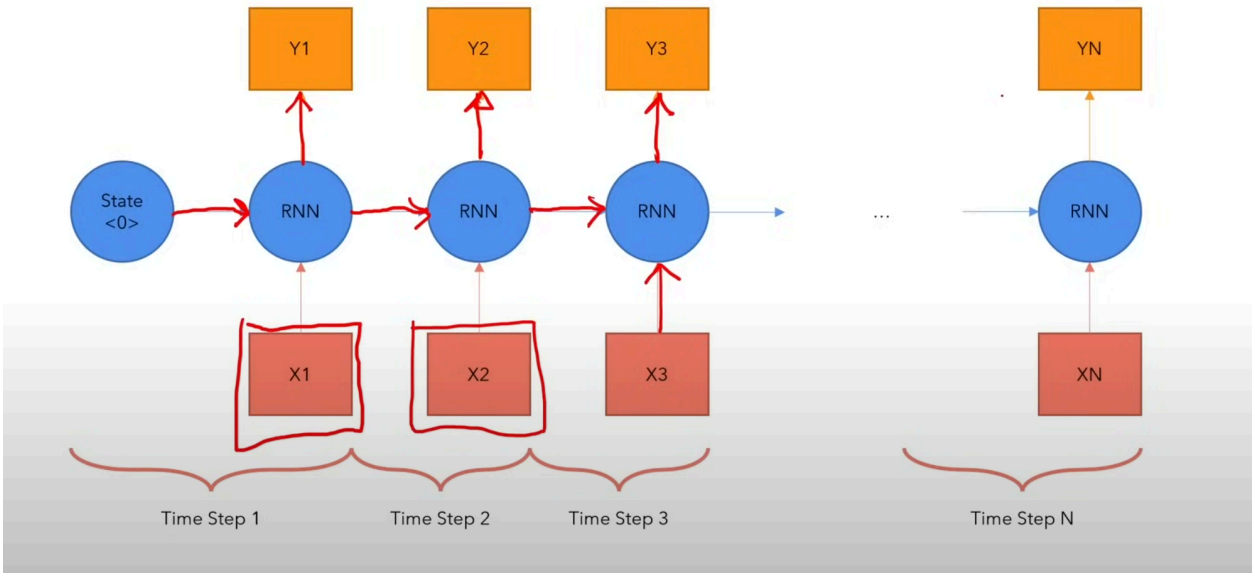
00:00 - Intro

01:10 - RNN and their problems

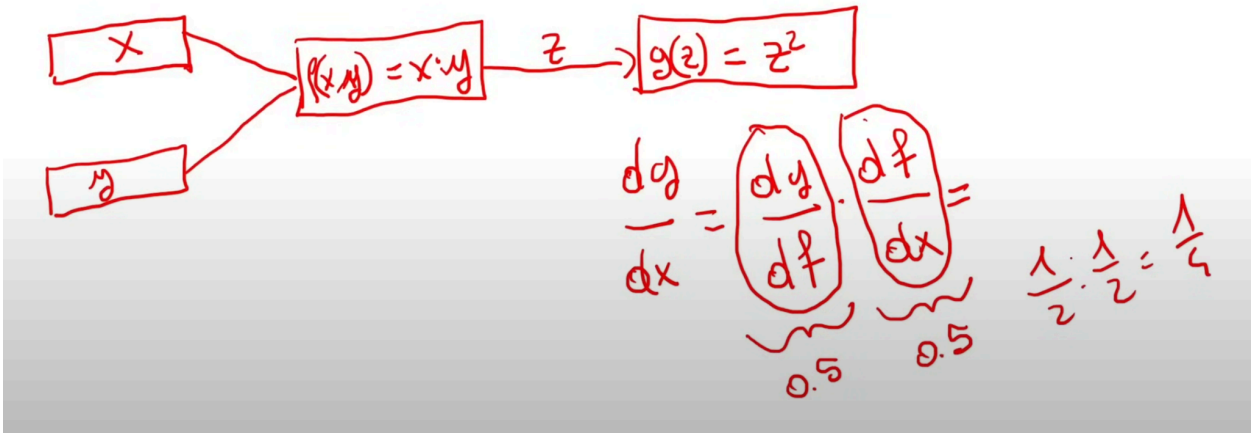
RNNs



Sequence to sequence

Input at a particular time step and hidden state of a previous time step -> output at a particular timestep

1) **Slow computation for long sequences**
2) **Vanishing or exploding gradients**
   Computational graph ->



If a particular node gradient is less than one, when multiplied with another no smaller than one - becomes even smaller

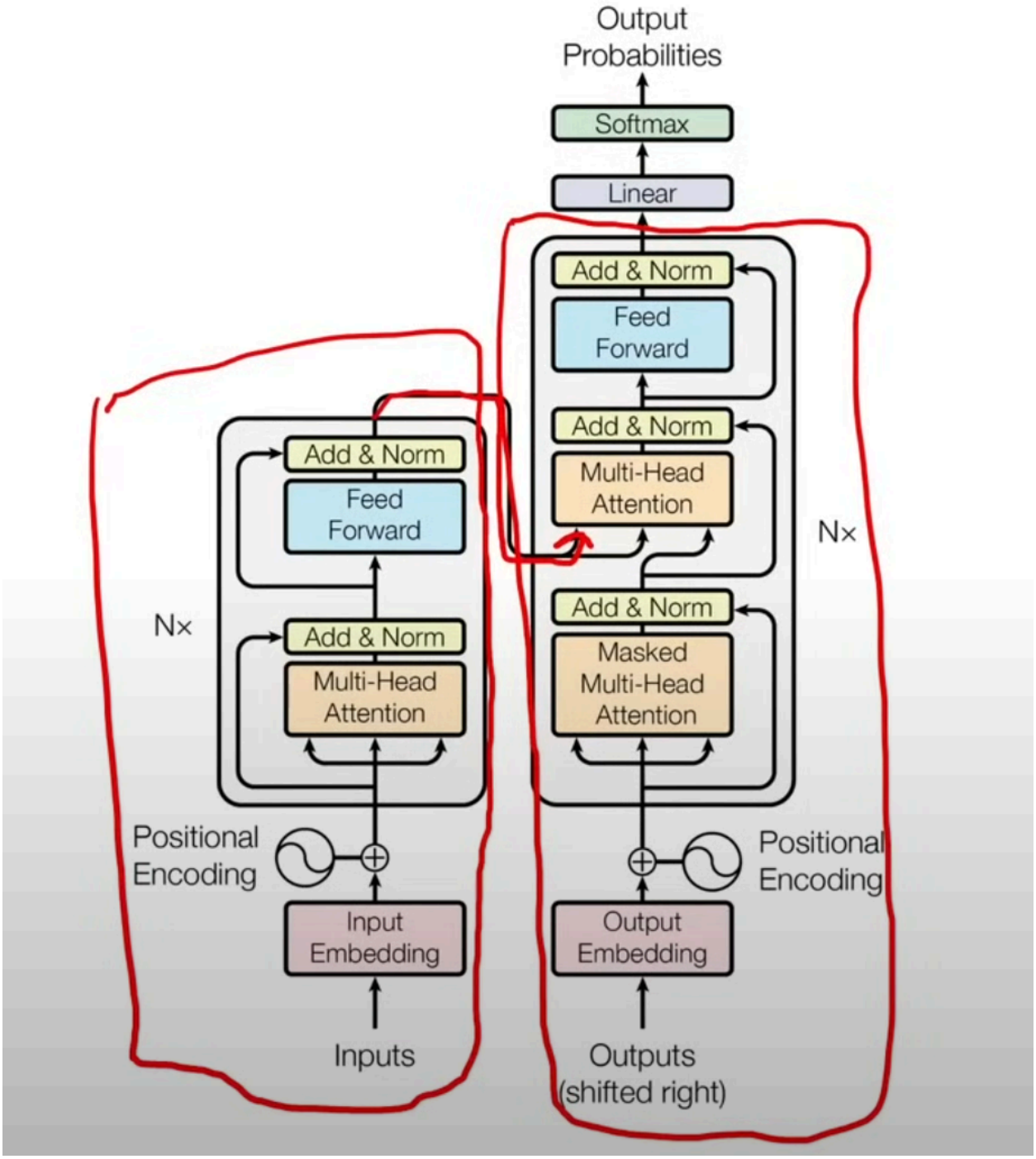
Similarly, numbers greater than one when multiplied with numbers of similar type would be even greater -> Vanishing and exploding gradients

