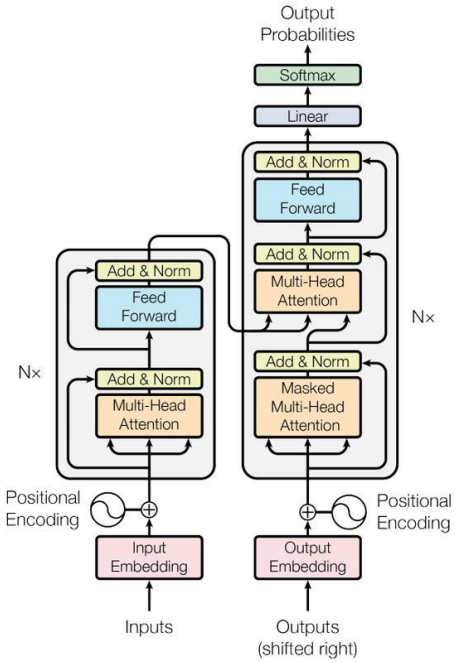
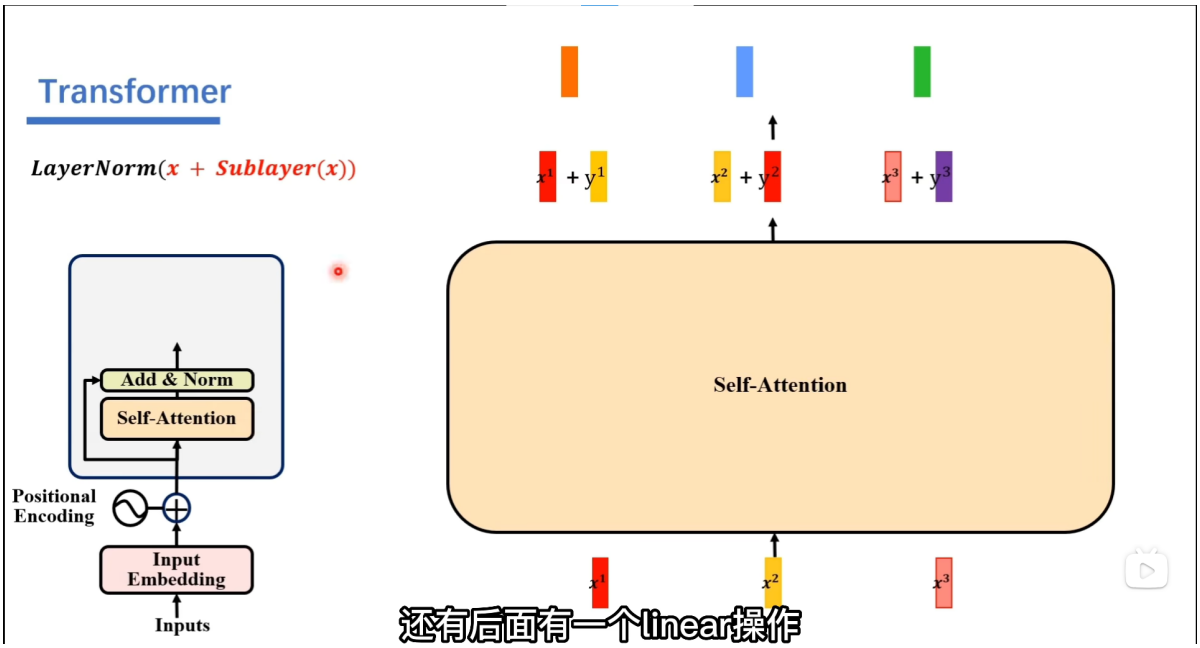


为什么这里还有`Output Embedding`呢因为这里是训练过程，**【就像分类任务，给定一个样本肯定还有它对应的label】**，如果是在机器翻译任务中，这里的input是待翻译的句子，而Output是翻译好的句子。

`Add & Norm`指残差和`layer normlization`层标准化【每经过一个层都需要操作一次】，这里的残差指的是self-attention它的输出向量要和原来的输入向量做一个相加操作，这里就是x_1和y_1相加.

