Building an RNN Model for Sentiment Analysis

Saikiran Komatineni

Abstract

In this paper, several strategies are explored to build the best possible RNN model that can be applied to sentiment analysis. The input text data is first tokenized and converted into matrices. The factors that were varied and tested include the layer architecture, the recurrent unit implemented, number of epochs, loss function, optimizer, and validation split size. The results present the properties of the best model with performance comparisons to all variants. This model can be expanded further for powerful applications such as checking for plagiarism.

1. Introduction

For the final project, one dataset was explored in order to solve problems related to the dataset using supervised learning algorithms. The dataset was obtained from Stanford (https://ai.stanford.edu/~amaas/data/sentiment/). For this application, the dataset used was IMDB movie Reviews.

The datasets were loaded and processed using the Python programming language (https://www.python.org/) through interactive Jupyter notebooks (https://jupyter.org/), along with the useful and necessary Python libraries: Tensorflow (https://www.tensorflow.org/), Numpy (http://www.numpy.org/), Matplotlib (https://matplotlib.org/), and Scipy (https://www.scipy.org/). The supervised learning algorithms were implemented using the Python machine learning library Sklearn (https://scikit-learn.org/). Data visualizations were made possible with the use of the Matplotlib (https://matplotlib.org/) Python library.

All of the datasets and Jupyter notebooks used for this project can be found on the Github repository:

https://github.com/skomatin/cogs_181

2. Methodology

2.1. General Approach

The objective was to evaluate the performance of each parameter and factors in data pre-processing and the network architecture. This was then used to build an optimal solution to sentiment analysis.

First, the input had to be ted into a format on which computations could be performed. For this, the text was tokenized using the Keras pre-processing package and dense vectors were formed. For training, the layers package from Keras was used to implement different types of recurrent unit layers. A dense layer was used as the last layer to receive an output value between 0 and 1.

Considering the simplicity of the expected output (positive or negative), it was hard to evaluate the improvement in the performance of each change in the parameters. Apart from the comparisions among training, testing and validation accuracies, the predicted value's magnitude was also taken into consideration. Given a range of inputs from very positive to very negative, the expected result is a linear drop in the predicted value. The linearity in the drop for each variation in parameters was compared to determine the performance for different types of input.

2.2. Tokenization

Generally, most algorithms process all data in the form of numbers as it is easier to perform computations with them. In this problem, the format of the input data is text which is written by humans. Therefore, it needs to be broken down into tokens and then words need to be converted into integers. The strategy implemented here is to represent each word (or token) as a number, which is analogous to one-hot encoding, and track the number of times each word appears.

First, a value (say x) is chosen for the maximum number of distinguishable words. It is passed into the tokenizer class from the preprocessing package in Keras. The input text data is then fit to the instantiated tokenizer object. The tokenizer converts all text into a uniform format by making everything lower case, removing punctuation, and builds a vocabulary of words. The final result is a matrix of numbers.

The next step is to ensure that the data is even in size. This is important because it allows us to train our model in batches of data rather than updating the model size for each input size. This can be done by either padding all texts till their length matches that of the longest text in the training dataset or by setting a maximum length and padding or truncating all the data to match that size. The latter option is more optimal as it requires less memory and is, therefore, more computationally efficient. Tensorflow's preprocessing package can again be used to automatically pad or truncate) the data to resize it to the specified size. Because the package does not provide a reverse function to convert tokens back to the text, a custom function was developed for that purpose.

The final step is to represent the dense vectors as integer matrices and this was implemented in the first layer of the RNN - the Embedding Layer. The embedding size, which was arbitrarily chosen in this case, is the size of the output from this layer.

2.3. RNN Architecture

The initial architecture setup included five layers: an embedding layer, three GRU layers, and a dense layer. The GRU layers, from the Keras layers package in Tensorflow, took sixteen, eight and four input sequences respectively. The training accuracy after the third epoch for this setup was 92.6% and testing accuracy was 86.6%. Although the results were acceptable, there was a slight notion of overfitting as there was still scope for improvement on the testing accuracy.

Therefore, the next attempt was to replace all GRU units with simpleRNN units to simplify the logic at each block of the recurrent unit. At a mathematical point of view, this unit is a fully connected layer where the output is fed back to the input. This reduces the number of parameters involved at each unit and therefore the number of computations as well. The model performance was worse.

The next step was to replace all three GRU blocks with LSTM blocks. The parameters were kept the same as that for GRU blocks to maintain consistency. Another variation was implemented by adding an additional GRU layer which took an input sequence of 32. This was a deeper network and it was also interesting to see if there was any correlation between the maximum number of words and the depth of the network. Although the improvement was slight, some improvement was observed.

A fourth change to the model was the addition of a dropout layer before the dense layer. The rate was arbitrarily set to 50% to test the network's performance when there is half as much feedback.

