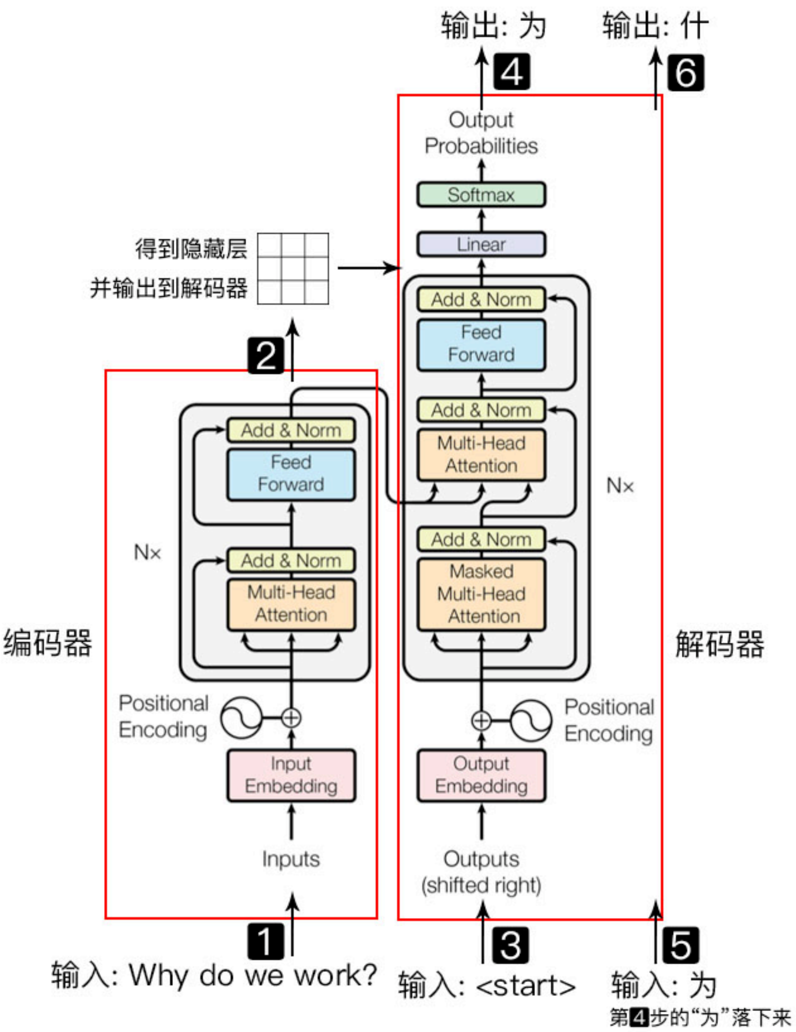
整体架构

过程

编码器负责把自然语言序列映射成为隐藏层， 解码器把隐藏层再映射为自然语言序列。

1. 输入自然语言序列到编码器：Why do we work?（为什么要工作）；
2. 通过编码器得到隐藏层，再输入到解码器；
3. 输入<start>（起始）符号到解码器；
4. 得到第一个字"为";
5. 将得到的第一个字“为”落下来再输入到解码器；
6. 得到第二个字”什";
7. 将得到的第二字再落下来，直到解码器输出*<end>*（终止符），即序列生成完成。



Transformer

```
class Transformer(nn.Module):
    # model_dim: Embedding Size, ffn_dim: FeedForward dimension
```

```python
    # n_layers: number of Encoder of Decoder Layer, n_heads: number of heads in Multi-Head Attention
    def __init__(self, src_vocab_size, src_max_len, tgt_vocab_size, tgt_max_len,
                 num_layers=6, num_heads=8, model_dim=512, ffn_dim=2048, dropout=0.0):
        super(Transformer, self).__init__()

        self.encoder = Encoder(src_vocab_size, src_max_len, num_layers, model_dim, num_heads, ffn_dim, dropout)
        self.decoder = Decoder(tgt_vocab_size, tgt_max_len, num_layers, model_dim, num_heads, ffn_dim, dropout)
        self.linear = nn.Linear(model_dim, tgt_vocab_size, bias=False)
        self.softmax = nn.Softmax(dim=2)

    def forward(self, src_seq, tgt_seq):
        src_mask = get_pad_mask(src_seq).int()
        tgt_mask = torch.gt((get_pad_mask(tgt_seq).int() + get_sequence_mask(tgt_seq).int()), 0).int()

        enc_output, enc_self_attn = self.encoder(src_seq, src_mask)
        dec_output, dec_self_attn, dec_enc_attn = self.decoder(tgt_seq, tgt_mask, enc_output, src_mask)
        output = self.softmax(self.linear(dec_output))

        return output, enc_self_attn, dec_self_attn, dec_enc_attn
```

