- 241019 开始【19、Transformer模型Encoder原理精讲及其PyTorch逐行实现-哔哩哔哩】 https://b23.tv/n4NHN6f
- 241020 torch.concat 、torch.unsqueeze
- 241021 mask

Transformer模型Encoder原理精讲及其PyTorch逐行实现

难点 细节：

- word embedding
- position embeddig
- encoder self-attention mask
- intra-attention mask
- Decoder self-attention mask
- Multi-head attention

首先回顾Transformer的结构，encoder+decoder共同实现序列建模的任务

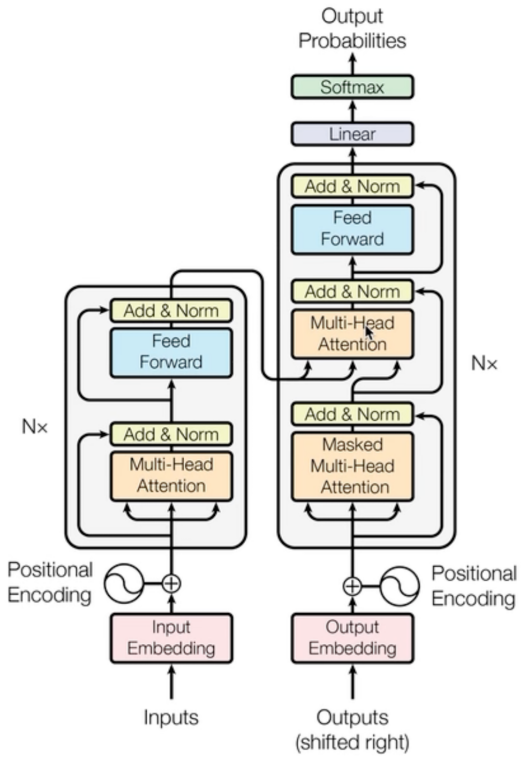


Figure 1: The Transformer - model architecture.

（口述）Transformer的结构

首先接收两个输入，inputs和outputs，分别进行inputs embedding和outputs embedding，由于是nlp任务，句子中token的顺序很重要，所以我们还需要位置编码 position encoding，再对inputs和outputs进行了位置编码以后，直接加到词嵌入向量，得到了带有位置信息的特征表示，接下来编码端，经过MAH、layerNorm&Add，再经过FFN，再接着layerNorm&Add得到编码端的输出，别忘了LayerNorm和Add这个模块涉及到残差连接，刚刚叙述的是一个encoder 的数据流动，在实际应用中，我们会堆叠n层，接下来解码端部分，再得到了带有位置信息的词嵌入表示以后，经过MMHA，带有mask的多头注意力机制，然后经过层归一化和残差连接，经过交叉注意力机制，query来自解码端上一层的输出，key和value都是来自编码端的输出，做好交叉多头注意力机制以后，经过层归一化和残差连接，经过前馈全连接层FFN，堆叠n层，得到解码器的输出，解码器的输出经过一个linear层 再进行softmax 得到最终的预测概率。

（口述）Transformer用到的三个mask机制，以及为什么?

三个mask机制，分别指的是编码端输入 由于padding字符的mask，为了一个batchsize中，所有长度不相同的样本，能构成一个矩阵，所以有pad字符，但是在后面进行input encoder的自注意力计算时，pad字符不能影响计算结果，所以需要mask;

第二个mask是解码端的mask，这个mask是涉及到因果的mask，因为Transformer是一个自回归模型，在进行运算时，为了并行计算，我们是把inputs和outputs一起喂给模型的，inputs直接给模型没事，但是outputs在得到最后的输出时，不能借助未来信息，只能是当前时刻及其之前时刻的输出，所以需要一个mask机制，这个mask是一个上三角矩阵，保证在预测当前输出时，不会借助未来信息。

第三个mask，是编码器和解码器的交互注意力，编码器的输出作为key和value，解码器的输出作为query（未完）

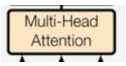
以机器翻译为例，encoder中 以源语言source language，比方说中文，我们把中文字符输入到encoder中，主要基于multi-head self attention以及FFN这两个模块，堆积很多层，最终得到一个上下文相关的一个单词编码，作为memory，decoder部分 我们把目标序列，比如说英文单词 作为output embedding输入进去，由于模型是一个自回归模型，所以output embedding需要有一个因果的mask，使得每个位置的单词，只以这个位置之前的单词作为输入

然后首先，output embedding也会首先经过 multi-head self attention 得到目标单词的表征，再结合memory，再经过multi-head attention 也叫intra-attention得到一个output embedding作为query，得到memory的表征，经过重复堆叠，得到decoder的输出，接着经过Linear和softmax映射到概率空间，完成序列建模

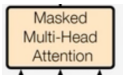
接下来会讲解：

- word embedding
- position embeddig
- encoder self-attention mask
- intra-attention mask
- Decoder self-attention mask
- Multi-head attention

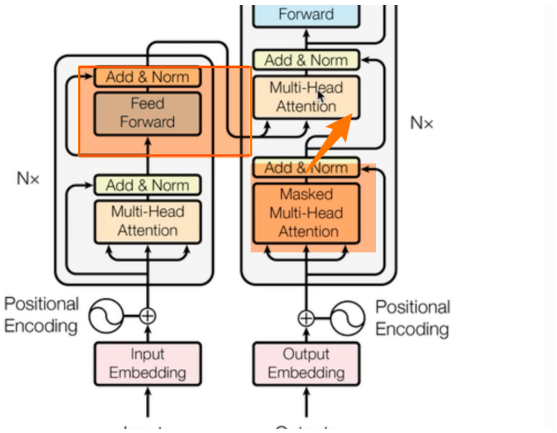
首先 word embedding，在实际的任务中如何实现word embedding；第二点由于Transformer是一个没有前提假设的一个结构，既没有局部性假设，也没有序列假设，所以需要一个位置信息，来表示nlp序列，也就是position encoding，也可以叫position embedding，第三个部分encoder self-attention mask，mask的目的是让这个模型更加高效的训练好，因为训练一般是小批次训练，也就是minibatch的训练模式，所谓mini batch的训练模式，也就是一次训练 会用多个样本序列，而不是一个样本一个样本的训练，由于序列长度不一样，于是需要mask来保证我们得到的表征，都是有效的表征，就是不希望我们得到的表征受PAD字符的影响



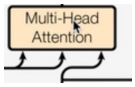
整个模型，有大的 三种mask，第一种是encoder这里的mask，encoder这里的multi head self Attention是序列自身对自身的关联性计算，而且encoder这里接收的输入不涉及因果，整个输入是一次性输入进去，encoder这部分的mask是比较简单的



第二个mask是decoder这里的mask，也就是 masked multi head Attention，这个mask也是序列本身对序列本身的一个关联性的表征计算，但是这里由于模型是一个自回归模型，所以在训练的时候，需要保证output embedding输入到模型中没有未来信息，所以这部分mask就是一个需要保证因果性的mask，所以这部分会稍微复杂一点



第三个mask就是encoder memory和最下面的MHA输出的，它们两个之间形成的一个Attention，可以看到最下面的分支可以作为query，encoder的输出可以作为key和memory，然后计算MHA，这部分的注意力机制的mask和encoder部分的mask是类似的，但是呢，这里的mask涉及到两个不同的序列，所以这里的mask，



它的query和key，是两个不同的序列，或者说序列长度可能都不一样，所以这是另外一种mask

最后就是这个 multi head self attention的实现，这个实现是比较简单的multi head 可以和batch是一样的意思，就可以当成是batch这个维度，head与head之间是独立的，类似样本与样本之间也是独立的，增加multi head就是把这个维度汇总到batch这个维度进行计算

接下来介绍，为什么要讲这6点，我们上图给出的是最初的Transformer的模型，现在已经很少用到这个模型了，有encoder、decoder这样的自回归的模型，虽然已经很少的完整用到这个模型，但是这里面的mask是会经常用到的，不论是encoder部分 自身对自身的mask，还是decoder部分因果的mask ，自身到自身 肯定是经常会用到的，因果mask也是会经常遇到，为什么呢？因为在实际的生产中，经常会遇到流式的去计算，这个Transformer的输出，不是等它一次性的输出完再返回给用户，而是流式的，算出一点就立刻返回给用户，这样的流式运算，必然用到因果的mask；

然后，不同序列的交叉attention，也是很常用的；position embedding也是，只要用Transformer必然用到 位置编码，只是这个位置编码 是 常数 的 还是可学习的 再看。而word embedding只要是做离散的字符输入 就会需要，很常用的

再讲完 实现以后 会讲怎么用pytorch的api来调用实现Transformer

绪论结束，开始写代码。

推荐的几个Transformer的项目，vit，swinTransformer，nlp领域的预训练，github Transformer搜索 会出现很多项目

首先，导入常用库：

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

第一步先讲，关于word embedding，首先以序列建模为例，肯定有source和target，考虑source sentence和target sentence，这些都是离散建模，也是可以用连续建模的

首先构建序列，序列的字符以索引的形式表示，首先构建source序列和target序列，再构建具体的source序列和target序列之前，先定义source length和target length，原序列的长度和目标序列的长度先假设一个值，然后跟长度，来随机的生成，单词的索引

src_len = torch.randint()

torch.randint函数 就是 随机生成一个整型的函数，randint的api up主之前的视频有讲过，

第一个位置要带入一个**最小值**，现在假设原序列最小值为2，

第二个位置为最大值，设原序列**最大值**为5，

接下来第三个位置是**形状**，原序列的形状就是batch size，所以要在最开始定义一个batch size，举例还是简单的 设 batch_size = 2

```
batch_zie = 2
src_len = torch.randint(2,5,(batch_size,))
tgt_len =
src_seq =
tgt_seq =
```

经过以上代码 就能生成 原序列的长度，同样的目标序列的长度，设成和src_len一样的，原序列有多少个，目标序列就会有多少个，保持一致。

接下来完成第一部分：

In [1]:
```
import torch
import numpy
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
src_len = torch.randint(2, 5, (batch_size,))
tgt_len = torch.randint(2, 5, (batch_size,))
print(src_len)
print(tgt_len)
```

```
tensor([3, 4])
tensor([2, 2])
```

打印输出，src_len = [3,4]、tgt_len = [2,2]，这是什么意思呢？就是说假设batch_size=2,然后原序列 source有两个句子，第一个句子长度为3，第二个句子长度为4；然后目标句子，第一个句子的长度为2，第二个句子的长度也为2，如果再运行一次，就会有变化，这样不好，所以为了说明，进行 hard code，强行指定系列长度，代码如下：

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

batch_size = 2

src_len = torch.Tensor([2,4]).to(torch.int32)
tgt_len = torch.Tensor([4,3]).to(torch.int32)

print(src_len)
print(tgt_len)
```

解读：原序列长度分别是2,4，目标序列长度分别是4,3

接着在有了序列长度以后，生成源序列，在长度给定的情况下使用torch.randint来生成具体的单词，单词也就是索引，索引的最小值 定义为1，最大值，假设单词表数目 是 8，然后设置这里的size，也就是长度L，L也就是从source len里取到的，这样我们生成了第一个句子和第二个句子,并将其放入列表中，接下来，我们打印，看长什么样子

```
src_seq = [torch.randint(1,8,(L,)) for L in src_len]
print(src_seq)
```

Out:[tensor([7,3]),tensor([6,7,1,3])]

这就是我们生成的两个源序列，同样的方法生成目标序列，目标序列的最大词表 也假设为8

```
tgt_seq = [torch.randint(1,8,(L,)) for L in tgt_len]
print(tgt_seq)
```

使用全局变量 max_num_src_words = 8 也就是源序列单词的总数，目标序列 max_num_tgt_words = 8 同理

于是，

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

src_seq = [torch.randint(1,max_num_src_words,(L,)) for L in src_len]
tgt_seq = [torch.randint(1,max_num_tgt_words,(L,)) for L in tgt_len]

print(src_seq,tgt_seq)
```

完整代码：

```
import torch
import numpy
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

#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)

src_seq = [torch.randint(1, max_num_src_words, (L,)) for L in src_len]
tgt_seq = [torch.randint(1, max_num_tgt_words, (L,)) for L in tgt_len]


print(src_seq, tgt_seq)
```

```
[tensor([7, 3]), tensor([6, 4, 5, 5])] [tensor([4, 3, 5, 1]), tensor([5, 1, 6])]
```

src_seq、tgt_seq就是 单词索引构成的句子，观察可知 不管是 src_seq还是tgt_seq 中的两个句子，长度都不一样，如果我们要以矩阵的形式输入到网络中，我们要保证句子的长度是一样的，也就是加一个Padding

具体怎么加呢，对每一个张量进行pad操作，pad是在functional的api中，使用F.pad函数，每个序列pad成最大长度，比如说source sequence最大长度是4，target sequence最大长度也是4，所以我们要把它pad成最大长度，因为是一个一维张量，比如说tensor([7,3])，F.pad第一个位置接收要pad的对象，第二个位置 接收 要pad的形状 接收的是元组 第一个数 是pad左边多少位，左边我们统一都不pad，右边pad的最大值，也就是**原序列的最大长度减去当前长度**（注意理解这句话），举个例子，比方说 当前句子长度为2：tensor([7,3])，而最大长度为4，所以 我们pad 两个0，第2个句子 tensor([6,4,5,5]),达到最大长度，所以不需要pad。

```python
import torch
import numpy
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

#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)

# 单词索引构成的句子
src_seq = [F.pad(torch.randint(1, max_num_src_words, (L,)), (0, max(src_len)-L)) for L in src_len]
tgt_seq = [torch.randint(1, max_num_tgt_words, (L,)) for L in tgt_len]


print(src_seq, tgt_seq)
```

```
[tensor([7, 3]), tensor([6, 4, 5, 5])] [tensor([4, 3, 5, 1]), tensor([5, 1, 6])]
```

