ChatGPT

In "encoder-decoder attention" layers, the queries come from the previous decoder layer,
and the memory keys and values come from the output of the encoder. This allows every
position in the decoder to attend over all positions in the input sequence. This mimics the
typical encoder-decoder attention mechanisms in sequence-to-sequence models such as
[38, 2, 9].
• The encoder contains self-attention layers. In a self-attention layer all of the keys, values
and queries come from the same place, in this case, the output of the previous layer in the
encoder. Each position in the encoder can attend to all positions in the previous layer of the
encoder.
• Similarly, self-attention layers in the decoder allow each position in the decoder to attend to
all positions in the decoder up to and including that position. We need to prevent leftward
information flow in the decoder to preserve the auto-regressive property. We implement this
inside of scaled dot-product attention by masking out (setting to −∞) all values in the input
of the softmax which correspond to illegal connections. See Figure 2.

This text provides an explanation of attention mechanisms within encoder-decoder architectures, particularly in the context of sequence-to-sequence models. Here's a breakdown of the key points:

1. Encoder-Decoder Attention

- **Queries**: These originate from the output of the **previous decoder layer**.
- **Keys and Values**: These are derived from the **output of the encoder**.
- **Purpose**: This design enables every position in the decoder to focus (or "attend") on all positions of the input sequence, ensuring rich contextual understanding.

- **Use Case**: Mimics traditional encoder-decoder mechanisms found in models like sequence-to-sequence frameworks.

2. Encoder Self-Attention

- **Inputs**: Keys, values, and queries all come from the **same source**—the encoder's previous layer output.
- **Behavior**: Every position in the encoder can attend to all other positions from the previous layer.
- **Outcome**: Builds contextual understanding across the input sequence within the encoder.

3. Decoder Self-Attention

- **Inputs**: Like in the encoder, keys, values, and queries are all derived from the **decoder's previous layer**.
- **Restriction**: Attention is limited to positions **up to and including the current position** to maintain the auto-regressive nature (important for generating sequences step by step).
- **Implementation**: Achieved by masking out invalid positions during the scaled dot-product attention computation (set to $-\infty$) so that the softmax ignores those connections.

4. Key Detail on Masking

- The **masking mechanism** ensures that no "future" positions in the sequence are considered, preserving the sequence generation's left-to-right flow.

These mechanisms form the foundation of modern transformer architectures, enabling tasks like machine translation, text generation, and summarization. Let me know if you'd like further elaboration or visual aids!

explain encoder decoder attention with examples and matrices and dimensions

Encoder-Decoder Attention with Examples, Matrices, and Dimensions

The **encoder-decoder attention** mechanism is a key component of transformer architectures, allowing the decoder to focus on relevant parts of the input sequence (produced by the encoder) while generating the output sequence.

Key Concepts

1. **Inputs:**
   - **Encoder output**: A representation of the input sequence (contextual embeddings).
   - **Decoder query**: The partial output sequence generated up to the current position.
2. **Output:**
   - A context-aware representation of the decoder query, influenced by the relevant portions of the input sequence.
3. **Mechanism:**
   - Queries (Q): From the decoder's previous layer output.
   - Keys (K) and Values (V): From the encoder's final layer output.
   - **Attention weights**: Computed as the scaled dot product of Q and K, followed by softmax normalization.

Step-by-Step Example

1. Inputs

- Input sequence (to encoder): `"I like cats"`
  Tokenized: [101, 200, 300]
- Output sequence (to decoder): `"Me gustan"` (partial output up to position t)
  Tokenized: [401, 500]

2. Encoder Output

- Suppose the encoder processes the input `"I like cats"`.
  Encoder produces a matrix (E) of shape:

  $$E \in \mathbb{R}^{T_{enc} \times d_{model}}$$

  where:

  - T_{enc}: Number of tokens in the input sequence (3 in this case).
  - d_{model}: Dimensionality of each token embedding (e.g., 512).

Example:

$$E = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} 0.1 & 0.3 & 0.5 & \dots & 0.8 \\ 0.2 & 0.4 & 0.6 & \dots & 0.9 \\ 0.3 & 0.5 & 0.7 & \dots & 1.0 \end{bmatrix}$$

3. Decoder Query

- Decoder produces a query matrix (Q) of shape:

$$Q \in \mathbb{R}^{T_{dec} \times d_{model}}$$

where:

  - T_{dec}: Number of tokens in the decoder sequence (2 in this case).

