Threat detection in online discussion using convolutional neural networks

Camilla Emina Stenberg

Thesis submitted for the degree of
Master in Informatics: Technical and scientific applications
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2017
Threat detection in online discussion using convolutional neural networks

Camilla Emina Stenberg
© 2017 Camilla Emina Stenberg

Threat detection in online discussion using convolutional neural networks

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo
Abstract

Unwanted behaviour in terms of threats of violence in online discussions are of common occurrence. We can find such behaviour in e.g. comment fields on Facebook, Youtube and in online news papers. With increasing use of these kind of discussion arenas, it generates a lot of work for moderators that have to, at worst manually go through comments, and remove the ones containing undesired content. In this project, we use a corpus with YouTube comments. The task will be to classify comments as containing violent threats or not. The comments in the corpus are manually annotated as "threat" or "non-threat". To attempt to solve this, we use deep learning techniques in combination with word embeddings. We have systematically explored the effects of a range of different choices regarding architecture and parameterization. In our result we find that threat detection using convolutional neural networks do not outperform earlier work on the same task.
Acknowledgments

I would like to express my sincere appreciation and gratitude to:

- My supervisors Lilja Øvrelid and Erik Velldal, for their support, help and patience throughout this project.
- My fellow master students.
- My beloved daughter Ella.
- Last but not least, I would like to express my deepest gratitude to my family for their unconditional love and support.
Contents

1 Introduction 1

2 Background 3
   2.1 Detecting unwanted behaviour in social media . . . . . . 3
   2.2 Deep learning for NLP . . . . . . . . . . . . . . . . . . . . 5
      2.2.1 Input layer . . . . . . . . . . . . . . . . . . . . . . . 6
      2.2.2 CNN architecture . . . . . . . . . . . . . . . . . . . . 8
   2.3 Deep learning for SA . . . . . . . . . . . . . . . . . . . . . 9
      2.3.1 Kalchbrenner, Grefenstette and Blunsom 2014 . . . . 9
      2.3.2 Kim 2014 . . . . . . . . . . . . . . . . . . . . . . . . . 10

3 Data set and tools 15
   3.1 YouTube data set . . . . . . . . . . . . . . . . . . . . . . . 15
   3.2 Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
      3.2.1 Keras . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
      3.2.2 GloVe . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

4 Experimental setup 21
   4.1 Data set splits and cross validation . . . . . . . . . . . . . 21
   4.2 Evaluation metrics . . . . . . . . . . . . . . . . . . . . . . . 23

5 Preliminary experiments 27
   5.1 Establishing some baselines . . . . . . . . . . . . . . . . . 27
      5.1.1 Non machine learning classifiers . . . . . . . . . . . . 27
      5.1.2 CNN baseline model . . . . . . . . . . . . . . . . . . . 29
   5.2 Number of epochs . . . . . . . . . . . . . . . . . . . . . . . 31
   5.3 Controlling for non-determinism . . . . . . . . . . . . . . . 31
   5.4 Bug causing extra training of embeddings . . . . . . . . . . 35

6 Experiments 37
   6.1 Adding an embedding layer . . . . . . . . . . . . . . . . . 37
      6.1.1 Static VS non-static embedding weights . . . . . . . . 38
      6.1.2 Unknown word initialization . . . . . . . . . . . . . . 39
   6.2 Scaling the loss function due to imbalanced data set . . . . 41
   6.3 Experimenting with different pre-trained embeddings . . . 43
List of Figures

2.1 Illustration of the architecture for the model by Kalchbrenner, Grefenstette and Blunsom (2014). A DCNN for the seven word input sentence. Word embeddings have size $d = 4$. The network has two convolutional layers with two feature maps each. The widths of the filters at the two layers are respectively 3 and 2. The (dynamic) k-max pooling layers have values $k$ of 5 and 3. Illustration: Kalchbrenner, Grefenstette and Blunsom (2014). ........................................ 11

2.2 Illustration of the architecture for the model by Kim (2014). Model architecture with two channels for an example sentence. Illustration: Kim (2014). ................................. 13

3.1 Examples of YouTube comments from the data set ........... 20

5.1 F-score for baseline run over 200 epochs. Comparison between train and validation results over the epochs ......... 32

5.2 Accuracy for baseline run over 200 epochs. Comparison between train and validation results over the epochs ......... 32

5.3 Loss for baseline run over 200 epochs. Comparison between train and validation results over the epochs .......... 33

6.1 Examples of unknown words from training set data .......... 40

6.2 Examples of false negative sentences for development results, using our final model. ................................. 58

6.3 Examples of false positive sentences for development results, using our final model. ................................. 59
List of Tables

3.1 YouTube Threat Corpus count data ........................................... 16
4.1 Partition of data into development set and held out test set ........... 22
4.2 Partition of development set into 5 folds for cross validation ........ 22
4.3 Confusion matrix with True/False Positives/Negatives ............... 24
5.1 Data set statistics ................................................................. 28
5.2 Confusion matrix for the majority class classifier ....................... 28
5.3 Confusion matrix for the minority class classifier ....................... 29
5.4 Confusion matrix for the random guess classifier ....................... 29
5.5 Confusion matrix for the weighted guess classifier ...................... 30
5.6 Non machine learning model results ....................................... 30
5.7 Baseline model results ......................................................... 31
5.8 10 runs of baseline model with different seeds, 5 epochs .......... 34
5.9 10 runs of baseline model with different seeds for initialization, same seed for model, 5 epochs .......................................................... 34
6.1 Model w/ embeddings trained on Wikipedia+Gigawords ............. 38
6.2 Number of tokens/unique words present in development data, not present in embedding corpora .......................... 40
6.3 Model w/ embeddings trained on Wikipedia+Gigawords dim 100, two different initialization methods for unknown words .......................................................... 41
6.4 Different ratios for static scaling of loss function ...................... 42
6.5 Different dynamic scaling of loss function. Formula
\( (\ln(\text{threats}) : \ln(100 - \text{threats}) \times x) \) used for scaling, where
\( x \) is varied through the experiments ........................................... 42
6.6 Results from use of different embedding dimensions for
pre-trained embeddings on Wikipedia + Gigaword, loss function scaling \( \ln(\text{threats}) : \ln(100 - \text{threats}) \) ....................... 43
6.7 Results from use of different embedding training sets ................. 45
6.8 Grid search window size and feature map size. F-score. Wikipedia + Gigaword 100 dim embeddings ......................... 46
6.9 Results from use of multiple window sizes with 128 feature maps .......................................................... 47
6.10 Results from use of multiple window sizes with 64 feature maps 48
6.11 Results from use of dropout on input layer 49
6.12 Results from use of dropout on hidden layer 50
6.13 Results from use of dropout on both input layer and hidden layer 51
6.14 Results from use of ADAM optimizer function, compared to the adadelta and the adagrad optimizer functions 51
6.15 Results from use of different batch sizes 52
6.16 Results from using different number of nodes in hidden layer 53
6.17 Results from model using multiple convolutional layers without dropout 54
6.18 Results from model using multiple convolutional layers with dropout (0.6) on input layer 54
6.19 Results from models: Concat. model: our final model using concat. conv. layers, Multiple seq. model: our model using multiple seq. conv. layers, Wester model: best dev. score achieved by Wester et al. (2016), Baseline CNN model: model described in sec. 5.1.2, Aksel BoW MaxEnt model: BoW MaxEnt model created by Wester et al. (2016) 56
6.20 Confusion matrix for development results, using our final model. Confusion matrix showing true positives, false positives, false negatives and true negatives 56
7.1 Data set partitioning into development set and held out set, showing number of sentences containing threats 62
7.2 Held out results for our model, compared to results provided by Wester et al. (2016) 62
Chapter 1

Introduction

Threats of violence is a common problem in social media. Several people, both individuals and social media platforms, are trying to combat this abusive behaviour. This kind of behaviour is not only causing negative social climate in the platforms struggling with it, but could also lead to actual physical violence.

The fact that social media platforms operates on such a large scale means that manually going through all posts/comments would be an almost impossible task. An automatic method for detecting threats is therefore crucial for being able to combat this problem. Such automatic detection methods could be of use also for the government in the process of investigation, or search of criminals in e.g. terror matters. This thesis will describe development and experiments with such automatic methods for detecting threats of violence in social media.

There has been a lot of research lately concerning unwanted behaviour in social media, and how to combat it. Several methods have been proposed for automatically dealing with the problem. A large part of the previous work done on the topic rely on rule-based or traditional machine learning methods. Recently, there has been an increase of research done using deep learning methods for natural language processing tasks. In this work we will investigate the application of deep learning to the task of threat detection. More specifically, we will use convolutional neural networks to solve this task.

We will start out by presenting some previous work done on this task, and other similar tasks (Chapter 2). We will first investigate work done on a task similar to ours. This includes both work done on detecting threats of violence, but also detecting what is called "cyberbullying". The work is done using YouTube and Twitter as data sets. We will then look at deep learning methods used for NLP purposes, and especially look into the architecture of convolutional neural networks. We will also take a closer
look at two studies regarding deep learning for sentiment analysis.

For our task, we are going to use an existing data set consisting of YouTube comments. The comments are split into sentences, and the whole data set contains a number of 9,845 comments split into 28,643 sentences. The existing data set is manually annotated. Our task will be to classify each sentence as containing threats of violence or not. The data set is described in detail in chapter 3. Chapter 3 also describes two main tools used while developing our classification model. This includes the python library Keras, used for creating our model. It also includes GloVe, used for the training of word embeddings.

A description of our experimental setup, including data set split, cross validation and evaluation metrics used, are given in chapter 4. Chapter 5 contains a description of our preliminary experiments. We will first establish some non machine learning baselines such as majority class classifier and random guess classifier. This is done to illustrate the intuitive difficulty of the task. We will then move on to introducing a CNN baseline model. This will be described in detail, and results for the baselines will be presented. The CNN baseline will also be used as a base for further experimentation.

There is a vast range of design choices available when configuring a CNN model, regarding both the basic architecture, input representations, and various parameters. We will systematically gauge the effect of several such choices. All our experimentation will be described in detail in chapter 6. Development results will be given consecutively through the chapter. The chapter will go through different experiments done to create a final model. This includes experimenting with different word embeddings, mitigating the problem of an unbalanced data set, different aspects regarding the model architecture, regularization and depth of the network. We will arrive at a final model that we will use for evaluation on a held out data set. We will also do an analysis of the errors made by the final model on the development data set. Chapter 7 in the thesis will be used for evaluating our final model on a held out test set, and an analysis of these results.
Chapter 2

Background

In this chapter we will go through some concepts and previous work done on task similar or relevant for our task. We will first investigate work done on the task of detecting unwanted behaviour in social media. This includes unwanted behaviour in both Twitter and YouTube comments. We will then look at methods and concepts used in deep learning for natural language processing (NLP) purposes. We will especially take a closer look at the convolutional neural network (CNN) architecture. At least, we will investigate previous work done on the task of sentiment analysis using CNNs.

2.1 Detecting unwanted behaviour in social media

Different methods have been developed for the task of detecting unwanted behavior from users on web sites, to avoid manually moderating comments. We will in this section review some previous work done on detecting unwanted behaviour in terms of threats of violence and "cyber-bullying" in both Twitter posts and YouTube comments. First we will look at a set of articles on the task of detecting threats of violence in Dutch tweets. Secondly we are going to look at a set of articles concerning detecting unwanted behaviour in YouTube comments.

The first articles we look into concern detecting threats of violence in Dutch tweets. Oostdijk and Halteren (2013a) describe a method of using n-grams to detect threats of violence in Dutch tweets. They have two different approaches for doing this. The first method uses manually constructed recognition patterns in form of bi- and trigrams, as well as skip bi- and trigrams. The second method described in Oostdijk and Halteren (2013a) is divided into two steps. In the first step machine learning was used to identify n-grams indicating threats of violence. The next step includes au-
thor identification, based on the use of the indicative bi-grams. They also do a follow-up work, where they try to improve the results (Oostdijk and Halteren 2013b). The difference from the earlier work is that they add a shallow parsing mechanism. The parsing is driven by handcrafted rules.

The next paragraphs will concern work done on the task of detecting unwanted behaviour in YouTube comments. Dinakar, Reichart and Lieberman (2011) describe a method using machine learning on a data set of YouTube comments. The aim of their work is to detect what they call "cyberbullying". The method is based on the use of a lexicon of negative words and part-of-speech tags that are correlated with bullying. The features used in the model are commonly observed uni- and bigrams.

Hammer (2014) also describes a method for detecting threats of violence using machine learning. The data set used by Hammer (2014) is a set of YouTube comments, and is a previous version of the one that we are going to use in our project. The aim of the model described is to classify a sentence as containing threats of violence or not. To do this, they use logistic LASSO regression analysis on bigrams of important words. They do, like Dinakar, Reichart and Lieberman (2011), use bigrams (skip-grams) from the training data as features for the model. The method uses a set of pre-compiled words that are correlated with threats of violence to extract these features.

