FINAL REPORT
diCtNN: A Dictionary-Enhanced CNN Approach for Classifying Hate Speech on Twitter

Carlos Eduardo Posada
c.posada@mpp.hertie-school.org

Maximilian Kupi*
m.kupi@mpp.hertie-school.org

Michael Bodnar
m.bodnar@mpp.hertie-school.org

Nikolas Schmidt*
n.schmidt@mpp.hertie-school.org

Abstract

Hate speech is a growing concern, particularly on social media, and automated methods have so far been sub-par at reliably detecting it. Challenges lie in the evasive nature of hate speech, particularly in the ambiguity and fast evolution of natural language. In this paper, we propose a new multi-class classification approach, aiming to address this issue. We implement a Convolutional Neural Network (CNN) architecture and fuse a pretrained BERT word embedding with a second embedding layer based on a dictionary of hateful words. We classify a dataset of 110,748 tweets into three categories: hateful, abusive, and normal. By adding the dictionary-enhanced input, we are able to increase our model's predictive power and increase the F1 macro score by seven percentage points in comparison to the simple BERT embedding.

1. Introduction[1]

With the widespread use of social media, hateful content has become a concerning issue [2]. Hateful content, such as comments on Twitter and Facebook, target individuals or groups based on ethnicity, national background, gender identity, sexual orientation, class or disability [12, p. 130 ff.]. Studies have shown that social media can serve as a propagation mechanism, and suggested that hateful online messages are predictors of real-world hate crimes [23]. In order to discourage violence, social networks and other online venues have sought to remove these harmful messages from their platforms. However, their manual removal is unfeasible due to the high volume of comments in a growing number of social media venues. Thus, most recently, automatic methods have been employed.

Detecting, classifying and removing hate speech is a complex task from a technical point of view. Some of the difficulties include [17]:

- The lack of a common definition of hate speech, which leads to different classification criteria as well as diverging databases of hateful comments. Hate speech detection tools which are trained on these databases can thus carry biases.
- The nuances of natural language, such as sarcasm, sentiment and double meanings, increase the difficulty of distinguishing between hate speech and innocuous expressions.
- Keeping up with the lexical evolution of hate speech leads to a need to constantly update the dictionaries.

Given these difficulties, the use of deep learning methods has been proposed as a novel approach to hate speech detection. However, the recently developed approaches come with certain new limitations, such as the use of user data —besides the actual text of the tweet or comment— to make the predictions, limiting the method's applicability across platforms [20]; and the difficulty to accurately discern between tweets or comments that are merely offensive and actual hate speech. To tackle these limitations, we developed diCtNN, a deep-learning classifier of hate speech in tweets. We utilise a Convolutional Neural Network (CNN) for a three-class classification problem and categorise tweets into *hateful*, *abusive*, and *normal* tweets.

In our baseline model, the input to the CNN is a tensor of tweets, which have been cleaned and then vectorised using a pretrained BERT model. The key feature in our enhanced, final model is a dictionary approach, in which we assign a value of hatefulness to every word in each tweet, based on an established dictionary of hateful terms. The tweet-

*These authors are sharing the project between the NLP and Python class.

[1]If not further specified, all code for the model, as well as for the grid search, training and testing, was written by the authors. The code can be accessed from this GitHub repository (clickable link) and the documentation can be accessed from readthedocs.io (clickable link)

level vector of hatefulness values is then transformed into a tensor and stacked with the tensor resulting from the BERT vectorisation of each tweet, resulting in a two-dimensional input for our CNN.

Our testing results show that this enhancement improves the model's predictive power as it helps to better distinguish between *hateful*, *abusive*, and *normal* tweets.

The next section will highlight important related work in the field of hate speech detection. Section 3 will outline the methods and theory employed in the development and evaluation of our diCtNN, while the section following thereafter will detail the experiments carried out and their results. Finally, the analysis section will provide a qualitative evaluation of our work. The main part of the paper will end with a conclusion.

2. Related Work

The literature on hate speech detection has evolved over the past years regarding its definition, its context of occurrence, and the applied computational methods. Initially, surface-feature extractions of the text (e.g. token and character n-grams) were combined with other methods (e.g. bag of words, word generalisation or term-frequency-inverse document frequency) for classification [33]. A variety of machine learning algorithms have been employed in past literature [25], for example support vector machines [11]. However, in recent years, neural networks have become the state of the art, in particular in combination with the use of different word-embedding techniques to represent and group text data in a vector space [14, 15, 26].

Research has also explored the utilisation of further characteristics of hate speech: sentiment analysis can work as a sub-approach to hate speech detection, focusing on strong negative sentiments to differentiate normal from hate speech [31, 33, 37]. However, this has not improved results substantially, in particular when considering the controversy regarding legally-binding definitions of hate speech: strong negative sentiment alone does not yet constitute hate speech.

On the other hand, approaches to identify hate speech based on the presence of offensive language (rather than sentiment) have proven to have limited accuracy as well, due to the possibly innocuous use of certain offensive words [3]. One way of addressing this is to diversify the classification problem, e.g. by including a category for *offensive / abusive* content, next to *hateful* and *normal* content. This gives both a differentiated orientation for manual tagging as well as more nuanced basis for deep-learning classification [7, 8].

Another way of identifying hate speech in online content is to include user data and other metadata into the analysis [19, 30, 36], with some authors even going so far as to rely solely on that type of data [21]. In these approaches, the focus lies more on identifying hateful user accounts and environments, rather than hateful content per se. This is mostly useful in specific circumstances of real-time online interactions, where deleting chat content retroactively misses the point. An example for this are online-gaming chats, where banning users is the routine practice [34]. However, metadata collection differs across platforms and thus impedes a model's applicability to other datasets or platform environments [20].

