DATE

Transformers

Attention is all you Need!

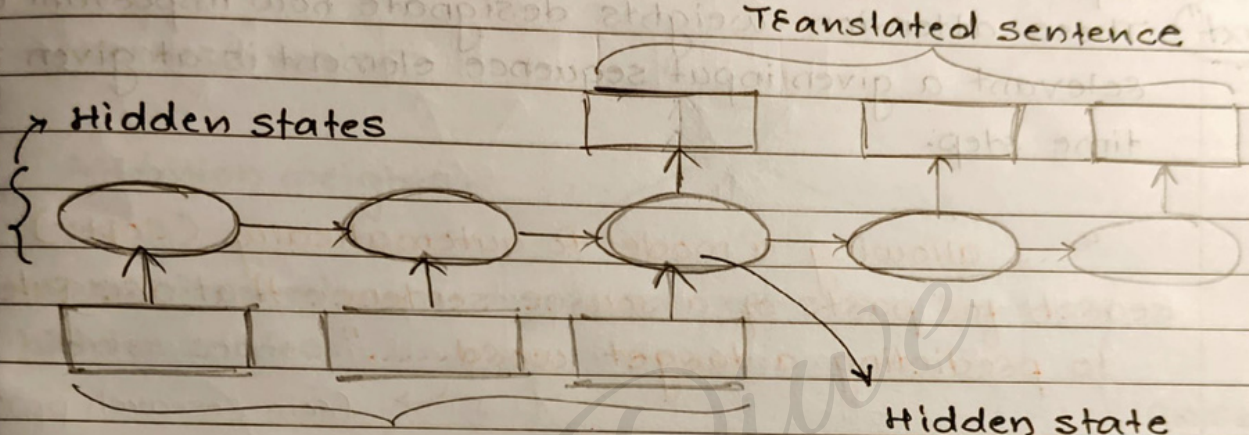


Fig: A traditional RNN encoder-decoder architecture for a seq2seq modelling task.

Limitation: RNN is trying to remember the entire input sequence via one single hidden unit before translating it. Compressing all the information into one hidden unit may cause loss of information, especially for long sequences.

Thus, similar to how humans translate sentences, it may be beneficial to have access to the whole input sequence at each time step.

DATE

Attention Mechanism:

- RNN access all input elements at each given time step.
- It assigns different attention weights to each input element. (i.e. for each word, the network learns a "context") These attention weights designate how important or relevant a given input sequence element is at given time step.

"... allowing a model to automatically (soft-) search for parts of a source sentence that are relevant to predicting a target word....."

→ from research ppr.

The original attention mechanism for RNNs.

Given: an input sequence $x = (x^1, x^2, \ldots, x^T)$, the attention mechanism assigns a weight to each element x^i and helps the model identify which part of the input it should focus on.

DATE

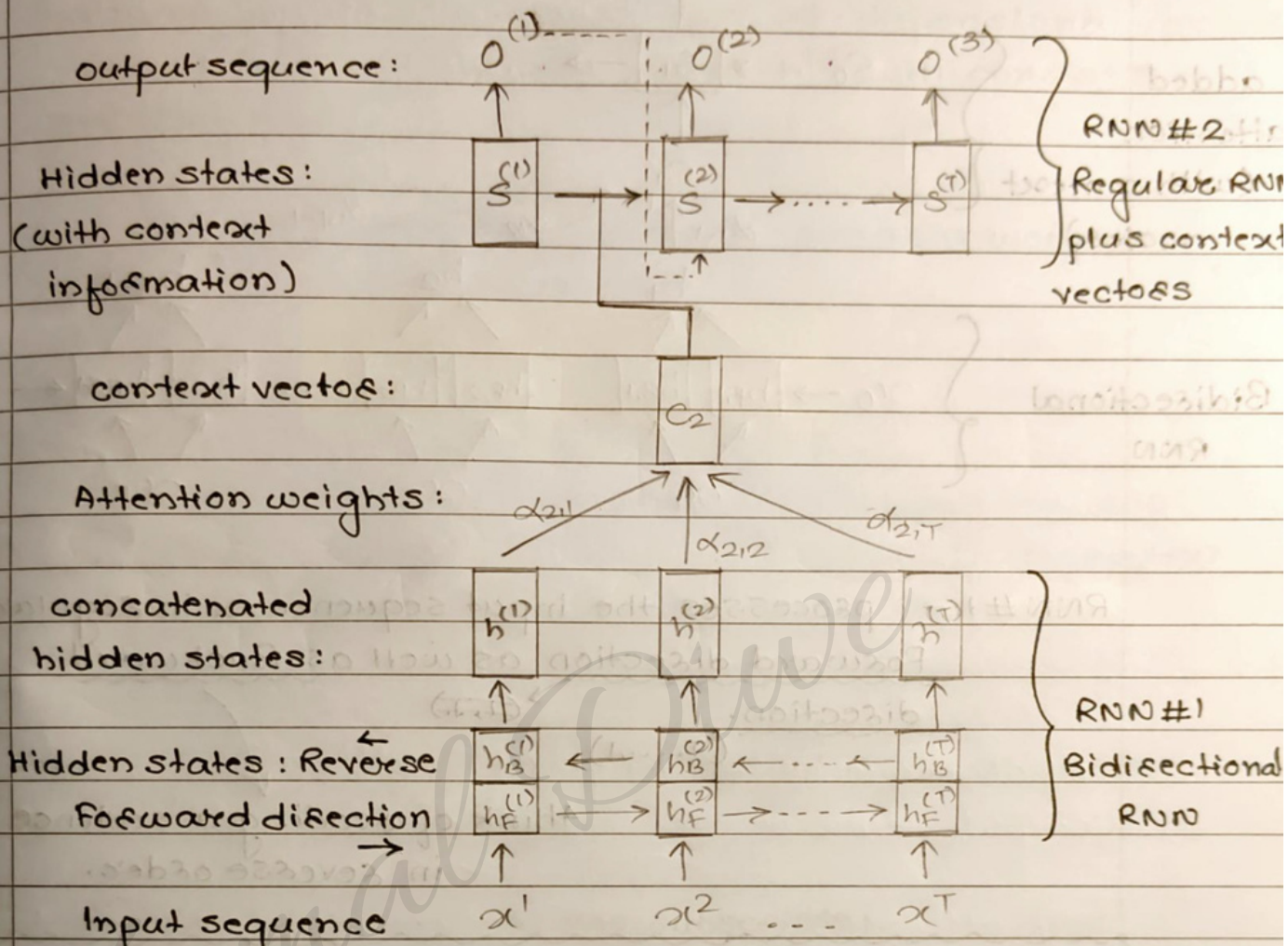
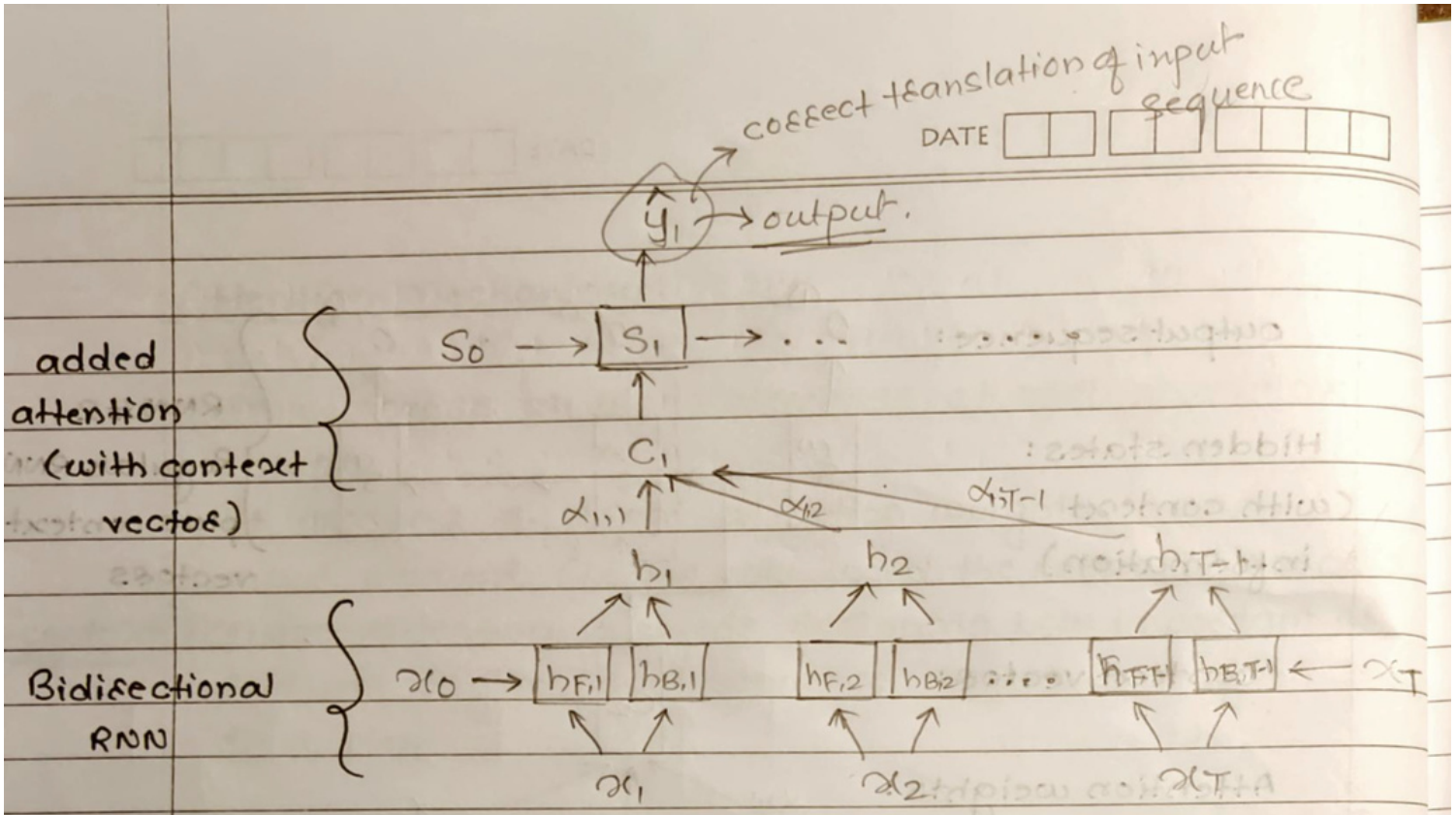