接下来 我们定义全局变量 我们定义一个量 表示最大长度 比如 5，即max_src_seq_len =5，target也是一样的 max_tgt_seq_len = 5,也就是我们定义的最大序列长度，默认填充0，运行查看结果

```
In [6]:
```

```python
import torch
import numpy
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

# 序列的最大长度
max_src_seq_len = 5
max_tgt_seq_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)

# 单词索引构成的句子
src_seq = [F.pad(torch.randint(1, max_num_src_words, (L,)), (0, max_src_seq_len-L)) for L in src_len]
tgt_seq = [torch.randint(1, max_num_tgt_words, (L,)) for L in tgt_len]


print(src_seq, tgt_seq)
```

```
[tensor([2, 5, 0, 0, 0]), tensor([6, 1, 4, 1, 0])] [tensor([5, 7, 7, 7]), tensor([7, 6, 4])]
```

从输出结果可以看出，句子已经被填充了，第一个句子长度本来是2（第一单词 是 2，第二个 单词是 5），填充成5，原序列的第二个样本 本来是 6,1,4,1，我们依然将其pad成5，每个句子的最大长度是5，目标序列同样的处理，还有一个细节，现在的输出 两个张量 放到了一个列表中，我们把它变成一个 tensor，二维tensor，也就是 batch size × max_seq_length 这样的二维tensor，通过torch.cat()函数实现

```
src_seq = [F.pad(
                  torch.randint(1,max_num_src_words,(L,)),
                  (0,max_src_seq_len - L)
                )
           for L in src_len
          ]
```

这样是不行的，还要把每个张量 变成二维的，用torch.unsqueeze，首先将每个样本变成二维的，然后再把两个样本 在第0维cat起来，代码

```
src_seq = [torch.randint(1,8,(L,)) for L in src_len]

max_num_src_words = 8

# src_len = tensor([7,3])
# torch.randint的参数 min, max, 形状
src_seq = [torch.randint(1,max_num_src_words,(L,)) for L in src_len]

# pad
[
  F.pad(torch.randint(1,max_num_src_words,(L,)),(0,max_src_seq_len - L))
  for L in src_len
]
# unsqueeze
[
  torch.unsqueeze(
    F.pad(torch.randint(1,max_num_src_words,(L,)),(0,max_src_seq_len -
L))     ,0)
  for L in src_len
]
# concat
torch.concat([
  torch.unsqueeze(
    F.pad(torch.randint(1,max_num_src_words,(L,)),(0,max_src_seq_len -
L))     ,0)
  for L in src_len
])
```

打印 输出 变成了 二维张量

In [7]:
```
import torch
import numpy
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

# 序列的最大长度
max_src_seq_len = 5
max_tgt_seq_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)

# 单词索引构成的句子
src_seq = torch.cat([torch.unsqueeze(F.pad(torch.randint(1, max_num_src_words, (L,)), (0, max_src_seq_len-L)), 0) \
                     for L in src_len])
tgt_seq = [torch.randint(1, max_num_tgt_words, (L,)) for L in tgt_len]


print(src_seq, tgt_seq)
```

```
tensor([[7, 2, 0, 0, 0],
        [4, 7, 2, 3, 0]]) [tensor([4, 1, 6, 7]), tensor([3, 3, 5])]
```

分步骤

```
# 单词索引 构成的句子
src_seq = [torch.randint(1,max_num_src_words,(L,)) for L in src_len]
print(src_seq)
# pad 填充
src_seq_pad = [F.pad(s,(0,max_src_seq_len - len(s))) for s in src_seq  ]
print(src_seq_pad)
# unsqueeze 变成二维 为了 concat
src_seq_pad_unsqueeze = [torch.unsqueeze(s,0) for s in src_seq_pad]
print(src_seq_pad_unsqueeze)
# concat
src_seq_pad_unsqueeze_concat = torch.concat([s for s in
src_seq_pad_unsqueeze])
print(src_seq_pad_unsqueeze_concat)
```

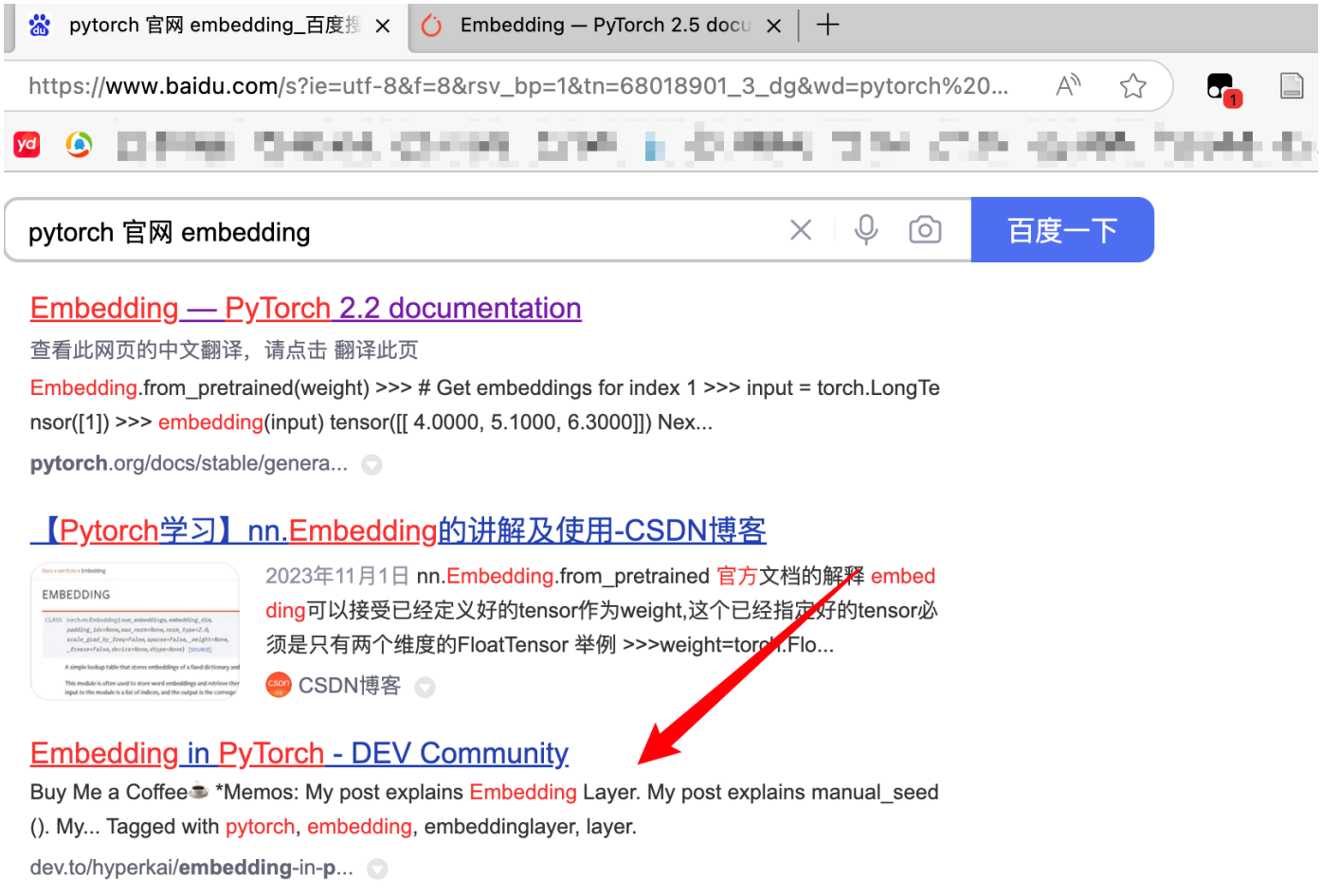
```
[tensor([7, 3]), tensor([1, 6, 4, 5])]
[tensor([7, 3, 0, 0, 0]), tensor([1, 6, 4, 5, 0])]
[tensor([[7, 3, 0, 0, 0]]), tensor([[1, 6, 4, 5, 0]])]
tensor([[7, 3, 0, 0, 0],
        [1, 6, 4, 5, 0]])
```

目标序列同理：

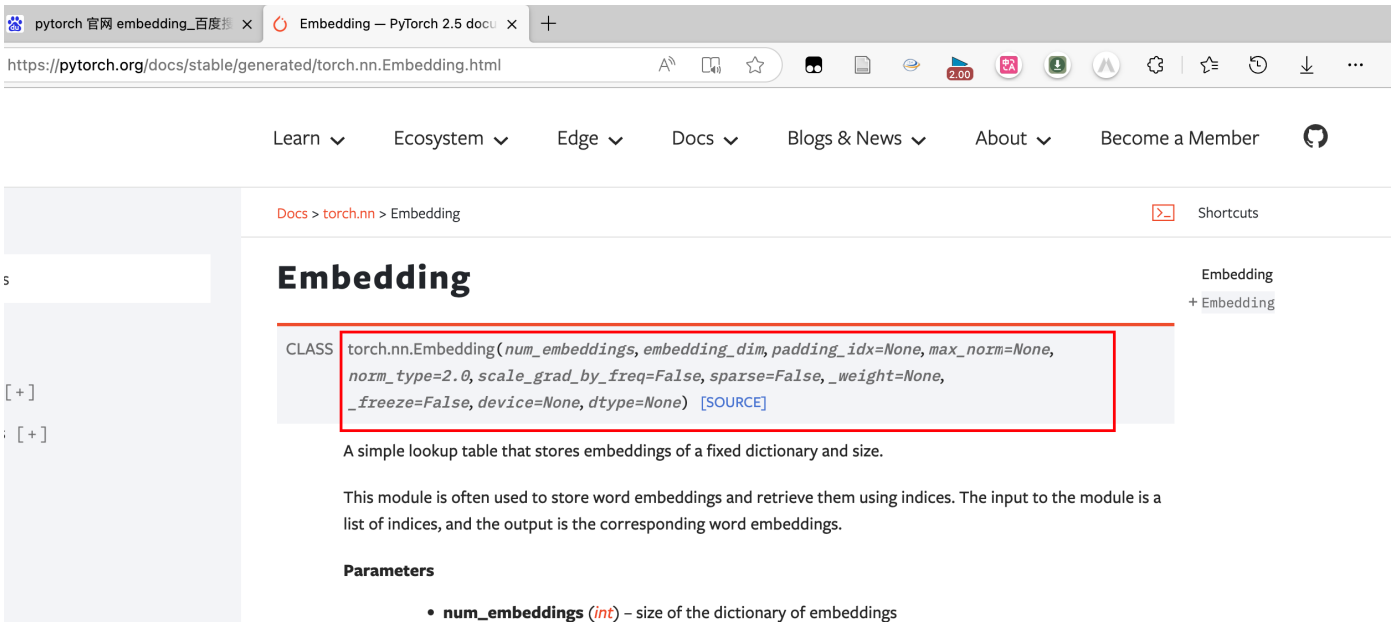
In [9]:
```
import torch
import numpy
import torch.nn as nn
import torch.nn.functional as F

# 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

# 序列的最大长度
max_src_seq_len = 5
max_tgt_seq_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)

# 单词索引构成的句子
src_seq = torch.cat([torch.unsqueeze(F.pad(torch.randint(1, max_num_src_words, (L,)), (0, max_src_seq_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_seq_len-L)), 0) \
                     for L in tgt_len])

print(src_seq)
print(tgt_seq)
```

```
tensor([[5, 7, 0, 0, 0],
        [3, 7, 3, 3, 0]])
tensor([[1, 5, 3, 5, 0],
        [6, 2, 1, 0, 0]])
```

batch size=2，一个batch 两个样本 默认填充0，单词索引构成了源句子和目标句子 并进行了padding，和 concat，以上得到了训练数据，接下来 构造embedding 怎么构造embedding？上网搜索 pytorch embedding的api



得到用法：



构造embedding，pytorch有api可以构造，有embedding类，通过索引获得每个单词的embedding，通过torch.nn.Embedding的api，构造embedding表，根据索引从表中取一行，embedding有两个参数，一个是num_embedings,也就是我们单词的数目要算上特殊字符P、S、E，第二个参数embedding_dim，嵌入维度

- **embedding_dim** (*int*) – the size of each embedding vector

因此，我们还需要 定义两个常量 一个是单词表大小`max_num_src_words` `max_num_tgt_words`，已经定义了，还有嵌入维度，model_dim，在原始论文中设置为512，在此举例子，方便观察，定义为8，在给出了单词表大小 和 嵌入维度，接下来 可以实例化一个 embedding类了，nn.Embedding(),分为 source embedding和target embedding，传入参数，单词表大小，max_num_src_words同时要记得+1，因为有padding在：nn.Embedding(max_num_src_words+1，)，接着传入model_dim，得到nn.Embedding(max_num_src_words+1，model_dim)，定义为src_embedding_table,准确来说是table

代码段：

```
model_dim = 8
src_embedding_table = nn.Embedding(max_num_src_words+1, model_dim)
```

同样的 定义 target embedding table

```
model_dim = 8
src_embedding_table = nn.Embedding(max_num_src_words+1, model_dim)
tgt_embedding_table = nn.Embedding(max_num_tgt_words+1, model_dim)
```

