Implementation of Rotary Embedding

Fundamental of Artificial Intelligence (IM20500572)

Krit Surintraboon

C1TB1707

krit.surintraboon.q7@dc.tohoku.ac.jp

1. Introduction

Positional embeddings play a critical role in transformer models, enabling them to capture the sequential structure of data, such as text. Traditionally, absolute positional embeddings have been used to represent the position of each token in a sequence by adding sinusoidal position vectors to the token embeddings. While this method is simple, it faces challenges such as limited sequence length, and encapsulation of distance between each token in the sequence.

To address these limitations, relative positional embeddings were introduced. Here, the model encodes the distance relationship between tokens, also allowing the scalability of the embeddings. However, they are computationally complex and slow.

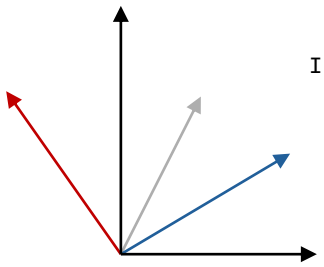
Rotary Position Embeddings (RoPE) address these challenges by integrating positional information directly into the self-attention mechanism. By rotating query and key embeddings based on token positions. Because the vector is rotated, the absolute position information can be implied by the rotated angle and the relative distance can be interpreted from the angle between 2 token's vectors.

In the following report, I will attempt to implement the rotary position embedding into the provided transformer code.

2. Background: Rotary positional embedding (RoPE)

Rotary Position Embeddings (RoPE) are a method for encoding positional information by directly modifying the query and key embeddings inside the self-attention mechanism.

In order to do that, RoPE rotates the query and key vector by an angle depending on the position of the token vector inside the sequence. As a simplified example, assuming that the embedded vectors are 2 dimensional. The embedding vector will be represented as shown below.



I thought I thought what I thought was right.

Figure 1: Intuitive diagram of rotating based on the position of the token in the sentences

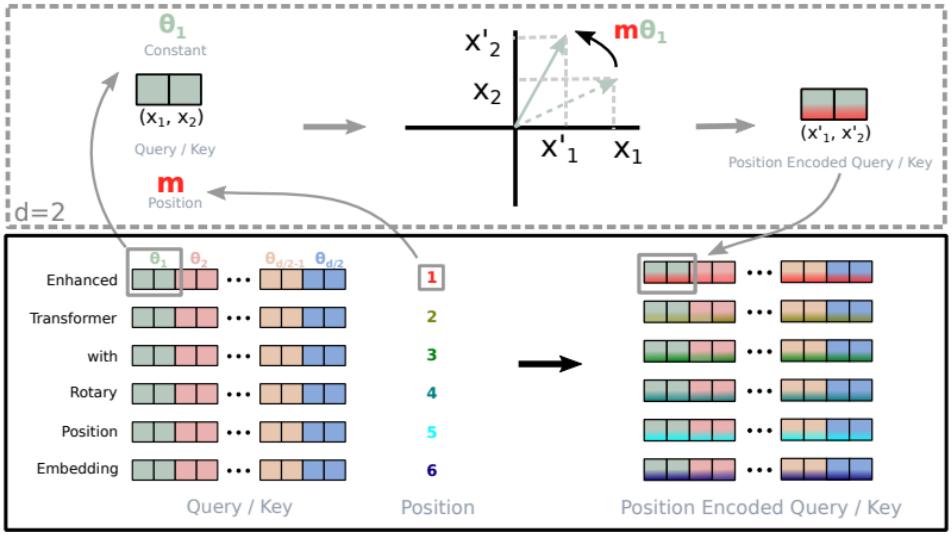


Figure 2: Rotary embedding scheme

Since the rotations are applied directly to the query and key vectors. The 2-dimensional mathematical formula can be expressed as follows.

