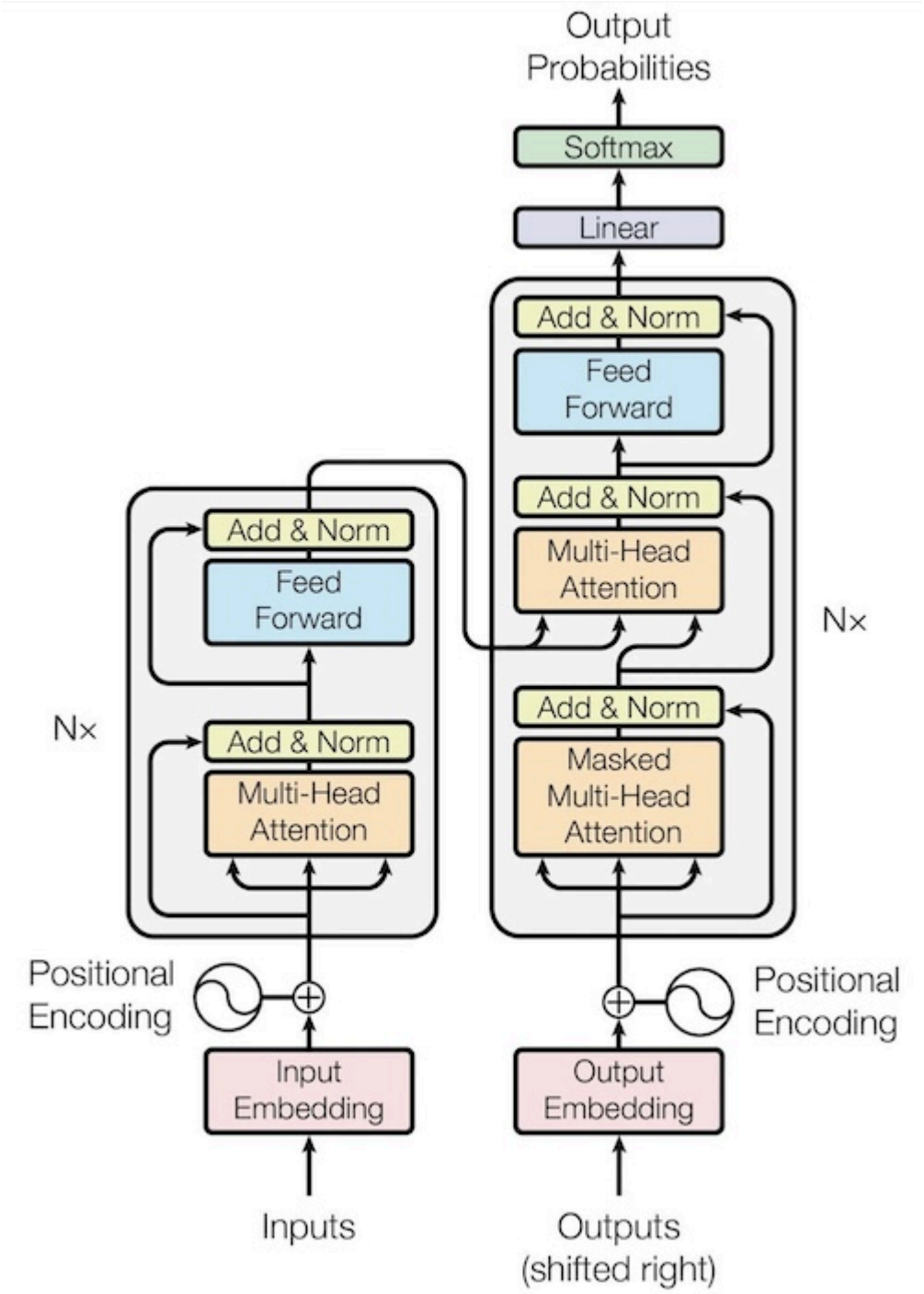
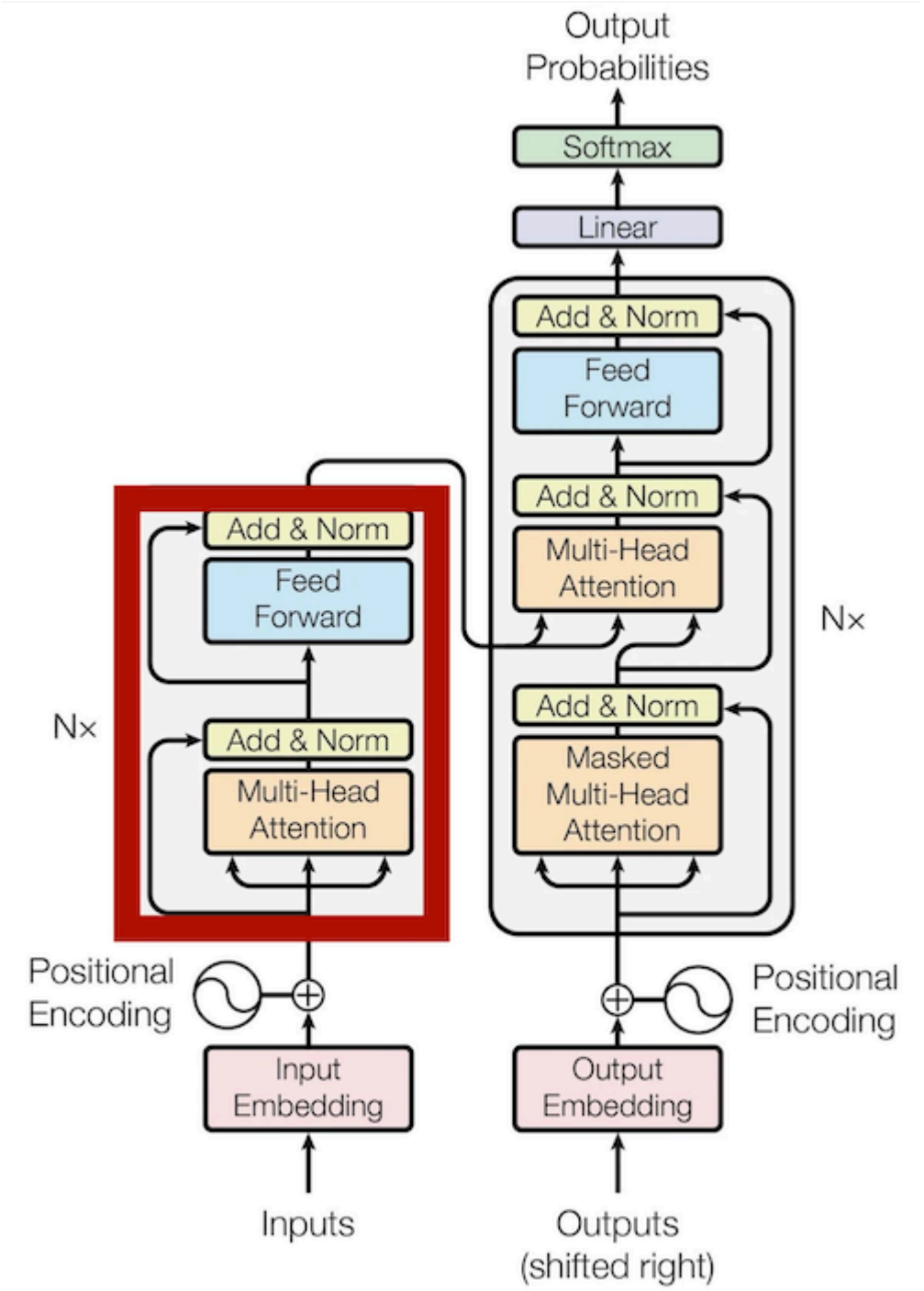
Step by Step to Build a Multi-Lingual Translation Language model with Transformer

