UNIVERSITY OF THESSALY

DIPLOMA THESIS

Intrusion Detection using Neural Networks

Author:
Ioannis KOSTIKAS

Supervisors:
Christos D. ANTONOPOULOS
Spyros LALIS
Nikolaos BELLAS

A thesis submitted in fulfillment of the requirements for the degree of Diploma

in the

Department of Electrical and Computer Engineering

Volos, 2020

UNIVERSITY OF THESSALY

Abstract

Department of Electrical and Computer Engineering

Diploma

Intrusion Detection using Neural Networks

by Ioannis KOSTIKAS

As the use of computer systems and smart devices continues to increase, so does their appeal as attack targets, and in turn the importance of defending against such attacks. Intrusion Detection Systems are an essential part of a good defensive strategy and this Thesis explores the effectiveness of neural networks for that purpose, with a focus on how "Long Short-Term Memory" (LSTM) Neural Networks perform in this task compared against Feed-Forward Neural Networks.

We carefully prepared and processed three network intrusion detection datasets based on the same packet captures, two of which we created for this Thesis. We used them to train Feed-Forward and LSTM neural network binary and multi-class classifiers and autoencoders. We also trained additional classifiers on reduced versions of the datasets which we produced using the autonencoders. In total, in this Thesis, we describe fifteen models.

Ultimately, the LSTMs we trained on a sequence dataset we created for this Thesis from packet headers, are shown to outperform Feed-Forward Neural Networks trained on flow-based datasets. Futhermore, binary classifiers perform better than multi-class classifiers, while both outperform autoenoders. The best model, an LSTM binary classifier, reaches an "Area Under the Curve" (AUC) of 99.9996%, an F_1 Score of 99.847% and a "False Positive Rate" (FPR) of 0.04%.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Περίληψη

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Ανίχνευση εισβολών με νευρωνικά δίκτυα

Ιωάννης Κωστίκας

Όσο η χρήση συστημάτων υπολογιστών και έξυπνων συσκευών συνεχίζει να αυξάνεται, τόσο αυξάνεται η ελκυστικότητά τους σαν στόχων επίθεσης και σαν αποτέλεσμα η σημασία άμυνας εναντίον τέτοιων επιθέσεων. Τα Συστήματα Ανίχνευσης Εισβολής αποτελούν ένα σημαντικό μέρος μιας καλής αμυντικής στρατηγικής. Η παρούσα εργασία ερευνά την αποτελεσματικότητά των νευρωνικών δικτύων για αυτό το σκοπό, με την έμφαση να δίνεται στην επίδοση των "Long Short-Term Memory" (LSTM) δικτύων σε σχέση με αυτή των Feed-Forward δικτύων.

Με μεγάλη προσοχή επεξεργαστήκαμε τρία dataset βασισμενα στα ίδια packet captures, δυο εκ των οποίων δημιουργήσαμε για αυτή την εργασία. Πάνω σε αυτά εκπαιδεύσαμε "Feed-Forward" και "Long Short-Term Memory" (LSTM) δίκτυα για binary και multi-class classification καθώς και autoencoders. Επιπλέον εκπαιδεύσαμε και μοντέλα πάνω σε datasets που δημιουργήσαμε χρησιμοποιώντας τους autoencoders. Συνολικά σε αυτή την εργασία περιγράφουμε 15 μοντέλα.

Τελικά τα LSTMs που εκπαιδεύσαμε σε features που δημιουργήσαμε για αυτή την εργασία από headers πακέτων, αποδεικνύονται καλύτερα σε σχέση με Feed-Forward δίκτυα που εκπαιδεύσαμε σε flow-based dataset. Επίσης, οι binary classifiers αποδεικνύονται καλύτεροι των multi-class, ενώ οι autoencoders έχουν χειρότερη απόδοση και από τα δυο. Το καλύτερο μοντέλο λοιπόν είναι ένα LSTM binary classifier με "Area Under the Curve" (AUC) 99.9996%, F_1 Score 99.847% και "False Positive Rate" (FPR) 0.04%.

Acknowledgements

First and foremost I wish to thank my supervisor, Prof. Christos D. Antonopoulos, for his guidance and for giving me the opportunity to work on such an interesting project.

I would also like to thank my family for their love and support.

Contents

List of Figures

List of Tables

List of Abbreviations

AE	**A**uto **E**ncoder
AI	**A**rtificial **I**ntelligence
AUC	**A**rea **U**nder the ROC **C**urve
BCE	**B**inary **C**ross **E**ntropy
CE	**C**ross **E**ntropy
DDoS	**D**istributed **D**enial **O**f **S**ervice
DL	**D**eep **L**earning
DNS	**D**omain **N**ame **S**ystem
DoS	**D**enial **o**f **S**ervice
FDR	**F**alse **D**iscovery **R**ate
FL	**F**ocal **L**oss
FN	**F**alse **N**egative
FP	**F**alse **P**ositive
FPR	**F**alse **P**ositive **R**ate
IDS	**I**ntrusion **D**etection **S**ystem(s)
IP	**I**nternet **P**rotocol
LSTM	**L**ong **S**hort-**T**erm **M**emory
MAE	**M**ean **A**bsolute **E**rror
ML	**M**achine **L**earning
MSE	**M**ean **S**quared **E**rror
NN	**N**eural **N**etwork
OSI	**O**pen **S**ystems **I**nterconnection
PCAP	**P**acket **Cap**ture
PCC	**P**earson **C**orrelation **C**oefficient
PPMCC	**P**earson **P**roduct-**M**oment **C**orrelation **C**oefficient
PPV	**P**ositive **P**redictive **V**alue
ReLU	**Re**ctified **L**inear **U**nit
RNN	**R**ecurrent **N**eural **N**etwork
ROC	**R**eceiver **O**perating **C**haracteristic
TCP	**T**ransmission **C**ontrol **P**rotocol
TN	**T**rue **N**egative
TNR	**T**rue **N**egative **R**ate
TP	**T**rue **P**ositive
TPR	**T**rue **P**ositive **R**ate
UDP	**U**ser **D**atagram **P**rotocol

Chapter 1

Introduction

1.1 Problem Statement

In the present day and age computer systems have become a permanent fixture as they are indispensable in their use in business and everyday life. However, this ubiquitousness makes them attractive targets for malicious actors.

Indeed, in the past couple of years there have been major security breaches, such as:

- In 2016, DNS provider Dyn was subjected to a distributed denial of service attack (DDoS) by a botnet, which impacted service of major websites [1].
- In 2017, credit reporting agency Equifax suffered a breach in which the data of approximately 148 million Americans was stolen, costing them a reported 1.4 billion dollars [2, 3].
- Also in 2017, the WannaCry ransomware attack, using stolen exploits developed by the NSA, infected more than 200,000 computers in 150 countries with costs estimated from hundreds of millions to billions of dollars [4].
- In 2018 popular online Q&A platform Quora was compromised by a malicious third party getting access to the information of approximately 100 million users [5].
- Over a period of 4 years, from 2014 to 2018, hackers accessed the information of 327 million reservations of hotel megachain Marriott [6].

It should be clear that improving defenses to such attacks is more important than ever, especially with the tendency for an increasing number of devices to be "smart" and networked (Internet of Things), providing even more attack targets.

Intrusion Detection Systems (IDSs) are an essential part of a good defense strategy against cyberattacks and an interesting area of research. They work by monitoring the network and hosts and use various methods, including signature-based methods and stateful protocol analysis, to detect threats. Due to the major successes of machine learning and neural networks in particular, in a variety of domains such as image classification [7], text generation [8], machine translation and game playing [9] to name a few, there has been interest in applying those techniques to improve IDSs.

1.2 Contributions

This Thesis focuses on exploring how different neural network architectures perform in the task of network intrusion detection, when applied to a modern dataset.

We select a flow-based network intrusion detection dataset, which is also accompanied by the packet captures it was created from. From those, we extract the packet headers and use them to derive features for a sequence dataset avoiding the need to inspect the potentially encrypted packet payloads. We also process those features into another flow-based dataset for fairer comparisons.

Then, we train and evaluate feed-forward neural networks on the flow-based datasets and LSTM neural networks on the sequence dataset and we experiment with different configurations.

Finally, we compare the performance of the LSTM and feed-forward neural networks and find that LSTMs perform better and do so consistently across all tasks.

1.3 Software

We made use of many pieces open-source of software during the completion this Thesis. For drawing neural networks we used "graph visualization software" Graphviz [10] and the Python package Daft [11], which "uses matplotlib to render probabilistic graphical models". For the heatmaps of the correlation matrices we used Seaborn [12], a "Python data visualization library", also based on maplotlib, and last but not least, we used Matplotlib [13] itself, which is "a comprehensive library for creating static, animated, and interactive visualizations in Python", for everything else.

Pandas [14, 15], a Python "data analysis and manipulation tool" proved invaluable for data manipulation, exploration and (pre)processing, as did Numpy [16, 17], "the fundamental package needed for scientific computing with Python". We also used scikit-learn [18], which is "a Python module for machine learning built on top of SciPy", for data preprocessing and calculating all the metrics for our results.

Finally, when it came to creating and training all the neural network models, we used Keras [19], "a deep learning API written in Python, running on top of the machine learning platform TensorFlow [20]" and TensorFlow itself, which is "an end-to-end open source platform for machine learning".

1.4 Thesis outline

The Thesis is structured as follows,

- Chapter 2, *Background & Literature Review*, is focused on providing necessary background information for this Thesis, starting from an overview of IDS types, continuing to describe relevant machine and deep learning concepts, as well as networking concepts, with an emphasis on the protocols' packet header structures. Finally, it finishes with a review of the relevant literature.

- Chapter 3, *Dataset Selection & Analysis*, first describes the process of selecting a dataset, followed by an in-depth description of the one selected. Additionally, the process of creating a sequence dataset and another flow-based one is described. Finally, the sequence dataset is thoroughly explored, the original flow-based one is cleaned, and the initial split to training and test sets of the datasets is done.

- Chapter 4, *Experimental Neural Network Model Evaluation for Intrusion Detection*, starts with an outline of data preprocessing and continues with how it was applied to the datasets used in this Thesis. Finally, it explores the best-performing

models that were trained for multi-class and binary classification and anomaly detection and compares them.

- Chapter 5, *Conclusions & Future Work*, concludes the Thesis with a summary of the findings and the process that was followed to discover them, and suggestions for future work.

Chapter 2

Background & Literature Review

2.1 Intrusion Detection Systems

The purpose of Intrusion Detection Systems (IDSs) is to identify potential intrusions or attacks, be they internal or external, against computer systems or networks. They achieve that goal by monitoring said networks and/or systems and analyzing the information they obtain for indicators of possible security incidents.

IDSs are broadly classified into two categories based on where they run,

- **Host Intrusion Detection Systems (HIDS)** that run on the hosts themselves and monitor them and their behaviour (e.g. system logs, running daemons and services, processes that are being executed, modified and created files etc.)
- **Network Intrusion Detection Systems (NIDS)** that are placed at strategic network points, such as network boundaries [21], and monitor ingress and egress network traffic or even traffic that doesn't cross network boundaries.

They can additionally be classified as signature-/misuse- based or anomaly-based by examining their analysis strategy.

- **Signature-based or Misuse-based detection** works by inspecting the information available to the IDS for matches against known threat signatures stored in a signature database. A signature in this context is an identifying pattern e.g. strings known to be present in a malicious binary.

  An issue inherent to this approach is that it lacks protection against new and unknown threats, as no signatures exist for them yet, as well as requiring the signature database be kept up-to-date.

- **Anomaly-based detection** assumes anomalous behaviour is likely malicious, so available information is monitored for abnormalities. This requires profiles of known good behaviour to be compared against, which are developed and updated by monitoring behaviour during normal activity.

  This strategy, contrary to signature based methods, is capable of detecting new and unknown threats, but can also have a high false alarm rate (i.e. frequently misidentify normal activity as malicious).

2.2 Machine Learning

Broadly, Machine Learning (ML) is a sub-branch of Artificial Intelligence (AI), that focuses on computer algorithms that learn to perform tasks without the need to be explicitly programmed; instead, they are trained on data from which they learn to

extrapolate. Furthermore, by being supplied with more data their performance usually improves.

For example, suppose one is trying to write an algorithm to identify pets from pictures, one solution would be trying to develop rules and heuristics for that task and write a program using them, however this, after taking into account all the variables, such as different animals and breeds, various fur colours, obstructions by other objects and different positions in the frame to name but a few, is a very difficult problem.

Instead, a machine learning algorithm/model can be trained on a dataset containing many pictures of pets, each labeled with the name of the animal present in it, and automatically learn to identify them.

Following are some useful concepts related to machine learning.

2.2.1 Machine Learning Dataset

A dataset tends to be a collection of observations $\vec{x} \in \Re^n$, also known as samples, examples or instances, with a number features x_i, also sometimes called variables, or attributes. Each observation is potentially associated with a label y.

For example, the well known "Iris Data Set" [22, 23] contains 150 observations about three species of iris flowers, each with 4 features, *Sepal Length in cm*, *Sepal Width in cm*, *Petal Length in cm*, *Petal Width in cm*, plus the class label y, which identifies the species of plant described by each observation.

In other words, for this dataset, $\vec{x} \in \Re^4$, with $x_1 =$ *"Sepal Length in cm"*, $x_2 =$ *"Sepal Width in cm"* etc. Every instance has values for each of the features e.g. for the first instance, $\vec{x_1} = [5.1, 3.5, 1.4, 0.2]$, though some may be missing and marked as such.

2.2.2 Types of Algorithms by learning method

Machine Learning algorithms learn from data and can be split into different categories based on how they learn, such as supervised, unsupervised and reinforcement learning. This Thesis concerns itself with the former two which are described below.

Supervised Learning

Supervised learning algorithms learn by training on labeled examples, that is to say, a given dataset includes both features $\mathbf{X}$ i.e. the input data, and the labels Y i.e. the desired outputs for said input data, and the algorithm, by training on input-output pairs $\{(\vec{x}, y) | \vec{x} \in \mathbf{X}, y \in Y\}$ tries to infer a function f such that $Y \approx f(\mathbf{X})$, which can then be used to make predictions about previously unseen data.

Based on whether the label takes discrete or continuous values, supervised learning problems can be further grouped into *classification* and *regression* problems respectively. For example, the problem of identifying the kind of pet described in Section 2.2 would be a multi-class classification problem, whereas trying to predict the value of a car in euros given some characteristics would be a regression problem. Finally, classification can be further divided into binary and multi-class classification if the number of values the label may take is two or more respectively.

Unsupervised Learning

In contrast to supervised learning algorithms, unsupervised ones are trained on datasests that are not labeled and usually try to uncover hidden patterns in the data. Since labeling data is often time intensive and expensive, requiring human labour, the ability to do without can be a big advantage. An example of unsupervised learning is clustering, where instances are grouped into clusters based on some measure of similarity.

2.2.3 Anomaly detection

Anomaly detection is the identification of observations, commonly referred to as anomalies, or outliers, that diverge from expected behaviour. Such anomalies may require special attention as they could be indicative of problems, such as atypical network traffic due to a security breach, or unusual transactions due to credit card fraud. It usually differs from supervised learning, and is characterized by, having a very small number of positive examples i.e. anomalies available to train on, e.g. a corporate network may have never been breached before, legitimate transactions are much more prevalent than fraudulent ones etc.

If the available data is labeled, anomaly detection can be treated as a binary classification problem with severe class imbalance. On the other hand, if it is unlabeled, the algorithm is trained under the assumption that the majority of observations belong to the "normal" class, therefore, observations that are significantly different from said majority are identified as anomalous, for example, using the distance of a point (observation) from its k^{th} nearest neighbor as its outlier score [24].

2.2.4 Training, Validation and Test set

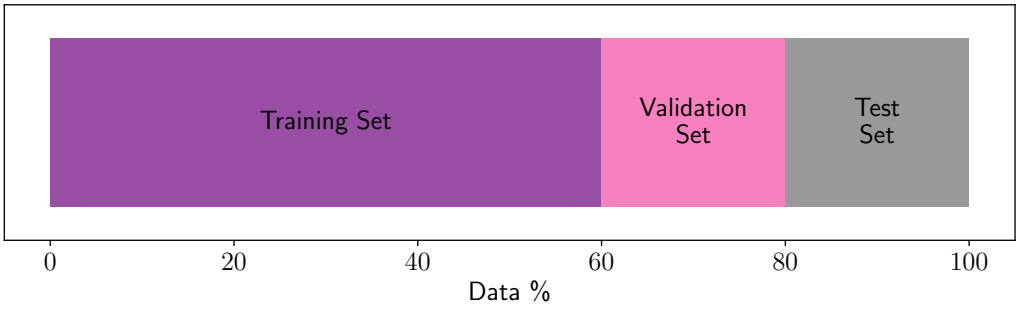

FIGURE 2.1: An illustration of splitting data into training, validation and test sets

The goal of training a machine learning algorithm is to use the trained model to make predictions about previously unseen data, called generalization, i.e. the algorithm should learn from the data it was presented with during the training phase and be capable of generalizing to unseen data and making accurate predictions about it.

In order to be able to reason about a model's performance, before training it on a dataset, the dataset should be split into a training, a validation and a test set.

The training set, as the name suggests, is used to train the model and fit its parameters, e.g. the weights and biases of the neurons in a neural network, whereas

the validation set is used for model selection, i.e. it is used to select the optimal hyperparameters for the model, for example, the number of hidden layers and neurons for a neural network, or the learning rate for gradient descent.

Finally, the test set is used as a proxy for new, unseen data i.e. data the model did not see during training, to allow for an unbiased evaluation of the model's performance, and should **only** be used with the final model.

If, instead, the test set is used many times to select the "best" model, the test set error of the chosen model will likely underestimate the true test error, sometimes significantly [25].

2.2.5 Underfitting, Overfitting & Model capacity

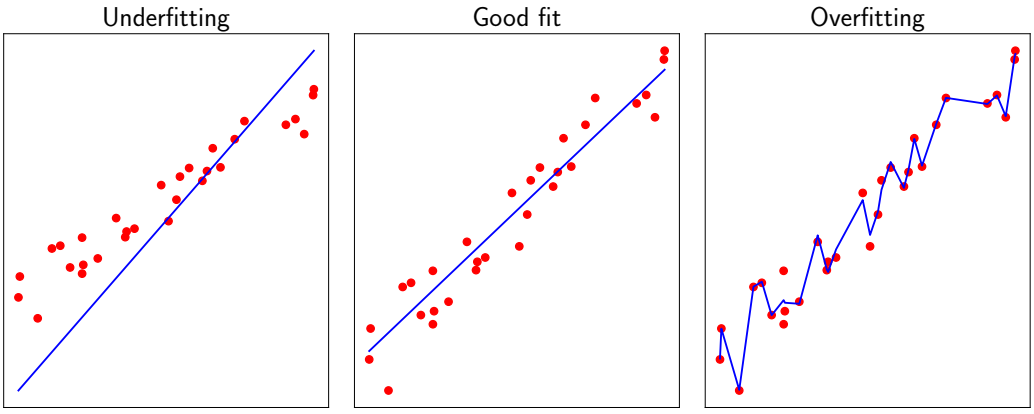

FIGURE 2.2: An example demonstrating the differences between underfitting, a good fit and overfitting.

A model's capacity is a way to think of how complicated a relationship a model can learn, that is, a higher capacity model is able to fit more complex data compared to a lower capacity one, for example, a model of the form $y = \alpha x + b$, versus $y = \alpha x$.

A model is said to be underfitting when the error on the training data, the training error, is too high, usually meaning the model is not powerful enough, in other words does not have enough capacity, to fit the training data well. This can usually be fixed by increasing the model's capacity by, in the case of neural networks for example, adding more layers, or increasing the number of neurons.

On the over hand, overfitting happens when a model performs so well on the training data that it starts to fit the noise, or peculiarities, specific to the particular training dataset to the detriment of its generalization performance. In other words, it has so much capacity that it can start memorizing the training set, including fluctuations present by chance; this can be seen clearly in Figure 2.2.

2.2.6 Regularization

Regularization is the general term used to describe a variety of techniques that help prevent overfitting. They usually do that by reducing the model's effective capacity by, for example, adding a term to the loss function that penalizes large weights, this keeps the model from being too flexible and can prevent it from fitting the noise.

Some popular methods used in this Thesis are described bellow.

Dropout

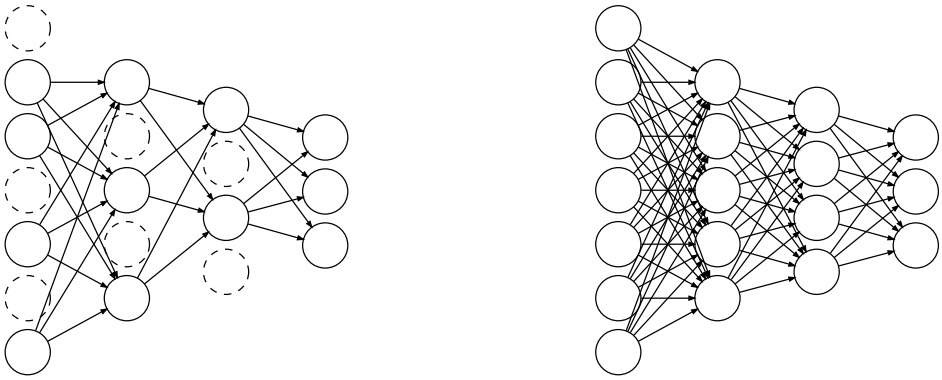

FIGURE 2.3: A simple neural network with and without dropout applied [26]

Dropout, proposed by Srivastava et al. [26] is a regularization technique for neural networks which works by randomly dropping some of the network's units/neurons, meaning their outputs are set to zero, with some probability p, usually $p \in [0.2, 0.5]$, during the training phase (see Figure 2.3). This forces neurons to learn more robust features, since they cannot depend on specific other neurons always being present.

