Performance Analysis On Sarcasm Based Data On Different combination of Deep Learning architectures tested on Facebook's FastText and Google's Glove embedding.

TABLE OF CONTENTS

S.NO	Chapter	PAGE NO.
1	Chapter -1 Introduction	7
2	Chapter – 2 Literature Review	8
3	Chapter -3 Methodology	10
4	Chapter 4 – Results and Discussion	16
5	Chapter 5 - Conclusion	17
6	Reference	18
7	Appendix	19

ABSTRACT

Sarcasm is a complicated linguistic term commonly found in e-commerce and social media sites. Sarcasm is often used to express a negative opinion using positive or intensified positive words in social media. This intentional ambiguity makes sarcasm detection, an important task of sentiment analysis. Failure to identify sarcastic utterances in Natural Language Processing applications such as sentiment analysis and opinion mining will confuse classification algorithms and generate false results. Several studies on sarcasm detection have utilised different learning algorithms. In this project we worked on fastText and Global vectors for word representation (GloVe) and have implemented 3 deep learning algorithms namely, CNN+ Bi-LSTM, GRU + Bi-LSTM and Bi-LSTM with glorot uniform weight initializer with various activation functions and optimizers. The main purpose of this project is to compare the accuracy scores of all possible combinations and identify the best performing model, based on the model's accuracy.

INTRODUCTION

Sarcasm uniquely is the expression of mockery by using words that exactly mean the opposite of what we are to convey. For example, a tweet/post, ‘‘It is a wonderful feeling to waste hours in traffic jams!’’ clearly indicates this discord between the actual situation of ‘‘being stuck in traffic jam’’ and the utterance content ‘‘wonderful.’’ This contrast and shift of sentiments in sarcastic expressions validates sarcasm as a special instance of sentiment analysis. Automatic sarcasm detection is typically a text classification problem, which relies on a variety of feature extraction and learning techniques. In many research works, twitter datasets are considered. But the major limitation is that sometimes the tweets considered would be replies of other tweets and due to lack of context sarcasm detection will become difficult. So, to overcome this kind of limitation we use a News Headlines dataset to perform classification.

LITERATURE REVIEW

1. Performance Evaluation of Word Embeddings for Sarcasm Detection- A Deep Learning Approach

   This paper aims to depict the relevance of word embeddings in sarcasm detection. This work emphasizes the significance of global context-based word embeddings in sarcasm detection based on the experimental results obtained.

   Algorithms Used: Bag of Words, TF-IDF, Word2Vec, Glove

2. Context-Based Feature Technique for Sarcasm Identification in Benchmark Datasets Using Deep Learning and BERT Model

   This study has focused on the context-based feature technique for sarcasm identification using deep learning, transformer learning, and conventional machine learning models. Two datasets were utilised for classification using the three learning models. The first model uses embedding-based representation via deep learning model with bidirectional long short- term memory (Bi-LSTM), a variant of recurrent neural network (RNN), by applying Global vector representation (GloVe) for the construction of word embedding and context learning. The second model is based on Transformer using a pre-trained Bidirectional Encoder representation and Transformer (BERT).

3. Sarcasm Detection Using Soft Attention-Based Bidirectional Long Short-Term Memory Model with Convolution Network

   The proposed deep learning model uses 8 layers: the input layer, embedding layer, BLSTM layer, attention layer, convolution layer, activation and ReLU layer, max pooling layer, and representation layer. To build word embeddings, GloVe, which generates a word vector table, is used.

4. CASCADE: Contextual Sarcasm Detection in Online Discussion Forums

   This paper is about sarcasm detection which focuses on semantic levels of text. Thus, a hybrid approach of both content and context driven sarcasm detection was proposed and its contextual information was obtained. Here stylometric word embedding along with content-based CNN were used to make significant classification.

5. Sarcasm detection in mash-up language using soft-attention based bi-directional LSTM and feature-rich CNN

The proposed model is a hybrid of bidirectional long short-term memory with a softmax attention layer and convolution neural network for real-time sarcasm detection. To evaluate the performance of the proposed model, real-time mash-up tweets are extracted on the trending political and entertainment posts on Twitter.

