自己实现一个Transformer

Dezeming Family

2024年10月7日

本文介绍如何根据Transformer的论文《Attention is all you need》，使用Pytorch从底层实现一个Transformer架构。本文代码见[7]的MyTransformer.zip文件。

目录

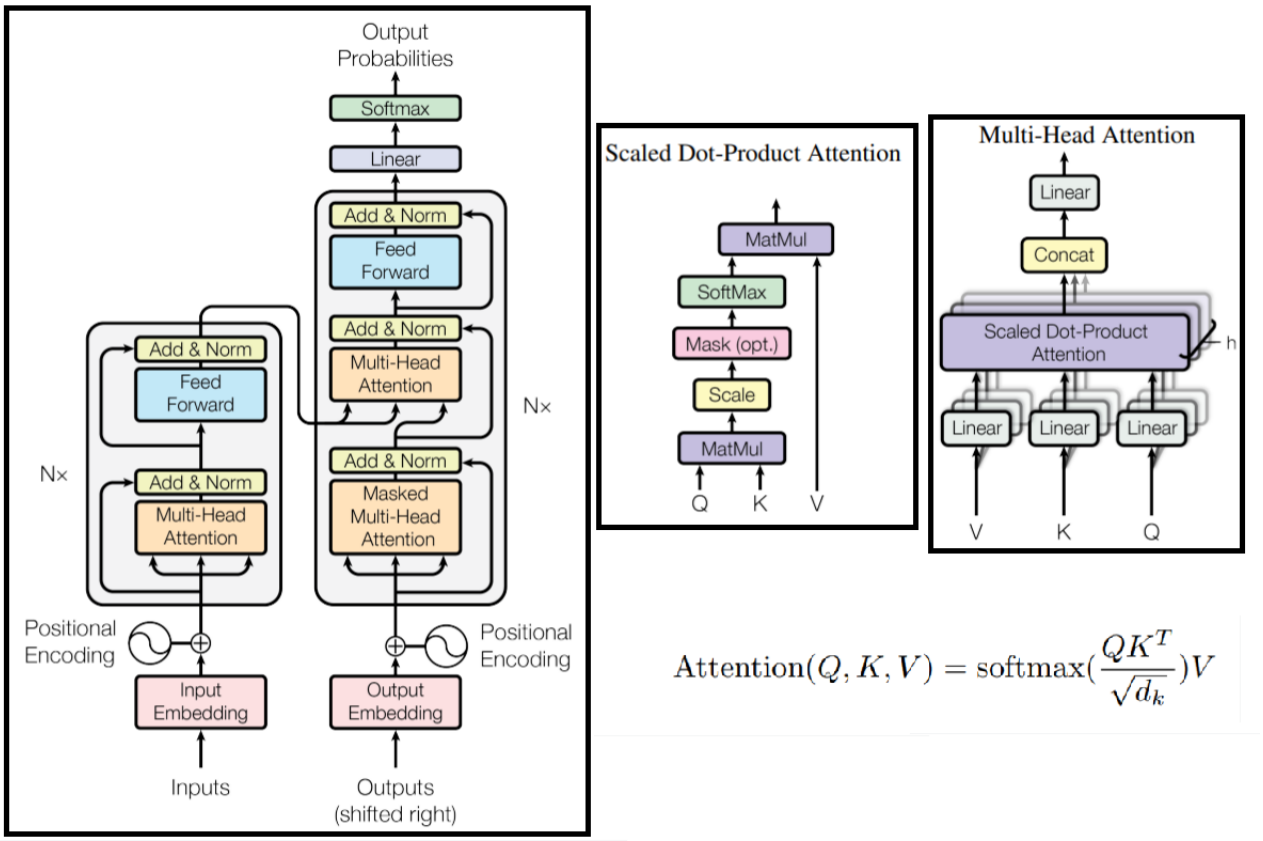
引言

尽管pytorch已经提供了很好用的transformer实现[1]，但对于NLP的初学者而言，自己实现一个Transformer仍是一件非常有意义的事情，尤其是能够真正熟悉其内部原理。Transformer的网上的教程以李沐和李宏毅老师的最为精彩，但缺少如何实现源码。有些教程的实现有不少错误，而且讲解不够细致。我们实现的版本会尽量与pytorch的官方实现功能一致。

