

# Transformer Model Summary



Attention Is All You Need.pdf  
PDF

Extract and explain contents clearly

The uploaded document "Attention Is All You Need" outlines the Transformer model, an attention-based architecture for sequence transduction tasks, replacing recurrent and convolutional layers with self-attention mechanisms. Here's an explanation of its key points:

---

## 1. Introduction

- Traditional sequence models like RNNs and CNNs are less parallelizable due to sequential computation constraints.
  - The Transformer introduces a fully attention-based architecture, achieving faster training and state-of-the-art results in machine translation tasks (e.g., English-German BLEU score: 28.4).
- 

## 2. Model Architecture

- **Encoder-Decoder Framework:** The Transformer follows this structure:
    - **Encoder:** Maps input sequences to continuous representations.
    - **Decoder:** Generates output sequences auto-regressively.
  - Both encoder and decoder consist of:
    - **Multi-Head Self-Attention Mechanism:** Captures relationships between different positions in a sequence.
    - **Feed-Forward Networks (FFN):** Applied to each position independently.
-

### 3. Attention Mechanism

- **Scaled Dot-Product Attention:**
    - Computes attention scores by taking dot products of query and key vectors, scaled by  $\sqrt{d_k}$ , and applies softmax to obtain weights.
    - Efficient due to optimized matrix operations.
  - **Multi-Head Attention:**
    - Uses multiple attention heads to focus on different parts of the sequence in parallel.
    - Improves the model's ability to capture diverse information.
- 

### 4. Positional Encoding

- Since the Transformer lacks recurrence, positional encodings are added to token embeddings to convey sequence order.
  - Uses sinusoidal functions to generate encodings, enabling extrapolation to unseen sequence lengths.
- 

### 5. Training and Regularization

- **Data:** Trained on large datasets like WMT 2014 for English-German and English-French translations.
  - **Hardware:** Efficient parallel training on GPUs (e.g., 12 hours for the base model on 8 P100 GPUs).
  - **Optimizer:** Uses the Adam optimizer with a custom learning rate schedule.
  - **Regularization:**
    - Dropout applied at various points.
    - Label smoothing (e.g., 0.1) improves BLEU scores.
-

## 6. Results

- Achieved superior BLEU scores compared to existing models while significantly reducing training costs.
  - Demonstrated effectiveness in other tasks like English constituency parsing.
- 

## 7. Advantages of Self-Attention

- Parallelization: Enables faster computation compared to recurrent models.
  - Shorter Path Lengths: Facilitates learning of long-range dependencies.
  - Interpretability: Attention heads often capture syntactic and semantic features.
- 

## 8. Conclusion

- The Transformer sets a new standard for sequence transduction models.
  - Future research aims to extend attention mechanisms to other modalities (e.g., images, audio) and improve efficiency for larger inputs.
- 

If you'd like a specific section or concept further elaborated, let me know!

[explain all sections](#)

## Detailed Explanation of All Sections in "Attention Is All You Need"

---

### 1. Introduction

This section discusses the limitations of traditional sequence models:

- **Recurrent Neural Networks (RNNs):** Effective for sequence modeling but slow due to sequential dependencies.

- **Attention Mechanisms:** Allow models to relate elements in sequences without regard to their distance, but are typically used alongside RNNs or CNNs.

The **Transformer** is introduced as an entirely attention-based architecture that removes recurrence and convolutions:

- **Advantages:**
    - Faster training through parallelism.
    - Superior performance, demonstrated in machine translation tasks (e.g., WMT 2014 English-German).
- 

## 2. Background

- Traditional architectures, such as **Extended Neural GPU**, **ByteNet**, and **ConvS2S**, utilize convolutions for parallelism.
- These models struggle with long-range dependencies because the number of operations increases with distance between elements.
- **Self-Attention:** Relates all positions in a sequence with a constant number of operations, enabling efficient computation.

The **Transformer** stands out as the first model to rely entirely on self-attention for sequence transduction, offering both performance and simplicity.

---

## 3. Model Architecture

The Transformer consists of an **encoder-decoder** structure:

### 3.1 Encoder and Decoder Stacks

- **Encoder:**
  - Composed of 6 identical layers.
  - Each layer includes:
    - Multi-head self-attention mechanism.

- Position-wise feed-forward network (FFN).
  - Residual connections and layer normalization.
  - **Decoder:**
    - Also has 6 identical layers.
    - Adds a third sub-layer: **encoder-decoder attention**, allowing the decoder to focus on relevant encoder outputs.
    - Includes masking in self-attention to ensure the model predicts tokens sequentially.
- 

## 3.2 Attention

### 3.2.1 Scaled Dot-Product Attention:

- Computes attention as a weighted sum of values.
- Formula:

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

- Scaled by  $\sqrt{d_k}$  to prevent large dot products from diminishing softmax gradients.

### 3.2.2 Multi-Head Attention:

- Projects queries, keys, and values into multiple subspaces and applies attention independently in each.
- Outputs are concatenated and linearly transformed.

Applications:

1. **Encoder-Decoder Attention:** Decoder attends to encoder outputs.
  2. **Encoder Self-Attention:** Each token in the input sequence attends to all others.
  3. **Decoder Self-Attention:** Ensures predictions depend only on previous tokens.
- 

## 3.3 Position-Wise Feed-Forward Networks

- Two fully connected layers with ReLU activation.

- Applied independently to each position.

---

### 3.4 Embeddings and Softmax

- Tokens are mapped to vectors via learned embeddings.
  - Shared weight matrix for input/output embeddings and softmax transformation.
- 

### 3.5 Positional Encoding

- Since the model lacks recurrence, positional encodings are added to embeddings.
- Uses sinusoidal functions to encode token positions:

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

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


---

## 4. Why Self-Attention

Self-attention is compared to recurrent and convolutional layers:

Aspect	Self-Attention	Recurrent Layers	Convolutional Layers
Path Length	Constant $O(1)$	Linear $O(n)$	Logarithmic $O(\log_k(n))$
Parallelism	High	Low	Moderate
Computational Cost	$O(n^2 \cdot d)$	$O(n \cdot d^2)$	$O(k \cdot n \cdot d^2)$

Advantages of self-attention:

- Parallel processing for faster computation.
  - Better handling of long-range dependencies.
- 

## 5. Training

### 5.1 Training Data:

- English-German (4.5M sentence pairs).
- English-French (36M sentence pairs).

### 5.2 Hardware:

- Trained on 8 NVIDIA P100 GPUs.
- Base model: 100k steps in 12 hours.
- Big model: 300k steps in 3.5 days.

### 5.3 Optimizer:

- Adam optimizer with learning rate schedule:

$$lrate = d_{model}^{-0.5} \cdot \min(\text{step}^{-0.5}, \text{step} \cdot \text{warmup\_steps}^{-1.5})$$

### 5.4 Regularization:

- Dropout applied to sub-layer outputs and embeddings.
  - Label smoothing ( $\epsilon_{ls} = 0.1$ ) improves BLEU scores.
- 

## 6. Results

- **Machine Translation:**
    - English-German: BLEU score of 28.4 (outperforming ensembles).
    - English-French: BLEU score of 41.8.
  - **Efficiency:** Training cost significantly lower than competing models.
- 

## 7. Model Variations

- Performance analysis with different configurations:
  - Increasing model size improves performance.
  - Dropout prevents overfitting.

- Sinusoidal and learned positional encodings yield similar results.
- 

## 8. Generalization to Other Tasks

- English Constituency Parsing:
    - Applied to Penn Treebank.
    - Outperformed most models, even in low-data settings.
- 

## 9. Conclusion

- The Transformer sets a new standard for sequence transduction.
  - Plans for future work include:
    - Extending to other input/output modalities (e.g., images, video).
    - Investigating restricted attention for large inputs.
- 

This detailed explanation covers every section of the paper. Let me know if you need further clarification on any part!

explain model architecture clearly

## Model Architecture of the Transformer

The Transformer architecture is a revolutionary model for sequence transduction, designed entirely around **attention mechanisms** without using recurrence or convolution. It comprises an **encoder-decoder** structure, which is detailed below:

---

### 1. Overview of Encoder-Decoder Structure



The Transformer architecture processes input sequences into continuous representations via the **encoder** and generates output sequences using the **decoder**.

