ASSIGNMENT 1C:

Comparing Classification Models.

Neural Networks and Natural Language Processing

Carolina Raymond Rodrigues.
Student ID: 210607000
ec21209@qmul.ac.uk

Contents

List of Figures

List of Tables

1 Introduction

Text classification is a NLP task in which according to its context, a text will be assigned to a label from a predefined set .

For this assignment a labelled dataset of IMDB has been provided. The task is to build a model that classifies the reviews as either positive or negative. Therefore, this task can be considered as a text classification problem.

Different classification models will be used in order to fulfil our purpose. First we will start using One-hot vectors. Secondly this vector will be replaced with word embeddings. Two techniques will be applied to obtain word embeddings. The first one consists on learning the embeddings jointly with the model, while the second one consists on using pre-trained embeddings.

Moreover, the effect of an additional dense layer in the network will be analysed, as well as the application of Convolutional Neural Networks (CNN) for text classification.

Before starting, we must download the dataset and pre-process it. The given dataset corresponds to IMDB Large Movie Review Dataset, which consists on 25000 reviews for training and other 25000 for testing. Futhermore, the number of of positive and negative reviews is the same (50% split).

The given dataset has been preprocessed. Reviews are not represented as a text, instead a vocabulary of the most common 10000 words in the data is build. This is done in order to stop the dataset from becoming too sparse, creating possible overfitting.

Textual data can't be used as input in deep learning models, therefore, it is necessary to transform the textual data into indices (create a word2idx dictionary). Each word of the vocabulary will be given an index, and reviews will be represented with this indices. In the case of having a review with a word that is not part of the vocabulary, we will give that word the index 2, that represents the string "UNK". Morevoer, the start of the review will be represented with the index 1 that means "START" in order to know when the review starts after applying padding ("PAD", index 0).

Padding consist on filling the unused space. This will be done to have reviews with a predefined length (256 in this case) and consequently a better performance will be obtained. In this case padding has been applied at the beginning.

Furthermore, in order to visualise the reviews after being pre-processed, a dictionary known as idx2word has been created. This dictionary is a reversal of the word2idx dictionary.

Now that the data has been prepared, we can start comparing the models.

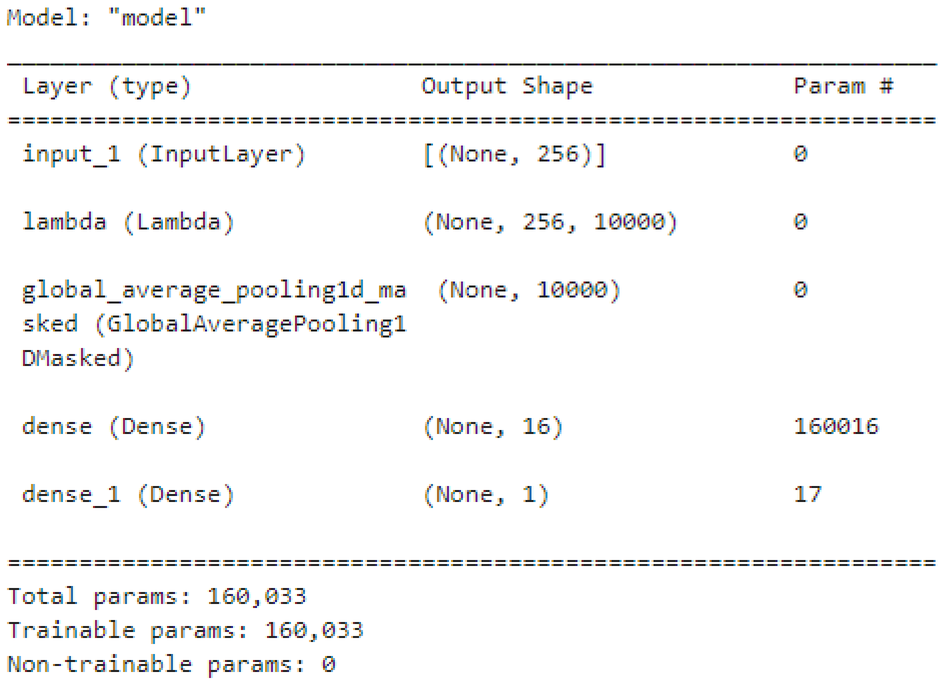
2 Build a neural network classifier using one-hot word vectors and train and evaluate it

In this section a neural network classifier using one-hot word vectors is evaluated.

A one-hot vector representation is a vector of dimension equal to the vocabulary size. This vector will be filled with 0s. Nevertheless, one of the elements' value will be 1. This value will correspond to the index that represents the word that is being represented.

The words in the reviews will be represented with one-hot vectors and be used as inputs in the next layer of the model. This layer will compute average on all word vectors in a sentence without considering padding. This will be done with help of the function GlobalAveragePooling1DMasked(). The resulting output will be introduced in a fully connected layer whose outcome will be again introduced in a single output node with the sigmoid activation function. If the output is 1 the model is predicting a positive layer, otherwise it is predicting a negative review.

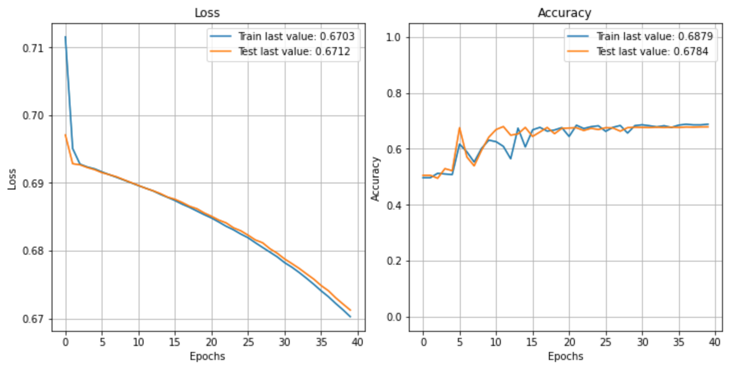
A visual summary of this model can be seen in Figure 1.



