⇒ Transformers:

What and why transformers?

Transformers in NLP are a type of Deep Learning model that use self attention mechanisms to analyze and process natural language data. They are encoder-decoder models that can be used for many applications, including machine translation.

↓

sequence to sequence task

e.g

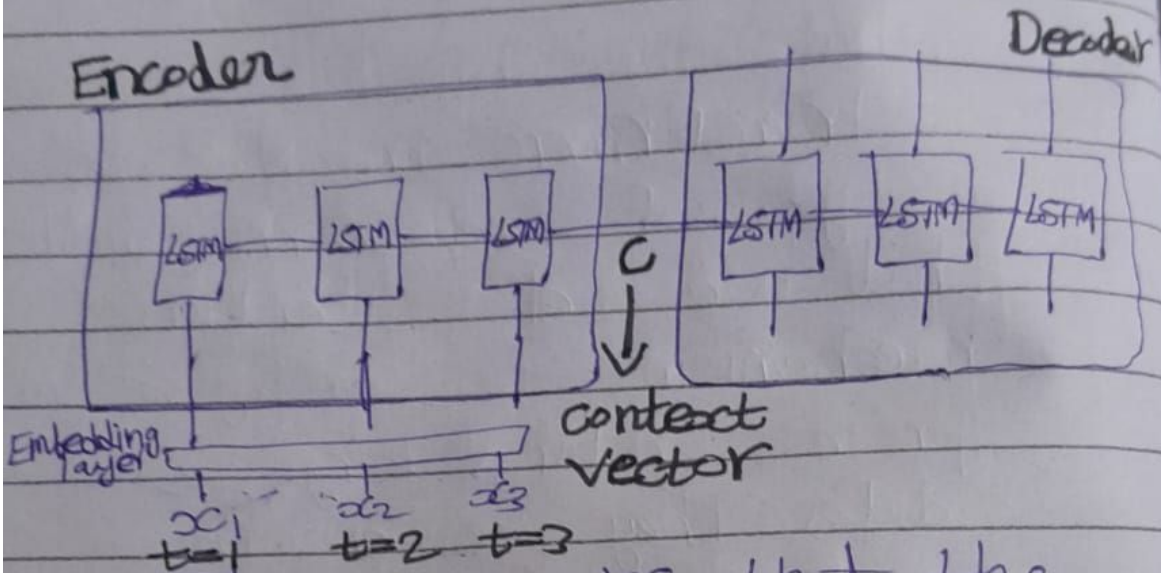
Language Translation → Google Translator

many-to-many:

inputs ⇒ many outputs ⇒ many

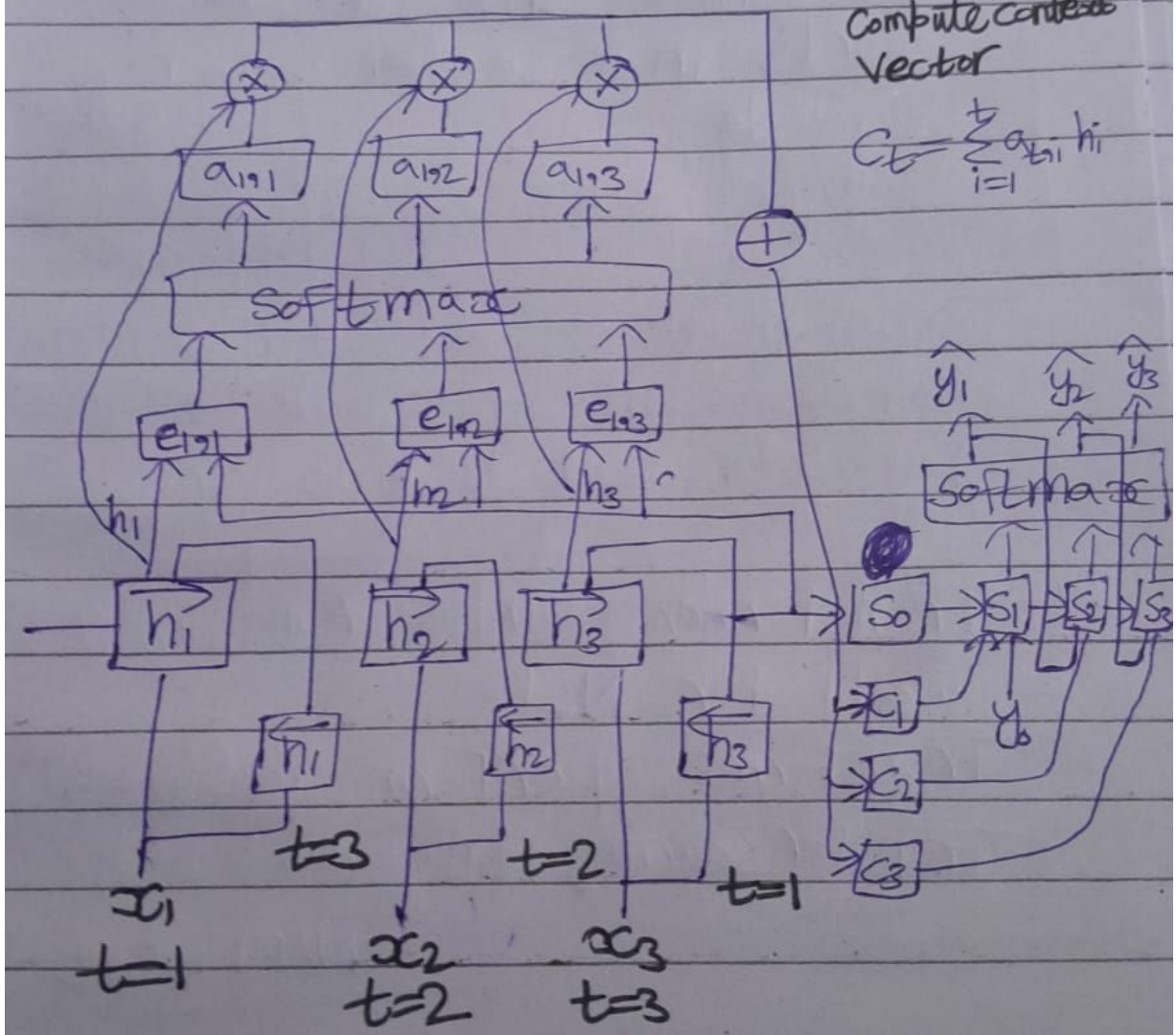
- Sequence of both inputs and outputs is important. (seq2seq task)

We can also use Encoder-Decoder for this task



⇒ The issue was that the context vector passing has more influe of latest timestamp i.e t=3 and as sentence get longer the influence of starting time stamps i.e t=1 becomes negligible so context vector is not sufficient and as length of sentence increases the blue score (performance) decreases.

To resolve this issue we used **Attention Mechanism** (passing additional context with context vector)



Problem with Attention Mechanism

① We are sending single word to a single timestamp So parallelly we cannot send all the words in a sentence So not scalable.

