时间线：

- 241021 开始【20、Transformer模型Decoder原理精讲及其PyTorch逐行实现-哔哩哔哩】 https://b23.tv/ImjriT9　59分钟的视频
- 241024
- 241025
- 241026 done

这部分内容继续是Transformer难点细节的部分，之前是word embedding、position embedding、encoder self attention mask，回顾首先是配置文件：

```
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F

# 关于word embedding，以序列建模为例
# 考虑source sentence 和 target sentence
# 构建序列，序列的字符以其在词表中的索引的形式表示
batch_size = 2

# 单词表大小
max_num_src_words = 8
max_num_tgt_words = 8
model_dim = 8

# 序列的最大长度
max_src_seq_len = 5
max_tgt_seq_len = 5
max_position_len = 5

#src_len = torch.randint(2, 5, (batch_size,))
#tgt_len = torch.randint(2, 5, (batch_size,))
src_len = torch.Tensor([2, 4]).to(torch.int32)
tgt_len = torch.Tensor([4, 3]).to(torch.int32)
```

接下来 第一步 源句子和目标句子：

```
# Step1: 单词索引构成源句子和目标句子，构建batch，并且做了padding，默认值为0
src_seq = torch.cat([torch.unsqueeze(F.pad(torch.randint(1, max_num_src_words, (L,)), (0, max(src_len)-L)), 0) \
                     for L in src_len])
tgt_seq = torch.cat([torch.unsqueeze(F.pad(torch.randint(1, max_num_tgt_words, (L,)), (0, max(tgt_len)-L)), 0) \
                     for L in tgt_len])
```

根据单词索引 构建源句子和目标句子 并且构建batch，然后做padding，默认为0，具体的做法，首先定义常量，比如batch size、单词表大小 max_num_src_words、模型的特定维度model_dim、序列的最大长度max_src_seq_len、位置的最大长度max_position_len，构造两个样本，每个样本的长度分别是2和4，这是源样本，目标样本是4和3，接下来随机构建两个源序列和目标序列，构建方法是根据randint函数生成，生成L个单词，并且做pad，这样就完成了构建序列的部分。

```
# Step2: 构造word embedding
src_embedding_table = nn.Embedding(max_num_src_words+1, model_dim)
tgt_embedding_table = nn.Embedding(max_num_tgt_words+1, model_dim)
src_embedding = src_embedding_table(src_seq)
tgt_embedding = tgt_embedding_table(tgt_seq)
```

序列得到以后，构建embedding，实例化两个Embedding的实例，相当于构建两个weight矩阵，然后我们把源序列的索引和目标序列的索引传入进来，得到 source embedding 和 target embedding，以上构建了词向量，接下来 我们构建 位置 向量

```
# Step3: 构造position embedding
pos_mat = torch.arange(max_position_len).reshape((-1, 1))
i_mat = torch.pow(10000, torch.arange(0, 8, 2).reshape((1, -1))/model_dim)
pe_embedding_table = torch.zeros(max_position_len, model_dim)
pe_embedding_table[:, 0::2] = torch.sin(pos_mat / i_mat)
pe_embedding_table[:, 1::2] = torch.cos(pos_mat / i_mat)

pe_embedding = nn.Embedding(max_position_len, model_dim)
pe_embedding.weight = nn.Parameter(pe_embedding_table, requires_grad=False)

src_pos = torch.cat([torch.unsqueeze(torch.arange(max(src_len)),0) for _ in src_len]).to(torch.int32)
tgt_pos = torch.cat([torch.unsqueeze(torch.arange(max(tgt_len)),0) for _ in tgt_len]).to(torch.int32)

src_pe_embedding = pe_embedding(src_pos)
tgt_pe_embedding = pe_embedding(tgt_pos)
```

位置向量，再去回顾论文

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

位置向量由两个三角函数构成的，特征维度是偶数维的话，用sin函数，特征维度是奇数维的话，用cos函数，里面都是一样的，构建position embedding并没有通过原始的赋值方法 for loop构建的，我们是通过两个矩阵相乘的方式构建的，分别构建了一个pos矩阵和i矩阵，pos矩阵`pos_mat = torch.arange(max_position_len).reshape((-1,1))`反映的是公式中pos变量的变化，每一行的pos值是不一样的，

i矩阵`i_mat = torch.pow(10000,torch.arange(0,8,2).reshape((1,-1))/model_dim)`反映的是特征维度的变化，每一列i的值是不一样的，然后将这两个矩阵，利用广播机制相乘或者叫相除`pe_embedding_table = torch.zeros(max_position_len,model_dim)`

`pe_embedding_table[:,0::2] = torch.sin(pos_mat / i_mat)`

`pe_embedding_table[:,1::2] = torch.cos(pos_mat / i_mat)`

得到最终的 pe_embedding_table，上面这几句 只是得到了 括号里面的部分 我们还要对括号里面的部分 取一个 sin 和cos，再赋给偶数[:,0::2] 特征维 或者 奇数[:,1::2]特征维

接下来 利用 pytorch的embedding的class构建一个pe embedding的实例：

`pe_embedding = nn.Embedding(max_position_len,model_dim)`

修改 weight，梯度设置成false

```
pe_embedding.weight =
nn.Parameter(pe_embedding_table,requires_grad=False)

src_pos = torch.cat([torch.unsqueeze(torch.arange(max(src_len),0)
for _ in src_len)]).to(torch.int32)

tgt_pos = torch.cat([torch.unsqueeze(torch.arange(max(tgt_len),0)
for _ in tgt_len)]).to(torch.int32)

src_pe_embedding = pe_embedding(src_pos)

tgt_pe_embedding = pe_embedding(tgt_pos)
```

这几行把位置索引传入到embedding中，得到source pe embedding和target pe embedding

第四步 构建encoder的self attention mask，因为encoder这部分 是完整序列的一个输入，是自身对自身的注意力机制的运算，所以mask构建的是一个邻接矩阵，mask是一个矩阵，每一行反映的是一个单词对所有单词的是否有效，不能说是关联性，是有效性，因为如果遇到pad的话，说明是无效的

```
# Step4: 构造encoder的self-attention mask
# mask的shape: [batch_size, max_src_len, max_src_len],值为1或-inf
valid_encoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(src_len)-L)),0) \
                                               for L in src_len]), 2)
valid_encoder_pos_matrix = torch.bmm(valid_encoder_pos, valid_encoder_pos.transpose(1, 2))
invalid_encoder_pos_matrix = 1-valid_encoder_pos_matrix
mask_encoder_self_attention = invalid_encoder_pos_matrix.to(torch.bool)

score = torch.randn(batch_size, max(src_len), max(src_len))
#print(score.shape, mask_encoder_self_attention.shape)

masked_score = score.masked_fill(mask_encoder_self_attention, -1e9)
prob = F.softmax(masked_score, -1)

print(src_len)
print(score)
print(masked_score)
print(prob)
```

