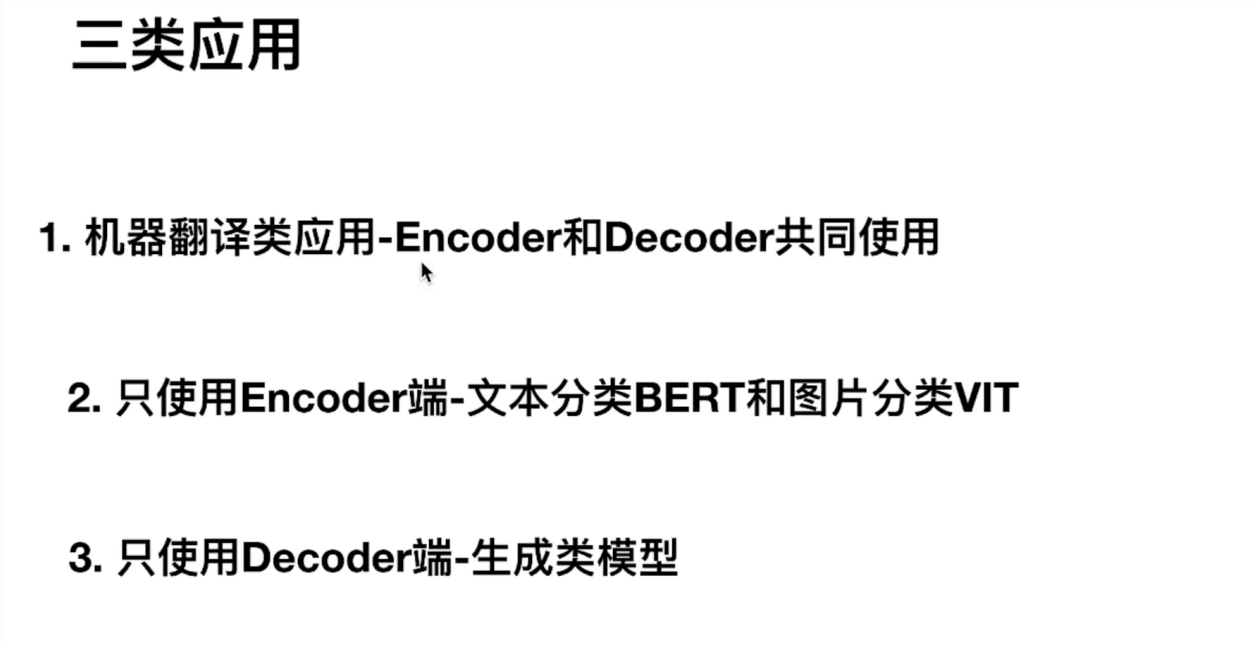
学习链接：视频链接： 【Transformer代码(源码Pytorch版本)从零解读(Pytorch版本） -哔哩哔哩】 https://b23.tv/3GlVDe2

时间线：

- 2024年10月16日 开始
- 2024年10月17日 要学会的内容是代码 这是讲解 方便以后回忆
- 241019 done

Transformer的三类应用：



从图像到图像叫做 encoder+decoder

代码的原链接：

https://github.com/graykode/nlp-tutorial/blob/master/5-1.Transformer/Transformer.ipynb

```
sentences = ['ich mochte ein bier P', 'S i want a beer', 'i want a beer E']
```

首先，从main函数开始，第一行是sentences，看到这行代码需要明白这三个句子，是一组句子，是一个样本，也就是这里的Batch size=1，这行代码有两个疑惑点需要解决 ① 这1、2、3三个句子分别代表的是什么？这三个句子是怎么被模型处理的？ ② 这三个句子的特殊符号，P、S、E分别代表的是什么

为了解决第一个疑惑点，首先串一下Transformer的架构，来看下面这张图

整体看是这样的

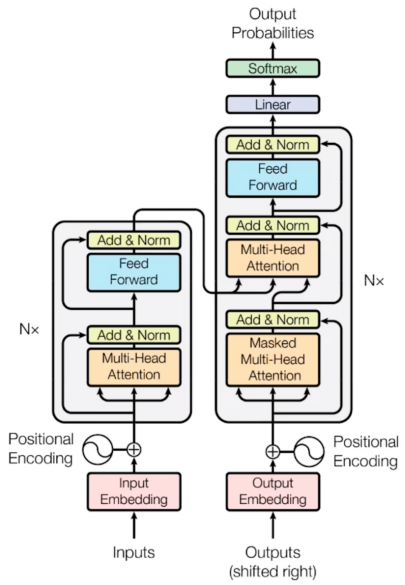


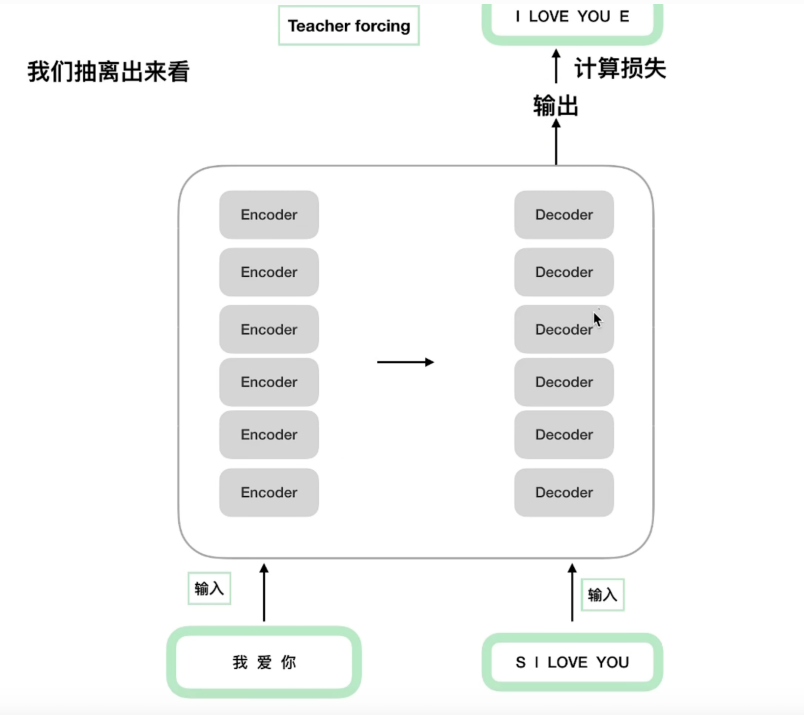
Figure 1: The Transformer - model architecture.

这是一张Transformer的架构图，看到这张图的第一眼，首先要明确，它有两个输入，首先是编码端的输入，Inputs；解码端的输入 Outputs

编码端的输入，经过词向量层和位置编码层，得到一个最终的输入，然后流经自注意力层，然后流经前馈神经网络层，得到一个编码端的输出；编码端的输出先放到这里，后续和解码端进行交互。

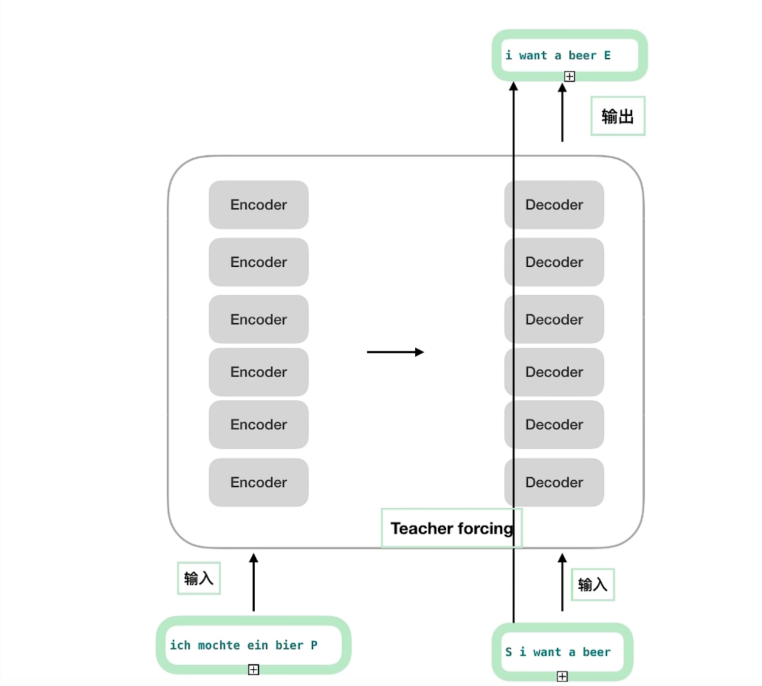
然后解码端的输入，同样经过词向量层，经过位置编码层，得到最终的输入，然后在这里经过了掩码自注意力层，也就是把当前单词之后的，全部要mask掉，然后流经交互自注意力层，在这个交互自注意力层，也就是把这个编码端的输出和这边解码端的信息，进行一个交互，Q矩阵来自我本身这个解码端，KV矩阵来自于编码器的输出，流经这个交互自注意力层，之后经过前馈神经网络层，然后最后得到一个输出。

我们把这个架构图，移植到一个机器翻译的例子，就应该是下图的样子



看到这个图，我们首先也要明白，有两个输入，就像前面Transformer的架构图，一个是编码端的输入，一个是解码端的输入，有一个输出，最后的解码端的输出。回到这个机器翻译的例子这张图，首先映入眼帘的是三个绿色的框。首先下面两个框框的句子代表的是输入，其中"我爱你"这一个句子是3个token，代表的是编码端的输入，这边这个绿色的框"S I LOVE YOU"代表的是解码端的输入，S特殊字符，我们先不用去管，后续会去讲，然后上面那个绿色的框框，一定要注意，这里的绿色的框框不是解码端的输出，再重复一遍，它不是解码端的输出，那它是什么呢？它是我解码端的真实标签。我在decoder输出之后呢，会去和这个真实标签去计算损失，不知道大家有没有理解，这里就相当于如果是一个文本分类任务的话，这里是一个"S I LOVE YOU"，这里这个上面这个绿色的框框，这里相当于是一个0,1标志，代表的是这个句子，它是一个二分类任务，是这个句子的真实标签是什么，在这里映射过来，在这个绿色的框框这里，这并不是一个输出，输出那里才是输出，一定要区分这里这个框是真实标签，不是输出。

那么接下来，我们把这行代码的这三个句子，1,2,3拿过来，拿到这个机器翻译的例子里来，就是这样的一个图



看这里，首先左边德语这个句子作为编码端的输入，右边英语这个句子作为解码端的输入，上面的绿色框作为是解码端的真实标签， 然后解码端的输出和真实标签去做损失；现在①这个疑惑点就解决了，三个绿色的框，三个句子分别代表了什么

接下来我们解决第②这个疑惑点，②这个疑惑点就是说，这个特殊字符 `sentences = ['ich mochte ein bier P', 'S i want a beer', 'i want a beer E']` P、S、E分别代表的是什么。首先这里的 `S` 和 `E` 是很容易理解的，这两个特殊字符分别是英文中 `Start` 和 `End` 的首字母。

