Department of Informatics

Master Thesis

# Utilizing the SHAP framework to bypass intrusion detection systems

*Author: Jonas Mossin Wagle*
*Supervisor: Øyvind Ytrehus*

June 1, 2021

## Abstract

The number of people connected to the internet is swiftly growing, and technology is increasingly integrated into our daily lives. With this increase, there is a surge of attacks towards the digital infrastructure. It is of great importance to understand how we can analyze and mitigate attacks to ensure the availability of the services we depend on. The purpose of this study is two-sided. The first is to evaluate different machine learning models in intrusion detection systems. We measured their performance on distributed denial of service(DDoS) attacks and explained them using SHAP values. Secondly, by using the SHAP values, we found the most important features and generated multiple variations of the same attacks to see how the different models reacted. Ultimately, we found that SHAP values have great potential as a base for generating more sophisticated attacks. In turn, the modified attacks were able to bypass intrusion detection systems.

## Acknowledgements

First of all, I want to direct a special thanks to my supervisor Øyvind Ytrehus for helping me formulate my thesis and always being available to answer my questions. Your invaluable insight, encouragement, and feedback have brought my work to a higher level.

Secondly, I want to thank my wonderful parents Anette and Carl, for your unconditional support and always being there for me. I also want to direct a special thanks to my dear brother Max for the random calls, funny discussions, and the incredible support you have shown me.

Finally, I would also like to thank all my friends I have got to know during my time in Bergen, and my flatmates in Hans Holmboes 8, Henrik Hexeberg, Christopher Hærem, Anders Hopland, Håkon Osland, and Sondre Omland. Even though the last year has been challenging, together, we made our final year fantastic. I will miss living with you all.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

In the age of digitalization, technology is more available than ever. Most people would never leave their smartphones at home, and people are dependent on digital services every day. The number of people connected to the Internet increases rapidly. Almost every major company delivers some form of service through the web. Cisco estimates that over 70 percent of the global population will have mobile connectivity by 2023[1]. Following the large influx of new users, there is a proportional increase of competitive services and controversial information on the web.

Governments or corporations may choose an unethical way of dealing with competitors or censoring information they do not acknowledge. Hence, the potential for gain and influence is immense for the attackers. From Ciscos's report, there was a 776 percent increase in Distributed Denial of Service(DDoS, discussed in section 3.1.2) attacks with throughput between 100Gbps and 400Gbps from 2018 to 2019. They also expect that the total number of DDoS attacks going to double from 2018 to 2023[1]. As attacks advance and become more powerful and sophisticated, we need to find new ways of mitigating them efficiently.

## 1.2 Goal

This thesis aims to develop and evaluate different neural network architectures' performance and robustness on various Distributed Denial of Service attacks. While this is a well-researched area, there have not been enough reports on whether or not neural networks perform on variations of the same attacks. This thesis will use the Shapley Additive Explanations(SHAP) framework to describe the proposed models and generate SHAP values. Utilizing the SHAP values from the SHAP framework, we design variations of attacks attempting to fool the models. The primary goal is to determine if it is possible to generate attacks based on the output from the SHAP framework. A secondary goal is to evaluate the viability of different machine learning models in intrusion detection systems.

## 1.3 Overview

We structured this thesis in chapters, where the next chapter discusses the field of intrusion detection systems(IDS), relevant related work, and contributions of this thesis. In the third chapter, background, we discuss necessary background knowledge. Namely, what a DDoS attack is and specific attacks. We also discuss machine learning and neural networks, model explanation with LIME, Shapley values, and the SHAP framework. In the end, we propose a lite version IDS based on the acquired knowledge. Chapter 4 discusses the proposed machine learning models, the experiments carried out, and their results. Chapter 5 discusses the viability of SHAP values as a base for generating attacks, the performance of the machine learning models, how the model reacted to the modified attacks, and future work.

# Chapter 2

# Related work

Much time and effort are devoted to researching how to utilize machine learning in intrusion detection systems(IDS).

Jiyeon Kim et al. [2] proposed a supervised convolutional neural network(CNN) for network intrusion detection against denial of service attacks. They prove that CNN can perform well at detecting Denial of Service(DoS) attacks. However, the results on Distributed Denial of Service (DDoS) attacks were not as good. In their discussion, they mention that the DDoS-HOIC attack resulted in a lot of false positives. The amount of false positives highlights that detecting a DDoS attack is complicated.

Farahnakian et al. [3] implemented a semi-supervised neural network consisting of a latent autoencoder(discussed in section 3.2.10) which then connects to a supervised layer. They show that an autoencoder can learn the latent features on the KDD-CUP'99 dataset and perform well at detecting attacks. KDD-CUP'99 does not contain a DDoS attack, which is the primary attack in this thesis. Also, the KDD-CUP'99 has received criticism and encouragement not to use the dataset as a benchmark[4].

Sharafaldin et al. [5] developed a realistic DDoS attack dataset, which is the primary dataset used in this thesis. They have enumerated shortcomings and weaknesses in other datasets(for example, in KDD'99) and tried to address them. They also showcased a few machine learning models and their performances, namely the ID3 algorithm, Naive Bayes, Random Forest and Logistic Regression.

Khraisat et al. [6] did an interesting study to highlighting the current challenges of intrusion detection systems and datasets that are available. They also showed that anomaly-based intrusion detection systems suffer from a high false-positive rate. Minimizing the number of false positives in intrusion detection systems is an important area of research. Although a lot has happened since 2019, the paper was an excellent starting point for this thesis.

Wang et al. [7] was the first to combine the SHAP framework to yield explanations for intrusion detection systems. The paper highlights the theoretical foundation SHAP has and that SHAP applies to any model. They show that one can gain a good insight into why a model makes a specific prediction. Historically, explaining the output from neural networks in intrusion detection systems has been a difficult task.

Papers [2, 3] do not emphasize the performance of the respective models with respect to DDoS attacks. In this thesis, we are going to propose four models based on [2] and [3]. We are going to utilize the dataset from [5] which eventually are going to confirm the results and viability of the models in [2] and [3]. Using the method from [7] we are going to explain the proposed models and find the essential features. Using these features, we are going to make modifications to the data from [5] and try to bypass our IDS. We will discuss the viability of using SHAP for generating new attacks when attacking an IDS and the robustness of the proposed models.

# Chapter 3

# Background

## 3.1 Introduction to Distributed Denial of Service

### 3.1.1 Denial of Service

On the web, there exist many servers hosting a variety of services. A Denial of Service (DoS) attack aims to delay or deny legitimate users access to some service. To delay or prevent users from accessing the service, the attacker disables the machine's ability to reply to the users' requests. Hampering the server's ability to respond is typically done by resource exhaustion. When exhausting the resources of some server, one takes up bandwidth, disk space, or memory capacity. To execute an attack, the attacker needs to find and exploit some weakness in the server or service. For example, in a TCP 3-way handshake, assume that some user wants to access a web page and send an ACK signaling that he/she wants to establish a connection. The server replies with an SYN-ACK accepting the connection. When the user receives the SYN-ACK, he/she replies with an ACK confirming and establishing the TCP connection. When the server is waiting for the final ACK, the connection is half open and stored in a table with limited space. An attacker can exploit the vulnerability and fill the table with half-open connections, making the server unable to open any new connection. In turn, this makes the server unavailable, and the attack is successful.

### 3.1.2 Distributed Denial of Service

Distributed Denial of Service (DDoS) is a subcategory of DoS attacks, where the attacker uses multiple units to exhaust a target service. Often in a DDoS attack, one wants to maximize the throughput sent to a destination. The definition of throughput (in the context of DoS attacks) is the volume of packets arriving successfully at a destination. We measure throughput in bits per second (bps). Typically a DDoS attack starts with an attacker that has control over multiple master nodes. These master nodes, in turn, have control over slave nodes. The slave nodes can be anything from a smart fridge to a PC. At the attacker's command, the slave nodes will flood traffic and exhaust the victim.

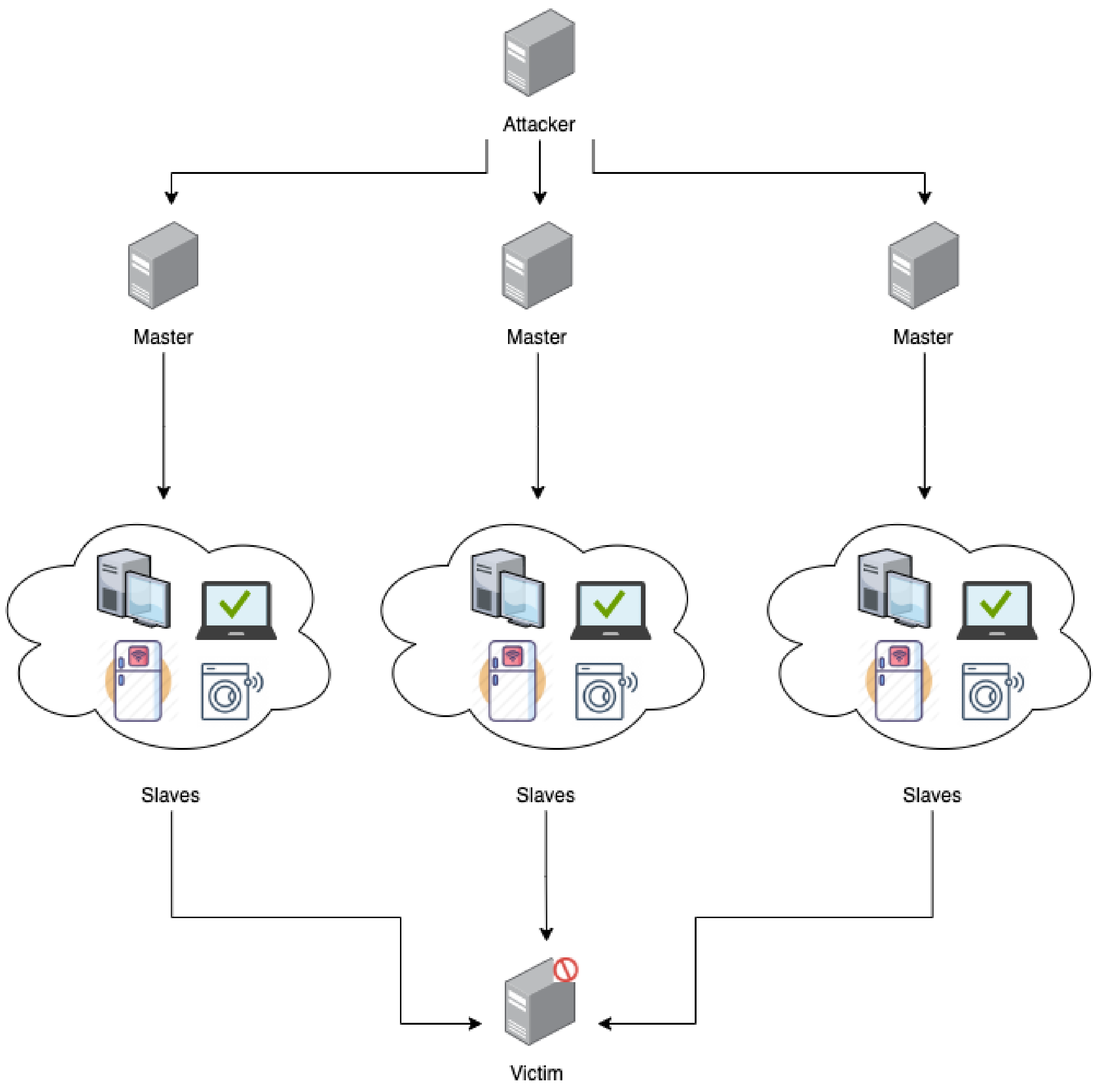


Figure 3.1: DDoS Attack

### 3.1.3 Reflection attacks

Reflection attacks, also known as amplification attacks, are a type of DDoS attack. An attacker needs four components to execute a reflection attacks[8].

