基于seq2seq with attention model 的英中翻译

1.项目背景

语言是人类沟通的重要手段，有时候在阅读外语或者听外语时，可能会存在一些因语言不通而造成的理解问题。

在观看外语视频的时候，某些视频因为缺少字幕，而造成了理解上的困难。为此，我希望写一个视频的语音识别+机器翻译的程序来实现再看外文视频时能够自动加中文字幕的功能。

而机器翻译就是其中的一环，我的机器翻译学习从这里开始。

就目前而言，最新的主流的机器翻译方案就是transform和bert. 模型的发展是由 基本的seq2seq、seq2seq with attention、transform、bert 这四个发展阶段，他们的架构越来越复杂，参数数量也越来越多，但是这并不代表任何情况下后面的复杂的模型都比前面的简单的模型好，因为这是需要数据做为支撑，如果本身数据量很小，而去选择参数非常多的模型，也很难发挥其模型的实力。同时，后面的模型也是由前面的一步步改进而来，而非一步到位的，学习、理解前面的简单的模型，对于学习理解后面的模型也有很大的帮助。

2.解决思路

2.1 问题分析

机器翻译就是从文本到文本的一种典型的模型，我们接受一段文本，通过一个模型，产生输出一段文本，这个模型理论上来说可以是任意的，但是普通的RNN网络具有不能处理输入输出长度不相等的问题，而另一种RNN的变种:seq2seq model就可以用来解决这个问题。

