66.1 Transformers and BERT

We already studied encoder-decoder models, attention mechanisms, LSTM and CNN.

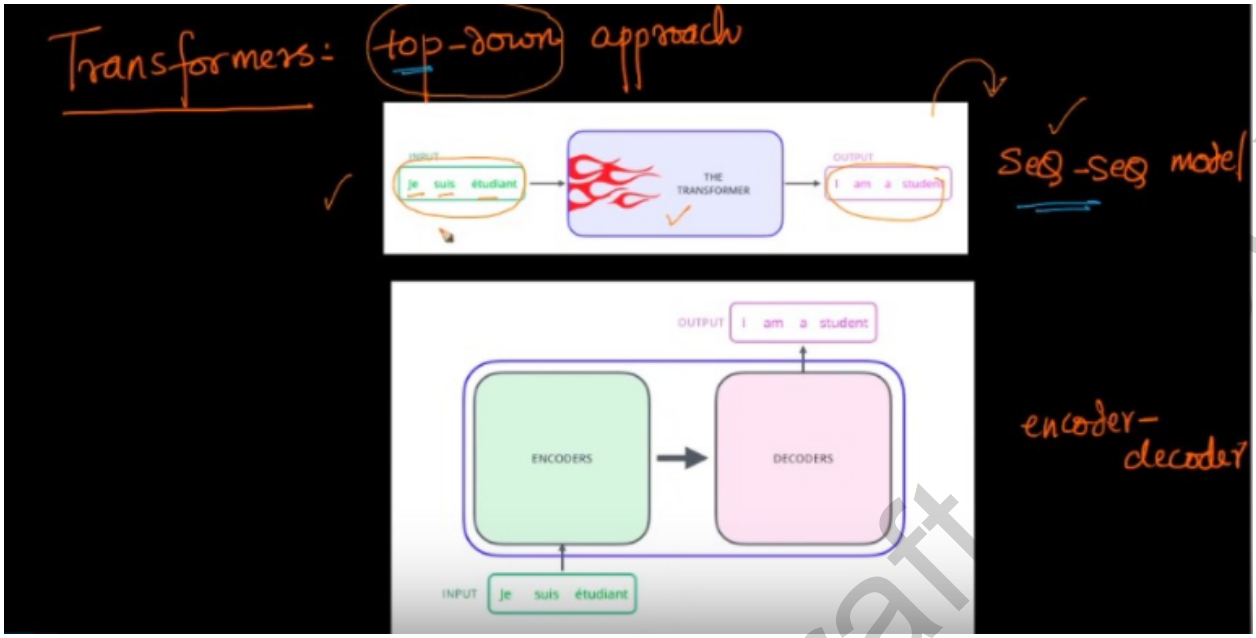
In this session, we are going to study about transformers and BERT. To understand BERT, we need to have a solid understanding of transformers.

To understand transformers, we need to have a solid understanding of attention mechanisms and encoder-decoder models.

We will follow a top-down approach in understanding transformers.

Transformers : It's basically a sequence to sequence model. We will see how it's defined.

1.) We have an input which is a vector in Rd (d -dimensional real space).
2.) We have a model which is nothing but the transformer.
3.) After passing the input to the model, we get an output which is also a vector in Rs (s-dimensional real space).



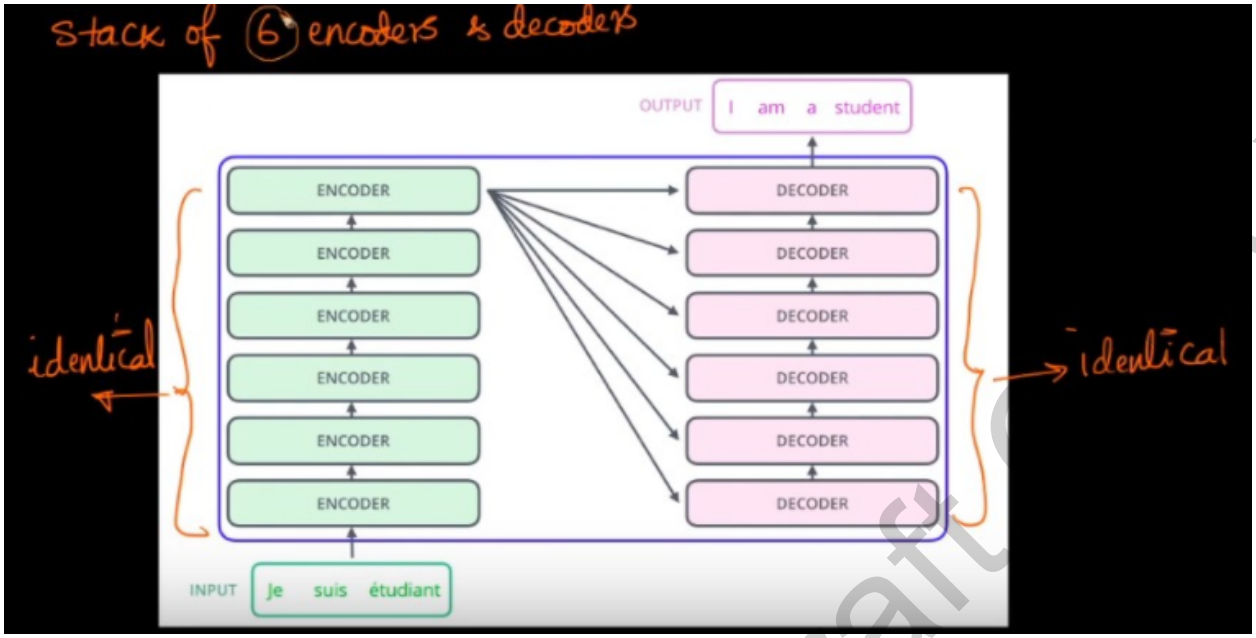
Timestamp : 05:08

An application where sequence to sequence models are used is language translation. For example, if we want to translate sentences in English to French, we can use transformer based models. Sequence to sequence model is an abstract concept where we have inputs,outputs and the model. The inputs and outputs are both sequences. One such model which implements this is transformers.

Let’s see what is inside the transformers.

We can see this from the second block diagram given above. It has both encoder and decoder models. The encoder takes the input and the decoder generates the output.

Let’s now look at how encoders and decoders are defined.



Timestamp : 07:04

If we unwrap the encoder model, we can see a sequence of 6 encoders being defined. The same goes for the decoder too.

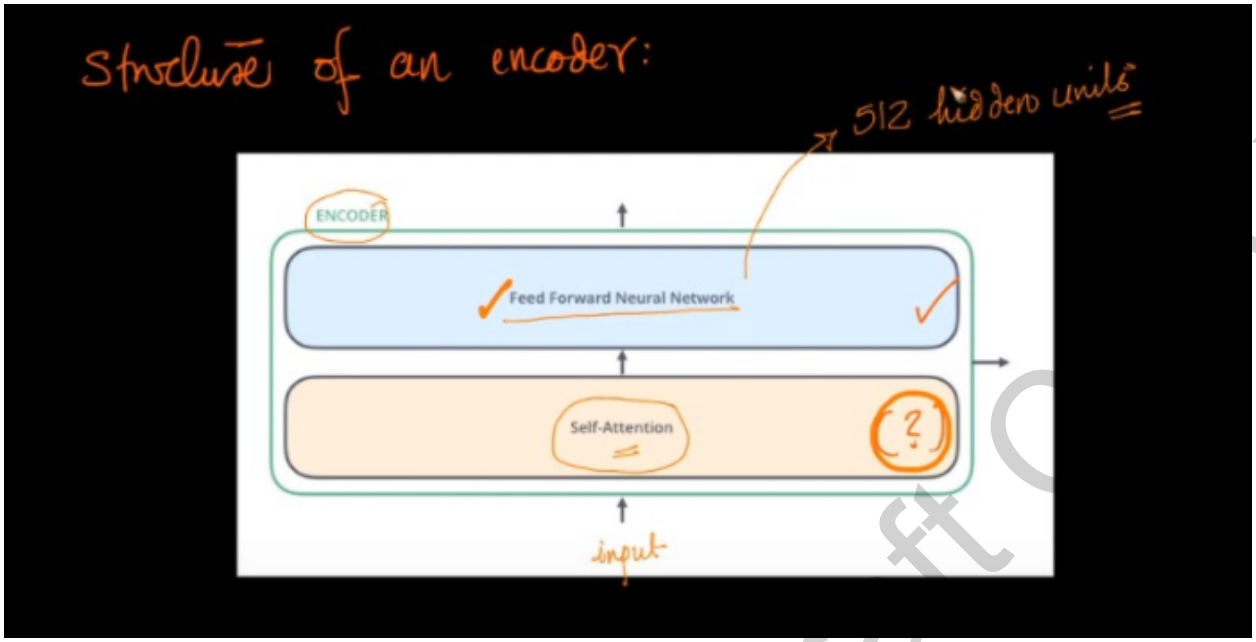
Why did we define a total of 6 models ?

In the research paper, they suggested using it. But in actual practice, you can choose the appropriate number. The condition is that the number of models should be the same for both encoder and decoder.

All the encoder models are identical and the decoder too. The encoder and the decoder are different.

We already know the flow of input in the sequence models. First, the input is passed to the first encoder model. Then, the output of it is passed to the second encoder model. This will continue until the final encoder block is reached. Then, the final output is sent to all the decoder models. Then, we get the final output.

Structure of encoder:



Timestamp : 12:26

The encoder block is divided into two blocks. They are namely self-attention and feed-forward neural networks. We already know feed-forward neural networks. In this paper, they suggested using 512 hidden units in this network.

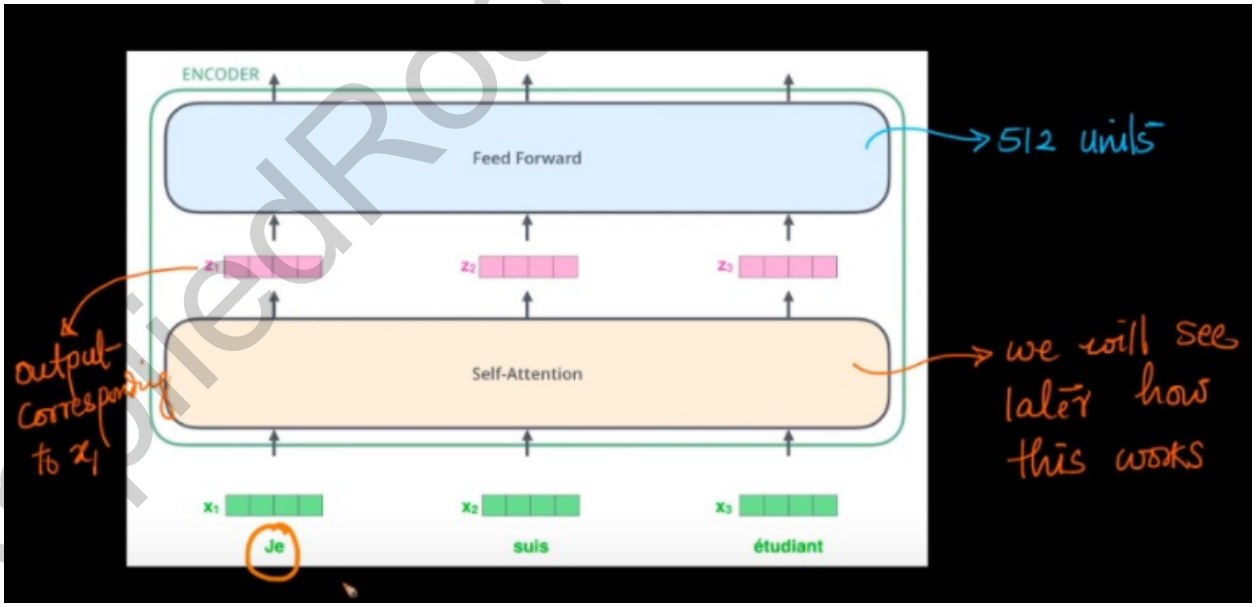
Now, we will discuss self-attention mechanisms.

Let’s recall our task. Our task is to perform language translation.

The inputs are actually a sequence of words i.e, a sentence. The model can’t interpret the words directly. It is defined to work with numbers. So, we need a procedure or algorithm which can convert these individual words in a sentence to a numerical representation. One such way is to use Word2Vec models. In simple terms, given a word, the Word2Vec model returns a vector belonging to Rd (d-dimensional real space).

Timestamp : 13:46

In the above illustration, we can see the word Je being represented as a 512 dimensional vector. The same goes for the other words too. This is achieved by the Word2Vec model. So, now we have a procedure to transform a word to a d-dimensional vector. Then, let's dive deeper into the actual self-attention mechanism.

