Unbiased Toxicity Competition Report

Jie Hao Liao (jhliao@utexas.edu), Tongrui Li (tongrui1998@gmail.com)

Abstract
Using machine learning to automatically detect toxic comments to promote a healthier environment has been one of the trending tasks recently. researchers from Jigsaw observed previous models tends to mark sentences with biased word “e.g. gay” as toxic regardless of the content of sentence. In this paper, we attempt to eliminate the bias introduced by the model by examining and using state of the art language models as well as traditional embeddings. We experimented with a variety of models using CNN, LSTM, and attention and eventually achieved a weighted ROC-AUC score of 0.9299 in the Kaggle competition.

Introduction
Define toxicity as anything “rude, disrespectful, or otherwise likely to make someone leave a discussion.” Online interactions have often given a sense of security from behind the screens, and that sense of security allows for harmful intentions without retribution. One can often find toxic speech on online forums, gaming communities, etc. Thus, the Conversation AI team, founded by Jigsaw and Google, organized multiple Kaggle competitions to research state-of-the-art models to accurately identify toxic speech and reduce them. We’re currently participating in the most recent one.

In the past, Jigsaw hosted competitions on toxicity to identify toxicity as well as different subtypes of toxicity. Since most of the toxic comments were frequently separable with identity attacks (e.g. “gay”) in the data of that competition, those models previously created frequently identified toxicity as comments that mentioned an identity. The current competition aims to identify toxicity but remove the unintended bias.

This report aims to evaluate some state-of-the-art models and techniques for identifying toxicity and provide possible techniques for eliminating identity biases. Techniques covered include using Long Short-Term Memory cells, Convolutions, Attention, and Pooling. Models covered include using the standard Recurrent Neural Networks, Convolutional Neural Networks, BERT, and ELMo.

Dataset
Our dataset, provided by Jigsaw, is sourced from the shut-down Civil Comments platform and extended by Jigsaw. Jigsaw pooled human annotations of toxic conversational attributes on the open archive for several examples of online conversation texts.

In the training set of the data, every row is provided an example comment and a fraction of human raters who believed the comment should be classified as toxic. Jigsaw would like the created models to achieve similar performance as the human ratings, where if more than 50% of the raters believed the comment should be toxic, the comment will be considered to be in the positive, toxic, class. Several other attributes, which are also fractions of human raters who rated the attribute as positive, are also given. These include the subtypes of toxicity, which needed to be predicted in the previous competitions, and possible identities mentioned in the comment.

In the evaluation set of the data, every row is once again provided an example comment and a fraction of human raters who believed the comment should be classified as toxic, but all other attributes provided in the training set is not included in the evaluation set.

Evaluation
Competitors are expected to output a probability of an example comment to be rated as toxic. Thus, the Conversation AI team developed a new metric which combines several submetrics to balance overall performance with various aspects of unintended bias (Borkan et al., 2019).

For the overall performance of the model, the evaluation uses ROC-AUC on the entire evaluation set.

Then, predictions on multiple identity subgroups are used to evaluate how well the model tackle bias in the evaluation set. The identity subgroups include ‘male’, ‘female’, ‘homosexual gay or lesbian’, ‘Christian’, ‘Jewish’, ‘Muslim’, ‘black’, ‘white’, ‘psychiatric or mental illness’ as categories.

For every subgroup, 3 different Bias ROC-AUCs are calculated. The first is the subgroup AUC, which is the ROC-AUC of predictions but only on the data which has information for that subgroup. The subgroup AUC judges how well the model does distinguishing between toxic and non-toxic comments that mentions the identity. The second is the BPSN AUC, or background positive, subgroup negative AUC, which is the ROC-AUC of predictions but only on the non-toxic examples that mentions the identity and the toxic examples which do not mention the identity. The BPSN AUC judges how well the model does not confusing the non-toxic identity examples with toxic examples that does not mention the identity. The third is the BNSP AUC, or background negative, subgroup positive AUC, which is the ROC-AUC of predictions but only on the toxic examples that mentions the identity and the non-toxic examples which do not mention the identity The BNSP AUC judges how well the model does not confusing the toxic identity examples with non-toxic examples that does not mention the identity.

