UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

IMPROVING CYBERTHREAT DISCOVERY IN OPEN SOURCE INTELLIGENCE USING DEEP LEARNING TECHNIQUES

Nuno Rafael Marques Dionísio

Mestrado em Segurança Informática

Dissertação orientada por:
Prof. Doutor Pedro Miguel Frazão Fernandes Ferreira
e co-orientada por Prof. Doutor Alysson Neves Bessani

2018

Acknowledgments

I would like to begin by thanking my advisors Prof. Pedro Ferreira and Prof. Alysson Bessani. From the start of my thesis they guided and taught me about how to conduct my research. Prof. Ferreira has given me tremendous insight into a research area which I have been fascinated by and desire to further pursue in my academic career.

To all my colleagues at the Large-Scale Informatics Systems Laboratory (LaSIGE) for providing an outstanding environment to work in. The companionship amongst friends and their passion for their research has drawn me into pursuing an academic path. An additional thank you to Fernando Alves for all the advice and for helping me with my thesis.

As it must never be forgotten, I would like to thank my family for providing me all the opportunities that I have had throughout all my life. A unique and most dear thank you to my mother, who unconditionally seeks the best for me.

As a cruel gesture to be left for last, but in no way least, I thank my love Jéssica Catarino. Thank you for all your support, for all your playfulness, for all your care. You have been a constant source of happiness in my life, thank you.

Funding

This work was supported by the European Comission through the H2020 DiSIEM project ref. G.A. nº 700692, and by the FCT through the R&D unit LaSIGE, ref. UID/-CEC/00408/2013.

To my love, family and friends.

Resumo

São cada vez mais recorrentes as intrusões cibernéticas que afetam organizações e empresas, resultando em falhas de infraestruturas críticas, fuga de informação sensível e perdas monetárias. Com um aumento de ameaças à confidencialidade, integridade e disponibilidade dos dados, as organizações procuram informações relevantes e atempadas sobre potenciais ameaças cibernéticas à sua infraestrutura.

Esta aquisição de informação é normalmente feita por um Centro de Operações de Segurança que tem por objetivo detetar e reagir a incidentes de segurança. Porém as suas capacidades de reação dependem da informação útil e atempada que este recebe sobre ameaças cibernéticas, atualizações de software urgentes e descobertas de vulnerabilidades. Para tal é necessário ter acesso a uma plataforma que seja ágil e capaz de agregar diversas fontes de dados.

Ainda que a abordagem possa utilizar outras fontes de dados, o *Twitter* age como agregador natural de informação, sendo possível encontrar especialistas, companhias de segurança e até grupos de *hackers* que partilham informação sobre cibersegurança. Este fluxo de informação pode ser aproveitado por uma equipa de cibersegurança para obter informação atempada sobre possíveis ameaças cibernéticas.

No entanto, mesmo focando em contas de interesse, é necessário implementar um sistema que consiga selecionar apenas os *tweets* que contêm informação relevante sobre a segurança de ativos presentes na infraestrutura que se quer monitorizar. Devido ao elevado fluxo de dados, da necessidade de um algoritmo eficiente e escalável, e da capacidade de adaptar o algoritmo a uma determinada infraestrutura, procurámos implementar algoritmos de aprendizagem profunda, que pertencem ao subconjunto de algoritmos de aprendizagem automática.

Aprendizagem automática (*Machine learning*) é uma área no domínio de Inteligência Artificial que procura desenvolver algoritmos capazes de, sem intervenção direta de um agente humano, ajustar os seus parâmetros para desempenhar com maior eficácia uma determinada tarefa. Por vezes, estes algoritmos são capazes de alcançar desempenho superior à de um agente humano que fosse efetuar uma mesma tarefa. Normalmente tais tarefas são repetitivas e envolvem uma quantidade exuberante de dados.

Aprendizagem profunda (*Deep learning*) é uma subárea de aprendizagem automática que tem vindo a receber atenção devido às suas capacidades. De forma geral esta é uma

área, que recorrendo aos avanços no poder de computação e da quantidade crescente de dados, é capaz de treinar redes neuronais que contêm várias camadas. Este tipo de redes neuronais são usualmente chamadas de redes profundas (*deep*) e distinguem-se das redes mais tradicionais que agora se consideram de rasas (*shallow*). Redes neuronais rasas normalmente contêm apenas uma ou duas camadas escondidas e uma camada de saída. Cada camada é composta por neurónios inter-conectados que normalmente possuem a mesma funcionalidade. Por outro lado, as redes neuronais profundas tendem a possuir mais camadas escondidas, com diferentes camadas funcionais.

Dois tipos de redes profundas que são frequentemente utilizadas são as redes neuronais convolucionais e as redes neuronais recorrentes. Redes neuronais convolucionais são frequentemente utilizadas para tarefas de visão computacional devido à sua capacidade de processamento espacial. Dado uma tarefa e um conjunto de dados, este tipo de rede é capaz de aprender automaticamente várias características e padrões de uma imagem. Este tipo de arquitetura também pode ser aplicado a tarefas de processamento de texto, sendo capaz de captar relações entre diferentes sequências de palavras. O outro tipo de rede neuronal que tem obtido excelentes resultados são as redes neuronais recorrentes. Estas são frequentemente utilizadas para tarefas que envolvam uma dimensão temporal, como por exemplo o processamento de voz ou de texto. Ao contrário das redes já descritas, as redes neuronais recorrentes possuem um estado interno que age como a sua memória. Este estado de memória é uma camada de neurónios que mantém a sua ativação ao longo de uma determinada sequência. Por exemplo, na tarefa de processamento de texto, a rede neuronal recorrente irá receber uma palavra de cada vez. Ao processar uma palavra o estado dos neurónios que constituem uma camada da rede é mantido para o processamento da próxima palavra.

O trabalho realizado nesta dissertação visa melhorar e estender as capacidades de um sistema, atualmente em desenvolvimento, através de algoritmos de aprendizagem profunda. O sistema atual é capaz de receber *tweets* e através de um classificador baseado em máquinas de vetores de suporte, selecionar os que contêm informação relevante. Apresentamos duas redes neuronais, sendo a primeira uma alternativa ao classificador existente e a segunda um complemento que permite a extração de informação relevante de uma *tweet*.

A primeira contribuição deste trabalho é a implementação de uma rede neuronal convolucional como alternativa ao classificador de máquinas de vetores de suporte. Ao inserir uma *tweet* na rede, cada palavra é convertida num vetor numérico que contem uma representação semântica. Após a camada de conversão temos a camada convolucional. Esta camada irá produzir mapas de características que reportam sobre a existência ou ausência de uma dada característica na *tweet* através da ativação dos seus neurónios. Depois, cada mapa de características é reduzido ao seu valor mais elevado, este valor refere-se às ativações dos neurónios que estão inseridos na camada convolucional. Esta

operação permite reduzir a complexidade computacional e eliminar informação redundante. Por fim, a camada de saída contem uma função de ativação do tipo sigmoide (softmax) que permite classificar um *tweet* como sendo positivo (contem informação relevante sobre ameaças de segurança) ou negativo (não contem informação relevante). Em comparação ao classificador baseado em máquinas de vetores de suporte, o nosso classificador mostra resultados superiores, nomeadamente na redução do número de falsos positivos. A segunda parte deste trabalho envolve a implementação de um modelo de reconhecimento de entidades nomeadas para extrair informação relevante dos *tweets* que possa ser utilizada para o preenchimento de um alerta de segurança ou um indicador de compromisso. Para este fim, utilizámos uma rede neuronal bidirecional de memória longa de curto prazo, um tipo de rede neuronal recorrente, e definimos 5 entidades que queremos encontrar (organização, produto, versões, ameaças e identificadores de repositórios de vulnerabilidades) mais uma entidade para a informação não relevante.

A primeira camada desta rede é semelhante à do classificador. No entanto, este modelo contém uma camada opcional, igual à camada de conversão, que usa os carateres das palavras para criar uma matriz. Desta forma, cada palavra é representada por uma matriz em que cada vetor representa o valor semântico de um caracter. Este conjunto de vetores é enviado para uma rede neuronal bidirecional de memória longa de curto prazo secundária. A rede recebe um vetor de cada vez e no final produz um vetor que corresponde ao estado interno que representa o contexto da palavra com base nos carateres. Esta representação é adicionada ao vetor numérico da palavra de forma a enriquecer a sua representação final. Depois, os vetores são enviados para a rede neuronal bidirecional de memória longa de curto prazo principal. Ao contrário da rede anterior em que apenas se extraiu o ultimo estado, nesta rede extraímos o estado a cada intervalo de tempo (a cada palavra de uma *tweet*). Por fim, temos a camada de saída onde uma matriz de pontuações $n \times k$ é criada. Nesta matriz, n é o número de palavras que constituiem a frase e k o número de entidades distintas que podem ser atribuídas a uma palavra. A atribuição de uma entidade a cada palavra é feita selecionando a entidade com a pontuação mais alta. Porém, este método não considera as palavras vizinhas quando atribui uma entidade. Um módulo opcional chamado campos aleatórios condicionais é capaz de calcular uma pontuação para uma sequência inteira de entidades através da criação de uma matriz $k \times k$, sendo k o número de entidades, que automaticamente irá aprender pontuações para a transição de uma entidade para outra. Este processo permite que o modelo seja capaz de tomar em conta não só o contexto de uma palavra mas também o contexto das palavras vizinhas. O modelo obteve bons resultados, ambas as métricas como a média harmónica F_1 e a exatidão obtiveram resultados superiores a 90%, apresentando-se como uma forma viável para um sistema de extração de informação relevante sobre cibersegurança.

Palavras-chave: redes neuronais, aprendizagem profunda, deteção de ameaças de segurança, redes convolucionais, redes neuronais recorrentes

Abstract

The cyberspace is facing a challenge regarding the increasing security threats that target companies, organizations and governments. These threats cause the failure of critical infrastructures, disclosure of private information and monetary losses.

In order to guard and be prepared against cyber-attacks, a security analyst ought to be properly informed of the latest software updates, vulnerability disclosures and current cyber-threats. This requires access to a vast feed of information from various sources. One option is to pay for the access to such services. However Open Source Intelligence, which is freely available on the internet, presents a valuable alternative, specifically social media platforms such as Twitter, which are natural aggregators of information.

In this dissertation, we present a pipeline that aims to improve and expand the capabilities of a cyberthreat discovery tool currently in development. This tool is capable of gathering, processing, and presenting security related tweets.

For this purpose, we developed two neural networks. The first is a binary classifier based on a Convolutional Neural Network architecture. This classifier is able to identify if a tweet contains security related information about a monitored infrastructure. Once a tweet is classified as containing relevant information, it is forwarded to a Named Entity Recognition model. This model is implemented by a Bidirectional Long Short-Term Memory network and aims to locate and identify pre-defined entities in a tweet that may be used for a security alert or to fill an Indicator of Compromise.

Our classifier achieves favourable results: comparing to the current Support Vector Machine binary classifier it achieves equal or superior True Positive Rate and significantly better True Negative Rate. On the other hand, our Named Entity Recognition model is also capable of achieving great results, presenting an efficient method of extracting important information from security related text, with results above 90%.

Keywords: neural networks, deep learning, threat detection, convolutional neural networks, recurrent neural networks

Contents

List of Figures

List of Tables

Glossary

AGI Artificial General Intelligence.

AI Artificial Intelligence.

BiLSTM Bidirectional Long Short-Term Memory.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

CRF Conditional Random Fields.

CVE Common Vulnerabilities and Exposures.

DiSIEM Diversity Enhancements for Security Information and Event Management.

DL Deep Learning.

ENISA European Network and Information Security Agency.

GPU Graphics Processing Unit.

ICT Information and Communications Technology.

IoC Indicator of Compromise.

LSTM Long Short-Term Memory.

ML Machine Learning.

NER Named Entity Recognition.

NLP Natural Language Processing.

NN Neural Network.

OSINT Open Source Intelligence.

RNN Recurrent Neural Network.

SIEM Security Information and Event Management.

SOC Security Operations Center.

SVM Support Vector Machine.

TNR True Negative Rate.

TPR True Positive Rate.

Chapter 1

Introduction

Threat intelligence is a matter of most importance for a Security Operations Centre (SOC) which seeks to monitor, maintain and secure an infrastructure. The capabilities of a SOC depend heavily on its ability to be informed about the most recent software updates, patches, mitigation measures, vulnerability disclosures and current cyber-threats. Currently, Security Information and Event Management (SIEM) systems are used to collect and analyse data from multiple sources, identify potential threats and take the appropriate security measures. However, these processes are not efficient or optimized. SOC analysts try to be up to date about cybersecurity through multiple Open Source Intelligence (OSINT) feeds, which may be a tedious and time-consuming endeavour, with no guarantees that the information will be relevant to the infrastructure under the SOC's concern.

Threat intelligence systems are still limited in their capabilities of collecting and processing (OSINT). The European Network and Information Security Agency (ENISA) released a report on the opportunities and limitations of current threat intelligence platforms [20]. Due to vast volumes of data, that currently lead to difficulties in finding useful information, ENISA argues for the use of advanced analytic capabilities to generate complex relations, automatic tagging and classification of data. Current research has shown that OSINT provides useful information to create security alerts and Indicators of Compromise (IoC) [46, 48, 62].

Twitter is a social media site, acting as a natural aggregator of information due to their large and diverse pool of users, ease of accessibility, timeliness, and the large volumes of data it produces. These properties remain true in regards to the network and information security field. From security researchers, companies, enthusiasts, and hacker groups, there is a rich and timely flow of security-related data. Recently, a Twitter user made public a zero day vulnerability, providing a proof-of-concept exploit, in Microsoft Windows' task scheduler.[1] The exploit was only made officially public on the 9th of September but the original tweet was posted on the 24th of August.[2] Sabottke et al.[62] classified such

[1] https://www.zdnet.com/article/windows-zero-day-vulnerability-disclosed-through-twitter/

[2] https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2018-8440

events, when a vulnerability is disclosed ahead of time before the official disclosure or an official patch, into three categories regarding their origin: disagreements about the panned disclosure date, coordination of the response to vulnerabilities in open-source software, and leaks from the coordinated disclosure process. Events like these, although not too frequent, are a concern, as there is a window of opportunity for an informed attacker to attempt the exploitation of a flaw.

The H2020 project Diversity Enhancements for Security Information and Event Management (DiSIEM) aims to tackle the challenges of collecting, processing and summarizing threat-related information from OSINT [18]. In this project, there is a Twitter cyberthreat detection framework currently in development which is capable of receiving data from Twitter, select the tweets which potentially contain relevant information and present these to a security analyst. The framework uses a binary classifier based on a Support Vector Machine (SVM) [14] to identify security-related tweets about pre-defined assets of an ICT infrastructure and then reduces the volume of information by clustering the security-related tweets.

The work presented in this thesis was developed in the context of the DiSIEM project, with the aim of exploring the potential of recent developments in Machine Learning (ML) to improve the classifier's performance and extend the end-to-end capabilities of the framework. In this context, we report on the development of Deep Learning (DL) neural network models to classify tweets as relevant or not to an ICT asset, and to perform Named Entity Recognition (NER) to extract useful information that can enrich IoCs. Besides improving IoCs, this information could be employed to improve a clustering stage that is currently used.

These neural network models are capable of computing features and patterns to solve a given complex task without the need to be specifically programmed to. Such complex tasks were previously done in part either by a human or by a long task-specific algorithm or were simply too taxing and complex for either option. The work developed in this thesis used a dataset related to an hypothetical ICT infrastructure and was further validated using datasets related to three real ICT infrastructures specified by three industrial partners of the DiSIEM project.

1.1 Motivation

The ability to efficiently collect and process OSINT is an important requirement for a modern SIEM system. Security analysts strive for awareness of the most relevant threats, security updates and adversarial campaigns that may be relevant for the ICT infrastructure they desire to monitor and secure. In order to do this, we require scalable algorithms capable of receiving large amounts of data and outputting timely, succinct and relevant security alerts. Collecting and processing OSINT is a fundamental approach in order to

be aware of the current threats in the cyberspace.

The tools and frameworks developed to aid in these endeavours should take advantage of state-of-the-art algorithms that have achieved great results in other fields. The task of collecting and processing text from OSINT and extracting relevant insight belongs to the Natural Language Processing (NLP) field. Recently, this field has received a boost in its capabilities thanks to the advances in DL.

The work described in this document is part of the DiSIEM project and as such shares the same motivations. More specifically for our contribution, we seek to improve the quality of the classifiers used to select tweets containing relevant information and provide a valid option of extracting information from these relevant tweets.

1.2 Objectives

The purpose of this dissertation is to improve and expand the capabilities of a Twitter cyberthreat detection system currently being developed in the DiSIEM project and to evaluate the applicability of DL and NLP methods in the realm of cybersecurity intelligence.

The system currently being developed is capable of receiving, processing and presenting tweets which contain relevant information regarding cyber-threats. It aims to provide relevant information to an analyst while avoiding presenting irrelevant information. The binary classifier being used in this framework is based on a SVM [14] algorithm and is used to identify the tweets containing valuable information. We seek to provide a DL alternative through the implementation of a Convolutional Neural Network (CNN) architecture for sentence classification [41]. Overall, we aim to build a classifier which is capable of selecting relevant tweets while achieving misclassification results as low as possible.