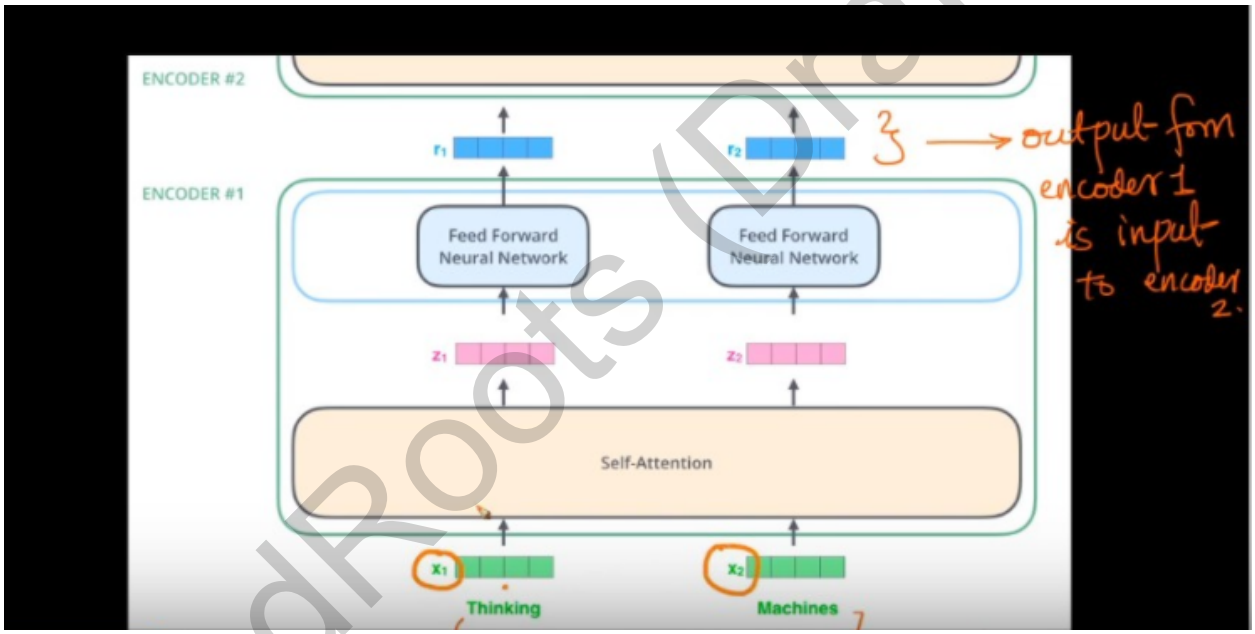


Timestamp : 14:28

Now, we are passing all the inputs to the self-attention block at once. This is quite different from RNN,LSTM networks etc. In RNN, we used to pass the inputs at each timestep only if the previous hidden state had been computed. So, we are dependent on the time. Here, we simply pass all the inputs at once.

Next for each input xi, we get a corresponding output zi. Still, we didn't unwrap the self-attention mechanism. As we mentioned earlier, we are following a top-down approach.

From now onwards, we will stick to the example given below.



Timestamp : 16:24

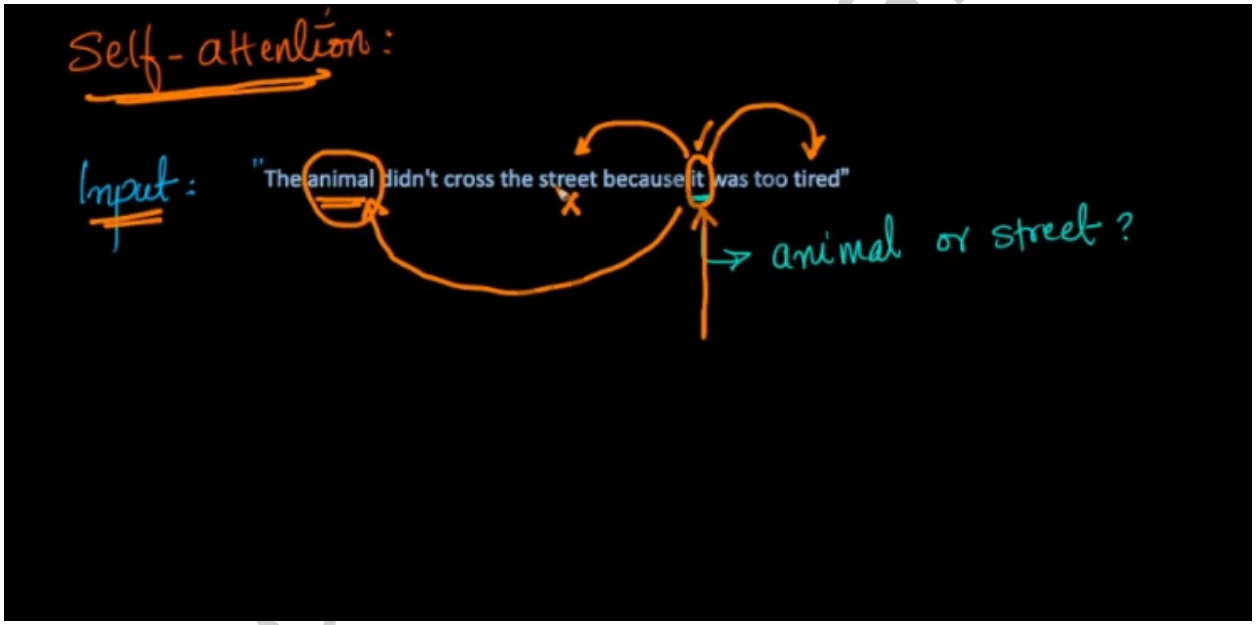
So, we have only two words.

We already discussed how zi are computed. We still didn't discuss self-attention mechanisms. So, now each zi is transformed into ri by a feed-forward neural network. Until now, we have discussed how xi is transformed into ri. Now, we also have the remaining 5 encoder blocks as mentioned earlier. So, now ri acts as the input to the second encoder block. This continues until we reach the final encoder block. This is how sequence

models work right ? The output of the previous model in the sequence is passed as the input to the next model in the sequence.

Self-attention :

Let's assume that the input sentence is "The animal didn't cross the street because it was too tired" . Let's understand this sentence. What does "it" refer to here ? Does it refer to the animal or street ? It's quite easy to find. The "it" here refers to the animal. Because, animals can only be tired and not the street. So, while reading the sentence we automatically understand that the word "it" here refers to the animal.



Timestamp : 19:09

What conclusions can we draw from the above example ?

1.) The word "it" refers to the animal. Even Though there is a gap between these words, there is still a relationship between them.
2.) So, generally each word in a sentence has some kind of relationship between the other words. The relationship might be strong or weak.

For example, the word “it” and “animal” had a strong relationship. But the word “it” and “street” had a weak relationship.

3.) While generating zi from xi, we need to incorporate these things. For example, while generating zi for the word “it’ , we need to give more importance to the words like animal, tired.

Why is it called self-attention ?

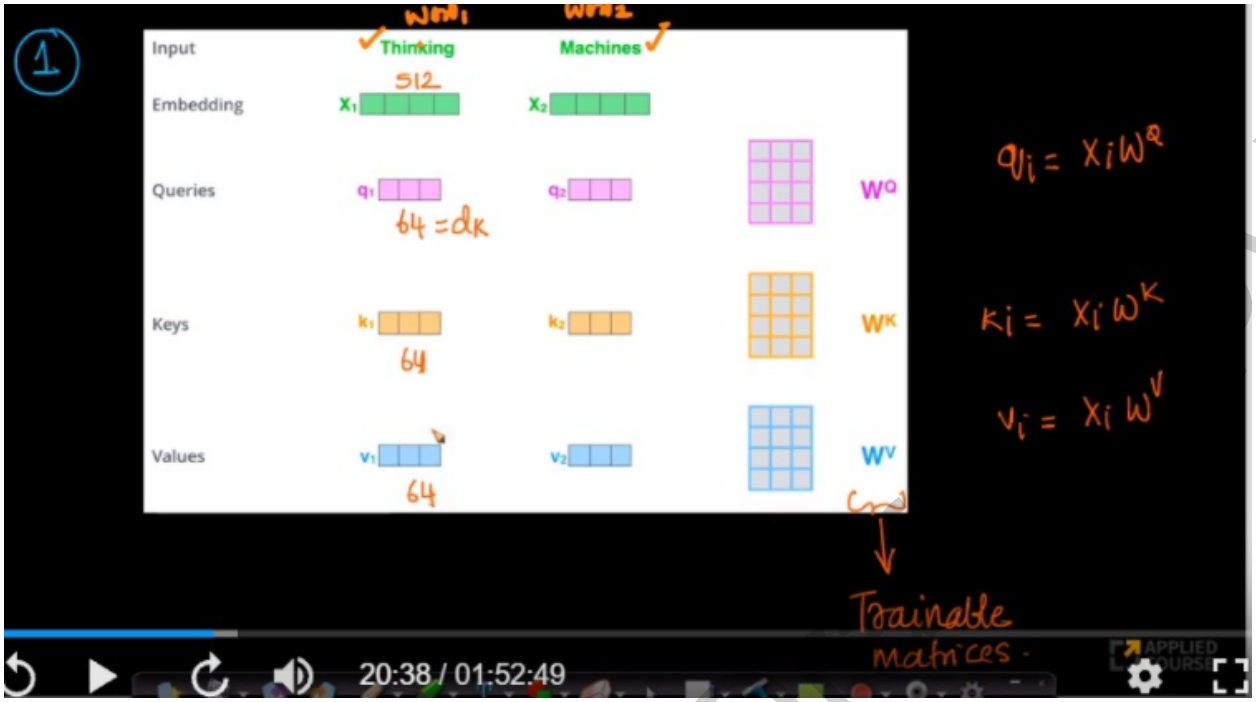
We are basically using the words in the same sentence to generate zi for each word xi.

Until now, we have a good understanding of how a word is dependent on other words in the same sentence.

Why do we actually care about it ?

To translate one language into another, we first need to understand the source language well. Without understanding it, we can’t build a good translation model or are unable to translate it well. Here, self-attention produces zi which are refined versions of the word2vec vectors i.e, xi’s.

How does it work ?



Timestamp : 20:38

In the self-attention mechanism, we have three trainable parameters namely W_Q , W_K and W_V .

W_Q (Query matrix) : Weight matrix corresponding to the queries.
W_K (Key matrix) : Weight matrix corresponding to the keys.
W_V (Value matrix) : Weight matrix corresponding to the values.

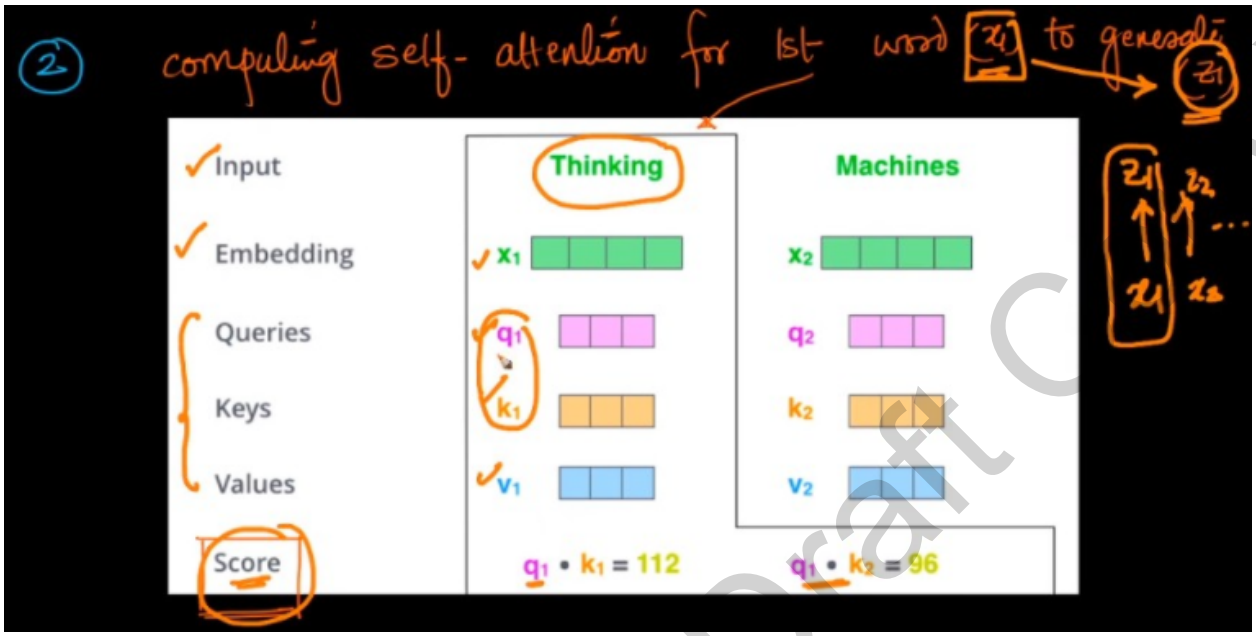
Each xi is transformed into qi,ki and vi.

qi = W_Q.xi
ki = W_K.xi
vi = W_V.xi

where . refers to matrix multiplication. This is nothing but a single layer neural network with linear activation function (z = W.x).

Computing the self-attention :

Right now, we will compute z1. The same logic works for z2 too.



