PREDICTIVE TEXT FOR CONTEXT ANALYZATION

-Gadicherla Sameer
(01FB15ECS101 PES UNIVERSITY)

This blog mainy aims to create a system which, given input as a part of a complete English sentence, can predict the next word in the sentence or generate the next sequence of words using Markov Chains and RNNs.

This has been chosen because analyzing the context of any text for a given situation is done unbiased without any human intervention. Usually this is done character by character but we are doing it word by word , which implies the dependency on previous words is captured than the previous character, which helps in remembering the context better because we train the RNN on the words directly.

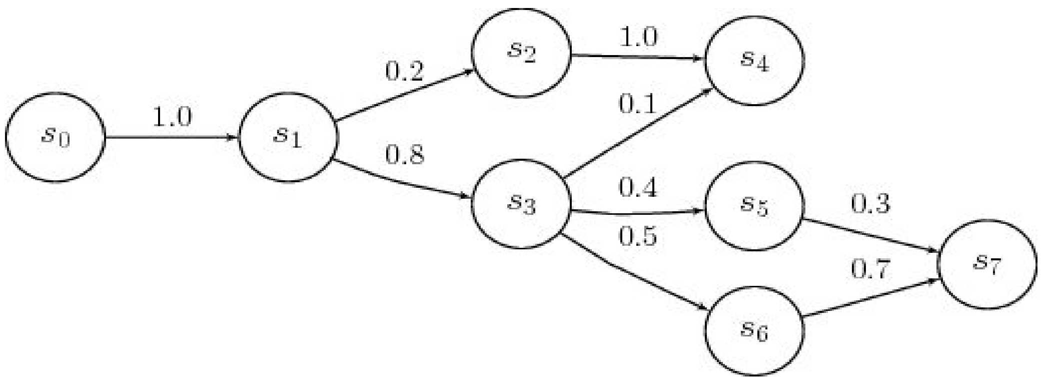
Thereby , to achieve this we have used two approaches :

1) The traditional markov chain approach which is not the main aim of the blog though.
2) RNN using the LSTM model

I'll just run through the basics of both markov chain and RNN first. Then let's proceed to the code.

MARKOV CHAIN :

It is a stochastic chain with markov property which states that **a stochastic process has the Markov property if the conditional probability distribution of future states of the process (conditional on both past and present states) depends only upon the present state, not on the sequence of events that preceded it.**


$$\begin{tikzcd}
 & & s_2 \arrow[r, "1.0"] & s_4 & \\
s_0 \arrow[r, "1.0"] & s_1 \arrow[ur, "0.2"] \arrow[dr, "0.8"'] & & & \\
 & & s_3 \arrow[uur, "0.1"] \arrow[r, "0.4"] \arrow[dr, "0.5"'] & s_5 \arrow[dr, "0.3"] & \\
 & & & s_6 \arrow[r, "0.7"'] & s_7
\end{tikzcd}$$

More in-detail about markov chains can be read here.

RNN and LSTM:

RNNs make use of sequential information. A traditional neural network assumes that all inputs (and outputs) are independent of each other.Since aim of our is to predict text, it is crucial to know the information from previous states, i.e. the previous words to predict the next word.RNNs are called recurrent because they perform the same task for every element of a sequence, with the output depending on the previous computations.RNNs thus have a "memory" element to capture information from previous calculations.

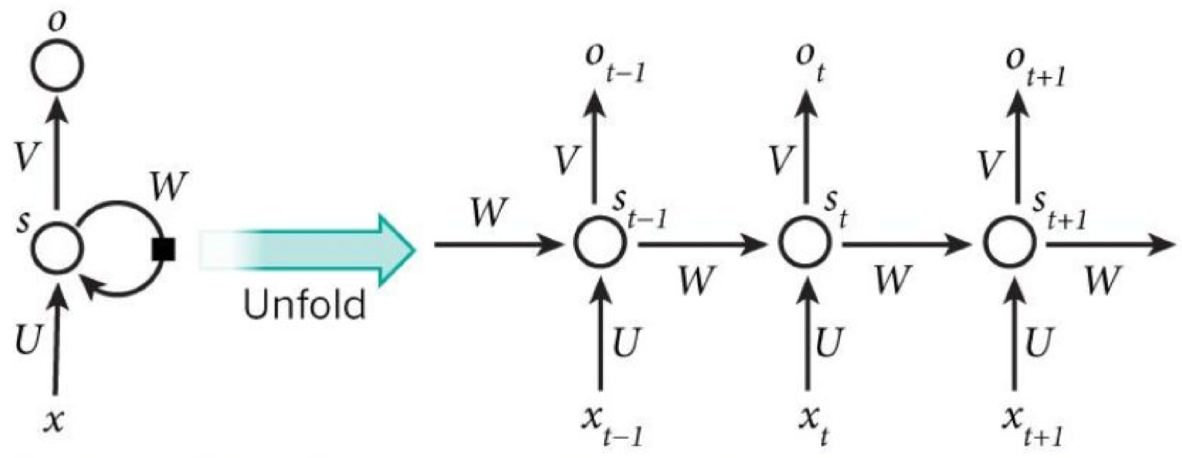


Figure 2 A typical RNN looks like one shown above. A RNN is unrolled (or unfolded) over time into a full network.

Need for LSTM and its advantages over RNN:

Two major problems with RNN: Vanishing and Exploding gradients.

- Vanishing Gradients: In traditional RNNs the gradient signal can be multiplied a large number of times by the weight matrix. Thus, the magnitude of the weights of the transition matrix can play an important role. If the weights in the matrix are small, the gradient signal becomes smaller at every training step, thus making learning very slow or completely stops it. This is called vanishing gradient.

- Exploding Gradients: refers to the weights in this matrix being so large that it can cause learning to diverge.

LSTM stands for Long Short Term Memory, is a special kind of RNN that learns long-term dependencies. The memory cell of LSTM is composed of four elements: input, forget and output gates and a neuron that connects to itself. To understand LSTM networks properly, I think Colah's blog is the right place .

So, basically going further , I am not going to stress much on the markov approach, I'll give a rough idea on how I used it to compare with RNNs and LSTMs.

