# lab 7

本次lab专注于使用不同种类的seq2seq模型实现机器翻译。我的报告按照三个模型来分块，把基础和拓展的task实现包含在了下述分析中，其中包含了我对seq2seq模型发展演变过程的认识。

## Vanilla seq2seq

是最基本的seq2seq机器翻译模型，我较为详细地剖析了从数据处理到模型部署的整个过程。

与绝大多数的seq2seq模型一样，最原始最基础的seq2seq由end to end的encoder和decoder两大部分构成。

### 数据处理(包含question1)

我对整个数据处理过程作了注释，如下：

```
#en_word_2_index是二维列表，共data个子列表，每个对应该data所含词在vocabulary中的index构成的列表（即wordID）
class MyDataset(Dataset):
    def __init__(self, en_data, ch_data, en_word_2_index, ch_word_2_index):
        self.en_data = en_data
        self.ch_data = ch_data
        self.en_word_2_index = en_word_2_index
        self.ch_word_2_index = ch_word_2_index
#重载对MyDataset的索引运算符，输入data序数，返回中英文data对应的词ID列表
    def __getitem__(self, index):
        en = self.en_data[index]
        ch = self.ch_data[index]

        en_index = [self.en_word_2_index[i] for i in en]
        ch_index = [self.ch_word_2_index[i] for i in ch]

        return en_index, ch_index

#基于整个dataset来对batch data进行处理
#输入的batch_datas实际上是这些data已经处理好的一堆wordID列表
    def batch_data_process(self, batch_datas):
        global device
        en_index, ch_index = [], []
        en_len, ch_len = [], []

        for en, ch in batch_datas:
            en_index.append(en)
            ch_index.append(ch)
            en_len.append(len(en))
            ch_len.append(len(ch))

        max_en_len = max(en_len)
        max_ch_len = max(ch_len)
        #下面的过程是对传入的wordID作padding以及修饰：
        # en word在最后添加<PAD>对应的ID，使各个wordID列表等长
```

```
        # ch word在起始位置添加<BOS>对应的ID，在结束位置添加<EOS>对应的ID，最后同样
padding一下（但每个ch word应该都是一个字构成，好像区别没有en大）
        #等长后，转为tensor加速运算
        en_index = [i + [self.en_word_2_index["<PAD>"]] * (max_en_len - len(i))
for i in en_index]
        ch_index = [
            [self.ch_word_2_index["<BOS>"]] + i + [self.ch_word_2_index["<EOS>"]]
+ [self.ch_word_2_index["<PAD>"]] * (
                    max_ch_len - len(i)) for i in ch_index]
        en_index = torch.tensor(en_index, device=device)
        ch_index = torch.tensor(ch_index, device=device)

        return en_index, ch_index

#重载len()函数
    def __len__(self):
        assert len(self.en_data) == len(self.ch_data)
        return len(self.ch_data)
```

- question 1 : 这里每一个batch中的句子都一样长吗？为什么？

一样长。通过上述注释分析中的padding过程处理为都和最长句向量维度对齐

## 模型实现（包含task1,2,3，question2,3）

```
#继承torch.nn中的Module，塞入参数把其中的embedding class以及lstm class实例化，
#embedding基于word2vec模型计算得到corpus中各个单词的embedding vector（分布式表示）
    #en_corpus_len表示en data构成的corpus中的单词数
    #参数encoder_embedding_num表示word的embedding vector的维度
#多层lstm构成的encoder，需要传入处理的embedding vector的维度以及encoder内部hidden
layer层数
    #encoder_hidden_num表示encoder lstm模型的hidden layer层数
class Encoder(nn.Module):
    def __init__(self, encoder_embedding_num, encoder_hidden_num, en_corpus_len):
        super().__init__()
        self.embedding = nn.Embedding(en_corpus_len, encoder_embedding_num)
        self.lstm = nn.LSTM(encoder_embedding_num, encoder_hidden_num,
batch_first=True)

#要注意区分：此处的forward是由好几层hidden lstm layers构成的encoder整体的forward，其输
出为最后一层hidden layer的输出
    #输入的是由 一个个仅wordID处为1的one-hot vector（或仅传入wordID，再自行转为one-hot
vector）组成的列表 构成的datas tensor
    def forward(self, en_index):
        # TODO 基础 - task 1：这里实现encoder的过程：首先获得en_index的词嵌入表示，然
后使用lstm模型获得最终输出。
        #作为函数调用embedding，返回值为把en_index datas中的one-hot vector替换为计算
得到的所需维度的embedding vector构成的二维tensor
        en_embedding = self.embedding(en_index)
        #作为函数调用lstm进行前向传播，传出值取为last hidding layer的输出
        _, encoder_hidden = self.lstm(en_embedding)
```

```
        #
        return encoder_hidden

#和encoder的初始化完全一致，仅在forward的输入值和返回值处有差别，下面对此作了解释
class Decoder(nn.Module):
    def __init__(self, decoder_embedding_num, decoder_hidden_num, ch_corpus_len):
        super().__init__()
        self.embedding = nn.Embedding(ch_corpus_len, decoder_embedding_num)
        self.lstm = nn.LSTM(decoder_embedding_num, decoder_hidden_num,
batch_first=True)

#输入为已经得到的上一个decoder的words output（或在第一个encoder hidden layer中人为的
words input）和上一个decoder输出的hidden；输出与之对应

    def forward(self, decoder_input, hidden):
        # TODO 基础 - task 2: 实现decoder的过程：首先获得decoder_input的词嵌入表示，
然后使用lstm模型获得最终输出。
        embedding = self.embedding(decoder_input)
        decoder_output, decoder_hidden = self.lstm(embedding, hidden)
        #
        return decoder_output, decoder_hidden


class Seq2Seq(nn.Module):
    def __init__(self, encoder_embedding_num, encoder_hidden_num, en_corpus_len,
decoder_embedding_num,
                 decoder_hidden_num, ch_corpus_len):
        super().__init__()
        self.encoder = Encoder(encoder_embedding_num, encoder_hidden_num,
en_corpus_len)
        self.decoder = Decoder(decoder_embedding_num, decoder_hidden_num,
ch_corpus_len)
        self.classifier = nn.Linear(decoder_hidden_num, ch_corpus_len)
        self.cross_loss = nn.CrossEntropyLoss()

    def forward(self, en_index, ch_index):
        decoder_input = ch_index[:, :-1]
        label = ch_index[:, 1:]

        # TODO 基础 - task 3: 实现seq2seq的过程。(调用之前实现的encoder,decoder)
        encoder_hidden = self.encoder(en_index)
        decoder_output, _ = self.decoder(decoder_input, encoder_hidden)

        pre = self.classifier(decoder_output)
        loss = self.cross_loss(pre.reshape(-1, pre.shape[-1]), label.reshape(-1))

        return loss
```