我们虽然会反复对照论文原文[2]来说明，但在我们开始本文之前，要求读者已经学习了李宏毅老师的WordEmbedding讲解和Transformer讲解（但不要求完全掌握），然后还需要看完李沐老师的论文精度Transformer。这里省去了过多的背景介绍。

0 1　原论文的架构究竟在做什么

在实现之前有必要先介绍一下原论文[2]的这个架构究竟是在做什么。Transformer的整体图如下：



该图分为三个框，其中最左边的框是一个整体架构，可以看到其中有三个多头自注意力模块，其中有两个是没有Mask的，一个是有Mask的。

看到该架构可能不免有疑问，为什么输出部分也有一个输入？为什么输出部分的输入有一个Mask。这里注意原文的目标是使用Transformer做翻译，翻译任务的特点是输入一个序列，然后输出一个序列。注意输出的序列是一个一个输出的，并且已经输出的内容会再作为输入（auto-regressive）。这里会把机器翻译中已经输出的部分再返回到输出部分的输入里。

Mask表示需要遮掩一些东西，考虑其原文应用场景：机器翻译。假如输入：“Today is a good day”，输出：“今天是个好日子”。（1）那么一开始输入“Today is a good day”时，预测的第一个网络输出应该是“今天”，这个输出需要将Outputs的输入全部mask掉，因为第一个网络输出仅仅与这整句英文有关。（2）Inputs部分不变。仍是这句英文，但此时Outputs已经有一个之前的输出“今天”了，那么再输入时，就把Outputs的第一个Embedding不Mask，其他位全都Mask。此时用“Today is a good day”和“今天”来预测下一个词语“是”。（3）然后用“Today is a good day”和“今天”、“是”来预测“个”。依次类推。其实具体实现时，会使用一个下三角矩阵对Outputs的输入进行Mask。

输出时，在初始输出中添加一个起始符，相当于将输出整体右移一位（图中的Shifted Right）。

0 2　Utility功能

为了更容易讲解，我们的代码给出了一些实用功能，放在了Utility.py文件中。

add_to_class()函数是从[3]中获得的，可以给一个已经创建好的类增加新的函数功能。即使这个类已经创建了对象，在调用此方法后该对象也可以获得该函数功能。使用该函数是为了避免一次性粘贴一整个类，导致篇幅过长不利于讲解。