所以，首先构建一个有效的矩阵，矩阵的维度是batch×max source len

当然，为了构造一个T×T的这样一个有效矩阵，必须要把矩阵变成 batch×1，因为有效维度只针对t这一维度进行相乘 batch这一维度是不用进行相乘的，基于bmm函数 计算 invalid encoder pos matrix；接下来1-有效矩阵 得到无效矩阵，转换成bool类型 得到mask矩阵；mask矩阵跟Q乘K的转置 维度是一样的，接下来进行相乘 或者叫 element product的操作，接下来看到score加mask之前 和 之后的操作；mask之后的数 都变成 负无穷的

```
tensor([2, 4], dtype=torch.int32)
tensor([[[ 0.3340,  2.2387, -0.1092,  0.5382],
         [ 0.6673,  0.9443,  1.0673, -0.3969],
         [-0.4240, -0.5947, -0.5410, -0.5625],
         [-0.0996, -0.0271,  1.7454,  0.6641]],

        [[-0.7846, -1.5655,  1.0464, -0.3124],
         [-0.0495,  0.3757, -0.2082, -1.4990],
         [-0.7114,  0.2253,  1.9388,  0.0783],
         [-0.2760, -1.0927, -2.2223, -1.3172]]])
tensor([[[ 3.3403e-01,  2.2387e+00, -1.0000e+09, -1.0000e+09],
         [ 6.6732e-01,  9.4426e-01, -1.0000e+09, -1.0000e+09],
         [-1.0000e+09, -1.0000e+09, -1.0000e+09, -1.0000e+09],
         [-1.0000e+09, -1.0000e+09, -1.0000e+09, -1.0000e+09]],

        [[-7.8462e-01, -1.5655e+00,  1.0464e+00, -3.1242e-01],
         [-4.9462e-02,  3.7565e-01, -2.0817e-01, -1.4990e+00],
         [-7.1140e-01,  2.2531e-01,  1.9388e+00,  7.8335e-02],
         [-2.7596e-01, -1.0927e+00, -2.2223e+00, -1.3172e+00]]])
tensor([[[0.1296, 0.8704, 0.0000, 0.0000],
         [0.4312, 0.5688, 0.0000, 0.0000],
         [0.2500, 0.2500, 0.2500, 0.2500],
         [0.2500, 0.2500, 0.2500, 0.2500]],

        [[0.1075, 0.0492, 0.6709, 0.1724],
         [0.2764, 0.4229, 0.2359, 0.0649],
         [0.0502, 0.1282, 0.7110, 0.1106],
         [0.5161, 0.2280, 0.0737, 0.1822]]])
```

再经过softmax变成0，如果这一行全部都是被mask的，那么概率就是均匀地，不过没关系因为最终在loss的计算时，也会加一个mask

以上是上次讲解的所有内容

为什么要用正余弦的位置编码?

是为了具有泛化能力，证明如下：

For every sine-cosine pair corresponding to frequency ω_k, there is a linear transformation $M \in \mathbb{R}^{2\times 2}$ (independent of t) where the following equation holds:

$$M.\begin{bmatrix}\sin(\omega_k.t)\\ \cos(\omega_k.t)\end{bmatrix} = \begin{bmatrix}\sin(\omega_k.(t+\phi))\\ \cos(\omega_k.(t+\phi))\end{bmatrix}$$

Proof:

Let M be a 2×2 matrix, we want to find u_1, v_1, u_2 and v_2 so that:

$$\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} . \begin{bmatrix} \sin(\omega_k . t) \\ \cos(\omega_k . t) \end{bmatrix} = \begin{bmatrix} \sin(\omega_k . (t + \phi)) \\ \cos(\omega_k . (t + \phi)) \end{bmatrix}$$

By applying the addition theorem, we can expand the right hand side as follows:

$$\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} . \begin{bmatrix} \sin(\omega_k . t) \\ \cos(\omega_k . t) \end{bmatrix} = \begin{bmatrix} \sin(\omega_k . t) \cos(\omega_k . \phi) + \cos(\omega_k . t) \sin(\omega_k . \phi) \\ \cos(\omega_k . t) \cos(\omega_k . \phi) - \sin(\omega_k . t) \sin(\omega_k . \phi) \end{bmatrix}$$

Which result in the following two equations:

$$\begin{aligned} u_1 \sin(\omega_k . t) + v_1 \cos(\omega_k . t) &= \cos(\omega_k . \phi) \sin(\omega_k . t) + \sin(\omega_k . \phi) \cos(\omega_k . t) && (1) \\ u_2 \sin(\omega_k . t) + v_2 \cos(\omega_k . t) &= -\sin(\omega_k . \phi) \sin(\omega_k . t) + \cos(\omega_k . \phi) \cos(\omega_k . t) && (2) \end{aligned}$$

By solving above equations, we get:

$$\begin{aligned} u_1 &= \cos(\omega_k . \phi) & v_1 &= \sin(\omega_k . \phi) \\ u_2 &= -\sin(\omega_k . \phi) & v_2 &= \cos(\omega_k . \phi) \end{aligned}$$

So the final transformation matrix M is:

$$M_{\phi,k} = \begin{bmatrix} \cos(\omega_k . \phi) & \sin(\omega_k . \phi) \\ -\sin(\omega_k . \phi) & \cos(\omega_k . \phi) \end{bmatrix}$$

w_k 是什么呢？

对照公式：
$$\begin{cases} PE(pos, 2i) = \sin(pos / 10000^{2i/d_{model}}) \\ PE(pos, 2i+1) = \cos(pos / 10000^{2i/d_{model}}) \end{cases}$$

w_k 是 $1 / 10000^{2i/d_{model}}$

pos 是 t or $t+\phi$

解释的是这：
$$M \cdot \begin{bmatrix} \sin(w_k \cdot t) \\ \cos(w_k \cdot t) \end{bmatrix} = \begin{bmatrix} \sin(w_k(t+\phi)) \\ \cos(w_k(t+\phi)) \end{bmatrix}$$

解释证明过程：

transformer 的位置编码有1个好处，即便设置的 Position embedding 最大长度为 2000，如果在 Inference 阶段遇到了 2000 的长度，该怎么办呢？ pos=2001 的时候，可以用 pos=2000 的数进行线性组合得到，而不必担心 M 超出范围的问题

$$M \begin{bmatrix} \sin(w_k t) \\ \cos(w_k t) \end{bmatrix} = \begin{bmatrix} \sin w_k(t+\phi) \\ \cos w_k(t+\phi) \end{bmatrix}$$

如果我们要计算 $PE_{t+\phi}$ $(t+\phi)$ 比较大的位置的表示，我们可以用 PE_t 时刻的 自的线性组合表示。

假设线性组合的系数是 2×2 的 2维矩阵，并且设元素矩阵分别为 v_1 v_2 u_1 u_2 形成矩阵相乘的公式

$$\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} \begin{bmatrix} \sin w_k t \\ \cos w_k t \end{bmatrix} = \begin{bmatrix} \sin w_k(t+\phi) \\ \cos w_k(t+\phi) \end{bmatrix}$$

然后 我们把右边的矩阵 用三角函数展开

$$\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} \begin{bmatrix} \sin w_k t \\ \cos w_k t \end{bmatrix} = \begin{bmatrix} \sin w_k t \cos w_k\phi + \cos w_k t \sin w_k\phi \\ \cos w_k t \cos w_k\phi - \sin w_k t \sin w_k\phi \end{bmatrix}$$

现有 4个未知数 u_1 u_2 v_1 v_2，2个等式

∴ 这个方程的解不唯一

仔细对比 方程的左右两边

$$\begin{bmatrix} u_1 \sin w_k t + v_1 \cos w_k t \\ u_2 \sin w_k t + v_2 \cos w_k t \end{bmatrix} = \begin{bmatrix} \sin w_k t \cos w_k\phi + \cos w_k t \sin w_k\phi \\ \cos w_k t \cos w_k\phi - \sin w_k t \sin w_k\phi \end{bmatrix}$$

u_1 可取 $\cos w_k\phi$ v_1 可取 $\sin w_k\phi$ 这4个值都是确定的

u_2 取 $-\sin w_k\phi$ v_2 可取 $\cos w_k\phi$

$u_1 = \cos w_k\phi$ $\quad v_1 = \sin w_k\phi$

$u_2 = -\sin w_k\phi$ $\quad v_2 = \cos w_k\phi$

∴ 要 M 可取
$$M_{\phi,k} = \begin{bmatrix} \cos w_k\phi & \sin w_k\phi \\ -\sin w_k\phi & \cos w_k\phi \end{bmatrix}$$