- **Encoder:**
    - Maps an input sequence  $(x_1, x_2, \dots, x_n)$  to a sequence of continuous representations  $(z_1, z_2, \dots, z_n)$ .
  - **Decoder:**
    - Generates an output sequence  $(y_1, y_2, \dots, y_m)$  one token at a time, using the encoder's representations and previously generated tokens.
- 

## 2. Encoder Stack

The encoder is composed of **6 identical layers**, and each layer has two main components:

### 2.1 Multi-Head Self-Attention Mechanism

- Allows the encoder to focus on different parts of the input sequence.
- Each position in the input sequence can attend to all other positions to capture dependencies, regardless of distance.

### 2.2 Feed-Forward Network (FFN)

- A fully connected network applied independently to each position.
- Formula:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- Two linear transformations with a ReLU activation in between.

### 2.3 Residual Connections and Layer Normalization

- Residual connections are added around each sub-layer:
$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x))$$
  - Helps stabilize training and improve gradient flow.
- 

## 3. Decoder Stack

The decoder is also made up of **6 identical layers**, each containing three sub-components:

### 3.1 Masked Multi-Head Self-Attention

- Similar to the encoder's self-attention but with **masking**.
- Masking ensures predictions for a position  $i$  depend only on positions  $< i$ , preserving auto-regression.

### 3.2 Encoder-Decoder Attention

- Queries come from the decoder, while keys and values come from the encoder outputs.
- Helps the decoder focus on relevant parts of the input sequence.

### 3.3 Feed-Forward Network (FFN)

- Same as in the encoder, applied independently to each position.
- 

## 4. Attention Mechanisms

The Transformer uses attention in three main ways:

1. **Encoder Self-Attention:** Each token in the input attends to all other tokens.
2. **Decoder Self-Attention:** Each token in the output attends to previous tokens (masked).
3. **Encoder-Decoder Attention:** Decoder queries attend to encoder outputs.

### 4.1 Scaled Dot-Product Attention

- Attention is computed as a weighted sum of values:

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

- $Q$ : Queries,  $K$ : Keys,  $V$ : Values.
- The scaling factor  $\sqrt{d_k}$  ensures stable gradients when softmax is applied.

### 4.2 Multi-Head Attention

- Instead of a single attention mechanism, queries, keys, and values are projected into multiple subspaces (heads).
- Formula:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O$$

- Each head computes attention independently, capturing diverse features.
  - Outputs are concatenated and linearly projected.
- 

## 5. Positional Encoding

Since the Transformer lacks recurrence or convolution, positional encoding is added to embeddings to represent token order.

- **Sinusoidal Function:**

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

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

- This encoding allows the model to generalize to sequences longer than those seen during training.
- 

## 6. Embeddings and Softmax

- **Input Embeddings:** Tokens are converted into continuous vectors.
  - **Output Layer:** Decoder outputs are transformed into probabilities over the vocabulary using a softmax layer.
- 

## 7. Key Advantages of the Transformer's Architecture

- **Parallelization:** All tokens in a sequence are processed simultaneously, unlike RNNs.
  - **Efficiency:** Multi-head attention and feed-forward networks allow faster computation.
  - **Flexibility:** Self-attention captures long-range dependencies effectively.
-

## Summary of Flow

1. **Input Sequence** → Embedded and Encoded → Encoder Output.
2. **Output Sequence** → Embedded and Decoded (using Encoder Outputs) → Final Sequence.

This architecture is the foundation for modern AI models like BERT, GPT, and others. Let me know if you'd like further elaboration on specific components!

explain multihead self attention mechanism clearly with matrices and dimensions and example

## Multi-Head Self-Attention Mechanism: Explained

The **multi-head self-attention** mechanism is a core part of the Transformer architecture. It allows the model to focus on different parts of the sequence simultaneously, capturing various types of relationships between tokens. Here's a detailed explanation with matrices, dimensions, and an example.

---

### 1. Overview

At its core, the attention mechanism computes a weighted sum of value vectors ( $V$ ) based on the similarity between queries ( $Q$ ) and keys ( $K$ ).

**Attention Formula:**

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

Where:

- $Q$ : Query matrix ( $n \times d_k$ )
- $K$ : Key matrix ( $n \times d_k$ )
- $V$ : Value matrix ( $n \times d_v$ )
- $n$ : Sequence length (number of tokens).
- $d_k$ : Dimensionality of keys and queries.
- $d_v$ : Dimensionality of values.

**Multi-Head Attention:** Instead of computing attention once, the mechanism uses multiple **attention heads**:

1. Input matrices are split into multiple **subspaces** (smaller dimensions).
  2. Attention is computed independently for each head.
  3. Outputs are concatenated and linearly transformed.
- 

## 2. Step-by-Step Breakdown

### Step 1: Input Matrices

Assume:

- Sequence length  $n = 4$  (e.g., "The cat sat down").
- Embedding size  $d_{\text{model}} = 8$ .

For simplicity, let's use these matrices (random values):

$$\text{Input Embeddings (X)} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

**Dimensions:**

- $X : (n \times d_{\text{model}}) = (4 \times 8)$ .
- 

### Step 2: Linear Projections

The input embeddings are linearly projected into  $Q$ ,  $K$ , and  $V$  using learned weight matrices.

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

Where  $W^Q, W^K, W^V$  are weight matrices of size:

$$W^Q, W^K, W^V : (d_{\text{model}} \times d_k) = (8 \times 4)$$

For  $h = 2$  heads, the embeddings are split across the heads:

- $d_k = d_v = d_{\text{model}}/h = 4$ .

Example:

$$W^Q = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Compute  $Q$ :

$$Q = XW^Q = \begin{bmatrix} 4 & 0 & 4 & 0 \\ 0 & 4 & 0 & 4 \\ 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Similar computations yield  $K$  and  $V$ .

---

### Step 3: Scaled Dot-Product Attention

For each head, compute attention scores.

#### 1. Dot Product:

$$\text{Score} = QK^\top$$

Example:

$$Q = \begin{bmatrix} 4 & 0 & 4 & 0 \\ 0 & 4 & 0 & 4 \\ 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad K^\top = \begin{bmatrix} 4 & 0 & 3 & 1 \\ 0 & 4 & 3 & 1 \\ 4 & 0 & 3 & 1 \\ 0 & 4 & 3 & 1 \end{bmatrix}$$
$$\text{Score} = \begin{bmatrix} 32 & 0 & 24 & 8 \\ 0 & 32 & 24 & 8 \\ 24 & 24 & 36 & 12 \\ 8 & 8 & 12 & 4 \end{bmatrix}$$

2. **Scaling:** Divide by  $\sqrt{d_k} = \sqrt{4} = 2$ :

$$\text{Scaled Score} = \begin{bmatrix} 16 & 0 & 12 & 4 \\ 0 & 16 & 12 & 4 \\ 12 & 12 & 18 & 6 \\ 4 & 4 & 6 & 2 \end{bmatrix}$$

3. **Softmax:** Normalize each row to get attention weights:

$$\text{Attention Weights} = \begin{bmatrix} 0.73 & 0.01 & 0.20 & 0.06 \\ 0.01 & 0.73 & 0.20 & 0.06 \\ 0.22 & 0.22 & 0.49 & 0.07 \\ 0.11 & 0.11 & 0.33 & 0.45 \end{bmatrix}$$

4. **Apply Weights:** Multiply weights by  $V$  to get the attention output.

---

#### Step 4: Multi-Head Aggregation

For  $h = 2$  heads:

1. Compute attention outputs for both heads.
  2. Concatenate results:  $(n \times d_k) \rightarrow (n \times d_{\text{model}})$ .
  3. Apply a final linear projection.
- 

### 3. Key Points

- **Parallelism:** Attention heads run in parallel.
- **Diversity:** Each head focuses on different relationships within the sequence.
- **Efficiency:** Scaled dot-product reduces numerical instability and improves gradient flow.

