TRANSFORMERS

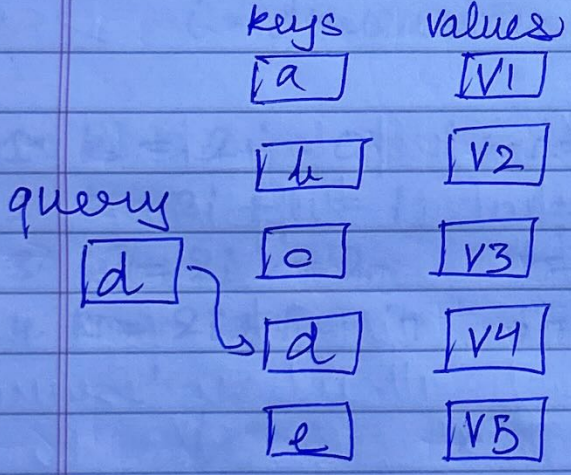
* Attention:
  - Every word's representation as a query to access & incorporate info from a set of values.

thus, transformer is fast for long sequences.

- No. of unparallelizable operations do not increase with sequence length.
- Max. Interaction Dist: $O(1)$, as all words interact at every layer.

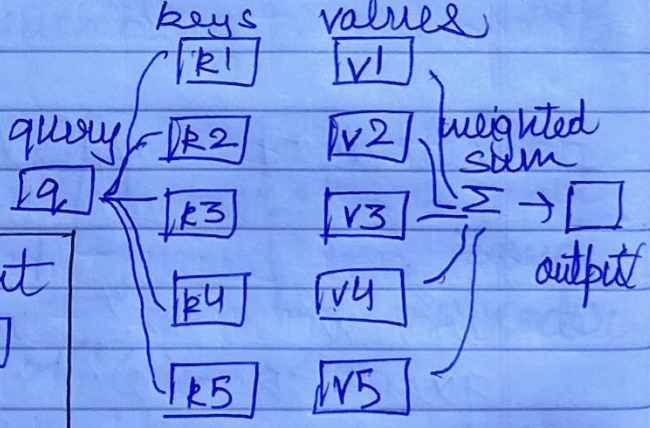
Analogy: Attention similar to look up table (dictionary).

DICTIONARY



query raised, matches a key & we get output as value. (100 %)

ATTENTION



query raised, matches all keys b/w 0 to 1, weighted sum of all values calculated, softmax output displayed.

Self Attention: keys, queries, values from same sequence.

(d,1) (d,V) (V,1) ; for one word, not entire sentence

For each w_i, let $x_i = E w_i$ where $E \in R^{d \times |V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices Q, K, V each in $R^{d \times d}$.

$q_i = Q x_i$ $\quad k_i = K x_i$ $\quad v_i = V x_i$

(d,1) (d,d) (d,1)

2. Compute pairwise similarities b/w keys & queries; normalise with softmax.

$e_{ij} = q_i^T k_j \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$

softmax basically

3. Compute output for each word as weighted sum of values

$o_i = \sum_j \alpha_{ij} v_j$

* Tailoring Self Attention for Sequence Models

→ Sequence Order of Word:

- Use vectors representing each sequence index.
- $p_i \in R^d$, for $i \in \{1, 2, \ldots, n\}$ are position vectors

$\tilde{x}_i = x_i + p_i$

embedding vector x_i

- We do it once at the beginning.

The position vectors can be:

→ Sinusodial position representations

$$P_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2\times\frac{d}{2}/d}) \\ \cos(i/10000^{2\times d/2/d}) \end{pmatrix}$$

- Periodicity indicates absolute position isn't as important
- However, extrapolation isn't possible.

→ P_i be learnable parameters $P \in R^{d \times n}$, each P_i is a column of that matrix.