An alternative or additional approach is to make use of a dictionary or lexicon of hateful terms to detect hate speech. This can be done by building a dictionary of terms or phrases from a manually tagged dataset [9, 22] or by matching content against an external dictionary. However, so far dictionaries have mostly been utilised in rule-based machine classification [18] — for example, in order to select potentially hateful content (such as tweets) from a broader database of content or to assess the classification quality of crowd-sourced manual tagging [3, 22]. To our knowledge, our approach of fusing a pretrained word embedding with a second embedding layer based on a dictionary of hateful words as input into a deep-learning architecture is a novelty in the literature.[2]

3. Proposed Method

To develop diCtNN, we followed a two-staged path: first, we established a CNN model architecture to solve the three-way classification problem; second, we implemented our dictionary approach in order to enhance the input into this CNN architecture.

Choice of model: CNN architectures typically excel at computer vision as well as image and video processing, but have also shown promising results in natural language processing (NLP) and speech recognition [13]. In NLP, other architectures such as Recurrent Neural Networks (RNNs) have also been used, including Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) RNN architectures [1]. However, in detecting hate speech, CNNs have been shown to offer distinctive advantages.

Namely, CNNs are adept at extracting features and recognising patterns from word combinations.[3] This specificity of CNNs is relevant for hate speech, where certain combinations of words occur often, while the sequence in which certain words occur is less important. Additionally, tweets are short[4], which makes the capability to detect sequences in a single tweet, as is the approach of RNN alternatives, less useful for our purposes. This is especially

[2] In addition, none of the previous approaches has taken into account an ambiguity measure or offensiveness rating of hateful terms (see Section 3).

[3] In contrast to this, RNNs rely on features found in sequences.

[4] 140 characters in the data we are using; Twitter changed the maximum character length of a tweet to 280 characters in 2017.

relevant when hate speech is determined by the occurrence of specific words[5], whereas modelling a longer sequence —such as done with GRU— can miss key words that determine the meaning within a sequence [38].

Furthermore, CNN architectures have been shown to be more efficient than RNN architectures, as RNNs take longer or more computing power to train [1].[6] For these reasons, we have chosen a CNN model architecture to distinguish between *hateful*, *abusive*, and *normal* tweets. We implemented that architecture in Python, using the PyTorch library [27].

Convolutional Neural Network: CNNs are a special kind of multi-layer perceptron that include additional layers called convolutional layers, in which the process of convolution occurs. In this process, a filter, which is a small matrix of weights or parameters, also called a kernel, is applied to the input data in matrix form in order to capture local correlations between points (e. g. spatial and temporal dependencies) [13], and at the same time reduce processing complexity and time [32]. In our baseline model, the input matrix and the kernel are matrices of height 1, i.e. vectors. For the implementation of our dictionary approach, however, we use 2-dimensional matrices.

During convolution, the kernel strides over the input matrix. While striding, it is matrix-multiplied with a new section of the input matrix. The results of the multiplication are summed up and after the kernel has stridden over the whole input matrix, a matrix of weighted sums is obtained. This output has reduced complexity over the input dataset and conserves the most essential features, defined by the kernel. The first layers of convolution extract low-level features of the input data, while later layers capture high-level features [32].

Padding is another characteristic of the convolution process. It consists of adding extra zero values to the input matrix, so that the kernel can cover its edges equally while striding. For our model, we performed padding before each convolution so that the convolved matrices had the same size as the input matrices and their dimensions were maintained throughout the network [32]. Besides padding, other relevant hyperparamenters have been set as follows, based on the usual conventions for text classification: the size of the kernel, which defines the shape of the convolution, has been set to three; and the stride, which controls the size of the steps the kernel takes while displacing over the input matrix, has been set to one [29].

After the convolutions, the final output matrices are flattened into a vector and fed into a multi-layer perceptron, which receives the extracted features and calculates the values for final classification [24]. The loss is computed by comparing the predicted and the true labels, in our case, using a multi-class cross-entropy loss function. Finally, the weights of the kernel are adapted based on the loss through backpropagation. This is how the network "learns" [13].

Enhancing vectorised input with a second dimension: As the key distinguishing feature in our model we enhance the vectorised input to the CNN with data derived from the Hatebase dictionary, an established dictionary of hateful terms.[7] This addresses the common challenge of distinguishing hate speech from abusive or normal tweets.

This method is then integrated into the vectorisation process of the cleaned tweets, which is implemented with a pretrained model of BERT [4]. BERT models are bidirectionally trained, so they achieve a better sense of language context than single-directional models. For efficiency and reliability reasons, we employ the BERT-base-uncased model [28]. The maximum vector length is set to 120 and padded with zeros. Both processes are applied to every tweet in our dataset and the resulting tensors are stacked. In order to stack the tensors, the tensor of the hatefulness values is stretched to the length of the tensor resulting from the BERT vectorisation (see next paragraph). The resulting matrix is then used as an input for the 2D-architecture of our model, as opposed to the 1D-architecture which we used in our baseline model, where the input consists only of the BERT-vectorised word embedding.

The additional input of the dictionary is designed to more clearly classify tweets in which unambiguously hateful terms are used. This innovative element distinguishes our architecture from other approaches, detailed in Section 2. Furthermore, our approach relies solely on text feature analysis and uses no metadata, user-specific or otherwise, thus classifying only content, not users, as hateful.

Convolution - how the model learns based on both inputs: The following explanation is based on the master documentation of PyTorch [27].

To understand the way we train our model to make it able to predict the hatefulness of tweets, we have to take a closer look at the mathematical concepts used, and follow one tweet through preprocessing and training. As discussed above, we use two different approaches and combine them in our model, as can be seen in Figure 1. While the BERT vectoriser produces a vector of a length that depends on the tokenisation of BERT, our dictionary approach uses another method of tokenisation to identify full nouns

[5]This approach in understanding hate speech underpins also our further investigation methods, see remainder of this section.

[6]While this may not be an issue in our application, as our data input is limited, striving for efficiency in use of computational power should be a matter of principle.