3) **Difficulty in accessing the information from long time ago**

In an RNN, output at a particular time step only depends on the previous time step hidden state, so although a little of information might've been transmitted across, but, more longer the chain, more information is lost

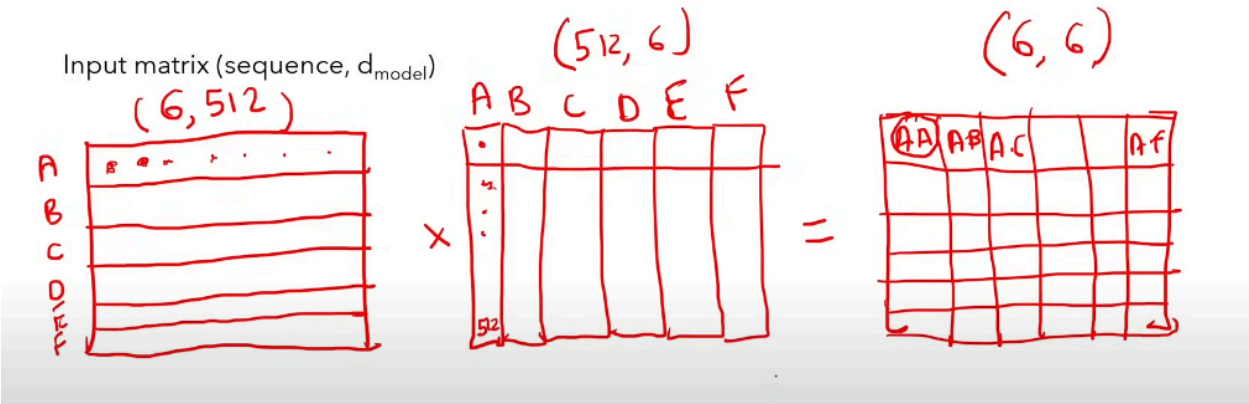
4)

08:04 - Transformer Model



Encoder and decoder, few connections from an encoder to a decoder

09:02 - Maths background and notations



12:20 - Encoder (overview)

12:31 - Input Embeddings

Tokens -> need not be words
Can be even smaller -> splitting words into multiple tokens

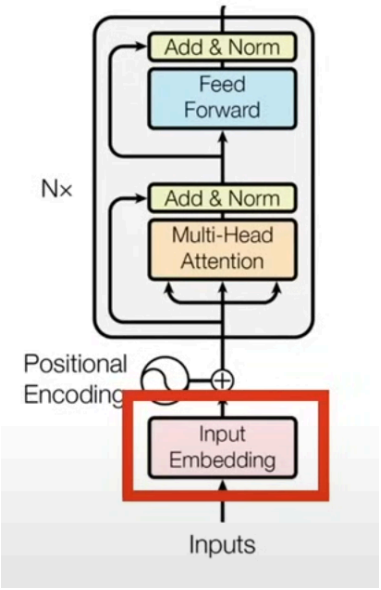
For now, each token -> one word

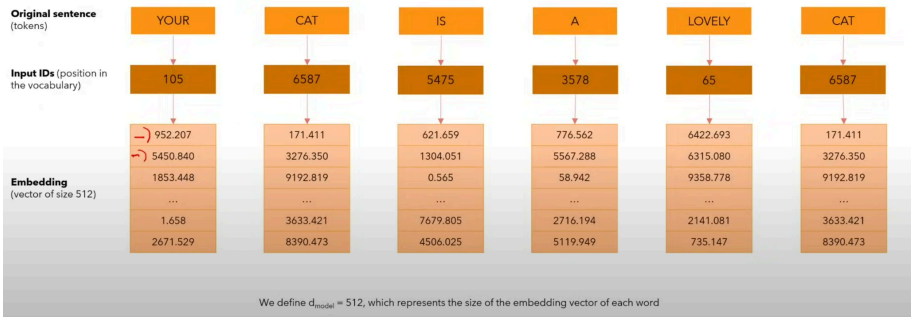
Input IDs -> position in **vocabulary**

d_{model} = 512 (In this example)

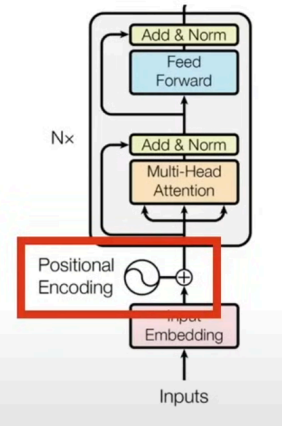
Size of the embedding

Latent factors that identify the meaning and context of words





15:04 - Positional Encoding



Each word should carry some information about its 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”

What is positional encoding?

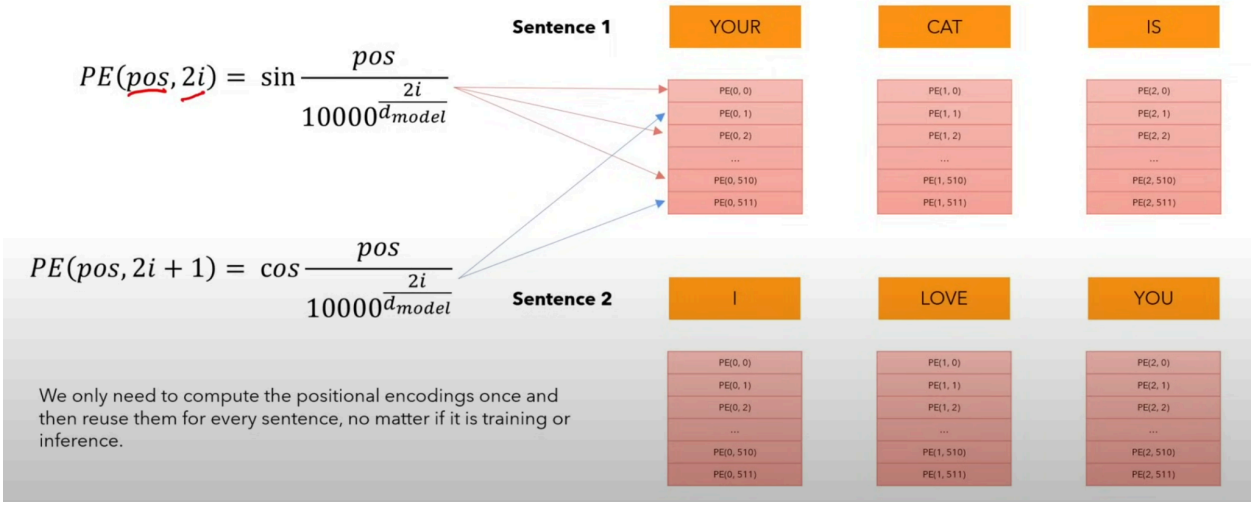
We can see that the word “what” is closer to “is” than to “encoding”. But, for a model everything is a token and it doesn’t have this spatial info that determines how far the words in an actual sentence are. We must give this to the model.

