Transformer

- **基本背景**

Transformer由论文《Attention is All You Need》提出，现在是谷歌云TPU推荐的参考模型。论文相关的Tensorflow的代码可以从GitHub获取，其作为Tensor2Tensor包的一部分。哈佛的NLP团队也实现了一个基于PyTorch的版本，并注释该论文。

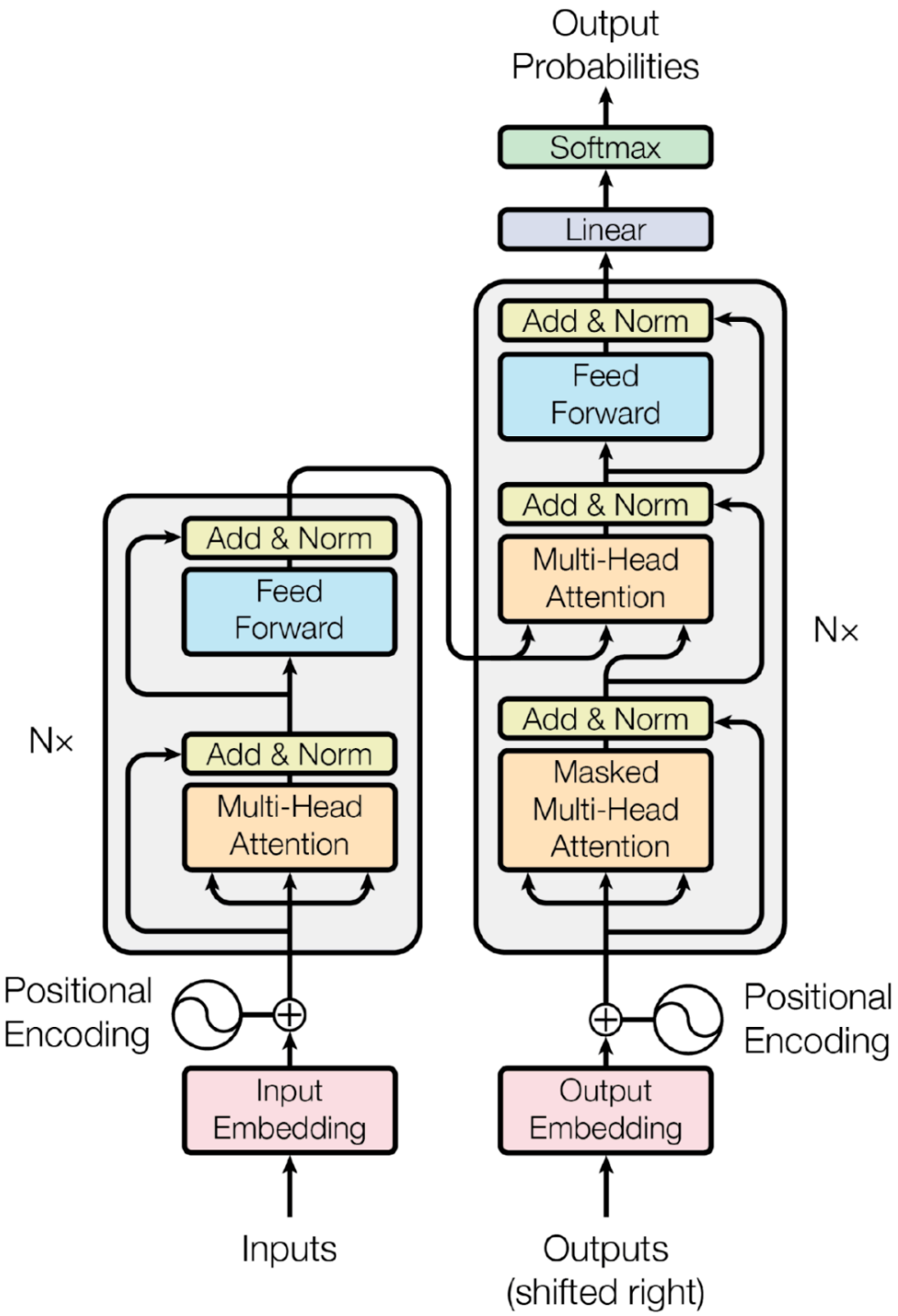
Transformer解决了RNN**无法并行计算**， **长距离信息遗忘**的问题, 并且**彻底摒弃RNN, CNN**结构来计算，转而使用**Attention** 注意力机制来解决问题。

1. Transformer 整体结构与工作流程

1.1 Transformer的整体结构

首先介绍Transformer的整体结构，传统的 Transformer是 `Encoder-Decoder` 结构。

下图是Transformer的总体结构 (**重要**) :



可以看到 **Transformer由Encoder(左)和Decoder(右)两个部分** 组成。

- Encoder（编码器）：输入序列特征编码
  - **前置处理**：词嵌入**token-embedding** 和 位置编码 **position-embedding**
  - **单层结构**（TransformerBlock）（**注意: 顺序执行**）：
    - 多头自注意力(MultiHeadAttention) 层（捕捉输入序列内部 token 依赖）
    - 第一次 Add & Norm（残差连接 + 层归一化, 目的: **保留原始信息， 稳定训练过程。**）
    - 前馈神经网络（FFN）：独立非线性变换 token 向量

        - 第二次 Add & Norm （残差连接 + 层归一化, 目的: **保留原始信息， 稳定训练过程。**）
    - **输出**：融合了序列内部依赖关系 + 非线性变换特征、分布稳定的 token， 维度保持不变的序列。

- Decoder（解码器）：目标序列自回归生成
    - **前置处理**：词嵌入**token-embedding** 和 位置编码 **position-embedding**，并加 **attention-mask**。
    - **单层结构**（TransformerBlock)（**注意: 顺序执行**）：
        - 多头自注意力(MultiHeadAttention) 层（捕捉输入序列内部 token 依赖）
        - 第一次 Add & Norm（残差连接 + 层归一化, 目的: **保留原始信息， 稳定训练过程。**）
        - 编码器 - 解码器注意力层：Query 来自 Decoder，Key/Value 来自 Encoder，关联输入与目标序列
        - 第二次 Add & Norm（残差连接 + 层归一化, 目的: **保留原始信息， 稳定训练过程。**）
        - FeedForward层 （进一步提取和增强 token 的深层特征）
        - 第三次 Add & Norm（残差连接 + 层归一化, 目的: **保留原始信息， 稳定训练过程。**）
    - **输出**：经**线性层 + Softmax**，输出目标 token 的概率分布，**逐 token 生成结果**。

1.2 Transformer的工作流程

Transformer的工作流程如下(**重要**)：

- 第一步 **嵌入 （Embedding）**

  分为 **词嵌入** 和 **位置编码** 两个子步骤。

    - **词嵌入**(`token-embedding`) 如何得到

      单词的 `Embedding` 有很多种方式可以获取，例如可以**采用** `Word2Vec`、`One-hot`、`Glove` 等算法预训练得到，也可以 **在** `Transformer` **中训练** 得到。

    - **位置编码** (`position-embedding`)

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d})$$

  其中，pos 表示单词在句子中的位置，d 表示 PE 的维度，$2i$ 表示偶数的维度，$2i+1$ 表示奇数维度。

  即$2i \leq d, 2i+1 \leq d$ 。(注意: PE 就是位置编码 `position-embedding`，且 $d_{PE} = d_{TE}$)

  使用这种公式计算 PE 有以下的好处：

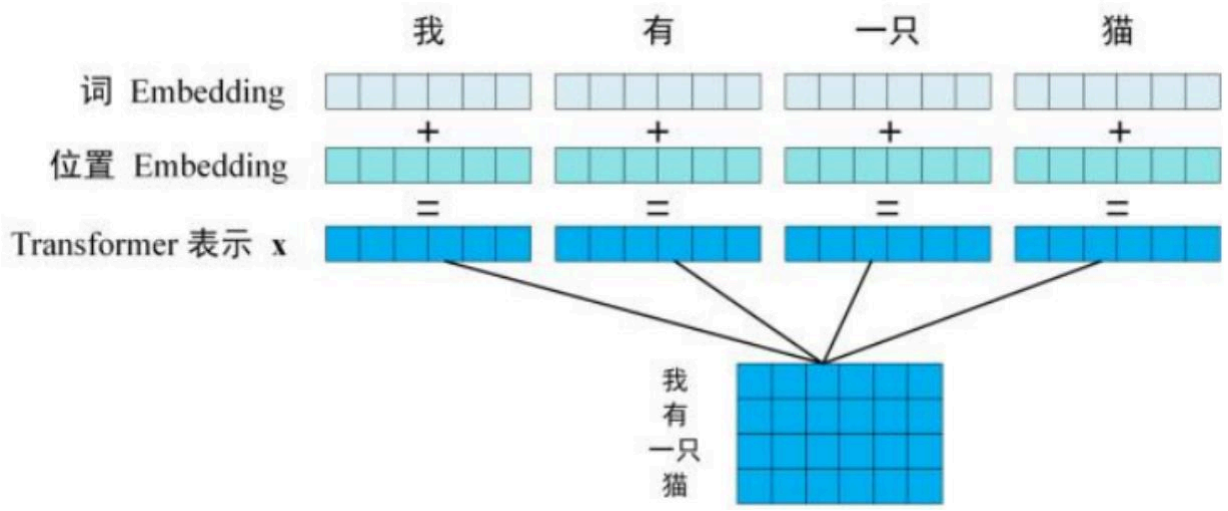
    1. **使位置向量能够适应比训练集里面所有句子更长的句子。**

       假设训练集里面最长的句子长度是20个token, 突然来了一个token长度为为21的句子, 则使用公式计算的方法可以计算出第 21 位的 Embedding。)
    2. **可以让模型容易地计算出相对位置。**

