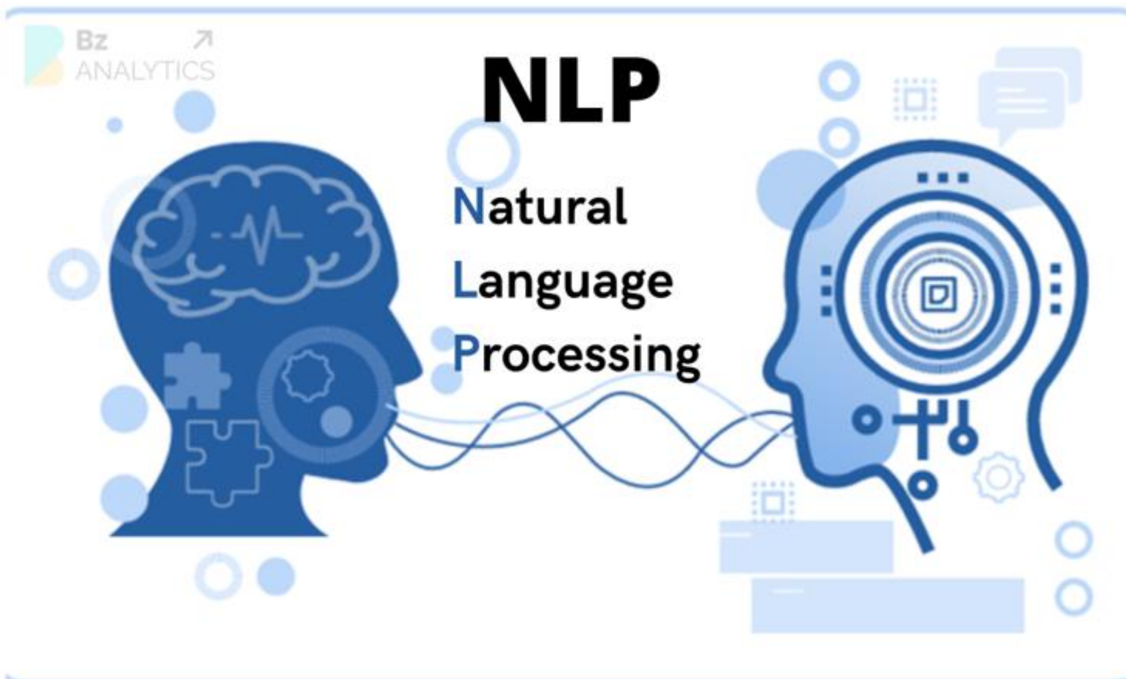
National Technical University of Athens

Voice and Natural Language Processing

Workshop 2: Sentiment Analysis



Authors: Katsaidonis Nikolaos 03121868

Tzamouranis Georgios 03121141

PREPARATORY WORK

! In the following queries, in those that did not require the use of both datasets, Semeval was used!

ISSUE 4

Fill in the blanks at position main.py:EX1 and print the first 10 labels from the training data and their correspondences to numbers.

We print the requests, as well as an example from each label.

```
Mapping κατηγορίας → αριθμού:
negative → 0
neutral → 1
positive → 2

Original labels (πριν το encoding):
[1 2 1 2 2 2 1 2 0 1]

 So the corresponding list with the actual values is:
['neutral', 'positive', 'neutral', 'positive', 'positive', 'positive', 'neutral', 'positive', 'negative', 'neutral']

 Lets print one neutral, one postive and one negative tweet:
Neutral:
 05 Beat it - Michael Jackson - Thriller (25th Anniversary Edition) [HD] http://t.co/A4K2B86PBv
Positive:
 Jay Z joins Instagram with nostalgic tribute to Michael Jackson: Jay Z apparently joined Instagram on Saturday and.. http://t.co/Qj9I4eCvXy
Negative:
 @etbowser do u enjoy his 2nd rate Michael Jackson bit? Honest ques. Like the can't feel face song but god it's so obvious they want MJ 2.0
PS C:\git repo nlp\slp-labs\lab3>
```

DEMAND 2:

Fill in the blanks at position dataloading.py:EX2 and print the first 10 examples from the training data.

We notice how the data is lists of tokens – words:

```
Lets print some tokenized examples:

Tokenized data 0: ['05', 'beat', 'it', '-', 'michael', 'jackson', '-', 'thriller', '(', '25th', 'anniversary', 'edition', ')', '[', 'hd', ']', 'http://t.co/a4k2b86pbv']

Tokenized data 1: ['jay', 'z', 'joins', 'instagram', 'with', 'nostalgic', 'tribute', 'to', 'michael', 'jackson', ':', 'jay', 'z', 'apparently', 'joined', 'instagram', 'on', 'saturday', 'and', '..', 'ht
tp://t.co/qj9i4ecvxy']

Tokenized data 2: ['michael', 'jackson', ':', 'bad', '25th', 'anniversary', 'edition', '(', 'picture', 'vinyl', '):', 'this', 'unique', 'picture', 'disc', 'vinyl', 'includes', 'the', 'original', '1', '
http://t.co/fkxhtoaauw']

Tokenized data 3: ['i', 'liked', 'a', '@youtube', 'video', 'http://t.co/aar3pjp2pi', 'one', 'direction', 'singing', '"', 'man', 'in', 'the', 'mirror', '"', 'by', 'michael', 'jackson', 'in', 'atlanta',
',', 'ga', '[', 'june', '26', ',']

Tokenized data 4: ['18th', 'anniv', 'of', 'princess', "diana's", 'death', '.', 'i', 'still', 'want', 'to', 'believe', 'she', 'is', 'living', 'on', 'a', 'private', 'island', 'away', 'from', 'the', 'publ
ic', '.', 'with', 'michael', 'jackson', '.']

Tokenized data 5: ['@oridaganjazz', 'the', '1st', 'time', 'i', 'heard', 'michael', 'jackson', 'sing', 'was', 'in', 'honolulu', ',', 'hawaii', '@', 'a', 'restaurant', 'on', 'radio', '.', 'it', 'was', 'a
', '.', 'b', '.', 'c', '.', 'i', 'was', '13', '.', 'i', 'loved', 'it', '!']

Tokenized data 6: ["'", 'michael', 'jackson', "'", 'appeared', 'on', 'saturday', '29', 'at', 'the', '9th', 'place', 'in', 'the', 'top', '20', 'of', "miami's", 'trends', ':', 'http://t.co/dxn2fwguhb', '
#trndnl']
```

DEMAND 3

Implement the getitem method of the SentenceDataset class (position dataloading.py:EX3) and print 5 examples in their original format and as returned by the SentenceDataset class.

```
Lets print 5 examples of the encoded sentnces:


Encoded example: [ 17261    961     21     12    786   1755     12   8966     24   8962
   2350   2493     25   2824  12315   5281 400001      0      0      0
      0      0      0      0      0      0      0      0      0      0
      0      0      0      0      0]
Example's label: 1
Example's length: 17

Encoded example: [  4791   9027   7698 109263     18  20557   5079      5    786   1755
     46   4791   9027   1897   1031 109263     14    278      6 400001
 400001      0      0      0      0      0      0      0      0      0
      0      0      0      0      0]
Example's label: 2
Example's length: 21

Encoded example: [   786   1755     46    979   8962   2350   2493     24   1836  11193
 400001     38   3007   1836   5977  11193   1013      1    930    177
 400001      0      0      0      0      0      0      0      0      0
      0      0      0      0      0]
Example's label: 1
Example's length: 21

Encoded example: [     42   5573      8 400001    975 400001     49   2192   4100      9
    301      7      1   6462      9     22    786   1755      7   1098
      2  12132   2824    345   1077      2      0      0      0      0
      0      0      0      0      0]
Example's label: 2
Example's length: 26
```

Now, encoded data is lists of numbers where each number assigns each word to the corresponding embedding vector. We notice that the length is many times shorter than the size of the list as this is the length before the zero padding we do so that all embeddings are max_length long.