DATASET SOURCE AND PREPROCESSING DONE:

1. Dataset source :
   - For markov chains :
     1. Corpus of Contemporary American English (COCA) from here

        - About Dataset: The Corpus of Contemporary American English (COCA) is the largest freely-available corpus of English, and the only large and balanced corpus of American English.
        - Attributes: We will be using a subset of COCA mainly from here. Hence, we will be using two, three, four and five-gram inputs to train the model each time.
        - Instance count: Each of the n-gram datasets contains the following number of n-grams:
          - 2-gram: 1,020,386
          - 3-gram: 1,020,010
          - 4-gram: 1,034,308
          - 5-gram: 1,044,269
        - This was the primary dataset for testing the goodness of fit for

every other dataset as it was exhaustive. We also trained using this dataset for next-word prediction as well as generating text.

1. Wikipedia Corpus
   - About dataset: This corpus contains the full text of Wikipedia, and it contains 1.9 billion words in more than 4.4 million articles.
   - Attributes: Wikipedia Corpus contains Wikipedia pages scraped and dumped into text files. Since it is only for testing, we used random articles with a total of 2286765 words.

2. RNN and LSTM:
   1. Narendra Modi speech in US : This has around 1000+ lines and is taken from a news website
   2. Narendra Modi speech on gst: This has around 2500+ lines and is also taken from a blog on web.
   3. Harry Potter Novel : It is the first book of the Harry potter series with has around 7000 lines and is has around 57 distinct words.

2. **Pre-processing steps performed:**

1. Wikipedia Corpus:
   - Read all the lines of the input file.
   - Remove all unwanted characters and punctuations.
   - Remove lines with very few words.
   - Convert everything to lower-case.
   - Run a sliding window on the words

2. Narendra Modi Speeches:
   - Same as for Wikipedia but further,
     - Distinct number of words are found
     - All words are represented as numbers

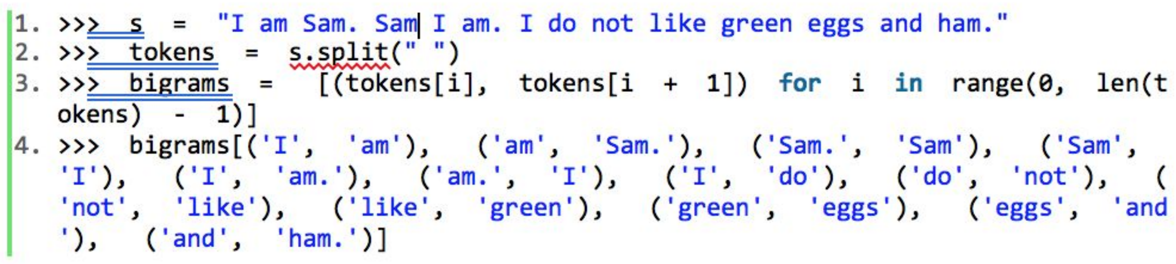
- All sentences are represented as a sequence of numbers
- All the sequences are padded to the largest sequence size with zeroes

Markov Chain(BASIC CODE WALKTHROUGH):

We create the training set by creating (n-gram , n+1 word , count) pairs and save these pairs in the SQL database.Then as said in pre-processing we also use the Wikipedia corpus, run. A sliding window and save the input,output pairs , which can also be used as training dataset but we are just using to test our model.So , in order to predict the next word , we give a simple SQL query to the database and return an ordered dictionary of word , count pairs which the client.py file will help us calculate the probability(if needed).We have used the database approach just because , it is very fast compared to vanilla lists , and practically impossible when the size of the text moves into GBs.

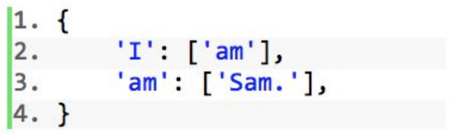
PSEUDO-CODE:

Create bi-grams:



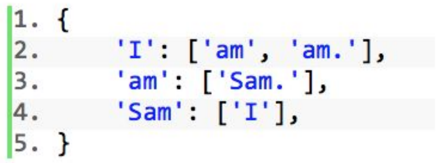
```
>>> s = "I am Sam. Sam I am. I do not like green eggs and ham."
>>> tokens = s.split(" ")
>>> bigrams = [(tokens[i], tokens[i + 1]) for i in range(0, len(tokens) - 1)]
>>> bigrams[('I', 'am'), ('am', 'Sam.'), ('Sam.', 'Sam'), ('Sam', 'I'), ('I', 'am.'), ('am.', 'I'), ('I', 'do'), ('do', 'not'), ('not', 'like'), ('like', 'green'), ('green', 'eggs'), ('eggs', 'and'), ('and', 'ham.')]
```

Markov Chain: `I am Sam.`



```
{
    'I': ['am'],
    'am': ['Sam.'],
}
```

Add `Sam I am.`

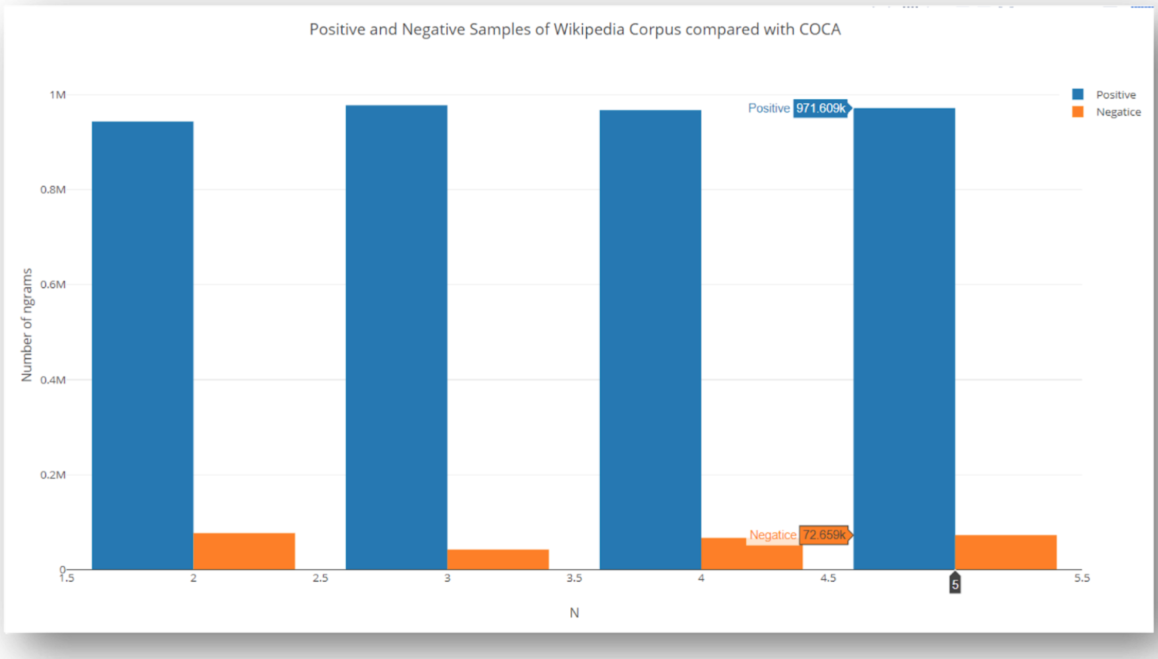


```
{
    'I': ['am', 'am.'],
    'am': ['Sam.'],
    'Sam': ['I'],
}
```