Dataset → huge → Not scalable w.r.t training

Transformers → use Self Attention Module → All words will be parallelly sent to Encoder

- Transformers works Amazingly well for NLP.
- Transforms perform really well in Transfer Learning → Multimodal task → NLP + Image

Transformers → AI Space → SOTA Models
(State of Art)

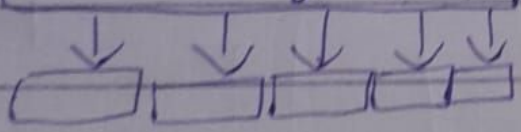
BERT GPT

Trained on huge data

⇒ Contextual Embedding → Self Attention

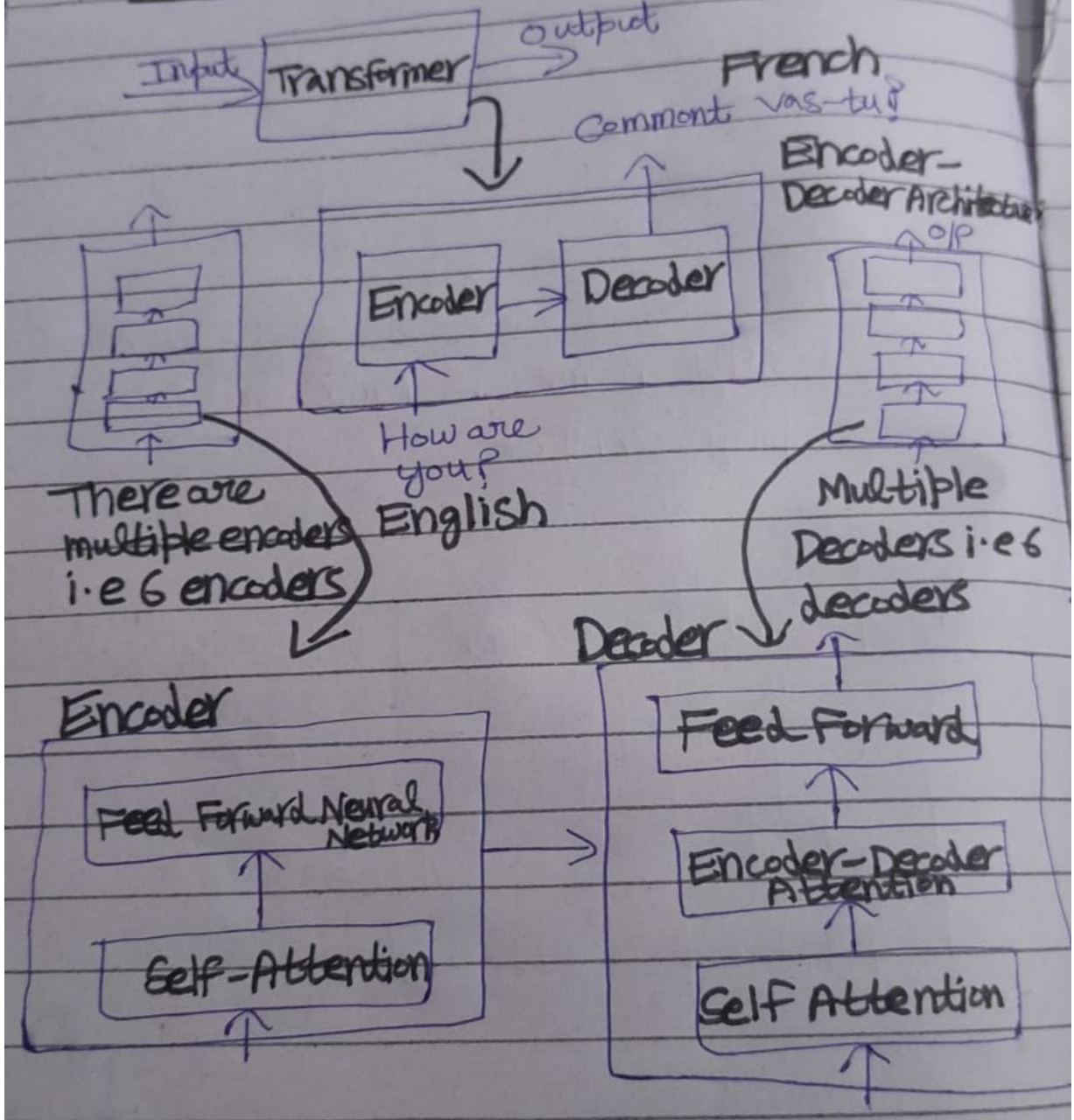
eg My name is Amir and I play cricket

Word2Vec → Embedding Layer (Here we get fixed Vectors)

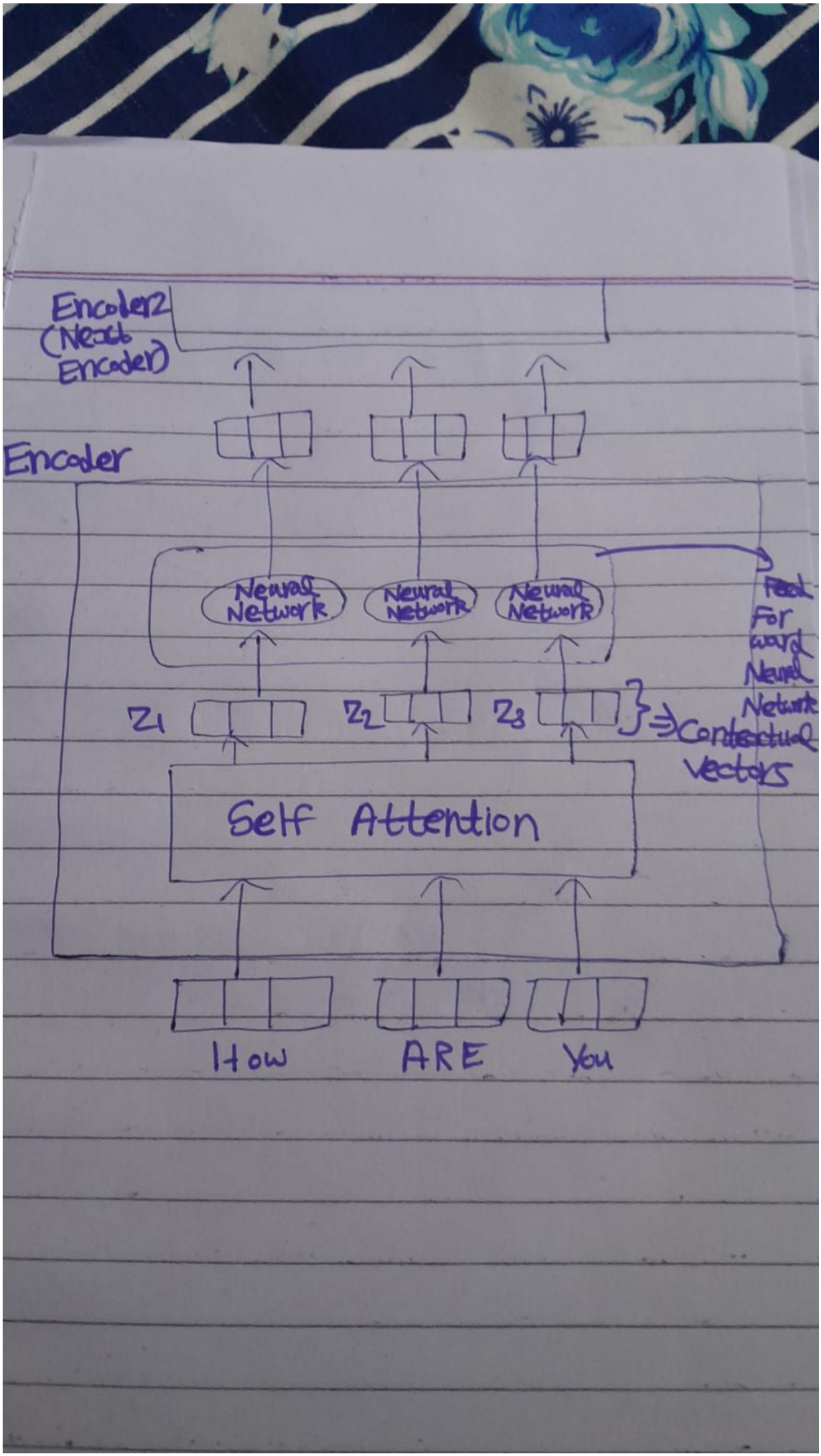


Contextual Vector Embedding (Here we get vectors that show relationship or context b/w words in a sentence using self attention)

⇒ Basic Transformer Architecture:



