In [17]:

```
#导入可能用到的包或模块
import collections
import io
import math
from mxnet import autograd, gluon, init, nd
from mxnet.contrib import text
from mxnet.gluon import data as gdata, loss as gloss, nn, rnn
```

In [18]:

```
# 将一个序列中所有的词记录在all_tokens中以便之后构造词典，然后在该序列后面添加PAD直到序列
# 长度变为max_seq_len，然后将序列保存在all_seqs中
PAD, BOS, EOS = '<pad>', '<bos>', '<eos>'

def process_one_seq(seq_tokens, all_tokens, all_seqs, max_seq_len):
    all_tokens.extend(seq_tokens)
    seq_tokens += [EOS] + [PAD] * (max_seq_len - len(seq_tokens) - 1)
    all_seqs.append(seq_tokens)

# 使用所有的词来构造词典。并将所有序列中的词变换为词索引后构造NDArray实例
def build_data(all_tokens, all_seqs):
    vocab = text.vocab.Vocabulary(collections.Counter(all_tokens),
                                  reserved_tokens=[PAD, BOS, EOS])
    indices = [vocab.to_indices(seq) for seq in all_seqs]
    return vocab, nd.array(indices)
```

In [19]:

```
# 在读取数据时，在句末附上“<eos>”符号，并可能通过添加“<pad>”符号使每个序列的长度均为max_seq_le
# 现在为法语词和英语词分别创建词典。法语词的索引和英语词的索引相互独立。

def read_data(max_seq_len):
    # in和out分别是input和output的缩写
    in_tokens, out_tokens, in_seqs, out_seqs = [], [], [], []
    with io.open('/home/lzj/桌面/fr-en-small.txt') as f:
        lines = f.readlines()
    for line in lines:
        in_seq, out_seq = line.rstrip().split('\t')
        in_seq_tokens, out_seq_tokens = in_seq.split(' '), out_seq.split(' ')
        if max(len(in_seq_tokens), len(out_seq_tokens)) > max_seq_len - 1:
            continue  # 如果加上EOS后长于max_seq_len，则忽略掉此样本
        process_one_seq(in_seq_tokens, in_tokens, in_seqs, max_seq_len)
        process_one_seq(out_seq_tokens, out_tokens, out_seqs, max_seq_len)
    in_vocab, in_data = build_data(in_tokens, in_seqs)
    out_vocab, out_data = build_data(out_tokens, out_seqs)
    return in_vocab, out_vocab, gdata.ArrayDataset(in_data, out_data)

# 将序列的最大长度设成7
max_seq_len = 7
in_vocab, out_vocab, dataset = read_data(max_seq_len)
```

In [20]:

```
#用含注意力机制的编码器—解码器来将一段简短的法语翻译成英语
# 编码器：在编码器中，我们将输入语言的词索引通过词嵌入层得到词的表征，然后输入到一个多层门控循环单元
#在前向计算后也会分别返回输出和最终时间步的多层隐藏状态

class Encoder(nn.Block):# 编码器定义
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 drop_prob=0, **kwargs):
        super(Encoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size) #词嵌入层
        self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob) #GRU门控循环单

    def forward(self, inputs, state): #前向计算
        # 输入形状是(批量大小, 时间步数)。将输出互换样本维和时间步维
        embedding = self.embedding(inputs).swapaxes(0, 1)
        return self.rnn(embedding, state)

    def begin_state(self, *args, **kwargs):#初始隐藏状态
        return self.rnn.begin_state(*args, **kwargs)



# 注意力机制
#将输入连结后通过含单隐藏层的多层感知机变换。其中隐藏层的输入是解码器的隐藏状态与编码器在所有时间步上
#并且使用tanh函数作为激活函数。输出层的输出个数为1，两个Dense实例均不使用偏差；

def attention_model(attention_size):
    model = nn.Sequential()
    model.add(nn.Dense(attention_size, activation='tanh', use_bias=False,
                       flatten=False),
              nn.Dense(1, use_bias=False, flatten=False))
    return model



# 注意力机制的输入包括查询项、键项和值项;
# 设编码器和解码器的隐藏单元个数相同。这里的查询项为解码器在上一时间步的隐藏状态，形状为(批量大小,
# 键项和值项均为编码器在所有时间步的隐藏状态，形状为(时间步数, 批量大小, 隐藏单元个数)。
# 注意力机制返回当前时间步的背景变量，形状为(批量大小, 隐藏单元个数)。


def attention_forward(model, enc_states, dec_state):
    # 将解码器隐藏状态广播到和编码器隐藏状态形状相同后进行连结
    dec_states = nd.broadcast_axis(
        dec_state.expand_dims(0), axis=0, size=enc_states.shape[0])
    enc_and_dec_states = nd.concat(enc_states, dec_states, dim=2)
    e = model(enc_and_dec_states)  # 形状为(时间步数, 批量大小, 1)
    alpha = nd.softmax(e, axis=0)  # 在时间步维度做softmax运算
    return (alpha * enc_states).sum(axis=0)  # 返回背景变量
```

