Introduction to Large Language Models (LLMs)

Prof. Tanmoy Chakraborty, Prof. Soumen Chakraborti
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture 15

Introduction to Transformer: Self & Multi-Head Attention

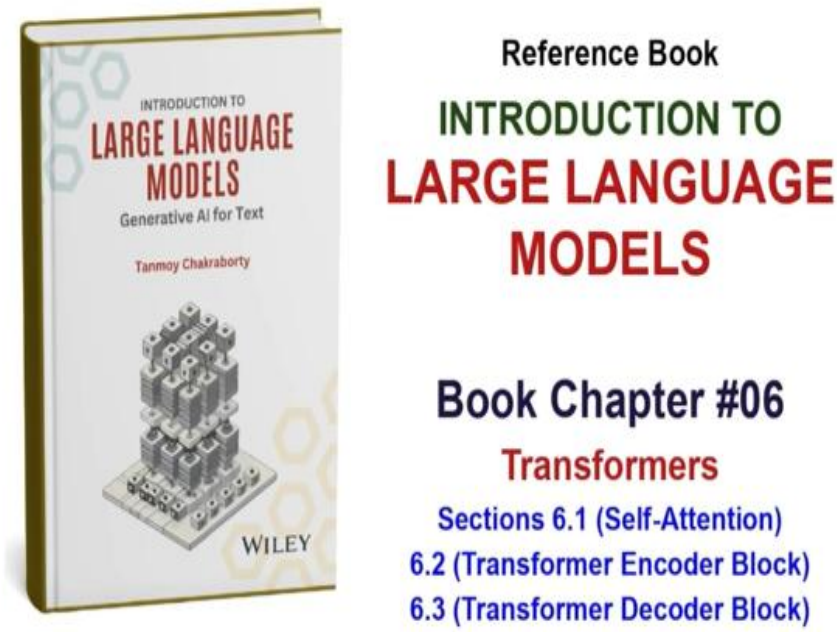
In the last class, we have been discussing attention. We completed RNN, different variations of RNN, LSTM, GRU and we also understood how back propagation through time happens. Specifically through LSTM and GRU how we can use gated mechanism to address the vanishing gradient problem and we also discussed in the last class about sequence to sequence model where I mean we discussed in the context of machine translation. where we have an encoder block. We have a decoder block. Encoder is responsible for scanning the source sentence.

Let's say the task is machine translation, Hindi to English. So an encoder is responsible. An encoder RNN is responsible for scanning the input source sentence. And the decoder RNN is responsible for translating the source sentence into the target sentence.

And we also discussed a problem called the bottleneck problem. I hope you remember. The bottleneck problem is that the last hidden state of the encoder block is responsible for storing the entire input. And so there's a bottleneck in terms of information storage. So to address this, we have introduced the concept of attention.

In the attention, the idea is that you have direct connection from the decoder to encoder, every decoder block has a connection to the encoder block, each of the encoder blocks, and we do some sort of dot product between the decoder state and the encoder state. And the dot product indicates a similarity between the encoder state and the decoder state. And then first we compute attention score. Attention scores are basically the similarity scores between a particular decoder and all the encoders. And then from attention score we pass this course through a shortmax which will give you a distribution, this is the attention distribution and then we consider the probability in the distribution as weights and then we

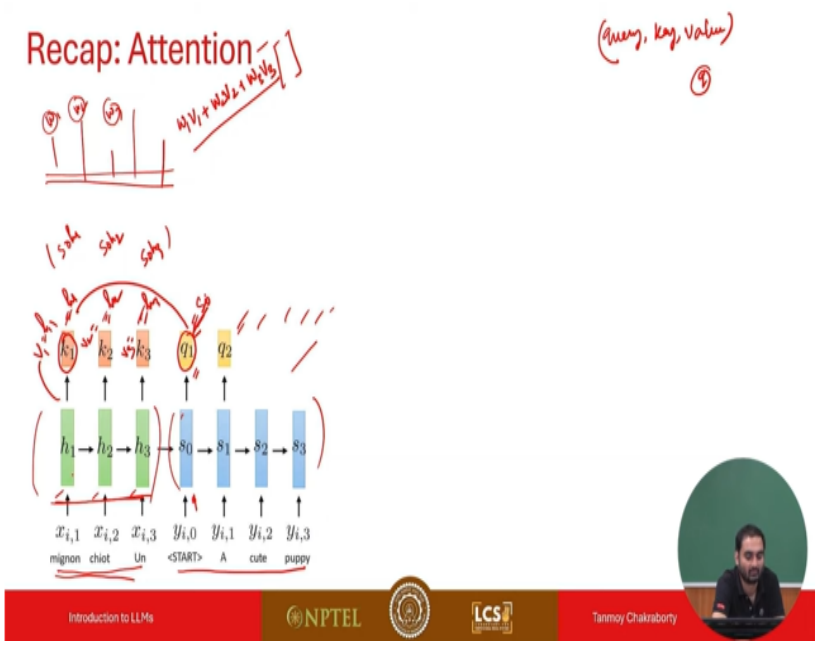
take the weighted average of the encoder states and that is going to give you the attention vector.



So in this class we will discuss the problems of the attention mechanism and we introduce a concept of self-attention and then we move gradually move towards discussing the concept of the transformer model.

So, the question that we are asking here is whether the attention that we have discussed so far is all we need or not. Maybe the attention mechanism that we discussed over is not enough, we need something else to address the problems that are already there. So in normal RNN what were the problems? The vanilla RNN the problems were number one the bottleneck problem, number two is vanishing gradient problem, number three is the I mean it is not you are not able to access the entire all the tokens in parallel right you have to do this thing in a sequential manner isn't it. Attention partly addresses the vanishing gradient problem, attention also addresses the bottleneck problem but attention till its current form that we discussed so far is not able to address the problem of the sequential access versus parallel access.

So, till now we access all these tokens, let us say we have token w1, w2, dot dot wn, we access all these tokens in sequence, meaning that at time t when you compute the hidden state, you have to wait till time t minus 1.



So in the last class again very quick recap of the last class attention right. By the way attention is not restricted to only the sequence to sequence modeling problem. I discussed last day that attention is a very generic concept right. You can use attention to a single RNN not necessarily a sequence to sequence problem where there are two RNNs right.

Now let us focus on the sequence to sequence model problem again. You are given an input sentence this and your task is to translate it to a target sentence. And these are the hidden states, h1, h2, h3 are the hidden states in the encoder. Now this is my encoder and this is my decoder, s0, s1, s2, s3 these are the decoder hidden states. What we do? In the normal attention what we do? We consider each of these decoder hidden states as a query and then what you do then we take let's say we are focusing on this decoder state, we basically take a dot product between q1, q1, k1, q1, k2, q1, k3, what are this k1, k2, k3 by the way.

So, in normal attention right last day I mentioned that these are called queries right and the encoder states are called values. You can also think of these encoder states as keys. We will introduce a concept of query. So far we talked about query value. We will introduce a concept of query key and value today, very same as the information retrieval concept, right? Where, let's say when we type a query, I mean, you basically search that query, right? With all the possible keys present in the documents. Now, what is the key in a document? Let's

say you want to retrieve a document. It is used for retrieve the document. What is the key? The key can be, let's say, the metadata, or the key can be the keywords, for example, which describe that particular document, right? So in a very vanilla setting, normal setting, what we do? We take the query, which is the user's query, and we look at the similarity between that query with each of the keys, each of the keywords corresponding to the documents, right? And we do some sort of similarity matching, let's say cosine similarity based matching, right? And then we identify a particular document that has the highest similarity, right? Then what we do, we retrieve the document. So we may not want to retrieve the entire document, right? We may want to retrieve a part of the document which is relevant to the user's query.

Let's see in a document there are multiple concepts but the user is interested in a specific concept, specific paragraph. So you may not want to retrieve the entire document, you may retrieve only that paragraph which is relevant. So here the entire document acts as a value. So you may not want to return the entire document, you want to return a part of the document. Now user's query acts as a query and the metadata information of every document acts as a key.

You measure a similarity between the query and the key. You measure the similarity between a query and all the keys corresponding to all the documents. And then based on the similarity, you return the corresponding value. The value is the document, the entire document. You may not want to return the entire document.