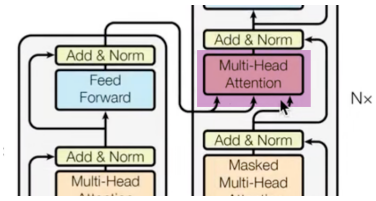
我们就可以用 PE 长度为 t 的位置编码 的线性组合表示 $t+\phi$ 位置的编码

得到正余弦 这种位置的推导

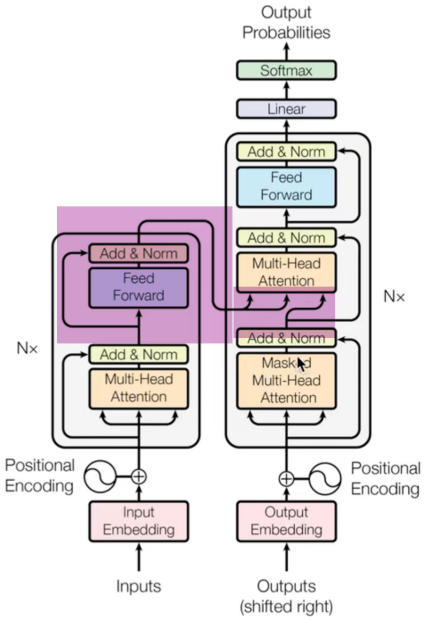
[done]

ϕ 的作用就是为了表明 如果 Inference 遇到的序列长度如果超过了 训练阶段的长度，我们可以用已用位置的线性组合找到其位置编码的表示

接下来 开始intra attention mask的讲解，就是对应论文中的这里：



首先，讲解decoder block的结构，decoder block分为三部分，第一部分是目标序列对目标序列自身的的MMHA掩码 多头自注意力，第二部分是目标序列 对原序列的MHA，第三部分是前馈全连接层，着重第二部分，第二部分是让目标序列的输出（MMHA）作为query，在query下，可以基于encoder的输出，称为memory，把memory当做key和value，query和key算出一个score，经过softmax函数计算出一个weight，再把权重 跟 value 进行一个加权求和，得到一个新的表征，这个表征 反映的是 目标序列跟源序列的一个相关性的表征



这里也会有一个mask，因为目标序列 每个样本的长度是不一样的，同时原序列的样本长度也是不一样的，而且一对之间 长度也是不一样的，所以需要一个mask 将原序列中某个单词某个位置 跟 目标序列中 某个位置 如果它们之间 有一个是pad的话 说明是无效字符，得到这样的掩码矩阵【？】

代码 构造 intra attention mask，首先公式Q乘以K的转置，首先明确Q乘以K的转置的shape是target len×source len，如果有batch size的话，那就是 batch size×target sequence len × source sequence len，所以我们构造的mask的形状也是这样的，那么怎么做呢？仿照 encoder self attention mask，我们可以看到

```
valid_encoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(src_len)-L)),0) \
                                               for L in src_len]), 2)
```

encoder self attention mask的第一步 是构造一个有效的encoder pos，复制下来：

```
# Step5: 构造intra-attention的mask
# Q @ K^T shape: [batch_size, tgt_seq_len, src_seq_len]
valid_encoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(src_len)-L)),0) \
                                               for L in src_len]), 2)
print(valid_encoder_pos)
```

```
tensor([[[1.],
         [1.],
         [0.],
         [0.]],

        [[1.],
         [1.],
         [1.],
         [1.]]])
```

打印看是什么东西，我们encoder的输入 batch size也就是 句子数=2，其中我们hard code 第一个句子长度=2，第二个句子长度=4，1代表单词有效，0代表pad字符；仿照valid encoder pos，我们构造valid decoder pos，对照着 改 src len改成tgt len，同样打印valid decoder pos：

```
# Step5: 构造intra-attention的mask
# Q @ K^T shape: [batch_size, tgt_seq_len, src_seq_len]
valid_encoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(src_len)-L)),0) \
                                               for L in src_len]), 2)
valid_decoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) \
                                               for L in tgt_len]), 2)
print(valid_encoder_pos)
print(valid_decoder_pos)
```

```
tensor([[[1.],
         [1.],
         [0.],
         [0.]],

        [[1.],
         [1.],
         [1.],
         [1.]]])
tensor([[[1.],
         [1.],
         [1.],
         [1.]],

        [[1.],
         [1.],
         [1.],
         [0.]]])
```

decoder第一个序列 长度=4，所以第二个tensor第一个句子长度=4，全是1，第二个tensor第二个句子长度=3，pad成4，有一个0，代表pad字符，得到这两个矩阵以后，进行batch的矩阵相乘，得到相关位置的邻接矩阵，首先 为了看着方便，打印shape，看到各自的shape都是[2,4,1]

```
# Step5: 构造intra-attention的mask
# Q @ K^T shape: [batch_size, tgt_seq_len, src_seq_len]
valid_encoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(src_len)-L)),0) \
                                               for L in src_len]), 2)
valid_decoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) \
                                               for L in tgt_len]), 2)
print(valid_encoder_pos.shape)
print(valid_decoder_pos.shape)
```

```
torch.Size([2, 4, 1])
torch.Size([2, 4, 1])
```

2是batch size；4是最大序列长度；1是为了做矩阵运算 扩维的，矩阵运算时 batch维度是不会参与的，只是sequence len这个维度参与的；同一个source对应着的同一个target进行有效矩阵的运算

接下来计算有效的交叉矩阵，torch.bmm函数 表示 batch的矩阵相乘没接下来传入valid decoder pos乘以 valid encoder pos，直接相乘是不行的 还需要 转置，第一维和第二维进行转置，前面矩阵的维度是241，后面矩阵的维度是214，这样做矩阵乘法变成244的张量，打印查看结果：

```
# Step5: 构造intra-attention的mask
# Q @ K^T shape: [batch_size, tgt_seq_len, src_seq_len]
valid_encoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(src_len)-L)),0) \
                                               for L in src_len]), 2)
valid_decoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) \
                                               for L in tgt_len]), 2)
valid_cross_pos = torch.bmm(valid_decoder_pos, valid_encoder_pos.transpose(1, 2))
print(valid_cross_pos)
```

```
tensor([[[1., 1., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 0., 0.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [0., 0., 0., 0.]]])
```

这样出现了 有效的位置，为了理解的直观 我们再打印：

```
print(valid_encoder_pos, valid_decoder_pos, valid_cross_pos)
```

```
tensor([[[1.],
         [1.],
         [0.],
         [0.]],

        [[1.],
         [1.],
         [1.],
         [1.]]]) tensor([[[1.],
         [1.],
         [1.],
         [1.]],

        [[1.],
         [1.],
         [1.],
         [0.]]]) tensor([[[1., 1., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 0., 0.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [0., 0., 0., 0.]]])
```

三个张量

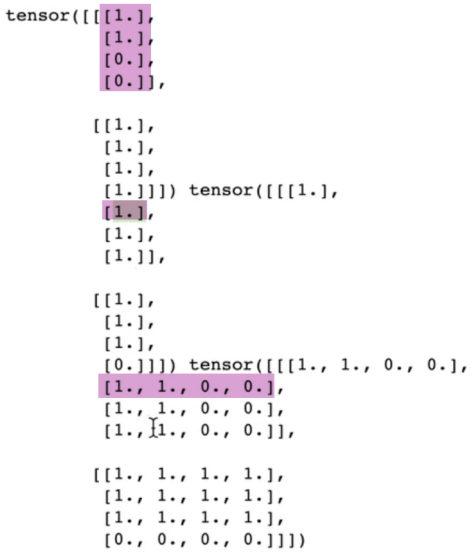
第一个张量 表示原序列的有效位置；

第二个张量 表示目标张量的有效位置；

第三个张量 表示目标序列 对源序列的有效性的关系

验证一下：首先第三个张量 由两个部分组成，第一个部分是第一个样本的，第二个部分是第二个样本的；第一个部分的第一行 反映的是目标序列的第一个单词对源序列 的 第一个序列的有效性，具体来说，

