

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

**Tor Traffic Detection Using Convolutional and
Recurrent Neural Networks**

Diploma Thesis

Georgios Simos

Supervisor: **Dimitrios Katsaros**

2nd Committee Member: **Lefteris Tsoukalas**

3rd Committee Member: **Athanasios Korakis**

Volos 2020



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

**ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ**

**Ανίχνευση Κίνησης Tor Με Χρήση Συνελικτικών και
Αναδρομικών Νευρωνικών Δικτύων***

Διπλωματική Εργασία

Γεώργιος Σίμος

Επιβλέπων: **Δημήτριος Κατσαρός**

2ο Μέλος Επιτροπής: **Ελευθέριος Τσουκαλάς**

3ο Μέλος Επιτροπής: **Αθανάσιος Κοράκης**

Βόλος 2020

**Αυτός είναι ο πραγματικός τίτλος της εργασίας. Εκ παραδρομής καταχωρήθηκε και εγκρίθηκε ο τίτλος που αναγράφεται στην επόμενη σελίδα.*



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

**ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ**

**Ανίχνευση Κίνησης Tor Συνελικτικών και Αναδρομικών
Νευρωνικών Δικτύων**

Διπλωματική Εργασία

Γεώργιος Σίμος

Επιβλέπων: **Δημήτριος Κατσαρός**

2ο Μέλος Επιτροπής: **Ελευθέριος Τσουκαλάς**

3ο Μέλος Επιτροπής: **Αθανάσιος Κοράκης**

Βόλος 2020

Dedicated to my family

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

«Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής».

Ο Δηλών



Σίμος Γεώργιος
20/08/2020

Copyright© 2020 by Simos Georgios

“The copyright of this thesis rests with the authors. No quotations from it should be published without the authors’ prior written consent and information derived from it should be acknowledged”

Abstract

Nowadays, due to the technological innovations and the increased volume of data accessible via the internet, the need for cyber security is increasing significantly and its achievement is becoming more and more demanding and important. For this reason, the scientific community is trying to find ways to ensure cyber security, first by identifying those who pose a threat and then by fighting them. The field of Deep Learning is decisively involved in this endeavour and is applied to the detection of such security threats to information systems. This Thesis will study the application of Convolutional (CNN) and Recurrent (RNN) Neural Networks for Tor Traffic Detection, using the time-based features of various network flows obtained. Tor is an open source software that achieves anonymous communication on the internet, making it a popular choice for malicious users to perform cyber attacks. Network flows and their time-based features are the dataset which the experiment is based on. The two Neural Networks which were implemented are intended to predict whether each network flow of the dataset is Tor or nonTor. With the training and testing of the Convolutional Neural Network model that was developed, an accuracy of 95.11 % is observed, and respectively, with the training and testing of the Recurrent Neural Network model that was developed, 95.81 % accuracy is observed. The implementation of the Neural Networks was accomplished with the Keras (API written in python) over the Tensorflow machine learning platform.

Περίληψη

Στην εποχή μας, λόγω των καινοτομιών της τεχνολογίας που κατακλύζουν την σύγχρονη ζωή, η ανάγκη για ασφάλεια στο διαδίκτυο αυξάνεται σημαντικά και γίνεται ολοένα και πιο απαιτητική η επίτευξή της. Ως αποτέλεσμα, η σύγχρονη επιστημονική κοινότητα, προσπαθεί να βρει τρόπους διασφάλισης του cyber security, πρώτα με την αναγνώριση αυτών που αποτελούν απειλή και μετέπειτα με την καταπολέμησή τους. Το πεδίο του Deep Learning μπαίνει αποφασιστικά σε αυτήν την προσπάθεια και εφαρμόζεται στην ανίχνευση τέτοιων απειλών ασφαλείας για τα πληροφοριακά συστήματα. Σε αυτή την Διπλωματική Εργασία θα μελετηθεί η εφαρμογή Συνελικτικών (CNN) και Αναδρομικών (RNN) Νευρωνικών Δικτύων για την ανίχνευση Tor Traffic, χρησιμοποιώντας τα σχετιζόμενα με χρόνους χαρακτηριστικά διαφόρων ροών διαδικτύου που ελήφθησαν. Το Tor είναι λογισμικό που επιτυγχάνει ανώνυμη επικοινωνία στο διαδίκτυο με αποτέλεσμα να είναι δημοφιλής επιλογή των κακόβουλων χρηστών για πραγματοποίηση επιθέσεων. Οι ροές διαδικτύου και τα σχετιζόμενα με χρόνους χαρακτηριστικά τους αποτελούν τα δεδομένα (dataset) τα οποία στηρίζεται το πείραμα. Τα δύο Νευρωνικά Δίκτυα που υλοποιήθηκαν έχουν σκοπό την πρόβλεψη για το αν η κάθε ροή που εξετάζεται είναι Tor ή nonTor. Με την εκπαίδευση και δοκιμή του μοντέλου του Συνελικτικού Νευρωνικού Δικτύου που αναπτύχθηκε, παρατηρείται ακρίβεια (accuracy) της τάξης του 95.11 % και αντίστοιχα με την εκπαίδευση και δοκιμή του μοντέλου του Αναδρομικού Νευρωνικού Δικτύου που αναπτύχθηκε, παρατηρείται ακρίβεια (accuracy) 95.81 %. Η υλοποίηση των Νευρωνικών Δικτύων πραγματοποιήθηκε με το Keras (API γραμμένο σε python) πάνω από την πλατφόρμα μηχανικής μάθησης Tensorflow.

Table of Contents

<i>Abstract.....</i>	<i>7</i>
<i>Περίληψη.....</i>	<i>8</i>
<i>Table of Contents.....</i>	<i>9</i>
<i>List of Abbreviations</i>	<i>11</i>
<i>CHAPTER 1.....</i>	<i>12</i>
<i>Introduction to Deep Learning.....</i>	<i>12</i>
1.1 Deep Learning General Overview.....	12
1.1.1 Examples of Deep Learning architectures.....	13
1.1.2 Fields that Neural Networks have been applied to	14
1.2 Convolutional Neural Networks (CNN).....	16
1.2.1 CNN Layers.....	16
1.2.2 Filter Hyperparameters.....	16
1.2.3 CNN Activation Functions	18
1.3 Recurrent Neural Networks (RNN)	19
1.3.1 RNN Loss Function.....	19
1.3.2 Backpropagation through time	20
1.3.3 RNN commonly used Activation Functions.....	20
1.3.4 Vanishing and Exploding Gradient.....	22
1.3.5 LSTM Networks.....	22
1.4 Deep Learning in Cyber Security	23
1.4.1 Common Cyber Security Attacks	24
1.4.2 Applications of Deep Learning in Cyber Security	25
<i>CHAPTER 2.....</i>	<i>26</i>
<i>Introduction to Tor Software</i>	<i>26</i>
2.1 Tor Software General Overview	26
2.2 Onion Routing Operation	26
2.2.1 Operation	27
2.2.2 Onion Services.....	27
2.3 Tor Implementations.....	28
2.3.1 Tor Browser	28
2.3.2 Tor Messenger	28
2.4 Tor Usage in Cyber Attacks	29
2.4.1 Onion Ransomware	29
<i>CHAPTER 3.....</i>	<i>31</i>
<i>Frameworks, APIs and Datasets</i>	<i>31</i>
3.1 Tensorflow Framework Overview.....	31
3.1.1 Features of Tensorflow	31

3.2 Keras API Overview	32
3.2.1 Features of Keras	32
3.3 Tor Traffic Detection Dataset.....	33
3.3.1 Meta - Information parameters of the Dataset.....	33
3.3.2 Dataset Explanation.....	34
CHAPTER 4.....	35
<i>Experiment Methodology.....</i>	35
4.1 Preparation.....	35
4.1.1 System Preparation	35
4.1.2 General Data Preparation	35
4.1.2 CNN and RNN Extra Data Preparation.....	39
4.2 CNN Model Implementation.....	40
4.3 CNN Model Training and Testing	43
4.4 RNN Model Implementation.....	46
4.5 RNN Model Training and Testing	51
4.6 CNN and RNN Models Comparison	54
CHAPTER 5.....	55
<i>Conclusions.....</i>	55
5.1 Conclusion	55
5.2 Future Work	55
<i>Bibliography.....</i>	56

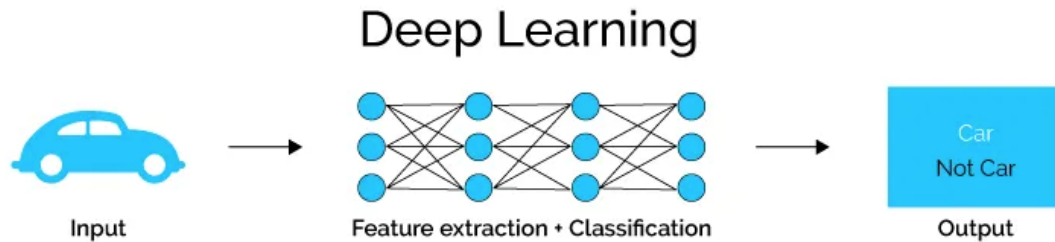
List of Abbreviations

AI	Artificial Intelligence
DNN	Deep Neural Network
CNN	Convolutional Neural Network
CONV	Convolution
POOL	Pooling
FC	Fully Connected
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
LSTM	Long Short Term Memory Network
SQL	Structured Query Language
DOS	Denial of Service
IDS/IPS	Intrusion Detection and Prevention System
NLP	Natural Language Processing
ANN	Artificial Neural Network
HTTPS	Hypertext Transfer Protocol Secure
UEBA	User and Entity Behaviour Analytics
TOR	The Onion Router
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
FIAT	Forward Inter Arrival Time
BIAT	Backward Inter Arrival Time
FLOWIAT	Flow Inter Arrival Time
FB PSEC	Flow Bytes Per Second
CSV	Comma Separated Values

CHAPTER 1

Introduction to Deep Learning

1.1 Deep Learning General Overview



1.1.1 Deep Learning Representation Example [1]

Deep Learning is a machine learning subfield which is associated with algorithms inspired by the structure and function of the brain, called artificial neural networks. So Deep Learning utilizes a hierarchical level of artificial neural networks to carry out the process of machine learning. The artificial neural networks mimic the human brain, with neuron nodes connected together like a web. While traditional programs build analysis with data in a linear way, the hierarchical function of deep learning systems enables machines to process data with a nonlinear approach [2]. In figure 1.1.1 we can see a Deep Learning classification example that classifies an object as a Car or Not Car. Figure 1.1.2 explains the idea of AI subsets and where Deep Learning belongs as a subset.

