Recurrent Neural Networks

本次作业是完成 NLP 当中一个简单的 task —— 语句分类（文本分类）

给定一个语句，判断有没有恶意（负面标 1，证明标 0）

请构建基于LSTM的网络结构，尽可能提高准确率，在验证集上获得至少80%准确率

Download Dataset

有三個文档，分别是 training_label.txt、training_nolabel.txt、testing_data.txt

- training_label.txt：共200000条数据，有 label 的 training data，（句子配上 0 or 1，+++$+++ 只是分隔符號，不要理它）
  - e.g., 1 +++$+++ are wtf ... awww thanks !
- training_nolabel.txt：共1178614条数据，沒有 label 的 training data（只有句子），用來做 semi-supervised learning
  - ex: hates being this burnt !! ouch
- testing_data.txt：共200000条数据，你要判斷 testing data 裡面的句子是 0 or 1

> id,text
>
> 0,my dog ate our dinner . no , seriously ... he ate it .
>
> 1,omg last day sooon n of primary noooooo x im gona be swimming out of school wif the amount of tears am gona cry
>
> 2,stupid boys .. they ' re so .. stupid !

In [3]:
```
# this is for filtering the warnings
import warnings
warnings.filterwarnings('ignore')
```

Utils

In [4]:
```
# utils.py
# 這個 block 用來先定義一些常用到的函数
import torch
import numpy as np
import pandas as pd
import  torch.optim as optim
import torch.nn.functional as F

def load_training_data(path = 'data/training_label.txt'):
    # 把 training 時需要的 data 讀進來
    # 如果是 'training_label.txt'，需要讀取 label，如果是 'training_nolabel.txt'
    if 'training_label' in path:
        with open(path, 'r', encoding='utf-8') as f:
```

```
            lines = f.readlines() # Return a list
            lines = [line.strip('\n').split(' ') for line in lines]
        x = [line[2:] for line in lines]
        y = [line[0] for line in lines]
        return x,y
    else:
        with open(path, 'r', encoding='utf-8') as f:
            lines = f.readlines()
            x = [line.strip('\n').split(' ') for line in lines]
        return x

def load_testing_data(path = 'data/testing_data.txt'):
    # 把 testing 時需要的 data 讀進來
    with open(path, 'r', encoding='utf-8') as f:
        lines = f.readlines()
        X = ["".join(line.strip('\n').split(',')[1:]).strip() for line in lines[
        X = [sen.split(' ') for sen in X] # 分词
    return X

def evaluation(outputs, labels):
    #outputs => probability (float)
    #labels => labels
    outputs[outputs>=0.5] = 1
    outputs[outputs<0.5] = 0
    correct = torch.sum(torch.eq(outputs, labels)).item()
    return correct
```

Train Word to Vector

In [5]:

```
# w2v.py
# 這個 block 是用來訓練 word to vector 的 word embedding
# 注意！這個 block 在訓練 word to vector 時是用 cpu，可能要花到 10 分鐘以上
import os
import sys
from gensim.models import word2vec
sys.path.append(os.pardir) #返回当前文件的父目录

path_prefix = './'
def train_word2vec(x):
    # 訓練 word to vector 的 word embedding
    model = word2vec.Word2Vec(x, vector_size=250, window=5, min_count=5, workers
    return model

if __name__ == "__main__":
    print("loading training data ...")
    train_x, y = load_training_data('training_label.txt')
    train_x_no_label = load_training_data('training_nolabel.txt')

    print("loading testing data ...")
    test_x = load_testing_data('testing_data.txt')

    # model = train_word2vec(train_x + train_x_no_label + test_x)
    model = train_word2vec(train_x + test_x)

    print("saving model ...")
    # model.save(os.path.join(path_prefix, 'model/w2v_all.model'))
    model.save(os.path.join(path_prefix, 'w2v_all.model'))
```

```
loading training data ...
loading testing data ...
saving model ...
```

