

Deep Learning for NLP

Konstantinos Malonas

sdi: 7115112200020

Fall Semester 2023

University of Athens Department of Informatics and Telecommunications

Artificial Intelligence II (M138, M226, M262, M325)

Abstract	3
Data processing and analysis	4
Pre-processing	4
Analysis	6
Data partitioning for train, test and validation	10
Vectorization	10
Algorithms and Experiments	12
Experiments	12
Initial neural network	12
Experiments with layers	14
Experiments with the learning rate	16
Experiments with learning rate scheduler	18
Experiments with the batch size	21
Experiments with the activation functions	23
Experiments with optimizers	27
Experiments with concatenated columns Text and Party	30
Table of trials	31
Hyper-parameter tuning	31
Optimization	32
Evaluation	32
Results and overall analysis	32
Best trial	32
Comparison with the first project	32
Bibliography	33

Abstract

For this project we will build a sentiment classifier that classifies tweets written in Greek language about the Greek elections. The classifier will classify the tweets as NEGATIVE, POSITIVE and NEUTRAL classes. First, we will start by exploring my data, which includes plotting barplots, word clouds, etc., to recognize potential patterns, understand the predominant sentiment classifications for tweets associated with each party, and so on. Then we will continue with experiments to find the best hyperparameters for our classifier, that we will build with the use of the PyTorch framework, and also we will plot the learning curves to observe the performance of the classifier we are building.

Data processing and analysis

Pre-processing

First we start with the pre-processing of our dataset. Initially we check if there are any null values in any column with the **info** method. We observe that there are no null values.

```
RangeIndex: 36630 entries, 0 to 36629
Data columns (total 4 columns):
#   Column      Non-Null Count  Dtype
---  -
0   New_ID      36630 non-null  int64
1   Text        36630 non-null  object
2   Sentiment   36630 non-null  object
3   Party       36630 non-null  object
dtypes: int64(1), object(3)
memory usage: 1.1+ MB
```

Turn categorical values to numerical with LabelEncoder

It is a best practice to turn the categorical values of our classes from our dataset to numerical. To achieve that, we use the **LabelEncoder** object. We turn our classes from NEGATIVE to 0, from NEUTRAL to 1, and lastly from POSITIVE to 2.

```
df_train_set['Sentiment'] = le.fit_transform(df_train_set['Sentiment'])
df_valid_set['Sentiment'] = le.fit_transform(df_valid_set['Sentiment'])
```

The preprocess_tweet function

In the data preprocessing stage, the **preprocess_tweet** function plays a crucial role in standardizing and cleaning the tweet data. This function performs several key operations to ensure that the text data is in a suitable format for analysis and modeling.

Firstly, we convert all text to lowercase, which helps in maintaining consistency as text data often contains a mix of uppercase and lowercase letters, and in most contexts, these variations are not meaningful. By standardizing the case, we reduce the complexity of the text and avoid treating the same words in different cases as different tokens.

Next, we remove mentions, which are words starting with the '@' character. Mentions in tweets usually refer to usernames and do not contribute meaningful information for our analysis. Similarly, URLs are removed since they often act as noise in the text analysis, being unique and not contributing to the overall understanding of the tweet's sentiment.

We also remove all non-Greek characters, focusing our analysis strictly on Greek text. This step is crucial as it eliminates irrelevant characters or symbols that might be present in the tweets but do not contribute to their semantic meaning.

Lastly, we remove Greek stopwords using the **stopwords-el.json** file. Stopwords are common words that appear frequently in the text but do not carry significant meaning and are often filtered out before processing text data. Removing these words helps in reducing the dimensionality of the data and focuses the analysis on the words that carry more meaning and sentiment.

```
nlp =
spacy.load('/kaggle/input/el-core-news-lg-2/el_core_news_lg/el_core_news
_lg-3.7.0')

def lemmatize_tokenize_text(text):
    doc = nlp(text)
    return ' '.join([token.lemma_ for token in doc])

df_train_set['Text'] =
df_train_set['Text'].apply(lemmatize_tokenize_text)
df_test_set['Text'] = df_test_set['Text'].apply(lemmatize_tokenize_text)
df_valid_set['Text'] =
df_valid_set['Text'].apply(lemmatize_tokenize_text)
```

If we print the values of the **Text** column of the train,test,validation sets we will notice that the words are lemmatized.

```
0    απολυμανση κοριοι απεντομωση κοριος απολυμανσε...
1    έξι νέος επιστολή μακεδονία καίνε νδ μητσοτάκη...
2    ισχυρός κκε δύναμη λαός βουλή καθημερινός αγώνας
3    μνημονιακότατο μερα25 εκλογες 2019 8 κκε
4    συγκλονιστικός ψυχασθένεια τσίπρας
Name: Text, dtype: object

0    κυριάκος μητσοτάκης ξέρω μουσείο βεργίνας μέσω...
1    συνέντευξη υποψήφιος βουλευτής νέος δημοκρατία...
2    εκλογή μαθητής φοιτητής ψηφίζω ίδιος τρόπος αγ...
3    γεννηματά κινναλ γίνομαι δεκανίκι κανενός ενδια...
4    κυριακός εκλογή οκτώβρης 1993 ξημερώματα δευτέ...
Name: Text, dtype: object

0    θελεις μιλησεις βοσκοτοπια αιγιαλος παραγραφή ...
1    τσίπρας ζητήζω αντιπολίτευση συμμετέχω διαδικα...
2    σωστος ελληνας δημοκρατης ελληνας εξωτερικου ε...
3    βλέπεις ενδιαφέρω μητσοτακηδας γιατί πήγε κότε...
4    συνέντευξη μητσοτάκης αίρεση 13ος σύνταξη αύξη...
Name: Text, dtype: object
```

The unique_words_num function

After we pre-processed our text we printed the unique words of each set with the use of **unique_words_num** function

```
def unique_words_num(tweets):
    # Function that counts the number of the unique words from the Text
    # column of each dataframe
    words = set()
    for tweet in tweets:
        words.update(tweet.split())
    return len(words)
```

Result:

Num of unique words in df_train_set: 58389

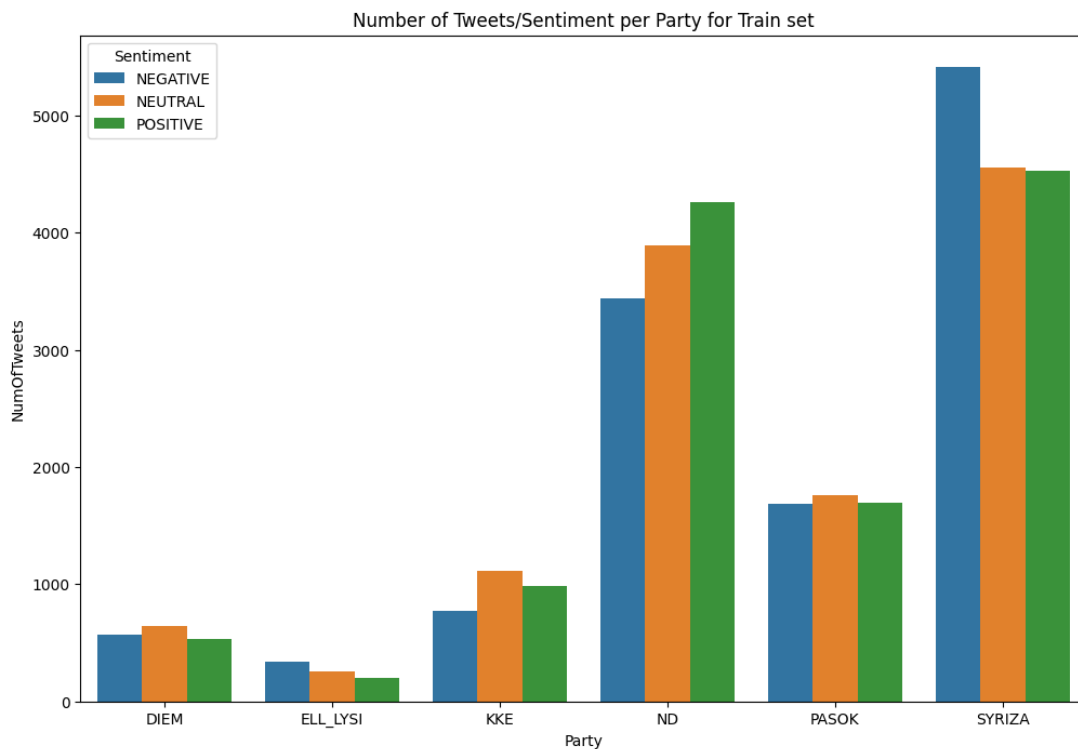
Num of unique words in df_test_set: 26263

