Deep Detection of Hate Speech in Text Through a Two-Pronged Approach

Johannes Skjeggestad Meyer

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Björn Gambäck, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

With the widespread use of online services like Facebook and Twitter, disseminating hateful messages has become a simple matter. These messages not only spoil the experience for other users of a service. There is also an increasing legal pressure for the services to prevent and remove such hate-spreading content. For this to be practically feasible, there is a need for systems that can automatically detect hate speech in text.

Research on automatic detection of hateful and abusive language has been an ongoing project over the last 20 years. However, the state-of-the-art is still not good enough to be practically usable for identifying hate speech in a fully automatic manner. Thus, this thesis continues the efforts to reach that goal.

With the increasing legal pressure to remove hate speech, and the multitude of services and platforms this pressure applies to, detection approaches are needed that do not depend on any information specific to a given platform. This is so that the approach can be used across several different platforms without being changed. For instance, the information stored about the text's author may differ between services, and so using such data would reduce the general applicability of the system. Therefore, the research in this thesis aims at avoiding any such information, using exclusively text-based input in the detection.

This thesis proposes a novel, Deep Learning-based approach to hate speech detection, using a two-pronged architecture that combines both Convolutional Neural Networks and Long Short-Term Memory-networks. The proposed architecture uses Character N-grams and Word Embeddings as inputs to its two prongs, which then merge and produce a final classification. The experiments show that this architecture, using its optimal configurations, performs better than most state-of-the-art systems.

Sammendrag

I dagens samfunn har bruken av internettjenester som Facebook og Twitter blitt gjennomgripende, og store mengder bruker-genererte data blir daglig lagt ut. Samtidig har dette gjort at spredning av hatefulle ytringer har blitt en enkel jobb. Denne typen meldinger kan ødelegge andre brukeres opplevelse av tjenestene. Nylig har det dessuten blitt et økende juridisk press på disse tjenestene til å forhindre og fjerne innhold som sprer og oppfordrer til hat. For at fjerning av slikt innhold skal være praktisk gjennomførbart trengs det systemer som er i stand til å oppdage hatprat i tekst automatisk.

Automatisk identifisering av hatefulle og krenkende ytringer har vært et aktivt forskningsfelt i 20 år. Likevel er de beste systemene fortsatt ikke gode nok til å skille ut hatprat på en helautomatisk måte. Av den grunn fortsetter denne masteroppgaven arbeidet med å nå dette målet.

På grunn av det økende juridiske presset til å fjerne hatprat, kombinert med det store antallet tjenester og plattformer dette angår, er det nødvendig med metoder som identifiserer hatprat uten å bruke informasjon som er spesifikk til en bestemt plattform. Med slik informasjon vil metoden være begrenset til bruk på den plattformen. For eksempel vil den informasjonen som ligger lagret om forfatteren av en tekst kunne variere mellom forskjellige tjenester, så bruk av slike data vil redusere brukbarheten på andre plattformer. Hvis slik informasjon unngås, derimot, vil metoden kunne brukes på mer generell basis. For å overholde dette kravet tar forskningen i denne masteroppgaven sikte på å unngå plattformsspesifikke data, og bruker derfor utelukkende tekstbaserte input.

Denne oppgaven foreslår en ny metode til identifisering av hatprat, som virker ved bruk av dyp læring. Mer spesifikt bruker metoden en todelt arkitektur, og kombinerer både konvolusjonelle nevrale nett (CNN) og langt-korttidsminne (LSTM)-nettverk. Den foreslåtte arkitekturen bruker tegnbaserte n-gram og ordvektorer som input til de to delene, som så slås sammen og gir en endelig klassifisering. Eksperimentene viser at denne arkitekturen, med de konfigurasjonene som gir best resultater, er bedre til å identifisere hatprat enn de aller fleste systemene beskrevet i nyere forskning.

Preface

This Master Thesis was written by Johannes Skjeggestad Meyer during the spring of 2018, as part of the Master of Science degree at the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor, Björn Gambäck, for great help with finding relevant literature, and for providing insightful feedback and commentary throughout the semester. In addition, I would also like to thank Elise Fehn Unsvåg for interesting discussions on the subjects covered in this thesis.

Johannes Skjeggestad Meyer
Trondheim, 4th June 2018

Contents

List of Figures

List of Tables

1 Introduction

Since the advent of the Internet, publishing one's opinions has become a simple matter. As time has passed, an increasing number of arenas have become available, from Internet fora and blogs, to microblog services like Twitter and social media like Facebook. However, in all arenas that are open to user generated content, there is a risk of some users misusing this opportunity to purposefully insult others, or even to convey hateful messages. This is often in breach of the given arena's terms and conditions, and sometimes, in some countries, even illegal. As such, detection of these messages is desirable.

In this thesis, a Deep Learning-based approach to achieve such detection will be described, along with the experiments performed in order to test and evaluate it it. To that end, this chapter first presents the motivation for doing such research. It then describes the goal and Research Questions underlying the work presented in the remaining chapters. A summary of the contributions of this work is then given, before the structure of the thesis is explained.

1.1 Background and Motivation

Research has been done on the automatic detection of insulting or abusive language for the past 20 years. In an early work, Spertus (1997) made a system for detecting so-called flames, or abusive messages, in e-mails, which she claimed often have more explicitly formulated flames than public statements. Later work has focused more on the publicly available messages.

Recently, there has been substantial increase in the interest into detection of abusive utterances from society at large. More specifically, this interest has been on the need to quickly remove hateful messages from online services like Twitter and Facebook. In the spring of 2017, the German cabinet accepted a proposition to force social network services to remove hateful content quickly, with fines of up to €50 million if breached. This proposal requires the services to remove obviously illegal content within 24 hours, while other, more uncertain cases have to be removed within 7 days (Thomasson, 2017).

In late September 2017, the European Commission (2017) published guidelines for online platforms on prevention and removal of content that incite hate, violence or terrorism. As part of these guidelines, the Commission advised that the online platforms should invest in technologies for automatic detection of illegal content of this kind. This is largely because completely manual moderation quickly becomes infeasible. On platforms like Twitter and Facebook, the amount of content to moderate becomes so large, manually going through everything is practically impossible. Even for smaller sites, going through everything within the 24 hour limit proposed by German authorities would be very difficult. At the same time, this law proposal only applies to platforms above a certain size, meaning manual moderation even less viable an option for those impacted. An alternative approach, not publishing anything until it has been moderated, would comply with the requirement of not allowing hate speech to be publicly available. However, the incurred delay would in many cases be unacceptable to users. The archetypical example of this is a chatroom, which would entirely lose its function with such delays. But also in other platforms, the impact on user experience could be detrimental to the platform's use. In other words, there is a great need for systems that automatically recognise hate speech in text, so that the requirements mentioned above can be fulfilled.

The need for detection systems is further emphasised by a recent study, showing that, on the Facebook-pages of Norwegian TV and news channels NRK and TV2, every 10th comment was hateful (Norsk Telegrambyrå (NTB), 2017). Adding the fact that these comments were extracted no sooner than 12 hours after being posted, so the respective media outlets had some time to moderate the debate, this illustrates how acutely such detection systems are needed.

The research presented in this thesis was aimed at further improving the quality of automatic hate speech detectors. Furthermore, this was done in a manner that attempted to minimise domain dependency, so that, in particular, minor actors in the online platform community may apply similar technology.

1.2 Goals and Research Questions

Goal Automatic detection of hate speech in text, independent of the online platform it originates from.

The goal of this research was to automatically and correctly classify text as hate speech or not hate speech. This was to be done purely based on the text available, so as not to make the classification dependent on domain specific knowledge and metadata, such as information about the author. That way, the hate speech detection could be applied to texts from various different online platforms.

Research question 1 What are found to be useful features and representations of text for hate speech identification, and what methods have been found effective for such detection?

In order to identify hate speech, suitable features and representations of the text must be used. In addition, the features must be suitable to the methods that apply them. To discover what features have been found to be effective, and in combination with what methods, a review of related literature was performed.

Research question 2 Would a deep learning model based on a two-pronged combination of a Convolutional Neural Network and Long Short-Term Memory-networks be effective at classifying tweets for hate speech?

Several different forms of Artificial Neural Networks have been used for hate speech identification. However, limited effort has been expended on approaches that combine different forms. Specifically, architectures that combine two parallel sections, and that combine Convolutional Neural Networks, which excel at detecting local patterns, with Long Short-Term Memory-networks, which are good at picking up on long-term dependencies, remained untested up until the work described in the following chapters.

1.3 Contributions

This thesis contributes to the area of hate speech detection through a review of literature on earlier research on the field, as well as through the introduction of a novel, deep learning-based approach to identification of hate speech in text. In short, the contributions of this work can be listed as:

1. Character n-grams and word embeddings are found to be the most useful input features for hate speech detection.

2. Character n-grams have been effective in many kinds of classifiers, while word embeddings have mainly been useful to deep learning-based approaches.

3. A two-pronged, deep learning-based architecture for detecting hate speech was introduced, combining Convolutional Neural Networks and Long Short-Term Memory-networks, and using both Character N-grams and Word Embeddings as input.

4. The proposed architecture was tested, using many different layer configurations. The system achieved an F_1-score of 79.24, calculated using macro average.

5. The achieved performance was higher than that of most comparable state-of-the-art systems.

1.4 Structure of the Thesis

Chapter 2 will present the background theory necessary to understand the work in this thesis, both in the area of machine learning and on aspects of Natural Language Processing.
Chapter 3 will describe prior work done in the field of hate speech detection, in terms of the methods and features used, as well as the developments and changes in what has been the target of detection.
Chapter 4 will present various data sets used in research on hate speech detection, along with a more thorough description on the data set used in the experiments presented in this thesis.
Chapter 5 will describe the proposed architecture for a hate speech detection-system in detail.
Chapter 6 will present the experiments performed as part of this work, along with the corresponding results.
Chapter 7 will provide an evaluation of the results from Chapter 6, and discuss these evaluations in regards to the Research Questions of Section 1.2.
Finally, Chapter 8 will give concluding remarks, and present ideas for future work.

2 Background Theory

This chapter provides the background theory for this thesis, including both the theory and methods necessary to understand the related work in Chapter 3, and the theory used in the research presented in this thesis. First, Section 2.1 will introduce different approaches used in traditional machine learning. Section 2.2 will then look more closely at the branch of machine learning called deep learning, as this is of particular importance to the thesis. In the end, Section 2.3 will present various features from Natural Language Processing that may be used in text classification.

2.1 Machine Learning

Machine learning has proven to be a necessary component of advanced text classification. There are many different approaches to machine learning. In the following, some of the most relevant methods for the area of hate speech identification are presented. First, this section will describe some standard, relatively simple machine learning approaches. Section 2.2 will then describe various approaches in the subfield of deep learning.

2.1.1 Logistic Regression

Logistic Regression (LR) is a regression model originating from the field of statistics. It was first described by Cox (1958) as a way of using regression when the variable one wishes to determine has discrete, categorical values. In its original form, Logistic Regression was made to determine binomial variables. That is, where variables have a binary domain, typically represented as $\{0, 1\}$. Later, it has been expanded to also work on multinomial variables, with different variations depending on whether the values are ordered or not.

In addition to its use in statistics, LR has become a much used supervised machine learning algorithm. Within this field, it is primarily used for solving binary classification problems. Being fairly straightforward and fast-learning, it has become a standard approach for solving such tasks (Russell and Norvig, 2014, Chapter 18).

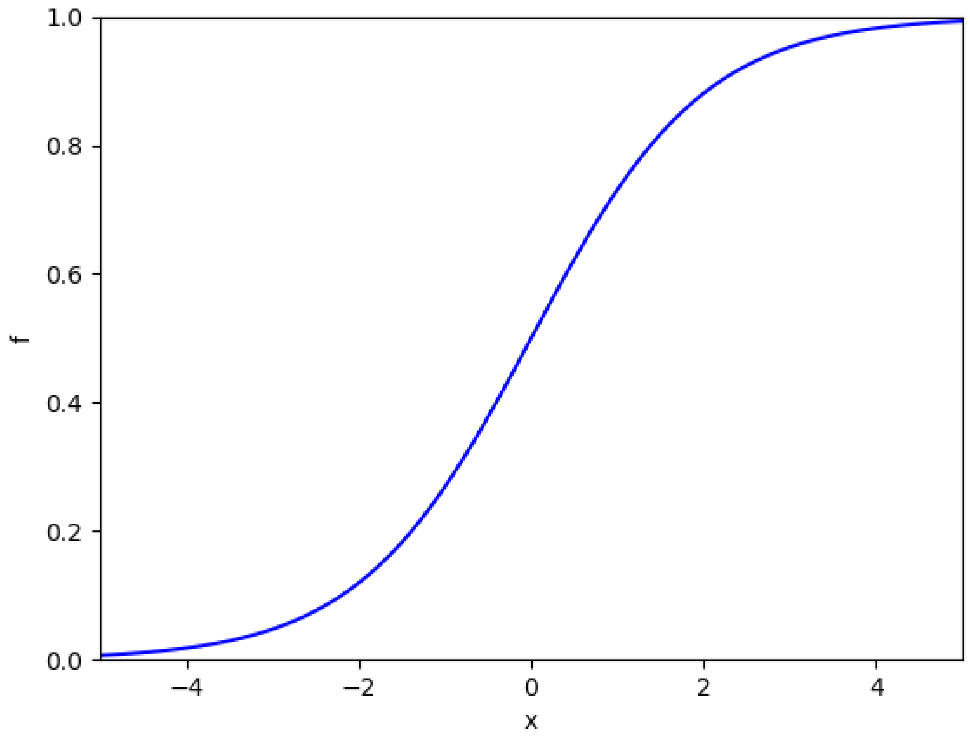

Figure 2.1: The shape of a sigmoid, or logistic, function.

At the heart of Logistic Regression is the logistic, or sigmoid, function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

In machine learning, the value of f, as depicted in Figure 2.1, is the estimated probability that, for instance, the comment x is hate speech (i.e. classified as 1, or positive). The function is often written a little differently when used in machine learning:

$$P(x; \theta) = \frac{1}{1 + e^{-\theta^T x}} \tag{2.2}$$

where $P(x; \theta)$ is shorthand for $P(y = 1|x, \theta)$. Here, θ is what the learning algorithm aims at discovering, while the T means using the transpose of this vector. θ is found using gradient descent or other approaches based on the maximum likelihood principle (Russell and Norvig, 2014, Chapter 18).

2.1.2 Support Vector Machines

A Support Vector Machine (SVM) is a machine learning model for binary classification tasks. An SVM considers the features of a data sample, in the form of a vector, as a point in a multidimensional hyperplane. The learning in an SVM consists of finding a linear separation between the data points of the different classes in this hyperplane.

In the original version of the algorithm, the dimensions used were simply the features of the data samples. However, in many cases, the classes are not linearly separable in the hyperplane specified by these dimensions. To solve this issue, Boser et al. (1992) proposed using what is now known as the kernel trick. In this trick, one maps the features of the samples into a higher-dimensional space. If the resulting space is of sufficiently high dimension, virtually any data set will be linearly separable. The resulting separator can be arbitrarily shaped, and often squiggly, in the original hyperplane, as on the left hand side of Figure 2.2. The separator of the SVM is set to the midpoint between the two dotted lines in the figure, which are determined by the samples closest to those of the opposite class in the higher-dimensional space, as shown on the right hand side of Figure 2.2. These samples are known as *support vectors*, and are the origin of the name Support Vector Machine. The support vectors are the only samples that need to be retained after training, as the separator is based solely on these, disregarding the rest of the samples. This makes SVMs quite memory efficient.

A problem with the kernel trick is that in the case of noisy data, the model is overfitted, with the separator going out of the way to account for the noisy data points. For instance, if the black circle in the 'nook' of the red separator line on the left half of Figure 2.2 was a noisy sample, it would cause a not insignificant deformation of the separator. Thus, the resulting classifier would have a worse performance than without that sample. To avoid this issue, Cortes and Vapnik (1995) proposed what is known as the *soft margin*-approach, in which one uses a not-quite-so-high dimensional space. This way, the separation of the two classes is not complete, as some data points violate any attempts at linear separation. For each wrongly classified sample, a penalty is assigned, dependent on how far across the border it falls. As data tend to be at least somewhat noisy, the soft margin-approach has become the standard use of SVMs.

2.1.3 Naïve Bayes

Naïve Bayes is a classification approach in machine learning that is, essentially, based on Bayes' rule from probability theory. Bayes law (Bayes, 1763) states that the probability of one event, A, conditional upon another event, B, can be calculated by using the opposite conditional probability and the probability of each

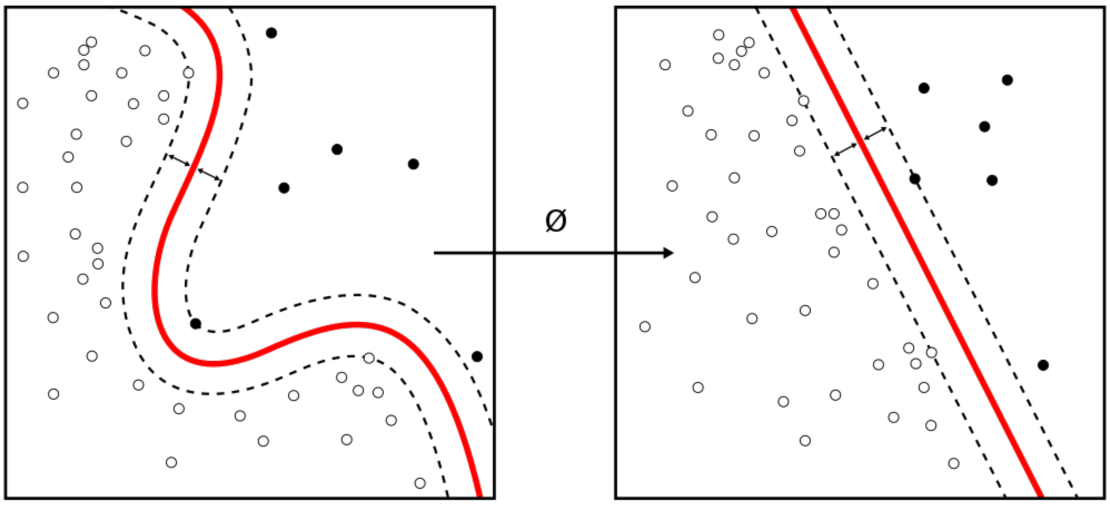

Figure 2.2: Illustration of the kernel trick of SVMs. The left half shows the separator in the original hyperplane, while the right shows the corresponding higher-dimensional linear separator. Figure by Sadreddini (2011), used under the Creative Commons Attribution-Share Alike 4.0 International licence.

event independently. More formally, it is defined as:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{2.3}$$

In a Naïve Bayes classifier, the more general version of this equation is used to determine which class is the more likely one given the attributes, or features. Specifically, the classifier calculates, for each class C, the probability:

$$P(C|x_1, ..., x_n) = P(C) \prod_{i=1}^{n} P(x_i|C) \tag{2.4}$$

where $x_1, ..., x_n$ are the features. The class with the highest probability is then the winner. The naïve part of these classifiers is their assumption that all the features are conditionally independent, thus allowing the use of a product of all the conditional probabilities. While this assumption is rarely accurate in the real world, and perhaps even less so for text classification, Naïve Bayes classifiers have proven quite useful. They are also resilient in the face of noisy or missing data (Russell and Norvig, 2014, Chapter 20).

2.1.4 Cross Entropy

Cross entropy is a method of calculating the deviation between two sets of values. It is, in other words, a way of calculating loss or error in a prediction. As such, cross entropy is not a machine learning approach in and of itself, but it is commonly used in the training of machine learning systems. In particular, it is widely used when the system outputs and the target labels represent probability distributions. The cross entropy, H, is calculated using the following formula:

$$H(P,T) = -\sum_{k=1}^{n} T_k log(P_k) \tag{2.5}$$

where P is the system's predicted values and T is the target label. n here is the length of the label vector. In the case of Logistic Regression, for instance, this length would be 2, and P would be of the form $[p, 1-p]$, where p is the value predicted by the regression.

Cross entropy is particularly useful in multiclass classification tasks, where the label is a vector of zeros with a one in the location corresponding to the given class. In this case, the system to be trained would output predictions stating the probability a sample belongs to each possible class. The cross validation would then compare these probabilities to the label, calculating the error. Due to the T_k-factor in Equation 2.5, the cross entropy would here disregard all the predictions but the one for the correct class, as the label value for these is zero. This can be avoided by using small, non-zero values instead. Simultaneously, the 1-value must be reduced correspondingly, in order to maintain a normalised state.

2.2 Deep Learning

In the types of machine learning described in Section 2.1, there is a requirement for a good representation of the data upon which the algorithm is meant to make its decision. In other words, the algorithms need appropriate *features* in order to work well. Finding the appropriate features for a certain task, however, can be highly difficult. Deep learning solves this issue by building representational hierarchies, with the lowest level typically working on much simpler representations of data than what is necessary in other approaches, in some cases even the raw data themselves. Higher levels then use increasingly complex concepts of the data. An example would be object recognition in pictures. The lowest level would look at the pixels of the image, while the next might identify edges. A higher level layer may, for instance, recognise shapes, with the highest level, or output level, recognising the specific kind of object. There are, however, some cases where unsupervised pretraining on unlabelled data can be used to find more informative representations, so that the actual, supervised hierarchy can start from a higher level. One such example is the word embeddings of Section 2.3.4.