We want the positional encoding to represent a pattern that can be learned by the model.

Just like an embedding vector sequence of some size, we also create a positional encoding vector sequence of the same size. We compute the values of the positional encoding sequence using the following two trigonometric expressions.

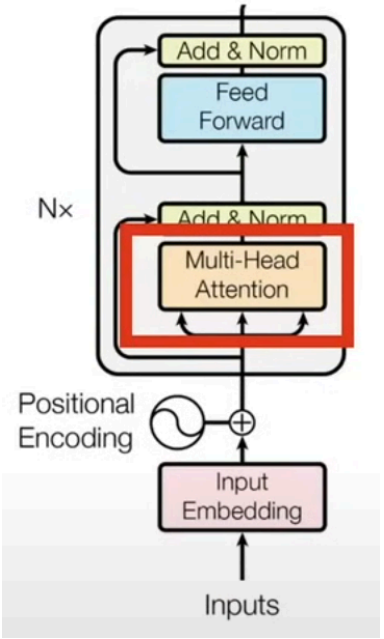
Encoder input is the sum of the input embeddings of a specific sentence and corresponding position embeddings. For a different sentence in the dataset, input embedding of the corresponding new sentence is added with the same positional encoding vector.

Note that, in the normal embedding, the word CAT has the same vector in both the positions in the sentence as the meaning of the cat in both the positions is the same. However, to signify that they are in different positions in the sentence, we have added the position embedding and this gave us two different position embedding vectors and thus different Encoder inputs.



We only need to compute the positional encodings once, and we use the same for every sentence, no matter if it is training or inference.

20:08 - Single Head Self-Attention



Self-Attention ?

Relate words to each other

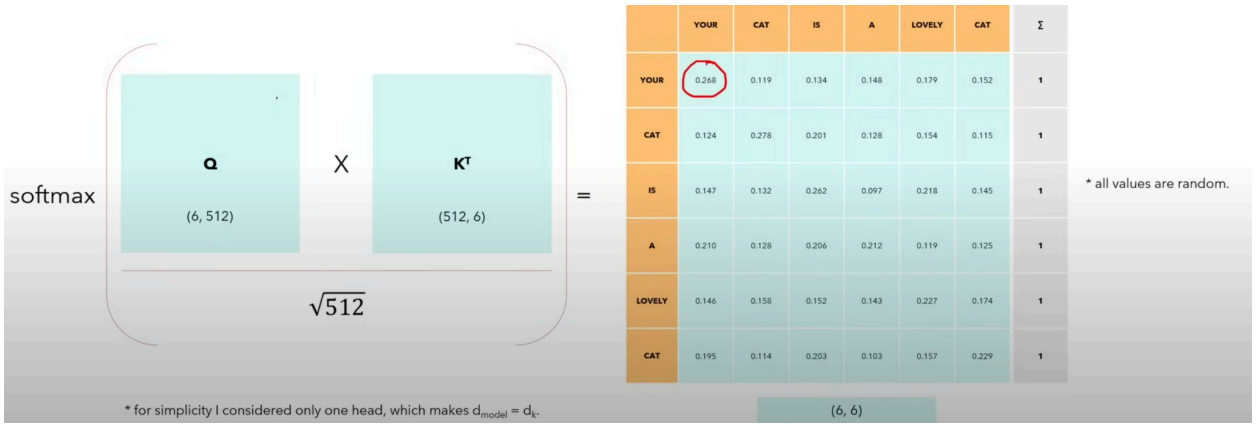
Input embeddings -> capture the meaning of the word

Positional encoding -> understand the position of the word in the given sentence

Imagine seq = 6, d_{model} = 512

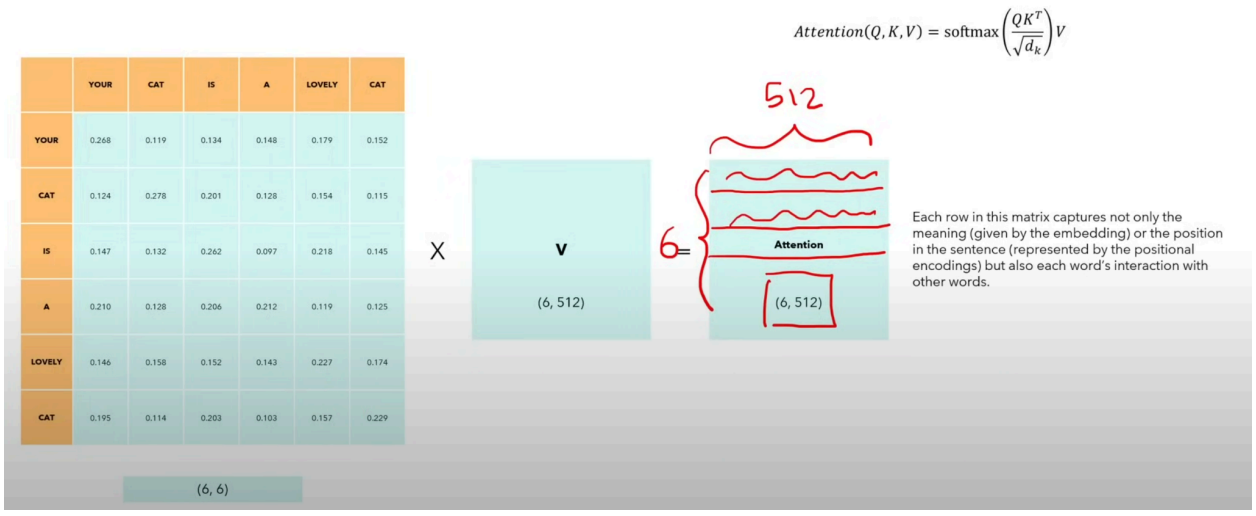
Q,K and V are the same input matrix !!!

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



We can see that when Q is multiplied with the transpose of the same matrix, divided with a constant value, we get a (6,6) matrix which when softmax is applied to all the values in a row, add up to value **'1'**.