Fig: RNN with attention mechanism

The attention-based architecture consists of two RNN models.

RNN #1 → Bidirectional ~~RNN~~ RNN which generates context vectoes c_i.

RNN # 2 → uses this context vectos, to generate the outputs.



RNN #1 – processes the input sequence x in regular forward direction as well as backward direction. $(T, \ldots, 1)$

think of reading a sentence in reverse order.

The reason behind this is to capture additional information since current inputs may have a dependence on sequence elements that came either before or after it in a sentence, or both.

consequently, reading the input sequence twice (i.e. forward and backward), we have two hidden states for each input sequence element.

DATE

For example, $h_F^{(i)}$ and $h_B^{(i)}$ are 128-dimensional vectors, the concatenated hidden state $h^{(i)}$ will consists of 256 elements.

The context vector of the ith input as a weighted sum:

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h^{(j)}$$

α_{ij} → attention weights over the input sequence $j = 1, \ldots, T$ in the context of the ith input sequence element.

Note: Each ith input sequence element has a unique set of attention weights.

attention weight α_{ij} has two subscripts:
- j refers to the index position of the input
- i corresponds to the output index position.

The attention weight α_{ij} is a normalized version of the alignment score e_{ij}, where the alignment score evaluates how well the input around position j matches with the output at position i.

The attention weight is computed by normalizing the alignment scores as follows:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{K=1}^{T} \exp(e_{ij})}$$

DATE

The transformer architecture also utilizes an attention mechanism, but unlike the attention-based RNN, it solely relies on the self-attention mechanism and does not include the recurrent process found in the RNN.

A transformer model processes the whole input sequence all at once instead of reading and processing the sequence one element at a time.

Self-Attention Mechanism - Basic Form

input sequence → length (T), $x^{(1)}, x^{(2)}, \ldots, x^{(T)}$
output sequence → $z^{(1)}, z^{(2)}, \ldots, z^{(T)}$

self-attention mechanisms are composed of three stages

1. derive attention weights : similarity between current input and all other inputs.
2. Normalize weights via softmax function.
3. compute attention value from normalized weights and corresponding inputs.

DATE

The output of self-attention, $z^{(i)}$ is the weighted sum of all T input sequences, $x^{(j)}$ (where $j = 1, \ldots, T$).

For instance, for the ith input element, the corresponding output value is computed as:

$$z^{(i)} = \sum_{j=1}^{T} \alpha_{ij} x^{(j)}$$

$z^{(i)}$ → as a context-aware embedding vector in input vector $x^{(i)}$ that involves all other input sequence elements weighted by their respective attention weights.

$$z^{(i)} = \sum_{j=1}^{T} \alpha_{ij} x^{(j)}$$

output corresponding to the ith input

weight based on similarity between current input $x^{(i)}$ and all other input.

How to compute the attention weights?

① compute the dot product between the current input element, $x^{(i)}$, and another element in the input sequence, $x^{(j)}$:

$$e_{ij} = x^{(i)T} x^{(j)}$$

repeat this for all inputs $j \in \{1, \ldots, T\}$

DATE

(2) then normalize, via the softmax function, as follows:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j=1}^{T} \exp(e_{ij})}$$

$$= \text{softmax}\left([e_{ij}]_{j=1,\ldots,T} \right)$$

summarize:

The three main steps behind the self-attention operation:

(1) For a given input element, $x^{(i)}$ and each jth element in the set $\{1, \ldots, T\}$, compute the dot product, $x^{(i)T} x^{(j)}$

(2) Obtain the attention weight, α_{ij}, by normalizing the dot products using the softmax function.

(3) Compute the output, $z^{(i)}$, as the weighted sum over the entire input sequence: $z^{(i)} = \sum_{j=1}^{T} \alpha_{ij} x^{(j)}$

