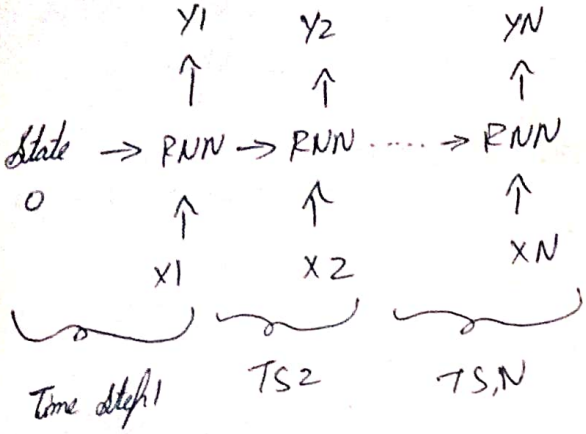
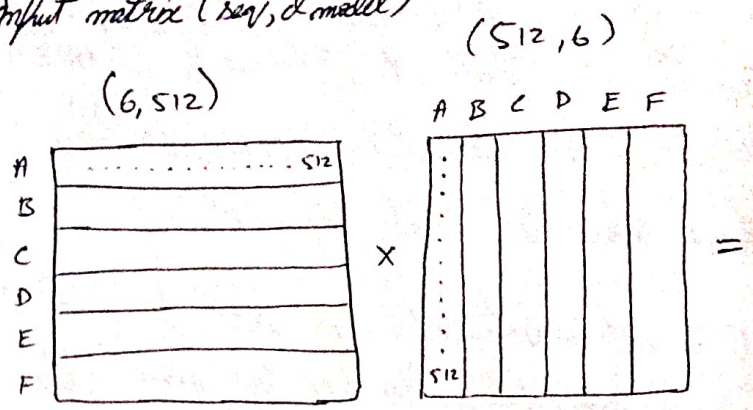
RNN, before transformer, RNN is used to handle the medium sized data for sequential tasking.

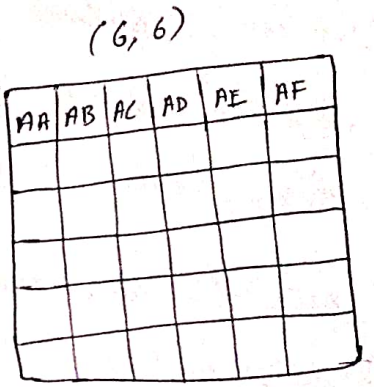


Problems with RNN

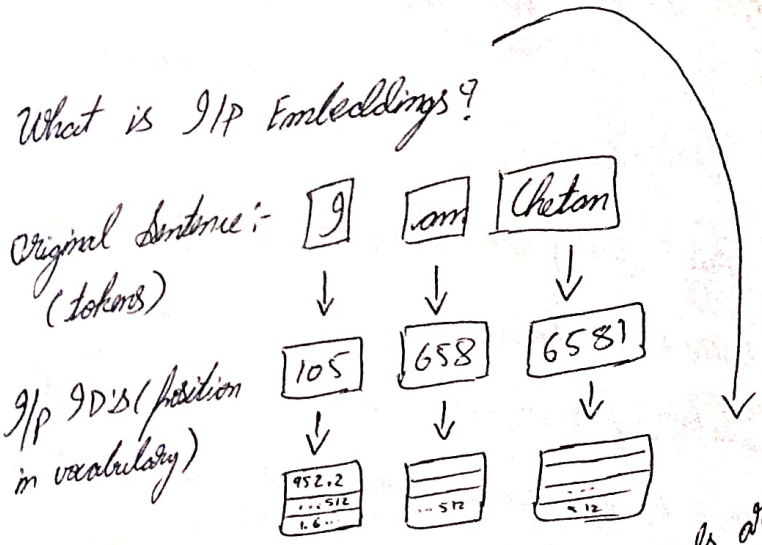
1. Slow computation for long seq.
2. Vanishing or exploding gradients.
   ↳ During backpropagation, gradients become very small (close to 0). & network stop learning.
   → Gradient become very large during bp, where weight updates too aggressively & model becomes unstable.
3. Difficulty in accessing information from long time ago.
   (last token which not highly depend on 1st token due to long chain).