Thus, give an input `'am'`, we get `'Sam.'`

RESULTS:

Positive v/s negative samples in Coca dataset:

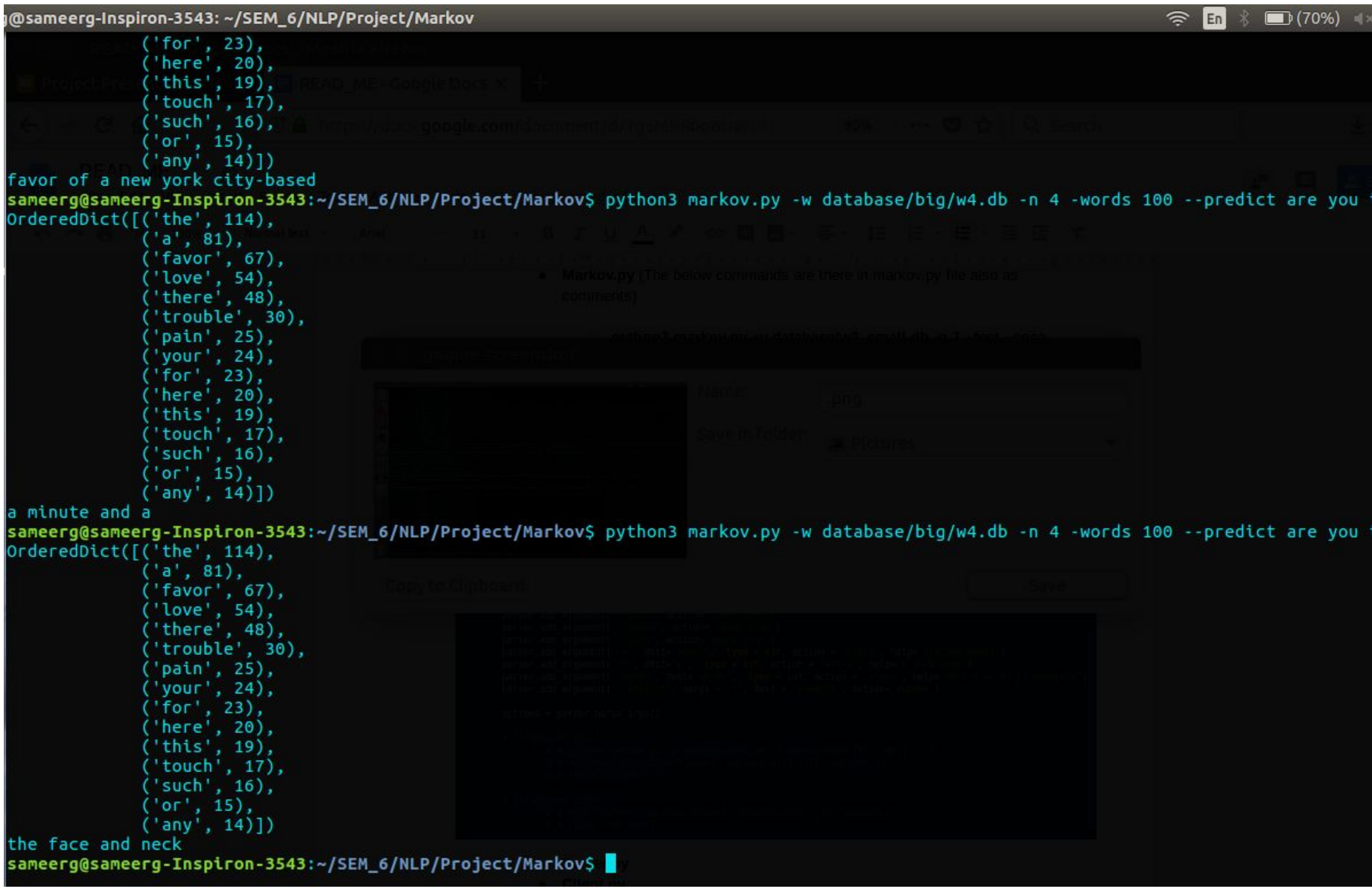


COCA when tested over wikipedia corpus , we found the following classification accuracies:

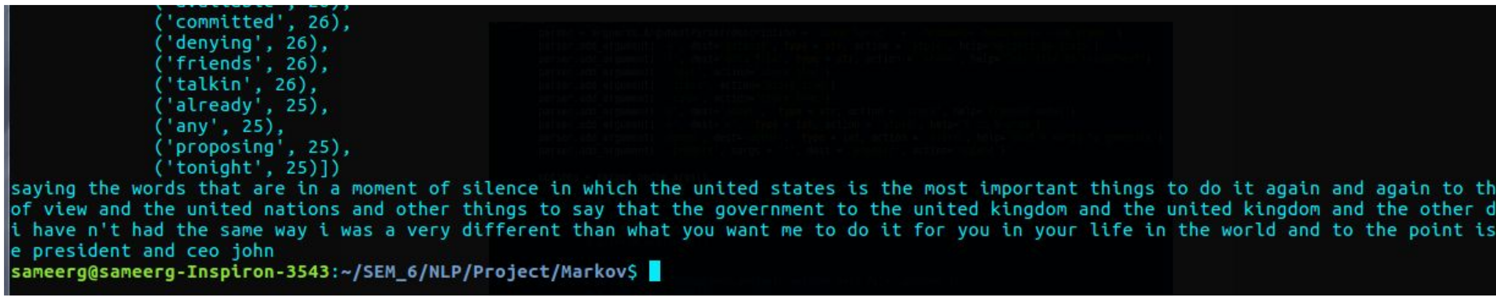
Table 1 Classification Accuracy and Error Rate for Wikipedia Corpus on COCA Dataset