对于固定长度的间距 k，$PE(pos+k)$ 可以用 $PE(pos)$ 计算得到。

因为底层逻辑可以使用 **正弦余弦和差角公式**。

`position-embedding` 和 `token-embedding` 相加 (二者维度一致) 之后得到 **Transformer表示**X。



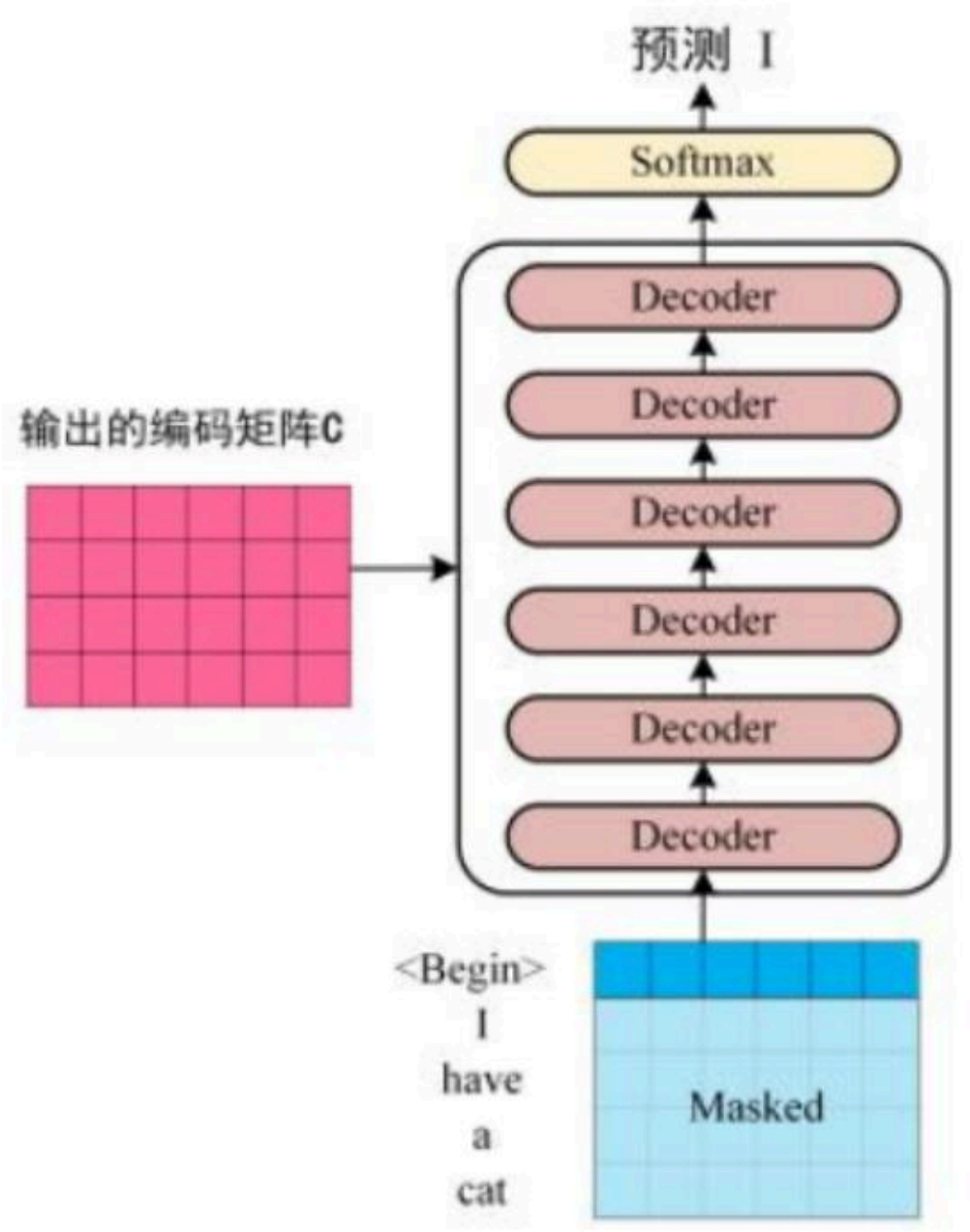
由于X 由 **Token** 的`Embedding`和它 **Position** 的`Embedding`**相加得到**，因此 X 包含了每个 **Token** 在**文本中的位置信息**和**词原本的意思**。

- 第二步 **编码**（**Encoding**）

将得到的单词表示向量矩阵X (如上图所示) 传入`Encoder`中，经过6个`Encoder`后可以得到句子所有单词 的编码信息矩阵 C，如下图。单词向量矩阵用 $X_n \cdot d$ 表示，n 是句子中单词个数，d 是表示向量的维度(论文中 $d = 512$)。每一个 `Encoder` 输出的矩阵维度与输入完全一致。

- 第三步 **解码** （**Decoding**）

  这一步将`Encoder`输出的编码信息矩阵 C 传递到 `Decoder` 中，`Decoder` 依次会根据当前预测过的单词预测下一个。



重点: 在使用的过程中，翻译到单词 i 的时候需要通过 `mask` (掩盖) 操作遮盖住 $i+1$ 之后的单词。

例如上图, `Decoder` 接收了 `Encoder` 的编码矩阵 C，然后首先输入一个翻译开始符 `""`，预测第一个单词 `"I"`；然后输入翻译开始符 `""` 和单词 `"I"`，预测单词 `"have"`，以此类推。这是 `Transformer` 使用时候的大致流程，接下来是里面各个部分的细节。

`Encoder` 和 `Decoder` 的工作过程中大量涉及到 `Self-Attention` （**自注意力机制**） 的概念, 接下来就要介绍。

2. Self-Attention (自注意力机制)

我们重点关注 `Multi-Head Attention` 以及 `Self-Attention` (`SingleHeadAttention`);

首先详细了解一下 `Self-Attention` 的内部逻辑。

2.1 Self-Attention 结构

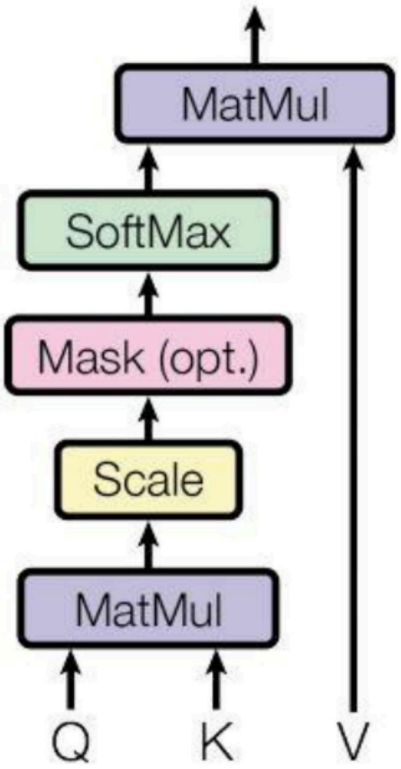
下图是 `Self-Attention` 的结构，在计算的时候需要用到矩阵 Q (查询), K (键), V (值)。

在实际中，`Self-Attention` 接收的是 **输入** 。

输入的组成:

- **单词的表示向量组成的矩阵 X**
- **上一个 Encoder block 的输出。**

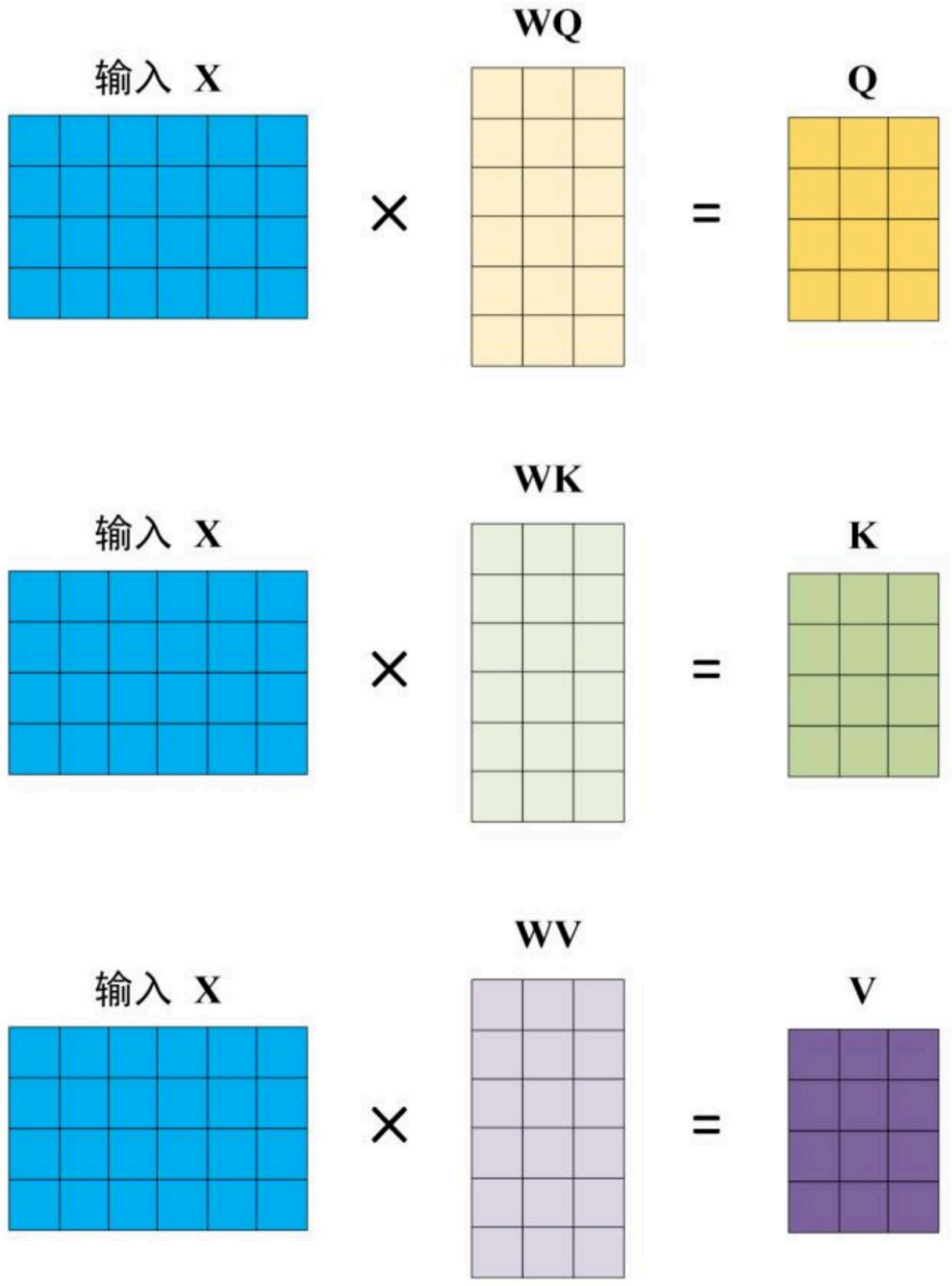
而 Q, K, V 正是通过 Self-Attention 的**输入**进行**线性变换**得到的。



2.2 Q, K, V 的计算

若 `Self-Attention` 的输入用矩阵 X 进行表示，则：

可以用线性变阵矩阵 W_Q, W_K, W_V 计算得到 Q, K, V。计算如下图所示：



(注意 X, Q, K, V 的每一行都表示一个 **token**。 X, Q, K, V本身表示的是 **token序列**。)

