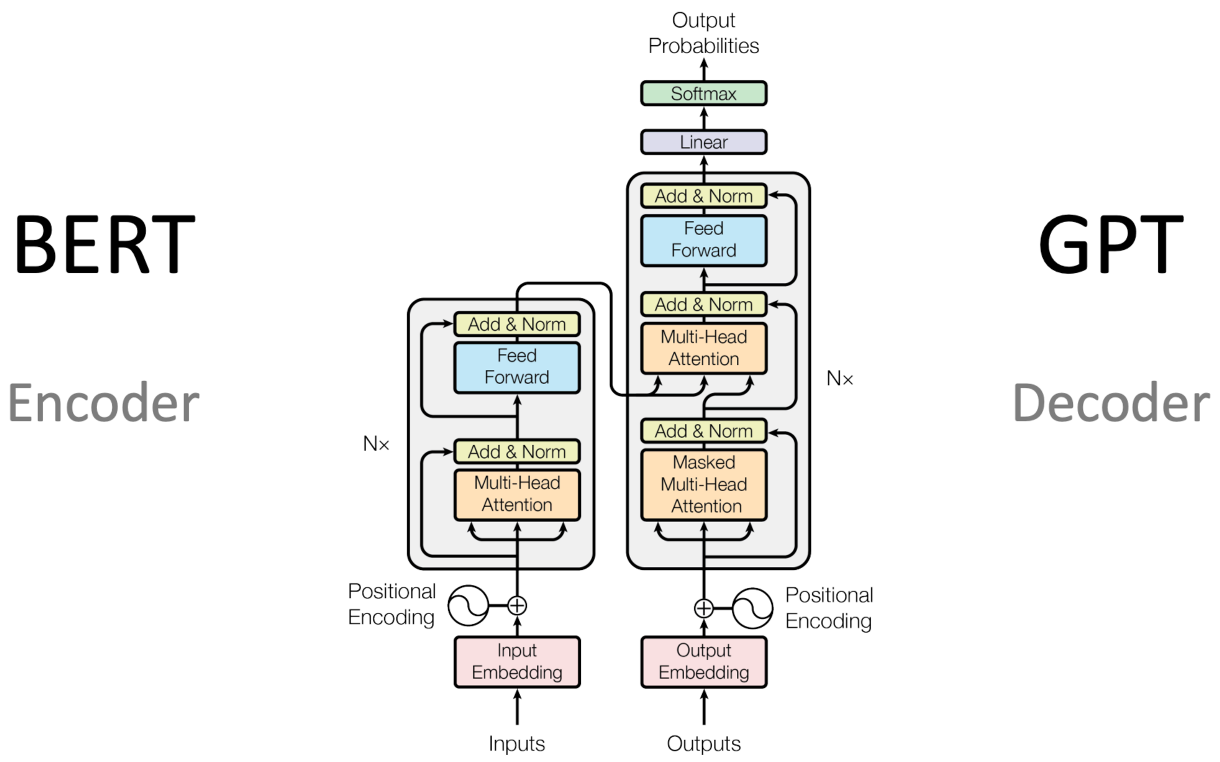
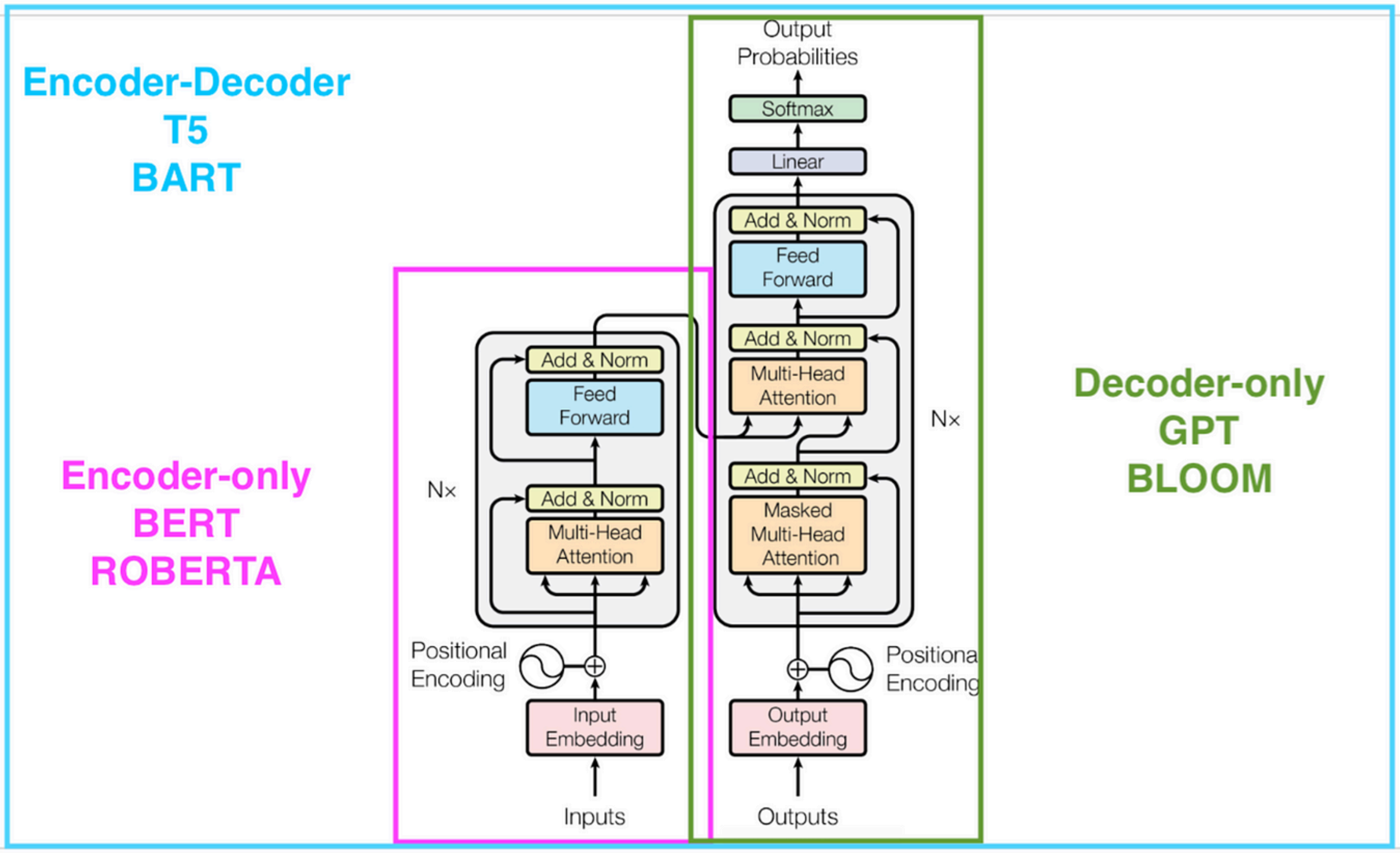
"Attention is All You Need" Transformer Architecture



The Transformer architecture revolutionized the field of natural language processing by eliminating recurrence and convolutions, relying entirely on attention mechanisms. Below is a comprehensive breakdown of its key components.

1. Encoder-Decoder Architecture



What?

- The Transformer follows an Encoder-Decoder architecture:
  - **Encoder**: Processes the input sequence and converts it into a continuous representation (embeddings).
  - **Decoder**: Uses the embeddings to generate an output sequence, such as a translated sentence.

