Signal and Natural Language Processing

Lab 3

Δημήτρης Καρποντίνης - 03400135

Θοδωρής Λυμπερόπουλος - 03400140

Contents

A. Preparation Lab **2**
1. Data Preprocessing 2
1.1. Label Encoding 2
1.2. Tokenization 2
1.3. Word Encoding 3
2. Model 4
2.1. Embedding Layer 5
2.2. Output Layer 6
2.3. Forward pass 6
3. Training 7
3.1. Loading the Dataset 7
3.2. Optimization 8
3.3. Training 8
3.4. Evaluation of the results 9

A. Preparation Lab **12**
Question 1 12
Question 2 12
Question 3 14
Question 4 18
Question 5 24

Preparation Lab

1. Data Preprocessing

1.1. Label Encoding

Using the `LabelEncoder` tool which is provided by the scikit-learn library, we transform our data using the following code:

```
le = LabelEncoder()
y_train = list(le.fit_transform(y_train))
y_test = list(le.transform(y_test))
n_classes = le.classes_.size
```

After initializing a `LabelEncoder` object, we apply `fit_transform` to the input data. Then, we simply apply `transform` to the test data, since the model has already been fitted. The last line of code calculates the number of different classes for our dataset. This corresponds to the number of unique labels. After printing the first 10 labels for the MR dataset, we get the following results:

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

1.2. Tokenization

In the dataloading script we create a `SentenceDataset`, which implements a Dataset of Sentences. It receives sentences and their corresponding sentiment labels, and produces a `Dataset` object. During the initialization method we expand word abbrevations (i.e. "won't" is replaced by "will not") in order to increase the model's performance.

Furthermore we tokenize the words and other useful symbols from each sentence. Specifically, we process the symbols *'.' ',' '?' ';'* along with the sentence words lowered. All other symbols are neglected.

In order to tokenize the sentences, we use the python library `nltk` and it's method `tokenize.word_tokenize`. We print the first 10 examples of

the training data from the corresponding SentenceDataset object.The described process is accomplised with the code seen below:

```
fix_step = [contractions(phrase) for phrase in X]
self.data =
[word_tokenize(re.sub(r"[^a-z|\.|\,|\;|\?]", " ", x.lower()))
for x in fix_step]
self.labels = y
self.word2idx = word2idx
```

We can see the results in figure 1.

```
[['the', 'rock', 'is', 'destined', 'to', 'be', 'the', 'st', 'century', 's', 'new', 'conan', 'and', 'that', 'he',
'is', 'going', 'to', 'make', 'a', 'splash', 'even', 'greater', 'than', 'arnold', 'schwarzenegger', ',', 'jean',
'claud', 'van', 'damme', 'or', 'steven', 'segal', '.'], ['the', 'gorgeously', 'elaborate', 'continuation', 'of',
'the', 'lord', 'of', 'the', 'rings', 'trilogy', 'is', 'so', 'huge', 'that', 'a', 'column', 'of', 'words',
'can', 'not', 'adequately', 'describe', 'co', 'writer', 'director', 'peter', 'jackson', 's', 'expanded',
'vision', 'of', 'j', '.', 'r', '.', 'r', '.', 'tolkien', 's', 'middle', 'earth', '.'], ['effective', 'but',
'too', 'tepid', 'biopic'], ['if', 'you', 'sometimes', 'like', 'to', 'go', 'to', 'the', 'movies', 'to', 'have',
'fun', ',', 'wasabi', 'is', 'a', 'good', 'place', 'to', 'start', '.'], ['emerges', 'as', 'something', 'rare',
',', 'an', 'issue', 'movie', 'that', 's', 'so', 'honest', 'and', 'keenly', 'observed', 'that', 'it', 'does',
'not', 'feel', 'like', 'one', '.'], ['the', 'film', 'provides', 'some', 'great', 'insight', 'into', 'the',
'neurotic', 'mindset', 'of', 'all', 'comics', 'even', 'those', 'who', 'have', 'reached', 'the', 'absolute',
'top', 'of', 'the', 'game', '.'], ['offers', 'that', 'rare', 'combination', 'of', 'entertainment', 'and',
'education', '.'], ['perhaps', 'no', 'picture', 'ever', 'made', 'has', 'more', 'literally', 'showed', 'that',
'the', 'road', 'to', 'hell', 'is', 'paved', 'with', 'good', 'intentions', '.'], ['steers', 'turns', 'in', 'a',
'snappy', 'screenplay', 'that', 'curls', 'at', 'the', 'edges', ';', 'it', 'is', 'so', 'clever', 'you', 'want',
'to', 'hate', 'it', '.', 'but', 'he', 'somehow', 'pulls', 'it', 'off', '.'], ['take', 'care', 'of', 'my', 'cat',
'offers', 'a', 'refreshingly', 'different', 'slice', 'of', 'asian', 'cinema', '.']]
```