Input matrix (seq, d model)





What is I/P Embeddings?



Embeddings (vector of size 512)

The step where words are converted into vectors that model can understand. The size of vector depends on the embedding dimension we choose.

What is Positional Encoding?

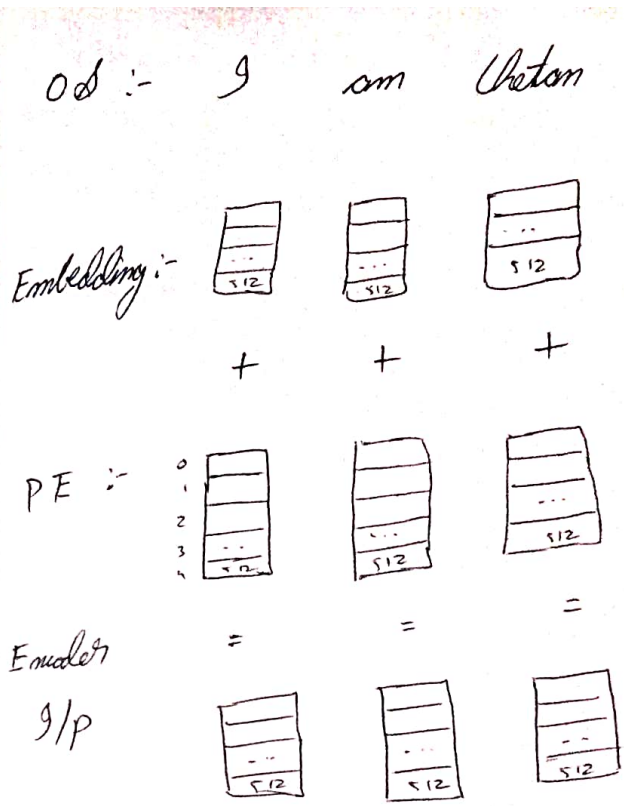
Transformers look at all words at once (not one-by-one like RNN) so they don't know the order of word.

PE is extra information we add to each word's embedding to tell the model the word's position in sentence.

→ Only computed once & reused for every sentence during training & inference.

→ It is calculated using sine/cosine function. even dimension uses sine & odd dimension uses cosine. They make PE more distinct & informative.

→ Only Sin & Cos, Because they give smooth curves & bounded b/w −1 & 1 gives stable patterns for positions. Tan is not used since it has discontinuous values & can jump to ∞.



$$PE(pos, 2i) = \sin \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$

$$PE(pos, 2i+1) = \cos \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$

What is Self-Attention?

→ It allows the model to relate words to each other in sentence, by assigning more weight to the words that are more relevant for understanding the meaning of each word.

Ex:- I am Chetan

* 'I' focus on "am" because they connect.
* "Chetan" give attention to 'I' because it's telling who.

In case, we consider seq length = 6 &

$d_{model} = d_k = 512$

For each word we perform 3 tasks,

* Query (Q) :- What I am looking for?
* Key (K) :- What do I contain?
* Value (V) :- My actual content.

These terms come from the database terminology or Python like dictionaries.

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

$$\text{Softmax}\left(\frac{\underset{(6,512)}{Q} \times \underset{(512,6)}{K^T}}{\sqrt{512}}\right) = \underset{(6,6)}{\square}$$

$$\underset{(6,6)}{\square} \times \underset{(6,512)}{V} = \underset{(6,512)}{\text{Attention}}$$

Each row in this matrix not only captures meaning or position in sentence but also each word's interact with other words.

→ In transformers, softmax is used in attention to decide which words should get more focus.

Softmax function takes list of no's (scores/vectors) & converts them into probabilities.

* All o/p's are b/w 0 & 1.
* The sum of all o/p's is 1.

Multi-Head attention is extension of self attention where multiple self attention operations runs in parallel, each head learns different relationship b/w words.

