UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

CYBERTHREAT DISCOVERY IN OPEN SOURCE INTELLIGENCE USING DEEP LEARNING TECHNIQUES

Eunice Picareta Branco

Mestrado em Informática

Dissertação orientada por:

Professor Doutor Pedro Miguel Frazão Fernandes Ferreira
Professor Doutor Alysson Neves Bessani

2017

Acknowledgments

I would first like to thank my advisor Professor Pedro Ferreira. The door to Professor Ferreira's office was always open whenever I ran into a difficult spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought that was needed.

I would like to thank my co-advisor, Professor Alysson Bessani, for his support throughout my dissertation.

I would also like to thank and acknowledge the Diversity Enhancements for Security Information and Event Management (DiSIEM) [1] project and Horizon 2020 programme [2] for giving me the opportunity to carry my research work in the midst of such an interesting European project and for granting me a nine month fellowship to carry out the work reported here.

A kind word is in order for the research unit and for its research team at the Department of Informatics in which I was involved: Large-Scale Informatics Systems Laboratory (LaSIGE) [3] and Navigators [4], which allowed me to grow as a researcher.

As important as it was to have technical support, I will never forget the support from my family and friends. A special thank you for all those who believed in me and helped me get where I am.

Finally, I must express my very deepest gratitude to my partner for providing me with unfailing support throughout my years of study and through the process of researching and writing this dissertation. This accomplishment would not have been possible without him. This dissertation stands as a testament to your unconditional love and encouragement. Thank you.

To all who helped me get here.

Resumo

Face à necessidade crescente de se processar grandes quantidades de dados relativos a ameaças de segurança, fomos cativados pelo desafio da descoberta de ameaças cibernéticas em fontes abertas através do uso de técnicas de aprendizagem automática. Em termos de dados, isto significa que trabalhámos com informação recolhida de fontes abertas como o Twitter.

O que distingue o nosso trabalho encontra-se no modo como escolhemos abordar este desafio. A nossa hipótese é a de que processar tais quantidades de dados através de métodos de aprendizagem automática representa uma vantagem significativa em termos de eficiência e adequação, pelo que recorremos a redes neuronais.

Escolhemos esta abordagem uma vez que as abordagens de aprendizagem automática têm vindo a ganhar destaque merecido uma vez que asseguram uma maneira robusta de resolver um número de tarefas extremamente complexas no contexto de problemas de *big data*.

Esta dissertação introduz conceitos e noções gerais em que o nosso trabalho se baseia, apresenta o trabalho relacionado consultado por forma a ser eventualmente útil em trabalhos futuros, apresenta também o trabalho que realizámos, os resultados obtidos, e elenca sugestões sobre linhas de progresso promissoras e trabalho futuro.

Antes de discutir resultados, é necessário começar por introduzir conceitos centrais, o primeiro dos quais sendo o de aprendizagem automática.

Aprendizagem automática (*machine learning*) pode ser definida como a área ou abordagem da inteligência artificial de forma a que o sistema tenha a aptidão de aprender e melhorar com a experiência. Isto significa que não é necessária programação explícita para resolver o problema de partida pois o sistema de aprendizagem procura por regularidades nos dados e adquire a capacidade de tomar melhores decisões com base nos dados de exemplo que recebe.

Aprofundando esta abordagem, uma rede neuronal é um paradigma de processamento inspirado no modo como processos biológicos nervosos, como os que ocorrem no cérebro humano, processam informação.

A chave deste paradigma é a conexão entre os elementos básicos do sistema. Este é composto por um grande número de elementos de processamento, os neurónios, organizados em rede que entregam as suas saídas uns aos outros para resolverem proble-

mas específicos, cabendo notar que uma rede neuronal é tipicamente condicionada no seu desenho pelo problema que se pretende que resolva, ou seja, é configurada para uma única aplicação (*e.g.* reconhecimento de padrões, classificação de dados, *etcetera*).

De entre as técnicas de aprendizagem automática, a aprendizagem profunda (*deep learning*) tem adquirido grande relevância e vários projectos têm procurado explorar as suas vantagens. Trata-se de uma subárea da aprendizagem automática, e em particular das redes neuronais, sendo que o que distingue esta abordagem consiste no facto de os dados de entrada passarem por várias camadas funcionais de neurónios, usualmente não lineares, até serem totalmente processados.

No nosso projecto, a rede neuronal foi aplicada na resolução do problema que consiste na classificação de *tweets* em itens que se referem a uma ameaça de segurança, ou itens não relevantes a esse respeito. Com essa finalidade, foi implementada uma rede neuronal convolucional, que comparativamente necessita de pouca intervenção humana para ser posta a funcionar.

A vantagem de se aliviar a necessidade de tal intervenção também se prende com o tipo da rede, que pode ser supervisionada ou não supervisionada. Em aprendizagem supervisionada, um conjunto de dados de treino injectado na rede é composto por pares de entrada/saída, sendo que a entrada é tipicamente composta por um vector e a saída é o resultado pretendido para a entrada respetiva. A rede é treinada sobre todo o conjunto de dados para depois ser aplicada a novas situações ou dados de entrada desconhecidos. É assim necessário que o algoritmo de processamento generalize a partir dos dados de treino.

No caso da aprendizagem não supervisada, os dados injectados na rede são apenas de entrada, o que obriga a rede a inferir funções que descrevem a possível estrutura subjacente aos dados, pois a sua classificação explícita não é fornecida à rede. Como os dados não estão associados à sua classificação, não é trivial avaliar a adequação do resultado obtido pela rede neste caso.

Outro conceito importante é o de redes profundas (*deep*) *vs.* rasas (*shallow*). As redes neuronais são organizadas por camadas. Estas camadas são compostas por nós inter-conectados que contêm funções de activação, compreendendo a camada de entrada, as camadas escondidas, que pode englobar várias camadas para processamento de dados, e a camada de saída.

O termo redes rasas é usado para descrever as redes que contêm apenas uma ou duas camadas escondidas, que são funcionalmente idênticas. No caso de redes profundas, estas tendem a ter mais camadas escondidas, com grupos de camadas com funcionalidades distintas. A terminologia mais comummente aceite é a de que para uma rede ser considerada profunda tem de conter pelo menos três camadas que são escondidas e funcionalmente distintas.

As redes convolucionais são redes profundas compostas por várias camadas com funções não lineares aplicadas em cada nó. Em redes normais, cada neurónio de entrada está conectado a um neurónio de saída na camada seguinte. As redes neuronais convolucionais, por sua vez, optam antes por aplicar convoluções sobre a camada de entrada para computar a saída, em que cada região de entrada está conectada a um neurónio de saída, consistindo numa rede de conexões locais.

Outro aspecto relevante das redes convolucionais é o de que durante a fase de treino, a rede aprende os valores dos seus filtros automaticamente baseando-se na tarefa a ser aprendida e executada. A última camada destas redes é então um classificador que usa as características (*features*) de alto nível inferidas pela rede.

Como acabámos de assinalar, uma rede profunda tem várias camadas escondidas e esse é o modelo da rede que adoptámos no nosso trabalho.

A primeira camada da nossa rede transforma palavras, e como consequência *tweets*, em vectores. Depois desta camada, passa-se às camadas de convolução, que iteram sobre os vectores de palavras embutidos (*word embeddings*) realizando convoluções sobre múltiplos filtros com janelas de dimensões diferentes. No nosso caso, optámos por ter três filtros, sendo que cada um itera sobre uma quantidade de palavras diferente para cada convolução.

De seguida, para evitar que a rede se torne demasiado específica aos dados de treino (*overfitting*), temos uma camada de abandono (*dropout*) que obriga 50% dos neurónios a desligarem-se por forma a que os neurónios não se co-adaptem em demasia e por conseguinte sejam capazes de aprender características utéis individuais e independentes. Por último, uma camada de *softmax* é usada para classificar os dados de saída como positivos (*tweet* que menciona ameaças de segurança), ou negativos (caso contrário).

Mesmo com uma rede convolucional, é preciso acertar vários parâmetros para que a rede seja eficiente e produza bons resultados. Após ter uma base de parâmetros com que a rede produz bons resultados, tratámos de avaliar com recurso a validação cruzada (*cross validation*) os parâmetros óptimos para a rede, variando apenas aqueles que verificámos que produziam a maior diferença nos resultados.

Um dos parâmetros que foi feito variar foi o tamanho de um *batch*. Na análise dos nossos resultados, verificamos que tamanhos menores de *batch* levam a resultados piores. Atribuímos estes resultados piores ao facto de a rede treinar demasiado sobre o mesmo conjunto de dados, pois um *batch* menor implica um número maior de passos (*steps*) sobre um mesmo conjunto de dados.

Outra procura de melhorar o desempenho da rede consistiu em tomar *tweets* que são positivos para uma dada infraestrutura e adicioná-los ao conjunto de dados para outra infraestrutura como *tweets* negativos (*e.g.* um *tweet* positivo para a Oracle é adicionado como um *tweet* negativo para o Chrome).

Em geral, o conjunto de dados de base obteve melhores resultados do que quando era assim modificado, sendo que atribuímos esta diferença ao facto de os dados de treino ficarem demasiado desequilibrados entre *tweets* positivos e negativos. De notar no entanto, que o conjunto de dados assim modificado teve, em geral, menos variância de resultados entre *batches*, devido provavelmente ao conjunto de dados de treino ser mais extenso.

Não obstante a diferença de parâmetros, em geral a nossa rede apresentou bons resultados. Face aos resultados francamente positivos obtidos achamos que a instalação da nossa solução num centro de segurança operacional é viável e ajudará a detectar informação relevante acerca de várias ameaças possíveis que é veiculada de forma massiva através de *tweets*.

Palavras-chave: inteligência artificial, processamento de linguagem natural, redes neuronais, aprendizagem profunda, detecção de ameaças de segurança

Abstract

Responding to an increasing need to process large amounts of data regarding security threats, in the present dissertation we are addressing the topic of cyberthreat discovery in Open Source Intelligence (OSINT) using deep learning techniques. In terms of data sources, this means that we will be working with information gathered in web media outlets such as Twitter.

What differentiates our work is the way we approach the subject. Our standpoint is that to process such large amounts of data through deep learning architectures and algorithms represents a significant advantage in terms of efficiency and accuracy, which is why we will make use of neural networks. We adopt this approach given that deep learning mechanisms have recently gained much attention as they present an effective way to solve an increasing number of extremely complex tasks on very demanding *big data* problems.

To train our neural networks, we need a dataset that is representative and as large as possible. Once that is gathered we proceed by formulating adequate deep learning architectures and algorithmic solutions. Our ultimate goal is to automatically classify *tweets* as referring, or not, to cyberthreats in order to assess whether our hypothesis gets confirmed.

This dissertation is also meant to introduce general concepts and notions on the basis of which our work is deployed and to provide an overview of related work in such a way that this may be useful for future work. It also aims at providing an account of the work undertaken and of the obtained results, and last but not least to suggest what we see as promising paths for future work and improvements.

Keywords: artificial intelligence, natural language processing, neural networks, deep learning, cyberthreat detection

Contents

List of Figures

List of Tables

List of Abbreviations

CNN	Convolutional Neural Network
CPU	Central Processing Unit
DiSIEM	Diversity Enhancements for Security Information and Event Management
GPU	Graphics Processing Unit
ICT	Information Communication and Technology
LaSIGE	Large-Scale Informatics Systems Laboratory
NLP	Natural Language Processing
OSINT	Open Source Intelligence
SIEM	Security Information and Event Management
SOC	Security Operating Center
SVM	Support Vector Machine
TNR	True Negative Rate
TPR	True Positive Rate

Chapter 1

Introduction

Artificial intelligence is a notion that can somewhat be traced back to ancient Greece. Mythological tales involving figures like Talos, a giant automaton made out of bronze to protect Crete from pirates and invaders, reflects that already then there was an existing desire to create a machine with an intelligence of its own.

From myth to present date, we witnessed the creation of computers that would come to perform tasks far too intellectually taxing and complex for humans. And, although much has been accomplished, it is quite ironic that the true challenge presented to artificial intelligence should lie in solving tasks that are effortless to humans.

Tasks such as face recognition or the ability to distinguish between two similar spoken words are akin to breathing for a person. The abstract and informal nature of these tasks makes it so that people perform them naturally for humans rely on intuition to do so. The same cannot be said for computers for until recently it was with great difficulty that they would approach such mundane tasks. This adversity stems from the fact that computers seem much more capable when trying to solve problems of a more formal nature. This perceived dissonance regarding computer learning ability is the reason why, in recent years, there has been a growing interest concerning the surmounting of this shortcoming.

1.1 Motivation

Against this background, we should focus on why we intend to conduct our research.

Security Information and Event Management (SIEM) systems [5] are a fundamental component of the ubiquitous Information Communication and Technology (ICT) infrastructures that form the backbone of our digital society. These systems are mostly used to monitor infrastructures using many types of sensors and tools and articulate the events and observations thus obtained to discover possible threats to the organization.

Currently, a major limitation of SIEMs is the lack of capacity to present relevant OSINT with high degree of efficiency.

The project Diversity Enhancements for Security Information and Event Management (DiSIEM) [1] includes in its aims to address these kind of limitations concerning SIEMs already deployed in production. This is a Horizon 2020 project [2] in the scope of which the present research work was carried out and to which work plan it contributes.

Our contribution to the DiSIEM project was focused on addressing the increasing need to process large amounts of openly available data regarding security threats as efficiently as possible, in particular those that circulate in social media.

Our intention to answer this need compels us to research and look more closely at convolutional neural networks and their application to the discovery of security threats mentioned in *tweets*. We consider this approach because deep learning mechanisms have recently gained much attention as they present an efficient way to solve an increasing number of extremely complex tasks on very demanding *big data* problems.

1.2 Objectives

The main objective of our dissertation is to perform cyberthreat discovery over open source intelligence media available on the web.

We are interested in determining whether or not a *tweet* contains valuable information regarding a cyberthreat to a certain ICT infrastructure. This translates into a classification task where we either have an output corresponding to a threat or another one which indicates to us that there is no threat being mentioned.

In terms of data, this means that we work with information gathered in open source intelligence media outlets such as Twitter, giving emphasis to the accounts of users who focus on security and cyberthreat detection.

What differentiates our work from the current state of the art rests on the way we choose to approach the subject. Our standpoint is that there is a significant advantage in terms of efficiency and adequacy when processing large amounts of data related to ICT infrastructures through the means of cutting edge end-to-end natural language processing (NLP) based on deep learning techniques, instead of falling back on other already more common pipelined NLP techniques.

