学院：数据科学与计算机学院　专业：计算机科学与技术　科目：自然语言处理
姓名：郑康泽　学号：17341213

# NLP第二次大作业报告

## 基于深度学习的中英机器翻译

（数据集为8000条训练集和1000条测试集）

## 一. 预处理

Python 3.7

Pytorch 1.3

## 二. 预处理

1. 分词
   本次利用的分词工具是jieba，首先对于训练集和测试集中的每个句子`sentence`，然后利用`jieba.lcut(sentence)`，将句子进行分词处理。因为分词结果可能会出现空格，而空格不能当做词，所以要对分词后的每个词`word`进行`word = word.strip()`，然后对最后分词形成的列表进行筛选，去掉空字符串`''`，这样就获得一个句子包含的所有词。注意这里并不用去掉标点符号，因为标点符号对于一个句子来说是有意义的。
2. 定义特殊符号
   对于每个句子，需要在句子的开头加上"<BOS>"，在句子的结尾加上"<EOS>"。并且对于训练时的同一批数据，需要将训练句子的长度补到等长max_length，所以在句子长度小于max_length的句子后面，需要添加"<PAD>"进行填充。还需要定义一个特殊符号"<UNK>"，因为我们建立的词典是基于训练集的语料，而在测试集的语料可能出现词典中没有的词，对于训练集中的这种的词，用"<UNK>"来替代。
   以上的特殊符号都当作词进行处理。

3. 创建词典
   根据分词结果，为源语言和目标语言创建词典。词典包括两种，一种是从词到数字的映射，另外一种是数字到词的映射。第一种词典是将文字转化为能喂入网络的数字数据，第二种词典是用于将预测结果转化为文字。

4. 整体流程
   首先将训练集中的源语言文本和目标语言文本进行分词处理，然后在每个句子的首尾添加相应的特殊符号，然后对于每个句子所含有的词语，对对应语言的词典进行更新。然后再对测试集的源语言文本和目标文本进行分词处理，并在首尾加上相应的特殊符号。在预处理这一部分，先不加"<PAD>"和"<UNK>"，这些特殊符号将在即将喂入网络之前才添加。

5. 代码展示（只展示训练数据的预处理代码，测试数据的预处理代码基本相同）

```
def preprocess_train():
    '''
    将训练集中的源语言文本和目标语言文本分词，并为两种语言建立词典
    '''
    global max_length
    with open(New_Train_Path[0], 'w', encoding='utf-8') as f:
        lines = open(Train_Path[0], 'r', encoding='utf-8').readlines()
        for line in lines:
            # 统一小写
            line = line.lower().strip()
            # 分词
            words = jieba.lcut(line)
            # 去空格
            for i in range(len(words)):
                words[i] = words[i].strip()
            while '' in words:
                words.remove('')
            # 更新词典
            for word in words:
                if word not in Source_Word2Num.keys():
                    Source_Word2Num[word] = len(Source_Word2Num)
            # 写入分词结果
            sentence = '<BOS> ' + ' '.join(words) + ' <EOS>' + '\n'
            f.write(sentence)

    with open(New_Train_Path[1], 'w', encoding='utf-8') as f:
        lines = open(Train_Path[1], 'r', encoding='utf-8').readlines()
        for line in lines:
            # 统一小写
            line = line.lower().strip()
            # 分词
            words = jieba.lcut(line)
            # 去空格
            for i in range(len(words)):
```

```
                words[i] = words[i].strip()
            while '' in words:
                words.remove('')
            # max_length = max(len(words), max_length)
            # 更新词典
            for word in words:
                if word not in Target_Word2Num.keys():
                    Target_Word2Num[word] = len(Target_Word2Num)
            # 写入分词结果
            sentence = '<BOS> ' + ' '.join(words) + ' <EOS>' + '\n'
            f.write(sentence)
        # print(max_length)

    # 将源语言词典写进文件
    with open(Source_Word2Num_Path, 'w', encoding='utf-8') as f:
        for (key, value) in Source_Word2Num.items():
            f.write(key + ' ' + str(value) + '\n')
    # with open(Source_Num2Word_Path, 'w', encoding='utf-8') as f:
    #     for (key, value) in Source_Word2Num.items():
    #         f.write(str(value) + ' ' + key + '\n')

    # 将目标语言词典写进文件
    with open(Target_Word2Num_Path, 'w', encoding='utf-8') as f:
        for (key, value) in Target_Word2Num.items():
            f.write(key + ' ' + str(value) + '\n')
    # with open(Target_Num2Word_Path, 'w', encoding='utf-8') as f:
    #     for (key, value) in Target_Word2Num.items():
    #         f.write(str(value) + ' ' + key + '\n')
```