3. Results

One of the most obvious observations was the improvement in accuracy as the number of epochs was increased. On average the training accuracy at the end of each epoch was as follows: 74%, 89%, and 92%. However, not all variations in parameters started with the same accuracy. Each change in the model or preprocessing resulted in a significant change (up to 5%) change in accuracy at the end of first epoch. The accuracies for almost all variants converged to a similar value over more epochs. One anomoly to this pattern was observed during the addition of a dropout layer. Training accuracies for this case at the end of each epoch are as follows: 71%, 86%,, and 83%.

The addition of another GRU layer and the replacement of the three GRU layers with LSTM layers also resulted in very similar training and testing accuracies. The training accuracy at the end of three epochs was approximately 92%.

Less successful outcomes were observed for the following changes: using simpleRNN as the recurrent unit, using 'post' padding/truncating instead of 'pre'. The training accuracy fell to 85% and 75% for the respective cases. Other changes that resulted in insifnificant improvement (training accurcy at 92%) include increasing embedded size to 200 and using different types of optimizers..

4. Discussion

For the majority of the results obtained, they were within expectations and some new and interesting conclusions could be made. The most basic conclusion from this is that results at the end of one or two epochs are not representative of the model.

When the dropout layer was added, the speculation was that there would be a significant change (positive and negative) in the performance because only a few keywords are sufficient to determine the positivity of a text. The dropout layer may suppress or highlight these keywords. The results are as expected because training accuracy was lower after the second epoch when compared to the end of the third epoch.

LSTM blocks have more parameters and therefore a higher complexity than GRU blocks. This, however, had no correlation with training accuracy. From this, it can be concluded that for a simple (binary) outputs, it is sufficient to simpler recurrent units such as GRU. However, simpleRNN is not sufficient. The computation was also much quicker for GRU blocks and that is intuitive as GRU requires fewer mathematical computations than LSTM.

5. Bonus Points

This project implementes new packages, that were not tught in class such as the tokenizer. Being able to convert human readable text into numerical matrices is a complex task that was achieved. In addition, numerous ways of improving the model were tested in this project which goes way beyond the usual number of parameters that we tune for homework. The math behind recurrent blocks is also explained in the analysis.

Another minor factor is that training was done on a new platform called Sage Maker by AWS because of the lack of access to UCSD Protected wifi. A lot of research into AWS was done in order to setup the project on Jupyter Lab. Click on this link for proof: https://drive.google.com/drive/folders/1PJUr-lBtcFtcuaWhT0torvzot_6YaBsw?usp=sharing

6. Conclusion

Unexpected results were obtained from the implementation of LSTM blocks. Additional complexity did not improve the training accuracy. The more the number of epochs, the better the model's performance, generally speaking. Additionally, for text type inputs the tokenization process also involves hyperparameters which must also be fine tuned during training. The best RNN model for sentiment analysis involves one embedding layer, four GRU layers, and a dense layer.

References

Maas, Andrew L., et al, Learning Word Vectors for Sentiment Analysis, Association for Computational Linguistics retrieved June, 2011 from http://www.aclweb.org/anthology/P11-1015

Magnus Erik Hvass Pedersen, Tensorflow tutorial 2018 from Magnus Erik Hvass Pedersen / GitHub

Hollet, Francois, et al, Keras, 2015 from https://keras.io

A. Appendix

A.1. Table Results

Table 1: Parameters varying factors and model architecture

Note: The embedding layer is not included for simplicity

Varying Factors	Implementation 1	Implementation 2	Implementation 3	Implementation 4	Implementation 5	Implementation 6
Layer Structure	3 GRU and 1 Dense	4 GRU and 1 Dense	3 LSTM and 1 Dense	3 simpleRNN and 1 Dense	3 GRU, 1 dropout and 1 Dense	3 GRU and 1 Dense
Recurring Unit	GRU	GRU	LSTM	simpleRNN	GRU	GRU
Optimizer	Adam	Adam	Adam	Adam	Adam	Adam
Loss Function	Binary Cross Entropy	Binary Cross Entropy	Binary Cross Entropy	Binary Cross Entropy	Binary Cross Entropy	Binary Cross Entropy
Validation Split	0.05	0.05	0.05	0.05	0.2	0.05
Number of epochs	3	5	3	3	3	3
Embedded size	8	8	8	8	8	200

Table 2: Training, Testing and Validation accuracies for 3 or more epochs

Result Factors	Implementation 1	Implementation 2	Implementation 3	Implementation 4	Implementation 5	Implementation 6
Training Accuracy (%)	Epoch 1: **74.31** Epoch 2: **89.31** Epoch 3: **92.61**	Epoch 1: **75.13** Epoch 2: **89.56** Epoch 3: **92.14**	Epoch 1: **74.61** Epoch 2: **89.56** Epoch 3: **92.14**	Epoch 1: **62.77** Epoch 2: **89.56** Epoch 3: **92.14**	Epoch 1: **71.17** Epoch 2: **86.26** Epoch 3: **83.39**	Epoch 1: **75.55** Epoch 2: **86.26** Epoch 3: **92.64**