Timestamp : 26:01

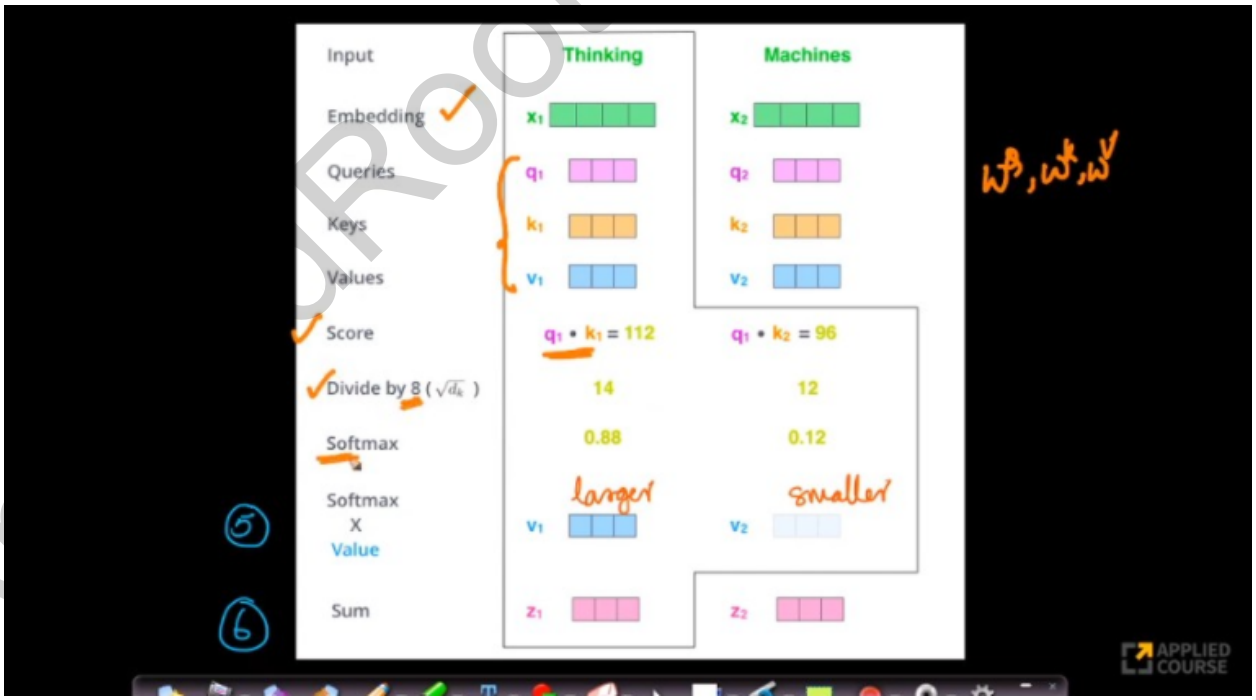
1.) Here, we are considering only two words x1 and x2 and we are going to compute z1.
2.) First, we will compute the qi,ki and vi for x1 and x2. It's already been discussed.
3.) Next, we need to calculate the score. This is done by computing the dot product between q1 and ki for all i. For example, score_1 = q1.k1
4.) To calculate z1, we are computing the score which is nothing but calculating the dot product between q1 and ki for all i. Basically, each word xi is represented in three different forms i.e, as a query,key and value. To find the relation between a word w and all other it's neighbour words in the sentence, we need to calculate the similarity or how similar it is to another word. This is what we want right ? We already saw in the previous example that the word "it" has some relation to the word "animal". This score actually establishes that relationship between the words.
5.) For a word x1, we have scores with other words right ? Here, the length of the score vector is 2 since we have only two words in a sentence. Each of the vectors qi,ki and vi are 64 dimensional vectors.

Next, we need to divide each element in the score vector by 8 which is nothing but the square root of 64 or generally the dimension of the query vector or key or value vector. The reason is to produce more stable gradients. While training, the researchers have encountered some problems with the gradients before normalizing it. So, to ensure stable gradients, we normalize them. Also, the dot products go larger in magnitude. This will cause problems while calculating the softmax function (it has the function **e** in it) and affects the gradient. To counter this effect, we are dividing by the square root of the dimension.

6.) Then, we pass the score vector to the softmax function to get the probability estimates. These values will be easier to interpret rather than the score itself.

7.) Assume the probability score vector computed in the 6th step is represented by p = <0.88 0.12> (We have only two words).

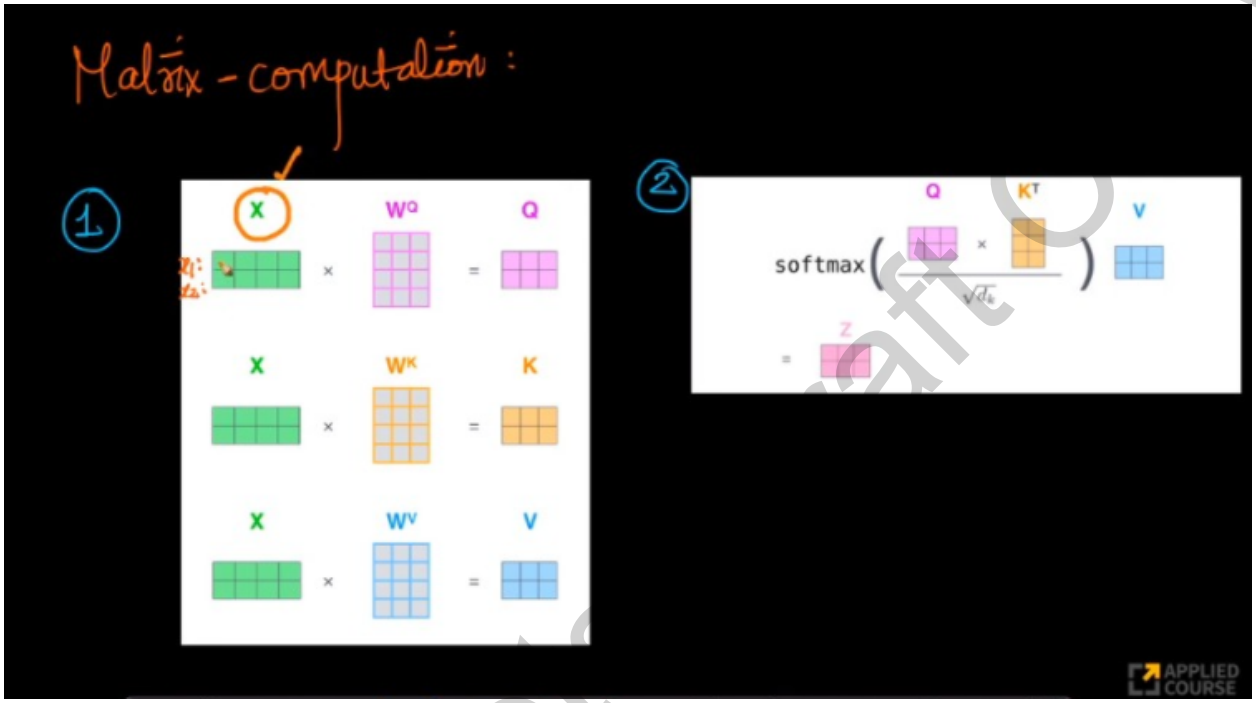
8.) z1 = p[i]*vi for all i and summing across it. v1 is the value vector corresponding to the word x1. To be precise a is 0.88*v1 + 0.12*v2.



Timestamp : 31:04

The same steps apply for computing z2 too.

Now, we are going to generalize the steps mentioned above in a compact notation.



Timestamp : 31:52

Here, the X matrix contains both x1 and x2.

First, we compute the Query matrix by performing the matrix multiplication between X and W_Q.

Similarly, we calculate the Key matrix and the Value matrix.

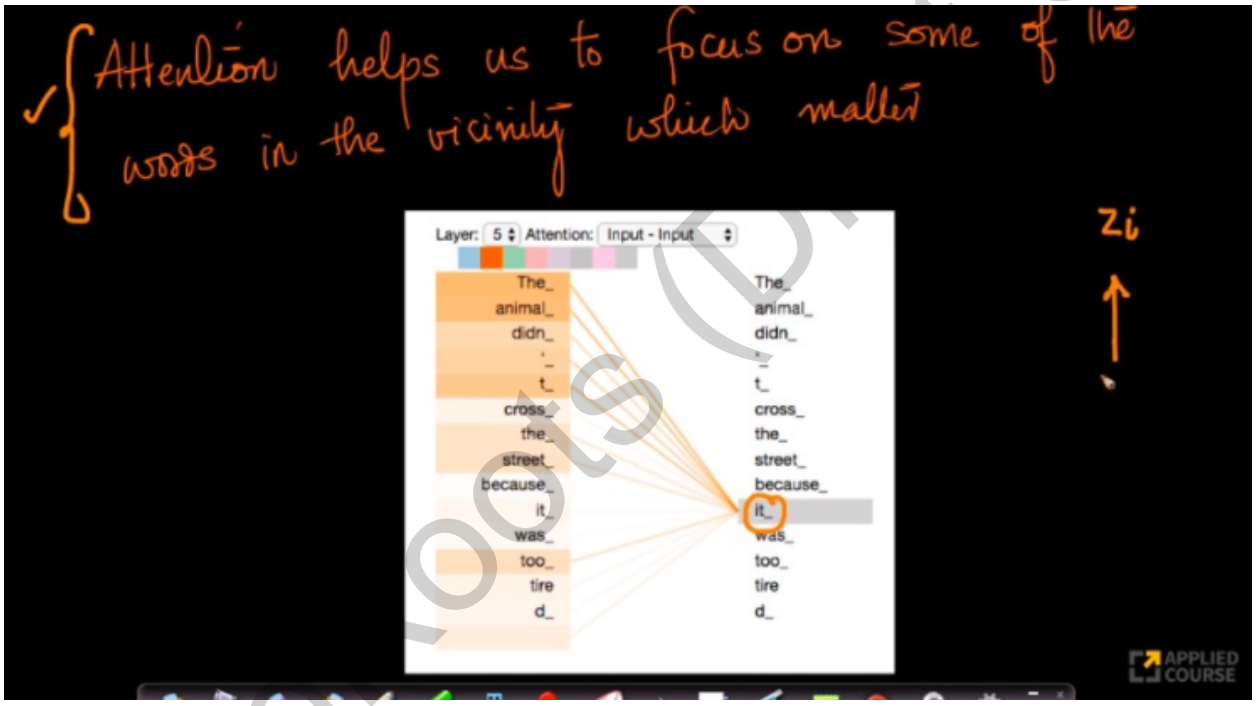
Now, we compute the matrix multiplication between the Query matrix and the transposed version of the Key matrix. This gives us the score matrix.

Then, we scale every entry in the matrix by 1/d where d is the square root of the dimensionality of the query vector.

Then, we apply the softmax function on top of it to get the probability estimates.

Then, we perform the multiplication with the value matrix to get the Z matrix.

The concept still remains the same. Only thing is we are using matrix notation to make our life simpler. Also, it speeds up the computation since we can leverage parallel computing techniques.



Timestamp : 34:51

In the diagram given above, we can see the relationship between the word “it” and other words in the sentence. The strength of the relationship is represented by how strong the color encoding is. We can see that the line between the word “animal” and “it” has strong color encoding. But, the line between the word “it” and “was_” has a dull color.

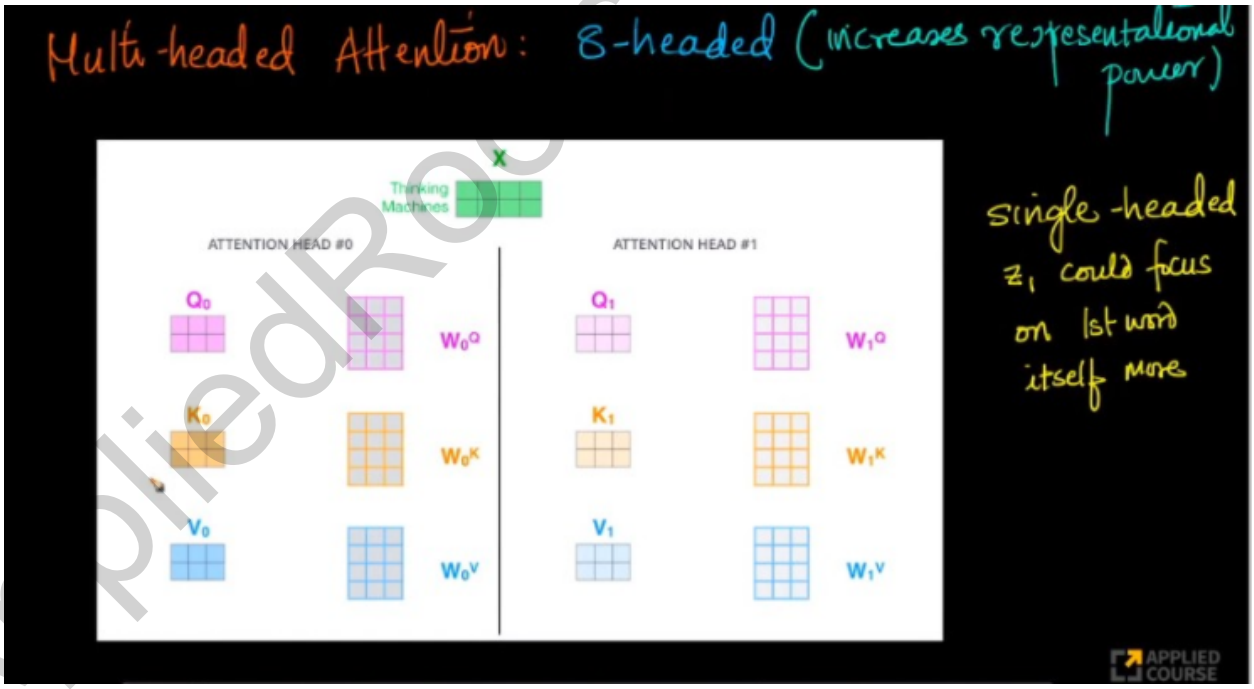
How are they color encoded ?

They simply used the final probability scores. Depending on the values, the colors are proportionally given. Basically, the darker the color is the larger the magnitude of the probability score.

This shows that the model has well understood the relationship between the words. This is what we want right ? This is actually similar to word2vec. In word2vec, if two words are similar (if the two words are in the neighbourhood specified by the window) then their vector representations are also similar. So, here, we give a high probability score to those words which are similar or hold a strong relationship.

