第 11 章
机器翻译

软件国际化（118n）可以用机器翻译技术翻译一些术语。电影字幕需要翻译。图书需要翻译。药品说明书需要翻译。一些大型国际会议需要机器翻译技术提供更好的同声传译。

11.1 语言检测

安装 langid 模块：

```
#pip3 install langid
```

通过命令行使用 langid 判断文本的语言类型：

```
#langid
>>> This is a test
('en', -54.41310358047485)
>>> Questa e una prova
('it', -35.41771221160889)
```

langid 也可以检测重定向的输入。

```
#langid <README.md
('en', -850.338193655014)
```

返回的值是语言的非标准化概率估计值。默认情况下，禁用计算精确概率估计值，但可以通过标志启用：

```
# langid -n <README.md
('en', 1.0)
```

11.2 信道模型

对于英译中来说，假设英文是一个中国人写的，中文经过噪音处理后成为英文了。对于中

译英来说，假设中文是一个英国人写的，英文经过噪音处理后成为中文了。

将句子 e 翻译成句子 c 的概率。

$$c=\underset{c}{\operatorname{argmax}}\, P(c)P(e|c)$$

将翻译任务分解成一个翻译模型和一个语言模型。

假设词是按顺序翻译的。则：

$$P(e|c)=\prod P\ (\text{英文词}|\text{中文词})$$

翻译模型中用到了词的翻译概率：P(英文词|中文词)。例如，在翻译句子“Although north wind howls , but sky still very clear.”时，要用到翻译概率：

P(despite|虽然)

P (however|虽然)

P (although|虽然)

P (northern|北)

P (north|北)

P (however|虽然)的计算公式：

$$P(\text{however}\,|\,\text{虽然})=\frac{\text{“虽然”对齐成“however”的次数}}{\text{“虽然”出现的次数}}$$

为了能够持续改进，翻译模型文件和语言模型文件都必须是人工可以很方便地编辑和修改的。

翻译模型反映是否忠实原文。语言模型使输出流畅。为了提高准确性，需要建立个性化的翻译模型，避免把“Sun”公司翻译成“太阳”。

简单的翻译模型可以不考虑上下文而只考虑单词之间的翻译概率，也就是说原文的基本意思是否准确表达出来了。语言模型一般采用 N 元模型。

从可能的译文中求出最佳译文，称为解码。首先要能够生成出正确的译文，这样解码才有意义。

去标记（Detokenization）是标记化的逆过程：在需要时删除标记之间的空格。例如，句子末尾的单词和句点（两者都是单独的标记）之间不应有空格。

```
Tokenized: Hello world !
Detokenized: Hello world!
```

去标记通常是机器翻译管道的最后一步。Sacremoses(https://github.com/alvations/sacremoses)包括去标记的实现。

11.3　词表

神经机器翻译（NMT）模型通常使用固定的词表进行操作，但翻译是一个开放式词表问题。

一个方法是通过回退到字典来解决未登录词的翻译问题。另外一种方法是将罕见和未知单词编码为子单元序列来进行开放式词汇翻译。

基于直觉，各种单词类可以通过比单词更小的单位进行翻译，例如名字可以通过字符复制翻译或音译，化合物可以通过成分翻译，同源词和借词（外来词）可以通过语音和形态变换翻译。

在训练过程中，Byte Pair Encoding(BPE)首先将词分成一个一个的字符，然后统计字符对出现的次数，每次将次数最多的字符对保存起来，直到循环次数结束。

```
import re
import sys
import collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram_pattern = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram_pattern + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w</w>' : 5, 'l o w e r</w>' : 2,
        'n e w e s t</w>' : 6, 'w i d e s t</w>' : 3}
num_merges = 15
for i in range(num_merges):
  pairs = get_stats(vocab)
  try:
    best = max(pairs, key=pairs.get)
  except ValueError:
    break
  if pairs[best] < 2:
     sys.stderr.write('no pair has frequency > 1. Stopping\n')
     break
  vocab = merge_vocab(best, vocab)
  print(best)
```