2.3 Self-Attention 的输出

得到矩阵 Q, K, V 之后就可以计算出 `Self-Attention` 的输出了，计算的公式如下：

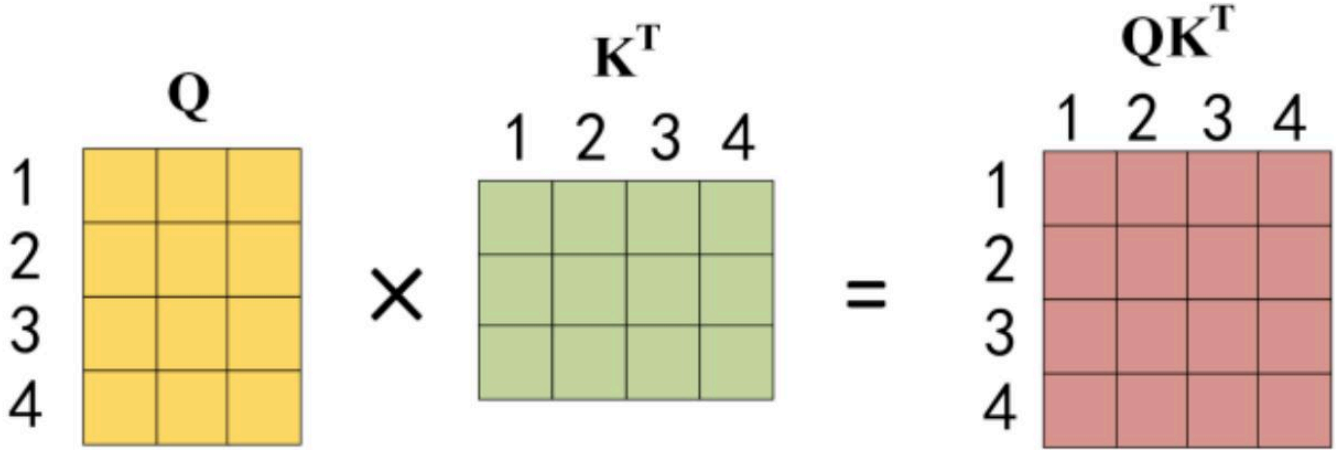
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

d_k 是 Q, K 矩阵的列数，即向量维度。

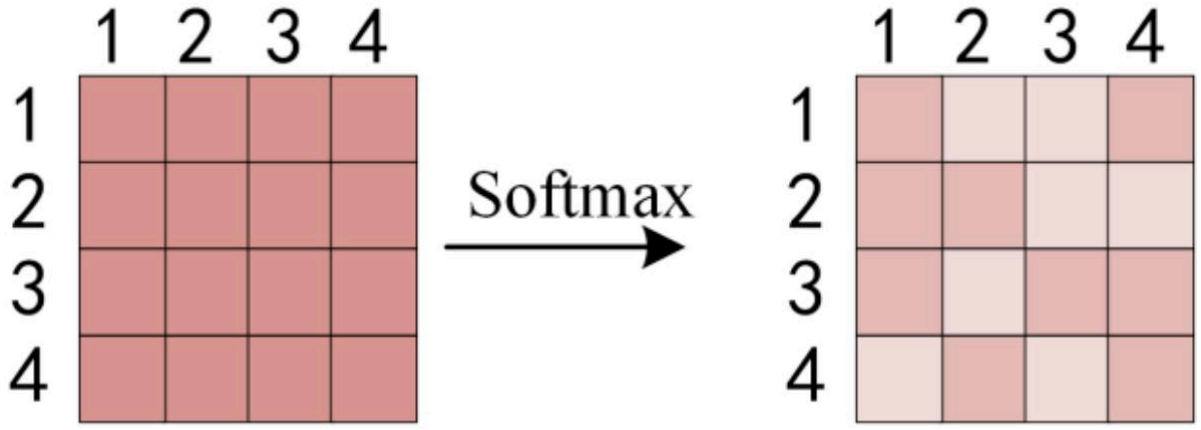
- $\sqrt{d_k}$ 是 **缩放因子**, 除以它可以防止内积的结果过大，进而 **防止计算过程中出现梯度爆炸或梯度消失**。

- $softmax$ 的作用是**将注意力得分转化为合法的概率分布**，以此**量化不同 token 对当前 token 的重要程度**。

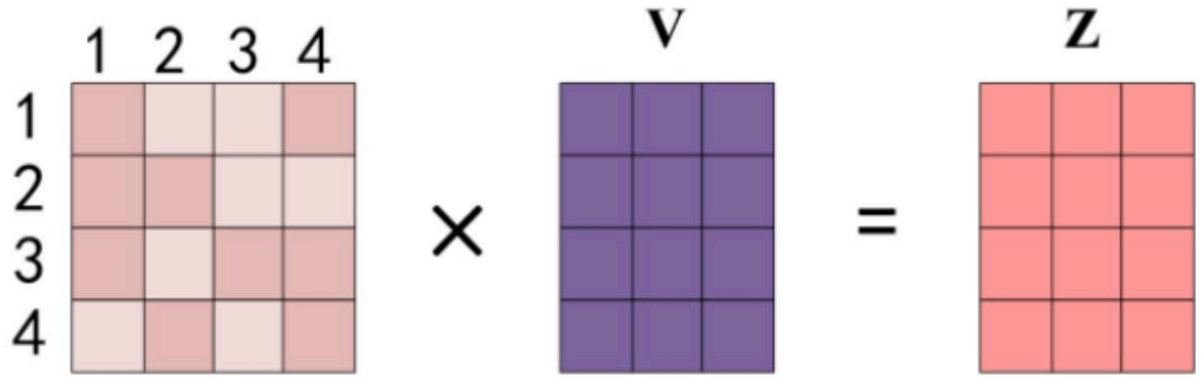
如下图, Q点乘K^T后，得到的矩阵行列数都为 n (n为组成文本token长度， 下图中 $n=4$)，这个矩阵可以表示token之间的attention强度。下图为Q乘以K^T，1234列表示的是句子中的token。



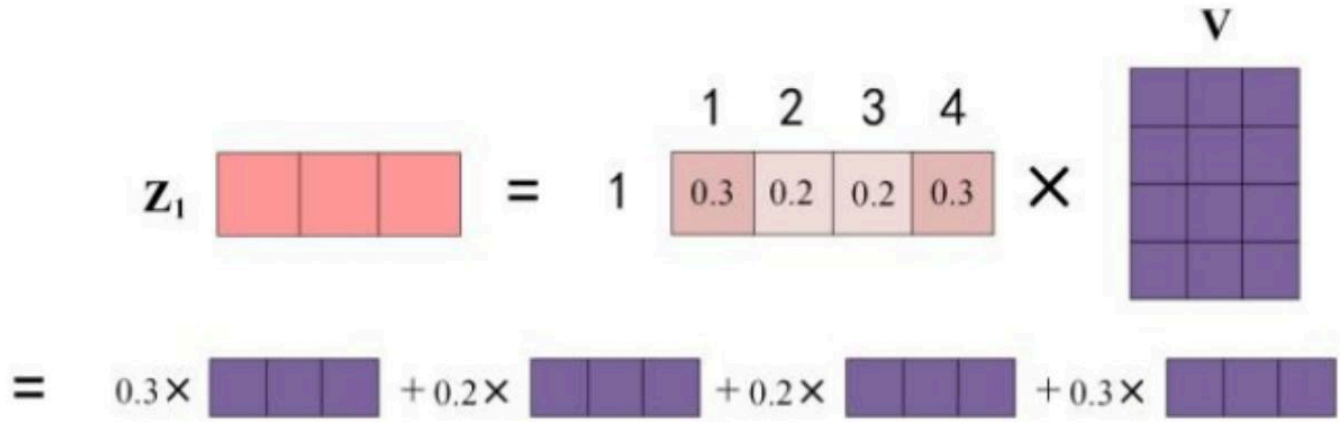
得到 QK^T 之后，还要除以**缩放因子**$\sqrt{d_k}$, 然后使用 `softmax` 映射，将每一行的和都变为1。



最后和 V 相乘，得到最终的输出 Z 。



计算示例：



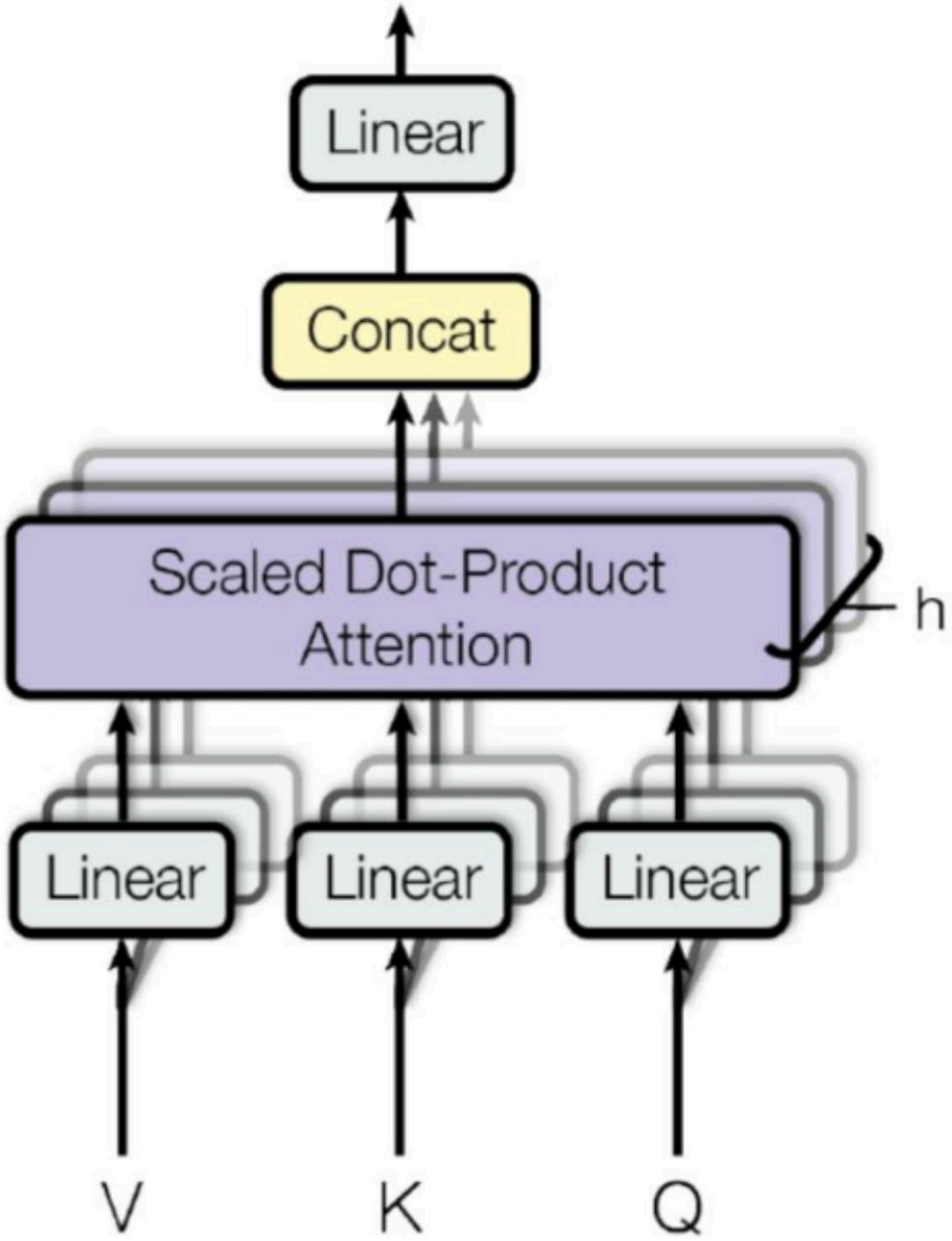
实现代码如下:

```
class EncoderSingleHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.head_size = config.head_size
        self.key = nn.Linear(config.n_embd, config.head_size)
        self.query = nn.Linear(config.n_embd, config.head_size)
        self.value = nn.Linear(config.n_embd, config.head_size)
        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x, padding_mask=None):

        # 对输入进行线性变换得到K, Q, V矩阵
        k = self.key(x)  # [batch_size, seq_len, head_size]
        q = self.query(x) # [batch_size, seq_len, head_size]
        v = self.value(x) # [batch_size, seq_len, head_size]
        batch_size, seq_len, hidden_dim = x.size()

        # 计算注意力得分（缩放点积）
        weight = q @ k.transpose(-2, -1)  # [batch_size, seq_len, seq_len]
        weight = weight / math.sqrt(self.head_size)  # 缩放避免梯度问题

        # Softmax归一化 + Dropout
        weight = F.softmax(weight, dim=-1)  # [batch_size, seq_len, seq_len]
        weight = self.dropout(weight)

        # 输出注意力结果
        return weight @ v  # [batch_size, seq_len, head_size]

class DecoderSingleHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.head_size = config.head_size
        self.key = nn.Linear(config.n_embd, config.head_size)
        self.query = nn.Linear(config.n_embd, config.head_size)
        self.value = nn.Linear(config.n_embd, config.head_size)
        self.register_buffer(
            "attention_mask",
```

```
            torch.tril(
                torch.ones(config.block_size, config.block_size)
            )
        )
        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x):

        # 对输入进行线性变换得到K, Q, V矩阵
        k = self.key(x) # [batch_size, seq_len, head_size]
        q = self.query(x) # [batch_size, seq_len, head_size]
        v = self.value(x) # [batch_size, seq_len, head_size]
        batch_size, seq_len, hidden_dim = x.size()

        # 计算注意力得分（缩放点积）
        weight = q @ k.transpose(-2, -1)
        weight = weight / math.sqrt(self.head_size)
        weight = weight.masked_fill(
            self.attention_mask[:seq_len, :seq_len] == 0,
            float('-inf')
        )

        # Softmax归一化 + Dropout
        weight = F.softmax(weight, dim=-1) # [batch_size, seq_len, seq_len]
        weight = self.dropout(weight)

        # 输出注意力结果
        return weight @ v  # [batch_size, seq_len, head_size]
```

