3、加载数据集

datasets.py : 构建中文和英文样本对

sentence1 = normalizeString(l[0]) # 英文，英文文本处理(大写转小写，过滤非法字符等)

sentence2 = cht\_to\_chs(l[1]) # 中文，繁体转简体

因为原始数据中有一些繁体字和中文大写问题，需要转换

import jieba

from utils import normalizeString

from utils import cht\_to\_chs

SOS\_token = 0 # 起始符

EOS\_token = 1 # 终止符

MAX\_LENGTH = 10 # 将长度过长的句子去掉

class Lang:

def \_\_init\_\_(self, name):

self.name = name

self.word2index = {} # 记录词对应的索引

self.word2count = {} # 记录每个词的词频

self.index2word = {

0: "SOS", 1: "EOS"

} # 记录索引到词

self.n\_words = 2 # 记录语料库中有多少种词，初始值为2(起始符+终止符)

# 对词进行统计

def addWord(self, word):

if word not in self.word2index: # 如果词不在统计表中，添加进统计表

self.word2index[word] = self.n\_words # 词的索引为该词是第几种的词

self.word2count[word] = 1

self.index2word[self.n\_words] = word

self.n\_words += 1 # 字典中的词数量+1

else: # 该词在统计表中

self.word2count[word] += 1

# 对句子进行分词

def addSentence(self, sentence):

for word in sentence.split(" "): # 将 "你 吃饭 了 吗 ？" 分割为 ["你"，“吃”,“吃饭”," 了"," 吗"] 的list数组

self.addWord(word) # 依次将每个词统计

# 文本解析

def readLangs(lang1, lang2, path):

lines = open(path, encoding='utf-8').readlines() # 拿到文本的所有行

lang1\_cls = Lang(lang1)

lang2\_cls = Lang(lang2)

pairs = [] # 记录样本对

for l in lines: # 逐行处理

l = l.split("\t") # 以Tab分割

sentence1 = normalizeString(l[0]) # 英文，英文文本处理(大写转小写，过滤非法字符等)

sentence2 = cht\_to\_chs(l[1]) # 中文，繁体转简体

seg\_list = jieba.cut(sentence2, cut\_all=False) # 调用结巴分词对中文进行分割，得到分词后的数组

sentence2 = " ".join(seg\_list) #将中文句子分词后的数组拼接为字符串。join() 方法用于把数组中的所有元素放入一个字符串。元素是通过指定的分隔符进行分隔的。

# 英文文本是天然分词的，不需要分词 # 向英文一样，通过空格拼接中文分词结果

if len(sentence1.split(" ")) > MAX\_LENGTH: # 过滤一些长句,大于10个词的的不统计

continue # 忽略当前的一次循环

if len(sentence2.split(" ")) > MAX\_LENGTH:

continue

pairs.append([sentence1, sentence2]) # [[“what are you doing?”,"你 在 干 什么"],....]

lang1\_cls.addSentence(sentence1) # 统计每种语言的词频

lang2\_cls.addSentence(sentence2)

return lang1\_cls, lang2\_cls, pairs

# 测试

lang1 = "en"

lang2 = "cn"

path = "../data/cmn.txt"

lang1\_cls, lang2\_cls, pairs = readLangs(lang1, lang2, path)

print(len(pairs))

print(lang1\_cls.n\_words)

print(lang1\_cls.index2word)

print(lang2\_cls.n\_words)

print(lang2\_cls.index2word)

代码是一个语言模型读取数据的预处理部分，目的是将源语言和目标语言的文本进行读取、分词和处理，以便在机器翻译模型中使用。

具体功能： 定义了一个Lang类，该类主要是记录每个词的出现频率和对应的索引，同时提供了一个方法用于将句子进行分词并更新词表。

readLangs函数用于读取源语言和目标语言的文本，将其分别进行预处理，并返回三个值：源语言Lang类、目标语言Lang类和处理好的样本对列表。

在readLangs函数中，逐行读取源文本中的内容，对每个样本进行处理：

用"\t"分割源语言和目标语言；

对源语言进行简单的处理，去除空格和标点符号等；

对目标语言进行繁体转简体和结巴分词处理；

将处理好的样本加入到样本对列表中，并更新源语言和目标语言的词表。

最后输出了样本对数量、源语言和目标语言的词汇量和词表，以及词表中的词汇和对应的索引。

4、搭建模型结构

import torch

import torch.nn as nn

import torch.nn.functional as F

from datasets import MAX\_LENGTH

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# 编码器

class EncoderRNN(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size):

super(EncoderRNN, self).\_\_init\_\_() # 完成类的初始化

self.hidden\_size = hidden\_size

self.embedding = nn.Embedding(input\_size, hidden\_size) # 词嵌入层, 第一个参数：字典大小，第二个参数：有多少维的向量表征单词。

self.gru = nn.GRU(input\_size=hidden\_size, hidden\_size=hidden\_size) # gru层。也可以选择lstm层或者其他网络作为编码结果

def forward(self, input, hidden):

embedded = self.embedding(input).view(1, 1, -1) # 转化为3维的，因为gru的输入要求是3维的

output = embedded

output, hidden = self.gru(output, hidden)

return output, hidden #返回gru输出的结果和隐藏层信息

# 初始化隐藏状态h0

def initHidden(self):

return torch.zeros(1, 1, self.hidden\_size, device=device) # gru的输入为3维的

# 实现两种解码RNN(不带attention + 带attention)

# 不带attention的解码器

class DecoderRNN(nn.Module):