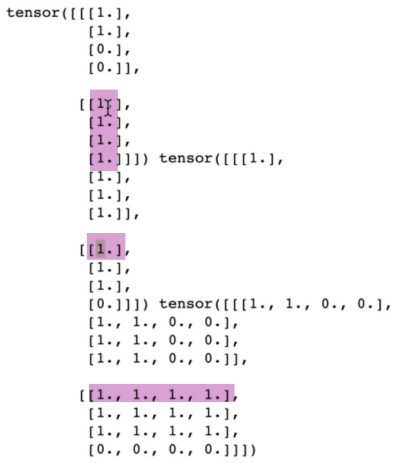
目标序列的第一个样本的第一个位置是1 表示这个位置是 有效的，源序列的第一个句子是1100，只有前两个是有效的，那么一交互一下，就得到了交互有效位置的 1100；



第二行1100表示；目标序列的第2个单词 对 源序列的第一个句子 同样也是 1100；

后面的两行1100也是同样的理解，因为目标序列的第一个句子都是有效位置，所以直接把源序列的有效性 直接 copy过来就可以了

这样 我们解释好了 第三个tensor的第一个部分



目标序列的第二个句子的第一个单词1，对源序列第二个句子1111的有效性

目标序列的第二个单词的前三个单词都是1，所以都是1111

最后 目标序列的最后一个单词是0 无效的，所以不管源序列 最后都是0

总之vliad cross pos反映的就是原序列跟目标序列的有效性 判断是否是有效的

计算好了 有效的 接下来 计算无效的 再转成 bool类型（计算有效 计算无效 布尔类型）

```
# Step5: 构造intra-attention的mask
# Q @ K^T shape: [batch_size, tgt_seq_len, src_seq_len]
valid_encoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(src_len)-L)),0) \
                                    for L in src_len]), 2)
valid_decoder_pos = torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L), (0, max(tgt_len)-L)),0) \
                                    for L in tgt_len]), 2)
valid_cross_pos_matrix = torch.bmm(valid_decoder_pos, valid_encoder_pos.transpose(1, 2))
invalid_cross_pos_matrix = 1-valid_cross_pos_matrix
mask_cross_attention = invalid_cross_pos_matrix.to(torch.bool)
print(mask_cross_attention)
```

```
tensor([[[False, False,  True,  True],
         [False, False,  True,  True],
         [False, False,  True,  True],
         [False, False,  True,  True]],

        [[False, False, False, False],
         [False, False, False, False],
         [False, False, False, False],
         [ True,  True,  True,  True]]])
```

```
# step5: 构造intra attention 的mask
# Q @ K^T shape:[batch_size ,tgt_seq_len,src_seq_len]

valid_encoder_pos =
torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L),
(0,max(src_len)-L)),0) for L in src_len]),2)

valid_decoder_pos =
torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L),
(0,max(tgt_len)-L)),0) for L in tgt_len]),2)

valid_cross_pos_matrix =
torch.bmm(valid_decoder_pos,valid_encoder_pos.transpose(1,2))
invalid_cross_pos_matrix = 1- valid_cross_pos_matrix
mask_cross_self_attention = invalid_cross_pos_matrix.to(torch.bool)
print(mask_cross_self_attention)
```

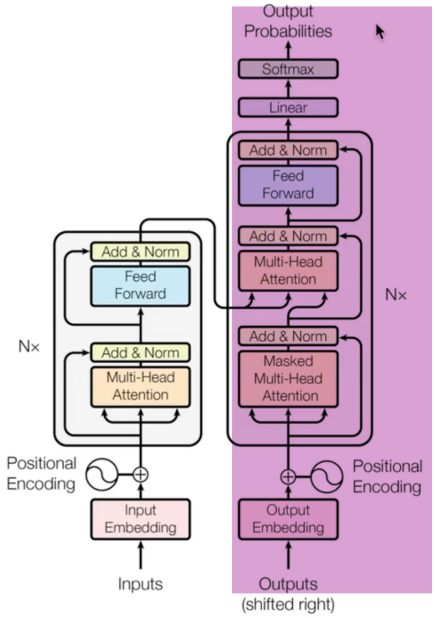
得到这个mask以后 对 score进行mask计算，具体地做法同样调用score.masked_fill函数，把attention为false的位置 都填充一个负无穷 很小的数 逻辑是一样 不再演示了

```
# 构造 encoder的self attention mask
# mask的shape: [batch_size,max_src_len,max_src_len], 值为1或-inf

valid_encoder_pos =
torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L),
(0,max(src_len)-L)),0) for L in src_len]),2)

valid_encoder_pos_matrix =
torch.bmm(valid_encoder_pos,valid_encoder_pos.transpose(1,2))
```

```
invalid_encoder_pos_matrix = 1-valid_encoder_pos_matrix

mask_encoder_self_attention = invalid_encoder_pos_matrix.to(torch.bool)

score = torch.randn(batch_size,max(src_len),max(src_len))
# print(score.shape,mask_encoder_self_attention.shape)

masked_score = score.masked_fill(mask_encoder_self_attention,-1e9)
prob = F.softmax(masked_score,-1)

# step5: 构造intra attention 的mask
# Q @ K^T shape:[batch_size ,tgt_seq_len,src_seq_len]

valid_encoder_pos =
torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L),
(0,max(src_len)-L)),0) for L in src_len]),2)

valid_decoder_pos =
torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L),
(0,max(tgt_len)-L)),0) for L in tgt_len]),2)

valid_cross_pos_matrix =
torch.bmm(valid_decoder_pos,valid_encoder_pos.transpose(1,2))
invalid_cross_pos_matrix = 1- valid_cross_pos_matrix
mask_cross_self_attention = invalid_cross_pos_matrix.to(torch.bool)
print(mask_cross_self_attention)
```

接下来 decoder self attention mask

这里的decoder self attention mask跟 encoder self attention 和intra attention都不太一样，因为原文的Transformer这个模型是一个自回归的解码模型，那什么叫自回归呢？就是在推理的部分 output不是一次性的把目标序列预测出来 而是第一次预测一个 然后把预测的那个送回来 作为输入；第二次呢 就是预测下一个 再送回来 再预测下一个 就是每一个单词都会建立在上一个单词预测的基础上 进行预测 可以理解为自回归 或者 条件概率的生成；然后为了让训练阶段可以和推理阶段保持一致（对着图理解）



那在训练阶段 需要对decoder的输入进行一定的遮掩，也就是在inference阶段看到几个 那么在训练阶段也就看见几个 不能看太多 看太多的话 就会违背 因果 ；相当于把答案告诉模型了，这就没有意义了；所以训练阶段 需要把每个位置的答案给遮住；然后让 decoder output预测答案，这样和inference保持一致，所以 decoder self attention 的mask是一个三角形的；可以理解为在 decoder预测第一步的时候，预测的第一步是一个特殊字符，当decoder预测第二个位置的时候，给出特殊字符和第一个目标值，所以应该是个三角形；具体的实现：