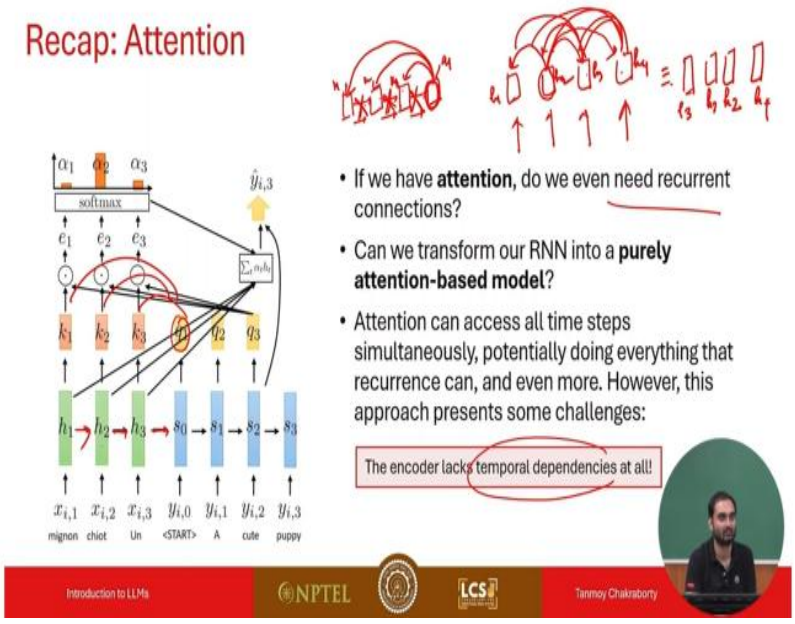
You want to return a specific part of the document which is relevant to the query. And how do you know which part is relevant? We already got the dot product. We know that this is the similarity and we do maybe another round of similarity between that query key dot product with all the values present here. Values are essentially all the paragraphs in a document and you want to return a part of the document or the entire document. The idea here is the same, exactly same.

In a vanilla attention, the one that we discussed so far, we have a query Q1 corresponding to this decoder state. How do we get this Q1? In a vanilla attention setup, this query is nothing but the hidden state itself. Q1 is basically H0. And what is K1? K1 is H1, K2 is

H2, K3 is H3 and so on and so forth. Then you take a dot product between S0, so S0, H1, S0, H2, S0, H3.

These are the attention scores and then we pass these attention scores through attention distribution. It will give you a distribution. So these are essentially weights, let us say W1, W2, W3 and so on and so forth. Right and then once you got the distribution, once you got the similarity score Now these similarity scores will act as weights, then what we do? Then, essentially you retrieve the corresponding value right. What is the value here? Let's say, along with k1, along with this key, you also have access to a value v1 this value is also generated from the hidden state in the attention setup this value is also H1, keys and values are same in attention.

So, values are basically so V1, V2, V3 are essentially H1, H2, H3 respectively. So after this weights then what we do? we take the weighted average of all the values meaning w1 v1 plus w2 v2 plus w3 v3. This will produce a vector and this vector is called the attention vector.



So, this is the vanilla attention mechanism and then again you take another decoder state, You take the dot product of the decoder states with all the encoder states, values, attention

scores, distribution and so on and so forth. So, this is RNN on top of RNN you have attention.

Now, here is the question that we are asking, the question is if a decoder state by the way as I mentioned this is not restricted to encoder decoder setting you can also think of a simple RNN like this single RNN not necessarily encoder decoder a single RNN and every RNN state can access to all the previous RNN states. So query can come here and these are basically keys k1, k2, k3 and you do the same operation. So, the question here we asking is as follows. If through attention mechanism, if a decoder state can access to all the encoder states, through attention mechanism we see that every decoder state can access to all the encoder states. Why do we need these recurrent dependencies? Why do we need these dependencies between H1 to H2? So if this guy can access to all the previous guys, why do we need these connections? These connections are the bottlenecks.

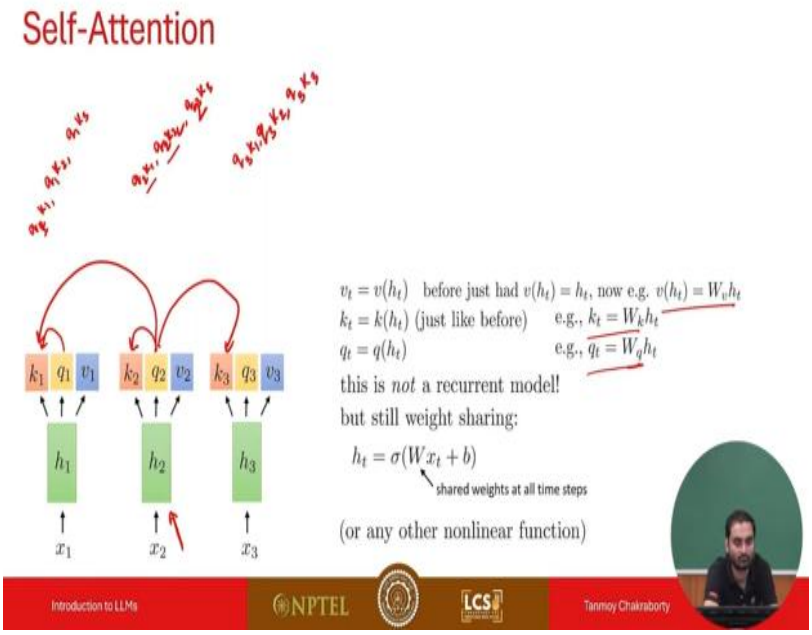
Why they are bottlenecks? Because of these recurrent connections, you are not able to access all these tokens in parallel. If these had not been there, you could have accessed all the tokens in parallel. Isn't it? So the question that we're asking is, do we really need the recurrent connections if we have attention? Okay so what we do in the self-attention mechanism is that we remove these dependencies. Okay if we remove these dependencies now tell me now it looks like this now the hidden states looks like this right and these are inputs. Now each of the states let us say this state can access to can basically access to all the other states.

This guy can access to this, this and this, this can have access to this and this. Now each of these operations are independent of each other, right. When you compute the operation for this guy, you can do the same thing for other guys also, okay. So this is the premise behind the self-attention mechanism but here is a problem. what is the problem? The problem is that when you discard this recurrent links, you basically lose out the information of the sequence, right.

You do not, so since this, now think about this, this is H1, H2, H3 and H4. H1, H2, at H1, at H2, H3, H4, you do all these attentions independently. So, this is equivalent to another network where let us say this is H3, this is H1, this is H2 and this is H3, H1, H2, H4. If you

shuffle the hidden states, you get the same output because there is no dependency between the hidden states. So it basically means that the temporal dependencies which are preserved in recurrent neural network that will not be preserved if we do self-attention, if we remove the recurrent connections.

How do we address them? We will discuss later.



But before that let us focus on the self-attention part. So in self-attention what we do? Let's say you are given inputs x1, x2, x3, x4, these are essentially tokens denoted by xt. We pass it through some sort of linear layer followed by non-linearity. This is a linear weight w, learnable weight and non-linearity and we obtain hidden states h1, h2, h3.

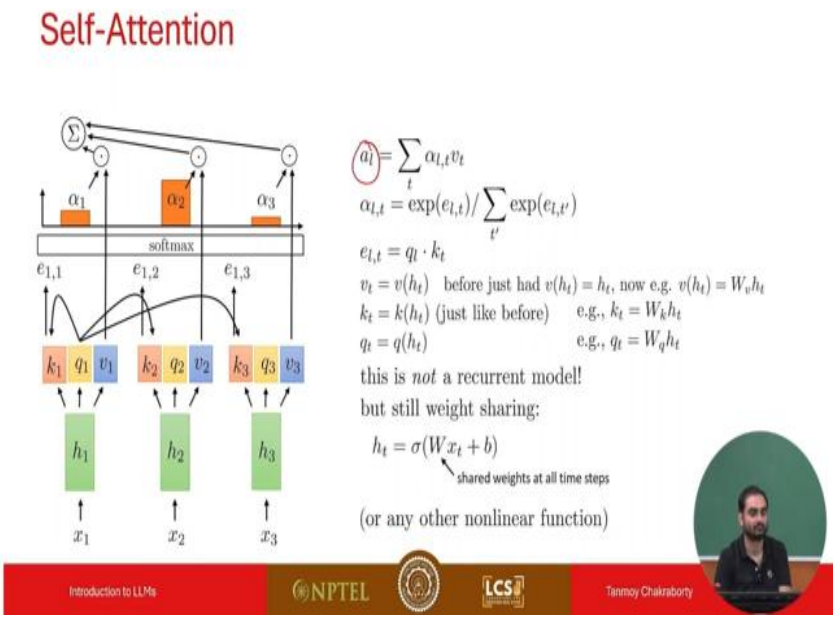
Okay now from each of the hidden states I will now generate three vectors, from each of the hidden states I will generate three vectors, one is called the query, other is called the key and the remaining is called the value. Okay and how do we do that? Very simple idea. How do we do that? We do this using three parameter matrices. For query generation, we use the parameter matrix WQ.