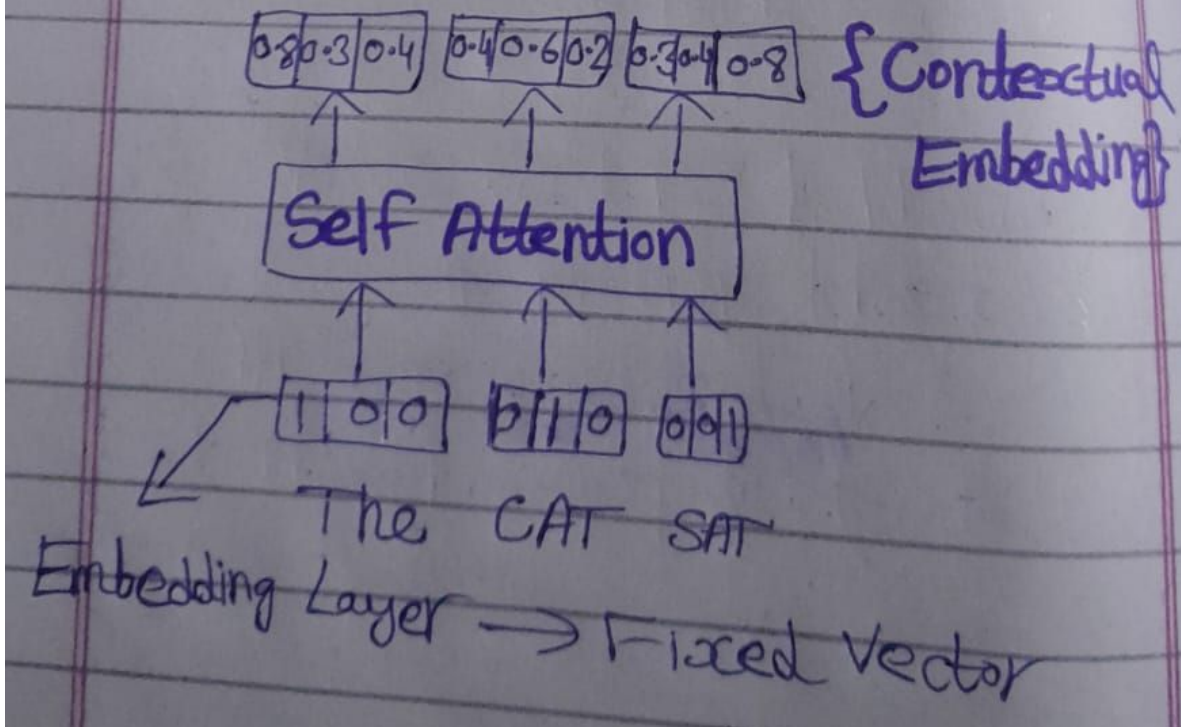
(Remaining on Next Register)


Encoder2
(Next
Encoder)
Encoder
Neural Network
Neural Network
Neural Network
Feed
For
ward
Neural
Network
z1
z2
z3
Contextual
vectors
Self Attention
How
ARE
You

⇒ Self Attention at Higher and detailed level:

Self attention also known as scaled dot-product attention is a crucial mechanism in transformer architecture to weigh the importance of different tokens in the input sequence relative to each other

Idea



1) Inputs: Queries, Keys, Values

Model → Queries, Keys, Values

(i). Query Vectors (Q):

Query Vectors represent the token for which we are calculating attention. They help determine the Importance of other tokens in the context of current token.

Importance:

.Focus Determination:

Queries help the model decide which parts of the sequence to focus on for each specific token. By calculating dot product b/w query vector and all key vectors, the model assess how much attention to give to each token relative to current token.

Contextual Understanding:

Queries contribute to understanding the relationship between current token and rest of sequence, which is essential for capturing dependencies and context

(ii) Key Vectors (K):

Key Vectors represent all the tokens in the sequence and are used to compare with query vectors to calculate attention scores.

(iii) Value Vectors (V):

Value vectors hold the actual information that will be aggregated to form the output of attention mechanism

- **Example:**

Input sequence = ["The", "CAT", "SAT"]
Embedding size = 4

① Token Embedding (using Embedding Layer)

$E_{The} = [1\ 0\ 1\ 0]$
$E_{CAT} = [0\ 1\ 0\ 1]$
$E_{SAT} = [1\ 1\ 1\ 1]$

② **Linear Transformation:**

Let create Q, K, V by (dot product) multiplying the embeddings by learned weight matrices W_Q, W_K and W_V.

Let's consider

$$W_Q = W_K = W_V = I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Q_{The} = [1\ 0\ 1\ 0] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Q_{The} = \begin{bmatrix} 1\times1+0\times0+1\times0+0\times0, & 1\times0+0\times1+1\times0+0\times0, & 1\times0+0\times0+1\times1+0\times0, & 1\times0+0\times0+1\times0+0\times1 \\ 0\times0, & 1\times0+0\times0+1\times1+0\times0, & 1\times0+0\times0+1\times0+0\times1, & \\ 0\times1 & & & \end{bmatrix}$$

As it is identity matrix so it remain same

$Q_{The} = [1\ 0\ 1\ 0]$

Similarly

(i) $Q_{The} = K_{The} = V_{The} = [1\ 0\ 1\ 0]$

(ii) $Q_{CAT} = K_{CAT} = V_{CAT} = [0\ 1\ 0\ 1]$

(iii) $Q_{SAT} = K_{SAT} = V_{SAT} = [1\ 1\ 1\ 1]$

③ Computing Attention scores:

(i) For the token "The"

$$Score(Q_{The}, K_{The}) = [1\ 0\ 1\ 0] \cdot [1\ 0\ 1\ 0]^T$$

$$= [1\ 0\ 1\ 0] \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$Score(Q_{The}, K_{The}) = [1\times1 + 0\times0 + 1\times1 + 0\times0]$$

$$\boxed{Score(Q_{The}, K_{The}) = 2}$$

$$Score(Q_{The}, K_{CAT}) = [1\ 0\ 1\ 0] \cdot [0\ 1\ 0\ 1]^T$$

$$\boxed{Score(Q_{The}, K_{CAT}) = 0}$$

$$Score(Q_{The}, K_{SAT}) = [1\ 0\ 1\ 0] \cdot [1\ 1\ 1\ 1]^T$$

$$\boxed{Score(Q_{The}, K_{SAT}) = 2}$$

(ii) For the token "CAT"

$\text{Score}(Q_{CAT}, K_{The}) = [0\ 1\ 0\ 1] \cdot [1\ 0\ 1\ 0]^T$

$\boxed{\text{Score}(Q_{CAT}, K_{The}) = 0}$

$\text{Score}(Q_{CAT}, K_{CAT}) = [0\ 1\ 0\ 1] \cdot [0\ 1\ 0\ 1]^T$

$\boxed{\text{Score}(Q_{CAT}, K_{CAT}) = 2}$

$\text{Score}(Q_{CAT}, K_{SAT}) = [0\ 1\ 0\ 1] \cdot [1\ 1\ 1\ 1]^T$

$\boxed{\text{Score}(Q_{CAT}, K_{SAT}) = 2}$

(iii) For the token "SAT"

$\text{Score}(Q_{SAT}, K_{The}) = [1\ 1\ 1\ 1] \cdot [1\ 0\ 1\ 0]^T$

$\boxed{\text{Score}(Q_{SAT}, K_{The}) = 2}$

$\text{Score}(Q_{SAT}, K_{CAT}) = [1\ 1\ 1\ 1] \cdot [0\ 1\ 0\ 1]^T$

$\boxed{\text{Score}(Q_{SAT}, K_{CAT}) = 2}$

$\text{Score}(Q_{SAT}, K_{SAT}) = [1\ 1\ 1\ 1] \cdot [1\ 1\ 1\ 1]^T$

$\boxed{\text{Score}(Q_{SAT}, K_{SAT}) = 4}$

Scaling:

We take up the scores and scale down by dividing the scores by the $\sqrt{d_k}$

Now

$d_k \Rightarrow$ dimension of key vector $= 4$

$\boxed{\sqrt{d_k} = 2}$

⇒ Scaling in the attention mechanism is crucial to prevent the dot product from growing too large to ensure stable gradient during training.