In [21]:

```
# 含注意力机制的解码器
# 编码器在最终时间步的隐藏状态作为解码器的初始隐藏状态
# 在解码器的前向计算中，先通过刚刚介绍的注意力机制计算得到当前时间步的背景向量。
# 由于解码器的输入来自输出语言的词索引，将输入通过词嵌入层得到表征，然后和背景向量在特征维连结。
# 再将连结后的结果与上一时间步的隐藏状态通过门控循环单元计算出当前时间步的输出与隐藏状态。
# 最后，最后将输出通过全连接层变换为有关各个输出词的预测，形状为(批量大小， 输出词典大小)。

class Decoder(nn.Block):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 attention_size, drop_prob=0, **kwargs):
        super(Decoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)#词嵌入层
        self.attention = attention_model(attention_size)#注意力模型，用于计算背景变量
        self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob)
        self.out = nn.Dense(vocab_size, flatten=False)# 全连接层，用于类别输出

    def forward(self, cur_input, state, enc_states):
        # 使用注意力机制计算背景向量
        c = attention_forward(self.attention, enc_states, state[0][-1])
        # 将嵌入后的输入和背景向量在特征维连结
        input_and_c = nd.concat(self.embedding(cur_input), c, dim=1)
        # 为输入和背景向量的连结增加时间步维，时间步个数为1
        output, state = self.rnn(input_and_c.expand_dims(0), state)
        # 移除时间步维，输出形状为(批量大小， 输出词典大小)
        output = self.out(output).squeeze(axis=0)
        return output, state

    def begin_state(self, enc_state):
        # 直接将编码器最终时间步的隐藏状态作为解码器的初始隐藏状态
        return enc_state
```

In [22]:

```
# 训练模型
# 定义 batch_loss函数用来计算一个小批量的损失。
# 解码器在最初时间步的输入是特殊字符 "BOS"， 之后，输入运用即强制教学的方法， 即解码器在某时间步的

def batch_loss(encoder, decoder, X, Y, loss):
    batch_size = X.shape[0]
    enc_state = encoder.begin_state(batch_size=batch_size)
    enc_outputs, enc_state = encoder(X, enc_state)
    # 初始化解码器的隐藏状态
    dec_state = decoder.begin_state(enc_state)
    # 解码器在最初时间步的输入是BOS
    dec_input = nd.array([out_vocab.token_to_idx[BOS]] * batch_size)
    # 我们将使用掩码变量mask来忽略掉标签为填充项PAD的损失
    mask, num_not_pad_tokens = nd.ones(shape=(batch_size,)), 0
    l = nd.array([0])
    for y in Y.T:
        dec_output, dec_state = decoder(dec_input, dec_state, enc_outputs)
        l = l + (mask * loss(dec_output, y)).sum()
        dec_input = y  # 使用强制教学
        num_not_pad_tokens += mask.sum().asscalar()
        # 当遇到EOS时，序列后面的词将均为PAD，相应位置的掩码设成0
        mask = mask * (y != out_vocab.token_to_idx[EOS])
    return l / num_not_pad_tokens

# 训练函数中，需要同时迭代编码器和解码器的模型参数
def train(encoder, decoder, dataset, lr, batch_size, num_epochs):
    encoder.initialize(init.Xavier(), force_reinit=True) # 编码器模型初始化
    decoder.initialize(init.Xavier(), force_reinit=True) # 解码器模型初始化
    enc_trainer = gluon.Trainer(encoder.collect_params(), 'adam',
                                {'learning_rate': lr}) # 参数训练器
    dec_trainer = gluon.Trainer(decoder.collect_params(), 'adam',
                                {'learning_rate': lr})
    loss = gloss.SoftmaxCrossEntropyLoss() # 交叉商损失函数
    data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
    for epoch in range(num_epochs): # 训练周期数
        l_sum = 0.0
        for X, Y in data_iter: # 每个小批量数据读取
            with autograd.record():
                l = batch_loss(encoder, decoder, X, Y, loss)
            l.backward() # 反向传播， 计算和存储中间变量和模型参数梯度;
            enc_trainer.step(1) # 参数更新
            dec_trainer.step(1)
            l_sum += l.asscalar() # 损失统计
        if (epoch + 1) % 10 == 0:
            print("epoch %d, loss %.3f" % (epoch + 1, l_sum / len(data_iter)))
```