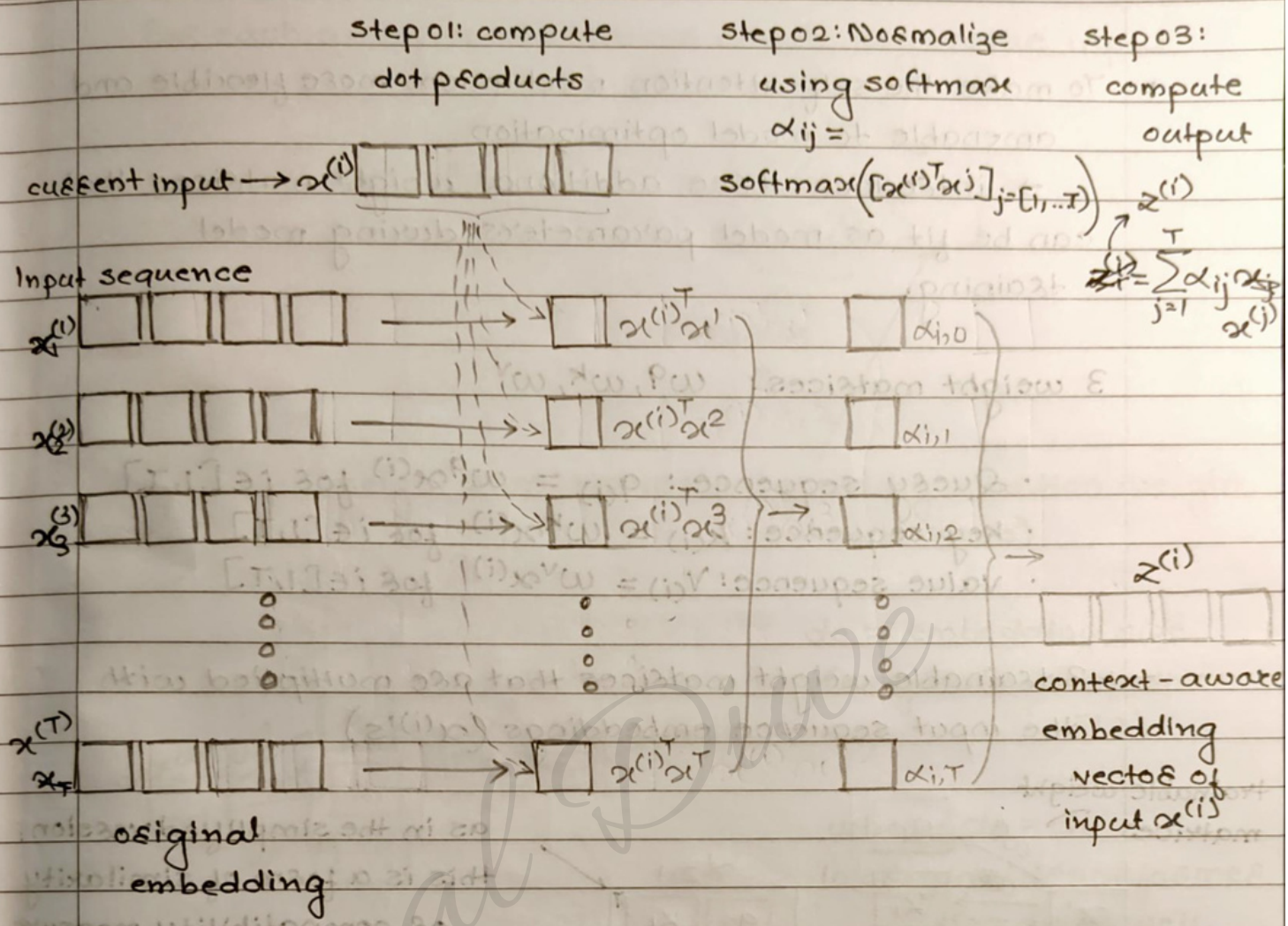


Fig: A basic self-attention process for illustration process

Parameterizing the Self-attention mechanism: Scaled dot-product attention

- Previous basic version did not involve any learnable parameters, so not very useful for learning a language model.

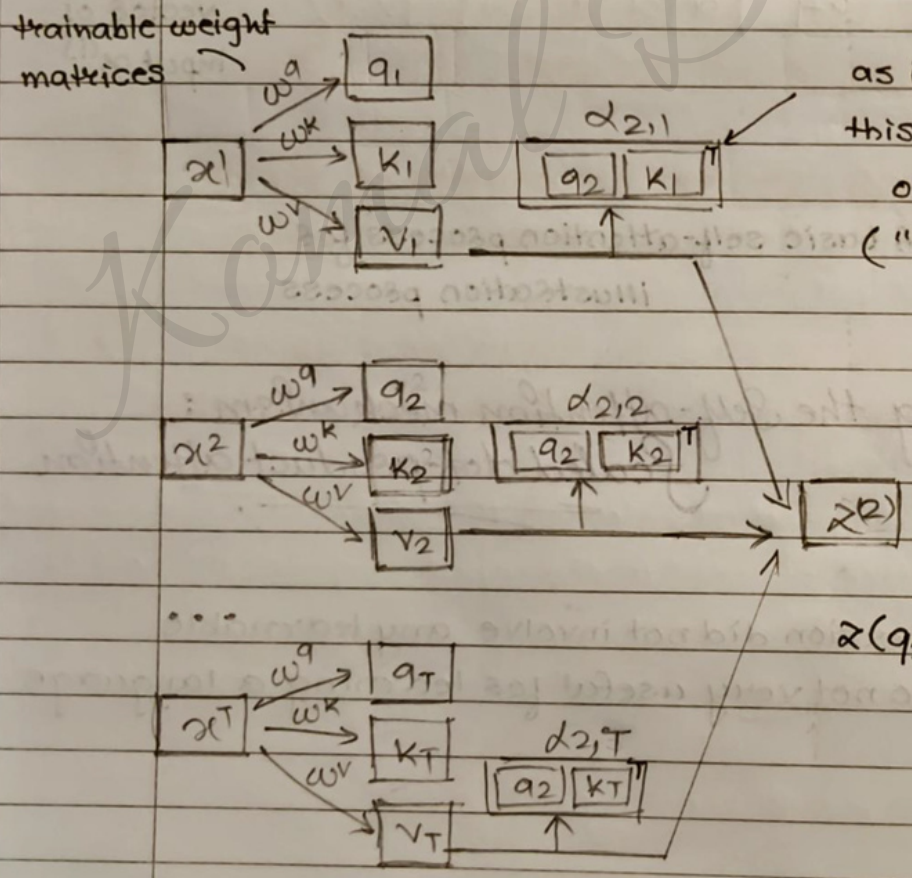
DATE

- To make the self-attention mechanism more flexible and amenable to model optimization
  → introduce three additional weight matrices that can be fit as model parameters during model training.

3 weight matrices: w^q, w^k, w^v

- Query sequence: $q_{(i)} = w^q x^{(i)}$ for $i \in [1, T]$
- key sequence: $k_{(i)} = w^k x^{(i)}$ for $i \in [1, T]$
- value sequence: $v_{(i)} = w^v x^{(i)}$ for $i \in [1, T]$

→ 3 trainable weight matrices that are multiplied with the input sequence embeddings ($x^{(i)}$'s)



as in the simplified version, this is a form of similarity or compatibility measure ("multiplicative attention")

$$z(q_2, K, V) = \sum_{i=1}^{T} \left[\frac{\exp(q_2 \cdot k_i^T)}{\sum \exp(q_2 \cdot k_j^T)} \times v_i \right]$$

