School of Data and Computer Science, Sun Yat-sen University

NLP - Final Project

17341111 Xuehai Liu

2020 年 1 月 14 日

目录

1 数据预处理

数据预处理主要分三大步骤:

- **定义特殊符号**

  定义一些特殊符号。其中“<pad>”加在较短序列后，直到同一 batch 内每个样本序列等长。而“<bos>”和“<eos>”符号分别表示序列的开始和结束，要求每个句子开头为“<bos>”，结尾为“<eos>”

- **分词**

  对语料集内的句子进行分词，中文选择 jieba 分词，英文使用了 NLTK 工具进行分词。

- **创建词典**

  根据上述分词结果分别为源语言和目标语言创建词典；源语言单词的索引和目标语言单词的索引相互独立

下面的代码 data_prepare.py 展示了预处理的主要过程。首先，构筑 lang 类，用于构筑词典。然后，根据词典将输入的句子转化为 tensor 向量，最后，根据得到的 tensor 向量集构造数据库 mydataset，便于后续使用 dataloader 进行并行化读取训练。

```
from __future__ import unicode_literals, print_function, division
from io import open
import  unicodedata
import string
import re
import random

import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F

device =torch.device("cuda" if torch.cuda.is_available() else "cpu")

PAD_token = 0
BOS_token = 3
EOS_token = 4
batch_size = 32
MAX_LENGTH = 150
'''
```

```
SOS_token = 0
EOS_token = 1

word2index = {}
word2count = {}
index2word = {0:"SOS",1:"EOS"}
n_words = 2
'''
#class lang: word->index and index->word
class Lang:
    def __init__(self,name):
        self.name =name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0:"<PAD>", 1:"<BOS>",2:"<EOS>"}
        self.n_words = 3
    def get_dicts(self,filename):
        with open(filename, 'r', encoding="utf-8") as f:
            for line in f.readlines():
                list = line.split()
                index = int(list[0])
                word = list[1]
                count = int(list[2])
                self.word2index[word] = index
                self.word2count[word] = count
                self.index2word = {self.word2index[word]: word for word in self.
                                                    word2index.keys()}
                self.n_words = len(self.word2index)



train_cn = "train_source_8000.txt"
train_en = "train_target_8000.txt"

def maxlen(list):
    max = 0
    for i in list:
        if(len(i)>max):
            max =  len(i)
    return  max
```

```python
def readlangs(lang1,lang2,reverse = False):
    sentences_seg = []
    with open(train_cn, "r", encoding="utf-8") as f:
        lines = f.readlines()

    for sentence in lines:
        list = sentence.split(" ")
        sentences_seg.append(list)

    maxl = maxlen(sentences_seg)
    for i, sentence in enumerate(sentences_seg):
        from copy import deepcopy

        newsen = deepcopy(sentence)
        if (len(sentence) < maxl):
            for j in range(maxl - len(sentence)):
                newsen.append("<PAD>")
        sentences_seg[i] = newsen
    lines = []
    for list in sentences_seg:
        str = ""
        for word in list:
            str += word + " "
        lines.append(str)

    with open(train_en,'r',encoding = "utf-8") as f2:
        lines2 = f2.readlines()
    pairs = [[]for i in range(len(lines))]
    for i,line in enumerate(lines):
        content = line.replace("\n", "")
        pairs[i].append(content)
    for i,line in enumerate(lines2):
        content = line.replace("\n", "")
        #content = normalizeString(content)
        pairs[i].append(content)
    input_lang = Lang(lang1)
    output_lang = Lang(lang2)
    if (reverse):
        pairs = [list(reversed(p)) for p in pairs]
    print(pairs[:10])
```

```
    return input_lang, output_lang, pairs


def prepareData(lang1,lang2,reverse = False):
    input_lang, output_lang, pairs = readlangs(lang1,lang2,reverse)
    print("read %s sentence pairs" % len(pairs))

    file_CN = "word_dict.txt"
    file_EN = "word_dict_en.txt"
    input_lang.get_dicts(file_CN)
    output_lang.get_dicts(file_EN)

    print("counted words:")
    print(input_lang.name,input_lang.n_words)
    print (output_lang.name,output_lang.n_words)

    #print(pairs[:10])
    return input_lang,output_lang,pairs

input_lang,output_lang,pairs = prepareData("Chinese","eng")


def langtoSentence(lang,sentence):
    list = []
    for word in sentence.split(' '):
        if(word != ''):
            list.append(lang.word2index[word])
    return list
    #return [lang.word2index[word] for word in sentence.split(' ')]

def tensorfromSentence(lang,sentence):
    index = langtoSentence(lang,sentence)
    return torch.tensor(index,dtype = torch.long,device = device).view(-1,1)

def tensorsFromPair(pair):
    input_tensor = tensorfromSentence(input_lang,pair[0])
    target_tensor = tensorfromSentence(output_lang,pair[1])
    return (input_tensor,target_tensor)

tensorpairs = [tensorsFromPair(pairs[i]) for i in range(len(pairs))]
```

```python
from torch.utils import data

class mydataset(data.Dataset):
    def __init__(self,pairs):
        self.x_data = [pairs[i][0] for i in range(len(pairs))]
        self.x_lens = [len(pairs[i][0]) for i in range(len(pairs))]
        self.y_data = [pairs[i][1] for i in range(len(pairs))]
        self.y_lens = [len(pairs[i][1]) for i in range(len(pairs))]

    def __len__(self):
        return len(self.x_data)
    def __getitem__(self, index):
        return self.x_data[index],self.y_data[index],self.x_lens[index],self.y_lens[
                                                                 index]

train_set = mydataset(tensorpairs)

train_loader = data.DataLoader(dataset=train_set, batch_size=32, shuffle=True,drop_last
                                                        = False)
#print(train_set[:2])
```

