Classification of encrypted traffic using deep learning

Andreas Rømer Hjorth
(s134835)

Salik Lennert Pedersen
(s134416)

Kongens Lyngby 2018

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Richard Petersens Plads
Building 324
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

As the amount of network traffic grows due to an increasing number of connected devices, expansion of network coverage and use of streaming services, there is a need for efficient bandwidth management. An important aspect of bandwidth management is classification of traffic as this allows for distribution and prioritization. At the same time the general consensus within network communication is to move towards more privacy and thus an increasing amount of network traffic is being encrypted. This challenges the common approaches used for classification of network traffic as they typically rely on the content being readable.

Napatech A/S deliver products used for analysis and storage of network traffic. They are faced with the challenge of discarding traffic generated by streaming services, which is a requested feature by their customers. The challenge is present as streaming services make use of the same protocols as other types of network traffic.

In this thesis we examine and evaluate different ways of classifying encrypted network traffic using neural networks. For this purpose we create a dataset with a streaming/non-streaming focus. The dataset comprises seven different classes, five streaming and two non-streaming. The thesis serves as a preliminary proof-of-concept for Napatech A/S. The most desirable approach for Napatech is to classify individual packets upon arrival. Our experiments show that this approach is infeasible due to the content being encrypted and therefore without any distinguishable information. We therefore propose a novel approach where the unencrypted parts of network traffic, namely the headers are utilized. This is done by concatenating the initial headers from a session thus forming a signature datapoint. In experiments using the header-based approach we achieve very promising results, showing that a simple neural network with a single hidden

layer of less than 50 units, can predict the individual classes with an accuracy of 96.4% and an AUC of 0.99 to 1.00 for the individual classes. The thesis hereby provides a solution to network traffic classification using the unencrypted headers.

Preface

This thesis was prepared at DTU Compute, department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment with the requirements of acquiring an M.Sc. in Computer Science and Engineering. The project has been completed between January 2nd and June 17th and is rated as 32.5 ECTS point for each author. The thesis has been conducted under the supervision of Professor Ole Winther, DTU Compute. Alex Omø Agerholm has been the supervisor for the thesis from Napatech A/S.

The thesis deals with the classification of network traffic using neural networks.

Lyngby, 17-June-2018

Andreas Rømer Hjorth
(s134835)

Salik Lennert Pedersen
(s134416)

Acknowledgements

We thank Napatech A/S for providing excellent office facilities, as well as providing a powerful ASUS ROG PC for training neural networks and generating network traffic. Thanks to Alex Omø Agerholm for valuable discussions throughout the process. Furthermore we would like to thank Allan Rosenstand Larsen for being so kind as to provide the Lenovo M93p PC that helped immensely in generating the datasets. Last but not least a very special thanks to Ole Winther for being supervisor on the project and helpful in providing insights and discussions along the way.

Contents

Summary (English) **i**

Preface **iii**

Acknowledgements **v**

1 Introduction **1**
1.1 Introduction . . . 1
1.2 Classical Approaches . . . 2
1.3 Napatech A/S . . . 3
1.4 Thesis Objective . . . 3
1.5 Related Work . . . 4
1.6 Source code . . . 5
1.7 Thesis Structure . . . 6

2 Machine learning **7**
2.1 Data exploration . . . 8
2.1.1 PCA . . . 8
2.1.2 t-SNE . . . 10
2.2 Neural Networks . . . 12
2.2.1 Neuron . . . 13
2.2.2 Network architecture . . . 14
2.2.3 Training a neural network . . . 14
2.2.4 Loss function . . . 16
2.2.5 Activation functions . . . 16
2.2.6 Regularization . . . 22
2.2.7 Evaluating a classifier . . . 24
2.2.8 Deep Learning . . . 28
2.2.9 Visualizing neural network decisions . . . 29

3 Network theory **33**
3.1 OSI . . . 33
3.2 Ethernet frame . . . 35
3.3 IP . . . 37
3.4 TCP . . . 38
3.5 UDP . . . 41
3.6 Encryption . . . 42
3.7 Session/Flow . . . 44

4 Dataset **45**
4.1 Existing datasets . . . 46
4.2 Classes . . . 46
4.3 Generating network traffic . . . 48
4.3.1 Selenium . . . 48
4.3.2 HTTP(S) getter . . . 49
4.3.3 Wireshark . . . 49
4.3.4 Pcap . . . 50
4.4 Post Processing . . . 51
4.5 How to streamline the process . . . 52

5 Experiments **55**
5.1 Payload experiments . . . 56
5.1.1 Data extraction . . . 56
5.1.2 Data exploration . . . 57
5.1.3 Exploratory Results . . . 61
5.1.4 Payload conclusion . . . 62
5.2 Header experiments . . . 63
5.2.1 Data extraction . . . 63
5.2.2 Data exploration . . . 64
5.2.3 Initial architecture . . . 68
5.2.4 Initial results . . . 68
5.2.5 Robustness . . . 71
5.2.6 Evaluation . . . 79
5.2.7 Visualizing decisions . . . 88

6 Discussion **93**
6.1 Dataset discussion . . . 94
6.1.1 Choice of classes . . . 94
6.1.2 Advertisements . . . 95
6.1.3 Choice of platform . . . 95
6.1.4 Header dataset . . . 96
6.1.5 Type of data . . . 96
6.1.6 Variety in each class . . . 97
6.2 Machine learning discussion . . . 97

6.2.1 Recall/precision tradeoff . . . 97
6.2.2 Temporal robustness . . . 98
6.2.3 Model size tradeoff . . . 98
6.2.4 Hyperparameter search . . . 99
6.2.5 Fixed input length . . . 99
6.2.6 Semi-supervised approach . . . 100

7 Conclusion **101**
7.1 Further Work . . . 103

A GitHub Repository **105**

B HTTP and HTTPS websites **107**
B.1 Linux/Chrome . . . 107
B.1.1 HTTP . . . 107
B.1.2 HTTPS . . . 108
B.2 Windows/Firefox . . . 108
B.2.1 HTTP . . . 108
B.2.2 HTTPS . . . 109

C Relevance plots **111**

D Header Plots **117**

E PC Specifications **121**
E.1 LinuxChrome . . . 121
E.2 WindowsFirefox . . . 122
E.3 WindowsChrome . . . 122
E.4 WindowsAndreas . . . 123
E.5 WindowsSalik . . . 123

F Header bytes **125**

G Cross-validation **129**

Bibliography **131**

Chapter 1

Introduction

> Take any old classification problem where you have a lot of data, and it's going to be solved by deep learning. There's going to be thousands of applications of deep learning.
>
> *Geoffrey Hinton*

1.1 Introduction

Technological advances has lead to an increase in overall network traffic. The many streaming services, expansion of network coverage and the major increase in the amount of devices connected [8], calls for optimal techniques for network bandwidth management. Even though bandwidth technology evolves in the attempt to serve the high demand, its current growth rate cannot compete with that of global internet traffic [25]. Network bandwidth management techniques can help ensure that the "most important" traffic on a network is prioritized. Some traffic may be prioritized higher if for example it is related to a service that has a quality of service (QoS) agreement e.g. some tasks can be more

important than others (gaming, VoIP[1] etc.). An important aspect of network bandwidth management is classification of the traffic on the network. Being able to determine which applications network packets belong to, will allow for a policy specification that can prioritize the right traffic.
A traffic classification task should be automated and be able to divide network traffic into several classes (application specific or predefined) based on characteristics available in the individual network packets or by a collection of such. The nature of this task thus seems very suitable to solve using a machine learning classification algorithm. We especially consider a neural network in order to solve the classification problem.

1.2 Classical Approaches

Different approaches have been utilized for the task of classifying network traffic. The most simple is port based classification, where the port number can inform about the protocol used for transmission. With the rise of Peer-to-peer (P2P) network traffic in which some protocols use ephemeral ports[2], that might fall in the range of other applications e.g. a P2P service might use port 80 which is normally associated with HTTP traffic. This might be done in order to obfuscate the traffic [19], hence port based classification has fallen out of favour. A limitation of only looking at the port number is that it does not say anything about the content, but rather give a suggestion on the protocol (HTTP, FTP, SSH etc.) used by the application.

Deep Packet Inspection (DPI) has shown to be superior to the port based approach when trying to classify P2P network traffic as shown by Sen et al. [43]. However with the increasing amount of encrypted network traffic [12] DPI has become less effective as any potential application signature has been encrypted. A way around this problem is to use a "middleman" approach such as the one SonicWall[3] offers. In this approach, a DPI certificate server is inserted between the communicating parties, such that the traffic is encrypted from client to certificate server and again from certificate server to the application server. This way the network traffic would appear as plain text in the DPI certificate server context. However this defeats the purpose of having encrypted traffic from a user perspective, and it is not always possible if correct end-to-end authentication is in place.

[1] Voice over IP is a group of technologies in which voice communication and multimedia is delivered by use of Internet Protocol networks.
[2] Short-lived transport protocol port used in Internet Protocol (IP) communication.
[3] https://www.sonicwall.com/.

1.3 Napatech A/S

Napatech is a global company founded in Denmark in 2003, they entered the Oslo Stock Exchange in 2013 [38]. Napatech have developed a suite of products all in the area of high speed network accelerators. They are designed for real-time network monitoring and analysis from 1Gbps to 200Gbps with zero packet loss. At the core of Napatech's product range is their SmartNICs, which is a range of network accelerators. They are designed to be used in ordinary servers, as such they are typically sold to OEMs[4]. Napatech's SmartNICs are used in a wide range of industries such as cyber security, telecom operators, financial services, cloud and data center, network management, infrastructure and defence. No matter the industry Napatech's SmartNICs provide insights into the performance and operation of a network. In order to ensure that zero packets are dropped Napatech's products are built around an FPGA[5] that can scale to handling the millions of packets flowing through the network. In order to ensure that applications can keep up with the hardware, buffering is added on top of the FPGA. FPGAs are programmable hardware that can be made highly specialized and therefore obtain a higher performance-pr-watt compared to a CPU that is very flexible in terms of doing different tasks and thus have a much lower performance-pr-watt. The story is the same for GPUs, they are not efficient enough when compared to FPGAs for this use case. Napatech have used their SmartNIC in one of their products called Pandion. Pandion is a network traffic recorder utilizing their core feature of capturing real time network traffic with zero packet loss and storing it all to disk. This requires a huge amount of storage. A recurring market request has been to make it possible to identify streaming traffic as this is of little to no interest to them. This is an increasingly larger problem as the amount of streaming traffic grows [4].

1.4 Thesis Objective

This thesis aims at exploring the possibilities of classifying network traffic by use of machine learning. Specifically we aim at distinguishing between streaming and non-streaming network traffic. As both streaming and non-streaming network traffic can be transmitted by a protocol such as HTTPS which use encryption, the solution has to predict the content type of packets and not only distinguish between protocols used for transmitting the packet. This thesis will explore the important aspects of the network traffic domain in order to create

[4] An Original Equipment Manufacturer (OEM) produces parts and equipment that might be marketed by another manufacturer.

[5] Field-Programmable Gate Array.

a dataset useful for this classification task, and explore a solution using neural networks.

1.5 Related Work

In this section we will describe some of the work done by others concerning classification of network traffic using machine learning. The majority of the papers work with network flows[6] or network sessions[7] or some features derived from those sessions/flows.

In a very recent paper, Michael et al. [34] use a very simple neural network consisting of a single hidden layer with 12 units and achieve a very good accuracy of 99% on a dataset containing 12 different classes. They create a 10-attribute feature vector based on statistics gathered from TCP sessions. One of the features they use is the port number which is a huge discriminator given the type of traffic they are dealing with. Furthermore they test the temporal stability of their model by training on a dataset generated in one day and testing on a dataset generated a year later on the same machine. They find that the model only perform slightly worse on the temporally displaced data as their accuracy drop from 99% to 96.7%. However when testing on data collected three years later than the training data they see a drop in performance to 89.6%.

Bar Yanai et al. [3] take a statistical approach in creating a hybrid method for classifying encrypted network traffic flows in real-time. They create a feature set containing 17 different features describing each recorded flow, such as *Client packet size variance* and *Transport protocol TCP/UDP*. The classification algorithm is a combination of a *k-nearest-neighbors* with $k = 1$ using Euclidean distance (2-norm) as the metric and the *k-means* algorithm. A hybrid approach is chosen as research done by Kim et al. [23] shows that *k-nearest-neighbors* has the steepest increase in run-time relative to the size of the training-data but achieve a high accuracy. *k-means* on the other hand runs fast but achieves a rather poor accuracy in comparison. During the training phase Bar Yanai et al. use *k-means* to form multiple clusters for each class (HTTP, Skype, etc.) and then afterwards associate each training sample with the closest centre. In order to classify new observations, Bar Yanai et al. find the closest centre of any cluster and all the training samples associated with that cluster, and then run *1-nearest-neighbor* in order to assign the same label as the *1-nearest-neighbor*

[6] A network flow is a uni-directional packet stream uniquely defined by a five-tuple: (source IP, source port, destination IP, destination port, protocol type).

[7] A network session is a bi-directional packet stream and can contain both directions of flows meaning that the source and destination port are interchangeable.

training sample.

In two recent papers by Wang et al. [50],[49] convolutional neural networks are used to classify malware and end-to-end encrypted traffic respectively. In both cases it is done by looking at the first 784 bytes of either a session or a flow of network traffic. In [50] 2D convolutions are applied using an architecture similar to LeNet-5 [30] as the 784 bytes input are treated as images by interpreting it as a *28-by-28* pixel grey scale image by translating the raw bytes as pixel intensities such that 0x00 is black and 0xFF is white. Using a simple convolutional network they are able to classify 20 different classes using two different setups. In the first setup they use two models, a binary classifier to classify malware/non-malware followed by two 10 class classifiers. In the second setup they have a single 20 class classifier. The three models are all sharing the same architecture except the number of classes (2, 10, 20 respectively) in the output layer. They receive an impressive average accuracy of $> 99\%$ across all three models.

In [49] Wang et al. use a somewhat similar architecture albeit with 1D convolutions instead of 2D convolutions. Using this modified neural network they create an end-to-end model which extract features from the raw bytes of the VPN-non-VPN ISCX dataset published by Draper-Gil et al. [6]. The dataset in which 6 classes are VPN versions of network traffic (Email, Chat, Streaming, File-Transfer, P2P, VoIP) and 6 classes is non-VPN versions of the same type of network traffic. Wang et al. conduct 4 different experiments where they attempt to classify the traffic into 2 (VPN/non-VPN), 6 (non-VPN), 6 (VPN) and 12 (All) classes. In the first experiment Wang et al. are able to identify which traffic is VPN and which is non-VPN with very high precision and recall $> 99.9\%$. In the second experiment, trying to classify the individual classes of the non-VPN traffic, Wang et al. achieve a lower precision (85.5%) and recall (85.8%). The third experiment, trying to classify the individual classes of the VPN traffic, Wang et al. achieve a decent precision (94.9%) and recall (97.3%). In the fourth and final experiment, trying to classify all classes, a similar pattern as in experiment two and three is seen. Namely that it is easier to classify traffic that is encrypted using protocol encryption (VPN) than traffic that is encrypted on the application layer (non-VPN). However they do note that there is some class imbalance in the dataset which might also be an issue as in general the underrepresented classes perform worse than classes with more data.

1.6 Source code

The relevant code created throughout this project can be found on GitHub (appx. A) where it has been made open source under the MIT licence.

1.7 Thesis Structure

This section outlines the structure of the thesis and provides a brief introduction to the content of each chapter. The structure is as follows:

- **Chapter 2: Machine learning** outlines the theoretical foundation from which we will try to construct a solution to the posed task.
- **Chapter 3: Network theory** outlines some of the theory needed to gain a better understanding of the network traffic domain.
- **Chapter 4: Dataset** describes how the dataset came to be, as well as describes some of the tools and practises behind the generation of the dataset.
- **Chapter 5: Experiments** describes the different experiments made in the process of classifying network traffic, as well as interpreting neural network decisions.
- **Chapter 6: Discussion** reflects on some of the findings and decisions made along the way.
- **Chapter 7: Conclusion** outlines the key findings as well as provide an overall conclusion to the project and suggests some interesting directions of further work.

Chapter 2

Machine learning

> [M]achines of this character can behave in a very complicated manner when the number of units is large.
>
> *Alan Turing*

In the previous chapter we introduced the problem and setting of classifying network traffic with the focus of separating streaming from non-streaming traffic.
The purpose of this chapter is to present some of the theoretical and mathematical foundations on which machine learning is based on.

In the first part of the chapter, methods of dimensionality reduction and visualization is presented. The second part of the chapter deals with the theory behind neural networks and presents a technique for visualizing what part of an input that is most relevant for a specific prediction made by the neural network.

2.1 Data exploration

When facing a machine learning task, one of the first steps towards a positive result is to obtain knowledge about the data. This can include many things, such as summary statistics of the different attributes. These statistics describe things like the minimum, maximum, mean and standard deviation of each individual feature, e.g. illustrated with box plots. Another important step can be to explore different plots of the dataset. Often the dataset in question is of very high dimensionality, which can be very hard to visualize and comprehend for humans. Tools such as PCA and t-SNE can be used to perform dimensionality reduction such that the dataset is easier to visualize and examine.

2.1.1 PCA

A **P**rincipal **C**omponent **A**nalysis is a mathematical tool invented by Pearson [41] in 1901 but was independently developed and named by Hotelling [18] in 1933. The idea behind a principal component analysis is to take a lot of correlated variables and reduce them to a smaller set of uncorrelated variables called principal components. As the principal components are uncorrelated it means that they are orthogonal to one another. The direction of the first principal component is the direction that accounts for as much variance in the data as possible, each succeeding principal component accounts for as much of the remaining variance in the data as possible under the constraint that it is orthogonal to the previous principal components. As the principal components account for an increasing amount of the variance in the data, they can be used for dimensionality reduction. It might be that a dataset of high dimensionality have all of its variance in only two directions. It is then possible to describe all datapoints using a linear combination of the first two principal components. It is however important to note that the dimensions that the principal components represent might not be interpretable. In order to find the principal components we calculate the eigenvectors and eigenvalues of the covariance matrix. Let $\mathbf{A}$ be a matrix of size $n \times m$ and $n \geq m$ for which m is the number of dimensions or features and n is the number of rows/datapoints. By subtracting the mean from each column we obtain the matrix $\mathbf{B}$:

$$\mathbf{B} = [b_{i,j}] = [a_{i,j} - \frac{1}{n}\sum_{i=0}^{n} a_{i,j}]$$

Then we calculate the covariance matrix $\mathbf{C}$.

$$\mathbf{C} = \frac{1}{n}\mathbf{B}^{\mathsf{T}}\mathbf{B}$$

Once the covariance matrix is found the eigenvectors and eigenvalues can be calculated. However a more convenient approach to perform PCA is to use singular value decomposition.

Singular Value Decomposition (SVD) is a way to decompose any real matrix $\mathbf{B}$ of size $n \times m$ with $n \geq m$ into three matrices

$$\mathbf{U} = \begin{bmatrix} | & | & | & | \\ \mathbf{u_1} & \mathbf{u_2} & \dots & \mathbf{u_m} \\ | & | & | & | \end{bmatrix}, \mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & \mathbf{0} & \\ & & \ddots & \\ & & & \sigma_m \\ & \mathbf{0} & & \end{bmatrix}, \mathbf{V} = \begin{bmatrix} | & | & | & | \\ \mathbf{v_1} & \mathbf{v_2} & \dots & \mathbf{v_m} \\ | & | & | & | \end{bmatrix}$$

such that:

$$\mathbf{B} = \mathbf{U\Sigma V^\mathsf{T}}$$

Where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal matrices such that:

$$\mathbf{U^\mathsf{T}U} = \mathbf{I}$$
$$\mathbf{V^\mathsf{T}V} = \mathbf{I}$$

and $\mathbf{\Sigma}$ have the property that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_M$. The fact that the columns of $\mathbf{V}$ and $\mathbf{U}$ are orthonormal gives some interesting properties such as $\mathbf{v_i^\mathsf{T} v_j} = 0$ if $i \neq j$ and otherwise 1. By using this property it is possible to compute:

$$\mathbf{V^\mathsf{T} v_i} = \begin{bmatrix} \mathbf{v_1^\mathsf{T}} \\ \vdots \\ \mathbf{v_i^\mathsf{T}} \\ \vdots \\ \mathbf{v_m^\mathsf{T}} \end{bmatrix} \mathbf{v_i} = \begin{bmatrix} \mathbf{v_1^\mathsf{T} v_i} \\ \vdots \\ \mathbf{v_i^\mathsf{T} v_i} \\ \vdots \\ \mathbf{v_m^\mathsf{T} v_i} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{e_i}$$

By using this it is possible to show that the eigenvectors and eigenvalues of the covariance matrix $\mathbf{C} = \frac{1}{n}\mathbf{B^\mathsf{T}B}$ can be found using SVD:

$$\mathbf{B^\mathsf{T}B} = \left(\mathbf{V\Sigma^\mathsf{T}U^\mathsf{T}U\Sigma V^\mathsf{T}}\right) = \mathbf{V\Sigma^2 V^\mathsf{T}} \tag{2.1}$$
$$(\mathbf{V\Sigma^2 V^\mathsf{T}})\mathbf{v_i} = \mathbf{V\Sigma^2 e_i} = \mathbf{V}\sigma_i^2\mathbf{e_i} = \sigma_i^2\mathbf{v_i} \tag{2.2}$$

The process is the same for $\mathbf{BB^\mathsf{T}}$:

$$\left(\mathbf{BB^\mathsf{T}}\right) = \mathbf{U\Sigma^2 U^\mathsf{T}} \tag{2.3}$$
$$\left(\mathbf{BB^\mathsf{T}}\right)\mathbf{u_i} = \mathbf{U\Sigma^2 U^\mathsf{T} u_i} = \sigma_i^2\mathbf{u_i} \tag{2.4}$$

From these equations we get that $\mathbf{v_i}$ is an eigenvector of $\mathbf{B}^\mathsf{T}\mathbf{B}$ with eigenvalue σ_i^2 and u_i is an eigenvector of $\mathbf{B}\mathbf{B}^\mathsf{T}$ also with eigenvalue σ_i^2 as we remember the eigenvalue equation:

$$\mathbf{B}\mathbf{x} = \lambda\mathbf{x}$$

What this equation tells us is that if a matrix $\mathbf{B}$ multiplied by a vector $\mathbf{x}$ gives the same vector $\mathbf{x}$ but scaled by a factor λ, then λ is an eigenvalue of $\mathbf{B}$ and $\mathbf{x}$ is an eigenvector. Calculating the SVD of a matrix $\mathbf{B}$ thus consist in finding the eigenvectors and eigenvalues of $\mathbf{B}\mathbf{B}^\mathsf{T}$ and $\mathbf{B}^\mathsf{T}\mathbf{B}$ as the columns of $\mathbf{U}$ contains the eigenvectors of $\mathbf{B}\mathbf{B}^\mathsf{T}$ where the columns of $\mathbf{V}$ are made up by the eigenvectors of of $\mathbf{B}^\mathsf{T}\mathbf{B}$. The matrix $\mathbf{\Sigma}$ is a diagonal matrix which contain the singular values or the square root of the eigenvalues from $\mathbf{B}\mathbf{B}^\mathsf{T}$ and $\mathbf{B}^\mathsf{T}\mathbf{B}$.

PCA algorithm In order to perform PCA on a matrix $\mathbf{A}$ of size $n \times m$ first we subtract the mean of each column to obtain a matrix $\mathbf{B}$. Optionally divide by the standard deviation if the columns are very differently scaled and one column therefore have much greater variance. The first principal components will be driven towards that column if the data is not normalized. Then compute the SVD $\mathbf{B} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathsf{T}$. The n first principal components are then given by $\mathbf{v_1}, \ldots, \mathbf{v_n}$.

2.1.2 t-SNE

t-distributed **S**tochastic **N**eighbour **E**mbedding (t-SNE) is a visualization technique that works well on datasets of high dimensionality. t-SNE can be used as a nonlinear dimensionality reduction tool, but is particularly well suited for visualization of high dimensionality data-sets. t-SNE was invented by Maaten and Hinton [32] in 2008 and builds upon SNE which was invented by Hinton and Roweis [17] in 2002. t-SNE works in three stages:

1. Compute the matrix of pairwise similarities $p_{j|i}$ on the original high-dimensionality dataset. $p_{j|i}$ is the conditional probability that a datapoint x_i would choose datapoint x_j as its neighbour given a Gaussian distribution centred at x_i. Mathematically this is expressed in :

$$p_{j|i} = \frac{e^{-d_{ij}^2}}{\sum_{k \neq i} e^{-d_{ik}^2}}$$

where

$$d_{ij}^2 = \frac{\|x_i - x_j\|^2}{2\sigma_i^2}$$

is the dissimilarities between x_i and x_j given by the scaled squared Euclidean distance. σ_i is the variance of the Gaussian which is centred at datapoint x_i and is found such that the entropy of the distribution over neighbours is equal to $log(k)$ where k is the perplexity which can be interpreted as some measure of the number of neighbours. The joint probabilities p_{ij} is given by:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$

where n is the number of datapoints. Symmetric SNE is used such that $p_{ij} = p_{ji}\ \forall i, j$.

2. An initial low-dimensional solution $\mathcal{Y}^{(0)} = \{y_1, y_2, \ldots, y_n\}$ is sampled from a normal distribution $\mathcal{N}(0, 10^{-4}I)$.

3. The low-dimensional joint similarities q_{ij} is computed by a Student-t distribution with one degree of freedom (Cauchy distribution):

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|y_k - y_l\|^2)^{-1}}$$

The cost function C which should be minimized is the Kullback-Leibler divergence between the joint probability distribution P, in the high-dimensional space and the joint probability distribution Q, in the low-dimensional space given by:

$$C = KL(P\|Q) = \sum_i \sum_j p_{ij} log \frac{p_{ij}}{q_{ij}}$$

The cost function is minimized using a gradient decent method where the gradient of the cost function C is given by:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1}$$

Which is then minimized over T iterations:

$$\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta \frac{\delta C}{\delta y_i} + \alpha(t)(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)})$$

where η is the learning rate and $\alpha(t)$ is the momentum which can be seen as a running average of the gradient.

In essence what t-SNE is trying to do is to make datapoints that are close together in a high dimensional representation, also be close together in a low dimensional representation. This is done by iteratively moving close points closer while repelling points that are farther away in the high dimensional representation.

2.2 Neural Networks

Artificial neural networks (ANN) are computing systems vaguely inspired by the architecture of the biological neural network found within animal brains. Much is still unknown about the brains capabilities to process information, though it is known that an essential part is its interconnected neurons that output spikes of electrical activity due to electrical activity outputted by other neurons. The performance of the human brain in complex tasks like facial, speech and image recognition is admirable and drives the interest of exploring ANN for accomplishing similar learning tasks using computers. Like a biological neural network, the artificial neural network is constructed from interconnected nodes similar to that of a biological neuron. The nodes are arranged in layers by directed links, and a network is usually constructed having an input layer, one or more hidden layers[8] and an output layer. Neural networks can be divided into two different classes, feedforward networks and feedback (recurrent) networks.
A feedforward neural network is acyclic, meaning that the information flows only in a forward direction, from the input layer towards the output layer. A recurrent neural network has at least one connection which is a cycle, meaning that a least one output of a neuron is used as feedback input for other neurons.

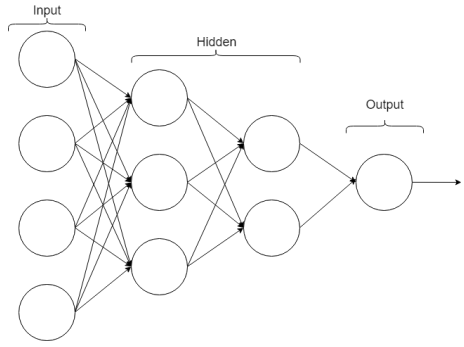


Figure 2.1: A feedforward neural network with two hidden layers.

From a mathematical perspective a neural network model can be viewed as a function that maps an input x to an output y, $f : \mathbf{x} \rightarrow \mathbf{y}$. Given a task in which some function $f^{'}$ maps from $\mathbf{x} \rightarrow \mathbf{y}$, a learning algorithm can be applied to a neural network such that it models a function f which approximates $f^{'}$ and can solve the task.

[8] Though a network can also have zero hidden layers.

2.2.1 Neuron

The artificial neuron is the processing unit of a neural network. A neuron consists of inputs, weights, a bias and an activation function that are all used to calculate an output. Each input is multiplied by the weight that expresses the importance of that input to the output of the neuron. A simple neuron is the one known from a Perceptron network [40]. A Perceptron is a single layer feedforward network with a neuron that takes several binary inputs $x_1, x_2, ...$, and outputs a single binary value.

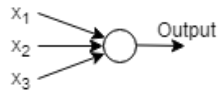


Figure 2.2: A Perceptron network.

The output value of the Perceptron is calculated by multiplying the weights with the inputs and then summing the term $\sum_k w_k \cdot x_k$. If the summed value exceeds some predetermined threshold, the neuron outputs a binary value of 1 and 0 otherwise.

$$\text{output} = \begin{cases} 1, & \text{if } \sum_k w_k \cdot x_k > \text{threshold} \\ 0, & \text{otherwise} \end{cases}$$

This neuron is thereby able to put different weights or importance on the binary input values, and if the sum of those weighted inputs is sufficiently large, the neuron will *fire* and output a value of 1. This is analogous to a simple description of the neurons within the human brain that fires when receiving a sufficient amount of input. To include our mention of bias in the notation for a Perceptron neuron, we can rewrite the evaluation term by moving the threshold to the other side of the inequality and calling it the bias, $b \equiv -threshold$. We also introduce the dot product operator in the expression for simplification.

$$\text{output} = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0, & \text{otherwise} \end{cases}$$

As the Perceptron is linear in its parameters and variables, the decision boundary is linear as well. Using such a model as a classifier will only allow it to reach an optimal solution if the classes are linearly separable.
Therefore modern neural networks most often make use of neurons with a non-linear activation function, such as the sigmoid, tanh, or the ReLU[9]. A single

[9] The rectified linear unit function is a piecewise linear function.

neuron of such will however not create a nonlinear decision boundary, but chaining those neurons in a multilayered neural network will allow the network to approximate any nonlinear function as proved by Cybenko in [5].