一　多头自注意力

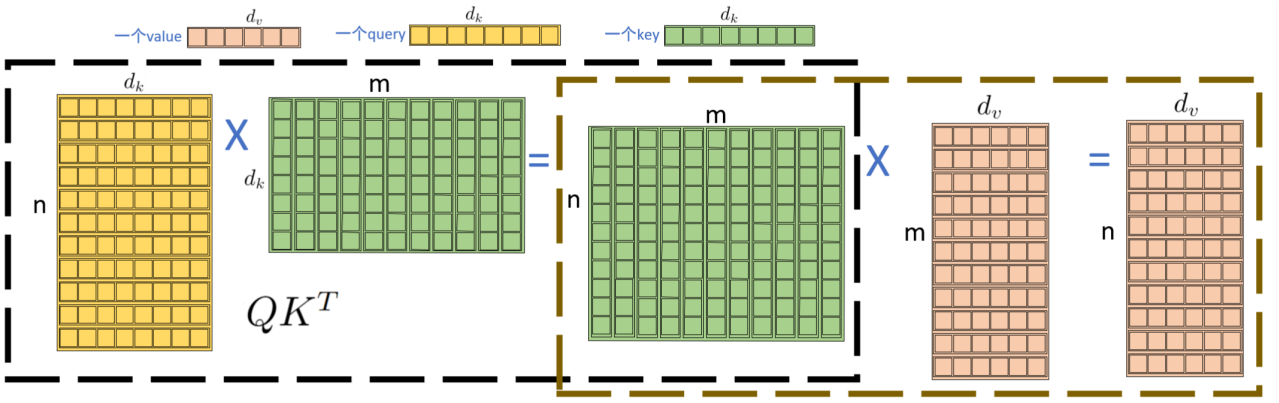
我们先讲解一个注意力模块，然后再过渡到多头注意力，最后描述自注意力。

1 1　注意力模块

当我们要输入一个序列时，输入维度通常是[batch size,squence length,vector dimension]。其中，squence表示一个序列的长度，例如“今天是个好日子”，转换为序列就包含[今天，是，个，好，日子]这五个元素，即squence length为5（后面squence length记作m）。每个序列中的每个元素都是一个词向量，通常是使用Word Embedding技术生成的，每个词向量的维度就是vector dimension。对于一次输入多个序列，即batch size大于1时，pytorch可以通过广播机制实现计算，我们可以不用考虑具体处理过程。

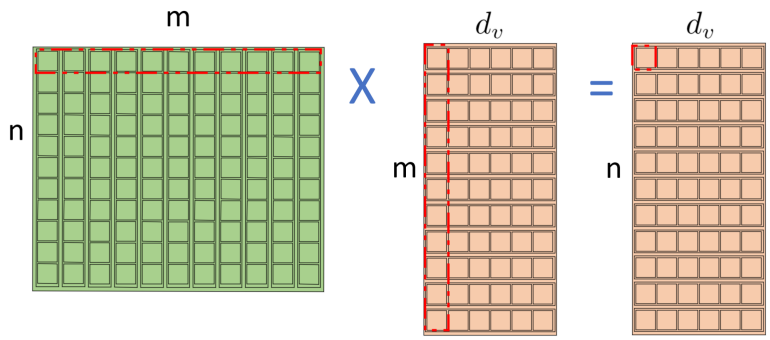
在一个注意力模块里，对一个value（一个词向量，维度为d_v）作用的权重是通过它的query和key计算得到的。对于一个序列，其中包含了m个词向量，注意力要求每个词向量的query都与其他词向量的key做点积相似度（余弦相似性，点积值越高，表明对其关注度会越高）。点积相似度越高，比如第3个词向量的query乘以第5个词向量的key越高，则表示这两个词之间相关性越高。

每个query和key的维度都是d_k。注意由于输入是一个包含m个词向量的序列，因此可以把keys打包为一个m行d_k列的矩阵，即论文里的K。query部分同理，但querys可以不为m个，假设有n个querys，打包成一个n行d_k列的矩阵，Q。因此向量key和query的点积就可以写为矩阵相乘的形式：QK^T，得到一个n行m列的矩阵，此时再与values构成的矩阵相乘：



图中的一个value就是一个词向量（维度为d_v）。如果一个Value序列长度是m，那么这个词向量矩阵就是m行d_v列的矩阵。softmax函数是对QK^T得到的矩阵（$n \times m$）的每一行做的softmax，做softmax和除以$\sqrt{d_k}$并不会改变其维度。如果Q的长度也是m（即$m = n$），则一个注意力模块的输出维度等于其输入的Value的维度。其实后面会讲到的自注意力的Q就是Value（只不过多头注意力会做一个线性变换，但维度是不会变的）。

那么这个相乘的过程怎么跟所谓的“注意力”有关呢？我们可以认为一个词向量的不同维度表示不同的信息，而这个$n \times m$的矩阵因为是对每一行做的software，可以表示一个查询query对于这个序列的每个维度的关注度：



上图中的$n \times m$的矩阵的第一行，表示对于整个序列的词向量的第一个维度的关注度概率分布。得到的结果放到输出的矩阵的最左上角。

1 2　多头注意力

了解完单个注意力模块以后，我们看一下多头注意力模块。

假如我们使用一大堆query（n很大），好像是完全可以的。但论文[2]中表示，这个网络可以学的参数太少了（尽管query和key的参数都能学，但还是很少），因为其中很多过程都是线性矩阵乘法和softmax，没有任何可以学的参数。于是该论文做了这么一种操作：先将原来的数据进行线性映射，映射为多个低维度结果，然后对低维度结果分别作注意力，之后将输出拼接成原来的维度：