1.3 Contributions

Overall, we contribute to the detection of mentions to security threats in *tweets* by means of end-to-end, deep learning based natural language processing techniques that resort to neural networks.

Our system runs on small text portions (*tweets*) to check their relevance, instead of being applied to long sequences of text, which has been the common procedure in research conducted thus far.

We simplify the process of cyberthreat detection in the sense that, instead of being forced to identify prominent features beforehand and instructing our network about them, we rely on our neural network to autonomously extract relevant information and learn key features relevant to solve our problem.

Within deep learning based NLP, we make use in particular of convolutional neural networks applied to natural language processing, which follows previous experimentation but it is still a recent technique given that this type of neural networks are usually preferred as a method for image classification.

1.4 Planning

Our dissertation comprehended six crucial stages represented in Table 1.1.

Month	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Stage 1: State of the art review	🟦	🟦							
Stage 2: Data gathering and pre-processing		🟦	🟦	🟦					
Stage 3: Architecture and algorithm implementation				🟦	🟦	🟦			
Stage 4: Methodology employment					🟦	🟦	🟦		
Stage 5: Result analysis							🟦	🟦	
Stage 6: Dissertation writing								🟦	🟦

Table 1.1: Gantt diagram of our work stages.

Reviewing literature and other sources with reference to deep learning methodologies and programming libraries was our first priority before further pursuing our objectives. We allocated the first couple of months to conducting this activity in order to consolidate our approach to our problem.

While performing the previous task, we proceeded to gather and pre-process security related data sets. This was a three month endeavour concurrent to the preceding and subsequent stages.

Our next stage focused on the implementation of an adequate deep learning architecture and algorithmic approach. We devoted three months for this phase and worked on it in parallel to both stages 2 and 4.

At this time, having a clear idea of our model, we employed methodologies using available deep learning programming libraries. This stage had a duration of three months and as before it was simultaneous to its former and ensuing stages.

Having done so, we then proceeded to the extraction and comparative analysis of results. We reserved two months for this task.

Our last stage was assigned to writing the dissertation.

Unfortunately, some unforeseen events, such as faulty data, forced us to redo part of our work which then delayed the result analysis, causing a slight skid in our schedule.

1.5 Document Structure

In this chapter, **Chapter 1 - Introduction**, we contextualize our work and present its motivation, objectives and contributions to the current state of the art. We also indicate our planning and, lastly, give an overview of the structure of our dissertation.

Chapter 2 - Basic Notions and Related Work gives a brief overview of the general concepts and notions on which our dissertation is based. We provide definitions that help understand how the topics of artificial intelligence, machine learning, deep learning, and neural networks relate to one another.

This chapter also reports on relevant literature that was consulted. It covers related work on deep learning methodologies and programming libraries, as well as other topics such as natural language processing and the discovery of cyberthreats to an infrastructure in open source intelligence. Our purpose here is to expand on some of the interesting ideas we have encountered and explore how they help us in the pursuit of our main objectives.

Chapter 4 - Specifications and Design is dedicated to the specifications and design of our model, in which a clear picture of our neural network is provided, and specify what our neural network is required to do.

Chapter 5 - Implementation comprehends our implementation and describes the realization of the concepts and ideas earlier developed.

Chapter 6 - Experimental Setup documents our experimental setup alongside with selected performance metrics, and provides a detailed analysis. Here we discuss the potential relevance of our achieved results and ponder on its impact.

Our final chapter, **Chapter 7 - Conclusions and Future Work**, includes our conclusions and musings regarding future work. We finish our dissertation with a summary of the major achievements of our project and a compilation of its main results. We also direct future undertakers of our work towards solutions and optimizations that we think will lead to further progress.

Chapter 2

Basic Notions and Related Work

One of the purposes of this chapter is to convey basic notions concerning the topics being addressed in this dissertation. We find it important to distinguish between some concepts and to provide a little bit of background. As such, we highlight the notions of artificial intelligence, machine learning, neural networks and deep learning.

Another purpose is to briefly review literature on deep learning methodologies and programming libraries, as well as other topics relevant for our work, such as natural language processing. We expand on some of the interesting ideas we have encountered and conjecture how they may help in the pursuit of our main objectives.

2.1 Artificial Intelligence

Even if the notion of artificial intelligence may date back to millennia ago, it was in 1956 [6] at a conference in Dartmouth that the term was coined. There, a series of research projects were kicked off aiming to create an autonomous artificially intelligent machine.

This envisioned device would therefore need to perform mental tasks as well as or better than a human being. With this purpose in mind, researchers formulated topics of knowledge that a computer should learn.

They proposed that for an artificially agent to be considered intelligent, it should be able to perform reasoning tasks. These would be tasks where a person could reason its way to an answer when presented with a problem, such as playing chess.

Another important aspect would be the ability to represent reality in such a way as humans perceive it. A computer would have to understand the real world in order to understand and interact with people. This means grasping what objects, words and other such things are. This would come to be termed as knowledge representation.

The third thing needed to produce such a machine would be to make it understand and navigate the world that we live in. Planning and navigation activities such as going from one place to another one in a safe way comprises a series of tasks, like recognizing where doors are and what paths are, which had to be quintessential to this projected artificially intelligent machine.

Computers would also have to learn to understand language, to be able to create and attribute meaning to sentences, and to translate between different languages. This set of natural language processing activities would be essential to the replication of human alike intelligence.

Lastly, computers would have to learn how to perceive the world as people do. That is to say, computers would have to learn how people see, hear, feel and smell.

The expectation was that all the above-mentioned discrete parts of human intelligence would lead to an emerging generalized intelligence that was not explicitly programmed. All these pieces would come together to give the machine a capability for emotional intelligence, creativity, moral reasoning, and intuition very much indistinguishable from a human being.

In short and as we know it nowadays, we can say that artificial intelligence [7] is a set of algorithms and techniques to mimic human behavior.

2.2 Machine Learning

Machine learning is a kind of approach in artificial intelligence by means of which the system has the ability to automatically learn and improve with experience [8]. This means that we can do without explicitly programming and developing the entire system, as the program itself looks for patterns in data and learns to make better decisions based on provided examples. The goal of machine learning is to analyse the structure of data, to fundamentally create theoretical distributions around the data that are well understood to the system.

Even though machine learning is not a new technology, its resurgence is due to factors like the ever growing volume of data nowadays, as well as cheaper and more powerful methods for accessing and processing the data. This means that it is now possible to quickly and (semi) automatically produce a model that can analyse complex data with high accuracy [8] [9] [10]. As an example of having a precise and speedy model, it is viable to process large amounts of incoming data in near real time to have a chance of identifying an opportunity, or avoiding a risk.

While we cannot say that all artificial intelligence is machine learning, it is safe to state that machine learning is one of its key techniques. Over the years, this approach came to include decision tree learning, inductive logic programming. clustering, reinforcement learning, and Bayesian networks amongst several others.

All of the above listed techniques are usually applied to problems that artificial intelligence set out to solve. So, for example, we have search and optimization applied to natural language processing and syntax parsing which allows the computer to better reproduce the intelligent behaviour of a human being. Constraint satisfaction is another good example of how machine learning approaches the challenges introduced by artificial intelligence, for it decomposes the problem and allows the computer to work in micro worlds. A practical and more concrete application of these techniques can be observed in how early models of e-mail separated spam from the rest of the mail, just to mention another example among many others.

2.3 Supervised, Unsupervised and Semi-supervised Learning

In supervised learning [11], a training data set, composed of input/output pairings, is fed to the learner which may acquire the relationship between the input and the output. This approach is used for regression and classification problems. In a regression problem, the goal is to predict results within a continuous range. In other words, input variables are tentatively mapped through some continuous function that is learned. In turn, in a classification problem, the prediction of results translates itself into a discrete output, in an attempt to map input variables into discrete categories.

In unsupervised learning [12], the approach is undertaken with little to no idea of what the results should look like. Structure can be derived from training data where there is not an explicitly coded relation between input and expected output. This can be achieved by clustering the data on the basis of relationships amongst the elements in the raw input. However, there are other methods of setting up unsupervised algorithms for associative memory that also falls into what is considered to be unsupervised learning.

The key difference between both approaches is that with unsupervised learning there is no ground truth encoded in the training dataset that may support the prediction of results.

As for semi-supervised learning [13], also sometimes referred to as reinforcement learning, it is an in-between approach where some data is labeled (by means of which input and output items are paired) even if most might not be. The challenge that presents itself here is how to treat data that has been mixed in this way.

Some aspects of supervised learning are not present in unsupervised learning. This is not necessarily a bad thing since, as explained before, one has to adapt the learning approach to the problem itself.

2.4 Neural Networks and Deep Learning

Of all machine learning techniques, deep learning has recently acquired great prominence and several papers such as [14] and [15] have expand on their advantages. Notorious achievements of deep learning include image processing and winning a game of Go [16], among many others.

As a machine learning approach, deep learning techniques feed copious amounts of data into neural networks. A neural network is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information, according to the work of Stergiou and Siganos [9]. The seminal work in this areas dates at least as far back as 1943 [17].

A neural network is composed of a large number of highly interconnected processing elements (neurons) feeding each other to solve specific problems. It is constrained by the purpose it will fulfill, which is to say that it is configured for a specific application, such as pattern recognition or data classification.

Neural networks are organized in layers composed by a number of interconnected neurons containing activation functions. The term shallow neural network is used to describe a neural network usually comprised of only one or two hidden layers. As for deep neural networks they tend to have several hidden layers with distinct functional roles, and in general a deep network tends to have no less than three hidden layers.

While a shallow neural network can perform any function, that network is heavier and more intricate. The intricacy of such a function has an adverse consequence causing the number of parameters to increase significantly. As we have encountered in our readings [18], there are conclusive results that deep neural networks better accommodate functions with less parameters.

One paper in particular [19] defends that the correct measurement of network complexity is not necessarily the number of parameters. Its authors prove that even when dealing with a much smaller number of parameters that one would make use of in shallow neural networks, deep neural networks can accomplish much better results with compositional functions. This inability of shallow neural networks ties in with the curse of dimensionality, meaning that certain learning algorithms perform poorly or not at all in high-dimensional data.

It is thus worth mentioning that deep learning concerns itself with constructing machine learning models that learn a hierarchical representation of data. In other words, deep learning is a branch of machine learning that uses a set of algorithms in order to model high level abstractions in data.

2.5 Machine Learning for Cyberthreat Detection

The usage of machine learning to discover references to cyberthreats in OSINT is an area still somewhat unexplored. Ritter et al. [20] resort to a weekly supervised approach to classify *tweets* and thus detecting cyberthreats mentioned by them.

Also based on information from Twitter, Sabottke et al. [21] uses a Support Vector Machine (SVM) classifier that correlates a threat reported by a reliable security feed with posts on that threat in Twitter. The work from Rodrigues [22] is also based on data from Twitter. In this work, data stream from Twitter is directed to the HP ArcSight platform to take advantage of an additional source of information. Like in [21], this approach classifies *tweets* on the basis of frequency of occurrence of manually weighted keywords.

Veeramachaneni et al. [23], in turn, implemented a system that analyses logs by resorting to the outcome of machine learning algorithms that is blended with a manually designed set of instructions.

As previous work undertaken in our research group, it should be mentioned the Masters dissertation of Correia [24], also with the major goal of detecting *tweets* that refer to cyberthreats, like our goal here. To pursue that objective, a machine learning approach resorting to SVM is used. Accordingly, extensive manual and explicit tests were undertaken in order to find out suitable features on which to base the learning procedure.

Another work in our research group is undergoing as a Doctoral research. An unpublished paper from Alves et al. [25] reports on that work where *tweets* are classified as possibly referring to cyberthreats by resorting to SVMs and also to Multi-Layer Perceptrons.

In regard to our own research, we hypothesize that deep neural networks are a promising alternative approach to our problem that is worth exploring. Deep neural networks allow for a much lower number of training parameters than in shallow networks. This approach also exempts us from having to perform extensive feature engineering to approximate suitable features with which to feed learning procedures, as required when working, for instance, with SVMs.

In our working problem, the data consists of *tweets*, whose content is basically conveyed as text, thus calling for Natural Language Processing (NLP) techniques to be resorted to. Additionally, in recent cutting edge progress on NLP based on deep learning, Convolutional Neural Networks have shown to permit highly competitive results. Hence in the next sections, we proceed by focusing on these topics and visit their basic notions and related work.

To the best of our knowledge this approach of ours, consisting of combining deep learning techniques with classification of *tweets* for cyberthreat detection, has not been explored before.

2.6 Natural Language Processing for *Tweet* Classification

Natural Language Processing (NLP) is a field in the confluence of artificial intelligence and linguistics that involves intelligent analysis of language.

Our gathered data consists of *tweets*, which are natural language texts, even if they are short ones. After comprehensive literature perusing, we decided to explore an approach similar to the one proposed in [26], where these authors work with raw words rather than features that have to be previously engineered.

This paper reports on a series of experiments with neural networks trained on top of pre-trained word vectors for sentence-level classification tasks. This approach takes the semantic context of the words into account instead of just making use of a bag of words model.

Another relevant and interesting aspect to this paper is the use of Convolutional Neural Networks which are usually employed in image classification but have recently proven to work very well in sentence classification. To address this aspect, in the section that follows, we delve into this topic and extricate key notions and aspects inherent to this type of neural network.

2.7 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [27] are composed by several layers of convolutions with nonlinear activation functions applied to the end results.

