Project 2 - Bonus: Transformer

21307140069 田沐钊

一、Transformer的原理简述

Transformer的核心思想是完全基于自注意力机制（self-attention mechanism），而不依赖于循环神经网络（RNN）或卷积神经网络（CNN）。它的设计使得模型能够并行计算，加速了训练过程，并且在长距离依赖关系上表现出色。

下面是Transformer模型的主要组成部分：

1. **输入表示**：输入序列首先通过一个嵌入层转换成向量表示。这个嵌入层将每个单词或标记映射到一个高维空间中的实数向量。
2. **位置编码**：由于Transformer没有显式的顺序信息（如RNN中的时间步），因此需要一种方式来编码输入序列中单词的位置信息。位置编码是一组向量，与嵌入向量相加，以表示每个输入标记在序列中的位置。
3. **自注意力机制**：这是Transformer的核心部分。它通过计算输入序列中每个位置与其他位置之间的相对重要性，来捕捉输入序列内部的依赖关系。自注意力机制使用三个线性变换来映射输入序列到查询（Query）、键（Key）和值（Value）的向量空间。
   - 查询（Q）：计算当前位置的重要性得分。
   - 键（K）：表示其他位置的信息。
   - 值（V）：根据键（K）的重要性得分来加权计算加权和。

   通过将查询（Q）、键（K）和值（V）的组合进行加权求和，可以获得自注意力机制的输出。这允许模型在不同位置之间建立关联，并将重要信息传递给后续层。
4. **多头注意力**：为了增强模型的表达能力和学习能力，Transformer使用了多个并行的自注意力机制，称为多头注意力。每个注意力头都有自己的查询、键和值的线性投影，通过将每个注意力头的输出连接起来，可以获得更丰富的表示能力。
5. **前馈神经网络**：在自注意力层的输出上，Transformer还包括一个前馈神经网络，它由两个全连接层组成。这个前馈神经网络对每个位置的表示进行独立的转换，增加了非线性和表示能力。
6. **层归一化**：在每个子层（自注意力和前馈神经网络）之后，Transformer使用层归一化来规范化每个位置的表示。这有助于模型的训练和稳定性。

基于此，transformer建立了名为编码器和解码器的结构，其具体原理如下：

编码器（Encoder）：
编码器由多个相同的层堆叠而成。每个层都包含两个子层：多头自注意力机制和前馈神经网络。

1. **多头自注意力机制**：在每个编码器层中，输入序列先经过一个多头自注意力层。这个自注意力层的输入包括三个部分：查询（Q）、键（K）和值（V）。这些输入都是通过线性变换从输入序列的嵌入表示中计算得到的。
   - 查询（Q）：计算当前位置的重要性得分。
   - 键（K）：表示其他位置的信息。
   - 值（V）：根据键（K）的重要性得分来加权计算加权和。

通过将查询（Q）、键（K）和值（V）的组合进行加权求和，可以获得自注意力机制的输出。这允许模型在不同位置之间建立关联，并将重要信息传递给后续层。

2. **前馈神经网络**：在自注意力层的输出上，编码器的每个层还包含一个前馈神经网络。这个前馈神经网络由两个全连接层组成，通过对每个位置的表示进行独立的转换，增加了非线性和表示能力。
3. **层归一化**：在每个子层（自注意力和前馈神经网络）之后，编码器使用层归一化来规范化每个位置的表示。这有助于模型的训练和稳定性。

编码器的输出是经过多个编码器层处理后的表示，这些表示包含了输入序列的语义和上下文信息。

解码器（Decoder）：
解码器也由多个相同的层堆叠而成。每个层包含三个子层：多头自注意力机制、编码器-解码器注意力机制和前馈神经网络。

1. **多头自注意力机制**：解码器的第一个子层是多头自注意力机制。它与编码器的自注意力机制类似，但还引入了一个遮蔽（masking）机制，以确保在生成每个位置的输出时，模型只能依赖于已经生成的部分，而不能依赖于将来的部分。
2. **编码器-解码器注意力机制**：解码器的第二个子层是编码器-解码器注意力机制。它允许解码器对编码器的输出进行关注。查询（Q）来自解码器的前一层，而键（K）和值（V）来自编码器的输出。这样，解码器可以根据编码器的信息来生成与输入序列相关的输出。
3. **前馈神经网络**：解码器的最后一个子层是前馈神经网络，与编码器的前馈神经网络类似。它对每个位置的表示进行独立的转换。
4. **层归一化**：在每个子层之后，解码器也使用层归一化来规范化每个位置的表示。