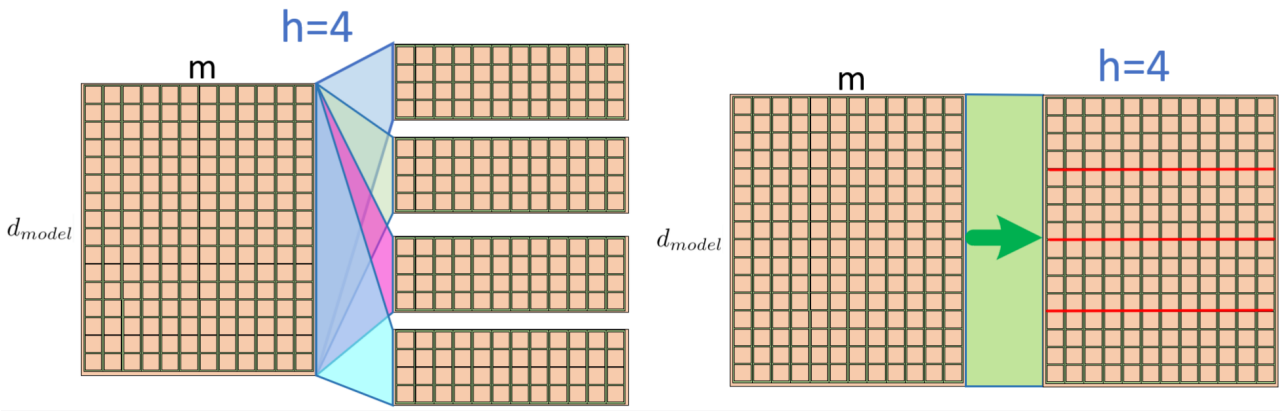
$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1,...,\text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_\text{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_\text{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_\text{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_\text{model}}$.

注意论文里对d_k和d_v进行了限制，使其相等。也对模型的输出进行了限制，输出$d_{model} = 512$。论文说Embedding层也是$d_{model} = 512$，这其实对应了单头注意力机制的d_v。此时我们的多头注意力机制里，d_v不再是一个词向量的维度，而是词向量映射后的维度，即如果要分别映射到八个不同的头（$h = 8$），那么$d_v = d_{model}/h = 64$。论文中设置$d_k = d_v = d_{model}/h = 64$，$h = 8$。

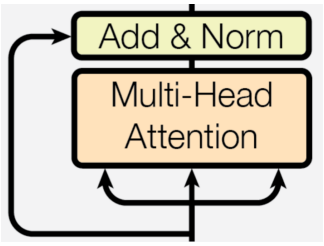
也就是说，对于m长的序列，每个词向量是d_{model}维度，构成$m \times d_{model}$的矩阵。这个矩阵经过VW_i^V变换，得到了$m \times d_v$的矩阵。然后将八个注意力头的输出拼在一起，又得到了$m \times d_{model}$的输出。因此，一个数据通过多头注意力模块后，其维度仍然不变。

在实际实现时，其实就是先对输入整体乘以一个$d_{model} \times d_{model}$的矩阵。下图中左边表示分别映射到四个低维度空间（$h = 4$），右边表示先映射到一个同等维度的空间，然后划分到四个区域。这两种方法得到的结果是完全一样的。实现时按照更容易实现的右边的方法进行。

