Character-Level Language Modeling

Priyanka Gopi, Dept. of Computer Science, UCF

Aim: To use LSTM for implementing language modeling that can generate next text character by character.

Methodology:

Beginning with obtaining all the necessary libraries, like NumPy, TensorFlow models and layers like the Dense, LSTM, Dropout, Activation, etc. as below:

```
import sys
import numpy as np

# TensorFlow and tf.keras
import tensorflow.compat.v1 as tf
from keras.utils import to_categorical
from keras.callbacks import ModelCheckpoint
from sklearn.model_selection import train_test_split
from keras.models import Model, Sequential
from keras.layers import Dense, Input, LSTM, Embedding, Dropout, Activation
from keras import utils as np_utils
from keras.preprocessing.text import Tokenizer
```

Fig 1 – Importing Libraries

A character-level language model utilizing LSTM in TensorFlow is implemented in the code below. Shakespeare's texts, which are loaded from a text file and changed to lowercase, are used to train the language model.

A dictionary is first established to map each character to an integer index and vice versa. This is done by first creating a set of all the text's unique characters. Iterating through the text with the specified stride and window size results in the creation of input/target sequences. The window length and stride are defined. The input sequence for each window is the first Tx character, and the target sequence is the following Tx character. Using the dictionary of characters to indices, these sequences are represented as lists of integers.

```
# loading the text file and convert texts to lowercase
with open('/content/drive/MyDrive/shakespeare.txt', 'r') as f:
    text = f.read().lower()

# create a set of all unique characters in the text
chars = sorted(list(set(text)))
```

```
[4] # create a dictionary to map characters to integers and vice versa
char_to_int = dict((c, i) for i, c in enumerate(chars))
idx_to_char = {i: c for i, c in enumerate(chars)}
```

Fig 2 – File opening and dictionary creation

Below images show the count of vocabulary and the text entirely, whose values are 38 and 93677, respectively.

```
[5] # size of the vocabulary (number of unique characters)
vocab_size = len(chars)
vocab_size
```

```
38
```

Fig 3(a) – Vocabulary size

```
[6] #Number of characters
n_chars = len(text)
n_chars
```

```
93677
```

Fig 3(b) – Total characters

We need to perform a typical pre-processing step for any text generation – which is the reshaping, normalization and one-hot encoding. From the below image, We now need to reshape the X data in order to use LSTM on it. Through the first line, the input data X is transformed from a 2D array of shape (n patterns * seq length, 1) to a 3D array of shape (n patterns, seq length, 1). Next, by dividing the input data X by the vocabulary size (vocab size), this line normalizes the data. This adjusts the values in X to the [0, 1] range, which can enhance the neural network's performance while it is being trained. Following this, the output variable y is encoded (One-hot) in this line. The output variable for text generation is often a sequence of integers that represents the character that comes after it. This sequence of numbers is converted using one-hot encoding into a sequence of binary vectors, each of which has a length equal to the vocabulary size (vocab size). All other values in the vector are 0 except for the index corresponding to the next character's integer value, which has a value of 1. All things considered, these preprocessing methods aid in getting the input and output data ready for using an RNN model to generate text.

```
# prepare the dataset of input to output encoded as integers

seq_length = 40
X = []
Y = []
for i in range(0, n_chars-seq_length,3):
 seq_in = text[i:i + seq_length]
 seq_out = text[i + seq_length]
 X.append([char_to_int[char] for char in seq_in])
 Y.append(char_to_int[seq_out])
```

Fig 4 – Generate X and y data for training

```
[9]  X = np.reshape(X, (n_patterns, seq_length, 1))

     X = X / float(vocab_size)   # normalize
     y = to_categorical(Y)   # one hot encode the output variable
```

Fig 5 – Preprocessing data

We then proceed to data modeling where we use Tensorflow framework to add layers and functionalities to each, thereby, building a Neural network with hidden layers.

For sequence modeling tasks like text generation, recurrent neural networks (RNNs) of the Long Short-Term Memory (LSTM) type are frequently employed. Using 512 units, the LSTM layer in this model can learn to model intricate patterns in the input sequence. The length of the input sequence is indicated by the (seq length, 1) input shape specification for the layer. The LSTM layer updates its hidden state at each timestep during training to reflect the context of the sequence up to that point. The input sequence is processed timestep by timestep by the LSTM layer. A regularization method called dropout aids in preventing overfitting in neural networks. During training, the Dropout layer in this model randomly eliminates (i.e., sets to zero) a portion of the input units. According to the dropout rate specified by the 0.2 parameter, 20% of the input units will be arbitrarily eliminated after each training period. This forces the model to acquire more reliable representations, which helps prevent the model from overfitting to the training data.

