Detailed Explanation of Self-Attention Mechanism

1 Introduction

This document provides a comprehensive explanation of the self-attention mechanism implemented in the provided code. Self-attention is a key component of transformer models, allowing them to weigh the importance of different words in a sequence relative to a given word.

2 Key Mathematical Notation

- $X \in \mathbb{R}^{L \times d_{\text{model}}}$: Input sequence of length L and dimension d_{model}
- $W_Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$: Query weight matrix
- $W_K \in \mathbb{R}^{d_{\text{model}} \times d_k}$: Key weight matrix
- $W_V \in \mathbb{R}^{d_{\text{model}} \times d_v}$: Value weight matrix
- $Q \in \mathbb{R}^{L \times d_k}$: Query matrix
- $K \in \mathbb{R}^{L \times d_k}$: Key matrix
- $V \in \mathbb{R}^{L \times d_v}$: Value matrix

3 Computing Query, Key, and Value Matrices

The first function in the code, `compute_qkv`, calculates the query, key, and value matrices from the input sequence:

```python
def compute_qkv(X, W_q, W_k, W_v):
    """
    Compute query, key, and value matrices for self-attention.

    Args:
        X: Input tensor of shape (seq_len, d_model)
        W_q: Query weight matrix of shape (d_model, d_k)
        W_k: Key weight matrix of shape (d_model, d_k)
        W_v: Value weight matrix of shape (d_model, d_v)

```

```
    Returns:
        Q: Query matrix of shape (seq_len, d_k)
        K: Key matrix of shape (seq_len, d_k)
        V: Value matrix of shape (seq_len, d_v)
    """
    # Compute query, key, and value matrices by multiplying input with respective weights
    Q = np.dot(X, W_q)  # Shape: (seq_len, d_k)
    K = np.dot(X, W_k)  # Shape: (seq_len, d_k)
    V = np.dot(X, W_v)  # Shape: (seq_len, d_v)

    return Q, K, V
```

Listing 1: Compute Query

3.1 Mathematical Representation

The calculation of Q, K, and V matrices is performed as follows:

$$Q = X \cdot W_Q \tag{1}$$
$$K = X \cdot W_K \tag{2}$$
$$V = X \cdot W_V \tag{3}$$

3.2 Explanation

- This function creates three different representations of the input data:
  - **Query** (Q): Represents what the current token is "looking for"
  - **Key** (K): Represents what each token in the sequence "contains"
  - **Value** (V): Represents the actual content of each token
- The operations are simple matrix multiplications between the input tensor X and respective weight matrices
- Each row in the resulting matrices corresponds to a position in the input sequence
- This linear transformation projects the input data into different subspaces for the attention mechanism

4 Self-Attention Mechanism

The second function, `self_attention`, implements the core self-attention mechanism:

```
def self_attention(Q, K, V, mask=None, scale=True):
    """
    Compute self-attention mechanism.

```

```python
    Args:
        Q: Query matrix of shape (seq_len, d_k)
        K: Key matrix of shape (seq_len, d_k)
        V: Value matrix of shape (seq_len, d_v)
        mask: Optional mask to apply to attention scores
        scale: Whether to scale attention scores by sqrt(d_k)

    Returns:
        output: Self-attention output of shape (seq_len, d_v)
    """
    # Get dimensions
    seq_len, d_k = K.shape

    # Step 1: Compute attention scores by multiplying Q with K transpose
    # Shape: (seq_len, seq_len)
    attention_scores = np.dot(Q, K.T)

    # Step 2: Scale attention scores by sqrt(d_k) for stable gradients
    if scale:
        attention_scores = attention_scores / np.sqrt(d_k)

    # Step 3: Apply mask if provided (used in decoder for causal attention)
    if mask is not None:
        attention_scores = attention_scores + mask

    # Step 4: Apply softmax to get attention weights
    # Shape: (seq_len, seq_len)
    attention_weights = np.exp(attention_scores) / np.sum(np.exp(attention_scores), axis=1, keepdims=True)

    # Step 5: Compute weighted sum of values based on attention weights
    # Shape: (seq_len, d_v)
    output = np.dot(attention_weights, V)

    return output
```

Listing 2: Self-Attention Mechanism

Let's break down this function step by step:

4.1 Step 1: Computing Attention Scores

$$\text{Attention Scores} = Q \cdot K^T \tag{4}$$

Explanation:

- This step calculates how much each token should attend to every other token
- The dot product between a query vector and a key vector measures their similarity

- The result is a matrix of shape (seq_len, seq_len) where entry (i, j) indicates how much position i should attend to position j
- In code: `attention_scores = np.dot(Q, K.T)`

4.2 Step 2: Scaling

$$\text{Scaled Scores} = \frac{\text{Attention Scores}}{\sqrt{d_k}} \tag{5}$$

Explanation:

- Scaling prevents the dot products from growing too large in magnitude as the dimension d_k increases
- Large dot products would push the softmax function into regions with extremely small gradients
- Scaling by $\sqrt{d_k}$ helps maintain more stable gradients during training
- In code: `attention_scores = attention_scores / np.sqrt(d_k)`

4.3 Step 3: Masking (Optional)

$$\text{Masked Scores} = \text{Scaled Scores} + \text{Mask} \tag{6}$$

Explanation:

- Masking is used to prevent certain positions from attending to others
- In decoder self-attention, masking prevents a token from attending to future tokens (causal masking)
- The mask typically contains $-\infty$ values for positions that should be ignored, which become 0 after softmax
- In code: `attention_scores = attention_scores + mask`

4.4 Step 4: Applying Softmax

$$\text{Attention Weights}_{i,j} = \frac{e^{\text{Masked Scores}_{i,j}}}{\sum_{k=1}^{seq\_len} e^{\text{Masked Scores}_{i,k}}} \tag{7}$$