用到的api torch.tril(l表示 lower u表示 upper)l代表 下三角矩阵 u代表上三角矩阵

先考虑 单个序列；单个序列：for L in tgt_len;L指的是 每一个目标序列的长度 ，对每个序列 构建三角矩阵；暂时不考虑左上、右上三角等，首先 构建一个方阵，并打印 查看一下，看看样子：

```
# Step6: 构造decoder self-attention的mask
tri_matrix = [torch.tril(torch.ones((L, L))) for L in tgt_len]
print(tri_matrix)
```

```
[tensor([[1., 0., 0., 0.],
        [1., 1., 0., 0.],
        [1., 1., 1., 0.],
        [1., 1., 1., 1.]]), tensor([[1., 0., 0.],
        [1., 1., 0.],
        [1., 1., 1.]])]
```

可以看到 首先构建了一个下三角矩阵，有两个张量 第一个张量 是4×4的下三角矩阵；第二个张量是3×3的，因为第一个序列 长度是4，第二个序列 长度是3

刚好这个矩阵都是满足要求的。

解码器在预测第一个位置的时候，解码器的输入 只给一个 特殊字符，因为解码器的输入和解码器的输出是有一个shift的，也就是解码器的输入 左 shift一位，这样刚好跟输出 有一个偏移；

当解码器预测第二个位置的时候，解码器的输入给的是 一个特殊字符和 第一个字符；

当解码器预测第三个位置的时候，解码器的输入给的是 特殊字符、第一个字符、第二个字符，让解码器预测第三个字符，总之就是这样的一个因果性；

这样的因果性 不仅在 encoder中用到 在decoder部分也会用到 只要是将 Transformer用于流式的生成 都会用到这种 因果的mask

这个讲解是每部分的功能实现 不是完整的工程演示，够了 啥不是局部到整体 积水成渊

上面 只是构建了张量的列表，最终是要构建张量出来

怎么构建张量呢？同样的 首先构建 有效的 decoder pos出来，首先pad成长度相同的，decoder的第一句是4个有效字符，第二句是3个有效字符，所以首先进行pad操作 F.pad,pad(左右上下)=(0,0,0,1)我们在下面pad一行，也不一定是一行 改成 max(tgt_len)-L，pad成最大长度

```
# Step6: 构造decoder self-attention的mask
valid_decoder_tri_matrix = [F.pad(torch.tril(torch.ones((L, L))), (0, 0, 0, max(tgt_len)-L)) for L in tgt_len]
print(valid_decoder_tri_matrix)
```

```
[tensor([[1., 0., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 1., 0.],
         [1., 1., 1., 1.]]), tensor([[1., 0., 0.],
         [1., 1., 0.],
         [1., 1., 1.],
         [0., 0., 0.]])]
```

可以看到 pad成功了 第二个张量 也变成了四行，首先保证张量的维度是一样的，当然我们不仅要pad行，列也要进行pad

```
# Step6: 构造decoder self-attention的mask
valid_decoder_tri_matrix = [F.pad(torch.tril(torch.ones((L, L))), (0, max(tgt_len)-L, 0, max(tgt_len)-L)) for L in tgt_
print(valid_decoder_tri_matrix)
```

```
[tensor([[1., 0., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 1., 0.],
         [1., 1., 1., 1.]]), tensor([[1., 0., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 1., 0.],
         [0., 0., 0., 0.]])]
```

这样就完全一样了，接下来把两个张量cat起来就好了

```
# Step6: 构造decoder self-attention的mask
valid_decoder_tri_matrix = torch.cat([F.pad(torch.tril(torch.ones((L, L))), (0, max(tgt_len)-L, 0, max(tgt_len)-L)) \
                                      for L in tgt_len])
print(valid_decoder_tri_matrix)
```

```
tensor([[1., 0., 0., 0.],
        [1., 1., 0., 0.],
        [1., 1., 1., 0.],
        [1., 1., 1., 1.],
        [1., 0., 0., 0.],
        [1., 1., 0., 0.],
        [1., 1., 1., 0.],
        [0., 0., 0., 0.]])
```

还要先进行扩维

```
# Step6: 构造decoder self-attention的mask
valid_decoder_tri_matrix = torch.cat([torch.unsqueeze(F.pad(torch.tril(torch.ones((L, L))), \
                                                            (0, max(tgt_len)-L, 0, max(tgt_len)-L)),0) \
                                      for L in tgt_len])
print(valid_decoder_tri_matrix)
```

```
tensor([[[1., 0., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 1., 0.],
         [1., 1., 1., 1.]],

        [[1., 0., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 1., 0.],
         [0., 0., 0., 0.]]])
```

这样就把两个张量pad起来了

结果解读：第一个序列全是下三角的，因为每个单词都是有效的，最后一个序列不是全下三角的，最后一行全为0，因为第二个序列最后一个单词是无效的，是我们pad的一个单词，如上我们确定了所有有效位置，打印形状

```
# Step6: 构造decoder self-attention的mask
valid_decoder_tri_matrix = torch.cat([torch.unsqueeze(F.pad(torch.tril(torch.ones((L, L))), \
                                                            (0, max(tgt_len)-L, 0, max(tgt_len)-L)),0) \
                                      for L in tgt_len])
print(valid_decoder_tri_matrix.shape)
#valid_decoder_tri_matrix = torch.bmm(valid_encoder_pos, valid_encoder_pos.transpose(1, 2))
```

```
torch.Size([2, 4, 4])
```

形状 已经是244的，这个就是decoder self mask三角的形式，再来看一下值，再来解释

```
# Step6: 构造decoder self-attention的mask
valid_decoder_tri_matrix = torch.cat([torch.unsqueeze(F.pad(torch.tril(torch.ones((L, L))), \
                                                            (0, max(tgt_len)-L, 0, max(tgt_len)-L)),0) \
                                      for L in tgt_len])
print(valid_decoder_tri_matrix.shape)
print(valid_decoder_tri_matrix)
```

```
torch.Size([2, 4, 4])
tensor([[[1., 0., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 1., 0.],
         [1., 1., 1., 1.]],

        [[1., 0., 0., 0.],
         [1., 1., 0., 0.],
         [1., 1., 1., 0.],
         [0., 0., 0., 0.]]])
```

张量分为两个部分，第一个部分表示decoder的第一个样本，第二个部分表示第二个样本，第一个样本第一行表示decoder的第一个输入对decoder其他所有序列的位置的一个相关性，因为第一步只要求输入第一个位置上的，所以第一行只有第一个元素为1，其他元素都为0；同样第二行就表示，第二个位置上的单词跟其他单词的相关性，总之为了满足自回归，我们需要这样一个因果的有效矩阵，得到有效矩阵以后就可以得到掩码矩阵，1-有效矩阵 再变成布尔类型，再打印：

```
# Step6: 构造decoder self-attention的mask
valid_decoder_tri_matrix = torch.cat([torch.unsqueeze(F.pad(torch.tril(torch.ones((L, L))), \
                                                            (0, max(tgt_len)-L, 0, max(tgt_len)-L)),0) \
                                      for L in tgt_len])
invalid_decoder_tri_matrix = 1-valid_decoder_tri_matrix
invalid_decoder_tri_matrix = invalid_decoder_tri_matrix.to(torch.bool)
print(invalid_decoder_tri_matrix)
```

```
tensor([[[False,  True,  True,  True],
         [False, False,  True,  True],
         [False, False, False,  True],
         [False, False, False, False]],

        [[False,  True,  True,  True],
         [False, False,  True,  True],
         [False, False, False,  True],
         [ True,  True,  True,  True]]])
```