[7]Davidson et al. [3], whose dataset makes up the smaller part of our merged dataset (20%), have also made use of the Hatebase dictionary to preselect tweets for manual labelling.

without lemmatisation and other changes.[8] This produces two vectors, which can be of different lengths. To put the value that results from the hate dictionary analysis into the vicinity of the actual BERT-encoded word, we stretch the dictionary-derived vector with linear interpolation. This way, the length of both vectors match, and the hate dictionary evaluation will be close to its BERT embedding.

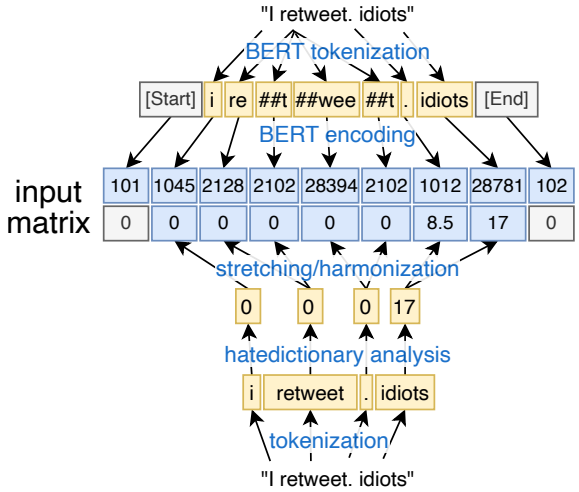

Figure 1. Preprocessing of tweets and combination of BERT vectorisation with hate dictionary approach (own visualisation)

To harmonise the input of our CNN over all the tweets, we pad the resulting matrix with zeros at the end until the matrix is of length 120 (see Figure 2).

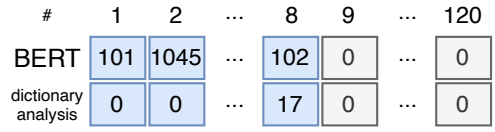

Figure 2. Padding of the preprocessed matrix to harmonise the data set for the CNN (own visualisation)

As discussed above, a convolution is used to combine multiple input values with different weights attached to them to an output value. We call the object that contains all the weights used in the calculation a kernel, which is defined through the weight matrix $[w_{ij}] = W \in \mathbb{R}^{3\times3}$.

The first layer gets a 2D matrix $X \in \mathbb{R}^{2\times120}$ of length 120 and height 2 as input. Based on the settings for kernel size, stride, and dilation the kernel is applied to a sub-matrix $\tilde{X} = [x_{ij}]$ of input matrix X.

[8] We ran different models with and without lemmatisation of the terms in the Hatebase dictionary and found that lemmatisation did not add to the predictive power, as many terms in the dictionary were not recognised and therefore not lemmatised in the first place.

We can define the application of the kernel to one submatrix mathematically as a function that takes the submatrix $\tilde{X}$ and a matrix of weights W to calculate a weighted sum, i.e. the convolution:

$$f : \mathbb{R}^{3\times3} \times \mathbb{R}^{3\times3} \to \mathbb{R}, \quad (\tilde{X}, W) \mapsto \sum_{i,j=1}^{3} w_{ij}x_{ij} =: y_*$$

The actual learning of the model is based on the weights that we train. A weight matrix is convoluted at every layer with sub-matrices of the input matrix to create a new output matrix. The matrix is padded again with zeros to ensure that the convolution with the matrix is not reducing the overall size of the matrix that is moving through the neural network. In Figure 3, a matrix with height 2 is padded with one row of zeros to allow a kernel of size 3 to reduce the matrix to an output of height 2 again.

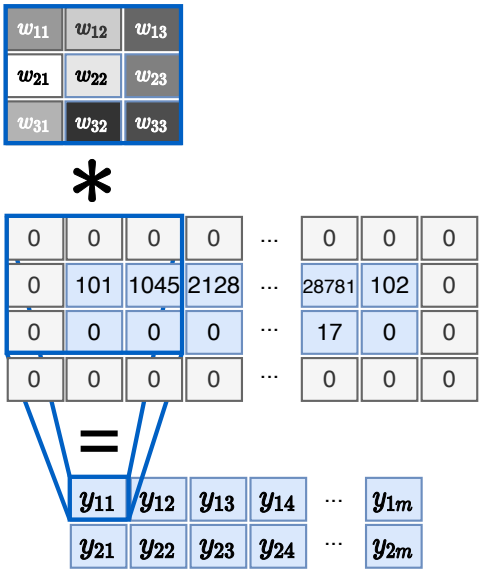

Figure 3. Convolution of weight matrix $W = [w_{ij}]$ with submatrix of padded input matrix to produce output matrix $Y = [y_{ij}]$ (own visualisation)

Each neuron in our network comes with its own weight matrix. The initial weights for these weight matrices W are generated randomly with a uniform distribution of values. All neurons in the first layer get the same input vector but will produce different convolutions due to the randomness of the weights. If all weights would be equal then the back-propagation algorithm that tweaks the weights after each training step would have to change all of them or choose randomly which one to adjust because all would have the same effect. Randomisation of the weights takes this decision from the algorithm by "breaking symmetry" in the network [10, p. 297]. Thus, the randomisation is the precondition for the network to learn.

PyTorch calculates the range of values from which it can draw from randomly by considering the amount of input values and the kernel size. The possible weights must be in the range $(-\sqrt{k}, +\sqrt{k})$, where k is defined through the kernel size and the input channels.

If we look at all the convolutions happening in order to produce the output of one neuron, we can see that an individual input value is included in multiple weighted sums. The choice of k ensures that neurons with different kernel size and/or input channel amount produce convolutions that are in the same numerical range. This happens through a kind of normalisation in the definition of k. Each neuron in the neural network has its own weights and draws them randomly from a uniform random distribution, e.g. $w_{11}, w_{12}, w_{21} \in \mathcal{U}(-\sqrt{k}, +\sqrt{k})$ like in Figure 4.