Figure 1: The first 10 examples

1.3. Word Encoding

The process of converting a sentence to a list of ids of the corresponding words of that sentence requires from us to determine the length of the resulting list. That is because different sentences have different sizes (number of words they contain) but the model requires a fixed input length. Thus, we calculate an optimal length, after a simple statistical analysis on the training set's sentences. The plot of the figure 2 shows the frequencies of the different lengths of the dataset. As we can see, the distribution of the lengths follows a Negative Binomial distribution. For simplicity, we considered it to be a Normal distribution and calculated the optimal value by adding two times the standard deviation to the mean value (a technique

for excluding outliers).

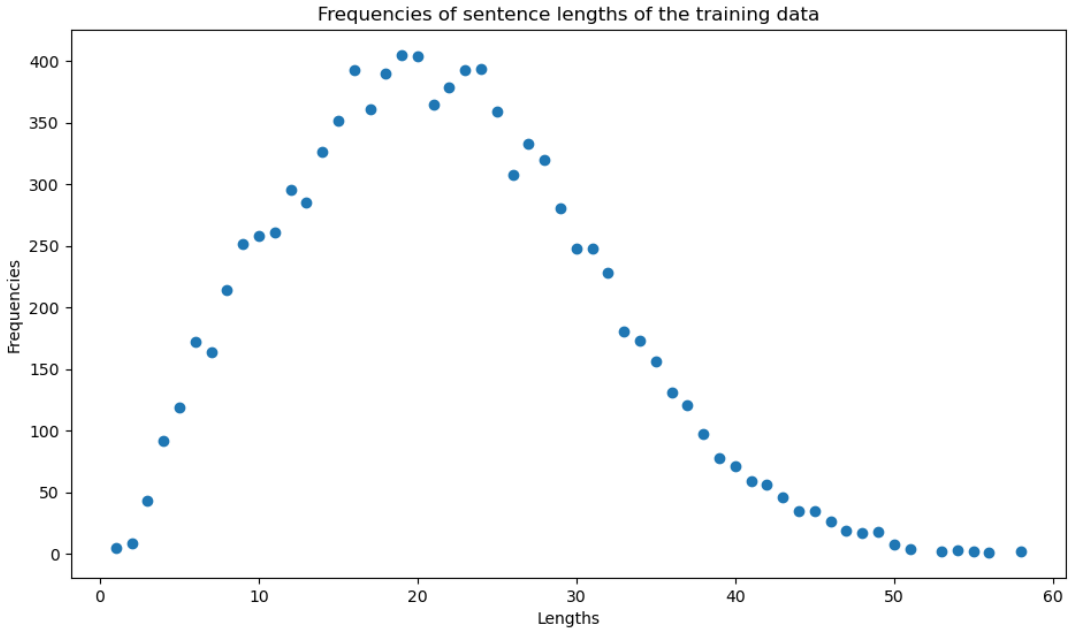

Figure 2: Frequencies of the sentences' lengths.

After finding the optimal max_length we either increased or decreased the number of indices in each sentence in order to comply with the max_length. In case of less words, we simply padded the sequence with zeros. This method allowed for the equality of sentence length, something that will prove crucial later on.

In Figure 3 we print two examples of the training set, after applying this transform. Specifically each example is divided in two parts. Firstly, we can see the example and its label both given in natural language. Secondly, we present the same sentence given by the word indices and padded to the max length, along with the encoded label and the initial sentence length.

