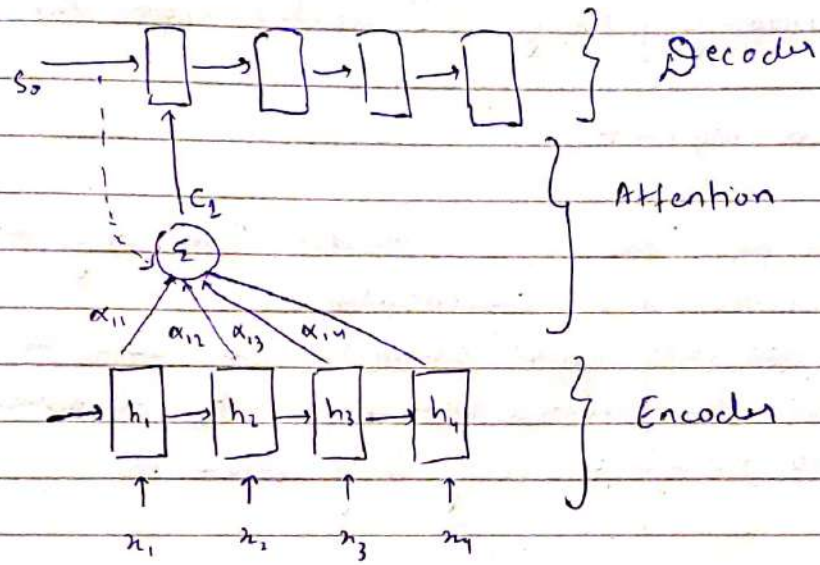
Date: / /

TRANSFORMERS

* Introduction to transformers

→ Limitations of Attention based RNNs



As we already know following are some formulas

- $c_t = \sum_{i=1}^{n} \alpha_{ti} h_i$ context vector for output y_t

n, number of words
t : time step of for decoder

- $\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{i=1}^{n} \exp(e_{ti})} = \text{softmax}(e_{ti})$

- $e_{ti} = V_{att} \tanh(U_{att} s_{t-1} + W_{att} h_i)$

(Note - notations are a little changed from earlier please manage)

Date: / /

Now, what we currently do is as follows

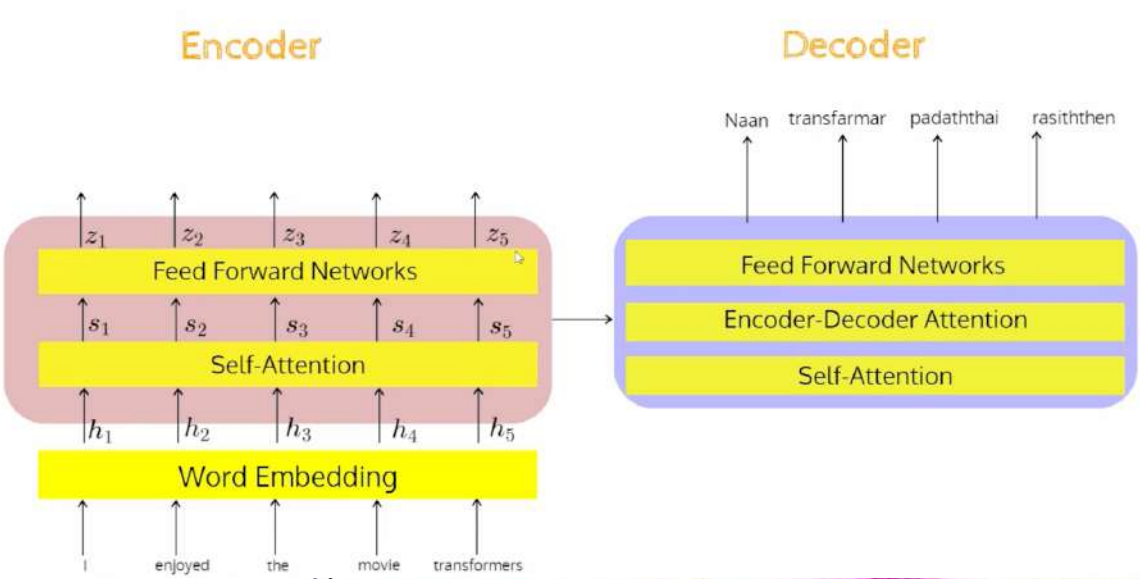
- Compute all the h_i's sequentially by iterating through encoder
- Once we have h_i's we can compute all ~~c_t's parallely~~ $\alpha_{t,i}$'s with each c_t parallely
- Then sequentially get y_t's (outputs) from the decoder

What can be improved,

- Initially we have all x_i's (input) then why do we want to compute h_i's sequentially
- So for a particular time stamp in the decoder we already have the previous time step in the decoder, so we want to parallelize every thing below it.

* Attention is all you Need (Encoder)

→ Following the above mentioned idea, we want parallelism in the encoder



Date: / /

Initially, we had the following formula for attention

- $f_{att} = V_{att} \tanh(U_{att} S_{t-1} + W_{att} h_i)$

  S_{t-1} → decoder time step; h_i → encoder timestep

Now, what we have done is

- $f_{att} = f_{att}(h_i, h_j)$

  h_i → encoder timestep; h_j → encoder timestep

  since both are encoder for attention computation this is called <u>Self Attention</u>

Note → we have also removed the reccurrent connections that were previously there.

Because of the above mentioned changes parallelism is not only possible per time step but also accross multiple timestep.

eg- for $h_1 \rightarrow \alpha_{11}, \alpha_{12}, \ldots \alpha_{1n}$

$h_2 \rightarrow \alpha_{21}, \alpha_{22}, \ldots \alpha_{2n}$

⋮

$h_n \rightarrow \alpha_{n1}, \alpha_{n2}, \ldots \alpha_{nn}$

we are able to compute because it no longer depends on decoder timestep

we are able to compute because all encoder words embeddings are available from get go

Date: / /

* Self Attention

→ Previously, we had

- $f_{att} = v_{att} \tanh (U_{att} s_{t-1} + W_{att} h_i)$

U_{att}: $d \times d$, s_{t-1}: $d \times 1$, W_{att}: $d \times d$, h_i: $d \times 1$

v_{att}: $d \times 1$, result: $d \times 1$

1×1

Here we have 3 vectors (v_{att}, s_{t-1}, h_i)

Following the same idea,

- $f_{att} = f_{att} (h_i , h_j)$

Step 1: lets say we have 3 matrix (W_Q , W_K , W_V)

W_Q: Query matrix ($d \times d$)

W_K: Key matrix ($d \times d$)

W_V: Value matrix ($d \times d$)

for h_1,

h_1^T [———] [ ] = [———] q_1^T

W_Q

Date: / /

h_1^T [———] $\begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$ = [———] k_1^T

W_K

[———] h_2^T $\begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$ = [———] V_1^T

W_V

∴ for h_1 we get (q_1, k_1, v_1) by matrix multiplication

similarly, for all h_i we get (q_i, k_i, v_i) using same matrix

Step 2

Now for each query we want the importance / weight of each key

[———] q_1^T $\begin{bmatrix} | & | & | \\ | & | & | \\ | & | & | \end{bmatrix}$ = [———] $q_1^T K$

k_1 k_2 ... k_n

similarly you can do it for all q_i's by keeping K matrix the same

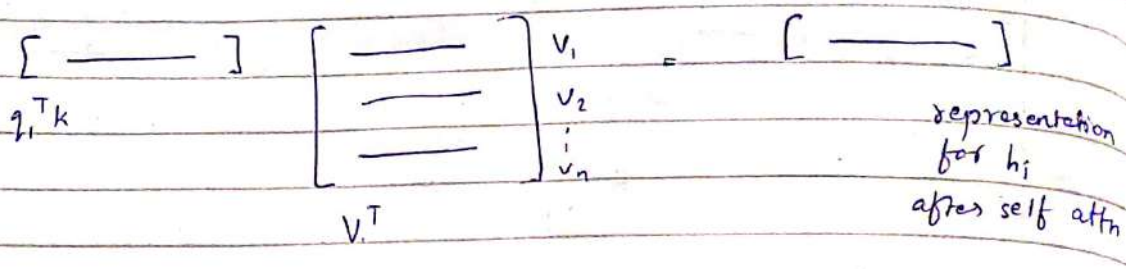
Step 3

Do a softmax on $q_1^T K$ to make probabilistic distribution

Step 4

Now, V_i's are representations of h_i's and we want to pick from V_i's based on the probability distribution we obtained in above step

