ANN-based Sentiment Analysis Approaches
By Ryan Westfall

Abstract

Sentiment Analysis is the subject matter of analyzing a sequence of text as an input and predicting if the input's sentiment is of some sort of classification. Tackling the problem of sentiment analysis, there are many ANN-based approaches to it. With so many competing approaches a question must be asked, how do these ANN-based approaches compare among each other on accuracy when tasked with sentiment analysis? The motivation to ask such a question would be for us to obtain more of a baseline understanding of these approaches and then be able to make some decision on what type of approach to choose in a practical implementation of them. So in answering this question, a dataset of over 20,000 tweets already labeled positive, negative, or neutral was employed. And after a preprocessing cleaning step on the tweets, different ANN models employing different underlying methods were built up and trained. There were three approaches tested: a Simple RNN approach, a Single Layer LSTM approach, and a CNN approach. After building up the experiment and collecting the results, the LSTM approach showed the most promise with the highest score of accuracy correctly classifying text 0.8326% of the time. It is reasonable to conclude that because of the unique feature of the LSTM to remember long term dependencies that it would perform better to the other two naive ANN approaches in sentiment analysis.

Introduction

Over the past many years, we have seen a huge increase of interest in social media sites like Facebook, Twitter, and LinkedIn. People have realized that these social media sites have the opportunity to be an invaluable source of information that allows them to obtain quick public opinion of their product and services. However, because of the very large number of posts and users, accurately tracking an individuals' sentiment of a product/service can be terribly complex. Still though, machine learning methods are capable of giving automated ways to obtain useful information from all these various sources.

The objective of my research is to use ANN approaches to accurately classify any online comment into either a "positive", "negative", or "neutral" sentiment. This idea is known as sentiment analysis and could be very useful for people wanting to gauge the general sentiment of practically anything. In my research I look to compare the accuracies of a simple RNN model, a single layer LSTM model, and a CNN model.

There have been other papers individually covering the RNN[1], LSTM[2], and CNN[3] that specifically look at their applicability for language modeling. In my research however, I wish to cover more of an overview of the three and specifically compare their results on accuracy

between each other. On another significant note, as compared to [4] that also performs a comparison of different ANN approaches in sentiment analysis, I am using the Keras library as opposed to developing the networks from scratch. I believe such a difference will significantly increase the accuracy of my results as compared to theirs because of the optimization efforts spent on such a large library like Keras.

For conducting this research, I used a dataset of 20,000 tweets labeled already as positive, negative, or neutral. By having such a labeled dataset, it will provide the supervised learning medium in which the three models to be tested will learn from. However before being able to learn from this dataset, some pre-processing must be done on the dataset for it to be usable by the models. This pre-processing includes a text cleaning step that gets rid of unnecessary text characters and a transformation step that turns the cleaned text into word embeddings, creating usable word vectors for the models. With the pre-processing now taken care of, we then build up three separate models of our three approaches using the Keras library.

Now having built up this experiment, the models' accuracies were tested. From the results that were collected, it was found that the LSTM model performed the best with an average accuracy of 0.8351%, followed by the CNN model with an average accuracy of 0.8326%, and lastly the RNN model with an average accuracy of 0.8255%. Another interesting statistic was that the accuracy for neutral sentiment prediction was of a significant margin higher than positive or negative sentiment prediction (around 10% higher).

From these results, it can be concluded that the difference in accuracy between these models was honestly pretty insignificant. This thus begs the question as to how overall accuracy could be affected by the dataset. The used dataset was found to have many more neutral sentiment labels as compared to the negative and positive sentiment labels. Such a difference could perhaps favor the accuracy outcomes of other approaches more than others, and perhaps this was the reason why neutral sentiment was easier to predict than positive or negative sentiment.

Approach

To build up this experiment, you absolutely need two things: input data and a model to make sentiment predictions. These models need correctly formatted input to perform the supervised learning needed to adjust its neuron's weights to improve its future ability to make sentiment predictions. In order to get this correctly formatted input though, we have to start from a source. Our source comes from a dataset from Kaggle titled " Sentiment Analysis: Emotion in Text tweets with existing sentiment labels". The dataset includes 20,000 tweets that have been labeled as either positive, negative or neutral. Upon closer inspection of the variability within these labels, it is found that 11,111 tweets are labeled as neutral, 5,537 as positive, and 5,861 as negative. Now to make these labeled tweets into correctly formatted input for our sentiment analysis models, we will first need to clean them. In this data cleaning we will be: removing URLs from the tweets, Tokenizing text, removing emails, removing new line characters,