Then, a generalized mean over the subgroups is calculated for every Bias AUC. The generalized mean weights more heavily towards lower scores to encourage the competitors to balance out the lower subgroup ROC-AUC scores. The final metric then averages all 4 ROC-AUCs, the overall ROC-AUC and the 3 Bias AUCs, as the final score of the model.

Validation

The Conversation AI team provided a benchmark model for the competitors to improve on. The benchmark model includes the entire evaluation algorithm that computes the four different ROC-AUCs if one were given both predictions and the gold label of a comment, which is what our models use as the criterion for validation. Our model samples 20% of the training data as the validation set and uses the rest as the training set for the validation. Validation of all ROC-AUCs is calculated every half epoch for two epochs. The validation results were consistent with the evaluation test results about 30% of the time, where an increase in the validation score generally indicated an increase in the evaluation score. Cross validation as a more solid validation method was also implemented, but due to the long training time of the models and time constraints proposed by the hard deadline of the project, the method has been temporarily abandoned.

Tokenizer

A tokenizer is required for the purpose of splitting a sentence into individual tokens, where each token should closely resemble an individual entity in the sentence. Since the dataset samples examples of online conversational texts, a lot of them were not written in a standard grammatical format, and hence could not be tokenized well with standard tokenizers. To counter this problem, we decided to use the Tweet Tokenizer from NLTK, which by design addresses tokenization in the Twitter texts domain. The addressed domain is similar to the conversational text domain of the entire dataset.

Notable features of the Tweet Tokenizer include: 1) preserve handles and hashtags in tokenization, 2) reduce repeating letters in the token (e.g. “baddddd” -> “baddd”) for a higher chance of a more uniform vocabulary, 3) preserves emojis, or facial expressions in text, and 4) limits consecutive repeating tokens (especially for punctuations) to occur no more than 3 times at once.

In addition, due to BERT’s requirement of needing the input to be tokenized in Word Piece fashion (Wu et al, 2016), BERT tokenizer from the huggingface repository has been used instead of the NLTK tokenizer to meet such requirement.

Preprocessing

Our preprocessing involves creating a more generalized and compact tokenization of the sentence to reduce noise in the dataset. Thus, most of the preprocessing below has been built around using the Tweet Tokenizer.

1) Remove non-alphanumeric characters from URLs. We found that a lot of the example texts which includes URLs are primarily advertisement spams, and spams examples have mostly non-toxic ratings. We want our model to be able to recognize spams as well as consider the toxicity of parts of the spam link, so we remove all non-alphanumeric characters and keep tokens that are common to all urls (e.g. ‘www’, ‘https’) and hopefully some English words. Then, we keep track of number of URLs that appeared in the example as a statistical feature.

2) Convert unicode emojis to words and remove uncommon characters. We define uncommon characters as characters that are not alphanumeric and not in the list of punctuations defined in the Python string library (!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~). These uncommon characters are mostly unicode emojis and unicode characters that are highly frequent in advertisement spams, which negatively correlates with the gold label. Also, We keep track of the number of uncommon characters that appeared in the example as a statistical feature.

   Example comment:
   Ｔｈｅ ｇｏｖｅｒｎｍｅｎｔ ｌｉｅｄ ｔｏ ｕｓ！
   —
   Ｉｎ ２００８ ...
   Preprocessed comment:
   The government lied to us! -- In 2008 ...

3) Convert to lowercase. Also, we kept track of the number of uppercase letters that appeared in the example as well as tokens which were all uppercase letters as statistical features, which weakly correlated with toxicity.

4) Expand contractions and remove possessives. The Tweet Tokenizer does not expand contractions and do not separate apostrophes in tokens, so for a more generalized representation of the sentence, we expanded all contractions the examples. Similar reasoning could be said for possessives.

