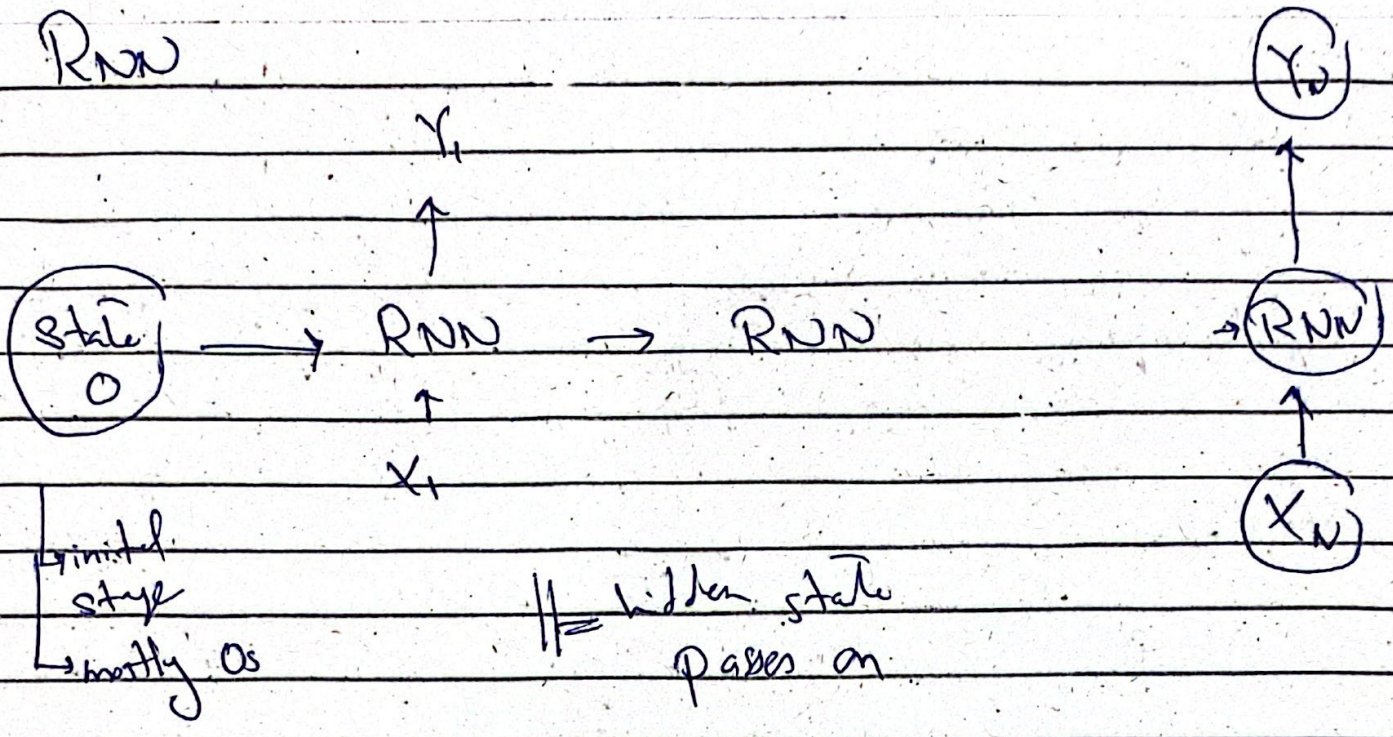
TRANSFORMERS:

RNN



RNN Limitations

- bad at remembering long sequences
- vulnerable to vanishing gradients
- They learn the sequence with timesteps

In order to get result for Timestamp, you'll have to wait ~~for~~ till network's $t-1$, that's why its computationally inefficient. So no parallelization

Transformers solved this problem by using attention-head.

Why transformers are better input

↓

Chatbot

↓

they use self-attention output

⇒

input → Encoder

Step 1: tokenize the sentence
2: feed it to embedding layer (Encoder)

	why	are	transformers	better
input:	2339	19081	2024	2488

	the	LSTMs
	2084	100

After feeds to embedding layer, embedding layer will convert each token into vector representation.

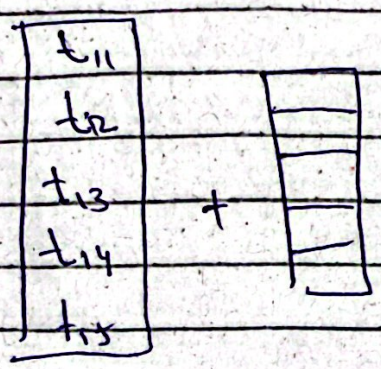
It would generate 6, 5×1 vectors

Step 3 the output from embedding layer will go through positional encoding

↳ If we replace 'LSTM' and transformers in "Why LSTM are better than Transformers."
↳ while meaning has changed but for model both sentences are equal as it is important to maintain positional information (sequence order).

encoding

Positional ~~indexes~~ adds positional vector into embedding vector which holds the info about position of each word.

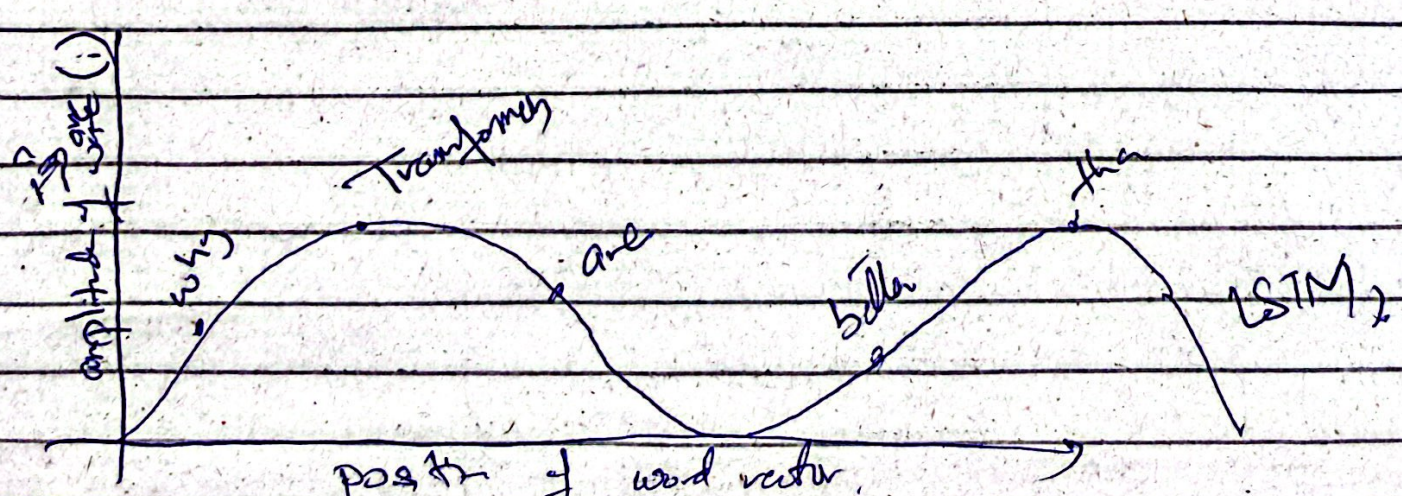


for each token

Embedding + Position vectors

Researchers used sine wave to compute position vector by using the

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{100^{2i/d_{model}}}\right)$$