接下来打印weight，查看 embedding table，这个embedding table是一个二维矩阵，每一行代表一个embedding向量，第0行是pad字符的embedding 向量，从第1行到第9行 是词表中单词的embedding向量，每个单词的索引是多少，对应的就去embedding table里得到索引向量

```
print(src_embedding_table.weight)
```

```
# 构造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)


print(src_embedding_table.weight)
```

```
Parameter containing:
tensor([[ 0.1276, -1.3390,  0.2808, -0.7933,  2.2213, -0.7257,  0.9357, -0.3396],
        [ 0.3780,  1.2084,  0.1707,  0.2238,  0.1941,  0.5684, -0.1049,  0.6978],
        [ 0.3413,  0.6086, -1.7950, -1.0479,  0.0943, -0.3531, -0.9345,  1.8496],
        [-1.7034,  0.6219, -2.0887, -0.6529,  0.5925,  0.3618,  1.2209,  0.1621],
        [ 1.3406,  0.1540,  1.6920,  1.9183, -1.7418, -0.3558,  0.4391, -0.3817],
        [ 1.6231, -0.8980,  0.4954,  0.0464, -1.7121, -0.0821,  2.6978, -2.2113],
        [ 0.1899,  0.0714,  0.6193,  1.7602, -0.3290, -0.5700, -2.8945,  1.8768],
        [ 0.4374, -0.5439, -0.1498,  0.1564, -0.8546,  0.0413,  0.8716, -0.2054],
        [-2.1064,  0.2820, -0.6767,  1.1465, -1.4897, -0.6556, -0.8321, -2.2265]],
       requires_grad=True)
```

接下来，得到source embedding，把source sequence 丢进 embedding table，并打印一下source embedding

```
src_embedding = src_embedding_table(src_seq)
print(src_embedding)
```

```
Parameter containing:
tensor([[ 0.7634,  0.1148, -0.3328,  1.0741,  0.0869, -0.2325, -0.6342,  1.4446],
        [ 1.3140,  3.0582,  0.2585, -1.4647,  0.7533,  1.1781, -0.0379, -0.1095],
        [ 1.1563,  0.7174, -0.3724,  0.3480,  0.5916, -0.7421, -1.0576, -0.9683],
        [-1.6377,  1.7896,  1.7968,  1.8414,  0.4291,  0.4070, -1.8871, -0.9931],
        [ 0.7567,  2.1290,  2.4800, -0.6606, -0.9341, -0.6014, -0.2183,  0.5124],
        [ 0.7781, -0.2291, -1.4048, -0.7616,  1.1395, -0.8143,  0.8146, -0.9229],
        [ 0.0331,  0.7554,  1.7491, -0.6220,  0.3246, -0.9675,  1.0886,  0.3425],
        [ 0.5976,  0.4943,  0.0780, -0.0733,  0.2128,  1.6107,  0.5834,  1.7277],
        [ 0.7860,  0.7582,  0.5563,  0.3092,  0.0633, -1.8518, -1.3804, -0.8642]],
       requires_grad=True)
tensor([[[ 0.5976,  0.4943,  0.0780, -0.0733,  0.2128,  1.6107,  0.5834,
           1.7277],
         [ 0.7567,  2.1290,  2.4800, -0.6606, -0.9341, -0.6014, -0.2183,
           0.5124],
         [ 0.7634,  0.1148, -0.3328,  1.0741,  0.0869, -0.2325, -0.6342,
           1.4446],
         [ 0.7634,  0.1148, -0.3328,  1.0741,  0.0869, -0.2325, -0.6342,
           1.4446],
         [ 0.7634,  0.1148, -0.3328,  1.0741,  0.0869, -0.2325, -0.6342,
           1.4446]],

        [[ 0.7567,  2.1290,  2.4800, -0.6606, -0.9341, -0.6014, -0.2183,
           0.5124],
         [ 1.3140,  3.0582,  0.2585, -1.4647,  0.7533,  1.1781, -0.0379,
          -0.1095],
         [ 0.5976,  0.4943,  0.0780, -0.0733,  0.2128,  1.6107,  0.5834,
           1.7277],
         [ 0.5976,  0.4943,  0.0780, -0.0733,  0.2128,  1.6107,  0.5834,
           1.7277],
         [ 0.7634,  0.1148, -0.3328,  1.0741,  0.0869, -0.2325, -0.6342,
           1.4446]]], grad_fn=<EmbeddingBackward>)
```

source embedding是根据source sequence得到的，把source sequence也打印出来

```
print(src_embedding_table.weight)
print(src_seq)
print(src_embedding)
```

输出结果：

```
Parameter containing:
tensor([[ 1.3680,  0.5344, -1.2157,  0.8786,  0.6017,  1.3696, -0.7503,  1.3044],
        [-0.3313,  1.8852, -0.3040,  1.6263,  1.5244,  0.3444, -0.0674,  0.0115],
        [-0.9653, -1.6769, -0.0799, -0.2548, -0.4064,  0.9611,  0.2275,  2.2370],
        [-0.1803,  1.3554,  0.9487,  1.2418,  0.7036, -0.8762, -0.8453, -1.0251],
        [ 1.0649,  0.4704,  0.9894, -0.2527,  0.9672,  1.3744,  0.0438,  2.0881],
        [ 0.9510,  0.2358, -0.3233,  0.9433, -0.0379,  0.4729,  2.1471,  1.7742],
        [ 1.1321,  3.1257, -1.1504,  0.4150, -1.4767,  0.6883,  0.0959,  1.0365],
        [-0.8520, -0.6287,  0.4630,  0.0439, -1.1333,  0.1906, -1.5407, -1.7028],
        [ 0.2650, -0.3006, -0.9795, -0.0842,  2.0579,  0.8820,  0.0497,  1.4250]],
       requires_grad=True)
tensor([[7, 1, 0, 0, 0],
        [2, 7, 4, 3, 0]])
tensor([[[-0.8520, -0.6287,  0.4630,  0.0439, -1.1333,  0.1906, -1.5407,
          -1.7028],
         [-0.3313,  1.8852, -0.3040,  1.6263,  1.5244,  0.3444, -0.0674,
           0.0115],
         [ 1.3680,  0.5344, -1.2157,  0.8786,  0.6017,  1.3696, -0.7503,
           1.3044],
         [ 1.3680,  0.5344, -1.2157,  0.8786,  0.6017,  1.3696, -0.7503,
           1.3044],
         [ 1.3680,  0.5344, -1.2157,  0.8786,  0.6017,  1.3696, -0.7503,
           1.3044]],

        [[-0.9653, -1.6769, -0.0799, -0.2548, -0.4064,  0.9611,  0.2275,
           2.2370],
         [-0.8520, -0.6287,  0.4630,  0.0439, -1.1333,  0.1906, -1.5407,
          -1.7028],
         [ 1.0649,  0.4704,  0.9894, -0.2527,  0.9672,  1.3744,  0.0438,
           2.0881],
         [-0.1803,  1.3554,  0.9487,  1.2418,  0.7036, -0.8762, -0.8453,
          -1.0251],
         [ 1.3680,  0.5344, -1.2157,  0.8786,  0.6017,  1.3696, -0.7503,
           1.3044]]], grad_fn=<EmbeddingBackward>)
```

输出结果解读：

上面是embedding table

中间是source sentence的单词id

下面是根据id从table中获取的embedding vector，上面是第一个句子的embedding vector，下面是第二个句子的embedding vector 上面的第一行，就是第一个单词的embedding vector，就是embedding table的索引 7，对应的张量，1呢，就是embedding table中对应的 索引1 ，接下来后面三个单词是pad，对应的embedding table的第0行

综上我们得到了source embedding ， target embedding是类似的

```
# 单词索引构成源句子和目标句子，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_seq_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_seq_len-L)), 0) \
                     for L in tgt_len])

# 构造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)



print(src_embedding_table.weight)
print(src_seq)
print(src_embedding)
```

这其实也是调用，embedding class的forward方法，forward方法在pytorch中，相当于python普通的class中的call方法，就是对一个实例 后面直接加一个括号用的就是call方法或者forward方法

综上得到 输入的embedding和输出的embedding 《==》 source embedding & target embedding

以上是所有 word embedding的讲解

在实际的项目中 构建的时候 首先要把文本或者句子 变成一个个的数字，这个数字就是每个句子或者每个单词在词典中所在的位置，第0个位置 留给 pad字符，首先得到索引，单词索引构成源句子 和 目标句子

然后再构建batch，构建batch的时候，要加入pad，pad默认为0 ，得到minibatch的索引以后，构建 embedding，并且以索引为输入得到样本的embedding，构造好embedding以后，接下来构造位置编码positon encoding 或者position embedding，原论文中 给出了 公式：

In this work, we use sine and cosine functions of different frequencies:

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

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

上次的视频中讲了为什么用这样的位置编码，用 sin cos embedding，

第一个就是 泛化能力比较强，即使在测试集中，遇到的没有见过的序列长度 也能根据已有的序列长度，求一个线性方程 和 系数求出来；

第二个就是具有对称性也具有唯一性，每个位置上的embedding是确定的，

接下来说明 位置编码 如何构建，

首先 embedding是一个二维张量或者说一个矩阵，行数应该是我们设置的最大的训练的目标序列，训练的最大长度，每一列是d_model,所以首先 我们定义一个最大长度，

pos embedding的最大长度：max_positon_len = 5，首先 定义序列编码的最大长度等于5，也就是假设训练中 所有样本长度最大等于5，dim就是 model dim已经定义了，这个position embedding也就是分为奇数列 和 偶数列，奇数列用 cos函数 偶数列 用 sin函数，但不管是sin函数 还是 cos函数，里面的值 都是一样的，所以可以理解为 两个矩阵的相乘 或者 可以随机初始化一个二维张量，做两个循环，把值填上，另外一种方法 想成 两个矩阵的相乘，接下来看 怎么做，首先先表示里面的东西，

$$\frac{pos}{1000^{\frac{2i}{d_{model}}}}$$

里面有两个变量，一个是pos，一个是i,pos是从0到最大长度的一个枚举或者遍历，具体代码：

构造一个pos的序列,变量名 pos_matrix

```
max_positon_len = 5
# 构造positon embedding
pos_mat = torch.arange(max_position_len)
```

打印pos_mat，输出 tensor([0,1,2,3,4])，pos_mat每一行都是一样的数，然后变成二维矩阵。就是先表示括号里的东西，然后分别加上 sin和cos，得到目标矩阵的偶数列和奇数列

重复分析：首先把括号里的东西表示出来，括号里面的东西分为两个变量，一个是pos变量，一个i变量，pos变量决定着行，i变量决定着列，也就要构建两个矩阵相乘，pos这个矩阵每一行的值是一样，i这个矩阵每一列的值都是一样的，这样就很简单了，首先写pos矩阵，pos这个矩阵我们已经定义了pos_mat，但是目前只是一个一维向量，我们要把它构成二维矩阵，用 reshape方法，torch.arange(max_position_len).reshape((-1,1)),打印，输出：

```
# 构造position embedding
pos_mat = torch.arange(max_position_len).reshape((-1, 1))

print(pos_mat)
```

```
tensor([[0],
        [1],
        [2],
        [3],
        [4]])
```

得到了pos矩阵 pos矩阵的每一行是不变的，目前还没有做broadcast，只有一列

接下来构建列矩阵，列矩阵就是i，定义为i_mat，i_mat 一起写成 10000（一万！）的2i除以 d_{model} 次方

i_mat 写成 2i，之前定义的model_dim 等于8，可以写i_mat = torch.arange(0,2,8)间隔是2，最大是8，这样 构建一个序列 最小值是0 最大值是8 间隔为2，就是因为 不管是奇数列 还是偶数列，里面用到的 表达式 是 一样的 实际上 只有一半的数目 对照公式 $\frac{pos}{1000^{\frac{2i}{d_{model}}}}$