Problems here are:

① Gradient Exploding

② Softmax saturation (vanishing Gradient Problem)

Now Let's

$Q = [2\ 3\ 4\ 1]$

$K_1 = [1\ 0\ 1\ 0]$

$K_2 = [0\ 1\ 0\ 1]$

Without scaling:

$Q \cdot K_1^T = 2\times1 + 3\times0 + 1\times1 + 1\times0 = 2+1$

$\boxed{Q \cdot K_1^T = 6}$

$Q \cdot K_2^T = 2\times0 + 3\times1 + 1\times0 + 1\times1 = 3+1$

$\boxed{Q \cdot K_2^T = 4}$

$\boxed{\text{Attention scores} = [6, 4]}$

Let we don't applied scaling

$$\text{Softmax}([6,4]) = \left[\frac{e^6}{e^6+e^4}, \frac{e^4}{e^6+e^4}\right]$$

$$\text{Softmax}([6,4]) = \left[\frac{e^6}{e^6(1+e^{-2})}, \frac{e^4}{e^4(e^2+1)}\right]$$

$$\text{Softmax}([6,4]) = \left[\frac{1}{(1+e^{-2})}, \frac{1}{(e^2+1)}\right]$$

$$\boxed{\text{Softmax}([6,4]) \approx [0.88, 0.12]}$$

⇓

Most of the attention weights is assigned to 1st key vector and very little to the 2nd vector

- There is hardly difference of 2 still so much difference in softmax so if difference become larger then problem

With scaling:

$[6, 4] \Rightarrow$ Attention scores

To scale divide by $\sqrt{d_k} = 2$

$[6/2, 4/2]$

$\boxed{[3, 2]} \Rightarrow$ Attention scores after scaling

Now apply Softmax

$$\text{Softmax}([3,2]) = \left[\frac{e^3}{e^3+e^2}, \frac{e^2}{e^3+e^2}\right]$$

$$= \left[\frac{e^3}{e^2(1+e^{-1})}, \frac{e^2}{e^2(e^1+1)}\right]$$

$$= \left[\frac{1}{(1+e^{-1})}, \frac{1}{(e^1+1)}\right]$$

$$\boxed{\text{Softmax}([3,2]) = [0.73, 0.27]}$$

$\Downarrow$

Attention weights

★ Here the attention weights are more balanced compared to the unscaled case.

④ Scaling:

We know

$d_K = 4$

$\sqrt{d_K} = 2$

For token "The":

$\text{Scaled\_score}(Q_{The}, K_{The}) = \frac{2}{2} = 1$

$\text{Scaled\_score}(Q_{The}, K_{CAT}) = 0/2 = 0$

$\text{Scaled\_score}(Q_{The}, K_{SAT}) = 2/2 = 1$

Similar scaling will be done for all other tokens.

) Apply Softmax:

For "The"

$$\text{Softmax}([1, 0, 1]) = \left[\frac{e^1}{e^1+e^0+e^1}, \frac{e^0}{e^1+e^0+e^1}, \frac{e^1}{e^1+e^0}\right.$$

$$\text{Softmax}([1, 0, 1]) = [0.4223, 0.1554, 0.4$$

⇓

Attention Weights for "The"

⑥ Weighted Sum of Values:

We multiply the attention weight by corresponding value vectors i.e actual vectors

For token "The":

$$Output_{The} = 0.4223 \times V_{The} + 0.1554 \times V_{CAT} + 0.4223 \times V_{SAT}$$

$$Output_{The} = 0.4223 \times [1\ 0\ 1\ 0] + 0.1554 \times [0\ 1\ 0\ 1] + 0.4223 \times [1\ 1\ 1\ 1]$$

$$Output_{The} = [0.4223, 0, 0.4223, 0] + [0, 0.1554, 0, 0.1554] + [0.4223, 0.4223, 0.4223, 0.4223]$$

$$\boxed{Output_{The} = [1.2669, 0.9999, 1.2669, 0.9999]}$$

⇓

Contextual Vector

So,

The

[1 0 1 0]

⇓

Self Attention

⇓ Contextual Embedding

[1.2669, 0.9999, 1.2669, 0.9999]

⇓

Contextual Vector

- **Summarized steps:**

1. Calculate Q, K, V
2. Compute Attention Scores
3. Scale Attention Scores
4. Perform Softmax
5. Weighted Sum of Values

⇒ Multi-head Attention:

- In Self Attention with Multi-heads actualy we make multiple query, Key and value vectors for a single token like following the steps behind and we get multiple contextual vectors (heads) for single token.
- It actually expands model's ability to focus on different positions of tokens

How we create multiple Q, K and V:

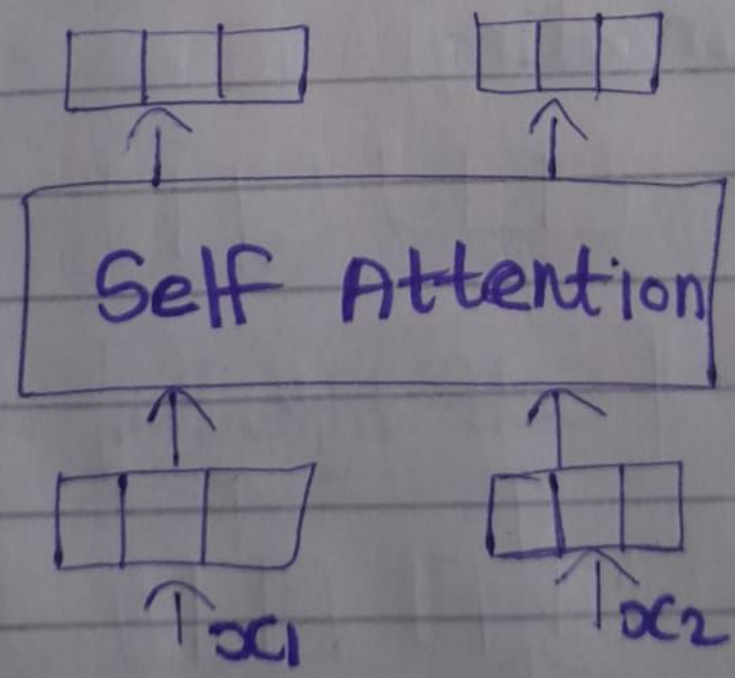
By initializing multiple different weights W_q, W_k and W_v.

(For remaining information see the repo)

Feed Forward Neural Network (In Repo)

⇒ Positional Encoding
(Representing order of sequence)

- Major Advantage of transformers is that it can process multiple words tokens parallelly.
- Major Drawback of transformers is that it lack the sequential structure (order) of words.



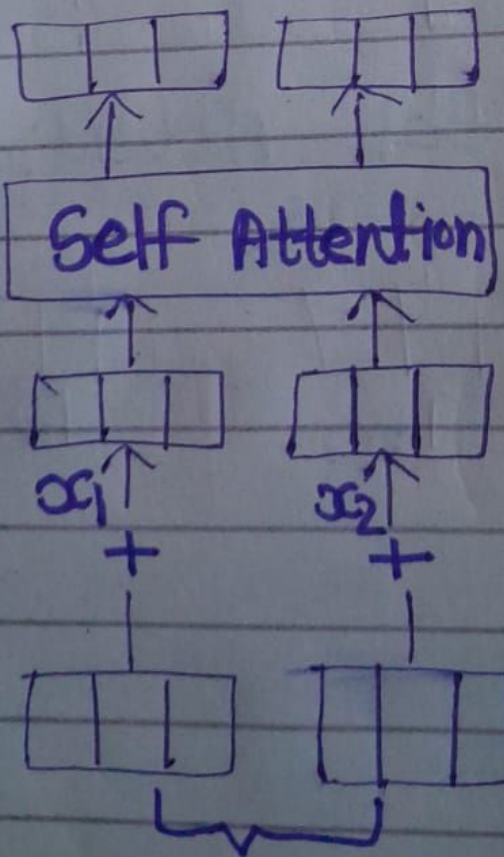
e.g

① Lion kills tiger

② Tiger kills lion

Both sentences are opposite, but as transformer we pass all words parallely and lack sequence(ordering) so it will take both as same.

⇒ To resolve this issue we use **Positional Encoding**



⇒ Types of Positional Encoding:

1) Sinusoidal Positional Encoding
2) Learned Positional Encoding
   ⇓ Positional Encodings are learned during training

1) Sinusoidal Positional Encoding:

It uses sine and cosine functions of different frequencies to create positional embeddings.