The Dense layer, a fully linked layer, applies a linear transformation to the output of the LSTM layer. The size of the vocabulary is equal to the number of units in the layer, which is specified by the y.shape[1] option (i.e., the number of unique characters in the input data). The layer's

activation function is set to softmax, which converts the layer's output to a vocabulary-specific probability distribution. In order to choose the most likely next character based on the context of the preceding characters, the text is generated using the model's ability to forecast the likelihood of each possible next character in the sequence.

```
import tensorflow.keras as keras

# define the LSTM model

model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
```

Fig 6 – Model architecture

A Keras callback called ModelCheckpoint enables you to save the model's weights during training. The value to be monitored during training is specified by the argument, 'monitor'. The weights will be saved anytime the training loss improves in this situation because the word "loss" is being utilized as the monitor.

```
[9] # define the checkpoint
filepath="/content/drive/MyDrive/Weights_new/weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True, mode='min')
callbacks_list = [checkpoint]
```

Fig 7 – Creating checkpoints by monitoring the loss metric

The checkpoints came with saving weights at epochs where the loss reduced, consequently. Below image shows a sample of saved weights files.

An optimizer is designed to reduce the loss. Here, we use the 'Adam' optimizer, which has a default learning rate of 0.001. The Keras method *model.compile()* compiles the model for training. The *loss* function is used to optimize the model during training. In this instance, the loss function is categorical crossentropy, which is frequently employed for multiclass classification issues like text generation. Adam, a well-known optimization technique that adjusts the learning rate for each parameter based on the gradient history, is employed here, as the optimizer.

The model is prepared for training after compilation using the fit() method, which accepts the training data as input along with a variety of other variables, including batch size, number of epochs, and validation data. The model is trained for 1000 epochs in this case, which is the number specified in the *epoch's* argument (i.e., iterations over the full training dataset). During training, the batch size option defines the number of samples that will be used in each batch. A list of Keras callbacks is specified in the *callbacks* argument and will be called during training.

```
[14] model.compile(loss='categorical_crossentropy', optimizer="adam")

model.fit(X, y, epochs=1000, batch_size=64, callbacks=callbacks_list)
```

Fig 8 – Optimization and data modeling on X and y train data

The *ModelCheckpoint* callback is used in this instance to save the model's weights during training. The screenshot of how the epochs were monitored and the files generated from the same, is shown below.

```
1461/1464 [============================>.] - ETA: 0s - loss: 0.0714
Epoch 355: loss did not improve from 0.05598
1464/1464 [==============================] - 11s 8ms/step - loss: 0.0715
Epoch 356/1000
1460/1464 [============================>.] - ETA: 0s - loss: 0.0742
Epoch 356: loss did not improve from 0.05598
1464/1464 [==============================] - 11s 8ms/step - loss: 0.0741
Epoch 357/1000
1461/1464 [============================>.] - ETA: 0s - loss: 0.0693
Epoch 357: loss did not improve from 0.05598
1464/1464 [==============================] - 11s 8ms/step - loss: 0.0694
Epoch 358/1000
1462/1464 [============================>.] - ETA: 0s - loss: 0.0713
Epoch 358: loss did not improve from 0.05598
1464/1464 [==============================] - 11s 8ms/step - loss: 0.0713
```

Fig 9 – Example of generated epochs

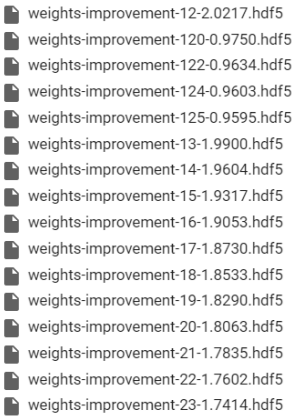

Fig 10 – Save hd5 files with improved weights

The final epoch file has the improved loss metric. Hence, we chose the weights (with a loss=0.012) to load into the model, as represented below,

```
[33] # load the network weights
     filename = "/content/drive/MyDrive/data/weights-improvement-914-0.0221.hdf5"
     model.load_weights(filename, True)
```

Fig 11 – Loading best weights to the model

The summary of the model shows the overall layers and the parameters tuned, if any.

```
[17] model.summary()

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 512)               1052672

 dropout (Dropout)           (None, 512)               0

 dense (Dense)               (None, 38)                19494

=================================================================
Total params: 1,072,166
Trainable params: 1,072,166
Non-trainable params: 0
_________________________________________________________________
```