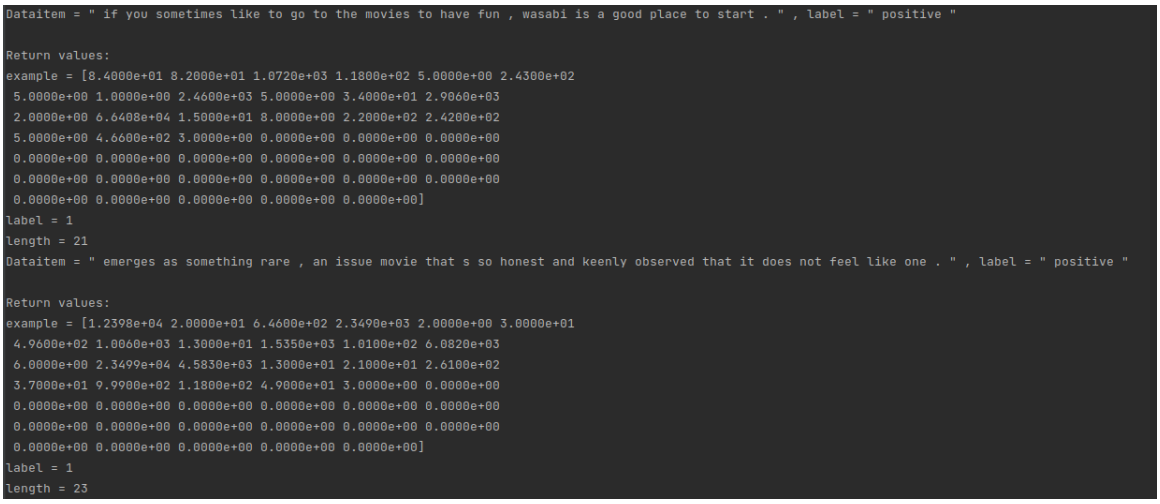

Figure 3: Frequencies of the sentences' lengths.

2. Model

2.1. Embedding Layer

We use an embedding layer to represent each word as a vector of fixed length 50. We make use of the GloVe dataset, which is available online. From the Word2Vec algorithm, it follows that the weights of this layer correspond to the embeddings of the words of GloVe. We thus find the corresponding embeddings for the words of our dataset (if they exist). The code below initializes such a layer and passes the precalculated embeddings as weight to that layer.

```
self.emb_layer = nn.Embedding(embeddings.shape[0],
embeddings.shape[1])
self.emb_layer.weight = nn.Parameter(torch.from_numpy(embeddings))
```

We then need to set this layer as frozen, since we don't want the error to backpropagate to it and update it. The embeddings are standard and should not change. They contain in themselves semantic information and do not account for the error.

```
self.emb_layer.weight.requires_grad_(trainable_emb)
```

At this point, we should note that there exists a much easier way to define

the embedding layer. It is the line of the code below:

```
self.emb_layer = nn.Embedding.from_pretrained(
torch.FloatTensor(embeddings), freeze=trainable_emb)
```

We will use it in the following models that we will develop.

2.2. Output Layer

We insert a hidden layer after embedding layer, with 25 neurons, and then we add a final layer with 2 neurons, in order to classify the examples into two categories. We apply a non-linear transformation to the output of the hidden layer in rder to increase the capacity of the model. Without it, the model would just perform a somewhat more complex, but still linear transformation of the data. Its power would be limited. Instead, by adding a non-linear transformation, the model is capable of simulating more complex functions. The code below produces those layers:

```
HIDDEN_LAYER_SHAPE = 25
self.hidden_layer = nn.Linear(embeddings.shape[1], HIDDEN_LAYER_SHAPE)
self.activation_func = nn.ReLU()
self.output = nn.Linear(HIDDEN_LAYER_SHAPE, output_size)
```

2.3. Forward pass

For the forward pass of a batch, the algorithm first passes every sentence through the embedding layer to acquire the embeddings of its words. This results in a $(128, 41, 50)$, where 128 is the batch size, 41 is the length of each sentence and 50 is the size of the embedding vector for each word.

Then, we construct a representation for each sentence, by finding the mean value of the non-zero dimensions of the words for each sentence. The resulting tensor will have a size of $(128, 50)$. Finally, we pass the tensor through the layers of our DNN model, to get the predictions for each sentence of the batch.

The code is simple due to the ability of PyTorch to work with tensors and applying each transformation of each of its elements. The code is shown

below:

```
embeddings = self.emb_layer(x)
representations = torch.sum(embeddings, dim=1)/lengths.unsqueeze(1)
representations = self.activation_func(self.hidden_layer
(representations))
logits = self.output(representations)
```

The mean value representation corresponds to an new vector/point in the embedding space, that takes into consideration each word of a sentence. If the words are strongly correlated (have the same meaning), the mean vector will be in the same area, sharing semantic information with those words. Otherwise, it will end up being somewhere between the areas that correspond to the meaning of the words, maybe in an area that is linked with a broader concept that generalizes these notions. In any case, this technique treats all words equally and provides an easy representation for a sentence. For a small number of words, the mean vector will be indicative of the meaning of the sentence, but as this number grows, information will be surely lost. Also, it would be possible for two uncorrelated sentences to have similar mean representations, since this transformation is not injective, and the compression of the information is very large.