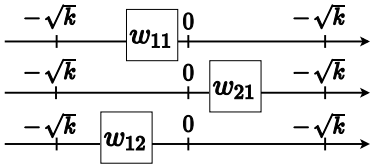

Figure 4. Random initialisation of kernel weights (own visualisation)

In each neuron the convolution is repeated for all submatrices of the vector (or only a part of it depending on kernel size, stride and dilation). This results in multiple matrix multiplication per neuron which applies the kernel to each sub-vector and outputs one matrix with the result of all the associated convolutions (see Figure 3).

Finally a bias B is added to the convolution. The initial bias B is drawn randomly from the same uniform distribution as before: $\mathcal{U}(-\sqrt{k}, +\sqrt{k})$.

While the weights W determine the focus of the calculations, the bias B directly influences the result of the activation function of the neuron. If the activation function is a Sigmoid function (see Figure 5) then e.g. a simple shift to the left (negative b) could reduce the whole neuron output to zero and thus make the information covered in the convolutions irrelevant for the network. The bias puts some neurons' activation threshold lower and some higher in the beginning, and allows the network to focus on the relevant information. As was mentioned above in relation to the weights, the initial randomisation of the bias allows the network to learn what to focus on.

The weights and the bias are adjusted depending on the loss after each training step through the back-propagation algorithm. In conclusion, the stacking of the two vectors in conjunction with the here described mathematical operations of a CNN network will ensure that our model draws on both – the BERT-vectorised as well as the dictionary vectorised – inputs to learn.

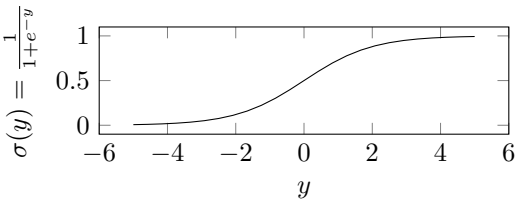

Figure 5. Sigmoid function (own visualisation)

Data preprocessing: The tweets were preprocessed in order to remove noise before the vectorisation process [16]. The text was transformed to lower case, and symbols[9], whitespaces and standalone digits were removed. Tweet-specific traits, i.e. URLs, mentions, reserved words, emojis and smileys were also removed.[10] The content of hashtags (i.e. "idiots" in the case of "#idiots") was preserved.

Model architecture: We use a CNN architecture with four hidden layers (three convolutional and one final linear layer) as well as a ReLU activation function, and batch normalisation after each convolution.[11] The initial number of channels is set to one, as we input one matrix per tweet. The output channels of the first, second, and third CNN layer have been set to 16, 32, and 64, respectively. As discussed above, we apply a kernel of size three with one row of zero padding and stride of one to preserve the feature dimensions throughout the network [6]. The out features of the final linear layer have been set to three in order to match the three classes of the tweet annotation. For a visualisation of the model architecture see Figure 6 on the next page.

[9] As BERT also vectorises punctuation, we employed one cleaning version where standard sentence punctuation (!?.,) was preserved, and tested this version against the one with full symbol removal (see Section 4).

[10] Using the tweet-preprocessor program from Said Özcan: `https://pypi.org/project/tweet-preprocessor/`.

[11] Modelled after a CNN template from: `https://adventuresinmachinelearning.com/convolutional-neural-networks-tutorial-in-pytorch/`

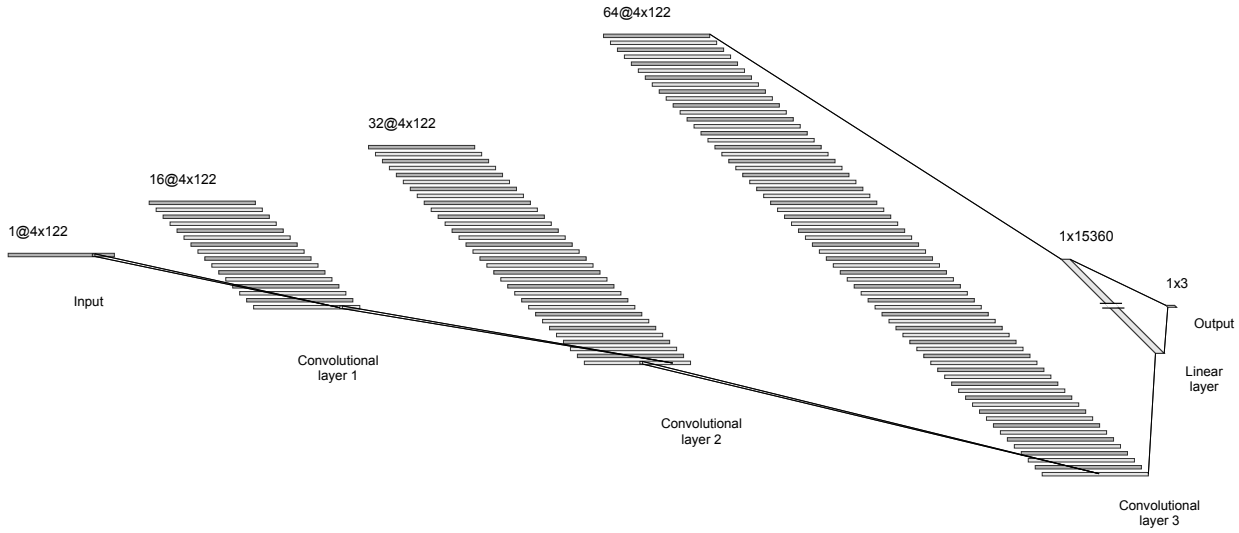

Figure 6. The employed CNN model architecture (including padding - own visualisation)

4. Experiments

Data: We use the following two datasets for our classification task:

- The dataset from Davidson et al. [3], which is composed of 25,000 tweets that have been preselected using the crowd-sourced *Hatebase* hate speech keyword lexicon (see Section 3) and manually labeled using the crowd-sourcing platform CrowdFlower
- The dataset from Founta et al. [8], which comprises 100,000 manually annotated tweets (again using CrowdFlower) whose annotation has been checked for statistical robustness by the authors

The datasets have the following labelling categories:

- Davidson et al.: *hate_speech*, *offensive_language*, *neither*
- Founta et al.: *hateful*, *abusive*, *normal*, *spam*

Since our focus does not lie on spam detection, we decided to delete all tweets in this category from the second dataset. Next, following the definitions of Founta et al. [8], according to which offensive and abusive language are two very similar categories, we decided to merge both datasets. The benefits of doing so are a more balanced dataset[12] as well as an increased sample size for the *hate* class, which was highly underrepresented in both datasets. Since the resulting dataset, composed of 110,748 tweets, was still rather unbalanced with respect to the *hate* category (see Figure 7), we used a stratified shuffle split, to finally divide our dataset into training (70%), validation (15%), and test set (15%), while ensuring that the classes would be distributed equally amongst all splits.

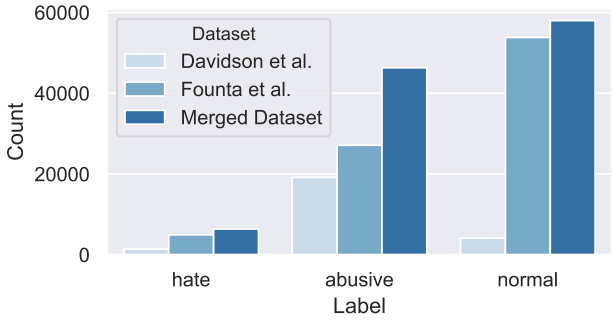

Figure 7. Counts for the respective classes in the datasets

Software: For coding and local testing we used Microsoft's Visual Studio Code, and employed GitHub to share and document the code. The project was coded and deployed in the programming language Python (Version 3.7) [35].

Hardware: To train our models on GPUs we used Google Colab. Furthermore, we executed computing intensive tasks, e.g. the vectorisation of the tweets, on a remote EC2 instance running Jupyter Lab on Amazon Web Services (AWS) through AWS SageMaker.

Evaluation method: As a first metric to evaluate whether the model was training correctly, we observed the evolution

[12]Both original datasets had a significant class imbalance between all three classes, while the resulting dataset is more balanced – at least between the *normal*, and *abusive* class (see Figure 7).

of the accuracy on the training and validation set over the training epochs. Furthermore, the decrease of the training and validation loss over the epochs was used as an indication for the model's convergence and generalisability. However, since accuracy tends to be highly misleading due to the class imbalance of the dataset, we further paid particular attention to the F1 macro score. Finally, in order to visualise and evaluate precision and recall of the three classes in an intuitive manner, we used confusion matrix plots.

Experimental details: To determine the best hyperparameters for our CNN model architecture, we made use of a grid search. We specified the following variables for the grid search:

- optimizer type (Adam, RMSprop, and SGD)
- learning rate (0.0001, 0.001, 0.01)
- whether to use a sampler in the data loader or class weights in the loss function to balance the dataset during training[13]
- whether or not to use a scheduler to decrease the learning rate (factor 0.1) based on the performance of the F1 macro score

During grid search we trained each model in the simple 1D version (without our diCtNN preprocessing approach) for 45 epochs with a batch size of 16. We then considered the results from the best epoch with respect to the F1 macro score. Based on the results from the grid search, the best hyperparameters were selected and the simple 1D version as well as the 2D version with diCtNN preprocessing was trained for 90 epochs respectively. Furthermore, we also trained the models on a second dataset where standard sentence punctuation has not been removed.[14] Figure 8 shows the F1 macro score plots for the 1D and 2D model on the dataset including standard sentence punctuation.[15] As can be seen, the 2D model has a steeper and higher learning curve; however it also starts overfitting at around forty epochs. After this final training routine, the best performing 1D and 2D models in terms of F1 macro score have been tested on the testing set, to compare the two approaches and ensure their generalisability.

Results: The best model from the grid search turned out to have the following specifications: Adam optimizer, 0.01 learning rate, class weights in the loss function, and a deactivated scheduler. After retraining the 1D and 2D model for 90 epochs, the trainings on the dataset which still contained sentence punctuation outperformed the ones with full symbol removal. For the 1D model the highest score was yielded in epoch 29, for the diCtNN 2D model in epoch 36

[13] In case the class weights were used, the shuffle mode in the data loader was automatically set to true in order to ensure more robust training.

[14] i.e. ".", ",", "?", "!" .

[15] The loss and accuracy plots can be found in the Appendix.

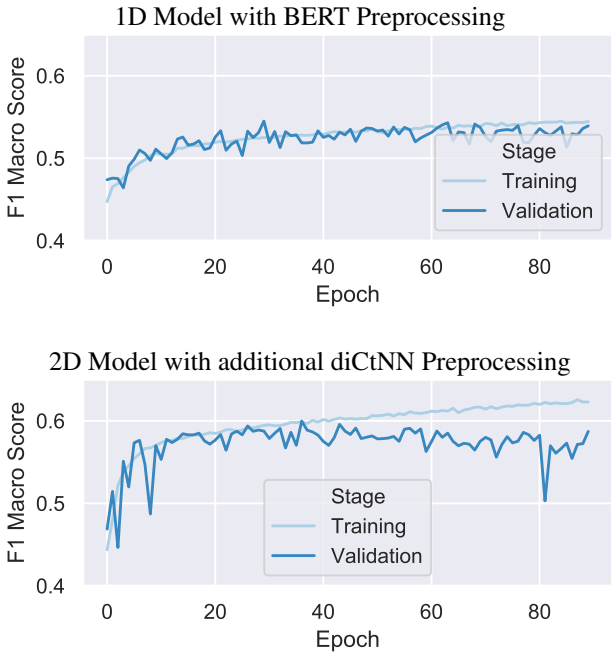

Figure 8. F1 macro score plot of the 1D and 2D model over 90 epochs

respectively. Testing these models on the testing set yielded the following results (see Table 1).