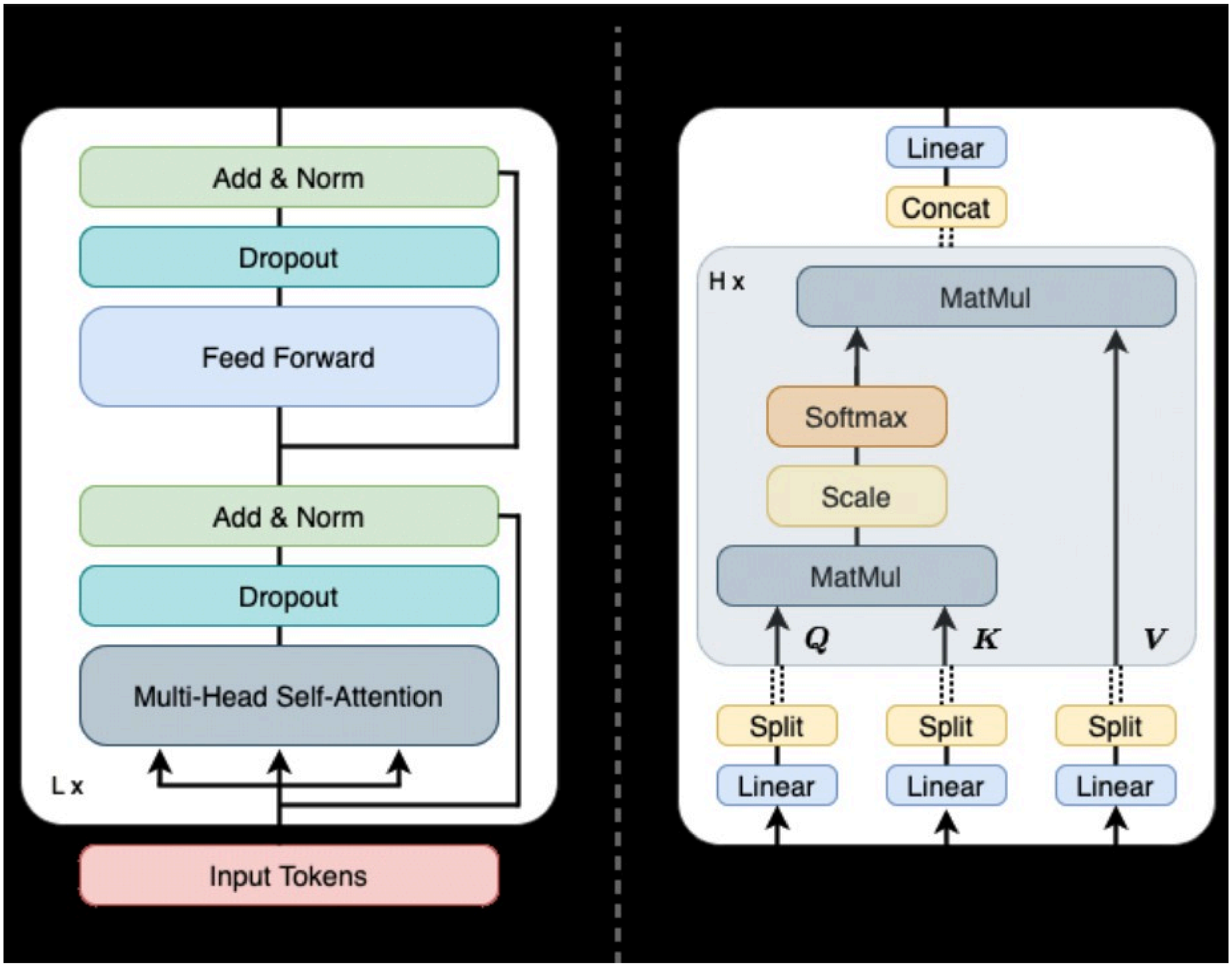
Why?

- The Encoder-Decoder structure allows the model to handle sequence-to-sequence tasks (e.g., translation, summarization) by mapping input sequences to output sequences while retaining context information.

Components

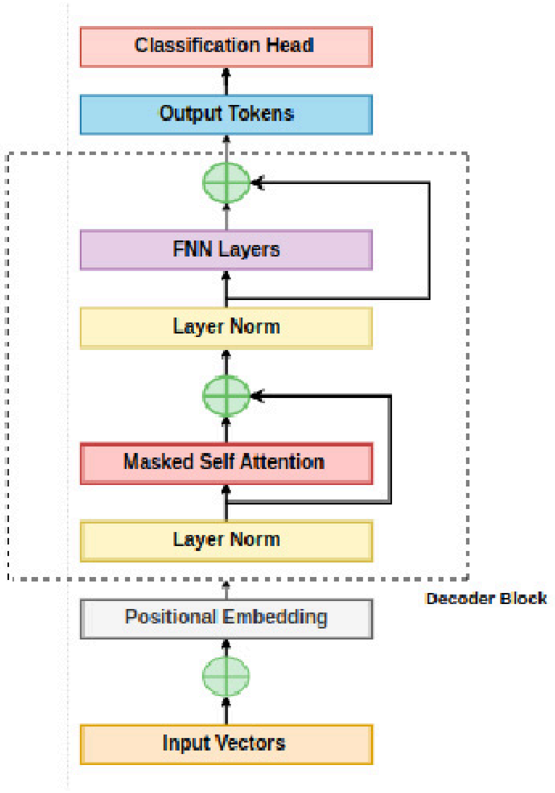
Encoder:

- The **encoder** consists of six identical layers, each with two main sub-layers: Multi-Head Self-Attention and FeedForward network.



Decoder:

- The **decoder** also has six layers but includes an additional Multi-Head Cross-Attention sub-layer to incorporate the encoder's output representations



2. Token Embeddings

What?

- Each input token is represented as a dense vector of fixed size (usually 512 or 1024 dimensions).

Why?

- The Transformer can't process raw text, so tokens must be embedded in continuous vector space for better semantic representation.

How?

- Example: Let's consider the sentence: **"I love NLP"**. Each word is mapped to an embedding vector:

Token	Embedding (512 dimensions)
"I"	[0.21, -0.34, 0.56, ...]
"love"	[-0.12, 0.41, -0.73, ...]
"NLP"	[0.45, -0.67, 0.12, ...]

These embeddings are passed to the next layer. This ensures the model gets dense, continuous representations of discrete tokens.

3. Position Encodings (Sinusoidal)

What?

- Position encodings provide positional information to the model, which otherwise lacks sequential awareness (unlike RNNs).