The work most related to our project is that of Wester "Threat detection in online discussions" (Wester et al. 2016). They are using the same data set that we are going to base our work on. The data set is collected and annotated, by Hammer (2014). The sentences were assembled into comments, and the data set also had to be partly re-annotated (Wester et al. 2016). The initial annotation was done on sentence-level, isolating each sentence from its context. The re-annotation was done because some sentences may change meaning when seen in a different context, as part of a comment. Some extra comments were also added. Wester et al. (2016) are training a machine-learned model using three different classifiers in their work: Maximum Entropy, Support Vector Machines and Random Forests. Both lexical (word form, lemma), morphosyntactic (Penn Treebank POS, Universal POS, Dependency Relation) and lexical semantic (Brown cluster label, WordNet synset) information sources are used for defining features. The conclusion of the work done by Wester et al. (2016) does show that there is no improvement in introducing the more complex features. According to their work, the best performance was observed for combinations of simple lexical features (BoW, lemmas, bi- and trigrams).

What the aforementioned work has in common is that they all use traditional methods, either relying on manually crafted rules/patterns and/or
using traditional linear machine learning based on manual feature engineering, for their models. These methods may be effective, but are extremely specific to the task. In contrast to these methods, the approach in our project will be to use word embeddings in combination with deep neural nets.

### 2.2 Deep learning for NLP

Deep learning has seen rapid development in different fields in recent years. One of the fields where deep learning has reached remarkable results is natural language processing. Deep learning is based on an old idea, but can now be realized in an effective manner. The old idea of artificial neural nets is based on the thought of how the brain works, and how it is able to recognize and discriminate. The algorithm was called perceptron (Rosenblatt 1958), and was one of the first artificial neural networks to be produced. What the perceptron does, is basically mapping an input vector to an output value using a weight vector and an activation function:

\[
f(x) = g(x \cdot w + b),
\]

where \(x\) is the input vector, \(w\) is the weight vector, \(b\) is a bias term and \(g\) is the activation function. The perceptron algorithm is using the Heaviside step function as activation function \((g)\) to output a boolean value:

\[
g(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}
\]

The perceptron algorithm has been developed further, and algorithms using "hidden layers" and non-linear activation functions have been introduced. A multilayer perceptron neural network with one hidden layer could be described as follows:

\[
NN_{MLP1}(x) = g(xW^1 + b^1)W^2 + b^2
\]

where \(x\) is the input vector, \(W^1\) and \(W^2\) weight matrices for respectively layer 1 and 2, \(b^1\) and \(b^2\) bias terms for respectively layer 1 and 2 and \(g\) the non-linear activation function. Commonly used non-linear activation functions are e.g. the sigmoid function (2.4) and the tanh (2.5) function:

\[
S(t) = \frac{1}{1 + e^{-t}}
\]

\[
S(t) = \frac{1}{1 + e^{-t}}
\]

5
\[\tanh(t) = \frac{2}{1 + e^{-2t}} - 1\]  \hspace{1cm} (2.5)

When using a vector \(w\), as in (2.1), the output will be a single value. If we instead use a matrix \(W\), as in (2.3), the output will be a vector. Each column in the matrix \(W\) is containing values that are used to compute the value of the respective position in the output vector. The values of the output vector will then represent e.g. probabilities for different classes. This is called a multiclass neural network. The values in the weight matrices are decided by training. A common technique for doing this is backpropagation. We will not go into the details of the training process as this is out of the scope for our focus here.

A neural network is said to be ‘deep’ when it has two or more hidden layers (Goldberg 2015). We can add more linear transformations and activation functions to (3), to make a neural network with 2 hidden layers:

\[NN_{MLP2}(x) = g^2(g^1(xW^1 + b^1)W^2 + b^2)W^3 + b^3\]  \hspace{1cm} (2.6)

The concept can be used to further experiment with multiple layers and different activation functions.

### 2.2.1 Input layer

When using neural networks for NLP problems, one of the difficulties have been to represent the input layer in a suitable way. The input to neural networks are vectors, and different approaches have been proposed to solve the problem of how to represent words as vectors. Common to the various approaches is that they are based on the distributional hypothesis. We will in this section first explain the distributional hypothesis. Secondly, we will look at two main approaches for representing words as vectors: count-based models and prediction-based models.

Distributional semantics is the theory of representing words based on their distributional properties. The core hypothesis in distributional semantics, known as the distributional hypothesis, states that words with similar meaning occur in similar contexts. Words occurring in similar contexts, should also have similar representations. One common way of representing context distributions is using vector space models. Vector space models are algebraic models used to represent contextual distributions of words as vectors. Similarity in meaning between word vectors can then be calculated by using geometric techniques such as euclidean distance or cosine similarity.

The traditional way of representing words in distributional semantics is
by obtaining word vectors using count-based models. A word vector is a n-dimensional vector for representing a word. Traditionally, word vector estimation has largely been made by frequency counting. Every word vector will then be as long as the vocabulary, and the elements of the vector will be representing counts of co-occurrence with other words (Jurafsky and Martin 2000). This count-based method is low-cost and plain, but it does also have some disadvantages. The method will often require a smoothing function to be applied to the word vectors. The smoothing function is applied to avoid zero probabilities when using the vectors in statistical computations. When having high-dimensional vectors with a lot of zero or low probabilities, there will also be a problem with vector sparsity. This is not necessarily a problem in itself, but as an input to an ANN it is beneficial with as low dimension as possible.

As a result of the problems with using count-based models for representing words, the idea of representing words as dense vectors in a low dimensional space, so-called word embeddings, came up. Different methods for doing this is proposed, and the methods are collectively referred to as prediction-based models. The prediction-based models are, like the count-based models, based on the words’ distributional properties. The models uses neural networks themselves for making dense, low-dimensional word vectors. We will in the following paragraph describe two main methods for creating word embeddings: GloVe and word2vec.

GloVe is an unsupervised learning algorithm for obtaining representations for words in terms of vectors. The training is done on global co-occurrence statistics for words in the corpus. As such it actually represents a middle ground of count-and prediction-based models. The training objective is to create word vectors such that the dot product of two words equals the logarithm of the probability of the words’ co-occurrence (Pennington, Socher and Manning 2014).

Mikolov et al. (2013) propose two different model architectures for computing vector representations of words. The first model they propose is the continuous bag-of-words (CBOW) model. The CBOW model takes a window of words, and learns to predict the word in the middle of the window using the remaining words. This is done by taking the sum of the vector representations of the remaining words. The training of the models is done by using neural networks. The second model they propose is the continuous skip-gram model. This architecture is similar to CBOW, but instead of predicting a word based on the context window, it tries to maximize classification of a word based on another word in the same sentence (Mikolov et al. 2013). It does, given the middle word, calculate the probability of other words being context words. We can say that the CBOW architecture predicts the word based on the context, while the skip-gram
architecture predicts the surrounding words given a word.

The word vectors could be used as input to a neural network for e.g. solving a text classification problem. For sentence classification, the length of word vectors are especially important because the vectors often need to be concatenated. This makes the prediction-based models more suitable for our project. There has also been done some research where the accuracy of the models has been considered. Baroni, Dinu and Kruszewski (2014) did a comparison of prediction-based models and count-based models. The results showed that prediction-based models outperform count-based models on all tasks. In their study, they used the word2vec toolkit with a CBOW approach.

2.2.2 CNN architecture

There are many different architectures used for neural networks. We will focus on convolutional neural networks (CNN) because we are working with sentence classification, and this architecture has been shown to be a good alternative for this kind of classification task. A convolutional neural network is a type of feed-forward artificial neural network. The convolutional approach was first introduced by Waibel et al. (1989), and is also called Time-Delay Neural Networks (TDNN). The model was used for phoneme recognition.

A basic CNN consists of a convolution part and a pooling part. We will first describe the convolution part. Each sentence given as input is first partitioned into each possible window of a chosen size. Padding is added in the start and at the end of the sentence. A sentence of length 4 and a window size of 3, will e.g. result in 6 windows. The words are translated into word vectors (embeddings), and the word vectors for each window are concatenated. A convolution filter is then applied to each of these window vectors. The convolution filter is a matrix where the number of rows equals the length of the concatenated window vector and the number of columns equals the number of features to capture. Each window vector is transferred through the filter, resulting in (number of windows) vectors of the same size.

Then, a pooling operation is applied. The most common pooling operation is the max-over-time pooling operation.

$$v_t = \max(f_t[i]) \quad \forall t$$

The max-over-time pooling operation outputs a single vector $v$ where each value $v_t$ represents the maximum value over all windows for feature $i$. $f_t$ represents the window vector belonging to window $t$. Another pooling
operation which are commonly used is average pooling. This simply takes the average of the values over all windows, instead of the max.

Collobert et al. (2011) describes a neural network including convolutional layers, and uses this on a range of NLP tasks, like the task of semantic role labeling. The filter for each convolutional layer of the network is applied to each possible window of a sequence of words. The aim of the convolutional layers is to extract local features around each possible window of size $d_{\text{win}}$. They use multiple convolutional layers, and the $t'$th output column of the $l'$th layer is computed as follows:

$$[f^{l}]_{t} = W^{l}[f^{l-1}]_{t} + b^{l} \quad \forall t$$  \hspace{1cm} (2.8)

Where $W^{l}$ is the weight matrix of layer $l$ and $b^{l}$ is the bias for layer $l$. The same weight matrix is used across all windows of the sequence.

Collobert et al. (2011) uses a max-over-time pooling operation, as described earlier, after the convolutional layers. This ensures a global feature vector where the most useful local features produced by the convolutional layers are captured.

### 2.3 Deep learning for SA

Deep learning (with a convolutional approach) have reached interesting results in NLP tasks, including sentiment analysis, in recent years. We will in the following paragraphs take a closer look at two pioneering studies done on the task of sentiment analysis using deep learning. The reason why we want to look at the task of sentiment analysis in particular is that it is closely relates to the task of threat detection. Sentiment analysis is, similar to threat detection, often performed on sentence level.

#### 2.3.1 Kalchbrenner, Grefenstette and Blunsom 2014

Kalchbrenner, Grefenstette and Blunsom (2014) describes a convolutional neural network using dynamic k-Max pooling (DCNN). The DCNN is tested on different sentiment analysis tasks: small-scale binary and multi-class sentiment prediction, six-way question classification and twitter sentiment prediction by distant supervision. They use word embeddings that are optimized during training, and construct a sentence matrix $s$ by con-
catenating these word embeddings. The convolutional layers are represented using matrices with weights that are computed during training. An illustration of the architecture is given in figure 2.1.

The difference from the work of Collobert et al. (2011) is that they use a k-max pooling operation between the convolutional layer and the non-linear feature function. The k-max pooling operation makes it possible to extract the k most active features in the sequence, while also preserving the order of the features. The k-max pooling operation takes the k maximum values of each row in the matrix to construct a new matrix. For a matrix with dimensions $s \times m$, the resulting matrix after a k-max pooling operation will have dimensions $k \times m$. A dynamic k-max pooling operation is a k-max pooling operation where $k$ is a function of the length of the sentence and the depth of the network. The pooling parameter is modeled as follows:

$$k_l = \max(k_{\text{top}}, \text{upper}((L - l)/L) \times s)$$  \hspace{1cm} (2.9)

where $l$ is the index of the current convolutional layer and $L$ is the total number of convolutional layers in the network. $s$ represents the sentence length and $k_{\text{top}}$ is a fixed max value. $\text{upper}$ means that the computed value is rounded upwards to nearest whole number.

The DCNN performs well on all the tasks and data sets tested for sentiment analysis. The methods that they compare the results with are mostly traditional methods such as support vector machines, different versions of Maximum Entropy, but also a Max-TDNN. The DCNN performs better on all tasks in comparison to these.

