ST. XAVIER'S COLLEGE [AUTONOMOUS], KOLKATA

COMPARATIVE STUDY OF MACHINE LEARNING MODEL PERFORMANCE FOR FAKE NEWS DETECTION

By

Kingshuk Mukherjee (544)

Priyadarshi Roy (545)

Aritro Rakshit (556)

Semester IV

Under Guidance of

Prof. Sonali Sen

M.Sc. Department of Computer Science

2019-2021

TABLE OF CONTENTS

INTRODUCTION

Machine learning has become a mainstay in the field of information technology [1]. It is being increasingly used for a variety of tasks that involve prediction and classification. Given the tremendous scope of application of this new field, it is only natural that this area is undergoing lots of research and development.

An important area in which machine learning has found widespread application is text classification. In this project, we have made a comparative study of some of the widely used machine learning models based on their text classification performance. All of the models used follow the principle of supervised learning, as it is one of the most successful approaches of machine learning [2].

Text Classification

Text classification problems have been widely studied and addressed in many real applications [3] over the last few decades. Especially with recent breakthroughs in Natural Language Processing (NLP) and text mining, many researchers are now interested in developing applications that leverage text classification methods. Most text classification and document categorization systems can be deconstructed into the following four phases: Feature extraction, dimension reductions, classifier selection, and evaluations. [4]

Why we chose fake news

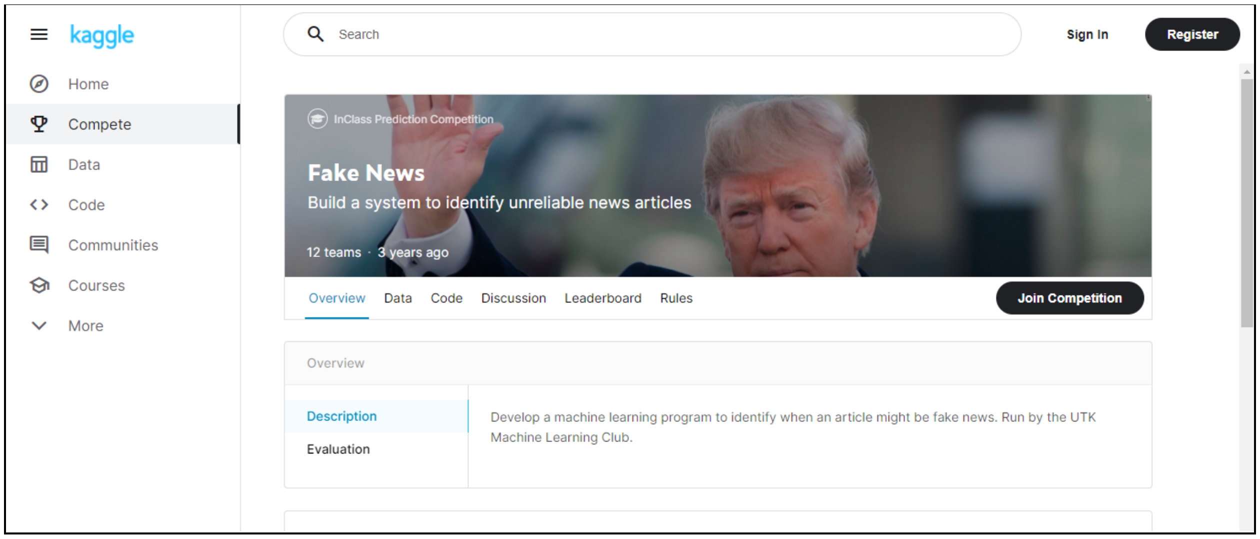
In this project, we have taken the example of fake news as text data. We could have chosen any other sort of text data like movie reviews. We focus on fake news because it is increasingly becoming a menace to our society, disrupting social harmony and sowing misinformation. We have judged attempts to stop or limit the spread of fake news as an utmost need.

But at the same time, it should be kept in mind that this project could have been carried out by considering any type of text data. We have not considered the fake-news specific features like source of news, date of publication etc. The main thrust of the project is to compare different text classification models, and to that end, we have considered the problem of fake news from a purely NLP-centric viewpoint.

An important point: "*Truth is always relative*", i.e., there is no absolute truth. We have to assume some news sources as true as we build the models. The datasets we have used have some data points marked as true and some false. We have assumed the datasets to be true in their judgment.

FAKE NEWS DATA SOURCES

The data used for this project was drawn from multiple online datasets, all in public domain. Seven different datasets were used. Six of the datasets were from Kaggle [5][6][7][8][9][10]. One dataset was downloaded from the ISOT Lab Datasets (University of Victoria) [11].



Develop a machine learning program to identify when an article might be fake news. Run by the UTK Machine Learning Club.

Fig 1. Dataset 1

TEXT PREPROCESSING

Data from the datasets cannot be fed directly into ML models. They require extensive study and thereupon, cleaning and modifying to remove irregularities reduce noise and decrease variance. Text cleaning and preprocessing is perhaps the most extensive and crucial step in any text related ML application or project.

There exist a whole suite of methods to clean and process text data. Some of these methods were implemented on each of the individual datasets before they were merged.

1. **Url removal**: Using regular expressions were removed from the articles.
2. **Lowercasing**: The text was converted into lowercase.
3. **Removal of all non alphabetic characters**: All non-alphabetic characters were stripped away from the data.
4. **Stopword removal**: Stop words are a set of commonly used words in a language. Examples of stop words in English are “a”, “the”, “is”, “are” and etc. The intuition behind using stop words is that, by removing low information words from text, we can focus on the important words instead. The NLTK library was used, specifically its premade collection of stopwords [12]. This collection was downloaded and all these stopwords were removed from the entire text corpus.

MERGED DATASET

Each of the datasets were separately preprocessed and then the preprocessed data files were merged to form a single dataset. Binary 0 and 1 labels were assigned to the real/fake news articles, respectively. This merged data was trimmed and further modified until it contained true and fake news in the ratio 1:1.

The resultant merged dataset contained a total of 1,20,000 articles which was used for training all the models.

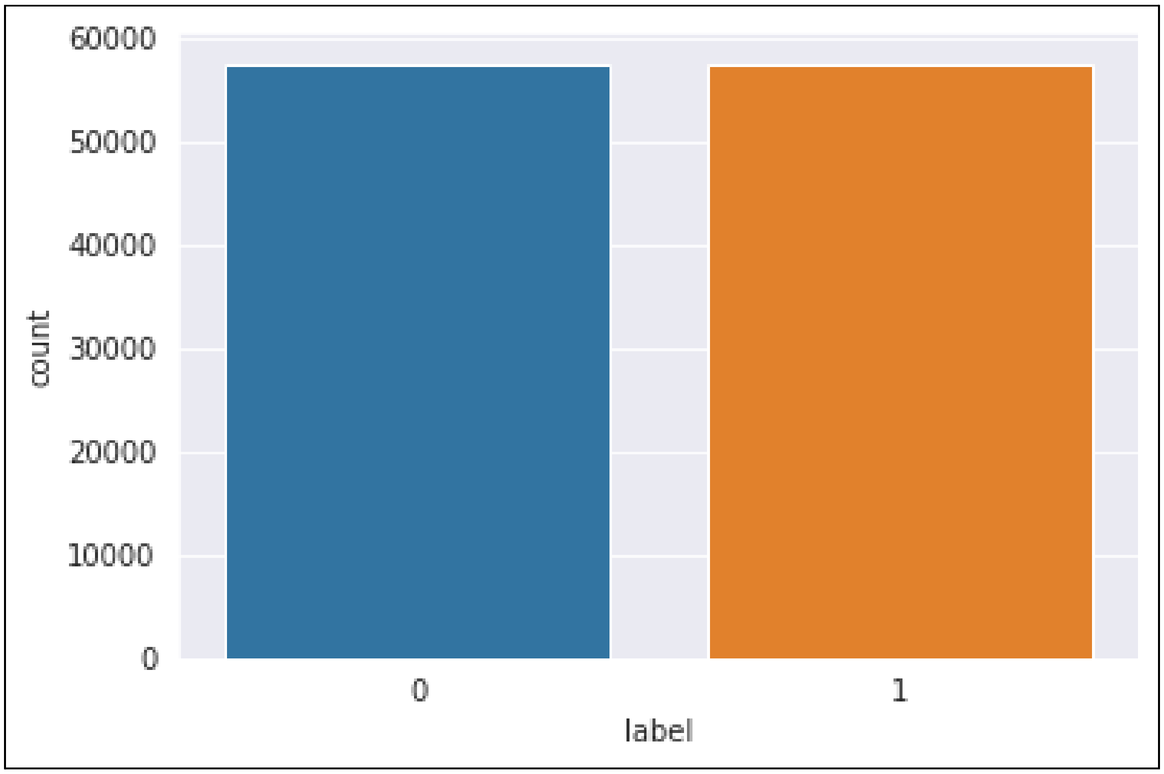
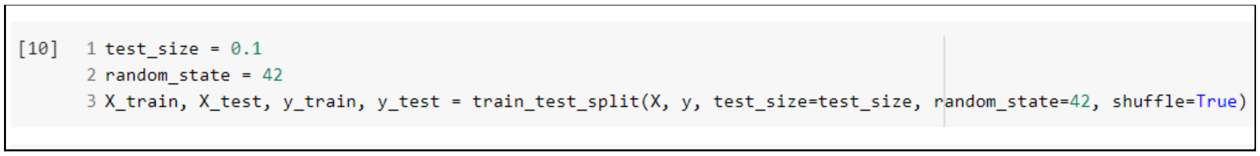


Fig 2. Distribution of true and fake news in merged dataset

PREPARING THE DATA FOR TRAINING

The entire dataset was split into training and validation datasets.

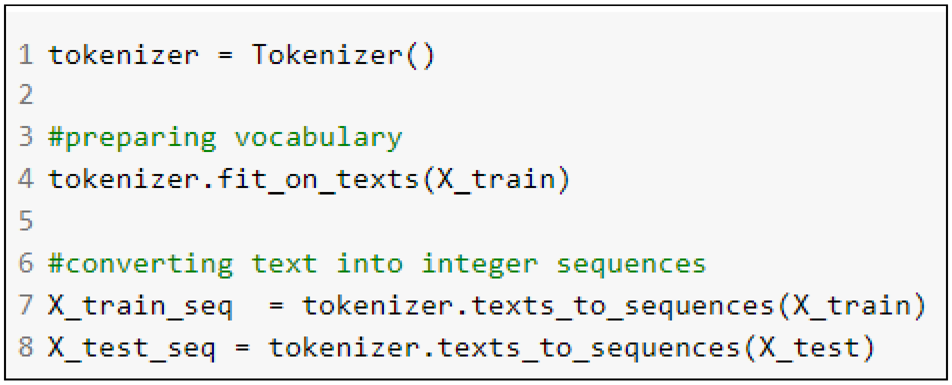


```
[10]  test_size = 0.1
      random_state = 42
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=42, shuffle=True)
```

Fig 3. Splitting dataset into train and validation data

Since most of the statistical algorithms, e.g machine learning and deep learning techniques, work with numeric data, therefore we have to convert text into numbers. Several approaches exist in this regard. However, the most famous ones are Bag of Words, TF-IDF, and word-embeddings. The Keras Tokenizer API [13] was used to perform the conversion of text corpus to stream of tokens and thereupon encoding into integer sequences.

The main steps taken to prepare the data are shown in code:



```
tokenizer = Tokenizer()

#preparing vocabulary
tokenizer.fit_on_texts(X_train)

#converting text into integer sequences
X_train_seq  = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)
```

Fig 4. Data preparation 1

Step 1: The Keras Tokenizer API was used to convert the entire training text corpus into a stream of tokens.