- question2: 为什么decoder_input舍弃最后一个字符，label舍弃第一个字符?

  在上面的数据处理过程中我们知道，传入模型的ch_index的数据都是修饰好了的，即添加了和，最后通过decoder生成的output不含,label要与其对齐，舍弃第一个字符。

而人为输入decoder_input时要和模型计算过程得到的中间量decoder_output对齐，而最后的不能作为信息传入，这样可以提升模型性能，更充分利用有效信息。

- question 3: 训练过程和测试过程decoder的输入有何不同?

正如上一个问题中所讨论的，我们在训练过程中把真正的完整的target中文句也传入了模型，作为decoder一开始的input和encoder最后一层hidden layer的输出一起作为decoder第一层hidden layer的输入；而测试过程不知道答案，一开始传入给decoder的只是包含一个。

## 模型改进思路

decoder过程中，我们生成一个个词是通过argmax取出最大概率的索引对应的词；这种做法是greedy的，每次生成一个词后效果不可逆，即我们无法倒回去找更优解；因此，我们可以把目标的单个词的概率最大转为词组的概率最大，这样可以更好地优化output句子的结构。

我们可以采用multi-layer，在层次维度上（相对于时间维度）增加seq2seq层，进一步提高模型的泛化性能。

## Attention Seq2Seq

task部分代码实现：

```
        # TODO 拓展 task a: 实现attention的计算。
        energy = torch.tanh(self.attn(torch.cat((hidden,encoder_outputs), dim=2)))
        attention_v = self.v(energy).squeeze(2)
        value = F.softmax(attention_v, dim=-1)
```

```
        # TODO 拓展 task b: 构造decoder的输入
        a = self.attention(s, encoder_output).unsueeze(1)
        c = torch.bmm(a, encoder_output).transpose(0,1)
        rnn_input = torch.cat((embedded,c), dim=2).transpose(0,1)
```

普通的seq2seq模型只把english sentence的信息通过最后一个hidden layer的output传给decoder来生成目标中文词，这种信息传递是局限的，无法很好地捕捉english sentence的词序信息和结构信息，attention机制就是为了解决这一问题；其引入了一个待训练的权重矩阵，使得decoder的输入更好地和一个个english word相直接联系。

## Transformer Seq2Seq

```
        # TODO 基础 task 4: 调用pytorch中transformer模块实现。
        src = self.pos_encoder(self.encoder_embedding(src))
        tgt = self.pos_encoder(self.decoder_embedding(tgt))
        output = self.transformer(src, tgt)
        output = self.decoder(output)
```

Attention is all you need论文中提出了self-attention机制和transformer模型，其彻底抛弃了此前RNN和LSTM所采用的时序模型，沿用attention的思想，提出了第一个"完全"采用self-attention构建出来的模型。其把输入句

子中单词直接的联系由O(n)调整为了O(1)，可以极好地捕捉词之间的联系，进而更好地encode句意。

这个模型可以采取数个trick，包括：

- 对模型正则化，除包含句子长度的变量，使得能更好地处理长句子。
- 采取和前面提到的seq2seq改进方案类似的多头注意力机制，增加transformer的层次。
- 对label作mask处理，避免"作弊"，从而增加问题难度，训练得到更有效的参数。

---

通过本次lab，我初步了解了seq2seq模型和attention机制的代码实现过程，看完了Stanford CS224n中相关的三节课程，阅读了《深度学习进阶—自然语言处理》一书中的部分章节，对NLP有了宏观上大概的了解，收获蛮大，体验不错~