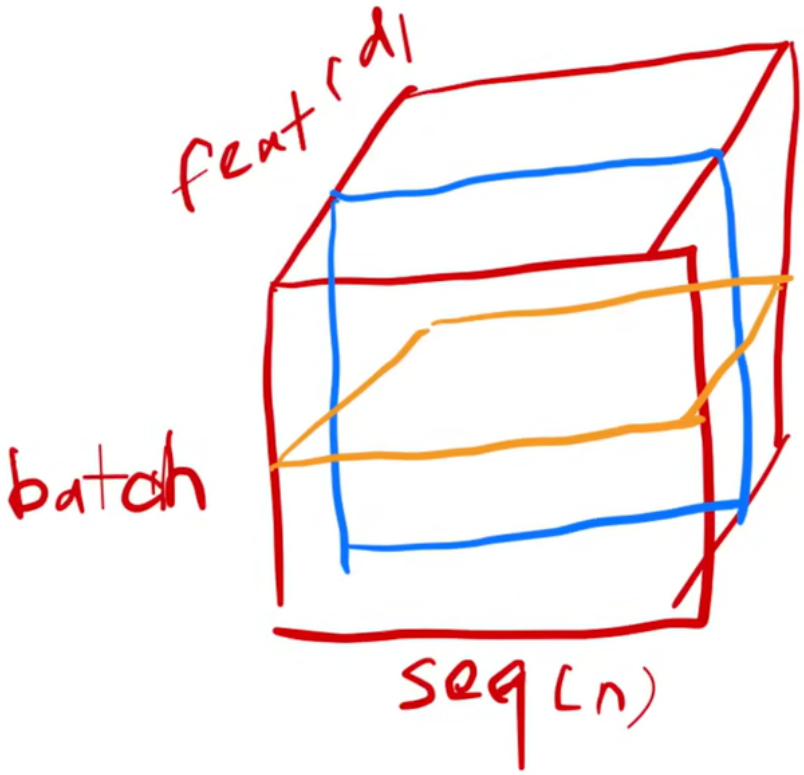
残差连接就是把网络的输入和输出相加，即网络的输出为`F(x)+x`，在网络结构比较深的时候，网络梯度反向传播更新参数时，容易造成梯度消失的问题，但是如果每层的输出都加上一个x的时候，就变成了`F(x)+x`，对x求导结果为**1**，所以就相当于每一层求导时都加上了一个常数项**1**，有效解决了梯度消失问题。



这里做的norm【层标准化】指的就是每个向量各自做一个标准化，也就是所有特征减去一个均值除以一个标准差，针对的是单个向量，即单个样本，transformer用的是`Layernorm`，**layernorm 实际上和下图一致是对每个 token 的 feature 单独求 mean**

$$y = \frac{x - \mathrm{E}[x\ ]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

1. **BatchNorm**：
    - 跨批次对所有样本进行归一化。
    - 在每个特征上独立计算均值和方差。
2. **LayerNorm**：
    - 对每个样本的所有特征进行归一化，不考虑批次中的其他样本。
    - 计算每个样本的均值和方差，然后对每个特征进行归一化。



输入的是三维向量，可以理解为有多句话（batch），每句话中有很多单词【batch中的每句话的长度不一，取最长的作为seq_len】，每个单词都有其词向量表示，所以这里用三维表示

batchnorm在这里是对一整个batch的单个feature去做标准化【这里假定映射为512维，图中表示为蓝线】batchnorm**针对的是特征**

Layernorm是针对一个样本【一个样本中有512个维度】做标准化【图中表示为黄线】Layernorm**针对的是样本**

```
[[w11, w12, w13, w14], [w21, w22, w23, w24], [w31, w32, w33, w34]
[w41, w42, w43, w44], [w51, w52, w53, w54], [w61, w62, w63, w64]]
```