Num of unique words in df_valid_set: 16639

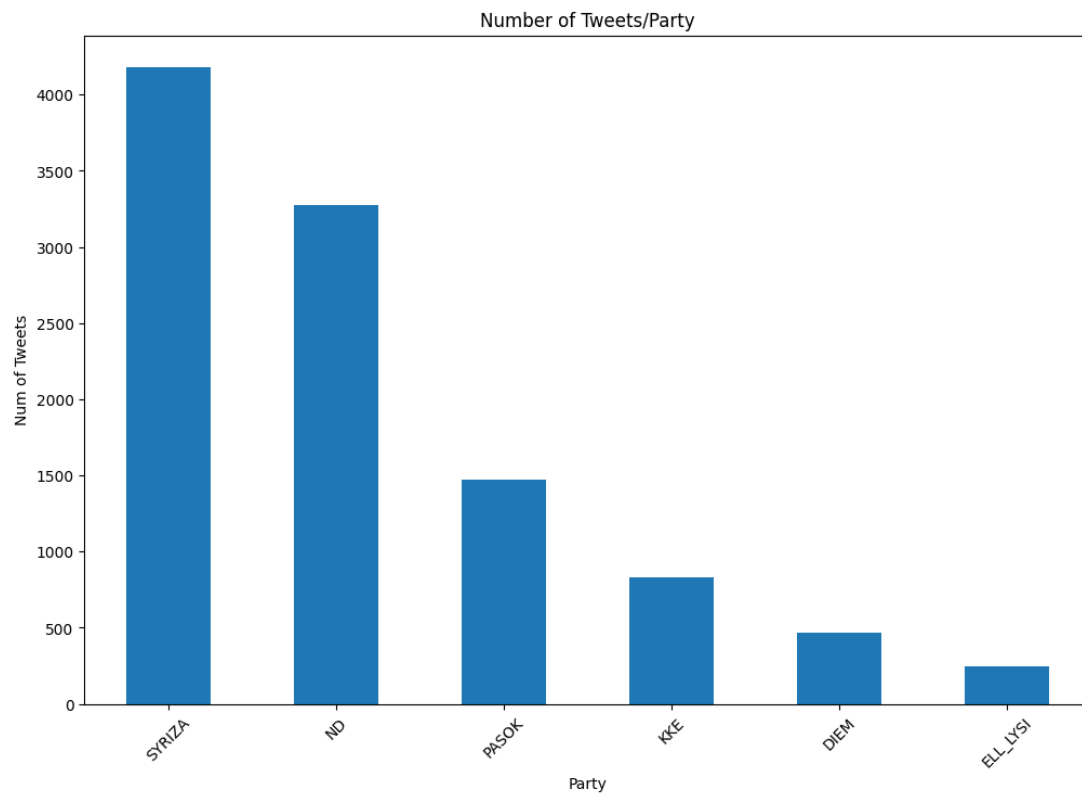
Analysis

For each set (train, validation and test) we plotted some barplots to visualize the number of tweets about each party and the emotion of these tweets. Also we plot the total number of tweets for each party. Also after lemmatization and the general preprocessing of the tweets we plotted the corresponding word cloud for each set.

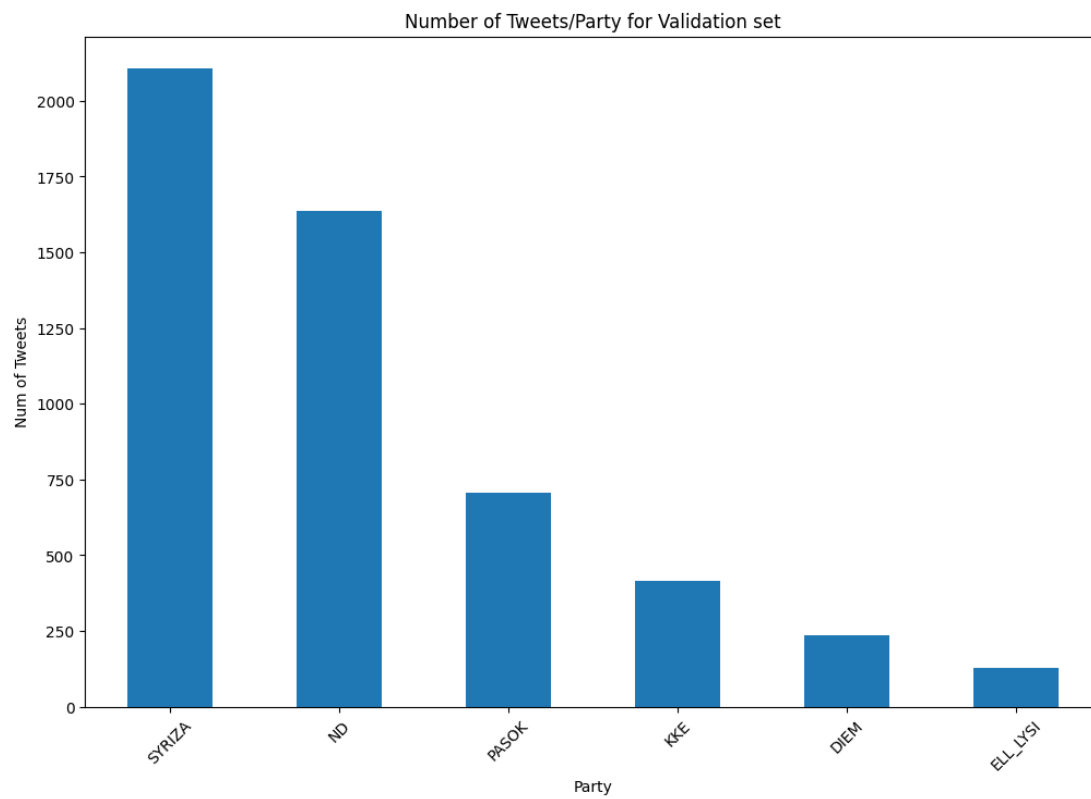
Number of tweets and sentiment for each party



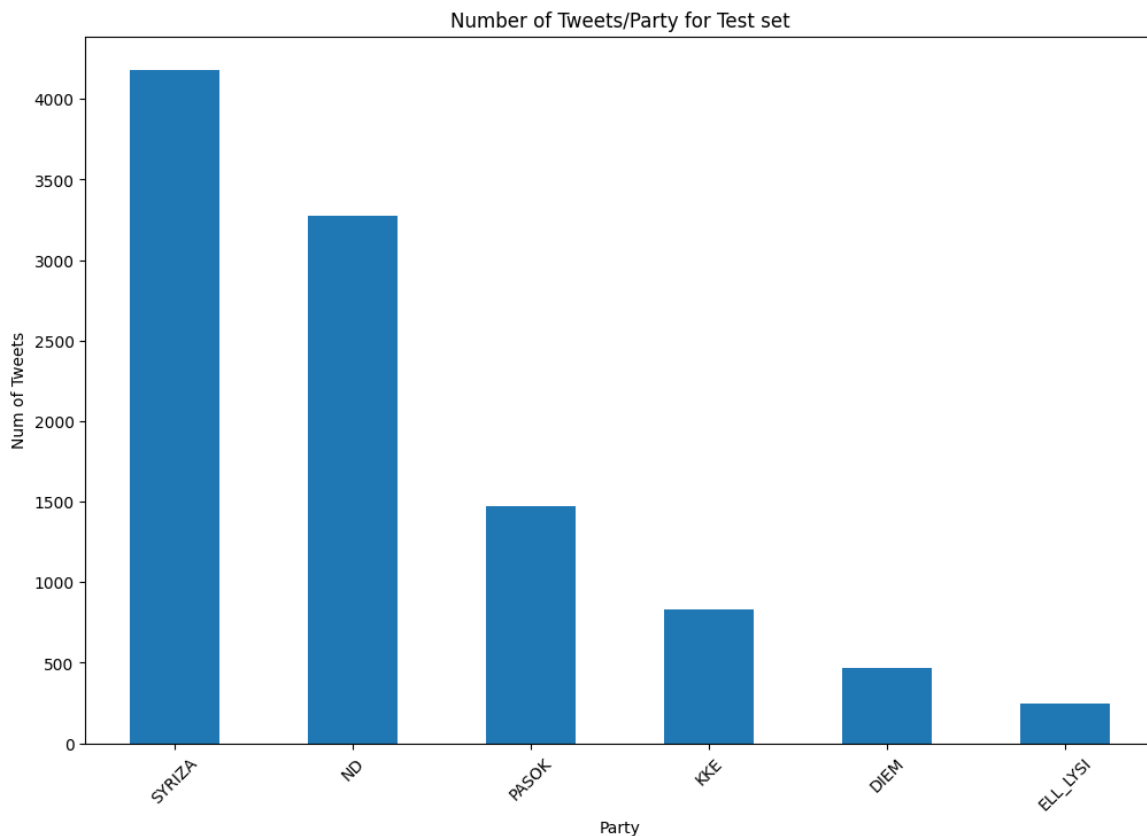
Number of tweets for each party for the train set



Number of tweets for each party from the validation set



Number of tweets for each party from the test set



Word cloud for train set

In our analysis, we have utilized word clouds as a visual tool to represent the data from our train, test, and validation sets. Word clouds, or tag clouds, are graphical representations where the frequency of each word in the text data is depicted with various font sizes. The more frequently a word appears in the dataset, the larger and bolder it appears in the word cloud. This visualization technique is particularly useful for quickly identifying key themes and terms that are most prominent in large volumes of text.

The word clouds generated for our datasets reveal significant insights, especially concerning the political context of the tweets. Given that our data is centered around the Greek general elections, it is not surprising to find that the names of the two dominant parties at the time, Syriza (Σύριζα) and Nea Demokratia (Νέα Δημοκρατία), along with the names of their party leaders, are among the most prominently featured words in the tweets. This observation aligns with the expected discourse during an election period, where discussion and commentary are often focused on the major political parties and their leaders.

[illegible]

In our project, we initially worked with the given datasets as they were originally partitioned into separate training, validation, and testing sets. This conventional partitioning is a standard practice in machine learning, ensuring that models are trained, tuned, and tested on distinct data subsets.

In order for our data to be used from the neural networks we will be building, they must be transformed into vectors of numbers. For that we have used the **Word2Vec** model.

Word2Vec is used in natural language processing to transform words into numerical vectors. It's based on neural networks and aims to capture the **contextual relationships** between words.

Word Embeddings: Word2Vec produces word embeddings, which are numerical representations of words in a high-dimensional space. In this space, semantically similar words are located close to each other.

Resulting Vectors: After training, each word in the model's vocabulary is associated with a fixed-size vector (e.g., 100 dimensions). These vectors capture semantic information about the words.

Vectorize function steps:

- Sentence splitting: Each sentence is split into separate words.
- Word vectorization: For each word in the sentence, the function retrieves its corresponding vector from the **Word2Vec** model. If a word is not in the Word2Vec vocabulary, is handled separately.
- Sentence vector representation: The vectors of individual words are then combined to form a single vector representing the entire sentence.
- Handling unknown words: If a sentence contains words not present in the Word2Vec vocabulary, these words won't contribute to the sentence vector. If there are no known words (i.e., `len(words_vecs) == 0`), we return a zero vector of the same dimensionality as the Word2Vec embeddings.

```
def vectorize(sentence, w2v_model):  
    words = sentence.split()  
    words_vecs = [w2v_model.wv[word] for word in words if word in  
w2v_model.wv]  
    if len(words_vecs) == 0:  
        return np.zeros(100)  
    words_vecs = np.array(words_vecs)  
    return words_vecs.mean(axis=0)
```

Below we can observe the five first rows of our vectorized training set:

```
0    [-0.39609137, 0.23412807, 0.417075, 0.08928634...  
1    [0.4789473, 1.1200287, 0.17128094, 0.11550739,...  
2    [0.18602599, 1.3021685, 0.509936, -1.3635322, ...  
3    [-0.5792007, 0.80550593, 0.71691084, -0.842413...  
4    [-0.20452408, 0.5811979, 0.8934625, 0.16438626...
```

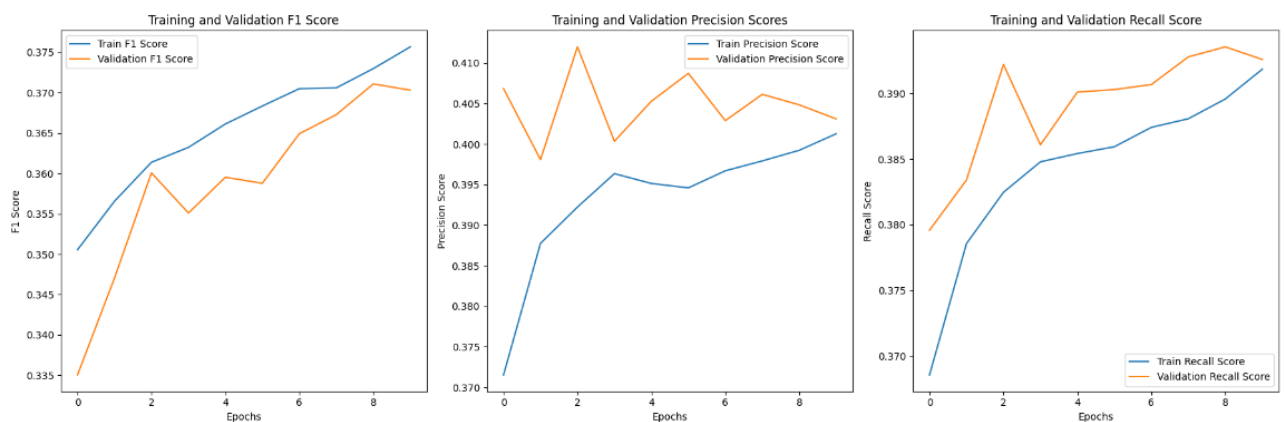
Algorithms and Experiments

Experiments

Initial neural network

Our first neural network consists of 2 hidden layers. We are also using the ReLu activation function. Additionally our nn consists of 100 neurons at the input layer, 128 neurons at the hidden layer 1, 64 neurons at the hidden layer 2 and 3 neurons at the output layer since we want to predict 3 classes. Below we have plotted the training-validation f1, recall and precision scores.

```
class Net(nn.Module):  
    def __init__(self, D_in, H1, H2, D_out):  
        super(Net, self).__init__()  
  
        self.linear1 = nn.Linear(D_in, H1)  
        self.act_1 = nn.ReLU()  
        self.linear2 = nn.Linear(H1, H2)  
        self.act_2 = nn.ReLU()  
        self.linear3 = nn.Linear(H2, D_out)  
  
    def forward(self, x):  
        x = self.act_1(self.linear1(x))  
        x = self.act_2(self.linear2(x))  
        x = self.linear3(x)  
        return x
```



Values during training

Epoch 1/10, Train Acc: 0.37, Valid Acc: 0.38, Train F1: 0.35, Valid F1: 0.34, Train Precision: 0.37, Valid Precision: 0.41, Train Recall: 0.37, Valid Recall: 0.38

Epoch 2/10, Train Acc: 0.38, Valid Acc: 0.38, Train F1: 0.36, Valid F1: 0.35, Train Precision: 0.39, Valid Precision: 0.40, Train Recall: 0.38, Valid Recall: 0.38

Epoch 3/10, Train Acc: 0.38, Valid Acc: 0.39, Train F1: 0.36, Valid F1: 0.36, Train Precision: 0.39, Valid Precision: 0.41, Train Recall: 0.38, Valid Recall: 0.39

Epoch 4/10, Train Acc: 0.38, Valid Acc: 0.39, Train F1: 0.36, Valid F1: 0.36, Train Precision: 0.40, Valid Precision: 0.40, Train Recall: 0.38, Valid Recall: 0.39

Epoch 5/10, Train Acc: 0.39, Valid Acc: 0.39, Train F1: 0.37, Valid F1: 0.36, Train Precision: 0.40, Valid Precision: 0.41, Train Recall: 0.39, Valid Recall: 0.39

Epoch 6/10, Train Acc: 0.39, Valid Acc: 0.39, Train F1: 0.37, Valid F1: 0.36, Train Precision: 0.39, Valid Precision: 0.41, Train Recall: 0.39, Valid Recall: 0.39

Epoch 7/10, Train Acc: 0.39, Valid Acc: 0.39, Train F1: 0.37, Valid F1: 0.36, Train Precision: 0.40, Valid Precision: 0.40, Train Recall: 0.39, Valid Recall: 0.39

Epoch 8/10, Train Acc: 0.39, Valid Acc: 0.39, Train F1: 0.37, Valid F1: 0.37, Train Precision: 0.40, Valid Precision: 0.41, Train Recall: 0.39, Valid Recall: 0.39

Epoch 9/10, Train Acc: 0.39, Valid Acc: 0.39, Train F1: 0.37, Valid F1: 0.37, Train Precision: 0.40, Valid Precision: 0.40, Train Recall: 0.39, Valid Recall: 0.39

Epoch 10/10, Train Acc: 0.39, Valid Acc: 0.39, Train F1: 0.38, Valid F1: 0.37, Train Precision: 0.40, Valid Precision: 0.40, Train Recall: 0.39, Valid Recall: 0.39

Mean values of metrics

Train accuracy	Validation accuracy	Train F1	Validation F1	Train Recall	Validation Recall	Train Precision	Validation Precision
0.38	0.39	0.37	0.36	0.38	0.39	0.39	0.40

Observations

Validation Accuracy VS Training Accuracy: The validation accuracy is generally the same as the training accuracy, which is typical sign of our model overfitting, sometimes validation accuracy surpasses training accuracy, which is unusual and might indicate variability in the data (which is logical since we have to do with tweets) or model's learning process.

F1 Scores: The F1 scores are relatively stable and hover around the same values throughout the epochs. This might suggest that the model is consistently predicting a certain class or set of classes well, but not necessarily improving across all classes.

Model Stability: The variability in accuracy and F1 score from epoch to epoch suggests that the model is not very stable. This could be due to several factors, including the choice of model architecture, learning rate, or the nature of the data itself.

Experiments with layers

We have performed experiments with 4, 6 and 8 hidden layers in order to achieve higher metrics.

Performance across different models:

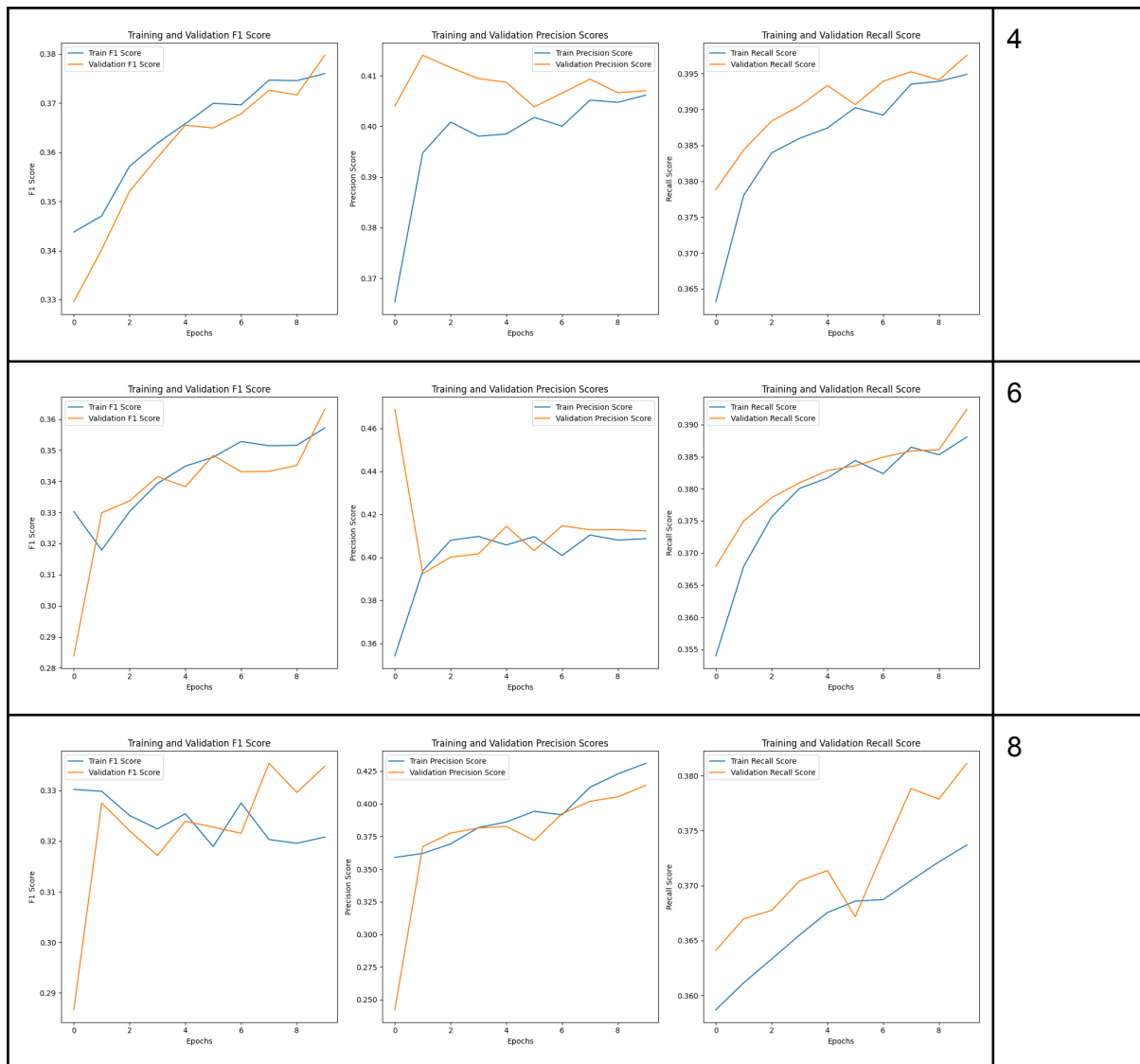
- **4 hidden layers:** This model achieves approximately 39% accuracy, 38% f1 score, 41% precision, and 40% recall on both training and validation sets by the 10th epoch. These results indicate a relatively balanced performance across metrics.
- **6 hidden layers:** The performance is similar to the 4-layer model in terms of accuracy and f1 score. However, there is a noticeable drop in precision in the initial epochs, which later stabilizes to be close to the 4-layer model.
- **8 hidden layers:** This model shows a slightly lower performance compared to the 4 and 6-layer models. The accuracy, f1 score, precision, and recall are consistently a bit lower across epochs.

Stability - Learning trends: All models show a consistent learning trend without major fluctuations, indicating stable learning. However, there is no significant improvement in performance as the number of hidden layers increases from 4 to 8. The slight decrease in performance with 8 hidden layers might suggest overfitting or that the model is becoming **too complex** for the given dataset.

Best result: From the metrics we got during training, the 4 hidden layer model seems to be the best performing. It achieves slightly higher accuracy and F1 scores compared to the 6 and 8-layer models, and its precision and recall are also comparatively balanced. The increment in layers does not proportionally improve the performance, suggesting that additional complexity might not be necessary for this dataset.

Below we can observe the plots for the training-validation f1, precision, recall scores.

F1 - Precision - Recall	Layers
-------------------------	--------



Mean values of training-validation f1, precision, recall scores

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Layers
0.36	0.36	0.40	0.41	0.39	0.39	4
0.34	0.34	0.40	0.41	0.38	0.38	6
0.32	0.32	0.39	0.37	0.37	0.37	8

Experiments with the learning rate

We will proceed with experiments with the learning rate. The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

The learning rate may be the most important hyperparameter when configuring a neural network. Therefore it is vital to know how to investigate the effects of the learning rate on model performance and to build an intuition about the dynamics of the learning rate on model behavior.

We also modified our training function in order to use **early stopping**. In machine learning, early stopping is a form of regularization used to avoid overfitting when training a learner with an iterative method, such as gradient descent. Such methods update the learner so as to make it better fit the training data with each iteration. Up to a point, this improves the learner's performance on data outside of the training set. Past that point, however, improving the learner's fit to the training data comes at the expense of increased generalization error. Early stopping rules provide guidance as to how many iterations can be run before the learner begins to over-fit. Early stopping rules have been employed in many different machine learning methods, with varying amounts of theoretical foundation.

In our case we stop our training if the validation f1 score hasn't improved after 20 epochs. We conducted experiments for learning rate **0.1**, **0.001**, **0.0001**, **0.00001**. Based on the results we got from the metrics we print during training we concluded to the following observations:

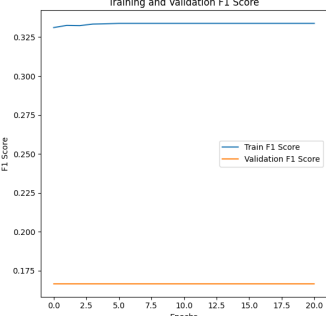
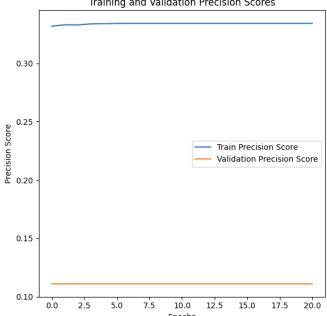
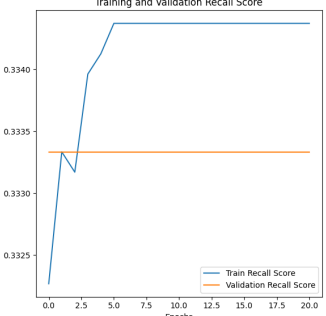
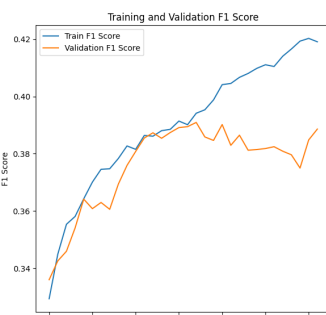
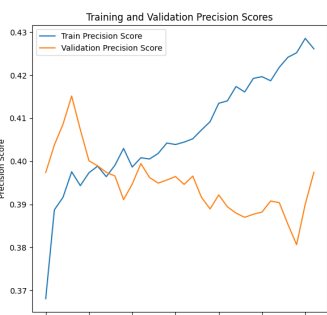
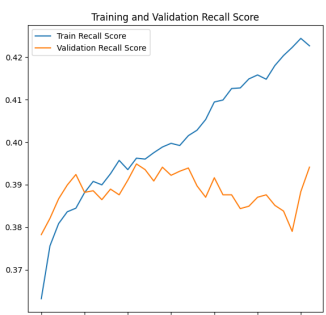
Learning rate 0.1: The model did not learn anything, as indicated by constant metrics across epochs. This suggests the learning rate was too high, causing the model to overshoot the optimal point in the loss landscape.

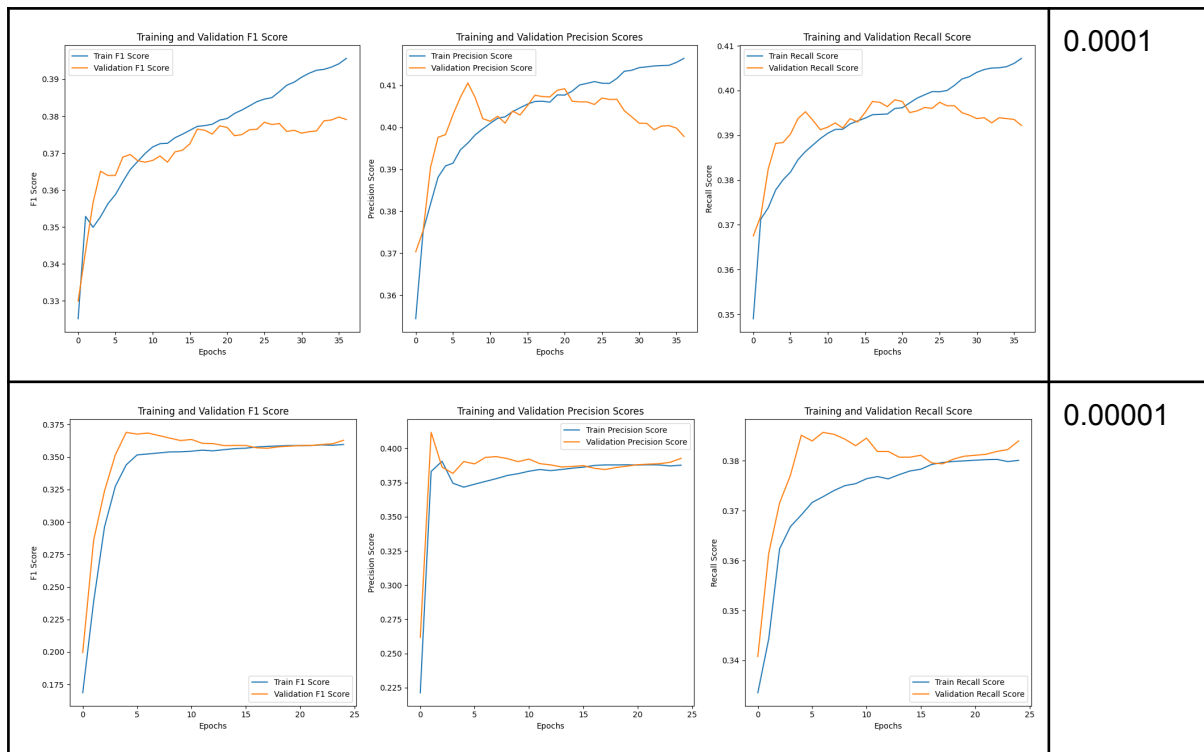
Learning rate 0.001: This rate shows the best overall performance with gradual improvements across epochs. The metrics like accuracy, f1 score, precision, and recall all improve steadily, indicating effective learning.

Learning rate 0.0001: While there is some learning, the improvements are slower and not as clear as with a learning rate of 0.001. This slower rate of improvement suggests that the learning rate might be a bit too low, causing slower convergence.

Learning rate 0.00001: Similar to 0.0001, there's learning, but it's slower. The model takes more epochs to reach a performance comparable to higher learning rates, indicating inefficiency.

Best result: Overall, a learning rate of **0.001** achieves a good balance between learning efficiency and model performance in our experiments. It enables the model to learn effectively without overshooting or being too slow to converge. For learning rates 0.1, 0.001, 0.0001 and 0.00001 the training stopped after 21, 32, 37, 25 epochs respectively.

F1 - Precision - Recall			Learning rate
<div>    </div>			0.1
<div>    </div>			0.001



Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Learning rate
0.33	0.17	0.33	0.11	0.33	0.33	0.1
0.39	0.38	0.41	0.39	0.40	0.39	0.001
0.38	0.37	0.40	0.40	0.39	0.39	0.0001
0.34	0.35	0.38	0.38	0.37	0.38	0.00001

Experiments with learning rate scheduler

Next we conducted experiments with various learning rate schedulers. Learning rate schedulers are used to make learning rate adaptive to the gradient descent optimization procedure, so you can increase performance and reduce training time.

In the neural network training process, data is feed into the network in batches, with many batches in one epoch. Each batch triggers one training step, which the gradient descent algorithm updates the parameters once. However, usually the learning rate schedule is updated once for each training epoch only.

We can update the learning rate as frequent as each step but usually it is updated once per epoch because we want to know how the network performs in order to determine how the

learning rate should update. Regularly, a model is evaluated with validation dataset once per epoch.

There are multiple ways of making learning rate adaptive. At the beginning of training, we may prefer a larger learning rate so we improve the network coarsely to speed up the progress. In a very complex neural network model, you may also prefer to gradually increase the learning rate at the beginning because you need the network to explore on the different dimensions of prediction. At the end of training, however, we always want to have the learning rate smaller. Since at that time, we are about to get the best performance from the model and it is easy to overshoot if the learning rate is large.

Therefore, the simplest and perhaps most used adaptation of the learning rate during training are techniques that reduce the learning rate over time. These have the benefit of making large changes at the beginning of the training procedure when larger learning rate values are used and decreasing the learning rate so that a smaller rate and, therefore, smaller training updates are made to weights later in the training procedure.

We conducted experiments with the StepLR, ExponentialLR and ReduceLROnPlateau schedulers.

StepLR scheduler: This scheduler led to steady improvements in all metrics over epochs. The early stopping was triggered after 30 epochs, with a mean training and validation accuracy of 0.40, and mean F1 scores of 0.38. The improvement pattern is consistent and gradual, which is a good sign of effective learning.

ExponentialLR scheduler: With the ExponentialLR, we achieved slightly better results compared to StepLR. The early stopping was triggered after 35 epochs, but the mean training accuracy reached 0.41, and the mean training F1 score was 0.40. This suggests a more effective learning process, possibly due to the more dynamic adjustment of the learning rate.

ReduceLROnPlateau scheduler: This scheduler shows a similar performance to the StepLR, with early stopping triggered after 38 epochs. The mean metrics are comparable to those achieved with the StepLR, suggesting that it didn't offer a significant advantage in this case.

Best result: Based on these observations, the **ExponentialLR** Scheduler seems to be the most effective for our neural network model. It leads to slightly better performance metrics in a comparable number of epochs. The ExponentialLR Scheduler continuously decays the learning rate over epochs, allowing for finer adjustments as the training progresses. This can be especially useful for converging to an optimal solution more effectively in complex models and datasets.

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Learning rate scheduler
------------------	--------------------	-------------------------	---------------------------	----------------------	------------------------	-------------------------

Experiments with the batch size

Batch size is one of the most important hyperparameters in deep learning training, and it represents the number of samples used in one forward and backward pass through the network and has a direct impact on the accuracy and computational efficiency of the training process. The batch size can be understood as a trade-off between accuracy and speed. Large batch sizes can lead to faster training times but may result in lower accuracy and overfitting, while smaller batch sizes can provide better accuracy, but can be computationally expensive and time-consuming.

The batch size can also affect the convergence of the model, meaning that it can influence the optimization process and the speed at which the model learns. Small batch sizes can be more susceptible to random fluctuations in the training data, while larger batch sizes are more resistant to these fluctuations but may converge more slowly.

It is important to note that there is no one-size-fits-all answer when it comes to choosing a batch size, as the ideal size will depend on several factors, including the size of the training dataset, the complexity of the model, and the computational resources available.

We did experiments with batch size equal with 8, 12, 32, 64, 128. Based on the results we received from each experiments we concluded to the following observations:

Batch size 8: Achieved the highest mean training accuracy (0.42) and f1 score (0.41). Early stopping was triggered after 52 epochs, indicating a slower convergence. This batch size leads to more frequent updates, which can help in better generalization but might be computationally expensive.

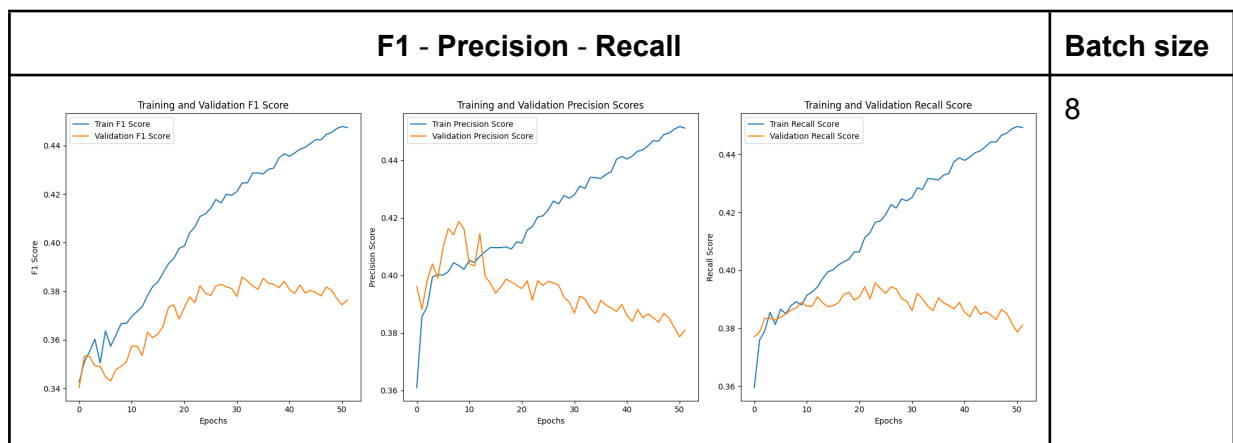
Batch size 12: Slightly lower mean training accuracy (0.40) and f1 score (0.39) compared to batch size 8. Early stopping was triggered after 34 epochs, faster than batch size 8. Offers a balance between update frequency and computational demand.

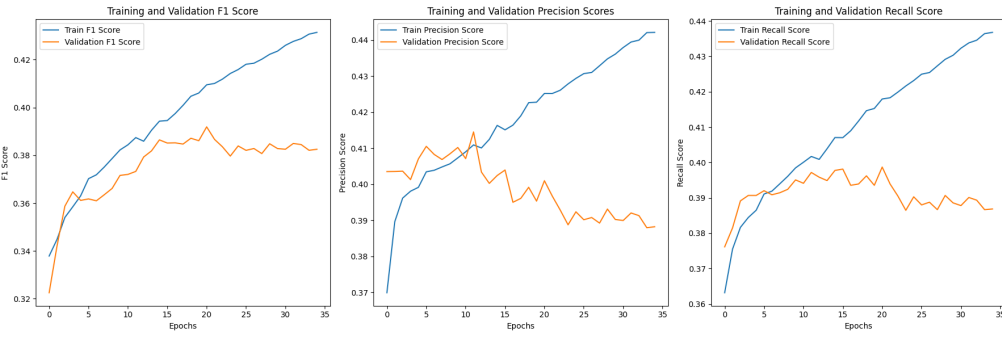
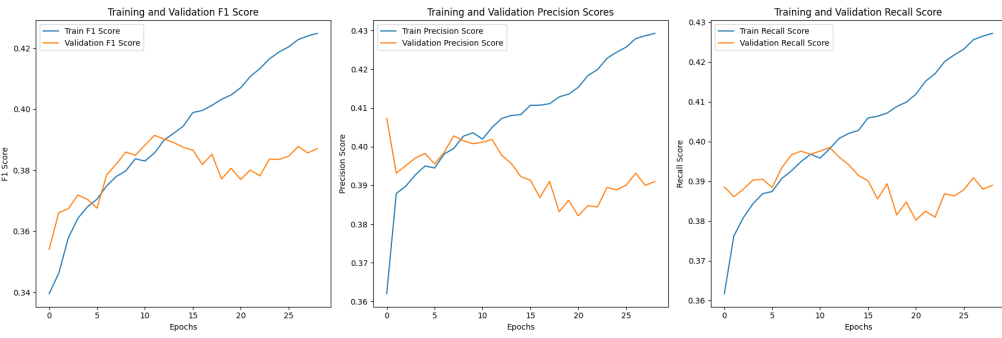
Batch size 32: Comparable performance to batch size 12 with a mean training accuracy of 0.41 and f1 score of 0.40. Early stopping was triggered after 35 epochs. Larger batch size can lead to more stable gradient updates but might reduce model's ability to generalize.

Batch size 64 and 128: Both show similar trends in training accuracy and f1 score, hovering around 0.40 and 0.39 respectively. Early stopping was triggered sooner (29 epochs for batch size 64 and 31 for batch size 128) than smaller batch sizes. Larger batches lead to quicker convergence but might miss out on finer features due to averaging effect in gradient calculation.

Best result: Based on the values we got for mean training-validation accuracy, f1, precision and recall we choose to use batch size equal to **8**.

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Batch size
0.41	0.37	0.42	0.39	0.42	0.39	8
0.39	0.37	0.41	0.40	0.40	0.39	12
0.40	0.38	0.42	0.40	0.41	0.39	32
0.39	0.38	0.41	0.39	0.40	0.39	64
0.39	0.38	0.41	0.40	0.40	0.39	128



	12
	32
	64
	128

Experiments with the activation functions

Sigmoid: It showed a gradual improvement over epochs but still had relatively low performance metrics. Sigmoid functions often suffer from the vanishing gradient problem, especially in deeper networks.

Tanh: This activation function provided slightly better results than Sigmoid. Tanh is similar to Sigmoid but ranges from -1 to 1, providing stronger gradients for negative values compared to Sigmoid. However, it still suffers from the vanishing gradient problem.

Leaky ReLU: This function showed a significant improvement over Sigmoid and Tanh, especially in terms of f1 score, precision, and recall. Leaky ReLU helps in preventing the dying ReLU problem by allowing a small gradient when the unit is not active.

Parametric ReLU (PReLU): The performance of PReLU is comparable to Leaky ReLU, with slightly better F1 scores and precision in some epochs. PReLU introduces a learnable parameter which allows adaptation during training, potentially leading to better performance.

Exponential Linear Unit: Provided good results, similar to Leaky ReLU and PReLU. ELU can help converge cost to zero faster and produce more accurate results. It tends to outperform other ReLU variants, especially in deeper networks.

Softmax: The use of Softmax in hidden layers resulted in significantly lower performance metrics across all categories. Softmax is typically used in the output layer for multi-class classification tasks and is not suitable for hidden layers as it normalizes layer outputs to a probability distribution, which is not beneficial in this context.

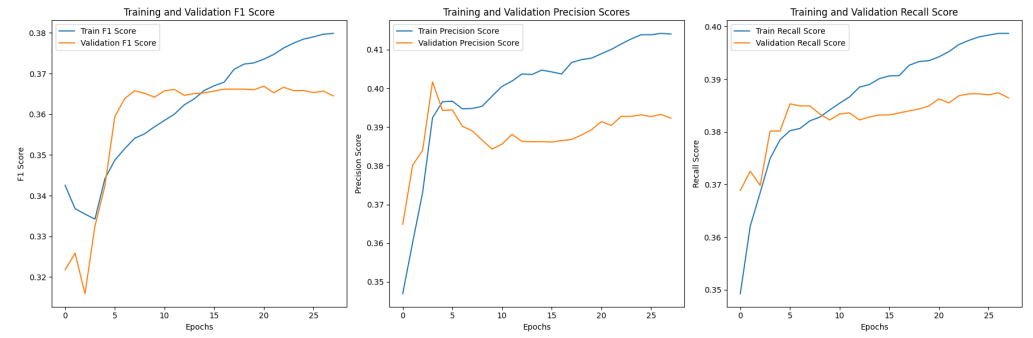
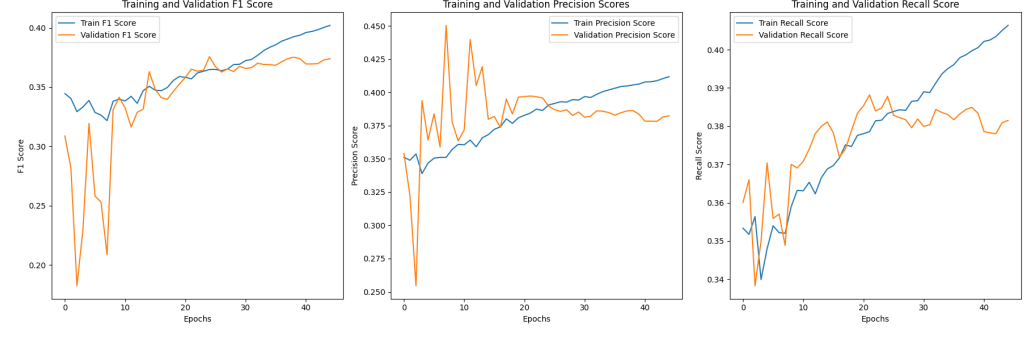
ReLU: ReLU shows a consistent slight improvement in training accuracy, f1 score, precision, and recall over the epochs. It demonstrates a decent performance among the activation functions we tested, with a balanced improvement across all metrics.

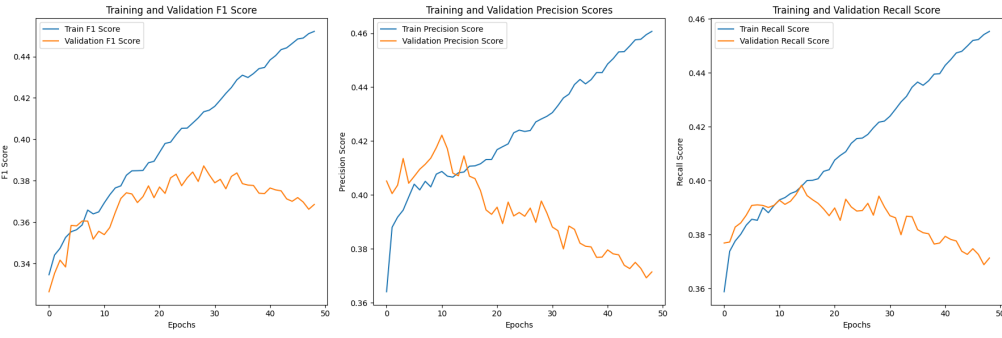
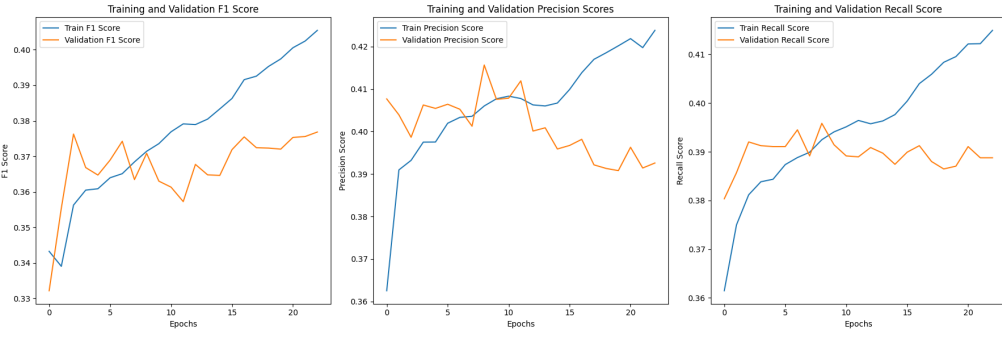
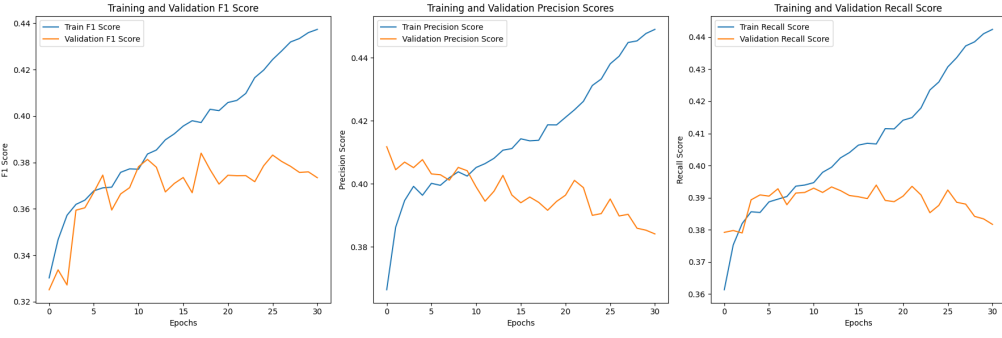
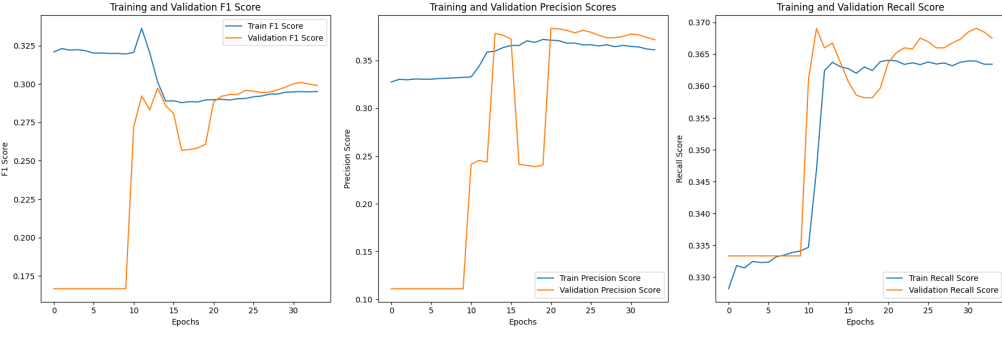
ReLU and Softmax at the output layer: Both training and validation accuracy improve gradually over epochs, indicating that the model is learning effectively. The mean training accuracy reaches 0.40, and the mean validation accuracy is 0.39, suggesting a good fit without significant overfitting. Similarly, the f1 score, which is a more robust measure than accuracy in many cases, also shows a stable increase.

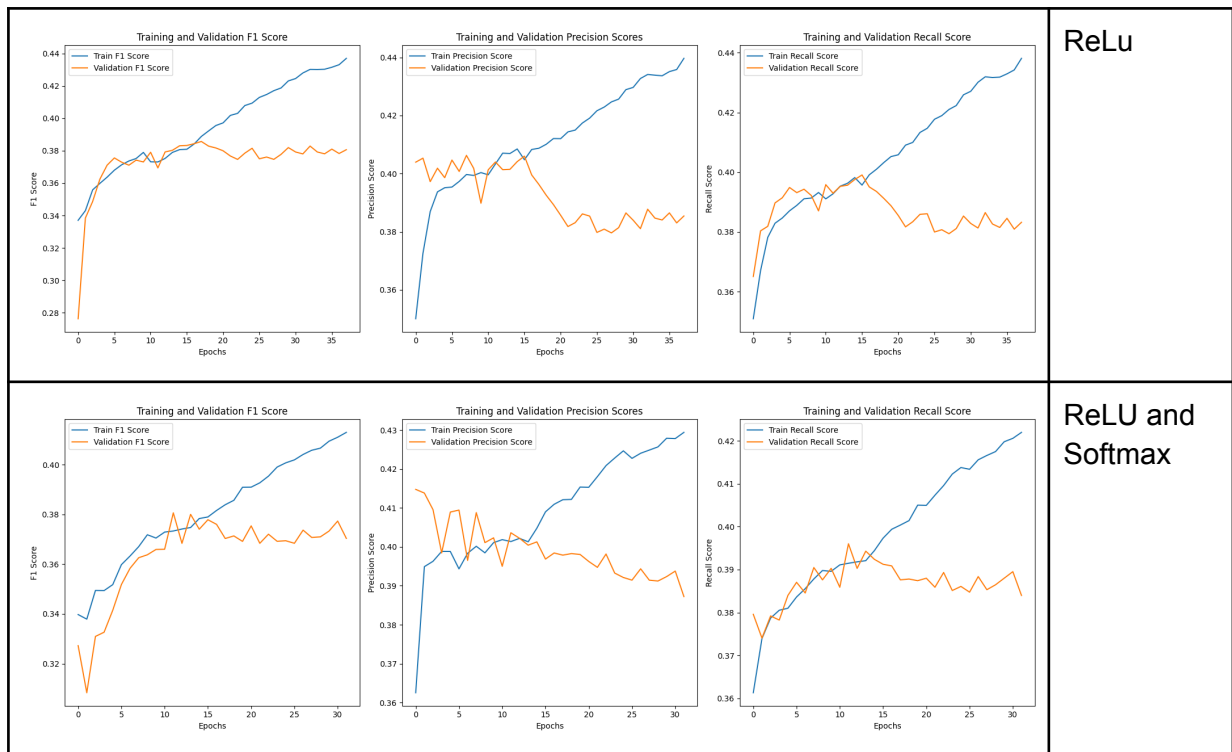
Best results: Based on these results, Leaky ReLU, PReLU, ELU and ReLU seem to be the best performing activation functions for your network. They consistently show higher accuracy, precision, recall, and F1 scores compared to Sigmoid, Tanh, and Softmax. Among these, **ReLU** shows a slightly better average in terms of **train accuracy** and **f1 scores** compared to Leaky ReLU, PReLU and ELU. However, the differences are not substantial, and all three perform similarly well.

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Activation function
0.36	0.36	0.40	0.39	0.39	0.38	Sigmoid
0.36	0.34	0.38	0.38	0.38	0.38	Tanh

0.40	0.37	0.42	0.39	0.41	0.39	Leaky ReLU
0.38	0.37	0.41	0.40	0.40	0.39	Parametric ReLU (PReLU)
0.39	0.37	0.42	0.40	0.41	0.39	Exponential Linear Unit
0.30	0.25	0.35	0.27	0.35	0.36	Softmax
0.39	0.37	0.41	0.39	0.41	0.39	ReLU
0.38	0.36	0.41	0.40	0.40	0.39	ReLU/Soft max

F1 - Precision - Recall			Activation function
 <p>The Sigmoid activation function results in stable performance metrics. The Training F1 Score (blue) increases from ~0.34 to ~0.38. The Validation F1 Score (orange) increases from ~0.32 to ~0.37. The Training Precision Score (blue) increases from ~0.35 to ~0.41. The Validation Precision Score (orange) increases from ~0.36 to ~0.39. The Training Recall Score (blue) increases from ~0.35 to ~0.40. The Validation Recall Score (orange) increases from ~0.37 to ~0.39.</p>			Sigmoid
 <p>The Tanh activation function results in highly volatile and noisy performance metrics. The Training F1 Score (blue) fluctuates between 0.20 and 0.40. The Validation F1 Score (orange) fluctuates between 0.20 and 0.38. The Training Precision Score (blue) fluctuates between 0.25 and 0.45. The Validation Precision Score (orange) fluctuates between 0.25 and 0.45. The Training Recall Score (blue) fluctuates between 0.34 and 0.40. The Validation Recall Score (orange) fluctuates between 0.34 and 0.40.</p>			Tanh

	<p>Leaky ReLU</p>
	<p>Parametric ReLU (PReLU)</p>
	<p>Exponential Linear Unit (ELU)</p>
	<p>Softmax</p>



Experiments with optimizers

Optimization is a process where we try to find the best possible set of parameters for a deep learning model. Optimizers generate new parameter values and evaluate them using some criterion to determine the best option. Being an important part of neural network architecture, optimizers help in determining best weights, biases or other hyper-parameters that will result in the desired output. We conducted experiments with SGD, Adagrad, RMSprop, Adam and AdamW.