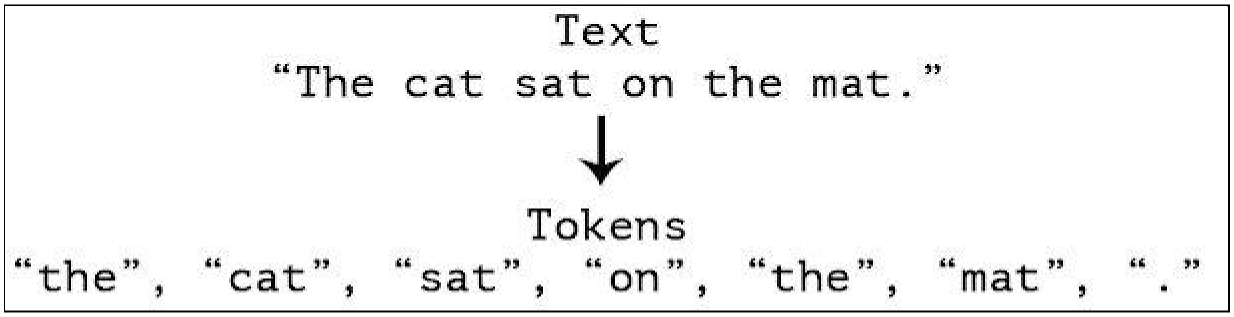
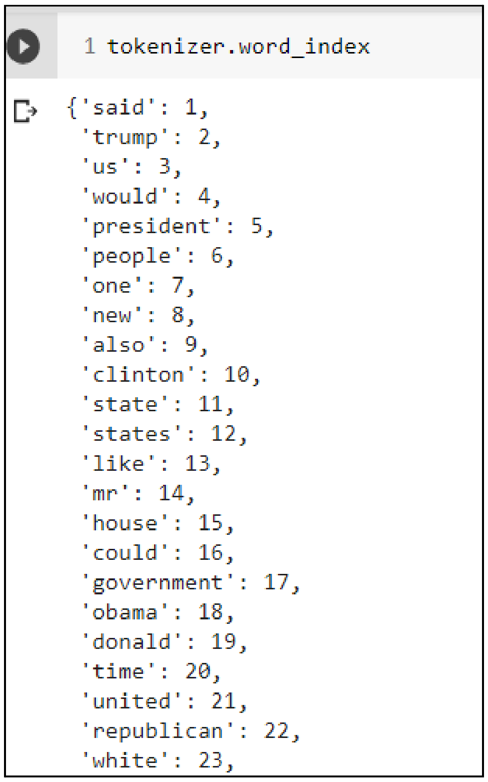

Text
“The cat sat on the mat.”
Tokens
“the”, “cat”, “sat”, “on”, “the”, “mat”, “.”

Fig 5. Example of tokenization

A byproduct of the tokenization process is the creation of a word index, which maps words in our vocabulary to their numeric representation, a mapping which will be essential for encoding our sequences.



```
tokenizer.word_index
```

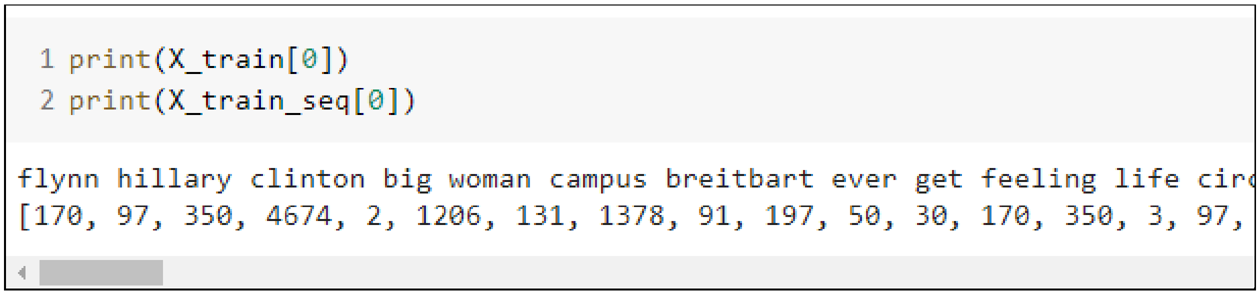
```
{'said': 1,
 'trump': 2,
 'us': 3,
 'would': 4,
 'president': 5,
 'people': 6,
 'one': 7,
 'new': 8,
 'also': 9,
 'clinton': 10,
 'state': 11,
 'states': 12,
 'like': 13,
 'mr': 14,
 'house': 15,
 'could': 16,
 'government': 17,
 'obama': 18,
 'donald': 19,
 'time': 20,
 'united': 21,
 'republican': 22,
 'white': 23,
```

Fig 6. Word index after tokenizing training dataset

Step 2: Now that we have tokenized our data and have a word to numeric representation mapping of our vocabulary, let's use it to encode our sequences. Here, we are converting our text sentences from something like "My name is Matthew," to something like "6 8 2 19," where each of those numbers match up in the index to the corresponding words. Since machine learning models work by performing computation on numbers, passing in a bunch of words won't work. Hence, sequences.

Based on the vocabulary generated from the training dataset during tokenization, both the training and validation datasets were converted into *sequences of integers*.

The **tokenizer.text_to_sequences()** function performs this conversion. The results look something like:



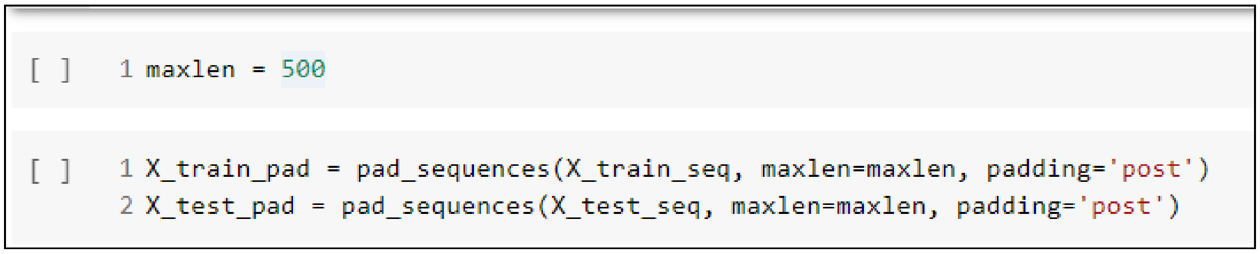
```
print(X_train[0])
print(X_train_seq[0])

flynn hillary clinton big woman campus breitbart ever get feeling life circ
[170, 97, 350, 4674, 2, 1206, 131, 1378, 91, 197, 50, 30, 170, 350, 3, 97,
```

Fig 7. Text corpus and corresponding integer sequence

The first line represents a corpus of text in the training dataset and the second line is the corresponding integer sequence generated.

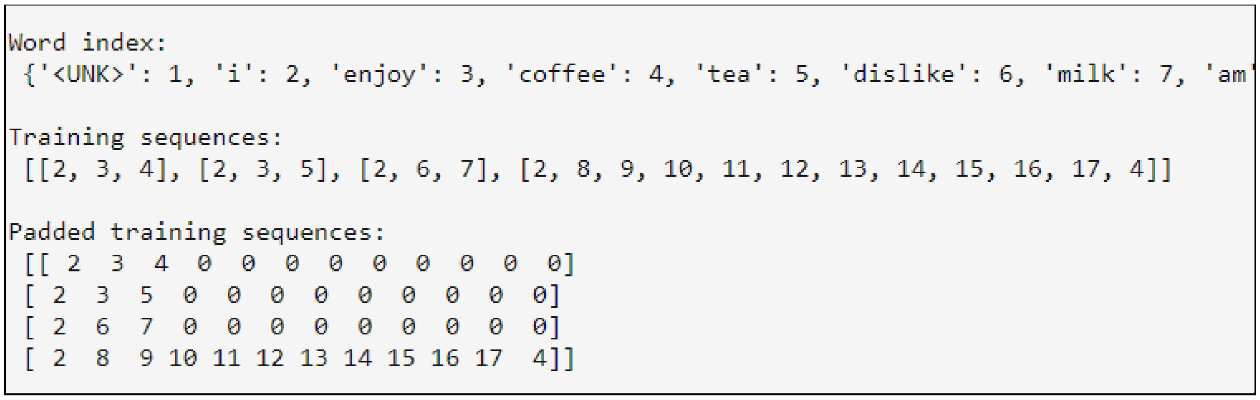
Step 3: For feeding into a ML model these sequences of integers need to be of the same length (input shape). A fixed maximum length value was used to pad all sequences with extra '0's at the end ('post') and also truncate any sequences longer than maximum length from the end ('post') as well. Here we use the TensorFlow (Keras) pad_sequences [14] module to accomplish this.



```
[ ]   maxlen = 500

[ ]   X_train_pad = pad_sequences(X_train_seq, maxlen=maxlen, padding='post')
      X_test_pad = pad_sequences(X_test_seq, maxlen=maxlen, padding='post')
```

Fig 8. Padding the integer sequences



```
Word index:
 {'<UNK>': 1, 'i': 2, 'enjoy': 3, 'coffee': 4, 'tea': 5, 'dislike': 6, 'milk': 7, 'am'

Training sequences:
 [[2, 3, 4], [2, 3, 5], [2, 6, 7], [2, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 4]]

Padded training sequences:
 [[ 2  3  4  0  0  0  0  0  0  0  0  0]
 [ 2  3  5  0  0  0  0  0  0  0  0  0]
 [ 2  6  7  0  0  0  0  0  0  0  0  0]
 [ 2  8  9 10 11 12 13 14 15 16 17  4]]
```

Fig 9. An example of tokenization and padding

To be noted: *Different maxlen values were used for padding, before feeding into different ML models.*

WORD EMBEDDINGS

A word embedding is a *learned representation* for text where words that have the same meaning have a similar representation.

It is this approach to representing words and documents that may be considered one of the key breakthroughs of deep learning on challenging natural language processing problems.

Word embeddings are in fact a class of techniques where individual words are represented as *real-valued vectors in a predefined vector space*. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network, and hence the technique is often lumped into the field of deep learning.

Key to the approach is the idea of using a dense distributed representation for each word. Each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands or millions of dimensions required for sparse word representations, such as a one-hot encoding. The distributed representation is learned based on the usage of words. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning. This can be contrasted with the crisp but fragile representation in a bag of words model where, unless explicitly managed, different words have different representations, regardless of how they are used.

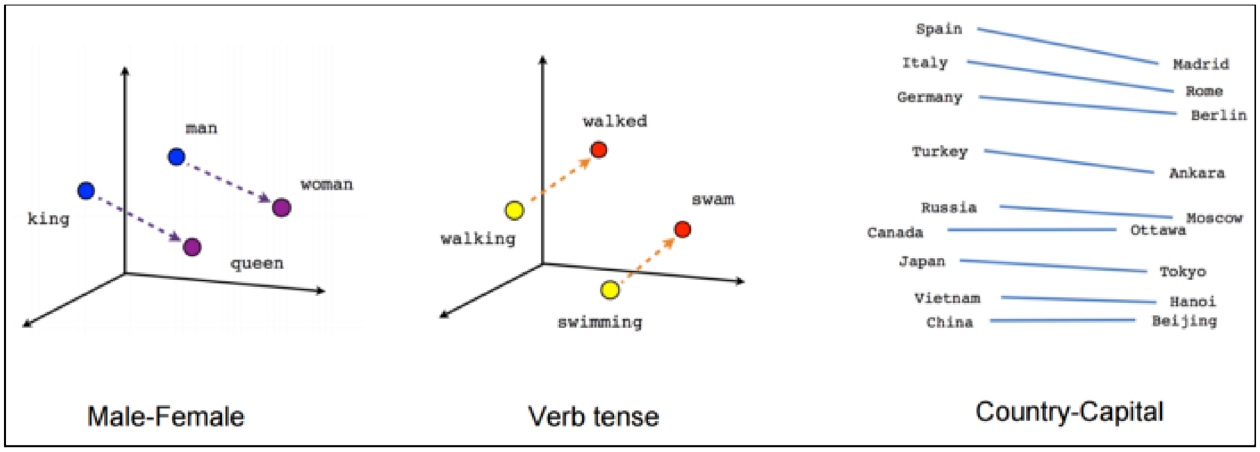


Fig 10. Word Vectors example

What are word embeddings trying to say?

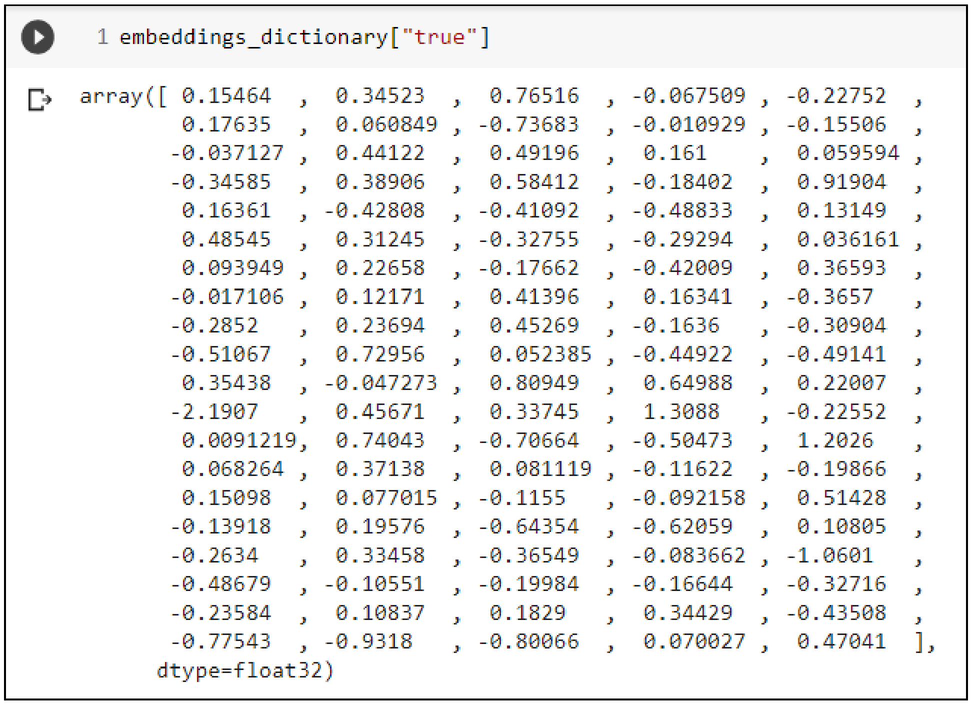
- Geometric relationship between words in a word embeddings can represent semantic relationship between words. Words closer to each other have a strong relation compared to words away from each other.

- Vectors/words closer to each other means the cosine distance or geometric distance between them is less compared to others.

- There could be a vector "male to female" which represents the relation between a words and its feminine. That vector may help us in predicting "king" when "he" is used and "Queen" when she is used in the sentence.

How do word embeddings look like?

Below is a single row of embedding matrix from GloVe representing the word '*true*' in 100 dimensions..



```
embeddings_dictionary["true"]
```

```
array([ 0.15464  ,  0.34523  ,  0.76516  , -0.067509 , -0.22752  ,
        0.17635  ,  0.060849 , -0.73683  , -0.010929 , -0.15506  ,
       -0.037127 ,  0.44122  ,  0.49196  ,  0.161    ,  0.059594 ,
       -0.34585  ,  0.38906  ,  0.58412  , -0.18402  ,  0.91904  ,
        0.16361  , -0.42808  , -0.41092  , -0.48833  ,  0.13149  ,
        0.48545  ,  0.31245  , -0.32755  , -0.29294  ,  0.036161 ,
        0.093949 ,  0.22658  , -0.17662  , -0.42009  ,  0.36593  ,
       -0.017106 ,  0.12171  ,  0.41396  ,  0.16341  , -0.3657   ,
       -0.2852   ,  0.23694  ,  0.45269  , -0.1636   , -0.30904  ,
       -0.51067  ,  0.72956  ,  0.052385 , -0.44922  , -0.49141  ,
        0.35438  , -0.047273 ,  0.80949  ,  0.64988  ,  0.22007  ,
       -2.1907   ,  0.45671  ,  0.33745  ,  1.3088   , -0.22552  ,
        0.0091219,  0.74043  , -0.70664  , -0.50473  ,  1.2026   ,
        0.068264 ,  0.37138  ,  0.081119 , -0.11622  , -0.19866  ,
        0.15098  ,  0.077015 , -0.1155   , -0.092158 ,  0.51428  ,
       -0.13918  ,  0.19576  , -0.64354  , -0.62059  ,  0.10805  ,
       -0.2634   ,  0.33458  , -0.36549  , -0.083662 , -1.0601   ,
       -0.48679  , -0.10551  , -0.19984  , -0.16644  , -0.32716  ,
       -0.23584  ,  0.10837  ,  0.1829   ,  0.34429  , -0.43508  ,
       -0.77543  , -0.9318   , -0.80066  ,  0.070027 ,  0.47041  ],
      dtype=float32)
```

Fig 11. A single row of embedding matrix representing the word '*true*' in 100 dimensions

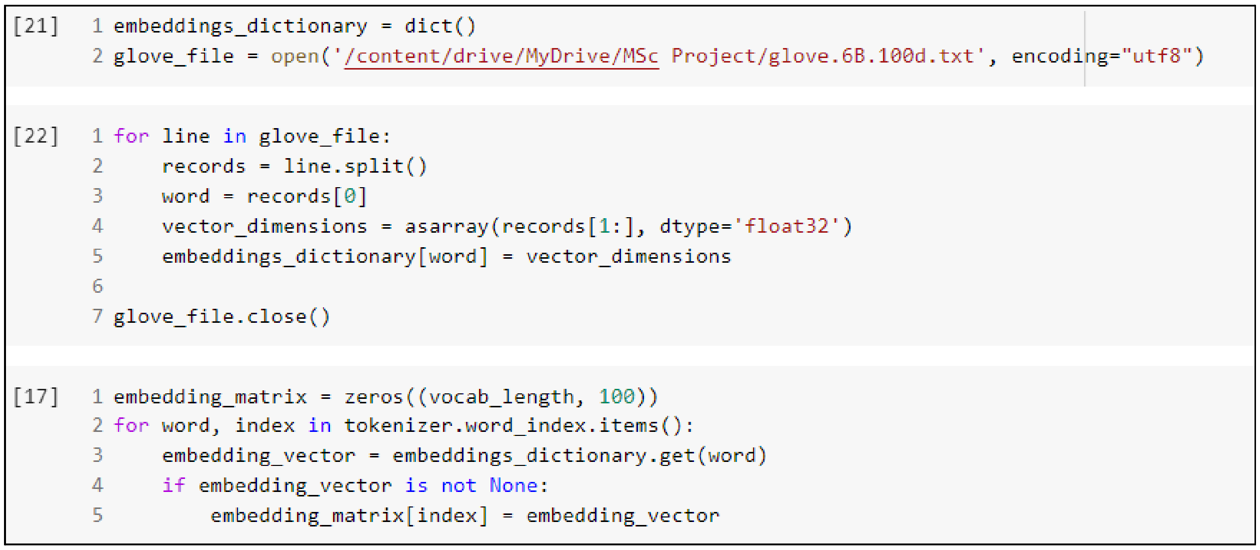
Pre-trained Embedding Matrix

The two famous pre-trained word embeddings used in machine learning are :

i) Word2Vec [15]
ii) GloVe [l6]

All models in this project use pre-trained 100 dimensional GloVe word embeddings.

Now that we know about word embeddings, we can finally get to the last step of data preparation for training.



```
[21]  embeddings_dictionary = dict()
      glove_file = open('/content/drive/MyDrive/MSc Project/glove.6B.100d.txt', encoding="utf8")

[22]  for line in glove_file:
          records = line.split()
          word = records[0]
          vector_dimensions = asarray(records[1:], dtype='float32')
          embeddings_dictionary[word] = vector_dimensions

      glove_file.close()

[17]  embedding_matrix = zeros((vocab_length, 100))
      for word, index in tokenizer.word_index.items():
          embedding_vector = embeddings_dictionary.get(word)
          if embedding_vector is not None:
              embedding_matrix[index] = embedding_vector
```

Fig 12. Loading GloVe Embeddings

Step 4:

- **Cell 1**: Loading pretrained 100 dimensional GloVe embeddings into variable *glove_file.*

- **Cell 2**: Next, we need to load the entire GloVe word embedding file into memory as a *dictionary of word to embedding array*.This is pretty slow. It might be better to filter the embedding for the unique words in the training data.

- **Cell 3**: Next, we need to create a matrix of one embedding for each word in the training dataset. We can do that by enumerating all unique words in the *Tokenizer.word_index* and locating the embedding weight vector from the loaded GloVe embedding. The result is a matrix of weights only for words we will see during training. This matrix of embeddings is fed as weights into the first *embedding layer* of each model.

MACHINE LEARNING MODELS

Several different models were implemented; the following sections describe each in detail. All models use pre-trained 100-dimensional GloVe [16] embeddings, which are not updated during training.

Logistic Regression

Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. Unlike linear regression which outputs continuous number values, logistic regression transforms its output using the logistic sigmoid function to return a probability value which can then be mapped to two or more discrete classes.

Sigmoid Function: In order to map predicted values to probabilities, we use the sigmoid function. The function maps any real value into another value between 0 and 1. In machine learning, we use sigmoid to map predictions to probabilities.

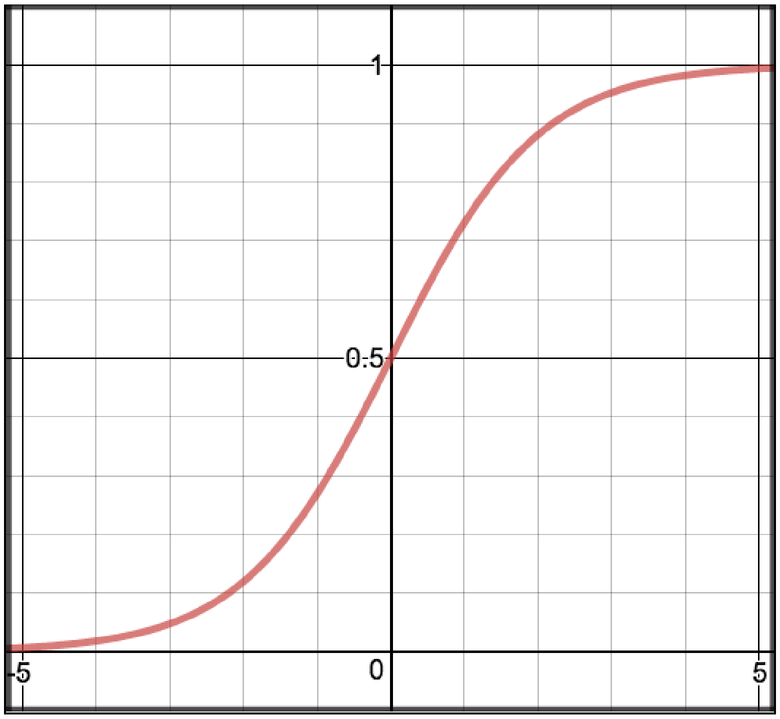


Fig 13. Sigmoid curve

This is a simple, linear model, that is implemented to serve as a baseline for comparison with other, more sophisticated models.

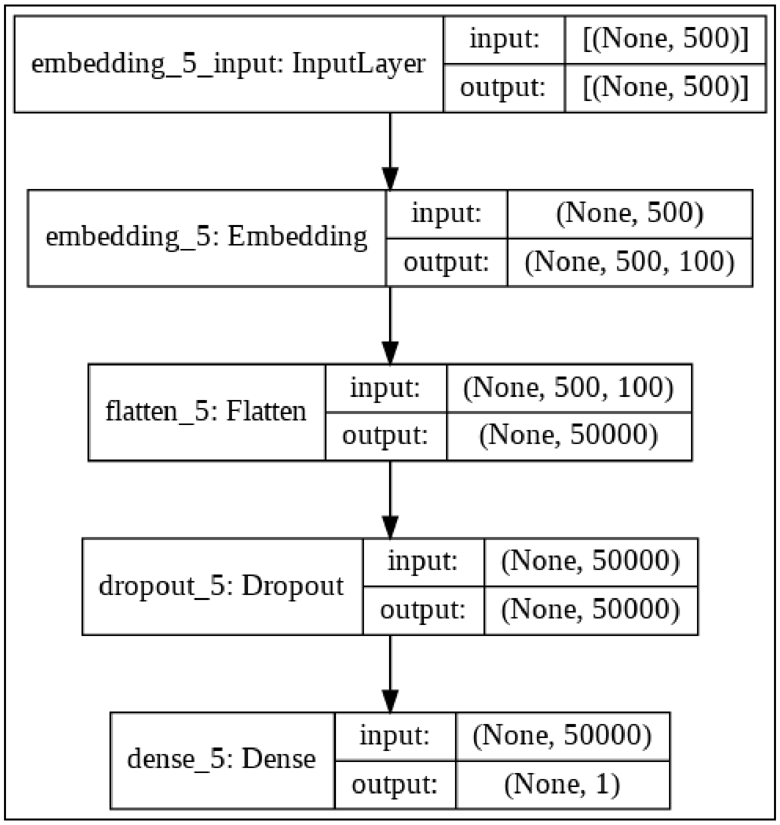


Fig 14. Logistic Regression model

Feedforward Neural Network

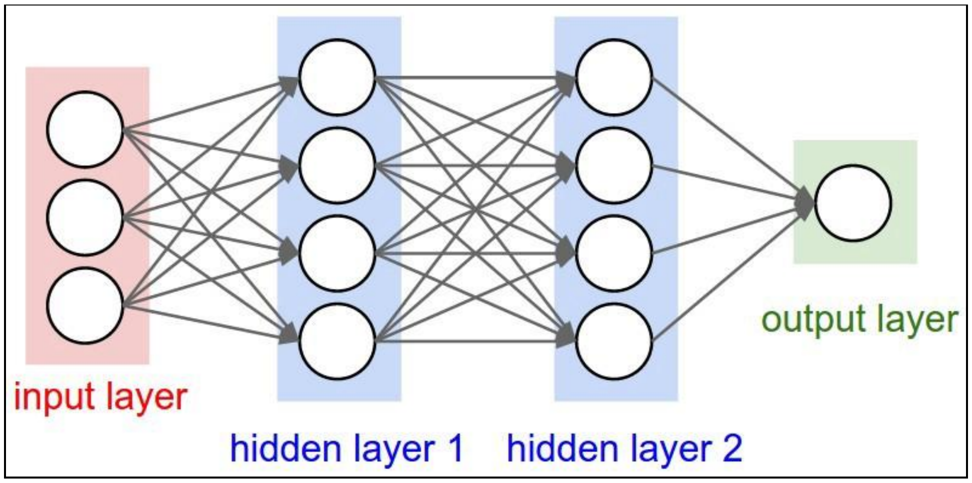


Fig 15. Neural Network

A simple 2 hidden layer Feedforward Neural Network is used as the next model. Dropout layers have been added to reduce overfitting.

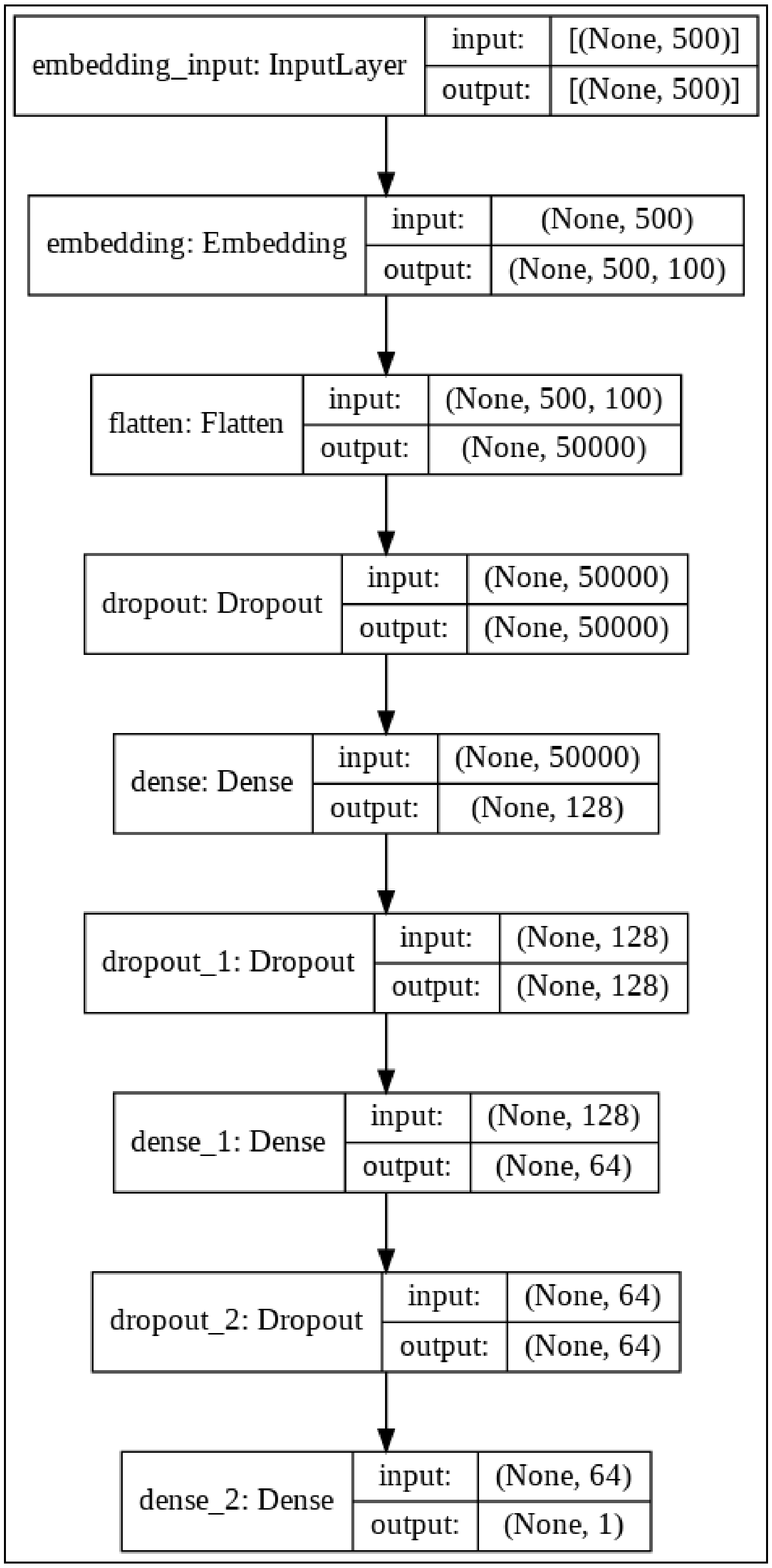


Fig 16. Neural Network model

Vanilla Recurrent Neural Network (RNN)

In a feed-forward neural network, the information only moves in one direction — from the input layer, through the hidden layers, to the output layer. The information moves straight through the network and never touches a node twice.

Feed-forward neural networks have no memory of the input they receive and are bad at predicting what's coming next. Because a feed-forward network only considers the current input, it has no notion of order in time. It simply can't remember anything about what happened in the past except its training.

In a RNN the information cycles through a loop. When it makes a decision, it considers the current input and also what it has learned from the inputs it received previously.

The two images below illustrate the difference in information flow between a RNN and a feed-forward neural network.

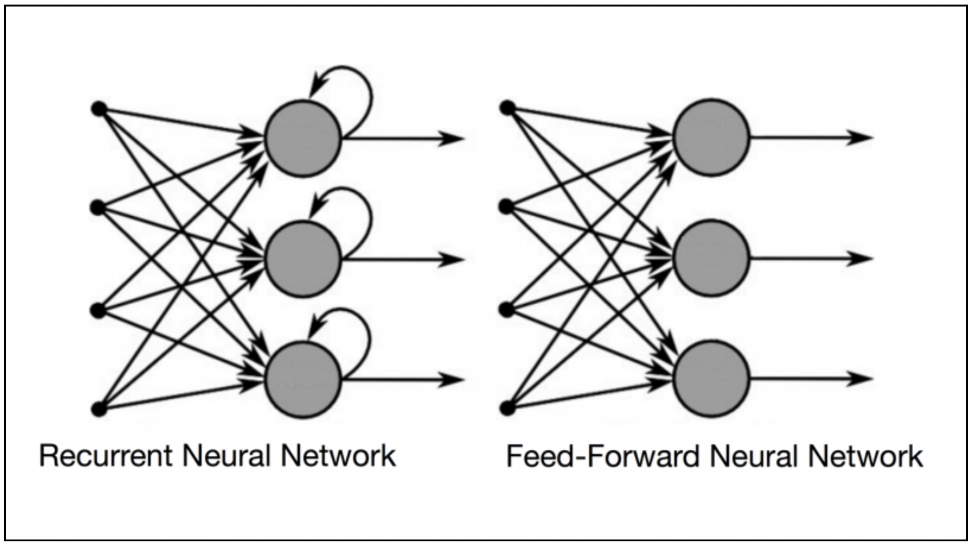


Fig 17. Recurrent NN vs Feedforward NN

A structure of a simple (Vanilla) RNN is shown:

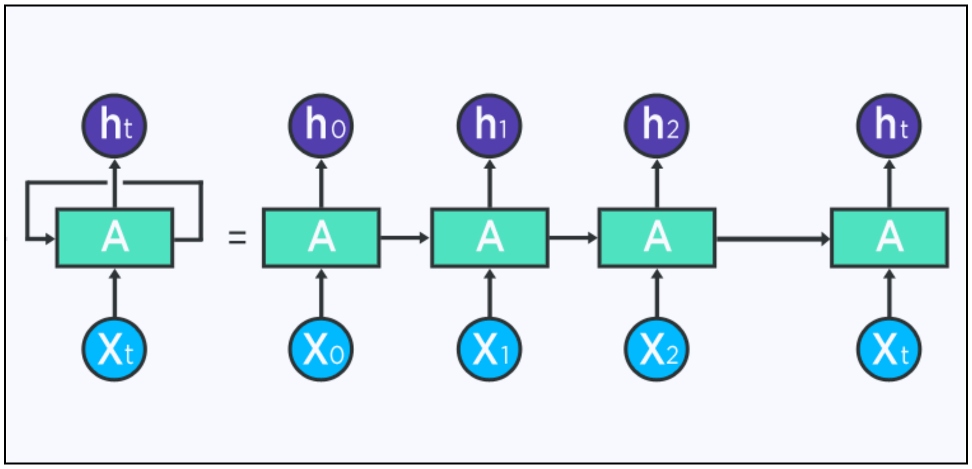


Fig 18. Vanilla RNN

A simple vanilla RNN structure was used to provide a basis for comparison for other complicated LSTM and GRU models.

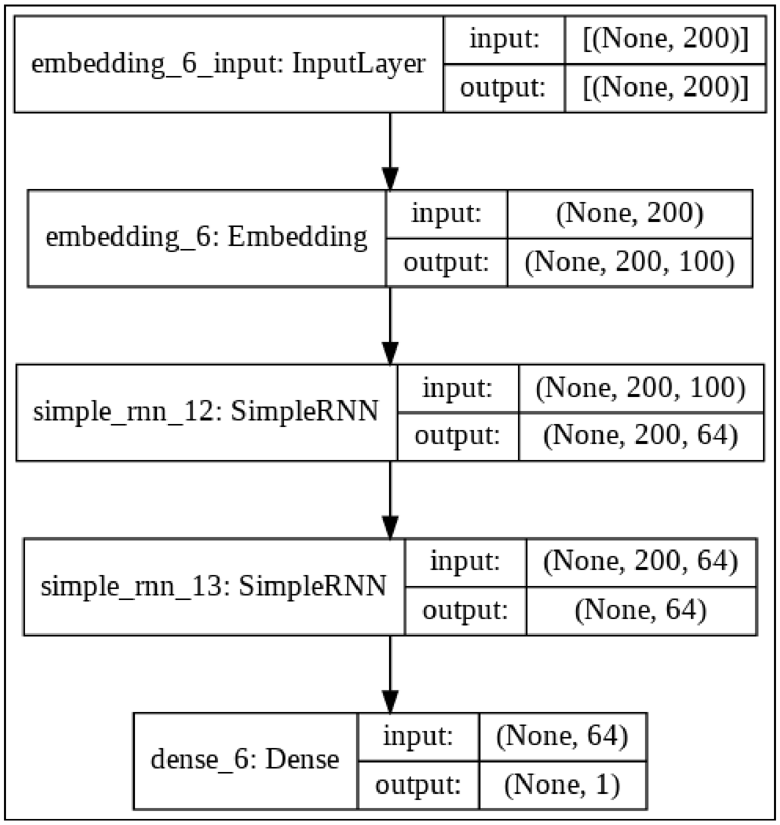


Fig 19. Vanilla RNN model

TWO ISSUES OF STANDARD RNN'S

There are two major obstacles RNNs have had to deal with, but to understand them, you first need to know what a gradient is.

A gradient is a partial derivative with respect to its inputs. If you don't know what that means, just think of it like this: a gradient measures how much the output of a function changes if you change the inputs a little bit.

You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. A gradient simply measures the change in all weights with regard to the change in error.

Exploding Gradients

Exploding gradients are when the algorithm, without much reason, assigns a stupidly high importance to the weights. Fortunately, this problem can be easily solved by truncating or squashing the gradients.

Vanishing Gradients

Vanishing gradients occur when the values of a gradient are too small and the model stops learning or takes way too long as a result.

Layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don't learn, **RNNs can forget what it's seen in longer sequences**, thus having a short-term memory.

This was a major problem in the 1990s and much harder to solve than the exploding gradients. Fortunately, it was solved through the concept of LSTM by Sepp Hochreiter and Juergen Schmidhuber [17].

Long Short Term Memory (LSTM)

Long short-term memory networks [17] are an extension for recurrent neural networks, which basically extends the memory. Therefore it is well suited to learn from important experiences that have very long time lags in between.

The units of an LSTM are used as building units for the layers of a RNN, often called an LSTM network.

LSTMs enable RNNs to remember inputs over a long period of time. This is because LSTMs contain information in a memory, much like the memory of a computer. The LSTM can read, write and delete information from its memory.

This memory can be seen as a gated cell, with gated meaning the cell decides whether or not to store or delete information (i.e., if it opens the gates or not), based on the importance it assigns to the information. The assigning of importance happens through weights, which are also learned by the algorithm. This simply means that it learns over time what information is important and what is not.

In an LSTM you have three gates: input, forget and output gate. These gates determine whether or not to let new input in (input gate), delete the information because it isn't important (forget gate), or let it impact the output at the current timestep (output gate).

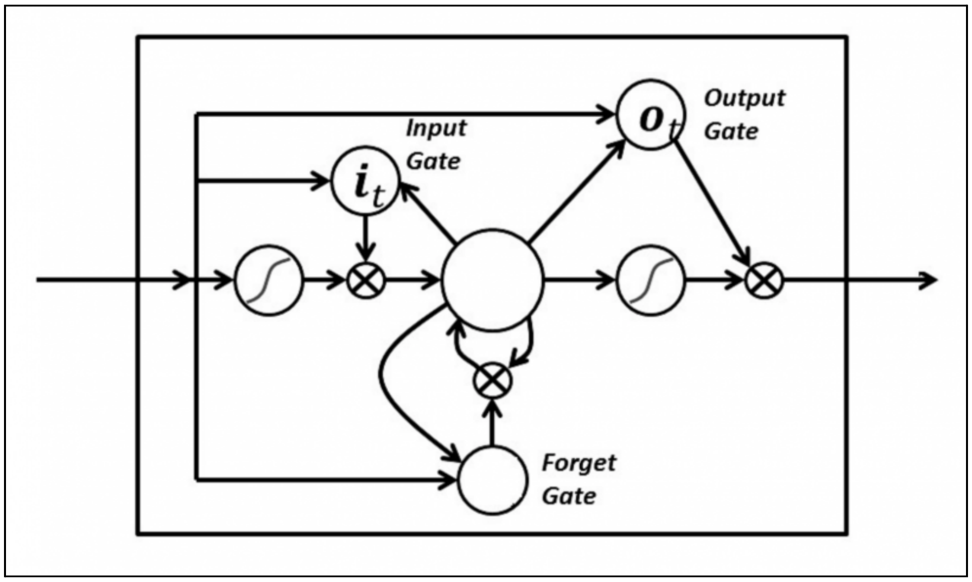


Fig 20. LSTM cell

The problematic issue of vanishing gradients is solved through LSTM because it keeps the gradients steep enough, which keeps the training relatively short and the accuracy high.

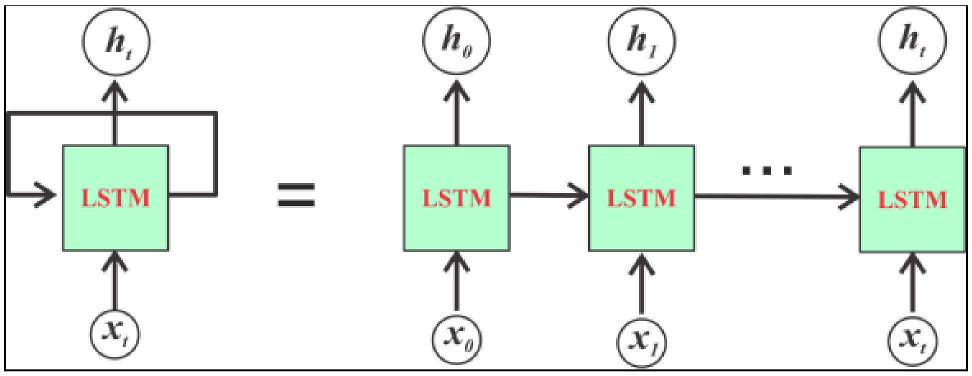


Fig 21. LSTM

A simple two-layer LSTM was used. The same structure as the Vanilla RNN was used with the simple RNN cells being replaced by LSTM cells:

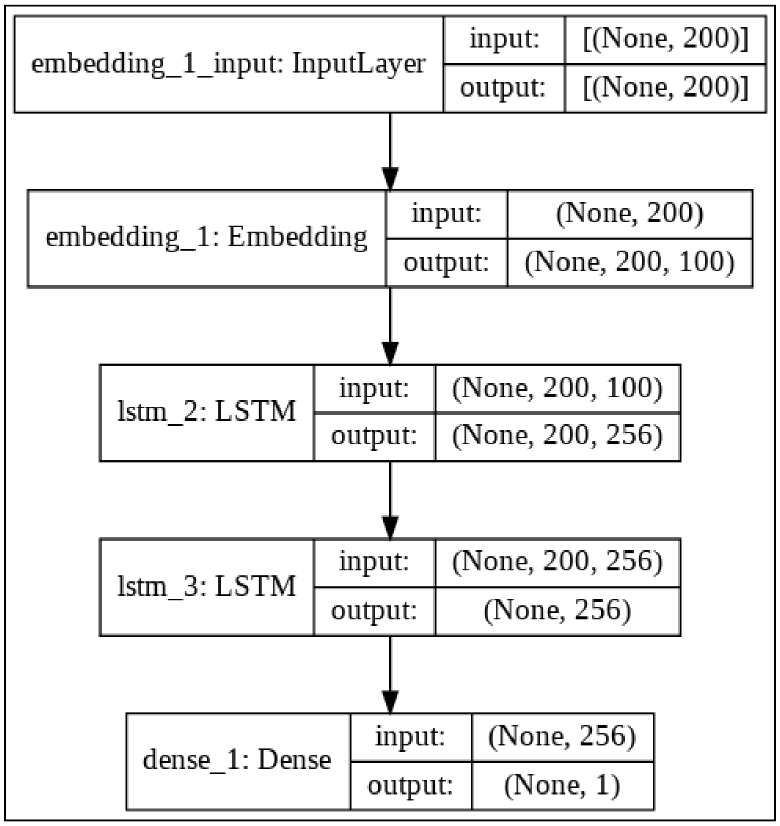


Fig 22. LSTM model

Gated Recurrent Unit (GRU)

The GRU is the newer generation of Recurrent Neural networks and is pretty similar to an LSTM. GRU's got rid of the cell state and used the hidden state to transfer information. It also only has two gates, a reset gate and update gate.

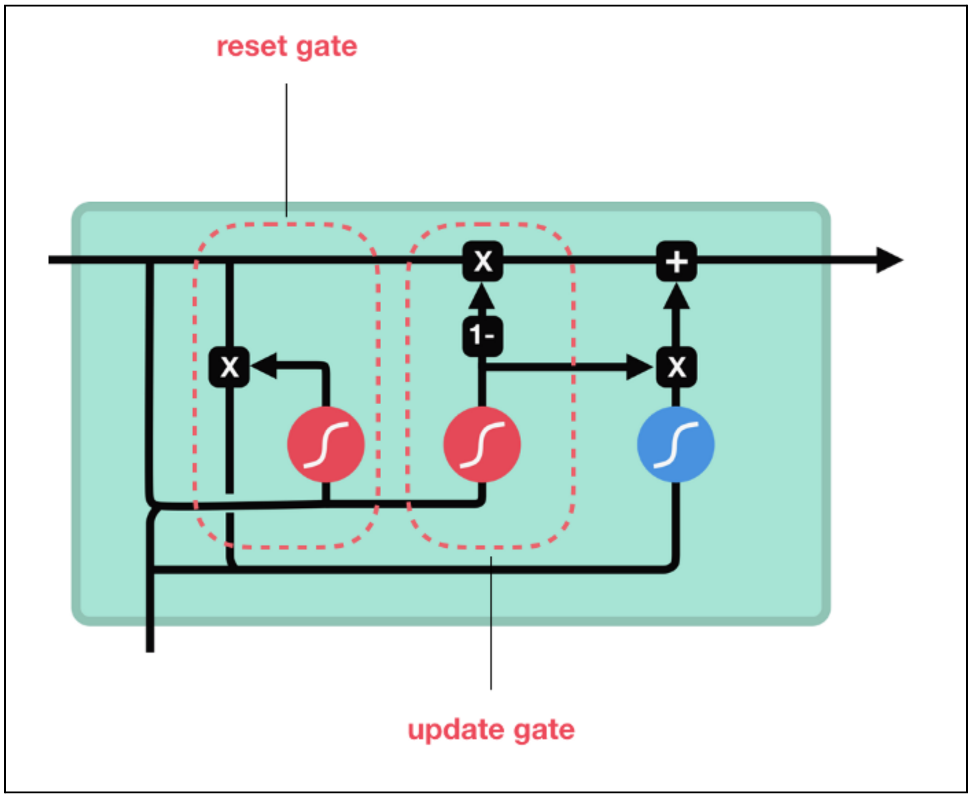


Fig 23. GRU cell

The same structure as RNN and LSTM was used, but the LSTM layers were simple replaced by GRU:

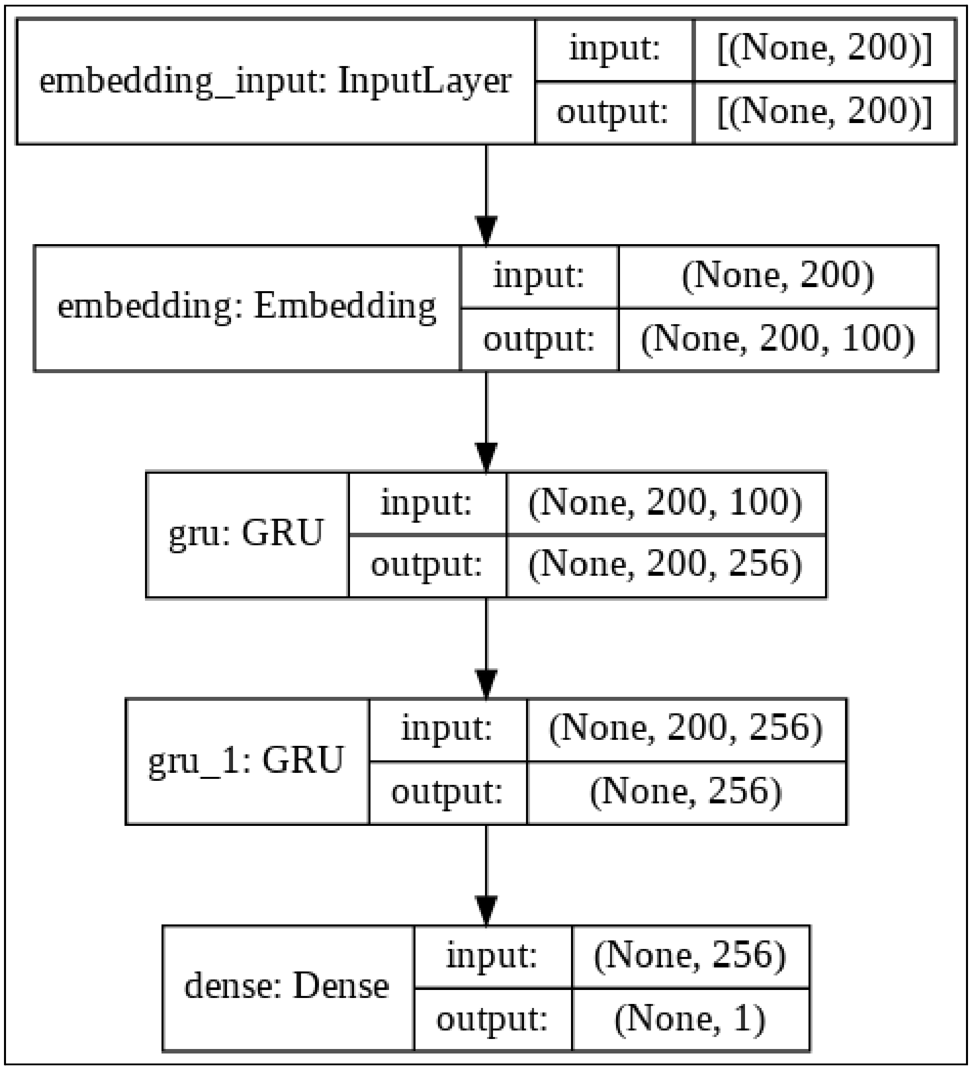


Fig 24. GRU model

Bidirectional LSTM

As a final RNN model, a bidirectional recurrent network with LSTM cells is also to be implemented.

