CMPS242 Homework #5 – Neural Network Tweet Classification

Benjamin Sherman

&

Zayd Hammoudeh

Team Name: "Who Farted?"

1 Homework Objective

Develop a long short term memory-based (LSTM) neural network that classifies tweets as being sent by either Hillary Clinton (@HillaryClinton) or Donald Trump (@realDonaldTrump).

2 Dataset Overview

The dataset is a comma-separated list of tweets from either @HillaryClinton or @realDonaldTrump. No additional information is provided beyond the tweet's text and potentially the class label. The training set consists of 4,743 tweets while the test set has 1,701 unlabeled tweets.

3 Tweet Text Preprocessor

Since an LSTM is a recurrent neural network, it can rely on both vocabulary and contextual information to perform classification. Extensive text preprocessing has the potential to destroy subtle textual information. As such, we performed very minimal text manipulation beyond vectorization. For example, we left all stop words in place. We removed some of the punctuation but left hashtags (#) and exclamation points as we theorized that they may be part of a specific user's unique tweet "signature." This results in a vocabulary of approximately eleven thousand words; it should be noted that a small percentage of this vocabulary is gibberish, one-off, shortened URLs. We believe that with additional effort, we may be able to derive useful information from these URLs, but we leave that as a future exercise.

4 Classifier Structure

At a minimum, this homework's LSTM classifier must have the structure shown in Figure 1. There are four primary components, namely the embedding matrix, one-hot vector, long short-term memory network, and feed-forward network, each of which are described in the following subsections.

4.1 Implementation Overview

As specified in the homework description, our network is written in Python (specifically version 3.5.3) using Google's TensorFlow library. Additional packages we used include: Pandas, NumPy, Scikit-Learn (for text preprocessing and vectorizing), and `mlxtend` ("Machine Learning Library Extension" for generating the one-hot).

4.2 One-Hot Vector Input

Neural networks perform best when information is encoded in a fixed-length, easily-decodable fashion. Humans encode words through an ordered series of letters. While this form is compact and well-suited for human consumption, it is not ideal for a machine input. Instead, it is much simpler for the learning algorithm if a word is represented as an N-dimensional bit vector with each of the bits representing one word in the vocabulary. Using this notation, each word would be represented by a vector of $N - 1$ zeros and a single 1 bit; this is the reason why this encoding scheme is known as "one-hot" vectors.

A sentence of v words is represented as an ordered sequence of v one-hot vectors (with some vectors potentially repeated if a word appears multiple times in the sentence). In our base architecture, each of these one-hot vectors is multiplied by the embedding matrix before being fed into the neural network.

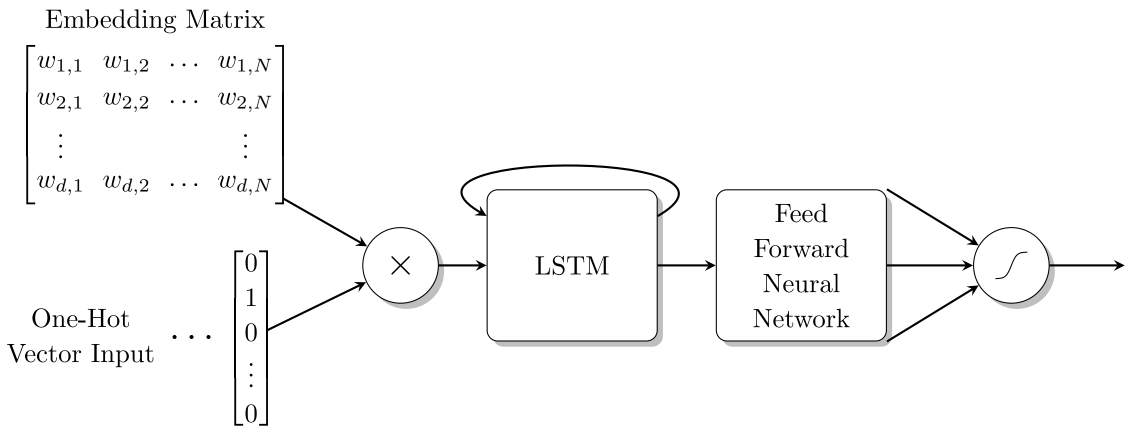

Figure 1: Base LSTM Neural Network Classifier

4.3 Embedding Matrix

As mentioned in Section 3, this dataset's vocabulary size, N, is greater than 11,000 words. Using one's personal computer to train a neural network of that size is impractical and will yield severely degraded results. TensorFlow includes built-in support from **embedding matrices**, which via matrix multiplication map the original sparse N-dimensional space into a much smaller and denser d dimensional space. (In our experiments, we used $d = 25$ as proposed by Ehsan in lecture.)