Another view of deep learning is that, rather than working on concepts found in the input data, it is a series of small, parallel sets of operations. In this view, the complete system can be seen as learning a program divided into several sequential steps, and each layer represents a further state of the system.

While many believe deep learning to be a relatively new field, it is really just the name that is new. The basic ideas behind deep learning stretch back to the field of *cybernetics*, which started out in the 1940s. It may be added that cybernetics started out as a field separate from that of digital computers, but gradually turned to build on these. Later, some of the same principles reappeared under the name *connectionism*, covering roughly the 1980s and the 1990s. After this, interest in the field was generally low until around 2006, when it returned under its current name (Goodfellow et al., 2016, Chapter 1).

The disappearances and reappearances of the field of deep learning coincide with the discovery of limitations to, in particular, the *perceptron*-model, and subsequent solutions to said problems. A perceptron is a simple model of a neuron. Simply put, the backbone of deep learning consists of networks of perceptrons and other, related models. These networks are known as artificial neural networks.

2.2.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is a simple model of the networks of neurons in the human brain. Each neuron takes a number of inputs, sums them, and produces an output, or *fires*, depending on an activation function. The exception is the input neurons, which merely transmit their values to the next layer. In the oldest versions of ANNs, the neurons were called perceptrons. In a perceptron, the activation function is a strict threshold. When later approaches changed the activation function, the name, too, was changed. For simplicity, therefore, all types will here be called *nodes*.

Every connection between two nodes in an ANN has a weight. The output from one node is multiplied with this weight before it is used in the sum of inputs to the activation function of the other. In addition, each node can have a bias. This often comes in the form of a weight from a *bias node* with constant output 1, which is included in the input sum of the node. Thus, the node can be described mathematically by:

$$a_j = g(\sum_{i=1}^{n} w_{i,j} a_i), \tag{2.6}$$

where a_i is the output from node i, g is the activation function, and $w_{i,j}$ is the weight of the connection between nodes i and j. The learning in an ANN happens by adjusting all the weights in the network. The challenge of how to do this in a good way is what has caused the historical disappearances and resurgences of deep learning.

While the descriptions here of neural nets, both above and in the following, are of networks of interconnected nodes, this is not the only way of considering them. Indeed, modern systems for implementing ANNs, such as TensorFlow[1], see the networks as series of matrix operations, as this is a much more efficient, yet analogous, approach. However, as the node-based view is much more intuitive, and the origin of the vocabulary surrounding ANNs, this model is used in the explanations here.

The simplest forms of ANNs are so-called feed-forward networks. Here, the nodes form layers, with each layer receiving input only from the previous layer. Similarly, it transfers its output only to the next layer. In the oldest versions of ANNs, the nodes were perceptrons with threshold activation functions. All ANNs at the time were single-layer neural networks, or perceptron networks, with all input nodes connected directly to the output nodes. This was because no-one knew how one could change weights at a deeper level. As a consequence, as Minsky and Papert (1969) showed, the networks could only solve linearly separable problems. In particular, even such a simple task as *exclusive or* (XOR) could not be learned by these systems. This discovery was what caused the first death of neural networks.

The resurgence of deep learning came around when the backpropagation algorithm was discovered. This algorithm provided a way of consistently updating weights not only in the last layer, but in deeper, so-called hidden layers. The backpropagation algorithm works by updating the weights through gradient descent based on each weight's influence on the end result, or, mathematically put:

$$\Delta w_i = -\eta \frac{\partial L}{\partial w_i} \tag{2.7}$$

for all weights w_i. Here, L is the loss, or error, in the results compared to the label of the current input sample, and η is the learning rate. As a threshold function is not differentiable, this meant one could no longer use the basic perceptrons. Instead, the easily differentiable sigmoid function of Equation 2.1, was used to approximate a threshold. Using backpropagation and the sigmoid function, one suddenly had the theoretical possibility of arbitrarily deep networks. This solved the problem from Minsky and Papert (1969), and allowed the ANNs to solve many challenging tasks.

Despite the great improvement of the sigmoid backpropagation approach, it showed some practical restrictions. As Figure 2.1 shows, when the output of the

[1]https://www.tensorflow.org/

sigmoid approaches 0 or 1, its gradient becomes very small. For weights deep in the network, the derivatives of many consecutive weights are multiplied. If several of these are small, which often becomes the case, the resulting adjustment to the weight in question becomes practically negligible. As a result, ANNs with more than a few hidden layers become practically impossible. This problem, called saturation, eventually caused the second death of deep learning.

The current wave of deep learning started with the realisation that one could use the Rectified Linear Unit (ReLU)-function as activation function. This is defined as $ReLU(x) = max(0, x)$, or, more clearly:

$$ReLU(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{2.8}$$

This approach avoids the saturation for high input sums in sigmoids, and so opens up the possibility of many more layers. There will still be saturation for negative input sums, but this has proven to be of much smaller effect. Still, researchers have made several variations on the ReLU function in attempts to mitigate this.

The kind of feed-forward networks described here, also known as multilayer perceptrons (MLP), can solve many problems. They are, however, not the only kind of networks used in modern deep learning. In one other version, a node can, in addition to the elements described above, send its output back to nodes earlier in the network, creating loops. These kinds of networks are called Recurrent Neural Network (RNN). Another variation are the Convolutional Neural Network (CNN). These are, in their basic form, strictly feed-forward, but the nodes use more complex functions than in MLPs. These kinds of deep learning networks will be described in the following.

2.2.2 Recurrent Neural Networks

As the name suggests, Recurrent Neural Networks (RNN) (Rumelhart et al., 1986) have recurrent values. In other words, the network includes loops. These loops enable an input to influence the outcomes of future inputs. This property makes RNNs highly suitable for sequential data where there is some relation between the data elements, such as in text. Furthermore, RNNs can treat sequences of much greater length than what is practically possible with MLPs (Goodfellow et al., 2016, Chapter 10).

Another useful property of RNNs is that, unlike MLPs, they can generally take input of varying size. If, for instance, the network runs on sequences of numbers as input, all that is needed to deal with an extra element is to run the network for one more round/time step. This is a highly useful feature in the case of text processing, as sentences, and texts in general, tend to vary greatly in length.

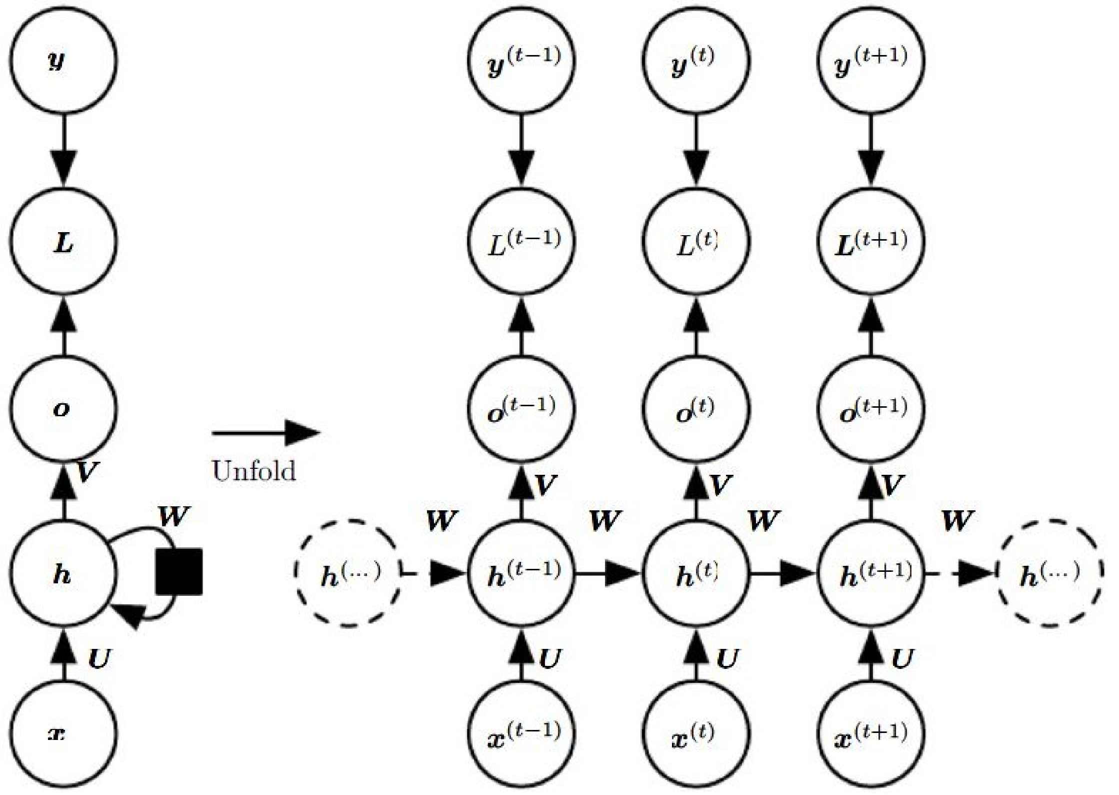

Figure 2.3: (Left) Graph of a simple RNN. (Right) Unfolded version of the graph, around time step t. With permission from Goodfellow et al. (2016).

While the recurrent element may make backpropagation seem difficult in RNNs, this is actually not the case. ANNs can generally be represented as graphs, with input nodes, output nodes and hidden layers in between. In RNNs, these graphs contain self arrows or arrows going against the feedforward direction. However, as illustrated in Figure 2.3, for any given input, the graph can be unfolded so that it is not all that different from a normal ANN. The main difference is that the contributions of each weight to the loss has to be calculated for every time step separately. The sum of these contributions over all time steps are then used as that input's influence on the weight updates.

One problem with regular RNNs is that the output can only be affected by preceding inputs. In some tasks, however, the output is really dependent on future inputs as well. A typical example of this is speech recognition, where both past and future phonemes affect the current one and its meaning. To solve this problem, Schuster and Paliwal (1997) proposed the bidirectional RNN, where an extra RNN is added that operates on the reversed input. In other words, it operates backwards through time. The normal and the reversed networks are combined in the output, allowing this to be influenced also by the future values of the sequence.

Another issue with RNNs is that recent inputs are much more influential than older ones. Thus, if the truly relevant input occurred early in a long sequence, the final output is virtually unaffected. The reason for this problem is that, in an RNN, the same function is used on the hidden state again and again. This has an effect similar to the problem making gradients all but disappear described in the ANN-section above. The effect can, to some degree, be mitigated using bidirectional RNNs, but this is not a particularly good solution, as the middle of long sequences would still largely be forgotten.

2.2.3 Long Short-Term Memory

The Long Short-Term Memory (LSTM)-networks are a particular kind of RNNs that aim at solving the above issue. They were first described by Hochreiter and Schmidhuber (1997), and further developed by Gers et al. (1999). In an LSTM, each 'node' is really a more complex object, with several elements. One such node is shown in Figure 2.4. The node has four elements; one cell and three gates, controlling input, output and forget rate, respectively. The cell is arguably the main part. This is where the information is stored from one step to the next. The input and forget gates enable the cell to retain only interesting information, and to remember it for longer than what a normal RNN would. At the same time, uninteresting information may be forgotten quickly. The output gate controls the extent to which the node should produce an output at that given time step. In Figure 2.4, the circles with the s-shapes represent differentiable activation functions, such as the sigmoid function. In addition, each of the gates has an activation function. Here, the sigmoid is the standard. The small circles with x-es represent element-wise multiplication.

As Figure 2.4 shows, the input values x_t, from time step t, are used several places in the node, both in the three gates and in the first activation function. The results of the activation function are then multiplied (pointwise) with the output of the input gate, and used to update the values in the cell. The updated cell value is the sum of this input value and the product of the previous cell value and the forget rate. This updated value is then run through another activation function and multiplied with the value of the output gate, thus producing the output of the node at the current time step. Formally, the workings of an LSTM-network are described with a set of equations, including one for each of the gates. The forget gate value $f_i^{(t)}$, controlling the information loss of the state of node i at time step t, is set by:

$$f_i^{(t)} = \sigma(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}) \tag{2.9}$$

where σ is the sigmoid function described in Section 2.1.1. $\boldsymbol{b}^f$ is the vector of

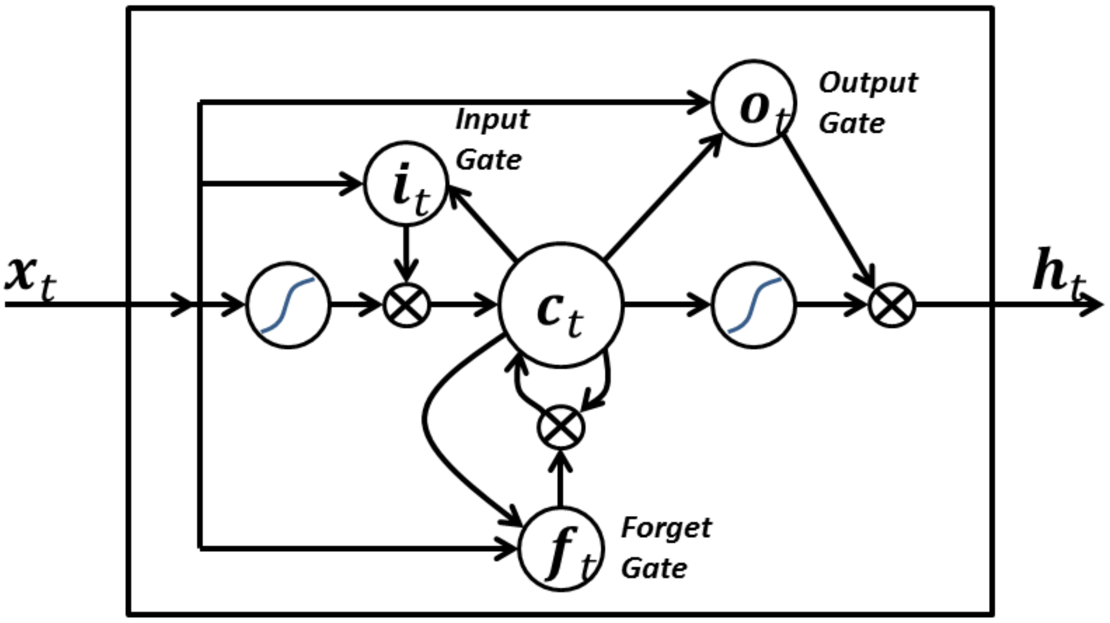

Figure 2.4: Illustration of a peephole node in an LSTM network. Adapted from Graves et al. (2013), under the Creative Commons Attribution-Share Alike 4.0 International licence.

biases to the forget gates, while U^f denotes the LSTM layer's matrix of forget gate input weights. W^f is the equivalent for the recurrent weights, controlling the impact of the LSTM-layer's outputs at the previous time step. The j in the sums go through each node in the LSTM layer.

The input gate's value, $g_i^{(t)}$, is similarly defined as:

$$g_i^{(t)} = \sigma(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}) \tag{2.10}$$

where $\boldsymbol{b}^g$, U^g and W^g are the input gate equivalents to the vector and matrices in Equation 2.9.

The input and forget gates work together to determine the next value of the internal state of each LSTM cell, $s_i^{(t)}$. Formally, this is done according to the following equation:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}) \tag{2.11}$$

where $\boldsymbol{b}$, U and W are the main biases, external input weights and recurrent input weights of the LSTM layer.

The output of the LSTM node at time step t, $h_i^{(t)}$, is set as:

$$h_i^{(t)} = tanh(s_i^{(t)})q_i^{(t)} \tag{2.12}$$

where $tanh$ is the hyperbolic tangent and $q_i^{(t)}$ is the value of the output gate, determined by:

$$q_i^{(t)} = \sigma(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}) \tag{2.13}$$

with $\boldsymbol{b}^o$, U^o and W^o being equivalent to the parameters of the forget and input gates. Combined, these equations determine the behaviour of an LSTM node.

The node shown in Figure 2.4 is of a particular kind called a peephole node. In this version, the gates also use the state of the cell from the previous time step, $s_i^{(t-1)}$, as part of calculating their outputs $f_i^{(t)}$, $g_i^{(t)}$ and $q_i^{(t)}$. This is illustrated by the arrows from the cell to the gates.

As Equations 2.9–2.13 imply, the LSTM is recurrent not just in the recurrence of the internal state s_i, but also at a higher level, with self-loops feeding a layer's outputs back to use as input at the next time step. Thus, the LSTM is really like a normal RNN, just with more complex nodes. Consequently, an LSTM has more parameters to learn than a simple RNN, with additional biases and two sets of weights for each of the three gates.

There is another version of the LSTM-node, called the Gated Recurrent Unit (GRU) (Cho et al., 2014), which reduces the number of gates, and thus the number of weights to learn, by using the same gate for the forget-rate and the updating influence of the input. In other words, a GRU has only two gates, called the *update* and *reset* gate, respectively.

LSTMs have been highly successful, particularly in tasks related to natural language processing, and have become one of the standard deep approaches to text processing tasks.

2.2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNN) (LeCun, 1989) are another form of ANNs. Unlike RNNs and LSTMs, a pure CNN is strictly feed forward. However, it functions in a different manner than conventional ANNs. CNNs work best on data with some grid-like structure. As such, they are very effective on image processing tasks. For instance, they have been extremely successful with object recognition in images. One example of this, and what may be said to have sparked the current, vast interest in CNNs, is the system of Krizhevsky et al. (2012), with its victory in the ImageNet competition.

While CNNs may be best known for their success in image processing, they also work on input with a one-dimensional structure. As such, they can be applied to text processing tasks as well.

What makes CNNs different from standard ANNs is the use of the *convolution* operator. In the convolution, a *kernel* is applied to the input. While not entirely relevant for this thesis, convolution over an image will be used to explain this process, as the operation here is fairly intuitive. The image, or input, is, in this case, a two-dimensional grid of pixels. The kernel is a much smaller two-dimensional grid, and it is applied to *neighbourhoods* of pixels. Each element in the kernel can be seen as a weight, which is multiplied by the corresponding pixel in the neighbourhood. All the resulting values from the neighbourhood are summed and stored in the corresponding location in the output. In the case of images, the output is also a two-dimensional grid. The kernel is then applied to the next neighbourhood. This is typically one row or column of pixels away. Once the kernel has been applied to the entire picture, the output is complete. Due to the non-unit size of the kernel, and it being applied only within the boundaries of the picture, the size of the output is typically somewhat smaller than that of the input. This can be remedied by using padding with certain values around the border of the image. Typically, the padding is filled with zeros; so-called zero-padding. In effect, the convolution operation means that the network only needs to learn a few weights for each layer, which are then shifted spatially and applied to all locations. When used in an ANN, the kernel is typically called a *filter*. One convolutional layer in a CNN can contain multiple filters, all running in parallel. That is, they are all applied to every location in the input. In the case of images, the convolution would then output one two-dimensional grid for each of the filters.

The convolution operation is a linear transformation. As such, in order to solve problems that are not linearly separable, the results are fed to a non-linear operator, such as ReLU. In addition, a third stage called the *pooling* stage or layer is often used. The effect of a pooling layer is to make the output less susceptible to small translations in the input. For example, if one wants to detect faces in a picture, detecting that there is an eye on the right and left side of the face is what is important, rather than the exact location of the eyes. One popular version of pooling is the *max pooling* (Zhou and Chellappa, 1988), where each output value is set to the maximum within a rectangular neighbourhood. Other types of pooling include taking the average of the values of the neighbourhood, or an average weighted over the distance from the centre of the neighbourhood.

In the case of a classification problem, the convolutional layers – including pooling – typically feed into fully connected, standard ANN layers. Here, the pooling layers may play a particularly important role. The pooling results work like summaries of their respective neighbourhoods. As such, they can be used to reduce the size of the input to the next layer. If the original input may be of varying size, as is the case for texts, the pooling can help conforming this so that all examples have the same number of input 'features' for the ANN part of the network.

2.2.5 Softmax

In ANNs, the activation values of the nodes in the final layer may have any sort of values. However, the point of interest is often in the relative difference between the different nodes. Specifically, it is often desirable to have some form of *normalised* output. Depending on the activation function used in the final layer, this is not necessarily a straightforward process. In the case of a sigmoid activation function, all values are in the range $(0, 1)$. Thus, each value can be normalised simply by dividing by the sum of all the values. The same applies to the case where the activation function is ReLU. However, for activation functions like the hyperbolic tangent and the identity function, the values may be negative. In such cases, normalising the values is not as simple. This is where *softmax* comes in. The softmax function, in addition to restricting all values to the range $(0, 1)$, also creates a probability distribution of the values (Goodfellow et al., 2016, Chapter 6). That is, the sum of all resulting values is 1. This is regardless of the input values to the function. In other words, it works with both positive and negative numbers, and is not restricted to a specific range. The softmax function is defined as:

$$softmax(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}} \tag{2.14}$$

where $\boldsymbol{z}$ is the vector of input values, and j is an index of this vector. In an ANN, $\boldsymbol{z}$ would be the node activation values of the layer, while j would denote a specific node. A consequence of the use of the exponential function here is that there is an intensifying effect. A value that is relatively much higher than the others gets an even stronger position in the output, while a value that is relatively much smaller than the others gets a disproportionately small output value. In other words, softmax strengthens tendencies in the input, providing an even higher ‘probability’ for nodes with higher values.