Regarding the expansion of the framework's current capabilities, we require a scalable and efficient method for extracting information from relevant tweets. A popular subtask of information extraction is NER, which seeks to locate and classify entities in a corpus of text. For this purpose we developed a Bidirectional Long Short-Term Memory (BiLSTM) neural architecture [45].

Besides fine-tuning the hyperparameters and design variables of both models, we study several architectural variations in order to extract knowledge about the applications of DL in the information security field. This work aims to guide future research and development of neural architectures for processing computer security-related text.

1.3 Contributions

We present an overview of the implementation of DL algorithms for the tasks of text classification and information extraction through named entity recognition. We explore in detail the architectures of two neural networks, the CNN and the BiLSTM, evaluating several configurable components, such as the design variables and hyperparameters. The evaluation procedure included an experimental dataset that has been previously used to train and evaluate a previous SVM classifier and a CNN classifier [4]. We use this dataset to both explore the configurable parameters and to evaluate our classifier against the SVM counter-part. Additionally, we manually labelled the experimental dataset such that we could train and evaluate a NER model. Due to the lack of alternatives, we do not compare this model with other approaches. However, we do perform a thorough exploration of the design variables in order to demonstrate that our approach is efficient and a valuable addition to the threat detection framework, namely in terms of IoC generation.

Besides the experimental dataset, we also present results obtained in three industrial case studies where we have taken specifications of ICT infrastructures from three partners of the DiSIEM project and demonstrate that our solutions are capable of obtaining great results in a real-word based scenario.

Our main contributions can be summarized as an improvement and an extension of a cyberthreat discovery tool through implementation of DL algorithms evaluated in the context of three industrial case studies. The system developed in this work is capable of receiving small pieces of text (e.g. a *tweet*), determine its relevance to an asset's security and, if deemed relevant, extract useful information in order to launch a security alert or improve an IoC. Besides providing an efficient processing pipeline, the solution presented in this dissertation is scalable and easily trainable. It only requires sets of labelled data, thus not requiring any manual feature extraction.

The work developed in this thesis allowed for the elaboration of a poster presented at the 3rd edition of the LASIGE Workshop where it received the best poster award[3] and a publication of a regular paper [17] at the 10th INForum - Simpósio de Informática (INForum 2018) with a nomination for best paper[4].

1.4 Document Structure

Chapter 2 elaborates on some of the main ideas behind our research and how it compares to similar work that has tackled security related information extraction from OSINT. We also define several notions that surround the technological effort of this dissertation such as artificial intelligence, machine learning, deep learning, neural networks, convolutional neural networks, recurrent neural networks and long short-term memory networks. In the

[3] https://lasige.di.fc.ul.pt/node/3360

[4] http://inforum.org.pt/INForum2018/premios-inforum

end, we provide an overview of the tools used for this project, namely the DL framework used to implement the NN architectures.

Chapter 3 lays out the pipeline that has been implemented. The objective of this section is to provide an overview of the overall architecture. We detail the flow of information as it goes through each block of our solution and how data is processed and transformed in each NN architecture.

In Chapter 4, we explore the architectural variations and fine-tune hyperparameters in attempt to obtain the best performance possible. We display our experiments and discuss the results. Then, we test our architecture on three different sets of data related to Information and Communications Technology (ICT) infrastructures of three companies from the DiSIEM project consortium. Closing the chapter, we discuss the results obtained and summarize the knowledge extracted from our experimentation.

Finally, in Chapter 5 we elaborate on the results achieved in the previous sections, how our solution performs and how viable it could be in a real world scenario. Furthermore, we elaborate on our future plans regarding research, deployment and overall improvement of the architecture's capabilities.

Chapter 2

Background and Related Work

In this chapter we brief on the technology and algorithms that lay the foundation for the work developed in this dissertation. We describe the basic architecture of the neural networks we developed for our tasks and we review and discuss previous work on cyberthreat intelligence and the application of deep learning techniques for NLP tasks.

2.1 Artificial Intelligence

Throughout history there have been some ideas that could be interpreted as a desire to create an artificial intelligent being, capable of advanced thinking, similar or superior to that of a human being.

However, the actual birth of Artificial Intelligence as a research field is set in 1956 at the Dartmouth Conference. One of the objectives of this field is to achieve the Artificial General Intelligence (AGI), a machine capable of reasoning and performing any intellectual task that a human being can, without being specifically programmed to do so. This is the goal of many research-focused organizations that have made great advancements in the field of AI [15, 54].

While the road to this goal has not yet been revealed, modern applications of AI fall into the "Narrow Artificial Intelligence" category. These machines are capable of learning to perform a specific task as well as humans or even better, when given enough examples to train upon.

The algorithms used in this dissertation fall into this category. We intend to create a system that, through labelled datasets, learns how to identify tweets which contain information about a potential threat to the infrastructure and then extract some specific elements to fill a security alert or an IoC.

2.1.1 Machine Learning

Machine Learning is currently the field of AI where most research is being done. As the name suggests, the machine is meant to "learn" about a given task. In the case of NNs, this

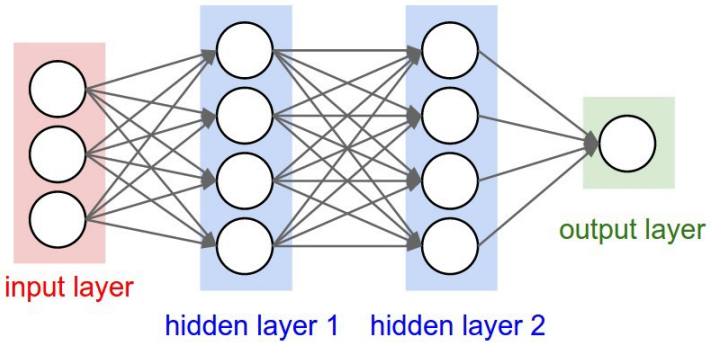

Figure 2.1: General Architecture of a Neural Network [37]. A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer.

learning process involves numerous training examples where the NN machine attempts to perform a task, such as classification, and depending on how far its predictions were, it will adjust its internal parameters in order to improve its accuracy. Once the machine is trained, it is capable of classifying new data, making predictions and even generating new samples. All of this without the need to write extensive code, detailing every routine or set of instructions.

There are many approaches in this field, Decision trees [58], Support Vector Machines [14], Clustering techniques such as K-means [35], Bayesian networks [22], Genetic Algorithms [24], Reinforcement Learning [68] but lately the big focus has been towards Deep Learning and Neural Networks [63].

2.1.2 Neural Networks

Known as *Artificial Neural Networks*, these algorithms are partially inspired by the structure and functionality of the human brain. A general representation of a feed-forward NN can be seen in Figure 2.1. They are usually organized in layers, each layer is made of a number of neurons (also known as *units*) which are interconnected with other neurons of the neighbouring layers, but not within the same layer. Notice that when addressing the number of layers of the network we do not count the input layer.

The overall process in which a network receives an input and outputs a prediction is called *feed-forward*. The data is fed through the *input layer*, then forwarded to the *hidden layers* where each neuron processes the information and finally in the *output layer* through an output function (e.g., a linear combiner or a *Softmax* function), an output is computed.

Multi-Layer Perceptron is one type of NN. In such architecture the input layer is where the data is fed to the network, the number of neurons present in this layer is equal to the amount of input features. Following an input layer we have the hidden layers. The majority of the feed-forward computation happens inside these layers. The neurons from

these layers receive the values from neurons of the previous layer, be it the input layer or another hidden layer, and compute their state through weighted sum of the inputs and addition of a bias value [6]. Both the weight and bias values are learnable parameters that the training algorithm adjusts during training. Besides these operations, in order to provide non-linearity properties these neurons often use a non-linear activation function. The most commonly used activation functions are the *Sigmoid* and *Tahn*. However, recently in DL architecture the most commonly used activation function for simple feed-forward networks is the *ReLU* [32]. These functions take a single number and perform a mathematical operation on it, offering non-linear properties to the NN, giving the capability to model more complex features for the task. Normally, the hidden layers are followed by a final layer meant to output a prediction. This layer is often called the output layer and its computational process is similar to a hidden layer layer. The difference is that in this layer the neurons do not perform an activation function, after their weighted sum and bias addition. Instead, this layer applies a specific function depending on the task at hand. In a classification problem, the most common function applied in this layer is a *Softmax* function, which squashes the values of each neuron into a sort of probability of how ”confident” a neuron is about a prediction. When outputting a prediction we simply pick the one which corresponds to the highest value.

During training, the probabilities that result from the *Softmax* function are used to adjust the NN parameters according to the prediction error. This is done through a learning rule that allows it to know how to modify the weights and biases in order to improve its accuracy. The most common rule used, which has had much success, is back-propagation [61]. In short, when a NN is first presented with an input, given that the weights and biases are randomly initialized, the network makes initial random predictions. By looking at the real result, the network can compute an error through a loss function. This loss function uses the distances of the probabilities outputted by the network and a representation of the correct prediction. Then, the back-propagation algorithm will compute a gradient which provides the “direction” in which the NN parameters should be adjusted. These gradients are used by optimization algorithms such as Stochastic Gradient Descent [40] or Adam Optimizer [42] which adjust the network’s weights and biases through various iterations in order to reduce the prediction error. To better adjust the network’s parameters, a common practice is to iterate through the training set several times. Each full iteration over the training set is called an *epoch*.

Overfitting

Overfitting is an issue to be kept in mind when designing, implementing and training a NN. It happens when a model includes more parameters than necessary and becomes too complex for the available data [30].

There are several techniques used to prevent this problem from occurring. In the work

described in this dissertation, it is important to address two of these techniques which we have implemented: *Dropout* [66] and L2 Regularization [51].

Dropout is a common technique where the key idea is to randomly drop neurons and their connections from the NN during training. In practice, we apply a *dropout* layer which, given a pre-determined probability, shuts off the communication of some neurons. This prevents neurons from co-adapting, meaning that neurons will not rely heavily on a few specific neurons, and in turn prevent against overfitting.

Another method to prevent overfitting is through regularization of the weights. L2 Regularization, also known as weight decay in ML, works by adding a penalization to the loss function. If we consider that the loss function measures how well the model performs, then this regularization function measures the complexity of a model. This complexity is usually calculated through the sum of the squares of all the weights. Meaning, weights with a high absolute value are more complex than weights with low absolute values. Through the optimization function, the model will adjust these outlier feature weights and prevent overfitting.

2.1.3 Deep Learning

A perhaps naive, but simple way of describing DL is by describing it as the usage of several (commonly a high number) hidden layers in a NN. Besides this increased depth provided by the increment in the number of layers, in DL the functionalities and data processing methods may also differ between the layers. In the past, the process of training a NN was computationally too expensive to permit the use of many hidden layers. However, with the constant increase in Central Processing Unit (CPU) computational power and employment of Graphics Procesing Units (GPUs) in ML tasks, training large NNs is now possible in commodity computers. Such development opened the gate for more complex neural architectures.

For instance, taking the example from Figure 2.1 we have three layers that process information. The first *hidden layer* contains 4 neurons, each connected to the neurons from the *input layer*. Thus between these layers there are 4×3 weights plus 4 biases, adding up to 16 learnable parameters. If we extend these calculations to the whole network we have a total of 41 learnable parameters. When considering modern neural architectures, we may be training up to several million parameters across multiple layers.

The architecture that we have been describing so far is often addressed as a *fully-connected neural network*. However in DL there are other architectures that have attracted a large amount of attention due to their improvement in real-world tasks, such as the *Convolutional Neural Network* [47] and *Long Short-Term Memory* [31].

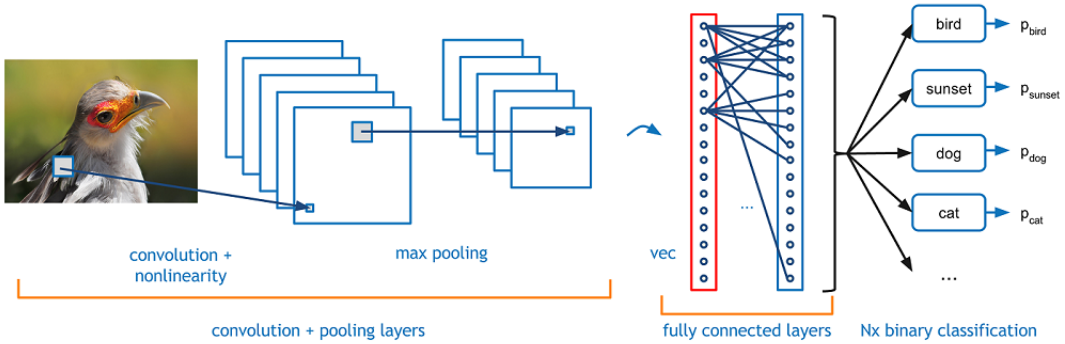

Figure 2.2: Basic architecture of a Convolutional Neural Network [16].

2.1.4 Convolutional Neural Networks

Many of the advances brought by DL can be attributed to CNNs. The main domain where these networks have been heavily deployed is computer vision. Large CNNs [43, 69] have been applied to numerous tasks [23, 39], often surpassing the results of the state-of-the-art at the time. Although these networks are most known for their applications in computer vision, they perform reasonably well in the NLP tasks [41, 36].

CNNs are, for the most part, similar to NNs such that their layers are constituted by neurons with trainable weights and biases. However, instead of each neuron from a previous layer being connected to all neurons from the next layer, the convolutional layer applies local connections to different groups of neurons, sharing the weight and bias values. A CNN is composed of multiple types of layers as shown in Figure 2.2. Traditionally, a CNN architecture for classification tasks starts with an *input layer*, a *convolutional layer*, a *pooling layer* and a finally a *output layer* containing a *Softmax* function to output a prediction.

Convolutional Layer

Not counting the *input layer*, the *convolutional layer* is usually the first layer in a CNN and where most of the computationally expensive operations happen. A convolutional layer consists of sets of learnable filters that perform operations upon an input matrix.

These filters can be thought of as sets of neurons, containing weights and biases, but unlike the traditional fully-connected networks these neurons are not connected to every neuron from the previous layer. These neurons are however connected locally to a sub-set of input neurons which are contained in the receptive field. A receptive field is the portion of the input being looked at by a filter.

When data is fed to the layer, usually in the form of a matrix, each filter is slid across the width and height of the input, performing operations at each stride similar to those from a hidden NN layer. The results obtained through the neurons' activations at each

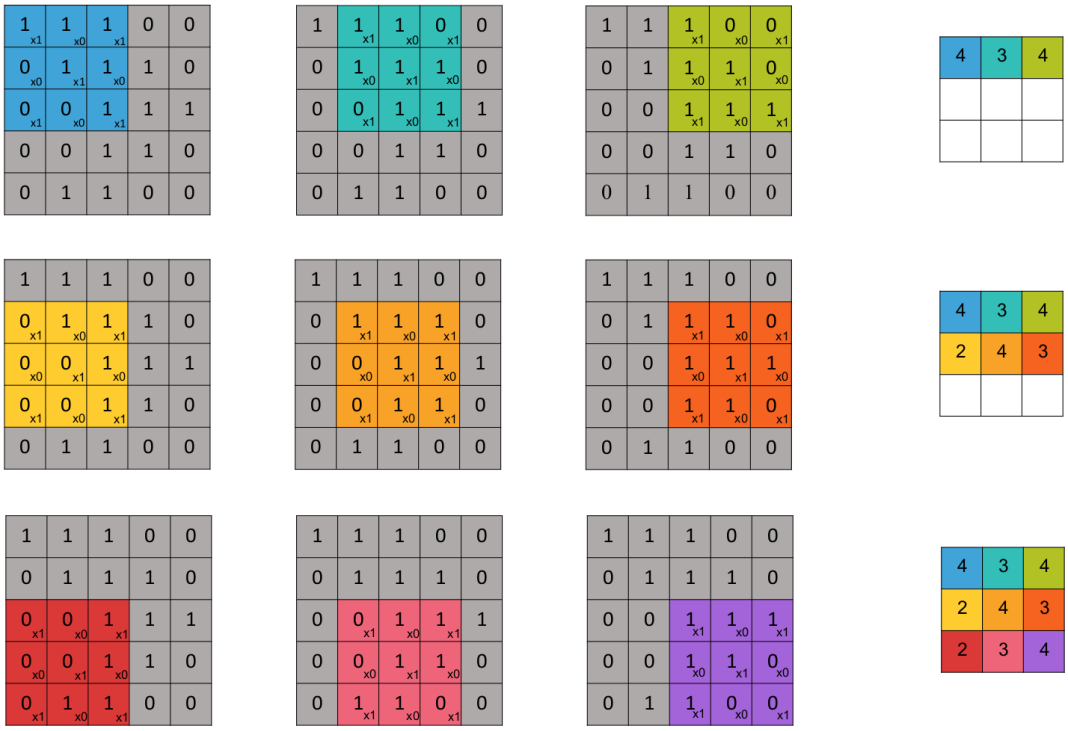

Figure 2.3: Example of a convolution in a sliding window function applied to a matrix of pixels [4].

stride are stored in a 2-dimensional feature map. These feature maps, also called activation maps, indicate how present a given feature is in the input. As we use more filters, we stack these feature maps along the depth dimension. The convolutional operation here described is exemplified in Figure 2.3.

