NLP\_自然语言处理项目(2)：seq2seq\_attention\_机器翻译

1、seq2seq\_attention\_机器翻译

seq2seq\_attention是一种基于神经网络的 机器翻译 模型，它通过 编码器 和 解码器 两个部分实现翻译功能。编码器将源语言句子转换为一个固定长度的向量表示，解码器则将这个向量作为输入，生成目标语言句子的翻译结果。

在seq2seq\_attention中，编码器和解码器都是由 循环神经网络（RNN）组成的。

编码器 将源语言句子中的每个单词依次输入RNN，每个时刻RNN的输出都会被传递到下一个时刻，直到最后一个时刻，最终得到源语言句子的向量表示。

解码器 的工作方式类似，但它不仅要考虑源语言句子的信息，还要根据当前生成的目标语言单词来不断调整生成下一个单词的概率分布。这就需要在解码器中引入注意力机制（attention mechanism），用来关注源语言句子中与当前要翻译的目标语言单词相关的部分，以便更准确地生成翻译结果。

具体来说，解码器会将每个时刻的输出向量与编码器中所有时刻的输出向量进行加权平均，以得到一个新的上下文向量。这个加权平均的权重是由注意力模型计算得出的，它会考虑源语言句子中每个单词与当前目标语言单词的相关性。最终，这个上下文向量会与当前时刻的解码器输入向量一起输入到解码器的RNN中，以生成下一个目标语言单词的概率分布。这个过程会不断迭代，直到生成了完整的目标语言句子。

seq2seq\_attention 相较于传统的seq2seq模型，能够更好地处理长句子和复杂的语法结构，从而提高翻译质量。

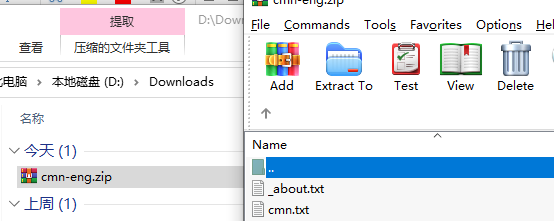
2、数据预处理

实现中文到英文的机器翻译

数据下载地址：

www.manythings.org/anki and tatoeba.org





工具类utils编写

utils.py : 构建中文和英文样本对

import matplotlib.pyplot as plt  
plt.switch\_backend('agg')  
import matplotlib.ticker as ticker  
import time  
import math  
import unicodedata  
from langconv import \*  
  
*# Turn a Unicode string to plain ASCII, thanks to  
# http://stackoverflow.com/a/518232/2809427*def unicode2Ascii(s):  
 return ''.join(  
 c for c in unicodedata.normalize('NFD', s)  
 if unicodedata.category(c) != 'Mn'  
 )  
  
def normalizeString(s):  
 *"""  
 # Lowercase, trim, and remove non-letter characters  
 :param s:  
 :return:  
 """* s = unicode2Ascii(s.lower().strip())  
 s = re.sub(r"([.!?])", r" \1", s)  
 s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)  
 return s  
  
*# 繁体字转简体字*def cht\_to\_chs(line):  
 line = Converter('zh-hans').convert(line)  
 line.encode('utf-8')  
 return line  
  
def asMinutes(s):  
 m = math.floor(s / 60)  
 s -= m \* 60  
 return '%dm %ds' % (m, s)  
  
def timeSince(since, percent):  
 now = time.time()  
 s = now - since  
 es = s / (percent)  
 rs = es - s  
 return '%s (- %s)' % (asMinutes(s), asMinutes(rs))  
  
def showPlot(points):  
 plt.figure()  
 fig, ax = plt.subplots()  
 *# this locator puts ticks at regular intervals* loc = ticker.MultipleLocator(base=0.2)  
 ax.yaxis.set\_major\_locator(loc)  
 plt.plot(points)

Langconv.py编写

*#!/usr/bin/env python  
# -\*- coding: utf-8 -\*-*from copy import deepcopy  
import re  
  
try:  
 import psyco  
 psyco.full()  
except:  
 pass  
  
try:  
 from zh\_wiki import zh2Hant, zh2Hans  
except ImportError:  
 from zhtools.zh\_wiki import zh2Hant, zh2Hans  
  
import sys  
py3k = sys.version\_info >= (3, 0, 0)  
  
if py3k:  
 UEMPTY = ''  
else:  
 \_zh2Hant, \_zh2Hans = {}, {}  
 for old, new in ((zh2Hant, \_zh2Hant), (zh2Hans, \_zh2Hans)):  
 for k, v in old.items():  
 new[k.decode('utf8')] = v.decode('utf8')  
 zh2Hant = \_zh2Hant  
 zh2Hans = \_zh2Hans  
 UEMPTY = ''.decode('utf8')  
  
*# states*(START, END, FAIL, WAIT\_TAIL) = list(range(4))  
*# conditions*(TAIL, ERROR, MATCHED\_SWITCH, UNMATCHED\_SWITCH, CONNECTOR) = list(range(5))  
  
MAPS = {}  
  
class Node(object):  
 def \_\_init\_\_(self, from\_word, to\_word=None, is\_tail=True,  
 have\_child=False):  
 self.from\_word = from\_word  
 if to\_word is None:  
 self.to\_word = from\_word  
 self.data = (is\_tail, have\_child, from\_word)  
 self.is\_original = True  
 else:  
 self.to\_word = to\_word or from\_word  
 self.data = (is\_tail, have\_child, to\_word)  
 self.is\_original = False  
 self.is\_tail = is\_tail  
 self.have\_child = have\_child  
  
 def is\_original\_long\_word(self):  
 return self.is\_original and len(self.from\_word)>1  
  
 def is\_follow(self, chars):  
 return chars != self.from\_word[:-1]  
  
 def \_\_str\_\_(self):  
 return '<Node, %s, %s, %s, %s>' % (repr(self.from\_word),  
 repr(self.to\_word), self.is\_tail, self.have\_child)  
  