2.2.6 Optimisers

While ANNs can be trained through standard gradient descent-training, with a set learning rate, other variations with adaptive learning rates tend to give better results (Schaul et al., 2013). One such variation is the Adam optimiser. The Adam algorithm, whose name is an abbreviation of ‘adaptive moments,’ was first described by Kingma and Ba (2014). As in other adaptive optimisers, Adam maintains separate learning rates for each of the parameters to be trained. In other words, the different weights in the network have individual learning rates. These are then modified throughout the learning process.

An important element of the Adam algorithm is the use of the momentum of learning. In the original form, described by Polyak (1964) and applicable to normal

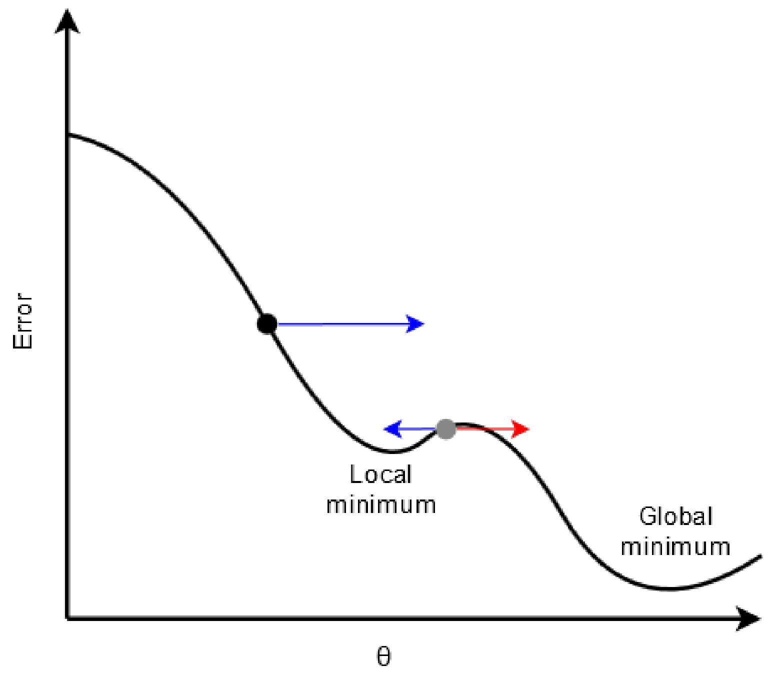

Figure 2.5: Gradient descent in a simple error/parameter landscape. The black orb shows the initial location in the landscape, while the grey orb shows the location after one step. The blue arrows show the influence of the current gradient, and the red arrow shows gradient with momentum. Without momentum, the system gets stuck in a local minimum.

gradient descent, the momentum method adds a new variable that represents the 'velocity' of the learning process. That is, it represents the direction and speed at which the learning moves through the parameter space. This is used to impact the movement at the next step of learning. Figure 2.5 illustrates one of the advantages with using momentum. θ represents the parameters whose error landscape is shown. When simply using gradient descent, the system will, at the second, grey location, move back towards the local minimum. As a consequence, it will get stuck in this 'bowl.' When the momentum of the previous step is included, the system manages to get over the ridge, and thus move further towards the global minimum. While this is a highly simplified example, with essentially only one parameter, the principle also applies to cases with many parameters and more complex error landscapes.

In the Adam algorithm, the momentum is represented as an estimate of the first moment of the gradient. This is in the form of a moving average of the gradient. In other words, it is an estimate of the gradient's mean value. In addition, the algorithm utilises estimates of the second moment, representing the variance of the gradient. This is estimated by a moving average of the squared gradient. At each step in training, these averages are updated with the new gradients. However, due to issues around the initialisation of the averages, both are biased towards zero. Therefore, they are subject to a bias correction before they are used further. After the bias-correction, they are used to update the system parameters. Specifically, this is done as:

$$\theta \leftarrow \theta - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \delta}} \tag{2.15}$$

where θ represents the parameters and α is the initial learning rate. $\hat{m}_t$ is the bias-corrected first moment estimate (mean) at training step t, and $\hat{v}_t$ is the bias-corrected estimate of the second moment at the same time step. δ is a small constant used for numerical stability, such as avoiding issues when $\hat{v}_t$ is zero. The suggested value of δ is 10^{-8}.

In addition to α and δ, there are two more hyperparameters in the Adam algorithm; β_1 and β_2. These control the decay rates of the first and second moment estimates, respectively. In other words, they control how much of the estimate comes from the current gradients, and how much is from the previous estimates. The suggested values of these parameters are 0.9 and 0.999, respectively. The algorithm generally works well when using the default values of both these parameters and the δ (Goodfellow et al., 2016, Chapter 8).

The Adam optimiser was not among the algorithms tested in the evaluations by Schaul et al. (2013), as it had not yet been published at the time those evaluations were performed. It is, however, among the more popular optimisation algorithms in use (Goodfellow et al., 2016, Chapter 8).

2.2.7 Regularisation

If a deep learning system, or any machine learning system, is trained simply using the techniques described above, it is likely to overfit to the training data. In other words, it will easily become so specialised it has difficulties solving similar problems not in the training set. However, the aim of the training is essentially to make a system that is generally applicable, in that it can be used on, as of yet, unseen data. There are many kinds of methods and strategies aimed at increasing the general applicability of a machine learning system. The use of such strategies is commonly known as *regularisation*.

One of the reasons for overfitting is that the learning continues for too long. In terms of a training set and a separate validation set, this means that the training continues beyond the point where performance on the validation set stops improving. At this stage, the performance on the training set keeps improving, but the validation set performance gets gradually worse. This issue can be mitigated by using *early stopping*. In early stopping, the system is, during training, regularly tested against a validation data set. The performance is then compared to previous validations. When the system starts to overfit, performance on the validation set will typically decline. As such, training is terminated at this point. To avoid terminating prematurely due to small dips or fluctuations in performance, the training continues until there have been a set number of validation rounds without improvements. The model with the best performance is then restored. Using this approach, overfitting due to excessive training is avoided.

While early stopping fights one cause of overfitting, it may not be sufficient to provide good generalisation. In ANNs, one indicator of overfitting is when weights get a high magnitude value, either in the positive or the negative direction. As such, there are methods that penalise high magnitude weights. The L^1- and L^2-regularisers are commonly used examples of this. In both methods, a penalty is added to the loss/error being minimised by the optimiser. The difference between the two methods lies in how this penalty is calculated. In L^1-regularisation, the absolute values of the weights are summarised and used as penalty. In L^2-regularisation, on the other hand, the basis of this sum is the square of the weights. With L^2-regularisation, the impact of small values on the penalty is diminished compared to L^1, due to the squaring of the values. Similarly, the impact of values greater than 1 is increased. As the initial weights of an ANN tend to be small, L^2-regularisation has less impact in the early stages of training than L^1-regularisation typically does. This is generally a good thing, as overfitting is unlikely to have occurred at this stage.

Another form of regularisation is more specifically directed at ANNs. Known as *dropout*, the technique was first described by Srivastava et al. (2014). When dropout is used on a traditional feed-forward ANN, it works by 'turning off' random nodes in the network. That is, the outputs of those nodes are blocked, not affecting the outcome of that particular run. The chance of any given node being turned off is set by a dropout probability. For each batch during training, the decision of whether to turn off is made anew for each node. This has the effect that the network that is seen, and, thus, is trained, differs between different batches. During testing and validation, however, all nodes are active. Only training a random subset of the nodes for each batch means that the risk of overfitting is reduced, as the network actually seen by the training procedure is different each time.

When dropout is applied to RNNs, it operates in a slightly different manner than described above. Instead of turning off a node for the entire mini batch, the decision is typically made separately for each time step. When the RNN is in the form of an LSTM or GRU, the state of a node is normally maintained despite the node being turned off. In other words, the temporal accumulation of data is not lost when a node is turned off. There are, however, variations of recurrent dropout that do not comply with either of these standards. One example is the approach described in Gal and Ghahramani (2016). Here, the dropout between layers is kept constant between time steps. In addition, dropout is applied to the states of nodes between time steps, with the same nodes dropping their state for all time steps.

There are many other forms of regularisation found in literature. However, the variations mentioned above are commonly used. They are also necessary to understand this thesis.

For a more thorough description of the various deep learning approaches described in the above sections, the interested reader is referred to Goodfellow et al. (2016).

2.3 Features from Natural Language Processing

In the realm of text classification, there are different features of the text that may be applied. These features can be purely based on the given text, or they could depend on a larger volume of text as a foundation. In the following, features of both kinds will be presented, though most belong to the latter group.

Most of the features in this section are based on words. In order to use these features, the text must first be processed in order to divide it into usable elements. This process is called tokenisation, and may include, e.g., turning all words into lowercase, so that one word at the beginning of a sentence, with a leading uppercase letter, can be treated similarly to the same word in lowercase occurring in the middle of a sentence.

Once the text has been tokenised, the words, or tokens, have to be represented in some manner. If there is a vocabulary based on a large volume of text, the simplest representation is with so-called one-hot vectors. These are vectors where exactly one element has the value 1, and the rest are set to 0. The vector is the size of the vocabulary, with the location of the 1 telling which word it represents. In addition, there must be a location to represent Out of Vocabulary (OOV) words; terms that did not occur in the texts the vocabulary is based on. Generally speaking, how to treat OOV-words is a challenge for all vocabulary-based text representation methods. The one-hot vectors have a very high dimensionality, and with only one location set to 1 at any time, the representation becomes very sparse.

2.3.1 Bag-of-Words

Bag-of-Words (BoW) is a text representation where the syntax of the text is completely disregarded. Each text is simply represented by a vector telling which words appear in it, and how many times they appear. This means that the vector

Sentence:	**brown**	**dog**	**fox**	**jumps**	**lazy**	**over**	**quick**	**the**
S1	1	0	1	0	0	0	1	1
S2	0	1	0	1	1	1	0	1

Table 2.1: Bag-of-Words-representation of sentences S1 and S2

is of the same kind as in the one-hot vector representation, only with more than one location in the vector having non-zero values. For instance, Table 2.1 shows a BoW-representation of the following sentences:

S1 The quick brown fox

S2 Jumps over the lazy dog

BoW is a very simple representation. However, like with one-hot vectors, it has a very high dimensionality. Even in the exceedingly simple example of Table 2.1, there are eight dimensions. In any practically useful example, the dimensionality would be vastly increased. Similarly, the sparsity of the representation would be far more pronounced, with the majority of the values being zero. The dimensionality issue can to some extent be mitigated by using stemming, so that different inflections of a word are represented the same way, but the vectors still get a very high dimension, with sparse representations. As such, BoW is not the best way of representing text, but it has inspired other, more useful, methods of representation.

2.3.2 Term Frequency – Inverse Document Frequency

Term Frequency – Inverse Document Frequency (TFIDF) is a concept from information retrieval. First described by Spärck Jones (1972), it combines the elements of term frequency with the inverse of the term's document frequency. More mathematically, term frequency can be described as:

$$TF(t;d) = \frac{\#\text{ times } t \text{ appears in } d}{\#\text{ words in } d} \tag{2.16}$$

where t is the term in question, and d is a document. The term frequency is considered a measure of how important the term is in the given document.

The IDF-part is a bit more complex. Here, it is not a single document that is considered, but a, preferably large, body of documents. The document frequency is a measure of how common the term is across the different documents. Terms that appear in all documents typically have little power to make distinctions between texts, and so they should be given a very small weight. Thus, the inverse document frequency is defined as:

$$IDF(t; D) = log\frac{N}{|\{d \in D : t \in d\}|} \tag{2.17}$$

where N is the number of documents in D, and the denominator is the number of documents that include the term t. This gives a higher weight to terms that only appear in a few documents. Combining the term frequency of Equation 2.16 with the inverse document frequency of Equation 2.17 then gives us the TFIDF:

$$TFIDF(t, d; D) = TF(t; d) \cdot IDF(t; D) \tag{2.18}$$

TFIDF originated, as mentioned, in information retrieval. In this area, it has been widely used in search engines, in order to evaluate a document's relevance to the search terms. Here, documents with a high prevalence of search terms that are relatively rare are considered more relevant and, thus, ranked higher. Use of common words like 'the,' however, have minimal influence due to the IDF-part of the TFIDF. This kind of words, carrying little to no information in a sentence, are often called *stop words*. Beyond its use in information retrieval, TFIDF has also been used in text classification tasks, as further described in Section 3.2.

2.3.3 N-grams

In n-grams, the consecutive elements within a window of size n are extracted, and may be used as features. This is done for all groups of n elements. The elements in question can be both whole words and single characters. When the window size is 1, the n-grams are called unigrams. For $n = 2$, the name is bigrams, while n-grams with $n = 3$ are called trigrams. For n-values higher than this, normal numbers are used. Character n-grams, unlike the approaches mentioned above, do not really require a volume of texts upon which to build the representations. Word n-grams, on the other hand, do. In fact, word unigrams are essentially the same as the Bag-of-Words-representations of Section 2.3.1.

When $n > 1$, word n-grams, unlike the BoW-approach, do pay some heed to the syntax of the text. For instance, the word bigrams of sentence S1 from Section 2.3.1 would be:

$$\{(\_, the), (the, quick), (quick, brown), (brown, fox), (fox, \_)\}$$

where '_' denotes the beginning or end of the text. The word ordering here determines which n-grams are found. This, however, implies that word n-gram features, like BoW-vectors, are very sparse, as there are a vast number of possible word combinations. This is compounded by the presence of conjugations and spelling errors. The former may be mitigated by preprocessing the text with stemming, but the latter is a real problem.

Character n-grams to a large extent avoid these issues. For one, there are much fewer combinations, there being only a few different characters. In addition, the stem of different inflections of a word will typically produce the same character n-gram. On a similar note, the influence of spelling errors will typically be smaller. N-grams have, as Section 3.2 will show, proven helpful in text classification tasks.

2.3.4 Word Embeddings

Word embeddings are a way of representing words that avoid the sparsity problem of one-hot vectors. They can be seen as an implementation of the famous quote by John Rupert Firth: 'You shall know a word by the company it keeps' (Firth, 1957). In other words, word embeddings, somewhat simplified, represent similar words in a similar way. More specifically, the word embeddings represent words as vectors of a much lower dimension than what is necessary for a one-hot vector representation. First introduced in Bengio et al. (2003), the values of these vectors are numeric, and, typically, neural nets are used to create the representations. As such, word embeddings are arguably most effective as input features for neural networks.

Word2vec (Mikolov et al., 2013a) is a much used method for creating word embeddings. In word2vec, there are two models upon which one may build the embeddings. These are the Continuous Bag-of-Words (CBoW) model and the Skip-gram model. The CBoW model is based on BoW of Section 2.3.1, in that, at training time, a word is represented using the BoW-representation of the words surrounding the target word. With a more metaphoric explanation, the word is estimated by the company it keeps.

The skip-gram model essentially works in the opposite way of the CBoW model. Instead of the surrounding words being used to predict the target word, the target word is used to predict the surrounding words. In both cases, the result of training, given an appropriately large volume of training data, is a numerical representation of words where similar words have short (Euclidean) distances between each other. This is a major advantage compared to one-hot vectors, where the distance between any two words is identical, at $\sqrt{2}$. This means that, in the case of ANNs, similar words produce similar reactions in the nodes, while very different words are likely to cause different reactions.

In addition to word2vec, Global Vectors (GloVe) is another method for creating word embeddings that is widely used. The GloVe-method was first described by Pennington et al. (2014). Both of the word2vec models mentioned above can be described as predictive, in that they attempt to predict what words belong to the given context. GloVe, on the other hand, is more explicitly a statistical approach, performing calculations on a matrix holding co-occurrence counts of different words. Specifically, the relationship between two words is determined using the ratios between the probabilities of those words' co-occurrence with other probe terms. This ratio works as an indicator of whether the probe word is more closely related to one of the two main words. For instance, if those two words are 'ice' and 'steam,' the ratio of probabilities for co-occurrence with the probe word 'solid' would indicate a greater relevance for 'ice.' Similarly, with the probe word 'gas,' the ratio would point to 'steam' as the more relevant term. This approach does not, however, see any difference between probe terms that are related to both of the main words, and those that are related to neither. For the example above, the probability ratio would be approximately 1 for both 'water' and 'fashion,' despite the highly different relevance to the terms 'ice' and 'steam.' As this indicates, the embeddings are created based on the differences in the use of words, rather than their similarities. The end result, despite the highly different approach to finding them, is similar to that of word2vec, with the embeddings of similar words having values marking coordinates short distances from each other in a Euclidean space.

An advantage of word embeddings, beyond that of short distances between similar words, is that the difference between related words is maintained across different categories. For instance, as Mikolov et al. (2013b) found, the difference between 'king' and 'queen' is similar to that between 'man' and 'woman,' which again is similar to that between 'uncle' and 'aunt.' Along the same lines, the difference between singular and plural form is fairly stable across different nouns. This allows systems using word embeddings to make connections between words, and may allow for generalisation over different forms, such as the singular/plural-distinction.

3 Related Work

As mentioned in the introduction, research into identification of abusive language has been going on for about 20 years. However, the approaches used have changed significantly over time. Early on, Spertus (1997) used a decision tree to classify text for flames, or insulting language, based on certain features of the text. These features were the output of a set of rules, each rule checking a sentence for the presence of words or formulations often found in insults, such as 'your so-called ...', or phrases that indicate absence of insults, e.g. 'thank.' The rules also included some site-specific elements; specifically, local *villains* and insulting terms, where speaking negatively of a villain was not considered a flame, while using one of the insulting terms *was* considered one. An example used by Spertus (1997) was the web site *NewtWatch*, focused on criticism of Newt Gingrich. Here, naturally, Newt Gingrich was considered a villain, and so insults directed at him were not considered flames.

In this chapter, the methods and various features and text representations used in abusive language detection will be described. First, Section 3.1 will describe the issues of using a rule-based approach like that of Spertus (1997). Section 3.2 will then describe other methods used in research on abusive language, together with the features applied in the different methods. In the end, Section 3.3 will look at what, exactly, the targets of classification has been in the research on abusive language detection, as these have varied considerably.

3.1 Rule-Based Approaches

There are some problems inherent in rule-based approaches to language-related classification tasks, like the one used by Spertus (1997). First of all, the authoring of the rules is very difficult. It requires a deep understanding of the language in general, and, in flame detection, how it is used in insults in particular. Secondly, it is susceptible to spelling errors. For example, Spertus (1997) noted that there were cases where 'your' being misspelled as 'you' caused insult-indicating rules to fire. Apart from such chance rule misfires, there is the possibility of intentionally writing obscene words wrongly, e.g. by using special characters like '@' or using leet-spellings like replacing 'e' with '3'. Yet another issue is that, if someone knows what the rules are, they can be avoided with relative ease.

One consequence of the first of these issues is that the system in question becomes deeply bound to one specific language. In the case of Spertus (1997), this language was English. To adapt the system to another language, the rules would have to be remade from scratch. Beyond that, the amount of work to make a rule-based system in the first place becomes prohibitive. As a consequence, later research has focused on other kinds of methods and features to use in the detection of abusive language.

3.2 Machine Learning-Based Approaches

The system described in Spertus (1997) did have a learning component, in the shape of a decision tree based on the results of the rules. However, the main part of the system was the rule base. The systems described in this section, on the other hand, relied more heavily on machine learning, using a multitude of different features and inputs.

In the system by Yin et al. (2009), the approach used in the detection differed greatly from Spertus (1997). Instead of using strictly rule-based features, the authors decided to use three different categories of features. The first category were local features, or features that can be extracted directly from the text. These were in the form of Term Frequency – Inverse Document Frequency (TFIDF)s, as described in Section 2.3.2, with high TFIDF-values indicating terms with high relevance for classification. The local features were represented as a vector holding the TFIDF-values of all terms in the vocabulary for the given input. The second category was a group of sentiment-based features. These were the features that were most similar to those of Spertus (1997). Yin et al. (2009) noted that harassing comments tend to contain foul language in conjunction to personal pronouns. As such, the features of this category contained three groups of TFIDFs. The first was the TFIDF of all second person pronouns, treated as if they were all the same term. The second was the equivalent for all other personal pronouns, while the third was the TFIDF of all the profanity terms in the authors' dictionary treated as one. In the third category, the authors used contextual features. The classification in Yin et al. (2009) was aimed at posts in online forums, and the authors noted that harassing comments tended to appear in clusters. In addition, they often had a somewhat different structure from the surrounding discussion. This was thus used to further improve classification. These three feature categories were then combined as input to a Support Vector Machine (SVM), like those described in Section 2.1.2.

Where Spertus (1997) used a decision tree learner and Yin et al. (2009) used an SVM to classify the text, Razavi et al. (2010) used a multilayer approach to the classification, with three layers of classifiers. The first step extracted the most helpful, i.e., discriminative, features, while the second layer used these to give a preliminary probability of whether the text contained flames or not. In addition, the authors used an Insulting and Abusive Language Dictionary (IALD), where each term and phrase was given an offensiveness value from 1 to 5. The second step of classification also outputted the number of terms identified from the different offensiveness groups of the IALD, as well as the distribution between them. The third and final step of the classification used all of this to give a final verdict. In all three steps, some variation of Naïve Bayes, described in Section 2.1.3, was used. Unlike Spertus (1997), this approach paid no heed to the syntactical structure of the text in question.