Another way to view dropout is that not one big network, but multiple smaller networks that share weights, called "thin" networks in the paper, are trained at training time and at test time, when dropout is no longer applied, the output of the network is an approximation of the average of the predictions of the many "thin" networks (assuming neuron outputs are scaled appropriately).

Early stopping

A neural network with enough capacity will see its training error keep decreasing as training continues. However, that does not necessarily translate to a decrease in validation error, indeed, at the point where the model starts overfitting the validation error will stop improving and in fact will start increasing as the model overfits more.

Early stopping is a technique that aims to stop that problem (Figure 2.4). It works by monitoring the validation loss, logging the best value so far as it occurs and stopping training once the validation loss has not shown improvement for some time exceeding a "patience" parameter p. When the best value at the time is logged, so are the parameters that produced it, so once training stops they can be used for the final model.

2.2.7 Optimization Algorithms

Optimization algorithms are algorithms that can be used to minimize (or maximize) functions. In the case of machine learning and neural networks they are usually used to minimize the model's cost function (see Section 2.2.8).

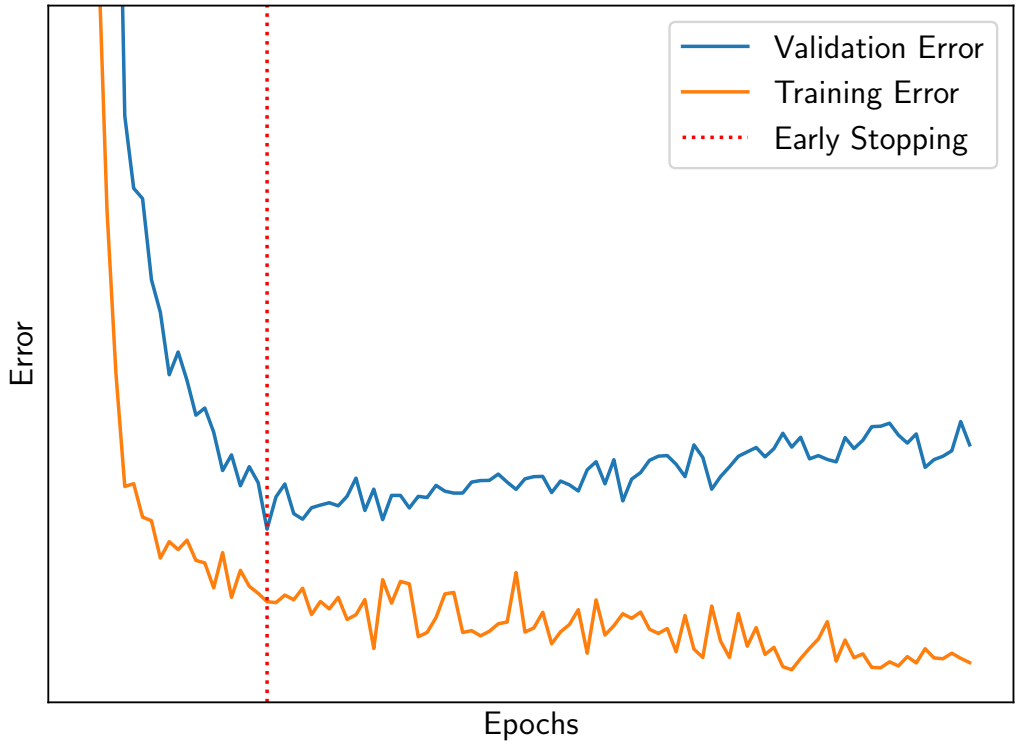

FIGURE 2.4: An example of early stopping. Training stops after p (patience) epochs have passed since the dotted line which marks the lowest validation loss.

Gradient Descent

Gradient Descent is an iterative optimization algorithm that can be used to minimize (i.e. find *local* minima of) differentiable functions. Intuitively, it works by taking small steps towards the direction of steepest descent, which can be shown to be that of the negative of the gradient, until it reaches a minimum.

In other words, given a diffirentiable function $f(x)$ parameterized by θ i.e. $f(x;\theta)$ and a learning rate λ, using gradient descent the next update for θ is given by

$$\theta_{n+1} = \theta_n - \lambda \nabla_\theta f$$

The algorithm converges once the gradient is zero, or, in practice, when its value is sufficiently small.

The learning rate λ is a hyperparameter which controls how big the step taken is. If it is too big the algorithm may overshoot the minimum and possibly keep overshooting it and not converge, or the gradient's value may explode and overflow. On the other hand, if λ is too small, the algoritm will be taking very small steps, meaning it will take a very long time to converge, or get stuck in a suboptimal local minimum.

In practice, when training neural networks, mini-batch gradient descent, or one of its variants (e.g. Adam), is used, which calculates the parameter update on a small batch of the dataset at a time, or "mini-batch", instead of using the whole dataset at once.

2.2.8 Loss/Cost Functions

A loss function assigns values or "costs" to the predictions a model makes, with wrong ones being assigned high costs and correct ones low. When training a model it is this loss function that is attempted to be minimized, which consequently means the model is making fewer and/or smaller mistakes. Depending on the type of problem e.g. regression or classification, there exist numerous loss functions, of which the ones used in this Thesis are described bellow, where y_i is the true label for the i_{th} observation, $\hat{y}_i$ is the model's prediction for that observation and n is the total number of observations.

Mean Squared Error Loss

The Mean Squared Error (MSE) loss is the average of the squared errors, defined as,

$$\textbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

and is often the loss function of choice for regression problems. Since the difference of the real and predicted labels is squared, larger errors are penalized more heavily.

Mean Absolute Error Loss

The Mean Absolute Error (MAE) loss is the average of the absolute errors, defined as,

$$\textbf{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Cross-Entropy Loss

The Cross-Entropy (CE) loss, averaged over all examples n, is defined as,

$$\textbf{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c\in C}\mathbf{I}\{y_i = c\}\log\hat{y}_{i,c}$$

where

$$\mathbf{I}\{\mathbf{P}\} = \begin{cases} 1, & \text{if } P \text{ is true} \\ 0, & \text{otherwise} \end{cases}$$

is the indicator function, C are the class labels, and $\hat{y}_{i,c} \in [0,1]$ is the probability the model assigns to observation i belonging to class c. It is often the loss function of choice for classification problems, be they binary or multi-class.

Specifically for the binary case, assuming $y_i \in \{0,1\}$, the binary CE loss, averaged over all examples n, can be defined as,

$$\textbf{BCE} = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i\log\hat{y}_i + (1-y_i)\log\left(1-\hat{y}_i\right)\right)$$

Focal Loss

The Focal Loss (FL) which was introduced by Lin et al. [27] aims at improving the performance of classifiers in the presence of extreme class imbalance by adjusting

the CE loss (see Section 2.2.8 above) so that the cost incurred by easy to classify examples is reduced. For binary classification and a single instance it is defined as,

$$FL(p_t) = -\alpha_t(1-p_t)^\gamma \log(p_t)$$

where $\gamma \geq 0$ is the "focusing parameter",

$$\mathbf{p_t} = \begin{cases} p, & \text{if } y = 1 \\ 1-p, & \text{otherwise} \end{cases}$$

and α_t is a balancing parameter defined similarly to p_t. The more confident the model is for an example, the more the term $(1-p_t)^\gamma$ downweights that example's contribution to the total loss.

2.2.9 Evaluation Metrics

		True Class	
		Positive	Negative
Predicted Class	Positive	True Positive TP	False Positive FP
	Negative	False Negative FN	True Negative TN

TABLE 2.1: Confusion Matrix

In this section the definitions of some well-known evaluation metrics are given, which can be used to measure a classifier's performance.

An easy way to summarize the performance of a binary classifier which makes predictions between two classes, for example, without loss of generality, "positive" and "negative", is through the use of a confusion matrix.

A confusion matrix is shown in Table 2.1, where a "True Positive" (*TP*) is a positive observation correctly identified as positive and similarly for "True Negative" (*TN*), whereas a "False Positive" (*FP*) is a negative observation misclassified as positive, and vice versa for "False Negative" (*FN*). Finally, P, N are the total number of positive and negative samples respectively.

The following metrics are defined with respect to those concepts:

- **Accuracy** is defined as the fraction of the number of correctly classified observations over the total number of observations i.e. what percentage of the model's predictions were correct,

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

and is perhaps the most well-known and easy to understand evaluation metric. However, it is not well-suited for use in the case of imbalanced datasets as it may misrepresent a classifier's performance, e.g. given a dataset with 10 positive and 990 negative samples, a classifier that always predicts the negative class would have an accuracy of 99% which seems very good, despite the classifier completely failing at correctly predicting the positive class.

- **Recall**, or **Sensitivity**, or **True Positive Rate (TPR)** is the ratio of correct positive predictions to total positive samples, or more simply, the percentage of positive samples correctly identified, and is defined as,

$$\text{Recall} = \frac{TP}{P} = \frac{TP}{TP + FN}$$

- **Specificity**, or **True Negative Rate (TNR)** is defined analogously to **Recall**, but for negative samples, as,

$$\text{Specificity} = \frac{TN}{N} = \frac{TN}{TN + FP}$$

- **Precision** or **Positive Predictive Value (PPV)** is the ratio of the number of observations correctly predicted positive, over the total number of positive predictions,

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **F_1score** is the harmonic mean of recall and precision, given by,

$$F_1 = 2\,\frac{Recall \cdot Precision}{Recall + Precision}$$

  and is a special case of the F_β score where precision and recall are assumed to be of equal importance.

- **Balanced Accuracy** is obtained by calculating the recall for each class and taking the average e.g. in the case of binary classification, the average of the *TPR* and *TNR*, i.e.

$$\text{Balanced Accuracy} = \frac{TPR + TNR}{2}$$

  Balanced Accuracy is much more appropriate to use for imbalanced datasets than accuracy when the minority class is of interest, for instance, using the same numbers as in the example for accuracy above, $\text{BA} = 1/2 \cdot (0/10 + 990/990) = 0.5$ better reflecting the classifier's inability to correctly predict positive examples.

- **False Discovery Rate** (**FDR**) is the ratio of samples incorrectly identified as positive to all samples predicted positive, given by

$$\text{FDR} = \frac{FP}{TP + FP} = 1 - PPV$$

  and is of particular interest to the problem of intrusion detection, as a high FDR could mean disgruntled users start paying less attention to the IDS' alarms since a large proportion of them would be false positives.

- **False Positive Rate** (**FPR**) is the ratio of samples incorrectly identified as positive to those that are negative, given by

$$\text{FPR} = \frac{FP}{N} = \frac{FP}{TN + FP} = 1 - TNR$$

  It is also important for IDS for similar reasons to the FDR.

ROC curve & AUC

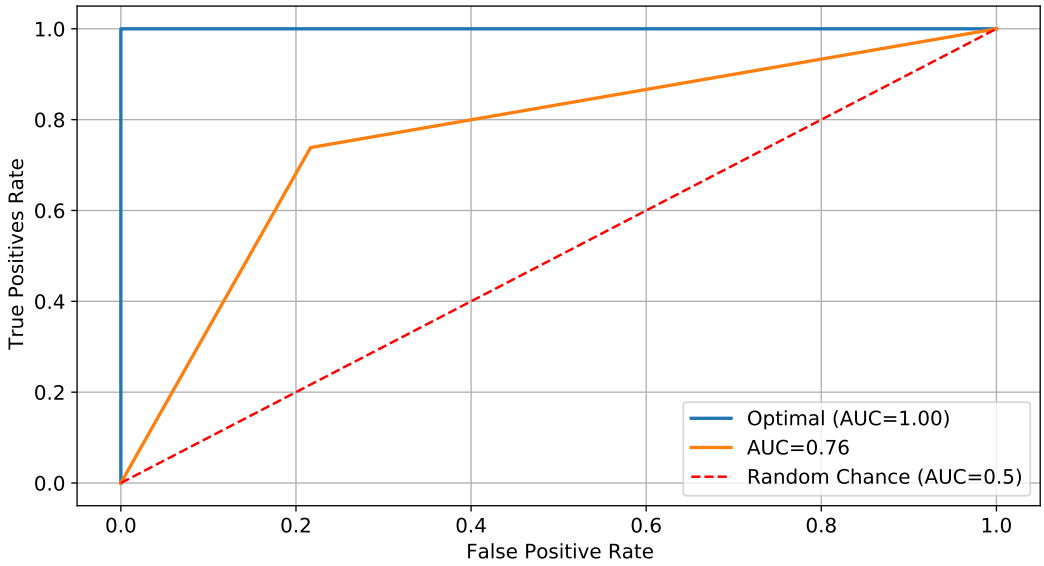

FIGURE 2.5: An example of a ROC curve, including the AUC

The ROC (Receiver Operating Characteristic) curve (depicted in Figure 2.5), which is a plot of the TPR versus the FPR, shows the performance of a binary classifier for all thresholds.

A threshold in this context is the value which separates which of the two classes the classifier predicts i.e. given a threshold τ, classifier output ρ and classes A, B, if $\rho \leq \tau$ the classifier is said to predict class A, otherwise it predicts class B, usually A = Negative, B = Positive.

The best thresholds are those which maximize the TPR and minimize the FPR i.e. those in the upper left corner of the plot.

The **AUC** (Area Under the ROC Curve) is a value which summarizes the ROC curve i.e. a classifiers performance for all thresholds, with higher value AUCs being generally better. The AUC is exactly what its name says it is, the area under the ROC curve.

2.2.10 Generalizing Metrics to Multi-Class Classifiers

		True Class		
		A	B	C
Predicted Class	A	True A	False A	False A
	B	False B	True B	False B
	C	False C	False C	True C

TABLE 2.2: A 3×3 Confusion Matrix

The metrics defined above can be generalized to the multi-class classification case, where for a problem with n classes the confusion matrix becomes $n \times n$, for example, for a problem with 3 classes, the confusion matrix becomes a 3×3 matrix shown in Table 2.2. They can further be summarized by taking the micro- or macro-average.

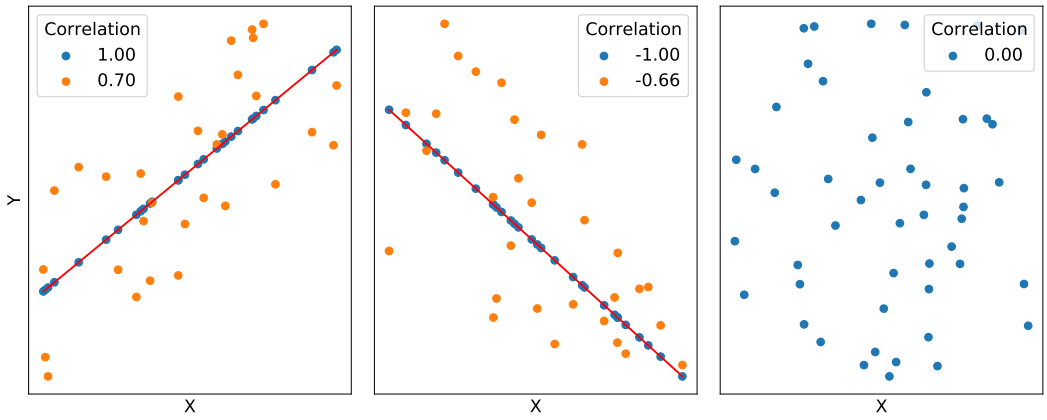

FIGURE 2.6: Scatterplots of variables with different PCC values

To calculate the micro-average means considering the *TP*s, *FP*s and *FN*s for all classes together, i.e. *TP* is the sum of the diagonal elements in the $n \times n$ confusion matrix, whereas the *FP* and *FN* are both equal to the sum of the off-diagonal elements and therefore to each other. As a result, in the micro-average case, the Precision, Recall, F_1 score and Accuracy are all equal.

Whereas, to calculate the macro-average means simply calculating the metric for each class and taking the arithmetic mean. For example, the Recall for class A in Table 2.2 above, is

$$Recall_A = \frac{TP}{TP + FN} = \frac{True\ A}{True\ A + (False\ B + False\ C)}$$

where *False B*, *False C* are those in the first column i.e. the ones where the true class is A but the classifier incorrectly predicts classes B and C respectively. $Recall_B$ and $Recall_C$ can be calculated in the same manner, and finally,

$$\text{Macro-Average Recall} = \frac{1}{3}\left(Recall_A + Recall_B + Recall_C\right)$$

2.2.11 Pearson product-moment correlation coefficient (PPMCC)

The PPMCC or PCC (Pearson correlation coefficient) for short, measures the strength of the linear relationship between two variables X and Y and is defined for a sample, as,

$$\mathbf{r}_{xy} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}$$

where n is the sample size, $x_i, y_i,\ i = 1, \ldots, n$ are sample points and $\overline{x}, \overline{y}$ are the sample means.

Its value is in the range $[-1, 1]$ where a value of 1 indicates the data are perfectly described by a line where as X increases so does Y and conversely, a value of -1 indicates the same, only as one variable increases the other decreases. A value of 0, on the other hand, implies the variables have no linear relationship, though this does not preclude a non-linear relationship. Finally, the PCC is symmetric i.e.

$corr(X, Y) = corr(Y, X)$, which in turn means the correlation matrix is also symmetric.

It should also be noted that two variables being correlated, even highly, does not necessarily mean a causal relationship exists.

Correlation Matrix

The correlation matrix of m variables is a symmetric $m \times m$ matrix whose element at position (i, j) is equal to the correlation between the i_{th} and j_{th} variables.

2.3 Deep Learning

Deep Learning (DL) is a subset of machine learning that encompasses a class of models inspired by how the brain works, called (artificial) neural networks (NNs). The "Deep" part of the name refers to the use multiple layers of neurons by those the networks.

2.3.1 Fully Connected Neural Networks

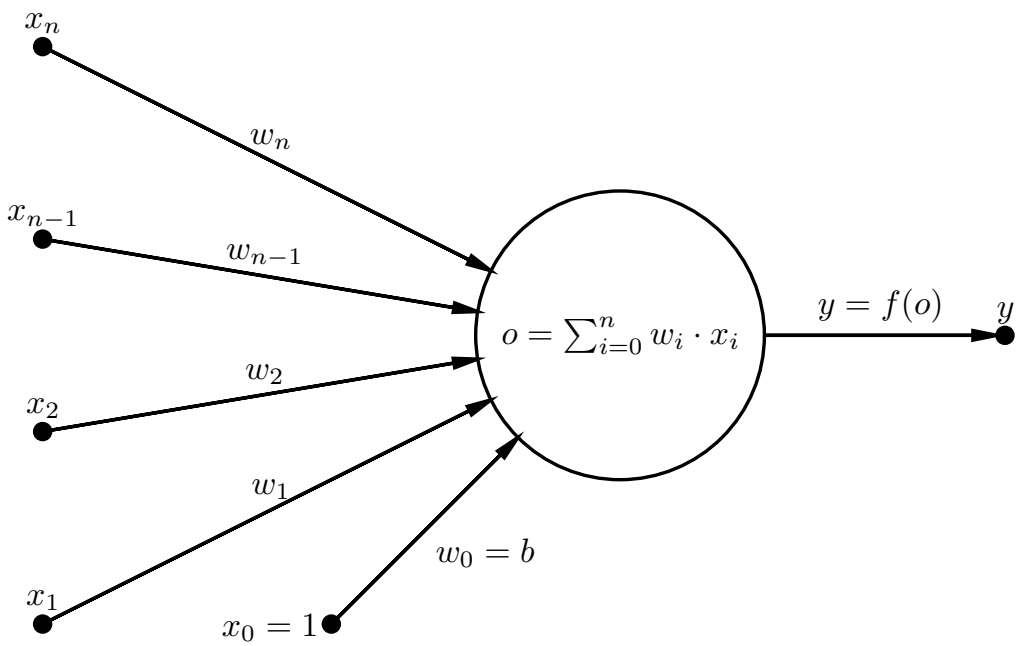

FIGURE 2.7: A neural network neuron

Much like the neuron is the basic unit of biological neural networks, the artificial neuron is the basic unit of artificial neural networks, although it is far simpler in comparison. An artificial neuron (Figure 2.7), or neuron, with input $\mathbf{x} = \{x_1, x_2, \ldots, x_{n-1}, x_n\}$, weight $\mathbf{w} = \{w_1, w_2, \ldots, w_{n-1}, w_n\}$, activation function, or non-linearity f, and output y, is defined as follows,

$$y = f\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right) = f\left(\mathbf{w}^\intercal \mathbf{x}\right)$$

where b is the bias vector, which can be folded into the inner product $\mathbf{w}^\intercal \mathbf{x}$ by adding an element $w_0 = b$ to the weight vector and $x_0 = 1$ to the input vector.

