Tone Analysis Using Convolutional Neural Networks

Author(s): Hamid Shah and Damien Kulla
Date: February 28, 2025
GitHub Repository URL: https://github.com/Hyperstop17/CMPSC497Midterm

Problem Definition

Traditional sentiment analysis categorizes text into broad emotional categories such as positive, negative, or neutral. However, we know that human expression is far more nuanced so our project focuses on tone analysis, where we classify sentences based on how they "sound" to a reader. Instead of binary or ternary sentiment classification, our model predicts one of 30 possible tone categories, including but not limited to "enthusiastic," "arrogant," "informative," and "cautionary."

Our approach employs a Convolutional Neural Network (CNN) with word embeddings to classify sentences into these predefined tones. CNNs, typically used in computer vision, have also shown promise in Natural Language Processing (NLP) tasks, particularly text classification.

Dataset Curation

The dataset consists of 3356 sentences, with an average of 124 sentences per category. An example of the dataset is as follows:

```
Your creativity is truly inspiring || Appreciative.
```

This is a relatively clean dataset that requires only minor cleaning. The data, however, was structured in an alphabetical order with respect to the labels. To fix this issue and make sure all labels are accounted for, we scrambled the dataset randomly and used that as the dataset to train on. Using python's built-in .split() function, we separate the data into a list of two parts with the structure [*sentence, label*]. These two parts are then added to a list one for sentences the other for labels, so when checking for accuracy, we can just compare the two indices of the separate lists. We used around an 80/20 split for our training and testing data, with 2680 sentences belonging to training and 676 belonging to testing.

Word Embeddings

We utilized a Convolutional Neural Network (CNN) for text classification. CNNs excel at extracting spatial hierarchies of features, which is beneficial for text analysis. For word embeddings we used three different embeddings. One was the keras native tokenizer. This tokenizer mainly was used for organizational purposes taking in the vocabulary from the sentences and converting them each to an integer based on when they are first encountered. They convert both the training and label data to create a target value for the embedded vectors to train towards. The two other embeddings used are common within the field of NLP, this being the Word2Vec embedding and GLoVe embedding. While different, both compare words over a large vector space and form dense vectors based on each word's similarity to other words in the corpus. These embeddings were combined in an embedding matrix of size 5000 x 100. The 5000 refers to the 5000 most common English words, while 100 is used as the max length of the sentences. These embeddings were merged by taking an average of the two embeddings and placing the value in the appropriate word index. If one of the models contains a word embedding that the other does not, then no average is taken, and the embedding is placed as normal. This is detailed in the following code:

```
#This combines the two embeddings by averaging them up

       if w2v_vector is not None and glove_vector is not None:

           embedding_matrix[i] = (w2v_vector + glove_vector) / 2

       elif w2v_vector is not None:

           embedding_matrix[i] = w2v_vector

       elif glove_vector is not None:

           embedding_matrix[i] = glove_vector
```

Glove Model

The glove model does not train the word embeddings with a function but instead utilizes an API of pre-trained word embeddings found in *glove-wiki-gigaword-100*. This model has around 27B tokens, containing a 1.2-million-word vocabulary. More details can be found in the following site: https://huggingface.co/fse/glove-wiki-gigaword-100.

Word2Vec Model

The Word2Vec model uses a list comprehension across the entire training dataset of sentences. These sentences are passed through a built-in function Word2Vec() before being embedded in a premade word to vec model shown in the following code:

```
w2v_model = Word2Vec(sentences=[sentence.split() for sentence in
training_data], vector_size=100, window=5, min_count=1, workers=4)

word_vectors = w2v_model.wv
```

Architecture:

The architecture uses a CNN as follows:

First is the embedding layer as established previously above. Then, the matrix is filtered through a convolution window of 5x5. The output matrix is set to 128. The convolution utilizes a rectified linear unit(ReLU) as its activation function. The output is then sent through a Max Pooling layer of size 2x2. This process is repeated to produce an output matrix of 64x64. This output matrix is flattened, that is, turned into a 1-dimensional vector as an output.

Training Optimization:

The model was optimized by minimizing the loss function after each epoch of training through the multilayered CNN. The loss was calculated using sparse categorical cross entropy which is necessary as it calculates the loss from a target vector which does not have to be one-hot per se, matching with our word embeddings which themselves are dense vectors. We utilized the ADAM optimizer as a way to minimize our loss function. ADAM takes an adaptive step size through a gradient calculation and utilizes stochastic gradient descent to get to the solution. A batch size of 32 examples were used per epoch. 25 total epochs were decided for minimizing the loss of our word embeddings.

Experiments and Changes Made:

Multiple Word Embeddings -

Initially, Word2Vec was the only utilized embedding within our code, however, upon adding GLoVe we saw increases in model accuracy of ~8% from an initial 53% model accuracy. Whether or not utilizing other word embeddings such as fasttext would improve model accuracy needs to be explored more closely.

Epoch Determination -

Within machine learning, determining the number of iterations you should go through your training data requires much experimentation. For our model, we found that the model peaked in training data accuracy after around 15-20 iterations. However, in deep learning it is known that further training can help increase testing accuracy even if the model has peaked in training accuracy. For this reason we found that 25 epochs ended up producing the best results without overfitting and creating diminishing returns.

Optimizer Choice -

ADAM is often considered the most efficient optimizer. It is often able to reach a local minimum within a few iterations and due to the use of batch sizes, can be computationally fast and efficient. With that said, multiple types of optimizers were tested, including variants of ADAM like Adamax and AdamW, alongside other optimizers such as RMSprop and Adagrad. With the ADAM variants, we found that they often took many more iterations to reach a minimum without any major effects to model accuracy. With other optimizers, a similar situation happened, however, these optimizers also tended to train much slower over each iteration. Given the hardware constraints of our team members (8GB RAM laptops), it was decided that ADAM would be the best optimizer.

Output:

Because of Tensorflow's clean interface, during the training phase we can observe active changes to the model during each epoch alongside other valuable information such as the time to execute the step function (in ms/step), the total training time for each epoch, and the accuracy at the end of each iteration. Furthermore, we can see validation accuracy at each training stage to make sure the model is trending towards a more accurate direction. During the building of the project, we printed some predictive outputs to make sure the model properly classified data. This output likely could be further improved by showing some wrong output predictions to detect possible issues with the neural network. Another possible data display improvement could be to graph the relative accuracies of each label and see which ones the model performs best at.

Overall Output and Improvements:

Testing Accuracy: ~61%

Training Accuracy: ~98%

The model underperforms compared to acceptable standards using word embeddings(generally above 70%). There are many ways the output could have been improved. From the data point of view, 124 sentences per label is quite small and more data would probably greatly help improve general training accuracy. It is also hard to determine how objective the dataset is as "tone" is a somewhat vague term, especially as some of the labels have overlapping qualities such as: "*direct*" and "*assertive*" or "*admiring*" and "*appreciative*". While CNNs are very effective models, they lack the context richness of more complex models such as a transformer or attention, which can form very strong relationships between words.

Key Takeaways:

One of the biggest lessons we learned was how to experiment with different parts of a model to produce greater accuracy in an output. Even though not every method worked, making small changes and rerunning the code helped achieve better outputs. We also learned how to properly process and clean data to feed as an input in an NLP task. This can often be a significant hurdle in many tasks in NLP, as there are a great many formatting issues that can arise and harm the accuracy of a model. This project was an excellent introduction to utilizing CNNs in a natural language setting and shows that this type of model does have real-world value. Reflecting on the changes made in the project, we also recognize that many improvements can be made to enhance model performance further.