我们要除以 d_{model} 对应到代码

```
i_mat = torch.arange(0,8,2)/model_dim
```

同样要reshape，先reshape，再除，reshape的行数 是 一行；列数 就是 -1，接下来打印i_mat的值

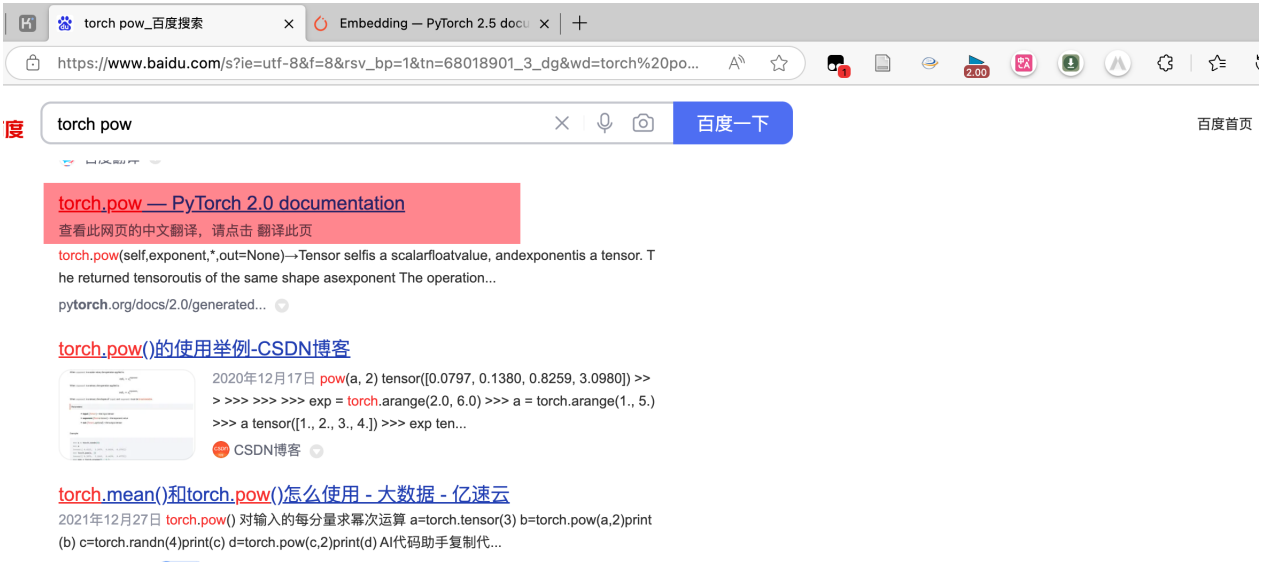
```
i_mat = torch.arange(0,8,2).reshape((1,-1))/model_dim
print(i_mat)
```

torch.arange的api分别是 min、max、patch

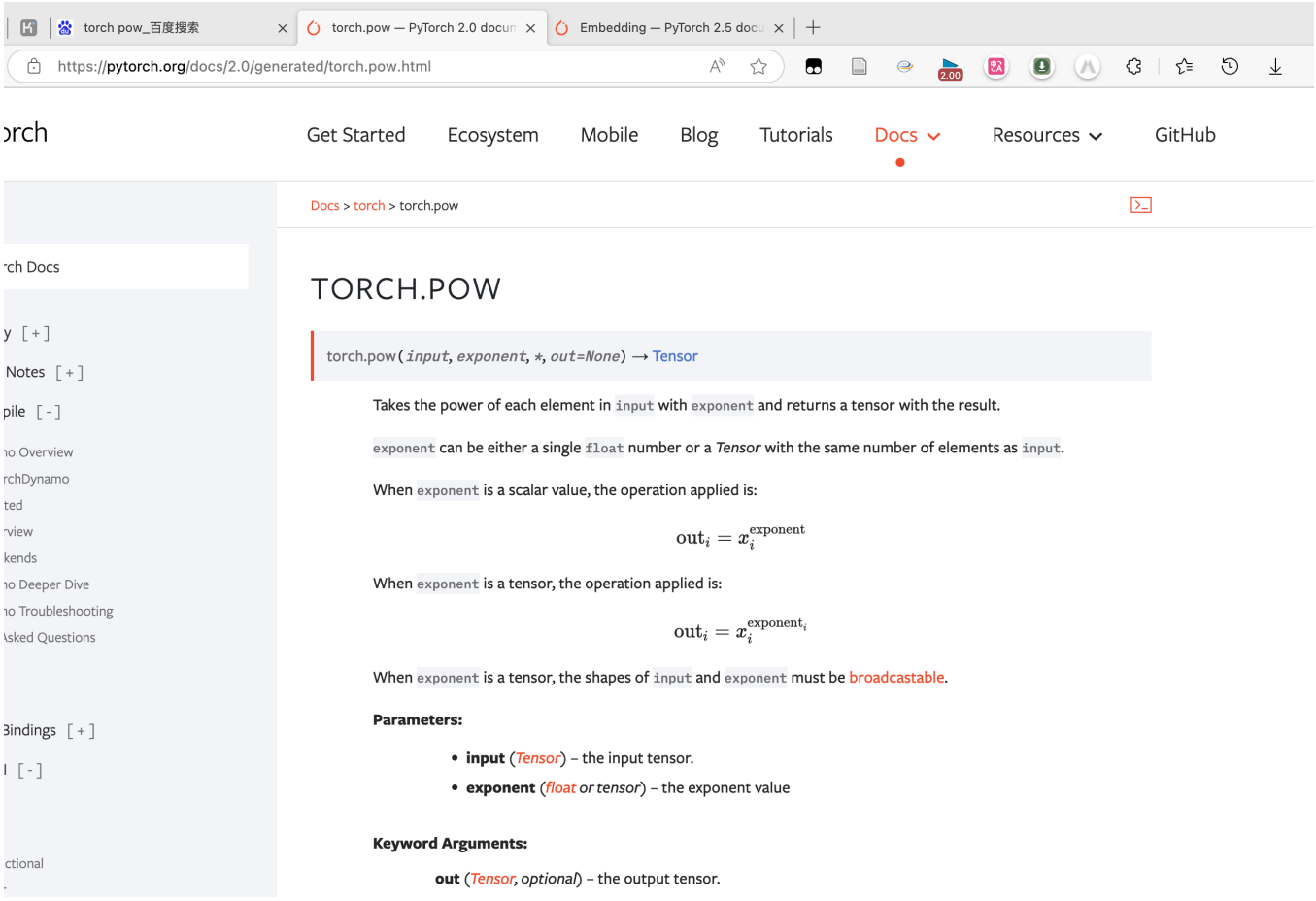
```
# 构造position embedding
pos_mat = torch.arange(max_position_len).reshape((-1, 1))
i_mat = torch.arange(0, 8, 2).reshape((1, -1))/model_dim

print(i_mat)
```

```
tensor([[0.0000, 0.2500, 0.5000, 0.7500]])
```

这样 得到了 i_mat矩阵1行4列 但是这个 i_mat矩阵还差一个10000（一万）次方，所以我们要搜一下 torch pow函数



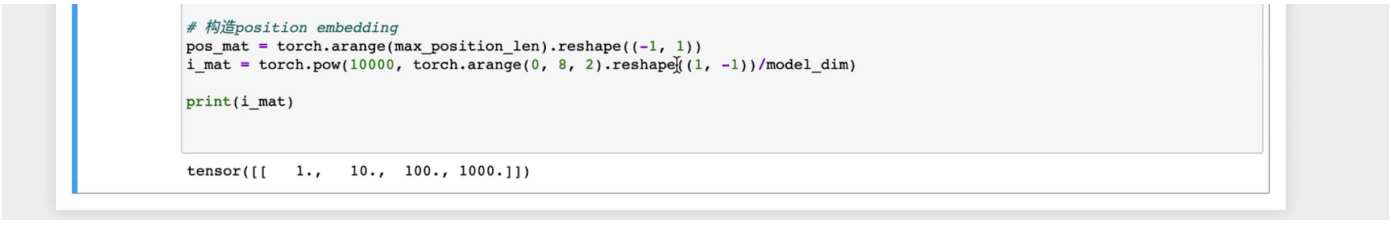
出现具体的用法：



具体的使用：

$$\frac{pos}{1000^{\frac{2i}{d_{model}}}}$$

```
i_mat = torch.pow(10000,torch.arange(0,8,2).reshape((1,-1))/model_dim)
print(i_mat)
```



10000为底，torch.arange(0,8,2).reshape((1,-1))/model_dim为幂的形式

我们已经得到了 pos_mat 还有i_mat，接下来构建 pe embeding table，首先随机初始化一个二维矩阵，大小分别是 max_position_len(总共的位置数据)× model_dim,

具体实现：

```
pe_embedding_table = torch.zeros(max_position_len,model_dim)
```

然后我们进行赋值就好了,分奇数和偶数

偶数列具体怎么表示呢？首先是 行不变，具体写法：pe_embedding_table[:,0::2]

接下来，偶数列 赋值为 sin函数 sin(pos_mat / i_mat)

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

```
# 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)

print(pe_embedding_table)
```

```
tensor([[ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.8415,  0.0000,  0.0998,  0.0000,  0.0100,  0.0000,  0.0010,  0.0000],
        [ 0.9093,  0.0000,  0.1987,  0.0000,  0.0200,  0.0000,  0.0020,  0.0000],
        [ 0.1411,  0.0000,  0.2955,  0.0000,  0.0300,  0.0000,  0.0030,  0.0000],
        [-0.7568,  0.0000,  0.3894,  0.0000,  0.0400,  0.0000,  0.0040,  0.0000]])
```

所有的偶数列 都有数了，现在看 奇数列 同样的方法 从1开始 cos函数

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

```
# 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)
```

```
tensor([[ 0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,  0.0000e+00,
          1.0000e+00,  0.0000e+00,  1.0000e+00],
        [ 8.4147e-01,  5.4030e-01,  9.9833e-02,  9.9500e-01,  9.9998e-03,
          9.9995e-01,  1.0000e-03,  1.0000e+00],
        [ 9.0930e-01, -4.1615e-01,  1.9867e-01,  9.8007e-01,  1.9999e-02,
          9.9980e-01,  2.0000e-03,  1.0000e+00],
        [ 1.4112e-01, -9.8999e-01,  2.9552e-01,  9.5534e-01,  2.9995e-02,
          9.9955e-01,  3.0000e-03,  1.0000e+00],
        [-7.5680e-01, -6.5364e-01,  3.8942e-01,  9.2106e-01,  3.9989e-02,
          9.9920e-01,  4.0000e-03,  9.9999e-01]])
```

如打印结果所示 实现了 positonal embedding 偶数列 是 sin函数 奇数列 是 cos 函数，这样是没有问题的，通过两个矩阵相乘的形式 构建 position embedding，当然也可以用两个for循环。分析为什么可以用矩阵相乘，我们分析公式

$$PE(pos, 2i) = sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

有两个变量 一个是 pos 一个是2i；pos变量代表行，i变量代表列；所以可以用两个矩阵相乘，一个矩阵 反应pos变化，一个矩阵 反应i变化，然后再利用 pytorch 的广播 broadcast 机制 也就是说 利用1×n的张量 乘以 一个 n×1 的张量，最终会变成一个n×n的张量，这里的乘 指的是 element-wise的乘法，而不是矩阵乘法，因为有广播机制，可以把张量 广播成同样的维度，然后再进行 逐个元素的 相乘 ，这样就得到 pe_embedding_table

得到 table以后，得到每个序列的 位置编码 pe_embedding，来看怎么得到，首先在 src_seq得到每个单词的id，首先实例化一个nn.Embedding，Embedding的行数是 max_position_len，列数或者dim维度就是 model_dim,变量名 pe_embedding

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

pe_embedding有wieght参数 我们把weight参数 改写一下，改写成 nn.Parameter，Parameter以上面的pe_embedding_table作为输入，同时把requires_grad 设置成 false，不要更新梯度，打印pe_embedding.weight【为什么要做这步？】,同时打印 pe_embedding_table

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

```
# 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)
print(pe_embedding.weight)
```

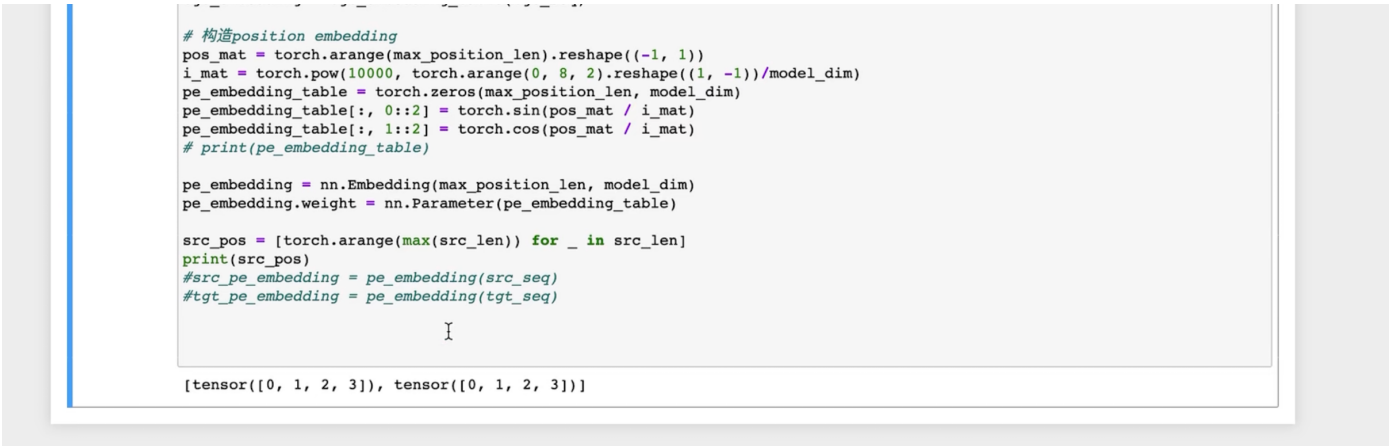
```
tensor([[ 0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,  0.0000e+00,
          1.0000e+00,  0.0000e+00,  1.0000e+00],
        [ 8.4147e-01,  5.4030e-01,  9.9833e-02,  9.9500e-01,  9.9998e-03,
          9.9995e-01,  1.0000e-03,  1.0000e+00],
        [ 9.0930e-01, -4.1615e-01,  1.9867e-01,  9.8007e-01,  1.9999e-02,
          9.9980e-01,  2.0000e-03,  1.0000e+00],
        [ 1.4112e-01, -9.8999e-01,  2.9552e-01,  9.5534e-01,  2.9995e-02,
          9.9955e-01,  3.0000e-03,  1.0000e+00],
        [-7.5680e-01, -6.5364e-01,  3.8942e-01,  9.2106e-01,  3.9989e-02,
          9.9920e-01,  4.0000e-03,  9.9999e-01]])
Parameter containing:
tensor([[ 0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,  0.0000e+00,
          1.0000e+00,  0.0000e+00,  1.0000e+00],
        [ 8.4147e-01,  5.4030e-01,  9.9833e-02,  9.9500e-01,  9.9998e-03,
          9.9995e-01,  1.0000e-03,  1.0000e+00],
        [ 9.0930e-01, -4.1615e-01,  1.9867e-01,  9.8007e-01,  1.9999e-02,
          9.9980e-01,  2.0000e-03,  1.0000e+00],
        [ 1.4112e-01, -9.8999e-01,  2.9552e-01,  9.5534e-01,  2.9995e-02,
          9.9955e-01,  3.0000e-03,  1.0000e+00],
        [-7.5680e-01, -6.5364e-01,  3.8942e-01,  9.2106e-01,  3.9989e-02,
          9.9920e-01,  4.0000e-03,  9.9999e-01]])
```