classmate

PAGE

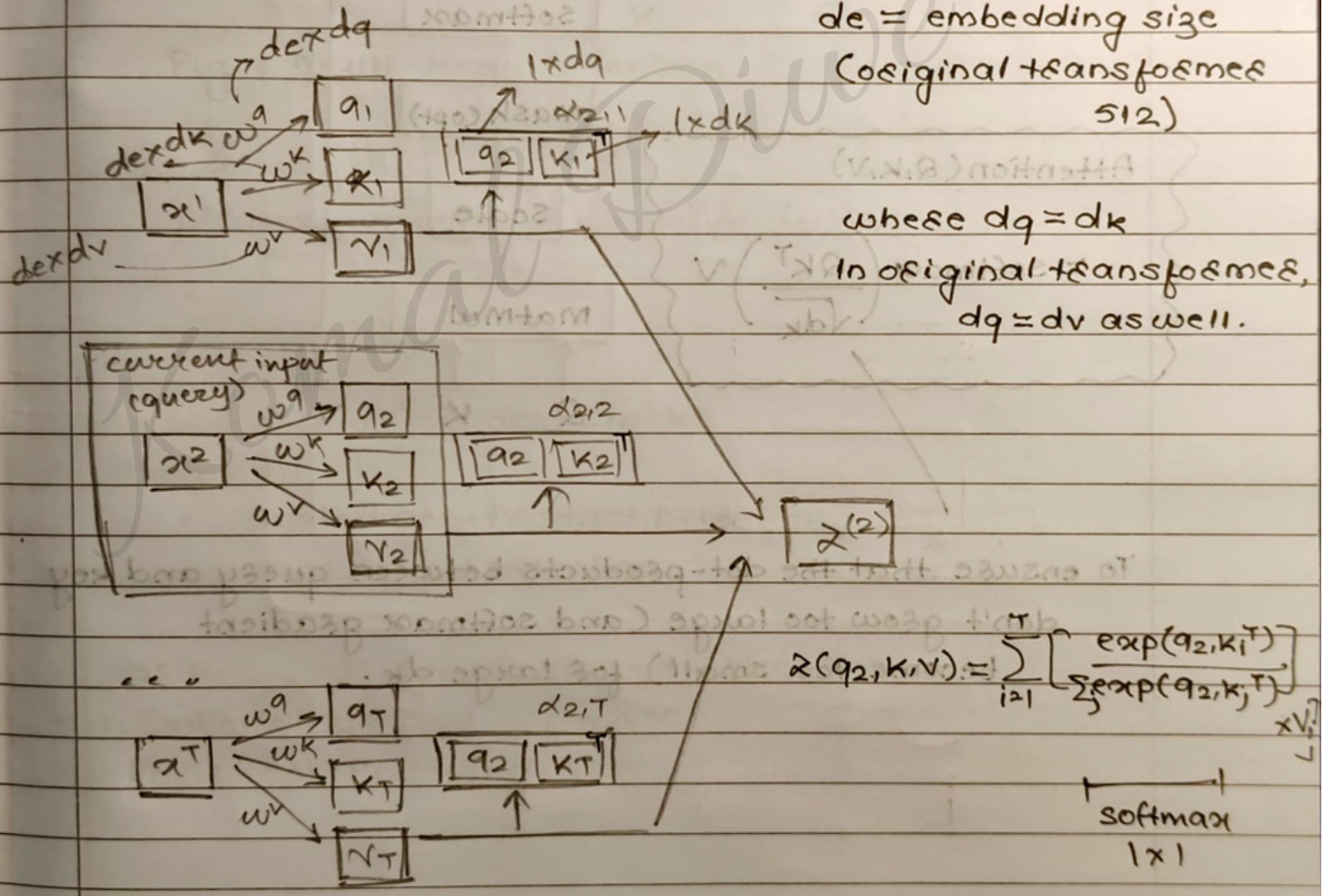
Komal Dive

DATE

For each query, model learns which key-value input it should attend to

$$z(q_2, K, V) = \sum_{i=1}^{T} \left[\underbrace{\frac{\exp(q_2 \cdot k_2^T)}{\sum_j \exp(q_2 \cdot k_j^T)}}_{\text{softmax}} \times v_i \right]$$

weight sum: values weighted by attention weight (softmax score)



de = embedding size (original transformer 512)

where dq = dk

In original transformer, dq = dv as well.

classmate

PAGE

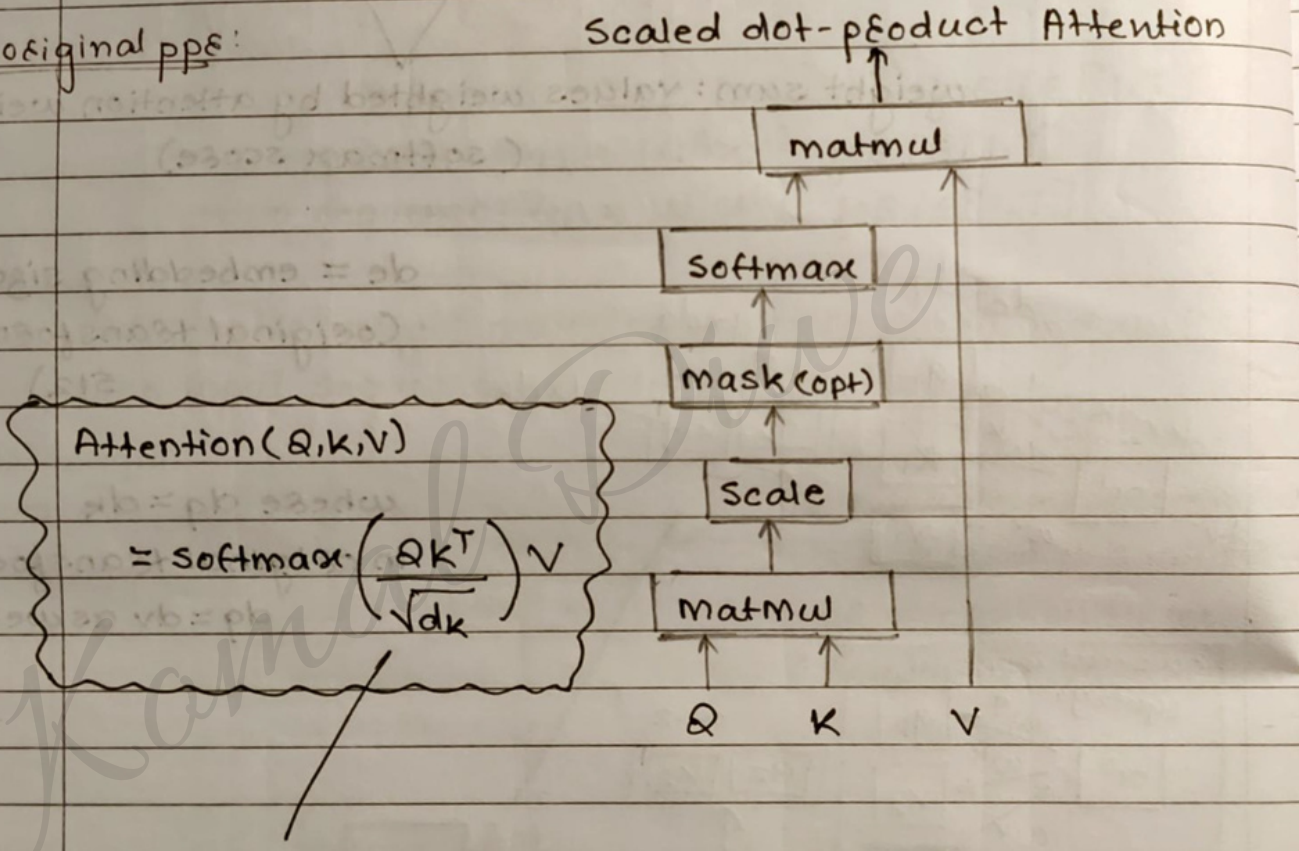
Komal Dive

Query, key and value terminology:

→ are inspired by information retrieval systems and databases.

Eg: if we enter a query, it is matched against the key values for which certain values are retrieved.

For original ppr:



To ensure that the dot-products between query and key don't grow too large (and softmax gradient become too small) for large d_k.

