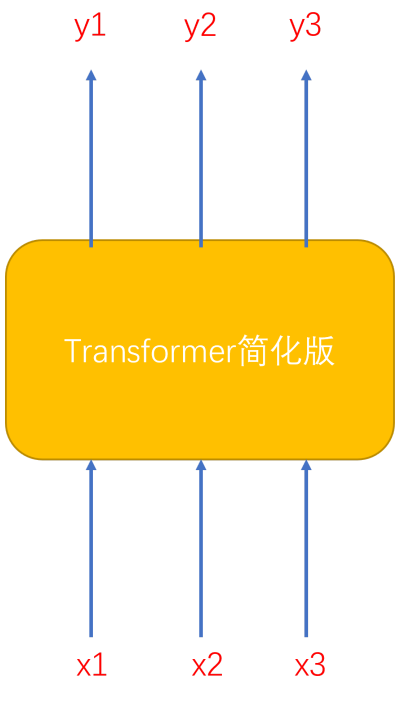
一个简单例子说明"设计位置编码"这件事的动机

Transformer简化计算假设：

由于Transformer内部计算较为复杂，这里把它简化为下面这种计算：
（在说明位置编码如何起作用这件事上，这个简化版和原版Transformer效果完全一样）



$$y_1 = x_1 * x_1 + x_1 * x_2 + x_1 * x_3 = x_1 * (x_1 + x_2 + x_3)$$

$$y_2 = x_2 * x_1 + x_2 * x_2 + x_2 * x_3 = x_2 * (x_1 + x_2 + x_3)$$

$$y_3 = x_3 * x_1 + x_3 * x_2 + x_3 * x_3 = x_3 * (x_1 + x_2 + x_3)$$

输入3个词，吐出3个词，吐出的y1，y2，y3分别是这样计算的：
y1=第1个词x1分别与每个词相乘，然后相加
y2=第2个词x2分别与每个词相乘，然后相加
y3=第3个词x3分别与每个词相乘，然后相加

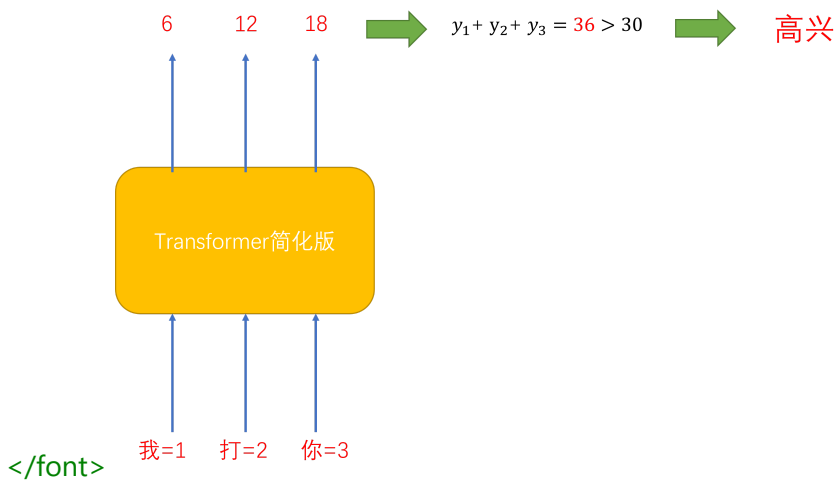
用上面简化版Transformer做情感分析的话，会出现什么问题？

汉字	汉字的编码
我	1
打	2
你	3

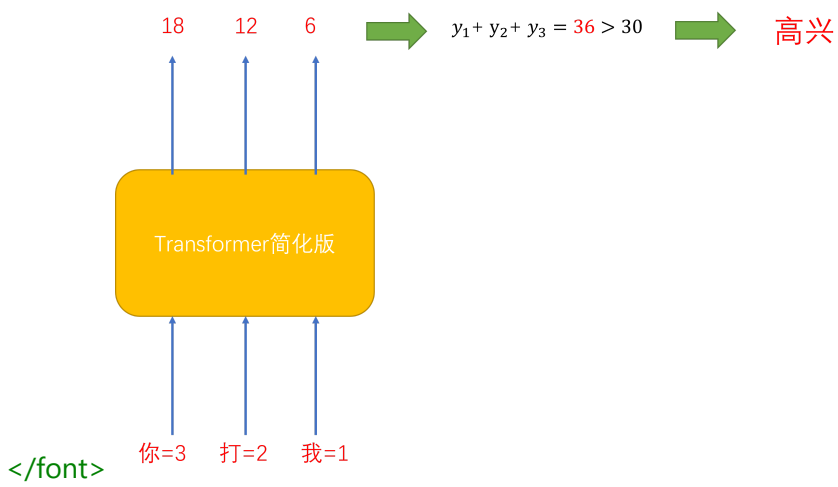
样本	标签（我的情感）
我打你=[1,2,3]	高兴
你打我=[3,2,1]	不高兴