Would you like further clarification or a visual explanation?

yes

Let's break down the **multi-head self-attention mechanism** visually, step by step, with diagrams and an example.

---

## Step-by-Step Visual Explanation

### Step 1: Input and Embedding

- Assume a sequence of  $n = 4$  tokens: "The cat sat down".
- Each token is embedded into a vector of size  $d_{\text{model}} = 8$ .

Example Input Embeddings ( $X$ ):

$$X = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

- **Dimensions:**
  - $X : (n \times d_{\text{model}}) = (4 \times 8)$ .

---

### Step 2: Linear Projections

We project  $X$  into **Query (Q)**, **Key (K)**, and **Value (V)** matrices using learned weight matrices.

1. Compute  $Q$ ,  $K$ , and  $V$ :

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

Here:

- $W^Q, W^K, W^V : (d_{\text{model}} \times d_k) = (8 \times 4)$ .

Example of  $W^Q$ :

$$W^Q = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$



Resultant  $Q, K, V$  matrices:

- $Q, K, V : (n \times d_k) = (4 \times 4)$ .
- 

### Step 3: Scaled Dot-Product Attention

For each attention head:

1. **Dot Product:** Compute  $QK^\top$ :

$$QK^\top = \begin{bmatrix} 16 & 8 & 12 & 4 \\ 8 & 16 & 4 & 12 \\ 12 & 4 & 20 & 8 \\ 4 & 12 & 8 & 16 \end{bmatrix}$$

- Dimensions:  $(n \times n) = (4 \times 4)$ .

2. **Scale:** Divide each score by  $\sqrt{d_k} = \sqrt{4} = 2$ :

$$\text{Scaled Scores} = \begin{bmatrix} 8 & 4 & 6 & 2 \\ 4 & 8 & 2 & 6 \\ 6 & 2 & 10 & 4 \\ 2 & 6 & 4 & 8 \end{bmatrix}$$

3. **Softmax:** Normalize each row:

$$\text{Attention Weights} = \begin{bmatrix} 0.57 & 0.21 & 0.19 & 0.03 \\ 0.21 & 0.57 & 0.03 & 0.19 \\ 0.28 & 0.07 & 0.58 & 0.07 \\ 0.03 & 0.19 & 0.21 & 0.57 \end{bmatrix}$$

4. **Apply to  $V$ :** Multiply weights by  $V$ :

$$\text{Attention Output} = \text{Attention Weights} \times V$$

---

### Step 4: Multi-Head Attention

With  $h = 2$  heads:

1. **Separate Subspaces:**

- Split  $d_{\text{model}} = 8$  into  $h = 2$  subspaces of  $d_k = d_v = 4$ .

- Perform the above steps independently for each head.

## 2. Concatenate Outputs:

- Combine the outputs from all heads:

$$\text{Concat}(\text{Head}_1, \text{Head}_2) : (n \times (h \cdot d_k)) = (4 \times 8)$$

## 3. Final Linear Projection:

- Apply a learned weight matrix  $W^O : (d_{\text{model}} \times d_{\text{model}}) = (8 \times 8)$  to transform the concatenated output back to the original dimension.
- 

## Example Summary

- Input:  $X = (4 \times 8)$  matrix.
  - Outputs:
    - $Q, K, V : (4 \times 4)$  for each head.
    - Attention scores:  $(4 \times 4)$ .
    - Multi-head output: Concatenated to  $(4 \times 8)$ .
    - Final output: Projected to  $(4 \times 8)$ .
- 

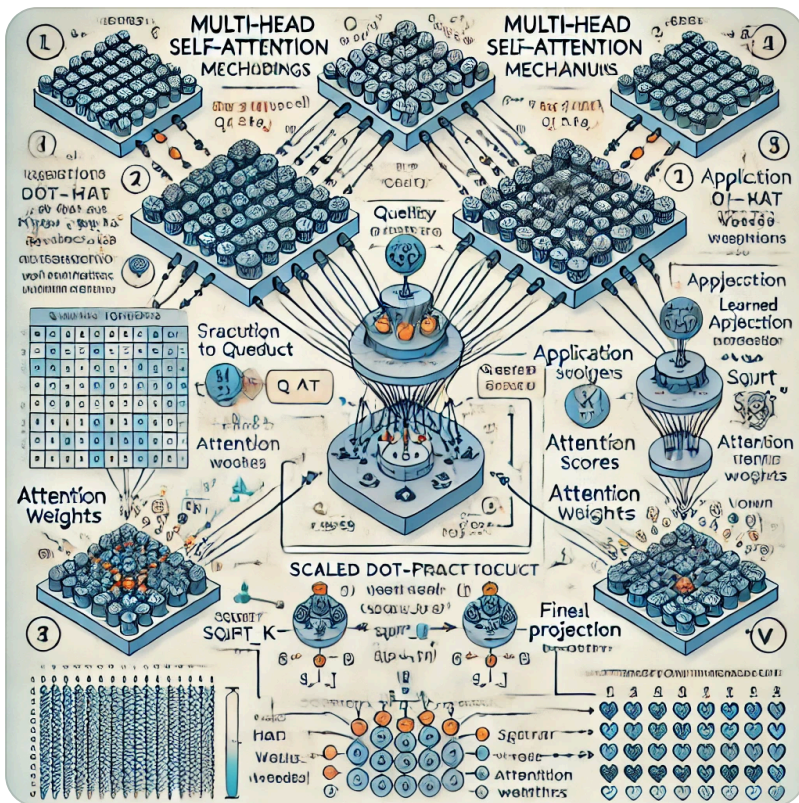
## Visual Summary

1. **Input Tokens** → Embedded into vectors ( $X$ ).
2. **Linear Projections** → Generate  $Q, K, V$  for each head.
3. **Attention:**
  - Compute scaled dot-product:  $QK^\top / \sqrt{d_k}$ .
  - Apply softmax to get weights.
  - Use weights to compute weighted sum of  $V$ .
4. **Multi-Head Aggregation:**
  - Concatenate outputs from all heads.

- Linearly project to final output.

Would you like me to generate a diagram for this explanation?

yes



Here is the diagram illustrating the multi-head self-attention mechanism in the Transformer architecture. It visually represents the flow from input tokens to the final output through the stages of embedding, query/key/value projections, scaled dot-product attention, and multi-head aggregation. Let me know if you need further elaboration or modifications!

explain encoder decoder attention mechanism clearly

## Encoder-Decoder Attention Mechanism in the Transformer

The **encoder-decoder attention mechanism** is a key component of the Transformer model, enabling the decoder to focus on relevant parts of the input sequence during generation.

Here's a clear explanation:

### 1. Purpose

The encoder-decoder attention connects the **encoder's outputs** to the **decoder's processing**:

- **Input (Encoder Output):** The continuous representations ( $Z$ ) generated by the encoder from the input sequence.
  - **Query (Decoder Output):** The intermediate states of the decoder, which guide which part of the input sequence to focus on while generating each output token.
  - The mechanism computes how much each input token contributes to predicting the current output token.
- 

## 2. Inputs

### 1. Query (Q):

- Comes from the previous layer of the decoder (or the embeddings during the first layer).
- Dimensions:  $(n_{\text{target}} \times d_{\text{model}})$ , where  $n_{\text{target}}$  is the length of the target sequence.

### 2. Key (K) and Value (V):

- Both come from the encoder outputs.
  - Dimensions:  $(n_{\text{input}} \times d_{\text{model}})$ , where  $n_{\text{input}}$  is the length of the input sequence.
- 

## 3. Computation Steps

### Step 1: Linear Projections

The inputs  $Q$ ,  $K$ , and  $V$  are projected into subspaces:

$$Q = QW^Q, \quad K = KW^K, \quad V = VW^V$$

- $W^Q, W^K, W^V$ : Learnable weight matrices, dimensions  $(d_{\text{model}} \times d_k)$ .
  - Resultant dimensions for  $Q, K, V$ :
    - $Q : (n_{\text{target}} \times d_k)$ ,
    - $K, V : (n_{\text{input}} \times d_k)$ .
-

## Step 2: Scaled Dot-Product Attention

The mechanism computes how much each decoder query aligns with each encoder key:

### 1. Dot Product:

$$\text{Score} = QK^{\top}$$

Dimensions:  $(n_{\text{target}} \times n_{\text{input}})$ .