DATE

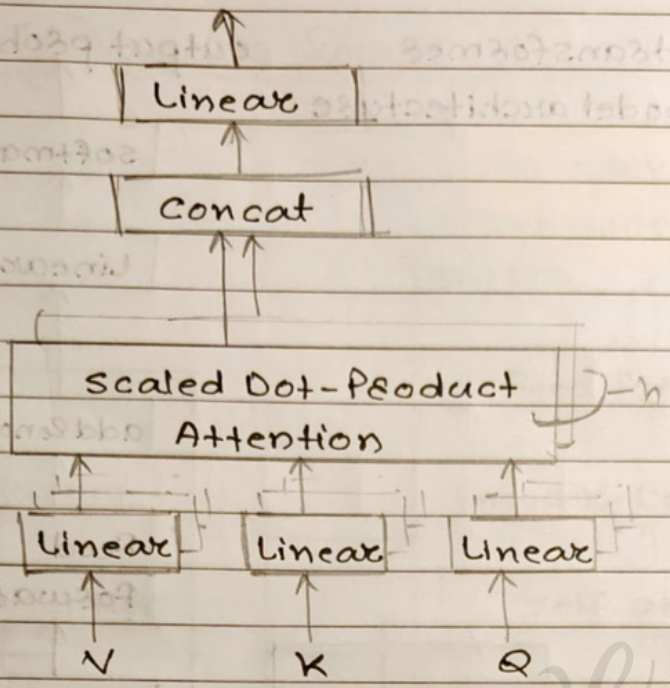


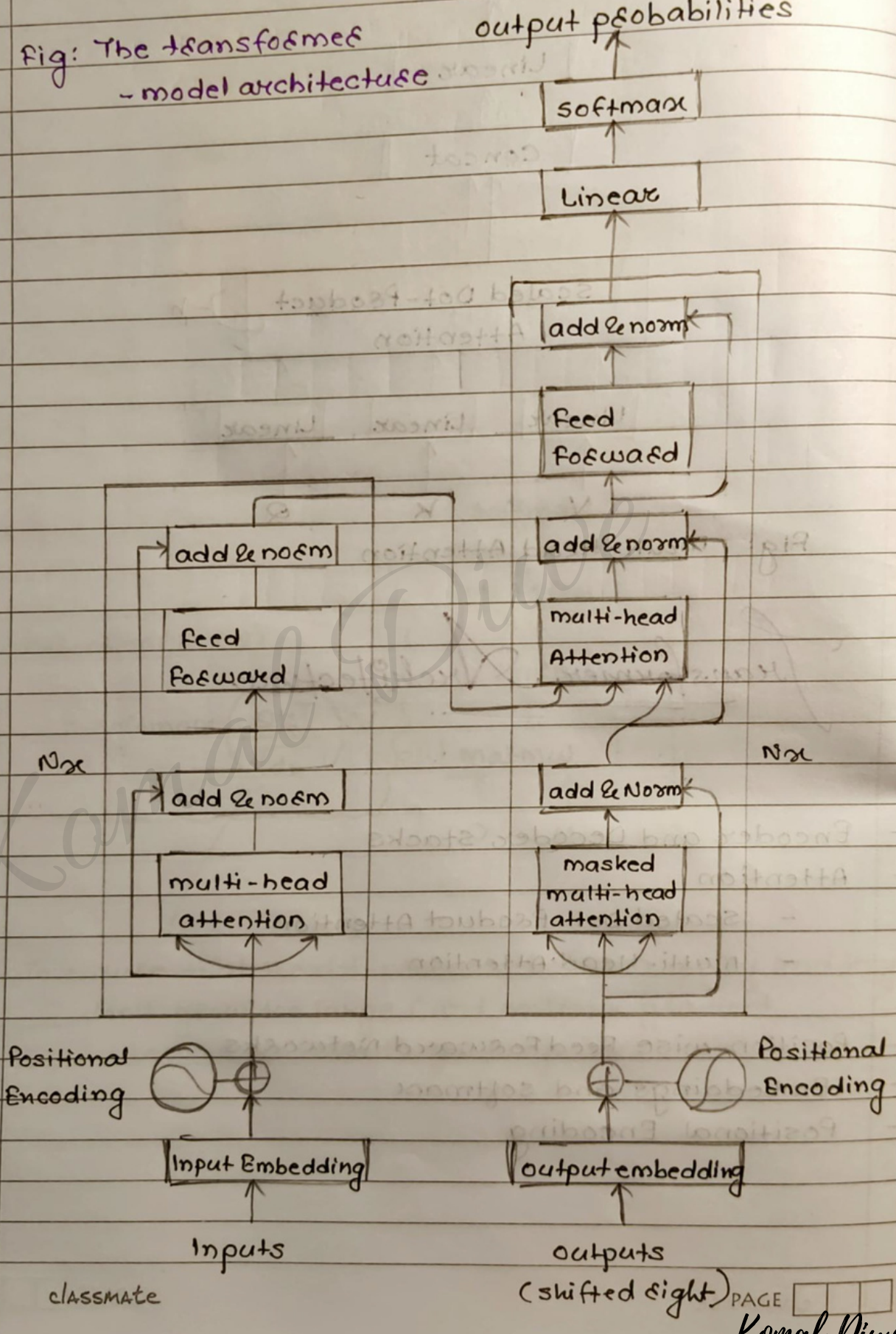
Fig: Multi-Head Attention

Transformer Architecture

- Encoder and Decoder Stacks
- Attention
  - Scaled Dot-Product Attention
  - Multi-Head Attention
- Position-wise Feed Forward Networks
- Embeddings and softmax
- Positional Encoding

DATE

Fig: The transformer - model architecture



classmate

PAGE

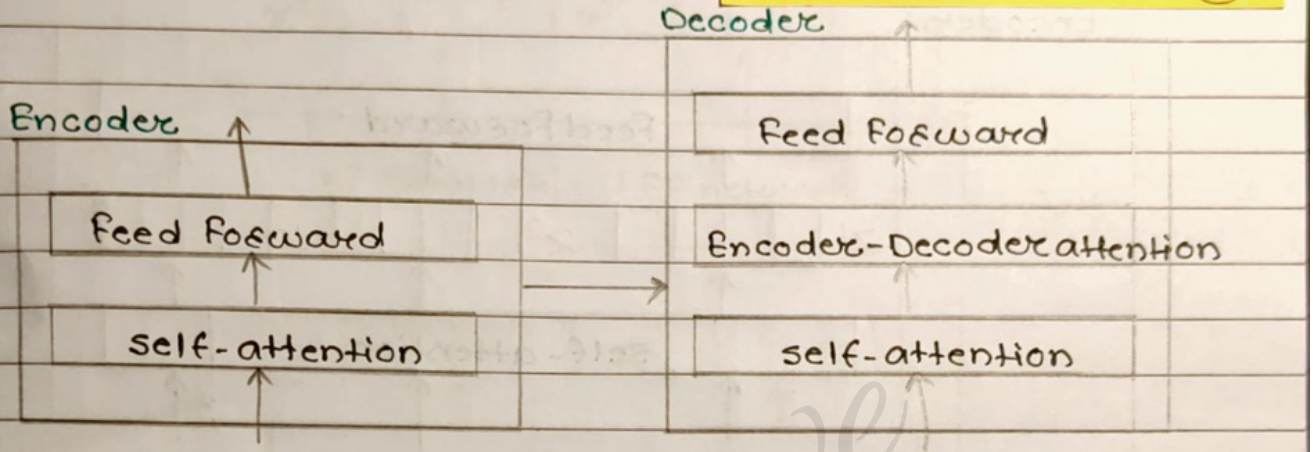
Komal Dive

DATE

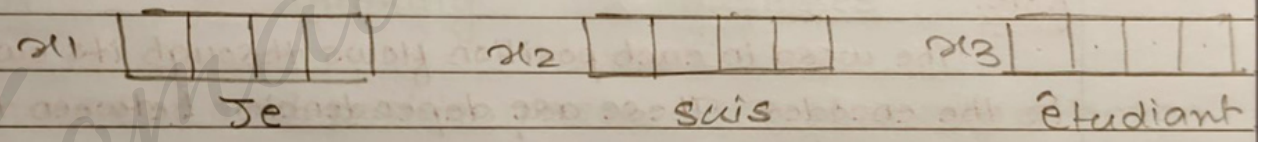
Encoder and Decoder Stack:

N = 6

Encoder & Decoder composed of a stack (6)



In general, we begin by turning each input word into a vector using an embedding algorithm.



Each word is embedded into a vector size 512.

