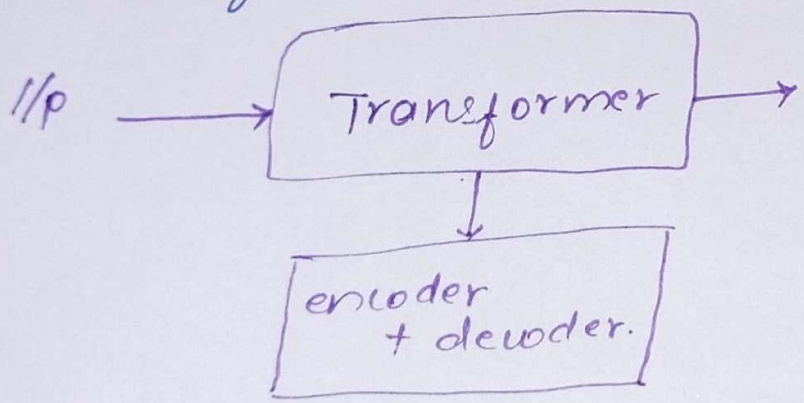
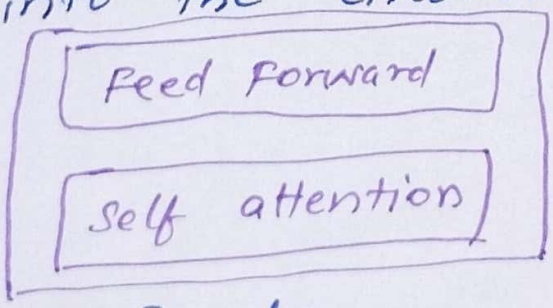
Transformers.

Transformer in natural language processing (NLP) is a type of deep learning model that use self-attention mechanism to analyse and process natural language data. They are encoder-decoder models that can be used for many applications including neural machine translation.



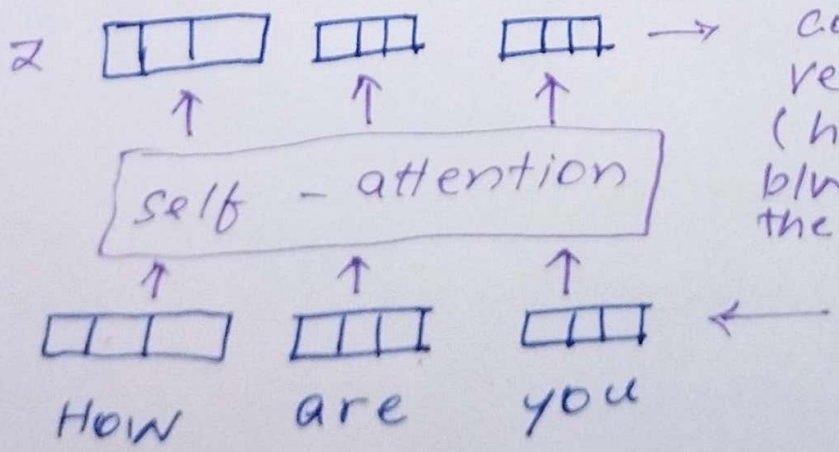
a view into the encoder will be.



Encoder.

Note: There will be a stack of encoder and decoder of N. i.e 6 in paper.

Dive into encoder's workflow.



Working of Self-attention Mechanism:

→ it is also known as scaled-dot product attention that allows the model to weigh the importance of different tokens in the input sequence relative to each other.

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

What does "it" in this sentence refer to? Is it referring to the street or the animal? Self-attention allows it to associate "it" with animal.

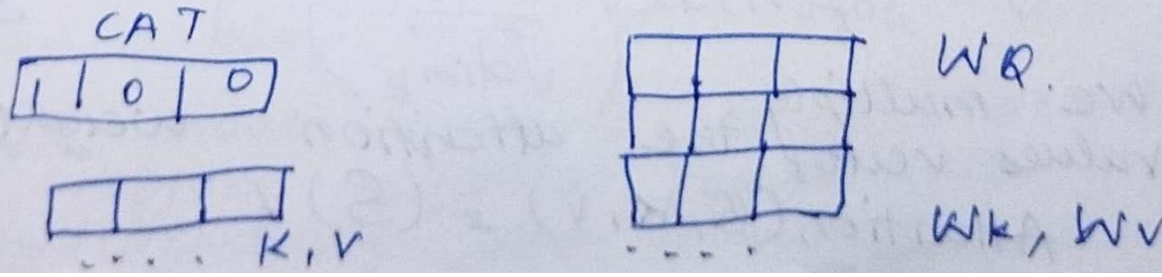
1. For each token, we have to compute three vectors:
   Query (Q), key (K) and Value (V).