3. Training

3.1. Loading the Dataset

We first load the data by calling the `SentenceDataset` class, and then create two Dataloader objects that transform the data to a suitable form, and split it randomly.

```
train_set = SentenceDataset(X_train, y_train, word2idx)
test_set = SentenceDataset(X_test, y_test, word2idx)
train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuf-
fle=True)
test_loader = DataLoader(test_set, batch_size=BATCH_SIZE, shuf-
fle=True)
```

The size of mini-batches is very important for the process of training. For a very small mini-batch with only one datum (also called SGD), the backpropagation process is computationally efficient, although it requires more time

for an epoch. It converges to a local optimum after maneuvering the parameter space. As the size of the mini-batch grows larger, the computation of the gradients is more expensive and the memory usage more intensive. But, overall, the training for an epoch is faster and more efficient. In case we use the whole dataset (also called BGD), the algorithm converges faster to a local minimum, without diverging much from its direction. This is because the random noise or outliers in the data do not contribute to the error. Some recent works (e.g. here) have shown that the local minimum of the BGD is sharper, which leads to a model's quality degradation and poor generalization. On the other hand, SGD converges to flat minimizers, with the same accuracy on the training set, but greater performance on the test set. Overall, there might be a trade-off between the two, the MBGD, that takes advantage of both the qualities of the two methods. In our case, the size of a mini-batch is 32.

We should also mention that the DataLoader shuffles the data every time before partitioning them to mini-batches. This is necessary for the model not to memorize the order of the data and answer respectively, without learning anything about the distribution that produces the data.

3.2. Optimization

The loss criterion used is the *BCEWithLogitsLoss*, since we trained our model on the MR dataset, that contains only two classes of text sentiment. Then we define the parameters of the model, by excluding those that should not be changed. They are those that their variable `requires_grad` is `True`, since the according variable of the embedding layer is equal to `False`. After that, we define an `Adam` optimizer. The code is shown below.

```
criterion = nn.BCEWithLogitsLoss()
parameters = (x for x in model.parameters() if x.requires_grad)
optimizer = torch.optim.Adam(parameters, lr=0.003,
weight_decay=0.00075)
```

3.3. Training

The training of the model is as follows: We iterate over the number of epochs, and call the `train_dataset` function for one step. The method internally uses the functionality of a DataLoader object to split the data

randomly in mini-batches. It then trains the model's parameters, by using a loss criterion and an optimizer passed as arguments. Afterwards, we call the `eval_dataset` method for both the training and evaluation dataset, to get the results of that timestep. We save the results of the training process in some python lists, that will be used in the following step (3.4) to evaluate the evolution of the model's accuracy. The code is shown below:

```
for epoch in range(1, EPOCHS + 1):

    train_dataset(epoch, train_loader, model, criterion, optimizer)

    train_loss, train_acc, (y_train_gold, y_train_pred) = eval_dataset
    (train_loader, model, criterion)

    test_loss, test_acc, (y_test_gold, y_test_pred) = eval_dataset
    (test_loader, model, criterion)
```

3.4. Evaluation of the results

The results of the training and evaluation process can be seen below. As we can see in Figure 4, the accuracy of the model reaches 71% after 50 epochs. The accuracy on the evaluation dataset follows a similar trend, although it fluctuates more. The same holds for the evolution of loss, in Figure 5. It diminishes over time to 0.58 (this is a number, not a percentage, since we calculate the BCEWithLogitsLoss).
In Figure 6 we can see the model's performance on the training and evaluation set after being trained for 50 epochs. The results agree for the two datasets. We observe that the class of negative texts has a higher precision and a lower recall than those of the positive class. Overall the F1-score is very similar for the two classes.

