

Symbiosis institute of technology



Skill development lab Article

Next Word Prediction Model

Submitted by :

Kavya Suthar  
IT 2  
PRN : 18070124037

## Index

Sr. No.	Title	Page No.
1	Abstract	2
2	Introduction	3
3	Pre-processing the dataset	4
4	Feature Engineering	5
5	LSTM	6
6	Creating the model	9
7	Evaluating the next Word prediction	11
8	Prediction	12
9	Literature review	14
10	References	15

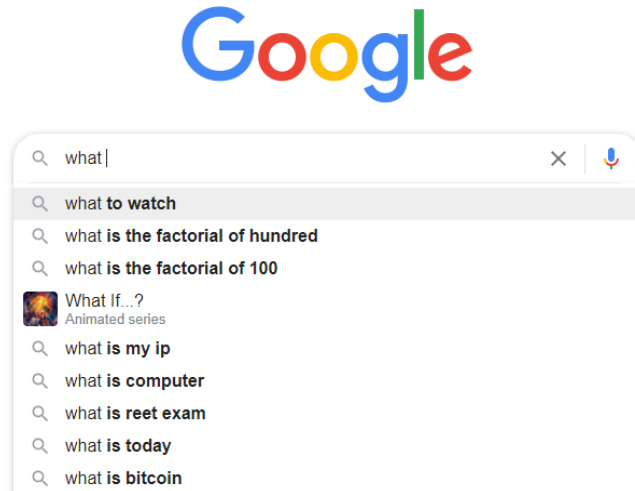
## Abstract

In this article, I've explained my LSTM model which is a Recurrent Neural Network(RNN) to predict the next word in a sentence. The aim of this project was to mimic google autocomplete or suggestions we see on our keyboards. This model gives out possible words that can complete the sentence.

Using LSTM in the Sequential model by keras I was able to reach 75.28 % of accuracy and reduce loss to 1.4. The dataset I've used to train this neural network is a book by arthur conan doyle. It's a novel using simple words. I've taken the first five words as features and the next word as a label. Data can be improved by inserting the general sentences people use in daily life. It couldn't be possible as i didn't have that enough data on myself like emails that's why i referred to the most basic approach for such kind of projects by using most feasible data.

## Introduction

We can see this model around us everyday. When we type what or how google already comes out with suggestions.



Electronic communication and mass communication are commonplace in today's real world social media. By reducing the use of typing time, the prediction of the next word in the written text can be very useful for everyday use and easy communication.

This Project is inspired by google autocomplete and how our keyboard comes up with different suggestions for us. I've used Natural language processing and used a Long short term memory(LSTM) layer in keras sequential model. It trains on a text extracted from a book.

Natural language processing has become a research area and is widely used in various programs. We tend to love texting and find that every time we try to type a text a suggestion pops up trying to predict the next word we want to write. This prediction process is one of the applications NLP works with. People have made great progress here and can use Recurrent neural networks in such a process.

The model will process the last word of a particular sentence and predict the next possible word. We will use natural language processing techniques, language modeling, and in-depth learning. We will begin by analyzing the data followed by the pre-processing of the data. We will then token this data and finally build a deeper learning model. An in-depth learning model will be developed using LSTMs.

## Pre-processing

The first step is to remove all the unnecessary data from the dataset. It's a text file so we'll read it using open and lower all words and store them in a variable. Then we'll remove all the special

characters so that it will remove the noise from the dataset and then we can select features and labels from the text.

```
data=open('/content/1661-0.txt','r')
data=data.read().lower()
raw_data=data
raw_data.replace(',','')
raw_data.replace('"',' ')
```