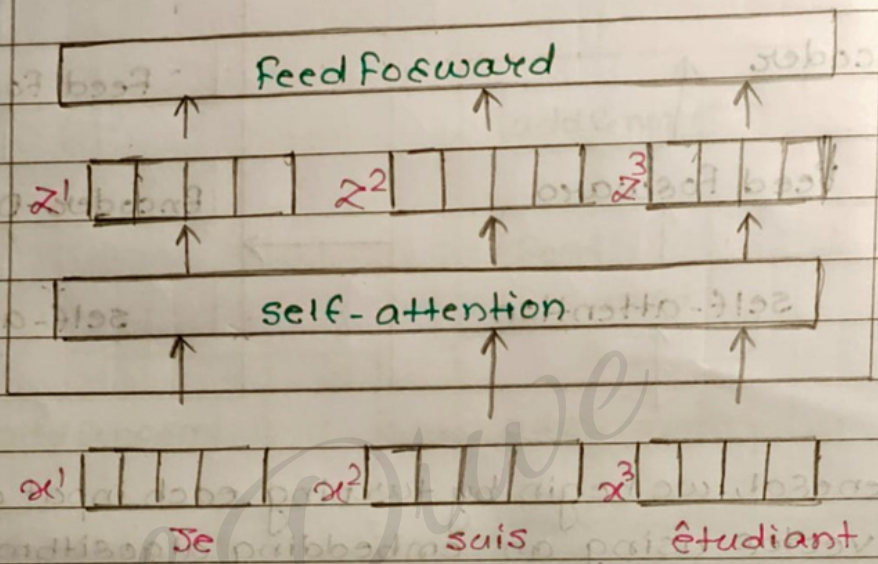
* Embedding only happens in the bottom-most encoder.

The abstraction i.e. common to all the encoders is that that they receive a list of vectors each of the size 512.
In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below.

DATE

size of this list → hyperparameter
(basically would be the length of the longest sentence in our training dataset).

Encoder:

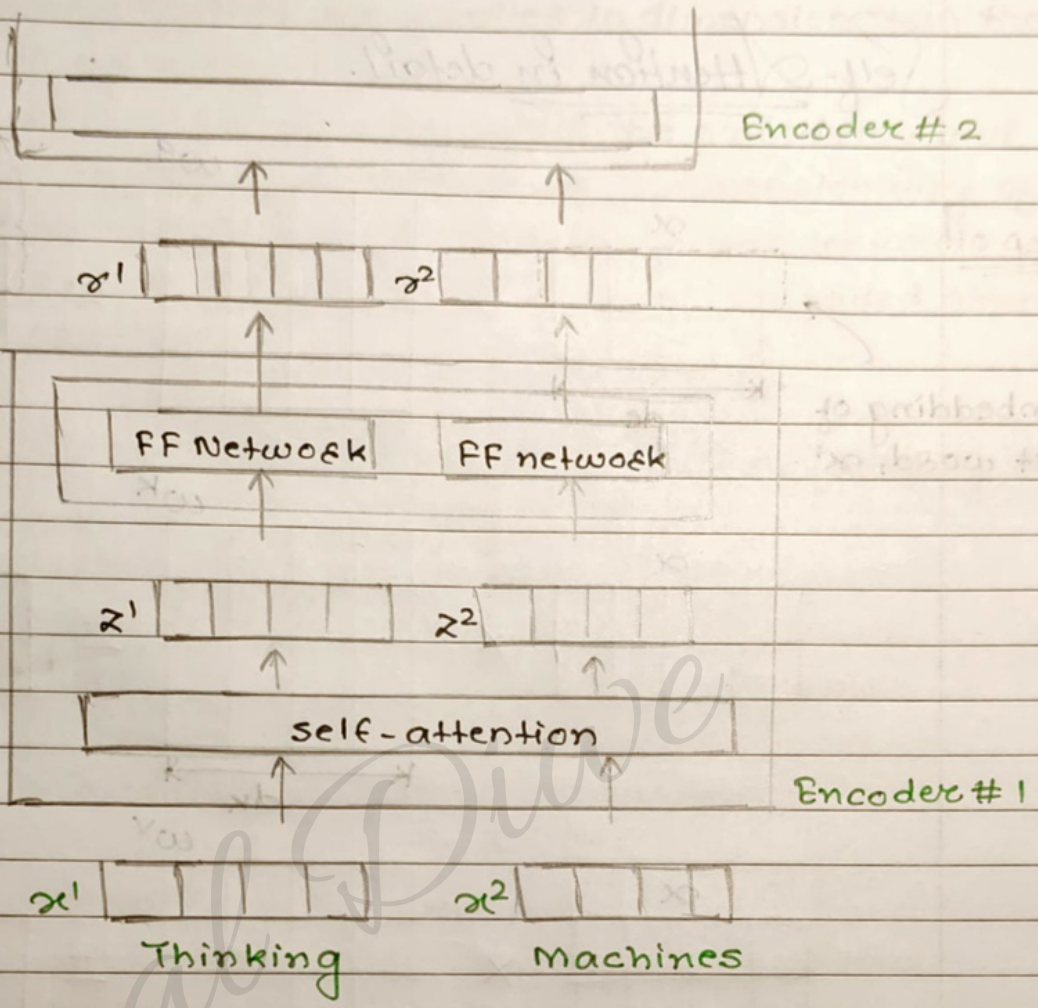


Note:

The word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer.

The FF layer doesn't have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

DATE

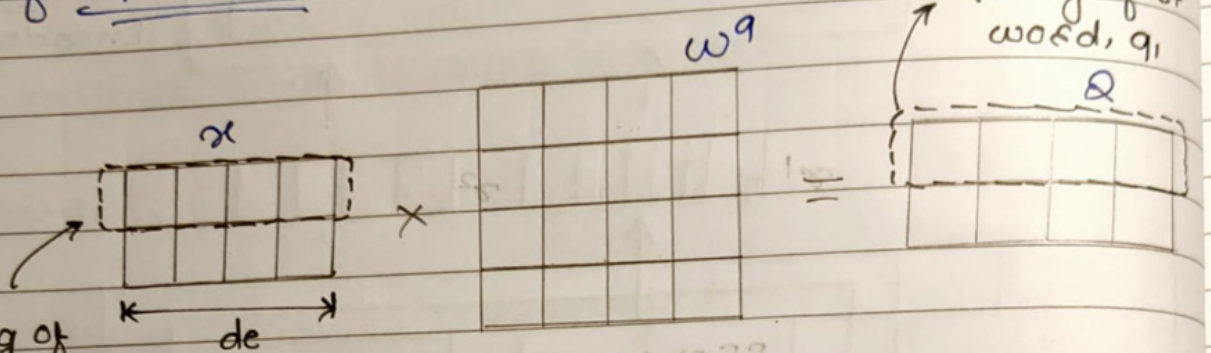


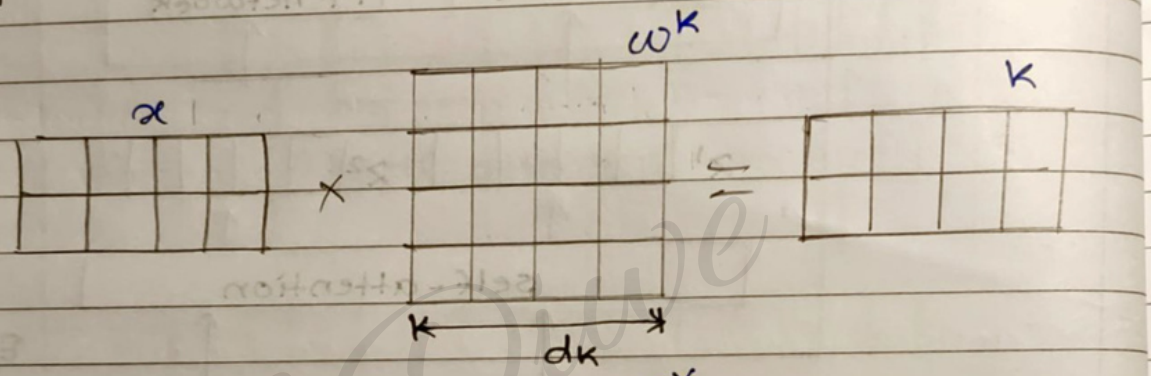
The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network
- the exact same network with each vector flowing through it separately.