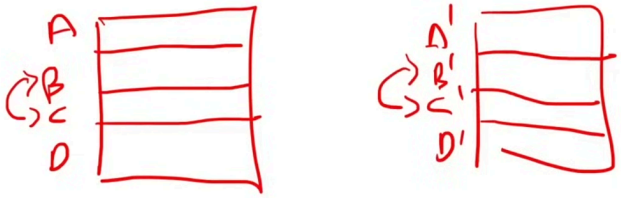
In collaborative filtering example in fastai course, we have created a user embedding and a movie embedding. Then we have cross multiplied both of them to understand how strong a user feels about certain aspects of the movie. So, basically to get a score or a rating of the user for the respective movie. Similarly, the above (6,6) matrix represents the relation of a word with respect to another word, higher the score meaning stronger the relationship with the other word. The value in the red circle in the above image is a result of matrix multiplication, in other words a dot product of the position encoded embedding of "YOUR" with itself. The next value in the matrix is the dot product of the "YOUR" positional encoded embedding with "CAT" positional encoded embedding.



Now, we can see that after multiplication with “V”, we regain the initial shape of input. But, in addition to capturing the meaning and positional encoding of every word, we also get the relation of each word with every other word.

Carefully observe the above matrix multiplication. We get the scores indicating the relationship between different words included into the **final special embedding** along with the meaning and positional encoding. This can be considered as our special embedding of the input sentence with all the above values.

- Self-Attention is permutation invariant.



 # Change in order of words just changes the order of results, but doesn't change the values.

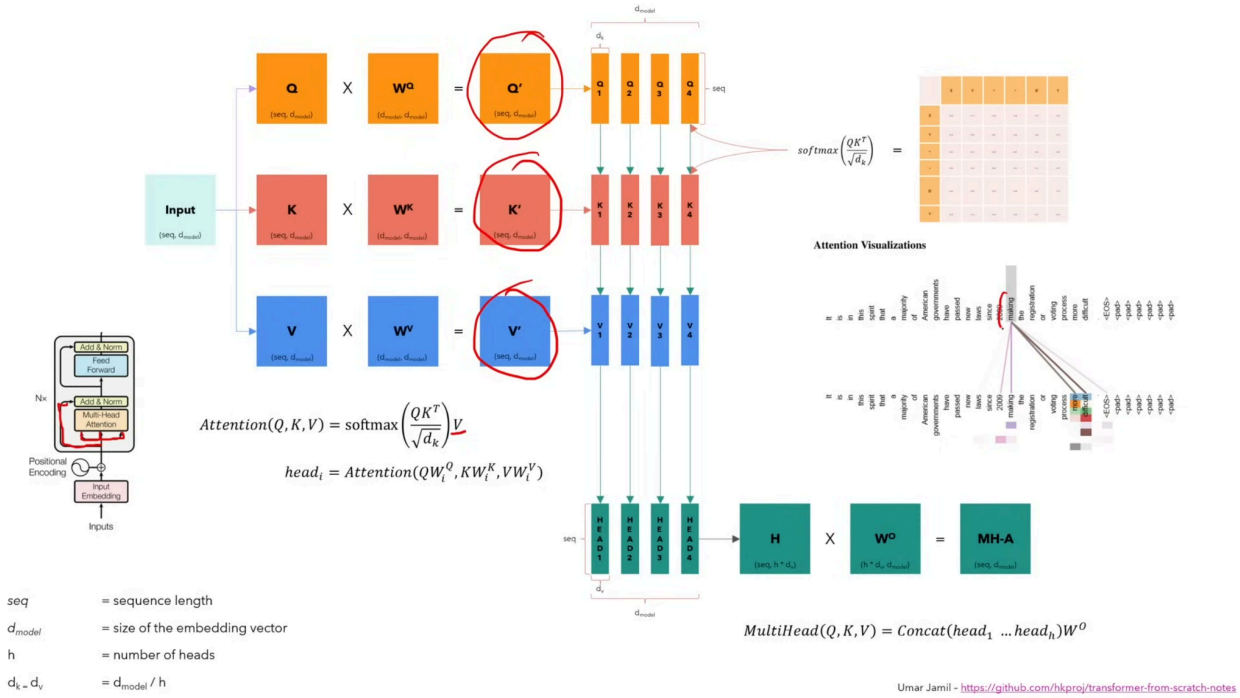
 # Self attention requires no parameters. (The only parameters are embeddings and positional encodings. But, self-attention doesn’t introduce new parameters as even d_k in the below formula is a constant.)

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

 # We expect the values along the diagonal to be highest as they represent the relation of the words with themselves.

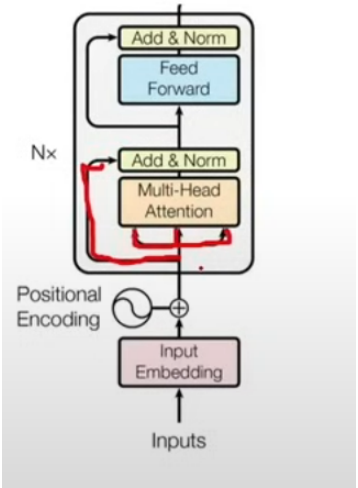
#If we want the model to not consider a specific word particularly, we can make the corresponding embedded and encoded values of the word -inf before applying softmax. We use this in decoders.

28:30 - Multi-Head Attention



Umar Jamil - https://github.com/hkproj/transformer-from-scratch-notes

Input is duplicated as four matrices - Query Q, Key K and value V and one copy is passed forward as shown in the subsequent figure. Now, each matrix (obviously of the same shape (seq,d_{model})) is multiplied by a parameter matrix of shape (d_{model},d_{model}) and this gives us new matrices Q', K' and V' of dimensions (seq,d_{model}).



Now each of these matrices are divided across the embedding dimension into multiple heads. So, each head contains all the words, but only a few of the embedding values. So, each head sees the word differently based on the embedding values it is using. In this example, we are using 4 heads. So, h=4.
$d_k = d_v = d_{model}/h$ is the size of each head's embedding.

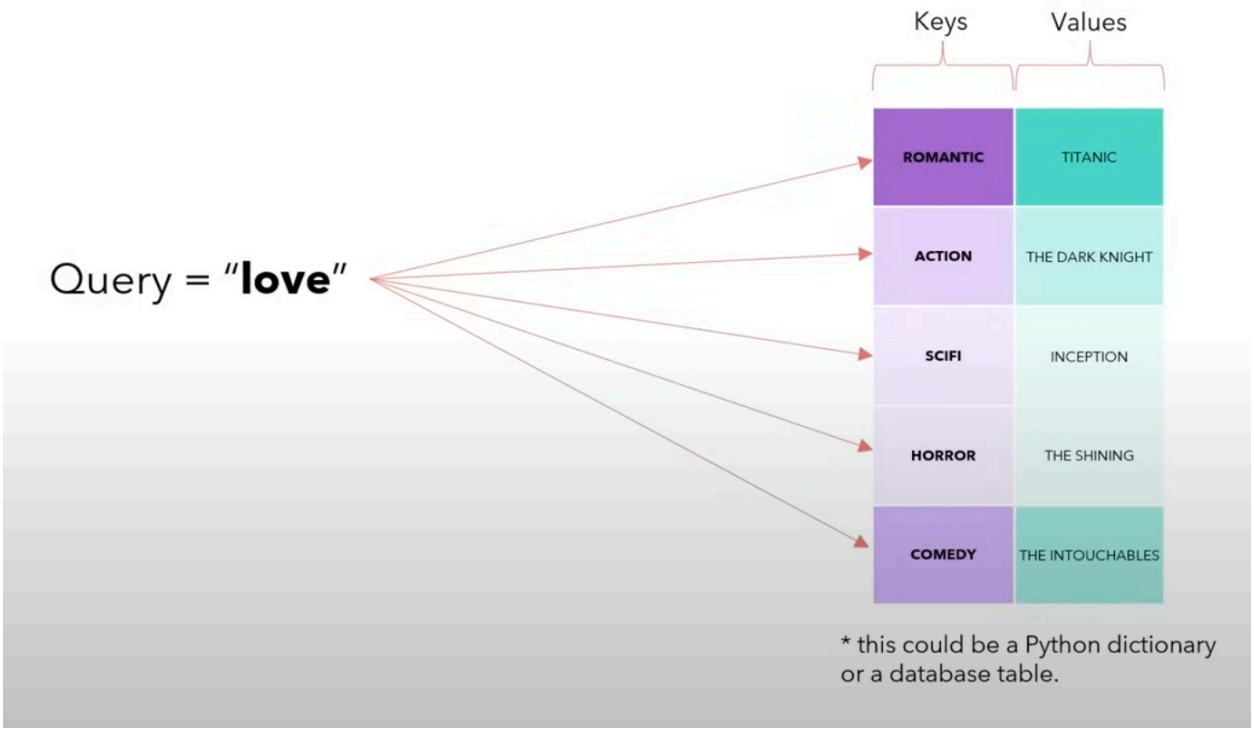
Now attention is applied to each Q', K', V' across different heads. head1, head2, head3, and head4 are the matrices of dimension (seq,d_v) that are generated as a result of this attention process across each head of.