1D Model with BERT Preprocessing

	Precision	Recall	F1 Score
hateful	0.21	0.26	0.23
abusive	0.69	0.62	0.66
normal	0.73	0.76	0.74
Accuracy			0.68
Macro Avg	0.54	0.55	0.54
Micro Avg	0.68	0.68	0.68

2D Model with additional diCtNN Preprocessing

	Precision	Recall	F1 Score
hateful	0.31	0.30	0.30
abusive	0.76	0.68	0.72
normal	0.77	0.83	0.80
Accuracy			0.74
Macro Avg	0.61	0.61	0.61
Micro Avg	0.74	0.74	0.74

Table 1. Classification report of the best 1D model and 2D model

As can be seen, our diCtNN outperformed the simple 1D model on all levels. Yet, in comparison to Davidson et al. [3], who trained classifiers on the smaller subset of our data and yielded an overall precision, recall and F1 score of above 0.90 with their best models, our performance metrics

seem rather low.[16] However, since Davidson et al. did not apply a deep neural network architecture for classification – instead they used linear support vector machines and logistic regression – and framed the problem as a one-versus-rest classification, which significantly reduced the problem of class imbalance, these results are hardly comparable.

5. Analysis

In order to check our basic assumption that *hateful* tweets contain more hateful words and hence mapping them on a second dimension would potentially improve the model performance, we calculated the average hatefulness of each tweet per label based on the respective tensor entries of the second dimension (see Table 2). As can be seen in the table, the average hate scores are distributed as expected.

	hateful	*abusive*	*normal*
Average Hate Score	126.28	89.91	56.26

Table 2. Average total hate scores of tweets per label

Comparing the two models' performance on the test set based on the confusion matrix (see Figure 9) shows that our approach outperforms the simple 1D BERT model: All percentages on the top left to bottom right diagonal increase, while all other percentages decrease. Hence, precision and recall for distinguishing between *hateful*, *abusive* and *normal* speech on twitter can be increased by applying our innovative diCtNN approach.

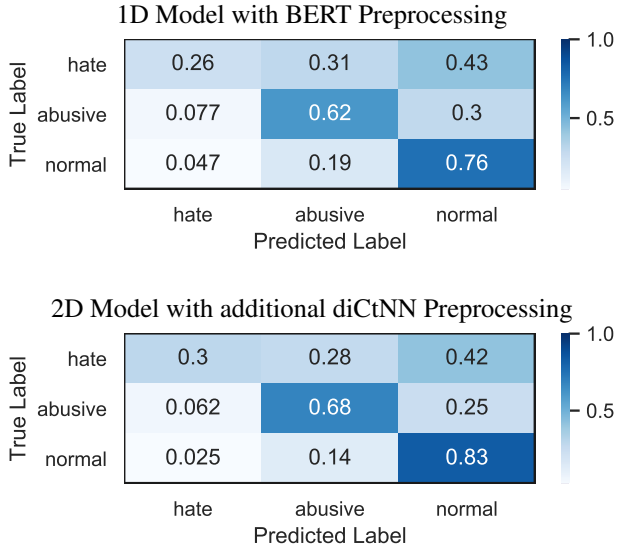

Figure 9. Confusion matrix of the 1D and the 2D model

However, as can be seen by the relatively high percentage in the top right of the matrix, both models still have problems with wrongly classifying hate speech as normal speech. Generally, this can partly be due to the specifically high class imbalance between those two classes: Since *hateful* is the smallest class, it is likely, just by random chance, that any algorithm mistakenly classifies a tweet from this class to belong to the biggest class, which is the *normal* class. However, taking a closer look at the *hateful* tweets which our diCtNN approach wrongly classified as *normal* revealed four further potential explanations: First, some tweets are labelled as *hateful*, but contain no usual *hateful* words, so our algorithm did not catch them. Example:

> *"@user1 @user2 @realDonaldTrump The only solution to not having children gassed to death is to bring down entire Syrian govt"*[17]

Second, some normal tweets have actually been wrongly labelled as *hateful*, as they, for example, only report about hate. Exemplary tweet:

> "*I just watched a video with a crowd of white ppl shouting nigga & going crazy to songs about black men killing each other & it made me so sad*".[18]

Third, our dictionary-preprocessing algorithm does not catch all the *hateful* words in some tweets. This might be due to the fact, that in the implemented version it does not match bi-grams, or the word simply is not contained in the *Hatebase* dictionary – as this generally only includes nouns. Example tweet that fulfils both these conditions:

> "*#AmericaIsOverParty I TOLD YALL CAUCASIAN, COUSIN FUCKING, INCEST FUCKS TO NOT VOTE FOR TRUMP NOW LOOK WHAT YALL DID*"

Fourth, and finally, the definition of hate speech, and which tweets fall under this definition, is still an ongoing debate [5], and sometimes also a matter of political perspective or even potentially diverging realities as can be seen in the first example tweet.

[16] Since they framed their problem as a one-versus-rest problem, these score are best compared to the micro F1 score.

[17] Any occurrence of abusive or hateful content in this paper is for exemplary purpose only and not meant to offend anyone.

[18] Interesting to note about this specific case is that our algorithm classified it "correctly in real terms" as normal. This might be the case because the hatefulness in the second dimension of our input matrix functions as a magnifying for the algorithm to know which words in the first dimension it should look at. And in this specific case the hate terms identified by the algorithm ('*crowd*', '*white*', & '*nigga*') are rather regularly used 'hate' words that are not particularly distinct for hate tweets, so the algorithm classified the tweet as normal. Of course, this rather speculative explanation would have to be further investigated using explainable AI approaches that actually show where the CNN is looking.