Date: / /



Lets formulate the above for h_1,

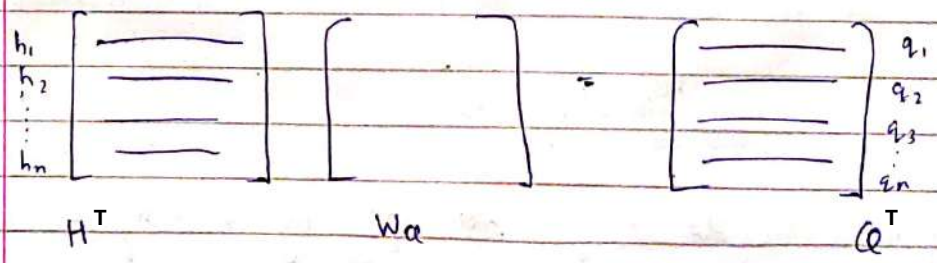
$$e_1 = [q_1 \cdot k_1, \; q_1 \cdot k_2, \; \ldots \; q_1 k_n]$$

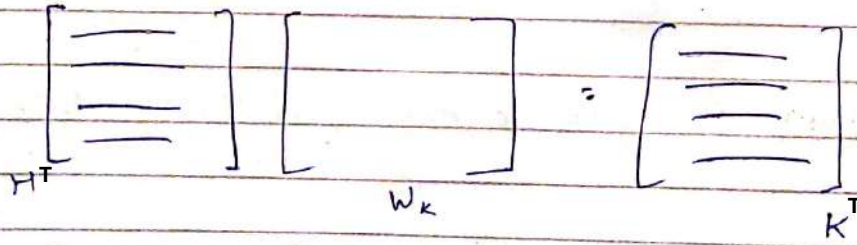
$$\alpha_{1j} = \text{softmax}(e_{1j})$$

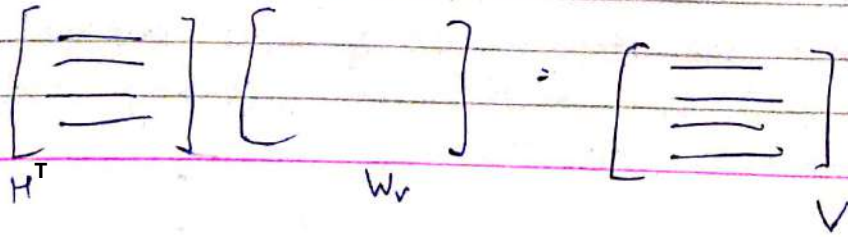
$$z_1 = \sum_{j=1}^{n} \alpha_{1j} \cdot v_j$$

→ Now, we also need to prove that we can compute all $(z_1 \ldots z_n)$ for $(h_1 \ldots h_n)$ in one go

Step 1



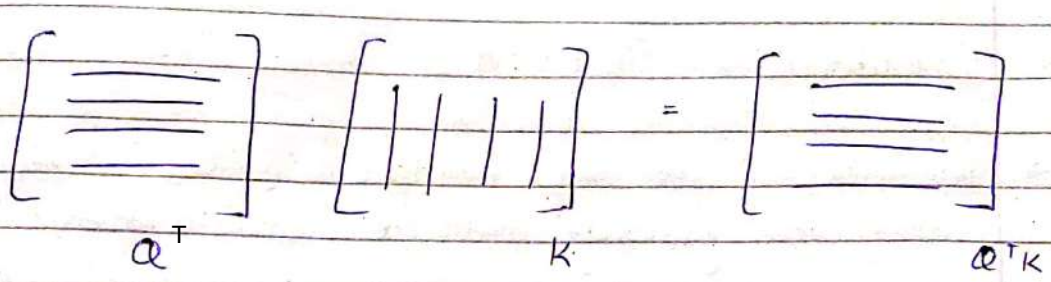




Ishan Modi

Date: / /

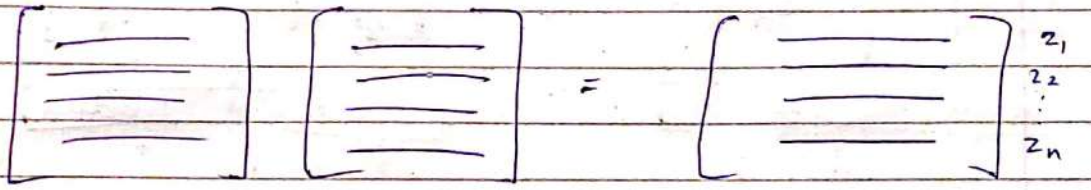
Step 2



Step 3

$$\text{Softmax}\left(\frac{Q^T K}{\sqrt{d_k}}\right) = \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

Step 4

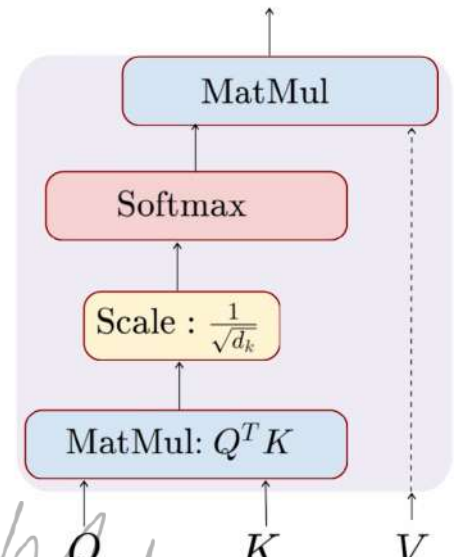


So final formulation of self Attn^T

$$= \text{Softmax}\left(\frac{Q^T K}{\sqrt{d_k}}\right) V^T$$

T T

→ ($\sqrt{d_k}$ is the scaling factor generally equal to dimension of key)

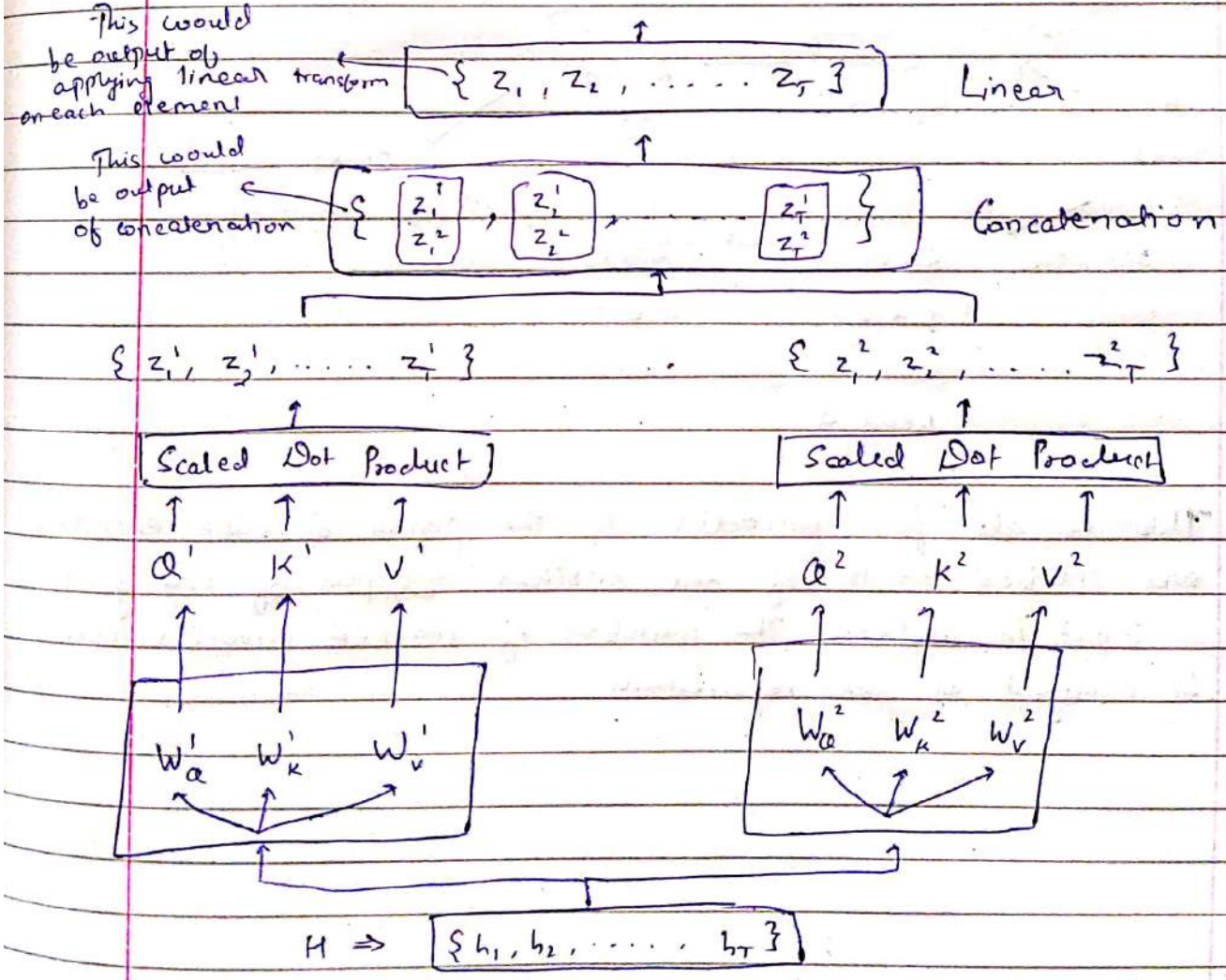


Date: / /

* Multi-Headed Attention

→ The idea is to use multiple (W_Q, W_K, W_V) matrices and generate multiple representations

→ Motivation is the way we use multiple filters to create ~~feature~~ multiple feature maps, where each feature map is created using one filter. At the end we concat all feature maps for further processing.

