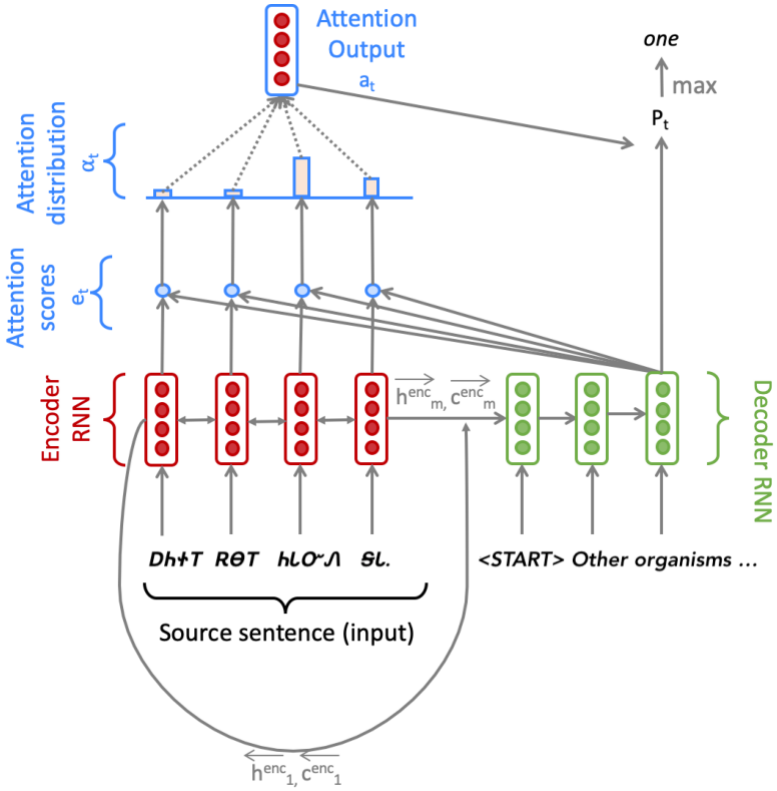
1. Neural Machine Translation with RNNs

代码实现的是一个解码器为BiLSTM，解码器为LSTM的NMT模型，注意力机制用的是乘法注意力（作业式(7)），源语言为小语种切罗基语（虽然这个语言在谷歌翻译中没有，不过在百度翻译里是有的，所以下面的一些问题是可以作弊的）。

关于模型结构BiLSTM，下面这张摘自作业中的图已经写得很明白了：



具体的数学公式笔者觉得跟双向RNN区别不大，自己看一下吧，反正代码里也不用实现，直接调用接口就行了。

- (a) 序列填补，非常基础的工作；

```
    max_length = max(list(map(len, sents)))
    sents_padded=[sent + [pad_token] * (max_length - len(sent)) for sent in
sents]
```

- (b) 初始化嵌入层，非常基础的工作，注意设置`torch.nn.Embedding`的`padding_idx`参数；

```
        self.source = nn.Embedding(len(vocab.src), embed_size,
padding_idx=src_pad_token_idx)    # 切罗基语字符嵌入层
        self.target = nn.Embedding(len(vocab.tgt), embed_size,
padding_idx=tgt_pad_token_idx)    # 英语单词嵌入曾
```

- (c) 初始化机器翻译模型的所有网络层，提示中都说明了每个变量对应作业文件里的哪个数学标记，非常好写：

```
        self.encoder = nn.LSTM(embed_size,hidden_size, bias=True,
bidirectional=True)
        self.decoder = nn.LSTMCell(embed_size + hidden_size, hidden_size,
bias=True)
        self.h_projection = nn.Linear(2 * hidden_size, hidden_size,
bias=False)                 # 双向LSTM得到双倍的隐层状态拼接后与W_h相乘，形状是h×2h
        self.c_projection = nn.Linear(2 * hidden_size, hidden_size,
bias=False)                 # 双向LSTM得到双倍的记忆元胞拼接后与W_c相乘，形状是h×2h
        self.att_projection = nn.Linear(2 * hidden_size, hidden_size,
bias=False)               # W_attProj形状是2h×h
        self.combined_output_projection = nn.Linear(3 * hidden_size,
hidden_size, bias=False)   # W_u形状是h×3h
        self.target_vocab_projection = nn.Linear(hidden_size,
len(vocab.tgt), bias=False)       # W_vocab形状是V_t×h
        self.dropout = nn.Dropout(p=dropout_rate)
```

