Classifiers for Sentiment Analysis

LIAO,WEN 15307110200

1 Task Description

Please design and implement a proper recurrent neural network based on LSTM or/and GRU for Sentiment Analysis. Data is available at http://deeplearning.net/tutorial/lstm.html

2 Data Acquisition

The IMDB dataset, which consists movie reviews from the IMDB website, will be the dataset on which we will perform sentiment analysis. The reviews are split into a training dataset and a test dataset, which both consist of 12500 positive samples and 12500 negative samples. Each sample appears in the form of a raw text. Our task is to train a classifier from the training dataset, and use the classifier to decide if each sample from the test dataset is positive or negative.

3 Preprocessing

To convert the data into a form that can be used as the input of a classifier, we need to vectorize each sample. NLTK, the python toolkit for natural language processing, will help us a lot with preprocessing. First, we tokenize the **raw text** with a tokenizer. Then use a lemmatizer to lemmatize each **token** so that the vocabulary size can be reduced. After that, we build a vocabulary of **lemmas** from the training set with **the most frequent 5000 lemmas**. The rest of the lemmas and the lemmas unseen in the training set will be treated as a special word, "<LFW>", i.e. the less frequent words so that the dimensionality of the **one-hot vectors** will be significantly reduced.

```python
import os
import nltk

train_pos = r"aclImdb\train\pos"
train_neg = r"aclImdb\train\neg"
test_pos = r"aclImdb\test\pos"
test_neg = r"aclImdb\test\neg"

#tokenization, lemmatization
def text(dirname):
    for val in os.walk(dirname):
        names = val[2]
    ret = []
    tokenizer = nltk.tokenize.TweetTokenizer(reduce_len=True,strip_handles=True)
    lemmatizer = nltk.WordNetLemmatizer()
    for name in names:
        file = open(dirname+"\\"+name,errors = 'ignore')
        for line in file:
            sent = []
            for word in tokenizer.tokenize(line):
                word = lemmatizer.lemmatize(word.strip(",.<>'#-$\":/*?!()").lower())
                if len(word)>0:
                    sent.append(word)
            ret.append(sent)
        file.close()
    return ret
train_pos_text = text(train_pos)
train_neg_text = text(train_neg)
test_pos_text = text(test_pos)
test_neg_text = text(test_neg)
```

```python
#shuffling
import numpy as np
from numpy import random as rd
train_data = np.array(train_pos_text + train_neg_text)
train_label = np.array([1]*len(train_pos_text) + [0]*len(train_neg_text))
index = np.arange(len(train_data))
rd.shuffle(index)
train_data = train_data[index]
train_label = train_label[index]

test_data = np.array(test_pos_text + test_neg_text)
test_label = np.array([1]*len(test_pos_text) + [0]*len(test_neg_text))
rd.shuffle(index)
test_data = test_data[index]
test_label = test_label[index]
```

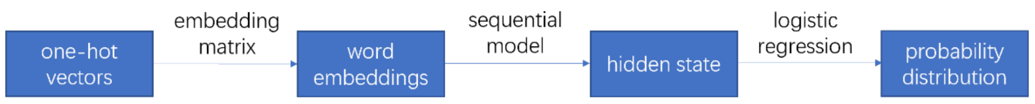
```python
#build the vocabulary
vocabulary = dict()
corpus = [train_pos_text,train_neg_text]
for texts in corpus:
    for text in texts:
        for word in text:
            if not word in vocabulary:
                vocabulary[word] = 1
            else:
                vocabulary[word] += 1
vocabulary = sorted(vocabulary.items(),key=lambda x:x[1],reverse = True)

#pick the most freqent 5000 lemmas
word2ix = {}
for word,freq in vocabulary[:5000]:
    word2ix[word] = len(word2ix)
word2ix['<LFW>'] = len(word2ix)

#one-hot vectors
def preprocess(seq,w2i):
    return torch.tensor([w2i[w] if w in w2i else w2i['<LFW>'] for w in seq],dtype = torch.long)
```

3 Model Design

Below shows the architecture of our models.