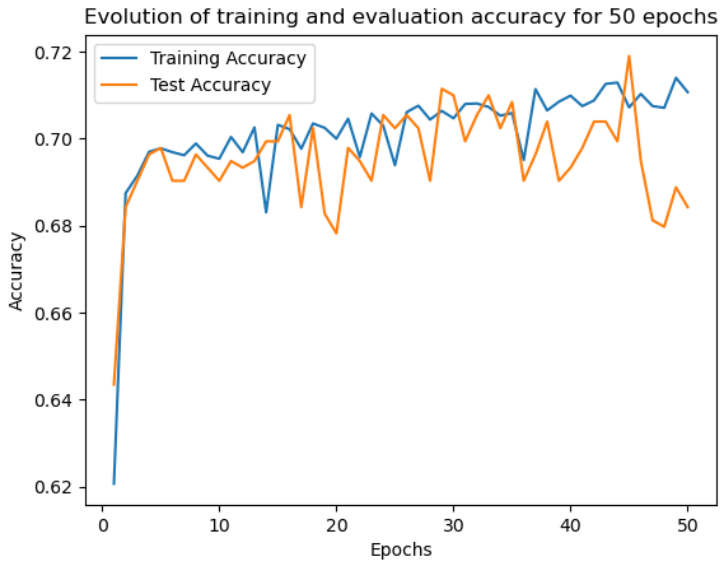

Figure 4: Evolution of model's accuracy for the training and evaluation set for 50 epochs.

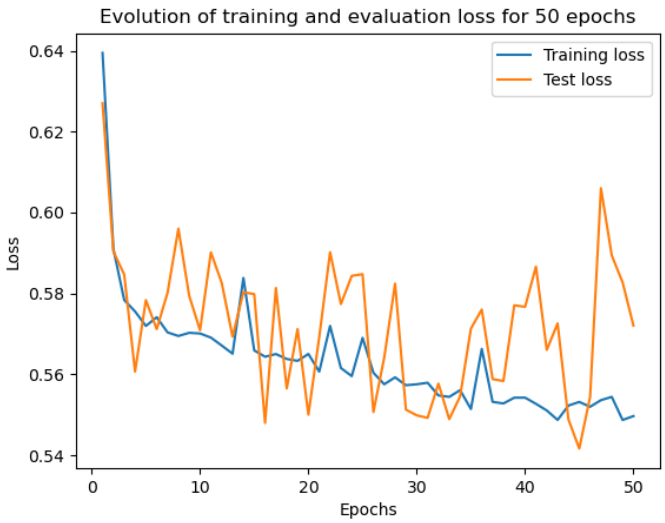

Figure 5: Evolution of loss for the training and evaluation set for 50 epochs.

```
The accuracy of the model on the training set is:  tensor(0.7107)
              precision    recall  f1-score   support

    negative       0.77      0.69      0.73      5609
    positive       0.65      0.74      0.69      4391

    accuracy                           0.71     10000
   macro avg       0.71      0.71      0.71     10000
weighted avg       0.72      0.71      0.71     10000

The accuracy of the model on the test set is:  tensor(0.6843)
              precision    recall  f1-score   support

    negative       0.75      0.66      0.70       374
    positive       0.62      0.71      0.66       288

    accuracy                           0.68       662
   macro avg       0.68      0.69      0.68       662
weighted avg       0.69      0.68      0.69       662
```

Figure 6: Performance metrics for the training and evaluation set.

Main Lab

Question 1

1. 1

In order to represent the embedding of a sentence as the concatenation of mean and max value, we write the following lines of code.

```
representations_sum = torch.sum(embeddings, dim=1)/lengths
.unsqueeze(1)
representations_max = torch.max(embeddings, dim=1)[0]
```

The first line, same as before, calculates the mean value of the word embeddings of a sentence, while the second one finds the word embedding with the max norm among the words of the sentence.

1. 2

The new representation contains more information than before. By applying the max norm, we estimate the distribution of the word embeddins of the sentence. We eliminate the space of the words but we also discover how much it expands. This might add valuable information about which words the distribution might contain.

Question 2

In the following questions, we will train different models to compare their performance. We should note that during each model's training we split the data to the test and train set with a random seed. We acknowledge the fact that this was a mistake. Splitting the dataset should be the same for the validation process, in order for the models to be comparable. Unfortunately, we didn't have the time to fix this error and retrain our models from the beginning.

In this step, we train a Vanilla LSTM model, that captures the long-term dependencies. We expect the model to yield better results than the baseline model.

2. 1

We first define the LSTM architecture in the init method. Our class is named `LSTMDNN` and contains the following lines of code.

```
self.emb_layer = nn.Embedding.from_pretrained(torch.
FloatTensor(embeddings), freeze=True)
self.lstm = nn.LSTM(50, 50, 1, batch_first=True)
self.output = nn.Linear(150, output_size)
```

The forward method receives the word embeddings and applies the LSTM transformation. We only keep the last timestep and exclude all other padded timesteps with the use of a for loop that excludes the zero padding in the end of each sentence.