## 三. 模型

1. Encoder

   Encoder中两层，一层是Embedding词嵌入层，另一层是Bidirection Lstm双向LSTM层。Embedding词嵌入层就是利用一个维度为[vocab_size, embedding_dim]的矩阵（vocab_size是词典中词语的个数，embedding_dim是一个词的词向量的维度），将代表每个单词的数字转化为长度为embedding_dim的向量。那么假设一个句子长度为seq_len，那么该句子经过Embedding层后，将会变成维度为[seq_len, embedding_dim]的矩阵。同理如果有batch_size个长度为seq_len的句子经过Embedding层的话，将会变成维度的[batch_size, seq_len, embedding_size]的高维矩阵。Bidirection Lstm用于发现句子中单词序列的某种潜在关系，如果Bidirection Lstm中隐单元数设置为nhidden，那么Bidirection Lstm中每个节点输出的高维矩阵的最后一维为nhidden。假设Embedding层的输出为[batch_size, seq_len, embedding_size]，那么经过Bidirection Lstm层后，维度将变成[batch_size, seq_len, 2 * nhidden]，乘

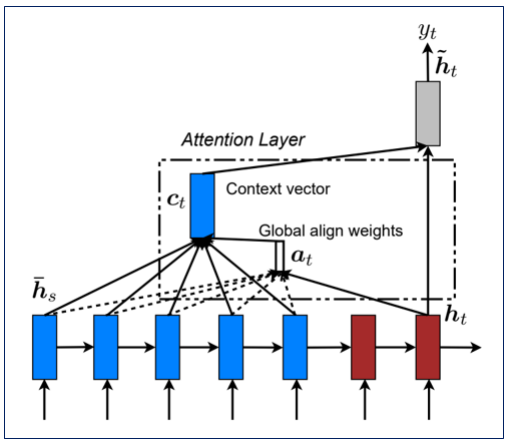
上二是因为Lstm是双向的，相当于有两层，两层的最后的节点的输出拼接起来的，Bidirection Lstm层的第一个节点的隐状态(h, c)随机初始化。

2. Decoder

Decoder的结构与Encoder相似，有一层Embedding层和一层单向的Lstm层。Decoder每次输入的都是batch_size个句子中的一个单词，即维度为[batch_size, 1, 1]，经过Embedding层后维度变成[batch_size, 1, embedding_dim]，然后经过单向的Lstm层后维度变成[batch_size, 1, nhidden]，这里的embedding_dim可以与Encoder的embedding_size不一样的，但这里nhidden必须与Encoder的nhidden，这是为了在计算注意力机制保证维度的匹配。可以看到，Decoder的Lstm只用到一个节点，那么该节点的初始隐状态是不是直接随机初始化呢？Luong这篇论文给出了答案，该节点的初始隐状态应该是Encoder的Bidireciton Lstm最后一个节点输出的隐状态(h, c)，但是因为Encoder的Lstm是双向的，所以需要将最后节点的隐状态改下维度，才能当做Decoder的Lstm的唯一一个节点的初始隐状态。注意，在基于注意力机制的机器翻译中，Decoder的输出并不是真正的输出。