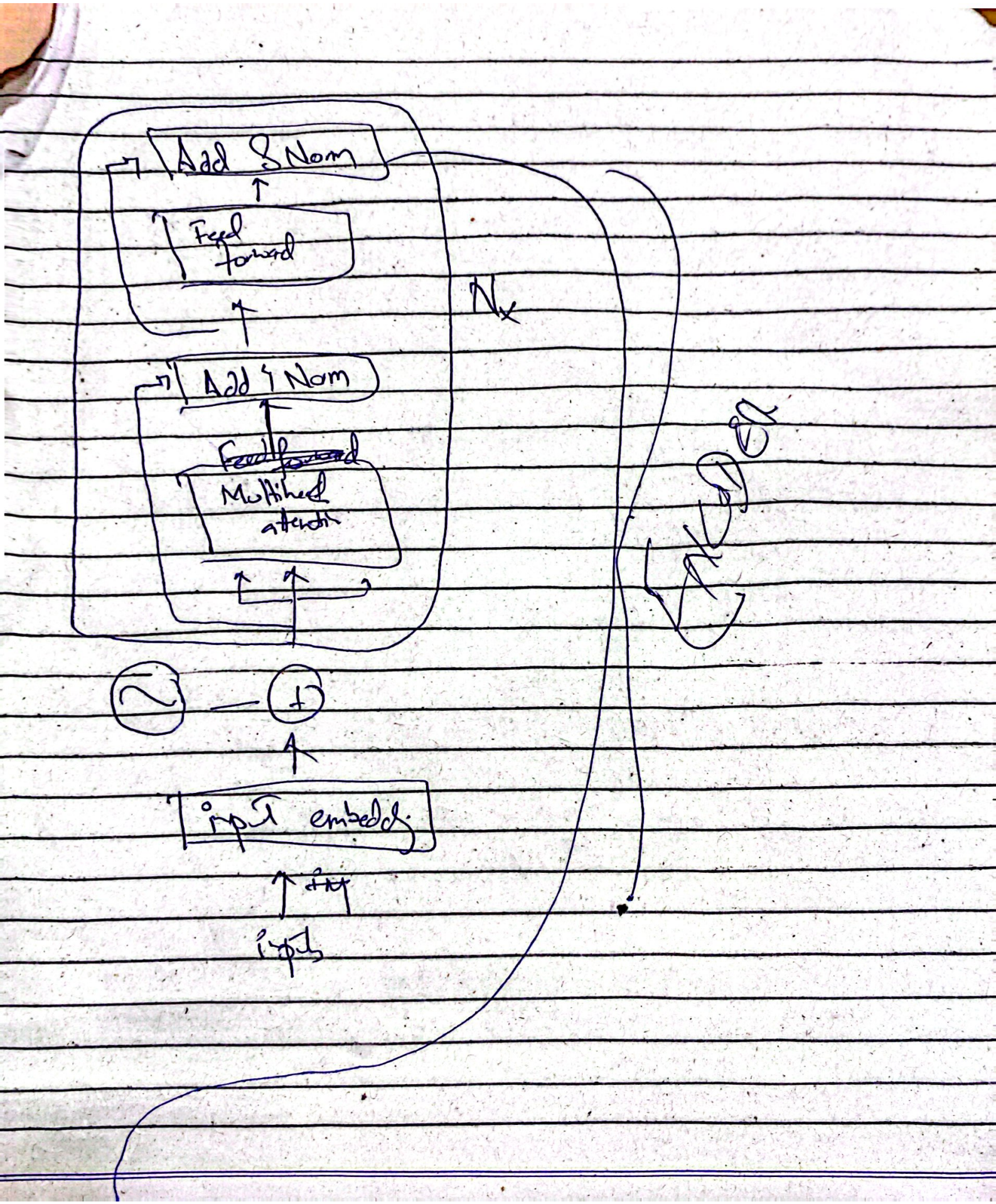
[contd]