Performance analysis is done to compare and validate the proposed softAttBiLSTM-feature−richCNN model. The model gives an accuracy of 92.71%

NOVELTY:

The word embedding approach used in our project is GloVe and fastText. fastText word embedding, which yields better results than GloVe isn't explored in the above papers. In most of the research work on sarcasm detection, twitter datasets or comments on Reddit, etc. are used. In these cases, due to lack of context the results may not be 100% accurate. Our approach is implemented on a labelled news headlines dataset.

METHODOLOGY

Requirements:

This project was executed in google colab with TensorFlow version 2.7.0. Google Colab is preferred as deep learning models as it provides a run-time fully configured for deep learning and is free of charge. In this work, we have used a news headlines dataset with 55328 records which are labeled as 1 & 0 (1-sarcastic, 0-not sarcastic). In this dataset 29,970 records are not sarcastic, i.e., labeled 0, and 25,385 records are sarcastic entries. For GloVe word embeddings, pre-trained word vectors from open-source project by Stanford. fastText word embedding is a text classification library created by Facebook. The pre-trained vectors were taken from Facebook's Research Repository.

Proposed System:

At first, we performed data pre-processing where we removed unnecessary noises from the data. The main libraries used to execute this task are Natural Language Toolkit (NLTK-to remove stop words and perform lemmatization) and Regular Expression (re). We have worked on 2-word embedding techniques namely fastText and GloVe word embedding to create the embedding matrix which is considered as the weights for the embedding layer.

The architecture techniques are implemented namely, CNN+Bi-LSTM, GRU + BI-LSTM and Bi-LSTM are used on the embedding layer for training. The final layer is used for classification and testing. Keras API is employed to execute these techniques. For performance analysis, we have used ReLu, LeakyReLu, Exponential linear unit & tanH activation functions and Adam & AdaGrad Optimizers.

Modules involved in the system are:

- **Data Pre-processing:**
    - In the news headlines dataset, unnecessary punctuations, URLS, numeric values, symbols, spaces are removed. Then the words are converted to lowercase.
    - Stop words are removed. Tokenization is performed to obtain the root word by stemming (WordNetLemmatizer).
    - The libraries used to execute the above-mentioned operations are Regular Expression and Natural Language Toolkit.

- **Word Embedding:**
    - We implemented 2-word embedding techniques namely, fastText and GloVe (Global Vector for Word Representation).
    - **fastText:**

      FastText is a library created by the Facebook Research Team for efficient learning of word representations and sentence classification. This model allows the creation of unsupervised learning or supervised learning algorithms for obtaining vector representations for words. Embedding space of 300 dimensions is used.
    - **GloVe:**

      The GloVe is an unsupervised learning technique that generates word vector representations. The resulting representations highlight intriguing linear substructures of the word vector space, and training is based on aggregated global word-word co-occurrence statistics from a corpus. Embedding space of 100 dimensions is used.
    - The vocabulary size is 21164 and the input sequence length is 30.

- **Deep Learning Model:**

  Here we have implemented 3 architectures for 2-word embedding techniques.

  First, we create an embedding layer using the vocabulary size, embedding dimension, embedding matrix as weights, and input maximum sequence length.

  - **CNN + Bi-LSTM:**

    After forming the embedding layer, we create a spatial dropout-1dim of 20 percent where it drops entire 1D feature maps instead of individual elements.

    Then we create a 1-dimensional convolution layer followed by Bi-directional LSTM and finally, we use the sigmoid activation layer to decide whether it's sarcastic or not.

  - **GRU + BI-LSTM**

    After forming the embedding layer, a GRU layer of 32 neurons with a dropout

    and recurrent dropout of 20 percent. Then we form a Bi-LSTM layer of 64 neurons and a hidden layer activation function is used. Then finally we use a dropout of 20 percent followed by sigmoid activation to decide whether sarcastic or not.