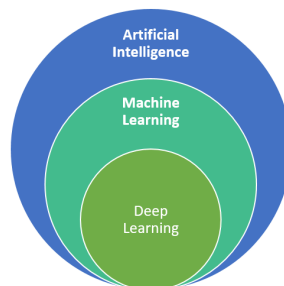


Figure 1: artificial intelligence, machine leaning and deep learning Source: Nadia BERCHANE (M2 IESCI, 2018)

1.1.2 Graph of AI subsets [3]

1.1.1 Examples of Deep Learning architectures

Neural Networks were inspired by information processing and distributed communication nodes in biological systems. The adjective “deep” in deep learning comes from the use of multiple layers in the network. Some Deep Learning [4] architectures that are implemented as Neural Networks are the following:

- **Deep Neural Networks**

Deep Neural Networks are Neural Networks with multiple layers between the input and output layers. The DNN finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a non-linear relationship. The network moves through the layers calculating the probability of each output.

- **Deep Belief Neural Networks**

Deep Belief Neural Networks are a class of Deep Neural Networks composed of multiple layers of hidden variables with connection between the layers but not between the hidden variables.

- **Recurrent Neural Networks**

Recurrent Neural Networks are a class of Neural Networks, which exhibit temporal dynamic behaviour. They are explained analytically in subchapter 1.3.

- **Convolutional Neural Networks**

Convolutional Neural Network is a class of Deep Neural Networks, most commonly applied to analysing imagery. They are explained analytically in subchapter 1.2.

1.1.2 Fields that Neural Networks have been applied to

Neural Networks have been applied to a lot of scientific fields. Some of these fields and their applications are explained in the list below.

- **Computer Vision**

Particularly, Convolutional Neural Networks have applications in object recognition, identification and detection. The best algorithms for the previous tasks are implemented by CNNs.

- **Machine Vision**

Deep Learning significantly expand machine vision capabilities in image processing.

- **Speech Recognition**

Deep feedforward and Recurrent Neural Networks algorithms contributed to the implementation of the end-to-end automatic speech recognition. Also in car systems, health care, military, telephony, education and people with disabilities.

- **Natural Language Processing**

Neural Networks algorithms have been applied to natural language processing tasks, such as automatic summarization, dialogue management, book generation and machine translation.

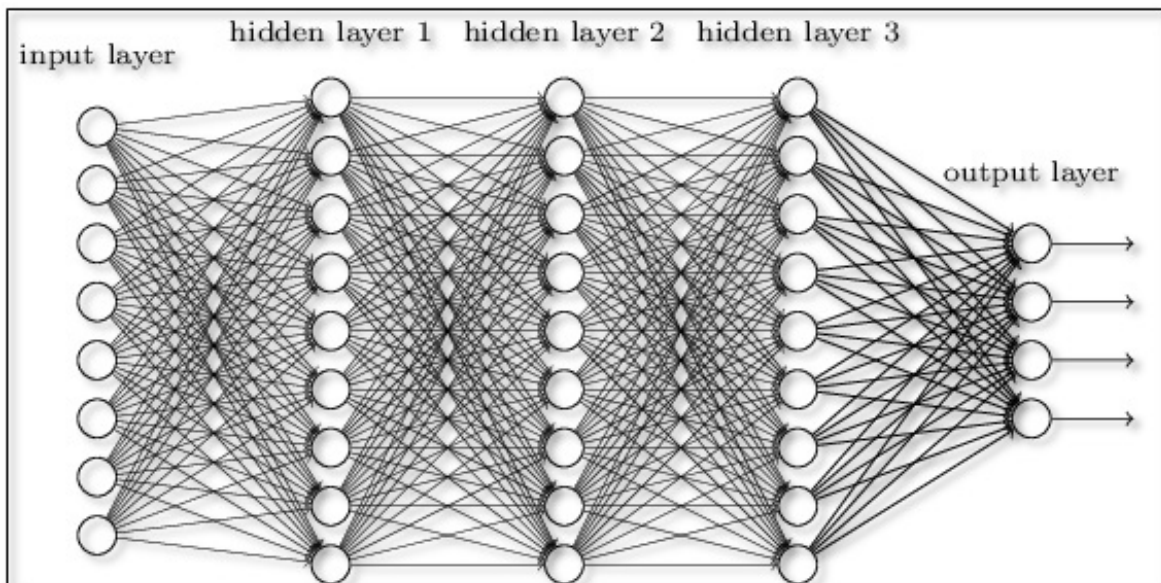
- Bioinformatics

Neural Network algorithms have been applied to protein sequences clustering, gene detection within a sequence and more generally, they have been applied to increase the understanding of biological processes.

- Drug Design

Deep Learning algorithms have been applied to Computer-aided drug design.

In all of these fields, Deep Learning have produced results comparable to and some times surpassing human expert performance. A Deep Neural Network is shown in figure 1.1.3.



1.1.3 Deep Neural Network [5]

1.2 Convolutional Neural Networks (CNN)

Convolutional Neural Network [6] is a specific type of neural network which is one of the main categories to do images recognition and classification. It is generally formed by the following layers:

1.2.1 CNN Layers

- **Convolution layer (CONV)**

It performs convolution operations in the input data and the output which is generated is a feature map or activation map.

- **Pooling (POOL)**

This layer can function either as Max Pooling or as Average Pooling. In Max Pooling, each operation selects the maximum value of current view. In Average Pooling, each operation selects the average values of the current view.

- **Fully Connected (FC)**

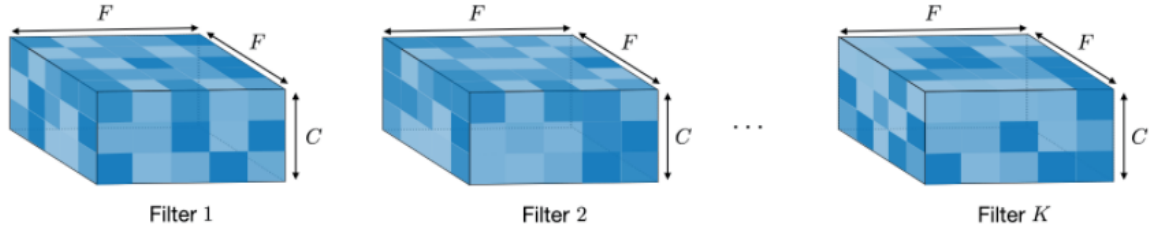
This layer operates on a flattened input and can be used for objectives optimizations such as class scores in the end of a CNN architecture.

1.2.2 Filter Hyperparameters

The convolution layer filters contain the following hyperparameters.

- **Dimensions of a filter**

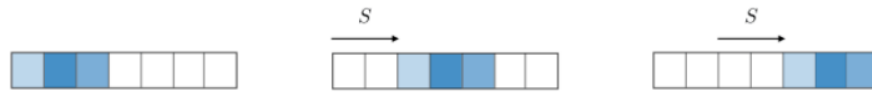
In the figure 1.2.1 we can see the application of K filters of size $F \times F$, which results in an output feature map of size $O \times O \times K$.



1.2.1 Filter Explanation

• Stride

In the figure 1.2.2 Stride is explained visually.



1.2.2 Stride Explanation

• Zero-padding

It is the process of adding P zeros to each side of the boundaries of the input. This value can be set through one of the three modes in figure 1.2.3 below:

Mode	Valid	Same	Full
Value	$P = 0$	$P_{\text{start}} = \left\lfloor \frac{S \lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rfloor$ $P_{\text{end}} = \left\lceil \frac{S \lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rceil$	$P_{\text{start}} \in [0, F - 1]$ $P_{\text{end}} = F - 1$
Illustration			
Purpose	<ul style="list-style-type: none"> No padding Drops last convolution if dimensions do not match 	<ul style="list-style-type: none"> Padding such that feature map size has size $\lceil \frac{I}{S} \rceil$ Output size is mathematically convenient Also called 'half' padding 	<ul style="list-style-type: none"> Maximum padding such that end convolutions are applied on the limits of the input Filter 'sees' the input end-to-end

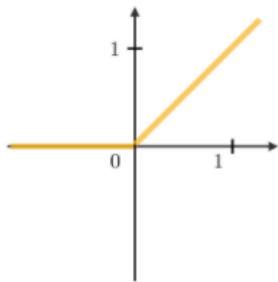
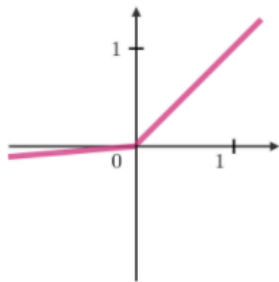
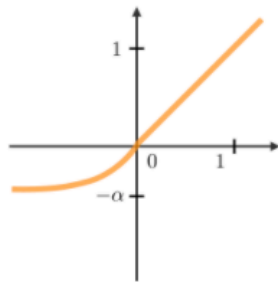
1.2.3 Zero-padding Modes

1.2.3 CNN Activation Functions

Commonly used Activation Functions for Convolutional Neural Networks are the following ones:

- **Rectified Linear Unit**

In figure 1.2.4 below, we can see the three different variants of ReLU function.