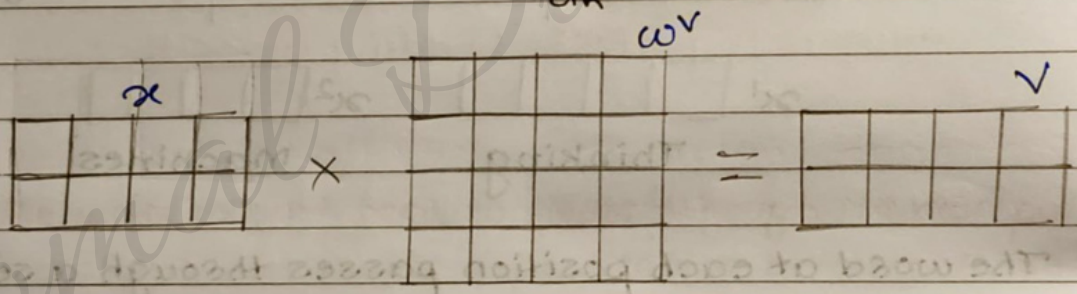
DATE

Self-Attention in detail.

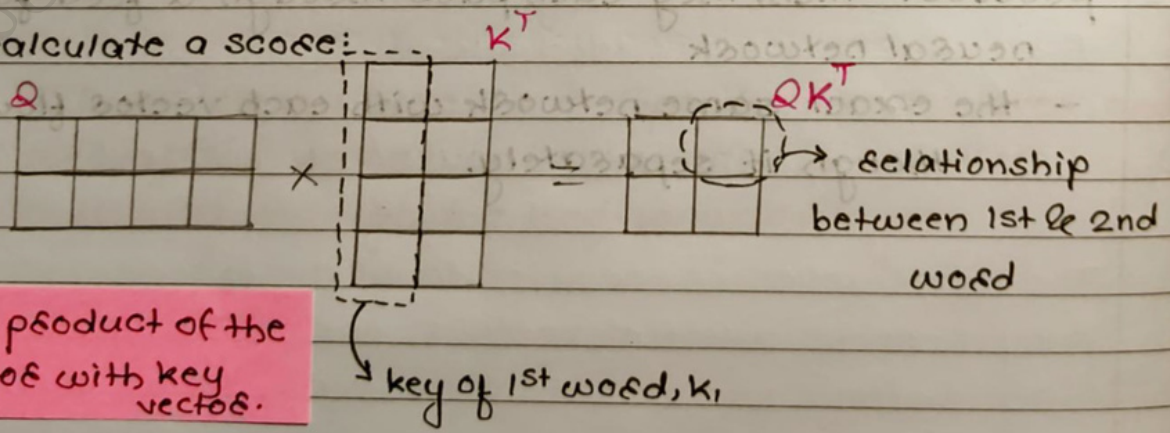
Step 01:







Step 02: calculate a score:



Score: dot product of the query vector with key vector.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

DATE

Note: these new vectors are smaller in dimension than the embedding vectors.
Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512.
They don't have to be smaller, this is an architecture choice to make the computation of multi-headed attention (mostly) constant.

step 03: (the square root of the dimension of the key vectors used in the paper – 64).

$$QK^T \;/\sqrt{dk} = QK^T/\sqrt{dk}$$

step 04:

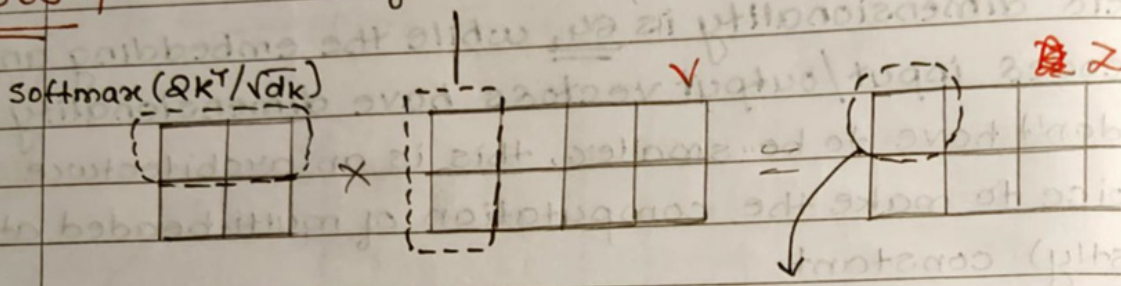
$$\text{softmax}\left(QK^T/\sqrt{dk}\right) =$$

* softmax normalizes the scores so they're all positive and add up to 1.

This softmax score determines how much each word will be expressed at this position.
Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

DATE

Step 05: 1st value of 1st & 2nd word



attention of 1st word with first value of words

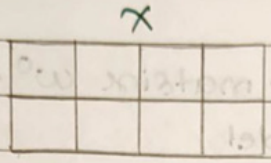
The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words.

The resulting vector is one we can send along to the feed-forward neural network.

With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices.

If we do the same self-attention calculation ~~we~~ outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices.

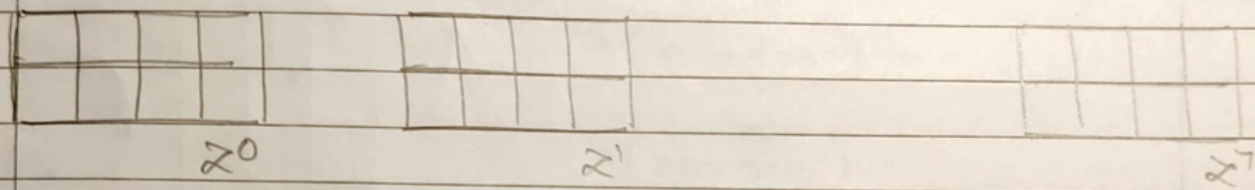
DATE



Thinking machines

↓ calculating attention separately in eight different attention heads

attention head #0 | attention head #1 | ..... | attention head #7

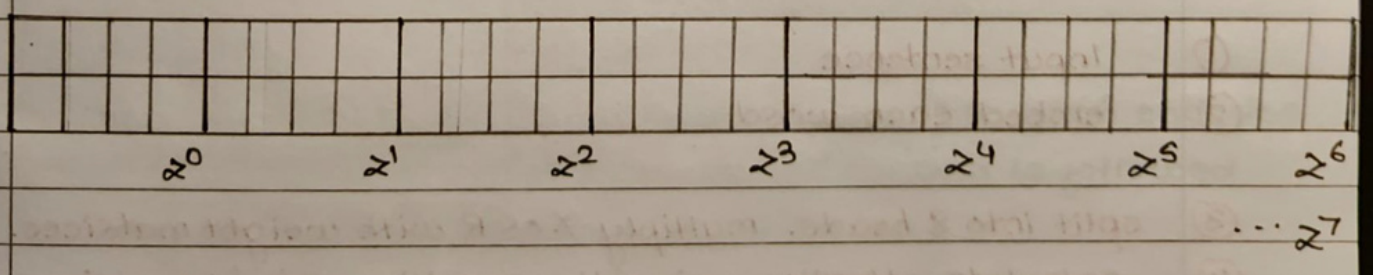


The feed-forward layer is not expecting eight matrices - it's expecting a single matrix (a vector for each word). so, we need a way to condense these eight down into a single matrix.

How do we do that?

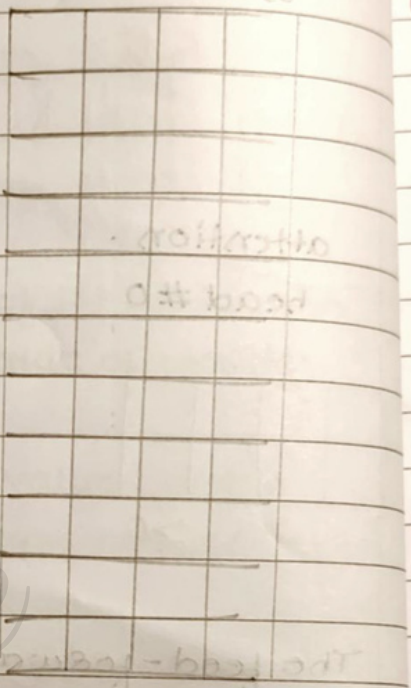
we concat the matrices then multiply them by an additional weights matrix ~~WO~~. W^0.

