Toxic Comment Classification Challenge

Carlos Arenas Gallego
Universitat Politecnica de Catalunya
Barcelona, Catalonia/Spain
carlosargal@gmail.com

Itziar Sagastiberri Fernandez
Universitat Politecnica de Catalunya
Barcelona, Catalonia/Spain
itzi.sagastiberri@gmail.com

Mireia Gartzia Ibabe
Universitat Politecnica de Catalunya
Barcelona, Catalonia/Spain
mgartziai@gmail.com

Abstract

Toxic comments are all over the internet, especially thank to the anonymity that this platform ensures. Companies like YouTube are currently very interested in automatically recognising these kind of comments to be able to either eliminate them or hiding them to younger audiences. In this paper, we took an already existing project dealing with this task and tried to improve the performance, which we managed to do slightly. However, this is still an ongoing problem as a small error over the huge amount of comments still means quite a lot of them are not correctly classified.

1. Introduction

In the current world that we live where we spend plenty of hours in the Internet we are constantly communicating with other people through different platforms. But what a lot of these platforms have in common is anonymity, which sometimes leads to people making quite offensive comments that they wouldn't otherwise do. That is why, it is very important to be able to identify toxic comments. The problem for this, is that webs like YouTube, Wikipedia... have an enormous amount of comments, so discerning those that are toxic among the huge amount of normal comments is a task that humans can't do. This is why, deep learning is a perfect method to help with this, since it works better with more data.

There are some difficulties to this task, which involves the orthography that may not make it easy to recognize the toxic words, or words that can be used in an offensive way although they are not actually an insult, like gay.

There is already a competition in Kaggle about this, so what we decided was to take one of the projects there and try to improve their results, which were a 98.22% of accuracy and 0.0482 of loss. For these, we want to use different configurations learnt in the seminar and see their effect in the task. This way, we would like to truly learn what we saw in class.

The categories we will use for the classification of toxic comments are: toxic, severe toxic, obscene, threat, insult and identity hate. Finally, the database is the one in the Kaggle competition, which consists on 159571 comments taken from Wikipedia comments and categorized in those 6 classes.

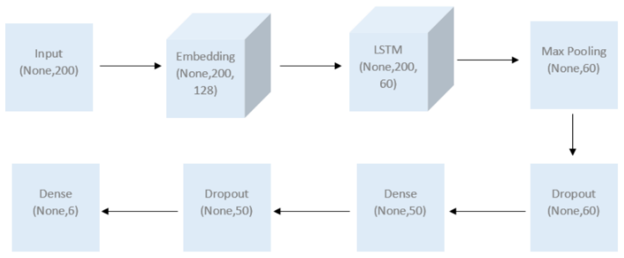


Figure 1: System architecture.

2. Baseline architecture

The project we took as a reference is the one for beginners in Kaggle, "Tackling toxic with Keras". The architecture of this project is the one shown in Figure 1; as we can see, it consists on an embedding layer, a LSTM with Maxpooling at the output and two fully connected layers with a dropout layer before each of them. This architecture has 6 outputs, representing each of the 6 classes, with a value of 0 or 1 depending if they fit into that category or not.

Let's now explain the preprocessing needed for the data and what each of the layers is for. The first thing we need to do is treat the data to be able to feed it to the LSTM. What it does is take words separately from each sentence, which receives the name of tokenization. The next step is indexing, which basically means constructing a dictionary with the words and assigning them a number, so that the final representation of the sentence would be a vector with a number in each position, representing the words. To better understand, this would be an example:

1. I love cats and love dogs

2. "I","love","cats","and", "love", "dogs"

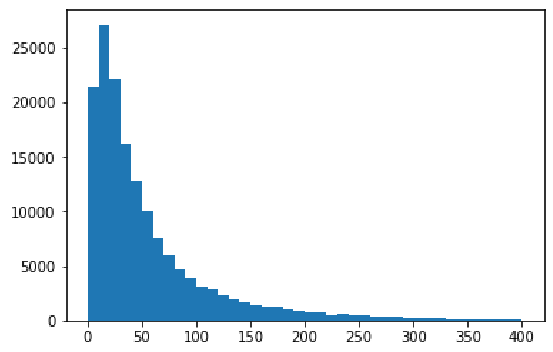


Figure 2: Length distribution of sentences

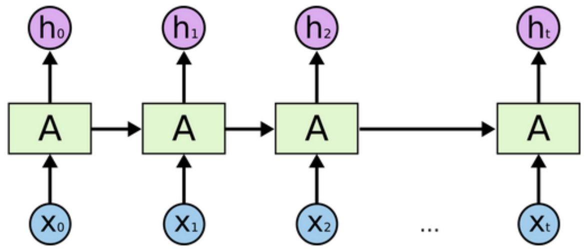


Figure 3: LSTM architecture

3. 1:”I”,2:”love”,3:”cats”,4:”and”,5:”dogs”