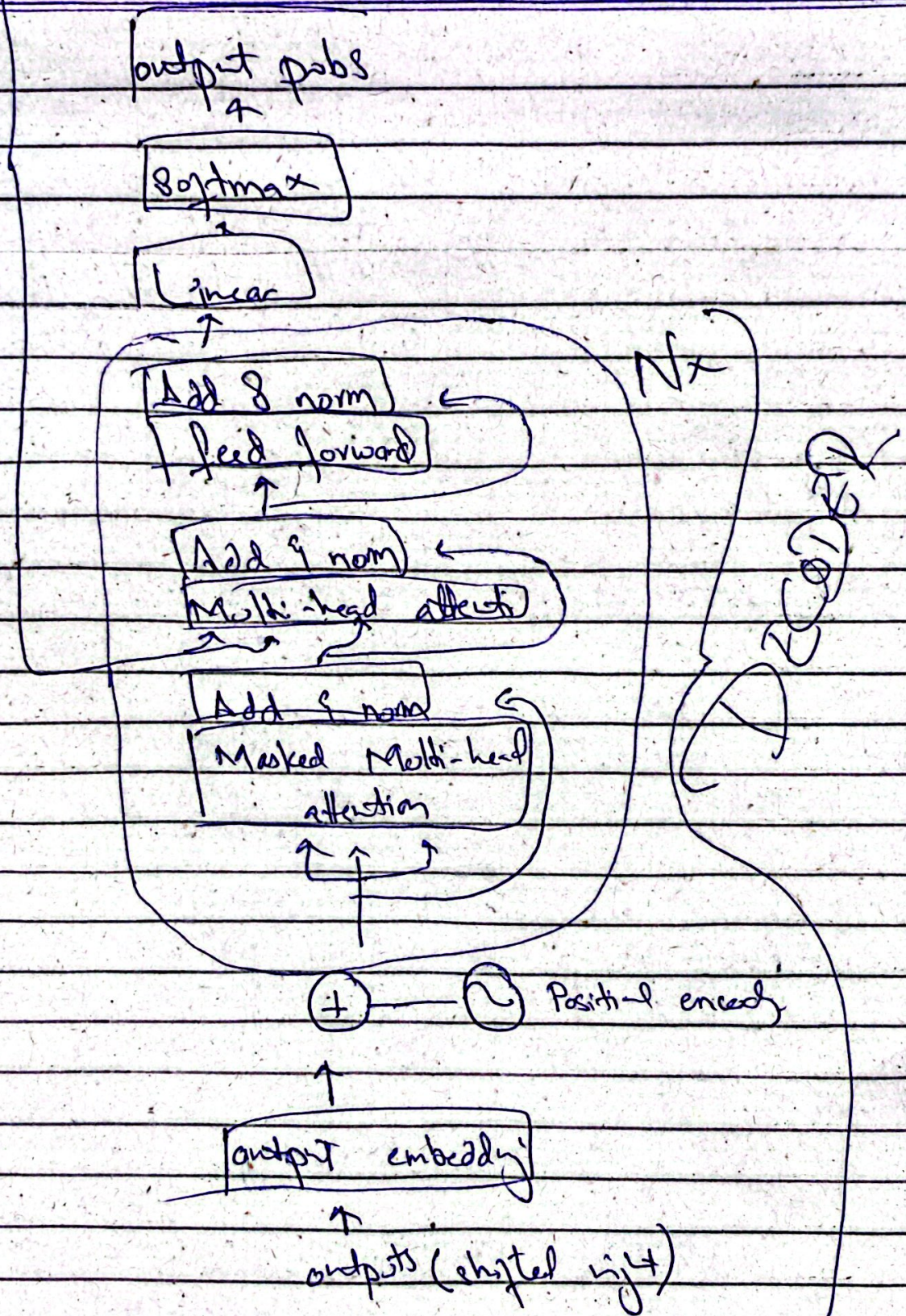
vanishing gradient

$$\frac{dy}{dx} = \frac{dy}{df} \cdot \frac{df}{dx}$$

Vanishing { if dy/df & df/dx are < 1, it will give to a more smaller number.

Exploding { if > 1, result would be a more greater number.

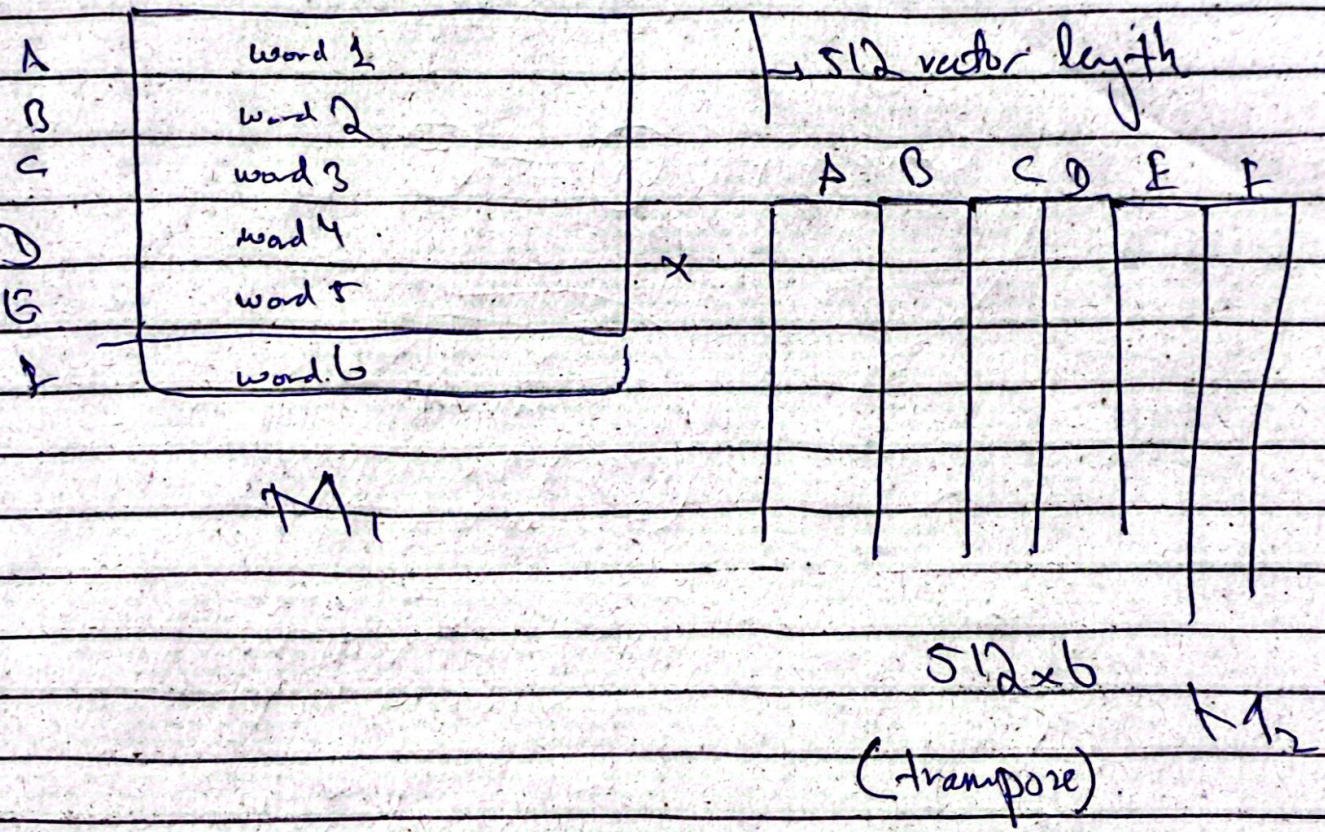

Add & Norm
Feed forward
Add & Norm
Multihead attention
Nx
input embeds
inputs


output probs
Softmax
Linear
Add & norm
Feed forward
Add & norm
Multi-head attention
Add & norm
Masked Multi-head attention
Nx
Positional encoding
output embedding
outputs (shifted right)

Notations

Input matrix (sequence, d_model)

6 × 512 — 6 words



⇒ result of both would be (6 × 6)

here (1,1)

AA	AB	AC	AD	AE	AF
BA	BB				
CA		CC			
DA			DD		
EA				EE	
FA					FF

6×6

here (1,1) is dot prod of A in M_1 & A in M_2

Embeddings are not fixed, they are parameter to our model

Encoder

input embedding

sentence (tokens): [Your] [cat] [is] [a] [lovely] [cat]

input IDs: [105] [6587] [575] [3578] [63] [6587]

Embedding (vector size 512) → 512×1 for each

our model will learn to change these 512 values for each token so that it could represent sentiment.

Due to 512 size its 512d model,

$d_{model} = 512$

Positional Encoding

↳ Each word should carry some info abt its position in sentence.

↳ we want model to treat words that are appear close to each other as close and words that are dist as dist.