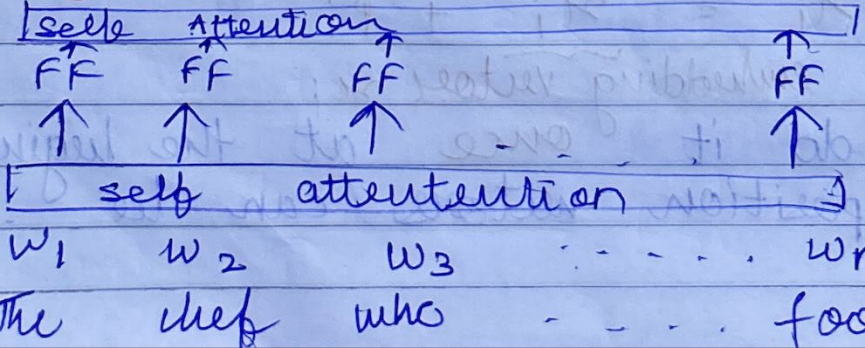
Pros	Con
Flexibility	Cannot extrapolate

- Adding Non-Linearities in Self Attention:

→ There are no element wise non linearities in self-attention, just re-average value vectors

→ Thus, we add a feed forward network to post process each output vector.

** $\begin{cases} M_i = MLP(\text{output}_i) \\ \quad = W_2 * ReLU(W_1 \text{output}_i + b_1) + b_2 \end{cases}$



The FF network processes the result of attention.

- Masking the future in Self-Attention

→ To use self attention in decoders we need to ensure we can't peek in the future.

→ Thus, to enable parallization, we mask out attention to future words by setting attention scores to $-\infty$.

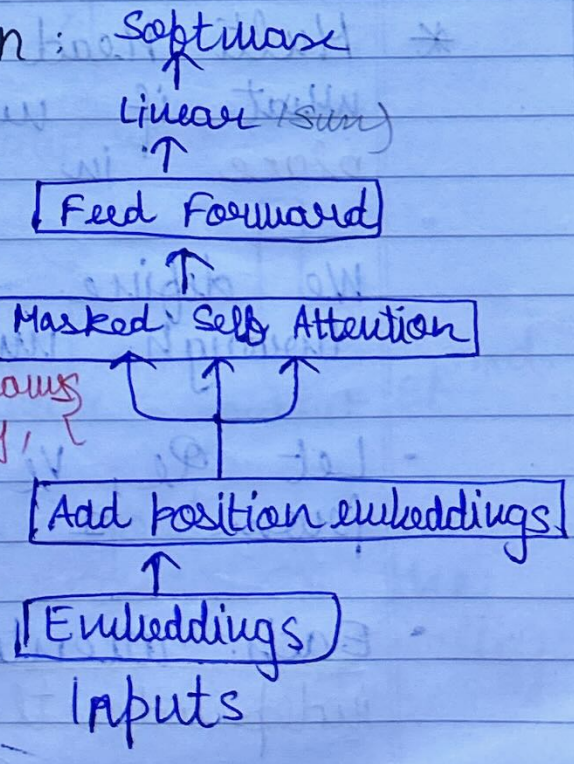
$$e_{ij} = \begin{cases} q_i^T k_j, & j \le i \\ -\infty & j > i \end{cases}$$

	Start	The	Chef	Who
Start		$-\infty$	$-\infty$	$-\infty$
The			$-\infty$	$-\infty$
Chef				$-\infty$
Who				

$-\infty$ part word will not be visible to the corresponding row's word.

