Poetry Generation

By Abraham William – 012551877
(CMPE257)

MS SE - Spring 2018

Introduction:-

From short stories to writing 50,000 word novels, machines are churning out words like never before. There are tons of examples available on the web where developers have used machine learning to write pieces of text, and the results range from the absurd to delightfully funny.

Thanks to major advancements in the field of Natural Language Processing (NLP), machines are able to understand the context and spin up tales all by themselves. Examples of text generation include machines writing entire chapters of popular novels like Game of Thrones and Harry Potter, with varying degrees of success.

In this article, we will use python and the concept of text generation to build a machine learning model that can write sonnets in the style of William Shakespeare.

Table of Contents

What are text generators?
Nowadays, there is a huge amount of data that can be categorized as sequential. It is present in the form of audio, video, text, time series, sensor data, etc. A special thing about this type of data is that if two events are occurring in a particular time frame, the occurrence of event A before event B is an entirely different scenario as compared to the occurrence of event A after event B.

However, in conventional machine learning problems, it hardly matters whether a particular data point was recorded before the other. **This consideration gives our sequence prediction problems a different solving approach.**

Text, a stream of characters lined up one after another, is a difficult thing to crack. This is because **when handling text, a model may be trained to make very accurate predictions using the sequences that have occurred previously, but one wrong prediction has the potential to make the entire sentence meaningless**. However, in case of a numerical sequence prediction problem, even if a prediction goes entirely south, it could still be considered a valid prediction (maybe with a high bias). But, it would not strike the eye.

Different Steps of Text Generation
Text generation usually involves the following steps:

4. Importing Dependencies
5. Loading of Data
6. Creating Character/Word mappings
7. Data Preprocessing
8. Modelling
9. Generating text

Let's look at each one in detail.

Importing Dependencies

import numpy as npimport pandas as pdfrom keras.models

import Sequentialfrom keras.layers

import Densefrom keras.layers

import Dropoutfrom keras.layers

import LSTMfrom keras.utils import np_utils

Loading the Data

text=(open("/Users/pranjal/Desktop/text_generator/sonnets.txt").read())

text=text.lower()

Here, we are loading a combined collection of all Shakespearean sonnets.The text file is opened and saved in *text*.

Creating character/word mappings

Mapping is a step in which we assign an arbitrary number to a character/word in the text. In this way, all unique characters/words are mapped to a number. This is important, because machines understand numbers far better than text, and this subsequently makes the training process easier.

characters = sorted(list(set(text)))n_to_char = {n:char for n, char in enumerate(characters)}char_to_n = {char:n for n, char in enumerate(characters)}

I have created a dictionary with a number assigned to each unique character present in the text. All unique characters are first stored in *characters* and are then enumerated.

It must also be noted here that I have **used character level mappings and not word mappings**. However, when compared with each other, a word-based model shows much higher accuracy as compared to a character-based model. This is because the latter model requires a much larger network to learn long-term dependencies as it not only has to remember the sequences of words, but also has to learn to predict a grammatically correct word. However, in case of a word-based model, the latter has already been taken care of.

But since this is a small dataset (with 17,670 words), and the number of unique words (4,605 in number) constitute around one-fourth of the data, it would not be a wise decision to train on such a mapping. This is because if we assume that all unique words occurred equally in number (which is not true), we would have a word occurring roughly four times in the entire training dataset, which is just not sufficient to build a text generator.

Data preprocessing:-

This is the most tricky part when it comes to building LSTM models. Transforming the data at hand into a relatable format is a difficult task.

I have braked it down the process into small parts to make it easier for you.

```
X = []
Y = []
length = len(text)
seq_length = 100
 for i in range(0, length-seq_length, 1):
    sequence = text[i:i + seq_length]
    label =text[i + seq_length]
    X.append([char_to_n[char] for char in sequence])
    Y.append(char_to_n[label])
```

Here, X is our train array, and Y is our target array.

seq_length is the length of the sequence of characters that we want to consider before predicting a particular character.