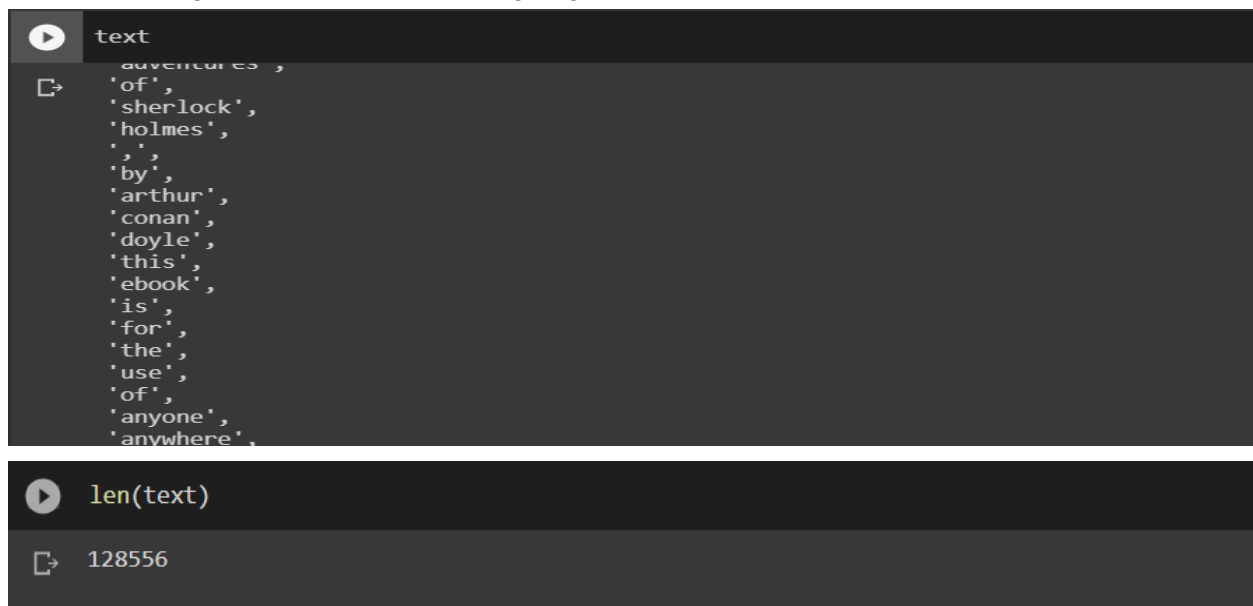
Then we can use ant tokenizer to tokenize the text and separate all the words individually without any special characters. Tokenization means splitting large text data, essays, or corpus into smaller segments. These subdivisions can be in the form of small documents or lines of text data. It can also be a dictionary of words. It refers to converting the raw text into a list of words that's been used in the raw text. It is the most basic processing in any natural language processing model to break down the data.

```
from nltk.tokenize import RegexpTokenizer
import nltk
nltk.download('punkt')
```

Punkt is a package used by natural language toolkit's RegexpTokenizer. This package divides a text into a list of sentences.

```
tokenizer=RegexpTokenizer(r'w+')
text=word_tokenize(raw_data)
```

After tokenizing this is how our text file going to look like



The screenshot shows a Jupyter Notebook interface with two cells. The first cell, titled 'text', contains a list of tokens from a text file, including 'adventures', 'of', 'sherlock', 'holmes', 'by', 'arthur', 'conan', 'doyle', 'this', 'ebook', 'is', 'for', 'the', 'use', 'of', 'anyone', and 'anywhere'. The second cell, titled 'len(text)', shows the output '128556', representing the total number of tokens in the list.

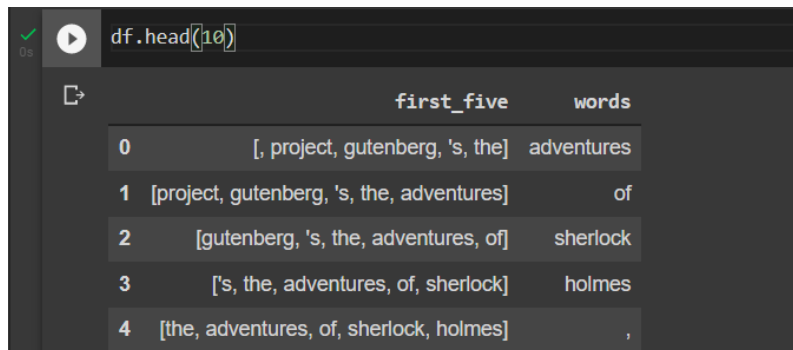
## Feature Engineering

Neural Networks are also well known to process languages But in Natural language processing the input that we provide to any model is not in the form of string rather we vectorize those

strings and convert those features to Numpy arrays so that we can apply mathematical equations to the input and understand the Language. Features Engineering is the process of creating features that can be used intraining a model.

It converts whatever information we have in our problem and turns it into numbers and we call it a feature matrix. If it is executed correctly then it can increase our accuracy and reduce loss.

To create a feature and label I've used the first consecutive word as feature and next word as label. So I created a pandas dataframe with two columns 'first\_five' and 'words'.



