NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Hate Speech Detection Using Neural Networks

Vaggelis S. Spithas

Supervisor: **Stamatopoulos Panagiotis,** Assistant Professor, UoA

ATHENS

SEPTEMBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αναγνώριση Ρητορικής Μίσους με τη Χρήση Νευρωνικών Δικτύων

Ευάγγελος Σ. Σπίθας

Επιβλέπων: **Σταματόπουλος Παναγιώτης,** Επίκουρος Καθηγητής, ΕΚΠΑ

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2020

BSc THESIS

Hate Speech Detection Using Neural Networks

Vaggelis S. Spithas
S.N.: 1115201500147

SUPERVISOR: **Stamatopoulos Panagiotis,** Assistant Professor, UoA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αναγνώριση Ρητορικής Μίσους με τη Χρήση Νευρωνικών Δικτύων

Ευάγγελος Σ. Σπίθας
Α.Μ.: 1115201500147

ΕΠΙΒΛΕΠΩΝ: **Σταματόπουλος Παναγιώτης,** Επίκουρος Καθηγητής, ΕΚΠΑ

ABSTRACT

Hate Speech consists the use of abusive or stereotyping speech against a person or a group of people, based on their race, religion, sexual orientation and gender. In modern days the Internet and social media made the spread of hatred a lot more easy and fast than the past, as well as gave people the ability to do so anonymously. The purpose of this Thesis is to create a model that can detect such content in social media with the use of Machine Learning.

Firstly, we will define the problem of hate speech detection and discuss about existing research in this field. Then, we will expand on the necessary theoretical background information regarding Machine Learning, focusing particularly on the neural networks that will later be used. Following that, we will implement and train our own model built with LSTM neural networks. Finally, we will present and discuss the results of our model.

SUBJECT AREA: Natural Language Processing

KEYWORDS: Hate Speech, Classification, Natural Language Processing, Neural Networks, Recurrent Neural Networks

ΠΕΡΙΛΗΨΗ

Η ρητορική μίσους αφορά τη χρήση υβριστικού ή στερεοτυπικού λόγου εναντίον ενός ατόμου ή μίας ομάδας ανθρώπων, βασισμένου σε χαρακτηριστικά όπως η φυλή, η θρησκεία, ο σεξουαλικός προσανατολισμός και το φύλο. Στις μέρες μας το Διαδίκτυο και τα Μέσα Κοινωνικής Δικτύωσης έχουν καταστήσει τη μετάδοση μίσους πολύ πιο εύκολη και γρήγορη από το παρελθόν, ενώ έδωσαν επίσης στον κόσμο την ευκαιρία να το κάνει ανώνυμα. Ο σκοπός αυτής της πτυχιακής είναι η δημιουργία ενός μοντέλου το οποίο θα μπορεί να ανιχνεύσει έκφραση ρητορικής μίσους στα μέσα κοινωνικής δικτύωσης με τη χρήση της Μηχανικής Μάθησης.

Αρχικά, θα ορίσουμε το πρόβλημα της ανίχνευσης ρητορικής μίσους και θα συζητήσουμε για την υπάρχουσα έρευνα στο πεδίο αυτό. Έπειτα, θα παραθέσουμε το απαραίτητο θεωρητικό υπόβαθρο αναφορικά με τη Μηχανική Μάθηση, εστιάζοντας ιδιαίτερα στα νευρωνικά δίκτυα τα οποία και θα χρησιμοποιήσουμε αργότερα. Στη συνέχεια, θα υλοποιήσουμε και θα εκπαιδεύσουμε το δικό μας μοντέλο χρησιμοποιώντας Μακροχρόνια Βραχυχρόνια Μνήμη. Τέλος, θα παρουσιάσουμε και θα σχολιάσουμε τα αποτελέσματα μας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Επεξεργασία φυσικής γλώσσας

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ρητορική μίσους, Ταξινόμηση, Επεξεργασία φυσικής γλώσσας, Νευρωνικά δίκτυα, Επαναλαμβανόμενα Νευρωνικά Δίκτυα

CONTENTS

1 INTRODUCTION **11**

2 BACKGROUND AND RELATED WORK **12**
2.1 Hate Speech detection . . . 12
2.2 Hate Speech Definition . . . 13
2.3 Related Work . . . 13
2.3.1 Automated Hate Speech Detection and the Problem of Offensive Language . . . 13
2.3.2 Deep Learning for Hate Speech Detection in Tweets . . . 14
2.4 Neural Networks . . . 14
2.4.1 Neurons . . . 14
2.4.2 Layers . . . 15
2.4.3 Activation Function . . . 16
2.4.3.1 The sigmoid function . . . 16
2.4.3.2 Tanh . . . 16
2.4.3.3 ReLU . . . 16
2.4.3.4 Softmax . . . 17
2.4.4 Training . . . 17
2.4.4.1 Loss Function . . . 17
2.4.4.2 Gradient Descent . . . 17
2.4.4.3 Backpropagation . . . 19
2.4.4.4 Training Risks . . . 19
2.4.5 Types of Neural Networks . . . 21
2.4.5.1 Feedforward Neural Networks . . . 21
2.4.5.2 Convolutional Neural Networks . . . 21
2.4.5.3 Recurrent Neural Networks . . . 21
2.4.5.4 LSTM . . . 23

3 HATE SPEECH DETECTION PROCESS **26**
3.1 Task Definition . . . 26
3.2 Toolset . . . 26
3.3 Datasets . . . 26
3.4 Preprocessing . . . 26
3.5 Text Representation . . . 27
3.5.1 Bag of Words . . . 27
3.5.2 Term Frequency - Inverse Document Frequency . . . 28
3.5.3 Word Embeddings . . . 28
3.6 Model Design . . . 29

4 RESULTS **31**
4.1 Comparison to similar works . . . 35

5 CONCLUSION AND FUTURE WORK **36**

5.1 Conclusions 36
5.2 Future Work 36

ABBREVIATIONS - ACRONYMS **37**

REFERENCES **39**

LIST OF FIGURES

2.1 A depiction of a neuron 15
2.2 Frequently used activation functions 16
2.3 Simulation of Gradient Descent's convergence after multiple iterations . . . 18
2.4 Examples of underfitting (left), overfitting (right) and the optimal balance between them (center). 19
2.5 The ideal point to stop training based on the training error and the test error. 20
2.6 A FNN before (a) and after (b) applying dropout 20
2.7 A Recurrent Neural Network, folded (left) and unfolded (right) 22
2.8 A bidirectional RNN 22
2.9 An LSTM cell 24
2.10 A GRU cell 25

3.1 Bag of Words example 27
3.2 TF-IDF example 28
3.3 Word Embeddings example 29

4.1 An example to portray the shortcomings of accuracy 31

LIST OF TABLES

4.1 Accuracy and F1-score of models with one LSTM layer and glove embeddings 32
4.2 Precision and F1-score for each class for models with one LSTM layer and glove embeddings 32
4.3 Accuracy and F1-score of models with two LSTM layers and glove embeddings 32
4.4 Precision and F1-score for each class for models with two LSTM layers and glove embeddings 33
4.5 Accuracy and F1-score of models with one LSTM layer and an embeddings layer 33
4.6 Precision and F1-score for each class for models with one LSTM layer and an embeddings layer 33
4.7 Accuracy and F1-score of models with two LSTM layers and an embeddings layer 34
4.8 Precision and F1-score for each class for models with two LSTM layers and an embeddings layer 34

1. INTRODUCTION

The technological advancement of the recent era has affected modern life and society all over the world greatly. In recent years, accessibility to sectors such as education, medicine, industry, transportation etc. has become a lot easier. Due to the convenience and efficiency provided by technology, our lives have improved significantly. Communications have also been affected by the continuous advance of technology. Nowadays, social media and social networking seem to have turned into an irreplaceable part of our lives, making communication easier than it ever was. According to a research in 2019 [15], only in USA, 79% of the population keep an online profile, while this percentage was only 8% in 2005 and 35% in 2010 [6]. Teenagers have an even higher percentage, also logging in their profiles in daily basis.

Human interactions have dramatically changed with the increase of social media use. Teenagers and adults from all over the world have the opportunity to communicate with each other and share ideas and thoughts very easily using their online profiles. Moreover, anonymity makes things even easier, since it gives people the ability to express themselves without being tracked. It becomes obvious that this evolution brings a multitude of advantages, but as most things, it harbors some disadvantages as well. In this work we will focus on a specific disadvantage; the spread of Hate Speech through the Internet and social media platforms.

Social media platforms should not be blamed for this, since hatred is apparent in many modern life aspects. However, they stand as a platform for people to come out and spread Hate Speech easily and anonymously. In an attempt to control Hate Speech, modern societies, such as USA and EU, have voted against such kind of public speech making it illegal. As a result, social media platforms are also putting a lot of effort in order to comply with this legislation and effectively eliminate comments that promote hatred. One way to eliminate Hate Speech is to review only posts reported by other platform users. Although, due to the massive use of social media the amount of data distributed through them is exceedingly high making it impossible for humans to track problematic and offensive content. This is also ineffective, because it relies on users' subjectivity and trustworthiness, as well as depending on their ability to thoroughly track and flag such content. For all the above reasons, it becomes obvious that for all social media platforms which wish to comply with global and EU laws against hatred (anti-racist laws etc), but also to protect their users that belong to a minority group and make their experience in the platform more pleasant, an automatic tool to detect hate speech is really important.