Sood et al. (2012) also used a multi-step approach to classification, performed on posts in a discussion forum about news articles. However, unlike Razavi et al. (2010), they ran several partial classifiers in parallel, with their collective outputs being used as input to a final classifier. The first of these used the stemmed versions of word bigrams (see Section 2.3.3) to detect whether the text contained an insult. In the second classifier, only those comments which were found to contain insults were inspected. Here, the classifier determined whether the insult was directed at an author of a previous comment, and as such someone related to the discussion, or if it was directed at some third party. Again, the decision was made based on stemmed bigrams. In other words, both of these classifiers made their decision based on the presence or absence of certain terms, using SVMs to determine the results. The third classifier looked at the comment's relevance to the article about which the given forum thread was concerned. This was done using TFIDF, where the document frequency was based on a number of Reuters articles. In other words, salient terms that appeared in both the comment and the topic article implied high relevance. This, again, suggested that the comment was of an acceptable kind. The fourth classifier used a sentiment analysis system, saying whether the tone of the comment was positive or negative, or neither. This evaluation was largely based on locating words typically found in positive or negative texts. The results of these four classifiers were then normalised and used as input to the last classifier, producing a final judgement. The resulting system performed significantly better than using just the first classifier. All in all, the entire system used single words and bigrams as its features, sometimes modifying them to account for conjugations and inflections.

Later, Mehdad and Tetreault (2016) looked at whether a character-based approach would be more effective than a word- or token-based one, such as that of Sood et al. (2012). Mehdad and Tetreault (2016) noted that a word-based approach is highly vulnerable to spelling errors, intentional or otherwise. Therefore, they proposed to use character n-grams, as described in Section 2.3.3, instead. While character n-grams are not unaffected by misspellings, the effect tends to be significantly smaller than on word-grams. The authors found that, indeed, the character-based approach worked better than token-based ones. This was consistent between two different systems, one Recurrent Neural Network (RNN)-based (see Section 2.2.2), and one SVM-Naïve Bayes hybrid, with up to 7% difference in terms of F_1-score (see Section 6.3). Of the two system varieties, the latter was the most effective, outperforming the state-of-the-art at the time.

From this point onward, much research has used character n-grams in its classification. One example is Waseem and Hovy (2016), who used n-grams from unigrams up to 4-grams. In addition, some other information was used, such as the gender and location of the author of the given tweet. The classification itself was done using logistic regression (see Section 2.1.1). Here, too, the performance was compared to that of word-based input, in the form of word n-grams. Again, characters outperformed words, with F_1-scores of 73.89 and 64.58, respectively.

Another paper that used character n-grams was Gambäck and Sikdar (2017). Here, however, the character n-grams were not used as the only text-based features. In addition to the characters, the input included word embeddings, as described in Section 2.3.4. These two groups of features were concatenated and input to a Convolutional Neural Network (CNN) (see Section 2.2.4). This approach is similar to that of Park and Fung (2017), except that in the latter, the authors created separate CNNs for the characters of the tweet and for the word embeddings. In addition, they created a hybrid version, with one part taking characters and another word embeddings, and where the two components were combined at a deeper level of the network. The system of Gambäck and Sikdar (2017) achieved an F_1-score of 78.29, while the Park and Fung (2017) system scored 82.7. However, as Section 7.1 will explain, these scores are not necessarily directly comparable.

Word embeddings have also been frequently used without consideration for character n-grams. For instance, Pavlopoulos et al. (2017a,b) used word embeddings as input to their RNN. The RNNs were in the form of a chain of GRU-units, described in Section 2.2.3, transforming the input word embeddings into hidden states. These were then used as input to a logistic regression function, deciding whether the comment in question was abusive or not. In addition, the RNNs were augmented by an attention mechanism, in the form of a Multilayer Perceptron (MLP) (see Section 2.2.1), giving an increased weight to the hidden states that were most likely to affect the outcome. The output of the attention mechanism was softmax-ed (see Section 2.2.5), and multiplied with the corresponding hidden states.

Another example of use of word embeddings is found in Badjatiya et al. (2017). Here, the authors used the embeddings as input to a Long Short-Term Memory (LSTM)-network (see Section 2.2.3). In one version of the system, the LSTM then made the decision whether the tweet in question contained hate speech. Another version used the output of the LSTM as input to a variation of a decision tree. In addition, several other variations, such as one using a CNN, were tried. The authors first trained the embeddings to fit the hate speech domain. To do this, they used both pretrained GloVe-embeddings (see Section 2.3.4) and random vectors as initial values. Interestingly, the best results were achieved using random vectors, outperforming the GloVe-version by 8% in terms of F_1.

Gao et al. (2017) also used an LSTM in their classification. This, however, was only one half of the system. In addition, there was a logistic regression system running in parallel. The LSTM was aimed at identifying implicit hate speech, while the logistic regression identified explicit slurs. The LSTM had word embeddings as its input, while the slur identification system worked on word unigrams. An interesting aspect of this research is that the learning was only weakly supervised. This means that it had a relatively small initial training set, but it gradually learned new slurs, and got new, labelled samples for further training of the LSTM. These samples were used as if supervised, despite not having labels set through supervision.

Founta et al. (2018), like Gao et al. (2017), also used a two-part system. Here, however, the two parts were more separate. One part used word embeddings, feeding them into an RNN-layer consisting of GRU-nodes. The other part was a separate, feed-forward ANN, running in parallel. Here the input was not text, but rather metadata of various forms. These metadata included, but were not restricted to, the number of emoji in the text, its sentiment and specific emotions, the popularity of the author in terms of followers and friends, and the age of his/her account. This was run through a six-layer ANN, whose output was combined with the output of the GRU. This was then used in a final classification layer.

The system described by Zhang et al. (2018), on the other hand, was purely text-based, using word embeddings as input to a CNN. After performing max-pooling on its output, the results were used in an RNN consisting of GRU-units.

Generally speaking, state-of-the-art systems used character n-grams or word embeddings as features. Each kind of feature has its own advantage. The character n-gram approach is, as mentioned, relatively hardy with regards to misspellings, while word embeddings allow related words to produce similar output. Word embeddings were mainly used in the deep learning-based approaches, while the state-of-the-art systems using other methods mostly used character n-grams. The n-grams are, however, successfully applied also in some deep learning-classifiers, particularly CNNs. Some systems, like that of Founta et al. (2018), also apply various forms of metadata and information about the author of the text. However, as the information available depends on what platform the texts are from, this makes the system more platform-dependent. The aim of this thesis is to achieve classification that is more independent of the origin platform of the texts. Thus, such platform-dependent systems have largely been omitted from this section.

3.3 Hate Speech Classification Targets

In the above, related works have been described in terms of *how* they classified the text. However, while all identified some flavour of abusive language, there is variation as to exactly what kind of abuse they looked at. Spertus (1997) looked at flames or insults, and took a context-dependent view, where some utterances could be insulting in some contexts while not being insulting in others. Fundamentally, however, she looked at the general notion of insults. This is a common feature among much of the early research, such as Yin et al. (2009); Razavi et al. (2010). In several of these general insult identifiers, foul language – e.g., curse words, obscenities – was a strong indicator of abusive language. This is true both for Yin et al. (2009) and for Razavi et al. (2010).

With time, the targets of identification have become more specific. Chen et al. (2012), for instance, had an extra focus on identifying cyber-bullying, as the goal of their research was to make Internet surfing safer for adolescents. Sood et al. (2012) kept their research within the realm of insults, but restricted their targets to be insults that had malicious intent, instead of insults in general.

Warner and Hirschberg (2012) was an early example of what may properly be termed hate speech detection. More specifically, their work was on identification of anti-Semitic utterances. Since then, there has been done much research on different aspects of hate speech. However, as noted by Waseem et al. (2017), different researchers rarely agree upon a definition of hate speech. They therefore proposed a typology for different categories of abusive language, depending on whether it is explicit or implicit, and whether it is directed at a specific individual or a more general group. Different kinds of abusive language detection would involve different categories in this typology. For example, identification of cyber-bullying would primarily concern itself with cases directed at individuals. Cases of hate speech, on the other hand, would typically lean more to the group-directed form, though not necessarily exclude the individual-directed kind. The early research in the field focused on the explicit kind of abusive language. This is arguably still the most used aspect, as, by its nature, implicit abuse is significantly more difficult to detect.

While this typology might make it easier to think about certain aspects of hate speech, it does not provide any definitions. The lack of an agreed-upon definition is problematic in that different researchers may seem to attempt the same thing, while what they really do differs substantially. For instance, Davidson et al. (2017) attempted to differentiate between hate speech and other offensive content as two different classes. However, they found many of the cases labelled as merely offensive to contain sexism, while most of the hate speech-labelled samples contained racism. In Waseem and Hovy (2016), the authors use these two categories as their classes of hate speech. In other words, Waseem and Hovy (2016) saw sexism as hate speech, while in Davidson et al. (2017), it was merely considered offensive. This discrepancy is, for a large part, due to the use of crowdsourcing in labelling the data sets. Waseem and Hovy (2016) asked specifically for the coders to identify these two classes, as the authors considered them good examples of hate speech. In Davidson et al. (2017), the distinction between hate speech and offensiveness was left somewhat more to the discretion of the coders. Thus, if the coders generally did not see sexism as hate speech, this would be reflected in the labels. In comparison, Waseem and Hovy (2016) labelled their data themselves, following a certain set of rules, in addition to using an external expert annotator. Thus, the definitions of the authors were clearly mirrored in the labels, with the labels of the expert working as corrections.

This difference based on the definitions used was further examined by Ross et al. (2017). Here, the authors had two groups labelling data. One group was shown the definition of hate speech used by Twitter, while the other used their personal perceptions of what constitutes hate speech. The results showed that, while the group with the definition appeared to align their opinions with what they saw the definition to mean, they disagreed on what exactly that meaning was, or how it applied to the tweets. Thus, while there was a high correlation between the two groups, the agreement between annotators, both within the groups and between them, was quite low. In other words, the definition used had, in effect, little use. This may be because the definition was of a very condensed and somewhat 'legalistic' form, and thus a bit impenetrable.[1] As such, using a more readable definition, e.g. the one proposed by Waseem and Hovy (2016), may be a better idea. The Waseem and Hovy (2016)-definition is described in detail in Section 4.3.

As these various definitions of hate speech and abusive language indicate, many different data sets have been used throughout the related work. In Chapter 4, a number of these will be explored in more detail.

[1]The definition used was: "You may not promote violence against or directly attack or threaten other people on the basis of race, ethnicity, national origin, sexual orientation, gender, gender identity, religious affiliation, age, disability, or disease. We also do not allow accounts whose primary purpose is inciting harm towards others on the basis of these categories." From the Twitter rules, as used by Ross et al. (2017)

4 Data Set

In supervised learning, there is a requirement for a labelled data set upon which the training can be done. The contents and labelling of the data set determine what the resulting system will be able to do. As such, having a data set with appropriate labels, and that is large enough to accommodate training and to be considered representative of the general input space, is a highly important part of creating an effective system.

This chapter will first provide a brief evaluation of data sets used in prior work related to hate speech identification. Then, there will be an introduction to Twitter and the Twitter API, which is where the data set of Waseem and Hovy (2016) – the data used in this thesis – is from. Finally, the Waseem and Hovy (2016) data set will be examined in more detail.

4.1 Data Sets in Prior Research

As evident in Section 3.3, there have been several different data sets used in the research on abusive language. A sample of these are shown in Table 4.1. However, as much of the prior research has not been focused specifically on hate speech, their data sets are not suitable for the research in this paper. One example of such a data set is the one used by Sood et al. (2012) with comments from *Yahoo! Buzz*, a news commenting site. That data set is focused on profanity, rather than hate speech. In other cases, the targets of the data fall decidedly within the realm of hate speech, but the data set is not publicly available. Warner and Hirschberg (2012), with their not–publicly available data set on anti-Semitism, is one such example.

Beyond the issues mentioned above, not all data sets are practically usable. For instance, while the data set of Ross et al. (2017) is both publicly available and aimed at hate speech, in the form of racism related to the 2016 refugee crisis, the data set consists of only 541 tweets. This is too little to be practically useful in machine learning. Furthermore, this data set is in German, making it ill suited for combination with other data sets. The data set compiled by Fortuna (2017) has some of the same issues. While the size of the data set is an order of magnitude larger than that of Ross et al. (2017), it is still not particularly big. Beyond this, the data set is in Portuguese, and so cannot easily be combined with other data

Paper	Labels	Size	Source
Waseem and Hovy (2016)	Sexism, Racism	16,914	Twitter
Waseem (2016)	Sexism, Racism	6,909	Twitter
Davidson et al. (2017)	Hate speech, Offensive	24,802	Twitter
Sood et al. (2012)	Profanity	6,354	Yahoo! Buzz
Golbeck et al. (2017)	Hate speech	35,000	Twitter
Ross et al. (2017)	Hate speech	541	Twitter
Fortuna (2017)	Hate speech	5668	Twitter
Pavlopoulos et al. (2017a)	Inappropriate	1.6M	Gazzetta
Warner and Hirschberg (2012)	Anti-Semitic, Hate speech	1,000	Yahoo!, American Jewish Congress

Table 4.1: An overview of data sets used in prior research.

sets.

The largest data set used in research on inappropriate language is the one in Pavlopoulos et al. (2017a,b), with 1.6 million comments from the Greek sports commenting site *Gazzetta*. However, the labels in this data set are based on the censoring of the web page. In other words, the 'positive' labels correspond to the comments the site's moderators found to be inappropriate in some way. As such, there may be a multitude of different reasons for the labelling, including, but not restricted to, hate speech. The data set being in Greek also means that deeper analysis on possible reasons for misclassifications becomes difficult.

While the data set from Davidson et al. (2017) may seem an interesting option, it is not without problems. As noted in Section 3.3, the justification for what label a sample received seems somewhat lacking, relying heavily on the interpretations of the various annotators. Golbeck et al. (2017), on the other hand, used an extensive set of rules when labelling their data. This data set is a very relevant option, with only the truly harassing samples being labelled as hate speech. However, for various reasons, this data set was not used in the research presented in this thesis. Waseem and Hovy (2016), like Golbeck et al. (2017), had a specific, albeit less extensive, set of criteria underlying their labels. These criteria will be described more closely in Section 4.3.

The data set of Waseem and Hovy (2016) was extracted from a larger accumulation of data. This larger collection was also used to create the data set of

Waseem (2016). As such, these sets are fairly similar in the kind of data they contain. Indeed, there is some overlap between them. However, while the Waseem and Hovy (2016) data set contains three kinds of labels, the data set of Waseem (2016) has four kinds. That is, in addition to marking samples as racist, sexist or neither, Waseem (2016) allows labelling samples as both racist *and* sexist. While this may provide a more correct set of labels, it does complicate combining the two data sets. One of the most practical choices for combining them would be to set the samples with the 'both'-label to either sexism or racism. However, what to select for any given sample may not be a straightforward choice. Alternatively, the data set of Waseem and Hovy (2016) could be treated as if it had all four kinds of labels, but with no samples receiving the fourth. Given that the data set was created without a label for cases that were both sexist and racist, the likelihood of there being many such cases, which would then end up being mislabelled, is low. That being said, this would leave the group of samples with the fourth label too small to be practically usable, as only about 1% of the Waseem (2016) data set was given this label. As such, less than 0.5% of the combined data would have this label. Another option is that the samples marked as 'both' could be removed. In all cases, however, there is a risk of samples occurring in both data sets to have different labels in each set. Because of these issues, combined with the fact that such combination would make comparisons to the results of prior research more difficult, the data sets were not combined in this research. Instead, only the one from Waseem and Hovy (2016), the larger of the two, was used.

The descriptions of data sets in this section have been brief, including only a selection of the myriad sets used in prior research. For a more exhaustive overview, the interested reader is referred to Fortuna (2017).

4.2 Twitter and the Twitter API

Twitter[1] is a widely used, online microblogging service that allows its users to post and read short messages, or *tweets*. For most of the lifetime of Twitter, the length of tweets has been restricted to 140 characters, but since 7 November 2017, this limit has been expanded to 280 characters. However, as all the available Twitter data sets on hate speech were collected prior to this expansion, none include tweets of more than 140 characters.

The widespread use of Twitter, with 330 million monthly users creating approximately 500 million tweets a day (Aslam, 2018), and the often colloquial forms of tweets make this a convenient, accessible way of studying aspects of public discourse. With these large amounts of data, there is also, naturally, a not insignificant amount of hate speech. The number of hate speech data sets collected from

[1]https://www.twitter.com

Twitter is an indication of this. The large amounts of data on Twitter, combined with the easy access through the Twitter API, generally makes it a favoured source of research material.

The Twitter API allows programmatic access to the contents of Twitter, both to tweets and to their authors. This can be done in a number of different ways, including retrieving specific tweets, tweets by a specific user, tweets related to a specific hashtag or subject, etc. Through the Twitter API, researchers can easily access data to create data sets. At the same time, the conditions for using the Twitter API[2] prohibits distribution of the contents of tweets. As such, a data set from Twitter is distributed in the form of the identifiers for the relevant tweets. If other researchers want to use the data set, they have to collect the tweets from Twitter themselves. However, users of Twitter can at any time delete their tweets, or even their account. In addition, Twitter may delete tweets they deem in breach of their conditions. If either of these occurs, the tweets in question are no longer accessible through the API. Thus, the entire data set is not necessarily available to new researchers who want to use it.

4.3 The Waseem-Hovy Data Set

The research in this paper was done using the data set of Waseem and Hovy (2016). As mentioned before, the tweets in this data set were labelled as either racist, sexist, or neither. To determine if a given tweet contained hate speech, a set of 11 rules were followed. As specified by Waseem and Hovy (2016, p.89), a tweet is hateful if it:

1. uses a sexist or racial slur.
2. attacks a minority.
3. seeks to silence a minority.
4. criticizes a minority (without a well founded argument).
5. promotes, but does not directly use, hate speech or violent crime.
6. criticizes a minority and uses a straw man argument.
7. blatantly misrepresents truth or seeks to distort views on a minority with unfounded claims.
8. shows support of problematic hash tags. E.g. "#BanIslam", "#whoriental", "#whitegenocide"
9. negatively stereotypes a minority.

[2]https://developer.twitter.com/en/developer-terms/agreement-and-policy

10. defends xenophobia or sexism.
11. contains a screen name that is offensive, as per the previous criteria, the tweet is ambiguous (at best), and the tweet is on a topic that satisfies any of the above criteria.

While some of these rules, such as number 1 and 10, are directed specifically at the categories of hate speech in focus in Waseem and Hovy (2016), most of them apply more generally to any kind of hate speech. As such, they can be used as a sort of 'definition' upon which to build data sets on other flavours of hate speech as well. Furthermore, both rule 1 and rule 10 could easily be expanded to consider other kinds of hate speech as well.

One quirk with these rules is that they generally do not pick up on hate speech directed at a majority. This, however, is more of a formulation issue rather than a complete oversight, as replacing 'minority' with 'group' or something similar would remedy the problem. Of more concern is the fact that the rules may be too far-reaching. Thus, they may include cases that, while some may consider them unsavoury, are not really hate speech. Censoring such cases as hate speech may be somewhat excessive, and even problematic in regards to freedom of speech. As such, these rules and their data set are not an optimal basis for research on hate speech detection, and any future data set made based on these rules should first consider how to avoid such excessiveness. That being said, having a consistent set of clear criteria upon which to set labels is arguably better than an approach that depends heavily on the interpretations of the annotators, as is the case in Ross et al. (2017).

Another issue with the Waseem and Hovy (2016) data set is the representativeness of the data. One aspect of this is the relatively high percentage of hate speech-samples, at about 30%. In the data set of Pavlopoulos et al. (2017a), too, about 30% of the samples were considered inappropriate, and there, the 'positive' label was not restricted to just hate speech,. This also included periods with stricter-than-normal requirements. Burnap and Williams (2014) found that 11% of their tweets, gathered in relation to a particularly hate-inducing event, included offensive or antagonistic content, while Davidson et al. (2017), with a somewhat stricter definition, found 5% of their data to contain hate speech. This means that the propensity of hate speech is higher in the training data than what the system would face in real use. More problematic, however, are the effects of the bootstrapping used to collect the data. For instance, the authors found that tweets related to the Australian TV-show *My Kitchen Rules* (MKR), that is, which used the hashtag '#MKR,' had a high tendency towards sexism. As a consequence, this hashtag became one of the 17 terms used to in the search for gathering data. Thus, a significant portion of the tweets labelled as sexist are related to this show and its participants. Another side of the issue is that the non-hate speech – henceforth

Version	**Neutral**	**Racist**	**Sexist**	**Total**
Original	11,559	1,972	3,383	16,914
Available	10,913	1,924	3,097	15,934

Table 4.2: Size of the classes in the data set, as reported in Waseem and Hovy (2016), and as available at the time of data collection.

called neutral – samples, too, were collected using the same key terms. As such, they are mainly on related issues, and many are of an arguably divisive nature. While this is good for learning on the border cases, the lack of 'normal' tweets may lead to unforeseen results if the end system is applied to arbitrary tweets. Herein lies the problem with the bootstrapping. In collecting certain kinds of tweets, normal discourse was overlooked. Analogously, the aspects of sexism and racism in focus through the key terms may not be representative of how they appear elsewhere. That being said, collecting sufficient amounts of hate speech data without some kind of bootstrapping would require an immense effort. Specifically, labelling the gathered data would quickly become prohibitive. As such, the sexist and racist data should be considered acceptably general. The neutral group would ideally contain more 'normal' samples. However, for comparative purposes, as well as for the sake of reproducibility, no such data were added for the research described here.

