokens as needed.
- **Tokenized Input**: <SOS> User: How are you?

Output from Decoder:

- **Output**: The output from the decoder corresponds to the desired response or target response in the conversation.
- **Objective**: The model is fine-tuned to generate appropriate responses based on the input context.
- **Example (Conversational Response)**:
  - Input: "User: How are you?"
  - Target Response: "I'm doing well, thank you."


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)