The *for loop* is used to iterate over the entire length of the text and create such sequences (stored in X) and their *true values* (stored in Y). Now, it's difficult to visualize the concept of true values here. Let's understand this with an example:

For a sequence length of 4 and the text "*hello india*", we would have our X and Y (not encoded as numbers for ease of understanding) as below:

X	**Y**
[h, e, l, l]	[o]
[e, l, l, o]	[ ]
[l, l, o,]	[i]
[l, o, , i]	[n]
….	….

Now, LSTMs accept input in the form of **(number_of_sequences, length_of_sequence, number_of_features)** which is not the current format of the arrays. Also, we need to transform the array Y into a one-hot encoded format.

X_modified = np.reshape(X, (len(X), seq_length, 1))

X_modified = X_modified / float(len(characters))

Y_modified = np_utils.to_categorical(Y)

We first reshape the array X into our required dimensions. Then, we scale the values of our *X_modified* so that our neural network can train faster and there is a lesser chance of getting stuck in a local minima. Also, our *Y_modified* is one-hot encoded to remove any ordinal relationship that may have been introduced in the process of mapping the characters. That is, *'a'* might be assigned a lower number as compared to *'z',* but that doesn't signify any relationship between the two.

Our final arrays will look like:

X_modified	Y_modified
[[0.44444444], [0.33333333], [0.66666667], [0.66666667]]	[0., 0., 0., 0., 0., 0., 0., 0., 1.]
[[0.33333333], [0.66666667], [0.66666667], [0.88888889]]	[1., 0., 0., 0., 0., 0., 0., 0., 0.]
[[0.66666667], [0.66666667], [0.88888889], [0.]]	[0., 0., 0., 0., 0., 1., 0., 0., 0.]
[[0.66666667], [0.88888889], [0.] [0.55555556]]	[0., 0., 0., 0., 0., 0., 0., 1., 0.]

Modelling

model = Sequential()

model.add(LSTM(400, input_shape=(X_modified.shape[1], X_modified.shape[2]), return_sequences=True))

model.add(Dropout(0.2))

model.add(LSTM(400))

model.add(Dropout(0.2))

model.add(Dense(Y_modified.shape[1], activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam')

We are building a sequential model with two LSTM layers having 200 units each.

The first layer needs to be fed in with the input shape. In order for the next LSTM layer to be able to process the same sequences, we enter the *return_sequences* parameter as *True.*

Also, dropout layers with a 20% dropout have been added to check for over-fitting. The last layer outputs a one hot encoded vector which gives the character output.

Generating Text

```
string_mapped = X[99]# generating charactersfor i in range(seq_length):
 x = np.reshape(string_mapped,(1,len(string_mapped), 1))
 x = x / float(len(characters))
   pred_index = np.argmax(model.predict(x, verbose=0))
   seq = [n_to_char[value] for value in string_mapped]
    string_mapped.append(pred_index)
    string_mapped = string_mapped[1:len(string_mapped)]
```

We start off with a random row from the X array, that is an array of 100 characters. After this, we target predicting another 100 characters following X. The input is reshaped and scaled as previously and the next character with maximum probability is predicted.

seq is used to store the decoded format of the string that has been predicted till now. Next, the new string is updated, such that the first character is removed and the new predicted character is included.

You can find the entire code on my git repo here. I've provided the training file, notebooks and trained model weights for your reference.

Experimenting with different models

The baseline model, when trained for 1 epoch with a batch size of 100, gave the following output:

's the riper should by time decease,his tender heir might bear his memory:but thou, contracted toet she the the the the the the the thethi the the the the the the the the the the

the the the the the the thethi the the the the the the the the the the the the the the the thethi the the the the the the the the the the the the the the the thethi the the the the the the the the the the the the the the the thethi the the the the the the the the th'

This output doesn't make much sense. It is nothing but a repetition of the same prediction, as if it's stuck in a loop. This is because language prediction models are way too complex when compared to the miniature model that we have trained.

