eNATIONAL UNIVERSITY OF HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY
FACULTY OF INFORMATION SCIENCE AND ENGINEERING

COURSE PROJECT REPORT

IE229 - ARTIFICIAL INTELLIGENCE

TOPIC:

Evaluation of popular Deep Learning architectures for Sentiment Analysis problem on the IMDB dataset.

Lecturer: Takashi Ninomiya
Huỳnh Văn Tín
Lưu Thanh Sơn
Nguyễn Thành Luân

Student group:

1. Nguyễn Văn Khoa	ID: 18520929
2. Lê Thị Thu Hằng	ID: 18520274
3. Nguyễn Thị Hồng Nhung	ID: 18521218

Ho Chi Minh City, 7/2021

LECTURE'S COMMENTS

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

PREFACE

Natural Language Processing or NLP is a field of Artificial Intelligence that gives the machines the ability to read, understand and derive meaning from human languages.

It is a discipline that focuses on the interaction between data science and human language, and is scaling to countless industries. Today, NLP is booming thanks to huge improvements in the access to data and increases in computational power, which are allowing practitioners to achieve meaningful results in areas like healthcare, media, finance, and human resources, among others.

In simple terms, NLP represents the automatic handling of natural human language like speech or text, and although the concept itself is fascinating, the real value behind this technology comes from the use cases. NLP can help humans with lots of tasks and the fields of application just seem to increase on a daily basis. Some of them are described below:

- NLP enables the recognition and prediction of diseases based on electronic health records and patient's own speech. This capability is being explored in health conditions that go from cardiovascular diseases to depression and even schizophrenia. For example, Amazon Comprehend Medical is a service that uses NLP to extract disease conditions, medications, and treatment outcomes from patient notes, clinical trial reports, and other electronic health records.
- Organizations can determine what customers are saying about a service or product by identifying and extracting information in sources like social media. This sentiment analysis can provide a lot of information about customers choices and their decision drivers.

- An inventor at IBM developed a cognitive assistant that works like a personalized search engine by learning all about users and then remind them of a name, a song, or anything they can't remember the moment they need it to.
- Companies like Yahoo and Google filter and classify user's emails with NLP by analyzing text in emails that flow through their servers and stopping spam before they even enter the inbox.
- To help to identify fake news, the NLP Group at MIT developed a new system to determine if a source is accurate or politically biased, detecting if a news source can be trusted or not.

And there are many other applications of NLP in everyday life, suggesting that this will be a core area in the future.

In this report, we will use Python programming language and Python libraries for Data Science research (eg. Numpy, Pandas, Pytorch,...) to install experimentally popular Deep Learning architecture for Sentiment Classification problem on the IMDB dataset. For easy tracking and clarification of information, the report will be divided into the following chapters:

- **Chapter 1**: Overview.
- **Chapter 2**: Dataset.
- **Chapter 3**: Experiments.
- **Chapter 4**: Summary of results.

Section 1. OVERVIEW

1.1 Problem Description

Sentiment analysis or sentiment classification falls into the broad category of text classification tasks supplied with a phrase or a list of phrases, and your classifier is supposed to tell if the sentiment behind that is positive, negative.

Consider the following phrases:

"Titanic is a great movie" -> Positive

"Titianic is not a greate movie" -> Negative

1.2 Challenges, targets

1.2.1 Challenges

- The sentiment analysis problem has many methods from many different approaches. Therefore, the comparison, evaluation and selection of appropriate methods to improve accuracy is a matter of concern and attention.
- Training the model for this task requires tons of computation costs, so we have to have suitable hardware (GPU or TPU).

1.2.2 Targets

In this report, we will build the different models that apply Convolutional Neural Network (CNN) and RNN to achieve higher accuracy than baseline model and upper 75% of accuracy.

Section 2. DATASET

2.1 Dataset description

The **IMDb Movie Reviews** dataset is a binary sentiment analysis dataset consisting of 50,000 reviews from the Internet Movie Database (IMDb) labeled as positive or negative. Negative reviews will be less than 4 out of 10 points, and positive comments will be greater than 7 out of 10 points. The dataset contains an equal number of positive and negative comments.

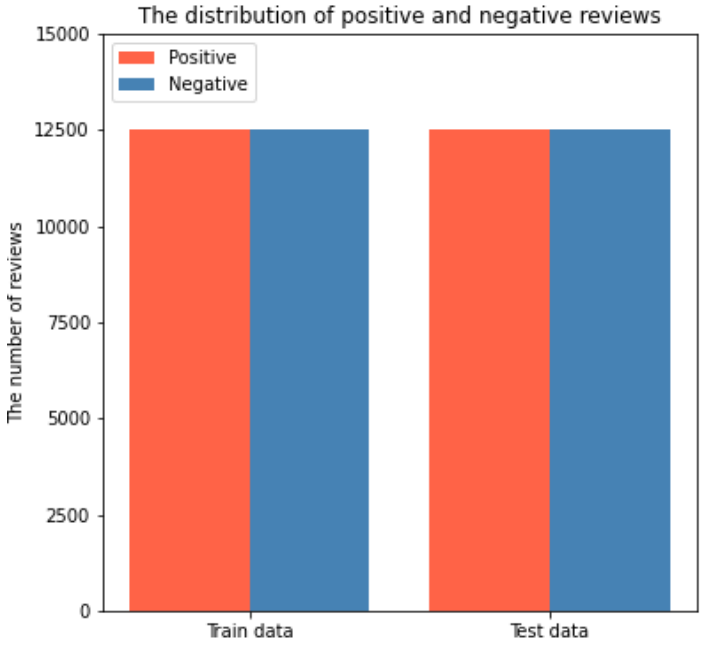

Figure: The distribution of positive and negative reviews

In this project, we divided the dataset into train and test datasets - 25000 training samples and 25000 test samples.