Mask

Transformer 模型里面涉及两种mask，分别是 padding mask 和 sequence mask。

padding mask 由于对输入设置了最大的输入长度（max sequence length），因此对短于最大长度的输入序列会在其后面填充0。但是这些填充的位置并没无意义，attention 机制不应该把注意力放在这些位置上，因此可以把 $mask = 1$ 对应的输入位置加上一个非常大的负数（或负无穷），这样经过 softmax 这些位置的概率就会接近0。

```python
def get_pad_mask(seq):
    mask = (seq == 0).unsqueeze(-2)
    return mask
```

sequence mask sequence mask 是为了使 decoder 不能看到未来的信息。对于一个序列，在 t 时刻解码输出只能依赖于 t 时刻之前的输出，而不能依赖 t之后的输出。可以构造一个矩阵，上三角的值全为1，下三角的值全为0，对角线也是0，值为1的位置即是加上负无穷的位置。

```python
def get_sequence_mask(seq):
    batch_size, seq_len = seq.size()
    mask = torch.triu(torch.ones((seq_len, seq_len), dtype=torch.uint8), diagonal=1)
    mask = mask.unsqueeze(0).expand(batch_size, -1, -1)

    return mask
```

Positional Encoding

位置编码用于提供语言的位置顺序信息，论文使用了 sin 和 cos 函数的变换：

$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$

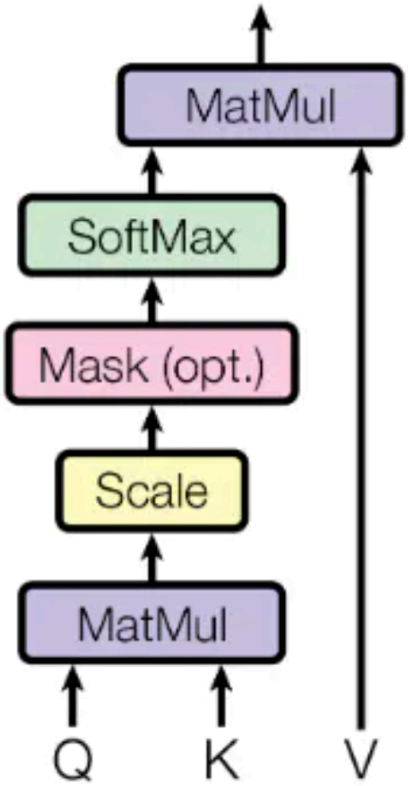
$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$

位置嵌入的维度为 $(max\ sequence\ length, emdedding\ size)$，$pos$ 指的是句中字的位置，取值范围是$[0, max\ sequence\ length)$，$i$指的是词向量的维度，取值范围是 $[0, embedding\ size)$.

```python
class PositionalEncoding(nn.Module):
    def __init__(self, max_seq_length, embed_size):
        super(PositionalEncoding, self).__init__()

        pe = np.array([[(pos / np.power(10000, 2 * (i // 2) / embed_size))
                        for i in range(d_model)] for pos in range(max_seq_length)])

        pe[:, 0::2] = np.sin(pe[:, 0::2])
        pe[:, 1::2] = np.cos(pe[:, 1::2])
        pe = torch.FloatTensor(pe).unsqueeze(0)  # (1, max_seq_length, d_model)
        self.register_buffer("pe", pe)

    def forward(self, x):
        # x: input--(batch size, max squence length)
        # (batch size, max seq length, emdedding size)
        return self.pe[:, :x.size(1), :].clone().detach()
```

Mutil-Head Attention

Scaled Dot-Product Attention



$X_{embedding} = EmbeddingLookup(X) + PositionalEncoding(X)$，其维度为 $(batch\ size, max\ sequence\ length, embedding\ size)$.

