Transformer Architecture: A Step-by-Step Breakdown

This document provides a detailed breakdown of the Transformer architecture, a revolutionary neural network model that has achieved state-of-the-art results in various natural language processing tasks. We will explore each layer and function within the Transformer, explaining its purpose and how it contributes to the overall model.

1. Tokenization

The first step in processing text data for a Transformer model is tokenization. This involves breaking down the input text into smaller units called tokens. These tokens can be words, sub-words, or even characters, depending on the specific tokenization method used.

- **Purpose:** To convert raw text into a numerical representation that the model can understand.
- **Methods:** Common tokenization methods include:
    - **Word-based tokenization:** Splits the text into individual words. Simple but can lead to large vocabularies.
    - **Subword tokenization (e.g., Byte Pair Encoding (BPE), WordPiece):** Splits words into smaller, more frequent subwords. Balances vocabulary size and handling of rare words.
    - **Character-based tokenization:** Splits the text into individual characters. Smallest vocabulary but can be less semantically meaningful.
- **Example:**
    - Input text: "The quick brown fox jumps over the lazy dog."
    - Word-based tokens: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]
    - Subword tokens (hypothetical): ["The", "quick", "brow", "n", "fox", "jump", "s", "over", "the", "lazy", "dog", "."]

2. Embedding Layer

Once the text is tokenized, each token needs to be converted into a vector representation. This is the role of the embedding layer.

- **Purpose:** To map each token to a high-dimensional vector space, where semantically similar tokens are located closer to each other.
- **Mechanism:** The embedding layer is a lookup table that maps each token index to a corresponding vector. These vectors are learned during the training process.
- **Output:** A matrix where each row represents the embedding vector for a corresponding token in the input sequence.
- **Example:** If the token "quick" has an index of 5 in the vocabulary, the embedding layer will output the vector associated with index 5.

3. Positional Encoding

Transformers, unlike recurrent neural networks (RNNs), do not inherently process sequential data in order. Therefore, positional encoding is crucial to provide the model with information about the position of each token in the sequence.

- **Purpose:** To inject information about the position of each token into its embedding vector.
- **Mechanism:** Positional encodings are added to the token embeddings. These encodings are typically generated using sine and cosine functions with different frequencies.
- **Formula:**
    - PE(pos, 2i) = sin(pos / (10000^{2i/d_{model}}))
    - PE(pos, 2i+1) = cos(pos / (10000^{2i/d_{model}}))
    - where:
        - pos is the position of the token in the sequence.
        - i is the dimension index.
        - d_{model} is the dimensionality of the embedding vectors.
- **Output:** A matrix of the same shape as the embedding matrix, where each row represents the positional encoding for the corresponding position in the sequence. This matrix is added element-wise to the embedding matrix.

4. Transformer Block (repeated)

The core of the Transformer architecture is the Transformer block, which is repeated multiple times. Each block consists of several sub-layers: Multi-Head Self-Attention, Add & Layer Normalization, and a Feedforward Neural Network.

4A. Multi-Head Self-Attention

This is the most important component of the Transformer. It allows the model to attend to different parts of the input sequence when processing each token.

- **Purpose:** To capture relationships between different tokens in the input sequence.
- **Mechanism:**
    1. **Linear Transformations:** The input embedding vectors are linearly transformed into three different sets of vectors: Queries (Q), Keys (K), and Values (V).
    2. **Scaled Dot-Product Attention:** The attention weights are calculated by taking the dot product of the Queries and Keys, scaling the result by the square root of the dimension of the Keys (to prevent the dot products from becoming too large), and then applying a softmax function.
    3. **Weighted Sum:** The attention weights are then used to compute a weighted sum of the Value vectors.
    4. **Multiple Heads:** This process is repeated multiple times in parallel, each with its own set of linear transformations. The outputs of all the heads are then concatenated and linearly transformed to produce the final output.
- **Formula:**
    - Attention(Q, K, V) = softmax((Q K^T) / $\sqrt{}$d_k) V
    - where:
        - Q is the matrix of Queries.
        - K is the matrix of Keys.
        - V is the matrix of Values.
        - d_k is the dimension of the Keys.
- **Output:** A matrix representing the context-aware representation of each token in the input sequence.

4B. Add & Layer Normalization

After the Multi-Head Self-Attention layer, the output is added to the original input (residual connection) and then normalized using Layer Normalization.

- **Purpose:**
    - **Residual Connection:** To allow the gradient to flow more easily through the network, preventing the vanishing gradient problem.
    - **Layer Normalization:** To stabilize the training process and improve generalization.
- **Mechanism:**
    - **Add:** The output of the Multi-Head Self-Attention layer is added to the original input embedding vectors.
    - **Layer Normalization:** Layer Normalization normalizes the activations of each layer across all the features for each training example.
- **Output:** A normalized matrix representing the context-aware representation of each token in the input sequence, with the original input information preserved.

4C. Feedforward Neural Network

This layer applies a feedforward neural network to each token independently.

- **Purpose:** To further process the context-aware representation of each token.
- **Mechanism:** The feedforward network typically consists of two linear transformations with a ReLU activation function in between.
- **Output:** A matrix representing the further processed context-aware representation of each token in the input sequence.

4D. Add & Layer Normalization

Similar to step 4B, the output of the Feedforward Neural Network is added to the input of the Feedforward Neural Network (residual connection) and then normalized using Layer Normalization.

- **Purpose:**
    - **Residual Connection:** To allow the gradient to flow more easily through the network.
    - **Layer Normalization:** To stabilize the training process and improve generalization.
- **Mechanism:**
    - **Add:** The output of the Feedforward Neural Network is added to the input of the Feedforward Neural Network.
    - **Layer Normalization:** Layer Normalization normalizes the activations of each layer across all the features for each training example.
- **Output:** A normalized matrix representing the final processed representation of each token in the input sequence after the Transformer block.

5. Output/Classification Layer (Linear + Softmax)

After passing through multiple Transformer blocks, the final output is fed into an output layer, which typically consists of a linear layer followed by a softmax function.

- **Purpose:** To map the final representation of each token to a probability distribution over the possible output classes.
- **Mechanism:**

        - **Linear Layer:** The linear layer transforms the final representation of each token into a vector of logits, where each logit represents the score for a particular output class.
        - **Softmax Function:** The softmax function converts the logits into a probability distribution over the output classes.
    - **Output:** A probability distribution over the output classes for each token in the input sequence. This distribution can be used for tasks such as text classification, machine translation, and text generation.

In summary, the Transformer architecture leverages self-attention mechanisms and residual connections to effectively capture long-range dependencies in sequential data. Its modular design, with repeated Transformer blocks, allows for scalability and adaptability to various NLP tasks. The combination of tokenization, embedding, positional encoding, self-attention, and feedforward networks enables the Transformer to achieve state-of-the-art performance in a wide range of applications.