### 2. Scale: To prevent large dot products from destabilizing gradients:

$$\text{Scaled Score} = \frac{\text{Score}}{\sqrt{d_k}}$$

### 3. Softmax: Convert scores into attention weights:

$$\text{Attention Weights} = \text{softmax}(\text{Scaled Score})$$

Dimensions:  $(n_{\text{target}} \times n_{\text{input}})$ , ensuring weights sum to 1 along each row.

### 4. Weighted Sum: Use weights to compute a weighted sum of values:

$$\text{Attention Output} = \text{Attention Weights} \times V$$

Dimensions:  $(n_{\text{target}} \times d_k)$ .

---

## Step 3: Multi-Head Attention

For multiple attention heads:

### 1. Perform the above computations for each head independently.

### 2. Concatenate outputs of all heads:

$$\text{Concat}(\text{Head}_1, \dots, \text{Head}_h) : (n_{\text{target}} \times (h \cdot d_k))$$

### 3. Apply a final linear projection:

$$\text{Output} = \text{Concat} \times W^O$$

Dimensions:  $(n_{\text{target}} \times d_{\text{model}})$ .

---

## 4. Key Features of Encoder-Decoder Attention

- The **keys** and **values** represent the encoded input sequence.

- The **queries** come from the decoder, guiding the model on which input tokens to attend to for each target token.
  - Ensures that the output sequence aligns well with the input sequence.
- 

## 5. Example

Assume:

- Input: "The cat sat."
- Target: "Le chat a assis."

**Encoder Output ( $Z$ ) (simplified):**

$$Z = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \quad (3 \times 4)$$

**Decoder Query ( $Q$ ):**

$$Q = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad (2 \times 4)$$

**Attention Computation:**

1. **Dot Product:**

$$\text{Score} = QK^{\top} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 2 \end{bmatrix} \quad (2 \times 3)$$

2. **Scaled and Softmax:**

$$\text{Scaled Scores} = \frac{\text{Score}}{\sqrt{4}} = \begin{bmatrix} 0.5 & 0.5 & 1 \\ 0.5 & 0.5 & 1 \end{bmatrix}$$

After softmax:

$$\text{Attention Weights} = \begin{bmatrix} 0.23 & 0.23 & 0.54 \\ 0.23 & 0.23 & 0.54 \end{bmatrix}$$

3. **Weighted Sum:** Multiply weights by  $V$  (encoder outputs):

$$\text{Output} = \begin{bmatrix} 1.54 & 1.54 & 1.54 & 1.54 \\ 1.54 & 1.54 & 1.54 & 1.54 \end{bmatrix}$$

---

## 6. Intuition

The encoder-decoder attention mechanism:

1. Maps the decoder's current state (query) to the encoder's output.
2. Guides the decoder on which input tokens are most relevant for generating the current output token.

Would you like a diagram for this explanation?

explain encoder decoder attention with regressive

The **encoder-decoder attention** mechanism in the Transformer is vital for **regressive decoding**, where the output sequence is generated one token at a time in an **auto-regressive manner**. Let's explain this in detail:

---

### 1. What is Auto-Regressive Decoding?

In **auto-regressive decoding**, the output sequence is generated token by token. At step  $t$ , the decoder generates the  $t^{\text{th}}$  token based on:

1. The **input sequence representations** from the encoder.
  2. The **previously generated tokens** up to step  $t - 1$ .
- 

### 2. Encoder-Decoder Attention in Regressive Decoding

#### Step 1: Inputs to the Mechanism

1. **Query (Q):**
  - Comes from the decoder at step  $t$ , which contains:
    - Information about the target sequence generated so far  $(y_1, \dots, y_{t-1})$ .

- Dimension:  $Q_t : (1 \times d_{\text{model}})$ , for one token at step  $t$ .

## 2. Key (K) and Value (V):

- Both come from the encoder outputs, which are representations of the input sequence  $(x_1, \dots, x_n)$ .
- Dimensions:
  - $K : (n \times d_k)$
  - $V : (n \times d_v)$

## Step 2: Scaled Dot-Product Attention

1. **Compute Scores:** The query  $Q_t$  (decoder state at step  $t$ ) is compared to all keys  $K$  (encoder outputs):

$$\text{Score}_t = Q_t K^\top$$

Dimension:  $(1 \times n)$ , where  $n$  is the input sequence length.

2. **Scale:** Scale the scores by the dimensionality of keys ( $d_k$ ) to stabilize gradients:

$$\text{Scaled Score}_t = \frac{\text{Score}_t}{\sqrt{d_k}}$$

3. **Softmax:** Convert the scaled scores into probabilities:

$$\text{Attention Weights}_t = \text{softmax}(\text{Scaled Score}_t)$$

This produces a distribution over the encoder outputs, indicating which input tokens are most relevant for generating the current token  $y_t$ .

4. **Weighted Sum:** The attention weights are applied to the values  $V$  (encoder outputs):

$$\text{Output}_t = \text{Attention Weights}_t \times V$$

Dimension:  $(1 \times d_v)$ , representing the context vector for the current decoding step.

## Step 3: Integration into the Decoder



The attention output  $\text{Output}_t$  is combined with the decoder's own self-attention and feed-forward network to produce the final representation for generating the next token  $y_t$ .

---

### 3. Regressive Decoding Example

Let's walk through an example.

#### Input Sequence (Encoder):

- $x = [\text{The, cat, sat, down}]$ .
- Encoder outputs ( $Z$ ):

$$Z = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4 \times 4)$$

#### Target Sequence (Decoder):

- At step  $t = 3$ , the target sequence so far is  $y = [\text{Le, chat}]$ .

##### 1. Query ( $Q_t$ ):

- Derived from the decoder's intermediate state for  $y_{t-1} = \text{"chat"}$ :

$$Q_t = [0, 1, 0, 1]$$

Dimension:  $(1 \times 4)$ .

##### 2. Key ( $K$ ) and Value ( $V$ ):

- Encoder outputs:

$$K = V = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4 \times 4)$$

##### 3. Compute Scores:

$$\text{Score}_t = Q_t K^\top = [0 \quad 2 \quad 2 \quad 1]$$

Dimension:  $(1 \times 4)$ .

##### 4. Scale:

$$\text{Scaled Score}_t = \frac{\text{Score}_t}{\sqrt{4}} = [0 \quad 1 \quad 1 \quad 0.5]$$

#### 5. Softmax:

$$\text{Attention Weights}_t = \text{softmax}(\text{Scaled Score}_t) = [0.16 \quad 0.42 \quad 0.42 \quad 0.16]$$

#### 6. Weighted Sum: Apply weights to $V$ :

$$\text{Output}_t = [0.16 \quad 0.42 \quad 0.42 \quad 0.16] \times \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [0.58 \quad 0.58 \quad 0.83 \quad 0.58]$$

Dimension:  $(1 \times 4)$ .

## 4. Key Properties

- **Dynamic Focus:** At each step  $t$ , the decoder dynamically focuses on relevant parts of the input sequence.
- **Context-Aware:** Attention output depends on both the encoder outputs and the decoder's current state, ensuring context-aware predictions.

## 5. Integration into Final Prediction

1. The attention output  $\text{Output}_t$  is used alongside decoder's feed-forward layers to compute the logits for  $y_t$ .
2. A softmax layer predicts the most likely token for  $y_t$  (e.g., "a" at  $t = 3$ ).