Fig 12 – Summary of the model with loaded weights

We need to start with a seed sequence of characters and feed it to the trained model to generate fresh text character by character. A function to *generate_text* is created, which takes a seed text as input and produces new text of a specific length. Using the dictionary, we first transform the seed text into a series of integer values in the function. With the seed sequence as our input, we then use the *model.predict* method to obtain the anticipated probability distribution across the potential following characters. To append a character to the seed sequence, we can sample one from this distribution using a specific sampling technique. We keep doing this until the text is generated with the necessary length, using the updated sequence as the model's input each time. Finally, we return the created integer sequence after being converted to text using the int to char dictionary.

```
[75] # generate characters

     def generate_text(text):
         text = str (text)
         vector=[char_to_int[char] for char in text [-seq_length:]]
         output = text
         for i in range(1000):
             iseq = np.reshape(vector,(1,len(vector), 1))
             iseq = iseq / float(len(chars))

             pred = np.argmax(model.predict(iseq, verbose=0))
             seq = [idx_to_char[value] for value in vector]

             output += idx_to_char[pred]

             vector.append(pred)
             vector = vector[1:len(vector)]

         return output
```

Fig 13 – Function to predict and sample texts

An input sequence was provided to the model, as below. And the texts of 1000 characters (As specified in the above function) were generated. The results were interesting. Not much of fault was found in the generated text when compared with the actual Shakespeare's sonnet.

Input sequences:

```
[93] #Input sequences for testing

     shapespeare1 = "Look in thy glass and tell the"
     shapespeare2 = "Thy youth's proud livery so gazed on now,"
     shapespeare3 = "thee living in posterity?"
```

Fig 14 – Input sequences for text generation

Predicted text :

```
[76] output1 = generate_text(shapespeare1.lower())
     print(output1)

look in thy glass and tell the face thou viewest,
now is the time that face should form another,
whose fresh repair if now thou not renewest,
thou dost beguile the world, unbless some mother.
for where is she so fair whose uneared womb
disdains the tillage of thy husbandry?
or who is he so fond will be the tomb,
of his self-love to stop posterity?
thou art thy mother's glass and she in thee
calls back the lovely april of her prime,
so thou through windows of thine age shalt see,
despite of wrinkles this thy golden time.
but if thou live remembered not to be,
die single and thine image dies with thee.

unthrifty loveliness why dost thou spend,
upon thy self thy beauty's legacy?
nature's bequest gives nothing but doth lend,
and being frank she lends to those are free:
then beauteous niggard why dost thou abuse,
the bounteous largess given thee to give?
profitless usurer why dost thou use
so great a sum of sums yet canst not live?
for having traffic with thy self alone,
thou of thy self thy sweet self dost deceive,
```

Fig 15 (a) Predicted text of 1000 character length

Original Sonnet :

```
Look in thy glass and tell the face thou viewest,
Now is the time that face should form another,
Whose fresh repair if now thou not renewest,
Thou dost beguile the world, unbless some mother.
For where is she so fair whose uneared womb
Disdains the tillage of thy husbandry?
Or who is he so fond will be the tomb,
Of his self-love to stop posterity?
Thou art thy mother's glass and she in thee
Calls back the lovely April of her prime,
So thou through windows of thine age shalt see,
Despite of wrinkles this thy golden time.
But if thou live remembered not to be,
Die single and thine image dies with thee.

Unthrifty loveliness why dost thou spend,
Upon thy self thy beauty's legacy?
Nature's bequest gives nothing but doth lend,
And being frank she lends to those are free:
Then beauteous niggard why dost thou abuse,
The bounteous largess given thee to give?
Profitless usurer why dost thou use
So great a sum of sums yet canst not live?
For having traffic with thy self alone,
Thou of thy self thy sweet self dost deceive,
```

(b) Original text from the train set

Next, we created the function to generate text of 400 characters as requested, from the sequence input. The texts generated are pretty good. In cases (i) and (ii), we see how the texts generated are quite like the sonnet, whereas in case of (iii), there are some mis-spelled words which are meaningless. However, it is crucial to keep in mind that the generated text is not intended to be an exact replica of the original sonnet because it is based on a model's interpretation of the linguistic patterns and themes found in the training data rather than an exact replication. Below are the results.

(i)

```
[51] shakespeare_text1 = generate_text(shapespeare1.lower())
     print(shakespeare_text1)

look in thy glass and tell the face thou viewest,
now is the time that face should form another,
whose fresh repair if now thou not renewest,
thou dost beguile the world, unbless some mother.
for where is she so fair whose uneared womb
disdains the tillage of thy husbandry?
or who is he so fond will be the tomb,
of his self-love to stop posterity?
thou art thy mother's glass and she in thee
calls back the lovely april of her
```