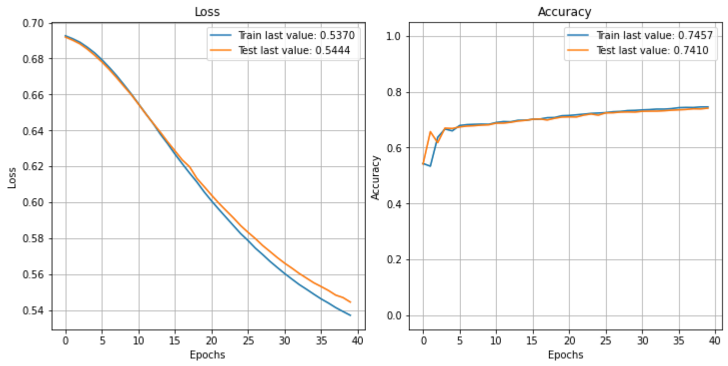
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 256)]             0

 lambda (Lambda)             (None, 256, 10000)        0

 global_average_pooling1d_ma  (None, 10000)            0
 sked (GlobalAveragePooling1
 DMasked)

 dense (Dense)               (None, 16)                160016

 dense_1 (Dense)             (None, 1)                 17

=================================================================
Total params: 160,033
Trainable params: 160,033
Non-trainable params: 0
```

Figure 1: One-hot vector model summary.

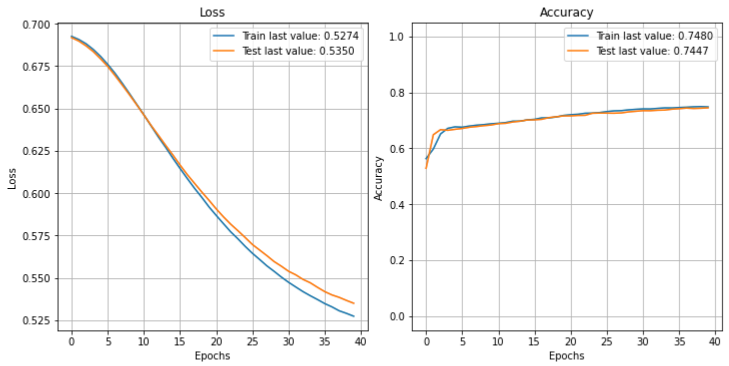
The model is compiled using a binary cross-entropy classifier (binary problem since the output is either positive (1) or negative (0)). Moreover, Adam optimiser is used (Stochastic Gradient Descent is used for training) and accuracy is used as performance metric. Different activation function have been used for the first fully connected layer.The results can be seen in the following figure.



(a) Sigmoid.



(b) Relu.



(c) Tanh.

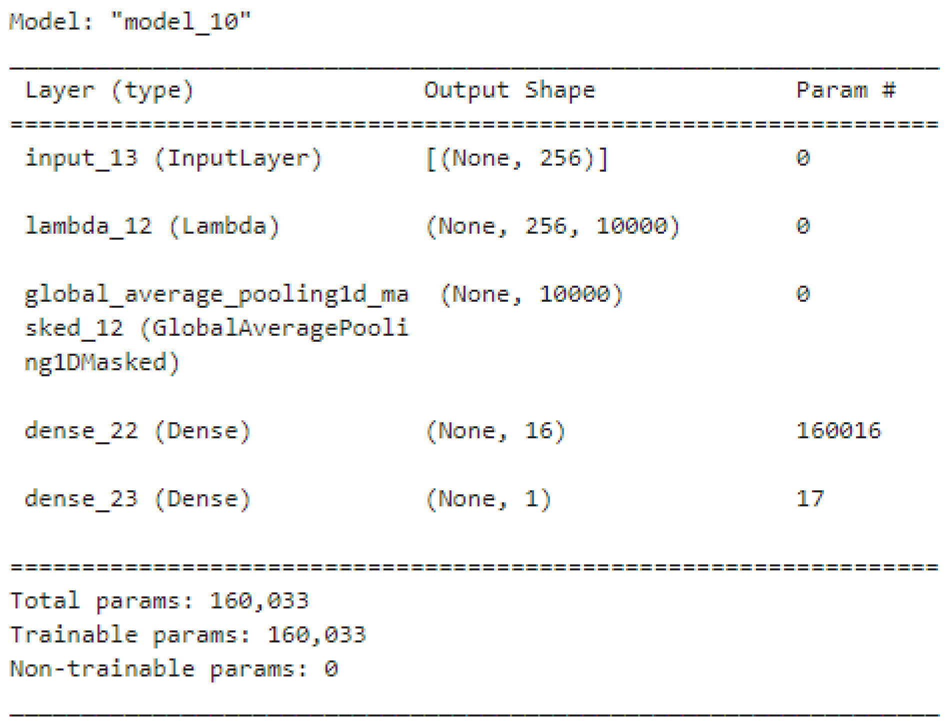
Figure 2: Results of the model training with different activation functions.

Since the best accuracy rate is obtained when the a tanh activation function is used (highest validation accuracy). We will use this function in the fully connected layer and the check the results obtained for the testing data. In this case, the accuracy obtained is 74.24% which is quite satisfactory.

3 Modify your model to use a word embedding layer instead of one-hot vectors, and to learn the values of these word embedding vectors along with the model.

Although one-hot vector are quite simple, the performance of the chosen model can improve if a representation that better understands the meaning of words is used. Therefore, we will use an embedding layer in order to represent the input data in a way in which similar word meaning have a similar representation.

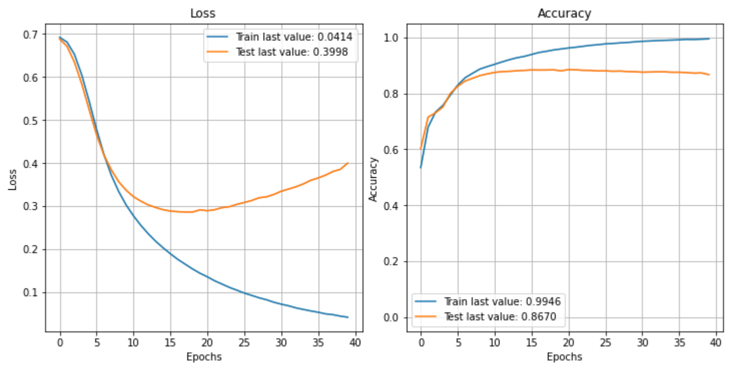
Adding this layer results in the following model (Figure 3):



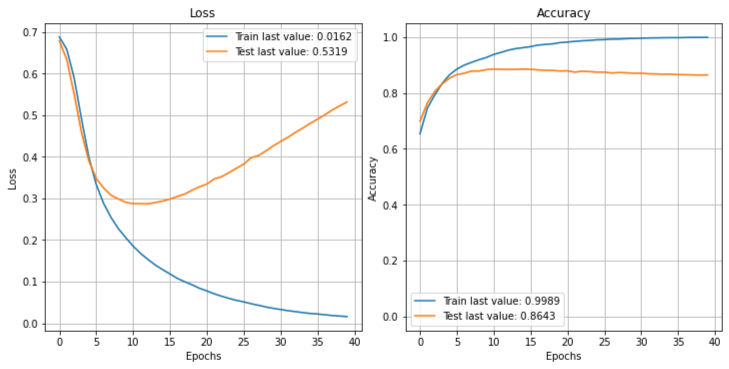
```
Model: "model_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_13 (InputLayer)       [(None, 256)]             0