Pooling Layer

A convolutional layer may result in a large amount of parameters, therefore increasing the computational effort. To balance the convolutional layer's output, it is a common practice to apply a pooling layer.

The pooling layer usually operates using the *Max* operation, hence being often named a Max-Pooling Layer. It takes a 2-dimensional window, similar to the receptive field in the convolutional layer, and selects the maximum value from that section. This window is slid across the width and height of the output feature maps from the convolutional layer, reducing the spatial size of the representation, thus reducing the amount of parameters and computation in the network. Besides helping with the computational effort, this operation also helps to prevent overfitting as redundant parameters and features are removed.

Once the pooling operation is complete, we flatten the results into a 1-dimensional array that is forwarded to the final fully-connected layer which will output a prediction.

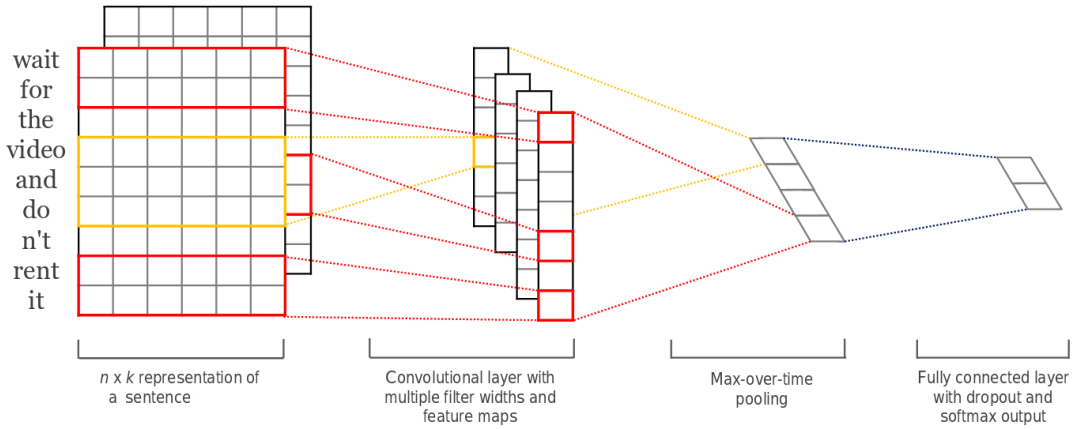

Figure 2.4: CNN architecture for sentence classification [41].

Sentence Classification

As our task does not involve computer vision, the network ought to be modified in order to be applicable to a NLP task. An architecture adapted for this is proposed in Kim [41] and is shown in Figure2.4.

In order to shape our sentences into a 2-dimensional matrix so that we may apply convolution, we require an embedding layer between the input layer and the convolutional layer. Through this layer we convert each word of a sentence into a numerical array called an embedded vector [9, 3]. These vectors hold a word's semantic value [49] and can be randomly initialized or imported from a pre-trained language model (e.g., Google News Word2Vec model [26]). The embedded word vectors are vertically stacked in order to form a $n \times k$ matrix.

Thus, this matrix can be forwarded into the convolutional layer where, as previously explained, through various filters the network will form feature maps. However, unlike the filters used in computer vision, where there are no restrictions on the height or width, the filters used in these architectures are usually bound in width to the length of the vector. Intuitively this makes sense as a filter should contain the whole extent of word vector in order to completely capture its semantic representation.

The resulting 1-dimensional feature maps are then sent to a Max-over-time-pooling Layer [11, 41]. Unlike the max-pooling operation explained before, where we define a window that will be slid through the feature maps dimensions, in this case the max-pooling operation will encompass the whole feature map. And as such, for each filter we extract one feature. All these features are then concatenated into a 1-dimensional vector and fed to the output layer. This final layer contains a fully-connected network, just as in the previous architecture, with a *dropout* function to prevent overfitting and the *softmax* function to compute the final prediciton.

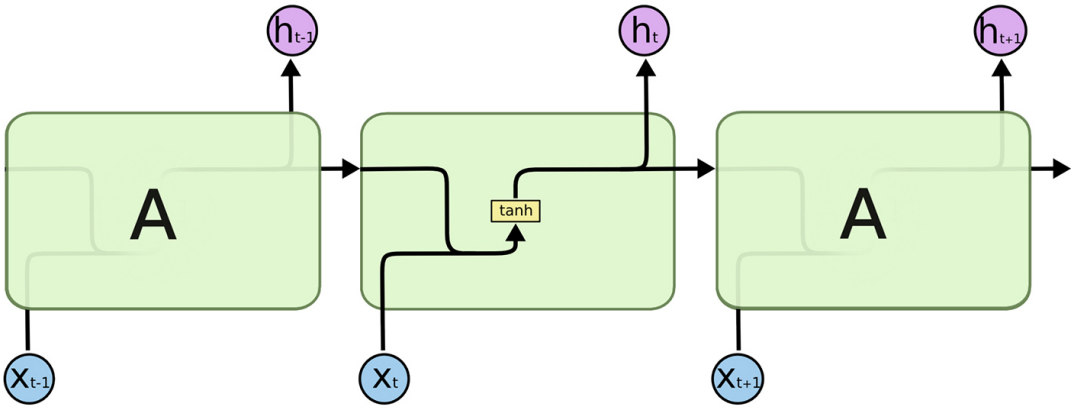

Figure 2.5: Unrolled Recurrent Neural Network [53].

2.1.5 Recurrent Neural Networks

Another pivotal neural architecture in DL is the Recurrent Neural Network(RNN). This kind of network has been achieving significant results in NLP tasks such as speech recognition [28] and language translation [7, 72].

The main idea behind these networks spurs from the necessity to retain knowledge to guide future decisions. Every time we read a document we associate value and meaning to each word depending on the ones that preceded it.

Traditional neural networks can not do this. It is unclear how a neural network could use its previous reasoning of an event to inform later ones.

Through the addition of loops into a network, we may allow information to persist. In Figure 2.5 we can see that the cell A is presented with an input x_t and outputs the value h_t. We can think of x_t as a word being read by the cell A at time t. The cell contains a hidden state, a vector of neurons, which receives an input x_t at every time step t and performs the traditional weight multiplication, bias addition and non-linear activation. Common practice in RNNs is to use the tanh activation function. However, unlike the traditional non-recurrent fully-connected networks, the result from each time step t is kept for the next time step $t + 1$, thus introducing memory.

However, these networks are faced with a particular challenge when they are required to retain information that may not have appeared so recently. As the length of an input sentence grows, a RNN may have difficulty in relating previous information with the most recent one.

Thankfully, there is a specific kind of RNN, which is responsible for the majority of the success behind RNNs, that does not suffer from this problem, the Long Short-Term Memory (LSTM) networks [31].

Long Short-Term Memory Networks

LSTM networks are quite complex when compared to regular NNs or CNNs and are capable of learning these critical long-term dependencies [31].

In Figure 2.6 we lay out a LSTM architecture. Like standard RNNs, this network can also be drawn as a loop or an unrolled chain as we have done. However, unlike the RNN in Figure 2.5 where we only have one neural network layer using a tanh activation function, the LSTM contains four different layers performing several different functions.

The horizontal line running through the top of the diagram is what is called the cell state. The LSTM has the ability to either add or remove information from this cell state through gates. These gates are a way to manipulate the information that is fed to the LSTM cell state. Normally they are composed of a sigmoid NN layer that outputs values between 0 and 1, which is used to describe how useful the information is and if it should be let through. Intuitively, 0 will mean that nothing should pass whereas 1 means everything should go through.

In short, the functionality of an LSTM can be described through three major steps [53]. The first step is often called the forget gate layer. Here the network decides which information should be thrown away from the cell state. It takes into account the previous state (h_{t-1}) and the new input (x_t) and outputs a number between 0 and 1. Then the following step is to decide which new information should be added to the cell state. We will have two components, the first is a sigmoid layer often called input gate layer that decides which values should be updated. The second component is a tanh activation layer that creates a vector of new candidate values which may be added to the cell state. These two are combined in order to create an update which will be added to the cell state. Finally, in the last step we have the output. The output will be based on the cell state but will suffer a minor transformation. First, we run through a sigmoid gate that decides which parts of the cell state should be in the output, then we pass the cell state through a tahn activation layer to keep the values between -1 and 1 and finally multiply it by the output from the sigmoid layer in order to output only the values we want.

In practice we tend to consider the LSTM to have two outputs at every time step, the cell state and the hidden state. If we intend to extract as many outputs as there are time steps, we usually retrieve the cell states which are unfiltered. On the other hand, when we intend to extract only one representation for the whole input, we tend to extract only the final hidden state.

2.2 Deep Learning Frameworks

With a surge in interest for DL and NNs, several companies and organizations have developed frameworks to ease the implementation of these techniques such as Google's TensorFlow [1], Facebook's Pytorch [55] or Microsoft's CNTK [64].

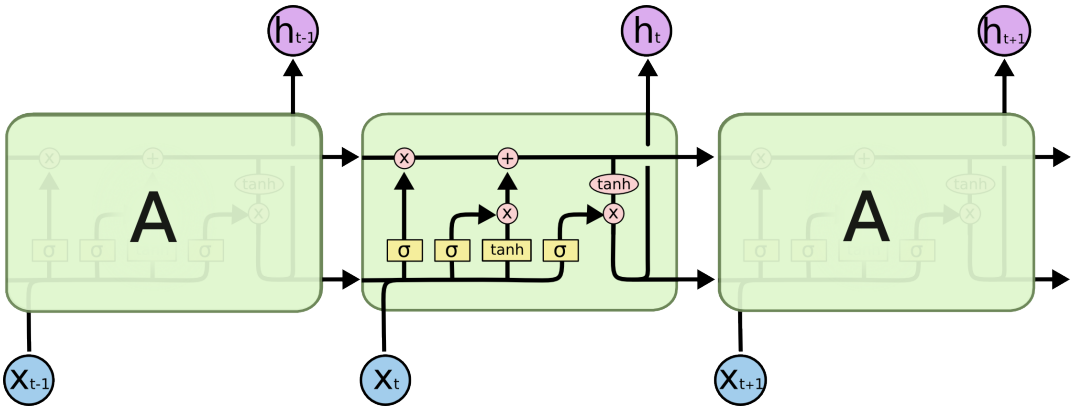

Figure 2.6: Unrolled Long Short-Term Memory Network Architecture [53].

Device	**Average time per Fold** (seconds)
i7-6850K	32.8
GTX 1080ti	2.8

Table 2.1: CPU vs GPU comparison.

In order to decide which framework would be best to use we took into account the capabilities provided, the quality of the documentation, its flexibility and extensibility, and finally, its official support and community activity. Thus, given the brief requirements we described, we decided to select TensorFlow [1] for its popularity, documentation, the numerous tools that come with it such as TensorBoard [1] and most importantly, given that it is relatively low-level, the capability to perform any action we may require before, after or between the layers of the model.

TensorFlow is an open source software library for numerical computation using data flow graphs, created and maintained by Google [1], capable of operating at large scale and in heterogeneous environments. The library is written in Python, C++ and CUDA, offering extensive flexibility through its Python API.

TensorFlow uses a dataflow graph to represent computation in terms of dependencies between operations [25] which are represented as nodes. These nodes contain tensors which are a generalization of a multidimensional data structure: from vectors and matrices up to any dimension [27].

A major advantage of using these frameworks is the ability to easily tap into the GPU's processing capabilities. Table 2.1 presents an experiment we conducted to demonstrate how much faster the GPU is at training a model than a CPU. This experiment involved the training of the same model described in Branco [4], using the same data, through a 10-fold cross validation method, which is detailed in Section 4.2.

[1] `https://www.tensorflow.org/guide/summaries_and_tensorboard`

2.3 Machine Learning in Cyberthreat detection

As previously stated, the work developed for this dissertation is inserted in the H2020 DiSIEM project. We aim to provide improvements to a framework currently being developed through the implementation of DL techniques. This framework, currently unpublished, proposes a methodology for collecting, processing and presenting security-related tweets. The pipeline is composed of text filtering, text feature extraction, binary classifiers and a novel clustering technique based on k-means. Our pipeline uses the same method for extracting *tweets*. However, we perform different pre-processing methods, we provide a DL alternative to the classifier and although we do not perform clustering, we present a DL technique to extract information from security relevant *tweets*.

In this section we provide an overview of previous work done in regards to threat intelligence from OSINT. A recent paper by Le Sceller et al. [46] presents SONAR, an automatic, self-learned framework that can detect, geolocate and categorize cyber security events in near real-time over the Twitter Stream. The framework makes use of the Twitter API and a list of keywords to retrieve a continuous stream of tweets. Overall, the architecture implemented is heavily keyword-based. The authors describe the framework in three phases. The first phase is similar to ours, the framework queries Twitter for a list of keywords in order to retrieve a continuous stream of tweets. However they query Twitter as a whole instead of focusing on a pre-defined set of accounts that are more likely to tweet security related content like we do. Moreover, the keywords that we use are not cyber threat related (e.g., attacks and vulnerabilities) but rather we query for a list of keywords related to assets from the infrastructure being monitored. The second phase of SONAR is about event detection. It uses a clustering technique to aggregate similar tweets that may be reporting about the same cybersecurity related event. These events are geolocated, classified and displayed on a user interface. Given that the system relies on its keywords being up-to-date, the third phase aims to find new keywords based on their co-occurrence with other previously defined tweets. These two phases differ from our scope since our aim is not to build a general information security news feed but rather a tool to gather the most recent information about threats, configurations or updates related to a pre-defined list of assets.

Sabottke et al. [62] implemented a Twitter-based exploit detector using a SVM classifier. The detector is capable of extracting vulnerability-related information from Twitter, augment it with additional sources and predict if the vulnerability is exploitable in a real-world scenario. The paper states that the detection of zero-day attacks before their public disclosure are not considered. Thus, as we have stated in our objectives before, our work differs in the way that we only use Twitter as a source and our classifier retrieves every security related tweet regarding a given asset. Meaning that we retrieve updates, configurations, exploits and potential zero-day vulnerabilities. One interesting component from this work that should be mentioned is the consideration of adversarial interference

in order to fool the classifier. The authors developed a threat model with three types of adversaries. These adversaries vary in their knowledge of the classifier and complexity of their poison-attacks that attempt to fool the classifier. As it stands, our system uses sets of pre-defined accounts that have been selected based on the likelihood that these users tweet about the security of elements belonging to the information technology infrastructure being protected. We have not yet implemented a reputation system capable of fetching new accounts, however the existence of adversarial agents should be kept in mind.

Regarding the task of information extraction in order to build IoC or security alerts, Liao et al. [48] present *iACE*. This tool is capable of extracting IoC in a fully automated way. However, one important difference is that it does so through the analysis of technical articles, which is a naturally richer information source than using Twitter alone. Furthermore, a major difference is the technique applied for the task. The authors take advantage of the predictability of some aspects from these articles such as a set of context terms and their grammatical relations. Our approach intends to explore the usability of DL techniques to extract these relations automatically. This capability aims to decrease the necessity for manual feature engineering.

2.4 Deep Learning in tweet analysis

In order to complement our overview of the state-of-the-art of threat detection in OSINT and given that our goal is to apply DL algorithms on tweets, in this section we provide a summary of previous work that has extracted information from Twitter and applied DL techniques in order to perform a given task.

A common application of DL algorithms in Twitter is sentiment analysis. Severyn et al. [65] proposes a CNN architecture similar to Kim [41] to conduct two subtasks of the SemEval 2015 Twitter Sentiment Analysis challenge.[2] The authors compare their results with the systems submitted to the tasks and show that their architecture was capable of ranking the first two positions in both subtasks. Rozental et al. [60] also used a DL method to perform sentiment analysis on Twitter data, this time in the proceedings of SemEval 2017, through a RNN architecture.[3]

Badjatiya et al. [2] experiments with several classifiers including, but not limited to, DL algorithms in order to detect hate speech in tweets. The authors report that the DL techniques perform significantly better than other methods.

Regarding NER, Jimeno-Yepes et al. [33] implemented a LSTM architecture for the annotation of medical entities mentioned in the Twitter stream in order to support tasks such as public health surveillance. The architecture presented by the authors outperforms the previous state-of-the-art and regarding future work, the authors plan on using the

[2]http://alt.qcri.org/semeval2015/task10/
[3]http://alt.qcri.org/semeval2017/task4/

architecture proposed by Lample et al. [45], which is the same architecture that we have based our own NER model on.

2.5 Final Remarks

In this chapter we have given a brief on the essential concepts of NN and DL that lay the groundwork for the work presented in the following chapter which will provide a detailed explanation of our system and how the data flows through the NN architectures. In this work, we explore the implementation of a CNN for text classification and a LSTM for information extraction, both of these architectures have been used for NLP tasks with significant success rates.

As described in this chapter, there is previous work on obtaining security-related information in OSINT, some of these also use Twitter as a source of data. However, none of these works explore the application of DL techniques in an end-to-end approach, form tweets to IoCs. As shown in Section 2.4, there is previous work on the application of DL techniques to Twitter data that often surpass the corresponding state-of-the-art methods.

Chapter 3

Architecture

In this chapter we describe our pipeline and the NN architectures that we have deployed. We begin by elaborating on the general problem, describing the current solution and what it seeks to achieve. Then, we establish where our pipeline will come into play, the elements it means to improve and the functionality it intends to add.