Each sample is converted as a sequence of one-hots vector in the preprocessing phase. We first multiply each one-hot vector with an embedding matrix to get a more dense vector, i.e. the word embedding. The embeddings will go through a sequential model to get the hidden state. In our model, sequential models like RNN, LSTM, GRU, BiLSTM and BiGRU will be considered. Then we will apply logistic regression classifier to the hidden state to get a Bernoulli distribution.

```python
class RNN(nn.Module):
    def __init__(self,voc_size,embed_dim,hid_dim,tag_size):
        super(RNN,self).__init__()
        self.hidden_dim = hid_dim
        self.word_embeddings = nn.Embedding(voc_size,embed_dim)
        self.rnn = nn.RNN(embed_dim,hid_dim)
        self.hidden2tag = nn.Linear(hid_dim,tag_size)
        self.hidden = self.hidden_init()
    def hidden_init(self):
        return torch.zeros(1,1,self.hidden_dim)
    def forward(self,sentence):
        embeds = self.word_embeddings(sentence)
        rnn_out,self.hidden = self.rnn(embeds.view(len(sentence),1,-1),self.hidden)
        tag_space = self.hidden2tag(rnn_out.mean(dim=0).reshape(1,-1)).reshape(1,-1)
        return F.log_softmax(tag_space,dim = 1)
```

```python
class BiRNN(nn.Module):
    def __init__(self,voc_size,embed_dim,hid_dim,tag_size):
        super(BiRNN,self).__init__()
        self.hidden_dim = hid_dim
        self.word_embeddings = nn.Embedding(voc_size,embed_dim)
        self.rnn = nn.RNN(embed_dim,hid_dim,bidirectional=True)
        self.hidden2tag = nn.Linear(hid_dim*2,tag_size)
        self.hidden = self.hidden_init()
    def hidden_init(self):
        return torch.zeros(2,1,self.hidden_dim)
    def forward(self,sentence):
        embeds = self.word_embeddings(sentence)
        rnn_out,self.hidden = self.rnn(embeds.view(len(sentence),1,-1),self.hidden)
        tag_space = self.hidden2tag(rnn_out.mean(dim=0).reshape(1,-1)).reshape(1,-1)
        return F.log_softmax(tag_space,dim = 1)
```

```python
class LSTM(nn.Module):
    def __init__(self,voc_size,embed_dim,hid_dim,tag_size):
        super(LSTM,self).__init__()
        self.hidden_dim = hid_dim
        self.word_embeddings = nn.Embedding(voc_size,embed_dim)
        self.lstm = nn.LSTM(embed_dim,hid_dim)
        self.hidden2tag = nn.Linear(hid_dim,tag_size)
        self.hidden = self.hidden_init()
    def hidden_init(self):
        return torch.zeros(1,1,self.hidden_dim),torch.zeros(1,1,self.hidden_dim)
    def forward(self,sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out,self.hidden = self.lstm(embeds.view(len(sentence),1,-1),self.hidden)
        tag_space = self.hidden2tag(lstm_out.mean(dim=0).reshape(1,-1)).reshape(1,-1)
        return F.log_softmax(tag_space,dim = 1)
```

```python
class BiLSTM(nn.Module):
    def __init__(self,voc_size,embed_dim,hid_dim,tag_size):
        super(BiLSTM,self).__init__()
        self.hidden_dim = hid_dim
        self.word_embeddings = nn.Embedding(voc_size,embed_dim)
        self.lstm = nn.LSTM(embed_dim,hid_dim,bidirectional=True)
        self.hidden2tag = nn.Linear(hid_dim*2,tag_size)
        self.hidden = self.hidden_init()
    def hidden_init(self):
        return torch.zeros(2,1,self.hidden_dim),torch.zeros(2,1,self.hidden_dim)
    def forward(self,sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out,self.hidden = self.lstm(embeds.view(len(sentence),1,-1),self.hidden)
        tag_space = self.hidden2tag(lstm_out.mean(dim=0).reshape(1,-1)).reshape(1,-1)
        return F.log_softmax(tag_space,dim = 1)
```

```
class GRU(nn.Module):
    def __init__(self,voc_size,embed_dim,hid_dim,tag_size):
        super(GRU,self).__init__()
        self.hidden_dim = hid_dim
        self.word_embeddings = nn.Embedding(voc_size,embed_dim)
        self.gru = nn.GRU(embed_dim,hid_dim)
        self.hidden2tag = nn.Linear(hid_dim,tag_size)
        self.hidden = self.hidden_init()
    def hidden_init(self):
        return torch.zeros(1,1,self.hidden_dim)
    def forward(self,sentence):
        embeds = self.word_embeddings(sentence)
        rnn_out,self.hidden = self.gru(embeds.view(len(sentence),1,-1),self.hidden)
        tag_space = self.hidden2tag(rnn_out.mean(dim=0).reshape(1,-1)).reshape(1,-1)
        return F.log_softmax(tag_space,dim = 1)
```

```
class BiGRU(nn.Module):
    def __init__(self,voc_size,embed_dim,hid_dim,tag_size):
        super(BiGRU,self).__init__()
        self.hidden_dim = hid_dim
        self.word_embeddings = nn.Embedding(voc_size,embed_dim)
        self.gru = nn.GRU(embed_dim,hid_dim,bidirectional=True)
        self.hidden2tag = nn.Linear(hid_dim*2,tag_size)
        self.hidden = self.hidden_init()
    def hidden_init(self):
        return torch.zeros(2,1,self.hidden_dim)
    def forward(self,sentence):
        embeds = self.word_embeddings(sentence)
        rnn_out,self.hidden = self.gru(embeds.view(len(sentence),1,-1),self.hidden)
        tag_space = self.hidden2tag(rnn_out.mean(dim=0).reshape(1,-1)).reshape(1,-1)
        return F.log_softmax(tag_space,dim = 1)
```

