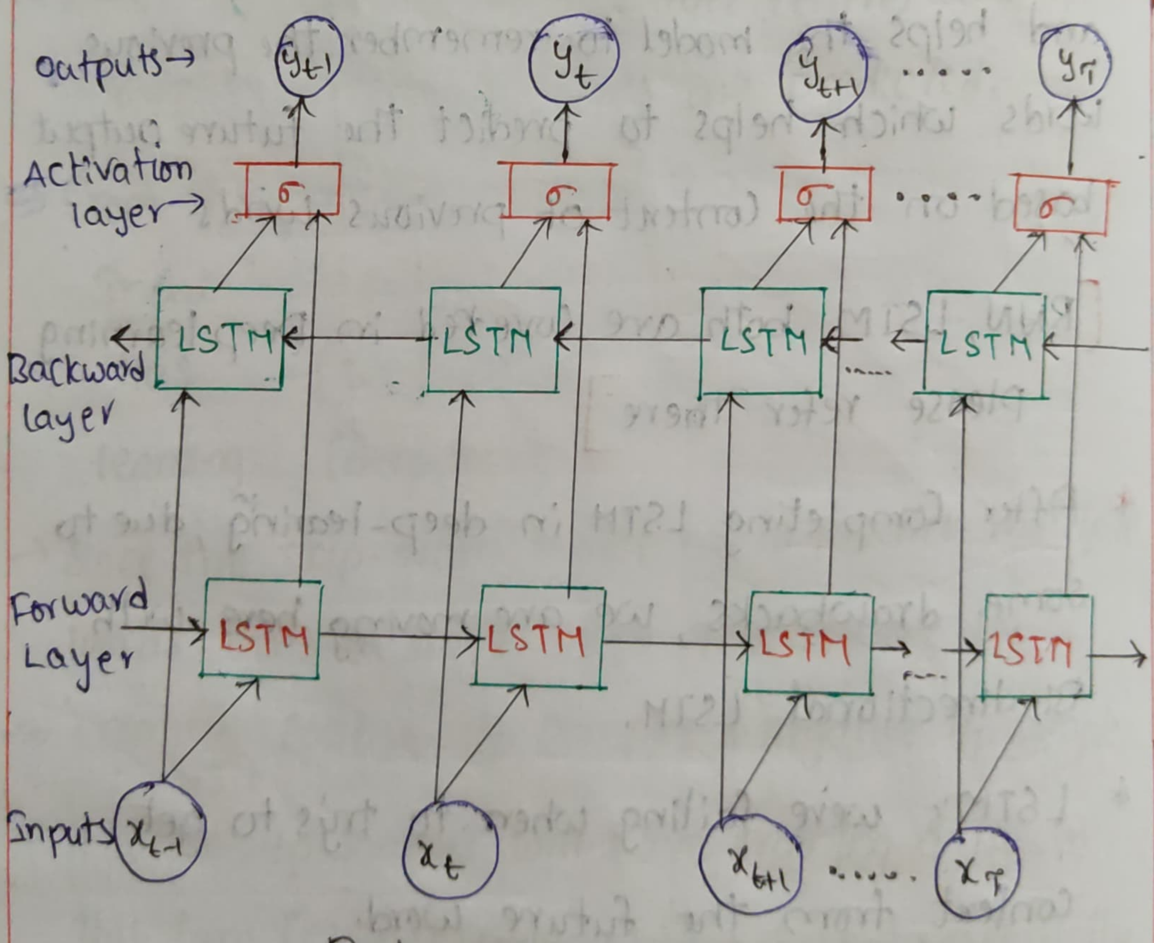
Bidirectional-LSTM:

Bidirectional LSTM networks function by presenting each training sequence forward and backward to two independent LSTM networks, both of which are coupled to the same output layer. This means that the Bi-LSTM contains comprehensive, sequential information about all points before and after each point in a particular sequence.



Bidirectional LSTM

→ In other words, rather than encoding the sequence in the forward direction only, we encode it in the backward direction as well and concatenate the results from both forward and backward LSTM at each time stamp. The encoded representation of each word now understands the words before and after the specific word.