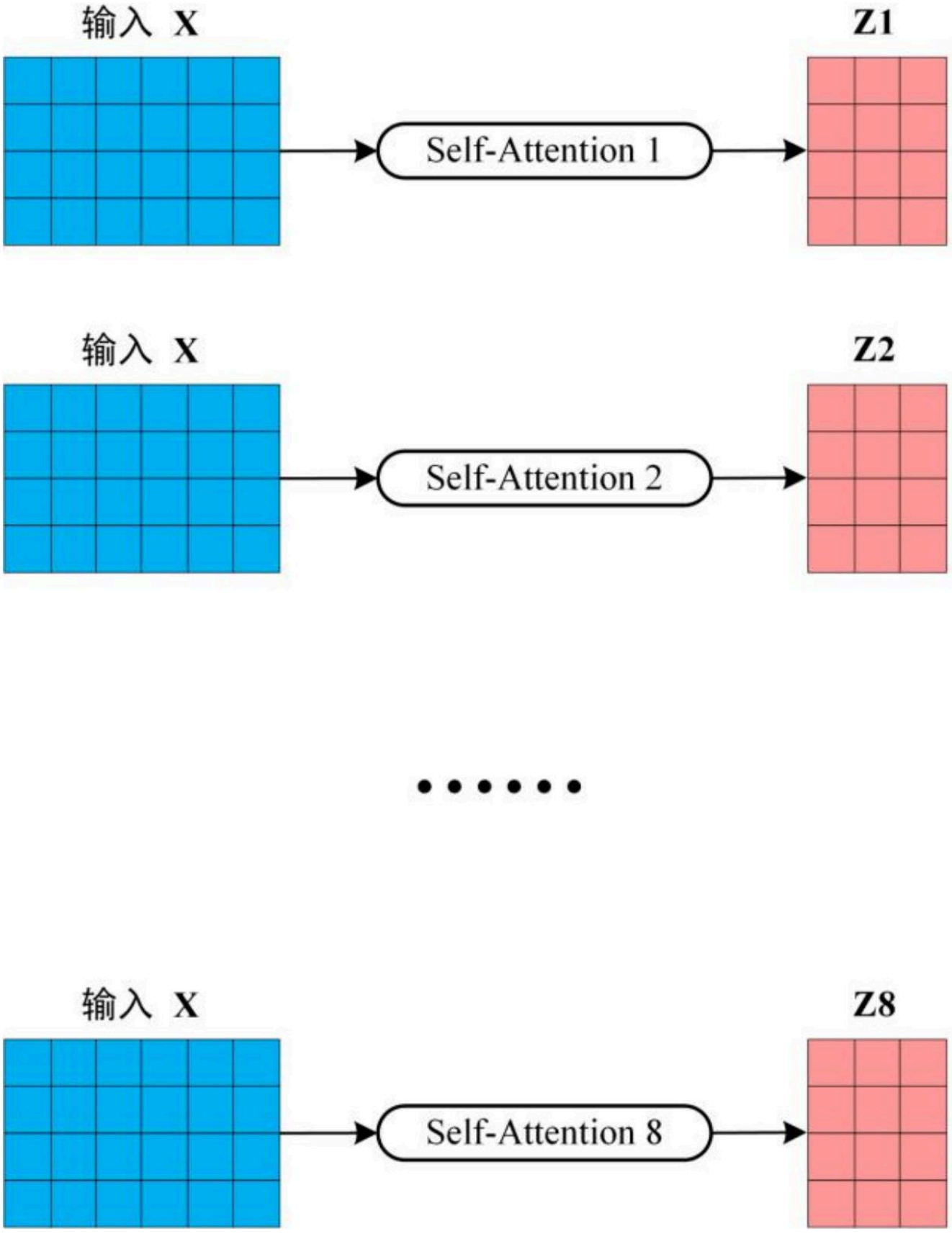
2.4 Multi-Head Attention

在上一步，我们已经知道怎么通过 `Self-Attention` 计算得到输出矩阵 Z，而 `Multi-Head Attention` 是由多个 `Self-Attention` 组合形成的，下图是论文中 `Multi-Head Attention` 的结构图。



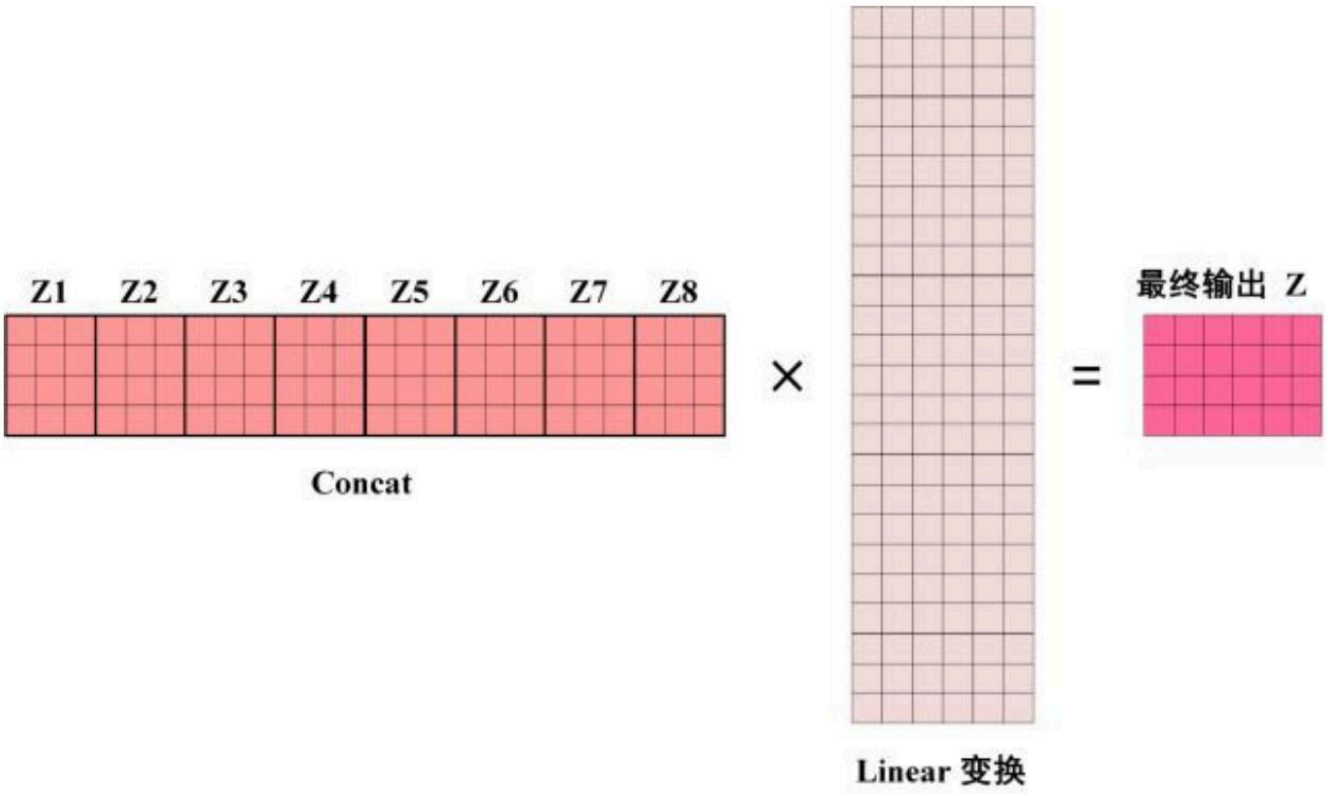
从上图可以看到 `Multi-Head Attention` 包含多个 `Self-Attention` 层，首先将输入 X 分别传递到 h 个不同的 `Self-Attention` 中，

计算得到 h 个输出矩阵 Z。下图是 $h = 8$ 时候的情况，此时会得到 8 个输出矩阵 Z。



得到 8 个输出矩阵 Z_1到 Z_8 之后，`Multi-Head Attention`将它们拼接在一起(`concat`)，然后传入一个`Linear`层，

得到 `Multi-Head Attention` 最终的输出 Z。



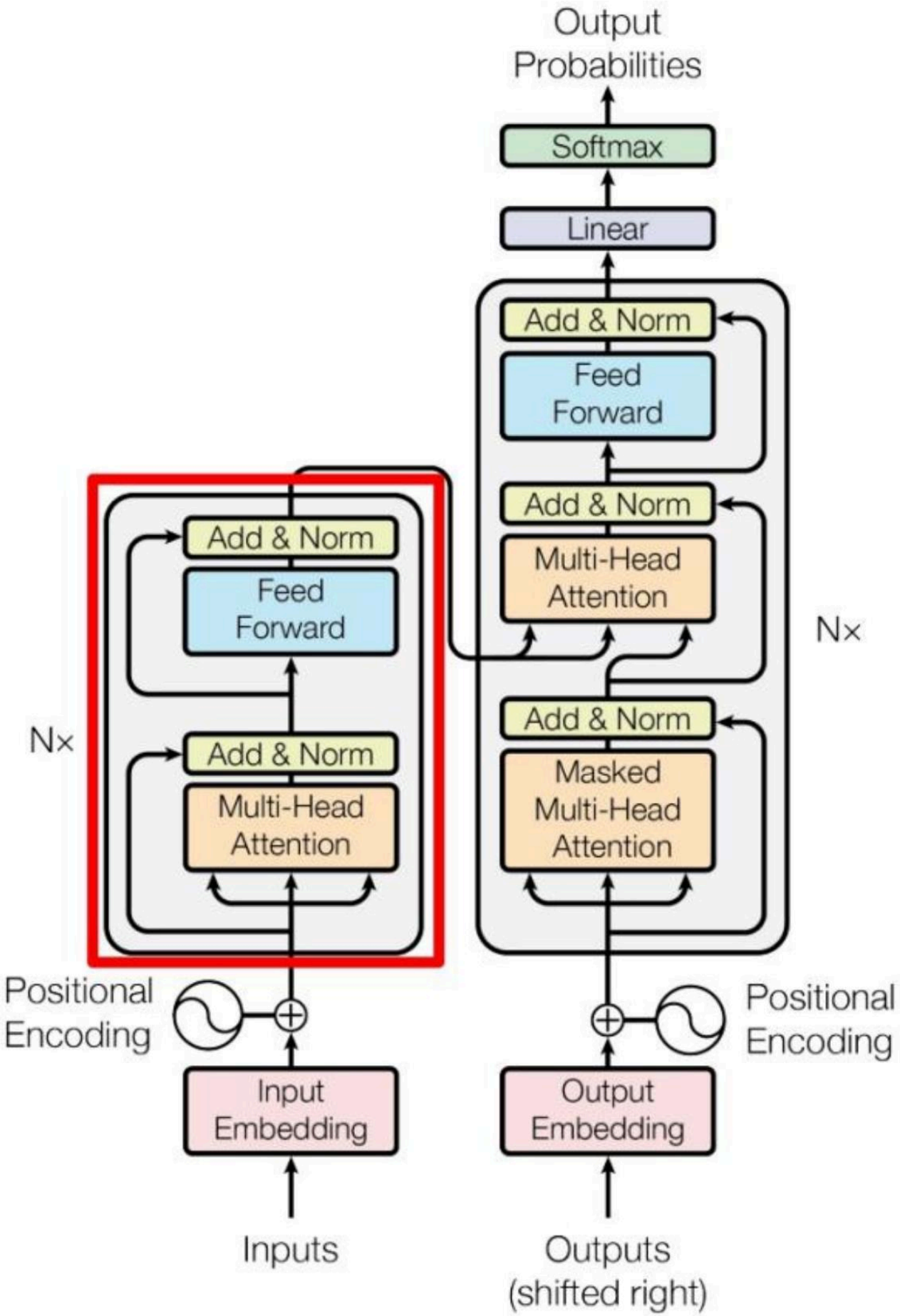
可以看到 `Multi-Head Attention` 输出的矩阵 Z 与其输入的矩阵 X 的维度是一样的。