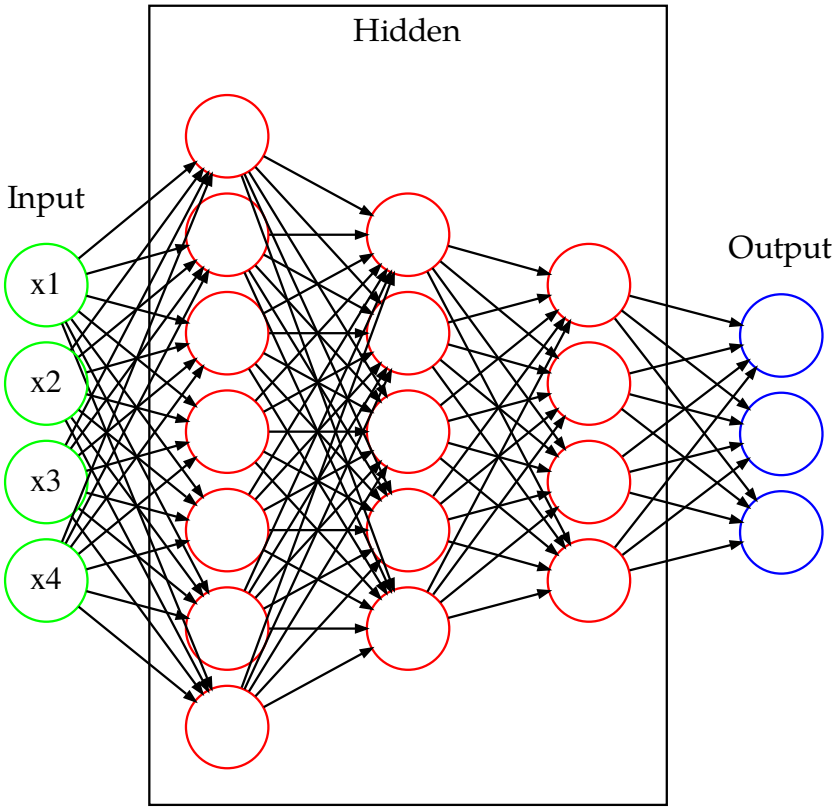

FIGURE 2.8: A Fully Connected Neural Network with 3 hidden layers

Neurons are further organized into layers where, in the fully connected case, the neurons of each layer i are connected to every neuron in the previous $\mathrm{i}-1$ and next $\mathrm{i}+1$ layers, hence the name.

Assuming all k_j neurons of a layer j share the same activation function f, which is not strictly necessary, and the input to the network has batch size *batch*, then the output of the j_{th} layer $a_j \in \Re^{k_j \times batch}$ is given by

$$a_j = f\left(\mathbf{W^j} a_{j-1}\right)$$

where $a_{j-1} \in \Re^{k_{j-1} \times batch}$ is the input to j_{th} layer/output of the $(j-1)_{\text{th}}$ layer and $\mathbf{W^j} \in \Re^{k_j \times k_{j-1}}$ is a matrix containing the j_{th}'s layers weights, i.e.

$$\mathbf{W^j} = \left[\mathbf{w_1^j}, \mathbf{w_2^j}, \ldots, \mathbf{w_{k_j-1}^j}, \mathbf{w_{k_j}^j}\right]^\intercal$$

where $\mathbf{w_n^j} \in \Re^{k_{j-1}}$ is the vector containing the weights for the j_{th} layer's n_{th} neuron.

The first layer, also called the input layer, takes the inputs, i.e. the feature values for an observation or for batches of observations, transforms them and then "sends"

them as input to the next layer, where once again, the inputs are transformed and forwarded to the next layer and so on and so forth until the last layer.

The last layer i.e. the one that produces the final output(s) of the model is called the output layer and the layers in-between the first and last layers are known as hidden layers.

2.3.2 Activation Functions

The activation function is a non-linear function that determines the output of a neuron and can be thought of as deciding whether a neuron should "fire". A multi-layer neural network without activation functions, or with linear activations would be as if the input $\mathbf{x}$ was simply multiplied by a single matrix $\mathbf{W} = \mathbf{W^n W^{n-1}} \dots \mathbf{W^2 W^1}$, where $\mathbf{W^i}$ is the weight matrix for the i_{th} layer, and the neural network would be much less powerful, essentially equivalent to a simple linear model.

Some of the most popular activation functions are presented bellow.

Sigmoid

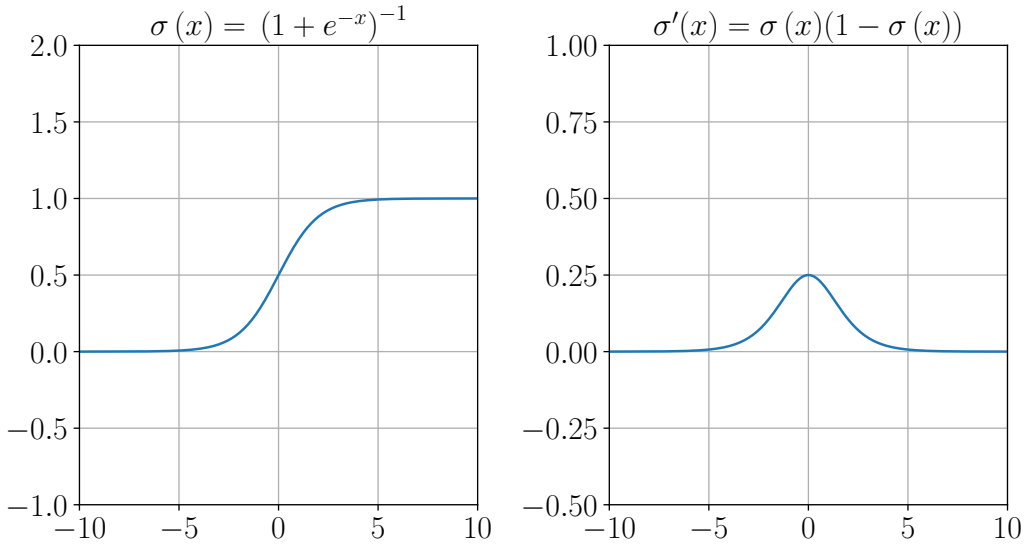

FIGURE 2.9: Sigmoid activation function

The sigmoid function takes a real valued input and maps it to the range $[0, 1]$. There has been a decline in its use as it suffers from a couple of problems, namely a) the vanishing gradient problem whereupon for more extreme inputs its gradient tends to become zero, thus making it difficult for backpropagation to update the network's weights, b) its output is not centered and c) it is expensive to calculate.

It is defined as

$$\sigma(x) = \frac{1}{e^{-x} + 1}$$

and its derivative is

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

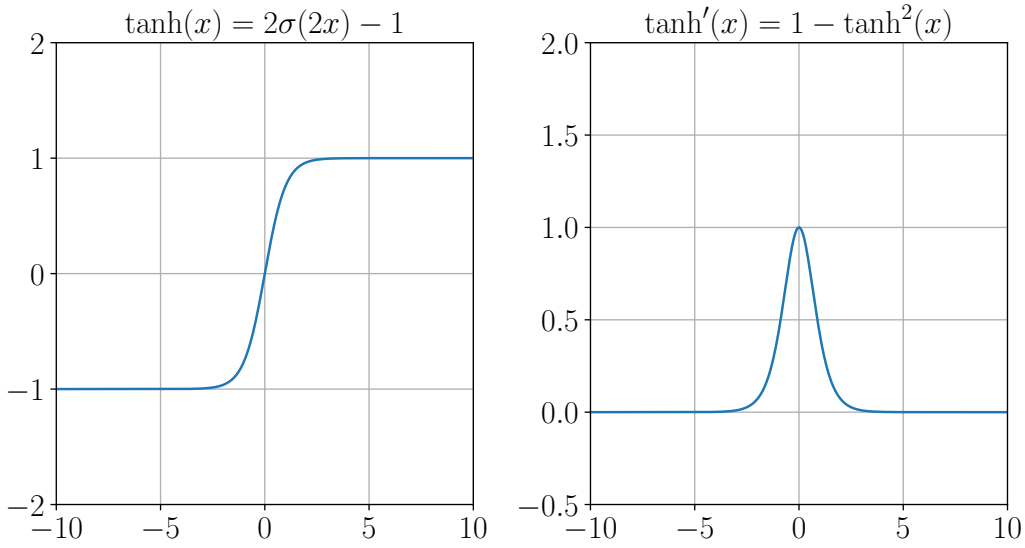

FIGURE 2.10: Tanh activation function

Tanh

The hyperbolic tangent function is essentially a scaled sigmoid with a range in $[-1, 1]$ and has the same issues it does, except it is 0 centered. It is defined as

$$\tanh(x) = 2\sigma(2x) - 1$$

and its derivative is

$$\tanh'(x) = 1 - \tanh^2(x)$$

ReLU

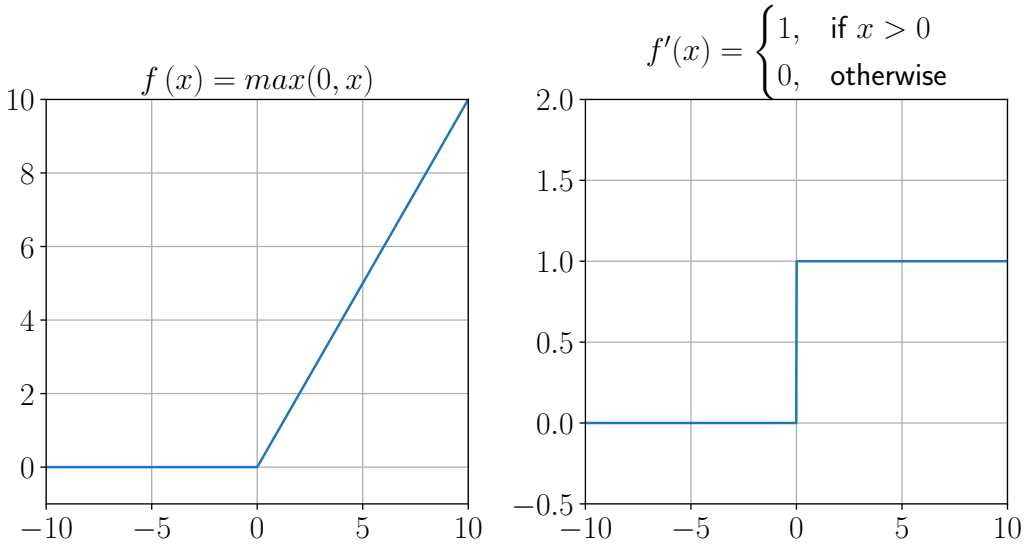

FIGURE 2.11: ReLU activation function

The rectified linear unit function is equal to the identity function for positive inputs and maps all other inputs to 0. Compared to the sigmoid and hyperbolic tangent it is less expensive to calculate and does not suffer from the vanishing gradient problem, however it can cause neurons to "die" meaning they stop activating.. It is given by,

$$f(x) = max(x, 0)$$

While its derivative is not actually defined at 0, by convention it is set to 0, and so it is

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

2.3.3 Backpropagation

Backpropagation is an algorithm for efficiently calculating the gradient of the loss function with respect to the weights (and biases) of a neural network. It is named so because the gradients "flow" backwards, starting from the output layer and finishing at the input layer as opposed to the normal flow of information from input layer to ouput layer. Its efficiency stems from the fact that it re-uses previously done computation, instead of naively calculating each layer's gradient individually.

It is used in conjunction with optimization algorithms (Section 2.2.7), such as mini-batch gradient descent or Adam which use the information it provides to update the network's weights.

2.3.4 Recurrent Neural Networks

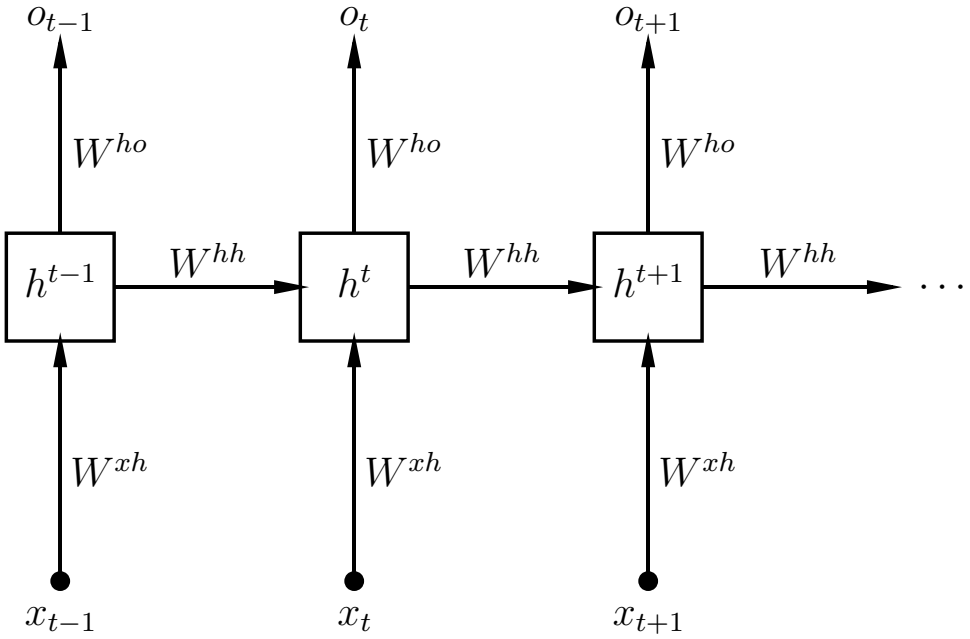

FIGURE 2.12: An unrolled RNN

A Recurrent Neural Network (RNN) is a neural network which tackles the problem of learning from sequences, as instead of treating each observation independently as is the case with feed-forward neural networks, it can be thought to contain a kind of "memory" (hidden state) which stores information about previous observations seen in a sequence. As a result, an RNN when making a prediction, considers not only the current input x_t, but important information about all previous inputs $x_1, x_2, \ldots, x_{t-2}, x_{t-1}$ so far.

Given an input $x_t \in \Re^d$ of sequence $x_1, x_2, \ldots, x_{n-1}, x_n$ and the previous hidden state $h_{t-1} \in \Re^h$, the next hidden state h_t is given by the following recurrence relation:

$$h_t = f(W^{hx}x_t + W^{hh}h_{t-1} + b)$$

where f is an activation function, $W^{hx} \in \Re^{h\times d}, W^{hh} \in \Re^{h\times h}$ are the weights and $b \in \Re^h$ is the bias. The initial hidden state h_0 is usually set to the zero vector. The output

$$o_t = f\left(W^{ho}h_t\right)$$

depends on the task e.g. f is the identity function for regression, or the softmax for multi-class classification.

Unfortunately, RNNs suffer from the exploding and vanishing gradient problems, and they struggle when it comes to learning long-term dependencies.

2.3.5 Long Short-Term Memory Networks

LSTMs, first introduced by Hochreiter and Schmidhuber [28] in 1997, seek to overcome the problem of RNNs with learning long-term dependencies. They introduce the concept of "gates" which control what information is stored in the memory, or cell state.

Each gate involves a point-wise multiplication between the output of a sigmoid, which is between 0 and 1 and the cell state, thus controlling how much of a component to let through [29].

An LSTM has 3 kinds of gates, the input gate i, the forget gate f and the output gate o which control what information is added to and removed from the memory, and the output from the LSTM cell respectively.

Given input $x_t \in \Re^d$, hidden state $h_t \in \Re^h$ and cell state $c_t \in \Re^h$, an LSTM's operation is described by the following equations:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{2.1}$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{2.2}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{2.3}$$

$$g_t = \tanh(W_g x_t + U_g h_{t-1} + b_g) \tag{2.4}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{2.5}$$

$$h_t = o_t \odot \tanh(c_t) \tag{2.6}$$

where $\odot$ is element-wise multiplication (Hadamard product), $f_t, i_t, o_t \in \Re^h$ are the outputs of the forget, input and output gates respectively, $g_t \in \Re^h$ is the cell input activation and $W_* \in \Re^{h\times d}, U_* \in \Re^{h\times h}$ are the weights and $b_* \in \Re^h$ are the biases.

Or, to put it into words, each of the 3 gates (eqs. 2.1 to 2.3) involve the input at timestep t, x_t, and the hidden state of the previous step h_{t-1} being multiplied by

weight matrices W_* and U_* and then summed with the bias b_*, and the result is then passed through a sigmoid and squashed to the range $[0, 1]$. The same happens with the input activation, only the non-linearity is a tanh which maps its input to $[-1, 1]$.

To update the cell state (eq. 2.5), the old cell state c_{t-1} is multiplied element-wise with the output of the forget gate, which controls how much of the old information should be forgotten, and the input activation g is multiplied element-wise with the input gate i which decides what new information should be added and the two results are summed to produce the new cell state c_t.

Finally, to get the new hidden vector (eq. 2.6) a tanh activation is applied to the new, updated cell state c_t and the result is multiplied elementwise with the output gate.

2.3.6 Autoencoders

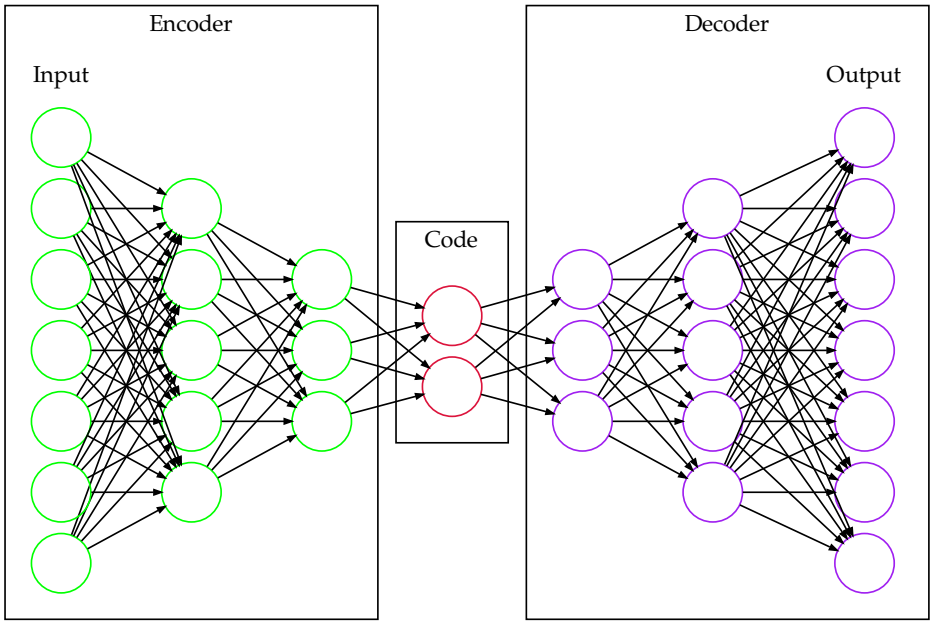

FIGURE 2.13: Example of an undercomplete AE

An autoencoder (AE) is a neural network that attempts to learn a "compressed" representation of its inputs, or "code" $\mathbf{c}$. Generally AEs can be thought of as having two parts, an encoder which maps the input $\mathbf{x}$ to a lower dimensional feature space

$$\mathbf{c} = \text{encoder}(\mathbf{x})$$

and a decoder which takes that compressed representation and tries to map it back to the original input

$$\hat{\mathbf{x}} = \text{decoder}(\mathbf{c})$$

The network is trained to minimize the reconstruction error between its inputs and outputs, in other words, it tries to learn weights such that $\hat{\mathbf{x}} \approx \mathbf{x}$.

To prevent the network from simply memorizing the data and instead force it to learn useful features several regularization methods may be applied, such as introducing a "bottleneck" by having lower dimensional hidden layers as shown in Figure 2.13 (undercomplete AEs), adding a sparsity constraint which penalizes neuron activations encouraging fewer neurons to fire at a time (Sparse AEs), or adding noise to the inputs while the AE tries to predict the unperturbed inputs (Denoising AEs).

2.4 Networking Concepts

In order to create a sequence dataset from raw packet captures, some fields of the headers of the Internet, Transmission Control and User Datagram protocols were selected as features. A brief description of the protocols and their headers follows.

2.4.1 Internet Protocol (IP)

The Internet Protocol exists at layer 3, also known as the network layer, of the OSI model and in effect underpins the Internet. It is a best-effort, connectionless protocol which governs how nodes in different networks communicate. It performs the routing of packets from a source to a destination address and also has the ability to split packets that are too large to be transmitted into fragments and reassemble them at another node.

IPv4 Header

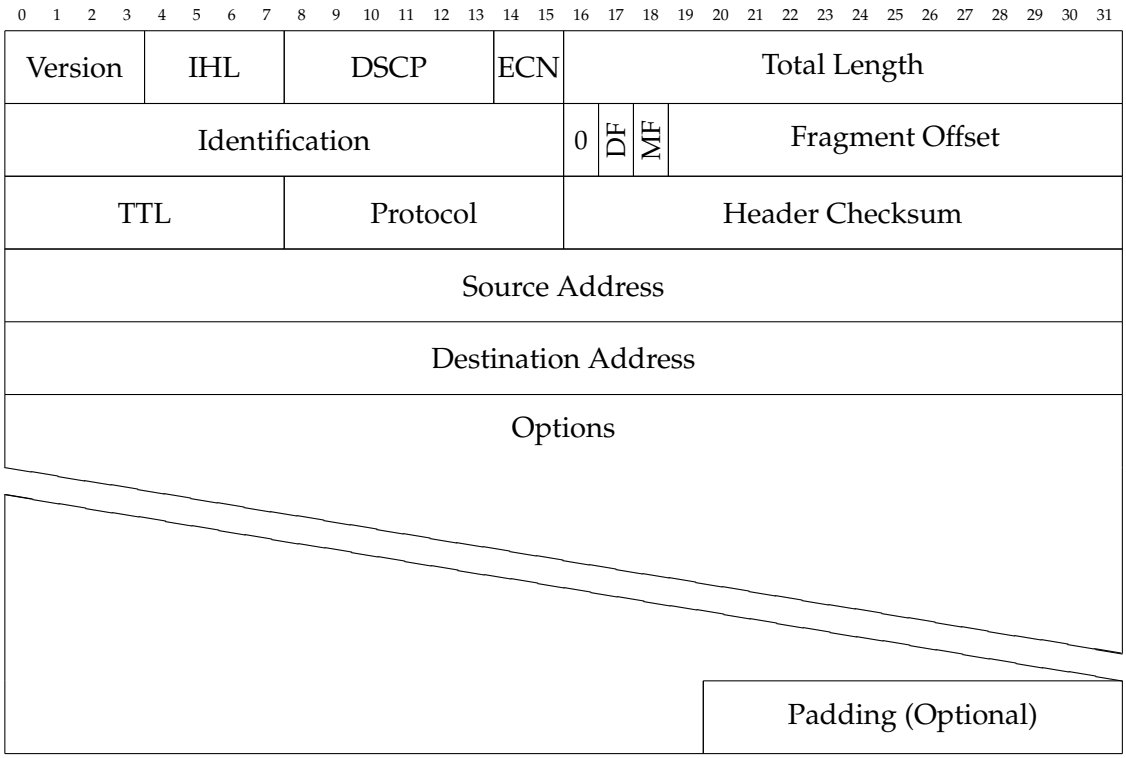

FIGURE 2.14: IPv4 Header [30]

The fields of the IPv4 header are as follows, as specified in RFC 791 [30] unless indicated otherwise.