→ lets take an example how Bi-LSTM works,

Ex:- "I will swim today".

In forward LSTM ⇒ t=0 t=1 t=2 t=3

I will swim today

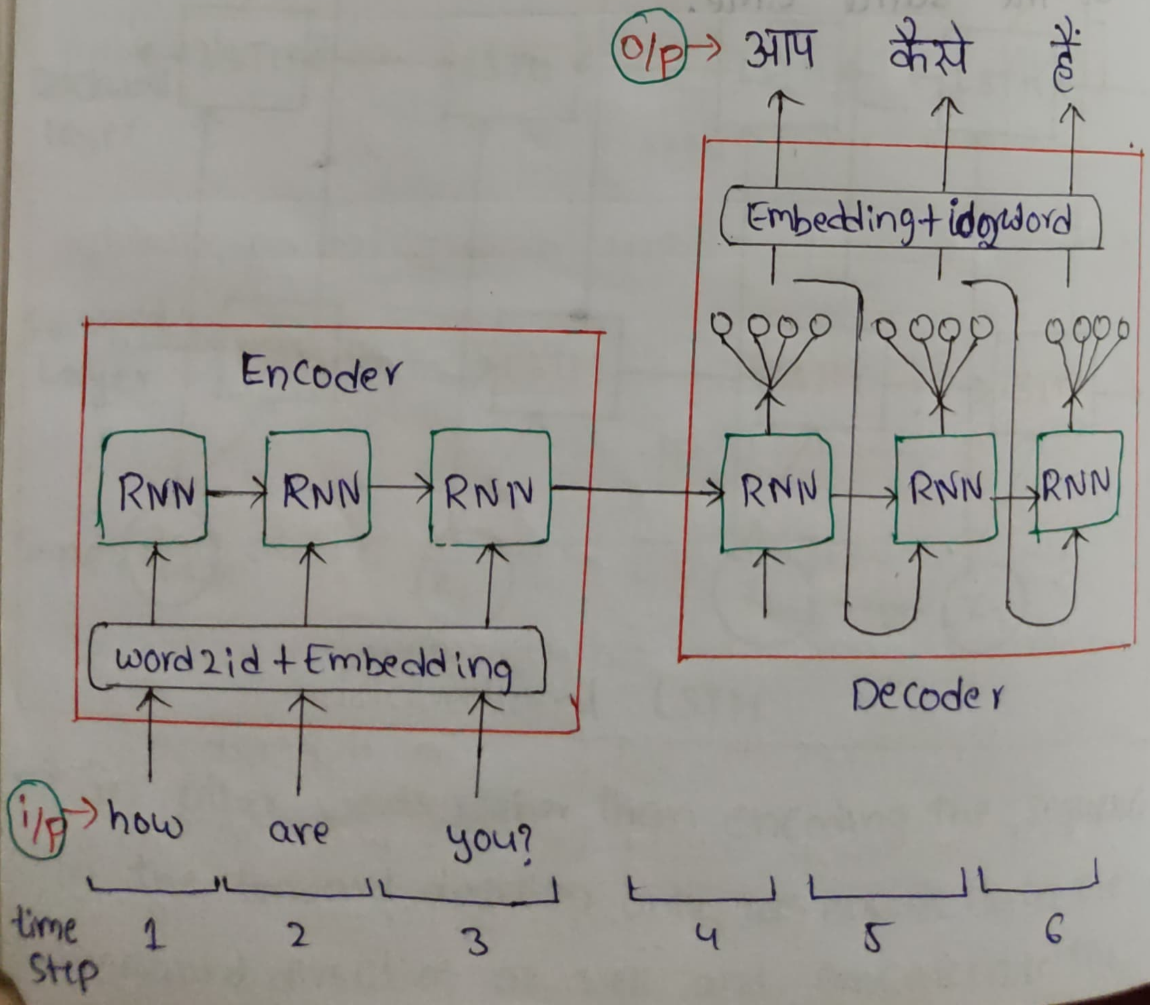
In Backward LSTM ⇒ t=0 t=1 t=2 t=3

today swim will I

→ Both forward and Backward LSTM's will work at the same time.