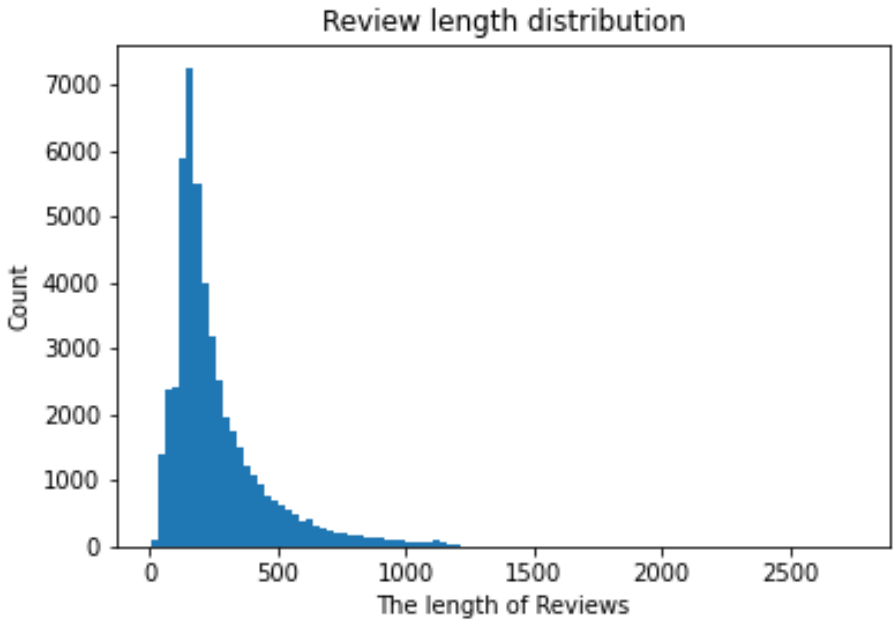

Figure: Review length distribution

2.2 Data Preparation

Text preprocessing aims to convert each sample into a numeric matrix, with each row of the matrix representing a word vector. The text preprocessing steps of this problem are as follows:

- Load dataset, tokenize the text, and sort the data in order of token column length

```
train_iter, test_iter = torchtext.datasets.IMDB(split = ('train', 'test'))
tokenizer = torchtext.data.utils.get_tokenizer('basic_english')
```

```
aclImdb_v1.tar.gz: 100%|██████████| 84.1M/84.1M [00:02<00:00, 29.6MB/s]
```

```
train_data = [(label, tokenizer(line)) for label, line in train_iter]
train_data.sort(key = lambda x: len(x[1]))
test_data = [(label, tokenizer(line)) for label, line in test_iter]
test_data.sort(key = lambda x: len(x[1]))
```

Figure: Load dataset

- Build a vocabulary set: *First*, we make a frequency list for each type of token. *Then*, rearrange the tokens in descending order of appearance frequency, and delete words with a frequency below the threshold. *Finally*, create the index by assigning an ID for each token, and we have created two variables, a vocabulary list (index to token) and a vocabulary index (token to index).

```
[7]  def make_vocab(train_data, min_freq):
       vocab = {}
       for label, tokenlist in train_data:
         for token in tokenlist:
           if token not in vocab:
             vocab[token] = 1
           vocab[token] += 1
       vocablist = [('<sos>', 0), ('<eos>', 0), ('<unk>', 0),('<pad>', 0)]
       vocabidx = {}
       for token, freq in vocab.items():
         if freq > min_freq:
           idx = len(vocablist)
           vocablist.append((token, freq))
           vocabidx[token] = idx
       vocabidx.update({'<unk>':0, '<pad>':1, '<cls>':2, '<eos>':3})
       return vocablist, vocabidx
```

```
[8]  vocablist, vocabidx = make_vocab(train_data, 5)
     print(vocablist)
     print(vocabidx)

     [('<sos>', 0), ('<eos>', 0), ('<unk>', 0), ('<pad>', 0), ('this', 75879), ('movie', 43422), ('is', 107222)
     {'this': 4, 'movie': 5, 'is': 6, 'terrible': 7, 'but': 8, 'it': 9, 'has': 10, 'some': 11, 'good': 12, 'eff
```

Figure: Create a vocabulary

- Add some special tokens to the text: Replace all the tokens not listed in the vocabulary list with the <unk> token. Add the <cls> token to the beginning of the sentences, and add the <eos> token to the end.

```
[9]  def preprocessing(data, vocab_idx):
       rr = []
       for label, tokenlist in data:
         tkl = ['<cls>']
         for token in tokenlist:
           tkl.append(token if token in vocab_idx else '<unk>')
         tkl.append('<eos>')
         rr.append((label, tkl))
       return rr
```

```
[44] train_data_prep = preprocessing(train_data, vocabidx)
     test_data_prep = preprocessing(test_data, vocabidx)
     for i in range(2):
       print(train_data_prep[i])
       print(test_data_prep[i])
```

```
('neg', ['<cls>', 'this', 'movie', 'is', 'terrible', 'but', 'it', 'has', 'some', 'good', 'effects', '.', '<eos>'])
('neg', ['<cls>', 'read', 'the', 'book', ',', 'forget', 'the', 'movie', '!', '<eos>'])
('neg', ['<cls>', 'i', 'wouldn', "'", 't', 'rent', 'this', 'one', 'even', 'on', 'dollar', 'rental', 'night', '.', '<eos>'])
('neg', ['<cls>', 'i', 'hope', 'this', 'group', 'of', 'film-makers', 'never', '<unk>', '.', '<eos>'])
```