def \_\_init\_\_(self, hidden\_size, output\_size):

super(DecoderRNN, self).\_\_init\_\_()

self.embedding = nn.Embedding(output\_size, hidden\_size)

self.gru = nn.GRU(input\_size=hidden\_size, hidden\_size=hidden\_size)

self.out = nn.Linear(hidden\_size, output\_size)

self.softmax = nn.LogSoftmax(dim=1)

def forward(self, input, hidden):

output = self.embedding(input).view(1, 1, -1)

output = F.relu(output)

output, hidden = self.gru(output, hidden)

output = self.softmax(self.out(output[0]))

return output, hidden

# 初始化隐藏状态h0

def initHidden(self):

return torch.zeros(1, 1, self.hidden\_size, device=device)

# 带attention

class AttenDecoderRNN(nn.Module):

def \_\_init\_\_(self, hidden\_size, output\_size, dropout\_p=0.1, max\_len=MAX\_LENGTH):

super(AttenDecoderRNN, self).\_\_init\_\_()

self.hidden\_size = hidden\_size

self.output\_size = output\_size

self.dropout\_p = dropout\_p

self.max\_len = max\_len

self.embedding = nn.Embedding(self.output\_size, self.hidden\_size)

self.attn = nn.Linear(self.hidden\_size \* 2, self.max\_len) #要对两个结果进行连接，因此要乘以2

self.attn\_combine = nn.Linear(self.hidden\_size \* 2, self.hidden\_size)

self.dropout = nn.Dropout(self.dropout\_p)

self.gru = nn.GRU(input\_size=self.hidden\_size, hidden\_size=self.hidden\_size)

self.out = nn.Linear(self.hidden\_size, self.output\_size)

def forward(self, input, hidden, encoder\_outputs):

embedded = self.embedding(input).view(1, 1, -1) # 一个

embedded = self.dropout(embedded)

atten\_weight = F.softmax(

self.attn(torch.cat([embedded[0], hidden[0]], 1)), # 将embedded和hidden进行拼接，来学习attention权重

dim=1

)

att\_applied = torch.bmm(

atten\_weight.unsqueeze(0),

encoder\_outputs.unsqueeze(0)

)

output = torch.cat([embedded[0], att\_applied[0]], dim=1)

output = self.attn\_combine(output).unsqueeze(0)

output = F.relu(output)

output, hidden = self.gru(output, hidden)

output = F.log\_softmax(self.out(output[0]), dim=1)

return output, hidden, atten\_weight

# 初始化隐藏状态h0

def initHidden(self):

return torch.zeros(1, 1, self.hidden\_size, device=device)

if \_\_name\_\_ == '\_\_main\_\_':

encoder\_net = EncoderRNN(5000, 256)

decoder\_net = DecoderRNN(256, 5000)

atten\_decoder\_net = AttenDecoderRNN(256, 5000)

tensor\_in = torch.tensor([12, 14, 16, 18], dtype=torch.long).view(-1, 1) # 定义输入并调整shape

hidden\_in = torch.zeros(1, 1, 256)

# 测试编码网络

encoder\_out, encoder\_hidden = encoder\_net(tensor\_in[0], hidden\_in)

print(encoder\_out)

print(encoder\_hidden)

# 测试解码网络

tensor\_in = torch.tensor([100])

hidden\_in = torch.zeros(1, 1, 256)

encoder\_out = torch.zeros(10, 256) # 第一维大小取决于MAX\_LENGTH,此处为10

out1, out2, out3 = atten\_decoder\_net(tensor\_in, hidden\_in, encoder\_out)

print(out1, out2, out3)

out1, out2 = decoder\_net(tensor\_in, hidden\_in)

print(out1, out2)

这是一个PyTorch的代码实现，用于构建一个Seq2Seq模型。该模型由编码器和解码器两个部分组成，其中编码器采用了GRU，解码器可以选择不带Attention或者带Attention。

具体分析代码的功能：

导入需要的PyTorch模块和变量（如设备类型、最大长度等）。

实现编码器部分，包括：初始化函数、词嵌入层、GRU层，以及前向传播函数。其中前向传播函数的输入是一个输入序列（input）和一个隐藏状态（hidden），输出是一个输出张量（output）和一个隐藏状态（hidden）。

实现不带Attention的解码器部分，包括：初始化函数、词嵌入层、GRU层、输出层和softmax层，以及前向传播函数。其中前向传播函数的输入是一个输入序列（input）和一个隐藏状态（hidden），输出是一个输出张量（output）和一个隐藏状态（hidden）。

实现带Attention的解码器部分，包括：初始化函数、词嵌入层、Attention层、GRU层、输出层和softmax层，以及前向传播函数。其中前向传播函数的输入是一个输入序列（input）、一个隐藏状态（hidden）和编码器的输出张量（encoder\_outputs），输出是一个输出张量（output）、一个隐藏状态（hidden）和注意力权重（atten\_weight）。

实现初始化隐藏状态的函数initHidden，用于初始化隐藏状态。

注意，此处使用了一个三层的神经网络：GRU层、输出层和softmax层。其中，GRU层的输出作为输入传递给输出层，输出层再将结果传递给softmax层进行计算。最终的输出结果是一个向量，代表每个单词的概率分布。由于使用了softmax函数，因此输出结果之和等于1，可以作为概率分布使用。