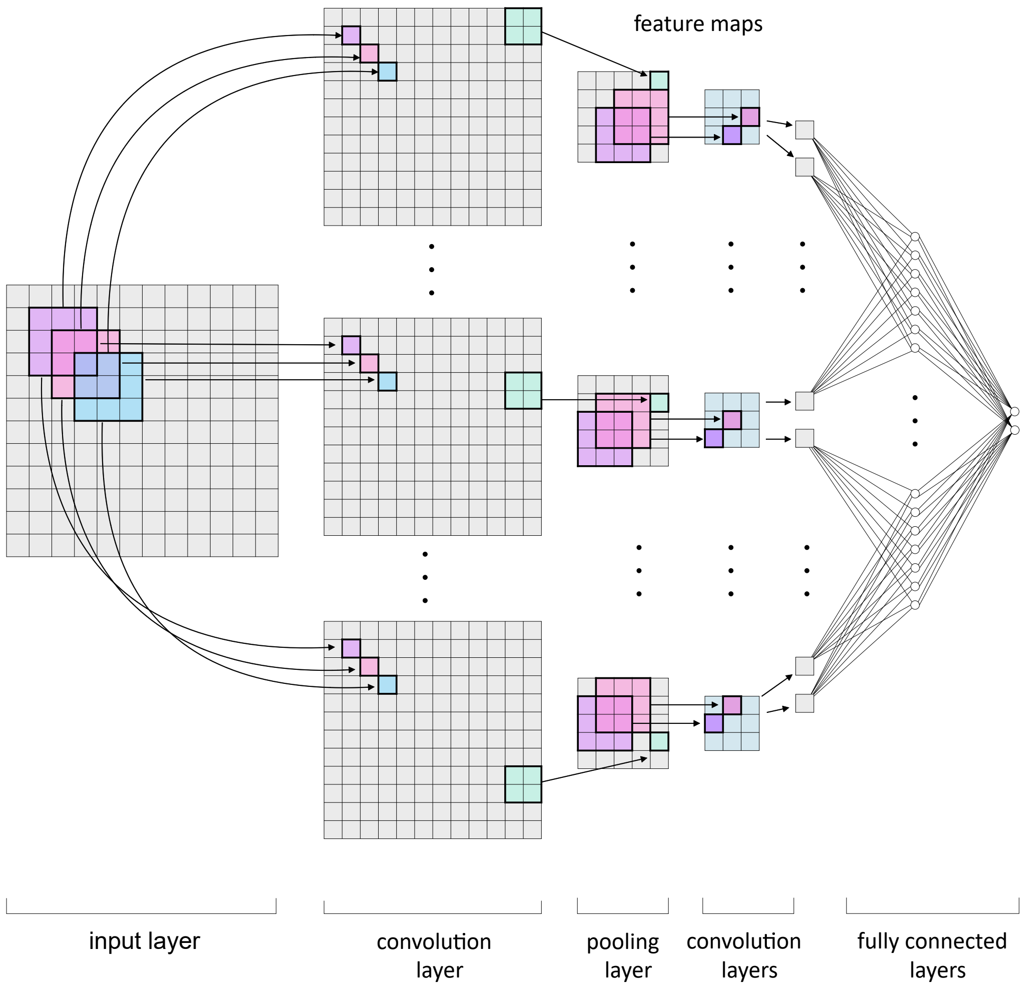


Figure 2.1: Example of a typical CNN.

In fully-connected feed forward neural networks, each input neuron is connected to each output neuron in the succeeding layer. CNNs do not follow this model of fully connected layers, opting instead to apply convolutions over the input layer to compute the output, where each region of the input is connected to a neuron in the output, resulting in a network with local connections.

Another relevant aspect of CNNs is that during the training phase, the neural network automatically learns the values of its filters based on the task it was assigned. The last layer of a CNN is then a classifier that uses these high-level features. In our case, this will determine whether or not a *tweet* contains information about a threat to certain ICT infrastructure component.

To better understand what a CNN is, we will now clarify a few concepts and explain some of the parts involved. We also include an image of what a typical CNN looks like in Figure 2.1.

A good reason to use CNNs is that they are very fast and provide an operation of convolution that helps to detect how the data are related to each other at the input of the network. Convolutions are essential in computer graphics and are typically implemented on a hardware level on graphics processing units [28].

2.7.1 Convolution

Given CNNs are commonly used in image classification, we resort to the classical example used to explain the convolution operation with the help of Figure 2.2.

This figure depicts a matrix of pixels (5x5, grey colored) in nine consecutive moments (from left to right and from top to bottom). It depicts also a sliding window (3x3, represented with different non grey colors) over that matrix at each one of those nine moments. With these nine views of the matrix of pixels, we can follow the operation of the convolution by observing the result of that sliding window, whose outcome is being collected in the convolved feature matrix (3x3), represented on the right column of Figure 2.2.

There are three views of this convolved feature (top to bottom), where each view corresponds to three of the nine moments that the larger matrix of pixels is going through. The correspondence between each one of these nine moments and its outcome recorded in that smaller matrix are represent by a specific non gray color. For instance, the sliding window at the fourth moment is depicted in yellow and its outcome is gathered in the yellow square of the convolved feature matrix.

In its different moments, the gray matrix in Figure 2.2 represents a convolutional layer.

A convolutional layer consists of a rectangular grid of neurons and each convolutional layer in a CNN requires that the preceding layer also be a rectangular grid of neurons. Each neuron then takes input from a rectangular section, covered by a sliding window, of the previous layer. For the neurons in a rectangular section, the weights

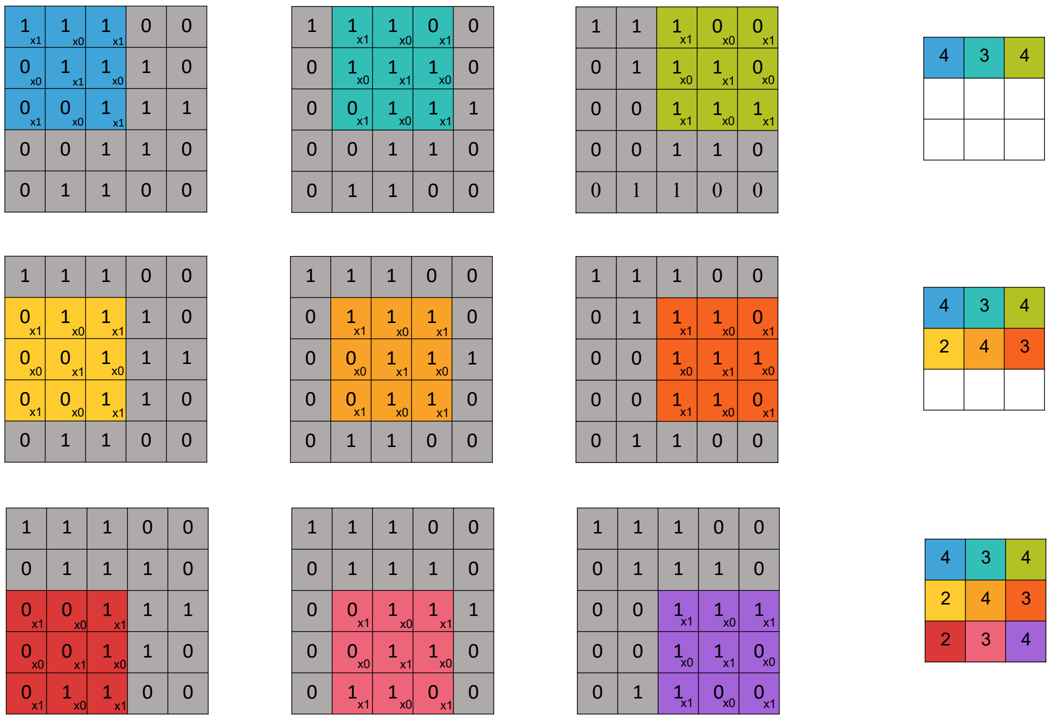


Figure 2.2: Example of a convolution in a sliding window function applied to a matrix of pixels.

are the same.

Hence, a convolutional layer gathers a compressed representation of its preceding layer. In that preceding layer, the weights of the neurons determine the so called convolution filter, which determines how the compression takes place.

Furthermore, in each convolutional layer, a CNN may have several sliding windows, or grids, such that each grid takes an input from all the grids in the previous layer, potentially using different filters.

In order to apply a CNN to natural language processing in an analogous way as it is applied to image processing, in the example of Figure 2.2, a sentence needs to get represented also as a matrix. In an initial stage, that matrix is made out of vectors, also known as word embeddings, such that each vector typically represents a word of the sentence.

The determination of a given vector for a given word is ultimately based on the frequency of co-occurrences of that word with other words within a certain window of context in a data set consisting of a collection of texts or sentences. To those frequencies, possibly sophisticated matrix composition and transformation techniques may then be applied.

These vectors may be learned on the fly as the neural network is being trained to handle a specific problem. Alternatively, the matrix that is formed by these vectors and is pretrained is suitable to be entered as the input for initial layers of CNNs.

2.7.2 Pooling

In CNNs, pooling layers subsample their input, most commonly applying a *max* or average operation to the result of each filter [29].

Pooling also provides a fixed output matrix size, which is generally required for classification. That is to say, pooling allows for variable sized sentences, and variable sized filters, ensuring the same output dimensions are fed to the final classifier layer.

Concerning output dimensionality, pooling helps reducing its size while preserving pertinent information. It is as if each filter detects a specific feature.

For example, pooling may play a role in detecting if a sentence contains a negation such as "not great". Provided that an expression like this may occur somewhere in the sentence, the result of applying the filter to that region will yield a large value, whereas it would return a small value in other regions.

When performing the *max* operation, information about whether or not the feature appeared in the sentence is being kept, while losing information about its location. This is of no concern to us because local information captured by filters is kept, thus ensuring that “not great” is very different from "great not".

2.7.3 Overfitting and Dropout

In the words of Hawkins [30], "overfitting is the use of models or procedures that violate parsimony, that is, that include more terms than are necessary or use more complicated approaches than are necessary".

In practical terms, overfitting occurs when the model or the algorithm fits the data too well. In other words, overfitting usually occurs when the model is too complex, meaning that there are too many parameters in relation to the number of trained or evaluated data.

Dropout is a common technique employed to prevent complex co-adaptations on training data which, consequently, helps to avoid overfitting the network [31]. A dropout layer works stochastically by rendering inactive a fraction of its neurons. This prevents neurons from co-adapting and forces them to learn useful features individually.

2.8 Convolutional Neural Networks and *Tweet* Classification

Having covered both CNNs and NLP topics, it is now time to explain how the first applies to the latter. The main reason for choosing CNNs is that they seem to be very apt for classifications tasks, such as the problem being addressed in this dissertation.

As we have previously stated, we are interested in determining whether or not a *tweet* contains valuable information regarding a cyberthreat to a certain ICT infrastructure component. This translates into a classification task for the network where we either have an output corresponding to a mention of a threat or another one where there is no mention of a threat.

In what concerns the architecture, we have encountered some papers [26][32][33] where the input layer receives a sentence comprised of concatenated word2vec [34] word embeddings, followed by a convolutional layer with multiple filters, a max-pooling layer, and finally a softmax classifier [35].

One particular downside to previous approaches to NLP based in deep learning is the resorting to pre-trained word vectors like word2vec [34] or GloVe [36]. However, one paper [37] presents a way of training a CNN from scratch, applying convolutions directly to one-hot vectors. A one-hot vector is used to distinguish each word from every other word in the vocabulary and consists of zeros in all cells with the exception of a single one in a cell used uniquely to identify the word.

It should also be noted that, in the aforementioned paper, and contrary to some of the existing literature [38], dropout was proved to have little beneficial effect on CNNs performance.

Figure 2.3 depicts the overall architecture for sentence classification whose model [37] we have decided to implement and adapt while experimenting with some of its hyperparameters. In particular, in the network we eventually used there are no fully connected layers before the *softmax* classifier.

In [39] the same author extends the previous model with an additional unsupervised "region embedding". This approach appears to produce excellent results for long texts, such as movie reviews, but their performance on short texts, like *tweets*, was not tested and thus was yet unclear. One of the contributions of our work is thus to extend that approach to short texts and show that it is also efficient in this case, and for our purpose.

Since one would think that pre-trained word embeddings applied to short texts would yield larger gains, we follow this approach in an attempt to achieve optimal results with our *tweet* classification.

The same authors have also found that max-pooling always beats average pooling, determining that filter sizes played a crucial part. However, since filter sizes are task-dependent, it means that a tailored adjustment is required for different problems.

Finally, regularization seemed to have little to no impact in the performed NLP tasks. Regularization is used to heavily penalize peaky weight vectors and preferring diffuse weight vectors. This has the effect of encouraging the network to use all of its inputs a little rather that some of its inputs a lot.

Another argument for the usage of CNNs in NLP is that, compared to something

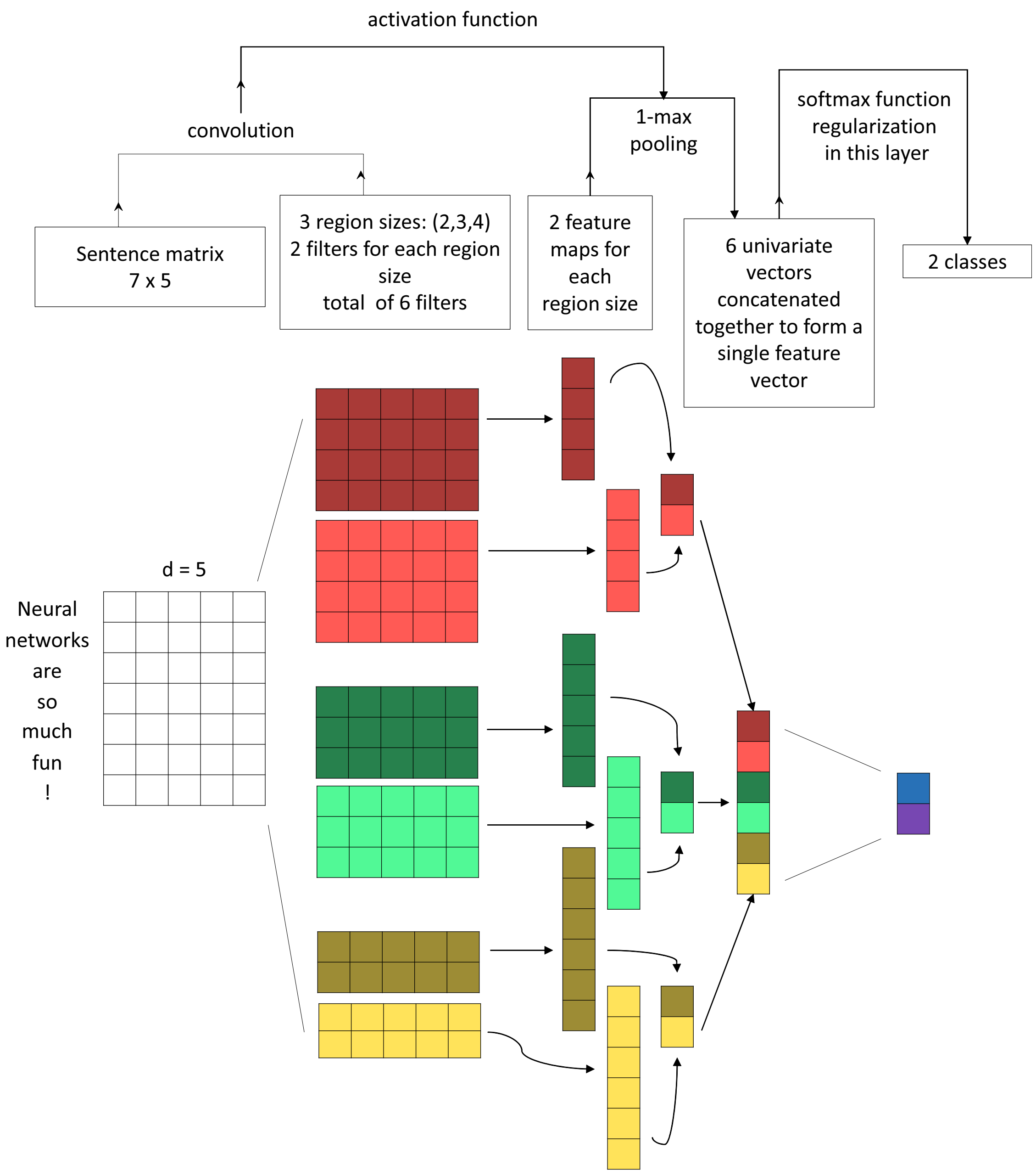

activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 x 5
3 region sizes: (2,3,4) 2 filters for each region size total of 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d = 5
Neural
networks
are
so
much
fun
!