Sequence-2-Sequence Model

Seq2Seq is a type of model in deep learning that is used for tasks such as machine translation, text summarization, and image captioning. The consists of two main components: an encoder and a decoder. The encoder takes a sequence of input data (such as a sentence in one language) and converts it into a fixed-length representation, called the "context vector". The decoder then uses this context vector to generate a sequence of output data (like sentence translated to other language).



→ The encoder and decoder are typically implemented as RNNs or LSTMs or Transformers. The encoder takes the input sequence, one token at a time and uses Neural network to update its hidden state, which summarizes the information in the i/p sequence. The final hidden state of encoder is then passed as the context vector to the decoder.

→ The decoder uses the context vector and an intial hidden state to generate the output sequence, one token at a time. At each time step, the decoder uses the current hidden state, the context vector and the previous output token to generate a probability distribution over the possible next tokens. The token with highest probability is then chosen as the output, and the process continues until the end of the output sequence is reached.

<u>Drawbacks</u>:-

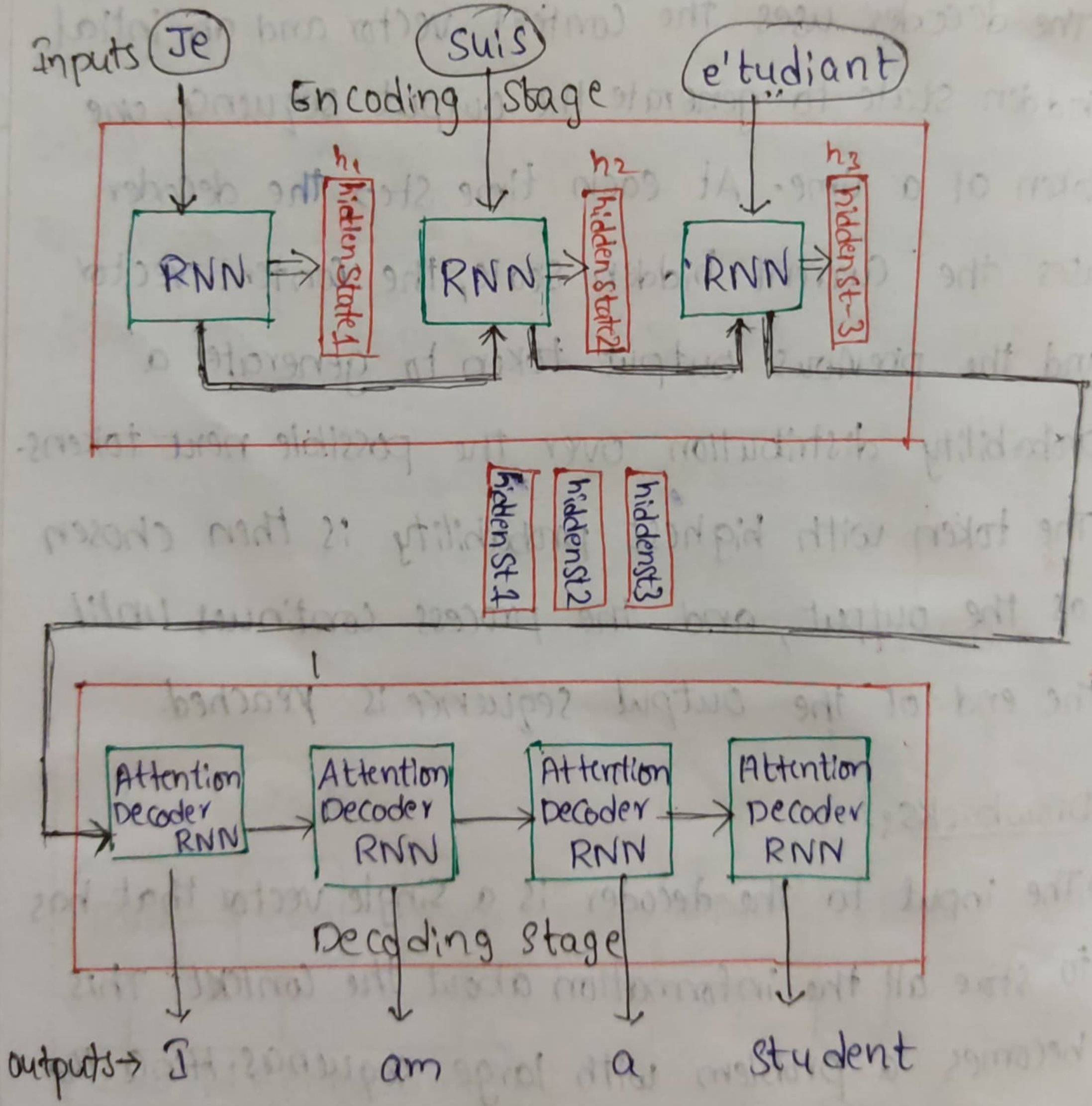
→ The input to the decoder is a single vector that has to store all the information about the context. This becomes a problem with large sequences. Hence the attention mechanism is applied which allows the decoder to look at the input sequence selectively.