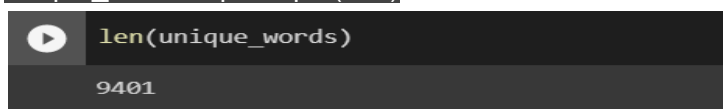
A Jupyter Notebook cell with the code `df.head(10)` is shown. The output is a pandas DataFrame with two columns: 'first\_five' and 'words'. The 'first\_five' column contains lists of the first five words of each sentence, and the 'words' column contains the sixth word, which serves as the label. The first five rows of the DataFrame are displayed.

	first_five	words
0	[, project, gutenber, 's, the]	adventures
1	[project, gutenber, 's, the, adventures]	of
2	[gutenber, 's, the, adventures, of]	sherlock
3	['s, the, adventures, of, sherlock]	holmes
4	[the, adventures, of, sherlock, holmes]	,

There are still small amounts of anomalies in the data but we have a large amount of data so it won't count.

We have also collected all the unique words from our tokenized text.

```
unique_words=np.unique(text)
```



A Jupyter Notebook cell with the code `len(unique_words)` is shown. The output is the integer value 9401.

```
len(unique_words)
9401
```

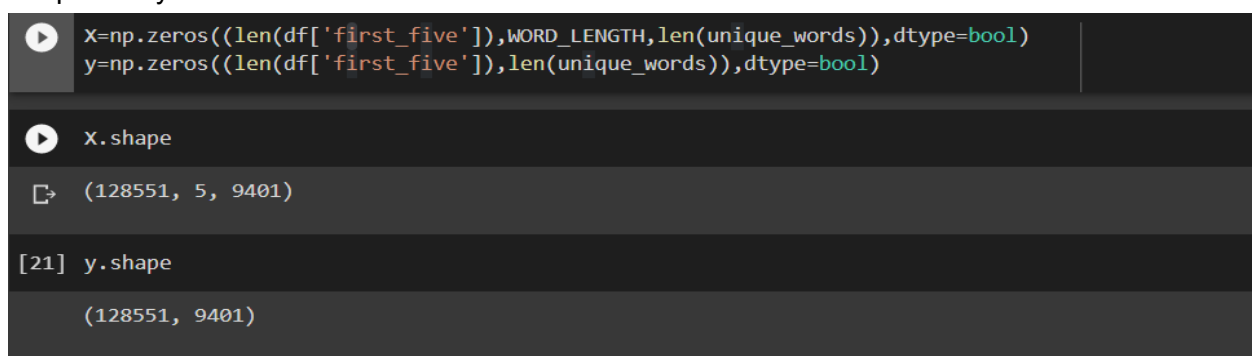
We'll convert this data into a feature matrix consisting only of numbers so that our model can work properly. For that i've created two numpy arrays of shape

```
( len(df['first_five']),WORD_LENGTH,len(unique_words))
```

and

```
( len(df['first_five']),len(unique_words))
```

Respectively.



A Jupyter Notebook cell with the code `X=np.zeros((len(df['first_five']),WORD_LENGTH,len(unique_words)),dtype=bool)` and `y=np.zeros((len(df['first_five']),len(unique_words)),dtype=bool)` is shown. The output is the integer value 9401.

```
X=np.zeros((len(df['first_five']),WORD_LENGTH,len(unique_words)),dtype=bool)
y=np.zeros((len(df['first_five']),len(unique_words)),dtype=bool)
```

A second Jupyter Notebook cell with the code `X.shape` is shown. The output is the tuple (128551, 5, 9401).

```
X.shape
(128551, 5, 9401)
```

A third Jupyter Notebook cell with the code `y.shape` is shown. The output is the tuple (128551, 9401).

```
y.shape
(128551, 9401)
```