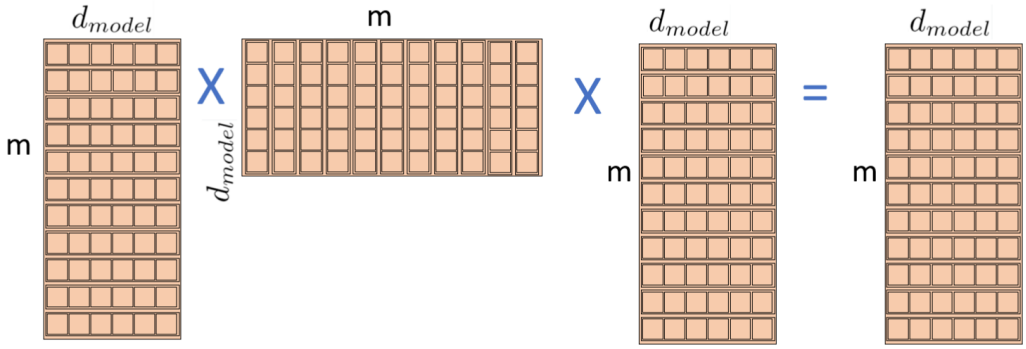


1 3　自注意力

自注意力这个概念非常简单，就是说K和Q和V都是同一个东西。换句话说，其实K和Q在一个模块里是固定的了（没法训练不同的K和Q）。在图中其实可以看到就是一个输入复制为了三份：

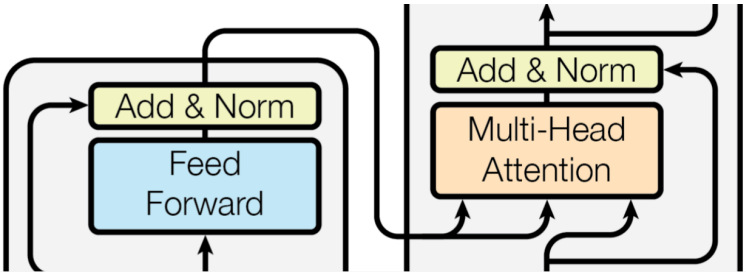


因为K和Q都是自己本身，所以被称为自注意力。先不考虑多头的情况，只考虑单头，那么现在其实可以看出，输出就是输入的加权和（矩阵A乘以矩阵B，就相当于把矩阵B的每行作为一个向量，互相线性组合得到新输出。这里作用的矩阵的每行就是一个词向量，因此相当于对这个序列的词向量进行线性组合）：



也就是说，如果是自注意力机制，即Q和K都是V，那么一个注意力模块里确实没有可以用于训练的参数了。所以此时设置多头自注意力就是必要的选择，即，使用多个线性层来映射到不同的低维空间。

在输出部分，除了Masked多头自注意力，还有非自注意力模块：



它输入的K和V来自于编码器，输入的Q来自于解码器。

二　多头自注意力的实现

我们的实现参考自[5]。

初始化

初始化只需要定义一些基本量，比如d_{model}和h（num_heads）。然后需要三个分别作用于Key、Value和Query的$d_{model} \times d_{model}$的矩阵。经过注意力作用后，还需要经过一层Linear层。代码为：

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model=256, num_heads=4):
        """
        d_model
        num_heads
        """
        super(MultiHeadAttention, self).__init__()
        self.d_model = d_model
        self.num_heads = num_heads
```

```
        assert d_model % num_heads == 0, "d_model must be divided by h"
        self.Wv = nn.Linear(d_model, d_model, bias=False)  # for Value
        self.Wk = nn.Linear(d_model, d_model, bias=False)  # for Key
        self.Wq = nn.Linear(d_model, d_model, bias=False)  # for Query
        self.Wo = nn.Linear(d_model, d_model, bias=False)  # for Linear
```

前向过程

前向过程为forward函数：

```
def forward(self, q, k, v, attention_mask=None, key_padding_mask=None)
```

具体过程为：(1) 对K、Q和V施加线性变换。(2) 进行维度变换，使K、Q和V被划分为多头输入。(3) 输入到注意力模块。(4) 输出结果contact。(5) 输出结果乘以线性层。

其中(1)和(5)就是简单的线性乘法。我们本小节关注(2)和(4)。

功能(2)即函数split_into_heads，是使用view函数实现的：

