iynil
青梅有终 莫逆有别/

+ 关注

1452 人赞同了该回答

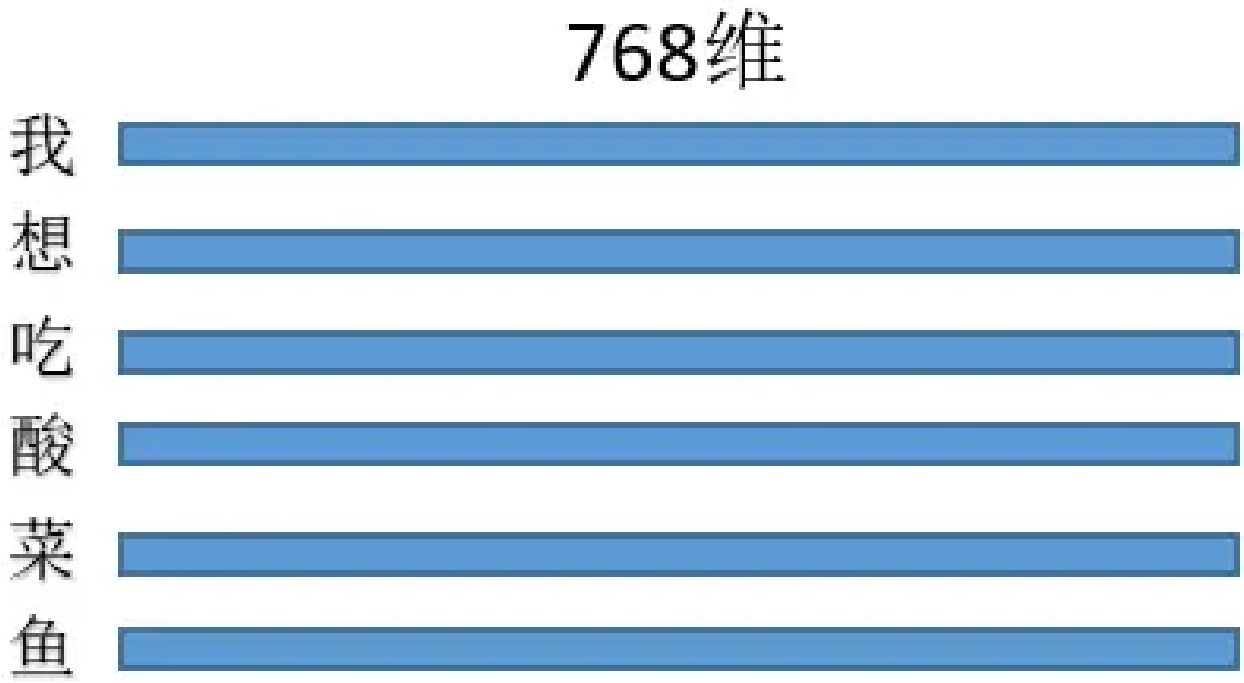
我们直接用torch**实现一个SelfAttention**来说一说：

首先定义三个线性变换矩阵，query, key, value：

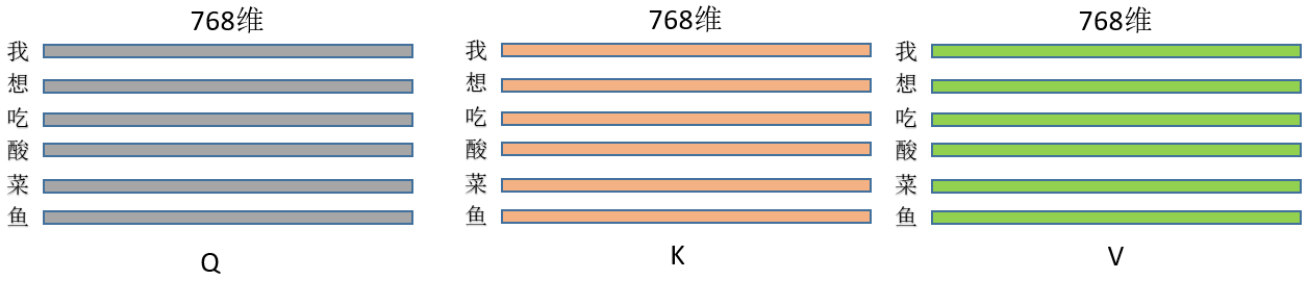
```
class BertSelfAttention(nn.Module):
    self.query = nn.Linear(config.hidden_size, self.all_head_size) # 输入768， 输出768
    self.key = nn.Linear(config.hidden_size, self.all_head_size) # 输入768， 输出768
    self.value = nn.Linear(config.hidden_size, self.all_head_size) # 输入768， 输出768
```