这里我们采用seq2seq with attention的模型。

2.2 seq2seq with attention详解

2.2.1 我们为什么需要seq2seq

下图是传统的RNN结构

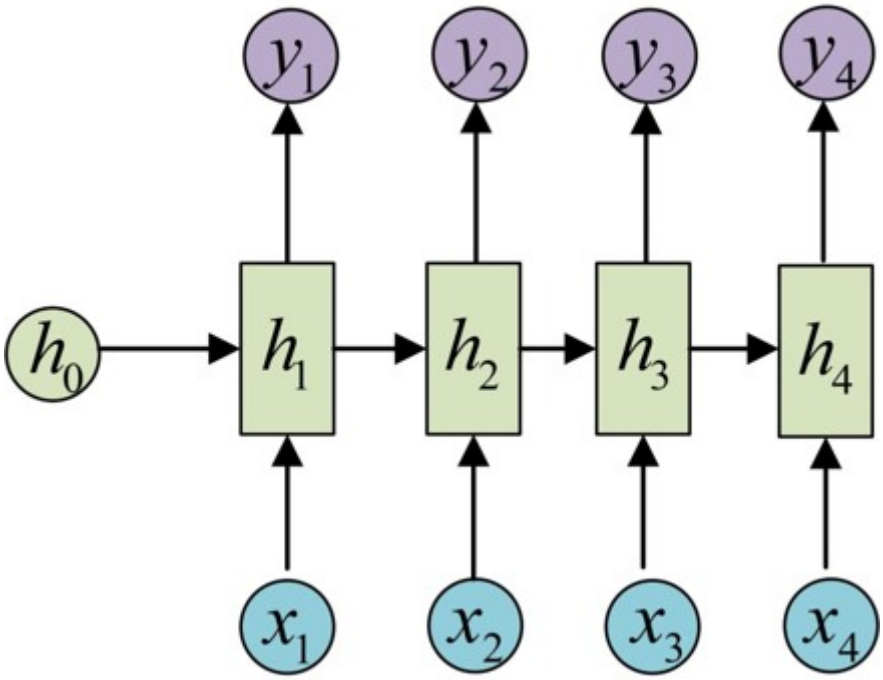

图1.传统RNN

其中

$h_i = f(Ux_i + Wh_{i-1} + b)$

$y_i = softmax(Vh_i + c)$

其参数U,W,V都是共享的

我们可以看到，Xi与Yi是一 一对应的，也就是说，传统的RNN的输入与输出必须是等长的,即N to N。

但这在一些条件下并不符合要求，比如在接下来的机器翻译中，源语言与目标语言的句子长度往往不相等，那么我们就需要一种输入与输出长度可以不相等的网络结构。

Seq2seq就是为解决这个问题而诞生的。

2.2.2 什么是seq2seq

简单来看，seq2seq的大体框架就是下面这样

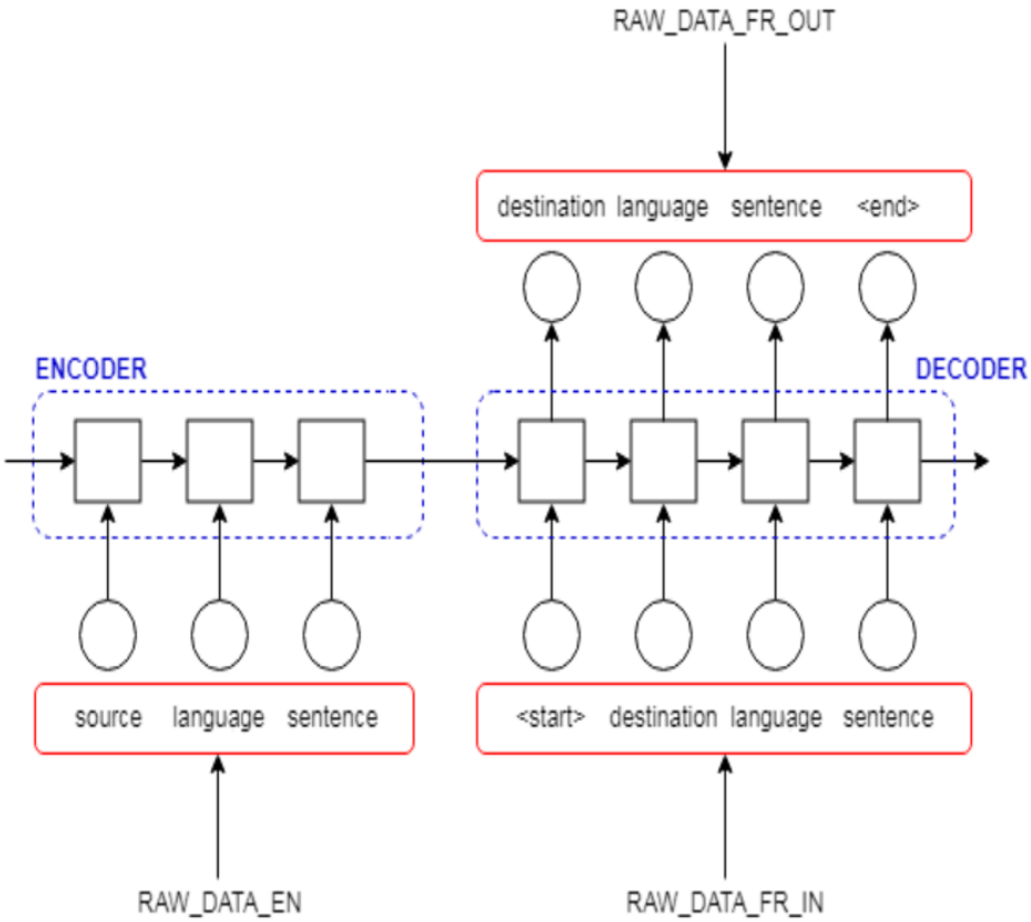

图2.seq2seq大体框架

下图展示了基本的seq2seq网络结构；

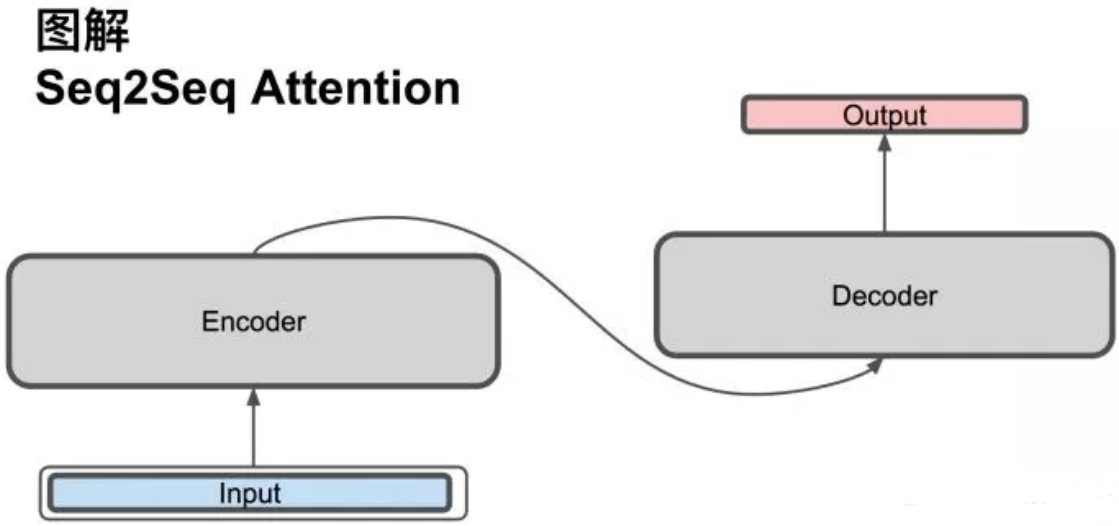

图3.seq2seq基本结构

实际上，seq2seq是RNN的一种变种网络，它是由两个RNN(左边的encoder和右边的decoder都是一个RNN)拼接而成。通过encode和decode网络来实现不定长序列之间的映射。实际实现中，RNN一般选用LSTM或者是GRU网络来一定程度上缓减长期记忆遗失的问题，但是这还不够，这也是后面attention机制被应用在seq2seq网络的原因。

虽然两个RNN各自的输入输出仍是相同。我们可以先通过encode网络生成input的语义向量，作为h0输入到decode中。

Decode网络的输入的长度即为整个网络的输出，与网络的输入的长度无关，这样就可以有一个长度的input到另一个长度的output的映射关系。就实现了N to M的映射。

我们可以看到，encode的隐藏层的输出都被丢掉，只有最后一层的输出被传给了decode，随着反复的向前传播、反向传播，input对output的影响会越来越小。这也是seq2seq网络的一个问题。

事实上，正因为Seq2seq是两个RNN的拼接，因此，基本的seq2seq也存在着基本RNN存在的问题----长期记忆的遗失（类似于梯度爆炸或消失）。
虽然在RNN中也有LSTM或GRU的解决思路，但是效果还不够好，于是，有人提出了带有attention机制的seq2seq网络解决这一问题。

2.2.3 带有注意力机制的seq2seq

下图展示了带有注意力机制的seq2seq网络结构

图4.摘自 Effective Approaches to Attention-based Neural Machine Translation

该图摘自2015年提出了Luong Attention的论文**Effective Approaches to Attention-based Neural Machine Translation**，这是作者用于说明该网络的一个示例。

但是我们直接看这个图，恐怕还是有点难看懂。因此，在看这个图之前，我需要说明一些预备知识。比较重要的内容有两个:

1.The alignment vector(即图中的 attention weights)

$$\boldsymbol{a}_t(s) = \text{align}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)$$
$$= \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)}$$