Multi-headed Attention :

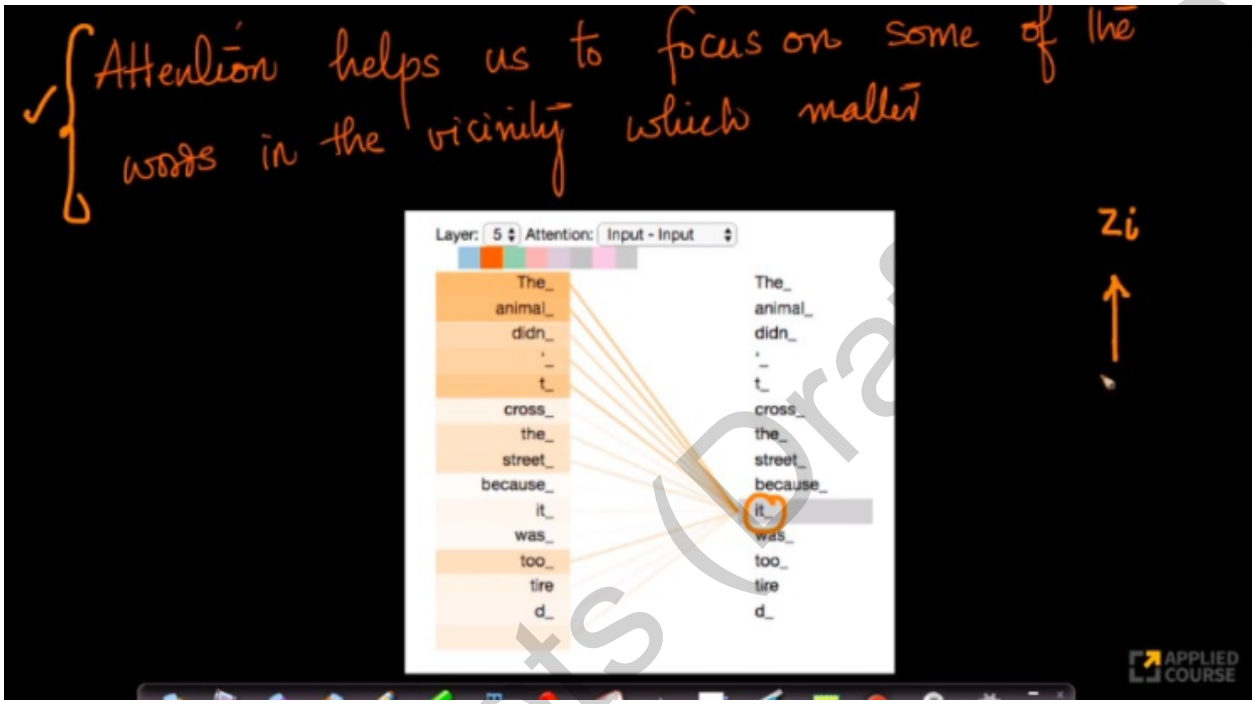
It's a simple concept. So, rather than having a single set of weight matrices for computing key,values and the queries, can we have multiple sets of it ? For example, please refer to the below image.



Timestamp : 40:55

We have W_i_Q,W_i_K and W_i_V for i ranging from 1 to 8 since it's an 8-headed attention.

Why do we need it ?



In the above image, we can see that the word "it" and "animal" have a strong relationship. But the same word "it" and "tired" have a weak relationship. Basically , the word "it" refers to the word "animal" and the animal is tired. So, we want the relationship between the word "it" and "tired" to be also strong or at least not weak. This is actually a complex relationship to discover right ? We as humans can understand. But, for the model to understand this, we need to make it complex. So, we go for multi-headed attention. What we expect is that, in other heads, this relationship is discovered and we get a strong relationship between the words we wish for. But there are also other hidden relationships present. For this only, we go for 8-headed attention. But, here the number of heads is a hyperparameter. It depends on the dataset. Multi-headed attention increases the representational power.

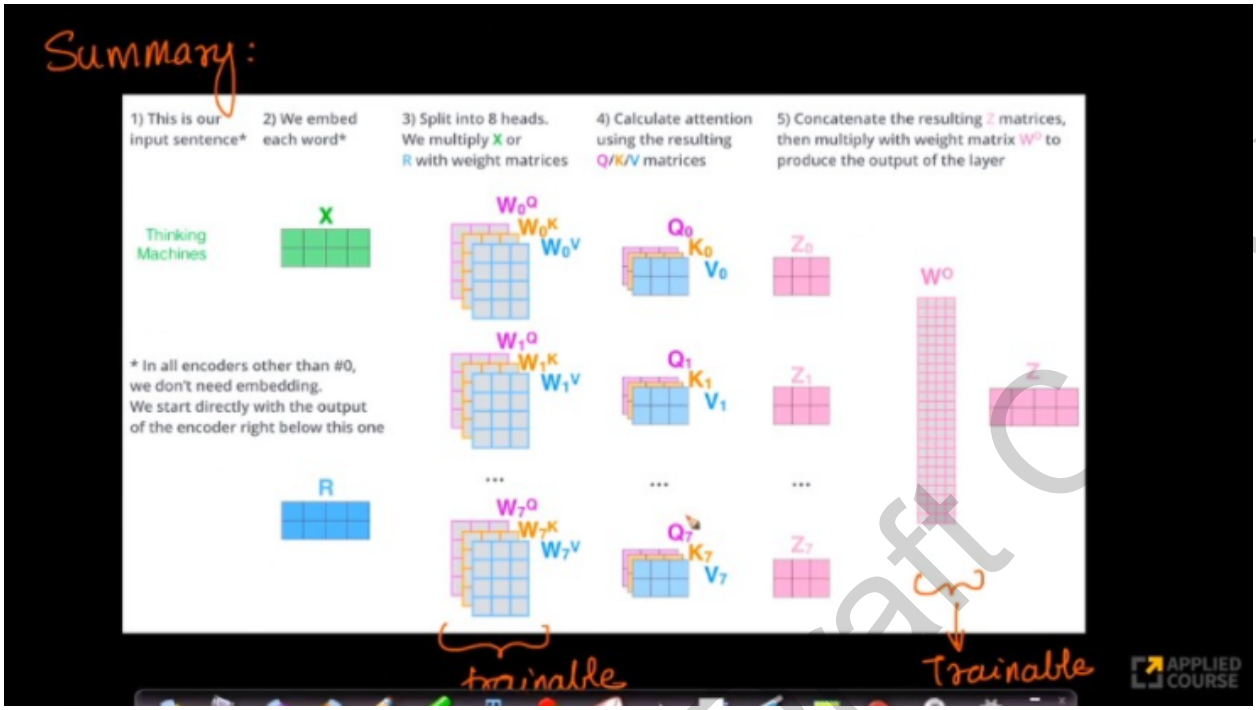
Timestamp : 42:35

Basically, for each head , we have a corresponding Zi right ? We already discussed how to compute Zi. Note : Here Zi is the matrix and it contains the vector representation for every word in the sentence. Please refer to the matrix notation we discussed earlier.

To get a single Z matrix from all the Zi's, we simply stack them as described above and then multiply with the matrix W_O to get the final Z. Here, W_O is the trainable matrix.

Note : A single encoder block contains multiple attention heads. Don't get confused with this.

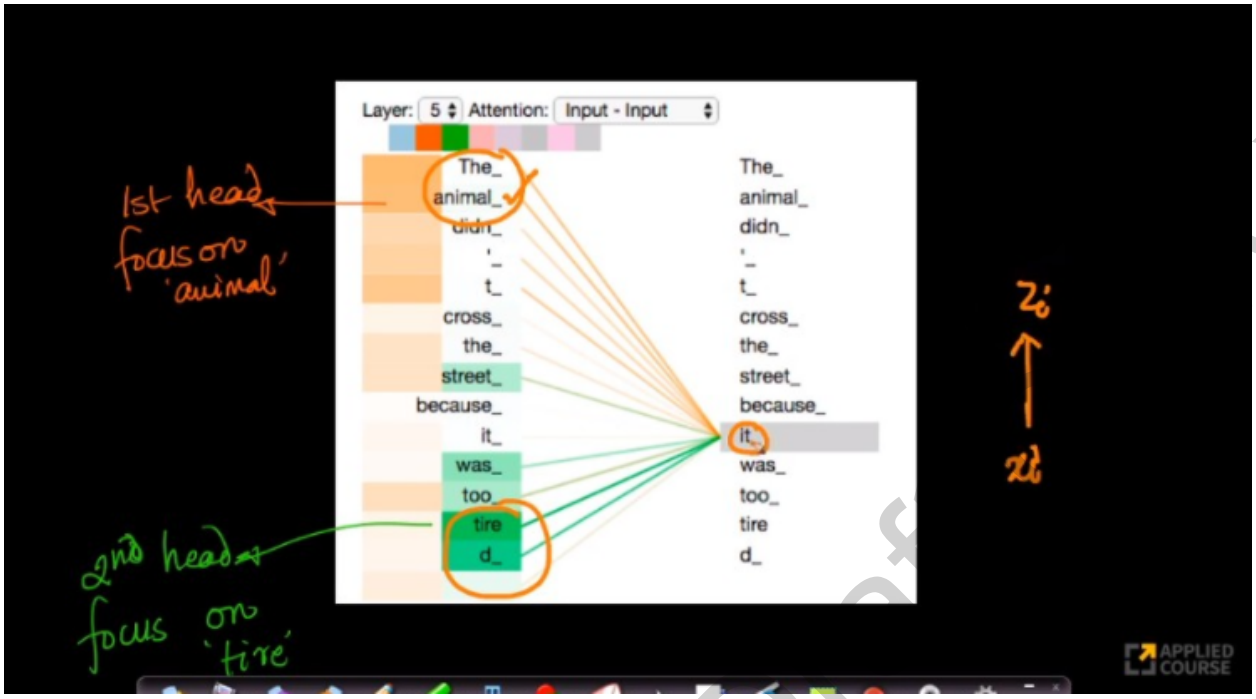
Summary:



Timestamp : 44:42

1.) We have the input matrix X where the embedding vectors are stacked.
2.) We have multiple attention heads each having three parameters namely W_i_O, W_i_K and W_i_V where i is the index specifying the attention head.
3.) Then, we compute the Qi,Ki,Vi as we discussed earlier.
4.) With this, we compute the Zi's.
5.) Then, finally we compute the Z using the matrix W_O.
6.) All the matrices mentioned above are trainable and updated using backpropagation.
7.) This is just an overview of the entire process and the details are mentioned earlier.

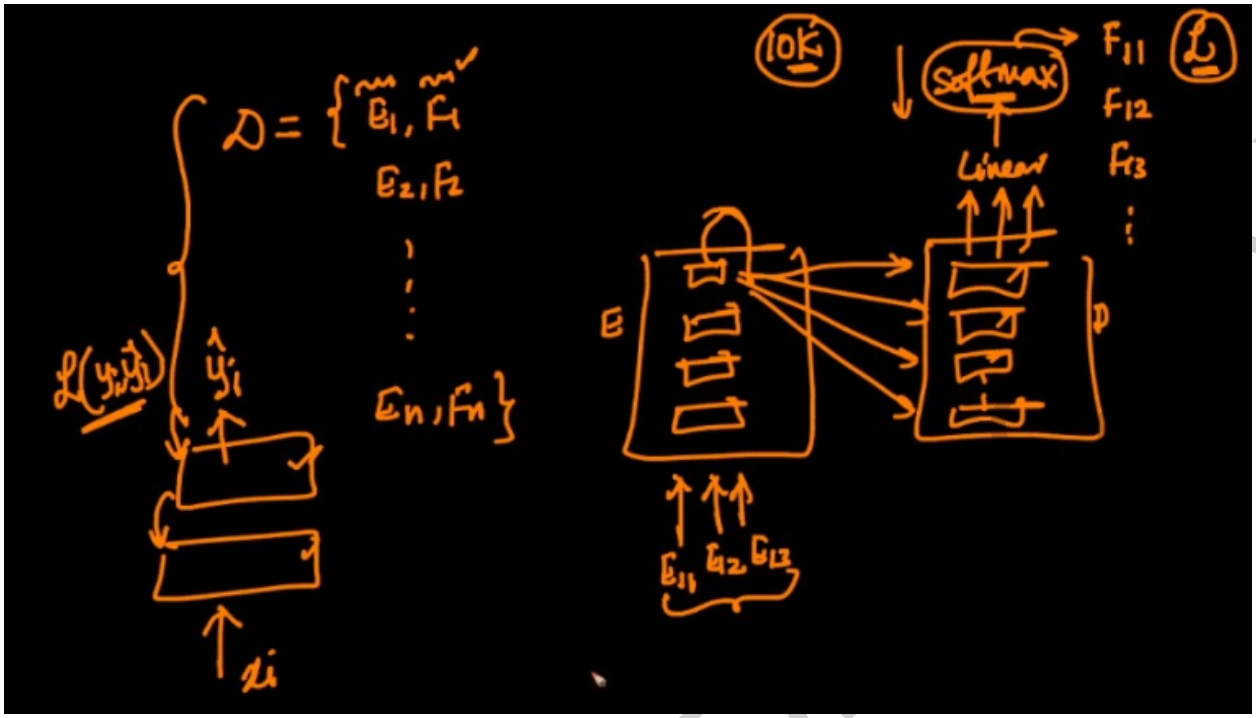
We already discussed why we need multiple attention heads. Now, we will see empirically the result.