Figure 2.3: CNN architecture for sentence classification.

like n-grams (contiguous sequences of n items from a given sequence of text or speech), CNNs are also efficient in terms of representation. When working with a broad vocabulary, computing more than 3-grams can quickly become expensive. Google itself does not provide anything beyond 5-grams.

Convolutional filters learn representations automatically, without the need to represent the entirety of the vocabulary. It is therefore reasonable to have filters of size larger than 5 when using CNNs.

Chapter 3

Specifications and Design

In the present chapter we specify what our neural network is required to do while providing a detailed description of the problem to be addressed and the necessary requirements for our envisioned approach.

3.1 Problem Statement

Our purpose is to develop a program and methodology that, after training with a dataset composed by textual input, can then accept other textual datasets of the same type as input and generate predictions concerning cyberthreats being referred to.

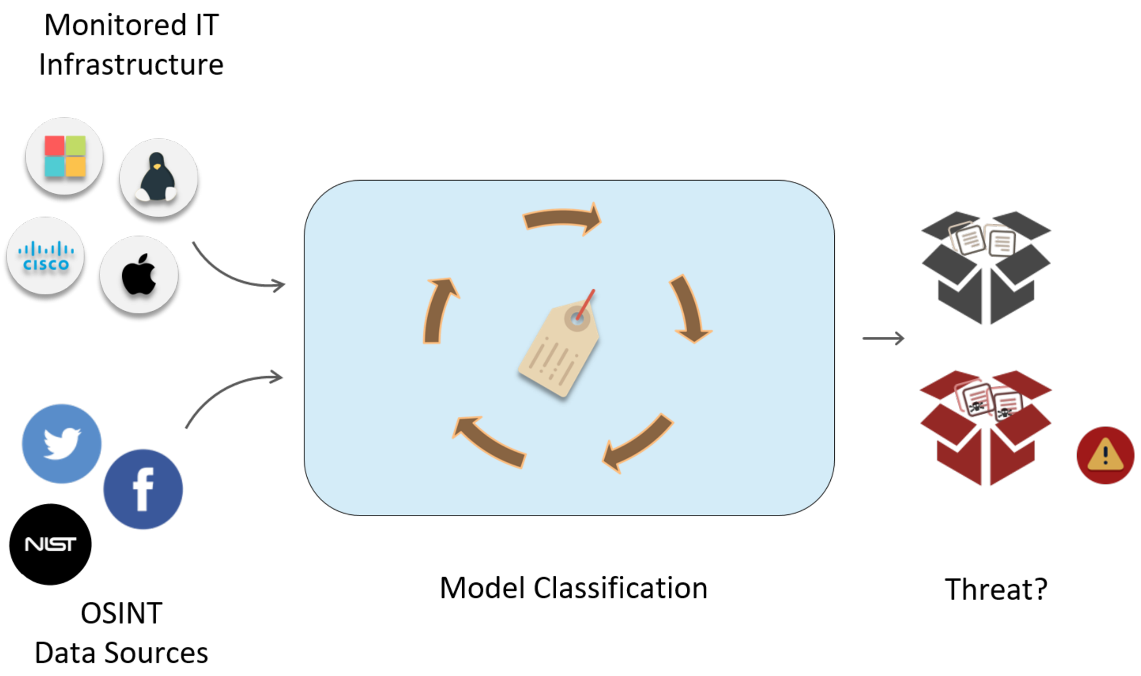


Figure 3.1: Diagram of the problem statement.

The generated output then serves as a means to determine whether each individual entry of the latter datasets contains information about security threats in relation to ICT infrastructure components specified by us or if these entries have instead no reference to threats to the surveyed systems and platforms.

In terms of real usage, our purpose is to devise a solution that ensures that security operation center analysts are delivered with relevant information about possible threats against the infrastructures for which they are responsible, thus seeking to select as much as possible *tweets* that actually refer to threats and not to select *tweets* that do not refer to these threats. This means we set out to aid in the task of sorting pertinent from irrelevant information regarding cyberthreat being mentioned in *tweets*, as depicted in Figure 3.1.

3.2 Requirements

To perform detection of references to cyberthreats, we chose to work with CNNs as they seem to be a promising method when applied to problems of this nature. The advantage of these particular neural networks will become apparent as we further discuss them and our employed methodology below and in forthcoming chapters.

A key advantage of our solution lies on the very little need for human guidance and supervision. In other words, our network has few requirements that have to be met by its user:

- In order for our network to function properly one must assure that its input is text. Two different datasets are required, a positive one, with *tweets* referring to cyberthreats, and a negative one, with *tweets* not referring to cyberthreats.
- Concomitantly, datasets must comprehend ICT infrastructure-related information. This is an imperative requirement for the positive dataset, while the negative dataset should include at least some information relative to systems or platforms we want to investigate.
- Lastly, every line of each dataset must correspond to a single entry, making it so that they are separated by a line break. These lines must always be preceded by the mention of the ICT infrastructure component which that *tweet* (in the line) reports to.

3.3 Data Collection

The data we feed our network with is of the utmost relevance as the accuracy of our results depend on them. Since the focus of our thesis is the implementation of a CNN, we rely on previously gathered data by other DiSIEM project colleagues, Correia [24] and Alves et al. [25]. Nonetheless, we provide a brief explanation on how the data was collected.

As explained by Alves et al. [25], in the collection stage a set of accounts is specified from which *tweets* will be gathered. These accounts are affiliated with security analysts and companies, hackers or researchers. The criteria to determine which accounts should be considered is based on the likelihood that these users *tweet* about the security of elements belonging to the information technology (IT) infrastructure being protected.

The collected dataset likely also includes *tweets* that are not relevant for the infrastructure the analyst wants to protect. As such, a filtering mechanism was devised which assumes that a *tweet* that acknowledges a threat to a certain IT infrastructure asset has to mention the asset properties.

3.4 Data Pre-Processing

Normalizing *tweet* representations is an initial and crucial step involved in data handling.

When pre-processing the collected data we first convert all characters to lower case and proceed to remove stop-words and hyperlinks alike. Characters, other than punctuation, not comprehended between [*a-z*] are also removed.

Numbers are also object of conversion and are written out in full (*e.g.*, “5" becomes “five"), while punctuation such as dots and hyphens are converted to their textual representation (“dot" and “hyphen"), as they may be relevant for software versions (*e.g.*, Google Chrome 4.5.1-2).

Having done so, it is now required that data be aggregated into two different datasets. Thanks to Correia [24], this step has been prepared for us and we have an easy way to access already labeled sets of positive and negative data.

To train and evaluate our neural network, our sample consists of several subdatasets. For every such subdataset, there is a set of positive *tweets* (with reference to security threats), and negative *tweets* (with no such reference) each containing tweets concerning only a specific infrastructure.

We then pre-process our entry data by prefixing every *tweet* with the respective infrastructure it concerns. Our positive data entries are therefore preceded by the name of the infrastructure component they mention, while our negative datasets entries allude to the same infrastructure regardless of not containing relevant information about it.

Having completed the previous stages, we proceed to pad each sentence to the maximum sentence length. For instance, if our longest *tweet* is 20 words long [40], all other *tweets* are going to have special `<PAD>` tokens appended to them until each tweets plus `<PAD>` tokens is of length 20. This allows us to efficiently batch our data since each example in a batch must be of the same length. We also pad our sentences at the beginning to match the maximum length of the name of the infrastructure relevant to us.

3.5 Frameworks

When considering frameworks for deep learning techniques there are some core aspects to be taken into account.

It is most desirable that the framework has a visible and participative community thus assuring good support if and when it becomes necessary. It is also crucial that the framework one works with has a stable library as well as helpful, updated and easily accessible documentation.

A framework should also have a good run-time performance. Since we are conducting research work, the framework we work with needs to have some flexibility that will allow us to experiment new things and develop new custom layers.

Before deciding ourselves for Tensorflow we investigated the benefits and downsides of a few promising and available frameworks which we now expand on.

3.5.1 Caffe

Caffe [41] has a very solid yet simple foundation. When using this framework one does not need to code, as one only needs to define the network with description files and train it. It also allows for Python integration and model training with it. This framework has one of the largest communities, which means there is good support and that doubts about emerging issues generally do not go unanswered.

In terms of documentation, Caffe is a bit out of date. This framework has a very complete library, but as we have found out, it can be a challenge to integrate new things. In other words, with Caffe one trades speed for stability.

Caffe's run-time performance is not the best but it is acceptable. It uses well-founded libraries operations such as convolution.

In terms of flexibility it has a good interface with Python and is compatible with new layers written in this language. However, it is not as transparent as other frameworks and one has to consult the source code to understand the underlying processes.

Caffe is a good library which hides the Graphics Processing Unit (GPU) and Central Processing Unit (CPU) integration from the developer. This framework has a very broad developer support and many branches that target different applications, causing it to be stable but somewhat difficult to deviate from the main branch.

Caffe has a vast set of pre-trained models for a variety of domains which could allow us to try different approaches without investing too much time in a definite solution.

It should also be noted that this framework was mainly developed to work with images which means that its library is somewhat lacking in solutions for text based problems like our own.

3.5.2 Theano

Theano [42] is a Python library that takes one's written code and translates it to C++. It has big supporting communities, including Google user groups and Github issue pages, although it is currently facing a transitioning period, as Google has adopted Tensorflow as its own framework.

Theano's documentation is both simple and informative and allows for fluid model development.

Due to Theano's simplicity to develop a new solution, it follows what is new which translates into a frail form of stability. This said, as far as one does not rely on the latest features, Theano is perfectly stable.

In terms of run-time performance, Theano presents a few issues regarding the compile time in which one has to wait before model execution.

In terms of flexibility, we know that with Theano it is really easy to develop something new. One only needs to take an already implemented layer or a function and modify to our purpose. It must however be noted that it is the user that has to write his own training code, even if the model is easily implementable.

The available implementations in Theano make it so that, regarding development, it is somewhat difficult to find the desired one, making the early stages a tad unstable.

From what we have gathered, there are scripts to convert Caffe pre-trained models. However, we have not encountered any insightful information on the reliability of these models or scripts.

3.5.3 Torch

Torch [43] is a Lua based library [44] and has the smallest community from all the four listed ones. Yet, from what we have gathered, the community albeit small is very responsive to any problem a user encounters.

Torch provides good documentation. However, for those too unfamiliar with Lua this documentation will not suffice and further digging on the behalf of the user is deemed necessary.

In terms of stability, Torch is solid even if with every minor change on some module it will require you to update others.

Run-time performance is the most powerful metric of Torch. It uses all the capability of any hardware one can use. One can switch between GPU and CPU through simple function calls and it is also very easy to use multiple-GPUs in data-parallel fashion. However, there is not any support for distributed training as of yet.

Torch is a very flexible framework which allows for any kind of deep learning architecture development with ease. Even very complicated convolutional neural networks or tangled natural language processing architectures are easily managed.

This appears to be a successful framework, in terms of development, which traces down what is new in the deep learning literature. New layers and functions recently proposed are always in the scope of Torch or at least any third part module.

Torch has a pretty good collection of pre-trained models. There is also an option to convert Caffe models to Torch with the aid of third party modules.

It should be noted that Torch is used extensively by Facebook and Twitter research teams for deep learning products and research.

3.5.4 Tensorflow

Tensorflow [45] is a library, written in Python and C++, that offers numerical computation with a flexible architecture which allows us to deploy computation to one or more CPUs using data flow graphs, something that is quite important for the high amount of computational power required by a CNN. Since TensorFlow was originally developed by the Google Brain Team with the purpose of supporting machine learning and deep neural networks research, it was a good fit to use in our application of a CNN.

This framework has one of the fastest growing communities, which is very responsive and helpful with issues that range from small code errors to architecture building doubts. However, because of its novelty, Tensorflow is still short of documentation, which can lead to a somewhat inconsistent library.

In terms of flexibility it has a good interface with Python, which was the chosen language for our work, and allows for extensive unit testing and self-verification. Therefore, to achieve a higher level of abstraction one has to use other libraries in conjunction with Tensorflow's (*e.g.*: NumPy [46]). Debugging can also prove to be an issue as error messages tend to be very cryptic.

A central aspect to TensorFlow is the notion of tensor. A tensor consists of a collection of primitive values structured into an array of any number of dimensions. It is a symbolic handle to an output of a TensorFlow operation. It does not hold the values of the operation's output, but it does provide a means of computing those values in a TensorFlow session.

A tensor can be passed as input to another operation, which builds a dataflow connection between operations. This connection enables TensorFlow to execute an entire graph that represents a large multi-step computation.

There is one final aspect that sets Tensorflow apart which is its model checkpoint feature. This feature allows the user to train a model for a while, stop and evaluate it, and then resume from that checkpoint to keep training the model.

3.5.5 Comparison of Frameworks

A summary of the principal features of the frameworks for neural processing just described is provided in Table 3.1.

	Commmunity support	**Documentation**	**Efficiency**	**Core**
Caffe	large	out of date	under performing	C++
Theano	large	good	compiling issues	Python
Torch	small	good	average	Lua
Tensorflow	supported by Google	poor	average	Python, C++

Table 3.1: Summary of principal features of neural networks frameworks.

