Transformer Interview Questions: Part 1

Sagar Sudhakara

December 2024

General Transformer Architecture

1. What are the key components of a Transformer model?

The key components of a Transformer model are:

- **Self-Attention Mechanism**: Captures relationships between tokens regardless of their positions in the sequence.
- **Multi-Head Attention**: Allows the model to focus on different parts of the input simultaneously.
- **Feedforward Neural Network (FFN)**: Processes each token independently after the attention mechanism.
- **Positional Encoding**: Introduces information about token positions in the sequence.
- **Layer Normalization**: Stabilizes training and accelerates convergence.
- **Residual Connections**: Helps gradient flow and mitigates vanishing gradient issues.
- **Softmax**: Normalizes attention scores to probabilities.

2. Why do Transformers replace recurrence with attention mechanisms?

Transformers replace recurrence with attention mechanisms for several reasons:

- **Parallelization**: Attention mechanisms process all tokens simultaneously, allowing for much faster computation compared to sequential recurrence.
- **Better Long-Range Dependency Handling**: Attention directly connects all tokens, regardless of their distance, whereas RNNs struggle with long-term dependencies due to vanishing gradients.
- **Scalability**: Attention scales more efficiently to large datasets and sequences by leveraging GPUs and TPUs effectively.

3. How is positional encoding implemented in Transformer models, and why is it necessary?

Implementation: Positional encodings are typically implemented as sinusoidal functions of token positions. For a position pos and dimension i:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

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

Here, d is the dimensionality of the model.
Necessity: Since attention mechanisms are permutation-invariant, positional encoding provides information about the sequence order, which is critical for tasks involving ordered data.

4. Can you explain the role of the encoder and decoder in the original Transformer architecture?

Encoder: Processes the input sequence and generates a sequence of continuous representations. It consists of:

- Multiple layers of self-attention and feedforward networks.
- Focuses solely on understanding the input data.

Decoder: Generates the output sequence while attending to the encoder's representations. It consists of:

- Self-attention to capture relationships in the target sequence.
- Encoder-decoder attention to align the target output with the input.
- Feedforward layers to refine predictions.

5. What are the advantages of Transformers over RNNs and CNNs for NLP tasks?

- **Parallelization**: Transformers process all tokens simultaneously, unlike RNNs, which are sequential.
- **Long-Range Dependencies**: Attention allows better modeling of distant relationships compared to RNNs and CNNs.
- **Scalability**: Transformers scale efficiently with data and model size.
- **Expressiveness**: Multi-head attention captures diverse aspects of token relationships.
- **Robustness**: No vanishing gradient issues that plague RNNs.

6. How do Transformers handle long-range dependencies?

Transformers use the **self-attention mechanism**, which assigns weights to all tokens in the input sequence based on their relevance to the current token. This mechanism directly connects distant tokens, effectively modeling long-range dependencies without relying on iterative steps.

7. What is the role of feedforward networks in the Transformer block?

Feedforward networks (FFNs) apply a non-linear transformation to the output of the attention mechanism for each token independently. Their role includes:

- **Refinement**: Adds additional expressiveness and complexity to the model.
- **Dimensionality Transformation**: Helps the model handle high-dimensional inputs effectively.
- **Token-Wise Processing**: Processes each token independently, complementing the relational focus of the attention mechanism.

Self-Attention Mechanism

1. What is self-attention, and how does it differ from traditional attention mechanisms?

Self-Attention: Computes the attention score of each token in a sequence with respect to all other tokens, including itself. This allows the model to understand relationships between all tokens in a single sequence.
Traditional Attention: Computes attention scores between tokens in one sequence (e.g., the target) and another sequence (e.g., the source). It is typically used for sequence-to-sequence tasks like machine translation.
Key Difference:

- **Scope**: Self-attention is intra-sequence (same sequence), while traditional attention is inter-sequence (between different sequences).
- **Purpose**: Self-attention captures contextual relationships within a sequence, whereas traditional attention aligns input and output sequences.

2. How is the scaled dot-product attention calculated?

Scaled dot-product attention is computed as follows:

1. **Input**: Query (Q), Key (K), and Value (V) matrices.
2. **Dot-Product**: Compute attention scores by taking the dot product of Q and $K^\top$:

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

3. **Scaling**: Scale the scores by the square root of the key dimension (d_k) to stabilize gradients:

$$\text{Scaled Scores} = \frac{QK^\top}{\sqrt{d_k}}$$

4. **Softmax**: Normalize the scores across each row to form a probability distribution:

$$\text{Attention Weights} = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)$$

5. **Weighted Sum**: Multiply the attention weights by V:

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

3. Why is the scaling factor ($1/\sqrt{d_k}$) used in self-attention?

The scaling factor is used to:

- **Prevent Overly Large Dot-Products**: When the dimensionality (d_k) of the key and query vectors is large, their dot-products can grow excessively, leading to very sharp gradients.
- **Stabilize Softmax**: Scaling the dot-product ensures the inputs to the softmax function are within a reasonable range, leading to smoother and more numerically stable probabilities.

4. What are the steps involved in computing multi-head attention?

1. **Linear Transformations**: Transform the input sequence into multiple Q, K, and V matrices using learned weight matrices. Each head uses a different set of weights.
2. **Scaled Dot-Product Attention**: Compute attention scores for each head independently.
3. **Concatenate Outputs**: Combine the outputs of all heads into a single matrix.
4. **Final Linear Transformation**: Apply a learned linear transformation to the concatenated output to produce the final multi-head attention output.