language translationIn the "Attention is All You Need" paper, the authors introduced the self-attention mechanism within the encoder-decoder architecture for building translation models. This guide provides a step-by-step tutorial on constructing a translation model using the Transformer architecture. We will code the encoder and decoder, train the model, save checkpoints, and perform inference. This post offers a comprehensive, hands-on approach to building a translation model with the Transformer. First lets take a look at this visual representation of the transformer architecture



Encoder

The encoder's purpose is to obtain the best contextual representation for the source language. It consists of multiple layers, and the input goes through these layers multiple times to yield optimal results. This iterative process allows the encoder to capture the nuances and dependencies within the source language. On the left side in the red circle is the encoder layer:



Positional Encoding

Before we dive into the encoder layers. Lets take a look at embedding and positional encoding. In simple terms, we need to mark the position of each token to understand the context. After all, 'I love you' and 'You love me' are very different statements-just ask anyone who's ever mixed them up in a text message! Why do we use sinusoidal positional encoding you may ask? please check this article for reference: https://arxiv.org/pdf/2106.02795

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len = 5000):
```

```
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        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))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, input):
        input = input + self.pe[:, :input.size(1)].detach()
        return self.dropout(input
```

Token embedding

Consider token embedding as a way to represent each token in a high-dimensional space. Each token is assigned a unique vector representation, which encodes its semantic and syntactic properties. By learning these embeddings during the training process, the model can capture the relationships and similarities between different tokens. The token embeddings are typically initialized randomly and then learned and updated during the training process. Heres the code for token embedding:

```
class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size, d_model, max_len=5000):
        super(TokenEmbedding, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.max_len = max_len

    def forward(self, input):
        # Truncate or pad the input sequences to max_len
        input = input[:, :self.max_len]
        return self.embedding(input
```

Self-Attention and Multi-head Attention

As the model processes each word in the input sequence, self-attention focuses on all the words in the entire input sequence, helping the model encode the current word better. During this process, the self-attention mechanism integrates the understanding of all relevant words into the word being processed. More specifically, its functions include:
Self-attention:

- Sequence Modeling: Self-attention can be used for modeling sequence data (such as text, time series, audio, etc.). It captures dependencies at different positions within the sequence, thus better understanding the context. This is highly useful for tasks like machine translation, text generation, and sentiment analysis.

- Parallel Computation: Self-attention allows for parallel computation, which means it can be effectively accelerated on modern hardware. Compared to sequential models like RNNs and CNNs, it is easier to train and infer efficiently on GPUs and TPUs (because scores can be computed in parallel in self-attention).

- Long-Distance Dependency Capture: Traditional recurrent neural networks (RNNs) may face issues like vanishing or exploding gradients when processing long sequences. Self-attention handles long-distance dependencies better because it doesn't require sequential processing of the input sequence.

- *Multi-head attention:*

- Enhanced Ability to Focus on Different Positions: Multi-head attention extends the model's ability to focus on different positions within the input sequence.

    - Multiple Sets of Query/Key/Value Weight Matrices: There are multiple sets of query, key, and value weight matrices (Transformers use eight attention heads), each randomly initialized. Similar to the self-attention mechanism, matrix X is multiplied by WQ, WK, and WV to produce the query, key, and value matrices.

```
 class ScaledProductAttn(nn.Module):
    def __init__(self, dropout = 0.1):
        super(ScaledProductAttn, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, query, key, value, attn_mask = None):
        _, _, _, d_k= query.shape
        assert d_k != 0
        attn = torch.matmul(query, key.transpose(-1, -2)) /
np.sqrt(d_k)
        if attn_mask is not None:
            attn = attn.masked_fill(attn_mask == False, float('-
inf'))
        attn = self.dropout(self.softmax(attn))
        context = torch.matmul(attn, value)
        return contex

class MultiHeadAttn(nn.Module):
    def __init__(self, n_head, d_model,dropout = 0.1):
        super(MultiHeadAttn, self).__init__()
        self.Q = nn.Linear(d_model, d_model)
```

```
            self.K = nn.Linear(d_model, d_model)
            self.V = nn.Linear(d_model, d_model)
            self.n_head = n_head
            self.scaled_dot_attn = ScaledProductAttn(dropout)
            self.dropout = nn.Dropout(p = dropout)
            self.norm = nn.LayerNorm(d_model)

        def forward(self, x,  attn_mask=None):
            batch_size, seq_len, d_model = x.shape
            h_dim = d_model // self.n_head
            assert h_dim * self.n_head == d_model
            Q = self.Q(x).view(batch_size, seq_len, self.n_head,
    h_dim).permute(0,2,1,3)
            K = self.K(x).view(batch_size, seq_len, self.n_head,
    h_dim).permute(0,2,1,3)
            V = self.V(x).view(batch_size, seq_len, self.n_head,
    h_dim).permute(0,2,1,3)
            # print(f"the shape of Q: {Q}")
            # print(f"the shape of K: {K}")
            # print(f"the shape of V: {V}")
            # print(f"attn_mask shape: {attn_mask.shape}")
            if attn_mask is not None:
                attn_mask = attn_mask.expand(batch_size, self.n_head,
    seq_len, seq_len)  # Expanding to [batch_size, n_head, seq_len,
    seq_len]
            # print(f"attn_mask shape after expansion:
    {attn_mask.shape}")
            attn_score = self.scaled_dot_attn(Q, K, V, attn_mask)
            attn_score =
    attn_score.permute(0,2,1,3).reshape(batch_size, seq_len, -1)