### 2.3.2 Kim 2014

Kim (2014) uses a variant of the CNN architecture described by Collobert et al. (2011). The tasks the model is tested on includes different sentiment analysis tasks, such as classifying positive/negative movie reviews, classifying sentences as subjective or objective, classifying sentences into 6 different question types and classifying customer reviews as positive/negative. They describe the model as follows:

A sentence of length $n$ is represented as

$$x_{i:n} = x_1 \circ x_2 \circ ... \circ x_n$$  \hspace{1cm} (2.10)

where $x_i \in R^k$ is the vector of dimension $k$ representing word $i$ in the sentence, and $\circ$ is the concatenation operator. The convolutional operation requires a filter $W \in R^{hk}$, where $h$ is the size of the window. The filter is
Figure 2.1: Illustration of the architecture for the model by Kalchbrenner, Grefenstette and Blunsom (2014). A DCNN for the seven word input sentence. Word embeddings have size $d = 4$. The network has two convolutional layers with two feature maps each. The widths of the filters at the two layers are respectively 3 and 2. The (dynamic) k-max pooling layers have values $k$ of 5 and 3. Illustration: Kalchbrenner, Grefenstette and Blunsom (2014).
applied to every possible window of the sentence. A new feature $c_i$ is then produced from a window of words $x_{i:h-1}$ by:

$$c_i = f(W \times x_{i:h-1} + b)$$  \hspace{1cm} (2.11)

where $b$ is the bias term and $f$ is a non-linear function. When the filter is applied to every possible window of the sentence, this will produce what they call a "feature map":

$$c = [c_1, c_2, ..., c_{n-h+1}]$$  \hspace{1cm} (2.12)

Like Collobert et al., they apply a max-over-time pooling operation. This means taking the maximum value of the "feature map" $c$ as the feature corresponding to this particular filter. The CNN model uses multiple filters with possibly various window sizes. The features form the penultimate layer and are passed to a fully connected softmax layer whose output is the probability distribution over labels.

Kim (2014) does, in contrast to Collobert et al. (2011), also experiment with having two "channels" of word vectors: one that is kept static throughout training and one that is fine-tuned via backpropagation. Each filter is applied to both channels, and the results are added to calculate $c_i$ (feature for filter $i$). An illustration of the model architecture with two "channels" are given in figure 2.2.

Another difference is that they employ so-called dropout on the penultimate layer. Dropout is a regularization technique used during training. They do this by using a masking vector of Bernoulli random variables.

They compare their results to quite a large range of other methods. The multichannel model outperforms the other models on the tasks SST-2 (Task of predicting movie reviews with binary labels) and CR (Task of predicting customer reviews of various products as positive/negative reviews.)
Figure 2.2: Illustration of the architecture for the model by Kim (2014). Model architecture with two channels for an example sentence. Illustration: Kim (2014).
Chapter 3

Data set and tools

In this chapter we will first present in detail the data set we will use for our task; a pre-annotated corpus of YouTube comments. Secondly, we will describe two main tools that we are going to use for solving our task; Keras and GloVe.

3.1 YouTube data set

We will in the following section describe the data set we are going to use in our project. The data set was collected by Hammer (2014), and further used by Wester et al. (2016). The data set consists of user generated text, in the form of comments from the well-known video platform YouTube. The comments are collected from 8 different videos. The videos chosen are videos with typically triggering topics and content, such as politics, immigration and religion. Each comment consists of a number of sentences and the sentences are manually annotated as containing threats of violence or not. This is represented with 1 (threat) and 0 (non-threat) in the data set. There is no formal definition of threat of violence used when annotating the sentences. There is e.g. no distinction between threats of violence and supporting threats of violence, and the implicit definition used seems to be relatively broad (Wester et al. 2016). The inter-annotator agreement was reported to be 98% by Hammer (2014). The set used consisted of 100 randomly chosen sentences annotated as non-threats and 20 randomly chosen sentences annotated as threats by the first annotator. The inter-annotator agreement was calculated using simple overlap, and not a measure correcting for unbalanced data.

The data set also contains information about which user posted which comment, using a commenter ID. Some examples from the data set are given in Figure 3.1. When we observe the data set, we see that xenophobia and racism are repeating elements in the comments.
Table 3.1: YouTube Threat Corpus count data

<table>
<thead>
<tr>
<th></th>
<th>Sentences</th>
<th>Comments</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threats</td>
<td>1,384</td>
<td>1,285</td>
<td>992</td>
</tr>
<tr>
<td>Non-threats</td>
<td>27,259</td>
<td>8,560</td>
<td>4,491</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>28,643</strong></td>
<td><strong>9,845</strong></td>
<td><strong>5,483</strong></td>
</tr>
</tbody>
</table>

Splitting of comments in the data set into sentences were done manually by Hammer (2014). Normalization and tokenization of the data set were done by Wester et al. (2016). We will in the following describe how the normalization and tokenization is done.

Normalization is done based on the assumption that lower-casing all-caps words will not remove relevant information. However, it is taken into account that all-caps words is an indicator of violent threats. It is therefore made an exception for all-caps words occurring as first word in a sentence. The following rules are used for normalization: "If a word is all caps, we lowercase it, except if it is the first word in the sentence, in which case it is capitalized. If words are capitalized, or in any other way mixed cased, we do not change their capitalization, no matter where in a sentence the word occurs." (Wester et al. 2016). For tokenization, the tokenizer in a toolkit called spaCy is used.

In table 3.1, we present counts for sentences, comments and users posting from the data set. The corpus contains a total of 9845 comments. The number of comments annotated with "containing threats of violence" is 1285. If we look at the numbers at sentence level, there are a total of 27259 sentences, where 1384 contains threats. This means that mostly, the sentences containing threats are not gathered in the same comments. We can also see that the violent comments are mostly posted by different users (992 different users posting 1285 violent comments).

An aspect of the data set that is worth mentioning, is the unequal distribution between comments/sentences containing threats and comments/sentences not containing threats. This is something that we have to take into account when training our model.
3.2 Tools

3.2.1 Keras

As a main tool for creating our models, we have chosen to use a library called Keras. There are several reasons why we want to build the models our self, instead of simply using e.g. the implementation by Kim (2014), which is created using Theano. When creating the model our self, we have an unique opportunity to start from scratch, and make our own choices based on our specific task while creating the model. We also want to experiment with different changes and extensions not implemented by Kim (2014), and this is much easier to do when using Keras.

Keras is a high-level neural networks library written in Python. The library is capable of running on top of either TensorFlow or Theano, which both are frameworks providing the ability of fast computation of mathematical expressions. We are going to use Keras to create our models, with Theano as backend. Keras provides a lot of possibilities also for creating convolutional neural network models. We will now go through the different layers of a CNN, and look at what possibilities Keras offers for the different layers of the model.

Embedding layer:
Keras provides different options for the input layer, including the ability of using pre-trained word embeddings and the ability to choose to prevent the nodes from being updated during training. For the embedding layer, there are 3 options:

1. No use of pre-trained embeddings. The initial embedding weights are randomly chosen, and the weights are updated during training.
2. Use of pre-trained embeddings, prevent the nodes from being updated during training. Pre-trained embeddings are given as input to the embedding layer, and the weights are kept static through the training.
3. Use of pre-trained embeddings, update nodes during training. Pre-trained embeddings are given as input to the embedding layer, and the weights are fine tuned during training.

We will experiment with all three of these options. Kim (2014) also investigates the difference between updating the nodes during training, or keeping them static. His results show that the version where the nodes are updated is largely the best.

Relevant parameters for the Embedding layer are "output_dim", where the dimension of the embeddings are defined, "weights", which is an optional parameter depending on whether you want to use pre-trained em-
beddings, and "trainable", where you decide whether you want the embedding weights to be updated during training.

**Convolution layer:**
For the convolution layer, Keras provides options for tuning different parameters. Relevant parameters are "nb_filter", which defines the number of filter maps, "filter length", which defines the size of the filter window and "activation", where the desired activation function is provided. There are several built-in activation functions such as softmax, rectified linear units and sigmoid. The layer also has a parameter called "border_mode", which defines whether to pad the input sequences.

**MaxPooling layer:**
Relevant parameters for the MaxPooling layer is "pool_length". This parameter is set to the factor by which to downscale. E.g. will pool length 2 halve the input.

**Flattening layer:**
A flattening layer simplifies flattens the input to one dimension. A input of dimension (22, 4) will e.g. result in dimension (88).

**Dense layer:**
The Dense layer in keras is simply a fully connected layer, where each unit or neuron is connected to each neuron in the next layer. The relevant parameters for this layer is "output_dim", which defines the dimension of the output, and "activation", where you provide the activation function to use.

**Training** There are also different choices that can be made when training the model. This include number of epochs and loss function. Keras has several built-in loss function such as e.g. crossentropy, mean squared error and cosine proximity.

### 3.2.2 GloVe

As mentioned in the previous section, Keras has support for using pre-trained word vectors as input to the model. This is represented as an embedding layer in the Keras model. For the embedding layer, it is possible to switch a "trainable"-parameter to either True or False, which gives the opportunity to either keep the embedding vectors static or updated during training. We will experiment with both of these options.

We are going to use word embeddings that are pre-trained using GloVe
As also discussed in section 2.2.1, GloVe is a machine learning algorithm for representing words as vectors. The algorithm is unsupervised, and training is done using co-occurrence statistics from a chosen corpus. The statistics are represented as a global word-word co-occurrence matrix. The GloVe model is basically a log-bilinear model with a weighted least squares training objective. The main intuition behind the model is the simple observation that ratios of word-word co-occurrence probabilities have the potential for encoding some kind of meaning (Pennington, Socher and Manning 2014).

The corpus to use for training can be chosen, but GloVe also provides 4 different sets of pre-trained embeddings on their website:

- Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocabulary size, lower case, 50 dim, 100 dim, 200 dim and 300 dim vectors)
- Common Crawl (42B tokens, 1.9M vocabulary size, lower case, 300 dim. vectors)
- Common Crawl (840B tokens, 2.2M vocabulary size, preserved case, 300 dim. vectors)
- Twitter (2B tweets, 27B tokens, 1.2M vocabulary size, lower case, 25 dim, 50 dim, 100 dim and 200 dim vectors)

The different sets of pre-trained embeddings is one of the reasons why Glove was chosen for our experiments in stead of e.g. word2vec. The corpora mentioned above are all from the web, and may all be relevant for our experiments. We will therefore experiment with all of them.
Comment #188, Commenter #115
0 STOP THE MUSLIM INVASION!!!!!
0 DEFEND EUROPE, DEFEND THE WESTERN WORLD!!!!!
0 BAN INTERRACIAL MARRIAGE, BAN IMMIGRATION!!!!!
1 DEPORT THESE PEASANTS!!!!!

Comment #289, Commenter #194
1 deport ALL of them back and burn down all mosques...

Comment #654, Commenter #411
1 ....thats why I kill you and your father and your mother and you brother.
0 Then it works, just ask the Reynes of Castamere

Comment #864, Commenter #526
0 YAY, I love it.
1 Hells yea, kill 'em all!
0 Yea, Yea, Yea!

Comment #8182, Commenter #4682
0 Well if Europeans stop going to war in the process killing close to 70 million of their own people they would not need immigrants to replenish their work force.
0 If you want immigrants to become Germans perhaps you should start treating them as equals and if you want them to stay home then allow them to develope technologies that will need them to stay home and work.

Figure 3.1: Examples of YouTube comments from the data set
Chapter 4

Experimental setup

In this chapter we will go through different aspects considering the experimental setup. First, we will how our data set is split. This includes both splitting the data set in a development set and a held-out test set, and splitting the development set into blocks used for cross validation. We will also describe and discuss the use of cross validation for our experiments. Secondly, we will discuss the use of metrics for evaluation. We will look at 4 different metrics, describe what they represent, and discuss which ones that are appropriate to use for our experiments.

4.1 Data set splits and cross validation

Before training our model, we have to split our data into a development set and a test set. 1/5 of the data will be used as a held out test set. For comparability we will reuse the train/test split defined by Wester et al. (2016). Every fifth comment is picked out of our original set, and stored in a separate file. This data will be used at the end of all our experiments, to evaluate our final model. The numbers for the partition are shown in table 4.1. With a total of 9,845 comments, this means 7,876 comments will be used while developing our model, while 1,969 comments will be reserved for held out testing. There is also nearly a 1:5-partitioning of the sentences, with 80.2 % sentences in the development set, and 19.8 % in the held out test set.

The remaining data will be used while developing our model. Because we are doing 5-fold cross validation, we have to split the development data into 5 blocks. When doing this partitioning, our assumption is that comments originating from the same YouTube movie is located together in the data set. We have chosen to partition the data into contiguous blocks. The reason why we do this is that we want to keep the sentences belonging to the same comments together. We also want to keep sentences originating from the same movie together, as much as possible. This will make our
model more general and robust. It will also ensure a more realistic setting. We can see from table 4.2 that the sentences are divided evenly among the folds. The disadvantage with this way of splitting our development data into folds, is that it leads to a somewhat uneven distribution of the threats. We can see from table 4.2 that number of threats in the different folds ranges from 150 to as much as 344. We do, after all, think that the benefits is greater than the disadvantages, and we will stick with this data set split.

For every iteration of the 5-fold, four blocks of data is used for training our model, while the remaining is used for validation. When all iterations are done, the results for each iteration are averaged.

Cross-validation is a mechanism that is useful when having limited amounts of development data. Limited amounts of data is a common issue, especially in the field of language technology. The reason for this is that all data has to be manually annotated before using it in a neural network. Cross-validation is a good way to get the most out of the data that we have.