Validation Accuracy (%)	Epoch 1: **85.68** Epoch 2: **88.88** Epoch 3: **89.76**	Epoch 1: **78.48** Epoch 2: **84.64** Epoch 3: **86.68**	Epoch 1: **82.24** Epoch 2: **89.32** Epoch 3: **87.28**	Epoch 1: **83.04** Epoch 2: **89.32** Epoch 3: **87.28**	Epoch 1: **78.72** Epoch 2: **89.52** Epoch 3: **80.02**	Epoch 1: **78.64** Epoch 2: **87.26** Epoch 3: **90.10**
Test Accuracy (%)	**86.62**	**86.72**	**85.25**	**80.44**	**85.07**	**86.34**

Epoch 4: **92.40%**
Epoch 5: **92.89%**
Epoch 6: **93.56%**
Epoch 7: **94.50%**

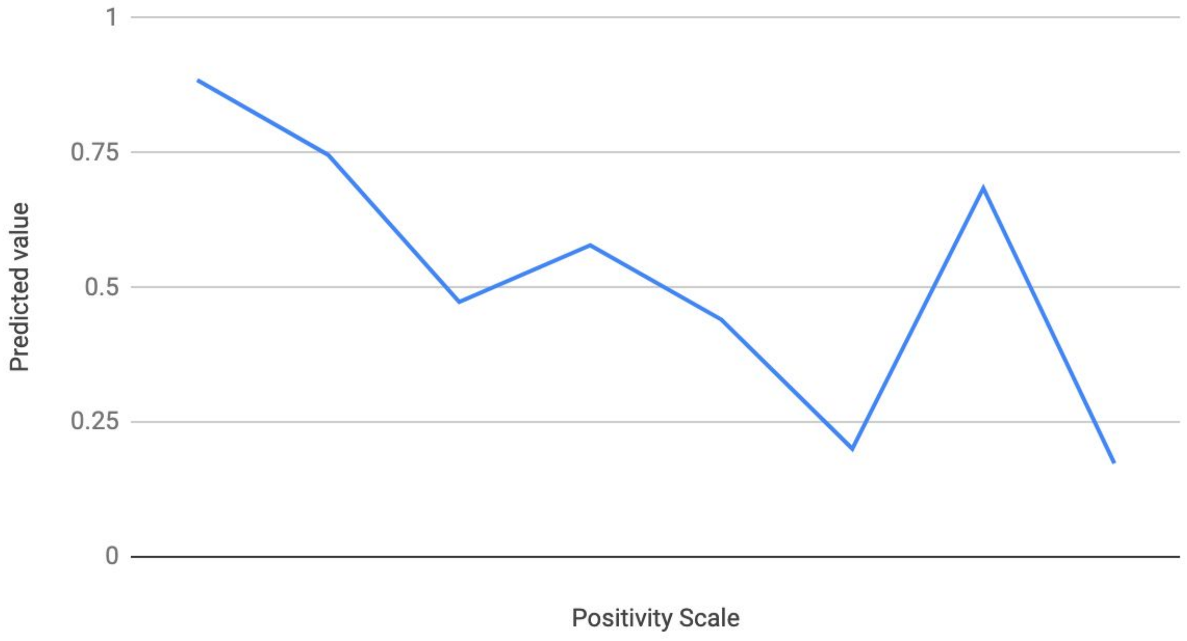
A.2. Graph Results

Input:
text1 = **"This movie is fantastic! I really like it because it is so good!"**
text2 = **"Good movie!"**
text3 = **"Maybe I like this movie."**
text4 = **"Meh ..."**
text5 = **"If I were a drunk teenager then this movie might be good."**
text6 = **"Bad movie!"**
text7 = **"Not a good movie!"**
text8 = **"This movie really sucks! Can I get my money back please?"**