            attn_score = self.dropout(attn_score)
            attn_score = self.norm(attn_score + x)
            return attn_score
```

We put everything togther, this is the code for the encoder layer

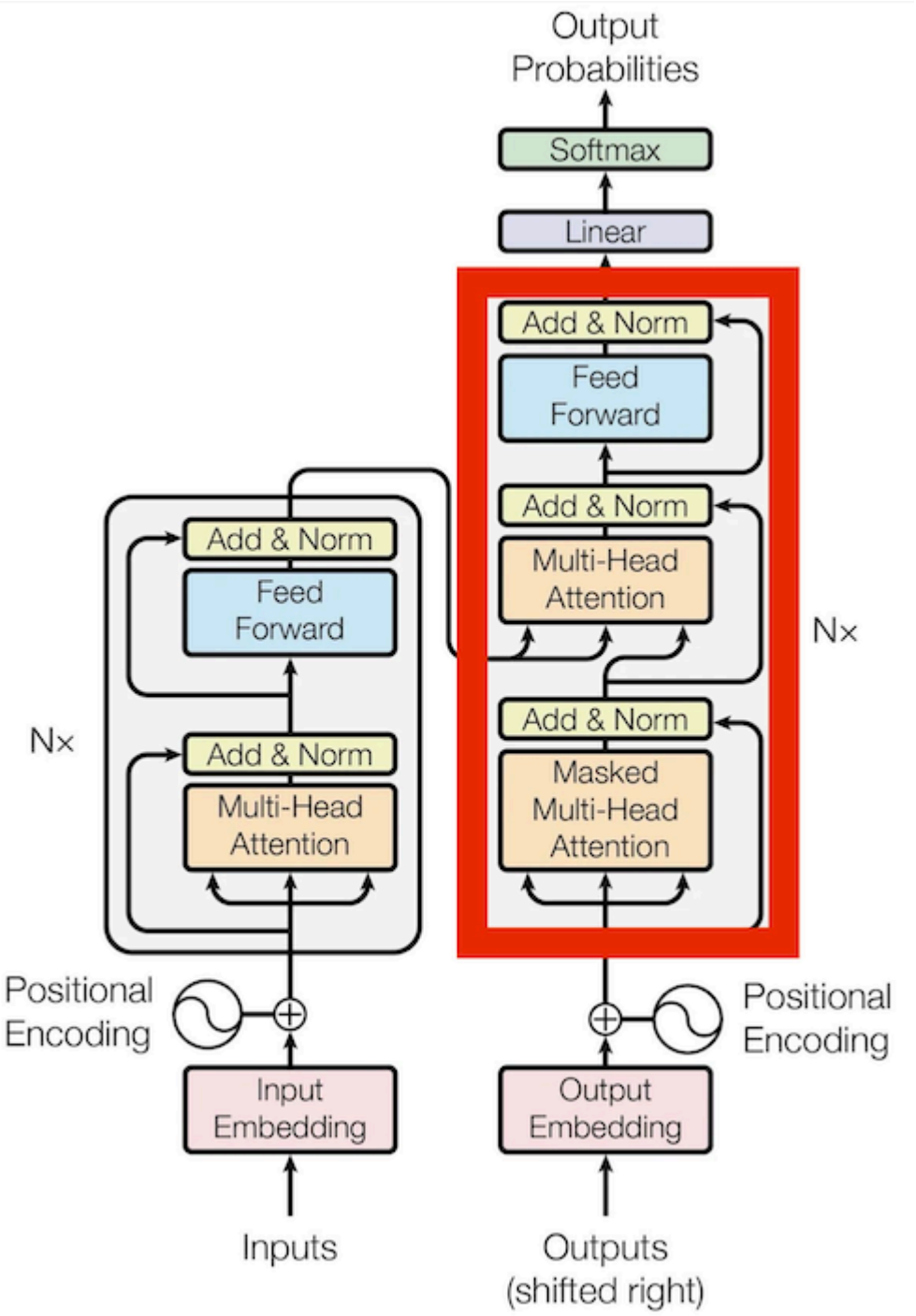
```
class EncoderLayer(nn.Module):
"""
  multi-head attention
  feedforward network
  normalization layers
  regularization