6. Conclusion

We have shown that enriching BERT vectorisation with content-based embedding via a dictionary approach can substantially improve a CNN model's performance in distinguishing between *hateful*, *abusive*, and *normal* tweets.[19] Future research might further improve our approach (e.g. implementing bi-gram matching of hateful terms or testing other neural network model architectures such as LSTMs or GRUs), and apply it to other datasets or compare it to state-of-the-art results of other approaches. With adaptation, this approach could also be applied to other text classification problems, provided that a widely accepted dictionary of terms for the specific domain exists.[20]

7. Acknowledgements

The authors would like to express their gratitude to Antigoni M. Founta of [8] for granting access to the complete dataset of their study.

8. Contributions

E. Posada was responsible for doing the cleaning of tweets, and coding the testing method of the model. He also provided the visualisation of the architecture and part of the code documentation.

M. Kupi was responsible for dealing with the datasets as well as for setting up the grid search and the tracking of relevant metrics during training and testing. He also oversaw the training and testing routine, visualised and analysed the model results, and did most part of the code documentation.

M. Bodnar coded the class to setup the neural network as well as the methods for the vectorisation of tweets in 1D and 2D. He also provided visualisations for mathematical aspects of the network.

N. Schmidt was responsible for developing, setting up and implementing the dictionary approach (in multiple versions) as well as the testing of (mainly preprocessing) functions. He also contributed to the vectorisation and cleaning functions.

9. Sharing project

M. Kupi and N. Schmidt share the project between the NLP and Python class.

[19] If deployed for real-time hate speech detection on Twitter, the dictionary from *Hatebase* would need to be vetted for consistency with the platform guidelines and regularly updated.

[20] We have coded our preprocessing functions with maximum flexibility for different specifications so as to ease such further application.

References

[1] H. Adel and H. Schütze. Exploring different dimensions of attention for uncertainty detection. *15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017 - Proceedings of Conference*, 1:22–34, 2017.

[2] M. Costello, R. Barrett-Fox, C. Bernatzky, J. Hawdon, and K. Mendes. Predictors of Viewing Online Extremism Among America's Youth. *Youth & Society*, page 0044118X18768115, 4 2018.

[3] T. Davidson, D. Warmsley, M. Macy, and I. Weber. Automated Hate Speech Detection and the Problem of Offensive Language. In *Proceedings of the 11th International AAAI Conference on Web and Social Media*, ICWSM '17, pages 512–515, 2017.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Google AI Language*, 2018.

[5] W. Dorris, R. R. Hu, N. Vishwamitra, F. Luo, and M. Costello. Towards automatic detection and explanation of hate speech and offensive language. In *IWSPA 2020 - Proceedings of the 6th International Workshop on Security and Privacy Analytics*, pages 23–29, 2020.

[6] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *ArXiv*, abs/1603.0, 2016.

[7] P. Fortuna, J. Rocha da Silva, J. Soler-Company, L. Wanner, and S. Nunes. A Hierarchically-Labeled Portuguese Hate Speech Dataset. In *Proceedings of the Third Workshop on Abusive Language Online*, pages 94–104, Florence, Italy, 2019. Association for Computational Linguistics.

[8] A.-M. Founta, C. Djouvas, D. Chatzakou, I. Leontiadis, J. Blackburn, G. Stringhini, A. Vakali, M. Sirivianos, and N. Kourtellis. Large Scale Crowdsourcing and Characterization of Twitter Abusive Behavior. Technical report, AAAI Press, 2018.

[9] N. D. Gitari, Z. Zuping, H. Damien, and J. Long. A lexicon-based approach for hate speech detection. *International Journal of Multimedia and Ubiquitous Engineering*, 10(4):215–230, 2015.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[11] M. O. Ibrohim and I. Budi. Multi-label Hate Speech and Abusive Language Detection in Indonesian Twitter. In *Proceedings of the Third Workshop on Abusive Language Online*, pages 46–57, Florence, Italy, 2019. Association for Computational Linguistics.

[12] J. B. Jacobs and K. Potter. *Hate Crimes : Criminal Law and Identity Politics*. Studies in Crime and Public Policy. Oxford University Press, Oxford, 2001.

[13] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, pages 1–62, 4 2020.

[14] Y. Kim. Convolutional Neural Networks for Sentence Classification. *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, 8 2014.

[15] R. Kshirsagar, T. Cukuvac, K. McKeown, and S. McGregor. Predictive Embeddings for Hate Speech Detection on Twitter. In *Proceedings of the 2nd Workshop on Abusive Language Online (ALW2)*, pages 26–32, Brussels, Belgium, 2018. Association for Computational Linguistics.

[16] T. Lata. Getting Your Text Data Ready for Your Natural Language Processing Journey, 2018.

[17] S. MacAvaney, H. R. Yao, E. Yang, K. Russell, N. Goharian, and O. Frieder. Hate speech detection: Challenges and solutions. *PLoS ONE*, 14(8):1–16, 2019.

[18] R. Martins, M. Gomes, J. J. Almeida, P. Novais, and P. Henriques. Hate Speech Classification in Social Media Using Emotional Analysis. In *2018 7th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 61–66, 2018.

[19] B. Mathew, R. Dutt, P. Goyal, and A. Mukherjee. Spread of Hate Speech in Online Social Media. *WebSci 2019 - Proceedings of the 11th ACM Conference on Web Science*, pages 173–182, 2019.

[20] J. S. Meyer and B. Gambäck. A Platform Agnostic Dual-Strand Hate Speech Detector. In *Proceedings of the Third Workshop on Abusive Language Online*, pages 146–156, Florence, Italy, 2019. Association for Computational Linguistics.

[21] F. Miró-Llinares, A. Moneva, and M. Esteve. Hate is in the air! But where? Introducing an algorithm to detect hate speech in digital microenvironments. *Crime Science*, 7(1):1–12, 2018.