N-gram	classification accuracy	Error Rate
2	92.44	7.6
3	95.84	4.16
4	93.53	6.47
5	93.042	6.958

Coming to the execution , we can see how markov model predicts a ordered dictionary of x words for a given n gram of words



```
@sameerg-Inspiron-3543: ~/SEM_6/NLP/Project/Markov
            ('for', 23),
            ('here', 20),
            ('this', 19),
            ('touch', 17),
            ('such', 16),
            ('or', 15),
            ('any', 14)])
favor of a new york city-based
sameerg@sameerg-Inspiron-3543:~/SEM_6/NLP/Project/Markov$ python3 markov.py -w database/big/w4.db -n 4 -words 100 --predict are you
OrderedDict([('the', 114),
            ('a', 81),
            ('favor', 67),
            ('love', 54),
            ('there', 48),
            ('trouble', 30),
            ('pain', 25),
            ('your', 24),
            ('for', 23),
            ('here', 20),
            ('this', 19),
            ('touch', 17),
            ('such', 16),
            ('or', 15),
            ('any', 14)])
a minute and a
sameerg@sameerg-Inspiron-3543:~/SEM_6/NLP/Project/Markov$ python3 markov.py -w database/big/w4.db -n 4 -words 100 --predict are you
OrderedDict([('the', 114),
            ('a', 81),
            ('favor', 67),
            ('love', 54),
            ('there', 48),
            ('trouble', 30),
            ('pain', 25),
            ('your', 24),
            ('for', 23),
            ('here', 20),
            ('this', 19),
            ('touch', 17),
            ('such', 16),
            ('or', 15),
            ('any', 14)])
the face and neck
sameerg@sameerg-Inspiron-3543:~/SEM_6/NLP/Project/Markov$
```



```
            ('committed', 26),
            ('denying', 26),
            ('friends', 26),
            ('talkin', 26),
            ('already', 25),
            ('any', 25),
            ('proposing', 25),
            ('tonight', 25)])
saying the words that are in a moment of silence in which the united states is the most important things to do it again and again to th
of view and the united nations and other things to say that the government to the united kingdom and the united kingdom and the other d
i have n't had the same way i was a very different than what you want me to do it for you in your life in the world and to the point is
e president and ceo john
sameerg@sameerg-Inspiron-3543:~/SEM_6/NLP/Project/Markov$
```

output for 100 words given a bigram , and trigram ,database w3.db is used, with starting by predicting the 3 gram , then last two words of the tri gram is used to predict the next 3-gram till 100 words are produced.If we see , actually the paragraph produced is a good one in context-wise also.

Now coming to LSTM approach , there are two ways of doing this ,

1. Training the model character by character.
2. Training the model word by word.

Design For the Character Model

In order to train our recurrent neural network, we use a large text file (we have used the Harry Potter books) as input. This large text file undergoes preprocessing and is stored in the form of two arrays X and Y defined as:

X:

which is a three dimensional array of the form X[i, t, char_indices[char]] where i is the index of each sentence, t is the index of each letter in each sentence and char_indices[char] indicates the letter encountered. Therefore, X is like the given input matrix.

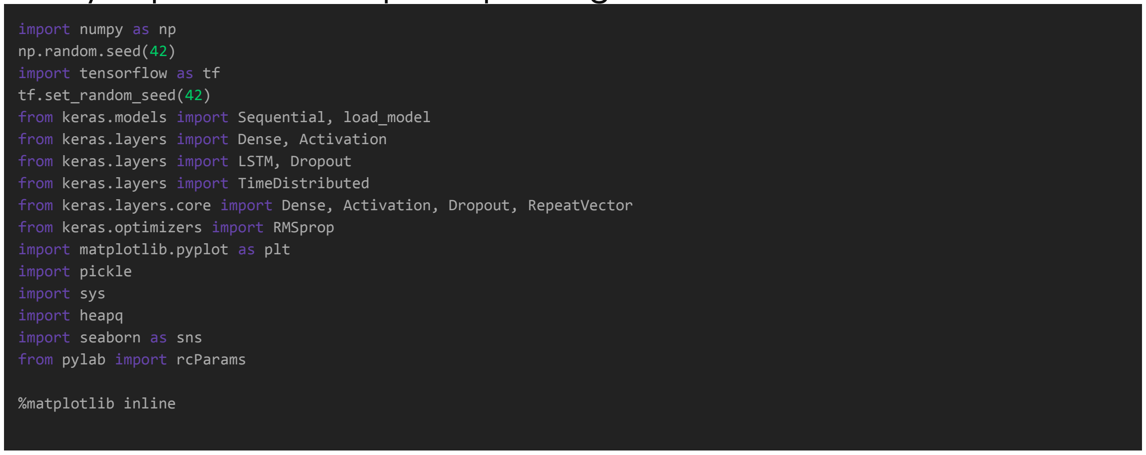
Y:

which is a two dimensional array of the form Y[i, char_indices[next_chars[i]]] where i is the index of each sentence and char_indices[next_chars[i]] indicates the letter which is encountered after encountering SEQUENCE_LENGTH (set to 40) characters of the sentence. Therefore, Y is like the expected output matrix.

We iterate through the text file at a step size of 3 characters, scanning through all characters, till SEQUENCE_LENGTH (store in list of sentences) and storing the next expected character after SEQUENCE_LENGTH (store in list of next_char). This is then used in the creation of the X and Y matrices.