Figure: Add special tokens to the text

- Make mini-batch: Convert to mini-batch format – from *list of label and token list pairs* to *list of N pairs of token list and labels list* (N is batch size).

```
def make_batch(data, batch_size):
  bb = []
  blabel = []
  btokenlist = []
  for label, tokenlist in data:
    blabel.append(label)
    btokenlist.append(tokenlist)
    if (len(blabel) >= batch_size):
      bb.append((btokenlist, blabel))
      blabel = []
      btokenlist = []
  if len(blabel) > 0:
    bb.append((btokenlist, blabel))
  return bb

train_data_batch = make_batch(train_data_prep, BATCHSIZE)
test_data_batch = make_batch(test_data_prep, BATCHSIZE)
for i in range(2):
  print(train_data_batch[i])
  print(test_data_batch[i])
```

```
([['<cls>', 'this', 'movie', 'is', 'terrible', 'but', 'it', 'has', 'some', 'good', 'effects', '.',
([['<cls>', 'read', 'the', 'book', ',', 'forget', 'the', 'movie', '!', '<eos>'], ['<cls>', 'i', 'h
([['<cls>', 'a', 'movie', 'best', 'summed', 'up', 'by', 'the', 'scene', 'where', 'a', 'victim', '<
([['<cls>', 'i', 'caught', 'this', 'film', 'late', 'at', 'night', 'on', 'hbo', '.', 'talk', 'about
```

Figure: Make mini-batch

- Padding: Since many texts in the mini-batch are analyzed together, the length of each text in the mini-batch must be equal. We align the length to the longest text in the mini-batch using inserting the <pad> token.

```
[ ]  def padding(data_batch):
       for tokenlists, labels in data_batch:
         maxlen = max([len(x) for x in tokenlists])
         for tkl in tokenlists:
           for i in range(maxlen - len(tkl)):
             tkl.append('<pad>')
       return data_batch

     train_data_padding = padding(train_data_batch)
     test_data_padding = padding(test_data_batch)
     for i in range(2):
       print(train_data_padding[i])
       print(test_data_padding[i])
```

```
([['<cls>', 'this', 'movie', 'is', 'terrible', 'but', 'it', 'has', 'some', 'good', 'effects', '.', '<eos>', '<pad>', '<pad>', '<pad>'
([['<cls>', 'read', 'the', 'book', ',', 'forget', 'the', 'movie', '!', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>',
([['<cls>', 'a', 'movie', 'best', 'summed', 'up', 'by', 'the', 'scene', 'where', 'a', 'victim', '<unk>', '<unk>', 'by', 'pulling', 's
([['<cls>', 'i', 'caught', 'this', 'film', 'late', 'at', 'night', 'on', 'hbo', '.', 'talk', 'about', 'wooden', 'acting', ',', 'unbeli
```

Figure: Padding

- Convert token to vocabulary index: Convert positive label to 1 and negative label to 0, and convert all the words to index using vocabulary index.

```
def word2idx(data, vocab_idx):
  rr = []
  for tokenlists, labels in data:
    id_labels = [0 if label == 'pos' else 1 for label in labels]
    idx_tokenlists = []
    for tokenlist in tokenlists:
      idx_tokenlists.append([vocab_idx[token] for token in tokenlist])
    rr.append((idx_tokenlists, id_labels))
  return rr

train_data_seq = word2idx(train_data_padding, vocabidx)
test_data_seq = word2idx(test_data_padding, vocabidx)
for i in range(2):
  print(train_data_seq[i])
  print(test_data_seq[i])
```

```
([[2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [2, 15
([[2, 1798, 27, 828, 55, 1565, 27, 5, 35, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [2, 1
([[2, 30, 5, 369, 370, 371, 192, 27, 372, 373, 30, 374, 0, 0, 192, 375, 11, 376, 377, 17, 50, 378, 379, 221, 380, 381
([[2, 15, 1238, 4, 39, 1132, 133, 25, 22, 4588, 14, 2266, 492, 3043, 144, 55, 1419, 471, 55, 7460, 3576, 14, 107, 31,
```

Figure: Convert word to index

Section 3. EXPERIMENTS

3.1 Embedding

Word embedding is one of the most popular representation of document vocabulary. A word embedding is a learned representation for text where words that have the same meaning have a similar representation.

This project use two techniques that can be used to learn a word embedding from text data:

3.1.1 Glove

The Global Vectors [1] for Word Representation, or GloVe, the algorithm is an extension to the word2vec method for efficiently learning word vectors, developed by Pennington et al. at Stanford. GloVe constructs an explicit word-context or word co-occurrence matrix using statistics across the whole text corpus.

3.1.2 Word2vec

Word2Vec [2] is a statistical method for efficiently learning a standalone word embedding from a text corpus. It was developed by Tomas Mikolov et al. at Google in 2013 as a response to make the neural-network-based training of the embedding more efficient. There are two flavors of this algorithm, namely:

- Continuous Bag-of-Words (CBOW): The CBOW model learns the embedding by predicting the current word based on its context.
- Continuous Skip-Gram Model: The continuous skip-gram model learns by predicting the surrounding words given a current word.