2.2.2 Network architecture

When designing an ANN the architecture has a large influence on which functions the network can model. A single layer feedforward neural network is not able to find the optimal solutions for problems with a nonlinear input/output relation. Adding a hidden layer between the input and output layer of an ANN will according to the universal approximation theorem [5] allow the neural network to approximate any continuous function to the desired precision by adding neurons with a nonlinear activation function to that hidden layer. Another approach to constructing an artificial neural network is known as deep learning. Deep learning models make use of more than one hidden layer in the network architecture. These type of networks can be difficult to interpret as the input is transformed numerous times before reaching the output layer of the network, however we can potentially approximate a more complex model with fewer units than a similarly performing network with fewer layers [33].

2.2.3 Training a neural network

As mentioned, the goal of a neural network is to approximate a function $f^{'}$ that maps from some input $\mathbf{x}$ to an output $\mathbf{y}$, $\mathbf{y} = f^{'}(\mathbf{x})$. A feedforward neural network defines a function $f(\mathbf{x}, \theta) = \hat{\mathbf{y}}$ for which θ is the learnable parameters of the network i.e. weights and biases.
When training a neural network we adjust the parameters θ such that $f(\mathbf{x}, \theta) \approx f^{'}$. As the true distribution $P(\mathbf{y}|\mathbf{x})$ is unknown to a machine learning task we perform what is known as empirical risk minimization, in which we try to reduce the expected generalization error by minimizing the error for a sample of the true distribution. To adapt the parameters of a neural network, such that the function $f(\mathbf{x}, \theta)$ approximates $f^{'}$ to an acceptable level of precision, a measure of fit is used. Most commonly used is the measure of likelihood and the principle of maximum likelihood estimation (MLE) that maximizes the probability of seeing the sample data. Equation 2.5 shows an expression of MLE in which the parameters $\hat{\theta}$ are chosen from the set Θ of possible parameters.

$$\hat{\theta} \in \{\arg\max_{\theta \in \Theta} \mathcal{L}(\theta; \mathbf{x})\} \tag{2.5}$$

As maximizing something corresponds to minimizing the negative of the same, we can treat the problem as a minimization problem in which we minimize the

negative log likelihood. This measure can also be described as the cross-entropy between the training data and the model distribution [16].

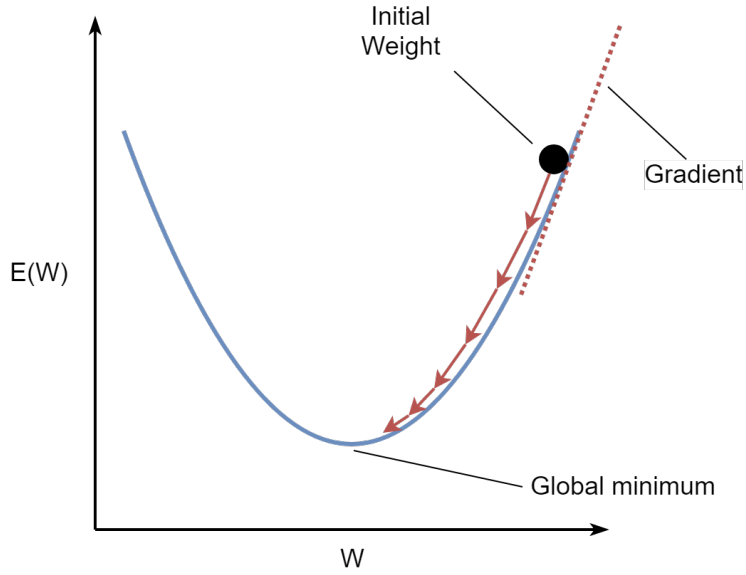


Figure 2.3: Intuition behind gradient decent.

When training a feedforward neural network the information forward propagate to the output layer where a loss is computed. The backpropagation algorithm as proposed by Rumelhart et al. [42] allows the information computed by a loss function to back propagate into the network to be able to compute the gradient of the cost function with respect to the parameters θ.
The core of the backpropagation algorithm is the chain rule, which is a rule for computing the derivative of a composition of two or more functions. Having $y = g(x)$ and $z = f(g(x)) = f(y)$, the chain rule states that $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$. This is useful as every layer of a neural network can be seen as a separate function.
Having calculated the gradients by use of the backpropagation algorithm a successful approach known as gradient-based learning methods is commonly applied. The aim of the learning algorithm is to minimize the loss function and thereby increase the likelihood of seeing our data.
The gradient-descent algorithm utilizes partial derivatives to adjust the parameters of the network such that the value of the loss function will decrease. Equation 2.6 demonstrates how the weights of a neural network might be updated when using gradient descent, t is the step, n is know as a learning rate.

$$W(t) = W(t-1) - n\frac{dE}{dW} \tag{2.6}$$

The intuition behind gradient-decent is shown in figure 2.3 where the weights

are updated such that the error $E(W)$ is minimized. Figure 2.3 is a very simple example where the global minimum can be reached via gradient-decent. This is not always the case when working with neural networks.

2.2.4 Loss function

A loss function, also known as a cost function or objective function is a measure of how good the predictions made by the neural network are compared to the ground truth. When training a neural network as a supervised learning task, the loss is often calculated as an average over the loss of each prediction.

$$L = \frac{1}{N} \sum_{i} L_i \tag{2.7}$$

When treating the learning task as a maximum likelihood problem in which we try to find a set of parameters that maximize the probability of seeing our data, the output of the network can be compared to the ground truth of our data to obtain a measure of similarity. The cross-entropy loss function measures the dissimilarity between two probability distributions, minimizing cross-entropy therefore corresponds to maximizing the probability of seeing the data. Equation 2.8 shows the notation of cross-entropy between the two probability distributions p and q.

$$H(p, q) = -\sum_{\mathbf{x}} p(\mathbf{x}) log(q(\mathbf{x})) \tag{2.8}$$

As the loss function is a function of the learnable parameters within the network it will possibly be a high-dimensional function in which a global minima is the aim of the gradient-descent based learning approach.
Working with the loss function of a multilayered network typically includes a loss function having a surface that is non-quadratic, non-convex and of high dimensionality with many local minima and saddle points [31]. This has resulted in the development of techniques such as batch learning and stochastic gradient descent that helps ensuring a better and faster convergence.

2.2.5 Activation functions

Activation functions can basically be divided into two different types:

1. Linear activation functions

2. Non-linear activation functions

The perceptron is a linear classifier and thus is not capable of convergence if the problem is not linearly separable. If the activation function is linear it does not matter how many hidden layers we stack, the final output is still a linear combination of the original input. This introduces a need for non-linearities/activation functions in neural networks as this allows for much greater descriptive power and flexibility.

Logistic sigmoid is a widely used activation function in classification problems as it resembles a step function in that the range of the output is bounded between 0 and 1, but it is a smooth, monotonic, yet non-linear function. This is great for representations of probabilities and classification tasks.

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

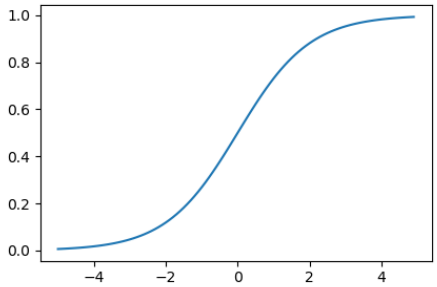


Figure 2.4: The sigmoid function.

As can be seen in figure 2.4, for input values between -2 and 2 the slope of the logistic function is very steep. This is desirable in a classification task as a small change in input value yields a large change in output and thereby a clear distinction between classes.

An attractive property of the sigmoid function is that its derivative can reuse the calculated sigmoid value:

$$\sigma'(x) = (1 - \sigma(x)) \cdot \sigma(x) = \frac{e^x}{(1 + e^x)^2}$$

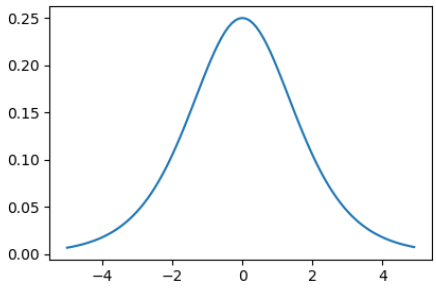


Figure 2.5: The derivative of the sigmoid function.

As can be seen in figure 2.5, one of the problems with using a sigmoid activation function is that for large values of x, both positive and negative, the corresponding gradient is going to be very small hence leading to a "vanishing gradients" problem. The "vanishing gradients" problem occurs when the neural network refuses to learn or is drastically slow in doing so. This problem becomes even more profound when adding more hidden layers as the derivative of the sigmoid function have a maximum value of 0.25. By using the chain rule of calculus to backpropagate through the layers we end up multiplying a lot of numbers that are less than 1, effectively making the gradient even smaller. In order to partially combat this phenomenon it is important to initialize the weights of the network properly, such that they are not to large (positive or negative). If the weights are large the gradient will be close to zero and the neurons will become saturated. This leads to neurons that have trouble learning.

Tangent hyperbolic (Tanh) is an activation function that looks much like the sigmoid function:

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

In fact tanh is just a scaled sigmoid:

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

As such tanh is still monotonic and differentiable just like the regular sigmoid function. Figure 2.6 shows that the function is still sigmoidal but with an output range from -1 to 1 and $tanh(0) = 0$.

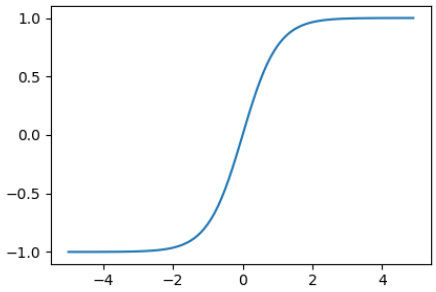


Figure 2.6: The tanh activation function.

The tanh function also has a nice derivative where caching can be used:

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

As can be seen in figure 2.7 the tanh function still suffers from the vanishing gradients problem albeit not to the same extend as the regular sigmoid function, because the maximum value of the derivative is 1 instead of 0.25. Furthermore tanh only maps values around 0 to near zero outputs. Large negative values are mapped to ≈ -1 where the regular sigmoid maps those values to ≈ 0. As tanh is symmetric around the origin it is preferred, over the regular sigmoid, as the output are likely to have an average around zero, where the output of the regular sigmoid is always positive and thus must have positive mean. By having a mean around zero, using tanh can usually lead to faster convergence as LeCun et al. [31] mentions. As with the regular sigmoid function, weight initialization is very important to avoid saturated neurons.

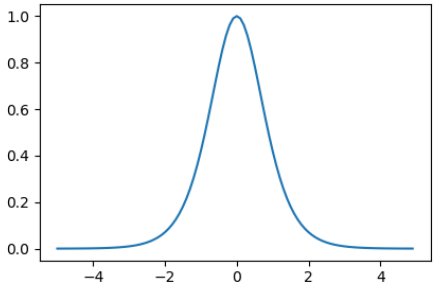


Figure 2.7: The derivative of the tanh function.

Rectified Linear Unit (ReLU) is an activation function that has fallen in favour recently, as state of the art within neural networks, i.e. deep learning, have shown that more layers help in learning a given task. ReLU does not have the same problems as the sigmoidal functions. ReLU is a monotonic function with an output range of 0 to $+\infty$

$$g(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

As can be seen in figure 2.8 the output of the ReLU function is 0 for negative input and linear (identity) for positive input.

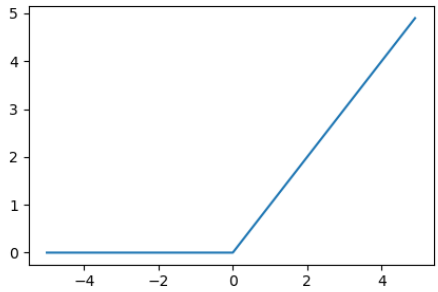


Figure 2.8: The ReLU activation function.

The derivative of the ReLU function is:

$$g'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{UNDEF} & \text{if } x = 0 \end{cases}$$

As can be seen in figure 2.9 the gradient is either 0 or 1 thus inhibiting the vanishing gradients problem that can be present when using sigmoidal function in deep neural networks. One problem that ReLU does have is dead neurons. Dead neurons happen when a gradient update adjusts all the weights of the neuron such that the input to the ReLU function is < 0 effectively making the gradient 0 as well, thus inhibiting updates and therefore is unlikely to recover. This can to some extend be avoided by using a learning rate that is not too large.

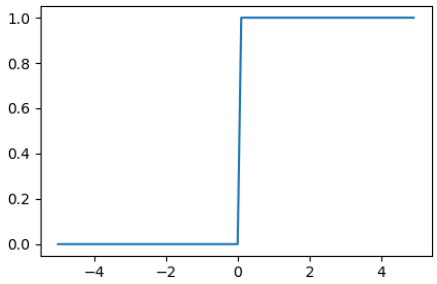


Figure 2.9: The derivative of ReLU activation function.

There are however variants of the ReLU function that try to mitigate the dying neuron problem by having a small slope for negative input instead of outputting zero. One such function is Leaky ReLU with the equation:

$$g(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Other variants exist (ELU, SELU, etc.) but in general it is just different variations on what range to map negative input to. In general regular ReLU will perform just fine but one should know that alternatives exist.

Softmax or normalized exponential function is a function that is often used as the activation function in the output layer of a neural network performing a multiclass classification task. Softmax can take a K-dimensional vector $\mathbf{x}$ of real values and transform them into a K-dimensional vector $s(\mathbf{x})$:

$$s : \mathbb{R}^K \to \{s \in \mathbb{R}^K | s_i > 0, \sum_{i=1}^{K} s_i = 1\}$$

This is very useful when performing classification tasks, as softmax typically will "highlight" the largest value and suppress the smaller values. However the softmax function is not scale invariant and as such will not perform the same for values that are in the range 0 to 1 as for values that are in the range 0 to 10. In some cases softmax might de-emphasize the maximum value and make it smaller, it will however still be the largest in the output vector.

$$s(\mathbf{x})_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

Due to the nature of the softmax function to put emphasis on a single value, the function is not well suited for classification problems where multiple labels

are present. In such cases it will be better to use another non-linearity such as a sigmoidal function and then interpret on the individual output values.

2.2.6 Regularization

A central problem of machine learning algorithms is that even though they easily can perform well on the training data, they often suffer from the problem of overfitting and therefore generalize badly, i.e. performing badly on new input. Many strategies are developed to make the algorithms generalize better, potentially at the cost of an increased training error. *"Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error"* [16]. An effective regularizer is one that reduces variance while not overly increasing the bias and thereby increase the models predictive performance.

One approach to prevent overfitting can be to reduce the size of the network, as a larger network is capable of modelling more complex functions to higher precision, such a network is also highly prone to overfitting. Reducing the network size can however restrict the network too much, thus preventing it from learning the task.

2.2.6.1 Dataset Augmentation

The ideal way of making a machine learning model generalize better is to train it on more data. The amount of data available is however often a great challenge in machine learning tasks. To overcome this, one can simulate data and add it to the training dataset. For a supervised classification task the model expects a dataset of (x,y)-pairs, allowing creation of new data by simple transformations of the input vector x. However, this must be done with some caution since the transformation must not results in a change of the correct class y. An example would be the well-known MNIST dataset [29], in which transforming the digit 6 could be done by rotating the features. A rotation of 180 degrees would make the correct class a 9 instead of a 6 which therefore acts as an incorrect labelled datapoint. In audio recognition tasks sounds can be overlayed to create new data. A method that is generally applicable is to add random noise to the input of the machine learning algorithm which is also considered as a data augmentation technique [16].

2.2.6.2 L^2 regularization

The L^2 regularization technique is also known as weight decay and is one of the most common forms of regularization. It is implemented by adding the term $\frac{1}{2} \cdot \lambda \cdot W^2$ to the loss function [22], having λ as regularization strength. Using this type of regularization results in a change to the weight updates such that peaky weight vectors are heavily penalized when updated. When performing a weight update by use of gradient descent, the weights are affected by the term $-\lambda \cdot W$ when using the L^2 regularization.

2.2.6.3 Early stopping

When training a model which is complex enough to overfit the training data, a strategy called early stopping can be applied. Early stopping is a commonly used regularization technique due to its simplicity and effectiveness. When monitoring the training error and the validation error during training of a model, one might experience that the training error continuously decreases while the validation error starts increasing again. This is a typical pattern seen when the model is overfitting to the training data. Normally, the model and its parameters are returned after a predetermined number of epochs, if the validation error is not at it lowest at this epoch, we must expect that the model would generalize better if we stopped training at the point where the validation error was at its lowest. This is exactly what early stopping accomplishes. Early stopping is implemented such that the training algorithm terminates when the validation error is not improving by a certain margin, over a pre-specified number of epochs. Thereby a model with an expected better generalization error is returned from the training algorithm.

2.2.6.4 Dropout

Dropout is an efficient technique that approximates that of combining exponentially many different neural network architectures [45]. The technique is called dropout as it is achieved by eliminating the output of a number of the non-output[10] units within the network. Eliminating the output of a unit can be done by multiplying the output value with zero which happens with a fixed probability p. Co-adaptions happen within the units of a neural network when the weights are updated. The update might change the unit such that it makes up for a "mistake" of another unit. These co-adaptions do not generalize to

[10] Being the units in the input layer and the hidden units.

unseen data and therefore leads to overfitting [45]. By using dropout, the presence of any hidden unit within the network is unreliable which prevents these co-adaptions from forming.

2.2.7 Evaluating a classifier

Evaluating the performance of a neural network used in a supervised learning task is highly dependent on the nature of the task. In general we calculate different metrics that each can contribute with information about how good or bad the neural network is performing.
As training of the neural network progresses, both loss and accuracy is often calculated on a validation set. The loss is calculated as described previously while the accuracy is the amount of true predictions divided by the total number of predictions. As the value of the loss function drops, the amount of true predictions should increase and thereby increase the accuracy of the neural network.
Accuracy can however not be used as the only performance measure due to the nature of some classification tasks. Solving a binary classification task with an unbalanced dataset where 70% of the observations belongs to class 1, a naive neural network would achieve 70% accuracy by always predicting 1, even though this would clearly not be a good model. A technique for dealing with this problem is to balance the classes such that no class is under- nor over-represented. This would make a naive classifier yield an accuracy of approximately $\frac{1}{C}$, where C is the number of classes.

2.2.7.1 Confusion matrix

A confusion matrix is a way to visualize the performance of a classifier. A confusion matrix is a $c \times c$ matrix, where c is the number of classes. Figure 2.10 shows a confusion matrix for a binary classification problem, which in the case of multiclass problems can be extended by adding columns and rows. Each column of the confusion matrix displays the instances of a predicted class while each row shows the actual class instance.

		Prediction: Positive	Prediction: Negative	Total
Ground truth	Positive	TP	FN	$TP + FN$
	Negative	FP	TN	$FP + TN$
	Total	$TP + FP$	$FN + TN$	N

Figure 2.10: An example of a confusion matrix.

The confusion matrix in figure 2.10 include the notion of *true positives* (TP), *true negatives* (TN), *false positives* (FP) and *false negatives* (FN) that are used when calculating metrics such as precision and recall.

2.2.7.2 Performance metrics

Precision or **P**ositive **P**redictive **V**alue is calculated by equation 2.9 and is the number of times that the classifier correctly predicted a class instance divided by the total number of times that the classifier predicted that class. This measure can be calculated over all classes or per class to obtain knowledge about the performance on individual classes.

$$PPV = \frac{TP}{TP + FP} \tag{2.9}$$

Recall or **T**rue **P**ositive **R**ate is calculated by equation 2.10 and is the number of times that the classifier correctly predicted a class instance divided by total number of instances within the dataset. As with precision, this metric is often calculated per class to obtain knowledge about the performance of the classifier on the individual classes.

$$TPR = \frac{TP}{TP + FN} \tag{2.10}$$

The approach of evaluating a model's performance is highly dependent on the aim of the task. A classic example of this is when trying to classify whether a bank transaction is fraud (positive class) or valid (negative class) i.e. a binary classification task. For such a model, it might be of very high importance that no valid transactions are marked as fraud as this might reject the transfer. When evaluating this model, precision is of higher importance than recall,

as this measures the fraction of fraudulent transactions (TP) out of the total number of positive predictions (TP + FP). However if the classifier is only used to flag the transaction, and the employees of the bank will manually decide if a flagged transaction is fraudulent or valid, it might be of much higher importance to have a high recall, since more fraudulent transactions are caught for further investigation, potentially including some false positives that would lower the precision of the classifier. The relationship between precision and recall is typically inversely related. Increasing the recall would typically make a classifier predict more false positive e.g. a classifier can always obtain a recall of one by only guessing the positive class in a binary classification task. An increased number of false positives decreases the measure of precision. In order to increase the precision of a classifier, the number of false positives should be lowered. This would most often result in having more false negatives, thus lowering the recall.

F-score is a measure that takes both precision and recall into account. A commonly used version of this is the F1-score which is an harmonic mean of precision and recall. Equation 2.11 shows the general formula of the F-score for which $\beta = 1$ when calculating the F1-score. Increasing the value of β such that $\beta > 1$ puts more weight on recall than precision and vice versa.

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}} \tag{2.11}$$

2.2.7.3 Receiver operating characteristics

A receiver operating characteristic (ROC) curve is created by plotting the true positive rate (TPR) as calculated in equation 2.10, against the false positive rate (FPR) as calculated in equation 2.12, for a binary classifier at different threshold settings. The effect of changing the threshold of a binary classifier is illustrated in figure 2.11

$$FPR = \frac{FP}{FP + TN} \tag{2.12}$$

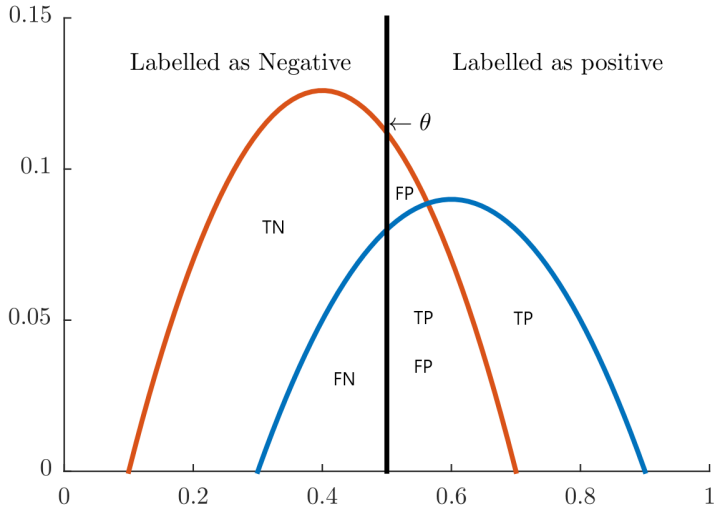


Figure 2.11: Figure displaying how a change of the threshold θ varies the amount of true positives, true negatives, false positives and false negatives. Datapoints belonging to the positive class are contained within the blue curve, while datapoints of the negative class are contained within the red curve. No classifier would be able to perfectly distinguish between the positive and negative class, as the two curves are overlapping. The figure is adapted from [48].

Altering the threshold of the classifier changes the proportion of TP, TN, FP and FN and thereby also the TPR and FPR. To create the ROC curve we calculate TPR and FPR at different threshold values and plot them against each other. An ROC plot has three critical points that are easily interpretable.

- TPR $= 0$, FPR $= 0$: All instances are predicted as the negative class.
- TPR $= 1$, FPR $= 1$: All instances are predicted as the positive class.
- TPR $= 1$, FPR $= 0$: Ideal model that perfectly separates the two classes.

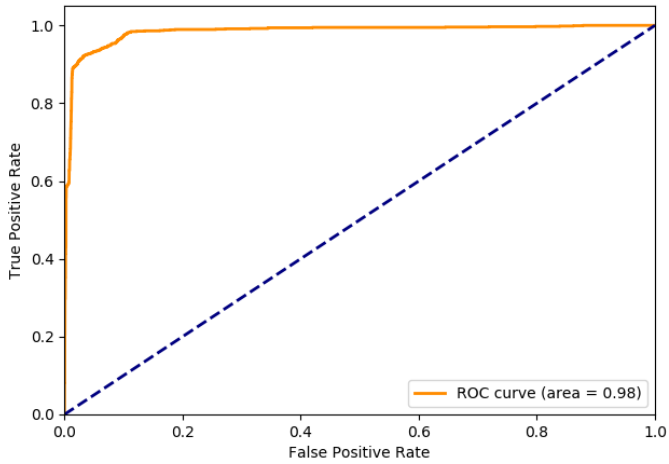


Figure 2.12: ROC curve with AUC = 0.98. The blue line illustrates the ROC curve for a random classifier.

A great advantage of using an ROC curve is that it deals with class imbalance. This is clear from the equations of TPR 2.10 and FPR 2.12 that are normalized by having the number of positive and negative datapoints respectively in the denominator. The **A**rea **U**nder **C**urve (AUC) can be calculated for the ROC curve and used as a performance metric when comparing classifiers. The AUC of the ROC curve qualifies as a performance evaluator since an ROC curve going through the point where TPR = 1 and FPR = 0 will yield a value of 1, while the AUC of a classifier with a curve going diagonally from TPR = 0, FPR = 0 to the point TPR = 1, FPR = 1 corresponds to a random classifier with an AUC of 0.5.

2.2.8 Deep Learning

Neural networks as a mathematical concept has existed for a long time e.g. the perceptron was invented in 1957 [11]. Backpropagation was invented in 1986 by Rumelhart et al. [42], but it is only in the last 10 to 20 years that neural networks have taken off. This is partly due to the fact that more data has become available, thereby making the models easier to train. A reason for the increased interest in deep learning is due to learning algorithms that have been written for GPUs rather than CPUs. GPUs work wery well for parallel computation, just think about the amount of pixels that have to be updated each second on a regular full HD monitor running at 60Hz, that is

$1920 \times 1080 \times 60Hz = 124416000$ pixels/s. Since neural network computations mostly consist of matrix/vector multiplication that are highly parallelizable, GPUs have a huge advantage over CPUs in that domain. Research by Lawrence et al. [28] have shown that, at least in their case, a speedup of more than 50x can be expected when using a single GPU over a dual core CPU. Furthermore by transitioning to more backpropagation friendly activation functions, such as ReLU instead of sigmoidal functions that tend to saturate, it is possible to create deeper neural networks than before. New techniques for regularization such as dropout [45] or batch normalization [21] have helped in training larger models without overfitting. New variants of stochastic gradient decent with momentum such as Adam [24] have also helped in reducing the amount of time required to train a deep network as they tend to converge faster. Finally software platforms like TensorFlow [1] have made the technologies more approachable and allow for more focus on the actual model rather than specific details of CPU/GPU management.
When talking about deep neural networks we talk about neural networks with more than one hidden layer. Recent advances in deep learning have shown that deeper networks tend to generalize better than shallow networks. Even though the universal approximation property holds for both shallow and deep networks, deep networks can do so using a much lower number of trainable parameters [33]. Deep convolutional neural networks have shown to be very effective in image recognition tasks, as the first layers extract high-level features from an image, such as edges, where the deeper layers work on finding more abstract features. Even though deep neural networks tend to perform better it can be quite hard to understand what is going on inside the "black box". One should take the predictions as an approximation that can not be fully trusted, as research have shown that neural networks are easily fooled [39]. However the fact that deeper networks tend to perform better in some tasks, such as image recognition, remain. This is also due to the fact that the era of big data is upon us. Deep neural networks need more training data than shallow networks as the first layers of a very deep neural network can be immensely slow to learn. As we get more data we also diminish the probability of overfitting, as a network with a lot of trainable parameters might tend to do so given a small number of training samples.
A general rule of thumb is to keep neural networks as small as possible, but just large enough so that they perform well on test data.

2.2.9 Visualizing neural network decisions

Neural networks often come with a high degree of predictive power, but as the models grow more complex it can be very hard for a human to interpret how a prediction was made, and what led the neural network to predict one class

over another. One way to visualize the important features of the input is called sensitivity analysis as proposed by Zurada et al. [52]. Another recently developed approach to visualize the important features, is called *Layer-wise Relevance Propagation* (LRP) and was proposed by Bach et al. [2]. LRP is a method that can identify the important input features by running a backward pass through the network. When applied to a neural network with ReLU activation functions, a special case of LRP can be seen as Deep Taylor decomposition as presented by Montavon et al. [35]. One should know the difference between sensitivity and decomposition techniques. More specific sensitivity analysis tries to identify what makes the input x more or less x, e.g. what makes the cat more or less a cat. Whereas the decomposition technique tries to reveal the characteristics of an input x, e.g. how the network characterizes a cat as a cat.

Sensitivity analysis is one approach to identify the important input features to a neural network. A sensitivity analysis typically evaluates the gradient of the neural network output function given some input. It attempts to measure the local effect and commonly defines a relevance score evaluated at some datapoint x as:

$$R_i(x) = \left(\frac{\partial f}{\partial x_i} \right)^2$$

Such that the highest score comes from the input that makes the gradient of the output function the largest i.e. which input feature the output is most sensitive towards. The sensitivity analysis as such satisfies the equation:

$$\sum_{i=1}^{d} R_i(x) = \|\nabla f(x)\|^2$$

Sensitivity analysis therefore does not explain the function $f(x)$ but rather its local slope. One nice thing about a sensitivity analysis is that it is rather easy to implement for neural networks as the gradients can be computed by backpropagation.