In our feature matrix in a numpy array consisting of zeros and ones. Where index of one defining what the word is with respect to position of that word in our unique\_words list.

```
x[:5]
array([[False, False, False, ..., False, False, True],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]],

      [[False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

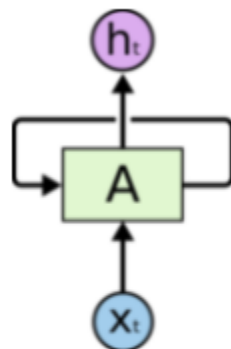
This is a quick view of our feature matrix.

## LSTM

### RNNs

People do not start their thinking from the beginning every second. As you read this story, you are going to understand each word based on your understanding of past words. You do not throw away everything and start thinking from the beginning again. Your thoughts can be persistent. Traditional neural networks cannot do this, and it seems to be a major shortcoming. For example, imagine you want to differentiate what kind of event happens all the time in a movie. It is not yet clear how the traditional neural network can use its thinking about previous events in the film to inform later.

Recurrent neural networks deal with this problem. Networks have loops in them, allowing information to flow.



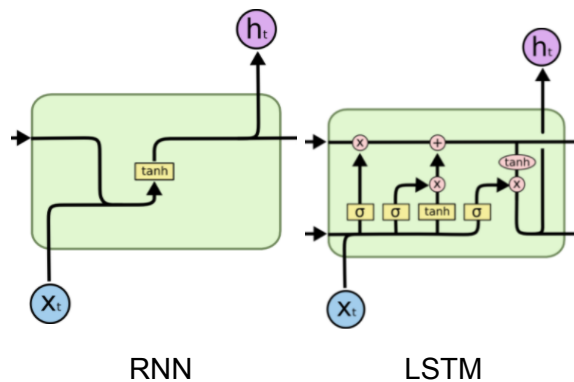
**Recurrent Neural Networks have loops.**

### Why LSTM

Sometimes, we only need to look at the latest details in order to perform the current task. For example, consider a language model that attempts to predict the next word based on the past

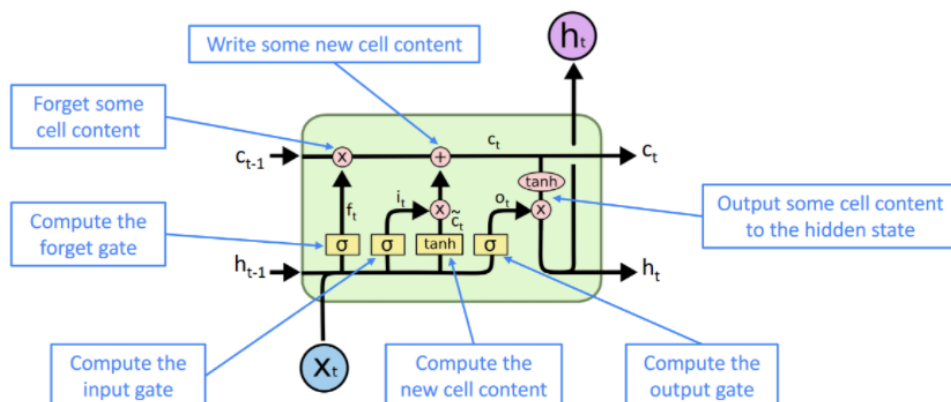
tense. If we try to predict the last word for "clouds in the sky," we do not need another theme - obviously the next word will be sky. In such cases, where the gap between the relevant information and the required area is small, RNNs may learn to apply the previous information. For such problems RNN is a very good neural Network but when we deal with large sentences RNN is unable to connect information between them.

Here LSTM takes over. Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. it has different repeating module, instead of

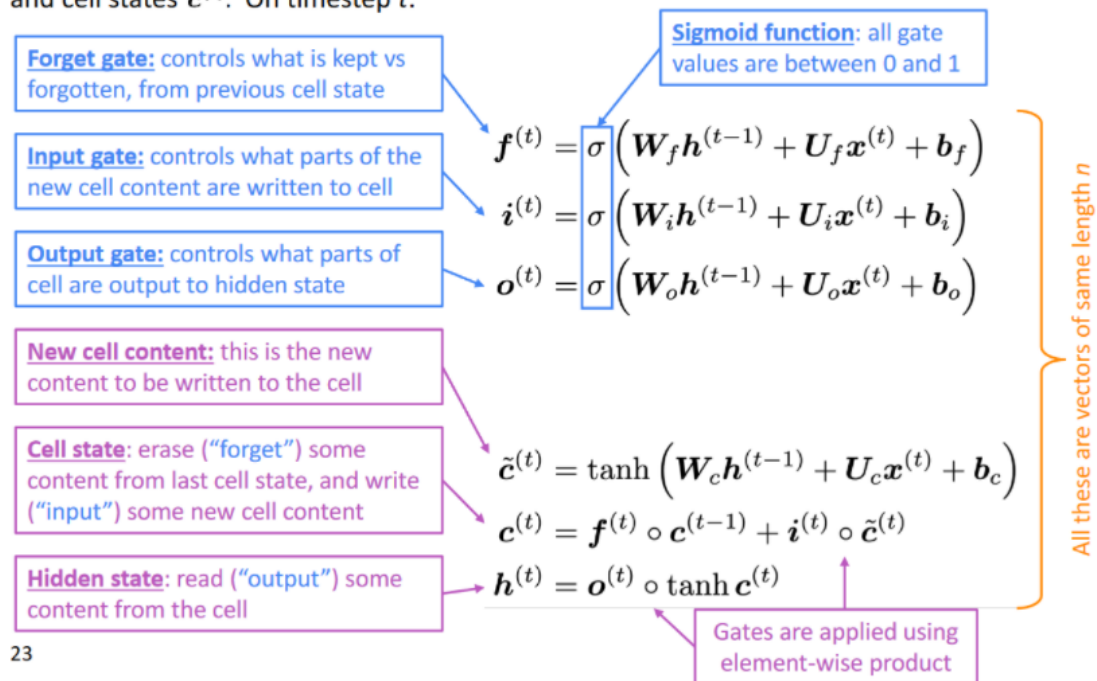


## LSTM

1. Instead one parameter in the LSTM cell state also travels through Networks. Cell state is the one that helps with long-term dependencies.
2. In one lstm cell it has three gates. Forget gate, Input gate and Output gate. Forget gate decides what information to forget in cell state. Input gate decides what to get added in cell state content. Output gate what to send to hidden state.
3. Forget Gate is widely used to properly control which information needs unnecessary deletion. The input gateway ensures that new information is added to the cell and the output confirms which parts of the cell are released in the next hidden state. The sigmoid function used in each gate statistic ensures that we can reduce the value to 0 or 1.



We have a sequence of inputs  $\mathbf{x}^{(t)}$ , and we will compute a sequence of hidden states  $\mathbf{h}^{(t)}$  and cell states  $\mathbf{c}^{(t)}$ . On timestep  $t$ :



23

<https://medium.com/@antonio.lopardo/the-basics-of-language-modeling-1c8832f21079>

4. The core idea: the key to LSTM is the cell shape, the horizontal line that passes over the drawing. The state of the cell is like a transmission belt. Go down straight across the chain, with just a little junction of the line. It is very easy for information to flow freely. LSTM has the ability to extract or add information to the cell environment, carefully controlled by structures called gates. Gates are a way of voluntarily passing information. They are built on the sigmoid neural net layer and direct replication function. The sigmoid layer gives the numbers between zero and one, which means how much each part should be transferred. The zero value means "don't pass anything," while the one value means "let it all go!". LSTM has three gates, namely, to protect and control the state of the cell.
5. The first step in our LSTM is to determine what information we will discard in the cell state. This decision is made by a sigmoid layer or relu called the "forgotten gate layer." It looks at  $\mathbf{h}_{t-1}$  and  $\mathbf{x}_t$ , and pulls a number between 0 and 1 in each number in the  $\mathbf{C}_{t-1}$  cell state. 1 represents "keeping this completely" while 0 stands for "completely discarding this."
6. Let's go back to our example of a language model trying to predict the next word based on all the past. In such a problem, the cell status may include the gender of the current title, in order to use the appropriate pronouns. When we see a new topic, we want to forget the gender of the old topic.



7. The next step is to determine what new information we will store in the cell environment. This has two parts. First, a sigmoid layer called the “input gate layer” determines what values we will review. Next, the tanh layer creates a vector of new candidate values,  $C \sim t$ , which can be added to the region. In the next step, we will combine these two to create a world update. In the example of our language model, we would like to add the gender of a new topic to the cell shape, in order to replace the old one we forget. Now it is time to update the status of the old cell,  $C_{t-1}$ , into the new state of  $C_t$ . The previous steps have decided what we should do, we just need to do it really. We repeat the old situation in  $f_t$ , forgetting the things we decided to forget earlier. Then we add  $* C \sim t$ . These are new candidate values, measured by how often we decide to review the value of each country.
8. In the case of the language model, this is where we actually discarded information about the gender of the old topic and added new information, as we decided in previous steps. Finally, we need to decide what to bring out. This output will be based on the status of our cell, but will be a filtered version. First, we use a sigmoid layer that determines which parts of the cell structure we will release. Then, we set the state of the cell using tanh (to push the values between  $-1$  and  $1$ ) and multiply by exiting the sigmoid gate, to extract only the parts we have decided to make. For example a language model, having recently seen a topic, may want to extract action-related information, in which case that is the following. For example, it may specify whether the subject is singular or plural, so that we know what the verb should be combined into when it is the following.
9. Finally, we need to decide what we’re going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we’re going to output. Then, we put the cell state through tanh (to push the values to be between  $-1$  and  $1$ ) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to. For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that’s what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that’s what follows next.

## Creating the model

I built a Sequential model. Then I created an embedded layer and specified the input size and output size. It is important to specify the input length as 1 as the prediction will be made with one specific word and we will get a response to that word. We will be adding an LSTM layer to our build. We will give it 128 units and ensure that we return the sequence as true. This is to ensure that we can transfer it to another LSTM layer. For the next layer of LSTM, we will transfer it to another 128 units but there is no need to specify the return sequence as it is automatically false. We will bypass this with an encrypted 128-node layer using a dense layer function with a com set as activation. Finally, we transfer it to an output layer with a specified volume size and

activation of softmax. Activating softmax ensures that we get a lot of opportunities in outputs equal to voice size. The entire code of our model structure is as shown below.

In the sequential model i added the following layers