Cross-validation is also used for strengthening the network against over-fitting. n-fold cross-validation is done by partitioning our development data into k blocks (folds). The network will start training on the data in the first n-1 folds, and use fold number n for validation. This will be repeated n times, such that all the n folds is used for validation one time. When training and validation is done, the results from the n models are

<table>
<thead>
<tr>
<th></th>
<th>Comments</th>
<th>Sentences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development set</td>
<td>7,876</td>
<td>22,958</td>
</tr>
<tr>
<td>Held out test set</td>
<td>1,969</td>
<td>5,685</td>
</tr>
</tbody>
</table>

Table 4.1: Partition of data into development set and held out test set

<table>
<thead>
<tr>
<th></th>
<th>Sentences total</th>
<th>Sentences containing threat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fold 1</td>
<td>4,592</td>
<td>150</td>
</tr>
<tr>
<td>Fold 2</td>
<td>4,592</td>
<td>235</td>
</tr>
<tr>
<td>Fold 3</td>
<td>4,592</td>
<td>176</td>
</tr>
<tr>
<td>Fold 4</td>
<td>4,591</td>
<td>344</td>
</tr>
<tr>
<td>Fold 5</td>
<td>4,591</td>
<td>198</td>
</tr>
</tbody>
</table>

Table 4.2: Partition of development set into 5 folds for cross validation
averaged and reported. The reason why cross-validation is used is that it prevents us from making decisions for model architecture and parameter tuning based on training and validation done only on specific parts of our data set. Cross-validation helps us understand how the performance varies across data, instead of making us wonder whether we were (un)lucky with our choice of training/validation data.

When looking at this in the context of our data set, we have to consider the size of the data set, time used for creating the model and the gains of using cross-validation. Neural nets are expensive in terms of time and resources. We are therefore not using leave-one-out or 10+-fold, but 5-fold cross-validation.

Cross validation is only used while developing our model and tuning parameters. Before testing on the held-out test set, the model will be retrained on the entire training set.

### 4.2 Evaluation metrics

Before we start experimenting, we also have to decide on what metrics we are going to use for evaluating our results. There are several metrics to choose from, and we want to choose one that is suitable for our specific task. The goal for our model is to correctly classify as many threats as possible, but also not to wrongly classify too many non-threats as containing threats. We will especially have to think about the fact that our data set is unbalanced (few sentences containing threats compared to the ones not containing threats) when deciding on an evaluation metric. Since sentences in our task are either positive (containing threat) or negative (not containing threats), we will use only binary classifiers. To be able to compute and talk about the binary metrics, we have to introduce some concepts: true/false positives and true/false negatives. True positives (TP) are the cases where a threat is classified as a threat, while false positives (FP) are the cases where a non-threat is classified as a threat. False negatives (FN) are the cases where a threat is classified as a non-threat, while true negatives (TN) are the cases where a non-threat is classified as a non-threat. This can be represented in a so-called confusion matrix, shown in table 4.3.

The **accuracy** metric is the first metric we will look at. It represents the fraction of the sentences that are classified correct by the classifier:

\[
\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}
\]
Table 4.3: Confusion matrix with True/False Positives/Negatives

<table>
<thead>
<tr>
<th></th>
<th>Classified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Threat</td>
</tr>
<tr>
<td>Annotated</td>
<td>Threat</td>
</tr>
<tr>
<td></td>
<td>Non-threat</td>
</tr>
</tbody>
</table>

Accuracy is generally a metric used a lot, and can give much intuitive information about the results without too much computation. Because of our unbalanced data set, we have to think differently about the accuracy-metric. E.g. will a static model classifying all sentences in our data set as non-threat achieve an accuracy of 0.95. This is a seemingly high score, while the model is not good at all when it comes to discriminating threats from non-threats.

Second, we will discuss the **precision** metric. Precision represents the fraction of the sentences classified as positive (Threats) that are actually positive (Threats). This is a good metric, but misses out on the part of how many Threats that were actually found by the classifier. A classifier that correctly classifies 1 Threat, and do not classify any other sentences as threats, will receive a precision of 1.00.

\[
Precision = \frac{TP}{TP + FP}
\]

The **recall** metric represents the fraction of the sentences actually positive (Threats) that are classified as positive (Threats). This measure takes care the problem mentioned with the precision metric. It measures the fraction of sentences containing threats actually found by the classifier. This is not either a good classifier by itself. We can e.g. come up with an example of a classifier classifying all sentences as positive. This will result in a recall score of 1.00, which is actually not giving any information about how good the model are when it comes to distinguish the threats from the non-threats.

\[
Recall = \frac{TP}{TP + FN}
\]

Another commonly used metric is a weighted score between recall and precision, named **F-score** (also called F1-measure). This is a good metric to use also when having an unbalanced data set like ours. This metric gives the most information based on our goals, and we are going to use this metric as a main measure on how good our models are. We will also
report on the other metrics discussed, where it is appropriate.

$$F - score = 2 \times \frac{precision \times recall}{precision + recall}$$
Chapter 5

Preliminary experiments

In this chapter we will establish some baselines, and explain how our preliminary experiments are carried out. We will first present some non machine learning baselines. We will then present our CNN baseline model, which is the one we will use as a main baseline. We will then discuss two aspects of creating a neural network model; number of epochs to be carried out, and controlling for non-determinism. At least, we will also discuss a bug in our code resulting in interesting results.

5.1 Establishing some baselines

In order to evaluate new results, we need former science to compare our results to. The baseline for a problem is typically the simplest possible approach to the task. This is done to give an intuition about how difficult the problem is. One possibility is to use a majority class baseline. Another approach could be to create a simple bag-of-words model (BoW) with traditional Maximum entropy (MaxEnt) classifier. This is done and used as a baseline in the work of Wester et al. (2016), where the same development set as ours are used. The MaxEnt model received an F-score of 0.6123, with a precision of 0.6777 and a recall of 0.5585 (Wester et al. 2016). Another approach could be to use an other simple model as baseline.

For our project we will use a simple convolutional neural network set-up as baseline, as well as some non machine learning classifiers. The choices done to create the models will be further explained in the next sections. The baseline models will be used to evaluate whether the changes to the model causes better results or not.

5.1.1 Non machine learning classifiers

We will in this section compute baseline results for 4 different non machine learning classifiers: Majority class classifier, minority class classifier, ran-
Table 5.1: Data set statistics

<table>
<thead>
<tr>
<th></th>
<th>Number</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of sentences</td>
<td>28643</td>
<td></td>
</tr>
<tr>
<td>Sentences annotated positive</td>
<td>1,384</td>
<td>4.83</td>
</tr>
<tr>
<td>Sentences annotated negative</td>
<td>27,259</td>
<td>95.17</td>
</tr>
</tbody>
</table>

Table 5.2: Confusion matrix for the majority class classifier

<table>
<thead>
<tr>
<th>Classified</th>
<th>Threat</th>
<th>Non-threat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotated</td>
<td>Threat</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Non-threat</td>
<td>0</td>
</tr>
</tbody>
</table>

dom guess classifier and weighted guess classifier. The results from such non machine learning classifiers can contribute in giving an intuition on how difficult the task is, and may help figuring out how to solve the task in the best way. We start out by repeating some statistics from our data set in table 5.1.

We will now illustrate how the different classifiers work using confusion matrices. The confusion matrix will show the relationship between True positives, False positives, True negatives and False negatives. We start out with the majority class classifier. The majority class classifier simply classifies all sentences as the majority class (non-threat). Confusion matrix for the majority class classifier is shown in table 5.2. The minority class classifier is the opposite of the majority class classifier, and will classify all sentences as threat. Confusion matrix for the minority class classifier is shown in table 5.3. The random guess classifier will randomly classify any sentence as threat or non-threat. This means half the sentences will be classified as threat and half the sentences will be classified as non-threat (in theory). Confusion matrix for the random guess classifier is shown in table 5.4. The weighted guess classifier will classify a sentence as threat with a probability of %threats in the data set, and non-threat with a probability of %non-threat in the data set. Confusion matrix for the weighted guess classifier is shown in table 5.5.

The results for all classifiers are presented, using the four different metrics described in 4.1.4, in table 5.6. The computations are done using the information provided in the confusion matrices and formulas presented in 4.1.4.
The non machine learning classifiers shows that there is no intuitive "easy" way to get high F-scores because of the unbalanced data set. We get a max accuracy of 95.17 by using the majority class classifier, and a max F-score of no more than 9.21, using the minority class classifier.

5.1.2 CNN baseline model

We will in this section describe the model and provide results for the model we are going to use as our main baseline. The baseline model will also be used as a base for further experiments.

As our baseline model, we are going to create a simplest possible model for our task. We are going to use a single layer convolutional network with one hidden layer. Further, we will use 5-fold cross validation and 5 epochs. This is chosen based on the experiments and discussion in section 4.5.

Our data set is already tokenized, and we will not perform any additional processing on this. When reading our data, all words are changed to lower case. Also, a word index is created when reading in our training data. When reading our validation data, all words not in the word index are changed to "*_UNKNOWN_*", and will be initialized with the same random word vector. All sentences are padded to fit the maximum sentence length.

For our baseline model, we will use randomly initialized word embeddings. The embedding weights are random numbers uniformly distrib-
<table>
<thead>
<tr>
<th>Classified</th>
<th>Threat</th>
<th>Non-threat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotated</td>
<td>Threat</td>
<td>28643<em>0.0483</em>0.0483</td>
</tr>
<tr>
<td></td>
<td>Non-threat</td>
<td>28643<em>0.0483</em>0.9517</td>
</tr>
</tbody>
</table>

Table 5.5: Confusion matrix for the weighted guess classifier

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>F-score</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Majority class</td>
<td>95.17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Minority class</td>
<td>4.83</td>
<td>9.21</td>
<td>100</td>
</tr>
<tr>
<td>Random guess</td>
<td>50.0</td>
<td>8.81</td>
<td>50.0</td>
</tr>
<tr>
<td>Weighted guess</td>
<td>90.80</td>
<td>4.83</td>
<td>4.83</td>
</tr>
</tbody>
</table>

Table 5.6: Non machine learning model results

uted in the interval (-0.05, 0.05). This is the default setting in Keras.

We will further explain the steps throughout the network. For our embedding layer, we will use internal embeddings with size 100. This means we are not using any pre-trained embeddings in our baseline model. Pre-trained embeddings will be introduced in the first step of creating our more complex CNN model. For the convolutional layer we use a window size of 3. The filter is of size 32, and rectified linear units is used as activation function. We have then added a max-pooling layer with pooling length 2, which will half the dimensionality of the convolutional layer. After the max-pooling layer, the output is reduced to one dimension. A fully connected layer with size 250 is added, and rectified linear units activation function is applied. Then a new fully connected output layer is added, and the size is reduced to 1 before a sigmoid function is applied. The model is trained to minimize the cross-entropy. We are further using the so-called ADAM optimizer in Keras. The choices done creating the model were made based on default settings in Keras and settings proposed in Keras tutorials.

The results for our baseline model are shown in the table 5.7, together with results from the BoW model described above and the majority class classifier. We can see that the BoW model actually provides better results in terms of F-score than our simple CNN model.
Before we can fix our baseline model, we have to decide the number of epochs to run when training our model. Number of epochs is an important choice to do when training a NN. One epoch is one forward pass and one backward pass of all the training data through the network. The network weights are updated for each epoch, and the number of epochs represents the number of times the training data is used to update the network weights.

To make the decision of number of epochs, we have done a run with 200 epochs, where we compare the train data set results to the validation data set results. Our held-out data set is still kept untouched. We have used the same configurations that we plan to use for our baseline model, and 5-fold cross-validation as described in section 4.1. F-score, accuracy and loss are used as metrics, and plots for the runs are shown in respectively figure 5.1, 5.2 and 5.3.

The goal of choosing the correct number of epochs is to prevent both over-fitting and under-fitting. If we look at the f-score-graph in figure 5.1, we can see that the graph for validation results stabilizes after around 5 epochs. For accuracy, the model has a top after around 10 epochs, and is then getting lower scores. For the loss graph, we get a somewhat different result. The model gets its lowest loss-score around 3-5 epochs, and it keeps increasing. If we look at all the scores together, it seems like around 3-5 epochs is a reasonable choice for our baseline.