"""
    def __init__(self, n_head, d_model, d_ff, dropout=0.1):
        super(EncoderLayer, self).__init__()
        self.multiheadattn = MultiHeadAttn(n_head, d_model, dropout)
        self.fnn = PoswiseFeedForwardNet(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
```

```
    def forward(self, x, attn_mask):
        x = self.norm1(x)
        attn_output = self.multiheadattn(x, attn_mask)
        x = x + self.dropout(attn_output)
        x = self.norm2(x)
        ff_output = self.fnn(x)
        x = x + self.dropout(ff_output)
        return x
```

Decoder

The decoder is a crucial component in the Transformer architecture, responsible for generating the translated tokens in the target language. Its main objective is to capture the dependencies and relationships among the translated tokens while utilizing the representations from the encoder. The decoder operates by first performing self-attention on each of the translated tokens in the source language. This self-attention mechanism allows the decoder to consider the context and dependencies within the translated sequence itself. By attending to its own previous outputs, the decoder can generate more coherent and contextually relevant translations. The red circlec is the decoder part of the architecture:



The decoder architecture consists of several key components, including multi-head self-attention, cross-attention, normalization, and regularization techniques. These components work together to enable the decoder to effectively capture the nuances and dependencies in the target language.

Masked Multi-Head attention:

We shared the multi-head attention class with the encoder:

```
class ScaledProductAttn(nn.Module):
    def __init__(self, dropout = 0.1):
        super(ScaledProductAttn, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
```

```
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, query, key, value, attn_mask = None):
        _, _, _, d_k= query.shape
        assert d_k != 0
        attn = torch.matmul(query, key.transpose(-1, -2)) / np.sqrt(d_k)
        if attn_mask is not None:
            attn = attn.masked_fill(attn_mask == False, float('-inf'))
        attn = self.dropout(self.softmax(attn))
        context = torch.matmul(attn, value)
        return context

class MultiHeadAttn(nn.Module):
    def __init__(self, n_head, d_model,dropout = 0.1):
        super(MultiHeadAttn, self).__init__()
        self.Q = nn.Linear(d_model, d_model)
        self.K = nn.Linear(d_model, d_model)
        self.V = nn.Linear(d_model, d_model)
        self.n_head = n_head
        self.scaled_dot_attn = ScaledProductAttn(dropout)
        self.dropout = nn.Dropout(p = dropout)
        self.norm = nn.LayerNorm(d_model)


    def forward(self, x,  attn_mask=None):
        batch_size, seq_len, d_model = x.shape
        h_dim = d_model // self.n_head
        assert h_dim * self.n_head == d_model
        Q = self.Q(x).view(batch_size, seq_len, self.n_head,
h_dim).permute(0,2,1,3)
        K = self.K(x).view(batch_size, seq_len, self.n_head,
h_dim).permute(0,2,1,3)
        V = self.V(x).view(batch_size, seq_len, self.n_head,
h_dim).permute(0,2,1,3)

        if attn_mask is not None:
            attn_mask = attn_mask.expand(batch_size, self.n_head, seq_len,
seq_len)  # Expanding to [batch_size, n_head, seq_len, seq_len]
        # print(f"attn_mask shape after expansion: {attn_mask.shape}")
        attn_score = self.scaled_dot_attn(Q, K, V, attn_mask)
        attn_score = attn_score.permute(0,2,1,3).reshape(batch_size,
seq_len, -1)

        attn_score = self.dropout(attn_score)
        attn_score = self.norm(attn_score + x)