11.4 词义消歧

“bank”可能翻译成银行，也可能翻译成河岸。翻译目标词取决于这个词在文本中表示的义项。WordNet 是英语的词汇数据库。其中包含一些常用词的词义义项。

下载 nltk 中的 WordNet 语料库：

```
>>> import nltk
>>> nltk.download('wordnet')
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\Administrator\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping corpora\wordnet.zip.
True
```

查询“bank”的同义词集合。

```
>>> from nltk.corpus import wordnet
>>> syns = wordnet.synsets("bank")
>>> print(syns[0].name())
bank.n.01
```

同义词集合的定义：

```
>>> print(syns[0].definition())
sloping land (especially the slope beside a body of water)
```

单词用例：

```
>>> print(syns[0].examples())
['they pulled the canoe up on the bank', 'he sat on the bank of the river and watched
the currents']
```

Lesk 算法是一个基于词典的词义消歧方法。Lesk 算法基于这样的假设：给定“邻域”（文本部分）中的单词将共享一个共同主题。

pywsd 是词义消歧技术的 Python 实现。

安装 pywsd：

```
#pip3 install -U nltk
#python3 -m nltk.downloader 'popular'
#pip3 install -U pywsd
```

使用 pywsd：

```
>>> from pywsd.lesk import simple_lesk
Warming up PyWSD (takes ~10 secs)... took 7.157701015472412 secs.
>>> sent = 'I went to the bank to deposit my money'
>>> ambiguous = 'bank'
>>> answer = simple_lesk(sent, ambiguous, pos='n')
>>> answer
Synset('depository_financial_institution.n.01')
>>> print(answer.definition())
a financial institution that accepts deposits and channels the money into lending
activities
```

11.5　词对齐

可以根据句子级对齐语料库实现词对齐。例如，假设训练语料库中有 2 个对齐的句子：

[("the house", "这 房子"), ("red house", "红 房子")]。挖掘出如图 11-1 所示的词对齐结果。

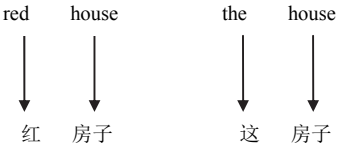


图 11-1 词对齐

可以使用 EM 算法实现词对齐。EM 算法通过交替执行期望（Expectation）阶段和最大化（Maximization）阶段，迭代直到收敛。其中：

- E 阶段：使用当前的翻译概率计算训练数据所有可能对齐的概率。
- M 阶段：使用这些对齐概率的估计去重新估计所有的翻译概率。

重复 E 和 M 阶段直到翻译概率值收敛为止。

初始阶段：

所有的词按相等可能对齐。

所有的 P(中文单词 | 英文单词)都相同。

例如这里有三个英文单词：{red,house,the}，三个中文单词{红,房子,这}。

翻译概率如表 11-1 所示。

表 11-1 翻译概率表

	红	房 子	这
red	1/3	1/3	1/3
house	1/3	1/3	1/3
the	1/3	1/3	1/3

例如：t(红|red)=1/3。

计算对齐概率 P(A, F | E)，就是翻译概率的连乘积。计算 P(A, F | E)的公式如下：

$$P(A,F|E)=\prod_{j=1}^{J} t(f_j|e_{a_j})$$

把所有句子的对齐概率都算一遍。例如，句子“red house”的对齐概率如下：

1/3*1/3 =1/9　　1/3*1/3 =1/9

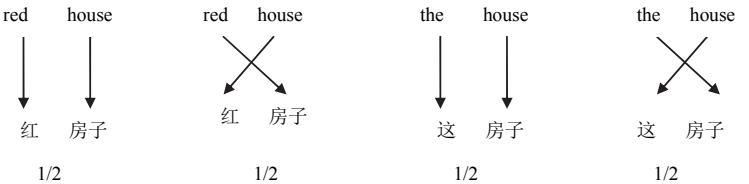
P(A,F|" red house") 的对齐概率

句子“the house”的对齐概率如下：

P(A,F|"the house") 的对齐概率

归一化，得到 P(A | F, E)