Example:

$$Q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} 0.4 & 0.6 & 0.8 & \dots & 0.9 \\ 0.5 & 0.7 & 0.9 & \dots & 1.0 \end{bmatrix}$$

4. Attention Calculation

1. **Dot Product of Q and K:** Keys (K) are identical to the encoder output (E). Compute attention scores:

$$\text{Attention scores} = Q \cdot K^T$$

Dimensions:

$$(T_{dec} \times d_{model}) \cdot (d_{model} \times T_{enc}) = T_{dec} \times T_{enc}$$

Example:

$$\text{Scores} = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \end{bmatrix}$$

2. **Scaled Dot Product and Softmax:** Scale scores by $\sqrt{d_{model}}$ and apply softmax to normalize across the encoder tokens:

$$\text{Weights} = \text{softmax}\left(\frac{\text{Scores}}{\sqrt{d_{model}}}\right)$$

Example:

$$\text{Weights} = \begin{bmatrix} 0.4 & 0.3 & 0.3 \\ 0.5 & 0.3 & 0.2 \end{bmatrix}$$

3. **Weighted Sum of Values (V):** Values (V) are also identical to the encoder output (E). Use weights to compute context vectors:

$$\text{Output} = \text{Weights} \cdot V$$

Dimensions:

$$(T_{dec} \times T_{enc}) \cdot (T_{enc} \times d_{model}) = T_{dec} \times d_{model}$$

Example:

$$\text{Output} = \begin{bmatrix} 0.2 & 0.5 & 0.7 & \dots & 0.9 \\ 0.3 & 0.6 & 0.8 & \dots & 1.0 \end{bmatrix}$$

Summary of Dimensions

Component	Shape	Description
Encoder Output	$T_{enc} \times d_{model}$	Representations of input tokens.
Decoder Query	$T_{dec} \times d_{model}$	Decoder embeddings for current sequence.
Attention Weights	$T_{dec} \times T_{enc}$	Influence of encoder tokens on decoder tokens.
Attention Output	$T_{dec} \times d_{model}$	Context-aware representations for decoder.

This process is repeated at each decoder layer to refine the attention and contextual understanding.

Would you like visualizations of this?

explain fully encoder decoder attention with examples and matrices and dimensions

Encoder-Decoder Attention: Full Explanation with Examples, Matrices, and Dimensions

Encoder-decoder attention is a crucial part of transformers used in tasks like machine translation. It allows the decoder to focus on specific parts of the input sequence while generating output. Here's a detailed explanation with examples, matrices, and dimensions.

Overview

1. **Encoder** processes the input sequence and produces a context matrix (E).
2. **Decoder** takes:
    - The encoder's context (E) as **keys (K) and values (V)**.
    - Its own previous layer output as **queries (Q)**.
3. The **attention mechanism** computes a weighted sum of the encoder's values based on the similarity between the decoder's queries and the encoder's keys.

Example

Input and Output Sequences

- **Input sequence**: `"I like cats"`
  Tokenized: [101, 200, 300]
  Input length (T_{enc}) = 3
- **Output sequence**: `"Me gustan"` (up to time step t)
  Tokenized: [401, 500]
  Output length (T_{dec}) = 2 (at this point)
- Embedding dimensionality (d_{model}): 4 (for simplicity).

Step-by-Step Explanation

1. Encoder Output

The encoder transforms the input sequence into contextual embeddings.

- Encoder output matrix (E):

$$E \in \mathbb{R}^{T_{enc} \times d_{model}} = \mathbb{R}^{3 \times 4}$$

  Example encoder output:

$$E = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 \end{bmatrix}$$

Each row represents the embedding of a token in the input sequence.

---

2. Decoder Query

The decoder generates a query matrix (Q) based on its own output up to time step t.