  - **BI-LSTM using Glorot uniform weight initializer**

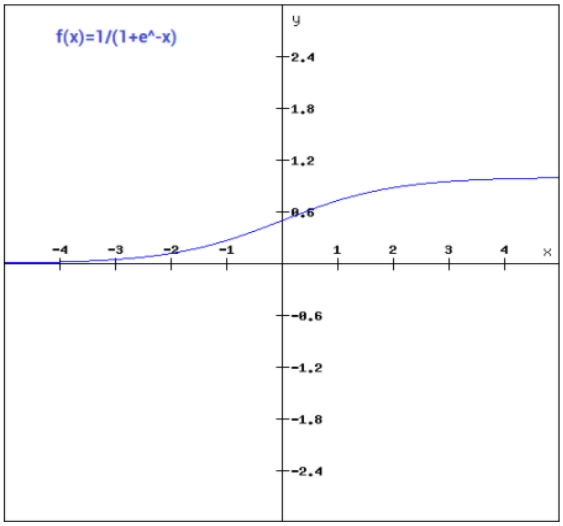
    After forming the embedding layer, we create a spatial dropout-1dim of 20 percent where it drops entire 1D feature maps instead of individual elements. Then we form a BI-LSTM layer with 128 units with a dropout and recurrent dropout of 20 percent. Then we create

a dense layer of 512 units with a glorious uniform weight initializer and activation function. Then finally we use a sigmoid activation to decide whether sarcastic or not.

As mentioned above, we have used activation functions like sigmoid, ReLu, LeakyReLu, Exponential linear unit & tanh.

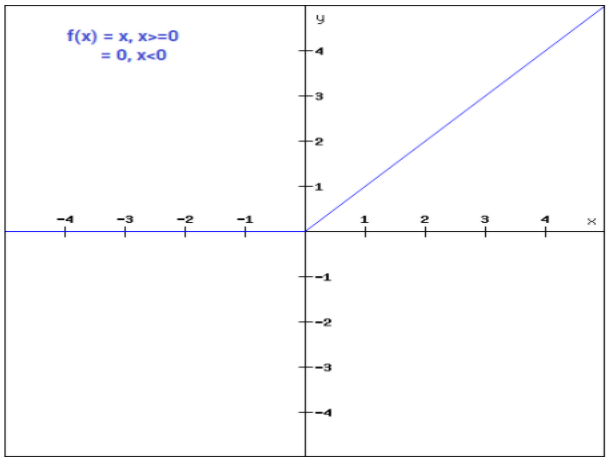
Sigmoid:

Sigmoid function. It is one of the most widely used non-linear activation function. Sigmoid transforms the values between the range 0 and 1.



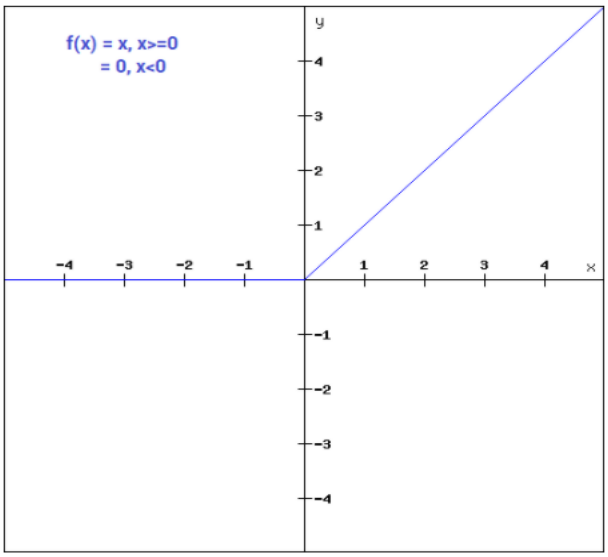
ReLu:

The ReLU function is another non-linear activation function that has gained popularity in the deep learning domain. ReLU stands for Rectified Linear Unit. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.