1. concatenate all the attention heads.



DATE

2. multiply with a weight matrix W^0 that was trained jointly with the model



3. The result would be the Z matrix that captures information from all the attention heads.
   we can send this forward to the FFNN.

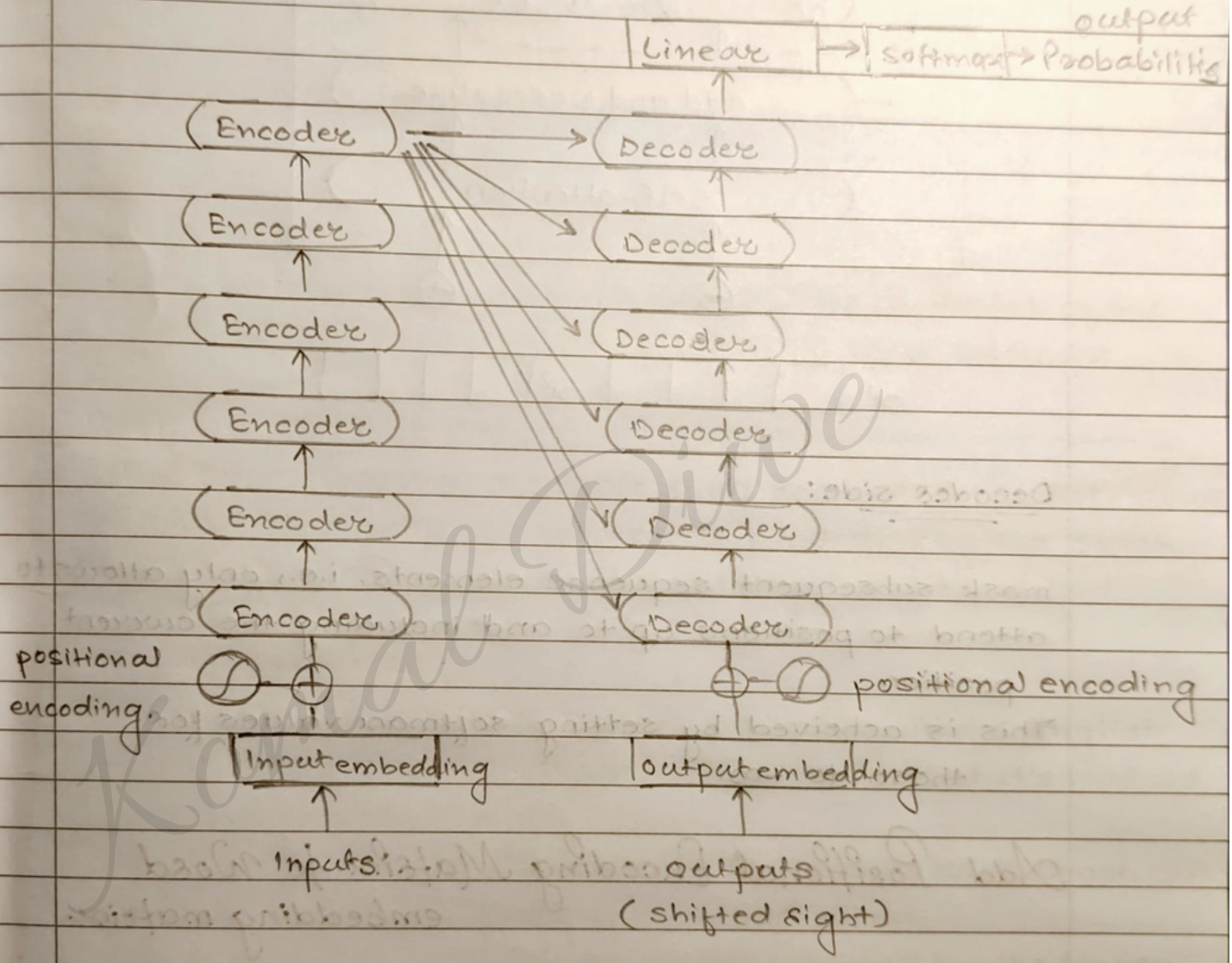
* In all encoders other than #0, we don't need need embedding.

Summarize:

① Input sentence
② embed each word
③ split into 8 heads. multiply X or R with weight matrices.
④ calculate attention using the resulting Q/K/V matrices.

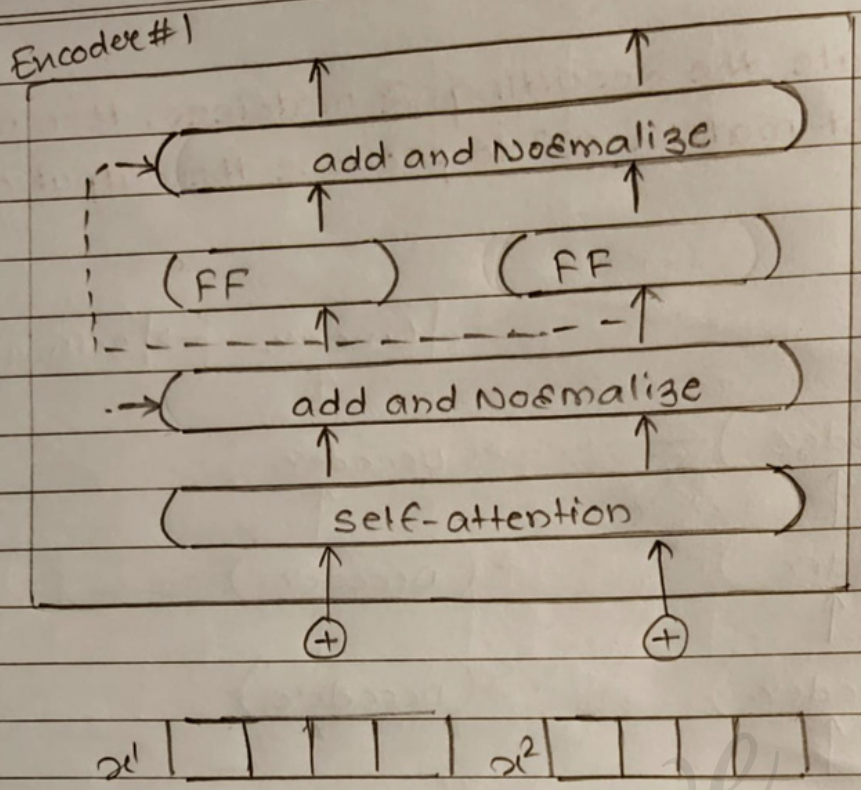
DATE

5. concatenate the resulting 8 matrices, then multiply with weight matrix W^o to produce the output of the layer.



Note: Each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization step.

DATE



Decoder side:

mask subsequent sequence elements, i.e., only allows to attend to positions up to and including the current position.
This is acheived by setting softmax values for those to $-\infty$.

Add Positional Encoding Matrix to Word embedding matrix.

- Scaled dot-product and fully-connected layer are permutation invariant.
- Sinusoidal positional encoding is a vector of small values (constants) added to the embeddings.

The "Encoder-Decoder Attention" layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the keys and values matrix from the output of the encoder stack.

- As a result, s
  embeddings
  sentence.

The Final Layer and Softmax Layer.

The decoder stack outputs a vector of floats.
How do we turn that into a word? That's the job of the final Linear layer which is followed by a softmax layer.

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

Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset.
This would make the logits vector 10,000 cells wide - each cell corresponding to the score of a unique word.
That is how, the output of the model followed by the Linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

- As a result, same word will have slightly different embeddings depending on where they occur in the sentence.

The Final Layer and Softmax Layer.

The decoder stack outputs a vector of floats.
How do we turn that into a word? That's the job of the final Linear layer which is followed by a softmax layer.

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

Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset.
This would make the logits vector 10,000 cells wide - each cell corresponding to the score of a unique word.
That is how, the output of the model followed by the Linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.