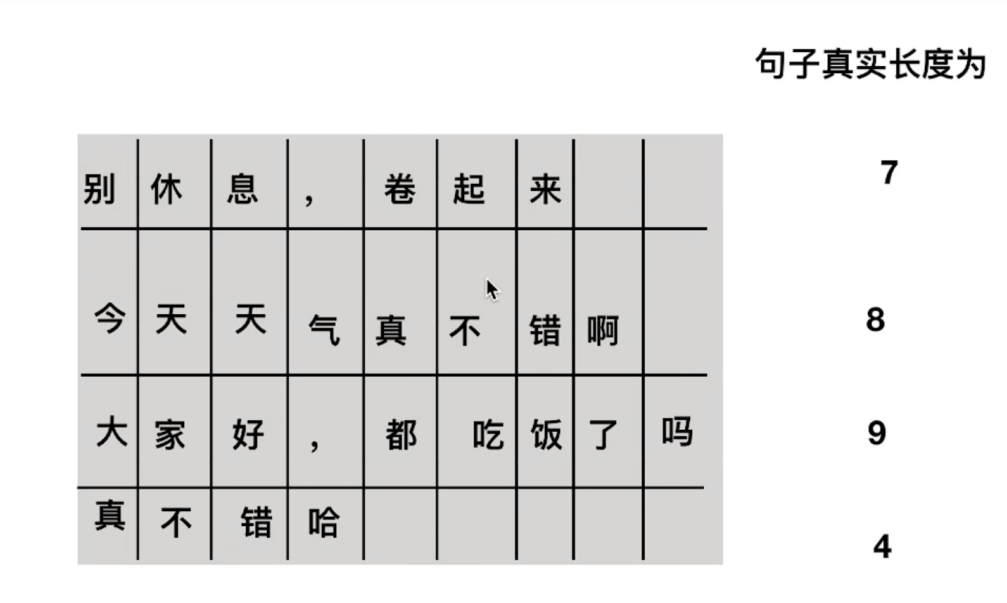
现在，来着重解释 `P` 是什么意思， `P` 代表 `Pad` 字符，也就是填充字符，我们这里举得例子设置的 `batch size=1` ，但是在我们真正训练的时候，为了加快训练速度 batch size往往不是 `1` ，比如说，我们把 `batch size` 设置成 4，如果我们把batch size设置成4，这里我们的句子应该是这样的：

```
sentences = ['ich mochte ein bier P', 'S i want a beer', 'i
want a beer E']
['ich mochte ein bier P', 'S i want a beer', 'i want a beer E']
['ich mochte ein bier P', 'S i want a beer', 'i want a beer E']
['ich mochte ein bier P', 'S i want a beer', 'i want a beer E']
```

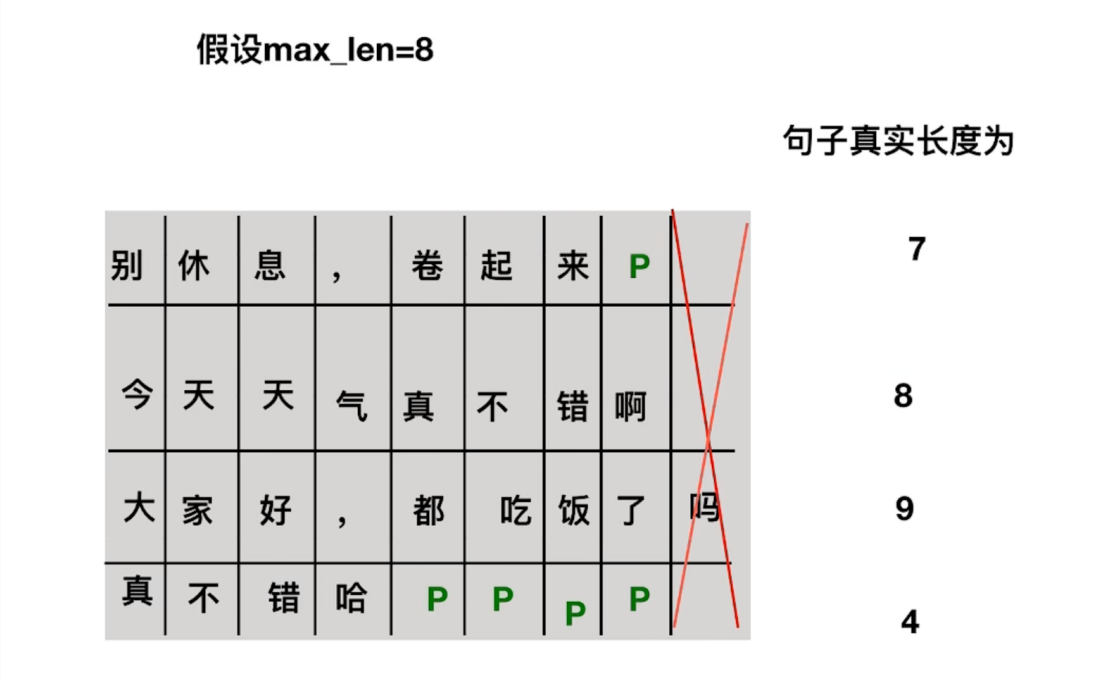
这里的batch size=4，一个batch size中，有4个样本，每个样本有3个句子，第一个句子是encoder

input，第二个句子是decoder input，第三个句子 ground truth

现在来看中文的例子，来看下面这张图：



这里我们的batch size=4，图中给的句子是一个batch size中，4个样本，每个样本的第一个句子，也就是encoder input的部分，这里的每一个句子都是编码端的输入部分，图中把另外两个框框都省略掉了，首先，还是要明确，一个batch在被模型处理掉的时候，为了加快速度，我们往往是用矩阵化的方式去运算，但是如果一个batch中，句子的长度是不一致的，那就不能组成一个有效的矩阵，所以一个常规的操作，就是我们设置一个最大长度 `max length`，比如我们设置为 `8`，那么大于8的部分，我们就要把截掉，不要它，小于8的部分，我们使用特殊字符 `Pad` 字符，去把它填充，于是，经过这样的处理，就变成下图：



大于8的部分，截掉，不要

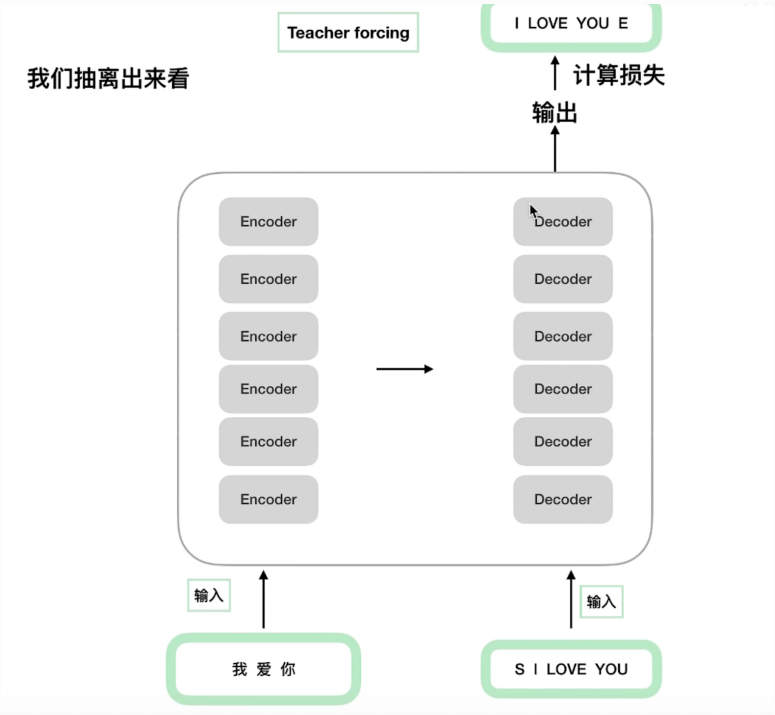
小于8的部分，Pad字符填充

如此一来，这个矩阵中的长度都是8，这样就组成了一个有效矩阵，从而可以被模型处理掉

一定要理解这个P，因为在后续的注意力层中，这个填充字符P，在后续的一个处理中，初学者很容易不明白或者产生误解，就是把 这个P置为 $-\infty$，让它不能对其他的字符产生效果，这边暂时不理解没事，后面将代码的时候，会详细讲这个问题。

在前面的讨论中，解决了两个疑惑点，第一个疑惑点，就是那3个句子，一个样本中的三个句子，分别代表的是：编码端的输入 encoder input、解码端的输入decoder input、解码端的真实标签 ground truth；第二个疑惑点是，三个特殊字符分别是什么，P代表填充字符，S代表的是 开始标志，E 代表的是一个结束标志。

接下来我们再看这个图，



【这段话 主要看右边 decoder端 】想说明的一点是，解码端是不能够并行的，就是说 当我们输入 为 `S` 的时候，输出为 `I`，然后我们这个 `I` 拿过来，作为，作为下一时刻的输入，然后经过 解码层 去输出当前的输出，不能并行的原因是因为 下一时刻的输入 `I` 取决于 或者说 依赖于 上一个时刻的输出，所以不能并行，但是为了加快 训练速度 和 收敛速度，我们常常有这么一种操作， 叫做 `Teacher forcing`

现在来解释，什么叫 teacher forcing，就是说我们把真实标签作为一种输入，真实标签作为 decoder input一起输入模型，这样处理的话，会涉及到一个问题， 比如说当前时刻我们想要输入 `S`，但是我们把所有的句子同时输入，那么模型就会看到 `I LOVE YOU` 这个信息，所以我需要使用 mask 标志，把当前单词的后面词，全部mask住。

【up又解释了一遍】我们需要把未来的词全部mask住，不让它看到当前时刻后面的单词，因为我们在预测的时候，是在一个一个的预测的，而不会一起输入进去，所以我们是看不到后面的信息的，我们要保证训练和预测的时候，形式是一致的。

接下来，我们来看代码 这一部分：

```
    # Transformer Parameters
    # Padding Should be Zero

    ## 构建词表
    src_vocab = {'P': 0, 'ich': 1, 'mochte': 2, 'ein': 3,
'bier': 4}
    src_vocab_size = len(src_vocab)

    tgt_vocab = {'P': 0, 'i': 1, 'want': 2, 'a': 3, 'beer': 4,
'S': 5, 'E': 6}
    number_dict = {i: w for i, w in enumerate(tgt_vocab)}
    tgt_vocab_size = len(tgt_vocab)

    src_len = 5 # length of source
    tgt_len = 5 # length of target

    ## 模型参数
    d_model = 512  # Embedding Size
    d_ff = 2048  # FeedForward dimension
    d_k = d_v = 64  # dimension of K(=Q), V
    n_layers = 6  # number of Encoder of Decoder Layer
    n_heads = 8  # number of heads in Multi-Head Attention
```

这些代码给出的是配置文件；

src_vocab 构建的是编码端的词表，构建词表是一个基本的操作，一个字典就是一个词表，词表是为了方便将中文字符或者说英文字符或者 其他语言的字符 对应为数字，以便被计算机更好的 识别

接着，tgt_vocab这个是解码端的词表，解码端和编码端可以共用一个词表

src_len = 5编码端的输入长度

tgt_len = 5解码端的输入长度

```
    d_model = 512  # Embedding Size
    d_ff = 2048  # FeedForward dimension
    d_k = d_v = 64  # dimension of K(=Q), V
    n_layers = 6  # number of Encoder of Decoder Layer
    n_heads = 8  # number of heads in Multi-Head Attention
```

上面这几行表示模型参数

d_model设置为512，代表的是我们的每一个字符转换为Embedding的时候，它的大小

d_ff代表的是前馈神经网络中，Linear层被映射到多少维度，这里设置的是2048

d_k、d_v后面再去讲

n_layers指的是6个encoder堆叠在一起，decoder也有一个×N，是一致的，把6个堆叠在一起

n_heads多头注意力的时候，把头分为几个，这里是分为8个

接下来，最关键的模型部分，

```
model = Transformer()
```

在写模型的时候，我们一定要遵循两个原则，第一个原则是从整体到局部，就是先把大的框架搭起来，然后再去完善细节部分，第二个特点就是一定要搞清楚，数据的流动形状，就是经过这个模型的处理，输入是什么形状，输出是什么形状