$$f_{\{q,k\}}(x_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos(m\theta) \end{pmatrix} \begin{pmatrix} W_{\{q,k\}}^{(11)} & W_{\{q,k\}}^{(12)} \\ W_{\{q,k\}}^{(21)} & W_{\{q,k\}}^{(22)} \end{pmatrix} \begin{pmatrix} x_m^{(1)} \\ x_m^{(1)} \end{pmatrix} \tag{1}$$

, where m is the position of the token in the sequence and $(x_m^{(1)}, x_m^{(2)})$ is embedded vector x_m expressed in 2D coordinates. In general case, Eq. (1) can be expanded by dividing the d-dimension space that contains the embedding of the token into $d/2$ sub-spaces of 2-dimension space, as shown in Eq. (2) and figure 2.

$$f_{\{q,k\}}(x_m, m) = R_{\theta,m}^d W_{\{q,k\}} x_m \tag{2}$$

, where

$$R_{\Theta,m}^d = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & \cdots & 0 & 0 \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & \cdots & 0 & 0 \\ 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \cdots & 0 & 0 \\ 0 & \sin(m\theta_2) & \cos(m\theta_2) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \cos(m\theta_d/2) & -\sin(m\theta_d/2) \\ 0 & 0 & 0 & \cdots & \sin(m\theta_d/2) & \cos(m\theta_d/2) \end{pmatrix} \tag{3}$$

The matrix $R_{\Theta,m}^d$ is the rotary matrix with predefined $\Theta = \{\theta = 10000^{-2(i-1)d}, i \in [1, 2, \ldots, d/2]$. Using the sparsity of the matrix, a more computational efficient realization using element wise operation can be expressed in Eq. (4). Here, $v \in \mathbb{R}^d$, representing the query or key vectors $W_{\{q,k\}} x_m$.

$$R^{d}_{\Theta,m}v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ \vdots \\ v_{d-1} \\ v_d \end{pmatrix} \otimes \begin{pmatrix} \cos(m\theta_1) \\ \cos(m\theta_1) \\ \cos(m\theta_2) \\ \cos(m\theta_2) \\ \vdots \\ \cos(m\theta_d/2) \\ \cos(m\theta_d/2) \end{pmatrix} + \begin{pmatrix} -v_2 \\ v_1 \\ -v_4 \\ v_3 \\ \vdots \\ -v_d \\ v_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin(m\theta_1) \\ \sin(m\theta_1) \\ \sin(m\theta_2) \\ \sin(m\theta_2) \\ \vdots \\ \sin(m\theta_d/2) \\ \sin(m\theta_d/2) \end{pmatrix} \tag{4}$$

3. Method: Implementation

In order to implement the rotary embedding into the provided transformer code efficiently, we will use Eq. (4) as a foundation. Firstly, we aimed to create sin and cosine tensors by creating two functions that will generate the sinusoidal tensors with the same dimension as the query and key tensors. In both functions, we first created a sinusoidal tensor with dimension equal to (seq_len, d_k) as shown in figure 3. Then, increase the dimension of the sinusoidal tensor such that it can be broadcasted and performed element-wise operation in the next step.