通过使用自注意力机制和位置编码，Transformer能够捕捉输入序列中的长距离依赖关系，并在多个任务中取得了显著的性能提升。

二、代码实现与部分展示

在NER任务中，实际并不需要decoder层进行解码，只需要使用encoder层最终输出一句话中每个单词对应每个标签的得分，然后利用交叉熵函数作为损失函数即可。

下面是transformer的代码实现和结构分析：

```
# 词嵌入层
class Embeddings(nn.Module):
    def __init__(self, dim_embed, vocab):
        super(Embeddings, self).__init__()
        self.dim_embed = dim_embed
        self.mapping = nn.Embedding(vocab, dim_embed)

    def forward(self, x):
        """ print('original shape =', x.shape)
        print('shape after embedding = ', self.mapping(x).shape) """
        return self.mapping(x) * math.sqrt(self.dim_embed)


# 位置编码层
class positional_encoding(nn.Module):
```

```
    def __init__(self, dim_embed, dropout, max_len):
        super(positional_encoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        pos_encoding = torch.zeros(max_len, dim_embed)
        pos = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, dim_embed, 2) * -(math.log(10000.0) /
dim_embed))

        pos_encoding[:, 0::2] = torch.sin(pos * div_term)
        pos_encoding[:, 1::2] = torch.cos(pos * div_term)
        pos_encoding = pos_encoding.unsqueeze(0)

        """ self.pos_encoding = pos_encoding """
        # 这么高级的方法我还是之后再用吧
        self.register_buffer('pos_encoding', pos_encoding)

    def forward(self, x):
        """ print('shape of pos_encoding = ', self.pos_encoding.shape) """
        x = x + self.pos_encoding[:, :x.shape[-2]]

        # 嘿，为什么要dropout
        return self.dropout(x)


# encoder和decoder会用到的多头注意力层
class multi_head_attention(nn.Module):
    def __init__(self, dim_embed, num_heads, dropout):
        super(multi_head_attention, self).__init__()
        self.dim_embed = dim_embed
        self.num_heads = num_heads
        self.dim_per_head = dim_embed // num_heads

        self.linear_q = nn.Linear(dim_embed, dim_embed)
        self.linear_k = nn.Linear(dim_embed, dim_embed)
        self.linear_v = nn.Linear(dim_embed, dim_embed)
        self.linear_out = nn.Linear(dim_embed, dim_embed)
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, input, mask=None):
        """ print('mask.shape = ', mask.shape) """
        if mask is not None:
            mask = mask.unsqueeze(1)
        batch_size = input.shape[0]

        query = self.linear_q(input).view(batch_size, -1, self.num_heads,
self.dim_per_head).permute(0, 2, 1, 3)
        key = self.linear_k(input).view(batch_size, -1, self.num_heads,
self.dim_per_head).permute(0, 2, 1, 3)
        value = self.linear_v(input).view(batch_size, -1, self.num_heads,
self.dim_per_head).permute(0, 2, 1, 3)
        """ print('Q.shape = ', query.shape) """
```

```
        output, self.attention_weights = compute_attention(query, key, value, mask)
        """ print('out_put.shape = ', output.shape) """
        output = self.dropout(output)

        # 为什么要做这些奇奇怪怪的维度转换啊
        output = output.permute(0, 2, 1, 3).contiguous().view(batch_size, -1,
self.dim_embed)

        return self.linear_out(output)


# 层归一化
class normalization(nn.Module):
    def __init__(self, size, eps=1e-6):
        super(normalization, self).__init__()
        self.var = nn.Parameter(torch.ones(size))
        self.mean = nn.Parameter(torch.zeros(size))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        std = x.std(dim=-1, keepdim=True)
        return self.var * (x - mean) / (std + self.eps) + self.mean


# 前馈层
class feed_forward(nn.Module):
    def __init__(self, dim_embed, dim_ff, dropout):
        super(feed_forward, self).__init__()
        self.linear1 = nn.Linear(dim_embed, dim_ff)
        self.linear2 = nn.Linear(dim_ff, dim_embed)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.linear2(self.dropout(self.relu(self.linear1(x))))


# 单个encoder块
class encoder(nn.Module):
    def __init__(self, dim_embed, num_heads, dim_ff, dropout):
        super(encoder, self).__init__()
        self.muti_head_attention = multi_head_attention(dim_embed, num_heads,
dropout)
        self.norm_1 = normalization(dim_embed)
        self.feed_forward = feed_forward(dim_embed, dim_ff, dropout)
        self.norm_2 = normalization(dim_embed)

    def forward(self, x, mask):
        """ print(x.shape) """
        x_atten = self.muti_head_attention(x, mask)
```

```
        """ print(x_atten.shape) """
        x = self.norm_1(x + x_atten)
        x_ff = self.feed_forward(x)
        x = self.norm_2(x + x_ff)
        return x


# encoder的堆栈
class encoder_stack(nn.Module):
    def __init__(self, num_layers, dim_embed, num_heads, dim_ff, dropout):
        super(encoder_stack, self).__init__()
        self.encoder_stack = nn.ModuleList([encoder(dim_embed, num_heads, dim_ff,
dropout) for _ in range(num_layers)])

    def forward(self, x, mask):
        for encoder in self.encoder_stack:
            x = encoder(x, mask)
        return x

# 用于transformer的多头注意力层
def compute_attention(Q, K, V, mask):
    d_k = Q.size()[-1]
    scores = torch.matmul(Q, K.transpose(-1, -2)) / math.sqrt(d_k)
    """ print(scores.shape)
    print('mask.shape= ', mask.shape) """
    mask = mask.unsqueeze(1) * mask.unsqueeze(-1)
    """ print('mask.shape = ', mask.shape) """
    scores = scores.masked_fill(mask == 0, -1e16)
    attention_weights = torch.softmax(scores, dim=-1)
    attention_weights = attention_weights.masked_fill(mask == 0, 0)
    return torch.matmul(attention_weights, V), attention_weights
```