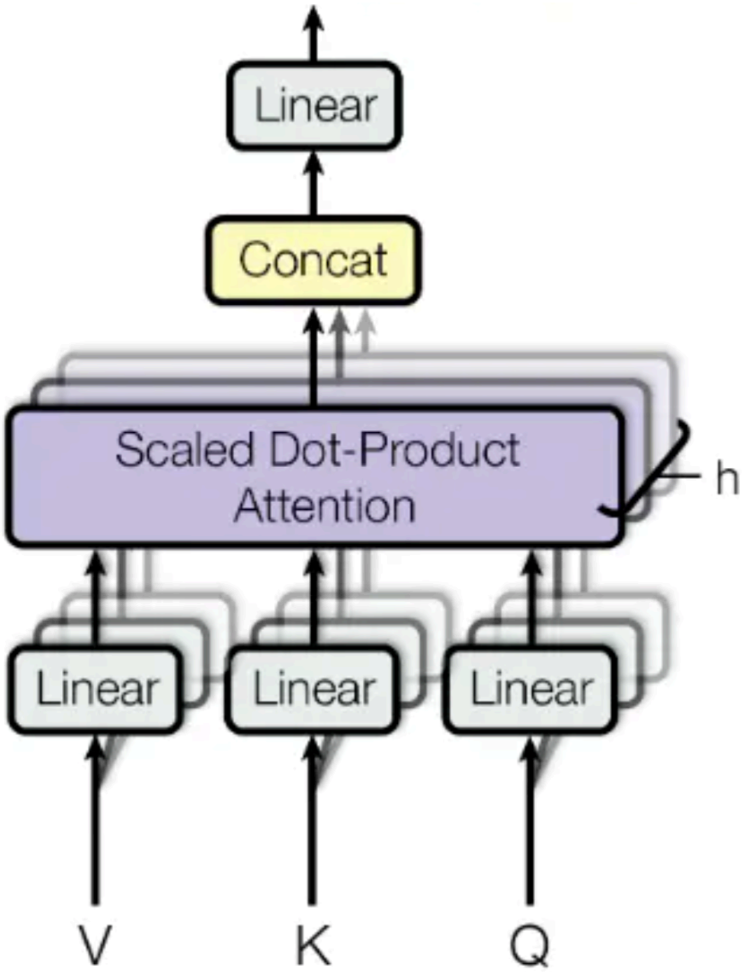
$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$

- 在 encoder 的 self-attention 中，Q、K、V 都来自上一层 encoder 的输出。对于第一层 encoder，其输入为$X_{embedding}$.
- 在 decoder 的 self-attention 中，Q、K、V 都来自上一层 decoder 的输出。对于第一层 decoder，其输入为$X_{embedding}$，但是对于decoder，不希望它能获得下一个时间步的信息，因此需要进行 sequence masking.
- 在 encoder-decoder attention 中，Q 来自 decoder 的上一层输出，K、V 来自 encoder 的输出.

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self, dropout=0.2):
        super(ScaledDotProductAttention, self).__init__()

        self.softmax = nn.Softmax(dim=2)
        self.dropout = nn.Dropout(dropout)

    def forward(self, q, k, v, scale=None, mask=None):
        # q,k,v (batch size, max seq length, embedding size)
        # attention (batch size, max seq length, max seq length)
        attention = torch.bmm(q, k.transpose(1, 2))
        if scale:
            attention = attention * scale
        if mask is not None:
            attention = attention.masked_fill_(mask.bool(), -np.inf)
```

```python
        attention = self.dropout(self.softmax(attention))
        # context (batch size, max seq length, embedding size)
        context = torch.bmm(attention, v)