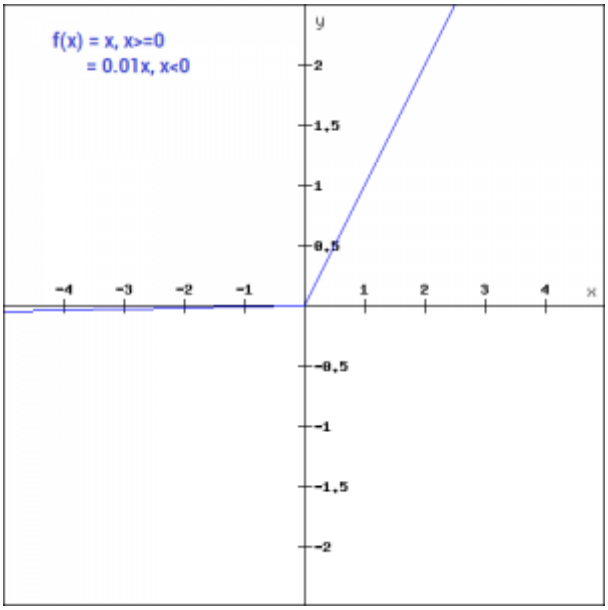
Tanh:

The tanh function is very similar to the sigmoid function. The only difference is that it is symmetric around the origin. The range of values in this case is from -1 to 1. Thus, the inputs to the next layers will not always be of the same sign.



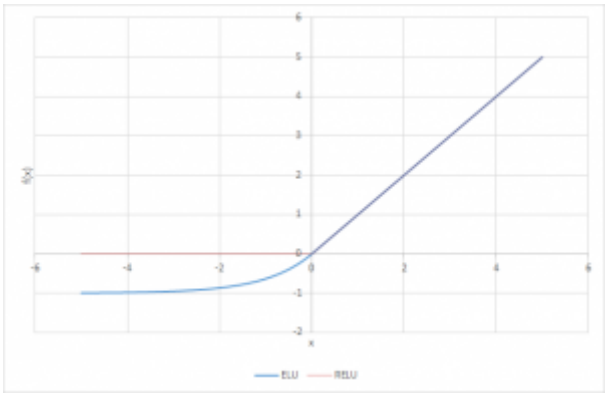
Leaky ReLU:

Leaky ReLU function is nothing but an improved version of the ReLU function. For the ReLU function, the gradient is 0 for x<0, which would deactivate the neurons in that region. Instead of defining the Relu function as 0 for negative values of x, we define it as an extremely small linear component of x.



Exponential Linear Unit:

Exponential Linear Unit or ELU for short is also a variant of Rectified Linear Unit (ReLU) that modifies the slope of the negative part of the function. Unlike the leaky relu and parametric ReLU functions, instead of a straight line, ELU uses a log curve for defining the negative values.



Weight initialisation technique glorot is used. Weight initialization is an important design choice when developing deep learning neural network models.

Adagrad Optimizer:

AdaGrad or adaptive gradient allows the learning rate to adapt based on parameters. It performs larger updates for infrequent parameters and smaller updates for frequent one. Because of this it is well suited for sparse data (NLP or image recognition)

Adam Optimizer:

In Adam instead of adapting learning rates based on the average first moment as in RMSP, Adam makes use of the average of the second moments of the gradients. Adam. This algorithm calculates the exponential moving average of gradients and square gradients. And the parameters of β1 and β2 are used to control the decay rates of these moving averages. Adam is a combination of two gradient descent methods, Momentum, and RMSP

RESULTS AND DISCUSSION

The model was trained at 50 epochs at a learning rate of 0.0001, Adam and Adagrad optimizers were used for all the models. The train-test-validation data split is 60%, 20% and 20% respectively.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df_preprocess['Sarcasm'], df_preprocess['target'], train_size = 0.6,test_size=0.4, random_state=8)
X_test,X_val,y_test,y_val = train_test_split(X_test,y_test, test_size=0.5, random_state=123)
print("Train Data size:", len(X_train))
print("Test Data size", len(X_test))
print("val Data size", len(X_val))
```