Equation 1: Equation for alignment vector

图5.The Alignment Vector

> The alignment vector是一个与输入源序列相同长度的的向量，它是在decoder的每个时间步里计算而来。
> 这个向量的每个值就是对应的源输入序列的score(或者说是可能性)。

那么alignment是如何产生的?
我们看下面两个计算公式

对于$a_t(s)$的计算，我相信如果比较敏感的话，就能一下子看出来，这实际上就是对score求softmax，也很好理解，就是将得分(权重)最高的选出来。

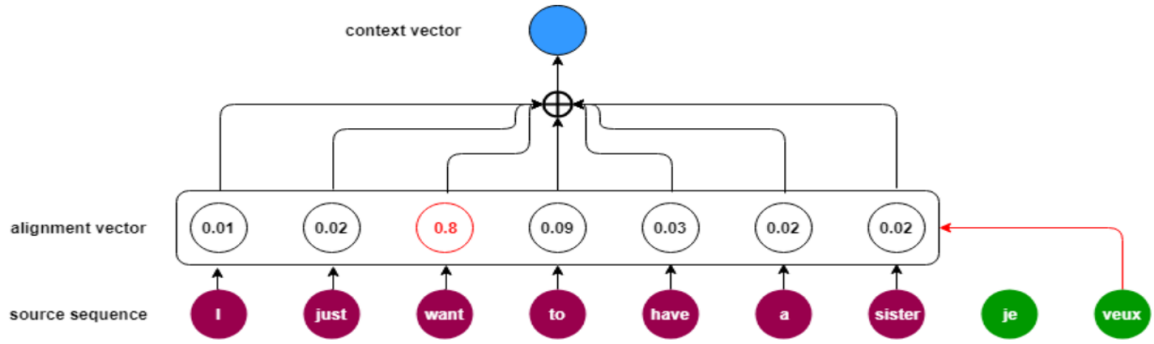

图6.The Alignment Vector计算方式

下图是Luong给出的三种计算score的方式。实际上除了Luong的attention之外，还有另一种attention:Bahdanau attention，不过他们大体上是很相似的，其中一处差别就是，Bahdanau attention 提倡只使用concat计算score而不使用其他的计算方式。

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_a [\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

Equation 2: Score functions

图7.score计算方式

2.The context vector(上下文向量)

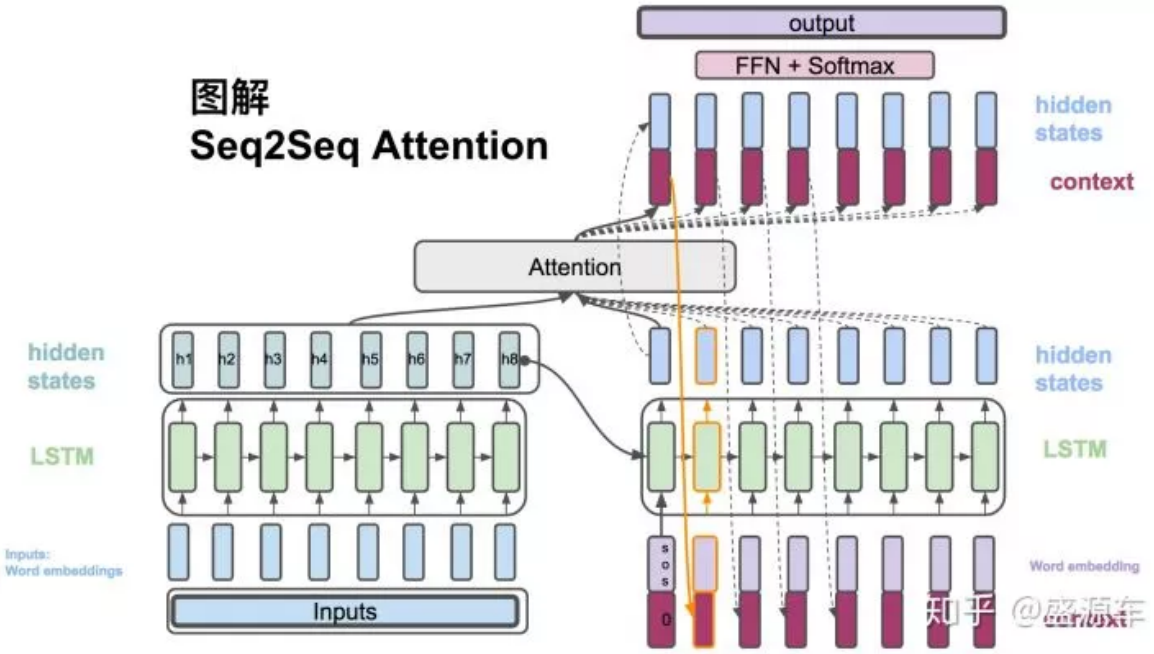

图8.The context vector

> The context vector 是我们用来计算最后的decoder的输出的，具体怎样计算，我会在后面给出说明。
>
> 它是encoder网络输出结果的加权平均。

其具体计算的方式即为 encode的输出与alignment vector做点积运算，即可得到context vector

$$c_i = \sum_{j=1}^{T_x} a_{ij} h_j$$

以上两个部分就是attention机制的核心，使用attention，简单来说，就是为了得到这个context vector。

当然如果没有attention，也可以认为有context vector，但是那个context vector只是基于ecoder的最后一个hidden state而与其他无关，同时这个context vector都是相同的，这代表所有的输入对于输出的影响是相同的，我们可以说这是没有注意到某些关键的元素的，效果并不好。attention则是通过加权平均，给予每个输入不同的权重，让机器从数据中学习，来获取某些关键的点，最后获得一个相对比较好的语义抽象--context vector.