Data Preprocess

In [14]:

```python
# preprocess.py
# 這個 block 用來做 data 的預處理
from torch import nn
from gensim.models import Word2Vec

class Preprocess():
    def __init__(self, sentences, sen_len, w2v_path="./w2v.model"):
        self.w2v_path = w2v_path
        self.sentences = sentences
        self.sen_len = sen_len
        self.idx2word = []
        self.word2idx = {}
        self.embedding_matrix = []
    def get_w2v_model(self):
        # 把之前訓練好的 word to vec 模型讀進來
        self.embedding = Word2Vec.load(self.w2v_path)
        self.embedding_dim = self.embedding.vector_size
    def add_embedding(self, word):
        # 把 word 加進 embedding，並賦予他一個隨機生成的 representation vector
        # word 只會是 "<PAD>" 或 "<UNK>"
        vector = torch.empty(1, self.embedding_dim)
        torch.nn.init.uniform_(vector)
        self.word2idx[word] = len(self.word2idx)
        self.idx2word.append(word)
        self.embedding_matrix = torch.cat([self.embedding_matrix, vector], 0)
    def make_embedding(self, load=True):
        print("Get embedding ...")
        # 取得訓練好的 Word2vec word embedding
        if load:
            print("loading word to vec model ...")
            self.get_w2v_model()
        else:
            raise NotImplementedError
        # 製作一個 word2idx 的 dictionary
        # 製作一個 idx2word 的 list
        # 製作一個 word2vector 的 list
        for i, word in enumerate(self.embedding.wv.index_to_key):
            print('get words #{}'.format(i+1), end='\r')
            #e.g. self.word2index['he'] = 1
            #e.g. self.index2word[1] = 'he'
            #e.g. self.vectors[1] = 'he' vector
            # self.word2idx[word] = len(self.word2idx)
            self.word2idx[word] = i
            self.idx2word.append(word)
            self.embedding_matrix.append(self.embedding.wv[word])
        print('')
        self.embedding_matrix = torch.tensor(self.embedding_matrix)
        # 將 "<PAD>" 跟 "<UNK>" 加進 embedding 裡面
        self.add_embedding("<PAD>")
        self.add_embedding("<UNK>")
        print("total words: {}".format(len(self.embedding_matrix)))
        return self.embedding_matrix
    def pad_sequence(self, sentence):
```

```
        # 將每個句子變成一樣的長度
        if len(sentence) > self.sen_len:
            sentence = sentence[:self.sen_len]
        else:
            pad_len = self.sen_len - len(sentence)
            for _ in range(pad_len):
                sentence.append(self.word2idx["<PAD>"])
        assert len(sentence) == self.sen_len
        return sentence
    def sentence_word2idx(self):
        # 把句子裡面的字轉成相對應的 index
        sentence_list = []
        for i, sen in enumerate(self.sentences):
            print('sentence count #{}'.format(i+1), end='\r')
            sentence_idx = []
            for word in sen:
                if (word in self.word2idx.keys()):
                    sentence_idx.append(self.word2idx[word])
                else:
                    sentence_idx.append(self.word2idx["<UNK>"])
            # 將每個句子變成一樣的長度
            sentence_idx = self.pad_sequence(sentence_idx)
            sentence_list.append(sentence_idx)
        return torch.LongTensor(sentence_list)
    def labels_to_tensor(self, y):
        # 把 labels 轉成 tensor
        y = [int(label) for label in y]
        return torch.LongTensor(y)
```

Dataset

In [7]:

```
# data.py
# 实现了 dataset 所需要的 '__init__', '__getitem__', '__len__'
# 好讓 dataloader 能使用
import torch
from torch.utils import data

class TwDataset(data.Dataset):
    """
    Expected data shape like:(data_num, data_len)
    Data can be a list of numpy array or a list of lists
    input data shape : (data_num, seq_len, feature_dim)

    __len__ will return the number of data
    """
    def __init__(self, X, y):
        self.data = X
        self.label = y
    def __getitem__(self, idx):
        if self.label is None: return self.data[idx]
        return self.data[idx], self.label[idx]
    def __len__(self):
        return len(self.data)
```

Model

In [8]:
```python
# model.py
# 這個 block 是要拿來訓練的模型，请构建基于LSTM的网络结构
import torch
from torch import nn

class LSTM_Net(nn.Module):
    def __init__(self, embedding, embedding_dim, hidden_dim, num_layers, dropout
        super(LSTM_Net, self).__init__()
        # 製作 embedding layer
        self.embedding = torch.nn.Embedding(embedding.size(0),embedding.size(1))
        self.embedding.weight = torch.nn.Parameter(embedding)
        # 是否將 embedding fix 住，如果 fix_embedding 為 False，在訓練過程中，emb
        self.embedding.weight.requires_grad = False if fix_embedding else True
        self.embedding_dim = embedding.size(1)
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        self.dropout = dropout
        self.lstm = nn.LSTM(input_size = embedding_dim, hidden_size = hidden_dim
        self.classifier = nn.Sequential(nn.Dropout(dropout),
                                        nn.Linear(hidden_dim, 1),
                                        nn.Sigmoid())

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    def forward(self, inputs):
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        inputs = self.embedding(inputs)
        x, _ = self.lstm(inputs, None)
        x = x[:, -1, :]
        x = self.classifier(x)

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return x
```