可以看到 pe_embedding_table,pe_embedding.weight是一样的，这是因为 构建了一个nn.Embedding，然后把weight改写一下，然后直接用pe_embedding作为一个函数，把上面的source sequence输入进来，得到source_pe_embedding

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

这里报错 index out of range，up主自己给自己写晕了，参数传错了，想讲的思想是是 借助pytorch embedding的api，来去以word embedding一样的方式 得到pe_embedding，只不过是把 embedding里的 weight给修改一下

src_seq是构成句子的单词的索引，需要传入的参数是 位置索引 ，简单来说 就是 基于source len和target len传，首先生成 source position，对source len 进行遍历，然后通过 torch.arange生成一系列的位置索引src_pos = [torch.arange(L) for L in src_len],每个source的sentence句子长度不一样，所以改成：src_pos = [torch.arange(max(src_len)) for _ in src_len] 并打印 src_pos print(src_pos);这里的思想是在说 我们的pe_embedding是以source pos作为输入的，我们可以打印一下：



```
# 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)

src_pos = [torch.arange(max(src_len)) for _ in src_len]
print(src_pos)
#src_pe_embedding = pe_embedding(src_seq)
#tgt_pe_embedding = pe_embedding(tgt_seq)
```

```
[tensor([0, 1, 2, 3]), tensor([0, 1, 2, 3])]
```

这样我们得到了原序列的pos，再把pos传入pe_embedding,tgt_pos是一样的操作，同样写一个target pos，而刚刚报错是因为传入的是单词索引，应该传入位置索引：

torch.cat torch.unsqueeze都是一点点尝试出来的

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

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)
```

输出结果：

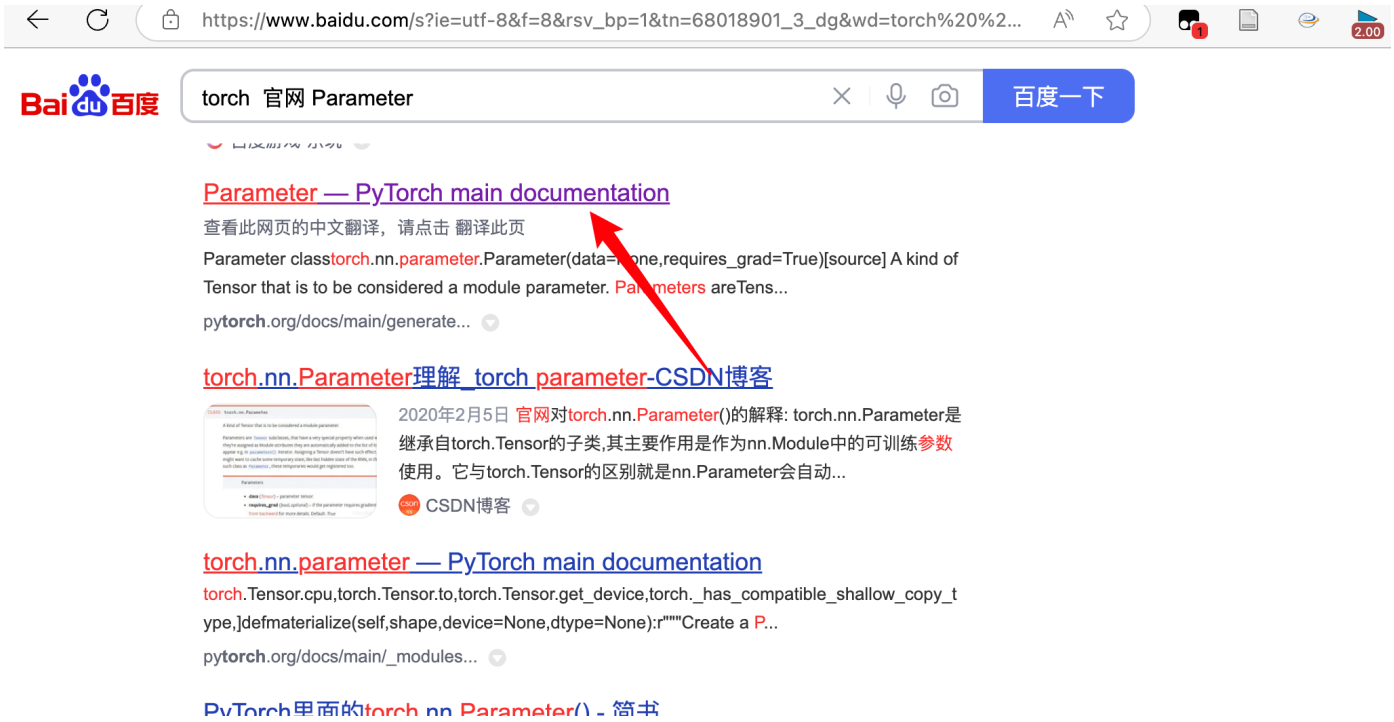
```
tensor([[[ 0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,
 0.0000e+00,
           1.0000e+00,  0.0000e+00,  1.0000e+00],
         [ 8.4147e-01,  5.4030e-01,  9.9833e-02,  9.9500e-01,  9.9998e-
03,
           9.9995e-01,  1.0000e-03,  1.0000e+00],
         [ 9.0930e-01, -4.1615e-01,  1.9867e-01,  9.8007e-01,  1.9999e-
02,
           9.9980e-01,  2.0000e-03,  1.0000e+00],
         [ 1.4112e-01, -9.8999e-01,  2.9552e-01,  9.5534e-01,  2.9995e-
02,
           9.9955e-01,  3.0000e-03,  1.0000e+00]],

        [[ 0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,
 0.0000e+00,
           1.0000e+00,  0.0000e+00,  1.0000e+00],
         [ 8.4147e-01,  5.4030e-01,  9.9833e-02,  9.9500e-01,  9.9998e-
03,
           9.9995e-01,  1.0000e-03,  1.0000e+00],
         [ 9.0930e-01, -4.1615e-01,  1.9867e-01,  9.8007e-01,  1.9999e-
02,
           9.9980e-01,  2.0000e-03,  1.0000e+00],
         [ 1.4112e-01, -9.8999e-01,  2.9552e-01,  9.5534e-01,  2.9995e-
02,
           9.9955e-01,  3.0000e-03,  1.0000e+00]]],
       grad_fn=<EmbeddingBackward0>)

tensor([[[ 0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,
 0.0000e+00,
           1.0000e+00,  0.0000e+00,  1.0000e+00],
         [ 8.4147e-01,  5.4030e-01,  9.9833e-02,  9.9500e-01,  9.9998e-
03,
           9.9995e-01,  1.0000e-03,  1.0000e+00],
         [ 9.0930e-01, -4.1615e-01,  1.9867e-01,  9.8007e-01,  1.9999e-
02,
           9.9980e-01,  2.0000e-03,  1.0000e+00],
         [ 1.4112e-01, -9.8999e-01,  2.9552e-01,  9.5534e-01,  2.9995e-
02,
           9.9955e-01,  3.0000e-03,  1.0000e+00]],

        [[ 0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,
 0.0000e+00,
```

```
          1.0000e+00,  0.0000e+00,  1.0000e+00],
        [ 8.4147e-01,  5.4030e-01,  9.9833e-02,  9.9500e-01,  9.9998e-
03,
          9.9995e-01,  1.0000e-03,  1.0000e+00],
        [ 9.0930e-01, -4.1615e-01,  1.9867e-01,  9.8007e-01,  1.9999e-
02,
          9.9980e-01,  2.0000e-03,  1.0000e+00],
        [ 1.4112e-01, -9.8999e-01,  2.9552e-01,  9.5534e-01,  2.9995e-
02,
          9.9955e-01,  3.0000e-03,  1.0000e+00]]],
       grad_fn=<EmbeddingBackward0>)
```

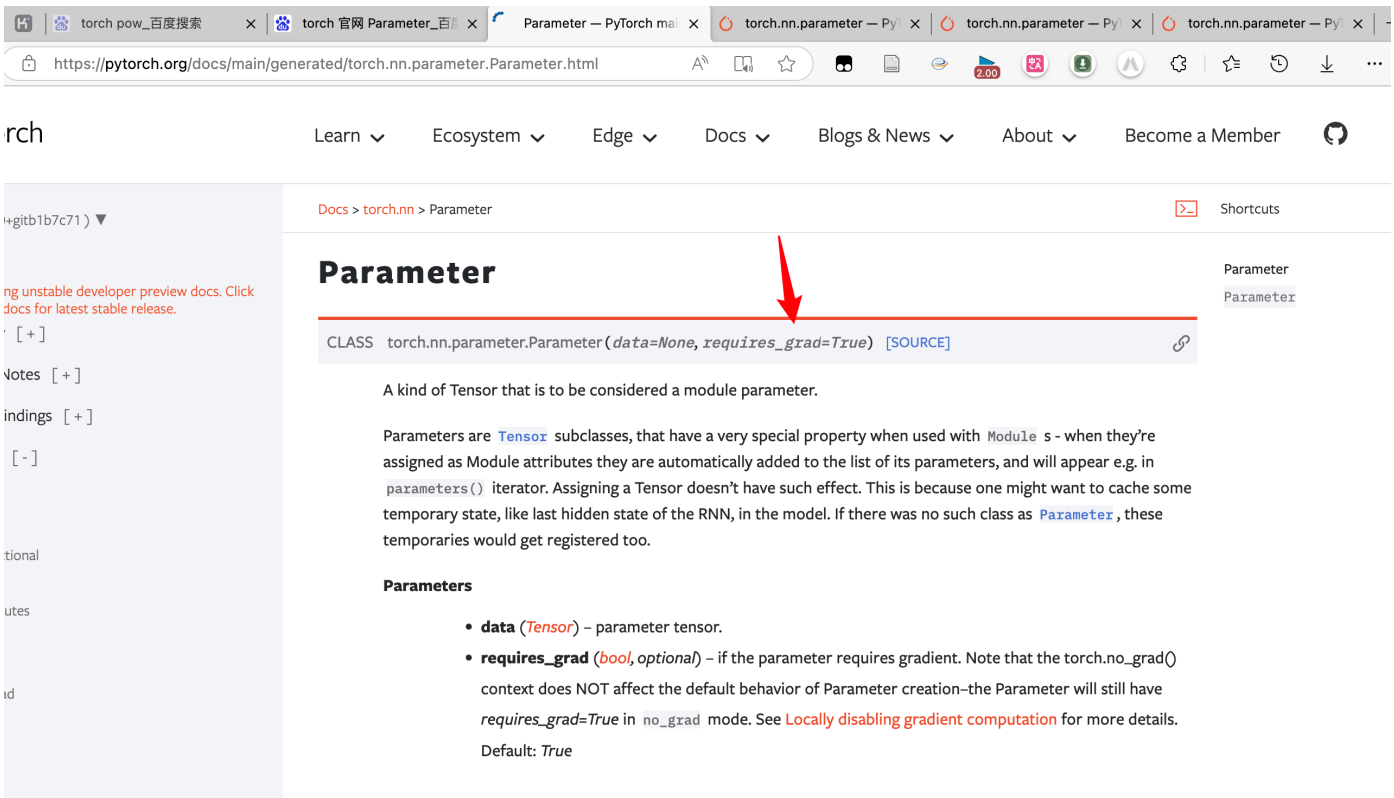
解读输出结果，model_dim=8，每个位置的编码是一样，pos=1对应于不同的单词在句子中的位置保持不变，对于pos=1，model_dim有不同，奇数用cos公式，偶数用sin公式,分别得到了 source embedding和target embedding


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

这个搜对了 关键词的重要性：



参数是： `requires_grad=True`



经过修改 我们得到了pe_embedding,并且把梯度设置成了 false

经过上面的步骤，我们构造了position embedding，构造的思路 是借助pytorch的nn.Embedding的api通过位置索引，直接得到位置embedding，运行错误的原因是传入的不是pos index而是 word index，word index指的是 word在单词中的索引，注意理解是怎么构造的pos embedding

以上是所有word embedding和position embedding的全部内容，接下来是 encoder self attention mask，看原文模型图

3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. We compute the matrix of outputs as:

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

文章用的是 缩放点积注意力，去掉Scaled，dot-product attention是很常见的就是乘法的注意力机制，注意力机制本质上就是对一个序列计算出新的表征，这个表征在注意力机制中是根据权重 对序列 加权求和；全连接网络也是一个表征，比如我们实例化一个全连接网络，nn.Linear(2,3)将一个二维的映射到三维的，只不过映射的权重既不是归一化的权重，也不是注意力算出来的，而是Linear层随机初始化的一个W，然后把W的每一列与这个2，输入的一行进行 加权求和得到一个新的映射，只不过 既不是归一化的 也不是注意力算出来的。

在注意力中 权重是算出来的，且是归一化的，通过softmax归一化的，权重怎么算呢，是根据Q和K的相似度算出来的，也就是query和key，总之在encoder中，Q和K都是基于word embedding经过两个linear层得到的Q和K，对于单个单词而言，两两单词之间，Q和K就是两个向量 算一个内积，就是两两相乘 再相加，这是一个单词与另一个单词的相似度，我们把这个单词跟这个序列中的所有单词 的相似度都算出来，然后再softmax得到一个归一化的概率，归一化居然不是缩放的作用，softmax的输入是负无穷到正无穷的，softmax的计算是e的x除以下面是一个求和符号e的x，就是一个归一化，所以softmax的输出用的是0到1之间，这样我们把一个负无穷到正无穷的范围简化到了0到1之间，同时softmax函数也是单调的，也就是括号里面相似度越大，softmax以后也会越大，是一个单调函数，保存相似度顺序，但又不是一个线性的放大，也就是说对于一个相似度的值，乘以0.1的输出 和 乘以0.2的概率输出 不是线性的

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

所以 需要缩放，scaled 除以根号dk，为了保证softmax概率出来以后，方差不要那么大，同时再往深层挖一点，就是说雅可比导数不要变成0

再重复一次，scaled的意思就是希望模型能够学好，如果dk很大，又不除以根号dk，softmax出来以后分布就很尖锐，雅可比矩阵 就会很小 通过代码进行演示

当前已有的所有代码

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

# --- 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_seq_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_seq_len-L)),0)
                                 for L in tgt_len])

# 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)

# 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)
```