### Table 5.7: Baseline model results

<table>
<thead>
<tr>
<th>Model</th>
<th>F-score</th>
<th>Accuracy</th>
<th>Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN with simple set-up</td>
<td>50.16</td>
<td>95.81</td>
<td>19.66</td>
</tr>
<tr>
<td>BoW with MaxEnt</td>
<td>61.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Majority class</td>
<td>0</td>
<td>95.17</td>
<td></td>
</tr>
</tbody>
</table>

5.3 **Controlling for non-determinism**

When starting the experiments, our analysis of our first rounds of preliminary experiments showed that some controlling for non-determinism had to be done. There are several sources of randomness in the training of a neural network. The randomness happens because of stochastic
Figure 5.1: F-score for baseline run over 200 epochs. Comparison between train and validation results over the epochs

Figure 5.2: Accuracy for baseline run over 200 epochs. Comparison between train and validation results over the epochs
Figure 5.3: Loss for baseline run over 200 epochs. Comparison between train and validation results over the epochs

variables used when initializing random embeddings and weights for the other layers. Also, a non-deterministic optimizer function (ADAM) is used, although this only causes minor change. Some random seeds can therefore yield better results than others. In order to quantify the effect of non-determinism we have done 10 example runs with different seeds for the baseline model to illustrate the issue. A more extensive study (using hundreds of different random seeds) of the issue of seed selection is done in Erhan et al. (2010). They also provide a histogram with results. Results for our runs for the baseline model are shown in table 5.8. The F-scores vary from 45.11 to 51.67 with a mean of 48.34. The standard deviation for the F-score, using the sample provided in table 5.8, is 1.70.

We have also done 10 runs with a model kept static, where only the embeddings are randomly initialized. For these runs, a seed is used to keep all parts of the model static except from the embedding initialization. This is done to isolate the non-determinism caused by the embedding initialization. When keeping the model static, the standard deviation is dramatically changed to as low as 0.37. The results from the 10 runs are shown in table 5.9.

Because of the illustrated problem with non-determinism, to get more easily comparable results, we are going to use the library function
<table>
<thead>
<tr>
<th>Run</th>
<th>F-score</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run 1</td>
<td>48.74</td>
<td>95.98</td>
</tr>
<tr>
<td>Run 2</td>
<td>48.13</td>
<td>95.76</td>
</tr>
<tr>
<td>Run 3</td>
<td>47.83</td>
<td>95.76</td>
</tr>
<tr>
<td>Run 4</td>
<td>47.89</td>
<td>95.58</td>
</tr>
<tr>
<td>Run 5</td>
<td>45.11</td>
<td>95.81</td>
</tr>
<tr>
<td>Run 6</td>
<td>47.27</td>
<td>95.86</td>
</tr>
<tr>
<td>Run 7</td>
<td>47.86</td>
<td>95.77</td>
</tr>
<tr>
<td>Run 8</td>
<td>49.31</td>
<td>95.97</td>
</tr>
<tr>
<td>Run 9</td>
<td>49.57</td>
<td>95.97</td>
</tr>
<tr>
<td>Run 10</td>
<td>51.67</td>
<td>95.99</td>
</tr>
</tbody>
</table>

Table 5.8: 10 runs of baseline model with different seeds, 5 epochs

<table>
<thead>
<tr>
<th>Run</th>
<th>F-score</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run 1</td>
<td>50.32</td>
<td>95.73</td>
</tr>
<tr>
<td>Run 2</td>
<td>50.17</td>
<td>95.85</td>
</tr>
<tr>
<td>Run 3</td>
<td>49.72</td>
<td>95.67</td>
</tr>
<tr>
<td>Run 4</td>
<td>50.07</td>
<td>95.79</td>
</tr>
<tr>
<td>Run 5</td>
<td>50.54</td>
<td>95.84</td>
</tr>
<tr>
<td>Run 6</td>
<td>49.85</td>
<td>95.85</td>
</tr>
<tr>
<td>Run 7</td>
<td>49.27</td>
<td>95.78</td>
</tr>
<tr>
<td>Run 8</td>
<td>49.97</td>
<td>95.67</td>
</tr>
<tr>
<td>Run 9</td>
<td>50.26</td>
<td>95.93</td>
</tr>
<tr>
<td>Run 10</td>
<td>50.33</td>
<td>95.83</td>
</tr>
</tbody>
</table>

Table 5.9: 10 runs of baseline model with different seeds for initialization, same seed for model, 5 epochs

Numpy.Random.Seed(). We simply set and fix the seed before doing any use of imports from Keras or Numpy. The seed makes us able to do several experiments with the same stochastic set-up. This means we can freeze all stochastic variables for later experiments. For all our experiments we will use the same seed, and the stochastic prerequisites for each experiment will therefore be consequently equal. This facilitates isolation the effect of other parameters that we will systematically tune. The seed is chosen randomly, and is not based on performance. According to Bengio (2012), the choice of random seed only has a slight effect on the result, and can mostly be ignored in general or for most of the hyper-parameter search process.

What is also worth mentioning, is that the standard deviation probably will be lower when we introduce pre-trained embeddings to our model. This will only make a minor difference, because the input initialization is
only a small part of the non-determinism in our model (5.9).

The cross validation assignment is also kept static, and is partitioned as described in 4.1.1.

We can also choose to keep the random embeddings static throughout the experiments, such that a word always will be assigned to the same randomly generated embedding every run. This will cause a standard deviation of (almost) 0. This is not possible for experiments with different embedding dimension, where there is no way of keeping the embeddings static. We will for these experiments take the standard deviation of 0.37 into account when interpreting the results.

5.4 Bug causing extra training of embeddings

While doing the CNN baseline experiments, we discovered a bug causing extra training of the embedding layer. The bug caused the embedding layer to not be reset between each of the loops in cross validation. This means that (except for the first loop) the embeddings were trained using the data also used for validation. This would of course not result in any reliable results, but the results were however unexpectedly high. We also investigated the results for each single loop of the cross validation, and observed that the results got better for every loop. This was not an surprise, as the embeddings became more over-fitted for each iteration.

Because of the unexpectedly high results achieved when having this bug in our code, we think it would be interesting to follow this up with further experimenting. We belive that more iterations of training the embedding layer could cause better results. We will therefore do an experiment when introducing our held out data set, where we will use this experience. We will of course not touch the held out data set while training our model, but we will do more iterations on the development set to train the embedding layer. The results for these experiments will be given in section 7.2.
Chapter 6
Experiments

In this chapter we will do experiments using Keras (described in section 3.2.1). We will use the CNN baseline model described in the previous chapter as a base for our experiments. We will start out by introducing pre-trained embeddings to our model. Secondly, we will investigate methods for mitigating the problem of unbalanced classes, before we do some more experimenting with word embeddings. We will then look into different aspects of our model architecture; window sizes, number of feature maps, concatenation of convolutional layers, dropout, optimizers, batch size and different number of both hidden and convolutional layers. At least, we will describe our final model in detail, and do an analysis of errors made by the model.

6.1 Adding an embedding layer

We will in this section look at the effect of adding pre-trained embeddings to our model. The embeddings are obtained from the GloVe website, and have been trained using the GloVe algorithm (Pennington, Socher and Manning 2014) on an unlabeled text corpus. We will start out experimenting with the impact of adding an embedding layer, and updating the embedding-weights during training or not.

As mentioned in section 3.2.2, Keras has functionality for adding an embedding layer with pre-trained embedding vectors to a model. For our baseline model, the embedding layer are simply vectors initialized with random values. These random values are updated during training. We will now experiment with initializing these embedding weights using pre-trained word vectors. We will also dig deeper into different ways of initializing word weights for words not present in the embedding corpus.

We will start by using the pre-trained GloVe embedding set trained on the Wikipedia + Gigawords corpus. We will later, in section 6.3, also look
Table 6.1: Model w/ embeddings trained on Wikipedia+Gigawords

<table>
<thead>
<tr>
<th>Model Description</th>
<th>F-score</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN with simple set-up</td>
<td>50.16</td>
<td>95.81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ embeddings 100, static</td>
<td>41.42</td>
<td>95.76</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ embeddings 100, non-static</td>
<td>50.88</td>
<td>96.29</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

at different embedding dimensions, and see what effect this has on our model. We will then look at the four different embedding sets mentioned in section 3.2.2 trained on other corpora, and what difference they make to the results of our model.

### 6.1.1 Static VS non-static embedding weights

We will first do an experiment which investigates the impact of training the embedding weights. Keras has a built-in parameter for the embedding layer where one can choose to let the embedding weights be kept static or not. For the static model, all word vectors are initialized with pre-trained word vectors from GloVe. Words not present in the word vector corpus are randomly initialized. All word vectors are kept static, while weights in the other layers are updated during training. For the non-static model, the word vector initialization is the same, but the embedding weights are updated during training.

We will run the experiments in this subsection with the pre-trained embeddings trained on the Wikipedia + Gigaword data set. This embedding set is trained on a corpus consisting of data from Wikipedia 2014 and a large news corpus called Gigaword 5. The data set is lower-cased, consists of 6 billion tokens, and the size of the vocabulary is 400K. We will later also test the pre-trained embeddings trained on the other data sets. We will first run experiments with embeddings of dimension 100. Later on, we will check what impact the embedding dimension has, and experiment with other embedding dimensions.

We can see from table 6.1 that adding the 100 dimension Wikipedia+Gigawords embeddings does make a substantial improvement on our model both for F-score and accuracy. We will do some more experimenting using embeddings with different dimensions and embeddings trained on different corpora in the next section.

We can see that the difference from not updating word vector weights
during training, to doing so, is huge. For the static model, the F-score is lower than the model without pre-trained embeddings, and also not much higher than the random/weighted guess classifier presented in table 5.6. The embeddings are not updated during training, which means all words not present in the embedding data set will be kept random throughout the training. This experiment shows that updating word vector weights is extremely important, and we will in the upcoming experiments always use the non-static version.

### 6.1.2 Unknown word initialization

When adding pre-trained embeddings to our model, we also have to deal with the fact that there are words occurring in our data set, that do not occur in the embedding data. We call these words "unknown" words. We will in the following explain our current method for solving this, and later on propose a second method. Unknown words occurring in our training data, but not in the embedding data set, have been included in the word index, and assigned to a random embedding vector with approximately the same variance as the embedding data. Unknown words occurring in the test data, but not in the train data, have been initialized with a common random embedding vector "*__UNKNOWN__*", also with approximately the same variance as the embedding data.

We wanted to find out how comprehensive the occurrence of these unknown words were, and made a count of unknown words in the different pre-trained embedding sets. The counts are shown in table 6.2. We can conclude with that most unknown words are unique, and that the number of unknown words is generally high. We do e.g. have 10,608 unknown tokens out of 371,198 in the dev. data for the Wikipedia + Gigawords embeddings. This also partly explains the low scores for the static embedding model described in the previous section. A high number of unknown words means high number of randomly initialized word embeddings which are not updated during training. Our prediction is that the embedding sets with the lowest number of unknown words will get the best results. Based on table 6.2, the common crawl lower-case embedding set will probably give the best results. This will be experimented with in section 6.3.2.

We have also extracted some examples of unknown words, presented in figure 5.1. The unknown words are mainly spelling mistakes, unknown languages, wrong use of punctuation, non-existing words and tokenization errors.
Table 6.2: Number of tokens/unique words present in development data, not present in embedding corpora

<table>
<thead>
<tr>
<th>Corpora</th>
<th>Unknown tokens</th>
<th>Unknown unique words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common crawl uncased</td>
<td>7,133</td>
<td>5,591</td>
</tr>
<tr>
<td>Common crawl cased</td>
<td>7,656</td>
<td>6,209</td>
</tr>
<tr>
<td>Wikipedia + Gigaword</td>
<td>10,608</td>
<td>8,109</td>
</tr>
<tr>
<td>Twitter</td>
<td>13,077</td>
<td>7,433</td>
</tr>
<tr>
<td>Total tokens in development data</td>
<td>371,198</td>
<td></td>
</tr>
<tr>
<td>Unique tokens in development data</td>
<td></td>
<td>23,717</td>
</tr>
</tbody>
</table>

Table 6.2: Number of tokens/unique words present in development data, not present in embedding corpora

Figure 6.1: Examples of unknown words from training set data

We have come up with two different ways of doing initialization of unknown words occurring in training data:

1. Different random initialization of each single unknown word in the training data. Words in the validation data, not present in the word index, are changed to "*__UNKNOWN__*", and will be initialized with the same random word vector. This is the method used until now. Most unknown words have low frequency. Many of them will not be updated at all, and keep the random initialized weights.

2. One common random embedding vector "*__UNKNOWN__*" given to all unknown words in the training data. This common "*__UNKNOWN__*"-vector is updated during training. Words in the validation data, not present in the word index, will be given the UNKNOWN word embedding. The UNKNOWN word embedding will represent "a
Table 6.3: Model w/ embeddings trained on Wikipedia+Gigawords dim 100, two different initialization methods for unknown words.

<table>
<thead>
<tr>
<th>Method</th>
<th>F-score</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN with simple set-up</td>
<td>50.16</td>
<td>95.81</td>
</tr>
<tr>
<td>+ embeddings 100, method 1</td>
<td>50.88</td>
<td>96.29</td>
</tr>
<tr>
<td>+ embeddings 100, method 2</td>
<td>51.30</td>
<td>96.34</td>
</tr>
</tbody>
</table>

low frequency word”. We believe that this second method could improve our model due to the high number of unique unknown words (table 6.2).