2.2.4 seq2seq with attention网络全貌

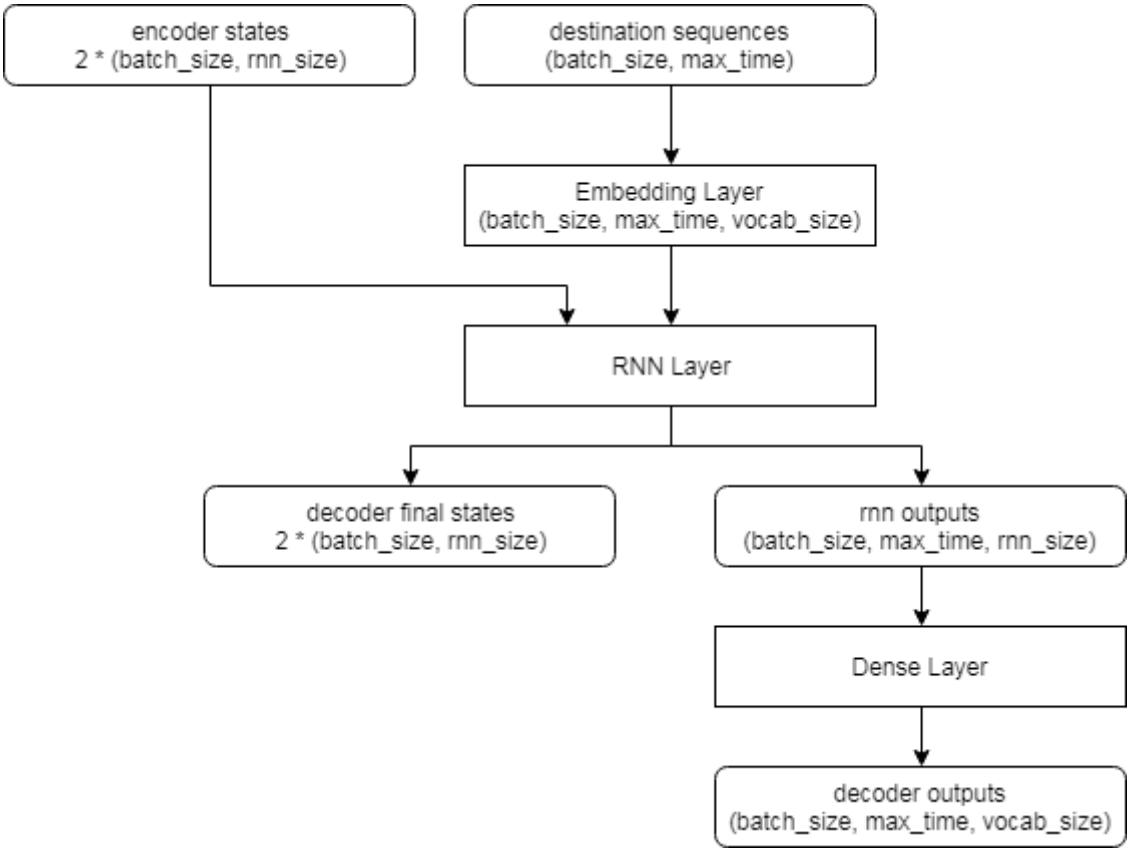

图9.seq2seq attention 整体架构

左侧为Encoder，右侧为Decoder，中间为Attention。

Encoder一般是由一层embedding层和多层RNN(一般选用LSTM或GRU)层组成，Decoder层也同样如此。

一个时间步的流程如下:

1. 从左边Encoder开始，输入转换为word embedding, 进入LSTM。LSTM会在每一个时间点上输出hidden states。如图中的h1,h2,...,h8。
2. 接下来进入右侧Decoder，输入为中文的句子，以及从encoder最后一个hidden state: h8。LSTM的是输出是一个hidden state (cell state不需要使用)。

3. 随后，Decoder的hidden state与Encoder所有的hidden states作为输入，放入Attention模块开始计算一个context vector，就是在2.2.3节我们提到的计算方法。

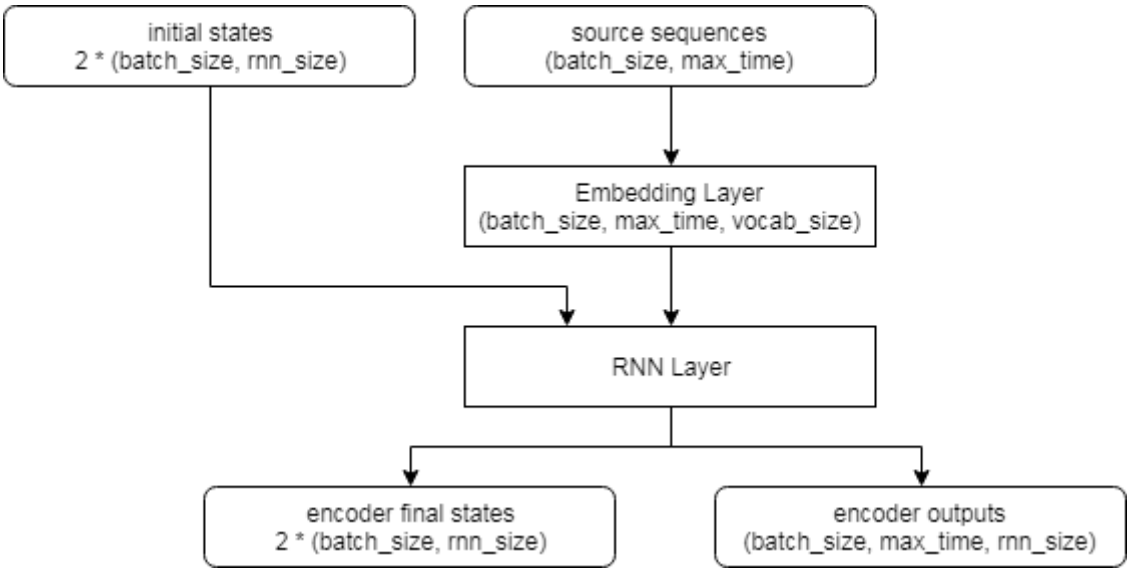

图10.seq2seq attention 第2个时间步

我们来到第2个时间步:之前的context vector可以作为输入和目标的单词串起来作为RNN（即LSTM）的输入。之后又回到一个hidden state。以此循环。也就是说，decoder每走过一时间步，就会在context vector产生一个值，并会与目标串连接作为下一个时间步decoder的输入，直到走到最后的时间步。