Now, concatenate along d_k dimension and this gives us the matrix H of dimension (seq,d_v*h) or (seq,d_{model}).

This is multiplied by another parameter matrix to give the MH-A output.

In the blue box, in the above diagram, we can see that relations of different words with another words using the score calculation using Q and K as seen in the self-attention section. However, we can see that the word “making” is related to different words at different heads as each head only has access to certain aspects/embeddings of words.

35:39 - Query, Key, Value



For instance, Keys are themes of the movies and Values are the movies or the corresponding themes. Now, if the query is **”love”**, it is actually understood using the input embedding for instance, 512 values that indicate different aspects of love. Then a dot product of the query with the **“key”**, basically Q*K’ gives us the scores, thereby revealing what themes are more in line with the given “Query” and are more far away(that have softmax value ~0).

37:55 - Layer Normalization

How is layer normalization different from batch normalization?

Batch normalization: Normalizing each feature independently across all the samples or batches. This makes sure that all features are within the same range of -1 to 1 and thus makes sure the gradients are not very big and biased for the features that are larger (As if there is a feature that is in the order of 1000s and another feature that is in the range of 1s, a unit change in both the features represent a very different quantity).

What is layer normalization?
Layer normalization is normalizing across all the features in a single sample. Thus, this involves taking the mean and standard deviation of all the features in a particular sample and normalizing using them. Then we apply scaling and shifting to the normalized values.

We also introduce two parameters, usually called **gamma** (multiplicative) and **beta** (additive) that introduce some fluctuations in the data, because maybe having all values between 0 and 1 may be too restrictive for the network. The network will learn to tune these two parameters to introduce fluctuations when necessary.

$$y_j = \gamma \hat{x}_j + \beta$$

How does it make sense to use the layer normalization? How do we get useful information and insights from normalizing across different features in a sample?

Take this example where we apply layer normalization to a text understanding scenario.

Step 1: Tokenize a Sentence

Suppose we have a simple sentence:

- Sentence: "The cat sits."

This sentence is tokenized into words (tokens):

- Tokens: ["The", "cat", "sits"]

Step 2: Embed the Tokens

Each token is converted into a vector (embedding) using an embedding layer, like Word2Vec or BERT. Let's assume each word is embedded into a 3-dimensional vector.

Token	Embedding (3D Vector)
"The"	[0.5, 0.1, 0.3]
"cat"	[0.7, 0.2, 0.6]
"sits"	[0.6, 0.3, 0.4]

Step 3: Apply Layer Normalization to Each Token's Embedding

Now, for each token, we will apply layer normalization **across its embedding dimensions (3D vector)**.

It is important to note that layer normalization is applied across its embedding dimensions. So, mean, variance and scale and shift happens to each token independently. (For instance, use the values 0.5,0.1 and 0.3 for the first token to calculate their mean and standard deviation.)

Step 4: Use the Normalized Embeddings for Further Processing

The normalized embeddings for each token:

- "The": [1.23, -1.23, 0]
- "cat": [0.93, -1.39, 0.46]
- "sits": [1.34, -1.06, -0.26]

These normalized embeddings can be used for further proceedings to be fed into the neural network. In this example, layer normalization is used to stabilize and normalize the embeddings of tokens in a sentence, allowing the model **to focus on the relationships between tokens rather than being influenced by the varying scales of raw embeddings.** This is particularly useful in tasks like text understanding, where the relative importance and interactions between words are critical.

Handling Inputs of Various Lengths

Scenario: Variable-Length Sentences

Consider two sentences of different lengths:

1. **Sentence 1:** "The cat sits."
    - Tokens: ["The", "cat", "sits"]
    - Number of tokens: 3
2. **Sentence 2:** "The dog runs quickly."
    - Tokens: ["The", "dog", "runs", "quickly"]
    - Number of tokens: 4

Each sentence is tokenized and embedded, resulting in vectors of different lengths (3 tokens for Sentence 1, and 4 tokens for Sentence 2).

How Layer Normalization Helps:

- **Normalization Across Each Token:** Layer normalization operates on each token's embedding vector independently of other tokens in the sequence. For each token, it normalizes across the dimensions of that token's embedding, regardless of how many tokens there are in the sentence.
- **Consistent Application:** Since the normalization is applied per token (and not across tokens), it works consistently regardless of whether a sentence has 3 tokens, 4 tokens, or more. The model can process sentences of different lengths without any need for special handling or adjustments.
- **Example:**

    - For "The cat sits." (3 tokens), layer normalization is applied to the embeddings of "The", "cat", and "sits" separately.
    - For "The dog runs quickly." (4 tokens), layer normalization is applied to the embeddings of "The", "dog", "runs", and "quickly" separately.
- Each token's embedding is normalized based on its own dimensions, independent of the total number of tokens in the sentence.