We have run our baseline model with both the methods, and reported the results in table 6.3. There is no randomness in our model besides the initialization of the random embeddings. We observe that the method using a common unknown embeddings which is trained through the epochs gives substantially better results. This second method causes an increase in both F-score and accuracy. We will therefore keep using this initialization method in the upcoming experiments.

### 6.2 Scaling the loss function due to imbalanced data set

Before further experimenting with word embeddings, we investigate some methods to mitigate the problem of unbalanced classes in our data set. There are several ways of dealing with an unbalanced data set. The problem of unbalanced classes is investigated in Huang et al. (2016), and they divide different techniques into two groups: data re-sampling and cost-sensitive learning. The data re-sampling group contains techniques such as undersampling; where a part of data from the over-represented class is removed, and oversampling; where data from the under-represented data is duplicated. The aim for these techniques is to modify the training data set to make the classifier equally good for all classes. We choose to discard these two methods because of the size of our data set. Undersampling would make our data set too small, and oversampling would increase the problem of over-fitting.

We will investigate a technique in the group of cost-sensitive learning. This technique will instead of altering the data, perform manipulations at algorithmic level to adjust the costs of misclassification. The method we are going to use is doing scaling of the loss function. This is done by,
We have tested what we call static scaling, where the scaling factor is independent of the number of threats, in the interval 1:1 - 1:20. We have also tested what we call dynamic scaling, where the scaling factor is computed using the logarithm of the percentage of threats / non-threats. We will test out different factors for the threat-part of the dynamic scaling, using the formula: $(\ln(\text{threats}) : \ln(100 - \text{threats}) \times x)$. We will vary the factor $x$ through the experiments. Experiments are run with the baseline setup, using Wikipedia + Gigawords embeddings and method 2 for unknown word initialization (section 6.1.2). All initial weights and embeddings (including the unknown-embedding) are identical throughout the experiments. The results for static and dynamic scaling respectively, are shown in table 6.4 and table 6.5.

More sentences will be classified as containing threats as the loss function is adjusted such that "threat"-class gains in importance. This leads to, generally, higher recall with larger scaling difference, and lower precision.

### Table 6.4: Different ratios for static scaling of loss function

<table>
<thead>
<tr>
<th>Scaling</th>
<th>F-score</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1</td>
<td>51.30</td>
<td>96.34</td>
<td>61.06</td>
<td>48.90</td>
</tr>
<tr>
<td>1:2</td>
<td>51.55</td>
<td>95.78</td>
<td>53.13</td>
<td>55.96</td>
</tr>
<tr>
<td>1:3</td>
<td><strong>54.26</strong></td>
<td>96.03</td>
<td>54.66</td>
<td>59.84</td>
</tr>
<tr>
<td>1:4</td>
<td>52.58</td>
<td>95.36</td>
<td>50.65</td>
<td>63.83</td>
</tr>
<tr>
<td>1:5</td>
<td>52.89</td>
<td>94.64</td>
<td>47.54</td>
<td>70.75</td>
</tr>
<tr>
<td>1:6</td>
<td>52.12</td>
<td>94.79</td>
<td>46.97</td>
<td>68.04</td>
</tr>
<tr>
<td>1:10</td>
<td>51.87</td>
<td>94.64</td>
<td>45.16</td>
<td>69.66</td>
</tr>
<tr>
<td>1:20</td>
<td>50.51</td>
<td>93.86</td>
<td>41.45</td>
<td>75.35</td>
</tr>
</tbody>
</table>

Table 6.5: Different dynamic scaling of loss function. Formula ($\ln(\text{threats}) : \ln(100 - \text{threats}) \times x$) used for scaling, where $x$ is varied through the experiments.

<table>
<thead>
<tr>
<th>Scaling</th>
<th>F-score</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.8$</td>
<td>52.56</td>
<td>96.22</td>
<td>58.44</td>
<td>52.43</td>
</tr>
<tr>
<td>$0.9$</td>
<td>54.34</td>
<td>95.92</td>
<td>54.33</td>
<td>60.52</td>
</tr>
<tr>
<td>$1$</td>
<td><strong>54.37</strong></td>
<td>95.79</td>
<td>51.70</td>
<td>63.47</td>
</tr>
<tr>
<td>$1.1$</td>
<td>53.19</td>
<td>96.09</td>
<td>55.95</td>
<td>56.51</td>
</tr>
<tr>
<td>$1.2$</td>
<td>53.77</td>
<td>96.04</td>
<td>55.71</td>
<td>57.65</td>
</tr>
</tbody>
</table>
Table 6.6: Results from use of different embedding dimensions for pre-trained embeddings on Wikipedia + Gigaword, loss function scaling $\ln(\text{threats}) : \ln(100-\text{threats})$ with larger scaling difference. Accuracy will be a bit lower due to fewer non-threats classified correct. The F-score has a peak at 1:3 scaling for the static scaling (F-score of 54.26), but the best scaling overall is the dynamic $\ln(\text{threats}) : \ln(100-\text{threats})$-scaling (F-score of 54.37). This is quite a big improvement from the F-score of 51.30 for the 1:1 scaling. At this point, the scores are somewhat biased towards the recall metric. For our further experiments, we will use the dynamic $\ln(\text{threats}) : \ln(100-\text{threats})$-scaling.

### 6.3 Experimenting with different pre-trained embeddings

#### 6.3.1 Different embedding dimensionality

We will in this subsection look at what impact the embedding dimensionality has on the results. We will experiment with using the pre-trained word embeddings and the dynamic log-scaling of the loss function described in the previous section. For the pre-trained GloVe embeddings, there are 4 different embedding sizes available for the Wikipedia + Gigaword training set: 50, 100, 200 and 300. For these experiments, we have to take into account that both the different word embeddings and the different random initialized unknown word embeddings are causing non-determinism. Based on empirical tuning, different number of epochs is used for the different embedding sizes. We have for the 50, 100, 200 and 300 dimensions used respectively 5, 5, 4 and 4 epochs. The results are shown in table 6.6.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>F-score</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>48.71</td>
<td>94.63</td>
<td>46.52</td>
<td>59.88</td>
</tr>
<tr>
<td>100</td>
<td><strong>54.37</strong></td>
<td>95.79</td>
<td>51.70</td>
<td>63.47</td>
</tr>
<tr>
<td>200</td>
<td><strong>54.57</strong></td>
<td>96.02</td>
<td>55.36</td>
<td>60.27</td>
</tr>
<tr>
<td>300</td>
<td>52.76</td>
<td>95.88</td>
<td>54.65</td>
<td>56.73</td>
</tr>
</tbody>
</table>

We can see from table 6.6 that when using pre-trained embeddings, the F-score (and accuracy) increases with higher dimensionality, except for the 300 dimension embeddings. We will treat this as an outlier caused by the non-determinism in our model due to the random seed chosen, and we
will nevertheless do some more experiments with different embedding sets with different number of dimensions in the next section.

6.3.2 Different training corpora

For the embedding layer we have decided to use pre-trained word embeddings trained using GloVe (Pennington, Socher and Manning 2014). We are in this section going to experiment with 4 embedding sets trained on 4 different corpora. Embedding sets trained on 3 of the corpora are lower-case, which means we will have to lowercase all tokens in our data set. The fourth embedding set is trained on a corpus where case is preserved, which means it consists of both upper case and lower case tokens.

We will run one experiment per embedding set, and all the models are created with the trainable-parameter set to True and with \( \ln(\text{threats}) : \ln(100-\text{threats}) \)-scaling of the loss function. We will in the following describe the 4 different embedding sets. The results from runs with the different embedding sets are shown in table 6.7.

**Wikipedia 2014 + Gigaword** 5: This embedding set is trained on a corpus consisting of data from Wikipedia 2014 and a large news corpus called Gigaword 5. The Wikipedia + Gigaword data set consists of 6 billion tokens. The size of the lower-cased vocabulary is 400K. The embedding set is available in 4 different number of dimensions: 50, 100, 200, 300. We experimented with all the mentioned dimensions in section 4.5.1.

**Twitter**: This embedding set it trained on a large corpus of tweets from the social media platform Twitter. The twitter data set consists of 27 billion tokens distributed over 2 billion tweets. The size of the lower-cased vocabulary is 1.2 million. The embedding data set is available in 4 different dimensionalities: 25, 50, 100, 200. For this experiment, we will use both the 100 and the 200 dimension embedding sets.

**Common crawl, lower-case**: Common crawl is a large web archive consisting of huge amounts of web pages. The common crawl data set used for training this embedding set consists of 42 billion tokens. The size of the uncased vocabulary is 1.9 million. The dimension of the vectors is 300.

**Common crawl, preserved case**: It is relevant for our task to also look at the difference between embeddings trained on lower-cased training corpora and corpora where the case is preserved. All the embedding sets we have described above are lower-cased, which means there is no difference
between lower case and upper case in the tokens. For our task, the fact that a word/sentence is written in upper case can say something about the meaning/sentiment. This embedding set is trained on a larger common crawl corpus (840 billion tokens), and the size of the vocabulary is 2.2 million. The size of the vectors is 300 dimensions.

We believe that all of these training corpora can be beneficial for our task because of the fact that they are all web text, and some of them are also more specifically user generated text.

As we can see from table 6.7 there are notable differences in results between the embedding sets. The results are consistent with the numbers in table 6.2. The embedding sets with fewest unknown words provides the best results and vice versa. The embeddings trained on the Common Crawl data set give slightly better results than the ones trained on the Wikipedia + Gigawords data set in terms of F-score. However, this data set is bigger than the others, and causes a huge increase in training time.

### 6.3.3 Embeddings conclusion

When choosing which embedding set to continue experimenting with, we have to take both performance and training time into consideration. The training time increases with both higher embedding dimensionality and with the size of the embedding set. We will in our upcoming experiments use the Wikipedia + Gigawords embedding set with dimension 100. This is largely due to the long training time for the larger embedding corpora.

### 6.4 Model architecture

As we recall from chapter 2, convolutional neural networks have been used for different tasks, also in the NLP field. We will in this section ex-
We will in this section experiment with different window sizes and number of feature maps. We want to take into account that there may be correlation between these two parameters, and will therefore do a grid search. We have chosen window sizes of 2 to 6 and feature map numbers of 16, 32, 64 and 128. Other parameters and the model architecture are the same as for the baseline model. We are using the (GloVe Wikipedia + Gigaword) embedding set with dimension size 100. The library function Numpy.Random.Seed() is used, which will keep all stochastic variables static throughout the experiments.

Table 6.8 shows the F-scores from the size-parameter grid search run using pre-trained embeddings. The conclusion is that, generally, the F-score gets higher with increased number of feature maps until some point (64-256). It also seems like, generally, that with higher window size, the lower number feature maps is beneficial.

The next thing to research is the correlation between the window sizes, and if concatenating the information from different window sizes will give us better results.
<table>
<thead>
<tr>
<th>Window sizes</th>
<th>F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 3</td>
<td>54.73</td>
</tr>
<tr>
<td>3, 4</td>
<td>55.39</td>
</tr>
<tr>
<td>4, 5</td>
<td>56.44</td>
</tr>
<tr>
<td>5, 6</td>
<td>54.46</td>
</tr>
<tr>
<td>2, 3, 4</td>
<td>56.52</td>
</tr>
<tr>
<td>3, 4, 5</td>
<td><strong>56.55</strong></td>
</tr>
<tr>
<td>4, 5, 6</td>
<td>55.51</td>
</tr>
</tbody>
</table>

Table 6.9: Results from use of multiple window sizes with 128 feature maps

### 6.4.2 Concatenating more window sizes

In this subsection, we are going to try concatenating multiple window sizes. This is done by Kim (2014), and he gets good results when using this technique. What is also mentioned in the paper is a comparison to the results provided by Kalchbrenner, Grefenstette and Blunsom (2014). The model without multiple filter widths and feature maps reports much worse results than the model with more capacity on the same task. This was for the task of movie review sentiment analysis. We will therefore investigate the impact of multiple window sizes has for our task.

Kim (2014) uses window sizes of 3, 4 and 5 concatenated for their model, and received good results for this combination. They use a feature map size of 100. We will in addition to this (3, 4, 5) try out some more combinations of window sizes. We will perform the experiments using both 64 and 128 as number of feature maps.

The window concatenation procedure is done by creating more independent "paths" in the model network after the embedding layer. Each of the paths has its own convolution layer, max pooling layer and flattening layer. After the mentioned layers, the results from the different "paths" are concatenated before being passed further through the network.

