Residual Connection (Skip Connection) :-

A Residual connection means : instead of only passing the transformed output of a layer, we also add the original input back to it.

$$\text{Output} = F(x) + x$$

Where,

x = input to the layer

$F(x)$ = output of the layer (after transformation)

Why ?

1. Helps gradients flow → reduces Vanishing gradient problem.

2. Keeps original information → Model doesn't forget the input.

3. Easier training → Model can learn "adjustments" (residuals) instead of the whole mapping.

Example :

If input = x = 5

and the layer computes $F(x) = 2$

Residual output = $F(x) + x = 2 + 5 = 7$.

In Transformers : Every Attention and Feedforward block has a residual connection around it, followed by LayerNorm.

Normalization in Transformers :

Keeps activations stable → avoids Exploding / Vanishing gradients.

Balances features so no token dominates.

Works with residual connections to stabilize sums.

uses LayerNorm (not BatchNorm) → better for sequence data.

In Short : Normalization = stability + smooth training + better covergonce.

Batch Normalization (Vs) Layer Normalization :

1. Batch Normalization :

⇒ Where it normalizes : Across the batch dimension (Example in the mini-batch).

⇒ How its work :
For each feature (dimension of the hidden layer), it computes mean and Variance across all Example in the batch.

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{i,j}, \quad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_{i,j}-\mu_j)^2$$

The Normalize Each feature :

$$\hat{x}_{i,j} = \frac{x_{i,j}-\mu_j}{\sqrt{\sigma_j^2+\epsilon}}$$

Intuition : Make each feature dimension have

mean 0 and Variance 1 across the batch.

Best for : computer Vision (CNNs), where batches are large and features behave consistently.

Example :

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Normalize across the batch dimension (rows), per feature (column).

Column 1 (features across samples) : [1, 4]

=> Mean = $(1+4)/2 = 2.5$

=> Variance = $((1-2.5)^2 + (4-2.5)^2)/2$

$= (2.25 + 2.25)/2 = 2.25$

=> Variance =

=> Std = $\sqrt{2.25} = 1.5$

=> Normalized = $[(1-2.5)/1.5, (4-2.5)/1.5]$

$= [-1.0, 1.0]$

Column 2 (features across samples) : [2, 5]

Mean = 3.5 , Variance = 2.25 , Std = 1.5

Normalized = $[(2-3.5)/1.5, (5-3.5)/1.5]$

$= [-1.0, 1.0]$

Column 3 (features across samples) : [3, 6]

Mean = 4.5, Variance = 2.25, Std = 1.5

Normalized = [(3-4.5)/1.5, (6-4.5)/1.5]

= [-1.0, 1.0]

BN Result :-

$$\begin{bmatrix} -1 & -1 & -1 \\ 1 & 1 & 1 \end{bmatrix}$$

Layer Normalisation :-

⇒ Where it normalizes : Across the features (hidden dimensions) of a single Example.

⇒ How it works :

for each sample, compute mean and Variance across all hidden units in that layer.

$$\mu_i = \frac{1}{H} \sum_{j=1}^{H} x_{i,j}, \quad \sigma^2 = \frac{1}{H} \sum_{j=1}^{H} (x_{i,j} - \mu_i)^2$$

Then normalize :

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

Intuition : Each token's hidden Vector is normalized independently, across its features.

Best for : NLP models (Transformers, BERT, GPT) where batch sizes can Vary and sequence length matters more than batch statistics.

Example:

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Normalize across features (columns) for each sample (row).

Row 1 (features = [1, 2, 3]):

Mean = $(1 + 2 + 3)/3 = 2$

Variance = $((1-2)^2 + (2-2)^2 + (3-2)^2)/3$

$= (1 + 0 + 1)/3 = 0.67$

Std ≈ 0.82

Normalized = $[(1-2)/0.82, (2-2)/0.82, (3-2)/0.82]$

$\approx [-1.22, 0, 1.22]$

Row 2 (feature = [4, 5, 6]):

Mean = $(4 + 5 + 6) + /3 = 5$

Variance = $((4-5)^2 + (5-5)^2 + (6-5)^2)/3$

$= (1 + 0 + 1)/3 = 0.67$

Std = 0.82

Normalized = $[(4-5)/0.82, (5-5)/0.82,$

$(6-5)/0.82] \approx [-1.22, 0, 1.22]$

LN Result:

$$\begin{bmatrix} -1.22 & 0 & 1.22 \\ -1.22 & 0 & 1.22 \end{bmatrix}$$

Why Transformers uses Layer Normalization (and not Batch Normalization)

Batch Normalization :

=> looks at all samples in a batch and normalizes them together.