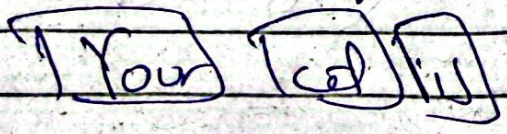
↳ we want the positional encoding to represent a pattern that can be learned by the model

For each word vector we create some special vector called positional encoding of same length ie 512 that we add to these embeddings.

Embedding token (512 × 1) + Positional encoding (512 × 1) = ENCODER INPUT (512 × 1)

↳ It is not learned. It is computed only once & fixed.

How positional encodings are calculated?



$PE(pos, 2i) = \sin\left(\frac{pos}{(1000)^{\frac{2i}{d_{model}}}}\right) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$ for even [0, 2, ...

↳ $PE(1,0)$

$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$ for odd (1, 3, ...

↳ $PE(1,1)$

Where d_{model} = 512

$PE(pos, 2i)$

↓

position of word inside sentence, starts from 0.

For each sentence, positional encodings will be same

After computing positional encodings once, we can reuse them for every sentence whether it's train, or inference.

Why Sine & Cosine fn?

diff fns like cos & sin naturally represent a pattern that model can recognize as ~~natural~~ continuous, so relative positions are easier to see for model

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

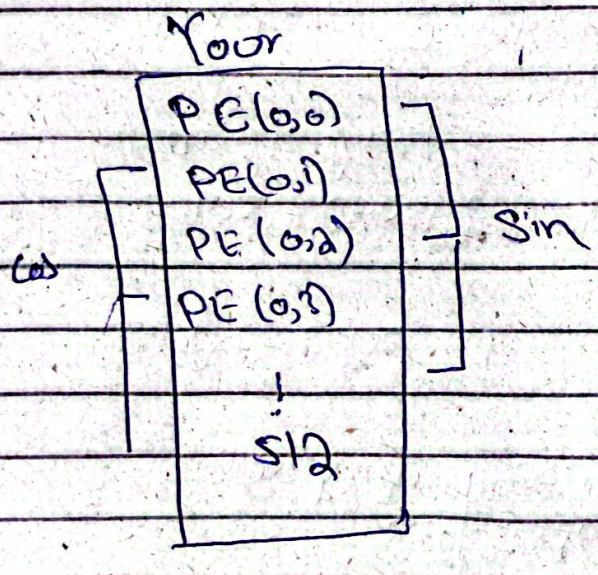
$$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

k: pos of object in sequence
d: dimension of output embedding space

$P(k, j)$ - positional fn for mapping a position k in the input sequence to index (k, j) of the positional matrix

n - User defined scalar, set to 10,000 by author.

i - used for mapping to column indices $0 \leq i < d/2$ with a single value of i maps to both sine and cosine.



$PE(pos, 2i)$

We will have same positional encoding for each sentence because positional encodings are calculated once and are reused for every sentence that our model will see - during inference or training.

Multi-head attention → self-attention with single head

⇒ Single head attention: self-attention is a mechanism that existed before they introduced transformers. They (Authors of the transformers) changed it to multi-head attention.

⇒ What is self-attention?

self attention matrix for input matrices (Q, K, V) is calculated as

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

to calculate self attention

Self-attention allows model to relate words to each other

Simple case

we consider the input sequence length seq = 6 and $d_{model} = d_k = 512$.

Matrix Q, K, V are just input sentence.

where Q, K, V are the concatenation of query, key and value vectors.

Self-attention because we are computing relation b/w each word in sentence to other word in sentence

Q - input sentence
(6, 512)
↳ 6 - embedding of these 6 tokens

| Your | cat | is | a | lovely | cat |

K^T - same sentence but transposed (512, 6)

⇒ ~~Q x K^T~~
~~(6, 6)~~

$$\frac{Q \times K^T}{\sqrt{512}} \quad \because d_k = 512.$$

Then softmax

softmax $\left(\frac{Q \times K^T}{\sqrt{d_k}}\right)$.

$\text{Softmax}\left(\frac{Q \times K^T}{\sqrt{512}}\right)$ =

	Your	cat	is	a	lovely	cat	Σ
Your	0.268	row1 x col2				row1 x col6	1
cat	0.124						1
is	0.147 ?						1
a							1
lovely							1
cat	row6 . col2						1

6 x 6

(1x1) first each value represents dot product of first row with the first column

Softmax makes these value in such a way that they sum upto 1.

So [1x1] is dot product of first word embeddings of word 'Your'

[1x2] = dot product of embedding of word 'Your' with embedding of word CAT 'cat'

number show identity of value

So now multiply with $V_{6\times512}$

$$\text{Softmax}\left(\frac{Q\times K^{T}}{\sqrt{512}}\right)V_2 \quad \text{Attention}$$

$$\boxed{\text{Softmax}\left(\frac{QK^{T}}{\sqrt{512}}\right)}_{6\times6} \times \underbrace{\boxed{V}}_{6\times512} = \boxed{\begin{matrix}\text{Attenti}\\ 6\times512\end{matrix}}$$

// Same matri shape as embedding

Now this new attn embedding represnt
- → meaning of the word - input embedding
- → posti of word (positional encoding)
- → relati b/w each words (attention) with all other words.

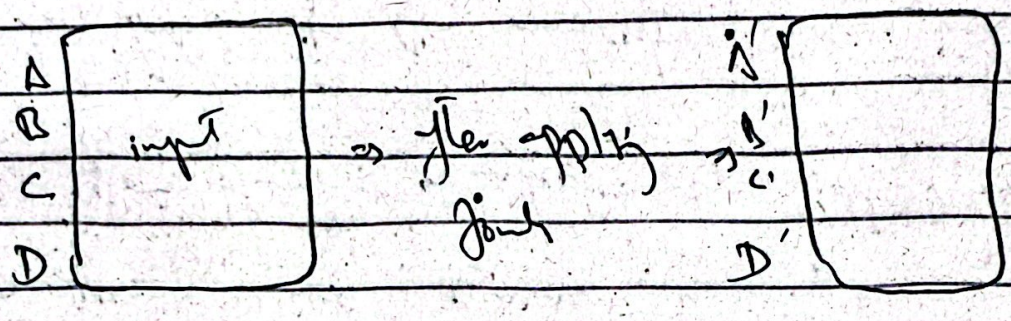
This is self-attenti (single head)

V represents the meaning or significance of each token within the context of the entire sentence. This info is crucial for the attention mechanism to effectively weigh the importance of each token when generating the output.

self attention captures the position in the sentence(given by positional encoding) but of each word's interaction with other words.

Self attention in detail

→ Its permutation invariant: (if we don't consider positional encoding)



Permutation invariance means that model's output remains the same even if the order of the input elements is changed. In context of self-attention, this means that the attention mechanism does not depend on absolute positions of elements in the input sequence. It only depends on the relationship b/w them the elements.