$$\frac{1/9}{2/9}=\frac{1}{2}$$



M 阶段根据 P(A | F, E)计算词的翻译概率。计算加权翻译计数 C(fj,ea(j)) += P(a|e,f)。得到的翻译概率表如表 11-2 所示。

表 11-2　更新后的翻译概率表

	红	房　子	这
red	1/2	1/2	0
house	1/2	1/2+1/2	1/2
the	0	1/2	1/2

例如，C(红,red)=0.5、C(房子,red)=0.5。

归一化行，按行加的值是 1，得到翻译概率如表 11-3 所示。

表 11-3　再次更新后的翻译概率表

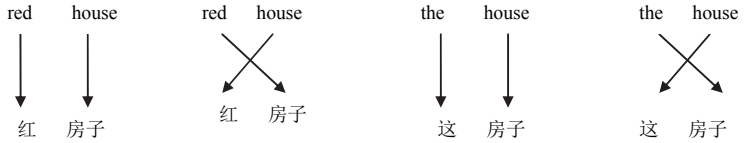
	红	房　子	这
red	1/2	1/2	0
house	1/4	1/2	1/4
the	0	1/2	1/2

例如，t(红|red)=0.5 t(房子|red)=0.5

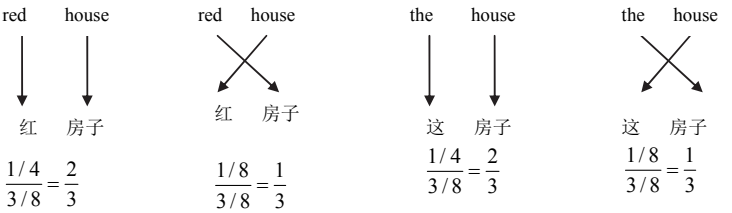
根据翻译概率乘积来估计 P(f | e)。

$$P(A,F|E)=\prod_{j=1}^{J} t(f_j|e_{a_j})$$

重新计算对齐概率 P(A, F | E)：



可以归一化进一步得到 P(A, F | E)准确的值。$P(A|E,F)=\frac{P(A,F|E)}{\sum_{A}P(A,F|E)}$



继续 EM 迭代，直到翻译参数收敛为止。

实现代码如下：

```
from operator import itemgetter
import collections
import decimal
from decimal import Decimal as D

#设置数值上下文
decimal.getcontext().prec = 4
decimal.getcontext().rounding = decimal.ROUND_HALF_UP

def mkcorpus(sentences:list):
    '''得到以词为单位的翻译语料库'''
    return [(es.split(), fs.split()) for (es, fs) in sentences]

def _constant_factory(value):
    '''用于均匀概率初始化的函数'''
    return lambda: value

def _train(corpus, loop_count=1000):
    f_keys = set()
    for (es, fs) in corpus:
        for f in fs:
            f_keys.add(f)
    #以均匀概率提供默认值
    t = collections.defaultdict(_constant_factory(D(1/len(f_keys))))

    #循环
```

```
    for i in range(loop_count):
        count = collections.defaultdict(D)
        total = collections.defaultdict(D)
        s_total = collections.defaultdict(D)
        for (es, fs) in corpus:
            #计算正规化因子
            for e in es:
                s_total[e] = D()
                for f in fs:
                    s_total[e] += t[(e, f)]
            for e in es:
                for f in fs:
                    count[(e, f)] += t[(e, f)] / s_total[e]
                    total[f] += t[(e, f)] / s_total[e]
        #估计概率
        for (e, f) in count.keys():
            t[(e, f)] = count[(e, f)] / total[f]

    return t

def train(sentences, loop_count=1000):
    corpus = mkcorpus(sentences)
    return _train(corpus, loop_count)
```

测试对齐：

```
sent_pairs = [("the house", "das Haus"),
              ("the book", "das Buch"),
              ("a book", "ein Buch")]
t = train(sent_pairs)
for k, v in t.items():
  print(k, v)
```

输出结果：

```
('the', 'das') 1
('the', 'Haus') 0.0004999
('house', 'das') 2.398E-299
('house', 'Haus') 1
('the', 'Buch') 7.52E-301
('book', 'das') 7.52E-301
('book', 'Buch') 1
('a', 'ein') 1
('a', 'Buch') 2.398E-299
('book', 'ein') 0.0004999
```