Furthermore, we describe in detail how the system processes a tweet, from the instant it is collected until the information is extracted.

3.1 Problem Statement

The system currently being developed in the context of the DiSIEM project allows analysts to specify the assets (machines, software, services) in their infrastructure. Then, the system will collect tweets related to such components, conduct the pre-processing that it requires and then (through a SVM binary classifier) it will identify possibly relevant tweets. To reduce redundancy a clustering algorithm is applied, once a cluster exceeds a given threshold of tweets, the tweet closest to its cluster's centroid is shown to the analyst. This process is depicted in Figure 3.1.

Globally, the system has three main objectives:

1. Maximize the amount of relevant information presented to the analyst;
2. Minimize the amount of irrelevant information presented to the analyst;
3. Aggregate related information.

The pipeline developed in this work is shown in Figure 3.2. It aims to replace the pre-processing, feature extraction and classifier stages in the original pipeline. Besides a minor alternative to the pre-processing stage, our first contribution aims to substitute the SVM classifier with a CNN. Our goal for this stage is simply to improve on the first two original objectives.

For the NER module we describe our Bidirectional Long Short-Term Memory (BiLSTM) network and its architectural variations. This part of our work adds a novel method

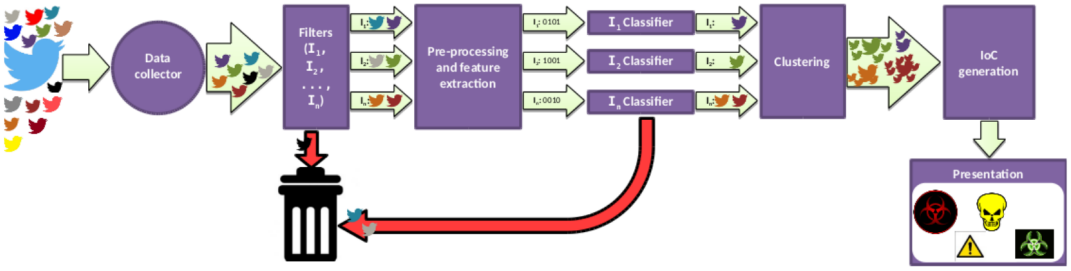

Figure 3.1: Twitter threat detection pipeline

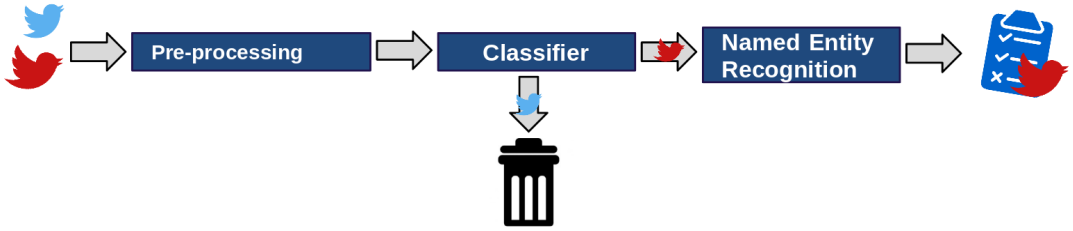

Figure 3.2: Deep Learning pipeline.

of information extraction to obtain security-related information that can be used to build an IoC, which is a semi-structured document descibring a potential cybersecurity threat.

3.2 Data Collection

As described by Branco [4] and Correia [13], in the data collection stage we establish a set of Twitter accounts. These accounts can be of security analysts, companies, hackers, users or security researchers. From these accounts we will retrieve a stream of tweets through Twitter's public API [70, 71].

3.3 Pre-Processing

In this stage we normalize the tweet representation. Regarding the processing that goes into cleaning a tweet we explore two options which we have named *full* and *partial*. Then we explain the following tokenization process until a tweet is ready to be fed to the network.

Another object of exploration is the padding, however this option is only relevant in the training stages. We present two options regarding the padding scheme, we named them as *static* and *dynamic*.

Further into this dissertation these options are object of experimentation and we evaluate if there is any benefit in choosing one over the other.

Original	Vuln: Oracle Java SE CVE-2015-2625 Remote Security Vulnerability https://t.co/mNpDapXWz2
Partial	vuln oracle java se cve-2015-2625 remote security vulnerability
Full	vuln oracle java se cve hyphen two thousand and fifteen hyphen two thousand and twenty-five remote security vulnerability

Table 3.1: Partial and Full tweet cleaning.

3.3.1 Representation

As stated before, we provide two options regarding the cleaning pipeline that a tweet must go through. Afterwards, we provide some additional details regarding unknown words that may appear in a testing or production setting.

Partial Cleaning

For the *partial* cleaning option we convert all characters in a sentence to lower case, remove all hyperlinks and special characters except the "." and "-" since these may be important to detect version numbers or identifications from vulnerability repositories such as the Common Vulnerabilities and Exposures (CVE) repository.

Full Cleaning

This second option is the one originally used in the DiSIEM project and it is similar to our *partial* cleaning pipeline, however numbers are also converted to their text form (e.g., "2" to "two"). This also applies to the punctuation that was spared previously, meaning "." and "-" are respectively converted to "dot" and "hyphen". Table 3.1 displays an example of both *partial* and *full* cleaning options.

Tokenization

After cleaning a tweet, we proceed to our tokenization process. Here we split the text into an array of words and substitute each word by an index. This index is obtained from our dictionary, which is built during the training process. Put simply, a new word is registered along with its corresponding index when it appears in the training datasets. When receiving an unknown word outside of the training process, we apply the index for the token `<UNK>`, purposely created for this situation.

3.3.2 Padding

Besides normalizing each tweet regarding their content, when feeding a batch to the network we require that each tweet contains the same length, this is specially important for

the training process. As we should expect, the tweets received will not have the same length and thus they should be padded with a token which we have defined as <PAD>.

There are two options for this: *static* or *dynamic*.

Static

In the static method we pad every tweet in the training dataset to the maximum length found within the set. During testing or deployment stages, if a received tweet exceeds the maximum value defined during the training stage then the text in excess is ignored.

Dynamic

On the other hand, the dynamic method pads the input tweets only after the batching process. Independently of the stage a network is being used, when feeding a group of tweets to the network, the padding process will use only the local maximum length and pad the batch accordingly.

3.4 Classifier

This section describes the implementation of our approach to the binary classifier. The architecture here described is based on the CNN for sentence classification by Kim [41].

3.4.1 Convolutional Neural Network

We describe the model by detailing the data flow during each step of the feed forward process.

The general architecture of the CNN is displayed in Figure 3.3. In short, the network receives a tweet in the input layer, then converts indexes to vectors in the embedding layer, followed by the convolutional layer containing several different kernels with the same number of filters, a max-over-time-pooling layer, and finally the output layer which applies dropout and a softmax function for classification.

Input Layer

The network receives a tweet which can be represented as a sequence of n words.

Embedding Layer

In order to capture the semantic representation of a word we require an embedding stage. Here, each token is converted into an embedded vector. These numerical vectors are d-dimensional and can be initialized randomly or imported from a pre-trained language model [49, 56, 34].

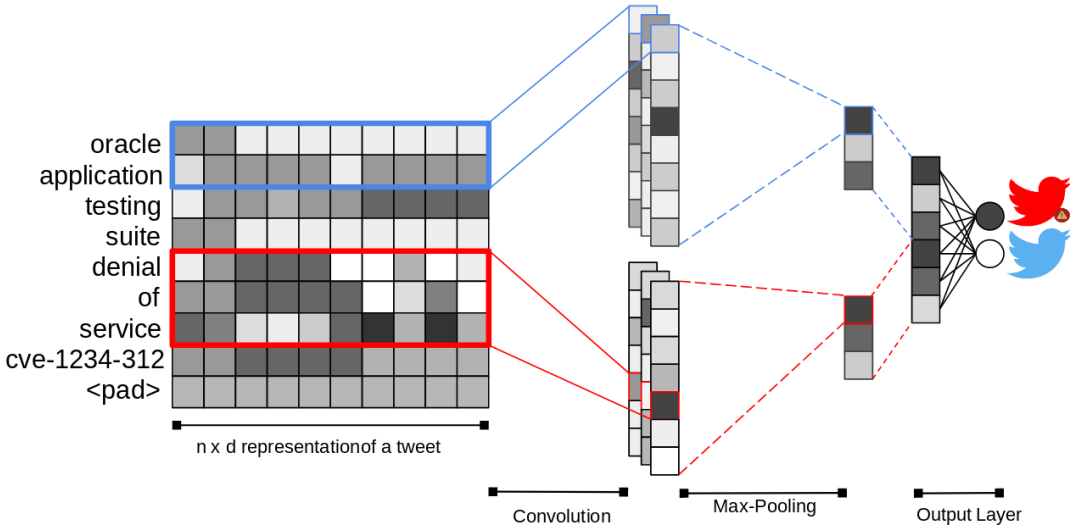

Figure 3.3: CNN architecture for sentence classification

Furthermore, these representations can be fine-tuned automatically during the training process, which allows even the randomly initialized vectors to capture some task related semantic value.

Once every tokenized word from a tweet is converted into an embedded vector, we have a $n \times d$ matrix representation of a tweet where n is the number of words and d is the length of the embedded vectors.

However, to be compliant with the convolutional operation, we require to add a third dimension which represents the number of channels. For example, in computer vision this relates to the RGB channels of an image. In our case it will be similar to a greyscale picture since we will only have one single channel. However, as we show further on, it is possible to extend our architecture to use multiple channels.

Thus, the final output of this layer is a $n \times d \times c$ matrix, where n and d have been established before and c is the number of channels.

Convolutional Layer

This layer receives the $n \times d \times c$ embedded matrix and applies a convolution operation.

The convolutional layer requires a set of k-kernels, each of these can be defined as a $h \times d \times f$ matrix. Each kernel is a group of filters where h is the height that corresponds to the number of sequential words in its receptive field, d is the length of the embedded vectors as defined before, and f is the number of filters contained in a kernel.

Kernels (k) can have different heights (h), thus all filters contained in a kernel have the same height. However, every kernel has the same number of filters (f) and all filters have the same length (d).

Now, each kernel will stride down the matrix applying a dot product operation with

every filter. This results in each $h \times d$ filter producing $n - h + 1$ nodes. These nodes form a feature map, where each node reports on how intensely a given feature is present.

As such, the output from this operation will be a $k \times (n - h + 1) \times 1 \times f$ matrix.

Using the example in Figure 3.3, we have 2 kernels with heights of 2 and 3 (h), a length of 10 (d) and 3 filters (f). The first kernel produces 3 feature maps (f), with a height of 8 (n - h + 1) and the second kernel also produces 3 feature maps (f), with a height of 7 (n - h + 1).

Max-over-time-pooling Layer

In order to reduce the computational effort and prevent overfitting, the max-over-time-pooling layer will select the maximum value from each feature map. This means that the resulting matrices from the convolutional layer will be reduced to $k \times f$ values, one value per filter.

These are concatenated, forming a tensor to be forwarded to the output layer. Considering 3.3 once more, as result of the convolution operation we have 2 sets of feature maps, with heights of 8 and 7 and both containing 3 feature maps. The max-over-time-pooling operation will take the maximum value from each feature map, leaving 3 nodes per set. Once concatenated, we will have a tensor of 6 values to be forwarded to the final layer.

Output Layer

Before performing a final computation and outputting a prediction, we apply a *dropout* function. This allows the network to generalise better, by randomly eliminating a fraction (according to a defined percentage) of nodes during the training process. Should be noted that the *dropout* function is only used in the training phase.

Finally, these values enter a fully connected NN and then a *softmax* function outputs the prediction, classifying if a tweet is (or not) mentioning a threat to the monitored ICT infrastructure.

3.5 Named Entity Recognition

Once a tweet is classified as relevant, we forward it to the NER model. The goal in this step is to extract some information from the tweet, such as the asset being targeted, the vulnerability being exploited and other relevant information.

In order to conduct such a task we have implemented a neural architecture for NER based on Lample et al. [45].

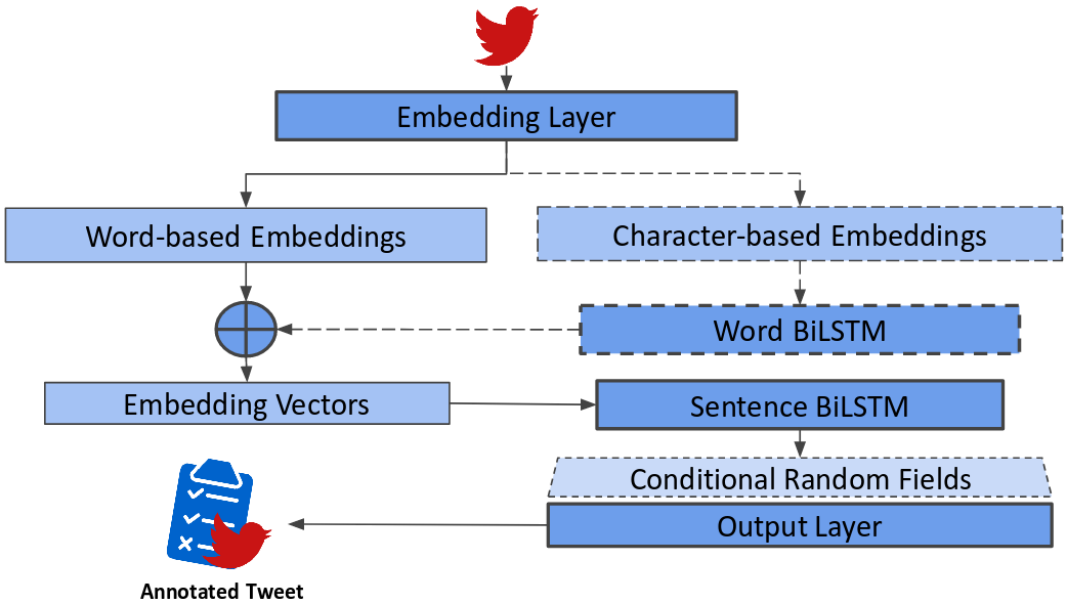

Figure 3.4: NER neural architecture for sequence tagging.

3.5.1 Bidirectional Long Short-Term Memory

Similarly to the way we described our CNN, we will explain this network by detailing its data flow, describing how the data is transformed during the feed-forward process, without going into the recurrent part of the model. The overview of the architecture is presented in Figure 3.4. To be noted that this architecture contains optional modules, presented as dotted boxes.

The network begins in the same way as the CNN, a tweet is received at the input layer and is sent to the embedding layer. The embedding layer is similar to the one from the CNN where each word is converted into an embedded word vector. Optionally characters can also be converted into embedded character vectors which are sent to a word BiLSTM layer. The results are concatenated with the embedded word vectors in order to enrich the representations. These numerical vectors are then sent to the sentence BiLSTM layer and finally to the output layer. In the output layer we have two ways of processing the prediction, we either use Conditional Random Fields (CRF) [44] or the traditional *softmax* function.

Input Layer

The network receives a tweet which can be represented as a sequence of n words.

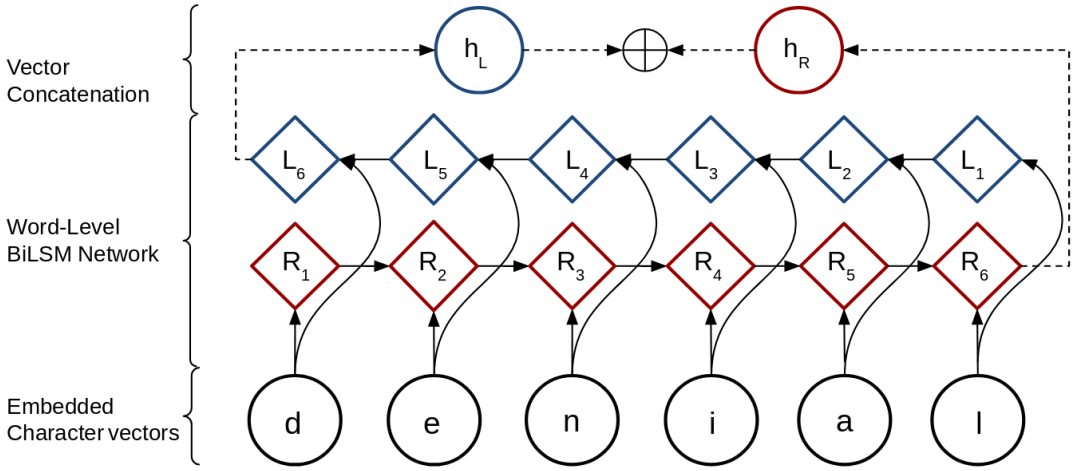

Figure 3.5: Word BiLSTM architecture.

Embedding Layer

The overall process is identical to the embedding layer from the CNN in terms of word-based embeddings. However, this layer can additionally produce embedded vectors for each character of a word, thus providing character-based embeddings.

Every word is converted into an embedded vector, resulting in a $n \times d_{word}$ matrix representation, where n is the number of words and d_{word} is the length of the embedded word vectors. Besides each word having a corresponding embedded word vector, it may optionally have a corresponding $n \times c \times d_{char}$ matrix of numerical vectors. Here, c is the number of characters in a word and d_{char} is the length of the embedded character vectors.

Word BiLSTM Layer

This is the first optional layer which is used when opting to use character-level embeddings in combination with the traditional word embeddings.

