Using Recurrent Networks to Classify Toxic Comments

Anish Balaji,[1] Kevin Chiang,[1] Pranay Kumar,[1] David Zhang[1]

[1]CS 194-129, University of California, Berkeley

As technology improves the ability and rate at which people are able to communicate, an increasingly large need to moderate and filter such communication expands beyond the capabilities of historically manual labor. With many online system turning to machine learning and the like to handle these issues automatically, the performance in accuracy, speed, and generalizability have to be evaluated objectively to determine the efficacy of such improvements. We examine a variety of recurrent neural networks to determine how important each layer of the model is in correctly classifying toxic comments and attempt to achieve state-of-the-art accuracy for sentiment analysis.

Introduction

In recent years, digital content and commentary have taken over the majority of public opinion spheres, and yet, are often known as cesspools of hate and obscenity. These less savory examples of human communication are only exacerbated in impact by the anonymity of content creation and the scales of viewership achieved. Given the overall harm, nearly all platforms have taken a stance to combat toxicity. While the goal is relatively straightforward, the execution of this task is rather complex.

Even excluding the massive gray area that is merely "controversial topics", manually moderating comments has proven to be extremely ineffective. With hundreds-of-thousands of hours of content published, and billions of hours consumed daily, a few thousand moderators can't be expected to keep up with the scale and subjectivity. Enter **recurrent neural nets**! With proven performance with language and sentiment analysis, and the easily consumable samples already labeled by moderators, machine learning seems like a prime candidate to automate the majority of content moderation, and free resources to investigate more serious or gray issues surrounding individual-scale harassment and offensive content disputes.

Some considerations to keep in mind when designing systems where safety and censorship are concerned is some notion of performance and strictness relative to the entire ecosystem of content. This potential issue can be largely mitigated by obtaining a large and balanced data set, but further verified and supported by keeping a close eye on **precision** and **recall** in addition to accuracy. In this context, precision answers the question - *"How many comments flagged for toxicity are actually toxic?"*, and recall answers the question - *"How many of the toxic comments were actually flagged?"*. While the exact industry application of a system like this could have a preference for either metric and accompanying strategy depending on the level of authority given to the autonomous system, it will be important to keep all of these metrics in mind when evaluating the final models.

Related Work

The idea of text sentiment analysis was first proposed by Volcani in 2001 (*1*). Volcani was the first to represent words and relationships between words in a numerical fashion - an embedding matrix! Our work and countless other breakthroughs in text analysis would not be possible without the embedding matrix. One up and coming effort to combat toxicity on the Internet is Perspective (*2*). Created by Google, Perspective is an API that uses machine learning models to score the perceived impact a comment might have on a conversation. The impact scores that the API produces could help comment thread moderators do their job and assist participants de-toxify their posts.

Dataset

We obtained our training and testing data from a Kaggle Toxic Comment Classification Challenge (*3*), which was provided by the owner of the contest, Jigsaw. Overall, the dataset contains over 100,000 comments taken from Wikipedia articles. Each of these comments were rated for toxicity by humans. There are six classes of toxicity we were trying to classify: toxic, severe toxic, obscene, threat, insult, and identity hate. The raw sentences and the labels were provided in a CSV file.

The raw data was quite messy and required extensive preprocessing. To ensure uniformity of comments while training, we changed all data to lowercase and removed newlines. Some comments had random usernames and IP addresses, which we omitted. Additionally, we split all contractions into their full forms (i.e. aren't $\rightarrow$ are not). Lastly, we obtained a list of common stopwords (words which do not contain important significance) and removed all those from our dataset.

Here are two examples of cleaned data, and their respective labels.

- "hey man , really trying edit war . guy constantly removing relevant information talking edits instead talk page . seems care formatting actual info . " $\rightarrow$ No negative sentiments
- "c*cksucker piss around work" $\rightarrow$ Toxic, Severe Toxic, Obscene, Insult

Baseline Model

We wanted our baseline model to not have any deep learning components, as all of our models are quite complex. Ultimately, we used a Naive Bayes-SVM model, based on (*4*). In the paper, the authors describe how using a NB-SVM architecture results in a robust model for sentiment analysis. The model featurizes an n-gram Bag of Words term-document matrix with Naive Bayes. For classification, the model simply uses a SVM to separate the comments into each of the six classes (mentioned above in the Dataset section).

In our training/testing, we found this baseline model to be quite effective. It achieved around 97% accuracy on all datasets. Additionally, this model trained extremely quickly. Fitting all 6 SVMs and producing classification labels each took less than 30 minutes - a fraction of the time it took to train our deep learning models.

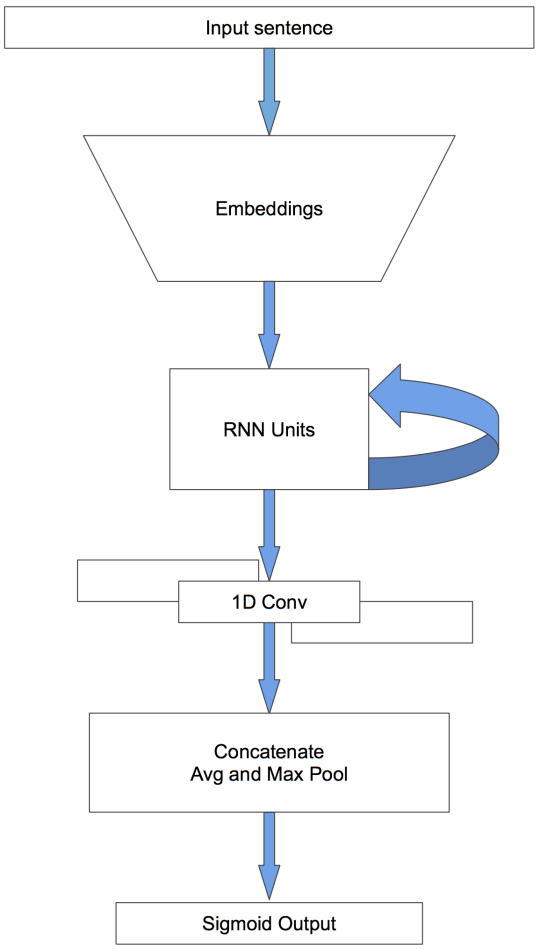
Recurrent Models

We built several recurrent models, comparing the differences in accuracy among similar models. As seen in figure 1, we take the cleaned sentences as input, embed them using an embedding algorithm, pass the embeddings into a recurrence, take a convolution, and output six sigmoid outputs, one for each toxic classification. We focus on comparing different embedding algorithms and different configurations for the recurrent section of the model.

We initially wanted to compare multiple embedding algorithms, such as LSA, Word2Vec, GloVe, FastText, and more state-of-the-art embeddings. However, we narrowed our focus to Word2Vec, GloVe, and FastText; Word2Vec and GloVe represent standard yet effective word embedding algorithms, while FastText represents a newer, faster, and more efficient algorithm to embed words.

For the model recurrence, we focus on training and comparing models using LSTM and GRU units, as well as varying the number of recurrent layers we stacked. We aimed to compare not only the accuracy between these different models, but also the time it took to train these models and how fast they plateaued. Based on these results, we can infer which models are recommended for the highest accuracy on unseen data, which models are recommended for faster training on weaker hardware, and which models serve as a compromise between accuracy and speed.

Figure 1: Model architecture



Model Details

We take in the input sentence and apply a spatial dropout of 0.35. This spatial dropout effectively drops random input words in order to decrease the probability that single words affect the model prediction. For example, profanities usually indicate some sort of toxic comment, but a positive spatial dropout probability allows our recurrent models to learn more features to classify toxic comments, rather than only focus on select profanities and potentially overfit to our training set. Next, we embed our sentence and pass it through our recurrence. We take the output and apply a 1-dimensional convolution. This convolutional layer has 64 filters, each with a kernel size of 3. The outputs of our convolutional layer are then passed through an average pool and a max pool, which are concatenated together. Finally, we pass the concatenated vector through a dense layer with a sigmoid activation, giving us 6 outputs, one for each type of toxic comment. We use the binary cross-entropy loss function and the ADAM optimizer to train our networks.

Word2Vec. We trained our own Word2Vec embedding matrix using the cleaned training data given to us. Each embedded word had to appear at least 5 times; otherwise, they were combined into the UNK token. The window size was 3, the sampling rate was 0.001, and the number of samples for negative sampling was 5 (these are standard values). We ran the algorithm for 100,000 iterations to generate a 50-dimensional and 150-dimensional embedding matrix, but we opted to fully train our final models with only the 150-dimensional embedding matrix due to time constraints.

GloVe. We also trained our own GloVe (*5*) embedding matrix using the cleaned training data. We also specified that each embedded word had to appear in the training data at least 5 times. The window size was set to 15 (this is the standard value). We ran the algorithm for 100 iterations to generate a 50-dimensional and 150-dimensional embedding matrix, but for similar reasons as mentioned above, opted to fully train our final models with the 150-dimensional embedding matrix.

FastText. We decided to use FastText (*6*) as our third embedding method. FastText provides two model options for creating the embeddings: CBOW (continuous bag-of-words) and skipgram. We found that the embeddings produced by these two model types did not produce a significant difference in the training/validation accuracies. As with GloVe and Word2Vec, all words in the corpus had to appear at least 5 times, and we trained our final models with a 150-dimensional embedding matrix.

LSTM. We set the output dimensionality of our LSTM unit to be 128. The activation function and recurrent activation function were the tanh function and hard sigmoid function, respectively; the kernel initialization and recurrent initialization were Xavier uniform and random orthogonal, respectively (these activation and initialization functions are Keras' default parameters). We also set both the input dropout and recurrent dropout to 0.15.

GRU. All parameters for the GRU unit were equal to the LSTM parameters, allowing us to minimize the effect of hyperparameters on our models' accuracy and compare the differences between an LSTM unit and GRU unit effectively.

Tools and Training Details. We ended up with 143,613 unique labeled training samples, with an additional 15,958 labeled samples set aside as a novel dataset. For each RNN-unit/embedding combination, we trained for 5 epochs, with a batch size of 32. Initially, we tried to train in a Jupyter notebook; we soon found out that due to memory constraints and leaks, this was impossible. Thus, we switched to simple Python scripts for all training. We used Keras as our TensorFlow wrapper - the prebuilt RNN units and network layers (i.e. Dropout) helped us immensely while creating the models. We trained our models on two desktops; the first contained an Intel i7 5820K and NVIDIA GTX 980, and the second contained an Intel i7 7700K and NVIDIA Titan Xp SLI.

We also realized that only obtaining statistics after every epoch was not enough to compare the models. Our solution was to insert custom callbacks into the training loop that saved loss and accuracy multiple times per epoch. These extra stats enabled us to better analyze the trends in our results.

Model		Precision	Recall	F1	Novel Accuracy
Word2Vec + LSTM	Epoch 2	0.787	0.714	0.744	0.982
	Epoch 5	0.754	0.698	0.721	0.981
Word2Vec + GRU	Epoch 2	0.785	0.705	0.735	0.982
	Epoch 5	0.755	0.714	0.730	0.981
GloVe + LSTM	Epoch 2	0.802	0.714	0.751	**0.983**
	Epoch 5	0.789	0.697	0.738	0.982
GloVe + GRU	Epoch 2	0.773	**0.753**	**0.758**	0.982
	Epoch 5	0.790	0.710	0.745	0.982
GloVe + 3 layer LSTM	Epoch 2	0.780	0.739	0.755	**0.983**
	Epoch 5	0.764	0.749	0.753	0.982
FastText + LSTM	Epoch 2	0.812	0.716	0.754	**0.983**
	Epoch 5	0.759	**0.753**	0.754	0.982
FastText + GRU	Epoch 2	0.782	0.741	0.754	**0.983**
	Epoch 5	0.797	0.686	0.733	0.982
FastText + 3 layer LSTM	Epoch 2	**0.820**	0.708	0.752	**0.983**
	Epoch 5	0.789	0.718	0.751	**0.983**

Figure 2: Various metrics for our recurrent models

Results

Figure 2 lists the precision, recall, F1 score, and novel dataset accuracy for all models at the end of epoch 2 and epoch 5. We immediately see a significant increase in accuracy for all recurrent models when compared to the baseline accuracy of 0.97. However, among recurrent models, the distinction becomes less clear.

Varying Number of Epochs. When we compare models at the end of epoch 2 versus at the end of epoch 5, we can see that almost all success measures are equal to or lower by epoch 5. This suggests that the models generally overfit the training data significantly, or at the very least plateau by the end of the second epoch. Figure 3 reinforces this observation, where we can see that the training loss continues to decrease and the training accuracy continues to increase in all models, while the novel loss begins to increase after the second epoch and the novel accuracy shows no noticeable increases in all models.

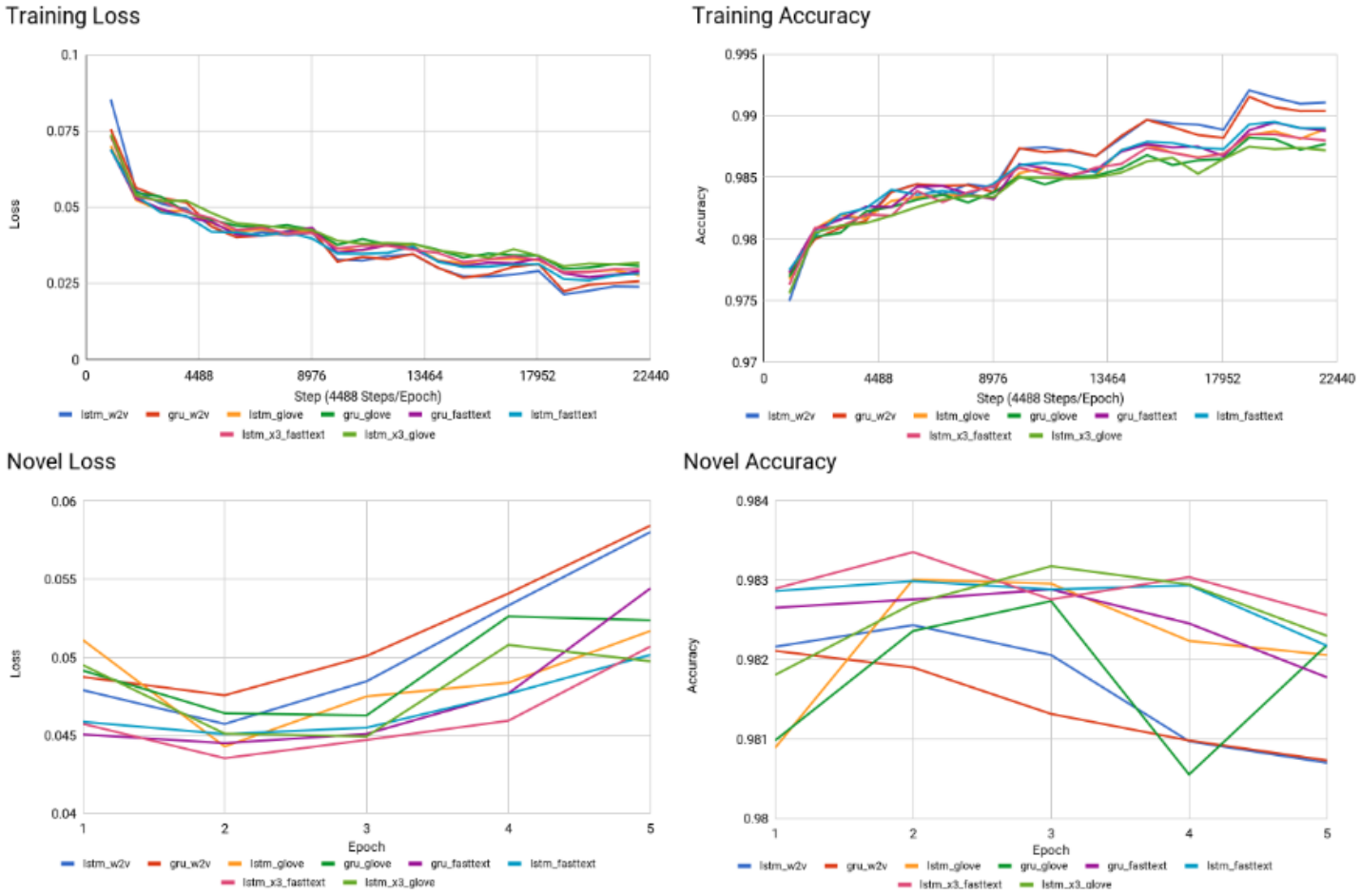


Figure 3: Training and novel data: loss and accuracy

Varying Embedding Algorithms. When we compare models that employ different embedding algorithms, we see that the Word2Vec models lag behind in all categories in comparison to their GloVe and FastText counterparts. The difference between the GloVe and FastText models is negligible and can be reasonably explained by the random weight initializations. In addition, figure 3 shows that the Word2Vec models overfit more severely than the GloVe and FastText models.

Varying RNN Units. When we compare models the employ different RNN units, we generally see that the LSTM models have higher success measures at the end of epoch 2 than their counterparts, but the GRU models have higher success measures at the end of epoch 5. Since LSTM units have more flexibility built in than GRU units, we can infer that this added flexibility helps the LSTM models perform better earlier. However, this also causes the LSTM models overfit the training data more than their corresponding GRU models, which is exactly what we see in our final results. One especially important fact not captured by the table is how fast the GRU models train: when all other parts of the network are equal, GRU models are able to train 33 percent faster than LSTM models.

Varying Recurrence Layers. When we compare models that employ a different number of recurrent layers, we see very little variation, if any. Thus, for these specific configurations, extra recurrent layers do not help boost accuracy for this task.

Sample Predictions. Figure 4 shows a few novel comments and their predicted outputs from the GloVe + LSTM model. As shown in the last case, we see that unstructured and rambling comments often confuse the models, predicting more toxicity than the comment implies. In general, the recurrent networks are able to capture negative sentiment accurately, while allowing for constructive criticism.

Comment (Censored)	Toxic	Severe Toxic	Obscene	Threat	Insult	Identity Hate
Thank you for understanding. I think very highly of you and would not revert without discussion.	~0	~0	~0	~0	~0	~0
The rapcore article cites 15 sources which back up all information stated, including the basics of the genre. The only person who is evading anything is Ridernyc, who avoids any instance in which he would be forced to admit that he is wrong.	~0	~0	~0	~0	~0	~0
DJ Robinson is g*y as hell! he sucks his d*ck so much!!!!!	0.999	0.163	0.976	~0	0.919	0.424
I WILL BURN YOU TO HELL IF YOU REVOKE MY TALK PAGE ACCESS!!!!!!!!!!!!!	0.920	0.007	0.120	0.796	0.060	0.007
F*ck off, you anti-semitic c*nt.	~1	0.565	0.999	0.012	0.945	0.972
Grrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr You suck suck suck suck suck suck suck suck su	0.971	0.974	0.996	0.008	0.902	0.005

Figure 4: Selected toxicity predictions from GloVe + LSTM

Conclusions and Future Work

For reference, the winning Kaggle entry obtained a novel accuracy of 0.988. While our models did not achieve state-of-the-art accuracy, we came very close without having to deviate from very straightforward and simple recurrent models. This is an interesting result itself; for these types of sentiment analysis tasks, simple recurrent networks are able to outperform non-recurrent networks by a significant margin, and complex modifications to these networks only result in modest increases in accuracy.

As discussed above, many of the recurrent models we compared obtain very similar success metrics. Thus, there is no best model. It seems that both GloVe and FastText work equally well for the embedding algorithm, while LSTMs work slightly better for the RNN unit. However, it is also not wrong to favor a faster-training model by switching LSTM units for GRU units.

To increase the accuracy of our models further, we could continue to tune our hyperparameters or increase the dimensionality of our recurrent units. We could also experiment with additions to our networks, such as adding attention modules or skip connections in our recurrence. We could also explore other networks other than recurrent neural networks entirely, such as end-to-end memory networks. A more diverse training dataset would also be an option worth exploring, since there are other social networking platforms with more extreme comments.

Other future directions also include attempting to classify other sentiments in sentences, applying sentiment analysis to speech and audio samples, and implementing these models to create a platform where viewpoints can be discussed in a civil manner.

Lessons Learned

Some of the challenges we faced along the way included training Keras models in Jupyter notebooks. Keras was not able to fully release memory allocated through Jupyter, leading to memory leaks on the GPU. We solved this by giving up on notebooks entirely and saving model data manually. Similarly, Keras itself has a bit of its own learning curve over Tensorflow, but once we got a handle of it, we could see the appeal of using a higher level API on top of Tensorflow. Also, by default, the models would only give us loss and accuracy at the end of each epoch, so we had to write custom callbacks to record more detailed data throughout training.

Another challenge was simply training all of the models within the given timeframe, since the time to train each model for 5 epochs essentially took an entire day, which slowed down our hyperparamater search and potentially led us to work with non-optimal hyperparameters. Thus, we learned how to minimize the number of epochs and parallelize the training of multiple models in order to efficiently tune our hyperparameters within a small timeframe; we hope to employ this tactic in our hyperparameter search in future projects.

It was quite surprising that all of the recurrent models that we tried were similar in all success metrics, and that made it a bit hard to make any conclusions from our experiments. However, we extended our experiments and ended up being able to make a different conclusion than what we expected. We also eventually found some small variations that matched up with what we expected (Word2Vec does the worst out of all embeddings).

Team Contributions

Anish:

- Initial end-to-end LSTM model
- Standard LSTM models with FastText
- 'Deep' LSTM model with FastText

Pranay:

- Cleaned all Training/Test Data
- Produced FastText embeddings
- Standard GRU models with FastText
- Various sections of the paper and poster

Kevin:

- Extended base RNN model to train new model architectures
- Produced GloVe embeddings
- Wrote custom callbacks for success metrics
- Hyperparameter search
- Trained GloVe models
- Visualized RNN results and novel predictions
- Half of presentation, various sections of paper and poster

David:

- Extended base RNN model to train new model architectures
- Produced Word2Vec embeddings
- Implemented checkpointing/saving for RNNs
- Hyperparameter search
- Trained Word2Vec models and deep GloVe LSTM model
- Implemented and trained baseline model
- Half of presentation, various sections of paper and poster

References

Our code for this project can be found at https://github.com/PranayJuneCS/Toxicity

1. System and Method for Determining and Controlling the Impact of Text. Volcani, Y., Fogel, D., 2001.
2. http://www.perspectiveapi.com/.
3. https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge.
4. Baselines and Bigrams: Simple, Good Sentiment and Topic Classification. Wang, S., Manning, C., 2012. ACL.
5. GloVe: Global Vectors for Word Representation. Pennington, J., Socher, R., Manning, C., 2015.
6. http://www.fasttext.cc.