Query (Q): represents the token being processed.

key (K): represents the token to be compared with.

value (V): Represents the information to be aggregated.

We create Q, K, V by multiplying the embeddings by learned weight matrices. W_Q, W_K, W_V.



For each token in the input sequence we have to calculate the Attention score.

Now that, We've obtained Q, K and V for all the token using dot product with learned weights W_Q, W_K and W_V respectively. Let's calculating attention score.

1. For each token the input sequence, find. $Q \cdot K^T$

2. divide the obtained value by the square root of dimension of key vector.

$$\frac{Q \cdot K^T}{\sqrt{dim_K}}$$ ← This is scaling (~~normalizing~~)

3. Apply Softmax.
Softmax is a mathematical function that converts a list of numbers into probabilities.
Takes a vector of raw numbers and transform them into a probability distribution where all values sum to 1.

$$S(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

$$S = softmax\left(\frac{Q \cdot K^T}{\sqrt{dim_K}}\right)$$

4. We multiply the attention weights by values vector.

$$Attention(Q, K, V) = (S)V..$$

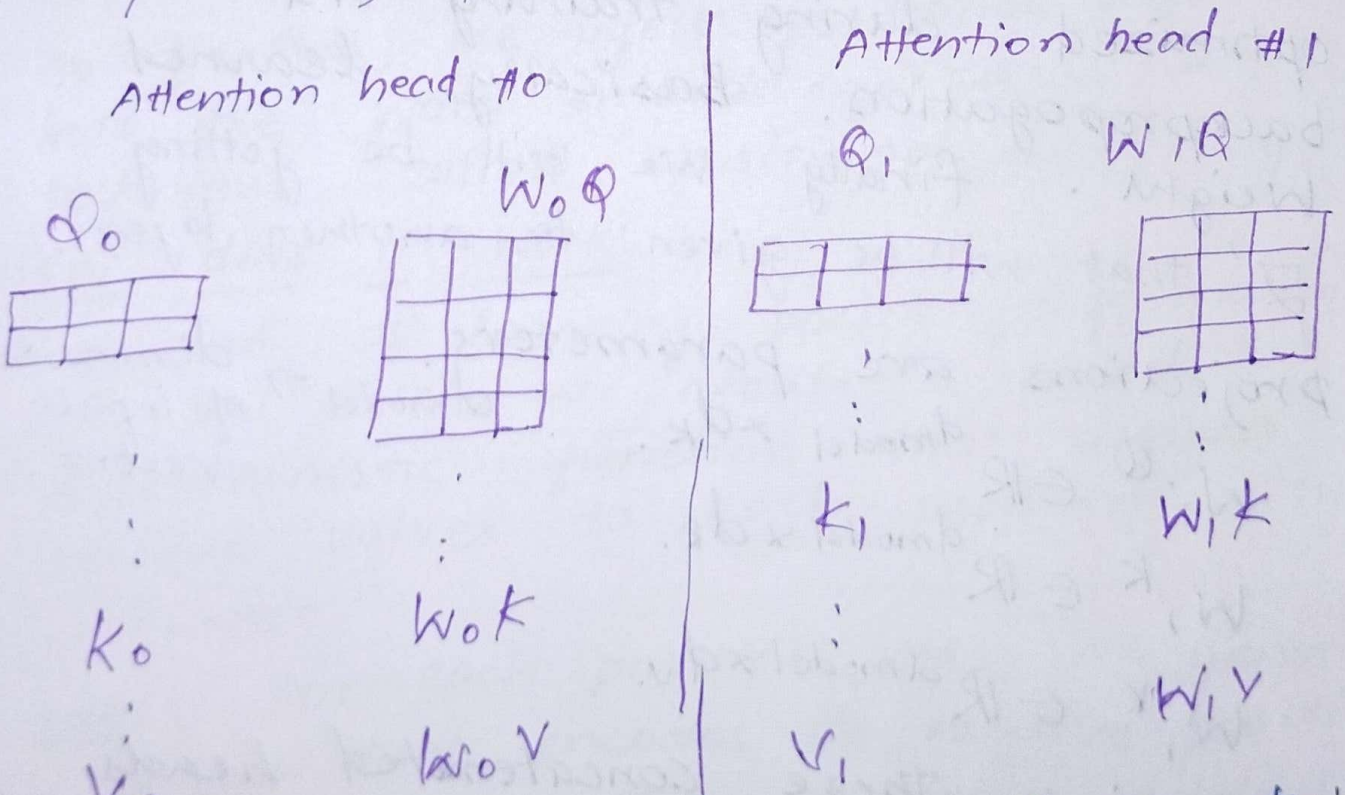
Multi-Head Attention.