Relevance analysis through LRP is a more recent method for visualizing and explaining the predictions of neural networks. Heatmaps made using LRP are typically easier to interpret than those made using gradient based methods, such as a sensitivity analysis. LRP works by running a backwards pass through the neural network as shown in figure 2.13. LRP should satisfy the conservation property such that each neuron receives a share of network output and then accordingly redistributes it to the previous layers until the input is reached. More formally the sum of assigned relevance in the input space should correspond to the total relevance detected by the model:

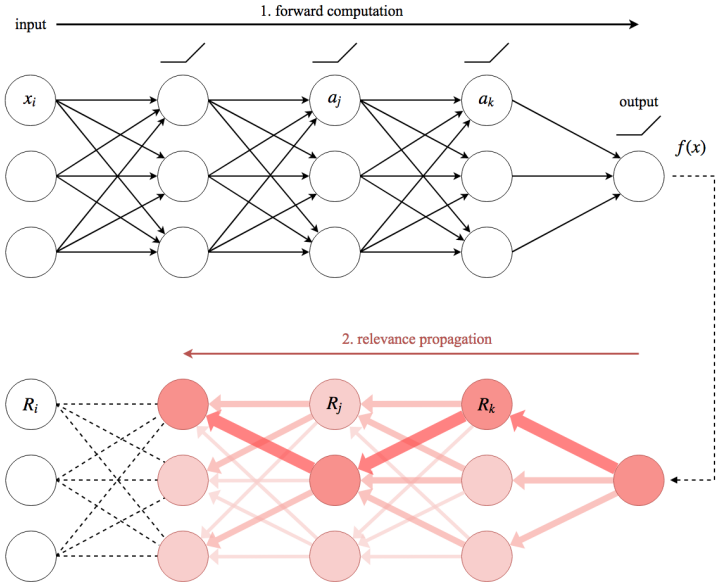


Figure 2.13: The LRP procedure of a deep neural network with ReLU activation functions. The red arrows show the relevance propagation flow. Inspired by [36].

$$\forall x : \sum_i R_i(x) = f(x)$$

In order to ensure this conservation property LRP must be implemented using a specific set of propagation rules. Let the neurons of the neural network be described by the following equation:

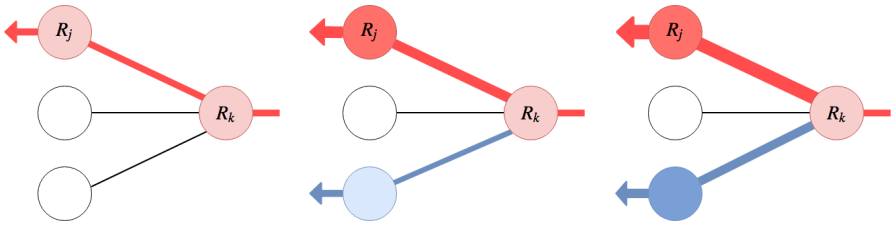
$$a_k = \sigma\left(\sum_j a_j w_{jk} + b_k\right)$$

Where a_k is the activation of the neuron, a_j is the activation from the previous layer and w_{jk} the weights and b_k the bias term. The activation function σ is positive and strictly increasing. One propagation rule shown by Bach et al. [2] to work well in practice is the $\alpha\beta$-rule given by:

$$R_j = \sum_k \left(\alpha \frac{a_j w_{jk}^+}{\sum_j a_j w_{jk}^+} - \beta \frac{a_j w_{jk}^-}{\sum_j a_j w_{jk}^-}\right) R_k$$

Where a term in the sum corresponds to the relevance passed back through the network from neuron k to neuron j. The w_{jk}^+ and w_{jk}^- denote the positive and

negative parts of the weights. It should be noted that the parameters α and β are chosen under the constraints $\alpha - \beta = 1$ and $\beta \geq 0$. As shown in figure 2.14 higher values of α results in more weight on the positive relevance being propagated back through the neural network. Higher values of β results in more negative relevance being propagated accordingly.



(a) $\alpha = 1, \beta = 0$ **(b)** $\alpha = 2, \beta = 1$ **(c)** $\alpha = 3, \beta = 2$

Figure 2.14: Figure showing the relevance distribution for different values of parameters α and β. Positive relevance is shown in red and negative shown in blue. Inspired by [36].

It has been shown by Montavon et al. [35] that choosing parameters $\alpha = 1, \beta = 0$ results in a more stable LRP for very deep networks such as GoogLeNet. By using those parameters the propagation rule only allows for positive relevance and hence reduces to:

$$R_j = \sum_k \frac{a_j w_{jk}^+}{\sum_j a_j w_{jk}^+} R_k$$

This special case of LRP where $\alpha = 1, \beta = 0$ when used on deep ReLU networks is equivalent to Deep Taylor decomposition as shown by Montavon et al. [35].

Chapter 3

Network theory

> In theory there is no difference
> between theory and practice,
> while in practice there is.
>
> *Benjamin Brewster*

In the previous chapter we introduced some of the different methods and theory behind various parts of machine learning.
In this chapter important aspects of working with network traffic will be presented.
The chapter is structured such that first an abstract conceptual model of network traffic is presented, this is followed by theory about the different layers of the model that relates to our problem at hand.

3.1 OSI

The **O**pen **S**ystems **I**nterconnection (OSI) model is useful when dealing with network traffic. It models network communication in abstract layers from 1 to 7.

L1 **Physical Layer**, the actual bits are transferred via a medium, e.g. RS-232, 100BASE-TX, 802.11b or any other method of transportation.

L2 **Data Link layer** is responsible for node-to-node data transfer, e.g. communication between two computers, a computer communicating with a switch/router or any other two nodes. The data link layer utilize the unique hardware address that is the MAC[11] address to deliver *frames* between nodes on the same LAN. Data link frames, as the PDU[12] is called, only operates on the same LAN and therefore any inter-network communication is done on higher layers.

L3 **Network Layer** is responsible for providing the functions and procedures that allow for inter-network delivery of a data sequence called a datagram or packet between two nodes. This is done by using an address such as the IPv4 address for the source and destination nodes. This is also the layer responsible for routing the datagrams between different LANs.

L4 **Transport Layer** is responsible for providing the functions and procedures that enable variable-length data sequences to be transferred between a source and a destination host. Using segmentation/desegmentation, flow control and error control the transport layer is responsible for the reliability of the link between two hosts, such that retransmission of missed segments is possible should the delivery fail. The Transport Control Protocol (TCP) and the User Datagram Protocol (UDP) of the IP suite are usually classified as transport layer protocols within the OSI model even though they do not adhere strictly to the model. TCP and UDP use port numbers to deliver the data to the correct socket of a given host.

L5 **Session Layer** is responsible for for the connections between two hosts. It manages the connection between a local and remote application. The session layer in the OSI model provides the possibility to close a session properly. TCP also has this property, which blurs the line somewhat between the transport layer and session layer, at least for TCP.

L6 **Presentation Layer** is responsible for establishing a context between application-layer entities, by translating the data from the network formats to something the application layer can accept. The presentation layer is able to serialize complex data structures into flat byte-strings and vice versa.

L7 **Application Layer** is the layer that is the closest to the end-user. The user can interact with this layer. On this layer we find protocols such as HTTP, SMTP and FTP.

[11] Media Access Control.
[12] Protocol Data Unit.

The first four layers can be seen as encapsulating the rest of the layers, e.g. in the TCP/IP suite the data link layer (Ethernet) adds MAC addresses to a frame where the payload is a network layer IP packet. The network layer (IP) adds IP addresses and other header fields to a payload that is a transport layer TCP segment. The transport layer (TCP) adds port numbers and other header fields to the payload that is data from the presentation layer as shown in figure 3.1.

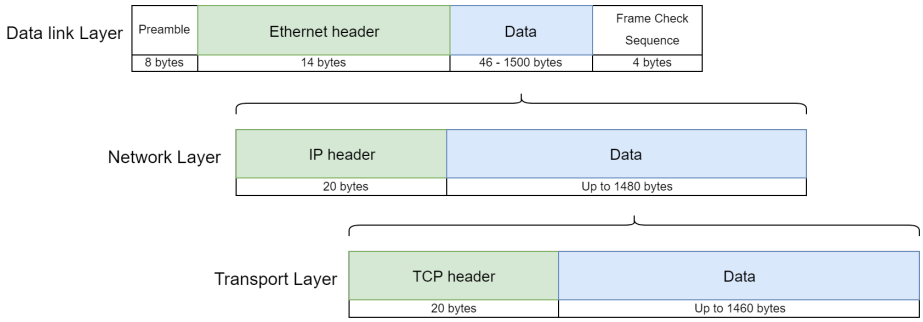


Figure 3.1: Ethernet example.

3.2 Ethernet frame

An Ethernet frame is the payload of a data packet sent via an Ethernet link. An Ethernet frame starts with the Ethernet header as shown in figure 3.2 which consists of the destination and source MAC addresses which are two unique 48-bit (6 bytes) hardware addresses assigned to the communicating network interfaces. These fields are followed by a type field which is 2 bytes long. The type field can be used to indicate the protocol encapsulated in the payload of the Ethernet frame. For instance IPv4 use 0x0800 as the type field.

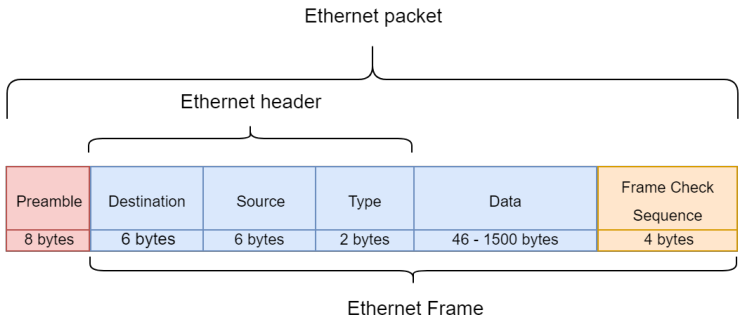


Figure 3.2: Ethernet frame.

The maximum transmission unit (MTU) of an Ethernet frame is typically 1500[13] bytes which gives a total maximum frame length of 1514 Bytes or 1518 if the frame check sequence (FCS) is included, however it is typically omitted. The FCS is a 32 bit cyclic redundancy check (CRC) that allows the receiver to detect corrupted data within the entire Ethernet frame. As Ethernet is a non deterministic protocol, meaning that any device can start sending at any point in time, there is a minimum frame length of 64 Bytes (including FCS) of the payload in order to ensure that the entire medium is occupied such that no collisions occur. For instance using a 100Mbit/s Ethernet card sending only one bit would only occupy ≈ 3 meters of medium. The way to calculate this is

$$100 \text{ Mbit/s} = 10^8 \text{ bit/s}$$

$$\frac{3 \times 10^8 \text{ m/s } (\approx \text{ speed of light})}{10^8 \text{ bit/s}} \approx 3 \text{ m/bit}$$

Therefore by using a minimum payload of 46 Bytes a minimum frame length of 64 Bytes is obtained, this means that each frame takes up:

$$64 \text{ Bytes} \times 8 \text{ bits/Byte} \times 3 \text{ m/bit} = 1536 \text{ m}$$

As the Ethernet specification says a maximum cable length of ≈ 100 m using CAT5e/CAT6 Ethernet cable this leaves a decent overhead. It should however be noted that this limitation is a legacy standard when using half-duplex Ethernet via copper cables. Most modern Ethernet networks run full-duplex meaning that separate wires are used for receiving and transmitting thus no collisions occur, and the cable length limitation is imposed to ensure little attenuation of the signal. It is possible to have a 10 km fibre optic link without problems as the physical properties and architecture allows for it e.g collisions do not happen as two different optical fibres are used for up- and down-link (full-duplex) and the attenuation of an optical fibre is much lower than a copper wire.

[13] Jumbo frames with a MTU of more than 1500 bytes exist but are typically not used.

3.3 IP

The Internet Protocol (IP) is the principal network layer protocol that all internet traffic use. It is at this layer the routers of the internet look in order to determine on what link to forward a packet. It is the task of the IP protocol to route packets from a source host to a destination host.

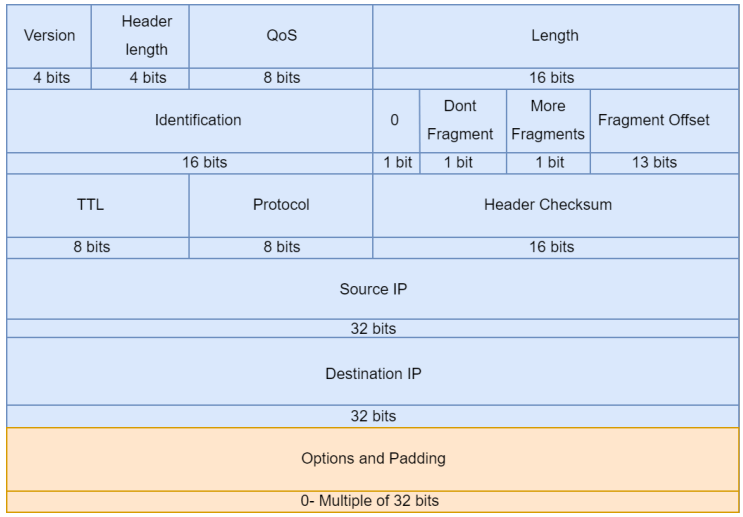


Figure 3.3: IPv4 Header.

IP Header The header of an IPv4 packet is at least 20 bytes long and can be up to 60 bytes long if additional options or padding are added to the header[14]. This IPv4 limit is imposed by the header length field being 4 bits long which gives a total of fifteen 32-bit words. When we look at the first 20 bytes of the IPv4 header, depicted in blue in figure 3.3, we notice a few important fields. First is the *version* field telling which version of the protocol the packet belongs to. The current standard is version 4 but version 6 is also in use. IP version 4 use 32-bit IP-address-fields as shown in figure 3.3. This allows for up to $2^{32} \approx 4.29$ billion different addresses. IP version 6 will increase the number of possible IP addresses by using 128-bits for each address. This gives $2^{128} \approx 3.4 \times 10^{38}$ different IPv6 addresses or enough to assign $\approx 2^{49}$ different addresses to each observable star in the known universe [10]. Another important field of the IP header is the *length* field which reveals the size of the packet and imposes a limit on the maximum length of an IP packet including the header itself, of 65535

[14] IPv6 headers are at least 40 bytes and can be much larger, since the standard allows for extension headers.

bytes (64KB). However, it might be that the data link layer has an even smaller MTU such as Ethernet which typically has an MTU of 1500 bytes. If the IP packet is larger than the MTU of the data link layer it will be fragmented. This is where the fragment fields of the IP header come into play. When an IP packet is fragmented, all fragments are treated as individual packets and they are not reassembled before they reach their destination. The *protocol* field tells whether the transport layer protocol is TCP or UDP indicated by a 6 or a 17 (decimal) respectively. The *Time-To-Live* field is decremented each time an IP packet is forwarded by a router. When it reaches zero the packet is discarded in order to avoid infinite loops. Lastly the *header checksum* field is used to check the integrity of the IP header to protect against corruption.

3.4 TCP

Transmission Control Protocol or TCP is one of the most used transport layer protocols today as it provides a reliable byte stream abstraction even over unreliable networks. TCP is widely used as a transport layer protocol for internet traffic today, such as email, file transfers or browsing the World Wide Web. TCP implements features to mitigate the potential problems that might occur over unreliable networks:

- Corrupted packets: TCP checksum is used to discard bad packets.
- Lost packets: TCP implements retransmission if a packet is lost.
- Delayed packets: TCP use adaptive timers to avoid retransmitting unnecessarily.
- Unordered packets: TCP use sequence numbers to restore the correct order.
- Sender tries to send faster than receiver can handle: TCP receiver signals to sender how much buffer space is available (windows size) such that the sender does not send more than the receiver can store.

TCP Header The TCP header is at least 20 bytes in size. The header can include up to 40 bytes of optional header data as shown in figure 3.4.

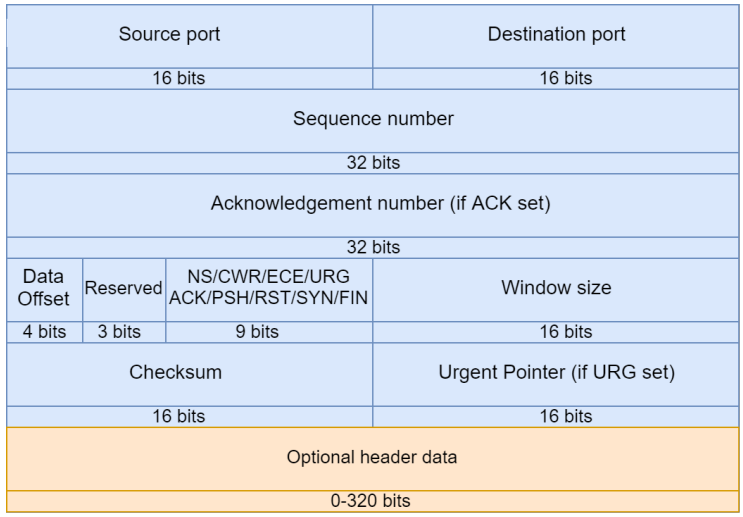


Figure 3.4: TCP Header.

From figure 3.4 we see the layout of the TCP header where some of the most important fields are the source and destination ports, as they are used to uniquely identify the flow that the traffic is part of. Other fields of interest are the checksum field and the window size. The checksum field is used to check for corruption of the TCP segment. The window size indicates the maximum amount of traffic the sender can send before having to wait for an acknowledgement (ACK).

Checksum The TCP checksum is used to check for corruption of the packet. The "pseudo" header of 12 bytes consisting of the source IP address, destination IP address, some reserved bits, the protocol field from the IP header and the TCP segment length is used in combination with the TCP header and data itself to compute the checksum. As the checksum field is part of the TCP header, it is assumed to be all zeroes when calculating the checksum as shown in figure 3.5. In order to calculate the TCP checksum a ones' compliment[15] is used on words of 16 bits each e.g.

$$\begin{aligned} & 1001101001010110 \\ + \; & 0000101110001110 \\ + \; & 0000110111001100 \\ = \; & 1011001110110000 \end{aligned}$$

[15] ones' compliment of a binary number is obtained by inverting all the bits (0 becomes 1 and vice versa).

ones' compliment of the result 1011001110110000 is 0100110001001111 which is the checksum. By using ones' compliment it becomes very easy to check for errors in the data as the receiver does the same calculations but with the senders checksum included. By adding the data together in 16 bit word chunks the output is a sum of 16 ones, if no errors were introduced during transit e.g.

$$
\begin{aligned}
& 1001101001010110 \\
+\ & 0000101110001110 \\
+\ & 0000110111001100 \\
+\ & 0100110001001111 \text{ (Checksum)} \\
=\ & 1111111111111111
\end{aligned}
$$

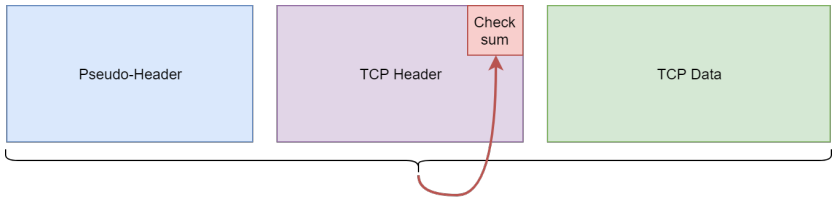


Figure 3.5: Calculating TCP checksum over pseudo header and TCP segment.

Handshaking In order to set up or close a TCP session between two hosts a three-way-handshake is initiated. Should a host (A) wish to connect to another host (B), A will send a message with the SYN bit set and a sequence number. This indicates that host A wish to synchronize sequence numbers. Host B replies with a message where the SYN and the ACK bits are set and the ACK number is set to the received sequence number plus one, indicating that host B is ready for the next message in the sequence. Host B inserts its own sequence number in the message for host A to receive. Host A receives the message and sends a message with the ACK bit set and with ACK number equal to host B sequence number plus one. Hosts A and B have now setup a TCP session between them. This process can be seen in figure 3.6

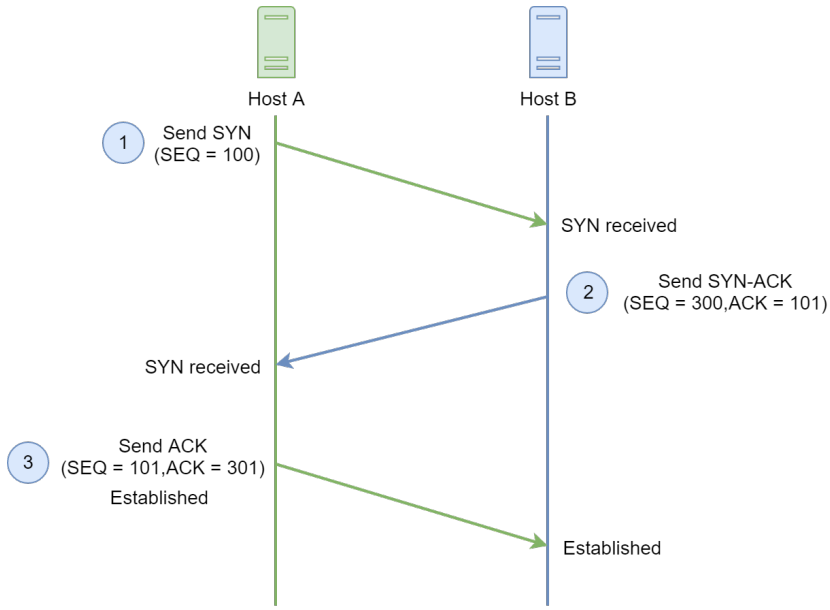


Figure 3.6: TCP Three-Way-Handshake.

3.5 UDP

User Datagram Protocol (UDP) is a core member of the IP suite protocols. UDP is a transport layer protocol that can be used instead of TCP. UDP does however not provide the same reliability features as TCP, e.g. no handshaking and no ACKs. This means that UDP does not provide a delivery guarantee, no ordering or duplicate protection. It does however provide port numbers and the same type of error/data integrity checking that is the ones' compliment checksum as TCP does. UDP is often preferred in time-sensitive applications as it might be better to drop a single packet rather than having to wait for a retransmission. Due to UDP not providing the same services as TCP it allows for much less bandwidth overhead and latency. UDP is therefore often used in network applications where perceived latency is critical such as audio and video communication. In order to make a network application using UDP reliable, the application-layer has to manage some of the same services as TCP, e.g. retransmission and reordering.

3.6 Encryption

From a basic point of view encryption is the process of encoding information such that only a holder of a decryption key can decode the information to its original form. Encryption is used widely across the internet as it is an insecure line, i.e. it does not preserve confidentiality and integrity. In general one can distinguish between two types of encryption used on the internet, application layer encryption and transport layer encryption. Both types ensure that the payload of the transport layer is encrypted and only the sender and receiver can know of the content.

Application Layer Encryption is used when encryption and decryption happens as close to the user as possible. By using application layer encryption the data is only visible in the application's memory space. When the information is sent to another application, that information is encrypted and the actual content is hidden. This is useful when an application does not want any other layers in the OSI model to be able to modify nor read the information. An example of application layer encryption could be PGP[16], a public key encryption program which is one of the most popular programs for encrypting emails.

Transport Layer Security (TLS) is the successor of SSL[17] which is now deprecated and prohibited from use by IETF[18]. They are both cryptographic protocols which aims to provide privacy and integrity in the network communication between a client and a server. By using public key cryptography TLS provide the means to ensure the identity of the communicating parties and an initialization vector TLS ensures that, should the same plain text message occur twice, then the ciphertext will not be the same.

TLS Handshake The TLS handshake is an additional handshake where a client and server negotiate how to send encrypted messages between them. As TLS runs over a reliable transport such as TCP the client and server first have to complete a TCP three-way-handshake as described in section 3.4

[16] Pretty Good Privacy.
[17] Secure Sockets Layer.
[18] Internet Engineering Task Force.

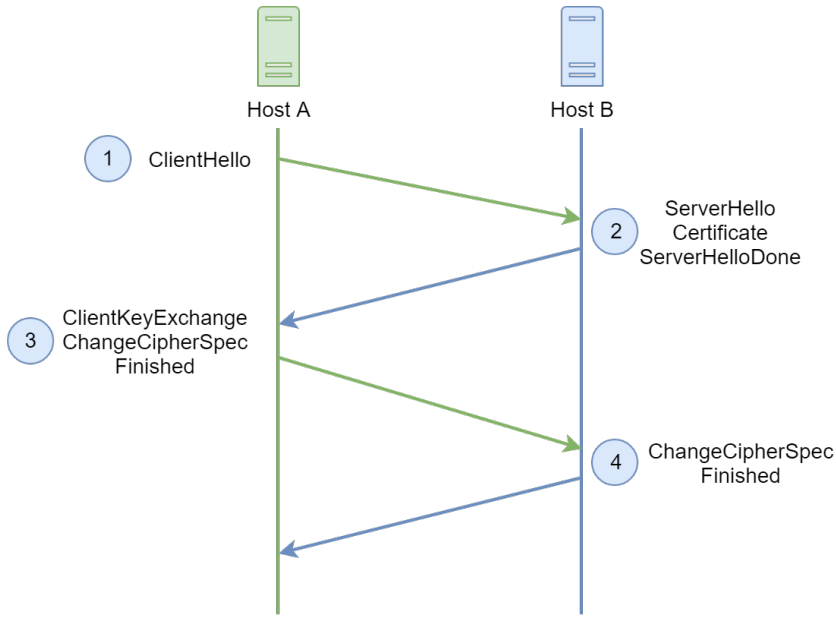


Figure 3.7: TLS Handshake.

The basic TLS handshake consists of several steps in order to set up an encrypted connection:

1. The client sends a *ClientHello* message which contains information about which TLS versions and ChipherSuites are supported in order of preference. The client also includes a random byte string or nonce[19].

2. The server responds with a *ServerHello* message that contains the chosen CipherSuite from the list provided by the client. The message also contains a session ID, another random byte string (nonce) and the servers digital certificate.

3. The client verifies the servers digital certificate and extracts the servers public key.

4. The client sends the random byte string that enables the client and server to compute the secret key they use to encrypt subsequent messages. The byte string itself is encrypted using the servers public key.

[19] Number used only once.

5. (Optional) if the server sent a *client certificate request* then the client sends a random byte string encrypted with the clients private key together with the clients digital certificate if it exist otherwise a *no digital certificate alert* is sent.
6. (Optional) the server verifies the clients certificate.
7. The client sends a *Finished* message which is encrypted using the secret key to indicate that the client part of the handshake is done.
8. The server also sends an encrypted *Finished* message to indicate that the server part of the handshake is done.

After the handshake is done and the client and server have established a shared secret key they can exchange messages that are symmetrically encrypted using the shared secret key.

3.7 Session/Flow

When dealing with network traffic it is necessary to distinguish between a session, a flow and an individual packet. A network flow is a uni-directional packet stream from one host to another. A flow is uniquely defined by a five tuple:

```
(source IP, source port, destination IP, destination port, protocol type)
```

As the session terminology varies between sources, we define a session as a bi-directional packet stream between two hosts. As a session can contain both directions of flows it means that the source and destination ports are interchangeable.

Chapter 4

Dataset

Pray, Mr. Babbage, if you put
into the machine wrong figures,
will the right answers come out?

Charles Babbage, Passage from the Life of a Philosopher

In the previous chapter we outlined the theory of network traffic. We have used this to create a dataset.
In this chapter we will describe some of the thoughts and work that went into creating a dataset for a supervised learning task.

The chapter begins with an introduction to the classes of the dataset. This is followed by how we have managed to capture network traffic and some of the post-processing mechanisms that has been applied to the captured data. Lastly some details on how to streamline the entire process is described.

4.1 Existing datasets

Existing datasets for classification of network traffic are widely available, however they all seem to have a different focus than what we were interested in. Some of them relies on extracted statistical features (e.g. mean time between packets, average packet size, etc.) from flows or sessions. We would like to avoid this type of feature extraction as a potential solution should be able to run in hardware at full speed (200Gbps) with millions of sessions at a time. The amount of data to be kept in memory while doing the necessary computations makes this solution unfeasiable. Other existing datasets that are not relying on extracting features have a different focus, such as malware detection [27], VPN-nonVPN [13] classification and intrusion detection [44]. As the focus of this thesis is to distinguish between streaming and non-streaming network traffic, we set out to create a dataset that serves our purpose.

Quality is imperative when creating a labelled dataset for a supervised learning task, so we avoid training on garbage. Garbage in, garbage out. To create a correctly labelled dataset, we had to set up a controlled environment for generating network traffic, capturing it, and labelling it.

4.2 Classes

As mentioned in a previous section, the focus of this thesis is classification of network traffic in order to identify video and audio streaming. As most streaming services use the same application-layer protocols (HTTP/HTTPS), we have simplified the problem by only including classes that utilize these. Protocols such as SMTP, FTP etc. is left out as they are not of particular interest in this streaming/non-streaming setting. We have chosen streaming services that are all commonly known in Denmark. The following will present the chosen classes and their characteristics:

HTTP: Regular unencrypted web traffic will be labelled *http*. This type of traffic is usually regular web browsing where a server is serving content on TCP port 80 [20]. Different types of web-browsing is needed to ensure diversity in the amount and length of network packets. Even though the traffic itself is not encrypted the TCP payload might still be encrypted at the application layer.

HTTPS: Regular web traffic, encrypted using SSL/TLS will be labelled *https*. This type of traffic is usually regular web browsing where a server is serving content on TCP port 443 [20]. Approximately half of the Internet traffic is using this protocol [12].

Netflix: Streaming data from Netflix will be labelled *netflix*. This is the actual video and audio stream from Netflix's CDN[20] and not the browsing of the Netflix homepage. The video/audio streams are served via HTTPS using TLS v1.2 encryption as described by Stewart and Long [46]. The streams are served on TCP port 443. Netflix use a JavaScript player to buffer and assemble fragments which appears to be in *ISO BMFF Byte Stream Format*.

HBO Nordic: Streaming data from HBO Nordic will be labelled *hbo*. The streams are served via HTTP over TCP port 80. Even though the streams are served over an unencrypted connection, the actual payload is protected by DRM[21]. HBO uses Adobe Flash Player to play the stream. As the traffic is unencrypted, using Wireshark, we observe a couple of things - The video stream is sent separate from the audio stream as two separate GET requests are made to HBO's CDN. The stream is sent in multiple fragments where a lot of fragments are requested in the beginning to build up an initial buffer. Once playing, the fragments are requested at a much slower rate. The fragments are buffered and assembled by Adobe Flash Player.