        return context, attention
```

Mutil-Head Attention



$Q = Linear(X_{embedding}) = X_{embedding}W_Q$

$K = Linear(X_{embedding}) = X_{embedding}W_K$

$V = Linear(X_{embedding}) = X_{embedding}W_V$

$MutilHead(Q, K, V) = Concat(head_1, \ldots head_n)W_O$

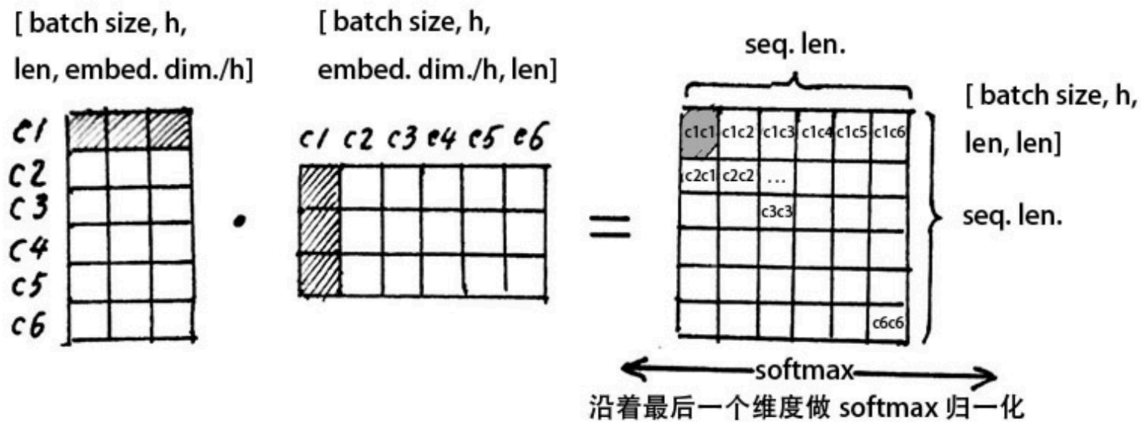
Q、K、V $(batch\ size, max\ sequence\ length, embedding\ size)$，$W_Q, W_K, W_V, W_O$ $(embedding\ size, embedding\ size)$.

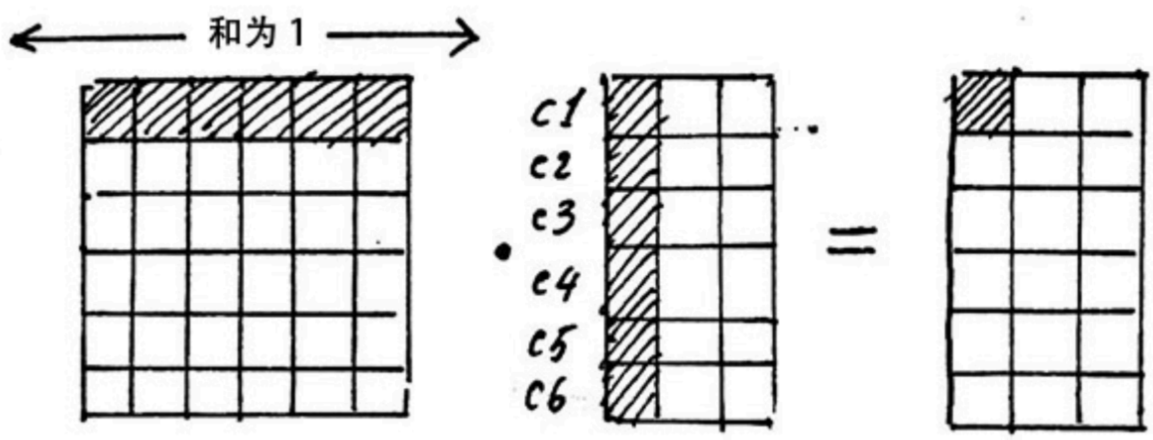
将 Q、K、V 通过一个线性映射之后把 $embedding\ size$分成 h 份，对每一份进行 Scaled Dot-Product Attention。然后，把各个部分的结果合并起来再经过线性映射，得到最终的输出。

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, model_dim=512, num_heads=8, dropout=0.2):
        super(MultiHeadAttention, self).__init__()

```

```python
        self.dim_per_head = model_dim // num_heads
        self.num_heads = num_heads
        self.linear_q = nn.Linear(model_dim, self.dim_per_head * num_heads)
        self.linear_k = nn.Linear(model_dim, self.dim_per_head * num_heads)
        self.linear_v = nn.Linear(model_dim, self.dim_per_head * num_heads)
        self.attention = ScaledDotProductAttention(dropout)
        self.linear = nn.Linear(model_dim, model_dim)
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(model_dim)

    def forward(self, q, k, v, mask=None):
        residual = q

        dim_per_head = self.dim_per_head
        num_heads = self.num_heads
        batch_size = q.size(0)

        # linear projection
        q = self.linear_q(q)
        k = self.linear_k(k)
        v = self.linear_v(v)

        # split by num_heads
        q = q.view(batch_size * num_heads, -1, dim_per_head)
        k = k.view(batch_size * num_heads, -1, dim_per_head)
        v = v.view(batch_size * num_heads, -1, dim_per_head)

        if mask is not None:
            mask = mask.repeat(num_heads, 1, 1)

        scale = (q.size(-1) // num_heads) ** -0.5
        context, attention = self.attention(q, k, v, scale, mask)

        # concat heads
        context = context.view(batch_size, -1, dim_per_head * num_heads)

        output = self.dropout(self.linear(context))
        output = self.layer_norm(residual + output)

        return output, attention
```