We trained our model on the large novel , Harry potter and the chamber of secrets. And the model is as shown below. We use keras and matplotlib in basic.

Firstly import all the required packages:



```
import numpy as np
np.random.seed(42)
import tensorflow as tf
tf.set_random_seed(42)
from keras.models import Sequential, load_model
from keras.layers import Dense, Activation
from keras.layers import LSTM, Dropout
from keras.layers import TimeDistributed
from keras.layers.core import Dense, Activation, Dropout, RepeatVector
from keras.optimizers import RMSprop
import matplotlib.pyplot as plt
import pickle
import sys
import heapq
import seaborn as sns
from pylab import rcParams

%matplotlib inline
```

```
sns.set(style='whitegrid', palette='muted', font_scale=1.5)

rcParams['figure.figsize'] = 12, 5
```

Then open the dataset text file. And check the corpus length.

```
path = 'HP2.txt'
text = open(path).read().lower()
print('corpus length:', len(text))path = 'HP2.txt'
text = open(path).read().lower()
print('corpus length:', len(text))
```

Which comes out to be 21000+ characters.

```
corpus length: 21938
```

Printing the number of unique characters , it gave 40

```
chars = sorted(list(set(text)))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))

print("unique chars:", len(chars))
```

```
unique chars: 40
```

Next, let's cut the corpus into chunks of 40 characters, spacing the sequences by 3 characters. Additionally, we will store the next character (the one we need to predict) for every sequence:

```
SEQUENCE_LENGTH = 40
step = 3
sentences = []
next_chars = []
for i in range(0, len(text) - SEQUENCE_LENGTH, step):
        sentences.append(text[i: i + SEQUENCE_LENGTH])
        next_chars.append(text[i + SEQUENCE_LENGTH])
print('num training examples:', len(sentences))
```

We get 7300 training examples.

```
num training examples: 7300
```

It is time for generating our features and labels. We will use the previously generated sequences and characters that need to be predicted to create one-hot encoded vectors using the char_indices map:

```
X = np.zeros((len(sentences), SEQUENCE_LENGTH, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
   for t, char in enumerate(sentence):
       X[i, t, char_indices[char]] = 1
   y[i, char_indices[next_chars[i]]] = 1
```

Then we will see how the training set looks like

```
sentences[99]
```

It gives the following example:

```
>>>'a firm called grunnings, which made dril'
```

We expect a letter ‘l’ to make it ‘drill’ , And yes , it gave a ‘l’ as the expected next char.

```
next_chars[99]
>>>'l'
```

Then we create the one hot of this

```
X[0][0]
>>>array([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,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False])
```

And the one hot of the expected output will be :

```
Y[0]
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False])
```

Let’s see how big are the inputs in terms of the shape to our LSTM

```
X.shape
(7300, 40, 40)
y.shape
```

```
(7300, 40)
```

The model we're going to train is pretty straight forward. Single LSTM layer with 128 neurons which accepts input of shape (40 — the length of a sequence, 40 — the number of unique characters in our dataset). A fully connected layer (for our output) is added after that. It has 40 neurons and softmax for activation function:

```
model = Sequential()
model.add(LSTM(128, input_shape=(SEQUENCE_LENGTH, len(chars))))
model.add(Dense(len(chars)))
model.add(Activation('softmax'))
model.summary()
```

The summary of the model is given , as we can see it takes very huge parameters to train , almost 1 million . That's great , it learns more.

```
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 128)               86528
_________________________________________________________________
dense_2 (Dense)              (None, 40)                5160
_________________________________________________________________
activation_2 (Activation)    (None, 40)                0
=================================================================
Total params: 91,688
Trainable params: 91,688
Non-trainable params: 0
```

Our model is trained for 20 epochs using RMSProp optimizer and uses 5% of the data for validation:

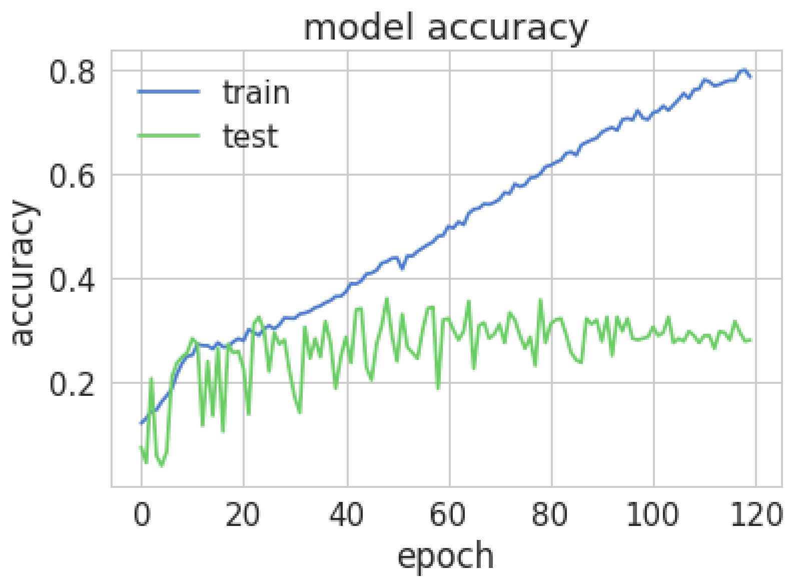
```
optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
history = model.fit(X, y, validation_split=0.05, batch_size=128, epochs=20,
shuffle=True).history
```

It took around 10 hours to train this for 120 epochs.