For key, we use WK. For value, we use WV. These are the learnable parameter matrices. It basically means that we multiply h1 with WQ, which will produce the query Q1. We

multiply H1 with Wk which will produce K1, we multiply this with Wv which will produce V1. Similarly, you multiply H2 with Wq which will produce Q2, Wk which will produce K2 and Wv which will produce V2.

And similarly, Q3, K3, and V3. The same set of parameter matrices are used to generate queries, keys, and values for all the tokens. The parameter matrices are shared. So value you obtain through WV, key you obtain from using WK, and query you obtain from WQ. These are the linear transformation. Then what we do? Same process, same as attention.

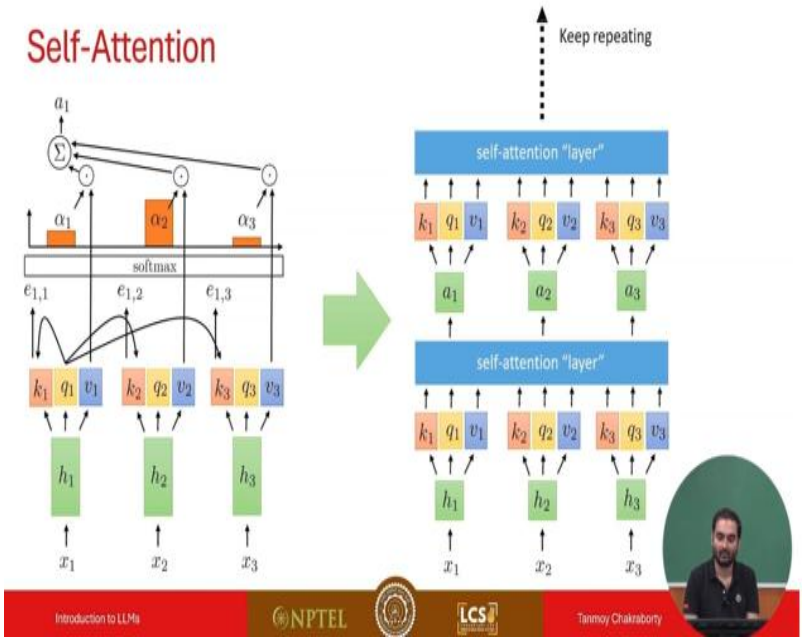
Let us say you are focusing on this one. So, this query and the key will basically be combined through dot product. So, we will get Q2, K1, then you do a dot query between Q2 and K2. So Q2, K1, Q2, K2, and then Q2, K3. So every query is multiplied with all the keys. So at this position, you have Q2, K1, Q2, K2, Q2,k3 At this position you have Q1, K1, Q1, K2, Q1, K3 and at this position you have Q3, K1, Q3, K2 and Q3, K3.



So, at this position let us say at this position Q1, K1, Q1, K2 and Q1, K3 at this position you have Q1, Q1. Now these are attention scores. We denote this scores using E, what is E, L, T? The index L corresponds to query and T corresponds to key. So, we got attention scores, then we pass this attention scores through a shortmax, then we will get the attention

distribution. So, this attention distribution, this probabilities alpha 1, alpha 2, alpha 3, they will act as weights and then these weights will be multiplied with the values.

So, essentially alpha 1 V1 plus alpha 2 V2 plus alpha 3 V3. So, this is my attention vector. Now so we can generate this attention vectors independently irrespective of the positions.



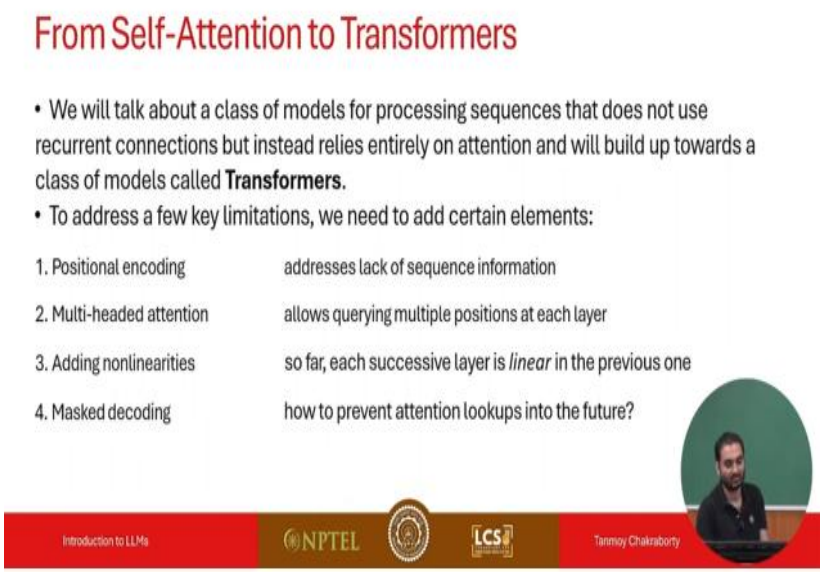
Now you can think of the entire operation that I discussed so far, this query key value generation then dot product and distribution etc.

You can think of this as a layer. So once we obtain the hidden states, you can think of the remaining part that we discussed over as a layer and within the layer all these things are happening. So we call this layer as self-attention layer. So we can repeat the process, the self-attention process multiple times. So we have the input. We basically place a self-attention block, right? The self-attention block will produce what? The self-attention block will produce attention vectors, right? And then on top of this, you can again add another self-attention block.

You can keep on doing this thing multiple times, right? Remember, for every self-attention layer, right? The query key value pair, the query key value matrices are unique. Meaning at the first layer, you have let us say WQ1, WQ, WK1, WV1. At layer 2, let us say at layer

2, you have WQ2, WK2, WV2. The matrices are not shared across layers. Matrices are shared across tokens, but not across layers.

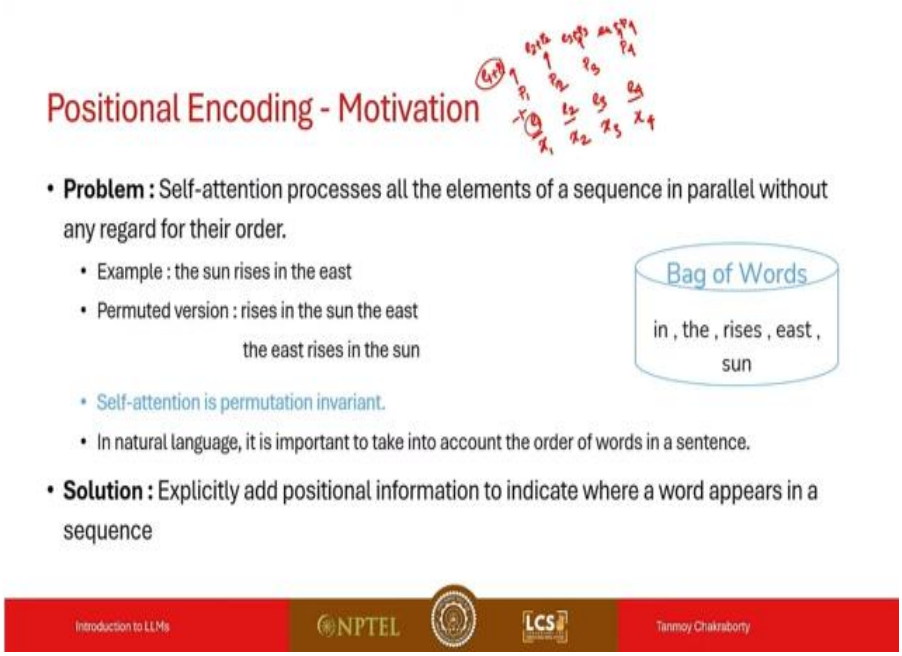
So as you have access to bigger and bigger, larger and larger data set, you keep on stacking the self-attention layers one by one. And you keep repeating this thing.



So this is one step towards the transformer model. So we understood what is self-attention. Through self-attention, you can access the tokens in parallel.

So in that case, you can use the modern CPU machines to process things in parallel. So the one method problem that all these RNN methods, LSTM, GRU, have problems, that is now addressed. Now the second problem that I mentioned earlier is that so for addressing the non-sequential access what we missed out is the position information. This is position invariant. So we need to basically come up with some sort of information which will give you the idea of the position of every token.

Let's say the man eats a cheese versus a cheese eats a man. Just the position of the word man and cheese actually matters. If you swap them, then the meaning will change. But in a self-attention mechanism, these things can't be distinguished.