In this chapter we reviewed the importance of dealing with Hate Speech and why using humans to detect such content is inefficient. Our goal for this our work, is to develop an automatic tool that can detect effectively such kind of content. In order to create such a tool we first need to transform the text of user comments into a form that is understandable by a computer program, more specifically a classifier in our case. Then we have to train the aforementioned classifier with an annotated dataset. In chapter 2, we introduce the Hate Speech detection task, we provide a formal definition of what constitutes Hate Speech, we present related work and research in the field and finally we give a detailed description of Neural Networks and how they work since they are a core concept of this Thesis. In chapter 3, we provide a detailed description of our model and the techniques we employed to achieve our goals, while in chapter 4 we present our results and compare them with other relevant work. Finally, we will conlude this Thesis in chapter 5 by giving a summary of our findings and discussing future work.

2. BACKGROUND AND RELATED WORK

In chapter 1, we discussed that the increase of the use of Hate Speech in Social Media is a major problem and we signified the importance of the design of an automated tool to point out such content. In section 2.1 we will formally describe the task of detecting Hate Speech. The classifier, which we will build in order to find out such content, needs to be trained and tested on an already annotated dataset. In Section 2.2 we provide a formal definition of Hate Speech as it is given by the European Committee of Ministers and we will also extend it. In section 2.3 we will discuss about relevant work that has been done in the field. Finally, in section 2.4, we will provide some background knowledge needed regarding Neural Networks.

2.1 Hate Speech detection

The goal of a Hate Speech Detection model is, given an input text T, to output True, if T contains Hate Speech and False otherwise. In order to perform this we will construct a model that will be trained with an already annotated dataset and then will be tested on unseen data in order to evaluate it.

Firstly, we will transform the input text in a format that can be understood and processed by our model via a text transformation method. The transformed data will then be fed into the machine learning algorithm, a Neural Network in our case, which will decide in which of two classes each element belongs. The process consists of a training phase and an evaluation phase. During the training phase, the classifier will be trained with the annotated data. Finally, after the training process is over, the classifier will be tested with data not encountered yet, in order to measure its accuracy. This will be done by counting how many correct predictions it achieved.

As we mentioned above, an annotated dataset is required in order to be used as input for our classifier. One way to obtain such a dataset is to have human annotators to label the dataset. A problem that can arise is that the annotators can interpret the Tweets objectively based on their personal education and background, due to absence of a formal and widely accepted definition of Hate Speech, and therefore conclude in annotations that do not agree with each other.

There has been done plenty of research regarding this problem. In their work Waseem 2016 [23], used 6,909 tweets annotated by amateurs in CrowdFlower in order to research the difference between amateur and expert annotators. For that reason, except from amateurs, they also had experts, such as feminist and anti-racist activists, annotate the tweets that failed a test. They also gave them the option to skip a tweet or to annotate it as noise. Eventually, they concluded in a low percentage of agreement between the two annotators group. A possible way to obtain decent annotation from amateurs is if only the data with high level agreement are taken into account . Compared to Waseem and Hovy 2016 [24], the aforementioned technique performs worse, which is caused by a high number of false positive results. It is also worth noting that this number is high even in the dataset annotated by experts.

2.2 Hate Speech Definition

As we discussed above, an annotated dataset is needed in order to construct our classifier. It is common practice to have humans annotate our data in order to find such a dataset . However, this process can be problematic as the labeling is based on each annotator's perception. Because of that, it is important for a formal definition to exist. Such a definition is provided by the European Committee of Ministers (Brown, 2017 [4]) as follows, "it covers all forms of expressions that spread, incite, promote or justify racial hatred, xenophobia, antisemitism or other forms of hatred based on intolerance". Furthermore, we can expand this definition by including speech that can be "insulting, degrading, defaming, negatively stereotyping or inciting hatred, discrimination or violence against people by targeting their race, ethnicity, nationality, religion, sexual orientation, disability, gender identity". In this work we will use already annotated datasets.

2.3 Related Work

In this section we will present related work in the field of Hate Speech detection. The most notable papers in the field are Davidson et al. 2017 [8] and Badjatiya et al. 2017 [1]. We will describe the features they used and the classification methods they employed, as well as present their results.

2.3.1 Automated Hate Speech Detection and the Problem of Offensive Language

The first work to be presented is Davidson et al. 2017 [8]. In this work, the authors aim to classify user comments as offensive language, expression of severe hate speech or neither. As a result, they label their dataset into the following three categories: hate speech, offensive language, or neither.

They argue that the bag-of-words approach, while it tends to have high recall it also leads to high rates of false positives because the appearance of offensive words to tweets not classified as hate speech can lead to their misclassification. In order to better identify the targets and intensity of hate speech they employed syntactic features, while other approaches, unfortunately, have been found to not be able to distinguish hate speech from offensive language.

They gathered their data by using the Twitter API and by searching for hate speech keywords from the Hatebase.org. Their datatset was then annotated with the use of CrowdFlower. They performed preprocessing to their data by making them to lower case, stemmed them with the Porter stemmer and then created unigram, bigram, and trigram features, each weighted by its TF-IDF value. In addition, they have used a sentiment lexicon for social media to include sentiment analysis as a feature. They experimented with various algorithms such as logistic regression with L1 and L2 regularization, naive Bayes, decision trees, random forests and linear SVM for the classification of their dataset. Their source code is publicly available on their GitHub page [1].

From their experiments, they concluded that the best technique is logistic regression with L2 regularization. They tested with 5-fold cross validation. Their final model achieved a precision of 0.91, a recall of 0.90 and a F1-score of 0.90. However, they found out that

[1] https://github.com/t-davidson/hate-speech-and-offensive-language

40% of actual hate speech is wrongly classified and only 5% of their offensive language was labeled as hate.

As future work, the authors want to study different uses of hate speech, such as the different target groups for it, and also delve into the characteristics of people expressing it.

2.3.2 Deep Learning for Hate Speech Detection in Tweets

The second work we will present is Badjatiya et al. 2017 [1]. In their work they used a dataset provided by Waseem and Hovy 2016 [24], which classifies tweets as racist, sexist or neither. For the representation of the text they tried various techniques. They used char n-grams, TF-IDF as well as Bag of Words vectors (BoWV). They also experimented with various classification techniques such as Logistic Regression, Random Forest, SVMs, Gradient Boosted Decision Trees (GBDTs) and Deep Neural Networks(DNNs).

They tested three neural network architectures, each initialised with random word embeddings or embeddings by GloVe. The three arcitectures they used are CNN, LSTM(RNN) and FastText with word vectors similar to BoW, except they were updated through backpropagation. Additionally, they attempted to use some other methods as well, such as SVMs and GBDTs. For the evaluation of their models they used 10-Fold cross validation and for their metrics they used weighted macro precision, recall and F1-scores. Their source code is also publicly available on their GitHub page [2].

They concluded that the best method was “LSTM + Random Embedding + GBDT” which was initialized to random vectors, trained the LSTM network, and then learned embeddings were used to train a GBDT classifier. For future work, they wish to test the importance of the network of the users.

2.4 Neural Networks

Artificial Neural Networks (ANN), or simply Neural Networks (NN), are inspired by their biological counterparts and they attempt to learn and process information. The core computational unit of the networks is the artificial neuron or simply neuron. NNs are comprised of a collection of connected neurons [16].

2.4.1 Neurons

The basic unit of computation in Neural Networks is the neuron. A neuron takes as input the weighted sum of a number of real values and then possibly adds a number to that sum, usually called the bias. Afterwards, a function is applied to the result of the sum, which is called the activation function, and the result of this function is the output of the neuron. The sum can be formally described as:

$$z = \sum_i w_i x_i + b$$

where x_i is each element of an input vector $\vec{x}$ and w_i is the weight for the corresponding input value. We can describe all the weights with a vector called $\vec{w}$. The dot product of

[2] https://github.com/pinkeshbadjatiya/twitter-hatespeech

the sum can be written as:

$$z = \vec{w} \cdot \vec{x} + b$$

If we call our activation function f then the output of the neuron can be written as:

$$y = f(z) = f(\vec{w} \cdot \vec{x} + b)$$

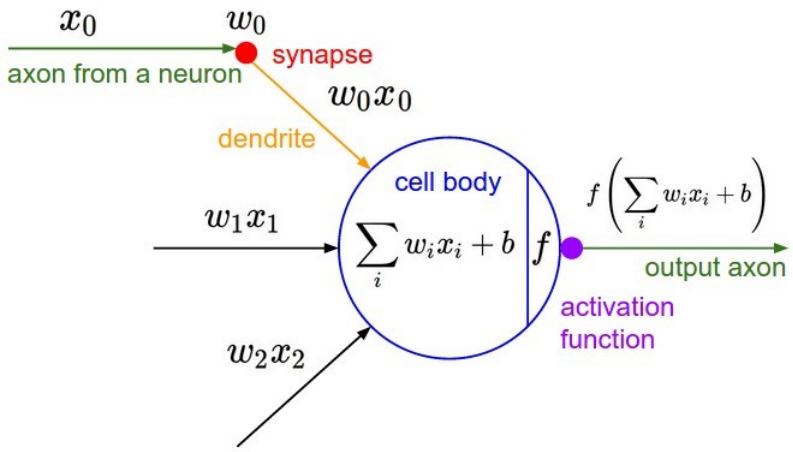

Figure 2.1: A depiction of a neuron

2.4.2 Layers

A neural network is comprised of multiple layers, each of which is comprised by one or more neurons. Every neural network has exactly one **input layer**, exactly one **output layer** and zero or more **hidden layers**.

