Text Prediction using Recurrent Neural Network

Table of Contents :

Chapter 1. : Introduction

Text prediction is a task that we use so often in our lives that we've taken it for granted. From the auto-fill feature in our messaging apps to search engines predicting search terms, text prediction technology saves our time and helps us to make our lives easier. It also links into other tasks such as text generation, which can eventually be used to write stories or longer paragraphs.

In this notebook, I will be using the **Brown corpus** to create and train a LSTM model to predict the next word in a sentence.

1.1 History

The Brown Corpus

The **Brown Corpus of Standard American English** was the first of the modern, computer-readable, general corpora. It was compiled by W.N. Francis and H. Kucera, Brown University, Providence, RI.

The corpus consists of one million words of American English texts printed in 1961. The texts for the corpus were sampled from 15 different text categories to make the corpus a good standard reference. Today, this corpus is considered small, and slightly dated. The corpus is, however, still used. Much of its usefulness lies in the fact that the Brown corpus layout has been copied by other corpus compilers. The **LOB corpus (British English) and the Kolhapur Corpus** (Indian English) are two examples of corpora made to match the Brown corpus. The availability of corpora which are so similar in structure is a valuable resource for researchers interested in comparing different language varieties, for example.

For a long time, the Brown and LOB corpora were almost the only easily available computer-readable corpora. Much research within the field of corpus linguistics has therefore been made using these data. By studying the same data from different angles, in different kinds of studies, researchers can compare their findings without having to take into consideration possible variation caused by the use of different data.

At the University of Freiburg, Germany, researchers are compiling new versions of the LOB and Brown corpora with texts from 1991. This will undoubtedly be a valuable resource for studies of language change in a near diachronic perspective.

The Brown corpus consists of 500 texts, each consisting of just over 2,000 words. The texts were sampled from 15 different text categories.

1.2 Steps Involved in the entire process :

1.2.1 Loading the Data

The data is in the form of tokenized sentences, and I'll store it into a data-frame "df".After storing sentences in the data-frame, I had added a **second column called sentences**, which is just the regular, untokenized sentences. Here, we can see the very first sentences of **The Fulton County Grand Jury said Friday**.

1.2.2 Preprocessing

	tokenized_sentences	sentences
0	[The, Fulton, County, Grand, Jury, said, Frida...	The Fulton County Grand Jury said Friday an in...
1	[The, jury, further, said, in, term-end, prese...	The jury further said in term-end presentments...
2	[The, September-October, term, jury, had, been...	The September-October term jury had been charg...
3	['', Only, a, relative, handful, of, such, rep...	`` Only a relative handful of such reports was...
4	[The, jury, said, it, did, find, that, many, o...	The jury said it did find that many of Georgia...

Here, we can see the very first sentences of **The Fulton County Grand Jury said Friday**.

1.2.3 Cleaning the Data

Now, I'll set all the **words to lower case**, so that our model won't differentiate things like Jury and jury. The next step involves vectorizing our words and encoding them in a way that an RNN can read and learn from.