ReLU	Leaky ReLU	ELU
$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$	$g(z) = \max(\alpha(e^z - 1), z)$ with $\alpha \ll 1$
		
• Non-linearity complexities biologically interpretable	• Addresses dying ReLU issue for negative values	• Differentiable everywhere

1.2.4 ReLU

- **Softmax**

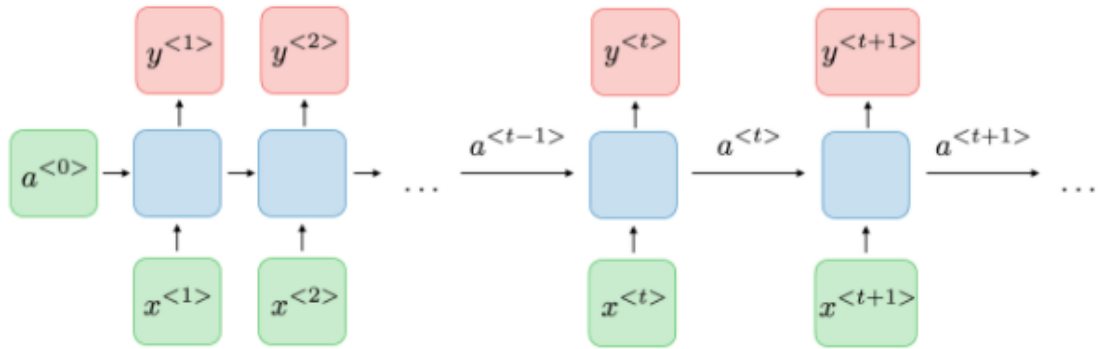
Figure 1.2.5 shows Softmax function definition and calculation formula.

$$p = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

1.2.5 Softmax

1.3 Recurrent Neural Networks (RNN)

Recurrent Neural Networks [7] can remember their past decisions and their new decisions are influenced by these past decisions. This means that an RNN can learn while training and moreover can learn things from past inputs while generating outputs. The general structure is shown in figure 1.3.1.



1.3.1 Architecture of a traditional RNN

1.3.1 RNN Loss Function

In RNN, the loss function L of all time steps is defined based on the loss at every time step as follows.

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}(\hat{y}^{<t>}, y^{<t>})$$

1.3.2 Loss function formula

1.3.2 Backpropagation through time

Timestep T - Loss L - Weight Matrix W. At timestep T, the derivative of the loss L with respect to weight matrix W is expressed as follows in figure 1.3.3.

$$\frac{\partial \mathcal{L}^{(T)}}{\partial W} = \sum_{t=1}^T \frac{\partial \mathcal{L}^{(T)}}{\partial W} \Big|_{(t)}$$

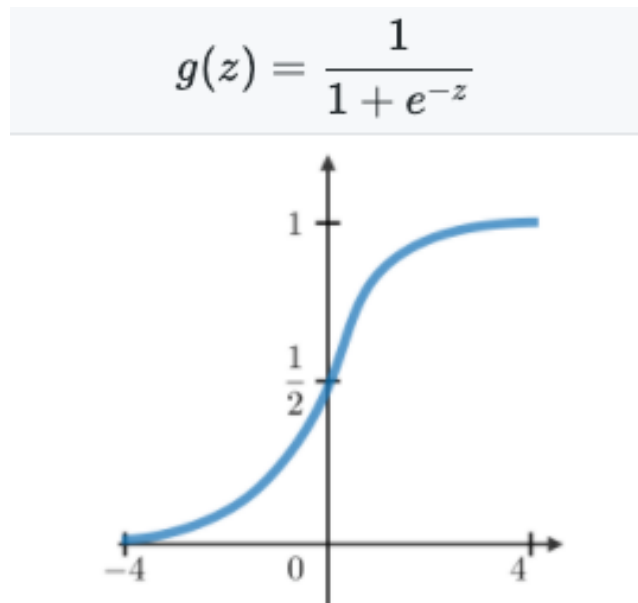
1.3.3 Backpropagation formula

1.3.3 RNN commonly used Activation Functions

The following activation functions are the most commonly used by RNNs.

- **Sigmoid**

Sigmoid function is described as follows.

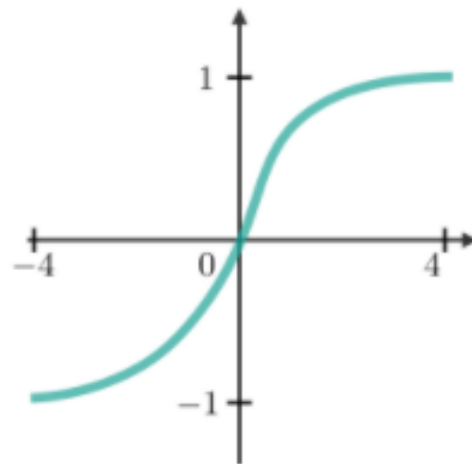


1.3.4 Sigmoid Function

- **Tanh**

Tanh function is described as follows.

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

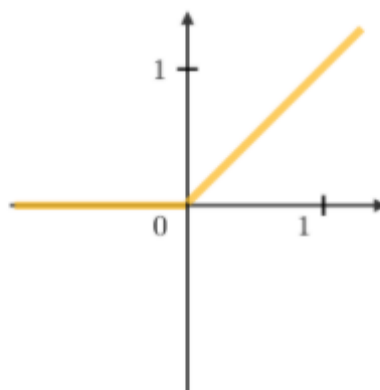


1.3.5 Tanh Function

- **ReLU**

ReLU function is described as follows.

$$g(z) = \max(0, z)$$



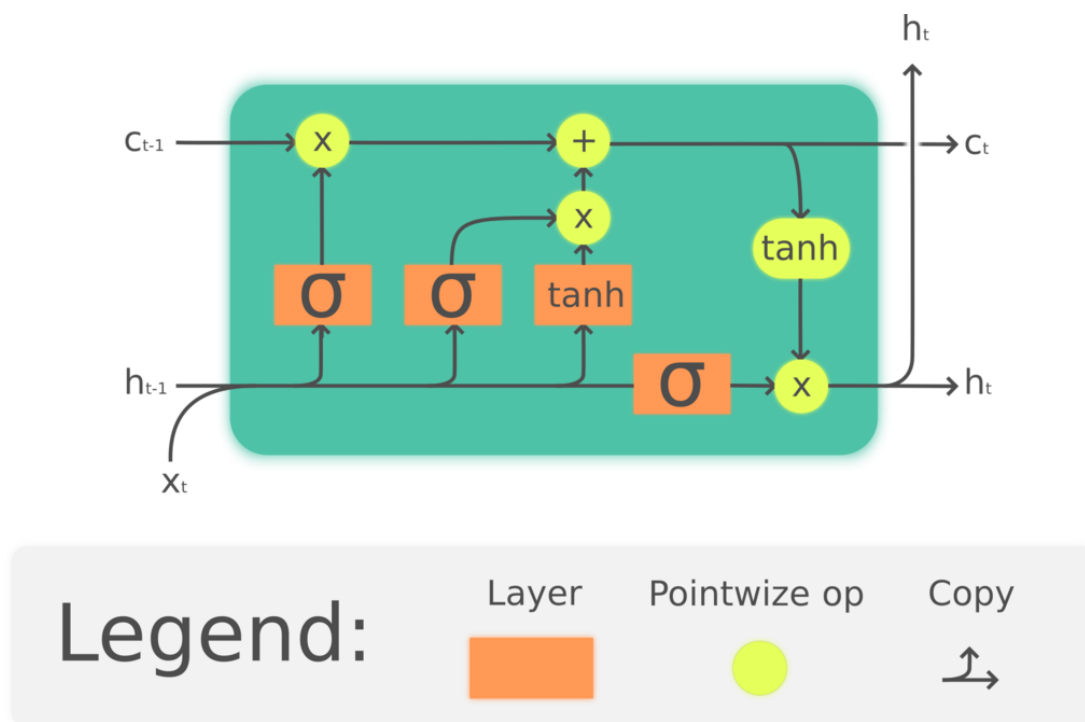
1.3.6 ReLU Function

1.3.4 Vanishing and Exploding Gradient

These phenomena are often observed in the context of Recurrent Neural Networks. Vanishing gradient happens because of the multiplicative gradient that can be decreasing with respect to the number of layers. On the other hand, Exploding Gradient happens because of the multiplicative gradient that can be increasing with respect to the number of layers. These two phenomena belong to the long-term dependency problem category. The solution to this problem has come by the implementation of LSTM Networks.

1.3.5 LSTM Networks

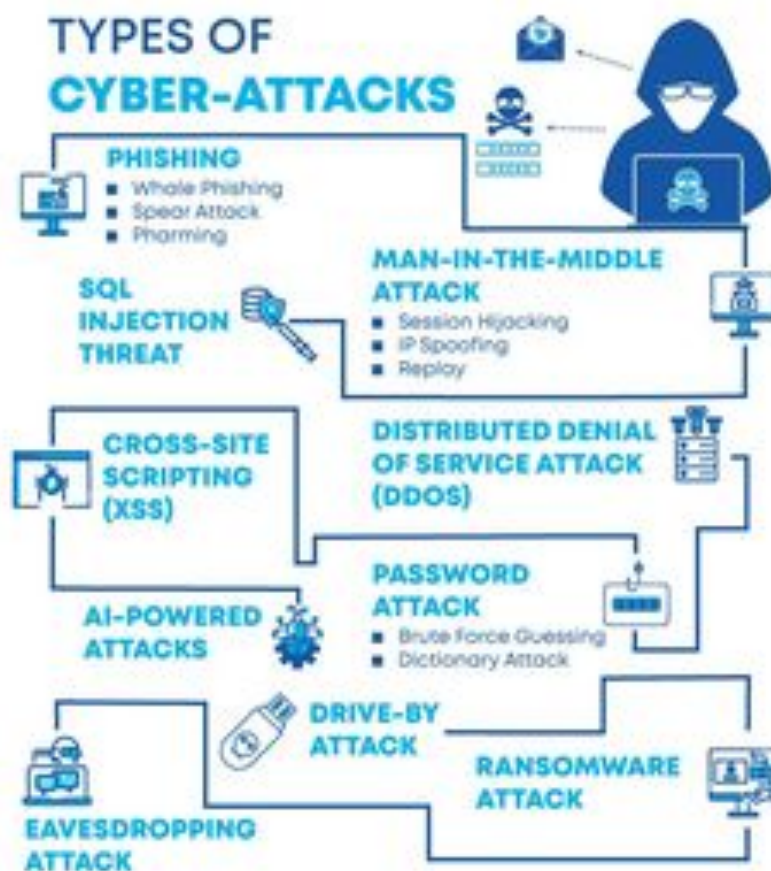
Long Short Term Memory Networks (LSTMs) [8] are a special kind of RNN, capable of learning long-term dependencies. LSTMs are designed in a way that gives a solution to the long-term dependency problem. They remember information for long periods of time and they have a chain like structure of four neural network layers interactive in a very special way.