The input layer receives the initial data, and the number of its neurons is usually equal to the number of features of the input data. The output layer can have one or more neurons and its output is the output of the network. The number of neurons of the output layer is dependent on the task. Between the input and the output layers there can be any number of layers, called the hidden layers. Networks that contain multiple hidden layers are called deep networks and as a result the subfield of Machine Learning that employs their use is called **Deep Learning**. Networks without a single hidden layer are only capable of representing linear separable functions or decisions (for example, the “AND” and “OR” problems but not the “XOR” problem). Although, these problems are rather trivial and usually they can be solved with simpler methods than NNs like Support Vector Machines (SVMs). Networks with exactly one hidden layer can approximate any function that contains a continuous mapping from one finite space to another, while two layers suffice to represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy [12].

While there is not a specific formula for the number of neurons to use in a hidden layer, it is important to decide them properly so we can ensure our model would perform well. If too few neurons are used then we can end up in underfitting, that is our model is not complex enough to capture the underlying structure of the data effectively. On the other hand, if too many neurons are used, we can end up in overfitting. In this case our model will not be able to generalise for unseen data because it learned too close to the training data.

There are actually many rule-of-thumb methods for a good decision for the amount of neurons for the hidden layers, such as the following [11]: The number of hidden neurons should be between the size of the input layer and the size of the output layer. The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer. The number of hidden neurons should be less than twice the size of the input layer. In the end, finding an appropriate number for the hidden layers, in order to find the right balance between underfitting and overfitting, comes down to trial and error.

2.4.3 Activation Function

The output of the activation function f of a neuron is the output of that neuron. For starters, the activation function must be a non linear function because it has been proven that a network of only linear activation functions can be reduced to network with a single layer. Such a network, unfortunately, is only able to solve linearly separable functions [7]. As a result, we desire the activation function to be non-linear. Furthermore, it is also needed to be differentiable. This is required by the optimization algorithm called backpropagation which will be used to train our model. We will explain it later.

The most frequently used ones are the following:

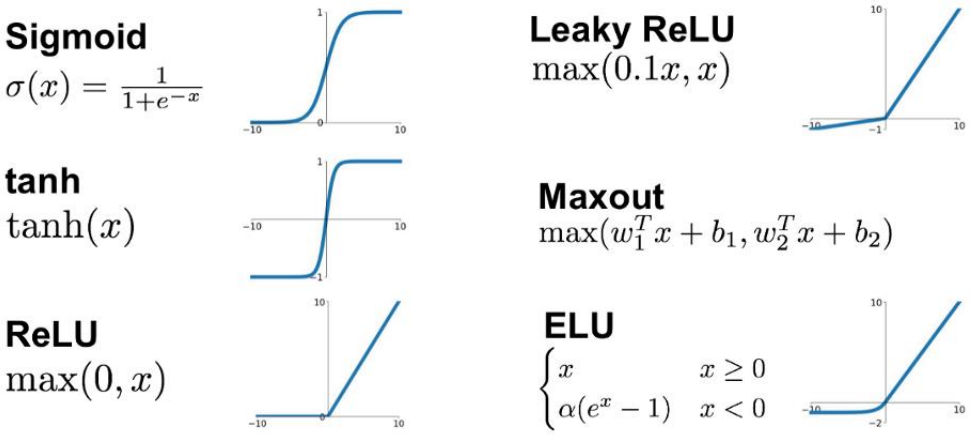

Figure 2.2: Frequently used activation functions

2.4.3.1 The sigmoid function

The sigmoid function, aka a special case of the logistic function, is $\sigma(x) = \frac{1}{1+e^{-x}}$. The sigmoid function squashes *non-linearly* its input into the range $[0, 1]$.

2.4.3.2 Tanh

A similar function to the sigmoid is $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Here the input values are squashed *non-linearly* into the range $[-1, 1]$

2.4.3.3 ReLU

Another simple *non-linearly* activation function is $ReLU(x) = max(0, x)$

2.4.3.4 Softmax

The softmax function is usually used in normalizing the output of a network to a probability distribution over predicted output classes. Softmax is frequently used as it is very common for neural networks to produce probabilities as output.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_i}}$$

2.4.4 Training

Neural Networks, in order to be able to perform their respective task, are trained with the use of a training set. Each element of the training set is comprised by an input vector $\vec{x}$ and a label y. The goal of the training phase is to find the values of parameters W_i and b_i for each layer i that lead to predictions $\hat{y}$ as close as possible to the actual label y with respect to a loss function L, that we will explain next.

2.4.4.1 Loss Function

In order to measure how close a prediction $\hat{y}$ is to y, we will use a function called the **loss function**, aka the **cost function**. The lowest the value of this function the highest the similarity of the prediction and the actual label. Therefore, the goal of training is to minimize the loss function over our training set.

One of the most common loss functions is **Mean Square Error (MSE)** and is computed as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Another very common loss function is **cross-entropy loss**. This function is the difference between two probability distributions for a given random variable or set of events. In a binary classification problem, the cross-entropy loss for a certain training instance x with predicted class $\hat{y}$ and actual class y is:

$$L_{CE}(y, \hat{y}) = -\log p(y|\vec{x}) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

In a multiclass problem, where we have C classes, the label $\vec{y}$ is a vector over the C classes of the true output probability distribution. In this case the cross-entropy loss for each prediction $\vec{\hat{y}}$ from a training instance $\vec{x}$ is calculated by

$$L_{CE}(\vec{\hat{y}}, \vec{y}) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

2.4.4.2 Gradient Descent

As we mentioned above, we want to minimize our loss function. In order to do that we will use an algorithm called **Gradient Descent (GD)**

Gradient descent is an iterative algorithm for finding a local minimum. In order to find the local minimum of a function we start with a random point and calculate the derivative (gradient) there and then we move towards the direction of the negative gradient. We repeat this process until either the gradient is 0, in which case we found our minimum, or until the gradient changes sign, in which case we decrease the step even more and keep moving towards the minimum. Even though this iterative algorithm does not guarantee that we will find the global minimum, and in fact it is very likely that we will not, it has been proven that if we can find a local minimum we can end up with good enough results without a very big amount of computations.

For a multi variable function $F(x)$ in order to go from a point a towards the local minimum we move to the direction of the negative gradient of F at a, $-\nabla F(a)$. It follows that, if $a_{n+1} = a_n - \gamma \nabla F(a_n)$ for $\gamma \in \mathbb{R}_+$ then $F(a_n) \geq F(a_{n+1})$. In other words the term $\gamma \nabla F(a)$ is subtracted from a because we want to move against the gradient, towards the local minimum. The term γ is called the **learning rate** and can be a constant or change in each iteration. The bigger the γ the faster we will converge towards the local minimum but it could also lead us to overshoot the minimum. A very little γ, on the other hand, can lead us more safely to the local minimum but it will take more time. Therefore, it is common to begin the learning rate at a higher value, and then slowly decrease it.

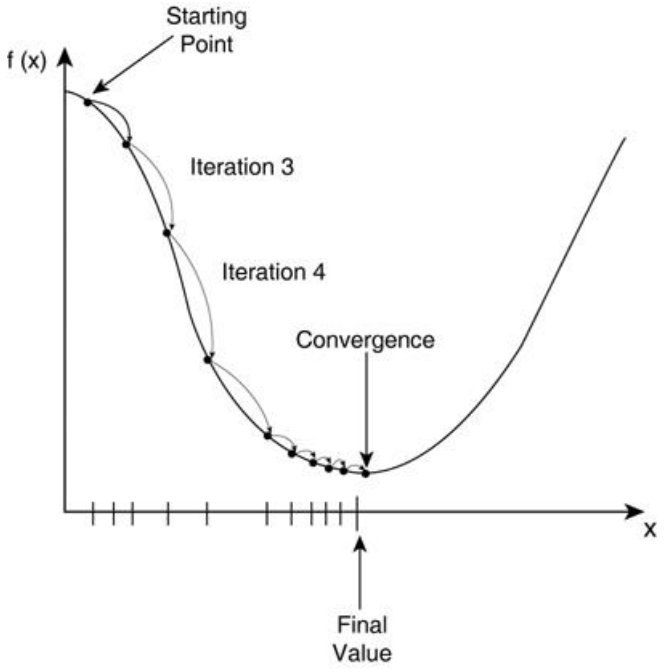

Figure 2.3: Simulation of Gradient Descent's convergence after multiple iterations

In practice, we use the whole dataset each time before updating the weights when we use GD to train an NN. As a result, we end up performing numerous calculations in each iteration for a single improvement.

For that reason variations of GD have been introduced, and in fact they are the ones that are mainly used. Two of the most popular are **Stochastic Gradient Descent (SGD)** and **Minibatch Gradient Descent**. SGD randomly shuffles all the training samples and updates the weights once after each full iteration, while Minibatch Gradient Descent splits the training data randomly into smaller subsets, called batches, and updates once after each batch. Both of them lead into converging faster than GD, but the odds of resulting in worse minimizations are higher.

2.4.4.3 Backpropagation

So far we have only talked about data going forward in a NN. That is, data entering in the input layer, passing through all the hidden layers and finally exiting through the output layer. That path is called the forward pass or forward propagation.

