587 Midterm: Using CNNs for Sentiment Classification

Ronak Singh

February 28, 2025

1 Introduction

A classical task in NLP is sentiment analysis: the classification of text as being positive, negative, or neutral. Sentiment analysis is valuable for understanding data like user feedback. In this project, I used the convolutional neural network (CNN) architecture to create a model that performs sentiment analysis on YouTube comments. In addition to training this model on a dataset of YouTube comments, I further evaluated the robustness of the model by observing how model performance changes when realistic typographical errors are introduced. We begin by specifying the problem definition and dataset curation methods. We continue by discussing details concerning word embeddings, the CNN architecture, and the training process. Next, we discuss the initial results of the trained model. We end our discussion by reviewing in-depth analysis and experiments related to the robustness of the model (when working with text with typos).

2 Problem Definition and Dataset Curation

2.1 Problem Definition

Sentiment analysis is a foundational task in NLP. The task requires a trained model to classify text as either being positive, negative, or neutral, depending on the perceived connotation of the text. One particularly challenging problem in sentiment analysis is accurately classifying noisy text, where text data contains irrelevant or ambiguous elements. This "noisy text" could be the result of spelling errors, typographical errors, or grammatical errors. One domain where such errors are present is in text that is available online. Text from online sources like tweets often contain special characters or intentionally inaccurate spelling (for example, "fyi" instead of "for your information"). In this project, I attempted to train a model that performs sentiment analysis on text that originates from another online source—YouTube comments.

2.2 Dataset Description

The dataset itself can be found on Kaggle (the link is in the `README.md` file). The dataset contains 17,872 YouTube comments, each of which have been pre-labeled with a corresponding sentiment (negative, neutral, positive). This dataset was chosen due to the quality of the data. Additionally, YouTube comments generally utilize less jargon and informal speech compared to tweets, making the baseline sentiment analysis task more straightforward (this will be beneficial when we introduce typographical errors in the data in Sec. 5).

One key limitation of the dataset is imbalance in the number of rows associated with each sentiment label. For example, 62% of YouTube comment are labeled as positive, while only 25% comments are neutral, and the remaining 13% are negative. As shown later in Sec. 4, this class imbalance in the dataset resulted in inconsistent performance metrics between classes in the trained model.

2.3 Data Preprocessing

In order to prepare the dataset to be used in training the model, a series of preprocessing and curating steps were performed. After loading the dataset, all rows with null values were dropped. Additionally, any duplicate rows were removed from the data, ensuring each YouTube comment in the dataset

corresponds to a single row. Once these preliminary preprocessing steps were performed, the comments themselves were cleaned. This cleaning process removed links (to websites), mentions (using @ to tag other users), hashtags (using # to tag a topic), and non-alphabetical characters from the comments. Links, mentions, hashtags, and non-alphabetical characters (like numbers) oftentimes aren't required to perform effective sentiment analysis, and their presence can sometimes hurt a model's performance. The last step of cleaning the comments involved converting all text to lowercase (for consistency). In addition to cleaning comments, additional columns were added to the dataset to hold the word count and character count of each respective comment.

Instead of sentiment labels being represented as strings (negative, neutral, positive), the dataset was modified to represent sentiment labels with integers (0, 1, and 2, repectively). After these preliminary data curation steps were applied, each YouTube comment was tokenized using the `nltk.tokenize` library. Finally, the dataset was divided into a training and test set using an 80/20 split.

3 Word Embeddings, Algorithm, and Training

3.1 Word Embeddings

A pre-trained Word2Vec model was utilized to generate word embeddings. Specifically, I used Google's `GoogleNews-vectors-negative300` model, a Word2Vec model that was pre-trained on the Google News corpus. The model has a vocabulary of 3 million words and generates 300-dimension embeddings vectors. The primary reason for using this pre-trained model for embeddings is because of the extremely large size of the training corpus (much larger than the YouTube comments dataset). I decided not to train my own Word2Vec model due to the presence of noisy data (containing typos and spelling mistakes) in the dataset, meaning that any embeddings model trained on the the YouTube comments dataset would also likely encode all of the spelling mistakes and writing errors present in the data.

Once the comments data was tokenized (using the `nltk.tokenize` library), embeddings were generated using the pre-trained Word2Vec model. All comments were padded to the length of the longest comment in the dataset (974).

3.2 Algorithm and Architecture

The sentiment analysis model is implemented as a CNN. The model itself processes sequences of text, and applies a series of convolutions to the sequences to learn patterns. The convolutional layer of the network has 300 input channels (corresponding to the 300-dimension embeddings vectors) and 100 output channels (one for each of the 100 convolutional filters that are applied). Multiple kernels of sizes 3, 4, and 5 are utilized at the convolutional layer. Intuitively, these varied kernel sizes should help in identifying patterns in words of different lengths.