DR TV: Streaming data from DR TV will be labelled *drtv*. The stream is served by Akamai[22] CDN via HTTPS using TLS v1.2 over TCP port 443. DR TV uses a JavaScript player to buffer and assemble MPEG-2 TS-files/fragments which are found in a m3u8 manifest file which is a plain text file format. Some DR TV streams are encrypted on the application layer while others are not.

Twitch: Streaming data from Twitch will be labelled *twitch*. The stream is served via HTTPS using TLS v1.2 over TCP port 443. Twitch use a JavaScript video player to buffer and assemble H264 MPEG-4 AVC TS-files/fragments which are found in a single m3u8 manifest if it is a prerecorded video. If it is a "live" stream the video player receives a new manifest file at a regular interval with the latest TS-files.

[20] Content Delivery Network.
[21] Digital Rights Management.
[22] CDN used by DR: https://www.akamai.com/dk/da/.

YouTube: Streaming data from YouTube will be labelled *youtube*. The stream is served via the QUIC[23] protocol over UDP port 443 using QUIC crypto until TLS v1.3 is ready [26], but only if the user is using the Google Chrome browser. If the user is using another browser such as Mozilla Firefox YouTube will stream using HTTPS with TLS v1.2. YouTube use a HTML5 video player to buffer and assemble fragments.

4.3 Generating network traffic

In order to label network traffic correctly, we needed a controlled environment. In this environment we needed several different frameworks, libraries and different techniques in order to create a correctly labelled dataset. In this section we will describe the most important ones.

We have used Python, because this allows for utilization of many different libraries. As the project includes capturing of network traffic, calculations of general statistical measures, visualizations and machine learning, Python allows us to include all of this functionality within the same project. The following sections will describe the most important aspects of how we captured network traffic.

4.3.1 Selenium

Selenium is a framework that can automate tasks within different browsers and is mainly used for web application testing purposes[24]. Utilizing the key interface of the framework called Selenium WebDriver, allows for opening a browser instance as a user would, and perform interactions within the browser by use of locating elements within the DOM[25].
In order to generate the vast amount of streaming data utilized within the project, the Selenium WebDriver has allowed us to automate the task of opening specific streaming services and starting the streaming process.
Having worked extensively with the Selenium WebDriver interface, we have found an important feature of instantiating a browser with predefined settings. This allows for enabling features like flash and saving login credentials of the different services within the predefined settings and eases our tasks greatly.

[23] Quick UDP Internet Connections.
[24] https://www.seleniumhq.org/
[25] Document Object Model.

4.3.2 HTTP(S) getter

In order to generate HTTP and HTTPS traffic in a controlled way, we used a traffic generator written in Python called *PyTgen*[26] however the code was written in Python 2 and we therefore made the necessary adaptions to make it run in Python 3. PyTgen works by sampling randomly every 10 seconds from a list of host addresses (see appx. B), requesting the site and repeating for a fixed amount of tries, we chose only one try in order to not cause any suspicion. Had we chosen more retries then PyTgen would wait for a random amount of time bounded by an upper limit which can be specified by the user. This process continues in a fixed time window in order to switch between traffic types for easier labelling.
The captured traffic is then filtered based on the IP addresses of the visited domains. This way we ensure that only network traffic to and from the selected hosts was saved for later use.

4.3.3 Wireshark

Wireshark is an open-source tool to analyze network packets. Such a tool is often referred to as a network analyzer or sniffer. Its main functionality is to capture network traffic as data flow across a network interface.
Capturing traffic with Wireshark requires the user to specify which interface it should listen to, this can be a wireless network card, an Ethernet port, USB port, Bluetooth port, etc. Wireshark displays the captured packets' information from the data link layer and up (without the preamble and frame checksum). Wireshark implements a graphical user-interface that presents the captured network traffic in a structured manner and provides a large range of functionality for performing analysis on the captured traffic. For each data link frame that is captured, information such as source, destination, protocol type, and payload (if any) is obtained. Wireshark provides filtering functionality for extracting packets with certain characteristics, for example if one wants to locate the packets coming from a specific source, a filter on the source ip can be applied. Wireshark can export a collection of packets in different file formats, among those is the pcap file format. Figure 4.1 shows how Wireshark displays the captured packets. Wireshark is also available as a terminal oriented version (tshark) that allows for capturing and displaying packets when a GUI is not needed. Tshark has been utilized to capture network traffic.

[26] https://github.com/reissmann/PyTgen.

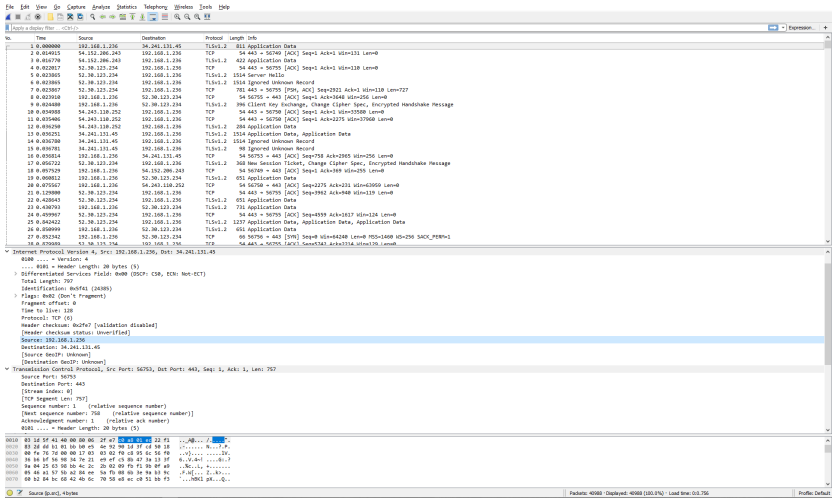


Figure 4.1: Loaded pcap file in Wireshark.

Capture filter In order to minimize the noise generated by other services we apply a capture filter to tshark. As the capturing was done on a remote server we had to use a VNC[27] server/client to control the process of capturing. The capture filter was used in order to disregard the VNC session traffic (not port 5901) and ensure that we only capture IPv4 packets (IP) in order to remove lower level protocols such as ARP[28] and STP[29]. As we were only interested in unicast communication, i.e. communication to and from our client PC, we removed the "noise" from broadcast and multicast announcements (not broadcast and not multicast). Putting it all together our capture filter looked like this:

```
port not 5901 and ip and not broadcast and not multicast
```

Listing 4.1: Capture filter

4.3.4 Pcap

Pcap is an API[30] for capturing network traffic. On Unix systems pcap comes with the library libpcap, while it on windows is most commonly known as WinPcap. As libpcap is the "de facto" standard library for network capturing on

[27] Virtual Network Computing.
[28] Address Resolution Protocol.
[29] Spanning Tree Protocol.
[30] Application Programming Interface.

Unix systems, its file format has also become the standard[31]. The extension of the file format is .pcap.

4.4 Post Processing

In order to process the captured network traffic we have applied several methods to extract the desired information. This section will outline the most important methods.

Scapy Scapy is a Python library which can manipulate, forge, decode, send and capture network packets. It can be used for reading pcap files and display the content of such, sorted by distinct network traffic flows. This tool is therefore of great use in a process that involves pcap files and the task of extracting specific information from those.

Custom session extractor Scapy comes with a built in "session" extractor, unfortunately it extract flows and not sessions as we pursue. Thus we created our own session extractor that groups packets flowing in both directions identified by the five tuple:

```
(source IP, source port, destination IP, destination port, protocol type)
```

where the source and destination IP and ports are interchangeable and packets transferred in either direction will be grouped together by our session extractor.

Correct labelling - threshold In order to ensure that only the correct streaming sessions were selected and labelled as their proper class, we decided to set a minimum threshold for how large the session should be in order to be labelled as the relevant streaming service. By manually analyzing several pcap files in Wireshark we found a threshold of 5000 packets in a session to be reasonable. The threshold is large enough to not include the browsing of the web pages done by the Selenium automation framework, but small enough to include most streaming even if the stream was slow to start.

[31] https://wiki.wireshark.org/Development/LibpcapFileFormat.

Packet representation As the packets come in the pcap format as a byte string we decided to interpret the individual bytes as integers in the range [0..255]. This way we get a vector of integers in the same length as the original packet.

Packet anonymiser In order to ensure that our machine learning model did not base its classification decisions on for instance UDP/TCP port numbers we decided to anonymise each frame. By setting the first 12 bytes of a packet to zero we removed both source and destination MAC address[32], as each address is 6 bytes long. Setting bytes 26 to 33 (inclusive) to zero removes the source and destination IP-addresses as each IPv4 address is 4 bytes long. Setting bytes 34 to 37 (inclusive) to zero we remove the port numbers. The port numbers are described using 2 bytes giving a port range from 0 to 65535. As described in chapter 3 the checksum of the IP header and the TCP/UDP header is calculated as a ones' compliment on 16bit words. As the checksum is calculated on the headers, the IP addresses and port numbers are used in the calculation. Since ones' compliment is a simple operation, it might show some correlation to the different IP addresses and port numbers and we therefore decided to remove the checksum from the IP header (bytes 24 and 25) and TCP/UDP header (TCP: bytes 50 and 51, UDP: bytes 40 and 41) of each packet. Appendix F.1 contains a description of the individual bytes. Figure 4.2 provides an overview of anonymised the header bytes.

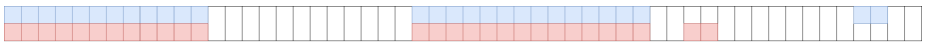


Figure 4.2: The anonymised header bytes. Blue indicates which bytes are anonymised in a packet from a TCP session. Red indicates which bytes are anonymised in a packet from a UDP session.

4.5 How to streamline the process

As we quickly discovered, capturing streaming traffic was a very slow process. Each streaming service needed to stream for 60 seconds in order for the sessions to be large enough to stand out from all other sessions.

Pandas and HDF format Hierarchical Data Format (HDF) is a file format designed to store large amounts of data. Experiencing that opening pcap-files

[32] Media Access Control address.

using Scapy could take more than an hour, we extracted the relevant fields from the pcap files and stored them in Pandas DataFrames, which we then saved in the HDF file format. By doing so we were able to open a HDF file using Pandas in approximately one second instead of an hour. The fields that we chose to extract and store were the following:

- **Time** the frametime as recorded by tshark.
- **IP destination** the IPv4 address of the destination host.
- **IP source** the IPv4 address of the source host.
- **Protocol** the protocol used in the session, TCP or UDP.
- **Port destination** the port used on the destination host.
- **Port source** the port used on the source host.
- **Bytes** the raw frame bytes as captured by tshark.
- **Label** the class-label of the session that the packet belongs to.

By storing those field we were able to uniquely identify and group each flow by its five-tuple and get the corresponding raw bytes and true label.

Multiprocess/Multithread In order to speed up the amount of streaming data captured. We looked into parallelisation of the streaming process. This was done by making the streaming services, that do not include a login procedure[33], multithreaded. This means multiple browsers was streaming from the same service at the same time, while capturing the network traffic. This way we would get a larger amount of data in each capture. As they are served by use of different ports and potentially also from different IP source addresses, the sessions are distinguishable even though streamed at the same time.

Another problem we faced was that of converting the pcap files to HDF format. A large amount of time was used to open each pcap file using Scapy in order to extract the relevant data for the Pandas Dataframe. We attempted a multithreaded approach for this issue. However, as Python use GIL[34] to ensure that only one thread run at a time, there was not a noticeable speedup by using

[33] Services in which a login is required often restrict the use of the service to a single tab in a browser. This is done such that one cannot share login credentials and utilize the service simultaneously.

[34] Global Interpreter Lock.

multiple threads. In order to avoid this limitation we moved to multiple processes, as they do not share the same memory space and does not have the GIL limitations that threads have within Python. In order to utilize multiple cores and to avoid communication between processes we gave each process a separate fraction of the files to be processed. Fortunately this proved to be a massive speedup.

Chapter 5

Experiments

> Don't spend your time looking around for something you want that can't be found.
>
> *Baloo (The Jungle Book)*

In the previous chapter we presented some of the thoughts and methods used when creating the dataset used in this thesis.
The purpose of this chapter is to describe the methods, experiments and results we have made in the process of classifying network traffic.

The chapter is structured such that first the experiments and results obtained when performing classification solely on the payload of the network packet is presented. This is followed by the experiments and results obtained when using the unencrypted headers of the network packet for the classification task.

5.1 Payload experiments

In this section we will explore the possibility of solving the classification task using transport layer payloads. This is a highly desirable solution for Napatech as a hardware implementation of such a classifier could be implemented using limited memory, making it applicable for real-time classification. The payload is either a TCP payload or a UDP payload and might be encrypted either using SSL/TLS[35] or on the application layer. We know from investigating the developer tools in the Google Chrome browser that Netflix sends a somewhat similar sequence in the beginning of each video segment being transferred. An example of this behaviour is shown in figure 5.1, it should however be noted that this byte string has been decrypted by the receiver before being displayed in Google Chrome.

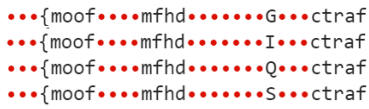


Figure 5.1: Example of beginning sequences taken from a Netflix video stream[36].

This behaviour is documented by W3C[37] as the *ISO BMFF Byte Stream Format*. The *moof* stands for Movie Fragment Box and *traf* stands for Track Fragment Box. This specific byte sequence thus makes it very likely that it originated from a movie streaming service.

5.1.1 Data extraction

To pursue this solution further, we extracted the payload from two h5 files where the traffic was captured on the same PC. One file containing Netflix sessions and another containing HTTPS browsing. As the classification should be based solely on the individual payload without any temporal information or information about previous packets, all maximum-length payloads were extracted, thereby all having the same length of 1460 bytes. By extracting maximum-length payloads we ended up with ≈ 45000 and ≈ 47000 payloads from Netflix and HTTPS respectively. The individual bytes values was then converted to

[35] This is not applicable for UDP as it requires a reliable transport protocol.
[36] https://www.netflix.com/watch/70276729.
[37] https://www.w3.org/2013/12/byte-stream-format-registry/isobmff-byte-stream-format.html.

integers in the range [0..255]. This way we get datapoints situated in a 1460 dimensional space.

5.1.2 Data exploration

In order to explore the data in greater detail, we perform dimensionality reduction through PCA.

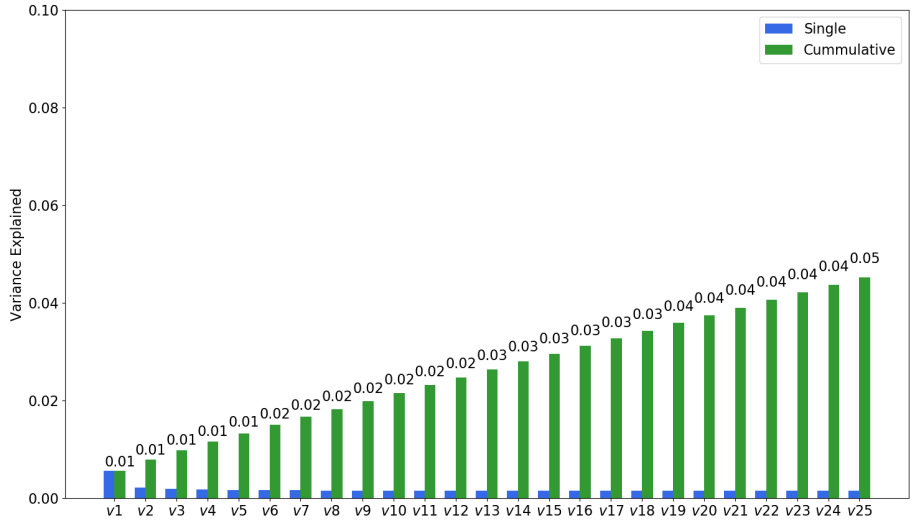


Figure 5.2: Cumulative plot of 25 first principal components when performing PCA on payloads taken from HTTPS and Netflix.

Figure 5.2 shows that the share of variance explained by the 25 first principal components is just about 5% and that each principal component only explains a minor fraction of the variance within the constructed dataset. This indicates that the datapoints are spread out in all dimensions. In figure 5.3 we see the projection of the data onto the first two principal components from which we see that the majority of the two classes in question are situated in the same cloud of datapoints. This is expected as PCA tries to maximize the variance explained by each component, and therefore might not capture if clusters are formed. This is where t-SNE is a more powerful tool. Should there be some clusters, then t-SNE will try to capture this information.

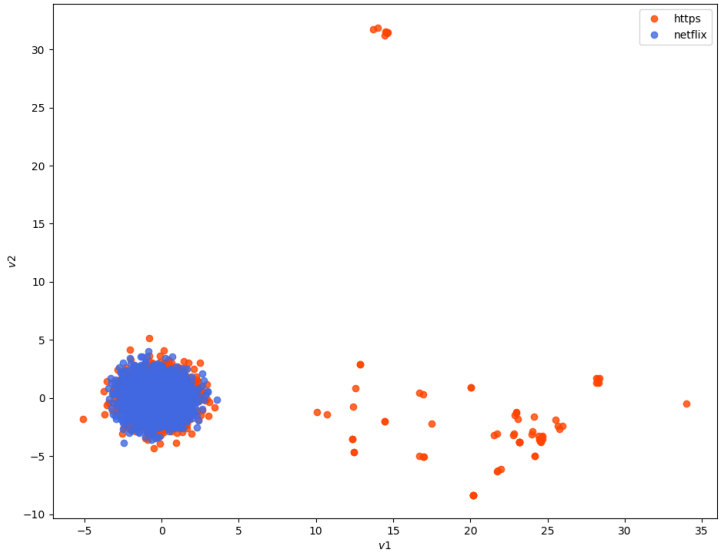


Figure 5.3: Plot of payloads taken from HTTPS and Netflix projected onto first two principal components. We see that the majority of *https* datapoints coincide with the *netflix* datapoints.

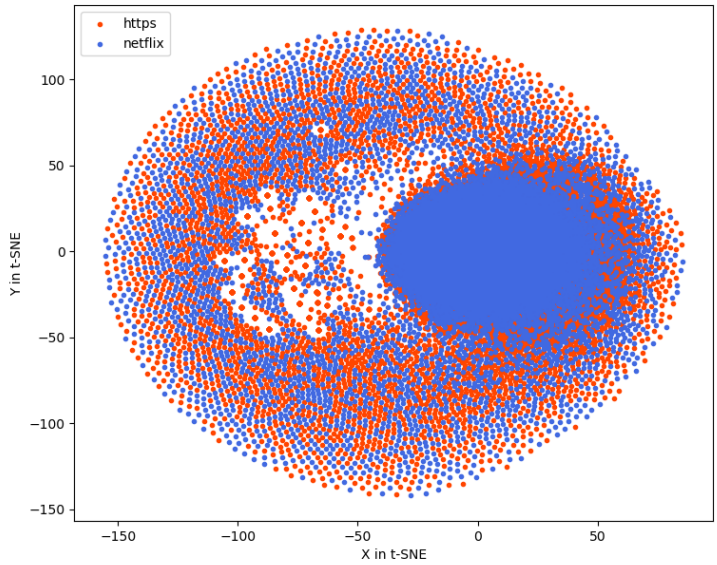


Figure 5.4: t-SNE plot with perplexity 10 over 5000 iterations of payloads taken from HTTPS and Netflix.

As can be seen in figure 5.4 there does not seem to be any clusters even with perplexity set to 10 which should capture even small clusters. We ran t-SNE with different perplexities in the range [2..100] and still no clusters seemed to be present. As t-SNE in the original version uses euclidean distance as metric and thus assumes local linearity, it might not perform well on a noisy dataset with high intrinsic dimensionality. Therefore we performed dimensionality reduction by PCA down to 50, 100 and 1000 dimensions and then ran t-SNE on the dataset of reduced dimensionality. All three runs showed similar results. The plot made from running t-SNE on the 50 dimensional dataset can be seen in figure 5.5, from which it is clear that the datapoints are distributed somewhat evenly equivalent with how the t-SNE technique would model random distributions according to Wattenberg et al. [51].

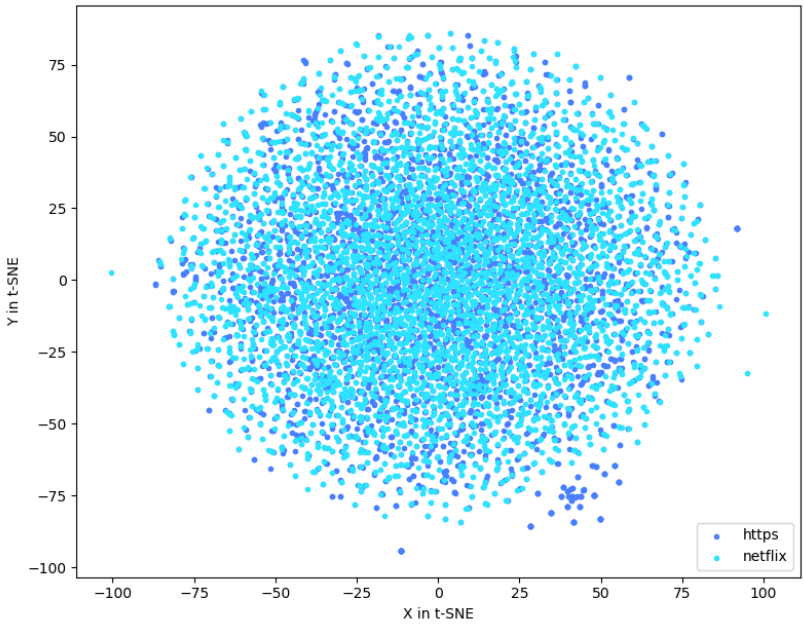


Figure 5.5: t-SNE plot with perplexity 10 over 1000 iterations of payloads taken from HTTPS and Netflix after PCA have been used to reduce number of dimensions to 50.

As the visualizations from PCA and t-SNE did not reveal any discriminative structure within the data, we looked into how the individual attribute values are distributed in the two classes. In figure 5.6 we see that in the Netflix payloads, the occurrence of the values [0..255] seem close to uniformly distributed, meaning

that no value occurs more frequently than others. We see that the *https* class is not distributed as evenly, for instance there is a higher occurrence of the number 48 (largest peak). In order to investigate whether if the number 48 is represented more frequently at a specific position in the payload, two heatmaps, one for each type of payload were created. The heatmaps are shown in figure 5.7a and 5.7b for which the index within the payload is mapped to the x-axis and the numerical value of the corresponding byte is mapped to the y-axis. Looking at figure 5.7a we see that the values of Netflix payloads are distributed evenly across the payload, as anticipated from the results in figure 5.6. As for *https* shown in figure 5.7b, we do see that some values e.g. 48 occurs more often than other values, however it appears that the value occur more often in all positions of the payload and is not bound to a specific location, this is indicated by the horizontal blue lines.

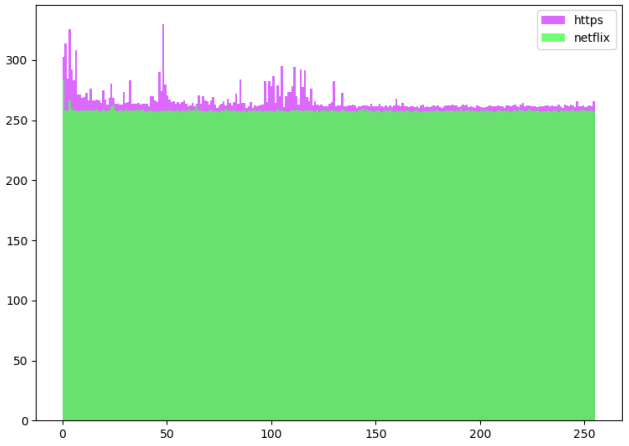
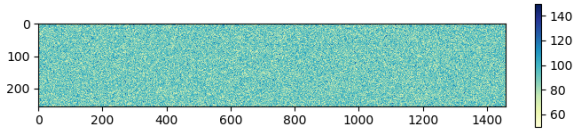
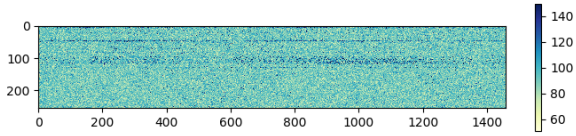


Figure 5.6: Histogram showing the amount of times a value in the range [0..255] occurs in a given class (y-axis scaled down by 1000).



(a) Netflix heatmap.



(b) HTTPS heatmap.

Figure 5.7: Heatmaps showing how byte values in the range [0..255] (y-axis) occur at index [0..1459] (x-axis) of the payloads for both the *netflix* class (a) and *https* class (b).

5.1.3 Exploratory Results

Even though the initial explorations of the dataset for the payload-based approach showed poor indications as a foundation for a classification task, we created small experiments using TensorFlow to train a neural network to classify individual packets from the two classes HTTPS and Netflix. In the first experiment we created a simple feed forward neural network with one hidden layer consisting of 100 units with a ReLU activation function and an output layer using the softmax function. Even though this is a relatively simple model it still has more than 146000 trainable parameters:

$$1460 \times 100 = 146000 \text{ (Weights)}$$
$$146000 + 100 = 146100 \text{ (Add bias)}$$
$$146100 + 100 \times 2 = 146300 \text{ (Add output weights)}$$
$$146300 + 2 = 146302 \text{ (Add output bias)}$$

The loss function used was cross-entropy and the Adam optimizer was used with a learning rate of 0.001. A batch size of 100 and 100 epochs was used. Of the $\approx$ 92000 datapoints we started with a training set of size 9200 in order to see if the network would fit the data at all. In that case we saw that the network was able to overfit the training data with a training accuracy of 1.0 after $\approx$ 50 epochs using early stopping, but the validation accuracy was just around 0.5 and the test accuracy was 0.506. The confusion matrix is shown in figure 5.8 where we see that the network does indeed not generalize to unseen

data as it appears to be randomly guessing. We then increased the amount of

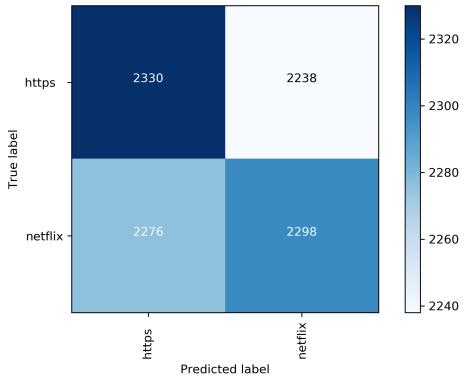


Figure 5.8: Confusion matrix after 50 epochs with accuracy of 0.506.

data to avoid overfitting. We found that when the amount of training data was larger than ≈ 40000 samples there were no signs of overfitting, but also no signs of any real learning, the validation and test accuracy would not go above 0.51 in any of our exploratory tests. We tried with more complex models having additional hidden layers each with up to 1000 hidden units, these did however not demonstrate any indications of actually learning the task.

5.1.4 Payload conclusion

Based on our experiments and visualizations it does not seem possible to classify individual network packets using a payload-based approach. The t-SNE plots revealed no discriminative structure within the data, and our experiments of training and testing on the data shows an ability to overfit the training set if the set is small enough or if the model is powerful enough, but it does not generalize to unseen datapoints. This indicates that there is no structure in the data to seperate the two classes which was somehow expected as the payloads are encrypted using TLS which is a public key encryption scheme having an initialization vector such that even the same message does not encrypt to the same cipher text.

5.2 Header experiments

In the previous section we described our findings when looking at a pure payload based classification approach. As described we did not find any promising structure or other features in an encrypted payload. In order to progress our classification task we investigated the unencrypted parts of the network packets: The individual headers of OSI layers 2,3 and 4, that is the headers of the data link layer, the network layer and the transport layer. We suspect that there might be a "signature" in the header that can be used to distinguish between the individual classes. However we do not believe that a single header of a session will be sufficient, since the TCP and TLS handshakes might be generic and thus not done in a specific way that would allow us to distinguish between the classes. Most streaming services use TCP as the transport layer protocol and TLS as encryption. Taking the first 16 headers from a session, the TCP three-way-handshake is the first three, then at least 4 for the setup of a secure TLS connection and then a few more to get the initial headers of the actual data transfer, hopefully provides sufficient information to form a signature datapoint. Wang et al. [49] has shown that it is possible to perform classification of end-to-end encrypted network traffic by using the first 784 bytes of a session. however Wang et al. include the payload in which we found no information useful for a classification task, therefore we choose to leave it out of our header-based experiments. Another common approach when performing network traffic classification is to create a feature vector based on the first 100 packets or so. This feature vector would then contain information about a given flow such as *mean-packet-size* and *mean-time-between-packets*. However as Napatech intend to implement a classification method in hardware it would be unfeasible as they potentially can have more than one million sessions at a time, all which should be stored in a buffer until sufficient packets has arrived and the feature vector can be composed. Choosing to use the first 16 headers of a session makes it a "simple" buffering exercise on the hardware level[38].

5.2.1 Data extraction

In order to create the dataset for the experiments we extracted the headers from the hdf files that we saved during the data generation process. From each session we extracted the headers of the data link layer, network layer and transport layer of the first 16 packets. This was accomplished by extracting the first 54 or 42 bytes for TCP and UDP packets respectively. Since there is a misalignment with the header length, the UDP header was zero padded up to 54 bytes before

[38] Stated in personal communication with Alex Omø Agerholm.

concatenating them. This results in input vectors of length $54 \cdot 16 = 864$ for both TCP and UDP as shown in figure 5.9.

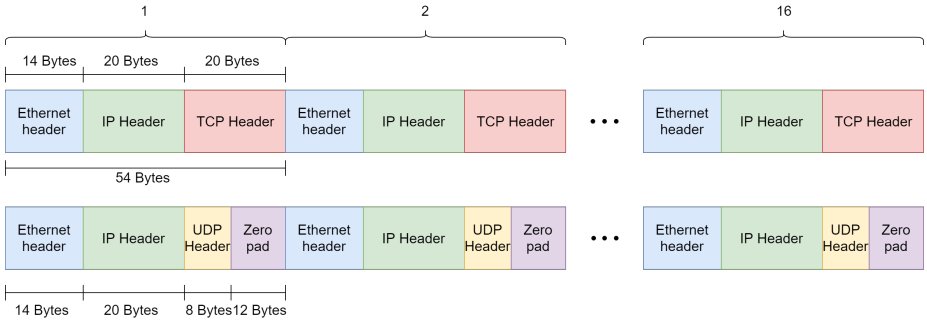


Figure 5.9: Datapoint creation by concatenation of headers from first 16 packets.