On the other hand, at the end of each iteration, that is when all the data of a batch have passed through the network, we can compute the gradient of the loss function with respect to the weights and the biases of the network and update them accordingly. Our goal is to update the weights and biases while trying to minimize the loss, and we achieve this with the gradient descent method mentioned above. This process is called **backpropagation**.

Neural networks can be described as a combination of function composition and matrix multiplication:

$$\hat{y}(x) = f^L(W^L f^{L-1}(W^{L-1} \cdots f^1(W^1) \cdot \cdot))$$

For a training set there will be a set of input-output pairs (x_i, y_i). The loss of the model is the cost of the difference between the predicted output $\hat{y}(x_i)$ and the target output y_i:

$$C(y_i, \hat{y}(x_i))$$

During the backpropagation process each weight can be calculated as $w_i' = w_i - \gamma \frac{\partial C}{\partial w_i}$ where w_i is the last value of the weight and $\frac{\partial C}{\partial w_i}$ is the partial derivative calculated with gradient descent and can be computed with the chain rule.

2.4.4.4 Training Risks

While training our model we usually split our data into two subsets, the training set that we use to train our model and the test set that we use after the training process in order to calculate its success rate. While training our model we want to avoid either **overfitting** or **underfitting**

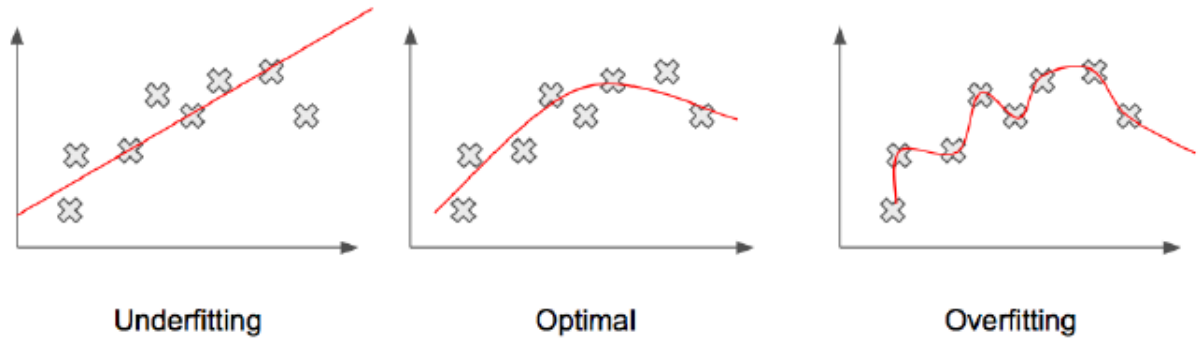

Figure 2.4: Examples of underfitting (left), overfitting (right) and the optimal balance between them (center).

Overfitting is when our model adapts too closely, or even exactly, to the training data and therefore fails to generalise and provide good predictions on unseen data despite the fact it achieves high accuracy for the training data. This usually happens as a result of

overtraining the model and/or it being more complex than can be justified by the structure of the training data. As a result, the model ends up with low accuracy in the test set.

Underfitting, on the other hand, occurs when our model is unable to adequately capture the underlying structure of the training data. This can be the result of undertraining the model or of it being too simple to fit the data for the task. Underfitting is reflected by low accuracy on both training and test sets.

Finding the right balance between the two is always important. In order to try and tackle these problems a technique named **Regularization** is utilized. One of the easiest and simplest regularization methods that are almost always applied, is to stop training when the test error starts increasing, despite the fact that the training error can be further minimized, in order to avoid overtraining our model. This technique is called early termination.

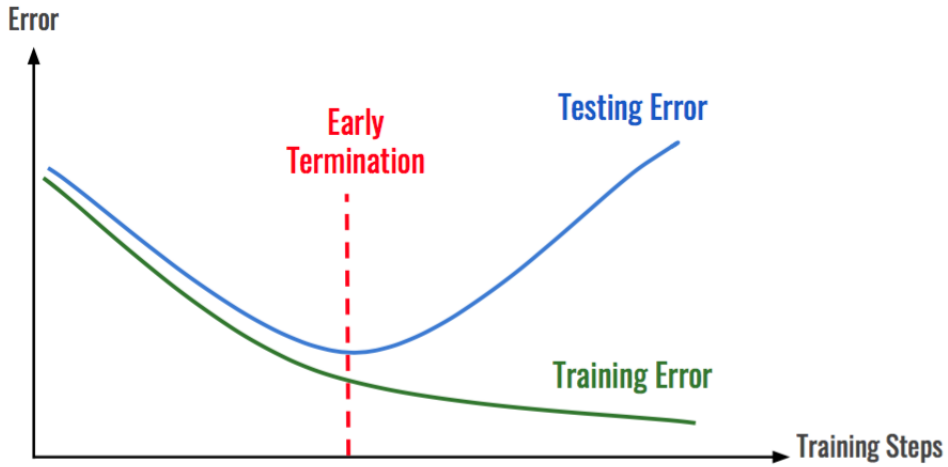

Figure 2.5: The ideal point to stop training based on the training error and the test error.

Another regularization method that is commonly applied is called dropout [22]. This method randomly deactivates neurons and/or connections between neurons during the training of a neural network, as shown in the image below.

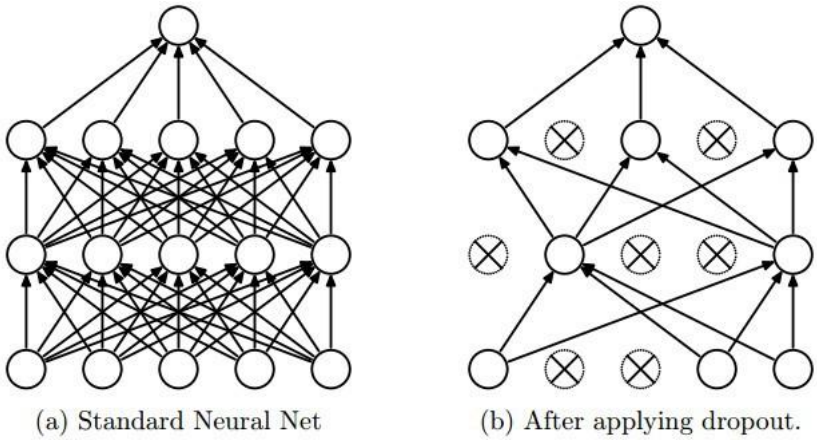

Figure 2.6: A FNN before (a) and after (b) applying dropout

This will make the network to not rely on specific neurons or connections in order to extract specific information during the training. After the training is finished, all the deactivated neurons and connections will be restored.

It is worth mentioning that the use of dropout layers make the training of models take longer

because more epochs are required to converge. Although, the training time of each epoch will be shorter because there are less neurons and connections [22].

2.4.5 Types of Neural Networks

Since we have discussed about all the major concepts of neural networks we will now introduce some kinds of neural networks that are widely used.

2.4.5.1 Feedforward Neural Networks

Feedforward Neural Networks (FNN) are the most common types of NNs. In these, the data moves only towards one direction since the neurons of one layer are only connected with neurons of the next layer. The data enters the network in the input layer, pass through all the hidden layers and exits the network from the output layer. The output of the output layer is the output of the network. An FNN containing at least one hidden layer (in addition, of course, to the input and the output layers) is more specifically called a **Multilayer Perceptron (MLP)**.

2.4.5.2 Convolutional Neural Networks

A special case of Neural Networks is **Convolutional Neural Networks (CNN)**. CNNs are regularized versions of MLPs. As their name implies convolutional networks employ the mathematical operation called convolution. These types of networks are very useful because they apply various filters and regularization techniques in order to decrease the number of parameters of the network. A simple example use of CNNs is for image recognition. Images are usually a number of pixels where each one has three colour planes, Red, Green and Blue. For a 48×48 image, we get $48 \cdot 48 \cdot 3 = 6912$ of parameters for the input layer which is a reasonable number. However, for a higher resolution image, such as a 4k image, whose resolution is 3840×2160, we will need $3840 \cdot 2160 \cdot 3 = 3 \cdot 24883200$ input parameters which is an exceptionally high number and definitely not manageable. Other uses of CNNs include image classification as well as video recognition.

2.4.5.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are another special form of MLPs. These networks allow the output of nodes to be fed not only in the next nodes in sequence, but also be fed as the input to the same nodes themselves or ones in previous layers. What this means is, that the output of each hidden layer is directed, except from the output layer, in the hidden layer again for the next input point in order. This is known as the hidden state. This allows them to exhibit temporal dynamic behavior. Derived from feedforward neural networks, RNNs use their internal state (memory) to process variable length sequences from input. This makes them useful for tasks such as speech recognition or handwriting recognition, because the output depends on previously seen information.