   Example comment:
   I don’t understand it. I don’t understand Jennifer’s question.
   Preprocessed comment:
   I do not understand it. I do not understand Jennifer question.

5) Replace list items with @LIST tag. We used tags with @ as the first character because the Tweet Tokenizer preserves Twitter handles that starts with @, and the @ tags do not have a pre trained embedding and is not part of the possible vocabulary in the domain.

6) Replace numbers with @NUMBER tag. We replace uncommon tokens in the examples with @UNK tags, and common numbers are especially rare. Thus, we replace possible number representations (e.g. 100, 2/3) with the @NUMBER tag.

7) Process certain punctuations and remove the rest:
   - Remove closing punctuations (e.g. [], (), {}, <>).

- Group !? together (e.g. ! ! ! ? !!? to ??!!!!!). A high number of these in an example usually correlated with a toxic label, but we want the tokenizer to generalize and reduce the number of these appearing in the tokenization. Then, we kept track of the number of them in the examples as a statistical feature.
- Replace * with - (e.g. sh*t to sh-t). The amount of censored words in a sentence correlates highly with a toxic label. However, the Tweet Tokenizer tokenizes * as a single entity, so we replaced it with -, which the Tweet Tokenizer do not tokenize as a single entity. It could also be argued that * could be a single entity by itself as a more generalized representation of a censored word, but we use pretrained embeddings which includes censored words, so we want to associate that with the entire word instead of the * entity.
- Replace & between tokens with 'and' (e.g. 'this & that' to 'this and that') and & in tokens with n (e.g. R&B to RnB). Similar reasoning as * because the Tweet Tokenizer tokenizes & as a single entity.
- Replaces many punctuations grouped together (e.g. %@#!) with @CENSOR tag.
- Remove +=\^`|~ punctuations from the examples.

8) Replace text emojis with their respective tags (e.g. :) to @SMILE). We only did this for the most common text emojis to unify their similar forms (e.g. :) :-) =)) as they possibly could have similar meanings in the examples.

From these, we reduced the vocabulary size of the examples by approximately 50,000 and generalized their representation in the model by a small amount, which consistently improved our AUC-ROC scores by 0.5.

Embedding

Embeddings provides a method to transfer a list of strings to a n-dimensional vector in order to make the string more comprehensible to the model. For the purpose of this paper, pre-trained embeddings from a variety of sources had been used for training the models. For the sake of runtime, only a subset of all embeddings was selected for each model. We utilized both traditional models such as Glove and Fasttext as well as some of the state-of-the-art deep models. A comprehensive list of all embedding used is shown below:

Glove (Common Crawl, Twitter) (Pennington et al.)
A few different pre-trained glove vector has been published by the Standard NLP group. For this paper, the 840 billion tokens with 300 dimension embedding trained on the text provided by Common Crawl and the 27 billion token with 200 dimension embedding trained on 2 billion tweets has been used. Embeddings trained on the Common Crawl dataset provides a more comprehensive vector and being less likely to suffer from missing embeddings due to its enormous training set and token space. Embeddings trained on Twitter tweets may yield vectors that are more suitable to the portion of that that differs from the standard domain of English, and hence were used in some of the models.

FastText (Bojanowski et al, 2017; Joulin et al., 2017)
We used pretrained fasttext embedding on 2 million word vectors trained on Common Crawl (600B tokens).