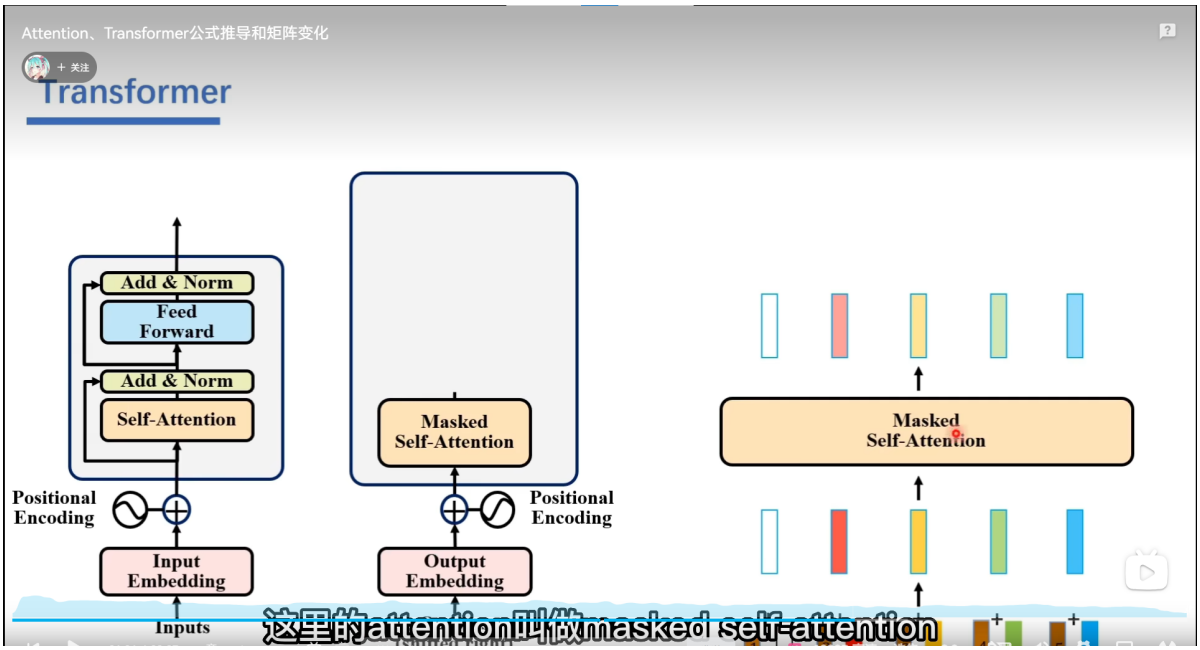
对于一个(2,3,4)的 tensor，`(w11+w12+w13+w14)/4` 是一个 mean，一共会有 2*3=6 个 mean。

```
import torch

batch_size, seq_size, dim = 2, 3, 4
embedding = torch.randn(batch_size, seq_size, dim)

layer_norm = torch.nn.LayerNorm(dim, elementwise_affine = False)
print("y: ", layer_norm(embedding))

eps: float = 0.00001
mean = torch.mean(embedding[:, :, :], dim=(-1), keepdim=True)
var = torch.square(embedding[:, :, :] - mean).mean(dim=(-1), keepdim=True)

print("mean: ", mean.shape)
print("y_custom: ", (embedding[:, :, :] - mean) / torch.sqrt(var + eps))


y:  tensor([[[-0.2500,  1.0848,  0.6808, -1.5156],
         [-1.1630, -0.7052,  1.3840,  0.4843],
         [-1.3510,  0.4520, -0.4354,  1.3345]],

        [[ 0.4372, -0.4610,  1.3527, -1.3290],
         [ 0.2282,  1.3853, -0.2037, -1.4097],
         [-0.9960, -0.6184, -0.0059,  1.6203]]])
mean:  torch.Size([2, 3, 1])
y_custom:  tensor([[[-0.2500,  1.0848,  0.6808, -1.5156],
         [-1.1630, -0.7052,  1.3840,  0.4843],
         [-1.3510,  0.4520, -0.4354,  1.3345]],

        [[ 0.4372, -0.4610,  1.3527, -1.3290],
         [ 0.2282,  1.3853, -0.2037, -1.4097],
         [-0.9960, -0.6184, -0.0059,  1.6203]]])
```