→ Difficulty in handling long input sequences.

Seq2Seq with Attention

An Attention model differs from a classic sequence to sequence model in two ways:

1. First, the encoder passes a lot more data to the decoder. Instead of passing the last hidden state of the encoding stage, the encoder passes all the hidden states to the decoder.



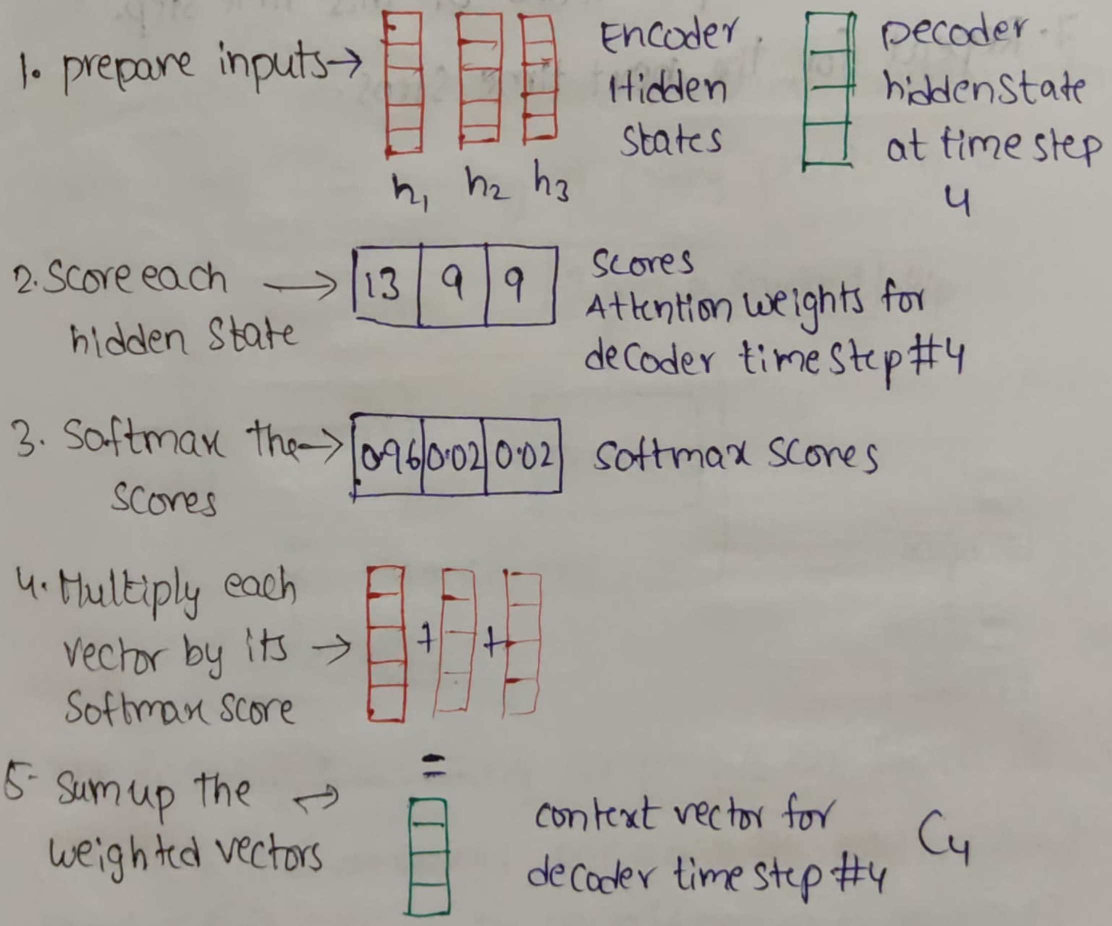
2. Second, an Attention decoder does an extra step before producing its output. In order to focus on the parts of the input that are relevant to this