~~Ali is good~~

rather than index/position of word in sequence, it depends on relationship b/w words.

mean token of

If we change B i.e position, the value of B' and C' would change but their positions will change.

→ Self attention requires no parameters.

↳ uptill now, interaction s/w words has been driven by their embedding & the positional encoding. This will change later.

↳ No parameter being learnt except embedding vectors

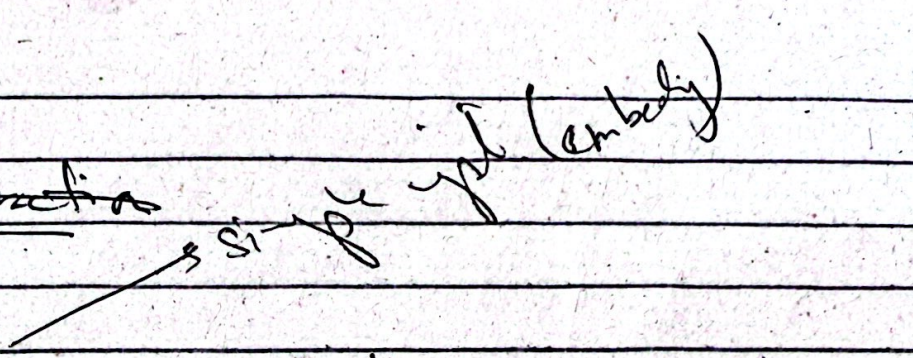
→ we except values along diagonal to be higher because it's dot product of embedding with itself.

→ If we don't want some position to interact i.e YOUR and CAT, put value = $-\infty$, before softmax.

$e^{-\infty} \approx 0$.

To be used in decoder

Multi-head interaction



Q - Query vector represents the current state of model or current position in input sequence. It captures the information that the model is currently interested in and helps determine which parts of the input sequence are most relevant to task.

K - Key: They key vector represents the information from all input tokens. It provides a way for the model to asses the relevance of each input token to the current query. Key vector is usually calculated using a learned embedding. It can also be transpose of Q i.e $K = Q^T$

(V) - value vector represents the learnt representation of the input tokens. It captures contextual info associated with each token, allowing the attention mechanism to weigh the importance of each token when generating output. The value vector is typically calculated using a learned embedding.

Seq - sequence length, d_{model} = Size of embedding vector

h = # of heads

$d_k = d_v$ $\qquad$ $d_v = d_{model}/h$

In positional encoding - we used pre-trained sequential vectors

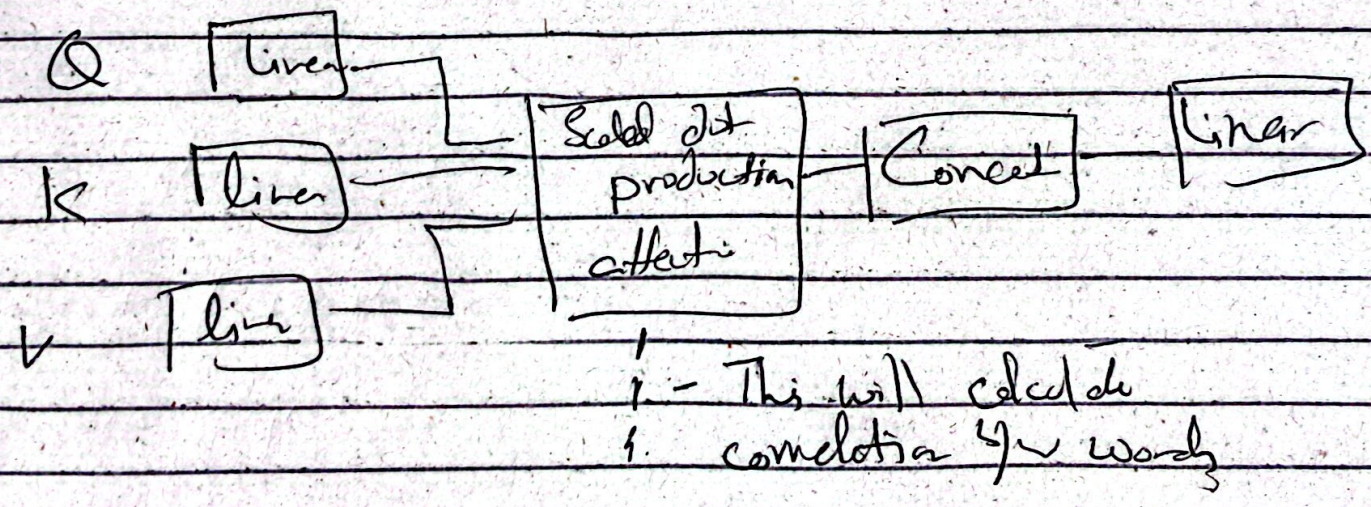
Query - raw input

Key - After doing preprocessing Query

* The we use that key to model best results | output

value

Input to Q, K, V are same (sentence)



1. - This will calculate correlation b/w words

Scaled dot Product Attention

- Lets see how this layer builds correlation

[wait]

Calculating attention comes primarily in three steps.

- First we take Q and each key and compute the similarity b/w two to obtain a weight.
- Second step is to use a softmax fn to normalize these weights and finally to weight these weights in conjuction with corresponding values and obtain the final attention

In current NLP, key = value mostly

⇒ Multi-Head Attention

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

$$d_k = \sqrt{512}$$

$$\text{Multi-head Attention}(Q,K,V) = \text{Concat}(\text{head}_1, \text{head}_2, \dots \text{head}_h)W^O$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

h - # of heads

$d_k = d_v = d_{model}/h$

Q, K, V = input sequence

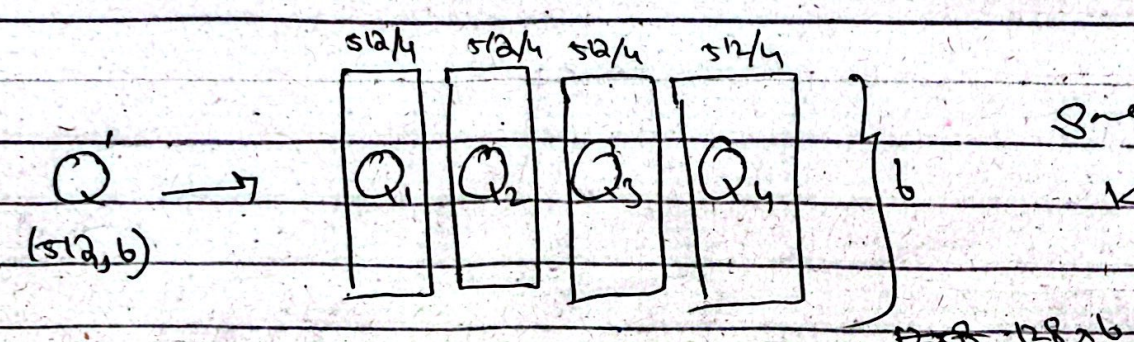
input (seq, d_{model}) → $6, 512$

$Q_{(6,512)} \times W^Q_{(512,512)} = Q'_{(6,512)}$

$K_{(6,512)} \times W^K_{(512,512)} = K'_{(6,512)}$

$V_{(6,512)} \times W^V_{(512,512)} = V'_{(6,512)}$

Now we have to split Q', K', V' into smaller matrices by ~~Sequence~~ d_{model} dimension