* Necessities of Self Attention:

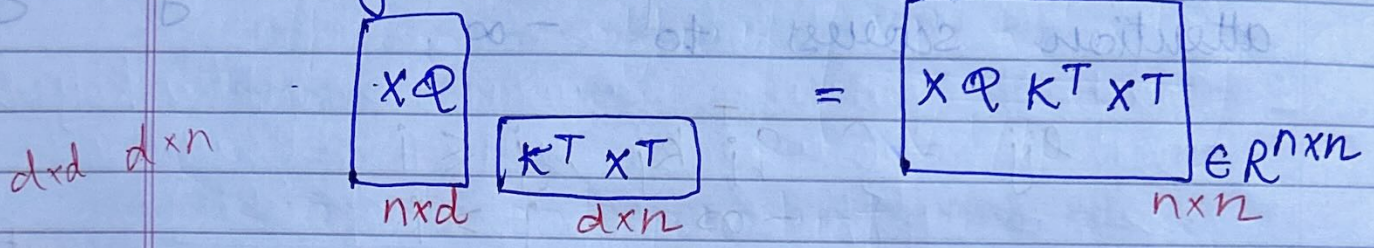
- Self Attention
- Position Representations
- Non Linearities
- Masking (decoder)

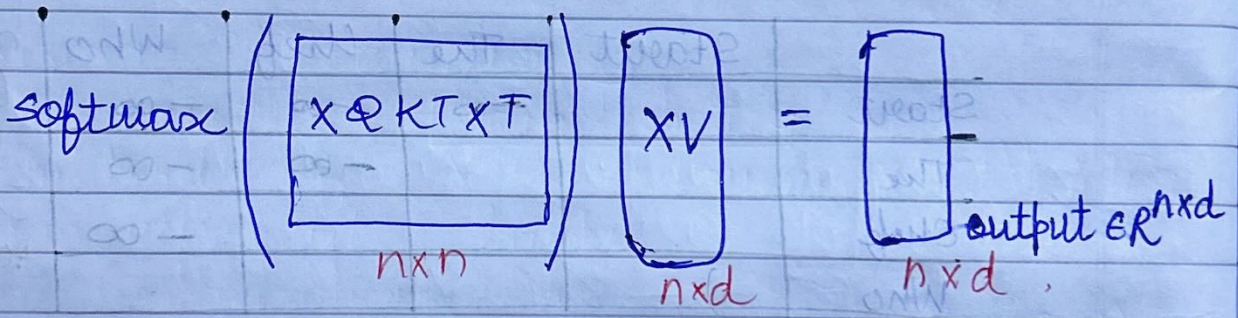


* Sequence Stacked form of Attention:

- Let $X = [x_1, x_2, \ldots ; x_n] \in R^{n \times d}$ be the concatenation of input vectors.
- Note that $XK, XQ, XV \in R^{n \times d}$ ($K, Q, V \in R^{d \times d}$)
- Output = $\text{Softmax}\left(XQ(XK)^T\right)XV \in R^{n \times d}$

Visually:





* Multi Head Attention:

What if we want to look in multiple place in the sentence at once?

We define multiple attention "heads" through multiple Q, K, V matrices.

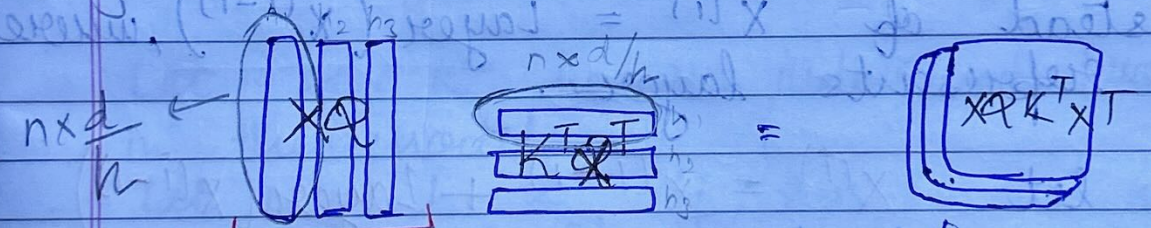
- Let $Q_l, V_l, K_l \in R^{d \times \frac{d}{h}}$ & l ranges from 1 to h.
- Each attention heads performs attention independently