1.2.4 Vectorizing Words

An **RNN can't take in a variable-length input like a sentence, so I need to encode my inputs into vectors**. I'll be using the **Word2Vec embedding**, which is just a shallow, **two-layer neural network that's trained to represent sentences as high-dimensional vectors which preserve the linguistic context of each sentence**. I going to use a library called **"gensim"**, which contains tons of useful functions for natural language processing and word encoding. Using gensim, I can train my Word2Vec model to encode words as vectors.
I've trained my model, I can output a sample vector. Below is Word2Vec's encoding of the word "city."

```
In [11]: print(w2v_model.wv['city'])
```

```
[ 3.26633245e-01 -6.68212533e-01 -9.12855625e-01  3.45170319e-01
 -1.11775899e+00  1.54762924e-01 -7.96462476e-01 -1.03137410e+00
 -6.98018149e-02  1.10705018e+00 -3.42731029e-01  6.56914711e-01
  1.38541591e+00  1.50882185e+00 -3.74704033e-01  3.19252223e-01
 -4.22558576e-01 -8.31875324e-01 -1.45990992e+00  1.11133409e+00
 -7.09058419e-02 -3.18833441e-01 -1.88513899e+00 -1.60529697e+00
  3.62765491e-01  7.54340112e-01 -2.58011055e+00  1.51000094e+00
 -3.66873622e-01 -2.37608761e-01  1.29750979e+00 -8.84758830e-01
 -7.49274671e-01  5.63361406e-01 -7.03528464e-01 -3.57520580e-01
 -1.13601160e+00  6.12639785e-01 -5.37439525e-01  5.61464548e-01
 -1.34325415e-01 -5.59870124e-01  9.74624217e-01 -2.20261991e-01
  7.67108142e-01  3.66510078e-02 -2.97020972e-01 -8.16036761e-01
  1.32485712e-02  4.15640235e-01 -5.89081585e-01  6.46569878e-02
  2.77508020e-01  2.42881641e-01  1.71544528e+00  6.84701502e-01
 -3.96226317e-01  4.61708933e-01  2.83550292e-01 -4.10739362e-01
  2.17882261e-01 -5.61081134e-02 -8.94377828e-01 -6.07130051e-01
  2.50020385e-01  1.23261392e+00 -2.22937718e-01 -1.82630471e-03
  3.76900405e-01  1.51798595e-02 -5.76438010e-02 -5.16988337e-01
  6.57830596e-01 -9.35891211e-01 -4.70166624e-01 -1.09210622e+00
  5.54458976e-01  1.32321790e-01 -4.47877526e-01 -1.30486572e+00
 -4.80912209e-01 -1.11723602e+00  1.42464833e-02  4.80527967e-01
 -5.74650884e-01 -2.02732682e+00  4.16941136e-01  4.33599800e-01
  1.94012690e+00  1.65297055e+00  5.62272966e-01 -7.15475738e-01
  1.74409544e+00  9.64795053e-01 -1.01150882e+00 -7.19265699e-01
  6.15103364e-01  1.34066761e-01 -6.73386753e-02  3.89235169e-01]
```

To get an idea of what Word2Vec is doing, we can call most_similar on a few sentences. This function tries to find the words with the highest similarity using the word vectors of the words. This may help us better understand what the vectorization is actually doing.

```
In [12]: print('School:', w2v_model.wv.most_similar('school', topn = 3))
         print('Smell:', w2v_model.wv.most_similar('smell', topn = 3))
         print('Election:', w2v_model.wv.most_similar('election', topn = 3))
```

```
School: [('college', 0.7191208004951477), ('schools', 0.7189204692840576), ('university', 0.6236478090286255)]
Smell: [('snake', 0.7361461520195007), ('flash', 0.6988798379898071), ('skin', 0.6638326644897461)]
Election: [('council', 0.773226261138916), ('campaign', 0.7560018301010132), ('report', 0.7070152759552002)]
```

To get an idea of what Word2Vec is doing, we can call most_similar on a few sentences. This function tries to find the words with the highest similarity using the word vectors of the words. This may help us better understand what the vectorization is actually doing.
As we can see, Word2Vec has learned something about the context of each word, as the outputs of most_similar seem to somewhat match up with each word. However, Word2Vec isn't perfect - some suggestions for 'Smell' and 'Election' don't make that much sense.

Visualizing Word Vectorization with Parts of Speech

Let's see if we can visualize how Word2Vec is encoding our words. We'll need to reduce the dimensionality of each word's vector to the plot, so we are using PCA.
With our reduced data, let's now visualize our words using a scatterplot colored by parts of speech. Let's see how good of a job of Word2Vec did. First, we'll need to get a list of unique words, as there's going to be a lot of repeated words in our dataset. To do so, we can use **gensim's wv.vocab,** which will return the list of unique words in the data. **Number of unique words: 14221**

Now, let's get the parts of speech, for each word, so we can color our data points. Luckily, nltk has **averaged_perceptron_tagger**, which can tag words with their part of speech. We can now go through and add a new column to our dataset, df_tag, the part of speech for each word.
We have the part of speech for each word, we can create a color map. There seems to be a lot of categories for parts of speech, so we can just generate a random color for each part of speech that hasn't

been assigned a color. With our finished color map, we can go ahead and plot our data. I've annotated a small sample of words to help with the visualization.