For each word n, the corresponding embedded character matrix $c \times d_{char}$ is fed to a BiLSTM network. This network contains two cells which read the input sequence of character vectors in opposite directions.

Both cells contain a hidden state h_s, this state is represented by a numerical vector which is updated at every time step (every character read). After reading every character in the embedded character matrix, the cell states are extracted and concatenated.

This process is exemplified in Figure 3.5. The network receives six vectors which constitute the word “denial”. These vectors are fed one by one to the LSTM cells, represented by the L and R diamonds. Once the final character is read by the cells we extract their hidden state vectors (h_R and h_L) and concatenate them. These vectors hold a character-level semantic representation from both left-to-right and right-to-left readings.

Regarding the data structure of the output, this layer outputs a $n \times (2 \times h_s)$ matrix of

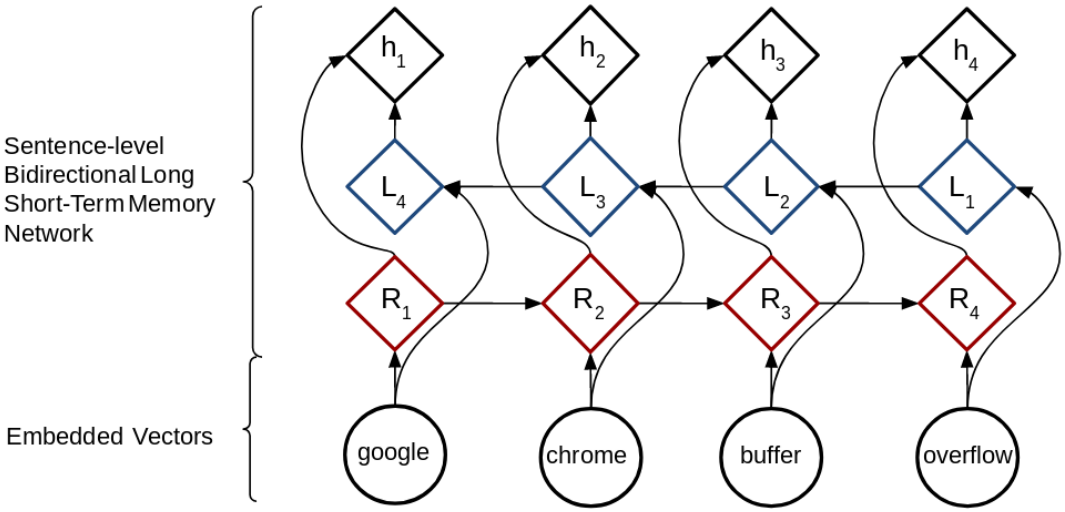

Figure 3.6: Sentence BiLSTM architecture and Conditional Random Fields.

vectors that are meant to be concatenated with the vectors from the $n \times d_{word}$ matrix.

Sentence BiLSTM Layer

The main BiLSTM network can either receive only the embedded word vectors from the embedding layer or receive both embedded word vectors and the resulting character-level representation vectors from the word BiLSTM layer. If the latter option is chosen, then these vectors are concatenated.

This leads to the input being a matrix of shape $n \times d$, where n is the number of words and d can either be equal to d_{word} in the case we only use word embeddings or $d_{word} + (2 \times h)$ in case we use both word and character embeddings.

Similar to the process described for the word BiLSTM layer, we feed the sentence to the sentence BiLSTM layer word by word. However, while in the previous BiLSTM layer we only retrieve the final hidden state, in this BiLSTM we extract the cell state at every timestep (every word read).

Thus, the output from this layer is a $n \times h_{sentence}$ matrix, $h_{sentence}$ is the length of the vector representing the BiLSTM's cell state.

Output Layer

In the final layer of the NER model, we have a fully-connected layer and two options from which the sequence prediction is going to be computed.

The fully-connected layer contains t neurons, with t being the number of possible labels that our model can predict. As an input for this layer we have the result from the previous sentence BiLSTM layer which is presented as a $n \times h_{sentence}$ matrix. We

perform the usual computational steps of a fully-connected neural network layer against every word, computing as a result a $n \times t$ matrix.

This matrix can be thought of as a score matrix. The t neurons have "looked" at each word, through the numerical values outputted in the sentence BiLSTM layer, and have given a score through its activations.

Then, we use this score matrix to compute the final sequence. As mentioned before, we have two options to perform this final computation.

The first option is a traditional softmax layer. For each word we have t neurons, using a Softmax function we can retrieve the label corresponding to the highest activated neuron. And thus, we output a sequence of n labels, one per each word. However, this solution has a certain limitation since each prediction only accounts for the t values from the word it aims to classify.

The second option is to use a conditional random fields layer. Instead of modelling labelling decisions independently, the CRF allows us to model them jointly. Given a sequence of words, a sequence of score vectors and a sequence of labels, a linear-chain CRF defines a global score to the sequence.

Lets consider as our sequence of words: $X = (x_1, x_2, x_3, ..., x_n)$, n being the number of words in the sentence. Since we intend to extract n labels, one for each x, let $y = (y_1, y_2, y_3, ..., y_n)$ be the labels found for a sentence of n words. Let our $n \times t$ matrix, outputted by the sentence BiLSTM layer, be represented by P, where $P_{i,j}$ would refer to the score of the j^{th} label for the i^{th} word.

In the CRF layer we will pick the sequence with the highest score, as such a prediction takes into account the whole sequence instead of independently computing the prediction for each word. A sequence's score is computed as follows [44, 45]:

$$s(X, y) = \sum_{i=0}^{n} A_{yi,yi+1} + \sum_{i=1}^{n} P_{i,yi}$$

A is a $t \times t$ transition score matrix, automatically learned during training, where $A_{i,j}$ contains the score of a label i being followed by a label j.

Finally, after computing the candidate sequences we apply a local softmax function and retrieve the label sequence with the highest score.

3.6 Final Remarks

In this chapter we have described the existing system in the context of the DiSIEM project, which we aim to improve. Besides this, we provide a clear problem statement and the architecture of a processing pipeline based on DL models developed in this work. Therefore, we explained in detail our NN architectures, including their hyperparameters and design variables which will be thoroughly explored in the following chapter.

Chapter 4

Experimental Evaluation

Having defined our architecture, this chapter reports on the experimentations regarding the architectural variations of both neural networks and the search performed in order to find an adequate set of NN design variables and hyperparameters.

The training algorithm used to optimize the models was the TensorFlow's implementation of the Adam Optimizer method [42]. Adam is an efficient stochastic optimization algorithm that only requires first-order gradients with little memory requirement.

4.1 Datasets

To evaluate our models we used data that has been collected and manually labelled in previous works[4, 13]. We used three datasets of tweets, one for training the models (**D1**) and the remaining two for testing and evaluation (**D2**, **D3**). The training dataset **D1** was collected in a time period of six moths across a set of Twitter accounts defined as **S1**. For the testing datasets, **D2** e **D3**, a second set of accounts **S2** was added. The testing set **D2** was collected for one month after the conclusion of **D1**'s collection, while the set **D3** was collected for two months after **D2**. Table 4.1 shows the most relevant information about these datasets, indicating the number of relevant (positive) and non-relevant (negative) tweets found.

Datasets	Time interval	Accounts	Positives	Negatives	Total
D1	01/11/2015 to 01/04/2016	S1	1697	2008	3705
D2	01/04/2016 to 15/05/2016	S1 + S2	536	4292	4828
D3	15/05/2016 to 10/07/2016	S1 + S2	1680	2153	3833

Table 4.1: Datasets used to train and test the models.

4.2 Training and Evaluation methodology

Regarding training and evaluation, we have adopted a 10-Fold Cross Validation procedure in order to increase the validity of the results obtained. This method consists in dividing the training set (**D1**) into 10 sub-sets of equal size and use one of these sub-sets for testing while using the remaining 9 for training. This training process occurs 10 times, alternating the sub-set used for testing. In each iteration, we also test the model against the testing sets (**D2**, **D3**). Finally, we average the testing results by dataset across all 10 folds.

In terms of plotting our results, we often show the results related to the testing set **D3** due to the date of its retrieval being more distant to the training set **D1**. Furthermore, in Section 4.5, the **D3** test set is not only separated from **D1** in terms of time but also contains a different set of accounts which make it a more robust testing set to evaluate our models.

4.3 Classifier

In this section we detail the experiments regarding our CNN classification model. We elaborate on the metrics that will be used for evaluation, list all the training parameters and hyperparameters of our model and finally report on all experimentations that we have done.

4.3.1 Evaluation Metrics

Before we begin reporting on our experiments it is necessary to declare the metrics through which we will evaluate the models performance.

For this task we sought to use the sensitivity (True Positive Rate TPR) and specificity (True Negative Rate TNR).

These metrics are calculated as follows:

$$TPR = \frac{TP}{TP + FN} \qquad TNR = \frac{TN}{TN + FP}$$

Where TP, TN, FP and FN represent, respectively, the tweets correctly classified as relevant, the tweets correctly classified as non-relevant, the tweets incorrectly classified as relevant and the tweets incorrectly classified as non-relevant.

A high TPR represents the model's capacity to correctly identify tweets containing relevant information. Given that we desire to retrieve as few non-relevant tweets as possible, we aim to obtain a high TNR which demonstrates the model's capacity to correctly identify non-relevant tweets.

Learning parameters	**Description**
learning rate	Value that controls how much the weights are adjusted depending on the loss gradient (Set to 0.01).
number of epochs	How many times do we feed the training data to the network (Set to 10).
batch size	Maximum number of tweets that the network receives at once (Set to 256).
Hyperparameters	**Description**
dropout probability	Probability to drop a neuron after the Max-over-time-Pooling Layer (Section 2.1.2).
l2 regularization lambda	L2 norm constrains on the weight vectors (Section 2.1.2).
Design variables	**Description**
number of kernels	Number of kernels to be used in the Convolutional Layer (Section 3.4.1).
kernel heights	Heights of the kernels to be used in the Convolutional Layer (Section 3.4.1).
number of filters	Number of filters in a kernel (Section 3.4.1).
word vector length	Number of dimensions for the word vectors (Section 3.4.1).
number of channels	Number of representations for the word vectors (Set to 1).
fully-connected layer	Structure of the final fully-connected network in the Output Layer (Section 3.4.1).
padding	Padding method, as described before this can be either static or dynamic (Section 3.3.2).
cleaning	Cleaning method, as described before this can be either full or partial (Section 3.3.1).

Table 4.2: List of hyperparameters and other configurable parameters.

4.3.2 Hyperparameters and model design variables

Here we lay out all the possible values that can be manually tweaked in order to fine-tune the models. Table 4.2 lists these hyperparameters and other training parameters that will be tuned in the experimentations that will follow.

4.3.3 CNN model design

Our first experimentation sought to explore the model variants proposed by Kim [41]. These variations are based on the architecture shown in Section 3.4 and are presented as follows:

1. **CNN-rand**: Uses randomly initialized word vectors which are modified during training.

2. **CNN-static**: Uses a pre-trained language model to initialize the word vectors which remain unchanged during training.

Models	D1 TPR / TNR	D2 TPR / TNR	D3 TPR / TNR
CNN-rand-128	0.94 / 0.96	0.88 / 0.97	0.85 / 0.95
CNN-rand-300	0.96 / 0.96	0.91 / 0.96	0.88 / 0.94
CNN-static	0.95 / 0.95	0.91 / 0.95	0.87 / 0.92
CNN-non-static	**0.95 / 0.96**	**0.92 / 0.98**	**0.89 / 0.95**
CNN-multichannel	0.93 / 0.97	0.88 / 0.98	0.84 / 0.96

Table 4.3: CNN architectural variations results.

3. **CNN-non-static**: Uses a pre-trained language model to initialize the word vectors and fine-tunes them during training.

4. **CNN-multichannel**: Uses two channels, two sets of word vectors, using a pre-trained language model to initialize both sets. However, only one is fine-tuned during training while the other one remains unchanged.

For this experimentation we used the same hyperparameters and design variables used by Branco [4]: three kernels of sizes 3, 4 and 5, with each kernel containing 128 filters, and a dropout probability of 50% in the output layer.

Another design variable is the word vector length, originally Branco only used the CNN-rand variant and set the vector length to 128. For comparison sake, we trained two CNN-rand variants, one with a vector length of 128 and the other one with a length of 300 (these variants will be labelled as CNN-rand-128 and CNN-rand-300). We have set it to 300 in order to be equal to the 300-dimensional vectors from the pre-trained Google News' Word2Vec [26, 49, 50] language model which we use for the pre-trained language model.

Regarding the pre-processing, we used the *full* cleaning option and *static* padding. Furthermore, all models were trained for 10 epochs, this value was chosen with no specific criteria other than a few trials.

The results are presented in Table 4.3. We can see a slight benefit in TPR when increasing the word vector length as shown in the results from the CNN-rand variants.

Overall, the variant that showed the best results appeared to be the CNN-non-static and as such we opted to use this architecture for future experimentation.

4.3.4 Grid Search

One of the foremost objectives when planning this dissertation was the necessity to find an appropriate CNN model design. This design was made by means of two grid search experiments using the evaluation metrics described.

Grid search is a common technique of hyperparameter optimization. Essentially it works as a parameter sweep or an exhaustive search through several subsets of values the

hyperparameters can take.

First Grid Search

In the first grid search we performed, we sought study the key design variables and hyperparameters of a CNN which we considered to be the *number of kernels*, *kernel height*, *number of filters* and *dropout probability*. This exploration resulted in testing a total of 1945 models, whose design variables and hyperparameters were varied as described:

- **Number of kernels**: varied from 2 to 34;
- **Kernel heights**: to understand the advantages of using smaller or larger filter heights, 3 cases were considered, small, medium and large, where the heights depended on the number of kernels:
  - For the small case, if the number of kernels was 2 the heights would be [2, 3], if the number of kernels was 3, the heights would be [2, 3, 4], and so on: [2, 3, 4, 5], [2, 3, 4, 5, 6], ... [2, 3, ..., 35];
  - For the medium case, if the number of kernels was 2 the heights would be [18, 19], if the number of kernels was 3 the heights would be [17, 18, 19], if the number of kernels was 4 the heights would be [17, 18, 19, 20], and so on: [16, 17, 18, 19, 20], [16, 17, 18, 19, 20 ,21], ..., [6, 7, 8, ..., 28, 29, 30];
  - For the large case, if the number of kernels was 2 the heights would be [34, 35], if the number of kernels was 3 the heights would be [33, 34, 35], and so on: [32, 33, 34, 35], ..., [11, 12, ..., 34, 35].
- **Number of Filters**: varied within the following set: 8, 16, 32, 64, 96, 128, 192, 256;
- **Dropout Rate**: varied within the following set: 0.33, 0.5, 0.66.

We summarize our findings in regards to how each of these design variables and hyperparameters influences the model's performance. Regarding the TNR, there were no conclusive results as to what could benefit this metric without decreasing the TPR, thus we base our conclusions on the TPR analysis.

Regarding the kernel heights, the small case tends to have better results. This is shown by the boxplot in Figure 4.1.

Although the number of kernels and number of filters do not appear to have a direct impact in the models performance, if we consider their combination, meaning the number of features, we may have a current cut-off, in terms of number of features, for which the model begins to overfit. The number of features is calculated through the product of the number of kernels and the number of filters. If we recall Section 3.4, this number relates

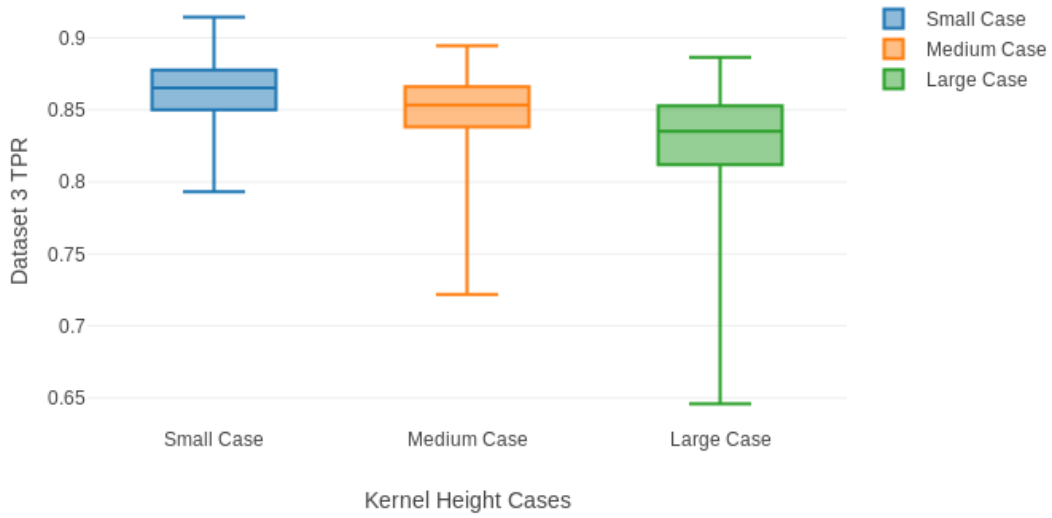

Figure 4.1: Box plot of the regarding TPR of the test set D3 and the kernel heights.

to the nodes after the max-over-time-pooling layer that are concatenated and sent to the final layer. As shown in Figure 4.3 the models appear to start decreasing in regards to their TPR beyond the 1500 features.