ISSUE 4

1) **Why do we initialize the embedding layer with pre-trained word embeddings?**

Because pre-trained embeddings (such as GloVe) have been **trained on huge bodies of text** (e.g. Wikipedia) already containing **semantic and syntactic information**. They allow us to **start with "linguistic knowledge"** ready-made, without learning it from scratch. That is, from making from scratch the word vectors that contain information about the correlation of words, We get these correlations ready. This can greatly increase performance, especially when our dataset is small or we don't have the time to train such a system.

2) **Why do we keep embedding layer weights frozen during training?**

If we let embeddings **be trainable**, they can be altered and lose their semantic information. Especially if we have **little data**, the model can "unlearn" or overfit. Thus, we trust the embeddings that have resulted from studying large data sets even if they are likely not good enough for the specific type of data we have (specifically for tweets).

ISSUE 5

Why do we put a nonlinear activation function in the penultimate layer? What difference would it make if we had 2 or more linear transformations in a row?

In general, we know that nonlinear activation functions, such as ReLU, tanh, etc., are suitable for creating more complex pattern-relationships on the data. On the other hand, linear activation functions are of the form:

$$\mathbf{y = Wx + b}$$

However, for data with complex correlations such as text they are not suitable. Considering now the possibility of listing several linear activation functions in order, we conclude that the result is equally a linear transformation of the input. For example, for n=2 linear AFs in a row we would have:

$$\mathbf{y = (W2 \cdot W1)x + (W2 \cdot b1 + b2) \equiv Wx + b}$$

Therefore, we would again have a linear result without the possibility of learning more complex patterns.

ISSUE 6

1) **If we consider that each dimension of the embedding space corresponds to an abstract concept, can you give an intuitive**

interpretation of what the representation you made describes (center-weight)?

What we did was to put each pot into tokens and then each token corresponded to a vector of dimension e.g. 50. Therefore, we have for each sentence 32 vectors (one) for each word. We take the mean of this of 32 vectors, and so each column of this vector is added up and divided by the number of non-zero values. In essence, what we did was to give an equivalent "weight" to each word/token without emphasizing the meaning of a word, and so we find something like the center of gravity of the sentence.

2) **Mention possible weaknesses of this approach to represent texts.**

The above approach may be simple and fast, but it may not lead to desired results. For example, the attribution of equal weight to articles, punctuation, nouns, verbs, etc. It may not be the smartest way to handle the situation. In addition, words at the beginning or end of the sentence may not be as semantic in relation to "medium" words. In general, mean pooling lacks in the way it attaches weight to the words of sentences both in terms of semantics and word order.

ISSUE 7

1) **What are the consequences of small and large mini-batches on model training?**

   The logic behind mini batches is that we bring in data "by batches" instead of one-by-one. This results in a gain in performance/time, especially in the case of a GPU that supports parallel processing. In addition, weights, which are updated per mini-batch, are now updated by data groups instead of being updated on each new data that comes in one by one. In this way, we lead the model to learn from more data before frivolously renewing its weights and calculating the gradients for each data.

Based on the above, the smaller the mini batches, the more noisy the weight modifications are since they are made on a small part of the data. In addition, small batches may help us avoid local minimums. On the other hand, for large mini batches we have more stable gradients while we may "stick" to local minimums more easily.

2) **We usually shuffle the order of mini-batches in the training data in each season. Can you explain why?**

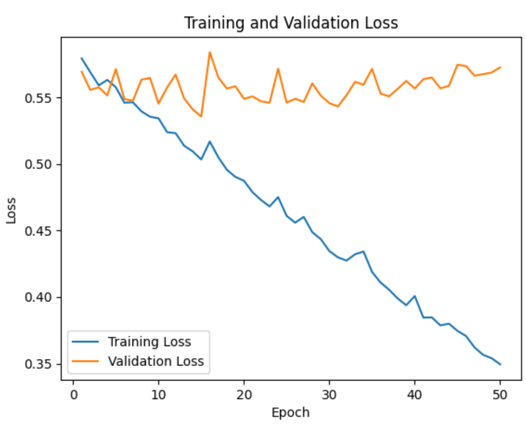
Bringing the batches in the same order certainly doesn't help the model learn the data no matter what order it comes in. By shuffling the data, we lead the model to ignore the sequence in which it comes and help it learn instead of "memorize."

DEMAND 10

For each of the 2 datasets provided, report the model's performance in the metrics: accuracy, F1 score (macro average), recall (macro average). Also, create graphs, showing the model's training and test loss curves per season.

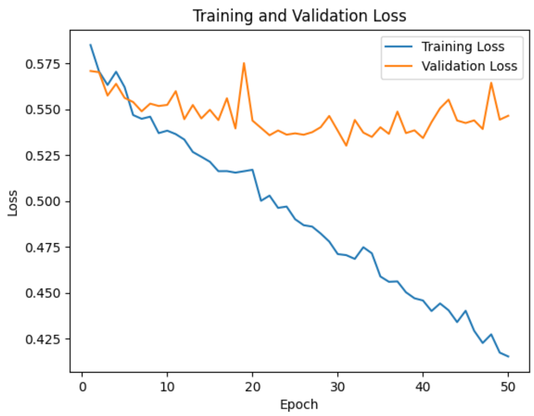
For each of the 2 data sets, we trained our model. Noticing overfitting mainly in the Seemal dataset, we decided to add **dropouts**. So, we trained the models for each data set with and without dropouts. For the metrics listed, we preferred to choose the **metrics of the best era**, i.e. the one with the smallest loss and print them. In addition, we made loss-epochs charts which we list below:

- **MR dataset without dropout:**




```
Best epoch: 31
Accuracy: 0.7039274924471299
F1: 0.7009550851832731
Recall: 0.712371085024476
```

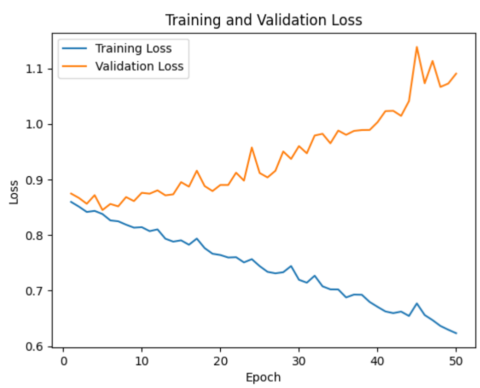
MR dataset with dropout:




```
Best epoch: 31
Accuracy: 0.7084592145015106
F1: 0.705962630286544
Recall: 0.7157879818594104
```

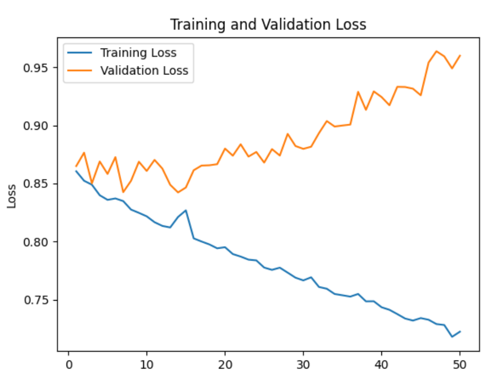
We notice that the dropout did not improve our model in the case of the MR dataset.