```
model = Sequential()  
model.add(LSTM(128, input_shape=(WORD_LENGTH, len(unique_words))))  
model.add(Dense(len(unique_words)))  
model.add(Activation('softmax'))
```

1. The LSTM layer is telling the model to choose LSTM as a Neural Network. And
2. Dense layer : A dense layer is a layer of deeply connected neural network, which means that each neuron in a dense layer receives input from all the neurons of its previous layer. The dense layer is found to be the most widely used layer in models. In the background, a dense layer performs matrix-vector multiplication. The values used in the matrix are actually parameters that can be trained and reviewed with the help of backpropagation. The output produced by the dense layer is a 'm' vector. Therefore, a dense layer is used primarily to change the vector size. The layers are compact and work with functions such as rotation, scale, vector translation.
3. Softmax Activation : It works like probability Distribution. This layer decides which word will have higher probability. This works as an output layer for our model. So the output we receive is the probabilities of a particular word occurring. And the word with highest probability could be the most possible prediction.

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Here z represent values from the neurons of output layer

Training the model

Finally we train the model with our feature matrix we created in the last to last section. Here, we train the model and maintain the best weights in nextword1.h5 so that we do not have to re-train the model over and over again and be able to use our saved model when needed. Here I have only trained with training data. However, you can choose to train with both train and confirmation data. The losses we have used are categorical\_crossentropy which includes cross-entropy losses between labels and speculation. The optimizer will be using keras RMSprop with a reading rate of 0.01 and will integrate our model into a matric loss.

Training at batch\_size = 128 and epoch=8  
Accuracy at last epoch 50.54