下面是对每个组件和层的结构概述：

1. Embeddings（词嵌入层）：
    - 输入参数：dim_embed（词嵌入的维度），vocab（词汇表大小）
    - 初始化：使用nn.Embedding创建一个词嵌入层对象，将词汇表大小和词嵌入的维度作为参数传入
    - 前向传播：将输入x传入词嵌入层，并乘以math.sqrt(dim_embed)进行缩放
2. positional_encoding（位置编码层）：
    - 输入参数：dim_embed（词嵌入的维度），dropout（Dropout的概率），max_len（输入序列的最大长度）
    - 初始化：使用torch.zeros创建一个形状为(max_len, dim_embed)的位置编码矩阵，然后根据位置编码公式计算位置编码值，并使用unsqueeze函数增加一维作为batch维度
    - 前向传播：将输入x加上位置编码矩阵的前x.shape[-2]个位置编码值
3. multi_head_attention（多头注意力层）：
    - 输入参数：dim_embed（词嵌入的维度），num_heads（注意力头的数量），dropout（Dropout的概率）

- 初始化：使用nn.Linear创建线性变换层，分别用于查询（linear_q）、键（linear_k）、值（linear_v）和输出（linear_out）
- 前向传播：对输入进行线性变换后，将结果按头数和头大小进行维度重排，然后调用compute_attention函数计算注意力权重和输出结果

4. normalization（层归一化）：
   - 输入参数：size（输入的维度），eps（用于稳定除法的小值）
   - 初始化：使用nn.Parameter创建可学习的参数变量var和mean
   - 前向传播：对输入进行层归一化操作，即减去均值除以标准差
5. feed_forward（前馈层）：
   - 输入参数：dim_embed（词嵌入的维度），dim_ff（前馈层的隐藏层维度），dropout（Dropout的概率）
   - 初始化：使用nn.Linear创建线性变换层，使用nn.ReLU创建激活函数relu
   - 前向传播：对输入进行线性变换、ReLU激活和Dropout操作
6. encoder（单个编码器块）：
   - 输入参数：dim_embed（词嵌入的维度），num_heads（注意力头的数量），dim_ff（前馈层的隐藏层维度），dropout（Dropout的概率）
   - 初始化：创建一个multi_head_attention对象（用于多头注意力）、一个normalization对象（用于层归一化）和一个feed_forward对象（用于前馈层）
   - 前向传播：依次经过多头注意力、层归一化、前馈层和再次层归一化操作
7. encoder_stack（编码器堆栈）：
   - 输入参数：num_layers（堆栈中编码器的层数），dim_embed（词嵌入的维度），num_heads（注意力头的数量），dim_ff（前馈层的隐藏层维度），dropout（Dropout的概率）
   - 初始化：创建一个由多个encoder对象组成的ModuleList
   - 前向传播：依次经过堆栈中的每个encoder对象

```
# a simple version, which only has an encoder stack
class Transformer(nn.Module):
    def __init__(self, dim_embed, vocab_size, max_len, num_encoder, num_heads,
dim_ff, dim_out, dropout):
        super(Transformer, self).__init__()
        self.embedding = Embeddings(dim_embed, vocab_size)
        self.positional_encoding = positional_encoding(dim_embed, dropout, max_len)
        self.encoder_stack = encoder_stack(num_encoder, dim_embed, num_heads,
dim_ff, dropout)
        self.linear = nn.Linear(dim_embed, dim_out)

    def forward(self, x, mask):
        x = self.embedding(x)
        x = self.positional_encoding(x)
        x = self.encoder_stack(x, mask)
        x = self.linear(x)
        return x
```

```
    def predict(self, x, mask):
        scores = self.forward(x, mask)
        tags_pred = torch.argmax(scores, dim=-1)
        return tags_pred
```