        return attn_score
```

Cross-Head Attention:

1. Adding special tokens: It is essential to add special tokens, such as the beginning-of-sequence (BOS) and end-of-sequence (EOS) tokens, to the training data for the

source language. These tokens mark the start and end of the translated sequence, providing the decoder with explicit boundaries for generating the output.

2. Cross-attention: Cross-attention is a mechanism that allows the decoder to attend to the relevant information from the encoder's outputs. At each decoding step, the decoder uses cross-attention to query the encoder's representations and extract the most relevant information for generating the next token. This enables the decoder to effectively utilize the source language context while producing the translation.

3. look-forward mask: This mask is crucial for preventing the decoder from attending to future tokens during the self-attention process. When generating a translated token at a particular position, the decoder should only consider the tokens that have been generated so far, up to that position. It should not have access to future tokens that are yet to be generated. The decoder mask is a binary matrix that has a triangular shape. It is constructed in such a way that the upper triangular part of the matrix is filled with zeros, and the lower triangular part (including the diagonal) is filled with ones. The mask is applied to the attention weights before the softmax operation in the self-attention mechanism. Here's an example of how the decoder mask works: 1 2 3 4 5 1 1 0 0 0 0 2 1 1 0 0 0 3 1 1 1 0 0 4 1 1 1 1 0 5 1 1 1 1 1 Some people might wonder about the specific details of cross-attention and how it differs from self-attention. In cross-attention, the queries come from the decoder's self-attention layer, while the keys and values come from the encoder's outputs. This allows the decoder to selectively focus on different parts of the source sequence based on its current state and the previously generated tokens. By employing these techniques and carefully designing the decoder architecture, the Transformer model can generate high-quality translations that capture the meaning and context of the source language accurately. class DecoderLayer(nn.Module): def **init**(self, n_head, d_model, d_ff, dropout=0.1): super(DecoderLayer, self).**init**()

print('self attn in decoder----------')

self.self_attn = MultiHeadAttn(n_head, d_model, dropout)

print('cross attn in decoder **********')

self.cross_attn = MultiHeadCrossAttn(n_head, d_model) self.feed_forward = PoswiseFeedForwardNet(d_model, d_ff, dropout) self.norm1 = nn.LayerNorm(d_model) self.norm2 = nn.LayerNorm(d_model) self.norm3 = nn.LayerNorm(d_model) self.dropout1 = nn.Dropout(dropout) self.dropout2 = nn.Dropout(dropout) self.dropout3 = nn.Dropout(dropout)

def forward(self, x, memory, src_mask=None, tgt_mask=None): x = self.norm1(x) x2 = self.self_attn(x, attn_mask=tgt_mask) x = x + self.dropout1(x2) x = self.norm2(x) x2 = self.cross_attn(x, memory, src_mask) x = x + self.dropout2(x2) x = self.norm3(x) x2 = self.feed_forward(x) x = x + self.dropout3(x2) return Then its time to put encoder decoder together: class EncoderDecoder(nn.Module): def **init**(self, encoder, decoder, device): super(EncoderDecoder, self).**init**() self.encoder = encoder self.decoder = decoder self.device = device

def padding_mask(self, input): input = input.to(self.device) input_mask = (input != 0).unsqueeze(1).unsqueeze(2).to(self.device) return input_mask

def target_mask(self, target): target = target.to(self.device) target_pad_mask = (target != 0).unsqueeze(1).unsqueeze(2).to(self.device) # shape(batch_size, 1, 1, seq_length) target_sub_mask = torch.tril(torch.ones((target.shape[1], target.shape[1]), device=self.device)).bool() # shape(seq_len, seq_len) target_mask = target_pad_mask & target_sub_mask # shape(batch_size, 1, seq_length, seq_length) return target_mask

def forward(self, input, target): input = input.to(self.device) target = target.to(self.device)

input_mask = self.padding_mask(input) target_mask = self.target_mask(target)

encoder feed through

encoded_input = self.encoder(input, input_mask)

decoder feed through

output = self.decoder(target, encoded_input, input_mask, target_mask)

return output Tokenization For tokenization, i used pre-trianed sentencepiece bpe tokenizer, and retrained with my data to obtain the tokenizer def

train_bpe_tokneizer(input_file, model_prefix): vocab_size = 10000 # Adjust the vocabulary size as needed model_type = 'bpe' # Use the BPE model type

Define the special tokens

bos_token = '<s>' eos_token = '</s>'

Train the SentencePiece model

spm.SentencePieceTrainer.train(input=input_file, model_prefix=model_prefix, vocab_size=vocab_size, model_type=model_type, shuffle_input_sentence=True, input_sentence_size=1000000, character_coverage=0.9995, bos_piece=bos_token, eos_piece=eos_token, pad_id=0, unk_id=1, bos_id=2, eos_id=3 Heres the training process:

Inference:

Example translation

src_sentence = "today is a good day" translated_sentence = translate(src_sentence, model, sp) print("Translated Sentence:", translated_sentence)

''' (German): Heute ist eine gute Tag '''

When implementing the decoder, there are a few important details to consider: