UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Deep Learning for NLP

Student name: *Kleopatra Karapanagiotou*
sdi: lt12200010

Course: *Artificial Intelligence II (M138, M226, M262, M325)*
Semester: *Fall Semester 2023*

Contents

1. Abstract

This is a multiclass classification task on Greek tweets, where given a tweet the classifier should classify it into one of our 3 classes: POSITIVE, NEGATIVE, NEUTRAL.

In this experiment series we will experiment with bidirectional stacked RNNs with LSTM/GRU cells trained with Word2Vec embedding input. For neural network model building and hyperparameter optimization PyTorch and Optuna frameworks were used respectively. Data preprocessing techniques, word2vec model training, saving and finetuning, network architectures, optimization search spaces, evaluation metrics and experiment visualizations are presented in the following sections of the report.

2. Data processing and analysis

2.1. Pre-processing

Preprocessing steps applied on the Text column of both the train and validation sets:

- Removal of NaN values
- Pattern removal: RT, @usernames, html, http, https, url links.
- Lowercasing
- Text normalization and replacement of multiple characters with one
- Removal of punctuation, Latin characters, numbers, and words with character length of 1.: this was done in order to remove single characters like 'κ' for 'και', which is a common abbreviation in Greek tweets.
- Removal of emojis
- Uppercasing
- Removal of stopwords excluding specific ones: ΟΧΙ, ΜΗΝ, ΔΕΝ, ΠΟΛΥ, ΚΑΘΟΛΟΥ, ΟΤΙ, since these stopwords when removed they can shift the meaning of the sentence: (e.g ΚΑΚΟΣ vs ΚΑΘΟΛΟΥ ΚΑΚΟΣ) or intensify it like: «ΧΕΙΡΟΤΕΡΟ vs. ΟΤΙ ΧΕΙΡΟΤΕΡΟ. »
- Text tokenization and Stemming with the Greek Stemmer[1], a stemmer generated based on the Paper of Georgios Ntais[2].
- Removal of NaN rows and updating of the whole dataframe , in case any row left empty after the preprocessing applied.
- Sentiment_mapping = {'NEUTRAL': 0, 'POSITIVE': 1, 'NEGATIVE': 2}
- For practical reasons removal of NaN values before and after applying preprocessing was skipped on the test set.

2.1.1. Data preparation with Torchtext

Torchtext is a companion package to PyTorch consisting of data processing utilities and popular datasets for natural language. Torchtext is a convenient library for tokenization, partitioning and batching of our textual data and saves us a lot of personal coding work.

In order to properly prepare my data to be fed in the LSTM/GRU models:

- I first applied pre-and post-processing steps in both train and validation data,

[1] https://gist.github.com/Patelis-GM/e1f8cf553f27ff40ed49db8c310611b3

[2] https://people.dsv.su.se/~hercules/papers/Ntais_greek_stemmer_thesis_final.pdf

extracted only the necessary columns ('Text' and 'Sentiment') and saved them into two new .csv files (train.csv, validation.csv)

- I used the **torchtext.data.Field** class to create the necessary fields for the TEXT and the LABEL. According to the Pytorch official documentation, the Field defines common datatypes that maintain a vocab object, which defines a set of possible values for elements of the field and their corresponding numerical representations (tensors). It also includes various parameters that can be set to represent the datatype in different ways. For the TEXT field, I set the sequential parameter to True to apply tokenization and tokenized the strings using the SpaCy tokenizer for the Greek language and especially I use the small model ('el_core_news_sm'). I also kept the uppercasing of the examples and selected to get a list containing their lengths and a tuple of padded minibatches with the include_lengths set to True parameter. For the LABEL field, since this is a multiclass classification, I did not need to set the dtype in the LABEL field. PyTorch expects the labels to be numericalized LongTensors, which is the default here.
- I used the **data.TabularDataset.splits** function to load the new cleaned and preprocessed dataframes (train,validation) in csv form and create the dataset with the fields.
- Then I built the vocabulary to index all the tokens. For the indexing, I finetuned Word2vec model on my train data, by using the same hyperparameters with which the w2v model was trained in Assignment_2 (Skipgram model, vector size 200, window size 4, min count 1, alpha 0.01, negative sampling, num samples 10, workers 3, subsampling 0.01, shrink_windows True, epochs 20). By default, any words that appear in the vocabulary, but do not appear in the chosen embeddings get initialized to a random vector from a normal distribution (with mean 0 and std 1). The maximum vocabulary size was the vocabulary size of the train data. More details about the workaround in training and applying the word2vec model in the embedding layers with Torchtext can be found in this medium article[3], from which I got inspired.

2.2. Analysis

Since the dataset Analysis was done exhaustively on the first assignment to end up on the above preprocessing steps, I skip the Analysis part. Since the only Preprocessing step, I added on this experiment series is the Stemming of tokenized words, to create word2vec embeddings on stemmed words of the train and the validation set. I t would be interesting to report in this section a sample content of the torchtext vocabulary and labels:

[3] https://rohit-agrawal.medium.com/using-fine-tuned-gensim-word2vec-embeddings-with-torchtext-and-pytorch-17eea2883cd

```
#Look at most common words:
print(TEXT.vocab.freqs.most_common(20))
#Tokens corresponding to the first 10 indices (0, 1, ..., 9):
print(TEXT.vocab.itos[:10]) # itos = integer-to-string
#Converting a string to an integer:
print(TEXT.vocab.stoi['ΜΗΤΣΟΤΑΚ']) # stoi = string-to-integer
#Class labels:
print(LABEL.vocab.stoi)
#Class label count:
LABEL.vocab.freqs
```

```
[('ΤΣΙΠΡ', 11141), ('ΜΗΤΣΟΤΑΚ', 9238), ('ΔΕΝ', 9157), ('ΝΔ', 8157), ('ΣΥΡΙΖ', 7571), ('ΕΚΛΟΓ', 48
20), ('ΟΤΙ', 4342), ('ΚΚΕ', 3515), ('ΚΥΡΙΑΚ', 2414), ('ΠΑΣΟΚ', 2280), ('ΣΚΑ', 2273), ('ΚΑΝ', 224
6), ('ΕΚΛΟΓΕΣ2019', 2156), ('ΚΙΝΑΛ', 2080), ('ΑΛΕΞ', 1620), ('ΨΗΦΙΣ', 1595), ('ΘΕΛ', 1549), ('ΕΛΛ
ΑΔ', 1529), ('ΜΕΓΑΛ', 1486), ('ΧΡΟΝ', 1485)]
['<unk>', '<pad>', '<sos>', '<eos>', 'ΤΣΙΠΡ', 'ΜΗΤΣΟΤΑΚ', 'ΔΕΝ', 'ΝΔ', 'ΣΥΡΙΖ', 'ΕΚΛΟΓ']
5
defaultdict(None, {'1': 0, '2': 1, '0': 2})


Counter({'1': 12207, '2': 12205, '0': 12201})
```

The label encoding has been changed during preprocessing and is maybe confusing with the index :
"1"[POSITIVE]: index 0
"2"[NEGATIVE]: index 1
"0"[NEUTRAL]: index 2

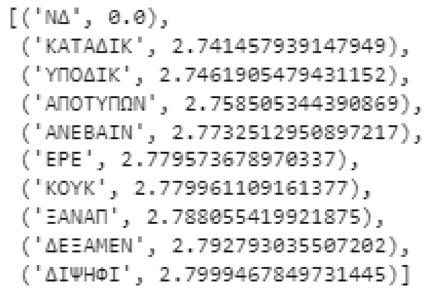
2.3. Data partitioning for train, test and validation

In our experiment setting our data are already partitioned on 3 .csv files, train, validation, and test. No further partitioning was performed. Training data were used for the training of the Network, validation data were used as the evaluation part of the network and the plotting of evaluation metric results and learning curves. And finally test data were used to predict the polarity of each tweet by creating a new column named: Predicted and adding the predictions of our trained classifier.
For train and validation data I defined dataloaders (iterators) with the **PytorchText BucketIterator** functionality. We iterate over these in the training/evaluation loop, and they return a batch of examples (indexed and converted into tensors) at each iteration. In the iterators, each index represents a token, and each column represents a sentence. The number of columns is equal to the number of the batch size (256). Also, the data are sorted in each batch according to the lambda function, which ensures that the batches are of the same length, which helps minimizing the amount of padding per example. Having batches with similar length examples provides a lot of gain for recurrent models (RNN, GRU, LSTM), because it allows us to provide the most optimal batches when training models with text data[4]. The examples are also shuffled in each epoch run during training.

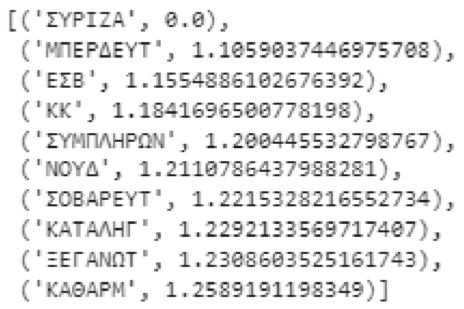
2.4. Vectorization

As mentioned previously, for embedding input in my models I trained a Word2vec model o n the train data only, saved and loaded it as pre-trained model and initialized the word em beddings for the Text field with these pre-trained vectors, through the build_vocab functio n. TorchText handles downloading the vectors and associating them with the correct word s in the vocabulary. The vocabulary size is 24032 tokens (including the <unk>,<pad>, <sos > and <eos>). To get an overview of the word2vec embedding representations, here are th e 10 most similar words for each party:

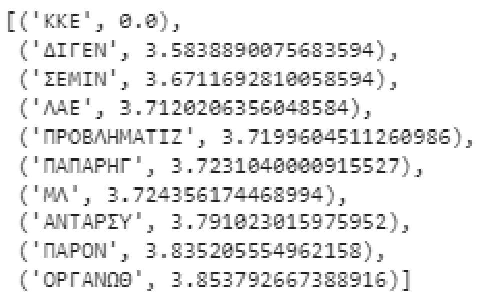
[4] https://gmihaila.medium.com/better-batches-with-pytorchtext-bucketiterator-12804a545e2a



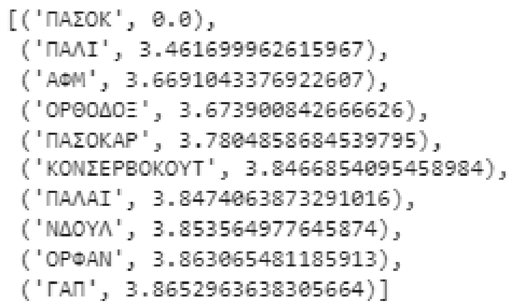
```
[('ΝΔ', 0.0),
 ('ΚΑΤΑΔΙΚ', 2.741457939147949),
 ('ΥΠΟΔΙΚ', 2.7461905479431152),
 ('ΑΠΟΤΥΠΩΝ', 2.758505344390869),
 ('ΑΝΕΒΑΙΝ', 2.7732512950897217),
 ('ΕΡΕ', 2.779573678970337),
 ('ΚΟΥΚ', 2.779961109161377),
 ('ΞΑΝΑΠ', 2.788055419921875),
 ('ΔΕΞΑΜΕΝ', 2.792793035507202),
 ('ΔΙΨΗΦΙ', 2.7999467849731445)]
```



```
[('ΣΥΡΙΖΑ', 0.0),
 ('ΜΠΕΡΔΕΥΤ', 1.1059037446975708),
 ('ΕΣΒ', 1.1554886102676392),
 ('ΚΚ', 1.1841696500778198),
 ('ΣΥΜΠΛΗΡΩΝ', 1.200445532798767),
 ('ΝΟΥΔ', 1.2110786437988281),
 ('ΣΟΒΑΡΕΥΤ', 1.2215328216552734),
 ('ΚΑΤΑΛΗΓ', 1.2292133569717407),
 ('ΞΕΓΑΝΩΤ', 1.2308603525161743),
 ('ΚΑΘΑΡΜ', 1.2589191198349)]
```

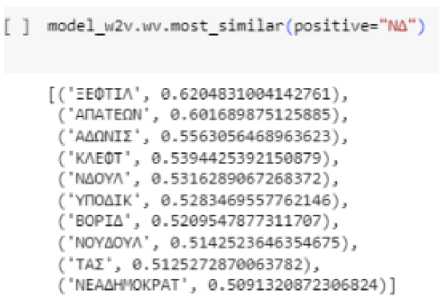


```
[('ΚΚΕ', 0.0),
 ('ΔΙΓΕΝ', 3.5838890075683594),
 ('ΣΕΜΙΝ', 3.6711692810058594),
 ('ΛΑΕ', 3.7120206356048584),
 ('ΠΡΟΒΛΗΜΑΤΙΖ', 3.7199604511260986),
 ('ΠΑΠΑΡΗΓ', 3.7231040000915527),
 ('ΜΛ', 3.724356174468994),
 ('ΑΝΤΑΡΣΥ', 3.791023015975952),
 ('ΠΑΡΟΝ', 3.835205554962158),
 ('ΟΡΓΑΝΩΘ', 3.853792667388916)]
```

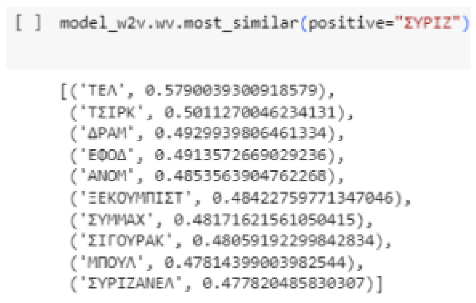


```
[('ΠΑΣΟΚ', 0.0),
 ('ΠΑΛΙ', 3.461699962615967),
 ('ΑΦΜ', 3.6691043376922607),
 ('ΟΡΘΟΔΟΞ', 3.673900842666626),
 ('ΠΑΣΟΚΑΡ', 3.7804858684539795),
 ('ΚΟΝΣΕΡΒΟΚΟΥΤ', 3.8466854095458984),
 ('ΠΑΛΑΙ', 3.8474063873291016),
 ('ΝΔΟΥΛ', 3.853564977645874),
 ('ΟΡΦΑΝ', 3.863065481185913),
 ('ΓΑΠ', 3.8652963638305664)]
```

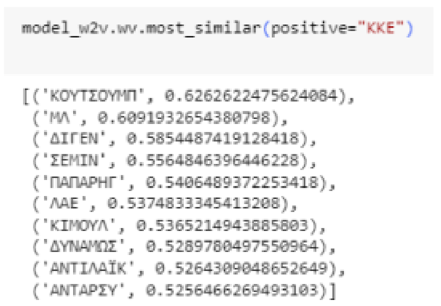
Comparing these with the Word2vec representations from previous assignment_2, we do observe some small differences, but this can be explained do to the fact that word2vec of the previous assignment was trained on more content (both train and validation data), which probably helped to capture similarities with more informative and emotionally connotated words (ΞΕΦΤΙΛΕΣ, ΤΣΙΡΚΟ, ΤΕΛΟΣ κλπ) and named entities (ΑΔΩΝΙΣ, ΚΟΥΤΣΟΥΜΠΑΣ κλπ)



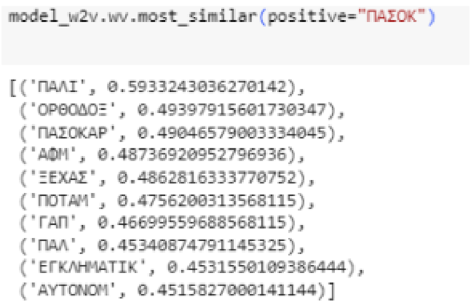
```
[ ] model_w2v.wv.most_similar(positive="ΝΔ")

    [('ΞΕΦΤΙΛ', 0.6204831004142761),
     ('ΑΠΑΤΕΩΝ', 0.601689875125885),
     ('ΑΔΩΝΙΣ', 0.5563056468963623),
     ('ΚΛΕΦΤ', 0.5394425392150879),
     ('ΝΔΟΥΛ', 0.5316289067268372),
     ('ΥΠΟΔΙΚ', 0.5283469557762146),
     ('ΒΟΡΙΔ', 0.5209547877311707),
     ('ΝΟΥΔΟΥΛ', 0.5142523646354675),
     ('ΤΑΣ', 0.5125272870063782),
     ('ΝΕΑΔΗΜΟΚΡΑΤ', 0.5091320872306824)]
```



```
[ ] model_w2v.wv.most_similar(positive="ΣΥΡΙΖ")

    [('ΤΕΛ', 0.5790039300918579),
     ('ΤΣΙΡΚ', 0.5011270046234131),
     ('ΔΡΑΜ', 0.4929939806461334),
     ('ΕΦΟΔ', 0.4913572669029236),
     ('ΑΝΟΜ', 0.4853563904762268),
     ('ΞΕΚΟΥΜΠΙΣΤ', 0.48422759771347046),
     ('ΣΥΜΜΑΧ', 0.48171621561050415),
     ('ΣΙΓΟΥΡΑΚ', 0.48059192299842834),
     ('ΜΠΟΥΛ', 0.47814399003982544),
     ('ΣΥΡΙΖΑΝΕΛ', 0.477820485830307)]
```



```
model_w2v.wv.most_similar(positive="ΚΚΕ")

[('ΚΟΥΤΣΟΥΜΠ', 0.6262622475624084),
 ('ΜΛ', 0.6091932654380798),
 ('ΔΙΓΕΝ', 0.5854487419128418),
 ('ΣΕΜΙΝ', 0.5564846396446228),
 ('ΠΑΠΑΡΗΓ', 0.5406489372253418),
 ('ΛΑΕ', 0.5374833345413208),
 ('ΚΙΜΟΥΛ', 0.5365214943885803),
 ('ΔΥΝΑΜΩΣ', 0.5289780497550964),
 ('ΑΝΤΙΛΑΪΚ', 0.5264309048652649),
 ('ΑΝΤΑΡΣΥ', 0.5256466269493103)]
```



```
model_w2v.wv.most_similar(positive="ΠΑΣΟΚ")

[('ΠΑΛΙ', 0.5933243036270142),
 ('ΟΡΘΟΔΟΞ', 0.49397915601730347),
 ('ΠΑΣΟΚΑΡ', 0.49046579003334045),
 ('ΑΦΜ', 0.48736920952796936),
 ('ΞΕΧΑΣ', 0.4862816333770752),
 ('ΠΟΤΑΜ', 0.4756200313568115),
 ('ΓΑΠ', 0.46699559688568115),
 ('ΠΑΛ', 0.45340874791145325),
 ('ΕΓΚΛΗΜΑΤΙΚ', 0.4531550109386444),
 ('ΑΥΤΟΝΟΜ', 0.4515827000141144)]
```

3. Algorithms and Experiments

3.1. Experiments

My experiment approach for this assignment was to define my RNN architectures and experiment with hyperparameter tuning with Optuna right from the start before making any brute force training. The chosen hyperparameters from Optuna best trial were used to train and evaluate two Bi_RNN architectures, one without the attention mechanism and one including the attention mechanism and report of the performance of this mechanism during training. Detailed report on the hyperparameter tuning search spaces, will be given in the 'Hyperparameter tuning' section.

Classes that build the models:

1. **BIDIRECTIONAL_RNN_LSTM_GRU(nn.Module)**, which constructs a Bidirectional Recurrent Neural Network classifier with either LSTM or GRU cells based on selection. Bidirectional RNNs process the sequence in both directions. In fact, two distinct RNNs are used: one for the forward and one for the reverse direction. As a result, a hidden state is the output of each RNN, which are concatenated to form a single hidden state. Both RNN models have the same structure, with the only difference being the cell type (GRU/LSTM) and the initialization of the hidden state. The hidden state for the LSTM is a tuple containing both the cell state and the hidden state, whereas the GRU only has a single hidden state. To initialize such a model, the following parameters were given: the cell type, the size of the vocabulary, the number of dimensions representing each word (200), the hidden size, the number of classes, the embedding vectors, the numbers of layers, the dropout, and the pad token index from the vocabulary. It applies two dropout layers and the RELU activation function. The **forward function** defines the forward pass of the inputs. The tweets are passed through the embedding layer added with a dropout layer to get the embeddings, then they are packed to process the non-padded elements. In this way, the computation is faster. The packed embeddings pass through the LSTM or GRU cells to learn from both directions. The packed output of the cells is unpacked to a tensor. Then, the final forward layer and the backward hidden layers are concatenated and passed through a dropout layer. Finally, the single hidden state is passed through the fully connected linear layer and then the RELU activation function to get the probability of the sequences.
2. **BIDIRECTIONAL_RNN_LSTM_GRU_ATTENTION(nn.Module):** This class is similar to the previous one, but includes attention. I tried to implement the Bahdanau (additive) Attention mechanism, for which I got inspired from the following notebook on github: Self-Supervised Euphemism Detection and Identification for Content Moderation5. The attention (self, output, final_state) function computes the weights for each sequence in the RNN's output and then computes the hidden state. The new hidden state from the attention is passed through the fully connected linear layer and then the RELU activation function to get the probability of the sequences.

Functions for model training, evaluation and testing:

- **train(model, iterator, optimizer, criterion, clip) function:** takes as parameters a defined model, an iterator, a defined optimizer and loss function and a number for gradient clipping. It performs the training phase. More particularly, it computes the predictions using the tuple of the Text, which includes tensors and the lengths of the tweets. It also uses gradient clipping to avoid exploding gradients, which is a common problem in recurrent neural networks, especially LSTM. By "exploding gradients" it means numerical overflow or underflow due to large updates to weights during training[6]. In this function norm clipping is used, which involves scaling the whole gradient if the L2 norm of the gradient vector exceeds a certain

[5] https://github.com/WanzhengZhu/Euphemism/blob/master/classification_model.py

[6] How to Avoid Exploding Gradients With Gradient Clipping - MachineLearningMastery.com

threshold. This method preserves the direction of the gradient and is generally the preferred method for gradient clipping. Finally, the function calculates and prints the loss as well as the accuracy of each epoch using the predictions and the true labels.

- **evaluate(model, iterator, criterion) function:** takes as parameters a defined model, an iterator, and a defined loss function. It performs the evaluation phase. More particularly, it computes and prints the loss and the accuracy of each epoch using the predictions and the true labels.
- **testing(model, iterator) function:** takes as parameters a defined model and an iterator. It makes predictions and returns a list of the predicted labels and the true labels.

Functions used for metric calculation:

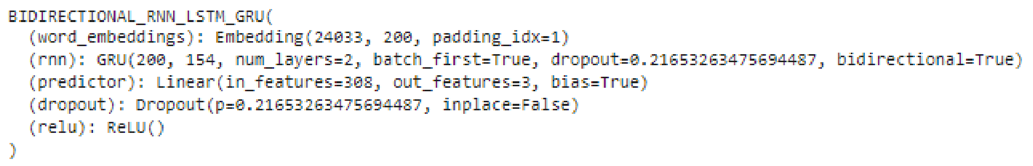
- **accuracy(preds, y_true) functon**: takes as parameters the predictions of labels of the model and the true labels. It changes the predictions into one dimensional tensors and calculates the accuracy. This function is specifically used in the train and evaluate functions to calculate and print the accuracy of each epoch.
- **calculate_metrics(y_true, preds) function**: takes as parameters the y true values and the predicted values. It calculates all the metrics asked for this assignment, namely the precision, F1 score and recall. It also returns the estimated the accuracy of the models and prints the classification matrix as a general summary of all the metrics.
- **plot_roc_auc(y_true, y_score, n_classes) function:** Takes as parameters the true class labels (y_true), predicted class probabilities (y_score), and the number of classes (n_classes). Computes the ROC curves and the Area Under the Curve (AUC) for each class, along with the micro-average and macro-average ROC curves and AUC. Plots the ROC curves for all classes, along with the micro-average and macro-average ROC curves.

Helper functions used:

- **count_parameters(model) function:** takes as parameter a defined machine learning model and returns the summation of its parameters.
- **epoch_time(start_time, end_time) function:** calculates the time in seconds during training and evaluation in each epoch
- **set_seed(seed_value) function:** sets the seed for reproducibility of the same results.

Overall, the experiments can be summarized in the following:

Network Architecture

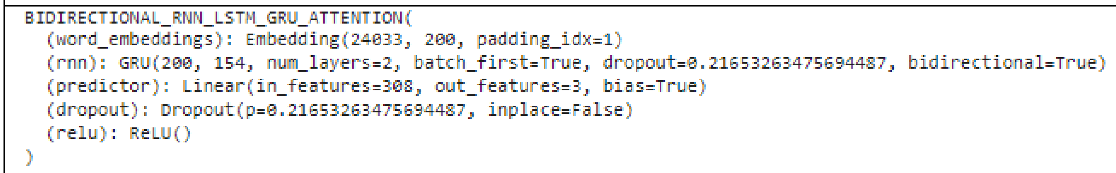


```
BIDIRECTIONAL_RNN_LSTM_GRU(
  (word_embeddings): Embedding(24033, 200, padding_idx=1)
  (rnn): GRU(200, 154, num_layers=2, batch_first=True, dropout=0.21653263475694487, bidirectional=True)
  (predictor): Linear(in_features=308, out_features=3, bias=True)
  (dropout): Dropout(p=0.21653263475694487, inplace=False)
  (relu): ReLU()
)
```

embedding_dim:200
batch_size: 256
lr: 0.0001
weight_decay: 1e-4
gradient clipping threshold :1.8
optimizer: Adam
loss: cross-entropy

Experiment_1: 40 epochs
Experiment_2: 10 epochs
Experiment_3: 20 epochs

Network Architecture

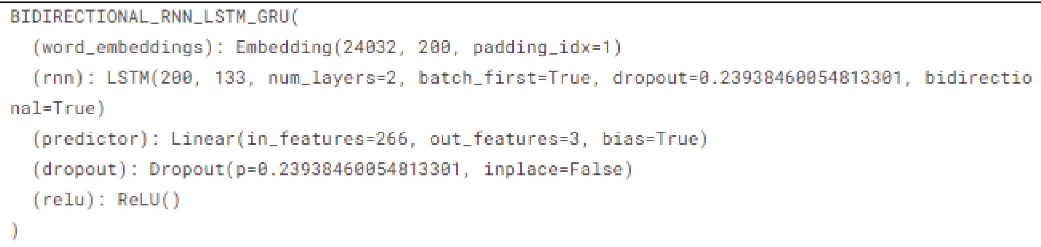


```
BIDIRECTIONAL_RNN_LSTM_GRU_ATTENTION(
  (word_embeddings): Embedding(24033, 200, padding_idx=1)
  (rnn): GRU(200, 154, num_layers=2, batch_first=True, dropout=0.21653263475694487, bidirectional=True)
  (predictor): Linear(in_features=308, out_features=3, bias=True)
  (dropout): Dropout(p=0.21653263475694487, inplace=False)
  (relu): ReLU()
)
```

embedding_dim:200
batch_size: 256
lr: 0.0001
weight_decay: 1e-4
gradient clipping threshold :1.8
optimizer: Adam
loss: cross-entropy

Experiment_4: 40 epochs
Experiment_5: 10 epochs

Network Architecture



```
BIDIRECTIONAL_RNN_LSTM_GRU(
  (word_embeddings): Embedding(24032, 200, padding_idx=1)
  (rnn): LSTM(200, 133, num_layers=2, batch_first=True, dropout=0.23938460054813301, bidirectio
nal=True)
  (predictor): Linear(in_features=266, out_features=3, bias=True)
  (dropout): Dropout(p=0.23938460054813301, inplace=False)
  (relu): ReLU()
)
```

embedding_dim:200
batch_size: 256
lr: 0.0001
weight_decay: 1e-4
gradient clipping threshold :5
optimizer: Adam
loss: cross-entropy

Experiment_6: 10 epochs

Network Architecture

```
BIDIRECTIONAL_RNN_LSTM_GRU_ATTENTION(
  (word_embeddings): Embedding(24032, 200, padding_idx=1)
  (rnn): LSTM(200, 133, num_layers=2, batch_first=True, dropout=0.23938460054813301, bidirectic
nal=True)
  (predictor): Linear(in_features=266, out_features=3, bias=True)
  (dropout): Dropout(p=0.23938460054813301, inplace=False)
  (relu): ReLU()
)
```

embedding_dim:200
batch_size: 256
lr: 0.0001
weight_decay: 1e-4
gradient clipping threshold :5
optimizer: Adam
loss: cross-entropy

Experiment_7: 10 epochs

3.1.1. Table of trials

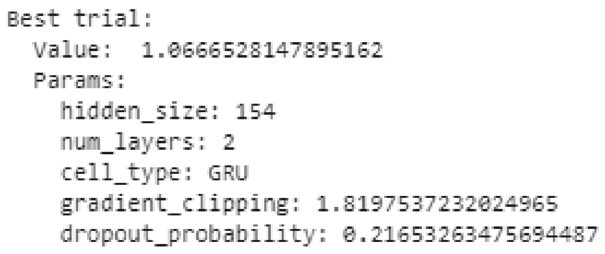
Trial	**Accuracy**	**Weighted avg Precision**	**Weighted avg Recall**	**Weighted avg F1**
Experiment_1	0.38	0.38	0.38	0.38
Experiment_2	0.40	0.42	0.40	0.38
Experiment_3	0.40	0.40	0.40	0.39
Experiment_4	0.38	0.38	0.38	0.38
Experiment_5	0.40	0.41	0.40	0.39
Experiment_6	0.40	0.42	0.40	0.39
Experiment_7 (Submission)	**0.41**	**0.42**	**0.41**	**0.39**

3.2. Hyper-parameter tuning

Following hyperparameters were tuned based on suggested trials of the Optuna hyperparameter optimization framework[7] : number of stacked RNNs, hidden size of embedding representation, cell type, dropout ratio and the clip value for gradient clipping to avoid exploding gradients. The experimentation included also some additional finetuning on the number of epochs, since the first experiment run (40 epochs) showcased significant overfitting and adjustments in epochs were necessary.

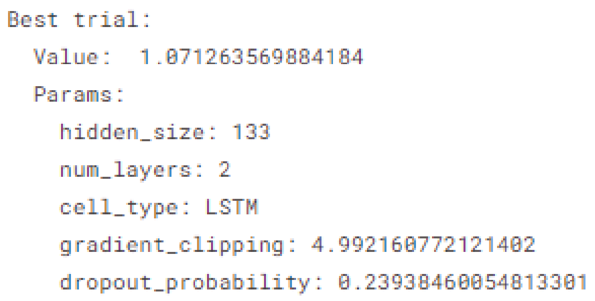
3.2.1. Defining Optuna search space and picking the chosen parameters as per Optuna best trial:

- Number of hidden layers (stacked RNNs): 1-3
- lstm/gru hidden embedding representation(hidden_size): 64-256
- Type of cells : LSTM | GRU
- Dropout probability : 0.2-0.7
- gradient: 1-5
- epochs: 1 (to speed up computation)
- Batch size: 256
- N_trials: 100
- Direction: minimize loss



```
Best trial:
  Value:  1.0666528147895162
  Params:
    hidden_size: 154
    num_layers: 2
    cell_type: GRU
    gradient_clipping: 1.8197537232024965
    dropout_probability: 0.21653263475694487
```

Best trial during experiments 1-5



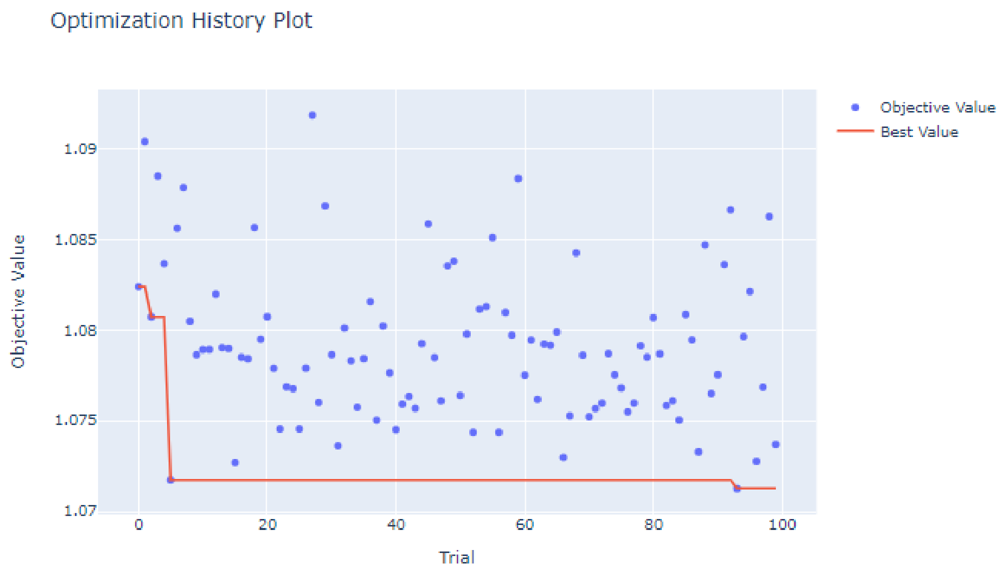
```
Best trial:
  Value:  1.071263569884184
  Params:
    hidden_size: 133
    num_layers: 2
    cell_type: LSTM
    gradient_clipping: 4.992160772121402
    dropout_probability: 0.23938460054813301
```

Best trial in Kaggle submission (experiments 6-7)

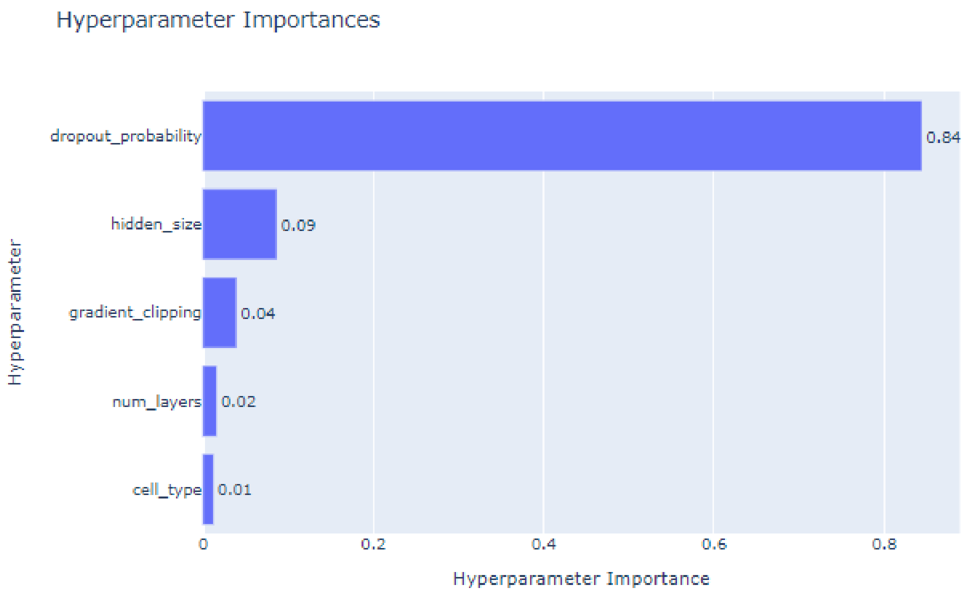
[7] Optuna - A hyperparameter optimization framework

3.3. Optuna visualizations (explainability)

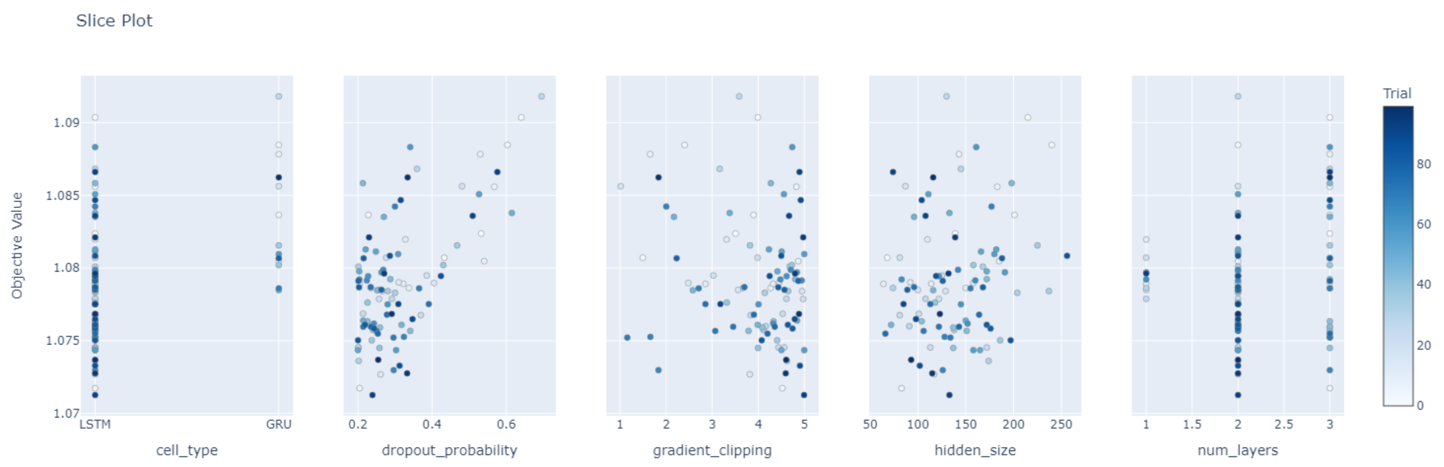
The visualization module of Optuna is a handful way to explain the optimization process during the study. The plots below visualize the optimization process for the Optuna run in the submission notebook (experiments 6-7). We observe that:



1) 100 trials were not enough to capture a good number of best values. We observe though, that when approaching to 100 trials we get a decrease in the validation loss, which is the objective value on the y-axis. We could increase the number of trials to check for lower values.



2) The dropout rate affects the model performance the most, whereas all the rest parameters (hidden state, clip value, number of stacked RNNs and the type of cell) seem to have a slight impact on performance.



The darkness of the dots corresponds to the number of the trial that produced them, the darker areas correspond to later trials.

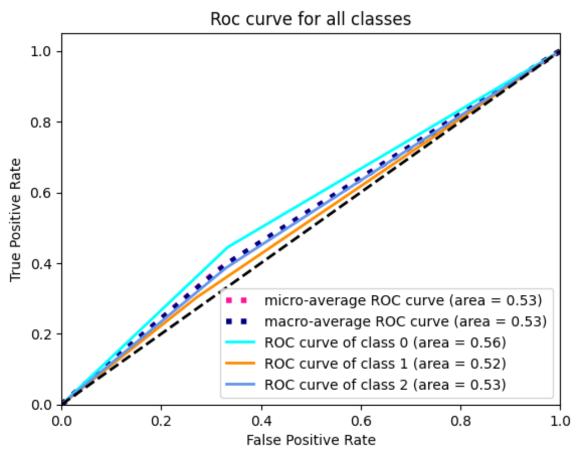
3) Although we know from the HP importance table that the cell type and the number of stacked RNNs didn't have a major effect on minimizing model's loss, from looking at the Slice Plot :
   a. GRU reaches a minimum loss of around 1.078 in the trials, whereas LSTM goes until 1.071
   b. Most trials are concentrated around the LSTM cell type.

   Since LSTM seems more capable of reaching lower validation loss values, we can continue only with this cell type and omit GRUs. Also, we know that LSTM cell is a good default for long distance dependencies due to its memory cell, which may be helpful for capturing the meaning in longer tweet text.

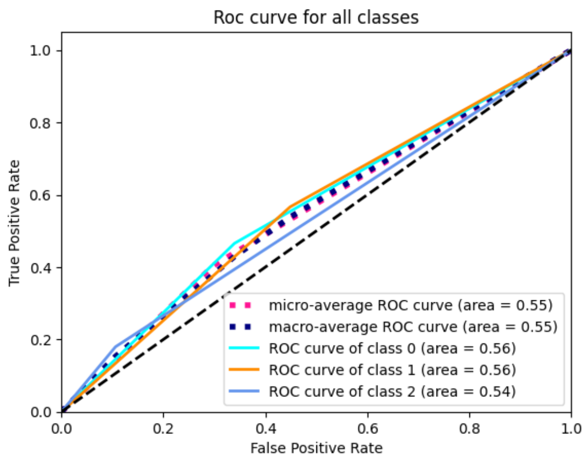
4) From 0.2 to 0.25 is a good range for dropout ratios, since in this range we see lower objective values. We do not see though a lot of trials concentrated around the lowest losses. The value 0.23 is a good ratio to keep.
5) Also, the clip value and the hidden size value do not show a consistent behavior around a specific validation loss value. We see that the values 5 (for gradient clipping threshold) and 133 (for hidden size) managed to approach the lowest validation loss value compared to the rest.
6) Finally concerning the number of RNN layers we see a consistent concentration of trials around the 2 layers. Meaning we can keep this value in our final model, as optimal choice.

3.4. Evaluation

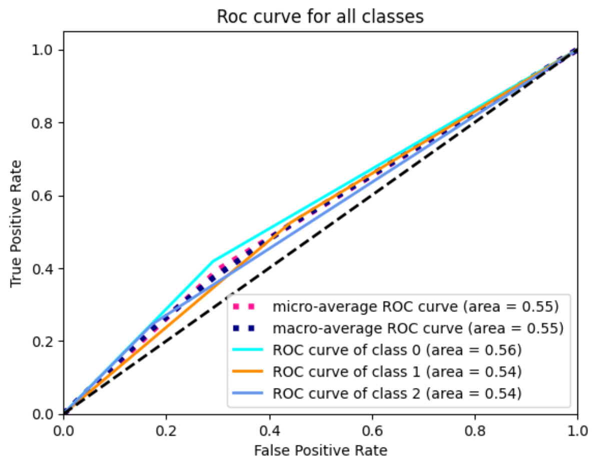
3.4.1. ROC curves.



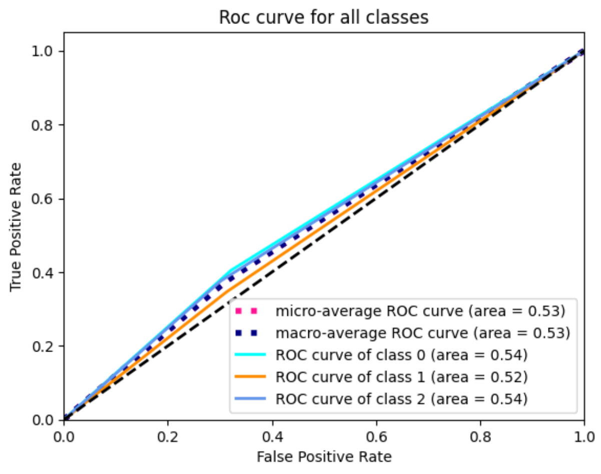
Experiment_1



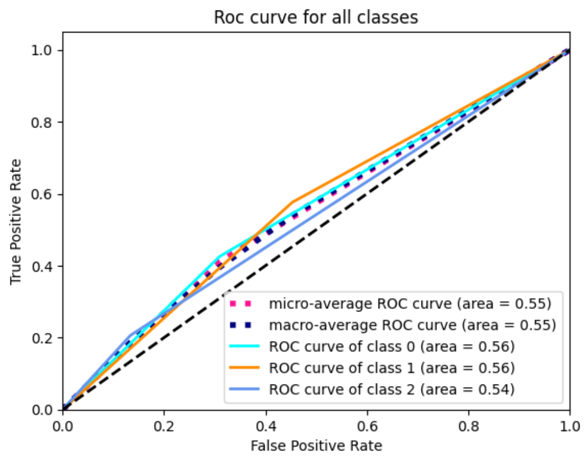
Experiment_2



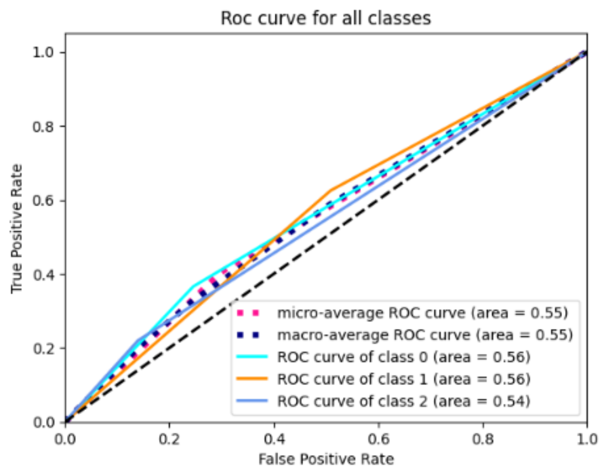
Experiment_3



Experiment_4



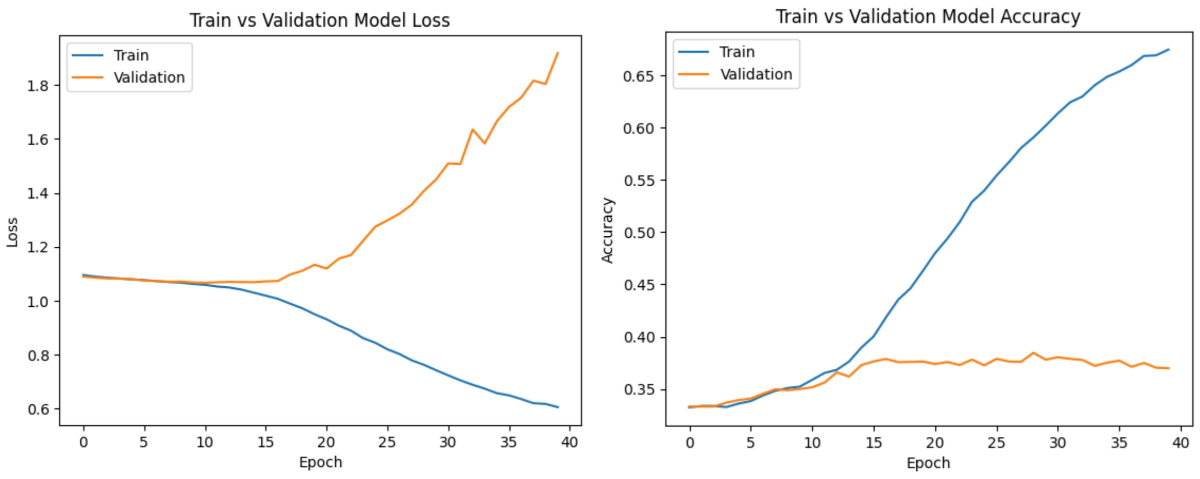
Experiment_5



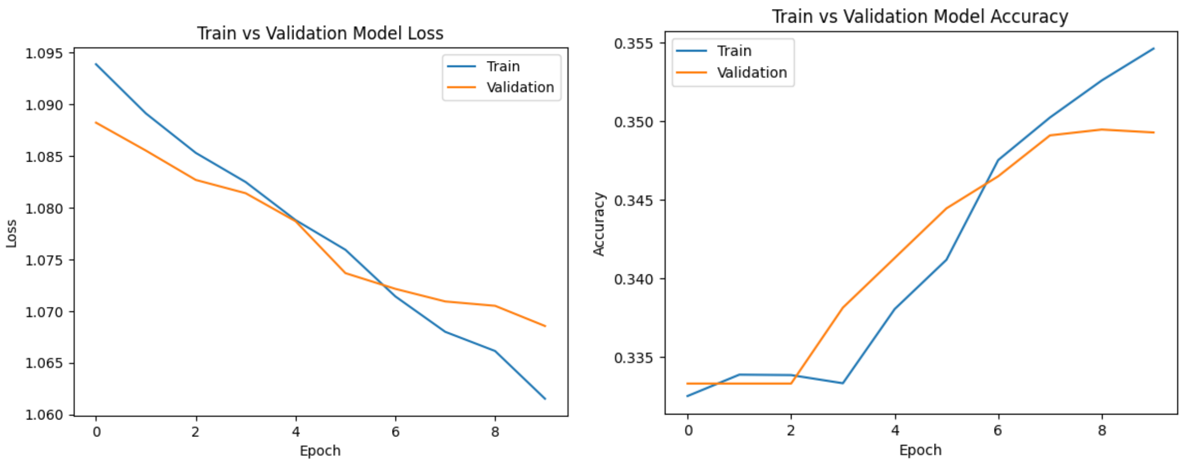
Experiment_6

Overall in all experiments we report an AUC of about 0.5, indicating that the models' performances are comparable to that of random chance. Especially biGRUs trained for 40 epochs indicated the lowest AUC (0.53), which may be due to the significant overfitting.

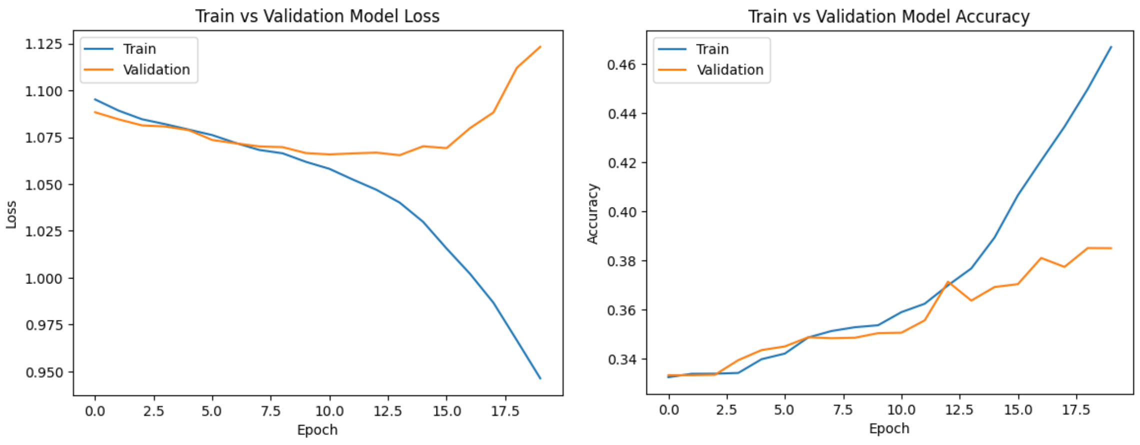
3.4.2. Learning Curves.



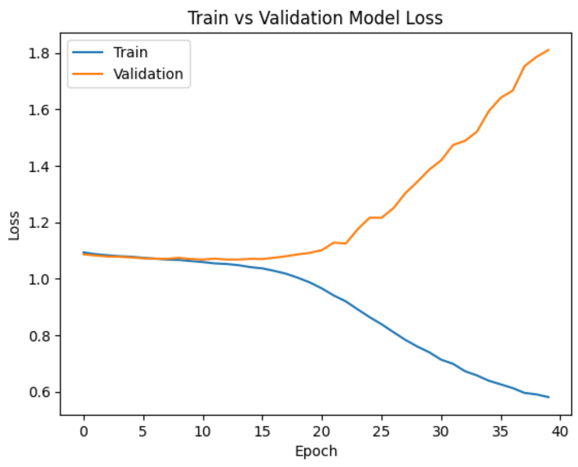
Experiment_1 (40_epochs)

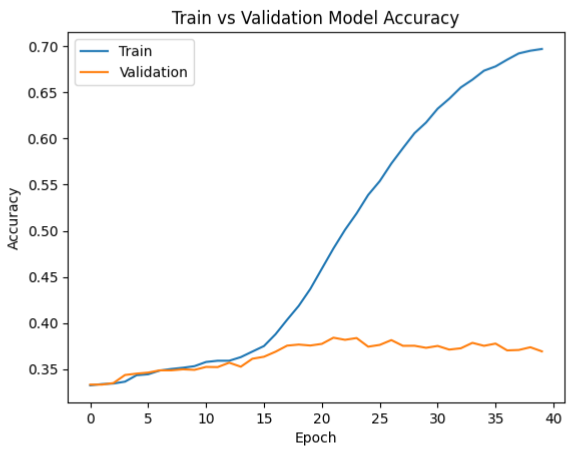


Experiment_2 (10_epochs)

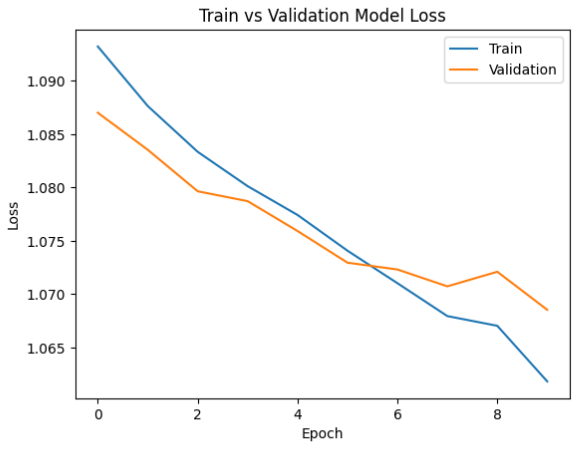


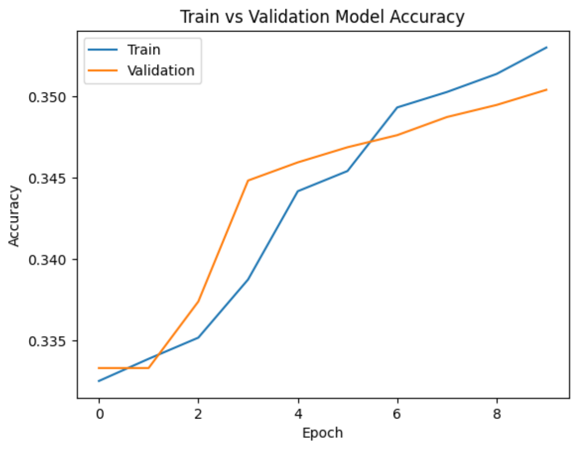
Experiment_3 (20_epochs)