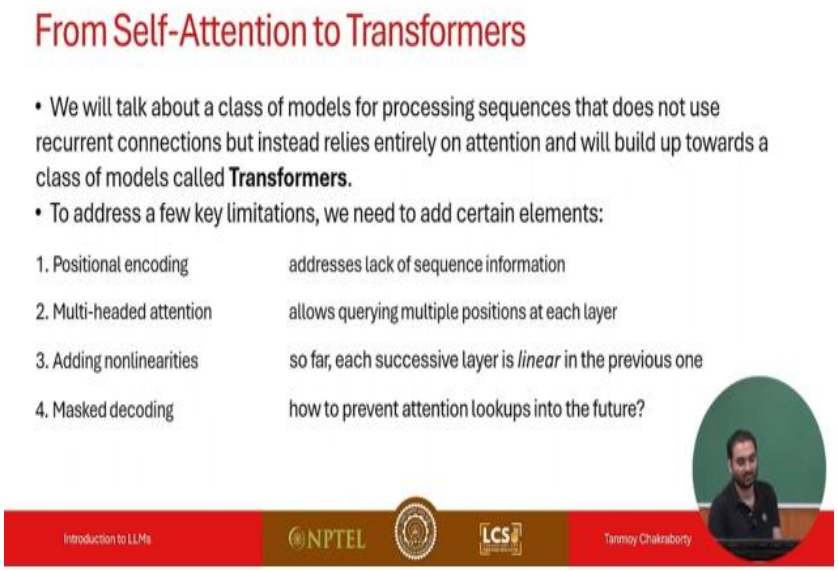
So now we'll discuss positional encoding. So positional encoding is a mechanism through which you can encode positions of tokens. I will not go into details of positional encoding in this lecture. I will take another separate lecture on positional encoding where we will talk about the encoding mechanism that was proposed in the transformer paper as well as some advanced positional encoding methods. In a high level what it does, the position encoding method does, it essentially gives a position information to every token. Let us say x1, x2, x3, x4 these are the inputs and let us say the corresponding embeddings are e1, e2, e3, e4.

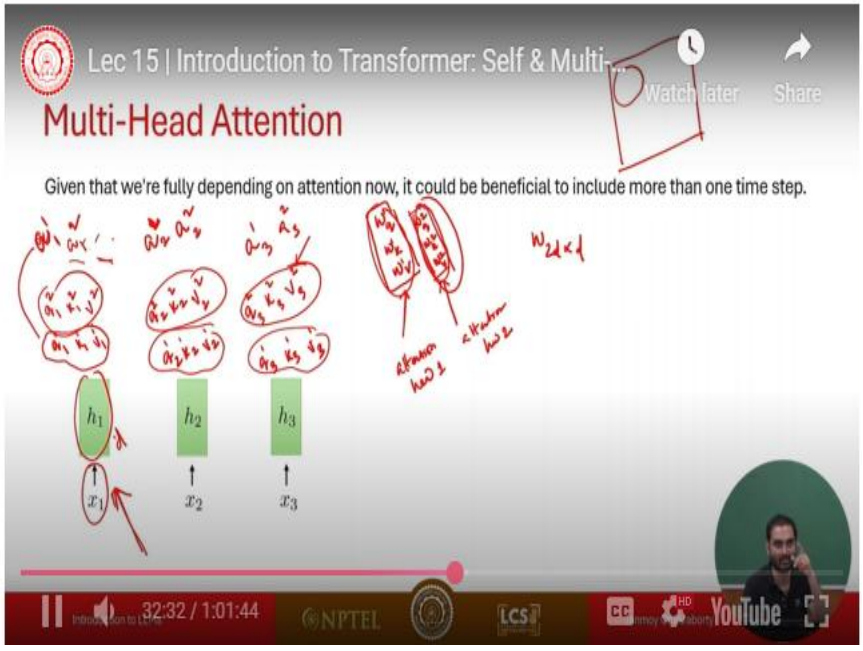
What are these embeddings? These embeddings can come from one hot encoding or this can be obtained from word to vehicle glove. what you do on top of this you derive the positional information of this tokens P1, P2, P3, P4. Now this is a vector U1, E2, E3, E4 these are vectors P1, P2, P3, P4 these are also vectors. So position is encoded through a vector and then you add them. Okay so here the resultant vector is E1 plus P1 right add means element addition is element to the addition right.

So E2 plus P2, E3 plus P3, E4 plus P4 this is now fed to the self-attention block. So instead of feeding E1 you now feed E1 plus P1. Now how do you get this P1 I will discuss later. For the time being, you assume that this P1, P2, P3, P4 are obtained using some process

and they are unique. They have certain properties, the distance properties, the sequence properties, these are all preserved.

So, we will not discuss this one right now.





Alright, let us move on to the next concept multi-header self-attention. Now let's see what additional thing we can do on top of the self-attention. Why we want to do so many things

because now we have access to huge amount of data. We can add as many parameters as we want.

So let's see the concept of multi-headed attention. So in multi-headed attention or multi-headed attention, in a single attention what we do, use query key and value matrices, and we obtain Q1, K1, V1, Q2, K2, V2 and Q3, K3, V3 using these three matrices. In multi-headed self-attention what we do, we repeat the same thing multiple times. meaning we have one set of parameter matrices we will take another set of parameter matrices right which would look like this WQ let us say this is 1 1 1, WQ 2, WK 2, WV 2 is another set of parameter matrices. So from first set of parameter matrices we obtained Q11, V11, this one, this one, so the superscript indicates different attention head.

Now using this we obtain another set of vectors Q12, K12, V12. q 2 2 k 2 2 v 2 2 and q 2 3 k 2 3 v 2 3, this is clear. So, the superscript indicates, so each of this parameter matrices, the set of this parameter matrices is called attention head. So this is attention head 1, this is attention head 2. So through this attention head, through one of each of these attention heads you generate an attention vector.

So through one attention head you generate one attention vector. this will be processed through another attention head. So, through attention head one you will essentially process all these things through the way that we discussed earlier and it will produce an attention vector which is let us say A11. You process all of these things and this will produce attention vector 2 So, a11, a12, this will be a21, a22, a31, a32. There are two attention heads.

So, every attention head will produce an attention vector. Now, for every token, we have two vectors in this example. Because we have two attention heads. You can have multiple such attention heads. Let's say in the transformer paper they tested their model with two setups.

Eight attention heads is the standard one. Now why do we need these attention heads? This is the question. Why do we need multiple such parameters? Now the idea essentially is influenced by the convolutional neural network architecture. So in CNN, if you are ever of CNN, you may know that when you focus on a specific patch of an image, let us say this

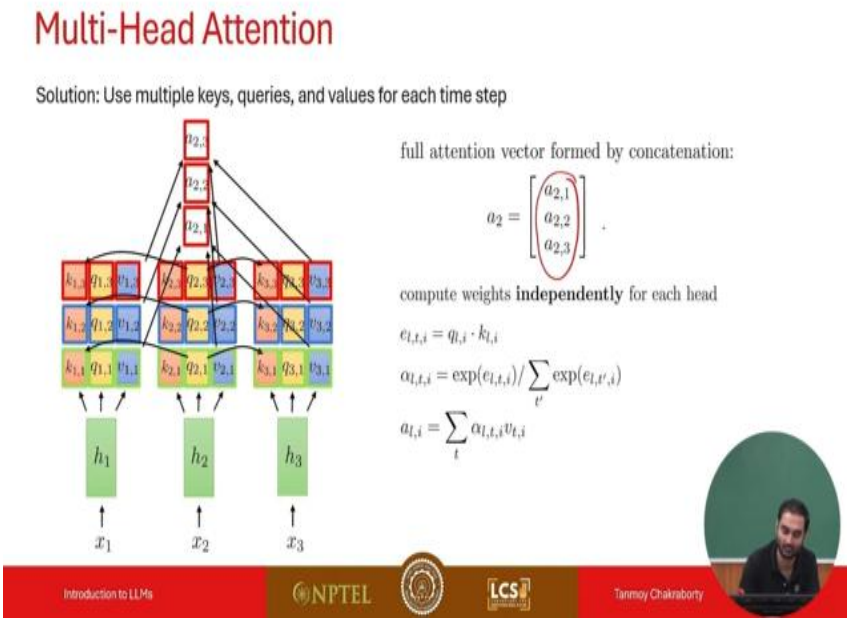
is the RGB vector and you focus on one part of the image, you use a kernel and you do max pooling, min pooling, whatever. In fact, we use multiple kernels on a single patch. And the hope is that each of the kernels will correspond to one particular aspect of that patch.

