机器翻译作业

作者:尹张森,2101839
本项目使用pytorch框架实现transformer[1]

结果展示

```
1
```

1.数据集和数据处理

使用预处理好的IWSLT'14 De-En数据集,源语言德语目标语言英语,使用双字节编码分词.

实现一个自己的DataSet,DataLoader就行了

2.Transformer

根据教材2的介绍,循环神经网络每个循环单元都有向前依赖性,也就是当前时间步的处理依赖前一时间步处理的结果。这个性质可以使序列的"历史"信息不断被传递,但是也造成模型运行效率的下降。特别是对于自然语言处理任务, 序列往往较长,无论是传统的 RNN 结构,还是更为复杂的 LSTM 结构,都需要很多次循环单元的处理才能够捕捉到单词之间的长距离依赖。由于需要多个循环单元的处理,距离较远的两个单词之间的信息传递变得很复杂.

Transformer 模型仅仅使用自注意力机制和标准的前馈神经网络,完全不依赖任何循环单元或者卷积操作。自注意力机制的优点在于可以直接对序列中任意两个单元之间的关系进行建模,这使得长距离依赖等问题可以更好地被求解。

下图展示了 Transformer 的结构。编码器由若干层组成(绿色虚线框就代表一层) 。每一层(Layer)的输入都是一个向量序列,输出是同样大小的向量序列,而Transformer 层的作用是对输入进行进一步的抽象,得到新的表示结果。不过这里的层并不是指单一的神经网络结构,它里面由若干不同的模块组成.

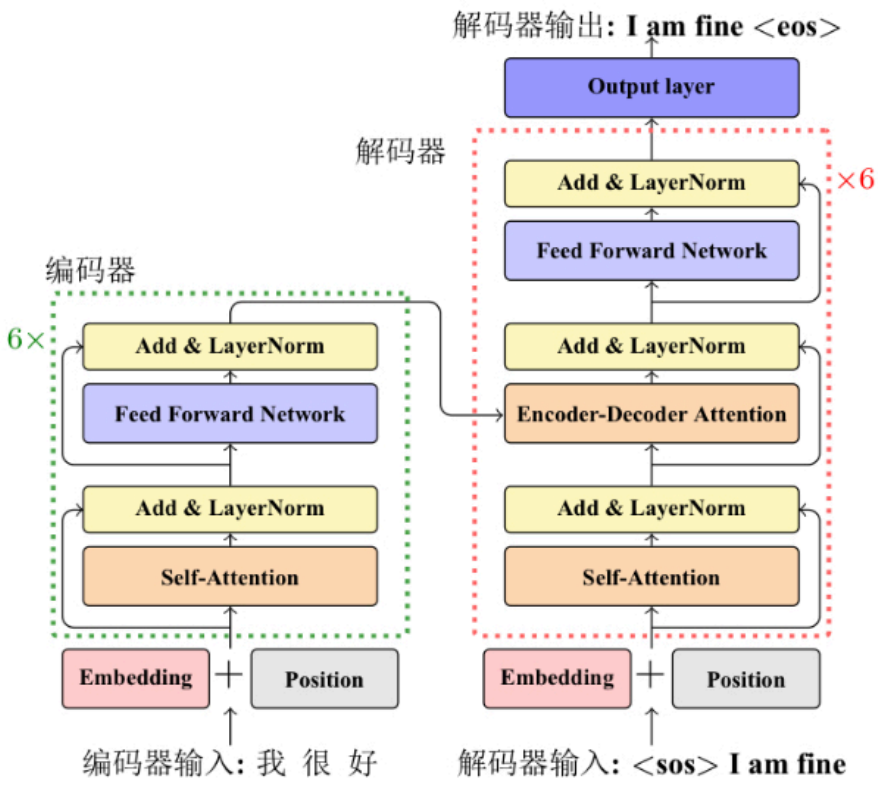

图 12.4 Transformer 结构

主要包括下列层:

- 嵌入层和位置编码
- 自注意力表示层(多头注意力机制)
- 前馈神经网络
- 残差连接和层标准化

嵌入层和位置编码层

嵌入层和位置编码层将编码器输入和解码器输入序列变成向量表示.