Multihead:
Q, K, V divided into Q_i, K_i, V_i

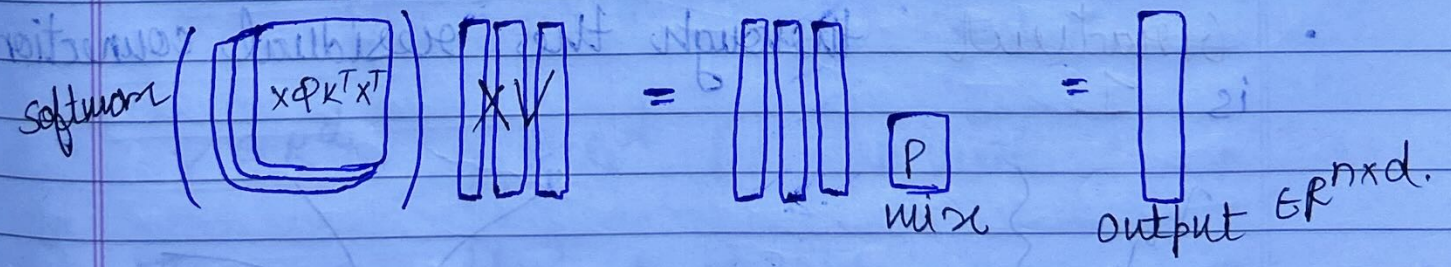
- Then the outputs of all heads are combined
- $\text{output}_i = \text{softmax}(XQ_iK_i^T X^T) \cdot XV_i$ where $\text{output}_i \in R^{d/h}$
- $\text{output} = [\text{output}_1, \text{output}_2, \ldots, \text{output}_h]Y$ where $Y \in R^{d \times d}$
- Multihead self attention is computationally efficient, even though we have h heads.

for our understanding (& XV, XK) in two steps, actually can be restaped directly

→ We compute $XQ \in R^{n \times d}$ & then reshape to $R^{n \times h \times \frac{d}{h}}$ (dimensionality of each head)
→ Then transpose to $R^{h \times n \times \frac{d}{h}}$, now head axis is like batch axis
→ Almost everything else is identical & matrices are the same size



3 = h, here, meaning remains the same by restaping, just that Q, K, V, X for each head are separated now.



- Generally (not always) it is good to have same no. of heads throughout & the no. of heads wrt to the dimensionality of Q, K, V.

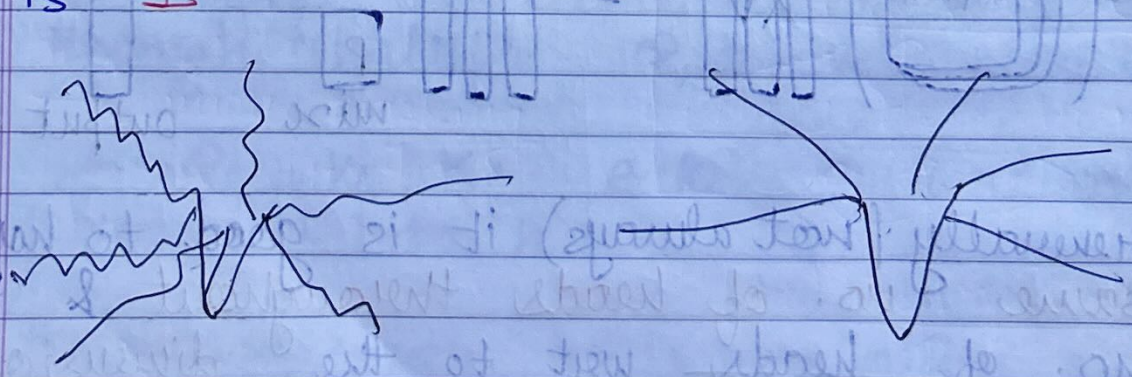
- When dimensionality d becomes too large, dot product become large & so input softmax will be large, making gradients small

- Thus we divide attention scores by $\sqrt{d/h}$ to stop the score from becoming large.

$$\text{output}_e = \text{softmax}\left(\frac{XQ_e K_e^T X^T}{\sqrt{d/h}}\right) * XV_e$$

↑ scaled dot product

* Residual Connections:
- Help models train better
- Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$, where i represents layer.
- We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$ (so we only have to learn "the residual" from previous layer.
- Gradient through the residual connection is 1.



no residual
GD (Gradient descent)

residuals
(GD)

Residual Connection: Implemented in the Add & Norm Layer after each attention & FFT layer.