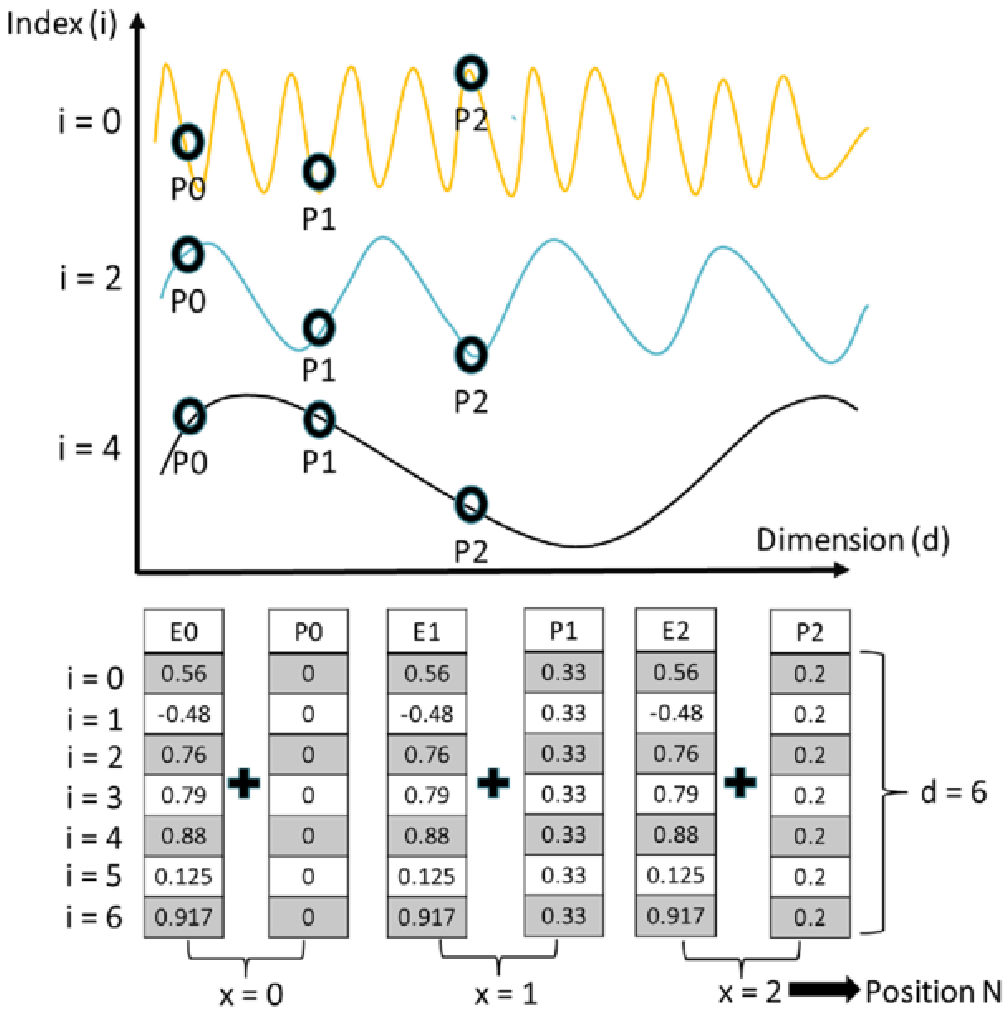
Why?

- Self-attention computes relationships without any regard for sequence order. Positional encodings help the model understand where a word is positioned in the input.

How?

- Position encodings are added to token embeddings using sine and cosine functions with different wavelengths.

For an input sequence: **"I love NLP"** (Length = 3 tokens), let's assume the model dimension $d_{model} = 4$ for simplicity.



	E0	P0	E1	P1	E2	P2
i = 0	0.56	0	0.56	0.33	0.56	0.2
i = 1	-0.48	0	-0.48	0.33	-0.48	0.2
i = 2	0.76	0	0.76	0.33	0.76	0.2
i = 3	0.79	0	0.79	0.33	0.79	0.2
i = 4	0.88	0	0.88	0.33	0.88	0.2
i = 5	0.125	0	0.125	0.33	0.125	0.2
i = 6	0.917	0	0.917	0.33	0.917	0.2

Formula:

- For even dimensions:

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

- For odd dimensions:

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

Calculate Position Encoding for Each Position

Position	PE (dim 0)	PE (dim 1)	PE (dim 2)	PE (dim 3)
0	sin(0) = 0	cos(0) = 1	sin(0) = 0	cos(0) = 1
1	sin(1/10000^0) ≈ 0.8415	cos(1/10000^0) ≈ 0.5403	sin(1/10000^0) ≈ 0.8415	cos(1/10000^0) ≈ 0.5403
2	sin(2/10000^0) ≈ 0.9093	cos(2/10000^0) ≈ -0.4161	sin(2/10000^0) ≈ 0.9093	cos(2/10000^0) ≈ -0.4161
3	sin(3/10000^0) ≈ 0.1411	cos(3/10000^0) ≈ -0.9899	sin(3/10000^0) ≈ 0.1411	cos(3/10000^0) ≈ -0.9899

Thus, the position encodings for a sequence of length 4 are:

$$PE = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0.8415 & 0.5403 & 0.8415 & 0.5403 \\ 0.9093 & -0.4161 & 0.9093 & -0.4161 \\ 0.1411 & -0.9899 & 0.1411 & -0.9899 \end{bmatrix}$$

Combining Position Encoding with Token Embeddings

Now that we have position encodings, we add them to the token embeddings to provide positional context.

Assuming we have token embeddings as follows for a sequence of length 4 (embedding dimension 4):

Token	Token Embedding
0	$[0.2, 0.1, 0.3, 0.4]$
1	$[0.5, 0.4, 0.3, 0.2]$
2	$[0.9, 0.8, 0.7, 0.6]$
3	$[0.1, 0.2, 0.3, 0.4]$

Combine token embeddings and position encodings:

1. For token 0:

$$\text{Token Embedding} + PE = \begin{bmatrix} 0.2 \\ 0.1 \\ 0.3 \\ 0.4 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 1.1 \\ 0.3 \\ 2.4 \end{bmatrix}$$

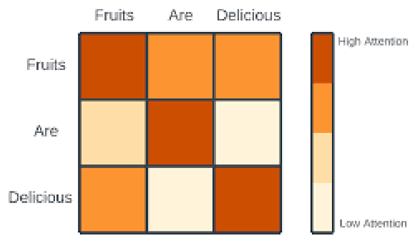
2. For token 1:

$$\begin{bmatrix}0.5\\0.4\\0.3\\0.2\end{bmatrix}+\begin{bmatrix}0.8415\\0.5403\\0.8415\\0.5403\end{bmatrix}=\begin{bmatrix}1.3415\\0.9403\\1.1415\\0.7403\end{bmatrix}$$

3. Repeat for tokens 2 and 3.

The final combined embeddings serve as inputs to the model, retaining both the semantic meaning of the tokens and their positions in the sequence.

4. Attention Mechanisms



a. Scaled Dot-Product Attention

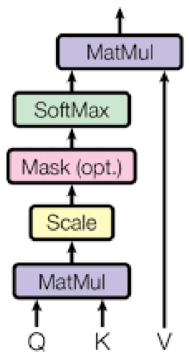
What?

- A core mechanism where attention weights are computed by taking the dot product between queries and keys.

Why?

- It allows the model to focus on relevant parts of the input by computing the similarity between tokens.

How?



- Given query Q, key K, and value V matrices, the attention mechanism computes:

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

  - d_k is the dimensionality of the key vectors, and the scaling factor $\frac{1}{\sqrt{d_k}}$ stabilizes gradients when d_k is large.
  - The softmax function normalizes the weights to focus on relevant tokens.

Step By Step Example

Given query Q, key K, and value V matrices:

Assume Q, K, V are 2x2 matrices (simplified to demonstrate):

Q	K	V
[0.1, 0.4]	[0.2, 0.5]	[1.2, 0.9]
[0.3, 0.7]	[0.6, 0.3]	[0.8, 1.1]

Dot-product of Q and K^T:

$$QK^T = \begin{bmatrix} 0.1 & 0.4 \\ 0.3 & 0.7 \end{bmatrix} \times \begin{bmatrix} 0.2 & 0.6 \\ 0.5 & 0.3 \end{bmatrix} = \begin{bmatrix} 0.28 & 0.22 \\ 0.54 & 0.42 \end{bmatrix}$$

Scale by $\sqrt{d_k}$ (where $d_k = 2$):

$$\frac{QK^T}{\sqrt{2}} = \frac{1}{\sqrt{2}} \times \begin{bmatrix} 0.28 & 0.22 \\ 0.54 & 0.42 \end{bmatrix} = \begin{bmatrix} 0.198 & 0.155 \\ 0.382 & 0.297 \end{bmatrix}$$

Apply softmax to get attention weights:

$$\text{softmax} = \text{softmax}\left(\frac{QK^T}{\sqrt{2}}\right) = \begin{bmatrix} 0.537 & 0.463 \\ 0.521 & 0.479 \end{bmatrix}$$

Multiply by V:

$$\text{Attention}(Q, K, V) = \begin{bmatrix} 0.537 & 0.463 \\ 0.521 & 0.479 \end{bmatrix} \times \begin{bmatrix} 1.2 & 0.9 \\ 0.8 & 1.1 \end{bmatrix} = \begin{bmatrix} 1.027 & 0.993 \\ 1.016 & 1.001 \end{bmatrix}$$

Thus, the final attention output is:

$$\begin{bmatrix} 1.027 & 0.993 \\ 1.016 & 1.001 \end{bmatrix}$$

Example:

- If you're translating "I love cats" into another language, the model attends more to "love" and "cats" when generating the word corresponding to "love."