```
@Utility.add_to_class(MultiHeadAttention)
def split_into_heads(self, x, num_heads):
    batch_size, seq_length, d_model = x.size()
    x = x.view(batch_size, seq_length, num_heads, d_model // num_heads)
    return x.transpose(1, 2)
```

抛去batch_size这个维度不谈，对于Value，转换成多头以后其他三个维度就是$[h, m, d_k]$。注意在多头注意力中，$d_k = d_{model}/h$。

将这种形状的数据输入到自注意力模块以后，得到的形状仍然是$[h, m, d_{model}]$，此时再变形回之前的$[m, d_{model}]$（回顾前面的描述，多头注意力模块输出的m其实是Q的序列长度）：

```
@Utility.add_to_class(MultiHeadAttention)
def combine_heads(self, x):
    batch_size, num_heads, seq_length, head_hidden_dim = x.size()
    return x.transpose(1, 2).contiguous().view(
        batch_size, seq_length, num_heads * head_hidden_dim)
```

点乘注意力

不考虑Mask的部分是非常简单的：

```
@Utility.add_to_class(MultiHeadAttention)
def scaled_dot_product_attention(
    self, query, key, value, attention_mask=None, key_padding_mask=None):
    d_k = query.size(-1)
    tgt_len, src_len = query.size(-2), key.size(-2)

    logits = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    attention = torch.softmax(logits, dim=-1)

    output = torch.matmul(attention, value)