Same for K' and V'

$$d_K = d_{model}/h = 512/4 = 128$$

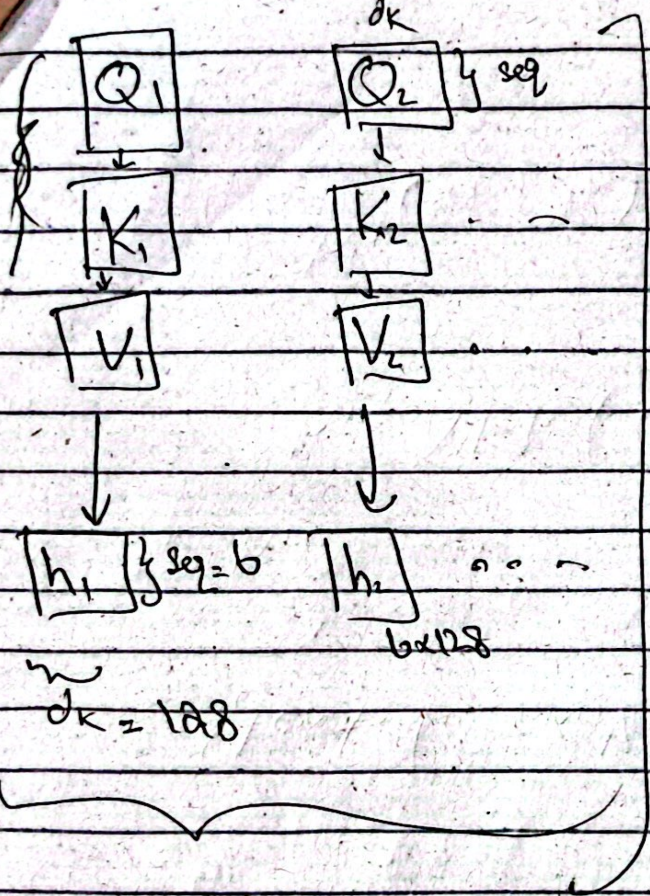
Now calculate attention ~~for~~ b/w these smaller matrices ~~using~~ Q_1, K_1, V_1 using expression

$$\text{Attention}(Q_1, K_1, V_1) = \text{softmax}\left(\frac{Q_1 K_1'}{\sqrt{128}}\right) \cdot V_1$$

for h_1

$$h_1 = \text{Attention}(Q_1', Q_2', V_1)$$

h_1 (seq, dv) $\quad dv = d_K$



dv because last calculation is done by multiply V.

Each of these is 6x128

dmodel

Combine all heads

(6 x 512)

~~MultiHead~~ (Q, K, V) = Concat(h1, h2, ... hn)

~~along the dimension~~

Combine all heads

$H = Concat(h_1, h_2, \ldots h_4)$

↳ $(b, 512)$ ↳ h_i is $(b, \frac{128}{d_v})$

Multiply result H by W_o

Multihead (QKV) $H \times W_o$

(seq, d_{model}) (d_{model}, d_{model})

↳ Because we want each head to work different aspects of each word.

↳ because a word can be noun in one case & verb & other depending on context.

So, one head may learn to relate as noun and other as adverb.

→ Attent ca be visualized.

When we calculate attent b/w Q_1 & K_1 is multiplied.

$$\text{softmax}\left(\frac{Q_1 K_1^T}{\sqrt{d_k}}\right)$$

we get a correlation matrix as seen before. This score represents intensity b/w two words.

Why Q, K, V?

	Keys	values
Query "love" →	Romantic	Titanic
→	Action	Dark knight
- - - →	Scifi	Inception

⇓
heaviness represent intensity

Add. and Norm.

↳ to introduce add & norm we need layer Normalization

layer Normalization

↳ input batch of n-items.

↳ each ~~embedd~~ item is an embedding vector

item 1

calculate μ_i & σ of each of these items independently from each other and we replace each value with another value given by

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_{ij}}{\sqrt{\sigma_{ij}^2 + \epsilon}}$$

Normalizing so, new values are in range 0 to 1

Also

$$\hat{x}_j = \frac{x_i - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \quad \text{or} \quad \frac{\hat{x} - \mu}{\sigma}$$

we multiply $\hat{x}_j$ with λ & add β

$$= \lambda(\hat{x}_j) + \beta$$

because for $\hat{x}_j$, it might be the case that we have restrict to zero where $\sigma \neq 0$.

& the network learn β, λ to introduce fluctuation.

Batch Norm vs layer Norm

↳ Same feature for all batch

↳ one item at a time

Add & Norm

feed forward

Add & Norm

Multi Head Attention

Encoder

Keys

values

Query

Add & Norm

Masked multi-head attention

Decoder

output embeddings are same in works as input embeddings

+ positional encoding

output embeddings

outputs

Multihead Attention

takes Keys & values from encoder and Query from decoder

↳ So attention is not self attention but cross attention

Query : Output of Masked multi-head attention

Masked Multi head attn is ~~attn~~ self attn of input sentence of decoder

Feed forward is just a fully connected layer

Masked Multi-head Attn

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

So, QK^T

	Your	cat	is
Your	0.268	$-\infty$	$-\infty$
cat	0.124	0.268	$-\infty$
is	0.147	0.123	0.262

⇒ Then softmax will replace $-\infty$ with 0.

→ we don't want Your to watch future words

→ Cat will only watch 'Your Cat'

We do this in multi head attention before applying softmax.

Inference & Training of a Transformer Model

→ Training

I love You

↓

Ti amo molto

Input: <SOS> I love you very much <EOS>

↳ Will model about what it start & what is end it ~~end~~ of sentence

↓

Encoder Input:

decoder input

[<SOS> Ti amo molto

↳ we made this sequence of fixed length ~~because~~ when by adding padding.