为了把结果导出到 MariaDB 数据库，首先创建表：

```
create table wordtrans(en text,     -- 英文单词
           de text,                 -- 德文单词
           prob float)              -- 翻译概率
```

然后写入数据库：

```
import pymysql.cursors

#连接到数据库
connection = pymysql.connect(host='localhost',
```

```
                        user='root',
                        password='sa',
                                    db='mysql',
                                    charset='utf8mb4',
                                    cursorclass=pymysql.cursors.DictCursor)

cursor=connection.cursor()

for k, v in t.items():
    cursor.execute('INSERT INTO wordtrans(en,de, prob) VALUES(%s, %s,%s)', (k[0],k[1], v))

connection.commit()  #提交更新

cursor.close()

cursor=connection.cursor()

cursor.execute("select * from wordtrans")
rows = cursor.fetchall()

for row in rows:
    print(row["en"],row["de"])
```

11.6 神经网络机器翻译

这里介绍使用神经网络方法实现一个法英机器翻译。使用 PyTorch 构建和训练模型。

可以使用 pip 安装 PyTorch：

```
# pip3 install torch torchvision
```

也可以先下载相应的安装包，然后从本地安装 whl 文件。在 Windows 操作系统下的一个安装过程如下：

```
>wget -c https://download.pytorch.org/whl/cpu/torch-1.0.1-cp37-cp37m-win_amd64.whl
>pip3 install torch-1.0.1-cp37-cp37m-win_amd64.whl
```

导入相关的包：

```
import unicodedata
import string
import re
import random
import time
import math

import torch
import torch.nn as nn
from torch.autograd import Variable
from torch import optim
import torch.nn.functional as F
```

定义一个常量来决定是使用 GPU（具体是 CUDA）还是 CPU。如果没有 GPU，则将其设置为 false。稍后当创建张量时，此变量将用于决定是将它们保留在 CPU 上还是将它们移动到 GPU。

```
USE_CUDA = True
```

下载英语到法语翻译句对。

```
#wget http://www.manythings.org/anki/fra-eng.zip
```

解压缩：

```
# unzip ./fra-eng.zip
```

查看语料库文件内容：

```
# more ./fra.txt
```

fra.txt 文件是以制表符分隔的翻译对列表：

```
I am cold.    J'ai froid.
```

需要每个单词的唯一索引，以便稍后用作网络的输入和目标。

```
SOS_token = 0   #开始标记，编号为 0
EOS_token = 1   #结束标记，编号为 1

class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2 # Count SOS and EOS

    def index_words(self, sentence):
        for word in sentence.split(' '):
            self.index_word(word)

    def index_word(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words   #给 word 一个索引编号
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1                      #增加索引编号值
        else:
            self.word2count[word] += 1
```

将 Unicode 字符转换为 ASCII，使所有内容都小写，并修剪大多数标点符号。

```
#将 Unicode 字符串转换为纯 ASCII 码
def unicode_to_ascii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

#小写化，修剪和删除非字母字符
def normalize_string(s):
    s = unicode_to_ascii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
```

为了读取数据文件，我们要将文件拆分为行，然后将行拆分成对。这些文件都是英语→其他语言，所以如果你想翻译其他语言→英语，可以用反向标志来反转对。

```
def read_langs(lang1, lang2, reverse=False):
    print("Reading lines...")

    #读取文件并分成行
    lines = open('../data/%s-%s.txt' % (lang1, lang2)).read().strip().split('\n')

    #将每一行拆分成对并进行标准化
    pairs = [[normalize_string(s) for s in l.split('\t')] for l in lines]

    #创建 Lang 实例
    if reverse:
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(lang2)
        output_lang = Lang(lang1)
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)

    return input_lang, output_lang, pairs
```

由于有很多例句，为了快速训练，将数据集修剪成相对简短的句子。这里最大长度是 10 个单词（包括标点符号），我们将过滤到转换为“I am”或“He is”等形式的句子（考虑删除撇号）。

```
MAX_LENGTH = 10

good_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s",
    "you are", "you re "
)

def filter_pair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) < MAX_LENGTH and \
        p[1].startswith(good_prefixes)

def filter_pairs(pairs):
    return [pair for pair in pairs if filter_pair(pair)]
```