嵌入层

使用pytorch的Embedding()函数实现嵌入

位置编码

位置编码是一个固定的矩阵大小是(序列长度seq_len,嵌入向量长度emb_size)
Transformer中使用的是不同频率的三角函数
$PE(pos, 2i) = sin(\frac{pos}{10000^{2i/emb\_size}})$
$PE(pos, 2i+1) = cos(\frac{pos}{10000^{2i/emb\_size}})$

很多地方的代码在增加位置编码时给嵌入向量乘了sqrt(d_model)

```
class EmdAndPos(nn.Module):
    '''
    处理好的句子序列,并给他加上position编码
    参数(emb_size=d_model, seq_len, dict_number, padding_idx)
    输入:(batch_size, seq_len)
    输出:(batch_size, seq_len, emb_size)
    test:
    input1 = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
    net = nn.Sequential(EmdAndPos(16,4,10))
    print(net(input1))
    '''
    def __init__(self,emb_size,seq_len,dict_number):
        super(EmdAndPos, self).__init__()
        self.emb = nn.Embedding(num_embeddings=dict_number, embedding_dim=emb_size, padding_idx=0)
        self.pos = self._position(emb_size,seq_len)

    def _position(self,emb_size,seq_len):
        '''
        pos只与位置有关,没有学习过程,句子中的每一个单词产生一个描述位置的与词嵌入等长的向量,整个句子产生一个(seq_len,emb_size)的矩阵
        输入:处理好的句子序列
        输出:输出(seq_len , emb_size)的矩阵
```

```
        '''
        PE = np.zeros((seq_len,emb_size))
        def func(pos,i,emb_size):
            if i%2 ==0:
                return math.sin(pos / (10000**(1.0*i/emb_size)))
            else:
                return math.cos(pos / (10000**(1.0*(i-1)/emb_size)))
        for xy,val in np.ndenumerate(PE):
            PE[xy]= func(xy[0], xy[1], emb_size)
        pos_matrix = torch.from_numpy(PE)
        return pos_matrix

    def forward(self, inputs):
        X = self.emb(inputs)
        # print(self.pos)
        # print("*"*80)
        # print(X)
        return X+self.pos
```

基于点乘的多头注意力机制

多头注意力机制就是在原来点乘注意力机制的基础上,把原来d_model长度的向量切分成heads份,运算后在连接起来.它的好处是允许模型在不同的表示子空间里学习.在很多实验里发现,不同的表示空间的头捕获的信息是不同的.
对于上一层输入的X(batch_size, seq_len, d_model),先使用线性变换(没有激活函数)分别映射成QKV(batch_size, seq_len, d_model).需要注意在解码器中QKV的来源不用,Q来源于源语言,KV是目标语言的仿射变换.
之后对Q,K,V进行切分,切分的参数矩阵维度分别是(heads, d_model, d_k),(heads, d_model, d_k)(heads, d_model, d_v);(d_k=d_v=d_model/heads),

```
self.W_Q = nn.Parameter(data=torch.tensor(heads, d_model,
d_k//heads),requires_grad=True)
self.W_K = nn.Parameter(data=torch.tensor(heads, d_model,
d_k//heads),requires_grad=True)
self.W_V = nn.Parameter(data=torch.tensor(heads, d_model,
d_v//heads),requires_grad=True)
self.register_parameter('multihead_proj_weight', None)
```

这样切分后的qkv向量进行运算后在连接起来可以获得一个(1,heads*d_k)的向量并且(heads*d_k=d_model),对输出向量右乘一个输出矩阵W(d_model,d_model)获得最终多头注意力机制的score.
但这样写的权重好像不会参与训练,所以我按照d2l里的多头网络写法改写了多头注意力的分片方法

新的方法实现方法是:将变换后的QKV使用一个变换函数分片(而不是用不同的权重去成),然后直接应用点积注意力机制计算,最后再逆变换回原来的shape.通过一个线性层输出
最终输出的矩阵O大小应为(batch_size, seq_len, d_model),并且为了方便显示注意力机制也可以输出一个注意力权重矩阵.

Mask操作

mask分为两部分

- 句长掩码padding mask

  在批量处理多个样本时(训练或解码) ,由于要对源语言和目标语言的输入进行批次化处理,而每个批次内序列的长度不一样,为了方便对批次内序列进行矩阵表示,需要进行对齐操作,即在较短的序列后面填充 0 来占位(padding 操作) 。而这些填充 0 的位置没有实际意义,不参与注意力机制的计算,因此,需要进行掩码操作,屏蔽其影响.