```
optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
history = model.fit(X, y, validation_split=0.05, batch_size=128, epochs=8, shuffle=True).history
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/rmsprop.py:130: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(RMSprop, self).__init__(name, **kwargs)
Epoch 1/8
955/955 [=====] - 376s 391ms/step - loss: 5.2461 - accuracy: 0.2407 - val_loss: 7.4013 - val_accuracy: 0.1245
Epoch 2/8
955/955 [=====] - 370s 387ms/step - loss: 4.8061 - accuracy: 0.2766 - val_loss: 7.3026 - val_accuracy: 0.1324
Epoch 3/8
955/955 [=====] - 367s 385ms/step - loss: 4.3859 - accuracy: 0.3175 - val_loss: 7.6986 - val_accuracy: 0.1389
Epoch 4/8
955/955 [=====] - 367s 384ms/step - loss: 3.9698 - accuracy: 0.3619 - val_loss: 8.1781 - val_accuracy: 0.1277
Epoch 5/8
955/955 [=====] - 369s 387ms/step - loss: 3.6131 - accuracy: 0.4027 - val_loss: 8.3959 - val_accuracy: 0.1199
Epoch 6/8
955/955 [=====] - 374s 392ms/step - loss: 3.3202 - accuracy: 0.4438 - val_loss: 8.7090 - val_accuracy: 0.1010
Epoch 7/8
955/955 [=====] - 395s 413ms/step - loss: 3.0752 - accuracy: 0.4788 - val_loss: 8.5708 - val_accuracy: 0.1159
Epoch 8/8
955/955 [=====] - 398s 416ms/step - loss: 2.9076 - accuracy: 0.5054 - val_loss: 8.8871 - val_accuracy: 0.1047
```

Training at batch\_size=256 and epochs=16  
Accuracy at last epoch = 75.28