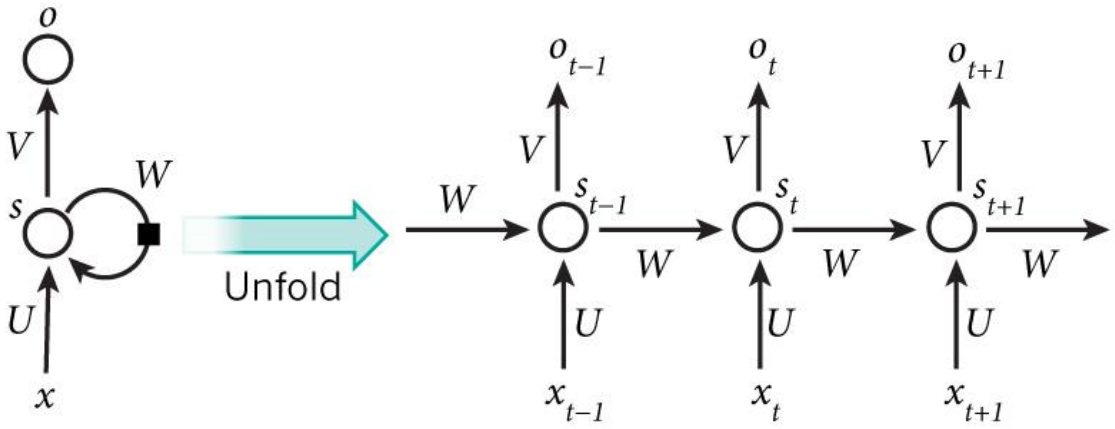

Figure 2.7: A Recurrent Neural Network, folded (left) and unfolded (right)

The normal depiction of RNNs is shown on the right in Figure 2.7. We can also depict them as normal neural networks like the left of the figure. In this case, if for example, we have a sequence of K words, we will have K layers, with each layer corresponding to each word. Let us now explain the notation of RNNs:

W, V and U are weights such as all the weights of MLPs. It is important to note that these weights are the same across all time steps. That is because they perform the same task with each different input. The input of each time step t is x_t, which would be a word in the hate speech detection problem. As we mentioned above RNNs have a hidden state. This is denoted by s_t which is the hidden state at the time step t. For the very first calculation at the time step 0, we require s_{-1}, which we usually initialise as 0. Finally, the network outputs o_t which is the output at time step t. In a word prediction problem, for example, that output would be the network's prediction for the next word. In case we are not interested about the intermediate outputs, because we may care only about the final output of the network, we can omit them.

A simple but commonly used extension of RNNs is that of **bidirectional RNNs** (figure 2.8). In practice these are two conventional RNNs stacked on top of each other. The intuition behind them is that the input flows from both directions, left to right and right to left. They are useful for certain problems that may require knowledge of future states. Such problems are for example text problems where we can extract grammatical or syntactical rules and make better predictions of the next word.

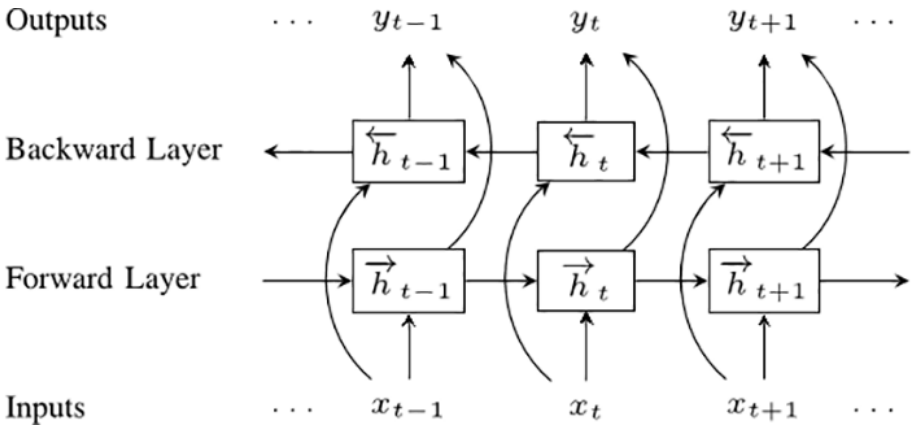

Figure 2.8: A bidirectional RNN

A major problem with RNNs is that of the vanishing gradient. This is the phenomenon where error gradients vanish exponentially quickly with the size of the time lag between important events and as a result temporal dependencies that span many time steps will effectively be discarded by the network. We will try to better illustrate this problem with an example.

The brown and black dog, which was playing with the cat, was a german shepherd.

(x_2) (x_4) (x_5) (x_{14}) (x_{15})

In a word prediction problem for the above example, if we want to predict the last two words ”german” and ”shepherd”, we would need knowledge from the very beginning of the sentence, more specifically we will have to take into account the words brown”, “black, and “dog” which describe the german shepherd. During the training phase of the model we would need to calculate the back propagation error of the term ”shepherd” all the way back to ”brown”. In order to do that we need to calculate the partial derivative $\frac{\partial h_{15}}{\partial h_2}$, which is written as $\frac{\partial h_{15}}{\partial h_2} = \frac{\partial h_{15}}{\partial h_{14}} \frac{\partial h_{14}}{\partial h_{13}} \cdots \frac{\partial h_2}{\partial h_1}$ because of the chain rule. The problem that arises is that when we multiply these gradients and their values are less than 1, it is possible that the loss of “shepherd” with regards to “brown” will approach 0, therefore “vanishing”. It becomes clear, that it is difficult to take into account words that appear near the beginning of long sequences. As a consequence, the word “brown” may not have any impact in the prediction of “shepherd” when doing a forward pass, since the weights were not updated because of the vanishing gradient [17].

Another problem that may also occur, and the ”opposite” of the above in a manner, is the problem of the **exploding gradient**. This issue emerges when we multiply gradients repeatedly with values greater than 1 and resulting with values exponentially high. This will, in turn, make the learning unstable. We can easily resolve this problem, to an extent, by clipping the gradients if their norm exceeds a given threshold. This method is called gradient clipping [20].

These are major problems from which RNNs suffer and because of that there has been done a lot of research surrounding them. As a result, there have some performance improvements in discrete cases [19]. In conventional RNN architectures, however, an efficient way to resolve these challenges, and particularly that of the vanishing gradients, has still not been discovered [13].

2.4.5.4 LSTM

Long short-term memory networks (LSTMs) were first proposed by Hochreiter and Schmidhuber (1997) [14]. Their main purpose was to help overcome problems caused by the conventional RNNs. Their key factor is that they are specialized in **long-term dependencies**, such as words from the beginning of a sentence or even a paragraph in a text example. Now let us delve deeper into LSTMs, as they will be our focus in this Thesis.

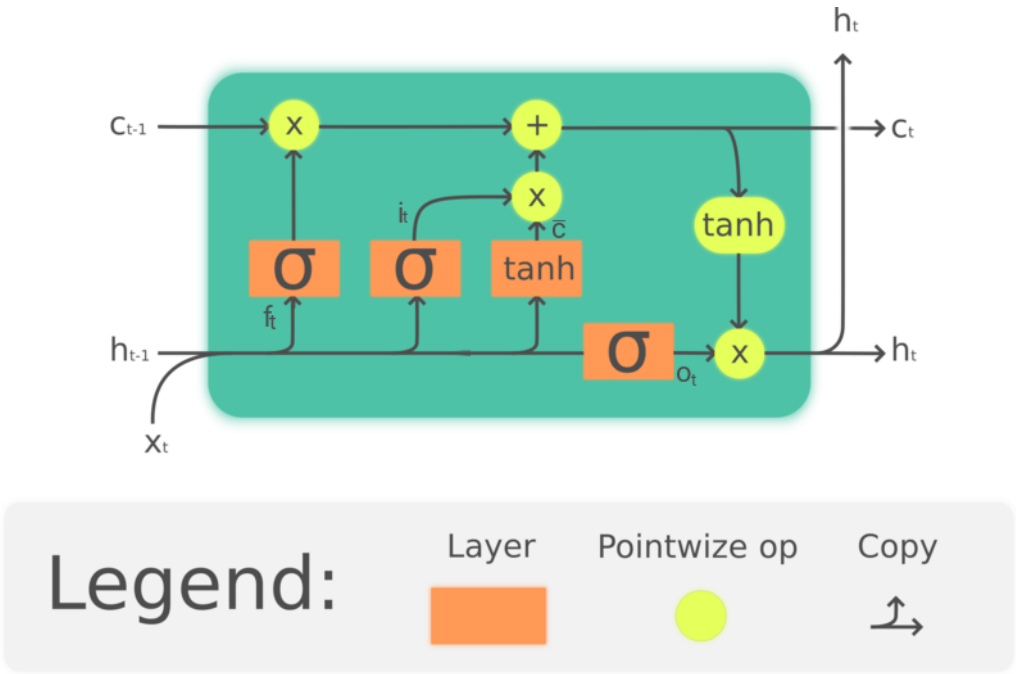

Figure 2.9: An LSTM cell

The core architecture of an LSTM unit is composed by the **cell state** and three ”regulators”, usually called gates, of the flow of information inside the LSTM unit: an **input gate**, an **output gate** and **a forget gate**.

- The **cell state** is essentially the ”memory” part of an LSTM as it stores information gathered over arbitrary time intervals. The gates regulate in what way information will be added or removed from the cell state. This regulated flow allows the network to learn long-term dependencies. The cell state is used for the production of the new hidden state. It is also worth noting that LSTMs can allow gradients to flow unchanged through the module making them able to partially solve the problem of the **vanishing gradient**. However, they still suffer from the **exploding gradient** problem.
- The **forget gate** is responsible for determining the information that should be kept or dropped from previous steps in the cell state.
- The **input gate** decides which values should be left in. It, also, gives weight to the values that will be passed, signifying their level of importance.
- Lastly, the **output gate** controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit.