The embedding matrix is a learned-object that also can encode relationships between words. For example, in one canonical example, the relationship,

$$\text{Rome} \ - \ \text{Italy} = \text{Beijing} \ - \ \text{China},$$

can be shown to be learnable using embedded matrices. This shows that embedding matrices can significantly improve a classifier's performance by deducing an inherent "meaning" to words.

4.4 Long Short-Term Memory Network Structure

Long short-term memory (LSTM) is a subtype of recursive neural networks. LSTMs are a core block in TensorFlow, whose primary hyperparameter is the number of neurons. Too few neurons in the system may cause the LSTM to not learn well. Similarly, too many neurons will result in overfitting. When using the embedding matrix, we observed that approximately 64 LSTM neurons yielded the best results. When we used a public word model as described in Section 8, the number of LSTM neurons had to match the model width, which in the case of Google's word2vec is rank 300.

4.5 Feed-Forward Neural Network Structure

The feed forward network structure we used is shown in Figure 2. The number of input nodes is dictated by the output of the LSTM. Our sole hidden layer has $m = 256$ fully-connected neurons. There are two output nodes (e.g., one for "Donald Trump" and the other for "Hillary Clinton"). A softmax function normalizes the output probabilities to ensure they sum to probability 1. We selected this paradigm as it simplified the TensorFlow implementation without affecting the quality of the results.

Our final feed-forward network design used the rectified linear and sigmoid activation functions for the hidden and output layers respectively. Each neuron in the network also had its own bias input.

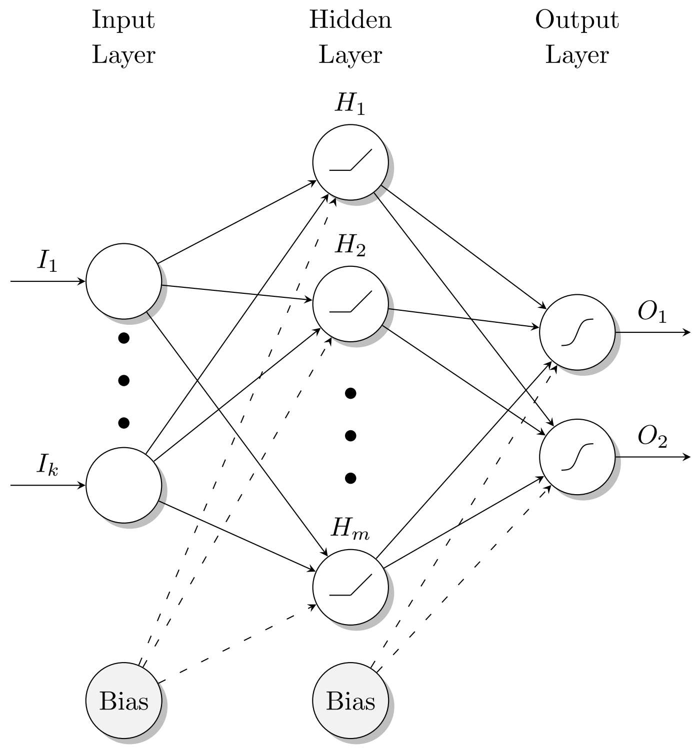

Figure 2: Base Structure of Our Feed-Forward Network

5 Experimental Results

As of the writing of this document, our team is ranked sixth in the Kaggle competition with a minimum log loss error of approximately 0.2. This section discusses only those experiments that align with the base requirements of the assignment. Additional and extra credit experiments are discussed in Sections 6, 7, and 8.

LSTMs have a greater tendency to overfit than most simple feed forward networks. We observed this potential to poorly generalize on this dataset where we could get log loss training errors of approximately 0.05 (or less), but test errors of 0.7, i.e. an increase of more than a order of magnitude. This overfitting was not due to excessive epochs. Rather these large divergences could be observed in fewer than 100 full batch trainings. Even when we did mini-batches, we did not see a significant decrease in the generalization error.