decoding time step, the decoder does the following.

1. Look at the set of encoder hidden states it received — each encoder hidden state is most associated with a certain word in the input sentence.

2. Give each hidden state a score.

3. Multiply each hidden state by its softmaxed score, thus amplifying hidden states with high scores, and drowning out hidden states with low scores.



→ This scoring exercise is done at each timestep on the decoder side.

→ Let's see the overview of how decoder works.

1. The attention decoder RNN takes in the embedding of the <END> token, and an initial decoder hidden state.

2. The RNN, processes its inputs, producing an output and a new hidden state vector(h_4). The o/p is discarded.

3. Attention step: We use the encoder hidden states and the h_4 vector to calculate a context vector(C_4) for this time step.

4. We concatenate h_4 and C_4 into one vector.

5. We pass this vector through a feedforward neural network (one trained jointly with the model)

6. The output of the feedforward neural networks indicates the output word of this time step.

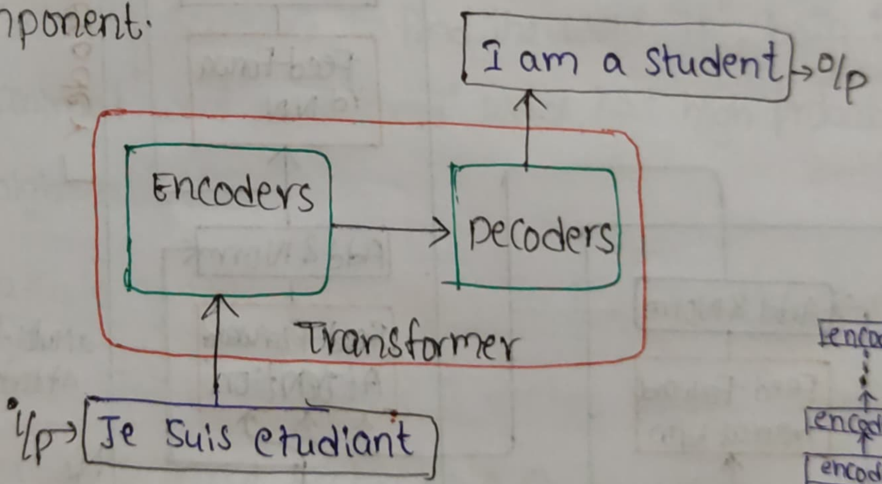
7. Repeat for the next time steps.

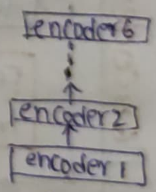
TRANSFORMER

The transformer - a model that uses attention to boost the speed with which the models can be trained. It relies entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution.

→ Transformer aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease.

→ We will try solving the transformer by dividing into components and looking at every process in a component.