We can see from table 6.9 and table 6.10 that the concatenation of window sizes caused a substantial improvement. This also fits well with the discussion in Kim (2014). The highest F-scores are provided by the models with 3 window sizes concatenated. We get the highest F-score for the model with 64 feature maps and window sizes 4, 5 and 6. Window sizes 3, 4 and 5 also provides good results for both the number of feature maps.
### Table 6.10: Results from use of multiple window sizes with 64 feature maps

<table>
<thead>
<tr>
<th>Window sizes</th>
<th>F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 3</td>
<td>54.54</td>
</tr>
<tr>
<td>3, 4</td>
<td>53.98</td>
</tr>
<tr>
<td>4, 5</td>
<td>53.73</td>
</tr>
<tr>
<td>5, 6</td>
<td>52.87</td>
</tr>
<tr>
<td>2, 3, 4</td>
<td>55.17</td>
</tr>
<tr>
<td>3, 4, 5</td>
<td>57.05</td>
</tr>
<tr>
<td>4, 5, 6</td>
<td><strong>57.67</strong></td>
</tr>
</tbody>
</table>

### 6.4.3 Regularization / dropout

The next step is to add some means of regularization to our model. Regularization makes it possible to run more epochs without encountering the problem of over-fitting. There are different regularization techniques that we could use, but we will here focus on dropout. Dropout is a regularization technique for neural network models proposed in a paper by Srivastava et al. (2014). Dropout is a powerful and simple regularization technique, and is commonly used. A description of the technique will be provided in the following.

The dropout regularization will randomly disable a fraction (e.g., 50%) of neurons in the applied layer. This is done to avoid over-fitting. This works by preventing neurons from co-adapting, and forcing them to learn individually useful features. Dropout is only used during training of the network, and is not used when evaluating the network. This technique will probably be especially beneficial for our model because of our small data set, which is likely to cause over-fitting with a complex model.

An extra parameter $p$ is introduced when adding dropout to our network. The $p$ represents the fraction of nodes to be randomly disabled during training. $p = 0$ means no dropout, while higher values for $p$ means more dropout. Typical values for the parameter are 0.2–0.5 (Srivastava et al. 2014).

We will experiment with dropout after both the input layer and the hidden layer. We will start out with the two approaches isolated, and afterwards do dropout on both the input and hidden layers. All experiments are done using feature map size 64 and concatenated windows of sizes 4, 5 and 6.
<table>
<thead>
<tr>
<th>Dropout rate</th>
<th>F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>current model</td>
<td>57.67</td>
</tr>
<tr>
<td>0.1</td>
<td>55.31</td>
</tr>
<tr>
<td>0.2</td>
<td>54.70</td>
</tr>
<tr>
<td>0.3</td>
<td>56.09</td>
</tr>
<tr>
<td>0.4</td>
<td>57.82</td>
</tr>
<tr>
<td>0.5</td>
<td>58.81</td>
</tr>
<tr>
<td>0.6</td>
<td><strong>58.85</strong></td>
</tr>
<tr>
<td>0.7</td>
<td>58.33</td>
</tr>
</tbody>
</table>

Table 6.11: Results from use of dropout on input layer

Number of epochs are increased with higher dropout rate, and all experiments uses a number of epochs in the interval 5-15.

**Using dropout on the input layer**

Srivastava et al. (2014) suggests a dropout rate of 0.2 for the input layer. We will in addition to this try out 0.1 and 0.3-0.7. The results are presented in table 6.11.

We can observe that that low dropout rates actually gives lower scores. When the dropout rate is increased, the scores increases. We get the highest F-score of 58.85 when using dropout rate 0.6. This is a relatively high dropout rate, but we may need to decrease the rate when adding dropout also to the hidden layer.

**Using dropout on hidden layer**

We will in these experiments use dropout only in the hidden layer, to isolate the dropout effect. According to Srivastava et al. (2014), the dropout rate for hidden layers is depending on the choice of number of hidden units. Higher $p$ requires higher number of hidden units, which slows down the network and can lead to under-fitting. Smaller $p$ may not produce enough dropout to prevent over-fitting. We will test out different dropout rates in the range 0.2–0.8.

The results (table 6.12) shows that our model does not benefit from using dropout at the hidden layer for any dropout rates. We get the highest F-score for our current model, without any dropout (F-score of 57.67). A dropout rate of 0.4 do however also give good results (F-score of 57.01).
Dropout rate & F-score \\
| current model | 57.67 |
| 0.2          | 54.30 |
| 0.3          | 54.31 |
| 0.4          | 57.01 |
| 0.5          | 53.53 |
| 0.6          | 55.15 |
| 0.7          | 53.53 |
| 0.8          | 50.66 |

Table 6.12: Results from use of dropout on hidden layer

We will nevertheless do some experimenting with using dropout on both the input layer and the hidden layer in order to investigate whether these interact.

**Using dropout on both input layer and hidden layer**

We will now try out dropout on both the input layer and the hidden layer. We will test out some different combinations of dropout rate. Even though a relatively high dropout rate was beneficial when using dropout on the input layer, this may not be the case when combining it with dropout on the hidden layer.

As we see in table 6.13, the best results are achieved when using high dropout rate for the input layer combined with a moderate dropout rate for the hidden layer. The best result is a F-score of 58.86 when using dropout rate 0.6 for the input layer and dropout rate 0.4 for the hidden layer. The results correspond well with the ones in table 6.11 and table 6.12. We will later on test changing the number of nodes in the hidden layer, and may in that context also change dropout rates.

**6.4.4 Optimizer function**

We have for all experiments done up to now used the ADAM optimizer function. We will in this section test if a different optimizer function could be beneficial for our model. The optimizer we are going to experiment with is the adadelta optimizer function (Zeiler 2012), used by Kim (2014).

We will use the same model architecture and parameters we have used up to now, and dropout rates of 0.6 (input layer) and 0.4 (hidden layer).
### Table 6.13: Results from use of dropout on both input layer and hidden layer

<table>
<thead>
<tr>
<th>Dropout input layer</th>
<th>0.0</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dropout hidden layer</td>
<td>0.0</td>
<td>57.67</td>
<td>54.70</td>
<td>56.09</td>
<td>57.82</td>
<td>58.81</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>54.30</td>
<td>56.74</td>
<td>57.72</td>
<td>57.89</td>
<td>57.72</td>
</tr>
<tr>
<td></td>
<td>0.4</td>
<td>57.01</td>
<td>56.27</td>
<td>57.29</td>
<td>57.75</td>
<td>58.57</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>53.53</td>
<td>56.57</td>
<td>58.17</td>
<td>57.67</td>
<td>58.29</td>
</tr>
<tr>
<td></td>
<td>0.6</td>
<td>55.15</td>
<td>55.67</td>
<td>57.00</td>
<td>56.70</td>
<td>57.43</td>
</tr>
<tr>
<td></td>
<td>0.7</td>
<td>53.53</td>
<td>54.13</td>
<td>54.05</td>
<td>56.30</td>
<td>57.17</td>
</tr>
</tbody>
</table>

Table 6.14: Results from use of ADAM optimizer function, compared to the adadelta and the adagrad optimizer functions

<table>
<thead>
<tr>
<th>Optimizer function</th>
<th>F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAM (current)</td>
<td>58.86</td>
</tr>
<tr>
<td>adadelta</td>
<td>53.64</td>
</tr>
</tbody>
</table>

We can see from table 6.14 that the results for using the adadelta optimizer are much worse than the model using ADAM optimizer function. The F-score when using the adadelta optimizer function is as low as 53.64. This may be partly caused by other parameters in the model tuned using the ADAM optimizer function. We will nevertheless continue using the ADAM optimizer for our upcoming experiments.

#### 6.4.5 Batch size

An important part of our model optimization is choosing an appropriate mini batch size. We are currently using a batch size of 128, which means 128 training sentences are used in each forward/backward pass during training. The choice of mini batch size is largely task specific, but small batches (32–512) are shown to often perform better than huge batches (Keskar et al. 2016). Keskar et al. (2016) suggests some different causes for this:

1. Large batch training tends to over-fit
2. Large batch training are attracted to saddle points
3. Large batch training lack the explorative properties of small batch training and tend to zoom-in on the minimizer that is closest to the initial point
4. Small batch and large batch training converge to qualitatively different minimizers with differing generalization properties.
Batch size is also reported by Kim (2014), which uses 50 as mini batch size. We will test batch sizes in the range 32–1024, and see which size fits best for our task.

Also worth mentioning is that we had to increase the number of epochs for these experiments. This were found out by systematically testing for the different experiments. Number of epochs up to 30 are used.

The results are presented in table 6.15. We find that our model performs best with a mini batch size of 512, which is larger than the current batch size. The F-score achieved is 63.19. We will continue using this batch size (512) for our upcoming experiments.

### 6.4.6 Number and size of hidden layers

Deep neural nets are commonly used for classification, and also in combination with convolutional layers. While deeper nets are commonly used for e.g. picture classification, it is more common to use shallower nets for text classification. In the models described in their articles, Kim (2014) and Kalchbrenner, Grefenstette and Blunsom (2014) do not use any hidden layers before the last fully connected layer. We will nevertheless experiment with some different hidden layers for our model, in addition to a model with no hidden layer.

A dropout layer with rate 0.6 after the input layer is used for all experiments. For the hidden layer, we have increased dropout rate for higher number of hidden nodes, and decreased dropout rate for lower number of hidden nodes. The dropout rates are chosen based on empirical tuning.

The results from experimenting with different number of hidden nodes and layers are provided in table 6.16. We observe that the best results
Table 6.16: Results from using different number of nodes in hidden layer

<table>
<thead>
<tr>
<th>Number of hidden nodes</th>
<th>F-score</th>
<th>Dropout rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>no additional hidden layer</td>
<td>62.37</td>
<td>-</td>
</tr>
<tr>
<td>125</td>
<td>61.44</td>
<td>0.3</td>
</tr>
<tr>
<td>125</td>
<td>59.37</td>
<td>0.4</td>
</tr>
<tr>
<td>250 (current)</td>
<td>62.19</td>
<td>0.4</td>
</tr>
<tr>
<td>375</td>
<td>62.21</td>
<td>0.4</td>
</tr>
<tr>
<td>500</td>
<td>61.28</td>
<td>0.4</td>
</tr>
<tr>
<td>500</td>
<td>60.78</td>
<td>0.5</td>
</tr>
</tbody>
</table>

are achieved for the model with no additional hidden layer, with a F-score of 62.37. Hidden layers do not result in any improvement for the performance of the network. This is probably because of the size of our data set, and also the complexity of our task. The work of Wester et al. (2016) also shows that the lexical features are the ones with most relevant information in our data set. This may very well be the reason why it is not beneficial with deeper networks extracting more abstract features.

6.4.7 Multiple convolutional layers

The model we have created and worked with up to now is quite similar to the one proposed in the Kim (2014) article. We will in this section experiment with a different model architecture, based on the architecture presented in the Kalchbrenner, Grefenstette and Blunsom (2014) article. What is special with this architecture is that two convolutional layers, with following k-max pooling layers is used. We will use a similar architecture, but with regular max pooling. This is because there is no support for using k-max pooling in Keras. We will also use the experiences gathered from earlier experiments when constructing the model. A description of the network will be provided in the following.

For the embedding layer, we use the same embeddings as the model with concatenated convolutional layers. The embeddings are trained on the Wikipedia + Gigawords data set, and has dimensionality 100. In the following we will experiment with a model using both 2 and 3 convolutional layers. Rectifier linear units activation function is used for all the convolutional layers. A max-pooling layer with pooling factor 2 will follow after each of the convolutional layers. A fully connected layer using a sigmoid activation function will follow after the convolutional layers. Binary cross-entropy is used as loss function, and for optimizer rule we will experiment with both adagrad (used by Kalchbrenner, Grefenstette and Blunsom (2014)) and ADAM (used in our concatenation model). A mini-batch
When choosing the size of the convolution windows, we have both taken our earlier experiments and the research provided by Kalchbrenner, Grefenstette and Blunsom (2014) into account. Window sizes of 4, 5 and 6, and 32 feature maps has been shown to be good choices for our task in our earlier experiments. These sizes were used for concatenated convolutional layers, but we will nevertheless experiment with using the same values for our model with multiple sequential convolutional layers. Kalchbrenner, Grefenstette and Blunsom (2014) experiments on a task of binary sentiment prediction in movie reviews and also a task of twitter sentiment prediction. For both these tasks they use two convolutional layers with window sizes of 7 and 5, and 6 and 14 for number of feature maps. We will also test this set up, even though our model is not identical to the model developed by Kalchbrenner, Grefenstette and Blunsom (2014). We will also experiment with som other window sizes and number feature maps, based on the two models mentioned.

