Lab 6：注意力机制

Charles

习题1

问题： 分析点积缩放模型可以缓解softmax函数梯度消失的原因

解答： 缩放点积模型为

$$s(\boldsymbol{x}, \boldsymbol{q}) = \frac{\boldsymbol{x}^\top \boldsymbol{q}}{\sqrt{D}}$$

其中 $s(\boldsymbol{x}, \boldsymbol{q})$ 是注意力打分函数，用于计算注意力分布 α_n

$$\alpha_n = \mathrm{softmax}(s(\boldsymbol{x}_n, \boldsymbol{q})) = \frac{\exp(s(\boldsymbol{x}_n, \boldsymbol{q}))}{\sum_{j=1}^N \exp(s(\boldsymbol{x}_j, \boldsymbol{q}))}$$

在未缩放（没有除以 $\sqrt{D}$ ）的情况下，点积模型的值通常会有比较大的方差， $\exp(s(\boldsymbol{x}, \boldsymbol{q}))$ 的方差也就越大，导致softmax的输出接近0或1，梯度趋于0，导致梯度消失；缩放后，缩小了 $s(\boldsymbol{x}, \boldsymbol{q})$ 的范围，相当于softmax的输入更集中了，从而缓解其梯度消失。

习题2

补全 `seq2seq.py` 和 `seq2seq_attention.py` 中的代码如下：

`seq2seq.py`：

```
class Seq2SeqModel(nn.Module):
    def __init__(self):
        super(Seq2SeqModel, self).__init__()
        self.vocab_size = UPPER_CHARS_NUMBER + 1
        self.embedding_layer = nn.Embedding(
            self.vocab_size, embedding_dim=EMBEDDING_LAYER_DIM
        )
        #######################################################
        # 填空
        # 创建一个输入维度为EMBEDDING_LAYER_DIM，隐藏层维度为GRU_HIDDEN_DIM的单层单向GRU
作为encoder
        self.encoder = nn.GRU(
            input_size=EMBEDDING_LAYER_DIM,
            hidden_size=GRU_HIDDEN_DIM,
            num_layers=1,
            batch_first=True,
        )
        #######################################################
        self.decoder = nn.GRU(
            input_size=EMBEDDING_LAYER_DIM,
            hidden_size=GRU_HIDDEN_DIM,
            num_layers=1,
            batch_first=True,
        )
        self.linear = nn.Linear(GRU_HIDDEN_DIM, self.vocab_size)

    def forward(self, encoder_X, decoder_X):
```

```
        ########################################################
        # encoder_X是encoder的输入序列，为(batch_size,sequence_size)的字符index
Tensor
        # decoder_X是decoder的输入序列，为(batch_size,sequence_size)的字符index
Tensor
        # 填空
        # 1. 使用编码器对输入的序列进行编码，得到当前的隐藏层状态(hidden_state)
        encoder_embedding = self.embedding_layer(encoder_X)
        decoder_embedding = self.embedding_layer(decoder_X)
        _, hidden_state = self.encoder(encoder_embedding)

        # 2. 使用encoder得到的隐藏层状态作为decoder的初始隐藏层状态(hidden_state)
        decoder_output, _ = self.decoder(decoder_embedding, hidden_state)

        # 根据decoder每一位的hidden state预测对应的字符可能是哪个
        logit = self.linear(decoder_output).view(-1, self.vocab_size)
        # decoder_output size: (batch_size,sequence_length,hidden_size)
        ########################################################
        return logit
    ...
```

运行结果：

```
step 4799 loss: 0.11361471563577652  -- using time 0.044183
step 4899 loss: 0.1384226530790329  -- using time 0.045430
step 4999 loss: 0.1118415966629982  -- using time 0.047246
model training all using time 187.491
Input: ['PEZXGABZPK'], Should get: KPZBAGXZEP, Model generate: KPZBAGXZEP  |The result is True
Input: ['NRJZILURWR'], Should get: RWRULIZJRN, Model generate: RWRULIZJRN  |The result is True
Input: ['GKWYBKEZES'], Should get: SEZEKBYWKG, Model generate: SEZEKBYWKG  |The result is True
Input: ['EYKMGQTEYT'], Should get: TYETQGMKYE, Model generate: TYETQGMKYE  |The result is True
Input: ['WKHXWVPIAU'], Should get: UAIPVWXHKW, Model generate: UAIPVWXHKL  |The result is False
Input: ['YMHDBIYMTH'], Should get: HTMYIBDHMY, Model generate: HTMYIBDHMY  |The result is True
Input: ['DXOYGCTECL'], Should get: LCETCGYOXD, Model generate: LCETCGYOXD  |The result is True
Input: ['UONEXCKVAA'], Should get: AAVKCXENOU, Model generate: AAVKCXENOO  |The result is False
Input: ['DZXSQFOKYJ'], Should get: JYKOFQSXZD, Model generate: JYKOFQSXZD  |The result is True
Input: ['PEVLUSADOH'], Should get: HODASULVEP, Model generate: HODASULVEP  |The result is True
```