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

$$\text{Multihead}(Q, K, V) = \text{concat}(\text{head}_1, \ldots, \text{head}_h) W^o$$

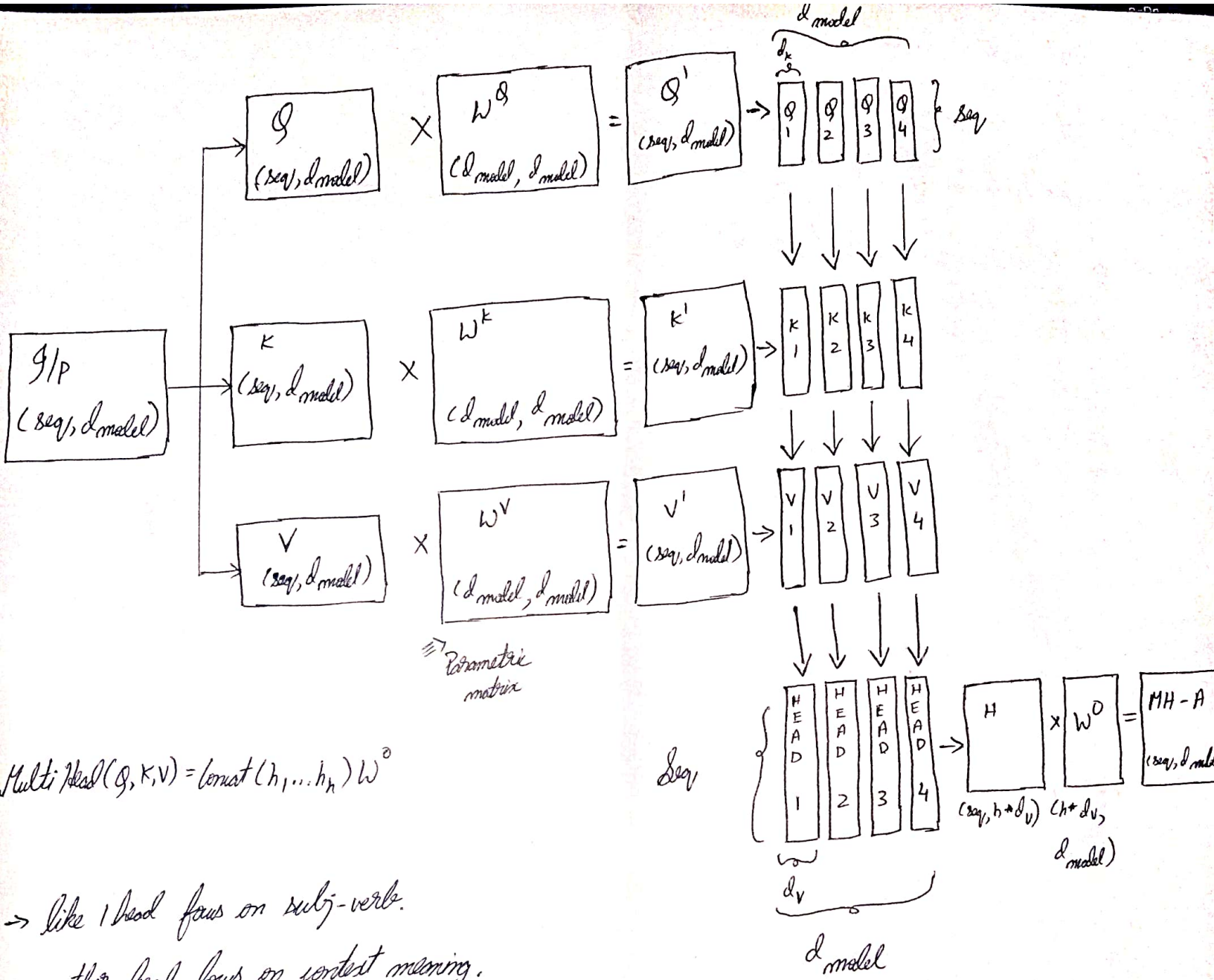
$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

seq = sequence length

d_{model} = size of embedding vector

h = number of heads

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



$$\text{MultiHead}(Q, K, V) = \text{Concat}(h_1, \dots h_h)\, W^O$$

→ like 1 head focus on subj-verb.