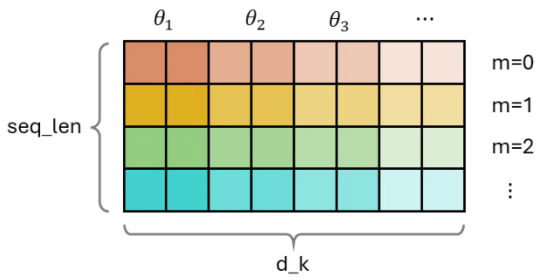


Figure 3: Last 2 dimension of the sinusoidal tensor

```
def generate_sin_embedding(seq_len, d_k):
    position = torch.arange(0, seq_len).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, d_k, 2, dtype=torch.float32) * -(math.log(10000.0) / d_k))
    # Compute sinusoidal embedding (cos for tensor 1)
    sinusoidal_emb = torch.zeros(seq_len, d_k)
    sinusoidal_emb[:, 0::2] = torch.sin(position * div_term)
    sinusoidal_emb[:, 1::2] = torch.sin(position * div_term)
    return sinusoidal_emb.unsqueeze(0).unsqueeze(0)  # Add batch dimension for broadcasting

def generate_cos_embedding(seq_len, d_k):
    position = torch.arange(0, seq_len).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, d_k, 2, dtype=torch.float32) * -(math.log(10000.0) / d_k))
    # Compute sinusoidal embedding (sin for tensor 2)
    sinusoidal_emb = torch.zeros(seq_len, d_k)
    sinusoidal_emb[:, 0::2] = torch.cos(position * div_term)
    sinusoidal_emb[:, 1::2] = torch.cos(position * div_term)
    return sinusoidal_emb.unsqueeze(0).unsqueeze(0)  # Add batch dimension for broadcasting

```

Next, we created a function that will receive a query or a key tensor, and output the tensor embedded with rotary positional embedding. In the first step (line 3-7), we generated another tensor that corresponds to $(-v_2, v_1, -v_4, v_3, \ldots, -v_d, v_{d-1})$ by negating and swapping the element in the last dimension of the inputted tensor. Then, we generated the sinusoidal tensors using the function we created previously (line 11-12). After that, the element-wise operation is performed according to Eq. (4) (line 14).

```
 1. def apply_rotary_emb(tensor1, d_k):
 2.     last_dim_size = tensor1.size()[-1]
 3.     swap_indices = torch.arange(0, last_dim_size, 2)
 4.     negate_indices = torch.arange(1, last_dim_size, 2)
 5.     tensor2 = swapped_tensor = tensor1.clone()
 6.     tensor2[..., swap_indices] = -tensor1[..., negate_indices]
 7.     tensor2[..., negate_indices] = tensor1[..., swap_indices]
 8.
 9.     #print("tensor2 size", tensor2.size())
10.
11.     sin = generate_sin_embedding(tensor1.size()[-2], d_k).to(tensor1.device)
12.     cos = generate_cos_embedding(tensor2.size()[-2], d_k).to(tensor1.device)
13.
14.     tensor_rotated = (tensor1*cos)+(tensor2*sin)
15.
16.     return tensor_rotated
17.
```

In the next step, we modified the attention function by applying the rotary embedding to the query and key tensors before the dot product is performed to compute the attention (line 5-7).

```
 1. def attention(query, key, value, mask=None, dropout=None, rotary_emb=None):
 2.
 3.     d_k = query.size(-1)
 4.
 5.     if rotary_emb is not None:
 6.       query = apply_rotary_emb(query, d_k)
 7.       key = apply_rotary_emb(key, d_k)
 8.
 9.     # print("q_size", query.size())
10.     # print("k_size", key.size())
11.
12.     "Compute 'Scaled Dot Product Attention'"
13.
14.     scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k) # Swapping last two
dimensions
15.     if mask is not None:
16.         scores = scores.masked_fill(mask == 0, -1e9)
17.     p_attn = scores.softmax(dim=-1)
18.     if dropout is not None:
19.         p_attn = dropout(p_attn) #fatte
20.     return torch.matmul(p_attn, value), p_attn
21.
```

Lastly, we modified the make_model function by removing the absolute positional embedding that is previously included (line 12-13)

```
 1. def make_model(
 2.     src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1
 3.):
 4.     "Helper: Construct a model from hyperparameters."
 5.     c = copy.deepcopy
 6.     attn = MultiHeadedAttention(h, d_model)
 7.     ff = PositionwiseFeedForward(d_model, d_ff, dropout)
 8.     model = EncoderDecoder(
 9.         Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
10.         Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N),
11.         Embeddings(d_model, src_vocab),
12.         Embeddings(d_model, tgt_vocab),
13.         Generator(d_model, tgt_vocab),
14.     )
15.
16.     # This was important from their code.
17.     # Initialize parameters with Glorot / fan_avg.
18.     for p in model.parameters():
19.         if p.dim() > 1:
20.             nn.init.xavier_uniform_(p)
21.     return model
22.
```

4. Result

In this section, we will compare the performance of the original transformer model with absolute positional embedding (provided code) and the tranformer model with the rotary positional embedding.