2 模型构建

第二部分进行模型的搭建。输入和输出都使用了 LSTM 作为神经单元，基于 pytorch 的框架。

2.1 编码器

编码器 encoder 部分，主要实现以下四点：

- 1. 根据源语言词典大小设置 word embedding 矩阵；用预训练词向量初始化
- 2.Encoder 使用双向 LSTM 或者双向 GRU;
- 3. Encoder 的初始隐藏状态选择全零或者随机向量；源句子的每个单词的 embedding 作为 Encoder 的相应时间步输入;
- 4. Encoder 返回 output 向量,其维度大小为 [src_length, batch_size, hid_dim*num_directions]; (hid_dim*num_directions) 是前向、后向隐藏状态的拼接; 该向量的第一维中第 i 个分量作为每个 batch 下源句子的第 i 时间步的隐藏状态

```
class EncoderRNN(nn.Module):

    # Input: (*), LongTensor of arbitrary shape containing the batch size
    # Output: (*, H), where * is the input shape and H = embedding_dim
    def __init__(self,in_size,hidden_size,dropout = 0.1 ):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size
        #in_size -> hidden_size
        self.embedding = nn.Embedding(in_size,hidden_size)
        self.dropout = dropout
        self.lstm = nn.LSTM(hidden_size,hidden_size,dropout = self.dropout, batch_first
                                                          = True)

    def forward(self,input, prevState,seq_length):
        embedded = self.embedding(input)
        out,state = self.lstm(embedded, prevState)
        return out,state

    def initHidden(self):
        return (torch.zeros(1, batch_size,self.hidden_size,device = device),
                torch.zeros(1, batch_size, self.hidden_size, device=device) )
```

2.2 解码器

Decoder 作为解码部分，使用了单项 LSTM，并根据目标语言的词典大小设置 word embedding 矩阵，使用了预训练词向量初始化，并实现了注意力机制 attention。