Timestamp : 46:29

As mentioned earlier, in the second attention head, the word “it” has a strong relationship with the word “tired”. This result shows that having multiple attention heads can be helpful in discovering multiple relationships.

Overview of the model :



Timestamp : 51:28

We have our Dataset D which is a collection of pairs of sentences in English and french. D = { (Ei,Fi) } for i ranging from 1 to n.

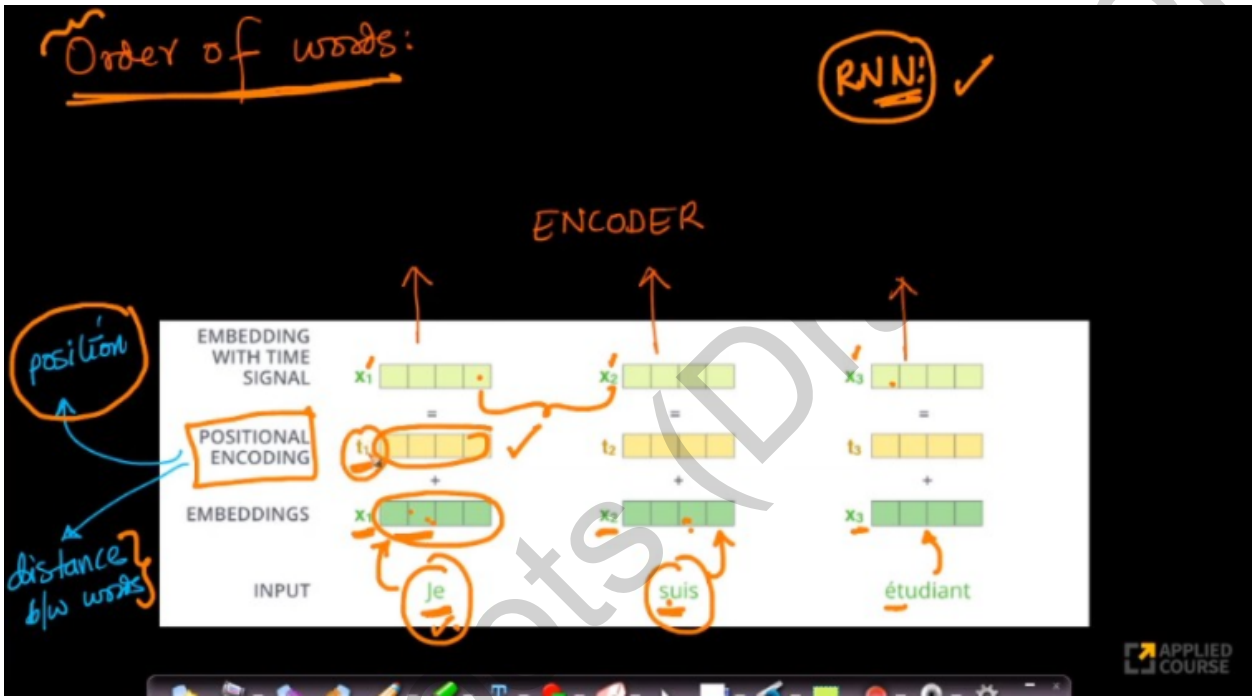
We pass all Ei to our encoder model. The encoder produces the output. This output is fed to the decoder. The decoder then generates the output by a linear layer. The output is then passed to the softmax function to generate the probability scores. Argmax operation is performed to find the corresponding french word in the vocabulary.

We will define a loss for it. Using backpropagation, we calculate the gradients of loss with respect to the trainable parameters and update it.

Positional encoder :

In RNN we have hidden states which represent the temporal dimension. For example, let's say we have a sentence "This course is the best". Let's say we pass this sentence to the RNN. We will get an output right ? Now,

let’s say we shuffle those words. Do we still get the same output ? Definitely not. Because, due to hidden states, we get different output. Please pause and ponder. But in the case of the transformer, we didn’t learn any temporal features from the input. We only learnt how each input is correlated to each other or how the relationship is. To learn the temporal features, we use positional encoding.



Timestamp : 55:48

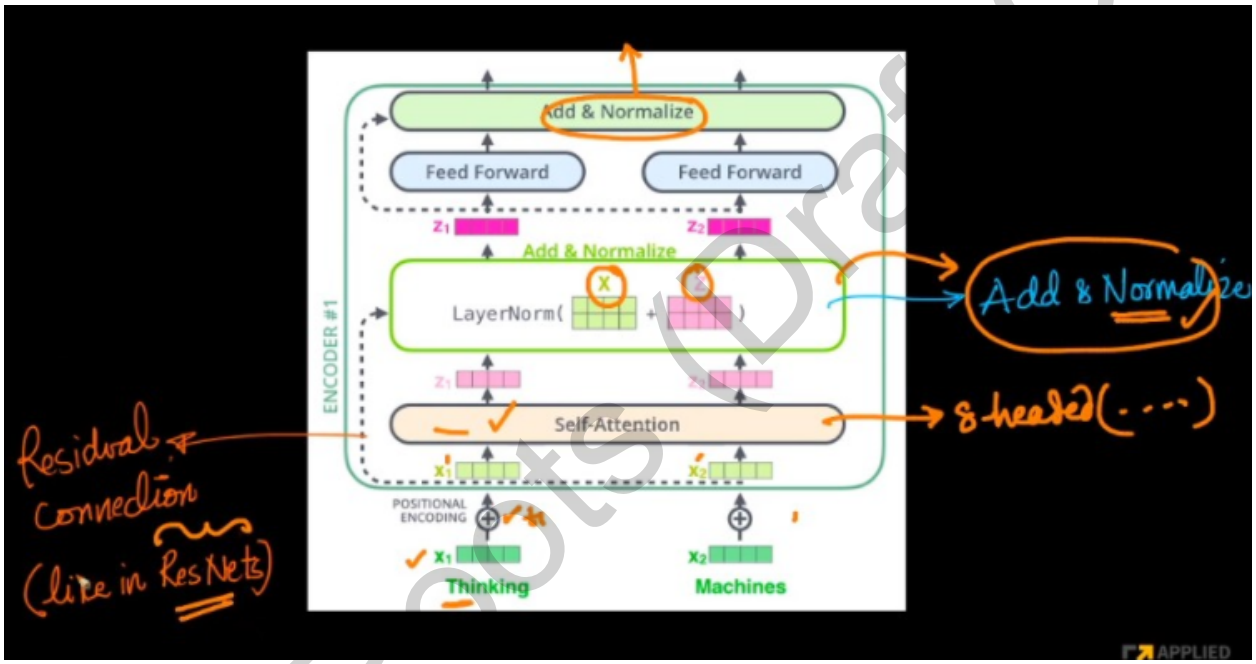
So, for each vector xi, we add a vector ti to it. This ti actually represents the time dimension. Basically, it encodes the fact that the word x_(t+i) comes after x_(t) for any t and i from integer space. Also, it preserves the relation that distance between x_(t+i) and x_(t) is always lesser than distance between x_(t+i+j) and x_(t) where j > i . For example , the distance between the vectors x2 and x1 is smaller than x3 and x1.

How to design ti’s ?

We advise you to read the research paper for this concept. It’s quite mathematically rigorous. We will discuss the intuition behind it.

1.) The distance between t_(i+j) and t_(i) is always less than t_(i+j+k) and t_(i) where k>j.
2.) Also, if we add t_i to x_i , it encodes the positional information.It also encodes the fact that the word x_(t+i) comes after x_(t) for any t and i

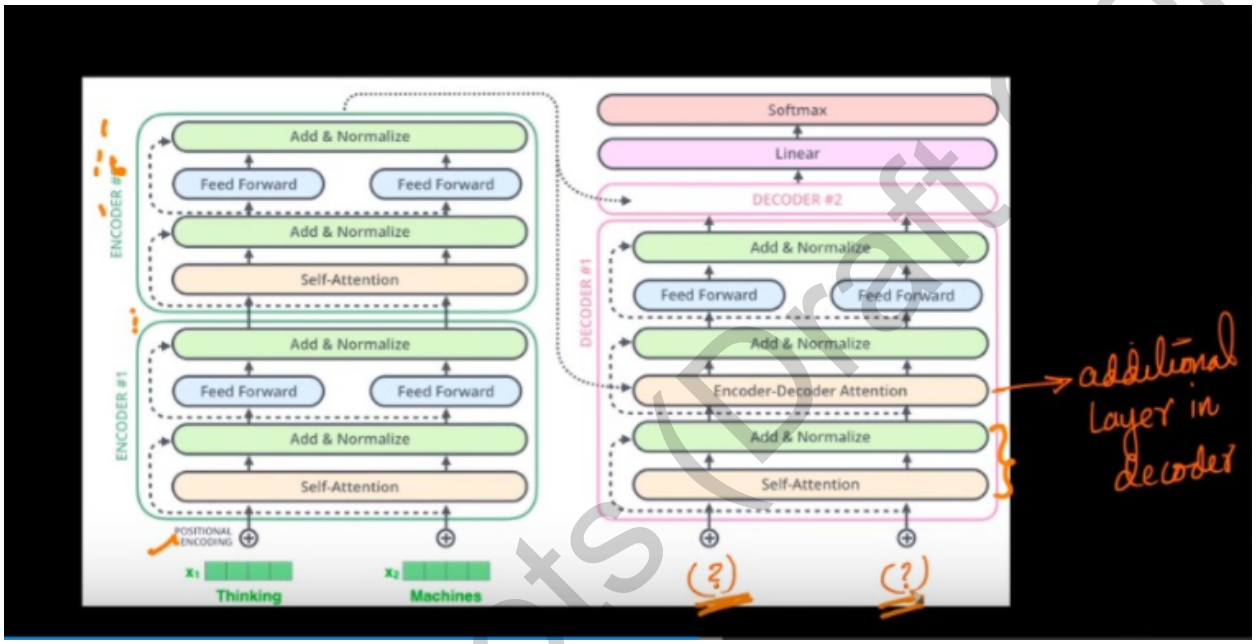
Let's again revisit the encoder block.



Timestamp : 01:01:04

1.) First, we get the embedding vector for each word in a sentence. These are represented as xi. When we have variable length of words in a sentence, we simply use padding. It's already been discussed earlier.
2.) These xi are then passed to the positional encoder where xi are added with ti.
3.) The resultant vector is fed to the self-attention block. This can contain multiple attention heads.
4.) The output of it is represented as zi.

5.) Then, we use residual connection here.We simply add xi to zi and normalize in the layer axis i.e, layer normalization. Basically, if zi is useless, then we can simply have xi itself.
6.) Then, the output zi is then passed to the feed forward neural network.
7.) Again,we perform Add and Normalization. This final output is sent to the next encoder block in the sequence.



Timestamp : 01:03:10

Decoder :

There are some similar layers in the decoder which are present in the encoder. These are namely Self-Attention,Add & Normalize and Feed Forward.

Encoder-Decoder Attention :

The beginning of the decoder is pretty much the same as the encoder. The input goes through an embedding layer and positional encoding layer to get positional embeddings. The positional embeddings get fed into the first

multi-head attention layer which computes the attention scores for the decoder's input.

This multi-headed attention layer operates slightly differently. Since the decoder is autoregressive and generates the sequence word by word, you need to prevent it from conditioning to future tokens. For example, when computing attention scores on the word "thinking", you should not have access to the word "machines", because that word is a future word that was generated after. The word "thinking" should only have access to itself and the words before it. This is true for all other words, where they can only attend to previous words.

We need a method to prevent computing attention scores for future words. This method is called masking. To prevent the decoder from looking at future tokens, you apply a look ahead mask. The mask is added before calculating the softmax, and after scaling the scores. Let's take a look at how this works.

Look-Ahead Mask

The mask is a matrix that's the same size as the attention scores filled with values of 0's and negative infinities. When you add the mask to the scaled attention scores, you get a matrix of the scores, with the top right triangle filled with negativity infinities.
The reason for the mask is because once you take the softmax of the masked scores, the negative infinities get zeroed out, leaving zero attention scores for future tokens. As you can see in the figure below, the attention score for "am" has values for itself and all words before it but is zero for the word "fine". This essentially tells the model to put no focus on those words.

This masking is the only difference in how the attention scores are calculated in the first multi-headed attention layer. This layer still has multiple heads that the mask is being applied to, before getting concatenated and fed through a linear layer for further processing. The

output of the first multi-headed attention is a masked output vector with information on how the model should attend to the decoder's input.
For the encoder-decoder attention layer, the encoder's outputs are the queries and keys , and the first multi-headed attention layer outputs are the values. This process matches the encoder's input to the decoder's input , allowing the decoder to decide which encoder input is relevant to put a focus on. The output of the second multi-headed attention goes through a pointwise feedforward layer for further processing.