准备数据的完整过程是：

- 读取文本文件并拆分成行，将行拆分成对。
- 规范化文本，按长度和内容进行过滤。
- 从成对的句子中制作单词列表。

```
def prepare_data(lang1_name, lang2_name, reverse=False):
    input_lang, output_lang, pairs = read_langs(lang1_name, lang2_name, reverse)
    print("Read %s sentence pairs" % len(pairs))

    pairs = filter_pairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
```

```
    print("Indexing words...")
    for pair in pairs:
        input_lang.index_words(pair[0])
        output_lang.index_words(pair[1])

    return input_lang, output_lang, pairs

input_lang, output_lang, pairs = prepare_data('eng', 'fra', True)

#打印示例对
print(random.choice(pairs))
```

把每个句子分成词并转换成一个张量，每个单词都用索引替换（来自之前制作的 Lang 索引）。

张量是一个用某种类型定义的多维数字数组，例如：FloatTensor 或 LongTensor。这里将使用 LongTensor 来表示整数索引数组。

可训练的 PyTorch 模块将变量作为输入，而不是普通的张量。变量基本上是一个张量，它能够跟踪图形状态，这使得 autograd（自动计算反向梯度）成为可能。

```
#返回索引列表
def indexes_from_sentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]

def variable_from_sentence(lang, sentence):
    indexes = indexes_from_sentence(lang, sentence)
    indexes.append(EOS_token)
    var = Variable(torch.LongTensor(indexes).view(-1, 1))
    if USE_CUDA: var = var.cuda()
    return var

def variables_from_pair(pair):
    input_variable = variable_from_sentence(input_lang, pair[0])
    target_variable = variable_from_sentence(output_lang, pair[1])
    return (input_variable, target_variable)
```

接下来建立模型。

seq2seq 网络的编码器是 RNN，它为输入句子中的每个单词输出一些值。对于每个输入词，编码器输出向量和隐藏状态，并将隐藏状态用于下一个输入词。

```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, n_layers=1):
        super(EncoderRNN, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)

    def forward(self, word_inputs, hidden):
        #注意：在整个输入序列上只运行一次该方法
        seq_len = len(word_inputs)
```

```
        embedded = self.embedding(word_inputs).view(seq_len, 1, -1)
        output, hidden = self.gru(embedded, hidden)
        return output, hidden

    def init_hidden(self):
        hidden = Variable(torch.zeros(self.n_layers, 1, self.hidden_size))
        if USE_CUDA: hidden = hidden.cuda()
        return hidden
```

解码器由四个主要部分组成——嵌入层将输入词转换为矢量；计算每个编码器输出的注意能量的层；RNN 层和输出层。

```
class BahdanauAttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, n_layers=1, dropout_p=0.1):
        super(AttnDecoderRNN, self).__init__()

        #定义参数
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p
        self.max_length = max_length

        #定义图层
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.dropout = nn.Dropout(dropout_p)
        self.attn = GeneralAttn(hidden_size)
        self.gru = nn.GRU(hidden_size * 2, hidden_size, n_layers, dropout=dropout_p)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, word_input, last_hidden, encoder_outputs):
        #请注意，我们只会向前运行一个解码器时间步，但会使用所有编码器输出

        #获取当前输入词的嵌入
        word_embedded = self.embedding(word_input).view(1, 1, -1) # S=1 x B x N
        word_embedded = self.dropout(word_embedded)

        #计算注意力并应用于编码器输出
        attn_weights = self.attn(last_hidden[-1], encoder_outputs)
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1)) # B x 1 x N

        #组合嵌入式输入词和上下文
        rnn_input = torch.cat((word_embedded, context), 2)
        output, hidden = self.gru(rnn_input, last_hidden)

        #最终输出层
        output = output.squeeze(0) # B x N
        output = F.log_softmax(self.out(torch.cat((output, context), 1)))

        #返回最终输出，隐藏状态和注意力权重
        return output, hidden, attn_weights
```