3.1.3 Implementation

- Download the pretrained embedding model and extract it.

```
[ ] !wget https://nlp.stanford.edu/data/glove.6B.zip

--2021-07-08 11:01:39--  https://nlp.stanford.edu/data/glove.6B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu)... 171.64.67.140
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: http://downloads.cs.stanford.edu/nlp/data/glove.6B.zip [following]
--2021-07-08 11:01:39--  http://downloads.cs.stanford.edu/nlp/data/glove.6B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.64.22
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: 'glove.6B.zip'

glove.6B.zip        100%[===================>] 822.24M  5.10MB/s    in 2m 41s

2021-07-08 11:04:21 (5.10 MB/s) - 'glove.6B.zip' saved [862182613/862182613]
```

```
[ ] !unzip /content/glove.6B.zip

Archive:  /content/glove.6B.zip
  inflating: glove.6B.50d.txt
  inflating: glove.6B.100d.txt
  inflating: glove.6B.200d.txt
  inflating: glove.6B.300d.txt
```

Figure: Download and extract pretrained Glove word vector

```
[ ] !wget -c "https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz"

--2021-07-08 11:08:04--  https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.217.42.118
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.217.42.118|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1647046227 (1.5G) [application/x-gzip]
Saving to: 'GoogleNews-vectors-negative300.bin.gz'

GoogleNews-vectors- 100%[===================>]   1.53G  65.0MB/s    in 27s

2021-07-08 11:08:32 (57.8 MB/s) - 'GoogleNews-vectors-negative300.bin.gz' saved [1647046227/1647046227]
```

```
[ ] !gzip -d GoogleNews-vectors-negative300.bin.gz
```

Figure: Download and extract pretrained Word2vec model.

- Load embedding model: With Glove word vector, we compute an index mapping words to known embeddings by parsing the data dump of pre-trained embeddings. And we load the Word2vec model to prepare to create an embedding matrix.

```
def LoadEmbeddingModel(path_emb):
  embeddings_index = {}
  with open(path_emb, encoding='utf-8') as f:
    for line in f:
      values = line.split()
      word = values[0]
      coefs = np.asarray(values[1:], dtype = 'float32')
      embeddings_index[word] = coefs
    f.close()
  print("Found %s word vector." % len(embeddings_index))
  return embeddings_index
```

```
[ ]  embeddings_index = LoadEmbeddingModel("/content/glove.6B.300d.txt")

     Found 400000 word vector.
```

Figure: Load embedding index (Glove)

```
import gensim.models.keyedvectors as word2vec
model = word2vec.KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin', binary=True)
```

Figure: Load embedding model (Word2vec)

- Create an embedding matrix by assigning the vocabulary with the pre-trained word embeddings

```
def EmbeddingMatrixCreater(word_index, emb_idx, emb_dim):
  embedding_matrix = np.zeros((len(word_index) + 1, emb_dim))
  for word, i in word_index.items():
    embedding_vector = emb_idx.get(word)
    if embedding_vector is not None:
      embedding_matrix[i] = embedding_vector
  return embedding_matrix
```

```
[ ]  embedding_matrix_glove = EmbeddingMatrixCreater(vocabidx, embeddings_index, 300)
```

Figure: Create an Glove embedding matrix

```
def EmbeddingMatrixCreater(word_index, ew2v_model, emb_dim):
  embedding_matrix = np.zeros((len(word_index) + 1, emb_dim))
  for word, i in word_index.items():
    try:
      embedding_vector = ew2v_model.wv[word]
      if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
    except:
      pass
  return embedding_matrix
```

```
embedding_matrix_word2vec = EmbeddingMatrixCreater(vocabidx, model, 300)
```

Figure: Create an Word2vec embedding matrix

- The embedding matrix to be used as weights in the embedding layer: if _weight Embedding layer is None, the weighted matrix is initialized as random values and will be updated by backpropagation. On the contrary, we load this embedding matrix into an Embedding layer.

```
torch.nn.Embedding(self.vocabsize, self.embed_dim, padding_idx=self.padding_idx, _weight=self.embed_weight)
```

Figure: Using weight for embedding layer

3.2 Baseline model

Baseline model is simple RNN model:

```
class Baseline(torch.nn.Module):
  def __init__(self, vocabsize, embed_dim, embed_weight, padding_idx):
    super(Baseline, self).__init__()

    self.vocabsize = vocabsize
    self.embed_dim = embed_dim
    self.padding_idx = padding_idx
    self.embed_weight = embed_weight

    self.emb = torch.nn.Embedding(self.vocabsize, self.embed_dim, padding_idx=self.padding_idx, _weight=self.embed_weight)
    self.l1 = torch.nn.Linear(300,300)
    self.l2 = torch.nn.Linear(300,2)

  def forward(self,x):
    e = self.emb(x)
    h = torch.zeros(e[0].size(), dtype=torch.float32).to(DEVICE)
    for i in range(x.size()[0]):
      h = F.relu(e[i] + self.l1(h))
    return self.l2(h)
```

Figure: Baseline model

```
vocabsize = len(vocabidx) + 1
embed_dim = 300
embed_weight = None
# embed_weight = weight_glove
# embed_weight = weight_w2v

padding_idx = vocabidx['<pad>']

# ------- Train ------
model = Baseline(vocabsize, embed_dim, embed_weight, padding_idx)
modelname = 'IMDB_model/baseline.model'
optimizer = torch.optim.Adam(model.parameters(), lr = LR)
loss_f = nn.CrossEntropyLoss()

train_acc_list, test_acc_list, train_loss_list, test_loss_list = train(model, optimizer, loss_f, train_data_seq, EPOCH, DEVICE, modelname)
```

Figure: Parameters in baseline model

We used a layer called nn.Embedding provided by PyTorch for word embedding, and set the embedding dimension to 300 and specified padding_idx. Set the weight for the Embedding layer in the original model at class, does not do this, but we want to try the embedding weight for this model (we build the embedding weight in the above section).