走完了所有的时间步后，context也同步地更新完成。这时候，$\hat{s}_t = tanh(W_c[c_t; s_t])$，即将context vector和decoder的hidden states连接；通过通过$p(y_t|y < t, x) = softmax(W_s\hat{s}_t)$来计算最后的输出概率

3.项目实现

3.1 数据处理

3.1.1数据集准备

点击这里下载数据集

这里我们选用的是'http://www.manythings.org/anki/'上的中英翻译的数据集，这个数据集是比较小的，只有20000条的数据，文件不到3MB，但是我们这次主要还是为了学习，所以并不是特别看重这个效果，要知道，谷歌训练自己的谷歌翻译的数据集都是TB级别的。

3.1.2 提取文本

```
def loadFile(filename=path_to_file):
    with open(filename, 'r', encoding='utf8') as f:
        raw_data = f.readlines()
    return raw_data
```

我们来看一下数据的格式是怎样的:

```
print(raw_data[:10])
```

输出结果:

```
['Hi.\t嗨。\n', 'Hi.\t你好。\n', 'Run.\t你用跑的。\n', 'Wait!\t等等! \n',
'Hello!\t你好。\n', 'I try.\t让我来。\n', 'I won!\t我赢了。\n', 'Oh no!\t不会吧。
\n', 'Cheers!\t乾杯!\n', 'He ran.\t他跑了。\n']
```

3.1.3 分词

我们要对英文和中文进行分词，如果是英文，不需要特别处理，直接按照空格分割即可；对于中文，我们通过分词工具jieba来实现。那么如何判断一个句子是英文还是中文？我们可以通过Unicode编码来判断，中文的Unicode编码是在4e00到9fff之间的，如果句子中出现了这个编码区间内的字符，即认为该句子是中文，并采用中文的处理方式。

```
def check_contain_chinese(w):
    flag = True
    flagtop=False
    for check_str in w:
        for ch in check_str:
            if u'\u4e00' >= ch or ch >= u'\u9fff':
                flag =  False
                flagtop=False
                break
        if not flagtop :
                break
    return flag

def preprocess_chinese(w):
    line=jieba.cut(w)
    w=[x for x in line]
    item=' '.join(w)
    return item

def preprocess_sentence(w):
    w = re.sub(r"([?.!,¿])", r" \1 ", w)#在单词与跟在其后的标点符号之间插入一个空格
    w = re.sub(r'[" "]+', " ", w)#多个空格合并为一个空格
    if check_contain_chinese(w):
        w=preprocess_chinese(w)
    else:
        w = w.lower().strip()#因为大小写是不影响含义的，我们不能说大写的ME与小写的me
的意思不同，                                    #因此全部转化为小写
    w = '<start> ' + w + ' <end>'
```

3.1.4 提取中英文

```
def create_dataset(path, num_examples):
    lines = io.open(path, encoding='UTF-8').read().strip().split('\n')

    word_pairs = [[preprocess_sentence(w) for w in l.split('\t')]  for l in
lines[:num_examples]]

    return zip(*word_pairs)
```

我们看一下现在English和Chinese的最后一个结果是怎样的:

```
en,ch = create_dataset(path_to_file, None)
print(en[-1])
print(ch[-1])
```

输出:

```
<start> if a person has not had a chance to acquire his target language by the time he's an adult , he's unlikely to be able to reach native speaker level in that language . <end>
<start> 如果 一個 人 在 成人 前 沒 有 機會習 得 目標 語言 ， 他 對 該 語言 的 認識 達到 母語者 程度 的 機會 是 相當 小 的 。 <end>
```

3.1.5文本序列化

我们通过TensorFlow.keras中的Tokenizer类来帮助我们完成这一工作，这里 `fit_on_texts` 用来对输入的文本产生一个字典，这个字典是一个文字到整数的映射;`texts_to_sequences` 将输入的文本按照刚刚生成的字典将文本的文字转化为一个个数字，生成整数的tensor； `pad_sequences` 这里的 `padding='post'` 是让刚刚产生的所有tensor扩展为相同的长度(即原本最长的tensor的长度)

```
def tokenize(lang):
    lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(
      filters='')
    lang_tokenizer.fit_on_texts(lang)

    tensor = lang_tokenizer.texts_to_sequences(lang)

    tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor,
                                                           padding='post')

    return tensor, lang_tokenizer
```

3.1.6加载清理好的数据集

```
def load_dataset(path, num_examples=None):
    # 创建清理过的输入输出对
    inp_lang, targ_lang = create_dataset(path, num_examples)

    input_tensor, inp_lang_tokenizer = tokenize(inp_lang)
    target_tensor, targ_lang_tokenizer = tokenize(targ_lang)

    return input_tensor, target_tensor, inp_lang_tokenizer,
targ_lang_tokenizer
```

```
# 尝试实验不同大小的数据集
num_examples = 20133
input_tensor, target_tensor, inp_lang, targ_lang = load_dataset(path_to_file,
num_examples)

# 计算目标张量的最大长度 （max_length）
max_length_targ, max_length_inp = max_length(target_tensor),
max_length(input_tensor)
```