3 Model Training and Evaluation

During the training phase, we use negative loss likelihood loss (NLLLoss) as the loss function and stochastic gradient descent(SGD) as the optimization algorithm. The dimensionality of the embedding and the hidden state is set as 50 and 70 respectively. Each model is trained for 4 epoches.

```
EMBED_DIM = 50
HID_DIM = 70
TAG_SIZE = 2
model = BiLSTM(VOC_SIZE,EMBED_DIM,HID_DIM,TAG_SIZE)
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(),lr = 0.1)
for epoch in range(4):
    i = 0
    for sentence,label in zip(train_data,train_label):
        model.zero_grad()
        model.hidden = model.hidden_init()
        sent = preprocess(sentence,word2ix)
        scores = model(sent)
        loss = loss_function(scores,torch.LongTensor([label]))
        loss.backward()
        optimizer.step()
        if i%1000 == 0:
            print(i,loss)
        if (i+1)%12500 == 0:
            with torch.no_grad():
                j = 0
                for sentence,label in zip(test_data,test_label):
                    model.hidden = model.hidden_init()
                    sent = preprocess(sentence,word2ix)
                    scores = model(sent)
                    j += int(scores[0][0]<scores[0][1])==label
                print(j/len(test_data))
        i+=1
```

The performance of the models, measured by accuracy, is shown as follows[1].

Model\Epoch	0.5	1	1.5	2	2.5	3	3.5	4
RNN	0.6727	0.7210	0.7623	0.7686	0.7884	0.7980	0.8125	0.8149
BiRNN	0.7382	0.7814						
LSTM	0.7030	0.7633	0.8068	0.8176	0.8364	0.8342	0.8464	0.8477
BiLSTM	0.7638	0.8091	0.8418	0.8416	0.8591	0.8633	0.8639	0.8684
GRU	0.7248	0.7772	0.8116	0.8182	0.8307	0.8329	0.8377	0.8316
BiGRU	0.7710	0.8165	0.8291	0.8494	0.8544	0.8582		

From the chart we can make a comparison of the performances:

$$\text{BiLSTM} \approx \text{BiGRU} > \text{LSTM} \approx \text{GRU} > \text{RNN}$$

The performance of RNN is reduced by the vanishing gradient problem, which disables RNN from capturing the long distance dependency between words, thus losing important context information. LSTM and GRU improves the performance by introducing the gate mechanism to solve the vanishing gradient problem. BiLSTM and BiRNN outperform the monodirectional models for they capture the context information on both sides.

[1] Some data are missing due to the limited time and computational capacity of the PC.