3. Attention

Attention部分是基于注意力机制的机器翻译中最重要的部分，Luong论文中提出来一个模型例子及三条重要的公式，模型如下：




三条公式为：

$$\alpha_{ts} = \frac{exp(score(\mathbf{h}_t, \bar{\mathbf{h}}_s))}{\sum_{s'=1}^{S} exp(score(\mathbf{h}_t, \bar{\mathbf{h}}_{s'}))} \qquad [Attention\ weights]$$

$$\mathbf{c}_t = \sum_s \alpha_{ts} \bar{\mathbf{h}}_s \qquad [Context\ vector]$$

$$\mathbf{a}_t = f(\mathbf{c}_t, \mathbf{h}_t) = tanh(\mathbf{W_c}[\mathbf{c}_t : \mathbf{h}_t]) \qquad [Attention\ vector]$$

其中，$score(\mathbf{h}_t, \bar{\mathbf{h}}_s)$有两种计算方式，其中一种称为Luong's multiplicative style，计算公式为$score(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \mathbf{h}_t^T \mathbf{W} \bar{\mathbf{h}}_s$，另外一种称为Bahdanau's additive style，计算公式为$score(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \mathbf{v}_a^T tanh(\mathbf{W_1}\mathbf{h}_t + \mathbf{W_2}\bar{\mathbf{h}}_s)$。
我选择的是计算比较简单的Luong's multiplicative style，因为上面的公式

是针对于编码器的Bidirection Lstm的一个节点的输出而言，跟代码中的输出维度有点差异，有点难理解，所以我将会用符合代码的输出维度解释。对于编码器的Bidirection Lstm的所有节点的输出，它的维度为[batch_size, seq_len, nhidden]（注意，输出本应该是[batch_size, seq_len, 2 * nhidden]，但我们通常将两个方向的输出的第三维度上的值相加，从而使输出的第三维度减半），然后让该输出经过一个全连接层，保持第三维的维度，即全连接之后的维度还是[batch_size, seq_len, nhidden]，然后将解码器的Lstm的输出乘上全连接层的输出的转置，即[batch_size, 1, nhidden] * [batch_size, nhidden, seq_len], 相乘得到的结果的维度为[batch_size, 1, seq_len]，然后对第三维度的值进行一个softmax即得到attention weights。接着再乘上编码器的Bidirection Lstm的输出，即[batch_size, 1, seq_len] * [batch, seq_len, nhidden]，得到的结果即是Context vector，维度为[batch_size, 1, nhidden]。接着将Context vector与解码器的Lstm的输出在第三维度上拼接，维度变成[batch_size, 1, 2 * nhidden]，然后该结果经过一个全连接层，将第三维度变成nhidden，再通过激活函数，再经过一个全连接层，将第三维度变成目标语言的字典大小vocab_size，也就是整个网络的输出（[batch_size, 1, vocab_size]）。对于第一个句子，预测词语的数字映射可以是[0, 1, vocab_size]中值最大的对应的索引。

4. 代码展示

    1. Encoder

```
class Encoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim, nhidden):
        super(Encoder, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.nhidden = nhidden

        self.embedding = nn.Embedding(self.vocab_size,
self.embedding_dim)
        self.bilstm = nn.LSTM(self.embedding_dim, self.nhidden,
bidirectional=True)

    def forward(self, x, hidden=None):
        seq_len, batch_size = x.size()
        if hidden is None:
            init_h = x.data.new(2, batch_size,
self.nhidden).fill_(0).float()
            init_c = x.data.new(2, batch_size,
self.nhidden).fill_(0).float()
        else:
            init_h, init_c = hidden
        embedding_x = self.embedding(x)
        # output(seq_len, batch_size, 2 * nhidden)
```

```
        # state:(h, c); h(2, batch_size, nhidden), c(2, batch_size,
nhidden)
        output, state = self.bilstm(embedding_x, (init_h, init_c))
        # output(seq_len, batch_size, nhidden)
        output = output[:, :, :self.nhidden] + output[:, :,
self.nhidden:]

        return output, state
```