 lambda_12 (Lambda)          (None, 256, 10000)        0

 global_average_pooling1d_ma  (None, 10000)            0
 sked_12 (GlobalAveragePooli
 ng1DMasked)

 dense_22 (Dense)            (None, 16)                160016

 dense_23 (Dense)            (None, 1)                 17

=================================================================
Total params: 160,033
Trainable params: 160,033
Non-trainable params: 0
_________________________________________________________________
```

Figure 3: Model summary with word embeddings.

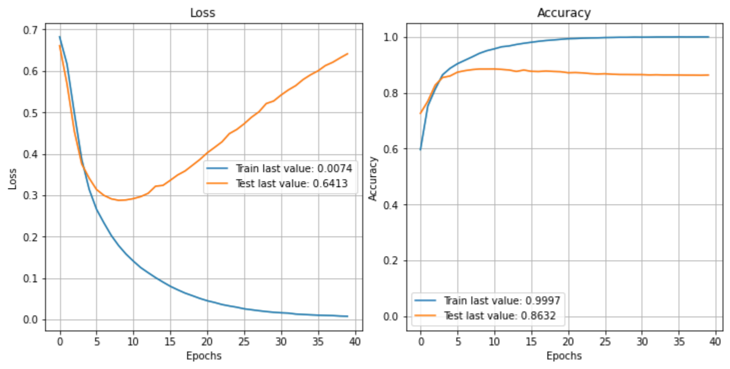
Just as before, the model is compiled using a binary cross-entropy classifier, Adam optimiser is used and accuracy is used as performance metric. The results of the training can be seen in the following figure where different activation functions for the fully connected layer have been used.



(a) Sigmoid.



(b) Relu.



(c) Tanh.

Figure 4: Results of the model training with different activation functions with embedding layer.

In the previous figure, we can see that the one that yields the best result is when the activation function of the hidden layer is the sigmoid function (we obtain a validation accuracy that is higher in comparison to the other two functions and the overfitting when the number of epoch are equal to 40 is lower). Therefore, it will be the chosen function.

Moreover, if we compare the results we obtain when we use one-hot vector and a word embedding layer, we can see that the later one results in a better model, the difference in accuracy of the validation set is quite significant. With word embeddings we obtain a 86.70%, while in the case of the one-hot vector we obtain 74.47%. This results make sense, as it has been said, word embedding captures meaningful information drawn from the context of the words, and therefore, they are able to generalise more effectively.

When using one-hot vector, the model only takes into account the words that the model has learned during the training process. Nevertheless, when embeddings are used, words that share a semantical meaning, will have similar embeddings and therefore, they will lead to a similar classification.

4 Adapt your model to load and use pre-trained word embeddings instead; train and evaluate it and compare the effect of freezing and fine-tuning the embeddings

Until now, we have learned word embeddings jointly with the model, however, learning our own embeddings might not be the best option due to two reasons:

- Sparsity of training data due to the large amount of rare words, what may lead to a poor word embedding representation.
- Large number of trainable parameters since the model has to learn the embeddings too.

A good way of dealing with this problem is to use pre-trained word embeddings. Concretely, for this assignment we will be using GloVe embeddings, that derive the relationship between the words from Global Statistics.

When it comes to using the pre-trained embedding layer, it is necessary to decide if the embeddings should be updated or not during the model's training. In the case of not, we will be talking about freezing pre-trained embeddings. Otherwise, we will fine-tune the embedding weights.

Now that we have an idea about GloVe embeddings, we can adapt our model to use pre-trained embeddings instead. We will start performing text classification using a Bag-of-Words model and then we will switch to LSTM.
However, before focusing on each one of the models, we must load the pre-trained embeddings. Two functions are used for this:

- readGloveFile(gloveFile): loads the GloVe file of the Standford NLP toolkit.
- createPretrainedEmbeddingLayer(wordToGlove, wordToIndex, isTrainable): builds the embedding layer. The argument isTrainable is a boolean. In the case of being False

the embedding layer will be freezed during training. Otherwise, the weights will be fine-tuned.

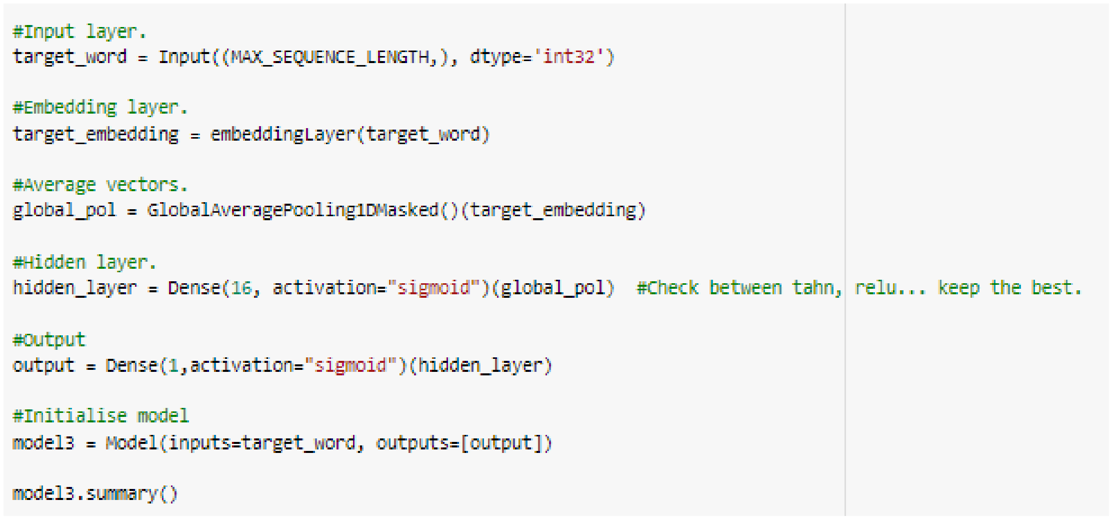
Once we have used these functions, we will have an embedding layer whose embedding size is equal to 300. Now that we have the embedding layer, we will start adapting it to our models.

4.1 Neural Bag-of-Words (BoW) using pre-trained word embeddings.

4.1.1 Freezing embedding layer.

The first thing we have to do is to build the embedding layer setting the isTrainable variable in the createPretrainedEmbeddingLayer function as False.

After having the pre-trained layer, we will follow the same procedure used to build the model in section 2, but instead of learning our own embedding using the Embedding function of the Keras framework, we will instead apply the pre-trained embedding layer as it can be seen in Figure 5

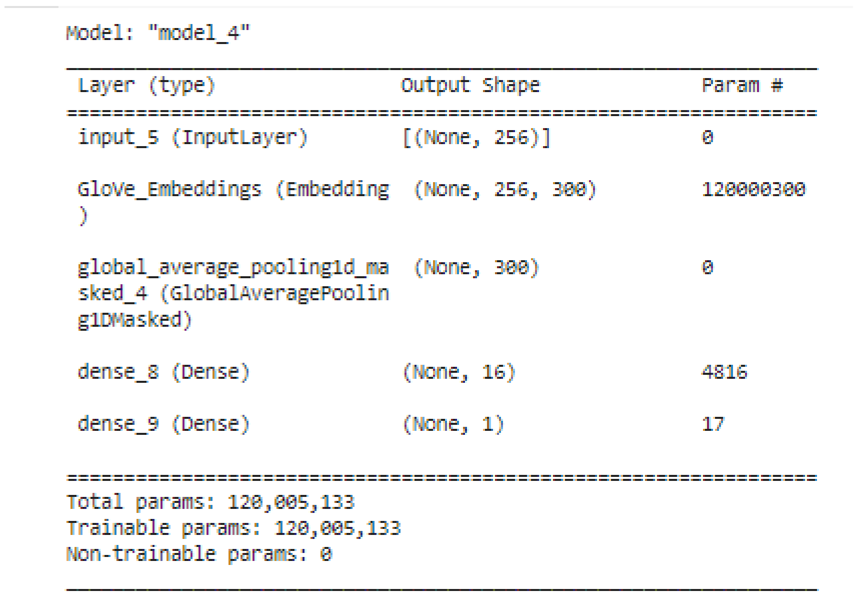


```
#Input layer.
target_word = Input((MAX_SEQUENCE_LENGTH,), dtype='int32')

#Embedding layer.
target_embedding = embeddingLayer(target_word)

#Average vectors.
global_pol = GlobalAveragePooling1DMasked()(target_embedding)

#Hidden layer.
hidden_layer = Dense(16, activation="sigmoid")(global_pol)  #Check between tahn, relu... keep the best.

#Output
output = Dense(1,activation="sigmoid")(hidden_layer)

#Initialise model
model3 = Model(inputs=target_word, outputs=[output])

model3.summary()
```

Figure 5: Model configuration for Neural BoW with frozen pre-trained embeddings.

As it can be seen in the previous figure, the hidden layer has been configured with a sigmoid activation function. The reason why we selected this function is the same as the ones given in the previous sections: after comparing the results obtained in the validation set for sigmoid, ReLU and tanh, the one that yield better results was the sigmoid function.

A summary of the resulting model can be found in Figure 8