Self-attention is powerful. but using only one attention mechanism has a limitation.

1. It learns only one type of relationship between words (or tokens)
2. A single attention head may miss out on some dependencies in complex sequence.

To overcome this, we split the attention mechanism into multiple heads, each attending to different parts of input independently.

Multi-head attention is that each head is independently parameterized.

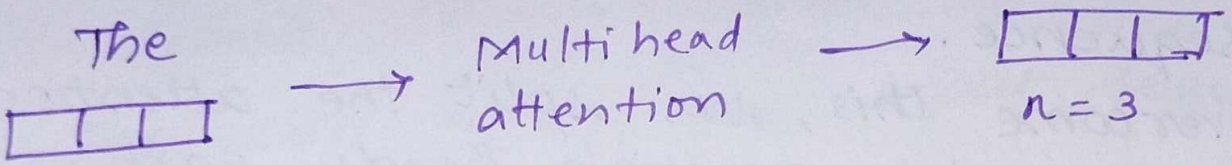


So we have multiple head like this each capture different relationship independently and parallely.

$MultiHead(Q, K, V) = Concat(head_1, \ldots head_h)W^O$

Where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

$W^O = ?$ W^O performs linear transformation on Concatenated heads.



W^O brings the concatenated head to same dimensions as the input embedding. It is initialized randomly (like all other weight matrices) and gets optimized during training via backpropogation. Basically, learned weight. finally we will be getting 'Z' that will be given to another layer.

projections are parameters.

$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$. d_{model} → dimension of input vector.

$W_i^K \in \mathbb{R}^{d_{model} \times d_k}$.

$W_i^V \in \mathbb{R}^{d_{model} \times d_v}$.

W^O projects these concatenated heads to dimension of input vector.

$Z \in \mathbb{R}^{d_{model}}$

Positional Encoding.

→ Since Transformers don't have recurrence (RNN) or convolutions (like CNN), they have no built-in way to understand the order of words in a sequence. That's where positional encoding (PE) comes to play.

→ Self-attention treats all token independently, it doesn't have any notion of sequence order.

"I love pizza"

"pizza love I"

A Transformer without positional encoding would treat these as identical.

To fix this, we inject positional information into the word embeddings using a technique called positional encoding.

How does it work!

Instead of learning position information like an RNN, we create a fixed, deterministic function that assigns unique values to each position in the sequence.

Idea: For each position pos, we generate a vector that encodes its relative position in the sequence. This vector is added to the word embeddings, so that model can distinguish different positions.

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

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

Where,

pos → position of the token in the sequence.

i → dimension index in the embedding.

d_{model} → total embedding dimension. (eg. 512 in original transformer)

Observations.

Even dimensions use 'sine'.

odd dimensions use 'cosine'.

This creates a pattern of smooth, periodic values.

Scaling with $10000^{\frac{2i}{d_{model}}}$

- Helps creates different wave length for different embedding dimension.
- Low-frequency signals capture global relationship
- High-frequency signals capture local relationship.

Decoder's Architecture.

The decoder is responsible for generating the output sequence. one token at a time. It is composed of

- → Input Embeddings.
- → positional encoding
- → Masked Multi-head self attention.
- → Encoder-decoder attention.
- → Feed-forward Neural Network (FFN)
- → Residual connection and layer Normalization.
- → Linear and softmax layer.

Masked Multi-head Self Attention.

What problem does it solves?

- → Standard self-attention allows each token to attend to all others.
- → In auto-regressive decoding, we don't want a token to see future tokens because that would be cheating. we want it to predict next token using past and itself.
- → MMSA modifies the sdf-attention mechanism by ensuring a token only sees itseft and past tokens.
- → It does this by introducing a making mechanism that forces future

attention scores to be $-\infty$, ensuring that softmax completly ignores future words.

