Abstract

A recurrent neural network (RNN) is a class of artificial neural network where connections between nodes form a directed graph along a sequence. This allows it to exhibit temporal dynamic behavior for a time sequence. We will construct a model that predicts the following or next word to an entered word in light of a couple of the past words. We will expand it a bit by using it to predict 5 more words rather than just 1. RNN models provide us a way to backtrack to the previous word in respect to the current word that we are engaged in. Thus, a connection can be established between the previously entered word and the word that is to be followed.This project will help to ensure that when people are entering sentences that dynamic prediction of the next word to the sentence can be made so that people don't have to keep on typing the same sentence over and over again. In short, this will make typing a lot easier.

Introduction

Keyboard prediction uses text that we enter overtime to build a custom, local "dictionary" of words and phrases that we've typed repeatedly. It then "scores" those words by the probability that we'll use or need it again. For example, if we type in "I" and the keyboard will offer "am" as the following word and also other words according to use case. We can accept one of their words or we can ignore the word and leave it as is and continue typing.

RNN's are relatively old, like many other deep learning algorithms. They were initially created in the 1980's, but can only show their real potential since a few years, because of the increase in available computational power, the massive amounts of data that we have nowadays and the invention of LSTM in the 1990's. Because of their internal memory, RNN's are able to remember important things about the input they received, which enables them to be very precise in predicting what's coming next. This is the reason why they are the preferred algorithm for sequential data like time series, speech, text, financial data, audio, video, weather and much more because they can form a much deeper understanding of a sequence and its context, compared to other algorithms.

Related Work

1. SwiftKey

This Android keyboard app uses artificial intelligence that enables it to learn automatically and predict the next word the user intends to type. Swiftkey features autocorrect and gesture typing for faster input. It intelligently learns your typing patterns and adapts to it. It has been acquired by Microsoft because of the many features that it offers.

2. Fleksy

Fleksy Keyboard is known to be the fastest keyboard app for Android. It holds the world record for its typing speed twice. Fleksy uses next-generation autocorrect and gesture control so that you can type accurately within less time. Swiping gesture is used to control standard functions, such as quickly adding punctuation, space, delete, and word corrections. Fleksy is highly customizable.

3. Chrooma Keyboard

Chrooma is quite similar to the Google keyboard, except that it provides much more customizable options than the Google keyboard does. You will find all essential features such as swipe typing, gesture typing, keyboard resizing, predictive typing, and autocorrect. Chrooma has a neural action row that helps you with emojis, numbers and punctuations suggestions. It has also added a Night mode feature that can change the color tone of the keyboard when enabled. You can also set the timer and program the Night mode.This keyboard app for Android is powered by smart artificial intelligence that provides you with more accuracy and better contextual prediction while typing.

Methods

Software Specifications:

Front End Tools:

1. Programming language - Python 3.6.8

2. Operating System - Ubuntu Linux,Windows

Back End Tools :

1. Jupyter Notebook

2. Anaconda

Libraries Used

1. Numpy: It is the fundamental package for scientific computing with Python and can be used as an efficient multi-dimensional container of generic data.
2. Pandas: It takes data (like CSV or TSV file, or a SQL database) and creates a Python object with rows and columns called data frame that looks very similar to table in a statistica software.
3. Random: It generates a random float uniformly in the semi-open range [0.0, 1.0)
4. Keras: Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.
5. Tensorflow: TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. It comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.
6. Skilearn: Scikit-learn (formerly scikits.learn) is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and

DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

6. Listdir: The method listdir() returns a list containing the names of the entries in the directory given by path.

7. Namedtuples: It is factory function for creating tuple classes with named fields.

8. Heapq- This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

9. Seaborn: Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

9. Dense: Dense implements the operation: output = activation(dot(input, kernel) + bias) where kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer (only applicable if use_bias is True).

10 Pickle: The pickle model is used for serializing and deserializing of python modules.

Methodology

Dataset

We have used Emma_by_Jane_Austen as a training corpus for our model. The text is not that large and our model can be trained relatively fast using a modest GPU. We have used the lowercase version of it.