- A server that can IP spoof
- A protocol that's vulnerable to reflection attacks, which often is UDP-based.
- A list of servers that supports the vulnerable protocol. This type of server is called a reflector.
- IP address of the victim

To perform a reflection attack, the attacker needs to spoof the IP of the requests. He/she spoofs the IP address of the victim. When the attacker sends traffic to a reflector with the spoofed IP address, the reflector replies with possibly amplified network traffic to the victim. The attacker will distribute packets randomly to his/her chosen reflectors which then, in turn, will bombard packets to the victim. Eventually, the victim's network gets congested, or the system's resources become depleted, and the targeted service gets taken down.

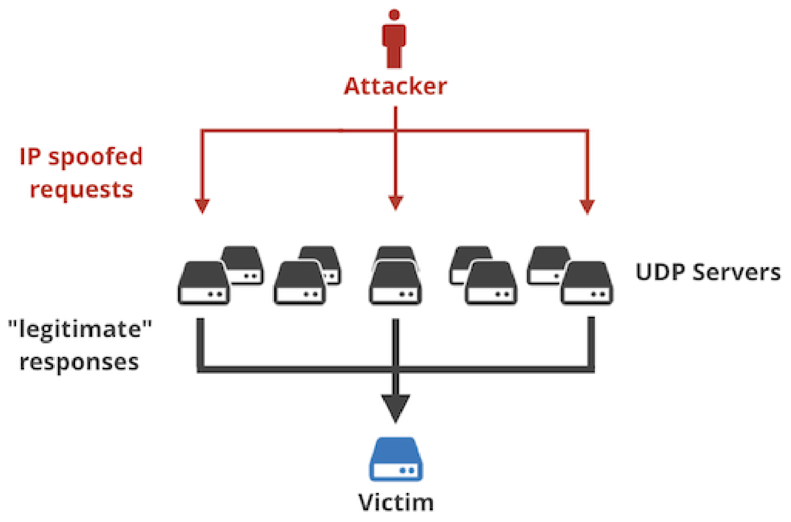


Figure 3.2: Reflection attack[8]

In reflection attacks, one often talks about the amplification factor. Imagine an attacker sending 1000 bytes to a reflector, and it replies to a victim with 30000 bytes. The ratio between the response and request from the reflector corresponds to the amplification factor. Hence the amplification factor of a 1000 bytes request versus a 30000 bytes response equals 30. Different attacks have a variety of amplification factors. A higher amplification factor means that the capacity of an attack is even more prominent. Below is a list of potential reflectors and some information about them.

**Domain Name Server(DNS)** Attackers send small spoofed requests to DNS resolvers, which then, in turn, send traffic to the target's IP address. To the victim, the traffic will seem to be benign. It has an amplification factor from 28-54[9].

**Lightweight Directory Access Protocol(LDAP)** Microsoft Windows Active Directory utilizes this protocol to verify user credentials. By sending tiny requests to vulnerable LDAP servers, they will reply to a victim's IP address with amplified network traffic. It has an amplification factor from 46-55[9].

**Microsoft SQL(MSSQL)** MSSQL is a database software provided by Microsoft. To perform an MSSQL reflection attack, the attacker sends a spoofed database query to an MSSQL server. The server will respond to the victim with vast amounts of traffic containing the query payload. It could potentially be a greatly amplified attack.

**NetBIOS** The NetBIOS Name Services' purpose is to facilitate programs on separate computers to communicate and create sessions to access shared resources and find each other over a local network. The NetBIOS attack works by sending spoofed queries which replies to the victim with potentially amplified traffic. NetBIOS attacks are typically performed over a local area network (LAN). This reflector has an amplification factor between 2.56 and 3.85 [10].

**Network Time Protocol(NTP)** The NTP protocol was designed to synchronize machines' clocks over a network. Vulnerable NTP servers have the "monlist" command enabled. An attacker would send spoofed NTP requests in an NTP attack that execute the "monlist" command. The server responds to the victim with an amplification factor between 20 - 200[11].

**Portmap** RPC portmap, also known as a port mapper, tells a client how to call a particular version of an Open Network Computing Remote Procedure Call (ONC RPC) service. This reflector has an amplification factor between 9.65 and 50.53 [10].

**Simple Network Management Protocol(SNMP)** This reflector is a very common network management protocol used for configuring and collecting information from network devices like servers, hubs, switches, routers, and printers. This reflector's amplification factor lies between 600 and 1700[12].

**Trivial File Transfer Protocol(TFTP)** TFTP is used to read, write or transfer files using a remote machine. When the attacker performs a TFTP attack, he/she sends a request to a vulnerable server asking for a file. The server, in turn, responds to the victim with the file. The TFTP reflector has an amplification of about 60[13].

### 3.1.4 Exploitation attacks

In an exploitation attack, the attacker utilizes weaknesses in a protocol to execute an attack, to take down a server/service.

#### SYN Flood

In a syn-flood attack, the attacker would send spoofed syn-packets to a vulnerable server and create half-open connections. The server will try to reply to the IP address with syn/ack packets. When the server does not get any replies, it will try to send multiple syn/ack packets to the spoofed IP address, leaving the connection half-open. Eventually, the server will not accept any more incoming connections, thus making the service unavailable [5].

#### UDP Flood

UDP-flood is a very simple attack where you flood a host with a large traffic flow to random ports. When the traffic volume gets large enough, the system would potentially crash or the performance could be severely degraded [5].

#### UDP-Lag

In a UDP-Lag attack, the goal is to harm a connection between clients and a server. Ultimately, the attacker intends to hog the bandwidth of other users such that the server/service becomes unavailable/unusable for them. The attack is carried out by either using a lag switch or a software that runs on a network that hogs the bandwidth of other users [5].

## 3.2 Machine Learning and Deep learning

The subfield of AI, machine learning, is a rapidly moving field. We have seen a surge of high-performing techniques that manages to outperform human accuracy on certain tasks, for example, the well-known AlphaGo Zero project[14]. The possibilities of machine learning are many.

In machine learning, we aim to find information about a data distribution by running a machine learning (ML) algorithm which outputs a model. The main idea of a machine learning model is that it can generalize to different inputs compared to the initially learned data. There are various algorithms with different complexity, and selecting an ML algorithm depends on the task to solve and the data available.

### 3.2.1 Data for learning

We separate data into four categories: numerical, categorical, time series, and textual data. Numerical data includes discrete and continuous numbers. Categorical data is non-numerical data that can be divided into groups and is usually denoted with a fixed number. Gender, animal, or native language are examples of categorical data. Time series is temporally ordered data, for instance, stock prices or bank transactions. Textual data is any text meaningful to learn from, where the textual data is converted into numerical vectors. Typically, the data used for training is called training data, while the data used to test the model's performance is called validation or test data. To solve ML tasks, we need an adequate amount of data for training and validation. Equally important is that the training data used for learning represents the target distribution we aim to learn. However, in real life, there is possibly noise and outliers skewing the distribution. Noisy data is data that has been rendered distorted or possibly even corrupt. Outliers are defined as an observation that lies an abnormal distance from other values from a population [15]. There exist multiple techniques to cope with poor data quality and reduce the impact of noise and outliers.

### 3.2.2 Machine learning methods

In machine learning, one typically separate different algorithms into two categories, supervised and unsupervised learning. The category represents how we provide feedback to the algorithm to facilitate learning.

#### Supervised learning

If we were to look at a supervised model as a black box, it takes an input $X_i$ and outputs a predicted label $y_i$. The way a supervised ML algorithm generates models is by learning from target labels corresponding to each input. It learns by comparing its predicted value with the true target label and then applying modifications to minimize the error. If the label is categorical, we call it a classification problem. An example of a classification problem could be to distinguish photos of dogs, cats, and rabbits. In this setting, the picture is the data point, and the animal is the label. Problems with continuous labels are called regressions, for instance, estimating a numerical function. A significant challenge with supervised learning is that acquiring data is expensive, as the data often needs to be labeled manually [16].

#### Unsupervised learning

In contrast to supervised learning, we do not have labels associated with the data. An unsupervised learning algorithm aims to learn features from the data itself and potentially discover hidden factors and patterns. Since an unsupervised algorithm often excels at learning hidden patterns, it is often utilized for anomaly detection. A typical example of an unsupervised problem is credit card fraud detection. [17]

#### Types of tasks we may want to solve using machine learning

- Classification - Supervised. Example: Classify what's in a picture

- Anomaly detection - Unsupervised. Example: Finding fraudulent bank transactions
- Density estimation - Unsupervised. Example: Finding out where and how your data is clustering
- Synthesis and sampling. Unsupervised. Example: Generating pictures
- Imputation - Both. Example: Finding missing values in data.
- De-noising - Unsupervised. Example: Given a damaged picture, try to recover the original picture.

### 3.2.3 Loss functions

When training a machine learning model, we need some loss function denoted as $\mathcal{L}$. The loss is used to get some intermediate differentiable result that we use to optimize the model further. Thus, the loss defines what is considered a good output and what is considered bad. Different loss functions exist, and selecting one depends on the data at hand and the task to solve. Some standard loss functions are listed below.

**Definition 3.2.1** (Mean Squared Error (MSE))**.** Given $n$ number of data points, the true value $X_i$ and corresponding predicted value from a model $X_i'$.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(X_i - X_i')^2$$

The mean squared error rewards errors in the range (-1, 1) and punishes errors less than -1 and greater than 1. When evaluating a prediction, we observe that a smaller MSE means a better prediction. It is possible to use MSE for classification problems, but it is usually utilized for regression.

**Definition 3.2.2** (Categorical cross-entropy)**.** Given $n$ number of data points, target label $y_i$ and corresponding predicted label $y_i'$

$$CE = -\sum_{i=1}^{n} y_i \cdot \log y_i'$$

Categorical cross-entropy is used for classification problems and rewards predicted probabilities that are closer to the target probabilities.

### 3.2.4 Performance measures

After training a model, we need to measure the performance of the model. In this thesis, we will solely use binary classifications. We denote 0 as false and 1 as true. When evaluating a binary classification, we put it into one of four different categories. Namely true positives(TP), false positives(FP), false negatives(FN) and true negatives(TN). Given a true value $y_i$ and predicted value $y_i'$ we define

$$(y_i = 1) \wedge (y_i' = 1) \implies TP \tag{3.1}$$
$$(y_i = 0) \wedge (y_i' = 1) \implies FP \tag{3.2}$$
$$(y_i = 1) \wedge (y_i' = 0) \implies FN \tag{3.3}$$
$$(y_i = 0) \wedge (y_i' = 0) \implies TN \tag{3.4}$$

After classifying every predicted output into the four categories and summing the total number of members of each category, we use it as input to the performance measurements. Selecting a performance measurement depends on whether or not the data is balanced and the task to solve.

**Definition 3.2.3** (Accuracy)**.** The accuracy is defined as

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

Accuracy is a performance measure that function best when the dataset is balanced. A balanced data set means that each class in the data set has about the same amount of samples.

**Definition 3.2.4** (Precision)**.** The precision is defined as

$$precision = \frac{TP}{TP + FN}$$

Precision measures the number of correctly predicted positive samples versus the total number of positive samples. The precision is often used when it is critical to find positive samples.

**Definition 3.2.5** (Recall)**.** Recall is defined as

$$recall = \frac{TN}{TN + FP}$$

Recall measures the ratio of correctly predicted negative cases versus all negatives in the dataset. Precision and recall are often used in some combination.

