NAME: Gaurav Raut

ID : 2038584

In [1]:
```
import os
import numpy as np
import pandas as pd
import pickle
import random
from tensorflow.keras.preprocessing.text import one_hot
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from keras.layers.embeddings import Embedding
from tensorflow.keras.preprocessing.text import Tokenizer
import tensorflow as tf
```

In [2]:
```
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

In [3]:
```
imdb_dir = './aclImdb/'
print(os.listdir(imdb_dir))
```

```
['.DS_Store', 'test', 'train']
```

In [4]:
```
print(os.listdir("./aclImdb/train/"))
```

```
['.DS_Store', 'neg', 'pos', 'urls_neg.txt', 'urls_pos.txt']
```

In [5]:
```
print(os.listdir("./aclImdb/test/"))
```

```
['.DS_Store', 'neg', 'pos', 'urls_neg.txt', 'urls_pos.txt']
```

In [6]:
```
# Assuming that there are maximum 10000 unique words across
# the reviews in the datasets
vocab_size = 10000

#specifies maximum number of words to read from a review
max_length = 100
```

In [7]:
```
#Reading data into memory along with the labels.

def get_imdb_data(datatype):
    imdb_dir='aclImdb/'

    #setting dataset directory path
    base_dir = os.path.join(imdb_dir, datatype)
    texts=[]
    labels=[]

    text_label = []

    label_value = {'neg':0, 'pos':1}

    for label_type in ['neg','pos']:
        dir_name = os.path.join(base_dir, label_type)

        for fname in os.listdir(dir_name):
            f =open(os.path.join(dir_name,fname), encoding='utf8')
```

```
            text_label.append((f.read(), label_value[label_type]))
            f.close()

    print(len(text_label))

    return text_label
```

In [8]:
```
def combine_data(*args):
    text_label = []

    for each in args:
        text_label.extend(each)

    return text_label
```

In [9]:
```
text_label = combine_data(get_imdb_data("train"), get_imdb_data("test"))
random.shuffle(text_label)

texts = []
labels = []