The dropout rate appeared to have no obvious impact in the model's performance as shown in Figure 4.2. Nonetheless, the model that achieved the best TPR while keeping an acceptable TNR used a dropout rate of 0.5.

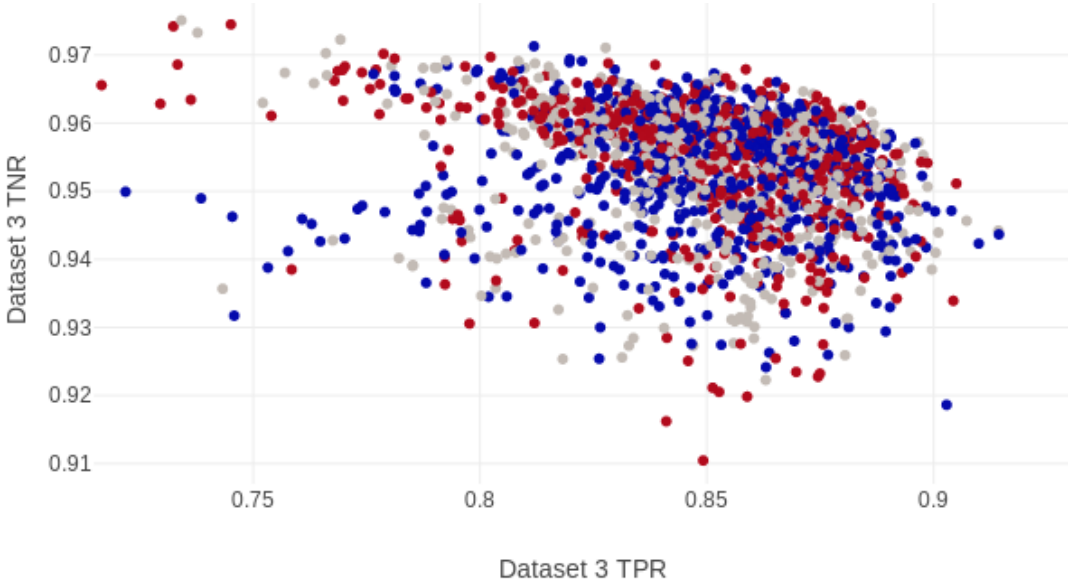

Figure 4.2: Scatter plot of the TNR and TPR of the testing set **D3**. The blue dots represent a dropout rate of 0.33, the grey dots represent 0.5 dropout rate, and the red dots represent a dropout rate of 0.66.

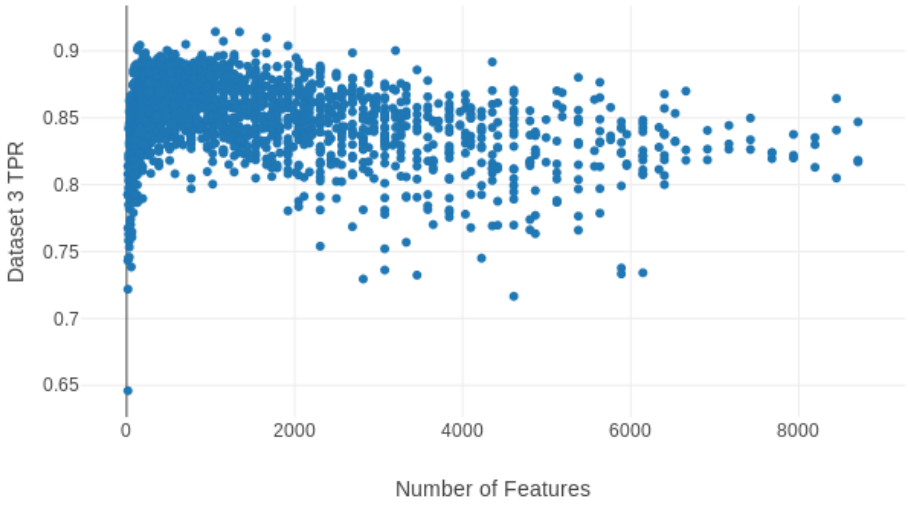

Figure 4.3: Scatter plot relating the TPR of the test set D3 and the number of features.

Second Grid Search

After the first grid search, we sought to evaluate additional design variables and hyperparameters.

For this second grid search we experimented with the *padding* methods, the *cleaning* methods, and the *L2 regularization lambda* value. We performed further adjustments to the *number of kernels* and their corresponding *kernel heights*, the structure of the *fully-connected layer*, and we experimented with different pre-trained language models for initializing the *embedding vectors*.

Just as before, we list how these variables were set:

- **Padding**: either *static* or *dynamic*;
- **Cleaning**: either *partial* or *full*;
- **L2 Regularization Lambda**: either 0 or 3;
- **Number of Kernels** and **Kernel Heights**:varied the number of kernels between 3 and 6. Kernel heights were varied incrementally either in a normal sequential manner (e.g., 2,3,4) or by parity (e.g., 'odd' : 3,5,7 or 'even' : 2,4,6);
- **Fully-connected Layer**: we tested the option of inserting an additional hidden layer at the fully-connected network; For this new layer we considered its neurons to be 128 or 256 nodes;

- **Embedding Vectors**: we tested another well-known language embedding model GloVE [56] which provided three sets of pre-trained vectors trained using different sources.

Regarding the padding and cleaning options, these showed to be irrelevant to the model's performance. However, it should be noted that the options *dynamic padding* and *partial cleaning* greatly reduce the amount of time a model takes to train.

The L2 regularization appears to generally improve models performance as shown in Figure 4.4. However, it should be noted that the best performing models did not use L2 regularization as can be observed in Table 4.4.

As per the number of kernels and kernel heights, although this experimentation provided improved results, we could not extract any tangible knowledge as to why these settings would perform better.

The addition of a hidden layer proved to be inconclusive as the models showed a significantly higher standard deviation in their results than the models that did not add the new hidden layer. Furthermore, in Table 4.4 we can see that 2 out of the 10 best performing models used an additional layer.

Finally, in the experimentations with the different pre-trained word vectors no model appeared be consistently better even though in Table 4.4 most models used the GloVE language model.

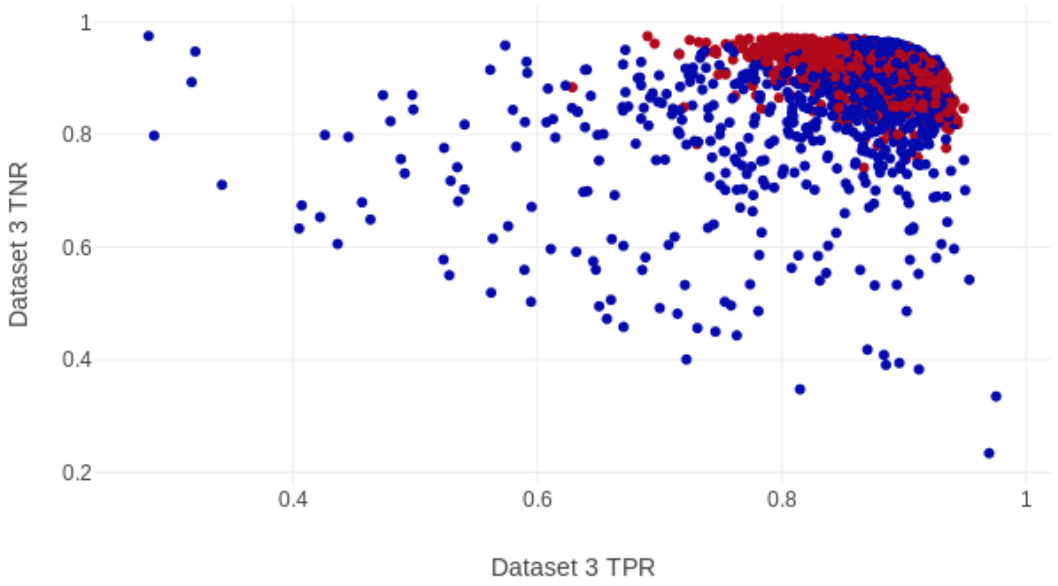

Figure 4.4: Scatter plot of the TPR and TNR of the testing set D3. The blue dots represent models that did not suffer L2 loss regularization (lambda = 0) , whereas the red dots have a L2 lambda of 3.0.

Model	L2 Lambda	Kernels	Number of Filters	Fully connected Layer	Embedding Model	D3		
						Accuracy	TPR	TNR
1	0	[3,5,7,9,11]	256	NA	GloVE	0.93 ± 0.01	0.90 ± 0.02	0.96 ± 0.02
2	0	[2,4,6]	128	NA	GloVE	0.93 ± 0.01	0.91 ± 0.02	0.95 ± 0.03
3	0	[2,4,6]	128	256	Word2Vec	0.93 ± 0.03	0.91 ± 0.05	0.95 ± 0.04
4	0	[2,4,6]	128	NA	GloVE	0.93 ± 0.00	0.90 ± 0.01	0.96 ± 0.02
5	0	[2,4,6]	256	NA	GloVE	0.93 ± 0.01	0.90 ± 0.02	0.96 ± 0.03
6	3	[3,5,7,9]	128	256	Word2Vec	0.93 ± 0.02	0.90 ± 0.04	0.95 ± 0.04
7	0	[3,5,7,9]	128	NA	GloVE	0.93 ± 0.01	0.91 ± 0.02	0.95 ± 0.02
8	0	[2,4,6,8]	256	NA	GloVE	0.93 ± 0.00	0.91 ± 0.02	0.95 ± 0.02
9	0	[2,4,6,8]	256	NA	Word2Vec	0.93 ± 0.01	0.89 ± 0.02	0.96 ± 0.02
10	0	[2,4,6]	256	NA	GloVE	0.93 ± 0.01	0.89 ± 0.02	0.96 ± 0.02

Table 4.4: 10 best performing models in the second grid search, ordered by **D3** accuracy results.

Classifier Comparison

Having completed our extensive hyperparameter and design variable experimentations, we present the best performing model in Table 4.5. The model was selected as the best performing model according to its **D3** accuracy.

The comparison between the best performing model with the SVM currently deployed is displayed in Figure 4.5. As we can see, the CNN model shows improvement on both metrics. Furthermore, the model presents a much better TNR, which in practice means there are fewer false positives which was also a declared objective of this work. This is important because a system that has a considerable amount of false positives will seem untrustworthy, making it more likely for a true positive to pass unnoticed by an analyst.

Hyperameters	**Value**
dropout probability	0.5
L2 regularization lambda	0.0
Design variables	**Value**
number of kernels	5
kernel heights	3, 5, 7, 9, 11
word vector length	300
embedding model	GloVE
number of channels	1
fully-connected layer	Not used
padding	dynamic
cleaning	partial
Learning parameters	**Description**
learning rate	0.01
number of epochs	10
batch size	256

Table 4.5: List of hyperparameters, design variables and learning parameters used for the best CNN classifier model.

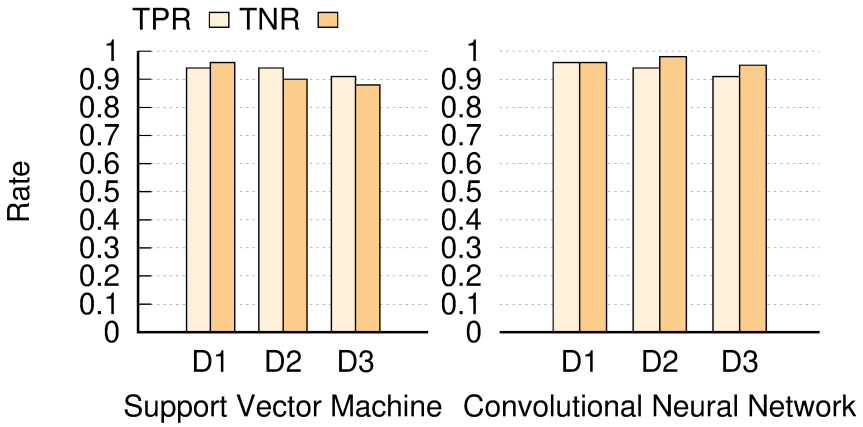

Figure 4.5: Comparison of the results of the Support Vector Machine and the best Convolutional Neural Network.

4.4 Named Entity Recognition

This section describes the experimentation that we have done with our NER classifier based on a BiLSTM network.

Similarly to what was done for the CNN classifier, we define our metrics of evaluation, list the training parameters and the hyperparameters, evaluate the optional modules that constitute the network and finally compare the obtained results.

4.4.1 Datasets

Unlike the classification task, we did not have access to prepared datasets to conduct our research on NER. For this purpose, we took the positive tweets from the **D1**, **D2** and **D3** and manually labelled each word.

Table 4.6 displays the labels that have been considered. We have partially based our entities on descriptions from the ENISA risk management glossary [21]. Given that this is a first approach and we have limited data for training, we have left the label descriptions broad enough so that the network is able to distinguish and learn to identify these entities in a given corpus of text. Once more training data is available, the number of entities could be extended, allowing for a more accurate taxonomy. For example, the label `VUL` includes both vulnerabilities and threats, this is mostly due to the small pool of data that we have available and partially due to Twitter being an informal source of data, meaning the majority of users do not follow any international standard when referring to security terms.

In Table 4.7 we present the number of labels in each dataset. Finally, we display in Table 4.8 an example of how the output from this model is expected to be, given the established labels.

Label	Description
O	Does not contain useful information.
ORG	Company or organization.
PRO	A product or asset.
VER	A version number, possibly from the identified asset or product.
VUL	May be referencing the existence of a threat or a vulnerability.
ID	An identifier, either from a public vulnerability repository (e.g., NVD) or from an update or patch.

Table 4.6: Named entities to be extracted from a tweet.

		Labels						
Datasets	**Nº of Tweets**	**O**	**ORG**	**PRO**	**VER**	**VUL**	**ID**	**Total**
D1	1697	9104	1205	3322	1147	4420	607	19805
D2	536	2984	497	1117	304	964	213	6079
D3	1680	9238	1612	3456	1041	2791	774	18912

Table 4.7: Number of Labels per Dataset.

4.4.2 Evaluation Metrics

In order to evaluate our models, we require a metric just as we defined for the CNN classifier. For this task, we have considered Precision, Recall, F_1 measure, and accuracy. Precision and Recall, P and R respectively, be calculated as follows:

$$P = \frac{TP}{TP + FP} \qquad\qquad R = \frac{TP}{TP + FN}$$

For sake of simplicity when comparing models, we decided use the F_1 measure and the accuracy. In order to do this we averaged the precision and recall across all labels and then computed F_1 as follows:

$$F_1 = 2 \times \frac{P \times R}{P + R}$$

Finally, we also considered the overall accuracy, which is simply the division between the number of correctly labelled words and the total number of words.

We compute these metrics for all labels. For example, considering label O, TP is the number of words correctly classified as O, FP is the number of words incorrectly classified as O, and FN the number of words that were O but were not correctly classified.

4.4.3 BiLSTM design variables

The BiLSTM model has several design variables that have to be specified and experimented with. Most importantly, this architecture has a set of optional modules that are

Tweet	vuln oracle java se cve-2015-2625 remote security vulnerability
Labelled Tweet	O ORG PRO PRO ID VUL VUL VUL

Table 4.8: Example of a input tweet and the output expected from the NER model.

able to provide new functionalities to the network as detailed in Section 3.5.1. We list all of the design aspects that will be object of study in Table 4.9.

Learning parameters	**Description**
learning rate	Value that controls how much the weights are adjusted depending on the loss gradient. (Set to 0.01) .
number of epochs	How many times do we feed the training data to the network. (Set to 10).
batch size	Maximum number of tweets that the network receives at once during training. (Set to 256).
Design variables	**Description**
pre-trained language model	Initialization method of the embedded word vectors (Section 3.5.1).
Use CRF	Boolean value, indicating if we use the Conditional Random Fields Layer (Section 3.5.1).
Use Characters	Boolean value, indicating if we use the character-level embeddings.
word vector length	Length for the embedded word vectors. Note: If using pre-trained vectors, this value will default to the pre-trained vectors' length (Section 3.5.1).
character vector dimension	Length for the embedded character vectors (Section 3.5.1).
word BiLSTM cell state	Length for the hidden cell state of the Word BiLSTM (Section 3.5.1).
sentence BiLSTM cell state	Length for the hidden cell state of the Sentence BiLSTM (Section 3.5.1).

Table 4.9: List of design variables for the BiLSTM network.

4.4.4 Grid Search

Similar to the process conducted for the CNN binary classifier, we sought to explore different configurations of the BiLSTM network for NER.

Architectural Variations

One of the main objects of interest of this our experiments is to evaluate the options of using character-level embeddings and the CRF. For such task, we define four architectures:

- **BiLSTM**: Baseline model, does not use character-level embeddings nor the CRF layer.
- **BiLSTM-Char**: Uses character-level embeddings, but does not use the CRF layer.
- **BiLSTM-CRF**: Uses the CRF layer, but does not use character-level embeddings.
- **BiLSTM-Char-CRF**: Uses both character-level embeddings and the CRF layer.