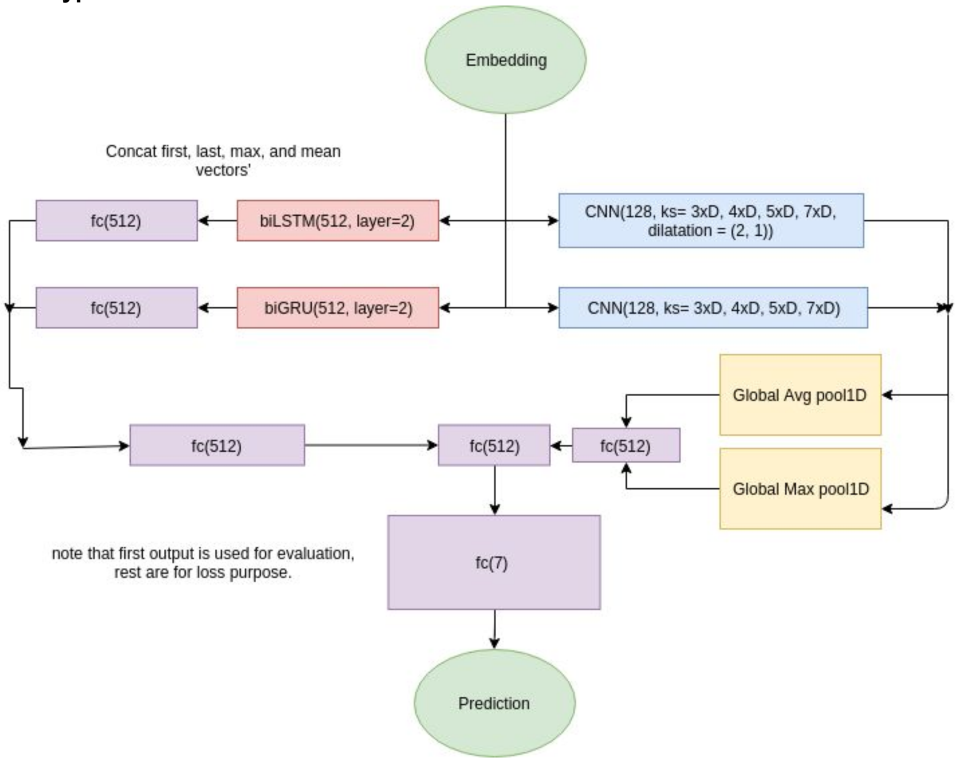
ELMo (Peters et al., 2018)
ELMo utilizes advantage of deep networks to create embeddings for a sentence by making the recurrent layers learn to predict both the next and previous token in the sentence on top of a character level CNN. Outputs from the last recurrent layer have been used as embedding from the model.

BERT (Devin et al., 2018)
Bert utilizes a transformer based model to create embeddings for the input words. It has been chosen as an embedding due to its property of yielding high results with substantially less training. Pretrained uncased base embedding was used in the model. In addition, we also fine-tuned the above embedding on the dataset and use that as an embedding layer.

Model

Up to this point, we had experimented with a few different model architectures. All of our models and their variants are listed below:

First type:


Embedding
Concat first, last, max, and mean vectors'
fc(512)
biLSTM(512, layer=2)
CNN(128, ks= 3xD, 4xD, 5xD, 7xD, dilatation = (2, 1))
fc(512)
biGRU(512, layer=2)
CNN(128, ks= 3xD, 4xD, 5xD, 7xD)
Global Avg pool1D
fc(512)
fc(512)
fc(512)
Global Max pool1D
note that first output is used for evaluation, rest are for loss purpose.
fc(7)
Prediction

CNN

A pure convolution network modified from the original TextCNN paper (Kim et al 2014) The main idea remained the same- using multiple 2d convolution rectangular filters followed by a pooling layer to extract information from the embedding layer. Instead of having 3 different convolution filters with kernel size 3, 4, 5, our model utilizes 4 different convolution filter with kernel size 3, 4, 5, and 7 and increased the output channel to 128. In addition, 4 additional convolution filter with a sequence dilation of 2 was applied. Both average pooling and max-pooling was used to pool the result from the result of the convolution and the output is hence concatenated. Finally, the result is passed through a dense layer to get the desired output. Note that due to the loss design, the output of the dense layer is 7 instead of 1.