Handling Various Batch Sizes

Scenario: Small vs. Large Batch Sizes

Consider two batches during training:

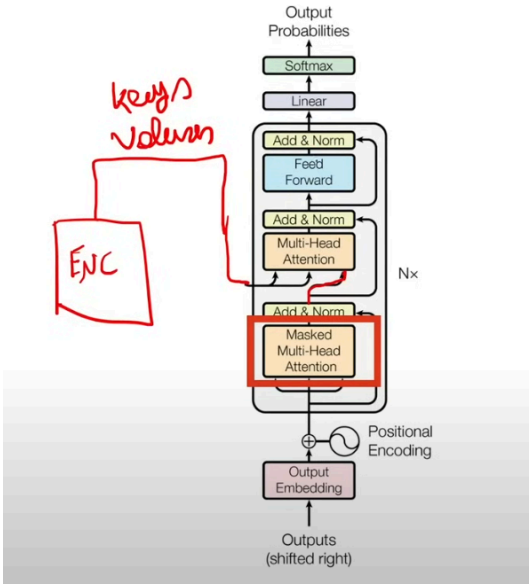
1. **Batch 1:** Contains two short sentences.
    - ["The cat sits.", "The dog runs."]
    - Batch size: 2
2. **Batch 2:** Contains multiple sentences, some short, some long.
    - ["The cat sits.", "The dog runs.", "The bird flies high above.", "The fish swims quickly in the clear water."]
    - Batch size: 4

How Layer Normalization Helps:

- **Independence from Batch Statistics:** Unlike batch normalization, which relies on statistics computed across the entire batch (like mean and variance of the batch), layer normalization computes mean and variance **within each token's embedding**. This means the effectiveness of normalization doesn't depend on the number of samples in the batch.
- **Robustness to Batch Size Changes:** Whether you have a batch size of 2 or 200, layer normalization will behave consistently because it normalizes within each sample (each token in this case) independently. This makes it highly robust to changes in batch size.
- **Example:**
    - **Batch 1 (Small Batch):** The embeddings for the tokens in "The cat sits." and "The dog runs." are normalized independently, without any dependence on the other sentence in the batch.
    - **Batch 2 (Large Batch):** Even with a larger batch containing more diverse sentences, layer normalization still applies per token, ensuring consistent treatment of each token's embedding without being influenced by the other tokens in the batch.

40:13 - Decoder (overview)

Decoder -> Instead of input embeddings it has **“output embeddings”** at its root. Then they undergo **positional encoding** and then are passed into **“Masked multi-head attention”**. Then they are **concatenated** and **layer normalized**. Now, these are given as **“Queries”** to the **multi-head attention layer** and the **“Keys”** and **“Values”** to the **multi-head attention layer** are provided by the **encoder**. Hence, this is not self-attention anymore and is called **“cross-attention”**. After this, it is still in the size of (seq,d_{model}) and hence it is passed through a **linear layer** and then through the **softmax layer** to give us **output probabilities.**



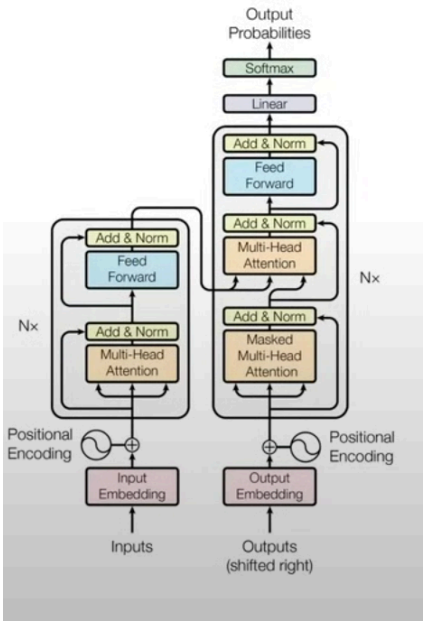
42:24 - Masked Multi-Head Attention

Our goal is to make the model causal: it means the output at a certain position can only depend on the words on the previous positions. The model must not be allowed to see the future words.

We remember how we have split our entire Q’,K’ and V’ into multiple heads that each see only a part of the embedding. And then we perform the matrix multiplication of respective Q and K in each head and divide them by the square root of the dimensions of the model and pass the entire thing into a softmax layer to be later multiplied with V’. However, before passing the entire scores to the softmax, we make all the upper diagonal elements **-inf**, which represent the words that are in future positions of the respective word of interest in each token case. We then apply softmax to this modified matrix and all these **-inf** values are computed to be zeroes.

44:59 - Training

Our goal is to understand the given English sentence and then translate it into an Italian sentence.



1. Encoder
The English sentence on which we append two special tokens in our vocabulary <SOS> and <EOS> is given as input to the encoder.

And then it is made into (seq,d_{model}) after adding input embeddings and positional encoding. Then after passing through the encoder block (MH-A), we get a **special embedding** as output of the same dimensions (seq,d_{model}). This special embedding has the information about, the meaning of each word, position information of each word and relation of each word with other words in the sentence (as it undergoes MH-A self attention).

2. Decoder
We need to convert the English sentence into an Italian sentence and thus we prepare the sentence Ti amo molto and prepend <SOS> to the beginning(Outputs shifted right in the above diagram shows that we prepend this <SOS>).

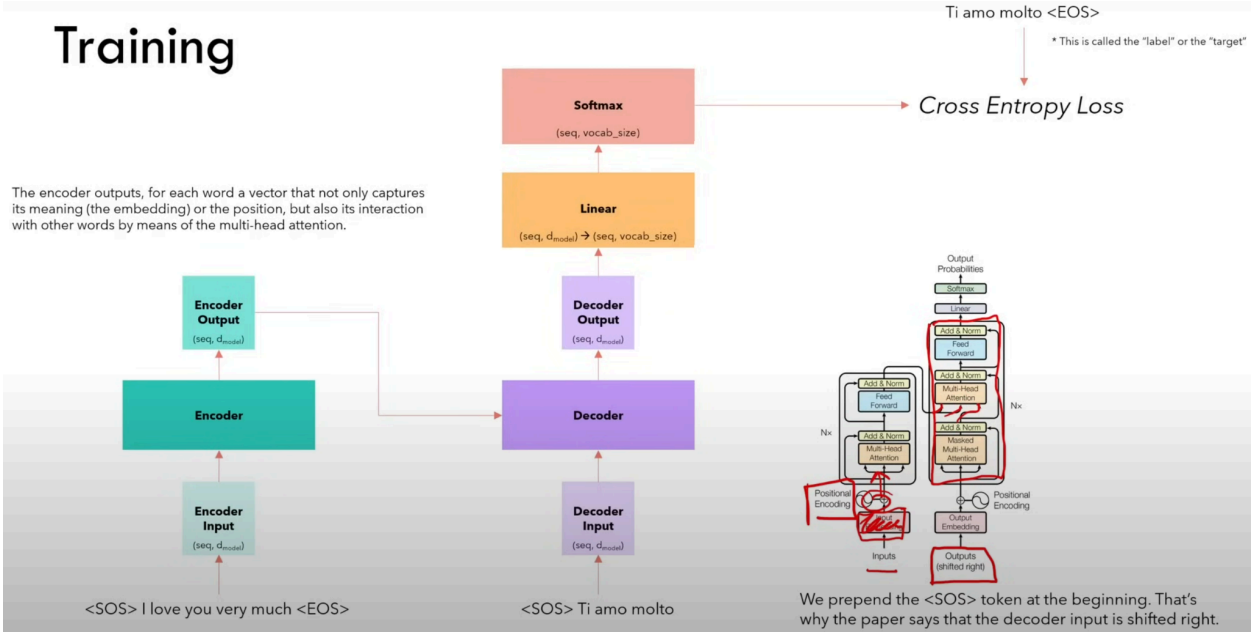
We normally add **'padding'** to this input sequence to make sure all the sequences handled by the transformer are of unit length.

Then after adding embeddings and positional encoding information, they are sent to "Masked MH-A" (causal mask) and then after layer normalization, they are sent to MH-A as queries where keys and values are provided by the encoder output(special embedding).

After a series of steps like shown in the diagram like layer normalization and fully connected layers followed by layer normalization once again, we are left with a decoder output of (seq,d_{model}). Now, how can we understand this embedding corresponding to a word from our dictionary or vocabulary?