1.3.7 LSTM Network Structure[9]

1.4 Deep Learning in Cyber Security

One of the most common and critical applications for deep learning algorithms is to improve cybersecurity solutions. This section is referred to some very common Cyber Security Attacks and the most common Applications of Deep Learning in Cyber Security. Some of them are described in the figure 1.4.1.[10]



1.4.1 Types of Cyber Attacks [11]

1.4.1 Common Cyber Security Attacks

The following cyber attacks can be irreparably critical for the systems which they infect. In particular attacks like:

- **Malware**

Malicious Software to damage devices, systems and networks.

- **Data Breach**

Non authorised access to valuable confidential personal data.

- **Social Engineering**

Users manipulation to allow attackers to grant access of critical data.

- **Phishing**

The act of sending infected emails cloaked as legitimate to trick victims into giving personal and critical data.

- **SQL injection**

A technique used by attackers to leverage vulnerabilities within SQL servers to access the database and run malicious code.

- **DOS attack**

Technique for flooding networks and servers with traffic and making them unavailable.

- **Insider Threats**

An attack caused by company insiders such as employees.

- **Advanced Persistent Threats**

Attacks capable of evading traditional defensive and perimeter security tools due to their stealthy nature.

1.4.2 Applications of Deep Learning in Cyber Security

Cyber Security have been backed up by Deep Learning techniques and applications as follows:

- **Intrusion Detection and Prevention Systems (IDS/IPS)**

These systems are useful in attacks like data breaches. They can detect malicious network activities and prevent intruders from granting access. Convolutional and Recurrent Neural Networks can improve these systems and make them smarter by analysing traffic with better accuracy.

- **Dealing with Malware**

Deep Learning algorithms can improve the systems that detect and deal with malware attacks by learning the system to recognise malicious activities and more advanced threats.

- **Spam and Social Engineering Detection**

Natural Language Processing (NLP) is a deep learning technique which can detect and block spam using language patterns and various statistical models.

- **Network Traffic Analysis**

ANNs can analyse HTTPS network traffic and detect malicious activities such as SQL injections and DOS attacks.

- **User Behaviour Analytics**

User and Entity Behavior Analytics (UEBA), after a learning period, they can detect suspicious activities that possible indicate an insider attack.

CHAPTER 2

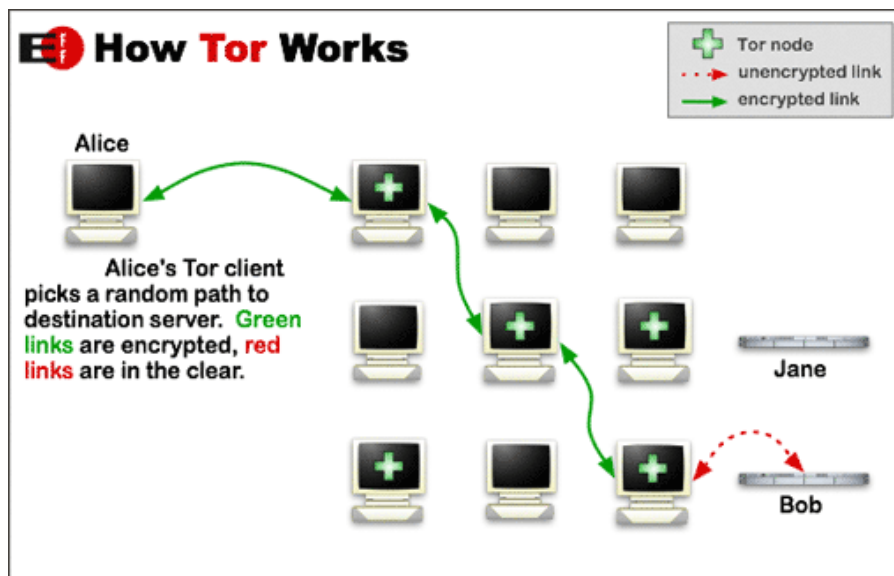
Introduction to Tor Software

2.1 Tor Software General Overview

The Onion Router (Tor) [12] software is an open-source software, which is developed in a way that enables anonymous communication. With Tor, network traffic is directed through a network which consists of more than seven thousand relays to hide user's location and usage from anyone conducting traffic analysis and surveillance. Tor's purpose is to protect users' privacy and their freedom and ability to conduct confidential communication without being monitored.

2.2 Onion Routing Operation

Onion Routing is a process in which the application layer of a communication protocol stack gets encrypted in a way that looks like the layers of an onion. It encrypts the data that is transferred and the next node destination IP address, many times and sends it through a virtual circuit consisting of successive random-selection Tor relays. Such an example is shown in figure 2.2.1.



2.2.1 Tor Traffic Transfer

2.2.1 Operation

Tor protects its users' identities and their online activity from network traffic surveillance by separating identification and routing. The implementation of onion routing operates with the help of onion routers, which they are provided by volunteers around the globe. These onion routers provide encryption in a multi-layered manner to ensure perfect forward secrecy between relays, contributing in users' anonymity in a network location. Anyone who eyedrops at any point along the communication channel cannot directly identify both ends because the IP addresses of the sender and the receiver are not both in cleartext at any hop of the channel. It is appeared to the receiver, that the exit node, which is the last node of the communication, is the originator of the communication, rather than the sender, whose node is originally the first one and actually the originator of the communication. In this way the receiver does not have a single clue of network traffic information about the identity of the sender.

2.2.2 Onion Services

Onion services are called the servers configured to receive inbound connections only through Tor. An onion service doesn't reveal its IP address and thus its network location, so it's accessed through its onion address. There is a distributed hash table within the Tor Network, which keeps the services' corresponding public keys and introduction points for giving the Tor Network the ability to understand the servers' addresses. It can route data to and from onion services while preserving the anonymity of both parties.

2.3 Tor Implementations

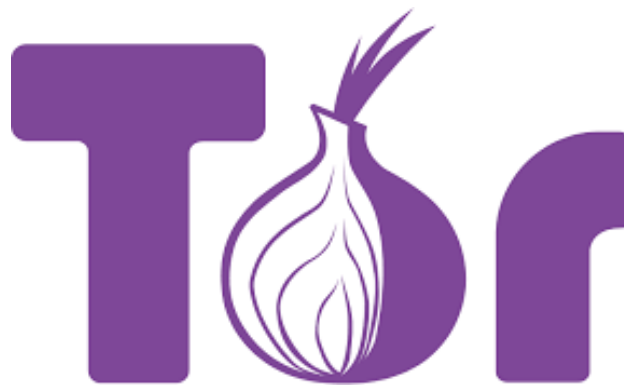
Tor's main implementation consists of up to 540.000 lines of code and is written in C, along with Python, Javascript and several other programming languages.

2.3.1 Tor Browser

The Tor Browser is the flagship and main application of the Tor Project. It was created as the Tor Browser Bundle by Steven J. Murdoch and announced in January 2008. The Tor Browser is implemented to automatically start Tor background processes and routes traffic through the Tor Network, providing users with instant anonymity regarding their network activity. When a Tor Browser session terminates, the browser deletes all HTTP cookies and the session's browsing history.

2.3.2 Tor Messenger

Tor Messenger Beta was released on October 2015 by Tor Project and is an instant messaging application which supports multiple different instant messaging protocols which provided anonymity. The Tor Messenger project was shut down in April 2018, after becoming vulnerable because of outdated software dependencies.



2.3.1 Tor Logo [13]

2.4 Tor Usage in Cyber Attacks

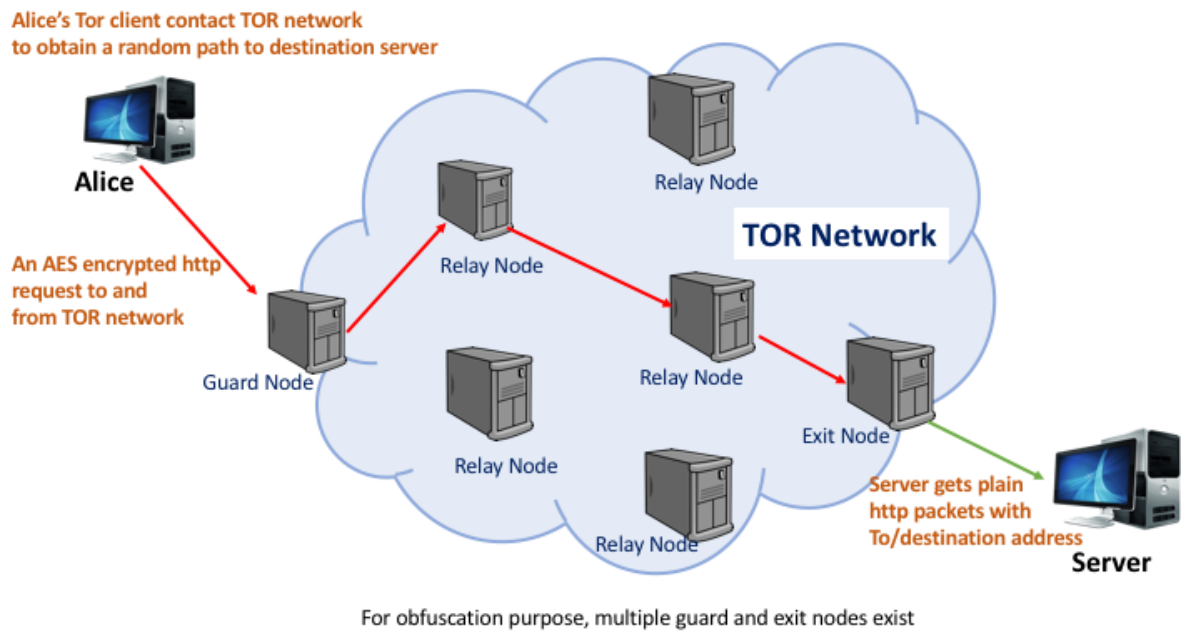
Cyber attacks [14] aim to accomplish stealing of the enterprise customer data, source codes, software keys and information that can be profitable. The adversaries traffic the stolen information along remote servers using encryption alongside the regular network traffic. For this kind of illegal trafficking, an anonymous network must be used that it makes it difficult for the network security engineers to monitor the traffic. Also, the data must be encrypted, rendering rule-based network intrusion tools and firewalls to be ineffective. This can be achieved by the usage of Tor network.