Let's try to train the very same model, but for a longer period of time.

A more trained model

This time we trained our model for 100 epochs and a batch size of 50. We at least obtained a non-repetitive sequence of characters, which contains a decent number of legitimate words. Also, the model learnt to produce a sonnet-like word structure.

*'T**he riper should by time decease,his tender heir might bear his memory:but thou, contracted to thine own besire,that in the breath ther doomownd wron to ray,dorh part nit backn oy steresc douh dxcel;for that i have beauty lekeng norirness,for all the foowing of a former sight,which in the remame douh a foure to his,that in the very bumees of toue mart detenese;how ap i am nnw love, he past doth fiamee.to diserace but in the orsths of are orider,waie agliemt would have me** '*

However, this model is still not good enough to produce quality content. So now we'll do what everyone does when a deep learning model is not producing decent results. Build a deeper architecture!

A deeper model

A wise man once said: if the model is not doing good, increase the number of layers! I'm going to do the same with my model. Let's add another LSTM layer with 400 units followed by a dropout layer of 0.2 fraction and see what we get.

"The riper should by time decease,his tender heir might bear his memory:but thou, contracted to the world's false sporoe,with eyes so dond touls be thy domfornds,which for memorion of the seasons new;mike own self-love to shou art constanthow can i then be oy love doth give,the rose looks fair, but fairer bomments age.now with the seas that i have seen dychlthat mot to teed of you, if h ho bontent,and all the ress of the heartai thee;the world will was is bound and farty "

The result is interesting. The grammar has enhanced itself, keeping the sonnet structure and punctuation intact. However, this still requires a lot of improvement. Let's try and explore a wider network, one with more number of units.

A wider model

I increased the number of units to 700 on each of the two LSTM layers. This tweak produced the following poetry:

"**The riper should by time decease,his tender heir might bear his memory:but thou, contracted to thine own bright eyes,feed'st thy sigh aare so beuider poise,oor maty dis surre that gairs mot me love thee;when i braye the would and lays in the eesire.than thmu disgrmed stand with my bootr still nsentente;tell ia to thou art thou wilt woon'thy sook,and touain then tor, give thy soue semping.whose whod the better of your befaiss nu sante,though i, suoll oas, and i lose bouh twa**"

This is a little disappointing at first, because the words have lost their meaning. But, what's interesting to note here is that there is some rhyme that is building up. The model is trying to understand poetry after all! But, we cannot compromise with meaningful words, right?

Let's put it all together in a one gigantic model.

A gigantic model

I increased the number of layers to three, each having 700 units and trained it for 100 epochs. The result produced is a magnificent piece of poetry. Take a look:

"**The riper should by time decease,his tender heir might bear his memory:but thou, contracted to thine own bright eyes,feed'st thy light's flame with self-substantial fuel,my beept is she breat oe bath dasehr ill:tirse do i pine and turfeit day by day,or gluttoning on all, or all away.Lxxviwhy is my verse so barren of new pride,so far from variation or quick change?why with the time do i not glance asideto new-found methods, and to compounds strange?why write i sti**l"

This not only has sensible words, but has also learnt to rhyme. We could have had a more sensible piece of art had the data that was fed into the network been cleaned properly! But as a starting piece, this model has more than done what it was asked. It is way more poetic than most humans could ever get!

Conclusion

What makes a text generator more efficient is its capability to generate relevant stories. This is being implemented by many models at the output level, to generate actual language-like text, which can be difficult to differentiate from one written by humans.

A sufficiently trained model on this framework gives some eye-popping results. Also, there are models which can generate clickbaits via an automated process and grab people's attention!

In all, text generators can find great applications, right from creating original art, to regenerating content that has been lost. One revolutionary application of such text generators could be the point where we could train them to write and manipulate code. Imagine a world where computer programs and algorithms can modify themselves, as and when required.