2. 2

The text is represented as a concatenated tensor of the outputs of the LSTM cells and the mean and max value of the embeddings, calculated as:
`torch.cat((representations_lstm, representations_sum, representations_max), dim=1)`
We then apply the linear layer to our representation.

As expected, the results of the LSTM model improve significantly the accuracy, by five units at least. The architecture of the model forces it to capture the structure of the language, and thus, create a more powerful representation. Although, in Figures 7, 8, 9 we notice that the LSTM's performance on the training set is much better than that of the validation set after some epochs. We might want to decrease the number of epochs the model is being trained to avoid overfitting.

Figure 7: Evolution of model's accuracy for the training and evaluation set for 50 epochs.

Question 3

3. 1

After having acquired the pretrained word embeddings we apply a linear transformation to them, and consequently to a tanh function, in order to produce the desired values. These values in turn will be normalized producing the attention weights used to compute the final sentence representation.

The results of the above model, computed on the test set and presented as a classification report, are given below:
As we can see from Figure 10, while there are some fluctuations in the test set loss it does seem to follow the decreasing curve of the training set loss.

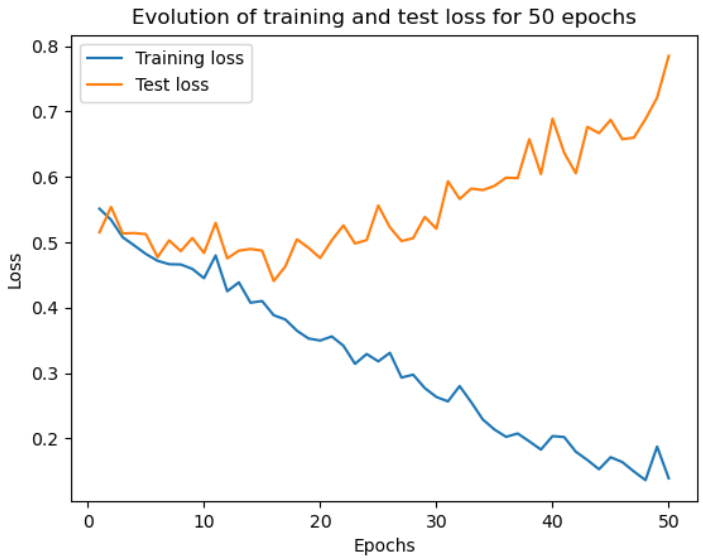

Figure 8: Evolution of LSTM's accuracy for the training and evaluation set for 50 epochs.

The accuracy of the model on the test set is: 0.7508

	precision	recall	f1-score	support
negative	0.71	0.77	0.74	306
positive	0.79	0.73	0.76	356
accuracy			0.75	662
macro avg	0.75	0.75	0.75	662
weighted avg	0.75	0.75	0.75	662

Figure 9: Classification report of LSTM.

In Figure 11 we show the corresponding accuracy graphs for training and test set.

As for the loss graph, the two curves are similar, with the final accuracy for the test set to be around 0.75, as we shall see below in the classification report.

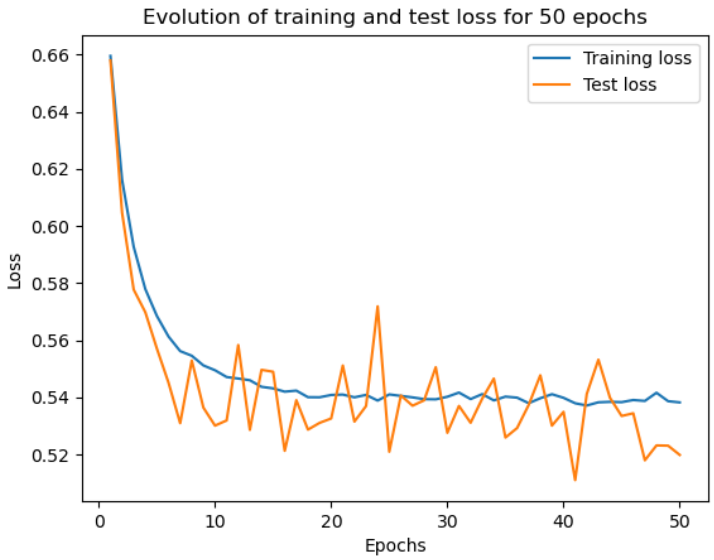

Figure 10: Attention-DNN loss graph

3. 2

We now present the results for the LSTM with attention network architecture.