注意，这里的query, key, value只是一种操作(线性变换)的名称，实际的Q/K/V是它们三个的输出

2. 假设三种操作的输入都是同一个矩阵（暂且先别管为什么输入是同一个矩阵），这里暂且定为长度为L的句子，每个token的特征维度是768，那么输入就是（L, 768），每一行就是一个字，像这样：



乘以上面三种操作就得到了Q/K/V，(L, 768)*(768,768) = (L,768)，维度其实没变，即此刻的Q/K/V分别为：



代码为：

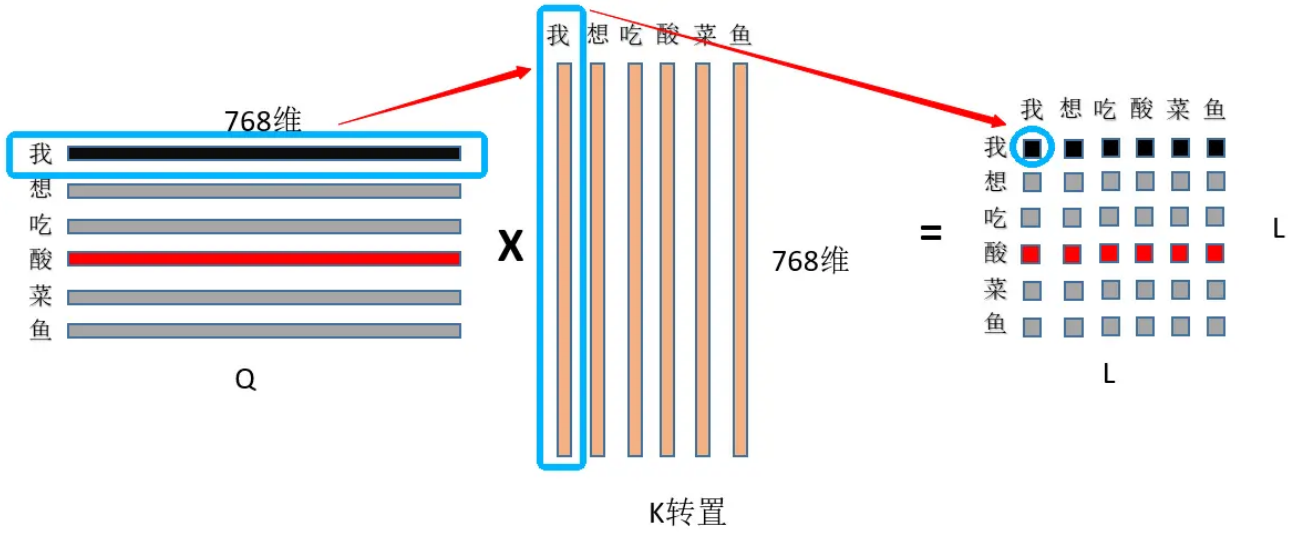
```
class BertSelfAttention(nn.Module):
    def __init__(self, config):
        self.query = nn.Linear(config.hidden_size, self.all_head_size) # 输入768， 输出768
        self.key = nn.Linear(config.hidden_size, self.all_head_size) # 输入768， 输出768
        self.value = nn.Linear(config.hidden_size, self.all_head_size) # 输入768， 输出768

    def forward(self,hidden_states): # hidden_states 维度是（L, 768）
        Q = self.query(hidden_states)
        K = self.key(hidden_states)
        V = self.value(hidden_states)
```