4.1 Experimental environment

Training hyperparameter	
Training GPU	NVIDIA H100 PCIe
Batch Size	32
Epochs Number	30
Accumulated Iteration	10
Base Learning Rate	1.0
Max Padding	72

4.2 Machine translation task

Here, we test the performance of both models by performing a machine translation task with 5 examples and calculating the average BLEU score for each model. In each example, same source text and the target text were used, as shown in the following.

```
Source Text (Input)         : <s> Eine Gruppe von Männern lädt <unk> auf einen Lastwagen </s>
Target Text (Ground Truth)  : <s> A group of men are loading cotton onto a truck </s>
```

Example	Absolute Positional Embedding	Rotary Embedding
# 1	0.50741	0.62778
# 2	0.44973	0.53934
# 3	0.50741	0.53934
# 4	0.50741	0.53934
# 5	0.41284	0.53934
Average	0.47696	0.55703

The attention scores in the encoder part, decoder part, and the cross-attention part for both embedding mechanisms were visualized and compared as follows.

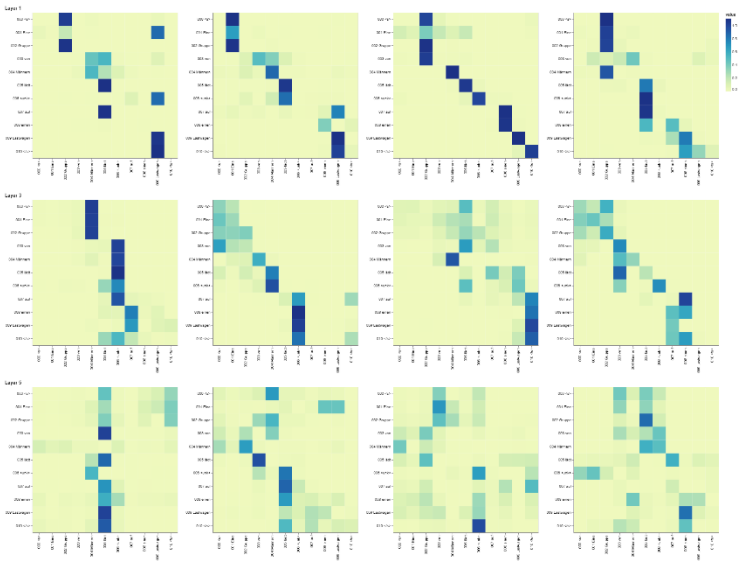


Figure 4: Self-attention score in the encoder part (Absolute positional embedding)

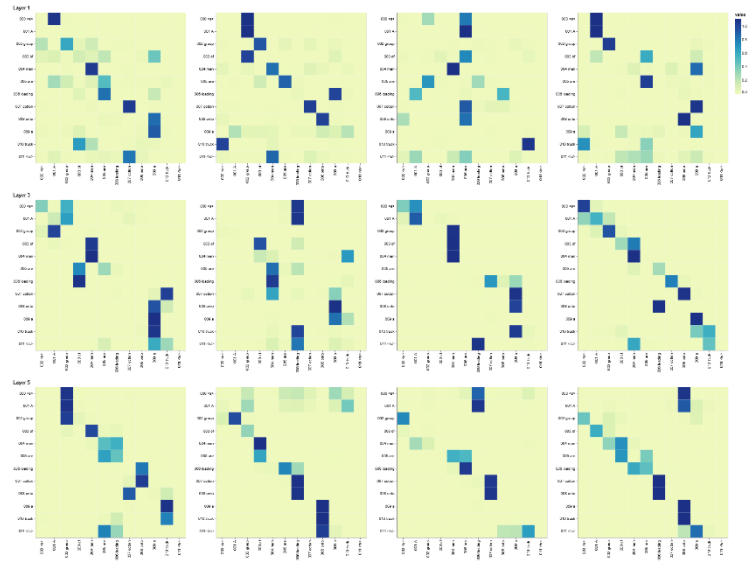


Figure 5: Self-attention score in the encoder part (Rotary positional embedding)