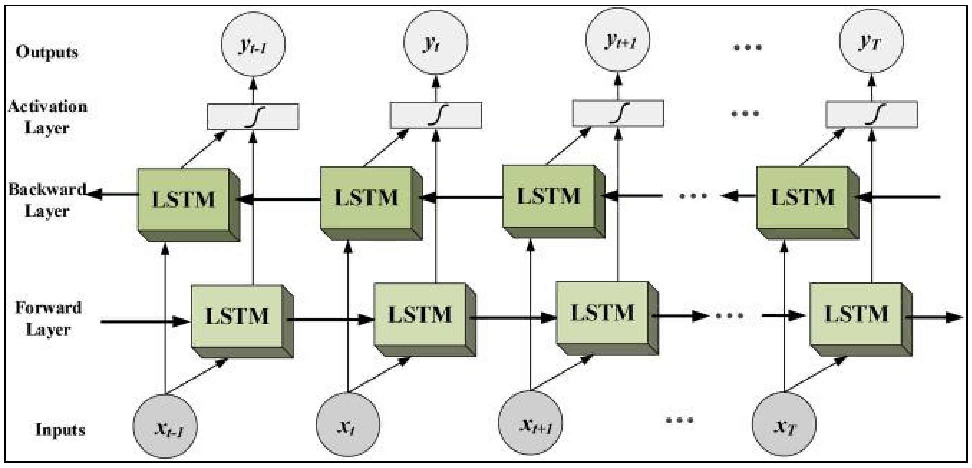
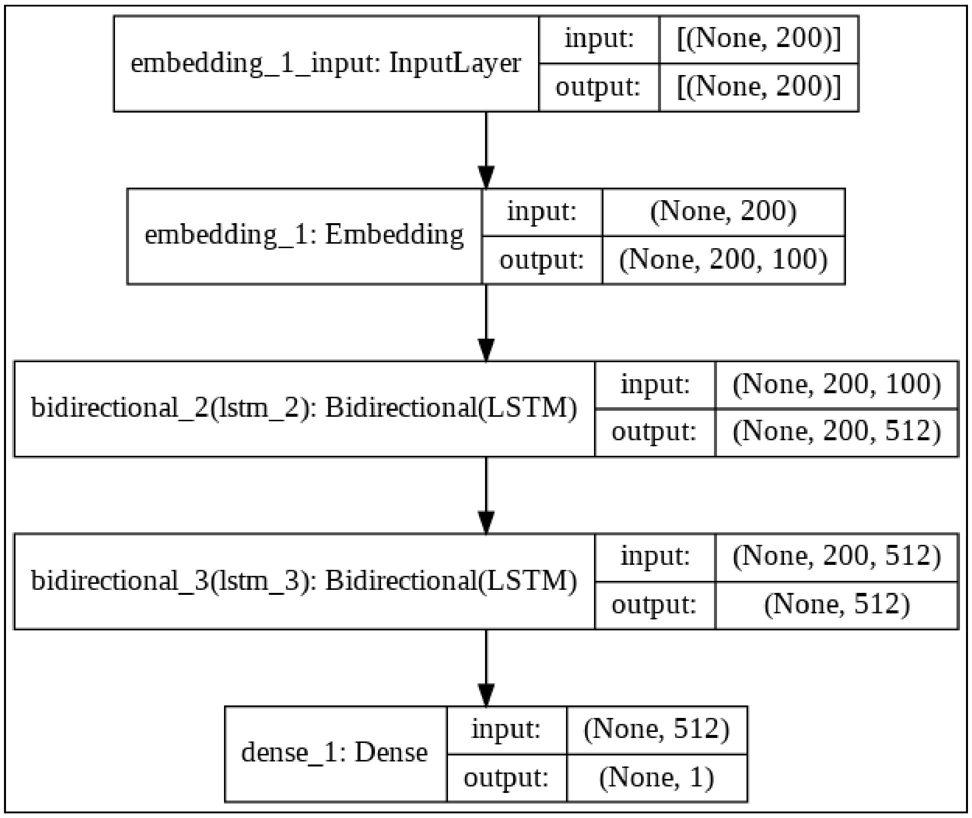


Fig 25. Bidirectional LSTM

The Bi-LSTM model used:



embedding_1_input: InputLayer	input:	[(None, 200)]
	output:	[(None, 200)]
embedding_1: Embedding	input:	(None, 200)
	output:	(None, 200, 100)
bidirectional_2(lstm_2): Bidirectional(LSTM)	input:	(None, 200, 100)
	output:	(None, 200, 512)
bidirectional_3(lstm_3): Bidirectional(LSTM)	input:	(None, 200, 512)
	output:	(None, 512)
dense_1: Dense	input:	(None, 512)
	output:	(None, 1)

Fig 26. Bi-LSTM model

Convolutional Neural Networks (CNN)

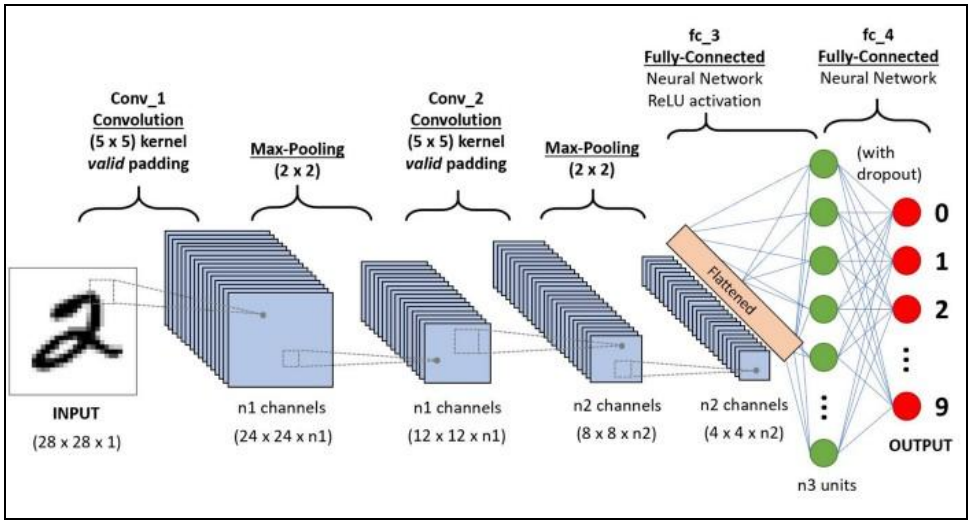


Fig 27. CNN

A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

A ConvNet is able to **successfully capture the spatial and temporal dependencies** in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

A different approach to use CNNs has been undertaken in this project. Instead of images, to which CNNs are generally applied, they have been modified to work to text. As with the previous models, all parameters fed to the model are kept the same. But the model itself has been adjusted to allow feeding of **single-dimensional data (text)** into it instead of two-dimensional (images).

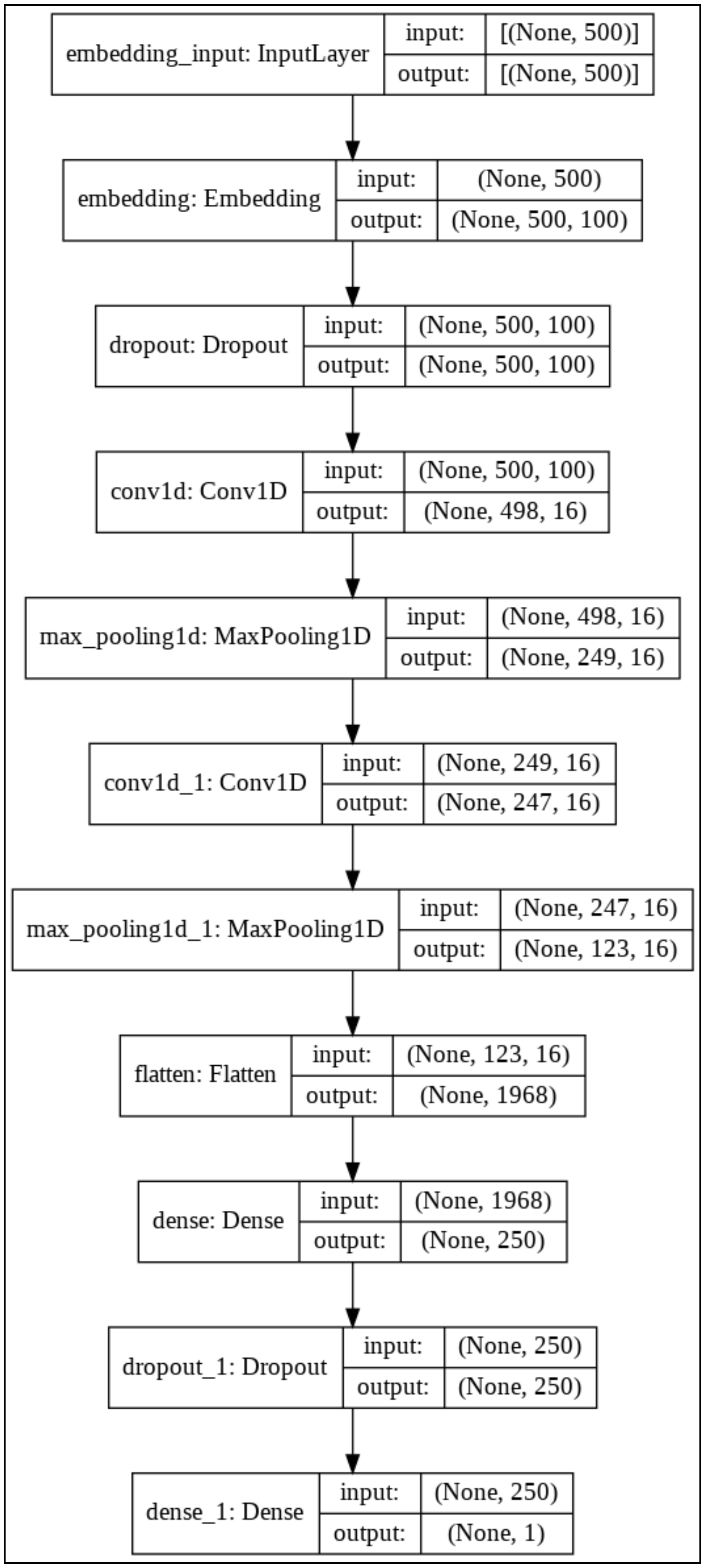


Fig 28. CNN model

TRAINING THE MODELS

***To be noted*:**

1. *Hyperparameters for each model were separately tuned to* ***achieve the maximum accuracy possible*** *on the training and validation sets.*

2. ***Model Checkpointing*** *and* ***Early stopping*** *was used to capture and save the model with* ***best validation accuracy****. All values below are with respect to the best model, irrespective of the number of epochs for which the model was trained.*

Model	**Maxlen**	**Optimizer**	**Learning rate**	**Loss**	**Batch size**	**Epochs**	**Best model epoch**	**Best model accuracy *(validation)***
Logistic Regression	500	SGD	0.01	Binary cross entropy	256	50	50	94.22%
Neural Network	500	Adam	0.01	Binary cross entropy	256	20	13	95.09%
Vanilla RNN	50	Adam	0.001	Binary cross entropy	256	10	2	90.64%
LSTM	200	Adam	(lr scheduler) initial value = 0.001 *	Binary cross entropy	256	10	7	97.15%
GRU	200	Adam	0.001	Binary cross entropy	256	20	17	**97.31%**
Bi-LSTM	200	Adam	0.0001	Binary cross entropy	256	10	9	95.70%
CNN	500	Adam	0.01	Binary cross entropy	256	20	20	96.01%