Advantages:
→ In a deep NN, where outputs of deeper layers are close to unity, it helps the layer to learn faster.
→ Avoids vanishing gradient as helps gradient flow as gradient is 1.
→ Improves training.

* Layer Normalisation:
• Trick to help models train faster.

• Intuition: Cut down uninformative variation in hidden vector values by normalising to unit mean & standard deviation within each layer.

Let $x_i \in R^d$ be an individual (word) vector in the model.
Let $\mu = \frac{1}{d}\sum_{j=1}^{d} x_j$; this is the mean, $\mu \in R$

$$\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2} \Rightarrow \text{Standard dev.}$$
$\sigma \in R$

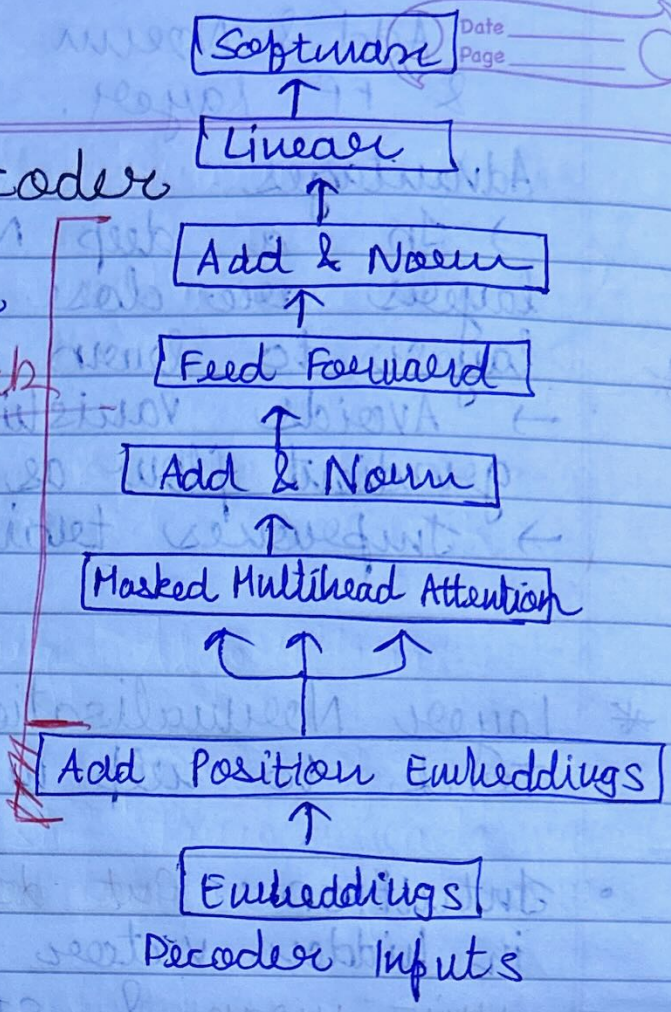
$\gamma \in R^d$, $\beta \in R^d$ be learned gain & bias.

Then: $$\text{output} = \frac{x - \mu}{\sqrt{\sigma + \varepsilon}} * \gamma + \beta$$
(μ is broadcasted (coust) to dimensionality)

Here: $x_{j+1} = x^{(j-1)} + \text{layer}(x^{(j-1)})$ } Residual connection.