```
Train Data size: 33196
Test Data size 11066
val Data size 11066
```

Comparison table:

Word Embedding	Architecture		Activation Function	Optimizer	Accuracy Score	Loss
GloVe	CNN+Bi-LSTM		ReLu	Adam	82.68%	39.26%
			LeakyRelu		83.47%	38.19%
			Exponential	Adagrad	54.30%	68.97%
			tanH		52.88%	69.1%
	GRU+Bi-LSTM		ReLu	Adam	78.54%	44.67%
			LeakyRelu		77.75%	45.78%
			Exponential	Adagrad	54.23%	69%
			tanH		53.38%	69.16%
	BI-LSTM-Glorot weight initializer		ReLu	Adam	81.66%	39.90%
			LeakyRelu		83.11%	38.35%
			Exponential	Adagrad	54.60%	68.85%
			tanH		53.88%	69.66%
fastText	CNN+Bi-LSTM		ReLu	Adam	87.70%	32%
			LeakyRelu		83.83%	37.53%
			Exponential	Adagrad	53.75%	69.02%
			tanH		54.06%	69.15%
	GRU+Bi-LSTM		ReLu	Adam	80.92%	41.85%
			LeakyRelu		77.29%	46.99%
			Exponential	Adagrad	53.60%	69.15%
			tanH		54.14%	69.05%
	BI-LSTM-Glorot weight initializer		ReLu	Adam	85.33%	35.16%
			LeakyRelu		82%	40.23%
			Exponential	Adagrad	54.29%	68.73%
			tanH		54.26%	69.03%

From the above table, we can conclude that the best model to perform sarcasm detection is fastText + CNN-Bi-LSTM with Relu and Adam optimizer which has a test accuracy of 87.70%.

In terms of word embedding, fastText works better than GloVe. Adam optimizer yields better results than AdaGrad.

CONCLUSION

Sarcasm is a specific type of sentiment and its automatic detection is a dynamic area of research. Sarcasm detection research has grown significantly. In the past few years, necessitating a look back at an overall picture that these works have led to. We do necessary pre-processing such as data cleaning, tokenization, removing stop words, performing lemmatization and make the dataset ready for feeding into the model. We test our dataset with five different architectures CNN+BILSTM, GRU+BILSTM, BILSTM (Glorot Weight initializer) for two different word embeddings namely fastText, Glove and check accuracy score, and compare the performance of each model to find out which better performs better.

REFERENCES

1. Annie Johnson, Karthik R," Performance Evaluation of Word Embeddings for Sarcasm Detection- A Deep Learning Approach", Vellore Institute of Technology, Chennai
2. Christopher ifeanyi eke ,Azah anir norman ,Liyana shuib," Context-Based Feature Technique for Sarcasm Identification in Benchmark Datasets Using Deep Learning and BERT Model",2021
3. Devamanyu Hazarika, Soujanya Poria, Sruthi Gorantla," CASCADE: Contextual Sarcasm Detection in Online Discussion Forums",2018
4. Le hoang son , Akshi kumar , Saurabh raj sangwan ," Sarcasm Detection Using Soft Attention-Based Bidirectional Long Short-Term Memory Model With Convolution Network",2019
5. Deepak Kumar Jain**,** Akshi Kumar**, "**Sarcasm detection in mash-up language using soft-attention based bi-directional LSTM and feature-rich CNN",2018

APPENDIX

Code:

Importing libraries:

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import tensorflow as tf
import sklearn
import re
import codecs
from tqdm import tqdm
from keras.preprocessing.text import Tokenizer
from tensorflow.keras.layers import Conv1D, Bidirectional, LSTM, Dense, Input, Dro
pout, SpatialDropout1D, GlobalMaxPooling1D
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
from tensorflow.keras.layers import RNN
import codecs   #Fastext
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from keras.preprocessing.sequence import pad_sequences
from google.colab import drive
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
```