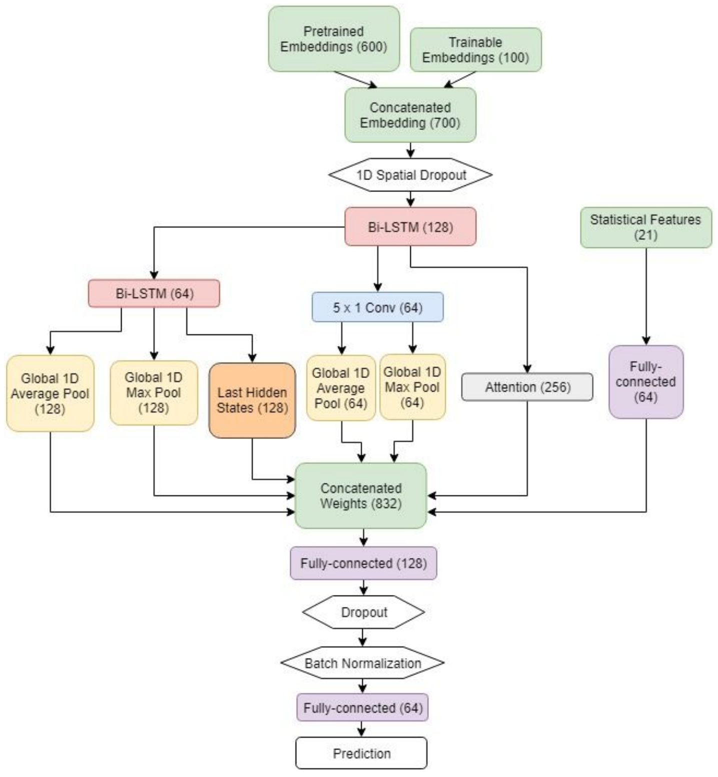
Bi-LSTM + CNN

To improve the scoring, a 2 layer bidirectional LSTM with 512 units is inserted at a parallel fashion to the CNN. Their results are then concatenated together and then pass through a dense layer. Outputs from both ends of the LSTM are obtained. In addition, two additional hidden vectors were obtained through max pooling and average pooling over the sequence dimension (Howard et al 2018). The 4 vectors were then concatenated and sent through a dense layer to reduce the dimension to 512. The output of the CNN was also passed through a dense layer to reduce dimension down to 512. The output of the LSTM and CNN is hence concatenated and then fed through a dense layer to get the desired output.

Bi-GRU + Bi-LSTM + CNN

We also varied the previous LSTM-CNN model by adding a bidirectional GRU with 512 units on top of the LSTM. The hidden vector at both ends plus the average pooled and max pooled vectors were obtained in a similar fashion from the GRU layer. The 4 hidden vectors are then concatenated and fed through a dense layer to reduce dimension to 512, after which it is combined with the reduced vector from LSTM and then fed through a dense layer to 512. The rest of the pipeline is the same as above.

Second model:



Bi-LSTM + Bi-LSTM + Statistical Features

We found that stacking Bi-LSTMs consistently do better than models that use a single recurrent layer. The first Bi-LSTM has 128 units and the second Bi-LSTM has 64 units. We concatenate the hidden states at both ends of the second Bi-LSTM with the average pool and max pool of concatenated forward and backward hidden states of all time steps in the second Bi-LSTM.

We also put the 21 statistical features through a fully-connected layer with Tanh activation to get 64 hidden units, and concatenated it with the outputs of the second Bi-LSTM.

The 5 hidden states are then fed through a fully-connected layer with Tanh activation to reduce the dimension to 128, and then another one with Sigmoid activation to output the probability prediction.

Bi-LSTM + Attention + Bi-LSTM + Statistical Features

While the second Bi-LSTM outputs good weights for prediction, we believe it could potentially convolute some useful information which exists in the first Bi-LSTM output. Thus, we applied Attention over the concatenated forward and backward hidden states of all time steps in the first Bi-LSTM layer.

Then, we concate the attention layer information with the 4 hidden states of the second Bi-LSTM layer mentioned in the previous model and fed it through the same stacked fully-connected layers of the previous model for the output.