removing distracting single quotes, removing all punctuation signs, lowercasing all text, detokenzing the text, and converting the list of texts to a Numpy array. Then after this cleaning we will transform the text into word embeddings, which consists of a label encoding step, and a data sequencing and splitting step. Now finally, we will build up three models that are built using the Simple RNN, the LSTM, and the CNN approaches by using the Keras library. We now have the correctly formatted input and three distinct models to train. After running 20 training epochs on all the models, we compare the accuracies among them by using a confusion matrix. Here a confusion matrix is just a table that is often used to describe the performance of a classification model on a set of test data for which the true values are known. With the results collected, we have now completed the experiment.

Cleaning

To get into a bit more of specifics with my experiment, it begins with the collection of a twitter dataset with over 20,000 tweets that are already labeled as either positive, negative, or neutral. With this dataset, I needed to transform the text into something a computer could understand. Computers are only able to understand floating point data, so this transformation step is required. Though before performing any transformation to our text, it must be cleaned. This cleaning process will remove nonessential text like URLs, emails, new line characters, and single quotes that get in the way of the transformation. The cleaning step will change the text like on the left to something like the text on the right.

```
['Chances are minimal =P i`m never gonna get my cake and stuff',
'like',
'DANGERously',
'test test from the LG enV2']
```

```
['chances are minimal never gonna get my cake and stuff',
'like',
'dangerously',
'test test from the lg env']
```

Implementing something like the code below will accomplish some of this basic cleaning.

```
def depure_data(data):

    #Removing URLs with a regular expression
    url_pattern = re.compile(r'https?://\S+|www\.\S+')
    data = url_pattern.sub(r'', data)

    # Remove Emails
    data = re.sub('\S*@\S*\s?', '', data)

    # Remove new line characters
    data = re.sub('\s+', ' ', data)

    # Remove distracting single quotes
    data = re.sub("\'", "", data)

    return data
```

After this step, we will have successfully cleaned our noisy textual dataset into a much simpler one.

Label Encoding and Data Sequencing/Splitting

As the dataset is categorical, we need to convert the sentiment labels from positive, negative, and neutral to a float type that our model can understand. To achieve this task, we'll implement the to_categorical method from Keras.

```
labels = np.array(train['sentiment'])
y = []
for i in range(len(labels)):
    if labels[i] == 'neutral':
        y.append(0)
    if labels[i] == 'negative':
        y.append(1)
    if labels[i] == 'positive':
        y.append(2)
y = np.array(y)
labels = tf.keras.utils.to_categorical(y, 3, dtype="float32")
del y
```

Then, we'll implement the Keras tokenizer as well as its pad_sequences method to transform our text data into 3D float data, otherwise our neural networks won't be able to be trained on it.

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop,Adam
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras import regularizers
from keras import backend as K
from keras.callbacks import ModelCheckpoint
max_words = 5000
max_len = 200