Design variables

For this section of our testing efforts, we focused on adjusting the main design variables of the BiLSTM network within the following ranges:

- **Word vector length**: varied within the following set: 100, 200, 300 for the random initialization cases.
- **Character vector length**: varied within the following set: 25,50,100
- **Word BiLSTM hidden state**: varied within the following set: 25,50,100
- **Sentence BiLSTM hidden state**: varied within the following set: 100, 200, 300
- **Embedding vectors Initialization**: we tested the same pre-trained language models that were used in the previous experimentations.

Unlike the grid searches performed for the CNN that used only one architecture (CNN-non-static), we performed the above variations over all architectures of the BiLSTM NER model.

Results

Here we report on our findings, after completing the grid search for the BiLSTM network. Regarding the embedding vectors initialization, we found that for the NER model the pre-trained vectors did not offer any improvement over the randomly initialization option. However, the word vector length appeared to be an important design variable since the majority of the 10 models with the highest F_1 results had their word vector length set to 300. Although models seemed to benefit from larger word vectors, the length of the character vectors did not display the same relation. Furthermore, the word BiLSTM hidden state did not seem to favour any specific value while the sentence BiLSTM hidden state appeared to favour a small hidden state (100). Table 4.10 presents the 10 models that achieved the highest F_1 values in regards to the testing set **D3**.

Besides this hyperparameter experimentation that we have reported on, our main goal was to analyse how the optional modules of the network affected the NER classifier's performance. The scatter plots of Figures 4.6 and 4.7 show the relation between the

Model	Embedding Model	Word vector length	Character vector length	Word hidden state	Sentence hidden state	D3 F_1
1	None	300	50	100	100	0.92193
2	GloVE	300	100	100	100	0.92094
3	None	200	50	50	100	0.92023
4	None	300	100	25	300	0.91995
5	None	300	100	50	100	0.91965
6	None	200	25	100	100	0.91935
7	None	200	100	25	100	0.91918
8	None	300	50	50	200	0.91897
9	GloVE	300	100	25	100	0.91893
10	None	300	25	50	200	0.91877

Table 4.10: 10 best performing models in the BiLSTM grid search, ordered by **D3** F_1 results.

accuracy and F_1 metrics of the test sets and the four different architectures that our model can have.

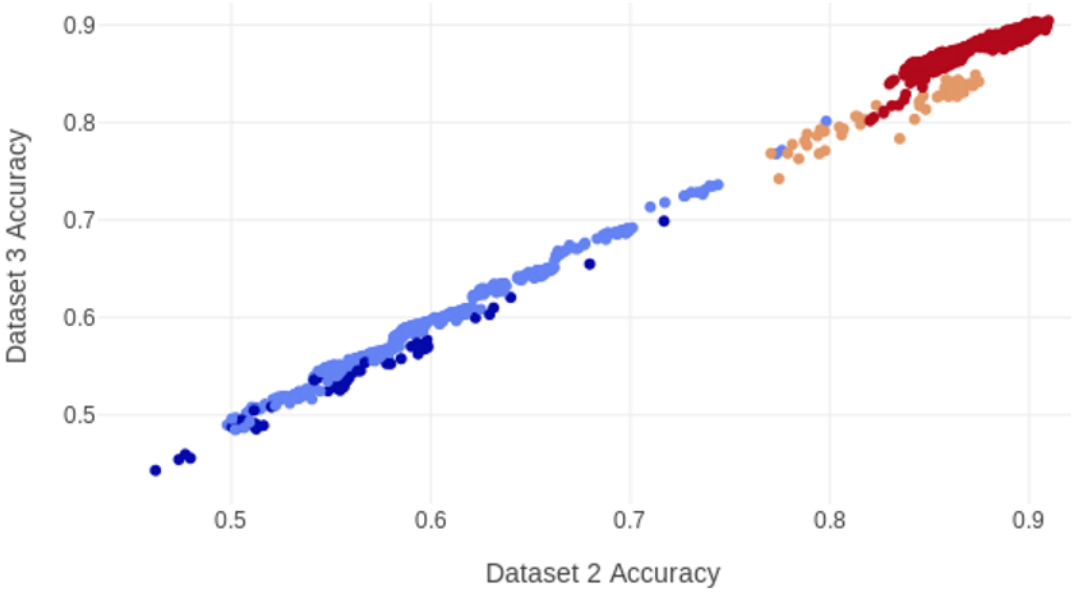

Figure 4.6: Scatter plot of the Models' accuracy in the test sets D2 and D3. The darker blue dots identify the BiLSTM baseline models, the lighter blue dots represent the BiLSTM-Char variants, the orange dots represent the BiLSTM-CRF variants and finally the red dots identify the BiLSTM-Char-CRF variants.

Both plots illustrate the BiLSTM baseline model as generally performing worse than the variants. On the other hand, the BiLSTM-Char-CRF variant appears to generally outperform all other variants and the baseline at both the F_1 and accuracy measures.

An interesting aspect is that although the BiLSTM-CRF variant performs better in the accuracy measure than the BiLSTM-Char, the latter performs better than the former in the F_1 measure.

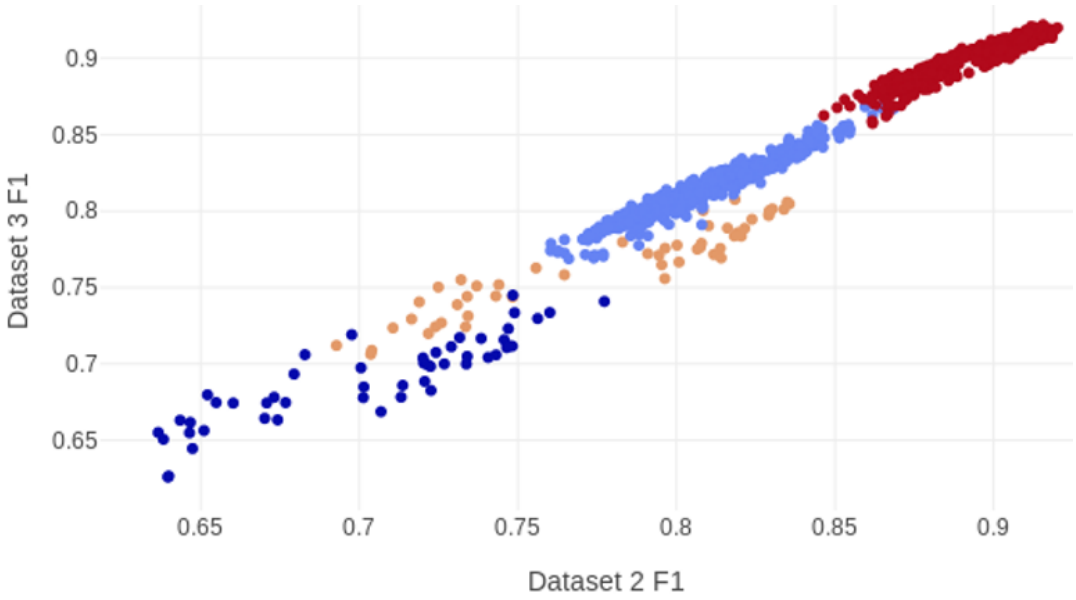

Figure 4.7: Scatter plot of the Models' F_1 score in the test sets D2 and D3. The darker blue dots identify the BiLSTM baseline models, the lighter blue dots represent the BiLSTM-Char variants, the orange dots represent the BiLSTM-CRF variants and finally the red dots identify the BiLSTM-Char-CRF variants.

We picked the best performing models from each variant and plotted their results in Figure 4.8. Besides the observations that we have made before about the variants and their performance, which can also be identified in this image, we noticed a second property of the usage of character-level embeddings. The two architectures that do not use character vectors (BiLSTM and BiLSTM-CRF) appear to slowly decay their performance over the datasets, from **D1** to **D3**.

On the opposite side, the models that use character-level embeddings (BiLSTM-Char and BiLSTM-Char-CRF) appear to be much more stable with an almost unnoticeable decrease between the training set and the testing sets. This may be due to the testing sets containing unknown words, meaning words that never appeared in the training set. In the case where the variants do not take advantage of character-level representations the word will only be represented by the `<UNK>` token's corresponding vector. Meanwhile, the variants that take advantage of these character-level representations will have the unknown words being represented by the `<UNK>` token's vector and the concatenated vector representation from the word BiLSTM layer which may be crucial to identify derivatives of certain words.

Finally, Table 4.11 displays the configuration of the best performing NER model which is labelled as BiLSTM-Char-CRF in Figure 4.8.

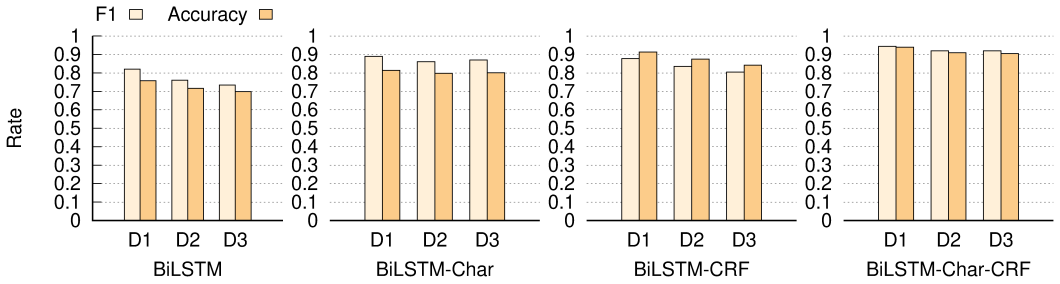

Figure 4.8: Bar plot of the best BiLSTM NER models from each variant.

Design variables	**Value**
word vector length	300
character vector length	50
word BiLSTM cell state	100
sentence BiLSTM cell state	100
embedding model	GloVE
use CRF	True
use characters	True
Learning parameters	**Description**
learning rate	0.01
number of epochs	5
batch size	256

Table 4.11: List of design variables and learning parameters used for the best NER model.

4.5 Case Study

This section shows the real-world application of the work that has been developed in this dissertation. As part of the H2020 DiSIEM project [18], we have taken a sample of the infrastructures from partners of the project to demonstrate the pratical application of our solution.

Each infrastructure is defined by a set of assets, which in turn are defined by a variable number of keywords. Table 4.12 presents the three infrastructures, denoted **A**, **B** and **C**, by showing the asset name and the keywords used.

We begin by describing the datasets that we have collected, then we display the results we have obtained by using the best architectures from the previous section and training these networks on the case study datasets.

A		B		C	
Assets	**Keywords**	**Assets**	**Keywords**	**Assets**	**Keywords**
CentOS	centos	Apache Web Server	apache http	Windows 7	windows, windows 7, win 7, win7
Ubuntu	ubuntu	Apache Tomcat	apache tomcat, tomcat	Windows 8.1	windows 8.1, win8.1, win 8.1
Debian	debian	Apache Struts	apache struts, struts	Windows 10	windows 10, win10, win 10
WildFly runtime	wildfly	Apache CouchDB	apache couchdb, couchdb	Windows 2003	windows 2003, win 2003, win2003
Apache Struts	apache struts, struts	Oracle WebLogic	web logic	Windows 2008	windows 2008, win 2008, win2008
Apache Storm	apache storm, storm	iPlanet	iplanet	Windows 2012	windows 2012, win 2012, win2012
Apache Zookeeper	apache zookeeper, zookeeper	Java	java	Windows 2016	windows 2016, win 2016, win2016
MySQL DBSM	mysql	Red Hat Enterprise Linux	red hat	Windows XP	windows xp, win xp, winxp
MongoDB	mongodb	CentOS	centos	Windows NT	windows nt, win nt, winnt
Elasticsearch engine	elasticsearch	SuseLinux	suselinux suse	Windows 2000	windows 2000, win 2000, win2000
Nessus	nessus	Oracle	oracle	IBM AIX	aix
OpenVAS	openvas	Microsoft	microsoft, ms	Red Hat Enterprise Linux	rhel, red hat, redhat
HIDS IDS	hids	SQL	sql	SCADA systems	scada
OSSEC IDS	ossec	MongoDB	mongodb	IE browser	internet explorer, iexplorer
Snort IDS/IPS	snort	JBoss App server	jboss	Chrome browser	google chrome, chrome
Suricata	suricata	OpenStack software	openstack	MS Office suite	microsoft office, ms office, office, word, excel, powerpoint
Bro network security monitor	bro	Red Hat Openshift	openshift	SAP software	sap gui, sap
Linux	linux	Docker Platform	docker	MS Visio	visio
		VMware vSphere	vsphere	.NET framework	.net
		Angular JS framework	angularjs	Adobe Acrobat reader	acrobat reader
		Selenium browser automation	selenium	Cute PDF writer	cute pdf writer
		MapR AI and Analytics	mapr	Apple Quicktime	quicktime
		Apache MapReduce	map-reduce, map reduce	Java	java jre
		MariaDB DBMS	mariadb	MS Silverlight	silverlight
		Drools BRMS	drools	Adobe Shockwave	shockwave flash
		Elastic Stack	elastic stack	McAfee AV	mcafee
		Apache Kafka	kafka	Linux	linux
		Citrix software	citrix		
		Splunk platform	splunk		
		SQLite library	sqlite		
		IDM MQ	ibm mq		
		Linux	linux		

Table 4.12: Assets and corresponding keywords used to extract tweets related to the ICT infrastructures A, B and C.

4.5.1 Datasets

We have used the same system to collect and label tweets over a period of 4 months for the 3 different ICT infrastructures described in Table 4.12.

Similarly to the process conducted in the previous chapter, we have split these datasets into 3 sub-sets. These sub-sets have been seperated by time interval and set of monitored accounts as shown in Table 4.13.

Datasets	Time Interval	Accounts	Positives	Negatives	Total
A1	21/11/2016 to 27/01/2017	S1	1074	694	1768
B1			1201	638	1839
C1			1293	1473	2766
A2	27/01/2017 to 27/02/2017		282	767	1049
B2			387	671	1058
C2			325	592	917
A3	27/02/2017 to 27/03/2017	S1 + S2	219	313	532
B3			250	247	497
C3			289	358	647

Table 4.13: Datasets corresponding to companies A, B and C.

4.5.2 Classifier

Using the architecture that achieved the best results, with the hyperparameter and design variables shown previously in Table 4.5, we trained three models, each with its corresponding training set (**A1**, **B1** or **C1**) and tested them against the corresponding testing sets (**A2** and **A3**, **B2** and **B3**, **C2** and **C3**). The results are presented in Figure 4.9.

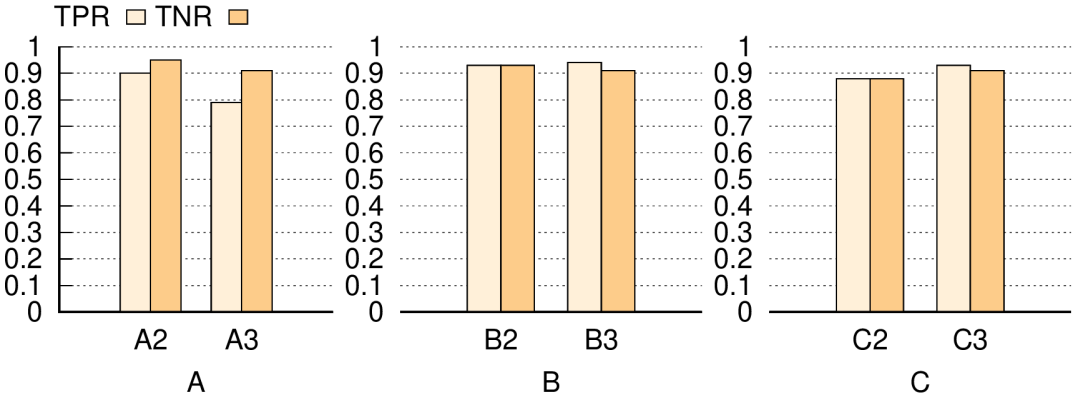

Figure 4.9: Bar plot showing the results of 3 models, trained on 3 different sets of training data and evaluated against their corresponding testing sets.

All models achieved slightly worse results when compared to the results obtained with the test datasets used before. This can be due to the limited amount of data available,

as well as to the sub-optimality of the architecture to the new datasets. This could be circumvented by modifying the design variables and build a model which is able to better fit the data. Thus, similarly to what was done previously to find the best architecture, we performed a grid search where we varied the following variables:

- **Kernels**: varied the number of kernels between 3 and 6. Kernel heights were varied incrementally either in a normal sequential manner (e.g., 2,3,4) or by parity (e.g., 'odd' : 3,5,7 or 'even' : 2,4,6);
- **Number of Filters**: varied within the following set: 64, 128, 192, 256;
- **Fully-connected Layer**: either None (no additional layer), 128 or 256;
- **Embedding Model**: either None (vectors are randomly initialized), Word2Vec or GloVE;
- **L2 Regularization Lambda**: either 0 or 3.

Figure 4.10 presents the results of the best models that were found in this grid search. Generally, we can see an improvement across all classifiers for TPR, with little modification to the TNR. Table 4.14 displays the architectures of all best models, including the experimental set and the case study datasets.