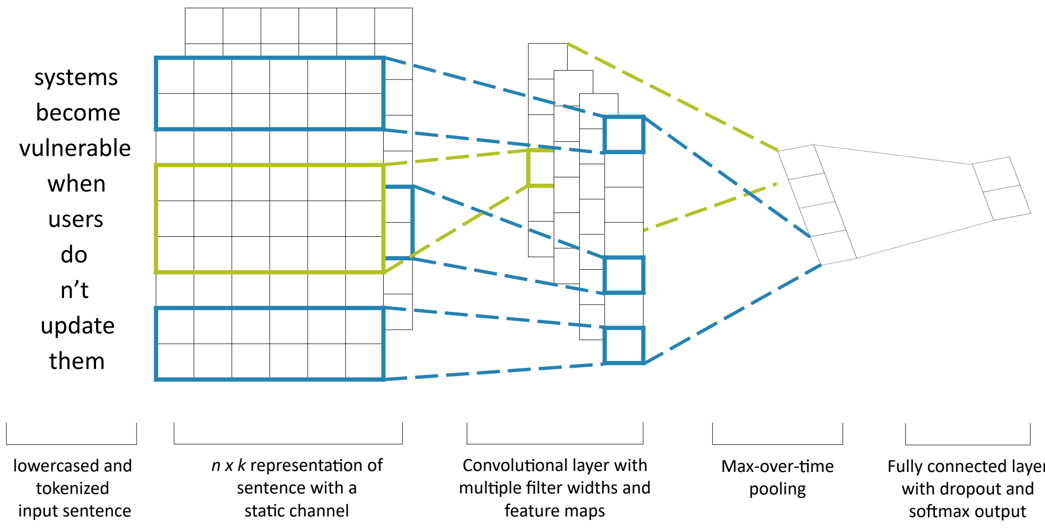
Chapter 4

Implementation

In this chapter, we describe the implementation of the concepts and ideas presented in previous chapters. We start by describing the model of our network, then proceed to the description of its parameters and conclude the chapter by presenting the pseudo code of the implemented solution.

4.1 Model

An overall sketch of the architecture of our model, based on [37] [47], is depicted in Figure 4.1.



systems
become
vulnerable
when
users
do
n't
update
them
lowercased and tokenized input sentence
n x k representation of sentence with a static channel
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 4.1: CNN architecture for sentence classification.

The first layer encodes words as vectors, also known as word embeddings.

After this first layer, there is a layer that uses multiple filter sizes to perform convolutions on the embedded word vectors.

The output of this convolutional layer is then normalized by max-pooling them into a long feature vector.

Finally, a dropout regularization is added to avoid overfitting the network. This network is then fully connected to a softmax layer which is used to classify its output as positive or negative. A positive instance case signals that the input sentence mentions a cyberthreat, while a negative one signals the absence of such a cyberthreat mention.

4.2 Data Loading Parameters

After having extensively covered in the previous chapter how data is collected and preprocessed, we now list the loading parameters of our network in Table 4.1.

Arguments	**Description**
`test_sample_percentage`	Percentage of the training data to use for validation.
`positive_data_file`	Data source for the positive data.
`negative_data_file`	Data source for the negative data.

Table 4.1: Data loading parameters.

4.2.1 Validation Sample

When feeding our network with the training data, we reserve a portion of our data to be left out and to be used to evaluate the precision with which the model was trained. Our default during tests was to always use 10% of the training dataset for this validation.

4.3 Model Hyperparameters

When implementing our network, it is important to setup a few hyperparameters given that they cannot be learned directly from the data in the standard training process and that they affect the topology of the model.

Table 4.2 lists all the model hyperparameters that are open to being setup and need to be specified. In the subsections below these parameters are described.

In the table, we also show the default values of the parameters. These values were taken from the previous works [26] [37], whose architecture we implemented and adapted, and were determined also in a task of sentence classification.

Arguments	Description
`embedding_dim`	Dimensionality of word embedding (default: 128).
`filter_sizes`	Comma-separated filter sizes (default: '3, 4, 5').
`num_filters`	Number of filters per filter size (default: 128).
`dropout_keep_prob`	Dropout keep probability (default: 0.5).
`l2_reg_lambda`	L2 regularization lambda (default: 0.0).

Table 4.2: Model hyperparameters.

4.3.1 Embedding Dimensionality

The embedding layer is the first layer of our network. Its function is to map word indexes into respective vector representations. As our network learns from the data, this becomes a lookup table for the vectorial representations of the words in our input.

By articulating these word indexes with the embedding matrix, the result is a dense matrix that will be fed to the convolution layer.

4.3.2 Filters

Given that each convolution produces tensors of different dimensions, we need to iterate through them, create a layer for each of them, and then merge the results into one big feature vector.

The number of filters per filter size we used in our approach is 128. Since we have three different filter sizes (3, 4, 5), this results in 384 filters. Each of these filters slides over the whole embedding, covering an amount of words according to its sliding window.

Performing *max*-pooling over the output of a specific filter size leaves us with a feature vector with size 128 for each filter. Once we have all the pooled output tensors from each filter size we combine them into one long feature vector of shape `[batch_size`, 384], as it corresponds to the total number of filters.

4.3.3 Dropout Probability

The fraction of neurons we keep enabled is defined by the `dropoutKeepProb` input to our network. We set this to 0.5 during training, and to 1 (disable dropout) during evaluation.

Since the number of neurons is given by `num_filters` × `filter_sizes`, this means that only about 192 neurons remain active after dropout.

4.3.4 Regularization

We also considered the application of $L2$ norm constraints on the weight vectors. However, as it was explained in [37], the authors' end results were not heavily impacted by these constraints and that is the reason why we chose not to enforce them. We also tested them with inconclusive results.

As dropout is an implicit form of regularization, maybe the fact of using it induces that explicit regularization produces little to no impact.

4.4 Training Parameters

The algorithm used for training was the Adam Tensorflow's optimizer [48], which employs a Stochastic Gradient Descent procedure.

Table 4.3 details the training parameters of our CNN.

Arguments	**Description**
`batch_size`	Batch Size (default: 64).
`num_epochs`	Number of training epochs (default: 200).
`evaluate_every`	Evaluate model on validation set after these steps (default: 100).
`checkpoint_every`	Save model after this many steps (default: 100).
`num_checkpoints`	Number of checkpoints to store (default: 5).

Table 4.3: Training parameters.

Batch Size

Our batch size corresponds to the number of training examples in one forward/backward pass in our CNN. It should be noted that the higher the batch size, the more memory space we need to train the network.

Epochs

We consider each epoch to be equivalent to each forward pass and one backward pass of all the training examples.

Evaluation

This parameter sets after how many steps our CNN should evaluate the model with the sample data set aside (`test_sample_percentage`) at the beginning.

Checkpoints

Checkpoints allow us to inspect or confirm if our CNN is progressing in the right direction without having to train the network until the very end.

4.5 Pseudo-Code

In order to provide an overall indication of how we implemented our model that meets the previously established requirements, we resort to the pseudo-code in Algorithm 1.

Algorithm 1 CNN Step

```
1: procedure TRAIN STEP (vocabSize, embedSize, filterSizes, numFilters)
2:     Let input_x be the input array with the tweets
3:     Let input_y be the array with the classification for each tweet
4:     Let W be the embedding matrix based on vocabSize, embedSize
5:     embedded_input = LOOKUP_EMBEDDING (input_x, W)
6:     pooled_outputs = [ ]
7:     for all filterSize do in filter_sizes
8:         filter_shape = [filterSize, embedSize, 1, numFilters]
9:         conv = CONVOLUTE (embedded_input, filter_shape)
10:        pool = MAX_POOL (conv, bias)
11:        pooled_outputs += pool
12:    end for
13:    h_pool = CONCATENATE (pooled_outputs)
14:    h_dropped = DROP_NEURONS (h_pool, drop_percentage)
15:    losses = SOFTMAX_CROSS_ENTROPY (h_dropped, input_y)
16:    predictions = MAX_SCORE (h_dropped)
17:    correct_predictions = REDUCE_MEAN (predictions)
18: end procedure
```

This pseudo-code resorts to functions of the Tensorflow framework which we used to implement our CNN, whose libraries [45] should be consulted for more details on the fully fledged specification of these functions.

The goal here is to convey a sense of how all the parts in our CNN come together to process our input data that are latter on classified by our softmax layer.

To achieve the training of the neural network, the training step represented in this pseudo-code is iterated until the number of steps given by the formula (not in the pseudo-code) is reached:

$$\textit{number of steps} = \frac{\textit{input size}}{\textit{batch size}} \times \textit{number of epochs} \tag{4.1}$$

where the input size is the length of the training data set, with the collection of (positively and negatively classified) tweets; the batch size (defined beforehand and instantiating `batch_size`) is the number of training examples in each forward/backward pass; and the number of epochs (defined beforehand and instantiating `num_epochs`) is the number of forward/backward passes of all training examples.

This training of the network is embedded in Tensorflow by means of which the network's loss function is optimized. Tensorflow has several built-in optimizers. We use the Adam optimizer [48].

Let us now focus on an individual training step, represented in the pseudo-code.

Each such training step is based on an iterative loop, which is preceded by a number of operations instantiating the necessary data structures.

The two arrays *input_x* and *input_y* have the same length, one containing the tweets of the training data and the other the respective classifications (either as positive or negative examples).

In turn, W is the matrix with the pre-trained word embeddings with the number of lines given by the size of the vocabulary (recorded in *vocabSize*) and the number of columns given by the length of the embedding vectors (recorded in *embedSize*).

The array with the training data, *input_x*, and the matrix with the pre-trained word embeddings, W, are used to lookup the embeddings for our input. They are passed as arguments of the Tensorflow's function LOOKUP_EMBBEDDING, whose results is stored in *embedded_input*. The result of this embedding operation is a 3-dimensional tensor containing the vectorial representation of the input.

Finally, before entering the core loop, the data structure *pooled_outputs* that is going to accumulate the partial outputs of the iteration is also initialized with an empty instantiation.

The iterative loop is responsible for ensuring the convolution. There will be three iterations of its body, each running 128 filters, such that the size of these filters are three, four and five, respectively, in the first, second and third iteration, as explained in detail above in section 4.3.2.

The Tensorflow's CONVOLUTE function is in charge of running the 128 filters (whose number is defined beforehand and recorded in its argument *numFilters*), for each of their three different sizes of the filters.

The output of this CONVOLUTE function, together with the default value (0.1, the weight for the convolution) for the *bias* argument, which aims to mitigate a biasing effect during the pooling, feed Tensorflow's MAX_POOL function. By max-pooling the output of the convolution operation, the dimensionality of the tensor of features is reduced to the most relevant ones. This helps to avoid over-fitting and reduces the computational cost by reducing the number of features the network has to learn.

At each iteration, the array resulting from max-pooling gets concatenated to the previous value of *pooled_outputs*, which eventually retains the concatenation of every such array.

Right after the exiting of the loop, the array resulting from that concatenation is transformed into a Tensorflow's tensor by means of the CONCATENATE function.

After having thus concatenated the pooled outputs of the convolution, a fraction of the network neurons is stochastically disabled, preventing them from co-adapting. The dropout ratio, encoded in *drop_percentage*, is set up as 50%.

Using this final feature vector, we calculate the loss, or the measurement of the error the network makes, by using the SOFTMAX_CROSS_ENTROPY function.

With that final vector, we also generate the predictions as the outcome of the composition of the function MAX_SCORE with the function REDUCE_MEAN. From the features identified, MAX_SCORE returns the weight of the one with the heaviest weight. REDUCE_MEAN returns the difference from the prediction to the ground truth.

This concludes the presentation of the pseudo-code, which represents a training step. As mentioned above, this training step is iterated to accomplish the training of the model.

As for the evaluation phase, each of its steps works pretty much in the same way as the training step just explained. The two key differences from an evaluation step to a training step are twofold. In the evaluation step, the Adam optimizer and the dropout do not exist.

Chapter 5

Experimental Setup

In this chapter we report on our experimental setup alongside with performance metrics that were adopted. A detailed analysis of results is also presented.

5.1 Infrastructure definition

During the course of our dissertation we have many times referred to ICT infrastructures. Table 5.1 groups all the infrastructures considered in our research work into four different clusters as defined in [24].

Group A is a simple representation of Cisco and Oracle products. Group B encompasses browsers. Group C relates to content management systems. And group D considers the operating systems. The last group (ABCD) refers to the case where one single classifier will be fed by *tweets* related to any of the four infrastructure groups.

Infrastructure	**Description**
A	oracle, cisco
B	google chrome, microsoft edge, mozilla firefox, internet explorer
C	wordpress, joomla
D	microsoft windows, linux
ABCD	A + B + C + D

Table 5.1: Infrastructure grouping.

5.2 Datasets

For the purpose of our experiments, we initially considered three datasets that have already been collected and properly labeled [24]. Their quantitative characterization is presented in Table 5.2.

Dataset	Infrastructure	Positive	Negative	Total
D1	A	556	514	1070
	B	217	497	714
	C	486	606	1092
	D	441	691	1132
	ABCD	1700	2308	4008
D2	A	177	249	426
	B	86	446	532
	C	138	900	1038
	D	138	2697	2835
	ABCD	539	4292	4831
D3	A	502	256	758
	B	420	362	782
	C	425	303	728
	D	336	1232	1568
	ABCD	1683	2153	3836

Table 5.2: Default datasets D1, D2, and D3.

The table also includes information about the collected *tweets*, making a distinction between positive, referring to *tweets* that mention a threat to a given part of the ICT infrastructure, and negative instances.

We highlight group ABCD because, despited having experimented with its individual components, we are mostly interested in results that work with all groups together.

The distinction between D1, D2 and D3 is explained on Section 5.2.2.

5.2.1 Twitter Accounts

Tables 5.3 and 5.4 list the Twitter accounts from which the *tweets* were extracted [24]. As indicated before, these accounts are affiliated with security analysts, companies, hackers, and researchers.

Account set S1			
inj3ct0r	slashdot	ThreatFeed	USCERT_gov
TrustedSec	dstrom	pikisec	gcluley
Anomali	Info_Sec_Buzz	SANSInstitute	hal_pomeran
briankrebs	vuln_lab	johullrich	SecurityWeek
Secunia	threatintel	drericcole	SecurityNewsbot
exploitdb	dangoodin001	F1r3h4nd	sans_isc
alienvault	ivspiridonov	MaldicoreAlerts	e_kaspersky

Table 5.3: Set of accounts S1 from which *tweets* were collected.