After a convolution, a ReLU activation function and max-pooling are applied to enable the model to learn complex patterns and extract the most prominent features. The pooled outputs are then concatenated across all filters, and a dropout layer with $p = 0.5$ is applied to avoid overfitting the data. Finally the output of the dropout layer is sent to a final fully-connected layer that outputs the predicted sentiment (0, 1, or 2 for negative, neutral, or positive, respectively).

3.3 Training Process

The results presented in Sec. 4 are based on a set of 14,299 unique YouTube comments. To train the neural network, I used the Adam Optimizer (implemented as *Adam* optimizer in PyTorch). I set the learning rate to 0.001 and the mini-batch size to 32. The model was trained for 15 epochs. All training and testing of the model was performed using PyTorch, and all training was performed on a rented NVIDIA A4000 GPU.

4 Results and Analysis

In this section, I present the initial results and analysis for the model trained on the YouTube comments sentiment analysis dataset. After training for 15 epochs, the model achieved an accuracy of 79.6% and an F1-score of 0.739 on the test dataset (of size $3,575$). These results place the model well within

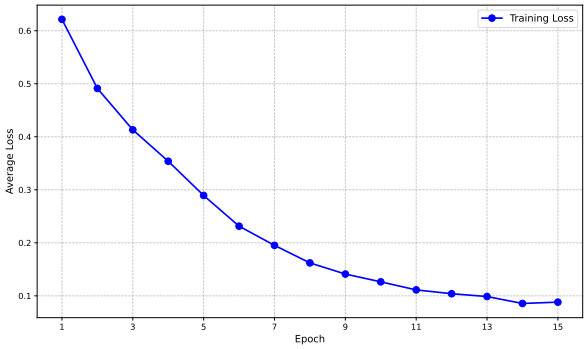

Figure 1: The changes in the model's training loss as training progressed (over 15 epochs).

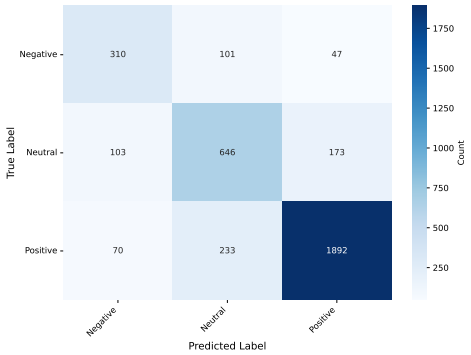

Figure 2: The final trained model's confusion matrix. Darker colors represent higher values. Note that due to the class imbalance, the *positive* label has the darkest associated colors.

the range of reasonable performance. As show in Fig. 1, the training loss changed as expected as training continued, and the curve of the average loss begins to "flatten" at around 15 epochs. The size of the class imbalance is especially prevalent in Fig. 2, as the model has clearly learned how to identify *positive* labels far better than the other labels. Additionally, it is interesting to note that the model appears to confuse the *neutral* and *positive* labels the most. If the dataset were more balanced, it is likely that the model would also improve (without making any additional changes).

5 Experiments and In-Depth Analysis

In this section, I present additional experiments I performed to assess the robustness of the trained model against typographical errors. I begin by describing my experiment design, and I continue by presenting the results and respective analysis.

5.1 Experimental Design

The objective of the first experiment is to assess how the presence of typographical errors affects the performance of the trained model. I begin by discussing how typographical errors are added to the dataset.

When considering methods to inject typographical errors into the dataset, one important consideration is to ensure that typos are realistic. This means that it is not realistic to insert random perturbations in each YouTube comment. Additionally, there exist multiple types of common typographical errors—not all typos are the same. As a result, I introduced different categories of typos into the dataset. The categories are `keyboard_proximity`, `transpose`, `delete`, and `duplicate_letter`.

`keyboard_proximity` errors are typos that result from accidentally typing a key that is nearby the letter you intend to press, rather than typing the actual letter. For example, if you intend to type *a*, you may accidentally type *q, w, s,* or *z*. This can lead to typos that cause you to type *qpple*,

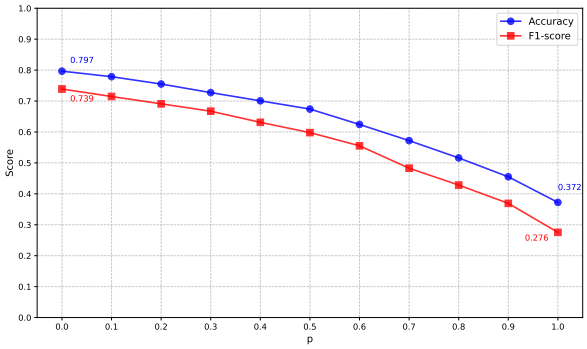

Figure 3: How the model's accuracy and F1-score change as the value of p (the proportion of words in the dataset with typos) increases. Notice that when $p = 1$ (all words have typos), the model only performs marginally better than randomized guessing.