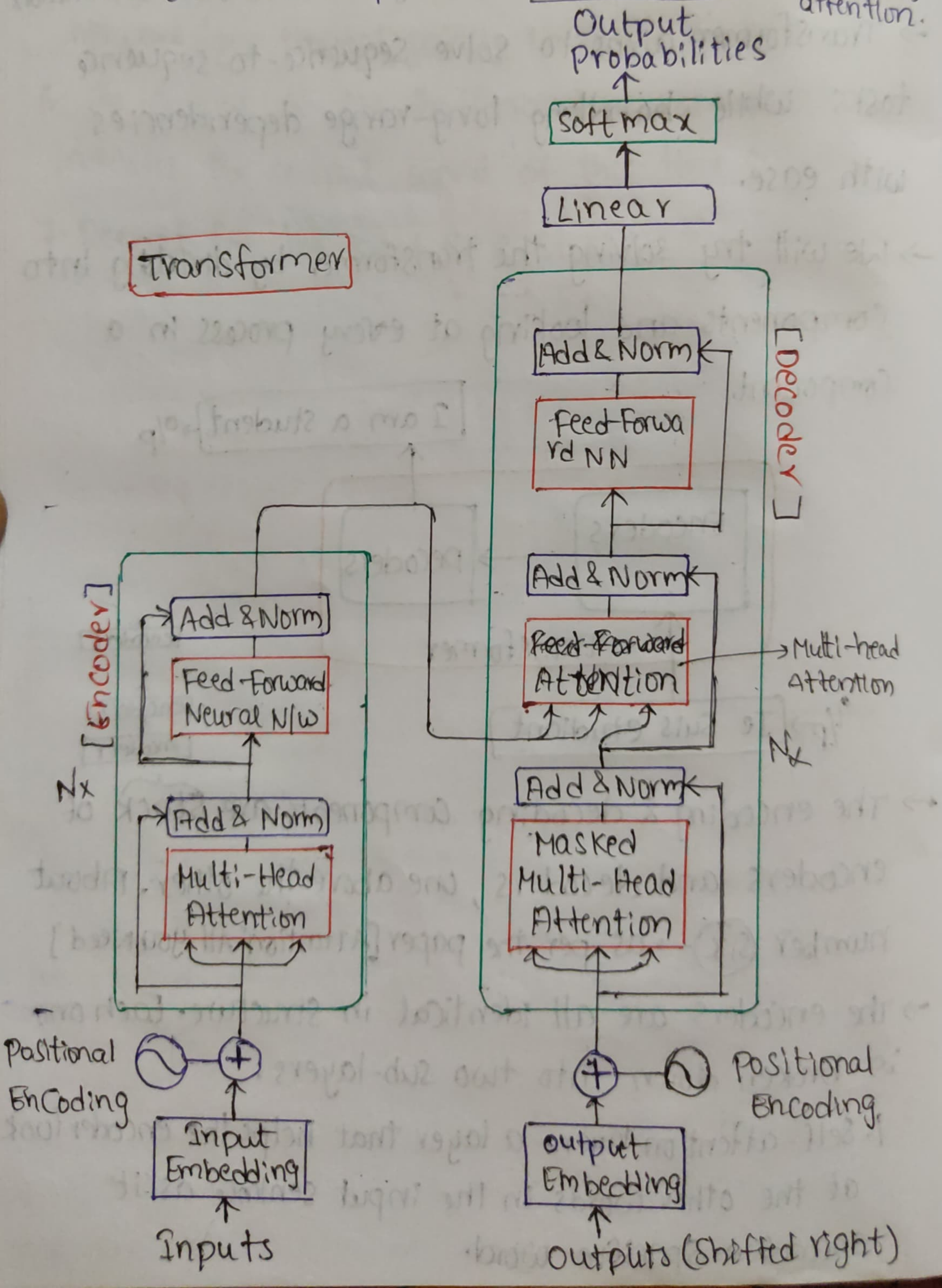
→ The encoding & decoding components are stack of encoders and decoders, one above the other. About number six → as per the paper [Attention All you Need]

→ The encoders are all identical in structure. Each one is broken down into two sub-layers.

1. Self-attention layer - a layer that helps the encoder look at the other words in the input sentence as it encodes a specific word.