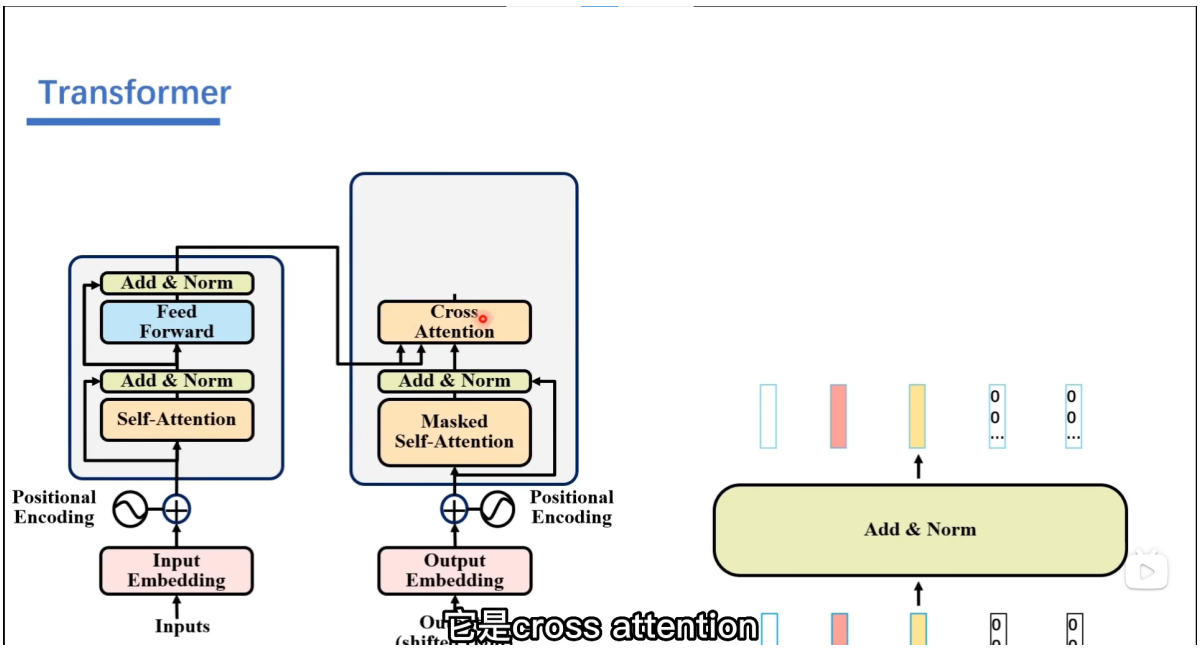
$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \qquad (2\text{-}15)$$

掩码机制

masked self-attention【基于掩码的自注意力层】是为了实现transformer的一大特性：自回归，即前一时间段的输出会作为后一时间段的输入。

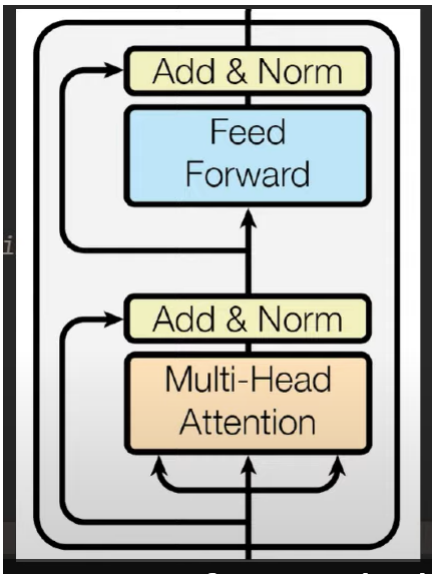


Cross Attention，它用来混合编码器和解码器的信息，Cross Attention会用解码器生成q，去查看编码器端生成的k，一起计算Attention score后，softmax后，将编码器的向量v按权求和得到cross attention的结果