Formula:

$$P.E(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$P.E(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where,

pos ⇒ position

i ⇒ is the dimention

d_{model} ⇒ is the dimensionality of embeddings

Example:
"The cat sat"

The → [0.1 0.2 0.3 0.4]
cat → [0.5 0.6 0.7 0.8]
sat → [0.9 1.0 1.1 1.2]

Now these are embedded direct vectors before passing them to self attention we have to compute positional encoded vectors using sinusoidal positional encoding.

We know,

1. $E(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$

2. $E(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$

For our example
d_model (dimensionality) = 4

For position pos=0 i.e "The"

$$P.E(0,0) = \sin\left(\frac{0}{10000^{0/4}}\right) = 0$$

$$P.E(0,1) = \cos\left(\frac{0}{10000^{0/4}}\right) = 1$$

$$P.E(0,2) = \sin\left(\frac{0}{10000^{2/4}}\right) = 0$$

$$P.E(0,3) = \cos\left(\frac{0}{10000^{2/4}}\right) = 1$$

$$P.E = [0, 1, 0, 1] \Rightarrow \text{"The"}$$

For position pos=1 i.e "Cat"

$$P.E(1,0) = \sin\left(\frac{1}{10000^{0/4}}\right) = 0.8415$$

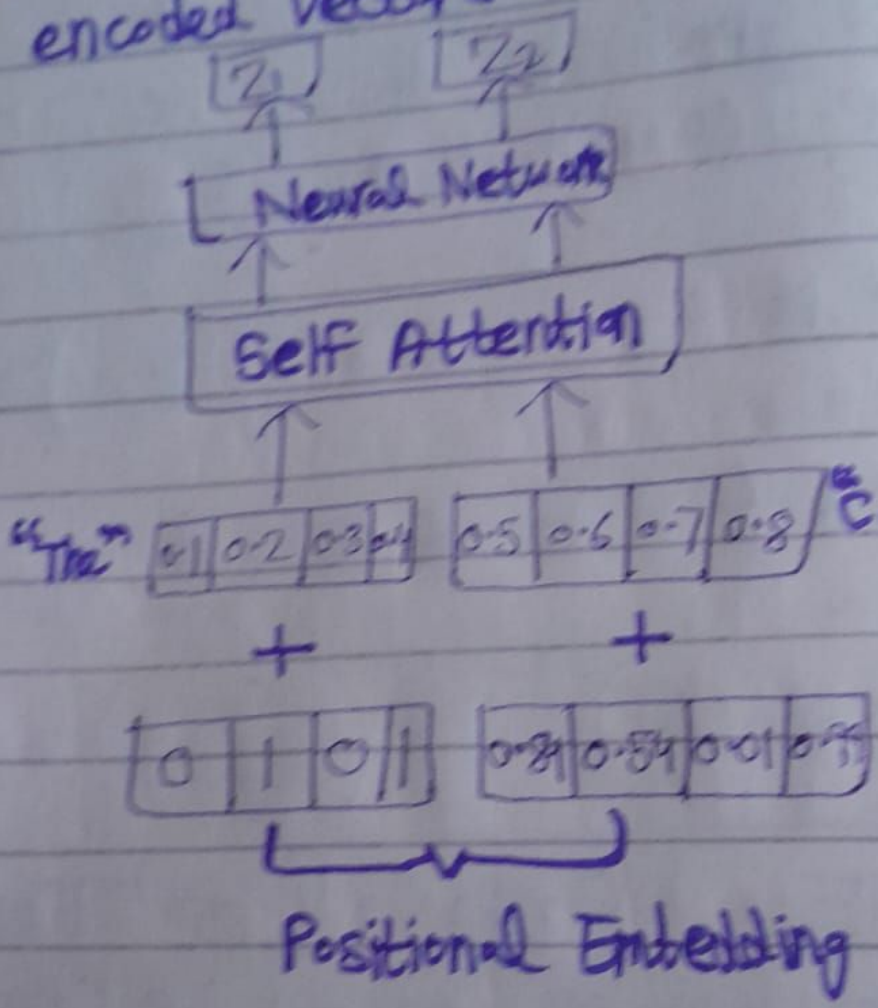
$$P.E(1,1) = \cos\left(\frac{1}{10000^{0/4}}\right) = 0.5403$$

$$P.E(1,2) = \sin\left(\frac{1}{10000^{2/4}}\right) = 0.01$$

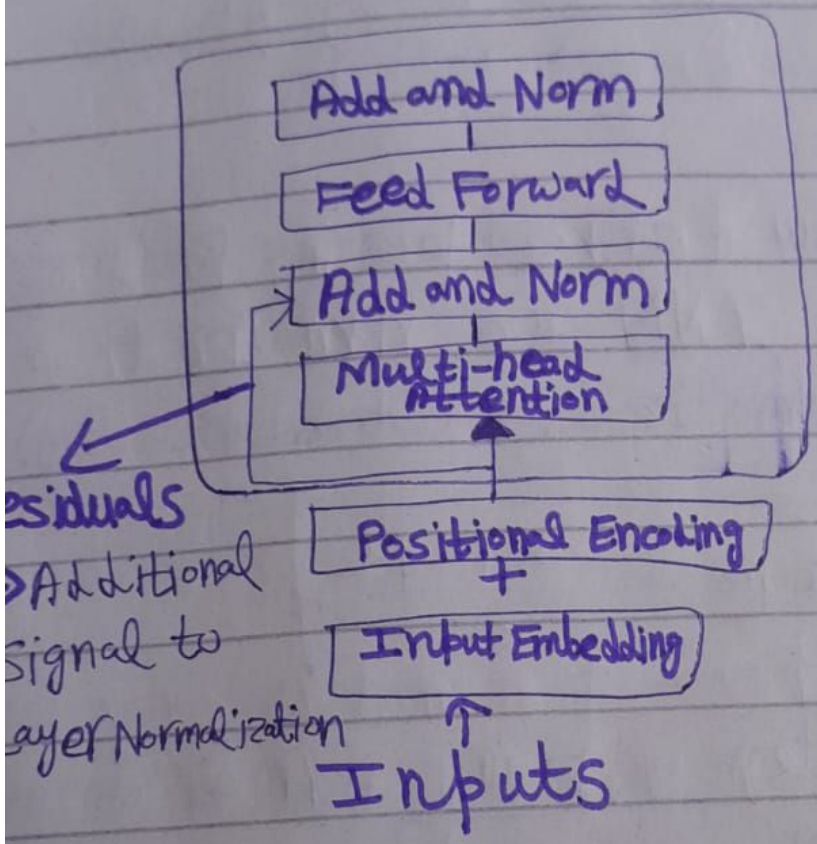
$$P.E(1,3) = \cos\left(\frac{1}{10000^{2/4}}\right) = 0.9995$$

$$P.E = [0.8415, 0.5403, 0.01, 0.9995] \Rightarrow \text{"Cat"}$$

Now we have find positional encoded vectors



⇒ Layer Normalization in transformers:



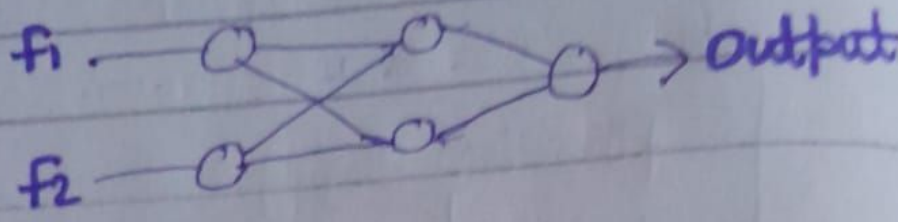
esiduals
>Additional
signal to
ayer Normalization

Normalization → Batch Normalization
Layer Normalization

Let we have dataset

f_1 (House Size)	f_2 (No. of Rooms)	Price
1200	2	45
1500	3	70
2000	3.5	80

Let ANN is:



- Now before passing features to ANN we normalize them

Normalization (Standard Scaling):

$$Z_{score} = \frac{x_i - \mu}{\sigma}$$

After applying normalization,

$$\mu = 0, \sigma = 1$$

⇒ **Advantages of Normalization:**

1. Improved training stability (as we will not face vanishing or exploding gradient problem)
2. Faster convergence (as data is zero-centered) (stable update during back propagation)