Introducing the Mask.

To prevent a token at position i from attending to future positions $j > i$, we introduce a mask matrix M, where

$$M_{ij} = \begin{cases} 0 & \text{if } j \leq i \text{ (keep current \& past tokens} \\ -\infty & \text{if } j > i \text{ (mask future tokens)} \end{cases}$$

for a sequence of length 4, the mask look like

$$M = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Modifying attention,

$$A_{masked} = \frac{Q K^T}{\sqrt{d_k}} + M$$

Apply softmax after this.

→ The diagonal and past tokens remain unchanged.

→ The future tokens become large negative number (effectively 0 after softmax)

$$e^{-\infty} = 0$$

MMSA Algorithm.

1. Compute Queries, keys and values.

$$Q = XW^Q, \quad k = XW^k, \quad V = XW^k$$

$X \rightarrow$ input sequence.

2. compute attention scores.

$$A = \frac{Q k^T}{\sqrt{d_k}}$$

3. Apply the mask.

$$A_{masked} = A + M$$

4. Apply softmax

$$\alpha = softmax(A_{masked})$$

5. Compute Weighted sum of values:

$$Z = \alpha V.$$

6. Concatenate multiple heads and apply W^o.

$$MMSA(x) = Concat(Z_1, Z_2 .. Z_N) W^o.$$

Summary.

→ Masking is essential for decoder's self attention to prevent cheating.

→ Mask is added to attention scores before softmax.

→ It follows same steps as encoder except with masking applied.

→ It is used in decoder only since the encoder sees the full input at once.

Residual connection and Layer Normalization.

→ Residual connective, also called skip connections, are crucial in deep neural network, especially in transformers.

→ They help preserve information, improve gradient flow, and speed up training.

→ Without them, deep neural network can suffer from vanishing gradients, making learning difficult.

Why we need Residual connections?

→ In very deep network, gradients shrink during backpropagation, making weight updates ineffective.

→ Each layer transforms inputs non-linearly, which can degrade useful information as it moves through the network.

→ Without shortcuts, the network struggles to learn deeper representations effectively.

Residual connection solve these problems by allowing information to bypass certain layer and flow more efficiently.

Residual connection.

A residual connection adds the original input of a layer (or block) to its output before applying activation.

$$y = F(x) + x$$

Where,

$x \rightarrow$ input of the layer.

$F(x) \rightarrow$ transformed output from the layer.

$y \rightarrow$ final output after adding x back.

This allows the network to learn modifications rather than the complety new transformations.

In Transformer, every sub-layer (self Attention & Feed-Forward Networks) uses residual connections.

Each Transformer block consists of

1. Multi-Head Self-Attention (MHA) → RC
2. Feed-Forward Network (FFN) → RC

After adding the residual connection, we apply Layer Normalization.

Output = LayerNorm ($x + F(x)$)

Layer Normilization (used in Transformer)

Layer Normalization (LN) is a technique that stabilizes activations by normilizing them across the features of each individual sample.

For an input X of shape (B, d_{model}), LN works as follow.

1. Compute mean μ_L across all features for each sample.

$$\mu_L = \frac{1}{d_{model}} \sum_{j=1}^{d_{model}} x_j$$

2. Compute variance σ_L^2 across features.

$$\sigma_L^2 = \frac{1}{d_{model}} \sum_{j=1}^{d_{model}} (x_j - \mu_L)^2$$

3. Normalize Activations.

$$\hat{x_j} = \frac{x_j - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}}$$

4. Scale and shift using learnable parameters. γ and β.

$$y_j = \gamma \hat{x_j} + \beta$$

Where

- γ (scale) and β (shift) are trainable parameter.
- ϵ is a small constant to prevent division by zero.

Feed Forward Network (FFN)

The FFN is a simple neural network with two linear transformation and a non-linear activation function in between. it operates on each position (token) independently and identically.

$$FFN(x) = ReLU(x W_1 + b_1) W_2 + b_2$$

Where,

$x \rightarrow$ input to FFN. (output from self attention mechanism)

$W_1, b_1 \rightarrow$ Weights and biases of the first linear layer.

$W_2, b_2 \rightarrow$ Weights and biases of the second linear layer.

$ReLU \rightarrow$ activation function : $ReLU(x) = max(0, x)$

Key Features.