接下来，我们来看Transformer的代码部分，最核心的架构代码

```
## 1. 从整体网络结构看，分为三个部分：编码层、解码层、输出层
class Transformer(nn.Module):
    def __init__(self):
        super(Transformer, self).__init__()
        self.encoder = Encoder() ## 编码层
        self.decoder = Decoder() ## 解码层
        ## 输出层d_model 是我们解码层 每个token输出的维度大小，
        ## 之后会做一个tgt_vocab_size大小的softmax
        self.projection = nn.Linear(d_model, tgt_vocab_size,
bias=False)
    def forward(self, enc_inputs, dec_inputs):
```

```
        ## 这里有两个数据进行输入，一个是enc_inputs 形状为
[batch_size,src_len] src_len就是max_length，主要作为编码段的输入，一
个dec_inputs，形状为[batch_size,tgt_len]，主要作为解码端的输入

        ## enc_inputs 作为输入，形状为 [batch_size,src_len]，输出由
自己的函数内部指定，想要什么指定输出什么，可以是全部tokens的输出，可以是特
定每一层的输出；可以是中间某些参数的输出；
        ## enc_outputs就是主要的输出
        ## enc_self_attns 这里是没记错的是 QK转置相乘之后 softmax 之
后的矩阵值，代表的是每个单词与其他单词的相关性
        enc_outputs, enc_self_attns = self.encoder(enc_inputs)

        ## dec_outputs是decoder主要输出，用于后续的linear映射
        ## dec_self_attns 类比于 enc_self_attns 是查看每个单词对
decoder中输入的其余单词的相关性
        ## dec_enc_attns是decoder中每个单词对encoder中每个单词的相关
性
        dec_outputs, dec_self_attns, dec_enc_attns =
self.decoder(dec_inputs, enc_inputs, enc_outputs)

        ## dec_outputs做映射到词表大小
        dec_logits = self.projection(dec_outputs) # dec_logits
: [batch_size x src_vocab_size x tgt_vocab_size]
        return dec_logits.view(-1, dec_logits.size(-1)),
enc_self_attns, dec_self_attns, dec_enc_attns
```

从架构图中，可以看出，Transformer分为三个部分，第一个编码段、第二个解码端、第三个输出端，输出部分要与真实标签做损失

整体看是这样的

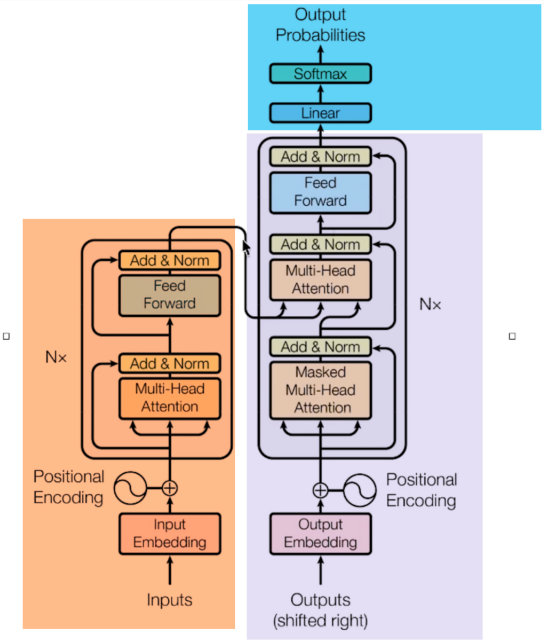


Figure 1: The Transformer - model architecture.

所以，在Transformer类中，init部分，要把3个部分都列出来，

```
    def __init__(self):
        super(Transformer, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()
        self.projection = nn.Linear(d_model, tgt_vocab_size,
bias=False)
```

从上到下，分别对应编码层、解码层、输出部分，接下来，我们重点说一下，输出层 `self.projection = nn.Linear(d_model, tgt_vocab_size, bias=False)` decoder 的输出层，如果是d_model，也就是512个维度，我们要做的就是把这512个维度，映射到词表大小，是解码端的词表大小，然后，之后做 softmax，去看当前时刻，哪个词出现的概率最大，这是我 projection 的一个作用，然后我们去看实现函数 `forward`

```
    def forward(self, enc_inputs, dec_inputs):

        enc_outputs, enc_self_attns = self.encoder(enc_inputs)

        dec_outputs, dec_self_attns, dec_enc_attns =
self.decoder(dec_inputs, enc_inputs, enc_outputs)

        dec_logits = self.projection(dec_outputs)
        # dec_logits : [batch_size x src_vocab_size x
tgt_vocab_size]

        return dec_logits.view(-1, dec_logits.size(-1)),
enc_self_attns, dec_self_attns, dec_enc_attns
```

`forward(self, enc_inputs, dec_inputs)`

对于我们整个Transformer，它是接收的两个输入，一个是编码端的输入，一个是解码端的输入，所以我们的forward 接收的是两个参数：`enc_inputs, dec_inputs`，那么它们的形状是什么呢?我们已经说过了，形状非常重要，对于encoder来讲，它们的输入是 $\text{batch\_size} \times \text{src\_len}$，对应的是 输入部分句子的长度，decoder这边是 $\text{batch\_size} \times \text{tgt\_len}$，解码端句子的长度 （sequence_length），统一起来，就是 batch_size × sequenc_length

接下来，讲解这三行代码：

```
enc_outputs, enc_self_attns = self.encoder(enc_inputs)

dec_outputs, dec_self_attns, dec_enc_attns =
self.decoder(dec_inputs, enc_inputs, enc_outputs)

dec_logits = self.projection(dec_outputs)
```

就是把在初始化函数中，已经放置好的编码端、解码端、以及输出层，通过数据流动串起来；

```
enc_outputs, enc_self_attns = self.encoder(enc_inputs)
```

首先是编码端的输入，流经了编码器，得到了编码端的输出

至于编码的输出这里，在我们自己最开始复现代码的时候，我们肯定是对输出什么是一头浆糊的，所以我们在写的时候，完全可以把这里置空，不用去管它，等实现完 encoder，边写边琢磨，要输出什么，或者你要输出什么，因为我们在写一个代码函数的时候，输出什么东西是由你自己函数内部制定的，你想要什么，可以指定输出什么，比如你可以指定全部 tokens，也就是全部 tokens的输出，也可以指定每一层 某个token的输出，也可以是中间某些参数的 输出，你都可以自己去指定 ；所以在最开始写的时候，如果去复现，你完全可以不用写的这么的 一步到位

这里 `enc_outputs` 是一个主要的输出；而这里的 `enc_self_attns` 更像是一个 Q、K矩阵转置相乘之后，经过softmax之后那个矩阵，代表的就是我每个单词和其他单词之间的一个相关性矩阵【得分矩阵】，这是为了看一下，哪个单词和哪个单词 更相似 更有关系，可能是为了可视化

如上，便是编码端的所有操作

```
dec_outputs, dec_self_attns, dec_enc_attns = self.decoder(dec_inputs, enc_inputs, enc_outputs)
```

然后，我的解码端接收的输入有：解码端的输入 dec_inputs 、以及编码端的输出 enc_outputs作为输入，对于解码端这边，最主要的是接收这两个输入，一个是解码端的输入，一个是编码端的输出，这个是输出是为了和解码端进行交互的，所以这个编码端的输出和编码器的输入是必须要的，其中 `self.decoder` 还有一个输入 `enc_inputs` 指的是 编码端的输入，这个输入up主也记不太清了 有没有用 后续再看代码，得到解码器的输出 `dec_outputs` 以后，我们让它流经 `self.projection` 层，具体开始讲这句代码：

```
dec_logits = self.projection(dec_outputs)
```

也就是做了一个映射词表的操作，这个就是Transformer整体架构图的一个代码，然后我们再去详细的看，每一个编码层、解码层、输出层每一个的代码，是如何实现的，我们先来看 encoder，也就是编码器层的代码：

```
## 2.Encoder部分包含三个部分：词向量embedding，位置编码部分，注意力层及后续的前馈神经网络
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        # 这个其实就是去定义生成一个矩阵，大小是 src_vocab_size * d_model
        self.src_emb = nn.Embedding(src_vocab_size, d_model)
        # 位置编码情况，这里是固定的正余弦函数，也可以使用类似词向量nn.Embedding
        self.pos_emb = nn.Embedding.from_pretrained(get_sinusoid_encoding_table(src_len+1, d_model),freeze=True)
        self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)])

    def forward(self, enc_inputs): # enc_inputs : [batch_size x source_len]
        enc_outputs = self.src_emb(enc_inputs) + self.pos_emb(torch.LongTensor([[1,2,3,4,0]]))
        enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs)
        enc_self_attns = []
        for layer in self.layers:
            enc_outputs, enc_self_attn = layer(enc_outputs, enc_self_attn_mask)
            enc_self_attns.append(enc_self_attn)
        return enc_outputs, enc_self_attns
```

继续看架构图encoder的部分，从整体到局部

整体看是这样的

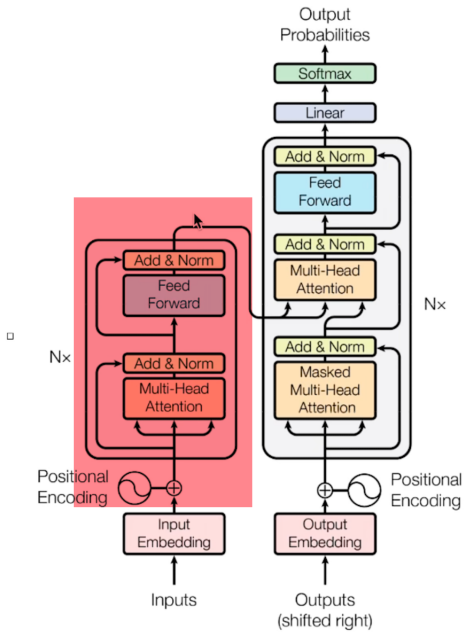


Figure 1: The Transformer - model architecture.

再去细分代码，代码包含三个部分，首先是词向量层`input embedding` 位置编码层`positonal encoding` 前馈神经网络和自注意力层，这两个构成一个`encoder layer`，因此初始化函数把这三部分先列出来，具体怎么实现，再去细分代码，init如下：

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.src_emb = nn.Embedding(src_vocab_size, d_model)
        self.pos_emb =
nn.Embedding.from_pretrained(get_sinusoid_encoding_table(src_le
n+1, d_model),freeze=True)
        self.layers = nn.ModuleList([EncoderLayer() for _ in
range(n_layers)])
```

```
## 2. Encoder 部分包含三个部分：词向量embedding，位置编码部分，注意力层及后续的前馈神经网络
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.src_emb = nn.Embedding(src_vocab_size, d_model)  ## 这个其实就是去定义生成一个矩阵，大小是 src_vocab_size * d_model
        self.pos_emb = PositionalEncoding(d_model) ## 位置编码情况，这里是固定的正余弦函数，也可以使用类似词向量的nn.Embedding获得一个可以更新学习的位置编码
        self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)]) ## 使用ModuleList对多个encoder进行堆叠，因为后续的encoder并没有使用词向量和位置编码，所以抽离出来；
```

逐行注释：

`self.src_emb = nn.Embedding(src_vocab_size, d_model)`

这里其实就是定义去生成一个矩阵，大小是`src_vocab_size * d_model`

这是一个词向量层，相当于定义一个词表，词表到 `d_model` 一个矩阵

```
self.pos_emb = PositonalEncoding(d_model)
```

位置编码情况，这里是固定的正余弦函数，也可以使用类似词向量的 `nn.Embedding` 获得一个可以更新学习的位置编码

这是一个位置编码层

PositonalEncoding 位置编码的具体实现函数

```
self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)])
```

使用ModuleList对多个encoder进行堆叠，因为后续的encoder并没有使用词向量和位置编码，所以抽离出来

layers层是把前馈神经网络和自注意力层组合

实现encoder的堆叠

接下来看 实现函数

```
def forward(self, enc_inputs):
    ## 这里我们的 enc_inputs 形状是:  [batch_size x source_len]

    ## 下面这个代码通过src_emb，进行索引定位，enc_outputs输出形状是[batch_size, src_len, d_model]
    enc_outputs = self.src_emb(enc_inputs)

    ## 这里就是位置编码，把两者相加放入到了这个函数里面，从这里可以去看一下位置编码函数的实现：3.
    enc_outputs = self.pos_emb(enc_outputs.transpose(0, 1)).transpose(0, 1)

    ##get_attn_pad_mask是为了得到句子中pad的位置信息，给到模型后面，在计算自注意力和交互注意力的时候去掉pad符号的影响，去看一下这个函数 4.
    enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs)
    enc_self_attns = []
    for layer in self.layers:
        ## 去看EncoderLayer 层函数 5.
        enc_outputs, enc_self_attn = layer(enc_outputs, enc_self_attn_mask)
        enc_self_attns.append(enc_self_attn)
    return enc_outputs, enc_self_attns
```

实现函数一定要看输入是什么

