
# 机器学习与自然语言处理实验五


| | |
|---|---|
| 院（系）名称 | 自动化科学与电气工程学院 |
| 学生姓名 | 赵怡然 |
| 学生学号 | ZY2103208 |
| 指导老师 | 秦曾昌 |


2022 年 06 月 18 日

# 一、 实验背景介绍

Seq2Seq 是一个 Encoder–Decoder 结构的网络，它的输入是一个序列，输出也是一个序列， Encoder 中将一个可变长度的信号序列变为固定长度的向量表达，Decoder 将这个固定长度的向量变成可变长度的目标的信号序列。

这个结构最重要的地方在于输入序列和输出序列的长度是可变的，可以用于翻译，聊天机器人，句法分析，文本摘要等

# 二、 实验目标

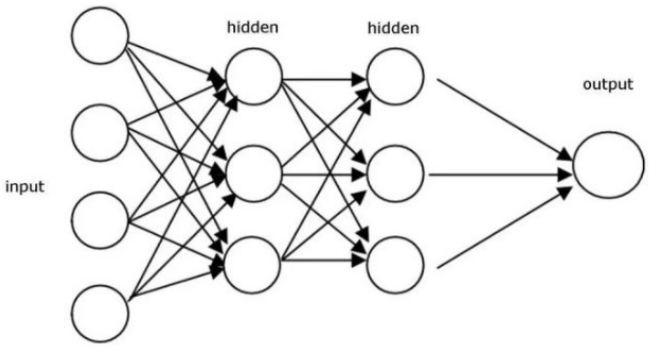
基于 Seq2seq 模型来实现文本生成的模型，输入英文，来生成对应中文翻译并做分析。

# 三、 相关原理

在 NLP 任务中，我们通常会遇到不定长的语言序列，比如机器翻译任务中，输入可能是一段不定长的英文文本，输出可能是不定长的中文或者法语序列。当遇到输入和输出都是不定长的序列时，可以使用编码器-解码器（encoder-decoder）模型或者 seq2seq 模型。其基本思想是编码器用来分析输入序列，解码器用来生成输出序列。

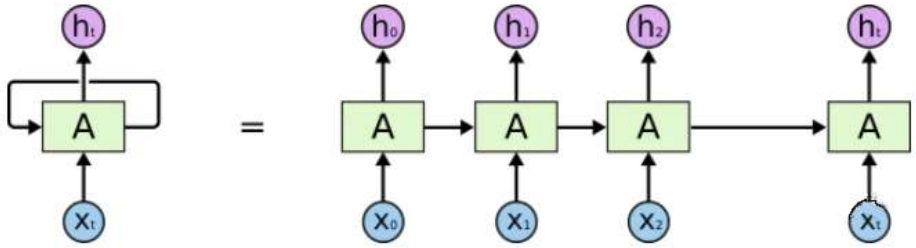
## 3.1 RNN

普通的神经网络中的神经元分布为：




每一个点的数值乘以相关权重，求和，通过激活函数得到下一层的每个点。但上述方式会带来一个问题。每个数据为独立变量，不存在相互关联。而对于连续数据来说，前一个点与后一个点之间存在关联。为解决这个问题，让神经网络发掘出前后点之间的关联，提出循环神经网络 RNN。

RNN 网络如下图所示：




前一个节点的信息传递给下一个节点作为输入，其运算公式为：

$$ht = tanh(W * xt + U * ht - 1 + b)$$

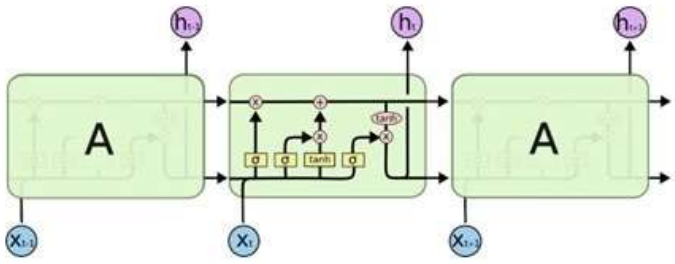
其中 xt 为 t 时刻的输入，ht 为 t 时刻的输出，W 为该节点的权重，tanh 为激活函数。当我们想知道 ht 时，ht 其实包含了 x0~xt 的所有数值信息。