for t, l in text_label:
    texts.append(t)
    labels.append(l)
```

```
25000
25000
```

In [10]:
```
texts_train = np.array(texts[:25000])
labels_train = np.array(labels[:25000])
texts_test = np.array(texts[25000:])
labels_test = np.array(labels[25000:])
```

In [11]:
```
np.shape(labels_test)
```

Out[11]:
```
(25000,)
```

In [12]:
```
encoded_training_documents = [one_hot(sentence, vocab_size)
                              for sentence in texts_train]
padded_training_documents = pad_sequences(
                                encoded_training_documents,
                                maxlen=max_length,
                                padding='post')
```

In [13]:
```
encoded_test_documents = [one_hot(sentence, vocab_size)
                              for sentence in texts_test]
# encoded_test_documents = [one_hot(data, len(data))]
padded_test_documents = pad_sequences(
                                encoded_test_documents,
                                maxlen=max_length,
                                padding='post')
```

In [14]:
```
model = Sequential()
model.add(Embedding(vocab_size,100, input_length = max_length))
model.add(Flatten())
model.add(Dense(64,activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
model.compile(optimizer ='adam',loss='binary_crossentropy',metrics=['acc'])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding (Embedding)        (None, 100, 100)          1000000   
_________________________________________________________________
flatten (Flatten)            (None, 10000)             0         
_________________________________________________________________
dense (Dense)                (None, 64)                640064    
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65        
=================================================================
Total params: 1,640,129
Trainable params: 1,640,129
Non-trainable params: 0
_________________________________________________________________
```

The model chosen here is Sequential model. Similarly 50% data is used for training and 50% for testing After that embedding layer is added. Turns positive integers (indexes) into dense vectors of fixed size. e.g. [14], [20]] [10.25, 0.1], [0.6, -0.2]] This layer can only be used as the first layer in a model. Embedding layer is taken as the first layer of the model. input_dim attribute determines the size of vocabulary. output_dim attribute determines the dimension of the dense embedding. max_length attribute determines the length of input sequences and is needed when connecting flatten and dense layers. variable vocab_size is taken input_dim attribute, 100 is taken as output_dim attribute and variable max_length is taken as input_length attribute. the dense outputs cannot be computed). A flatten layer with 10000 neurons is added. This layer flattens the input A hidden layer with 64 neurons and relu activation is added to introduce non linearity to the training process and avoid vanishing or exploding gradient descend. after that a single neuron is added as output layer with sigmoid activation Since, the problem is binary classification, the model uses binary crossentropy as it's loss function and adam as it's optimizer.RMSprop is chosen as optimizer as it is the de facto optimizer in case of sequential models as it helps to optimize the model more than other optimizers. The major advantage of using RMSprop optimzer is its adaptive learning rate This is in contrast to the SGD algorithm. SGD maintains a single learning rate throughout the network learning process. We can always change the learning rate using a scheduler whenever learning plateaus. But we need to do that through manual coding. The sigmoid function becomes asymptotically either zero or one which means that the gradients are near zero for inputs with a large absolute value.This makes the sigmoid function prone to vanishing gradient issues which the ReLU does not suffer as much. ReLU has an attribute that can be viewed as both positive and negative depending on how you look at it. Because ReLU is essentially a function that returns zero for negative inputs and identity for positive inputs, it's easy to get zeros as outputs, resulting in dead neurons. ReLU has an attribute which can be seen both as positive and negative depending on which angle you are approaching it. The fact that ReLU is effectively a function that is zero for negative inputs and identity for positive inputs means that it is easy to have zeros as outputs and this leads to dead neurons. Dead neurons, on the other hand, may appear to be a bad thing, but in many cases, they aren't because they allow for sparsity. In some ways, the ReLU is analogous to an Ll regularization, in that it reduces some weights to zero, resulting in a sparse solution. Sparsity is something that, while it often leads to better model generalization, it can also have a negative effect on performance.A good practice when using ReLU is to initialize the bias to a small number rather than zero so that you avoid dead neurons at the beginning of the training of the neural network which might prevent training in general.

In [15]:
```
history_one = model.fit(padded_training_documents, labels_train, epochs=10, verbose = 1
```

```
Epoch 1/10
782/782 [==============================] - 10s 13ms/step - loss: 0.5370 - acc: 0.6956
Epoch 2/10
782/782 [==============================] - 10s 13ms/step - loss: 0.1107 - acc: 0.9647
Epoch 3/10
782/782 [==============================] - 10s 13ms/step - loss: 0.0157 - acc: 0.9955
Epoch 4/10
782/782 [==============================] - 10s 13ms/step - loss: 0.0014 - acc: 0.9997
Epoch 5/10
782/782 [==============================] - 10s 13ms/step - loss: 1.5048e-04 - acc: 1.000
0
Epoch 6/10
782/782 [==============================] - 10s 13ms/step - loss: 6.1830e-05 - acc: 1.000
0 1s - loss: 6.216
Epoch 7/10
782/782 [==============================] - 10s 13ms/step - loss: 3.9526e-05 - acc: 1.000
0 4s - loss: 4.0549e-05 - ac - ETA: 3s - lo
Epoch 8/10
782/782 [==============================] - 10s 13ms/step - loss: 2.5271e-05 - acc: 1.000
0
Epoch 9/10
782/782 [==============================] - 10s 13ms/step - loss: 1.6547e-05 - acc: 1.000
0 1s -
Epoch 10/10
782/782 [==============================] - 11s 14ms/step - loss: 1.0787e-05 - acc: 1.000
0
```

In [16]:
```
loss, accuracy = model.evaluate(padded_test_documents,labels_test,verbose=1)
```