The data set of Waseem and Hovy (2016) originally contained 16,914 tweets labelled for racism and sexism. However, due to the tweets only being available through Twitter, 980 of these tweets had been deleted by the time the data were collected. This left a data set of 15,934 samples.

As Table 4.2 shows, most of the deleted tweets were from the neutral group. As this is the largest group, it is also where the impact of deletion is the smallest. The smallest group, on the other hand, is where the loss is the lowest; more than 97% of the racist-labelled tweets were still available at the time of collection. This being the smallest group, it is also where the impact of the loss on training would be greatest, so the small number of deleted tweets from this group is a good thing. The group with the greatest loss relative to size, however, is the group with sexist-labelled tweets. Even here, though, more than 91% of these tweets still remained. In total, the loss constitutes less than 6% of the tweets in the original data set.

5 Architecture

As Section 3.2, on state-of-the-art classifiers, explained, the input forms that have proven best for hate speech identification are word embeddings and character n-grams. In accordance with these findings, the system described in this chapter uses both forms as input. However, the character- and word-based inputs are initially treated separately, in a two-pronged approach. Specifically, the system providing the actual classification of hate speech or non-hate speech consists of three main components. The first two components – the prongs – work in parallel. These operate on the word and character-based inputs, respectively. The last component combines the output of the previous two, and determines the final classification. The high-level architecture of the system is illustrated in Figure 5.1.

In order to avoid the vast complexity of manually implementing all aspects of neural networks, the system described here was implemented using TensorFlow, the deep learning framework described in Section 2.2.1. The preprocessing is done outside of TensorFlow, but all the three main components operate on this platform.

This chapter will first describe the text preprocessing step of the system. Then, each of the system components will be described in turn; first the word-based part, followed by the character-treating component, and then the final classifying component. Finally, the regularisation strategies used in the system will be described.

5.1 Text Preprocessing

While neural networks do not require the kind of elaborate features as are often used in other forms of machine learning, they cannot take pure text as their input. As such, some preprocessing is necessary. Before the actual text processing begins, however, the text samples are divided into mini batches. Generally, each mini batch contains 20 samples. This is, however, not a strict system requirement. For instance, the number of samples in the data set in Section 4.3 is not divisible by 20, so at least one batch would have to have a different size.

Once the partitioning into mini batches is complete, the treatment of the actual text can begin. Each text sample, or tweet, is treated in two disjoint ways; one to create the character representation of the text, the other to create the word

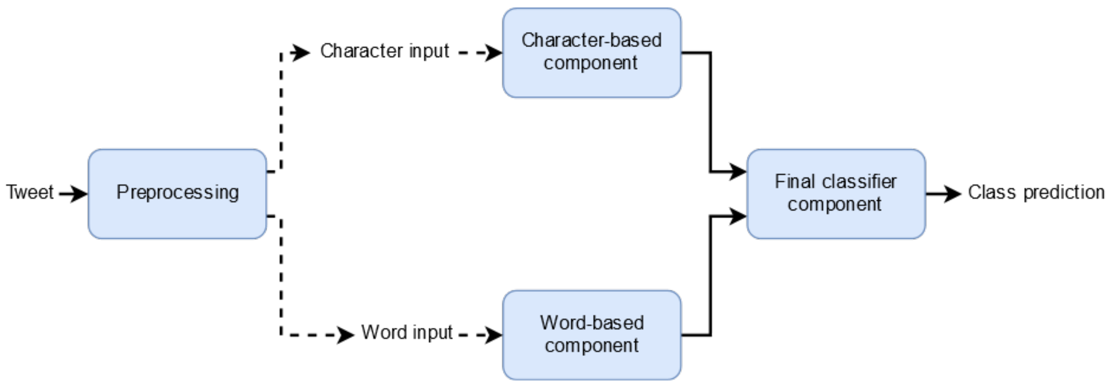

Figure 5.1: Illustration of the high-level architecture of the system.

representation. In both cases, the *tweet-preprocessor*[1] is used to modify the tweets. However, the way this modification is done differs between the two preprocessing routines.

In the character-based preprocessing, each tweet is first cleared of any emoji it contains. This is done using the tweet-preprocessor. The text is then converted to the lowercase representation. Following this, each character, in order, is transformed into a one-hot vector representation, as described in Section 2.3. As the texts in this research are all in English, the vector has one slot for each of the 26 letters in the English alphabet. In addition, because of their distinctive meanings on Twitter, the symbols '#' and '@' have their own slots in the vector. There is another slot for spaces, as well as one for numeric characters. Finally, there is one slot for any character that does not fall into any of the above categories. This means that each character is represented as a one-hot vector of length 31.

While the number of characters varies between the tweets, the neural network requires all samples in a mini batch to be of the same size. This is because the input to the TensorFlow-system, for reasons of efficiency, is in the shape of an array; in this case, the array is 3-dimensional. Specifically, the samples of the mini batch comprise the outermost dimension, and the character vectors the innermost. To accommodate this equal-length requirement, the samples of each mini batch are zero-padded – that is, padded with only zero-valued vectors – to the length of the longest sample of that mini batch. This zero-padding is added to the end of the sample, so that any redundant elements occur *after* the actual information contained in the sample. The post-data placement of the padding is necessary so that the later system components can detect the actual differences in length, and ignore the redundant part.

As for the word-based part of preprocessing, the texts are treated somewhat

[1]https://pypi.python.org/pypi/tweet-preprocessor/0.5.0

differently. The tweet-preprocessor is used to replace emoji, URLs and Twitter-mentions with placeholder terms. Once this is done, hashtags in the tweet are split into single words. This is done by splitting terms at capital letters. Hashtags written all in capitals, however, are not split up. This way, many composite hash tags may carry meaning into the input, not just appear as unknown terms. It is, however, not a foolproof approach; some composite terms may avoid notice, and there is a possibility that some terms are mistakenly divided. That being said, the impact of the latter kind of error is likely to be much smaller than the gain, and detecting only some of the compositions is better than detecting none. Thus, the hashtag split is performed as described.

Once these special cases of a tweet have been treated, the texts are converted to lowercase. Then, punctuation and other symbols are removed. All symbols that are not alphanumeric or whitespace characters are replaced with a space. This is done as replacement rather than complete removal because some characters, such as a slash, often are the only characters dividing distinct words. In addition, the restriction on length of tweets occasionally causes people to cut the spaces after punctuation. As such, replacing the characters with spaces helps distinguish different words that may otherwise have been considered unknown terms.

After removing punctuation and special characters, the tweets are tokenised by splitting on whitespace characters. As a consequence of the replacement of punctuation, the tweets may contain several consecutive whitespaces. However, in the tokenisation process, the result of such cases come in the form of empty strings. After splitting the tweet into terms, empty string-terms are ignored, and each of the remaining words is transformed into its word embedding representation. The word embeddings used here are pretrained on external data sets, so as to avoid an additional source of overfitting due to the relatively small size of the Waseem and Hovy (2016)-data set. In particular, the bootstrapping used to assemble the data set could cause any resulting word embeddings to carry implications that would not hold in a more general case, thus making them more domain dependent. While such implications could, conceivably, improve the results of this research, it would do so on the wrong premises. As such, pretrained embeddings are used. In addition, two different kinds of embeddings are available. One version is a pretrained set of word2vec-embeddings, while the other is a set of GloVe-embeddings; both types were described in Section 2.3.4. The word2vec-embeddings have a dimensionality of 300, while the vectors of the GloVe-embeddings are of size 200. Out of Vocabulary (OOV) words are given a random value of the corresponding dimensionality.

Like with the character-representations of the tweets, the word-representations, once transformed into embeddings, must be padded so that all samples in a batch are of the same length. Again, this zero-padding is appended to the end of the samples. Once this is done, the data are ready to be used as input to the first two components of the system.

5.2 Word Input Component

The part of the system working on the word-based input, that is, on word embeddings, is in the form of a Long Short-Term Memory (LSTM)-network, as described in Section 2.2.3. The nodes in this network are of the kind described by Hochreiter and Schmidhuber (1997).

In addition to this standard LSTM-setup, the architecture allows for the use of a bidirectional LSTM-network. Here, each layer has one set of nodes going through the input in the normal order, and another set running through the input backwards. For each time step, the outputs from these two sets are concatenated. This concatenated value is used as the input to the next layer at the corresponding time step.

While the input to this component is in the form of mini batches, where padding makes sure all samples are of equal length, the system only operates on the parts of a sample that actually have some content. As a consequence, some of the samples will end up having zero-padded values in the output of the last LSTM-layer. In other words, even samples in the same mini batch will have different numbers of time step outputs with any actual content. However, the output of the component must be of constant size even across mini batches, in order for the last classification component to work. This means that the full output state of the last LSTM-layer cannot be used. Instead, the component's output for each sample should be the output state of the LSTM at the last *relevant* time step for that sample. This is extracted by finding the non-padded lengths of the different samples, and collecting the LSTM-output at the time step corresponding to the last element. The sample lengths are also necessary in order to restrict the LSTM's operations to the relevant time steps, to ensure correctness. They are calculated using the sum of the number of time steps for which the input is non-zero.

5.3 Character Input Component

The part of the system working specifically on the character-based input is, in itself, divided into two parts; one convolutional and one recurrent, as described in Sections 2.2.4 and 2.2.2, respectively. The architecture here is inspired by that of Zhou et al. (2015) in how it combines these two elements. Figure 5.2 illustrates a simple configuration of this part of the system. The first part of this component takes the input and performs a 1-dimensional convolution, using multiple filters of

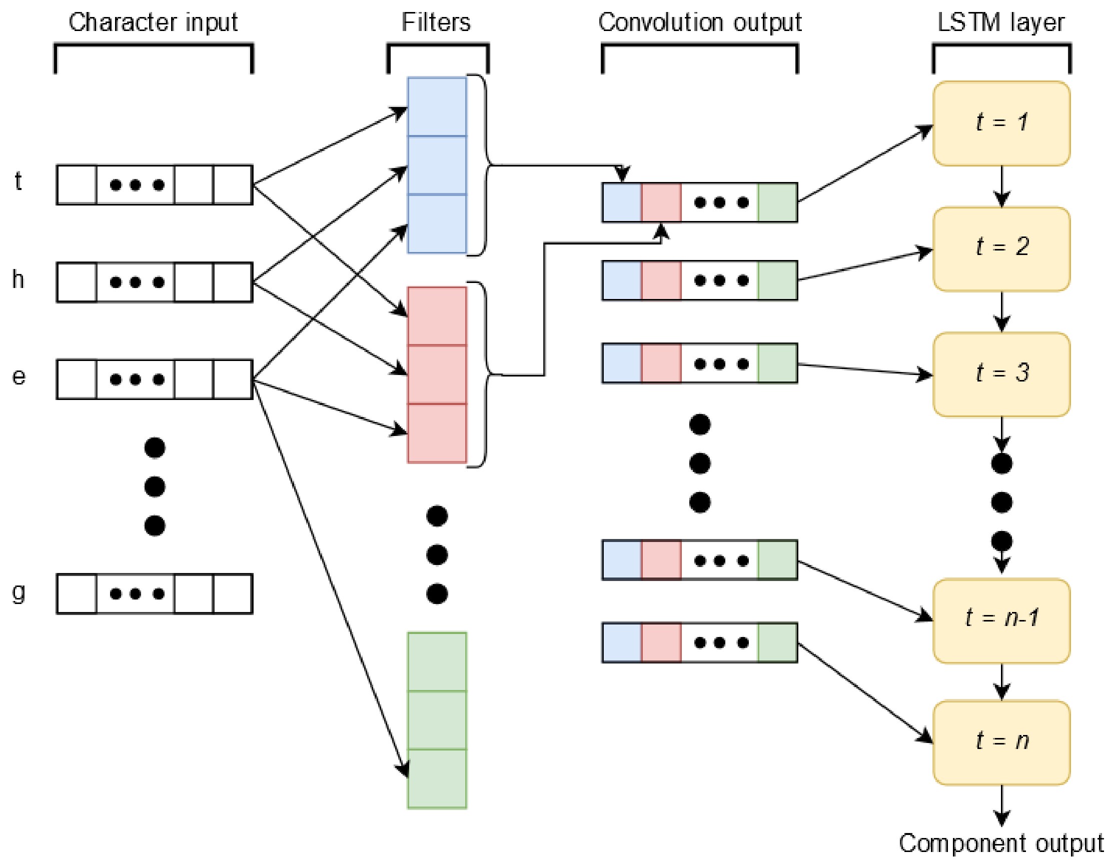

Figure 5.2: Basic architecture of the character-based component. Each filter runs through the entire input, with the results of all filters at any location being combined. These are then fed into the LSTM at the appropriate time step. The LSTM's output at the last time step is the output of the component as a whole.

size n. This essentially means that the input is treated as character n-grams. The output of this convolution is sorted by locations in the input, so that the results of the different filters at any given location appear together. This way, the results of the convolutions are in a form that imitates the time steps of an LSTM. The architecture allows for several layers of convolution before this first part of the network component is finished.

In the second part of the character-based component, the results of the convolution are input to an LSTM. This LSTM works on the same principles as the one described in Section 5.2. In this case, the sample lengths used in the LSTM are calculated from the output of the convolutions. They are then used to extract the output of the component.

The character-based component uses an LSTM completely separate from that of the word-based part due to the highly different lengths of the inputs. While the character-input may include samples of a length up to 140, and, accordingly, approximately the same length as input to the LSTM, the number of words in a tweet is necessarily much lower. The maximum possible number of words in a tweet would be 70, if every word contained one character. This, however, is beyond unlikely, as illustrated by the fact that the average number of words in the Waseem and Hovy (2016)-data set, as used in this research, is 15. In comparison, Waseem and Hovy (2016) found that, excluding spaces, the average number of characters in a tweet exceeded 115, with some small variation between the different classes. Berg and Gopinathan (2017), using a data set of 655,268 tweets, found that an average tweet consisted of 12 words, making up a total of 66 characters. This great difference in length, combined with the difference in input dimensionality, means that any manner of combining the two groups as input to one LSTM becomes problematic and unnatural. Thus they are treated separately, before being combined in the last component of the system.

5.4 Final Classifying Component

Whereas different input samples have varying length, and the above two components thus have to treat irregularity in input size, the final component requires fixed-size inputs. Consequently, as described above, the outputs at the last relevant – that is, last non-zero – time step for each of the first two components are combined and used as input to the last component. As Figure 5.3 shows, this combination is done by merging the two output vectors of each sample, and then feed the result into a fully connected, feed-forward network. Note that the input layer here simply provides data for subsequent layers; there is no activation function applied at this point.

In the output layer, this feed-forward Artificial Neural Network has one node for each possible label. In other words, there is one node for each of the classes of sexist, racist and neutral texts. While the hidden nodes of the network use ReLU (see Section 2.2.1) as their activation function, the output layer uses a linear activation. That is, the weighted sum of a node's inputs is used directly as its output. This output is then run through a softmax layer, returning a probability distribution on which class a sample belongs to, as described in Section 2.2.5.

When training the system, the classification error of a sample is calculated using cross entropy, as described in Section 2.1.4. The gradients of each weight's contributions to these losses are then calculated. After this, the gradients of the entire mini batch are accumulated, and used to update the system's weights according to the Adam optimiser, as described in Section 2.2.6. Note that these updates apply

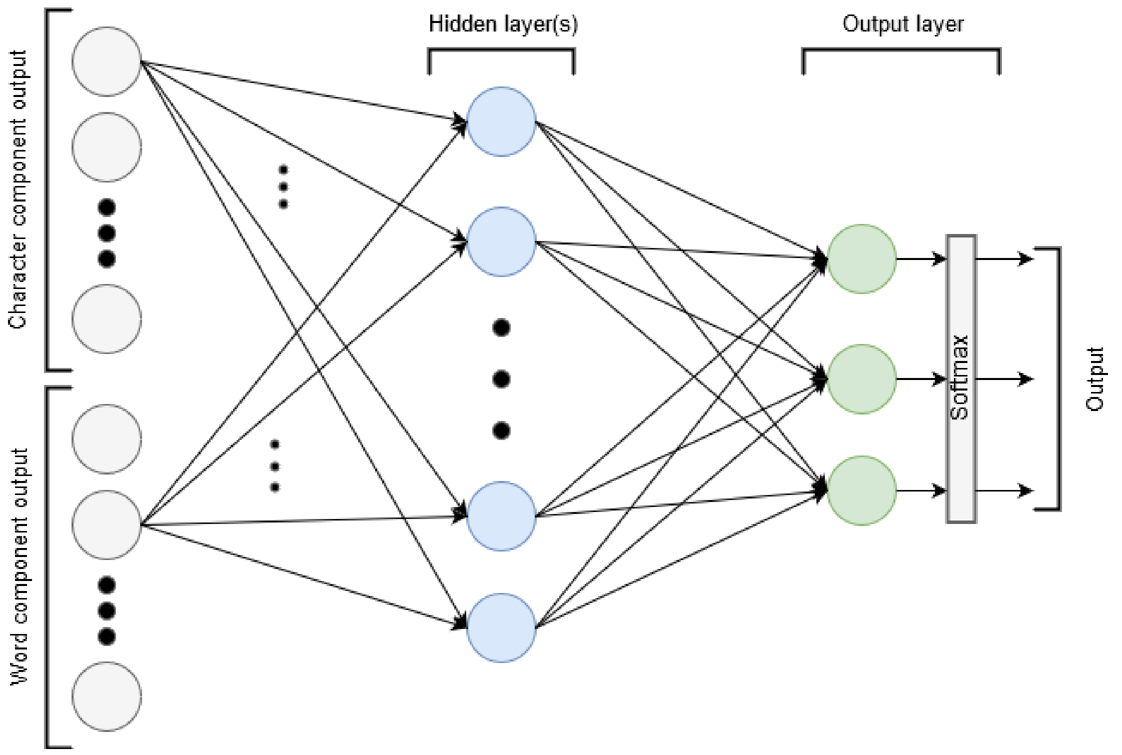

Figure 5.3: Architecture of the final component. The input nodes are the output values of the previous two components, feeding into one or more fully connected hidden layers, before the output is calculated and softmax-ed.

to the entire system, not just the last component.

5.5 System Regularisation

In order to avoid overfitting the network to the training data, some regularisation is necessary. The primary means of regularisation in this system is dropout, as described in Section 2.2.7. The dropout is applied to the dense layers of the final component, as well as the LSTMs of the character- and word-based components. In the LSTMs, the dropout nodes vary from one time step to the next, and no dropout is applied to the states of the LSTM. The dropout is not applied to the CNN-part of the character-based component. However, it is applied to the input of the LSTM-part. In other words, dropout is not applied between convolutional layers, but is applied to the output of the last convolution.

Aside from dropout, the system uses L^2-regularisation (see Section 2.2.7. The L^2-penalty is calculated using all non-bias weights in the system, then added to the cross entropy classification error.

As a final tool against overfitting, the system uses early stopping, so that the training does not continue for too long. Combined, these three regularisers reduce overfitting in the system, thus increasing its general applicability.

6 Experiments and Results

In order to determine the effectiveness of a system, it must be tested through experiments. Furthermore, a system based on Artificial Neural Networks (ANN), described in Section 2.2, also requires rigorous experimentation to find the best configurations. Specifically, these configurations detail the layer sizes and the number of layers.

This chapter presents the experiments used to evaluate the performance of the architecture described in Chapter 5. In so doing, different configurations were explored, in order to find those that worked best. First, Section 6.1 describes the high-level plan of experiments used in the research. Section 6.2 then provides the detailed descriptions of the settings used in the different experiments, before Section 6.3 presents the experimental results.

6.1 Experimental Plan

The experiments performed in this research were centred around discovering configurations of the system described in Chapter 5 that successfully classify hate speech. This was done by varying the layer sizes of the neural networks, as well as varying the *number* of layers used in the different components. In addition, the system was tested using both bidirectional and unidirectional LSTMs. In order to evaluate the effects of the variations consistently, the sizes of the word-based and the character-based components were changed separately. That is, when the sizes of the character-based component were changed, the word-based part was kept constant, and vice versa. This was done so that the best configuration of each component could be found independently, reducing the number of configurations to explore. As for variations in the number of layers, these were, for similar reasons, also made independently by component. Furthermore, in the character-based component, the number of convolutional and LSTM layers were changed separately. In the experiments with changes to the convolution, variations in the length of the convolutional filters were also made. These experiments aimed at solving Research Question 2 from Section 1.2, on whether a CNN-LSTM-combination would be effective in classifying hate speech. Specifically, they tried to find an effective configuration for such a system.

In addition to the the experiments described above, the system was tested using only the character-based input. In other words, the word-based component was excluded from the system. This was done in order to evaluate the effectiveness of the CNN-LSTM combination on the character input. As such, these experiments related to Research Question 1, in that they investigated the use of pure character n-gram input, and, to a lesser degree, Research Question 2, studying the CNN-LSTM-combination of the system. For comparative purposes, this group of experiments also included a version testing the use of only the word-based input, disabling the character-based component.