- **Semeval dataset without dropout:**




```
Best epoch: 5
Accuracy: 0.5990719635297949
F1: 0.5722971038767537
Recall: 0.5977653503880589
```

- **Semeval dataset with dropout:**




```
Best epoch: 14
Accuracy: 0.6023282318463041
F1: 0.5798631991790472
Recall: 0.5970020934442773
```

We noticed that in the Semeval dataset the validation loss has a large increase which indicates overfitting since the model does not perform as well in testing as in the train (train loss is constantly falling). With the addition of dropouts, we managed to get a slightly better model both in terms of loss and accuracy.

Issue 11 – MR Dataset: Emotion Recognition with ChatGPT

As part of Issue 11, an experiment was carried out on emotion recognition using the ChatGPT model, using the MR (Movie Reviews) dataset. The dataset included film reviews categorized into two categories: positive and negative.

Procedure

- 20 proposals from each category (positive/negative) were selected, a total of 40 proposals.

- Sentences were printed using scripts and saved in a JSON file (samples_MR.json).

- Suggestions were submitted to ChatGPT with four different prompts.

- The model's responses were evaluated for accuracy based on the actual labels.

Prompts Used

Prompt 1: Simple sorting without justification, with 1 word answer per sentence.

Below I give you film reviews. For each one, tell me if it is "positive" or "negative".

1. A visually stunning yet emotionally hollow film.

2. The performances were top-notch and the script witty.

3. ...

Prompt 2: Sort with a short justification for each sentence.

For each of the film reviews below, please tell me:

1. Whether it's "positive" or "negative"

2. A brief justification for your decision

Reviews:

1. ...

2. ...

Prompt 3: Sort by providing keywords that influenced the decision.

I want you to analyze the reviews below. For each:

1. Tell me if it's "positive" or "negative"

2. Identify the most important words or phrases that influenced your decision

3. Briefly explain why you came to this conclusion

Prompt 4: Sort in table format with sentiment, keywords, justification.

For each of the reviews below, put the results in a table with the following columns:

| Number | Emotion (positive/negative) | Keywords | Justification |

Accuracy Results

Prompt	Accuracy
Prompt 1 – Simple Answer	45.0%
Prompt 2 – With justification	92.5%
Prompt 3 – With keywords & justification	90.0%
Prompt 4 – Table with keywords & explanation	90.0%

Error Analysis

Prompt 1 showed the lowest accuracy, as it did not provide ChatGPT with any guidance or context beyond the sentence. In contrast, Prompts 2–4 performed much higher (>90%).

Justification of Answers and Keywords

Prompts 2–4 provided valuable information on how to make a decision about the model. The keywords identified by ChatGPT were often associated with strong positive or negative sentiment. Examples:

Positive keywords: great, charming, stunning, intelligent, entertaining

Negative keywords: weak, forgettable, pointless, boring, cliché

Also, the model provided relevant justifications for most of the responses, such as "the criticism refers to a weak plot and mediocre direction".

Conclusions

The experimental process showed that ChatGPT's performance in emotion classification is significantly influenced by how clear and explanatory the prompt is. Providing context, keywords, and justifications visibly improves the quality of the classification.

Question 11 – Semeval2017A: Emotion Recognition with ChatGPT

For the second part of Issue 11, we applied sentiment recognition techniques with ChatGPT using the Semeval 2017 Task 4A dataset. This dataset includes short tweets categorized into three categories: positive, negative, and neutral.

Procedure

- At least 20 tweets from each category were selected, for a total of 60.

- Proposals were printed and saved in JSON for reference (samples_Semeval2017A.json).

- Tweets were analyzed by ChatGPT using four different prompts.

- The predictions were recorded and the accuracy for each prompt was calculated.

Prompts Used

Prompt 1: Simple sorting, one word per tweet.

Below I give you tweets. For each one, tell me if it's "positive," "negative," or "neutral."

He responded with a single word for each sentence.

1. Just watched the keynote! Impressive work by Apple.

2. This service keeps getting worse. I’m tired of it.

3. Well, the day went by faster than I expected.

...

Prompt 2: Sort and justify each tweet.

For each of the following tweets:

1. Tell me the feeling ("positive", "negative", "neutral")

2. Offer a brief justification for your choice

Tweets:

1. ...

2. ...

Prompt 3: Sort by keywords and brief explanation.

For each of the tweets:

1. Tell me if it's "positive", "negative", or "neutral"

2. Write 2-3 keywords that influenced your decision

3. A brief explanation of why you ended up in this category

Prompt 4: Complete table with columns for sentiment, keywords, and justification.

For each tweet, fill in the following in a table:

| No. | Emotion (positive/negative/neutral) | Keywords | Explanation |

Accuracy Results

Prompt	Accuracy
Prompt 1 – One word	50.0%
Prompt 2 – With justification	33.3%
Prompt 3 – With Keywords	58.3%
Prompt 4 – Table	70.0%

Analysis and Errors

There was significant variation in performance between the different prompts. Prompt 2 had the weakest performance probably due to unclear instructions and inconsistent interpretation. Prompt 4 performed best, demonstrating the importance of structured presentation of information to the model.

Important Words and Interpretations
The results of Prompt 3 and 4 show that ChatGPT was mainly based on words such as: "love", "awesome", "great", "boring", "worst", "alone", "yay", "#fail", "unbelievable", etc. Several neutral tweets were misclassified due to ambiguous expressions or irony.

Conclusions
The analysis of the Semeval dataset highlighted the difficulty of classifying neutral sentences from the model, due to the lack of clear emotional content. However, it was found that the structure and completeness of the prompt significantly improves accuracy. ChatGPT provided adequate justifications and keywords when asked, which enhances the interpretability of the results.

QUESTIONS

Question 1

1.1) Calculate the representation of each sentence u (Equation 2) as the concatenation of the mean pooling and the maximum pooling of the wordembeddings of each sentence, E = (e1, e2, ..., eN).

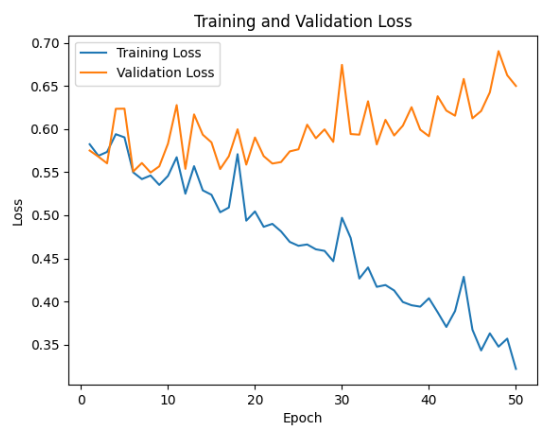
$$u = [\mathrm{mean}(E) \| \max(E)]$$

1.2) What is the difference(s) between this representation and the original? What more information could it extract? Answer briefly.