直接把上面的汉字编码丢到Transformer当中计算出y1，y2，y3，
然后约定当y1+y2+y3超过30的时候，我的情感是高兴的，否则就是不高兴的

先把"我打你"丢进去，看看预测的情感是什么



再把"你打我"丢进去，看看预测的情感是什么



问题分析：

由上图可以看出来，“我”这个单词，不管是打别人的那个“我”，还是被别人打的那个“我”，
不管位置怎么变，经过Transformer后出来的编码都是18，都是同一种表示，虽然现实生活中
“我”只需要一个代号就行，放在不同的句子里我们人都能正常理解，但是对于计算机来说用
同一个代号来表示“我”并不是一件好事，就好比上面，不管我打你，还是你打我，算出来的数完全一样，
预测的都是高兴，这就是Transformer的毛病所在(准确来说是自注意力的毛病)

这个毛病的学术描述：自注意力不能从句子中获取位置信息

解决方案：

既然单纯从上面的单词词典查出来的编码没有位置信息，我们直接做一个位置编码词典
然后我在上面原本的编码的基础上，加上一个位置编码，然后这个拿来作为我们的编码，拿这个丢进Transformer当中。
即便都是"我"，在计算机看来，只要处于不同位置，他们就不是同一个“我”。
引用赫拉克利特的名言：人不能两次踏入同一条河流。
意思就是，这个时间（位置）的我，和下个时间的我已经不是同一个我了。

汉字词典

汉字	汉字的编码
我	1
打	2
你	3

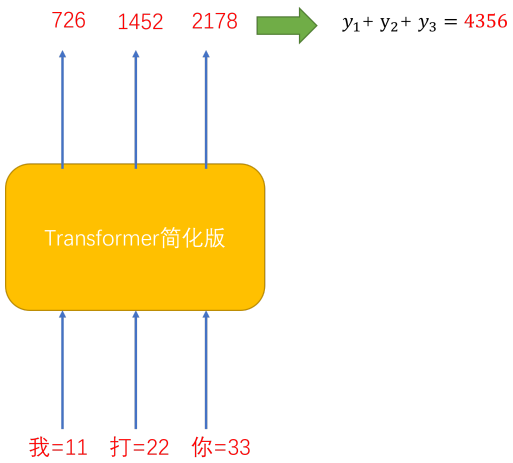
位置编码词典

绝对位置	位置的编码
1	10
2	20
3	30

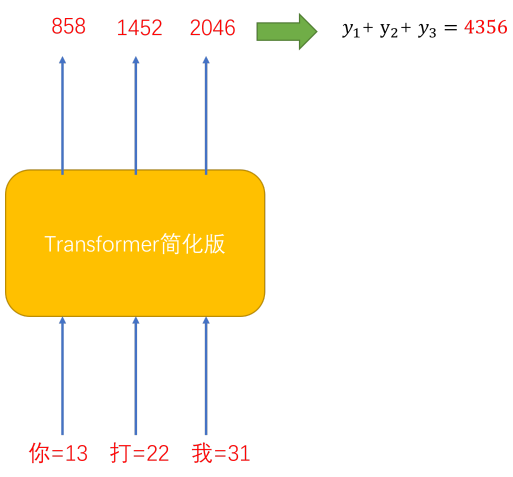
考虑了位置编码后的编码

样本	查词典得到的编码	位置编码	两个编码相加作为真正的编码
我打你	[1,2,3]	[10,20,30]	[11,22,33]
你打我	[3,2,1]	[10,20,30]	[13,22,31]

先把“我打你”丢进去，可以看出“我”变成了726



再把“你打我”丢进去，可以看出“我”变成了858



由上面计算结果我们可以看出，仅仅只是将第一个单词加上位置编码10，第二个单词加上位置编码20，第三个单词加上位置编码30，
就可以让不同位置的“我”经过Transformer后变得不一样，从而解决编码没有位置信息这个毛病。但是上述位置编码依旧存在一些毛病，
第一点：如果把输出相加后的结果看成句子的语义，上面想加后还是一样的，这点可以在后面加个分类头，通过按权重相加来解决；
第二点：位置编码的数字几乎是汉字编码的10倍，两者数量级不一样，导致输入编码几乎是纯位置信息，原始汉字信息被盖住了，实际上
这样通过Transformer训练，学习的是基于位置信息作出的预测，跟你的汉字几乎没多大关系，所以需要对汉字编码或者位置编码进行适当的缩放；
（做过实验的可以检查这部分代码，是不是莫名其妙的给编码除以根号D之类的，这个不是为了缩小方差加速模型收敛而设计的，而是上面讲的这个原因）

其他可考虑的位置编码

1.加个RNN或者LSTM,通过RNN后出来的编码是具备位置信息的（简单来说就是同一个字，在句子的不同位置，通过RNN后出来的编码不一样了）
缺点：需要大量的语料来学习位置信息，比如有"我打你"这句话，还需要有"你打我"这样位置不太一样的句子让模型学习调整参数。
2.假如汉字是二维编码，那么用二维编码当作位置编码的很好的一个选择就是，平面上画个单位元，想象一个点在圆周上运动，然后不同时刻这个点的位置的二维向量（sinwt，coswt）就能拿来当位置编码，t就是运动的时刻，对应句子中单词位于第几个位置，w是运动频率，可以随便调整，如果汉字编码是100维，这100维的运动信息就需要调整w，使得不同维度上的运动频率不一样，这就是三角位置编码
3.相对位置编码，没仔细看计算公式，就不提了。