Bi-LSTM + Attention + CNN + Bi-LSTM + Statistical Features

With similar reasoning as the previous model, we believe a convolution layer of the first Bi-LSTM can generate different representations of a comment that is not shown in both the Attention and the second Bi-LSTM layer.

We apply a 1D convolution with a kernel size of 5 and 64 channels over the concatenated forward and backward hidden states of all time steps in the first Bi-LSTM layer. Then, we take the global max pool and average pool of the 64 channels and concatenated with the Attention layer and 4 hidden states of the second Bi-LSTM layer mentioned in the previous model. Again, we fed the concatenated vector through the same stacked fully-connected layers of the previous model for the output.

Training Method

Method:

Since all of the models are variants from deep networks, gradient descent was used to train the models. Each of the model listed above was trained independently from each other with all mention layer initialized from scratch. Mini batches of data first went through the preprocess, then was tokenized GPU acceleration using local machines, Kaggle, and AWS had been used to facilitate faster training.

1 dimensional spatial dropout has been applied on the embeddings before training. Dropout and batch normalization has been applied on the outputs of the first fully-connected layer.

Hyperparameters
We had conducted a random search of parameters. They are slightly different for each model below are the values that generally worked well:
Optimizer: Adam
Learning rate: 5e-4, 1e-3
Max length of the sequence: 65, 128, 185, 200
Epoch: 1, 2, 4
Batch size: 16, 32, 128, 512

The learning rate did not seem to greatly affect the performance of the model. A lower learning rate made the model required more epoch for the model to converge.
We found that it is generally more favorable to increase the sequence length to a higher value. However, a higher max length of sequence value leads to a higher cost in runtime in general.
The number of epochs varied inversely with the learning rate. A smaller learning rate generally required more epoch to train. For the learning rate above, we found that the loss generally stopped decreasing significantly after 2 epoch. 4 epoch do have some advantage over 2 epoch but at the cost of double the runtime.
Because of the enormous training data, a higher batch size significantly benefited the training time. No significant evaluation difference was observed between batch size of 16 and 512.

Loss (Approximate ROC AUC (Yan L. et al., 2003))
One of the most fundamental ways to eliminate bias is to adjust the training objective to incorporate the bias. Since the training objective is not differentiable by nature, we used an approximation of the ROC AUC loss. This achieved better results than the Weighted BCE Loss when the models were predicting with 90% accuracy on validation, but started to overfit very quickly.

Weighted BCE Loss
One of the commonly used objective for a binary classification task like this is the binary cross entropy loss, which maximizes the log likelihood of the probability of the objective. The loss is structured as the sum of two BCE logit loss. The first loss adjust weights according to the subcategory as shown below:

Π is an indicator function that outputs either 0 or 1.
$loss_1 = BCE(weight = mean(1 + \Pi(identity\ attack) + \Pi(toxic\ and\ no\ identity\ attack) + \Pi\ (not\ toxic\ and\ identity\ attack))$
$loss_2 = BCE\ for\ sub\ category\ prediction$
$loss_{weighted\ BCE} = loss_1 / mean(loss_1's\ weight) + loss_2$

In an intuitive aspect, the above weight value puts a heavier weight towards the situation where the sentence has an identity attack but not classified as toxic and vice versa.

Evolution (Salimans et al., 2017)
We also tried to apply a two direction hill climbing evolutionary algorithm by applying a small gaussian noise to the parameters of the model (Salimans et al 2017). Hence suppose Θ represents all the parameters of the model and G is a random gaussian noise:

1. $\Theta_{pos} \rightarrow \Theta + G$
2. $\Theta_{neg} \rightarrow \Theta - G$
3. Get a batch from training sample and evaluate
4. model performance on Θ_{pos}, Θ_{neg}, Θ, determine Θ_{best} from the previous set
5. $\Theta \rightarrow \Theta_{best}$
6. Repeat

Hence, the reward obtained from step 4 can be set as the true weighted ROCAUC matrix since no gradient calculation is required. The model should therefore converge a better optimal position.