实现代码如下

```
class DecoderMultiHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.heads = nn.ModuleList(
            [
                DecoderSingleHeadAttention(config)
                for _ in range(config.n_heads)
            ]
        )
        self.proj = nn.Linear(config.n_embd, config.n_embd)
        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x):
        output = torch.cat(
            [head(x) for head in self.heads],
            dim=-1
        )
        output = self.proj(output)
        output = self.dropout(output)
        return output


class EncoderMultiHeadAttention(nn.Module):
    def __init__(self, config):
```

```
        super().__init__()
        self.heads = nn.ModuleList(
            [
                EncoderSingleHeadAttention(config)
                for _ in range(self.n_heads)
            ]
        )
        self.proj = nn.Linear(config.n_embd)
        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x):
        output = torch.cat(
            [head(x) for head in self.heads],
            dim=-1
        )
        output = self.proj(output)
        output = self.dropout(output)
        return output
```

3. Encoder 结构


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

上图红色部分是 `Transformer` 的 `Encoder block` 结构，可以看到是由`Multi-Head Attention`, `Add & Norm1`, `Feed Forward`, `Add & Norm2` 组成的。刚刚已经了解了 `Multi-Head Attention` 的计算过程，现在了解一下`Add & Norm` 和 `Feed Forward` 部分。

3.1 Add & Norm

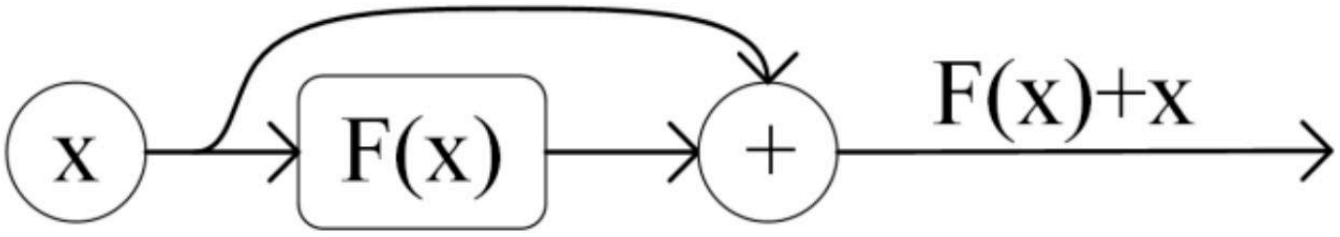
Add & Norm 层由 Add 和 Norm 两部分组成，其计算公式如下：

$$\text{LayerNorm}\big(X + \text{MultiHeadAttention}(X)\big)$$
$$\text{LayerNorm}\big(X + \text{FeedForward}(X)\big)$$

其中 X 表示 `Multi-HeadAttention` 或者 `FeedForward` 的输入，

$MultiHeadAttention(X)$ 和 $FeedForward(X)$ 表示**输出** (**输出**与 X 的维度是一致，所以可以相加)。

- Add指残差连接，通常用于解决多层网络训练的问题，可以保留原始信息，在 ResNet 中经常用到.



- Norm指归一化，通常用于RNN结构，会将每一层神经元的输入都转成均值方差都一样的，这样可以加快收敛。使训练更稳定。

3.2 Feed Forward

`FeedForward` 层比较简单，是一个两层的全连接层，第一层的激活函数为 Relu，第二层不使用激活函数，对应的公式如下。

$$\max(0, XW_1 + b_1)W_2 + b_2$$

X是输入，`FeedForward` 最终得到的输出矩阵的维度与X一致。

实现代码:

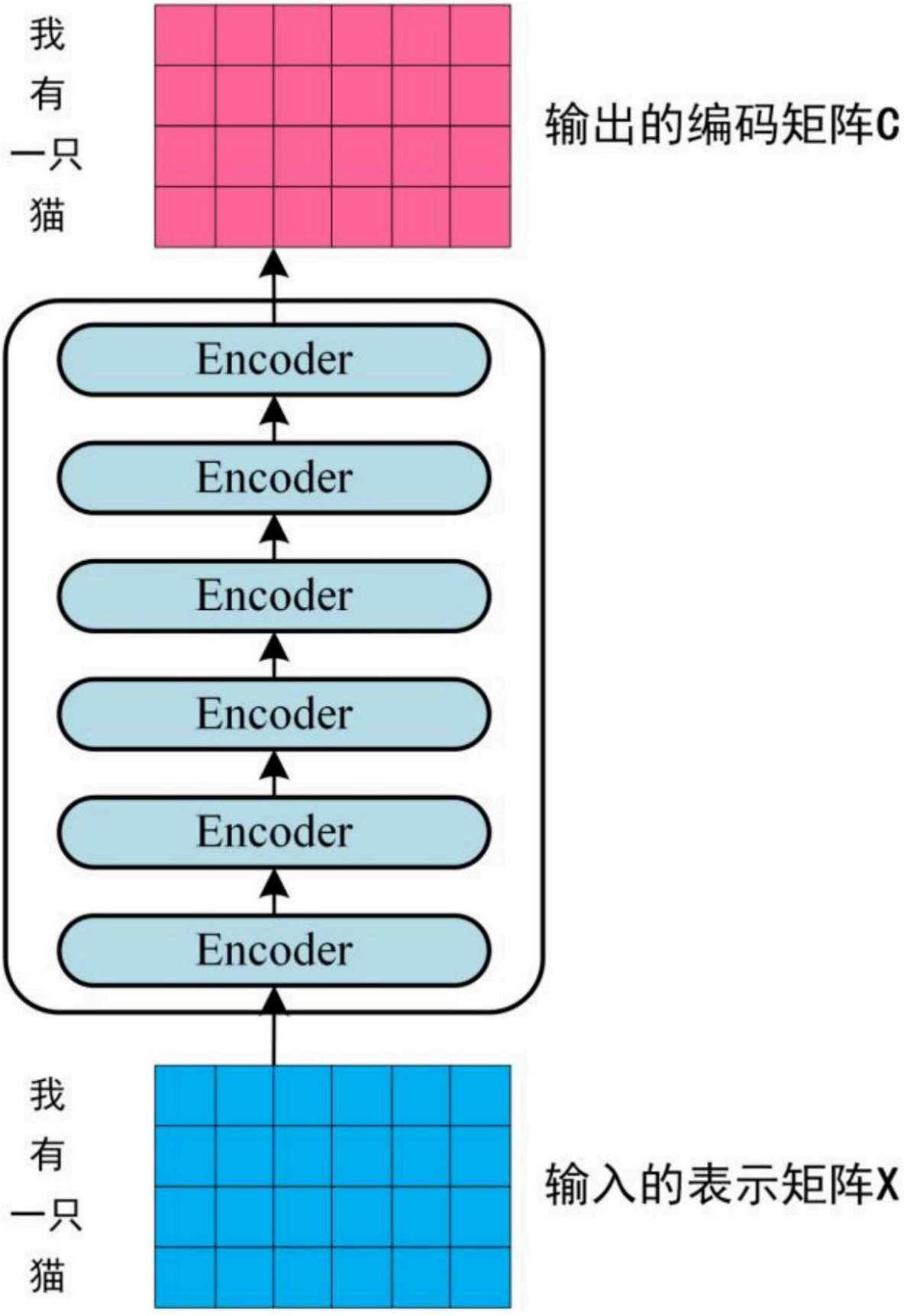
```
class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(config.n_embd, config.hidden_dim),
            nn.GELU(),
            nn.Linear(config.hidden_dim, config.n_embd),
            nn.Dropout(config.dropout)
        )

    def forward(self, x):
        return self.net(x)
```

3.3 组成 Encoder

通过上面描述的 `Multi-Head Attention`, `Feed Forward`, `Add & Norm` 就可以构造出一个 `Encoder block`,

第一个 `Encoder block` 的输入为句子单词的表示向量矩阵，后续 `Encoder block` 的输入是前一个

`Encoder block` 的输出，最后一个 `Encoder block` 输出的矩阵就是编码信息矩阵 C，这一矩阵后续会用到 Decoder 中。



实现代码:

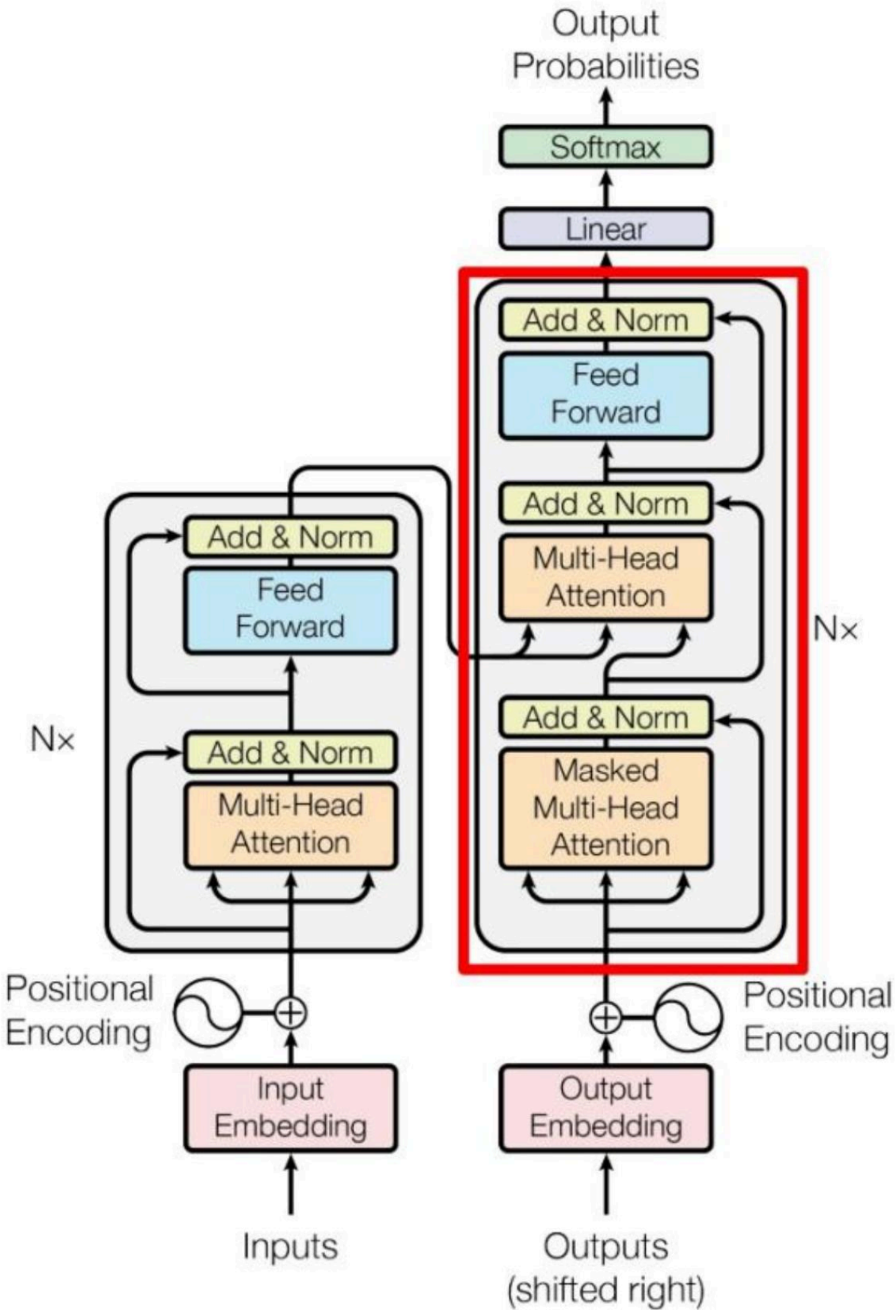
```
class EncoderBlock(nn.Module):
    def __init__(self, config):
```

```
        super().__init__()
        self.att = = EncoderMultiHeadAttention(config)
        self.ffn = FeedForward(config)
        self.ln1 = nn.LayerNorm(config.n_embd)
        self.ln2 = nn.LayerNorm(config.n_embd)

    def forward(self, x, padding_mask=None):
        x = x + self.att(self.ln1(x))  # 残差连接
        x = x + self.ffn(self.ln2(x)) # 残差连接
        return x

class Encoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embedding_table = nn.Embedding(config.vocab_size, config.n_embd)
        self.position_embedding_table = nn.Embedding(config.block_size, config.n_embd)
        self.blocks = nn.Sequential(
            *[EncoderBlock(config) for _ in range(config.n_layer)]
        )
        self.ln_final = nn.LayerNorm(config.n_embd)
        self.apply(self._init_weights)

    def _init_weights(self, module):

        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
        elif isinstance(module, nn.LayerNorm):
            torch.nn.init.zeros_(module.bias)
            torch.nn.init.ones_(module.weight)

    def forward(self, idx, targets=None):
        batch_size, seq_len = idx.size()
        token_emb = self.token_embedding_table(idx)
        pos_emb = self.position_embedding_table(
            torch.arange(seq_len, device=idx.device)
        ).unsqueeze(0)
        x = token_emb + pos_emb  # [batch_size, seq_len, n_embd]
        x = self.blocks(x)
        x = self.ln_final(x)

        loss = None
        if targets is not None:
            batch_size, seq_len, n_embd = x.size()
            logits = nn.Linear(n_embd, config.vocab_size, bias=False).to(x.device)(x)
            logits = logits.view(batch_size * seq_len, config.vocab_size)
            targets = targets.view(batch_size * seq_len)
            loss = F.cross_entropy(logits, targets)
            return logits, loss
        return x, loss
```

4. Decoder 结构

下图红色部分为 `Transformer` 的 `Decoder block` 结构，与 `Encoder block` 相似，但是存在一些区别：

- 包含两个 `Multi-Head Attention` 层。
- 第一个 `Multi-Head Attention` 层采用了 `Masked` 操作。
- **第二个 `Multi-Head Attention` 层的 K, V 矩阵使用 `Encoder` 的编码信息矩阵 C 进行计算，而 Q 使用上一个 Decoder block 的输出计算**。
- 最后有一个`Softmax`层计算下一个翻译单词的概率。


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

4.1 第一个 Multi-Head Attention

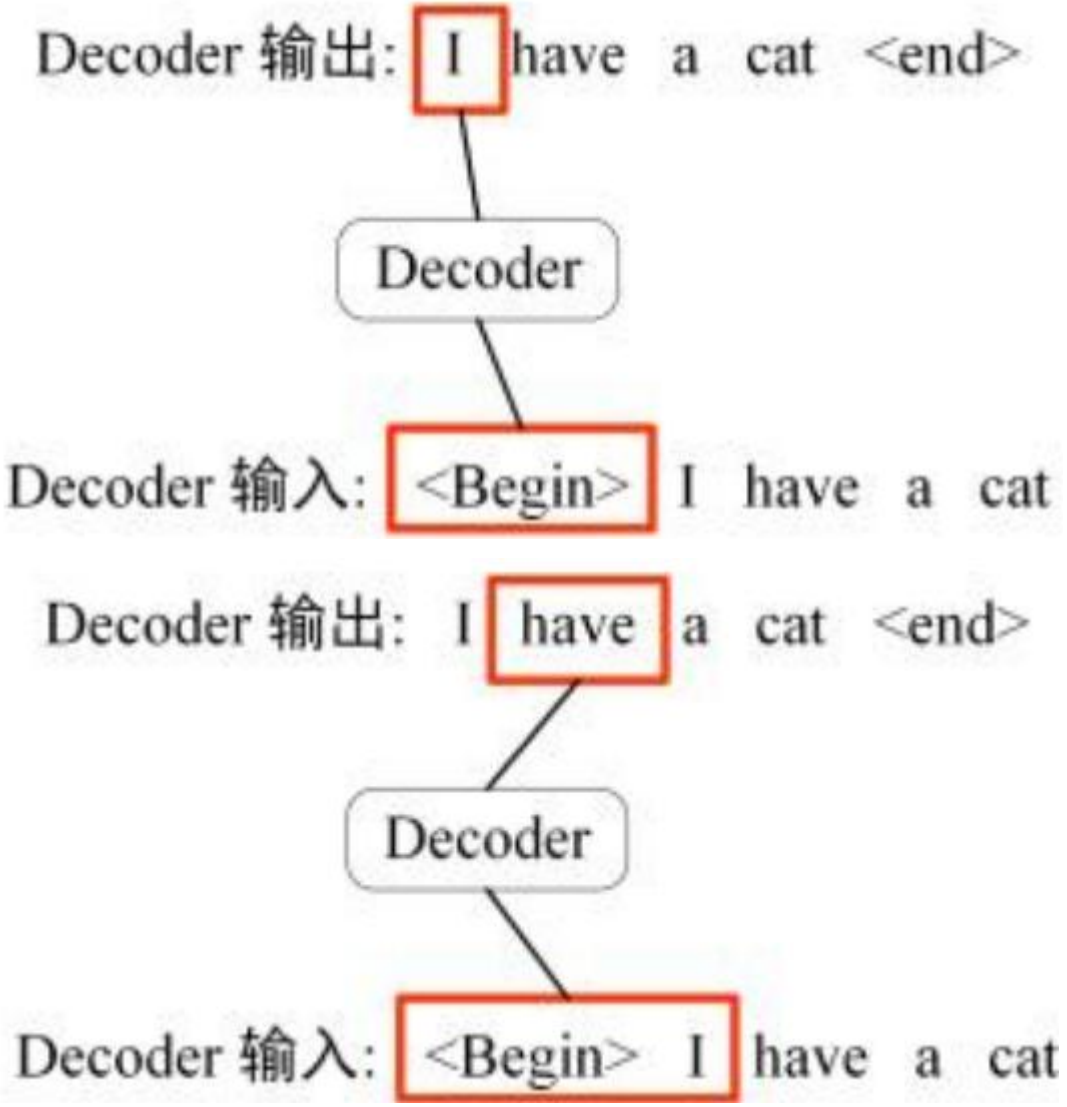
`Decoder block` 的第一个 `Multi-Head Attention` 采用了 **Masked** 操作。

在机器翻译的例子中, 翻译的过程中是顺序翻译的，即翻译完第 i 个单词，才可以翻译第 $i+1$ 个单词。

通过 **Masked** 操作可以**防止第 i 个单词知道 $i+1$ 个单词之后的信息**。

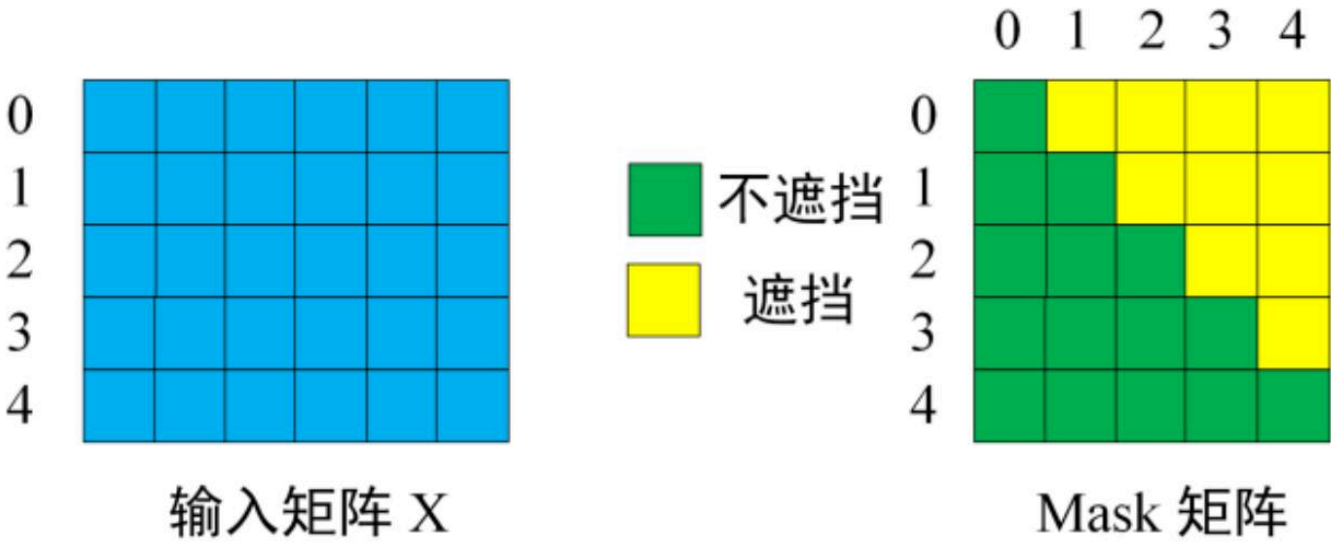
下面以 "我有一只猫" 翻译成 "I have a cat" 为例，了解一下 Masked 操作。

下面的描述中使用了类似Teacher Forcing的概念，在Decoder的时候，是需要根据之前的翻译，求解当前最有可能的翻译，如下图所示。首先根据输入""预测出第一个单词为"I"，然后根据输入"I"预测下一个单词"have"。