- **Version** (4 bits): Contains the IP version which for IPv4, as the name suggests, is always 4.
- **IHL (Internet Header Length)** (4 bits): The length of the internet protocol header varies due to the Options field whose length also varies, so the IHL is the length of the header in 32 bit words i.e. it points to the start of the data.
- **DSCP (Differentiated Services Codepoint)** (6 bits) [31]: This field, along with the one bellow, supersede the old IPv4 ToS (type of service) field. It offers an improvement to the Internet Protocol by allowing for a scalable way to discriminate between types of network packets, for instance packets that require low latency may be marked as such.
- **ECN (Explicit Congestion Notification)** (2 bits) [32]: ECN provides another way for routers to indicate imminent network congestion, in addition to dropping packets. It is interesting to note that ECN cannot function without support from the transport layer protocol.
- **Total Length** (16 bits): Total packet length (header and data) in 8 bit bytes. It is 16 bits long meaning the maximum packet size is 65,535 bytes. It is also required that every host be capable of handling packets at least as large as 576 bytes.
- **Identification** (16 bits) [33]: It is used to uniquely identify the fragments of a non-atomic datagram i.e. one that still may be fragmented, or one that already has been.
- **Flags** (3 bits): Control flags, each one 1 bit long, are described bellow in the order they appear in the header,
  - **Reserved**: must be zero
  - **DF** (Don't Fragment): If it is set, the datagram must not be fragmented, even if that means it cannot reach its destination, in which case it must be discarded.
  - **MF** (More Fragments): If it is set, the datagram is not the final fragment. If it is the final one, or the packet is not fragmented, it is cleared i.e. zero.
- **Fragment Offset** (13 bits): It represents the position of the fragment in the original unfragmented datagram, in 64 bit chunks, with the first fragment having offset 0.
- **TTL (Time to Live)** (8 bits): It is there to ensure that packets cannot persist indefinitely in the internet by, for example, going in loops. It is supposed represent the number of seconds that a packet may persist, but it is decremented by at least one by every node that processes the packet. When it reaches zero the packet is discarded.
- **Protocol** (8 bits): It specifies the protocol used in the data portion of the datagram, such as TCP or UDP.
- **Header Checksum** (16 bits): A checksum of the header used to check for errors. It needs to be recomputed whenever fields in the header change, such as the TTL at every hop.
- **Source Address** (32 bits): The source IP address.

- **Destination Address** (32 bits): The destination IP address.
- **Options** (length varies): A field that is not necessarily present in datagrams and may include a variable number of options.
- **Padding** (length varies): It is added as needed to ensure the packet header is an integer number of 32 bit words in length. It is set to zero.

2.4.2 Transmission Control Protocol (TCP)

TCP resides at layer 4, also known as the transport layer, of the OSI model. It is a connection based protocol that provides reliable in-order communication between hosts. A connection is established through a 3-way handshake which is involves a 3 packet exchange hence the name. TCP achieves reliability by having error-checking to identify errors, acknowledgment numbers to indicate which segments have been received, and being capable of retransmiting segments that were damaged or lost. To ensure segments can be ordered at the destination, TCP associates each one with a sequence number.

TCP Header

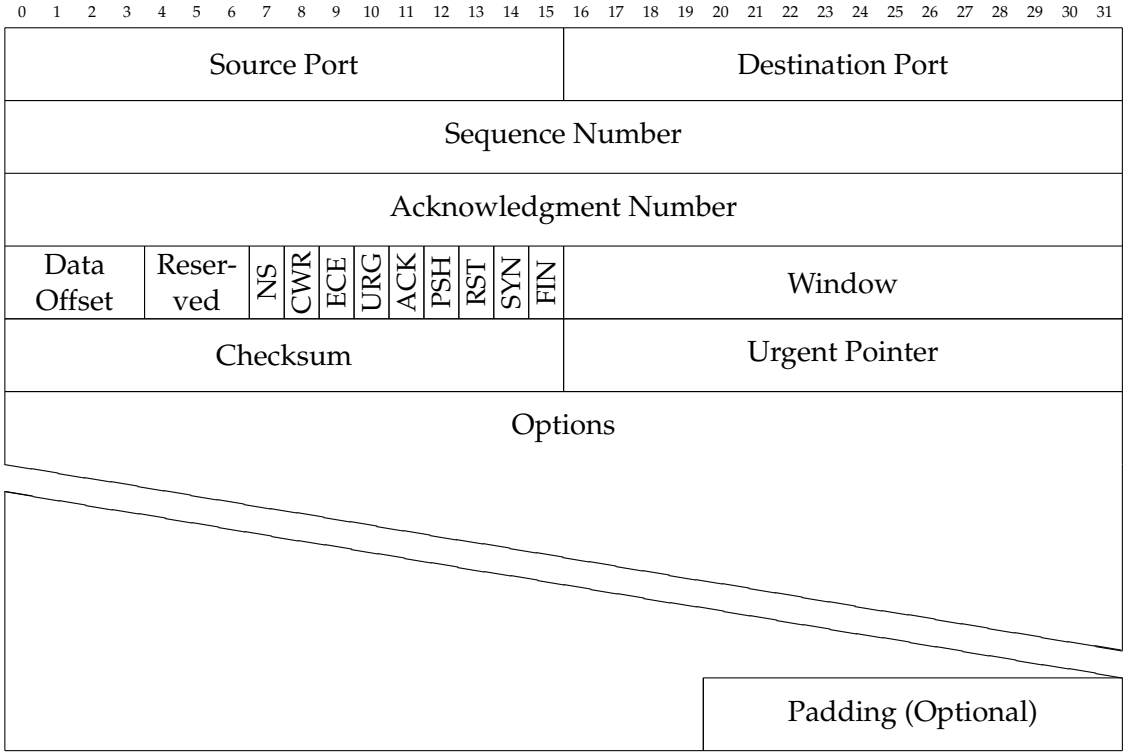

FIGURE 2.15: TCP header [34]

The fields are as follows, as specified in RFC 793 [34] unless indicated otherwise.

- **Source Port** (16 bits): The source port number.
- **Destination Port** (16 bits): The destination port number.
- **Sequence Number** (32 bits): If SYN is set, this is the initial sequence number, otherwise it is that of the first data byte in this segment.

- **Acknowledgment Number** (32 bits): If ACK is set, this is the value of the next sequence number expected by the sender of this segment.
- **Data Offset** (4 bits): Similar to the IHL for IP, this is the length of the TCP header measured in 32 bit words, which varies due to the Options field of the TCP header, which may or may not be present.
- **Reserved** (3 bits): A field reserved for future use. It must be set to zero.
- **Control Bits** (9 bits): Each flag is 1 bit long, and they are described bellow in the order they appear in the header,
  - **NS** [35]: An experimental field that prevents the concealment of "marked" i.e. with ECN set to 11, Congestion Experienced (CE), packets.
  - **CWR** (Congestion Window Reduced) [32]: Set to signify to the receiver that a segment with the ECE flag set was received by the sender of this one and handled appropriately, so the receiver can stop setting it.
  - **ECE** (ECN echo) [32]: ECE being set indicates that the sender of this segment received a packet with the IP ECN flag set to 11 (CE), meaning congestion experienced. This is done in order for the receiver of this segment to be notified of said congestion and act accordingly. On the other hand, if SYN is also set, this only indicates that the sender is ECN capable.
  - **URG**: If set, the Urgent Pointer field is significant.
  - **ACK**: If set, the Acknowledgment Number field is significant.
  - **PSH**: Push function. Since received data may be buffered, this flag exists to indicate that all data received so far should be immediately pushed to the receiver, if it is set.
  - **RST**: Reset the connection.
  - **SYN**: Synchronize sequence numbers. Used during the 3-way handshake.
  - **FIN**: Sender has no more data to send.
- **Window Size** (16 bits): This field specifies the number of, by default bytes, though that may be changed with an option, the sender is willing to receive.
- **Checksum** (16 bits): A checksum of the TCP header, data and a pseudo header comprised of the Source Address, Destination Address and Protocol fields of the IP header, as well as the TCP length, which is the header plus the data length. It is used for error checking.
- **Urgent Pointer** (16 bits): If the URG flag is set, it points to the byte after the urgent data as a positive offset from this segment's sequence number.
- **Options** (length varies): A field that is not necessarily present in segments and may include one or more options.
- **Padding** (length varies): It is added as needed to ensure that the TCP header is a multiple of 32 bit words in length. It is set to zero.

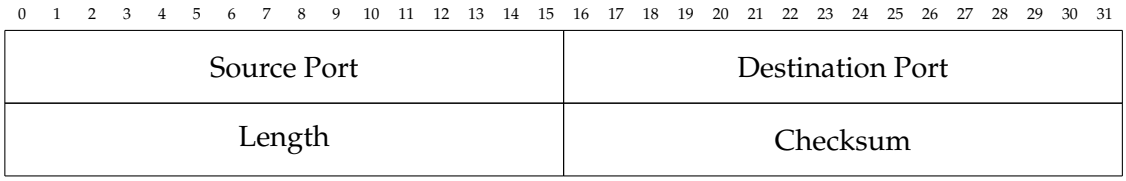

FIGURE 2.16: UDP header [36]

2.4.3 User Datagram Protocol (UDP)

UDP is also a layer 4, i.e. transport layer, protocol. It is a simple, connectionless protocol, that unlike TCP makes no attempt to ensure reliability, meaning any such logic is pushed to the user application. This also makes it fast, since it requires less processing and there are no retransmissions.

UDP Header

The fields are as follows, as specified in RFC 768 [36].

- **Source Port** (16 bits): The source port number if it is meaningful, otherwise it is set to zero.
- **Destination Port** (16 bits): The destination port number.
- **Length** (16 bits): Length in 8 bit bytes of the UDP header and data.
- **Checksum** (16 bits): A checksum of the UDP header, data and a pseudo header comprised of the Source Address, Destination Address and Protocol fields of the IP header, as well as the UDP length field. It is used for error checking and if it is set to zero it means that no checksum was generated.

2.4.4 Port Number Ranges

Ports can be split into three categories based on the port number:

- "System/Well-Known Ports" whose port numbers are from 0 to 1024 and are controlled and assigned by IANA (Internet Assigned Numbers Authority). Generally, they require extra privileges to use i.e. bind a socket to, e.g., for Linux, a user needs to be root or have the "CAP_NET_BIND_SERVICE" capability.
- "User/Registered Ports" are those with port numbers from 1024 to 49151, which can generally be used by all users without a need for extra privileges, and for which a list of the services that use them is maintained by IANA.
- "Dynamic/Private/Ephemeral Ports" which are those that are left i.e. port numbers from 49152 to 65535 and which cannot be registered by IANA.

It is worth noting that some systems may choose alternative ranges for the ephemeral ports.

2.5 Literature Review

This section aims to present relevant literature in the field of network intrusion detection, that is, research about the application of machine learning and neural networks in this domain.

In 2013, Revathi and Malathi [37] trained a variety of algorithms for multi-class classification on the NSL-KDD dataset [38]. The algorithms were trained both on the full number of features and later an optimally selected subset, using the "Correlation-based Feature Selection" technique. A random forest classifier trained on the reduced dataset performed the best, with an average accuracy score of 98.88%, outperforming J48, SVMs, CART and Naive Bayes.

In 2015, Ingre and Yadav [39] investigated the performance of multi-layer neural networks on the NSL-KDD dataset, testing various architectures with different numbers of hidden layers and training algorithms, reaching an accuracy of 81.2% at binary classification with an artificial neural network with 21 layers, trained on 29 of 41 features with the Levenberg-Marquardt algorithm, and a best accuracy of 79.9% at multi-class classification, with a 23 layer network trained on all the features with BFGS Quasi-Newton backpropagation.

In 2017, Dias et al. [40] proposed an artificial neural network architecture with one hidden layer trained on the KDDCUP'99 dataset [41] with the goal of multi-class classification. They varied the number of neurons in the hidden layer, attaining an average detection rate of 99.9%. However, the network failed at detecting one type of attack consistently, with a detection rate of 51.9% and the authors citing the low number of samples for that class as the culprit.

In 2017 Bontemps et al. [42] proposed the use of LSTM RNNs on a time series version of the KDDCUP'99 dataset, specifically focusing on detecting DoS Neptune attacks. Each prediction of the LSTM is compared to the future real value and the relative error is recorded; if it is over a threshold, a point is considered anomalous.

At each timestep, if the density of anomalous points exceeds a threshold α and the average relative error a threshold β a sequence is flagged as a collective anomaly.

In 2016, Javaid et al. [43] used self-taught learning (STL), a two-step approach wherein unlabeled data -not necessarily from the same distribution as the labeled data, but at least a relevant one- are used to train an algorithm that attempts to learn a good feature representation of the data, followed by expressing the labeled data in that representation and using the result for, in this case, classification.

They applied this technique on the NSL-KDD dataset, using a sparse autoencoder for the feature learning and soft-max regression for the classification and compared its performance to using soft-max regression directly on the dataset instead of the learned features.

STL generally performed better, especially in the binary classification case, where it achieved an accuracy of 88.39% and an f-measure of 90.4% whereas for the 5-class classification it achieved an accuracy of 79.1% and a weighted f-measure of 75.76%.

In 2018, Shone et al. [44] proposed a feature representation learning neural network architecture, named stacked non-symmetric deep auto-encoder (NDAE), which functions similarly to deep autoencoders (DAE) but without the decoder stage.

The encoded representations learned by the model were used to train a random forest (RF) classifier which reached an average accuracy of 97.85% for the 5 class KDD99 dataset, and 85.42% and 89.22% for the 5 and 13 class NSL-KDD respectively, although it consistently struggled with classes that had few examples. When

compared to Deep Belief Networks (DBNs), stacked NDAEs performed better except for the classes with few examples, and also proved faster to train compared against DBNs with the same number of layers.

In 2017, Agarap [45] combined a gated recurrent unit network (GRU), which is a variant of an LSTM, with a support vector machine (SVM) which was used as the final layer of the network, and compared it against the conventional GRU-Softmax model. The 2013 Kyoto University honeypot dataset [46] was used and the GRU-SVM model outperformed the GRU-Softmax model, achieving an accuracy of 84.15% versus 70.75% in binary classification. It also proved faster to train and do inference with.

In 2017, Yin et al. [47] chose to try an RNN architecture on the NSL-KDD dataset and explored how the learning rate and number of nodes affected the classifier's performance, ultimately getting an accuracy score of 83.28% on the binary classification problem and 81.29% on the 5 class classification problem.

Radford et al. [48] proposed an unsupervised method inspired by natural language processing (NLP) using LSTMs. They used the ISCX IDS dataset [49] which is a dataset of network flows and produced ordered sequences by grouping flows that occur between IP pairs within the same hour, called dyad-hours, reaching an AUC score of 0.84 with their best performing model, based on protocol byte sequences e.g., from their paper, $\text{IP}_a\text{IP}_b : \text{TCP} : 10|\text{TCP} : 12|\text{UDP} : 04$ where each token is in the form $\textit{Protocol} : \textit{floor}(\textit{log}_2(\textit{bytes}))$.

Ahmim et al. [50] proposed a hierarchical model for intrusion detection and evaluated it on the CICIDS2017 dataset [51], where two classifiers were trained on the dataset, with the first making a binary "Attack"/"Benign" prediction and the second a multi-class classification between "Benign" and the types of attacks present in the dataset. Finally, these outputs in addition to all the dataset features were used to train a third classifier which also made the final decision between "Benign" or an attack type.

The authors tried various combinations of classifiers and accomplished their best result with REP Tree, Jrip and Forest PA as the first, second and third classifiers respectively, which was an overall detection rate of 94.475%, an accuracy of 96.665% and a false alarm rate of 1.145%.

Zhang et al. [52] chose a hierarchical deep learning model, using a convolutional neural network (CNN) to extract features from the first 160 bytes of packets from packet captures which were then fed into an LSTM. The hybrid model's performance was compared against only using a CNN or an LSTM.

The hierarchical model had a better F_1 score of 99.88% compared to the others on the binary classification task on the CICIDS2017 dataset, but, for multi-class classification, the plain CNN achieved a better F_1 score of 99.94% versus 99.91% for the hybrid model.

The models were also trained on the CTU dataset, where the hierarchical model had the best scores for the F_1 score metric, 99.87% and 99.82% for binary and multi-class classification respectively.

In 2019, Ferrag et al. [53] applied seven deep learning models on the CSE-CIC-IDS2018 dataset, such as deep, recurrent and convolutional neural networks, deep auto encoders and others, comparing their performances over different hyperparameter values with CNNs having the best accuracy score of 97.376%.

Abdulhammed et al. [54] applied two dimensionality reduction techniques, SAEs and Principal Component Analysis (PCA) on the CICIDS2017 flow dataset, reducing

the original 81 features to as few as 59 and 10 respectively and training various machine learning algorithms on these learned representations, achieving satisfactory results.

The best results for binary classification were achieved by Random Forests (RFs) with an f-measure of 0.997 when applied to either the results of PCA or SAE, although PCA produced far fewer features for those results (10 vs. 70).

For multi-class classification the best f-measure, 99.7% was also achieved by RFs applied to the result of PCA though this time using 30 PCA features. The authors conclude that, based on their experiments, PCA is superior to SAE.

Chapter 3

Dataset Selection & Analysis

3.1 Selecting the right Dataset

Selecting a good dataset for one's task is of vital importance, as a model is only as good as the data it is trained on. This section deals with exactly that issue, i.e. the process of selecting the right dataset for this Thesis.

3.1.1 Network datasets

Network traffic datasets typically consist of either packets, or flows. Packet-based datasets are usually in the libpcap [55] format (or the "next generation" pcap format, pcapng [56]), which contains a record for each captured packet. Flow-based datasets contain aggregated information about packet flows e.g. the number of transmitted packets or the average bytes per packet for the flow.

When deciding which dataset to select, the following properties, which closely match those outlined by Ring et al. [57], were considered,

- **Dataset age**: As time passes the character of network traffic (i.e. traffic patterns and contents) changes, and so do attack types as new attacks are developed, therefore more recently created datasets are more likely to be representative of current real world network traffic and attacks.
- **Attack Traffic Variety**: It is preferred various attacks be present in the dataset, in the hopes that, by being exposed to a variety of attacks, the models better learn what constitutes one versus what constitutes normal behaviour, and are therefore more likely to detect new and unknown attacks.
- **Normal User Behaviour**: The dataset should also include traffic of normal behaviour, preferably not simulated, in order for the model to be competent at recognizing it, thereby keeping false alarms as low as possible to avoid disturbing normal operation.
- **Data Volume**: Deep learning is "data-hungry" and generally requires large numbers of examples to train on, so the dataset has to be reasonably large.
- **Labeled**: Most the models trained for this Thesis fall under the aegis of supervised learning, which requires a labeled dataset.

Finally, the dataset should also be **packet-based** and include the network traffic in the pcap(ng) format since this Thesis concerns itself with how models trained on sequence data perform against models trained on flow data.

In the following sections, one of, if not the most popular network intrusion detection dataset and why it was rejected is described, followed by a description of that which was finally selected.

3.1.2 KDD'99 Dataset

Likely the most well-known dataset for the task of intrusion detection is the KDD Cup 1999 dataset [41], which was used for the International Knowledge Discovery and Data Mining Tools Competition in 1999 and was generated by the extracting appropriate features from the raw tcpdump data of the DARPA 98 [58] dataset.

However, despite still being used for research today, the dataset it was derived from came into criticism, concerning how well the generated data approximated real newtwork traffic [59], as did itself [60], though efforts have been made to rectify some of its deficiencies [38].

Even so, the dataset is currently more than 20 years old, so it seems unlikely to closely match modern network traffic.

3.1.3 CICIDS2017 Dataset

The CICIDS2017 dataset [61] is a publicly available dataset [51] which includes more than 50 gigabytes of raw network traffic data in the pcap format, as well as a flow-based dataset comprised of 79 features generated from said pcap data by CICFlowMeter [62], a network traffic flow generator. The dataset aims to fulfill eleven criteria suggested by Gharib et al. [63] as vital for a comprehensive benchmark dataset, namely 1) Complete Network, 2) Complete Traffic, 3) Labelled Dataset, 4) Complete Interaction, 5) Complete Capture, 6) Available Protocols, 7) Attack Diversity, 8) Heterogeneity, 9) Feature Set and 10) MetaData

This dataset possesses all the properties described above in Section 3.1.1 except for the normal user traffic not being synthetic: it is recent, big, labeled and contains data about a variety or attacks. It was therefore selected for use.

The following sections describe how the normal traffic was generated as well as the attacks that were used.

Benign Traffic

The benign traffic was artificially generated through the use of B-profiles [64], a technique which seeks to extract the abstract behaviour of a group of human users.

The process works by building individual user profiles and clustering them. The cluster centroids are then regarded as the abstract behaviour of each cluster's users, "B-profiles", and can then be used to generate realistic benign traffic.

