Stanford CS224N - I

* Issues with RNN's

- RNNs unroll L → R / R → L
- This induces linear locality i.e Nearby word affect each other's meaning
- RNN's are sequential thus, they take O(seq len) steps to ~~in~~ for distant words to interact.
- Forward & backward passes have O(seq len) of unparallelizable operations.
- This is becoz future RNN hidden states cant be computed without computing all the previous states.

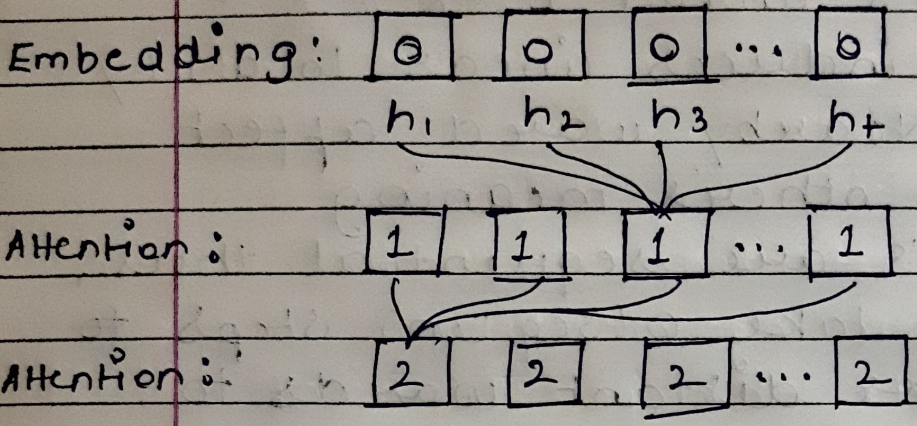
* How does attention treats this?

Attention for a single sentence-

It treats each word rep. as a query to access.

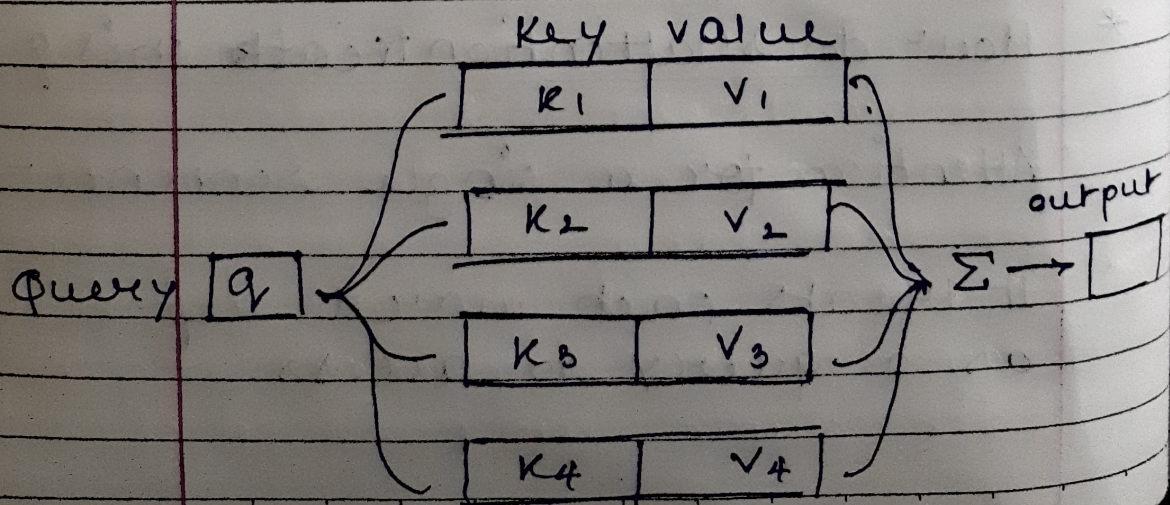
→ No. of 'unparallelizable operations' do not increase with seq len

The nos. in boxes rep the no. of unparallelizable operations



* Query, Key & Value

In attention, the query matches all the keys softly to weight between 0 & 1. The key's 'values' are multiplied by weights & summed.



* Self-Attention

Let $w_{1:N}$ be a seq of words in vocab V, like Zuko made his uncle tea.