```
782/782 [==============================] - 2s 2ms/step - loss: 1.0709 - acc: 0.8185
```

In [17]:
```
accuracy
```

Out[17]:
```
0.8185200095176697
```

In [18]:
```
print('Accuracy for 25000 reviews : %f' % (accuracy*100),'\n','Loss for 25000 reviews
```

```
Accuracy for 25000 reviews : 81.852001
 Loss for 25000 reviews    : 1.0708569288253784
```

In [19]:
```
data = "The movie is good. bad  and very bad bad "

pre_encoded_documents = [one_hot(data, len(data))]
pre_padded_documents = pad_sequences(
                                pre_encoded_documents,
                                maxlen=max_length,
                                padding='post')
```

In [20]:
```
def predict_output(padded_documents):
    result = model.predict(pre_padded_documents)
    print(result)

    if result > 0.5:
        return "pos"
```

```
    else:
        return "neg"
```

In [21]:
```
# np.array(predict_output)
print(predict_output(pre_padded_documents))
```

```
[[0.00093126]]
neg
```

Using 5000 movie reviews from the dataset

In [22]:
```
updated_texts_train = np.array(texts[:5000])
updated_labels_train = np.array(labels[:5000])
updated_texts_test = np.array(texts[:5000])
updated_labels_test = np.array(labels[:5000])
```

In [23]:
```
np.shape(updated_labels_test)
```

Out[23]:
```
(5000,)
```

In [24]:
```
updated_encoded_training_documents = [one_hot(sentence, vocab_size)
                              for sentence in updated_texts_train]
updated_padded_training_documents = pad_sequences(
                                updated_encoded_training_documents,
                                maxlen=max_length,
                                padding='post')
```

In [25]:
```
updated_encoded_test_documents = [one_hot(sentence, vocab_size)
                              for sentence in updated_texts_test]
# encoded_test_documents = [one_hot(data, len(data))]
updated_padded_test_documents = pad_sequences(
                                updated_encoded_test_documents,
                                maxlen=max_length,
                                padding='post')
```

In [26]:
```
model_two = Sequential()
model_two.add(Embedding(vocab_size,100, input_length = max_length))
model_two.add(Flatten())
model_two.add(Dense(64,activation = 'relu'))
model_two.add(Dense(1, activation = 'sigmoid'))
model_two.compile(optimizer ='adam',loss='binary_crossentropy',metrics=['acc'])
model_two.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, 100, 100)          1000000   
_________________________________________________________________
flatten_1 (Flatten)          (None, 10000)             0         
_________________________________________________________________
dense_2 (Dense)              (None, 64)                640064    
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 65        
=================================================================
Total params: 1,640,129
Trainable params: 1,640,129
Non-trainable params: 0
_________________________________________________________________
```

In [27]:
```
history_two = model_two.fit(updated_padded_training_documents, updated_labels_train, ep
```

```
Epoch 1/10
157/157 [==============================] - 3s 13ms/step - loss: 0.6764 - acc: 0.5599
Epoch 2/10
157/157 [==============================] - 2s 13ms/step - loss: 0.1290 - acc: 0.9587
Epoch 3/10
157/157 [==============================] - 2s 13ms/step - loss: 0.0059 - acc: 1.0000
Epoch 4/10
157/157 [==============================] - 2s 13ms/step - loss: 0.0014 - acc: 1.0000
Epoch 5/10
157/157 [==============================] - 2s 13ms/step - loss: 7.4866e-04 - acc: 1.0000
Epoch 6/10
157/157 [==============================] - 2s 13ms/step - loss: 4.9558e-04 - acc: 1.0000
Epoch 7/10
157/157 [==============================] - 2s 12ms/step - loss: 3.3907e-04 - acc: 1.0000
Epoch 8/10
157/157 [==============================] - 2s 13ms/step - loss: 2.7014e-04 - acc: 1.0000
Epoch 9/10
157/157 [==============================] - 2s 12ms/step - loss: 1.8529e-04 - acc: 1.0000
Epoch 10/10
157/157 [==============================] - 2s 12ms/step - loss: 1.4498e-04 - acc: 1.0000
```

In [28]:
```
loss_two, accuracy_two = model_two.evaluate(updated_padded_test_documents, updated_labe
accuracy_two
```