```
optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer,
metrics=['accuracy'])
history = model.fit(X, y, validation_split=0.05, batch_size=256,
epochs=16, shuffle=True).history
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/rmsprop.py:130: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(RMSprop, self).__init__(name, **kwargs)
Epoch 1/16
478/478 [=====] - 330s 685ms/step - loss: 2.6684 - accuracy: 0.5395 - val_loss: 8.6891 - val_accuracy: 0.1075
Epoch 2/16
478/478 [=====] - 318s 665ms/step - loss: 2.1040 - accuracy: 0.6066 - val_loss: 9.2636 - val_accuracy: 0.1002
Epoch 3/16
478/478 [=====] - 316s 661ms/step - loss: 1.8334 - accuracy: 0.6439 - val_loss: 9.2480 - val_accuracy: 0.0916
Epoch 4/16
478/478 [=====] - 333s 697ms/step - loss: 1.6749 - accuracy: 0.6717 - val_loss: 9.6839 - val_accuracy: 0.0963
Epoch 5/16
478/478 [=====] - 324s 678ms/step - loss: 1.5732 - accuracy: 0.6902 - val_loss: 9.7636 - val_accuracy: 0.0949
Epoch 6/16
478/478 [=====] - 311s 651ms/step - loss: 1.5010 - accuracy: 0.7055 - val_loss: 9.6521 - val_accuracy: 0.0916
Epoch 7/16
478/478 [=====] - 320s 670ms/step - loss: 1.4642 - accuracy: 0.7164 - val_loss: 9.7028 - val_accuracy: 0.0896
Epoch 8/16
478/478 [=====] - 322s 673ms/step - loss: 1.4288 - accuracy: 0.7266 - val_loss: 9.9319 - val_accuracy: 0.0871
Epoch 9/16
478/478 [=====] - 335s 700ms/step - loss: 1.4127 - accuracy: 0.7339 - val_loss: 10.1014 - val_accuracy: 0.0831
Epoch 10/16
478/478 [=====] - 324s 679ms/step - loss: 1.4024 - accuracy: 0.7397 - val_loss: 9.9654 - val_accuracy: 0.0888
Epoch 11/16
478/478 [=====] - 321s 671ms/step - loss: 1.4101 - accuracy: 0.7428 - val_loss: 9.9525 - val_accuracy: 0.0856
Epoch 12/16
478/478 [=====] - 322s 674ms/step - loss: 1.4169 - accuracy: 0.7452 - val_loss: 10.2386 - val_accuracy: 0.0884
Epoch 13/16
478/478 [=====] - 320s 669ms/step - loss: 1.4308 - accuracy: 0.7487 - val_loss: 10.0794 - val_accuracy: 0.0773
Epoch 14/16
478/478 [=====] - 336s 703ms/step - loss: 1.4418 - accuracy: 0.7502 - val_loss: 10.1888 - val_accuracy: 0.0795
Epoch 15/16
478/478 [=====] - 327s 685ms/step - loss: 1.4622 - accuracy: 0.7515 - val_loss: 10.1107 - val_accuracy: 0.0790
Epoch 16/16
478/478 [=====] - 325s 679ms/step - loss: 1.4800 - accuracy: 0.7528 - val_loss: 10.1137 - val_accuracy: 0.0783
```

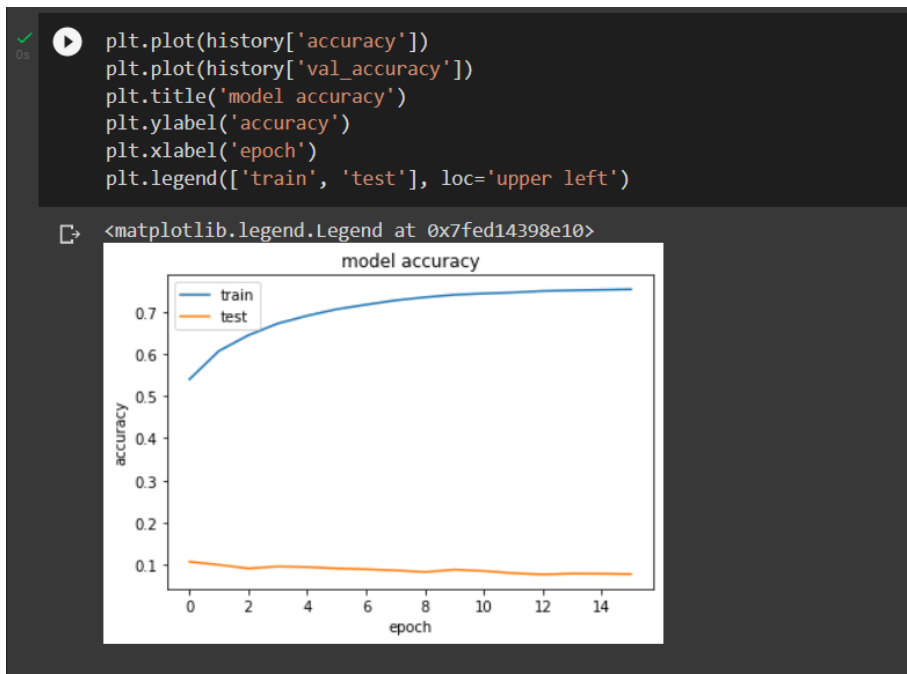
```
325s 679ms/step - loss: 1.4800 - accuracy: 0.7528 - val_loss: 10.1137 - val_accuracy: 0.0783
```

## Evaluating the next Word prediction

Let's have a look at how this model is going to behave based on its accuracy and loss change and as you can see the increasing model accuracy with epoch numbers. We feed the model data size of 128551 rows and trained models. As we can see increasing epoch size further not

going to help with the accuracy and also we're going to need better resources and data to train this model if we're to increase the accuracy. The accuracy attained on this model was 75.28% and lowest loss was visible at maximum value of epoch or last epoch.

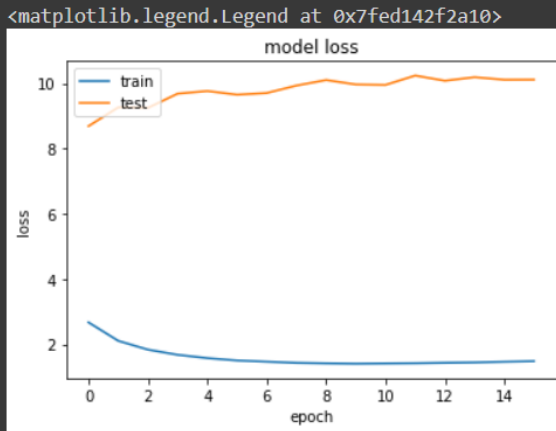
To further increase accuracy and reduce loss we can do few things. Some of the most important model options are made in the concept phase, however many model testing tasks tend to avoid critical testing in this phase. Usually a peer review panel will begin its efforts by explicitly accepting all the important assumptions made to find a conceptual model and devoting all its attention to building model and application categories. Also, peer review of the latest version of the almost complete model may find the basic concept model flawed. Finally, data should be evaluated at this point to ensure the availability of model development data, input parameters, and testing. The result of this process is the selection of a computer model that deals with problem detection, data availability, and transparency requirements.



