```
        # 矩阵转置
        k = k.transpose(1,2)
        q = q.transpose(1,2)
        v = v.transpose(1,2)

        # 计算 attention
        scores = attention(q, k, v, self.d_k, mask, self.dropout)

        # 连接多个头并输入到最后的线性层
        concat = scores.transpose(1,2).contiguous().view(bs, -1, self.d_model)

        output = self.out(concat)

        return output
```

2.1.3 前馈层

前馈层接受自注意力子层的输出作为输入，并通过一个带有 Relu 激活函数的两层全连接网络对输入进行更加复杂的非线性变换。实验证明，这一非线性变换会对模型最终的性能产生十分重要的影响。

$$\mathrm{FFN}(\boldsymbol{x}) = \mathrm{Relu}(\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{2.4}$$

其中 $\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2$ 表示前馈子层的参数。实验结果表明，增大前馈子层隐状态的维度有利于提升最终翻译结果的质量，因此，前馈子层隐状态的维度一般比自注意力子层要大。

使用 Pytorch 实现的前馈层参考代码如下：

```
class FeedForward(nn.Module):

    def __init__(self, d_model, d_ff=2048, dropout = 0.1):
        super().__init__()

        # d_ff 默认设置为 2048
        self.linear_1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear_2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        x = self.dropout(F.relu(self.linear_1(x)))
        x = self.linear_2(x)
```

2.1.4 残差连接与层归一化

由 Transformer 结构组成的网络结构通常都是非常庞大。编码器和解码器均由很多层基本的 Transformer 块组成，每一层当中都包含复杂的非线性映射，这就导致模型的训练比较困难。因此，研究者们在 Transformer 块中进一步引入了残差连接与层归一化技术以进一步提升训练的稳定性。具体来说，残差连接主要是指使用一条直连通道直接将对应子层的输入连接到输出上去，从而避免由于网络过深在优化过程中潜在的梯度消失问题：

$$\boldsymbol{x}^{l+1} = f(\boldsymbol{x}^l) + \boldsymbol{x}^l \tag{2.5}$$

其中 $\boldsymbol{x}^l$ 表示第 l 层的输入，$f(\cdot)$ 表示一个映射函数。此外，为了进一步使得每一层的输入输出范围稳定在一个合理的范围内，层归一化技术被进一步引入每个 Transformer 块的当中：

$$LN(\boldsymbol{x}) = \alpha \cdot \frac{\boldsymbol{x} - \mu}{\sigma} + b \tag{2.6}$$

其中 μ 和 σ 分别表示均值和方差，用于将数据平移缩放到均值为 0，方差为 1 的标准分布，α 和 b 是可学习的参数。层归一化技术可以有效地缓解优化过程中潜在的不稳定、收敛速度慢等问题。

使用 Pytorch 实现的层归一化参考代码如下：

```
class NormLayer(nn.Module):

    def __init__(self, d_model, eps = 1e-6):
        super().__init__()

        self.size = d_model

        # 层归一化包含两个可以学习的参数
        self.alpha = nn.Parameter(torch.ones(self.size))
        self.bias = nn.Parameter(torch.zeros(self.size))

        self.eps = eps

    def forward(self, x):
        norm = self.alpha * (x - x.mean(dim=-1, keepdim=True))  \
        / (x.std(dim=-1, keepdim=True) + self.eps) + self.bias
        return norm
```

2.1.5 编码器和解码器结构

基于上述模块，根据图2.1所给出的网络架构，编码器端可以较为容易实现。相比于编码器端，解码器端要更复杂一些。具体来说，解码器的每个 Transformer 块的第一个自注意力子层额外增加

了注意力掩码，对应图中的**掩码多头注意力**（Masked Multi-Head Attention）部分。这主要是因为在翻译的过程中，编码器端主要用于编码源语言序列的信息，而这个序列是完全已知的，因而编码器仅需要考虑如何融合上下文语义信息即可。而解码端则负责生成目标语言序列，这一生成过程是自回归的，即对于每一个单词的生成过程，仅有当前单词之前的目标语言序列是可以被观测的，因此这一额外增加的掩码是用来掩盖后续的文本信息，以防模型在训练阶段直接看到后续的文本序列进而无法得到有效地训练。

此外，解码器端还额外增加了一个多头注意力（Multi-Head Attention）模块，使用交叉注意力（Cross-attention）方法，同时接收来自编码器端的输出以及当前 Transformer 块的前一个掩码注意力层的输出。查询是通过解码器前一层的输出进行投影的，而键和值是使用编码器的输出进行投影的。它的作用是在翻译的过程当中，为了生成合理的目标语言序列需要观测待翻译的源语言序列是什么。基于上述的编码器和解码器结构，待翻译的源语言文本，首先经过编码器端的每个 Transformer 块对其上下文语义的层层抽象，最终输出每一个源语言单词上下文相关的表示。解码器端以自回归的方式生成目标语言文本，即在每个时间步 t，根据编码器端输出的源语言文本表示，以及前 $t-1$ 个时刻生成的目标语言文本，生成当前时刻的目标语言单词。

使用 Pytorch 实现的编码器参考代码如下：

```
class EncoderLayer(nn.Module):

    def __init__(self, d_model, heads, dropout=0.1):
        super().__init__()
        self.norm_1 = Norm(d_model)
        self.norm_2 = Norm(d_model)
        self.attn = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, dropout=dropout)
        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)

    def forward(self, x, mask):
        x2 = self.norm_1(x)
        x = x + self.dropout_1(self.attn(x2,x2,x2,mask))
        x2 = self.norm_2(x)
        x = x + self.dropout_2(self.ff(x2))
        return x


class Encoder(nn.Module):

    def __init__(self, vocab_size, d_model, N, heads, dropout):
        super().__init__()
        self.N = N
        self.embed = Embedder(vocab_size, d_model)
        self.pe = PositionalEncoder(d_model, dropout=dropout)
        self.layers = get_clones(EncoderLayer(d_model, heads, dropout), N)
        self.norm = Norm(d_model)
```