The mean pooling representation that we had so far gave a general idea of the meaning of the sentence which was based on the equal meaning of each word as long as we passed the average. Now, with the merger of mean-max pooling, we get for each sentence a vector of twice the dimensions that contains information about the most "important" words of the sentences. With max pooling, we keep the maximum value of all words per dimension and thus add to the model the ability to give weight to words with great emotional – semantic interest.

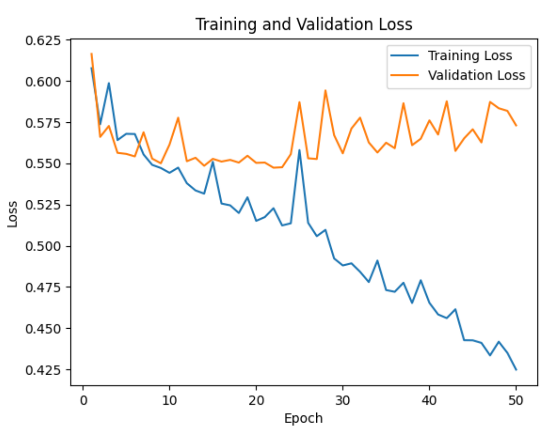
Below, we list the corresponding charts and metrics as in previous queries, using **mean-max pooling**:

- **MR dataset without dropout and mean-max pooling:**



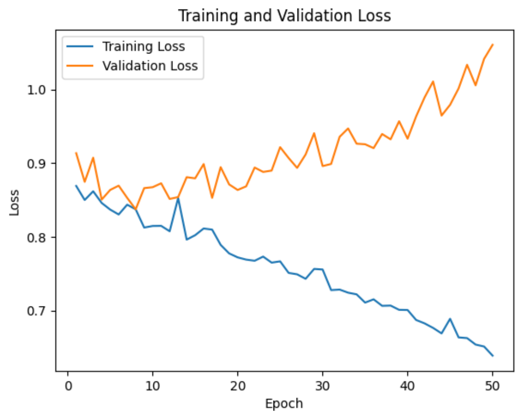

```
Best epoch: 10
Accuracy: 0.6888217522658611
F1: 0.6871771344680461
Recall: 0.6928778541260711
```

- **MR dataset with dropout and mean-max pooling:**



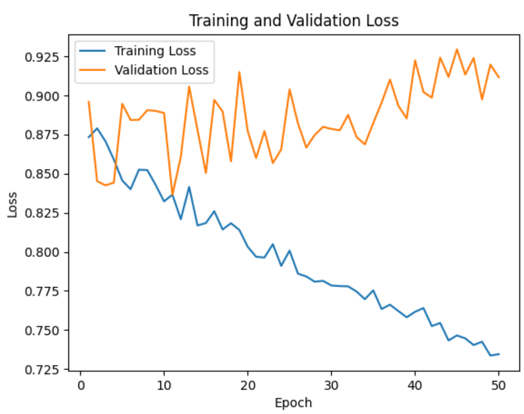

```
Best epoch: 22
Accuracy: 0.6782477341389728
F1: 0.6725930225808324
Recall: 0.6914758020236882
```

- **Semeval dataset without dropout and mean-max pooling:**




```
Best epoch: 8
Accuracy: 0.6011071312276132
F1: 0.5807617160491193
Recall: 0.5952204843941504
```

- **Semeval dataset with dropout and mean-max pooling:**




```
Best epoch: 11
Accuracy: 0.6055845001628134
F1: 0.5749072818015479
Recall: 0.6079614969092957
```

Based on our results, accuracy using mean-max pooling improved (by a small percentage) for the Semeval dataset model but not for the MR dataset. It is also worth noting that in general, dropout models are slightly better in both loss and accuracy. In cases where the dropout model is slightly worse, more seasons should probably be used so that the model can learn better.

For a better visualization of the results, we list the above conclusions regarding the metrics in tables:

- **MR dataset:**

-	-	**Accuracy**	**F1 score**	**Recall**
	not dropout	0.7039	0.7009	0.7124

mean pooling	**dropout**	0.7085	0.7059	0.7158
mean-max pooling	**not dropout**	0.6888	0.6872	0.6929
	dropout	0.6782	0.6726	0.6915

- **Semeval dataset:**

-	-	Accuracy	F1 score	Recall
mean pooling	**not dropout**	0.5991	0.5723	0.5978
	dropout	0.6023	0.5799	0.5970
mean-max pooling	**not dropout**	0.6011	0.5808	0.5952
	dropout	0.6056	0.5749	0.6080

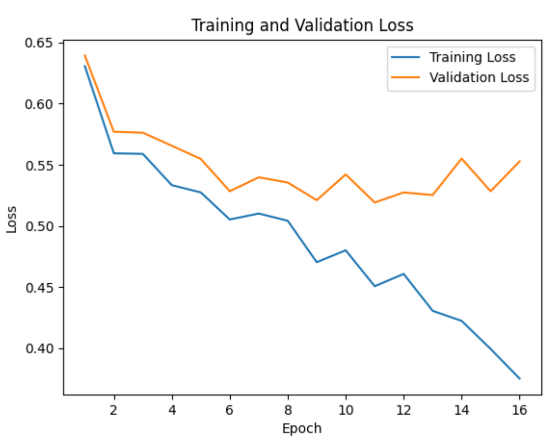
Question 2

Then, we will experiment with the use of retrospective neural networks, specifically **LSTMs**. As we are asked, during training, we will use **early stopping** after 5 consecutive increases in validation loss.

Bidirectional LSTM is an extension of classical LSTM, which allows the network to process input in both directions, both left-to-right and right-to-left. In this way, Each word acquires a representation that incorporates information from the entire context of the sentence. This approach is especially useful in language tasks such as sentiment analysis, where the meaning of a word can be influenced by both previous and subsequent words.

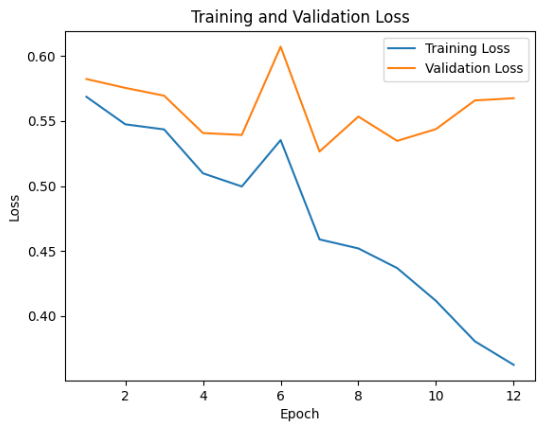
Below, we list the validation/test loss charts for LSTM using early stopping as well as the metrics we extracted from the test set.