```
def forward(self, enc_inputs): # enc_inputs : [batch_size x source_len]
```

encoder接受一个输入，也就是 `encoder_inputs` 形状 `batch_size × sourcez_len`

```
enc_outputs = self.src_emb(enc_inputs)
```

下面这个代码通过 `src_emb` 进行索引定位， `enc_output` 输出形状是 `[batch_size,src_len,d_model]`

这行代码引用的是init代码的词向量的这个代码 `self.src_emb`

这个部分要做的就是，通过索引定位 把对应数字的词向量提取出来 最后形成一个矩阵

enc_outputs矩阵的形状是 `[batch_size x source_len × d_model]`

比如一个字符，比如说 `我` 这个字符对应的是 `1` 这个数字，那么就去 `nn.Embedding` 这个里面，去找 `1` 对应的向量，然后拿过来，这就是这行代码实现的功能

简单说，就是把字符，转化为数字，然后把数字转化为 `embedding`

其实 字符 转化为数字 已经做到了 这里就是把 数字 索引 转化为 embedding 转化为对应的向量

```
enc_outputs =
self.pos_emb(enc_outputs.transpose(0,1)).transpose(0,1)
```

这里是位置编码，把两者相加放入到这个函数，从这里可以看一下位置编码函数的实现：3.

接下来看 位置编码层 主要实现 两个作用

第一个是 接收 enc_outputs，即 词向量之后的输出，作为 位置编码的输入，然后把位置编码 和 词向量 进行相加

接下来 我们看 位置编码 是怎么实现的：

```python
## 3. PositionalEncoding 代码实现
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()

        ## 位置编码的实现其实很简单，直接对照着公式去敲代码就可以，下面这个代码只是其中一种实现方式；
        ## 从理解来讲，需要注意的就是偶数和奇数在公式上有一个共同部分，我们使用log函数把次方拿下来，方便计算；
        ##假设我的demodel是512，那么公式里的pos代表的从0，1，2，3..511的每一个位置，2i那个符号中i从0取到了255，那么2i对应取值就是0,2,4...510
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)## 这里需要注意的是pe[:, 0::2]这个用法，就是从0开始到最后面，补长为2，其实代表的就是偶数位置
        pe[:, 1::2] = torch.cos(position * div_term)##这里需要注意的是pe[:, 1::2]这个用法，就是从1开始到最后面，补长为2，其实代表的就是奇数位置
        ## 上面代码获取之后得到的pe:[max_len*d_model]

        ## 下面这个代码之后，我们得到的pe形状是：[max_len*1*d_model]
        pe = pe.unsqueeze(0).transpose(0, 1)

        self.register_buffer('pe', pe)  ## 定一个缓冲区，其实简单理解为这个参数不更新就可以

    def forward(self, x):
        """
        x: [seq_len, batch_size, d_model]
        """
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)
```

讲解：

之前Transformer面试的时候，都会要求，手写位置编码代码！

位置编码的实现 就是对照着公式 去敲代码 图中给出的时候 位置编码的一种实现方式

位置编码的实现公式：

位置编码公式

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

两个共有的部分：e^(-(2i)/dmodel*log(10000))

比如现在512个维度

position就是从0到511，代表的是每个位置，2i代表的是偶数，2i+1代表的是奇数，也就是偶数和奇数分别使用不同的函数，一个使用 sin函数，一个使用 cos函数 ，摘出共有的部分，写代码

```
div_term = torch.exp(torch.arange(0,d_model,2).float()* (-
math.log(10000.0) / d_model ))
```

为什么？没看明白这，看明白了

$$
\begin{aligned}
&pos \times \text{公共部分} \\
&\text{公共部分} = \frac{1}{10000^{\frac{2i}{d_{model}}}} \\
&\text{取对数：} e^{log\frac{1}{10000^{\frac{2i}{d_{model}}}}} \\
&= e^{-log10000^{\frac{2i}{d_{model}}}} \\
&= e^{-\frac{2i}{d_{model}} \times log10000} \\
&= e^{\frac{2i}{d_{model}} \times -log10000}
\end{aligned}
\tag{2}
$$

接下来，代码 position × 共有部分

```
pe[:,0::2] = torch.sin(position * div_term)
# 这里注意pe[:,0::2]这个用法，就是从0开始到最后面，步长为2，其实就是代表的偶数位置
pe[:,1::2] = torch.cos(position * div_term)
# 这里注意pe[:,1::2]这个用法，就是从1开始到最后面，步长为2，其实就是代表的奇数位置
#上面代码 获取之后 得到的 pe的形状 就是 =[max_len * d_model]
```

这里注释讲解的很清楚了

这里我们加一个维度的变换

```
pe = pe.unsqueeze(0).transpose(0,1)
```

经过处理之后，代码形状变为 `max_len × 1× d_model`

```
self.register_buffer('pe',pe)
```

这行代码的作用就是定义一个缓冲区，简单理解为参数不更新

接下来看 forward 函数，forward 函数传入的是一个 词向量，经过词向量的一个参数，接下来和位置编码相加，得到输出 从而得到 encoder_layer的输入：

```
def forward(self,x):
  # x : [seq_len,batch_size,d_model]  为什么这的batch_size在中间
  x = x + self.pe[:x.size(0),:]
  return self.dropout(x)
```

以上是位置编码函数的实现。

```
enc_self_attn_mask = get_attn_pad_mask(enc_inputs,enc_inputs)
enc_self_attns = []
for layer in self.layers:
  # EncoderLayer层函数5.
  enc_outputs,enc_self.attn =
layer(enc_outputs,enc_self_attn_mask)
  enc_self_attns.append(enc_self_attn)

return enc_outputs,enc_self_attns
```

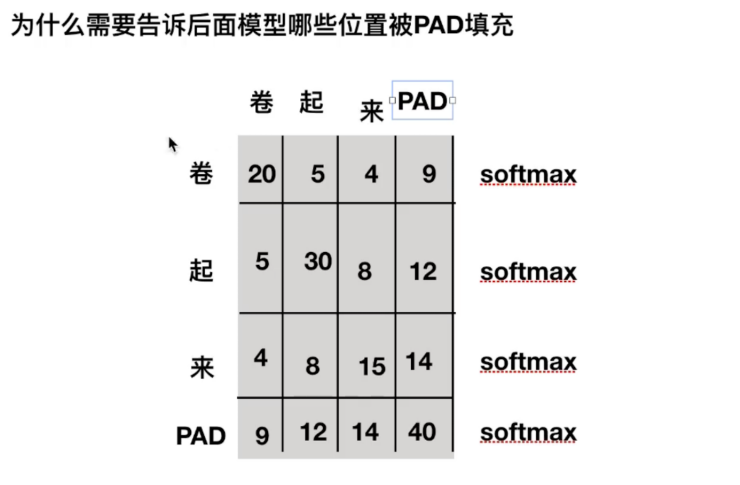
讲解 `enc_self_attn_mask = get_attn_pad_mask(enc_inputs,enc_inputs)`

get_attn_pad_mask是为了得到句子中pad的位置信息，得到模型后面，在计算自注意力和交叉注意力的时候去掉pad符号的影响

get_attn_pad_mask这个函数 非常重要，实现起来不难，但确实很好的考察Transformer细节的知识点，这个函数实现的目的，告诉后面的模型以及后面的层，在原始句子的输入中，哪些部分，是被pad符号填充的，在最开始的时候，我们说过，一个Batch不可能长度都一致，所以为了更好地组成矩阵被模型处理我们会设置一个max_length最大长度，大于最大长度的被截断丢弃，小于最大长度的作用pad填充，既然最开始的时候，用pad符号填充了，首先我们需要告诉后面的模型，那些是被pad符号填充的，那么我们怎么告诉后面的模

型是被填充了呢？要解决两个问题，首先是为什么要告诉后面的模型，其次，怎么告诉后面的模型。

现在处理第一个问题，为什么要告诉：



首先，注意力矩阵，有这个公式：

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

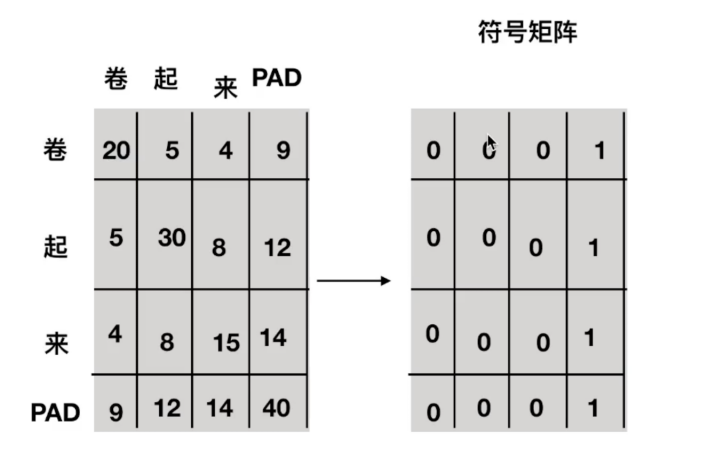
这个图上的数字，是公式 softmax 里的表达 $\frac{QK^T}{\sqrt{d_k}}$

表示的是 每个单词 对 其他单词 的相似性，具体来说 卷 和 卷的相似性，卷和起 的相似性，卷和来的相似性 ，卷和 pad 的相似性，其实说相似性并不准确，因为还没有进行 softmax 函数，但是可以当成相似性来理解。

接下来，我们以 卷 这个相似性为例，在得到这个矩阵之后，每一横行 进行 softmax ，做 softmax

是为了 得到概率，知道这个四个字符 `卷、起、来、PAD` 对于 卷来说 哪个更重要，比如 卷 最重要占0.8，起 还可以 占0.1，那么问题来了 PAD字符是我原来句子中不存在的，是我为了组成一个有效的矩阵，填充到里面的，那么在计算相似性的时候，当然不应该把PAD这个符号计算在内，但是在QK矩阵相乘除以根号dk之后，我们发现这里确实有一个值：9，因此我们要想办法去掉它，这就引出了第二个问题，怎么告诉后面的模型这个信息呢?

我们需要用到一个符号矩阵，把PAD符号所在的位置 设置为1，不是PAD符号的 设置为 0，刚刚那个代码得到的就是一个 符号矩阵



enc_self_attn_mask = get_attn_pad_mask(enc_inputs,enc_inputs)

接下来，我们来看具体的内部 `get_attn_pad_mask` 是怎么实现的：

```
## 4. get_attn_pad_mask

## 比如说，我现在的句子长度是5，在后面注意力机制的部分，我们在计算出来QK转置除以根号之后，softmax之前，我们得到的形状
## len_input * len*input  代表每个单词对其余包含自己的单词的影响力

## 所以这里我需要有一个同等大小形状的矩阵，告诉我哪个位置是PAD部分，之后在计算计算softmax之前会把这里置为无穷大；

## 一定需要注意的是这里得到的矩阵形状是batch_size x len_q x len_k，我们是对k中的pad符号进行标识，并没有对k中的做标识，因为没必要

## seq_q 和 seq_k 不一定一致，在交互注意力，q来自解码端，k来自编码端，所以告诉模型编码这边pad符号信息就可以，解码端的pad信息在交互注意力层是没有用到的；

def get_attn_pad_mask(seq_q, seq_k):
    batch_size, len_q = seq_q.size()
    batch_size, len_k = seq_k.size()
    # eq(zero) is PAD token
    pad_attn_mask = seq_k.data.eq(0).unsqueeze(1)  # batch_size x 1 x len_k, one is masking
    return pad_attn_mask.expand(batch_size, len_q, len_k)  # batch_size x len_q x len_k
```

代码有注释，可以帮助理解

接下来，逐行讲解