Query, Key, and Value Vectors in Attention Mechanisms

Overview

Query, key, and value vectors are fundamental components of attention mechanisms in transformer models. They allow the model to focus on relevant parts of the input when processing sequences.

1. **Query (Q)**: Represents what we're looking for

2. **Key (K)**: Represents what we have available
3. **Value (V)**: Represents the content we want to retrieve

How They Work Together

1. For each position in the sequence, we compute attention scores by comparing its query vector with the key vectors of all positions.
2. These scores are then used to create a weighted sum of the value vectors.

Detailed Explanation

1. **Creation**:
   - Q, K, and V are created by multiplying the input embeddings with learned weight matrices.
   - Q = X * W_Q
   - K = X * W_K
   - V = X * W_V
     (Where X is the input and W_Q, W_K, W_V are learnable parameter matrices)
2. **Attention Calculation**:
   - Attention scores = softmax((Q * K^T) / sqrt(d_k))
   - Where d_k is the dimension of the key vectors
3. **Output**:
   - The final output is the weighted sum of value vectors
   - Output = Attention scores * V

Query, Key, and Value: Library Catalog and Attention Mechanism

Library Catalog System

1. A user submits a search query for books.
2. The system matches this query against book categories (keys).
3. Books in categories that closely match the query are given higher priority.
4. The system retrieves book information (values) based on these priorities.
5. A list of books is presented, ordered by relevance to the user's query.

Attention Mechanism in Neural Networks

1. The model processes input data as query vectors.
2. These query vectors are compared with key vectors.
3. Attention scores are calculated based on the similarity between queries and keys.
4. The model uses these scores to weight the importance of value vectors.
5. A weighted sum of value vectors forms the output, focusing on the most relevant information.

Technical Implementation in Attention Mechanisms

- Attention scores = softmax((Q * K^T) / sqrt(d_k))
- Output = Attention scores * V
- Q, K, and V are matrices containing query, key, and value vectors for all elements in the input sequence. The term d_k represents the dimension of the key vectors.
- This mechanism enables neural networks to dynamically focus on different parts of the input, processing complex sequences of information effectively.

b. Multi-Head Self-Attention

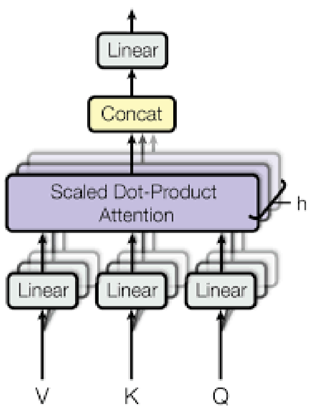
What?

- Instead of using a single attention mechanism, Multi-Head Self-Attention splits the input into multiple attention "heads."

Why?

- This allows the model to learn multiple relationships and representations from different subspaces, capturing richer interactions between tokens.

How?



- For each input sequence, the model computes several attention heads in parallel:

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \mathrm{head}_2, \ldots, \mathrm{head}_h)W^O$$

  - Each head head_i performs scaled dot-product attention on Q, K, V, and the results are concatenated and linearly transformed using W^O.

c. Multi-Head Cross-Attention

What?

- In the decoder, Multi-Head Cross-Attention uses the encoder's output as keys and values while the query comes from the decoder.

Why?

- It helps the decoder focus on relevant parts of the encoded input sequence, improving translation and generation quality.

How?

Multi-Head Cross-Attention differs from self-attention in that the queries Q come from the decoder, while the keys K and values V come from the encoder output.

Step-by-Step Example

Assume:

- We are working with two heads.
- Queries Q come from the decoder and have shape $(2, 3)$ for a sequence length of 2 and hidden size 3.
- Keys K and Values V come from the encoder with the same shape $(2, 3)$.

Here are example inputs:

Decoder Query Q	Encoder Key K	Encoder Value V
[0.2, 0.5, 0.1]	[0.3, 0.8, 0.5]	[0.9, 1.2, 0.6]
[0.7, 0.1, 0.9]	[0.1, 0.2, 0.4]	[0.5, 0.4, 0.9]

Step 1: Linear projections for Q, K, and V for each head.

Assume each head has its own projection weights (randomly initialized for simplicity):

- $W_Q = \begin{bmatrix} 0.1 & 0.3 & 0.2 \\ 0.2 & 0.4 & 0.6 \end{bmatrix}$
- $W_K = \begin{bmatrix} 0.5 & 0.1 & 0.3 \\ 0.4 & 0.7 & 0.2 \end{bmatrix}$
- $W_V = \begin{bmatrix} 0.2 & 0.6 & 0.3 \\ 0.8 & 0.2 & 0.4 \end{bmatrix}$

Now, compute the projections:

For Q_1 (query for head 1):

$$Q_1 = Q \times W_Q^T = \begin{bmatrix} 0.2 & 0.5 & 0.1 \\ 0.7 & 0.1 & 0.9 \end{bmatrix} \times \begin{bmatrix} 0.1 & 0.3 \\ 0.3 & 0.2 \\ 0.2 & 0.6 \end{bmatrix} = \begin{bmatrix} 0.27 & 0.27 \\ 0.36 & 0.93 \end{bmatrix}$$

Similarly for K_1 and V_1:

$$K_1 = K \times W_K^T = \begin{bmatrix} 0.3 & 0.8 & 0.5 \\ 0.1 & 0.2 & 0.4 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0.1 \\ 0.1 & 0.7 \\ 0.3 & 0.2 \end{bmatrix} = \begin{bmatrix} 0.44 & 0.79 \\ 0.26 & 0.21 \end{bmatrix}$$

$$V_1 = V \times W_V^T = \begin{bmatrix} 0.9 & 1.2 & 0.6 \\ 0.5 & 0.4 & 0.9 \end{bmatrix} \times \begin{bmatrix} 0.2 & 0.6 \\ 0.6 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} = \begin{bmatrix} 0.84 & 1.14 \\ 0.75 & 0.95 \end{bmatrix}$$

Repeat this process for the second head (using different projection weights), and you get corresponding values for Q_2, K_2, and V_2.

Step 2: Scaled Dot-Product Attention

For the first head, compute attention scores using Q_1 and K_1:

1. Compute the dot-product of Q_1 and K_1^T:

$$Q_1 K_1^T = \begin{bmatrix} 0.27 & 0.27 \\ 0.36 & 0.93 \end{bmatrix} \times \begin{bmatrix} 0.44 & 0.26 \\ 0.79 & 0.21 \end{bmatrix} = \begin{bmatrix} 0.32 & 0.13 \\ 0.73 & 0.22 \end{bmatrix}$$

2. Scale by $\sqrt{d_k}$ (here, $d_k = 2$):

$$\frac{Q_1 K_1^T}{\sqrt{2}} = \frac{1}{1.414} \times \begin{bmatrix} 0.32 & 0.13 \\ 0.73 & 0.22 \end{bmatrix} = \begin{bmatrix} 0.226 & 0.092 \\ 0.516 & 0.156 \end{bmatrix}$$

3. Apply softmax to get attention weights:

$$\text{softmax}\left(\begin{bmatrix} 0.226 & 0.092 \\ 0.516 & 0.156 \end{bmatrix}\right) = \begin{bmatrix} 0.533 & 0.467 \\ 0.588 & 0.412 \end{bmatrix}$$

4. Multiply by V_1 to get the attention output:

$$\text{Attention Output} = \text{Attention Weights} \times V_1 = \begin{bmatrix} 0.533 & 0.467 \\ 0.588 & 0.412 \end{bmatrix} \times \begin{bmatrix} 0.84 & 1.14 \\ 0.75 & 0.95 \end{bmatrix} = \begin{bmatrix} 0.80 & 1.04 \\ 0.79 & 1.09 \end{bmatrix}$$

Repeat this process for the second head.