=> Work great in images (large batches, pixels are independent

=> problem in NLP / Transformers → batch size can be small and sentence lengths vary → unstable

Layer Normalization :

=> Looks at one token (Word) at a time and normalizes across its features.

=> Doesn't care about batch size.

=> Perfect for text → Keeps each word's hidden state stable.

Simple Analogy :

BatchNorm : " class average " → depends on how many students are in the class.

LayerNorm : " your own score normalized against your subjects " → doesn't care about class size, always consistent.

Inshort : Transformers use LayerNorm because :

1. Works even with small batches.
2. Works with variable - length sentences.
3. Same behavior in training & testing.

Feed-Forward Network (FNN) in Transformers:

The FNN is a small neural network inside each Transformer block. It processes each token's Vector individually (no interaction with other tokens).

FFN in Deep learning (vs) FNN in Transformers:

Aspect	FNN in DL	FNN in Transformers
purpose	Maps input → output (classification, regression, etc)	Refines each Token's representation after attention.
Input	Whole dataset features (Eg: image pixels, tabular data, word embeddings)	One token Vector at a time (Eg: 512-d Vector)
Application	Applied once to Entire input	Applied independently to every token (same weight shared)
Structure	Fully connected layers + activation (Eg: ReLU, GELU)	Same: 2 linear layers + activation (usually RELU / GELU)
Hidden size	Depends on task (user-defined)	usually 4 x layer than model dimension (Eg: 512 → 2048 → 512)
Role	End-to-End learner for tasks	Adds non-linearity + Expressive power inside Transformer block
Analogy	Whole system brain (thinking & deciding)	Individual token's private thought after group discussion (attention).

Masked Multi-Head Attention :-

⇒ used in decoder's first attention layer.

Problem it solves :

While generating a sequence (Eg : a translation), the model must not peek at future words.

Example : if we're predicting the 3rd word ; the model should only look at words 1 and 2, not 4 and 5, ...

How it works :

Normal self-attention lets each token attend to all tokens.

Masking = block future tokens by setting their attention scores to $-\infty$.

After softmax → probabilities for those future tokens become 0.

Math :

Self-attention score :

$$\text{Score}(Q, K) = \frac{QK^T}{\sqrt{d_k}}$$

Masked Version :

$$\text{Score}(Q, K) = \begin{cases} \frac{QK^T}{\sqrt{d_k}}, & j \le i \\ -\infty, & j > i \end{cases}$$

Where,

i = current token position

j = future position

Why we must not peek at future words ?

1. Casuality in language :

If it "peaked ahead", it would be cheating - it woumldn't be true generation.

2. During Inference (generation) :

At test time, the model doesn't know the future (because it hasn't generated it yet).

So training must mimic this condition - that's why we ~~enforce~~ enforce masking during training training.

3. Analogy :

Imagine predicting the next word in a sentence game.

If you secretly saw the whole sentence beforehand, you'd ace it - but you wouldn't have learned how to predict.

Masking ensures the model learns to predict step by step.

Cross - Attention :

used in decoder after masked self - attention.

Problem it solves :

Decoder needs to use encoder's knowledge (the input sentence) while generating output.

How it works :

Query (Q) = decoder hidden states (target tokens)
Key (K) & Values (V) = encoder outputs (input context)

This allows each decoder token to look at the entire encoded input.

Math :

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

But here ;

Q = decoder states

K, V = encoder outputs

This way, decoder words "align" with relevant source words.

Transformer Architecture : Encoder & Decoder

Transformers have two main parts :

Encoder → understands the input

Decoder → Generates the output

This Design is mainly used in sequence to sequence tasks (like translation).

Encoder :

Input : sequence of tokens (Eg : Sentence in English)

Steps inside Each Encoder block :

1. Embedding + Positional Encoding

2. Multi-Head Attention

3. Residual + LayerNorm

4. Feed-Forward Network (FNN)

5. Residual + Layer Norm (again)

This block repeated N times (Eg: 6 to 12 layers)

Output: Context-rich token Embeddings.