2. Attention

```
class Attention(nn.Module):
    def __init__(self, nhidden):
        super(Attention, self).__init__()
        self.nhidden = nhidden
        self.dense = nn.Linear(self.nhidden, self.nhidden)

    def forward(self, encoder_output, target_output):
        # encoder_output(batch_size, seq_len, nhidden)
        encoder_output = encoder_output.transpose(0, 1)
        # target_output(batch_size, 1, nhidden)
        target_output = target_output.transpose(0, 1)
        # dense_encoder_output(batch_size, nhidden, seq_len)
        dense_encoder_output = self.dense(encoder_output).transpose(1,
2)
        # score(batch_size, 1, seq_len)
        score = torch.bmm(target_output, dense_encoder_output)
        # attention_weight(batch_size, 1, seq_len)
        attention_weight = F.softmax(score, dim=2)
        # context(batch_size, 1, nhidden)
        context = torch.bmm(attention_weight, encoder_output)
        # context(1, batch_size, nhidden)
        context = context.transpose(0, 1)
        return context
```

3. Decoder

```
class Decoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim, nhidden):
        super(Decoder, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.nhidden = nhidden

        self.embedding = nn.Embedding(self.vocab_size,
self.embedding_dim)
        self.attention = Attention(self.nhidden)
        self.lstm = nn.LSTM(self.embedding_dim, self.nhidden)
        self.dense1 = nn.Linear(2 * self.nhidden, self.nhidden)
        self.dense2 = nn.Linear(self.nhidden, self.vocab_size)

    def forward(self, x, encoder_output, encoder_state):
        embedding_x = self.embedding(x)
        output, state = self.lstm(embedding_x, encoder_state)
```

```
        # context(1, batch_size, nhidden)
        context = self.attention(encoder_output, output)
        # context_output(1, batch_size, 2 * nhidden)
        context_output = torch.cat((context, output), 2)
        # dense1_output(1, batch_size, nhidden)
        dense1_output = F.torch.tanh(self.dense1(context_output))
        # decoder_output(1, batch_size, vocab_size)
        decoder_output = F.softmax(self.dense2(dense1_output), dim=2)

        return decoder_output, state
```

# 四. 训练

1. 训练数据的准备
   本来应该是对每批训练数据进行padding，即每批数据的长度可能不一。但觉得每个batch都这么做太耗时，所以就干脆把所有的句子都padding到一个定长了。但这种做法显然是不可取的，不过为了训练快点，因为每次训练的时间都特别长，最后的效果也不怎么样。
2. Decoder的输入
   对于Decoder的输入，有一个Teacher Forcing Ratio来控制，即这个概率控制Decoder本次输入的是标准答案还是根据上次的输出做出的预测。可以利用`random.random()`来模拟这个概率，当`random.random() < Teacher_Forcing_Ratio`时，Decoder输入正确答案，否则Decoder输入根据上次输出做出的预测
3. 损失函数
   根据Decoder的输出，与标准答案的词向量做交叉熵，作为损失函数。每个epoch的loss是每次batch的loss的和，而每次batch的loss是所有预测词的词向量与标准答案的词向量做交叉熵除以词数。
4. 代码展示
   1. 将文本转为句向量

```
def sen2vec(text_path, word2num, padding=True):
    '''
    :param text_path: 文件路径
    :param word2num: 词语到数字的映射的字典
    :param padding: 是否padding
    :return: 句向量
    '''
    lines = open(text_path, 'r', encoding='utf-8').readlines()
    max_len = max([len(line.strip().split()) for line in lines])

    data = []
    for line in lines:
        line = line.strip().split()
```

```
            vec = []
            for word in line:
                if word in word2num.keys():
                    vec.append(word2num[word])
                else:
                    vec.append(word2num['<UNK>'])
            if padding:
                # 长度不足
                if len(vec) < max_len:
                    vec += (max_len - len(vec)) * [word2num['<PAD>']]
            data.append(vec)
        return data
```