```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 256)]             0

 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

 global_average_pooling1d_ma  (None, 300)              0
 sked_4 (GlobalAveragePoolin
 g1DMasked)

 dense_8 (Dense)             (None, 16)                4816

 dense_9 (Dense)             (None, 1)                 17

=================================================================
Total params: 120,005,133
Trainable params: 120,005,133
Non-trainable params: 0
_________________________________________________________________
```

Figure 6: Neural BoW with frozen pre-trained embeddings model summary.

The model is compiled using a binary cross-entropy classifier, Adam optimiser is used and accuracy is used as performance metric. The results of the training can be seen in the following figure

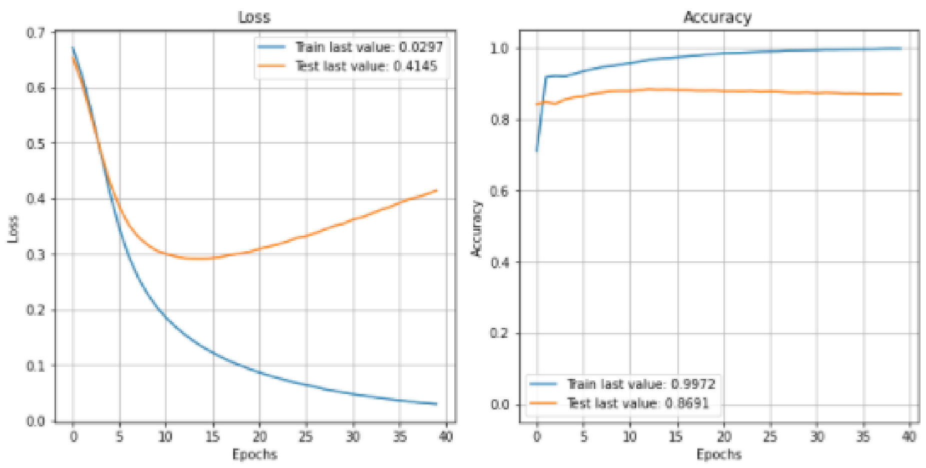
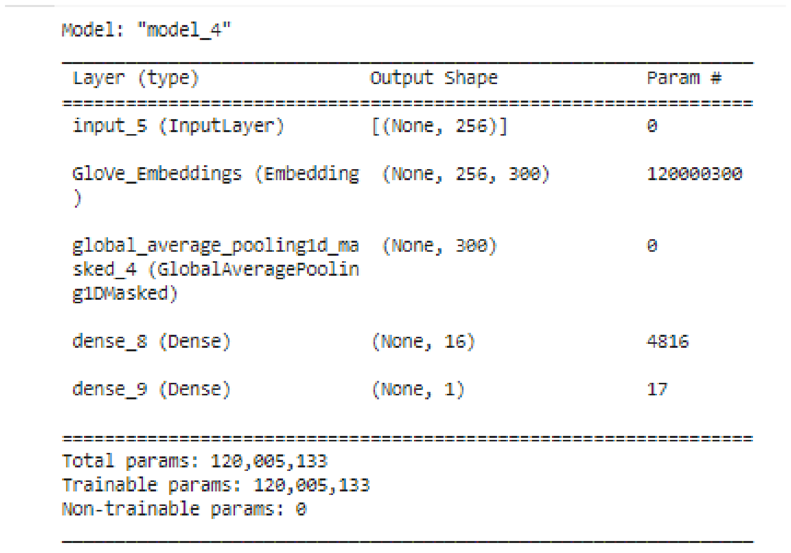


Figure 7: Neural BoW with frozen pre-trained embeddings accuracy and loss plot.

The model with frozen pre-trained embeddings provides better results than the ones implemented in the previous sections, since the accuracy has increased to 86.91%. As it was said in the beginning of this section, we expected better results since we where trying to solve the sparsity and the large amount of training parameters problems.

4.1.2 Fine tuning embedding weights.

For Fine tuning embedding weights, the same configuration as the one in the previous section is required. Nevertheless, we must highlight that we must load again the embedding layer using the createPretrainedEmbeddingLayer() function. However, the isTrainable parameter must be now change to True in order to perform the weight updates during training. The following configuration has been implemented.



```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 256)]             0

 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300
 )

 global_average_pooling1d_ma  (None, 300)              0
 sked_4 (GlobalAveragePoolin
 g1DMasked)