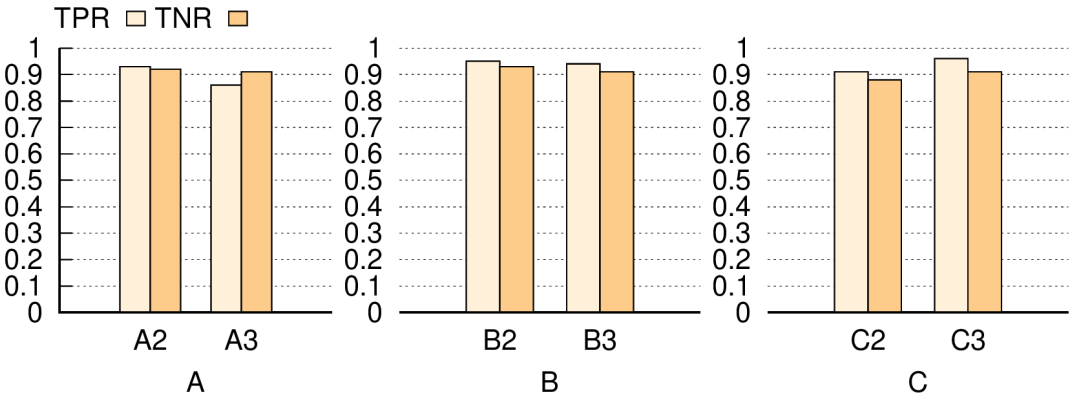

Figure 4.10: Bar plot showing the best resulting models obtained through Grid Search and 10-fold Cross Validation training.

Dataset	Kernels	Number of Filters	L2 lambda	Fully-connected Layer	Embedding Model
Experimental	[3,5,7,9,11]	256	0.0	None	GloVE
A	[3,5,7]	64	0.0	None	Word2Vec
B	[3,5,7]	256	3.0	None	GloVE
C	[3,5,7,9]	256	0.0	None	Word2Vec

Table 4.14: Architectures of all the best performing models of each dataset.

The models from the case study datasets have fewer parameters compared to the model from the experimental dataset. Since the case study datasets have less training data than the experimental dataset, the architecture that best performed in the experimental grid search may in fact overfit when used to train smaller datasets. And as such, we can see that in the case of the infrastructure **A**, the best model used fewer kernels and fewer filters than the experimental architecture.

4.5.3 Named Entity Recognition

We tested our NER model through the same process of picking the best performing architecture and using it to train a model for each dataset in the case study. The configuration used for the training of these models is displayed in Table 4.11.

In order to conduct such task, we had to manually label these datasets. The number of labels per dataset is displayed in Table 4.15.

	Labels						
Datasets	**O**	**ORG**	**PRO**	**VER**	**VUL**	**ID**	**TOTAL**
A1	3006	149	1364	479	1280	529	6807
B1	4855	478	2292	567	2465	974	11631
C1	7305	1367	2737	1408	2755	851	16423
A2	1531	36	554	208	388	207	2924
B2	2080	144	778	285	540	265	4092
C2	1957	231	522	199	487	131	3527
A3	1087	38	486	119	496	135	2361
B3	1134	77	591	96	535	194	2627
C3	1387	286	437	145	604	217	3076

Table 4.15: Distribution of labels per Dataset.

The results are presented in Figure 4.11. In comparison to the results obtained with the experimental datasets, these models display equally good results, capable of identifying around 90% of valuable information in a tweet.

However, as discussed before, these datasets are smaller than the original datasets used for the model evaluation. With an increment in the size of these datasets we could extract more accurate and reliable results. Nonetheless, these results are sufficient in order to show that our approach to extract security related information from a tweet may be in practice a valuable addition to the Twitter threat detector framework's capabilities.

4.6 Discussion

Taking into account all the experimentation conducted in this chapter, we ought to assess if our results are in line with the main objectives of this work.

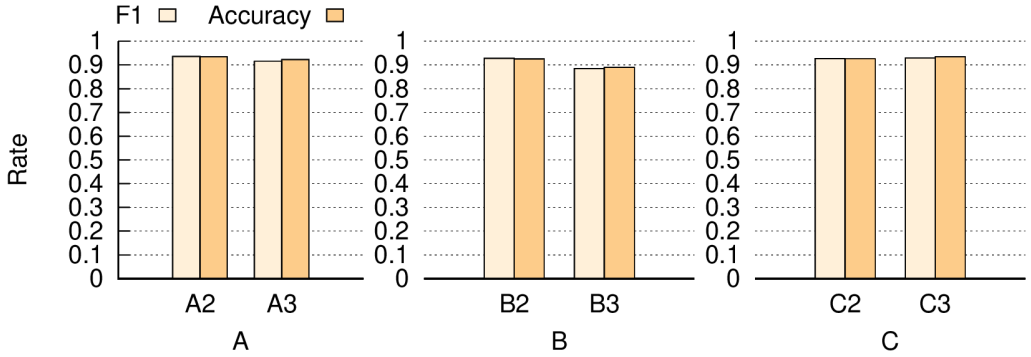

Figure 4.11: Bar plot showing the results of the 3 NER models, trained on 3 different sets of training data and evaluated against their corresponding testing sets.

First, we sought to improve the binary classifier's performance through a DL technique known as a CNN. As shown in Figure 4.5, our model either matched or surpassed the current SVM classifier's performance in regards to both TPR and TNR. Furthermore, the latter metric showed a significant improvement, confirming that our binary classifier substantially reduces the number of false positives which is an important objective of the work. In regards to the architecture exploration, including both the hyperparameters and other design variables, we believe that we have conducted a thorough search of the CNN capabilities for this task. We have observed that some settings appear to be optimal for these models, such as the usage of a pre-trained language model to initialize the word vectors and the usage of small sized filters. Depending on the data available, it is important to adjust some design variables in order to obtain a model that achieves satisfactory results. The most important design variables to adjust are the number of kernels, their distinct heights and the number of filters. This is supported by the results of Figures 4.9 and 4.10. The first figure illustrates the results obtained using the same architecture to train three different models, one for each infrastructure. This architecture's design variables are the result from the various experimentation conducted throughout Section 4.3. Additionally, the second figure shows the results of the best models found through an individual grid search for each of the case study datasets. As observed, the results from the latter models generally show an improvement. This leaves us to conclude that some form of model optimization process (e.g., grid search) should be conducted whenever the datasets are expanded or whenever the architecture is being applied to a different dataset.

Regarding the expansion of the Twitter threat detector framework's current capabilities, we explored the possibility of using a NER model to extract relevant information from a pool of relevant tweets. For such task, we required datasets to be manually labelled. Due to the limited amount of data, we had to limit the specificity of these labels and kept a general description for each of them in order to have a reasonable amount of entities to train from. Overall, the results were highly favourable, being over 90% in both evaluation metrics (accuracy, F_1 score). In the architectural exploration we tested the us-

age of optional modules such as the addition of character-level representations and a CRF Layer. According to Figure 4.8, besides both of these modules improving the model's performance, the character-level representations appear to provide stability across the training and testing sets. Although we did not conduct a grid search in the case study, the results were satisfactory enough in order to justify the efficiency of this architecture to extract information from a corpus of text.

Chapter 5

Conclusion

This work explored the implementation of DL algorithms to improve and extend the capabilities of a Twitter-based cyberthreat detection framework. We implemented a CNN binary classifier to provide an alternative to the current SVM binary classifier which aims to identify tweets containing relevant information regarding the cybersecurity of pre-determined assets of an ICT infrastructure. A second major contribution from this work was a BiLSTM neural network model for NER. This model is capable of locating and identifying relevant chunks of security related information from a corpus of text (e.g., a tweet).

We experimented with the architectural components of both networks, their hyper-parameters, and design variables in order to extract knowledge from their relations and to justify the final configurations for our models. We found that in regards to the CNN classifier, the models usually saw better results when using pre-trained word vectors and small heights for the kernels. Other design variables, such as the number of kernels and filters, appear to be heavily related to the amount and the quality of training data available. Although the usage of pre-trained vectors did not seem to affect theBiLSTM NER models' results, their optional architectural modules appeared to be the most important setting to be configured. We concluded that the usage of character-level representations and the addition of the CRF Layer improved a model's performance.

Both our contributions were successful in regards to the established goals. Regaring the experimental datasets, the CNN classifier shows better results in comparison with the SVM, specially in regards to the number of false positives which have substantially decreased. This allows to reduce the amount of irrelevant information that would have been presented to an analyst. On the other hand, our BiLSTM NER model revealed to be highly accurate, with both accuracy and a F_1 measure of 90%. Furthermore, with the case study datasets we confirmed that our solution is effective. The CNN classifiers achieved TPRs and TNRs of 90% on almost all test sets and the NER models also showed great performance, achieving 90% in both accuracy and F_1 on almost all the test sets.

With these results, we conclude that our binary classifier is a valuable alternative to

the current SVM classifier and that our NER architecture is capable of extracting relevant information from tweets. This information could be used to fill an IoC or a security alert which in turn would be presented to a SOC.

5.1 Future Work

Through this work, we have shown the possibility of integrating state-of-the-art DL algorithms to improve the capabilities of cyberthreat detection tools in OSINT and to provide SOC analysts with relevant information about a given threat.

Nonetheless, there are still several research opportunities that can help further improve and extend the capabilities of our pipeline. In this work we have explored the usage of CNNs for a binary classifier and a BiLSTM neural network for a NER model. However, we did not explore the possibility of using RNNs for the binary classification task. This type of NNs are often deployed in NLP related tasks, as we have shown with our NER model, and can provide better results as shown in the comparative study by Yin et al. [73].

Regarding alternative architectures for the NER, Strubell et al. [67] proposes an architecture based on CNNs that provides equivalent results in comparison with a BiLSTM architecture equal to the one we have deployed, but with better training speeds. As we desire these models to be efficient and scalable, this alternative should be analysed as well as other types of RNNs which tend to be perform as well as LSTM networks, such as the Gated Recurrent Unit [8].

Another interesting route regarding the architecture of these classifiers is to find ways to combine our models. For example, through techniques such as transfer learning [57] which focuses on using a trained model and retrain it on a different but similar task. Another alternative is the implementation of a Multi-Task Learning model [5], an approach that seeks to train the binary classifier and the NER model at the same time. Collobert et al. [10] proposes such an architecture to tackle NLP tasks, in this case the authors only share the embedding layers across models which leads to a better generalization of the semantic relations between vectors.

Regardless of neural architecture, we explored the usage and benefit of pre-trained word vectors. Through such language models we can extract vectors which contain semantic representations of given words. In our experimentations, we used vectors which were trained on common corpora of text. However, a valuable addition to our architecture could be the usage of word vectors that have been trained on information security related text, such as entries in the National Vulnerability Database [52] or information security blogs. To our knowledge, there is no publicly available source from which we can retrieve such vectors from. Thus, future work could focus on choosing an existing language model (e.g., Word2Vec [49] or GloVE [56]), collecting security related text and training a model that would be used for several information security related tasks.

Furthermore, once the system is ready to be deployed we will require a procedure through which we can monitor the model's performance. As we collect new data and use it to evaluate the deployed model, if the model's performance appears to be degrading over time, then future research should focus on a procedure through which we can train a new model and substitute the deployed one. There are several challenges that will have to be considered such has the method through which we substitute the deployed model. One possibility is to take down the currently deployed model and upload the new one, but the system will be unavailable while the transition occurs. Another possibility is to load the new model alongside the deployed model and then take the latter down which would keep the system online. However, this method would require more memory resources, since there would be an instance where both models would be loaded in memory.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[2] Pinkesh Badjatiya, Shashank Gupta, Manish Gupta, and Vasudeva Varma. Deep learning for hate speech detection in tweets. pages 759–760, 2017.

[3] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[4] Eunice Picareta Branco. Cyberthreat discovery in open source intelligence using deep learning techniques. Master's thesis, Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, 2017. `http://hdl.handle.net/10451/30699`.

[5] Rich Caruana. Multitask learning. *Mach. Learn.*, 28(1):41–75, July 1997.

[6] Maureen Caudill. Neural networks primer, part i. *AI Expert*, 2(12):46–52, December 1987.

[7] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[8] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[9] 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, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

[10] 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*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

[11] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel P. Kuksa. Natural language processing (almost) from scratch. *CoRR*, abs/1103.0398, 2011.

[12] F. Concone, A. De Paola, G. L. Re, and M. Morana. Twitter analysis for real-time malware discovery. In *2017 AEIT International Annual Conference*, pages 1–6, Sept 2017.

[13] Andre Correia. Aprendizagem automática em larga escala nas redes sociais para a descoberta de ameaças de segurança. Master's thesis, Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, July 2016. `http://hdl.handle.net/10451/24882`.

[14] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[15] DeepMind. Deepmind - about us. `https://deepmind.com/about/`. Accessed: 23/06/2018.

[16] Adit Deshpande. A beginner's guide to understanding convolutional neural networks. `https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/`. Accessed: 05/07/2018.

[17] Nuno Dionísio, Pedro Ferreira, and Alysson Bessani. Aprendizagem profunda e ameaças de cibersegurança em fontes abertas. *INForum*, 2018.

[18] DiSIEM. H2020 diversity enhancements for security information and event management project. `http://disiem-project.eu/`. Accessed: 30/11/2017.

[19] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, March 2016.

[20] ENISA. Exploring the opportunities and limitations of current threat intelligence platforms. `https://www.enisa.europa.eu/publications/exploring-the-opportunities-and-limitations-of-current-threat-intel` Accessed: 19/09/2018.

[21] ENISA. Risk management - glossary. https://www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/risk-management-inventory/glossary. Accessed: 19/09/2018.

[22] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Mach. Learn.*, 29(2-3):131–163, November 1997.

[23] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[24] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[25] Google. Graphs and sessions. In *TensorFlow Programmer's Guide*, https://www.tensorflow.org/programmers_guide/graphs. Accessed: 30/11/2017.

[26] Google. Pre-trained word and phrase vectors. https://code.google.com/archive/p/word2vec/. Accessed: 26/07/2018.

[27] Google. Tensors. In *TensorFlow Programmer's Guide*, https://www.tensorflow.org/programmers_guide/tensors. Accessed: 30/11/2017.

[28] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[29] Erik Hallström. Introduction to tensorflow — cpu vs gpu. https://medium.com/@erikhallstrm/hello-world-tensorflow-649b15aed18c. Accessed: 30/11/2017.

[30] Douglas Hawkins. The problem of overfitting. 44:1–12, 05 2004.

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

[32] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153, Sept 2009.

[33] Antonio Jimeno-Yepes and Andrew MacKinlay. Ner for medical entities in twitter using sequence to sequence neural networks. In *ALTA*, 2016.

[34] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. Bag of Tricks for Efficient Text Classification. *ArXiv e-prints*, July 2016.

[35] Anil K. Jain. Data clustering: 50 years beyond k-means. In *Pattern Recognition Letters*, volume 31, pages 651–666, 06 2010.

[36] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188, 2014.

[37] Andrej Karpathy. Neural networks. In *Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition*, http://cs231n.github.io/neural-networks-1/. Accessed: 22/06/2018.

[38] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/, May 2015. Accessed: 22/11/2017.

[39] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, pages 1725–1732, Washington, DC, USA, 2014. IEEE Computer Society.

[40] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952.

[41] Yoon Kim. Convolutional neural networks for sentence classification. In *EMNLP 2014*, New York University, September 2014.

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

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

[44] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[45] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *CoRR*, abs/1603.01360, 2016.

[46] Quentin Le Sceller, ElMouatez Billah Karbab, Mourad Debbabi, and Farkhund Iqbal. Sonar: Automatic detection of cyber security events over the twitter stream. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, pages 23:1–23:11, New York, NY, USA, 2017. ACM.

[47] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[48] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, Zhou Li, Luyi Xing, and Raheem Beyah. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 755–766, New York, NY, USA, 2016. ACM.

[49] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. *ArXiv e-prints*, January 2013.

[50] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed Representations of Words and Phrases and their Compositionality. *ArXiv e-prints*, October 2013.

[51] Andrew Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 78–, New York, NY, USA, 2004. ACM.

[52] NIST. National vulnerability database. `https://nvd.nist.gov/`. Accessed: 06/08/2018.

[53] Christopher Olah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs, August 2015. Accessed: 22/11/2017.

[54] OpenAI. About openai. `https://openai.com/about/`. Accessed: 23/06/2018.

[55] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

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

[57] L. Y. Pratt. Discriminability-based transfer between neural networks. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 204–211. Morgan-Kaufmann, 1993.

[58] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[59] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[60] Alon Rozental and Daniel Fleischer. Amobee at semeval-2017 task 4: Deep learning system for sentiment detection on twitter. *CoRR*, abs/1705.01306, 2017.

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

[62] Carl Sabottke, Octavian Suciu, and Tudor Dumitras. Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 1041–1056, Washington, D.C., 2015. USENIX Association.

[63] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[64] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM.

[65] Aliaksei Severyn and Alessandro Moschitti. Twitter sentiment analysis with deep convolutional neural networks. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '15, pages 959–962, New York, NY, USA, 2015. ACM.

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

[67] Emma Strubell, Patrick Verga, David Belanger, and Andrew McCallum. Fast and accurate sequence labeling with iterated dilated convolutions. *CoRR*, abs/1702.02098, 2017.

[68] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[69] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[70] Twitter. Tweepy, twitter python api. `https://www.tweepy.org/`. Accessed: 19/07/2018.

[71] Twitter. Twitter api documentation. `https://developer.twitter.com/en/docs`. Accessed: 19/07/2018.

[72] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[73] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of CNN and RNN for natural language processing. *CoRR*, abs/1702.01923, 2017.