注意力模型：

```
class Attn(nn.Module):
    def __init__(self, method, hidden_size, max_length=MAX_LENGTH):
```

```
        super(Attn, self).__init__()

        self.method = method
        self.hidden_size = hidden_size

        if self.method == 'general':
            self.attn = nn.Linear(self.hidden_size, hidden_size)

        elif self.method == 'concat':
            self.attn = nn.Linear(self.hidden_size * 2, hidden_size)
            self.other = nn.Parameter(torch.FloatTensor(1, hidden_size))

    def forward(self, hidden, encoder_outputs):
        seq_len = len(encoder_outputs)

        #创建存储注意力的变量
        attn_energies = Variable(torch.zeros(seq_len)) # B x 1 x S
        if USE_CUDA: attn_energies = attn_energies.cuda()

        #计算每个编码器输出的能量
        for i in range(seq_len):
            attn_energies[i] = self.score(hidden, encoder_outputs[i])

        #将能量归一化为 0 到 1 范围内的权重，尺寸调整为 1 x 1 x seq_len
        return F.softmax(attn_energies).unsqueeze(0).unsqueeze(0)

    def score(self, hidden, encoder_output):

        if self.method == 'dot':
            energy = hidden.dot(encoder_output)
            return energy

        elif self.method == 'general':
            energy = self.attn(encoder_output)
            energy = hidden.dot(energy)
            return energy

        elif self.method == 'concat':
            energy = self.attn(torch.cat((hidden, encoder_output), 1))
            energy = self.other.dot(energy)
            return energy
```

为了确保编码器和解码器模型正常的一起工作，使用假词输入进行快速测试：

```
encoder_test = EncoderRNN(10, 10, 2)
decoder_test = AttnDecoderRNN('general', 10, 10, 2)
print(encoder_test)
print(decoder_test)

encoder_hidden = encoder_test.init_hidden()
word_input = Variable(torch.LongTensor([1, 2, 3]))
if USE_CUDA:
    encoder_test.cuda()
    word_input = word_input.cuda()
encoder_outputs, encoder_hidden = encoder_test(word_input, encoder_hidden)

word_inputs = Variable(torch.LongTensor([1, 2, 3]))
```

```
    decoder_attns = torch.zeros(1, 3, 3)
    decoder_hidden = encoder_hidden
    decoder_context = Variable(torch.zeros(1, decoder_test.hidden_size))

    if USE_CUDA:
        decoder_test.cuda()
        word_inputs = word_inputs.cuda()
        decoder_context = decoder_context.cuda()

    for i in range(3):
        decoder_output, decoder_context, decoder_hidden, decoder_attn = decoder_test(word_
inputs[i], decoder_context, decoder_hidden, encoder_outputs)
        print(decoder_output.size(), decoder_hidden.size(), decoder_attn.size())
        decoder_attns[0, i] = decoder_attn.squeeze(0).cpu().data
```

输出如下：

```
EncoderRNN (
  (embedding): Embedding(10, 10)
  (gru): GRU(10, 10, num_layers=2)
)
AttnDecoderRNN (
  (embedding): Embedding(10, 10)
  (gru): GRU(20, 10, num_layers=2, dropout=0.1)
  (out): Linear (20 -> 10)
  (attn): Attn (
    (attn): Linear (10 -> 10)
  )
)
torch.Size([1, 10]) torch.Size([2, 1, 10]) torch.Size([1, 1, 3])
torch.Size([1, 10]) torch.Size([2, 1, 10]) torch.Size([1, 1, 3])
torch.Size([1, 10]) torch.Size([2, 1, 10]) torch.Size([1, 1, 3])
```