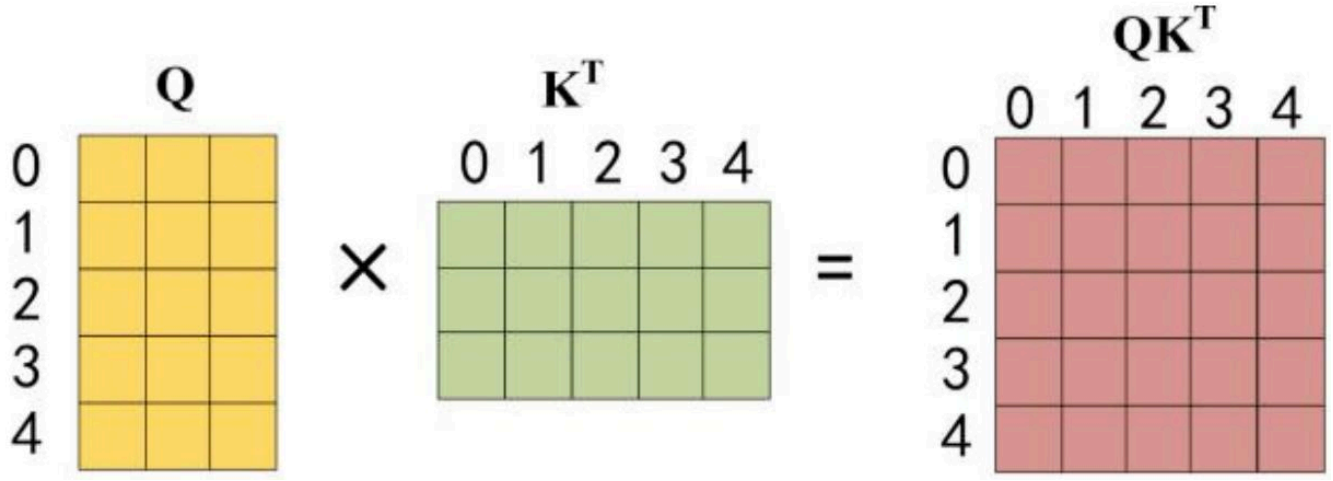


Decoder 可以在训练的过程中使用 Teacher Forcing 并且并行化训练，即将正确的单词序列（I have a cat) 和对应输出（I have a cat）传递到 Decoder。那么在预测第 i 个输出时，就要将第 $i+1$ 之后的单词掩盖住，注意 Mask 操作是在 Self-Attention 的 Softmax 之前使用的，下面用 012345 分别表示 "I have a cat".

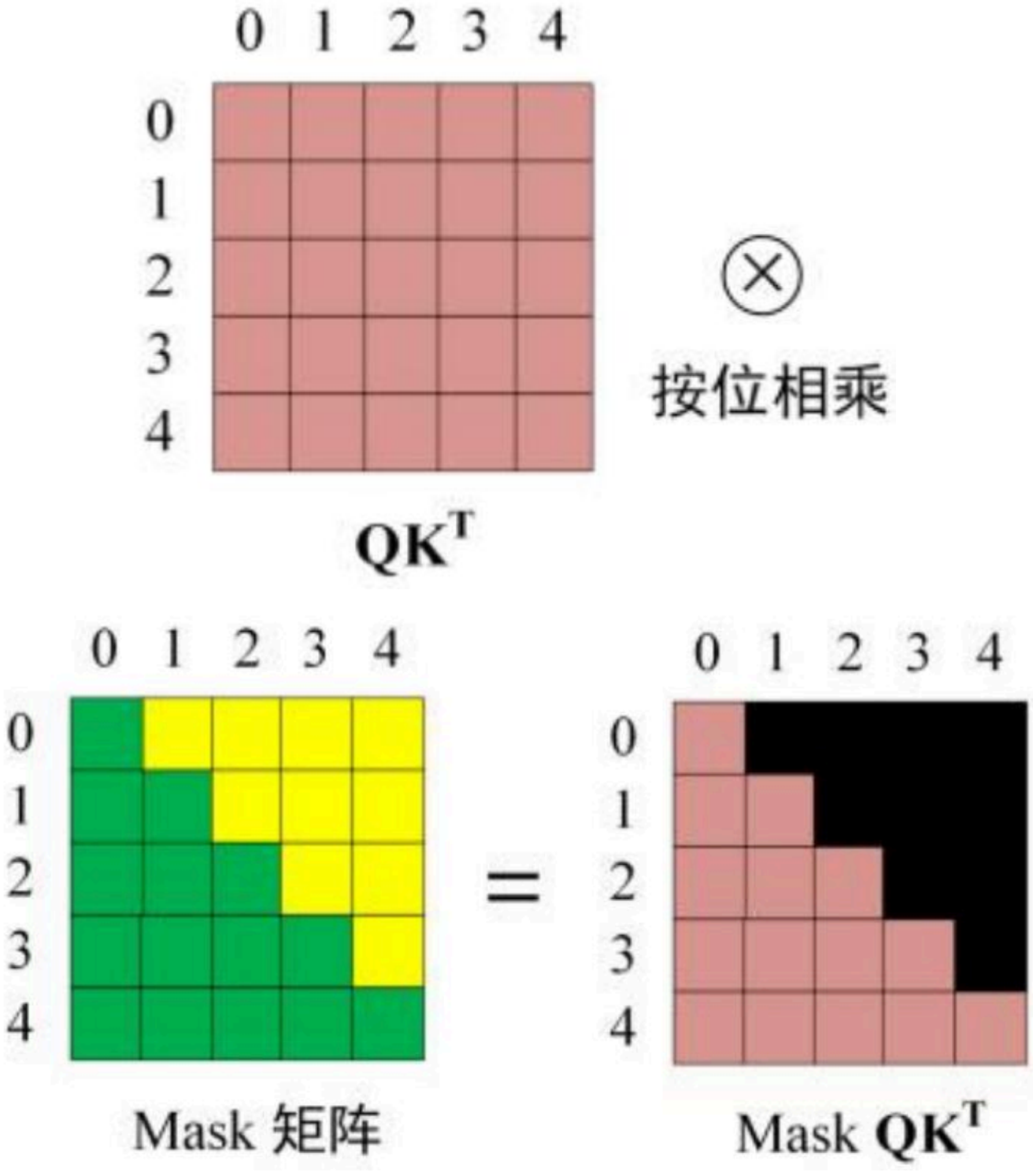
- 第一步：是Decoder的输入矩阵和Mask矩阵，输入矩阵包含"I have a cat" $(0, 1, 2, 3, 4)$ 五个单词的表示向量，Mask是一个 5×5 的矩阵。在Mask可以发现单词0只能使用单词0的信息，而单词1可以使用单词0,1的信息，即只能使用之前的信息。