**Definition 3.2.6** (F1-Score)**.** The F1-score is defined as

$$f_1 = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

F1 score is utilized when we need to consider both the number of false positives and false negatives. The F1 score will also be a good choice even when there is a class imbalance.

### 3.2.5 Anatomy of a learning system

#### Overfitting and underfitting

Machine learning algorithms aim to utilize training data to generate a model that generalizes well. However, many factors can influence the training process. Hence, it is possible to end up with a model that cannot correctly predict and generalize to unseen data. In section 3.2.1 we mentioned that data could have outliers and noise that skew the distribution. These impacting factors can render it challenging to learn from the dataset. Two major problems that arise when training a model is overfitting and underfitting. Overfitting is when the trained model has a low training error and high test error. Contrary, when the model has a high training error, we call it underfitting. Overfitting and underfitting are tightly coupled with bias and variance.

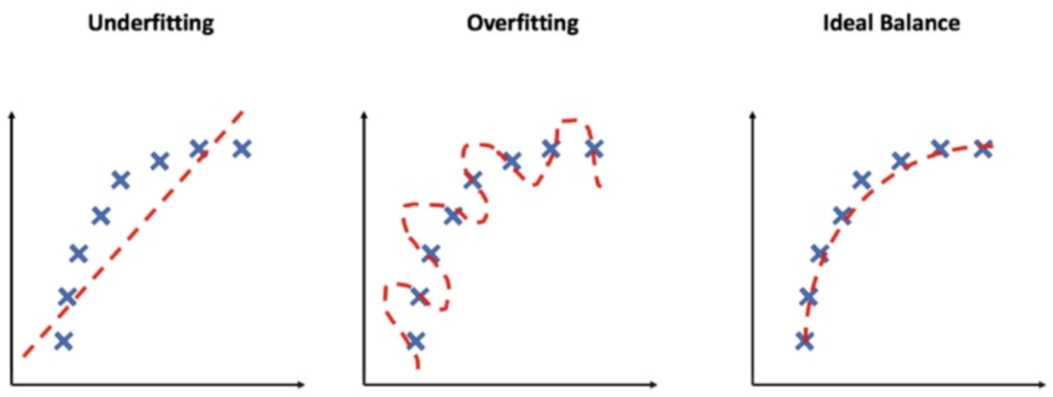


Figure 3.3: Underfitting and overfitting, source: See appendix D

#### Bias and variance

Ideally, we would want to know an exact function describing the data we have. However, that is rarely possible. We can use a machine-learning algorithm to train a model to estimate the function. When training a model with some training data, we want to capture true relationships in the data. The concept of a machine learning algorithm not being able to model relationships in the data is called bias. Bias is something we define as modeling error. The difference between how the model fits the training and validation set is called variance. Variance is a way to see how much the computed estimate varies when we use different training and validation samples. Higher variance means a more sensitive model, and is often associated with overfitting, while a high bias indicates underfitting.

#### Capacity

Capacity refers to the ability to model/fit some range of functions. A low capacity indicates that you will underfit, while with high capacity you are more likely to overfit.

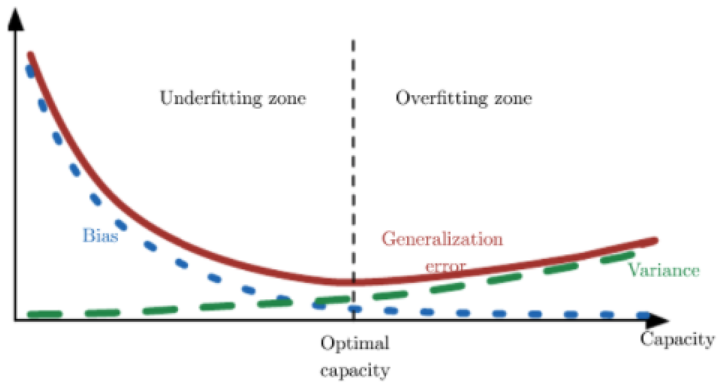


Figure 3.4: Bias-variance trade-off

### 3.2.6 K-Fold Cross-Validation

When trying to train an optimal machine learning model, we need to utilize our data to the best extent. The solution is to first split the data into a training set and a test set. The test set is put aside to evaluate the final model performance. The training set is subsequently split into K number of folds, depending on how many parameter combinations to try out. The folds are structured so that each data point is used $k-1$ times in training and precisely one time for validation, significantly reducing our bias. Each fold is split into its own fold-training set and fold-validation set for a specific hyper-parameter combination. The model with some hyperparameter set is trained on a fold-training set and gets a performance score based on the fold-validation set. After iterating through the k folds, the model with the best performance score gets selected and is retrained with all available data except the test set we set aside earlier. After training, the final model performance is evaluated using the test set. After determining the performance score, the model is retrained using all data, and we save its weights for later use.

### 3.2.7 Linear models

One of the most simple models is the linear model. Linear models assume there is a linear relationship in the data. The model function is in the form [18]

$$y_i' = \beta_0 + \beta_1 x_{i1} + \epsilon_i$$

where $y_i'$ is the predicted value for $x_i$, the hyperparameters of the model are the slopes $\boldsymbol{\beta}$, and $\epsilon_i$ is the ith unpredictable random disturbance. Using mean squared error as a loss function, we can fit a line to minimize the loss. An advantage of the linear model is that it is straightforward to understand why it makes a certain prediction. The main disadvantage is that linear models cannot fit most data distributions since they can only fit straight lines.

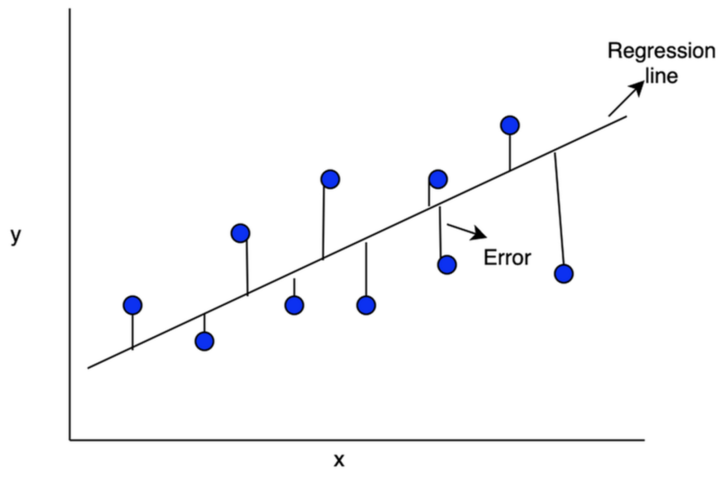


Figure 3.5: Example of linear regression, source: Appendix D

Until now, we have discussed simple regression, but it is possible to generalize the model into higher dimensions, which is called multiple regression. If we were to increase the dimension to $n$ dimensions, the model function would look like

$$y_i' = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_n x_{in} + \epsilon_i$$

The advantage and disadvantage of simple regression also applies to multiple regression.

### 3.2.8 Deep Learning

Deep neural networks consist of multiple layers in between the input and output layer. The layers consist of neurons that take multiple weights as input and then output an activation value that is passed to the next layer.

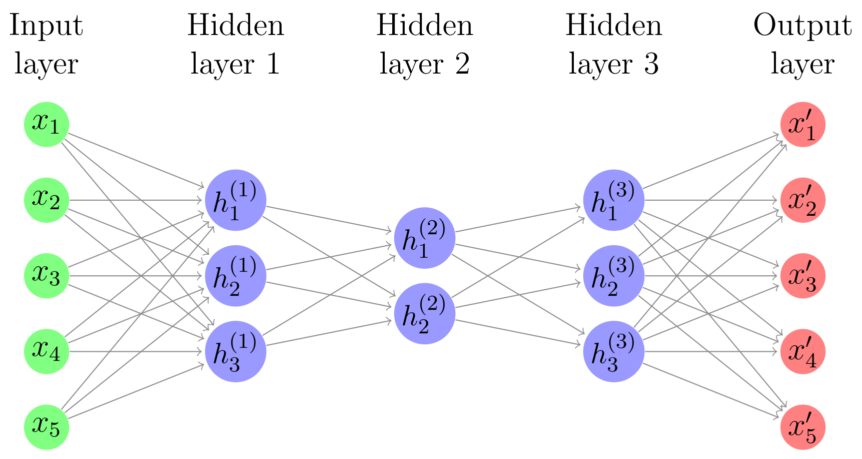


Figure 3.6: A deep neural network (autoencoder)

### Neurons

A neuron has multiple weights as inputs and outputs some value based on its input weights and bias. $f$ denotes the activation function. A neural network consists of many neurons.

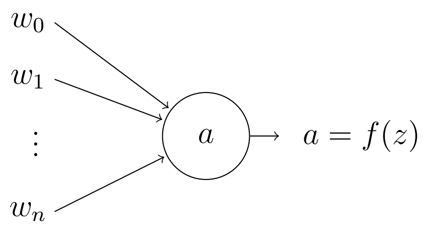


Figure 3.7: A single neuron

### Activation functions

An activation function defines how a neuron will calculate an input from the previous layer to the next layer. Below is a list of activation functions and how they work [19].

**Rectified Linear Unit(ReLU)** is defined as

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

ReLU is one of the most simple activation functions to use and is computationally inexpensive. It will avoid the vanishing gradient problem. However, there is a catch, and that is it has a zero-gradient when x is below zero. If the weight ends up being less than zero, its gradient will never make it go above zero again and thus will not recover, which is considered a "dead" neuron. The derivative of the ReLU function is

$$\frac{\partial f(z)}{\partial z} = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \\ undefined & otherwise \end{cases}$$

When the gradient of ReLU is undefined, one can select to either set the value to zero or one.

**LeakyReLU** is defined as

$$f(z) = \begin{cases} z & z > 0 \\ \alpha z & z \leq 0 \end{cases}$$

While similar to ReLU, it has the advantage of solving the dying neuron problem. The derivative is defined as

$$\frac{\partial f(z)}{\partial z} = \begin{cases} 1 & z > 0 \\ \alpha & z < 0 \end{cases}$$

Like ReLU the derivative is undefined when $x$ is zero. You can either choose to set it equal to 1 or $\alpha$

**Sigmoid** is defined as

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

The main advantage of the Sigmoid activation function is that it is mathematically efficient to work with. However, it is computationally inefficient due to the negative exponentiation, which is costly, and often makes ReLU superior. The derivative of the Sigmoid function is

$$\frac{\partial f(z)}{\partial x} = f(z)(1 - f(z))$$

Once again, the derivative is smooth to work with mathematically but fails computationally due to negative exponentiation. While Sigmoid is less used in hidden layers, it excels when used in the output layer, especially for autoencoders.

### Neural network layers

The first layer of any neural network is the input layer, and the last layer is the output layer. In between the input and output layer, there are the hidden layers. The layer type determines how the weights connect to the neurons from the previous layer to the current one. In this thesis, we use the Dense, Convolutional, MaxPooling, and Unpooling layers.

**Dense layer** is simply a fully connected layer, which means each neuron is connected to every neuron in the previous layer. The example neural network above(figure 3.6) consists of fully connected layers.

**Convolutional Layer** When discussing the convolutional layer, it makes sense first to define a kernel. Let's imagine that we have an image with three channels. A kernel is a small $k\,x\,d$ grid containing $k\,x\,d$ weights that we put on top of an image channel. Each channel of an image has its own kernel. The kernel will connect all local values and calculate a new real value and slide over the entire image and calculate a new value for each unique position. The stride parameter defines the step size when sliding over a channel. After calculating a value for each unique kernel position, it generates a new channel with less than or equal dimension to the channel in the previous layer. Each channel is added together, forming the output from a filter. In a convolutional layer, we also have to specify how many filters we want. A filter is a collection of all the kernels. In a 2D convolution, the filter and kernel are interchangeable because there is only one channel in the image. Hence each filter has one kernel. The idea of a convolution is that each filter behaves a little bit differently than the others. Consequently, each filter is catching some new patterns in the data.[20].