3. 然后来实现这个操作：

$$Attention(Q, K_i, V_i) = softmax(\frac{Q^T K_i}{\sqrt{d_k}})V_i$$

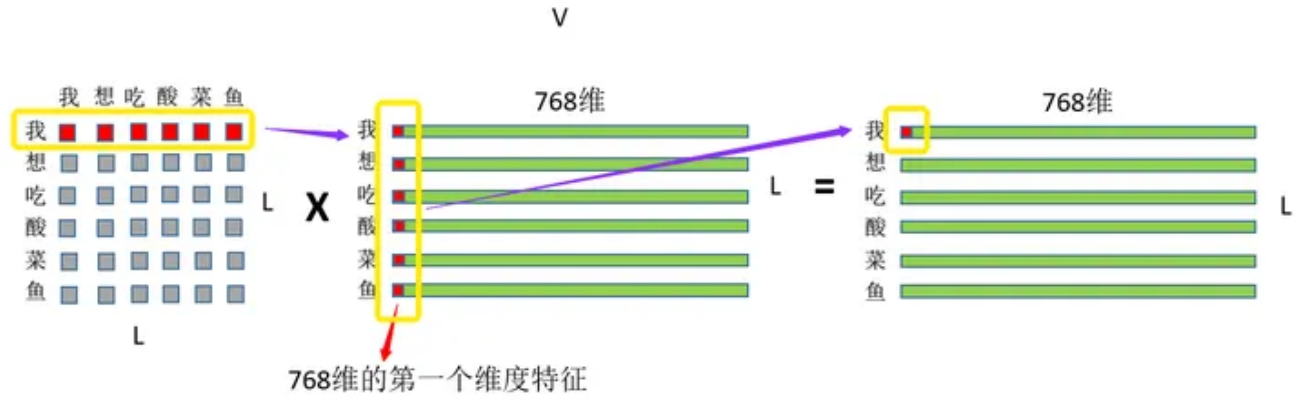
① 首先是Q和K矩阵乘，（L, 768）*(L, 768)的转置=（L,L），看图：



首先用Q的第一行，即“我”字的768特征和K中“我”字的768为特征点乘求和，得到输出（0，0）位置的数值，这个数值就代表了“我想吃酸菜鱼”中“我”字对“我”字的**注意力权重，**然后显而易见输出的第一行就是“我”字对“我想吃酸菜鱼”里面每个字的注意力权重；整个结果自然就是“我想吃酸菜鱼”里面每个字对其它字（包括自己）的注意力权重（就是一个数值）了~

② 然后是除以根号dim，这个dim就是768，至于为什么要除以这个数值？主要是为了缩小点积范围，确保softmax梯度稳定性，具体推导可以看这里：Self-attention中dot-product操作为什么要被缩放，然后就是为什么要softmax，一种解释是为了保证注意力权重的非负性，同时增加非线性，还有一些工作对去掉softmax进行了实验，如PaperWeekly：线性Attention的探索：Attention必须有个Softmax吗？

③ 然后就是刚才的**注意力权重**和**V**矩阵乘了，如图：



注意力权重 x VALUE矩阵 = 最终结果

首先是“我”这个字对“我想吃酸菜鱼”这句话里面每个字的注意力权重，和V中“我想吃酸菜鱼”里面每个字的第一维特征进行**相乘再求和，**这个过程其实就**相当于用每个字的权重对每个字的特征进行加权求和**，然后再用“我”这个字对对“我想吃酸菜鱼”这句话里面每个字的注意力权重和V中“我想吃酸菜鱼”里面每个字的第二维特征进行相乘再求和，依次类推~最终也就得到了（L,768）的结果矩阵，和输入保持一致~

整个过程在草稿纸上画一画简单的矩阵乘就出来了，一目了然~最后上代码：

```
class BertSelfAttention(nn.Module):
    def __init__(self, config):
        self.query = nn.Linear(config.hidden_size, self.all_head_size) # 输入768， 输出768
        self.key = nn.Linear(config.hidden_size, self.all_head_size) # 输入768， 输出768
        self.value = nn.Linear(config.hidden_size, self.all_head_size) # 输入768， 输出768

    def forward(self,hidden_states): # hidden_states 维度是（L，768）
        Q = self.query(hidden_states)
        K = self.key(hidden_states)
        V = self.value(hidden_states)

        attention_scores = torch.matmul(Q, K.transpose(-1, -2))
        attention_scores = attention_scores / math.sqrt(self.attention_head_size)
        attention_probs = nn.Softmax(dim=-1)(attention_scores)

        out = torch.matmul(attention_probs, V)
        return out
```

4. 为什么叫**自**注意力网络？因为可以看到Q/K/V都是通过同一句话的输入算出来的，按照上面的流程也就是一句话内每个字对其它字（包括自己）的权重分配；那如果不是自注意力呢？简单来说， 来自于句A，来自于句B即可~

5. 注意，K/V中，如果同时替换任意两个字的位置，对最终的结果是不会有影响的，至于为什么，可以自己在草稿纸上画一画矩阵乘；也就是说注意力机制是没有位置信息的，不像CNN/RNN/LSTM；这也是为什么要引入位置embeding的原因。