The Transformer: Let's code from scratch

Eng. Ahmed Métwalli - Eng. Alia Elhefny

1 Introduction

Transformers revolutionized the field of machine learning, particularly in Natural Language Processing (NLP), Vision, and Multimodal Learning. Unlike RNNs, Transformers process sequences in parallel using the **attention mechanism**, enabling scalability and efficiency.

This document provides:

- A breakdown of the Transformer architecture.
- Manual computation of self-attention.
- Python implementations of core components.

2 Core Components of the Transformer

The Transformer is composed of two main components:

1. **Encoder**: Processes input sequences into contextual representations.
2. **Decoder**: Generates output sequences based on encoder representations.

Each consists of:

1. Multi-Head Attention Layer.
2. Feed-Forward Neural Network (FFNN).
3. Residual Connections and Layer Normalization.

3 Python Implementation

We now define each component of the Transformer, starting with positional encoding.

3.1 Positional Encoding

Positional encoding provides the Transformer with information about token positions in the sequence.

```python
import torch
import math
import torch.nn as nn

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        """
        Args:
        - d_model: Dimension of the embedding vector (e.g., 512).
        - max_len: Maximum length of the input sequence.
        """
        super(PositionalEncoding, self).__init__()

        # Initialize a zero matrix for positional encodings
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))

        # Compute sin for even indices and cos for odd indices
        pe = torch.zeros(max_len, d_model)
        pe[:, 0::2] = torch.sin(position * div_term)  # Even indices
        pe[:, 1::2] = torch.cos(position * div_term)  # Odd indices

        # Store as a buffer (not a parameter)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        """
        Add positional encoding to input embeddings.

        Args:
        - x: Input embeddings of shape (batch_size, seq_len, d_model)

        Returns:
        - Encoded embeddings with positional information added.
        """
        return x + self.pe[:, :x.size(1), :]
```

Explanation:

- Even dimensions use sine; odd dimensions use cosine.
- $10000^{\frac{2i}{d_{\text{model}}}}$ ensures each dimension is scaled uniquely.

3.2 Scaled Dot-Product Attention

The scaled dot-product attention mechanism computes attention scores for tokens in a sequence.

```python
def scaled_dot_product_attention(q, k, v, mask=None):
    """
    Args:
    - q: Query matrix of shape (batch_size, num_heads, seq_len, d_k)
    - k: Key matrix of shape (batch_size, num_heads, seq_len, d_k)
    - v: Value matrix of shape (batch_size, num_heads, seq_len, d_v)
    - mask: Mask for padding (optional)

    Returns:
    - Weighted sum of values (attended output)
    - Attention weights
    """
    d_k = q.size(-1)  # Dimension of the keys
    scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)  # Compute QK^T / sqrt(d_k)

    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)  # Mask padding tokens

    attention_weights = torch.softmax(scores, dim=-1)  # Normalize scores
    output = torch.matmul(attention_weights, v)  # Weight the values by attention

    return output, attention_weights
```

Explanation:

- Computes similarity between tokens (QK^T).
- Scales scores by $\frac{1}{\sqrt{d_k}}$ to avoid large values.
- Applies softmax to normalize scores into probabilities.
- Returns weighted values and attention probabilities.

3.3 Multi-Head Attention

Multi-head attention allows the Transformer to focus on different parts of a sequence simultaneously.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        """
        Args:
        - d_model: Dimensionality of input embeddings.
        - num_heads: Number of attention heads.
        """
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_k = d_model // num_heads  # Dimension per head
        self.num_heads = num_heads

        # Linear layers for Q, K, V transformations
        self.q_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)

        # Final linear layer
        self.out_linear = nn.Linear(d_model, d_model)

    def forward(self, x):
        """
        Args:
        - x: Input of shape (batch_size, seq_len, d_model)

        Returns:
        - Attended output of shape (batch_size, seq_len, d_model)
        """
        batch_size, seq_len, _ = x.size()

        # Transform inputs into Q, K, V matrices
        q = self.q_linear(x).view(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)
        k = self.k_linear(x).view(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)
        v = self.v_linear(x).view(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)

        # Apply scaled dot-product attention
        attended_values, _ = scaled_dot_product_attention(q, k, v)

        # Concatenate attention heads and pass through final linear layer
        attended_values = attended_values.transpose(1, 2).
```

```
            contiguous().view(batch_size, seq_len, -1)
        output = self.out_linear(attended_values)

        return output
```

Explanation:

- Splits input embeddings into multiple heads.
- Each head focuses on different relationships in the sequence.
- Concatenates the results and projects them into the original space.