Preprocessing:

```
def clean_text (text):
    text = text.lower ()
    text = re.sub (r"[^  sa-zA-Z0-
9 @  []]", '', text) # Removes punctuation #removes character for bracket one item
    text = re.sub (r"w *  d +  w *", '', text) # Remove digits
    text = re.sub (' s {2,}', "", text) # Removes unnecessary spaces
    return text
sarc_clean = []
```

```
for t in sarc:
    sarc_clean.append(clean_text(t))
corpus = []
for text in sarc_clean:
    corpus.append(text.replace("'","").replace(",","").replace("[","").replace("]"
,""))
```

Tokenization

```
def tokenize(texts):
    tokenizer = nltk.RegexpTokenizer(r'\w+')

    texts_tokens = []
    for i, val in enumerate(texts):
        text_tokens = tokenizer.tokenize(val.lower())

        for i in range(len(text_tokens) - 1, -1, -1): #tokens in reverse order
            if len(text_tokens[i]) < 4:
                del(text_tokens[i])

        texts_tokens.append(text_tokens)

    return texts_tokens
```

Removing Stop Words

```
def removeSW(texts_tokens):
    stopWords = set(stopwords.words('english'))
    texts_filtered = []

    for i, val in enumerate(texts_tokens):
        text_filtered = []
        for w in val:
            if w not in stopWords:
                text_filtered.append(w)
        texts_filtered.append(text_filtered)

    return texts_filtered
```

Lemmatization

```
def lemma(texts_filtered):
    wordnet_lemmatizer = WordNetLemmatizer()
    texts_lem = []

    for i, val in enumerate(texts_filtered):
        text_lem = []
        for word in val:
            text_lem.append(wordnet_lemmatizer.lemmatize(word, pos="v")) #verb par
ts of speech
```

```
        texts_lem.append(text_lem)

    return texts_lem
```

Extracting Fasttext Vectors:

```
embeddings_index = {}
f = codecs.open('/content/gdrive/MyDrive/wiki.simple.vec', encoding='utf-8')
for line in tqdm(f):
    values = line.rstrip().rsplit(' ')
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('found %s word vectors' % len(embeddings_index))
```

Extracting GloVe vectors

```
import codecs   #GloVe
embeddings_index = {}
f =codecs.open('/content/gdrive/MyDrive/glove.6B.100d.txt', encoding='utf-8')
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Found %s word vectors.' % len(embeddings_index))
```

Vocabulary Size:

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(df_preprocess.Sarcasm)

word_index = tokenizer.word_index
vocabulary_size = len(tokenizer.word_index) + 1
print("Vocabulary Size :", vocabulary_size)
```

Embedding Matrix:

```
embedding_matrix = np.zeros((vocabulary_size, EMBEDDING_DIM))
for word, i in word_index.items():
  embedding_vector = embeddings_index.get(word)
  if embedding_vector is not None:
    embedding_matrix[i] = embedding_vector
embedding_matrix
BATCH_SIZE = 1024
EPOCHS = 100
EMBEDDING_DIM = 300 (100 for GloVe)
MAX_SEQUENCE_LENGTH = 30
```

Forming Embedding Layer:

```
embedding_layer = tf.keras.layers.Embedding(vocabulary_size,
                                            EMBEDDING_DIM,
                                            weights=[embedding_matrix],
                                            input_length=MAX_SEQUENCE_LENGTH,
                                            trainable=False)
```

Test-Train Split:

```
X_train, X_test, y_train, y_test = train_test_split(df_preprocess['Sarcasm'], df_p
reprocess['target'], train_size = 0.6,test_size=0.4, random_state=8)
X_test,X_val,y_test,y_val = train_test_split(X_test,y_test, test_size=0.5, random_
state=123)
print("Train Data size:", len(X_train))
print("Test Data size", len(X_test))
print("val Data size", len(X_val))