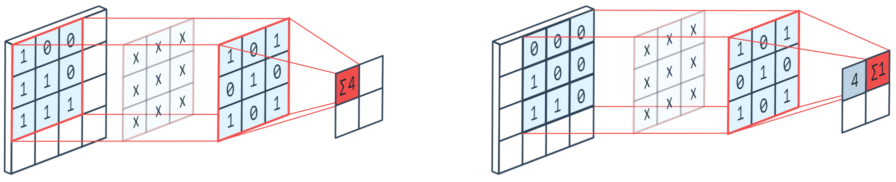


Figure 3.8: 2D convolution, source: see **appendix D**

**Pooling** Much like the convolutional layer, we have something similar to the kernel, namely a pool. The pool is also a small grid put on top of an image and can have any $k\,x\,d$ dimension. However, in contrast to the convolutional layer, there are no trainable weights in the pooling layer. In the pooling layer, we assign a value from the pool using different techniques like MaxPooling, where we use the max value from the pool. AveragePooling is another pooling layer, where we take the average of all values in the pool and use that as the output value. Strides work precisely the same in pooling as for convolutional layers [20].

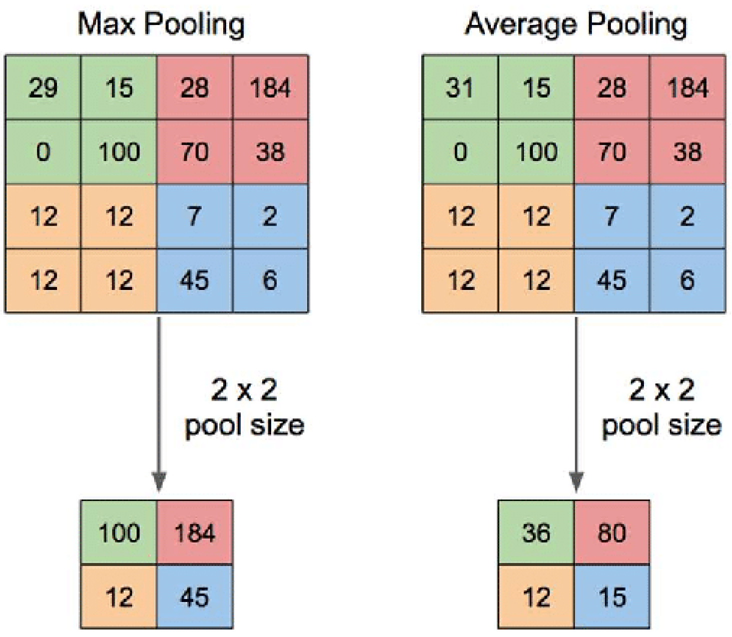


Figure 3.9: Max pooling and average pooling, source: see **appendix D**

**Transposed convolution** In a convolutional layer, we specified that the output dimension is less than or equal to the input dimension. Transposed convolutions aim to output a dimension that is higher than the input. In a transposed convolution, a kernel is used almost the same way as in a convolutional layer. The kernel gets slid over each position of the image and outputs a matrix. If we apply a 2x2 kernel on top of a 2x2 image, we end up with a 3x3 output. The process is further showcased in the figure below.

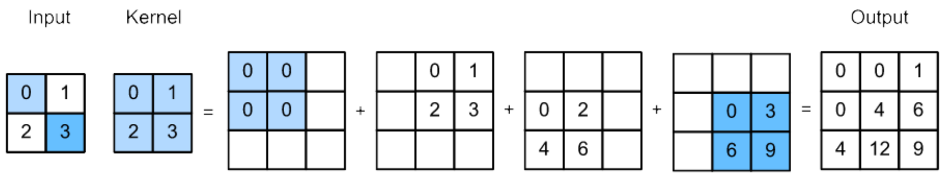


Figure 3.10: The complete transposed convolution operation

### Forward pass and back propagation

This section is based on the work from [21]. Neural networks are trained using a forward pass to get some output from the neural network. Using the output from the forward pass in the backpropagation step, it then calculates the gradient of the loss function with respect to the weights. Using the gradients and some learning rate $\alpha$, the network updates its weights by going in the opposite direction of the gradient with step size $\alpha$. Let us consider a simple neural network with one input layer, two hidden layers, and one output. It takes an input $x$ with 3 features. The output value is denoted $y'$. The weights in the network are initiated with random values. Each neuron in the network has its own activation value. When referring to the activation value of $h_1^{(1)}$ we denote it $a_1^{(1)}$. $a^{(1)}$ is a vector for all the activations in the first layer. We denote $b^{(1)}$ for the bias of the first hidden layer. Biases are commonly initialized to 0. We also use the Sigmoid activation function since it is easy to use mathematically.

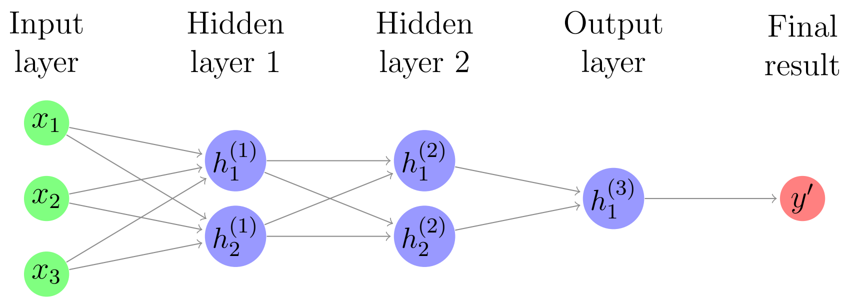


Figure 3.11: A simple deep neural network

When the neural network updates its weights, it first does a forward pass to get some output from the model. To do a forward pass, we go through the following calculation to get the output $y'$

**Forward pass: The first layer**

$$\begin{aligned}
a^{(1)} &= \sigma(z^{(1)}) \\
z^{(1)} &= w^{1T}x + b^{(1)} \\
&= \begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + b^{(1)} \\
&= \begin{bmatrix} \sum_{i=1}^{3} w_{i\,1}^{(1)} x_i + b^{(1)} \\ \\ \sum_{i=1}^{3} w_{i\,2}^{(1)} x_i + b^{(1)} \end{bmatrix} \\
a^{(1)} &= \begin{bmatrix} \sigma(\sum_{i=1}^{3} w_{i\,1}^{(1)} x_i + b^{(1)}) \\ \\ \sigma(\sum_{i=1}^{3} w_{i\,2}^{(1)} x_i + b^{(1)}) \end{bmatrix}
\end{aligned}$$

**Forward pass: The second layer**

$$\begin{aligned}
a^{(2)} &= \sigma(z^{(2)}) \\
z^{(2)} &= w^{2T}a^{(1)} + b^{(2)} \\
&= \begin{bmatrix} w_{1\,1}^{(2)} & w_{2\,1}^{(2)} \\ \\ w_{1\,2}^{(2)} & w_{2\,2}^{(2)} \end{bmatrix} \begin{bmatrix} a_1^{(1)} \\ \\ a_2^{(1)} \end{bmatrix} + b^{(2)} \\
&= \begin{bmatrix} \sum_{i=1}^{2} w_{i\,1}^{(2)} a_i^{(1)} + b^{(2)} \\ \\ \sum_{i=1}^{2} w_{i\,2}^{(2)} a_i^{(1)} + b^{(2)} \end{bmatrix} \\
a^{(2)} &= \begin{bmatrix} \sigma(\sum_{i=1}^{2} w_{i\,1}^{(2)} a_i^{(1)} + b^{(2)}) \\ \\ \sigma(\sum_{i=1}^{2} w_{i\,2}^{(2)} a_i^{(1)} + b^{(2)}) \end{bmatrix}
\end{aligned}$$

**Forward pass: The third layer**

$$\begin{aligned}
y' = a^{(3)} &= \sigma(z^{(3)}) \\
z^{(3)} &= w^{3T}a^{(2)} \; + \; b^{(3)} \\
&= \begin{bmatrix} w_1^{(3)} & w_2^{(3)} \end{bmatrix} \begin{bmatrix} a_1^{(2)} \\ \\ a_2^{(2)} \end{bmatrix} + b^{(3)} \\
&= \sum_{i=1}^{2} w_i^{(3)} a_i^{(2)} + b^{(3)} \\
a^{(3)} &= \sigma(\sum_{i=1}^{2} w_i^{(3)} a_i^{(2)} + b^{(3)})
\end{aligned}$$

**Back propagation** Now that we have calculated the predicted output $y' = a^{(3)}$, we want to calculate the loss. Let us use MSE.

$$L = \frac{1}{1}\sum_{i=1}^{1}(y - y')^2 = (y - y')^2$$

In backpropagation, we want to find how changing the weights and biases will affect the loss. First, we want to find the rate of change of the loss in terms of the weights and biases, the gradient. After calculating the gradient, we update the hyperparameters by moving in the opposite direction with a step length of $\alpha$, the learning rate. First, let us define some derivatives that we will need when calculating the derivatives of both weights and biases. Note that $a$ signals any activation value.

$$\begin{aligned}
\frac{\partial L}{\partial a} &= -2(y - a) \\
\frac{\partial L}{\partial z} &= \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \\
a &= \frac{1}{1+e^{-z}} \\
\frac{\partial a}{\partial z} = \frac{-1}{(1+e^{-z})^2}e^{-z}(-1) &= \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} = a(1 - a) \\
\frac{\partial L}{\partial z} &= -2(y - a)(1 - a)a
\end{aligned}$$

Using the derivatives, we can now derive a generic way of finding the derivative of the loss in terms of weights for any given layer.

$$\begin{aligned}
\frac{\partial L}{\partial w} &= \frac{\partial L}{\partial z}\frac{\partial z}{\partial w} \\
z &= wx + b \\
\frac{\partial z}{\partial w} &= x \\
\frac{\partial L}{\partial w} &= -2(y - a)(1 - a)ax
\end{aligned}$$

In this equation, x denotes the input to the neuron, a means the output, and y is the true value. Now, let's do the same calculations in terms of bias.

$$\begin{aligned}
\frac{\partial L}{\partial b} &= \frac{\partial L}{\partial z}\frac{\partial z}{\partial b} \\
\frac{\partial z}{\partial b} &= 1 \\
\frac{\partial L}{\partial b} &= -2(y - a)(1 - a)a
\end{aligned}$$

Now, using the generic functions above, we end up with the following equations for each set of weights.

$$\frac{\partial L}{\partial w^{(3)}} = -2(y - y')(1 - y')y'a^{(2)}$$
$$\frac{\partial L}{\partial w^{(2)}} = -2(y' - a^{(2)})(1 - a^{(2)})a^{(2)}a^{(1)}$$
$$\frac{\partial L}{\partial w^{(1)}} = -2(a^{(2)} - a^{(1)})(1 - a^{(1)})a^{(1)}x$$

Let us do the same for bias

$$\frac{\partial L}{\partial b^{(3)}} = -2(y - y')(1 - y')y'$$
$$\frac{\partial L}{\partial b^{(2)}} = -2(y' - a^{(2)})(1 - a^{(2)})a^{(2)}$$
$$\frac{\partial L}{\partial b^{(1)}} = -2(a^{(2)} - a^{(1)})(1 - a^{(1)})a^{(1)}$$

For each layer, update their respective weights and biases with the following equations.

$$w = w - \alpha\frac{\partial L(w,b)}{\partial w}$$

$$b = b - \alpha\frac{\partial L(w,b)}{\partial b}$$

It is called backpropagation because the weights are updated backward from the last layer to the first layer.