 dense_8 (Dense)             (None, 16)                4816

 dense_9 (Dense)             (None, 1)                 17

=================================================================
Total params: 120,005,133
Trainable params: 120,005,133
Non-trainable params: 0
_________________________________________________________________
```

Figure 8: Model configuration for Neural BoW with frozen pre-trained embeddings.

And the results obtained after training are:

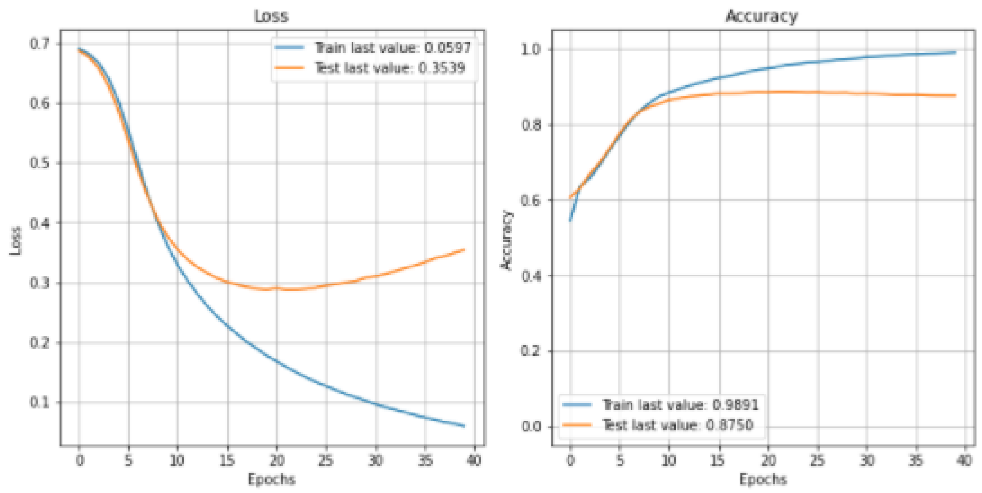


Figure 9: Model configuration for Neural BoW with frozen pre-trained embeddings.

As we can see, fine-tuning the weights of the embedding layer resulted to be beneficial for the given task, since the accuracy of the validation set has slightly increased (87.5% vs 86.91%). The reason behind this, is that we have a large training dataset and therefore, gradient updates is not deploying the connection between similar embeddings. In the case of having a small dataset, it is likely that we will not obtain a better result with the fine-tuning method.

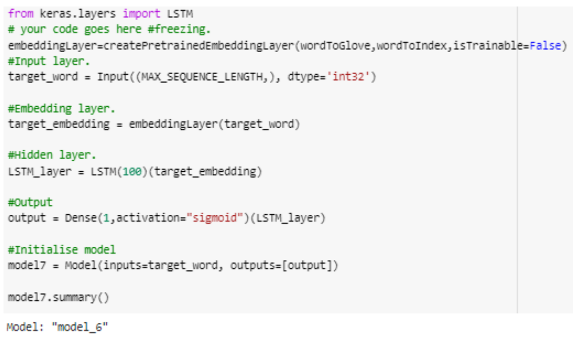
4.2 LSTM with pre-trained word embeddings.

In this subsection, text classification is performed using Long-Short-Term Memory (LSTM) neural networks. The advantage of this model is that it has a long term memory.

The main problem of using the BoW algorithm is that words are trained separately, having no actual meaning as a sentence. Thus, the predictions made by the model are based on pure statistics and no semantical meaning is taken into account.

On the contrary, if appropriate embeddings are used with the LSTM model, the output will provide a better accuracy since the model will be able to find the actual meaning of the sentences in the reviews.

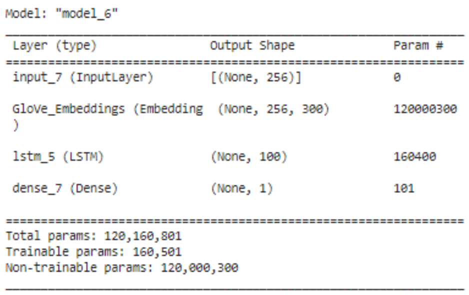
The LSTM model will be defined in the following way: The first layer is the embedding layer, where input words are represented with a dense vectorial representation. The second layer is the LSTM layer with 100 neurons. Finally, a single output dense layer is used whose activation function is the sigmoid. In Figure 10 we can see the code and a summary of the implemented model.



```
from keras.layers import LSTM
# your code goes here #freezing.
embeddingLayer=createPretrainedEmbeddingLayer(wordToGlove,wordToIndex,isTrainable=False)
#Input layer.
target_word = Input((MAX_SEQUENCE_LENGTH,), dtype='int32')

#Embedding layer.
target_embedding = embeddingLayer(target_word)

#Hidden layer.
LSTM_layer = LSTM(100)(target_embedding)

#Output
output = Dense(1,activation="sigmoid")(LSTM_layer)

#Initialise model
model7 = Model(inputs=target_word, outputs=[output])

model7.summary()
```

```
Model: "model_6"
```

(a) LSTM code.



```
Model: "model_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 256)]             0

 Glove_Embeddings (Embedding  (None, 256, 300)         120000300
 )

 lstm_5 (LSTM)               (None, 100)               160400

 dense_7 (Dense)             (None, 1)                 101