Fig 16 (a) Predicted text of 400-character length for shapespeare1

(ii)

```
[90] #Text generation 3

     shakespeare_text2 = generate_text_400(shapespeare2.lower())
     print(shakespeare_text2)

thy youth's proud livery so gazed on now,
will be a tattered weed of small worth held:
then being asked, where all thy beauty lies,
where all the treasure of thy lusty days;
to say within thine own deep sunken eyes,
were an all-eating shame, and thriftless praise.
how much more praise deserved thy beauty's use,
if thou couldst answer 'this fair child of mine
shall sum my count, and make my old excuse'
proving his beauty by succession t
```

Fig 16 (b) Predicted text (400-character length) for shapespeare2

(iii)

```
#Text generation 3

shakespeare_text3 = generate_text_400(shapespeare3.lower())
print(shakespeare_text3)

leads summer on thee,
the parn mf toue that lo toue sote canken th thee,
whene thou mhs'st tho whrl woite no wamde tho benuty oafn the blned that benngd the clne so thee,
when yhu bet ey lans ball ant fere whel thou dest ro geed the cank shee thet goeee
th thee,
where yhu hen lo heart which tho geld she blnude to the benean whth thee wout pelf io whnter stblt,
what shou dost banu wilh then ganeen flr thy love, t
```

Fig 16 (c) Predicted text of 400-character length for shapespeare3 input

To obtain the perplexity, the data preprocessing was done. The next-line characters were removed in order to make sure that the train and test data were fairly created. The corpus data '*text'* was later split into two halves as below:

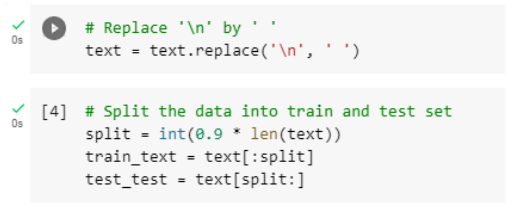

Fig 17 – Removal of '\n' and splitting of corpus

The model perplexity on the untrained model is obtained to be 28.07.

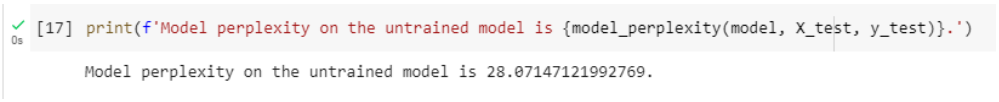

Fig 18 – Model perplexity on untrained model

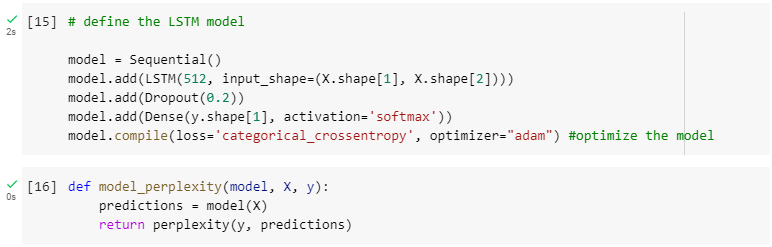

Fig 19 – Model building and perplexity calculation

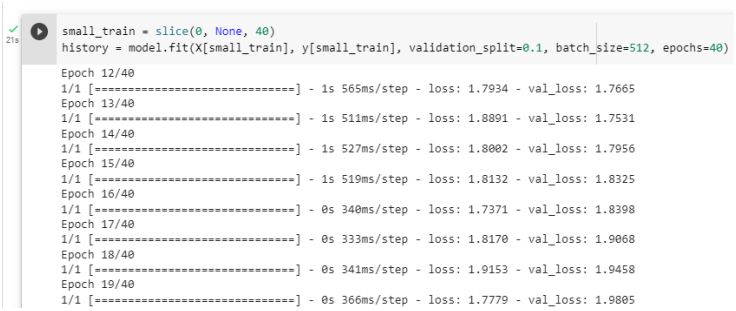

Fig 20 – Model.fit function run over 40 epochs

Perplexity is basically a metric used for performance evaluation of a language model, which tells us about the predictability of characters/words of the model. It is mathematically, the exponentiated average negative log-likelihood of the test data:

$$\text{Perplexity} = \exp(- \text{sum}(\log P(w\_i)) / N)$$

Eqn 1 - Perplexity formula

Where P(w_i) implies the P(ith word) assigned by the model from the test data, N is the total words count in the test set.

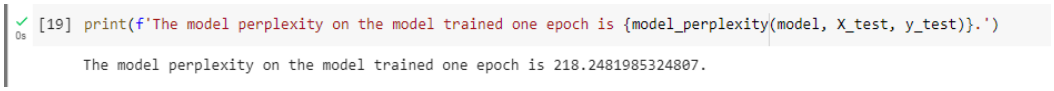

Fig 21 – Model perplexity on trained model

The model perplexity we got on training with 40 epochs is **218.24**, which could be considered a decent score to improvise further. A lower perplexity value shows that the predictability of the model is good.