The results from the experimenting with different number of convolutional layers are shown in table 6.17. We observe that none of the models using multiple sequential convolutional layers outperform the model using concatenated convolutional layers (F-score of 62.37). Some of the models do however receive good results. The best results are obtained using window sizes 4, 5 and 6, number of feature maps 64 for all layers, and ADAM optimizer function. The parameter tuning for the multiple sequen-

<table>
<thead>
<tr>
<th>Window sizes</th>
<th>Feature maps</th>
<th>Optimizer</th>
<th>F-score</th>
<th>Dropout</th>
</tr>
</thead>
<tbody>
<tr>
<td>7, 5</td>
<td>6, 14</td>
<td>adagrad</td>
<td>56.23</td>
<td>-</td>
</tr>
<tr>
<td>4, 5, 6</td>
<td>64, 64, 64</td>
<td>ADAM</td>
<td>50.00</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.17: Results from model using multiple convolutional layers without dropout

<table>
<thead>
<tr>
<th>Window sizes</th>
<th>Feature maps</th>
<th>Optimizer</th>
<th>F-score</th>
<th>Dropout</th>
</tr>
</thead>
<tbody>
<tr>
<td>7, 5</td>
<td>6, 14</td>
<td>adagrad</td>
<td>56.13</td>
<td>0.6</td>
</tr>
<tr>
<td>4, 5, 6</td>
<td>64, 64, 64</td>
<td>ADAM</td>
<td>61.07</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 6.18: Results from model using multiple convolutional layers with dropout (0.6) on input layer

size of 512 will be used for all experiments. Experiments are run both with and without dropout after the input layer. We have used a dropout rate of 0.6, which is the same as used before.
tial convolution model was not as comprehensive as for the model with concatenation architecture, and this may partly explain the lower scores.

6.5 Final model

6.5.1 Description and results

After experimenting with different model architectures and tuning of parameters, the conclusion is that the model with concatenated convolutional layers is the best for our task. We will in the following describe this final model in detail.

The model input is represented using pre-trained word embeddings trained on the Wikipedia + Gigawords dataset using the GloVe algorithm. The dimension of the embeddings used are 100. A dropout with dropout rate of 0.6 is used on the input layer. After the input layer, a concatenation of 3 convolution layers with following max-pooling follows. The convolution layers have window sizes of 4, 5 and 6, and they have 64 feature maps each. Then follows a fully connected layer using a sigmoid activation function. The loss function used is binary crossentropy, and adam optimizer function is used. We use a batch size of 512, and train our model using 30 epochs.

The F-score for our final model is presented in table 6.19 together with the development results achieved by Wester et al. (2016) and results for models used as baselines. We observe that our final model outperform all models used as baselines, including the Bag-of-Words Maximum Entropy model (F-score of 61.23). Our model is however not able to come any close to the results achieved by Wester et al. (2016) (F-score of 69.93). Our final model ran on the development data set achieves a F-score of 62.37.

6.5.2 Analysis of errors

We will now take a closer look at the errors our model make when classifying sentences. We provide a confusion matrix in table 6.20 for running our final model on the development data set. We have used the same 5-fold cross-validation, and then summed up the results. The confusion matrix shows true positives, false positives, true negatives and false negatives for the model. We also want to compare these findings with the ones provided in the work of Wester et al. (2016). Our model is actually able to reveal a higher number of threats (720), compared to their 668. Our model also
### Table 6.19: Results from models

<table>
<thead>
<tr>
<th>Model</th>
<th>F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concat. model</td>
<td>62.37</td>
</tr>
<tr>
<td>Multiple seq. model</td>
<td>61.07</td>
</tr>
<tr>
<td>Wester model</td>
<td>69.93</td>
</tr>
<tr>
<td>Baseline CNN model</td>
<td>50.16</td>
</tr>
<tr>
<td>Baseline minority class</td>
<td>9.21</td>
</tr>
<tr>
<td>Wester BoW MaxEnt model</td>
<td>61.23</td>
</tr>
</tbody>
</table>

### Table 6.20: Confusion matrix for development results, using our final model

<table>
<thead>
<tr>
<th></th>
<th>Classified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Threat</td>
</tr>
<tr>
<td>Annotated</td>
<td>728</td>
</tr>
<tr>
<td></td>
<td>411</td>
</tr>
</tbody>
</table>

The model created by Wester et al. (2016) do however obtain a better F-score caused by higher number of true negatives (21,671) and lower number of false positives (181).

In order to further examine the results, we have also picked out some examples of false positive and false negative sentences. The examples are given in figure 6.2 and 6.3, respectively. We will start out examining the sentences that were wrongly classified as non-threats, annotated as threats (False negatives). Some of the examples are obviously threats of violence, like "we are killing them" and "let us kill all the minorities". Others, like the one stating "get a bomb and kill yourself then" are less obvious, and one can question the fact that this is even a threat at all. Many of the sentences not captured by our classifier also contains unusual expressions of threats of violence like "genocide", "breivik" and "holocaust".

For the sentences wrongly classified as threats, annotated as non-threats, we observe many sentences containing words associated with violence such as "kill", "killing" and "death". One sentence "why not just kill them
as painless as possible” is taken out of context, and the subject is actually animal slaughter. One of these examples, containing “all spoilt idiots like you can starve to death”, we would say are maybe annotated wrongly.
this solution to 'the race problem' is genocide.

you want a war you fuckin kike, we would make the holocaust look like fuckin disneyland.

let the majority decide everything, and preferable let us kill all the minorities and all the people who have a different way of thinking and seeing the world than the majority.

also what's that about just 10 minutes of some assholes screaming bullshit don't need to be blocked lol it's just show how stupid they are about their religion like with the muhammad in familly guy lol zzz you guys shud think about it you came to other coutry and you act like your the kings i do nt like stereotypes but get a bomb and killurself then

if i were running things, all muslims would be deported to somalia, all mosques would be razed tomorrow, and islam would be banned as an intolerant, violent, dangerous political party.

please produce more anders breiviks please!

we are killing them

sounds like it's time to start castrating these camel-molesting savages.

Figure 6.2: Examples of false negative sentences for development results, using our final model.
so if the first part say, love your not muslim neighbour, and the last part say kill them all.

you deserve or the non white immigrants, you can get infact i hope america and europe fascist trade policy, impedes on itself so all spoilt idiots like you can starve to death.

we should synthesis all the good things in all cultures and get rid of the bad.

if they deport all immigrants then whites are just going to be bored and start killing each other lol

let us damn their allies until death.

why not just kill them as painless as possible

what about setting fire yo a synagoge when its fully visited ... i could baricade the exists and set fire to it ahahahahah, that thought makes me so happy, i think it ’s something jesus would want me to do, kill our enemy ’s he would in fact give me a place in his kingdome ... kill the anti christs.

Figure 6.3: Examples of false positive sentences for development results, using our final model.
Chapter 7

Results

We will in this chapter test our final model using the held out data set. We will first briefly describe the held out data set. Secondly, we will present the results for the held out data set, and perform an analysis of these.

7.1 Held out data set

In section 4.1 we described the partitioning of our entire data set into a development data set and a held out test data set. The held out data set is kept unseen through all our experiments, and is first introduced now. The reason why we want to use an unseen data set when evaluating our final model is that it gives a more realistic setting, and we are not able to fit and tune our model to a set of unseen sentences. Some statistics comparing number of threats in the development set and the held out set are shown in table 7.1. The held out set contains, as mentioned in section 4.1, 1/5 of the comments. We can see that the distribution of the sentences containing threats is largely equal. For the development set, 4.80% of the sentences contains threats. For the held out set, 4.94% of the sentences contains threats. Training of our final model is done on the entire development set, and tested on the held out set.

7.2 Results and analysis

The results for testing our model on the held out set is shown in table 7.2. Results from the model developed by Wester et al. (2016) are also shown in the table for comparison. We want to repeat the fact that the results presented for the model by Wester et al. (2016) is arrived at using the same data set partitioning. This means that their model is both trained and tested at the same data set as ours, thus making the results directly comparable.

Our model receives an F-score of 65.29 on the held out test set. The biggest
Table 7.1: Data set partitioning into development set and held out set, showing number of sentences containing threats.

<table>
<thead>
<tr>
<th></th>
<th>Sentences</th>
<th>Containing threats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development set</td>
<td>22,958</td>
<td>1,103</td>
</tr>
<tr>
<td>Held out set</td>
<td>5,685</td>
<td>281</td>
</tr>
</tbody>
</table>

Table 7.2: Held out results for our model, compared to results provided by Wester et al. (2016).

<table>
<thead>
<tr>
<th></th>
<th>F-score</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our model</td>
<td>65.29</td>
<td>63.39</td>
<td>68.11</td>
</tr>
<tr>
<td>Wester model</td>
<td>68.85</td>
<td>74.90</td>
<td>63.70</td>
</tr>
</tbody>
</table>

surprise when looking at the held out results for our model, is that the held out results are substantially higher than the development results (F-score of 62.37). This was a quite big improvement, and was not expected. We did however take precautions when experimenting, to avoid over-fitting. This was done both by using cross-validation, and in the way we chose to split our data set into contiguous folds.

We have also presented scores for precision and recall in the table (7.2). There are interesting differences in terms of precision and recall when comparing our scores with the ones provided by Wester et al. (2016). Our model gets higher score for the recall metric, with a score of 68.11 compared to 63.70 for the model by Wester et al. (2016). The model by Wester et al. (2016) do however get higher score for the precision metric, with a score of 74.90 compared to our 63.39. This corresponds well with the discussion in 6.5.2, where we compare our findings in terms of TP, FP, TN and FN to Wester et al. (2016). The reason for this fact is probably largely because of our attempt to mitigate the problem of unbalanced classes in our data set, which causes our model to classify more sentences as positive (threats).

We also did an experiment with extra training of the embeddings, as mentioned in section 5.4. We used the same model as described in section 6.5.1, but added 4 extra iterations of training for the embeddings. For each loop, all weights in the network were reset except from the embedding weights. All loops were run for 30 epochs. We observed the results for testing on the held out test set after each loop. The conclusion after the experimenting is that extra training of embeddings do not cause any improvement for our model. The results in terms of F-scores for the 5 loops were as follows: 65.29 (no extra training of embeddings), 62.72, 61.66, 60.15
and 58.88 (4 extra loops of training the embeddings). We can see from the numbers the model is clearly over-fitting with increased number of tuning loops for the embedding layer.
Chapter 8

Conclusion

In this thesis, we have explored the task of detecting threats of violence in social media, using convolutional neural networks. An existing data set of annotated YouTube comments was used in order to solve this task. Our goal was to develop a threat classifier using neural methods, more specifically convolutional neural networks. To the best of our knowledge this is the first application of deep learning methods to the task of threat detection. The task was solved using Keras for creating our model, and GloVe for training word embeddings. Parameter and architecture choices were made by repeatedly training our model and tracking the results.

We have systematically explored the effects of a range of different choices regarding architecture and parametrization. The choice of embeddings to use was done based on both performance and training time. We ended up using embeddings with dimensionality 100, trained on the Common Crawl lower cased data set. We explored the differences between keeping the embedding weights static versus updating the embedding weights during training. The version using non-static embedding weights outperformed the static version, and is used in the final model. We also explored different ways of initializing words not present in the embedding set. The best results were obtained using a method where one common random embedding vector "*__UNKNOWN__*" is given to all unknown words in the training data. This common "*__UNKNOWN__*"-vector is updated during training. Words in the validation data, not present in the word index, will also be given the UNKNOWN word embedding. This method performed best due to the high number of unique words in our data set.

To mitigate the problem of unbalanced classes in our data set, we investigated some methods for dealing with this. Specifically, we made use of a cost-sensitive learning technique scaling the loss function. We performed several experiments to find the best scaling ratio, and ended up with a log-scaling $\ln(\text{threats}) : \ln(100 - \text{threats})$, which is biased towards the threat-class.
A grid search was performed to find the size of the windows and the number of feature maps in our network. We also experimented with concatenating convolutional layers with different window sizes. The best results were obtained using 3 concatenated convolutional layers with window sizes of 4, 5 and 6, and 64 feature maps each. We followingly investigated the effect of applying dropout to different layers of our model. The regularization technique improved the performance of our model, and the best results were obtained using a dropout rate of 0.6 in the input layer and a dropout rate of 0.4 on the hidden layer. We also experimented with different mini batch sizes, and found that size 512 performed best on our task.

We also wanted to investigate the effect of adding more hidden layers to our network. The conclusion after experimenting with this was that the most beneficial configuration for our task was actually using no additional hidden layers at all. This might be due to the limited size of the training data.

The last part of our experimental work dealt with testing another model architecture, using multiple sequential convolutional layers. This architecture had been shown to be effective in previous work. The model with this architecture did not outperform the model using concatenated convolutional layers. The parameter tuning for the multiple sequential convolution model was not as comprehensive as for the model with concatenation architecture, and this may partly explain the lower scores.

Our final model achieved a F-score of 65.29. The results did not outperform the results by Wester et al. (2016), which developed and tested their model on the same data set.

Future work on the task of threat detection using CNN should research the use of multiple convolutional layers, and more thorough tuning of parameters for these. Secondly, other types of word embeddings (e.g. word2vec) should be tested. Other neural approaches like RNNs should also be experimented with.
Bibliography


67