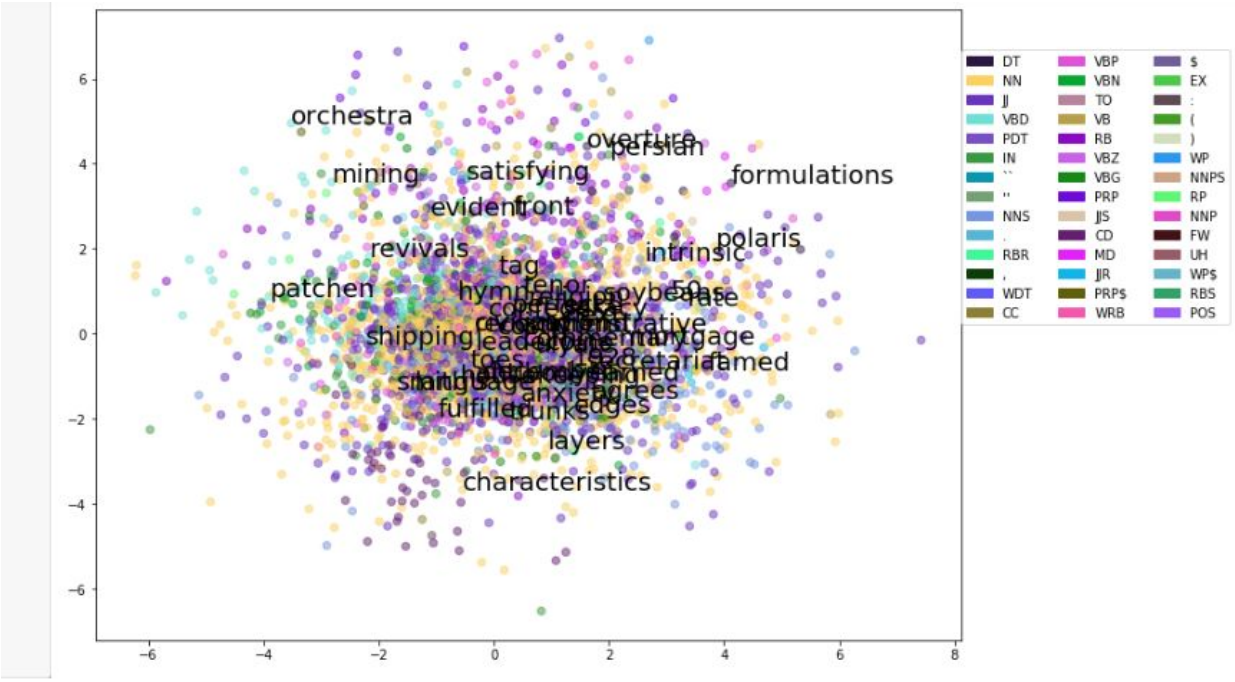
Additional information--

IN: Preposition or subordinating conjunction

NN: Noun, singular or mass

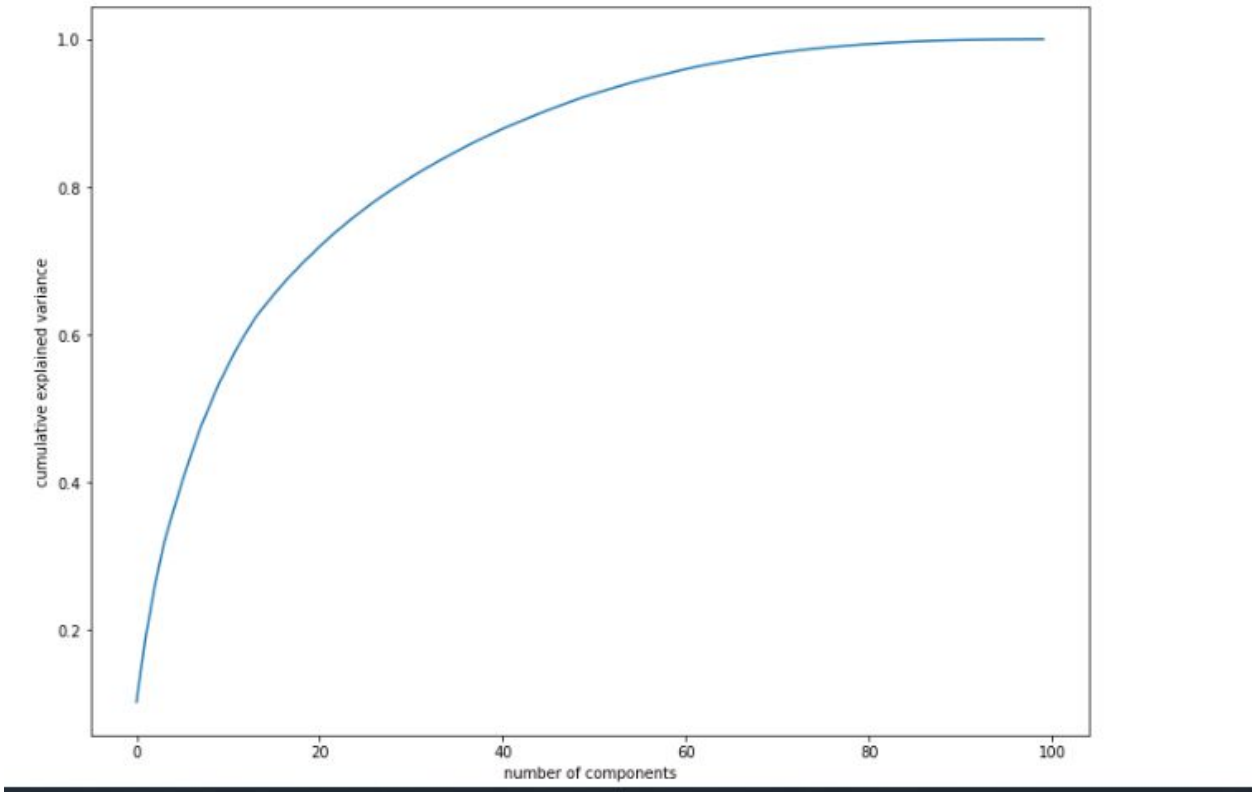
DT: determiner

JJ: adjective or numeral, ordinal



Nltk also has a help function which describes the part of speech a word belongs to.

However, this scatter plot doesn't seem to show anything really useful. It can't seem to find any meaningful patterns in the data. This isn't all that unreasonable **With 100-dimensional data being reduced to two dimensions**, we're certainly going to lose a lot of explained variance. Let's **plot the number of components against the explained variance** to check:



Our plot seems to verify our suspicions i.e we simply can't see anything meaningful because **two dimensions only explain 10-15% of our variance**.

Sliding Window

What is a sliding window?
A sliding window is a number of "latest" words that are actually passed into our neural network. Suppose the value was 5. Then, if we give our neural network a sentence like "The color of my house is red. What color is your", we will pass into our neural network: [".", "What", "color", "is", "your"], an array of length 5. This is done so that we have a standard number of inputs that we give our neural network.