- (d) 按照代码里的提示一步步走，通过测试：`python sanity_check.py 1d`

```
        X = self.model_embeddings.source(source_padded)
# (seqlen, batchsize, embsize)
        X = pack_padded_sequence(X, source_lengths)
        enc_hiddens, (last_hidden, last_cell) = self.encoder(X)
# last_hidden / last_cell : (2, batchsize, hiddensize)
        enc_hiddens = pad_packed_sequence(enc_hiddens)[0]
# (seqlen, batchsize, embsize)
        enc_hiddens = enc_hiddens.permute(1, 0, 2)
# (batchsize, seqlen, embsize)
        init_decoder_hidden = self.h_projection(torch.cat((last_hidden[0],
last_hidden[1]), dim=1))     # (batchsize, 2 * hiddensize) -> (batchsize,
hiddensize)
        init_decoder_cell = self.c_projection(torch.cat((last_cell[0],
last_cell[1]), dim=1))           # (batchsize, 2 * hiddensize) ->
(batchsize, hiddensize)
```

注意需要下载`nltk_data`的`punkt`分词器，建议提前离线下载好。

- (e) 通过测试：`python sanity_check.py 1e`

```
        enc_hiddens_proj = self.att_projection(enc_hiddens)             #
(batchsize, seqlen, hiddensize)
        Y = self.model_embeddings.target(target_padded)                 #
(tgtlen, batchsize, embsize)
        for Y_t in torch.split(Y, 1):
            Y_t = torch.squeeze(Y_t, dim=0)                             #
(1, batchsize, embsize) -> (batchsize, embsize)
            Ybar_t = torch.cat((Y_t, o_prev), dim=1)                    #
(batchsize, embsize + hiddensize)
            dec_state, o_t, e_t = self.step(Ybar_t, dec_state, enc_hiddens,
enc_hiddens_proj, enc_masks)
            o_prev = o_t
            combined_outputs.append(o_t)
        combined_outputs = torch.stack(combined_outputs, dim=0)         #
(tgtlen, batchsize, hiddensize)
```

- (f) 这一问始终不能通过测试，但是我觉得应该是没有问题才对，不排除测试样例出错的可能性。

- (g) 这个应该说的是做mask的用处，本质上在句子中抹去一些单词以达到屏蔽或选择特定元素的目的，这有点类似dropout以及ReLU激活函数的用处，可以防止模型过拟合。

  还有一个就是更重要的点是mask将`src_len`之前的设为1，之后的设为0，因为在序列模型的生成中不能通过未来的数据作为**已知**的信息，而注意力机制会产生这个问题，因此通过mask来修正模型对未来数据的已知。

接下来依次执行`run.bat vocab`，`run.bat train_local`，`run.bat train`，`run.bat test`，这里有一个问题就是可能`win32file`这个库无法导入，报错为：

```
ImportError: DLL load failed: 找不到指定的程序。
```

在Win11上实在是摆不平这个问题，只能到虚拟机上跑了。

- (h) BLEU指标达到13.068，符合要求。
- (i) 一些简单的看法：
  - (1) 点积注意力只能用于解码器隐层和编码器隐层维度相同，乘法则没有这个限制，且点积本身就是乘法的一种特殊形式，但是点积用起来很方便，不需要找权重矩阵W；
  - (2) 加法注意力揭示了解码器与编码器隐层更复杂的关联（使用了更多的参数），理论上效果会更好，但是加法注意力本身运算复杂度更高，容易使得模型训练时间加长。

2. Analyzing NMT Systems

- (a) 切罗基语是一种**多义合成语言**，单词的每个字符都表示该单词的一部分含义，因此使用字符级别的编码是更合适的，而且这样做相较于单词级别的编码，词嵌入空间也更小，可以简化模型。
- (b) 提示中说前缀是一种语素，即前缀也表达了某种含义，切罗基语中的字符可以表示一种前缀，因此每个字符都是具有独立语义的，应当使用字符级别的编码。
- (c) 所谓多语言训练，指的是在源语言平行语料较少的情况下，可以考虑将源语言与平行语料较多的语言混合起来一起训练（比如将切罗基语和中文混合起来，一起翻译成英文，基于这两混合的平行语料训练机器翻译模型）。