Forward propagation is calculated with the forward() function. When emb(x) is executed, the input x (all token columns in the mini-batch) is embedded together. After that, initialize the intermediate state of RNN. The RNN is calculated for the tensor **e** in order from the beginning of the token string. Finally, use Linear layer as the output.

3.3 TextCNN

3.3.1 TextCNN model

CNN (Convolutional Neural Network) [3] is a class of deep, feed-forward artificial neural networks (where connections between nodes do *not* form a cycle). It uses a variation of multilayer perceptrons designed to require minimal preprocessing. Originally invented for computer vision, CNN models have subsequently been shown to be effective for NLP and have achieved excellent results in semantic parsing.

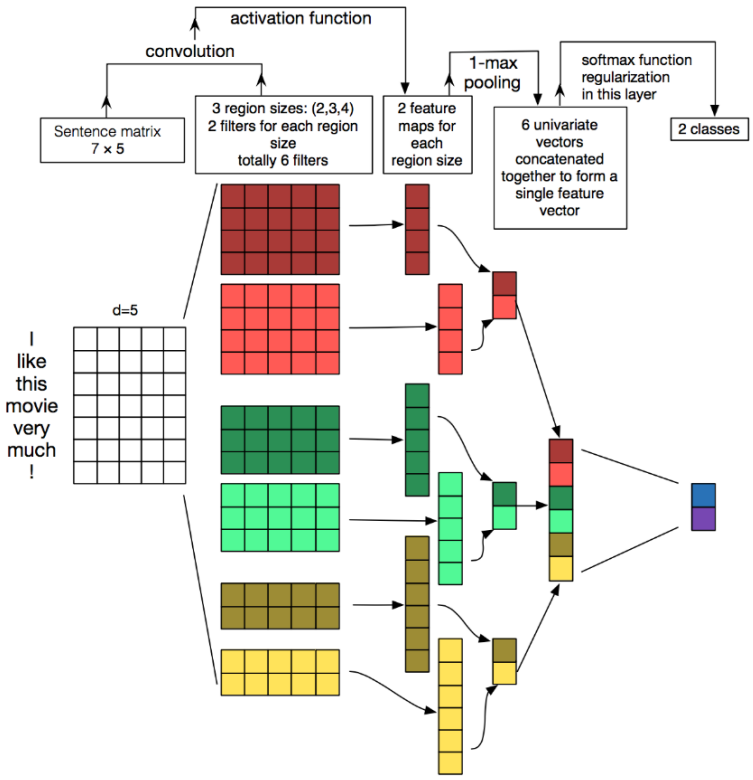

Figure: The simple CNN architecture for classifying texts

The network we will build in this project looks roughly as follows:

- The first layer embeds words into low-dimensional vectors.
- The next layer performs convolutions over the embedded word vectors using multiple filter sizes.
- We max-pool the result of the convolutional layers into a long feature vector, then add dropout regularization, and classify the result using a softmax layer.

3.3.2 Implementation

```
class textCNN(torch.nn.Module):
    def __init__(self, vocabsize, embed_dim, embed_weight, padding_idx, dropout_p, n_filter, filter_sizes = [2,3,4]):
        super(textCNN, self).__init__()
        self.vocabsize = vocabsize + 1
        self.embed_dim = embed_dim
        self.embed_weight = embed_weight
        self.padding_idx = padding_idx
        self.filter_sizes = filter_sizes
        self.n_filter = n_filter
        self.dropout_p = dropout_p

        self.emb = torch.nn.Embedding(self.vocabsize, self.embed_dim, padding_idx=self.padding_idx, _weight=self.embed_weight)

        self.conv_0 = torch.nn.Conv2d(in_channels=1,
                                      out_channels=self.n_filter,
                                      kernel_size=(self.filter_sizes[0], 300))
        self.conv_1 = torch.nn.Conv2d(in_channels=1,
                                      out_channels=self.n_filter,
                                      kernel_size=(self.filter_sizes[1], 300))
        self.conv_2 = torch.nn.Conv2d(in_channels=1,
                                      out_channels=self.n_filter,
                                      kernel_size=(self.filter_sizes[2], 300))
        # Full connected layer.
        self.fc = torch.nn.Linear(len(self.filter_sizes) * self.n_filter, 2)

        # Drop out to reduce over-fitting.
        self.dropout = torch.nn.Dropout(self.dropout_p)

    def forward(self, x):
        x = x.permute(1, 0)

        x = self.emb(x)
        x = x.unsqueeze(1)
        # Conv
        conv_0 = F.relu(self.conv_0(x).squeeze(3))
        conv_1 = F.relu(self.conv_1(x).squeeze(3))
        conv_2 = F.relu(self.conv_2(x).squeeze(3))

        pooled_0 = F.max_pool1d(conv_0, conv_0.shape[2]).squeeze(2)
        pooled_1 = F.max_pool1d(conv_1, conv_1.shape[2]).squeeze(2)
        pooled_2 = F.max_pool1d(conv_2, conv_2.shape[2]).squeeze(2)

        cat = self.dropout(torch.cat((pooled_0, pooled_1, pooled_2), dim=1))
        h = self.fc(cat)
        return h
```

Embedding layer

Convolutional layer

Fully Connected layer

Dropout

Figure: TextCNN model

- **Embedding layer:** We used the Embedding layers to convert words into word embedding. We use a pre-defined word embedding available from the library, and there are many various embeddings available open-source like *Glove* and *Word2Vec* (we have presented in the previous section). The input size "+1" because we are considering the index that refers to the padding, in this case it is the index.