训练函数定义如下：

```
    teacher_forcing_ratio = 0.5
    clip = 5.0

    def train(input_variable, target_variable, encoder, decoder, encoder_optimizer,
decoder_optimizer, criterion, max_length=MAX_LENGTH):

        #两个优化器的零梯度
        encoder_optimizer.zero_grad()
        decoder_optimizer.zero_grad()
        loss = 0

        #获取输入和目标句子的大小
        input_length = input_variable.size()[0]
        target_length = target_variable.size()[0]

        #单词运行通过编码器
        encoder_hidden = encoder.init_hidden()
        encoder_outputs, encoder_hidden = encoder(input_variable, encoder_hidden)

        #准备输入和输出变量
        decoder_input = Variable(torch.LongTensor([[SOS_token]]))
        decoder_context = Variable(torch.zeros(1, decoder.hidden_size))
        decoder_hidden = encoder_hidden # Use last hidden state from encoder to start decoder
```

```
        if USE_CUDA:
            decoder_input = decoder_input.cuda()
            decoder_context = decoder_context.cuda()

        #选择是否使用教师强制
        use_teacher_forcing = random.random() < teacher_forcing_ratio
        if use_teacher_forcing:

            #教师强制：用实际的目标作为下一个输入
            for di in range(target_length):
                decoder_output,  decoder_context,  decoder_hidden,  decoder_attention  =
decoder(decoder_input, decoder_context, decoder_hidden, encoder_outputs)
                loss += criterion(decoder_output, target_variable[di])
                decoder_input = target_variable[di] # Next target is next input

        else:
            #没有教师强制：使用网络自己的预测作为下一个输入
            for di in range(target_length):
                decoder_output,  decoder_context,  decoder_hidden,  decoder_attention  =
decoder(decoder_input, decoder_context, decoder_hidden, encoder_outputs)
                loss += criterion(decoder_output, target_variable[di])

                #从输出中获取最可能的单词索引
                topv, topi = decoder_output.data.topk(1)
                ni = topi[0][0]

                decoder_input = Variable(torch.LongTensor([[ni]])) # Chosen word is next input
                if USE_CUDA: decoder_input = decoder_input.cuda()

                #在句子结尾处停止（使用已知目标时不需要）
                if ni == EOS_token: break

        #反向传播
        loss.backward()
        torch.nn.utils.clip_grad_norm(encoder.parameters(), clip)
        torch.nn.utils.clip_grad_norm(decoder.parameters(), clip)
        encoder_optimizer.step()
        decoder_optimizer.step()

        return loss.data[0] / target_length
```

用于在给定当前时间和进度的情况下打印已用时间和估计剩余时间的辅助函数。

```
def as_minutes(s):
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

def time_since(since, percent):
    now = time.time()
    s = now - since
    es = s / (percent)
    rs = es - s
    return '%s (- %s)' % (as_minutes(s), as_minutes(rs))
```

运行训练：

```
    attn_model = 'general'
    hidden_size = 500
    n_layers = 2
    dropout_p = 0.05

    #初始化模型
    encoder = EncoderRNN(input_lang.n_words, hidden_size, n_layers)
    decoder = AttnDecoderRNN(attn_model, hidden_size, output_lang.n_words, n_layers,
dropout_p=dropout_p)

    #将模型移动到 GPU
    if USE_CUDA:
        encoder.cuda()
        decoder.cuda()

    #初始化优化器和损失函数
    learning_rate = 0.0001
    encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate)
    criterion = nn.NLLLoss()
```

然后设置变量以跟踪进度：

```
    #配置训练
    n_epochs = 50000
    print_every = 1000

    #跟踪经过的时间和平均运行时间
    start = time.time()
    print_loss_total = 0
```

为了实际训练，我们多次调用训练函数，随时打印摘要。

```
    #开始训练
    for epoch in range(1, n_epochs + 1):

        #获取此周期的训练数据
        training_pair = variables_from_pair(random.choice(pairs))
        input_variable = training_pair[0]
        target_variable = training_pair[1]

        #运行训练函数
        loss = train(input_variable, target_variable, encoder, decoder, encoder_optimizer,
decoder_optimizer, criterion)

        #跟踪损失
        print_loss_total += loss
        plot_loss_total += loss

        if epoch == 0: continue

        if epoch % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print_summary = '%s (%d %d%%) %.4f' % (time_since(start, epoch / n_epochs), epoch,
epoch / n_epochs * 100, print_loss_avg)
```

```
        print(print_summary)
```