此外，decoder 从 encoder 的最后一个隐藏状态获取 hi 作为 decoder 的初始隐藏状态。Decoder 的输入有如下两种方式：

- a. Teacher Forcing: 直接使用训练数据的标准答案 (ground truth) 的对应上一项作为当前时间步的输入；
- b. Curriculum Learning: 使用一个概率 p, 随机决定选择使用 ground truth 还是前一个时间步模型生成的预测，来作为当前时间步的输入。

```
MAX_LENGTH = 95

class AttenDecoderRNN(nn.Module):
    def __init__(self,hidden_size,output_size,dropout_p = 0.1, max_length = MAX_LENGTH):
        super(AttenDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size,self.hidden_size)
        self.attn = nn.Linear(self.hidden_size *2, self.max_length)
        self.atten_combine = nn.Linear(self.hidden_size *2, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.lstm = nn.LSTM(self.hidden_size,self.hidden_size, batch_first = True)
        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)
        embedded = self.dropout(embedded)
        cat = torch.cat((embedded,hidden[0].transpose(0,1)), dim = 2)
        fc =  self.attn(cat).squeeze(1)
        attn_weights  =F.softmax( fc, dim =1 )
        attn_applied = torch.bmm(attn_weights.unsqueeze(1),encoder_outputs)

        output = torch.cat( (embedded[0],attn_applied[0]),1 )
        output = self.atten_combine(output).unsqueeze(0)
```

```
        output = F.relu(output)
        output,state = self.lstm(output,hidden)
        output = self.out(output)

        return output,hidden,attn_weights

    def initHidden(self):
        return (torch.zeros(1,batch_size,self.hidden_size,device = device),
                torch.zeros(1, batch_size, self.hidden_size, device=device) )
```

3 模型训练

模型训练部分，需要注意的是对于 decoder 的每一个时间步需要单独使用交叉熵计算损失，此外，<pad> 标记不能被计入损失，因此可以使用 pytorch 提供的交叉熵函数的 ignore_index 参数来设置。其他的部分和之前机器学习的基本方法都是一样的，先前向传播，计算损失，将梯度反向传播到网络中进行训练。训练多个 epoch，根据在验证集的表现选择最佳模型用作测试。