Attack Traffic

Traffic from various different attacks is present in the dataset, more specifically,

- **Brute-Force**: A brute-force attack consists of an attacker making multiple attempts trying to guess hidden and/or sensitive information, for example, passwords, DNS subdomains or website directories and files.

  While simple brute force attacks involve trying all possible combinations (e.g. of valid password characters) that is often practically impossible, so more sophisticated methods are used, such as dictionary attacks.

  In dictionary attacks all words in a "dictionary", possibly an actual dictionary, or lists containing relevant words (e.g. passwords that have been previously leaked, collections of common subdomain names or website directory and file names etc.), are tried.

Additionally, some tools offer the capability of enhancing those dictionaries by using rules to generate more guesses, for instance, replacing letters with common substitutes e.g. “i” with “1”, or appending file extensions to file names.

This dataset includes SSH and FTP password brute-forcing.

- **Heartbleed** [65, 66]: Heartbleed was a security vulnerability in the popular OpenSSL software library disclosed in 2014. It could be exploited by sending a malformed SSL heartbeat packet to a vulnerable client or server and allowed attackers to read up to 64KB of memory they should not have access to at a time. It was found by researchers that it was possible to use the vulnerability to steal secrets, such as private keys, usernames and passwords [66].

- **DoS (Denial of Service)**: In a DoS attack an attacker attempts to render the victim machine (e.g. a web server) inaccessible to its intended users, usually by either exhausting its available resources (e.g. network bandwidth, memory, threads etc.), or sending requests crafted to trigger crashes.

- **DDoS (Distributed Denial of Service)**: A DDoS attack is a type of DoS attack in which multiple (usually) compromised systems attack the same target most often by overwhelming it with great volumes of traffic. DDoS attacks can be difficult to block since the traffic originates from multiple sources.

- **Port Scan**: Port scanning is “a technique that sends client requests to a range of service port addresses on a host” [67]. It is performed to discover ports which have services listening, so they may be further explored, or targeted with exploits.

- **Botnet**: A botnet refers to a number of, usually infected, machines, which can be sent instructions from the attacker to perform various tasks, such as initiate attacks e.g. DDoS attacks, steal data e.g. keylogging, send spam etc.

- **Web**: Web attack is a moniker given to a variety of attacks targeting vulnerable web applications, such as SQL injections, XSS (Cross-site-scripting) and others.

  The dataset includes SQLIs, XSS and brute-forcing a web application password.

- **Infiltration**: According to the paper accompanying the dataset, an “infiltration attack” refers to attacking the network from the inside, from a machine that has already been compromised and can continue to perform various other attacks against other hosts in the network.

3.2 CICIDS2017 Flow Dataset

As already mentioned, the pcap data is accompanied a flow dataset. It is spread over 8 files as seen in Table 3.1, where in each file each row contains information about a “packet flow", or “flow" and a class label.

Looking at the CICFlowMeter source code [62], a flow is identified by the five-tuple (IP source address, source port, IP destination address, destination port, protocol), where the duration of the flow i.e. the difference in timestamps of the last and first packets in the flow must be less than a variable named “flowTimeOut”, whose default value is 120 seconds. If the duration of a flow exceeds “flowTimeOut”, it is split into multiple.

File Name	Traffic
Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv	Benign, DDoS
Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv	Benign, PortScan
Friday-WorkingHours-Morning.pcap_ISCX.csv	Benign, Bot
Monday-WorkingHours.pcap_ISCX.csv	Benign
Thursday-WorkingHours-Afternoon-Infilteration.pcap_ISCX.csv	Benign, Infiltration
Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv	Benign, Brute Force, Sql Injection, XSS
Tuesday-WorkingHours.pcap_ISCX.csv	Benign, FTP-Patator, SSH-Patator
Wednesday-workingHours.pcap_ISCX.csv	Benign, DoS GoldenEye, DoS Hulk, DoS Slowhttptest, DoS slowloris, Heartbleed

TABLE 3.1: CICIDS2017 Flow Dataset Files

After merging the files, there is a total of 2,830,743 observations, with 78 features plus the class label. The dataset includes 15 different classes, and is heavily imbalanced (Table 3.2a), with over 80% of all observations belonging to the "BENIGN", i.e. "non-malicious", class and the rest being dominated, unsurprisingly, by the types of attacks that involve making large numbers of connections and therefore produce many flows, such as DoS attacks (Table 3.2b).

After inspecting the CICFlowmeter source code in order to better understand what each feature represents, there was reason to suspect some were duplicates, so before any extensive analysis the data requires "cleaning".

Class	Total	Percentage
BENIGN	2273097	80.300366
DoS Hulk	231073	8.162981
PortScan	158930	5.614427
DDoS	128027	4.522735
DoS GoldenEye	10293	0.363615
FTP-Patator	7938	0.280421
SSH-Patator	5897	0.208320
DoS slowloris	5796	0.204752
DoS Slowhttptest	5499	0.194260
Bot	1966	0.069452
Brute Force (Web)	1507	0.053237
XSS	652	0.023033
Infiltration	36	0.001272
Sql Injection	21	0.000742
Heartbleed	11	0.000389

(A) Class distribution for all Classes

Malicious Class	Percentage
—	—
DoS Hulk	41.437220
PortScan	28.500160
DDoS	22.958472
DoS GoldenEye	1.845795
FTP-Patator	1.423484
SSH-Patator	1.057481
DoS slowloris	1.039369
DoS Slowhttptest	0.986109
Bot	0.352553
Brute Force (Web)	0.270243
XSS	0.116920
Infiltration	0.006456
Sql Injection	0.003766
Heartbleed	0.001973

(B) Class distribution for Malicious Classes

TABLE 3.2: Breakdown of Class distribution for Flow Dataset

3.2.1 Data Cleaning

Firstly, a minor issue, was that the class labels in "Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv" contain a strange, non-printable byte sequence, likely in place of a hyphen, which was replaced with such.

Out of the 78 features, some do not change value at all in any of the files i.e. they are always have the same value, in this case zero, so they are dropped as they contain no information. Specifically they are *"Bwd PSH Flags"*, *"Bwd URG Flags"*, *"Fwd Avg Bytes/Bulk"*, *"Fwd Avg Packets/Bulk"*, *"Fwd Avg Bulk Rate"*, *"Bwd Avg Bytes/Bulk"*, *"Bwd Avg Packets/Bulk"* and *"Bwd Avg Bulk Rate"*.

Furthermore, feature *"Fwd Header Length"* is duplicated, so one instance was dropped; inspecting the code it appears there was an error and it was calculated twice, instead of *"Bwd Header Length"*.

Other features that are duplicates are *"Total Fwd Packets"* and *"Subflow Fwd Packets"* as well as *"Total Backward Packets"* and *"Subflow Bwd Packets"*, since their values are equal for all rows, so the latter of each pair is dropped.

Additionally, the pairs of features *"Avg Fwd Segment Size"*, *"Fwd Packet Length Mean"* and *"Avg Bwd Segment Size"*, *"Bwd Packet Length Mean"* also appear to be duplicates. Specifically, only 1 out of 2,830,743 rows for the first pair and 62 out of 2,830,743 for the second are not equal, while the maximum relative difference, taking the latter of each pair as the reference, is approximately 1.39e−13 and 9.94e−10 respectively, which can almost certainly be attributed to rounding errors, so the former of each pair was dropped.

Another peculiarity of the flow dataset is that, despite being named *X Flag Count*, where *X* is a TCP flag, the values of those features is either zero or one, making it unlikely that they are actually counts, considering, for example, that a TCP handshake includes two packets with the SYN flag set, so the count would be expected to be two for at least some flows.

The final list of features after removing all duplicates and constant features follows,

- For the **Flow**:
  - Duration in microseconds
  - Min, Max, Mean, Std & Var Packet Length
  - Packets/s
  - Bytes/s
  - Down/Up Packet Ratio
  - Min, Max, Mean & Std IAT
  - Min, Max, Mean & Std Idle Time
  - Min, Max, Mean & Std Active Time
  - TCP flag set for: FIN, SYN, RST, PSH, ACK, URG, CWR & ECE
  - Average Packet size
  - Destination Port
- For both **Forward and Backward directions**:
  - Total Number of Packets
  - Total, Min, Max, Mean & Std Packet Length

Total, Min, Max, Mean & Std IAT

Initial TCP Window Size

Average Bytes per Subflow

Packets/s

Total TCP Header Length (i.e. sum of length of all TCP headers)

- In the **Forward direction only**:

  Total number of PSH, URG Flags

  Min TCP Header Size

  Total number of Active data packets (i.e. with non-zero packet length)

- Label

LIST 3.1: Original Flow Dataset Final Features

Where, *packet length* refers to the number of data bytes in a TCP segment, *idle time* refers to the time before a flow became *active* and vice versa.

Finally, the dataset contains (relatively few) missing values (NaNs) and infinite values, as well as numerous negative values where they are nonsensical, except as indicators of missing values, so they and the infinite values are treated as such.

For feature *"Init_Win_bytes_backward"* a number of missing values occur when *"Total Backward Packets"* is equal to zero and are therefore easy to fill in with zeroes as, with no packets sent in the backward direction, the initial TCP window would never be set, hence zero.

3.2.2 Training/Test Split

As explained in Section 2.2.4 the dataset should be split into training, validation and test sets. Usually this is done by shuffling the dataset and splitting it in the desired ratio e.g. 80% of the observations assigned to the training set and the remaining 20% to the test set.

In this case however, as the dataset is heavily imbalanced, special care must be taken to ensure all classes are present in both splits. Consideration must also be taken so that no duplicate observations exist between the sets to avoid "training on the test set" and potentially overestimating the classifier's performance.

To achieve that, first a training/test set ratio is chosen, in this case 85/15, and the split is done in a stratified fashion with respect to the class label, which ensures the distribution of classes in both sets are similar to that of the original dataset, and finally any duplicate observations between the training set and test set are removed from the former.

The final numbers are 2,222,522 datapoints for the training set and 424,612 for the test set.

3.2.3 Correlation Matrix

The heatmap in Figure 3.1 shows there is a considerable number of pairs of features with high degrees of correlation,

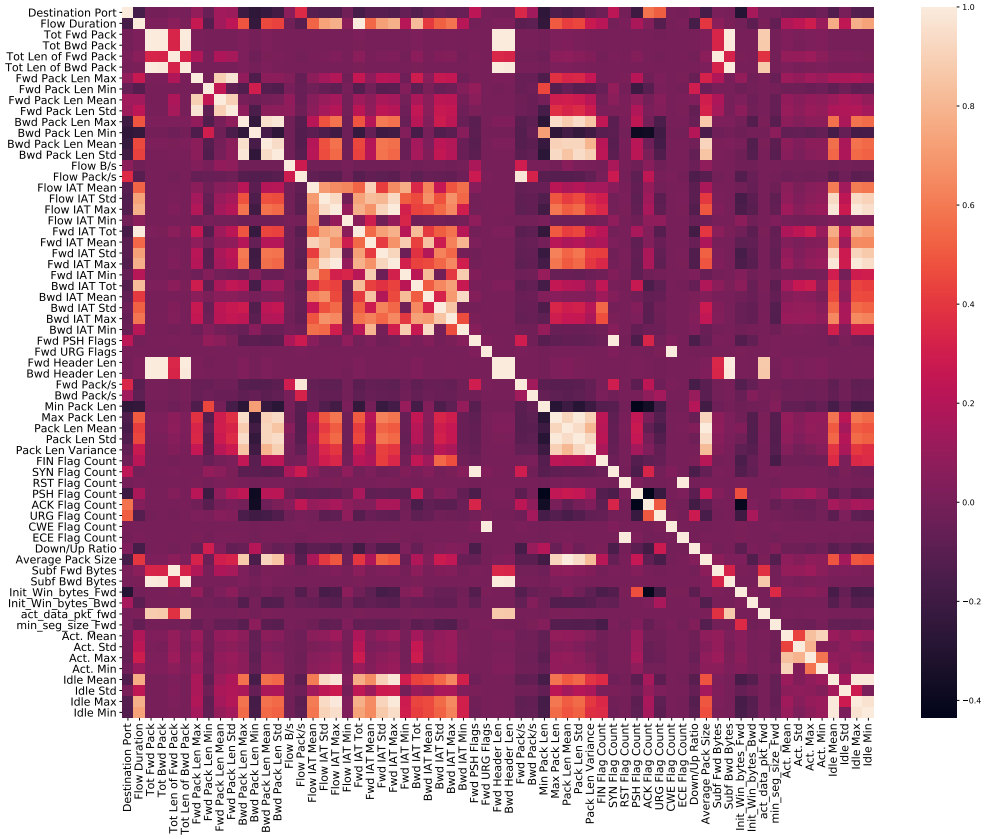

FIGURE 3.1: Heatmap of the correlation matrix for the Orginal Flow Dataset

$$0.50 < abs(corr) \leq 0.80 : 104 \text{ pairs}$$
$$0.80 < abs(corr) \leq 0.90 : 28 \text{ pairs}$$
$$0.90 < abs(corr) \leq 1.00 : 62 \text{ pairs}$$

with the highest being (absolute correlation over 0.999),

1. “Fwd URG Flags”, “CWE Flag Count”: 1.000
2. “Fwd PSH Flags”, “SYN Flag Count”: 1.000
3. “Total Length of Bwd Packets”, “Subflow Bwd Bytes”: 1.000
4. “Total Length of Fwd Packets”, “Subflow Fwd Bytes”: 0.9999
5. “Total Backward Packets”, “Bwd Header Length”: 0.9994
6. “Total Fwd Packets”, “Fwd Header Length”: 0.9994

3.3 Creating a Sequence & Flow dataset

Since a goal of this Thesis is comparing packet and flow classifiers, the pcap data needs to be processed accordingly, as described in the following section. Furthermore, to make it a fairer comparison and overcome some of the issues of the original dataset, the processed data was used to derive a new flow dataset.

3.3.1 Processing Raw Packet Capture Data

File Name	Approx. Size	Contains
Monday-WorkingHours.pcap	11Gb	Normal Activity
Tuesday-WorkingHours.pcap	11Gb	Normal Activity, FTP Bruteforce, SSH Bruteforce
Wednesday-WorkingHours.pcap	13Gb	Normal Activity, DoS/DDoS, Heartbleed
Thursday-WorkingHours.pcap	7.8Gb	Normal Activity, Web Attacks, Infiltration
Friday-WorkingHours.pcap	8.3Gb	Normal Activity, Botnet, Port Scan, DDoS

TABLE 3.3: CICIDS2017 raw PCAP data information

The pcap data is in 5 separate files (Table 3.3), each containing non-malicious/benign traffic, denoted as "normal activity" and attack/malicious traffic, except for Monday's file which contains only the former. Based on the information provided on the dataset website [51] and manual inspection using wireshark [68], the packets from each class were extracted into separate files and further split based on the transport layer protocol, i.e. UDP or TCP.

In order to transform the pcap data into an appropriate format for the algorithms, since they cannot work with pcaps, tshark [69] was used, which is a network protocol analyzer. The packet fields that were extracted are shown in Table 3.4. Originally, the field *ip.hdr_len* was also extracted, but was later dropped due to being constant.

It should be noted that a field being marked with "*" in the table, means the feature is not present in the packet itself, but is either recorded in the packet capture, in the case of *frame.time_epoch*, or, in the case of *tcp.stream* and *udp.stream*, generated by tshark. Finally, an additional feature, *label*, was manually added to each file, containing the appropriate class for each packet.

The first row in the resultant tab-separated value (TSV) file contains a header, indicating what each field is, and each following row represents information extracted from a single packet. The appropriate "stream" feature for each packet can be used to group them into flows, meaning that, essentially, each file contains multiple sequences each identifiable by a unique stream in said file, which can be used to train algorithms that require sequence data, and are also suitable to recalculate the features of the flow dataset accompanying the pcaps.

At this stage, the data was split into 25 files and needed to be merged which first required adjusting the stream values, so they remain unique per class.

The last issue remaining was merging the files with different protocols as each contained different features, which was achieved by adding any missing fields for UDP packets and having them set to zero, except for *tcp.hdr_len* which was set to 8,

Field	Value
frame.time_epoch*	Arrival time of the captured frame as seconds from the unix epoch
ip.len	IP Total Length
ip.src	IP source address
ip.dst	IP destination address
tcp.stream/udp.stream*	The TCP/UDP stream index as dissected by tshark
tcp.srcport/udp.srcport	TCP/UDP source port
tcp.dstport/udp.dstport	TCP/UDP destination port
tcp.hdr_len	TCP header length
tcp.flags.cwr	TCP CWR flag
tcp.flags.ece	TCP ECE flag
tcp.flags.urg	TCP URG flag
tcp.flags.ack	TCP ACK flag
tcp.flags.push	TCP PUSH flag
tcp.flags.reset	TCP RESET flag
tcp.flags.syn	TCP SYN flag
tcp.flags.fin	TCP FIN flag
tcp.window_size_value	TCP Window size

TABLE 3.4: Fields extracted from TCP/UDP packets using tshark

as that is always the length of the UDP header, and renaming the features *udp.stream*, *udp.srcport* and *udp.dstport* into their corresponding TCP "equivalents" as the overwhelming majority of packets/streams were TCP.

Class	# Flows	% of Total
BENIGN	1112569	71.694000
PORTSCAN	160369	10.334186
DOS-HULK	158977	10.244486
DDOS	95683	6.165817
DOS-GOLDENEYE	7647	0.492773
DOS-SLOWHTTPTEST	4217	0.271744
FTP-PATATOR	3992	0.257245
DOS-SLOWLORIS	3894	0.250930
SSH-PATATOR	2979	0.191967
BOTNET	737	0.047492
XSS	636	0.040984
BRUTEFORCE	113	0.007282
SQLI	13	0.000838
INFILTRATION	3	0.000193
HEARTBLEED	1	0.000064

(A) Flows per Class in sequence dataset

Class	# Packets	% of Total
BENIGN	51450213	92.078555
DOS-HULK	2247118	4.021584
DDOS	1280602	2.291846
PORTSCAN	323736	0.579378
SSH-PATATOR	163320	0.292288
FTP-PATATOR	111611	0.199746
DOS-GOLDENEYE	106177	0.190021
HEARTBLEED	49296	0.088223
DOS-SLOWLORIS	47591	0.085172
DOS-SLOWHTTPTEST	39662	0.070982
BRUTEFORCE	22546	0.040350
INFILTRATION	15592	0.027904
BOTNET	9872	0.017668
XSS	8958	0.016032
SQLI	140	0.000251

(B) Packets per Class in sequence dataset

3.3.2 Sequence Dataset

This dataset which was derived as described in the previous section (Section 3.3.1), is 55,876,434 rows long, meaning it contains information about an equal number of packets, split over 1,551,830 streams and contains 15 classes and 18 features (Table 3.4), including the label. It is imbalanced with respect to the class label, with over 70% of flows (Table 3.5a) and 92% of packets (Table 3.5b) belonging to the "BENIGN" class.

Training/Test Split

Due to the class imbalance present in the dataset, the same procedure outlined in Section 3.2.2 was followed, with each flow being considered a single datapoint. Additionally, within the "BENIGN" class the (*ip.src*, *ip.dst*) pair per flow were also taken into account.

However, an issue exists with the "HEARTBLEED" class, namely, that it consists of only one flow, which is temporarily fully assigned to the training set. This issue is dealt with in Section 4.2.1.

An 85/15 split is chosen, with the final numbers being 1,319,048 flows totaling 49,521,821 packets for the training set, and 232,781 flows totaling 6,305,317 packets for the test set.

Data Exploration

Exploring the data visually or using statistical methods can help with gaining new insights.

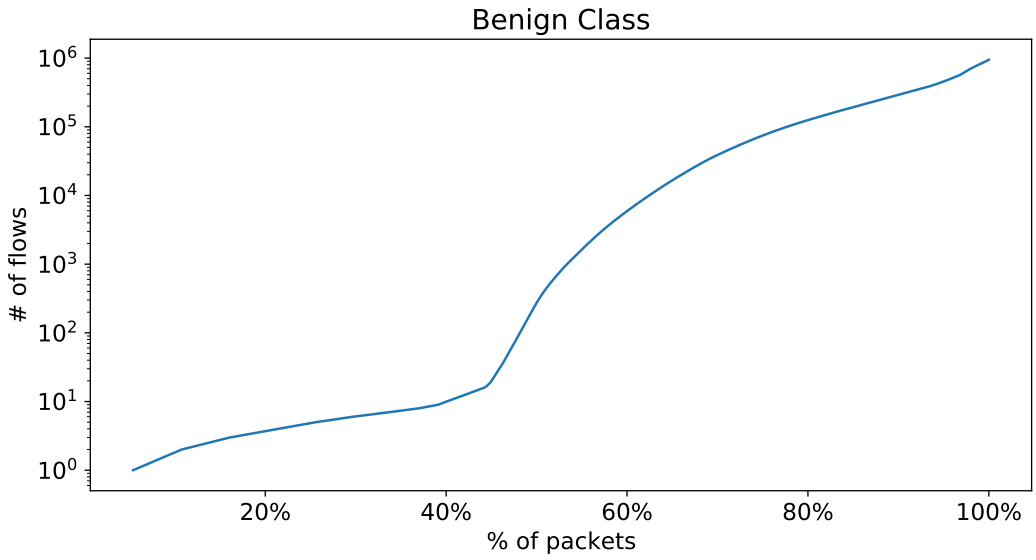

FIGURE 3.2: Number of Flows vs. Percentage of Packets for the "BENIGN" class

This section explores the training set of the sequence dataset, which contains, approximately, 49.5 million observations/packets and 1.32 million flows.