SGD: Showed steady but relatively slow improvement over epochs. Achieved a mean training accuracy of 0.36 and validation accuracy of 0.38. The f1 scores, precision, and recall are relatively lower compared to other optimizers, indicating a slower convergence rate.

Adagrad: Demonstrated consistent improvement in performance metrics across epochs. Achieved similar mean training and validation accuracies (0.38) as SGD but with slightly higher f1 scores, precision, and recall. It adapts the learning rates for each parameter, which seems beneficial for the model.

RMSprop: Showed improved performance compared to SGD and Adagrad, with a mean training accuracy of 0.40 and validation accuracy of 0.39. Higher f1 scores, precision, and recall compared to SGD and Adagrad. This optimizer is known for its capability to handle non-stationary objectives and noisy gradients, which might have contributed to its better performance.

Adam: Delivered the highest mean training accuracy (0.41) and comparable validation accuracy (0.39) among the tested optimizers. f1 scores, precision, and recall are higher than those achieved with SGD and Adagrad, and on par with RMSprop. Adam's adaptive learning rate mechanism could be the reason for its kinda effective performance.

AdamW: Performance is quite similar to that of the Adam optimizer with a slight edge in some metrics. Achieved a mean training accuracy of 0.40 and validation accuracy of 0.39. f1 scores, precision, and recall are consistent with Adam's results. The separation of weight decay from gradient updates could have contributed to its efficiency.

In summary, **Adam** and **AdamW** optimizers seem to provide the best performance for our model in terms of training accuracy, f1 score, precision, and recall. RMSprop also shows good performance, slightly trailing behind Adam and AdamW

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall	Optimizer
0.34	0.34	0.37	0.39	0.36	0.38	SGD
0.35	0.36	0.38	0.39	0.38	0.38	Adagrad
0.38	0.38	0.41	0.40	0.40	0.39	RMSprop
0.40	0.37	0.41	0.39	0.41	0.39	Adam
0.39	0.37	0.41	0.39	0.40	0.39	AdamW

F1 - Precision - Recall			Optimizer
<div> <div>Training and Validation F1 Score</div> </div> <div> <div>Training and Validation Precision Scores</div> </div> <div> <div>Training and Validation Recall Score</div> </div>			SGD
<div> <div>Training and Validation F1 Score</div> </div> <div> <div>Training and Validation Precision Scores</div> </div> <div> <div>Training and Validation Recall Score</div> </div>			Adagrad
<div> <div>Training and Validation F1 Score</div> </div> <div> <div>Training and Validation Precision Scores</div> </div> <div> <div>Training and Validation Recall Score</div> </div>			RMSprop
<div> <div>Training and Validation F1 Score</div> </div> <div> <div>Training and Validation Precision Scores</div> </div> <div> <div>Training and Validation Recall Score</div> </div>			Adam



Experiments with concatenated columns Text and Party

We concatenated the Text and the Party columns in order to achieve a higher f1 score without success. As you can observe from the following results the metrics are pretty much the same as before when we used only the Text column.

Mean training f1	Mean validation f1	Mean training precision	Mean validation precision	Mean training recall	Mean validation recall
0.40	0.37	0.42	0.39	0.41	0.39

F1 - Precision - Recall

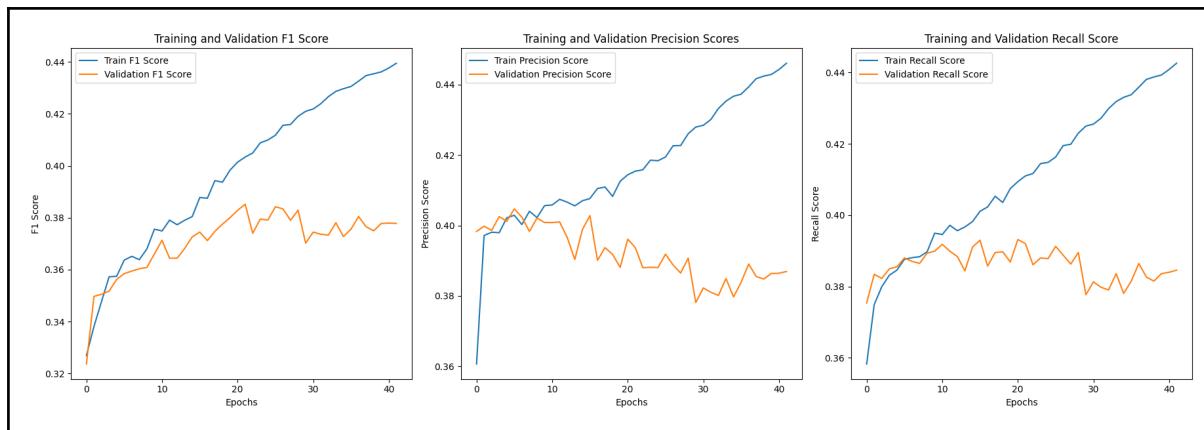


Table of trials

Below is the table of trials where you can observe the values which led our model to perform slightly better, even though the change was insignificant.

Hyperparameter	Value	F1 train	F1 validation
layers	4	0.36	0.36
learning rate	0.001	0.39	0.38
learning rate scheduler	ExponentialLR	0.40	0.38
batch size	8	0.41	0.37
activation functions	ReLU	0.39	0.37
optimizers	Adam	0.40	0.37

Concatenation of Text and Party columns	Concatenated Text/Party columns	0.40	0.37
---	---------------------------------	------	------

Hyper-parameter tuning

For the hyper-parameter tuning we conducted experiments with the layers, the learning rate, the learning rate scheduler, the batch size, the activation functions, the optimizers and also we concatenated two columns in order to achieve higher metrics. We found that the values for the hyper-parameters that led the model to perform slightly better were 4 layers, 0.001 for the learning rate, ExponentialLR for the learning rate scheduler, 8 for the batch size, ReLU for the activation functions and Adam for the optimizer of our model. Additionally we observed that even when we concatenated the Text and Party columns we didn't achieve better results.

In all our trials, the evident **overfitting**, since almost every time the train score is greater than the validation score, of our model suggests that the unique characteristics of tweet data present a unique challenge. This overfitting may derive from the inherently noisy and sparse nature of text data derived from social media, where slang, abbreviations, and diverse linguistic expressions prevail. Such characteristics can lead the model to learn patterns specific to the training set that do not generalize well to unseen data. Additionally, the limitation of a relatively small dataset could exacerbate the issue, as the model might not be exposed to a sufficiently varied representation of the problem space.

Optimization

For the optimization we used various optimizers like SGD, Adagrad, RMSprop, Adam and AdamW. It was evident for every experiment that our model is very unstable from the fluctuation of our metrics. The best optimizers were Adam and AdamW.

Evaluation

For evaluation of every hyperparameter and for every experiment we were plotting the learning curves that was showing the f1, recall and precision training and validation scores. We focused mostly on increasing the f1 score since we are building a classifier.

f1: F1 Score is a measure that combines recall and precision. As we have seen there is a trade-off between precision and recall, F1 can therefore be used to measure how effectively our models make that trade-off. One important feature of the F1 score is that the result is zero if any of the components (precision or recall) fall to zero. Thereby it penalizes extreme negative values of either component.

Precision: Precision is a metric that gives you the proportion of true positives to the amount of total positives that the model predicts. It answers the question “Out of all the positive predictions we made, how many were true?”

Recall: Recall focuses on how good the model is at finding all the positives. Recall is also called true positive rate and answers the question “Out of all the data points that should be predicted as true, how many did we correctly predict as true?”

Results and overall analysis

Best trial

The best trial was for the number of **layers** equal to **4**, **learning rate** equal to **0.001**, **learning rate scheduler** equal to **ExponentialLR**, **batch size** equal to **8**, **activation function** equal to **ReLU** and **optimizer** equal to **Adam**.

Comparison with the first project

In comparison with the first project where we used LogisticRegression of scikit-learn, we achieved a slightly higher f1 training and validation score using Pytorch neural networks, although the difference is insignificant.

Bibliography

- <https://www.kaggle.com/code/mlwhiz/multiclass-text-classification-pytorch>
- <https://turbolab.in/text-classification-with-keras-and-glove-word-embeddings/>
- <https://coderzcolumn.com/tutorials/artificial-intelligence/how-to-use-glove-embeddings-with-pytorch>
- <https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Text-Classification/blob/master/utils.py>
- <https://www.analyticsvidhya.com/blog/2020/01/first-text-classification-in-pytorch/>
- https://en.wikipedia.org/wiki/Early_stopping
- <https://www.sabrepc.com/blog/Deep-Learning-and-AI/Epochs-Batch-Size-Iterations>
- https://en.wikipedia.org/wiki/Activation_function
- <https://machinelearningmastery.com/using-optimizers-from-pytorch/>