We'll pick a sliding window size of 10. Also, some methods are defined here to grab the word vectors and create masks, which make sure that all sentences are longer than our sliding window's value, and that all the vocabulary in the sentence is part our Word2Vec vocabulary.

	tokenized_sentences	sentences	first_10_words	next_word_after_10_words	class_vec
0	[the, fulton, county, grand, jury, said, frida...	The Fulton County Grand Jury said Friday an in...	[[-1.0120853, -1.4098665, 0.20928295, 0.693554...	atlanta's	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
3	[``, only, a, relative, handful, of, such, rep...	`` Only a relative handful of such reports was...	[[1.4401522, 1.841007, -0.07958667, 1.2485093,...	,,	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
4	[the, jury, said, it, did, find, that, many, o...	The jury said it did find that many of Georgia...	[[-1.0120853, -1.4098665, 0.20928295, 0.693554...	registration	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
5	[it, recommended, that, fulton, legislators, a...	It recommended that Fulton legislators act `` ...	[[1.4286807, -0.5122561, -0.43811736, 1.323680...	laws	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
6	[the, grand, jury, commented, on, a, number, o...	The grand jury commented on a number of other ...	[[-1.0120853, -1.4098665, 0.20928295, 0.693554...	'	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...

Our pre-processing is looking great! Let's move on to the RNN.

The RNN

Now, we're going to create our RNN, which will take in our input vector and predict the next word. This is a classification task, as we need to predict one word out of the individual word classes in our entire vocabulary.

LSTM

What is going on here?

We're adding layers to our sequential model. Firstly, we'll use **3 LSTM layers** to create a multi-dimensional recurrent neural network. LSTMs are useful for us because they allow us to go back a certain amount of words once we have a result and continue training. Since we set a sliding window earlier, we can give our **LSTM an input size with the width equal to the sliding window. The LSTM will then train one word in the past, get a result, and then train the same result with two words into the past**, etc... This is incredibly helpful to us because we want to be able to look a certain depth into the past to predict our next word.

Out of experimentation, we found that using many LSTM layers is better, so we've added three. A Dropout layer helps us with overfitting—it takes out a certain amount of random terms from our LSTM system in order to cross-validate on them. The Dense layer and Activation allow us to actually get a probability for each word. We do want this because we want to predict more than one word sometimes (such as in iOS, where three words are shown above the keypad), and so knowing the probabilities of each word will help us sort the word predictions from most likely to least likely.

```
WARNING:tensorflow:From C:\Users\Angarak James\Anaconda3\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:
1630: calling BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated an
d will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, None, 128)         117248
_________________________________________________________________
lstm_1 (LSTM)                (None, None, 128)         131584
_________________________________________________________________
lstm_2 (LSTM)                (None, 128)               131584
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 14221)             1834509
_________________________________________________________________
activation (Activation)      (None, 14221)             0
=================================================================
Total params: 2,214,925
Trainable params: 2,214,925
Non-trainable params: 0
_________________________________________________________________
```

Now, we can split our dataset into train and test sets, and then format the data such that it's the right shape for our network.

Evaluation

Let's evaluate our model. The following methods help split up a normal sentence into tokens. The predict method is useful to define now because we will use it quite a bit to test.
Why are so many words being repeated?

After a while the model gets lost in its own wording. Since it only takes 10 words in the past, if all of the 10 words are the model's own words, then naturally, we can expect that the model will become less and less guided over time.

This is also a result of underfitting. In order to predict more successfully, we may, in the future, have a larger sliding window, or a flexible sliding window. We would definitely need a far larger dataset to make real-world predictions, like iOS does. But for our small(ish) datasets and small(ish) epochs, it looks like we've done fine.

By plotting accuracy and loss against the progression of epochs in our model training history, we can estimate the optimal number of epochs