⇒ **Batch Normalization:**

In batch Normalization we take whole column like f_i (house size) compute μ (mean) and σ (standard deviation) and apply formula

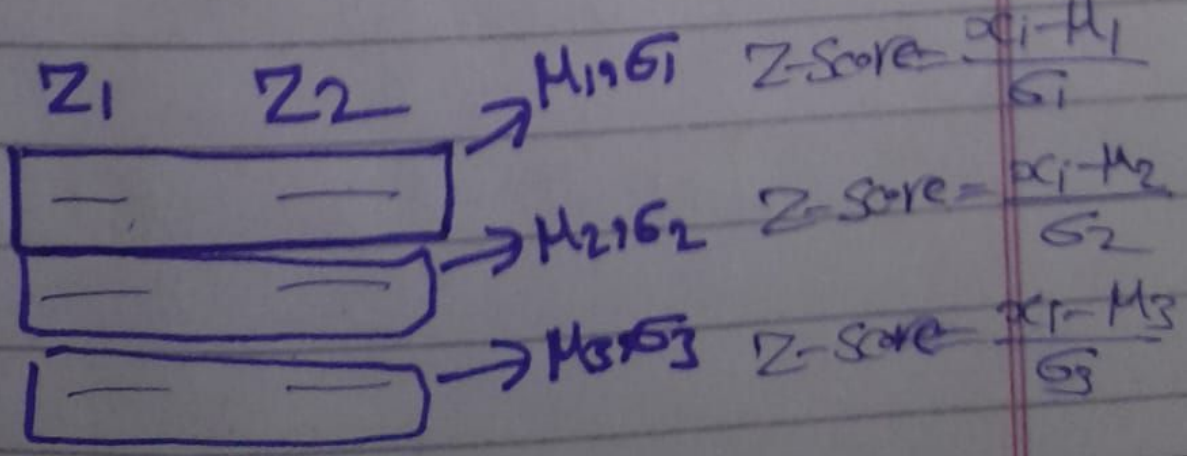
$$\text{Z-score} = \frac{x_i - \mu}{\sigma}$$

to every value in column.

This is usually used in ANN.

- But if the column contains multiple zeros then batch normalization will also affect Non-zero values.
- That's why, in **Transformers** we use **Layer Normalization.**

⇒ **Layer Normalization**



- In Layer Normalization we compute μ and σ for every layer (row) and apply

$$Z\text{-score} = \frac{x_i - \mu}{\sigma}$$

to the entries of that row.

$\gamma, \beta \rightarrow$ Learnable parameters
scale and shift parameters

- These parameters control layer Normalization that how much normalization we want to do or whether do or not.

$$y = \gamma \left[\frac{z_i - \mu_i}{\sigma_i} \right] + \beta$$

• Example:

(1) "CAT" = [2.0, 4.0, 6.0, 8.0]

(2) Parameters:

$\gamma = [1.0, 1.0, 1.0, 1.0]$ → Learned Scale

$\beta = [0.0, 0.0, 0.0, 0.0]$ → Shift

Scale and Shift parameters

Step1: Compute the mean

$$\mu = \frac{\sum x_i}{n} = \frac{2.0+4.0+6.0+8.0}{4}$$

$$\boxed{\mu = 5.0}$$

Step2: Compute the variance

$$\sigma^2 = \frac{\sum (x_i - \bar{x})^2}{n}$$

$$\sigma^2 = \frac{[(2.0-5.0)^2 + (4.0-5.0)^2 + (6.0-5.0)^2 + (8.0-5.0)^2]}{4}$$

$$\boxed{\sigma^2 = 5.0}$$

Step:03 Normalize the input

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Let

$\varepsilon = 1e^{-5}$ ⇒ To avoid division by zero

$$\sqrt{\sigma^2 + \varepsilon} = \sqrt{5.0 + 1e^{-5}} = \sqrt{5.0001}$$

$$\boxed{\sqrt{\sigma^2 + \varepsilon} = 2.236}$$

$$\hat{x}_1 = \frac{2.0 - 5.0}{2.236} \approx -1.34$$

$$\hat{x}_2 = \frac{4.0 - 5.0}{2.236} \approx -0.45$$

$$\hat{x}_3 = \frac{6.0 - 5.0}{2.236} \approx 0.45$$

$$\hat{x}_4 = \frac{8.0 - 5.0}{2.236} \approx 1.34$$

$$\boxed{\hat{x} = [-1.34, -0.45, 0.45, 1.34]}$$

⇓

Normalized Vector

Step:4) Scale and Shift:

$$y_i = \gamma_i \cdot \hat{x}_i + \beta_i$$

We know $\gamma = [1.0, 1.0, 1.0, 1.0]$

$\beta = [0.0, 0.0, 0.0, 0.0]$

$\hat{x} = [-1.34, -0.45, 0.45, 1.34]$

$y_1 = \gamma_1 x_1 + \beta_1 = 1.0(-1.34) + 0.0 = -1.34$

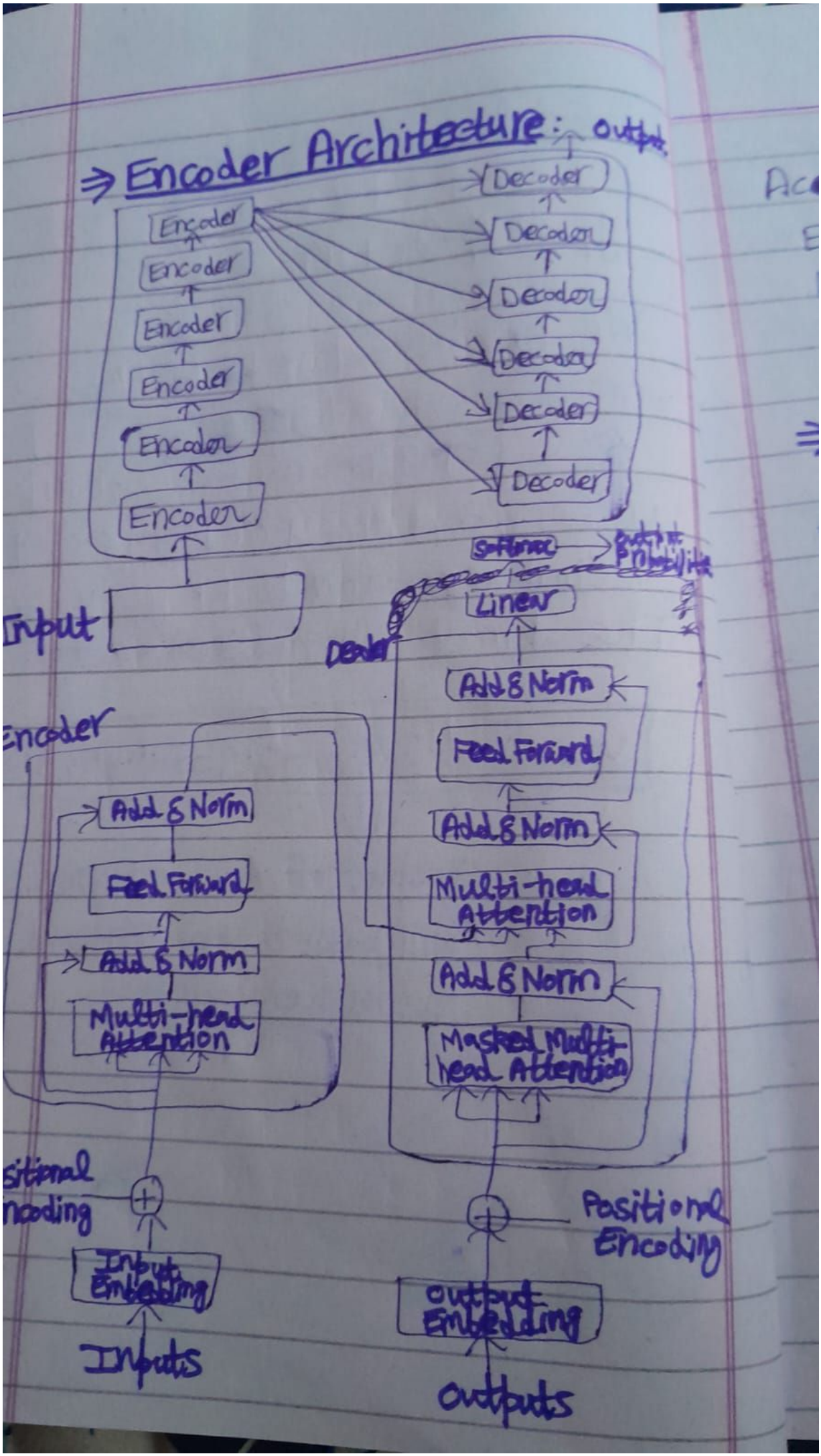
Similarly calculate for y_2, y_3, y_4

The final Normalized Vector:

$$y = [-1.34, -0.45, 0.45, 1.34]$$

⇓

Output of Layer Normalization (will pass to Feed Forward Neural Network)