    # output shape: (batch_size, num_heads, sequence_length, d_model)
```

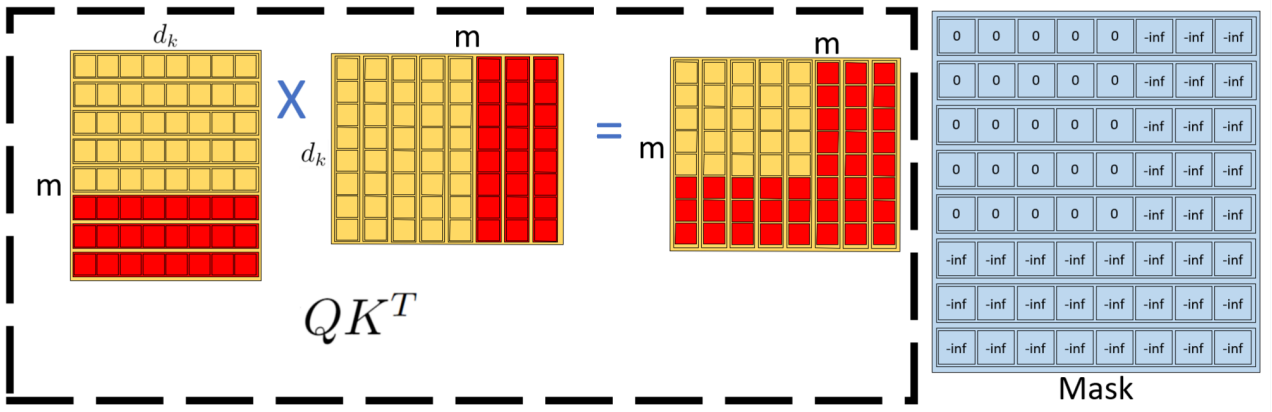
```
13      return output, attention
```

但很多人会对这里的Mask产生很多误解。为了更容易去讲解，我们需要明确实际的Transformer的一些参数。实际实现时，Query的序列长度和Value的序列长度需要保持一致，而且编码器和解码器的Query、Key和Value的序列长度是一样的。也就是说，之前的m是一定等于n的，这样才能保证计算得到的注意力矩阵（QK^T）是一个方阵。

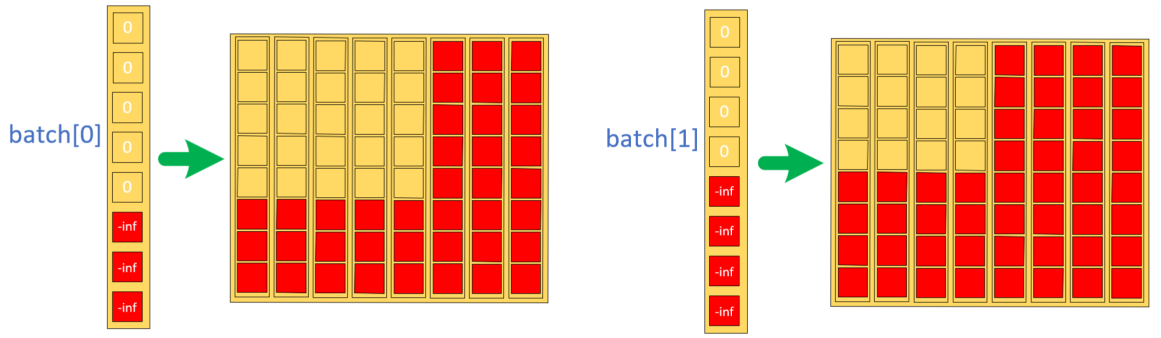
Padding部分掩码的实现

Mask有两种，一种是对注意力的掩码，另一种是对Padding的掩码。对注意力的掩码前面讲过，主要功能是为了防止未来的输出影响现在的结果。比如对于机器翻译，我们需要让解码器仅看到已经输出的部分。对Padding的掩码需要这么理解：我们一个序列有短有长，但是序列长度应该与最长的可能输入一致，所以会有一些空余位。例如“今天、是、个、好、日子”，序列长是5，但“鸡、你、太、美”，序列长是4，此时就需要在运算中屏蔽掉后面的部分。这个屏蔽过程要放在计算注意力矩阵相乘之后、计算softmax之前。其实就是给对应的位增加一个绝对值非常大的负数，使得其通过softmax之后的值无限接近于0。

先看padding部分如何进行Mask。假设设定的序列长度为$m = 8$，如下图所示，但输入序列长度为5，标红的部分表示需要padding的部分。然后QK矩阵相乘得到的需要padding的部分和Mask的对应关系也显示在了图上。注意由于Mask直接加到原来的QK^T矩阵上，然后再对矩阵的每一行求softmax，这就起到了屏蔽的作用，因为加上一个无穷小的值然后再做softmax以后约等于0，表示该权重不再起作用。



实现时，输入的key_padding_mask的维度可以是[batch_size,m]或者[batch_size,m,m]。如果是[batch_size,m]，则最后一个维度表示要mask的位。需要先使用unsqueeze方法扩展为[batch_size,1,m,m]，然后将batch里的每个mask转换为上面所显示的样子。即：



自注意力部分掩码的实现

再看自注意力部分如何计算Mask。注意，本来Decoder一次只输出对下一个词的预测，比如机器翻译问题，输入是“Today is a good day”，当编码器输入开始标志时（前面讲过，Shifted right就是移位让第一个位是开始标志），输出“今天”。然后把输入喂给编码器，开始标志和“今天”喂给解码器时，输出“是”。这整个过程按理来说应该要靠循环来实现。但这里大家已经知道，多头注意力模块的输出维度和输入维度是一致的，都是可能输入的序列的最长长度（这里表示为m），因此，就需要在自注意力里进行

一些操作，使得当要输出“是”时，解码器的输入中的“是、一个、好、天”不会再对其造成影响。为了讲解更容易，我们忽略开始标志。

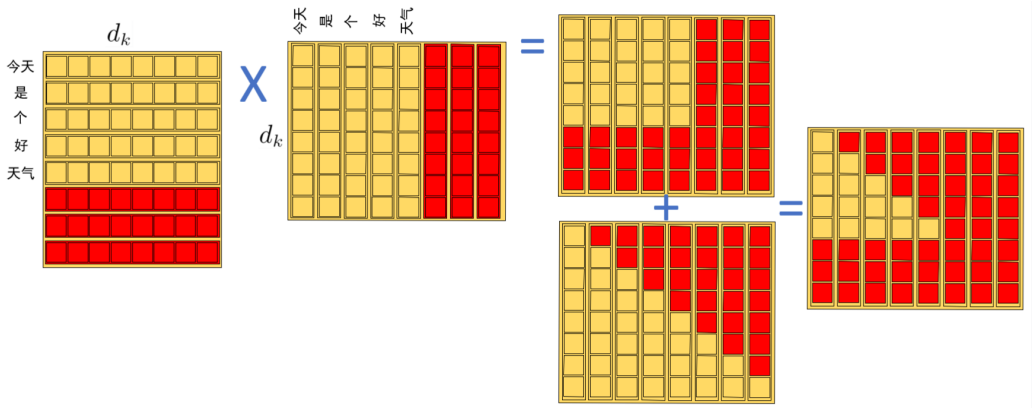
我们先介绍如何生成Mask矩阵，然后再说明其能够遮挡未来信息的原因。下面的代码根据输入的size生成一个上三角矩阵，然后用1减去该上三角矩阵（相当于进行0-1反转）。之后将为1的元素填充0，将为0的元素填充负无穷大：