- Decoder query matrix (Q):

$$Q \in \mathbb{R}^{T_{dec} \times d_{model}} = \mathbb{R}^{2 \times 4}$$

Example decoder query:

$$Q = \begin{bmatrix} 0.2 & 0.4 & 0.6 & 0.8 \\ 0.1 & 0.3 & 0.5 & 0.7 \end{bmatrix}$$

---

3. Attention Mechanism

Step 1: Compute Attention Scores

The attention scores are computed as the dot product of Q and K^T (where $K = E$).

$$\text{Scores} = Q \cdot K^T$$

- Dimensions:

$$(T_{dec} \times d_{model}) \cdot (d_{model} \times T_{enc}) = T_{dec} \times T_{enc}$$

Example computation:

$$Q \cdot K^T = \begin{bmatrix} 0.2 & 0.4 & 0.6 & 0.8 \\ 0.1 & 0.3 & 0.5 & 0.7 \end{bmatrix} \cdot \begin{bmatrix} 0.1 & 0.5 & 0.9 \\ 0.2 & 0.6 & 1.0 \\ 0.3 & 0.7 & 1.1 \\ 0.4 & 0.8 & 1.2 \end{bmatrix}$$

Resulting attention scores:

$$\text{Scores} = \begin{bmatrix} 0.3 & 0.7 & 1.1 \\ 0.26 & 0.62 & 0.98 \end{bmatrix}$$

Step 2: Scale the Scores

The scores are scaled by dividing by $\sqrt{d_{model}}$ to stabilize gradients:

$$\text{Scaled Scores} = \frac{\text{Scores}}{\sqrt{d_{model}}}$$

For $d_{model} = 4$:

$$\text{Scaled Scores} = \begin{bmatrix} 0.15 & 0.35 & 0.55 \\ 0.13 & 0.31 & 0.49 \end{bmatrix}$$

Step 3: Apply Softmax

Softmax is applied to normalize the scores across the encoder positions (T_{enc}):

$$\text{Attention Weights} = \text{softmax}(\text{Scaled Scores}, \text{axis} = 1)$$

Example:

$$\text{Attention Weights} = \begin{bmatrix} 0.2 & 0.3 & 0.5 \\ 0.25 & 0.35 & 0.4 \end{bmatrix}$$

Step 4: Compute the Context Vector

The context vector is the weighted sum of the encoder's values ($V = E$):

$$\text{Context} = \text{Attention Weights} \cdot V$$

- Dimensions:

  $$(T_{dec} \times T_{enc}) \cdot (T_{enc} \times d_{model}) = T_{dec} \times d_{model}$$

  Example computation:

  $$\text{Context} = \begin{bmatrix} 0.2 & 0.3 & 0.5 \\ 0.25 & 0.35 & 0.4 \end{bmatrix} \cdot \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 \end{bmatrix}$$

  Resulting context:

  $$\text{Context} = \begin{bmatrix} 0.62 & 0.72 & 0.82 & 0.92 \\ 0.65 & 0.75 & 0.85 & 0.95 \end{bmatrix}$$

4. Output of Encoder-Decoder Attention

The context vector (Context) is passed to the next decoder layer, providing information about the relevant parts of the input sequence for the current decoding step.

Summary of Dimensions

Component	Dimensions
Encoder Output ($E = K = V$)	$T_{enc} \times d_{model}$
Decoder Query (Q)	$T_{dec} \times d_{model}$
Attention Scores	$T_{dec} \times T_{enc}$
Attention Weights	$T_{dec} \times T_{enc}$
Context Vector (Context)	$T_{dec} \times d_{model}$

Let me know if you'd like to delve deeper into any specific part!

explain decoder self attention Similarly, self-attention layers in the decoder allow each position in the decoder to attend to
all positions in the decoder up to and including that position. We need to prevent leftward
information flow in the decoder to preserve the auto-regressive property. We implement this
inside of scaled dot-product attention by masking out (setting to −∞) all values in the input
of the softmax which correspond to illegal connections.

Decoder Self-Attention: Detailed Explanation with Examples, Matrices, and Dimensions

Decoder self-attention is a mechanism that allows each position in the decoder to attend to all previous positions, including itself, while preventing attention to future positions. This ensures the **auto-regressive property**, crucial for generating sequences step by step in tasks like text generation or machine translation.

Key Characteristics

1. **Input Source**: Keys (K), queries (Q), and values (V) all come from the decoder's own output.
2. **Masking**: A triangular mask is applied to prevent attending to future tokens, ensuring left-to-right sequence generation.
3. **Purpose**: Enables the decoder to use information from already-generated tokens while maintaining causality.

Step-by-Step Explanation

1. Inputs to Self-Attention

Suppose the decoder is generating an output sequence:
Partial output at step t: `"Me gustan"`, tokenized as [401, 500].