A caveat exist when extracting the first 54/42 bytes of a packet. If the packets of a session are fragmented, only the first packet will contain the transport layer (TCP/UDP) header, while the following fragments will contain payload data after byte number 34[39]. In order to avoid this, byte number 20, in which the three most significant bits are *zero, don't fragment* and *more fragments* flags in the IP header, were checked. Thus if a value of $0 <$ header[20] < 64 is present, then we skip that session as we know it is fragmented. We also check if header[21] $\neq 0$ as byte number 21 is the fragment offset which should be zero if the packet is not a fragment. Given that we anonymise each packet by removing MAC addresses, IP addresses, port numbers and checksums, we get a sparse input vector with a lot of zeros. We chose to do so rather than only extracting the fields of importance, since it allows for easier interpretation of which bytes in the original header the model bases a prediction on.

5.2.2 Data exploration

The dataset we created initially was generated on a PC provided by Napatech. The PC was running Ubuntu 17.10 and the browser used when streaming was Google Chrome version 64. Therefore we name the dataset LinuxChrome (LC). It consists of 19901 individual datapoints which distributes among the different classes in the following way:

[39] It still contains the headers of the data link layer and the network layer.

- *drtv:* 2486 datapoints
- *hbo:* 3663 datapoints
- *http:* 2313 datapoints
- *https:* 2194 datapoints
- *netflix:* 3134 datapoints
- *twitch:* 2970 datapoints
- *youtube:* 3141 datapoints

As we see the dataset is not perfectly balanced, but if we want to, we can balance it by randomly sampling a given amount from each class, either oversampling the underrepresented classes or by removing datapoints from classes that contain the most samples. In general we do not want to remove data, given the quality is good, as the general consensus within machine learning tasks is *the more the better*. When we look at the amount of datapoints, assuming a minute of streaming pr. datapoint for all classes except *http* and *https*, we get a total of 15394 minutes of stream or more than 10 days of streaming "watched".

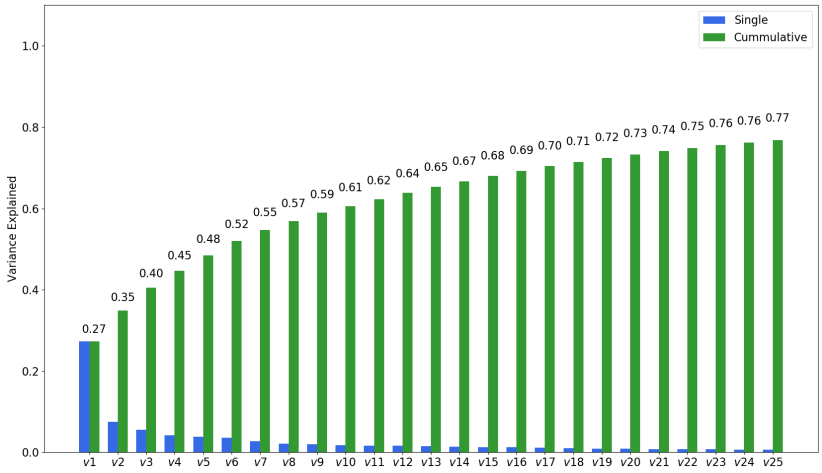


Figure 5.10: Plot of variance explained by different principal components from LinuxChrome dataset of the first 16 headers.

Performing PCA on the LC dataset showed a much more promising picture than what we saw in the payload analysis. Figure 5.10 shows that the first 25 principal

components account for more than 77% of the variance in the data. Looking at figure 5.11 in which the 864 dimensional dataset has been projected onto the first two principal components, we see that the classes are more separated than what we saw with the payload-based approach. Furthermore the non-streaming classes *http* and *https* in particular are separated from the majority of the other classes, only the *hbo* class coincide with them. In order to investigate further, t-SNE was performed on the LC dataset. Figure 5.12 shows that the t-SNE algorithm is able to find a low dimensional mapping of the high dimensional dataset that looks very promising for our classification task. The classes are grouped in clusters and even though the individual classes are divided into several clusters, they seem to be somewhat grouped in the same area of the plot.

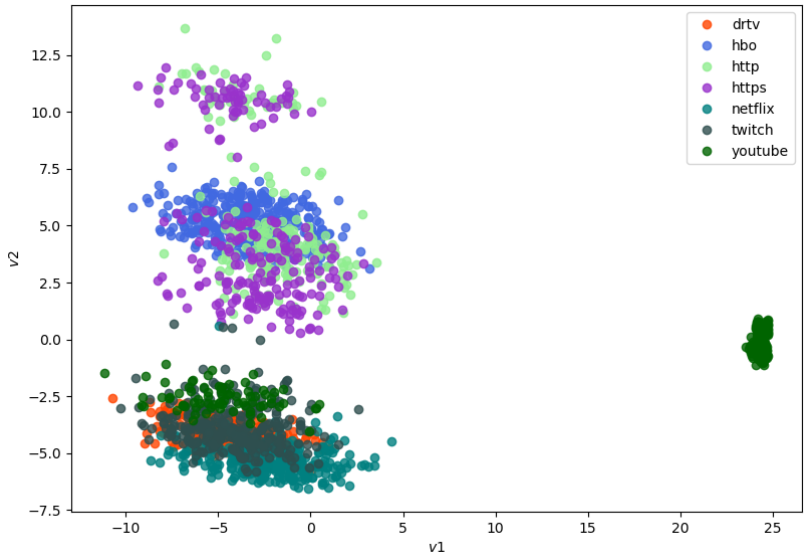


Figure 5.11: LinuxChrome dataset of the first 16 headers projected on v1 and v2.

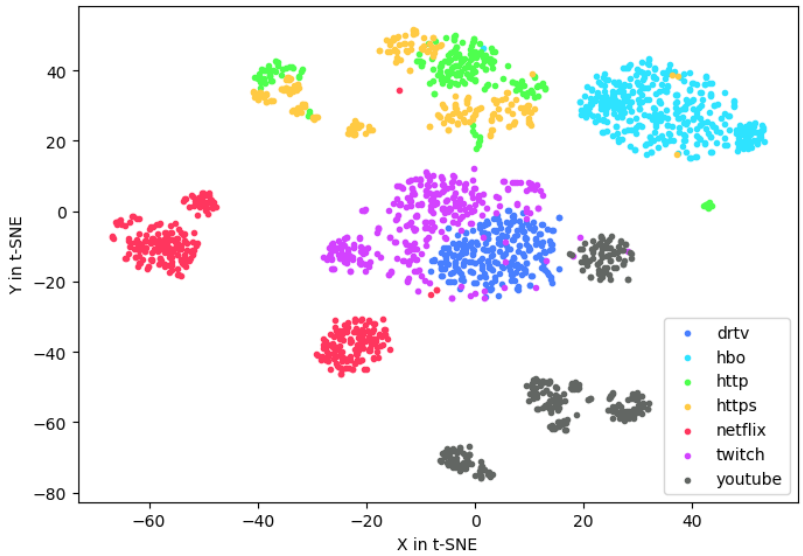


Figure 5.12: t-SNE plot with perplexity 30 over 1000 iterations of the first 16 headers from the LinuxChrome dataset.

Based on this analysis we believe that the approach taken with the LC dataset provides a good basis for a classification task. In fact by looking at the t-SNE plot it seems possible to perform classification of individual classes, and not only classification in the streaming/non-streaming context.

5.2.3 Initial architecture

In order to test if the t-SNE plot was indicative of whether the approach taken with the LC dataset was suited for our learning task, the dataset was split into train, validation and test by using an 80/10/10 split. A simple neural network, consisting of a single hidden layer with 50 units having ReLU activation functions, was implemented. The output layer comprised a 7 (one for each of the defined classes) neuron layer using the softmax activation function. As a loss function, cross-entropy was chosen. In order to minimize cross-entropy a gradient based approach was used. As the consensus is that gradient decent methods with momentum seem to converge faster, Adam [24] with a learning rate of 0.001 was used. By implementing early stopping with a patience of 20 epochs and a min-delta of 0.05, training was stopped if the cross-entropy calculated on the validation set did not decrease by 0.05 in 20 epochs. The batch size was set to 100.

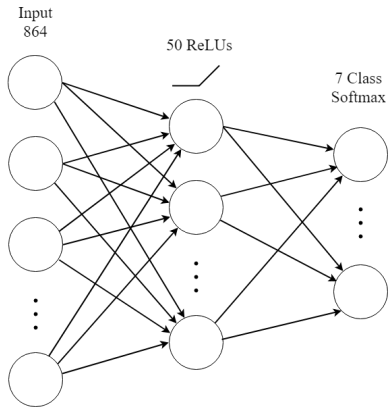
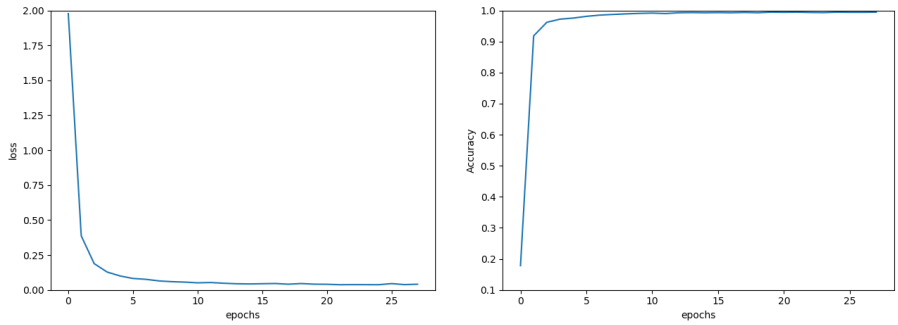


Figure 5.13: Brief overview of the architecture of the neural network with a single hidden layer.

5.2.4 Initial results

When training the aforementioned model on the LC dataset, we see from figure 5.14a that the validation loss decreases very quickly and figure 5.14b shows that the validation accuracy increases accordingly.



(a) Epochs vs. loss.

(b) Epochs vs. accuracy.

Figure 5.14: (a) Plot showing how the validation loss decreases when training on the LinuxChrome dataset (b) Plot Showing how the validation accuracy increases when training on the LinuxChrome dataset.

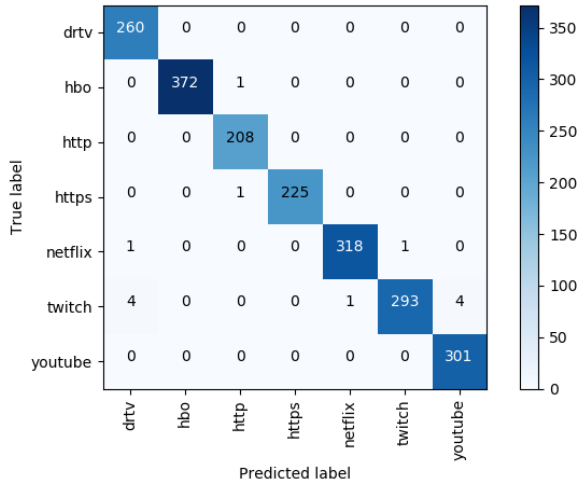


Figure 5.15: Confusion matrix when training and testing with a 80/10/10 split on data from LinuxChrome dataset of the first 16 headers. Accuracy of 0.993.

When testing the trained model on the test-split of the LC dataset we get the confusion matrix shown in figure 5.15. Looking at the confusion matrix in figure 5.15 we see a very impressive result, especially in the streaming/non-streaming context. Only a single streaming (*hbo*) session has been labelled as

non-streaming (*http*) and we see that not a single time has a non-streaming session (*http/https*) been labelled as a streaming session. This result is made clear by the streaming/non-streaming confusion matrix shown in figure 5.16.

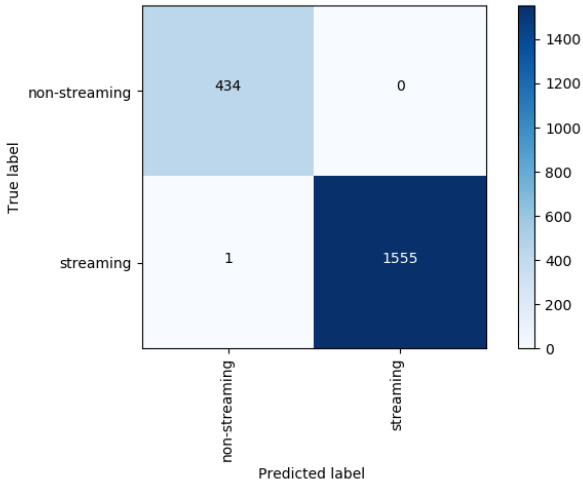


Figure 5.16: Confusion matrix of the streaming/non-streaming context created from results shown in figure 5.15. Accuracy of 0.999.

From the streaming/non-streaming confusion matrix in figure 5.16 the following performance metrics can be obtained:

$$\text{Non-streaming precision} = \frac{434}{434+1} = 0.998$$
$$\text{Non-streaming recall} = \frac{434}{434+0} = 1$$
$$\text{Streaming precision} = \frac{1555}{1555+0} = 1$$
$$\text{Streaming recall} = \frac{1555}{1555+1} = 0.999$$

The fact that not a single non-streaming session has been misclassified seems very promising. From Napatech's point of view[40] it is better to save a streaming session that should not have been saved than to discard a non-streaming session that should have been saved. Thereby the recall of the non-streaming class is most important. A perfect recall of the non-streaming class would also yield a perfect precision of the streaming class[41].

[40] Stated in personal communication with Alex Omø Agerholm.
[41] Assuming that we have at least one prediction of the streaming class.

5.2.5 Robustness

In order to test how well the trained model generalize to network traffic generated on another PC, another dataset called WindowsFirefox (WF) was created. The WF dataset was created by running the same Python scripts as with the LC dataset, but the creation differs in the following areas:

- **OS:** WF was generated on a PC running Windows 10 Professional rather than Ubuntu 17.10
- **Browser:** WF was generated using the Mozilla Firefox Quantum browser
- **Location:** WF was generated in a different physical location, hence another ISP[42] was used.

This WindowsFirefox (WF) dataset consist of 26292 individual datapoints with the following class distribution:

- **drtv:** 5243 datapoints
- **hbo:** 2386 datapoints
- **http:** 3722 datapoints
- **https:** 3378 datapoints
- **netflix:** 4321 datapoints
- **twitch:** 4216 datapoints
- **youtube:** 3026 datapoints

In order to test how well the model generalize to new data it was trained on the LC dataset and tested on the entire WF dataset. We have previously shown how the model is able to learn the representation of the LC dataset. However, when looking at the confusion matrix shown in figure 5.17 it becomes clear that the model trained solely on the LC dataset does not perform well when being tested on the WF dataset. It obtains an accuracy of 0.438 which is better than random guessing, but not acceptable in a real world scenario.

[42] Internet Service Provider.

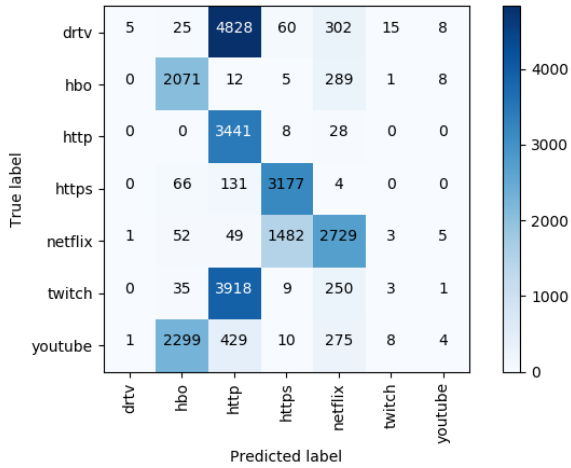


Figure 5.17: Confusion matrix when training on LinuxChrome and testing on WindowsFirefox both with the first 16 headers. Accuracy of 0.438.

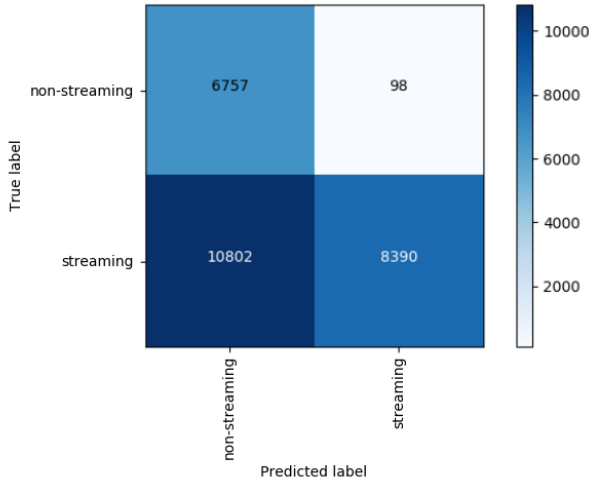


Figure 5.18: Confusion matrix of the streaming/non-streaming context created from results shown in figure 5.17. Accuracy of 0.582.

From the streaming/non-streaming confusion matrix in figure 5.18 it is clear that the classifier does not qualify for the task of separating the two classes. It has an accuracy of 0.582 and the following performance metrics:

$$\text{Non-streaming precision} = \frac{6757}{6757 + 10802} = 0.385$$
$$\text{Non-streaming recall} = \frac{6757}{6757 + 98} = 0.986$$
$$\text{Streaming precision} = \frac{8390}{8390 + 98} = 0.988$$
$$\text{Streaming recall} = \frac{8390}{8390 + 10802} = 0.437$$

Looking at the numbers we see one positive tendency which is that non-streaming recall is very good. However, we also see that more than half of the streaming sessions are labelled as non-streaming. For Napatech this is not an optimal solution as the classifier is not good at recognizing the streaming sessions.

By applying t-SNE on the LC and WF datasets we gain much more insight into why the model trained on the LC dataset perform remarkably worse when tested on the WF dataset. As can be seen in figure 5.19 it becomes clear that the only classes where the two datasets overlap in the 864-dimensional space is the *hbo*, *http* and *https* classes, this is also reflected in the confusion matrix in figure 5.17 in which these three classes are the only ones being predicted somewhat well. The other classes are separated from one another when comparing between the two datasets. We do however see similarities between the datasets, e.g. LC-twitch (purple +) and LC-drtv (dark-blue +) are situated near each other, similarly we see this for WF-twitch (purple o) and WF-drtv (dark-blue o). It becomes apparent that the datasets must come from two different distributions but it would seem as the individual classes are still separable and thus it should be possible to merge the datasets and get decent results. By merging the two datasets and creating a 80/10/10 split we see from figures 5.20a and 5.20b that it is indeed possible to learn both representations of each class.

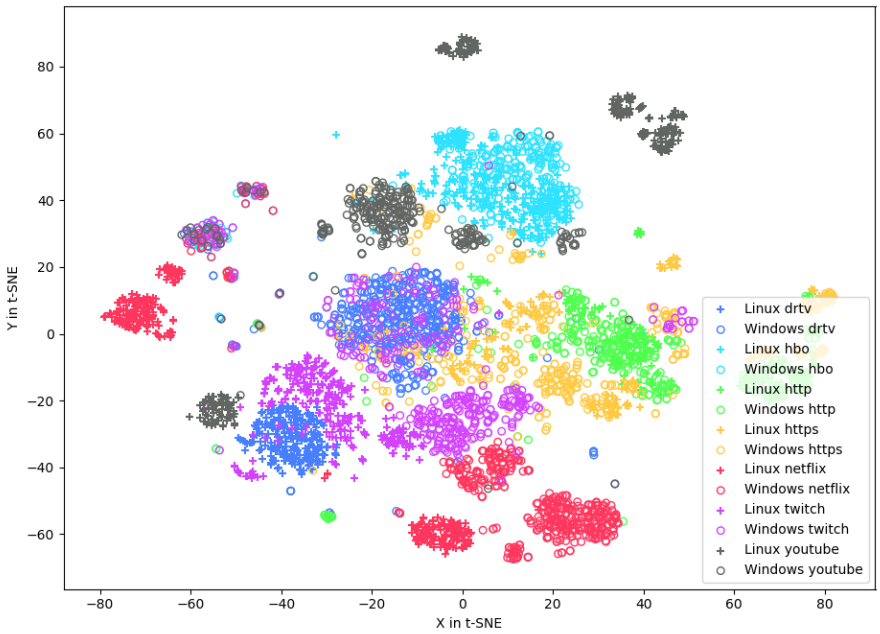
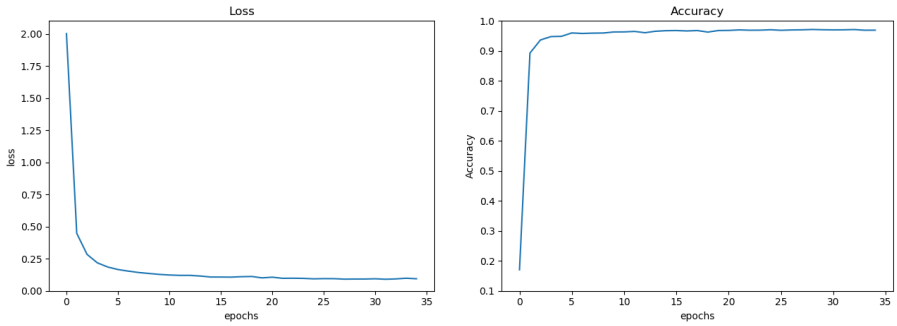


Figure 5.19: t-SNE plot with perplexity 30 over 1000 iterations of the first 16 headers from the LinuxChrome and WindowsFirefox datasets, linux is represented by a '+' and windows is represented by a 'o'.



(a) Epochs vs. loss.

(b) Epochs vs. accuracy.

Figure 5.20: (a) Plot showing how the validation loss decreases when training and testing with a 80/10/10 split on data from LinuxChrome and WindowsFirefox merged (b) Plot Showing how the validation accuracy increases when training and testing with a 80/10/10 split on data from LinuxChrome and WindowsFirefox merged.

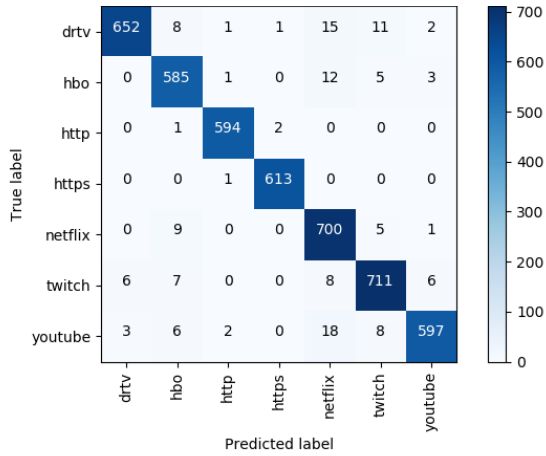


Figure 5.21: Confusion matrix when training and testing with a 80/10/10 split on data from LinuxChrome and WindowsFirefox merged, both with the first 16 headers. Accuracy of 0.970.

In fact when looking at the confusion matrix in figure 5.21 we see that the neural network has little to no problems in separating the different classes as the overall accuracy is 0.97. When reducing the problem to streaming vs. non-streaming the performance becomes even more impressive, shown by the resulting confusion matrix in figure 5.22.

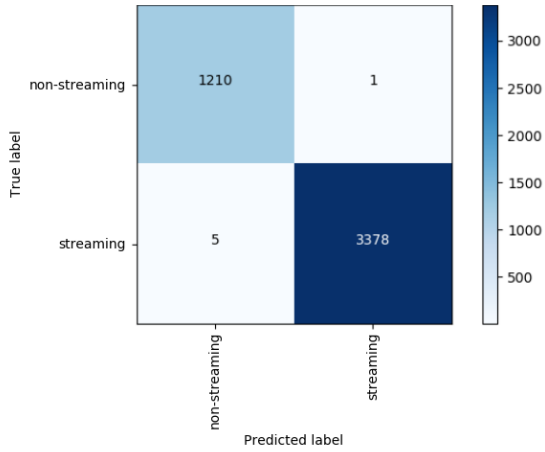


Figure 5.22: Confusion matrix of the streaming/non-streaming context created from results shown in figure 5.21. Accuracy of 0.999.

Comparing how the two datasets were generated, the following differences were identified:

- **OS:** The difference in operating system might make a difference in how the network traffic is handled.
- **Browser:** The difference in browser might make a difference. We know that Google Chrome supports the QUIC protocol while Mozilla Firefox Quantum does not.
- **ISP:** The internet service providers could use different services such as Netflix Open Connect[43] servers which are located at the edge (individual ISPs) of the internet.
- **Hardware:** The different hardware used when generating the network traffic might also make a difference.

In order to investigate what could be the culprit of our initial poor generalisation, a small set of new datasets was created:

- In order to test if the physical location, ISP and the hardware makes a huge difference in the underlying dataset distributions, two small datasets were created. One called WindowsAndreas (WA) and another called WindowsSalik (WS). Both were created on a PC running Microsoft Windows 10 and using the Google Chrome browser. One was created in Copenhagen with Yousee as the ISP (WA), the other was created in Holbæk with Fibia as the ISP (WS).
- In order to test if the browser alone makes a difference, a small dataset called WindowsChrome (WC) was created on the same PC as the WindowsFirefox (WF) dataset.

The hardware and software specifications behind each dataset can be found in appendix E. In table 5.1 a brief overview of all the datasets based on the 16 header approach can be found.

[43] https://openconnect.netflix.com/en/.

	LC	WF	WC	WA	WS
drtv	2486	5243	2010	703	1063
hbo	3663	2386	572	284	676
http	2313	3477	665	874	815
https	2194	3378	565	940	827
netflix	3134	4321	979	463	474
twitch	2970	4216	1627	708	1102
youtube	3141	3026	618	430	589
total	19901	26047	7036	4402	5546

Table 5.1: Table showing how the different classes are distributed in the different datasets. All dataset are based on the first 16 headers of a given session.

By training on the WS dataset and testing on the WA dataset we try to isolate the impact on the performance of the model caused by other factors than the browser and operating system. It should however be noted that the datasets are somewhat small compared to the LC and WF datasets. From table 5.1 we see that the classes of the datasets are not balanced, however by inspecting the confusion matrix shown in figure 5.23 we see that the performance is not affected to the same degree as in the LC vs. WF experiment. The imbalance of the classes does not seem to have a huge impact on the predictive power of the model, e.g. the *netflix* and *youtube* classes are somewhat underrepresented in the WS dataset, but the confusion matrix shows very good precision and recall for both classes when testing on WA.

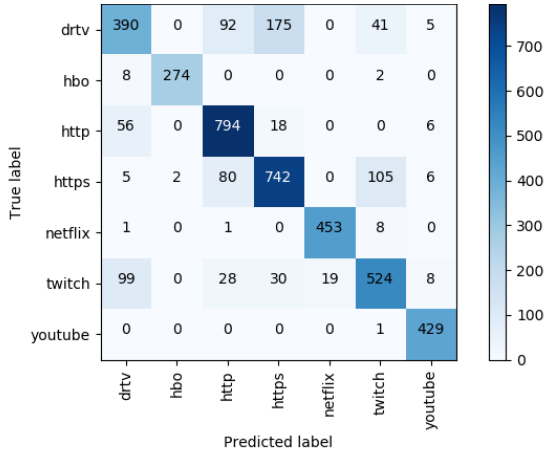


Figure 5.23: Confusion matrix when training on the WindowsSalik dataset and testing on the WindowsAndreas dataset, both with the first 16 headers. Accuracy of 0.820.

From this experiment it becomes clear that the ISP, hardware or physical location is not the main culprit of the bad generalization performance. The performance is not as good as when testing on a collection of samples from the same dataset in a train/test split, but the smaller size of the dataset might influence how well the model can be trained.

By training on the WF and testing on the WC dataset we try to examine if the specific browser influences the performance of the model. The WF dataset was generated using the Mozilla Firefox Quantum browser whereas the WC dataset was generated on the same PC using the Google Chrome browser. The confusion matrix shown in figure 5.24 displays that the type of browser does make a difference. The fact that Mozilla Firefox Quantum use TCP with TLS to communicate with YouTube, where Google Chrome use QUIC over UDP which is the reason behind the misclassifications of *youtube* datapoints. In fact the misclassified YouTube traffic accounts for more than 6 percentage points of the loss in accuracy. Besides from the fact that the browsers allow the use of different transport layer protocols in some cases, the results reflect that there are more similarities than differences between the distributions.

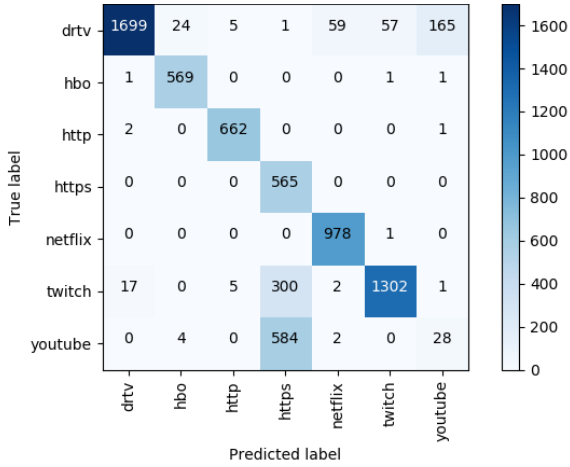


Figure 5.24: Confusion matrix when training on the WindowsFirefox dataset and testing on the WindowsChrome dataset, both with the first 16 headers. Accuracy of 0.825.