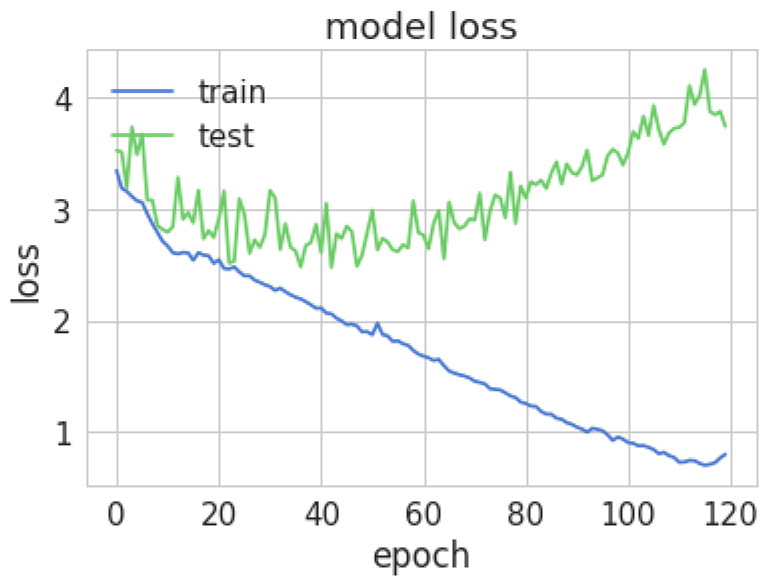
```
model.save('keras_model.h5')
pickle.dump(history, open("history.p", "wb"))
```

Now evaluating the model , we could achieve a accuracy of 78%.

```
plt.plot(history['acc'])
plt.plot(history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left');
```



```
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');
```



Then , we prepare input by creating

a function

```
def prepare_input(text):
        x = np.zeros((1, 40, len(chars)))

        for t, char in enumerate(text):
        x[0, t, char_indices[char]] = 1.

        return x
```

And get the required formatted input

```
array([[[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]]])
```

This below function samples the given data:

```
def sample(preds, top_n=3):
        preds = np.asarray(preds).astype('float64')
        preds = np.log(preds)
        exp_preds = np.exp(preds)
        preds = exp_preds / np.sum(exp_preds)

        return heapq.nlargest(top_n, range(len(preds)), preds.take)
```

And then we write two other functions to test our input which are self understandable.

```
def predict_completion(text):
        original_text = text
        generated = text
        completion = ''
        while True:
        x = prepare_input(text)
        preds = model.predict(x, verbose=0)[0]
        next_index = sample(preds, top_n=1)[0]
        next_char = indices_char[next_index]

        text = text[1:] + next_char
        completion += next_char

        if len(original_text + completion) + 2 > len(original_text) and next_char == ' ':
        return completion



def predict_completions(text, n=3):
        x = prepare_input(text)
        preds = model.predict(x, verbose=0)[0]
        next_indices = sample(preds, n)
        return [indices_char[idx] + predict_completion(text[1:] + indices_char[idx]) for idx
in next_indices]
```

Then , we create a test input of sequences as shown below and predict.

```
quotes = ["Mr. Dursley was the director of a firm called Grunnings, which made drills. He was
a big, beefy man with hardly any neck, although he did have a very large mustache.",
"At half past eight, Mr. Dursley picked up his briefcase, pecked Mrs. Dursley on the cheek,
and tried to kiss Dudley good-bye but missed,"
,"he'd gotten into terrible trouble for being found on the roof of the school kitchens"]

for q in quotes:
        seq = q[:40].lower()
        print(seq)
        print(predict_completions(seq, 5))
        print()

>>>mr. dursley was the director of a firm c
   ['lout... ', 'er ', 'as ', 'houlded ', 'it ']

   at half past eight, mr. dursley picked u
   ['itew ', 'phow ', 'she ', 'moffed ', 'whordind ']

   he'd gotten into terrible trouble for be
   ['chouddare. ', 'lhed ', 'theded ', 'eked ', 'kne ']
```

Design For the WORD Model:
We do normal pre-processing like said in the pre-processing section and further we have two approaches here , using the whole sentence to learn the model , and other is a trade-off between character and the whole sentence , where we create three word pairs and train the model where the first two words will be the input and the third word will be the output.

Three word model is explained below and the full sentence model , will be all same except the input creation part.

Load the required packages:

```
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding
```

We write a `genarate_sequence` function which is used for testing. We then load the US speech text file by Narendra Modi. I have a used a smaller dataset for present , we can even try this on larger datasets like before. But , I wanted to checkout , how well my model can speak for a nation(Just for fun).

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
         # encode the text as integer
         encoded = tokenizer.texts_to_sequences([in_text])[0]
         # pre-pad sequences to a fixed length
         encoded = pad_sequences([encoded], maxlen=max_length, padding='pre')
         # predict probabilities for each word
         yhat = model.predict_classes(encoded, verbose=0)
         # map predicted word index to word
         out_word = ''
         for word, index in tokenizer.word_index.items():
                 if index == yhat:
                         out_word = word
                         break
         # append to input
         in_text += ' ' + out_word
    return in_text
```

Then we use nltk for preprocessing:

```
import nltk
a = open("NAMO1.txt",encoding = "ISO-8859-1")
wtext = a.readlines()  ##all text in the textfile
while '\n' in wtext:
        wtext.remove('\n')
for i in wtext:
        j=i.strip('\n')
        wtext[wtext.index(i)] = j
x = ""
```

```
for i in wtext:
        x= x + " " + i
# vector representation
tokenizer = Tokenizer()
tokenizer.fit_on_texts([x])
encoded = tokenizer.texts_to_sequences([x])[0]
print("\n\n" , len(encoded))
```

All the above preprocessing after done , finally gives us a final vector representation of the whole text . Here `fit_on_texts` function is the one which counts the number of distinct words in the dataset and further orders them and gives each word a number.

We further create the 3 word input for our network. Before doing that we check our vocabulary size

```
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
```

It says 505 distinct words

```
 Vocabulary Size: 505
```

```
sequences = list()
for i in range(2, len(encoded)):
    sequence = encoded[i-2:i+1]
    sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
print("\n",sequences[0])
```

We see that the input is now a three word tuple as shown below

```
Total Sequences: 1192
[148, 149, 150]
```

Then we pad the sequences so that our model takes input of fixed size:

```
max_length = max([len(seq) for seq in sequences])
print(max_length)
print(type(sequences))

import numpy as np
sequences = pad_sequences(sequences, maxlen=max_length,padding='pre')

print('Max Sequence Length: %d' % max_length)
```

We can see the output below ,that it is fixed to size of three now.