For each w_i, $x_i = Ew_i$ where E is an embedding matrix

1. Transform each word emb with weight matrices K, Q, V

$q_i = Qx_i$ query
$k_i = Kx_i$ key
$v_i = Vx_i$ value

2. Compute pairwise similarity btw keys and queries; normalize with softmax.

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

This basically means, how much i should look at j.

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

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

* Sequence order

Self attention doesn't build in order information, so we encode the position of words in our keys, queries & values

We add p_i to our input of embedding matrix x_i. Thus, the 'positional' encoding becomes:

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

* Learned absolute position rep:

We let all p_i to be learnable parameters. This way each position gets to be learned to fit the data, so most systems use this. But it can't handle beyond the sequence length Each column rep position of word

it has seen during training. This is why models have a input token limit.

* Non-linearity in self attention

When we add multiple self attention layers, it just re-avg the value vectors and ends up becoming one big self-attention layer.

To fix this, we add a FFN to post-process each value vector. Basically, the FFN processes the result of attention

* Masking future self-attention

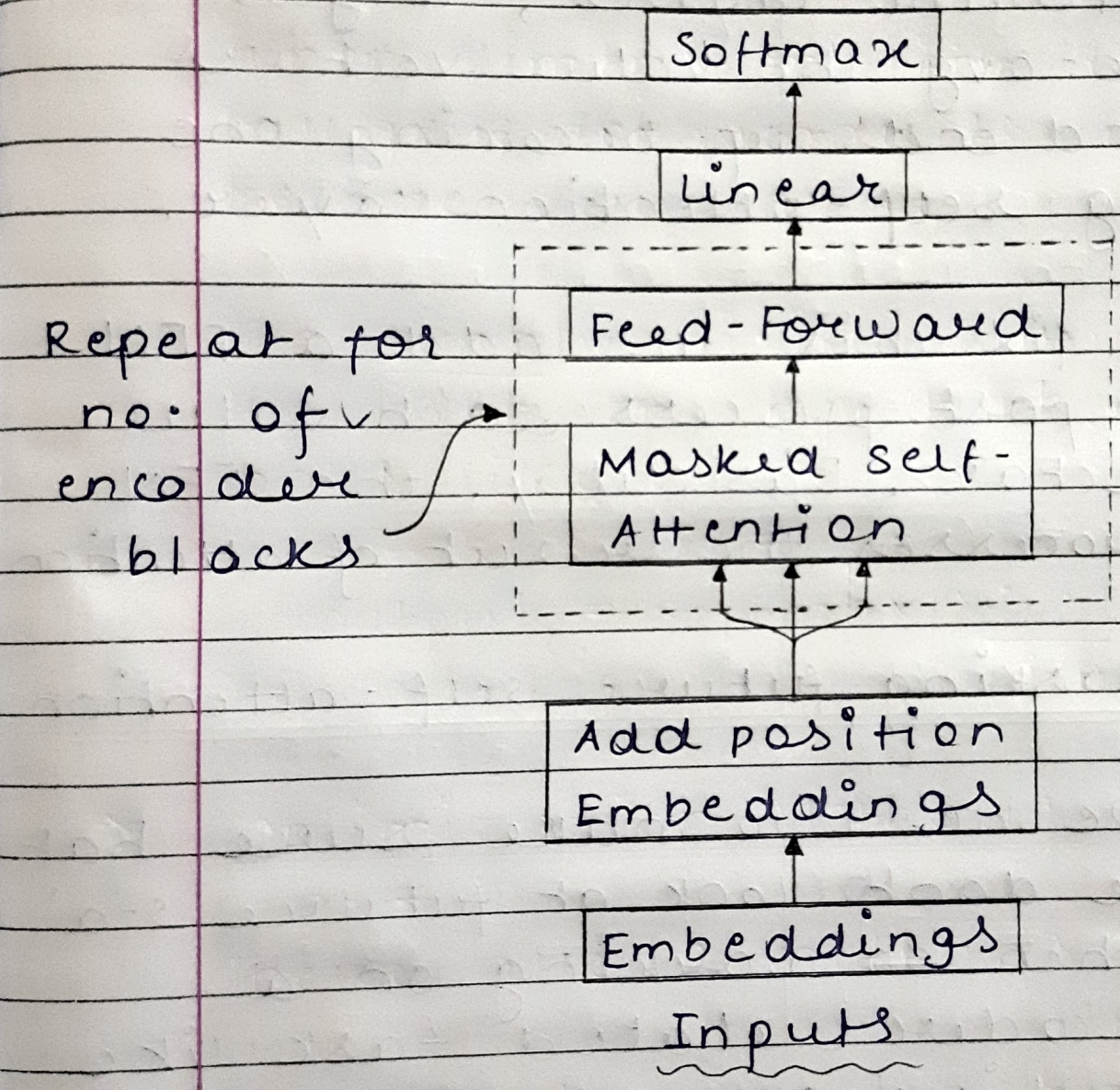
We need to make sure that we dont 'look at future s-a' when ~~do~~ working on a decoder related task like machine translation and language modelling.

