**《深度学习理论与实践》实验报告**

**姓 名：** 徐云哲  **学 号：** 21200211047

**年 级：** 20级 **专 业：** 计算机软件与理论

1. **实验名称**

**Experiment：Text Sentiment Classification**

1. **实验要求**

1. Write the Python code of a RNN or CNN to solve the sentiment classification problem, and predict some examples.

1. **实验过程**

在本实验中，将应用预训练的词向量和含多个隐藏层的双向循环神经网络来判断一段不定长的文本序列中包含的是正面还是负面的情绪。

使用[斯坦福的IMDb数据集（Stanford’s Large Movie Review Dataset）](http://ai.stanford.edu/~amaas/data/sentiment/)作为文本情感分类的数据集。

**1、导包：**

import collections

import os

import random

import time

from tqdm import tqdm

import torch

from torch import nn

import torchtext.vocab as Vocab

import torch.utils.data as Data

import torch.nn.functional as F

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

**2、解压数据集：**

!cp -r /content/drive/My\ Drive/Text\ Sentiment\ Classification/aclImdb\_v1.zip ./  #将google云盘中的数据集压缩文件拷贝到当前运行环境

!unzip aclImdb\_v1.zip  #将数据集压缩文件解压

**3、读取数据：**

def read\_imdb(data\_root, folder='train'):

    data = []

    for label in ['pos', 'neg']:

        folder\_name = os.path.join(folder, data\_root, label)

        for file in tqdm(os.listdir(folder\_name)):

            with open(os.path.join(folder\_name, file), 'rb') as f:

                review = f.read().decode('utf-8').replace('\n', '').lower()

                data.append([review, 1 if label == 'pos' else 0])

    random.shuffle(data)

    return data

DATA\_ROOT = 'aclImdb\_v1/'

data\_root = os.path.join(DATA\_ROOT, "aclImdb")

train\_data, test\_data = read\_imdb('train', data\_root), read\_imdb('test', data\_root)

# 打印训练数据中的前五个sample

for sample in train\_data[:5]:

    print(sample[1], '\t', sample[0][:50])

**4、数据预处理：**

读取数据后，先根据文本的格式进行单词的切分，再利用 [torchtext.vocab.Vocab](https://torchtext.readthedocs.io/en/latest/vocab.html#vocab) 创建词典。

def get\_tokenized\_imdb(data):

    '''

    @params:

        data: 数据的列表，列表中的每个元素为 [文本字符串，0/1标签] 二元组

    @return: 切分词后的文本的列表，列表中的每个元素为切分后的词序列

    '''

    def tokenizer(text):

        return [tok.lower() for tok in text.split(' ')]

    return [tokenizer(review) for review, \_ in data]

def get\_vocab\_imdb(data):

    '''

    @params:

        data: 同上

    @return: 数据集上的词典，Vocab 的实例（freqs, stoi, itos）

    '''

    tokenized\_data = get\_tokenized\_imdb(data)

    counter = collections.Counter([tk for st in tokenized\_data for tk in st])

    return Vocab.Vocab(counter)

vocab = get\_vocab\_imdb(train\_data)

print('# words in vocab:', len(vocab))

词典和词语的索引创建好后，就可以将数据集的文本从字符串的形式转换为单词下标序列的形式，以待之后的使用。

def preprocess\_imdb(data, vocab):

    '''

    @params:

        data: 同上，原始的读入数据

        vocab: 训练集上生成的词典

    @return:

        features: 单词下标序列，形状为 (n, max\_l) 的整数张量

        labels: 情感标签，形状为 (n,) 的0/1整数张量

    '''

    max\_l = 500  # 将每条评论通过截断或者补0，使得长度变成500

    def pad(x):

        return x[:max\_l] if len(x) > max\_l else x + [0] \* (max\_l - len(x))

    tokenized\_data = get\_tokenized\_imdb(data)

    features = torch.tensor([pad([vocab.stoi[word] for word in words]) for words in tokenized\_data])

    labels = torch.tensor([score for \_, score in data])

    return features, labels

**5、创建数据迭代器：**

利用 [torch.utils.data.TensorDataset](https://pytorch.org/docs/stable/data.html?highlight=tensor%20dataset#torch.utils.data.TensorDataset)，可以创建 PyTorch 格式的数据集，从而创建数据迭代器。

train\_set = Data.TensorDataset(\*preprocess\_imdb(train\_data, vocab))

test\_set = Data.TensorDataset(\*preprocess\_imdb(test\_data, vocab))

batch\_size = 64

train\_iter = Data.DataLoader(train\_set, batch\_size, shuffle=True)

test\_iter = Data.DataLoader(test\_set, batch\_size)

for X, y in train\_iter:

    print('X', X.shape, 'y', y.shape)