2. The output of the self-attention layer are fed to a feed-forward Neural network. The exact same feed-forward network is independently applied to each position.

→ The decoder has both these 2 layers, but between them is an attention layer that helps the decoder focus on relevant parts of input sentence (like Seq2Seq attention).



Steps:-

(1) Convert each input word into a vector using an Embedding algorithm. As per the paper, each word is embedded into vector size 512.

* Each word in the sentence will be passed or flows parallelly throughout the encoder (All words At a time).

→ These vectors will be passed through the self-Attention layer and then to feed-forward Neural Network, then sends out the output to the next encoder.

→ Let's take an example.

"the animal didn't cross the street because it was too tired."

⇒ In this sentence, to find the word "it", both the "animal" word and "tired" word has high probability or high attention.

(2) Calculate Self-Attention to the input vectors

→ 1. First step is to create 3 vectors from each of the encoder's input vectors → a Query vector, a Key vector and a value vector. These vectors are created by multiplying the i/p embedded vectors with weights. The resultant 3 vectors are smaller in dimension (like 64) than Embedding vector.

→ 2. Second step to calculate a score. Let's take 1st word or vector from i/p. The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring.

so, for word #1 the first score would be the dot product of $q1$ and $k1$. The second score would be dot product of $q1$ and $k2$.

Input	"Thinking"	"Machines"
Embedding	x_1	x_2
Queries	q_1	q_2
Keys	k_1	k_2
Values	v_1	v_2
Score	$q_1 \cdot k_1 = 112$	$q_2 \cdot k_2 = 96$
Divide by 8 ($\sqrt{d_k}$)	14	12
Softmax	0.88	0.12
Softmax × Value	v_1	v_2
Sum	z_1	z_2

→3. Divide the scores by 8 (square root of dimension of the key vector $\Rightarrow \sqrt{64} = 8$). This leads to more stable gradients.

→4. Then pass the result through a softmax function. Softmax normalizes the scores so they're all positive and add up to 1.

→5. Multiply each value vector by the softmax score this helps to keep intact the values of the word(s) we want to focus on, and drown out irrelevant words (by multiplying with tiny no's like 0.001)

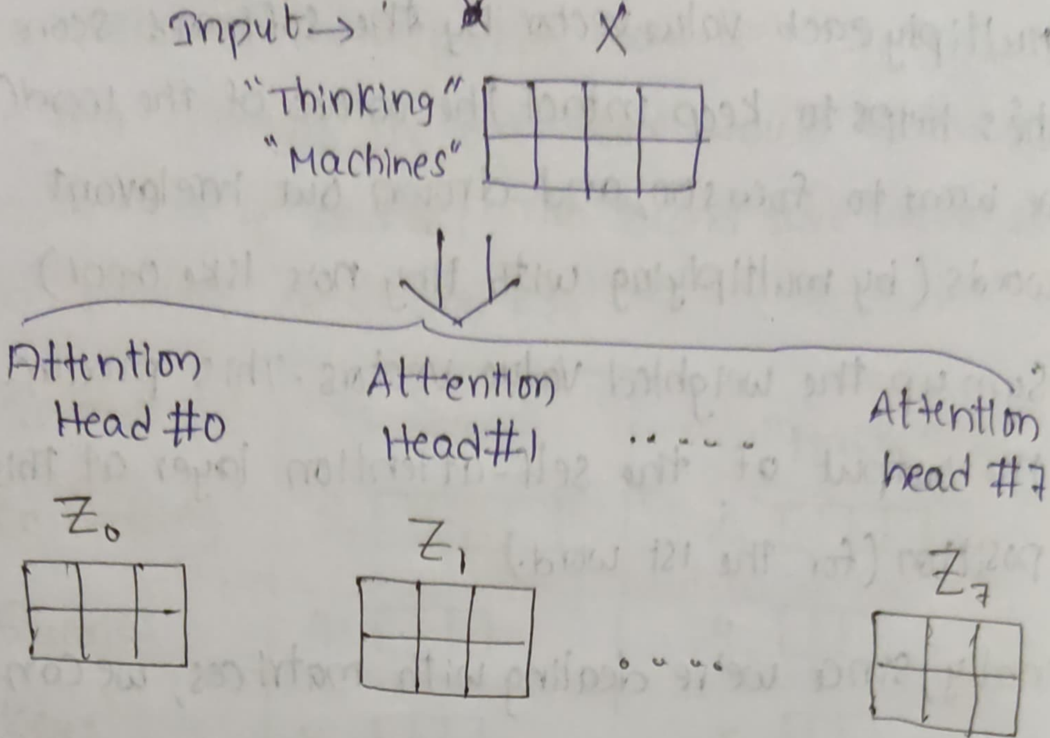
→6. Sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the 1st word.)