Step 3: Concatenate the outputs from both heads

Assume after computing both heads, we get these attention outputs:

- Head 1 Output: $\begin{bmatrix} 0.80 & 1.04 \\ 0.79 & 1.09 \end{bmatrix}$
- Head 2 Output: $\begin{bmatrix} 0.72 & 1.01 \\ 0.65 & 1.12 \end{bmatrix}$

Concatenate these outputs:

$$\text{Concatenated Output} = \begin{bmatrix} 0.80 & 1.04 & 0.72 & 1.01 \\ 0.79 & 1.09 & 0.65 & 1.12 \end{bmatrix}$$

Step 4: Apply linear projection using W_O

Finally, apply a projection to the concatenated output using a weight matrix W_O (randomly initialized):

Let $W_O = \begin{bmatrix} 0.3 & 0.7 \\ 0.6 & 0.2 \\ 0.4 & 0.5 \\ 0.9 & 0.8 \end{bmatrix}$.

Project the concatenated output:

$$\text{Final Output} = \text{Concatenated Output} \times W_O = \begin{bmatrix} 0.80 & 1.04 & 0.72 & 1.01 \\ 0.79 & 1.09 & 0.65 & 1.12 \end{bmatrix} \times \begin{bmatrix} 0.3 & 0.7 \\ 0.6 & 0.2 \\ 0.4 & 0.5 \\ 0.9 & 0.8 \end{bmatrix} = \begin{bmatrix} 2.008 & 2.032 \\ 2.011 & 2.118 \end{bmatrix}$$

Thus, the final output of the multi-head cross-attention layer is:

$$\text{Final Output} = \begin{bmatrix} 2.008 & 2.032 \\ 2.011 & 2.118 \end{bmatrix}$$

d. Masked Multi-Head Attention (Causal)

What?

- In the decoder, this variant ensures that the model only attends to previous positions in the sequence when generating the next token.

Why?

- This prevents "cheating" by ensuring that future information is not accessible during training.

How?

- A mask is applied to the attention weights, setting the weights corresponding to future tokens to $-\infty$ before applying softmax, ensuring those tokens receive zero attention.

Step-by-Step Example

Assuming we have:

- Sequence length = 4 (tokens).
- Embedding dimension $d_{model} = 4$.

Initialize Queries, Keys, and Values

Let’s define:

- **Decoder Queries** Q (4 tokens):

$$Q = \begin{bmatrix} 0.5 & 0.1 & 0.2 & 0.3 \\ 0.4 & 0.3 & 0.7 & 0.8 \\ 0.6 & 0.8 & 0.1 & 0.9 \\ 0.2 & 0.4 & 0.6 & 0.5 \end{bmatrix} \quad \text{(shape: } (4, 4))$$

- In self-attention, K and V are the same as Q:

$$K = V = Q$$

Linear Projections for Q, K, and V

Assuming our projection weights for the first head are as follows:

- $W_Q = W_K = W_V = \begin{bmatrix} 0.1 & 0.3 & 0.5 & 0.2 \\ 0.2 & 0.4 & 0.6 & 0.1 \\ 0.7 & 0.1 & 0.8 & 0.3 \\ 0.6 & 0.2 & 0.4 & 0.5 \end{bmatrix}$

Compute the projections:

$$Q' = Q \times W_Q^T$$

(Similar calculations will be done for K and V).

Assuming the calculations give:

$$Q' = K' = V' \approx \begin{bmatrix} 0.43 & 0.41 & 0.50 & 0.35 \\ 0.73 & 0.85 & 0.64 & 0.58 \\ 0.60 & 0.73 & 0.49 & 0.46 \\ 0.52 & 0.70 & 0.36 & 0.47 \end{bmatrix}$$

Compute Scaled Dot-Product Attention

Compute the dot-product of Q' and K'^T:

$$Q'K'^T = \begin{bmatrix} 0.43 & 0.41 & 0.50 & 0.35 \\ 0.73 & 0.85 & 0.64 & 0.58 \\ 0.60 & 0.73 & 0.49 & 0.46 \\ 0.52 & 0.70 & 0.36 & 0.47 \end{bmatrix} \times \begin{bmatrix} 0.43 & 0.73 & 0.60 & 0.52 \\ 0.41 & 0.85 & 0.73 & 0.70 \\ 0.50 & 0.64 & 0.49 & 0.36 \\ 0.35 & 0.58 & 0.46 & 0.47 \end{bmatrix}$$

Scale by $\sqrt{d_k}$ (where $d_k = 4$):

$$\text{Scaled Scores} = \frac{Q'K'^T}{\sqrt{4}} = \frac{1}{2} \begin{bmatrix} 0.9 & 0.4 & 0.5 & 0.3 \\ 0.8 & 0.7 & 0.4 & 0.6 \\ 0.6 & 0.5 & 0.6 & 0.7 \\ 0.4 & 0.3 & 0.5 & 0.6 \end{bmatrix}$$

Create the Causal Mask:

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

Apply the Mask:
Add the mask to the scaled scores:

$$\text{Masked Scores} = \begin{bmatrix} 0.45 & -\infty & -\infty & -\infty \\ 0.6 & 0.5 & -\infty & -\infty \\ 0.4 & 0.3 & 0.6 & -\infty \\ 0.3 & 0.4 & 0.5 & 0.6 \end{bmatrix}$$

Apply Softmax:
Apply the softmax function to each row to get the attention weights:

For example, for the first row:

$$\text{Softmax}(\text{Masked Scores}[0]) = \text{softmax}\begin{bmatrix} 0.45 & -\infty & -\infty & -\infty \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

Continuing this for each row yields:

$$\text{Attention Weights} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{e^{0.6}}{e^{0.6}+e^{0.5}} & \frac{e^{0.5}}{e^{0.6}+e^{0.5}} & 0 & 0 \\ \frac{e^{0.4}}{e^{0.4}+e^{0.3}+e^{0.6}} & \frac{e^{0.3}}{e^{0.4}+e^{0.3}+e^{0.6}} & \frac{e^{0.6}}{e^{0.4}+e^{0.3}+e^{0.6}} & 0 \\ \frac{e^{0.3}}{e^{0.3}+e^{0.4}+e^{0.5}+e^{0.6}} & \frac{e^{0.4}}{e^{0.3}+e^{0.4}+e^{0.5}+e^{0.6}} & \frac{e^{0.5}}{e^{0.3}+e^{0.4}+e^{0.5}+e^{0.6}} & \frac{e^{0.6}}{e^{0.3}+e^{0.4}+e^{0.5}+e^{0.6}} \end{bmatrix}$$

For simplicity, let’s assume we calculate these softmax values and get:

$$\text{Attention Weights} \approx \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.67 & 0.33 & 0 & 0 \\ 0.36 & 0.27 & 0.36 & 0 \\ 0.25 & 0.25 & 0.25 & 0.25 \end{bmatrix}$$

Multiply by V' to get the attention output:**

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

Calculating for each token:

For token 0:

$$\text{Attention Output}[0] = 1 \times V'[0] + 0 \times V'[1] + 0 \times V'[2] + 0 \times V'[3] = V'[0] = \begin{bmatrix} 0.43 \\ 0.41 \\ 0.50 \\ 0.35 \end{bmatrix}$$

For token 1:

$$\text{Attention Output}[1] = 0.67 \times V'[0] + 0.33 \times V'[1] = 0.67 \begin{bmatrix} 0.43 \\ 0.41 \\ 0.50 \\ 0.35 \end{bmatrix} + 0.33 \begin{bmatrix} 0.73 \\ 0.85 \\ 0.64 \\ 0.58 \end{bmatrix}$$

Performing this calculation yields:

$$\text{Attention Output}[1] \approx \begin{bmatrix} 0.56 \\ 0.56 \\ 0.53 \\ 0.47 \end{bmatrix}$$

Continue this for tokens 2 and 3.