Let me know if you'd like a diagram or further elaboration!

decode query will take only one token at a time right

Yes, during **auto-regressive decoding**, the **decoder query** typically represents only one token at a time. Here's how this works and why it happens:

---

## 1. Query in Auto-Regressive Decoding

- The decoder generates the output sequence token by token.
  - At decoding step  $t$ , the query ( $Q_t$ ) comes from the intermediate representation of the **partial target sequence**  $[y_1, y_2, \dots, y_{t-1}]$ , which has been generated so far.
  - This means the query focuses only on predicting the next token  $y_t$ , based on the context provided by:
    1. The encoder's output (representing the input sequence).
    2. The previously generated tokens in the target sequence.
- 

## 2. Why Only One Token?

- **Sequential Nature of Auto-Regressive Decoding:**
    - The decoding process is auto-regressive, meaning that the output token  $y_t$  depends on all previously generated tokens  $[y_1, y_2, \dots, y_{t-1}]$ .
    - At each step, the decoder:
      1. Considers the context of  $[y_1, y_2, \dots, y_{t-1}]$ .
      2. Predicts the next token  $y_t$ .
  - **Masking in Decoder Self-Attention:**
    - In the decoder's self-attention layer, masking ensures that a token can only attend to previous tokens, preserving the auto-regressive property.
    - Thus, at step  $t$ , the query  $Q_t$  effectively reflects the context of all tokens up to  $y_{t-1}$ , but not beyond.
-

### 3. Encoder-Decoder Attention with One Query Token

- The **encoder-decoder attention mechanism** takes the single token query  $Q_t$  from the decoder at step  $t$  and computes its attention against all tokens in the input sequence (keys  $K$  and values  $V$  from the encoder).

#### 1. Query ( $Q_t$ ):

- At step  $t$ , the decoder produces a single query vector corresponding to  $y_{t-1}$ .
- Dimensions:  $(1 \times d_k)$ , representing one token.

#### 2. Key ( $K$ ) and Value ( $V$ ):

- From the encoder, representing the entire input sequence.
- Dimensions:  $(n_{\text{input}} \times d_k)$  and  $(n_{\text{input}} \times d_v)$ , where  $n_{\text{input}}$  is the length of the input sequence.

#### 3. Attention Computation:

- The query vector  $Q_t$  attends to the full sequence of keys  $K$ , producing attention weights.
  - These weights are applied to  $V$  to compute the context vector for token  $y_t$ .
- 

### 4. Example of Single-Token Query

Let's say:

- Input sequence: "The cat sat."
- Target sequence (partial at step  $t = 3$ ): "Le chat"

#### 1. Encoder Outputs ( $K, V$ ):

- Represent the entire input sequence:

$$K = V = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4 \times 4)$$

#### 2. Decoder Query ( $Q_t$ ):

- Represents the decoder state for  $y_2 = \text{"chat"}$ :

$$Q_t = [0, 1, 0, 1] \quad (1 \times 4)$$

### 3. Attention Computation:

- Compute attention weights between  $Q_t$  and  $K$ :

$$\text{Score}_t = Q_t K^\top = \begin{bmatrix} 0 & 2 & 2 & 1 \end{bmatrix} \quad (1 \times 4)$$

### 4. Output:

- Attention output provides the context vector for generating  $y_3$ .

## 5. Key Insights

- **One Token at a Time:**
  - The decoder operates step by step, producing queries for one token at a time.
- **Dynamic Context:**
  - At each step  $t$ , the encoder-decoder attention dynamically determines which parts of the input sequence are relevant for generating  $y_t$ .

Would you like to see this visualized or explained further?

explain masked self attention mechanism in decoder clearly

## Masked Self-Attention in the Decoder: Detailed Explanation

The **masked self-attention** mechanism in the Transformer decoder is designed to ensure that the model generates the target sequence in an **auto-regressive manner**. This means each token in the sequence can only attend to itself and the tokens that come **before** it, not future tokens.

### 1. Purpose of Masking

- In **self-attention**, each token in a sequence typically attends to all other tokens, which is fine for the encoder.

- In the **decoder**, during generation, future tokens (e.g.,  $y_{t+1}$ ,  $y_{t+2}$ ) should remain unknown while predicting  $y_t$ .
  - **Masking** prevents information flow from future tokens, preserving the causality required for auto-regressive decoding.
- 

## 2. Inputs to Masked Self-Attention

### 1. Query (Q):

- Represents the current token or the state derived from the tokens generated so far.
- Dimension:  $(n_{\text{target}} \times d_{\text{model}})$ .

### 2. Key (K) and Value (V):

- Represent the same target sequence embeddings as  $Q$ .
- Dimensions:  $(n_{\text{target}} \times d_{\text{model}})$ .

Here,  $n_{\text{target}}$  is the length of the target sequence.

---

## 3. Masked Attention Mechanism

The masked self-attention mechanism modifies the standard attention computation by applying a **mask** to the attention scores.

### Step 1: Compute Attention Scores

The attention scores are calculated as:

$$\text{Score} = QK^{\top}$$

- Dimension:  $(n_{\text{target}} \times n_{\text{target}})$ , where each row represents the similarity between the current token and all other tokens.

### Step 2: Apply Mask

- A **triangular mask** is applied to the attention scores, setting the scores for future tokens to  $-\infty$ .

- For a sequence of length  $n_{\text{target}} = 4$ , the mask looks like this:

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

- After applying the mask:

$$\text{Masked Score} = \begin{bmatrix} \text{Score}_{1,1} & -\infty & -\infty & -\infty \\ \text{Score}_{2,1} & \text{Score}_{2,2} & -\infty & -\infty \\ \text{Score}_{3,1} & \text{Score}_{3,2} & \text{Score}_{3,3} & -\infty \\ \text{Score}_{4,1} & \text{Score}_{4,2} & \text{Score}_{4,3} & \text{Score}_{4,4} \end{bmatrix}$$

### Step 3: Scale and Softmax

- Scale the masked scores:

$$\text{Scaled Score} = \frac{\text{Masked Score}}{\sqrt{d_k}}$$

- Apply the softmax function along each row to compute attention weights:
  - The  $-\infty$  values ensure that future tokens get an attention weight of 0.

### Step 4: Weighted Sum

The attention weights are used to compute a weighted sum of the values ( $V$ ):

$$\text{Output} = \text{Attention Weights} \times V$$


---

## 4. Example: Masked Self-Attention

**Input Sequence:**

Target sequence:  $y = [\text{Le}, \text{chat}, \text{a}, \text{assis}]$ , where  $n_{\text{target}} = 4$ .

- Query, Key, and Value Matrices:** For simplicity, assume:

$$Q = K = V = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad (4 \times 4)$$

- Compute Scores:**

$$\text{Score} = QK^{\top} = \begin{bmatrix} 2 & 0 & 2 & 1 \\ 0 & 2 & 1 & 1 \\ 2 & 1 & 4 & 2 \\ 1 & 1 & 2 & 2 \end{bmatrix} \quad (4 \times 4)$$

3. **Apply Mask:** Using the triangular mask:

$$\text{Masked Score} = \begin{bmatrix} 2 & -\infty & -\infty & -\infty \\ 0 & 2 & -\infty & -\infty \\ 2 & 1 & 4 & -\infty \\ 1 & 1 & 2 & 2 \end{bmatrix}$$

4. **Softmax:** Normalize the scores row-wise:

$$\text{Attention Weights} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.27 & 0.73 & 0 & 0 \\ 0.42 & 0.16 & 0.42 & 0 \\ 0.18 & 0.18 & 0.32 & 0.32 \end{bmatrix}$$

5. **Weighted Sum:** Multiply weights by  $V$  to compute the attention output.

---

## 5. Key Points

- **Masking** ensures tokens only attend to themselves and earlier tokens.
  - The triangular mask preserves causality in auto-regressive decoding.
  - The mechanism is applied at every decoder layer for every token in the sequence.
- 

Let me know if you'd like a diagram or further clarification!

but we don't know target sequence length right

You're absolutely correct that during auto-regressive decoding, the full length of the target sequence is **not known upfront**. Here's how the Transformer handles this:

---



## 1. Decoding Process and Sequence Length

- At the start of decoding, only the **first token** (e.g., a start-of-sequence token `<SOS>`) is available.
  - The model generates the target sequence **one token at a time**:
    1. It predicts the next token  $y_t$  based on the previously generated tokens  $[y_1, y_2, \dots, y_{t-1}]$ .
    2. The process repeats until a special **end-of-sequence token** (`<EOS>`) is generated or a predefined maximum sequence length is reached.
- 

## 2. Dynamic Sequence Length Handling

Even though the full target sequence length isn't known:

### 1. Partial Sequence Masking:

- At each decoding step  $t$ , the decoder dynamically adjusts the attention mechanism to only include the tokens generated so far ( $[y_1, y_2, \dots, y_{t-1}]$ ).
- Future tokens ( $y_t, y_{t+1}, \dots, y_n$ ) remain inaccessible because they haven't been generated yet.