```
157/157 [==============================] - 0s 2ms/step - loss: 1.2101e-04 - acc: 1.0000
```

Out[28]: 1.0

In [29]:
```
print('Accuracy for 5000 reviews : %f' % (accuracy_two*100),'\n','Loss for 5000 reviews
```

```
Accuracy for 5000 reviews : 100.000000
 Loss for 5000 reviews    : 0.00012101118772989139
```

In [30]:
```
data = "The movie is good. bad  and very bad bad "

updated_pre_encoded_documents = [one_hot(data, len(data))]
updated_pre_padded_documents = pad_sequences(
                                updated_pre_encoded_documents,
                                maxlen=max_length,
                                padding='post')
def updated_predict_output(padded_documents):
    result = model_two.predict(updated_pre_padded_documents)
    print(result)

    if result > 0.5:
        return "pos"

    else:
        return "neg"
```

In [31]:
```
print(updated_predict_output(updated_pre_padded_documents))
```

```
[[0.62815475]]
pos
```

Since, there is variation of data to train and test the model, the model may mispredict some of the test and train data. In the case of model that uses 20 percent of the data (mode12), the model may suffer from oversampling of a class and undersampling of another as the data are not split randomly rather the data are chosen according to index. Since, the variation in input while training the data is very less for mode12, it can be said that although mode12 may have high accuracy, it may still mispredict in the real world scenario.

In [ ]:

Part 2: Classification using Pre-Trained Model

In [32]:
```
maxlen = 100 #truncate revies over 100 words
training = 10000 # trains on 10000 samples, we selected small training which is able to
validation = 10000 # validates on 10000 samples
max_words = 10000 # considers only the top 10000 wordsin the dataset
```

I am going to split a large sample of text into words because this sentiment analysis is the part of Natural Lanaguge processing where each word needs to be captured and subjected to further analysis like classifying and counting them for a particular sentiment. For example: Text = "God is Great! I won a lottery." After tokenizing this text tokenizing_word = ['God' , 'is', 'Great', '!', 'I', 'won', 'a', 'lottery', '.']

In [33]:
```
tokenizer= Tokenizer(num_words = max_words)
tokenizer.fit_on_texts(texts_train)
sequences = tokenizer.texts_to_sequences(texts_train)
word_index = tokenizer.word_index
print('Total unique tokens : ', len(word_index))
```

```
Total unique tokens :  90724
```

Tokenizer converts each text in a sentence to a sequence of integer. num_words attribut determines the maximum number of words to keep. fit_on_texts updates internal vocabulary based on a list of texts. texts_to_sequences converts the input to sequence of integer. tokenizer. (word_index + 1) returns the total number of words

In [34]:
```
data = pad_sequences(sequences , maxlen = maxlen)
labels = np.asarray(labels_train)
print('Data shape  :', data.shape)
print('Label shape :', labels.shape)
```

```
Data shape  : (25000, 100)
Label shape : (25000,)
```

In [35]:
```
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]
```

In [36]:
```
X_train = data[:training]
y_train = labels[:training]
X_test = data[training:training + validation]
y_test = labels[training:training + validation]
```

In [37]:
```
X_train.shape
```

Out[37]:
```
(10000, 100)
```

In [38]:
```
X_test.shape
```

Out[38]:
```
(10000, 100)
```

In [39]:
```
y_train.shape
```

Out[39]: (10000,)

In [40]:
```
y_test.shape
```

Out[40]: (10000,)

In [46]:
```
## Loading embeding information into memory
file = open('preTrained_we.txt', encoding = 'utf8')
embedding_index =  dict ()
for line in file:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:],dtype = 'float32')
    embedding_index[word] = coefs

file.close()