BATCH_SIZE = 1024
EPOCHS = 100

X_train = pad_sequences(tokenizer.texts_to_sequences(X_train),
                        maxlen = MAX_SEQUENCE_LENGTH)
X_test = pad_sequences(tokenizer.texts_to_sequences(X_test),
                       maxlen = MAX_SEQUENCE_LENGTH)
X_val = pad_sequences(tokenizer.texts_to_sequences(X_val),
                       maxlen = MAX_SEQUENCE_LENGTH)
print("Training X Shape:",X_train.shape)
print("Testing X Shape:",X_test.shape)
print("Val X Shape:",X_val.shape)
```

ARCHITECTURES:

CNN+Bi-LSTM

```
inputs = tf.keras.Input(shape=(MAX_SEQUENCE_LENGTH,), dtype="int32")
x = embedding_layer(inputs)
x = SpatialDropout1D(0.2)(x)
# Conv1D + LSTM (bidirectional)
x = Conv1D(64, 5, activation='relu')(x)
x = Bidirectional(LSTM(64, dropout=0.2, recurrent_dropout=0.2))(x)
x = Dense(512, activation="relu")(x)
x = Dropout(0.5)(x)
outputs = Dense(1, activation="sigmoid")(x)
model_B = tf.keras.Model(inputs, outputs)
model_B.compile(optimizer=Adam(0.0001), loss='binary_crossentropy',
              metrics=['accuracy'])

es=EarlyStopping(monitor='val_loss',
                 mode='min',
                 verbose=1,
                 patience=5)
```

```
reduce_lr = ReduceLROnPlateau(factor=0.1,min_lr = 0.02, monitor = 'val_loss',

                                    verbose = 1)

history_B = model_B.fit(X_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
                    validation_data=(X_val, y_val), callbacks=[es, reduce_lr])
```

GRU+ Bi-LSTM:

```
IN = tf.keras.Input(shape=(MAX_SEQUENCE_LENGTH,), dtype="int32")

x = embedding_layer(IN)
x = tf.keras.layers.GRU(32, dropout=0.2, recurrent_dropout=0.2, return_sequences=T
rue)(x)

x = Bidirectional(LSTM(64, dropout=0.2, recurrent_dropout=0.2))(x)
x = Dense(256, activation="relu")(x)
x = Dropout(0.2)(x)

OUT = Dense(1, activation="sigmoid")(x)

model_E = tf.keras.Model(IN, OUT)

model_E.compile(optimizer=Adam(0.0001), loss='binary_crossentropy',
              metrics=['accuracy']) #Adagrad optimizer
history_E = model_E.fit(X_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS,
                    validation_data=(X_val,y_val), callbacks=[es, reduce_lr])
```

BI-LSTM using Glorot uniform weight initializer:

```
inner = tf.keras.Input(shape=(MAX_SEQUENCE_LENGTH,), dtype="int32")

x = embedding_layer(inner)
x = SpatialDropout1D(0.2)(x)

# Conv1D + LSTM (bidirectional)
x = Bidirectional(LSTM(128, dropout=0.2, recurrent_dropout=0.2))(x)

x = Dense(512, kernel_initializer = 'glorot_uniform',activation="relu")(x)
x = Dropout(0.5)(x)

outer = Dense(1, activation="sigmoid")(x)

model_F = tf.keras.Model(inner, outer)

model_F.compile(optimizer=Adam(0.0001), loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
history_F = model_F.fit(X_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, va
lidation_data=(X_val, y_val), callbacks=[es, reduce_lr])
```

Test Accuracy and Loss Score:

```
score = model_B.evaluate(X_test, y_test, verbose = 0)
y_pred = model_B.predict(X_test)
print('Test loss:', score[0]*100,'%')
print('Test accuracy:', score[1]*100,'%')
```

Plot Commands:

```
plt.plot(history_F.history['accuracy'])
plt.plot(history_F.history['val_accuracy'])
plt.ylabel('Accuracy')
plt.title('BI-lstm with word embedding fasttext')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
```

```
plt.plot(history_F.history['loss'])
plt.plot(history_F.history['val_loss'])
plt.ylabel('loss')
plt.title('BI-lstm with word embedding fasttext')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
```