Account set S2			
TenableSecurity	JoomlaTips	Microsoft	fstenv
securitywatch	sjzaib	linuxfoundation	HPE_Security
securityaffairs	SecurityMagnate	ChidoDike	googlechrome
zer0element	Cisco	Sec_Cyber	wordpressdotcom
notsosecure	Dell	ptracesecurity	packet_storm
CyberExaminer	linuxtoday	msftsecurity	RokaSecurity
SCMagazine	securityninja	LinuxSec	Oracle
DMBisson	cyberopsy	hack3rsca	firefox
lennyzeltser	OWASP_Java	CiscoSecurity	wpbeginner
IT_securitynews	_WPScan_	NytroRST	YoKoAcc
teamcymru	d_plusk	joomla	SecurityCrap
WordPress	threatpost	Windows	jasonlam_sec
MicrosoftEdge	Rootsector	crackerhacker00	threatmeter

Table 5.4: Set of accounts S2 from which *tweets* were collected.

5.2.2 Time Frame

Our three datasets D1, D2 and D3 were collected during three different periods of time as shown in Table 5.5, where the collection time frame and the corresponding account sets may be consulted.

Dataset	Time period			Account set
D1	01-11-2015	to	01-04-2016	S1
D2	01-04-2016	to	15-05-2016	S1, S2
D3	15-05-2016	to	10-07-2016	S1, S2

Table 5.5: Time period of *tweet* collection.

5.2.3 Data balancing

Having trained and evaluated our CNN with these datasets, a number of observations emerged. First and foremost, we found the datasets to be somewhat small to properly assess our problem. We also felt that the proportion of positive and negative data is unbalanced enough to bias our network towards negative classification.

The first issue was straightforward to address as more negative data can be obtained from the already existing pool of data. We did so by considering that a positive entry regarding, for example, Cisco must be negative for Oracle and *vice versa*. Table 5.6 shows the resulting datasets from this expansion of the already existing negative entries. However, this correction of the first problem comes at the expense of the aggravation of the second one.

5.3 Metrics

In order to determine whether our neural network is producing good results, we adopt metrics to measure its performance. To do so, it is important to clarify the concepts of sensitivity and specificity.

5.3.1 Sensitivity

Sensitivity, also referred to as True Positive Rate (TPR), is a statistical measure that assesses the proportion of positives correctly identified as such:

$$TPR \quad = \quad \frac{true \;\; positives}{true \;\; positives \; + \; false \;\; negatives} \tag{5.1}$$

In order to determine sensitivity we first create a tensor named `ones_tensor` that is the same size as the input tensor of the batch being analysed. This tensor consists only of ones which represents the positive result/*tweet*.

`input_y` is the tensor that contains the correct results for that particular batch. This tensor also has the same size as the batch but, unlike the previous two, it is composed of both ones and zeros that match positive and negative *tweets*, respectively.

Dataset	**Infrastructure**	**Positive**	**Negative**	**Generated negatives**	**Total**
D1	A	556	514	556	1626
	B	217	497	651	1365
	C	486	606	486	1578
	D	441	691	441	1573
	ABCD	1700	2308	2134	6142
D2	A	177	249	177	603
	B	86	446	258	790
	C	138	900	138	1176
	D	138	2697	138	2973
	ABCD	539	4292	711	5542
D3	A	502	256	502	1260
	B	420	362	1260	2042
	C	425	303	425	1153
	D	336	1232	336	1904
	ABCD	1683	2153	2523	6359

Table 5.6: Complementary datasets D1, D2, and D3.

Additionally, we have a `predictions` tensor which is the output result of Tensorflow's analysis for each batch. This tensor will contain the result of the analysis, meaning that it will be composed of both ones and zeros corresponding to the positive and negative nature of the result/ *tweet*.

The `predictions` tensor will be compared to the input tensor in order for the Tensorflow network to learn and for us to analyse the performance of the network.

Having gathered these tensors, we then perform a logical `AND` operation which will return a tensor containing all true positives:

$$true\ \ positives \quad = \sum_{n=0}^{input\ \ length} input\_y_n \ \wedge \ predictions_n \tag{5.2}$$

Finally, TPR is calculated as the ratio of true positives over the ones counted on the `input_y` tensor:

$$TPR = \frac{true\ \ positives}{count\_ones(input\_y)} \tag{5.3}$$

5.3.2 Specificity

Specificity, also called the True Negative Rate (TNR) measures the proportion of negatives that are correctly identified as such:

$$TNR = \frac{true\ \ negatives}{true\ \ negatives\ +\ false\ \ positives} \tag{5.4}$$

We apply the dual procedure employed for determining sensitivity (TPR) as we are instead interested on the negative input and predictions (TNR), for which we turn zeros into ones, and *vice versa*, with ¬:

$$true\ \ negatives = \sum_{n=0}^{input\ \ length} \neg\, input\_y_n \ \wedge\ \neg\, predictions_n \tag{5.5}$$

$$TNR = \frac{true\ \ negatives}{count\_zeros(input\_y)} \tag{5.6}$$

5.4 Cross Validation

In order to assess how our results are capable of generalizing to an independent data set, we trained and evaluated our network rotating the percentage of a testing data subset, following a ten-fold cross validation.

After shuffling the data, we parcel it into ten different subsets. The performed evaluation then iterates over all testing subsets in order for us to be able to determine the average and standard deviation of the evaluation scores. For each fold, 90% of the remaining dataset is used to train the model and the remaining 10%, that is the fold at stake, for testing its performance. The scores presented in the fold lines of D1 correspond to those obtained on that 10% testing fold.

Chapter 6

Results and Discussion

After having established which infrastructures were tested, provided an overview of our datasets, and explained the employed metrics and evaluation methodology, we are now ready to present the obtained results.

6.1 Results

The scores are first presented, to which their discussion follows, by the end of the present chapter.

Cross Validation

Tables 6.1 to 6.6 show the TPR and TNR scores obtained as a result of the cross-validation procedure, concerning datasets D1 (Tables 6.1 and 6.2), D2 (Tables 6.3 and 6.4) and D3 (Tables 6.5 and 6.6).

For each dataset D1 to D3, the tables show the result for both the default version (Tables 6.1, 6.3, and 6.5) and the complementary version (Tables 6.2, 6.4, and 6.6), which includes artificially generated negative data.

In the tables, the lines numbered by fold indicate the scores obtained for each one of the ten folds, and the final two lines indicate their average and standard deviation.

The scores in the numbered lines for D2 and D3 (Tables 6.3 to 6.6) were obtained by running the respective model trained with D1 for the same line number and batch size, and for the same dataset version, either default or complementary.

D1 - default										
Fold	Batch size 16		Batch size 32		Batch size 64		Batch size 128		Batch size 256	
	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR
1	0.97	0.98	0.97	0.97	0.97	0.97	0.99	0.97	0.98	0.97
2	0.96	0.95	0.97	0.94	0.96	0.96	0.98	0.95	0.98	0.96
3	0.94	0.98	0.93	0.99	0.95	0.99	0.95	0.98	0.94	0.98
4	0.97	0.96	0.96	0.97	0.95	0.97	0.98	0.95	0.97	0.96
5	0.92	0.96	0.94	0.96	0.99	0.94	0.96	0.97	0.98	0.97
6	0.90	0.96	0.95	0.96	0.94	0.95	0.95	0.95	0.93	0.95
7	0.94	0.95	0.94	0.95	0.95	0.96	0.95	0.96	0.93	0.95
8	0.88	0.96	0.90	0.97	0.94	0.96	0.90	0.97	0.92	0.97
9	0.93	0.96	0.92	0.98	0.94	0.97	0.92	0.97	0.94	0.97
10	0.94	0.97	0.94	0.97	0.97	0.97	0.98	0.96	0.98	0.98
Average	0.93	0.96	0.94	0.96	0.96	0.96	0.96	0.96	0.96	0.96
Standard deviation	0.03	0.01	0.02	0.01	0.01	0.01	0.03	0.01	0.02	0.01

Table 6.1: Results obtained in the testing subsets of the cross-validation procedure, for the default version of the dataset D1.

D1 - complementary										
Fold	Batch size 16		Batch size 32		Batch size 64		Batch size 128		Batch size 256	
	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR
1	0.94	0.94	0.95	0.95	0.95	0.96	0.94	0.95	0.93	0.96
2	0.96	0.95	0.94	0.97	0.94	0.97	0.96	0.98	0.95	0.97
3	0.97	0.95	0.93	0.98	0.95	0.96	0.96	0.96	0.95	0.96
4	0.97	0.96	0.94	0.97	0.97	0.96	0.96	0.97	0.95	0.97
5	0.92	0.97	0.95	0.97	0.92	0.97	0.94	0.98	0.95	0.97
6	0.91	0.95	0.94	0.97	0.94	0.96	0.93	0.97	0.93	0.95
7	0.96	0.95	0.96	0.95	0.93	0.96	0.97	0.95	0.97	0.95
8	0.92	0.95	0.92	0.94	0.90	0.96	0.93	0.95	0.93	0.96
9	0.93	0.95	0.94	0.96	0.90	0.96	0.90	0.95	0.90	0.96
10	0.95	0.95	0.96	0.95	0.96	0.95	0.94	0.97	0.97	0.96
Average	0.94	0.96	0.94	0.96	0.94	0.96	0.94	0.96	0.94	0.96
Standard deviation	0.02	0.01	0.01	0.01	0.02	0.01	0.02	0.01	0.02	0.01

Table 6.2: Results obtained in the testing subsets of the cross-validation procedure, for the complementary version of the dataset D1.

D2 - default										
Fold	Batch size 16		Batch size 32		Batch size 64		Batch size 128		Batch size 256	
	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR
1	0.70	0.98	0.71	0.98	0.75	0.98	0.90	0.98	0.81	0.98
2	0.60	0.98	0.78	0.98	0.78	0.98	0.81	0.98	0.79	0.99
3	0.70	0.98	0.64	0.98	0.79	0.98	0.85	0.98	0.87	0.99
4	0.71	0.98	0.75	0.98	0.70	0.98	0.86	0.98	0.82	0.98
5	0.66	0.98	0.66	0.98	0.83	0.98	0.81	0.98	0.86	0.98
6	0.62	0.99	0.68	0.98	0.73	0.98	0.80	0.98	0.82	0.98
7	0.77	0.98	0.69	0.98	0.72	0.98	0.82	0.98	0.80	0.98
8	0.58	0.98	0.72	0.99	0.79	0.97	0.78	0.99	0.83	0.98
9	0.73	0.98	0.67	0.98	0.59	0.98	0.72	0.99	0.75	0.99
10	0.56	0.98	0.68	0.98	0.79	0.98	0.83	0.98	0.82	0.98
Average	0.66	0.98	0.70	0.98	0.75	0.98	0.82	0.98	0.82	0.98
Standard deviation	0.07	0.00	0.04	0.00	0.07	0.00	0.05	0.00	0.03	0.00

Table 6.3: Results obtained by applying each fold's model trained with default D1 to the default version of the D2 dataset.

D2 - complementary										
Fold	Batch size 16		Batch size 32		Batch size 64		Batch size 128		Batch size 256	
	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR
1	0.79	0.98	0.82	0.98	0.78	0.98	0.75	0.98	0.75	0.98
2	0.61	0.98	0.73	0.98	0.76	0.98	0.77	0.98	0.77	0.98
3	0.68	0.97	0.68	0.98	0.71	0.97	0.78	0.98	0.82	0.98
4	0.78	0.97	0.66	0.98	0.84	0.97	0.82	0.97	0.63	0.98
5	0.71	0.98	0.76	0.97	0.73	0.98	0.73	0.98	0.80	0.98
6	0.78	0.97	0.68	0.98	0.83	0.97	0.71	0.98	0.82	0.98
7	0.73	0.98	0.79	0.98	0.73	0.98	0.72	0.98	0.71	0.98
8	0.76	0.98	0.82	0.96	0.71	0.97	0.80	0.98	0.74	0.98
9	0.87	0.97	0.83	0.97	0.80	0.98	0.74	0.98	0.78	0.98
10	0.71	0.97	0.78	0.98	0.82	0.97	0.81	0.98	0.79	0.98
Average	0.74	0.98	0.75	0.98	0.77	0.98	0.76	0.98	0.76	0.98
Standard deviation	0.07	0.00	0.06	0.01	0.05	0.00	0.04	0.00	0.06	0.00

Table 6.4: Results obtained by applying each fold's model trained with complementary D1 to the complementary version of the D2 dataset.

D3 - default										
Fold	Batch size 16		Batch size 32		Batch size 64		Batch size 128		Batch size 256	
	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR
1	0.88	0.83	0.89	0.85	0.90	0.88	0.96	0.84	0.93	0.90
2	0.84	0.89	0.91	0.83	0.91	0.87	0.94	0.86	0.91	0.90
3	0.88	0.89	0.87	0.91	0.91	0.90	0.93	0.90	0.94	0.92
4	0.88	0.83	0.90	0.87	0.89	0.86	0.94	0.82	0.94	0.89
5	0.86	0.89	0.87	0.91	0.93	0.87	0.94	0.90	0.94	0.87
6	0.85	0.90	0.86	0.84	0.89	0.81	0.92	0.89	0.93	0.90
7	0.89	0.89	0.89	0.90	0.89	0.89	0.92	0.89	0.93	0.85
8	0.83	0.89	0.89	0.89	0.92	0.83	0.91	0.91	0.93	0.89
9	0.90	0.88	0.87	0.89	0.84	0.89	0.90	0.94	0.90	0.93
10	0.83	0.89	0.88	0.90	0.92	0.86	0.93	0.84	0.93	0.89
Average	0.86	0.88	0.88	0.88	0.90	0.86	0.93	0.88	0.93	0.89
Standard deviation	0.02	0.02	0.02	0.03	0.03	0.03	0.02	0.03	0.01	0.02

Table 6.5: Results obtained by applying each fold's model trained with default D1 to the default version of the D3 dataset.

D3 - complementary										
Fold	Batch size 16		Batch size 32		Batch size 64		Batch size 128		Batch size 256	
	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR	TPR	TNR
1	0.91	0.92	0.93	0.92	0.90	0.93	0.84	0.93	0.87	0.92
2	0.86	0.93	0.91	0.93	0.91	0.93	0.91	0.92	0.92	0.94
3	0.90	0.90	0.85	0.94	0.89	0.92	0.90	0.92	0.92	0.92
4	0.91	0.92	0.80	0.93	0.93	0.92	0.93	0.92	0.81	0.93
5	0.91	0.93	0.92	0.91	0.90	0.93	0.90	0.93	0.92	0.92
6	0.94	0.90	0.89	0.94	0.92	0.92	0.86	0.93	0.90	0.93
7	0.87	0.94	0.90	0.93	0.87	0.94	0.89	0.92	0.89	0.92
8	0.93	0.92	0.95	0.89	0.89	0.93	0.88	0.92	0.89	0.93
9	0.95	0.91	0.92	0.91	0.94	0.93	0.90	0.93	0.93	0.92
10	0.90	0.91	0.93	0.92	0.92	0.91	0.91	0.93	0.90	0.93
Average	0.91	0.92	0.90	0.92	0.91	0.93	0.89	0.93	0.90	0.93
Standard deviation	0.03	0.01	0.04	0.02	0.02	0.01	0.03	0.01	0.04	0.01

Table 6.6: Results obtained by applying each fold's model trained with complementary D1 to the complementary version of the D3 dataset.