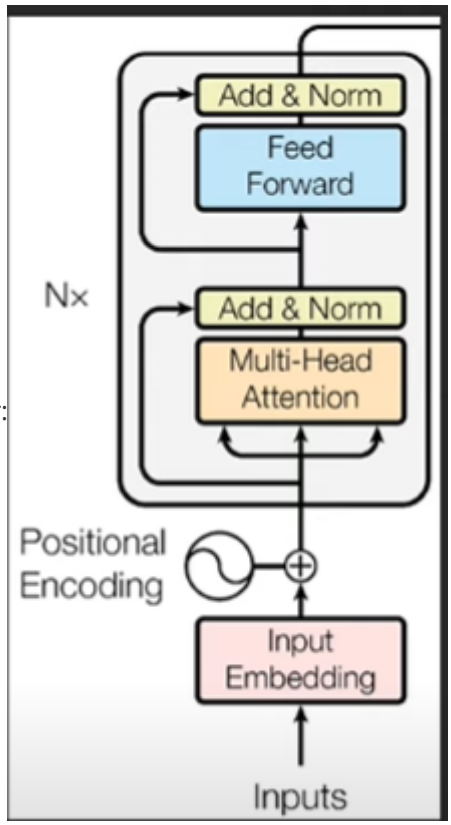
Youtube代码

TransformerBlock:




```
class TransformerBlock(nn.Module): # 输入的size和输出的相同
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion * embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion * embed_size, embed_size),
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, value, key, query, mask):
        attention = self.attention(value, key, query, mask)

        # Add skip connection, run through normalization and finally dropout
        x = self.dropout(self.norm1(attention + query)) # attention + query是残差
连接
        forward = self.feed_forward(x) # 前馈网络
        out = self.dropout(self.norm2(forward + x))
        return out
```

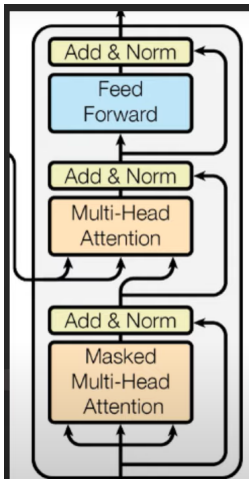
Encoder:




```
class Encoder(nn.Module):
    def __init__(
        self,
        src_vocab_size, # 源词汇表的大小，即输入词汇的总数
        embed_size, # 嵌入层的大小，也是模型中特征的维度
        num_layers, #  编码器中 Transformer 块的数量。
        heads,
        device,
        forward_expansion,
        dropout,
        max_length, # 输入序列的最大长度
    ):

        super(Encoder, self).__init__()
        self.embed_size = embed_size
        self.device = device
        self.word_embedding = nn.Embedding(src_vocab_size, embed_size) #  一个词嵌入层，将输入的单词索引转换为对应的嵌入向量
        self.position_embedding = nn.Embedding(max_length, embed_size) # 一个位置嵌入层，为输入序列的每个位置提供位置信息

        self.layers = nn.ModuleList(
            [
                TransformerBlock(
                    embed_size,
                    heads,
                    dropout=dropout,
                    forward_expansion=forward_expansion,
                )
                for _ in range(num_layers)
            ]
        )
```

```
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        N, seq_length = x.shape
        positions = torch.arange(0, seq_length).expand(N,
seq_length).to(self.device)  # 这里的位置是可以学习的，可以考虑改成正余弦波

        out = self.dropout(
            (self.word_embedding(x) + self.position_embedding(positions))
        )

        # In the Encoder the query, key, value are all the same, it's in the
        # decoder this will change. This might look a bit odd in this case.
        for layer in self.layers: # 在 Transformer编码器中，查询、键和值都是相同的，即前
一层的输出
            out = layer(out, out, out, mask)

        return out
```