Training Other Corpora

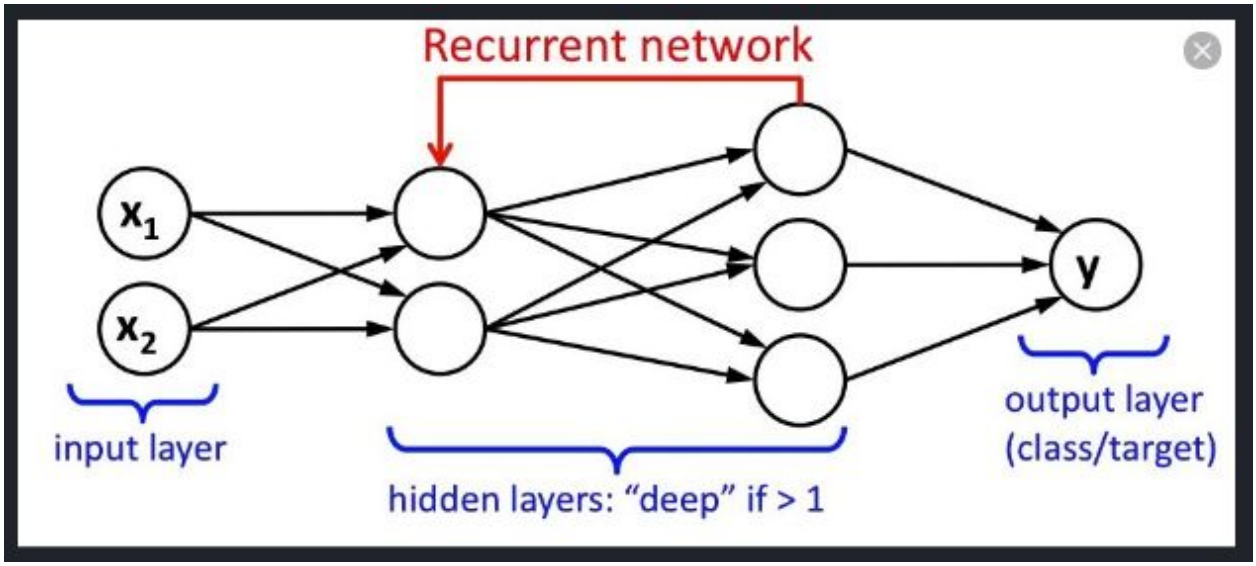
This is the basic structure of our program, all in one method. All of the hyper-parameters are parameters of this method. We can test our model trained on different text corpora. Here we've done the Brown corpus (**compilation of text from 500 sources of varied genres**), the Web Text corpus (**more informal text from conversations, movie scripts, reviews, and advertisements**), and the Inaugural Address corpus (collection of text from the 55 presidential addresses). More options of text corpora can be found here: https://www.nltk.org/book/ch02.html.

Literature Survey

Recurrent Neural Network

Recurrent Neural Network(RNN) is a type of **Neural Network** where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is the Hidden state, which remembers some information about a sequence.

RNN has a **“memory”** which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

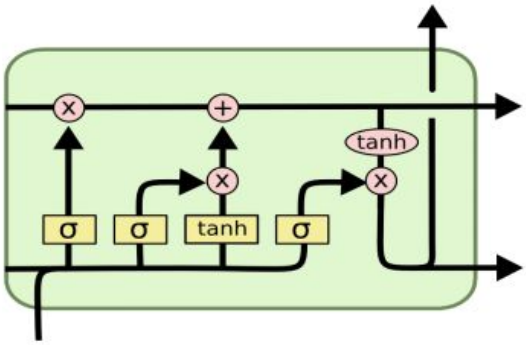


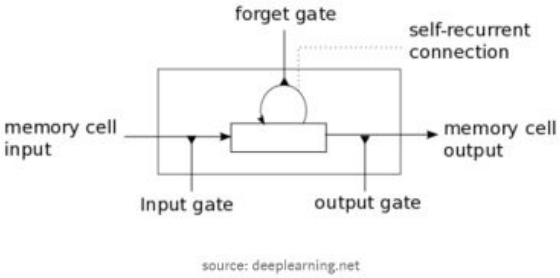
LSTM Long Short Term Memory

Long Short Term Memory(LSTM) is a kind of recurrent neural network. In RNN output from the last step is fed as input in the current step. LSTM was designed by Hochreiter & Schmidhuber. It tackled the problem of long-term dependencies of RNN in which the RNN cannot predict the word stored in the long term memory but can give more accurate predictions from the recent information. As the gap length increases RNN does not give an efficient performance. LSTM can by default retain the information for a long period of time. It is used for processing, predicting and classifying on the basis of time-series data.

Structure Of LSTM:

LSTM has a chain structure that contains four neural networks and different memory blocks called **cells**.