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

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

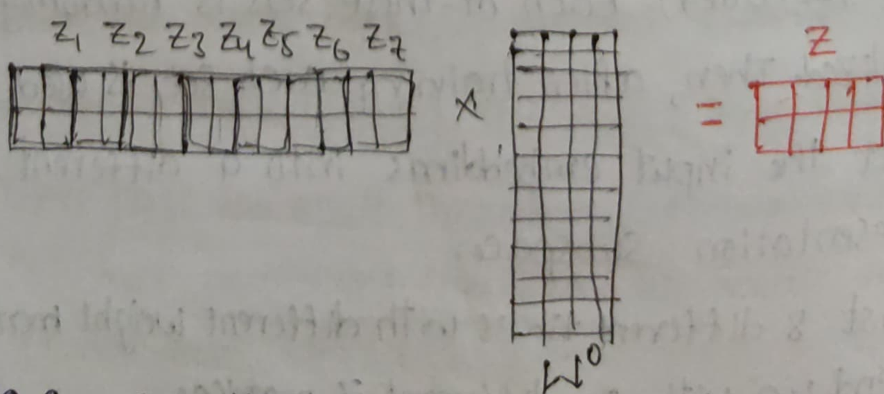
3. Need to calculate Multi-headed attention, since we only calculated single self-attention. For multi-headed we have not only one, but multiple sets of Query/key/value weight matrices (The transformer uses 8 attention heads, so we end up with 8 sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings into a different representation subspace.

→ so just 8 different times with different weight matrices, we end up with 8 different Z matrices.



→ Here we have with 8-Z matrices. then we will forward it to feed-forward Network. But FFNN only expecting a single matrix (vector of one word). So, we need to Convert this 8 matrices → 1 matrix.

4. 1. Concatenate all the attention heads [1 to 8]
2. Multiply with a weight matrix W^O that was trained jointly with the model.
3. The result would be Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



→ We can say this is Multi-Head Attention.

(5). To give the model a sense of the order of the words, we add positional vectors - the values of which follow a specific pattern.

(6.) Residuals - In the architecture of the encoder, each sub-layer (self attention, FFNN) in each encoder has a residual connection around it, and is followed by a layer-normalization step.

→ This 'add & normalization passes through decorders also.

(7) Decoders - Decorders components are similar to encoders. The encoder start by processing the i/p sequence. The o/p of the top encoder is then transformed into a set of attention vectors K & V. These are used by each decoder in its "encoder-decoder" layer which helps the decoder focus on appropriate places in the input sequence.

→ The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its o/p. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like encoders did. And just like we did with the encoder i/ps, we embed and add positional encoding to those decoder i/ps to indicate position of each word.

→ The self-attention layers in the decoder operate in a slightly different way than the one in encoder.

→ In decoder, the self-attention layer is only allowed to attend to earlier positions in the o/p sequence. This is done by masking future positions (setting to $-\infty$) before the softmax step in self-att. calculation.

→ The "Encoder-Decoder Attention" layer works just like multi-headed attention, except it creates Queries matrix from the layer below it and takes the keys and values matrix from the o/p of the encoder stack.

8.) Linear and Softmax layers:-

→ The linear layer is a simply fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

→ The softmax layer than turns those scores into probabilites. The cell with highest probability is chosen, and the word associated with it is produced as the output for this time step.