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(data)
sequences = tokenizer.texts_to_sequences(data)
tweets = pad_sequences(sequences, maxlen=max_len)
```

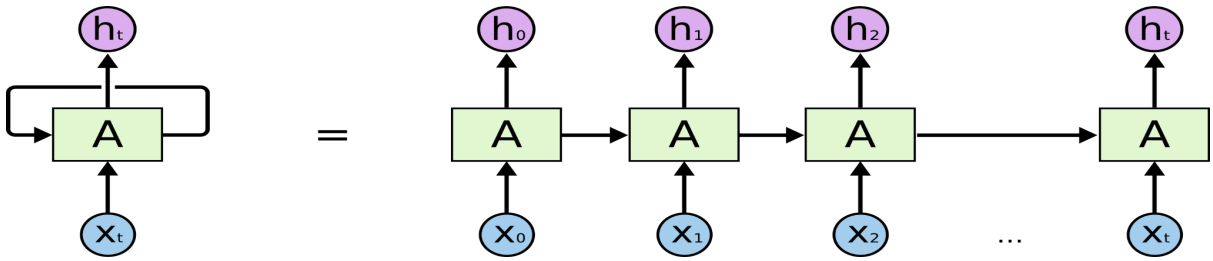
Word Embeddings Layer

We will now transform our sentences into word embeddings. These word embeddings are floating point vectors that have information of that particular word packed into a few dimensions. In our experiment we will learn these word vectors from scratch instead of using a pre-trained word embedding bank. Obtaining these word embeddings from scratch is a lot like how an ANN would learn its correct weights, we will start with random word vectors; and as it learns, it will add meaningful word vectors to its definition bank. By using the Keras library, we will be able to easily implement the word embedding by adding it on as the initial layer to our

three ANNs. With the setup for the experiment discussed, I will now introduce the three approaches experimented upon:

Recurrent Neural Network

A RNN treats words of a sentence as separate input occurring at time ‘t’and uses the activation value at ‘t-1’, as an input in addition to the input at time ‘t’. Thus, one of the notable features of a RNN is its advantage of sharing features learned across different positions of text. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. You can probably imagine how this type of architecture would be useful for NLP. It can take entire sentences as input and relationally determine word’s meanings by the context of other words.



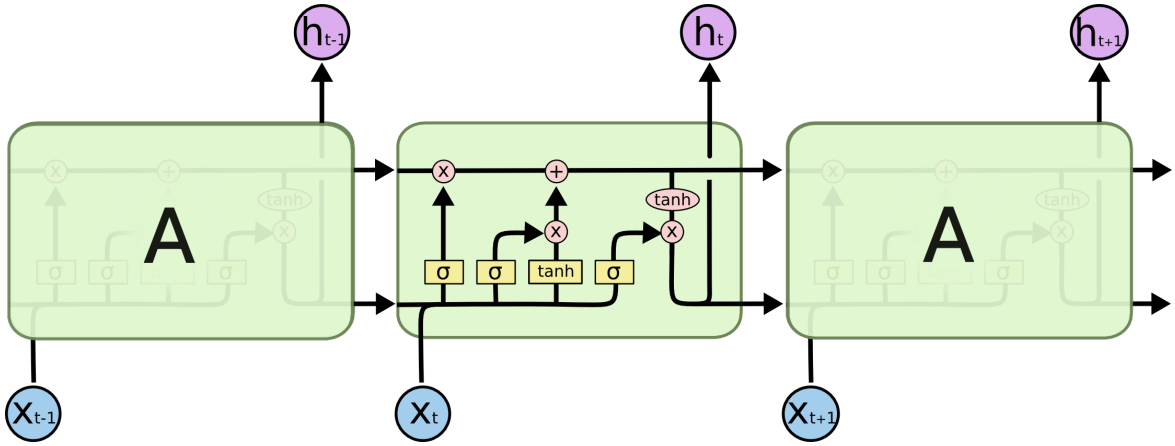
We will implement the Simple RNN with the below code and specified parameters using Keras:

```
##### Simple RNN Model #####
model0 = Sequential()
model0.add(layers.Embedding(max_words, 15))
model0.add(layers.SimpleRNN(15))
model0.add(layers.Dense(3,activation='softmax'))

model0.compile(optimizer='rmsprop',loss='categorical_crossentropy', metrics=['accuracy'])
# # Implementing model checkpoins to save the best metric and do not lose it on training.
checkpoint0 = ModelCheckpoint("best_model0.hdf5", monitor='val_accuracy', verbose=1,
save_best_only=True, mode='auto', period=1,save_weights_only=False)
history = model0.fit(X_train, y_train, epochs=20,validation_data=(X_test, y_test),callbacks=[checkpoint0])
```

Long Short Term Memory Network

The LSTM network was explicitly created to avoid the long-term dependency problem found in RNNs. The LSTM has the same control flow as RNN’s though: it processes the data sequentially passing on information as it propagates forward. The key to LSTMs is its cell state, which is like a conveyor belt that information is able to flow on unchanged. You can think of the cell state as like the memory of the network. So because the cell state can carry information throughout the sequence processing, in theory information from earlier time steps could be carried all the way to the last time step, reducing the effects of short-term memory. In charge of cell state though, are the LSTM’s gates. There are three specific types of gates within a LSTM cell: a forget gate, an input game, and an output gate. The forget gate decides what information should be thrown or kept away. The input gate decides which information should be added to the cell state. And the output gate determines what the next hidden state should be.



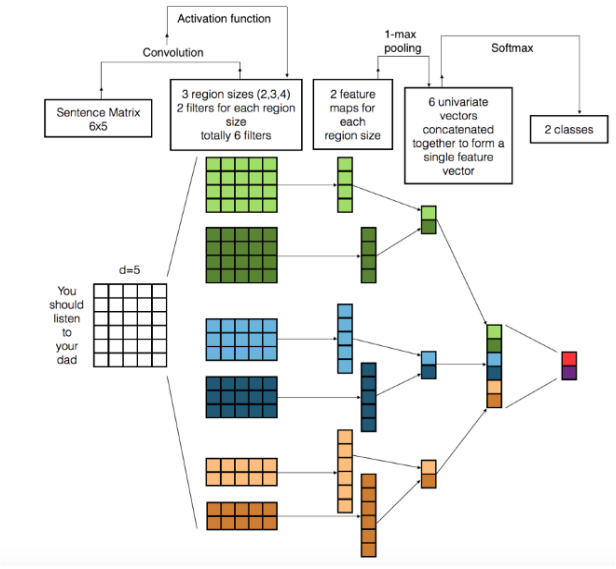
We will implement the LSTM with the below code and specified parameters using Keras:

```
##### Single LSTM Layer Model #####
model1 = Sequential()
model1.add(layers.Embedding(max_words, 20))
model1.add(layers.LSTM(15,dropout=0.5))
model1.add(layers.Dense(3,activation='softmax'))

model1.compile(optimizer='rmsprop',loss='categorical_crossentropy', metrics=['accuracy'])
#Implementing model checkpoins to save the best metric and do not lose it on training.
checkpoint1 = ModelCheckpoint("best_model1.hdf5", monitor='val_accuracy', verbose=1,
save_best_only=True, mode='auto', period=1,save_weights_only=False)
history = model1.fit(X_train, y_train, epochs=20,validation_data=(X_test, y_test),callbacks=[checkpoint1])
```

Convolutional Neural Network

CNNs treat data as spatial. Instead of neurons being connected to every neuron in the previous layer like feed-forward networks, they are instead only connected to neurons close to it and all have the same weight. This simplification in the connections means the network upholds the spatial aspect of the dataset. The word convolutional refers to the filtering process that takes place. A regular CNN is made up of multiple layers, with a couple of layers that make it unique: the convolutional and pooling layers. The convolutional layer works by placing a filter over an array, thus creating something called a convolved feature map. Then we have the pooling layer, which downsamples the sampling size of a particular feature map, outputting a pooled feature map. Now in NLP for CNNs, they can capture the spatial relationships in text by the word embeddings. By analyzing the combined word embeddings of a sentence it can start to understand the spatial relationships of the words and thus the sentiment of the sentence.



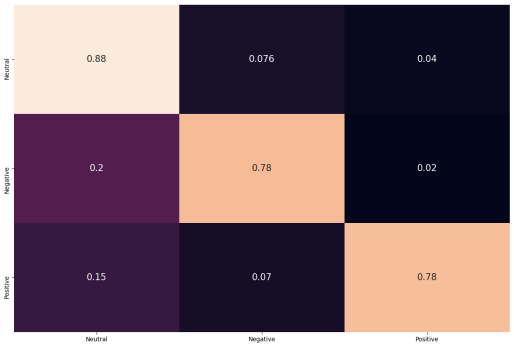
We will implement the CNN with the below code and specified parameters using Keras:

```
#####CNN Model #####
from keras import regularizers
model2 = Sequential()
model2.add(layers.Embedding(max_words, 40, input_length=max_len))
model2.add(layers.Conv1D(20, 6, activation='relu',kernel_regularizer=regularizers.l1_l2(l1=2e-3, l2=2e-3),bias_regularizer=regularizers.l2(2e-3)))
model2.add(layers.MaxPooling1D(5))
model2.add(layers.Conv1D(20, 6, activation='relu',kernel_regularizer=regularizers.l1_l2(l1=2e-3, l2=2e-3),bias_regularizer=regularizers.l2(2e-3)))
model2.add(layers.GlobalMaxPooling1D())
model2.add(layers.Dense(3,activation='softmax'))
model2.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['acc'])
# #Implementing model checkpoints to save the best metric and do not lose it on training.
checkpoint3 = ModelCheckpoint("best_model2.hdf5", monitor='val_acc', verbose=1,save_best_only=True, mode='auto', period=1,save_weights_only=False)
history = model2.fit(X_train, y_train, epochs=20,validation_data=(X_test, y_test),callbacks=[checkpoint3])
```

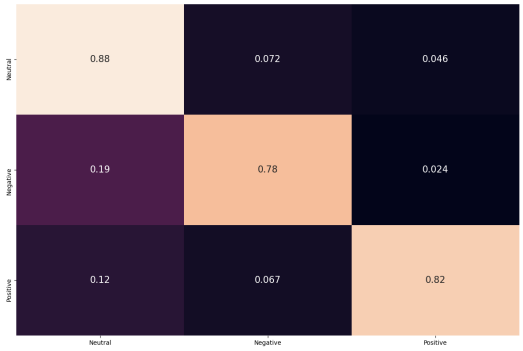
Results

Below are the results on the accuracies of different sentiment predictions within the three models. I used a confusion matrix to display these results. So for our specific use of them, the left side of the matrix are the true values and the bottom side is what our model predicted.

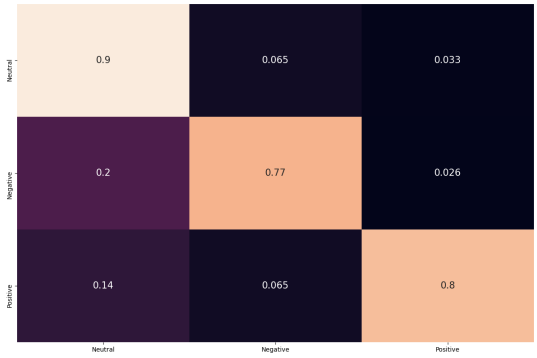
Simple RNN Model Results: Average Accuracy: 0.8255%. 78% of positive predictions were actually positive. 78% of negative predictions were actually negative. 88% of neutral predictions were actually neutral.



Single Layer LSTM Model Results: Average Accuracy: 0.8351%. 82% of positive predictions were actually positive. 78% of negative predictions were actually negative. 88% of neutral predictions were actually neutral.



1D Convolutional Model Results: Average Accuracy: 0.8326%. 80% of positive predictions were actually positive. 77% of negative predictions were actually negative. 90% of neutral predictions were actually neutral.



Discussion

Comparing these results to each other, we can see that LSTMs have the best average accuracy by but a thin margin. We can also see that although LSTMs have the best average accuracy, the CNN model has a higher score in accuracy for neutral sentiment prediction. And although the Simple RNN doesn't have a single sentiment prediction that's the best among the three, it is to mention that it was the fastest model to train.

Honestly though, these results leave much to be desired. There is a lack of difference between the results of the three approaches tested to draw any grand conclusions. I do find these very similar results quite surprising as other accuracy measurements like in [4] have their LSTM network coming in with an accuracy of 72.5% and their CNN network coming in with an accuracy of 66.7%. I suspect this wide variance in results between us comes down to our implementation of the approaches. While they implemented their approaches by scratch, I used

the Keras library that will obviously be much more optimized. I also suspect that differences in our datasets could lead to them having their average accuracy lower.

Conclusion

Based on only my results, my conclusion would be that among the three approaches tested, they will produce very similar accuracy results and thus the difference in their methodologies is kinda insignificant. However having studied the literature surrounding this field, I believe that by a sizable degree in accuracy the LSTM should perform best, the CNN performs next best, and the RNN performs worst by a significant margin. That is why I find my results so surprising. It is also reasonable to assume that the reason why neutral sentiment prediction had a significant higher accuracy than positive or negative sentiment prediction was because the dataset had a much higher number of tweets labeled as neutral. Because of this, the reason why my three models might be so close in value could be explained by the features found in the used dataset. Things like comment length, slang, and label distribution could play a really important role in which model approach would perform best.

Future Work

A good way to expand upon this research would be to add other approaches to the comparison. Particularly in mind, I would like to experiment with the Self-Attention Network. This network is very recently discovered and is making a lot of headway into natural language processing. I find these networks interesting because of their ability to be parallelized, which seems odd due to the sequential processing of text naturally abundant in this field. On top of adding other approaches to compare against, it would be interesting to expand the metrics from only accuracy to something more broad. Although accuracy is the obvious most important metric, there are other things to consider like training time, resource cost, and industry viability.

Bibliography

[1] Mikolov, Tomáš, et al. "Recurrent neural network based language model." *Eleventh annual conference of the international speech communication association*. 2010.

[2] Sundermeyer, Martin, Ralf Schlüter, and Hermann Ney. "LSTM neural networks for language modeling." *Thirteenth annual conference of the international speech communication association*. 2012.

[3] Zhang, Ye, and Byron Wallace. "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification." *arXiv preprint arXiv:1510.03820* (2015).

Sosa, Pedro M., and Shayan Sadigh. "Twitter sentiment analysis with neural networks." *Academia. edu* (2016).

[4] Sosa, Pedro M. "Twitter sentiment analysis using combined LSTM-CNN models." *Eprint Arxiv* (2017): 1-9.