2. 根据Teacher Forcing Ratio选择Decoder的输入

```
for j in range(len(train_target[0]) - 1):
    # 不是第一次则随机
    if j:
        # 输入正确答案
        if random.random() < Teacher_Force_Ratio:
            decoder_input = batch_train_target[j].unsqueeze(0)
            decoder_output, decoder_state = decoder(decoder_input,
encoder_output, decoder_state)
        # 输入上一次输出
        else:
        _, decoder_input = torch.max(decoder_output, 2)
        decoder_output, decoder_state = decoder(decoder_input,
encoder_output, decoder_state)
    # 第一次只能是<BOS>
    else:
        decoder_input = batch_train_target[j].unsqueeze(0)
        decoder_output, decoder_state = decoder(decoder_input,
encoder_output, decoder_state)
    # 交叉熵
    per_batch_loss += criterion(decoder_output.squeeze(),
batch_train_target[j + 1])
```

# 五. 测试

1. Decoder的输入
   在测试过程，Decoder的输入永远都是对于上次输出做出的预测。
2. 何时停止预测
   当预测出来的为"<EOS>"或者预测的句子长度到达的最大长度时，停止预测。
3. 集束搜索Beam Search
   设beam width为k，则对于Decoder的第一次输出，选取概率值前k大的作为候选答案，然后分别输入到Decoder，之后每次就只选择最大概率作

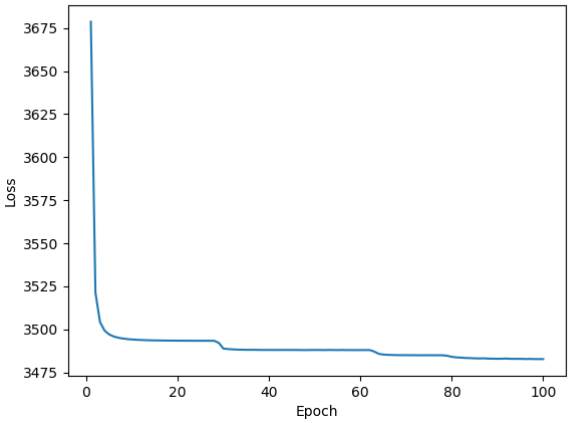
为答案，即生成k个备选句子，最后再从这k个备选句子里选出概率和最大的作为最终的答案。

4. BLEU的计算

利用`nltk.translate.bleu_score.sentence_bleu`计算即可。

## 六. 结果展示

模型效果十分差，尽管loss有在收敛，但收敛得特别慢。翻译出来的句子与标准答案之间的BLEU几乎为零。由于时间关系，并太多没有时间去尝试不同的Teacher Forcing Ratio对于模型的影响。下面是每个epoch的loss的图像：




## 七. 总结与思考

本次大作业的难度比上次大作业的难度要大得多，并且训练的时间也要更长，100个epochs基本上要训练1天，并且由于开始的时间比较晚，所以也没有时间去做调参之类的工作。本次大作业的难处主要是在于理解Attention这个机制到底是怎么算的，由于那三条公式是对于Encoder的Bidirection Lstm的每个节点的输出来计算的，让我很难理解，最终在参考了参考资料终于搞懂了，但是写代码的时候又晕了，因为维度的问题，我debug了很久。其次是Beam Search的问题，理解起来不困难，但实现起来就不知道从而下手，最终在同学的帮助下，终于实现了Beam Search。

虽然这次模型的效果不是很好，但也学习到了基于Attention机制的神经机器翻译，至少自己也会算、会写代码了，收获还是颇丰的。

## 八. 参考资料

1. pytorch.org/tutorias/intermediate/seq2seq_translation_tutorial.html
2. tensorflow.google.cn/tutorials/text/nmt_with_attention
3. https://www.tensorflow.org/tutorials/text/nmt_with_attention
4. https://arxiv.org/abs/1508.04025v5