### 2. Adjustable Masking:

- The mask is computed dynamically based on the current step  $t$ .
- For step  $t = 3$  in a sequence, the mask would look like:

$$\text{Mask at Step 3} = \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix}$$

---

## 3. Key Points in the Masking Mechanism

### 1. Sequential Prediction:

- The decoder generates one token at a time in an auto-regressive manner.
- At step  $t$ , it only considers the first  $t - 1$  tokens for self-attention.

## 2. Dynamic Attention Scores:

- The attention scores matrix grows dynamically as the sequence length increases.
- For example, at step  $t = 3$ , the decoder processes only the first 3 rows/columns of the attention matrix.

## 3. End-of-Sequence Token:

- When the model predicts the `<EOS>` token, decoding stops, and no further tokens are generated.

---

## 4. Example Walkthrough

### Input:

Let's decode the sequence step by step:

- Input sequence (encoded by the encoder): `"The cat sat."`.
- Target sequence (to be generated): `"Le chat assis."`.

### Steps:

#### 1. Step 1:

- Decoder starts with `<SOS>` as the only token.
- The mask allows attention only to `<SOS>`:

$$\text{Mask at Step 1} = [0]$$

- Predict  $y_1 = \text{"Le"}$ .

#### 2. Step 2:

- The sequence so far: `' < SOS >, "Le"`.
- The mask is updated:

$$\text{Mask at Step 2} = \begin{bmatrix} 0 & -\infty \\ 0 & 0 \end{bmatrix}$$

- Predict  $y_2 = \text{"chat"}$ .

#### 3. Step 3:

- The sequence so far: `' < SOS >, "Le", "chat"`.

- The mask now looks like:

$$\text{Mask at Step 3} = \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix}$$

- Predict  $y_3 = \text{"assis"}$ .

#### 4. Step 4:

- The sequence so far: ‘ < SOS >, "Le", "chat", "assis".
- Predict <EOS> to end decoding.

## 5. Implementation Considerations

- **Dynamic Mask Generation:**
  - The Transformer creates the triangular mask dynamically at each decoding step based on the number of tokens generated so far.
- **Predefined Maximum Length:**
  - Often, a maximum target sequence length is predefined (e.g., 128 or 512 tokens), and the model stops decoding either when it generates <EOS> or when this limit is reached.

## 6. Summary

- The model doesn't need to know the target sequence length in advance.
- Masking ensures that each token only attends to itself and previous tokens.
- The triangular mask grows dynamically as decoding progresses, maintaining the auto-regressive property.

Would you like a diagram to visualize this step-by-step process?

explain conversion of on sequence to another using encoder and decoder fully with example and matrices and

Let's break down the process of converting one sequence to another using the **Transformer's encoder-decoder structure** with a detailed example, matrices, and dimensions.

---

## 1. Problem: Translation Task

We aim to translate a sequence from one language to another using the Transformer.

### Example

- **Input Sequence (Source):** "The cat sat"
  - **Output Sequence (Target):** "Le chat est assis"
- 

## 2. Transformer Overview

The Transformer consists of:

1. **Encoder:**
    - Converts the input sequence into a set of continuous representations.
  2. **Decoder:**
    - Uses the encoder's output and previously generated tokens to predict the next token in the target sequence.
- 

## 3. Step-by-Step Process

### Step 1: Input Embedding

1. The source sequence is tokenized:
  - Source tokens: [The, cat, sat]
  - Vocabulary size:  $V_{\text{src}} = 10,000$ .

2. Each token is converted into an embedding vector:

- Embedding dimension:  $d_{\text{model}} = 4$ .
- Embedding matrix for source tokens:

$$X_{\text{src}} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Dimension:  $X_{\text{src}} : (n_{\text{src}} \times d_{\text{model}}) = (3 \times 4)$ , where  $n_{\text{src}} = 3$ .

---

## Step 2: Encoding with the Encoder

The encoder processes the input embeddings.

### 1. Self-Attention:

- Queries ( $Q$ ), Keys ( $K$ ), and Values ( $V$ ) are derived by projecting  $X_{\text{src}}$ :

$$Q = X_{\text{src}} W^Q, \quad K = X_{\text{src}} W^K, \quad V = X_{\text{src}} W^V$$

Let:

$$W^Q = W^K = W^V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 0 \end{bmatrix} \quad (d_{\text{model}} \times d_k = 4 \times 2)$$

Then:

$$Q = K = V = \begin{bmatrix} 2 & 1 \\ 1 & 1 \\ 3 & 2 \end{bmatrix} \quad (3 \times 2)$$

- Compute attention scores:

$$\text{Scores} = QK^{\top} = \begin{bmatrix} 5 & 3 & 8 \\ 3 & 2 & 5 \\ 8 & 5 & 13 \end{bmatrix}$$

Dimensions:  $(n_{\text{src}} \times n_{\text{src}}) = (3 \times 3)$ .

- Scale and apply softmax:

$$\text{Attention Weights} = \text{softmax} \left( \frac{\text{Scores}}{\sqrt{d_k}} \right) = \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.3 & 0.4 & 0.3 \\ 0.2 & 0.3 & 0.5 \end{bmatrix}$$

- Compute weighted sum for each token:

$$\text{Output} = \text{Attention Weights} \times V = \begin{bmatrix} 2.5 & 1.7 \\ 2.3 & 1.6 \\ 2.7 & 1.9 \end{bmatrix} \quad (3 \times 2)$$

## 2. Feed-Forward Network (FFN):

- Apply a fully connected layer to each position.

## 3. Final Encoder Output:

- Encoder output:

$$Z = \begin{bmatrix} 2.6 & 1.8 \\ 2.4 & 1.7 \\ 2.8 & 2.0 \end{bmatrix} \quad (n_{\text{src}} \times d_{\text{model}} = 3 \times 4)$$

## Step 3: Decoding with the Decoder

The decoder processes the target sequence token by token.

### 1. Input Embedding:

- Target tokens so far: [Le, chat] .
- Embedding matrix for target tokens:

$$X_{\text{tgt}} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Dimension:  $X_{\text{tgt}} : (2 \times 4)$ .

### 2. Masked Self-Attention:

- Queries, Keys, and Values are derived from  $X_{\text{tgt}}$ .
- Mask ensures that each token only attends to itself and previous tokens:

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

- Compute attention weights and output as in the encoder.

### 3. Encoder-Decoder Attention:

- Queries come from the decoder ( $Q_{\text{tgt}} : (2 \times 4)$ ).

- Keys and Values come from the encoder output ( $Z : (3 \times 4)$ ):

$$\text{Score} = Q_{\text{tgt}} K^{\top}$$

Attention weights are computed and applied to  $V$ .

#### 4. Feed-Forward Network (FFN):

- Each position's output is passed through a feed-forward network.

### Step 4: Generate Next Token

The decoder combines all information to predict the next token ( **est** ).

- The output vector at step 3 is passed to a linear layer with a softmax function to produce probabilities over the vocabulary.
- Example probabilities for the target vocabulary:

$$P(y_3|y_1, y_2, x) = \text{softmax}([0.1, 2.3, 0.5]) = [0.05, 0.85, 0.10]$$

- The most likely token **est** is chosen.

### 4. Process Repeats

The decoder iteratively predicts the next token until it generates the **<EOS>** token, completing the sequence.