Final output after processing:

$$\text{Attention Output} = \begin{bmatrix} V'[0] \\ V'[1] \\ V'[2] \\ V'[3] \end{bmatrix}$$

Assuming the calculations yield:

$$\text{Attention Output} \approx \begin{bmatrix} 0.43 & 0.41 & 0.50 & 0.35 \\ 0.56 & 0.56 & 0.53 & 0.47 \\ 0.45 & 0.38 & 0.53 & 0.49 \\ 0.40 & 0.41 & 0.44 & 0.46 \end{bmatrix}$$

Concatenate Outputs for Multiple Heads

If we had h heads, we would concatenate the outputs of each head and then linearly project the result.

Let's say our outputs from 2 heads are:

$$\text{Head 1 Output} = \begin{bmatrix} 0.43 & 0.41 & 0.50 & 0.35 \\ \dots \end{bmatrix}$$

$$\text{Head 2 Output} = \begin{bmatrix} 0.34 & 0.56 & 0.57 & 0.45 \\ \dots \end{bmatrix}$$

Concatenating gives us:

$$\text{Concat Output} = \begin{bmatrix} 0.43 & 0.41 & 0.50 & 0.35 & 0.34 & 0.56 & 0.57 & 0.45 \\ \dots \end{bmatrix}$$

Final Linear Projection

Apply a linear transformation to the concatenated output:

$$\text{Final Output} = \text{Concat Output} \times W_O$$

where W_O is the output projection weight matrix.

5. FeedForward Block

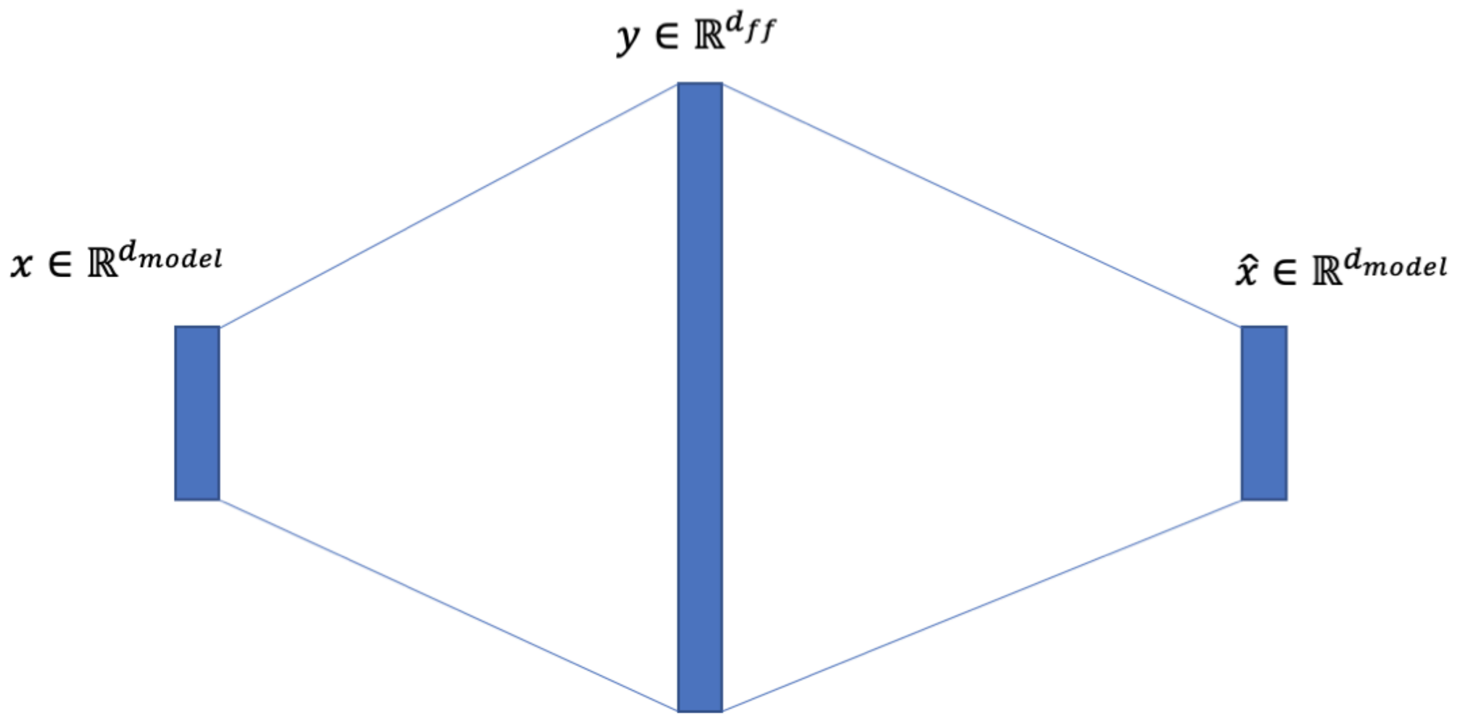
What?

- A two-layer fully connected neural network applied independently to each position in the sequence.

Why?

- It helps model non-linear transformations and enhances the model’s expressive power.

How?


$y \in \mathbb{R}^{d_{ff}}$
$x \in \mathbb{R}^{d_{model}}$
$\hat{x} \in \mathbb{R}^{d_{model}}$

The **FeedForward block** consists of two linear transformations with a ReLU activation in between.

For simplicity, assume input to the FeedForward block is a 2x2 matrix:

Input	Weight 1 W_1	Weight 2 W_2
[0.5, -0.3]	[0.2, 0.4]	[0.6, -0.2]
[-0.2, 0.8]	[0.1, 0.7]	[-0.4, 0.9]

Step-by-Step Process

Step 1: First linear transformation

Apply the first linear layer. Assume:

- $W_1 = \begin{bmatrix} 0.5 & 0.2 \\ 0.4 & -0.1 \\ 0.3 & 0.7 \end{bmatrix}$
- Bias $b_1 = \begin{bmatrix} 0.1, 0.1 \end{bmatrix}$

Multiply the input by W_1 and add b_1:

$$\text{Output}_1 = \text{Input} \times W_1 + b_1 = \begin{bmatrix} 0.5 & -0.3 & 0.1 \\ -0.2 & 0.8 & 0.3 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0.2 \\ 0.4 & -0.1 \\ 0.3 & 0.7 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.49 & 0.24 \\ 0.32 & 0.41 \end{bmatrix}$$

Step 2: Apply ReLU activation

ReLU sets any negative values to 0:

$$\text{Output after ReLU} = \text{ReLU}(\text{Output}_1) = \begin{bmatrix} 0.49 & 0.24 \\ 0.32 & 0.41 \end{bmatrix}$$

Step 3: Second linear transformation

Apply the second linear transformation with $W_2 = \begin{bmatrix} 0.1 & -0.3 \\ 0.6 & 0.2 \end{bmatrix}$ and bias $b_2 = [0.05, 0.05]$:

$$\text{Final Output} = \text{ReLU output} \times W_2 + b_2 = \begin{bmatrix} 0.49 & 0.24 \\ 0.32 & 0.41 \end{bmatrix} \times \begin{bmatrix} 0.1 & -0.3 \\ 0.6 & 0.2 \end{bmatrix} + \begin{bmatrix} 0.05 & 0.05 \end{bmatrix} = \begin{bmatrix} 0.197 & 0.002 \\ 0.296 & 0.041 \end{bmatrix}$$

Thus, the final output of the FeedForward Network is:

$$\text{Final Output of FFN} = \begin{bmatrix} 0.197 & 0.002 \\ 0.296 & 0.041 \end{bmatrix}$$

6. Add & Layer Normalization (Residual Network)

What?