Train

In [9]:
```python
# train.py
# 這個 block 是用來訓練模型的
import torch
from torch import nn
import torch.optim as optim
import torch.nn.functional as F

def training(batch_size, n_epoch, lr, model_dir, train, valid, model, device):
    total = sum(p.numel() for p in model.parameters())
    trainable = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print('\nstart training, parameter total:{}, trainable:{}\n'.format(total, t
    model.train() # 將 model 的模式設為 train，這樣 optimizer 就可以更新 model 的
    criterion = nn.BCELoss() # 定義損失函數，這裡我們使用 binary cross entropy lo
    t_batch = len(train)
    v_batch = len(valid)
    optimizer = optim.Adam(model.parameters(), lr=lr) # 將模型的參數給 optimizer
    total_loss, total_acc, best_acc = 0, 0, 0

    for epoch in range(n_epoch):
        total_loss, total_acc = 0, 0
```

```
        # 這段做 training
        for i, (inputs, labels) in enumerate(train):
            inputs = inputs.to(device, dtype=torch.long) # device 為 "cuda"，將
            labels = labels.to(device, dtype=torch.float) # device為 "cuda"，將
            optimizer.zero_grad() # 由於 loss.backward() 的 gradient 會累加，所以
            outputs = model(inputs) # 將 input 餵給模型
            outputs = outputs.squeeze() # 去掉最外面的 dimension，好讓 outputs 可
            loss = criterion(outputs, labels) # 計算此時模型的 training loss
            loss.backward() # 算 loss 的 gradient
            optimizer.step() # 更新訓練模型的參數
            correct = evaluation(outputs, labels) # 計算此時模型的 training accu
            total_acc += (correct / batch_size)
            total_loss += loss.item()
            print('[ Epoch{}: {}/{} ] loss:{:.3f} acc:{:.3f} '.format(
                epoch+1, i+1, t_batch, loss.item(), correct*100/batch_size), end
        print('\nTrain | Loss:{:.5f} Acc: {:.3f}'.format(total_loss/t_batch, tot

        # 這段做 validation
        model.eval() # 將 model 的模式設為 eval，這樣 model 的參數就會固定住
        with torch.no_grad():
            total_loss, total_acc = 0, 0
            for i, (inputs, labels) in enumerate(valid):
                inputs = inputs.to(device, dtype=torch.long) # device 為 "cuda",
                labels = labels.to(device, dtype=torch.float) # device 為 "cuda"
                outputs = model(inputs) # 將 input 餵給模型
                outputs = outputs.squeeze() # 去掉最外面的 dimension，好讓 outpu
                loss = criterion(outputs, labels) # 計算此時模型的 validation lo
                correct = evaluation(outputs, labels) # 計算此時模型的 validatio
                total_acc += (correct / batch_size)
                total_loss += loss.item()

            print("Valid | Loss:{:.5f} Acc: {:.3f} ".format(total_loss/v_batch,
            if total_acc > best_acc:
                # 如果 validation 的結果優於之前所有的結果，就把當下的模型存下來以
                best_acc = total_acc
                #torch.save(model, "{}/val_acc_{:.3f}.model".format(model_dir,to
                torch.save(model, "{}/ckpt.model".format(model_dir))
                print('saving model with acc {:.3f}'.format(total_acc/v_batch*10
        print('---------------------------------------------')
        model.train() # 將 model 的模式設為 train，這樣 optimizer 就可以更新 mode
```

Test

In [19]:

```
# test.py
# 這個 block 用來對 testing_data.txt 做預測
import torch
from torch import nn
import torch.optim as optim
import torch.nn.functional as F

def testing(batch_size, test_loader, model, device):
    model.eval()
    ret_output = []
    with torch.no_grad():
        for i, inputs in enumerate(test_loader):
            inputs = inputs.to(device, dtype=torch.long)
            outputs = model(inputs)
            outputs = outputs.squeeze()
            outputs[outputs>=0.5] = 1 # 大於等於 0.5 為正面
```

```
            outputs[outputs<0.5] = 0 # 小於 0.5 為負面
            ret_output += outputs.int().tolist()

    return ret_output
```