As already mentioned in Section 3.3.2, the "BENIGN" class has an order of magnitude more packets than those of all the others combined, however further examination reveals that a small number of flows, the top nine by number of packets, contain a disproportionate number of packets, 39.14% of the total "BENIGN" packets, as can be seen in Figure 3.2. Considering they are between two Windows 10 machines and the same IP which is allocated to Microsoft, and their sizes, they could likely be attributed to Windows Update.

Actually, the majority of flows consist of few packets, with a mean of 37.58 (23.99 if the top nine flows are excluded) and a median of 10 packets per flow.

Class	Attacker	CWR	ECE	URG	ACK	PUSH	RESET	SYN	FIN
BENIGN	0	0.0018	0.0029	0.0011	91.3142	10.9226	0.6053	2.2666	1.9354
	1	–	–	–	–	–	–	–	–
BOTNET	0	0.0000	0.0000	0.0000	84.7614	15.2142	0.0000	15.2386	15.2142
	1	0.0000	0.0000	0.0000	100.0000	15.3516	0.0000	14.1307	14.1307
BRUTEFORCE	0	0.0000	0.0000	0.0000	100.0000	94.2899	0.0000	1.4128	1.4128
	1	0.0000	0.0000	0.0000	99.2565	49.1071	0.0077	0.7358	0.7358
DDOS	0	0.0000	0.0000	0.0000	100.0000	19.3747	0.0000	18.5186	23.5595
	1	0.0000	0.0000	0.0000	87.4647	12.5291	12.5353	12.5290	0.0000
DOS-GOLDENEYE	0	0.0000	0.0000	0.0000	99.0658	19.4007	0.9342	19.2634	18.9798
	1	0.0000	0.0000	0.0000	86.8075	15.7445	3.9992	13.1925	0.0000
DOS-HULK	0	0.0000	0.0000	0.0000	99.9985	16.0583	0.0015	15.8963	16.3954
	1	0.0000	0.0000	0.0000	70.4781	15.1361	23.2368	14.2702	4.7251
DOS-SLOWHTTPTEST	0	0.0000	0.0000	0.0000	96.7612	14.8347	3.2388	25.8433	14.5830
	1	0.0000	0.0000	0.0000	28.7597	20.3039	3.1223	68.1180	5.6612
DOS-SLOWLORIS	0	0.0000	0.0000	0.0000	82.7610	4.7652	17.2390	39.9089	4.6718
	1	0.0000	0.0000	0.0000	75.4147	63.2950	1.2497	23.3356	1.7318
FTP-PATATOR	0	0.0000	0.0000	0.0000	88.2601	41.1877	11.7399	5.8916	5.8916
	1	0.0000	0.0000	0.0000	90.9023	63.5393	0.0000	9.0977	9.0897
HEARTBLEED	0	0.0000	0.0000	0.0000	100.0000	7.8529	0.0000	0.0048	0.0000
	1	0.0000	0.0000	0.0000	99.9930	4.3010	0.0035	0.0035	0.0035
INFILTRATION	0	0.0000	0.0000	0.0000	99.9801	8.0310	0.0000	0.0199	0.0099
	1	0.0000	0.0000	0.0000	99.9816	7.1020	0.0184	0.0369	0.0369
PORTSCAN	0	0.0000	0.0000	0.0000	99.9027	0.1149	98.7600	0.7096	0.1149
	1	0.0000	0.0000	0.0000	1.2167	0.0874	0.6221	98.3216	0.0788
SQLI	0	0.0000	0.0000	0.0000	100.0000	22.2222	0.0000	20.3704	20.3704
	1	0.0000	0.0000	0.0000	82.2581	19.3548	0.0000	17.7419	17.7419
SSH-PATATOR	0	0.0000	0.0000	0.0000	99.9638	45.3980	0.0386	3.0535	3.0909
	1	0.0000	0.0000	0.0000	95.4593	71.9961	0.0036	4.5371	4.5085
XSS	0	0.0000	0.0000	0.0000	100.0000	55.3425	0.0000	21.0176	21.1350
	1	0.0000	0.0000	0.0000	89.2277	28.3651	0.0000	10.7723	10.8325

TABLE 3.6: Percentage of packets that had the corresponding TCP flag set by Class and source (attacker, if "Attacker" column set to 1, victim otherwise)

Looking at which flags were set (Table 3.6), it can be seen that the CWR, ECE and URG flags appear to almost never be set, indeed they are only set for the "BENIGN" class, and only for 0.0018%, 0.0029% and 0.0011% of "BENIGN" class packets respectively. Furthermore, they are only set in 0.133%, 0.12% and 0.017% of "BENIGN"

class TCP flows respectively, or approximately half if UDP flows are included.

Another interesting observation is that class "PORTSCAN" appears to almost always have the SYN flag set for the attacker and RESET for the victim, which intuitively matches the behaviour of a portscan where the attacker makes many short connections to the victim to scan the state of the ports and the victim replies with a packet with the RESET flag set to indicate that the port is not listening.

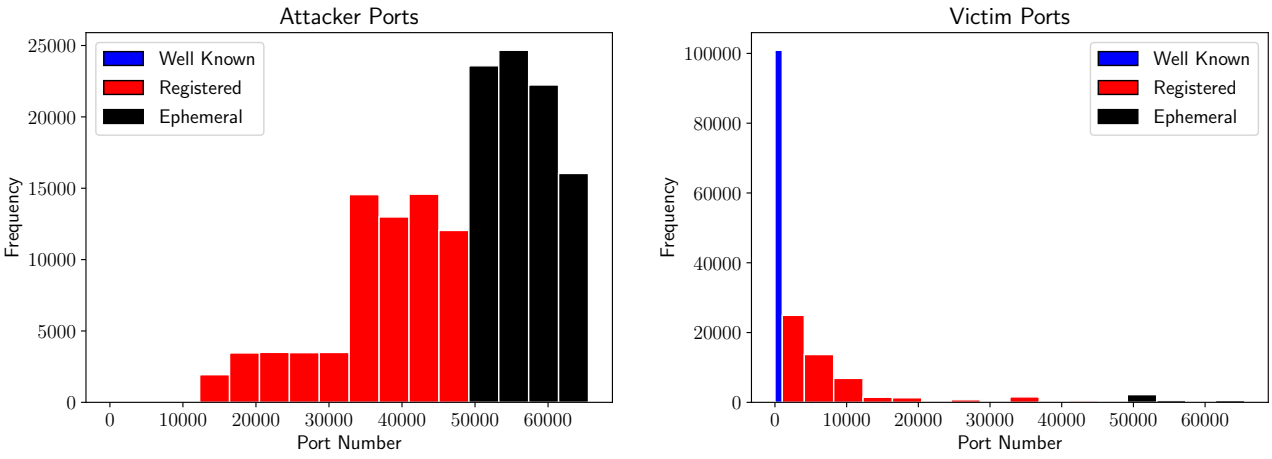

FIGURE 3.3: Port number histograms for attack traffic, for attackers and victims

The histograms in Figure 3.3, where ports are categorized based on the port number, as described in Section 2.4.4, show that victim ports tend to be in the "well-known" range which is usually where various services commonly run (port 22: SSH, port 80: HTTP, port 443: HTTPS etc.), whereas attacker ports tend to belong to the "registered" and "ephemeral" ranges which tend to be used by default.

Correlation Matrix

Looking at the heatmap/correlation matrix in Figure 3.4, the strongest correlation is between *tcp.srcport* and *tcp.dstport* and is -0.887604, which is reasonable considering services tend to listen at low-numbered ports while clients' connections usually originate from high-numbered ports. Other features with absolute correlations coefficients greater than 0.5 are *tcp.flags.ece* and *tcp.flags.cwr*, $corr = 0.62$, not surprising since they, in a sense, work together as explained in Section 2.4.2, and finally *tcp.srcport* and *tcp.dstport* with *ip.len*, with values of -0.58 and 0.59 respectively. Given what was already mentioned about which ports services and clients are more likely to use, this makes sense since services are likely to send bigger packets e.g. a client sends an HTTP GET request and the server replies with a whole webpage.

3.3.3 Producing a new Flow dataset

For the reasons laid out in Section 3.3, a new flow dataset with the same features as the original was produced by running the sequence data described in Section 3.3.2 through a custom program. Specifically, due to splitting packets into flows by tshark's *tcp.stream* and *udp.stream* fields, they will in all likelihood be different to those identified in the original dataset, and using the same flows for both the packet and flow classifiers should make for a fairer comparison.

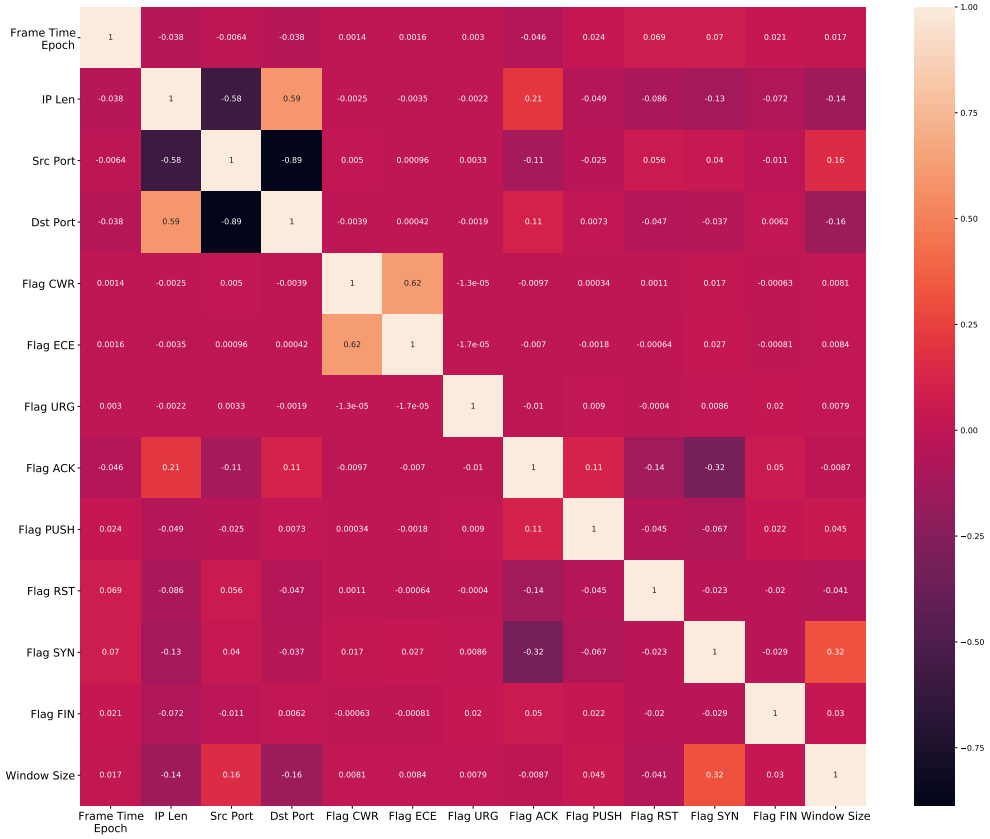

FIGURE 3.4: Heatmap of the correlation matrix for the Sequence Dataset

Additionally, some peculiarities of the original dataset (Section 3.2.1) were also addressed, such as the missing *Bwd Header Length* feature, *X Flag Counts* not being counts and the missing values for various features.

To be consistent with the original dataset, streams/flows with durations exceeding 120 seconds were split into multiple, each with a max duration of 120 seconds. Moreover, flows consisting of a sole packet were removed.

The final features of the new flow dataset were,

- For the **Flow**:

  Duration in seconds

  Min, Max, Mean, Std & Var Packet Length

  Packets/s

  Bytes/s

  Down/Up Packet Ratio

  Min, Max, Mean & Std IAT

  Min, Max, Mean & Std Idle

  Min, Max, Mean & Std Active

  Total number of CWR, ECE, ACK, RST, SYN, URG & FIN TCP flags

  Destination and Source Port

- For both **Forward and Backward directions**:

Total Number of Packets

Total, Min, Max, Mean & Std Packet Length

Total, Min, Max, Mean & Std IAT

Initial TCP Window Size

Mean Packet number per Subflow

Mean Bytes (Packet Length) per Subflow

Packets/s

Total TCP Header Length (i.e. sum of length of all TCP headers)

Total number of PUSH Flags

- In the **Forward direction only**:

  Min Header Size

  Total number of Active data packets (i.e. with non-zero packet length)

- Label

LIST 3.2: New Flow Dataset Final Features

Where,

- *Packet Length* refers to the number of "useful" bytes, i.e. data bytes, in a TCP segment, calculated by subtracting the IP and TCP header lengths from the IP "Total Length" field.
- *IAT*: 'Inter-Arrival Time' refers to the time elapsed between two successive packets in a flow, sent in either, or a specific direction.
- *Idle* phases are phases in a flow where no packets have been sent for time exceeding a threshold τ, in this case, one second.
- *Active* phases are the complements of idle phases, i.e. phases during which there are no lulls in the flow for time exceeding the threshold τ.
- *Subflow*: When a flow has been idle, each non-idle period is defined to be a *subflow*.

Training/Test Split

The same procedure outlined in Section 3.2.2 was followed, with the same split ratio i.e. 85% of the data assigned to the training set and 15% to the test set, which translates to 1,767,299 flows/datapoints for the training set and 312,674 for the test set.

Correlation Matrix

The heatmap in Figure 3.5 shows there is a considerable number of pairs of features with high degrees of correlation for the new flow dataset as well,

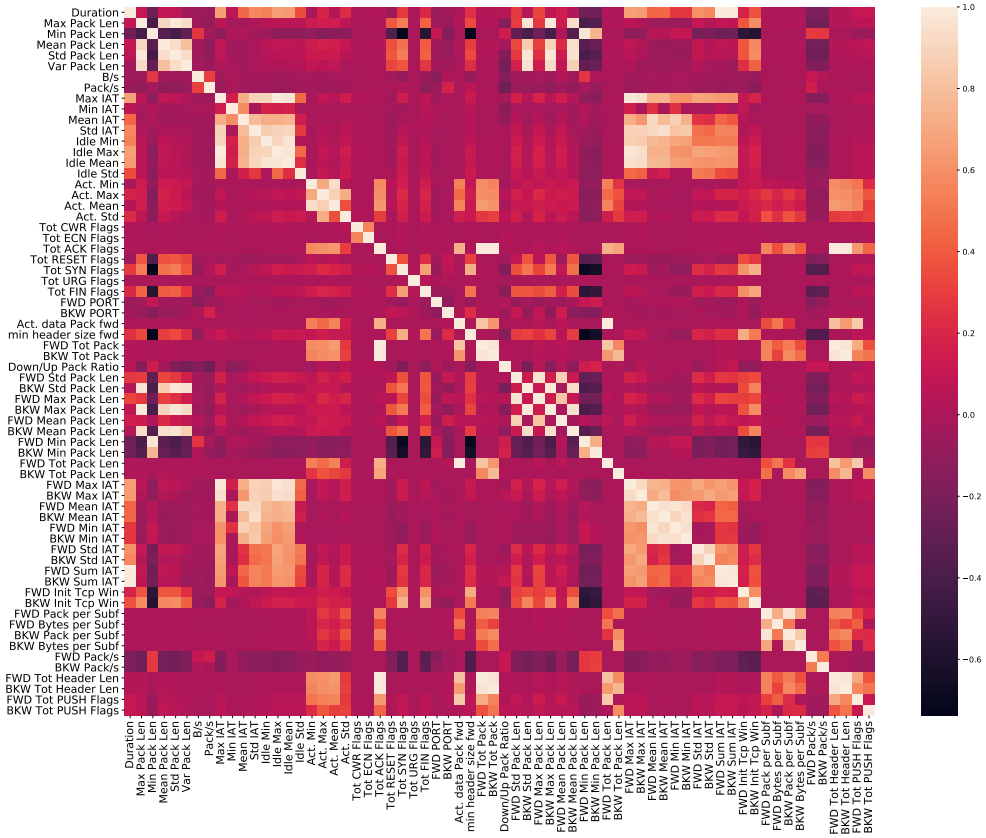

FIGURE 3.5: Heatmap of the correlation matrix for the New Flow Dataset

$$0.50 < abs(corr) \leq 0.80 : 164 \text{ pairs}$$
$$0.80 < abs(corr) \leq 0.90 : 27 \text{ pairs}$$
$$0.90 < abs(corr) \leq 1.00 : 56 \text{ pairs}$$

with the highest being (absolute correlation over 0.999),

1. “Max IAT”, “Idle Max”: 0.9999
2. “active data packets fwd”, “FWD Total Packet Length”: 0.9998
3. “FWD Total Packets”, “FWD Total Header Length”: 0.9994
4. “BKW Total Packets”, “BKW Total Header Length”: 0.9991

Chapter 4

Experimental Neural Network Model Evaluation for Intrusion Detection

4.1 Data Preprocessing

Certain processing steps need to be performed before the data is ready to be used to train models, a process known as "Data Preprocessing/Preparation". They are as follows,

- **Data Cleaning -** where problems in the dataset, such as inaccuracies or errors are corrected. Performed in Section 3.2.1.
- **Dealing with Missing Values -** it is the first step to transforming the data as all the following techniques assume there is no missing data in the dataset. Usually when handling missing values, either the observations containing them are removed from the dataset, or even the features, usually when the percentage of missing values is very high e.g. 90% of a feature's values are NaNs, or, alternatively, the missing values are imputed, meaning that they are substituted with a computed value e.g. the mean, median or mode of the feature, or by training a model to do the substitution.
- **Encoding Categorical Variables -** Generally, models expect numeric inputs, so categorical variables need to be transformed to such. Two popular methods are,

  Ordinal Encoding - where each of the n categories is mapped to an integer value from 0 to $n-1$. However, doing this implies some order exists between the categories, so it is not always appropriate, for example, if the variable is "colour" with values "red", "green" and "blue", applying ordinal encoding could imply to the model "red" $<$ "green" $<$ "blue" which is false and unwanted.

  One-Hot Encoding - where each category is mapped to a new feature, with the values of 0 and 1 indicating its absence or not respectively. It should be noted that n categories should generally be mapped to $n-1$ new features, with the remaining category being present when all the others are absent i.e. all $n-1$ new features are equal to 0. Using the "colour" example from above, two new features would be added, e.g. *colour_ red (R)* and *colour_ green (G)*, so observations with colours "red", "green" and "blue" would have $(R=1, G=0)$, $(R=0, G=1)$, $(R=0, G=0)$ set respectively. This method fixes the issue of ordinal encoding at the expense of introducing more features.

- **Feature Selection -** which involves selecting a subset of features for training by discarding irrelevant, low-information or redundant features.
- **Feature Construction -** where new features are created by combining existing ones, e.g. using a feature *timestamp* to create a new feature *Duration* by subtracting the start timestamp from the end timestamp for an event.
- **Removing Duplicates -** where, as the name suggests, duplicate observations are removed.
- **Feature Scaling & Normalization -** which scales the data to a new range, which can help make training faster. Some popular methods are,

  Standardization - which is calculated feature-wise as

  $$x' = \frac{x - \overline{x}}{s}$$

  where $\overline{x}$ is the sample mean and s is the sample standard deviation of a feature. The scaled data has zero mean and unit variance.

  Min-max scaling - which is also calculated feature-wise as follows,

  $$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

  where x_{max}, x_{min} are the maximum and minimum values of the feature respectively, and maps the data to the range $[0, 1]$.
- **Handling Imbalanced Data -** In classification, when the number of observations per class is significantly different between classes e.g. in binary classification, 80% of examples belonging to class A, while only 20% to class B, the data is considered "imbalanced". Training on imbalanced data can cause the model to focus less on the minority class. Furthermore, using improper evaluation metrics can overestimate the classifier's performance, as explained in Section 2.2.9.

  Some of the most popular techniques to ameliorate class imbalance are:

  Undersampling the majority class(es), meaning only using a fraction of the available examples.

  Oversampling the minority class(es), essentially increasing the number of samples in the minority class(es) with copies of existing observations.

  Generating synthetic observations from the minority class(es), which increases their cardinality like oversampling does, only instead of repeating samples, new but similar to existing ones are created with algorithms such as SMOTE and ADASYN.

  Cost-sensitive training An alternative to resampling methods is increasing the cost of misclassifying the minority class(es), giving them more "importance" during model training.

  These techniques can also be combined e.g. undersampling the majority class and generating synthetic examples for the minority class. However, it should be noted that using them is not always necessary if a classifier performs well enough without them, nor are they guaranteed to improve performance.

- **Transforming Labels -** The final step before the data is ready, is to transform the labels to the format required by the cost function of the given framework/library.

4.2 Prerocessing Steps

This section describes the preprocessing that was applied to the three datasets, first focusing procedures specific to each dataset and finishing with general steps that were applied to all.

Firstly, at this point for the flow datasets, 15% of each training set is set aside, in a stratified manner with respect to the class as described in Section 3.2.2, to form validation sets.

4.2.1 Sequence Dataset Specific

While the average number of packets per flow is 37.58 (48.42 when the "BENIGN" class is considered alone), there exist extremely long flows, as described in Section 3.3.2, some with as many as (approximately) 2.5 million packets. In order to avoid said extremely long flows dominating within their class, flows are capped to a length of 50,000 packets.

In order to train models more efficiently, the number of packets a model is expected to receive, *window*, should be fixed. *Window* can not be very wide in order to avoid having to split most flows into multiple *windows*: by setting it to a number as low as 20 packets, which the value we chose for this Thesis, over 70% of flows are included in one piece (a single *window*).

Flows with numbers of packets exceeding *window* are handled by being split into *window* sized flows. Finally, flows with only 1 packet are discarded and those left are padded to *window* size.

It is at this point that 15% of the training set is set aside as a validation set. Furthermore, the "HEARTBLEED" class due to only having one flow could not be split before this point. After being divided into new, *window* sized flows, it is split in the same manner as the others i.e. 15% for testing, 85% for training and 15% of that for validation.