In the equations below, the lowercase variables represent vectors. Matrices W_q and U_q contain, respectively, the weights of the input and recurrent connections, where the subscript q can either be the input gate i, the output gate o, the forget gate f or the memory cell c, depending on the activation being calculated. In this section, we are thus using a ”vector notation”. So, for example, $c_t \in \mathbb{R}^h$ is not just one cell of one LSTM unit, but contains h LSTM unit's cells. The dimensionality of the weight matrices and the bias vector are $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$ where d and h refer to the number of input features and number of hidden units, respectively. The notation of the activation functions is as follows,

σ_g is the sigmoid function, σ_c is the hyberbolic tangent function and σ_h is the hyperbolic tangent function as well or, as the peephole LSTM paper[9][10] suggests, $\sigma_h(x) = x$. The variables that we will use are $x_t \in \mathbb{R}^d$ as the input vector to the LSTM unit, $f_t \in \mathbb{R}^h$ as the forget gate's activation vector, $i_t \in \mathbb{R}^h$ as input/update gate's activation vector, $o_t \in \mathbb{R}^h$ as the output gate's activation vector, $h_t \in \mathbb{R}^h$ as the hidden state vector also known as output vector of the LSTM unit, $\bar{c}_t \in \mathbb{R}^h$ as the cell input activation vector and $c_t \in \mathbb{R}^h$ as the cell state vector. Since we cleared out the variables and the notation we will use, let us continue with introducing the equations for the forward pass of an LSTM unit with a forget gate.

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\bar{c} &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \bar{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

It becomes clear that the LSTM networks are more expensive to train, but their ability to overcome the vanishing gradient problem makes them a very common choice for many applications of recurrent networks. There have been introduced many variations of the LSTM, although, the basic one, that we just introduced, is still one of the more commonly used ones. The most notable variation of LSTM is the Gated Recurrent Unit (GRU), introduced by Cho et al [5] in 2014. GRU (Figure 2.10) differs from LSTM in that it combines the forget and input gates into a single update gate and it also merges the cell state with the hidden state. Despite their rising popularity they still fall short compared to LSTMs in certain tasks. It has been recently proven by Weiss et al. 2018 [25] that the LSTM is "strictly stronger" than the GRU as it can easily perform unbounded counting, while the GRU cannot. That's why the GRU fails to learn simple languages that are learnable by the LSTM. Furthermore, Britz et al. 2017 [3] have shown that LSTM cells consistently outperform GRU cells in "the first large-scale analysis of architecture variations for Neural Machine Translation".

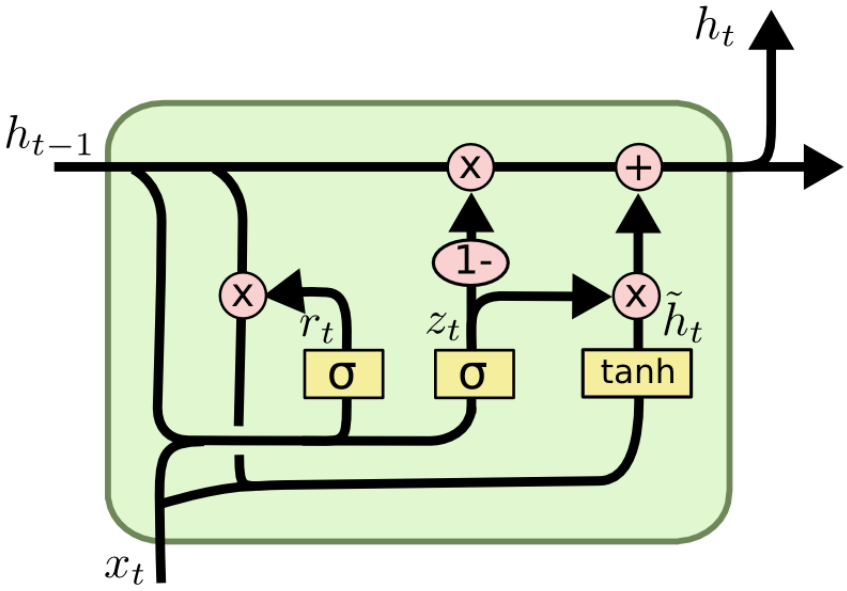

Figure 2.10: A GRU cell

3. HATE SPEECH DETECTION PROCESS

3.1 Task Definition

Our goal is to create an automatic tool that utilises Neural Networks, and more specifically LSTMs, in order to detect hate speech in text. Our focus would be on data extracted from social media, and more specifically the social media platform Twitter.

In this chapter, we will describe the dataset we used and any preprocessing we performed on it. Moreover, we will discuss about different ways of transforming our data from text format to a format that can be accepted by our models. Finally, we will construct LSTMs, that we will then train with the aforementioned processed and transformed data in order to be able to conclude whether a provided input consists hate speech or not.

3.2 Toolset

The development of the code was done in Python 3.6. We used the prominent library Keras, which runs Google's TensorFlow backend, in order to design and train our RNNs. Keras makes building powerful NNs very easy and simple as it has a high level of abstraction. More specifically it allows us to specify layers, activation and loss functions, compile and finally train a model in a matter of a few lines. Some other notable Python libraries were also used in order to assist us in various ways, some of which are pandas that allows easy data handling, numpy that is a high level math library and others.

3.3 Datasets

As we have mentioned above, we will need a set of annotated data that our model will use to be trained and evaluated. In this Thesis our focus is Hate Speech in social media so we will need data extracted from the multitude of social media platforms that exist. The dataset we will use is the one used by Davidson et. al (2017) [8]. In their work, the authors gathered the data using the Twitter API and by searching for hate speech keywords from the Hatebase.org. The data was then labeled by using CrowdFlower. This dataset contains $24,783$ Tweets, each one classified in one of three classes, hate speech, offensive language, or neither. In this work, we will focus in binary classification, that is, we will have only two target classes, hate speech and neutral. Therefore, in order to eliminate the extra class we decided to combine hate speech and offensive language in one hate speech class.

3.4 Preprocessing

When training a machine learning model, a lot of times, it is important to process the data so as to enhance their quality. In our case we will first remove the text that indicates if a tweet is a retweet. Moreover, we make the text to lowercase. We will, then, do what is called unpacking contractions, that is replacing "I'm" with "I am" and so on. Furthermore, we will remove stop words, dates and times. Stop words are considered words that offer

no additional information. In this work, we used the stopwords provided by the nltk python library. In addition, we replace urls, hashtags and user names with tags. Meanwhile, we enclose in tags censored swear words, such as words of the form f**k, repeated occurrences, that are also replaced by a single one, and words that appear in all capitals. Finally, we replace all whitespaces with a single space character.

Consider for example the following tweet *"This is just an example #tweet THAT also links to www.foo.bar!!!! F**k"*. Also let us assume that the words "it" and "an" are stopwords. This tweet will then be transformed to *"this just example <hashtag> <allcaps> that <\allcaps> also links to <url> <repeated> ! <\repeated> <censored> f**k <\censored>"*

3.5 Text Representation

Since we processed our text and kept only useful info, we will then need to train our model. But in order to do that we have to transform the data in a format that our LSTM NNs can accept and process it.

3.5.1 Bag of Words

The simplest text representation technique is **Bag of Words** (BoW). In this model, the text is split into its words and then a histogram is created showing the frequency of each word in each text. The Bag of Words method does not take into account the syntax or grammar of the original text. A downside of BoW is that highly frequent words start to dominate in the document but may not contain as much "informational content" to the model as rarer but perhaps domain specific words.

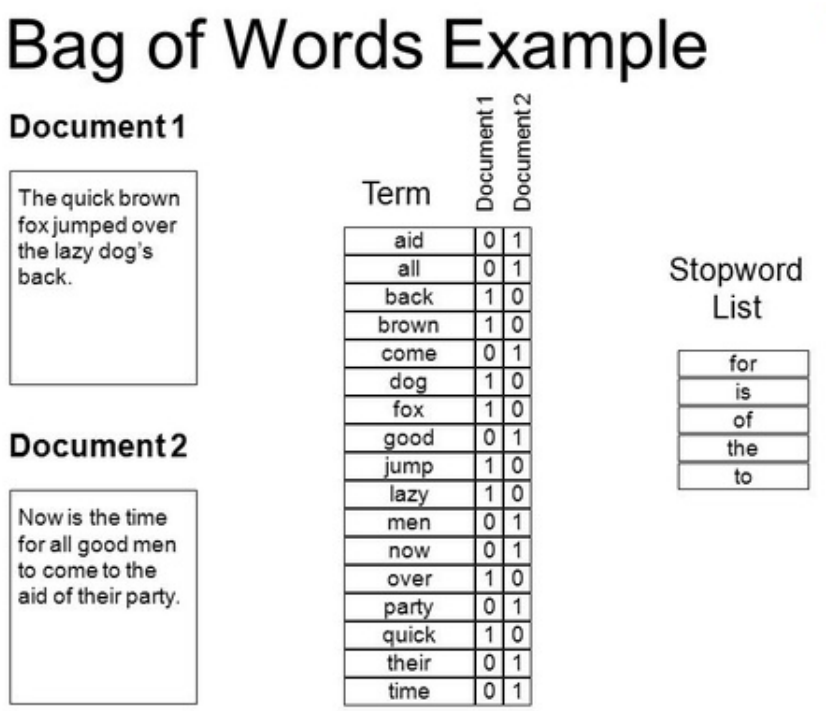

Figure 3.1: Bag of Words example

3.5.2 Term Frequency - Inverse Document Frequency

In order to anticipate the main issue of BoW, another approach has been introduced, **Term Frequency - Inverse Document Frequency** (TF-IDF). TF-IDF is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. It is comprised of two statistics, **term frequency** and **inverse document frequency**.