print('Total loaded word vectors',len(embedding_index))
```

```
Total loaded word vectors 400000
```

In [47]:
```
## limiting words into embedding files

embedding_dim = 100 ## each word vector will be size of 100
embedding_matrix = np.zeros((max_words, embedding_dim))
for word , i in word_index.items():
    if i < max_words:
        embedding_vector = embedding_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] =  embedding_vector
```

In [48]:
```
model_three = Sequential()
model_three.add(Embedding(vocab_size, 100 , input_length = max_length))
model_three.add(Flatten())
model_three.add(Dense(64,activation = 'relu'))
model_three.add(Dense(1, activation =  'sigmoid'))
model_three.layers[0].set_weights([embedding_matrix])
model_three.layers[0].trainable = False ## No more training as word embeddings
model_three.compile(optimizer ='adam',loss='binary_crossentropy',metrics=['acc'])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 100, 100)          1000000
_________________________________________________________________
flatten (Flatten)            (None, 10000)             0
_________________________________________________________________
dense (Dense)                (None, 64)                640064
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65
=================================================================
Total params: 1,640,129
Trainable params: 1,640,129
Non-trainable params: 0
_________________________________________________________________
```

In [49]:
```
history_three = model_three.fit(X_train, y_train, epochs=10,  batch_size = 32, validati
```

```
Epoch 1/10
```

```
313/313 [==============================] - 3s 7ms/step - loss: 0.7042 - acc: 0.6062 - va
l_loss: 0.5932 - val_acc: 0.6852
Epoch 2/10
313/313 [==============================] - 2s 6ms/step - loss: 0.4972 - acc: 0.7724 - va
l_loss: 0.5792 - val_acc: 0.7038
Epoch 3/10
313/313 [==============================] - 2s 6ms/step - loss: 0.4032 - acc: 0.8340 - va
l_loss: 0.5944 - val_acc: 0.7081
Epoch 4/10
313/313 [==============================] - 2s 6ms/step - loss: 0.3277 - acc: 0.8669 - va
l_loss: 0.6330 - val_acc: 0.7039
Epoch 5/10
313/313 [==============================] - 2s 7ms/step - loss: 0.2518 - acc: 0.9132 - va
l_loss: 0.6919 - val_acc: 0.6969
Epoch 6/10
313/313 [==============================] - 2s 7ms/step - loss: 0.2201 - acc: 0.9240 - va
l_loss: 0.7682 - val_acc: 0.6949
Epoch 7/10
313/313 [==============================] - 2s 6ms/step - loss: 0.1837 - acc: 0.9370 - va
l_loss: 0.8739 - val_acc: 0.6814
Epoch 8/10
313/313 [==============================] - 2s 6ms/step - loss: 0.1469 - acc: 0.9516 - va
l_loss: 0.9610 - val_acc: 0.6816
Epoch 9/10
313/313 [==============================] - 2s 7ms/step - loss: 0.1088 - acc: 0.9704 - va
l_loss: 0.9944 - val_acc: 0.6829
Epoch 10/10
313/313 [==============================] - 2s 7ms/step - loss: 0.0858 - acc: 0.9799 - va
l_loss: 1.0924 - val_acc: 0.6808
```

In [50]:

```python
loss_three, accuracy_three = model_three.evaluate(padded_test_documents,labels_test,ver
print('Accuracy for Pretrained data : %f' % (accuracy_three*100),'\n','Loss for Pretrai
```