Feature Selection & Construction

From the source and destination IP addresses, *ip.src* and *ip.dst*, a new categorical feature named *attacker_src* is created, after which they are removed from the dataset. That feature takes values zero and one, with the former indicating a packet sent from a potential victim i.e. a host the IDS would be protecting and the latter indicating a packet sent from an external host to the protected network (or the gateway/firewall), meaning *attacker_src* also implicitly indicates the packet direction. In the case where the communication is between two protected hosts, it only indicates the direction, with the packets of the host that initiated communication having *attacker_src* set to zero.

Another recorded feature is *frame.time_epoch*, however the absolute capture time of the packet is not relevant, so it is used to derive two new, more meaningful, features, *frame.inter_arrival_time* and *frame.direc_inter_arrival_time*, and then dropped. The former refers to the inter-arrival time between successive packets in the flow regardless of direction, and the latter to the inter-arrival times between successive packets in the same direction e.g. given the following for packets $(0, 0)$, $(1, 0.5)$, $(0, 0.6)$,

$(1, 1.2)$ in the format $(\text{direction}, \text{timestamp})$ the former would be equal to $(0, 0.5, 0.1, 0.6)$ while the latter would be equal to $(0, 0, 0.6, 0.7)$

Finally, the features *tcp.flags.cwr*, *tcp.flags.ece* and *tcp.flags.urg* are removed, as are only set for the "BENIGN" class and even then very rarely (for 0.133%, 0.12% and 0.017% of TCP flows respectively, and even less if UDP flows are included).

4.2.2 Original Flow Dataset Specific

Class	# Missing	% Missing
BENIGN	688785	44.250576
DoS Hulk	110	0.087618
PortScan	97	0.108585
DDoS	14	0.015136
Bot	9	0.634249
FTP-Patator	5	0.109409
DoS GoldenEye	5	0.067268
Heartbleed	3	37.500000
Infiltration	1	3.846154
Brute Force (Web)	0	0.000000
Sql Injection	0	0.000000
XSS	0	0.000000
SSH-Patator	0	0.000000
DoS slowloris	0	0.000000
DoS Slowhttptest	0	0.000000

(A) Observations having at least one missing value per class

Feature	# Missing	% Missing
Init_Win_bytes_forward	685750	36.299528
Init_Win_bytes_backward	680434	36.018131
Flow IAT Min	2096	0.110950
Flow Packets/s	1290	0.068285
Flow Bytes/s	1268	0.067120
Flow IAT Max	85	0.004499
Flow Duration	85	0.004499
Flow IAT Mean	85	0.004499
min_seg_size_forward	27	0.001429
Fwd Header Length	27	0.001429
Bwd Header Length	17	0.000900
Fwd IAT Min	13	0.000688

(B) Missing values per feature

The original flow dataset has missing values, displayed for the training set per class in Table 4.1a. Out of 1,889,143 instances in the training set, 689,029 or 36.47% have missing values distributed over the features as seen in Table 4.1b, which are imputed using an iterative method, "IterativeImputer" from scikit-learn [18], where each feature is estimated from the others. The missing values in the validation and test sets are then imputed using the model trained on the training set.

4.2.3 General Preprocessing Steps

Transforming Port Features

In all experiments, for every dataset, their features that had to do with port numbers (*Destination Port* for the original flow dataset, *FWD Port* and *BKW Port* for the new flow dataset and *tcp.srcport* and *tcp.dstport* for the sequence dataset) were processed in the following way: those features happen to take specific values based on the service configurations of the network the data was gathered from and the random high-numbered ports the OS happened to use when making connection to those services, but this is not something the model should learn to rely on. Therefore, the port numbers are binned into three categories 'Well-Known", "Registered" and "Ephemeral" further described in Section 2.4.4, which are then one-hot encoded.

De-duplicating Instances

There existed some duplicate examples in the datasets, though not across training, validation or test sets due to careful splitting. Encoding the port numbers to only three categories further increased that number, so in this step all duplicate instances are removed.

Feature Scaling/Standardization

The features of each dataset that are boolean/dichotomous are left as is. The others are standardized i.e. transformed to have unit variance and zero mean. It should be noted that the validation and test sets are standardized using the mean and standard deviation computed from the corresponding training sets to avoid data dredging. For the sequence dataset, the padding appended to flows shorter than *window* is not taken into account when calculating the mean and standard deviation, that is to say, it is removed before each feature is standardized and is then reinserted.

Further Scaling & Clipping

For some experiments on the flow datasets, additional scaling was applied before and/or after standardizing as described above. Specifically, features with ranges exceeding some number, e.g. 1,000,000, had either the square root function, or the common logarithm function applied to them to reduce said range before standardizing.

Another transformation that was tried in some experiments was clipping the data after it had been standardized. Given an interval $[min, max]$, clipping means that values in the data d outside the interval are clipped to the values of the endpoints i.e.

$$\text{clip}(min, max, d) = \begin{cases} min, & \text{if } d < min \\ max, & \text{if } d > max \\ d, & \text{otherwise} \end{cases}$$

4.3 Experiments: Typical Parameters

The data was preprocessed as described above, with further scaling (Section 4.2.3) not applied unless indicated otherwise.

Early stopping, described in Section 2.2.6, was used for all models. Furthermore, the learning rate was decreased when the monitored metric, either validation loss or validation AUC stopped improving for a number of epochs n, using Keras' [19] "ReduceLROnPlateau" callback.

Generally, relatively large batch sizes were selected, as testing showed no significant difference in model performance, and they drastically decreased training time, allowing for more model configurations to be tested. Additionally, for the classification experiments, having larger batch sizes increases the likelihood that instances from minority classes are present in each batch.

The Adam optimizer [70] was selected as the optimizer for all models, with all its hyperparameters set to the default values of the Keras' implementation save for the learning rate, which was chosen through trial and error.

Finally, for the binary classifiers, considering the imbalanced nature of the datasets, the decision thresholds were selected such that the F_1 Scores on the respective validation sets were maximized. The same process was adjusted for the autoencoder models for selecting the best reconstruction error thresholds.

4.4 Experiments: Classification

For classification, both binary and multi-class classification models were trained for each dataset.

4.4.1 Original Flow Dataset

Class	Test Set # Obs.	Binary Recall	Multi-Class Precision	Multi-Class Recall	Multi-Class F_1 Score
BENIGN	309,727	99.909	99.464	99.790	99.627
Web Attack - Brute Force	224	98.661	87.500	6.250	11.667
Web Attack - Sql Injection	3	66.667	0.000	0.000	0.000
Web Attack - XSS	98	94.898	0.000	0.000	0.000
DoS Hulk	26,349	97.909	98.052	97.818	97.935
Infiltration	5	0.000	100.000	20.000	33.333
SSH-Patator	558	97.491	100.000	77.957	87.613
FTP-Patator	982	97.352	95.779	97.047	96.409
DoS slowloris	847	99.410	97.995	92.326	95.076
DoS Slowhttptest	799	99.374	90.185	97.747	93.814
Heartbleed	2	0.000	66.667	100.000	80.000
PortScan	967	99.690	98.814	60.290	74.888
DDoS	19,204	99.948	99.844	99.885	99.865
Bot	255	40.000	96.078	38.431	54.902
DoS GoldenEye	1,543	99.417	99.208	97.472	98.333

TABLE 4.2: For the "Original Flow Dataset", the Recall (TPR) of the Binary Classifier for each Class, along with the Precision, Recall and F_1 Score for the Multi-Class Classifier for each class.

Binary Classification

For this model, additional scaling was applied to the data, where features with ranges exceeding 100,000 had the square root function applied to them to reduce it, and the data was clipped to the range $[-5, 5]$ after being standardized.

The best model had 3 layers, with 110 neurons each. The focal loss function (Section 2.2.8) was used, with the balancing parameter set to the class frequency of the majority class. The learning rate was set to 0.0007, dropout to 0.5 and the batch size to 512.

Table 4.3a indicates a powerful model, with a high F_1 Score of 98.95% and AUC of 99.989%, with low False Positive (0.091%) and False Discovery (0.551%) Rates. Table 4.2 reveals, unsurprisingly, that the model faces issues correctly predicting those classes with few examples in the dataset. It also struggles with the "BOT" class, possibly due to its behaviour being less obviously malicious given the information

Binary Classifier Metrics	%
Accuracy	99.703
Balanced Accuracy	99.190
AUC	99.988
Precision	99.449
Recall	98.472
F_1 Score	98.958
Specificity	99.909
FDR	0.551
FPR	0.091

(A) Original Flow Dataset: Binary Classifier Metrics

Multi-Class Classifier Metric		%
Accuracy	—	99.34
	Balanced	65.67
AUC	OVO	98.64
	OVR	99.87
Macro Avg.	Precision	81.97
	Recall	65.67
	F_1 Score	68.23
Weighted Avg.	Precision	99.31
	Recall	99.34
	F_1 Score	99.28

(B) Original Flow Dataset: Multi-class Classifier Metrics

available to the model than e.g., a "PORTSCAN" attack, combined with a still relatively low number of examples.

Multi-Class Classification

A 2 layer model with 128 neurons in each layer, trained by minimizing the crossentropy loss performed best. The learning rate was set to 0.001, dropout to 0.5 and the batch size to 128. This model doesn't perform very well, struggling with many classes, as Tables 4.2 and 4.3b show.

4.4.2 New Flow Dataset

Binary Classification

For this model, the same additional scaling was applied to the data as was to the binary classifier for the original flow dataset, that is to say, the square root was applied to features having ranges over 100,000 and the data was clipped to the range $[-5, 5]$ after being standardized.

The model had 3 layers each with 96 neurons, the learning rate was set to 0.001, dropout to 0.5 and the batch size to 512, with the binary crossentropy loss being minimized.

According to Table 4.4, the model was generally successfull at detecting a variety of attacks, only having major trouble with classes "INFILTRATION" and "SQLI", both of which had exceedingly few examples, 4 and 2 respectively.

Even so, its overall ability to detect attacks was high (Recall $=$ 99.958%), even while keeping both FDR and FPR very low, at 0.311% and 0.084% respectively.

Multi-Class Classification

The model had 2 layers of 128 neurons each, with the learning rate set to 0.001, dropout to 0.5, the batch size to 128 and the crossentropy loss being chosen as the loss function.

While this model detected most attacks adequately (Table 4.4), it really struggled with all those with very low cardinality i.e. the same as the binary classifier, with

Class	Test Set # Obs.	Binary Recall	Multi-Class Precision	Multi-Class Recall	Multi-Class F_1 Score
BENIGN	246,196	99.916	99.962	99.912	99.937
BOTNET	110	99.091	99.091	99.091	99.091
DDOS	14,390	100.000	99.965	99.951	99.958
DOS-GOLDENEYE	1,147	100.000	99.912	98.518	99.210
DOS-HULK	24,021	99.992	99.913	99.979	99.946
DOS-SLOWHTTPTEST	767	99.739	93.325	98.435	95.812
DOS-SLOWLORIS	856	99.416	98.465	97.430	97.945
SSH-PATATOR	448	99.107	99.773	98.214	98.988
FTP-PATATOR	600	99.833	99.833	99.667	99.750
HEARTBLEED	2	100.000	25.000	50.000	33.333
INFILTRATION	4	0.000	0.000	0.000	0.000
PORTSCAN	24,019	99.983	99.399	99.850	99.624
SQLI	2	0.000	0.000	0.000	0.000
BRUTEFORCE (WEB)	17	100.000	87.500	82.353	84.848
XSS	95	96.842	93.407	89.474	91.398

TABLE 4.4: For the "New Flow Dataset", the Recall (TPR) of the Binary Classifier for each Class, along with the Precision, Recall and F_1 Score for the Multi-Class Classifier for each class.

Binary Classifier Metrics	%
Accuracy	99.925
Balanced Accuracy	99.937
AUC	99.996
Precision	99.690
Recall	99.958
F_1 Score	99.824
Specificity	99.916
FDR	0.311
FPR	0.084

(A) New Flow Dataset: Binary Classifier Metrics

Multi-Class Classifier Metric		%
Accuracy	—	99.89
	Balanced	80.86
AUC	OVO	99.90
	OVR	99.97
Macro Avg.	Precision	79.70
	Recall	80.86
	F_1 Score	79.99
Weighted Avg.	Precision	99.89
	Recall	99.89
	F_1 Score	99.89

(B) New Flow Dataset: Multi-class Classifier Metrics

the addition of the "HEARTBLEED", and less prominently, "BRUTEFORCE (WEB)" and "XSS" classes.

4.4.3 Sequence Dataset

Various LSTM network configurations were tried, and although all of them performed very well, that is, having AUCs exceeding 0.9999 and F_1 scores over 0.9979, the simpler models described bellow performed marginally better.

Class	Test Set # Obs.	Binary Recall	Multi-Class Precision	Multi-Class Recall	Multi-Class F_1 Score
BENIGN	327,797	99.960	99.994	99.961	99.977
BOTNET	125	100.000	99.206	100.000	99.602
DDOS	14,398	99.979	100.000	99.986	99.993
DOS-GOLDENEYE	1,168	100.000	99.742	99.229	99.485
DOS-HULK	23,905	99.996	99.908	99.900	99.904
DOS-SLOWHTTPTEST	645	100.000	99.688	99.225	99.456
DOS-SLOWLORIS	638	100.000	96.078	99.843	97.925
SSH-PATATOR	1,341	99.776	100.000	99.702	99.851
FTP-PATATOR	1,194	100.000	100.000	100.000	100.000
HEARTBLEED	369	100.000	100.000	100.000	100.000
INFILTRATION	6	0.000	0.000	0.000	0.000
PORTSCAN	3,252	100.000	96.248	99.385	97.791
SQLI	2	100.000	100.000	50.000	66.667
BRUTEFORCE (WEB)	144	100.000	100.000	98.611	99.301
XSS	137	100.000	99.275	100.000	99.636

TABLE 4.6: For the "Sequence Dataset", the Recall (TPR) of the Binary Classifier for each Class, along with the Precision, Recall and F_1 Score for the Multi-Class Classifier for each class.

Binary Classifier Metrics	%
Accuracy	99.961
Balanced Accuracy	99.966
AUC	99.9996
Precision	99.722
Recall	99.973
F_1 Score	99.847
Specificity	99.960
FDR	0.278
FPR	0.040

(A) Sequence Dataset: Binary Classifier Metrics

Multi-Class Classifier Metric		%
Accuracy	—	99.95
	Balanced	89.72
AUC	OVO	99.51
	OVR	99.37
Macro Avg.	Precision	92.68
	Recall	89.72
	F_1 Score	90.64
Weighted Avg.	Precision	99.95
	Recall	99.95
	F_1 Score	99.95

(B) Sequence Dataset: Multi-class Classifier Metrics

Binary Classification

The LSTM network had 128 units, the learning rate was set to 0.003 and dropout to 0.4. The crossentropy loss was selected with a batch size of 256.

As shown in Table 4.7a the model performs very well, having a very high AUC ($>$ 99.99%) and relatively high F_1 Score (99.84%). It appears to only struggle significantly with the "INFILTRATION" class (Table 4.6), possibly due to it having very few examples. Surprisingly, despite having even fewer examples available, class "SQLI" is recognised as an attack without fail.

Multi-Class Classification

The multi-class classification model had 50 units, the learning rate was set to 0.002, dropout to 0.3, with the crossentropy loss being minimized and a batch size of 256.

This model also performs quite well, except for misclassifing "INFLITRATION" attacks as "BENIGN" activity, just like the binary classifier. That, in addition to getting one of the two "SQLI" instances wrong, hurts its performance.

4.5 Experiments: Anomaly detection

For anomaly detection autoencoders were trained on the flow datasets, and LSTM-autoencoders on the sequence dataset.

Autoencoders, which were described in Section 2.3.6, can be used to detect anomalous examples in the following manner: they are only trained on "normal" data and since they try to minimize the reconstruction error and "abnormal" datapoints are assumed to be dissimilar to "normal" ones in some manner, it is reasonable to expect that when presented with "abnormal" datapoints, they will struggle to accurately reconstruct them. In other words, the reconstruction error for "abnormal" samples will be higher than that for "normal" ones.

So, after training them, a reconstruction error threshold, different for each model, is chosen and instances whose reconstruction error surpasses that threshold are classified as anomalies. LSTM-autoencoders work much in the same way as autoencoders, only they are capable of efficiently dealing with sequence data as the encoder and decoder parts are fully-fledged LSTMs.

The steps outlined in Section 4.1 are followed for data preparation, only, before the standardization happens, all "abnormal" instances are removed from the training (and validation) sets, since the models should only be trained on the "normal" instances. The removed "abnormal" instances are used instead for model and optimal threshold selection.

All the best models had an additional constraint placed on them which penalized model layer outputs, specifically, a Keras' L_1 activity regularizer.

Another thing to note is that all the models were symmetric i.e. the decoder is a mirrored encoder e.g. given n features, encoder layer sizes of $[64, 32]$ and a code dimension of 16, the layer sizes of the full model would be $[64, 32, 16, 32, 64, n]$. The final n-sized layer is there to match the input and output shapes.

Finally, the MSE loss (Section 2.2.8) function was selected to be minimized during training, and for predictions, the MSE and MAE were calculated as potential reconstruction errors for every model.

4.5.1 Original Flow Dataset

The best model had layer sizes of $[100, 60]$ for the encoder and a code dimension of 33. The learning rate was set to 0.003, L_1 regularizer strength to $5e{-}5$ and batch size to 256. The MAE reconstruction error performed better when classifying anomalies.

It performed poorly, as shown in Table 4.8b. While its Recall is 90.2%, its Precision is very low (50.25%) and FPR quite high, at almost 15%.

Class	Test Set	
	# Obs.	TPR
BENIGN	309,727	85.0559
Web Attack - Brute Force	224	9.3750
Web Attack - Sql Injection	3	33.3333
Web Attack - XSS	98	2.0408
DoS Hulk	26,349	88.5499
Infiltration	5	100.0000
SSH-Patator	558	78.8530
FTP-Patator	982	62.1181
DoS slowloris	847	99.6458
DoS Slowhttptest	799	99.7497
Heartbleed	2	100.0000
PortScan	967	59.4623
DDoS	19,204	96.4174
Bot	255	40.3922
DoS GoldenEye	1,543	97.8613

(A) Original Flow Dataset: Per class TPR

Metrics	%
Accuracy	85.7939
Balanced Accuracy	87.6298
AUC	94.3070
Precision	50.2536
Recall	90.2037
F_1 Score	64.5472
Specificity	85.0559
FDR	49.7464
FPR	14.9441

(B) Original Flow Dataset: Anomaly Detection Metrics

Class	Test Set	
	# Obs.	TPR
BENIGN	246,196	92.6888
BOTNET	110	4.5455
DDOS	14,390	99.8332
DOS-GOLDENEYE	1,147	93.4612
DOS-HULK	24,021	99.5712
DOS-SLOWHTTPTEST	767	99.8696
DOS-SLOWLORIS	856	92.0561
SSH-PATATOR	448	95.7589
FTP-PATATOR	600	99.3333
HEARTBLEED	2	100.0000
INFILTRATION	4	50.0000
PORTSCAN	24,019	50.9014
SQLI	2	0.0000
BRUTEFORCE (WEB)	17	82.3529
XSS	95	2.1053

(A) New Flow Dataset: Per class TPR

Metrics	%
Accuracy	90.3120
Balanced Accuracy	87.0992
AUC	94.0108
Precision	75.0644
Recall	81.5097
F_1 Score	78.1544
Specificity	92.6888
FDR	24.9356
FPR	7.3112

(B) New Flow Dataset: Anomaly Detection Metrics

4.5.2 New Flow Dataset

The best model had layer sizes of $[128, 64]$ for the encoder and a code dimension of 17. The learning rate was set to 0.007, L_1 regularizer strength to 0.001 and batch size to 128. The MSE reconstruction error performed better when classifying anomalies.

This model also does not perform very well, it has a F_1 Score of 78%, and its FPR is high with a value of 7.3%.

Class	Test Set # Obs.	TPR
BENIGN	327,797	98.1281
BOTNET	125	40.8000
DDOS	14,398	99.9722
DOS-GOLDENEYE	1,168	97.5171
DOS-HULK	23,905	99.9414
DOS-SLOWHTTPTEST	645	98.1395
DOS-SLOWLORIS	638	99.0596
SSH-PATATOR	1,341	99.4780
FTP-PATATOR	1,194	99.8325
HEARTBLEED	369	100.0000
INFILTRATION	6	33.3333
PORTSCAN	3,252	37.0234
SQLI	2	100.0000
BRUTEFORCE (WEB)	144	97.9167
XSS	137	100.0000

(A) Seq. Dataset: Per class TPR

Metrics	%
Accuracy	97.7770
Balanced Accuracy	96.7365
AUC	99.0093
Precision	88.0290
Recall	95.3449
F_1 Score	91.5410
Specificity	98.1281
FDR	11.9710
FPR	1.8719

(B) Seq. Dataset: Anomaly Detection Metrics

4.5.3 Sequence Dataset

The best model had encoder layer sizes of $[64, 32]$ and a code dimension of 16. The learning rate was set to 0.003, L_1 regularizer strength to $1e{-}4$ and batch size to 256.

Once the code for an input sequence had been calculated, the decoding worked as follows: the code vector was repeated *timestep* times, and given as input to the decoder, which produced one sequence point per step.

To get a scalar reconstruction error, the average of the *timestep*-dimensional vector containing the reconstruction errors for each packet/timestep of each sequence was taken. The MSE reconstruction error performed better for classifying anomalies.

This model outperformed those trained on the flow-based datasets as can be seen in Table 4.10b.