模型的训练代码如下：

```
teacher_forcing_ratio = 0.5
def trainIters(encoder, decoder, n_iters, print_every = 1000, plot_every = 100,
                                            learning_rate = 0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0
    plot_loss_total = 0

    encoder_optimizer = optim.SGD(encoder.parameters(),lr = learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(),lr = learning_rate)
    training_pairs = [tensorsFromPair(random.choice(pairs)) for i in range(n_iters)]
    #ignore the PAD
    criterion = nn.CrossEntropyLoss(ignore_index = PAD_token)

    train_loader = data.DataLoader(dataset=train_set, batch_size= batch_size, shuffle=
                                                True)


    loss = criterion

    print("Begin training")
```

```
for iter in range(n_iters):
    encoder_ht, encoder_ct = encoder.initHidden(batch_size)
    decoder_ht, decoder_ct = decoder.initHidden(batch_size)

    #train_step:
    for step,(input_tensor,target_tensor,lenx,leny) in enumerate(train_loader):
        encoder_optimizer.zero_grad()
        decoder_optimizer.zero_grad()

        input_length = lenx
        target_length = leny

        seq_lengths, idx = torch.tensor(lenx).sort(0, descending=True)

        input_tensor = torch.tensor(input_tensor).to(torch.int64).to(device) # (
                                            batch_size, seq_size)
        target_tensor = torch.tensor(target_tensor).to(torch.int64).to(device) # (
                                            batch_size, seq_size)
        input_tensor = input_tensor.reshape(batch_size,-1)
        target_tensor = target_tensor.reshape(batch_size,-1)
        print (input_tensor.shape)
        print(target_tensor.shape)

        #print (input_tensor.shape)  torch.Size([32, 81]) batch_size,seq_size
        #print (target_tensor.shape) torch.Size([32, 94]) batch_size,seq_size
        encoder_outputs, (encoder_ht, encoder_ct) = encoder(input_tensor, (
                                            encoder_ht, encoder_ct),
                                            seq_lengths)

        decoder_input = torch.tensor([BOS_token] * batch_size).reshape(batch_size, 1
                                            ).to(device)  # <BOS>
        decoder_ht, decoder_ct = encoder_ht, encoder_ct

        decoder_hidden = (decoder_ht, decoder_ct)

        max_dst_len = target_tensor.shape[1]
        all_decoder_outputs = torch.zeros((max_dst_len, batch_size, decoder.
                                            output_size))

        use_teacher_forcing = random.random() < teacher_forcing_ratio
```

```
        if (use_teacher_forcing):
            # teacher forcing: feed the target as the next input, else use net's own
                                                                     output
            for di in range(max_dst_len):
                decoder_output, decoder_hidden, decoder_attention = \
                    decoder(decoder_input, decoder_hidden, encoder_outputs)
                decoder_input = target_tensor[:,di].reshape(batch_size,1)  # detach
                                                                     from teacher as input
                all_decoder_outputs[di] = decoder_output.transpose(1, 0)
        else:
            for di in range(max_dst_len):
                decoder_output, decoder_hidden, decoder_attention = \
                    decoder(decoder_input, decoder_hidden, encoder_outputs)
                topv, topi = decoder_output.topk(1)
                decoder_input = topi.squeeze().detach()  # detach from history as
                                                                     input
                all_decoder_outputs[di] = decoder_output.transpose(1, 0)

        loss_f = criterion(all_decoder_outputs.permute(1, 2, 0).to(device).to(device
                                                                 ), target_tensor)
        loss = loss_f.item()

        loss.backward()

        encoder_optimizer.step()
        decoder_optimizer.step()

    print_loss_total += loss
    plot_loss_total += loss

    if(iter % print_every == 0 ):
        print_loss_avg = print_loss_total / print_every
        print_loss_total  = 0
        print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                     iter, iter / n_iters * 100, print_loss_avg))
    if iter % plot_every == 0:
        plot_loss_avg = plot_loss_total / plot_every
        plot_losses.append(plot_loss_avg)
        plot_loss_total = 0
showPlot(plot_losses)
```

4 模型测试

模型测试，在每个时间步下根据神经网络单元输出的概率来预测结果，并使用 beam-search 搜索的方法。测试时，decoder 当前时间步输入为上一时间步的输出。预测出结束符 <EOS> 时生成句子结束。

4.1 集束搜索 Beam search

贪心搜索只选择了概率最大的一个，而集束搜索则选择了概率最大的前 k 个。这个 k 值也叫做集束宽度（Beam Width），集束搜索的过程如下：

- 得到第一个输出的概率分布，选择概率最大的前 k 个.
- 前 k 个输出分别作为 Decoder 的输入，得到 k 个概率分布，然后再选择概率和最大的前 k 个序列
- 重复上述过程，最终可以得到最优的 k 个搜索结果。