但激活函数 tanh 会带来梯度消失的问题，tanh 输出的数值是<=1 的。如果权值 W 小于 1，在重复多次相乘后，最终权值趋近于 0（可参考 0.9^100＝2.6e-05），没有任何差异；如果权值 W 大于 1，则重复多次相乘后趋近于无穷大（1.1^100 = 1.3e+04），而无穷大的梯度放在 tanh 中只会无限趋近于 1，没有任何差异。这种趋近于 0 或 1 的现象被称为梯度消失（梯度弥散）与梯度爆炸。因此，对于长链的时间序列，远端的数值点对于现在的输出起不到任何作用，这个现象称为长期依赖。

为改进长期依赖的问题，推出改进方法：LSTM。

## 3.2 LSTM

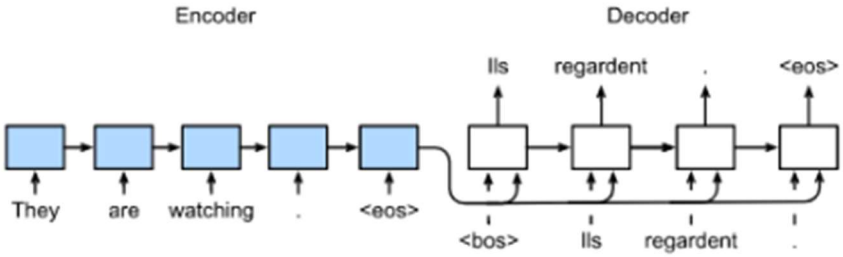
LSTM（Long Short Term Memory ） 长短期记忆神经网络。在 RNN 的基础上加入了遗忘机制，选择性的保留或遗忘前期的某些数据，且不再采用乘法而是加法以避免梯度爆炸的问题。




## 3.3 Seq2Seq

实验采用两个 LSTM 设计 Encoder-Decoder 结构，形成 Seq2Seq 模型，并将其应用于英文翻译。LSTM 的 Encoder 可以使用长度可变的序列作为输入，将其

转换成固定长度的隐藏状态。换言之，输入序列（源）的信息被 编码 到循环神经网络编码器的隐藏状态中。为了一个接着一个的生成输出序列的标记，独立的循环神经网络解码器是基于输入序列的编码信息和输出序列已经生成的标记（例如在语言模型的任务中）来预测下一个标记。下图演示了如何在机器翻译伤中使用两个循环神经网络进行序列到序列学习。




在上图中，特定的“<eos>”表示序列结束标记。一旦输出序列生成此标记，模型就可以停止执行预测。在循环神经网络解码器的初始化时间步，有两个特定的设计决定。首先，特定的“<bos>”表示序列开始标记，它是解码器的输入序列的第一个标记。其次，使用循环神经网络编码器最终的隐藏状态来初始化解码器的隐藏状态。

## 四、 实验结果与分析

### 4.1 实验流程

1. 数据集读取：读取中英文本语料库 data/translate.csv 文件共两列，一列英文，一列中文。每行中英文意思相同。

2. 数据预处理：基于 pytorch 中的 Dataset 类构建数据集，根据提前准备好的字母、汉字、标点符号的 Index 表得到各文本段的字向量，每次训练的 batch 中最大文本长度大小，对文本的字向量了进行填充，使得每个 batch 下的向量长度相同。