首先实现 softmax部分：首先随机生成相似度结果score向量torch.randn()通过正态分布随机生成，假设长度是5，打印socre

```
# softmax演示
score = torch.randn(5)  # = QK^T
print(score)
```

接下来 使用softmax 进行归一化，归一化的维度是-1维，也就是最后一维

```
prob = F.softmax(score,-1)
print(prob)
```

```
# softmax演示
score = torch.randn(5)
prob = F.softmax(score, -1)
print(score)
print(prob)
```

```
tensor([-0.4324, -0.2013, -1.8387, -1.1955, -1.3862])
tensor([0.2979, 0.3754, 0.0730, 0.1389, 0.1148])
```

解读 输出结果 第一个结果 是 score，第二个结果 是 score归一化以后得prob，prob表示query这个单词，跟整个序列中每个单词的相似度，prob越大，表示两个单词之间的关联性越大，演示的目的是 对 socre乘以一个系数，定义一个常量alpha，设alpha1 = 0.1，alpha2 = 10，然后 我们把score分别乘 alpha1和alpha2，看分别有什么变化

```
# softmax演示
alpha1 = 0.1
alpha2 = 10
score = torch.randn(5)  # = QK^T

prob1 = F.softmax(score*alpha1,-1)
prob2 = F.softmax(score*alpha2,-1)

print(prob1)
print(prob2)
```

```
# softmax演示
alpha1 = 0.1
alpha2 = 10
score = torch.randn(5)
prob1 = F.softmax(score*alpha1, -1)
prob2 = F.softmax(score*alpha2, -1)

print(prob1)
print(prob2)
```

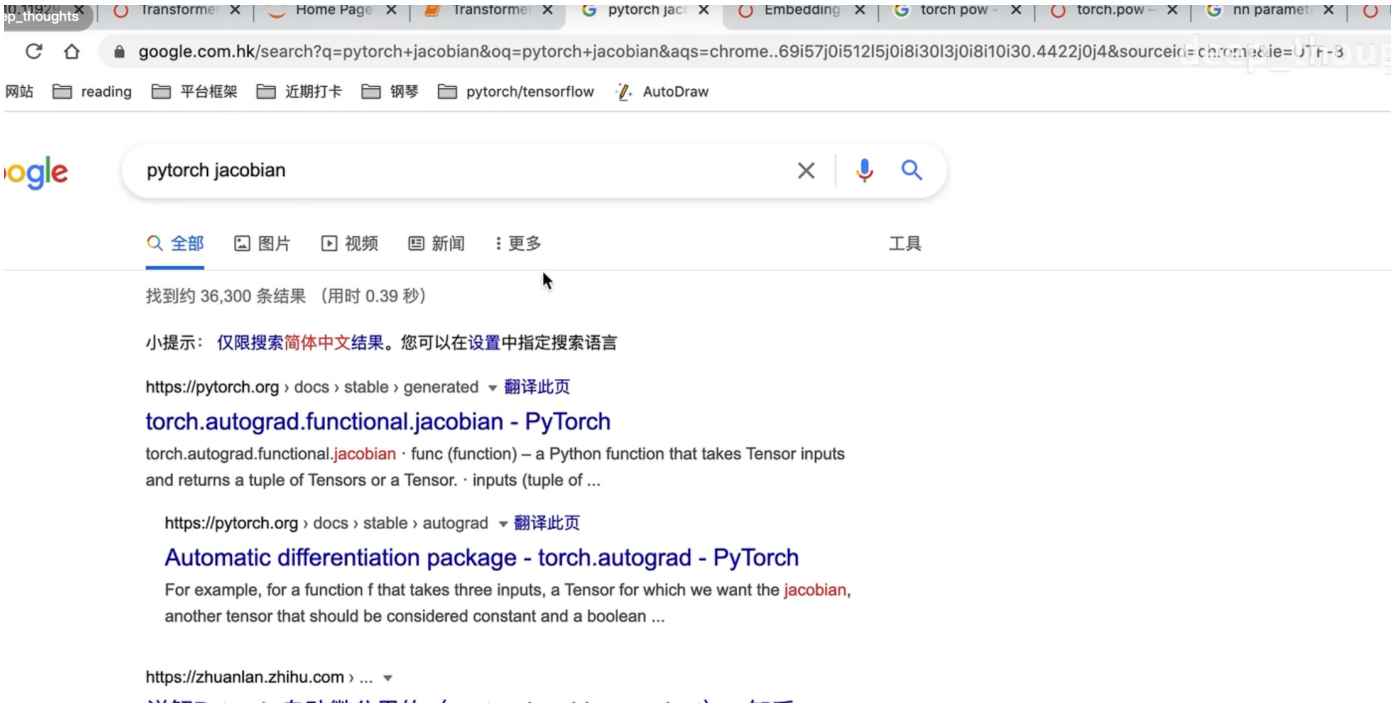
```
tensor([0.1969, 0.2135, 0.2080, 0.2051, 0.1765])
tensor([2.7361e-04, 9.1754e-01, 6.6278e-02, 1.5913e-02, 4.8829e-09])
```

查看 输出结果 第一行就对score乘以0.1的结果，结果方差没那么大，差不多

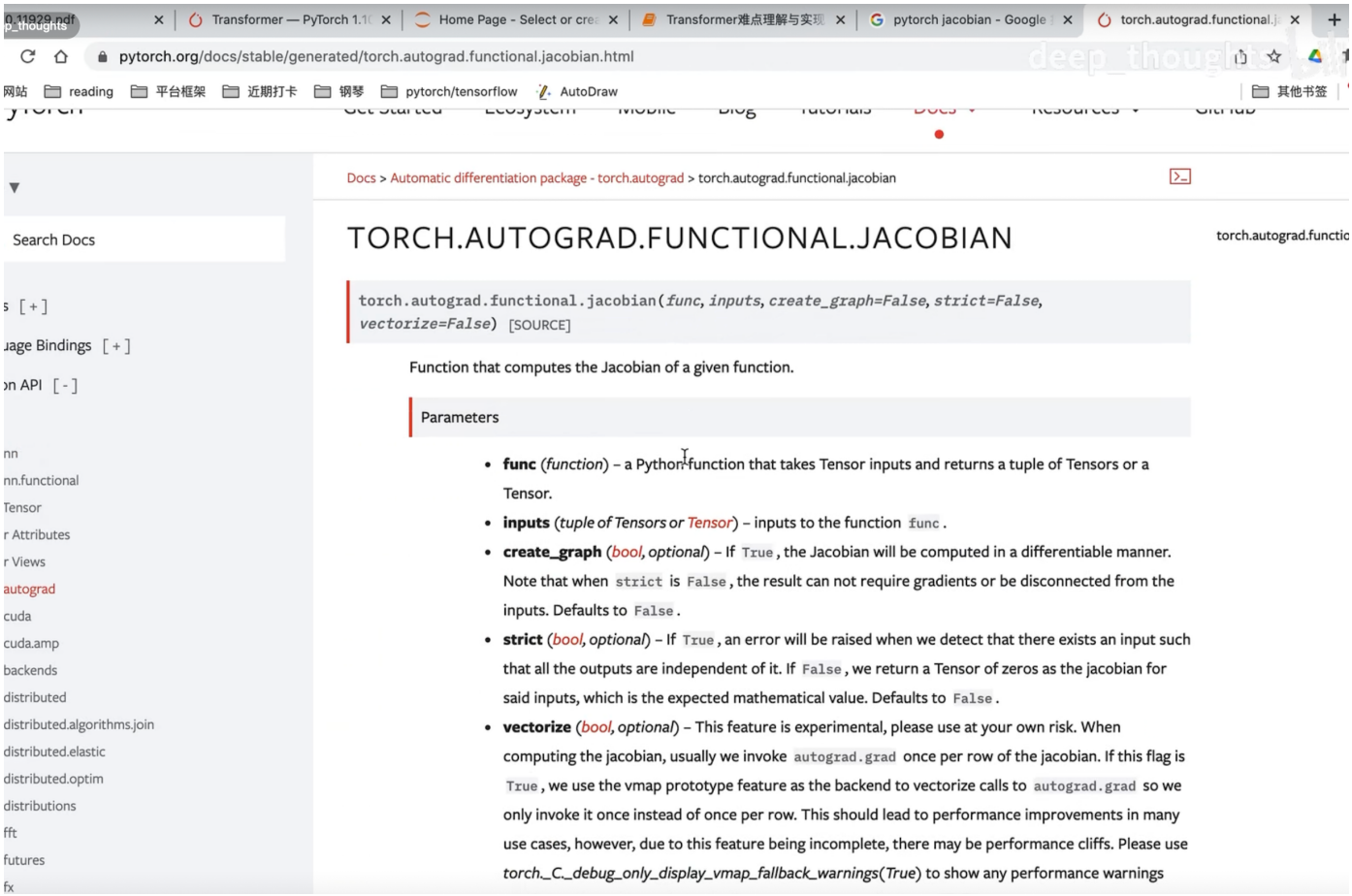
对score乘以10的时候，这时候概率差别就很大了，大的是0.9，小的就接近于0了

如上演示了 如果我们对score进行一个缩放的话 并不是一个线性的，乘以一个越大的值，概率大的越来越大，概率小的越来越小，这是在概率上反映的一个结果，如果是用雅可比的话又是怎么样的呢?