Beyond varying the setup configurations of the network itself, the effects of using different word embeddings, as described in Section 2.3.4, were explored. As mentioned in Section 5.1, this research applied pretrained word embeddings in order to remove one source of overfitting. The majority of the experiments were done using the 300-dimensional word2vec-embeddings.[1] These were trained on a selection of the Google News data set, with approximately 100 billion words. In order to compare results from different embeddings, however, the best configurations discovered in the above sets of experiments were also tested using pretrained GloVe-embeddings.[2] These embeddings were trained specifically on Twitter, using 2 billion tweets. The embedding data set contains embeddings of several different dimensionalities. Of these, the highest, at 200 dimensions, was used. The experiments investigating the effects of what word embeddings are used were connected to Research Question 1 of Section 1.2, in that they looked at what kinds of input work best for hate speech identification. However, they also had practical impact on Research Question 2.

6.2 Experimental Setup

In the experiments, the hyperparameters of the Adam optimiser had the values suggested by Kingma and Ba (2014), as these values have generally been found to produce good results (Goodfellow et al., 2016, Chapter 8). The probability of 'switching off' nodes due to dropout was set to 0.5, in accordance with the suggestions of Srivastava et al. (2014). Furthermore, all experiments were run using 10-fold cross validation, and with mini batches of size 20. In the cross validation, stratified folds were used. In other words, the data were divided so that each fold had approximately the same distribution between the different classes. This was done to ensure that none of the folds would end up with no samples of a class, thus giving a misleading validation result.

Aside from the parameters mentioned above, there is a coefficient restricting the impact of the L^2-regularisation. In the first group of experiments, this coefficient

[1]Embeddings available at https://code.google.com/archive/p/word2vec/

[2]Embeddings available at https://nlp.stanford.edu/projects/glove/

was given the value 0.001, as this value is commonly used. However, after the initial round of experiments were run, it was discovered that smaller values produced better results. As such, later experiments used a value of 0.0002 for this coefficient. This transition will be explicitly specified in the following.

Experiment Group 1

The first group of experiments revolved around the configuration of the system. As explained in Section 6.1, this was done by making variations to the components separately. As such, the unmodified part conformed to a default setup. In the first half of these experiments, each kind of nodes had one layer. In other words, the character-based component had one convolutional layer, followed by one LSTM layer. Similarly, the word-based component had one LSTM layer, and the dense, feed-forward part had one hidden layer. In the first experiment, the entire system had the default configuration. Here, the word-based component had one layer of 150 LSTM nodes. This dimensionality was chosen because it reduces the number of dimensions from the word embeddings, going down to half the size in the case of word2vec, without decimating the information carried through. In the character-based part of the system, the convolutional layer had 64 filters of length 3. The filter length here denotes the n in the character n-grams. This was set to 3 as such n-grams have proven useful in prior work (Waseem and Hovy, 2016; Mehdad and Tetreault, 2016). The choice of 64 convolution filters was made because 64 is a power of two approximately twice the length of the character vectors. As such, it is significantly greater than the character vector size, while at the same time smaller than the size of each filter – that is, 3×31. The LSTM layer of the character-based component had 100 nodes. This number was chosen, in part, to balance the impact of the word- and character-based components on the final classifier component. At the same time, the number should not be too much higher than the dimensionality of the convolution output, which is the same as the number of filters used in the convolution. With these settings for the first two components, the input to the final component contained 250 elements; 150 from the word-based part, and 100 from the character-based one. In the other end of the component, there were three output nodes; one for each class. Some rules of thumb suggest that the number of nodes in a hidden layer should be somewhere between the number of input nodes and the number of output nodes (Basheer and Hajmeer, 2000). While such rules may not be entirely reliable, the hidden layer size was set to 120; near the average of the input and output sizes. Throughout the experiments, the size of this hidden layer was given a value of approximately half the dense component input size.

In the first experiment, the system configuration described above, from here on called the default setup, was used. In addition, the bidirectional version of this default configuration was tested. Here, each direction of the LSTMs had

Experiment	**Conv. dim.**	**Char. dim.**	**Word dim.**	**Dense dim.**
Default	64×3	100	150	120
100 Filters	100×3	100	150	120
Char. dim. 50	50×3	50	150	100
Char. dims. 512, 256	512×3	256	150	200
Word dim. 50	64×3	100	50	75
Word dim. 100	64×3	100	100	100
Word dim. 200	64×3	100	200	150
Word dim. 250	64×3	100	250	175

Table 6.1: Experimental settings, in dimensionality, for experiment group 1. Char. dim. and Word dim. represent the sizes of the LSTM layers.

the dimensionality described above, thus giving the input to the final component twice the number of dimensions in the unidirectional case. Consequently, the hidden layer dimensionality was doubled. After running these experiments, the system was tested with varying configurations in the character-based component, as shown in Table 6.1. Specifically, the system was tested using 100 convolutional filters along with the 100 dimensions of the character LSTM. Then, the size was cut down to 50 for both number of filters and LSTM layer size, in order to see if a smaller network may perform better. Next, the system was tested with a much higher character component dimensionality, using 512 filters and an LSTM layer size of 256. In all cases, the bidirectional version of the setup was tested as well. However, as the bidirectional versions have essentially the same settings as the unidirectional equivalents – dimensionality of the dense hidden layer excepted – they are not listed in Table 6.1.

After exploring the effects of various configurations in the character-based component, experiments were performed where the dimensionality of the word-based component was changed. In these experiments, the character-based part of the system had the default configuration described above. The system was then, as Table 6.1 shows, run with word-LSTM sizes of 50, 100, 200 and 250, respectively. The corresponding configuration with a layer size of 150 had already been tested in the initial experiment.

Experiment Group 2

In the next group of experiments, variations were made to the convolutional part of the character-based component. Specifically, the system performance with filter length 4 was tested; then, an extra layer of convolution was added, with combina-

Experiment	**Conv. layer 1**	**Conv. layer 2**
64 filters; length 4	64×4	—
64, 64 filters; lengths 3, 3	64×3	64×3
64, 128 filters; lengths 3, 3	64×3	128×3
128, 64 filters; lengths 3, 3	128×3	64×3
64, 64 filters; lengths 3, 4	64×3	64×4
64, 128 filters; lengths 3, 4	64×3	128×4
128, 64 filters; lengths 3, 4	128×3	64×4
64, 64 filters; lengths 4, 3	64×4	64×3
64, 128 filters; lengths 4, 3	64×4	128×3
128, 64 filters; lengths 4, 3	128×4	64×3
64, 64 filters; lengths 4, 4	64×4	64×4
64, 128 filters; lengths 4, 4	64×4	128×4
128, 64 filters; lengths 4, 4	128×4	64×4

Table 6.2: Experimental settings, in the convolution, for experiment group 2. Notation: *#filters* × *filter length.*

tions of length 3- and length 4-filters being used. In addition, the change in value of the coefficient restraining the impact of the L^2-regularisation occurred at this point, so these experiments, and all the following, used a coefficient value of 0.0002. The rest of the system had the setup found to be best in the above experiments. That is, as described in Section 6.3 and Table 6.6, the rest of the system maintained the default setup. The configurations used in the convolutional part of the system are shown in Table 6.2. In the first of these experiments, the convolutional layer used filters of length 4. The remainder of this experiment group performed convolution in two layers. First, both of these layers used filters of size 3. Setups with both layers having the same number of filters, and with the number of filters increasing or decreasing by layer, were all tested. Specifically, the standard number of filters in these experiments was 64, with the layers using a higher number having 128 filters. Next, the same three experiments were performed where the first convolutional layer used filters of length 3, and the second layer, length 4. Then, the order was reversed, with the first layer filters having length 4 and the second layer length 3. Finally, the three experiments were run with both layers using filters of length 4. In this group, only unidirectional LSTMs were used, not bidirectional. This was because the focus of these experiments were on the effects of the convolution, not the LSTMs.

Experiments	**Char. LSTM**	**Word LSTM**	**Conv. setup**
Two char.-LSTM layers	2×100	1×150	64×3
Two word-LSTM layers	1×100	2×150	64×3
Two conv layers $(64 \times 3, 64 \times 3)$, and two char.-LSTM layers	2×100	1×150	64×3, 64×3
Two conv layers $(64 \times 4, 64 \times 3)$, and two char.-LSTM layers	2×100	1×150	64×4, 64×3
Two LSTM layers each	2×100	2×150	64×3

Table 6.3: Experimental settings for experiment group 3.

Experiment Group 3

In addition to the experiments on multilayer convolution, configurations using two-layer LSTMs were tested. In these experiments, the system was evaluated using two same-sized layers in the LSTM part of the word- and character-based components, respectively. In the first of these experiments, as shown in Table 6.3, the character-based component's LSTM was given two layers of size 100, with the rest of the system having the settings of the default configuration. The second experiment tested having two 150-dimensional LSTM layers in the word-based component, reverting the character-based component back to the default. Further, the system was tested with both two convolutional layers *and* two LSTM layers in the character-based part. In this case, two settings of the convolutional section were tried. The first was a 'default-like' version, with the two convolutional layers each having 64 filters, all of length 3. The second was the version found to perform best in the second group of experiments described above. As Section 6.3 will show, this configuration had two layers of 64 convolutional filters, with the first layer's filters having length 4, and the second layer's being of length 3. These two settings were combined with the two 100-dimensional LSTM layers as the third and fourth in this group of experiments. As the last experiment in this group, the default configuration was expanded to two LSTM layers in each of the system components holding LSTMs. In other words, this experiment had two LSTM layers in the character-based part and two in the word-based part of the system.

Experiment Group 4

Next, the word-based component was disabled, and the performance of just using the character input was tested. First, the default setup was used, with one convolutional layer and one LSTM layer. The word-based LSTM was removed, and the dense layer was reduced to 50 nodes; half the number of LSTM nodes used.

Experiments	**Conv. setup**	**Word dim.**	**Dense dim.**
Default, char. only	64×3	—	50
Two conv. layers, char. only	64×4, 64×3	—	50
Default, word only	—	150	75

Table 6.4: Experimental settings for experiment group 4.

Experiments	**Conv. setup**	**Char. dim.**	**Word dim.**
Default, GloVe	64×3	100	150
Two conv. layers, GloVe	64×4, 64×3	100	150

Table 6.5: Experimental settings for experiment group 5. Again, char. dim. and word dim. denote the LSTM layer sizes.

After this experiment was done, the equivalent was done using the best-performing configuration in the above experiments, namely where the system had two convolutional layers of 64 filters each, with filters of length 4 in the first layer, and of length 3 in the second. For comparative purposes, the system was then tested using just the word-based input. Here, too, the default setup was used as basis, meaning the configuration of this experiment used one LSTM layer of size 150. The main settings of this group are shown in Table 6.4.

Experiment Group 5

In the last group of experiments, the optimal configurations found through all of the above experiments were used to compare the performance of using the two different forms of pretrained word embeddings mentioned in Section 6.1: word2vec and GloVe. In other words, as shown in Table 6.5, the word embedding types were tested using the configuration with two convolutional layers with 64 filters and filter lengths 4 and 3, respectively. The character-part LSTM had one layer of size 100, while the word-based LSTM had one layer of size 150. The fully connected hidden layer of the final component, as described in Section 5.4, had 120 nodes. In addition, the two types of word embeddings were compared using the default setup, as that configuration provided the primary basis of comparison throughout this research.

6.3 Experimental Results

In this section, the main results of the experiments described above will be presented. The results are in the form of system performance in the respective experiments. This performance is described using the measures *precision*, *recall* and F_1, as these are the main measures used by Waseem and Hovy (2016), and in other papers reporting on the same data set. In terms of True Positives (TP), False Positives (FP) and False Negatives (FN), for a given class, C, these measures can be defined as:

- Precision: Fraction of samples classified as C that also have the label C. More formally: $Precision = \frac{TP}{TP+FP}$
- Recall: Fraction of samples with a C-label that are also classified as C. More formally: $Recall = \frac{TP}{TP+FN}$
- F_1: Harmonic mean of precision and recall. Formally: $F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$

There are two ways of calculating the precision and recall on a multiclass data set; *micro average* and *macro average*. Micro average, in this case, corresponds to the accuracy of the system, reporting the fraction of correctly classified samples. In other words, precision and recall are equal when micro averaging over multiple classes in a data set. It is primarily suited for cases where the distribution between classes is uneven, and the results of the larger class are the main point of interest. In the data set from Waseem and Hovy (2016), there is significant difference in the sizes of the three classes. However, the main interest in the experiments described in this chapter lies in the performance on the smaller classes; that is, in the ability to detect hate speech. As such, the results are reported using macro averaging. This means that the precision and recall are calculated for each class independently, before they are averaged to create the overall precision and recall. Macro averaging is primarily used for cases where the class labels are evenly distributed, though it can also work to increase the impact of the smaller classes in an unevenly distributed case, as is the case here. It is worth noting that the performance reported in this section generally would seem better if calculated using micro average, as the system had a higher performance on the 'Neutral' class. However, as the main focus of this research is on the other two classes, values are reported using macro averages.

The first group of experiments, as described in Section 6.2, tested different dimensionalities in the various layers of the system. The results of these experiments are shown in Table 6.6, for both the uni- and bidirectional versions of the configurations. Note that the reported results are averaged over the 10-fold cross validation. As the table shows, the default setup worked best in terms of both

Experiment Setup	*Unidirectional*			*Bidirectional*		
	Precision	**Recall**	**F$_1$**	**Precision**	**Recall**	**F$_1$**
Default	79.12%	**75.87%**	**77.46**	79.00%	**75.97%**	**77.46**
100 Filters	79.06%	74.90%	76.93	79.08%	75.18%	77.08
Char. dim. 50	79.42%	74.94%	77.11	79.18%	75.53%	77.31
Char. dim. 512,256	79.59%	75.24%	77.35	**79.39%**	74.87%	77.06
Word dim. 50	**79.87%**	74.58%	77.13	79.14%	75.18%	77.11
Word dim. 100	79.30%	74.67%	76.92	79.14%	75.04%	77.04
Word dim. 200	79.10%	75.38%	77.20	78.81%	75.67%	77.21
Word dim. 250	79.24%	74.67%	76.89	79.18%	75.12%	77.10

Table 6.6: Performance of the system in the first group of experiments.

recall and F$_1$-score. Several other configurations had better precision, such as the version where the word-based, unidirectional LSTM had a layer size of 50. However, the corresponding recall values were comparatively lower than in the default setup. This applies to both the unidirectional and the bidirectional versions of the experiments.

As mentioned in Section 6.2, the remaining experiment groups, that is, groups 2 through 5, used a different value for the coefficient controlling L^2-regularisation. As such, all results reported from here on out were found using this new value of 0.0002. Table 6.7 thus shows a different performance for the default setup than that included in Table 6.6. In addition to this updated performance of the default setup, Table 6.7 holds the results of the second group of experiments, concerning variations in the convolutional part of the character-based component.

The default setup, as Table 6.7 shows, still has one of the highest scores on recall and, to a slightly lesser degree, F$_1$. Indeed, only the best-performing setup in this group of experiments had a higher recall. This configuration, with two layers of 64 convolutional filters where the first layer's filters were of length 4, and the second layer's of length 3, had a substantially better performance than the rest of the setups. The recall of this system was 0.5 percentage points higher than that of the second best, i.e. the default setup. Furthermore, it had the second highest score on precision, resulting in a F$_1$-score of 79.10, compared to the second highest at 78.58.

Table 6.8 shows the results of the third group of experiments, on varying the number of LSTM layers in the system. The table also includes the performance of the default setup, as shown in Table 6.7, for comparative purposes. The experiment using two unidirectional LSTM layers in the character-based component of the default system setup and the one based on the optimal configuration from

Experiment setup	**Precision**	**Recall**	**F$_1$**
Default setup	79.50%	77.33%	78.40
64 filters; length 4	80.53%	76.03%	78.22
64, 64 filters; lengths 3, 3	80.45%	76.18%	78.26
64, 128 filters; lengths 3, 3	79.88%	76.58%	78.19
128, 64 filters; lengths 3, 3	79.40%	76.75%	78.05
64, 64 filters; lengths 3, 4	80.01%	76.93%	78.44
64, 128 filters; lengths 3, 4	79.71%	76.20%	77.92
128, 64 filters; lengths 3, 4	80.00%	76.07%	77.98
64, 64 filters; lengths 4, 3	80.51%	**77.73%**	**79.10**
64, 128 filters; lengths 4, 3	80.35%	76.88%	78.58
128, 64 filters; lengths 4, 3	80.48%	76.12 %	78.24
64, 64 filters; lengths 4, 4	**80.57%**	76.34%	78.40
64, 128 filters; lengths 4, 4	79.72%	76.89%	78.28
128, 64 filters; lengths 4, 4	79.55%	76.84%	78.17

Table 6.7: Performance of the experiments on variation of the convolutional segment of the character-based component. The numbers in the setup descriptions denote the number of filters at each consecutive layer, along with their corresponding filter lengths.

the previous group of experiments – that is, with filters of length 4 in the first of two convolutional layers – showed a marked increase in precision compared to the default setup. However, the recall in these experiments was significantly weaker than in the default case, decreasing with 0.97 percentage points or more. This resulted in F$_1$-scores that, despite the higher precision, were lower than the performance of the default system setup. Similar results were found for the experiment using two LSTM layers in the word-based part of the system, as well as in the bidirectional versions of said setup and the equivalent two-character-LSTM setup. In these cases, however, the increase in precision was less marked than in the first two experiments. The experiment using two convolutional layers with all filters at length 3, and the one using two LSTM layers in each of the two system components, both had lower precision than the default setup. Because of this, these two configurations had the two lowest F$_1$-scores in this experiment group.

In addition to the variations to the system tested above, experiments were performed where the word-based component was disabled completely, using just the character input. The results of these experiments are shown in Table 6.9. The table also holds the performance of the corresponding system *with* the word input, for ease of comparison. Interestingly, both of the character-only systems outperformed

Experiment setup	**Precision**	**Recall**	**F$_1$**
Default setup	79.50%	**77.33%**	**78.40**
Two character-LSTM layers	80.21%	76.20%	78.15
Two character-LSTM layers, bidirectional	79.73%	76.59%	78.13
Two word-LSTM layers	79.61%	76.31%	77.92
Two word-LSTM layers, bidirectional	79.69%	76.38%	78.00
Two convolutional layers ($64 \times 3, 64 \times 3$) and two character-LSTM layers	79.39%	76.09%	77.71
Two convolutional layers ($64 \times 4, 64 \times 3$) and two character-LSTM layers	**80.29%**	76.36%	78.27
Two LSTM layers each	79.40%	76.38%	77.86

Table 6.8: Results of the experiments on use of multiple LSTM layers. Performance of the default setup are included for comparative purposes.

Experiment setup	**Precision**	**Recall**	**F$_1$**
Default setup	79.50%	77.33%	78.40
Default, characters only	**81.38%**	77.18%	**79.23**
Two conv. layers ($64 \times 4, 64 \times 3$)	80.51%	77.73%	79.10
Two conv. layers ($64 \times 4, 64 \times 3$), char. only	80.23%	**77.84%**	79.01
Default, words only	79.99%	77.07%	78.50

Table 6.9: Results of the experiments using only characters as input. Results of the corresponding configurations with word input is included for comparative purposes.

the default system setup, which has generally performed well. Furthermore, the characters-only version of the default setup showed the highest precision of all the experiments in this research. This, combined with a fairly high recall, gave this experiment the highest F$_1$-score in the group. As for the characters-only version of the configuration with two convolutional layers, the recall was higher than in the version including word-based input. However, at the same time, the precision was lower, resulting in a slightly worse F$_1$-score. Notably, it still outperformed the word-inclusive default setup on all measures. Table 6.9 also includes the performance of the default setup with the character-based component disabled. While this configuration was outperformed by the character-only systems, it, too, performed better than the default setup using all inputs.

The last experiments in this research examined the effects of using GloVe-embeddings rather than word2vec-embeddings as input to the word-based system component. These results, with the corresponding word2vec-performance, are shown in Table 6.10. In terms of F_1-score, both of the tested configurations improved when changing to GloVe. The default setup improved on all measures, though the improvement in precision was very slight. In the configuration with two convolutional layers, the precision got worse when changing to GloVe-embeddings. However, the recall of this setup using GloVe was the highest recorded throughout this research, outperforming the second best – from the same configuration with the word-based component disabled – with more than 0.4 percentage points. In addition, the precision, while lower than the equivalent word2vec-performance, was still acceptably high. As such, the resulting F_1-score was 79.24, which is higher than any other configuration in these experiments.

Experiment setup	**Precision**	**Recall**	$\mathbf{F_1}$
Default setup, word2vec	79.50%	77.33%	78.40
Default setup, GloVe	79.52%	77.71%	78.60
Two conv. layers ($64 \times 4, 64 \times 3$), word2vec	**80.51%**	77.73%	79.10
Two conv. layers ($64 \times 4, 64 \times 3$), GloVe	80.22%	**78.28%**	**79.24**

Table 6.10: Results of the experiments using GloVe-embeddings as input. The corresponding word2vec-results are included for comparison.

7 Evaluation and Discussion

Chapter 6 presented the experiments performed in this research and their results. This chapter will first provide further evaluation and interpretation of those results, in Section 7.1. Then, in Section 7.2, these findings will be discussed in relation to the Research Questions of Section 1.2.