=================================================================
Total params: 120,160,801
Trainable params: 160,501
Non-trainable params: 120,000,300
_________________________________________________________________
```

(b) Model summary.

Figure 10: Building the LSTM model

As it has been explained previously, two type of configurations can be analysed depending on the training of the embedding layer. Both of these architectures will be studied in the following sub-subsections.

4.2.1 Freezing embedding layer.

For the freezing embedding weights, we must load again the embedding layer using the createPretrainedEmbeddingLayer() function, where the isTrainable parameter must be set to False, so weight updates are not performed during training.

The validation set accuracy value obtained during training is 58.39%. While in the case of the test accuracy we obtain 59.01%. And therefore, our model does not overfit (training accuracy 62.53%)

4.2.2 Fine tuning embedding weights.

For the fine-tune embedding layer, embedding weights, we must load again the embedding layer using the createPretrainedEmbeddingLayer() function, where the isTrainable parameter must be set to False, so weights updates are performed during the training process.

The results show that fine-tuning the weights of the embedding layer provides worst results in comparison to the frozen embedding layer (the accuracy of the validation set is 57.47%).

Additionally, if we compare the LSTM model with the model implemented in Assignment 1 part B, we can see that the validation set accuracy value the performance has dropped significantly. In lab 2, we had an accuracy value of 89.05%. However, when using the GloVe pre-trained embeddings, the best accuracy value we can obtain is 58.39% with weights fine-tuning. The reason behind this drop is that, as we have explained, with LSTM, the semantical meaning between words is taken into account. However, sometimes each part of a sequence does not matter equally, and using a pre-trained embeddings may lead to bias results depending on the order of the words and the dataset.

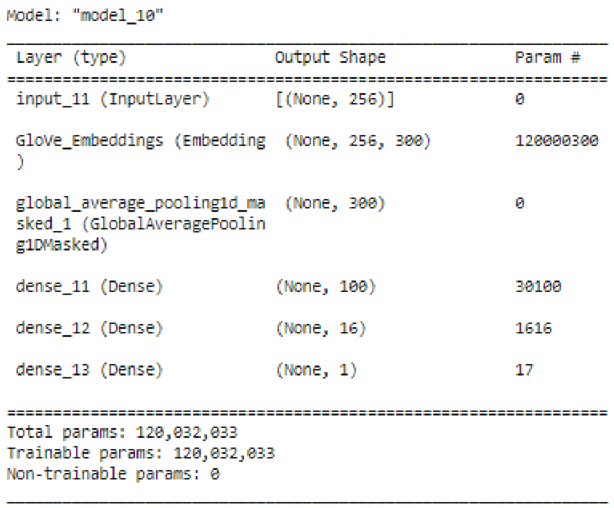
All this suggests that using pre-trained GloVe embeddings for LSTM models is not appropriate. A good solution for this may be using Convolutional Neural networks, as it will be explained in the Section 6.

5 Add another fully-connected layer to your network.

In this section we analyse the effects of introducing additional hidden layer to the Neural averaging network model presented in section 3. We will start adding one layer and then introduce one more. The results can be seen in the following subsections.

5.1 Add one hidden layer.

The extra layer has been included between the average pooling layer and the first fully connected layer. The model summary can be seen in Figure 11



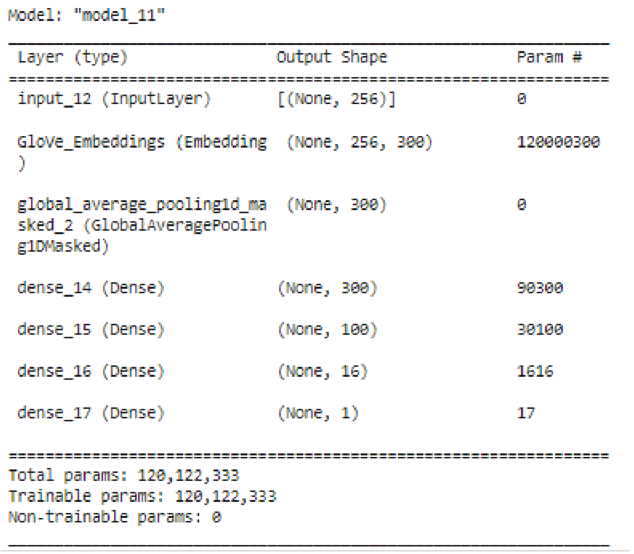
```
Model: "model_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_11 (InputLayer)       [(None, 256)]             0         
                                                                 