```
def generate_square_subsequent_mask(size: int):
    mask = (1 - torch.triu(torch.ones(size, size), diagonal=1)).bool()
    mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(
        mask == 1, float(0.0))
    return mask
```

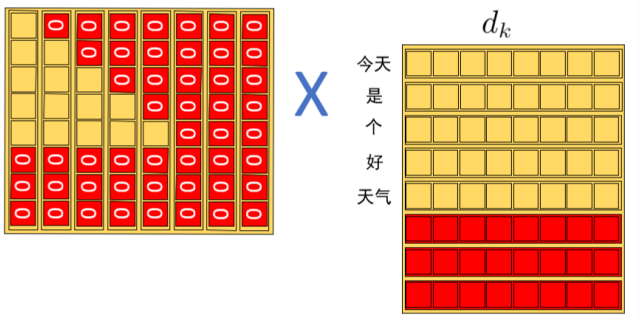
得到的结果为如下矩阵（输入size=8），这个矩阵也会被加到$\frac{QK^T}{\sqrt{d_k}}$上。

0	-inf	-inf	-inf	-inf	-inf	-inf	-inf
0	0	-inf	-inf	-inf	-inf	-inf	-inf
0	0	0	-inf	-inf	-inf	-inf	-inf
0	0	0	0	-inf	-inf	-inf	-inf
0	0	0	0	0	-inf	-inf	-inf
0	0	0	0	0	0	-inf	-inf
0	0	0	0	0	0	0	-inf
0	0	0	0	0	0	0	0

注意这个Mask的宽和高维度都是序列长度m，对于多头注意力，其降低的维度并不是序列这个维度，而是词向量的维度，因此Mask其实是在序列维度上进行的屏蔽。把Attention的Mask和Padding的Mask一起加进去，图示为：



然后将相加后的结果除以$\sqrt{d_k}$，之后再做softmax，此时的矩阵为0的部分如下图所示：



这个注意力矩阵的第一行表示“今天”的自注意力，为1。第二行表示“今天”和“是”之间的自注意力。可以看到，Masked多头注意力块的输出的$m \times d_{model}$的矩阵，在序列维度上每下一列的结果都比之前多了一个新的词向量的词向量间的互注意力。比如，第一行就是“今天”的编码；第二行就是“今天”和“是”的编码注意力组合；第三行就是“今天”和“是”和“一”的编码注意力组合。

三　位置编码和整个编解码架构的实现

位置编码部分

位置编码比较简单，不再赘述。

Piecewise Feed Forward的实现

该层就是线性层加一个ReLU激活，实现在了PositionWiseFeedForward类中。

一个注意力Block的实现

EncoderBlock类将多头注意力层打包，然后加上残差连接，构成一个Block块。

整个编码器的实现

整个编码器就是将位置编码、N个注意力Block和Piecewise Feed Forward打包在一起。过程较为简单，参考代码即可。

参考文献

[1] https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html

[2] https://arxiv.org/abs/1706.03762

[3] https://d2l.ai/index.html

[4] https://jalammar.github.io/illustrated-transformer/

[5] https://zhuanlan.zhihu.com/p/690355021

[6] https://blog.csdn.net/m0_64148253/

[7] https://github.com/feimos32/Deep-Learning—Pytorch-Implementation