- Term Frequency $tf(t, d)$ is the number of times a word t appears in a text d.
- Inverse Document Frequency $idf(t, D)$, where D is the dataset, is a measure of how much information the word provides, i.e., if it is common or rare across all documents.

TF-IDF is calculated as $tfidf(t, d, D) = tf(t, d) \cdot idf(t, D)$. The tf and idf terms, as well as the $tfidf$ value can be computed with other ways too, but these are the simplest and most common ones.

Sentence 1 : The car is driven on the road.

Sentence 2: The truck is driven on the highway.

In this example, each sentence is a separate document (A,B).

Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	log(2/2) = 0	0	0
Car	1/7	0	log(2/1) = 0.3	0.043	0
Truck	0	1/7	log(2/1) = 0.3	0	0.043
Is	1/7	1/7	log(2/2) = 0	0	0
Driven	1/7	1/7	log(2/2) = 0	0	0
On	1/7	1/7	log(2/2) = 0	0	0
The	1/7	1/7	log(2/2) = 0	0	0
Road	1/7	0	log(2/1) = 0.3	0.043	0
Highway	0	1/7	log(2/1) = 0.3	0	0.043

Figure 3.2: TF-IDF example

While both Bow and TF-IDF are very simple to implement and to understand they come with some shortcomings. Mainly, because they do not take into account word order they ignore the context of a sentence. They also do not take into account word semantics.

3.5.3 Word Embeddings

Word Embeddings is a very common representation technique, that maps each word in a language's vocabulary to a vector of real numbers. This method allows to represent similar words with similar vectors and take into account word semantics. While used in conjunction with RNNs, each word's vector can be fed sequentially to it and as a result take into consideration word order.

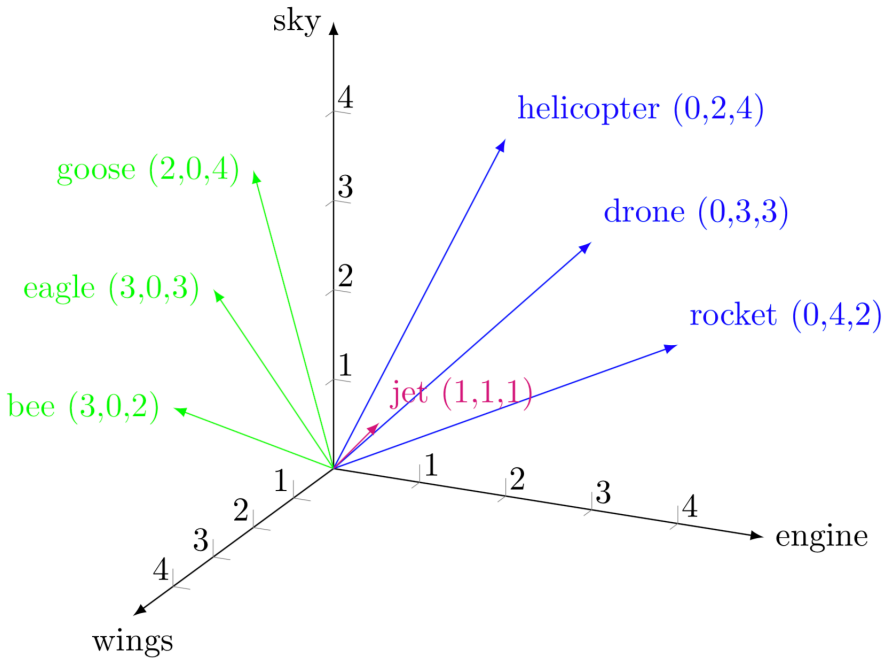

Figure 3.3: Word Embeddings example

There are many ways to generate embeddings for words. In this Thesis we will use the already trained GloVe [21] word embeddings with 200 dimensions. In case a word does not exist in the the GloVe embeddings we generate a random embedding for it.

Another technique, that we will also use, is to produce word embeddings by adding an embedding layer as the input layer of an LSTM model. What this layer does, is to take a word id as input and produce embeddings for it. In order to implement this technique, we map each word to an id and then feed to the network a sequence of ids instead of a sequence of words. This layer is then trained along with the rest of the model and learns to produce word embedings.

3.6 Model Design

Since we have prepared our data, it is now time to train the model. We will experiment with one and two LSTM layers. Having multiple LSTM layers can help identify and learn complex relations between the data, but the more layers are added the more likely is to end up overfiting our model. Because of that we will compromise with a small amount of LSTM layers. In general, we can experiment with the number of LSTM layers and pick the best.

Another technique, we will attempt, is building bidirectional models. As mentioned earlier, in bidirectional models we process data both from left to right and from right to left. This is useful in text data because it can help us better understand the full context of a sequence.

After the last LSTM layer, we add a dropout layer. This is a common practice in order to avoid overfitting. We, then, average the outputs of each time step in order to get a single vector for each tweet and in turn feed this in a fully connected layer of 32 nodes. Lastly, the output of those 32 nodes are passed to our output layer, which is comprised by two neurons, each of which outputs the probability of the input belonging in one of our two classes. In addition, when we will not use prepared embeddings, another layer will be

added before the first LSTM layer, that will be trained and learn to produce embeddings for the words.

The model will use the **Adam** (Adaptive Moment Estimation) [18] optimizer, which is a modification of the stochastic gradient descent we mentioned earlier. For the loss function the **cross-entropy** loss function was used.

4. RESULTS

In this chapter we will provide the results of our models and compare them in order to find which one performs the best. We will also compare our work with other similar works.

In order to measure how good a model is we will need some metrics that can help us identify when a model performs good or not. The first metric we will use is **accuracy**. Accuracy is one of the most common metrics used in ML and is defined as $\frac{\text{correct predictions}}{\text{total predictions}}$. However, sometimes accuracy is not the best choice. We will portray how accuracy can fall short in the following example.

Assume the following confusion matrix

		Predicted/Classified	
		Negative	**Positive**
Actual	**Negative**	998	0
	Positive	1	1

Figure 4.1: An example to portray the shortcomings of accuracy

In this example the accuracy is 99.9% which is incredibly high. One can believe that this is a very good model that almost never fails. Although, that is not the case, as it can be seen that its performance for classifying positive elements is not as good. More specifically, while in this task it misclassified a single element, this might important in some tasks. It becomes obvious that accuracy is not always the best choice for a metric.

For that reason we will use a second metric as well called **F1-score**. F1-score is calculated from the precision and recall of the test, where the precision is the number of correctly identified positive results divided by the number of all positive results, including those not identified correctly, and the recall is the number of correctly identified positive results divided by the number of all samples that should have been identified as positive. The F1 score is given by the following type:

$$F1 = 2\frac{precision \cdot recall}{precision + recall}$$

We will all also look how well our classifier performs with regards to each class individually. We will do that by examining the precision and the F1-score for each class.

Below we present the results for all the different models we tried. We highlight with green the best results in each column.

Table 4.1: Accuracy and F1-score of models with one LSTM layer and glove embeddings

Layers	Epochs	Bi-directional	Accuracy	F1-score
1	5	Yes	92.38	84
1	10	Yes	93	84
1	15	Yes	93.98	84
1	20	Yes	93.8	84
1	5	No	93.32	93
1	10	No	93.59	88
1	15	No	94.64	90
1	20	No	94.88	92

Table 4.2: Precision and F1-score for each class for models with one LSTM layer and glove embeddings

Layers	Epochs	Bi-directional	Hate precision	Hate F1-score	Neutral precision	Neutral F1-score
1	5	Yes	84	91	0	0
1	10	Yes	84	91	0	0
1	15	Yes	84	91	86	11
1	20	Yes	84	91	0	0
1	5	No	95	96	82	77
1	10	No	88	93	84	44
1	15	No	97	94	64	74
1	20	No	95	95	76	76

Table 4.3: Accuracy and F1-score of models with two LSTM layers and glove embeddings

Layers	Epochs	Bidirectional	Accuracy	F1-score
2	5	Yes	83.59	84
2	10	Yes	83.86	84
2	15	Yes	83.90	81
2	20	Yes	95.27	95
2	5	No	90.74	84
2	10	No	93.71	84
2	15	No	92.53	65
2	20	No	94.96	94

Table 4.4: Precision and F1-score for each class for models with two LSTM layers and glove embeddings

Layers	Epochs	Bidirectional	Hate precision	Hate F1-score	Neutral precision	Neutral F1-score
2	5	Yes	84	91	0	0
2	10	Yes	84	91	0	0
2	15	Yes	87	89	40	36
2	20	Yes	98	97	81	86
2	5	No	84	91	0	0
2	10	No	84	91	0	0
2	15	No	100	73	32	48
2	20	No	98	94	67	76

Table 4.5: Accuracy and F1-score of models with one LSTM layer and an embeddings layer

Layers	Epochs	Bidirectional	Accuracy	F1-score
1	5	Yes	83.67	84
1	10	Yes	83.71	84
1	15	Yes	83.43	84
1	20	Yes	83.86	84
1	5	No	92.81	93
1	10	No	93.59	88
1	15	No	93.82	94
1	20	No	83.78	84

Table 4.6: Precision and F1-score for each class for models with one LSTM layer and an embeddings layer