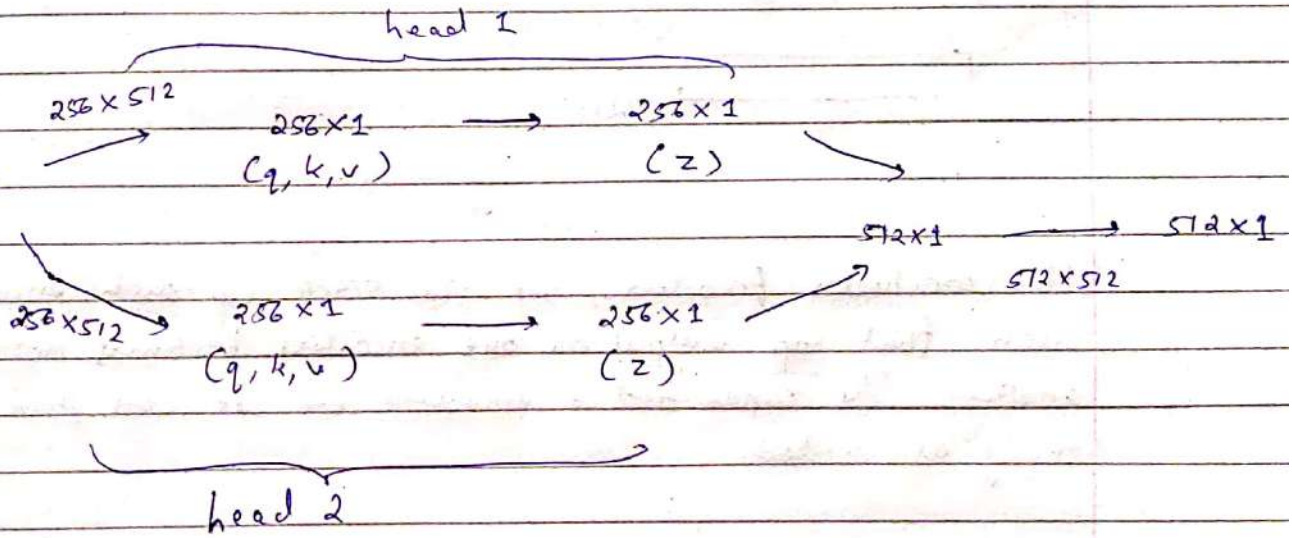


Here we have only shown 2 heads, you can have as many heads as needed.

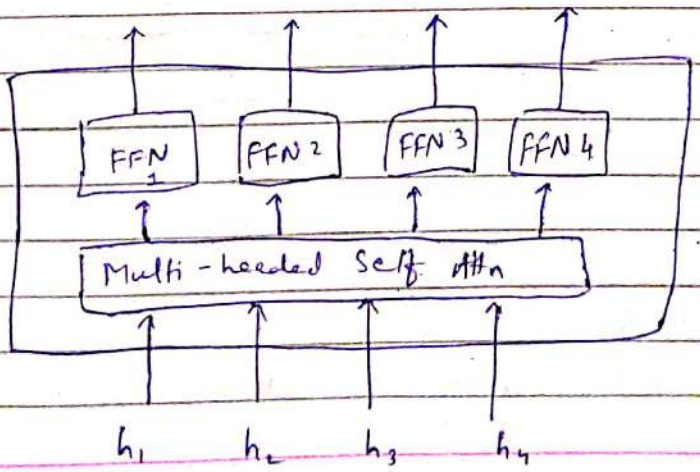
Date: / /

In practice, we adjust the dimensions of (W_q, W_k, W_v) across all heads such that query, key & value vectors are less in dimension than h's (input)

Thus each head will generate z's which will be less than h's in dimension. But when we concatenate the z's across all heads the dimension becomes on par or even more than h's which is then scaled by linear layer



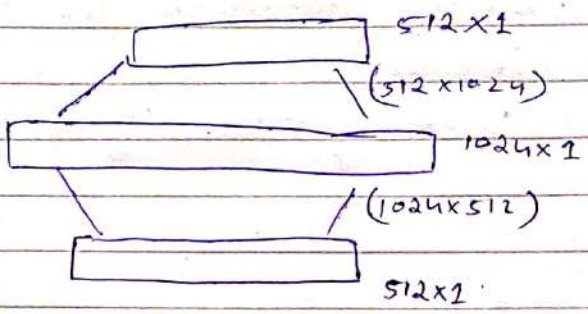
* Feed forward Network (FFN)



Date: / /

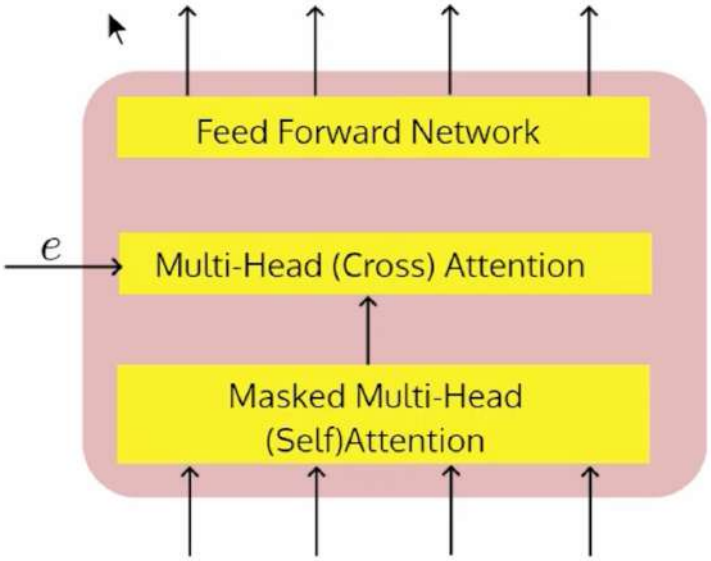
Unlike linear layer block in multiheaded Attn, which had only one projection to the required dimension, FFNs contain hidden layers.

FFN



This concludes Encoders, we can stack up such encoders such that ~~inp~~ output on one encoders becomes input to another. The paper had 6 encoders but we can pick as many as needed.

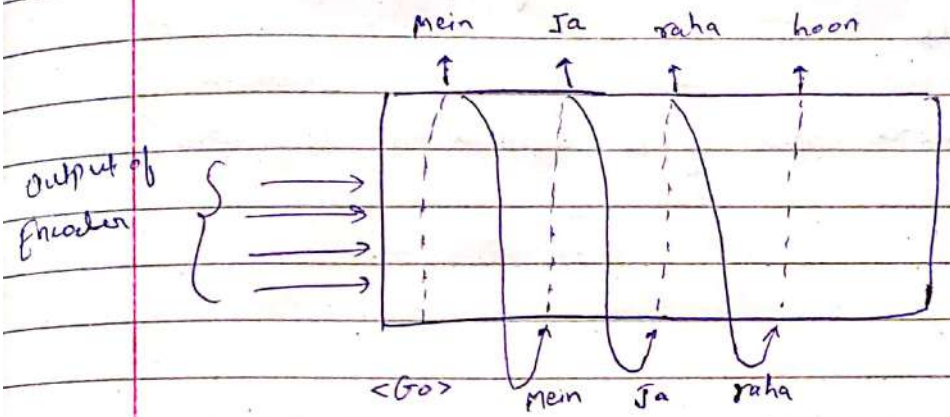
* Decoder



Date: / /

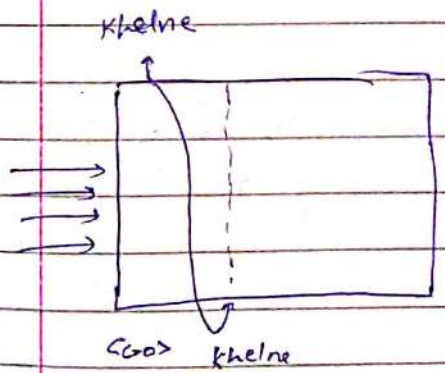
* Teacher Forcing

→ As we can see, decoder are [auto-regressive], meaning output at time step one is given as an input along with all the previous inputs at timestep 2 and so on..



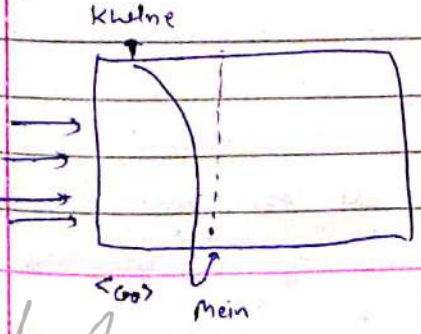
original sentence - I am going

Now, what if the word at the first timestamp is predicted wrong ?