  这种做法的合理性来源于迁移学习，即假定不同语言之间存在概率分布上的共性，因此使用中英互译语料训练得到的模型也可以套用在切罗基语译英语的模型上。

  另一种解释是机器学习模型架构是具有共通性的，训练数据少因而难以将模型参数训练收敛到最优解，但是如果我们用另一种语言的平行语料来进行预训练，可以得到模型参数的一个较好的初始点（比如距离最优解很近），从这个初始点开始进行训练，即便是平行语料较少的切罗基语，也可以较好地收敛到最优解。

- (d) 可以到https://www.cherokeedictionary.net/去查询切罗基语的单词含义（但是我查了几个好像都没有结果，不知道具体是什么操作）。
  - (1) 这个属于代词错误，应该是切罗基语中各个人称代词是不加区分的。这种问题类似英文中的uncle,aunt翻译到中文中有若干单词与其对应，这种问题通常是比较无解的，但是这里显然还有另一个问题，就是前后代词不一致，那么就应该考虑建立文本序列长距离的联系，可以使用LSTM或GRU来挖掘前后代词距离较远的情况下的语义联系。
  - (2) 这个应该是遇到了一个训练集中不存在的切罗基语的单词（零射问题），解决方案可能只能是扩充训练集语料，否则感觉是比较无解的。
  - (3) 这个应该是某种固定用法的翻译错误，类似英文中俚语翻译到中文可能会出这种偏差，解决方案是可能可以通过建立俚语字典（引入强制规则）。
- (e) 这一问需要完全运行完代码得到output/test_outputs.txt才能作答，代码运行时间是比较长的：
  - (1) 找了一个there is no distinction between，的确是在chr_en_data/train.en中出现过，这说明机器翻译系统的确有学习到整个序列的特征。

当然肯定可以找到一个没有出现过的短语，那我又可以说机器翻译系统的解码过程是逐字解码的，出现新的短语也不奇怪。

  - 仍然以there is no distinction between为例，下一个单词是them，但是在训练集中下一个单词是Jew，说明模型的解码并没有在做**贪心解码**，而是在寻求全局最优（可见[slides]中机器翻译解码算法的相关内容）。

- (f) 关于BLEU在[notes]部分笔注已做详细记录（5.3节），但是好像跟这里的公式有一些区别。

$$p_n = \frac{\sum_{\text{ngram}\in c} \min\left(\max_{i=1,2,\ldots,n} \text{Count}_{r_i}(\text{ngram}), \text{Count}_c(\text{ngram})\right)}{\sum_{\text{ngram}\in c} \text{Count}_c(\text{ngram})} \quad (a4.2.1)$$

$$\text{BP} = \begin{cases} 1 & \text{if } \text{len}(c) \geq \text{len}(r) \\ \exp\left(1 - \frac{\text{len}(r)}{\text{len}(c)}\right) & \text{otherwise} \end{cases}$$

$$\text{BLEU} = \text{BP} \times \exp\left(\sum_{n=1}^{k} \lambda_n \log p_n\right)$$

其中c是机器翻译得到的序列，r_i是标准翻译的序列（可能会有多个），$\text{len}(r)$是从r_i中找一个长度最接近c的序列（如果有多个长度最近的则选那个最短的），k是指定的最长的ngram，一般取4，λ_i是一系列累和为1的权重系数。