From this experiment we see that the browser itself does not have a huge impact on the representation of a datapoint, other than the fact that browsers can dictate which transport layer protocols are being used. One should account for this when training the model, by including datapoints from all variants of a

service in the training data. From the previous two experiments it would seem as if the main source of generalization error is the operating system. A final test to confirm this hypothesis was conducted by training on the LC dataset and testing on the WC dataset. When comparing figure 5.25 from this experiment and figure 5.17 we see similar results. The accuracy drops dramatically to 0.344 and 0.438 respectively. The results from the experiments with training and testing on datasets generated on different operating systems, leads us to believe that the actual implementation of the TCP/IP stack are very different among operating systems. This is confirmed to some extend in the fact that methods such as TCP/IP stack fingerprinting [44] exists. These fingerprinting techniques rely on features within the protocol specification that are left up to the specific implementation. Features such as *initial packet size, initial Time-To-Live, Window Size* etc. all of which are header fields that are used in our classifier.

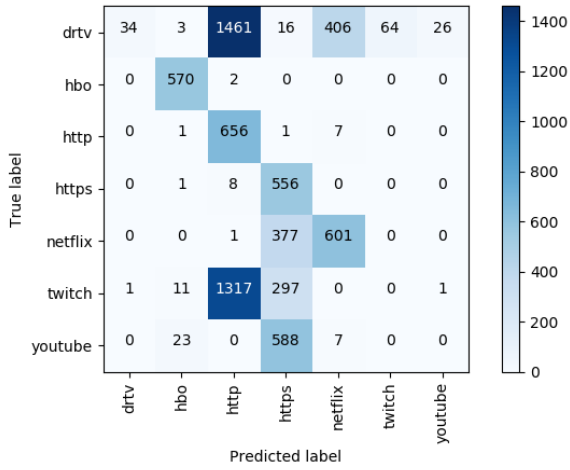


Figure 5.25: Confusion matrix when training on the LinuxChrome dataset and testing on the WindowsChrome dataset, both with the first 16 headers. Accuracy of 0.334.

5.2.6 Evaluation

Based on our experiments we find that it is possible to distinguish between streaming and non-streaming using our concatenated header approach. There seems to be a signature in how the session is set up, which can be captured

[44] TCP/IP stack fingerprinting is a technique that can infer a remote host's operating system.

using information from the first 16 headers. Furthermore we find that in order to learn a general representation, training data from different operating systems and browsers are needed, hence the best approach is to gather as much data as possible and then merge it all in order to include the variation within each class. As there are a finite number of operating systems and browsers, this approach is somewhat scaleable. According to StatCounter [14], Microsoft Windows as of 04-2018 accounts for more than 81% of the internet traffic generated by desktop operating systems. Apple OS X accounts for more than 13% thus simply by generating samples from those two families of operating systems $\approx 95\%$ of desktop internet traffic is accounted for.
In order to show that the header-based approach is a satisfactory solution to the classification task, all the datasets were merged and a 80/10/10 split was used to train and test our single hidden layer feedforward neural network. The result of that particular experiment can be seen in the confusion matrix shown in figure 5.26 which shows that the model is able to predict the individual classes with an average accuracy of 0.964 in a 10-fold cross-validation setting[45]. Figure 5.27 shows the ROC curves which supports the good predictive power of the model.

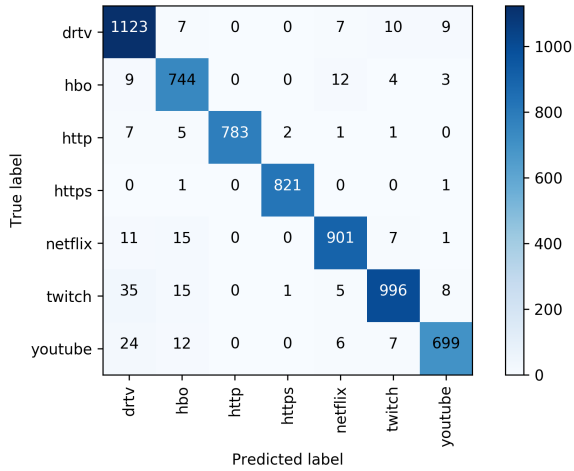


Figure 5.26: Confusion matrix when training and testing with a 80/10/10 split on data from all datasets merged, all with the first 16 headers. Accuracy of 0.964.

[45] Cross-validation has been applied to estimate the generalization error of the classifier. Appendix G shows the results of each fold when applying 10-fold cross-validation.

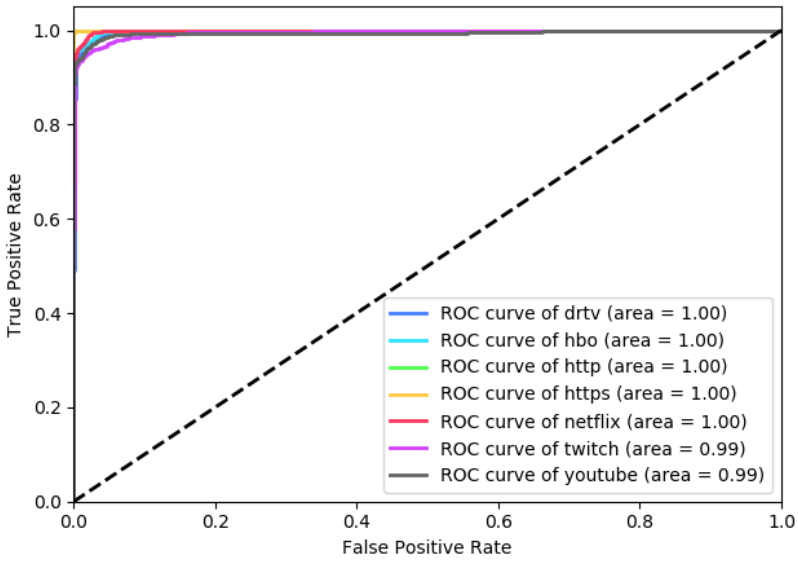


Figure 5.27: ROC curves when using the first 16 headers on a neural net-work with 50 hidden units. Note that the legends of the AUC values for the individual classes are rounded.

Translating the results to the streaming/non-streaming context yields an accuracy of 0.999, as shown in figure 5.28. This corresponds to an AUC of ≈ 1, as can be seen in the plot of the ROC curve in figure 5.29. In appendix D, PCA and t-SNE plots can be found for the merged dataset where the described tendencies of the previous experiments are also present. Mainly that the first 25 principal components account for a large portion of the variance within the merged dataset as shown in figure D.1. Furthermore the t-SNE plot in figure D.3 shows that the individual clusters seem separable. Our results confirm this, and in the streaming/non-streaming context we get the following metrics as calculated from the results shown in the confusion matrix in figure 5.28.

$$\text{Non-streaming precision} = \frac{1552}{1552+3} = 0.998$$

$$\text{Non-streaming recall} = \frac{1552}{1552+2} = 0.999$$

$$\text{Streaming precision} = \frac{4736}{4736+2} \approx 1.0$$

$$\text{Streaming recall} = \frac{4736}{4736+3} \approx 1.0$$

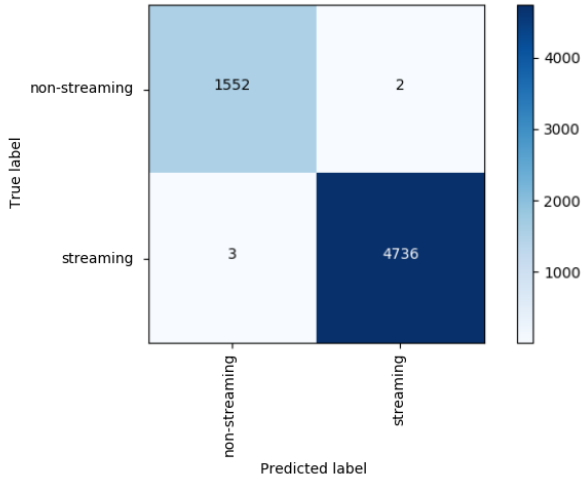


Figure 5.28: Confusion matrix of the streaming/non-streaming context created from results shown in figure 5.26. Accuracy of 0.999.

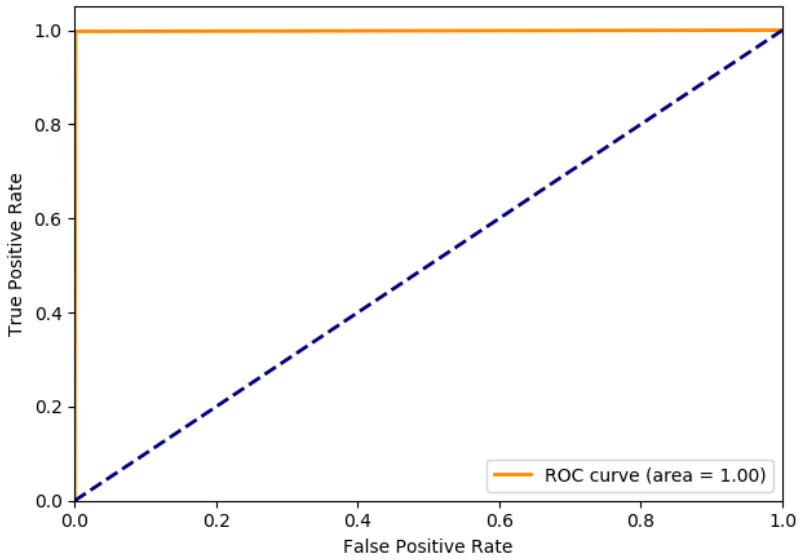


Figure 5.29: ROC curve of streaming class when using the first 16 headers on a neural network with 50 hidden units. AUC of ≈ 1.

Train	Test	TestAcc	Precision	Recall
LC	80/10/10 split	0.99	0.99	0.99
LC	WF	0.44	0.33	0.44
LC + WF	80/10/10 split	0.96	0.96	0.96
S	A	0.85	0.85	0.85
WF	WC	0.82	0.84	0.82
LC	WC	0.33	0.48	0.34
All	80/10/10 split	0.96	0.96	0.96

Table 5.2: Table showing experimental results of the different datasets. All results are for the full 7 class classification task.

Logistic regression The performance of the relatively simple neural network architecture indicate that the learning task is not highly complex. To further investigate this claim we apply a logistic regression model as a baseline classifier. The logistic regression model is similar to that of a linear regression, however the dependent variable is a binary outcome, i.e. 0 or 1. This can be extended to a multiclass problem by a strategy called "one-vs-rest", in which K-1 logistic classifiers are trained having K as the number of classes. This approach is also known by the name multinomial logistic regression.
Training a logistic regression classifier with a 80/10/10 split of all data merged using 16 headers, we obtain an overall accuracy of 0.94 and average precision, recall and f1-score of 0.94. The results are displayed in the confusion matrix, figure 5.30.

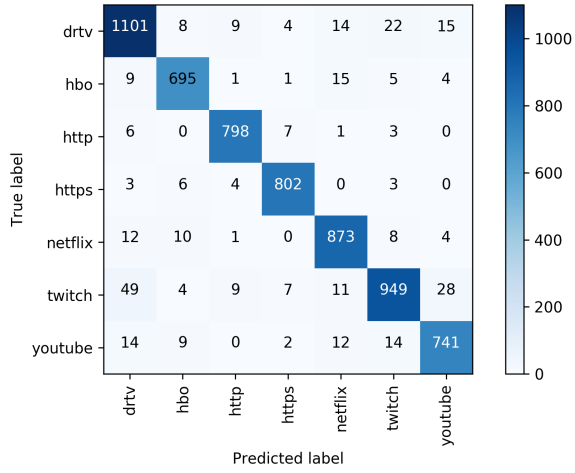


Figure 5.30: Confusion matrix for the logistic regression model.

The performance of the logistic regression classifier supports the claim that the learning task is to some extent linearly separable and why the neural network can obtain a good performance despite its small size.

How small can we go As the underlying goal of this thesis is to find a solution that is acceptable in a hardware implementation capable of performing real-time classification the amount of calculations should be kept at a minimum. The calculations involved when performing classifications using a neural network is at its core matrix- and vector-multiplications, for which the input length and the amount of trainable parameters influence the size of these matrices. Ideally we want to achieve a good performance with little computational effort. To investigate the options of this, we extracted 1, 2, 4, 8, 16, 32, 64 and 128[46] headers for all the captured sessions and trained and tested our model. In figure 5.31 the number of headers is plotted against the overall accuracy. From this we see that in order to have an accuracy of more than 90%, at least the first 8 headers are needed. Using 8 headers instead of 16 cuts the input length in half, from $54 \cdot 16 = 864$ to $54 \cdot 8 = 432$, and since we only have a single hidden layer this have a huge impact on the number of parameters. It should however be noted that as the amount of headers go up, the amount of datapoints go slightly down as not all sessions have the required amount of packets. This affects the *http* and *https* classes in particular. However almost identical performance can be achieved by using 8, 16 or 32 headers.

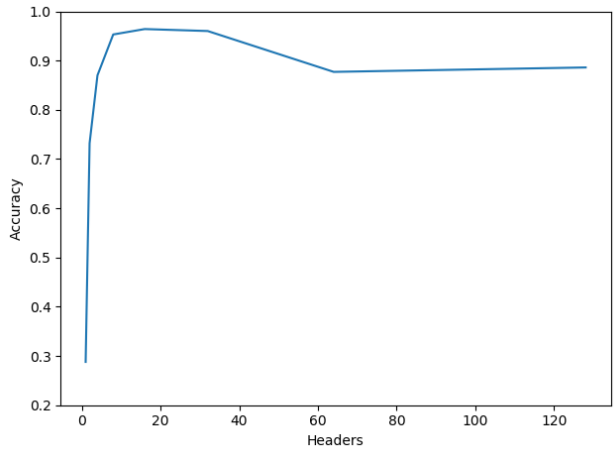


Figure 5.31: Number of headers extracted from sessions plotted against accuracy.

[46] The options of using more than 16 headers was explored to see how the model would perform with more information included.

From all our experiments it becomes apparent that the learning task itself is not that difficult. In fact our simple model has no problem in learning the representation of our data, and does so very fast as can be seen in one of the epochs vs accuracy plots, 5.14b. In order to drive the number of parameters down, a linear search on the number of hidden units was conducted in order to show how well the network performs with fewer neurons in the hidden layer. In figure 5.32 we see that the representation is easily learned by even a very simple neural network. From a network with 5 hidden units the accuracy is already above 95%, it should however be noted that it takes much longer for the network to reach convergence.

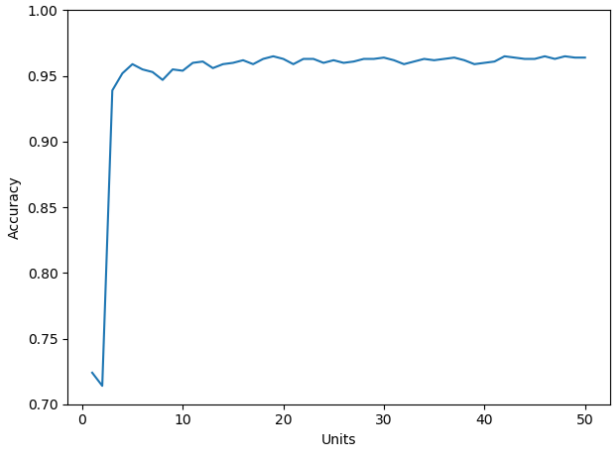


Figure 5.32: Number of units in hidden layer plotted against accuracy.

Thus by training a neural network with 5 hidden units having ReLU activation functions, on the merged dataset with the first 8 headers, the number of trainable parameters is 2207:

$$\begin{aligned} 432 \cdot 5 &= 2160 && \text{(Weights)} \\ 2160 + 5 &= 2165 && \text{(Bias)} \\ 5 \cdot 7 &= 35 && \text{(Output weights)} \\ 35 + 7 &= 42 && \text{(Output bias)} \\ 2165 + 42 &= 2207 && \text{Total parameters} \end{aligned}$$

This very simple neural network achieves an overall accuracy of 0.926 when classifying all 7 classes in a 80/10/10 split of the merged datasets with 8 headers. The ROC curves for all classes are shown in figure 5.33 in which the AUC ranges from 0.97 to 1.00 depending on the individual class.

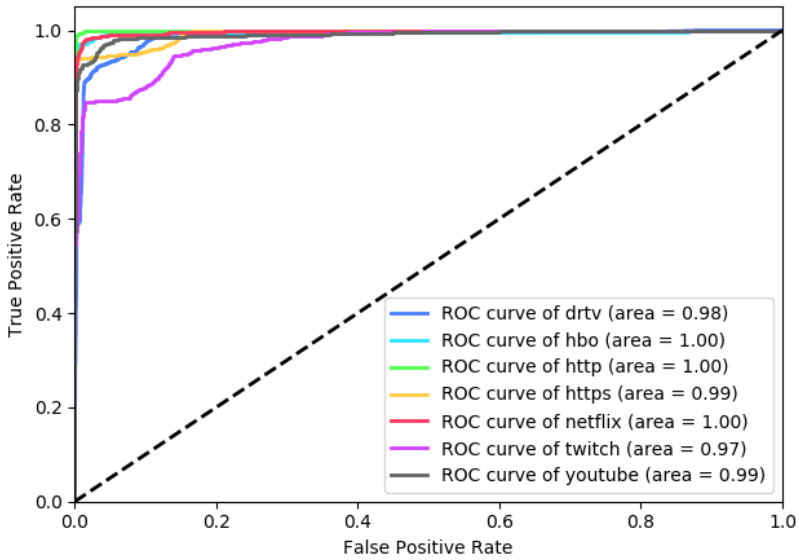


Figure 5.33: ROC curves when using the first 8 headers on a neural network with 5 hidden units.

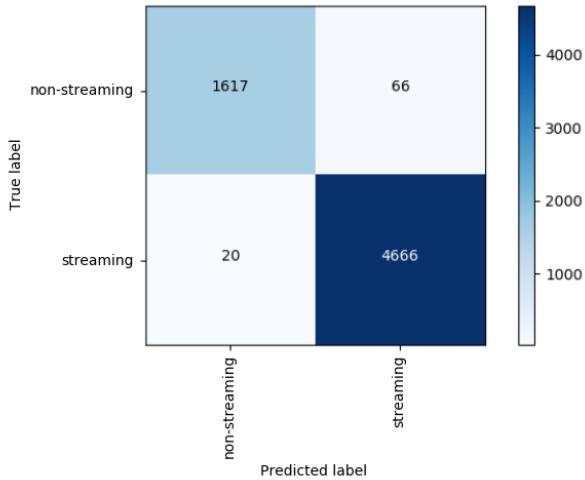


Figure 5.34: Confusion matrix of the streaming/non-streaming context. Accuracy of 0.986.

Converting the results to the streaming/non-streaming task, the accuracy is an impressive 0.986, which is reflected by the confusion matrix in figure 5.34. Figure 5.35 showing the ROC curve with an AUC of 0.98, confirms that this very simple network have an strong predictive power.

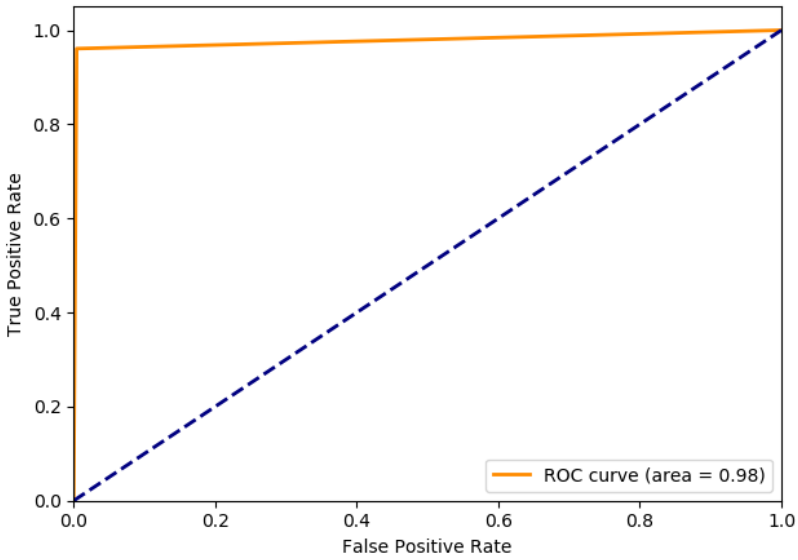


Figure 5.35: ROC curve of streaming class when using the first 8 headers on a neural network with 5 hidden units. AUC of 0.98.

From these results we see that the performance is worse than in the original setting with 16 headers and more hidden units. However, it clearly shows that the learning task is relatively simple and even this simple network beats the logistic regression baseline classifier using 8 headers[47].

In order to scale the header-based approach to more services it is important to have enough training data to learn the representation of a new class. As generating new data is a very time consuming task, only a bare minimum of datapoints should be generated. From figure 5.36 we see that in our 7 class task, around 15000 datapoints allows for a decent accuracy. This gives just around 2000 datapoints from each class in order for a neural network to learn the representation.

[47] This result is shown in D.7.

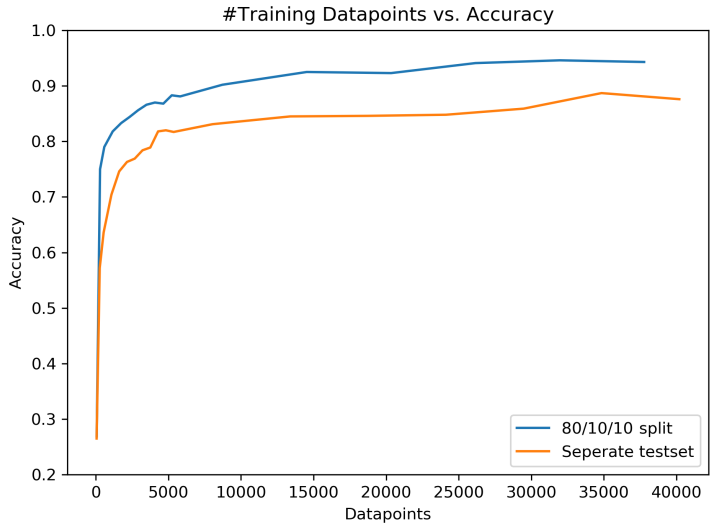


Figure 5.36: Number of datapoints in training set plotted against accuracy. Separate testset is WindowsAndreas.

5.2.7 Visualizing decisions

As the approach of using concatenated headers to perform classification of network traffic is a novel approach, it is essential to investigate what parts of the input are most relevant when the neural network makes a prediction. In order to avoid certain pitfalls such as classifying based on an IP-address or port number, we have anonymised each header as previously described. Therefore it is interesting to investigate the neural network which otherwise appears to be a "black box". For this task, the Layer-Wise Relevance Propagation (LRP) is very useful. It is particularly useful when dealing with images, as the heatmaps show what part of an image is relevant in making a prediction. Since our input vector is not an image, we have organized the input in such a way that the headers have been "stacked". The y-axis of the heatmap is the header number [0..15] in the session, and the x-axis is the byte number [0..53] in a given header. In appendix F a full list of what the individual bytes of a header represent can be found, which is useful for translating the generated heatmaps. The heatmaps shown in this section are derived from our 16 concatenated headers dataset, thus the dimensions of the plot will be 16 rows by 54 columns. The neural network used in making these heatmaps is our original model with a single hidden layer comprised of 50 hidden units having a ReLU activation function. The relevance heatmaps are created by showing the network a correctly predicted datapoint

and run a forward pass followed by the LRP algorithm, this will propagate the relevance back through the layers until it reaches the input. The input is reshaped and coloured such that darker areas are the most relevant. Since the nature of LRP is that relevance calculated for a single example and not averaged over a class, the examples shown here and in appendix C have been chosen by plotting 20 correctly predicted datapoints of each class, from which a representative example has been selected. That is, an example where at least half of the plots looked nearly identical. The sensitivity heatmaps have been created by showing the same datapoint as the LRP, but rather than running LRP, the gradients are propagated back through the network as described in section 2.2.9. We will not display relevance plots for all classes, but only for those which we found most interesting.

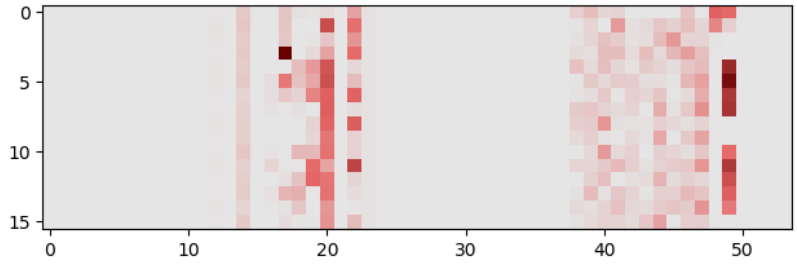


Figure 5.37: Relevance plot of representative sample from "drtv" class.

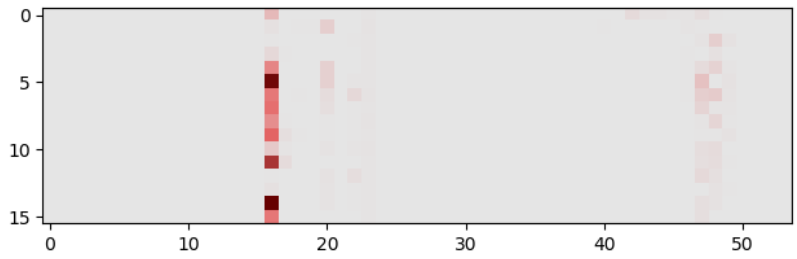


Figure 5.38: Sensitivity plot of representative sample from "drtv" class.

Figure 5.37 shows a relevance heatmap of a *drtv* datapoint comprised of 16 headers, while figure 5.38 shows a sensitivity heatmap for the same datapoint. DRTV streaming is served via HTTPS and thus is a TCP session with TLS. In both heatmaps we see a lot of grey fields which are fields, such as MAC-, IP-address and port-numbers, that have been anonymised before training. In

figure 5.37 we see in the fourth header that byte number 17 is very relevant in predicting that this is a *drtv* example. The fourth header in a session using TCP with TLS would typically belong to the ClientHello message sent from the client to the server, unless the TCP three-way handshake has failed. Byte number 17 contains the least significant bits of the *length* field in the IP header, which might be used to distinguish it from other datapoints, as the length of the ClientHello message might be shorter than the corresponding packet in a session not using TLS, such as a *http* session. Furthermore we see that especially byte number 49, which is the TCP *window size*, is relevant from the ACK message after the ClientHello message, and then through all the headers belonging to the TLS four-way handshake. In general we see that a lot of fields get some amount of relevance, however a field such as the *type* field in the Ethernet header (byte number 12 and 13) does not get any relevance, which is expected as no datapoints differ in that field, hence it is not relevant to discriminate between the individual classes. Figure 5.38 shows that in general the network is very sensitive towards changes in byte number 16 which is the most significant bits of the total length field of the IP header.

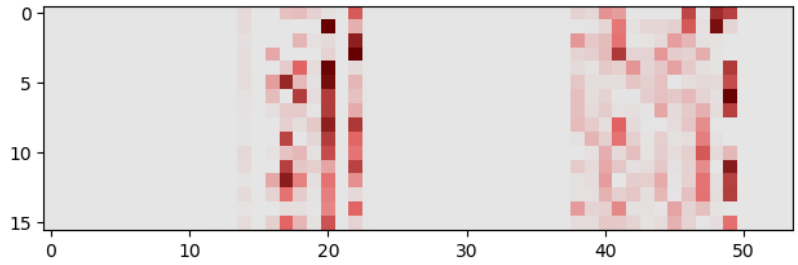


Figure 5.39: Relevance plot of representative sample from "https" class.

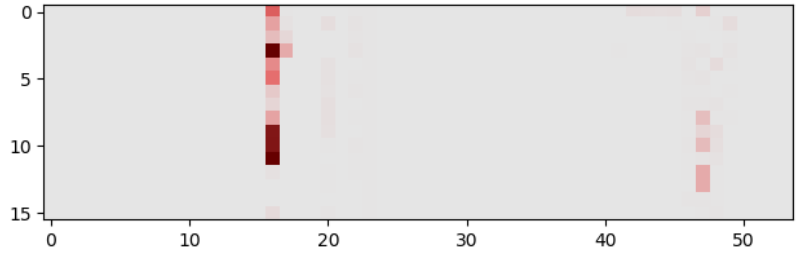


Figure 5.40: Sensitivity plot of representative sample from "https" class.

Figure 5.39 shows a relevance heatmap of the *https* class, which is regular web browsing served via TCP with TLS, and as such is served much in the same way as the previous *drtv* sample. However, the heatmap of this class resembles something different than figure 5.37. From header number 5, which in a TCP with TLS session is part of the TLS handshake, there is a lot of relevance on byte number 20, in which the first three bits are fragmentation flags of the IP header. As the data extraction process discarded fragmented sessions we know that it has to be the *don't fragment* flag that is responsible for the relevance shown[48]. It might be because the *don't fragment* flag is set to indicate to intermediate routers not to fragment the IP payload. Another possibility is that the flag is not explicitly set and the session is not fragmented. Furthermore the *time-to-live* field of the IP header (byte number 22) is showing a lot of relevance during the TCP handshaking and again after the TLS handshake has been completed. As with the previous *drtv* example the heatmap show a lot of relevance in the TCP Window Size (byte number 49) but in the *https* case already in the first and second header of the datapoint, as well as later in the session. It is interesting to see that the *Total Length* field of the IP header (byte number 17) also seems to be more relevant than in the case of *drtv*, especially in the last part of the session. This might be due to the payload length of regular web browsing not necessarily is as long as that of a streaming session, since video and audio in general takes more space than the text that make up a website. Figure 5.40 shows a sensitivity heatmap of the same *https* datapoint, but just as with the previous *drtv* example we see that the network is very sensitive to changes in byte number 16. When comparing figure 5.38 and 5.40 we see that the *https* example shows sensitivity to the total length earlier in a session than the *drtv* example does.

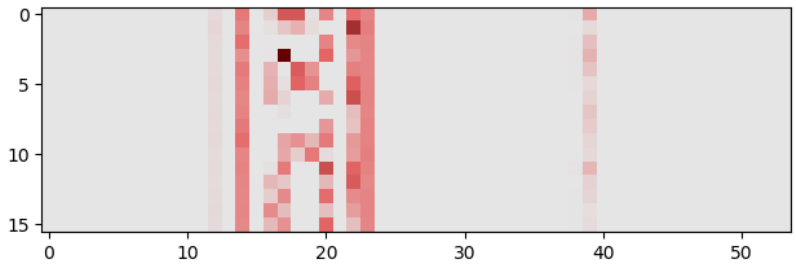


Figure 5.41: Relevance plot of representative sample from "youtube" class served via UDP.