Results, Trials, and Errors

Model	**Validation score (%)**	**Test score (%)**
Baseline (CNN)	-	88.22
CNN (GLoVe Twitter Dataset)	-	88.03
CNN (GLoVe Twitter + Common Crawl Dataset)	-	88.85
LSTM only (ELMo)	-	90.53

CNN (GLoVe Twitter + Common Crawl + Fine-tuned BERT Embeddings)	-	90.86
CNN + LSTM (GLoVe Twitter + Common Crawl + Fine-tuned BERT Embeddings)	-	91.87
CNN + LSTM + GRU (GLoVe Twitter + Common Crawl + Fine-tuned BERT Embeddings)	-	91.701
Evolved CNN + LSTM + GRU (GLoVe Twitter + Common Crawl + Fine-tuned BERT Embeddings)	89.930	-
Fine-tuned BERT	-	90.950
Bi-LSTM (300d of replacing GLoVe Common Crawl, FastText, 300d Trainable Embedding with each other)	-	1 Epoch: 90.992 2 Epochs: 91.725
Bi-LSTM (700d of concatenated GLoVe Common Crawl + FastText + 100d Trainable Embedding)	-	91.848
Bi-LSTM + Bi-LSTM + Preprocessing (700d of concatenated GLoVe Common Crawl + FastText + 100d Trainable Embedding)	1 Epoch: 91.60 2 Epochs: 92.23	92.29
Bi-LSTM + Bi-LSTM + Preprocessing + Statistical Features (Same embeddings as above)	1 Epoch: 91.80 2 Epochs: 92.30	-
Bi-LSTM + CNN + Bi-LSTM + Preprocessing (Same embeddings as above)	1 Epoch: 91.91 2 epochs: 92.22	-
Bi-LSTM + Attention + Bi-LSTM + Preprocessing (Same embeddings as above)	1 Epoch: 91.90 2 Epochs: 92.29	92.28
Bi-LSTM + Bi-LSTM + Preprocessing + Global Max and Average Pool of 2nd Bi-LSTM (Same embeddings as above)	1 Epoch: 92.03 2 Epochs: 92.34	92.22
Bi-LSTM + Attention + CNN + Bi-LSTM + Preprocessing + Global Max and Average Pool of 2nd Bi-LSTM (Same embeddings as above)	1 Epoch: 92.00 2 Epochs: **92.35**	-
Ensemble of previous models (majority voting)	-	**92.99**

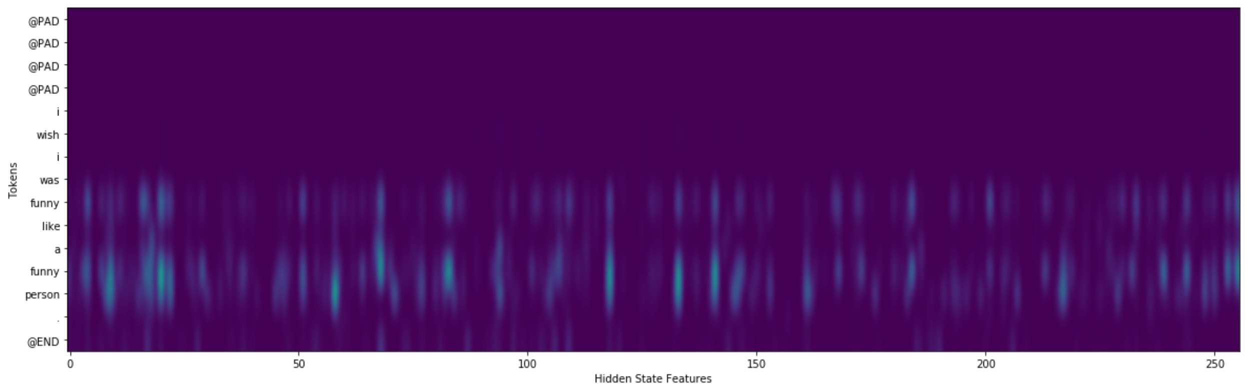
The performance of each of our model are shown above. Due to time constraint and an allowance of a maximum of 5 submissions per day, we were unable to complete the entire table as intended. Validation score represents the weighted AUC ROC on a 20% sample of the training set. Test score represents the actual benchmarked score conducted on the test set on kaggle.