- Loss Graph:

  The results clearly show that overfitting has occurred. The results could be better if we trained the model for 10 epochs instead of 50.

- Accuracy Graph:

  In accordance with the loss graph we can also deduct here that the number of epochs should be reduced to 10 in order to avoid overfitting.

Figure 11: Attention DNN accuracy graph

- Classification Report:
  The report shows that the model is better fitted towards the positive class.

```
The accuracy of the model on the test set is: 0.7508
              precision    recall  f1-score   support

    negative       0.78      0.74      0.76       348
    positive       0.73      0.76      0.74       314

    accuracy                           0.75       662
   macro avg       0.75      0.75      0.75       662
weighted avg       0.75      0.75      0.75       662
```

Figure 12: Attention DNN results

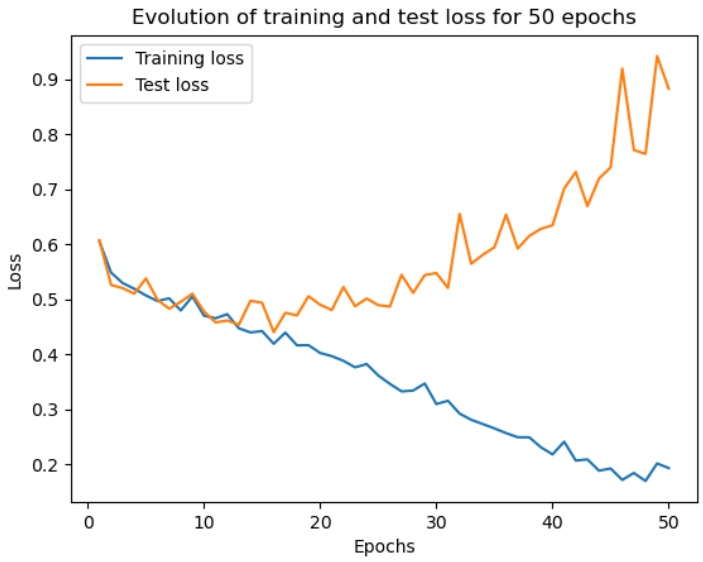

Figure 13: Lstm Attention loss graph

Question 4

4. 1

In this section of the exercise we train a bidirectional lstm model.

Figure 14: Lstm Attention accuracy graph

The accuracy of the model on the test set is: 0.7266

	precision	recall	f1-score	support
negative	0.67	0.76	0.71	292
positive	0.79	0.70	0.74	370
accuracy			0.73	662
macro avg	0.73	0.73	0.73	662
weighted avg	0.73	0.73	0.73	662

Figure 15: Lstm Attention classification report

- Loss Graph:

  Similarly to the vanilla lstm loss graph we see overfitting occurring, with the train loss ending up again around 0.2. The main difference with the vanilla lstm is that the test loss function seems to increase slower in the bidirectional model, giving us a smaller final loss.

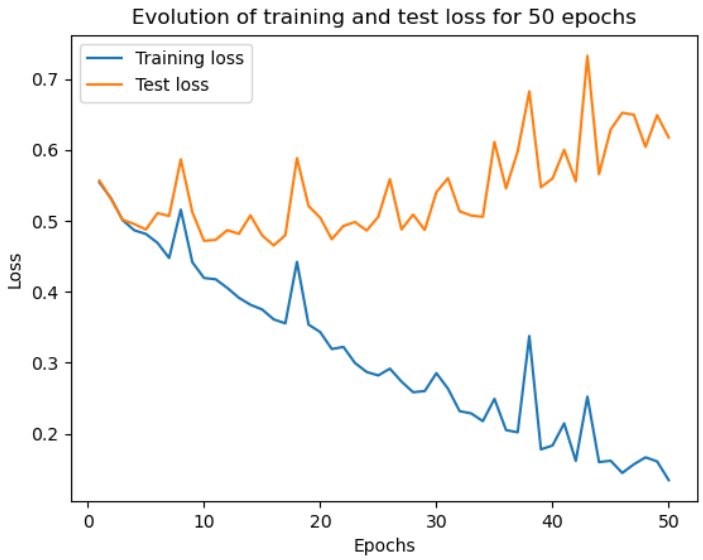

Figure 16: Lstm Bidirectional loss graph

- Accuracy Graph:
  Comparing the above accuracy graph with the matching vanilla graph, we observe an almost constant accuracy for different epochs, in both cases. Specifically both models have an accuracy around 0.75, with the vanilla model having the better score at 0.7508 compared to the bidirectional model's score at 74.62%.