Attention 的含义





$softmax(\frac{QK^T}{\sqrt{d_k}})V$

先计算Q、K 的点积，两个向量越相似，其点积越大。比如第一个字与第一个字的点积，即$c1$行与$c1$列相乘得到的$c1c1$，代表了第一个字的注意力机制，注意力矩阵的第一行指的是第一个字与所有6个字的相似度。

然后再点乘V的列，从而使得每个字向量都含有当前句子所有字向量的全部信息。

Position-wise FeedForward Network

这是一个全连接网络，包含两个线性变换和一个非线性函数（ReLU）。

$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$

输入x $(batch\ size, max\ sequence\ length, model\ dim)$

W_1 $(model\ dim, ffn\ dim)$

$W2$ $(ffn\ dim, model\ dim)$

输出 $(batch\ size, max\ sequence\ length, model\ dim)$

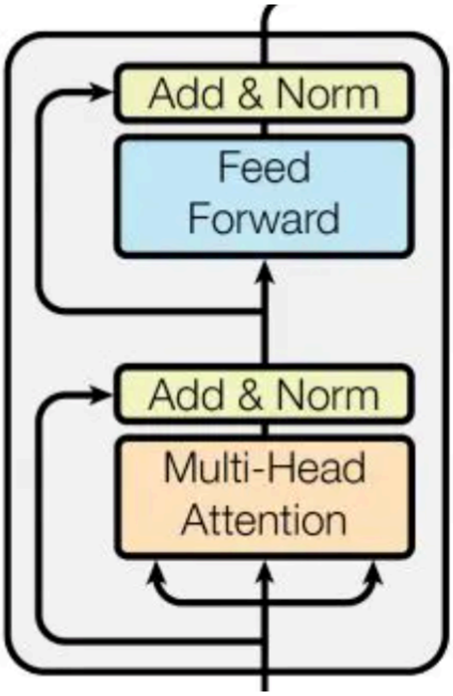
```
class PositionalWiseFeedForward(nn.Module):
    def __init__(self, model_dim=512, ffn_dim=2048, dropout=0.2):
        super(PositionalWiseFeedForward, self).__init__()

        self.linear1 = nn.Linear(model_dim, ffn_dim)
```

```python
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(ffn_dim, model_dim)
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(model_dim)

    def forward(self, x):
        residual = x

        x = self.relu(self.linear1(x))
        x = self.dropout(self.linear2(x))
        output = self.layer_norm(x + residual)

        return output
```