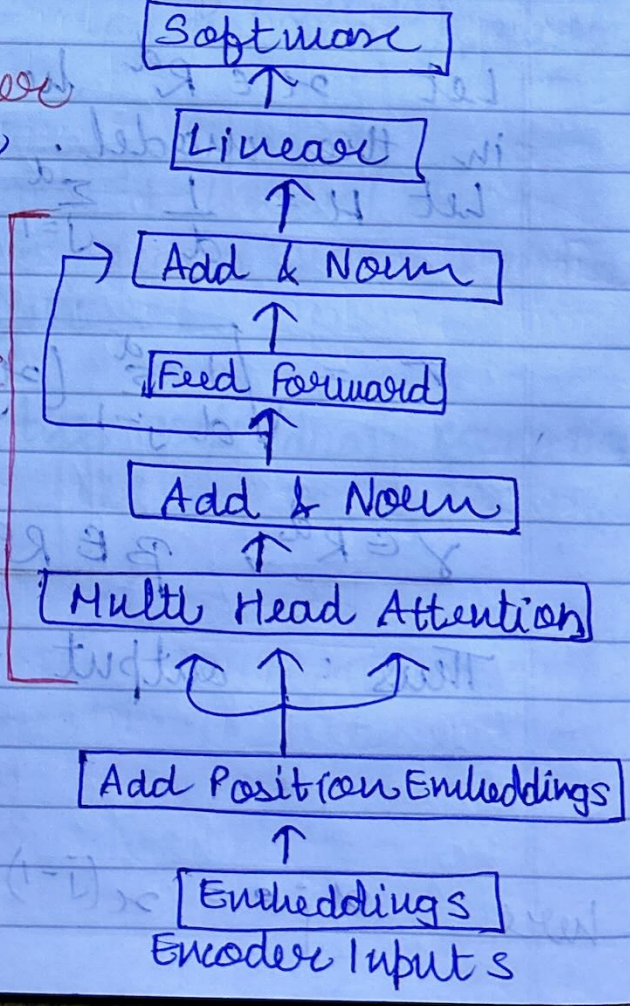
* Transformer Decoder

- Decoder is a stack of blocks.
- Each block consists of:
  - Self Attention
  - Add & Norm
  - Feed Forward
  - Add & Norm