 \_\_repr\_\_ = \_\_str\_\_  
  
class ConvertMap(object):  
 def \_\_init\_\_(self, name, mapping=None):  
 self.name = name  
 self.\_map = {}  
 if mapping:  
 self.set\_convert\_map(mapping)  
  
 def set\_convert\_map(self, mapping):  
 convert\_map = {}  
 have\_child = {}  
 max\_key\_length = 0  
 for key in sorted(mapping.keys()):  
 if len(key)>1:  
 for i in range(1, len(key)):  
 parent\_key = key[:i]  
 have\_child[parent\_key] = True  
 have\_child[key] = False  
 max\_key\_length = max(max\_key\_length, len(key))  
 for key in sorted(have\_child.keys()):  
 convert\_map[key] = (key in mapping, have\_child[key],  
 mapping.get(key, UEMPTY))  
 self.\_map = convert\_map  
 self.max\_key\_length = max\_key\_length  
  
 def \_\_getitem\_\_(self, k):  
 try:  
 is\_tail, have\_child, to\_word = self.\_map[k]  
 return Node(k, to\_word, is\_tail, have\_child)  
 except:  
 return Node(k)  
  
 def \_\_contains\_\_(self, k):  
 return k in self.\_map  
  
 def \_\_len\_\_(self):  
 return len(self.\_map)  
  
class StatesMachineException(Exception): pass  
  
class StatesMachine(object):  
 def \_\_init\_\_(self):  
 self.state = START  
 self.final = UEMPTY  
 self.len = 0  
 self.pool = UEMPTY  
  
 def clone(self, pool):  
 new = deepcopy(self)  
 new.state = WAIT\_TAIL  
 new.pool = pool  
 return new  
  
 def feed(self, char, map):  
 node = map[self.pool+char]  
  
 if node.have\_child:  
 if node.is\_tail:  
 if node.is\_original:  
 cond = UNMATCHED\_SWITCH  
 else:  
 cond = MATCHED\_SWITCH  
 else:  
 cond = CONNECTOR  
 else:  
 if node.is\_tail:  
 cond = TAIL  
 else:  
 cond = ERROR  
  
 new = None  
 if cond == ERROR:  
 self.state = FAIL  
 elif cond == TAIL:  
 if self.state == WAIT\_TAIL and node.is\_original\_long\_word():  
 self.state = FAIL  
 else:  
 self.final += node.to\_word  
 self.len += 1  
 self.pool = UEMPTY  
 self.state = END  
 elif self.state == START or self.state == WAIT\_TAIL:  
 if cond == MATCHED\_SWITCH:  
 new = self.clone(node.from\_word)  
 self.final += node.to\_word  
 self.len += 1  
 self.state = END  
 self.pool = UEMPTY  
 elif cond == UNMATCHED\_SWITCH or cond == CONNECTOR:  
 if self.state == START:  
 new = self.clone(node.from\_word)  
 self.final += node.to\_word  
 self.len += 1  
 self.state = END  
 else:  
 if node.is\_follow(self.pool):  
 self.state = FAIL  
 else:  
 self.pool = node.from\_word  
 elif self.state == END:  
 *# END is a new START* self.state = START  
 new = self.feed(char, map)  
 elif self.state == FAIL:  
 raise StatesMachineException('Translate States Machine '  
 'have error with input data %s' % node)  
 return new  
  
 def \_\_len\_\_(self):  
 return self.len + 1  
  
 def \_\_str\_\_(self):  
 return '<StatesMachine %s, pool: "%s", state: %s, final: %s>' % (  
 id(self), self.pool, self.state, self.final)  
 \_\_repr\_\_ = \_\_str\_\_  
  
class Converter(object):  
 def \_\_init\_\_(self, to\_encoding):  
 self.to\_encoding = to\_encoding  
 self.map = MAPS[to\_encoding]  
 self.start()  
  
 def feed(self, char):  
 branches = []  
 for fsm in self.machines:  
 new = fsm.feed(char, self.map)  
 if new:  
 branches.append(new)  
 if branches:  
 self.machines.extend(branches)  
 self.machines = [fsm for fsm in self.machines if fsm.state != FAIL]  
 all\_ok = True  
 for fsm in self.machines:  
 if fsm.state != END:  
 all\_ok = False  
 if all\_ok:  
 self.\_clean()  
 return self.get\_result()  
  
 def \_clean(self):  
 if len(self.machines):  
 self.machines.sort(key=lambda x: len(x))  
 *# self.machines.sort(cmp=lambda x,y: cmp(len(x), len(y)))* self.final += self.machines[0].final  
 self.machines = [StatesMachine()]  
  
 def start(self):  
 self.machines = [StatesMachine()]  
 self.final = UEMPTY  
  
 def end(self):  
 self.machines = [fsm for fsm in self.machines  
 if fsm.state == FAIL or fsm.state == END]  
 self.\_clean()  
  
 def convert(self, string):  
 self.start()  
 for char in string:  
 self.feed(char)  
 self.end()  
 return self.get\_result()  
  
 def get\_result(self):  
 return self.final  
  
  
def registery(name, mapping):  
 global MAPS  
 MAPS[name] = ConvertMap(name, mapping)  
  
registery('zh-hant', zh2Hant)  
registery('zh-hans', zh2Hans)  
del zh2Hant, zh2Hans