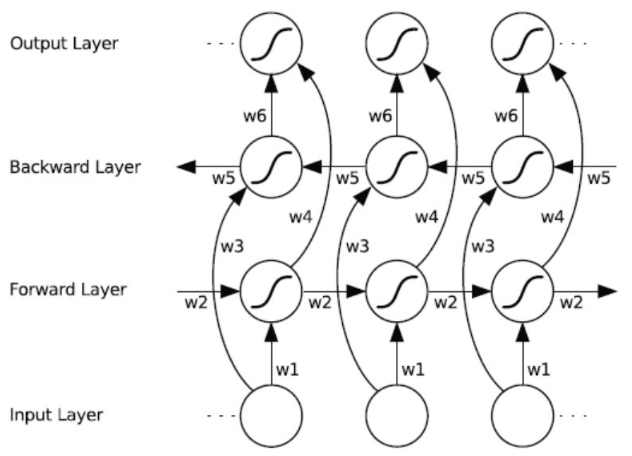
    break

print('#batches:', len(train\_iter))

# X torch.Size([64, 500]) y torch.Size([64])

# batches: 391

**6、使用双向循环神经网络进行情感分类**

****

利用 [torch.nn.RNN](https://pytorch.org/docs/stable/nn.html?highlight=rnn#torch.nn.RNN) 或 [torch.nn.LSTM](https://pytorch.org/docs/stable/nn.html?highlight=lstm#torch.nn.LSTM) ，可以很方便地实现双向循环神经网络，下面是以 LSTM 为例的代码。

class BiRNN(nn.Module):

    def \_\_init\_\_(self, vocab, embed\_size, num\_hiddens, num\_layers):

        '''

        @params:

            vocab: 在数据集上创建的词典，用于获取词典大小

            embed\_size: 嵌入维度大小

            num\_hiddens: 隐藏状态维度大小

            num\_layers: 隐藏层个数

        '''

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

        self.embedding = nn.Embedding(len(vocab), embed\_size)

        # encoder-decoder framework

        # bidirectional设为True即得到双向循环神经网络

        self.encoder = nn.LSTM(input\_size=embed\_size,

                                hidden\_size=num\_hiddens,

                                num\_layers=num\_layers,

                                bidirectional=True)

        self.decoder = nn.Linear(4\*num\_hiddens, 2) # 初始时间步和最终时间步的隐藏状态作为全连接层输入

    def forward(self, inputs):

        '''

        @params:

            inputs: 词语下标序列，形状为 (batch\_size, seq\_len) 的整数张量

        @return:

            outs: 对文本情感的预测，形状为 (batch\_size, 2) 的张量

        '''

        # 因为LSTM需要将序列长度(seq\_len)作为第一维，所以需要将输入转置

        embeddings = self.embedding(inputs.permute(1, 0)) # (seq\_len, batch\_size, d)

        # rnn.LSTM 返回输出、隐藏状态和记忆单元，格式如 outputs, (h, c)

        outputs, \_ = self.encoder(embeddings) # (seq\_len, batch\_size, 2\*h)

        encoding = torch.cat((outputs[0], outputs[-1]), -1) # (batch\_size, 4\*h)

        outs = self.decoder(encoding) # (batch\_size, 2)

        return outs

embed\_size, num\_hiddens, num\_layers = 100, 100, 2

net = BiRNN(vocab, embed\_size, num\_hiddens, num\_layers)