2.4.1 Onion Ransomware

Onion Ransomware [15] is a cyber attack, which uses Tor Network in order to achieve anonymous communication across the internet. The “Onion” is an encrypting ransomware, which encrypts user’s data and uses a virtual countdown mechanism to frighten victims into paying for their data decryption in Bitcoins. This malware hide its malicious behaviour and make it difficult for security engineers to track its creators, because it uses the Tor Network.

In more detail, the “Onion” communicates with command and control servers, that they belong somewhere inside the Tor Network. Anonymity protects the identity of cyber attackers and the use of a powerful cryptographic scheme makes data decryption impossible, even in the case of traffic interception between the “Onion” and the server. In figure 2.4.2, we can see the number of onion ransomware attacks per country, which were detected.

In the following example in figure 2.4.1, we can see that Alice’s encrypted request traffic gets through multiple nodes before reaching the destination address. This path is completely random and is obtained by TOR network.



2.4.1 Representation of TOR communication between Alice and a destination server.

Country	Number of users attacked
Russia	24
Ukraine	19
Kazakhstan	7
Belarus	9
Georgia	1
Germany	1
Bulgaria	1
Turkey	1
United Arab Emirates	1
Libya	1

2.4.2 Onion Ransomware attacks that were detected

CHAPTER 3

Frameworks, APIs and Datasets

3.1 Tensorflow Framework Overview

The Tensorflow [16] framework is the one that is used for implementing the neural networks that are needed in this thesis. Tensorflow is a deep learning library that offers data flow programming which performs a range of machine learning tasks. It runs on multiple CPUs and GPUs and mobile operating systems. It is open-source and it has several wrappers in several programming languages like Python, C++ and Java. It is developed and maintained by Google.

3.1.1 Features of Tensorflow

The features of Tensorflow that should be taken into consideration and are important and helpful for Deep Learning projects are the following:

- Enables faster debugging with Python tools
- Dynamic models with python control flow
- Supports custom and higher-order gradients
- Offers multiple levels of abstraction for building and training models
- Quick model training and deployment
- Provides the flexibility and control with APIs
- Well-structured and easy to understand documentation

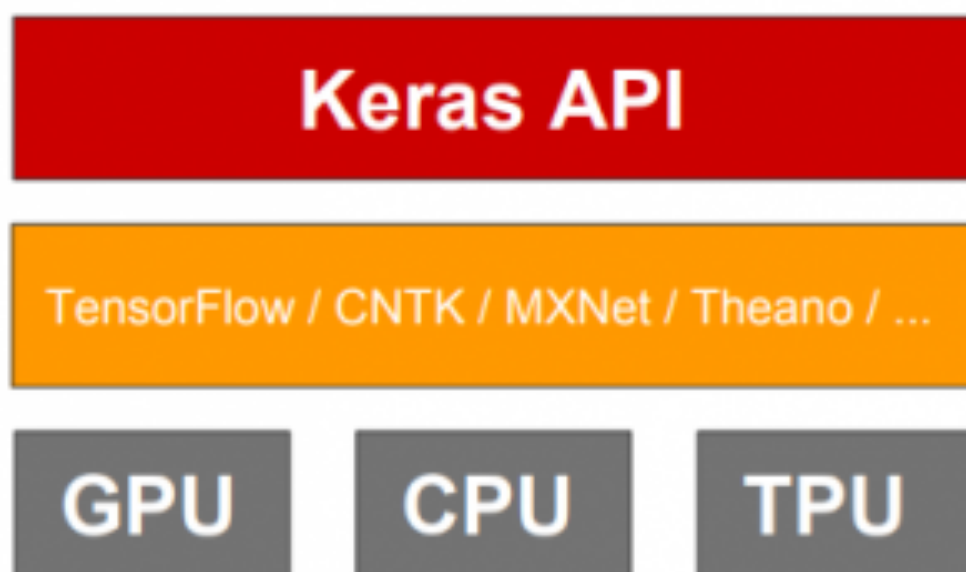
3.2 Keras API Overview

The Keras API [17] is the one that is used for the implementation of this thesis' neural networks. It is an open source Neural Network library written in Python that runs on top of Tensorflow. It is modular, fast and easy to use. It is a useful library for constructing any deep learning algorithm.

3.2.1 Features of Keras

The features of Keras that programmers should take into consideration for choosing it as his project's API are the following:

- Multi backend and multi-platform
- Easy models production
- Convolutional and Recurrent Neural Networks support
- Expressive, flexible and apt for innovative research
- Highly modular neural networks library written in Python
- Provides fast experimentation
- Well-structured and easy to understand documentation
- Focus on user experience



3.2.1 Tensorflow and Keras Communication [18]

3.3 Tor Traffic Detection Dataset

The data that were used for the data experiments in this thesis were obtained from Habibi Lashkari [19] at the University of New Brunswick.

3.3.1 Meta - Information parameters of the Dataset

Their data consist of the following time-based features extracted from the analysis of the university internet traffic:

- FIAT

Forward Inter Arrival Time (Mean, Min, Max, Std)

The time between two packets sent forward direction.

- BIAT

Backward Inter Arrival Time (Mean, Min, Max, Std)

The time between two packets sent backwards.

- FLOWIAT

Flow Inter Arrival Time (Mean, Min, Max, Std)

The time between two packets sent in either direction

- ACTIVE

The amount of time a flow was active before becoming idle (Mean, Min, Max, Std)

- IDLE

The amount of time a flow was idle before becoming active (Mean, Min, Max, Std)

- FB PSEC

The duration of the flow. Flow bytes per second. Flow packets per second

There are also more flow-based parameters in the dataset. A sample instance of the dataset is shown below:

```
Source IP, Source Port, Destination IP, Destination Port,
Protocol, Flow Duration, Flow Bytes/s, Flow Packets/s, Flow IAT
Mean, Flow IAT Std, Flow IAT Max, Flow IAT Min, Fwd IAT Mean, Fwd
IAT Std, Fwd IAT Max, Fwd IAT Min, Bwd IAT Mean, Bwd IAT Std, Bwd
IAT Max, Bwd IAT Min, Active Mean, Active Std, Active Max, Active
Min, Idle Mean, Idle Std, Idle Max, Idle Min, label
10.0.2.15,53913,216.58.208.46,80,6,435,0,4597.7011494253,435,0,435,
435,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,nonTOR
```

3.3.1 CSV file screenshot of the data labels and the first network flow sample

3.3.2 Dataset Explanation

Two different datasets have been merged to create the main dataset which is used in this experiment. The first one is the Tor dataset which was generated by the ISCXFlowMeter application and contains the flows and all necessary parameters. The second one is a public dataset of encrypted traffic generated by Draper-Gil, which includes the same applications on the same network. All flows from the Tor dataset labeled as Tor and all flows from Draper-Gil labeled as nonTor. This data merge is used as an input for the neural networks of this experiment.

CHAPTER 4

Experiment Methodology

4.1 Preparation

4.1.1 System Preparation

The steps I took for preparing my system for this thesis' experiment are the following:

- Installation of Miniconda 3 64bit for MacOSX to create the Python 3 environment for the experiment
- Installation of Tensorflow above the Python 3 environment
- Activation of Tensorflow environment

After these steps the system is ready to support Python 3 code using the tensorflow.keras libraries for the implementation of the Neural Networks needed in this project.

4.1.2 General Data Preparation

The project data was mentioned and explained in Chapter 3, subchapter 3.3 of this thesis. I prepared the data, focusing in keeping time-based features in the input and dropping the features which are irrelevant to Tor traffic detection.

More specifically, I read the data from the CSV type merged file and I saved it as data frame using panda library. I dropped the following irrelevant columns:

'Destination IP', 'Source IP', 'Source Port', 'Destination Port', 'Protocol'.

Next, I checked for null values and I filled in with previous row values for NaN calculation prevention. The input x was assigned as the starting data frame without the 'label' column. The output y was assigned as a single column data frame 'label' with values 'TOR' or 'nonTOR'. This process is shown in the figure 4.1.1 below:

```
import numpy as np
import pandas as pd

data = pd.read_csv("./CSV/Scenario-A/merged_5s.csv")

# check missing values
data.isnull().any().value_counts()

# fill the missing values with the previous values in that row
data.fillna(method = 'ffill', inplace = True)

# check again and confirm
data.isnull().any().value_counts()

drops = [' Destination IP', 'Source IP', ' Source Port', ' Destination Port', ' Protocol']
data.drop(drops, axis=1, inplace=True)

nullColumns = data.columns[data.isna().any()].tolist()

data[nullColumns].isna().sum()
data.shape

x = data.drop('label', axis = 1)
y = data['label']

print(x.shape, y.shape)

y = pd.DataFrame(y, columns=['label'])
print(data.head())
```

4.1.1 Python Code for Data Input and Output Separation

The data shape for x is (84194, 23) and for y is (84194,). The data features in the set after the drops are shown in the following figures 4.1.2 and 4.1.3, where only the data head is printed:

(84194, 23) (84194,)

	Flow Duration	Flow Bytes/s	Flow Packets/s	Flow IAT Mean	\
0	435	0.0	4597.701149	435.0	
1	259	0.0	7722.007722	259.0	
2	891	0.0	2244.668911	891.0	
3	1074	0.0	1862.197393	1074.0	
4	315	0.0	6349.206349	315.0	

	Flow IAT Std	Flow IAT Max	Flow IAT Min	Fwd IAT Mean	Fwd IAT Std	\
0	0.0	435	435	0.0	0.0	
1	0.0	259	259	0.0	0.0	
2	0.0	891	891	0.0	0.0	
3	0.0	1074	1074	0.0	0.0	
4	0.0	315	315	0.0	0.0	

	Fwd IAT Max	...	Bwd IAT Max	Bwd IAT Min	Active Mean	Active Std	\
0	0	...	0	0	0	0	
1	0	...	0	0	0	0	
2	0	...	0	0	0	0	
3	0	...	0	0	0	0	
4	0	...	0	0	0	0	