```
782/782 [==============================] - 2s 2ms/step - loss: 1.7633 - acc: 0.5120
Accuracy for Pretrained data : 51.204002
 Loss for Pretrained data    : 1.7633371353149414
```

In [51]:

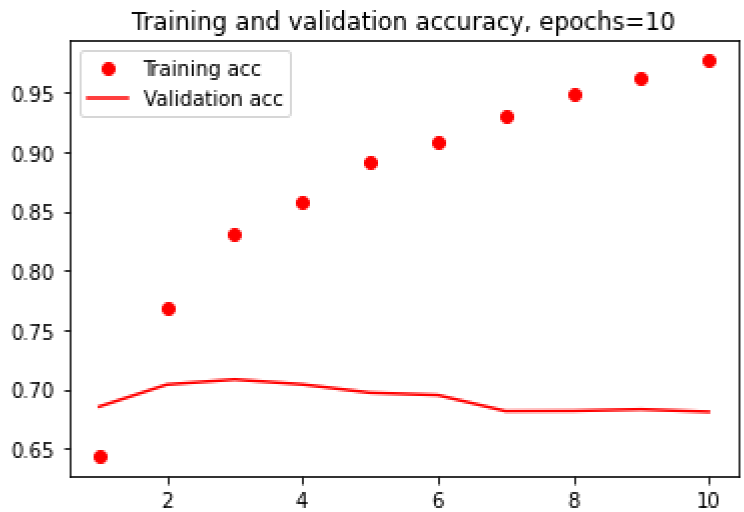
```python
data_pre = "The movie is good. bad  and very bad bad"

pretrained_encoded_documents = [one_hot(data_pre, len(data_pre))]
pretrained_padded_documents = pad_sequences(
                              pretrained_encoded_documents,
                                maxlen=max_length,
                                padding='post')
def predict_output(padded_documents):
    pretrained_result = model_three.predict(pretrained_padded_documents)
    print(pretrained_result)

    if pretrained_result > 0.5:
        return "pos"

    else:
        return "neg"
# np.array(predict_output)
print(predict_output(pretrained_padded_documents))
```

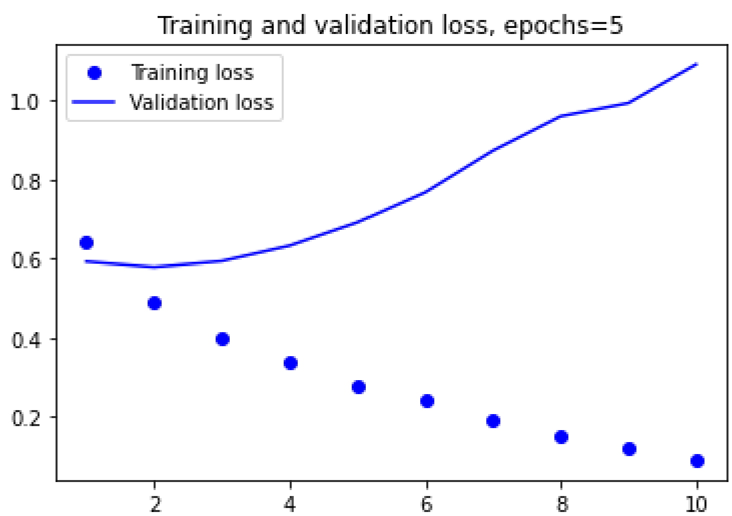
```
[[0.81898814]]
pos
```

In [52]:

```python
import matplotlib.pyplot as plt
acc = history_three.history['acc']
val_acc = history_three.history['val_acc']
loss = history_three.history['loss']
```

```
val_loss = history_three.history['val_loss']
epochs = range(1, len(acc) + 1)
# plot epochs and acc with red circle markers
plt.plot(epochs, acc, 'ro', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.title('Training and validation accuracy, epochs=10')
plt.legend()
plt.figure()
# plot epochs and loss with blue circle markers
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss, epochs=5')
plt.legend()
plt.show()
```





When a neural network is trained, the model gains knowledge using weights. Rather than training another neural model from scratch, the extracted knowledge from the previous model and be extracted and used in the other model. The pre-trained weights can be used as a starting point to train a network. The weights the can further be optimized to provide better performance for the model.If the problem statement we have at hand is very different from the one on which the pretrained model was trained — the prediction we would get would be very inaccurate. It was expected that the pre-train model provide better accuracy compared to the model that do not implement pre-train word embeddings but the accuracy score of the pretrained model is lower compared to the model.