In [23]:

```
# 创建模型实例并设置超参数。然后就可以训练模型了,超参数未验证，并非最优;

embed_size, num_hiddens, num_layers = 64, 64, 2 # 超参数设定
attention_size, drop_prob, lr, batch_size, num_epochs = 10, 0.5, 0.01, 2, 100 # 超参
encoder = Encoder(len(in_vocab), embed_size, num_hiddens, num_layers,
                  drop_prob) #编码器
decoder = Decoder(len(out_vocab), embed_size, num_hiddens, num_layers,
                  attention_size, drop_prob) # 解码器
train(encoder, decoder, dataset, lr, batch_size, num_epochs) # 当前超参数下， 开始训练
```

```
epoch 10, loss 0.408
epoch 20, loss 0.178
epoch 30, loss 0.081
epoch 40, loss 0.027
epoch 50, loss 0.005
epoch 60, loss 0.003
epoch 70, loss 0.002
epoch 80, loss 0.002
epoch 90, loss 0.001
epoch 100, loss 0.001
```

In [26]:

```
# 利用贪婪搜索， 预测不定长序列

def translate(encoder, decoder, input_seq, max_seq_len):
    in_tokens = input_seq.split(' ')
    in_tokens += [EOS] + [PAD] * (max_seq_len - len(in_tokens) - 1)
    enc_input = nd.array([in_vocab.to_indices(in_tokens)]) # 词转索引
    enc_state = encoder.begin_state(batch_size=1) #编码器初始隐藏状态
    enc_output, enc_state = encoder(enc_input, enc_state) # 编码器前向计算， 输出 和 隐
    dec_input = nd.array([out_vocab.token_to_idx[BOS]]) # 解码器输入
    dec_state = decoder.begin_state(enc_state) # 解码器初始时刻隐藏状态
    output_tokens = []
    for _ in range(max_seq_len):
        dec_output, dec_state = decoder(dec_input, dec_state, enc_output) # 解码器 前
        pred = dec_output.argmax(axis=1) # 预测
        pred_token = out_vocab.idx_to_token[int(pred.asscalar())] # 索引 转 词语
        if pred_token == EOS:  # 当任一时间步搜索出EOS时，输出序列即完成
            break
        else:
            output_tokens.append(pred_token)
            dec_input = pred  # 将当前时刻的输出作为下一时刻的输入
    return output_tokens



# 测试一下模型。输入法语句子“ils regardent.”，翻译后的英语句子应该是“they are watching.”。
input_seq = 'ils regardent .'
translate(encoder, decoder, input_seq, max_seq_len)
```

Out[26]:

```
['they', 'are', 'watching', '.']
```

In [27]:

```
# 使用 bleu 评价翻译结果, 定义实现

def bleu(pred_tokens, label_tokens, k):
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, k + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[''.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[''.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[''.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score


# 定义一个辅助打印函数
def score(input_seq, label_seq, k):
    pred_tokens = translate(encoder, decoder, input_seq, max_seq_len)
    label_tokens = label_seq.split(' ')
    print('bleu %.3f, predict: %s' % (bleu(pred_tokens, label_tokens, k),
                                      ' '.join(pred_tokens)))
```

In [28]:

```
score('ils sont canadiens .', 'they are canadian .', k=2)
```

```
bleu 0.658, predict: they are actors .
```

In [29]:

```
score('ils regardent .', 'they are watching .', k=2)
```

```
bleu 1.000, predict: they are watching .
```

In [ ]:

```

```