首先 回顾一下pytorch的雅可比

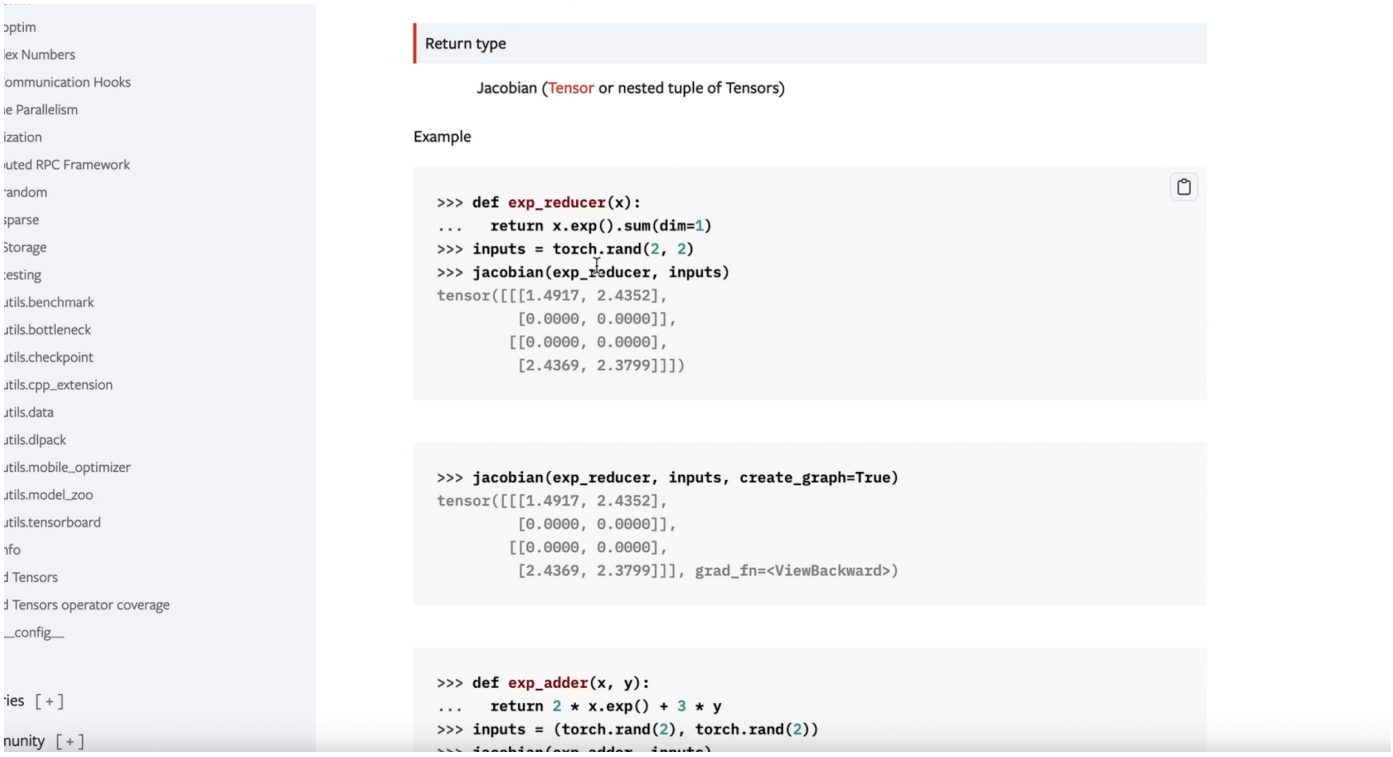


up主在自动微分 有讲过这个api，可以求两个向量之间的雅可比矩阵



看例子：

构建一个函数，有一个输入进去，就能算出输入对输出的雅可比矩阵



首先 构建一个函数softmax_func,函数接收socre作为输入，返回就是F.softmax(score),接下来计算雅可比矩阵，雅可比矩阵的api torch.autograd.functional.jacobian(),第一个参数 传入 函数名称，第二个参数 传入score，定义一个变量joca_mat,接下来 对比socre乘以alpha1，得到joca_mat1，和score乘以alpha2

```
# softmax演示
alpha1 = 0.1
alpha2 = 10
score = torch.randn(5)
prob1 = F.softmax(score*alpha1,-1)  # -1的作用是什么?
prob2 = F.softmax(score*alpha2,-1)

def softmax_func(score):
    return F.softmax(score)

jaco_mat1 =
torch.autograd.functional.jacobian(softmax_func,score*alpha1)
jaco_mat2 =
torch.autograd.functional.jacobian(softmax_func,score*alpha2)

print(jaco_mat1)
print(jaco_mat2)
```

```
# softmax演示
alpha1 = 0.1
alpha2 = 10
score = torch.randn(5)
prob1 = F.softmax(score*alpha1, -1)
prob2 = F.softmax(score*alpha2, -1)
def softmax_func(score):
    return F.softmax(score)
jaco_mat1 = torch.autograd.functional.jacobian(softmax_func, score*alpha1)
jaco_mat2 = torch.autograd.functional.jacobian(softmax_func, score*alpha2)

print(jaco_mat1)
print(jaco_mat2)
```

```
tensor([[ 0.1335, -0.0359, -0.0356, -0.0305, -0.0314],
        [-0.0359,  0.1751, -0.0509, -0.0436, -0.0448],
        [-0.0356, -0.0509,  0.1742, -0.0432, -0.0445],
        [-0.0305, -0.0436, -0.0432,  0.1554, -0.0381],
        [-0.0314, -0.0448, -0.0445, -0.0381,  0.1588]])
tensor([[ 2.4486e-16, -1.6752e-16, -7.7339e-17, -1.4364e-23, -2.4906e-22],
        [-1.6752e-16,  2.1609e-01, -2.1609e-01, -4.0133e-08, -6.9588e-07],
        [-7.7339e-17, -2.1609e-01,  2.1609e-01, -1.8528e-08, -3.2127e-07],
        [-1.4364e-23, -4.0133e-08, -1.8528e-08,  5.8662e-08, -5.9668e-14],
        [-2.4906e-22, -6.9588e-07, -3.2127e-07, -5.9668e-14,  1.0172e-06]])
```

第一个矩阵 score乘以0.1的效果，这个矩阵相当于在做梯度后向传播的时候，中间的链式的梯度传播，乘以0.1的时候，可以看到中间的数值是比较稳定的，没有出现梯度消失的问题，

如果我们对梯度乘以10的话，可以看到雅可比矩阵 很多地方都变成了0，如果梯度导数变成0的话，就无法训练了，因为后面的参数接收不到梯度信号了 参数也就无法更新了

以上这个例子说明，在数学上可以证明 如果Q和K的向量的点乘 方差是dk的，方差是比较大的，所以我们可以来除以一个根号dk，这样的数 方差就变成了1 ，目的就是把score 方差变得小一点，于是softmax概率出来以后就不会那么尖锐，那另外一方面，雅可比矩阵出来 导数不会变成0

以上的讲解 回答了 scale的重要性 什么是缩放

接下来 讲解 encoder self-attention mask

```
# 构造encoder的self attention mask
```

构造mask之前，首先看一下公式

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

一般我们把mask放到哪里呢》一般是放到softmax里面的，也就是说我们希望被mask的值，在softmax里面变成负无穷的，softmax是一个单调函数，如果输入是负无穷的话 输出就接近于0，也就是输出概率为0，刚好反映了两两单词之间的关联性变成0

所以我们需要构造一个mask矩阵，这个矩阵如果是跟score相乘的话，要么元素为1，要么元素为负无穷，并且这个mask矩阵，和原矩阵维度是一样的，如果我们做mini batch训练的话，里面的矩阵是一个 batch size×t，为什么呢？因为Q的大小是batch size×t(embedding的维度)，key的大小也是 batch size×t(embedding的维度)，K还有一个转置 所以两个矩阵 可以相乘，最后的结果是 t × t的一个量，t是序列长度，简单理解就是

一句话中单词的个数 或者 max_seq_len

接下来，我们构建一个mask，mask的形状：batch size × max_src_len × max_src_len，就是encoder 的self attention mask的shape是 batch size × max_src_len × max_src_len，跟括号里 $\frac{QK^T}{\sqrt{d_k}}$ 结果的维度是一样的，进行的是element-wise的结果的相乘，且值为1 或者 -inf负无穷，负无穷的话 经过softmax归一化结果为0，1的话 相乘归一化结果不变

具体怎么做呢？要用到之前 src_len 这个量，src_len长度是 batch size，每个元素代表的每个样本的长度，于是我们要构建mask的话，首先构建一个valid_encoder_pos,这个什么意思呢？就是 我们首先构建一个有效的编码器的位置,怎么构建呢？首先对src_len 进行一个遍历，`src_len = [2,4]`，src len的第一个元素是2，说明首先构建一个torch.ones,`valid_encoder_pos = torch.ones(L) for L in src_len`,构建一个有效矩阵，有效矩阵就代表这个位置是有效的，

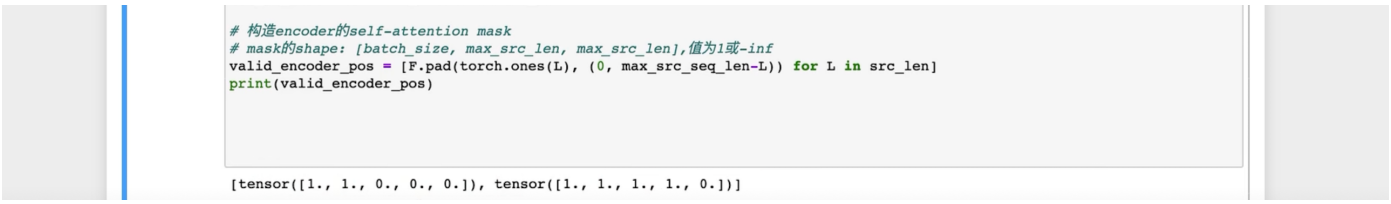
如果是0的话 代表这个位置是无效的，然后对 src_len 进行一个遍历，src_len就是每个序列的一个长度，第一向量是2的话，代表这个向量有两个1，但是pos 应该是一个矩阵 一个张量，所以等下我们会进行一个pas，现在我们先把结果打印一下，

```
valid_encoder_pos =[torch.ones(L) for L in src_len]

print(valid_encoder_pos)
```

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

以上 得到了原序列的 有效位置，第一个句子有效位置是前两个，第二个句子，有效位置是前四个，但是训练的时候，我们以最大长度训练，也就是之前定义的 max_src_seq_len来构建mini batch，所以我们需要对有效位置进行padding操作，把句子padding成最大长度，F.pad()，第一个参数接收要pad的对象，第二个参数接收要pad的长度，左边不pad，右边pad成最大长度，具体代码：

```
valid_encoder_pos = [F.pad(torch.ones(L),(0,max_src_seq_len - L)) for L
in src_len]

print(valid_encoder_pos)
```



结果解读：

如图，我们把有效位置pad好了，也就说对于原序列而言，第一个句子有两个单词，第二个句子只有4个单词，但是我们都pad成5了，因为我们要做minibatch的训练，假设我们最大长度为

```
# 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_seq_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_seq_len-L)), 0) \
                     for L in tgt_len])
```

看一下之前的代码，我们都pad成 max_src_seq_len，无论是word embedding，当然了，也可以不自己指定max_src_seq_len，而是统一pad成 最大的句子长度

```
# 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_seq_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_seq_len-L)), 0) \
                     for L in tgt_len])

# 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)

# 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)

# 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 = [F.pad(torch.ones(L), (0, max_src_seq_len-L)) for L in src_len]
print(valid_encoder_pos)
```

改成这样的：

```
#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)

# 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])

# 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)

# 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)
```

```
tgt_pe_embedding = pe_embedding(tgt_pos)

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

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

我们改成 max(src_len)以后，就是把所有句子长度都pad成 最大的句子长度了，在我们src_seq中，最大句子长度是4，所以都pad成4了，所以上图输出结果长度都是4了。

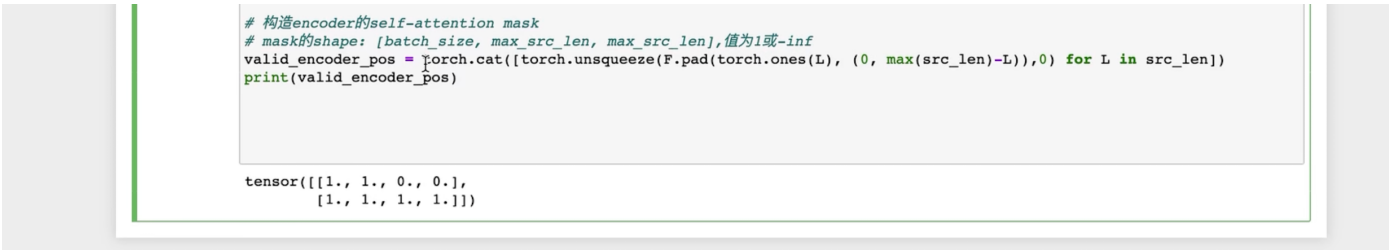
继续观察输出结果，现在的结果都是列表，我们把它改成一个张量，步骤如下：

把每个张量unsqueeze成一个二维张量，扩充维度 扩0维，这样就变成两个二位张量：


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

输出结果结果：变成两个二位张量的列表，然后进行cat操作，拼接，得到一个有效的编码器的位置矩阵，

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



batch size=2，一个batch里面两个样本，之前up主在度量学习讲过邻接矩阵，邻接矩阵就是反应元素 两两之间的一个相关性，现在我们也可以这样做，就是说我们把两个有效矩阵给乘起来，充分反应两两之间的相关性，但是这里我们每一行代表一个【没听清】，每一行之间是无关的，我们首先要扩维，

```
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]),1)
```

扩 哪一围呢 扩第一维

```
# 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]), 1)
print(valid_encoder_pos)
```

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

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

这样 我们得到了一个2×1×4的一个张量，我们打印它的shape：

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

# 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]), 1)
print(valid_encoder_pos.shape)
```

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

得到的是一个2×1×4的一个有效位置的张量，2表示batch size，4表示每个句子 pad以后的最大长度，我们把两个这样的矩阵 给它乘起来，就构成了一个2×4的邻接矩阵，做一个torch batch matrix multiply，torch.bmm()第一个位置

改了一个数字，变成 2×4×1的矩阵

```
# 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)
#torch.bmm()
print(valid_encoder_pos.shape)
```

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

对这两个矩阵相乘，它乘以它本身的一个转置，因为我们之前算邻接矩阵也是这样算的，两个向量的矩阵相乘，就能得到两两之间的关联性，代码：

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

我们首先查看形状 对不对

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

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

是 2×4×4的，接下来 我们查看具体的数值：

```
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))
print(valid_encoder_pos_matrix)
```

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

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

现在我们得到了两个邻接矩阵，结果解读：

第一个是第一个样本的，

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

第二个是 第二个 样本的

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

那么 再回顾一下，第一个样本的长度是2和4，第一个样本长度是2，第一行表示第一个句子的第一个单词 对其他单词的一个关系，第一个单词和第一个单词 都是可建立关联性的，因为这两个单词都是确实存在的，然后为什么后面两个元素使0呢？因为后面两个单词是pad的

把src len也打印出来 看得更清晰一点：

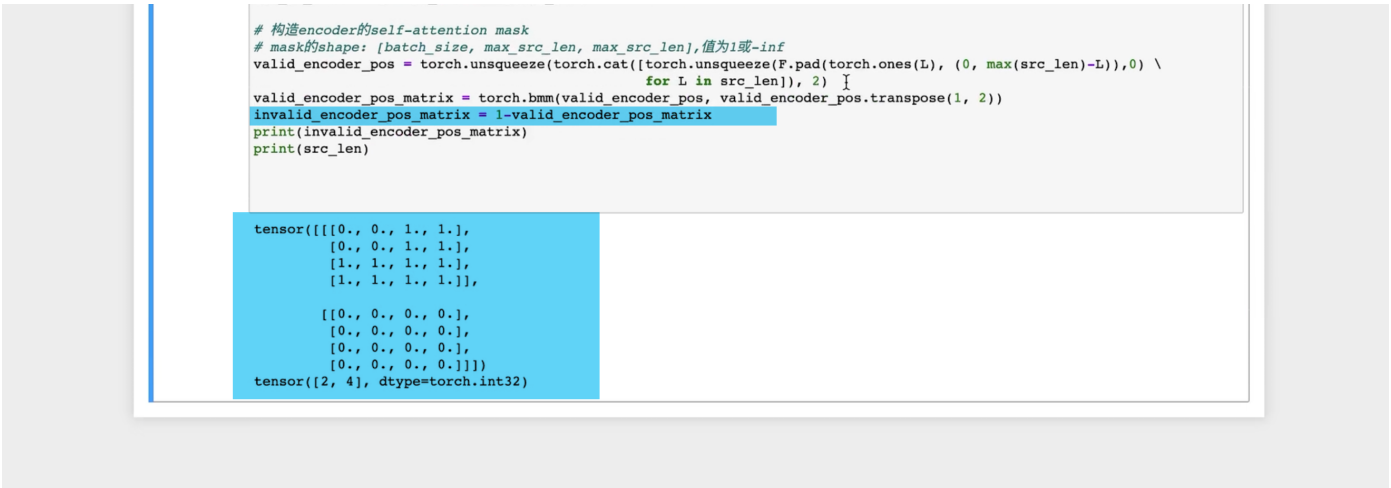
```
print(valid_encoder_pos_matrix)
print(src_len)
```