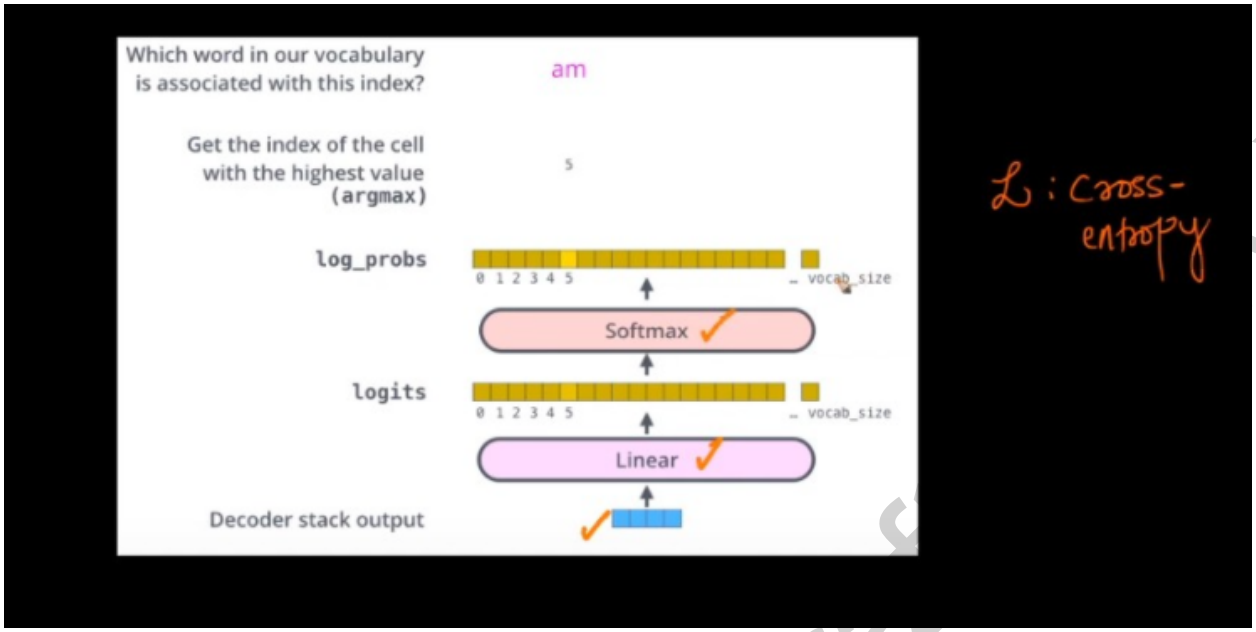
For the diagrams mentioned above, please refer to here

Linear Classifier and Final Softmax for Output Probabilities:

The output of the final pointwise feedforward layer goes through a final linear layer that acts as a classifier. The classifier is as big as the number of classes you have. For example, if you have 10,000 classes for 10,000 words, the output of that classier will be of size 10,000. The output of the classifier then gets fed into a softmax layer, which will produce probability scores between 0 and 1. We take the index of the highest probability score, and that equals our predicted word.

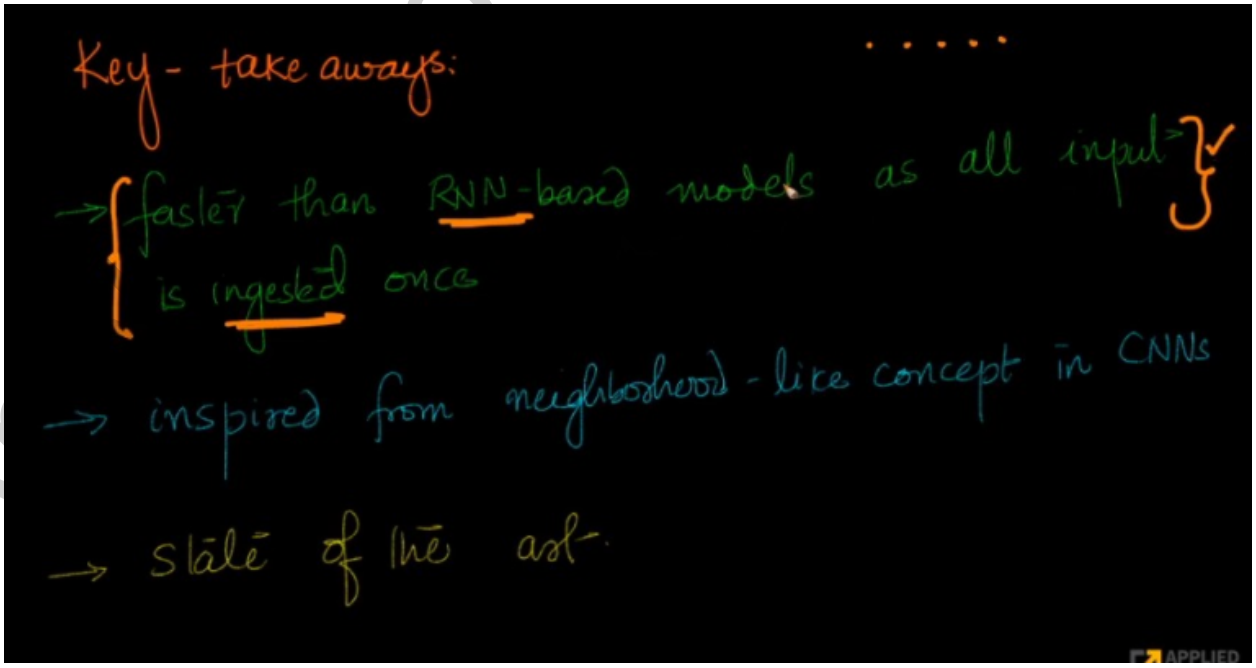
The decoder then takes the output, adds it to the list of decoder inputs, and continues decoding again until a token is predicted. For our case, the highest probability prediction is the final class which is assigned to the end token.

The decoder can also be stacked N layers high, each layer taking in inputs from the encoder and the layers before it. By stacking the layers, the model can learn to extract and focus on different combinations of attention from its attention heads, potentially boosting its predictive power.



Timestamp : 01:10:03

The logits are passed to the softmax function which are nothing but the output of the linear layer. After passing through the softmax function, we get the probability values. The vector is of size vocab_size as mentioned above. Then, we perform the argmax operation to pick the word. We can use the cross-entropy loss function here.



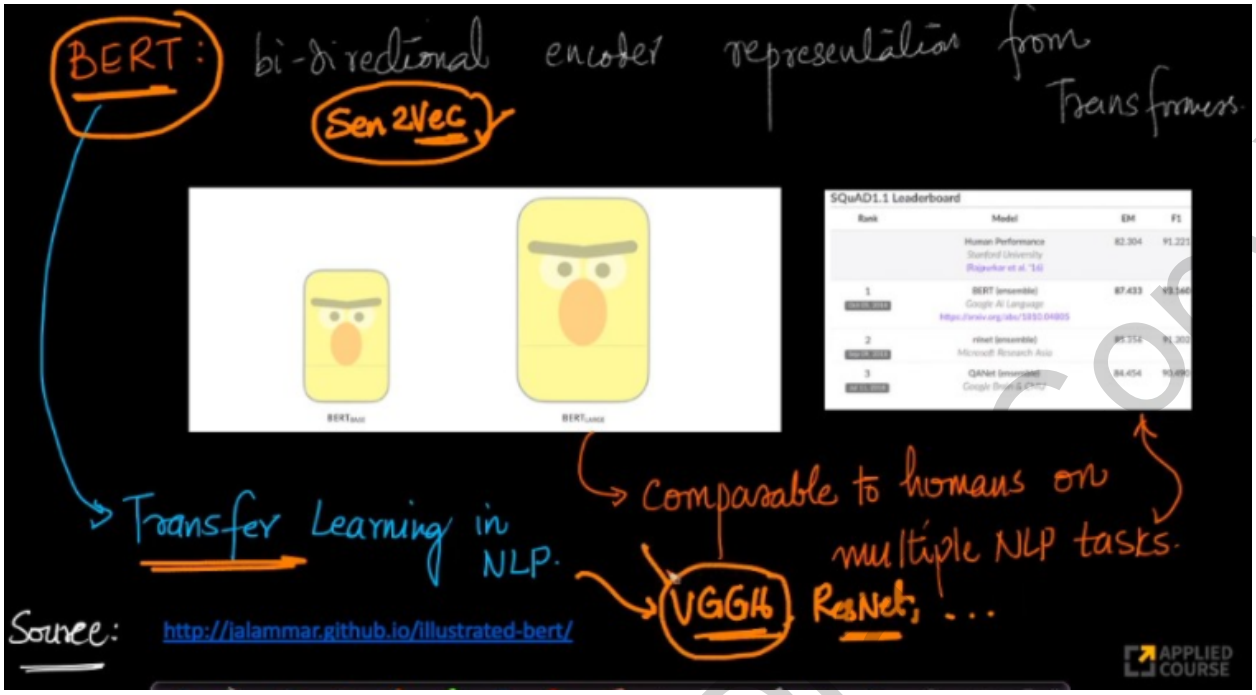
Timestamp : 01:11:05

Advantages :

1.) It's faster than RNN-based models as all input is ingested once. In RNN, we rely on the hidden states i.e, the next output is computed only if the previous hidden state is computed. In the transformer, it's not the case. Here, we pass all the inputs at once and can easily leverage techniques in parallel processing. This is because it simply involves matrix multiplication.
2.) It's inspired from the neighborhood concept in CNN's. Here, while calculating the self-attention we are basically computing how similar the words are. In CNN, we have the freedom to choose a number of filters right ? Here, we are free to choose the number of attention heads. They both share the same. When we have more attention, then basically we can learn complex relationships. Similarly, if we have more filters, then we can learn many patterns from the data.
3.) It's state of the art technique.

BERT : Bidirectional Encoder Representations from Transformers.

Objective : Given a sentence, we need to produce a vector representation of it.

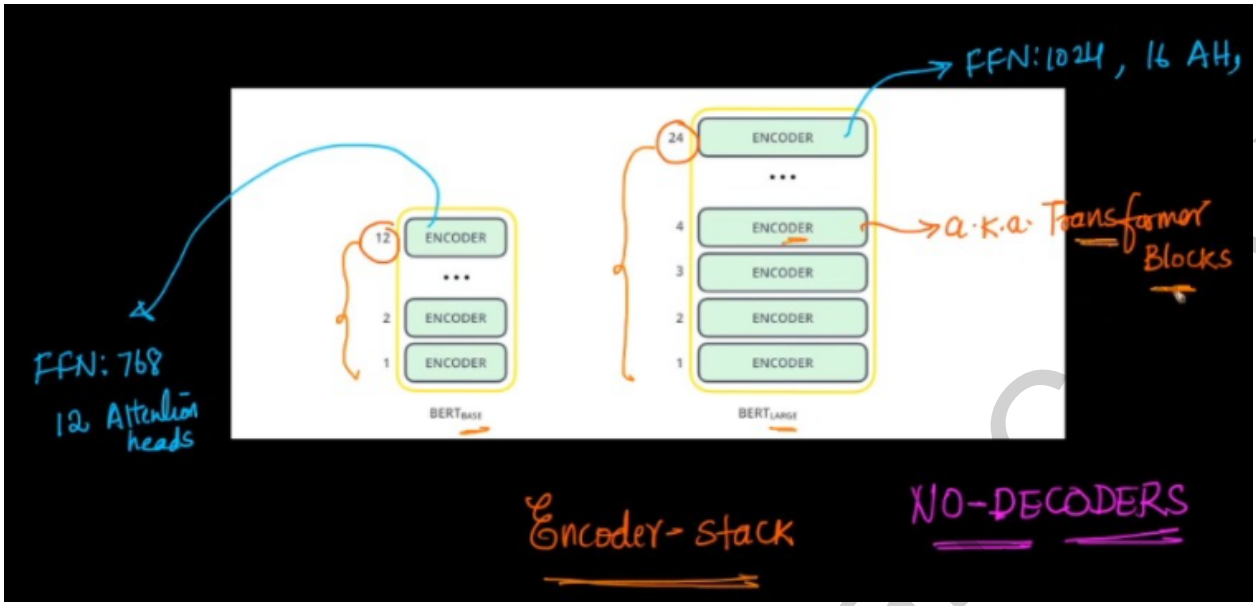


Timestamp : 01:27:19

For example in computer vision tasks, we use VGG16, ResNet trained on other datasets for our purpose. This is called transfer learning. This was not quite possible for nlp tasks.

There are two versions of BERT namely BERT base and BERT large.

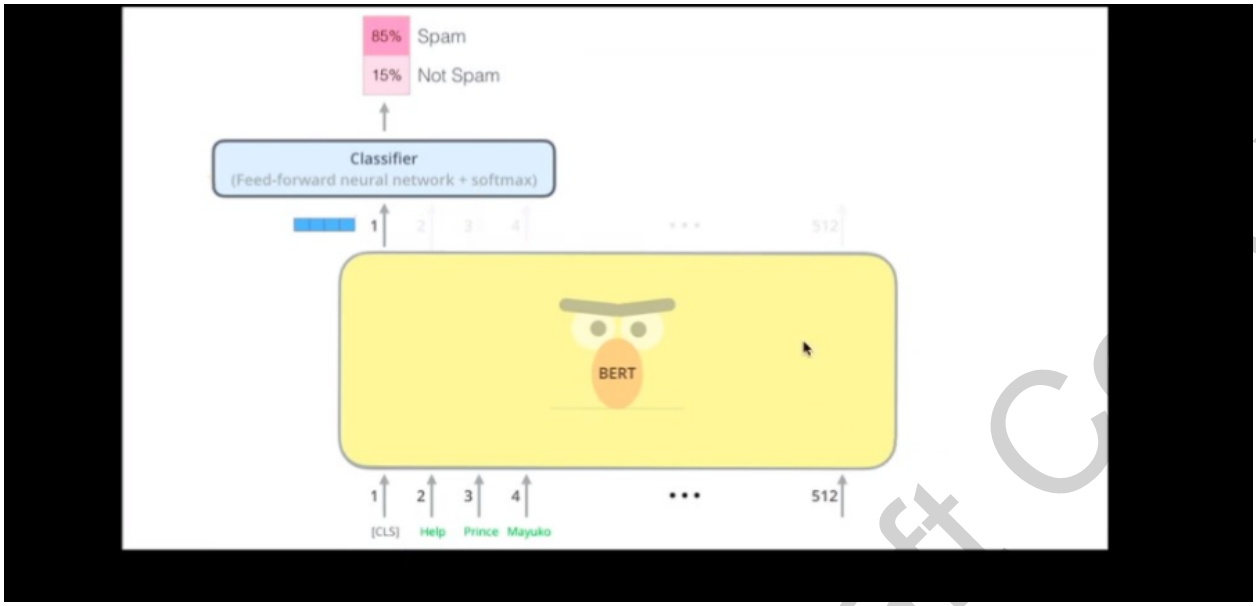
SQuAD1.1 is a competition where the task is based on Question and Answering. So, basically given a paragraph and a set of questions, we need to answer those questions based on the understanding of the paragraph. In this task, human performance was measured. The f1 score was 91.2. For the same task, the BERT model surprisingly produced a score of 93.160



Timestamp : 01:30:46

The BERT base has 12 encoder blocks and BERT large has 24 encoder blocks. In the BERT base, the feed forward neural network has 768 hidden units and in BERT large has 1024 hidden units in the feed forward neural network. BERT large has 16 attention heads and BERT base has 12 attention heads.