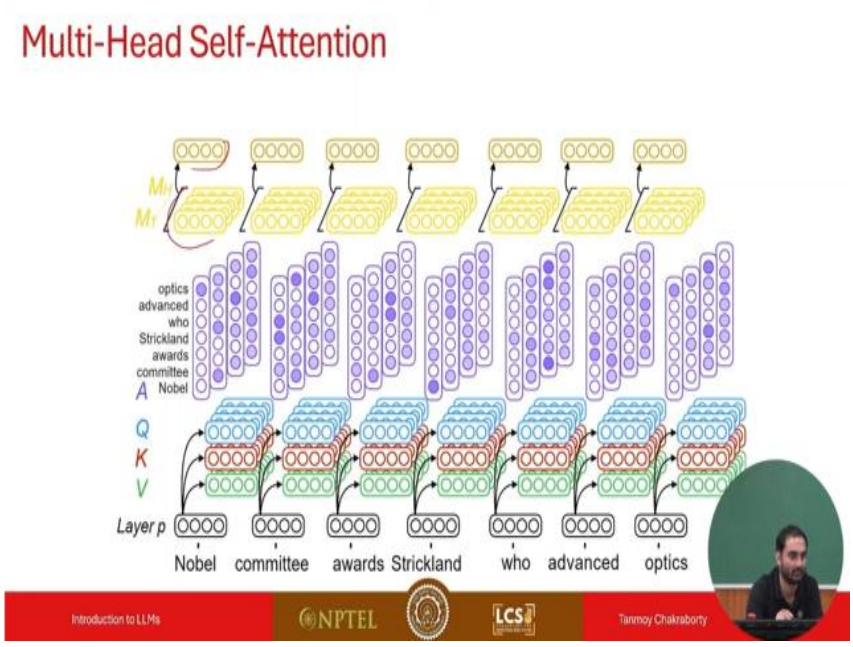
Let's say there's an image of a dog. And you want to classify this whether it's a dog or a cat. You want to look at the ear, the eyes, nose, and tail, whatever. These are different aspects. So when you focus on a particular part of the image, one kernel will be responsible for fetching information of the eye.

Another kernel will focus on fetching information of the ear for example. Another kernel will focus on fetching information of a tail or nail whatever. So, you are essentially fetching different aspects of a specific region through different kernels, through different filters. Here also you use this attention heads to fetch different information from a specific token. Now what are these information? Let's say you are interested in whether you are interested in understanding whether this specific token is used as a subject or not.

So one attention head will look at that aspect. You are also interested in knowing whether this is a pronoun or noun or not. POS tagging. So another attention head will look at that aspect. You want to understand whether this is a named entity or not.

Another attention head will look at these aspects. Now you can ask who will tell you that these attention heads will look at these aspects?





This is just for the illustration purpose. We do not know which attention is corresponding to which aspect, but as we increase the number of attention heads, we basically assume that different aspects of that token will be captured through different heads. This is called multi-head attention, multi-headed attention.

So these are heads, okay. Now for every token, we have multiple attention vectors. What is the size of this vector? The size of this vector is same as the input vector size. Let's say

the dimension is d, each of these attention vectors will have dimension of d, right. So if there are two attention heads, what's the resultant vector size d and d, right. Then what to do with these two vectors because remember the next layer has this self-attention block whose input is a vector of size d.

But now you obtain two vectors of size d. What to do? You concatenate them. If you concatenate them, the size will be 2d. How do you get that d vector from a 2d vector? You pass it through a linear layer whose size is, linear layer is basically a matrix W of size 2d cross d. This will project a 2D vector to a D vector, this is called output matrix we will discuss later, okay. So, we need another projection which will project a 2D or if there are 8 vectors, 8 attention 8 D vector to a D vector, okay.

Multi headed addition is clear. This is just for the purpose of illustration. So, in this example there are three heads you see there are three attention vectors right and three attention vectors are concatenated. Let us look at this very interesting illustration see here. This is the sequence right, these are the tokens. For every token I generate a query which is the blue vector key red vector and value the green vector right and then this is the attention vector right when we use a single attention head right and so this, so, sorry, these are essentially not attention vectors these are essentially the similarity values right meaning the attention scores.

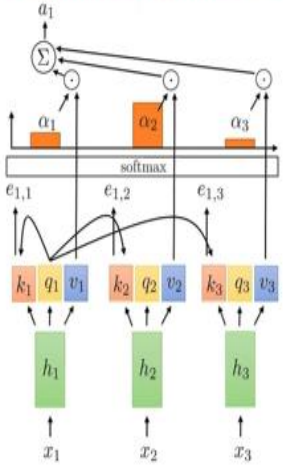
From attention scores you obtain the attention vector for every token. Now in multi headed you have multiple sub vectors, multiple sub vectors, multiple sub attention distributions and then what you do? You concatenate them and pass it through some sort of linear layer and this will result in a vector which has the same dimension of the input vector.

From Self-Attention to Transformers

- We will talk about a class of models for processing sequences that does not use recurrent connections but instead relies entirely on attention and will build up towards a class of models called transformers.
- To address a few key limitations, we need to add certain elements:

1. Positional encoding	addresses lack of sequence information
2. Multi-headed attention	allows querying multiple positions at each layer
3. Adding nonlinearities	so far, each successive layer is *linear* in the previous one
4. Masked decoding	how to prevent attention lookups into the future?

Self-Attention Is "Linear"



$$k_t = W_k h_t \quad q_t = W_q h_t \quad v_t = W_v h_t$$

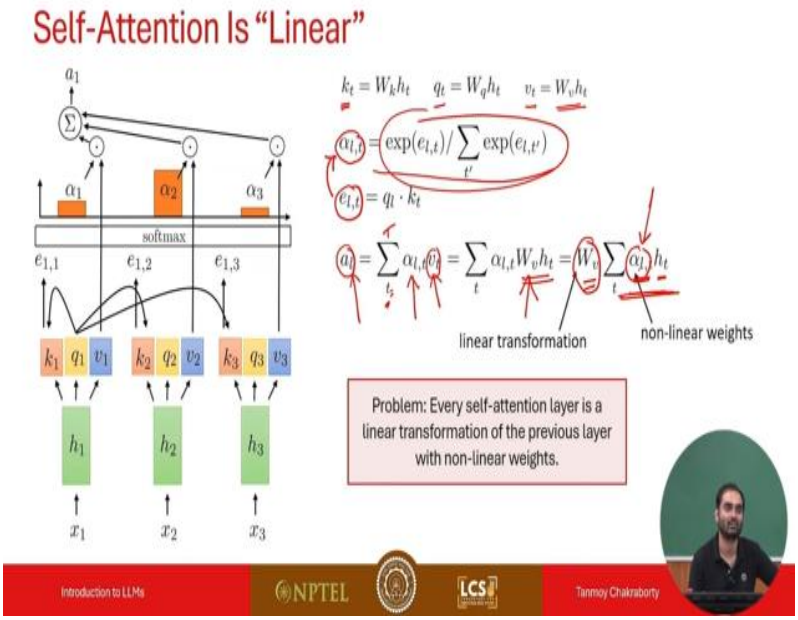
$$\alpha_{l,t} = \exp(e_{l,t}) / \sum_{t'} \exp(e_{l,t'})$$

$$e_{l,t} = q_l \cdot k_t$$

$$a_l = \sum_t \alpha_{l,t} v_t = \sum_t \alpha_{l,t} W_v h_t = W_v \sum_t \alpha_{l,t} h_t$$

linear transformation — non-linear weights

Problem: Every self-attention layer is a linear transformation of the previous layer with non-linear weights.



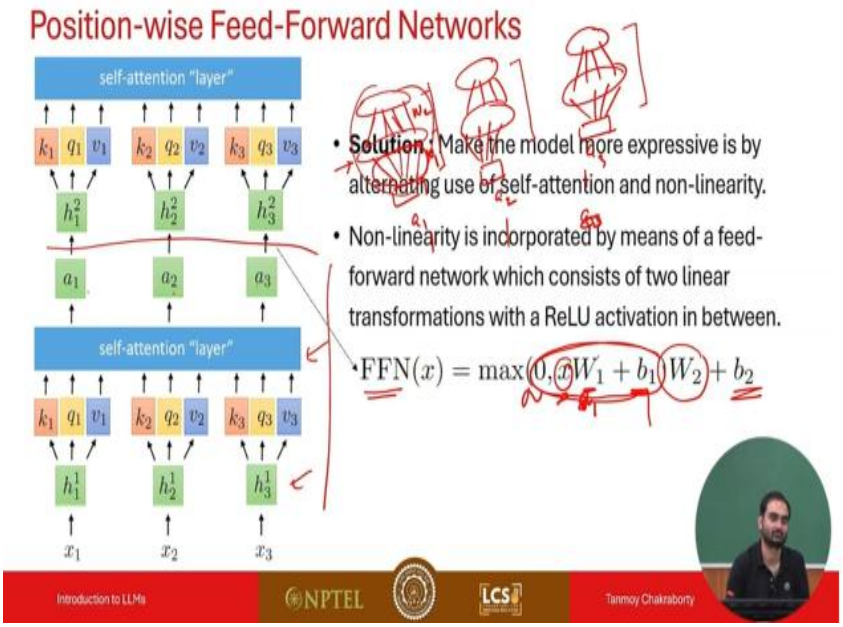
Let us move on to the next concept. So we discussed three things, one is self-attention, position encoding we briefly discussed, we understood at least what it does and then multi head self-attention. So far what we have discussed, all these operations are linear. By the way what is the complexity of the self-attention block? If there are n number of tokens, so, every token is attending to all the other tokens including itself.