If our model support seq length of 1000, here we have 4 tokens (<SOS> Ti amo molto), will add 996 of padding to make the sequence long enough

To get back from embedding to the relative dictionary

Decoder output (b, 512) → [Linear layer] → (b, vocab size)

& it will tell, for every embedding it sees, its (token) position in our vocab.

label: Ti amo molto <EOS>

this is what if we expect from our model give "<SOS> I love you very much"

Cross entropy loss

Then calculate ŷ via softmax.

Cross entropy loss (ŷ vs label)

but why <SOS> & <EOS> tokens

we expect our model to output

Ti amo molto <EOS>

↳ this is label / target

Here, in decoder input, our sequence length is 4 '<SOS> Te amo molto'
Lyer dit have padding here.

and we want
"Ti amo molto <EOS>"

So, when our model will see <SOS>, it will output first token as output
"Ti"

and when our model will see "Ti", it will see output 'amo' and when it will see "amo" it will output "molto" and when it will see "molto" it will output <EOS> which indicates translation is done.

This all happens in one time step.

In RNN we have n time steps to map m in input sequence into n output sequence, but this problem is solved with transformer.

Linear layer to project embeddings to vocab

Inference.

I love you very much

↓

Ti amo molto

Encoder input - <SOS> I love you very much <EOS>

Decoder input: <SOS> and with Paddings

logits: output of last layer is known as logits (linear layer)

↳ linear layer to project embeddings back to our vocab. This projection is called logits.

logits will go to Softmax

After softmax it will produce 'Ti'

that would be the step = 1.

Training happen in one pass, while inference we'll do it token by token and we'll also see what's the cost

At time step = 2, we won't calculate encoder output again because our english sentence didn't change

for T = 2,

decoder input = "<SOS> Io"

at end we'll get next token "amo" is appended.

for T = 3,

decoder input: "<SOS> Io amo"

end output = "<EOS> molto"

for T = 4,

decoder input: "<SOS> Io amo molto"

end output = <EOS>

It twice 4 time step

→ Inference strategy.

→ We selected, at every step, the word with most optimum value. | Greedy search

→ A better strategy is to select i.e. each level step the top B ~~levels~~ words and evaluate all the possible next words for each of them and at each step, keeping the top B most probable sequences.

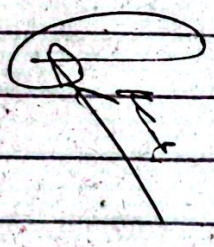
↑ Beam Search

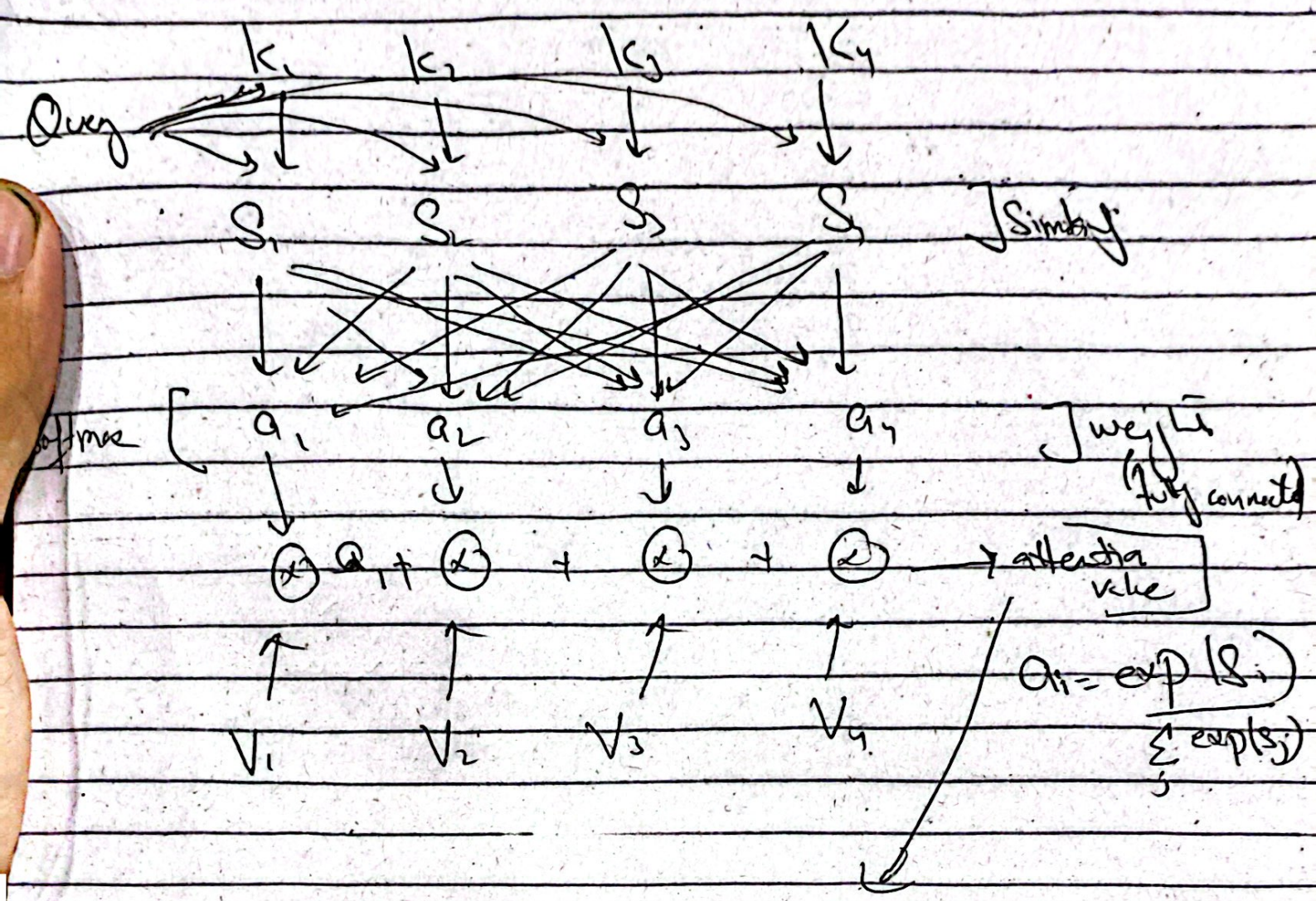
[wait]

$$\text{Similarity} = f(Q, K) = \begin{cases} Q^T K & \text{- dot product} \\ \frac{Q^T K}{\sqrt{d_k}} & \text{- Scaled dot product} \\ Q^T W K & \text{- General dot product} \\ W_Q^T Q + W_K^T K & \text{- additive similarity} \end{cases}$$

$Q^T W K$:

projects query Q into new space using weight matrix W and then taking dot product with K





attention value = $\sum_i a_i V_i$

s_i in $q^T W k$, we don't specify W. There are values that get optimized by NN itself. s_i, W will be adjusted by back propagation so that NN can learn on its own what might be a good space to project Q into

K_i, V_i are vectors

S_i, a_i are scalars

without positional encoding (transformer), it will treat words like sequence (Bag of words)

Multi-head attention

by we feed in a vector (input) then multihead will compute attention b/w every position and every other position. So we have vectors that embed words in each one of these position and now, we simply carry out an attention, computation where we treat each word as a query, and find some keys that correspond to other words in the sentence and then take a convex combination of the corresponding values. They values V are going to be same as key, then take dot product of that to produce better embeddings

So idea is that this attention will pick each word, combine it with some of other words through the attention mechanism to essentially produce a better embedding that maps together into from pairs of words

So, this encoding block will repeat Nx times meaning 6 blocks of block.