We use a linear layer that modifies (seq,d_{model}) into (seq,vocab_size). So, it will tell for every embedding we see, what is the token in our vocabulary that it corresponds to. We then pass it through a softmax layer to see what words or tokens have the highest probability.

Now, we have an output of the model and we know that the model is supposed to give the output Ti amo molto <EOS> -> label/target. We calculate the cross entropy loss between the two and backpropagate the loss derivatives across the weights.



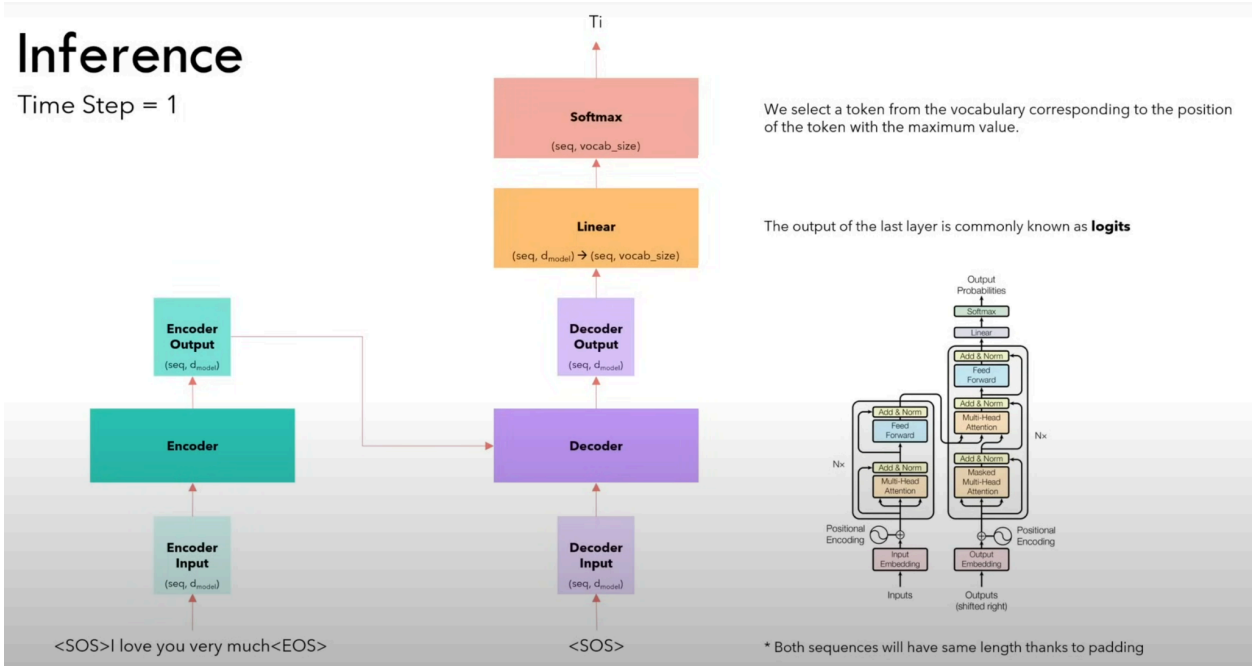
We understand the importance of <SOS> and <EOS> tokens in inference.

All the training happens in one time step !!! as opposed to a RNN which takes n time steps to deal with sequences of length 'n'. This is the reason we can use transformers on very long sequences (sentences) and still get results without that big of a problem.