4.6 Experiments: Training Classifiers on Autoencoder codes

While autoencoders were shown in Section 4.5 to fare poorly when used to directly classify examples based on the reconstruction error, they could instead be used as a dimensionality reduction technique. This was done by removing the decoder part of the networks selected in Section 4.5, giving them their respective datasets' training, validation and test sets as input and getting the reduced dimensionality "codes" as output. That output was then used to train new binary and multi-class classifiers. For all those models the crossentropy loss was used.

4.6.1 Original Flow Dataset

The number of features of this dataset was reduced to 33, however examining the new features revealed that 3 of them had a constant value of zero and were therefore removed, further dropping the number of features to 30.

Class	Test Set # Obs.	Binary Recall	Multi-Class Precision	Multi-Class Recall	Multi-Class F_1 Score
BENIGN	309,727	99.90	99.51	99.88	99.69
Web Attack - Brute Force	224	19.64	33.59	19.20	24.43
Web Attack - Sql Injection	3	66.67	0.00	0.00	0.00
Web Attack - XSS	98	3.06	100.00	2.04	4.00
DoS Hulk	26,349	97.30	99.26	97.31	98.27
Infiltration	5	0.00	100.00	40.00	57.14
SSH-Patator	558	96.24	99.55	78.85	88.00
FTP-Patator	982	97.25	98.16	97.96	98.06
DoS slowloris	847	99.29	98.67	96.58	97.61
DoS Slowhttptest	799	97.25	89.50	98.12	93.61
Heartbleed	2	100.00	100.00	100.00	100.00
PortScan	967	71.15	97.32	71.35	82.34
DDoS	19,204	99.79	99.71	99.88	99.80
Bot	255	40.00	98.10	40.39	57.22
DoS GoldenEye	1,543	97.47	98.69	97.73	98.21

TABLE 4.11: For the "Original Flow Dataset" and Classifiers trained on the AE "codes", the Recall (TPR) of the Binary Classifier for each Class, along with the Precision, Recall and F_1 Score for the Multi-Class Classifier for each class.

Binary Classifier Metrics	%
Accuracy	99.4795
Balanced Accuracy	98.4272
AUC	99.9571
Precision	99.4027
Recall	96.9519
F_1 Score	98.1620
Specificity	99.9025
FDR	0.5973
FPR	0.0975

(A) Original Flow Dataset (AE codes): Binary Classifier Metrics

Multi-Class Classifier Metric		%
Accuracy	—	99.44
	Balanced	69.29
AUC	OVO	98.52
	OVR	99.82
Macro Avg.	Precision	87.47
	Recall	69.29
	F_1 Score	73.23
Weighted Avg.	Precision	99.42
	Recall	99.44
	F_1 Score	99.40

(B) Original Flow Dataset (AE codes): Multi-class Classifier Metrics

Binary Classification

The best model had 2 layers, with 256 neurons each. The learning rate was set to 0.003, dropout to 0.4 and the batch size to 256.

The model does not perform badly, with an AUC of 99.95% and a F_1 Score of 98.16%, especially compared to its reconstruction error counterpart, though Table 4.11 shows it struggling detecting many kinds of attacks.

Multi-Class Classification

A 2 layer model with 128 neurons in each layer performed best. The learning rate was set to 0.001, dropout to 0.35 and the batch size to 256.

Notably, this model scores comparably (e.g. AUC OVR 99.82% vs. 88.87%) or better (e.g. Balanced Acc. 69.29% vs 65.67) in all metrics compared against the multi-class model trained on the full (i.e. with the full number of features) dataset (see Tables 4.12b, 4.3b).

4.6.2 New Flow Dataset

Class	Test Set # Obs.	Binary Recall	Multi-Class Precision	Multi-Class Recall	Multi-Class F_1 Score
BENIGN	246,196	98.82	97.12	99.65	98.37
BOTNET	110	3.64	88.60	91.82	90.18
DDOS	14,390	98.36	95.86	97.77	96.81
DOS-GOLDENEYE	1,147	82.04	87.05	66.78	75.58
DOS-HULK	24,021	98.98	98.61	97.53	98.07
DOS-SLOWHTTPTEST	767	77.44	88.43	80.70	84.39
DOS-SLOWLORIS	856	87.38	91.21	80.02	85.25
SSH-PATATOR	448	87.95	98.17	95.76	96.95
FTP-PATATOR	600	97.67	97.53	98.67	98.09
HEARTBLEED	2	100.00	0.00	0.00	0.00
INFILTRATION	4	0.00	0.00	0.00	0.00
PORTSCAN	24,019	74.52	97.34	73.50	83.75
SQLI	2	0.00	0.00	0.00	0.00
BRUTEFORCE (WEB)	17	82.35	63.64	82.35	71.79
XSS	95	0.00	0.00	0.00	0.00

TABLE 4.13: For the "New Flow Dataset" and Classifiers trained on the AE "codes", the Recall (TPR) of the Binary Classifier for each Class, along with the Precision, Recall and F_1 Score for the Multi-Class Classifier for each class.

The number of features of this dataset was reduced to 17.

Binary Classification

The first layer of the best model had 128 neurons and the second 96, the learning rate was set to 0.007, dropout to 0.4 and the batch size to 256.

This model struggles at detecting many kinds of attacks (Table 4.13) and has a high FPR of 1.1%.

Multi-Class Classification

The model had 3 layers of 128 neurons each, with the learning rate set to 0.0003, dropout to 0.4 and the batch size to 256.

As shown in Tables 4.13 and 4.14b it does not perform very well and certainly worse than its counterpart trained on the full dataset (compare against Tables 4.4 and 4.5b).

Binary Classifier Metrics	%
Accuracy	96.7180
Balanced Accuracy	93.8722
AUC	99.5920
Precision	95.3282
Recall	88.9211
F_1 Score	92.0133
Specificity	98.8233
FDR	4.6718
FPR	1.1767

(A) New Flow Dataset (AE codes): Binary Classifier Metrics

Multi-Class Classifier Metric		%
Accuracy	—	97.12
	Balanced	64.30
AUC	OVO	99.40
	OVR	99.78
Macro Avg.	Precision	66.90
	Recall	64.30
	F_1 Score	65.28
Weighted Avg.	Precision	97.09
	Recall	97.12
	F_1 Score	96.96

(B) New Flow Dataset (AE codes): Multi-class Classifier Metrics

4.6.3 Sequence Dataset

Class	Test Set # Obs.	Binary Recall	Multi-Class Precision	Multi-Class Recall	Multi-Class F_1 Score
BENIGN	327,797	99.96	99.99	99.96	99.97
BOTNET	125	95.20	99.20	99.20	99.20
DDOS	14,398	99.91	99.97	99.98	99.98
DOS-GOLDENEYE	1,168	99.83	98.46	98.63	98.55
DOS-HULK	23,905	99.93	99.87	99.84	99.85
DOS-SLOWHTTPTEST	645	99.69	99.06	98.45	98.76
DOS-SLOWLORIS	638	99.37	95.77	99.37	97.54
SSH-PATATOR	1,341	99.33	99.92	99.11	99.51
FTP-PATATOR	1,194	100.00	99.92	100.00	99.96
HEARTBLEED	369	99.73	99.73	100.00	99.86
INFILTRATION	6	0.00	0.00	0.00	0.00
PORTSCAN	3,252	99.02	96.25	99.42	97.81
SQLI	2	100.00	0.00	0.00	0.00
BRUTEFORCE (WEB)	144	99.31	100.00	97.92	98.95
XSS	137	100.00	99.28	100.00	99.64

TABLE 4.15: For the "Sequence Dataset" and Classifiers trained on the AE "codes", the Recall (TPR) of the Binary Classifier for each Class, along with the Precision, Recall and F_1 Score for the Multi-Class Classifier for each class.

As explained in Section 4.5.3, for the LSTM-Autoencoder model, a "code" was computed for each sequence and then repeated *timestep* times as input to the decoder. It is that code that the new classifiers were trained on, meaning instead of sequences, their inputs are single 16-dimensional vectors. In other words, these models are not LSTMs, but simple neural networks.

Binary Classifier	
Metrics	%
Accuracy	99.9403
Balanced Accuracy	99.8827
AUC	99.9970
Precision	99.7213
Recall	99.8056
F_1 Score	99.7634
Specificity	99.9597
FDR	0.2787
FPR	0.0403

(A) Sequence Dataset (AE codes): Binary Classifier Metrics

Multi-Class Classifier		
Metric		%
Accuracy	—	99.93
	Balanced	86.12
AUC	OVO	98.76
	OVR	99.29
Macro Avg.	Precision	85.83
	Recall	86.12
	F_1 Score	85.97
Weighted Avg.	Precision	99.93
	Recall	99.93
	F_1 Score	99.93

(B) Sequence Dataset (AE codes): Multi-class Classifier Metrics

Binary Classification

The best network had 256 neurons in the first layer and 64 in the second. The learning rate was set to 0.007, dropout to 0.35 and the batch size to 256.

The model performed generally well (AUC $>$ 99.99%, F_1 Score of 99.76%), only having significant problems with the "INFILTRATION" class and to a much lesser degree with the "BOTNET" class.

Multi-Class Classification

The model had 256 neurons in the first layer and 128 in the second. The learning rate was set to 0.0007, dropout to 0.35 and the batch size to 256. It had problems with the "INFILTRATION" and "SQLI" classes, but did not perform too badly (Table 4.16b).

4.7 Experiments: Comparisons

Metric (%)	Binary Classifiers			Multi-Class Classifiers			Autoencoders		
	LSTM	New NN	Orig NN	LSTM	New NN	Orig NN	LSTM	New NN	Orig NN
Balanced Acc.	99.966	99.937	99.190	89.72	80.86	65.67	96.74	87.10	87.63
AUC (OVO)	99.999	99.996	99.988	99.51	99.90	98.64	99.01	94.01	94.31
F_1 Score (Macro)	99.847	99.824	98.958	90.64	79.99	68.32	91.54	78.15	64.55

Metric (%)	Binary Classifiers (AE)			Multi-Class Classifiers (AE)		
	LSTM	New NN	Orig NN	LSTM	New NN	Orig NN
Balanced Acc.	99.883	93.872	98.427	86.12	64.30	69.29
AUC (OVO)	99.997	99.592	99.957	98.76	99.40	98.52
F_1 Score (Macro)	99.763	92.013	98.162	85.97	65.28	73.23

TABLE 4.17: Comparing Performance by neural network architecture[1]

As shown in Table 4.17, it is clear that the LSTMs consistently outperformed the Feed-Forward neural networks, even in the anomaly detection section where performances were generally poor.

Overall, binary classifiers performed the best, followed by the multi-class classifiers, with the autoencoders being a distant third. When the autoencoders were only used for dimensionality reduction, the classifiers trained on those datasets achieved far better results than the autoencoders did directly, however still not as good as the models trained directly on the datasets.

[1]New/Old NN indicate neural networks trained on the new and original flow datasets respectively, (AE) indicates the classifiers were trained on datasets after dimensionality reduction

Chapter 5

Conclusions & Future Work

5.1 Conclusions

In this Thesis, we trained and evaluated a variety of neural network models for network intrusion detection. First, we explored the available datasets and selected an appropriate flow-based one. From the packet captures accompanying it we extracted the IP, TCP and UDP headers from each packet and used them to create features for a sequence dataset. We also created an additional flow-based dataset by aggregating information from those features.

We then proceeded to analyze, clean and preprocess all the datasets and using them we trained many models and experimented with various model configurations e.g. different numbers of layers, learning rates, dropouts etc. Specifically, we first trained feed-forward neural networks and LSTMs for binary and multi-class classification. Then, we tried a different approach using autoencoders, initially for anomaly detection and then for dimensionality reduction. We also reported the configurations that gave the best results.

We discovered that using autoencoders directly for classification has poor results, but models trained on the reduced datasets they can produce perform much better. Finally, our experiments clearly showed that LSTMs perform better and do so consistently over all tasks, with the best LSTM model achieving an AUC over 99.999% and a FPR of 0.04%.

5.2 Future Work

One disadvantage of LSTM neural networks are the long training times, so it would be interesting to evaluate whether replacing them with temporal convolutional networks (TCNs) speeds up training times while achieving comparable results.

Another exciting avenue of research would be to try to improve performance by combining data coming from the network with additional data from the host and training models on that.

Bibliography

[1] Nicky Woolf. "DDoS attack that disrupted internet was largest of its kind in history, experts say". In: *The Guardian* (Oct. 26, 2016). URL: https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet.

[2] Lee Mathews. "Equifax Data Breach Impacts 143 Million Americans". In: *Forbes* (Nov. 7, 2017). URL: https://www.forbes.com/sites/leemathews/2017/09/07/equifax-data-breach-impacts-143-million-americans/.

[3] Mathew J. Schwartz. "Equifax's Data Breach Costs Hit $1.4 Billion". In: *BankInfoSecurity* (May 13, 2019). URL: https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473.

[4] Elizabeth Piper. "Cyber attack hits 200,000 in at least 150 countries: Europol". In: *Reuters* (May 14, 2017). URL: https://www.reuters.com/article/us-cyber-attack-europol/cyber-attack-hits-200000-in-at-least-150-countries-europol-idUSKCN18A0FX.

[5] Adam D'Angelo. "Quora Security Update". In: (Dec. 4, 2018). URL: https://blog.quora.com/Quora-Security-Update.

[6] Shaun Nichols. "Marriott: Good news. Hackers only took 383 million booking records ... and 5.3m unencrypted passport numbers". In: *The Register* (Jan. 4, 2019). URL: https://www.theregister.co.uk/2019/01/04/marriott_stolen_passport_numbers/.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[8] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI Blog* 1.8 (2019), p. 9.

[9] David Silver et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". In: *arXiv preprint arXiv:1712.01815* (2017).

[10] John Ellson et al. "Graphviz—open source graph drawing tools". In: *International Symposium on Graph Drawing*. Springer. 2001, pp. 483–484.

[11] Dan Foreman-Mackey et al. *daft-dev/daft: Minor bugfix*. Version v0.1.1. Apr. 2020. DOI: 10.5281/zenodo.3747801. URL: https://doi.org/10.5281/zenodo.3747801.

[12] Michael Waskom et al. *mwaskom/seaborn: v0.10.1 (April 2020)*. Version v0.10.1. Apr. 2020. DOI: 10.5281/zenodo.3767070. URL: https://doi.org/10.5281/zenodo.3767070.

[13] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[14] The pandas development team. *pandas-dev/pandas: Pandas 1.1.0*. Version v1.1.0. July 2020. DOI: 10.5281/zenodo.3964380. URL: https://doi.org/10.5281/zenodo.3964380.

[15] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56 –61. DOI: 10.25080/Majora-92bf1922-00a.

[16] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[17] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), p. 22.

[18] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[19] François Chollet et al. *Keras*. https://keras.io. 2015.

[20] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[21] Karen Scarfone and Peter Mell. *Guide to Intrusion Detection and Prevention Systems (IDPS)*. Tech. rep. NIST CSRC, 2012. URL: https://csrc.nist.gov/publications/detail/sp/800-94/rev-1/draft.

[22] Ronald A Fisher. "The use of multiple measurements in taxonomic problems". In: *Annals of eugenics* 7.2 (1936), pp. 179–188.

[23] Edgar Anderson. "The Species Problem in Iris". In: *Annals of the Missouri Botanical Garden* 23 (), pp. 457–509. ISSN: 0026-6493. DOI: 10.2307/2394164. URL: https://biostor.org/reference/11559.

[24] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. "Efficient algorithms for mining outliers from large data sets". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 2000, pp. 427–438.

[25] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[26] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[27] Tsung-Yi Lin et al. *Focal Loss for Dense Object Detection*. 2017. arXiv: 1708.02002 [cs.CV].

[28] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[29] Christopher Olah. "Understanding lstm networks". In: (2015).

[30] *INTERNET PROTOCOL*. RFC 791. Sept. 1981. URL: https://tools.ietf.org/rfc/rfc791.txt.

[31] *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474. Dec. 1998. URL: https://tools.ietf.org/rfc/rfc2474.txt.

[32] *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001. URL: https://tools.ietf.org/rfc/rfc3168.txt.

[33] *Updated Specification of the IPv4 ID Field*. RFC 6864. Feb. 2013. URL: https://tools.ietf.org/rfc/rfc6864.txt.

[34] *Transmission Control Protocol*. RFC 793. Sept. 1981. URL: https://tools.ietf.org/rfc/rfc793.txt.

[35] *Robust Explicit Congestion Notification (ECN) Signaling with Nonces*. RFC 3540. June 2003. URL: https://tools.ietf.org/rfc/rfc3540.txt.

[36] *User Datagram Protocol*. RFC 768. Aug. 1980. URL: https://tools.ietf.org/rfc/rfc768.txt.

[37] S Revathi and A Malathi. "A detailed analysis on NSL-KDD dataset using various machine learning techniques for intrusion detection". In: *International Journal of Engineering Research & Technology (IJERT)* 2.12 (2013), pp. 1848–1853.

[38] *NSL-KDD dataset*. URL: https://www.unb.ca/cic/datasets/nsl.html.

[39] Bhupendra Ingre and Anamika Yadav. "Performance analysis of NSL-KDD dataset using ANN". In: Jan. 2015, pp. 92–96. DOI: 10.1109/SPACES.2015.7058223.

[40] L. Dias et al. "Using artificial neural network in intrusion detection systems to computer networks". In: Sept. 2017, pp. 145–150. DOI: 10.1109/CEEC.2017.8101615.

[41] *KDD Cup 1999 dataset*. 1999. URL: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[42] Loic Bontemps et al. *Collective Anomaly Detection based on Long Short Term Memory Recurrent Neural Network*. 2017. arXiv: 1703.09752 [cs.LG].

[43] Ahmad Javaid et al. "A deep learning approach for network intrusion detection system". In: *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. 2016, pp. 21–26.

[44] Nathan Shone et al. "A deep learning approach to network intrusion detection". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 2.1 (2018), pp. 41–50.

[45] Abien Fred Agarap. *A Neural Network Architecture Combining Gated Recurrent Unit (GRU) and Support Vector Machine (SVM) for Intrusion Detection in Network Traffic Data*. 2017. arXiv: 1709.03082 [cs.NE].

[46] Jungsuk Song, Hiroki Takakura, and Yasuo Okabe. "Description of kyoto university benchmark data". In: *Available at link: http://www. takakura. com/Kyoto_data/BenchmarkData-Description-v5. pdf [Accessed on 15 March 2016]* (2006).

[47] Chuanlong Yin et al. "A deep learning approach for intrusion detection using recurrent neural networks". In: *Ieee Access* 5 (2017), pp. 21954–21961.

[48] Benjamin J. Radford et al. *Network Traffic Anomaly Detection Using Recurrent Neural Networks*. 2018. arXiv: 1803.10769 [cs.CY].

[49] Ali Shiravi et al. "Toward developing a systematic approach to generate benchmark datasets for intrusion detection". In: *computers & security* 31.3 (2012), pp. 357–374.

[50] Ahmed Ahmim et al. *A Novel Hierarchical Intrusion Detection System based on Decision Tree and Rules-based Models*. 2018. arXiv: 1812.09059 [cs.CR].

[51] *CICIDS2017 dataset*. URL: https://www.unb.ca/cic/datasets/ids-2017.html.

[52] Yong Zhang et al. "Network intrusion detection: Based on deep hierarchical network and original flow data". In: *IEEE Access* 7 (2019), pp. 37004–37016.

[53] Mohamed Amine Ferrag et al. "Deep Learning Techniques for Cyber Security Intrusion Detection: A Detailed Analysis". In: *6th International Symposium for ICS & SCADA Cyber Security Research 2019 6*. 2019, pp. 126–136.

[54] Razan Abdulhammed et al. "Features dimensionality reduction approaches for machine learning based network intrusion detection". In: *Electronics* 8.3 (2019), p. 322.

[55] *Libpcap File Format*. URL: https://wiki.wireshark.org/Development/LibpcapFileFormat.

[56] *PCAP Next Generation (pcapng) Capture File Format*. URL: https://github.com/pcapng/pcapng.

[57] Markus Ring et al. *A Survey of Network-based Intrusion Detection Data Sets*. 2019. arXiv: 1903.02460 [cs.CR].

[58] *1998 DARPA Intrusion Detection Evaluation Dataset*. 1998. URL: https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset.

[59] John McHugh. "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory". In: *ACM Transactions on Information and System Security (TISSEC)* 3.4 (2000), pp. 262–294.

[60] Mahbod Tavallaee et al. "A detailed analysis of the KDD CUP 99 data set". In: *2009 IEEE symposium on computational intelligence for security and defense applications*. IEEE. 2009, pp. 1–6.

[61] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization." In: *ICISSP*. 2018, pp. 108–116.

[62] *CICFlowMeter*. URL: https://github.com/ahlashkari/CICFlowMeter.

[63] Amirhossein Gharib et al. "An evaluation framework for intrusion detection dataset". In: *2016 International Conference on Information Science and Security (ICISS)*. IEEE. 2016, pp. 1–6.

[64] Iman Sharafaldin et al. "Towards a reliable intrusion detection benchmark dataset". In: *Software Networking* 2018.1 (2018), pp. 177–200.

[65] *The Heartbleed Bug, CVE-2014-0160*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[66] *The Heartbleed Bug*. URL: https://heartbleed.com/.

[67] *Internet Security Glossary, Version 2*. RFC 4949. Aug. 2007. URL: https://tools.ietf.org/rfc/rfc4949.txt.

[68] *Wireshark Network Protocol Analyzer*. URL: https://www.wireshark.org/docs/man-pages/tshark.html.

[69] *Tshark Network Protocol Analyzer*. URL: https://www.wireshark.org/docs/man-pages/wireshark.html.

[70] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].