- **Convolutional layer:** Three convolutional layers are not stacked. Each convolutional layer is defined by a specific kernel size; the default kernel size in this model is [2, 3, 4]. The output of each convolution is reduced using max-pooling (max-pooling layers are used in the forward() function).
- **Dropout layer:** Each tensor is concatenated to create a single tensor, and a dropout layer is added to prevent overfitting before the pooled layer to the fully connected layer.
- **Fully Connected layer:** Using a linear layer to get the probability of belonging to each class.

The ***forward*** function will take the vector of tokenized words and pass them through the embedding layer. Subsequently, each embedded sentence will be passed through each of the convolutional and max-pooling layers. Finally, the resulting vectors will be concatenated and reduced to be introduced to the linear layer.

```
vocabsize = len(vocabidx)
embed_dim = 300
embed_weight = None
padding_idx = vocabidx['<pad>']
dropout_p = 0.5
n_filters = 200

# ------- Train ------
model = textCNN(vocabsize, embed_dim, embed_weight, padding_idx, dropout_p, n_filters)
modelname = 'IMDB_model/textcnn.model'
optimizer = torch.optim.Adam(model.parameters(), lr = LR)
loss_f = nn.CrossEntropyLoss()

train_acc_lst, test_acc_lst, train_loss_lst, test_loss_lst = train(model, optimizer, loss_f, train_data_seq, EPOCH, DEVICE, modelname)
test(model, modelname, test_data_seq, DEVICE)
```

Figure: Parameter in TextCNN model

Parameters and hyper-parameters:

- vocabsize: the dictionary size of the corpus.
- emb_weight: the learnable weights of the embedding model we created previously (optional).
- emb_dim: dimension of the word vector.

- padding_idx: the index of token padding.
- dropout_p: probability of dropout layer
- n_filters: number of channels produced by the convolution layer.

3.4 LSTM & Bi-LSTM model

3.4.1 Introduction

Recurrent neural networks (RNN) are a class of neural networks that are helpful in modeling sequence data. Works on the principle of saving the output of a particular layer and feeding this back to the input to predict the output of the layer.

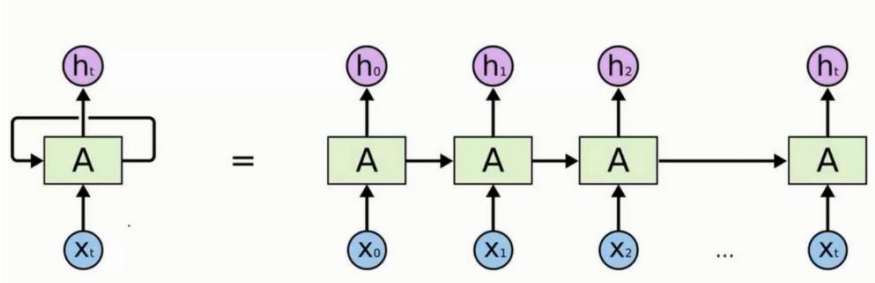

Figure: Recurrent Neural Networks

Recurrent Neural Networks enable the model of time-dependent and sequential data problems; however, RNNs suffer from the problem of vanishing gradients and make the learning of long data sequences difficult. LSTMs help solves this problem.

Long Short-Term Memory Networks (LSTMs) [4] is a special kind of Recurrent Neural Network - LSTMs enable remembering inputs over a long time because LSTMs contain information in a memory. The LSTM can read, write and delete information from its memory based on the importance of information (learned over time).

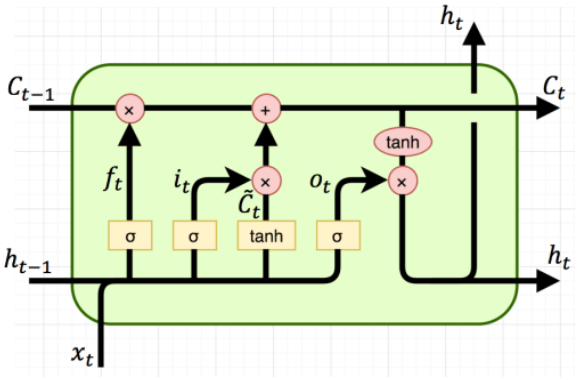

Figure: Long Short-Term Memory Networks

Bidirectional LSTMs [5][6] are an extension of traditional LSTMs that can improve model performance on sequence classification problems. Bi-LSTM allows the LSTM model to learn the input sequence both forward and backward and concatenate both interpretations.

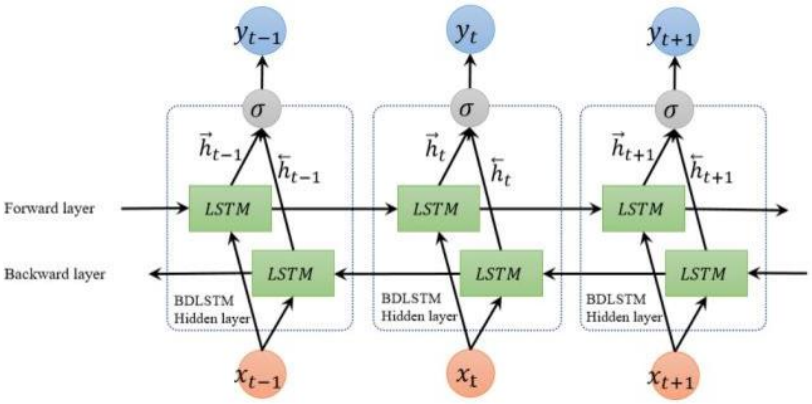

Figure: Bidirectional LSTMs

3.4.2 Implementation

There are some layers in this model:

- The first layer is an ***Embedding layer***, used to convert words into word embedding.