Encoder

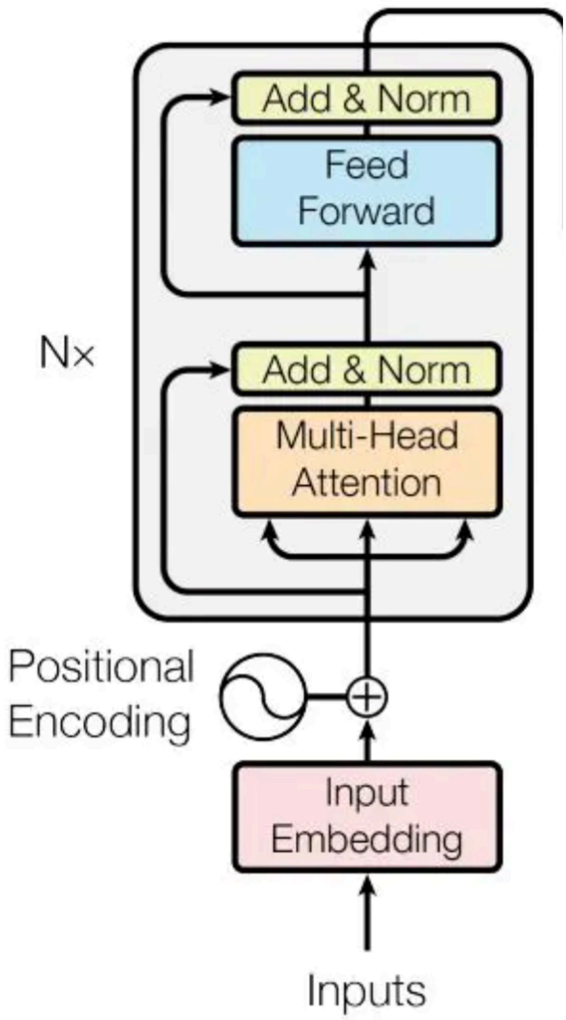
EncoderLayer

EncoderLayer 的每一层都由两部分组成：第一部分是 Multi-Head Self-Attention Mechanism，第二部分是 Position-Wise Feed Forward Network，这两部分都有一个 Residual Connection 和 Layer Normalization.



```python
class EncoderLayer(nn.Module):
    def __init__(self, model_dim=512, num_heads=8, ffn_dim=2048,
dropout=0.2):
        super(EncoderLayer, self).__init__()

        self.attention = MultiHeadAttention(model_dim, num_heads, dropout)
        self.feed_forward = PositionalWiseFeedForward(model_dim, ffn_dim,
dropout)

    def forward(self, enc_inputs, enc_self_attn_mask=None):
        context, attention = self.attention(enc_inputs, enc_inputs,
enc_inputs, enc_self_attn_mask)
        output = self.feed_forward(context)

        return output, attention
```

Encoder

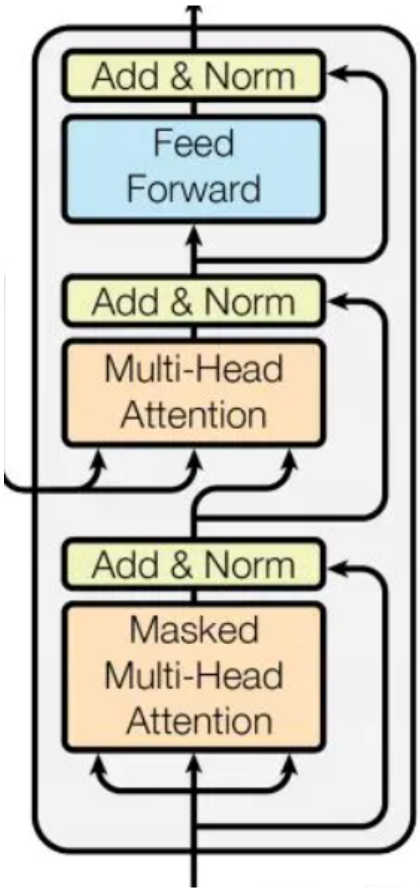



```python
class Encoder(nn.Module):
    def __init__(self, vocab_size, max_seq_length, num_layers=6,
model_dim=512,
```

```python
                  num_heads=8, ffn_dim=2048, dropout=0.2):
        super(Encoder, self).__init__()

        self.word_embed = nn.Embedding(vocab_size, model_dim,
padding_idx=0)
        self.pos_embed = PositionalEncoding(max_seq_length, model_dim)
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(model_dim)
        self.layer_stack = nn.ModuleList([EncoderLayer(model_dim,
num_heads, ffn_dim, dropout)
                                          for _ in range(num_layers)])

    def forward(self, inputs, slf_attn_mask):
        embed = self.word_embed(inputs) + self.pos_embed(inputs)
        output = self.layer_norm(self.dropout(embed))

        attentions = []
        for encoder in self.layer_stack:
            output, attention = encoder(output, slf_attn_mask)
            attentions.append(attention)

        return output, attentions
```