```
def get_attn_pad_mask(seq_q,seq_k):
  batch_size,len_q = seq_q.size()
  batch_size,len_k = seq_k.size()
  # eq(zero) is PAD token
  pad_attn_mask = seq_k.data.eq(0).unsqueeze(1)
  # batch_size × 1 × len_k,one is masking
  # batch_size × len_q × len_k
  return pad_attn_mask.expand(batch_size,len_q,len_k)
```

逐行

```
batch_size,len_q = seq_q.size()
batch_size,len_k = seq_k.size()
```

分别得到 batch size、输入的长度和 输出的长度

注意这里传进来的是 `def get_attn_pad_mask(seq_q,seq_k)` seq_q、seq_k 注意这两个输入 并不是完全一致，自注意力层的时候 是一致的，但是在交互注意力层的时候，q来自 解码层、k来自 编码端，两者是不一致的

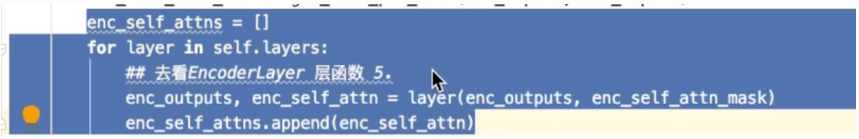
```
pad_attn_mask = seq_k.data.eq(0).unsqueeze(1)
```

这行代码 就是去看 输入的 seq_k 里面 哪些 是 PAD 符号 然后 把这个位置 置为 true

eq(0) 也就是说 如果对应的位置是0，pad符号 对应到 数字上就是 0 我们之前说过的 如果等于0 就设置为true 就是为1 也就是经过了一个转换 把 数字为0的部分 也就是为PAD 的那个数字 对应的索引部分 转换成 true 这个符号 对应的就是 1

`return pad_attn_mask.expand(batch_size,len_q,len_k)` 然后 它重复了多少次呢 重复了 输入为 q的 这个长度 也就是 len_q,最后得到的这个形状是 `batch_size × len_q × len_k`

接下来 我们继续来看 encoder 后面的代码



这部分代码 就是encoder 三个部分中的最后一个部分 就是前馈神经网络和 自注意力神经网络的一个组成部分，因为它是一个 堆叠的 所以说 这里有一个循环 把每一层的输出 作为 下一层的输入 所以我们只需要看一层代码就可以了 `enc_outputs,enc_self_attn = layer(enc_outputs,enc_self_attn_mask)` 首先接收的是 上一层 编码器的输出 和 传给每一层 的 attn_mask, enc_self_attn_mask这个是需要传给每一层的，这个是通过

```python
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.src_emb = nn.Embedding(src_vocab_size, d_model)  ## 这个其实就是去定义生成一个矩阵，大小是 src_vocab_size * d_model
        self.pos_emb = PositionalEncoding(d_model) ## 位置编码情况，这里是固定的正余弦函数，也可以使用类似词向量的nn.Embedding获得一个可以更新学习的位置编码
        self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)]) ## 使用ModuleList对多个encoder进行堆叠，因为后续的encoder并没有使用词向量和位置编码，所以抽离出来；

    def forward(self, enc_inputs):
        ## 这里我们的 enc_inputs 形状是： [batch_size x source_len]

        ## 下面这个代码通过src_emb，进行索引定位，enc_outputs输出形状是[batch_size, src_len, d_model]
        enc_outputs = self.src_emb(enc_inputs)

        ## 这里就是位置编码，把两者相加放入到了这个函数里面，从这里可以去看一下位置编码函数的实现；3.
        enc_outputs = self.pos_emb(enc_outputs.transpose(0, 1)).transpose(0, 1)

        ##get_attn_pad_mask是为了得到句子中pad的位置信息，给到模型后面，在计算自注意力和交互注意力的时候去掉pad符号的影响，去看一下这个函数 4.
        enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs)
        enc_self_attns = []
        for layer in self.layers:
            ## 去看EncoderLayer 层函数 5.
            enc_outputs, enc_self_attn = layer(enc_outputs, enc_self_attn_mask)
            enc_self_attns.append(enc_self_attn)
        return enc_outputs, enc_self_attns
```

红框代码得到的信息

接下来 我们来看

```
enc_outputs,enc_self_attn =
layer(enc_outputs,enc_self_attn_mask)
```

`layer` 这个函数 是怎么实现的， 如下：

```python
## 5. EncoderLayer ：包含两个部分，多头注意力机制和前馈神经网络
class EncoderLayer(nn.Module):
    def __init__(self):
        super(EncoderLayer, self).__init__()
        self.enc_self_attn = MultiHeadAttention()
        self.pos_ffn = PoswiseFeedForwardNet()

    def forward(self, enc_inputs, enc_self_attn_mask):
        ## 下面这个就是做自注意力层，输入是enc_inputs，形状是[batch_size x seq_len_q x d_model] 需要注意的是最初始的QKV矩阵是等同于这个输入的，去看一下enc_self_attn函数 6.
        enc_outputs, attn = self.enc_self_attn(enc_inputs, enc_inputs, enc_inputs, enc_self_attn_mask) # enc_inputs to same Q,K,V
        enc_outputs = self.pos_ffn(enc_outputs) # enc_outputs: [batch_size x len_q x d_model]
        return enc_outputs, attn
```

首先来看 初始化函数是怎么实现的 首先来看 init函数 一个是 自注意力层 一个是 前馈神经网络层

前馈神经网络层 很简单 就是一个 linear层 `self.pos_ffn = PoswiseFeedForwardNet()`

但是 自注意力层 使用的是一个 多头注意力层 也是整个代码 最核心的部分

`self.enc_self_attn = MultiHeadAttention()`

还是继续看 forward函数，从forward函数 来看 自注意力层

首先 forward函数的自注意力层 接收了 4个输入

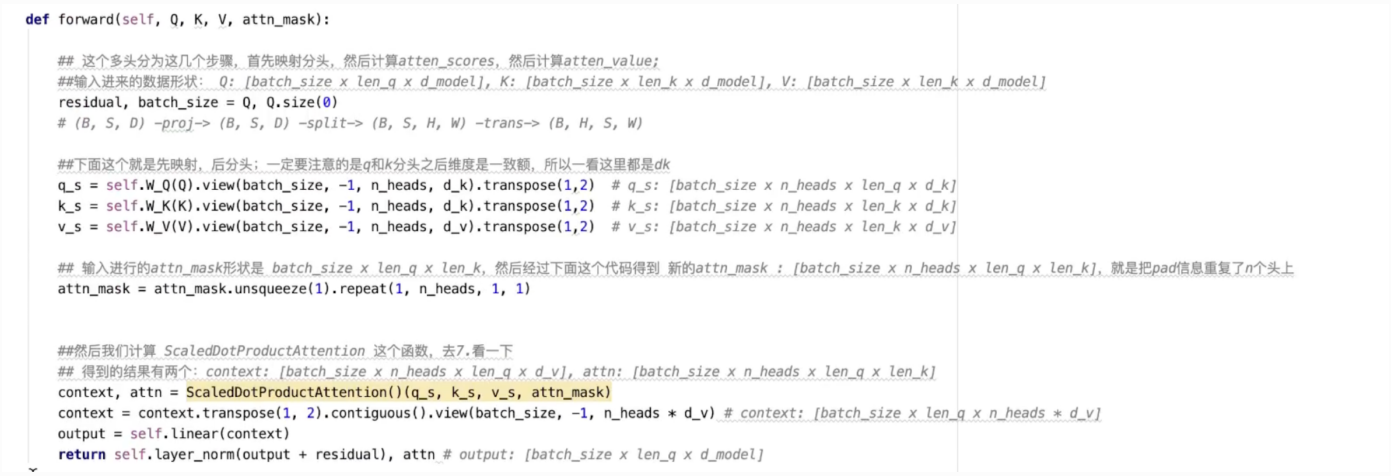
```
enc_outputs,attn =
self.enc_self_attn(enc_inputs,enc_inputs,enc_inputs,enc_self_at
tn_mask)
```

enc_self_attn_mask 这个 输入 就是 将PAD的位置 设置为1的 符号矩阵

enc_inputs,enc_inputs,enc_inputs 这三个输入 分别代表 最原始的 qkv

然后，我们来看 多头注意力机制是怎么实现的

```
## 6. MultiHeadAttention
class MultiHeadAttention(nn.Module):
    def __init__(self):
        super(MultiHeadAttention, self).__init__()
        ## 输入进来的QKV是相等的，我们会使用映射linear做一个映射得到参数矩阵Wq, Wk,Wv
        self.W_Q = nn.Linear(d_model, d_k * n_heads)
        self.W_K = nn.Linear(d_model, d_k * n_heads)
        self.W_V = nn.Linear(d_model, d_v * n_heads)
        self.linear = nn.Linear(n_heads * d_v, d_model)
        self.layer_norm = nn.LayerNorm(d_model)
```



```
def forward(self, Q, K, V, attn_mask):

    ## 这个多头分为这几个步骤，首先映射分头，然后计算atten_scores，然后计算atten_value;
    ##输入进来的数据形状： Q: [batch_size x len_q x d_model], K: [batch_size x len_k x d_model], V: [batch_size x len_k x d_model]
    residual, batch_size = Q, Q.size(0)
    # (B, S, D) -proj-> (B, S, D) -split-> (B, S, H, W) -trans-> (B, H, S, W)

    ##下面这个就是先映射，后分头；一定要注意的是q和k分头之后维度是一致额，所以一看这里都是dk
    q_s = self.W_Q(Q).view(batch_size, -1, n_heads, d_k).transpose(1,2)  # q_s: [batch_size x n_heads x len_q x d_k]
    k_s = self.W_K(K).view(batch_size, -1, n_heads, d_k).transpose(1,2)  # k_s: [batch_size x n_heads x len_k x d_k]
    v_s = self.W_V(V).view(batch_size, -1, n_heads, d_v).transpose(1,2)  # v_s: [batch_size x n_heads x len_k x d_v]

    ## 输入进行的attn_mask形状是 batch_size x len_q x len_k，然后经过下面这个代码得到 新的attn_mask : [batch_size x n_heads x len_q x len_k]，就是把pad信息重复了n个头上
    attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads, 1, 1)


    ##然后我们计算 ScaledDotProductAttention 这个函数，去7.看一下
    ## 得到的结果有两个：context: [batch_size x n_heads x len_q x d_v], attn: [batch_size x n_heads x len_q x len_k]
    context, attn = ScaledDotProductAttention()(q_s, k_s, v_s, attn_mask)
    context = context.transpose(1, 2).contiguous().view(batch_size, -1, n_heads * d_v) # context: [batch_size x len_q x n_heads * d_v]
    output = self.linear(context)
    return self.layer_norm(output + residual), attn # output: [batch_size x len_q x d_model]
```

这部分代码 一定要 搞清楚 数据流动形状

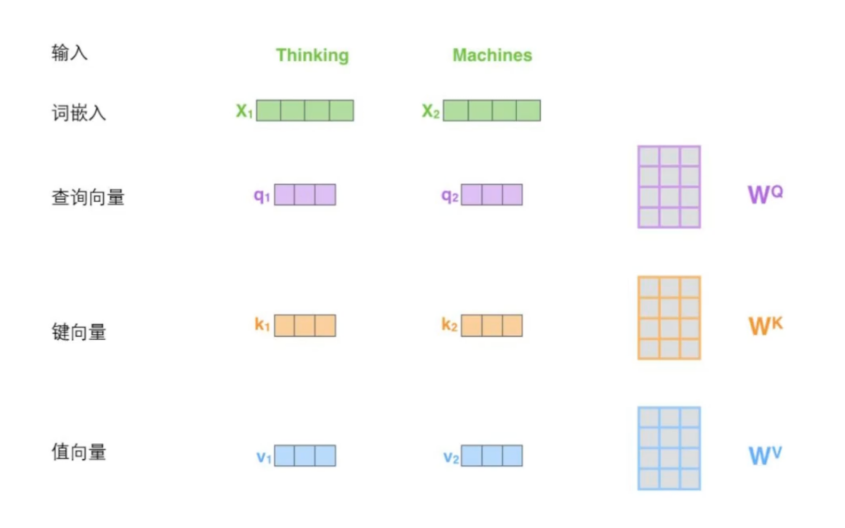
首先，在初始化函数中，前三行是得到一个 映射矩阵