```
3
<class 'list'>
Max Sequence Length: 3
```

Now , split the data into input and output sequences:

```
sequences = array(sequences)
X, y = sequences[:,:-1],sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
```

The model is now created as shown below;

```
model = Sequential()
model.add(Embedding(vocab_size,128, input_length=max_length-1))
model.add(LSTM(200))
model.add(Dense(vocab_size, activation='softmax'))
print(model.summary())

Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 2, 128)            64640
_________________________________________________________________
lstm_1 (LSTM)                (None, 200)               263200
_________________________________________________________________
dense_1 (Dense)              (None, 505)               101505
=================================================================
Total params: 429,345
Trainable params: 429,345
Non-trainable params: 0
_________________________________________________________________
None
```

I have used the adam and the rms prop optimizer , for the model with 0.05% validation data split and trained it for 150 epochs. I have achieved different accuracies for different combinations which I will put up in the end.

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

history=model.fit(X, y, validation_split=0.05,epochs=150, verbose=1, shuffle = True)
```

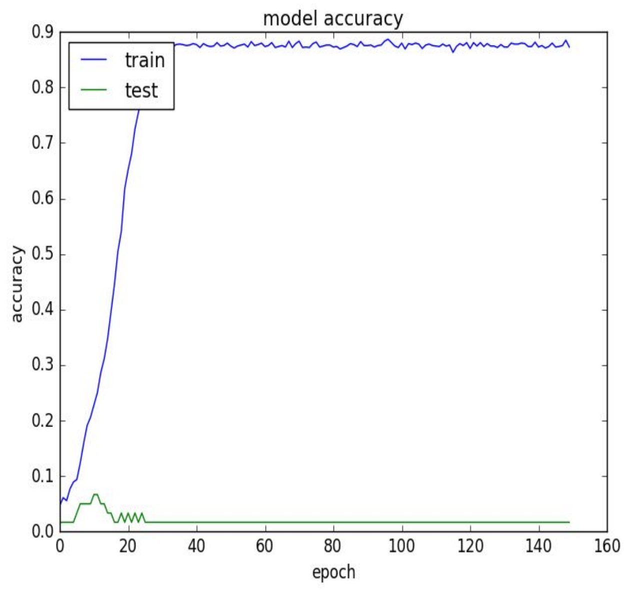
The output during the last five epochs is shown.

```
Epoch 146/150
1132/1132 [==============================] - 1s 777us/step - loss: 0.2131 - acc: 0.8728 -
val_loss: 11.1676 - val_acc: 0.0167
Epoch 147/150
1132/1132 [==============================] - 1s 851us/step - loss: 0.2111 - acc: 0.8737 -
val_loss: 11.1835 - val_acc: 0.0167
Epoch 148/150
1132/1132 [==============================] - 1s 972us/step - loss: 0.2141 - acc: 0.8754 -
val_loss: 11.2105 - val_acc: 0.0167
Epoch 149/150
1132/1132 [==============================] - 1s 766us/step - loss: 0.2107 - acc: 0.8852 -
```

```
val_loss: 11.1748 - val_acc: 0.0167
Epoch 150/150
1132/1132 [==============================] - 1s 787us/step - loss: 0.2152 - acc: 0.8728 -
val_loss: 11.2066 - val_acc: 0.0167
```

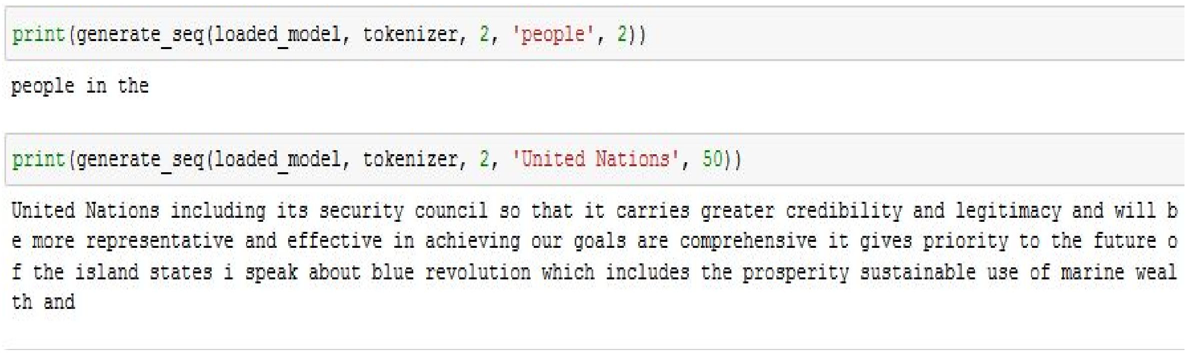
Validating the model:

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



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

As the training set is small for present, we aren’t achieving higher validation accuracies . But if trained on dataset with more validation split percentage we can achieve higher val_acc.



```
print(generate_seq(loaded_model, tokenizer, 2, 'people', 2))
```

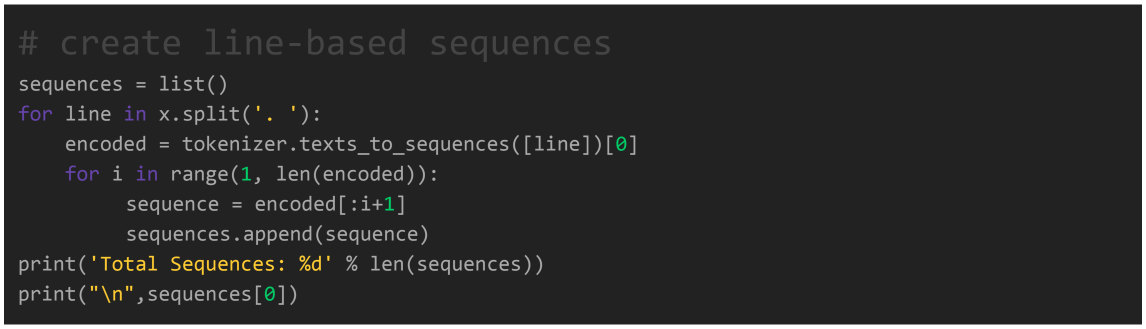
```
people in the
```

```
print(generate_seq(loaded_model, tokenizer, 2, 'United Nations', 50))
```

```
United Nations including its security council so that it carries greater credibility and legitimacy and will b
e more representative and effective in achieving our goals are comprehensive it gives priority to the future o
f the island states i speak about blue revolution which includes the prosperity sustainable use of marine weal
th and
```