7.1 Evaluation

Throughout the experiments described in Chapter 6, the results had some consistencies. Notably, as can be seen from the various tables in Section 6.3, the recall values of all system configurations were lower than the corresponding precision. Furthermore, while not visible in said tables, the recall had much greater variations between the different classes. This is illustrated by Table 7.1, which shows the precision and recall of the three classes, as achieved by the default system setup with the coefficient controlling L^2-regularisation set to 0.0002. In other words, this table shows the details of the default setup-performance introduced in Table 6.7. While there was some variation between the different experiments, the general patterns found in Table 7.1 were the same throughout the research. Specifically, all the setups had the best performance on the recall of neutral samples, and the worst on sexist recall. The recall of sexist samples was also where the main difference from the change in value of the L^2-coefficient, as described in Section 6.2, occurred. Using the original value of this coefficient, the recall on sexist samples, averaged over 10-fold validation, was mostly in the range 53–58%, whereas with the updated coefficient value, the averages were mainly in the range 60–65%. In both cases, the difference between two single runs could be close to 20%. However, despite this large range, there were only a very few single runs in all of the experiments where the sexist recall was not the lowest. As mentioned, the cross validation average was always lowest for sexist recall. Thus, this was also the main hindrance in achieving high F_1-scores.

Whereas the recall on sexist samples saw great fluctuations in performance, the recall of neutral samples, which generally had the highest score of the various classes and measures, was more stable. The cross validation averages were in the range 90–91%, and this was also where the large majority of single runs scored. Occasionally, the score would be outside of this range, but still between 87% and

Measure	**Neutral**	**Sexist**	**Racist**	**Average**
Precision	86.81%	77.52%	74.15%	79.50%
Recall	90.04%	66.00%	75.94%	77.33%

Table 7.1: Precision and recall of the default system setup by class.

		Actual class		
		Neutral	**Sexist**	**Racist**
Predicted	**Neutral**	984	103	42
	Sexist	60	206	1
	Racist	47	1	149

Table 7.2: Confusion matrix of a single run with the default system setup.

93%. In general, the performance on neutral samples was the most stable; the precision on neutral samples, while not quite as stable as the recall, did not display large variations.

The remaining three groups, namely the precision and recall of racist samples and the precision of sexist samples, all had a performance mainly in the 70s. In regards to the precision of sexist and racist samples, the performance on the sexist ones was mainly higher than on the racist ones. At the same time, they tended to display the opposite variations, so that when one had a better performance than usual, the other performed worse. As such, the overall precision did not fluctuate quite as much as the variations of the individual measures would indicate. Table 7.2, which shows the confusion matrix of a single run by the system using the default setup, displays a typical distribution of correct and incorrect classifications.

In essence, the tendencies described above mean that the system had the most trouble recognising sexist hate speech. There was less trouble with false positives, though the fraction of these, at 20–30% for the hate speech classes, was not insignificant. A noteworthy point is that virtually all misclassifications were, in some way, related to the neutral class; they were either neutral samples classified as hate speech, or sexist or racist samples classified as neutral. As in Table 7.2, there were very few cases of misclassifications between the hate speech categories, at only 0–3% of their number; in most system runs, this fraction was close to 1%. Over all samples in the classifications, this makes up less than 0.5% of the samples, with even the worst instances having no more than 0.7% misclassifications between the two hate speech classes. In other words, the difficulties of the system lie in the hate speech/non-hate speech distinction; it displays no problems with mixing the two hate speech categories.

In the following, particularities in the results of the various experiments in Chapter 6 will be discussed.

Uni-/Bidirectional LSTMs

The results found in Table 6.6, from the first group of experiments, show that there was no consistent improvement from using bidirectional LSTMs over unidirectional ones. Most of the configurations had some improvement in recall. At the same time, all of them scored lower on precision. This lead to varying effects on the F_1-scores of the different setups, depending on which of precision and recall experienced the greatest changes. Some configurations, like the one using 50 convolutional filters and a character-LSTM layer size of 50, had a marked improvement in terms of F_1-score. Others, most notably the version with 512 convolutional filters and a character-LSTM size of 256, got overall worse when introducing bidirectional LSTMs instead of unidirectional ones. Yet other setups experienced almost no changes in the F_1-score, such as the default setup, where the bidirectional F_1-score was identical to that of the unidirectional case.

The small variations between unidirectional and bidirectional performance should not be unexpected. As described in Chapter 5, the system works by extracting the LSTM output from the last time step of each sample. This is also the first time step for the reverse part of the bidirectional LSTMs. With only one-layer LSTMs, this means that only the input from that one time step has any influence on the end result. As such, the backwards LSTM has very little impact. Furthermore, there is very little foundation for training the backwards LSTM, with the actual recurrent parts being left completely untrained. This is because only the first input is used in the later parts of the system, and so only this one time step contributes to the error. Thus, there are no weight updates between time steps, and the recurrence is ignored. In effect, using bidirectional, one-layer LSTMs in this system greatly increases the number of weights to be learned, but without really making use of them, and without providing proper training.

The bidirectional experiments shown in Table 6.8, on the other hand, have at least one layer of LSTMs where the backwards direction is fully utilised. Here, the outputs of both the forwards and backwards part of the first layer are used as input to the corresponding time steps of the second layer. Notably, the backward output is included in the input to the forward part of the next layer, and vice versa. Yet, there is still no consistent change from uni- to bidirectional LSTMs. The version with two LSTM layers in the word-based system component shows a slight improvement in all measures. However, the configuration with two LSTM layers in the character-based component, while displaying an increase in recall at almost 0.4 percentage points, had a marked decrease in precision, resulting in a slightly lower F_1-score than the unidirectional version. A possible explanation of these different reactions will be discussed further below.

Convolutional Variations

In the second experiment group described in Section 6.2, investigating the impact of variations to the convolutional part of the character-based component, some interesting patterns were found. As Table 6.7 shows, for any given combination of feature lengths, the system with 64 filters in each of the two convolutional layers outperformed those with more filters. Indeed, ever the weakest of these 64×64-configurations, with all filters at length 3, was only outperformed by two of the larger setups, regardless of filter lengths. This confirms the findings in Table 6.6, that increasing the number of filters in a convolution does not improve the system performance, but rather reduces it.

Table 6.7 also shows that, for any given combination on number of filters, the version with length 4 filters in the first layer and length 3 in the second outperformed the others. For most of the combinations, the version with length 4 on all filters placed second. This suggests that character 4-grams are more effective than character 3-grams in terms of hate speech detection. At the same time, the configuration using only one convolutional layer, with the filter length set to 4, performed worse than the default setup. This in spite of the second highest precision in the entire group of experiments, as this configuration also had the lowest recall. As such, it seems like character 4-grams are more effective only when they go through at least two layers of convolution.

LSTM Layer Variations

In the experiments varying the number of LSTM layers used, shown in Table 6.8, all variations were, as mentioned in Section 6.3, found to perform worse than the default, single-layered configuration. Furthermore, every configuration performed worse than the corresponding one-layered version – including the configuration with two convolutional layers with filters of length 3, which already had a relatively low performance. One possible explanation for this is that adding an additional LSTM layer significantly increases the number of weights to be learned in the system. A consequence of increasing the number of weights is that the system starts overfitting more easily. While the system regularisation, described in Section 5.5, counteracts overfitting, it does so imperfectly. The increased overfitting could thus lead to reduced generality, resulting in lower performance.

Single Input Type

As mentioned in Section 6.3 and shown in Table 6.9, both versions of the default setup using just one kind of input outperformed the version using both character and word input. This suggests that something in the combination of the character- and word-based components might hinder the workings of the system. A possible explanation for this issue was given by Founta et al. (2018). As described in Section 3.2, they, too, had a deep learning system where there were two components running in parallel on separate inputs, only to be combined at a later point. In training this system, they found that the two components had different convergence rates, leading to one part dominating the training, not letting the other part reach its potential. In addition, they noted the risk that training the two components together could lead to unintended and unpredictable interactions between the two parts, due to modifying the weights of both at the same time. In the system described in Chapter 5, the two parallel components are likely to have quite different convergence rates. This is because of the highly different nature of the input types used, with one being a set of one-hot vectors and the other having the more implicitly meaningful representation of word embeddings, as described in Section 2.3.4. As such, it is quite likely that different convergence rates hampered the training of the system, reducing the resulting performance.

This issue may also have affected the experiments concerning variations in the layer sizes and the number of layers, as changes in the size of a system component will change its rate of convergence. For instance, this is a possible explanation for the difference between the uni- and bidirectional versions of the system using two LSTM layers, as shown in Table 6.8. The word-component, using highly informative word embeddings, will essentially have a higher convergence rate than the character-based component; apart from its far less informative one-hot vector input, the latter component will, in the default configuration, have more weights to learn than the former. When using two word-LSTM layers, the difference in number of weights is altered. In changing to bidirectional LSTMs, the complexity of the word-based component increases much more than that of the one-layered, character-based one, as a consequence of the different usefulness of LSTM layers described further above. As such, the rate of convergence of the word-based part is likely to slow down, allowing the two components to be trained more on equal terms. In the version with two LSTM layers in the *character*-based component, on the other hand, the imbalance in complexity worsens when changing to bidirectional LSTMs. This coincides with the performance dropping, though only slightly. These examples indicate that different convergence rates may indeed impact the system performance. However, there is not enough information to say definitely that this is the cause of the changes in performance; it is merely a possible explanation.

Embedding Types

The results in Table 6.10 show that the system consistently performed better when using the GloVe-embeddings rather than the word2vec ones. This is likely a result of the origin of both the data set used, as described in Chapter 4, and the different embeddings. The Waseem and Hovy (2016) data set was collected from Twitter. This is also where the data used to train the GloVe-embeddings were collected. The word2vec-embeddings, on the other hand, were trained on data from Google News, which does not have the same kind of character restrictions as Twitter. As such, the way words are used may differ between the platforms, resulting in different relations in the embeddings. The Twitter-based embeddings may thus be better suited to how messages are formulated in the hate speech data set. Furthermore, the word2vec-embeddings were trained after removing stop words (see Section 2.3.2) from the training data. As such, when using the word2vec-embeddings, the stop words were considered Out of Vocabulary terms and given a random value. With the average number of words in the tweets of the Waseem and Hovy (2016) data set being 15, the impact of not having a meaningful representation of stop words could be significant. Using the GloVe-embeddings, however, this issue is avoided, as these embeddings include representations of typical stop words. These arguments provide a plausible explanation as to why the GloVe-based system performed better than the word2vec-based version.

Comparison to state-of-the-art

Several other papers have tested their hate speech detection systems on the data set by Waseem and Hovy (2016). Table 7.3 shows the performance of several of these. Notably, as mentioned in Section 4.1, Waseem (2016) introduced another, but related, data set. The results presented here, however, show the performance on the Waseem and Hovy (2016) data set. Gambäck and Sikdar (2017), on the other hand, used the data set from Waseem (2016), while Park and Fung (2017) used both data sets combined. However, the results of both systems are still included in the table. Zhang et al. (2018) reported performance on multiple data sets, but only in terms of F_1-score. The score listed in Table 7.3 is the performance on the Waseem and Hovy (2016) data set.

A problem with the results shown in Table 7.3 is that different papers have used different methods to calculate the performance. While all papers use precision, recall and F_1, some calculate these using micro averaging, while others use macro averages. These are not directly comparable, and produce quite different results. As such, Table 7.3 includes both the macro and micro averaged performance of the optimal configuration found in Chapter 6. A further complication is that only one of the papers specify which average they use; Zhang et al. (2018) specify

System	Precision	Recall	F_1
Two conv. layers (64×4,64×3), GloVe	80.22%	78.28%	79.24
Two conv. layers (64×4,64×3), GloVe, micro avg	84.14%	84.14%	84.14
Waseem and Hovy (2016)	72.87%	77.75%	73.89
Waseem (2016)	—	—	53.43
Waseem (2016), binary classification	—	—	70.05
Gambäck and Sikdar (2017)	85.66%	72.14%	78.29
Zhang et al. (2018)	—	—	82
Park and Fung (2017)	82.7%	82.7%	82.7
Founta et al. (2018)	84%	83%	83
Badjatiya et al. (2017)	93.0%	93.0%	93.0
Badjatiya et al. (2017), without decision trees	83.9%	84.0%	83.9

Table 7.3: Comparison of the system performance to other papers using the data set from Waseem and Hovy (2016).

that they use micro averaging in their calculations. In the case of Park and Fung (2017), their use of micro averaging can be elicited from the results, which include performance on all three classes separately. For the rest of the papers, assumptions must be made based on the differences between precision, recall and F_1-score. By the nature of micro averaging, precision and recall in a multiclass classification are virtually the same. When using macro averaging, getting almost identical values is much less likely. Because of this, the conclusion can be drawn that Waseem and Hovy (2016), Waseem (2016) and Gambäck and Sikdar (2017) all used macro averages. The results from Waseem (2016) shown in Table 7.3 do not include precision and recall. However, the primary results of the paper, using the Waseem (2016) data set, do include all measures, and the conclusion can be drawn based on these. In addition, the table shows two different performances for the Waseem (2016) system. Here, the first is the multiclass performance, while the second is the performance when considering the identification as a binary problem, with racism and sexism combined into one category.

In the case of Founta et al. (2018) and Badjatiya et al. (2017), the similarity across the measures indicate that the results probably were calculated using micro average. This is, however, not entirely certain. It is also worth noting that the performance presented from Founta et al. (2018) is that of using only the text as input, not the metadata. This is because the metadata, as described in Section 3.2, included information about the text author. The research in this thesis specifically avoids using such information, and so the performance using just text is the better grounds for comparison.

As the macro averaged performance in Table 7.3 shows, the system described in Chapter 5, using the optimal configuration with two convolutional layers and GloVe-embeddings found in Chapter 6, outperformed the Waseem and Hovy (2016) system. It also had a higher performance, in terms of F_1-score, than the system of Gambäck and Sikdar (2017). However, this paper used a slightly different data set, and so the comparison is not entirely valid. In the case of the system by Waseem (2016), the approach described in Chapter 5 performed significantly better, particularly compared to the multiclass version. However, these results are not for the primary data set of Waseem (2016), which had markedly higher performance.

Using the micro averaged performance as basis for comparison, the system of Chapter 5 clearly outperforms the systems of both Zhang et al. (2018) and Park and Fung (2017). It also outperforms the system of Founta et al. (2018), when this is restricted to using text as input. However, the best system from Badjatiya et al. (2017) performed much better than the system from this thesis. Interestingly, this applied to all of the system versions using Gradient Boosted Decision Trees (GBDT), but not to any versions not using them. Table 7.3 also shows the performance of the best non-GBDT version, which is lower than that found in Chapter 6.

All in all, the system described in Chapter 5 outperforms many state-of-the-art systems. In particular, its performance is better than that of all the systems not using GBDTs. This is, as mentioned, given that the input is restricted to purely text-based, not allowing things like information about text authors.

7.2 Discussion

As described in Section 1.2, the goal of this thesis was to achieve automatic detection of hate speech in text, in a way that did not depend on any information specific to the online platform said texts were posted on. This was further specified by two Research Questions. In the following, the findings of this thesis will be discussed in terms of these Research Questions.

Research Question 1 What are found to be useful features and representations of text for hate speech identification, and what methods have been found effective for such detection?

As Chapter 3 – in particular, Section 3.2 – describes, research on hate speech detection has attempted many kinds of input features, and many different classification methods. In the early research, the input types used were highly language dependent, utilising language specific syntax features and the presence of certain words. Later, these kinds of features were exchanged for more general text representations as input. Specifically, the use got more directed towards word- and character n-grams (see Section 2.3.3). Here, the words were typically represented using word embeddings, as described in Section 2.3.4. While word embeddings are language dependent, in that they are representations of a specific language, the system using them is less so. The language of the embeddings used in training do restrict the system; however, it would work similarly well when trained on embeddings from another language, given that the input texts are of the corresponding language. This is significantly less language dependent than, say, looking for nouns following the word 'you,' as was done in Spertus (1997). On a similar note, a system trained on using character n-grams is only dependent on the language used in the training samples and, to a lesser degree, the character representations used. For instance, languages using the Latin alphabet would mostly work using the same representations, though there are some differences between the languages. However, for languages using different alphabets, e.g., Cyrillic and Greek, the character representations would have to be changed in order to make the transition from one to the other. Notably, character n-grams of this kind only work for alphabetical languages. In languages like Chinese, where each symbol carries an intrinsic meaning, character n-grams would essentially work like word n-grams – though they are still called character n-grams.

The specifics of how word embeddings and character n-grams have been used vary between different papers. Some, such as Gambäck and Sikdar (2017), use both types at the same time, while others, e.g., Waseem and Hovy (2016); Pavlopoulos et al. (2017a), used only one of the types. In Mehdad and Tetreault (2016), word and character n-grams were used separately, in order to compare their performance. Notably, the word n-grams in Mehdad and Tetreault (2016) were not in the form of word embeddings; rather, they were token n-grams. Their results showed that character n-grams were the more effective input format. Similarly, Waseem and Hovy (2016) initially performed a comparison of word – that is, token – and character n-grams, finding that use of character n-grams gave a significantly better performance. These findings are supported by the results shown in Table 6.9 from Chapter 6. The system described in Chapter 5 performed better when using only character input than it did when using only word-based input. In this case, the word input *was* in the form of word embeddings, while the characters were treated as n-grams. However, while this seems to confirm the findings of Mehdad and Tetreault (2016), the system setup is not quite the same for the two types of input, with the characters having a layer of convolution before the LSTM layer. As such, the argument could be made that the results are not directly comparable, thus invalidating that conclusion. Indeed, Gambäck and Sikdar (2017) had the opposite results; their word2vec-embeddings were found to outperform character n-grams. Essentially, this means that it is uncertain what works best of word embedding and character n-grams. However, both versions are found to work well in hate speech detection, and without some of the issues of the features used in the early research on the field.

As for the methods used in hate speech detectors, early research used traditional machine learning approaches such as those described in Section 2.1. Yin et al. (2009), for instance, used a Support Vector Machine (SVM) for their system. Some, like Razavi et al. (2010), used multiple steps of classification; in that particular case, three levels of Naïve Bayes-based classifiers (see Section 2.1.3). Some more recent research has also used traditional machine learning approaches. For instance, Waseem and Hovy (2016) used Logistic Regression (see Section 2.1.1) in their system. However, with the resurgence of Deep Learning in recent years, much research has moved its focus to such approaches instead. Here, several variations have been tested. For instance, Gambäck and Sikdar (2017) used a CNN, while Pavlopoulos et al. (2017a) used an RNN, in the form of Gated Recurrent Unit (GRU)-units. Others, like Zhang et al. (2018), used combinations of neural network-types. Specifically, Zhang et al. (2018) used a CNN followed by a GRU-based RNN. Yet others tested several different types, and deep learning approaches combined with more traditional methods. Badjatiya et al. (2017) is a notable example of this, testing both a CNN-based and an LSTM-based system, and using them in combination with Gradient Boosted Decision Trees (GBDT).

While many different methods for hate speech detection have been explored, comparing their performance is not straightforward due to the use of various different data sets. The differences in what, exactly, the targets of the classifications have been, as discussed in Section 3.3, only work to further complicate this. However, the goals of the more recent research – detecting some sort of hate speech – is arguably a harder problem than that of much early research – detecting more explicit flames and profanity. Among the systems that focused on what may justifiably be called hate speech detection, several were tested on the data set from Waseem and Hovy (2016), as shown in Table 7.3. As such, the methods used in these systems can be compared somewhat more directly. In general, deep learning methods seem to perform better than purely traditional machine learning approaches. This is not, however, an absolute truth. Mehdad and Tetreault (2016) found their SVM-Naïve Bayes classifier to perform better than their RNN-based system. On the other hand, the CNN-based system of Gambäck and Sikdar (2017) outperformed the Logistic Regression-based system of Waseem and Hovy (2016). Similarly, the system described in Chapter 5, and several other deep learning-based systems, also performed better than that of Waseem and Hovy (2016).

In general, pure deep learning systems – pure as in not including any traditional methods – seem to perform better than pure traditional approaches. However, as noted at the end of Section 7.1, the hybrid system described by Badjatiya et al. (2017), combining an LSTM with a GBDT, performed significantly better than all of the pure ANN-methods. Regarding method-input combinations, word embeddings have mainly been used in systems based on neural networks. The ANN-based methods generally use such embeddings and/or character n-grams as input; other text-based features are mostly used with traditional methods, in earlier research. There are cases where other features are included in the input of ANN systems, but these tend to contain domain-specific data, such as information about the author. One such architecture was described by Founta et al. (2018). As the goal of this thesis is in the area of domain-independent hate speech detection, papers relying on such information have largely been omitted. When restricted to text-based features, traditional methods, just like deep learning-based ones, often use character n-grams. Notably, this applies to the more recent research; older approaches typically used more high-level features.

Research Question 2 Would a deep learning model based on a two-pronged combination of a Convolutional Neural Network and Long Short-Term Memory-networks be effective at classifying tweets for hate speech?

Chapter 5 described a system architecture using the characters of a text as input to a Convolutional Neural Network, which was then followed by an LSTM network. In addition, the words of the text were transformed into word embeddings, and used as input to an additional LSTM network. The outputs of these networks were then combined, and used as input to a normal, feed-forward ANN, producing the final classifications. This system, whose architecture conforms to the descriptions of the Research Question, was tested through the experiments of Chapter 6. As previously noted, this system outperforms most state-of-the-art systems. Thus, one may justifiably say that, at least in a relative perspective, the two-pronged CNN-LSTM combination does effectively identify hate speech in tweets. However, there are some issues regarding the exploration of the system architecture and the described performance that need to be addressed before any conclusions can be drawn.