	Active Max	Active Min	Idle Mean	Idle Std	Idle Max	Idle Min
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0

4.1.2 Input x head

```

label
0 nonTOR
1 nonTOR
2 nonTOR
3 nonTOR
4 nonTOR

```

4.1.3 Output y head

The number of samples in the dataset is 84194. We have 84194 rows of merged samples. 69686 of them are nonTOR samples and the rest 14508 are TOR samples.

```
In [15]: print(y.groupby('label').size())
```

label	
TOR	14508
nonTOR	69686

dtype: int64

4.1.4 Number of samples

Using the python3 numpy library, the x and y can be converted from data frames to numpy arrays. The input x data frame is getting converted to a numpy array with float64 type values. The output y is getting converted to a numpy array with zeros and ones values by replacing the 'TOR' values with ones and 'nonTOR' values with zeros. In addition the input x is getting checked for infinite values and if these values exist, then they are getting replaced with zeros for preventing NaN calculations. The code for this process is shown in the figure below.

```
import tensorflow as tf
from sklearn.model_selection import train_test_split, GridSearchCV

y_train = np.asarray(y)

for i in range(0, 84194):
    if(y_train[i] == "TOR"):
        y_train[i] = 1
    else:
        y_train[i] = 0

y_train = np.asarray(y_train).astype(np.float64)

# check x_train array for inf values
assert not np.any(np.isnan(x))
x_train = np.asarray(x).astype(np.float64)
array_has_inf = np.isinf(x_train)

# replace inf values with zeros
x_train[~np.isfinite(x_train)] = 0
array_has_inf = np.isinf(x_train)

y = y_train.copy()
x = x_train.copy()
```

4.1.5 Conversions

The data preparation for x and y arrays ends with the train and test split. Neural Networks implementation requires trainable and testable data. Trainable data is the data which the neural network uses for learning. Testable data is the data which the neural network uses to test itself after learning.

The function which is used for separating the data to trainable and testable is *train_test_split* from *sklearn.model_selection* library. The test size is assigned to 30 per cent of the data and the rest 70 per cent is the data which is trainable. The split is implemented both for the input x and output y as follows.

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state=10)
```

4.1.6 Data Split for training and testing

4.1.2 CNN and RNN Extra Data Preparation

Before x_train, x_test, y_train and y_test can be used to feed neural network models, they first need to be reshaped into 3D numpy arrays, as we can see in figure 4.1.7 below.

```
x_train_cnn = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
x_test_cnn = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

y_train_cnn = y_train.reshape(y_train.shape[0], y_train.shape[1], 1)
y_test_cnn = y_test.reshape(y_test.shape[0], y_train.shape[1], 1)

x_train_rnn = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
x_test_rnn = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

y_train_rnn = y_train.reshape(y_train.shape[0], y_train.shape[1], 1)
y_test_rnn = y_test.reshape(y_test.shape[0], y_train.shape[1], 1)
```

4.1.6 Neural Networks Input and Output reshaping

4.2 CNN Model Implementation

The python code for the Convolutional Neural Network model is shown in the figure 4.2.1 and it will be explained analytically right after.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Conv1D, MaxPooling1D, Dropout, Dense

# cnn model
model = Sequential()
num_classes = 2
model.add(Conv1D(filters=64, kernel_size=3, activation='tanh', input_shape=x_train_cnn.shape[1:]))
model.add(Conv1D(filters=64, kernel_size=3, activation='tanh'))
model.add(Dropout(0.5))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100, activation='tanh'))
model.add(Dense(num_classes, activation='softmax'))
print(model.summary())

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

result = model.fit(x_train_cnn, y_train_cnn, verbose=1, epochs=10, batch_size=32,
                  validation_data=(x_test_cnn, y_test_cnn))
```

4.2.1 CNN Model Python Code

- *model = Sequential()*

Sequential function [20] provides training and inference features on this model.

- *model.add(Conv1D(filters, kernel_size, activation))*

1D convolution layer [21] creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs.

Function arguments

- *filters* “64”: The number of output filters of the convolution.
- *kernel_size* “3”: An integer specifying the length of the 1D convolution window.
- *activation* “tanh”: The activation function used.
- *input_shape* “(23, 1)”: Tuple of integers showing the input shape.

- *model.add(Dropout(0.5))*

The Dropout layer [22] randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by $1/(1 - \text{rate})$ such that the sum over all inputs is unchanged.

Function argument

- *rate* “0.5” : Float between 0 and 1. Fraction of the input units to drop.

- *model.add(MaxPooling1D(pool_size = 2))*

Max pooling [23] operation for 1D temporal data downsamples the input representation by taking the maximum value over the window defined by *pool_size*.

Function argument

- *pool_size* “2” : Integer size of the max pooling window.

- *model.add(Flatten())*

Flatten [24] function flattens the input without affecting *batch_size*.

- *model.add(Dense(units, activation))*

Dense layer [25] implements the operation:

$$\text{output} = \text{activation}(\text{dot}(\text{input}, \text{kernel}) + \text{bias})$$

where *activation* is the activation function, *kernel* is a weights matrix created by the layer, and *bias* is a bias vector created by the layer.

Function arguments

- *units* : Positive integer. Dimensionality of the output space.
- *activation* : Activation function

It is used two consecutive times in the model as follows:

```
model.add(Dense(100, activation = "tanh"))
```

```
model.add(Dense(num_classes = 2, 'softmax'))
```

- `model.compile(loss = 'sparse_categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])`

The compile method [26] configures the model for training.

Function arguments

- *SparseCategoricalCrossentropy loss* : Computes the cross entropy loss between the labels and predictions. This cross entropy loss function is useful when there are two or more label classes and the label are provided as integers.
 - *Adam optimizer* : It is the optimizer [27] that implements the Adam algorithm. It is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.
 - *Accuracy metric* : It is the metric [28] that calculates how often predictions equal labels. It creates two local variables, total and count which are used to compute the frequency with which predicted outputs match true outputs. It is an idempotent operation that divides total by count.
- `model.fit(x_train_cnn, y_train_cnn, verbose=1, epochs=10, batch_size=32, validation_data=(x_test_cnn, y_test_cnn))`

The fit method trains the model for a fixed number of iterations on a dataset.

Function arguments

- *x_train_cnn* : CNN input numpy array for training
- *y_train_cnn* : CNN output numpy array for training
- *x_test_cnn* : CNN input numpy array for testing
- *y_test_cnn* : CNN output numpy array for testing
- *Epochs 10* : 10 iterations to train the model
- *Batch_size 32* : Number of sample per gradient update
- *validation_data* : Data on which to evaluate the loss and accuracy

4.3 CNN Model Training and Testing

The Convolutional Neural Network model summary is shown in the figure 4.3.1 below. All the layers and functions that are used in the model are listed.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 21, 64)	256
conv1d_1 (Conv1D)	(None, 19, 64)	12352
dropout (Dropout)	(None, 19, 64)	0
max_pooling1d (MaxPooling1D)	(None, 9, 64)	0
flatten (Flatten)	(None, 576)	0
dense (Dense)	(None, 100)	57700
dense_1 (Dense)	(None, 2)	202

4.3.1 CNN Model Summary

So, after the neural network training we can see the 10 iteration results about the metrics accuracy and loss.

```
Train on 58935 samples, validate on 25259 samples
Epoch 1/10
58935/58935 [=====] - 23s 385us/sample - loss: 0.1968 - accuracy: 0.9205 - val_loss: 0.189
0 - val_accuracy: 0.9234
Epoch 2/10
58935/58935 [=====] - 23s 383us/sample - loss: 0.1612 - accuracy: 0.9357 - val_loss: 0.147
9 - val_accuracy: 0.9397
Epoch 3/10
58935/58935 [=====] - 23s 388us/sample - loss: 0.1530 - accuracy: 0.9387 - val_loss: 0.147
2 - val_accuracy: 0.9376
Epoch 4/10
58935/58935 [=====] - 23s 398us/sample - loss: 0.1486 - accuracy: 0.9396 - val_loss: 0.141
8 - val_accuracy: 0.9445
Epoch 5/10
58935/58935 [=====] - 25s 426us/sample - loss: 0.1434 - accuracy: 0.9417 - val_loss: 0.141
5 - val_accuracy: 0.9393
Epoch 6/10
58935/58935 [=====] - 27s 467us/sample - loss: 0.1449 - accuracy: 0.9392 - val_loss: 0.130
2 - val_accuracy: 0.9446
Epoch 7/10
58935/58935 [=====] - 26s 449us/sample - loss: 0.1420 - accuracy: 0.9424 - val_loss: 0.129
4 - val_accuracy: 0.9449
Epoch 8/10
58935/58935 [=====] - 27s 456us/sample - loss: 0.1421 - accuracy: 0.9421 - val_loss: 0.150
0 - val_accuracy: 0.9437
Epoch 9/10
58935/58935 [=====] - 27s 466us/sample - loss: 0.1437 - accuracy: 0.9411 - val_loss: 0.150
5 - val_accuracy: 0.9337
Epoch 10/10
58935/58935 [=====] - 27s 459us/sample - loss: 0.1396 - accuracy: 0.9416 - val_loss: 0.123
8 - val_accuracy: 0.9511
```

4.3.2 CNN Model Training and Testing Results

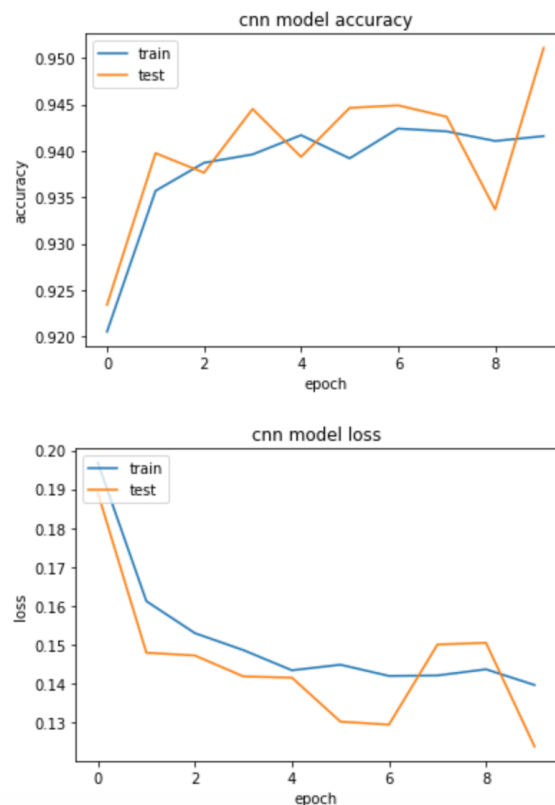
We can notice that we have 4 metrics for the network evaluation. The 'accuracy' is the training accuracy and the 'val_accuracy' is the testing accuracy of the model. The 'loss' metric is the training loss and the 'val_loss' metric is the testing loss of the model. In 10th iteration we can see that we have reach the maximum 'val_accuracy' value of all the previous iterations. More analytically, the values of the 10th iteration are:

val_accuracy = 0.9511, accuracy = 0.9416, val_loss = 0.123, loss = 0.1396

These numbers mean that the model presented a training accuracy of 94,16 % on train dataset and a validation accuracy of 95,11 % in test dataset. The model predicted right 95,11% of the testing samples during validation.