This could mess up all the subsequent predictions.

A solution to this would be



Though it predicted "Khelne" wrongly we give the correct thing "Mein" in the input. We do this for the first few epochs forcing the correct word as input & later stop forcing once network has learnt relevant amount

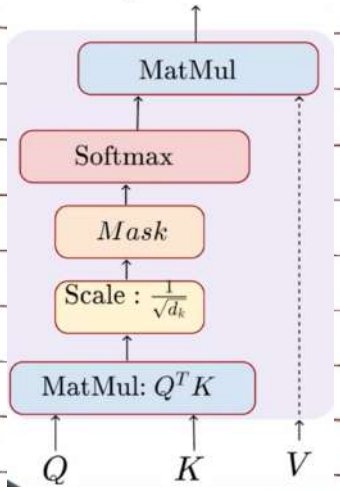
Date: / /

* Masked - Multi-Head Self Attention

→ As we know, we cannot pass all the words at a given timestamp to the decoder

At a given timestamp we can only pass all the previously predicted words.

Following is where masking is applied in self-attn block



How ~~does~~ does masking work,

$$= \begin{bmatrix} q_1k_1 & q_1k_2 & q_1k_3 \\ q_2k_1 & q_2k_2 & q_2k_3 \\ q_3k_1 & q_3k_2 & q_3k_3 \end{bmatrix} + \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & -\infty \end{bmatrix} \Bigg\} \text{Masking}$$

$$= \begin{bmatrix} q_1k_1 & -\infty & -\infty \\ q_2k_1 & q_2k_2 & -\infty \\ q_3k_1 & q_3k_2 & q_3k_3 \end{bmatrix}$$

$$\text{Softmax}\left(\begin{bmatrix} q_1k_1 & -\infty & -\infty \\ q_2k_1 & q_2k_2 & -\infty \\ q_3k_1 & q_3k_2 & q_3k_3 \end{bmatrix}\right) = \begin{bmatrix} S_{11} & 0 & 0 \\ S_{21} & S_{22} & 0 \\ S_{31} & S_{32} & S_{33} \end{bmatrix} \Bigg\} \text{Softmax}$$

Basically we are adding $(-\infty)$ to all the next words in the sequence so that output after softmax becomes 0 so that no importance is given to next word in sequence

Date: / /

Note → Masking also happens in encoder,

token length → dimension of vector representing each word
Sequence length → total no of tokens allowed.

So for an encoder if the number of tokens are less than sequence length, padding is done

These new tokens generated using padding should be ignored at the time of self attention, this is done using masking

* Cross Attention

→ As we know from earlier, query is the word for which we are trying to get representation. So the query part in cross attention comes from decoder & key, value part comes from encoder

→ Following is formula for masking - multi-head attention

- Decoder representation = $\text{Softmax}\left(\frac{QK^T}{\sqrt{d}} + M\right)V = (D_i)$ — & Encoder representation

- Encoder representation = output from encoder. = (E_i)

D_i (T×d) → $D_i \cdot W_Q$ (T×d · d×d) → Q (T×d)

E_i (T×d) → $E_i \cdot W_K$ (T×d · d×d) → K (T×d)

E_i (T×d) → $E_i \cdot W_V$ (T×d · d×d) → V (T×d)

After this steps similar to self attention are followed

Note :- D_i & E_i can also have different dimension

eg D_i = $T_1 \times d$

E_i = $T_2 \times d$

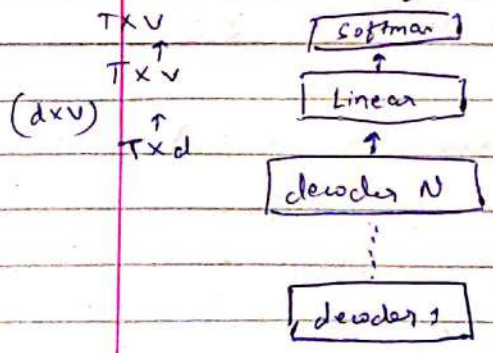
but at the end of decoder we need output similar to input ie $T_1 \times d$. If we do all calculations with $D_i \to T_1 \times d$ & $E_i \to T_2 \times d$, the decoder block will give us $T_1 \times d$ as the output because

$$T_1 \times T_2 \cdot T_2 \times d \to T_1 \times d$$
$$(Q K^T) \cdot V \to (\text{output})$$

* Feed Forward Network (Similar to Encoder, Masked attention output is feed to FFN)
* Linear & Softmax

→ We can stack up as many decoders on top one another, original paper had 6.

→ At the end of final decoder we have linear layer + softmax



So the linear layer has $d \times V$ weights & it projects to V dimension & softmax gives probability distribution

$$T \times d \cdot d \times V \to T \times V \xrightarrow{\text{softmax}} T \times V$$

decoder output linear output probability distribution in each row

Note - till now we have discussed attention with the following formula -

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

Here following are the Q, K, V matrices

$$Q = \begin{bmatrix} | & | & | & | \\ q_1 & q_2 & q_3 & \cdots & q_n \end{bmatrix}$$

$$K = \begin{bmatrix} | & | & | & | \\ k_1 & k_2 & k_3 & \cdots & k_n \end{bmatrix} \qquad V = \begin{bmatrix} | & | & | & | \\ v_1 & v_2 & v_3 & \cdots & v_n \end{bmatrix}$$

$$A = \begin{bmatrix} | & | & | & | \\ a_1 & a_2 & a_3 & \cdots & a_n \end{bmatrix}$$

But if we the above formula can also be written as,

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

this is possible with the following Q, K, V matrices

$$Q = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_n \end{bmatrix}$$

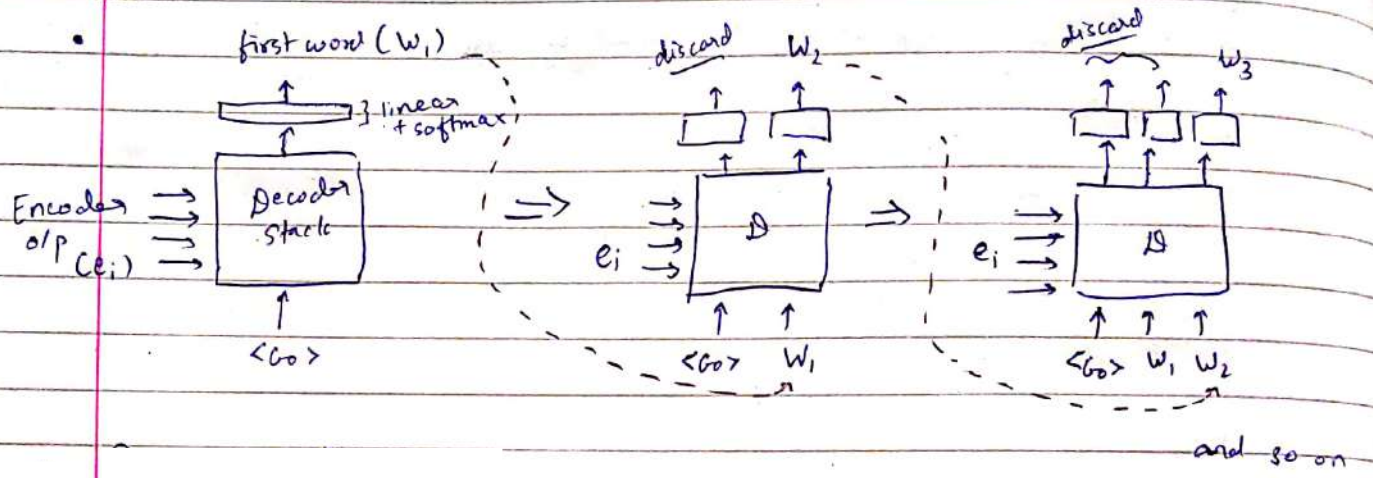
$$K = \begin{bmatrix} k_0 \\ k_2 \\ k_3 \\ \vdots \\ k_n \end{bmatrix}, \quad V = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix}$$

$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix}$$

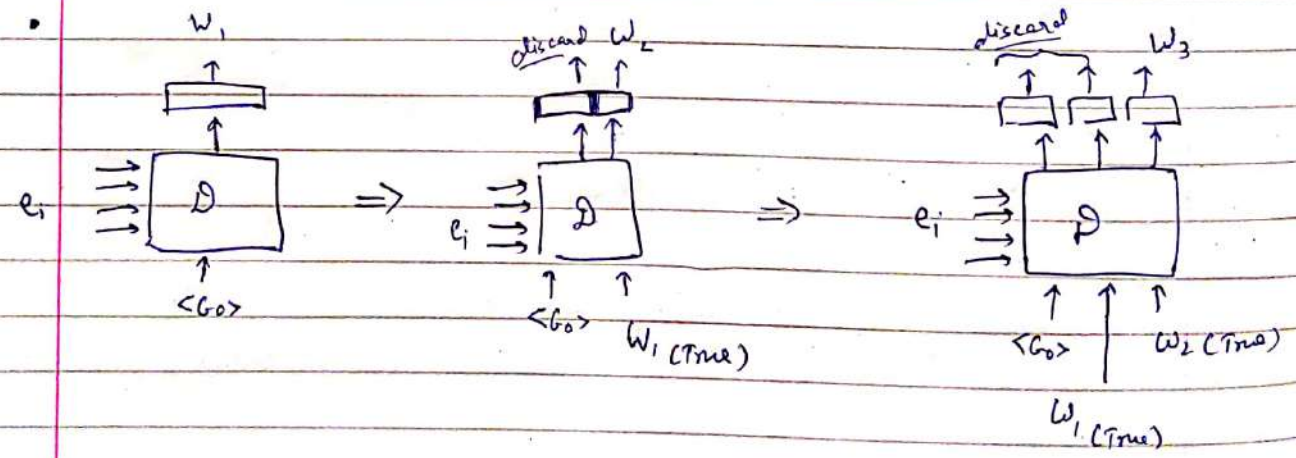
Date: / /

Note - Lets discuss how decoder actually work.