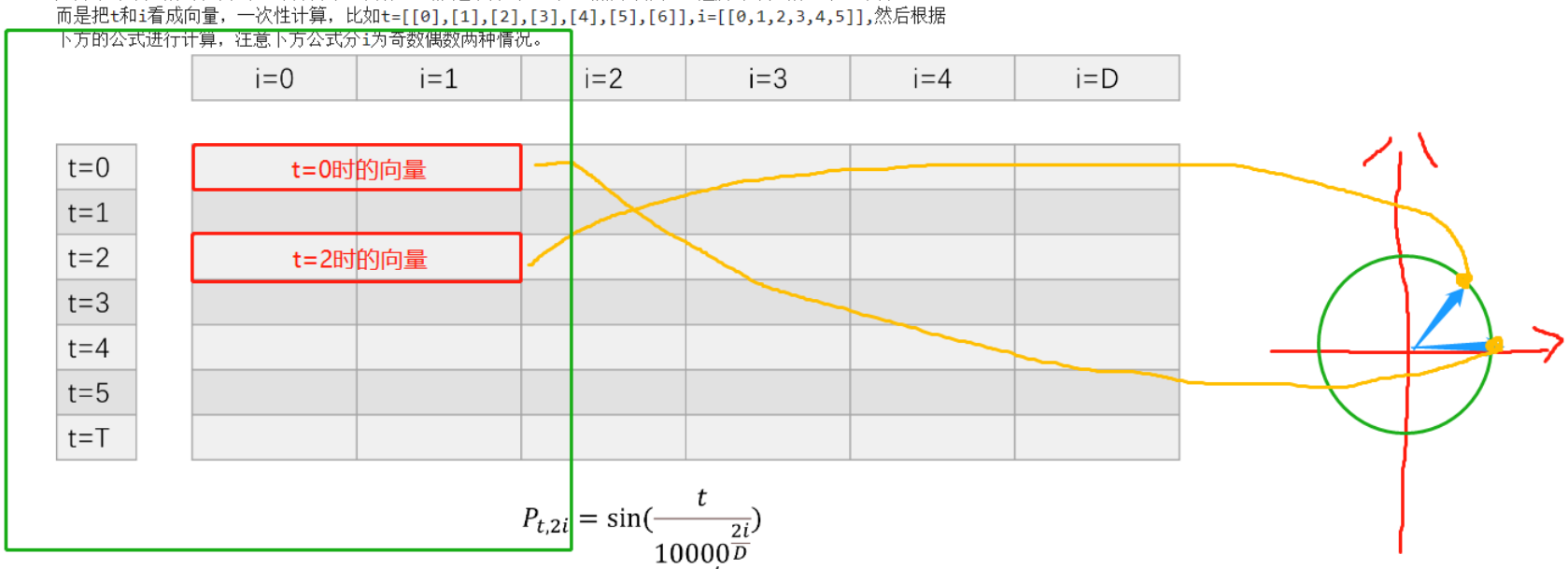
各种位置编码的优越性如何?

目前没看过基于位置编码的相关研究，不清楚基于不同位置编码能给实验结果带来什么不同，但个人感觉，可能差不多，基本带来不了多少提升，如果有什么明显提升，大概率是模型的其他部分在起作用。

附录：三角位置编码

位置编码

下图是一个位置编码矩阵表，t=0所在行表示第0个位置的单词的编码，维度为D，和单词维度一样。
矩阵中每个元素的值由t和i计算而来，具体公式就是下方的公式，当然不会用for遍历每个元素一个一个算，
而是把t和i看成向量，一次性计算，比如t=[[0],[1],[2],[3],[4],[5],[6]],i=[[0,1,2,3,4,5]],然后根据下方的公式进行计算，注意下方公式分i为奇数偶数两种情况。



$$P_{t,2i} = \sin(\frac{t}{10000^{\frac{2i}{D}}})$$

$$P_{t,2i+1} = \cos(\frac{t}{10000^{\frac{2i}{D}}})$$

我们只看前2维

In [ ]:
```
import torch.nn as nn
class PositionEmbedding(nn.Module):
    def __init__(self,max_length,embedding_dim):
        super(PositionEmbedding,self).__init__()
        t=torch.arange(max_length).unsqueeze(-1)
        i=torch.arange(embedding_dim).unsqueeze(0)
        self.W=torch.Tensor(size=[max_length,embedding_dim])  # 不需要学习，不用Parameter类
        # print(t.shape,i.shape,self.W.shape)

        i_even=i[0,0::2]
        self.W[:,0::2]=torch.sin(t/torch.pow(torch.tensor(10000.),2*i_even/embedding_dim))

        i_odd=i[0,1::2]
        self.W[:,1::2]=torch.cos(t/torch.pow(torch.tensor(10000.),2*i_odd/embedding_dim))

    def __call__(self,idx):
        return self.forward(idx)

    def forward(self,idx):
        return self.W[idx]
```