In our experience, LSTMs can be significantly more sensitive to variations in their hyperparameters than simple fully connected feed forward networks. With our optimal configuration, we were able to achieve a minimum log-loss test error of approximately **0.3** with the LSTM. This required the use of an open-source word model as described in Section 8. Using a trained embedding matrix, our optimal score was 0.38. Overall, we expect that with one or two more people and additional time, we could have achieved results similar to some of the other teams.

6 Extra Credit #1: Bag of Words Model

In the "bag of words" model, each textual input, i.e., tweet, is transformed into an unordered set of words. Hence, the contextual and word ordering information is discarded. This approach removes any sequential relation in the data; hence, the LSTM added no specific value for training. As such, we removed the LSTM when performing this experiment and instead trained with just the embedding matrix and the feed-forward network.

Using the previously described neural-network structure, we were able to get 100% accuracy on the complete training set. Likewise, we get a best-case test set error of **0.20070**.

7 Extra Credit #2: Neural Network Experiments

Below are additional experiments we tried beyond the base requirements.

7.1 Extra Credit #2a: Hidden Layer Activation Functions

We experimented with three activation configurations for the hidden layer. In addition to rectified linear, we also tried a "pass-through" activation where the neuron's output was simply $\mathbf{w} \cdot \mathbf{x} + b$ ($\mathbf{w}$ is the weight vector, $\mathbf{x}$ the input, and b the bias term). We also tried to use the sigmoid function for the hidden layer. However, these two other activation functions took more epochs to converge (an increase of approximately 200 to 1,000 training rounds) and yielded worse testing error.

7.2 Extra Credit #2b: Additional Neurons in the Hidden Layer

There is a general (but not strict) correlation between the number of neurons in the hidden layer and the complexity of the function the network can learn. However, additional neurons increase the possibility of overfitting. We experimented with three different quantities of hidden layer neurons, namely 128, 256, and 512. We observed that using either 128 and 512 neurons increased the test set's log loss by approximately **0.24**. In addition, 512 neurons significantly increased the training time (by a factor of two times). In contrast, 256 hidden layer neurons had a training error of $\sim$0.20; that is why we selected $d = 256$ as discussed in Section 4.5.

7.3 Extra Credit #2c: Multiple Feed-Forward Hidden Layers

As illustrated in Figure 2, our feed-forward network only had a single-hidden layer. We settled on this architecture after experimenting with both two and three hidden layers. Similar to the findings in Section 7.2, increasing the complexity of the feed-forward network by adding more layers had a dual deleterious effect of increasing the training time and reducing the reported score on the test set.

7.4 Extra Credit #2d: Optimizer Selection

TensorFlow optimizers control the calculation of error gradients as well as the application of these gradients to variables. TensorFlow includes a suite of different optimizers, with each implementing a different learning algorithm, including: `AdamOptimizer` (Adaptive Moment Estimation), `GradientDescentOptimizer` (Stochastic Gradient Descent), `AdagradOptimizer` (Adaptive Gradient), `MomentumOptimizer` (Momentum Method), etc.

As mentioned in Section 5, tuning an LSTM's learning rate hyperparameter can be challenging. Likewise, the set of learning rates that work well for one optimizer may be mutually exclusive to another optimizer's set of learning rates. In all our experiments, the `AdamOptimizer` significantly outperformed the other optimizers. It converged faster and with lower loss on the training set than all the others. The improvement it provided in the training set's log loss could be as high as **0.5** on the *training* set.

8 Extra Credit #3: Use of Open-Source Word Models

Figure 1 shows the use of an embedding matrix to reduce the dimension of the input and to improve learning performance. Rather than having TensorFlow learn an embedding matrix solely on its own, an alternative is to use publicly available word models. The benefits of this approach include the use of a significantly larger training set and improved embedding matrix coverage for words that appear only in the test set. For example, the Google News word model (which we used), contains over three million words and was trained on more than 100 billion words. In contrast, the tweets in this homework contained only about 11,000 words.

By switching to Google's word model, we saw a log loss improvement of approximately 0.2.