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300 
 )                                                               
                                                                 
 global_average_pooling1d_ma  (None, 300)              0         
 sked_1 (GlobalAveragePoolin                                     
 g1DMasked)                                                      
                                                                 
 dense_11 (Dense)            (None, 100)               30100     
                                                                 
 dense_12 (Dense)            (None, 16)                1616      
                                                                 
 dense_13 (Dense)            (None, 1)                 17        
                                                                 
=================================================================
Total params: 120,032,033
Trainable params: 120,032,033
Non-trainable params: 0
_________________________________________________________________
```

Figure 11: Model summary with one extra hidden layer.

Adding one hidden layer is beneficial for the model since the accuracy has increased to 87.17% in the validation set and 85.72% in the test set. This is because adding a layer implies learning features from the input data that can be helpful for making predictions.

5.2 Add two hidden layers.

The resulting model after adding two extra dense layers can be seen in Figure 12



```
Model: "model_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_12 (InputLayer)       [(None, 256)]             0         
                                                                 
 GloVe_Embeddings (Embedding  (None, 256, 300)         120000300 
 )                                                               
                                                                 
 global_average_pooling1d_ma  (None, 300)              0         
 sked_2 (GlobalAveragePoolin                                     
 g1DMasked)                                                      
                                                                 
 dense_14 (Dense)            (None, 300)               90300     
                                                                 
 dense_15 (Dense)            (None, 100)               30100     
                                                                 
 dense_16 (Dense)            (None, 16)                1616      
                                                                 
 dense_17 (Dense)            (None, 1)                 17        
                                                                 
=================================================================
Total params: 120,122,333
Trainable params: 120,122,333
Non-trainable params: 0
_________________________________________________________________
```

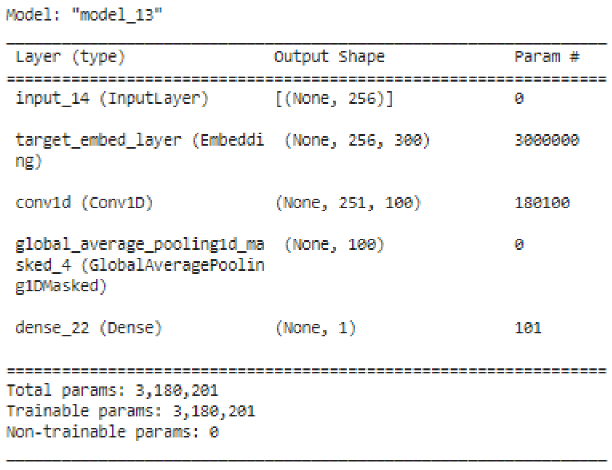
Figure 12: Model summary with two extra hidden layers.

Adding two hidden layers to the model does not provide satisfactory results. The model's validation accuracy has slightly decreased to 87%. The reason behind this is that adding layers makes the model more complex and therefore, after a certain threshold, it will not able of generalising well enough causing a overfitting problem. However, we are still getting a better result to the one obtained when no extra hidden layers were added (86.70%).

6 Build a CNN classifier, and train and evaluate it. Then try adding extra convolutional layers, and conduct training and evaluation.

LSTM trains and predicts the meaning of the entire sequence rather than just the meaning of individual words. However, the order of words will lead to bias. A good solution to this problem is using CNNs, since it will identify patterns on the sentences regardless of words position, and therefore, it might provide better results.

In order to implement the CNN model, we will use the Conv1 function from the Keras framework. This is a one dimensional convolutional layer whose kernel size is equal to 6 and the dimensionality of the output space will be 100. Additionally, the next layer is the pooling layer where max pooling is performed using the previously implemented GlobalAveragePooling1DMasked() function, where the maximum value of the resulting convolutional channels will be saved. Finally, the last layer is the output layer with one single neuron and sigmoid activation function. Figure 13 summarises the model explained in this paragraph.