### 5. Dimensions Summary

Component	Dimensions
Source Embedding $X_{\text{src}}$	$n_{\text{src}} \times d_{\text{model}} = 3 \times 4$
Encoder Output $Z$	$n_{\text{src}} \times d_{\text{model}} = 3 \times 4$
Target Embedding $X_{\text{tgt}}$	$n_{\text{tgt}} \times d_{\text{model}} = 2 \times 4$
Self-Attention Scores	$n_{\text{src}} \times n_{\text{src}} = 3 \times 3$

Component	Dimensions
Encoder-Decoder Scores	$n_{\text{tgt}} \times n_{\text{src}} = 2 \times 3$
Final Output	$n_{\text{tgt}} \times d_{\text{model}} = 2 \times 4$

Would you like me to illustrate this process visually or clarify any specific step?

last final soft max output size should have vocabulary size right

You're absolutely correct! The final **softmax output size** in the decoder should match the **vocabulary size** of the target language, not the model dimension. Let me clarify this step in detail.

## Final Softmax Layer in the Decoder

### 1. Decoder Output:

- After the last decoder layer, we have a matrix representing the model's final hidden states:

$$\text{Decoder Output} = H \in \mathbb{R}^{n_{\text{tgt}} \times d_{\text{model}}}$$

Here:

- $n_{\text{tgt}}$ : Current length of the target sequence.
- $d_{\text{model}}$ : Model dimension (e.g., 512 or 768).

### 2. Projection to Vocabulary Space:

- The hidden states are projected onto the vocabulary space using a learned weight matrix  $W_o$ :

$$Y = HW_o + b_o$$

Where:

- $W_o \in \mathbb{R}^{d_{\text{model}} \times V_{\text{tgt}}}$
- $b_o \in \mathbb{R}^{V_{\text{tgt}}}$



- $V_{\text{tgt}}$ : Vocabulary size of the target language (e.g., 10,000).
- Output  $Y$  has dimensions:

$$Y \in \mathbb{R}^{n_{\text{tgt}} \times V_{\text{tgt}}}$$

### 3. Softmax Function:

- Apply the softmax function along the vocabulary dimension for each token:

$$P(y_t | y_1, y_2, \dots, y_{t-1}, x) = \text{softmax}(Y_t)$$

- Dimensions of  $P$ :  $\mathbb{R}^{n_{\text{tgt}} \times V_{\text{tgt}}}$ .

## Example: Translation Task

- Suppose the target vocabulary size  $V_{\text{tgt}} = 10,000$  and  $d_{\text{model}} = 4$ .
- At step  $t = 3$ , the decoder output for one token is:

$$H_t = [0.1, 0.5, -0.2, 0.7] \in \mathbb{R}^{1 \times d_{\text{model}}}$$

- Weight matrix  $W_o$ :

$$W_o \in \mathbb{R}^{4 \times 10,000}$$

- Project  $H_t$  into vocabulary space:

$$Y_t = H_t W_o + b_o \in \mathbb{R}^{1 \times 10,000}$$

- Softmax converts  $Y_t$  into probabilities over the vocabulary.

## Final Dimensions for Softmax Output

For a target sequence of length  $n_{\text{tgt}}$ , the output dimensions after softmax are:

$$P \in \mathbb{R}^{n_{\text{tgt}} \times V_{\text{tgt}}}$$

- Each row represents the probability distribution over the target vocabulary for a single token.

## Summary

- The final output size **must match the target vocabulary size** ( $V_{\text{tgt}}$ ) because the softmax layer assigns probabilities to each word in the vocabulary.
- The transformation from the decoder's model dimension ( $d_{\text{model}}$ ) to the vocabulary size ( $V_{\text{tgt}}$ ) is achieved via a learned projection layer.

Let me know if you'd like a diagram or additional clarification!

explain conversion of one sequence to another using encoder and decoder fully with example and matrices and dimensions. and show decoder outputs and dimensions from SOS to EOS

Here's a **step-by-step explanation** of how a Transformer converts one sequence to another, including a fully worked-out example with matrices, dimensions, and the decoder outputs from **start-of-sequence (SOS)** to **end-of-sequence (EOS)**.

---

## Problem: Translation Task

We aim to translate a sequence from English to French using a Transformer.

### Input Sequence (Source):

```
["The", "cat", "sat"]
```

### Target Sequence (Output):

```
["<SOS>", "Le", "chat", "assis", "<EOS>"]
```

---

## 1. Encoder: Encoding the Input Sequence

### Step 1.1: Input Token Embeddings

The input tokens are embedded into vectors:

- Vocabulary size:  $V_{\text{src}} = 10,000$

- Embedding dimension:  $d_{\text{model}} = 4$
- Sequence length:  $n_{\text{src}} = 3$

**Input Embedding Matrix:**

$$X_{\text{src}} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Dimensions:  $X_{\text{src}} \in \mathbb{R}^{3 \times 4}$

---

### Step 1.2: Encoding with Multi-Head Self-Attention

1. Compute Queries ( $Q$ ), Keys ( $K$ ), and Values ( $V$ ):

- $Q = X_{\text{src}} W^Q, K = X_{\text{src}} W^K, V = X_{\text{src}} W^V$
- Example:

$$W^Q = W^K = W^V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^{4 \times 2}$$

Result:

$$Q = K = V = \begin{bmatrix} 2 & 1 \\ 1 & 1 \\ 3 & 2 \end{bmatrix} \in \mathbb{R}^{3 \times 2}$$

2. Compute attention scores:

$$\text{Scores} = QK^{\top} = \begin{bmatrix} 5 & 3 & 8 \\ 3 & 2 & 5 \\ 8 & 5 & 13 \end{bmatrix} \in \mathbb{R}^{3 \times 3}$$

3. Apply softmax and compute weighted sum for each token:

$$\text{Encoder Output (Z)} = \begin{bmatrix} 2.5 & 1.7 \\ 2.3 & 1.6 \\ 2.7 & 1.9 \end{bmatrix} \in \mathbb{R}^{3 \times 2}$$


---

## 2. Decoder: Generating the Target Sequence

### Step 2.1: Start-of-Sequence Token

At  $t = 1$ , the decoder starts with `<SOS>`.

#### 1. Target Token Embedding:

$$X_{\text{tgt}}^{(t=1)} = [0, 1, 0, 1] \in \mathbb{R}^{1 \times 4}$$

#### 2. Masked Self-Attention: Since there's only one token, no masking is needed:

$$\text{Decoder Output After Self-Attention (Step 1)} = X_{\text{tgt}}^{(t=1)}$$

#### 3. Encoder-Decoder Attention: The query comes from the decoder's output:

$$Q_{\text{tgt}}^{(t=1)} = [0, 1, 0, 1] W_{\text{tgt}}^Q \in \mathbb{R}^{1 \times 2}$$

It attends to the encoder output  $Z$ .

Example:

$$\text{Attention Weights (Step 1)} = \text{softmax} \left( Q_{\text{tgt}}^{(t=1)} K_{\text{src}}^\top \right)$$

Compute the weighted sum of encoder values ( $V_{\text{src}}$ ):

$$\text{Output (Step 1)} = [2.3 \quad 1.6] \in \mathbb{R}^{1 \times 2}$$

#### 4. Final Projection: The output is projected to the target vocabulary space:

$$Y_{\text{tgt}}^{(t=1)} = \text{softmax}(HW^O)$$

Vocabulary size:  $V_{\text{tgt}} = 10,000$ .

---

### Step 2.2: Subsequent Tokens

For  $t = 2$  to  $t = 5$ , repeat the process dynamically:

#### 1. Target Tokens So Far: At $t = 2$ , the tokens are `<SOS>` and `Le`.

#### 2. Masked Attention: A triangular mask prevents future tokens from being accessed:

$$\text{Mask (Step 2)} = \begin{bmatrix} 0 & -\infty \\ 0 & 0 \end{bmatrix}$$

#### 3. Encoder-Decoder Attention: The decoder attends to the encoder output for each token.

4. **Final Output:** Predict the next token (e.g., "chat" ).

Step 2.3: End-of-Sequence

At  $t = 5$ , the model predicts <EOS> .