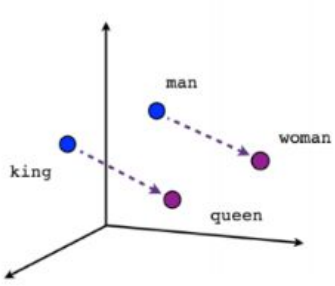




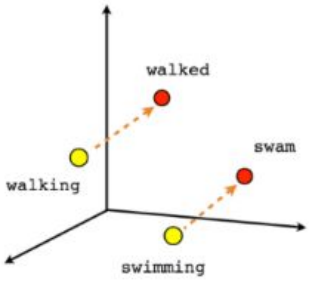
Word2Vec

Word2vec is a two-layer neural net that processes text by “vectorizing” words. Its input is a text corpus and its output is a set of vectors: feature vectors that represent words in that corpus. While Word2vec is not a deep neural network, it turns text into a numerical form that deep neural networks can understand.

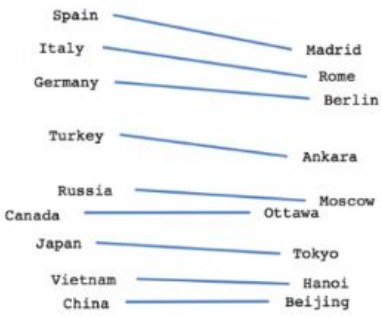
The purpose and usefulness of Word2vec is to group the vectors of similar words together in vector space. That is, it detects similarities mathematically. Word2vec creates vectors that are distributed numerical representations of word features, features such as the context of individual words. It does so without human intervention.



Male-Female



Verb tense



Country-Capital

Chapter 2.: Discussion on Implementation of Results

Our model has an accuracy of **14.72%**. I have collected this result on the training phase, hope for the testing phase and by tunning some more of the parameters, we can expect a higher accuracy.

Why are so many words being repeated?

After a while the model gets lost in its own wording. Since it only takes 10 words in the past, if all of the 10 words are the model's own words, then naturally, we can expect that the model will become less and less guided over time.

Chapter 3.: Conclusion and Future Enhancements

It seems like the model found the most success with the Brown corpus, although the model's improvement seems to slow after 6-8 epochs for all of the corpora. This is also a result of **underfitting**. In order to predict more successfully, we may, in the future, have a **larger sliding window, or a flexible sliding window**. We would definitely need a far larger dataset to make real-world predictions as iOS or Android does. But **for our small datasets and small epochs, it looks like we've done fine.**

References

Annotated Bibliography:

1) What are RNNs and LSTMs:

Recurrent Neural Networks (An Introduction to Concept) (https://www.youtube.com/watch?v=6niqTuYFZLQ) This video offers a basic introduction to recurrent neural networks (RNNs). Although it's rather long, the content from 9:00 - 22:00 is pretty useful.

Understanding LSTM Networks (https://colah.github.io/posts/2015-08-Understanding-LSTMs/) This article explains what RNNs and LSTMs are and how they work.

Why are deep neural networks hard to train? (http://neuralnetworksanddeeplearning.com/chap5.html) This article explains the exploding and unstable gradient problems that come with a deep recurrent neural network.

2) Finding Text Datasets:

Accessing Text Corpora and Lexical Resources (https://www.nltk.org/book/ch02.html) along with (https://www1.essex.ac.uk/linguistics/external/clmt/w3c/corpus_ling/content/corpora/list/private/brown/brown.html). This page describes the different text corpora, or text collections, available in the NLTK library. We trained our model using the Brown corpus, which compiles text from 500 sources including news, mystery, and fiction texts, and the Gutenburg Corpus, which contains a small collection of electronic book texts.

3) Processing Text and Converting Words to Vectors:

Tokenization and Parts of Speech(POS) Tagging in Python's NLTK library (https://medium.com/@gianpaul.r/tokenization-and-parts-of-speech-pos-tagging-in-pythons-nltk-library-2d30f70af13b) This page describes NLTK's part of speech tagger and provides a list of the different parts of speech. We used this in our visualization of Word2Vec to color our plot by parts of speech.