此外，我调用了 nltk 的计算 bleu 库，并进行了平滑操作。实现代码如下：

```python
from nltk.translate.bleu_score import SmoothingFunction, sentence_bleu
# https://cloud.tencent.com/developer/article/1042161

def testrow(input, output, src_lang, dst_lang, encoder, decoder, beam_width=2,
                                               print_flag=False):
    input, input_len = input
    if PAD_token in input[:input_len] or PAD_token in output or len(input) > flags.
                                                    seq_size or len(output) > flags.seq_size
                                                    :
        return None, None, -1
    encoder.eval() # set in evaluation mode
    decoder.eval()

    x = torch.tensor(input).to(device).reshape(1,-1)
    seq_len = torch.tensor([input_len]).to(torch.int64).to(device)
    # encoder
    encoder_ht, encoder_ct = encoder.initHidden(1)
    encoder_outputs, (encoder_ht, encoder_ct) = encoder(x, (encoder_ht, encoder_ct),
                                                    seq_len)

    decoder_input = torch.tensor([BOS_token] * 1).reshape(1,1).to(device) # <BOS> token
    decoder_ht, decoder_ct = encoder_ht, encoder_ct # use last hidden state from encoder

    # decoder
```

```
# run through decoder one time step at a time
max_len = int(flags.seq_size*1.5)
decoder_attentions = torch.zeros(max_len,flags.seq_size)
path = [(BOS_token, 0, [])]  # input, value, words on the path
for t in range(max_len):
    new_path = []
    flag_done = True
    for decoder_input, value, indices in path:
        if decoder_input == EOS_token:
            new_path.append((decoder_input, value, indices))
            continue
        elif len(path) != 1 and decoder_input in [BOS_token, PAD_token]:
            continue
        flag_done = False
        decoder_input = torch.tensor([decoder_input]).reshape(1, 1).to(device)

        decoder_output, (decoder_ht, decoder_ct), decoder_attn = decoder(
                                                      decoder_input,
                                                                           (

                                                                           encoder_outputs

        decoder_attentions[t] = decoder_attn.transpose(1, 2).cpu().data

        softmax_output = F.log_softmax(decoder_output, dim=2)
        top_value, top_index = softmax_output.data.topk(beam_width)
        top_value = top_value.cpu().squeeze().numpy() + value
        top_index = top_index.cpu().squeeze().numpy()
        for i in range(beam_width):
            ni = int(top_index[i])
            new_path.append((ni, top_value[i], indices + [ni]))
    if flag_done:
        _, value, decoded_index = new_path[0]
        break
```

```
        else:
            new_path.sort(key=lambda x: x[1] / len(x[2]), reverse=True)  # normalization
            path = new_path[:beam_width]

    if not flag_done:
        _, value, decoded_index = path[0]
    decoded_words = []
    for ni in decoded_index:
        word = dst_lang.index2word[ni]
        decoded_words.append(word)


    pad_index = np.where(output == PAD_token)
    if len(pad_index[0]) == 0:
        pad_index = len(output)
    else:
        pad_index = pad_index[0][0]
    filter_outtext = list(filter("<PAD>".__ne__,output[:pad_index]))
    decoded_index = list(filter("<PAD>".__ne__,decoded_index))
    sm = SmoothingFunction()
    bleu = sentence_bleu([filter_outtext],decoded_index,smoothing_function=sm.method4)
    print(output[:pad_index])
    print(decoded_index)
    print("Bleu score: {}".format(bleu))
    res_words = " ".join(decoded_words)
    print("< {}".format(src_lang.getSentenceFromIndex(input)))
    print("= {}".format(dst_lang.getSentenceFromIndex(filter_outtext)))
    print("> {}".format(res_words))

    return decoded_words, decoder_attentions[:t+1, :flags.seq_size], bleu


def evaluation(dataset, src_lang, dst_lang, encoder, decoder, beam_search=False,
                                            beam_width=2):
    start_time = time.time()
    bleus = []
    for i,(input, output) in enumerate(dataset):
        input = src_lang.getSentenceIndex(input,0,False)
        input_len = len(input)
        input = src_lang.padIndex(input,flags.seq_size)
        if len(input) == 0:
```

```
            continue
        output = dst_lang.getSentenceIndex(output,0,False)
        res_words, attention, bleu = testrow((input,input_len),output,src_lang,dst_lang,
                                              encoder,decoder,beam_width=
                                              beam_width)
        if res_words != None:
            bleus.append(bleu)
    avg_bleu = np.mean(bleus)
    return avg_bleu
```