```
self.W_Q = nn.Linear(d_model,d_k * n_heads)
self.W_K = nn.Linear(d_model,d_k * n_heads)
self.W_V = nn.Linear(d_model,d_v * n_heads)
```

就是把 d_model 分别映射到 d_k * n_heads 、d_k * n_heads 、d_v * n_heads

注意到 前两个 是相等的 都是 d_k * n_heads ， 因为需要保证 最后得到的 Q K矩阵 维度是相同的

从图片来理解：



首先，自注意力层，输入是什么？输入是最原始的Q、K、V

而在图中，我们的输入是什么？输入的是词向量之后的一个输出，也就是词嵌入，也就是把Thinking转变成x1，Machines转变为x2,词嵌入batch=1，length=2，d_model=4,然后我们生成三个矩阵 ， 也就是三个参数矩阵 WQ、Wk、Wv，这三个参数矩阵的作用是什么呢？它分别和这两个输入做计算，生成对应的 q、k、v ， 也就是 查询向量、键向量 和 值向量，也就是我们使用词嵌入 和 三个参数矩阵 去计算 对应的 查询向量、键向量 和 值向量，在这个代码实现的时候，是把词嵌入 复制了三份，也就是Q、K、V复制了三份

```
# MultiHeadAttention
def forward(self,Q,K,V,attn_mask):
```

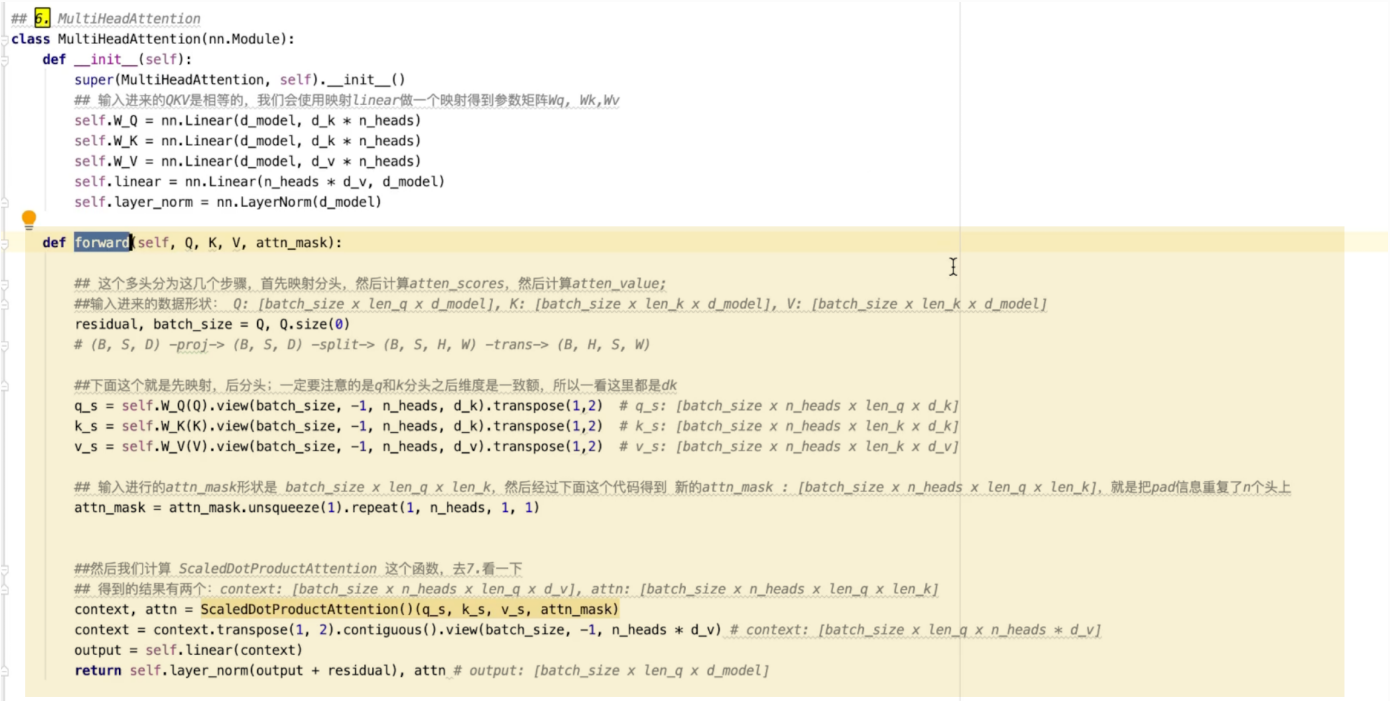
调用：

```
# EncoderLayer

enc_outputs,attn =
self.enc_self_attn(enc_inputs,enc_inputs,enc_inputs,enc_self_at
tn_mask)
# enc_inputs to same Q,K,V
```

在编码端做自注意力的时候，是把enc_inputs复制了三份，分别赋值为Q，K，V

分别和对应的参数矩阵做计算，现在思考 为什么是复制三份 而不是一份传进去，原因是因为在解码端还有交互注意力，它的Q矩阵来自于解码端，KV矩阵来自编码端，两者并不相同，所以接收三个输入，接下来看forward函数：



```
## 6. MultiHeadAttention
class MultiHeadAttention(nn.Module):
    def __init__(self):
        super(MultiHeadAttention, self).__init__()
        ## 输入进来的QKV是相等的，我们会使用映射linear做一个映射得到参数矩阵Wq, Wk,Wv
        self.W_Q = nn.Linear(d_model, d_k * n_heads)
        self.W_K = nn.Linear(d_model, d_k * n_heads)
        self.W_V = nn.Linear(d_model, d_v * n_heads)
        self.linear = nn.Linear(n_heads * d_v, d_model)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, Q, K, V, attn_mask):

        ## 这个多头分为这几个步骤，首先映射分头，然后计算atten_scores，然后计算atten_value;
        ##输入进来的数据形状： Q: [batch_size x len_q x d_model], K: [batch_size x len_k x d_model], V: [batch_size x len_k x d_model]
        residual, batch_size = Q, Q.size(0)
        # (B, S, D) -proj-> (B, S, D) -split-> (B, S, H, W) -trans-> (B, H, S, W)

        ##下面这个就是先映射，后分头；一定要注意的是q和k分头之后维度是一致额，所以一看这里都是dk
        q_s = self.W_Q(Q).view(batch_size, -1, n_heads, d_k).transpose(1,2)  # q_s: [batch_size x n_heads x len_q x d_k]
        k_s = self.W_K(K).view(batch_size, -1, n_heads, d_k).transpose(1,2)  # k_s: [batch_size x n_heads x len_k x d_k]
        v_s = self.W_V(V).view(batch_size, -1, n_heads, d_v).transpose(1,2)  # v_s: [batch_size x n_heads x len_k x d_v]

        ## 输入进行的attn_mask形状是 batch_size x len_q x len_k，然后经过下面这个代码得到 新的attn_mask : [batch_size x n_heads x len_q x len_k]，就是把pad信息重复了n个头上
        attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads, 1, 1)


        ##然后我们计算 ScaledDotProductAttention 这个函数，去7.看一下
        ## 得到的结果有两个：context: [batch_size x n_heads x len_q x d_v], attn: [batch_size x n_heads x len_q x len_k]
        context, attn = ScaledDotProductAttention()(q_s, k_s, v_s, attn_mask)
        context = context.transpose(1, 2).contiguous().view(batch_size, -1, n_heads * d_v) # context: [batch_size x len_q x n_heads * d_v]
        output = self.linear(context)
        return self.layer_norm(output + residual), attn # output: [batch_size x len_q x d_model]
```

Q最开始，我们得到的输入是什么形状呢？看注释 batch_size × len_q × d_model

K 的形状 batch_size × len_**k** × d_model、V的形状： batch_size × len_**k** × d_model

```
## 6. MultiHeadAttention
class MultiHeadAttention(nn.Module):
    def __init__(self):
        super(MultiHeadAttention, self).__init__()
        ## 输入进来的QKV是相等的，我们会使用映射linear做一个映射得到参数矩阵Wq, Wk,Wv
        self.W_Q = nn.Linear(d_model, d_k * n_heads)
        self.W_K = nn.Linear(d_model, d_k * n_heads)
        self.W_V = nn.Linear(d_model, d_v * n_heads)
        self.linear = nn.Linear(n_heads * d_v, d_model)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, Q, K, V, attn_mask):

        ## 这个多头分为这几个步骤，首先映射分头，
```

```
        ## 然后计算atten_scores，然后计算atten_value;
        ## 输入进来的数据形状：
        ## Q: [batch_size x len_q x d_model],
        ## K: [batch_size x len_k x d_model],
        ## V: [batch_size x len_k x d_model]
        residual, batch_size = Q, Q.size(0)
        # (B, S, D) -proj-> (B, S, D) -split-> (B, S, H, W) -
trans-> (B, H, S, W)

        ##下面这个就是先映射，后分头；
        # 一定要注意的是q和k分头之后维度是一致额，所以一看这里都是dk
        q_s = self.W_Q(Q).view(batch_size, -1, n_heads,
d_k).transpose(1,2)           # q_s: [batch_size x n_heads x
len_q x d_k]
        k_s = self.W_K(K).view(batch_size, -1, n_heads,
d_k).transpose(1,2)           # k_s: [batch_size x n_heads x
len_k x d_k]
        v_s = self.W_V(V).view(batch_size, -1, n_heads,
d_v).transpose(1,2)           # v_s: [batch_size x n_heads x
len_k x d_v]

        ## 输入进行的attn_mask形状是 batch_size x len_q x len_k,
        ## 然后经过下面这个代码得到 新的attn_mask : [batch_size x
n_heads x len_q x len_k]，就是把pad信息重复了n个头上
        attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads,
1, 1)


        ##然后我们计算 ScaledDotProductAttention 这个函数，去7.看一
下
        ## 得到的结果有两个：context: [batch_size x n_heads x
len_q x d_v], attn: [batch_size x n_heads x len_q x len_k]
        context, attn = ScaledDotProductAttention()(q_s, k_s,
v_s, attn_mask)
        context = context.transpose(1,
2).contiguous().view(batch_size, -1, n_heads * d_v) # context:
[batch_size x len_q x n_heads * d_v]
        output = self.linear(context)
        return self.layer_norm(output + residual), attn #
output: [batch_size x len_q x d_model]
```

首先来看 这行代码

```
q_s = self.W_Q(Q).view(batch_size, -1, n_heads,
d_k).transpose(1,2)
```

下面这个就是先映射，后分头；

首先 经过W_Q矩阵做了一个映射得到了 W_Q矩阵，然后分头，分头的时候 用view 函数，把矩阵 分成（n_heads） 8个头，每个头是 d_k 维数

同理：先映射 后分头

```
q_s = self.W_Q(Q).view(batch_size, -1, n_heads,
d_k).transpose(1,2)
k_s = self.W_K(K).view(batch_size, -1, n_heads,
d_k).transpose(1,2)
v_s = self.W_V(V).view(batch_size, -1, n_heads,
d_v).transpose(1,2)
```

这三行代码非常重要，以后在自己写代码，涉及到多头注意力机制的时候，基本上都是仿写或者原封不动 拿来主义，这里有一个细节点 就是在分头的时候，这里得到的 qk矩阵 维度一定会是相同的，要不然最后维度不相同 就不能相乘 这是需要记住的细节点

在得到 对应的查询向量，键向量、值向量，以后，看这行代码

```
attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads, 1, 1)
```

就是说把哪些符号是 PAD 信息，然后也分头，重复 n_heads 次数，给每个头一个信息，给每个头 一个信息

```
context, attn = ScaledDotProductAttention()(q_s, k_s, v_s,
attn_mask)
```

这行代码实现的函数

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

具体地函数实现：