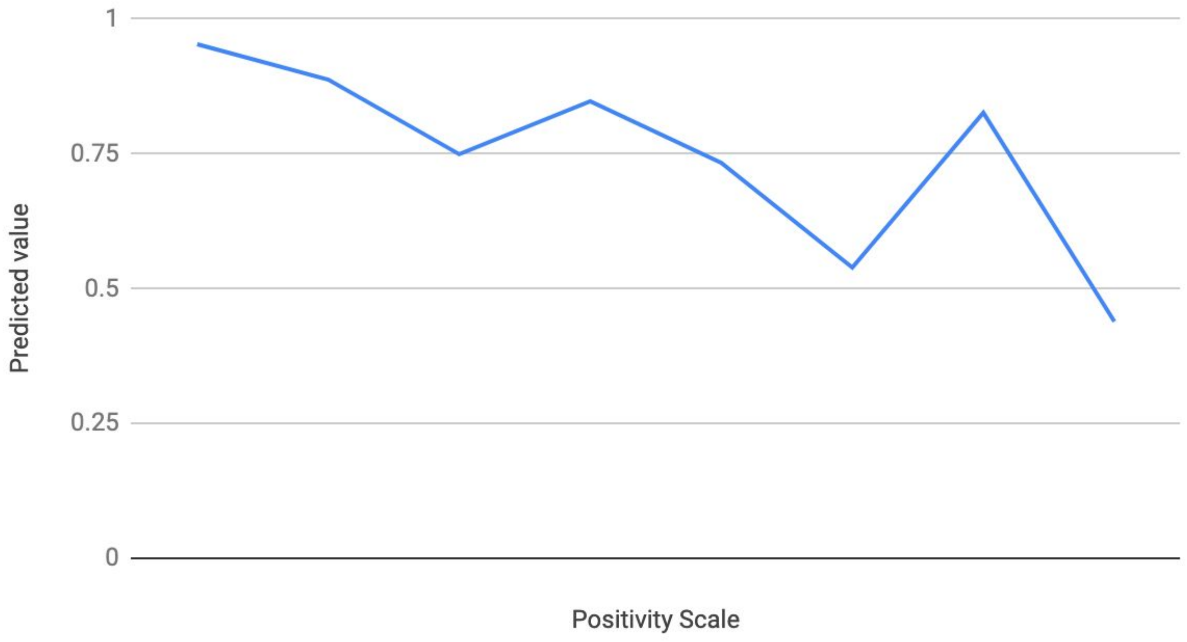
Note: The data can be generally observed to be linearly decreasing on a positivity scale

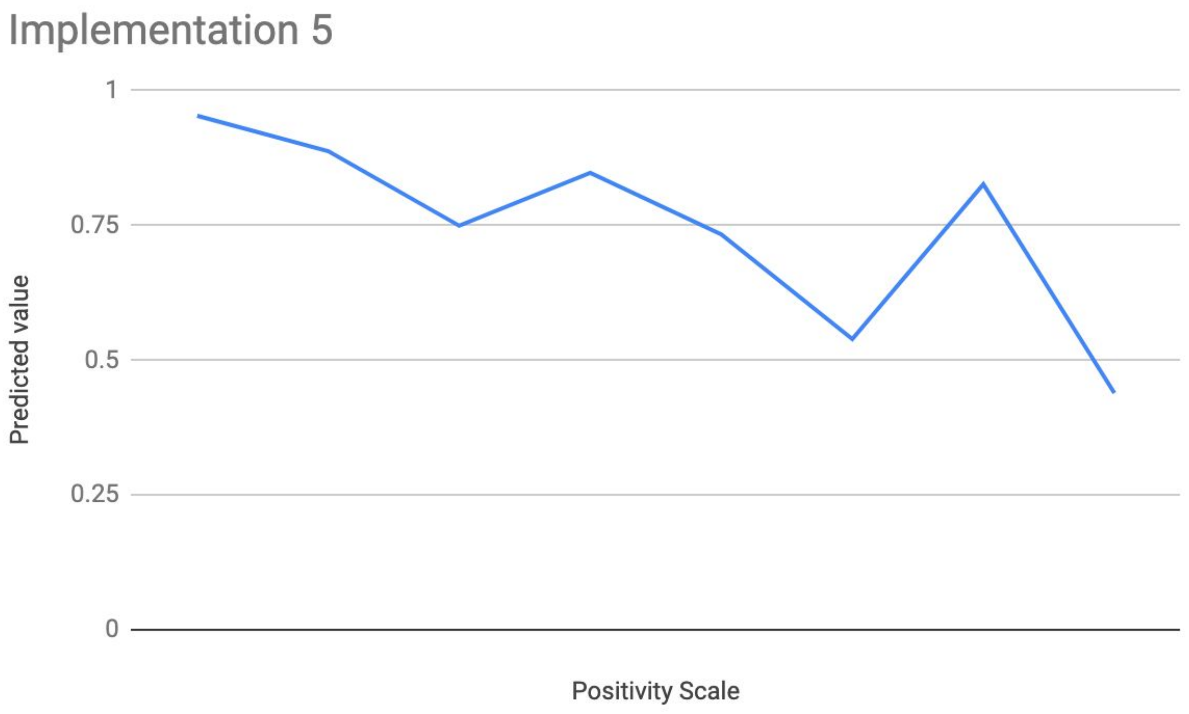
Graph 1-3: Predicted value for each text

Implementation 1



Implementation 2




Implementation 5
1
0.75
0.5
0.25
0
Predicted value
Positivity Scale