CASE - 1 Training phase without teacher forcing



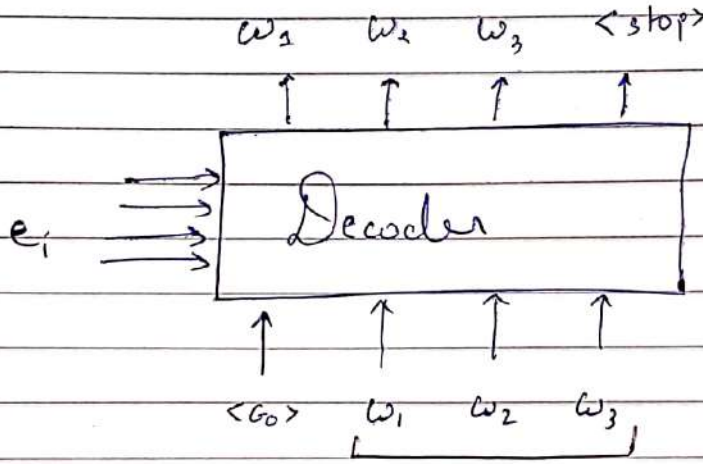
CASE - 2 Training phase with teacher forcing



Here we are not using the predicted labels/tokens at all in next iteration

Since we are not using the predicted token at each iteration & we have the entire input during training, we can run the decoder in parallel during training under the teacher forcing scheme

Though teacher forcing is faster, it has to be used along with normal training, so model learns to deal with mistakes at inference.



↳ These come from the training data & are same irrespective of what is predicted

- Other points to ~~remain~~ remember

- Self attention matrix → $(T+1) \times (T+1)$ because sequence starts with <Go> & ends with <END>
- For every word entire decoder has to be iterated
- Masking is required in subsequent decoder blocks in a decoder stack irrespective of teacher forcing.

* Positional Encodings

→ Motivation - Currently, up until now we don't have any positional information in the words. If we switch words in input given to encoder, it won't affect output.

→ Following are our initial thoughts on how to get position into token

- w_1 + $[1, 1, \dots]$ w_2 + $[2, 2, 2 \dots]$ w_3 + $[3, 3 \dots]$ and so on

  Plus is not feasible because if sequence length is 256 than you will end up adding 256 to last token & 256 is huge when compared to random initialization

- w_1 + $[1, 0, \dots]$ w_2 + $[0, 1, \dots]$ w_3 + $[0, 0, 1, \dots]$ and so on

  use one-hot encodings

The notion of distance is lost, eg $dist(w_1, w_2) = \sqrt{2}$ distance is calculated between one-hot encoding of w_1 & w_2

Date: / /

now, the dist(w_1, w_3) = $\sqrt{2}$, essentially distance between w_1 & w_3 should be more.

- w_1 + [....] α_1 ; w_2 + [...] α_2 ; w_3 + [...] α_3 and so on } positional encoding are learnable parameters

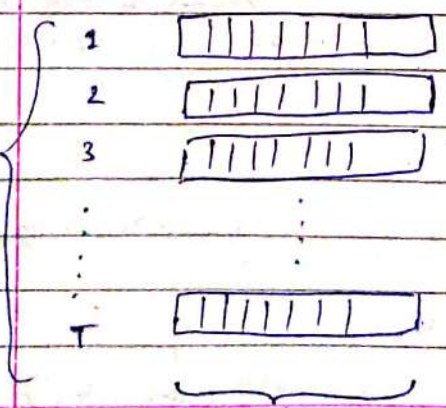
This also has an issue, if we have encountered only max sequence length of 40 during training, then only 40 positional encodings will be learned. At inference if a sentence has more than 40 tokens we won't have any learned ~~each~~ encodings

* Sinusoidal Embedding

→

$$PE_{(j,i)} = \begin{cases} \sin\left(\frac{j}{10000^{2i/d_{model}}}\right), & i = 0, 2, \ldots, d_{model} \\ \cos\left(\frac{j}{10000^{2i/d_{model}}}\right), & i = 1, 3, \ldots, d_{model} \end{cases}$$

Here,



j = position of token in sequence

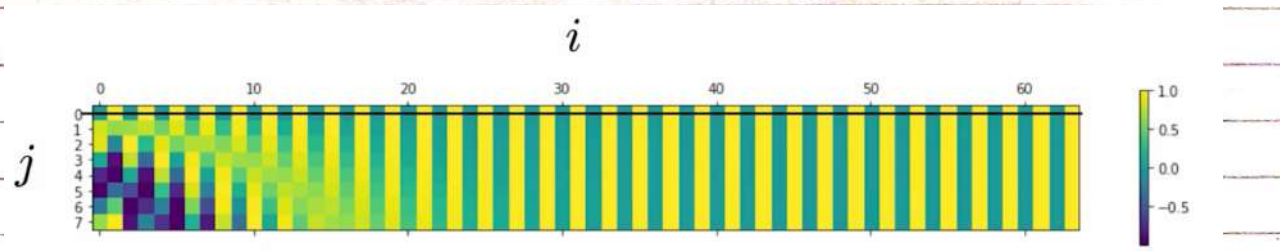
i = value inside token embedding

d_{model} = length of token embedding

Date: / /

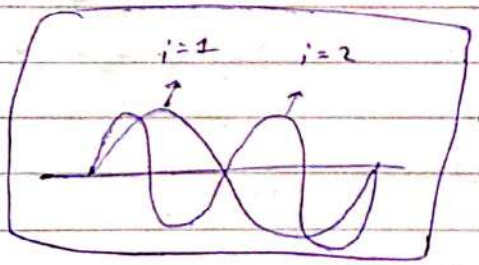
Note - If "i" is ~~odd~~ even we use sin function & if it is odd we use cosine function.

→ Following is a diagram



→ In the following function,

$$\sin\left(\frac{j}{10000^{2i/d_{model}}}\right)$$, for a fixed j as the value of i increases the frequency decreases



→ graph representation of frequency

→ By using sinusoidal function,

- We are creating a dependence between j & i, every value at i positions in the embedding depend on j.
- Notion of distance is also there. Eg- 32 word will be close to 31 st word & 33 rd word & distance increases as we move in either direction

Date: / /

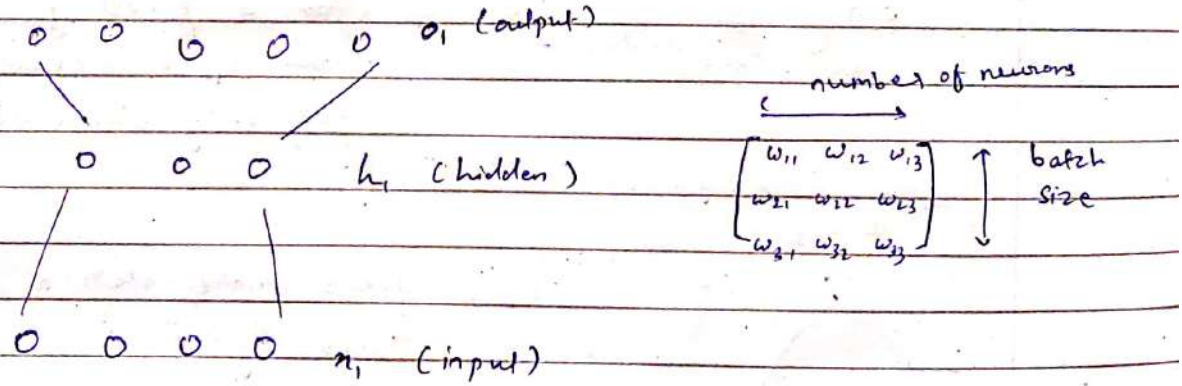
* Residual Connection & Layer Normalization

→ Residual Connection

Since there is a stack of encoder & a stack of decoder in transformer, and individually these encoder & decoder also contain attention layers, there is a chance of vanishing gradient. To deal with that we use residual connections

→ Layer Normalization

- It is very similar to Batch Norm



let consider the above Neural Network.