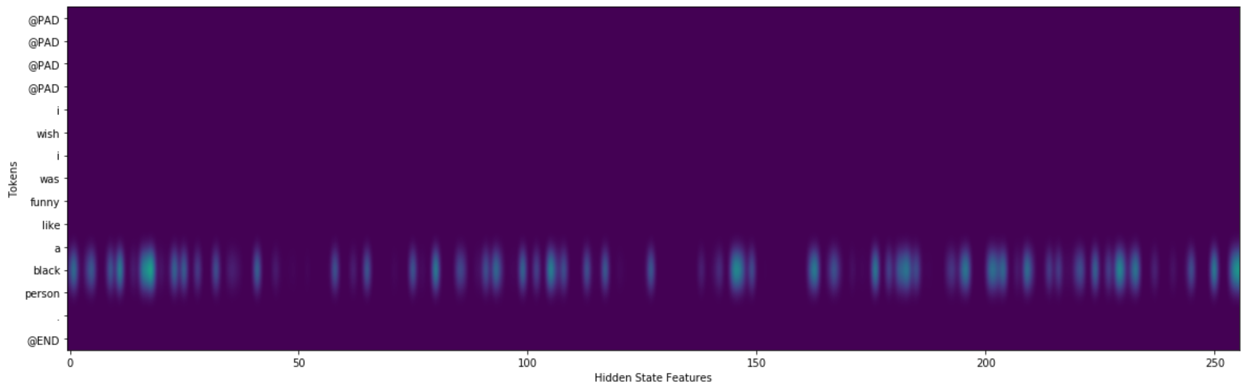
The baseline model is published by the official host from kaggle. As seen from the result, CNN along generally performed worse than LSTM possible due to the inability to have a more direct reasoning for words that are far away. Implementing a more sophisticated embedding such as BERT improved the result roughly by 2%. CNN with LSTM improved roughly 2% than CNN along counterpart. Having a GRU layer on top of LSTM did not seem to benefit the model much. In addition, evolution did only minimal work

Using BERT and ELMO along did not seem to yield a good result by them self possible due to the fact that some comment was not in standard English.

For the Bi-LSTM models, it seems that preprocessing and stacking Bi-LSTMs on top of each other generated the best results, improving the ROC AUC score by 0.5%. 2 epochs of training for all models consistently did better than 1 epoch of training. While CNN, Attention, Max, and Average Pooling on top of the stack Bi-LSTMs did not make much of a difference in performance, combining all of them improved the ROC AUC score by approximately 0.05% in validation.

Below are sample weights of the activations of the attention layers on the hidden states units of the first Bi-LSTM layer. The sentence “i wish i was funny like a funny person.” is compared against “i wish i was funny like a black person.” We can see that in the first output, the model pays strong attention to the word ‘funny’. However, all this information is weak when used in the presence of the word ‘black’. The prediction outputs are negative for the first sentence and positive (is toxic) for the second, indicating that our models did not tackle biases well and still overfit the identity words to correlate greatly with toxicity.





We finally conducted an ensemble of all of our top scoring models. The final ensemble granted roughly a 1% increase in terms of the weighted ROC AUC score.

Conclusion

In this paper we experimented with a variety of model and embeddings to perform well in toxic comment classification while eliminating identity biases. We ended up at results that do well in toxicity classification, but we did not do well at eliminating identity biases. We believe that if given more time, we could predict the identity attribute of each row, which we could then use as weights on model to eliminate bias.