So, complexity is order of n square. So, and we will see that this attention, self attention is a linear operation. Although it looks very complicated, but ultimately it is a linear operation.

We will show this thing. Let us see. And we will show this thing using one attention head. one attention head w1 q1 wq wq wk wv. So, let us say this is my key query qt kt and vt and what is alpha lt alpha lt is the. is the score after, So ELt is the similarity dot product and alpha Lt is computed after this which is the short max output and the attention vector is the probability, the weight and the corresponding value, alpha L t times V t and this ranges from t, what is t? t indicates the keys. and l indicates the query or values whatever right all these positions. So, t ranges from 1 to t whatever t is a if you replace vt by wv ht you will get this one right wv is independent of t right you can keep this thing outside.

Now it is essentially alpha lt which is the attention score right and ht, ht is the hidden state, the input. So essentially what is happening is that you are multiplying the hidden state width and weight, you are multiplying the hidden state width and weight and this weight is computed through some sort of non-linear operation. So the weight is a non-linear operation right but this multiplication is a linear operation okay this weight is coming through a non-linear process but ultimately this is a linear operation. This is linear in the hidden states right, this multiplication is again a linear projection.

So self-attention is linear. If self-attention is linear, we all know that linear models have many problems. We always want to add more non-linearity at different parts of the network. So far we have not added any non-linearity. How do we add non-linearity?



We add non-linearity to another layer which we call feed forward layer. Okay what is a feed forward network? we all know what's a feed forward network is right you have multiple layers right it's a multi-layer perceptron type right multiple layers you feed some input and those inputs are combined and then some sort of non-linear operation happens right and then you pass it through to another layer some again another round of non-linear operation and so on.

This non-linear operations can be sigmoid, can be short max, can be ReLU, right, GELU whatever activation function you want to take. So let's see how it works. So let's look at this part till this part. So we have this input we applied a self attention layer single head self attention layer and you obtain this three attention values attention vectors.

A1, A2, A3. What we do, so this is position 1, so this is A1, this is A2 and this is A3. Now remember A1, A2, A3 are vectors. What we do, we apply a feed forward network at every position, meaning we will apply a feed forward network like this. There is a two layer kind of, I mean one hidden layer, something like this. So, this is a feed forward network there are two layers. So, what it will do it will essentially do this thing. So, this is the attention vector that we obtained right x whatever x Okay you first pass it through one layer right meaning you multiply this with a matrix which is W1 and the bias. Right and you do some sort of non-linear operation here right and then you pass this, pass the resultant one through another matrix W2, this is W2, this is W1 and the bias and this feed forward network, this non-linear operation can be sigmoid in the paper they use ReLU but you can use sigmoid or other operations also. So how many parameters are introduced here? If you discard bias for the time being, two parameter matrices, W1 and W2.

And these parameter matrices are shared across all the tokens again. And this is called position wise feed forward network. Position, position wise feed forward network. Why? Because at every position, at every position I have a feed forward network.

Whereas attention, self-attention was not position wise. Self-attention was applied across all the tokens. Now this is position-wise feedforward network. What is the purpose of this position-wise feedforward network? Think about it carefully. So self-attention essentially allows you to exchange information from different tokens. So from one token you get some information and you add that information to another token and so on and so forth.

So all this message passing basically happens in this self-attention block. You can think of this as some sort of memory fetching. What is memory fetching? Let's assume that all these attention vectors are stored in a memory. Okay, so when we fetch, let's say when we consider one attention vector as a key, as a query and other attention vector as a key and so on and so forth and then you do all the dot product, you are essentially fetching one

vector from the memory, another vector from another part of the memory and you are doing some sort of operations, some sort of mixing is happening, right and you produce a result, you basically produce another set of, another configuration of the memory. The actual processing happens in feed forward network.

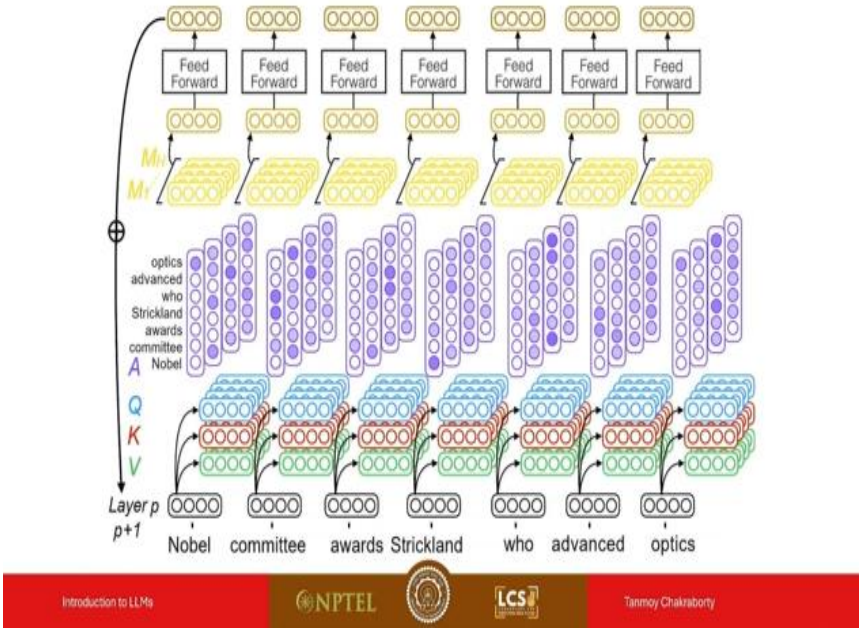
You fetch information from memory and you do some processing. Again you fetch information from memory to do processing. Because this is non-linear, so this is non-linear right, feed forward network is non-linear because you are applying multiple non-linear functions, activations. So you can think of self-attention as fetching information from memory or message passing and feed forward network as actual processing unit. Feed forward network is also considered as internal memory of the transformer model.

Because you see the amount of parameters that are being used here, W1 and W2 these are two huge matrices. Now generally the dimension of this one the dimension of every embedding is 512 generally but you can use you can increase. Now another thing I forgot to mention is that if you look at the shape of the feed forward network. It first projects the vector to a higher dimension meaning a D vector is projected to a 2D vector or 3D vector, this layer, the first hidden state and that 3D vector will be again projected to a D vector. Now the size of the dimension of the hidden state is a hyper parameter.

If you want to increase the memory, you increase the size of the hidden state, right. The parameter matrix will also be increased there are two parameters w1 and w2 the size of these two matrices also increase okay. Now why do we need this up projection down projection why not down position up projection? If we do a down projection we will discuss later when we discuss the inner principle of transformer when we do down sampling then up sampling down projection up projection you lose many information it is always better to project it to a high dimensional space and then you down sample it okay. So what we have learned so far input, self-attention and feed-forward. So, self-attention block, feed-forward block, these two blocks constitute one block of the transformer model.

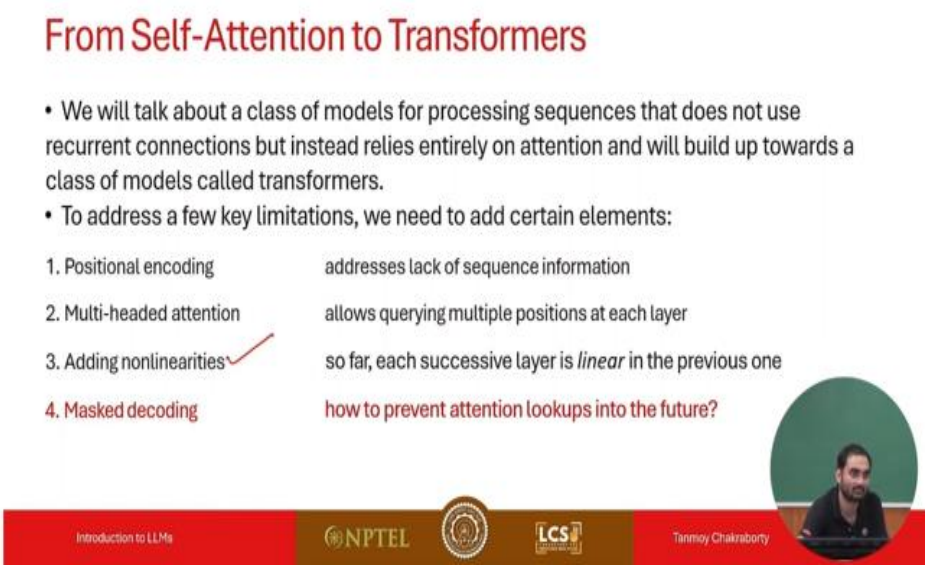
And you repeat this block multiple times. So, every block has a self-attention and feed-forward network. In a vanilla transformer model, they repeated it 6 times and 12 times. Transformer base model and transformer large model.