- **MR dataset LSTM:**



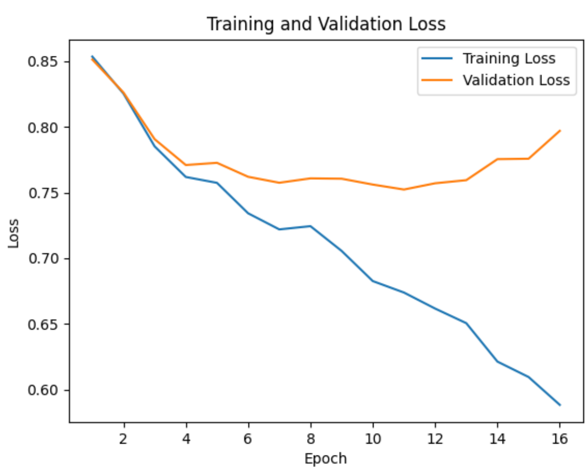

```
Early stopping triggered at epoch 16

✅ Final Evaluation (Test Set):
Test Loss: 0.5021
Accuracy: 0.7432
F1 Score (macro): 0.7432
Recall (macro): 0.7433
```

- **MR LSTM dataset with bidirectional:**



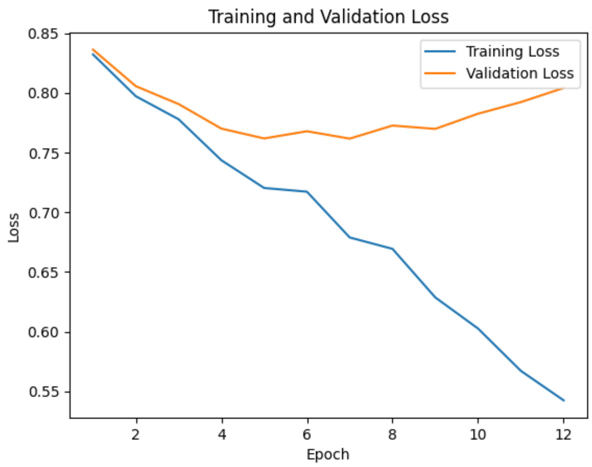

```
Early stopping triggered at epoch 12

✅ Final Evaluation (Test Set):
Test Loss: 0.5010
Accuracy: 0.7447
F1 Score (macro): 0.7438
Recall (macro): 0.7482
```

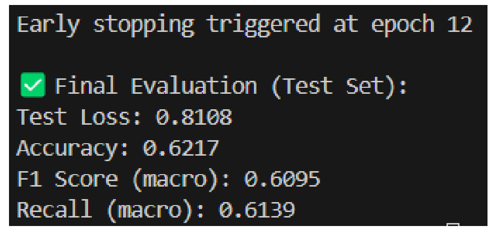
- **Semeval dataset LSTM:**




```
Early stopping triggered at epoch 16

✅ Final Evaluation (Test Set):
Test Loss: 0.8467
Accuracy: 0.6030
F1 Score (macro): 0.5907
Recall (macro): 0.6035
```

- **Semeval LSTM dataset with bidirectional:**





- **MR dataset LSTM:**

	Accuracy	**F1 score**	**Recall**
not bidirectional	0.7432	0.7432	0.7433
bidirectional	0.7447	0.7438	0.7482

- **Semeval dataset LSTM:**

	Accuracy	**F1 score**	**Recall**
not bidirectional	0.6030	0.5907	0.6035

bidirectional	0.6217	0.6095	0.6139

We observe that bidirectional LSTM increases the quality of the model by a minimum in both datasets compared to simple LSTM, which was to be expected as words are affected by both subsequent and previous words.

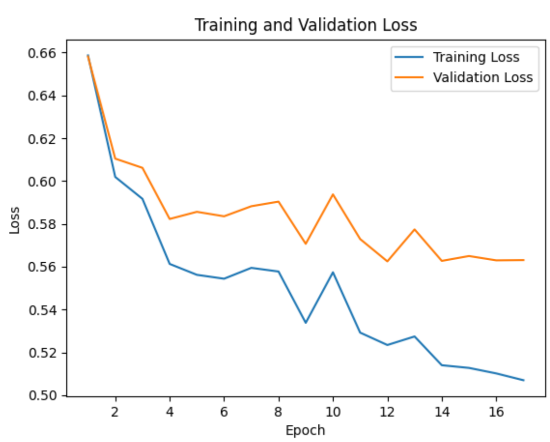
In addition, from the curves it can be seen that the attention mechanism significantly reduced overfitting compared to previous models. (descending curve)

It is worth noting that in the case of the MR dataset, there was an increase in accuracy compared to the simple DNN model. However, in Semeval we got almost the same (a few centimeters worse) metrics compared to DNN. However, we remind you that the metrics we printed in DNN refer to the validaton per era keeping the best of them (minimum loss) and therefore the comparison remains partially unfair for the LSTM metrics.

Question 3

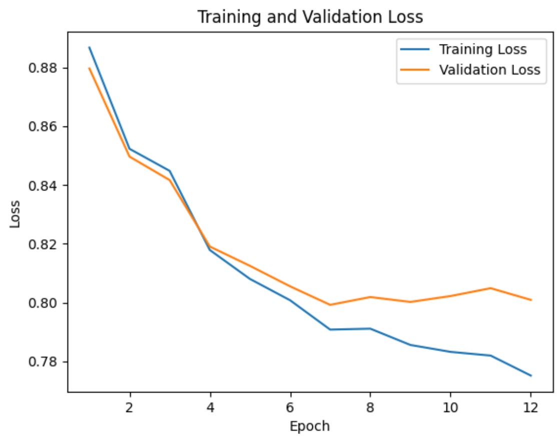
In this question we use an attention mechanism. We applied average pooling to the final representations. Compared to LSTM, we noticed a big difference in training time, which decreased significantly.

- **MR dataset Attention:**




```
Early stopping triggered at epoch 17

Final Evaluation (Test Set):
Test Loss: 0.5324
Accuracy: 0.7145
F1 Score (macro): 0.7145
Recall (macro): 0.7146
```

- **Semival dataset Attention:**




```
Early stopping triggered at epoch 12

Final Evaluation (Test Set):
Test Loss: 0.8467
Accuracy: 0.6055
F1 Score (macro): 0.5807
Recall (macro): 0.6082
```

3.2) What are the queries, keys and values that exist in the attention class. Head and the position_embeddings defined in attention. SimpleSelfAttentionModel; You can consult the tutorial "Let's build GPT: from scratch, in code, spelled out" for your answer.

In the context of the self-attention mechanism, queries (Q), keys (K) and values (V) are different views (linear transformations) of the word embeddings of a sentence. More specifically:

- Keys: They encode the contents of each word — "what each word means."
- Queries: They express the information that each word is "looking for" — "what I'm paying attention to."
- Values: This is the actual information that will be gathered, weighted based on attention scores.