3. Encoder 编码器：基于 pytorch 中的 nn.Module 构建编码器，使用 LSTM，输入英文，拿到输出的隐层编码值（特征值）。将隐层编码值交给后续 Decoder 解码器。

4. Decoder 解码器：拿到基于 LSTM 的 Encoder 编码器中的隐层编码值（特征值），再次通过 LSTM 得到 decoder_output。

5. 分类并求交叉熵损失：将 Decoder 得到的特征通过线性层将特征转化为类

别的分类器。由预测值（分类类别）和标签计算得到交叉熵损失。

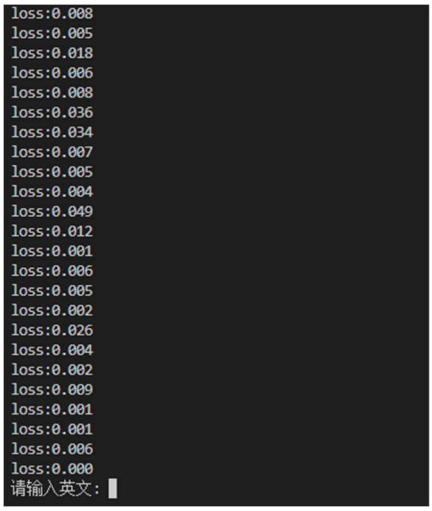
6. 训练：通过 adam 优化器，设置学习率，进行迭代训练。

7. 预测：将用户输入的英文文本转化为 Index 向量，然后交给 Encoder，得到隐层编码值。通过 Decoder 和分类器得到具体的中文字。

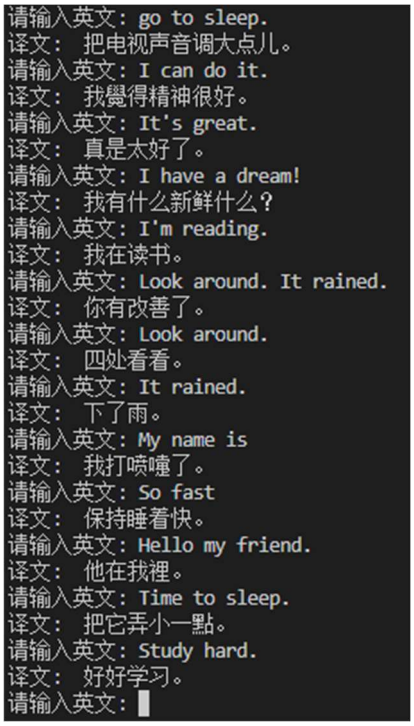
实验所有代码基于 Pytorch 为了加速训练使用 N 卡的 cuda 核心。

## 4.2 实验结果

设置参数为训练语句共 2000 条；batch 大小为 2；epoch 每个样本的训练次数为 100 词；学习率为 0.001。训练结束后交叉熵损失小于 0.001。如下图所示：




翻译结果如下：



由结果可以看到，由于训练的样本只有 2000 条。对于样本中出现过的英文段，能够进行准确的翻译，但对于没有出现过的短语，翻译的准确率并不高。

将语料库所有样本全部进行训练的情况下，在没有设定合适的参数下，训练需要大量的时间，大约 1h 左右。交叉熵损失只能降低到 0.4 左右。翻译效果并不够好。

## 4.3 结果分析

观察上表，可以看出通过 LSTM 构建 Seq2Seq 模型完成英文翻译的任务。但是效果并没有理想的好。仍需要后续的优化，和相关知识的学习。扩大大数据集，在合适的参数下，应该会有更好的效果。

## 附录：代码

```
'''
基于 Seq2seq 模型来实现文本生成的模型,输入英文,来生成对应中文翻译并做分析。
'''

import torch
import torch.nn as nn
import pandas as pd
from torch.utils.data import Dataset,DataLoader
import pickle

def get_datas(file = "datas\\translate.csv",nums = None):
  ''' Function: get_datas

  读取中英文本语料库
  data/translate.csv 文件共两列，一列英文，一列中文。每行中英文意思相同.

  Parameters:
    file: string - 语料库 csv 相对地址
    nums: int - 语料库读取文本段个数

  Return: (List, List) - (英文文本段, 中文文本段)

  '''

  all_datas = pd.read_csv(file)
  en_datas = list(all_datas["english"])
  ch_datas = list(all_datas["chinese"])