### 3.2.9 Optimizers

An optimizer defines how a neural network learns. Assume we have weights and biases $w$ and $b$, a loss function $\mathcal{L}$, and that $\mathcal{L}(w, b)$ denotes the loss using a subset of the data defined by an optimizer. The aim for an optimizer is to update $w$ and $b$ such that $\mathcal{L}(w, b)$ is minimized. The optimizer doesn't know the terrain of the loss and has to find its way to minima iteratively. An iteration is defined as an epoch. The optimizer may get stuck at local minima, thus not finding its way to a global minimum. [22]

#### Gradient descent

In an epoch, $t$, gradient descent updates the weights $w$ and biases $b$ based on the loss of the entire data with step size $\alpha$ using the following formulas

$$w^{(t+1)} = w^{(t)} - \alpha\nabla w^{(t)}, \qquad \text{where } \nabla w^{(t)} = \frac{\partial\mathcal{L}(w^{(t)}, b^{(t)})}{\partial w^{(t)}}$$

$$b^{(t+1)} = b^{(t)} - \alpha \nabla b^{(t)}, \qquad \text{where } \nabla b^{(t)} = \frac{\partial \mathcal{L}(w^{(t)}, b^{(t)})}{\partial b^{(t)}}$$

Calculating the loss of the entire dataset will yield a high value, making each step size huge. Hence, naively using stepping size $\alpha$ could lead to the model stepping past a minimum [22].

### Stochastic gradient descent

Section 3.2.8 showed the process of how a neural network can update its weights, which is called stochastic gradient descent (SGD). SGD works precisely like gradient descent. However, instead of calculating the entire dataset's loss and updating $w$ and $b$, SGD calculates the loss of each data point to update $w$ and $b$. In contrast to gradient descent, SGD will often take small steps, leading to the model being unable to reach a minimum. The reason why the models often fail to reach a minimum is that noisy samples can make the model update the parameters wrongfully[22].

### Mini-batch gradient descent

Mini-batch SGD was proposed to combat how the SGD optimizer is affected by noisy samples. Mini-batch SGD works by updating parameters for every set of $N$ samples instead of for each sample in the data [22]. The collection of $N$ samples is the mini-batch.

### SGD with momentum

SGD with momentum, also called momentum, aims to solve regular SGD's problem with noisy samples by accelerating SGD in the relevant direction. To achieve said acceleration, momentum adds the past update vector multiplied with friction constant $\gamma$ to the current update vector[22]

$$v_w^{(t)} = \gamma\, v_w^{(t-1)} + (1-\gamma)\nabla w^{(t)}$$
$$v_b^{(t)} = \gamma\, v_b^{(t-1)} + (1-\gamma)\nabla b^{(t)}$$

Note that the default value of is often set to $\gamma = 0.9$. Updating the weights and biases in momentum looks like

$$w^{(t+1)} = w^{(t)} - \alpha\, v_w^{(t)}$$
$$b^{(t+1)} = b^{(t)} - \alpha\, v_b^{(t)}$$

### RMSProp

RMSProp is an adaptive learning rate method that also solves SGD's problem with noisy samples by accelerating learning in the relevant direction. To achieve the acceleration, we define the following parameters [22]

$$s_w^{(t)} = \gamma\, s_w^{(t-1)} + (1-\gamma)\nabla (w^{(t)})^2$$

$$s_b^{(t)} = \gamma\, s_b^{(t-1)} + (1-\gamma)\nabla (b^{(t)})^2$$

Like with momentum, $\gamma = 0.9$ is the default value. Note that $(w^{(t)})^2$ and $(b^{(t)})^2$ uses the element-wise squaring operation. RMSProp updates the bias and weights with the following equations

$$w^{(t+1)} = w^{(t)} - \alpha \frac{\nabla w^{(t)}}{\sqrt{s_w^{(t)}} + \epsilon}$$

$$b^{(t+1)} = b^{(t)} - \alpha \frac{\nabla b^{(t)}}{\sqrt{s_b^{(t)}} + \epsilon}$$

$\epsilon$ denotes a small number, which is typically set to $10^{-8}$ to prevent 0-division.

### Adaptive Moment Estimation

Adaptive Moment Estimation(Adam) is the optimizer used by all the proposed models in this thesis and is an adaptive learning rate method that combines momentum and RMSProp. Adam works by computing an adaptive learning rate for each parameter in the model. The Adam optimizer begins by initalizing $s_w^{(0)} = 0$, $v_w^{(0)} = 0$, $s_b^{(0)} = 0$, and $v_b^{(0)} = 0$. Then, on iteration $t$, Adam first calculates the momentum part [22]

$$v_w^{(t)} = \gamma_1 \, v_w^{(t-1)} + (1 - \gamma_1)\nabla w^{(t)}$$

$$v_b^{(t)} = \gamma_1 \, v_b^{(t-1)} + (1 - \gamma_1)\nabla b^{(t)}$$

where $\gamma_1$ is the momentum friction. Subsequently, Adam needs to calculate the RMSProp part

$$s_w^{(t)} = \gamma_2 \, s_w^{(t-1)} + (1 - \gamma_2)\nabla (w^{(t)})^2$$
$$s_b^{(t)} = \gamma_2 \, s_b^{(t-1)} + (1 - \gamma_2)\nabla (b^{(t)})^2$$

where $\gamma_2$ is the friction constant from RMSProp. After calculating the momentum part and RMSProp part, it bias corrects them by using the following equations

$$v_{w,c}^{(t)} = \frac{v_w^{(t)}}{1 - \gamma^t} \quad , \quad v_{b,c}^{(t)} = \frac{v_b^{(t)}}{1 - \gamma^t}$$

$$s_{w,c}^{(t)} = \frac{s_w^{(t)}}{1 - \gamma^t} \quad , \quad s_{b,c}^{(t)} = \frac{s_b^{(t)}}{1 - \gamma^t}$$

Lastly, Adam performs the parameter updates using the following equations

$$w^{(t+1)} = w^{(t)} - \alpha \frac{v_{w,c}^{(t)}}{\sqrt{s_{w,c}^{(t)}} + \epsilon}$$

$$b^{(t+1)} = b^{(t)} - \alpha \frac{v_{b,c}^{(t)}}{\sqrt{s_{b,c}^{(t)}} + \epsilon}$$

The default values in the Adam optimizer are $\gamma_1 = 0.9$, $\gamma_2 = 0.999$, and $\epsilon = 10^{-8}$. The learning rate $\alpha$ needs to be fine-tuned in the cross-validation process.

### 3.2.10 Autoencoders

An example of an autoencoder can be seen in 3.6. There are two main types of autoencoders, latent and sparse. Generally, an autoencoder consists of an encoder part and a decoder part. The encoder transforms the input to a latent or sparse space. The decoder tries to decode the output from the encoder to the original input[3]. Given an input $x$, encoder $e$, decoder $d$, want to find two functions that estimates $d(e(x)) = x$. In a dense autoencoder, we want to impose a bottleneck in the model [23]. We hope that the dense model manages to extract the important features and learns to reconstruct valid data while being bad at reconstructing undesirable input.

## 3.3 Model interpretation

Model interpretability can be divided into two categories: global interpretability and local interpretability[24]. Global interpretability means the users can understand the model directly from its overall structure, while local interpretability exams an input and tries to find out why the model makes a certain decision[7]. We will use the Shapley Additive Explanations(SHAP)[25] framework to evaluate and explain our model outputs. The SHAP framework combines LIME[26] and Shapley Values[27]. The three next sections are based on [7].

### 3.3.1 Local Interpretable Model-Agnostic Explanations

Local Interpretable Model-Agnostic Explanations(LIME) was proposed in [26]. In deep learning, assume some input and that the input is put into a black box. This black box yields some output. The output could be what animal is in a picture. Alternatively, in our case, whether or not there is an attack going on. In a neural network, we often have high complexity and often high dimensional data. It would be beneficial both from an attacking and defending standpoint to know why a model gives a certain prediction, which LIME aims to solve.

**What does Model-Agnostic mean?** It simply means that it can be applied to any model. The LIME framework treats every machine learning model as a black box.

**What does local mean?** When visualizing the global decision boundaries of a complex machine learning model, it would look like a non-linear decision boundary that would not make any sense. What LIME does is to "zoom" in and look very locally around the data point you want to explain, such that you can use a linear model to understand the relationship. This local model is called a surrogate model and should yield good approximations locally.

**How does LIME work?** First, it permutes the original data and makes a data set solely consisting of permutations. Then it calculates the distance between the permutations and the original observations. After that, it makes predictions on new data using the complex model to explain and selects m features best describing the complex model from the permuted data. After that, it fits a surrogate model using the permuted data set with the m features and similarity scores as weights. The feature weights from the surrogate model will make up the explanations and describe the local behavior.

**How is the surrogate model calculated?** The local surrogate model can be calculated in the following way

$$\xi(x) = \underset{g \in G}{\arg\min}\{\mathcal{L}(m, g, w^x) + \Omega(g)\}$$

where $g$ represents the explanation model for the instance x, $G$ is the family of possible explanations, $\mathcal{L}$ is the loss function used to measure how close the predictions from the explanation model are to the original model, $m$ represents the original model, $w^x$ defines the weight between the sampled data and the original data. If the sampled data is similar to the original data, the weight is greater, and vice versa. $\Omega(g)$ represents the complexity of model $g$.

### 3.3.2 Shapley Value

The Shapley Value was introduced by Shapley in [27] and is a technique used in game theory. It is a way of determining the productive factor of each individual in a coalition. The technique can also be applied in machine learning, where the coalition is the data point, and an individual is the instance of some feature. In machine learning, the Shapley value is the average contribution of a feature value to the prediction in all possible coalitions[7]:

$$\phi_i(m, x) = \sum_{z' \subseteq \{x'_1, .., x'_n\}\{x'_i\}} \frac{|z'|!(M - |z'| - 1)!}{M!} \cdot [m(z' \cup x'_i) - m(z')]$$

where $z'$ is a subset of the features used in the model, $x$ is the vector of feature values of the instance to be explained. $M$ is the number of features, $m(z')$ is the prediction for feature values in set $z'$. When calculating $m(z')$, the ith feature is masked out and then simulated by drawing random instances or the random values of the ith feature from the dataset. Calculating the Shapley value requires computation time and power since there are $2^k$ possible coalitions of the feature values.

### 3.3.3 Shapley Additive Explanations

SHAP is a framework by Lundberg and Lee[25]. As previously stated, SHAP combines LIME and Shapley values. The innovation is that SHAP represents the Shapley value as a linear model. That particular way of looking at it connects Shapley values and LIME. Each prediction has its SHAP values and thus its explanation. SHAP calculates the explanation of an instance x in the following way

$$g(z') = \phi_0 + \sum_{j=1}^{M} \phi_j \, z'_j$$

where $g$ is the explanation model, $z'$ is the coalition vector. $z' \in \{0, 1\}^M$. 1 specifies that the entry is the same as in the original, while 0 means it is different. $M$ is the maximum coalition size. $\phi_j \in \mathbb{R}$ is the feature attribution for the feature $j$ for instance $x$. A larger $\phi_j$ means a bigger positive impact on the prediction made by the model.