- 未来信息掩码 future mask,

  对于解码器来说,由于在预测的时候是自左向右进行的, 即第 t 时刻解码器的输出只能依赖于 t 时刻之前的输出。 且为了保证训练解码一致,避免在训练过程中观测到目标语言端每个位置未来的信息,因此需要对未来信息进行屏蔽。具体的做法是:构造一个上三角值全为-inf 的 Mask矩阵, 也就是说, 在解码器计算中, 在当前位置, 通过未来信息掩码把序列之后的信息屏蔽掉了,避免了 t 时刻之后的位置对当前的计算产生影响.

其中,future mask 只对解码器起作用,padding mask在embedding层前生成一个句子表mask,然后作为参数给多头注意力机制

```
# padding
def _make_padding_mask(self, seq, seq_len, pad=0):
        '''把idx的做成padding_mask'''
        mask = (seq==pad)
        mask.bool()
        mask = mask.unsqueeze(1).expand((-1, seq_len, -1))
        return mask
```

Encode块

结合前面写的部分,直接像拼积木一样拼起来就行了,论文里使用了6个encode块和6个decoder块

Docoder

解码器也类似Encoder堆积木,但是Encoder-decoder在写的时候有一些问题:

- 这一模块的qkv不同,q是解码器每个位置的表示,kv变成了编码器每个位置的表示
- 在训练阶段和eval阶段,这一模块的运算逻辑是不同的

3.模型的训练

因为没有显卡资源,只能把模型放到Colab上运行,只进行了参数调整

```
testloss = 0
batch_size = 1024
lr = 0.0001
dict_number = 10148
epochs = 20
n_layers = 6
seq_len = 32
heads =4
d_model = 512
hidden_dim = 1024
norm_shape = [seq_len, d_model]
```

wramup训练

动态变化学习率,在刚开始训练时大幅降低学习率,随epoch减弱减低幅度

训练结果

完成项目时遇到的困难和感想

这是本人的第一个NLP的深度学习项目,写一个Transformer模型对我来说是一个相当有挑战型的问题,不出意外的是做这个项目的时候出现了各种各样的问题,

- bpevocab表中的字符不全,有些在验证集中的字符缺失
- 在写多头神经网络时,发现matmul,bmm和@对不同tensor.shape的效果是不同的,
- 对多头注意力机制的理解错误,一开始把注意力权重当成类似全连接层的权重了,把qkv分片在使用不同的权重求注意力得分,后来看了网上别人的论文解读和代码才发现这个错误,
- padding mask,这个mask因为理解错误,我把mask做成了对称矩阵,于是在softmax中依然会有一行是[nan,...nan],导致全部tensor变成nan
- 我在写生成位置向量函数时使用了np和tensor.form_numpy(),而np生成的是64位的数,tensor.form_numpy()的dtype是float64(很无语),于是出现了数据类型不一致的问题而反复报错,最后使用tensor.form_numpy().float()让数据一致
- AddNorm,这个层我后来才发现其实可以不用写一个类,当初也想了半天,不知道怎么把前馈网络和多头注意力层放到AddNorm中间残差,这明显是经验不足的问题,又浪费时间又写了

没用的代码.

- bpe分词我现在还不会,测试集不知道怎么分成bpe,模型跑出来的句子还是分词后的,不知道怎么变成完整的句子
- 没有显卡,gpu调试很不方便,而且只能白嫖colab训练,出了问题得先在本地找问题改代码,然后push到github,然后再colab上Git clone,太折磨了
- 模型里新生成的tensor要放入对应的device里,不然也会有报错,比如在GPU上跑是数据在GPU上,此时新生成的位置编码默认在cpu上,会发生错误
- 以前没写过NLP,模型train和eval直接抄别人的了,测试集上的ppl因为分词问题搞不出来
- paperswithcode上MT的指标好像是BLEU,不过我不会
- 可能是初始化的问题第一轮ppl达到了8000多,原因未知

参考文献

[1]Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C]//Advances in neural information processing systems. 2017: 5998-6008.