Contents

1 Problem: 1

Consider a Multi-head Transformer Algorithm as shown in the figure below. The algorithm consists of one encoder and one decoder (similar to the one discussed in the class). Assume 8 attention heads, embedding size of 256. Assume 16-bit floating point numbers for inputs, outputs and all intermediate operations. Also notice that some of the blocks have been skipped.

- Logical implementation of transformer is attached in zip file with name **"transformer.py"** .

1.1 Part a

- **Problem :** Implement the transformer algorithm.
- **Introduction**

  The Transformer model is a neural network architecture that employs self-attention mechanisms rather than traditional recurrence or convolution operations. This design allows for efficient parallelization and improved handling of long-range dependencies in sequences. Our single-layer implementation consists of an encoder and a decoder, which together transform an input sequence into an output sequence.

- **Encoder Block**

  The encoder processes the input sequence and generates a contextualized representation. The key components of the encoder include:

  Embedding and Positional Encoding : Each token in the sequence is represented as a d_{model}-dimensional vector. To preserve positional information, sinusoidal functions are added to the embeddings:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right), \tag{1}$$

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

  where pos represents the token's position and i indexes the embedding dimensions.

- **Multi-Head Self-Attention**

  Multi-head attention consists of multiple self-attention computations, each with its own learned weight matrices. The final output is obtained by concatenating the results of all attention heads and applying a linear transformation.

  The process can be described as follows:

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

  where each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

  where:

  - W_i^Q, W_i^K, W_i^V are learnable weight matrices for each attention head.
  - Q, K, V are the query, key, and value matrices from the input.
  - W^O is a weight matrix applied after concatenation.

  Self-attention is computed using the scaled dot-product formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{3}$$

  where $d_k = \frac{d_{model}}{h}$.

- **Residual Connection and Normalization**

  To stabilize training, a residual connection is applied followed by layer normalization:

$$Y = \text{LayerNorm}(X + \text{Attention}(Q, K, V)). \tag{4}$$

- **Feed-Forward Network (FFN)**

  A two-layer fully connected network with a ReLU activation is applied:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2. \tag{5}$$

  Again, a residual connection and normalization are applied:

$$Z = \text{LayerNorm}(Y + \text{FFN}(Y)). \tag{6}$$

- **Decoder Block**

  The decoder generates the output sequence using three main sub-layers:

- **Masked Multi-Head Self-Attention**

  This layer prevents positions from attending to future positions, ensuring autoregressive generation:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V, \tag{7}$$

  where M is a mask that enforces causality.

- **Cross-Attention (Encoder-Decoder Attention)**

  The decoder uses multi-head attention over the encoder's output. Queries come from the decoder, while keys and values originate from the encoder:

$$\text{CrossAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{8}$$

- **Feed-Forward Network**

  Similar to the encoder, the FFN is applied, followed by residual connections and normalization.