To efficiently enable parallelization, we mask out the

attention to future words in this manner:

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

* **Self-Attention building block**



* Multi-Head attention

This contains multiple attention heads tasked with diff types of attention. For ex, 1st head might attend to meaning of the words, 2nd head might check the relevance and so on.

$X = [x_1; \ldots; x_n]$ be concatenation of input vectors.

$$\text{Output} = \text{softmax}(XQ(XK)^T)XV$$

I. We take query-key dot pro in one matrix multiplication $XQ(XK)^T$

II. Softmax above step and matrix multiply with XV to compute weighted average

Let the no. of attention heads be 'h', and 'i' ranges 1 to h. We get $Q_i, K_i, V_i \in R^{d \times d/h}$

We compute $output_i$ for each head and concatenate it in one final output.

$output = [output_1; \cdots; output_h]\, Y$

* **Scaled dot product**

When dimensionality d becomes large, dot products also become very large

We simply divide our ~~output~~ scores $_i$ by $\sqrt{d/h}$ to stop that. It looks like :

$$output_i = softmax\left(\frac{XQ_i(XK_i)^T}{\sqrt{d/h}}\right) XV_i$$

* **Optimization Methods**

There are 2 methods :

1. **Residual connections**

In deep neural networks, we usually face the problem of vanishing & exploding gradients.

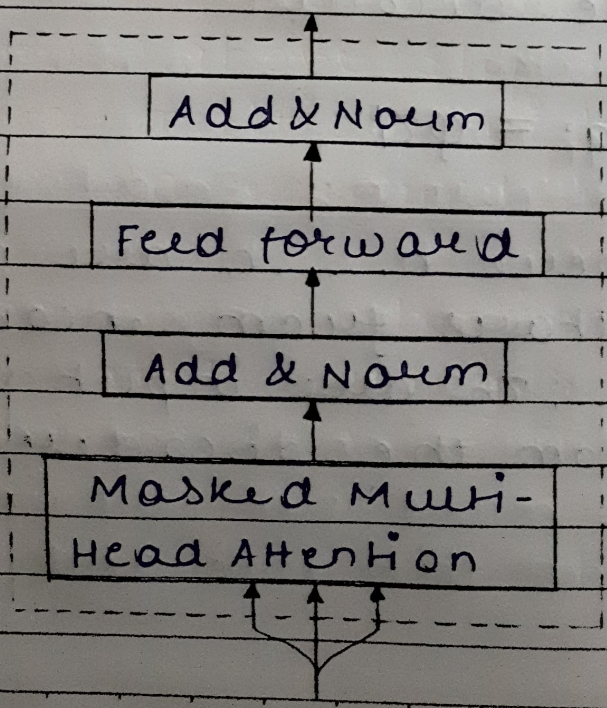
so instead of $x^{(i)} = layer(x^{(i+1)})$
we use $x^{(i)} = x^{(i+1)} + layer(x^{(i+1)})$

This way the model only has to learn how much to change the input to get correct output. This change is 'residue'.

2. Layer Normalization

The basic idea is to normalize gradients in each layer. This is achieved by normalizing the unit mean and standard deviation within each layer.

* The transformer decoder



* Transformer Encoder-Decoder

The encoder architecture is mostly similar to decoder. Main difference is that encoder uses multi-head and not masked multi-head attention because it is better to look at all attentions in input sequence for an encoder.

The keys and values are drawn from the encoder (like memory)

$$k_i = Kh_i, \quad v_i = Vh_i$$

The queries are drawn from the decoder

$$q_i = Qz_i$$

Where $h_1, \ldots, h_n$ are output vectors from encoder and $z_1 \ldots, z_n$ are input vectors from the decoder.