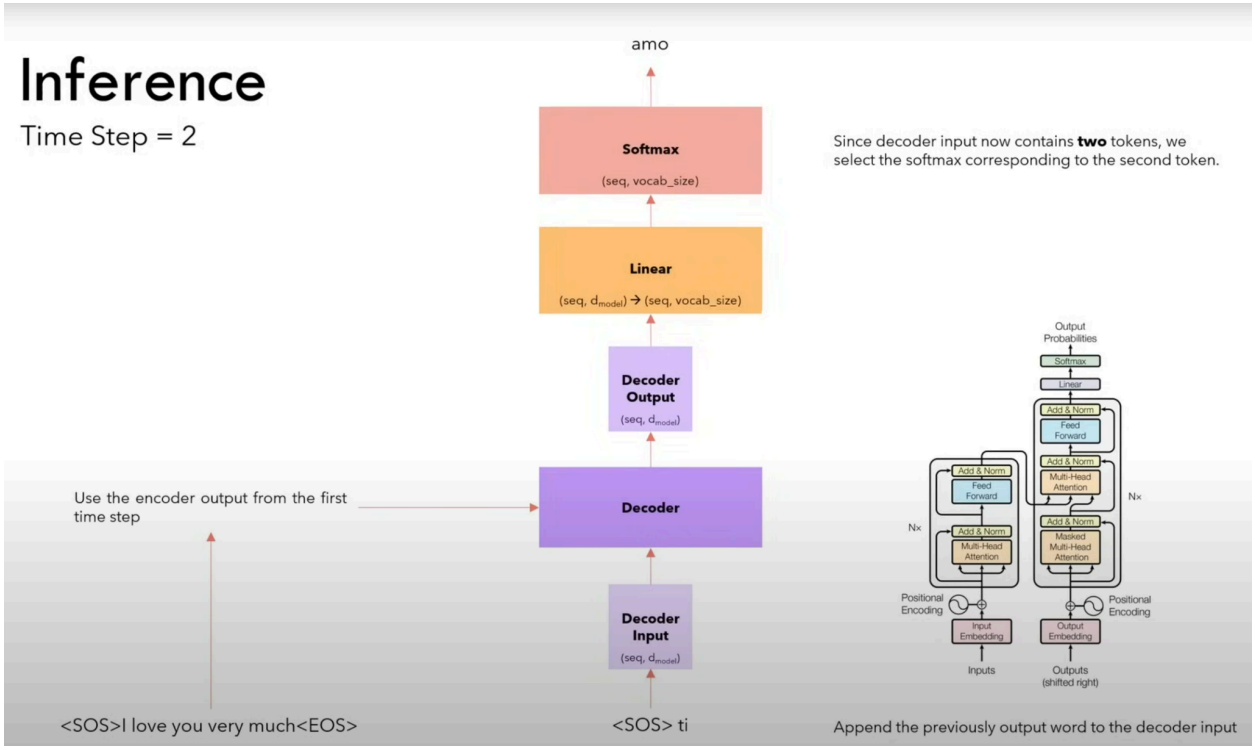
52:09 - Inference

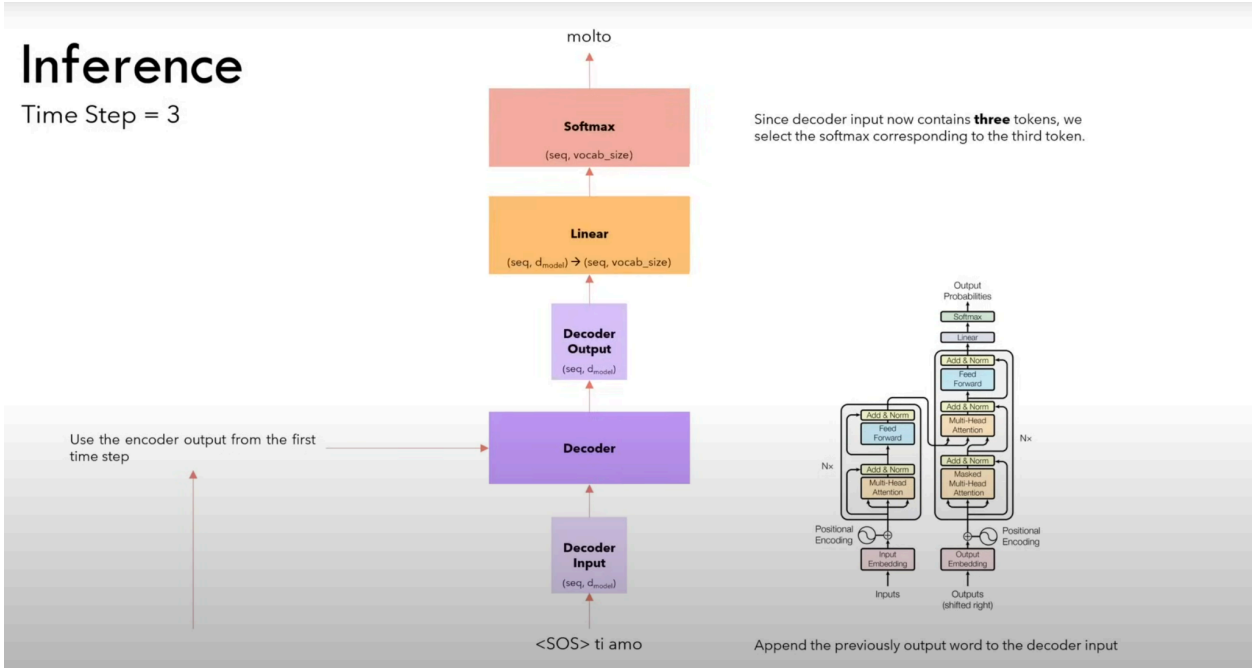
In inference, we give the input sentence initially to the encoder after dividing it into tokens, adding input embeddings and positional encoding. We then pass it as input to the encoder and we get a special embedding that understands the meaning, position and relation between words as output.

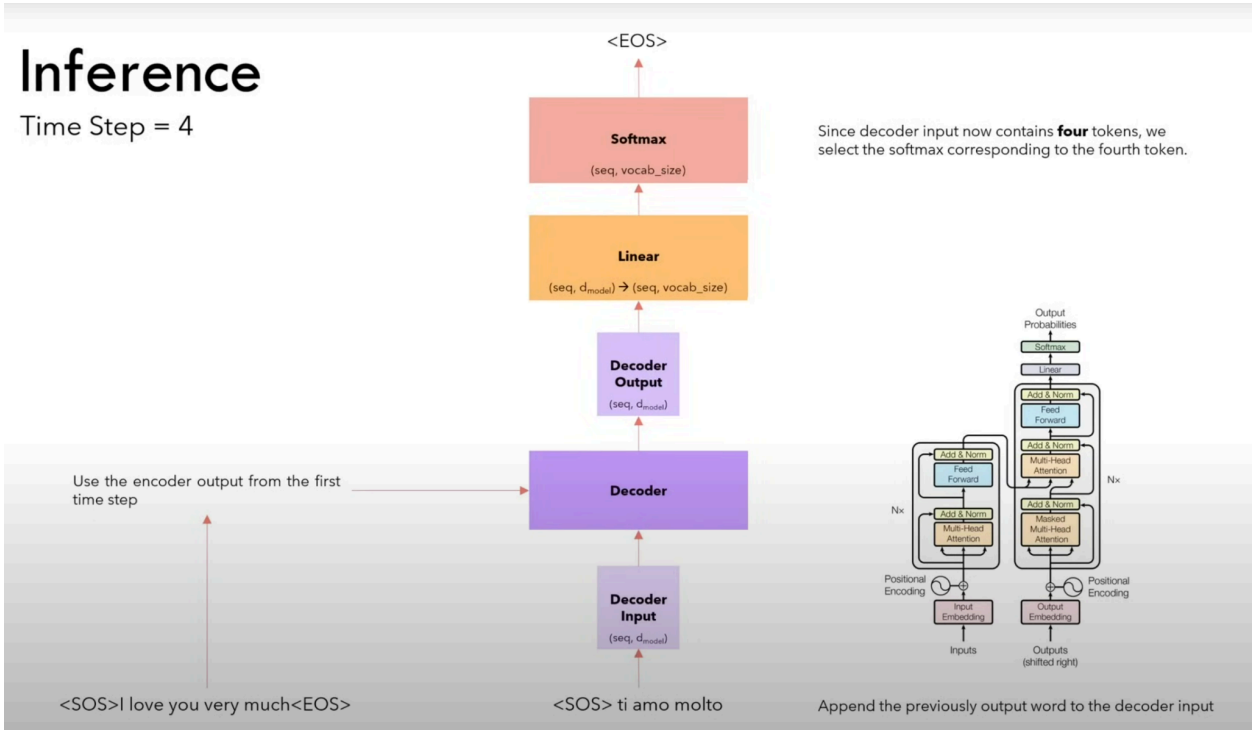
Since, we do not have the output sentence in our inference, we just give the padded <SOS> token's embeddings as the inputs to the decoder. Then the decoder will take the key and values from the special embeddings given by the encoder and will give a decoder output. Then, we need a linear layer to project into vocabulary by outputting logits and after passing through softmax and getting the word with the maximum probability, hopefully, we get the output token "Ti" if the model is trained correctly. **This happens in one time step.**



Now, at timestep2, we don't need to recalculate encoder output again as the English sentence is still the same. We can use the encoder output special embedding from the first time step here as key and values in the decoder. We take the output of the previous sentence from the decoder, append it to the existing input to the decoder and then get a new output token.







The inference stops when the model outputs EOS.

Inference strategy

- We selected, at every step, the word with the maximum softmax value. This strategy is called **greedy** and usually does not perform very well.
- A better strategy is to select at each step the top *B* words and evaluate all the possible next words for each of them and at each step, keeping the top *B* most probable sequences. This is the **Beam Search** strategy and generally performs better.

Implementation of the paper

1. **Input embeddings**
   Multiplied by sqrt(d_{model})
2. **Positional encoding**
   sin and cos formulae modified a little bit to log space
3. **Layer normalization**
4. **Feed forward**

   ### 3.3 Position-wise Feed-Forward Networks

   In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

   $$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (2)$$

   While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer has dimensionality $d_{ff} = 2048$.

   Can be seen as passing through two linear layers with weights W1,W2 and biases b1,b2 respectively.

5. **Multi-Head Attention**
   Input sequence copied three times -> Q, K and V (all are same as, we are coding encoder)
6. **Skip connections or residual connections**
7. **In the figure, it is just figuratively showing the structure of one encoder block and one decoder block. However, actually it indicates that there are Nx encoder blocks with each previous block's outputs acting as inputs to the next block and the output of the final block is the "special embedding" we talked about.**

Important changes as of now to the code

1) **Features in some arguments in the code**
2) **Linear layer also added with the log of the softmax layer in the given code**

Tokenizer

Different kinds of tokenizers -> BPE tokenizer, word level tokenizer, subword level tokenizer, word par tokenizer

In this tutorial we are using a word level tokenizer -> separates words using space

Special tokens -> padding, SOS, EOS