- Decoder's current input:

  $$\text{Decoder embeddings: } D \in \mathbb{R}^{T_{dec} \times d_{model}}$$

  where:

  - T_{dec}: Number of tokens generated so far ($T_{dec} = 2$).
  - d_{model}: Dimensionality of embeddings (e.g., $d_{model} = 4$).

  Example:

  $$D = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \end{bmatrix}$$

2. Queries, Keys, and Values

In self-attention, Q, K, and V are derived from the same matrix (D).

- Queries (Q):

  $$Q = W_Q \cdot D \quad \in \mathbb{R}^{T_{dec} \times d_k}$$

- Keys (K):

  $$K = W_K \cdot D \quad \in \mathbb{R}^{T_{dec} \times d_k}$$

- Values (V):

$$V = W_V \cdot D \quad \in \mathbb{R}^{T_{dec} \times d_v}$$

Here, W_Q, W_K, and W_V are learnable weight matrices. For simplicity, assume $d_k = d_v = d_{model}$.

---

3. Compute Attention Scores

The attention scores are computed as the scaled dot product of Q and K^T:

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

- Dimensions of Q: $\mathbb{R}^{T_{dec} \times d_k}$.
- Dimensions of K^T: $\mathbb{R}^{d_k \times T_{dec}}$.
- Result: $\text{Scores} \in \mathbb{R}^{T_{dec} \times T_{dec}}$.

Example scores (before masking):

$$\text{Scores} = \begin{bmatrix} 1.0 & 0.8 \\ 0.9 & 1.1 \end{bmatrix}$$

---

4. Apply Masking

A triangular mask is applied to ensure each position attends only to itself and previous positions.
Mask:

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

After masking:

$$\text{Masked Scores} = \begin{bmatrix} 1.0 & -\infty \\ 0.9 & 1.1 \end{bmatrix}$$

---

5. Compute Attention Weights

The softmax function is applied row-wise to the masked scores to normalize them:

$$\text{Weights} = \text{softmax}(\text{Masked Scores})$$

Example:

$$\text{Weights} = \begin{bmatrix} 1.0 & 0.0 \\ 0.4 & 0.6 \end{bmatrix}$$

---

6. Compute Attention Output

The attention output is a weighted sum of the values (V):

$$\text{Output} = \text{Weights} \cdot V$$

Example values (V):

$$V = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.8 \end{bmatrix}$$

Output:

$$\text{Output} = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ 0.34 & 0.44 & 0.54 & 0.64 \end{bmatrix}$$

Summary

1. Decoder self-attention allows each token to focus on itself and previous tokens.
2. A mask prevents access to future tokens, ensuring causality.
3. The output is a contextually enriched embedding that the decoder uses for subsequent layers and final predictions.

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

in decoder attention multi head attention is there or not?

Yes, **multi-head attention** is used in the **decoder self-attention** mechanism as well as in **encoder-decoder attention** within transformer architectures. Multi-head attention is a core component of the Transformer model and is applied uniformly across all attention mechanisms, including those in the decoder.

Multi-Head Attention in Decoder Attention

Types of Attention in the Decoder:

1. **Decoder Self-Attention**:
   - Each token in the decoder attends to all previous tokens (up to and including itself).
   - Uses masking to prevent access to "future" tokens, preserving the auto-regressive property.
   - Multi-head attention is applied here to allow the model to learn from multiple representation subspaces simultaneously.
2. **Encoder-Decoder Attention**:
   - Each token in the decoder attends to the entire sequence produced by the encoder.
   - Multi-head attention enables the decoder to focus on various parts of the encoder's output for richer context.

How Multi-Head Attention Works in Decoder Attention

1. Splitting into Multiple Heads

- The queries (Q), keys (K), and values (V) are projected into multiple smaller dimensions:

  $$Q_h = W_Q^h \cdot Q, \quad K_h = W_K^h \cdot K, \quad V_h = W_V^h \cdot V$$

  for each head h ($h \in [1, H]$) where H is the number of attention heads.
- For example, if $d_{model} = 512$ and $H = 8$, each head operates on $\frac{d_{model}}{H} = 64$-dimensional subspaces.