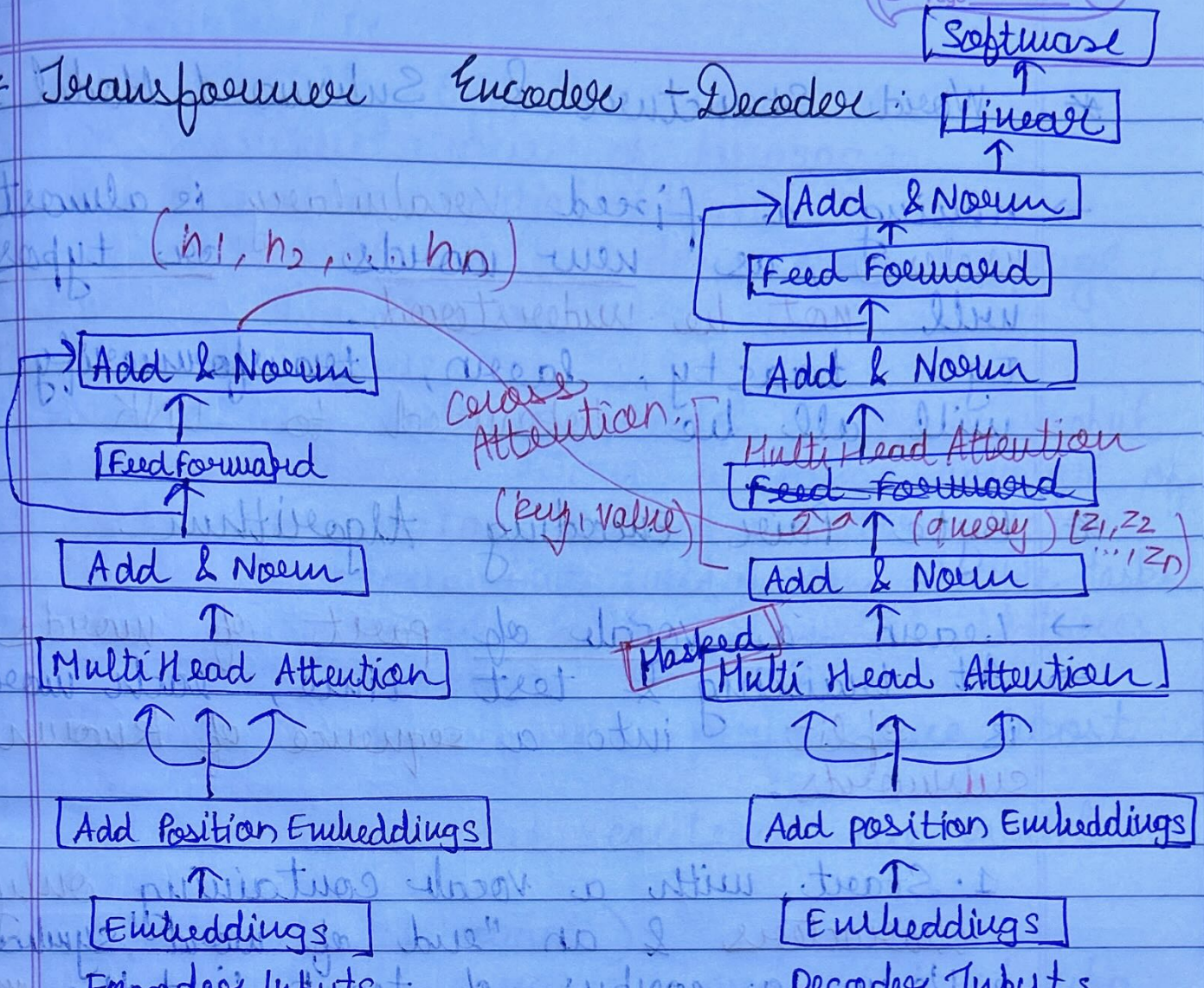


* Transformer Encoder

- Now, transformer decoder is unidirectional as for language models.
- However, if we want ~~bidirectional~~ bidirectional context, we do:
  → remove masking from encoder



* Transformers Encoder - Decoder :



→ let $h_1, \ldots, h_n$ be output of encoder : $x_i \in R^d$
→ let $z_1, \ldots, z_n$ be input from decoder : $z_i \in R^d$

$$k_i = K h_i \quad v_i = V h_i \} \text{ from encoder}$$
$$q_i = Q z_i \} \text{ from decoder.}$$

* Drawbacks of Transformer :
- Quadratic computation
- Position representations.

* Word Structure & Subword Model:

→ Having a fixed vocabulary is almost useless, as new words or typos will not be understood.
Eg: taasty, laern, transformerify will all be tokenised to UNK.

* Byte Pair Encoding Algorithm:

→ Learn a vocab of part of words
⇒ At training & test time, each word is split into a sequence of known subwords.

1. Start with a vocab containing only characters & an "end of word" symbol.
2. Using a corpus of text, find the most common adjacent characters "a, b"; add "ab" as a subword.
3. Replace instances of the character pair with the new subword, repeat until desired vocab size

Here, the model learns through contextual representations.
For eg: I record the record.
both the "records" will have their diff. meaning here, while word2vec would just fail.