5. How does self-attention help in parallelizing computations in Transformers?

Self-attention allows for **parallelization** because:

- All tokens in the sequence are processed simultaneously rather than sequentially.
- Attention scores between all token pairs are computed in a single matrix operation using Q, K, and V.
- This eliminates the need for iterative computations, as required in RNNs, enabling efficient utilization of modern hardware like GPUs and TPUs.

6. Why do we use masking in self-attention layers, especially in decoder blocks?

Masking is used for two main reasons:

- **Prevent Information Leakage (Decoder)**:
  - In sequence generation tasks, future tokens should not influence the prediction of the current token.
  - A **causal mask** ensures that attention weights for future tokens are set to zero.
- **Handle Padding Tokens**:
  - Padding tokens are added to ensure consistent sequence lengths.
  - A **padding mask** ensures that padding tokens do not contribute to the attention computation by setting their weights to zero.

Mathematics of Query, Key, and Value

1. What are queries, keys, and values in the context of Transformers?

In the context of Transformers:

- **Query (Q)**: Represents the vector that is used to "ask" for information. It is derived from the current token's representation and is compared against all other tokens.
- **Key (K)**: Represents the vector that holds the information to be retrieved. It is associated with each token and is used to match the query.
- **Value (V)**: Represents the actual information content associated with each token. After the query is matched with the key, the value is returned as the relevant output.

These vectors are derived from the input sequence using learned projections (weights).

2. How is the attention score between two tokens computed?

The attention score between two tokens is computed using the **dot product** of the query and key vectors:

$$\text{Attention Score} = Q_i \cdot K_j^\top$$

Where:

- Q_i is the query vector for token i,
- K_j is the key vector for token j,
- $\cdot$ represents the dot product.

This score indicates how much token i should attend to token j.

3. Why is the softmax function applied to the attention scores?

The **softmax** function is applied to the attention scores to:

- **Normalize the Scores**: Softmax converts the raw attention scores into a probability distribution, ensuring that they sum to 1. This enables the model to focus on relevant tokens more strongly while suppressing less important ones.
- **Emphasize Relevant Tokens**: By applying softmax, the model can focus on the most relevant tokens (higher attention weights) and diminish the impact of irrelevant tokens (lower attention weights).

The softmax formula for attention scores is:

$$\text{Attention Weights} = \text{softmax}\left(\frac{Q_i \cdot K_j^\top}{\sqrt{d_k}}\right)$$

where $\sqrt{d_k}$ is the scaling factor, as discussed earlier.

4. Derive the self-attention mechanism mathematically using matrix operations.

Let's break down the self-attention mechanism:

1. **Input**: An input sequence of tokens $\{x_1, x_2, ..., x_n\}$, where each token is represented by a vector in $\mathbb{R}^d$.

2. **Linear Projections**: For each token, compute the query, key, and value vectors through learned weight matrices W_Q, W_K, W_V:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

   Where X is the matrix of input token embeddings, and W_Q, W_K, W_V are learned weight matrices.

3. **Scaled Dot-Product Attention**: Compute the attention scores between all tokens:

$$A = \frac{QK^\top}{\sqrt{d_k}}$$

   Where A is the matrix of attention scores.

4. **Apply Softmax**: Normalize the attention scores using softmax:

$$\text{Attention Weights} = \text{softmax}(A)$$

5. **Compute Output**: Multiply the attention weights by the value matrix V to obtain the final output:

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

The resulting matrix O is the output of the self-attention mechanism.

5. What happens if the query and key vectors are not aligned in dimensionality?

If the query and key vectors are not aligned in dimensionality (i.e., their dimensions d_q and d_k are not the same), the dot product operation $Q_i \cdot K_j^\top$ is not well-defined. Therefore, the model cannot compute the attention scores properly.
To address this issue:

- The dimensions of Q and K are usually made equal through learned projections (i.e., using weight matrices W_Q and W_K to map both to the same dimensionality).
- If $d_q \neq d_k$, we project the query and key vectors into a shared space of dimension d_{common} to perform the dot product.

6. How does multi-head attention improve the model's ability to capture diverse relationships in the input sequence?

Multi-head attention improves the model by allowing it to:

- **Capture Different Aspects of Relationships**: Each attention head learns different attention patterns, capturing various relationships in the input sequence. For example, one head might focus on syntactic dependencies, while another might focus on semantic relationships.
- **Increase Capacity**: By using multiple heads, the model has the capacity to attend to different parts of the input in parallel, improving its ability to capture diverse relationships.
- **Enhanced Representation**: Combining the outputs of multiple heads enables the model to aggregate a richer representation of the input sequence, which is more expressive than using a single attention mechanism.

7. What is the role of learned projections for queries, keys, and values?

The learned projections for queries, keys, and values serve to:

- **Transform Input Representations**: The input tokens are transformed into different spaces for queries, keys, and values. These projections are learned during training, allowing the model to adapt to the best representations for each type of vector.
- **Ensure Compatibility**: The projections ensure that queries and keys are in the same space, making the dot-product operation well-defined. Similarly, the value projections enable the model to generate meaningful output from the attention weights.
- **Increase Model Expressiveness**: By using different learned projections for each of Q, K, and V, the model can learn diverse ways to represent the same input tokens and their interactions, improving its ability to capture complex relationships in the data.