[48] As shown in 3.3, byte number 20 consist of a *zero* followed by the *don't fragment* and *more fragments* flags, while the remainder is part of the *fragment offset*. As we discard all fragmented sessions it is only the *don't fragment* flag that can take on different values.

Figure 5.41 shows a relevance heatmap from the *youtube* class served via the QUIC protocol. From figure 5.41 it is easy to see that QUIC is a UDP-based protocol from the fact that the network puts high relevance on byte number 23, which is the *protocol* field of the IP header. The value of the field will be either 6 (TCP) or as in this case 17 (UDP). Furthermore we see that byte number 14 (IP version and header length) show signs of relevance in all the headers. This is peculiar as it is very likely to be the same among all of the datapoints, namely 0100 0101 meaning IP version 4 and a header length of 5 meaning 20 bytes ($5 \cdot 16$bit words). Figure 5.42 shows a sensitivity heatmap for the same datapoint where the network again seem to be sensitive to byte number 16, but also shows some sensitivity to byte number 23 which is understandable since this is a *youtube* point where UDP can be present.

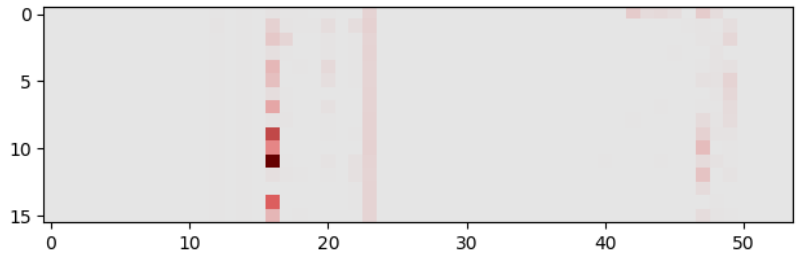


Figure 5.42: Sensitivity plot of representative sample from "youtube" class.

The visualizations shown in this section makes it clear that our model learns the representation of the dataset based on different fields of the concatenated headers. Furthermore we see from the sensitivity plots that in general the model is sensitive to changes in the first part of the total length field of the IP header. It is however not clear why the model puts relevance on fields that should be the same across all datapoints, as such the neural network is still somewhat of a black box, albeit LRP and sensitivity heatmaps provides some insights and confirms that the model learns a representation in which the whole datapoint is used in making a prediction and not a single field is responsible for all the relevance in the prediction.

Chapter 6

Discussion

> The aim of argument, or of discussion, should not be victory, but progress.
>
> *Joseph Joubert*

In the previous chapter we presented the experiments and the results of classifying network traffic, with special emphasis on distinguishing between streaming and non-streaming.

In this chapter we will discuss and reflect on the results and findings. Furthermore we will discuss some of the choices involved in the process of starting a machine learning project from having no data to a proof-of-concept model.

In the first part of the chapter we will discuss the different aspects and some potential pitfalls when creating a dataset and elaborate on some aspects of the related process. In the second part of the chapter we will discuss some of the choices made regarding the machine learning model.

6.1 Dataset discussion

A machine learning project is only as good as the data that is available. A major challenge of this project has been that no dataset was readily available which satisfied our needs. As the overall task was to investigate whether or not it was possible to classify network traffic such that streaming could be distinguished from other types of traffic, we needed to create our own dataset that could serve this purpose.

6.1.1 Choice of classes

The first task was to choose the classes that should be included in the dataset. We chose to include five different streaming services that are all commonly known in Denmark. To obtain diversity within the data we chose streaming services of different kinds, i.e. Netflix, HBO and DR TV contains longer shows, where Twitch and YouTube can have both short, long and live shows. We chose the largest streaming services that we know of, as we assumed that a lot of the traffic Napatech would like to identify would come from one of those services. To include network traffic that is not streaming, we chose to create two classes called *http* and *https* that contain network traffic transmitted by use of the HTTP and HTTPS protocols. We chose to include both encrypted and unencrypted web browsing as classes. Encrypted because it was important to investigate whether or not it was possible to distinguish between encrypted streaming and encrypted non-streaming. Unencrypted because it accounts for a large fraction of the internet traffic. In retrospect having the classes *hbo* and *netflix* makes the process of capturing data much slower as we wanted to obtain balanced classes. HBO and Netflix requires a login, and they are thereby able to impose a limit of one stream at a time to a user. YouTube, Twitch, DR TV does not impose such limits and thus can stream in ten or more tabs at the time. The consequence of this is that it takes 10 times longer to collect the same amount of Netflix and HBO sessions as it does for one of the other classes. For all classes different nuances exist, e.g all streaming services offer streaming in different resolutions, effectively changing how the datapoints might look. We know from Dubin et al. [7] that it is possible to classify the resolution of a streaming service, which leads us to believe that the datapoint associated with different resolutions will be situated in different places on the manifold of the true distribution. For the classes *http* and *https* different types of browsing exist, from light browsing to heavy browsing on webpages that include a lot of additional resources e.g JavaScript files, stylesheets and pictures etc. In our case only light browsing is included in the dataset by use of the PyTgen tool. We expect that a file download would be represented differently than light browsing.

6.1.2 Advertisements

When generating streaming data, some of the major platforms such as YouTube and Twitch display ads in the first part of the video clip. These ads are not hard-coded in the video, and as such are served by another server, before the actual video stream starts. This means that potentially a fraction of the YouTube and Twitch datapoints are actually advertisements and not the video stream served by the service of interest. The advertisements are typically also video/audio streams, and as such we can potentially have a datapoint labelled *youtube* that is in fact an ad, which potentially is not served by YouTube itself but rather another ad delivery service. However due to acceptable performance by our model on the generated dataset we have not investigated further how big of an issue this is, and whether this is an issue at all. One could argue that in a streaming/non-streaming context, these advertisements should also be classified as streaming.

6.1.3 Choice of platform

In order to generate the dataset within the timeframe of the project, a number of restrictions to the dataset has been made. Only IPv4 traffic was recorded since IPv4 traffic is what the majority of network traffic consist of.

In order to generate the web browsing traffic we chose to use URLLIB3 with the help of PyTgen to generate the traffic, and not by use of Selenium, since PyTgen can generate a lot of traffic very fast. We justify this approach since URLLIB3 makes a GET request to a given URL which is the same as what a browser would do, but without the overhead of a GUI.

As for the choice of operating systems and browsers, we initially started by generating traffic on our development machine which ran Ubuntu 17.10. As Google Chrome is by far the most used browser on the internet according to StatCounter [15] it seemed like the obvious choice for creating a dataset (LinuxChrome). As for our first additional dataset (WindowsFirefox), Microsoft Windows 10 and Mozilla Firefox Quantum was the most contrasting setup we could think of, since both the operating system and the browser are completely different. By doing so we implicitly chose to exclude the use of UDP in the form of YouTube via QUIC in the dataset. The LinuxChrome dataset contains YouTube streams served over both UDP and TCP as the initial traffic generation was done with default settings in Selenium, in which QUIC is not enabled. It becomes apparent that in order to create a robust dataset, multiple OS' and browsers must be included as our experiments show that a dataset created on a

Linux PC does not generalize to data generated by a Windows PC. In order for this project to leave the proof-of-concept stage, investigations should be made as to what kind of network traffic can be expected once the model is deployed.

6.1.4 Header dataset

The header dataset has been created for ease of use and interpretation, as such each datapoint contains a lot of zeros since each header has been anonymised. By anonymising headers we set 28 bytes to zero out of the 54/42 bytes available in a single TCP/UDP header. This leads to a sparse input vector. In order to minimize the amount of trainable parameters an approach could have been to extract the fields of the headers that have not been set to zero. Furthermore some fields can be left out entirely such as the entire data-link layer header, since the MAC addresses have been anonymised and the *type* field show little to no relevance in the relevance plots shown in section 5.2.7. Since the header of an IP packet is variable in size from 20 to 60 bytes, important information could be stored in the optional 40 bytes. By checking byte number 14 (Version and IP header length) of each packet, we have made sure that our dataset does not contain any optional IP header data, which could have pushed the information of the TCP/UDP header out of the datapoint, as we only extract the first 54 bytes of a packet. It should however be noted that it is possible that the options of the IP header could be used in classification. Because the options field is not typically used and the performance of the model was acceptable, further investigation has not been carried out. Due to the nature of network traffic, transferred over the internet, there is no guarantees that the packets arrive in the correct order or arrive at all. TCP tries to provide some of these guarantees. but that is only seen from a higher layer in the OSI model perspective. When capturing the raw network traffic the out-of-order packet, retransmits etc. are all captured. Since each datapoint consist of concatenated headers of a session, some of these errors could be present. We have however not checked for this type of error in the datapoints as we believe that it merely reflects the real world scenario.

6.1.5 Type of data

The dataset is only created by looking at regular Ethernet traffic. Cellular network traffic works in a very different way than Ethernet in which more fragmentation is seen because the different technologies might have different MTU[49]

[49] Maximum Transmission Unit.

sizes. In order to make a truly robust dataset, different types of network traffic should be included, but since data capturing is very time consuming this should only be considered if deemed necessary.

6.1.6 Variety in each class

When generating the dataset we know that the *http* and *https* classes have been generated from a limited selection of URLs which are shown in appendix B. Ideally, all datapoints should have been generated using a different URL, but that would have been an enormous task in itself. Furthermore we see from the experiments, especially figure 5.17 that it does not seem as if the classes *http* and *https* are the biggest issue. In fact *http* and *https* have very good recall, suggesting that even though the URLs used in the training set (LC) differ from the URLs used in the test set (WF), we predict almost all of them correctly. When it comes to generating the streaming traffic, there is no guarantee that all datapoints are different video streams since the streaming generation framework has been devised to randomly select available content from the front page of the streaming service, e.g. HBO has a limited amount of content available from their front-page, thereby the same Star Wars movie might have been selected numerous times to create the *hbo* datapoints. Since we have shown that the header-based approach shows promising results, we are interested in the setup/negotiation phase of a session and not the actual contents of a streaming session, thus we do not believe that this an issue as the negotiation should not be affected by the chosen content.

6.2 Machine learning discussion

Within this project our focus has been on exploring how classification of network traffic could be solved as a supervised learning task using neural networks. The following will go through some of the important aspects for the solution and discuss some potential alternatives.

6.2.1 Recall/precision tradeoff

As mentioned in section 2.2.7, evaluating the performance of a machine learning model should be done in accordance with the type of problem that one is trying to solve. In section 2.2.7 we gave an example of a model, concerning the

detection of fraud in financial transactions, in which recall or precision can be of highest importance. Within the domain of this thesis, and in general for the task that Napatech is trying to overcome, it is of high importance that we do not mark a session as streaming if it is not. Classifying a non-streaming session as streaming would filter out the session and it will therefore not be included in the further processing phase. Assuming that streaming is the positive class of a binary classifier, the precision of that classifier can typically be increased at the cost of recall by introducing a threshold variable. The classifier will then only report a detection if the value $\hat{y} = P(y = 1|\mathbf{x})$ exceeds the threshold. This idea can be extended to a multiclass problem in which refusing to make a prediction would correspond to predicting the negative class in the binary classifier.

6.2.2 Temporal robustness

We have not tested the temporal robustness of the model due to the timeframe of this project, and the overhead of generating data. However Michael et al. [34] have tested a neural network model on network traffic generated one and three years later than the training data. Their results show that there is a drop in accuracy of $\approx 2 - 3$ percentage points after one year. After three years the drop is $\approx 9 - 10$ percentage points. This means that retraining on new data is required if decent performance is to be maintained. Another challenge of temporal character is that streaming services evolve and can potentially change architecture overnight. We saw this happen with DR TV in January 2018 where it changed from a Adobe Flash based solution to a HTML5/JS solution. This had a huge impact on how the network traffic looked, since the Flash based approach used regular unencrypted HTTP as application layer protocol, while the HTML/JS approach uses HTTPS. Currently there is no way to guard one self against such drastic changes. But retraining is needed in order to ensure good performance throughout the supported lifetime of a product.

6.2.3 Model size tradeoff

As this thesis works as an exploratory analysis whether or not it is possible to classify network traffic, in order to potentially deploy a machine learning model on a Napatech SmartNIC, it is necessary to take the complexity of the model into account. For our header-based approach we have shown that reasonable results are achievable by using the headers of the first 8-16 packets. This means that the packets have to be buffered on the network card before being, either discarded in the case of streaming or written to a disk. As a Napatech SmartNIC can work at up to 200 Gbps and can handle more than two million new flows

each second, and have millions of sessions at any given time[50] this leads to quite a lot of classifications each second and a need for a large enough buffer to store all the packets that have not been classified. Assuming one million sessions in a buffer, a maximum Ethernet frame size of 1518 Bytes (1514 + 4 CRC) results in the need of ≈1.5GB buffer-space per header used in a datapoint. Thus for the approach using the headers of the first 8 packets of a session, almost 12GB is needed as a buffer on the network card. Currently a Napatech 200 Gbps card is equipped with 12GB DDR4 RAM as buffer [37]. However, if only the relevant fields from the header need to be buffered, then instead of buffering up to 1518 bytes from each packet, at most 54 bytes from each packet will be buffered, effectively reducing the storage problem by up to a factor of 28.

Another tradeoff has to be made between the performance and the amount of trainable parameters. A prediction from a neural network is effectively multiplication of matrices and vectors. In that sense the smaller the network the better, in order to speed up classification. We have shown that reasonable results can be achieved by using as little as 5 units in the hidden layer of our neural network model. This small architecture combined with a non-sparse input vector only consisting of extracted relevant fields, could effectively lower the amount of trainable parameters to less than 1100, since the input vector would not be 432 bytes but rather $(54 - 28) \cdot 8 = 208$ bytes, at most. This small neural network architecture is supported by the fact that a simple logistic regression classifier is able to achieve almost similar results, thus it seems as if the classes in our setting are almost linearly separable.

6.2.4 Hyperparameter search

We have not used an extensive grid or random search over hyperparameters. This is due to the fact that this project serves as a proof-of-concept, and the performance of the model has been quite good from the start. As mentioned, we have looked into the number of units in the hidden layer, but not only in order to increase performance, but rather to investigate how small we could make the network while still maintaining acceptable performance.

6.2.5 Fixed input length

Our model requires a fixed input length effectively meaning that maybe more data than necessary have to be buffered from each session. One can easily

[50] Stated in personal communication with Alex Omø Agerholm.

imagine that some classes does not require 8 or 16 packets of a session in order to be able to make a classification with high accuracy. To counter this fixed input size an approach could be to create a recurrent neural network (RNN) model which takes one packet at a time and first makes a class prediction once a certain probability threshold has been reached. By using this approach the fixed size input is no longer needed. However multiple parallel RNNs, one for each session, would have to be deployed effectively making this unfeasible with the throughput a Napatech SmartNIC handles. A more feasible approach while still accepting various discrete input lengths, could be to train 16 different models, based on our simple model, one which takes a single header as input, another that takes the first two headers as input etc. Then once a prediction from one of the models surpasses a certain confidence threshold the classification is made. However this does require that the model calculations are able to keep up with the throughput, as there is a risk that headers from the same session are input to 16 different models effectively slowing the classification.

6.2.6 Semi-supervised approach

Striving to solve any machine learning task as a supervised learning task includes the need of a labelled dataset that is large enough to learn the task. This is a common limitation for achieving good performance of the chosen machine learning algorithm [16]. Within this project, a rather limited number of relevant classes, representing both streaming and non-streaming network traffic was chosen in order to make the data generating process achievable. Erman et al. [9] demonstrate how a semi-supervised approach can cope with this limitation. A semi-supervised approach would allow one to capture network traffic from a larger number of applications and only label a few sessions belonging to each detected class[51]. This approach would also benefit from being able to handle new classes, in contrast to a supervised learning solution. The model in a supervised classification task maps an input $\mathbf{x}$ to one of K possible classes. If an input belongs to a class i and $i \notin K$, the model would still attempt to map $\mathbf{x}$ to one of the K classes, which is obviously wrong. A semi-supervised learning approach could leave the newly detected class as unlabelled.

[51] It might be that only a few sessions of the most recurring classes are labelled. Classes with very little occurrence within the network is of lesser importance.

Chapter 7

Conclusion

> The strongest arguments prove nothing so long as the conclusions are not verified by experience. Experimental science is the queen of sciences and the goal of all speculation.
>
> *Roger Bacon*

Throughout this project we have investigated a solution to network traffic classification using neural networks. The problem posed by Napatech is of high relevance as the increase in overall network traffic [25], calls for optimal solutions in order to obtain good network management. The classical approaches to network traffic classification are being challenged by the major focus on privacy and security in network communication, e.g. this is somewhat achievable by use of protocols that utilize encryption, which hinder an approach like DPI. Aiming to solve the problem as a supervised learning task made the data generation process very demanding and it has therefore been a large part of the work. The workflow for capturing and labelling the sessions has been optimized such that additional datasets have been and easily can be generated. Expanding the solution to a larger scale in which more services would be considered by the classifier might, as previously discussed, benefit from turning towards a

semi-supervised learning task, as this could ease the time-consuming process of creating a dataset. Having captured and created a structured dataset, we have examined two different approaches of learning a solution to the problem by use of neural networks. The payload-based approach and the header-based approach. As shown in section 5.1, a payload-based approach is highly doubtable due to encryption. We demonstrate by use of both t-SNE, PCA and other plots of the dataset, that we cannot distinguish between streaming and non-streaming. A conclusion which is also reflected in the classification experiments using a neural network.

On the other hand, the header-based approach has shown very promising results for the classification task. The solution use the unencrypted header of the network packets and concatenates the first 16 in a session. By use of t-SNE we have shown how the classes within the generated dataset cluster, supporting the claim of being feasible to solve as a classification task. For this task, the rather simple neural network model having one hidden layer with 50 hidden units yielded an accuracy of 96.4% when distinguishing between all applications (classes) recorded in the created dataset. This corresponds to an AUC close to 1.00 for the individual classes. For the streaming/non-streaming task, the accuracy was 99.9%. Having yielded such promising results, lead us to investigate which part of the input that the classifier give most attention to when making its prediction for each of the classes. This revealed that our model base its predictions on multiple parts of the input, i.e. fields like *window size*, *time-to-live* and fragmentation flags are relevant for most classes. An important finding within this project has been the impact of the client's operating system on the class representation. This has been revealed by creating additional datasets in which parameters such as ISP, browser and OS were changed. For the solution to work on Napatech's SmartNIC, it is therefore important to include this variation when creating the dataset for training a model. As there exist a finite number of operating systems and Windows and MacOS make up for the largest part of desktop generated network traffic [14], this is not a problem for the scalability of the solution. The thesis shows that classification of network traffic is possible based on a solution that utilize the unencrypted headers of network packets.

7.1 Further Work

Due to the limited timeframe of this project, many experiments, adaptions and tests have been left for future work. Some of these are, including more classes within the dataset and/or look into a semi-supervised approach in order to lighten the amount of work needed to label the traffic in a controlled environment.

Furthermore the entire hardware aspect needs to be researched in more detail, to ensure that the classification process can be implemented to run fast enough, for it to be of any practical use. Thus we would like to explore in detail the buffer vs. model size tradeoff. With the implementation in hardware also comes considerations on temporal robustness, which needs to be examined in order to figure out how often the model should be retrained, and if this is feasible with the amount of work needed to generate new network traffic.

Appendix A

GitHub Repository

`https://github.com/SalikLP/classification-of-encrypted-traffic` is the link to our public GitHub repository which contains the code created in the different stages of this thesis. The code is made open source under the MIT licence which is a short, permissive software license. This means that as long as you include the original copyright and licence notice in any copy of the software/-source, you can do whatever you want. [47].

Appendix B

HTTP and HTTPS websites

This appendix contains the websites used by PyTgen in order to generate http and https network traffic.

B.1 Linux/Chrome

The websites listed below were the top results where the page is prefixed with *HTTP* or *HTTPS* respectively when searching for *HTTP* and *HTTPS* on `https://www.google.dk/` using our Linux machine with the Google Chrome browser.

B.1.1 HTTP

- `http://naturstyrelsen.dk/`
- `http://www.valutakurser.dk/`
- `http://ordnet.dk/ddo/forside`
- `http://www.speedtest.net/`
- `http://bygningsreglementet.dk/`
- `http://www.ft.dk/`
- `http://tv2.dk/`
- `http://www.kl.dk/`
- `http://www.symbiosis.dk/`
- `http://www.noegletal.dk/`

- http://novonordiskfonden.dk/da
- http://frida.fooddata.dk/
- http://www.arbejdsmiljoforskning.dk/da
- http://www.su.dk/
- http://www.trafikstyrelsen.dk/da.aspx
- http://www.regioner.dk/
- http://www.geus.dk/UK/Pages/default.aspx
- http://bm.dk/
- http://www.m.dk/#!/
- http://www.regionsjaelland.dk/Sider/default.aspx
- http://www.trafikstyrelsen.dk/da.aspx

B.1.2 HTTPS

- https://www.dr.dk/
- https://da.wikipedia.org/wiki/Forside
- https://en.wikipedia.org/wiki/Main_Page
- https://www.dk-hostmaster.dk/
- https://www.cph.dk/
- https://translate.google.com/
- https://www.borger.dk/
- https://www.sdu.dk/da/
- https://www.sundhed.dk/
- https://www.facebook.com/
- https://www.ug.dk/
- https://erhvervsstyrelsen.dk/
- https://www.nets.eu/dk-da
- https://www.jobindex.dk/
- https://www.rejseplanen.dk/webapp/index.html
- https://yousee.dk/
- https://www.sparnord.dk/
- https://gigahost.dk/
- https://www.information.dk/
- https://stps.dk/
- https://www.skat.dk/
- https://danskebank.dk/privat
- https://www.sst.dk/

B.2 Windows/Firefox

The websites listed below were the top results where the page is prefixed with *HTTP* or *HTTPS* respectively when searching for *HTTP* and *HTTPS* on https://www.google.dk/ using our Windows machine with the Mozilla Firefox browser

B.2.1 HTTP

- http://www2.hm.com/da_dk/index.html
- http://www.euroinvestor.dk/
- http://tv2.dk/
- http://vejr.tv2.dk/
- http://bygningsreglementet.dk/
- http://litteraturlisteautomaten.dk/
- http://www.su.dk/
- http://www.regionsjaelland.dk/Sider/default.aspx
- http://live.rideforbund.dk/

- http://engelsk.gyldendal.dk/
- http://www.trafikstyrelsen.dk/da.aspx
- http://www.regioner.dk/
- http://denstoredanske.dk/
- http://library.au.dk/
- http://www.kl.dk/
- http://www.windpower.org/
- http://cdon.dk/
- http://ordnet.dk/ddo/forside
- http://nyheder.tv2.dk/
- http://ni.dk/
- http://ddd.dda.dk/
- http://www.ft.dk/

B.2.2 HTTPS

- https://da.wikipedia.org/wiki/HTTPS
- https://en.wikipedia.org/wiki/HTTPS
- https://www.instantssl.com/ssl-certificate-products/https.html
- https://www.grammatip.com/
- https://meebook.com/
- https://www.netproever.dk/
- https://da-dk.facebook.com/
- https://profil.yousee.dk/
- https://www.minuddannelse.net/Home/Forside#ATHS
- https://translate.google.com/?hl=da
- https://www.borger.dk/
- https://sdfekort.dk/spatialmap
- https://ordbog.gyldendal.dk/
- https://ucpraktikportal.dk
- https://www.dr.dk/tv/live/dr1
- https://scholar.google.dk/
- https://minlaering.dk/
- https://webtv.stofa.dk/
- https://hbdansk.systime.dk/
- https://danskebank.dk/privat
- https://www.sdu.dk/da

Appendix C

Relevance plots

This appendix contains some representative LRP- and sensitivity plots

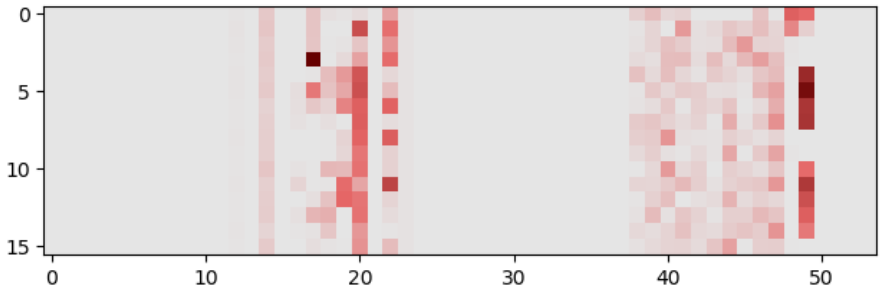


Figure C.1: Relevance plot of representative sample from *drtv* class using LRP

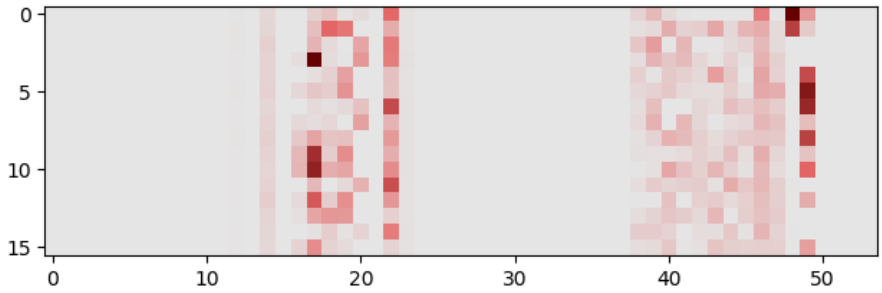


Figure C.2: Relevance plot of representative sample from *hbo* class using LRP

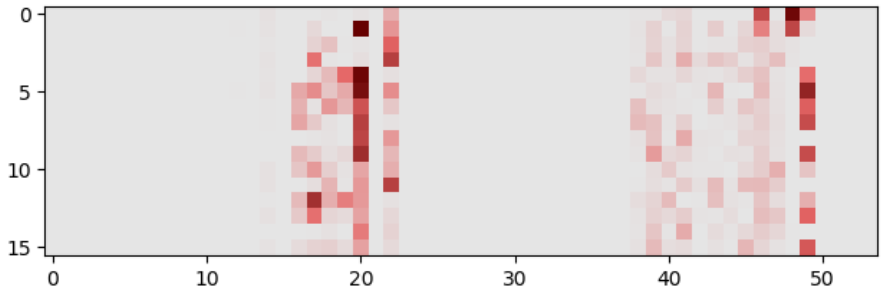


Figure C.3: Relevance plot of representative sample from *http* class using LRP

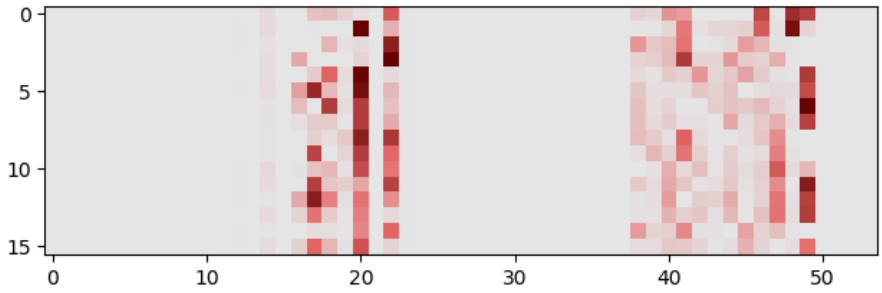


Figure C.4: Relevance plot of representative sample from *https* class using LRP

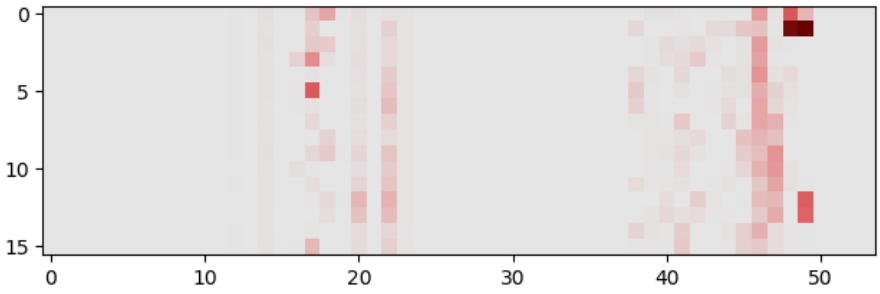


Figure C.5: Relevance plot of representative sample from *netflix* class using LRP

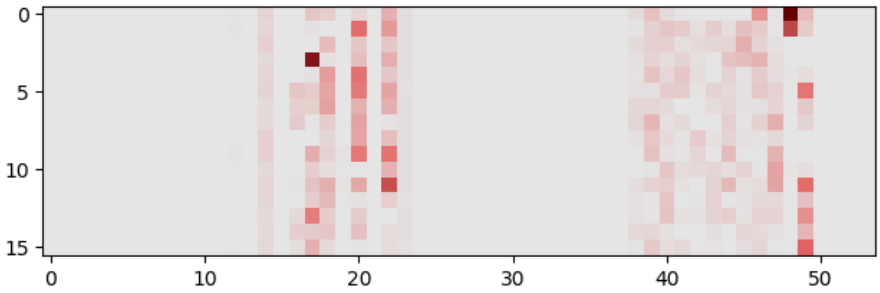


Figure C.6: Relevance plot of representative sample from *twitch* class using LRP

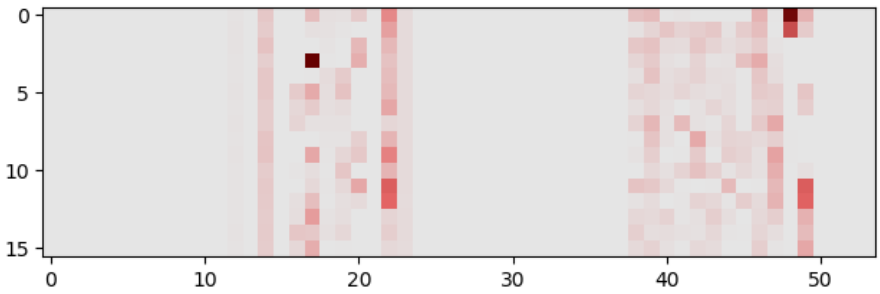


Figure C.7: Relevance plot of representative sample from *youtube* class served via TCP using LRP