Below we can see the plot of accuracy on the training and validation datasets over training epochs (iterations). In addition, in the same figure we can see the plot of loss on the training and validation datasets over training epochs (iterations).

`dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])`



4.3.3 CNN Model Accuracy and Loss

The two previous plots have been generated using Python's Matplotlib library after the CNN training and testing process and will be explained analytically in the following paragraphs.

In the CNN model accuracy plot, we can notice that the training accuracy (blue colour line) fluctuates around the same value after the first iteration and for the remaining of the iterations. The value change looks a lot smoother than the value change of the testing accuracy (orange colour line), which displays more abrupt changes. Testing accuracy value reaches the maximum point in the last iteration of the validation, in which we can see the orange line reaching and slightly getting over 0.950. It is important to be mentioned, that both accuracies start from a lower value and abruptly increase.

In the CNN model loss plot, we can notice that the training loss continuously fluctuates above the testing loss until the 6th iteration, in which the testing loss (orange colour line) gets over the training loss (blue colour line). This happens until 8th iteration, in which testing loss seems to subside. Testing loss reaches its minimum value in the last iteration. Just as in the model accuracy plot, it is clear that testing metrics behave in a more abrupt way. It is important to be mentioned, that both losses start from a higher value and abruptly decrease.

During the Data Preparation and CNN model implementation, many methods with different parameters, different functions and arguments were tested. I ended up with the best model of the ones that were implemented during this testing period.

4.4 RNN Model Implementation

The python code for the Recurrent Neural Network model is shown in the figure 4.4.1 and it will be explained analytically right after.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Dense, LSTM

# rnn model
model = Sequential()
initializer = tf.keras.initializers.GlorotUniform()

model.add(LSTM(128, input_shape=(x_train_rnn.shape[1:]), activation='tanh', return_sequences=True, kernel_initializer=initializer))
model.add(Dropout(0.2))

model.add(LSTM(64, input_shape=(x_train_rnn.shape[1:]), activation='tanh', return_sequences=True, kernel_initializer=initializer))
model.add(Dropout(0.2))

model.add(LSTM(32, input_shape=(x_train_rnn.shape[1:]), activation='tanh', return_sequences=False, kernel_initializer=initializer))
model.add(Dropout(0.2))

num_classes = 2;
model.add(Dense(num_classes, activation='softmax', kernel_initializer=initializer))
print(model.summary())

opt = tf.keras.optimizers.Adam(lr = 1e-3, decay = 1e-5)
model.compile(loss='sparse_categorical_crossentropy', optimizer=opt, metrics=['accuracy'])

result = model.fit(x_train_rnn, y_train_rnn, epochs = 30, validation_data=(x_test_rnn, y_test_rnn))
```

4.3.3 RNN Model Python Code

1. *model = Sequential()*

Sequential function provides training and inference features on this model.

2. *Initializer = tf.keras.initializers.GlorotUniform()*

Initializers [29] define the way to set the initial random weights of keras layers. Especially, GlorotUniform initializer is also called Xavier uniform initializer and draws samples from a uniform distribution within $[-limit, limit]$, where $limit = \sqrt{6 / (fan_in + fan_out)}$, where fan_in is the number of input units in the weight tensor and fan_out is the number of output units.

3. `model.add(LSTM(128, input_shape=x_train_rnn.shape[1:]), activation='tanh', return_sequences=True, kernel_initializer=initializer)`

LSTM layer or Long Short Term Memory layer [30]

Function arguments

- *units* “128”: Dimensionality of the output space.
- *activation* “tanh”: The activation function used.
- *input_shape* “(23, 1)”: Tuple of integers showing the input shape.
- *return_sequences* “True”: Choice to return the last output in the output sequence.
- *kernel_initializer* “GlorotUniform”: Initializer for the kernel weights matrix, used for the linear transformation of the inputs.

4. `model.add(Dropout(0.2))`

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by $1/(1 - \text{rate})$ such that the sum over all inputs is unchanged.

Function argument

- *rate* “0.2”: Float between 0 and 1. Fraction of the input units to drop.

5. `model.add(LSTM(64, input_shape=x_train_rnn.shape[1:]), activation='tanh', return_sequences=True, kernel_initializer=initializer)`

LSTM layer or Long Short Term Memory layer

Function arguments

- *units* “64”: Dimensionality of the output space.
- *activation* “tanh”: The activation function used.
- *input_shape* “(23, 1)”: Tuple of integers showing the input shape.

- *return_sequences* “True” : Choice to return the last output in the output sequence.
- *kernel_initializer* “GlorotUniform” : Initializer for the kernel weights matrix, used for the linear transformation of the inputs.

6. *model.add(Dropout(0.2))*

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by $1/(1 - \text{rate})$ such that the sum over all inputs is unchanged.

Function argument

- *rate* “0.2” : Float between 0 and 1. Fraction of the input units to drop.

7. *model.add(LSTM(32, input_shape=x_train_rnn.shape[1:]), activation='tanh', return_sequences=False, kernel_initializer=initializer)*

LSTM layer or Long Short Term Memory layer

Function arguments

- *units* “32” : Dimensionality of the output space.
- *activation* “tanh” : The activation function used.
- *input_shape* “(23, 1)” : Tuple of integers showing the input shape.
- *return_sequences* “False” : Choice not to return the last output in the output sequence.
- *kernel_initializer* “GlorotUniform” : Initializer for the kernel weights matrix, used for the linear transformation of the inputs.

8. *model.add(Dense(units, activation))*

Dense layer implements the operation:

$$\text{output} = \text{activation}(\text{dot}(\text{input}, \text{kernel}) + \text{bias})$$

where activation is the activation function, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer.

Function arguments

- *units* : Positive integer. Dimensionality of the output space.
- *activation* : Activation function
- *kernel_initializer* : Kernel weights matrix initializer

It is used in the model as follows:

```
model.add(Dense(num_classes = 2, activation='softmax',  
kernel_initializer=tf.keras.initializers.GlorotUniform()))
```

9. *opt = tf.keras.optimizers.Adam(lr = 1e-3, decay = 1e-5)*

Setting Adam as the model optimizer.

Function arguments

lr (learning rate) "1e-3" : floating point value

decay (decay rate) "1e-5" : floating point value

10. *model.compile(loss = 'sparse_categorical_crossentropy', optimizer =
opt(9), metrics = ['accuracy'])*

The compile method configures the model for training.

Function arguments

- *SparseCategoricalCrossentropy loss* : Computes the cross entropy loss between the labels and predictions. This cross entropy loss function is useful when there are two or more label classes and the label are provided as integers.

- *Adam optimizer* : It is the optimizer that implements the Adam algorithm. It is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.
- *Accuracy metric* : It is the metric that calculates how often predictions equal labels. It creates two local variables, total and count which are used to compute the frequency with which predicted outputs match true outputs. It is an idempotent operation that divides total by count.

11. `model.fit(x_train_rnn, y_train_rnn, epochs=30, batch_size=32 (default), validation_data=(x_test_rnn, y_test_rnn))`

The fit method trains the model for a fixed number of iterations on a dataset.

Function arguments

- *x_train_rnn* : RNN input numpy array for training
- *y_train_rnn* : RNN output numpy array for training
- *x_test_rnn* : RNN input numpy array for testing
- *y_test_rnn* : RNN output numpy array for testing
- *Epochs 30* : 30 iterations to train the model
- *Batch_size 32* : Number of sample per gradient update (default : 32)
- *validation_data* : Data on which to evaluate the loss and accuracy

4.5 RNN Model Training and Testing

The Recurrent Neural Network model summary is shown in the figure 4.5.1 below. All the layers and functions that are used in the model are listed.

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 23, 128)	66560
dropout (Dropout)	(None, 23, 128)	0
lstm_1 (LSTM)	(None, 23, 64)	49408
dropout_1 (Dropout)	(None, 23, 64)	0
lstm_2 (LSTM)	(None, 32)	12416
dropout_2 (Dropout)	(None, 32)	0
dense (Dense)	(None, 2)	66

4.5.1 RNN Model Summary

So, after the neural network training we can see the last 10 iteration results of a total 30 iterations, about the metrics accuracy and loss.