这样我们得到这个decoder，自注意力的掩码矩阵，我们看到是一个三角形的，true表示我们要掩码的score；我们随机生成一个score，这里的score是T×T（target len），然后我们mask掉，就是调用score.masked_fill()这个函数经常会用到，第一个参数传入要mask的对象 invalid_decoder_tri_matrix，元素位置为true的地方，mask成 非常非常小的数，得到的东西 定义成masked_score:

```
# 构造decoder self-attention 的mask
valid_decoder_tri_matrix =
torch.cat([torch.unsqueeze(F.pad(torch.tril(torch.ones((L,L))),
                                 (0,max(tgt_len)-L,0,max(tgt_len)-L)),0)
                                 for L in tgt_len])
invalid_decoder_tri_matrix = 1-valid_decoder_tri_matrix
invalid_decoder_tri_matrix = invalid_decoder_tri_matrix.to(torch.bool)
print(invalid_decoder_tri_matrix)

score = torch.randn(batch_size,max(tgt_len),max(tgt_len))
masked_score = score.masked_fill(invalid_decoder_tri_matrix,-1e09)
```

接下来对masked score进行一个softmax函数，传给prob，接下来打印查看结果：

```
score = torch.randn(batch_size, max(tgt_len), max(tgt_len))
masked_score = score.masked_fill(invalid_decoder_tri_matrix, -1e9)
prob = F.softmax(masked_score, -1)
print(tgt_len)
print(prob)
```

```
tensor([4, 3], dtype=torch.int32)
tensor([[[1.0000, 0.0000, 0.0000, 0.0000],
         [0.7358, 0.2642, 0.0000, 0.0000],
         [0.4788, 0.2536, 0.2675, 0.0000],
         [0.3182, 0.1064, 0.4926, 0.0828]],

        [[1.0000, 0.0000, 0.0000, 0.0000],
         [0.0944, 0.9056, 0.0000, 0.0000],
         [0.5990, 0.0344, 0.3666, 0.0000],
         [0.2500, 0.2500, 0.2500, 0.2500]]])
```

如图是一个注意力的权重矩阵，上面是第一个样本，下面是第二个样本；

第一个样本，第一次解码的时候，只注意到第一个样本，第二次解码的时候注意到前两个单词，第三次解码的时候，注意到前三个单词，第四次解码的时候，注意到前四个单词，由于第一个句子的target sequence=4，所以每次单词都是有效的；

第二个序列，同样，第一次解码的时候只注意到第一个单词，后面的单词遮住，因为后面已经是答案了，我们看最后一行 都是0.25 ，因为最后一行 第二个句子的最后一个单词 本来就是pad的单词，所以score都是被mask掉的，所以概率都是平均的。这就是decoder self attention mask。这是很重要的 很多地方都会用到这个因果的掩码。这部分所有代码如下：

```
# 构造decoder self-attention 的mask
valid_decoder_tri_matrix =
torch.cat([torch.unsqueeze(F.pad(torch.tril(torch.ones((L,L))),

(0,max(tgt_len)-L,0,max(tgt_len)-L)),0)
                                                        for L in
tgt_len])
invalid_decoder_tri_matrix = 1-valid_decoder_tri_matrix
invalid_decoder_tri_matrix = invalid_decoder_tri_matrix.to(torch.bool)
# print(invalid_decoder_tri_matrix)

score = torch.randn(batch_size,max(tgt_len),max(tgt_len))
masked_score = score.masked_fill(invalid_decoder_tri_matrix,-1e09)
prob = F.softmax(masked_score,-1)
print(tgt_len)
print(prob)
```

接下来 下一步，构建self attention，对照公式：

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

写一个函数 scaled_dot_product_attention,传进来的输入参数 Q、K、V还有mask 命名为 attn_mask

```
def scaled_dot_product_attention(Q,K,V,attn_mask):
```

第一步计算Q乘K，同样的调用bmm,因为Q和V现在都是batch的，同时也是batch乘以multi head的，传入Q和K，同时K要转置，transpose(-1,-2)因为它的维度 或者四维 或者三维

```
score = torch.bmm(Q,K.transpose(-2,-1))/torch.sqrt(model_dim)
```

接下来 对score进行mask `masked_score = score.masked_fill(attn_mask,-1e9)` 这里的mask可能是encoder的 可能是decoder的 也可能是encoder decoder intra的部分;利用 attn_mask对score掩码的部分赋值 一个很小的数

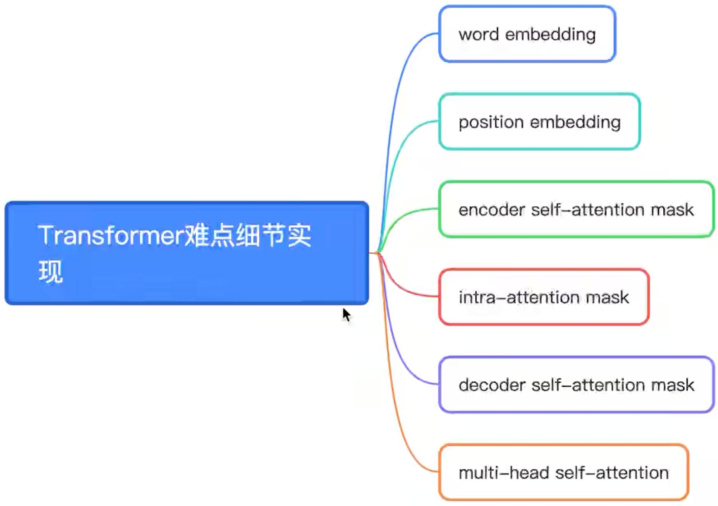
接下来 -1维度 进行softmax，得到一个prob `prob = F.softmax(masked_score,-1)`

再用 torch.bmm函数 把 prob和V，进行一个加权求和 实际上就是一个矩阵相乘，得到 context `context = torch.bmm(prob,V)`最后返回 context`return context`

```
# 构建scaled self-attention
def scaled_dot_product_attention(Q,K,V,attn_mask):
    # shape of Q,K,V:(batch_size*num_head,seq_len,model_dim/num_head)
    score = torch.bmm(Q,K.transpose(-2,-1))/torch.sqrt(model_dim)
    masked_score = score.masked_fill(attn_mask,-1e9)
    prob  = F.softmax(masked_score,-1)
    context = torch.bmm(prob,V)
    return context
```

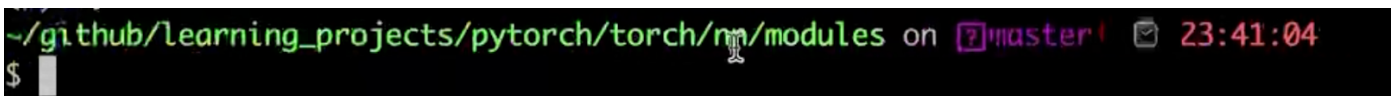
需要注意的是 这里Q、K、V的形状：`batch_size*num_head,seq_len,model_dim/num_head`

当然 这里 seq_len 可能长度不一样 因为这里的Q不一定是和K和V是一样的，涉及到itra attention的时候 它们的长度就不一样；以上是scaled_dot_product_attention，以上 这些部分基本讲完了：



接下来 我们再来看下Transformer源码

首先我们来看Transformer整个的class，在pytorch中，Transformer的源码位于pytorch下面的torch下面的nn.modules的Transformer.py里面的



写了一个Transformer的class

```
class Transformer(Module):
    r"""A transformer model. User is able to modify the attributes as needed. The architecture
    is based on the paper "Attention Is All You Need". Ashish Vaswani, Noam Shazeer,
    Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and
    Illia Polosukhin. 2017. Attention is all you need. In Advances in Neural Information
    Processing Systems, pages 6000-6010. Users can build the BERT(https://arxiv.org/abs/1810.04805)
    model with corresponding parameters.
```