4. 1,2,3,4,2,5

Since we need vectors that have the same length, the next step in the project is truncating and padding the sentence vector to a fixed size. The length distribution of the sentences in the database is shown in Figure 2. As we can see, most sentences are below 150, but in the original project, they decided to use 200 just to be sure; all vectors are padded or truncated to that size. The padding is done from the left, because LSTMs remember better what they are fed at the end, so in order to preserve the information, it needs to be at the end of the vector.

The first layer of our network is an embedding layer, which projects our vocabulary into a space that depends on the distance of the surrounding words in a sentence. This way, what they achieve is reducing the model dimensions.

Right after, the output of the embedding is fed to the LSTM layer. LSTMs are a kind of network that is very used in language processing because it takes into account all inputs in context. We can see how a LSTM works in Figure 3. There are different configurations that can be done. In this project what they did was taking all the outputs in the

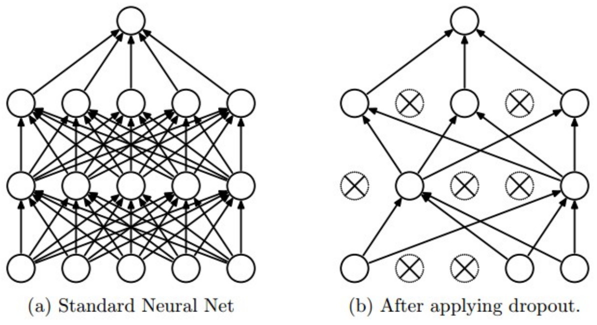


Figure 4: Dropout layer effect

iterations and then feeding the output to a Maxpooling layer so that it only has a representation per sentence.

The output, as usual, is fed to fully connected layers. In this case we have two fully connected layers, the first one with a ReLu activation function. The second one, which gives us the final output, has a sigmoid activation function and 6 outputs, as said before.

Before each fully connected layer, they introduce a dropout layer, which randomly disables 10% of the connections in the network, which leads to better generalization. The best way to understand this is looking at the picture in Figure 4.

3. Experimental setup

3.1. Tuning hyperparameters

We began by trying different values for the length of sentences (number of words/sentence) in the range of [80, 250] and concluded that the baseline length (200) gave the best results. However, most of the trainings were done with length 100 because the training time was halved. Moreover, we tuned the length of the embedded vector in the [64, 256] range and checked that the baseline length (128) was the optimum. We also modified the batch size to values between 16 and 64 and saw that the original value (32) was the best option.

A Dropout layer disables some nodes in order to force the nodes in the next layer to handle the representation of the missing data. After several experiments, the optimal configuration was to set the first dropout layer to drop out 20% of the nodes and leave the second one at 10%.

Lastly, we tested several values for the learning rate. The result was that a higher value of this hyperparameter (0.0025 instead of 0.001) improved the initial results on the original model.

3.2. Early Stopping

Early stopping is a type of regularization to limit overfitting of the training data. It stops training once the performance on the validation dataset starts to degrade by, in our case, monitoring the validation loss. This regularization can save a lot of time and may allow us to use more elaborate resampling methods to evaluate the performance of our model.

The configuration of the early stopping function we used is the following: maximum number of epochs of 20 and patience of 2 epochs.

3.3. Different architectures

Several modifications done to the baseline model were tested:

- Add Batch Normalization layer after LSTM. By adding BN, the internal covariate shift in the network is reduced, leading to the use of higher learning rates.
- Bidirectional LSTM. It runs the inputs in two ways: from past to future and from future to past. It preserves information from both past and future and can better understand the context.
- 1 LSTM without the MaxPooling1D layer. The LSTM is modified so that only the output of the last recursion is passed to the next layer instead of the outputs of each recursion. Therefore, we no longer need the MaxPooling layer and so we omit it.
- 2 LSTM without the MaxPooling1D layer. The original LSTM is fed into a second LSTM where only the output of the last recursion is passed to the next layer. The logic of the previous point is used to delete the MaxPooling layer.

All the above architectures obtain more or less the same performance. The one that gives better results is the 1 LSTM without the MaxPooling1D layer.