```
## 7. ScaledDotProductAttention
class ScaledDotProductAttention(nn.Module):
    def __init__(self):
        super(ScaledDotProductAttention, self).__init__()

    def forward(self, Q, K, V, attn_mask):
        # 输入进来的维度分别是 [batch_size x n_heads x len_q x
d_k]
        # K: [batch_size x n_heads x len_k x d_k]
        # V: [batch_size x n_heads x len_k x d_v]
        # 首先经过matmul函数得到的scores形状是 :
        # [batch_size x n_heads x len_q x len_k]
        scores = torch.matmul(Q, K.transpose(-1, -2)) /
np.sqrt(d_k)

        # 然后关键词地方来了，下面这个就是用到了我们之前重点讲的
attn_mask,
        # 把被mask的地方置为无限小，softmax之后基本就是0，对q的单词不起
作用
        scores.masked_fill_(attn_mask, -1e9)
        # Fills elements of self tensor with value where mask
is one.
        attn = nn.Softmax(dim=-1)(scores)
        context = torch.matmul(attn, V)
        return context, attn
```

```
scores = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(d_k)
```

Q和K转置相乘，然后除以 根号 dk

最重要的来了，把attn_mask里面 PAD的位置 置为无穷小，使得softmax之后 为0，这样对其他单词就不会有作用

```
scores.masked_fill_(attn_mask, -1e9)
```

接着 每一横行 做softmax

```
attn = nn.Softmax(dim=-1)(scores)
```

然后 乘以对应的 v 矩阵，然后得到输出

```
context = torch.matmul(attn, V)
```

以上代码实现了功能

```
context, attn = ScaledDotProductAttention()(q_s, k_s, v_s,
attn_mask)
```

接着，后面的代码为常规操作

```
context = context.transpose(1, 2)
          .contiguous()
          .view(batch_size, -1, n_heads * d_v)
# context: [batch_size x len_q x n_heads * d_v]
output = self.linear(context)
return self.layer_norm(output + residual), attn
# output: [batch_size x len_q x d_model]
```

以上是整个encoder部分的代码，接下来是解码端的代码，解码端的代码和编码端非常的类似

```
# 9. Decoder

class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.tgt_emb = nn.Embedding(tgt_vocab_size, d_model)
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([
          DecoderLayer() for _ in range(n_layers)])
```

```
    def forward(self, dec_inputs, enc_inputs, enc_outputs):
        # dec_inputs : [batch_size x target_len]
        dec_outputs = self.tgt_emb(dec_inputs)
        # [batch_size, tgt_len, d_model]
        dec_outputs = self.pos_emb(
          dec_outputs.transpose(0,1)).transpose(0, 1)
        # [batch_size, tgt_len, d_model]

        # get_attn_pad_mask 自注意力层的时候的pad 部分
        dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs,
dec_inputs)

        # get_attn_subsequent_mask 这个做的是自注意层的mask部分
        # 就是当前单词之后看不到，使用一个上三角为1的矩阵
        dec_self_attn_subsequent_mask
=get_attn_subsequent_mask(dec_inputs)

        ## 两个矩阵相加，大于0的为1，不大于0的为0，为1的在之后就会被
fill到无限小
        dec_self_attn_mask = torch.gt(
          (dec_self_attn_pad_mask +
dec_self_attn_subsequent_mask), 0)


        ## 这个做的是交互注意力机制中的mask矩阵，
        # enc的输入是k，我去看这个k里面哪些是pad符号，给到后面的模型；
        # 注意哦，我q肯定也是有pad符号，但是这里我不在意的，之前说了好多
次了哈
        dec_enc_attn_mask = get_attn_pad_mask(dec_inputs,
enc_inputs)

        dec_self_attns, dec_enc_attns = [], []
        for layer in self.layers:
            dec_outputs, dec_self_attn, dec_enc_attn = layer(
              dec_outputs,
              enc_outputs,
              dec_self_attn_mask,
              dec_enc_attn_mask)
            dec_self_attns.append(dec_self_attn)
            dec_enc_attns.append(dec_enc_attn)

        return dec_outputs, dec_self_attns, dec_enc_attns
```

首先 init，也分为三个部分，词向量、位置编码、以及后面堆叠n个解码层

```
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.tgt_emb = nn.Embedding(tgt_vocab_size, d_model)
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([
          DecoderLayer() for _ in range(n_layers)])
```

接收的输入，编码端的输入、编码端的输出、以及解码端的输入

为什么需要编码端的输入？在交互的时候，告诉解码端，哪些是PAD符号

```
def forward(self, dec_inputs, enc_inputs, enc_outputs):
```

接下来，词嵌入 & 位置编码

```
# dec_inputs : [batch_size x target_len]
dec_outputs = self.tgt_emb(dec_inputs)
# [batch_size, tgt_len, d_model]
dec_outputs =
self.pos_emb(dec_outputs.transpose(0,1)).transpose(0, 1)
# [batch_size, tgt_len, d_model]
```

整个decoder可讲的部分其实并不多，因为整体上和encoder很类似，但是有两个地方，需要强调一下，第一个就是在做自注意力层的时候，要做两个mask，一个是PAD符号，首先解码端的输入 也有PAD符号，这是第一个mask，解码端输入的mask，第二个mask，当前单词之后，看不到的mask；所以要计算这两个部分的mask，而在做交互注意力机制的时候，只需要对编码器的PAD，哪些符号是PAD的，做mask操作就可以了

一定要记住，一个是自注意力层，一个是交互注意力层

在自注意力层做两个mask，一个是对自身pad符号的mask，一个是对当前单词之后看不到的单词的mask

在交互注意力层这一部分，只有编码层哪些部分是PAD的，那部分单词做mask操作就可以了

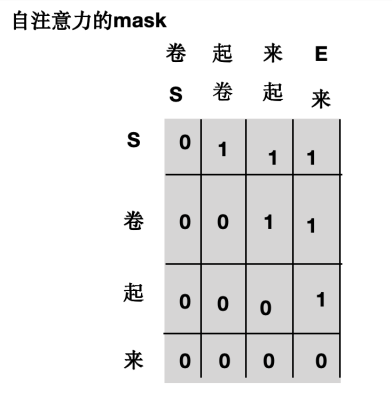
具体地代码如下：

```
dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs,
dec_inputs)
```

首先这个代码，之前已经看过了，这个是生成一个符号矩阵，输入的是dec_inputs，也就是得到的是decoder inputs里面，哪些符号是PAD的，设为1，得到一个符号矩阵，和刚才讲的一样

```
dec_enc_attn_mask = get_attn_pad_mask(dec_inputs, enc_inputs)
```

这个函数很有意思，这行代码就是把当前单词之后看不到的部分，做mask，在实现的时候，其实就是做一个上三角矩阵，上三角为1的矩阵，也就是说：



输入S在看的时候，只能看到S，看不到 卷起来；输入为 卷 的时候，只能看到 S和卷，看不到起来；所以形成一个上三角矩阵为1的矩阵

再来看：

```
dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs,
dec_inputs)
dec_enc_attn_mask = get_attn_pad_mask(dec_inputs, enc_inputs)
```

这两行代码 得到两个mask矩阵，两者相加，大于0的部分置为1，小于或者等于0的部分置为0，

```
dec_self_attn_mask = torch.gt(
          (dec_self_attn_pad_mask +
dec_self_attn_subsequent_mask), 0)
```

也就是说 经过这行代码的处理 还是得到一个符号矩阵，为1的部分是被mask的部分，为0的部分，不去做操作

```
dec_enc_attn_mask = get_attn_pad_mask(dec_inputs, enc_inputs)
```

交互注意力层 只对编码器的PAD做mask操作

```
## from https://github.com/graykode/nlp-tutorial/tree/master/5-
1.Transformer

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import math


def make_batch(sentences):
    input_batch = [[src_vocab[n] for n in
sentences[0].split()]]
    output_batch = [[tgt_vocab[n] for n in
sentences[1].split()]]
    target_batch = [[tgt_vocab[n] for n in
sentences[2].split()]]
    return torch.LongTensor(input_batch),
torch.LongTensor(output_batch), torch.LongTensor(target_batch)
```

```
## 10
def get_attn_subsequent_mask(seq):
    """
    seq: [batch_size, tgt_len]
    """
    attn_shape = [seq.size(0), seq.size(1), seq.size(1)]
    # attn_shape: [batch_size, tgt_len, tgt_len]
    subsequence_mask = np.triu(np.ones(attn_shape), k=1)  # 生
成一个上三角矩阵
    subsequence_mask =
torch.from_numpy(subsequence_mask).byte()
    return subsequence_mask  # [batch_size, tgt_len, tgt_len]


## 7. ScaledDotProductAttention
class ScaledDotProductAttention(nn.Module):
    def __init__(self):
        super(ScaledDotProductAttention, self).__init__()

    def forward(self, Q, K, V, attn_mask):
        ## 输入进来的维度分别是 [batch_size x n_heads x len_q x
d_k]  K: [batch_size x n_heads x len_k x d_k]  V: [batch_size
x n_heads x len_k x d_v]
        ##首先经过matmul函数得到的scores形状是 : [batch_size x
n_heads x len_q x len_k]
        scores = torch.matmul(Q, K.transpose(-1, -2)) /
np.sqrt(d_k)

        ## 然后关键词地方来了，下面这个就是用到了我们之前重点讲的
attn_mask，把被mask的地方置为无限小，softmax之后基本就是0，对q的单词不
起作用
        scores.masked_fill_(attn_mask, -1e9) # Fills elements
of self tensor with value where mask is one.
        attn = nn.Softmax(dim=-1)(scores)
        context = torch.matmul(attn, V)
        return context, attn


## 6. MultiHeadAttention
class MultiHeadAttention(nn.Module):
    def __init__(self):
        super(MultiHeadAttention, self).__init__()
```

```
        ## 输入进来的QKV是相等的，我们会使用映射linear做一个映射得到参数矩阵Wq, Wk,Wv
        self.W_Q = nn.Linear(d_model, d_k * n_heads)
        self.W_K = nn.Linear(d_model, d_k * n_heads)
        self.W_V = nn.Linear(d_model, d_v * n_heads)
        self.linear = nn.Linear(n_heads * d_v, d_model)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, Q, K, V, attn_mask):

        ## 这个多头分为这几个步骤，首先映射分头，然后计算atten_scores，然后计算atten_value;
        ##输入进来的数据形状： Q: [batch_size x len_q x d_model], K: [batch_size x len_k x d_model], V: [batch_size x len_k x d_model]
        residual, batch_size = Q, Q.size(0)
        # (B, S, D) -proj-> (B, S, D) -split-> (B, S, H, W) -trans-> (B, H, S, W)

        ##下面这个就是先映射，后分头；一定要注意的是q和k分头之后维度是一致额，所以一看这里都是dk
        q_s = self.W_Q(Q).view(batch_size, -1, n_heads, d_k).transpose(1,2)  # q_s: [batch_size x n_heads x len_q x d_k]
        k_s = self.W_K(K).view(batch_size, -1, n_heads, d_k).transpose(1,2)  # k_s: [batch_size x n_heads x len_k x d_k]
        v_s = self.W_V(V).view(batch_size, -1, n_heads, d_v).transpose(1,2)  # v_s: [batch_size x n_heads x len_k x d_v]

        ## 输入进行的attn_mask形状是 batch_size x len_q x len_k，然后经过下面这个代码得到 新的attn_mask : [batch_size x n_heads x len_q x len_k]，就是把pad信息重复了n个头上
        attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads, 1, 1)


        ##然后我们计算 ScaledDotProductAttention 这个函数，去7.看一下
        ## 得到的结果有两个：context: [batch_size x n_heads x len_q x d_v], attn: [batch_size x n_heads x len_q x len_k]
```

```
        context, attn = ScaledDotProductAttention()(q_s, k_s,
v_s, attn_mask)
        context = context.transpose(1,
2).contiguous().view(batch_size, -1, n_heads * d_v) # context:
[batch_size x len_q x n_heads * d_v]
        output = self.linear(context)
        return self.layer_norm(output + residual), attn #
output: [batch_size x len_q x d_model]