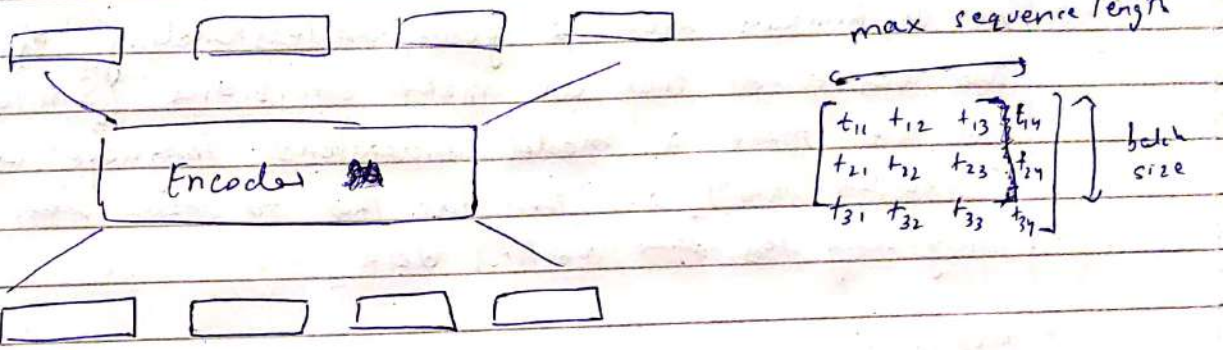
- For Batch normalization,

all the values within the batch are normalized, so each column will be normalized individually (normalization takes place accross column). ie - (w_{11}, w_{21}, w_{31}) are normalized

- For Layer normalization,

all the values at the layer are normalized, so the each row will be normalized individually. (normalization happens accross rows) ie- (ω_{11}, ω_{12}, ω_{13} are normalized)

Now, how does layernorm work in transformers



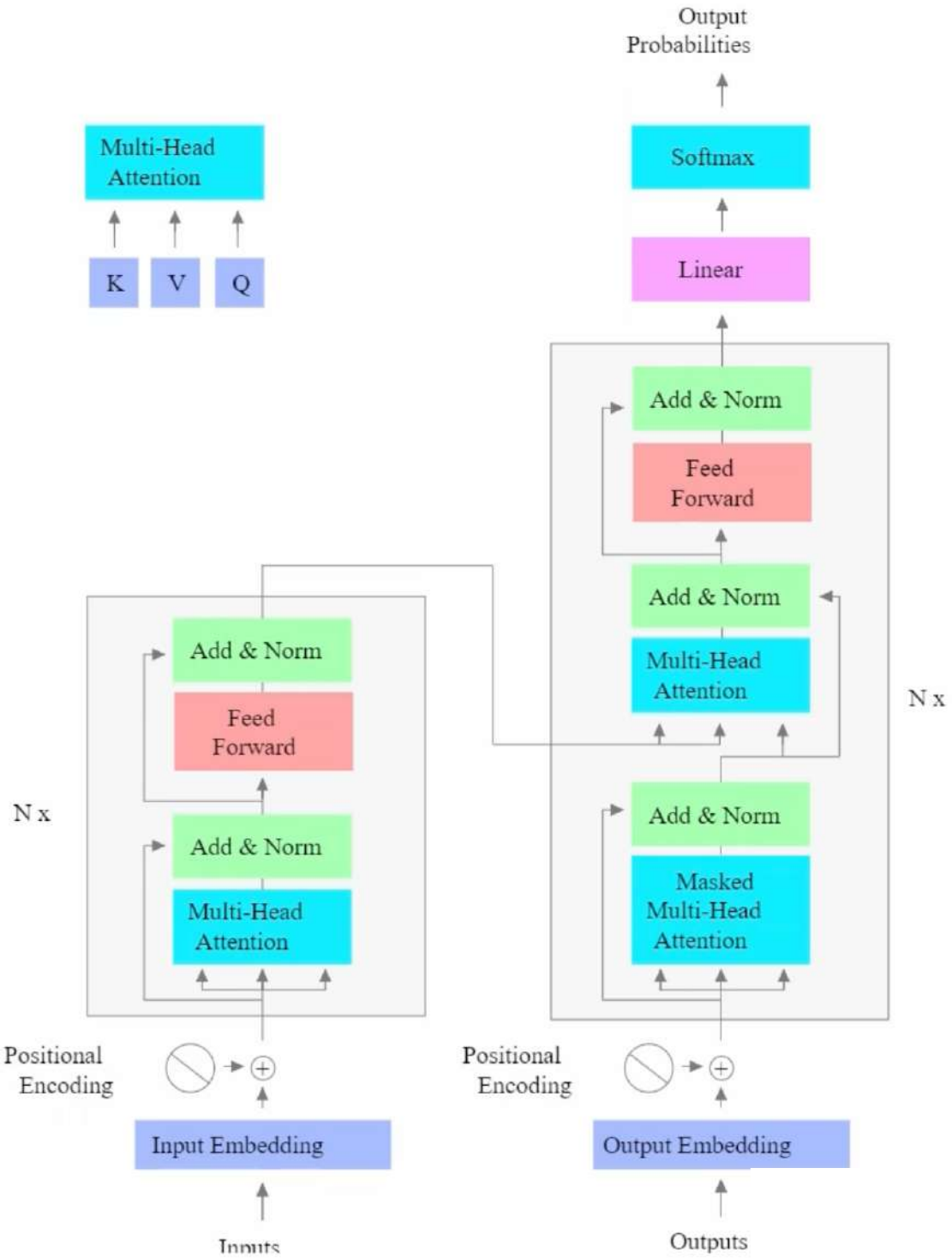
So layernorm is calculated accross each row, also each token is dmodel (dimensional). Thus the layer norm will be calculated for the following

$$(t_{111} + t_{112} + \ldots + t_{11d}) + (t_{121} + t_{122} + t_{123} + \ldots t_{12d})$$

$$+ (t_{131} + t_{132} + \ldots + t_{13d}) + (t_{141} + t_{142} + \ldots + t_{14d})$$

Similar for other rows, as well.

Transformer Architecture



Date: / /

Decoder-Only-Models

* Language Modelling

→ Labelled data is always limited, what if we can use un-labelled data for language modelling

→ We as humans have a good understanding of language and based on that we make decisions. (Similarly can we make a model understand language using raw-text (unlabelled data)) and [then use that to make ~~desi~~ decisions using very less ~~data~~ labelled data] -

pretraining → Supervised - Finetuning

→ What is language modelling ?

let say, we have $\{x_1, x_2, \ldots x_n\} \in V$, we is the vocabulary & x_i's are all the words

We can construct a lot of different sentences using these words. Now when we are reading/refering some text the probability of some sentences occurring is higher because they are more common than the other sentences

The probability of a sentences appearing can be given as follows,

$$f : (X_1, X_2, X_3, \ldots X_n) \rightarrow [0, 1]$$

function — some probabilistic function suggesting words will occur in the fixed sequence

Page No.
Date: / /

Now,

$$f = p(X_1) \cdot p(X_2 \mid X_1) \cdot p(X_3 \mid X_1, X_2) \cdot p(X_T \mid X_1, X_2, \ldots X_{T-1})$$

probability distribution over vocab that gives first word (marginal probability)

probability distribution over the vocab that gives second word given previous (first) word (conditional probability)

All other are also conditional probability

$$= p(X_1) \cdot \prod_{i=2}^{T} p(X_i \mid X_1, \ldots, X_{i-1})$$

→ How do we link pre-training, with the above formula of language modeling?

So, lets pick a task of predicting next token in a sequence

Here you don't need labelled data, from raw-text available you can give the model T tokens and tell it to predict $T+1^{th}$ token.

Also, each term in the above formula, can be calculated by the previously seen transformer architecture.

Following diagram is showing, the equation & its relation with transformers,

Date: / /

Causal Language Modelling (CLM)

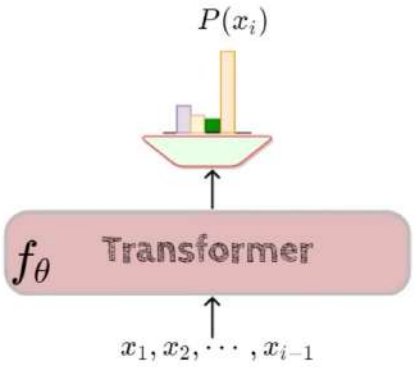
$$P(x_1, x_2, \cdots, x_T) = \prod_{i=1}^{T} P(x_i | x_1, \cdots, x_{i-1})$$

$$= P(x_1)P(x_2|x_1)P(x_3|x_2, x_1) \cdots P(x_T|x_{T-1}, \cdots, x_1)$$

We are looking for f_θ such that

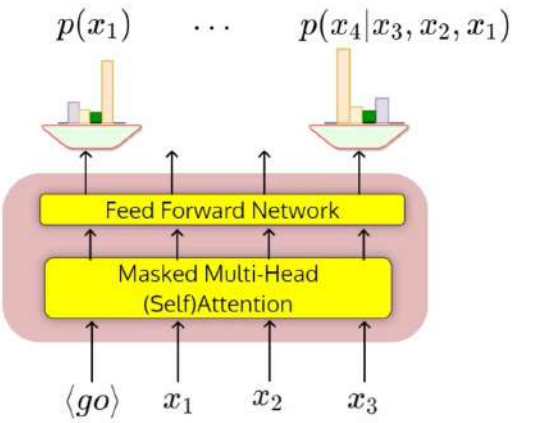
$$P(x_i|x_1, \cdots, x_{i-1}) = f_\theta(x_i|x_1, \cdots, x_{i-1})$$

Can f_θ be a transformer?