```

✓ [35] plt.plot(history['loss'])
      plt.plot(history['val_loss'])
      plt.title('model loss')
      plt.ylabel('loss')
      plt.xlabel('epoch')
      plt.legend(['train', 'test'], loc='upper left')

```



## Predicting Using our Next word prediction model

First we need to convert our input into a vector that our model can read

```

def prepare_input(text):
    SEQUENCE_LENGTH = len(text)
    x = np.zeros((1, SEQUENCE_LENGTH, len(unique_words)))
    for t, char in enumerate(text):
        x[0,t,np.where(unique_words==char)[0]]=1

    return x

```

This will provide us a 3d numpy array same we used to train our model.

```
[52] prepare_input(inp.split()).shape
```

```
(1, 5, 9401)
```

```
[53] prediction=model.predict(prepare_input(inp.split()))
```

As we used the softmax layer at the end of our model output will be a probability distribution and so more probable words would be with the word with highest probability.

```
[54] prediction  
array([[1.8298991e-09, 1.4608927e-11, 8.0503811e-13, ..., 6.3834477e-08,  
        1.1796907e-05, 8.0671038e-13]], dtype=float32)
```

For our input

```
inp="In his eyes she eclipses"
```

Output would be

```
np.where(prediction == np.amax(prediction))  
(array([0]), array([504]))  
[57] unique_words[504]  
'and'
```

The output we received was by a model we trained on the data of a novel so we can personalize this if i could collect my own data from my chats or things i wrote like email or blogs.

## Literature review

Sr. No	Authors	Journal/Year	Methodology	Key findings
1	Partha Pratim Barman , Abhijit Boruah	8th International Conference on Advances in Computing and Communication (ICACC-2018)	In this paper, they present a Long Short Term Memory network (LSTM) model which is advanced model of Recurrent Neural Network(RNN) for instant messaging, where the goal is to predict next word to complete the sentence	They compared accuracy of their rnn and lstm models and were able to achieve accuracy of 72 %
2	Christopher Olah	Posted on August 27, 2015	Discussed how LSTM is functioned and how similar to rnn	LSTM is better at long term dependencies

## References

LSTM Networks : <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

NLP and deep learning :

<https://towardsdatascience.com/next-word-prediction-with-nlp-and-deep-learning-48b9fe0a17bf>

NWP model : <https://thecleverprogrammer.com/2020/07/20/next-word-prediction-model/>

Understanding LSTM :

<https://www.analyticsvidhya.com/blog/2021/08/predict-the-next-word-of-your-text-using-long-short-term-memory-lstm/>

Colab link

<https://colab.research.google.com/drive/1KfeYNduAwAPU7NX3323KxdpM59jzj1EV?usp=sharing>