*decay steps = 10000, decay rate = 0.9

Table 1. Comparing hyperparameters of models

ACCURACY AND LOSS CURVES

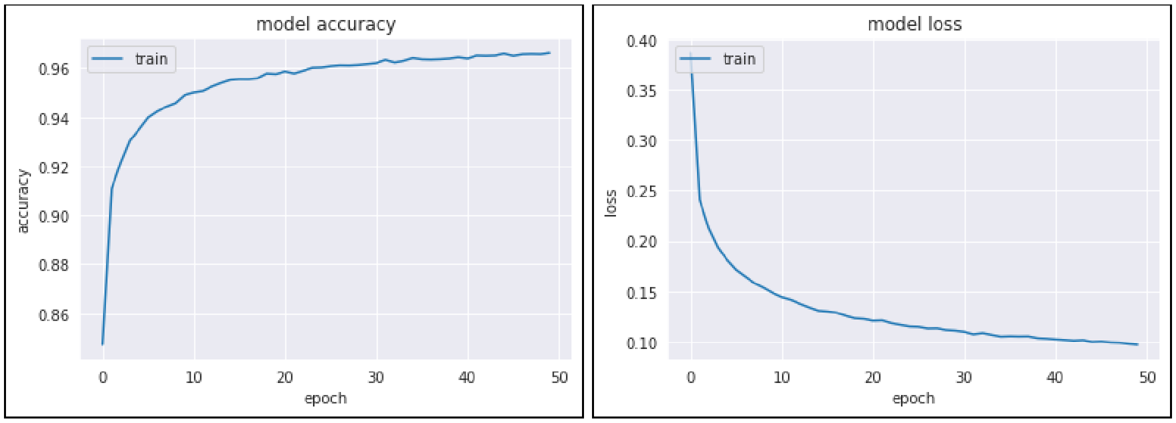


Fig 29. Model accuracy and loss curves for Logistic Regression

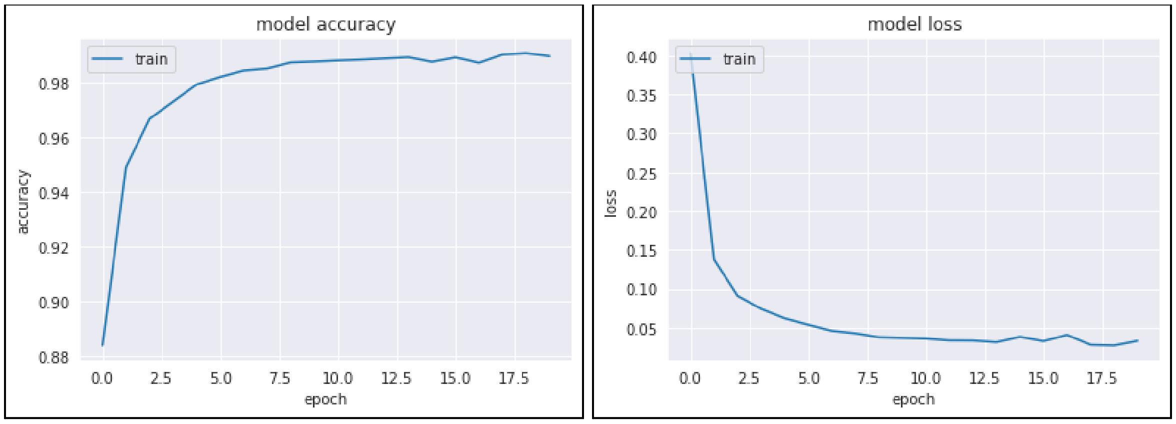


Fig 30. Model accuracy and loss curves for Neural Network

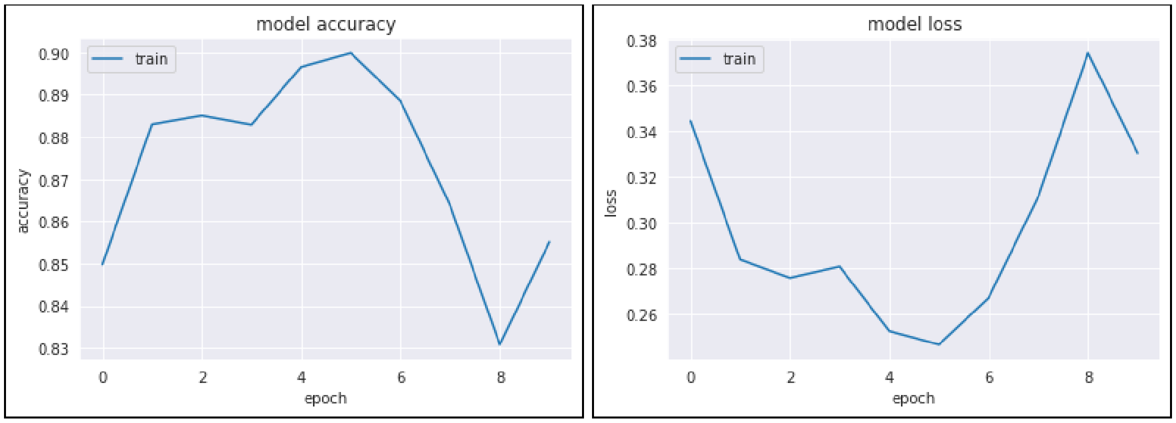


Fig 31. Model accuracy and loss curves for Vanilla RNN

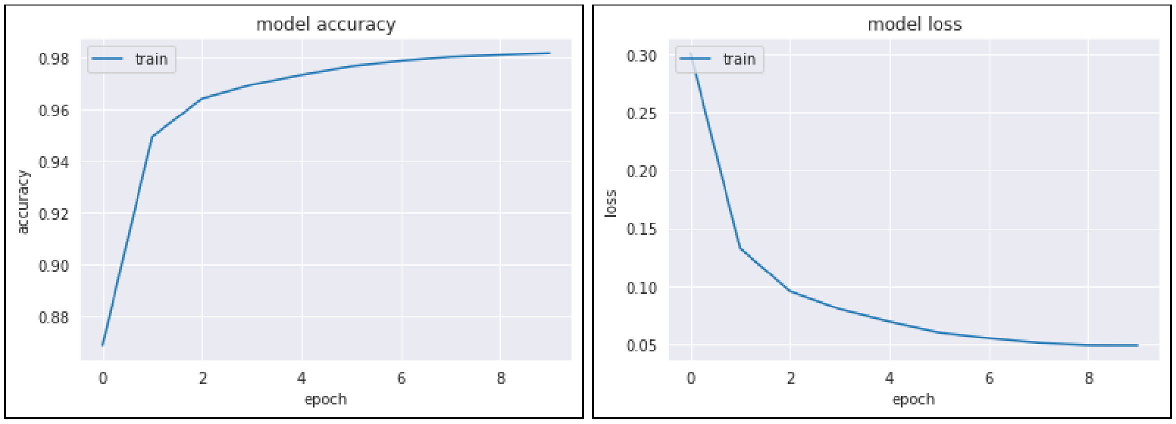


Fig 32. Model accuracy and loss curves for LSTM

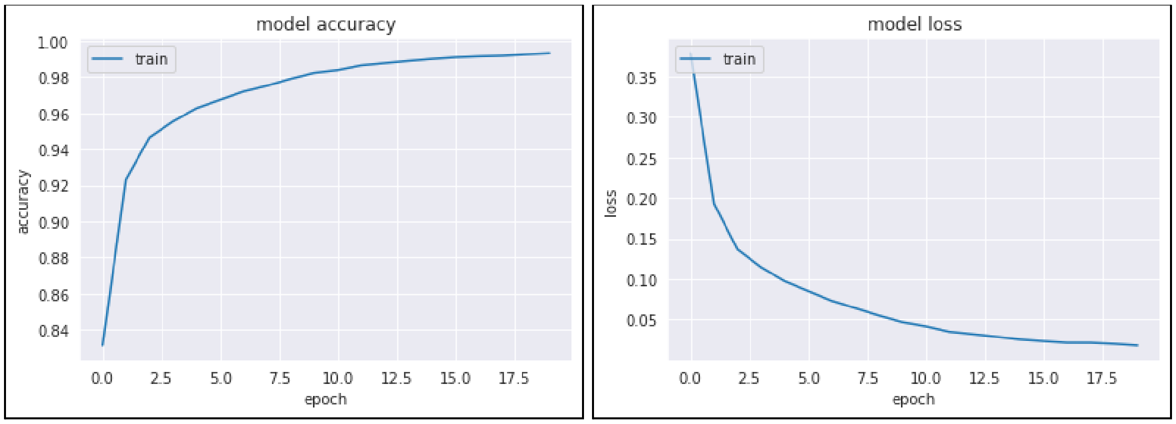


Fig 33. Model accuracy and loss curves for GRU

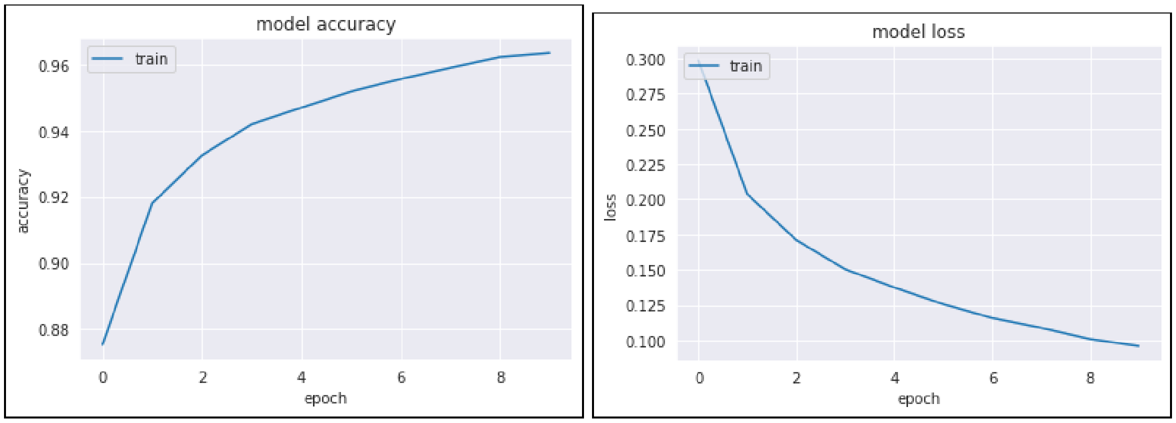


Fig 34. Model accuracy and loss curves for Bi-LSTM

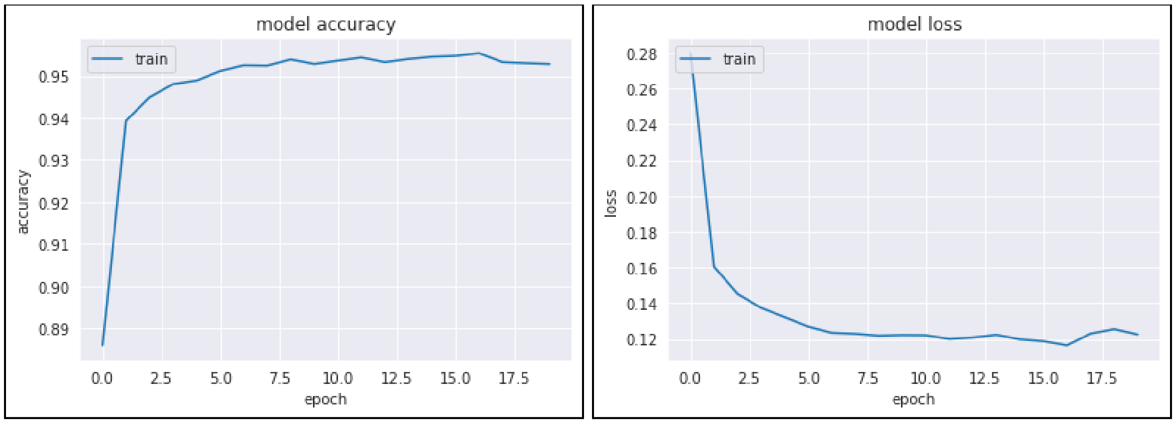
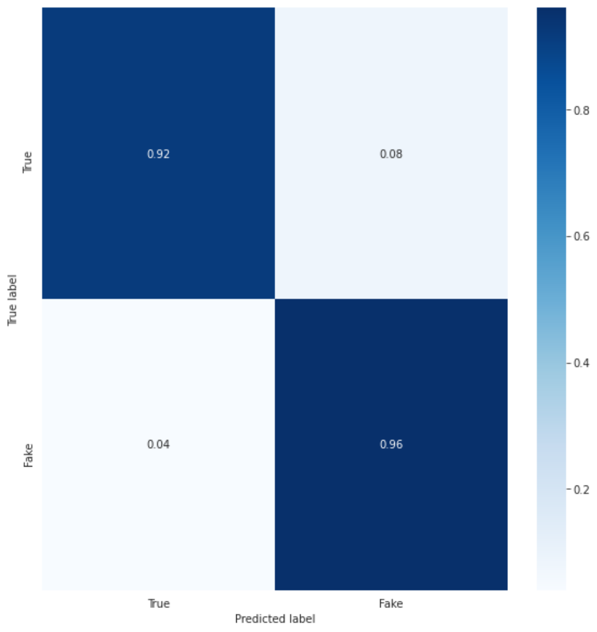
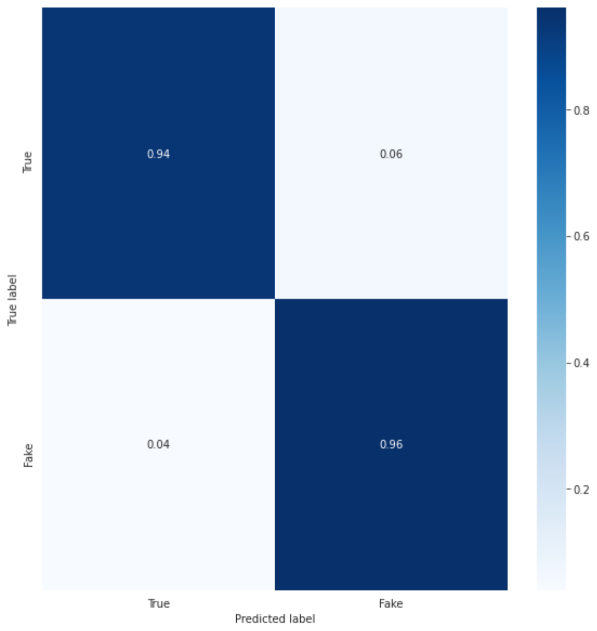
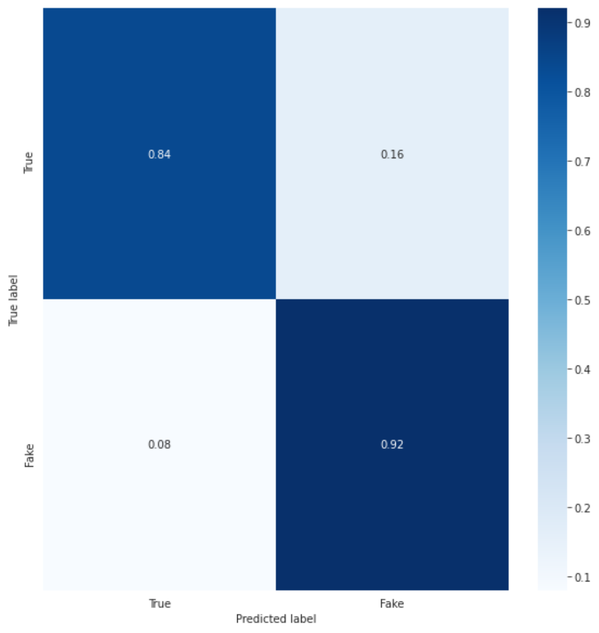
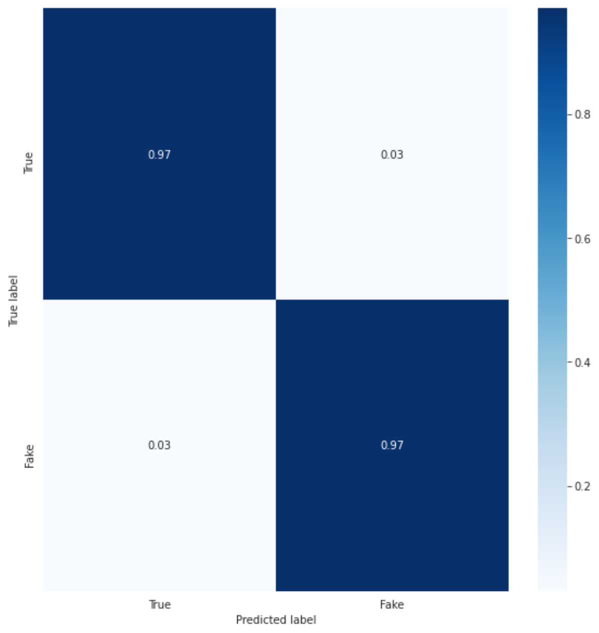
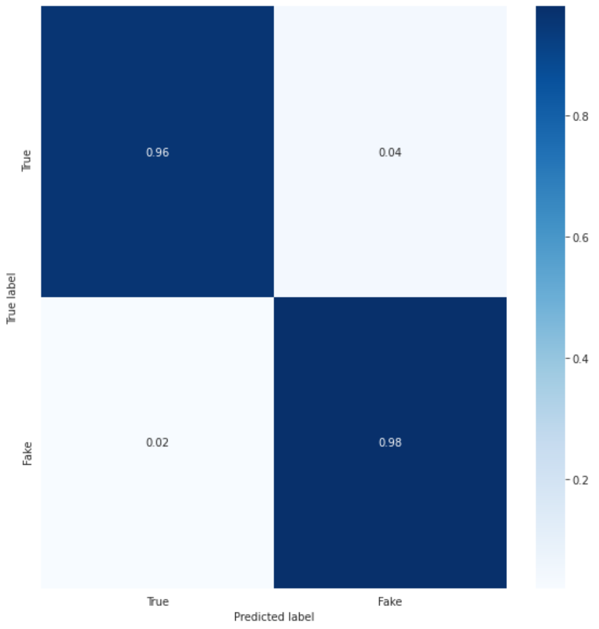
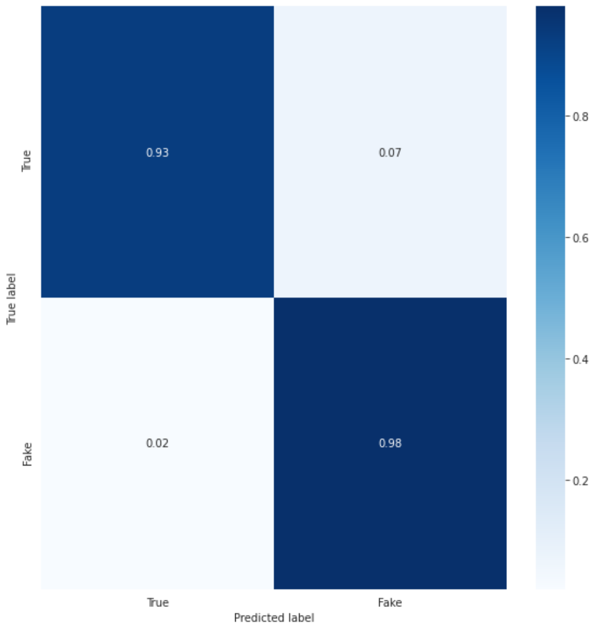


Fig 35. Model accuracy and loss curves for CNN

CONFUSION MATRIX

Model	Confusion Matrix
Logistic Regression	
Neural Network	

Vanilla RNN	
LSTM	

GRU	 0.96 0.04 0.02 0.98 True Fake True label True Fake Predicted label 0.8 0.6 0.4 0.2
Bi-LSTM	 0.93 0.07 0.02 0.98 True Fake True label True Fake Predicted label 0.8 0.6 0.4 0.2

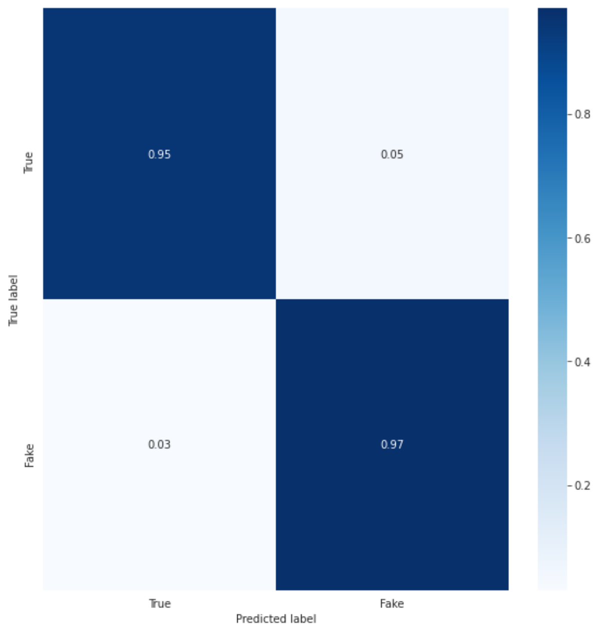
CNN	

Table 2. Confusion matrices of models

PRECISION, RECALL AND F1-SCORE (MACRO-AVERAGED)

Model	Precision	Recall	F1
Logistic Regression	0.94	0.94	0.94
Neural Network	0.95	0.95	0.95
Vanilla RNN	0.88	0.88	0.88
LSTM	**0.97**	**0.97**	**0.97**
GRU	**0.97**	**0.97**	**0.97**
Bi-LSTM	0.96	0.96	0.96
CNN	0.96	0.96	0.96

Table 3. Precision, Recall and F1-scores

Precision: What proportion of **predicted Positives** is truly Positive? (TP/(TP+FP))

Recall: What proportion of **actual Positives** is correctly classified? (TP/(TP+FN))

F1-score = 2 × (precision × recall)/(precision + recall)

OBSERVATIONS AND INFERENCES

1. As can be seen in Table 1, the maxlen (input sequences length) and epoch values are different for different models. This warrants an explanation...

   As was mentioned previously, each model was trained individually to optimize their performance. *The hyperparameters were tuned to maximize the validation data accuracy.*

   For certain models increasing the maxlen to higher values resulted in less than optimum results and very slow training times. So it has been kept to 200 for most models whereas in certain models a maxlen of 500 could also be incorporated.

   The *epoch values were kept to the minimum*, beyond which the models showed no or limited improvement.

2. All the models that were used performed quite well except for the Vanilla RNN.

   The RNN, LSTM, GRU, Bi-LSTM networks all have the same architecture with the RNN cells replaced by LSTM in the LSTM model, which in turn were again replaced by GRU cells in the GRU model and so on.

   Except the RNN all other models had more or less similar performance during training. The RNN could not optimize its performance for input sequences more than a maxlen of 50. Maxlen values of 100 and above were also tested and other hyperparameters were also tuned but with no positive increase in performance.