The transformer architecture can be,
- Encoder Only
- Decoder Only
- Encoder - Decoder

* Decoder Only Models



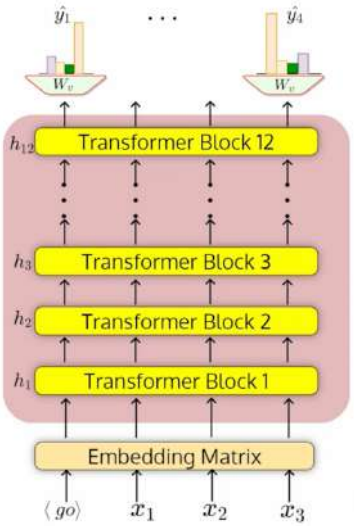
This is a basic diagram for decoder-only model and here the joint probability for the sequence is

$$f = p(x_1)\ p(x_2|x_1)\ p(x_3|x_1, x_2)\ p(x_4|x_1, x_2, x_3)$$

Date: / /

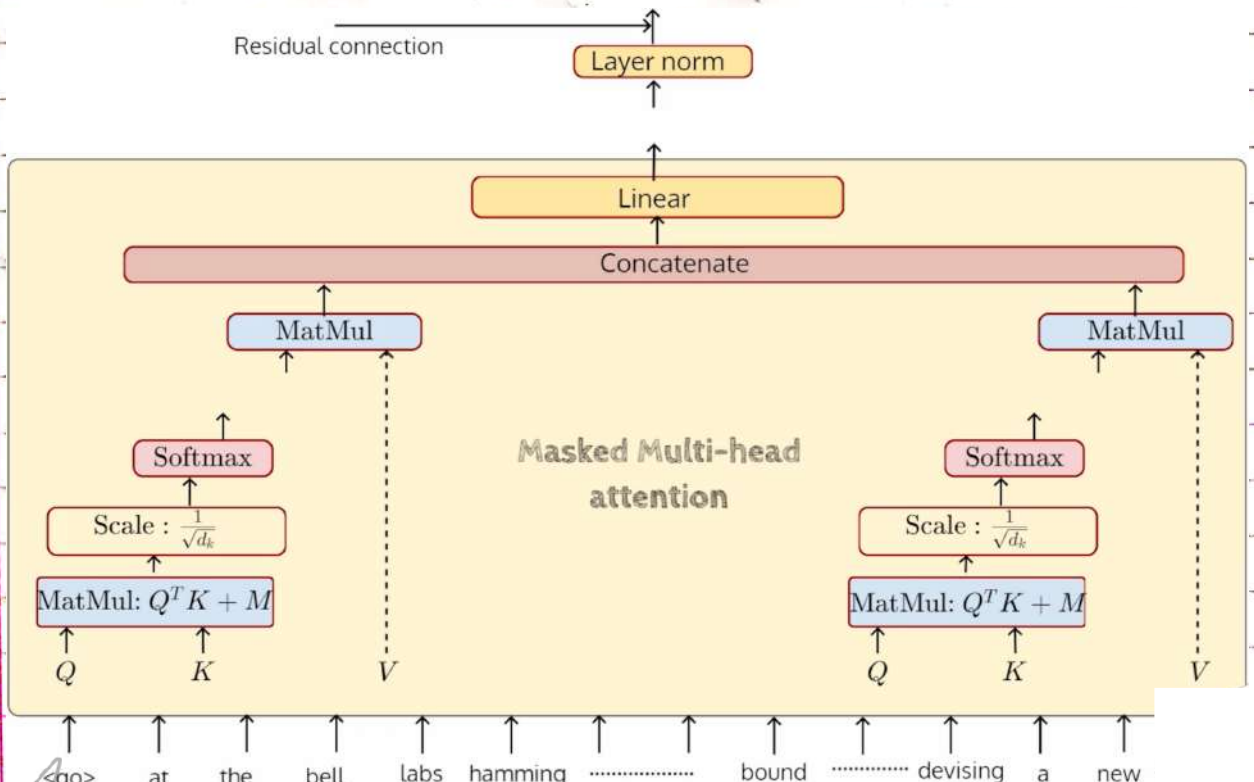
* Generative Pretrained Transformers (GPT)

Note - The transformer block in the figure bellow is similar to the one from previous section ie - decoder only models



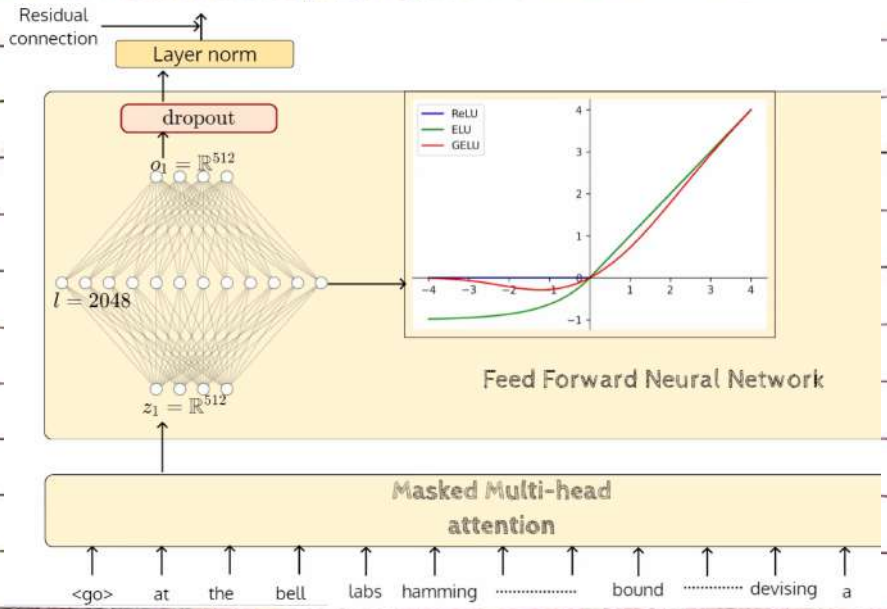
→ Now, in the previous section we also saw that there are two main components of decoder only models, following are the dig diagrams of each component wrt GPT architecture

- Masked Multi-Head Self Attention



Date: / /

- Feed Forward Network



→ Let us now discuss the model configurations of GPT

(1) Tokenizer - Byte Pair Encoding
(2) Vocab Size - 40478
(3) Embedding Dimension - 768
(4) Decoder Layer (Transformer Blocks) - 12
(5) Context Size (sequence length) - 512
(6) Attention heads - 12
(7) FFN hidden Dimension - 768 x 4 ⇒ 3072

→ Let us now count the total parameters in the model

(1) ~~Input~~ Embedding Layers

- Token Embeddings = 40478 x 768
  = 31 x 10^6
  = 31 M

Ishan Modi

- Positional Embeddings = 512×768
  = 0.3×10^6
  = 0.3 M

Total = 31.3 M

(2) Attention layers

Per Block

W_Q, W_K, W_V all have equal number of parameters / dimension

Per Head

$\therefore W_Q = W_K = W_V = (768 \times 64)$

So in one head we have

$3 \times (768 \times 64) = 147 \times 10^3$

Each layer has 12 heads

$12 \times 147 \times 10^3 = 1.7$ M

Linear layer

$768 \times 768 = 0.6$ M

Total = 1.7M + 0.6M = 2.3 M

Each GPT has 12 blocks

Total = $12 \times 2.3 = 27.6$ M

Date: / /

(3) Feed Forward layers (FFN)

Per Block

$$2 \times (768 \times 3072) + \underbrace{3072 + 768}_{\text{bias}}$$

$$= 4.7 \times 10^6 = 4.7\,M$$

GPT has 12 blocks

Total = $12 \times 4.7 = 56.4\,M$

→ Combining (1), (2), (3) → we have 117 M parameter in GPT model

Note - Final layer needs to generate a distribution over the vocab, therefore given 768 dim embedding it needs to generate 40478 dim output. Same is applied for all of the sequence length

Thus we need 768×40478 matrix for the transformation and it is shared, because token embeddings also have same sized matrix.