- ***LSTM layers*** has the input size is the embedding dimension and the output size is the hidden dimension. If the *bidirectional = True*, become a bidirectional LSTM. The num_layers is the number of recurrent layers, eg. *num_layers=2* would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results
- Then, we used ***dropout layer*** to prevent overfitting.
- The last layer is a ***Fully Connected layer*** with softmax layer to classifiy whether the review is of positive/negative sentiment.

```
class LSTM(nn.Module):
  def __init__(self,  vocab_size, embed_dim, embed_weight, padding_idx, hidden_dim, n_layers, output_size, dropout_p, bidirectional =False):
    super(LSTM,self).__init__()
    self.embed_dim = embed_dim
    self.hidden_dim = hidden_dim
    self.embed_weight = embed_weight
    self.padding_idx = padding_idx
    self.n_layers = n_layers
    self.vocab_size = vocab_size
    self.output_size = output_size
    self.dropout_p = dropout_p
    self.n_directions = 1
    self.bidirectional  = bidirectional
    if self.bidirectional :
      self.n_directions = 2

    # embedding and LSTM layers
    self.embedding = torch.nn.Embedding(self.vocab_size, self.embed_dim, padding_idx=self.padding_idx, _weight=self.embed_weight)

    # LSTM layer
    self.lstm = nn.LSTM(input_size=self.embed_dim,hidden_size=self.hidden_dim, bidirectional  = self.bidirectional ,
                        num_layers=self.n_layers, batch_first=True, dropout=((0 if self.n_layers == 1 else self.dropout_p)))

    # Drop-out
    self.dropout = nn.Dropout(self.dropout_p)

    # linear and sigmoid layer
    self.fc = nn.Linear(self.hidden_dim*self.n_directions , self.output_size)

  def forward(self,x, hidden):
    batch_size = x.size(1)
    embeds = self.embedding(x).permute(1,0,2)
    lstm_out, hidden = self.lstm(embeds, hidden)
    out = self.dropout(lstm_out)
    out = self.fc(out)
    return F.log_softmax(out[:,-1], dim=-1)

  def init_hidden(self, batch_size):
      ''' Initializes hidden state '''
      h0 = torch.zeros((self.n_layers*self.n_directions,batch_size,self.hidden_dim)).to(DEVICE)
      c0 = torch.zeros((self.n_layers*self.n_directions,batch_size,self.hidden_dim)).to(DEVICE)
      hidden = (h0,c0)
      return hidden
```

Figure: LSTM & Bi-LSTM model

The init_hidden() function initialize the hidden state used for LSTM layer. We set the initial states to zero, but the network will learn to adapt to that initial state. And we used it in train() function:

```
def train(model, optimizer, loss_f, train_data, test_data, epochs, device, modelname, batchsize):
  train_acc_lst = []
  test_acc_lst = []
  train_loss_lst = []
  test_loss_lst = []

  model.to(device)
  for epoch in range(epochs):
    loss = 0
    train_total = 0
    train_correct = 0
    model.train()
    for tokenlists, labels in train_data:
      tokenlists = torch.tensor(tokenlists, dtype=torch.int64).transpose(0,1).to(device)
      labels = torch.tensor(labels, dtype=torch.int64).to(device)

      optimizer.zero_grad()
      hidden = model.init_hidden(batchsize) # Init hidden state
      y = model(tokenlists, hidden)
      batchloss = loss_f(y, labels)
      batchloss.backward()
      optimizer.step()
      loss = loss + batchloss.item()

      train_total += len(labels)
      train_correct += (y.max(dim=1)[1] == labels).sum()

    train_acc = train_correct.item()/float(train_total)
    print('epoch', epoch, ": loss", loss)
    print("Train accuracy:", train_acc)
```

Figure: Training function in LSTM

```
vocabsize = len(vocabidx) + 1
embed_dim = 300
embed_weight = None
# embed_weight = weight_glove
# embed_weight = weight_w2v

bidirectional = False # If True, Bi-LSTM model

padding_idx = vocabidx['<pad>']
hidden_dim = 300
dropout_p = 0.5
n_layers = 2
output_size = 2

# ------- Train ------
model = LSTM(vocabsize, embed_dim, embed_weight, padding_idx, hidden_dim, n_layers, output_size, dropout_p)
modelname = 'IMDB_model/lstm.model'
optimizer = torch.optim.Adam(model.parameters(), lr = LR)
loss_f = nn.CrossEntropyLoss()

lstm_train_acc_lst, lstm_test_acc_lst, lstm_train_loss_lst, lstm_test_loss_lst = train(model, optimizer, loss_f, train_data_seq, test_data_seq, EPOCH, DEVICE, modelname, BATCHSIZE)
test(model, modelname, test_data_seq, DEVICE, BATCHSIZE)
```

Figure: Parameter in LSTM & Bi-LSTM model

Parameter in LSTM model:

- vocabsize: the dictionary size of the corpus.
- embed_dim: dimension of the word vector.

- embed_weight: the learnable weights of the embedding model we created previously (optional).
- bidirectional: if bidirectional is True, becomes a bidirection LSTM.
- padding_idx: Index of padding token.
- hidden_dim: the number of features in the hidden state of LSTM layer.
- dropout_p: probability of Dropout layer
- n_layers: the number of recurrent layers in LSTM layer.
- output_size: the number of classes.