[22] H. Mulki, H. Haddad, C. Bechikh Ali, and H. Alshabani. L-{HSAB}: A {L}evantine Twitter Dataset for Hate Speech and Abusive Language. In *Proceedings of the Third Workshop on Abusive Language Online*, pages 111–118, Florence, Italy, 2019. Association for Computational Linguistics.

[23] K. Müller and C. Schwarz. Fanning the Flames of Hate: Social Media and Hate Crime. *SSRN Electronic Journal*, 2019.

[24] R. Newatia. How to implement CNN for NLP tasks like Sentence Classification, 2019.

[25] C. Nobata, J. Tetreault, A. Thomas, Y. Mehdad, and Y. Chang. Abusive language detection in online user content. In *25th International World Wide Web Conference, WWW 2016*, pages 145–153, New York, New York, USA, 2016. International World Wide Web Conferences Steering Committee.

[26] R. Orac. Identifying Hate Speech with BERT and CNN, 2019.

[27] PyTorch. TORCH.NN, 2019.

[28] T. Rajapakse. To Distil or Not To Distil: BERT, RoBERTa, and XLNet, 2020.

[29] D. Rao and B. McMahan. Chapter 4: Feed-Forward Networks for Natural Language Processing. In *Natural Language Processing with PyTorch*, chapter 4. O'Reilly Media, 1 edition, 2019.

[30] M. H. Ribeiro, P. H. Calais, Y. A. Santos, V. A. Almeida, and W. Meira. Characterizing and detecting hateful users on twitter. In *12th International AAAI Conference on Web and Social Media, ICWSM 2018*, pages 676–679, 2018.

[31] A. Rodríguez, C. Argueta, and Y. Chen. Automatic Detection of Hate Speech on Facebook Using Sentiment and Emotion Analysis. In *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pages 169–174, 2019.

[32] S. Saha. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way, 2018.

[33] A. Schmidt and M. Wiegand. A Survey on Hate Speech Detection using Natural Language Processing. *Proceedings of the Fifth International Workshop on Natural Language Processing for Social Media*, pages 1–10, 2017.

[34] W. Stoop, F. Kunneman, A. van den Bosch, and B. Miller. Detecting harassment in real-time as conversations develop. In *Proceedings of the Third Workshop on Abusive Language Online*, pages 19–24, Florence, Italy, 2019. Association for Computational Linguistics.

[35] G. Van Rossum and F. L. Drake Jr. *Python Tutorial*. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995.

[36] Z. Waseem and D. Hovy. Hateful Symbols or Hateful People? Predictive Features for Hate Speech Detection on Twitter. In *Proceedings of the NAACL Student Research Workshop*, pages 88–93, San Diego, California, 2016. Association for Computational Linguistics.

[37] H. Watanabe, M. Bouazizi, and T. Ohtsuki. Hate Speech on Twitter: A Pragmatic Approach to Collect Hateful and Offensive Expressions and Perform Hate Speech Detection. *IEEE Access*, pages 13825–13835, 2018.

[38] W. Yin, K. Kann, and M. Yu. Comparative Study of CNN and RNN for Natural Language Processing. *arXiv e-prints*, 2017.

10. Appendix

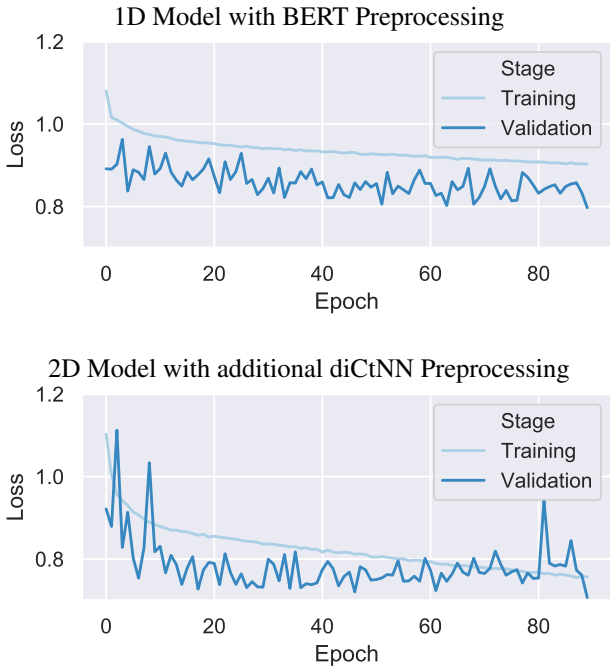

Figure 10. Loss plot of the 1D and 2D model over 90 epochs

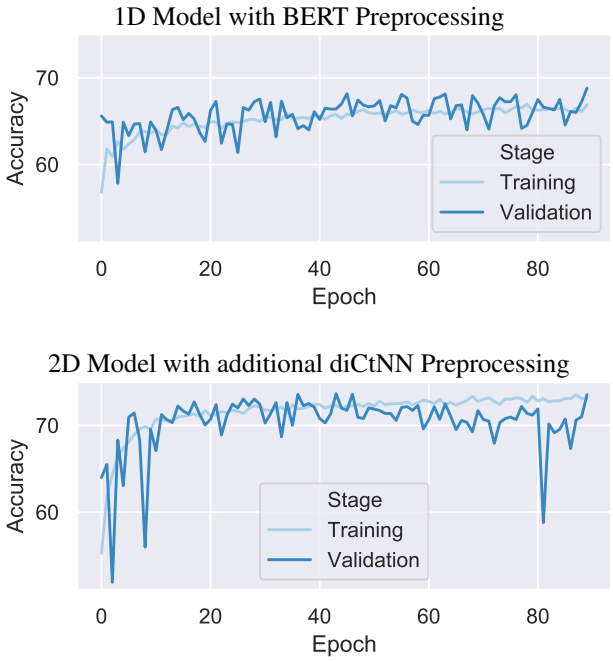

Figure 11. Accuracy plot of the 1D and 2D model over 90 epochs