Explanation:

- The softmax function converts scores into a probability distribution over all positions

- Each row of the resulting matrix sums to 1
- These weights determine how much each token's value contributes to the output at each position
- In code: `attention_weights = np.exp(attention_scores) / np.sum(np.exp(attention_scores), axis=1, keepdims=True)`

4.5 Step 5: Computing Weighted Sum

$$\text{Output} = \text{Attention Weights} \cdot V \tag{8}$$

Explanation:

- Each position's output is a weighted average of all value vectors
- The weights come from the attention distribution calculated in Step 4
- This allows the model to focus on relevant parts of the input sequence
- The output has shape (seq_len, d_v), the same as the value matrix V
- In code: `output = np.dot(attention_weights, V)`

5 Example Usage

The code includes an example to demonstrate the self-attention mechanism:

```python
# Example usage
if __name__ == "__main__":
    # Example input
    X = np.array([[1, 0], [0, 1]])
    W_q = np.array([[1, 0], [0, 1]])
    W_k = np.array([[1, 0], [0, 1]])
    W_v = np.array([[1, 2], [3, 4]])

    # Compute query, key, and value matrices
    Q, K, V = compute_qkv(X, W_q, W_k, W_v)

    # Apply self-attention
    output = self_attention(Q, K, V)

    print("Self-Attention Output:")
    print(output)
```

Listing 3: Example Usage

5.1 Analysis of the Example

Let's trace through this example step by step:

Step 1: Set up input and weight matrices

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{9}$$

$$W_Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{10}$$

$$W_K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{11}$$

$$W_V = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{12}$$

Step 2: Compute Q, K, V matrices

$$Q = X \cdot W_Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{13}$$

$$K = X \cdot W_K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{14}$$

$$V = X \cdot W_V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{15}$$

Step 3: Compute attention scores

$$\text{Attention Scores} = Q \cdot K^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^T \tag{16}$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{17}$$

Step 4: Scale attention scores

$$\text{Scaled Scores} = \frac{\text{Attention Scores}}{\sqrt{d_k}} = \frac{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} \end{bmatrix} \tag{18}$$

Step 5: Apply softmax to get attention weights For the first row:

$$\text{Softmax}\left(\frac{1}{\sqrt{2}}, 0\right) = \frac{\left(e^{\frac{1}{\sqrt{2}}}, e^0\right)}{\left(e^{\frac{1}{\sqrt{2}}} + e^0\right)} \tag{19}$$

$$= \frac{\left(e^{\frac{1}{\sqrt{2}}}, 1\right)}{e^{\frac{1}{\sqrt{2}}} + 1} \tag{20}$$

$$\approx \left(\frac{e^{0.7071}}{e^{0.7071} + 1}, \frac{1}{e^{0.7071} + 1}\right) \tag{21}$$

$$\approx (0.6699, 0.3301) \tag{22}$$

For the second row:

$$\text{Softmax}\left(0, \frac{1}{\sqrt{2}}\right) = \frac{\left(e^0, e^{\frac{1}{\sqrt{2}}}\right)}{\left(e^0 + e^{\frac{1}{\sqrt{2}}}\right)} \tag{23}$$

$$= \frac{\left(1, e^{\frac{1}{\sqrt{2}}}\right)}{1 + e^{\frac{1}{\sqrt{2}}}} \tag{24}$$

$$\approx \left(\frac{1}{1 + e^{0.7071}}, \frac{e^{0.7071}}{1 + e^{0.7071}}\right) \tag{25}$$

$$\approx (0.3301, 0.6699) \tag{26}$$

So our attention weights matrix is approximately:

$$\text{Attention Weights} \approx \begin{bmatrix} 0.6699 & 0.3301 \\ 0.3301 & 0.6699 \end{bmatrix} \tag{27}$$

Step 6: Compute weighted sum of values

$$\text{Output} = \text{Attention Weights} \cdot V \tag{28}$$

$$\approx \begin{bmatrix} 0.6699 & 0.3301 \\ 0.3301 & 0.6699 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{29}$$

$$\approx \begin{bmatrix} 0.6699 \cdot 1 + 0.3301 \cdot 3 & 0.6699 \cdot 2 + 0.3301 \cdot 4 \\ 0.3301 \cdot 1 + 0.6699 \cdot 3 & 0.3301 \cdot 2 + 0.6699 \cdot 4 \end{bmatrix} \tag{30}$$

$$\approx \begin{bmatrix} 0.6699 + 0.9903 & 1.3398 + 1.3204 \\ 0.3301 + 2.0097 & 0.6602 + 2.6796 \end{bmatrix} \tag{31}$$

$$\approx \begin{bmatrix} 1.6602 & 2.6602 \\ 2.3398 & 3.3398 \end{bmatrix} \tag{32}$$

6 Interpretation of Self-Attention

Self-attention allows the model to weigh the importance of different positions in a sequence when representing each position:

- Each position attends to all positions in the sequence, creating a context-aware representation
- The strength of attention between positions depends on their content (through the Q and K matrices)
- Positions with similar query and key representations will have stronger attention weights
- The ultimate representation of each position is a weighted combination of all value vectors
- This mechanism enables the model to capture long-range dependencies in the sequence

7 Conclusion

The implemented self-attention mechanism follows the formulation in the "Attention Is All You Need" paper, which introduced the Transformer architecture. The key steps are:

1. Transform input into query, key, and value representations
2. Compute attention scores through dot products between queries and keys
3. Scale the scores and optionally apply masking
4. Convert scores to probability distributions using softmax
5. Create the output as weighted sums of values based on attention weights

This self-attention mechanism is the fundamental building block that allows Transformer models to process sequences without recurrence or convolution, leading to more parallelizable and effective sequence modeling.