How to use a trained BERT model for a classification task ?

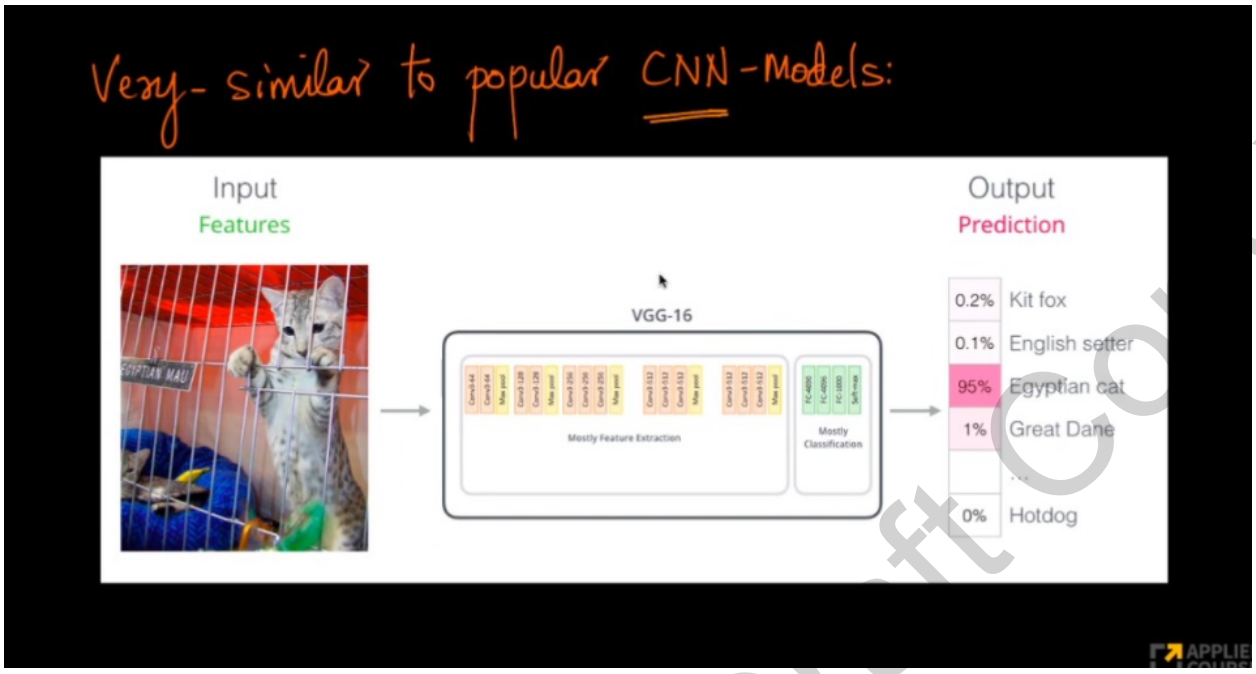


Timestamp : 01:35:46

Let’s say we want to detect whether an email is spam or not. So, an email is nothing but a sentence or bunch of words. Let’s say we have a trained BERT model with us.

Please refer to the above image.

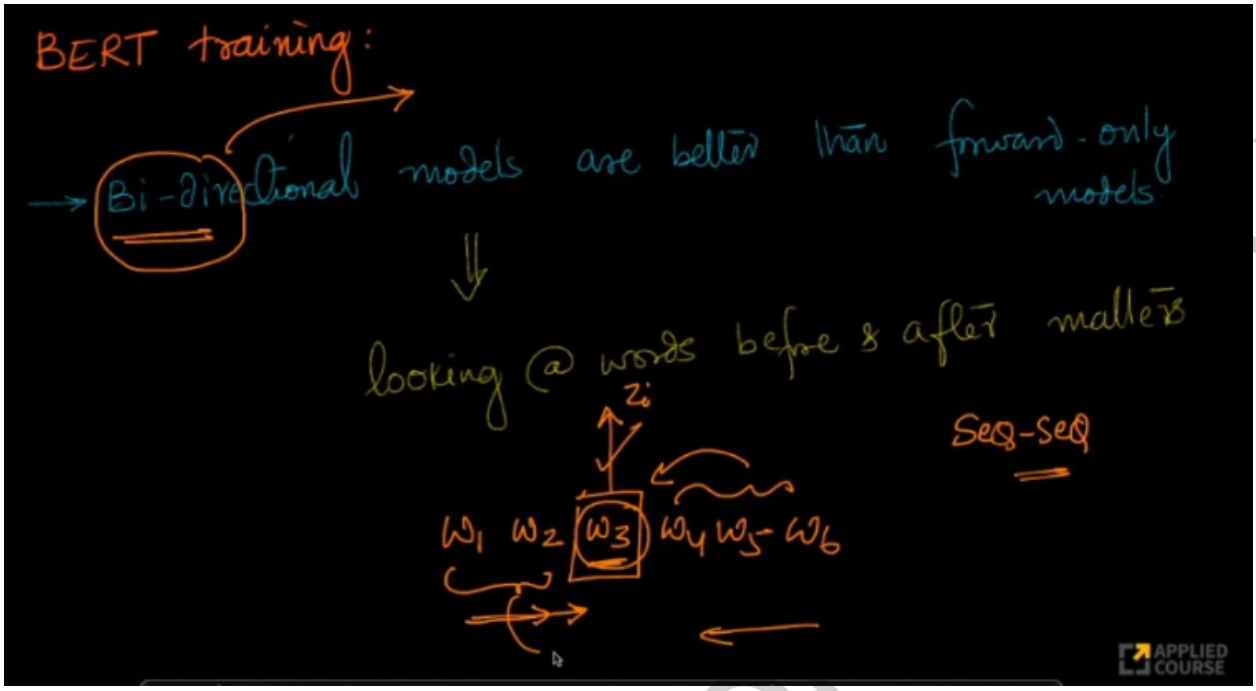
First, we mention <CLS> or give it as a token to specify we are performing a classification task. Then, we simply pass the sentence to it. The model will output a 768 dimensional vector for each of the input tokens. Then, we only take the 768 dimensional vector corresponding to the <CLS> token and then use a feed forward neural network to output the logits. The logits are then passed to the softmax function to output probabilities. A sentence can contain the utmost 512 words. This is because BERT was trained with utmost 512 words only.



Timestamp : 01:35:54

This is actually very similar to the task we perform in computer vision, which is transfer learning. We use a pretrained model and then replace the last layer with a feed forward neural network which outputs based on the number of classes we have. Then, we freeze all other layers except the last one. Then, we train only the final layer.

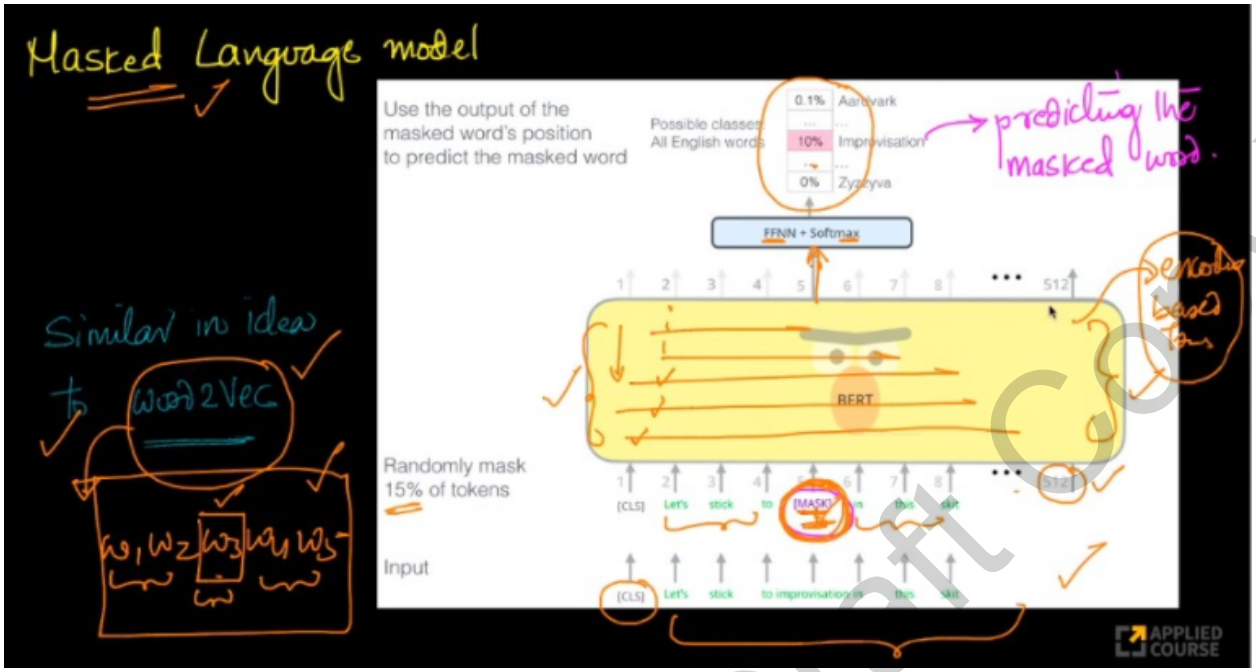
BERT training :



Timestamp : 01:37:47

Bi-directional models are better than forward-only models. In a traditional sequence to sequence model implemented with RNN, we only model one direction. This is not enough for some sentences. Because, for some sentences, a word may depend on the future words too. So, we need to model in both directions.

Masked Language Model :



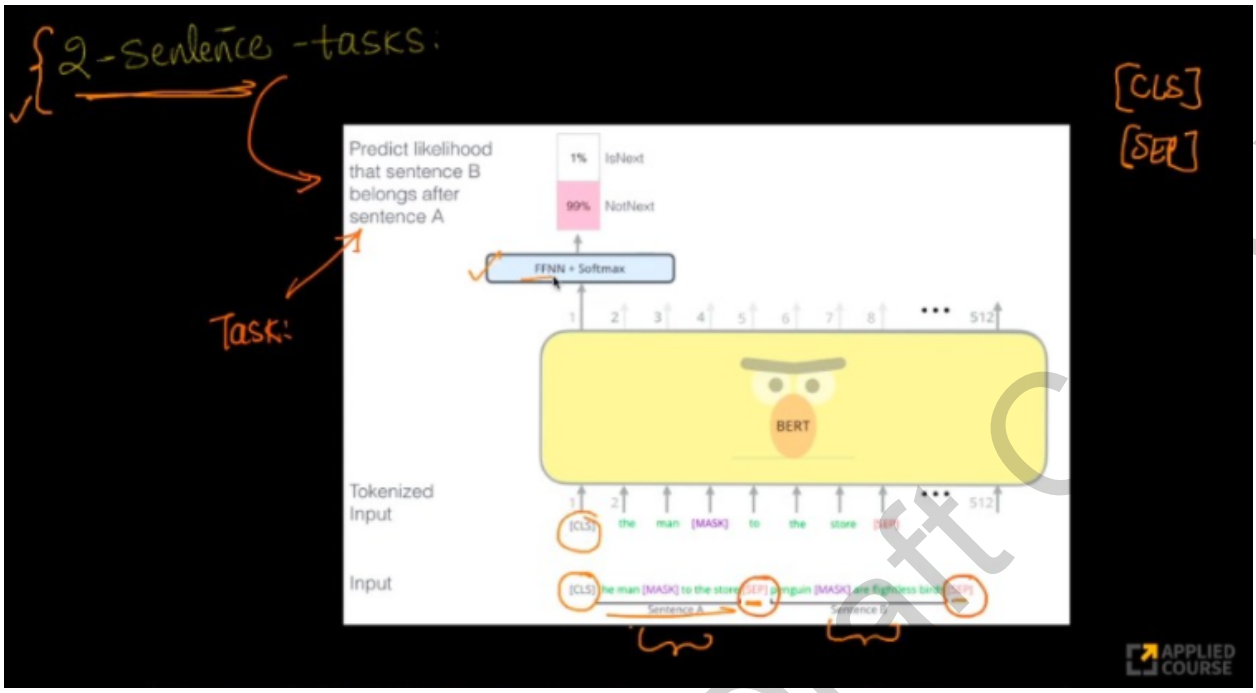
Timestamp : 01:42:05

This is similar to the training of Word2Vec. In word2vec, we basically predict the target word depending on its context words. The same basically applies here too.