Main

In [17]:

```
# main.py
import os
import torch
import argparse
import numpy as np
from torch import nn
from gensim.models import word2vec
from sklearn.model_selection import train_test_split

# 通過 torch.cuda.is_available() 的回傳值進行判斷是否有使用 GPU 的環境，如果有的話
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 處理好各個 data 的路徑
train_with_label = os.path.join(path_prefix, 'training_label.txt')
train_no_label = os.path.join(path_prefix, 'training_nolabel.txt')
testing_data = os.path.join(path_prefix, 'testing_data.txt')

w2v_path = os.path.join(path_prefix, 'w2v_all.model') # 處理 word to vec model 的

# 定義句子長度、要不要固定 embedding、batch 大小、要訓練幾個 epoch、learning rate
sen_len = 20
fix_embedding = True # fix embedding during training
batch_size = 128
epoch = 5
lr = 0.001
# model_dir = os.path.join(path_prefix, 'model/') # model directory for checkpoi
model_dir = path_prefix # model directory for checkpoint model

print("loading data ...") # 把 'training_label.txt' 跟 'training_nolabel.txt' 讀
train_x, y = load_training_data(train_with_label)
train_x_no_label = load_training_data(train_no_label)

# 對 input 跟 labels 做預處理
preprocess = Preprocess(train_x, sen_len, w2v_path=w2v_path)
embedding = preprocess.make_embedding(load=True)
train_x = preprocess.sentence_word2idx()
y = preprocess.labels_to_tensor(y)

# 製作一個 model 的對象
model = LSTM_Net(embedding, embedding_dim=250, hidden_dim=150, num_layers=1, dro
model = model.to(device) # device為 "cuda", model 使用 GPU 來訓練（餵進去的 inpu

# 把 data 分為 training data 跟 validation data（將一部份 training data 拿去當作
X_train, X_val, y_train, y_val = train_x[:180000], train_x[180000:], y[:180000],

# 把 data 做成 dataset 供 dataloader 取用
train_dataset = TwDataset(X=X_train, y=y_train)
val_dataset = TwDataset(X=X_val, y=y_val)

# 把 data 轉成 batch of tensors
train_loader = torch.utils.data.DataLoader(dataset = train_dataset,
                                           batch_size = batch_size,
```

```
                                           shuffle = True,
                                           num_workers = 0)

val_loader = torch.utils.data.DataLoader(dataset = val_dataset,
                                           batch_size = batch_size,
                                           shuffle = False,
                                           num_workers = 0)

# 開始訓練
training(batch_size, epoch, lr, model_dir, train_loader, val_loader, model, devi
```

```
loading data ...
Get embedding ...
loading word to vec model ...
get words #24694
total words: 24696
sentence count #200000
start training, parameter total:6415351, trainable:241351

[ Epoch1: 1407/1407 ] loss:0.435 acc:21.094
Train | Loss:0.49846 Acc: 75.041
Valid | Loss:0.45482 Acc: 78.055
saving model with acc 78.055
-----------------------------------------------
[ Epoch2: 1407/1407 ] loss:0.314 acc:21.875
Train | Loss:0.44490 Acc: 79.041
Valid | Loss:0.43537 Acc: 79.329
saving model with acc 79.329
-----------------------------------------------
[ Epoch3: 1407/1407 ] loss:0.439 acc:18.750
Train | Loss:0.42829 Acc: 80.068
Valid | Loss:0.43912 Acc: 79.061
-----------------------------------------------
[ Epoch4: 1407/1407 ] loss:0.378 acc:19.531
Train | Loss:0.41632 Acc: 80.748
Valid | Loss:0.42638 Acc: 80.190
saving model with acc 80.190
-----------------------------------------------
[ Epoch5: 1407/1407 ] loss:0.414 acc:19.531
Train | Loss:0.40466 Acc: 81.320
Valid | Loss:0.42176 Acc: 80.339
saving model with acc 80.339
-----------------------------------------------
```

Predict and Write to csv file

In [20]:

```
# 開始測試模型並做預測
batch_size = 128
print("loading testing data ...")
test_x = load_testing_data(testing_data)
preprocess = Preprocess(test_x, sen_len, w2v_path=w2v_path)
embedding = preprocess.make_embedding(load=True)
test_x = preprocess.sentence_word2idx()
test_dataset = TwDataset(X=test_x, y=None)
test_loader = torch.utils.data.DataLoader(dataset = test_dataset,
                                           batch_size = batch_size,
                                           shuffle = False,
                                           num_workers = 0)
print('\nload model ...')
```

```
model = torch.load(os.path.join(model_dir, 'ckpt.model'))
outputs = testing(batch_size, test_loader, model, device)

# 写到csv存档
tmp = pd.DataFrame({"id": [str(i) for i in range(len(test_x))], "label": outputs
print("save csv ...")
tmp.to_csv('predict.csv', index=False)
print("Finish Predicting")
```

```
loading testing data ...
Get embedding ...
loading word to vec model ...
get words #24694
total words: 24696
sentence count #200000
load model ...
save csv ...
Finish Predicting
```

请描述你搭建的RNN架构、训练过程和准确率如何

回答：