（以上部分主要在layers_of_transformer.py和my_transformer.py文件中）

三、实验结果

测试过程主要在my_transformer_test.ipynb文件中，下面是训练20个epoch后的check结果：

```
              precision    recall  f1-score   support

       B-PER     0.8181    0.6471    0.7226      1842
       I-PER     0.7669    0.4430    0.5616      1307
       B-ORG     0.7916    0.6428    0.7095      1341
       I-ORG     0.7553    0.5260    0.6201       751
       B-LOC     0.8748    0.7681    0.8180      1837
       I-LOC     0.7311    0.6770    0.7030       257
      B-MISC     0.8692    0.7354    0.7967       922
      I-MISC     0.8009    0.5347    0.6412       346

   micro avg     0.8190    0.6365    0.7163      8603
   macro avg     0.8010    0.6218    0.6966      8603
weighted avg     0.8150    0.6365    0.7116      8603
```

并且可以观察到loss在逐渐降低

```
878it [00:18, 48.44it/s]
epoch: 0, loss: 0.19956177348750462
878it [00:15, 55.88it/s]
epoch: 1, loss: 0.11108615746878614
878it [00:15, 56.35it/s]
epoch: 2, loss: 0.07672787645029791
878it [00:15, 56.58it/s]
epoch: 3, loss: 0.05636093800925652
878it [00:15, 56.47it/s]
epoch: 4, loss: 0.044960459153823434
878it [00:15, 56.51it/s]
epoch: 5, loss: 0.036904211694922015
878it [00:15, 57.24it/s]
epoch: 6, loss: 0.031616593195827665
878it [00:15, 56.64it/s]
epoch: 7, loss: 0.028366923281829593
878it [00:15, 56.36it/s]
epoch: 8, loss: 0.026536352305626922
878it [00:15, 55.78it/s]
epoch: 9, loss: 0.022897199987407864
878it [00:15, 55.84it/s]
epoch: 10, loss: 0.021601729594586856
```

```
878it [00:15, 56.37it/s]
epoch: 11, loss: 0.01938380400720742
878it [00:15, 56.78it/s]
epoch: 12, loss: 0.018651901925127723
878it [00:15, 56.23it/s]
epoch: 13, loss: 0.018495793857791775
878it [00:15, 56.35it/s]
epoch: 14, loss: 0.016961502086004967
878it [00:15, 56.91it/s]
epoch: 15, loss: 0.015312478304510472
878it [00:15, 56.28it/s]
epoch: 16, loss: 0.015494537625169055
878it [00:15, 56.44it/s]
epoch: 17, loss: 0.01493827163475894
878it [00:15, 56.34it/s]
epoch: 18, loss: 0.014777711498308959
878it [00:15, 56.31it/s]
epoch: 19, loss: 0.014467934739472586
```

虽然由于时间的不足，并未进行进一步的训练，但足以看出该模型的运行情况良好，性能正常。

四、关于上一个bonus中CRF的补充

在手写CRF的部分中，由于时间问题未能给出实际的训练结果，下面是训练后的效果（训练过程在my_CRF_test.ipynb文件中）：

```
   micro avg     0.8812    0.9111    0.8959      8437
   macro avg     0.6353    0.6755    0.6497      8437
weighted avg     0.8840    0.9111    0.8965      8437
```