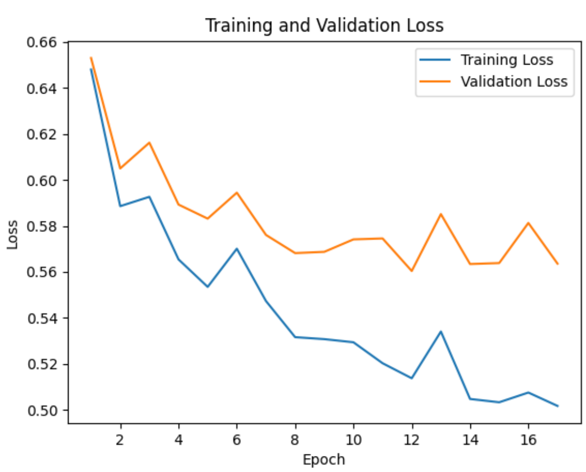
Essentially, each word compares its query with all the keys (through an internal product) in order to decide which words to pay attention to. Then, a weighted average of the corresponding values is calculated, thus giving the attention output.

In addition, the position embeddings present in the SimpleSelfAttentionModel class are added to word embeddings to incorporate position information (i.e., word series) into the input, as self-attention itself is independent of the sequence of tokens.

Question 4

In this question, we train a multihead attention mechanism and print the model's performance. Specifically, we select 3 heads of equal size each (head_size = dim // n_head). Each head essentially processes a different part of the vectors (since we have different keys, values, queries and therefore different weight tables for each head) and therefore extracts information of different semantics. (e.g. another head for verbs, another for aggressive determinations, etc.)

- **MR dataset Multihead Attention:**




```
Early stopping triggered at epoch 17

Final Evaluation (Test Set):
Test Loss: 0.5243
Accuracy: 0.7221
F1 Score (macro): 0.7218
Recall (macro): 0.7227
```

- **Semival dataset Multihead Attention:**


```
Early stopping triggered at epoch 23

Final Evaluation (Test Set):
Test Loss: 0.8145
Accuracy: 0.6228
F1 Score (macro): 0.6038
Recall (macro): 0.6242
```

- **MR dataset Attention mechanisms:**

	Accuracy	**F1 score**	**Recall**
One head	0.7145	0.7145	0.7146
Multihead	0.7221	0.7218	0.7227

- **Semeval dataset Attention mechanisms:**

	Accuracy	**F1 score**	**Recall**
One head	0.6055	0.5807	0.6082
Multihead	0.6228	0.6038	0.6232

As can be seen from the tables above, the multi-head coupling improved the performance of our model. This is to be expected, as the multi-head attention mechanism allows the model to focus on different aspects of the proposal simultaneously through multiple attention heads. Each head can learn different patterns or relationships between words, thus enhancing the expressiveness of the final representation of the sentence.

Question 5

5.1)What is the difference between Transformer and MultiHead AttentionModel?

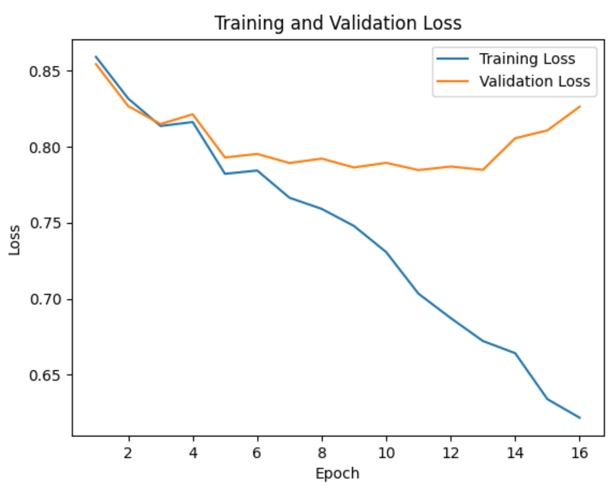
In this question, we will experiment with the use of the Transformer model. This model is based exclusively on the attention mechanism, but differs in its complexity. While the Multi Head Attention model applies to the input once **attention** and then **feedforward** (Linear - > ReLU - > Linear -> Dropout) while in Transformer we have **multiple repeating blocks** (layers) from such levels. Thus, he has the ability to understand more complex relationships since attention is applied several times more.

First, we made a Transformer **with n_heads = 3 and n_layer=3.** Below we list our results:

- **MR dataset Transformer:**


```
Final Evaluation (Test Set):
Test Loss: 0.5377
Accuracy: 0.7372
F1 Score (macro): 0.7372
Recall (macro): 0.7372
```

- **Semeval dataset Transformer:**



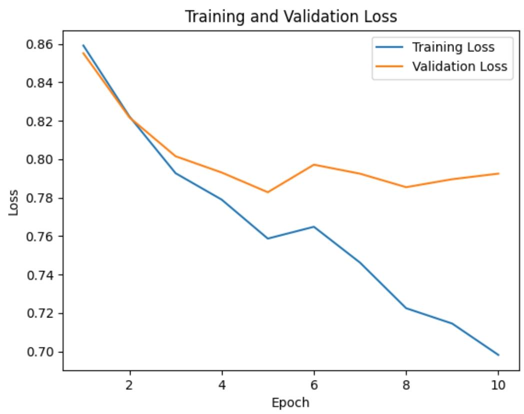

```
Early stopping triggered at epoch 16

Final Evaluation (Test Set):
Test Loss: 0.8159
Accuracy: 0.6232
F1 Score (macro): 0.6050
Recall (macro): 0.6204
```

Compared to the MultiHead model, the Transformer model slightly increased performance in both data sets. **Aggregate comparisons will be presented in tables below.**

5.2) Do you experiment with different parameter values?

We chose the **Semeval** data set to experiment with the parameters:

- **n_heads=6, n_layer = 3**

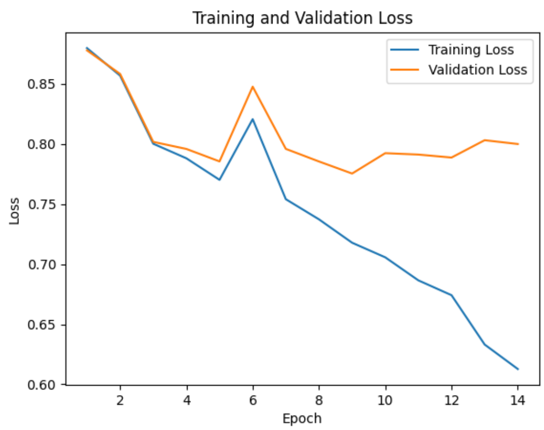


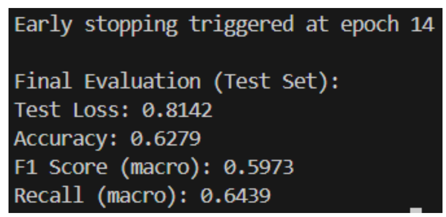

```
Early stopping triggered at epoch 10

Final Evaluation (Test Set):
Test Loss: 0.8302
Accuracy: 0.6221
F1 Score (macro): 0.5980
Recall (macro): 0.6278
```

- **n_heads=8, n_layer = 4**


```
Early stopping triggered at epoch 16

Final Evaluation (Test Set):
Test Loss: 0.8234
Accuracy: 0.6233
F1 Score (macro): 0.6144
Recall (macro): 0.6112
```

- **n_heads=8, n_layer = 6 (default)**





```
Early stopping triggered at epoch 14

Final Evaluation (Test Set):
Test Loss: 0.8142
Accuracy: 0.6279
F1 Score (macro): 0.5973
Recall (macro): 0.6439
```

5.3) What are their default values in the classic Transformer architecture?