- Residual connections skip the sub-layers, ensuring gradient flow during training. Layer normalization stabilizes the learning process.

Why?

- The residual connections prevent vanishing gradients and improve convergence, while LayerNorm helps normalize activations.

How?

- The input to each sub-layer (e.g., Multi-Head Attention) is transformed as:

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

  - This "add" operation ensures the input flows through directly, while normalization keeps the activations stable.

Let's break down the **Add & Layer Normalization** process step by step with a mathematical example.

Add Residual Connection
The residual connection helps in training by adding the input of a layer back to its output.

Given:

- x = the input to a sub-layer (e.g., attention or feed-forward layer)
- $\text{sublayer}(x)$ = the output from that sub-layer (e.g., output from the attention mechanism or feed-forward block)

The residual connection is simply:

$$\text{Residual Output} = x + \text{sublayer}(x)$$

Layer Normalization
Layer Normalization ensures that the output is normalized, with a mean of 0 and a variance of 1, applied to each individual training example (not across the entire batch, which is done in BatchNorm).

The formula for Layer Normalization is:

$$\text{LayerNorm}(z) = \frac{z-\mu}{\sigma} \cdot \gamma + \beta$$

Where:

- z = the input to the LayerNorm (i.e., the result of the residual connection)
- μ = mean of the input
- σ = standard deviation of the input
- γ = scaling factor (learnable parameter)
- β = bias term (learnable parameter)

Step by Step Example

Let's go through a detailed step-by-step calculation.

Given:

- Input to sub-layer: $x = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$
- Sub-layer output: $\text{sublayer}(x) = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$

Add Residual Connection

Perform element-wise addition of x and $\text{sublayer}(x)$:

$$z = x + \text{sublayer}(x) = \begin{bmatrix} 2 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 3 \\ 7 \end{bmatrix}$$

Now, $z = \begin{bmatrix} 3 \\ 7 \end{bmatrix}$ is the input to the LayerNorm function.

Layer Normalization

Calculate the mean (μ):

The mean of z is:

$$\mu = \frac{3+7}{2} = 5$$

Calculate the variance (σ^2):

The variance is the average of the squared differences from the mean:

$$\sigma^2 = \frac{(3-5)^2 + (7-5)^2}{2} = \frac{(-2)^2 + (2)^2}{2} = \frac{4+4}{2} = 4$$

Therefore, the standard deviation (σ) is:

$$\sigma = \sqrt{4} = 2$$

Normalize the input:

Subtract the mean from each element and divide by the standard deviation:

$$\hat{z} = \frac{z-\mu}{\sigma} = \frac{\begin{bmatrix}3\\7\end{bmatrix} - 5}{2} = \frac{\begin{bmatrix}-2\\2\end{bmatrix}}{2} = \begin{bmatrix}-1\\1\end{bmatrix}$$

Scale and shift (with γ and β):
Apply the learnable scaling (γ) and shift (β) terms. Let's assume $\gamma = 1$ and $\beta = 0$ (i.e., no scaling or shifting):

$$\text{LayerNorm}(z) = \gamma \cdot \hat{z} + \beta = 1 \cdot \begin{bmatrix}-1\\1\end{bmatrix} + 0 = \begin{bmatrix}-1\\1\end{bmatrix}$$

Final Output:
The final normalized output is:

$$\text{LayerNorm}(z) = \begin{bmatrix}-1\\1\end{bmatrix}$$

Training Time Token Flow

How Training Happens:

1. The input tokens are embedded and encoded.
2. Each encoder layer performs self-attention and passes the representation to the next layer.
3. The decoder generates tokens sequentially, attending to both past outputs and encoder outputs.
4. The training objective minimizes the difference between the predicted tokens and the true output using cross-entropy loss.

Inference Process

1. During inference, the decoder generates tokens one by one.
2. Multi-Head Cross-Attention focuses on relevant parts of the encoder output at each decoding step.
3. Tokens are predicted using softmax and fed back into the decoder to generate the next token.

Decoding Strategies

What is a Decoding Strategy?

A **decoding strategy** refers to the method used by a sequence generation model (like a language model) to generate output sequences from predicted probability distributions over tokens at each time step.

Types of Decoding Strategies (One-liner for each):

1. **Greedy Decoding**: Selects the most probable token at each step without considering future predictions.
2. **Beam Search**: Explores multiple possible sequences by maintaining the top k most likely sequences at each step, balancing exploration and exploitation.

3. **Top-k Sampling**: Samples the next token from the top k most probable tokens instead of selecting the highest probability one.
4. **Top-p (Nucleus) Sampling**: Samples from the smallest group of tokens whose cumulative probability exceeds a threshold p.

Each decoding strategy balances efficiency, exploration, and output quality depending on the task and model behavior.

Greedy Decoding

Greedy decoding is a simple and intuitive sequence generation technique used in machine learning models, especially in **sequence-to-sequence (Seq2Seq)** tasks like **language translation**, **text generation**, and **summarization**. The idea behind greedy decoding is to generate a sequence by **selecting the most probable token** at each time step until the sequence is complete.

What is Greedy Decoding?

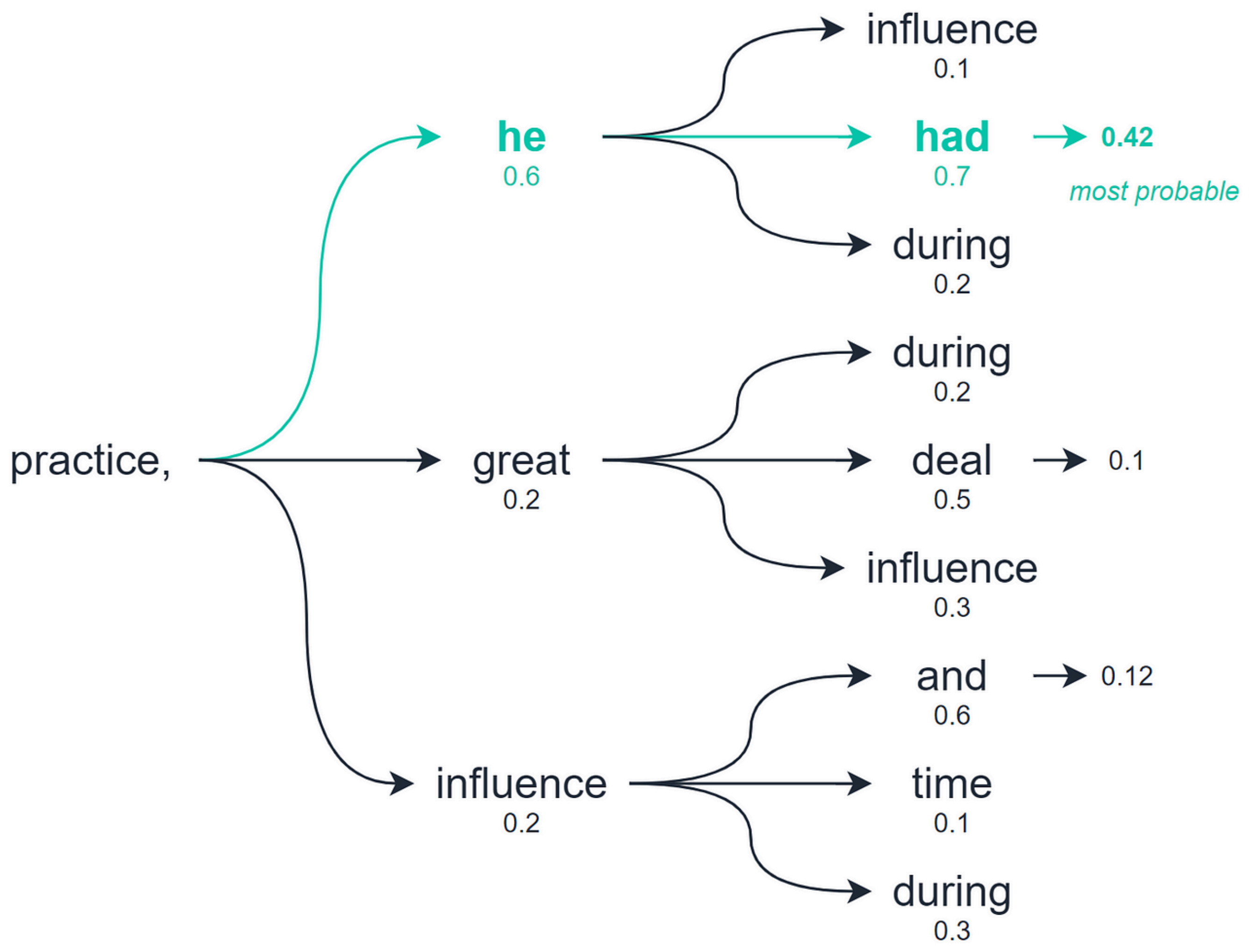
- **Greedy decoding** selects the token with the highest probability at each time step without considering the long-term consequences of that choice.
- It doesn't look ahead to evaluate the overall sequence; it simply takes the highest probability option at each step and continues to the next.
- It's a **deterministic** approach, meaning that for a given input, greedy decoding will always generate the same output.