RNN and LSTM

RNNs define a recurrence relation over time steps which is given by:

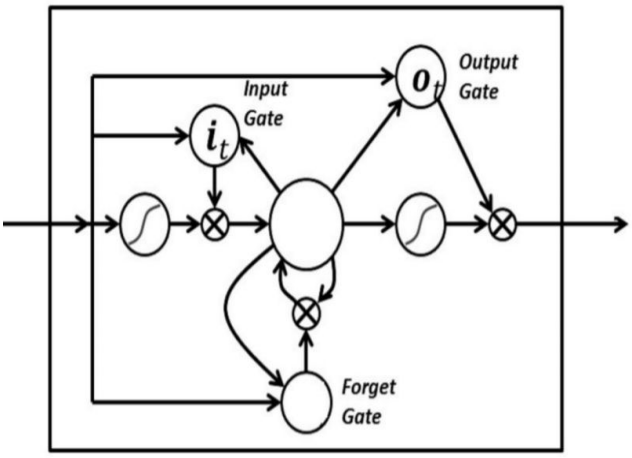
$$S_t = f(S_{t-1} * W_{rec} + X_t * W_x)$$

Where St is the state at time step t, Xt an exogenous input at time t, Wrec and Wx are weights parameters. The feedback loops gives memory to the model because it can remember information between time steps.

RNNs can compute the current state St from the current input Xt and previous state St−1 or predict the next state from St+1 from the current St and current input Xt. Concretely, we will pass a sequence of 40 characters and ask the model to predict the next one. We will append the new character and drop the first one and predict again. This will continue until we complete a whole word.

Two major problems torment the RNNs - vanishing and exploding 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. Conversely, the exploding gradient refers to the weights in this matrix being so large that it can cause learning to diverge.

LSTM model is a special kind of RNN that learns long-term dependencies. It introduces new structure — the memory cell that is composed of four elements: input, forget and output gates and a neuron that connects to itself:



LSTMs fight the gradient vanishing problem by preserving the error that can be backpropagated through time and layers. By maintaining a more constant error, they allow for learning long-term

dependencies. On another hand, exploding is controlled with gradient clipping, that is the gradient is not allowed to go above some predefined value.

Preprocessing of Dataset

To clean the data we had and remove the unwanted characters, we have used nted characters, we have use methods associated with re(regular expressions). We use re.sub to substitute unwanted characters with a blank space.

```
    Remove unwanted characters and extra spaces
text = re.sub(r'\n', ' ', text)
text = re.sub(r'[{}@_*>()\\#%+=\[\]]','', text)
text = re.sub('a0','', text)
text = re.sub('\'92t','\'t', text)
text = re.sub('\'92s','\'s', text)
text = re.sub('\'92m','\'m', text)
text = re.sub('\'92ll','\'ll', text)
text = re.sub('\'91','', text)
text = re.sub('\'92','', text)
text = re.sub('\'93','', text)
text = re.sub('\'94','', text)
text = re.sub('\.','. ', text)
text = re.sub('\!','! ', text)
text = re.sub('\?','? ', text)
text = re.sub(' +',' ', text)
return text
```

We find all unique chars in the corpus and create char to index and index to char maps:

```
unique chars: 48
```

Next, we will 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:

```
num training examples: 298423
```

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:

For single training sequence:

```
'tenberg ebook of emma, by jane austen th'
```

The character that needs to be predicted for it is:

```
'i'
```

We have 298423 training examples, each sequence has a length of 40 with 48 unique chars.

Building the model

For building our model, the initial training is pretty straight forward. A single LSTM layer with 128 neurons which accepts input of shape (40 - the length of a sequence, 48 - the number of unique characters in our dataset) is used. A fully connected layer (for our output) is added after that. It has 48 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'))
```

Softmax function, is a wonderful *activation function* that turns numbers aka logits into probabilities that sum to one. Softmax function outputs a vector that represents the probability distributions of a list of potential outcomes.

Training

We have trained our model for 20 epochs using RMSProp optimizer. We have also used 5% of the data for validation.

Saving

Once training has finished, we will have to save our model as shown below in the figure

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

And load it back, just to make sure it works:

```
model = load_model('keras_model.h5')
history = pickle.load(open("history.p", "rb"))
```

Evaluation

Let’s have a look at how our accuracy and loss change over training epochs:

Keras allows you to list the metrics to monitor during the training of your model.

You can do this by specifying the “metrics” argument and providing a list of function names (or function name aliases) to the compile() function on your model. Metric values are recorded at the end of each epoch on the training dataset. If a validation dataset is also provided, then the metric recorded is also calculated for the validation dataset.

Accuracy is special. Regardless of whether your problem is a binary or multi-class classification problem, you can specify the ‘acc‘ metric to report on accuracy.

Training

loss: 1.1990 - acc: 0.6273

Validation

val_loss: 1.7106 - val_acc: 0.5370

A metric function is similar to a loss function, except that the results from evaluating a metric are not used when training the model whereas loss function calculates how bad the prediction is of a single training instance and is used for training.

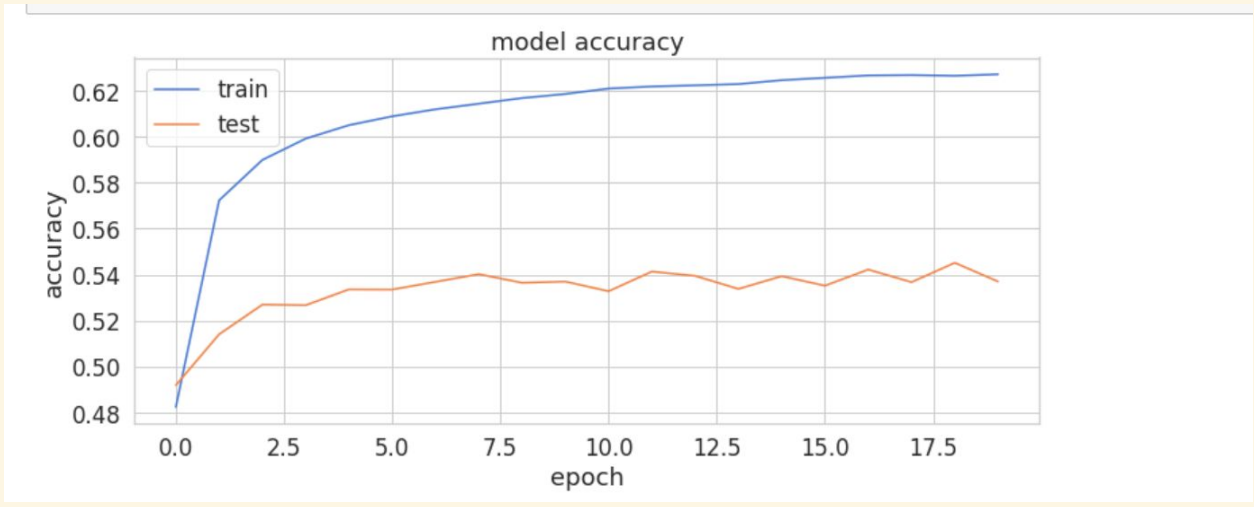


Fig. Model Accuracy

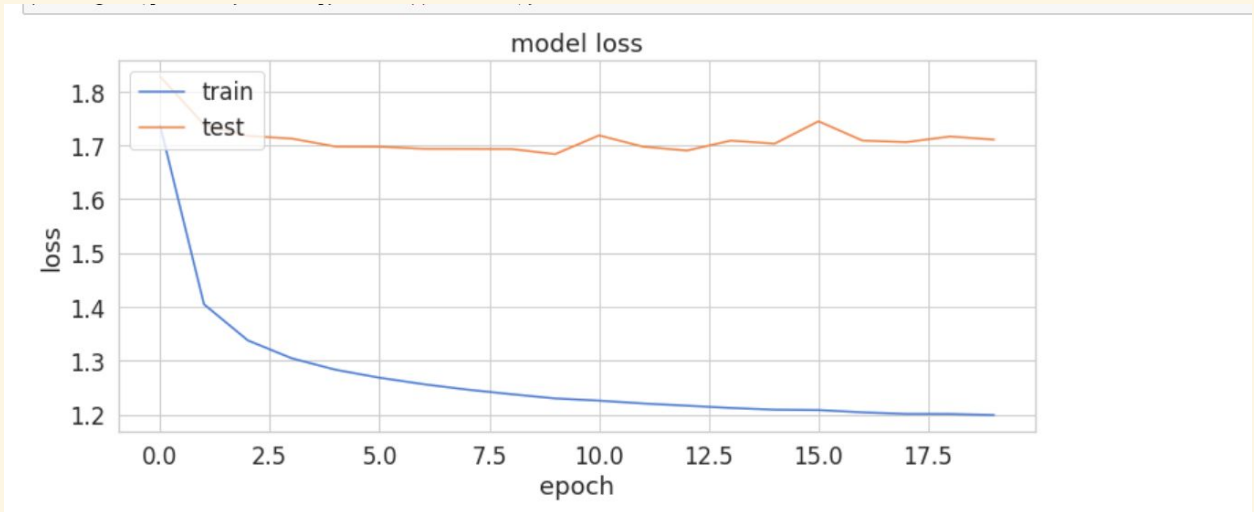


Fig. Model Loss

Testing

We use some helper functions for this purpose. We start by preparing our input text. Our sequences must be 40 characters long. So we make a tensor with shape (1, 40, 48), initialized with zeros. Then, a value of 1 is placed for each character in the passed text. We must not forget to use the lowercase version of the text Heap function allows us to ask our model what are the next n most probable characters.

Another function predicts next character until space is predicted (you can extend that to punctuation symbols, right?). It does so by repeatedly preparing input, asking our model for predictions and sampling from them. The final piece of the puzzle - predict_completions wraps everything and allow us to predict multiple completions:

Sample input and predicted outputs:

```
it is not a lack of love, but a lack of
['the ', 'him ', 'mr. ', 'a ', 'it. ']

had it taken place only once a year, it
['was ', 'is ', 'should ', 'had ', 'a ']

oh great i found your reservation, you a
['re ', 'nd ', 'lways ', 's ', ' seem ']

oh i am trying to figure out how to do t
['he ', 'o ', 'ime ', 'ake ', 'ender ']
```

A general figurative view of how actually the words are predicted by the word predictor:

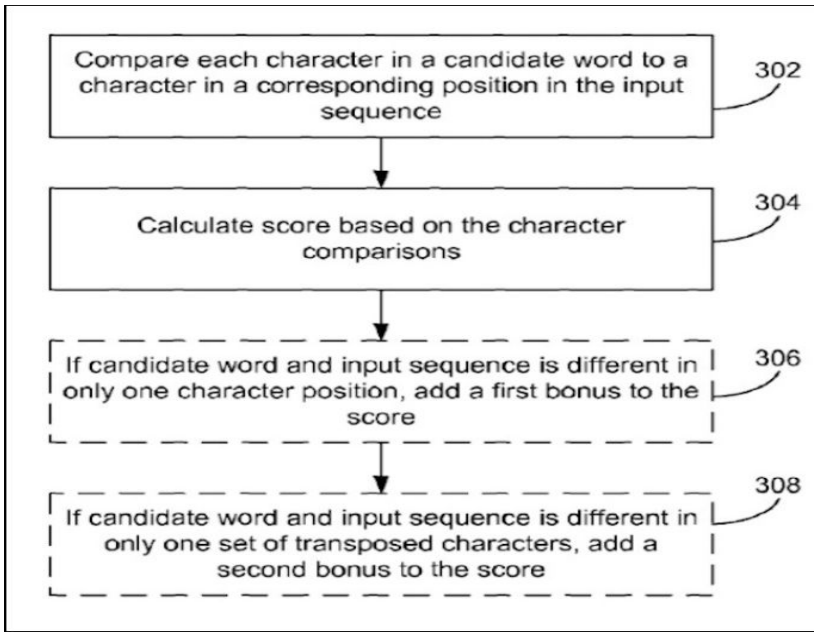


Fig: Word predictor

Shortcomings and Difficulties

We faced many difficulties while doing this project and due to such difficulties we have had to face some shortcomings in this project. Some of the difficulties we faced while doing this project are:

- A lot of time had been spent on finding the correct functions and syntax for the work as different version of tensorflow and keras use different functions and same functions have different names or changed parameters so finding the correct functions and the parameters for the functions was quite a bit of a headache.
- Lot of time had to be devoted to the training of the model because it took us a long time to train, we then opted to use google colaborator to train the model as it also provided us with a fast gpu for processing the model .
- As training the model took such a significant amount of time, so we only trained the model using Emma_by_Jane_Austen as a corpus and didn't use a large dataset for the model but the model provided us with words which actually have meaning instead of randomly combined characters and we were quite happy with that.

Due to these difficulties we had to face some shortcomings from what we had previously imagined for this project. Some of the shortcomings are:

- We had planned to make this project predict sentence of word-wise prediction, but due to time constraints we have limited it to one word prediction.
- The model only starts suggesting new words after we enter 40 characters taking the inputs as reference so it doesn't start after just one or two words as it was trained that way.
- Also, due to time constraints we were not able to create a front end for the project and results will have to be shown in colaborator.
- Also, as our model is trained in the google colaborator and we have not created a separate application for the project it will not work unless we have an internet connection.

Conclusion

In this project, we have used Recurrent Neural Networks (RNN) and Long Short Term Memory (LSTM) to create a word prediction system which can predict the next word the user will type on the basis of what the user has typed so far.

References

1. smartphones.gadgethacks.com/how-to/predictive-text-app-is-so-accurate-you-dont-even-need-look-your-phone-type-0142508/(Accessed Date: April 15 2019)

2. medium.com/@curiousNupur/how-does-swiftkey-predict-your-next-keystrokes-b048ef67267d (Accessed Date: April 15, 2019)

3. www.tensorflow.org/tutorials/sequences/recurrent(Accessed Date:May 2, 2019)

4. skymind.ai/wiki/lstm(Accessed Date: May 2, 2019)

5. towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b (Accessed Date: May 10, 2019)

6. medium.com/@curiousily/making-a-predictive-keyboard-using-recurrent-neural-networks-tensorflow-for-hackers-part-v-3f238d824218(Accessed Date: May 10, 2019)

7. https://github.com/suriyadeepan/neural-author