A significant difference between LIME and SHAP is how they weigh the sampled instances. LIME would weigh similar samples higher. In contrast to this SHAP weighs small coalitions(few 1's) and large coalitions(many 1's) with large weights. Lundberg et al. proposed the SHAP kernel in [25] as

$$\pi_x(z') = \frac{(M - 1)}{\binom{M}{|z'|}|z'|(M - |z'|)}$$

Lundberg et al. showed that linear regression with the specified kernel would result in Shapley values. Estimating SHAP values of some instance $x$ is done by following the five steps below[7]

- Sample coalitions $z'_k \in \{0, 1\}^M, \ k \in \{1, \dots, K\}$
- Convert $z'_k$ to the original feature space: $m(h_x(z'_k))$. $h_x$ maps value to features where the coalition vector is 0 from another data instance.
- Compute the weight for each $z'_k$ using $\pi_x(z'_k)$
- Fit weighted linear model using the following loss function $\mathcal{L}$

  $$\mathcal{L}(m, g, \pi_x) = \sum_{z' \in Z} [m(h_x(z')) - g(z')]^2 \, \pi_x(z')$$

  where Z is the training data.
- Get Shapley values $\phi_k$, the coefficients from the linear model.

What makes the SHAP framework so useful is how fast it calculate the estimated Shapley values. It does it in polynomial time rather than exponential time, which is the case for regular Shapley values[25].

## 3.4 Describing the data

### 3.4.1 Time series

Time series data is simply data that has a temporal ordering. When analyzing the data, it is important to maintain the original order when learning.

**Definition 3.4.1** (Univariate time series)**.** A univariate time series $X = \{x_t\}_{t \in T}$ is an ordered set of real-valued observations, where each observation is recorded at a specific time $t \in T \subseteq \mathbb{Z}^+$ [28].

**Definition 3.4.2** (Multivariate time series)**.** A multivariate time series $X = \{x_t\}_{t \in T}$ is defined as an ordered set of k-dimensional vectors, each of which is recorded at a specific time $t \in T \subseteq \mathbb{Z}^+$ and consists of k real-valued observations $x_t = (x_{1t}, ..., x_{kt})$[28].

### 3.4.2 Capturing packets

When deciding to send some form of data from the application layer through the internet, the data has to go through some layers. Primarily, the data will be split into segments and transported through the transport layer. These segments, in turn, will be encapsulated in an IP datagram(packet) in the network layer. Then they're being sent down to the link layer and being encapsulated in a frame. Then the frame will be sent through the physical network layer. When receiving the frame, it will go the reverse way and unpacking each encapsulation. We record every packet being encapsulated and those who are un-encapsulated[5].

### 3.4.3 Packets vs Flows

Every packet contains some information in the header and some data(payload). When generating flows, we measure how many packets flow through, the header size and payload size measured against time, and create one unique flow per session. The flows are ordered by the starting time of each flow. [5]

### 3.4.4 CicFlowMeter and the UNB DDoS Data set

CiCFlowMeter is an open-source software developed by the University of New Brunswick's cybersecurity department. CiCFlowMeter records network traffic and translates the traffic into flows. The original paper states that it records 80 features, but this has been updated and increased to 84[5]. Using the CiCFlowMeter software, the University of New Brunswick researchers developed a realistic data set containing benign and DDoS traffic. When simulating the attacks, the researchers had set up two separate networks—a victim network with simulated benign traffic and an external attack network. As stated by Sharafaldin et al. [5] the top priority when generating the data set was to generate realistic benign traffic. They made a profiler that extracts behavior from 25 real users.

The data was captured on two separate days. On each day, the researchers executed attacks towards the victim network from the attack network.

| **Attacks** | **Attack times** |
|---|---|
| PortMap | 09:43 - 09:51 |
| NetBIOS | 10:00 - 10:09 |
| LDAP | 10:21 - 10:30 |
| MSSQL | 10:33 - 10:42 |
| UDP | 10:53 - 11:03 |
| UDPLag | 11:14 - 11:24 |
| SYN | 11:28 - 17:35 |

Table 3.1: The first day / training set

| **Attacks** | **Attack times** |
|---|---|
| NTP | 10:35 - 10:45 |
| DNS | 10:52 - 11:05 |
| LDAP | 11:22 - 11:32 |
| MSSQL | 11:36 - 11:45 |
| NetBIOS | 11:50 - 12:00 |
| SNMP | 12:12 - 12:23 |
| SSDP | 12:27 - 12:37 |
| UDP | 12:45 - 13:09 |
| UDPLag | 13:11 - 13:15 |
| WebDDoS | 13:18 - 13:29 |
| SYN | 13:29 - 13:34 |
| TFTP | 13:35 - 17:15 |

Table 3.2: The second day / test set

What makes this data set so useful is that it both contains malicious traffic and benign traffic. And it's also relatively updated compared to other data sets.

## 3.5 Processing the data

The data from UNB consists of network flow data. There are 84 columns where 77 of which are parameters that can be used for training. The first step is to evaluate potential obsolete columns. Then there are several steps we have to go through.

### 3.5.1 Correlation

Assuming two columns in a dataset $X$ and $Y$, we could want to find out if there is a linear association between them, which we call the correlation. The correlation value ranges from negative 1 to plus 1 where -1 means perfect negative correlation and +1 means perfect positive correlation. If the number is close to zero it means that there is no correlation[29]. Does some feature correlate? One could want to minimize the number of features being used to reduce the model size and increase efficiency. Some correlation is natural, but if some features correlate nearly perfectly, we can try to substitute one feature for all the highly correlated ones. To find features that represent each other, we use the Union Find algorithm. Substituting could lead to some degradation in performance and increase both inference speed and training efficiency. To decide whether or not to remove columns is done in the cross-validation process. Assuming normally distributed data, we find the correlations between X and Y using the following formula[29]

$$p_{X,Y} = \frac{\mathbb{E}[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X\,\sigma_Y}$$

### 3.5.2 Null values

In the data there are null values or values that are defined as infinite. We have simply decided to remove all rows containing either. This affects quite a lot of rows. However the data set is large enough that makes it acceptable to do it. One could also use regression, the mean or max value to impute the value of a row.

### 3.5.3 Normalization

There are many ways to normalize the data. The normalization function will be a tunable parameter when finding the optimal model in our K-fold cross-validation process.

#### MinMax scaling

The goal of the MinMax scaler is to scale the data to a specific range. This range is often $[0,1]$. Since we only need values in the range $[0,1]$ we scale using the following formula:

$$X_{sc} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

#### Standard scaling

When standard scaling, the idea is to normalize the data with some mean $\mu$ and standard deviation $\sigma$. The formula is given as

$$z = \frac{x-\mu}{\sigma}$$

$$\mu = \frac{1}{N}\Sigma_{i=1}^{N}(x_i)$$

$$\sigma = \sqrt{\frac{1}{N}\Sigma_{i=1}^{N}(x_i-\mu)^2}$$

#### Robust scaling

This type of scaling has some similarities to MinMax scaling. However, instead of using the minimum and maximum values to scale, it uses the first and third quartiles. Because it uses quartiles, it is less sensitive to outliers. The formula is as follows:

$$X_{sc} = \frac{X - Q_1(X)}{Q_3(X) - Q_1(X)}$$

### 3.5.4 Undersampling

Random undersampling is a simple technique used to balance a dataset with a given ratio[30]. Assume a dataset with corresponding labels $X$ and $y$, where $y$ contains the classes $c_1$ and $c_2$. If the number of datapoints in the $c_1$ class is less than $c_2$, we can sample the difference by selecting a random sample from $c_1$ without replacement until we have an equal amount of data in each class.

## 3.6 Intrusion Detection System

An Intrusion Detect System(IDS) is a piece of software whose purpose is to detect malicious activity on a network or a system. Thus, one often divides into network-based IDS(NIDS) and host-based IDS(HIDS). An IDS mainly uses two detection methods, signature-based and anomaly-based.

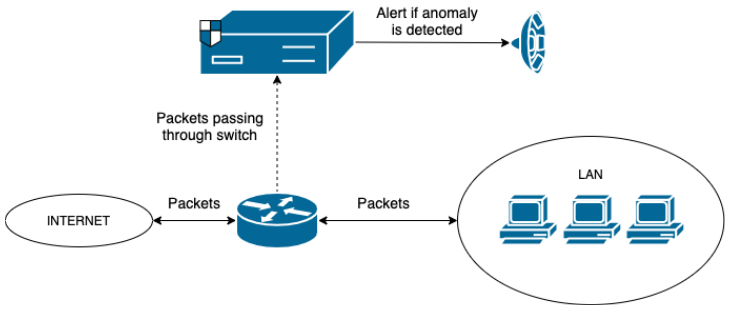


Figure 3.12: Network Intrusion Detection System

### 3.6.1 Signature-based IDS

A signature-based IDS is looking for patterns. An example of this could be a particular byte sequence in network traffic. One would have a database containing a variety of attack signatures. When evaluating network traffic, the system would try to match the traffic/behavior against the signatures in the database. The upside is that it can easily detect known attacks. However, it will not be able to generalize and detect unknown attacks. Since it effortlessly detects known attacks, it generates a minimal amount of false positives, given that attacks are clearly defined. Also, when the administrator gets an alarm, he/she would quickly see what type of signature(s) that triggered the alert.

### 3.6.2 Anomaly-based IDS

As previously stated, signature-based IDS are bad at detecting new kinds of attacks. To combat unknown attacks, researchers proposed an anomaly-based IDS. To build an anomaly-based IDS, one must first train a machine learning model that learns benign user behavior. Anything that differs from normal behavior would be classified as an anomaly and alert an administrator. While properly configured signature-based IDS have few false positives, anomaly-based approaches often have many false positives.

## 3.7 Proposing an anomaly-based Intrusion Detection System

### 3.7.1 Included and excluded functionality

Developing an IDS requires a lot of effort and features. This thesis focuses on developing and analyzing deep learning models and determining their robustness, upsides, and shortcomings. The main goal is to simulate variations of attacks to see if we can improve the throughput of an attack while concealing it from the IDS. Given the time available, it would be way too ambitious to make a complete IDS. That is why this thesis will employ a more basic implementation.

### 3.7.2 Focusing on one specific type of attack

Over the last decade, we have seen a surge of Distributed Denial of Service(DDoS) attacks. According to Neustar, they reported a 154% increase in attacks from 2019-2020[31]. Because DDoS attacks are in itself pretty easy to perform and can have devastating consequences, this thesis's focus will be to mitigate DDoS attacks.

### 3.7.3 Defining an attack / anomaly

There are many ways to define an anomaly. An efficient method to do this is by using the Mean Squared Error[Definition 3.2.1] reconstruction error threshold. Where $n$ is the number of features, $X_i$ is the actual value and $X_i'$ is the reconstructed $X_i$ from the machine learning model. Now we want to define some maximum allowed reconstruction error on our data. That is going to be the threshold that separates what is defined as an attack and what is not. An efficient definition of this threshold is simply taking the largest reconstruction error in the training set or test set

$$threshold = \max_{\forall X_i \in X\_train}(MSE(X_i, X_i'))$$

A second way is to use the interquartile range(IQR) as a baseline for a threshold selection, where we would set the threshold to 1.5 times the IQR plus the third quartile.

$$threshold = \frac{3}{2}IQR + Q_3 = \frac{3}{2}(Q_3 - Q_1) + Q_3 = \frac{5Q_3 - 2Q_1}{2}$$

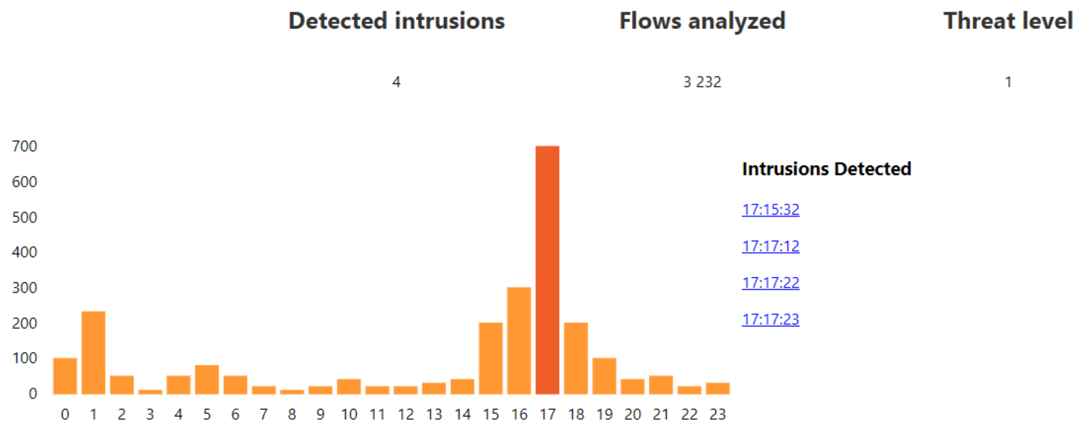
It is also possible to look at a histogram to determine where it is suitable to put an anomaly threshold; however, this is a bit harder to implement in a cross-validation process because we utilize an automatic pipeline.