Why Greedy Decoding?

- **Speed**: Greedy decoding is computationally efficient because it makes a single pass over the sequence.
- **Simplicity**: There's no need to explore multiple possible paths, making it straightforward to implement.
- However, greedy decoding may not always find the **globally optimal sequence** because it ignores other potentially promising tokens in favor of the immediate best choice at each time step. This can lead to suboptimal results, especially in cases where later tokens could improve the sequence.

How Greedy Decoding Works:

Let's go through the process step by step using an example.



Step-by-Step Explanation with Example

Consider a language generation task where the model is trying to predict a sentence given a starting token.

Assumptions:

- You have a trained language model that, at each step, predicts a probability distribution over the vocabulary.
- For simplicity, assume the vocabulary contains four words: `the` , `cat` , `sat` , `on` , and an `<END>` token.

Let's say the model is tasked with generating the sentence **"The cat sat on"**.

Step 1: Initialization

- You start with an initial input token, typically a special start-of-sequence token, denoted as `<START>` .
- At time step $t = 0$, the model predicts the probability distribution of the first token.

Model's Output:

$$P(\text{Token}) = [P(\text{the}), P(\text{cat}), P(\text{sat}), P(\text{on}), P(\text{<END>})]$$

Let's assume the probabilities for the first token are:

$$P(\text{Token}) = [0.8, 0.05, 0.05, 0.05, 0.05]$$

Step 2: Choose the Most Likely Token

In greedy decoding, you select the token with the **highest probability** at each step. In this case, the token `"the"` has the highest probability:

$$P(\text{the}) = 0.8$$

So the model chooses **"the"** as the first word in the generated sequence.

Step 3: Update the Input

- The token **"the"** is now fed back as input to the model to predict the next token.
- The model generates a new probability distribution over the vocabulary for the second token.

Let's assume at time step $t = 1$, the output probabilities are:

$$P(\text{Token}) = [0.1, 0.7, 0.1, 0.05, 0.05]$$

Here, the token `"cat"` has the highest probability:

$$P(\text{cat}) = 0.7$$

Thus, the model selects **"cat"** as the next token.

Step 4: Repeat the Process

- You repeat this process for each subsequent time step.
- The token `"cat"` is fed back as input, and the model predicts the next token based on the updated context.

Let's assume the following probability distributions for the next steps:

At $t = 2$:

$$P(\text{Token}) = [0.05, 0.05, 0.85, 0.03, 0.02] \quad (\text{Most probable: "sat"})$$

At $t = 3$:

$$P(\text{Token}) = [0.02, 0.03, 0.1, 0.85, 0.01] \quad (\text{Most probable: "on"})$$

So, the model chooses the tokens `"sat"` and `"on"` in steps 3 and 4.

Step 5: End the Sequence

At each time step, the model also predicts a probability for the special `<END>` token, which signals the end of the sequence.

At time step $t = 4$:

$$P(\text{Token}) = [0.01, 0.01, 0.01, 0.02, 0.95] \quad (\text{Most probable: "<END>"})$$

Since `<END>` has the highest probability, the model stops generating tokens.

Final Sequence:
The final sequence generated by greedy decoding is:

$$\text{"The cat sat on"}$$

Mathematical Representation of Greedy Decoding

At each time step t, greedy decoding selects the token y_t with the maximum probability $P(y_t \mid y_{1:t-1})$, where $y_{1:t-1}$ is the sequence generated so far:

$$y_t = \arg\max P(y_t \mid y_{1:t-1}, \text{input})$$

This process continues until the `<END>` token is generated, or a predefined maximum sequence length is reached.

Advantages of Greedy Decoding

- **Efficiency**: Since it only considers the highest-probability token at each step, greedy decoding is computationally efficient.
- **Deterministic Output**: The same input will always generate the same output sequence.

Limitations of Greedy Decoding

- **Suboptimal Sequences**: By choosing the most probable token at each step, greedy decoding ignores tokens that might lead to a better overall sequence if chosen at earlier steps. It's **greedy** in the sense that it only focuses on short-term rewards (highest probability at each step) without considering long-term consequences.
  For example, in language translation, the most probable word in a local context might not result in the best translation when considered as part of the entire sentence.
- **No Exploration**: Greedy decoding doesn't explore alternative tokens or paths, unlike more sophisticated decoding strategies like **Beam Search**, which keeps track of multiple possible sequences.

Comparison with Beam Search

- In contrast to greedy decoding, **beam search** keeps track of multiple hypotheses (possible sequences) at each time step. It selects the top k sequences (where k is the beam width) rather than just one, making it less likely to get stuck in a suboptimal path.
- Beam search can yield better overall sequences but at a higher computational cost.

Greedy Decoding Example: Summary

Let's summarize the step-by-step process of greedy decoding with a table, using the same example as before:

Time Step	Model's Predicted Probability Distribution	Selected Token
$t = 0$	$[0.8, 0.05, 0.05, 0.05, 0.05]$	"the"
$t = 1$	$[0.1, 0.7, 0.1, 0.05, 0.05]$	"cat"
$t = 2$	$[0.05, 0.05, 0.85, 0.03, 0.02]$	"sat"
$t = 3$	$[0.02, 0.03, 0.1, 0.85, 0.01]$	"on"
$t = 4$	$[0.01, 0.01, 0.01, 0.02, 0.95]$	`<END>`

Summary Table

Component	Input	Operation	Output
Token Embedding	"I love NLP"	Token to Dense Vector	[0.21, -0.34, 0.56, ...]
Position Encoding	Token Embedding + Pos	Sine/Cosine Positional Info Added	[0.56, -0.33, 0.45, ...]
Multi-Head Attention	Q, K, V	Self-Attention & Weighted Averaging	Attention-weighted matrix
FeedForward Block	Attention Output	Linear -> ReLU -> Linear	Transformed features
Add & LayerNorm	Input + SubLayer Output	Residual Connection + Layer Normalization	Normalized features