```
Model: "model_13"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_14 (InputLayer)       [(None, 256)]             0

 target_embed_layer (Embeddi  (None, 256, 300)         3000000
 ng)

 conv1d (Conv1D)             (None, 251, 100)          180100

 global_average_pooling1d_ma  (None, 100)              0
 sked_4 (GlobalAveragePoolin
 g1DMasked)

 dense_22 (Dense)            (None, 1)                 101

=================================================================
Total params: 3,180,201
Trainable params: 3,180,201
Non-trainable params: 0
_________________________________________________________________
```
Figure 13: CNN model summary.

After training the model, we can see that it converges (Figure 14) and reaches an accuracy on the validation set is 85.35% and 84.17% for the test set.

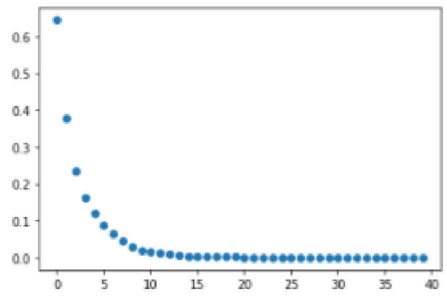
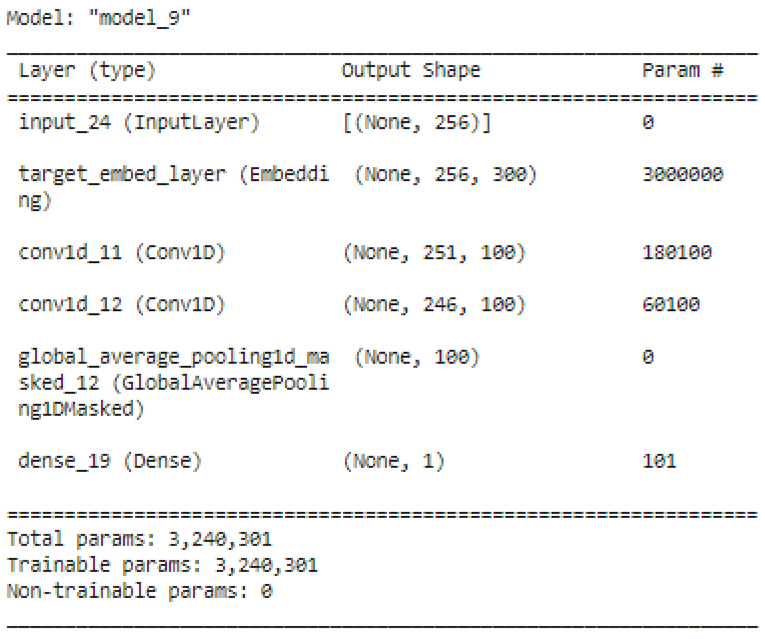


Figure 14: CNN training loss.

These results are quite disappointing, since the accuracy values obtained with the LSTM with scratch learned embeddings was higher (89.05%). The main reason behind this is that even thought CNN identify important patterns it is hard to obtain contextual information in most of the cases. It might be interesting to combine both of them to perform text classification. CNN can be used to extract higher-level features while LSTM to capture long-term dependencies between word sequences.

However lets see, if one more convolutional layer improves the performance or not. The summary of this model can be seen in the following figure.



```
Model: "model_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_24 (InputLayer)       [(None, 256)]             0

 target_embed_layer (Embeddi  (None, 256, 300)         3000000
 ng)

 conv1d_11 (Conv1D)          (None, 251, 100)          180100

 conv1d_12 (Conv1D)          (None, 246, 100)          60100

 global_average_pooling1d_ma  (None, 100)              0
 sked_12 (GlobalAveragePooli
 ng1DMasked)

 dense_19 (Dense)            (None, 1)                 101

=================================================================
Total params: 3,240,301
Trainable params: 3,240,301
Non-trainable params: 0
_________________________________________________________________
```

Figure 15: CNN summary with two convolutional layers.

The results of the training process show no improvements. Although, the model converges a bit faster and the training loss has been reduced since it went from 2.7056e-04 to 1.7474e-05 (Figure 16), the validation accuracy has decreased (85.35% vs 84.75%)

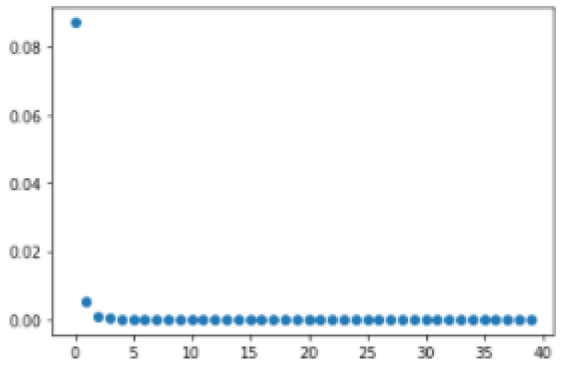


Figure 16: CNN loss with two convolutional layers.

As we said adding more layers help to extract more patterns and makes the model more complex. However, adding to many layer may lead to overfitting the data and consequently a higher number of false positives will be obtained.

7 Conclusions

In the following table a summary of the obtained results in each section is provided:

Model	Validation accuracy (%)	Test accuracy (%)
One-hot encoding	74.47	74.24
Scratch word embeddings	86.7	86.02
BoW with GloVe frozen embeddings	86.91	84.89
BoW with GloVe fine-tuned embeddings	87.5	86.08
LSTM with GloVe frozen embeddings	58.39	59.01
LSTM with GloVe fine-tuned embeddings	57.47	57.02
Neural averaging with one extra layer	87.17	85.72
Neural averaging with two extra layer	87.00	85.72
CNN	85.35	84.17
CNN with one extra layer	84.75	82.92

Table 1: Summary of accuracy values for each model.

As it can be seen, in this case the best option to use is the BoW with pre-trained with GloVe fine-tuned embeddings. Nevertheless, if we go back to lab 2, the performance of the LSTM with scratch word embeddings was even better (89.05%). Therefore, for the given task the best option is using this model, since the model finds the actual semantic meaning of the sentences in the reviews, improving therefore the performance.