### 3.7.4 Developing an IDS

To create a dashboard front-end, we utilize the React framework coupled with Typescript. Our backend utilizes python and the Flask framework for the API and Tensorflow/Keras for machine learning. The data is streamed to the front-end using Websockets.

#### Designing the dashboard

In the dashboard of the IDS, one can get an overview of the network flows from the last twenty-four hours. Should the IDS alert of an anomalous flow, the user can click on the detected anomaly to further inspect it and its parameters. Also, a user can fetch explanations of anomalous flows by clicking an explain-anomaly button. Below is a screenshot of the dashboard.



Detected intrusions 4

Flows analyzed 3 232

Threat level 1

Intrusions Detected

17:15:32

17:17:12

17:17:22

17:17:23

Figure 3.13: IDS Dashboard

#### Designing the backend

Using the CiCFlowMeter[32], we continuously record traffic flows from a network interface that writes flows to a CSV file. Using Python and a generator function, we monitor said CSV file. In a Python script, we poll the generator function monitoring the CSV file, yielding all flows that have not yet been seen. A seen flow is a flow that the generator function has yielded. The flows are aggregated into batches of 100. Each batch is sent to the machine learning part of the backend analyzing the submitted flows. Subsequently, we receive the result from the backend indicating if there were an anomaly in the batch. If an anomaly is present, the API receives instructions to update the front end with the timestamp of the anomaly and its associated data. After the API is instructed, the discovered anomaly gets added to a MongoDB collection. An overview of the infrastructure and dataflow is visualized below

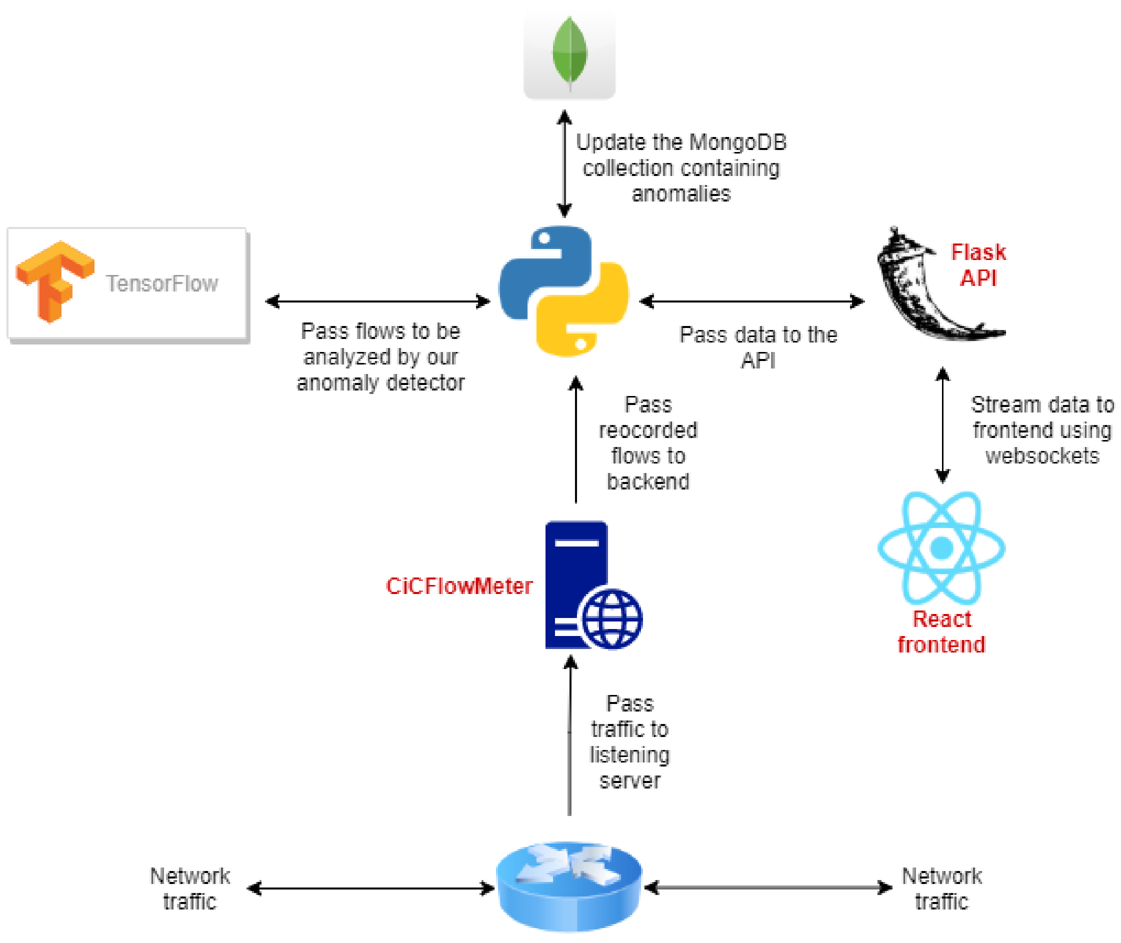

Update the MongoDB collection containing anomalies
TensorFlow
Pass flows to be analyzed by our anomaly detector
Pass data to the API
Flask API
Pass reocorded flows to backend
Stream data to frontend using websockets
CiCFlowMeter
React frontend
Pass traffic to listening server
Network traffic
Network traffic

Figure 3.14: IDS Dashboard

# Chapter 4

# Experiments and Results

## 4.1 Research method

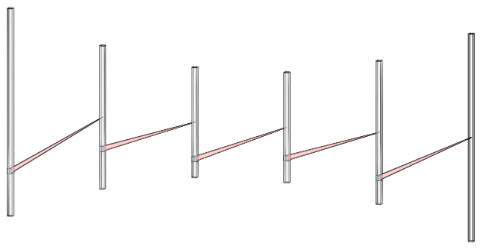
A variety of models have been tested in a Jupyter Notebook environment when determining the candidate model architectures for this thesis—the models with the best performance on a small proportion of the data set and the most interpretable model were selected. Hence, we selected simpler models. Every model that was a part of the initial selection process is specified in Appendix C. The selected models are optimized one by one using KFold cross-validation to determine their optimal hyperparameters. Common hyperparameters are scaler and learning rate. However, each model has its own set of hyperparameters as well. To train and evaluate the models, we use the CiCDDoS2019 dataset from the researchers at the University of New Brunswick[5]. The data is captured on two separate days. Several other papers, for example, [33, 34] combine the two days of data to make one extensive dataset and do a train-test split to train a model and measure its performance. Combining the two days of data could lead to information leakage and overfitting, making it impossible to evaluate properly how a model performs on unknown attacks. Thus, we follow how the original creators of the CiCDDoS2019 dataset[5] structured the training process, using day one for training and tweaking the models and set aside the second day of data for further model evaluation, and in our case, modifying the attacks.

Using KFold cross-validation, we optimize each of our proposed models, where the number of folds $k$ equals the number of hyperparameter combinations. After each candidate model is optimized, we want to test their robustness. The first step will be to execute attacks from day 2 of the dataset. We measure the performance and examine each set of SHAP values. Analyzing the SHAP values, we find what features contributing to the model correctly classifying the fraudulent flow. Utilizing the SHAP values to find what features the model deems as important, we try to modify the attack to maximize the throughput of the attack while also attempting to make the model misclassify fraudulent flows as benign. How well the model manages to withstand the modification will determine its robustness.

## 4.2 Candidate model architectures

The candidate model architectures are two supervised and two unsupervised models. As stated above, simpler models were selected over more complex ones since they are easier to explain, require less memory, and have fast inference compared to RNN-LSTM[33] networks and GANs[35].

### 4.2.1 Unsupervised dense autoencoder

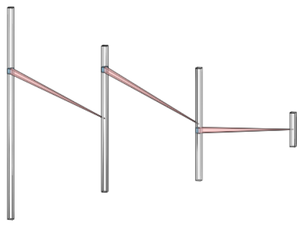


$$77 \rightarrow d_1 \rightarrow d_2 \rightarrow d_2 \rightarrow d_1 \rightarrow 77$$

Figure 4.1: The proposed dense Autoencoder

We begin by proposing a dense unsupervised autoencoder. The neural network takes a 77-dimensional vector as input and outputs a reconstructed 77-dimensional vector. We use the dense layer in each hidden layer. The dimensions of the first and last hidden layer are $d_1$, and the dimension of the two middle layers is $d_2$. In between the $d_1$ and $d_2$ on both sides of the network, we use dropout with a rate set to 25%. The network in itself is simple yet powerful enough to model complex relations. Using cross-validation we want to find the following optimal hyperparameters: how many neurons in dense layers $d_1$ and $d_2$, activation function in the hidden layers $f$, learning rate $\alpha$ and type of scaler $s$. We use the Sigmoid activation function in the output layer. When training the unsupervised dense autoencoder we only utilize benign data. The reason being we want the autoencoder to be good at reconstructing benign flows and bad at reconstructing fraudulent flows.

### 4.2.2 Supervised dense neural network



$$77 \rightarrow d_1 \rightarrow d_2 \rightarrow 2$$

Figure 4.2: The proposed supervised dense neural network

Subsequently, we define the supervised dense neural network, which takes a 77-dimensional input and outputs a 2-dimensional vector. The first dimension of the output corresponds to the probability of the input being benign and the second dimension being an attack. The hidden layers consist of dense layers. Like the previous model, we use dropout in between $d_1$ and $d_2$ with a rate set to 25%. When optimizing the model with cross-validation, we want to find the following hyperparameters: how many neurons in dense layers $d_1$ and $d_2$, activation function in the hidden layers $f$, learning rate $\alpha$ and type of scaler $s$. We use the Sigmoid activation function in the output layer. When training the sueprvised dense neural network we utilize both benign and fraudulent data. We make sure to have a balanced dataset using the undersampling technique described in Section 3.5.4.