  We expected the bidirectional model to be the one with the better accuracy and expect that this slight difference between them is somewhat caused by the resplitting of the data between each model's training. In other words, we assume that in case the train set was the same for both the vanilla and bidirectional lstm models the accuracy score the bidirectional model would be greater.

Figure 17: Lstm Bidirectional accuracy graph

- Classification Report:

```
The accuracy of the model on the test set is: 0.7462
              precision    recall  f1-score   support

    negative       0.76      0.74      0.75       339
    positive       0.73      0.75      0.74       323

    accuracy                           0.75       662
   macro avg       0.75      0.75      0.75       662
weighted avg       0.75      0.75      0.75       662
```

Figure 18: Lstm Bidirectional classification report

As we said before the overall accuracy of the model on the test set is 0.7462. By looking in the class specific scores we can see that the results between the classes are similar, with no one class having better scores for every metric.

4. 2

In this section we implement a bidirectional lstm with an attention mechanism.
The model will be compared with the corresponding lstm network with an attention mechanism produced at section 3.2.

- Loss Graph:

Figure 19: Lstm Attention Bidirectional loss graph

While there are considerably more fluctuations to this model compared to the previous one (attention lstm) there is also a much smaller loss for each corresponding epoch.
Indeed the loss seems to be around 0.55 across epochs while previously there was a considerable increase reaching for the final epoch to 0.9.

- Accuracy Graph:

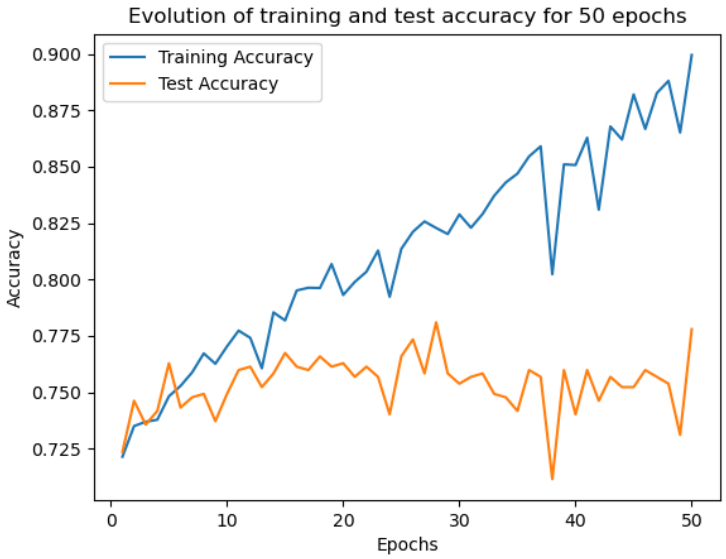

Figure 20: Lstm Attention Bidirectional accuracy graph

We observe that in comparison with the previous attention lstm model, there are considerable more variations for both the test and training accuracy curve.
The final accuracy score is, as expected, considerably greater for this model than the original one, with this one having a score of 0.779 in contrast with the previous lstm with attention model which has a score of 0.726.

- Classification Report:

```
The accuracy of the model on the test set is: 0.7779
              precision    recall  f1-score   support

    negative       0.81      0.76      0.78       352
    positive       0.75      0.80      0.77       310

    accuracy                           0.78       662
   macro avg       0.78      0.78      0.78       662
weighted avg       0.78      0.78      0.78       662
```

Figure 21: Lstm Attention Bidirectional classification report

Almost all metrics for both classes seem to be greater for this model, compared to the previous one. This of course is the expected outcome. Also we note that this is the best model with respect to accuracy.

Question 5

5. 1

We now set the whole training process to be in a python method, which we call `run`, that takes as an argument the model we want to train (e.g. BaselineDNN, LSTMDNN and so on, defined in the models.py script). From the model's name we define a path, where the method will save the model in that path, if it is not already saved. The lines of code to achieve that are the following:

```
load_model = os.path.exists('models/' + model_name)
if load_model:
   ...
else:
   torch.save(model, 'models/' + model_name)
```

In other case, the method will load the model from that directory. Thus, we find the best model by running all models and comparing the results, and then calling that method with the best model as an argument. Since it already exists in the directory, it will be loaded with the following lines of code:

```
model = net["model_class"](output_size, embeddings)
model.load_state_dict(net["model_state_dict"])
```

5. 2

5. 3