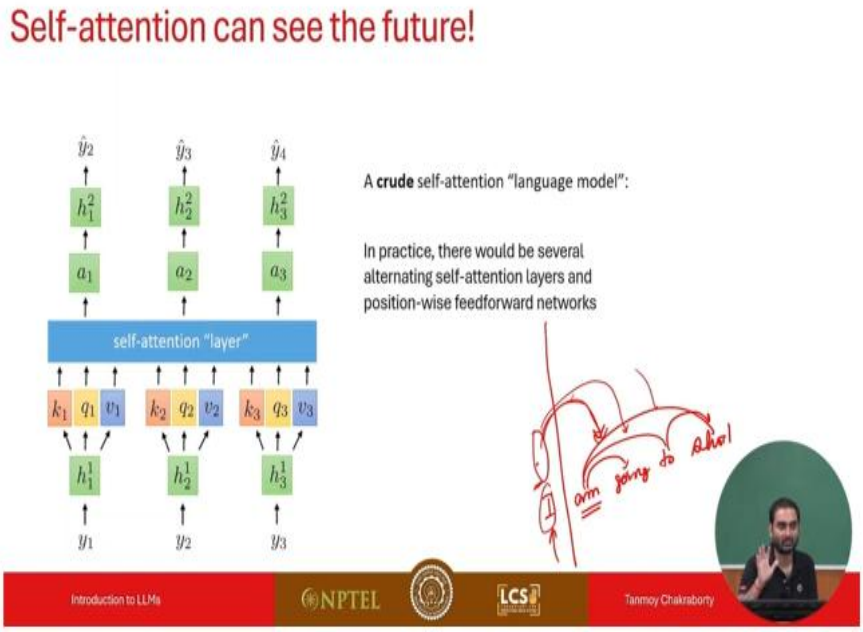
Transformer large, there are 12 such layers. and transformer base there are six such layers. Let us move on.

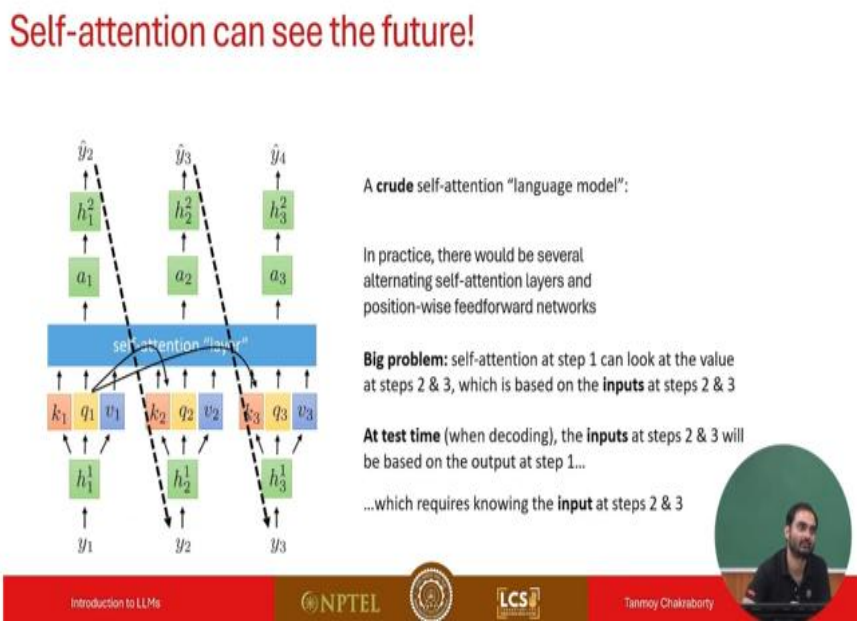


Now this is the resultant picture now. So far we discussed till this part. Now we have a feed forward network which will result in another vector.

So we have added non-linearity.







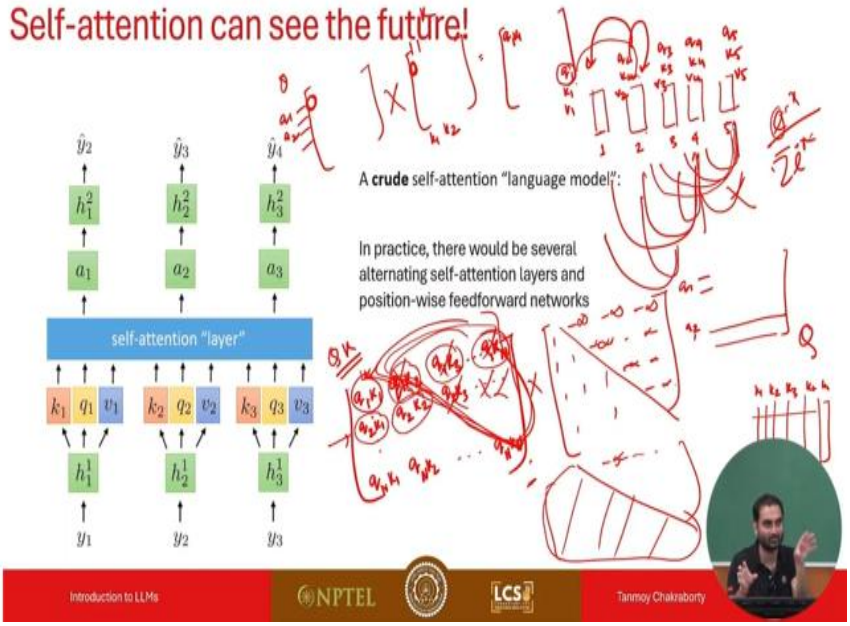
Now let us look at masked attention. We have missed out a very subtle details in self-attention. If you remember self-attention, every query is attending to all the keys present in the sequence. It means that if the sequence is let us say I am going to school, this guy is also attending to going to school, who is also attending to school and so on and so forth. If you think of decoder, remember this encoder-decoder model that I discussed earlier, right? You have an encoder, you have a decoder. What is the responsibility of decoder? Decoder

responsibility to generate or to translate this sentence into the target sentence, right? Let's say, mai school jaa rahu, Hindi, right? And you want to translate it to English.

So, when we generate I, right? When you generate I, the corresponding decoder hidden state doesn't know what is going to come at the next state, at the next timestamp because you generated till this part. So this guy should not know what is or so this guy will not know what is going to come at the second position, what is going to come in the third position, fourth position and so on and so forth. So a hidden state of a decoder will not have access to the future hidden states. Whereas in the encoder, encoder is given, the sequence is given, the input in this sentence is given.

So each word has access to all the other words. But decoder, the English sentence is not given to us during the inference time. So I generate one token, then based on that I will generate the next token and so on and so forth. So the normal self-attention mechanism will not be applied in decoder, okay.

What we do here, we apply something called a mask self-attention.



What is a mask self-attention? Let me draw this, okay. Let's say now here I will discuss this thing in two ways, one during the training time what will happen and two during the

test time what will happen, okay. Let us say this is the decoder okay, 1, 2, 3, 4, 5, now remember during the training time you are given both the input and output centers, source and target centers, you know the entire target centers right, you know the entire target centers during the training time but during the test time you do not know right. So in the training time if you access the future decoder states that will be a cheating for the test time because at the test time you won't have access to the future states. So you have to do some tricks in the self-attention block such that in the training time also the decoder block, the decoder hidden state will not get access to the future decoder state.

So what we will do, look at here. Let's say this is Q1, K1, V1, Q2, K2, V2, Q3, K3, V3, Q4, K4, V4, Q5, K5, V5. These are query key and value vectors obtained from every decoder state. We need to make sure that let us say this hidden state will have access to all the previous hidden states, this is obvious, but the fourth hidden state will not have access to this one, will only have access to third, second and first. The third hidden state will have access to only second and first and of course the state itself, third. The second hidden state will have access to state 2 and state 1 and so on and so forth. How do we do this? In practice now remember in practice all these vectors will not do vector operation will basically do matrix operation right meaning I have a matrix called a query matrix Q right where all the queries are present let us say Q1, Q2, Q3, Q4 right these are queries.