Having been presented in Tables 6.1-6.6, the results are summarized graphically in Figures 6.1-6.8 for better interpretation and analysis.

Pareto Lines

Figures 6.1 to 6.4 illustrate the distribution in a Pareto line of how well each batch size performed for both our default and complementary datasets.

When analysing these graphs, it is important to take into account the scale adopted to make it easier to differentiate between different points.

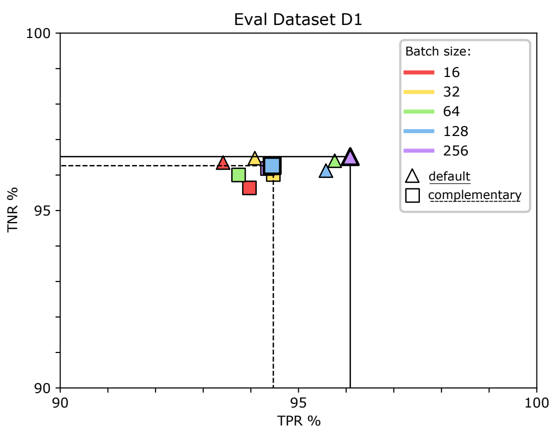


Figure 6.1: Pareto line for D1.

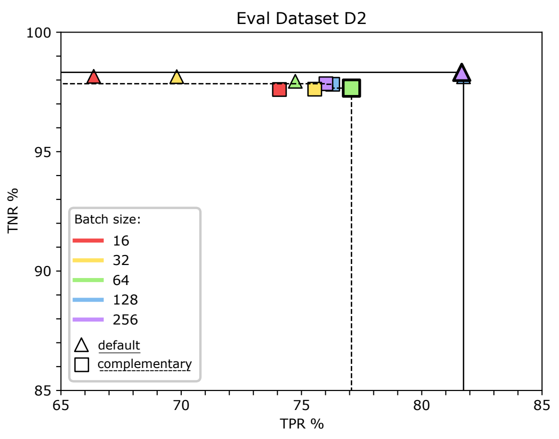


Figure 6.2: Pareto line for D2.

From Figures 6.1 and 6.2, it is possible to observe that batch size 256 and the default version dominate the results in D1 and D2, with a larger margin for TPR.

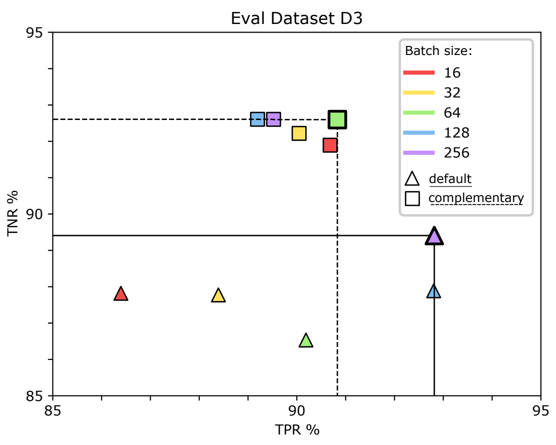


Figure 6.3: Pareto line for D3.

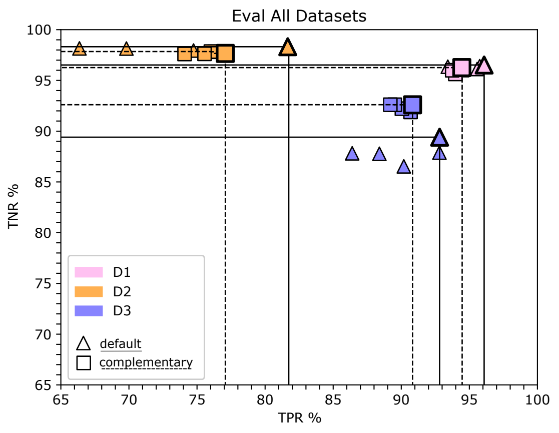


Figure 6.4: Pareto line for all datasets (D1, D2, and D3).

From Figure 6.3, one observes that the domination observed in the previous two Figures is not so conclusive: the larger batch sizes (128 and 256) with the default version have better TPR scores. However, concerning TNR, the respective scores are dominated also by larger batch sizes but with the complementary version. There are two scores that are clearly better.

From Figure 6.4, it is possible to observe the relation between the results of the D1, D2 and D3 datasets, which are organized in well defined clusters. D1 and D2 have high scores for TNR but D1 clearly outperforms D2 in terms of TPR score. D2 displays a higher TNR most likely due to the skewed ratio between negative and positive data. With D3, in turn, some degradation of performance is observed, but with both TNR and TPR scores keeping a good balance among them.

The results for D2 are notorious given the unbalance between TPR and TNR, mostly because of the worst results in TPR, irrespective of the dataset version, default or complementary. This is probably due to the combination between the circumstances that the datasets are small and that in D2 there is a larger unbalance between positive and negative examples.

Euclidean Distance

An alternative way of visualizing the performance of the data is comparing the distance between each set's TPR and TNR and the optimal point (100% TPR and TNR). The following graphs show how far the average rates were from a perfect run for each batch size.

Tables 6.5 and 6.6 display the average euclidean distance obtained per batch size in each dataset, considering the default and complementary version, respectively.

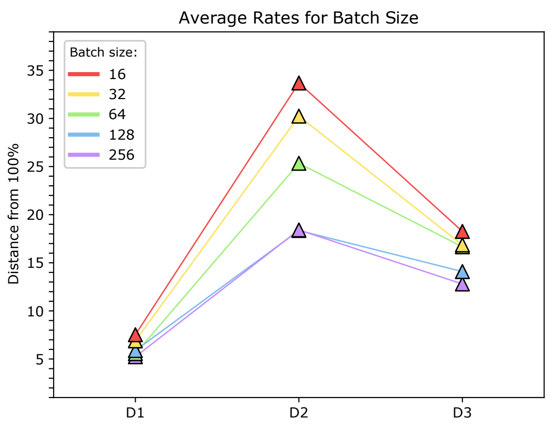


Figure 6.5: Average euclidean distances to the optimal result obtained using default datasets.

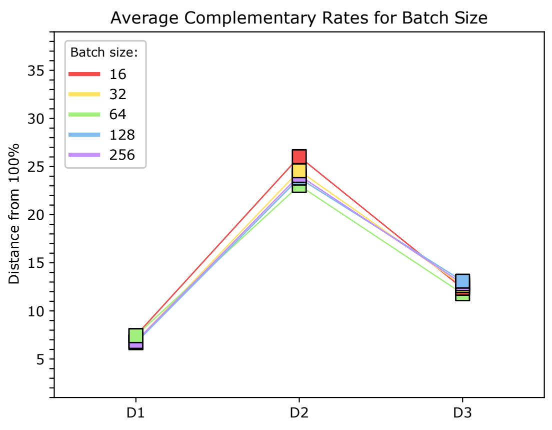


Figure 6.6: Average euclidean distances to the optimal result obtained using complementary datasets.

One can observe in Figures 6.5 and 6.6 that there exists higher variability in dataset D2, mostly with the default version.

With respect to the complementary version, the results show less variability, with batch size 64 being the one with better results both for D2 and D3.

Concerning the default version, in turn, the largest the batch the better the scores, with the largest batch size 256 supporting the best result in all datasets. It is possible that this batch size is coming close to the best euclidean distance for these experimental circumstances once its scores show little improvement to the immediately better ones, obtained with batch size 128.

Bar Charts

Finally, to enable comparison with the results obtained in [25] , we also present our data grouped in rectangular bars with lengths proportional to the TPR and TNR scores. We first present the graphs for the default datasets in Figure 6.7 followed by the complementary datasets in Figure 6.8.

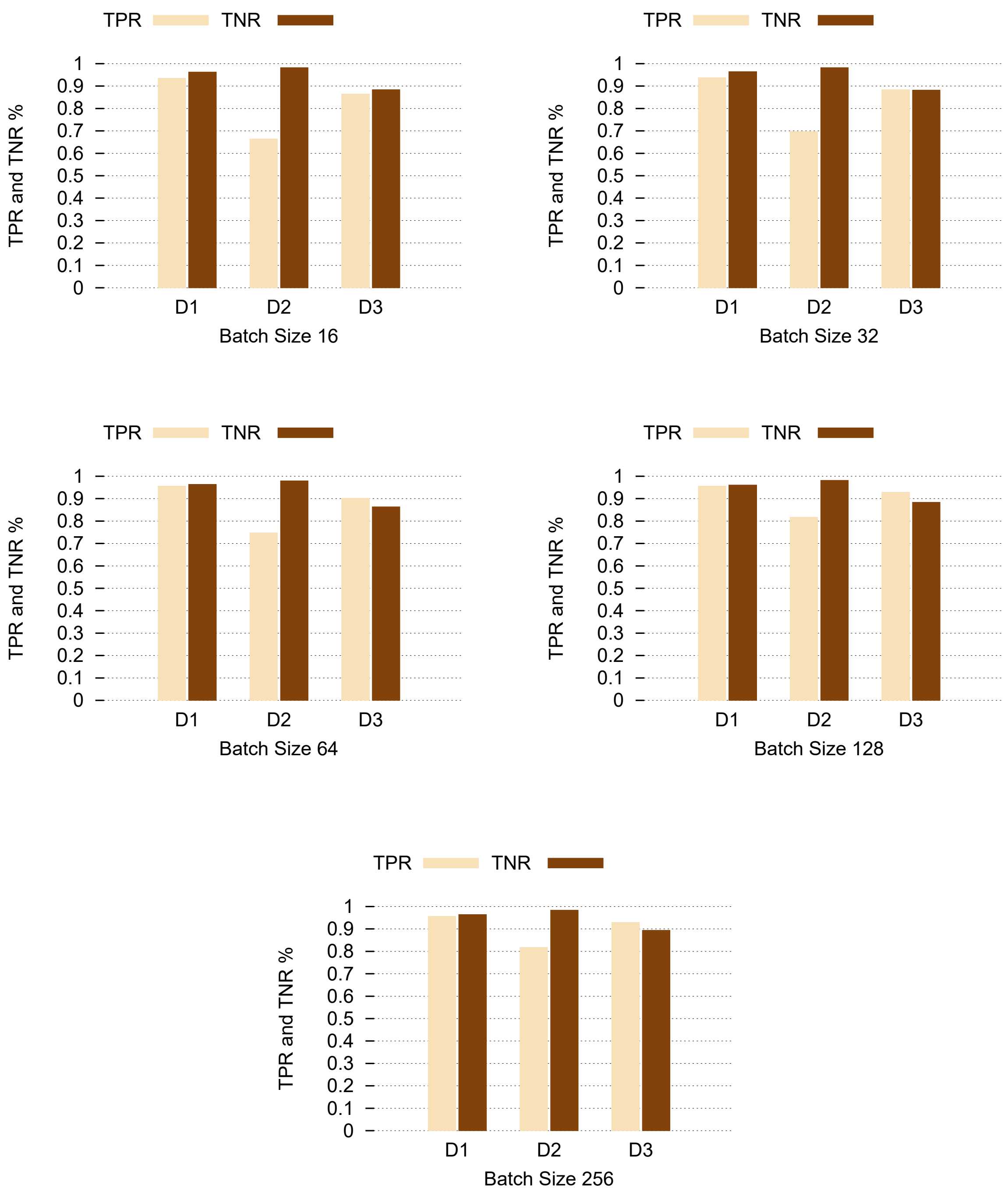


Figure 6.7: TPR and TNR for the default dataset by batch size.

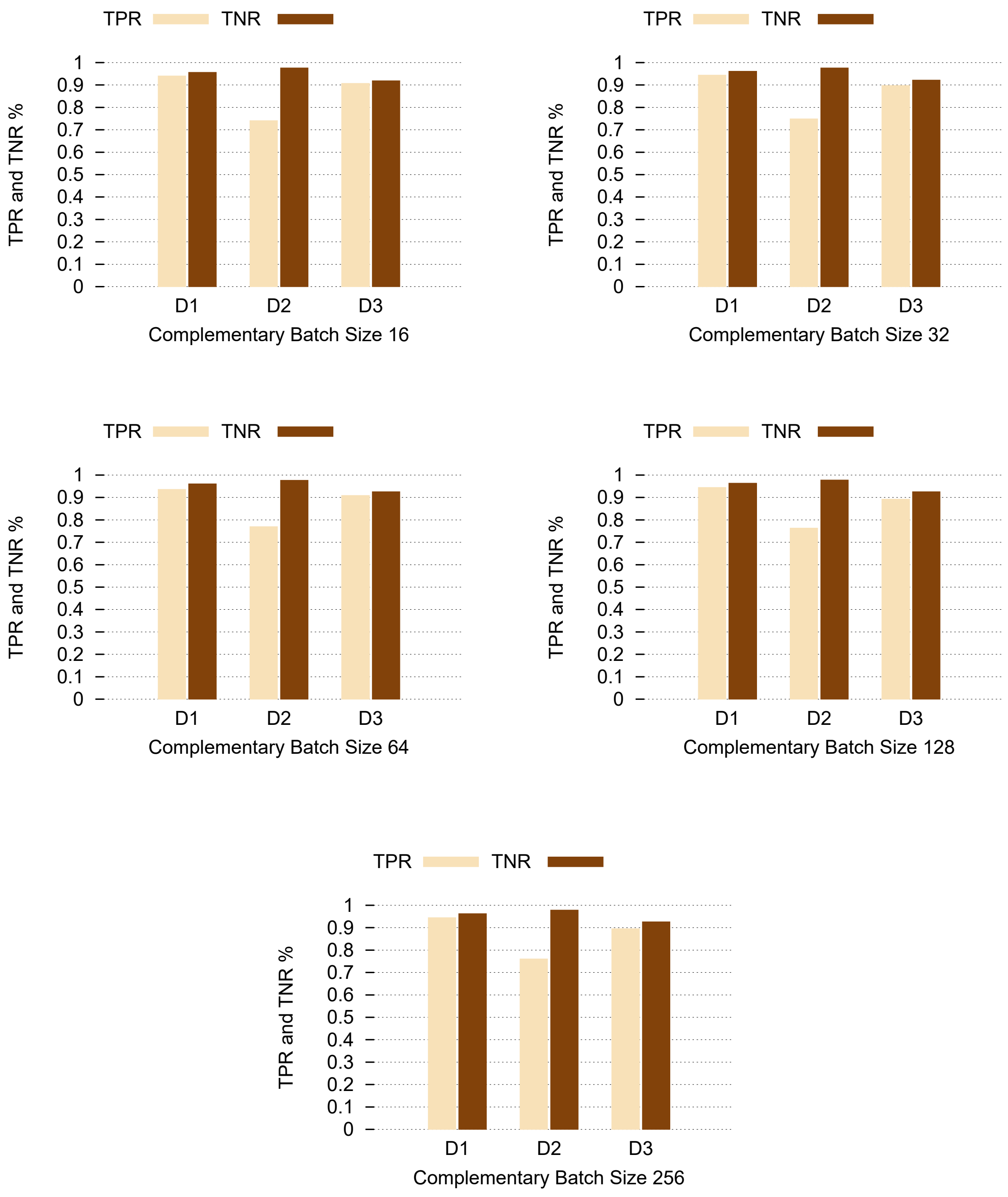


Figure 6.8: TPR and TNR for the complementary dataset by batch size.