Encoder Architecture:
Encoder
Encoder
Encoder
Encoder
Encoder
Encoder
Decoder
Decoder
Decoder
Decoder
Decoder
Decoder
Input
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-head Attention
Add & Norm
Masked Multi-head Attention
Add & Norm
Feed Forward
Multi-head Attention
Input Embedding
Inputs
Output Embedding
Outputs
Positional Encoding

According to research paper,
Embedding vector dimensions = 512
No. of encoders & Decoders = 6
Q, K, V = 64
Head Attention = 8

⇒ **Residual Connection (Skip connection): (Why to use)**

1) **Adressing the vanishing Gradient problem:**
Residual connection create a short paths for gradients to flow directly through the network so gradient remains sufficiently large.

2) **Improve gradient flow**
Convergence will be faster

3) **Enables training of deeper networks**

⇒ Feed Forward Neural Network (Why to use).

① Adding Non-linearity

② Processing each position independently

We know

Self Attention captures relationships

Feed Forward Network

↓

Each token representation Independently

↓↓

Transforming these representation further and allows the model to learn richer representation.

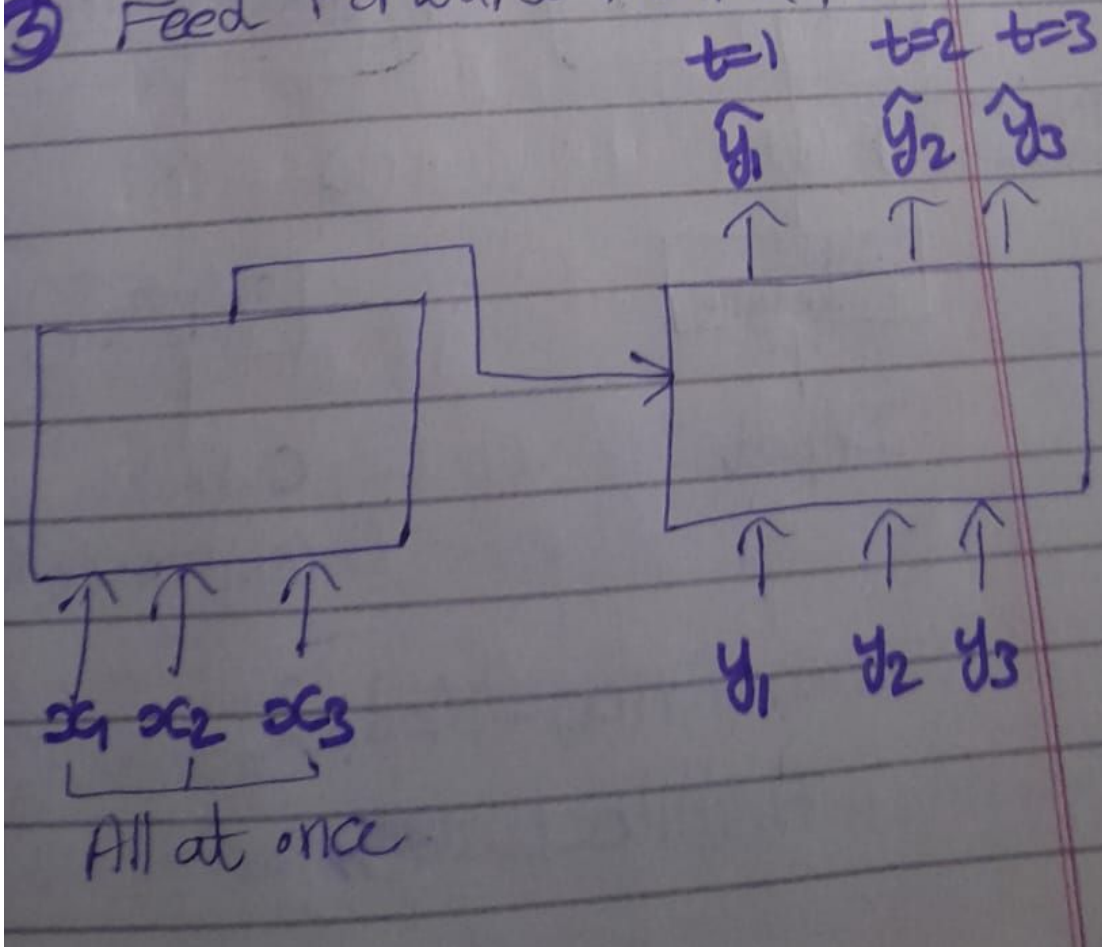
③ Adds Depth to the model as More depth more learnings and performance

⇒ Decoders in Transformers:

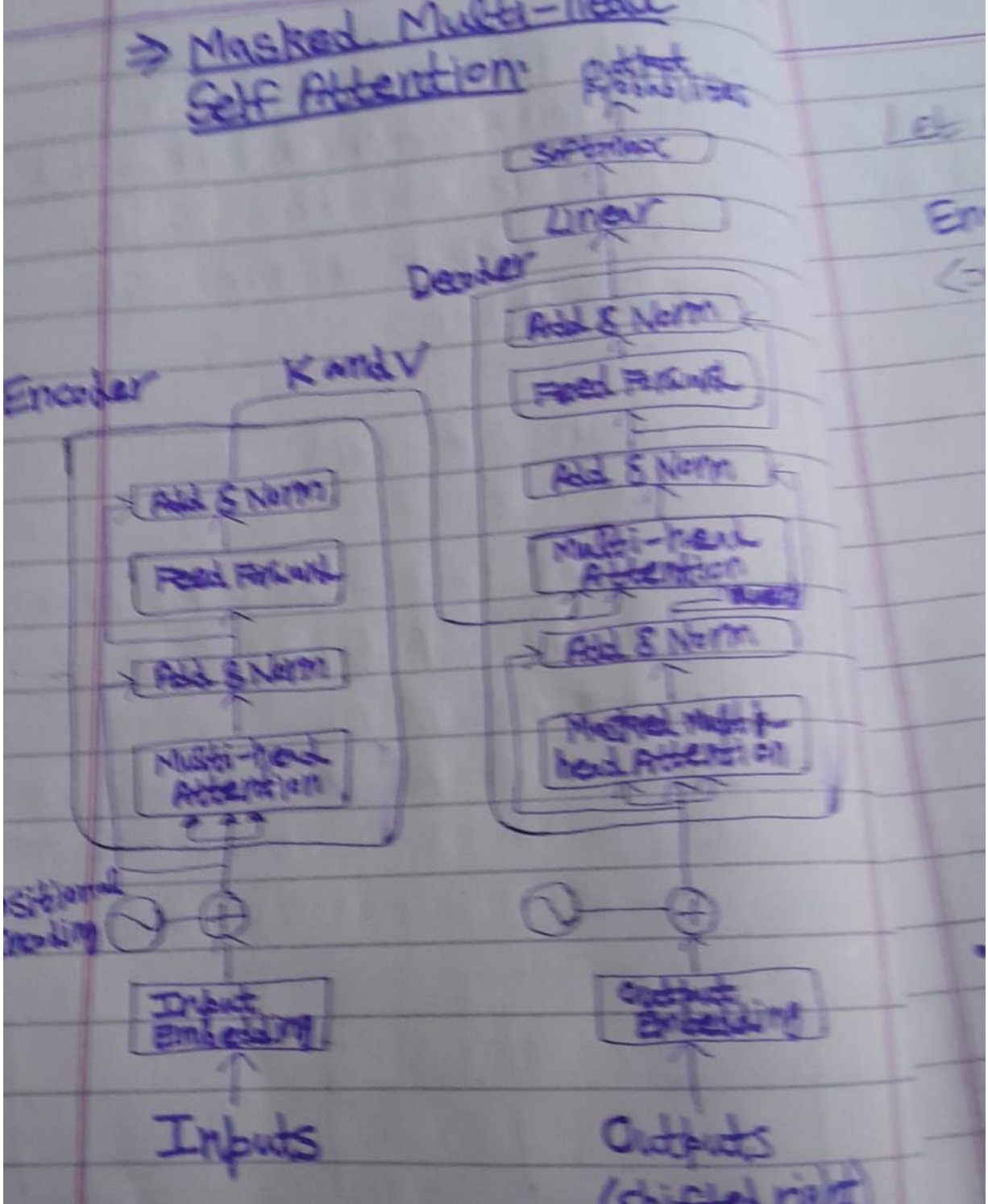
The transformer decoder is responsible for generating the output sequence one at a time using encoder's output and previously generated tokens