In first block, will look at pairs and in next pair of pairs and so on.

So, we are combining group of words.

→ Multi-head attn → Add & Norm → repeat.

Add & Norm

encoder output

↳ will be our embedding per position. It captures the original word but also info from other words that it attended to in the network. & it can be thought of a large embedding of all those words corresponding to each position

Decoder

Masked Multihead

b. to combines output words with previous output words.

Multi-head Attention

compute multiple attentions per query with different weights.

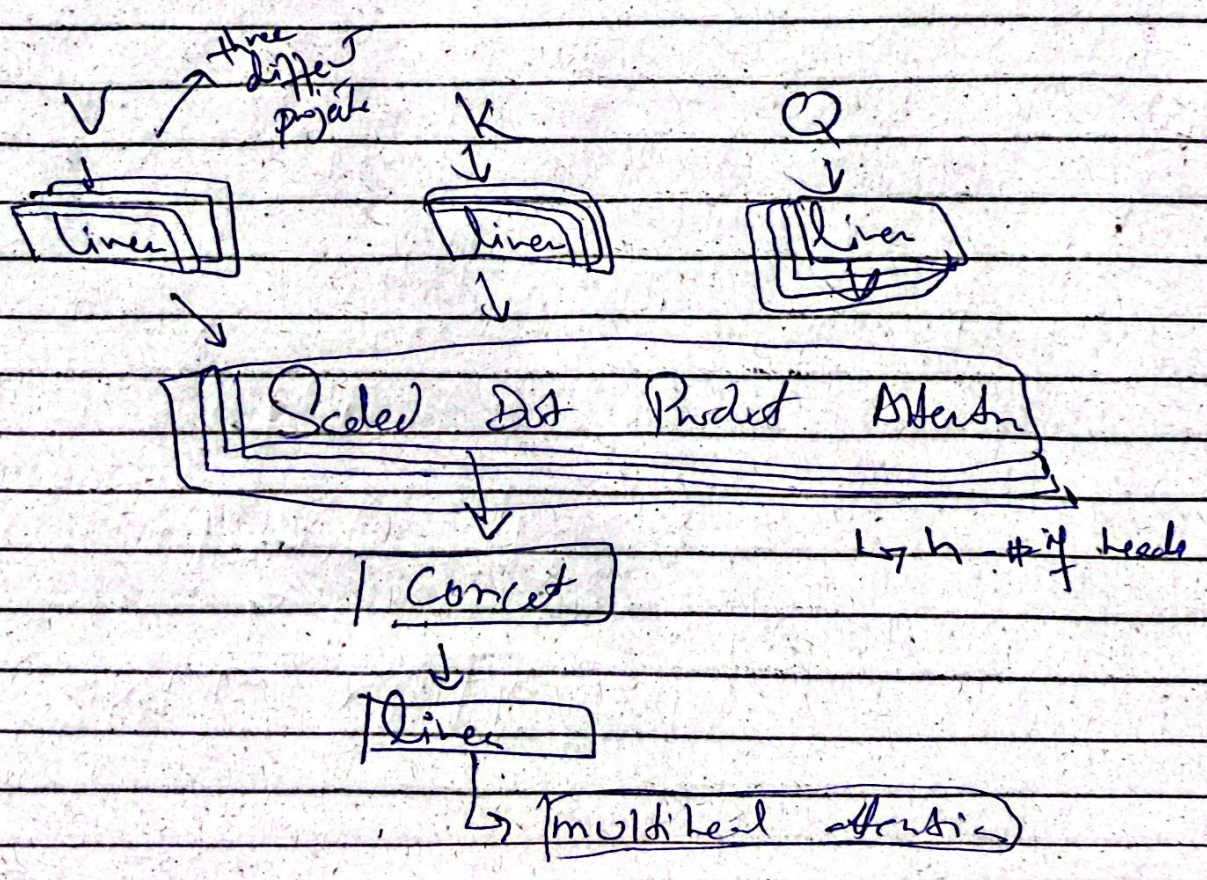
$$\text{multihead}(Q,K,V) = W^{O}\,\text{concat}(h_1, h_2, \dots h_n)$$

$$\text{head}_i = \text{attention}(W_i^{Q}Q, W_i^{K}K, W_i^{V}V)$$

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

The general way of thinking about attention is that we have some key-value pairs, then their is query that going to compare to each key and the keys with greater similarity are going to have higher weights and then, we can take weighted combination of corresponding values to produce the output.

So, we are going to feed K, V, Q into some linear layer, then we'll calculate scaled dot product attention.



masked Attention $(Q, K, V) = \text{Softmax}\left(\frac{Q^T K + M}{\sqrt{d_k}}\right) V$

↳ output value should not depend on future values.

before Softmax, put value $= -\infty$

Training

Teacher forcing: when you train an NN, you have both input sentence and output sentence/label and then you assume that my outputs are correct everywhere so I'm gonna feed that as decoder input, so I'm gonna feed correct output words are previous positions and then I'll simply try to predict next word based on that.

GPT

Jacobian

derivative = slope
↳ can work for $f(x)$ but not for $f(x,y)$

so derivative ≠ slope

On a similar note, integrals are height ↑ as area under the curve

$$\text{Area} = \int_a^b f(x)\,dx$$

Lets rethink derivative & integrals

Linear maps :

linear maps (from $\mathbb{R}^2 \to \mathbb{R}^2$) are required to have these properties

1. → parallel lines stay parallel
2. → Even spacing persered
3. → Origin is fixed

⇒ Determinant is a scaling factor for area (2D)
↳ length (1D)

Avoid overfitting

- Increase size & quality of your data
- Data Augmentation
- Early Stopping
- Regularization
- Dropout
- Decrease model complexity
- Feature selection
- Ensemble
- Cross Validation
- Transfer learning

Gradient accumulation:

- with .grad = grad.zero()

Avoid underfitting

- Increase model complexity
- increase amount of training data
- Feature engineering
- Increase # of epochs
- Use regularization (reduce regularization strength)
- ~~cross validation~~
- Ensemble learning.