→ another head focus on context meaning.

→ another may track word order or position.

Normalization means adjusting values measured on different scales to common scale.

It helps training by keeping all features on a similar scale keeps features balanced, helps in model to learn faster & work better.

Why Normalization in Deep Learning?

* Scaling features helps the model updates weights efficiently, so training is quicker.
* It works well on new, unseen data.
* Keeps gradient stable helps to avoid Vanishing gradient & Exploding gradient problem.

* Normalizing within layers maintains stable feature distribution. (Batch Normalization)

Where to apply normalization in DL?

→ I/p data :- Normalizing features before feeding them into n/w like standard preprocessing.

→ Activations :- We can also normalize o/p of hidden layers to make training faster & more stable, especially in deep n/w's.

<u>Batch Normalization</u> :- Normalizes each features across the batch. Which it keeps mean close to 0 & $\sqrt{S.D}$ (Variance) close to 1.

* Gradients become more stable.
* Reduces overfitting by smoothing activations.

It's Limitations :-

→ The mean & S.D are calculated within current mini batch. When batch become small the mean & S.D are not representative enough.

→ long seq uses small batches, making BN less reliable.

→ Padding issue.

Layer Normalization :- Normalizes the i/p's across features for each individual example, not across the batch.

→ It ensures each example has 0 mean & Unit variance across feature.

Why Transformers use LayerNorm :-

* No dependency on batch size.
* Better for sequential data → each token's features normalized independently.

Mean = 0, centers the data around 0. Ensures the +ve & -ve values balance out.

Variance = 1, scales data so all features have the same range. Prevents features with large values from dominating learning.

By setting these keeps data balanced, makes training faster & more stable.

RMS (Root Mean Square) Normalization :-

They normalizes activations using the root mean square (RMS) of values in layer. It does not subtract the mean like LayerNorm.

→ It includes learnable scaling parameter.

→ But usually no shifting parameter.

RMS Norm is preferred in scenario where computational efficiency matters. Many LLM's uses this, ex :- Deepseek, LLaMa etc due to its efficiency.

Add (Residual Connection), In each encoder / decoder block after attention or FFN the o/p is added back to original i/p of that block.

* Helps the model to skip layer if needed.
* Makes it easier to train very deep n/w's.
* Helps gradients flow better.

Each encoder & decoder layer contains a <u>Feed-Forward neural n/w</u> after attention part.

FNN takes each token's vector, transforms it with a small 2-layer neural n/w.

where <u>1st Linear layer</u> → expands vector → makes it bigger (eg 512 → 2048).

Activation (ReLU / GELU) :- Adds non-linearity.

<u>2nd Linear layer</u> → shrinks it back (2048 → 512).

This makes the representation more expressive & useful for next transform layer.

Decoder in transformer that is responsible for generating the o/p sequence one token at a time.

→ The encoder start by processing the i/p seq. The o/p of top encoder is then transformed into a set of vectors K & V.

→ So these values are used by each decoder in its "encoder-decoder attention."

→ The decoder Queries (Q) are compared with encoder's keys (K). Based on these match, the decoder picks out relevant values (V) from the encoder.

→ These helps the decoder focus on right words from the i/p until a special end symbol tells the decoder when to stop.

→ After each step, the predicted word is sent back into decoder for the next step.

→ Positional Encoding

→ Masked Multi-Head Attention

→ Encoder-Decoder Attention (cross attention).

The Linear + Softmax is applied at every decoding step.

A Linear layer is a simple neural n/w layer that performs a linear transformation of its i/p. Where as, the abstract decoder o/p that converts vector into a vocabulary sized score vector called logits. Then, softmax makes it a probability distribution so model can pick the most likely word.

Softmax is a mathematical function that takes a list of raw scores & converts them into probabilities.

What is Masked multi head attention (Decoder)?

→ It is a mechanism in decoder that allows the model to attend previous tokens but not future tokens in o/p sentence.

Without masking the decoder will see the entire target sentence at once making training unrealistic.