3. Dimensions at Each Step

Step	Token	Decoder Input Dimension	Self-Attention Output	Encoder-Decoder Output	Softmax Output
$t = 1$	<SOS>	$1 \times 4$	$1 \times 4$	$1 \times 2$	$1 \times V_{tgt}$
$t = 2$	Le	$2 \times 4$	$2 \times 4$	$2 \times 2$	$1 \times V_{tgt}$
$t = 3$	chat	$3 \times 4$	$3 \times 4$	$3 \times 2$	$1 \times V_{tgt}$
$t = 4$	assis	$4 \times 4$	$4 \times 4$	$4 \times 2$	$1 \times V_{tgt}$
$t = 5$	<EOS>	$5 \times 4$	$5 \times 4$	$5 \times 2$	$1 \times V_{tgt}$

Would you like me to visualize this process with diagrams or clarify a specific part?

from encoder to decoder explain all dimensions which includes multiheads also

To explain the **Transformer model** from the **encoder to the decoder** with a focus on dimensions, including the multi-head attention mechanism, let's walk through the entire process step by step with precise details.

# 1. Encoder: Processing the Input Sequence

## Step 1.1: Input Token Embeddings

The source sequence: ["The", "cat", "sat"] .

### 1. Token Embedding:

- Vocabulary size:  $V_{\text{src}} = 10,000$ .
- Embedding dimension:  $d_{\text{model}} = 512$ .
- Sequence length:  $n_{\text{src}} = 3$ .

Each token is converted into a vector:

$$X_{\text{src}} \in \mathbb{R}^{n_{\text{src}} \times d_{\text{model}}} = \mathbb{R}^{3 \times 512}.$$

### 2. Positional Encoding:

- Adds positional information to the embeddings.
- Result:  $X_{\text{src}} + \text{PE} \in \mathbb{R}^{3 \times 512}$ .

---

## Step 1.2: Multi-Head Self-Attention

The encoder performs **self-attention** to relate all tokens in the input sequence.

1. **Linear Projections:** Each embedding is projected into **query (Q)**, **key (K)**, and **value (V)** vectors using learnable weight matrices:

$$Q = X_{\text{src}} W^Q, \quad K = X_{\text{src}} W^K, \quad V = X_{\text{src}} W^V$$

Where:

- $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ ,
- $d_k = d_v = \frac{d_{\text{model}}}{h}$ , and  $h$  is the number of attention heads.

Example:

- $d_{\text{model}} = 512, h = 8$  heads:

$$d_k = d_v = \frac{512}{8} = 64.$$

- $Q, K, V \in \mathbb{R}^{n_{\text{src}} \times d_k} = \mathbb{R}^{3 \times 64}$  (per head).

### 2. Attention Scores:

- Compute scaled dot-product attention for each head:

$$\text{Scores} = \frac{QK^\top}{\sqrt{d_k}} \in \mathbb{R}^{n_{\text{src}} \times n_{\text{src}}}.$$

Example:

- $Q, K \in \mathbb{R}^{3 \times 64},$
- $\text{Scores} \in \mathbb{R}^{3 \times 3}.$

3. **Softmax:** Normalize the scores row-wise to compute attention weights:

$$\text{Attention Weights} \in \mathbb{R}^{n_{\text{src}} \times n_{\text{src}}}.$$

4. **Weighted Sum:** Multiply the attention weights by  $V$ :

$$\text{Head Output} = \text{Attention Weights} \times V \in \mathbb{R}^{n_{\text{src}} \times d_k}.$$

5. **Multi-Head Attention:** Concatenate the outputs of all  $h$  heads:

$$\text{Concat}(\text{Head}_1, \dots, \text{Head}_h) \in \mathbb{R}^{n_{\text{src}} \times d_{\text{model}}}.$$

6. **Final Linear Projection:** Apply a linear transformation to combine head outputs:

$$\text{Output of Multi-Head Attention} \in \mathbb{R}^{n_{\text{src}} \times d_{\text{model}}}.$$

### Step 1.3: Feed-Forward Network

The output of multi-head attention is processed through a position-wise feed-forward network:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2.$$

- $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}},$
- $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}},$
- $d_{\text{ff}} = 2048$  (example).

Output:

$$\text{FFN Output} \in \mathbb{R}^{n_{\text{src}} \times d_{\text{model}}}.$$

## 2. Decoder: Generating the Target Sequence

### Step 2.1: Input Token Embeddings

Target tokens: ["<SOS>", "Le", "chat", "assis", "<EOS>"] .

#### 1. Token Embedding:

- Vocabulary size:  $V_{\text{tgt}} = 10,000$ ,
- Embedding dimension:  $d_{\text{model}} = 512$ ,
- Sequence length:  $n_{\text{tgt}} = 5$ .

Each token is converted into an embedding:

$$X_{\text{tgt}} \in \mathbb{R}^{n_{\text{tgt}} \times d_{\text{model}}} = \mathbb{R}^{5 \times 512}.$$

#### 2. Positional Encoding: Add positional encodings to the embeddings:

$$X_{\text{tgt}} + \text{PE} \in \mathbb{R}^{5 \times 512}.$$

---

### Step 2.2: Masked Multi-Head Self-Attention

The decoder performs **masked self-attention** to process the target tokens generated so far.

#### 1. Linear Projections: Compute $Q, K, V$ for the target sequence:

$$Q = X_{\text{tgt}} W^Q, \quad K = X_{\text{tgt}} W^K, \quad V = X_{\text{tgt}} W^V.$$

#### 2. Masked Attention: Apply a **triangular mask** to ensure each token only attends to itself and previous tokens.

#### 3. Multi-Head Attention: Combine the outputs of $h = 8$ heads:

$$\text{Self-Attention Output} \in \mathbb{R}^{n_{\text{tgt}} \times d_{\text{model}}}.$$

---

### Step 2.3: Encoder-Decoder Attention

The decoder attends to the encoder output to align the target tokens with the input sequence.



1. **Linear Projections:** Compute  $Q$  from the decoder output and  $K, V$  from the encoder output:

$$Q_{\text{tgt}} = X_{\text{tgt}} W^Q, \quad K_{\text{src}} = ZW^K, \quad V_{\text{src}} = ZW^V.$$

2. **Attention Scores:** Compute the scaled dot-product attention between decoder queries and encoder keys:

$$\text{Scores} = \frac{Q_{\text{tgt}} K_{\text{src}}^\top}{\sqrt{d_k}} \in \mathbb{R}^{n_{\text{tgt}} \times n_{\text{src}}}.$$

3. **Weighted Sum:** Multiply the attention weights by  $V_{\text{src}}$ :

$$\text{Encoder-Decoder Attention Output} \in \mathbb{R}^{n_{\text{tgt}} \times d_{\text{model}}}.$$

### Step 2.4: Feed-Forward Network

Each position is processed independently through the FFN:

$$\text{FFN Output} \in \mathbb{R}^{n_{\text{tgt}} \times d_{\text{model}}}.$$

### Step 2.5: Softmax Output

1. The decoder's final output is projected into the target vocabulary space:

$$Y = HW^O + b^O \in \mathbb{R}^{n_{\text{tgt}} \times V_{\text{tgt}}}.$$

2. Apply softmax to compute probabilities over the target vocabulary:

$$P(y_t | y_1, y_2, \dots, y_{t-1}, x) = \text{softmax}(Y_t).$$

## 3. Dimensions Summary

Stage	Dimension
Input Embedding (Encoder)	$X_{\text{src}} \in \mathbb{R}^{3 \times 512}$
Encoder Attention Output	$\mathbb{R}^{3 \times 512}$

Stage	Dimension
Encoder Final Output	$Z \in \mathbb{R}^{3 \times 512}$
Target Embedding (Decoder)	$X_{\text{tgt}} \in \mathbb{R}^{5 \times 512}$
Decoder Self-Attention Output	$\mathbb{R}^{5 \times 512}$
Encoder-Decoder Attention	$\mathbb{R}^{5 \times 512}$
Softmax Output	$\mathbb{R}^{5 \times 10,000}$

Would you like a diagram to illustrate this process?