Section 4. SUMMARY OF RESULTS

To install the program for the task, we use the PyTorch framework. Setting hyper-parameter: EPOCH = 10, LR = 0.00001, BATCHSIZE = 32. We train the networks using the ***Adam optimizer*** by minimizing ***the cross-entropy,*** and results shown in table below:

	No Embedding	Glove Embedding	Word2Vec Embedding
Baseline	0.676577	0.673415	**0.797415**
TextCNN	0.827985	0.831866	**0.841869**
LSTM	0.794254	0.795535	**0.860435**
Bi-LSTM	0.793613	0.812060	**0.857595**

Table: The result table

Most of the new models we installed achieve better than the Baseline model and achieve more than expected accuracy (except the Baseline model with Glove embedding). The best performing model is LSTM with Word2Vec embedding with accuracy 86.04%.

The change of loss during learning:

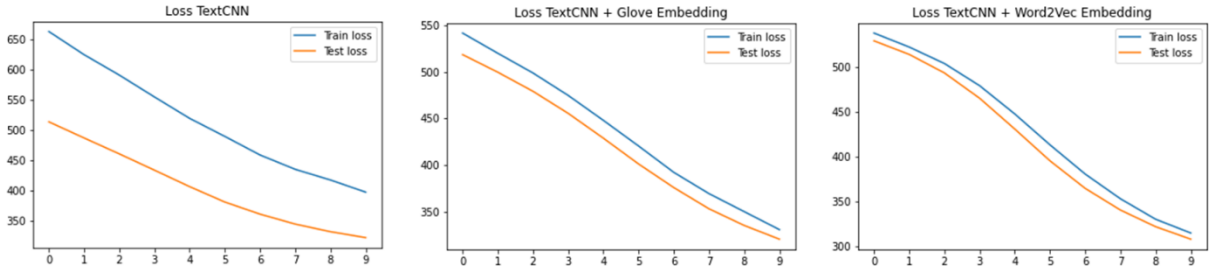

Figure: Loss graph in TextCNN model

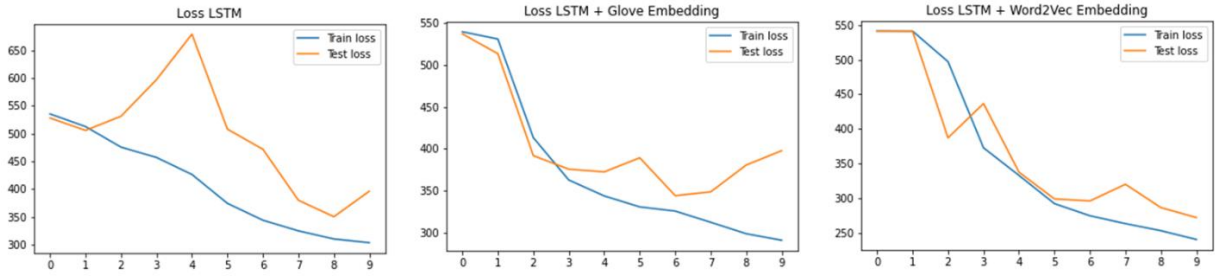

Figure: Loss graph in LSTM model

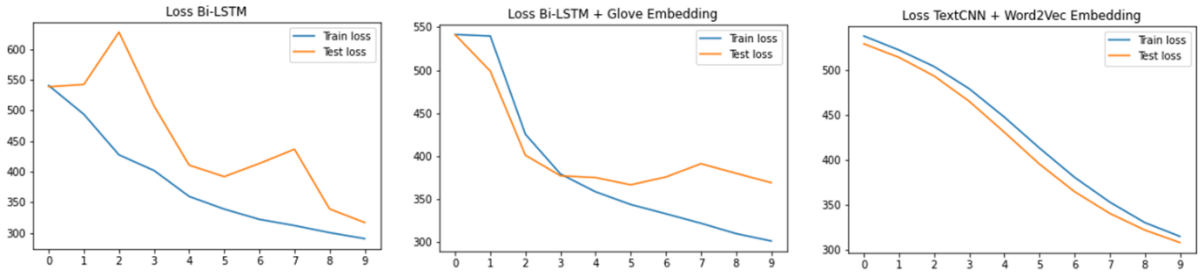

Figure: Loss graph in Bi-LSTM model

Section 5. CONCLUSION

We achieved our target of building models for the this task of reaching over 75% on accuracy metric. LSTM model with Word2Vec embedding achieve 86.04% and is the best model. In future, we aim to use BERT model for solving this task.

REFERENCES

[1] J. Pennington, R. Socher, and C. D. Manning, 'Glove: Global vectors for word representation', in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, 'Efficient estimation of word representations in vector space', *arXiv preprint arXiv:1301.3781*, 2013.

[3] A. Rakhlin, 'Convolutional neural networks for sentence classification', *GitHub*, 2016.

[4] S. Hochreiter and J. Schmidhuber, 'LSTM can solve hard long time lag problems', *Advances in neural information processing systems*, pp. 473–479, 1997.

[5] A. Graves, S. Fernández, and J. Schmidhuber, 'Bidirectional LSTM networks for improved phoneme classification and recognition', in *International conference on artificial neural networks*, 2005, pp. 799–804.

[6] A. Graves and J. Schmidhuber, 'Framewise phoneme classification with bidirectional LSTM and other neural network architectures', *Neural networks*, vol. 18, no. 5–6, pp. 602–610, 2005.