## 8. PoswiseFeedForwardNet
class PoswiseFeedForwardNet(nn.Module):
    def __init__(self):
        super(PoswiseFeedForwardNet, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=d_model,
out_channels=d_ff, kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=d_ff,
out_channels=d_model, kernel_size=1)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, inputs):
        residual = inputs # inputs : [batch_size, len_q,
d_model]
        output = nn.ReLU()(self.conv1(inputs.transpose(1, 2)))
        output = self.conv2(output).transpose(1, 2)
        return self.layer_norm(output + residual)



## 4. get_attn_pad_mask

## 比如说，我现在的句子长度是5，在后面注意力机制的部分，我们在计算出来QK转置除以根号之后，softmax之前，我们得到的形状
## len_input * len*input  代表每个单词对其余包含自己的单词的影响力

## 所以这里我需要有一个同等大小形状的矩阵，告诉我哪个位置是PAD部分，之后在计算计算softmax之前会把这里置为无穷大；

## 一定需要注意的是这里得到的矩阵形状是batch_size x len_q x len_k，我们是对k中的pad符号进行标识，并没有对k中的做标识，因为没必要
```

```
## seq_q 和 seq_k 不一定一致，在交互注意力，q来自解码端，k来自编码端，所以告诉模型编码这边pad符号信息就可以，解码端的pad信息在交互注意力层是没有用到的；

def get_attn_pad_mask(seq_q, seq_k):
    batch_size, len_q = seq_q.size()
    batch_size, len_k = seq_k.size()
    # eq(zero) is PAD token
    pad_attn_mask = seq_k.data.eq(0).unsqueeze(1)  # batch_size x 1 x len_k, one is masking
    return pad_attn_mask.expand(batch_size, len_q, len_k)  # batch_size x len_q x len_k


## 3. PositionalEncoding 代码实现
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()

        ## 位置编码的实现其实很简单，直接对照着公式去敲代码就可以，下面这个代码只是其中一种实现方式；
        ## 从理解来讲，需要注意的就是偶数和奇数在公式上有一个共同部分，我们使用log函数把次方拿下来，方便计算；
        ## pos代表的是单词在句子中的索引，这点需要注意；比如max_len是128个，那么索引就是从0，1，2，...,127
        ##假设我的demodel是512，2i那个符号中i从0取到了255，那么2i对应取值就是0,2,4...510
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)## 这里需要注意的是pe[:, 0::2]这个用法，就是从0开始到最后面，补长为2，其实代表的就是偶数位置
        pe[:, 1::2] = torch.cos(position * div_term)##这里需要注意的是pe[:, 1::2]这个用法，就是从1开始到最后面，补长为2，其实代表的就是奇数位置
        ## 上面代码获取之后得到的pe:[max_len*d_model]
```

```
        ## 下面这个代码之后，我们得到的pe形状是：[max_len*1*d_model]
        pe = pe.unsqueeze(0).transpose(0, 1)

        self.register_buffer('pe', pe)  ## 定一个缓冲区，其实简单理解为这个参数不更新就可以

    def forward(self, x):
        """
        x: [seq_len, batch_size, d_model]
        """
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)


## 5. EncoderLayer ：包含两个部分，多头注意力机制和前馈神经网络
class EncoderLayer(nn.Module):
    def __init__(self):
        super(EncoderLayer, self).__init__()
        self.enc_self_attn = MultiHeadAttention()
        self.pos_ffn = PoswiseFeedForwardNet()

    def forward(self, enc_inputs, enc_self_attn_mask):
        ## 下面这个就是做自注意力层，输入是enc_inputs，形状是 [batch_size x seq_len_q x d_model] 需要注意的是最初始的QKV矩阵是等同于这个输入的，去看一下enc_self_attn函数 6.
        enc_outputs, attn = self.enc_self_attn(enc_inputs, enc_inputs, enc_inputs, enc_self_attn_mask) # enc_inputs to same Q,K,V
        enc_outputs = self.pos_ffn(enc_outputs) # enc_outputs: [batch_size x len_q x d_model]
        return enc_outputs, attn


## 2. Encoder 部分包含三个部分：词向量embedding，位置编码部分，注意力层及后续的前馈神经网络

class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.src_emb = nn.Embedding(src_vocab_size, d_model)
 ## 这个其实就是去定义生成一个矩阵，大小是 src_vocab_size * d_model
```

```
        self.pos_emb = PositionalEncoding(d_model) ## 位置编码情况，这里是固定的正余弦函数，也可以使用类似词向量的nn.Embedding获得一个可以更新学习的位置编码
        self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)]) ## 使用ModuleList对多个encoder进行堆叠，因为后续的encoder并没有使用词向量和位置编码，所以抽离出来；

    def forward(self, enc_inputs):
        ## 这里我们的 enc_inputs 形状是： [batch_size x source_len]

        ## 下面这个代码通过src_emb，进行索引定位，enc_outputs输出形状是[batch_size, src_len, d_model]
        enc_outputs = self.src_emb(enc_inputs)

        ## 这里就是位置编码，把两者相加放入到了这个函数里面，从这里可以去看一下位置编码函数的实现；3.
        enc_outputs = self.pos_emb(enc_outputs.transpose(0, 1)).transpose(0, 1)

        ##get_attn_pad_mask是为了得到句子中pad的位置信息，给到模型后面，在计算自注意力和交互注意力的时候去掉pad符号的影响，去看一下这个函数 4.
        enc_self_attn_mask = get_attn_pad_mask(enc_inputs, enc_inputs)
        enc_self_attns = []
        for layer in self.layers:
            ## 去看EncoderLayer 层函数 5.
            enc_outputs, enc_self_attn = layer(enc_outputs, enc_self_attn_mask)
            enc_self_attns.append(enc_self_attn)
        return enc_outputs, enc_self_attns

## 10.
class DecoderLayer(nn.Module):
    def __init__(self):
        super(DecoderLayer, self).__init__()
        self.dec_self_attn = MultiHeadAttention()
        self.dec_enc_attn = MultiHeadAttention()
        self.pos_ffn = PoswiseFeedForwardNet()

    def forward(self, dec_inputs, enc_outputs, dec_self_attn_mask, dec_enc_attn_mask):
```

```
        dec_outputs, dec_self_attn =
self.dec_self_attn(dec_inputs, dec_inputs, dec_inputs,
dec_self_attn_mask)
        dec_outputs, dec_enc_attn =
self.dec_enc_attn(dec_outputs, enc_outputs, enc_outputs,
dec_enc_attn_mask)
        dec_outputs = self.pos_ffn(dec_outputs)
        return dec_outputs, dec_self_attn, dec_enc_attn

## 9. Decoder

class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.tgt_emb = nn.Embedding(tgt_vocab_size, d_model)
        self.pos_emb = PositionalEncoding(d_model)
        self.layers = nn.ModuleList([DecoderLayer() for _ in
range(n_layers)])

    def forward(self, dec_inputs, enc_inputs, enc_outputs): #
dec_inputs : [batch_size x target_len]
        dec_outputs = self.tgt_emb(dec_inputs)  # [batch_size,
tgt_len, d_model]
        dec_outputs = self.pos_emb(dec_outputs.transpose(0,
1)).transpose(0, 1) # [batch_size, tgt_len, d_model]

        ## get_attn_pad_mask 自注意力层的时候的pad 部分
        dec_self_attn_pad_mask = get_attn_pad_mask(dec_inputs,
dec_inputs)

        ## get_attn_subsequent_mask 这个做的是自注意层的mask部分，
就是当前单词之后看不到，使用一个上三角为1的矩阵
        dec_self_attn_subsequent_mask =
get_attn_subsequent_mask(dec_inputs)

        ## 两个矩阵相加，大于0的为1，不大于0的为0，为1的在之后就会被
fill到无限小
        dec_self_attn_mask = torch.gt((dec_self_attn_pad_mask +
dec_self_attn_subsequent_mask), 0)
```

```
        ## 这个做的是交互注意力机制中的mask矩阵，enc的输入是k，我去看这个k里面哪些是pad符号，给到后面的模型；注意哦，我q肯定也是有pad符号，但是这里我不在意的，之前说了好多次了哈
        dec_enc_attn_mask = get_attn_pad_mask(dec_inputs, enc_inputs)

        dec_self_attns, dec_enc_attns = [], []
        for layer in self.layers:
            dec_outputs, dec_self_attn, dec_enc_attn = layer(dec_outputs, enc_outputs, dec_self_attn_mask, dec_enc_attn_mask)
            dec_self_attns.append(dec_self_attn)
            dec_enc_attns.append(dec_enc_attn)
        return dec_outputs, dec_self_attns, dec_enc_attns


## 1. 从整体网路结构来看，分为三个部分：编码层，解码层，输出层
class Transformer(nn.Module):
    def __init__(self):
        super(Transformer, self).__init__()
        self.encoder = Encoder()  ## 编码层
        self.decoder = Decoder()  ## 解码层
        self.projection = nn.Linear(d_model, tgt_vocab_size, bias=False) ## 输出层 d_model 是我们解码层每个token输出的维度大小，之后会做一个 tgt_vocab_size 大小的softmax
    def forward(self, enc_inputs, dec_inputs):
        ## 这里有两个数据进行输入，一个是enc_inputs 形状为[batch_size, src_len]，主要是作为编码段的输入，一个dec_inputs，形状为[batch_size, tgt_len]，主要是作为解码端的输入

        ## enc_inputs作为输入 形状为[batch_size, src_len]，输出由自己的函数内部指定，想要什么指定输出什么，可以是全部tokens的输出，可以是特定每一层的输出；也可以是中间某些参数的输出；
        ## enc_outputs就是主要的输出，enc_self_attns这里没记错的是QK转置相乘之后softmax之后的矩阵值，代表的是每个单词和其他单词相关性；
        enc_outputs, enc_self_attns = self.encoder(enc_inputs)

        ## dec_outputs 是decoder主要输出，用于后续的linear映射；dec_self_attns类比于enc_self_attns 是查看每个单词对decoder中输入的其余单词的相关性；dec_enc_attns是decoder中每个单词对encoder中每个单词的相关性；
```

```
        dec_outputs, dec_self_attns, dec_enc_attns =
self.decoder(dec_inputs, enc_inputs, enc_outputs)

        ## dec_outputs做映射到词表大小
        dec_logits = self.projection(dec_outputs) # dec_logits
: [batch_size x src_vocab_size x tgt_vocab_size]
        return dec_logits.view(-1, dec_logits.size(-1)),
enc_self_attns, dec_self_attns, dec_enc_attns



if __name__ == '__main__':

    ## 句子的输入部分,
    sentences = ['ich mochte ein bier P', 'S i want a beer', 'i
want a beer E']


    # Transformer Parameters
    # Padding Should be Zero
    ## 构建词表
    src_vocab = {'P': 0, 'ich': 1, 'mochte': 2, 'ein': 3,
'bier': 4}
    src_vocab_size = len(src_vocab)

    tgt_vocab = {'P': 0, 'i': 1, 'want': 2, 'a': 3, 'beer': 4,
'S': 5, 'E': 6}
    tgt_vocab_size = len(tgt_vocab)

    src_len = 5 # length of source
    tgt_len = 5 # length of target

    ## 模型参数
    d_model = 512  # Embedding Size
    d_ff = 2048  # FeedForward dimension
    d_k = d_v = 64  # dimension of K(=Q), V
    n_layers = 6  # number of Encoder of Decoder Layer
    n_heads = 8  # number of heads in Multi-Head Attention

    model = Transformer()

    criterion = nn.CrossEntropyLoss()
```

```
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    enc_inputs, dec_inputs, target_batch =
make_batch(sentences)

    for epoch in range(20):
        optimizer.zero_grad()
        outputs, enc_self_attns, dec_self_attns, dec_enc_attns
= model(enc_inputs, dec_inputs)
        loss = criterion(outputs,
target_batch.contiguous().view(-1))
        print('Epoch:', '%04d' % (epoch + 1), 'cost =',
'{:.6f}'.format(loss))
        loss.backward()
        optimizer.step()
```