3.1.7划分训练集与验证集

```
# 采用 80 - 20 的比例切分训练集和验证集
input_tensor_train, input_tensor_val, target_tensor_train, target_tensor_val
= train_test_split(input_tensor, target_tensor, test_size=0.2)
```

3.1.8设置训练结构

这里可以修改每个buffer的大小、每个batch的大小、embedding层的维度，和输入的unit数量

```
BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
steps_per_epoch = len(input_tensor_train)//BATCH_SIZE
embedding_dim = 256
units = 1024
vocab_inp_size = len(inp_lang.word_index)+1
vocab_tar_size = len(targ_lang.word_index)+1
```

3.1.9处理dataset

我们通过`tf.data.Dataset`创建TensorFlow的数据集格式，并打乱、按batch切分数据集

```
dataset = tf.data.Dataset.from_tensor_slices((input_tensor_train,
target_tensor_train)).shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)
```

3.2网络结构

3.2.1 Encoder层

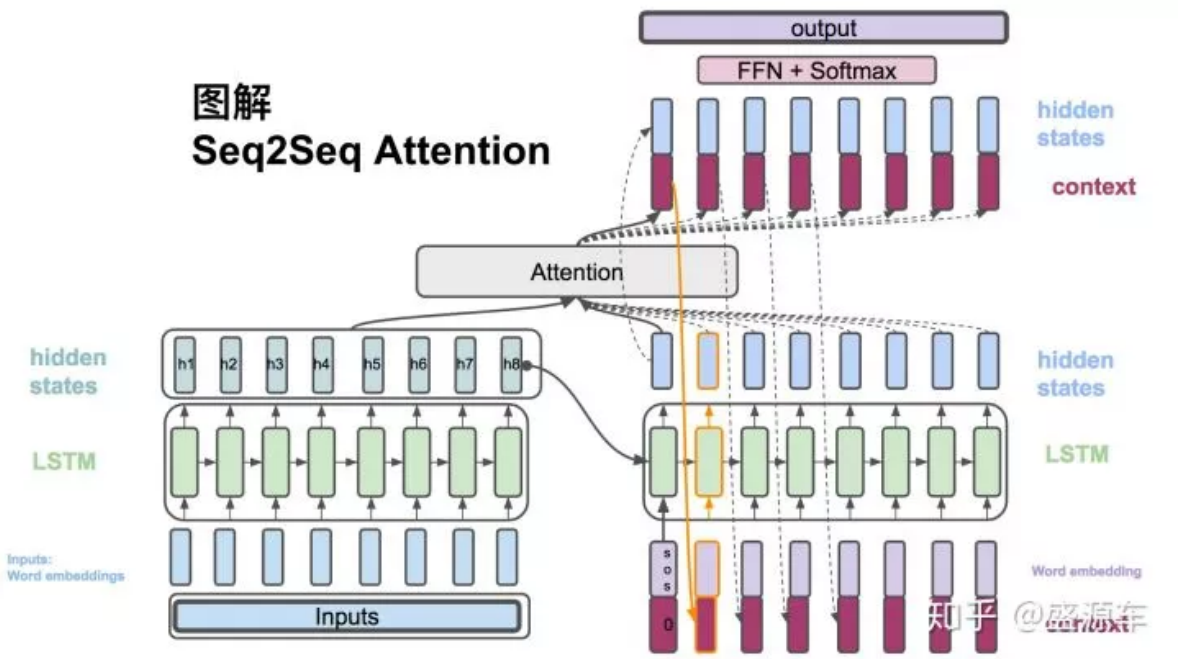

图11.encoder网络结构

Encoder由一层embedding层以及一层或多层RNN组成，这在前面已经有过说明。这里，为了减少运算量，我们的RNN选用一层GRU。

```
class Encoder(tf.keras.Model):
```

```
    def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
        super(Encoder, self).__init__()
        self.batch_sz = batch_sz
        self.enc_units = enc_units
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = tf.keras.layers.GRU(self.enc_units,
                                       return_sequences=True,
                                       return_state=True,

recurrent_initializer='glorot_uniform')

    def call(self, x, hidden):
        x = self.embedding(x)
        output, state = self.gru(x, initial_state = hidden)
        return output, state

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))
```

3.2.2 Attention层

```
class BahdanauAttention(tf.keras.layers.Layer):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, query, values):
        # 隐藏层的形状 == （批大小，隐藏层大小）
        # hidden_with_time_axis 的形状 == （批大小，1，隐藏层大小）
        # 这样做是为了执行加法以计算分数
        hidden_with_time_axis = tf.expand_dims(query, 1)

        # 分数的形状 == （批大小，最大长度，1）
        # 我们在最后一个轴上得到 1， 因为我们把分数应用于 self.V
        # 在应用 self.V 之前，张量的形状是（批大小，最大长度，单位）
        score = self.V(tf.nn.tanh(
            self.W1(values) + self.W2(hidden_with_time_axis)))

        # 注意力权重 （attention_weights） 的形状 == （批大小，最大长度，1）
        attention_weights = tf.nn.softmax(score, axis=1)

        # 上下文向量 （context_vector） 求和之后的形状 == （批大小，隐藏层大小）
        context_vector = attention_weights * values
        context_vector = tf.reduce_sum(context_vector, axis=1)

        return context_vector, attention_weights
```

我们创建attention层,设置输入单元为10

```
attention_layer = BahdanauAttention(10)
attention_result, attention_weights = attention_layer(sample_hidden,
sample_output)
```