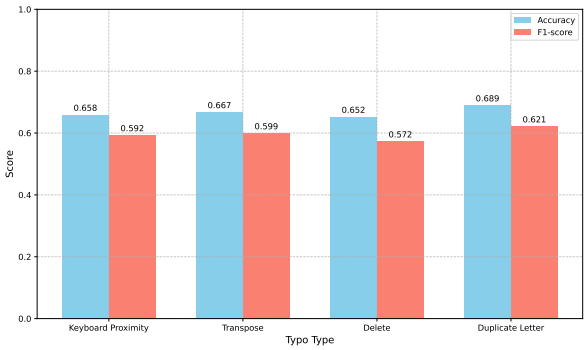

Figure 4: How different categories of typos affect the model's accuracy and F1-score when $p = 0.5$.

rather than *apple*. These errors can be especially prevalent when typing on a smaller virtual keyboard. `transpose` errors are typos that swap adjacent letters in a word. This can lead to typos that cause you to type *paple*, rather than *apple*. These errors can be prevalent when a user types quickly. `delete` errors are typos where a letter is omitted from a word. Oftentimes, these errors are the result of users unintentionally spelling words incorrectly (although they can also be a result of quick typing). This can lead to typos that cause you to type *mispell*, rather than *misspell*. Finally, `duplicate_letter` errors are typos where a letter is repeated in a word. Once again, these typos can be the result of users typing too quickly, or even misunderstanding how a word is actually spelled. This can lead to typos that cause you to type *applle*, rather than *apple*.

Each of these methods allows us to introduce artificial—but realistic—typographical errors into the dataset. Next, I will continue by discussing how these typos are used in the experiment.

When injecting typographical errors into the data, I control the percentage p of words that contain a typo. For example, if $p = 0.1$, 10% of the words in the dataset will contain one of the four typo categories previously described (the typo category of each respective typo is randomly decided). In the first experiment, I examine how the value of p (the proportion of words in the dataset with typos) affects the performance of the trained model. Due to the imbalanced classes in the dataset, we will examine *both* the accuracy and the F1-score.

5.2 Experimental Results

As shown in Fig. 3, it is evident that both the accuracy and the F1-score of the model drop as more typos are added to the dataset. Interestingly, it appears that the effects of these typos "compound". As a result, going from a dataset with 10% to 20% of words containing typos gives us a much smaller drop in performance compared to going from a dataset with 90% to 100% of words containing typos. While it is unrealistic for $p = 1$ in a real dataset, if offers interesting insight to understanding how the trained CNN model reacts to extremely noisy data.

5.3 The Effects of Different Typos

I ran one additional experiment to gain insight into which typos have the largest impact on the performance of the model. In other words, I wanted to see which category of typo the trained model struggled with the most.

To do this, I fixed the proportion of words with typos to be $p = 0.5$ (half of all words in the dataset contain typos). Then, I changed which category the typos fell into for each run of the experiment. This means that for one trial, all typos introduced in the dataset were `keyboard_proximity` errors, and for another trial all typos introduced in the dataset were `transpose` errors, and so on. As shown in Fig. 4, the model responded similarly to all of the typos, although it appears that `duplicate_letter` errors have the least impact on the model's performance. On the other hand, `delete` errors appear to have the largest impact on the model's performance. This makes sense, as deleting characters from words can change the meaning of the word altogether.

6 Lessons and Experiences Learned

All in all, I learned many lessons throughout the development of this project and its associated experiments. I gained a deeper understanding of how CNNs can be used in NLP tasks like sentiment analysis (I previously had only used CNNs with image data). Additionally, I gained new insight into how noisy data (in the form of artificially-introduced typographical errors) can impact a model's performance (and how these hits in performance aren't necessarily linear). I also gained further experience working with PyTorch and pretrained Word2Vec models.

On the more technical side, one lesson I learned was the importance of specifying the seed when working with "random" functions, such as when splitting the dataset into a training and test set. Specifying the seed ensures that data can be reproduced and is consistent. Additionally, I ran into issues with Jupyter notebooks running out of memory and automatically restarting the kernel—I was able to work through these issues with consistent troubleshooting. Additionally, I initially didn't understand the reason for adding padding when generating a dataset of embeddings vectors from a corpus of text. After a period of debugging, I finally understood the purpose of adding this padding (to have consistent dimensions).