```

```
29.   if nums == None:
30.     return en_datas,ch_datas
31.   else:
32.     return en_datas[:nums],ch_datas[:nums]
33.
34.
35. class MyDataset(Dataset):
36.   ''' Class: MyDataSet
37.
38.   基于 pytorch Dataset 类构建数据集，继承并重写了
__getitem__(), __init__(), __len__()三个方法.
39.
40.
41.   '''
42.   def __init__(self,en_data,ch_data,en_word_2_index,ch_word_2_index)
:
43.     ''' Function: __init__
44.
45.     初始化数据集
46.
47.     Parameters:
48.       en_data: List[str] - 英文数据
49.       ch_data: List[str] - 中文数据
50.       en_word_2_index: List[int] - 英文字母对应 index
51.       ch_word_2_index: List[int] - 中文汉字对应 index
52.
53.     '''
54.     self.en_data = en_data
55.     self.ch_data = ch_data
56.     self.en_word_2_index = en_word_2_index
57.     self.ch_word_2_index = ch_word_2_index
58.
59.   def __getitem__(self,index):
60.     ''' Function: __getitme__
61.
62.     返回数据集指定 Index 下，根据通过的英文字母和汉字对应 Index 列表，返回
英文文段和对应的中文文段的向量.
63.
64.     Return: (List[int], List[int])
65.
66.     '''
67.     en = self.en_data[index]
68.     ch = self.ch_data[index]
69.
```

```
70.     en_index = [self.en_word_2_index[i] for i in en]
71.     ch_index = [self.ch_word_2_index[i] for i in ch]
72.
73.     return en_index,ch_index
74.
75.
76.   def batch_data_process(self,batch_datas):
77.     ''' Function: batch_data_process
78.
79.     替换掉 pytorch 下的 Dataset 里的默认填充方法 collate_fn,
80.     解决中英文文本对应向量长度不一致问题, 按照每个 batch 最大的长度进行填充. 并添加开始标识符和结束标识符.
81.
82.     Parameters:
83.       batch_datas: List[Tuple(List[int], List[int])] - 一个 batch 下的所有文本的中英文向量
84.
85.     Returns: torch.Tensor - 返回字向量
86.
87.     '''
88.     global device
89.     en_index , ch_index = [],[]
90.     en_len , ch_len = [],[]
91.
92.     for en,ch in batch_datas:
93.       en_index.append(en)
94.       ch_index.append(ch)
95.       en_len.append(len(en))
96.       ch_len.append(len(ch))
97.
98.     max_en_len = max(en_len)
99.     max_ch_len = max(ch_len)
100.
101.       en_index = [ i + [self.en_word_2_index["<PAD>"]] * (max_en_len - len(i))   for i in en_index]
102.       ch_index = [[self.ch_word_2_index["<BOS>"]]+ i + [self.ch_word_2_index["<EOS>"]] + [self.ch_word_2_index["<PAD>"]] * (max_ch_len - len(i))   for i in ch_index]
103.
104.       en_index = torch.tensor(en_index,device = device)
105.       ch_index = torch.tensor(ch_index,device = device)
106.
107.
108.       return en_index,ch_index
```

```
109.
110.
111.     def __len__(self):
112.       ''' Function: __len__
113.
114.       返回数据集长度，若中英文文本数据长度个数不一致抛出错误.
115.
116.       '''
117.       assert len(self.en_data) == len(self.ch_data)
118.       return len(self.ch_data)
119.
120.
121.   class Encoder(nn.Module):
122.     ''' Encoder
123.
124.     基于 pytorch.nn.Module 构建编码器，使用 LSTM，输入英文，拿到输出的
隐层
125.
126.     '''
127.
128.     def __init__(self,encoder_embedding_num,encoder_hidden_num,en_
corpus_len):
129.
130.       ''' Function: __init__
131.
132.       Parameters:
133.         encoder_embedding_num: int - 嵌入(embedding)层数量
134.         encoder_hidden_num: int - 隐层数量
135.         en_corpus_len: int - 构成英文语料库字符的长度，用于创建字向量.
136.
137.
138.       '''
139.       super().__init__()
140.       self.embedding = nn.Embedding(en_corpus_len,encoder_embeddin
g_num)
141.       self.lstm = nn.LSTM(encoder_embedding_num,encoder_hidden_num
,batch_first=True)
142.
143.     def forward(self,en_index):
144.       ''' Function: forword
145.
146.       输入英文文段向量，根据 LSTM 返回隐藏层编码
147.
148.       '''
```

```
149.         en_embedding = self.embedding(en_index)
150.         _,encoder_hidden =self.lstm(en_embedding)
151.
152.         return encoder_hidden
153.
154.
155.
156.     class Decoder(nn.Module):
157.       ''' Decoder
158.
159.       基于 pytorch.nn.Module 构建解码器, 使用 LSTM
160.
161.       Parmameters: 与 Encoder 相似
162.
163.       '''
164.       def __init__(self,decoder_embedding_num,decoder_hidden_num,ch_
corpus_len):
165.         super().__init__()
166.         self.embedding = nn.Embedding(ch_corpus_len,decoder_embeddin
g_num)
167.         self.lstm = nn.LSTM(decoder_embedding_num,decoder_hidden_num