为了方便解下面的题，笔者编写了一段用于计算BLEU的代码：

```
# -*- coding: utf-8 -*-
# @author: caoyang
# @email: caoyang@163.sufe.edu.cn
# 计算BLEU指数

import numpy
from collections import Counter

def calc_bleu(nmt_translation, reference_translations, lambdas, k=4):
    # 权重系数的长度应当与ngram的最大长度相同
    assert len(lambdas) == k

    # 期望输入的是已经分好词的两个语句序列，否则需要首先进行分词
    if isinstance(nmt_translation, str):
        nmt_translation = nmt_translation.split()
    for i in range(len(reference_translations)):
        if isinstance(reference_translations[i], str):
            reference_translations[i] = reference_translations[i].split()

    # 变量初始化
    nmt_ngram_counters = []                                 # 储存机器翻译序列的中所有的
ngram短语，并记录它们在机器翻译序列的中出现的次数
    reference_ngram_counters = []                           # 储存机器翻译序列的中所有的
ngram短语，并记录它们在机器翻译序列的中出现的次数
    p_ns = []                                               # 储存所有p_n的取值
    length_nmt_translation = len(nmt_translation)    # 机器翻译序列的长度len(c)

    # 计算len(r)
    length_reference_translation_min = float('inf')
    flag = float('inf')
    for reference_translation in reference_translations:
        length_reference_translation = len(reference_translation)
        error = abs(length_reference_translation - length_nmt_translation)
        if error <= flag and length_reference_translation <=
length_reference_translation_min:
```

```
            length_reference_translation_min = length_reference_translation
            flag = error

    # 统计机器翻译序列中的ngram频次
    for n in range(k):
        ngrams = []
        for i in range(length_nmt_translation - n):
            ngrams.append(' '.join(nmt_translation[i:i + n + 1]))
        nmt_ngram_counters.append(dict(Counter(ngrams)))
    # print(nmt_ngram_counters)
    # print('-' * 64)

    # 统计标准翻译序列中的ngram频次
    for reference_translation in reference_translations:
        reference_ngram_counters.append([])
        for n in range(k):
            ngrams = []
            for i in range(len(reference_translation) - n):
                ngrams.append(' '.join(reference_translation[i:i + n + 1]))
            reference_ngram_counters[-1].append(dict(Counter(ngrams)))
    # print(reference_ngram_counters)
    # print('-' * 64)

    # 计算p_n
    for n in range(k):
        p_n_numerator = 0       # p_n的分子部分
        p_n_denominator = 0     # p_n的分母部分
        for ngram in nmt_ngram_counters[n]:
            p_n_numerator += min([max([reference_ngram_counters[i]
[n].get(ngram, 0) for i in range(len(reference_ngram_counters))]),
nmt_ngram_counters[n][ngram]])
            p_n_denominator += nmt_ngram_counters[n][ngram]
        p_n = p_n_numerator / p_n_denominator
        p_ns.append(p_n)

    # 计算BP
    if length_nmt_translation > length_reference_translation_min:
        bp = 1
    else:
        bp = numpy.exp(1 - length_reference_translation_min /
length_nmt_translation)

    # 计算BLEU
    bleu = bp * numpy.exp(sum([lambda_ * numpy.log(p_n) for lambda_, p_n in
zip(lambdas, p_ns)]))
    return bleu

reference_translations = [
    'the light shines in the darkness and the darkness has not overcome it',
    'and the light shines in the darkness and the darkness did not
comprehend it',
]

nmt_translations = [
    'and the light shines in the darkness and the darkness can not
comprehend',
    'the light shines the darkness has not in the darkness and the trials',
]
```

```
for nmt_translation in nmt_translations:
    bleu = calc_bleu(nmt_translation=nmt_translation,
                     reference_translations=reference_translations,
                     lambdas=[.5, .5, .0, .0],
                     k=4)
    print(bleu)
```

- (1) 直接运行上面的代码，可得：

$$\mathrm{BLEU}(c_1) = 0.877 \quad \mathrm{BLEU}(c_2) = 0.797 \tag{a4.2.2}$$

  我认同这个结果，c_1确实翻译得要更正确。

- (2) 注释掉78行再运行，可得：

$$\mathrm{BLEU}(c_1) = 0.716 \quad \mathrm{BLEU}(c_2) = 0.797 \tag{a4.2.3}$$

  现在我就不那么认同这个结果了。

- (3) 译文通常都是不唯一的，因此如果只有一个参考译文作为评判标准，的确是不够客观与准确的。在有多参考译文的情况下，BLEU表现得更好。

- (4) 与人类评估相比的优势：简便快捷，可以批量操作，省时省力；不容易出现疲劳误判等；

  与人类评估相比的劣势：BLEU本质上只考虑了短语的频数特征，并没有考察译文的流畅性，完全可以构造出一个BLEU值很高但完全读起来狗屁不通的译文（比如颠倒一下前后句）。此外BLEU的表现受参考译文数量制约。