Decoder:

Input: previously generated tokens (Eg: partial translated sentence)

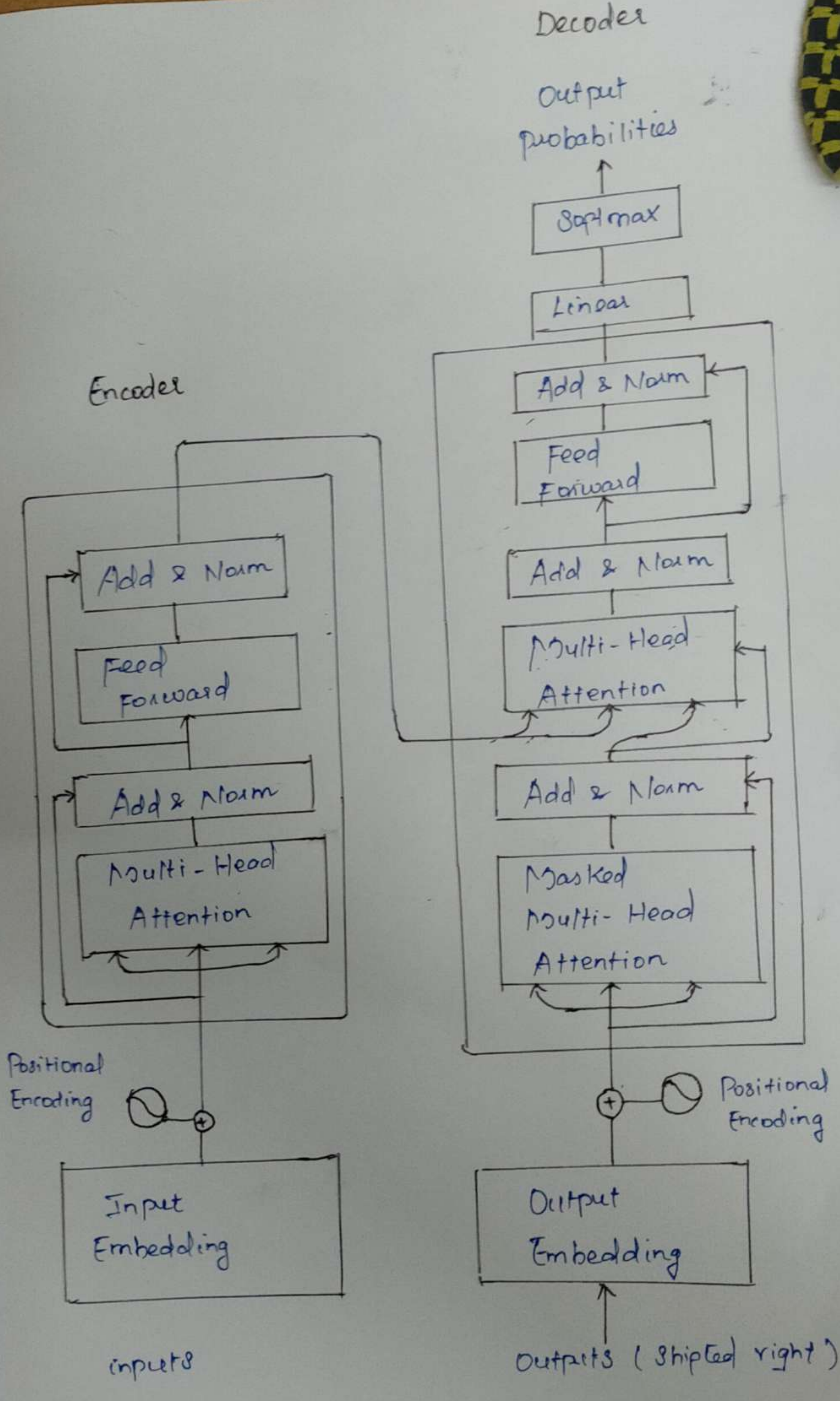
Goal: Generate the next token.

Steps inside each decoder block:

1. Embedding + Positional Encoding.
2. Masked Multi-Head Self-Attention
2. Residual + Layer Norm
4. Cross Attention (Key Difference from Encoder)
5. Residual + LayerNorm
6. Feed-Forward Network (FNN)
7. ~~Rea~~ Residual + Layer Norm.

Repeated N times.

Final Layer: Linear + Softmax →
predicts probability of next word.


Decoder
Output probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Positional Encoding
Output Embedding
Outputs (Shifted right)
Encoder
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Positional Encoding
Input Embedding
inputs