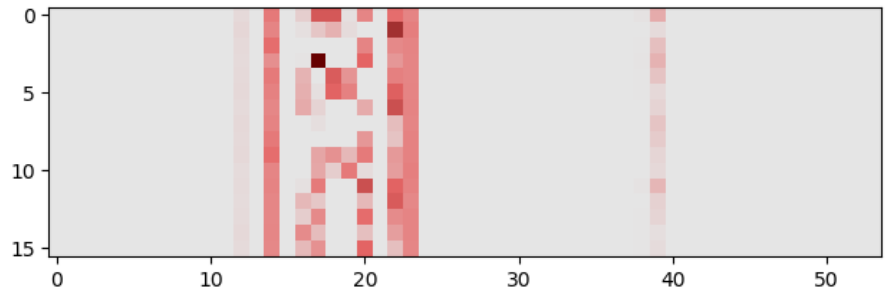


Figure C.8: Relevance plot of representative sample from *youtube* class served via UDP using LRP

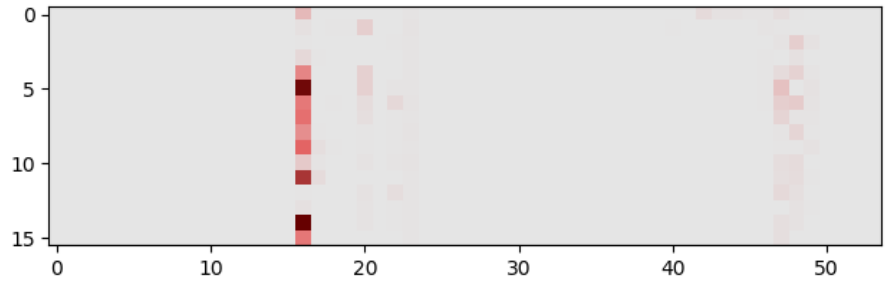


Figure C.9: Sensitivity plot of representative sample from *drtv* class

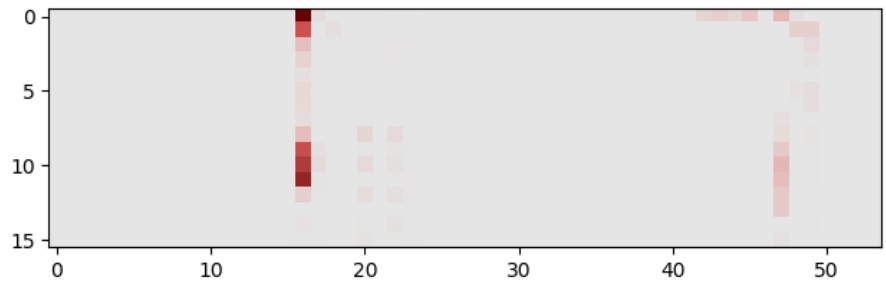


Figure C.10: Sensitivity plot of representative sample from *hbo* class

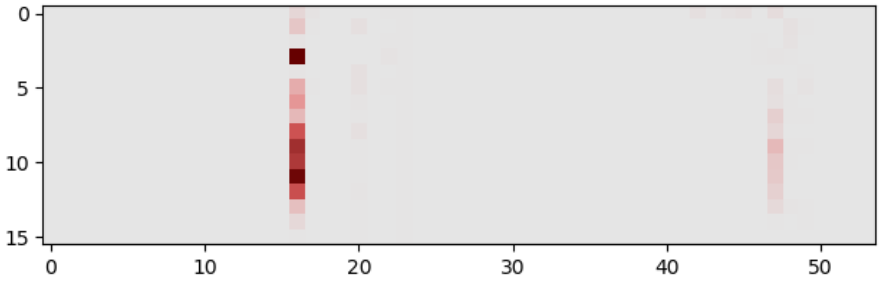


Figure C.11: Sensitivity plot of representative sample from *http* class

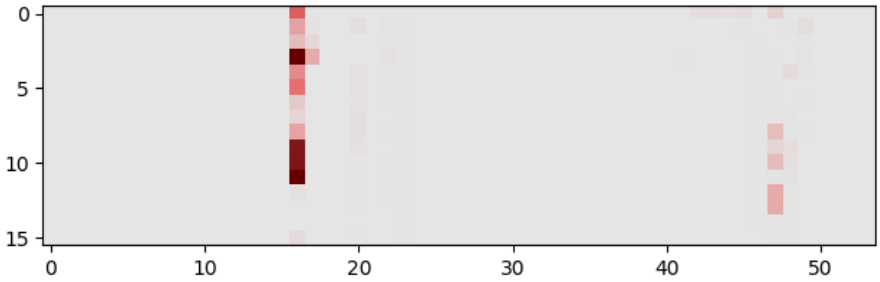


Figure C.12: Sensitivity plot of representative sample from *https* class

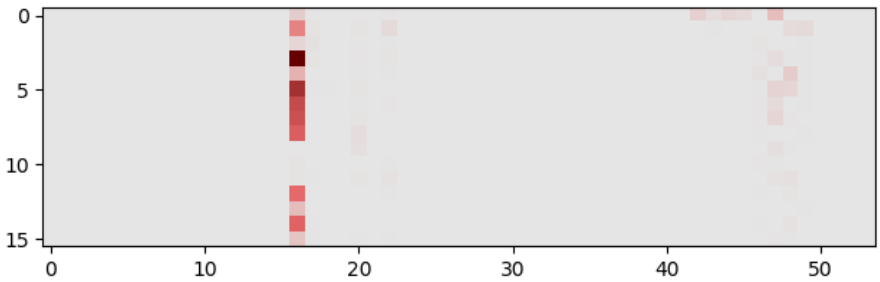


Figure C.13: Sensitivity plot of representative sample from *netflix* class

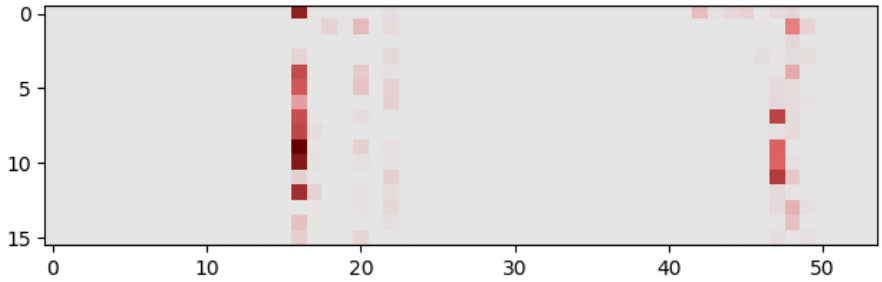


Figure C.14: Sensitivity plot of representative sample from *twitch* class

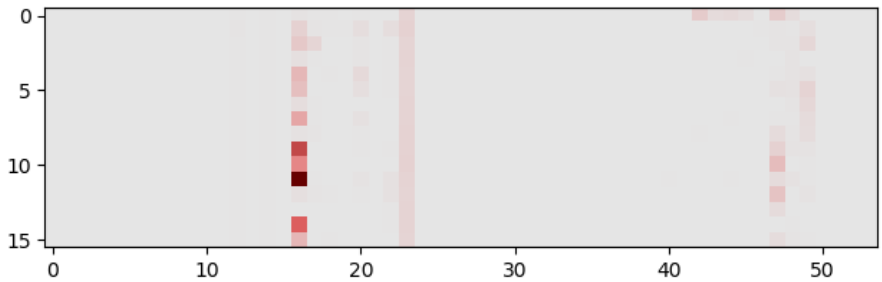


Figure C.15: Sensitivity plot of representative sample from *youtube* class

Appendix D

Header Plots

This appendix contains additional plots for the header experiments made on the 16 and 8 header datasets.

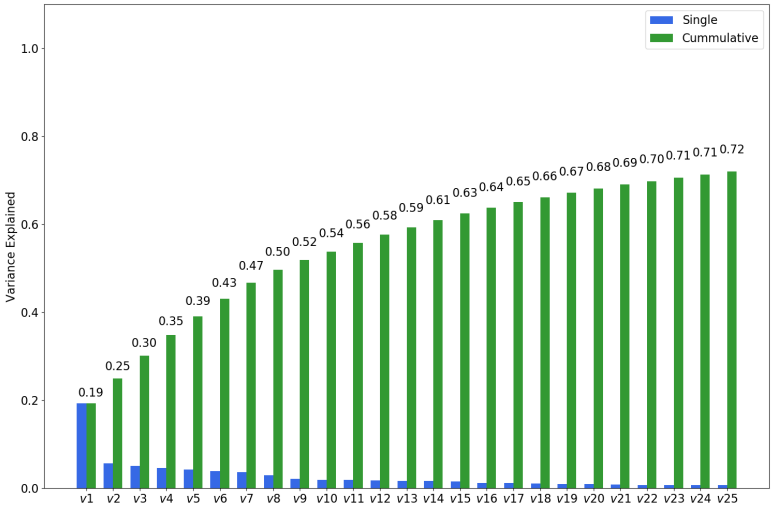


Figure D.1: Cumulative plot of the first 25 principal components when performing PCA on the merged dataset of the first 16 headers.

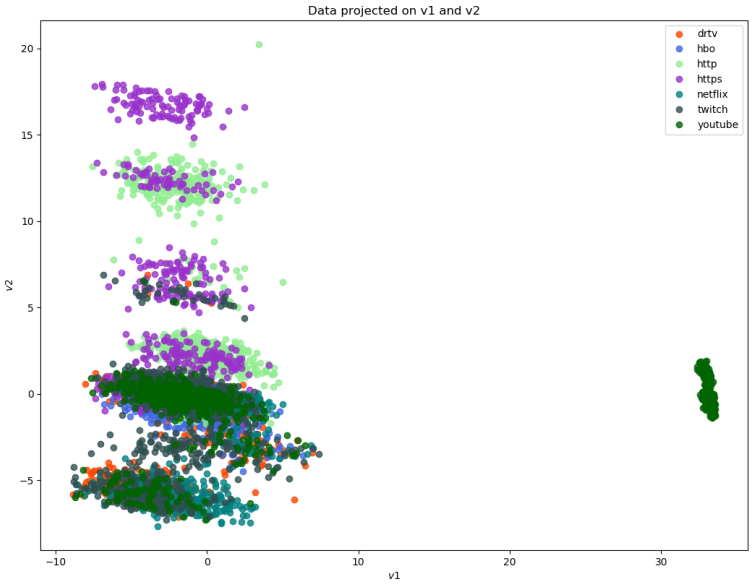


Figure D.2: Plot of merged dataset of first 16 headers projected onto first two principal components

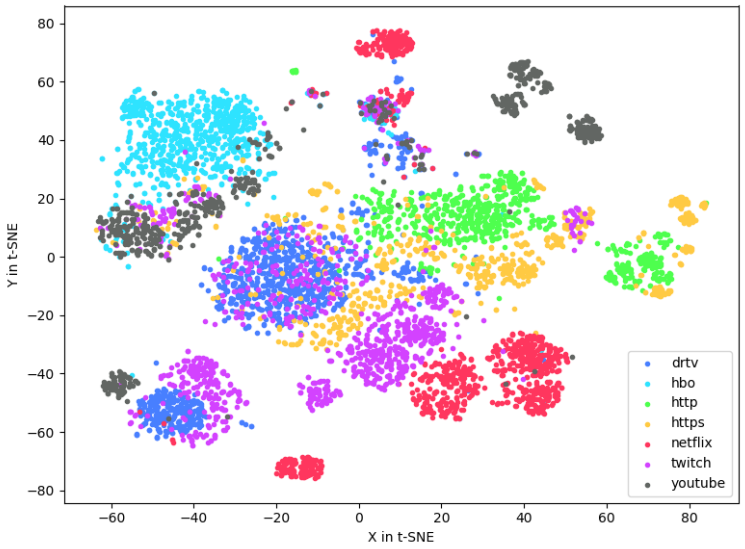


Figure D.3: t-SNE plot with perplexity 30 over 1000 iterations of first 16 headers from the full dataset

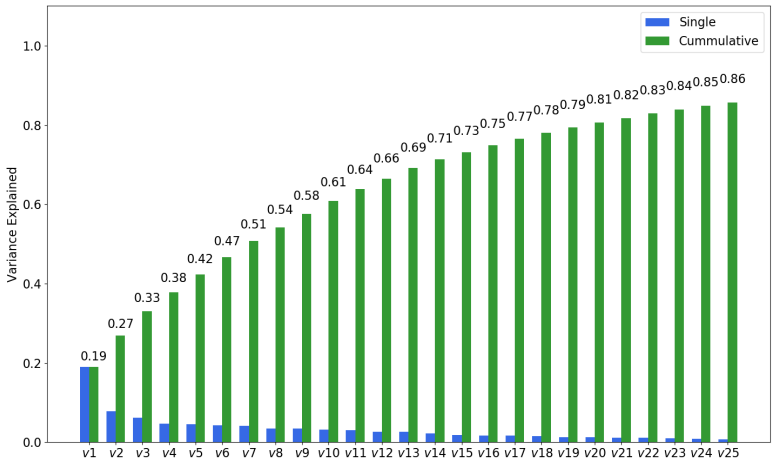


Figure D.4: Cumulative plot of the first 25 principal components when performing PCA on the merged dataset of the first 8 headers.

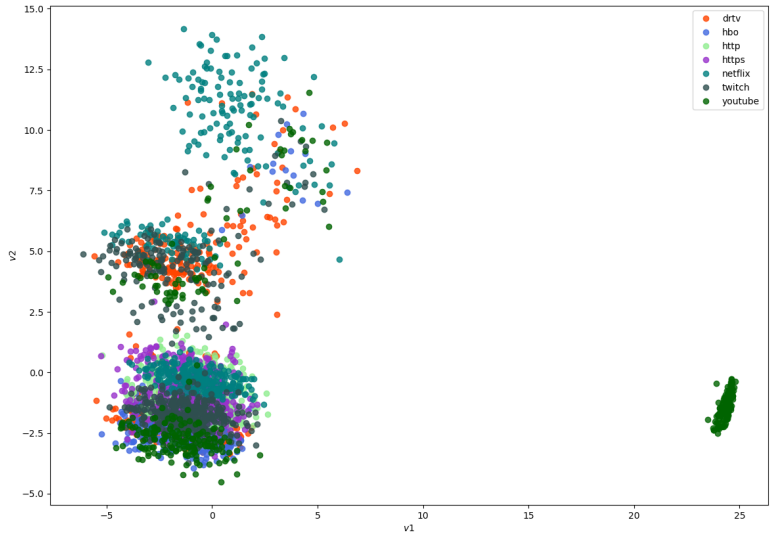


Figure D.5: Plot of merged dataset of first 8 headers projected onto first two principal components

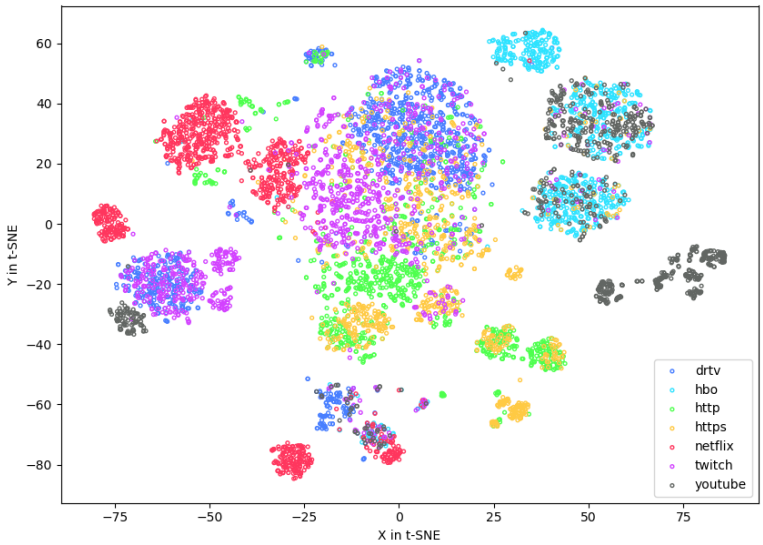


Figure D.6: t-SNE plot with perplexity 30 over 1000 iterations of first 8 headers from the full dataset

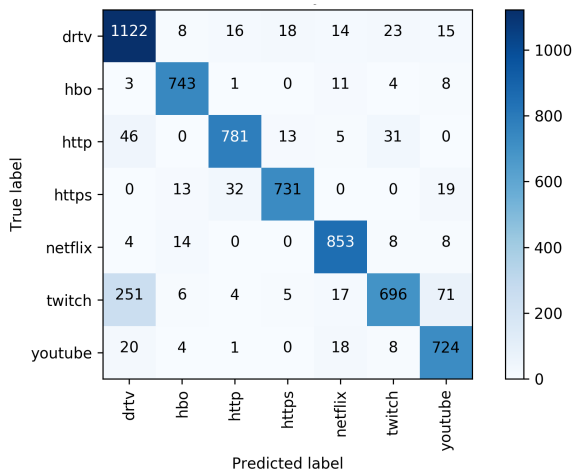


Figure D.7: Confusion matrix showing the results of a logistic regression model using 8 headers.

Appendix E

PC Specifications

This appendix contains the specifications for the different PCs used to generate network traffic.

E.1 LinuxChrome

This PC is an ASUS ROG GT51CH Tower PC used for dataset generation and training the neural networks. It has the following specs:

- **CPU:** Intel Core i7-7700K
- **RAM:** 64 GB of DDR4
- **OS:** Ubuntu Linux version 17.10
- **Browser:** Google Chrome 64
- **Location**: Napatech A/S in Søborg
- **GFX:** 2x NVIDIA GTX 1080 Ti (SLI)
- **NIC:** Intel Ethernet Connection I219-V
- **TShark:** version 2.4.2

E.2 WindowsFirefox

The PC is a Lenovo M93p used solely for dataset generation. It has the following specs:

- **CPU:** Intel Core i5-4570
- **RAM:** 16 GB of DDR3
- **OS:** Microsoft Windows 10 Professional
- **Browser:** Mozilla Firefox Quantum 59.0.2
- **Location**: Holbæk
- **GFX:** 1x NVIDIA GeForce 630 GT
- **NIC:** Intel(R) Ethernet Connection I217-LM
- **TShark:** version 2.4.6

E.3 WindowsChrome

The PC is the same Lenovo M93p as the WindowsFirefox one:

- **CPU:** Intel Core i5-4570
- **RAM:** 16 GB of DDR3
- **OS:** Microsoft Windows 10 Professional
- **Browser:** Google Chrome 65
- **Location**: Holbæk
- **GFX:** 1x NVIDIA GeForce 630 GT
- **NIC:** Intel® Ethernet Connection I217-LM
- **TShark:** version 2.4.6

E.4 WindowsAndreas

This is Andreas private laptop PC. An MMVision with the following specs:

- **CPU:** Intel Core i5-4200M
- **RAM:** 8GB of DDR3
- **OS:** Microsoft Windows 10 Home
- **Browser:** Google Chrome 64
- **Location**: Copenhagen
- **GFX:** NVIDIA GeForce GTX 765M
- **NIC:** Intel® Centrino® Advanced-N 6235
- **TShark:** version 2.4.6

E.5 WindowsSalik

This is Saliks private desktop PC, it is a DIY PC:

- **CPU:** Intel Core i7-4770K
- **RAM:** 24 GB of DDR3
- **OS:** Microsoft Windows 10 Professional
- **Browser:** Google Chrome 65
- **Location**: Holbæk
- **GFX:** 1x NVIDIA GTX 970
- **NIC:** Killer E2200 Gigabit Ethernet Controller
- **TShark:** version 2.4.6

Appendix F

Header bytes

This appendix contains an overview and a key to what the individual byte indices in a single header means.

Byte number	Protocol	Field
0	Ethernet	Destination MAC Address
1	Ethernet	Destination MAC Address
2	Ethernet	Destination MAC Address
3	Ethernet	Destination MAC Address
4	Ethernet	Destination MAC Address
5	Ethernet	Destination MAC Address
6	Ethernet	Source MAC Address
7	Ethernet	Source MAC Address
8	Ethernet	Source MAC Address
9	Ethernet	Source MAC Address
10	Ethernet	Source MAC Address
11	Ethernet	Source MAC Address
12	Ethernet	Ethernet type
13	Ethernet	Ethernet type
14	IP	IP version and header length
15	IP	Type Of Service
16	IP	Total length byte 1
17	IP	Total length byte 2

Byte number	Protocol	Field
18	IP	Identification byte 1
19	IP	Identification byte 2
20	IP	3 bit Fragment flags and 5 bits Fragment offset MSB
21	IP	Fragment offset LSB
22	IP	Time To Live
23	IP	Protocol
24	IP	Header checksum byte 1
25	IP	Header checksum byte 2
26	IP	Source IP address byte 1
27	IP	Source IP address byte 2
28	IP	Source IP address byte 3
29	IP	Source IP address byte 4
30	IP	Destination IP address byte 1
31	IP	Destination IP address byte 2
32	IP	Destination IP address byte 3
33	IP	Destination IP address byte 4
34	TCP/UDP	Source Port byte 1
35	TCP/UDP	Source Port byte 2
36	TCP/UDP	Destination Port byte 1
37	TCP/UDP	Destination Port byte 2
38	TCP	Sequence number byte 1
38	UDP	Length of data byte 1
39	TCP	Sequence number byte 2
39	UDP	Length of data byte 2
40	TCP	Sequence number byte 3
40	UDP	Checksum byte 1
41	TCP	Sequence number byte 4
41	UDP	Checksum byte 2
42	TCP	ACK number byte 1
43	TCP	ACK number byte 2
44	TCP	ACK number byte 3
45	TCP	ACK number byte 4
46	TCP	TCP Header length or nonce
47	TCP	TCP Flags (ACK, SYN, FIN ...)
48	TCP	Window Size byte 1
49	TCP	Window Size byte 2
50	TCP	Checksum byte 1
51	TCP	Checksum byte 2
52	TCP	Urgent Pointer byte 1
53	TCP	Urgent Pointer byte 2

Byte number	Protocol	Field

Table F.1: Table showing the meaning of each byte in a single header. Bytes 0-37 are the same between TCP and UDP headers, but bytes 38-41 are different between TCP and UDP. Bytes 42-53 are only present in a TCP header.

Appendix G

Cross-validation

This appendix contains the results of running 10-fold cross-validation on the 16 header merged dataset on a model with a single hidden layer with 50 units and a 7 unit softmax output.

Fold	Train acc	Test acc
1	0.969	0.960
2	0.971	0.963
3	0.971	0.967
4	0.970	0.968
5	0.973	0.964
6	0.972	0.962
7	0.970	0.969
8	0.971	0.962
9	0.967	0.962
10	0.967	0.967

Table G.1: Table showing the train and test accuracy of folds 1 to 10 when running 10 fold cross validation

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLOS ONE*, 10(7):1–46, 07 2015. doi: 10.1371/journal.pone.0130140. URL `https://doi.org/10.1371/journal.pone.0130140`.

[3] R. Bar Yanai, M. Langberg, D. Peleg, and L. Roditty. *Realtime Classification for Encrypted Traffic*, pages 373–385. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-13193-6. doi: 10.1007/978-3-642-13193-6_32. URL `https://doi.org/10.1007/978-3-642-13193-6_32`.

[4] Cisco VNI. Cisco Visual Networking Index: Forecast and Methodology, 2016–2021. `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html`, 2017. [Online; accessed 13-Jun-2018].

[5] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[6] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani. Characterization of encrypted and vpn traffic using time-related features. 2016.

[7] R. Dubin, A. Dvir, O. Pele, O. Hadar, I. Richman, and O. Trabelsi. Real time video quality representation classification of encrypted http adaptive video streaming-the case of safari. *arXiv preprint arXiv:1602.00489*, 2016.

[8] Ericsson. Internet of Things forecast. `https://www.ericsson.com/en/mobility-report/internet-of-things-forecast`, 2018. [Online; accessed 04-Jun-2018].

[9] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson. Offline/realtime traffic classification using semi-supervised learning. *Performance Evaluation*, 64(9-12):1194–1213, 2007.

[10] ESA. How many stars are there in the universe. `https://www.esa.int/Our_Activities/Space_Science/Herschel/How_many_stars_are_there_in_the_Universe`, 2018. [Online; accessed 01-Jun-2018].

[11] R. Frank. The perceptron a perceiving and recognizing automaton. *Cornell Aeronautical Laboratory, Buffalo, NY, USA, Tech. Rep*, pages 85–460, 1957.

[12] Gebhart, Gennie. We're Halfway to Encrypting the Entire Web. `https://www.eff.org/deeplinks/2017/02/were-halfway-encrypting-entire-web`, 2017. [Online; accessed 04-Jun-2018].

[13] G. D. Gil, A. H. Lashkari, M. Mamun, and A. A. Ghorbani. Characterization of encrypted and vpn traffic using time-related features. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 407–414, 2016.

[14] GlobalStats. Desktop Operating System Market Share Worldwide. `http://gs.statcounter.com/os-market-share/desktop/worldwide`, 2018. [Online; accessed 14-Jun-2018].

[15] GlobalStats. Browser Market Share Worldwide. `http://gs.statcounter.com/browser-market-share`, 2018. [Online; accessed 14-Jun-2018].

[16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[17] G. E. Hinton and S. T. Roweis. Stochastic neighbor embedding. In *Advances in neural information processing systems*, pages 857–864, 2003.

[18] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[19] J. Hurley, E. Garcia-Palacios, and S. Sezer. Classification of p2p and http using specific protocol characteristics. In *Meeting of the European Network of Universities and Companies in Information and Communication Engineering*, pages 31–40. Springer, 2009.

[20] IANA. Service name and transport protocol port number registry, 2018. URL `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml`.

[21] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[22] Karpathy, Andrej. CS231n Convolutional Neural Networks for Visual Recognition. `http://cs231n.github.io/neural-networks-2/#reg`, 2018. [Online; accessed 04-Jun-2018].

[23] H. Kim, K. C. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet traffic classification demystified: myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT conference*, page 11. ACM, 2008.

[24] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] Kipp, Scott. Exponential bandwidth growth and cost declines. `https://www.networkworld.com/article/2187538/tech-primers/exponential-bandwidth-growth-and-cost-declines.html`, 2018. [Online; accessed 04-Jun-2018].

[26] A. Langley and W.-T. Chang. QUIC crypto, 2013.

[27] A. H. Lashkari, A. F. A. Kadir, H. Gonzalez, K. F. Mbah, and A. A. Ghorbani. Towards a network-based framework for android malware detection and characterization. In *Proceeding of the 15th international conference on privacy, security and trust*, 2017.

[28] J. Lawrence, J. Malmsten, A. Rybka, D. A. Sabol, and K. Triplin. Comparing tensorflow deep learning performance using cpus, gpus, local pcs and cloud.

[29] Y. LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[30] Y. LeCun, L. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, et al. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective*, 261:276, 1995.

[31] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 1998.

[32] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[33] H. Mhaskar, Q. Liao, and T. A. Poggio. When and why are deep networks better than shallow ones? In *AAAI*, pages 2343–2349, 2017.

[34] A. K. J. Michael, E. Valla, N. S. Neggatu, and A. W. Moore. Network traffic classification via neural networks. Technical report, University of Cambridge, Computer Laboratory, 2017.

[35] G. Montavon, S. Lapuschkin, A. Binder, W. Samek, and K.-R. Müller. Explaining nonlinear classification decisions with deep taylor decomposition. *Pattern Recognition*, 65:211 – 222, 2017. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2016.11.008. URL `http://www.sciencedirect.com/science/article/pii/S0031320316303582`.

[36] G. Montavon, W. Samek, and K.-R. Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 2017.

[37] Napatech A/S. Data sheet - Napatech 200G Compact Solution NT200A01 2-PORT 100G. `https://www.napatech.com/support/resources/data-sheets/napatech-200g-compact-solution/`, 2018. [Online; accessed 23-May-2018].

[38] Napatech A/S. Napatech - About us. `https://www.napatech.com/about/`, 2018. [Online; accessed 11-Jun-2018].

[39] A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. 2015.

[40] V. K. Pang-Ning Tan, Michael Steinbach. *Introduction to data mining*. Pearson, 2014.

[41] K. Pearson. On lines and planes of closest fit to systems of points in space. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (SIGMOD)*, 1901.

[42] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[43] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proceedings of the 13th international conference on World Wide Web*, pages 512–521. ACM, 2004.

[44] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *Proceedings of fourth international conference on information systems security and privacy, ICISSP*, 2018.

[45] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[46] R. Stewart and S. Long. Improving high–bandwidth tls in the freebsd kernel. 2016.

[47] tl;drLegal. MIT Licence expalined in plain english, 2018. URL `https://tldrlegal.com/license/mit-license#summary`.

[48] M. N. S. Tue Herlau and M. Mørup. *Introduction to Machine Learning and Data Mining*. 2017. version 5.

[49] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang. End-to-end encrypted traffic classification with one-dimensional convolution neural networks. In *Intelligence and Security Informatics (ISI), 2017 IEEE International Conference on*, pages 43–48. IEEE, 2017.

[50] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng. Malware traffic classification using convolutional neural network for representation learning. In *2017 International Conference on Information Networking (ICOIN)*, pages 712–717, Jan 2017. doi: 10.1109/ICOIN.2017.7899588.

[51] M. Wattenberg, F. Viégas, and I. Johnson. How to use t-sne effectively. *Distill*, 2016. doi: 10.23915/distill.00002. URL `http://distill.pub/2016/misread-tsne`.

[52] J. M. Zurada, A. Malinowski, and I. Cloete. Sensitivity analysis for minimization of input data dimension for feedforward neural network. In *Circuits and Systems, 1994. ISCAS'94., 1994 IEEE International Symposium on*, volume 6, pages 447–450. IEEE, 1994.