5 实验结果分析

5.1 实验配置说明

实验使用了以下几个库, 并基于 python3.6 运行。

- pytorch 1.3 + cuda 10.1
- nltk
- jieba

5.2 损失函数

使用下述代码绘制损失函数的变化情况。

```
import matplotlib.pyplot as plt
plt.switch_backend("agg")
import matplotlib.ticker as ticker
import numpy as np

def showPlot(points):
    plt.figure()
    fig,ax = plt.subplots()
    loc = ticker.MultipleLocator(base = 0.2)
    ax.yaxis.set_major_locator(loc)
    plt.plot(points)
```

结果如图:

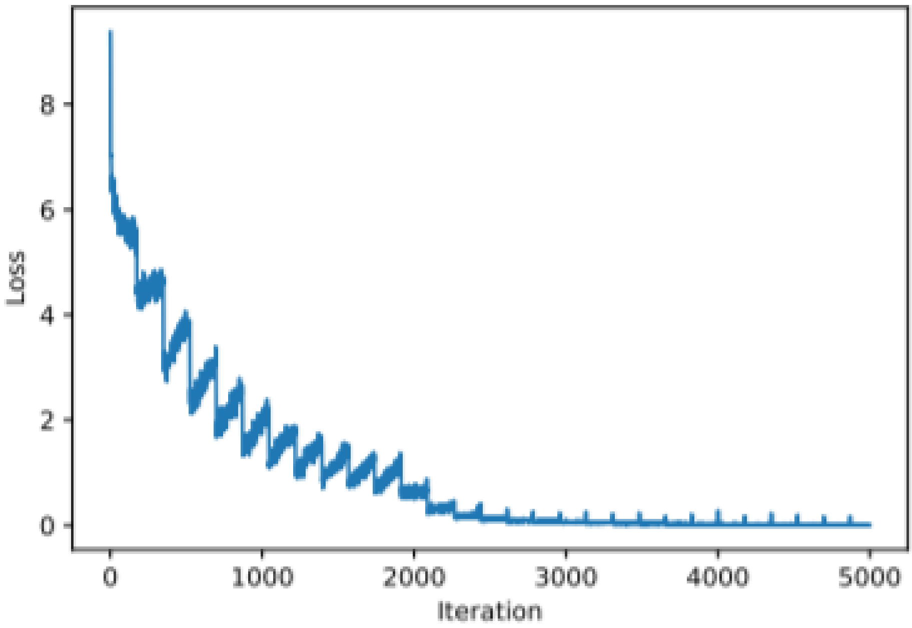

图 1: loss

可以看到，loss 函数在训练过程中不断下降并在 4000 轮左右时已经收敛，后续继续会导致过拟合。因此我选择了在 4000 轮左右模型不再出现 loss 下降的时候作为最优模型。

此外，teacher_forcing_ratio 的值很大程度上影响了模型的收敛速度。经过我的多次实验，使用模型本身的输出进行训练的效果是较差的。这将导致模型的收敛速度很慢（甚至不收敛），此外也并不能使模型获得更好的翻译效果。

5.3 bleu

最终在各个数据集上取得的 bleu 值如下：

- 训练集 0.61
- 验证集 0.14
- 测试集 0.12

可以看到，尽管训练集上取得的翻译成果看起来已经不错了，这个模型在验证集和测试集上的成绩仍然是比较差的。这个原因应当归咎于数据集的数量较少，导致模型在碰到很多之前没有或者很少见过的单词的时候，将很难将他们翻译成功。

5.4 心得体会

综上所述，本次实验可以看到是一次相当复杂的实验了，我投入了相当多的时间进行代码的编写和训练，但是也确实收益匪浅。经过本学期的两次 NLP 实验，我对人工智能在 NLP 领域上的运用熟练了很多，也能够管中窥豹地去了解一些 NLP 的前沿知识了，感谢助教和老师，能够为我们提供这样宝贵的机会。