Layers	Epochs	Bidirectional	Hate precision	Hate F1-score	Neutral precision	Neutral F1-score
1	5	Yes	84	91	0	0
1	10	Yes	84	91	0	0
1	15	Yes	84	91	0	0
1	20	Yes	84	91	0	0
1	5	No	97	96	74	79
1	10	No	84	91	0	0
1	15	No	97	96	79	81
1	20	No	84	91	0	0

Table 4.7: Accuracy and F1-score of models with two LSTM layers and an embeddings layer

Layers	Epochs	Bidirectional	Accuracy	F1-score
2	5	Yes	83.71	84
2	10	Yes	83.86	84
2	15	Yes	83.55	84
2	20	Yes	83.86	84
2	5	No	83.71	84
2	10	No	92.73	93
2	15	No	93.86	94
2	20	No	83.78	84

Table 4.8: Precision and F1-score for each class for models with two LSTM layers and an embeddings layer

Layers	Epochs	Bidirectional	Hate precision	Hate F1-score	Neutral precision	Neutral F1-score
2	5	Yes	84	91	0	0
2	10	Yes	84	91	0	0
2	15	Yes	84	91	0	0
2	20	Yes	84	91	0	0
2	5	No	84	91	0	0
2	10	No	96	96	77	78
2	15	No	96	96	84	81
2	20	No	84	91	0	0

We notice, that in many cases, the F1-score and precision of the Neutral class are zero. This is because the specific models classify everything in the hate speech class. Meanwhile, we see that the overall accuracy and F1 scores of those models are relatively high. One possible reason, for this, is because there are a lot more tweets in the hate speech class than the neutral speech class in our dataset. So, even if everything ends up classified as hate the the overall accuracy is not affected that much.

Looking into our dataset we see that there are $24,783$ tweets in total, of which $20,620$ are classified as hate speech and $4,163$ are classified as neutral speech. Our training data consists of 80% of our dataset, and more specifically $16,489$ tweets classified as hate speech and $3,337$ tweets classified as neutral speech. The tweets classified as hate speech consist around 80% of our training set. Because of this large difference some of our models ended with poor performance when detecting neutral speech. One way to eliminate this problem is to find a more uniform dataset where the amount of elements in each class will not have such big difference. We can also try and make more complex models that can capture the underlying structure of the data better, but if we make it too complicated we risk overfitting our model as mentioned earlier.

Another thing, to point out, is the fact that the models that used **GloVe** embeddings performed in general better than the models that trained a layer to generate embeddings. This could possibly happen due to the fact that GloVe embeddings are in general better and more fitting for our task compared to the ones generated by our model.

The best overall model is the one that uses two bidirectional LSTM layers, GloVe embeddings and was trained for 20 epochs. This model has the best overall accuracy and

F1-score, as well as the best or near best precision and F1-score in each individual class. This was in general expected as this model uses two LSTM layers that allow it to learn more complex patterns. It is also bi-directional which allows it to have a better view of the full context of each tweet, while we also trained it for 20 epochs allowing it more time to learn.

4.1 Comparison to similar works

The first work we will compare to ours is by Akanksha et al. 2020 [2]. In their work they used the same dataset with ours but they also expanded it with another dataset of 15k tweets in order to balance it. Eventually they used 9600 tweets as the training set, 3600 for each of the three original classes and 800 tweets as the test set. They tried deep learning models with one and three LSTM bi-directional and non bi-directional layers followed by three fully connected layers and finally the output layer. Their models achieved an accuracy between 83% and 86%. As we can see our best model outperformed theirs since the best accuracy we achieved was 95%.

Another work we will consider for our comparisons is that of Zhang et al. 2018 [26]. In their architecture, the first layer was a word emeddings layer that mapped the input to embeddings of 300 dimensions. They then added a dropout layer and its output was fed into a 1-D convolutional layer with 100 filters. This convolves the input into a 100×100 representation which is further downsampled by a 1-D max pooling layer. They, then, use a GRU layer followed by another 1-D max pooling layer which essentially flattens the output. Finally, a softmax layer predicts the probability for each class. They used various datasets for their experiments including the same as us which is the one in our interest. They used F1 score as a metric for their model and achieved a score of 94%, almost as much as our best F1 score of 95%.

5. CONCLUSION AND FUTURE WORK

5.1 Conclusions

In this study, we discussed why it is important to deal with the Hate Speech phenomenon. The massive use of the Internet, accompanied with the increase of hatred against minorities in modern societies, has led to an influx of Hate Speech within online platforms. The large amount of data that is shared through the Internet has made the detection of Hate Speech by humans very difficult. Therefore, it becomes clear that other methods should be employed in order to effectively detect Hate Speech. For that reason it is important for automatic tools to be created that can effectively tackle this problem.

As part of this thesis we implemented and compared various classifiers that classify tweets as neutral speech or hate speech. These classifiers are LSTM neural networks and were trained with a dataset of $24,783$ tweets. We concluded that the best one was built with two bi-directional LSTM layers, used GloVe embeddings and was trained for 20 epochs. We consider this model to be quite successful, as not only it had the best overall accuracy and F1-score, but it performed very well in deciding between the two classes.

5.2 Future Work

A potential future improvement is to include more features such the contribution of word roles (e.g. POS tags) or twitter specific features and combine them with improved preprocessing, to avoid possible noise in the related features. We can further expand on this and instead of single words utilize n-grams, that is sequences of n consecutive tokens. In practice we used one-grams, also called unigrams, in this work.

We could also experiment with adding more LSTM layers or another dense layer before the output one in the hopes that a deeper network will be able to learn more complex patterns of the data and potentially achieve better results.

Other than that, future work may also include training a similar model on a different dataset.

Lastly, we could use characteristics of the people expressing hate speech, as mentioned by Davidson et al. 2017 [8], as well as test the importance of the user network features, as mentioned in Badjatiya et al., 2017 [1] in order to improve hate speech detection overall.

ABBREVIATIONS - ACRONYMS

ANN	Artificial Neural Network
BoW	Bag of Words
CNN	Convolutional neural network
DNN	Deep Neural Network
FNN	Feedforward Neural Networks
GBDT	Gradient Boosted Decision Tree
GD	Gradient Descent
GRU	Gated Recurrent Unit
LSTM	Long-short term memory
MGD	Minibatch Gradient Descent
MLP	Multilayer Perceptron
MSE	Mean Square Error
NLP	Natural Language Processing
NN	Neural Network
RNN	Recurrent neural network
SGD	Stohastic Gradient Descent
SVM	Support Vector Machine
TF-IDF	Term frequency - inverse document frequency

BIBLIOGRAPHY

[1] Pinkesh Badjatiya, Shashank Gupta, Manish Gupta, and Vasudeva Varma. Deep learning for hate speech detection in tweets. *Proceedings of the 26th International Conference on World Wide Web Companion - WWW '17 Companion*, 2017.

[2] Akanksha Bisht, Annapurna Singh, H. Bhadauria, Jitendra Virmani, and Kriti . *Detection of Hate Speech and Offensive Language in Twitter Data Using LSTM Model*, pages 243–264. 03 2020.

[3] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures, 2017.

[4] Alexander Brown. What is hate speech? part 1: The myth of hate. *Law and Philosophy*, 36(4):419–468, 2017.

[5] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[6] Teresa Correa, Amber Hinsley, and Homero Gil de Zúñiga. Who interacts on the web?: The intersection of users' personality and social media use. *Computers in Human Behavior*, 26:247–253, 03 2010.

[7] George Cybenko. Approximation by superpositions of a sigmoidal function. math cont sig syst (mcss) 2:303-314. *Mathematics of Control, Signals, and Systems*, 2:303–314, 12 1989.

[8] Thomas Davidson, Dana Warmsley, Michael Macy, and Ingmar Weber. Automated hate speech detection and the problem of offensive language, 2017.

[9] Felix Gers and E. Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *Neural Networks, IEEE Transactions on*, 12:1333 – 1340, 12 2001.

[10] Felix Gers, Nicol Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of Machine Learning Research*, 3:115–143, 01 2002.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, chapter 7.8, pages 239–242. 2016.

[12] Jeff Heaton. *Introduction to Neural Networks for Java*, chapter 5, pages 128–131. Heaton Research, Inc., 2nd edition, 2008.

[13] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

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

[15] Clement J. Percentage of u.s. population who currently use any social media from 2008 to 2019. `https://www.statista.com/statistics/273476/percentage-of-us-population-with-a-social-network-profile/#statisticContainer`.

[16] Daniel Jurafsky and James Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, volume 2. 02 2008.

[17] Ben Khuong. The basics of recurrent neural networks (rnns). `https://medium.com/towards-artificial-intelligence/whirlwind-tour-of-rnns-a11effb7808f`.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[19] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2012.

[20] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

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

[22] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[23] Zeerak Waseem. Are you a racist or am I seeing things? annotator influence on hate speech detection on Twitter. In *Proceedings of the First Workshop on NLP and Computational Social Science*, pages 138–142, Austin, Texas, November 2016. Association for Computational Linguistics.

[24] Zeerak Waseem and Dirk Hovy. Hateful symbols or hateful people? predictive features for hate speech detection on Twitter. In *Proceedings of the NAACL Student Research Workshop*, pages 88–93, San Diego, California, June 2016. Association for Computational Linguistics.

[25] Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision rnns for language recognition, 2018.

[26] Ziqi Zhang, David Robinson, and Jonathan Tepper. Detecting hate speech on twitter using a convolution-gru based deep neural network. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web*, pages 745–760, Cham, 2018. Springer International Publishing.