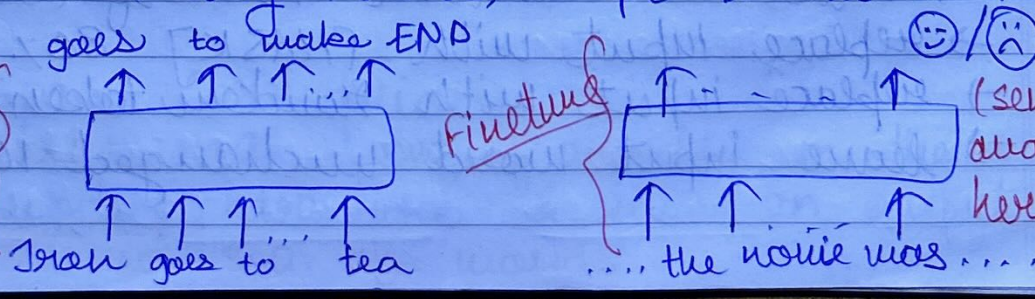
Pretraining is
Exceptionally Effective in:
→ representations of language
→ parameter initialisation
→ probability distribution over language.

labelled data is expensive.

This is because:
→ Pretraining allows model to learn useful representations from a large amount of unlabeled data, making it more data efficient when fine tuning on specific tasks.
→ Helps the model learn general features of the language.
→ Faster convergence, performance boost.

By predicting the content the model can learn:
→ patterns (fibonacci)
→ understand scenario based off text info.
→ using content can remember male/female or singular/plural.
→ predict geography & understand world.

Pretraining Steps:
Step 1: Pretrain (on language modelling) lots of text (unstructured data), learn general things.
Step 2: Finetune (on your task)
Not many labels, adapt to the task



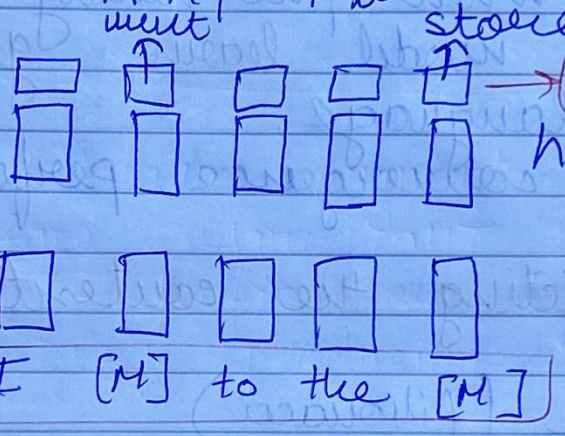
* Pretraining Encoders:
→ Encoders get bidirectional context, so language modelling is not possible.

chosen randomly

Idea: replace some fraction of words in the input with a special [MASK] token; predict these words.

$h_1, \ldots, h_T = \text{Encoder}(w_1, \ldots, w_T)$

$y_i \sim A h_i + b$



either seq is $\tilde{x}$

Only add loss terms from words that are "masked out". If $\tilde{x}$ is the masked version of x we're learning $P_\theta(x|\tilde{x})$

masked LM

* BERT: Bidirectional Encoder Representations from Transformers

Masked LM for Bert:
- Predict a random 15% of (sub)word:
→ replace input with [MASK] 80%
→ replace input with random token 10%
→ leave input word unchanged 10%

- In the pretraining input to BERT was two separate contiguous chunks of text.
- Although this was not necessary to predict the "next word" it gave the model a hard time.

Intuition:

We're coming up with hard problems for the network to solve such that by solving them it has to learn a lot about language. We're defining those problems by making simple transformations removing info, & both randomly.

- The best usecase of BERT is NOT next-word prediction or generating a large sequence.
It works well masked inputs.

* Fine Tuning & Light Weight Fine Tuning:
→ works great, → expensive,
→ memory intensive

Prefix Tuning, Prompt Tuning.
→ initialising some parameters in the beginning & only fine tuning them.
→ We can also split input matrix into low rank matrices as $A \in R^{d \times k}$, $B \in R^{k \times d}$

* Span Corruption:
Masks a seq of words in the encoder decoder structure & gives an output of masked seq. of words.

Majority of the language models use decoder only, the reason is speculated to be that's its easier (less complex) & does not require data to split/divide in encoder/decoder.

GPT → BERT → GPT2 → GPT3

more focussed on the generation part of sent.

much larger than GPT2, upto 100 million more.