6.2 Discussion

As expected, overall results are slightly worse in D2 and D3 when compared to D1. This effect stems from the fact that the data in D2 and D3 are subsequent to D1 and come from a wider range of Twitter accounts, thus including more diverse text styles. As time passes and new accounts are added, we expect this effect to manifest with greater impact.

Focusing on the results obtained in datasets D2 and D3 we see that the classifiers maintain high TPR and TNR scores. However we also observe that D2 exhibits a significant drop in TPR. This might be explained by the fact that this dataset has the smallest number of positive *tweets* and the largest imbalance between positives and negatives (see Table 5.6).

In most cases, the TNR is higher than the TPR. A few exceptions can be noted in D1 default (batch size 64, 128, and 256), where TPR ties with TNR. In D3 default there are also a few occasions where TPR is higher than TNR, more specifically in batch size 64, 128, and 256.

In general a higher TNR might be explained by the imbalance between positively and negatively labeled data in the training data sets, which favors the TNR.

Overall, the default and complementary datasets differ only by small margins in the TNR and TPR scores. There is however a smaller variance in results for D2 in the complementary dataset comparing to the default dataset. As noted before, this derives from the unbalanced data ratio between positive and negative *tweets*, which then prompts an artificial TNR boost. Despite producing better results in D2, the complementary dataset does not yield better overall results.

From the graphs it is also apparent that smaller batch sizes produce overall worse results which we attribute to overtraining. When evaluated with smaller batch sizes, the models are updated more frequently with partial changes related only to portions of the data. These frequent, partial updates might induce a certain level of overtraining, thus fitting the model inadequately.

Comparing these results to those achieved previously by means of Support Vector Machines and Muli-Layer Perceptrons [25], we observe that for D1, both results are comparable, irrespective of the datasets version (default or complimentary). For D2, we achieve a substantially higher TNR with both versions, while the respective TPR scores are lower in both versions. Finally for D3, our results are better independently of the datasets version, with the complimentary achieving the best result by a small margin.

Chapter 7

Conclusions and Future Work

In this chapter we provide a summary of the major achievements of our work and a compilation of its main results.

Our purpose was to develop a program and methodology that, after training with a dataset composed by *tweets*, can then accept and evaluate other datasets of the same type generating an output which determines if the previously unseen *tweet* mentions a threat to predefined Information Communication and Technology (ICT) infrastructures.

To perform this cyberthreat detection, we employed a convolutional neural network (CNN) that requires little human guidance.

Our findings indicate that smaller batch sizes produce worse results which we attribute to overtraining. Most likely this occurs because smaller batch sizes impose a high number of total steps despite not covering a large enough extent of the dataset in each step.

We also learned that, as the batch size gets larger, a model trained over the default datasets generally outperforms the respective complementary one even if the former tends to have a higher variance in the result. We attribute this to the data imbalance of our datasets regarding the positive to negative ratio.

In terms of real usage, we think that the solution developed is highly competitive. It seems like a viable solution for Security Operating Center (SOC) analysts to detect only the most relevant information in *tweets* about possible threats against the infrastructures for which they are responsible.

7.1 Future Work

During the course of the work, new ideas and goals spread beyond what we could hope to do within the available time. We now dedicate some sections to provide a starting point for someone else to expand on the work we started.

7.1.1 Model and Training Parameters

Unfortunately, due to the limited time available, it was not viable to test every hyper-parameter of the CNN and every training parameter.

We are confident that we tinkered with the one that influences the output the most, namely batch size.

We think that it would be useful to experiment with increasing further the batch size. Additionally, we think that it would be useful to experiment with other parameters such as filter sizes, number of convolution filters and word embeddings dimension (regarding the CNN design), and the number of epochs and regularization parameter (regarding the learning process).

7.1.2 Relation Extraction and Relation Classification

Nguyen and Grishman [49] explore CNNs for Relation Extraction and Relation Classification tasks. In addition to word vectors, the authors use the relative positions of words to the entities of interest as an input to the convolutional layer. This models assumes that the positions of the entities are given, and that each example input contains one relation. Both Sun et al. [50] and Zeng et al. [51] have explored similar models and we believe that this could represent an improvement to the model we used.

Another interesting use case of CNNs in Natural Language Processing (NLP) can be found in [52] and [53]. These papers describe how to learn semantically meaningful representations of sentences that can be used for information retrieval.

7.1.3 Word Embeddings

Most CNN architectures learn embeddings for words and sentences but not all papers focus on this aspect of training or investigate how meaningful the learned embeddings are.

Weston et al. [54] present a CNN architecture that predicts hashtags for Facebook posts, while at the same time generating meaningful embeddings for words and sentences. These learned embeddings are then successfully applied to another task, namely the recommendation of potentially interesting documents to users.

We think it would be interesting to experiment with this type of architecture and reach a model that is a hybrid between the one we implemented and the one proposed in these papers, along the lines of the seminal work by Collobert and Weston [55].

7.1.4 Character- and Subword-level CNNs

So far, all models presented were based on words. But there has also been research on applying CNNs directly to characters. The model proposed by Santos and Zadrozny

[56] learns character-level embeddings, joins them with pre-trained word embeddings, and uses a CNN for Part of Speech tagging.

In the same line of research, some more recent works of Bojanowski et al. [57] and Joulin et al. [58] also present interesting results regarding this idea. Its library can be consulted in [59].

Zhang et al. [60] explore the use of CNNs to learn directly from characters, without the need for any pre-trained word embeddings. The authors use a relatively deep network with a total of 9 layers, and apply it to Sentiment Analysis and Text Categorization tasks. Results show that learning directly from character-level input works very well on large datasets, but underperforms with simpler models on smaller datasets.

Kim et al. [61] also investigate the application of character-level convolutions to Language Modeling, using the output of the character-level CNN as the input to an Long Short-Term Memory neural network at each time step. The same model is applied to various languages.

This approach could be extended to our model when working with larger datasets allowing us to have a truly scalable solution for cyberthreat detection in OSINT.

References

[1] Diversity Enhancements for Security Information and Event Management (DiSIEM) Project. `http://disiem-project.eu/`. DiSIEM | Scalable information extraction and machine learning algorithms. Accessed: 2016-11-23.

[2] Horizon 2020. `https://ec.europa.eu/programmes/horizon2020/`. Horizon 2020 | The EU Framework Programme for Research and Innovation. Accessed: 2016-11-23.

[3] LaSIGE. `http://www.lasige.di.fc.ul.pt/`. LaSIGE | Large-Scale Informatics Systems Laboratory. Accessed: 2017-02-27.

[4] Navigators. `http://www.navigators.di.fc.ul.pt/wiki/Main_Page`. Navigators | Distributed systems research team. Accessed: 2016-11-02.

[5] Igor Kotenko and Andrey Chechulin. Attack modeling and security evaluation in siem systems. *International Transactions on Systems Science and Applications*, 8: 129–147, 2012.

[6] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. A proposal for the dartmouth summer research project on artificial intelligence. *AI magazine*, 27(4):12, August 2010.

[7] Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

[8] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[9] Christos Stergiou and Dimitrios Siganos. Neural networks., 2010.

[10] Rémi Domingues, Francesco Buonora, Romain Senesi, and Olivier Thonnard. An application of unsupervised fraud detection to passenger name records. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 54–59. IEEE, 2016.

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Overview of supervised learning. In *The elements of statistical learning*, pages 9–41. Springer, 2009.

[12] Zoubin Ghahramani. Unsupervised learning. In *Advanced lectures on machine learning*, pages 72–112. Springer, 2004.

[13] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. Semi-supervised learning (chapelle, o. et al., eds.; 2006) [book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542, 2009.

[14] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Domain adaptation for large-scale sentiment classification: A deep learning approach. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 513–520, 2011.

[15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[17] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[18] Hrushikesh Mhaskar, Qianli Liao, and Tomaso Poggio. Learning functions: when is deep better than shallow. *arXiv preprint arXiv:1603.00988*, 2016.

[19] Hrushikesh N Mhaskar and Tomaso Poggio. Deep vs. shallow networks: An approximation theory perspective. *Analysis and Applications*, 14(06):829–848, 2016.

[20] Alan Ritter, Evan Wright, William Casey, and Tom Mitchell. Weakly supervised extraction of computer security events from twitter. In *Proceedings of the 24th International Conference on World Wide Web*, pages 896–905. International World Wide Web Conferences Steering Committee, 2015.

[21] Carl Sabottke, Octavian Suciu, and Tudor Dumitras. Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits. In *USENIX Security Symposium*, pages 1041–1056, 2015.

[22] Bernardo de Simas Gaspar Rodrigues. *Open-source intelligence em sistemas SIEM*. PhD thesis, 2015.

[23] Kalyan Veeramachaneni, Ignacio Arnaldo, Vamsi Korrapati, Constantinos Bassias, and Ke Li. Aiˆ 2: training a big data machine to defend. In *Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and*

Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS), 2016 IEEE 2nd International Conference on, pages 49–54. IEEE, 2016.

[24] André Marques Correia. Aprendizagem automática em larga escala nas redes sociais para a descoberta de ameaças de segurança. Master's thesis, 2016.

[25] Fernando Alves, André Correia, Pedro M. Ferreira, and Alysson Bessani. Processing tweets for cybersecurity threat awareness. 2017. *Unpublished paper.*

[26] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[27] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[28] Denny Britz. Understanding Convolutional Neural Networks for NLP. `http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/`, Feb 2016. Accessed: 2017-07-13.

[29] Chen-Yu Lee, Patrick W Gallagher, and Zhuowen Tu. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. In *Artificial Intelligence and Statistics*, pages 464–472, 2016.

[30] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

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

[32] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[33] Peng Wang, Jiaming Xu, Bo Xu, Cheng-Lin Liu, Heng Zhang, Fangyuan Wang, and Hongwei Hao. Semantic clustering and convolutional neural network for short text categorization. In *ACL (2)*, pages 352–357, 2015.

[34] Tomas Mikolov and team. word2vec: Vector Representations of Words (Tensorflow). `https://www.tensorflow.org/tutorials/word2vec`. Accessed: 2017-07-13.

[35] Fengyu Cong, Andrew Leung, and Qinglai Wei. *Advances in Neural Networks-ISNN*. Springer, 2017.

[36] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *In EMNLP*, 2014.

[37] Ye Zhang and Byron Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820*, 2015.

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

[39] Rie Johnson and Tong Zhang. Semi-supervised convolutional neural networks for text categorization via region embedding. In *Advances in neural information processing systems*, pages 919–927, 2015.

[40] Peter Norvig. English letter frequency counts: Mayzner revisited or etaoin srhldcu. *Norvig. com*, 2013.

[41] Caffe. `http://caffe.berkeleyvision.org/`. Caffe | Deep Learning Framework. Accessed: 2016-10-05.

[42] Theano. `http://deeplearning.net/software/theano/`. Theano | A Python framework for fast computation of mathematical expressions. Accessed: 2016-10-03.

[43] Torch. `http://torch.ch/`. Torch | A scientific computing framework for LuaJIT. Accessed: 2016-10-17.

[44] Lua. `http://lua.space/webdev/the-best-lua-web-frameworks`. Lua |The Lua Community Blog. Accessed: 2016-10-17.

[45] Tensorflow. `https://www.tensorflow.org/`. Tensorflow | An open-source software library for Machine Intelligence. Accessed: 2017-07-11.

[46] Numpy. `http://www.numpy.org/`. Numpy | Fundamental package for scientific computing with Python. Accessed: 2016-10-17.

[47] Denny Britz. Artificial Intelligence, Deep Learning, and NLP: Implementing a Convolutional Neural Network for Text Classification in Tensorflow. `http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow/`, Feb 2016. Accessed: 2017-07-13.

[48] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[49] Thien Huu Nguyen and Ralph Grishman. Relation extraction: Perspective from convolutional neural networks. In *VS@ HLT-NAACL*, pages 39–48, 2015.

[50] Yaming Sun, Lei Lin, Duyu Tang, Nan Yang, Zhenzhou Ji, and Xiaolong Wang. Modeling mention, context and entity with neural networks for entity disambiguation. In *IJCAI*, pages 1333–1339, 2015.

[51] Daojian Zeng, Kang Liu, Siwei Lai, Guangyou Zhou, Jun Zhao, et al. Relation classification via convolutional deep neural network. In *COLING*, pages 2335–2344, 2014.

[52] Jianfeng Gao, Li Deng, Michael Gamon, Xiaodong He, and Patrick Pantel. Modeling interestingness with deep neural networks, June 13 2014. US Patent App. 14/304,863.

[53] Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Grégoire Mesnil. A latent semantic model with convolutional-pooling structure for information retrieval. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 101–110. ACM, 2014.

[54] Jason Weston, Sumit Chopra, and Keith Adams. # tagspace: Semantic embeddings from hashtags. 2014.

[55] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[56] Cicero D Santos and Bianca Zadrozny. Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1818–1826, 2014.

[57] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[58] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[59] Facebookresearch. *facebookresearch/fastText*. `https://github.com/facebookresearch/fastText`, Jul 2017.

[60] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

[61] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *AAAI*, pages 2741–2749, 2016.