首先 我们看它的init函数，如果我们要调用这个class的话

```
def __init__(self, d_model: int = 512, nhead: int = 8, num_encoder_layers: int = 6,
             num_decoder_layers: int = 6, dim_feedforward: int = 2048, dropout: float = 0.1,
             activation: Union[str, Callable[[Tensor], Tensor]] = F.relu,
             custom_encoder: Optional[Any] = None, custom_decoder: Optional[Any] = None,
             layer_norm_eps: float = 1e-5, batch_first: bool = False, norm_first: bool = False,
             device=None, dtype=None) -> None:
    factory_kwargs = {'device': device, 'dtype': dtype}
    super(Transformer, self).__init__()
```

我们需要传入一些d_model，nhead，encoder的层数，decoder的层数以及全连接的维度，主要是这些，这是我们实例化好了Transformer这个实例，然后调用，去看它的forward函数：

```
def forward(self, src: Tensor, tgt: Tensor, src_mask: Optional[Tensor] = None, tgt_mask: Optional[Tensor] = None,
            memory_mask: Optional[Tensor] = None, src_key_padding_mask: Optional[Tensor] = None,
            tgt_key_padding_mask: Optional[Tensor] = None, memory_key_padding_mask: Optional[Tensor] = None) -> Tensor:
    """Take in and process masked source/target sequences.

    Args:
        src: the sequence to the encoder (required).
        tgt: the sequence to the decoder (required).
        src_mask: the additive mask for the src sequence (optional).
        tgt_mask: the additive mask for the tgt sequence (optional).
        memory_mask: the additive mask for the encoder output (optional).
        src_key_padding_mask: the ByteTensor mask for src keys per batch (optional).
        tgt_key_padding_mask: the ByteTensor mask for tgt keys per batch (optional).
        memory_key_padding_mask: the ByteTensor mask for memory keys per batch (optional).
```

在forward函数，需要给什么呢？需要给source 也就是源序列的词向量 src，目标序列的词向量 tgt,然后就是 源序列的mask `src_mask`，目标序列的mask `tgt_mask` `src_mask` 就是我们之前讲的encoder mask，tgt_mask就是decoder mask，memory_mask就是之前讲的 intra mask，其他的都可以不用写，然后就可以得到解码器的输出，不过这个输出是还有没有经过softmax的，还需要加一个全连接 再加一个概率的预测；这个是 class部分，然后实际的实现在activation.py，在activation.py里面，写了一个MultiheadAttention class

```
class MultiheadAttention(Module):
    r"""Allows the model to jointly attend to information
    from different representation subspaces.
    See `Attention Is All You Need <https://arxiv.org/abs/1706.03762>`_.

    .. math::
        \text{MultiHead}(Q, K, V) = \text{Concat}(head_1,\dots,head_h)W^O

    where :math:`head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)`.

    Args:
        embed_dim: Total dimension of the model.
        num_heads: Number of parallel attention heads. Note that ``embed_dim`` will be split
            across ``num_heads`` (i.e. each head will have dimension ``embed_dim // num_heads``).
        dropout: Dropout probability on ``attn_output_weights``. Default: ``0.0`` (no dropout).
        bias: If specified, adds bias to input / output projection layers. Default: ``True``.
        add_bias_kv: If specified, adds bias to the key and value sequences at dim=0. Default: ``False``.
        add_zero_attn: If specified, adds a new batch of zeros to the key and value sequences at dim=1.
```

这个class 也没干什么事，它最终调用的是 另外一个函数

```
if not self._qkv_same_embed_dim:
    attn_output, attn_output_weights = F.multi_head_attention_forward(
        query, key, value, self.embed_dim, self.num_heads,
        self.in_proj_weight, self.in_proj_bias,
        self.bias_k, self.bias_v, self.add_zero_attn,
        self.dropout, self.out_proj.weight, self.out_proj.bias,
        training=self.training,
```

调用的是 multi_head_attention_forward，这个函数是在上一层functional.py里面的

```
def multi_head_attention_forward(
    query: Tensor,
    key: Tensor,
    value: Tensor,
    embed_dim_to_check: int,
    num_heads: int,
    in_proj_weight: Tensor,
    in_proj_bias: Optional[Tensor],
    bias_k: Optional[Tensor],
    bias_v: Optional[Tensor],
    add_zero_attn: bool,
    dropout_p: float,
    out_proj_weight: Tensor,
    out_proj_bias: Optional[Tensor],
    training: bool = True,
    key_padding_mask: Optional[Tensor] = None,
    need_weights: bool = True,
    attn_mask: Optional[Tensor] = None,
ter   ../functional.py\
```

就是在functional.py里面 有一个 multi_head_attention_forward函数，这个函数干的事情就是之前讲的 scaled_dot_product_attention以及怎么去算mask，接下来梳理一下这个函数：

```
def multi_head_attention_forward(
    query: Tensor,
    key: Tensor,
    value: Tensor,
    embed_dim_to_check: int,
    num_heads: int,
    in_proj_weight: Tensor,
    in_proj_bias: Optional[Tensor],
    bias_k: Optional[Tensor],
    bias_v: Optional[Tensor],
    add_zero_attn: bool,
    dropout_p: float,
    out_proj_weight: Tensor,
    out_proj_bias: Optional[Tensor],
    training: bool = True,
    key_padding_mask: Optional[Tensor] = None,
    need_weights: bool = True,
    attn_mask: Optional[Tensor] = None,
    use_separate_proj_weight: bool = False,
    q_proj_weight: Optional[Tensor] = None,
    k_proj_weight: Optional[Tensor] = None,
    v_proj_weight: Optional[Tensor] = None,
    static_k: Optional[Tensor] = None,
    static_v: Optional[Tensor] = None,
) -> Tuple[Tensor, Optional[Tensor]]:
    r"""
```

这个函数接收的输入，包括qkv,num heads,project weight,bias,还有attn mask

QKV是 embedding的维度

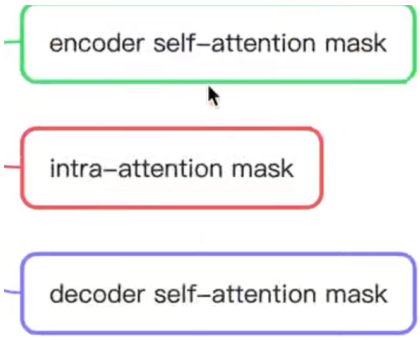
首先算出每个头的维度

```
if isinstance(embed_dim, torch.Tensor):
    # embed_dim can be a tensor when JIT tracing
    head_dim = embed_dim.div(num_heads, rounding_mode='trunc')
else:
    head_dim = embed_dim // num_heads
```

head dim等于 embedding dim除以 num head，首先把这个维度算出来，然后呢，我们需要 对QKV做一个映射，原本输入的QKV就是 word embedding要么是 source embedding 或者target embedding当然 也可能是 中间层的；第一步要先对这三个量 进行一个 projection 操作，也就是分别进入三个全连接网络，得到新的QKV

```
if not use_separate_proj_weight:
    q, k, v = _in_projection_packed(query, key, value, in_proj_weight, in_proj_bias)
```

接下来 算Attention mask，Attention mask函数里写得很复杂 其实原理前面 都已经讲过了，也就是要搞懂三个mask，分别是什么原理就可以了



第一个是encoder mask、然后是intra Attention mask；和decoder mask；把这三个原理搞懂了 就不用管它怎么实现的，自己就可以简单的实现了；得到mask以后，，就会把mask给它填充进去，也就是最终调用的是scaled_dot_product_attention

```
    # (deep breath) calculate attention and out projection
    #
    attn_output, attn_output_weights = scaled_dot_product_attention(q, k, v, attn_mask, dropout_p)
    attn_output = attn_output.transpose(0, 1).contiguous().view(tgt_len, bsz, embed_dim)
    attn_output = linear(attn_output, out_proj_weight, out_proj_bias)
```