* The First linear layer typically expands the dimensionality of the input (eg. from 512 to 2048), and the second layer projects it back to the original dimension (eg. 2048 to 512). This expansion allow the model to learn more complex representation.

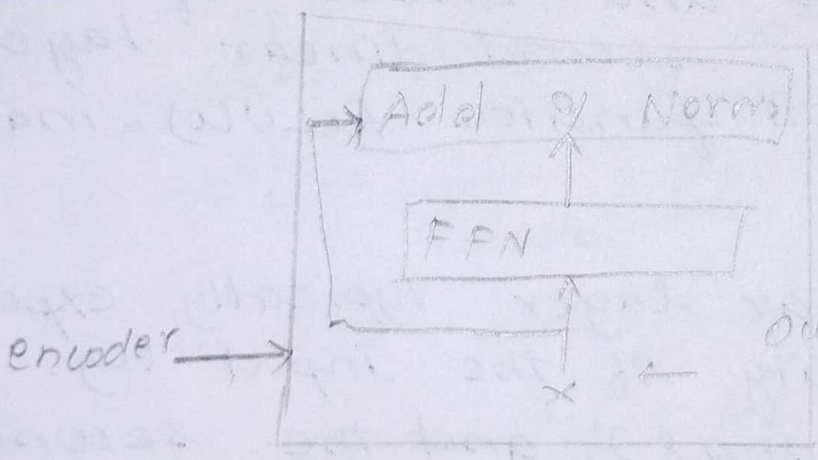
* The FFN is applied independently to each token in the sequence. This means it doesn't consider interactions between token, unlike self-attention.

- The ReLU activation introduces non-linearity enabling the model to capture more complex pattern.

importance:

→ It adds capacity to the model, allowing it to learn intricate transformation of the data after the self-attention mechanism has aggregated info from other ~~network~~ tokens.

→ It provides a way to model complex, non-linear relationship within the data.



$$\text{Output} = \text{LayerNorm}\left(x + \text{FFN}(x)\right)$$

LayerNorm → Layer Normalization.

FFN(x) → Apply activation

x → Residual/Skip connection.

Linear and softmax Layer.

The Linear and softmax layer is the final two layers in the transformer Architecture. These two layers work together to convert numerical representation into probability distributions over the vocabulary, which is essential for generating meaningful output.

Linear Layer.

* The Linear Layer is a simple fully connected neural network that projects the vector produced by the stack of decoders (Logits vectors).
* It maps the transformed feature vectors to the output vocabulary space. This is done by Linear transformation.

$$Z = XW + b.$$

Where,

$X \rightarrow$ input (output of last transformer layer)

$W \rightarrow$ learnable weight matrix of shape (d_{model}, V)

$b \rightarrow$ bias vector of shape (V)

$Z \rightarrow$ logits output (before applying softmax)

* Reduces dimension from d_{model} to V (vocab size)
* create logits that softmax converts into probability.

Softmax Layer.

After the linear transformation, we get raw scores (logits) for each token in the vocabulary. But these numbers doesn't mean anything on their own.

[Recap]
Softmax is a function that converts a vector of raw scores (logits) into a probability distribution. It ensures that:

- All output values are between 0 and 1
- The sum of all output values is 1.

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

where,

$z_i \rightarrow$ logit for the i^{th} word.

$N \rightarrow$ The total number of elements in z.

example.

$$z = [2.0, 1.0, 0.1]$$

$$\text{Softmax}(z) = \left[\frac{e^{2.0}}{e^{2.0}+e^{1.0}+e^{0.1}}, \frac{e^{1.0}}{\dots}, \frac{e^{0.1}}{\dots}\right]$$

$$\approx [0.66, 0.24, 0.10]$$

∴ The model predicts the first token with probability of 66%.

Workflow of Linear and Softmax Layer:

1. compute Logits.
each token representation is passed through a linear layer to map it to a vocab-size space.

d_{model} (512) → Linear layer → Vocab size (30,000)

2. Apply Softmax.
Softmax converts logits into probability so we can pick the most likely word.

Logits before Softmax:
["hello": 5.2, "world": 2.3, "cat": 1.8]

After Softmax.
["hello": 0.91, "world": 0.05, "cat": 0.4]

hello has highest probability.

let say, the input sentence.
"The cat sat on the..."

logits:
["table": 6.3, "mat": 5.9, "dog": 2.1, "house": 1.5]

Softmax:
["table": 0.50, "mat": 0.45, "dog": 0.03, "house": 0.02]

Since 'table' has highest probability (0.50), it's the predicted next word.