3.2.3 Decoder层

```
translate(u" we  like it")
```

```
<generator object evaluate.<locals>.<genexpr> at 0x0000022102D390C8>
Input: <start> we like it <end>
Predicted translation: 我們 喜歡 它 。 <end>
```


图12.Decoder网络结构

一个Decoder层由一层embedding层、多层RNN(LSTM/GRU)、一个全连接层(dense)、attention层、一个softmax层组成。

```
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units, batch_sz):
        super(Decoder, self).__init__()
        self.batch_sz = batch_sz
        self.dec_units = dec_units
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = tf.keras.layers.GRU(self.dec_units,
                                       return_sequences=True,
                                       return_state=True,

recurrent_initializer='glorot_uniform')
        self.fc = tf.keras.layers.Dense(vocab_size)

        # 用于注意力
        self.attention = BahdanauAttention(self.dec_units)

    def call(self, x, hidden, enc_output):
        # 编码器输出 （enc_output） 的形状 == （批大小，最大长度，隐藏层大小）
        context_vector, attention_weights = self.attention(hidden,
enc_output)

        # x 在通过嵌入层后的形状 == （批大小，1，嵌入维度）
        x = self.embedding(x)

        # x 在拼接 （concatenation） 后的形状 == （批大小，1，嵌入维度 + 隐藏层大
小）
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

        # 将合并后的向量传送到 GRU
        output, state = self.gru(x)

        # 输出的形状 == （批大小 * 1，隐藏层大小）
        output = tf.reshape(output, (-1, output.shape[2]))

        # 输出的形状 == （批大小，vocab）
        x = self.fc(output)

        return x, state, attention_weights
```

像前面一样，我们创建decoder层

```
decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)
```

至此，我们的网络结构的三个大组件，就像搭乐高积木那样，已经搭建完毕。

3.3 优化器与损失函数

这里我们直接通过tensorflow提供的接口，使用Adam优化器和均方差的损失函数

```
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_mean(loss_)
```

3.4训练与评价

定义训练函数:基本就是将输入传入encoder和decoder，计算损失，梯度下降

在训练的过程中，我们使用了教师强制，你可能会问了，什么是教师强制?

“教师强制”的概念是使用实际目标输出作为每个下一个输入，而不是使用解码器的猜测作为下一个输入。 使用教师强制会导致其收敛更快。用一种比较直观的方式解释的话，那就是，最开始的时候，网络还什么都不会，这时候第一个时间步训练时，它的输出可能都是错的，那么我们拿一个错的输出作为下一个输入，显然也难以得到正确的结果。那么我们通过使用"教师强制"，将正确答案作为输入而不是错的答案，就可以帮助这个网络更快地学到这些潜在的东西。但并不是任何时候都要用"教师强制"，这里一篇文章表明，当使用受过训练的网络时，可能会显示不稳定。

概括一下，就是对于完全没有训练过的网络，使用“教师强制”会帮助我们加快训练速度，但是如果这个网络已经训练地比较多了，那么我们就不该再使用“教师强制”。

```

@tf.function
def train_step(inp, targ, enc_hidden):
    loss = 0

    with tf.GradientTape() as tape:
        enc_output, enc_hidden = encoder(inp, enc_hidden)

        dec_hidden = enc_hidden

        dec_input = tf.expand_dims([targ_lang.word_index['<start>']] *
BATCH_SIZE, 1)

        # 教师强制 - 将目标词作为下一个输入
        for t in range(1, targ.shape[1]):
          # 将编码器输出 （enc_output） 传送至解码器
```

```
            predictions, dec_hidden, _ = decoder(dec_input, dec_hidden, enc_output)

            loss += loss_function(targ[:, t], predictions)

            # 使用教师强制
            dec_input = tf.expand_dims(targ[:, t], 1)

    batch_loss = (loss / int(targ.shape[1]))

    variables = encoder.trainable_variables + decoder.trainable_variables

    gradients = tape.gradient(loss, variables)

    optimizer.apply_gradients(zip(gradients, variables))

    return batch_loss


```

我们训练10个epoch，每2个epoch保存一次模型，这样在下次训练时无需从头开始，接着上一次训练时保存的epoch接着训练即可。

```

EPOCHS = 10

for epoch in range(EPOCHS):
    start = time.time()

    enc_hidden = encoder.initialize_hidden_state()
    total_loss = 0

    for (batch, (inp, targ)) in tqdm(enumerate(dataset.take(steps_per_epoch))):
        batch_loss = train_step(inp, targ, enc_hidden)
        total_loss += batch_loss

        if batch % 100 == 0:
            print('Epoch {} Batch {} Loss {:.4f}'.format(epoch + 1,
                                                         batch,
                                                         batch_loss.numpy()))
  # 每 2 个周期（epoch），保存（检查点）一次模型
    if (epoch + 1) % 2 == 0:
        checkpoint.save(file_prefix = checkpoint_prefix)

    print('Epoch {} Loss {:.4f}'.format(epoch + 1,
                                        total_loss / steps_per_epoch))
    print('Time taken for 1 epoch {} sec\n'.format(time.time() - start))


```

评价函数基本上与训练是一致的，只是这里不需要使用教师强制

```
def evaluate(sentence):
    attention_plot = np.zeros((max_length_targ, max_length_inp))