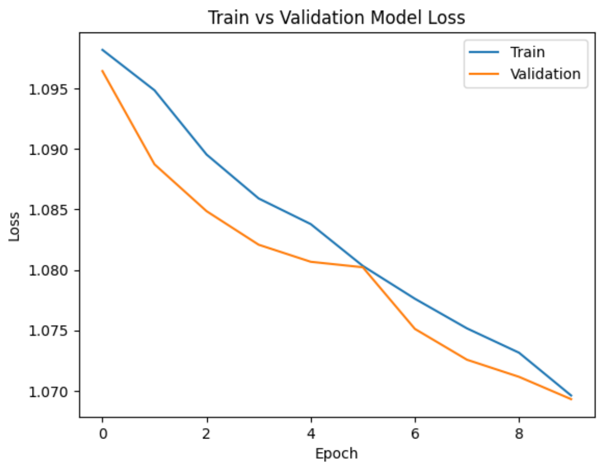


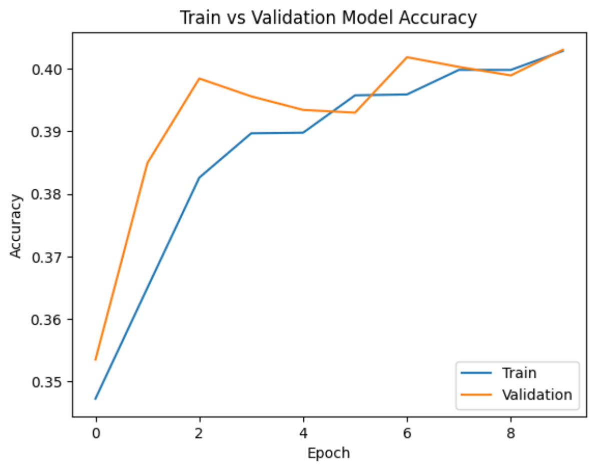
Experiment_4 (40 epochs)





Experiment_5 (10_epochs)





Experiment_6 (10_epochs)

From all the above experiments we see a significant effect of the epochs in during training. After the 10th epoch, the validation loss starts increasing with the training loss decreasing, whereas the validation accuracy stabilizes with the training accuracy steeply increasing. These tendencies in training create big gaps in the learning curves for the models in 20 and 40 epochs.
Concerning the 10_epoch models we see that experiment_6 indicating a higher validation accuracy and lower validation loss, compared to the previous experiments.

4. Results and Overall Analysis

4.1. Results Analysis

Considering the bidirectional nature of the models, and the quality of the word2vec embeddings, I was expecting the model to improve, it seemed though that neither the model architecture nor the addition of the additive attention mechanism in the biGRU and biLSTM contributed substantially to the performance. Furthermore, the choice on cell type did not seem to affect the model's performance. One important factor that may be still preventing the model to learn is the static word2vec embedding representations. Contextual word embeddings, like BERT embeddings, since they capture meaning from the surrounding words, will be a better alternative to explore as input embedding layer in the above networks.[8]

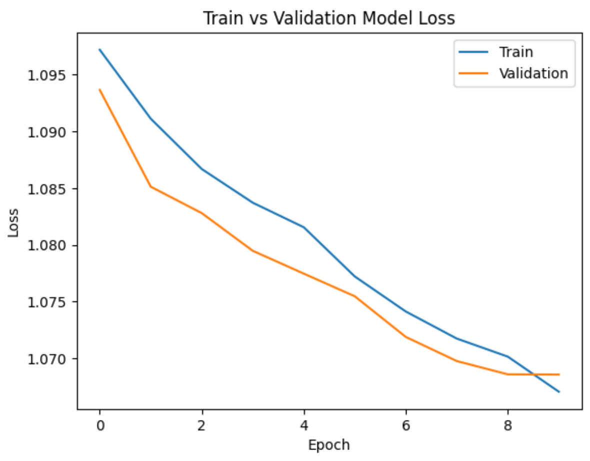
Some steps that could be taken for further experimentation:

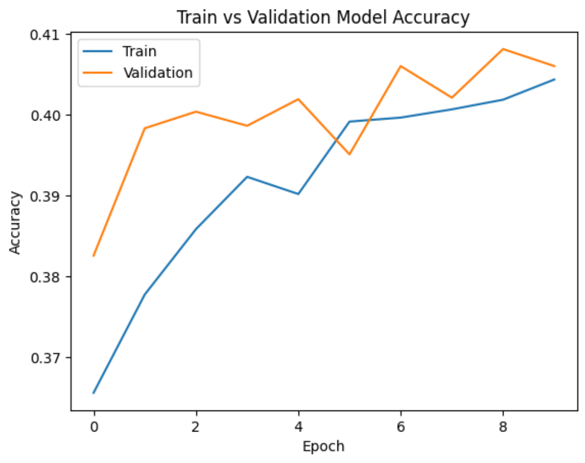
- Adjust the preprocessing steps on the text (e.g. keep the punctuation characters) to see if the model gives attention weight to punctuation characters.
- Add skip connections in the multi-layer RNN
- Experiment with dot product (Luong)[9] attention since it is considered faster and more space-efficient in practice.
- Since dropout was an important hyperparameter for our models, we can experiment with other regularization techniques like batch normalization and layer normalization.
- Freeze embeddings and let the network learn from the pre-trained representations produced by word2vec.

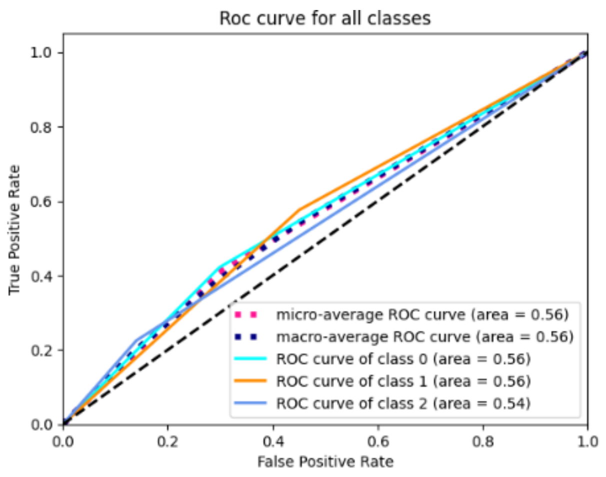
[8] https://github.com/bentrevett/pytorch-sentiment-analysis/blob/main/legacy/6%20-%20Transformers%20for%20Sentiment%20Analysis.ipynb

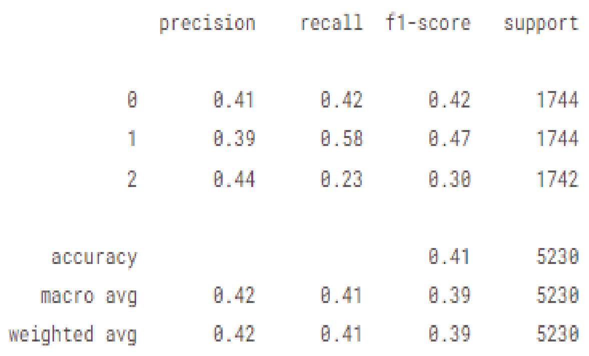
[9] https://paperswithcode.com/paper/effective-approaches-to-attention-based

4.1.1. Best trial.







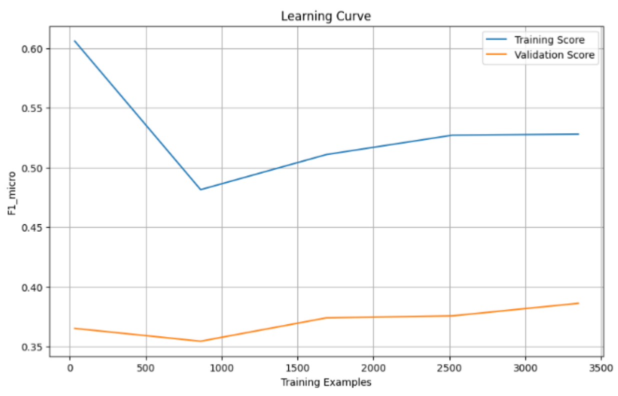


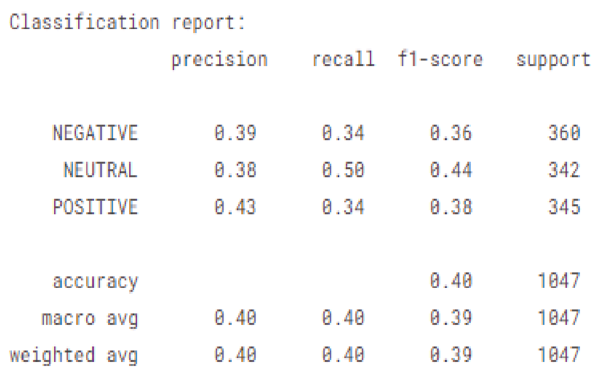
```
              precision    recall  f1-score   support

           0       0.41      0.42      0.42      1744
           1       0.39      0.58      0.47      1744
           2       0.44      0.23      0.30      1742

    accuracy                           0.41      5230
   macro avg       0.42      0.41      0.39      5230
weighted avg       0.42      0.41      0.39      5230
```

The best trial is an LSTM with Attention, the hyperparameters of which are found in Experiment_7 above. We see a slight contribution of 0.1 of the attention mechanism in the classifier's performance, yet it is not enough to claim that the attention was a crucial addition to the model. Considering the previous experiments and the plots of the best trial, we could increase the epochs from 10 to 13 to see if we reach a more optimal validation loss.