```
Epoch 21/30
67355/67355 [=====] - 321s 5ms/sample - loss: 0.1349 - accuracy: 0.9461 - val_loss: 0.1465
- val_accuracy: 0.9399
Epoch 22/30
67355/67355 [=====] - 316s 5ms/sample - loss: 0.1364 - accuracy: 0.9445 - val_loss: 0.1218
- val_accuracy: 0.9526
Epoch 23/30
67355/67355 [=====] - 319s 5ms/sample - loss: 0.1289 - accuracy: 0.9479 - val_loss: 0.1179
- val_accuracy: 0.9556
Epoch 24/30
67355/67355 [=====] - 320s 5ms/sample - loss: 0.1240 - accuracy: 0.9507 - val_loss: 0.1212
- val_accuracy: 0.9531
Epoch 25/30
67355/67355 [=====] - 319s 5ms/sample - loss: 0.1227 - accuracy: 0.9498 - val_loss: 0.1141
- val_accuracy: 0.9560
Epoch 26/30
67355/67355 [=====] - 313s 5ms/sample - loss: 0.1222 - accuracy: 0.9509 - val_loss: 0.1178
- val_accuracy: 0.9524
Epoch 27/30
67355/67355 [=====] - 314s 5ms/sample - loss: 0.1236 - accuracy: 0.9499 - val_loss: 0.1155
- val_accuracy: 0.9553
Epoch 28/30
67355/67355 [=====] - 325s 5ms/sample - loss: 0.1198 - accuracy: 0.9517 - val_loss: 0.1148
- val_accuracy: 0.9581
Epoch 29/30
67355/67355 [=====] - 317s 5ms/sample - loss: 0.1201 - accuracy: 0.9515 - val_loss: 0.1127
- val_accuracy: 0.9545
Epoch 30/30
67355/67355 [=====] - 313s 5ms/sample - loss: 0.1161 - accuracy: 0.9527 - val_loss: 0.1132
- val_accuracy: 0.9539
```

4.5.2 RNN Model Training and Testing Results

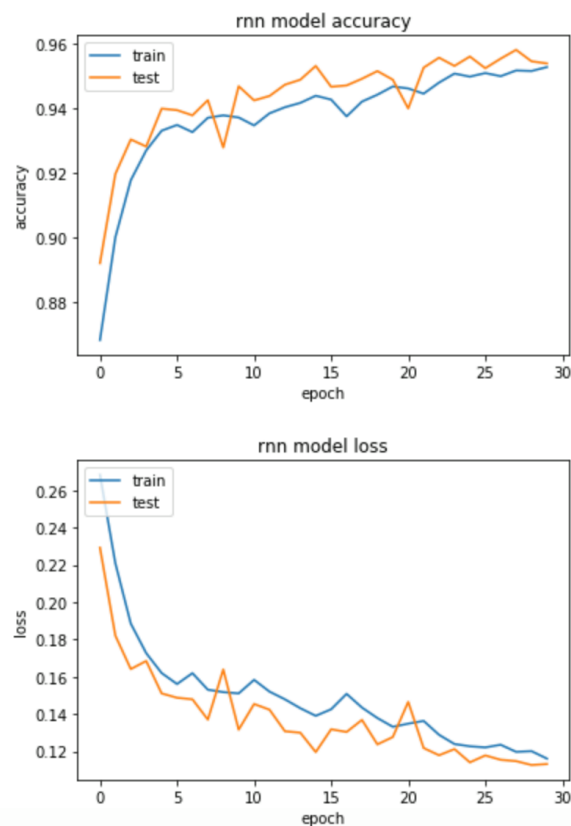
We can notice that we have 4 metrics for the network evaluation. The ‘accuracy’ is the training accuracy and the ‘val_accuracy’ is the testing accuracy of the model. The ‘loss’ metric is the training loss and the ‘val_loss’ metric is the testing loss of the model. In 10th iteration we can see that we have reach the maximum ‘val_accuracy’ value of all the previous iterations. More analytically, the values of the 28th iteration are:

val_accuracy = 0.9581, accuracy = 0.9517, val_loss = 0.1148, loss = 0.1198

These numbers mean that the model presented a training accuracy of 95,17 % on train dataset and a validation accuracy of 95,81 % in test dataset. The model predicted right 95,81% of the testing samples during validation.

Below we can see the plot of accuracy on the training and validation datasets over training epochs (iterations). In addition, in the same figure we can see the plot of loss on the training and validation datasets over training epochs (iterations).

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```



4.5.3 RNN Model Accuracy and Loss

The two previous plots have been generated using Python's Matplotlib library after the RNN training and testing process and will be explained analytically in the following paragraphs.

In the RNN model accuracy plot, we can notice that both training (blue colour line) and testing (orange colour line) accuracies start from a lower value of approximately 0.85 and increase abruptly in the very first iterations. Then, both accuracies gradually increase for the remaining 25 iteration of a total 30 iterations. In that time, we can notice that testing accuracy presents exactly two cases, in which reaches values below training accuracy, during a small amount of time though. Beyond these two cases, testing and training accuracy fluctuate in a same way above 0.94. Testing accuracy reaches its maximum value of slightly above 0.95 in 28th iteration of the validation process.

In the RNN model loss plot, we can notice that both training (blue colour line) and testing (orange colour line) start from a higher value and decrease significantly during the first 5 iterations. After the first 5 and for the remaining 25 iterations both losses gradually decrease. In the majority of the time, testing accuracy stays below training accuracy, except for the two cases which match the other two cases in accuracy plot in an aspect of time, in which we notice that testing loss gets above training loss. Testing loss gets its minimum value slightly before the last iteration.

During the Data Preparation and RNN model implementation, many methods with different parameters, different functions and arguments were tested. I ended up with the best model of the ones that were implemented during this testing period.

4.6 CNN and RNN Models Comparison

Now that training and testing of both CNN and RNN have been completed, we can make some observations in terms of comparison. This thesis' Recurrent Neural Network model is slightly more accurate than the Convolutional. As we can notice validation accuracy for the RNN reaches 95.81%, while CNN reaches a validation accuracy of 95.11%. Being two completely different models in terms of structure and functionality, they had to train in a different iteration number. This thesis' CNN is a simpler and smaller model than this thesis' RNN. For CNN to reach its maximum accuracy, seemed better to be trained only for 10 epochs. In addition the training and validation duration for each iteration was very little comparing to the RNN model, resulting to a very little total duration for training and testing. For RNN model to reach its maximum accuracy, seemed better to be trained for 30 epochs. Every iteration was much more time consuming, resulting to a very big total duration for training and testing. Below in the figure 4.6.1 we can see the comparisons with actual numbers and values.

	Learning Iterations	Iteration Duration	Total Duration	Accuracy
CNN	10	20 sec	200 sec	95.11%
RNN	30	300 sec	9000 sec	95.81%

4.6.1 CNN and RNN Comparison

CHAPTER 5

Conclusions

5.1 Conclusion

In the final step of this thesis, it is important to be mentioned that we implemented two high accuracy neural models, which are able to detect Tor network flows in an efficient way. The most accurate Neural Network model that were implemented is RNN, which slightly overcame CNN's accuracy. In a more general way, we used Convolutional and Recurrent logic to contribute in a cyber security tool, which achieves binary classification of Tor and nonTor connections.

5.2 Future Work

For future work, the Tor issue can be approached as a detection issue, using convolutional and recurrent methods in a different classification problem. This classification will focus on characterisation of Tor Traffic in the identification of applications within Tor traffic. For example browsing, file transfer, audio, video streaming, email applications etc. This type of classification has already been implemented using other types of neural networks and deep learning algorithms, that differ from convolutional and recurrent.

Bibliography

1. Deep Learning Spreads: <https://semiengineering.com/deep-learning-spreads/>
2. Deep Learning By MARSHALL HARGRAVE: <https://www.investopedia.com/terms/d/deep-learning.asp>
3. Artificial Intelligence vs Machine Learning vs Deep Learning: <https://medium.com/datadriveninvestor/artificial-intelligence-vs-machine-learning-vs-deep-learning-8ade79eed0cb>
4. Wikipedia: https://en.wikipedia.org/wiki/Deep_learning
5. Deep Neural Networks: <https://www.kdnuggets.com/2020/02/deep-neural-networks.html>
6. Convolutional Neural Networks cheatsheet By Afshine Amidi and Shervine Amidi: <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks>
7. Recurrent Neural Networks cheatsheet By Afshine Amidi and Shervine Amidi: <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks>
8. Understanding LSTM Networks: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>
9. Wikipedia: https://en.wikipedia.org/wiki/Long_short-term_memory#/media/File:The_LSTM_cell.png
10. 5 Amazing Applications of Deep Learning in Cybersecurity: <https://www.infocycle.com/blog/2019/08/13/5-amazing-applications-of-deep-learning-in-cybersecurity/>
11. Everything You Need To Know About Cyber Security Attacks And How To Prevent Them: <https://www.mygreatlearning.com/blog/types-of-cyber-attacks-and-why-cybersecurity-is-important/>
12. Wikipedia: [https://en.wikipedia.org/wiki/Tor_\(anonymity_network\)](https://en.wikipedia.org/wiki/Tor_(anonymity_network))
13. Tor: <https://www.torproject.org/>

14. Using the Power of Deep learning for Cyber Security (Part 1): <https://www.analyticsvidhya.com/blog/2018/07/using-power-deep-learning-cyber-security/>
15. The Onion Ransomware (Encryption Trojan): <https://www.kaspersky.co.in/resource-center/threats/onion-ransomware-virus-threat>
16. Keras vs Tensorflow: Must Know Differences!: <https://www.guru99.com/tensorflow-vs-keras.html>
17. TensorFlow: A Flexible, Scalable & Portable System: <https://www.infoq.com/presentations/tensorflow/>
18. DEEP LEARNING WITH PYTHON: <http://junyelee.blogspot.com/2018/01/deep-learning-with-python.html>
19. Habibi Lashkari A., Drapel Gil G., Mamun M. And Ghorbani a., “[Characterization of Tor Traffic using Time based Features](#)”, Proceedings of the 3rd International Conference on Information Systems Security and Privacy - Volume 1, pages 253-262, 2017.
20. The Sequential class: <https://keras.io/api/models/sequential/>
21. Conv1D layer: https://keras.io/api/layers/convolution_layers/convolution1d/
22. Dropout layer: https://keras.io/api/layers/regularization_layers/dropout/
23. MaxPooling1D layer: https://keras.io/api/layers/pooling_layers/max_pooling1d/
24. Flatten layer: https://keras.io/api/layers/reshaping_layers/flatten/
25. Dense layer: https://keras.io/api/layers/core_layers/dense/
26. Model training APIs: https://keras.io/api/models/model_training_apis/
27. Adam : <https://keras.io/api/optimizers/adam/>
28. Accuracy Metrics : https://keras.io/api/metrics/accuracy_metrics/#accuracy-class
29. Layer weight initializers: <https://keras.io/api/layers/initializers/>
30. LSTM layer: https://keras.io/api/layers/recurrent_layers/lstm/