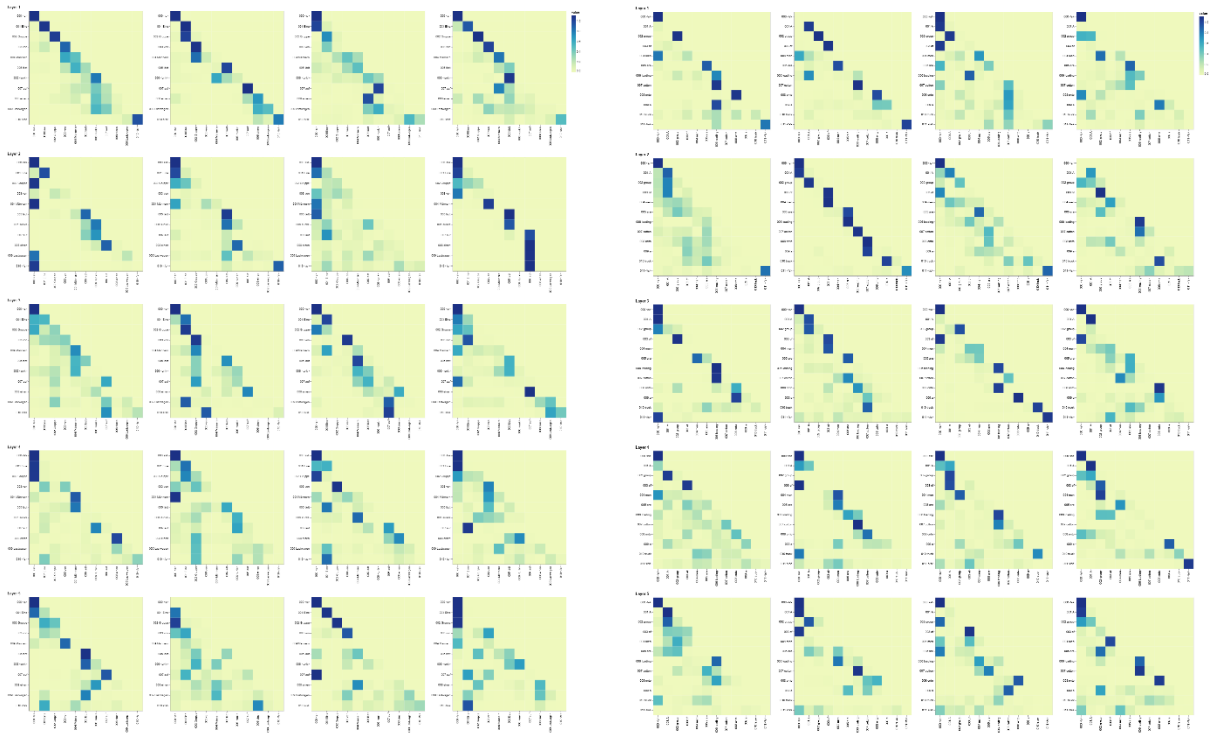


Figure 6: Self-attention score in the decoder part
Absolute positional embedding [left], Rotary positional embedding [right]