`seq2seq_attention.py`：

```
class Seq2SeqModel(nn.Module):
    ...
    def get_context_vector(self, encoder_out, hidden_state):
        # 用于得到得到聚合信息向量
        # 先计算attention分数后，再根据attention分数得到聚合信息向量(sum(a_1 *
encoder_hidden_state_1 + a_2 * encoder_hidden_state_2 + ...))
        # encoder_out: (batch_size, sequence_length, hidden_size)
        # hidden_state: (1, batch_size, hidden_size)
        weight = torch.bmm(
            self.attention_W(encoder_out), hidden_state.permute(1, 2, 0)
        )  # ( batch_size, sequence_length ,1 )
        ########################################################
        # 填空 对计算出的attention分数进行softmax归一化(要注意对哪一维度的值进行归一化)
        weight = torch.softmax(weight, dim=1)  # (batch_size_sequence_length,1)
        ########################################################
        context_vectors = torch.sum(
            weight * encoder_out, dim=1, keepdim=True
        )  # (batch_size, 1, hidden_size)
        return context_vectors
```

```python
    def decoding(self, encoder_out, decoder_X, hidden_state):
        all_outputs = []
        for i in range(SEQ_LENGTH):
            input_X = decoder_X[:, i].unsqueeze(1)  # (batch_size,1)
            input_embedding = self.embedding_layer(input_X)  # 字符的embedding
            #######################################################
            # 填空
            # 1. 得到聚合信息向量(用attention机制计算得到的那个)
            context_vectors = self.get_context_vector(encoder_out, hidden_state)
            # 2. 将字符的embedding与聚合信息向量进行拼接，以作为decoder的输入
            input_embedding = torch.cat([input_embedding, context_vectors],
dim=-1)
            #######################################################
            output, hidden_state = self.decoder(input_embedding, hidden_state)
            all_outputs.append(output)
        all_outputs = torch.cat(
            all_outputs, dim=1
        )  # (batch_size,sequence_length,hidden_size)
        return all_outputs, hidden_state

    def get_next_token(self, encoder_out, input_X, hidden_state):
        # 用于预测，根据encoder的状态、当前时刻的字符输入以及上一时刻得到的hidden state预测
下一个字符是啥
        # input_X shape: (batch_size, 1)
        # encoder_out shape: (batch_size, sequence_length, hidden_size)
        # hidden_state shape: (1, batch_size,hidden_size)
        #######################################################
        # 填空
        # 可以参考一下seq2seq的get next token函数哦
        input_embedding = self.embedding_layer(input_X)
        context_vector = self.get_context_vector(encoder_out, hidden_state)
        decoder_input = torch.cat([input_embedding, context_vector], dim=-1)
        output, new_hidden_state = self.decoder(decoder_input, hidden_state)
        logit = self.linear(output).squeeze(1)
        output = torch.argmax(logit, dim=1)
        #######################################################
        # output size: (batch_size)  (对应根据decoder当前隐藏层状态，通过线性层分类得到
的，模型认为的最有可能的输出字符的对应index)
        # new_hidden_state (1,batch_size,hidden_size) (当前隐藏层状态)
        return output, new_hidden_state
    ...
```

运行结果：

```
step 4799 loss: 0.4455140233039856  -- using time 0.084810
step 4899 loss: 0.17195142805576324  -- using time 0.126499
step 4999 loss: 0.14972487092018127  -- using time 0.089681
model training all using time 422.258
Input: ['GNRIVMTUWN'], Should get: NWUTMVIRNG, Model generate: NWUTMVIRNG  |The result is True
Input: ['JWNODFTIOA'], Should get: AOITFDONWJ, Model generate: AOITFDONWJ  |The result is True
Input: ['CTNPLQQRTT'], Should get: TTRQQLPNTC, Model generate: TTRQLPNTCC  |The result is False
Input: ['LASMYFKBHB'], Should get: BHBKFYMSAL, Model generate: BHBKFYMSAL  |The result is True
Input: ['KENOIVNQPF'], Should get: FPQNVIONEK, Model generate: FPQNVIONEK  |The result is True
Input: ['WIDUJGBXYT'], Should get: TYXBGJUDIW, Model generate: TYXBGJUDIW  |The result is True
Input: ['TVORUIQKSI'], Should get: ISKQIUROVT, Model generate: ISKQIUROVT  |The result is True
Input: ['ZQORDNIUMC'], Should get: CMUINDROQZ, Model generate: CMUINDROQZ  |The result is True
Input: ['SOJONKOJQQ'], Should get: QQJOKNOJOS, Model generate: QQJONKONKO  |The result is False
Input: ['NUMUDEBSRH'], Should get: HRSBEDUMUN, Model generate: HRSBEDUMUN  |The result is True
```