If you do not do matrix operation then you would not be able to leverage the GPUs and TPUs for example, tensor processing units. And then let us say the key vector, the key matrix contains all the keys, let us say these are keys and then let us say keys are stored column wise, k1, k2, k3, k4, k5. So this is your Q and this is K, Q1, Q2, K1, K2 when you multiply them and this multiplication is essentially element wise multiplication, this is not a matrix multiplication when you multiply like this, it is an element wise multiplication. So this part will be multiplied with this guy right. So, the resultant matrix will be q 1 k 1 let me write in this way here q 1 k 1 q 1 k 2 q 1 k 3 these are the all dot products right all dot products dot dot dot q 1 k n where n is the size of the decoder.

Similarly, dot dot dot here qn k1 qn k2 dot dot dot qn kn. What is this one? This one is q2 k1 q2 k2 q2 k3 and so on and so forth. Look at here. So is it allowed for the Q1 to get

multiplied with K2? No. Q1 is not allowed to be multiplied with K2 because K2 is the future state with respect to Q1.

So this is not allowed. This is also not allowed. This is also not allowed. Only this is allowed. Look at this part, this is allowed Q2, K1, yes Q2 can attend to K1, Q2 can also attend to K2, so this is allowed, but the rest are not allowed, this is not allowed, this is allowed. So, if you think of it carefully, essentially you are dissecting the matrix diagonal wise and the upper part of the diagonal elements are not allowed and lower part of the diagonal is allowed right. How do we do this thing computationally, I mean coding wise? So in the training during the training time although you can generate the entire matrix but you are not allowed to process the entire matrix you are allowed to process only the lower diagonal part of it not the upper diagonal part of it. What we do? This QK matrix will be multiplied with another matrix which would look like this one.

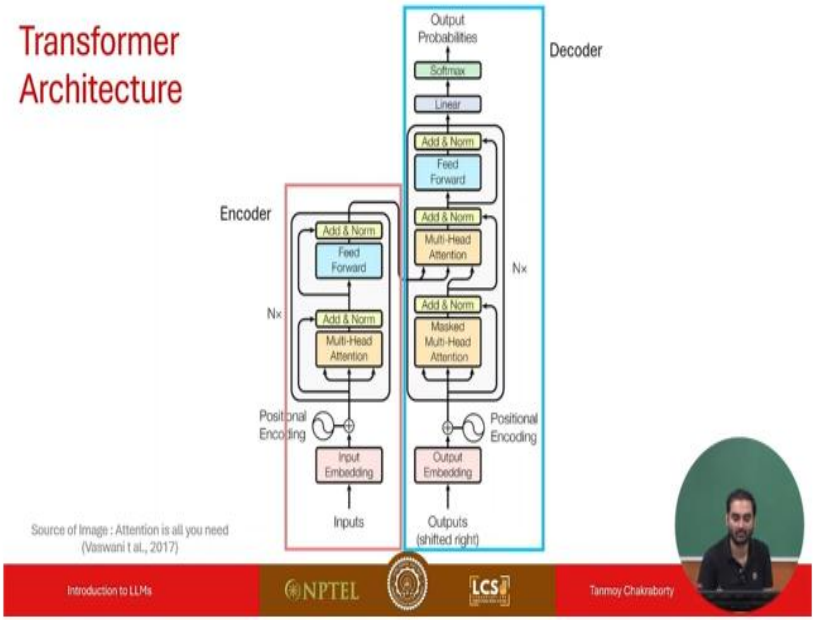
This is the diagonal part the same dimension minus infinity why minus infinity I will tell you. If you multiply this, then what will happen? This part will be replaced by minus infinity right in the resultant matrix and the lower part will remain the same And this part will be minus infinity and this will remain the same. Now why are we doing this minus infinity multiplication? Because remember, once we saw, so these elements are query key dot product, right? After query key dot product, what is the next operation? Short max, we need to pass them through short max, right? And what is the short max formula e to the power x? Minus dot product, right? So when you multiply this with minus infinity it will produce what? Let's say there is an element, let's say e to the power minus infinity is what? 0, after short max this part will become 0 and the remaining part will essentially follow the short max rule. So in this way you will not allow the model to look to basically attend to the future states. Now this is happening during the training time when you have access to the future states also.

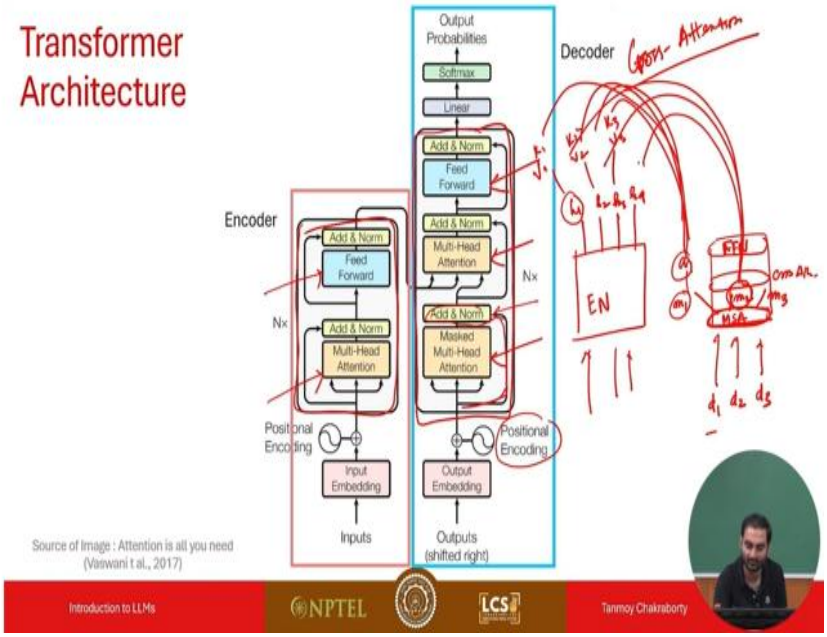
Now during the test time what will happen? Now during the test time you already learned WQ, WK, WV. The WQ, WK, WB are learned based on all these things. So you pass, let's say you generated the first token, which is I. you generate the first token, that token will be passed through all these weights, layers and so on, it will produce a distribution, from the distribution you sample one token and then that token will be passed through again the

same set of parameters, it will generate another token and so on and so forth. By the way so far I only mentioned that this layers are stacked right, self-attention feed forward, self-attention feed forward.

So the last layer will also have self-attention feed forward right okay and the outputs will basically be vectors. Now I hope you remember the difference between the unmasked self-attention in the encoder and masked self-attention in the decoder. unmask self-attention you have access to the entire sequence, mask attention you don't have access to the future. There are some parts which are still missing. So far whatever we discussed, so far encoder block, encoder stack and decoder stack, they are not linked.

Are they linked so far? Whatever we discussed so far. encoder blocks in encoder sequence is processed independently with a set of parameters matrices WQKV and this is also processed independently. Parameters are also not shared. So I need some mechanism which can connect the decoder with the encoder. How do we do that?





We will do that using another layer called cross attention.

What is a cross attention? This is encoder and let's say these are final outputs. Let's say H1, H2, H3, H4, last layer and this is my decoder block. In the decoder block, first component is what? What is the first component? The first component is masked self-attention. And we also discussed feed forward network, FFN.

And all these attentions are multi-headed. In between this, we will introduce another sub-block called cross-attention. In cross attention what we will do? Let's say this is the decoder D1, D2, D3 and after the mask self-attention you obtained let's say M1, M2, M3. So here also I will apply this self-attention concept, query key value. But here the query, so unlike the previous self-attention where every token generated a query and a queue and a value, here the query will come from the decoder token and key and value vectors will come from the encoder token. Meaning that from here you will generate key say K1, V1, K2, V2, K3, V3.

How do you generate? Again you need another set of parameters and from M1 we generate Q1. So, Q1 will attend to all the keys in the encoder. Similarly Q2 which will be generated from here, this will attend to all the keys in the encoder and so on and so forth. This is

called cross attention. Is this masked or unmasked? Do I need to mask it or unmask it? There is no concept of masking at all here because here decoder states are not attending to each other.

Decoder state will act as a query and encoder states will act as keys and values. It is not that a decoder state will attend to another decoder state. So the masking is not needed at all. So this constitutes, so now we are done with a single decoder block of transformer. And look at here, this is a single decoder block of a transformer. multi headed mask multi headed self attention then this is cross attention and this is feed forward network and this is the block of an encoder we have unmasked multi headed self attention and feed forward network.

Now you repeat these blocks in number of times. We missed out few details here for example we missed out this positional encoding. we missed out the details of this add non-block right, we also missed out the details of this links this is called residual connection. So, residual connection add non-block and positional encoding we will discuss in the last in the next lecture. Thank you.