2. Parallel Attention Calculation

Each head computes its attention scores independently:

$$\text{Attention}_h = \text{softmax}\left(\frac{Q_h \cdot K_h^T}{\sqrt{d_k}}\right) \cdot V_h$$

3. Concatenation of Heads

The outputs of all heads are concatenated and linearly transformed:

$$\text{MultiHead}(Q, K, V) = W_O \cdot \text{Concat}(\text{Attention}_1, \ldots, \text{Attention}_H)$$

where W_O is a learnable weight matrix.

Advantages of Multi-Head Attention in Decoder

1. **Richer Representations**: Different heads can focus on different aspects of the sequence, such as syntax, semantics, or specific patterns.
2. **Parallel Processing**: Attention is computed in parallel across multiple heads, making it efficient and effective.
3. **Learning Subspaces**: The model learns diverse features from different subspaces, improving overall performance.

Conclusion

Yes, **multi-head attention is present in both decoder self-attention and encoder-decoder attention**. It plays a vital role in enabling the decoder to understand the input context (encoder output) and generate coherent output sequences.

explain Decoder attention with multi heads and weights and dimensions

Decoder Attention with Multi-Head Attention: Weights and Dimensions

Multi-head attention in the **decoder** comprises two types:

1. **Decoder Self-Attention**: Tokens in the decoder attend to earlier tokens in the output sequence.
2. **Encoder-Decoder Attention**: Decoder attends to the output of the encoder for context.

Both use multi-head attention to improve representational power. Here's a detailed explanation, including weights, matrices, and dimensions.

Overview of Multi-Head Attention

1. **Inputs**:
   - **Queries (Q)**: From the decoder's previous layer.
   - **Keys (K) and Values (V)**:
     - **Decoder self-attention**: From the decoder's previous layer.
     - **Encoder-decoder attention**: From the encoder output.
2. **Mechanism**:
   - Divide Q, K, and V into multiple "heads."
   - Perform scaled dot-product attention independently for each head.
   - Concatenate the results and apply a linear transformation.

Step-by-Step Explanation

1. Inputs

- Suppose:
  - Decoder input sequence: `"Me gustan"`, tokenized as [401, 500].
  - Embedding dimension: $d_{model} = 8$ (for simplicity).
  - Number of heads: $H = 2$.
- Decoder input embeddings (D):

$$D \in \mathbb{R}^{T_{dec} \times d_{model}} = \mathbb{R}^{2 \times 8}$$

Example:

$$D = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 & 1.3 & 1.4 & 1.5 & 1.6 \end{bmatrix}$$

---

2. Linear Transformations

Each head has separate linear transformations for Q, K, and V. Suppose $d_k = d_v = \frac{d_{model}}{H} = 4$.

Queries (Q_h):

- Linear transformation:
$$Q_h = D \cdot W_Q^h, \quad W_Q^h \in \mathbb{R}^{d_{model} \times d_k}$$
- Example weight matrix for head 1:
$$W_Q^1 = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \\ \vdots & \vdots & \vdots & \vdots \\ 0.5 & 0.6 & 0.7 & 0.8 \end{bmatrix} \quad \in \mathbb{R}^{8 \times 4}$$
- Result:
$$Q_h \in \mathbb{R}^{T_{dec} \times d_k} = \mathbb{R}^{2 \times 4}$$

Keys (K_h) and Values (V_h):

Similarly:

$$K_h = D \cdot W_K^h, \quad V_h = D \cdot W_V^h$$

---

3. Scaled Dot-Product Attention

For each head:

$$\text{Attention}_h = \text{softmax}\left(\frac{Q_h \cdot K_h^T}{\sqrt{d_k}}\right) \cdot V_h$$

- Dimensions:
  - $Q_h \in \mathbb{R}^{T_{dec} \times d_k}$

- $K_h^T \in \mathbb{R}^{d_k \times T_{dec}}$
- Result: Attention weights $\in \mathbb{R}^{T_{dec} \times T_{dec}}$
- Final output: $\text{Attention}_h \in \mathbb{R}^{T_{dec} \times d_k}$

---

4. Concatenation and Linear Transformation

- Concatenate the results of all heads:
$$\text{Concat}(\text{Attention}_1, \text{Attention}_2, \ldots, \text{Attention}_H) \in \mathbb{R}^{T_{dec} \times d_{model}}$$
- Apply a final linear transformation:
$$\text{Output} = \text{Concat} \cdot W_O, \quad W_O \in \mathbb{R}^{d_{model} \times d_{model}}$$

Masking in Decoder Self-Attention

To preserve the auto-regressive property:

- Mask future tokens by setting their attention scores to $-\infty$.
- Example attention matrix for $T_{dec} = 3$:

$$\text{Masked Scores} = \begin{bmatrix} 1 & -\infty & -\infty \\ 1 & 1 & -\infty \\ 1 & 1 & 1 \end{bmatrix}$$

Final Dimensions

- Input embeddings: $\mathbb{R}^{T_{dec} \times d_{model}}$
- Queries, Keys, Values per head: $\mathbb{R}^{T_{dec} \times d_k}$
- Attention output per head: $\mathbb{R}^{T_{dec} \times d_k}$
- Concatenated output: $\mathbb{R}^{T_{dec} \times d_{model}}$

Conclusion

Multi-head attention in the decoder allows parallel processing of different representation subspaces. Masking ensures causality, and the final linear transformation integrates information across all heads. This makes the decoder capable of generating coherent, sequential outputs.

show with masked attention also

Decoder Attention with Multi-Head Masked Attention: Weights, Dimensions, and Masking

Masked attention in the decoder ensures the **auto-regressive property**—i.e., at each position in the output sequence, a token can only attend to itself and previous tokens, not future tokens. This is achieved by applying a **causal mask** during the computation of attention scores.

Let's step through a **multi-head masked attention mechanism** with all the necessary weights, dimensions, and the masking process.

Scenario Setup

1. **Decoder input sequence**: `"Me gustan"`, tokenized as [401, 500].
2. **Embedding dimensionality (d_{model})**: 8.
3. **Number of attention heads (H)**: 2.
4. **Length of decoder sequence (T_{dec})**: 2.
5. **Per-head dimensions**:
   - Query/key/value dimensions: $d_k = d_v = \frac{d_{model}}{H} = 4$.

Step-by-Step Explanation

1. Input Embeddings

The decoder input embeddings matrix (D):

$$D \in \mathbb{R}^{T_{dec} \times d_{model}} = \mathbb{R}^{2\times 8}$$

Example:

$$D = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 1.1 & 1.2 & 1.3 & 1.4 & 1.5 & 1.6 \end{bmatrix}$$

---

2. Linear Transformations

For each head h, separate projection weights W_Q^h, W_K^h, and W_V^h are applied:

$$Q_h = D \cdot W_Q^h, \quad K_h = D \cdot W_K^h, \quad V_h = D \cdot W_V^h$$

- Projection weight matrices for **head 1** ($h = 1$):

$$W_Q^1, W_K^1, W_V^1 \in \mathbb{R}^{d_{model} \times d_k} = \mathbb{R}^{8\times 4}$$

- Output dimensions for **queries, keys, and values** per head:

$$Q_h, K_h, V_h \in \mathbb{R}^{T_{dec} \times d_k} = \mathbb{R}^{2\times 4}$$

---

3. Compute Attention Scores

For each head, compute the **scaled dot-product attention scores**:

$$\text{Scores}_h = \frac{Q_h \cdot K_h^T}{\sqrt{d_k}}$$

- Dimensions:

$$Q_h \in \mathbb{R}^{T_{dec} \times d_k}, \quad K_h^T \in \mathbb{R}^{d_k \times T_{dec}}$$

  Result:

$$\text{Scores}_h \in \mathbb{R}^{T_{dec} \times T_{dec}} = \mathbb{R}^{2\times 2}$$

Example scores before masking:

$$\text{Scores}_h = \begin{bmatrix} 1.0 & 0.8 \\ 0.9 & 1.1 \end{bmatrix}$$

---

4. Apply Masking

To prevent "future" attention, a **causal mask** is applied. This mask sets invalid positions to $-\infty$, ensuring they contribute zero weight after the softmax.