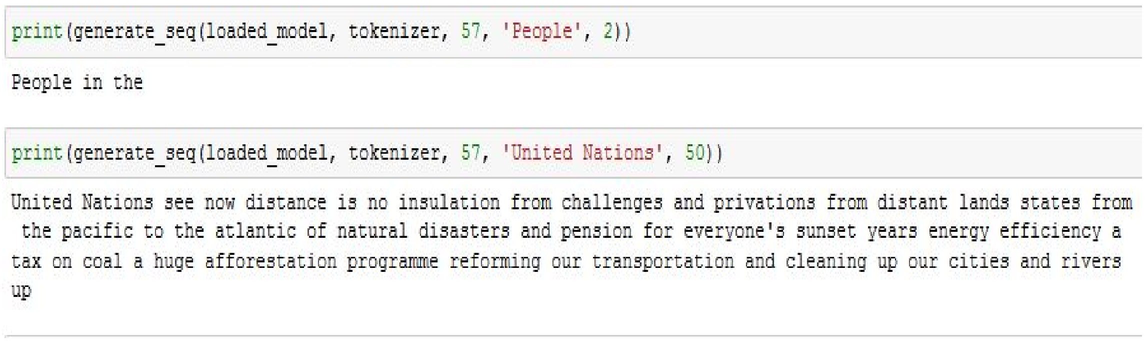
We can see that , the above paragraph generated makes very much good sense and is close to the one which the prime minster has spoken.

Coming to the whole sentence approach , we create line based sequences instead of three word. Everything else remains same as the three word approach.



```
# create line-based sequences
sequences = list()
for line in x.split('. '):
    encoded = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(encoded)):
        sequence = encoded[:i+1]
        sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
print("\n",sequences[0])
```

The below is the output for the sentence based approach.



```
print(generate_seq(loaded_model, tokenizer, 57, 'People', 2))
```

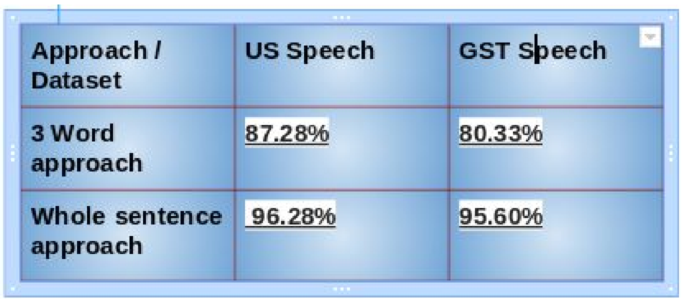
```
People in the
```

```
print(generate_seq(loaded_model, tokenizer, 57, 'United Nations', 50))
```

```
United Nations see now distance is no insulation from challenges and privations from distant lands states from
 the pacific to the atlantic of natural disasters and pension for everyone's sunset years energy efficiency a
tax on coal a huge afforestation programme reforming our transportation and cleaning up our cities and rivers
up
```

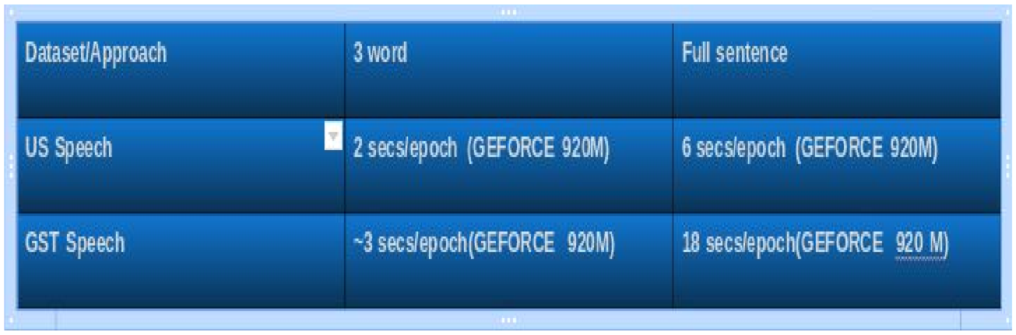
We see that both the models have pretty much given a meaningful set of words as output. But the sentence approach is close the actual data but the three word trade off looks very meaningful too.

Similarly we can use the GST speech also. Below are the accuracies which I have achieved using the above model over the two datasets.



Approach / Dataset	US Speech	GST Speech
3 Word approach	87.28%	80.33%
Whole sentence approach	96.28%	95.60%

And all this training has been done on my laptop (which is not recommended unless you have a good GPU for larger datasets).



Dataset/Approach	3 word	Full sentence
US Speech	2 secs/epoch (GEFORCE 920M)	6 secs/epoch (GEFORCE 920M)
GST Speech	~3 secs/epoch(GEFORCE 920M)	18 secs/epoch(GEFORCE 920 M)

Concluding this blog , I want to tell that :

- The RNN is more accurate than the n-gram model because it can retain more history than just the previous 2 tokens and using word vectors instead of discrete tokens for each word allows it to generalize and make predictions from similar words, not just exact matches.
- The RNN takes longer to train (e.g. 8 hours vs 8 seconds, about 4000x longer), but would easily be able to fit into memory for applications like phone text entry - it also performs word predictions a bit slower than the n-gram, but not enough to be noticeable to an end-user.
- As we can see that the outputs of word by word model were very very better than character by character.
- And the training time is also less for the word by word model.
- The two tradeoffs that is the sentence and the three word approaches in word by word models were compared in the previous sections accordingly to the dataset.

My motivation for doing this blog is train the architecture on musical compositions of a particular singer and try to generate predictive musical notes which can uniquely represent his compositions.It's just an extension to this approach . I have my plans on how to do this , if you have any please post it out in the comments section(sameergadicherla56@gmail.com).

References:

1. https://en.wikipedia.org/wiki/Language_model
2. Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. Learning long-term dependencies with gradient descent is difficult. Neural Networks, IEEE Transactions on, 5(2):157–166, 1994.
3. Dauphin, Yann N, de Vries, Harm, Chung, Junyoung, and Bengio, Yoshua. Rmsprop and equilibrated adaptive learning rates for non-convex optimization. arXiv preprint arXiv:1502.04390, 2015.