* Pretraining & Finetuning

Pretraining

→ lets discuss the pre-training configurations

(1) Batch size - 64
(2) Sequence length - 512
(3) hidden dimension / embedding length - 768
(4) Loss -

$$-\sum_{\text{inputs sequences}} \sum_{\text{tokens in sequence}} (\log \hat{y}_i)$$

Date: / /

(5) Optimizer - Adam with cosine learning rate scheduler
(6) Strategy - Teacher forcing.

Finetuning

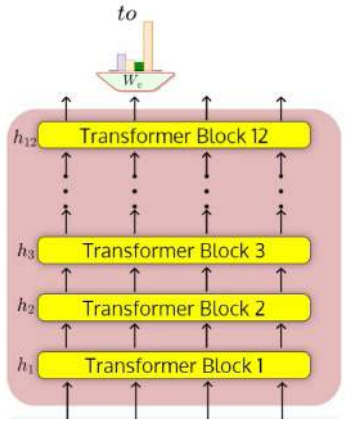
Note - Fine-Tuning can be done in the following ways

- Freezing all the pre-trained weights in the network, add a linear layer head / small neural network, and change only those added weights during backpropagation for required task
- Freezing the bottom k-layers, other as above
- Don't freeze the pre-trained model at all, other as above.

→ Some use cases of finetuning can be

- Sentiment Analysis
- Question / Answering
- Textual Entailment etc

Text Generation is also a usecase, where given a sequence of tokens you can generate new tokens

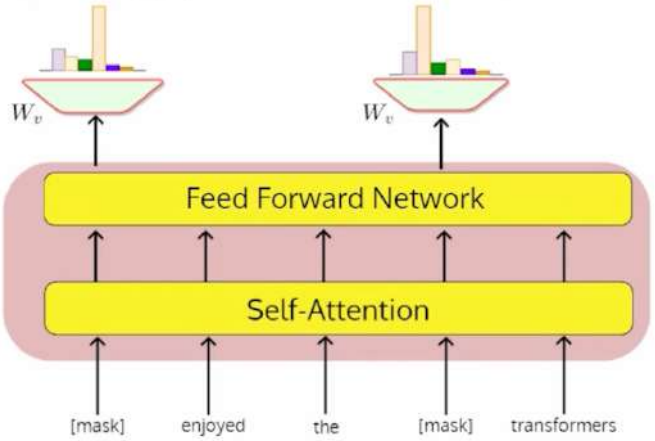


The only problem here is that this seems to be de deterministic ie - get same o/p everytime

Date: / /

* Encoder - Only - Models

- Why do we need encoder only models ?
  - decoder only models are auto-regressive & they look at past tokens to predict future tokens
  - Future & Past tokens both can contribute to the prediction of the current token.



- Now, we used causal language modeling for decoder-only model. In case of encoder-only-models we use something called as masked language modeling

$$P(y_i \mid x_1, x_2, \ldots x_i = [\text{mask}], \ldots, x_T)$$

Here we will provide x_j's as the input & will mask (x_j where $i = j$) & will try to predict

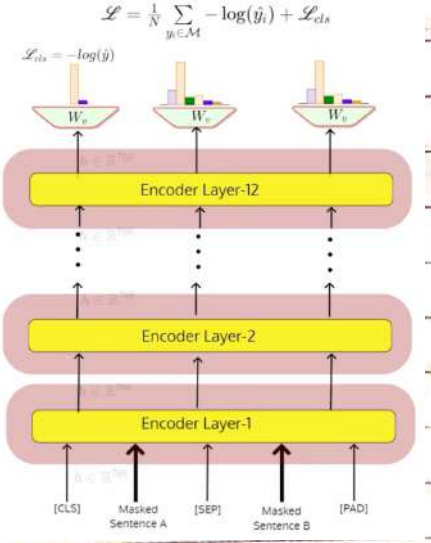
y_i for the mask (x_i) given the x_j's

Date: / /

* BERT (Bidirectional Encoder Representations of Transformers)

→ Continuing from the previous section, the question is how does masking work internally ?

Answer is, you introduce a new token called [MASK] → (has its own embedding) into the vocabulary. Then you pass it through the network and try to learn the actual word through back propagation



→ Following is the model configuration of BERT

(1) Tokenizer - Word Piece
(2) Vocab Size - 30522
(3) Embedding Dimension - 768
(4) Encoder layers - 12
(5) Sequence length - 512
(6) Attention heads per layer - 12
(7) FFN layer Dimension - 768x4 - 3072

Date: / /

→ Lets now count the total number of parameters of the model

(1) Embedding layers

- Token Embeddings = 30522 × 768 = 23 M
- Segment Embeddings = 2 × 768 = 1536
- Positional Embeddings = 512 × 768 = 0.4 M

} We will learn more in pre-training section

Total = 23.4 M

(2) Attention layers

per block / per head

$W_k = W_q = W_v = 768 \times 64$

3 of them = $768 \times 64 \times 3$

for 12 heads = $768 \times 64 \times 3 \times 12 = 1.7$ M

Also, Linear layer = $768 \times 768 = 0.6$ M

Total 12 blocks = 2.3×12

= 27.6 M

(3) FFN Layer = $(2 \times 3072 \times 768) + 3072 + 768$

= 4.7 M

For total 12 blocks, = 4.7×12

= 56.4 M

Date: / /

Adding, all of them, (1),(2),(3) it is around 110 M

* Pretraining

→ Masked Language Modeling (MLM)

We know from previous discussion, how masked language modelling works. But we don't know how many tokens to mask and some other details

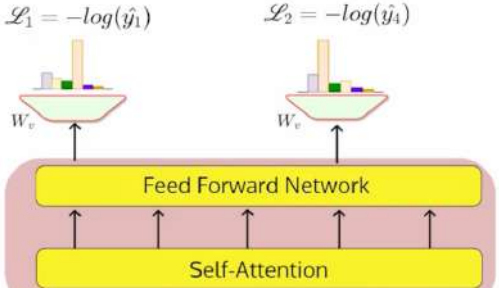
- 15 % of the words are masked from the input sequence
- 80 % of the total masked token are replaced with [MASK] embedding
  10 % of the total masked tokens are replace with random word embeddings
  10 % are unchanged

Eg - Lets say sequence length is 200 tokens

∴ 15 % of 200 → 30 words are selected for masking

80 % of 30 → 24 words are replaced with [MASK]
10 % of 30 → 3 words " " " random word
10 % of 30 → 3 words " remain unchanged

$$\mathscr{L} = \frac{1}{N} \sum_{y_i \in \mathcal{M}} -\log(\hat{y}_i)$$



Date: / /

→ Next Sentence Prediction (NSP)

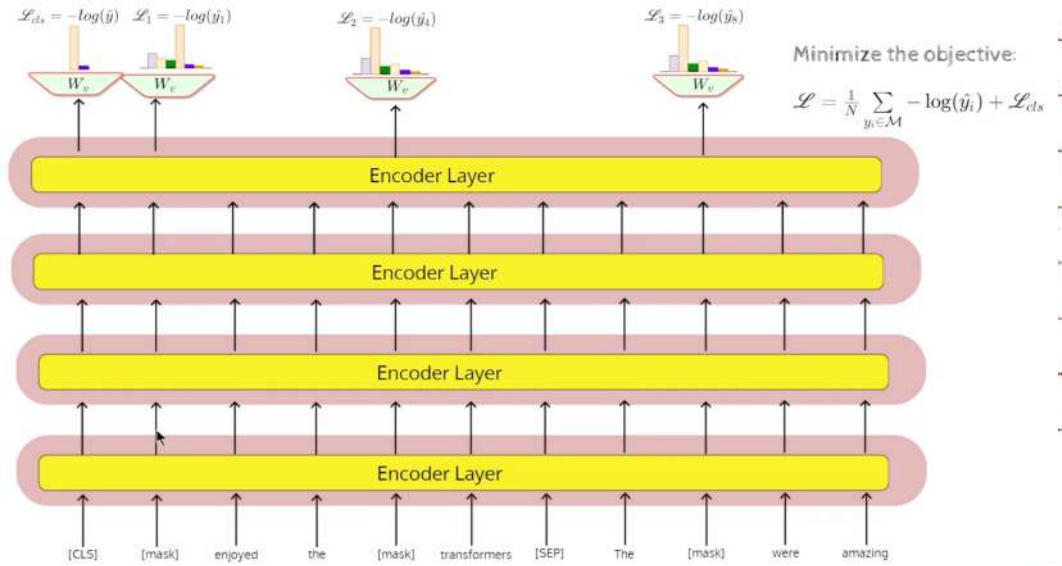
- Introduce [CLS] token that denotes beginning of input sequence
- Introduce [SEP] token that denotes sentence break
- Segment Embedding were introduced, dim ⇒ 768 × 2

ie → $768 \times 1 = SE_A$ & $768 \times 1 = SE_B$

Denotes that it is the first sentence

Denotes that it is the 2nd sentence

- 50% of the sentences had correct sequence A for other 50% of the sentences 2nd sentence was picked randomly from the corpus.



Ishan Modi