   Reasons for this may be the fact that *RNN can only remember short term information* and as a result long term dependencies in longer sequences are overlooked by it. The learning curve for RNN also shows similar behaviour corresponding to the *vanishing gradient problem*.

   Quite interestingly there is a sharp drop in accuracy. This is also a reason why the epochs for the RNN model were kept to a minimum, since the model's accuracy dropped for longer epochs and it was unnecessary to train it any longer.

3. The models which performed the best considering all relevant metrics like accuracy, precision, recall and f1-score are the **LSTM** and **GRU** models. These models have shown to perform best with sequential data and it is of no surprise that they excelled in this study.

4. Finally the CNN model performed better than expected with text data. Although CNN models are rarely used for text-classification purposes, they certainly require more research and are a promising area in NLP.

FUTURE SCOPE

A complete, production-quality classifier for fake news will incorporate many different features beyond the vectors corresponding to the words in the text. For fake news detection, we can add as features the source of the news, including any associated URLs, the topic (e.g., science, politics, sports, etc.), publishing medium (blog, print, social media), country or geographic region of origin, publication year, as well as linguistic features not exploited in this exercise—use of capitalization, fraction of words that are proper nouns, and others.

We also believe that CNNs are a promising avenue worth exploring further. Finally, more sophisticated models—for example, pointer and highway networks— definitely merit investigation.

REFERENCES

[1] Simon, Annina & Deo, Mahima & Selvam, Venkatesan & Babu, Ramesh. (2016). An Overview of Machine Learning and its Applications. International Journal of Electrical Sciences & Engineering. Volume. 22-24.
[2] F. Chollet, "Deep Learning With Python", Manning Publications, 2017.
[3] Jiang, M.; Liang, Y.; Feng, X.; Fan, X.; Pei, Z.; Xue, Y.; Guan, R. Text classification based on deep belief network and softmax regression. Neural Comput. Appl. 2018, 29, 61–70.
[4] Kowsari, Kamran & Jafari Meimandi, Kiana & Heidarysafa, Mojtaba & Mendu, Sanjana & Barnes, Laura & Brown, Donald & Id, Laura & Barnes,. (2019). Text Classification Algorithms: A Survey. Information (Switzerland). 10. 10.3390/info10040150.
[5] https://www.kaggle.com/c/fake-news/overview
[6] https://www.kaggle.com/rchitic17/real-or-fake
[7] https://www.kaggle.com/clmentbisaillon/fake-and-real-news-dataset
[8] https://www.kaggle.com/mohamadalhasan/a-fake-news-dataset-around-the-syrian-war
[9] https://www.kaggle.com/jruvika/fake-news-detection
[10] https://www.kaggle.com/ruchi798/source-based-news-classification
[11] https://www.uvic.ca/engineering/ece/isot/datasets/
[12] https://www.nltk.org/_modules/nltk/corpus.html
[13] https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer
[14]https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences
[15] https://code.google.com/archive/p/word2vec/
[16] https://nlp.stanford.edu/projects/glove/
[17] Hochreiter, S., and Schmidhuber, J., Long Short-Term Memory, Neural Computation, 1997