- 第二步：接下来的操作和之前的 Self-Attention 一样，通过输入矩阵X计算得到Q,K,V矩阵。然后计算Q和K^T的乘积QK^T

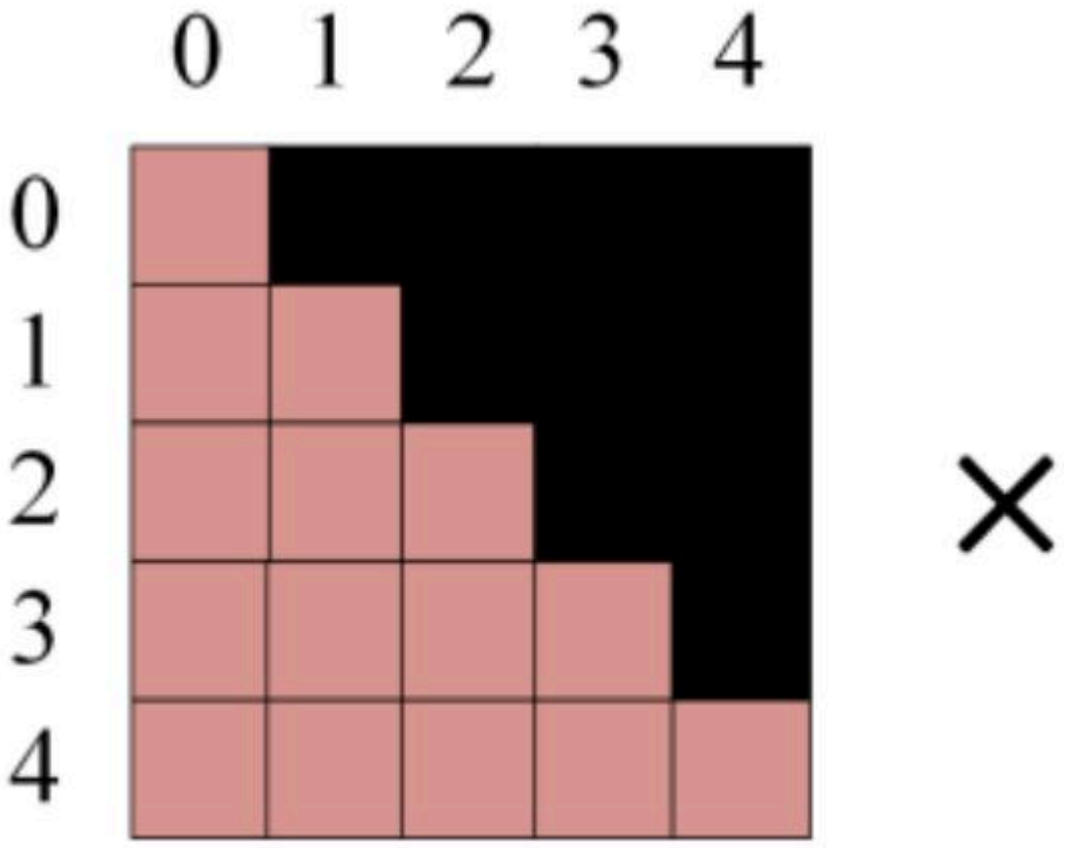


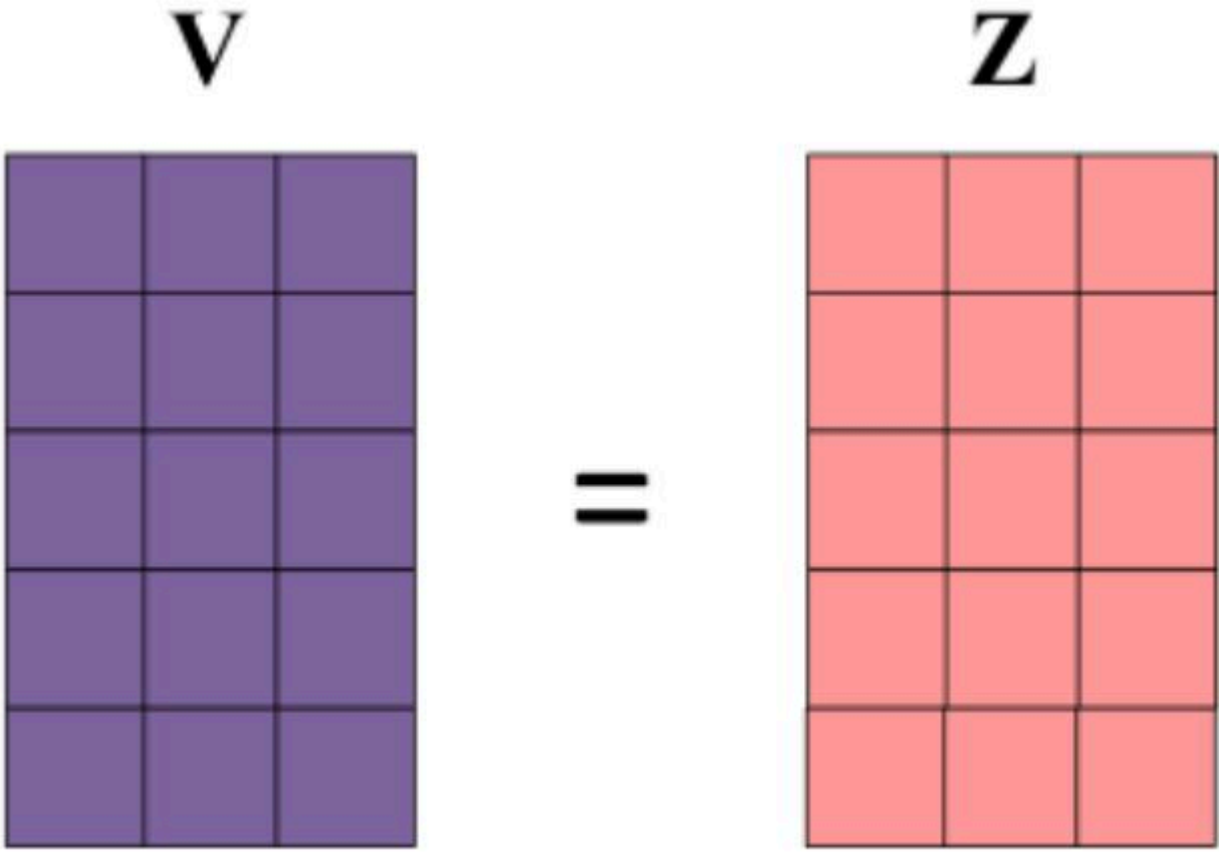
- 第三步：在得到 QK^T 之后需要进行Softmax，计算attention score，我们在Softmax之前需要使用Mask矩阵遮挡住每一个单词之后的信息，遮挡操作如下：



得到Mask QK^T之后在Mask QK^T上进行Softmax，每一行的和都为1。但是单词0在单词1,2,3,4上的attention score都为0。

- 第四步：使用MaskQK^T与矩阵V相乘，得到输出Z，则单词1的输出向量Z1是只包含单词1信息的。


0 1 2 3 4
0
1
2
3
4
×
Mask $\mathbf{QK^T}$


V
Z
=

- 第五步：通过上述步骤就可以得到一个Mask Self-Attention的输出矩阵Zi，然后和Encoder类似，通过Multi-Head Attention拼接多个输出Zi然后计算得到第一个Multi-Head Attention的输出Z，Z与输入X维度一样。

4.2 第二个 Multi-Head Attention

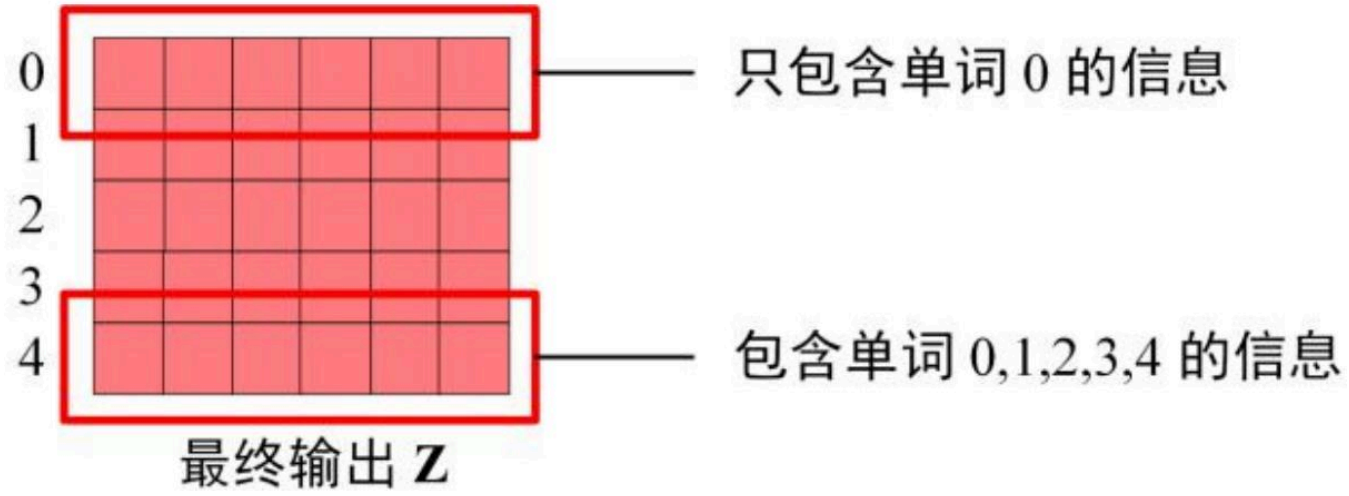
Decoder block 第二个 Multi-Head Attention 变化不大，主要的区别在于其中 Self-Attention 的 K, V 矩阵不是使用上一个 Decoder block 的输出计算的，而是使用 Encoder 的编码信息矩阵 C 计算的。

根据Encoder的输出C计算得到K,V，根据上一个Decoder block的输出Z计算Q(如果是第一个Decoder block则使用输入矩阵X进行计算)，后续的计算方法与之前描述的一致。

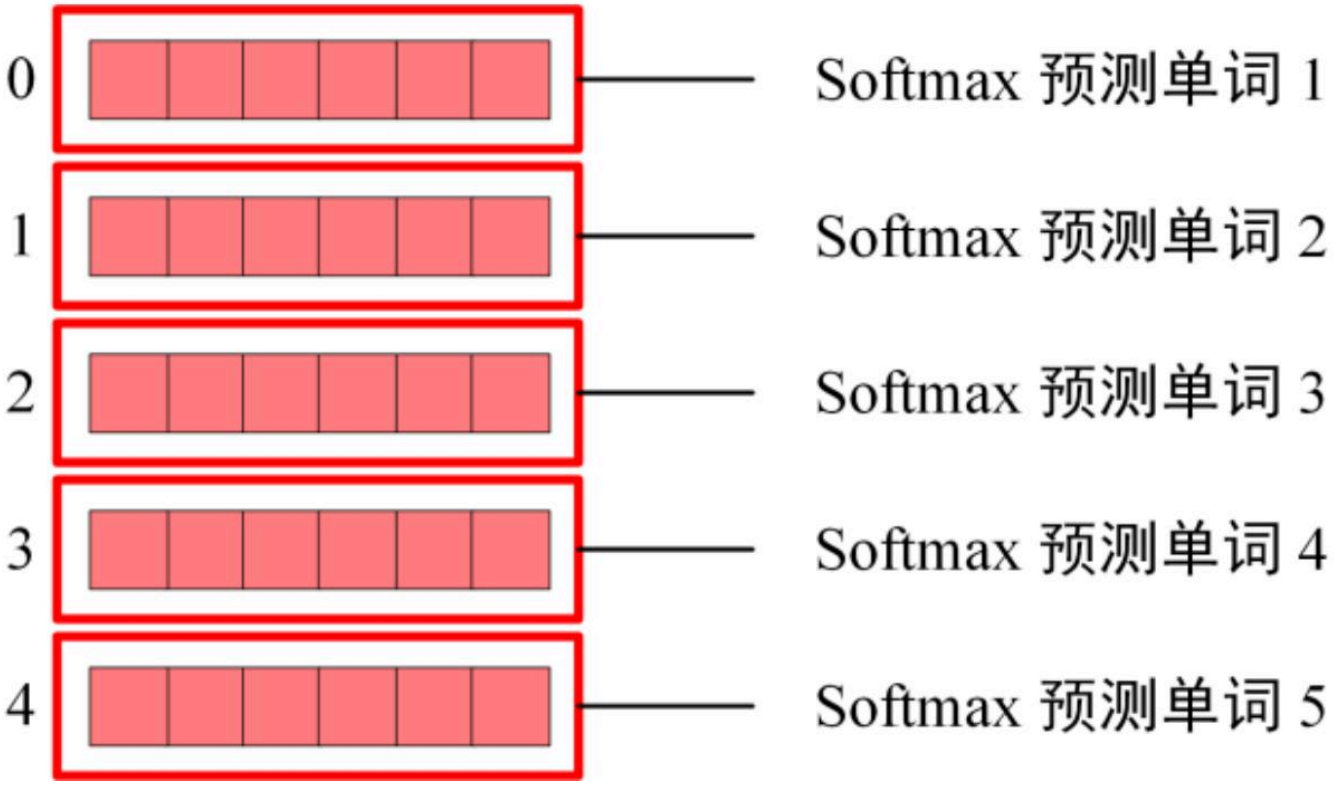
这样做的好处是在 Decoder 的时候，每一位单词都可以利用到 Encoder 所有单词的信息（这些信息无需 Mask）。

4.3 Softmax 预测输出单词

Decoder block 最后的部分是利用 Softmax 预测下一个单词，在之前的网络层我们可以得到一个最终的输出 Z，因为 Mask 的存在，使得单词 0 的输出 Z0 只包含单词 0 的信息，如下：



Softmax 根据输出矩阵的每一行预测下一个单词：



这就是 Decoder block 的定义，与 Encoder 一样，Decoder 是由多个 Decoder block 组合而成。

实现代码:

```
class DecoderBlock(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.att = DecoderMultiHeadAttention(config)
        self.ffn = FeedForward(config)
        self.ln1 = nn.LayerNorm(config.n_embd)
        self.ln2 = nn.LayerNorm(config.n_embd)

    def forward(self, x):
        x = x + self.att(self.ln1(x))
        x = x + self.ffn(self.ln2(x))
        return x

class Decoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embedding_table = nn.Embedding(config.vocab_size, config.n_embd)
        self.position_embedding_table = nn.Embedding(config.block_size, config.n_embd)
        self.blocks = nn.Sequential(
            *[DecoderBlock(config) for _ in range(config.n_layer)]
        )
        self.ln_final = nn.LayerNorm(config.n_embd)
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
        self.token_embedding_table.weight = self.lm_head.weight
        self.apply(self._init_weights)
```

```
    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
        elif isinstance(module, nn.LayerNorm):
            torch.nn.init.zeros_(module.bias)
            torch.nn.init.ones_(module.weight)

    def forward(self, idx, targets=None):
        batch_size, seq_len = idx.size()
        token_enb = self.token_embedding_table(idx)
        pos_emb = self.position_embedding_table(
            torch.arange(seq_len, device=idx.device)
        ).unsqueeze(0)
        x = token_enb + pos_emb
        x = self.blocks(x)
        x = self.ln_final(x)
        logits = self.lm_head(x)
        if targets is None:
            loss = None
        else:
            batch_size, seq_len, vocab_size = logits.size()
            logits = logits.view(batch_size * seq_len, vocab_size)
            targets = targets.view(batch_size * seq_len)
            loss = F.cross_entropy(logits, targets)
        return logits, loss
```

5. Transformer 总结

- Transformer 与 RNN 不同，可以比较好地并行训练。
- Transformer本身是不能利用单词的顺序信息的，因此需要在输入中添加位置Embedding，否则Transformer就是一个词袋模型了。
- Transformer 的重点是 Self-Attention 结构，其中用到的 Q, K, V 矩阵通过输出进行线性变换得到。
- Transformer 中 Multi-Head Attention 中有多个 Self-Attention，可以捕获单词之间多种维度上的相关系数 attention score。