可以使用 nvidia-smi 命令监控训练过程中的 GPU 使用情况。

评估与训练大致相同，但没有目标。评估时，将解码器的预测反馈给自己。每次预测一个单词时，都会将它添加到输出字符串中。如果它预测了 EOS 这个词，就会停在那里。

```
def evaluate(sentence, max_length=MAX_LENGTH):
    input_variable = variable_from_sentence(input_lang, sentence)
    input_length = input_variable.size()[0]

    #通过编码器运行
    encoder_hidden = encoder.init_hidden()
    encoder_outputs, encoder_hidden = encoder(input_variable, encoder_hidden)

    #为解码器创建起始向量
    decoder_input = Variable(torch.LongTensor([[SOS_token]])) # SOS
    decoder_context = Variable(torch.zeros(1, decoder.hidden_size))
    if USE_CUDA:
        decoder_input = decoder_input.cuda()
        decoder_context = decoder_context.cuda()

    decoder_hidden = encoder_hidden

    decoded_words = []
    decoder_attentions = torch.zeros(max_length, max_length)

    #运行解码器
    for di in range(max_length):
        decoder_output, decoder_context, decoder_hidden, decoder_attention = decoder
(decoder_input, decoder_context, decoder_hidden, encoder_outputs)
        decoder_attentions[di,:decoder_attention.size(2)] += decoder_attention.squeeze(0).
squeeze(0).cpu().data

        #从输出中选择顶部的单词
        topv, topi = decoder_output.data.topk(1)
        ni = topi[0][0]
        if ni == EOS_token:
            decoded_words.append('<EOS>')
            break
        else:
            decoded_words.append(output_lang.index2word[ni])

        #下一个输入是选择的单词
        decoder_input = Variable(torch.LongTensor([[ni]]))
        if USE_CUDA: decoder_input = decoder_input.cuda()

    return decoded_words, decoder_attentions[:di+1, :len(encoder_outputs)]
```

从训练集中评估随机句子并打印输入、目标和输出以做出一些主观质量判断：

```
def evaluate_randomly():
    pair = random.choice(pairs)

    output_words, decoder_attn = evaluate(pair[0])
    output_sentence = ' '.join(output_words)
```

```
    print('>', pair[0])
    print('=', pair[1])
    print('<', output_sentence)
    print('')
```

执行 evaluate_randomly()函数：

```
evaluate_randomly()
```

输出如下：

```
> je suis ambitieux .
= i m ambitious .
< i m ambitious . <EOS>
```

11.7 机器翻译的评价

狭义而言，机器翻译的评价一般仅指机器译文质量的人工评价或自动评价。评价机器翻译最常用的标准：译文的可理解性和译文的忠实度。人工评估机器翻译费时而且昂贵。

可以使用基于 n-gram 的 BLEU 值通过对比机器译文和人工参考译文的重合程度来自动评价机器翻译。

最基本的方法是将候选翻译的 n-gram 与参考翻译的 n-gram 进行比较并计算匹配数。BLEU 值由两部分组成，修正后的 n-gram 精度和简洁惩罚。

BLEU 的输出始终是介于 0 和 1 之间的数字。此值表示候选文本与参考文本的相似程度，值接近 1 表示更相似的文本。很少有人类翻译得分为 1，因为这表明候选文本与其中一个参考译文相同。因此，没有必要获得 1 的分值。因为有更多的机会匹配，添加额外的参考翻译将增加 BLEU 值。

0 到 1 之间的 BLEU 值也可以乘以 100，这样 BLEU 值范围变成 0～100。可以通过 BLEU 值了解参考译文的流畅度。

可以使用 nltk 下的 BLEU 模块计算 BLEU 值。

```
import nltk

hypothesis = ['It', 'is', 'a', 'cat', 'at', 'room']           #机器翻译的结果
reference = ['It', 'is', 'a', 'cat', 'inside', 'the', 'room'] #参考译文
#可能有几个参考译文
BLEUscore = nltk.translate.bleu_score.sentence_bleu([reference], hypothesis)
print(BLEUscore)
```

11.8 本章小结

虽然还没有机器翻译系统能提供无限制文本的全自动高质量机器翻译，但许多全自动系统都可以产生合理的输出。如果领域受到限制和控制，则机器翻译的质量将大大提高。