As usual, we append the <CLS> token to denote that we are performing classification. Then, we randomly mask 15% of the input tokens i.e, we replace the words by the token <MASK> . Then, we pass it to the BERT model. Our aim is to predict what the <MASK> words contain. For this, we use a feed forward neural network and a softmax layer to output probabilities. By this way, we are learning the dependency of a target word with other words. Then , we train as usual. We already looked at how we train the encoder block in a transformer. The same goes here.

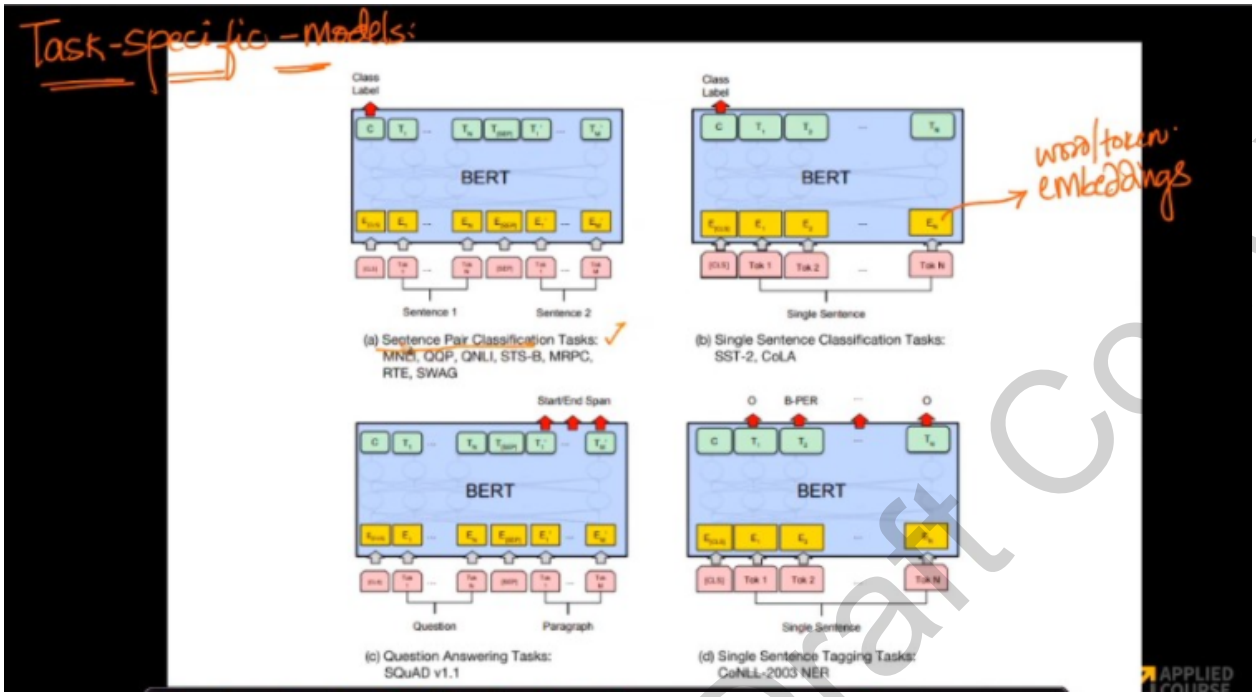
How to use BERT for 2-sentence tasks ?

Our aim is to predict the likelihood that sentence B belongs after sentence A.



Timestamp : 01:44:11

So, we add the <CLS> token at the start of the sentence as discussed earlier. Then, to separate the two sentences, we add the <SEP> token after the completion of the first sentence. Then, we randomly mask the inputs using the <MASK> token. Then we finally pass the vector corresponding to the <CLS> token to the feed forward neural network to get the logits vector. Then, we pass this to the softmax function to get probabilities. Using cross entropy loss, and backpropagation we can train it end to end.

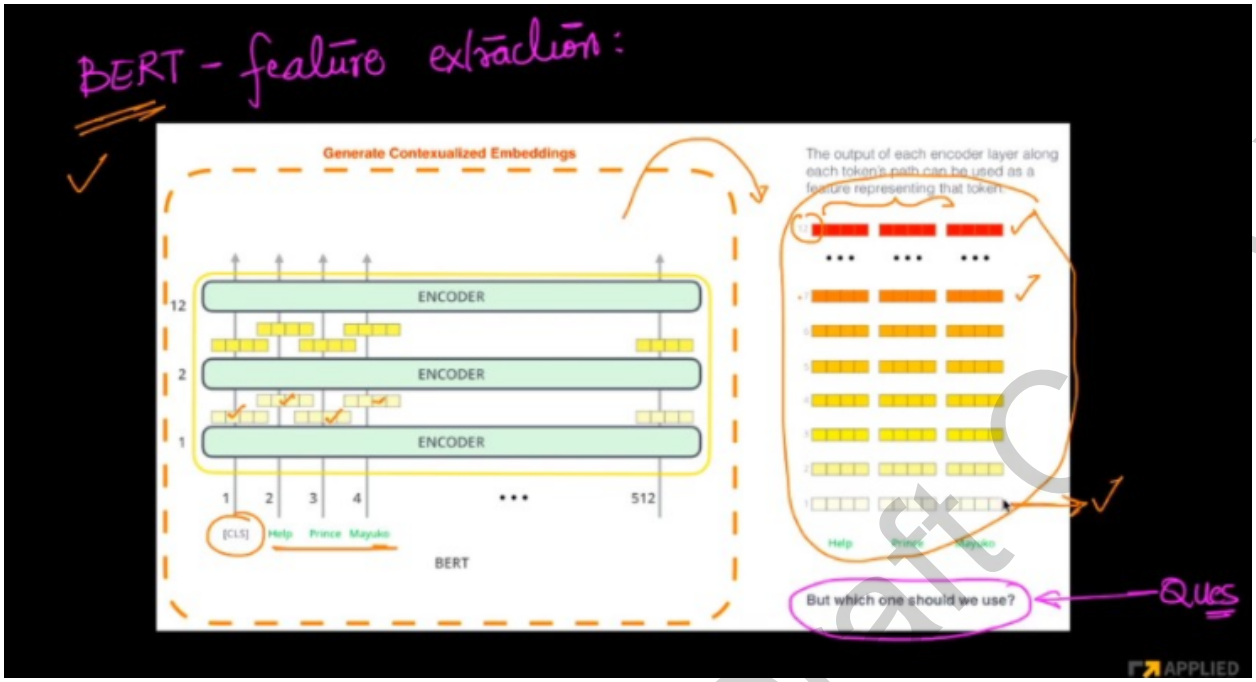


Timestamp : 01:44:46

In the above image, the same BERT model is used for multiple tasks.

1.) In the first case, we are doing sentence pair classification. Basically, given two sentences we need to predict whether they are similar or not or any other outcome we wish for. We already discussed it.
2.) In this we do a single sentence classification task which is trivial.
3.) In the question answering tasks, we have two inputs namely question and th paragraph. Our goal is to generate the answer for it. So, here we need to output a sequence of words with <START> and <END> tags. We take the outputs as mentioned in the above image.
4.) Here, we need to perform single sentence tagging i.e, parts of speech tagging. This is also simple. We have tags right ? So, we can build a softmax layer at the end to predict these tags. Basically tags are also classes. So, it's a classification problem.

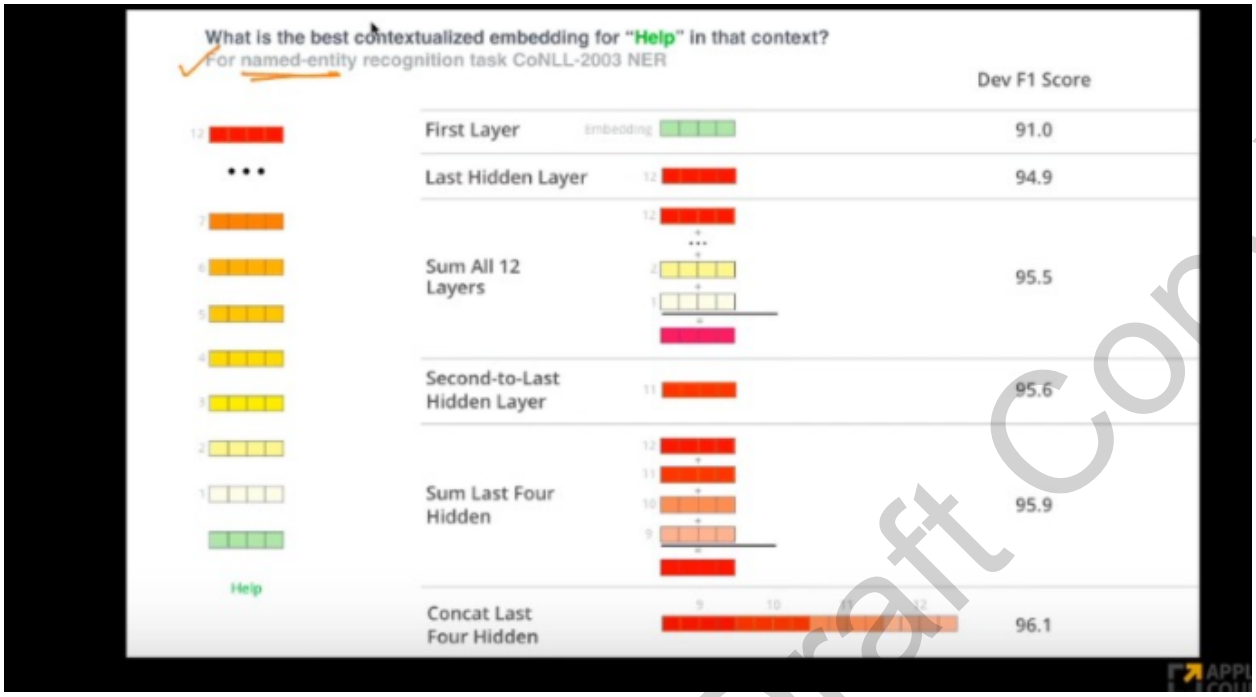
Using BERT as a feature extractor .



Timestamp : 01:47:45

Given a sentence, we want to get the features corresponding to it. For this ,we pass the sentence to the model. The model itself contains multiple encoder blocks. Each encoder block takes in an input and produces an output. We simply collect all the outputs and represent them as features.

But which one should we use among them for the given sentence ?



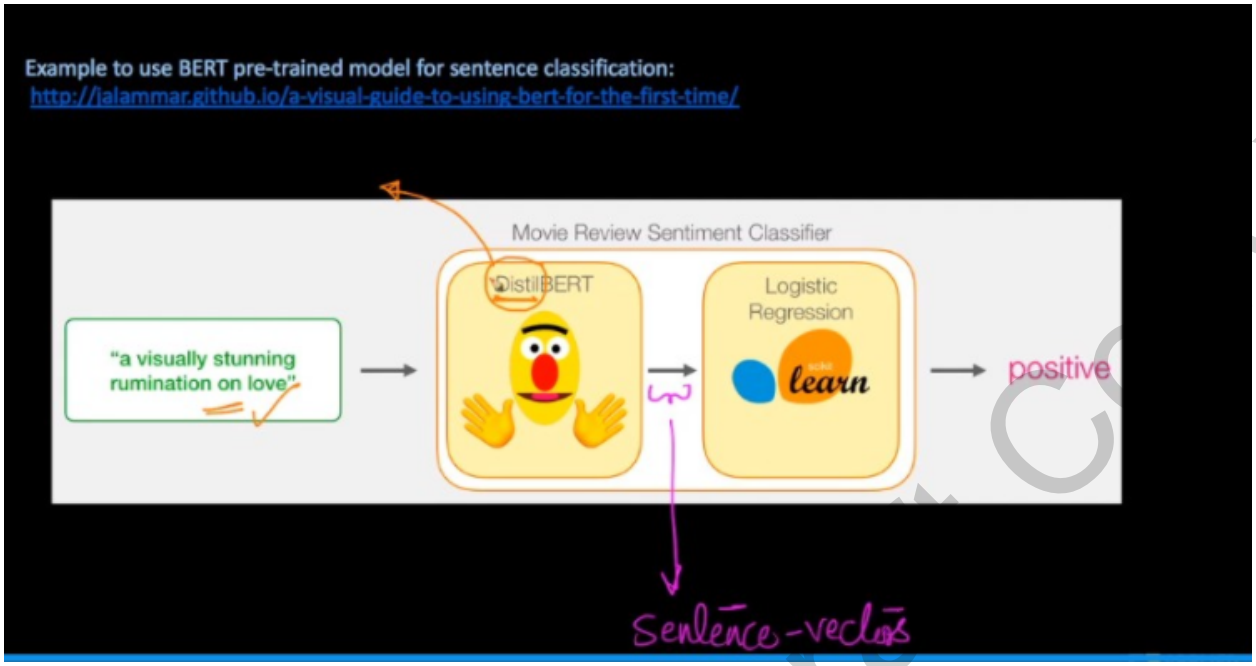
Timestamp : 01:48:27

So, there are multiple approaches for it. Please refer to the above image.

What is the best contextualized embedding for the word “Help” in that context ?

Each approach is discussed with the corresponding F1 Score.

We can see that if we concat the last four hidden layers, it gives the best score.



Timestamp : 01:50:07

We can use the DistilBERT model to implement this. Then, we can use the logistic regression or any classification algorithm to use the logits returned by the BERT model for outputting the probabilities.