Based on the paper "Attention is ALL You Need" the default parameters are:

n_head = 8

n_layer = 6

embedding_dim = 512

feedforward_dim = 2048 (4 x embding_dim)

dropout = 0.1

As you will see above, we also tried n_head=8, n_layer=6 but the results are obviously not comparable to paper since we have different dimensions of embeddings.

Semeval dataset:

	Accuracy	F1 score	Recall
n_head=3 **n_layer=3**	0.6232	0.6050	0.6204
n_head=6 **n_layer=3**	0.6221	0.5980	0.6278
n_head=8 **n_layer=4**	0.6233	0.6144	0.6112
n_head=8 **n_layer=6**	0.6279	0.5973	0.6439

We note that the more we increased the hyperparameters, the more the training time increased.

In the case where we increased the number of heads from 3 to 6, we got slightly worse results. This may be related to the overfitting caused. The third experiment is barely better than the first, while the default values

Total comparison tables Attenuation – Multi Attenuation - Transformer for each dataset:

- **MR dataset:**

	Accuracy	F1 score	Recall
One head	0.7145	0.7145	0.7146

Multihead	0.7221	0.7218	0.7227
Transform	0.7372	0.7372	0.7372

- **Semeval dataset (keeping best parameters performance):**

	Accuracy	**F1 score**	**Recall**
One head	0.6055	0.5807	0.6082
Multihead	0.6228	0.6038	0.6232
Transform	0.6279	0.5973	0.6439

In both datasets, the **Transformer model appears to be slightly better** compared to simple attention mechanisms (judging by accuracy).

Question 6

In this query you will use Pre-Trained Transformer models to categorize emotion. Select at least 3 models for each dataset and extend the code appropriately. Compare their performance and tabulate the results.

As requested, we selected 3 models from the **Hugging Face Model Hub** (https://huggingface.co/models) for each dataset. We made sure that they were trained in sentiment analysis as well as that the number of their tags fit each dataset.

- For the **MR** dataset we have chosen the following, which we list along with the metrics we obtained during testing:

Model 1: siebert/sentiment-roberta-large-english

```
Dataset: MR
Pre-Trained model: siebert/sentiment-roberta-large-english
Test set evaluation
  accuracy: 0.9259818731117825
  recall: 0.9259818731117825
  f1-score: 0.9259803530069484
```

Model 2: distilbert-base-uncased-finetuned-sst-2-english

```
Dataset: MR
Pre-Trained model: distilbert-base-uncased-finetuned-sst-2-english
Test set evaluation
  accuracy: 0.8912386706948641
  recall: 0.8912386706948641
  f1-score: 0.891213847502191
```

Model 3: textattack/bert-base-uncased-SST-2

```
Dataset: MR
Pre-Trained model: textattack/bert-base-uncased-SST-2
Test set evaluation
  accuracy: 0.8987915407854985
  recall: 0.8987915407854985
  f1-score: 0.898739552393846
```

	Accuracy	**F1 score**	**Recall**
Siebert	0.926	0.926	0.926
distilbert	0.891	0.891	0.891
textattack	0.899	0.899	0.899

Based on accuracy, the **siebert model** is considered the best for the MR dataset with a very high accuracy rate!

The **siebert**, **distilbert,** and **textattack** models are pre-trained to analyze sentiment over different datasets. The former has been trained in a variety of sources (Amazon, Yelp, **IMDB**, etc.) for general use, while the other two are trained specifically in SST-2 (Stanford Sentiment Treebank), which **includes reviews of movies** labeled positive/negative. **Classification of 2 categories.**

The fact that they are trained specifically for films justifies the high performance of all three.

- For the **Semeval** dataset we chose the following, which we list along with the metrics we got during testing:

Model 1: cardiffnlp/twitter-roberta-base-sentiment

```
Dataset: Semeval2017A
Pre-Trained model: cardiffnlp/twitter-roberta-base-sentiment
Test set evaluation
  accuracy: 0.7237870400521003
  recall: 0.7229454214750545
  f1-score: 0.7222115953560642
PS C:\git_repo_nlp\slp-labs\lab3>
```

Model 2: finiteautomata/bertweet-base-sentiment-analysis

```
Dataset: Semeval2017A
Pre-Trained model: finiteautomata/bertweet-base-sentiment-analysis
Test set evaluation
  accuracy: 0.7177629436665581
  recall: 0.7301871228078923
  f1-score: 0.718050644575488
```

Model 3: yiyanghkust/finbert-tone

```
Dataset: Semeval2017A
Pre-Trained model: yiyanghkust/finbert-tone
Test set evaluation
  accuracy: 0.5097688049495278
  recall: 0.3866509029782952
  f1-score: 0.3329611976374523
```

	Accuracy	**F1 score**	**Recall**
cardiffnlp	0.724	0.723	0.722
finiteautomata	0.718	0.730	0.718
yiyanghkust	0.510	0.387	0.333

Based on accuracy, the cardiffnlp model is considered the best for the Semeval dataset. On the other hand, yiyannghkust has very low accuracy, which we expected since it is trained in financial communication texts and the contrast with tweets is obvious.

Question 7

In this query you will train/fine-tune Pre-Trained Transformer models to categorize emotion. Leverage the code you'll find in finetune_pretrained.py file and run locally with a limited dataset for a few seasons (as given). Transfer the code to a notebook in Google Colab8, convert it appropriately, and fine-tune for optimal performance. Test at least 3 models for each dataset and report the results in tables.

In this question, we will take the models we have already used (for comparable results) and fine-tune them. Fine tuning is the process where we take models already trained on huge datasets and train them additionally on our own dataset. Thus, models who have learned to generalize well enough and "know" many words, acquire a specialization in our own data set in order to do our work.

Initially in the finetune_pretrained file we will fill in the code that we need by training for a few seasons and then we will move on to Collab for using GPUs with more seasons.

We ran locally for the **MR dataset** 5 seasons and fine-tuned the **textattack** model. Then we did a final test on the test data to have a better comparison. We got an accuracy of 87.61%.

```
Final test accuracy: 0.8761
PS C:\git_repo_nlp\slp-labs\lab3>
```

The training time was quite long. We tried to run on GPUs locally but got error for space in VRAM. We continue with the code in Collab, which will also be given to the files as "fine_tuning.ipynb" executed so you can see the results. In collab, we changed the **n_samples=40 to 100** and train for **5 seasons** since with the entire dataset we got a run time error.

We list the performance of each model in testing (after training) in tables for the 2 datasets:

- **MR dataset:**

Model	**Accuracy**
Siebert	0.926
Fine tuned siebert	0.863
distilbert	0.891
Fine tuned distilbert	0.896

textattack	0.899
Fine tuned textattack	0.884

- **Semeval dataset:**

Model	**Accuracy**
cardiffnlp	0.724
Fine tuned cardiffnlp	0.690
finiteautomata	0.718
Fine tuned finiteautomata	0.708

yiyanghkust	0.510
Fine tuned yiyanghkust	0.414

It is obvious that the decrease in the performance of some models is due to the small part of the data sets that we were forced to use due to a lack of resources. The model in this case overfits the few samples we provided.