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

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
tensor([2, 4], dtype=torch.int32)
```

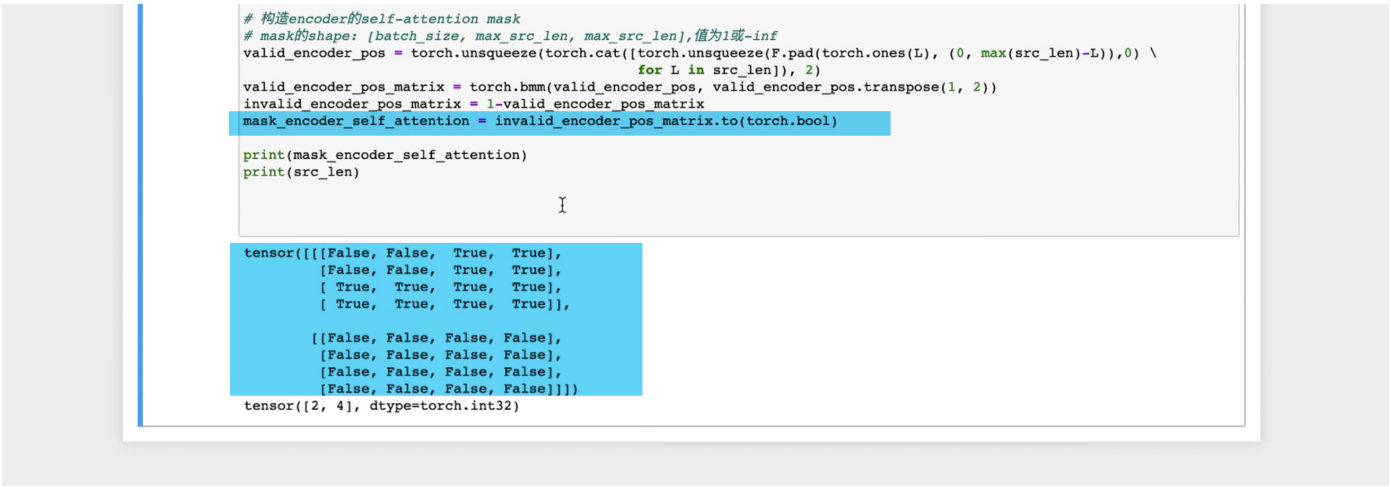
src len表示源序列 有两个句子，第一个句子长度是2 第二个句子长度是4，有效关联阵是4，因为都pad成了4，第一个矩阵（样本）的第一行，表示第一个单词 与 其他位置连接的有效性 [1., 1., 0., 0.]，第一个单词 肯定只和第一个单词 和 第二个单词 有效 因为后面两个单词都是 pad的，同样第二个单词 也只跟 第一个单词和第二个单词有效，第三行全是0 [0., 0., 0., 0.]，是因为第三个单词本身就是无效的 是pad来的，第四个单词 也是pad的，如此 解释完了第一个样本的邻接矩阵；

第二个样本 因为序列长度本身就是4，所以第二个样本所有单词之间 都是有效的；

有效矩阵的讲解结束，接下来讲解怎么得到mask，先计算 无效矩阵，计算方法1-有效矩阵



以上 是无效矩阵 编码 此时0就代表有效、 1代表无效、所有pad的地方 都是 无效的，接着我们把它变成bool型，然后打印



此时 mask矩阵 算出来了，true表示 那些位置 需要 mask，false表示 那些位置 不需要 mask

具体用怎么用呢，我们设置一个score torch.randn()　batch size应该是2，序列长度是 max(src_len),max(src_len)

```
score = torch.randn(batch_size,max(src_len),max(src_len))
```

score是个方阵，表示每个词对其他词的相关程度

我们把这个假想的score，使用score_masked_fill这样的函数，第一个传入的是bool型的张量，为true的地方，就mask成一个负无穷，并定义成 masked_score

```
masked_score = score.masked_fill(mask_encoder_self_attention,-np.inf)
```

并且对 masked_score进行一个 softmax，在-1维度进行计算，得到prob，并打印prob

```
prob = F.softmax(masked_score,-1)

print(prob)
```

同时打印 score，masked score、source len

总结一下 现在在做的事 就是生成一个mask encoder self attention 也就是说 编码器的自注意力的 mask矩阵，是一个bool矩阵，然后基于这个矩阵 来对 score 进行 一个 mask，具体 mask的条件，如果这个mask矩阵中 为 true的地方，就填充成 负无穷-inf 就是这样的 然后再对masked_score计算 softmax，来去看 prob 这里并不是一个三角矩阵 因为 encoder不涉及因果的mask

代码：

```
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, -np.inf)
prob = F.softmax(masked_score, -1)

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

结果：

```
prob = F.softmax(masked_score, -1)

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

```
tensor([2, 4], dtype=torch.int32)
tensor([[[-0.6906,  1.5425, -0.7806, -2.0940],
         [ 0.6842, -0.9452,  0.1954, -0.3177],
         [ 3.0649,  1.5718,  0.2037,  0.9629],
         [ 1.1305, -0.4425, -0.9613, -0.9523]],

        [[ 1.2172, -0.3087,  0.6016,  0.5752],
         [ 0.8417,  0.2233, -0.5997, -0.2892],
         [-0.1357, -0.5631, -0.5580, -0.0171],
         [ 0.3416, -0.5482,  0.5281,  2.3243]]])
tensor([[[-0.6906,  1.5425,    -inf,    -inf],
         [ 0.6842, -0.9452,    -inf,    -inf],
         [   -inf,    -inf,    -inf,    -inf],
         [   -inf,    -inf,    -inf,    -inf]],

        [[ 1.2172, -0.3087,  0.6016,  0.5752],
         [ 0.8417,  0.2233, -0.5997, -0.2892],
         [-0.1357, -0.5631, -0.5580, -0.0171],
         [ 0.3416, -0.5482,  0.5281,  2.3243]]])
tensor([[[0.0968, 0.9032, 0.0000, 0.0000],
         [0.8361, 0.1639, 0.0000, 0.0000],
         [   nan,    nan,    nan,    nan],
         [   nan,    nan,    nan,    nan]],

        [[0.4378, 0.0952, 0.2366, 0.2304],
         [0.4766, 0.2568, 0.1128, 0.1538],
         [0.2912, 0.1899, 0.1909, 0.3279],
         [0.1012, 0.0416, 0.1220, 0.7352]]])
```

结果解读：masked score这里

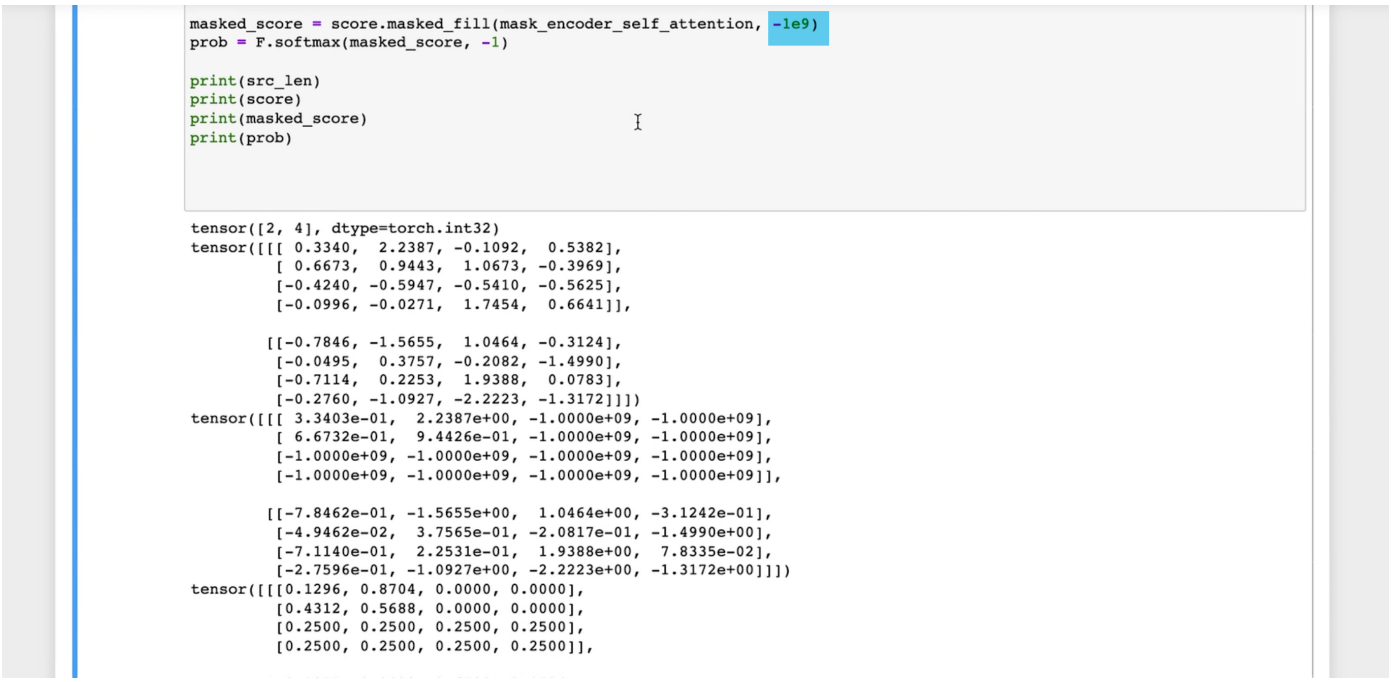
第一个是源序列的长度 source len [2,4]

第二个打印的是 原本的score

第三个打印的张量是 masked score

第四个打印的是 算出来的 注意力的一个权重

这里有一处小改动 没有用np.inf了



```
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)
```

```
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]],
```

第一个 source len

第二个 score

第三个 masked,pad的那个地方 都变成了 一个很小很小的数第一个句子序列长度为2 所以后面两列以及后面两行都变成了一个无效的score

第四个 得到masked score以后，计算prob，用softmax，权重 pad的地方进行mask 负无穷，在softmax的时候，概率输出为0；（观察prob的第一行）序列的第一个单词，只对前面的两个单词 是有注意力权重的 后面两个单词 注意力权重 都是0，因为后面mask掉的 变成负无穷；（prob的第二行）同样第二个单词也是一样的；（prob的第三行和第四行），可以看到 概率prob是均衡的，为什么呢？因为4个都是mask掉的，四个都mask掉了，在归一化的时候，就都是均匀概率了；后面第二个样本，得到的prob，因为没有进行pad，所以概率值都是很正常的，没有被mask掉的部分

以上是所有 encoder self attention mask的内容，就可以理解为一个关系矩阵，没有下三角，也不是因果的，encoder是一次性输入进去的；

复述一遍，masked encoder矩阵是如何得到的？

```
# 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
                                                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)
```

首先，我们是得到一个valid，得到一个有效矩阵，有效矩阵是通过两个向量之间进行一个矩阵相乘，就能得到一个两两之间的有效性，然后我们再对有效矩阵 取一个反，得到 无效矩阵，把无效矩阵 变成一个 bool型的，就得到一个 masked （masked encoder self attention）的矩阵，把masked的矩阵跟score进行一个逐元素的一个相乘，得到masked score，接下来 对 masked score求一个 softmax就可以得到attention的prob；观察prob可以看到 被mask的部分 元素为0，如果全部mask掉 那么概率就是相等的，相等的也会得到一个注意力的上下文，但是没关系，我们在最后在算loss的时候，会有一个loss的mask，也就是说在最后求损失函数的时候，同样还有一个mask，这个mask是对目标序列，进行mask，实现 的功能是 不该预测的位置 我们就不考虑它的交叉熵。

讲完了，讲的内容有：

- word embedding 怎么从单词索引得到word embedding
- Position embedding的构建
- 怎么得到 encoder的self attention mask

本节课 所讲的所有代码如下：

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

# --- 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])

# 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)

# 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)

# 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)
```

scale的重要性：

```
# softmax演示
alpha1 = 0.1
```

```
alpha2 = 10
score = torch.randn(5)
prob1 = F.softmax(score*alpha1,-1)  # -1的作用是什么?
prob2 = F.softmax(score*alpha2,-1)

def softmax_func(score):
    return F.softmax(score)

jaco_mat1 =
torch.autograd.functional.jacobian(softmax_func,score*alpha1)
jaco_mat2 =
torch.autograd.functional.jacobian(softmax_func,score*alpha2)

print(jaco_mat1)
print(jaco_mat2)
```