,batch_first=True)
168.
169.       def forward(self,decoder_input,hidden):
170.         embedding = self.embedding(decoder_input)
171.         decoder_output,decoder_hidden = self.lstm(embedding,hidden)
172.
173.         return decoder_output,decoder_hidden
174.
175.
176.     def translate(sentence):
177.       ''' Function: translate
178.
179.       根据训练好的模型 ENCODER 和 DECODER 翻译文本
180.
181.       '''
182.       global en_word_2_index,model,device,ch_word_2_index,ch_index_2
_word
183.       en_index = torch.tensor([[en_word_2_index[i] for i in sentence
]],device=device)
184.
185.       result = []
186.       encoder_hidden = model.encoder(en_index)
```

```
187.     decoder_input = torch.tensor([[ch_word_2_index["<BOS>"]]],devi
ce=device)
188.
189.     decoder_hidden = encoder_hidden
190.     while True:
191.       decoder_output,decoder_hidden = model.decoder(decoder_input,
decoder_hidden)
192.       pre = model.classifier(decoder_output)
193.
194.       w_index = int(torch.argmax(pre,dim=-1))
195.       word = ch_index_2_word[w_index]
196.
197.       if word == "<EOS>" or len(result) > 50:
198.           break
199.
200.       result.append(word)
201.       decoder_input = torch.tensor([[w_index]],device=device)
202.
203.     print("译文: ","".join(result))
204.
205.
206.   class Seq2Seq(nn.Module):
207.     '''Seq2Seq
208.
209.     Seq2Seq 模型由 ENCODER 和 DECODER 两个 LSTM 部分构成
210.
211.     '''
212.     def __init__(self,encoder_embedding_num,encoder_hidden_num,en_
corpus_len,decoder_embedding_num,decoder_hidden_num,ch_corpus_len):
213.       super().__init__()
214.       self.encoder = Encoder(encoder_embedding_num,encoder_hidden_
num,en_corpus_len)
215.       self.decoder = Decoder(decoder_embedding_num,decoder_hidden_
num,ch_corpus_len)
216.       self.classifier = nn.Linear(decoder_hidden_num,ch_corpus_len
)
217.
218.       self.cross_loss = nn.CrossEntropyLoss()
219.
220.     def forward(self,en_index,ch_index):
221.       decoder_input = ch_index[:,:-1]
222.       label = ch_index[:,1:]
223.
224.       encoder_hidden = self.encoder(en_index)
```

```
        decoder_output,_ = self.decoder(decoder_input,encoder_hidden
)

        pre = self.classifier(decoder_output)
        loss = self.cross_loss(pre.reshape(-1,pre.shape[-
1]),label.reshape(-1))

        return loss


if __name__ == "__main__":
    device = "cuda:0" if torch.cuda.is_available() else "cpu"

    with open("datas\\ch.vec","rb") as f1:
        _, ch_word_2_index,ch_index_2_word = pickle.load(f1)

    with open("datas\\en.vec","rb") as f2:
        _, en_word_2_index, en_index_2_word = pickle.load(f2)

    ch_corpus_len = len(ch_word_2_index)
    en_corpus_len = len(en_word_2_index)

    ch_word_2_index.update({"<PAD>":ch_corpus_len,"<BOS>":ch_corpu
s_len + 1 , "<EOS>":ch_corpus_len+2})
    en_word_2_index.update({"<PAD>":en_corpus_len})

    ch_index_2_word += ["<PAD>","<BOS>","<EOS>"]
    en_index_2_word += ["<PAD>"]

    ch_corpus_len += 3
    en_corpus_len = len(en_word_2_index)


    en_datas,ch_datas = get_datas(nums=2000) # 取语料库中的前2000条
进行模型训练
    encoder_embedding_num = 50
    encoder_hidden_num = 100
    decoder_embedding_num = 107
    decoder_hidden_num = 100

    batch_size = 2
    epoch = 100 # 每个样本参与训练次数
    lr = 0.001 # 学习率
```

```
265.
266.     dataset = MyDataset(en_datas,ch_datas,en_word_2_index,ch_word_
2_index)
267.     dataloader = DataLoader(dataset,batch_size,shuffle=False,colla
te_fn = dataset.batch_data_process)
268.
269.     model = Seq2Seq(encoder_embedding_num,encoder_hidden_num,en_co
rpus_len,decoder_embedding_num,decoder_hidden_num,ch_corpus_len)
270.     model = model.to(device)
271.
272.     opt = torch.optim.Adam(model.parameters(),lr = lr)
273.
274.     for e in range(epoch):
275.       for en_index,ch_index  in dataloader:
276.         loss = model(en_index,ch_index)
277.         loss.backward()
278.         opt.step()
279.         opt.zero_grad()
280.
281.       print(f"loss:{loss:.3f}")
282.
283.     while True:
284.       s = input("请输入英文: ")
285.       translate(s)
```