In [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

# ==========================
# Multi-Head Attention Module
# ==========================
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_dim, num_heads, dropout):
        super().__init__()
        assert emb_dim % num_heads == 0, "Embedding dimension must be divisible by
        self.emb_dim = emb_dim
        self.num_heads = num_heads
        self.head_dim = emb_dim // num_heads  # 每个头的维度

        # 线性变换，投影到 Q, K, V
        self.Q_linear = nn.Linear(emb_dim, emb_dim)
        self.K_linear = nn.Linear(emb_dim, emb_dim)
        self.V_linear = nn.Linear(emb_dim, emb_dim)

        # 最终投影层
        self.linear_out = nn.Linear(emb_dim, emb_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, query, key, value, mask=None):
        batch_size = query.shape[0]

        # 计算 Q, K, V
        Q = self.Q_linear(query).view(batch_size, -1, self.num_heads, self.head_dim)
        K = self.K_linear(key).view(batch_size, -1, self.num_heads, self.head_dim).
        V = self.V_linear(value).view(batch_size, -1, self.num_heads, self.head_dim)

        # 计算注意力得分
        attn_scores = torch.einsum("bnqd, bnkd -> bnqk", Q, K) / (self.head_dim ** (

        # 应用 mask（处理填充或自回归）
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, float("-inf"))

        attn_probs = F.softmax(attn_scores, dim=-1)
        attn_probs = self.dropout(attn_probs)

        # 加权求和
        attn_output = torch.einsum("bnqk, bnvd -> bnqd", attn_probs, V)
        attn_output = attn_output.transpose(1, 2).contiguous().view(batch_size, -1,

        return self.linear_out(attn_output)


# ==========================
# Transformer Encoder Block
# ==========================
class TransformerEncoderBlock(nn.Module):
    def __init__(self, emb_dim, num_heads, forward_dim, dropout):
        super().__init__()

        self.self_attn = MultiHeadAttention(emb_dim, num_heads, dropout)

        self.norm1 = nn.LayerNorm(emb_dim, eps=1e-6)
        self.norm2 = nn.LayerNorm(emb_dim, eps=1e-6)
        # 前馈网络
        self.ffn = nn.Sequential(
            nn.Linear(emb_dim, forward_dim),
            nn.ReLU(),
            nn.Linear(forward_dim, emb_dim),
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        attn_output = self.self_attn(x, x, x, mask)
        x = self.dropout(self.norm1(attn_output + x))  # Add & Norm

        ffn_output = self.ffn(x)
        x = self.dropout(self.norm2(ffn_output + x))  # Add & Norm

        return x


# ==========================
# Transformer Encoder
# ==========================
class TransformerEncoder(nn.Module):
    def __init__(self, vocab_size, emb_dim, num_layers, num_heads, forward_dim, drop
        super().__init__()

        self.emb_dim = emb_dim
        self.embedding = nn.Embedding(vocab_size, emb_dim)
        self.pos_encoding = nn.Embedding(max_len, emb_dim)

        self.layers = nn.ModuleList([
            TransformerEncoderBlock(emb_dim, num_heads, forward_dim, dropout)
            for _ in range(num_layers)
        ])

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        batch_size, seq_len = x.shape
        device = x.device

        positions = torch.arange(0, seq_len, device=device).unsqueeze(0).expand(batc

        sum_emb = self.embedding(x) + self.pos_encoding(positions)
        out = self.dropout(sum_emb)

        for layer in self.layers:
            out = layer(out, mask)

        return out


# ==========================
# Transformer Decoder Block
# ==========================
class TransformerDecoderBlock(nn.Module):
    def __init__(self, emb_dim, num_heads, forward_dim, dropout):
        super().__init__()

        self.self_attn = MultiHeadAttention(emb_dim, num_heads, dropout)
        self.cross_attn = MultiHeadAttention(emb_dim, num_heads, dropout)

        self.norm1 = nn.LayerNorm(emb_dim, eps=1e-6)
        self.norm2 = nn.LayerNorm(emb_dim, eps=1e-6)
        self.norm3 = nn.LayerNorm(emb_dim, eps=1e-6)

        self.ffn = nn.Sequential(
            nn.Linear(emb_dim, forward_dim),
            nn.ReLU(),
            nn.Linear(forward_dim, emb_dim),
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, encoder_out, src_mask, tgt_mask):
        attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(self.dropout(attn_output) + x)  # Add & Norm

        attn_output = self.cross_attn(x, encoder_out, encoder_out, src_mask)
        x = self.norm2(self.dropout(attn_output) + x)  # Add & Norm

        ffn_output = self.ffn(x)
        x = self.norm3(self.dropout(ffn_output) + x)  # Add & Norm

        return x


# ==========================
# Transformer Decoder
# ==========================
class TransformerDecoder(nn.Module):
    def __init__(self, vocab_size, emb_dim, num_layers, num_heads, forward_dim, drop
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, emb_dim)
        self.pos_encoding = nn.Embedding(max_len, emb_dim)

        self.layers = nn.ModuleList([
            TransformerDecoderBlock(emb_dim, num_heads, forward_dim, dropout)
            for _ in range(num_layers)
        ])

        self.linear_out = nn.Linear(emb_dim, vocab_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, encoder_out, src_mask, tgt_mask):
        batch_size, seq_len = x.shape
        device = x.device

        positions = torch.arange(seq_len, device=device).unsqueeze(0).expand(batch_s

        sum_emb = self.embedding(x) + self.pos_encoding(positions)
        out = self.dropout(sum_emb)

        for layer in self.layers:
            out = layer(out, encoder_out, src_mask, tgt_mask)

        return self.linear_out(out)


# ==========================
# Transformer Model (Encoder-Decoder)
# ==========================
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, emb_dim, num_layers, num_head
        super().__init__()

        self.encoder = TransformerEncoder(src_vocab_size, emb_dim, num_layers, num_h
        self.decoder = TransformerDecoder(tgt_vocab_size, emb_dim, num_layers, num_h

    def forward(self, src, tgt, src_mask, tgt_mask):
        encoder_out = self.encoder(src, src_mask)
        output = self.decoder(tgt, encoder_out, src_mask, tgt_mask)
        return output
```