The perhaps most important issue is that the system is really best at detecting non-hate speech. Its ability to actually detect hate speech is significantly lower than its ability to recognise the more or less benign cases. If the neutral classifications generally could be trusted, this would be helpful, as it would essentially reduce the number of samples that were potentially hateful. However, a significant fraction of the hate speech samples were misclassified as neutral; this was particularly true for sexist samples. In addition, many of the samples classified as hate speech really belonged to the neutral class – that is, the precision was low. As such, the practical usability of the system is restricted; if used for automatic moderation of a web site or platform based on user generated content, a fair amount of hate speech would avoid notice. Furthermore, many acceptable texts would be wrongfully censored as hate speech. From a liberty-of-speech perspective, this would be unacceptable. Even in the case of the less demanding task of semi-automatic moderation, with the system providing initial evaluation, followed by final decisions from a human, the relatively low recall scores of the hate speech classes means that the usefulness is not very high. Specifically, the low recall means that all samples evaluated as 'neutral' have to be checked anyhow, while the low precision disallows automatic removal of hateful posts. To be practical for use in semi-automatic moderation, the recall would have to be significantly higher, so that few hateful samples would go unnoticed. Then, almost all cases of hate speech could be found by looking through the samples the system labelled as hateful. This would work even when the precision is significantly lower than the recall, since the samples testing positive for hate speech would be further moderated by a human. As these arguments indicate, the system's performance in the comparisons above only means that the described architecture is effective in a relative perspective; in an absolute perspective, it is less so. At the same time, it is clearly not ineffective, with its F_1-performance at almost 80.

Beyond this practicality-issue, there is the flaw in the experiments mentioned in Section 7.1: that most of the bidirectional experiments provided little to no actual difference from the unidirectional equivalents. This was, as mentioned, because of the single-layered LSTMs, and the way data were extracted from the output for further use in the system. Due to this flaw, the experiments in Chapter 6 did not explore the difference between unidirectional and bidirectional LSTMs as thoroughly as intended. While the configurations displaying the highest performances contained only single-layer LSTMs, there is a possibility that multilayer, bidirectional LSTMs, if better explored, might lead to further improvements. In addition, the algorithm used for extracting the relevant LSTM output could be adjusted so that it obtains the data from the last time step of the backward direction – that is, from the first actual time step – in addition to the last output from the forward direction. With this modification, the system would make full use of bidirectional LSTMs even in the single-layer case.

As mentioned in Section 7.1, different convergence rates in the first two system components may have caused the training to work sub-optimally. This, in turn, would lead to sub-optimal performance. In essence, this means that the full capabilities of the various configurations of Section 6.2 may not have been seen; the system could actually be more effective at detecting hate speech than the results indicate. Both this and the issue with bidirectionality show that some aspects of the exploration of the system architecture were incomplete; better results may be found by introducing some changes to how the system operates and is trained.

The experiments in Chapter 6 showed that the system performed better when using GloVe-vectors as its word input than when using the word2vec-embeddings. As explained in Section 7.1, this may, in part, be because the GloVe-embeddings were trained on tweets, while the word2vec-embeddings were not. The inclusion/exclusion of stop words likely comprises the main difference in performance. However, if the specifically tweet-based training does have an impact on the performance, this means that the improvement is due to platform specific information. While the Research Question is specifically concerned with detecting hate speech in tweets, the overarching goal of this research aims at performing such detection independently of what platform the texts are from. As such, advantages resulting from using word embeddings trained specifically on data from the same platform are undesirable from a generality perspective. A similar argument could, conceivably, be made on the character representations used, with their inclusion of the special characters '@' and '#.' As these symbols have a particular meaning in tweets, such an argument would have some validity. At the same time, this use of the symbols is not restricted to Twitter; they are also used on other platforms, though the prevalence may vary. Furthermore, their being part of the representation is unlikely to have much negative impact on performance on text from platforms where they are not common, as the CNN is likely to ignore characters that are absent in the training data. As such, the inclusion of these symbols in the character representations is acceptably generalisable. It should be mentioned, though, that the inclusion is likely to have more impact on tweets than on texts from platforms where the symbols are used less frequently – but are still used. In the case of the GloVe-embeddings, this possible generality issue does not affect the comparison to other state-of-the-art systems in Section 7.1. Even if the GloVe results were considered not just partially, but completely invalid – a preposterous supposition, given that the meanings of words are, fundamentally, the same on Twitter as elsewhere – the word2vec-based equivalent performs similarly compared to the other systems. Indeed, the micro averaged performance of the word2vec-system is slightly higher than that of the GloVe-based version, with an F_1-score of 84.19. As such, the evaluations of how effectively the two-pronged, CNN-LSTM-based system described in Chapter 5 identifies hate speech are valid regardless of domain-specific advantages like those described above.

All in all, the architecture of Chapter 5, with its two-pronged CNN-LSTM combination, is relatively effective at detecting hate speech in tweets. It outperforms most state-of-the-art systems; however, it is still not effective enough to be of much practical use in moderation. There is also one system, described by Badjatiya et al. (2017), that performs significantly better. At the same time, there are some possibilities for improving the detection rates of the system. On a further note, the architecture is set up to be applicable not just to tweets, but to content from any platform based on user generated texts. Notably, the system in the experiments of Chapter 6 was trained solely on tweets, and so it would likely perform poorly on other types of text without further training. Furthermore, the system could be used on texts of other languages, given appropriate training data and that word embeddings of the corresponding language are used.

8 Conclusion and Future Work

As described in Section 1.2, the goal of this thesis was to enable automatic, domain-independent detection of hate speech in text. This was further specified by two Research Questions; one concerning what features and methods have proved to be useful in such detection, and the other looking at whether a model specifically based on a two-pronged combination of a Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM)-networks would be effective at identifying hate speech.

As Chapter 3 showed, much research has been done in the area of detecting abusive language. This research has used various different methods, with different kinds of features extracted from the texts. In the recent, state-of-the-art approaches, the features used have mainly been word embeddings and character n-grams. Each of these have their own advantages. With word embeddings, related words provide similar input values, enabling the system to treat them in a consistent manner. Character n-grams, on the other hand, are tolerant in regard to spelling errors – intended or otherwise. In terms of methods, the state-of-the-art approaches tend towards the use of Artificial Neural Networks (ANN) of various kinds.

While the recent research appear to have reached a consensus on what kinds of features are effective for detection, opinions on *targets* vary. With the growing focus on hate speech in society at large (Thomasson, 2017; European Commission, 2017), the research has arguably become more directed towards hate speech, rather than generally abusive language. However, as described by Waseem et al. (2017), and further shown in Section 3.3, the definitions for what constitutes hate speech differ. This is mirrored in the available data sets, with each having its own definition as the basis of its target values. Due to the use of different data sets and measures, comparing results between papers is not straightforward. However, several recent studies have reported results on the data set of Waseem and Hovy (2016).

The two-pronged CNN-LSTM combination, described in Chapter 5, which used both word embeddings and character n-grams as input, performed relatively well on the Waseem and Hovy (2016)-data set. Specifically, the resulting system did well when using two layers of convolution on the character input, with diminishing filter lengths, combined with single layer LSTMs in both prongs. Using multiple layers of LSTMs, on the other hand, did not help the detection, and actually reduced the performance.

In comparison to other detection approaches whose performance was reported for the Waseem and Hovy (2016) data set, the architecture of Chapter 5 was indeed effective at identifying hate speech in text. It performed better than all but one of the comparable, state-of-the-art systems; only the LSTM-GBDT system described by Badjatiya et al. (2017) achieved higher scores in terms of F_1-scores.

While the architecture described in this thesis was quite effective at hate speech detection compared to state-of-the-art systems, it was – as noted in Section 7.2 – still not effective or reliable enough to be practically usable for automatic moderation of platforms with user generated content. As such, further research in the field is still necessary before systems are good enough to be relevant for the recommendations by the European Commission (2017), described in Chapter 1.

8.1 Contributions

This thesis contributes to the area of hate speech detection in texts through a review of related research, which showed that character n-grams and word embeddings have proven the best text-based input features for identifying hate speech. It also showed that character n-grams were used successfully in methods from both traditional machine learning and deep learning, while word embeddings were useful mainly in deep learning approaches. Further, a discussion was made on the variations in classification targets though the history of hate speech detection research.

The main contribution of this thesis is the novel, deep learning-based approach to hate speech detection described in Chapter 5, with its two-pronged architecture combining CNNs and LSTMs, and its use of both characters and word embeddings as input. After thorough exploration of this architecture, the approach was found to perform well compared to state-of-the-art systems, achieving a macro averaged F_1-score of 79.24.

8.2 Future Work

During the evaluation of the results found in the experiments, some issues were discovered that inspired ideas for further research. Foremost among these is the possibility that different convergence rates in the architecture's word-based and character-based components may have led to a reduced performance, as mentioned in Section 7.1. An idea for avoiding the problems of different convergence rates is to train the system using an interleaving technique, as described by Founta et al. (2018), so that only one of the two parallel system components is trained at any given time. In other words, the weights of one component are kept constant during the training on one mini batch, and the weights of the other for the next mini batch. This way, the training of each component is done independently of the other, preventing undesirable cross-interactions between them.

As mentioned in Section 7.1, most of the experiments using bidirectional LSTMs did not have much use of the backwards-running part. Indeed, all the bidirectional experiments ignored much of the potential information. Further work on the architecture described in Chapter 5 could change the manner in which the information for use in the final component is extracted. Such a change would, if done correctly, extract the information from the last relevant time step for unidirectional LSTMs and the forwards-running part of bidirectional ones, while the backwards-running part of bidirectional LSTMs would have the information from the first time step extracted. With this change, the resulting system would be able to make full use of the bidirectionality, thus enabling exploration of the true effect of changing from uni- to bidirectional LSTMs.

Section 7.2 discussed the possibility that some of the experiments of Chapter 6 had an advantage due to the use of a Twitter-based data set. While the influence of this domain-specific advantage was deemed to be minor, exploring these effects in more detail would be desirable. Specifically, this could be done by comparing the performance of using the two sets of word embeddings on a hate speech data set that was not collected from Twitter. Thus, any difference between the word embedding types would be from reasons other than domain-dependence. However, finding a suitable data set for such experiments is not straightforward. As Section 4.1 showed, many of the data sets used in abusive language detection-research are not really on hate speech, e.g., Sood et al. (2012). Others, like that of Warner and Hirschberg (2012), are not publicly available. Yet others, like the data set by Pavlopoulos et al. (2017a), are not in English, and so are not suitable for these experiments. If an appropriate data set is found, a notable aspect of the experiments is that the impact of having or lacking representations for stop words would probably be reduced, as the text samples, free of the – previous – 140 character restriction of Twitter, are likely to have a higher number of words than in the data set of Waseem and Hovy (2016). As such, the fraction of words that are stop words would likely differ between the data sets.

Along the lines of using different data sets, another interesting angle would be to look at how well a system trained on a Twitter data set would perform on a non-Twitter data set, and vice versa. This would help determine how generalisable said system is across different platforms and domains, which would be a highly desirable property; it would, in a practical setting, allow use on platforms where there is not sufficient data for proper training. Such experiments, however, would require two data sets in the same language, with the same kind of labels. As such, creation of at least one new data set would likely be necessary to make these comparisons.

Beyond these ideas for research using non-Twitter data sets, it would also be interesting to investigate how well the architecture of Chapter 5 would perform on other Twitter data; in particular, testing the performance on the data set by Golbeck et al. (2017) would be desirable.

Badjatiya et al. (2017) found that those of their systems which used Gradient Boosted Decision Trees (GBDT) performed significantly better than the pure deep learning-approaches. As such, an idea for further research would be to test the impact of using the architecture of Chapter 5 in combination with a GBDT. If the patterns from Badjatiya et al. (2017) hold, this would substantially increase performance.

Bibliography

Salman Aslam. Twitter by the numbers: Stats, demographics & fun facts, January 2018. URL https://www.omnicoreagency.com/twitter-statistics.

Pinkesh Badjatiya, Shashank Gupta, Manish Gupta, and Vasudeva Varma. Deep learning for hate speech detection in tweets. In *Proc. of the 26th International Conf. on World Wide Web Companion*, WWW '17 Companion, pages 759–760, Perth, Australia, 2017. International World Wide Web Conferences Steering Committee.

Imad A. Basheer and Maha Hajmeer. Artificial neural networks: Fundamentals, computing, design, and application. *Journal of Microbiological Methods*, 43(1): 3 – 31, 2000.

Thomas Bayes. LII. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, F. R. S. communicated by Mr. Price, in a letter to John Canton, A. M. F. R. S. *Philosophical Transactions*, 53:370–418, 1763.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3:1137–1155, February 2003.

Per-Christian Berg and Manu Gopinathan. A deep learning ensemble approach to gender identification of tweet authors, June 2017.

Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proc. of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, Pittsburgh, Pennsylvania, USA, 1992. ACM.

Pete Burnap and Matthew L. Williams. Hate speech, machine classification and statistical modelling of information flows on Twitter: Interpretation and communication for policy decision making. In *Internet, Policy & Politics*, September 2014.

Ying Chen, Yilu Zhou, Sencun Zhu, and Heng Xu. Detecting offensive language in social media to protect adolescent online safety. In *2012 International Conf.*

on Privacy, Security, Risk and Trust and 2012 International Conf. on Social Computing, pages 71–80, Amsterdam, Netherlands, September 2012.

KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv e-print*, abs/1409.1259, 2014.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

David R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20:215–242, 1958.

Thomas Davidson, Dana Warmsley, Michael W. Macy, and Ingmar Weber. Automated hate speech detection and the problem of offensive language. *arXiv e-print*, abs/1703.04009, 2017.

European Commission. Security union: Commission steps up efforts to tackle illegal content online, September 2017. URL http://europa.eu/rapid/press-release_IP-17-3493_en.htm?utm_source=Nyhetsbrevet&utm_campaign=64603ed604-hd3917&utm_medium=email&utm_term=0_e53510dee0-64603ed604-161679417.

John R. Firth. A synopsis of linguistic theory 1930-1955. In *Studies in Linguistic Analysis*, pages 1–32. Philological Society, Oxford, England, United Kingdom, 1957.

Paula Fortuna. Automatic detection of hate speech in text: an overview of the topic and dataset annotation with hierarchical classes. Master's thesis, Universidade do Porto, Portugal, June 2017.

Antigoni-Maria Founta, Despoina Chatzakou, Nicolas Kourtellis, Jeremy Blackburn, Athena Vakali, and Ilias Leontiadis. A unified deep learning architecture for abuse detection. *arXiv e-print*, abs/1802.00385, 2018.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Proc. of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 1027–1035, 2016.

Björn Gambäck and Utpal Kumar Sikdar. Using convolutional neural networks to classify hate-speech. In *Proc. of the 1st Workshop on Abusive Language Online*, pages 85–90, Vancouver, Canada, August 2017. ACL.

Lei Gao, Alexis Kuppersmith, and Ruihong Huang. Recognizing explicit and implicit hate speech using a weakly supervised two-path bootstrapping approach. *arXiv e-print*, abs/1710.07394, 2017.

Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: continual prediction with LSTM. In *9th International Conf. on Artificial Neural Networks*, pages 850–855, Edinburgh, Scotland, United Kingdom, January 1999. Institution of Engineering and Technology.

Jennifer Golbeck, Zahra Ashktorab, Rashad O. Banjo, Alexandra Berlinger, Siddharth Bhagwan, Cody Buntain, Paul Cheakalos, Alicia A. Geller, Quint Gergory, Rajesh Kumar Gnanasekaran, Raja Rajan Gunasekaran, Kelly M. Hoffman, Jenny Hottle, Vichita Jienjitlert, Shivika Khare, Ryan Lau, Marianna J. Martindale, Shalmali Naik, Heather L. Nixon, Piyush Ramachandran, Kristine M. Rogers, Lisa Rogers, Meghna Sardana Sarin, Gaurav Shahane, Jayanee Thanki, Priyanka Vengataraman, Zijian Wan, and Derek Michael Wu. A large labeled corpus for online harassment research. In *Proc. of the 2017 ACM on Web Science Conference*, WebSci '17, pages 229–233, Troy, New York, USA, June 2017.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016.

Alex Graves, Abdel rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conf. on Acoustics, Speech and Signal Processing*, pages 6645–6649, Vancouver, BC, Canada, May 2013.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, November 1997.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

Yann LeCun. Generalization and network design strategies. In *Connectionism in perspective*, pages 143–155. Elsevier, Zürich, Switzerland, 1989.

Yashar Mehdad and Joel Tetreault. Do characters abuse more than words? In *Proc. of the 17th Annual Meeting of the Special Interest Group on Discourse and*

Dialogue, pages 299–303, Los Angeles, CA, USA, September 2016. Association for Computational Linguistics.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv e-print*, abs/1301.3781, 2013a.

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proc. of the 2013 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013b.

Marvin Minsky and Seymour A. Papert. *Perceptrons.* MIT Press, Cambridge, MA, USA, 1969.

Norsk Telegrambyrå (NTB). Hver tiende kommentar på Facebook-sidene til TV 2 og NRK er hatefull, November 2017. URL https://journalisten.no/2017/11/hver-tiende-kommentar-pa-facebook-sidene-til-tv-2-og-nrk-er-hatefull.

Ji Ho Park and Pascale Fung. One-step and two-step classification for abusive language detection on Twitter. In *Proc. of the 1st Workshop on Abusive Language Online*, pages 41–45, Vancouver, Canada, August 2017. ACL.

John Pavlopoulos, Prodromos Malakasiotis, and Ion Androutsopoulos. Deep learning for user comment moderation. In *Proc. of the 1st Workshop on Abusive Language Online*, pages 25–35, Vancouver, Canada, August 2017a. ACL.

John Pavlopoulos, Prodromos Malakasiotis, and Ion Androutsopoulos. Deeper attention to abusive user content moderation. In *Proc. of the 2017 Conf. on Empirical Methods in Natural Language Processing*, pages 1136–1146, Copenhagen, Denmark, September 2017b.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

Amir H. Razavi, Diana Inkpen, Sasha Uritsky, and Stan Matwin. *Offensive Language Detection Using Multi-level Classification*, pages 16–27. Springer, Ottawa, Canada, 2010.

Björn Ross, Michael Rist, Guillermo Carbonell, Benjamin Cabrera, Nils Kurowsky, and Michael Wojatzki. Measuring the reliability of hate speech annotations: The case of the European refugee crisis. In *Proc. of NLP4CMC III: 3rd Workshop on Natural Language Processing for Computer-Mediated Communication*, volume 17, pages 6–9, Bochum, Germany, September 2017. Bochumer Linguistische Arbeitsberichte.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, October 1986.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education Limited, Harlow, Essex, England, United Kingdom, Third, New International edition, 2014.

Ali Sadreddini. Kernel machine, April 2011. URL https://commons.wikimedia.org/wiki/File:Kernel_Machine.svg.

Tom Schaul, Ioannis Antonoglou, and David Silver. Unit tests for stochastic optimization. *CoRR*, abs/1312.6055, 2013.

Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, November 1997.

Sara Owsley Sood, Elizabeth F. Churchill, and Judd Antin. Automatic identification of personal insults on social news sites. *Journal of the American Society for Information Science and Technology*, 63(2):270–285, 2012.

Ellen Spertus. Smokey: Automatic recognition of hostile messages. In *Proc. of the Fourteenth National Conf. on Artificial Intelligence and Ninth Conf. on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 1058–1065, Providence, Rhode Island, USA, 1997. AAAI Press.

Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Emma Thomasson. German cabinet agrees to fine social media over hate speech, April 2017. URL http://uk.reuters.com/article/uk-germany-hatecrime-facebook/german-cabinet-agrees-to-fine-social-media-over-hate-speech-idUKKBN1771FK.

William Warner and Julia Hirschberg. Detecting hate speech on the World Wide Web. In *Proc. of the Second Workshop on Language in Social Media*, LSM '12, pages 19–26, Montreal, Canada, 2012. Association for Computational Linguistics.

Zeerak Waseem. Are you a racist or am I seeing things? Annotator influence on hate speech detection on Twitter. In *Proc. of the First Workshop on NLP and Computational Social Science*, pages 138–142. Association for Computational Linguistics, 2016.

Zeerak Waseem and Dirk Hovy. Hateful symbols or hateful people? Predictive features for hate speech detection on Twitter. In *Proc. of the 2016 Conf. of the North American Chapter of the Association for Computational Linguistics*, pages 88–93, San Diego, CA, USA, 2016. Association for Computational Linguistics.

Zeerak Waseem, Thomas Davidson, Dana Warmsley, and Ingmar Weber. Understanding abuse: A typology of abusive language detection subtasks. In *Proc. of the 1st Workshop on Abusive Language Online*, pages 78–84, Vancouver, Canada, August 2017. ACL.

Dawei Yin, Brian D. Davison, Zhenzhen Xue, Liangjie Hong, April Kontostathis, and Lynne Edwards. Detection of harassment on Web 2.0. In *Proc. of the Content Analysis in the Web 2.0 Workshop at WWW2009*, Madrid, Spain, 2009.

Ziqi Zhang, David Robinson, and Jonathan Tepper. Detecting hate speech on Twitter using a convolution-GRU based deep neural network. *Lecture notes in computer science*, March 2018.

Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis C. M. Lau. A C-LSTM neural network for text classification. *arXiv e-print*, abs/1511.08630, 2015.

Yitong Zhou and Rama Chellappa. Computation of optical flow using a neural network. In *IEEE 1988 International Conf. on Neural Networks*, volume 2, pages 71–78, San Diego, CA, USA, July 1988.