```

```
    sentence = preprocess_sentence(sentence)
    print(i for i in sentence.split(" "))
    inputs = [inp_lang.word_index[i]
              for i in sentence.split(' ')]
    inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs],

maxlen=max_length_inp,
                                                           padding='post')
    inputs = tf.convert_to_tensor(inputs)

    result = ''

    hidden = [tf.zeros((1, units))]
    enc_out, enc_hidden = encoder(inputs, hidden)

    dec_hidden = enc_hidden
    dec_input = tf.expand_dims([targ_lang.word_index['<start>']], 0)

    for t in range(max_length_targ):
        predictions, dec_hidden, attention_weights = decoder(dec_input,
                                                             dec_hidden,
                                                             enc_out)

        # 存储注意力权重以便后面制图
        attention_weights = tf.reshape(attention_weights, (-1, ))
        attention_plot[t] = attention_weights.numpy()

        predicted_id = tf.argmax(predictions[0]).numpy()

        result += targ_lang.index_word[predicted_id] + ' '

        if targ_lang.index_word[predicted_id] == '<end>':
            return result, sentence, attention_plot

        # 预测的 ID 被输送回模型
        dec_input = tf.expand_dims([predicted_id], 0)

    return result, sentence, attention_plot
```

3.5 训练结果测试

我们定义一个translate方法来实现翻译:

```
def translate(sentence):
    result, sentence, attention_plot = evaluate(sentence)

    print('Input: %s' % (sentence))
    print('Predicted translation: {}'.format(result))
```

```
translate(u" i need you")
```

```
<generator object evaluate.<locals>.<genexpr> at 0x000002214C87E7C8>
Input: <start> i need you <end>
Predicted translation: 我 需要 你 。 <end>
```

图13.正确的测试-1

```
translate(u" be quiet")
```

```
<generator object evaluate.<locals>.<genexpr> at 0x000002210092B548>
Input: <start> be quiet <end>
Predicted translation: 保持 安静 ！ <end>
```

图14.正确的测试-2

```
translate(u" i will go there")
```

```
<generator object evaluate.<locals>.<genexpr> at 0x000002210092B548>
Input: <start> i will go there <end>
Predicted translation: 我 明天 要 去 那裡 。 <end>
```

图15.正确的测试-3

```
translate(u"we")
```

```
<generator object evaluate.<locals>.<genexpr> at 0x000002214F157A48>
Input: <start> we <end>
Predicted translation: 我們 很 合作 。 <end>
```

图16.正确的测试-4

```
translate(u" we believe we will make a better future")
```

```
<generator object evaluate.<locals>.<genexpr> at 0x0000022153821A48>
Input: <start> we believe we will make a better future <end>
Predicted translation: 我們 不想 再 跟 我們 不高興 。 <end>
```

图17.错误的测试-1

```
translate(u"please be quiet")
```

```
<generator object evaluate.<locals>.<genexpr> at 0x000002210092B548>
Input: <start> please be quiet <end>
Predicted translation: 请 敲门 。 <end>
```

图18.错误的测试-2

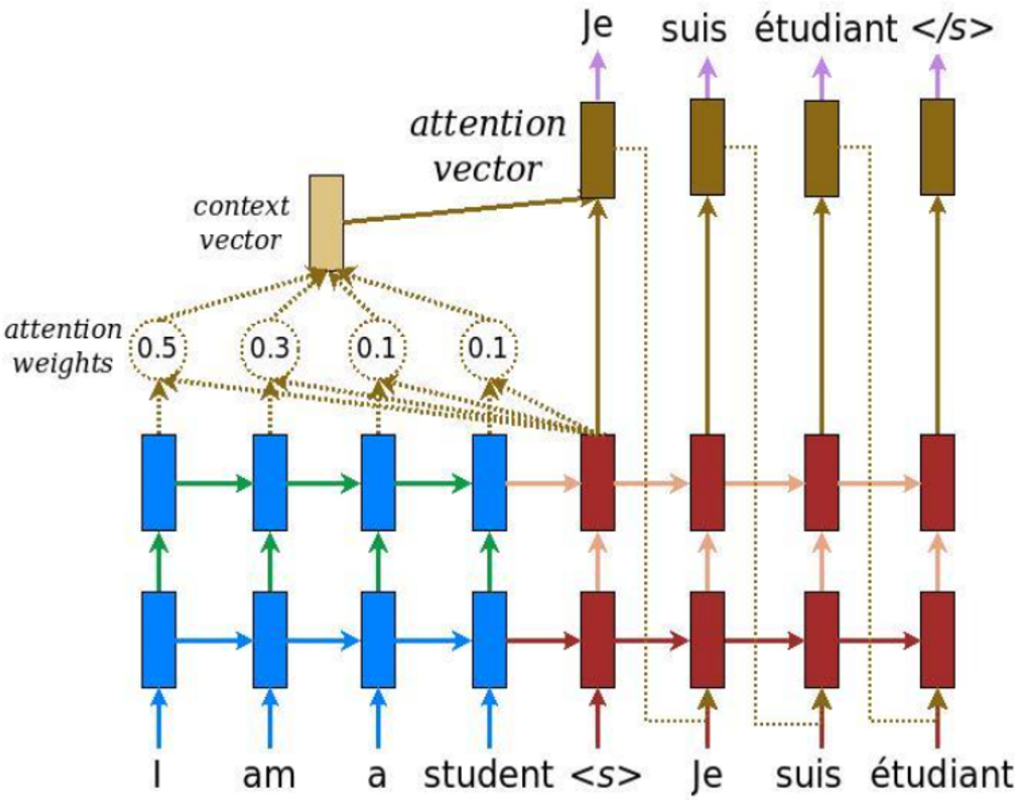

图19.错误的测试-3

我们可以看到，对于一些比较简单的、不是太长也不是只有单个词的时候，还是有一定的翻译效果，这说明其也确实学到了一些东西；但是显然，这个还不够准确，对于单个的词和比较长的句子的翻译效果很差。不过，我认为这也与我使用的数据集很小有关。

以上，就是本次 seq2seq with attention 英中翻译的全部内容。