Decoder

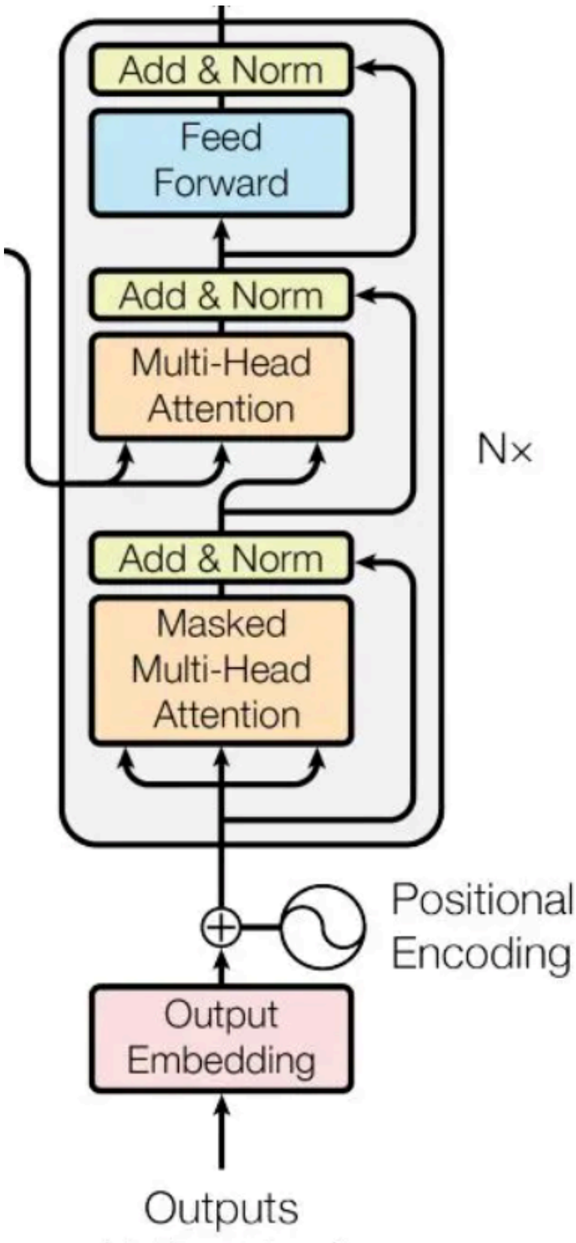
DecoderLayer



DecoderLayer 每一个层包括以下3个部分：第一个部分是 multi-head self-attention mechanism，第二部分是 multi-head context-attention mechanism，第三部分是 position-wise feed-forward network，这三个部分的每一个部分都有一个 residual connection 和 Layer Normalization.

```
class DecoderLayer(nn.Module):
    def __init__(self, model_dim=512, num_heads=8, ffn_dim=2048, dropout=0.2):
        super(DecoderLayer, self).__init__()

        self.self_attn = MultiHeadAttention(model_dim, num_heads, dropout)
        self.dec_enc_attn = MultiHeadAttention(model_dim, num_heads, dropout)
        self.feed_forward = PositionalWiseFeedForward(model_dim, ffn_dim, dropout)

    def forward(self, dec_inputs, enc_outputs, slf_attn_mask=None, dec_enc_attn_mask=None):
        # self attention: all inputs are decoder inputs
        dec_outputs, self_attn = self.self_attn(dec_inputs, dec_inputs, dec_inputs, slf_attn_mask)

        # context attention: query is decoder's outputs, key and value are encoder's inputs
        dec_outputs, dec_enc_attn = self.dec_enc_attn(dec_outputs, enc_outputs, enc_outputs, dec_enc_attn_mask)

        dec_outputs = self.feed_forward(dec_outputs)

        return dec_outputs, self_attn, dec_enc_attn
```

Decoder




```python
class Decoder(nn.Module):
    def __init__(self, vocab_size, max_seq_length, num_layers=6, model_dim=512,
                 num_heads=8, ffn_dim=2048, dropout=0.2):
        super(Decoder, self).__init__()

        self.word_embed = nn.Embedding(vocab_size, model_dim, padding_idx=0)
        self.pos_embed = PositionalEncoding(max_seq_length, model_dim)
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(model_dim)
        self.layer_stack = nn.ModuleList([DecoderLayer(model_dim, num_heads, ffn_dim, dropout)
                                          for _ in range(num_layers)])

    def forward(self, tgt_seq, tgt_mask, enc_outputs, src_mask):
        embed = self.word_embed(tgt_seq) + self.pos_embed(tgt_seq)
```

```
        dec_outputs = self.layer_norm(self.dropout(embed))

        self_attns, dec_enc_attns = [], []
        for decoder in self.layer_stack:
            dec_outputs, self_attn, dec_enc_attn = decoder(dec_outputs, enc_outputs, slf_attn_mask=tgt_mask, dec_enc_attn_mask=src_mask)
            self_attns.append(self_attn)
            dec_enc_attns.append(dec_enc_attn)

        return dec_outputs, self_attns, dec_enc_attns
```

参考资料

https://juejin.im/post/5b9f1af0e51d450e425eb32d#heading-13

https://github.com/jadore801120/attention-is-all-you-need-pytorch