Word embeddings: how to transform text into numbers (https://monkeylearn.com/blog/word-embeddings-transform-text-numbers/) This page compares uses of word vectors versus one-hot encoders, suggests how to evaluate their reasonableness, and explains how word vectors are created.

models.word2vec – Word2vec embeddings (https://radimrehurek.com/gensim/models/word2vec.html) This page briefly mentions different methods of word embeddings and gives examples of how to use Word2Vec. Vector Representations of Words (https://www.tensorflow.org/tutorials/representation/word2vec) This tutorial describes the uses of word embedding, how the Word2Vec model works, and how to implement a model.

How to get started with Word2Vec — and then how to make it work (https://medium.freecodecamp.org/how-to-get-started-with-word2vec-and-then-how-to-make-it-work-d0a2fca9dad3) This article explains what Word2Vec is and how to use the Gensim implementation.

Predicting a word using Word2vec model (https://datascience.stackexchange.com/questions/9785/predicting-a-word-using-word2vec-model) This page suggests ways to fill in a missing word based on context words including using the Word2Vec function predict_output_word().

4) Implementing the Model:

Recurrent Neural Networks (An Introduction to Implementation) (https://www.tensorflow.org/tutorials/sequences/recurrent) This tutorial illustrates how to implement a recurrent neural network using tensorflow.

Making a Predictive Keyboard using Recurrent Neural Networks (with LSTMs in tensorflow) (https://medium.com/@curiousily/making-a-predictive-keyboard-using-recurrent-neural-networks-tensorflow-for-hackers-part-v-3f238d824218) This blog post gives a general introduction to the advantages of using a recurrent neural network and demonstrates how to make one using LSTM units (Long short-term memory) to avoid the vanishing gradient problem. Given input data, the completed program continues to predict characters until it predicts a space.

Text generation using a RNN with eager execution (https://www.tensorflow.org/tutorials/sequences/text_generation) This guide illustrates how to implement and train a text prediction model using Shakespeare text. The final model predicts text character by character.

Understanding LSTM and its Quick Implementation in Keras for Sentiment Analysis (https://towardsdatascience.com/understanding-lstm-and-its-quick-implementation-in-keras-for-sentiment-analysis-af410fd85b47) This is another explanation of RNNs and LSTMs which includes an example of how to use them for Sentiment Analysis on Yelp reviews.

How to Develop a Character-Based Neural Language Model in Keras (https://machinelearningmastery.com/develop-character-based-neural-language-model-keras/) This is a general tutorial to create a model that predicts a user-defined number of characters given an input of ten characters.

Implementing a Multi-Layer RNN for a Character-Level Language Model in Torch (https://github.com/karpathy/char-rnn) This guide demonstrates how to use Torch to create and train a RNN that outputs text character by character.

5) Formatting Model for Input Data of Variable Length:

Training an RNN with examples of different lengths in Keras (https://datascience.stackexchange.com/questions/26366/training-an-rnn-with-examples-of-different-lengths-in-keras) This page suggests ways to adjust your model to make predictions on inputs of variable character and word length.

Variable Sequence Lengths in TensorFlow (https://danijar.com/variable-sequence-lengths-in-tensorflow/) This blog explains how to implement RNNs for variable-length inputs padded with 0s

6) Training and Testing:

How to Diagnose Overfitting and Underfitting of LSTM Models (https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/) This post shows you how to plot the training history of your model and determine whether it is overfit/underfit or properly fitted. It also explains why you might want to plot multiple runs.

LSTM Overfitting (https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/discussion/46494) This page offers some suggestions to deal with overfitting a model on a small dataset.

RNN Training Tips and Tricks (https://towardsdatascience.com/rnn-training-tips-and-tricks-2bf687e67527) This page suggests ways to refine your model and deal with overfitting and underfitting.

Save and restore models (https://www.tensorflow.org/tutorials/keras/save_and_restore_models) This tutorial explains different methods of saving trained Tensorflow models.