Then we tried to increase the number **of n_samples** and the number of **seasons from 5 to 15** and retrain the models in order to get better performance. However, due to a lack of resources, we were unable to run the training at Collab. However, after several attempts, we trained the small yiyanghkust model on **more samples with 10 seasons** and got an accuracy of 0.492 (**8% higher than the previous fine tuned model**)

It seems logical that **as we increase the number of seasons and samples, in general fine tuning is trained better and better in our own datasets and we get a better result.**

Question 8

The code for the tutorial "Let's build GPT: from scratch, in code, spelled out", mentioned above, is available here https://colab.research.google.com/drive/1JMLa53HDuA-i7ZBmqV7ZnA3c_fvtXnx-?usp=sharing.

Log in to ChatGPT10 and ask:

- **explain the code to you in detail**
- **evaluate the code**
- **rewrite some parts of it (refactoring)**

Cite examples from the dialogue you conducted and comment on its performance in the above.

We gave the code to chatgpt, mainly parts that needed further explanation. Below we list some parts of our dialogue:

Example 1: Explanation of the Bigram Language Model

User:
Can you explain to me what BigramLanguageModel does?

ChatGPT:
Yes! BigramLanguageModel generates predictions for the next character based only on the current character. It has an embedding table of dimensions (vocab_size, vocab_size) and is trained to learn the probability of switching from one character to the next.

Example 2: Refactoring

User:
generate() seems very repetitive. Can you make it clearer?

ChatGPT:
Definitely! We can extract the track:

```python
logits = self(idx_cond)
logits = logits[:, -1, :]
probs = F.softmax(logits, dim=-1)
```

Example 3: Evaluating the Code

User:
What do you think of this code? Would you consider it clean?

ChatGPT:
The code is extremely clean and instructive. It uses good practices such as modular classes, dropouts, layer norms, and is ideal for educational purposes. However, for production it could be refactored into separate files/modules.

Example 4: Explaining the Model Training Part

User:
Can you explain to me how the generate() function that generates new text works? Especially this part: (code given)

ChatGPT:

```python
def generate(self, idx, max_new_tokens):
    for _ in range(max_new_tokens):
        logits = self(idx)
        logits = logits[:, -1, :]  # get the logits for the last token
        probs = F.softmax(logits, dim=-1)
        idx_next = torch.multinomial(probs, 1)  # sample from the probabilities
        idx = torch.cat((idx, idx_next), dim=1)  # append the predicted token
    return idx
```

1. **Επανάληψη για Κάθε Νέο Token:**
   `for _ in range(max_new_tokens):` – Η συνάρτηση θα δημιουργήσει μέχρι `max_new_tokens` νέα tokens, όπου κάθε νέο token εξαρτάται από τα προηγούμενα.
2. **Πρόβλεψη του Επόμενου Token:**
   `logits = self(idx)` – Χρησιμοποιώντας το τρέχον `idx` (εισαγωγή που περιλαμβάνει ήδη τα προηγούμενα tokens), το μοντέλο προβλέπει τα logits για το επόμενο token.
3. **Επιλογή του Τελευταίου Token:**
   `logits = logits[:, -1, :]` – Εδώ επιλέγουμε μόνο το τελευταίο token από την έξοδο, γιατί το μοντέλο κάνει προβλέψεις για κάθε token ξεχωριστά, και εμείς θέλουμε μόνο την πρόβλεψη για το επόμενο.
4. **Υπολογισμός Πιθανοτήτων:**
   `probs = F.softmax(logits, dim=-1)` – Με τη συνάρτηση **softmax**, τα logits μετατρέπονται σε πιθανότητες για το επόμενο token. Κάθε πιθανότητα αντιπροσωπεύει την πιθανότητα του κάθε χαρακτήρα να είναι το επόμενο στο κείμενο.
5. **Δειγματοληψία από τις Πιθανοτητες:**
   `idx_next = torch.multinomial(probs, 1)` – Αντί να επιλέγουμε πάντα την πιο πιθανή πρόβλεψη, κάνουμε **δειγματοληψία** (sampling) από την κατανομή πιθανοτήτων για να προσδώσουμε **τυχαία ποικιλία** στο παραγόμενο κείμενο.
6. **Ενημέρωση Εισόδου:**
   `idx = torch.cat((idx, idx_next), dim=1)` – Ενημερώνουμε το `idx` με το νέο προβλεπόμενο token για να το χρησιμοποιήσουμε στην επόμενη επανάληψη και να παράγουμε το επόμενο token.

Example 5: Explaining the Model Training Part

User:
Can you explain to me what exactly this part of the code does, which is about training the model?

ChatGPT:

python Αντιγραφή

```python
for iter in range(max_iters):
    # forward pass
    logits, loss = model(x_batch, y_batch)

    # backward pass
    loss.backward()

    # update weights
    optimizer.step()
    optimizer.zero_grad()

    # log progress
    if iter % log_interval == 0:
        print(f"iter {iter}/{max_iters}, loss: {loss.item()}")
```

Αυτό το κομμάτι είναι υπεύθυνο για την **εκπαίδευση** του μοντέλου σε κάθε επανάληψη. Αναλύοντας βήμα-βήμα:

1. **Προβλέψεις (Forward Pass):**
   `logits, loss = model(x_batch, y_batch)` – Η είσοδος είναι το `x_batch` (τα δεδομένα εισόδου) και το `y_batch` (οι ετικέτες). Το μοντέλο κάνει προβλέψεις για την επόμενη τιμή και υπολογίζει το `loss` βάσει της διαφοράς με το πραγματικό αποτέλεσμα.
2. **Οπισθοδιάδοση (Backward Pass):**
   `loss.backward()` – Υπολογίζεται το **gradient** του loss σε σχέση με τα βάρη του μοντέλου χρησιμοποιώντας την αλγορίθμο οπισθοδιάδοσης (backpropagation).
3. **Ενημέρωση Βαρών:**
   `optimizer.step()` – Ο optimizer ενημερώνει τα βάρη του μοντέλου χρησιμοποιώντας τα gradients που υπολογίστηκαν στην προηγούμενη φάση. Αυτός ο βήμας προσπαθεί να ελαχιστοποιήσει το loss.
4. **Μηδενισμός των Gradients:**
   `optimizer.zero_grad()` – Επαναφέρει τα gradients σε 0 για να μην επηρεάσουν τις επόμενες επαναλήψεις.
5. **Καταγραφή Προόδου:**
   `print(f"iter {iter}/{max_iters}, loss: {loss.item()}")` – Κάθε `log_interval` εμφανίζεται η πρόοδος της εκπαίδευσης και το τρέχον loss για να παρακολουθούμε την απόδοση του μοντέλου.

Using ChatGPT to **analyze, evaluate, and refactor** the tutorial was highly effective. Provided:

- Understandable explanations for each stage.
- Hints for cleaner code.
- Analysis of the performance of the transformer and the training loop.

Folder:

The folder contains the main **files that were modified**, the **report,** what **files were used in the request 11 of preparation** as well as the **notebook** of question 7. In addition, we saved our models in most of the queries, so we also list the folder with **the "saved_models".**