4.2. Comparison with the first project

Comparing the best model results with those of the best model on the first project (TfIdf unigrams) max_features 1000 +3 numeric, L1 regularization, saga, C: 1.0, trained with Logistic Regression), we observe a slight improvement in this project. It is worth noting that on the first project a different data partitioning approach was followed. Both train and validation data were splitted into train/test and dev/test with a ratio of 80/20. This is why on the classification report below we see less data (1047). Considering learning curves in project_1 we plotted learning curves of F1 score on train and validation data. It was observed that all models were overfitting on the train data, and this was accounted due to the use of a linear model which is probably not suitable for our data since they are not linearly separable.



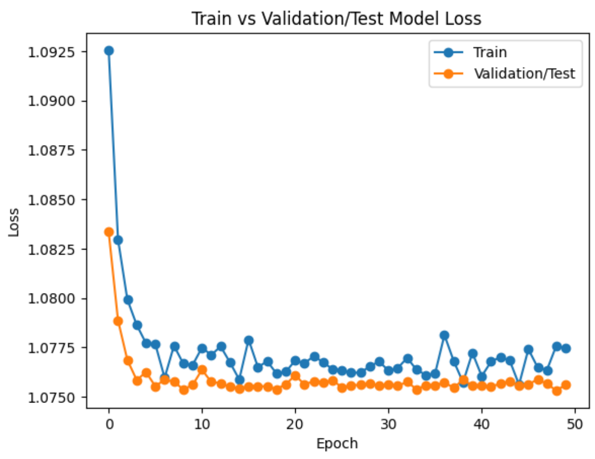


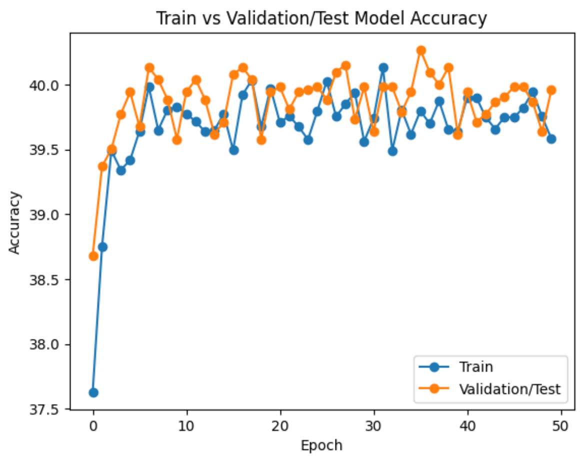
```
Classification report:
              precision    recall  f1-score   support

    NEGATIVE       0.39      0.34      0.36       360
     NEUTRAL       0.38      0.50      0.44       342
    POSITIVE       0.43      0.34      0.38       345

    accuracy                           0.40      1047
   macro avg       0.40      0.40      0.39      1047
weighted avg       0.40      0.40      0.39      1047
```

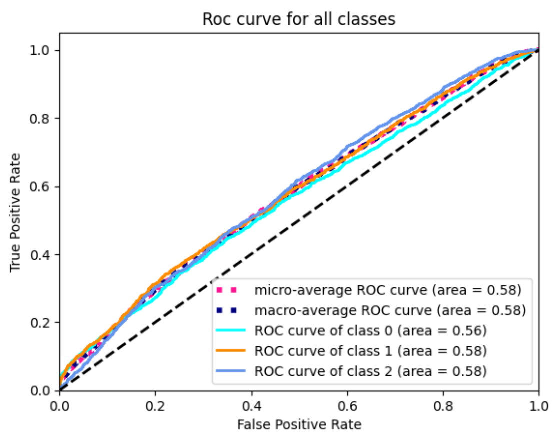
4.3. Comparison with the second project

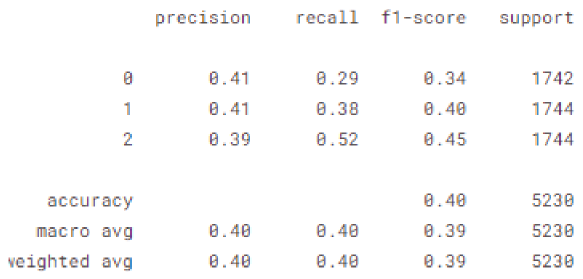
In project 3 we attempted to address the limitations of the unidirectional FeedForward Network of project 2. Considering the hyperparameters and the evaluation results of the 2nd project, it is evident that a single-layer FeedForward network with weight initialization, additional regularization techniques like scaling and batch normalization, a different optimizer (RMSprop) and learning rate with exponential scheduling, as well as 50 epochs and half batch size (128) gave a performance similar to that of the current project. In project 2 it was evident that learning rate scheduling and the use of leaky ReLU variant helped the model stabilize its validation loss and prevent overfitting.
In project 3 we do not observe such stabilization of validation loss, which implies that more regularization techniques are needed. Nevertheless, it is worth noting that with 10 epochs training we managed to get the same results as with project 2, meaning that the bidirectional nature of the network helped it learn faster.

Network Architecture	FFNN (RReLU) n_layers: 1 n_units_l0: 43
Weight initialisation	He
Regularization techniques	Scaling (StandardScaler) Dropout (0.4) Batch Normalisation
Optimizer	RMSprop
Learning rate	lr: 0.004
Lr_scheduler	Exponential (gamma=0.5)
n_epochs	50
Batch_size	128
output activation function	Softmax








```
              precision    recall  f1-score   support

           0       0.41      0.29      0.34      1742
           1       0.41      0.38      0.40      1744
           2       0.39      0.52      0.45      1744

    accuracy                           0.40      5230
   macro avg       0.40      0.40      0.39      5230
weighted avg       0.40      0.40      0.39      5230
```

5. Bibliography

References

[1] Agrawal, R. (2021, April 11). Using fine-tuned Gensim Word2Vec Embeddings with Torchtext and Pytorch. Medium. https://rohit-agrawal.medium.com/using-fine-tuned-gensim-word2vec-embeddings-with-torchtext-and-pytorch-17eea2883cd

[2] Mastering Hyperparameter Tuning with Optuna: Boost Your Machine Learning Models! (n.d.). Www.youtube.com. Retrieved February 3, 2024, from https://www.youtube.com/watch?v=t-INgABWULw&ab_channel=RyanNolanData

[3] Google Colaboratory. (n.d.). Colab.research.google.com. Retrieved February 3, 2024, from https://colab.research.google.com/github/ariepratama/python-playground/blob/master/dl-for-nlp-pytorch-torchtext.ipynb

[4] Tweet Sentiment Analysis Using LSTM With PyTorch. (n.d.). Www.nbshare.io. Retrieved February 3, 2024, from https://www.nbshare.io/notebook/754493525/Tweet-Sentiment-Analysis-Using-LSTM-With-PyTorch/

[5] Nutan. (2022, March 30). Twitter Sentiment Classification In PyTorch. Medium. https://medium.com/@nutanbhogendrasharma/twitter-sentiment-classification-in-pytorch-7379fa75693c

[6]Why do we “pack” the sequences in PyTorch? (n.d.). Stack Overflow. Retrieved February 6, 2024, from https://stackoverflow.com/questions/51030782/why-do-we-pack-the-sequences-in-pytorch

[7]Mihaila, G. (2020, November 13). 🐶 Better Batches with PyTorchText BucketIterator. Medium. https://gmihaila.medium.com/better-batches-with-pytorchtext-bucketiterator-12804a545e2a

[8]Loomis, C. (2021, February 16). Visualizing Hyperparameters in Optuna. Optuna. https://medium.com/optuna/visualizing-hyperparameters-in-optuna-86c224bd255f

[9] Papers with Code - Effective Approaches to Attention-based Neural Machine Translation. (n.d.). Paperswithcode.com. Retrieved February 10, 2024, from https://paperswithcode.com/paper/effective-approaches-to-attention-based

[10]Trevett, B. (2024, February 10). bentrevett/pytorch-sentiment-analysis. GitHub. https://github.com/bentrevett/pytorch-sentiment-analysis/tree/main

[11]L15.7 An RNN Sentiment Classifier in PyTorch. (n.d.). Www.youtube.com. Retrieved February 10, 2024, from https://www.youtube.com/watch?v=KgrdifrlDxg&ab_channel=SebastianRaschka