**7、加载预训练的词向量**

由于预训练词向量的词典及词语索引与我们使用的数据集并不相同，所以需要根据目前的词典及索引的顺序来加载预训练词向量。

cache\_dir = "/home/kesci/input/GloVe6B5429"

glove\_vocab = Vocab.GloVe(name='6B', dim=100, cache=cache\_dir)

def load\_pretrained\_embedding(words, pretrained\_vocab):

    '''

    @params:

        words: 需要加载词向量的词语列表，以 itos (index to string) 的词典形式给出

        pretrained\_vocab: 预训练词向量

    @return:

        embed: 加载到的词向量

    '''

    embed = torch.zeros(len(words), pretrained\_vocab.vectors[0].shape[0]) # 初始化为0

    oov\_count = 0 # out of vocabulary

    for i, word in enumerate(words):

        try:

            idx = pretrained\_vocab.stoi[word]

            embed[i, :] = pretrained\_vocab.vectors[idx]

        except KeyError:

            oov\_count += 1

    if oov\_count > 0:

        print("There are %d oov words." % oov\_count)

    return embed

net.embedding.weight.data.copy\_(load\_pretrained\_embedding(vocab.itos, glove\_vocab))

net.embedding.weight.requires\_grad = False # 直接加载预训练好的, 所以不需要更新它

**8、训练模型**

def evaluate\_accuracy(data\_iter, net, device=None):

    if device is None and isinstance(net, torch.nn.Module):

        device = list(net.parameters())[0].device

    acc\_sum, n = 0.0, 0

    with torch.no\_grad():

        for X, y in data\_iter:

            if isinstance(net, torch.nn.Module):

                net.eval()

                acc\_sum += (net(X.to(device)).argmax(dim=1) == y.to(device)).float().sum().cpu().item()

                net.train()

            else:

                if('is\_training' in net.\_\_code\_\_.co\_varnames):

                    acc\_sum += (net(X, is\_training=False).argmax(dim=1) == y).float().sum().item()

                else:

                    acc\_sum += (net(X).argmax(dim=1) == y).float().sum().item()

            n += y.shape[0]

    return acc\_sum / n

def train(train\_iter, test\_iter, net, loss, optimizer, device, num\_epochs):

    net = net.to(device)

    print("training on ", device)

    batch\_count = 0

    for epoch in range(num\_epochs):

        train\_l\_sum, train\_acc\_sum, n, start = 0.0, 0.0, 0, time.time()

        for X, y in train\_iter:

            X = X.to(device)

            y = y.to(device)

            y\_hat = net(X)

            l = loss(y\_hat, y)

            optimizer.zero\_grad()

            l.backward()

            optimizer.step()

            train\_l\_sum += l.cpu().item()

            train\_acc\_sum += (y\_hat.argmax(dim=1) == y).sum().cpu().item()

            n += y.shape[0]

            batch\_count += 1

        test\_acc = evaluate\_accuracy(test\_iter, net)

        print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f, time %.1f sec'

              % (epoch + 1, train\_l\_sum / batch\_count, train\_acc\_sum / n, test\_acc, time.time() - start))

由于嵌入层的参数是不需要在训练过程中被更新的，所以我们利用 filter 函数和 lambda 表达式来过滤掉模型中不需要更新参数的部分。

lr, num\_epochs = 0.01, 5

optimizer = torch.optim.Adam(filter(lambda p: p.requires\_grad, net.parameters()), lr=lr)

loss = nn.CrossEntropyLoss()

train(train\_iter, test\_iter, net, loss, optimizer, device, num\_epochs)

**8、评价模型**

def predict\_sentiment(net, vocab, sentence):

    '''

    @params：

        net: 训练好的模型

        vocab: 在该数据集上创建的词典，用于将给定的单词序转换为单词下标的序列，从而输入模型

        sentence: 需要分析情感的文本，以单词序列的形式给出

    @return: 预测的结果，positive 为正面情绪文本，negative 为负面情绪文本

    '''

    device = list(net.parameters())[0].device # 读取模型所在的环境

    sentence = torch.tensor([vocab.stoi[word] for word in sentence], device=device)

    label = torch.argmax(net(sentence.view((1, -1))), dim=1)

    return 'positive' if label.item() == 1 else 'negative'

predict\_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'great'])

# 'positive'

predict\_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'bad'])

# 'negative'