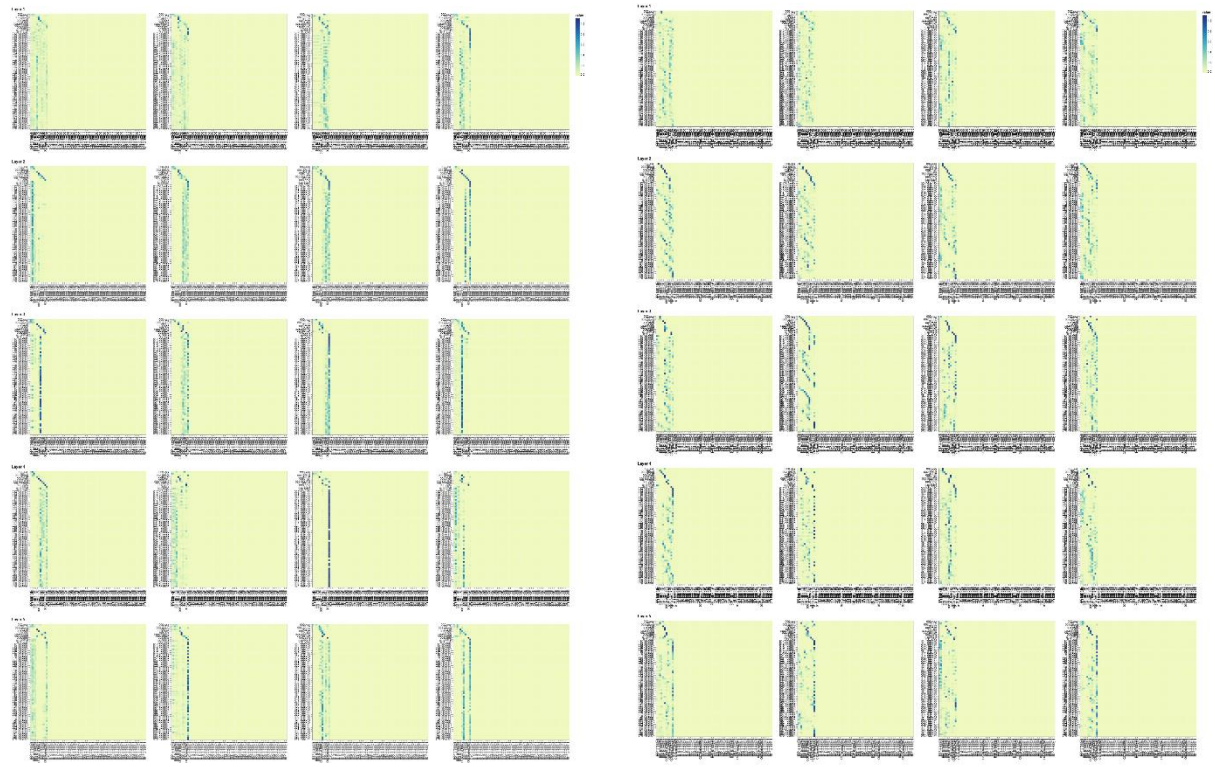


Figure 7: Cross-attention score in the decoder part
Absolute positional embedding [left], Rotary positional embedding [right]

5. Discussion

The BLEU score comparison results demonstrate that rotary embedding outperforms absolute positional embedding across all examples, with higher and more consistent scores. That is, rotary embedding shows a stable performance with BLEU score around 0.53934 across four examples, while absolute positional embedding varied, dropping as low as 0.41284 in example #5. On average, rotary embedding score showed 8.01% improvement over the absolute positional embedding.

The comparison of absolute positional embedding and rotary embedding is further illustrated through the self-attention score visualizations in Figures 4, 5, and 6. From these figures, it can be seen that the rotary embedding captured the more diverse attention pattern in the initial layers. After that, the attention pattern become sharper as it passes through more layer. This is in contrast to the absolute positional embedding, where the attention pattern becomes more diverse as it passes through more layers. This attention pattern likely contributes to the improved BLEU scores of rotary embeddings, as the last layer show more consistent pattern.

6. Conclusion

In conclusion, rotary embedding, which is the embedding that incorporates both absolute and relative distances outperforms absolute positional embedding. The sharper and more structured attention patterns in the final layer of the rotary embedding model likely contribute to its higher BLEU score. This rotary embedding a robust choice for tasks requiring positional awareness such as machine translation.