Mask matrix:

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

Masked scores:

$$\text{Scores}_h = \begin{bmatrix} 1.0 & -\infty \\ 0.9 & 1.1 \end{bmatrix}$$

5. Compute Softmax

Apply softmax row-wise to convert scores into attention weights:

$$\text{Attention}_h = \text{softmax}(\text{Scores}_h)$$

Example:

$$\text{Softmax}(\text{Scores}_h) = \begin{bmatrix} 1.0 & -\infty \\ 0.9 & 1.1 \end{bmatrix} \rightarrow \begin{bmatrix} 1.0 & 0.0 \\ 0.4 & 0.6 \end{bmatrix}$$

6. Compute Weighted Values

Use the attention weights to compute the weighted sum of values (V_h):

$$\text{Output}_h = \text{Attention}_h \cdot V_h$$

- Dimensions:

$$\text{Attention}_h \in \mathbb{R}^{T_{dec} \times T_{dec}}, \quad V_h \in \mathbb{R}^{T_{dec} \times d_k}$$

  Result:

$$\text{Output}_h \in \mathbb{R}^{T_{dec} \times d_k} = \mathbb{R}^{2 \times 4}$$

7. Concatenate Heads

Concatenate the outputs of all heads:

$$\text{Concat}(\text{Output}_1, \text{Output}_2) \in \mathbb{R}^{T_{dec} \times d_{model}}$$

Example (if $H = 2$):

$$\text{Concat} = \begin{bmatrix} 0.4 & 0.3 & 0.2 & 0.1 & 0.5 & 0.6 & 0.7 & 0.8 \\ 0.9 & 1.0 & 0.8 & 0.7 & 1.3 & 1.4 & 1.2 & 1.1 \end{bmatrix}$$

---

8. Final Linear Transformation

Apply a final linear transformation to combine all heads:

$$\text{Output} = \text{Concat} \cdot W_O, \quad W_O \in \mathbb{R}^{d_{model} \times d_{model}}$$

Final output dimensions:

$$\text{Output} \in \mathbb{R}^{T_{dec} \times d_{model}} = \mathbb{R}^{2 \times 8}$$

Summary

1. **Masked attention ensures causality** by using a triangular mask to block future tokens.
2. **Multi-head attention enables learning from multiple representation subspaces**, enriching the model's capability.
3. The final output is passed to the next layer of the decoder or used to generate predictions.

This mechanism is fundamental to auto-regressive generation tasks like machine translation and text generation!

we apply masks for each head?

Yes, **masks are applied independently to each head** in multi-head attention. The masking ensures that the **auto-regressive property** (causality) is preserved for each attention head in the **decoder self-attention** mechanism.

Why Apply Masks Per Head?

1. **Independent Computation**: Each attention head calculates attention scores independently for its subspace of representations.
2. **Causality**: To prevent a token from "seeing" future tokens, a mask is applied to the attention scores before the softmax operation for each head.
3. **Consistency**: This ensures that even with multiple attention heads, the decoder only attends to positions it is allowed to.

How Masks Are Applied

The scaled dot-product attention for each head involves these steps:

Step 1: Compute Attention Scores

For each head:

$$\text{Scores}_h = \frac{Q_h \cdot K_h^T}{\sqrt{d_k}}$$

- Dimensions:

  $$Q_h \in \mathbb{R}^{T_{dec} \times d_k}, \quad K_h^T \in \mathbb{R}^{d_k \times T_{dec}}$$

  Result:

  $$\text{Scores}_h \in \mathbb{R}^{T_{dec} \times T_{dec}}$$

Step 2: Apply the Mask

A **causal mask** (triangular mask) is used to block attention to future tokens. The mask matrix M has the following form for $T_{dec} = 4$:

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

- **Mask application**:

$$\text{Scores}_h = \text{Scores}_h + M$$

- This ensures that any attention to future tokens (beyond the current position) results in a large negative value, effectively making their softmax output near zero.

Step 3: Softmax

The masked scores are normalized using the softmax function:

$$\text{Attention}_h = \text{softmax}(\text{Scores}_h) \cdot V_h$$

- Dimensions:
  - $\text{Scores}_h \in \mathbb{R}^{T_{dec} \times T_{dec}}$
  - $V_h \in \mathbb{R}^{T_{dec} \times d_v}$
  - Result:

$$\text{Attention}_h \in \mathbb{R}^{T_{dec} \times d_v}$$

Step 4: Concatenate Heads

The outputs of all H heads are concatenated:

$$\text{Concat}(\text{Attention}_1, \ldots, \text{Attention}_H) \in \mathbb{R}^{T_{dec} \times d_{model}}$$

- This is followed by a linear transformation with weights $W_O \in \mathbb{R}^{d_{model} \times d_{model}}$.

Conclusion

Yes, **masks are applied for each attention head** during the computation of decoder self-attention. This ensures that no matter the number of heads, the causality constraint is preserved independently within each head. The process of masking, softmax, and computation happens per head before combining the outputs.