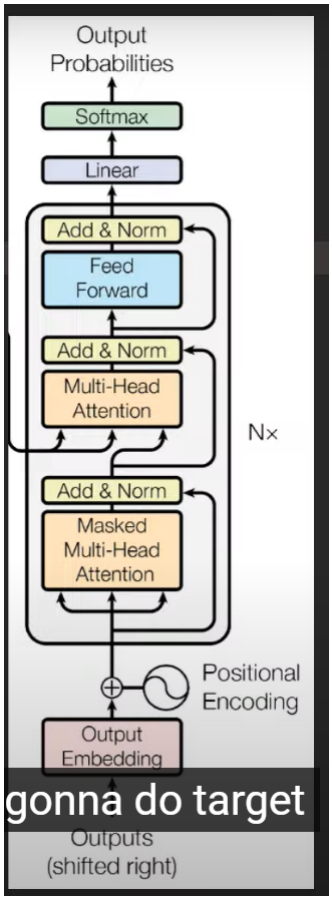
DecoderBlock:




```
class DecoderBlock(nn.Module):
    def __init__(self, embed_size, heads, forward_expansion, dropout, device): #
forward_expansion将输入张量映射到一个更高维度的空间
        super(DecoderBlock, self).__init__()
        self.norm = nn.LayerNorm(embed_size)
        self.attention = SelfAttention(embed_size, heads=heads)
        self.transformer_block = TransformerBlock(
            embed_size, heads, dropout, forward_expansion
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, value, key, src_mask, trg_mask):
        # src_mask: 源掩码，用于在编码器-解码器注意力中屏蔽不相关信息。
        # trg_mask: 目标掩码，用于在自注意力中屏蔽填充（padding）或其他不需要关注的位置
        attention = self.attention(x, x, x, trg_mask)
        # 每个DecoderBlock中，前一步骤的输出（经过自注意力和归一化处理后的结果）将作为查询
（query）输入到下一个TransformerBlock中。
        query = self.dropout(self.norm(attention + x))
        # query 来自解码器，而value和key来自编码器的输入

        # 编码器-解码器之间的注意力机制
```

```
        out = self.transformer_block(value, key, query, src_mask)
        return out
```

Decoder:



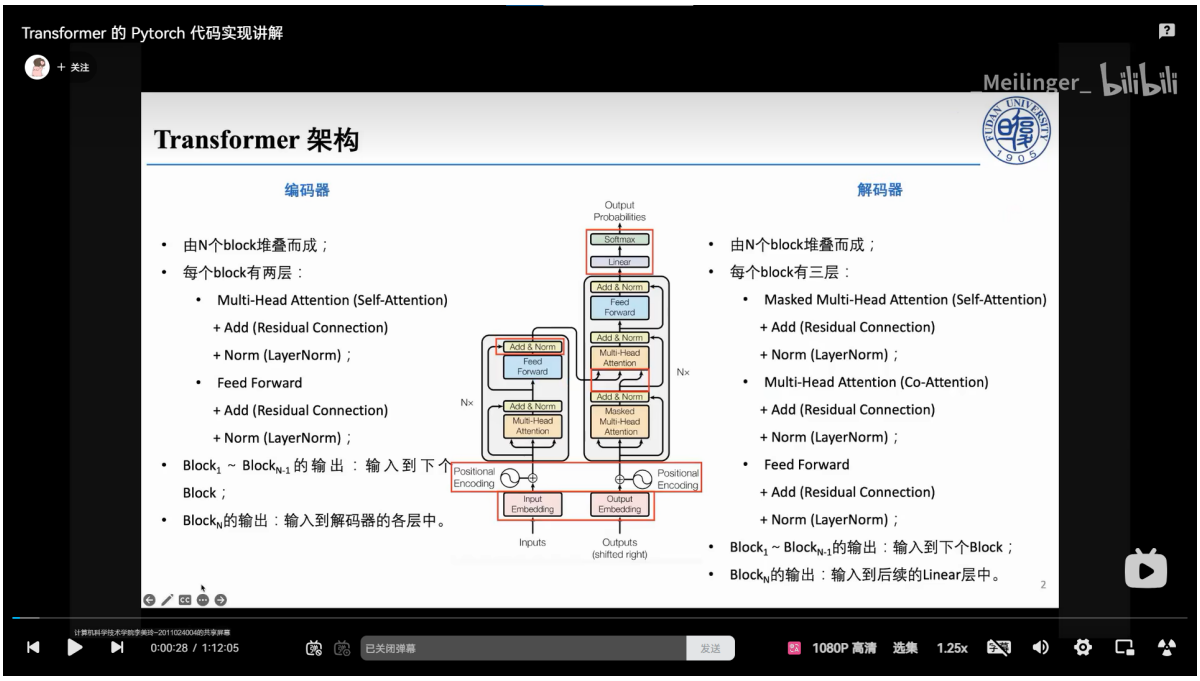

```
class Decoder(nn.Module):
    def __init__(
        self,
        trg_vocab_size,
        embed_size,
        num_layers,
        heads,
        forward_expansion,
        dropout,
        device,
        max_length,
    ):
        super(Decoder, self).__init__()
        self.device = device
        self.word_embedding = nn.Embedding(trg_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList(
            [
                DecoderBlock(embed_size, heads, forward_expansion, dropout,
device)
                for _ in range(num_layers)
            ]
        )
        self.fc_out = nn.Linear(embed_size, trg_vocab_size)
        self.dropout = nn.Dropout(dropout)
```

```
    def forward(self, x, enc_out, src_mask, trg_mask):
        N, seq_length = x.shape
        positions = torch.arange(0, seq_length).expand(N,
seq_length).to(self.device)
        x = self.dropout((self.word_embedding(x) +
self.position_embedding(positions)))

        for layer in self.layers:
            x = layer(x, enc_out, enc_out, src_mask, trg_mask)

        # x:前一解码器层的输出,第二个和第三个enc_out参数分别代表编码器的输出，用作编码器-解码器注意力机制的键（K）和值（V）
        out = self.fc_out(x)
        # 解码器中的每个 DecoderBlock 都包含自注意力和编码器-解码器注意力机制，允许模型在生成序列时考虑内部和外部的上下文信息。
        return out
```

```
class Transformer(nn.Module):
    def __init__(
            self,
            src_vocab_size,  # 源语言的词汇表大小
            trg_vocab_size,  # 目标语言的词汇表大小
            src_pad_idx,  # 源语言的填充索引，用于在序列中标记填充位置
            trg_pad_idx,  # 目标语言的填充索引
            embed_size=512,
            num_layers=6,
            forward_expansion=4,
            heads=8,
            dropout=0,
            device="cpu",
            max_length=100,
    ):
        super(Transformer, self).__init__()

        self.encoder = Encoder(
            src_vocab_size,
            embed_size,
            num_layers,
            heads,
            device,
            forward_expansion,
            dropout,
            max_length,
        )

        self.decoder = Decoder(
            trg_vocab_size,
            embed_size,
            num_layers,
            heads,
```

```
            forward_expansion,
            dropout,
            device,
            max_length,
        )

        self.src_pad_idx = src_pad_idx
        self.trg_pad_idx = trg_pad_idx
        self.device = device

    def make_src_mask(self, src):  # src 是源序列的索引张量
        print(src)
        src_mask = (src != self.src_pad_idx).unsqueeze(1).unsqueeze(2)
        # (N, 1, 1, src_len)
        # (src != self.src_pad_idx)比较源序列xrc中的每个索引，找出不等于填充索引
self.src_pad_idx的位置，结果是一个由布尔值组成的张量
        return src_mask.to(self.device)
        # 返回一个在填充位置为0（False），其他位置为1（True）的掩码张量

    def make_trg_mask(self, trg):  # 为解码器的自注意力机制生成掩码，以防止解码器在生成序
列时看到未来的信息（“掩蔽未来信息”）
        N, trg_len = trg.shape
        trg_mask = torch.tril(torch.ones((trg_len, trg_len))).expand(
            N, 1, trg_len, trg_len
        )
        # torch.tril(torch.ones((trg_len, trg_len))) 创建一个下三角矩阵，对角线和左下部
分为1，右上部分为0。
        # 这个矩阵表示在目标序列中，每个位置只能关注到它之前（包括自身）的位置

        return trg_mask.to(self.device)
        # 返回一个形状为 (batch_size, 1, trg_len, trg_len) 的掩码张量，其中未来的信息位
置为0

        # make_src_mask 确保编码器在处理源序列时忽略填充位置，从而只关注实际的输入数据。
        # make_trg_mask 确保解码器在生成每个词时，只能使用已经生成的词（包括目标序列中的起始
标记）和编码器的输出，而不能利用未来词的信息，这有助于避免在序列生成中产生错误依赖。

    def forward(self, src, trg):
        src_mask = self.make_src_mask(src)
        trg_mask = self.make_trg_mask(trg)
        enc_src = self.encoder(src, src_mask)
        out = self.decoder(trg, enc_src, src_mask, trg_mask)
        # 第一个是目标语言的输入，第二个是经过掩码处理的encoder的输入，第三个参数是
src_mask，第四个参数trg_mask
        return out
```

更加详细的对于transformer架构的解析



$Block_N$的输出【即最后一个Block的输出会输入到解码器的各层中】