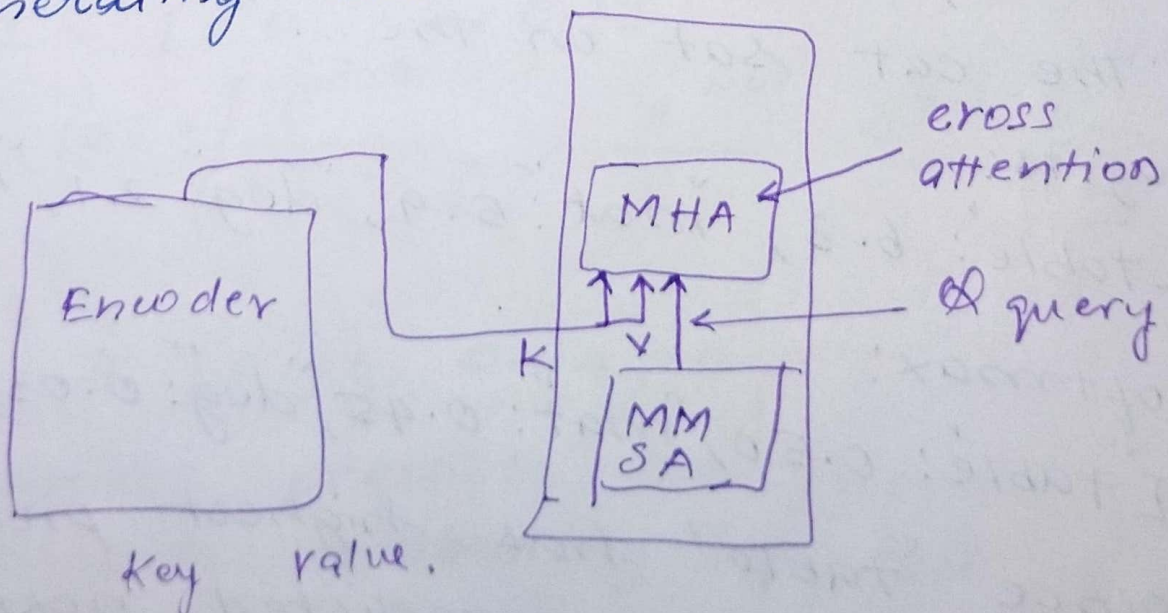
Cross-Attention (Encoder-Decoder Attention)

Cross-Attention allows the decoder to focus on specific parts of the input sequence (processed by the encoder) while generating each word. This mechanism helps the model translate or generate text accurately by leveraging the context provided by the encoder.

In cross attention

- Queries (Q) comes from decoder.
- Keys (K) and Values (V) come from the encoder's output.

This means the decoder is querying the encoded input information to decide which part of the input is relavant for generating the next word.



Mathematical Breakdown

Given:

X_{enc}: Encoder's output

X_{dec}: Decoder's hidden states

B: Batch size

N: Sequence length of the input (encoder)

M: Sequence length of the target (decoder)

d_{model}: Dimension of model.

Step 1: Linear projection.

Compute Queries, keys and values.

$$Q = X_{dec} W^Q, \quad K = X_{enc} W^K, \quad V = X_{enc} W^V$$

$W^Q \rightarrow$ projection matrix for Queries.

$W^K \rightarrow$ projection matrix for keys

$W^V \rightarrow$ " " values.

Step 2. compute Scaled dot-product Attention.

$$A = \frac{Q \cdot K^T}{\sqrt{d_k}}$$

$d_k \rightarrow$ dimension of key vector.

Step 3. Apply softmax to obtain Attention weights.

$$\alpha = \text{softmax}(A)$$

Step 4. Weighted sum of values.

$$Z = \alpha V.$$

Why is cross-Attention Important?

In translation task.

→ Encoder compresses the entire input sequence's meaning.

→ Decoder uses cross-attention to selectively focus on relavant words/phrases from the input while generating each word in target language.

Multi-Head cross-Attention.

Instead of a single attention mechanism, we use multiple heads to capture different aspects of the input sequence.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(h_1, h_2 .. h_n)W^O$$

Where each head is computed as

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$