3 main components of decoder are:

1. Masked Multi-head Attention
2. Multi-head Attention (Encoder-Decoder Attention)
3. Feed Forward Neural Network



⇒ Masked Multi-head Self Attention:



Transformer-Model Architecture

Let we have a dataset

English
$\langle x_1 x_2 x_3 \rangle$

Urdu
$\langle y_1 y_2 \rangle$

⇓

We have to pass this to decoder

⇓

We will do **zero padding** for making sequence length equal. i.e

$\langle y_1, y_2, 0 \rangle$ (Shifted Right)

Now this output will be zero padded.

i) Input Embedding and Positional Encoding:

Let

Input = [4 5 6 7]

output = [1 2 3] ⇒ We will pad it

Output = [1 2 3 0]

Let every single element in output is represented by 4 dimensional vector

Output embedding:

[[0.1, 0.2, 0.3, 0.4],
[0.5, 0.6, 0.7, 0.8],
[0.9, 1.0, 1.1, 1.2],
[0.0, 0.0, 0.0, 0.0]]

- Now we will add **Positional encoding** to it. Let positional encoded vectors are all 0 So output embedding will remain the same.

② Linear Projection for Q, K and V

Create Query, Key and value vectors

Let $W_Q = W_K$, $W_V = I$ (Identity Matrix)

So,

Q = output Embedding • W_Q

K = output Embedding . W_K

V = output Embedding . W_V

AS $W_Q = W_K = W_V = I$ (Identity)

So,

Q = K = V = [[0.1, 0.2, 0.3, 0.4],
[0.5, 0.6, 0.7, 0.8],
[0.9, 0.10, 0.11, 0.12],
[0.0, 0.0, 0.0, 0.0]]

3) Scaled Dot product Attention Calculation:

$$Scores = Q * K^T / \sqrt{d_K}$$

∴ Dividing by $\sqrt{d_K}$ for scaling

$\therefore \sqrt{d_K} = 2$

$$Scores = Q * K^T / 2$$

We know how to do this from encoder

Scores = [[0.3, 0.7, 1.1, 0.0],
[0.7, 1.9, 3.1, 0.0],
[1.1, 3.1, 5.1, 0.0],
[0.0, 0.0, 0.0, 0.0]]

⇒ Masked Application:

It helps managing the structure of the sequence being processed and ensures the model behaves correctly during training and inferencing

Reasons:

① Handelling variable length sequences with padding mask.

Purpose:

(i) To handle sequences of different length in batch

(ii) To ensure that padding sequences/tokens which are added to make sequences of uniform length don't affect model prediction.

e.g

Input → Sequence 1 → [1, 2, 3]

Output → Sequence 2 → [4, 5]

↓ Zero padding

[4, 5, 0]

↓

Influence the attention mechanism

↓

Lead to incorrect or biased predictions

Padding Mask ensures that padding tokens are ignored

Masking → Padding Mask
→ Look Ahead Mask

dding Mask for sequence 1: [1 1 1]
(As no zero padding)

dding Mask for sequence 2: [1 1 0]
(As last one is zero-padded

⇒ Look Ahead Mask

* Maintain auto regressive property
- To ensure that each position in the decoder output sequence can only attend to the previous position but no future position.

e.g

[4, 5, 0] ⟶ [1, 1, 0]

1D Padding Mask

Convert 1D to 2D Mask

Token 1 attends to 1, 2

Token 2 attends to 1, 2

Token 3 can't attend anything

1	1	0
1	1	0
0	0	0

Padding Mask

* Look Ahead Mask:

[[1 0 0]
 [1 1 0]
 [1 1 1]]

(As we don't need future context)

⇒ Combine padding and look ahead mask:

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

(Do element wise multiplication

[[1, 0, 0],
 [1, 1, 0],
 [0, 0, 0]]

⇒ Combine Mask

- **Mask:**
  Now let's compute mask for the example we are doing

Scores: [[0.3, 0.7, 1.1, 0.0],
[0.7, 1.9, 3.1, 0.0],
[1.1, 3.1, 5.1, 0.0],
[0.0, 0.0, 0.0, 0.0]]

Look ahead Mask = [[1 0 0 0]
[1 1 0 0]
[1 1 1 0]
[1 1 1 1]]

Padding Mask extended to 2D format:

[[1, 1, 1, 0],
[1, 1, 1, 0],
[1, 1, 1, 0],
[0, 0, 0, 0]]

Combined mask = Look ahead Mask * Padding Mask

We will do element wise multiplication

$$\begin{bmatrix} [1, 0, 0, 0], \\ [1, 1, 0, 0], \\ [1, 1, 1, 0], \\ [0, 0, 0, 0] \end{bmatrix}$$

⇓ (wherever there is 0 mark $-\infty$) (Add $-\infty$ to 0) (As it zero out the influence when softmax is applied)

$$\begin{bmatrix} [1, -\infty, -\infty, -\infty], \\ [1, 1, -\infty, -\infty], \\ [1, 1, 1, -\infty], \\ [-\infty, -\infty, -\infty, -\infty] \end{bmatrix}$$

Masked score:

Now we will do element wise multiplication of scores and mask

$$\begin{bmatrix} [0.3, -\infty, -\infty, -\infty], \\ [0.7, 1.9, -\infty, -\infty], \\ [1.1, 3.1, 5.1, -\infty], \\ [-\infty, -\infty, -\infty, -\infty] \end{bmatrix}$$

⇛ Apply softmax

Softmax score =
Softmax (masked scores)

Softmax score = [[1.0, 0.0, 0.0, 0.0, 0.0],
[0.3, 0.7, 0.0, 0.0, 0.0],
[0.1, 0.3, 0.6, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0, 0.0]]

⇛ Weighted sum of values:

Attention output = Softmax scores * V

V → Value Vectors

⇒ Encoder Decoder Multi-head Attention:

① Encoder output → Sending Attention vectors K and V.

② Masked Multihead → Sending attention vector Q

So, encoder-decoder multi-head attention will get K and V (key and value vectors) from encoder and Q (query vector) from masked multi-head attention, and then it will perform all the things similarly as performed by encoder multi-head attention and output passed to Add & Norm layer.

A timestamp=1 Some token (start token) will get passed to masked multi-head attention and output will be generated, for t=2 previous output will get passed to decoder.

Linear

⇒ The final and Softmax Layer:

- We get vectors as output from decoder. In this layer we will see how we will convert output vectors to words.

Linear Layer:

The Linear Layer is a simple fully connected neural network that projects the vector produced by decoders.

- It generates a large vector called **logits vector**

- **Logits vector size = model vocabulary size**

Softmax Layer:

Softmax Layer turns those scores into probabilities.

- The cell with highest probability

is choosen and the word associated with it is produced as output.

- Then we will calculate loss function and back propogate to reduce the loss.