3.4. Initialization with pre-trained word embeddings

The main objective of using pre-trained word embeddings is to take advantage of other existing work and to boost the accuracy of the network.

We used 100 Dimensional GloVe (Global Vectors for Word Representation) embeddings of 400k words computed on English Wikipedia. It is a well known embedding technique based on factorizing a matrix of word co-occurrence statistics.

The approach used to implement this pre-trained embeddings to our model was the following:

1. Prepare an embedding matrix that maps our vocabulary words into the pre-trained embedding vector space.
2. Load this matrix into the embedding layer and set to be frozen. Therefore, the embedding vectors will not be updated during training.

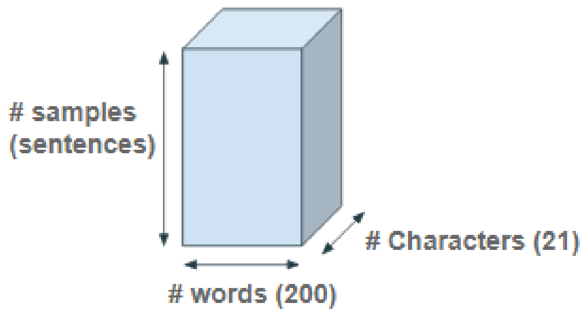
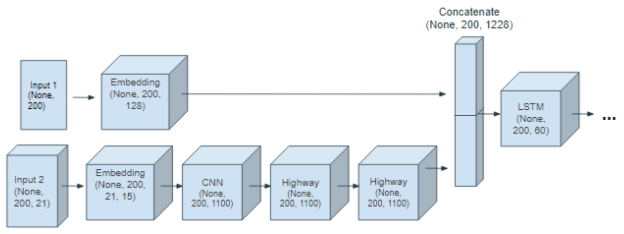


Figure 5: New data input.



Concatenate (None, 200, 1228)
Input 1 (None, 200)
Embedding (None, 200, 128)
LSTM (None, 200, 60)
...
Input 2 (None, 200, 21)
Embedding (None, 200, 21, 15)
CNN (None, 200, 1100)
Highway (None, 200, 1100)
Highway (None, 200, 1100)

Figure 6: New input architecture.

3.5. Combining multiple inputs

In order to combine multiple inputs, we first need to create a new data input based on characters instead of words. Then, tokenization and indexing is computed to those characters. Padding needs to be done to each word by adding zeros at the end of the word to be able to feed the result in the Embedding layer. We can see the dimensions of the input in Figure 5

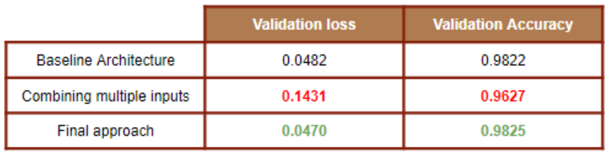
Moreover, both input tensors (the character- and word-based) need to be processed separately.

Finally, we concatenate both tensors and feed them into the LSTM layer. The final architecture needed for this can be seen in Figure 6

4. Results

The configuration of our final approach is detailed: Final configuration

- Hyper-parameters tuned: Dropout layer 1 with 0.2 and Dropout layer 3 with 0.1
- Early stopping, which decides to stop at the second epoch
- 1 LSTM with the outputs of the final time step without MaxPooling1D layer
- Initialization with the pre-trained word embedding



	Validation loss	Validation Accuracy
Baseline Architecture	0.0482	0.9822
Combining multiple inputs	0.1431	0.9627
Final approach	0.0470	0.9825

Figure 7: Results comparison.

The results are summed up in Figure 7. The best results were obtained using our final approach, where the validation loss is 0.047 and validation accuracy is 0.9825. As we can observe, these results are slightly better than the ones on the original model. On the other hand, combining multiple inputs gave the worst results, where the validation loss and accuracy, 0.1431 and 0.9627 respectively, are considerably lower.

5. Conclusions

This challenge allows to achieve very good results with a quite basic structure. After exploring deeply almost all possible configurations to our baseline architecture, we came up with some drawbacks. For instance, it is difficult to improve the original results as they were already very positive.

Another conclusion we obtained was that increasing the complexity of the architecture does not always ensure better results. It seems that due to bad orthography, a character approach may be better, but this is not the case.

Regarding future work, we could consider starting from another baseline architecture or constructing one of our own from zero and perhaps, including attention-based mechanisms.

6. References

- https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge
- https://www.kaggle.com/sbongo/for-beginners-tackling-toxic-using-keras
- https://github.com/jarfo/kchar
- https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html