在这个函数里面：

```
def _scaled_dot_product_attention(
    q: Tensor,
    k: Tensor,
    v: Tensor,
    attn_mask: Optional[Tensor] = None,
    dropout_p: float = 0.0,
) -> Tuple[Tensor, Tensor]:
    r"""
    Computes scaled dot product attention on query, key and value tensors, using
    an optional attention mask if passed, and applying dropout if a probability
    greater than 0.0 is specified.
    Returns a tensor pair containing attended values and attention weights.

    Args:
        q, k, v: query, key and value tensors. See Shape section for shape details.
        attn_mask: optional tensor containing mask values to be added to calculated
            attention. May be 2D or 3D; see Shape section for details.
        dropout_p: dropout probability. If greater than 0.0, dropout is applied.
```

```
B, Nt, E = q.shape
q = q / math.sqrt(E)
# (B, Nt, E) x (B, E, Ns) -> (B, Nt, Ns)
attn = torch.bmm(q, k.transpose(-2, -1))
if attn_mask is not None:
    attn += attn_mask
attn = softmax(attn, dim=-1)
if dropout_p > 0.0:
    attn = dropout(attn, p=dropout_p)
# (B, Nt, Ns) x (B, Ns, E) -> (B, Nt, E)
output = torch.bmm(attn, v)
return output, attn
```

这个函数 和之前写的 是类似的，首先 对 q进行一个 scale，除以这个维度 的开方`q = q/math.sqrt(E)`

然后把 q乘以 k的转置 得到 attn`attn=torch.bmm(q,k.transpose(-2,-1))`

进而 我们把 attn 给mask一下`attn+=attn_mask`

这里用的是 加法；加法的mask就是说 有效的地方就是0；无效的地方是一个 负无穷；如果是用的乘法的话，有效的地方是1，无效的地方也是负无穷；如果使用的masked_fill这个函数的话，同样的 有效的地方是1，无效的地方是 负无穷；这里要区分一下 如果是加法就会不一样；

然后再经过softmax，得到概率`attn = softmax(attn,dim=-1)`

这里还加了一个 dropout层 `attn  = dropout(attn,p=dropout_p)`

最后把这个概率 跟V进行一个加权求和 `output=torch.bmm(attn,v)` 也就是bmm操作 得到 output

以上是整个 pytorch 关于 Transformer的一个实现；以上难点细节也全部讲完了

还有loss的mask、全连接层就比较简单了

所有的代码：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy

# step1 --- 定义常量 ---

batch_size = 2

# 单词表大小，单词表中有多少个单词
```

```
max_num_src_words = 8
max_num_tgt_words = 8


# 词嵌入维度
model_dim = 8

# 序列的最大长度
max_src_seq_len = 5
max_tgt_seq_len = 5
max_position_len = 5

# batch size = 2,源 第一个句子长度 = 2，第二个句子长度 = 4
src_len = torch.Tensor([2,4]).to(torch.int32)
tgt_len = torch.Tensor([4,3]).to(torch.int32)

# 单词索引构成源句子和目标句子，构建batch，并且做padding，默认值为0
src_seq =
torch.cat([torch.unsqueeze(F.pad(torch.randint(1,max_num_src_words,
(L,)),
                                                 (0,max(src_len)-L)),0)
                                                 for L in src_len])

tgt_seq =
torch.cat([torch.unsqueeze(F.pad(torch.randint(1,max_num_tgt_words,
(L,)),
                                                 (0,max(tgt_len)-L)),0)
                                                 for L in tgt_len])

# step2  构造word embedding
src_embedding_table = nn.Embedding(max_num_src_words+1,model_dim)
tgt_embedding_table = nn.Embedding(max_num_tgt_words+1,model_dim)

src_embedding = src_embedding_table(src_seq)
tgt_embedding = tgt_embedding_table(tgt_seq)

# step3 构造position embedding
pos_mat = torch.arange(max_position_len).reshape((-1,1))
i_mat = torch.pow(10000,torch.arange(0,8,2).reshape((1,-1))/model_dim)

pe_embedding_table = torch.zeros(max_position_len,model_dim)
pe_embedding_table[:,0::2] = torch.sin(pos_mat / i_mat)
pe_embedding_table[:,1::2] = torch.cos(pos_mat / i_mat)
```

```
# print(pe_embedding_table)

pe_embedding = nn.Embedding(max_position_len,model_dim)
pe_embedding.weight =
nn.Parameter(pe_embedding_table,requires_grad=False)

src_pos = torch.cat([torch.unsqueeze(torch.arange(max(src_len)),0) for _
in src_len]).to(torch.int32)
tgt_pos = torch.cat([torch.unsqueeze(torch.arange(max(tgt_len)),0) for _
in tgt_len]).to(torch.int32)

# print(src_pos)

src_pe_embedding = pe_embedding(src_pos)
tgt_pe_embedding = pe_embedding(tgt_pos)
# print(src_pe_embedding)
# print(tgt_pe_embedding)

# step4  构造 encoder的self attention mask
# mask的shape: [batch_size,max_src_len,max_src_len], 值为1或-inf

valid_encoder_pos =
torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L),
(0,max(src_len)-L)),0)
                                                for L in src_len]),2)

valid_encoder_pos_matrix =
torch.bmm(valid_encoder_pos,valid_encoder_pos.transpose(1,2))

invalid_encoder_pos_matrix = 1-valid_encoder_pos_matrix

mask_encoder_self_attention = invalid_encoder_pos_matrix.to(torch.bool)

score = torch.randn(batch_size,max(src_len),max(src_len))
# print(score.shape,mask_encoder_self_attention.shape)

masked_score = score.masked_fill(mask_encoder_self_attention,-1e9)
prob = F.softmax(masked_score,-1)

# step5: 构造intra attention 的mask
# Q @ K^T shape:[batch_size ,tgt_seq_len,src_seq_len]
```

```
valid_encoder_pos =
torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L),
(0,max(src_len)-L)),0)
                                                  for L in src_len]),2)

valid_decoder_pos =
torch.unsqueeze(torch.cat([torch.unsqueeze(F.pad(torch.ones(L),
(0,max(tgt_len)-L)),0)
                                                  for L in tgt_len]),2)

valid_cross_pos_matrix =
torch.bmm(valid_decoder_pos,valid_encoder_pos.transpose(1,2))
invalid_cross_pos_matrix = 1- valid_cross_pos_matrix
mask_cross_self_attention = invalid_cross_pos_matrix.to(torch.bool)
# print(mask_cross_self_attention)

# step6 构造decoder self-attention 的mask
valid_decoder_tri_matrix =
torch.cat([torch.unsqueeze(F.pad(torch.tril(torch.ones((L,L))),

(0,max(tgt_len)-L,0,max(tgt_len)-L)),0)
                                                              for L in
tgt_len])
invalid_decoder_tri_matrix = 1-valid_decoder_tri_matrix
invalid_decoder_tri_matrix = invalid_decoder_tri_matrix.to(torch.bool)
# print(invalid_decoder_tri_matrix)

score = torch.randn(batch_size,max(tgt_len),max(tgt_len))
masked_score = score.masked_fill(invalid_decoder_tri_matrix,-1e09)
prob = F.softmax(masked_score,-1)
# print(tgt_len)
# print(prob)


# step7 构建scaled self-attention
def scaled_dot_product_attention(Q,K,V,attn_mask):
    # shape of Q,K,V:(batch_size*num_head,seq_len,model_dim/num_head)
    score = torch.bmm(Q,K.transpose(-2,-1))/torch.sqrt(model_dim)
    masked_score = score.masked_fill(attn_mask,-1e9)
    prob  = F.softmax(masked_score,-1)
    context = torch.bmm(prob,V)
    return context
```