### 4.2.3 Unsupervised convolutional autoencoder

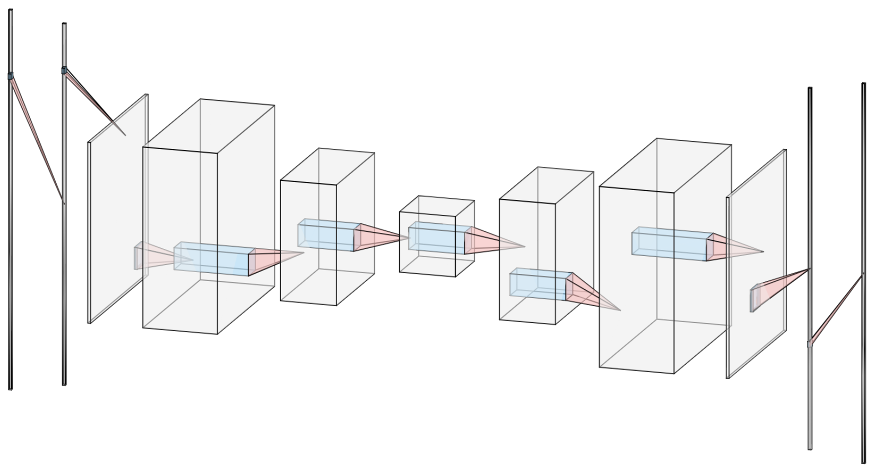


Figure 4.3: The proposed CNN Autoencoder

As a third model, we propose an unsupervised convolutional autoencoder. It takes a 77-dimensional vector as input and outputs a reconstructed 77-dimensional vector. The first hidden layer is a dense layer with 64 neurons reshaped into an $(8x8)$ image. We use a dense layer to ensure a smooth divisible dimension, which we need for the convolutional layers. After the dense layer, three convolutional layers with a kernel size of (2,2) follow. The final layers consist of three transposed convolutional layers, a dense layer with 64 neurons, and the output layer. We also try to substitute the second convolutional layer with the MaxPooling layer in the model selection process. MaxPooling can be effective in highlighting large feature values, which is often apparent in an attack. We use the Sigmoid activation function in the output layer. When optimizing the model, we want to find the following hyperparameters: Amount of filters per convolutional layer $\omega_i$, activation function in the hidden layers $f$, learning rate $\alpha$, and type of scaler $f$. Like the unsupervised dense autoencoder, the unsupervised CNN autoencoder training process utilize benign flows. Flows that are poorly reconstructed are considered an anomaly.

### 4.2.4 Supervised convoltional neural network

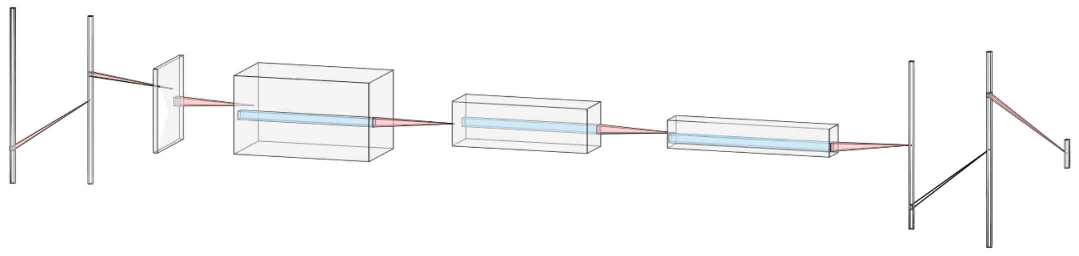


Figure 4.4: The proposed CNN Autoencoder

Finally, as the last model, we propose the supervised convolutional neural network (CNN). It takes a 77-dimensional input and outputs a 2-dimensional vector working the same way as in the supervised dense neural network. After the input layer, there is a dense layer with 64 neurons that we reshape into an (8x8) image. Following the dense layer are three convolutional layers, each with a kernel size of (2,2). The last convolutional layer is then flattened and connected to a dense layer with 64 neurons. Lastly, we connect the dense layer to the 2-dimensional output layer that utilizes the Sigmoid activation function. We use the same hyperparameters as in the convolutional autoencoder. Like the model in Section 4.2.2, undersampling is used to acquire a balanced dataset of benign and fraudulent flows.

## 4.3 Explaining features using SHAP values

We devise Algorithm 1 to utilize the SHAP framework to explain an unsupervised model and find what features it weighs the most. Algorithm 1 is based on the work from [7].

**Algorithm 1:** Find the m most impacting features using SHAP values

```
Input: X - A dataset containing benign flows, S - A dataset containing 20
       fraudulent flows, and $f$ - the model
Output: topMfeatures - the top six impacting features
/* The procedure is slightly different for the supervised models   */
S' ← f(S)
mses = MSE(S,S')                                      // Column-wise MSE
selected_indicies ← top_6_highest_mse_cols(mses)
permuted_data ← shap.sample(X, 100)
explainer ← shap.KernelExplainer(model, permuted_data)
shap_values ← explainer.shap_values(S)
top_m_features ← { }                                  // An empty dictionary
for each feature i ∈ selected_indicies do
    top_three_indicies ← find_top_three_SHAP_columns(shap_values, i)
    for j in top_three_indicies do
        top_m_features[j] += 1
m_most_important_feautres = find_m_largest_values_indicies(top_m_feautres, m)
return m_most_important_feautres
```

The algorithm works by first finding the six most poorly reconstructed columns. Then we permute 100 data points from the original dataset $X$ to form the background data for the explainer. We want to make a permuted dataset from $X$, because we want to capture important aspects of the data using only 100 data points. If we were to increase the number of background samples from 100, it would considerably increase the computation time of the SHAP explainer. Lundberg et al.[25] provide a function inside their framework to permute the data. After the permutation, we want to initialize the explainer using the permuted data. Using the explainer, we find the SHAP values from the fraudulent dataset S. Since S contains 77 columns, then the SHAP values variable has a dimension of 77x77. Thus, each column in S has its own set of SHAP values. To determine the top $m$ impacting features, we iterate through each of the six most poorly reconstructed columns and find the top three highest impacting features for that column. When a column is a part of another column's top three features, it gains 1 point. The top $m$ features with the highest amount of points get selected as the $m$ most important features. Note that the process is considerably easier for the supervised models. We know that the second dimension of the output vector corresponds to the flow being an attack. We simply use the SHAP explainer and its explanations directly on the output vector's second dimension and find the top $m$ most impacting features.

**Example run on an NTP attack**

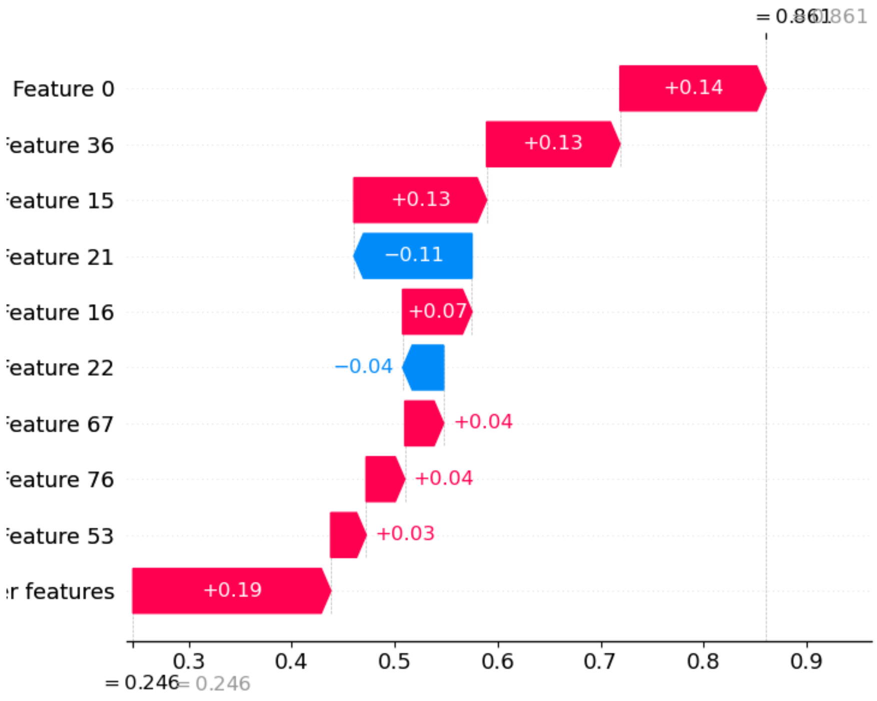


Figure 4.5: Example SHAP values for an NTP attack

Using Algorithm 1 and examining 20 anomalies from the NTP reflection attack, we can see that the top six most influential features for these particular anomalies are Protocol, Fwd Pkts/s, Flow Pkts/s, Fwd IAT Mean, Flow IAT Mean, Flow IAT Std. Each row in the plot tells us how much the features have contributed to the model's prediction. In the bottom left of the plot, one can observe what value to expect from a benign flow, while in the top right corner is the predicted value for the current flow.

## 4.4 Training and performance on unmodified attacks

### 4.4.1 Unsuperivsed CNN Autoencoder

The candidate hyperparameters of this model are the learning rate $\alpha \in \{$ 0.01, 0.001, 0.0005, 0.0001 $\}$ , the number of filters in the convolutional layers, $(\omega_1, \omega_2, \omega_3) \in \{$ (128,40,40), (64, 40, 20), (64, 8, 8), (20,4,2) $\}$, *scaler* $\in \{$ MinMax scaler, Standard scaler, Robust scaler $\}$, activation function $f \in \{$ Sigmoid, ReLU, LeakyReLU $\}$ and what reconstruction error threshold to use. In the hyperparameter selection we also attempt to substitute the second convolutional layer with a MaxPooling layer.

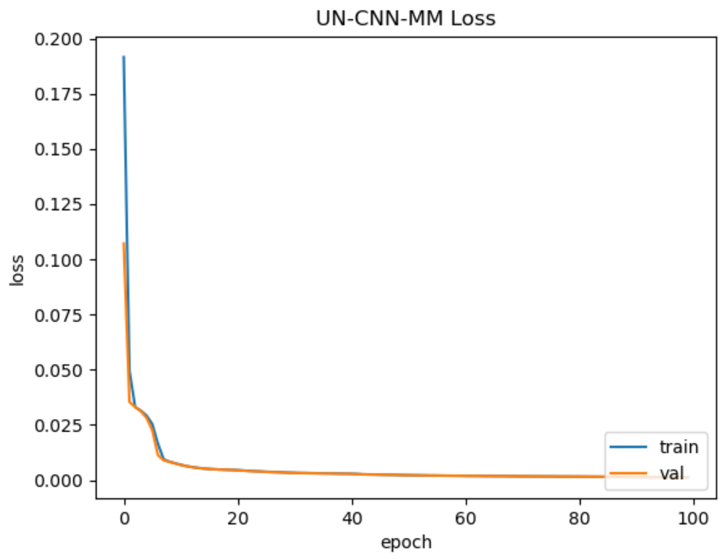


Figure 4.6: Training and validation loss for the optimal model

By observing Figure 4.6, we see that the validation loss follows the training loss. Hence, the model does not seem to overfit. After KFold cross validation, we end up with the following parameters: $\alpha = 0.005$ $f = MinMax$ Scaler, $\omega_1 = 64$, $\omega_3 = 8$, substituting the second convolutional layer with a MaxPooling layer and, and $f = \text{ReLU}$. We also ended up using the interquartile range threshold selection technique defined in Section 3.7.3.

| **Attack Type** | **F1-Score** | **Precision** | **Recall** | **Accuracy** |
|---|---|---|---|---|
| DrDoS NTP | 0.741 | 0.686 | 0.835 | 0.761 |
| DrDoS DNS | 0.474 | 0.351 | 0.870 | 0.611 |
| DrDoS LDAP | 0.286 | 0.186 | 0.884 | 0.535 |
| DrDoS MSSQL | 0.184 | 0.119 | 0.826 | 0.472 |
| DrDoS NetBIOS | 0.279 | 0.182 | 0.874 | 0.528 |
| DrDoS SNMP | 0.222 | 0.152 | 0.780 | 0.466 |
| DrDoS SSDP | 0.626 | 0.569 | 0.752 | 0.661 |
| DrDoS UDP | 0.649 | 0.521 | 0.914 | 0.718 |
| UDPLag | 0.038 | 0.023 | 0.823 | 0.423 |
| TFTP | 0.000 | 0.000 | 1.000 | 0.500 |

Table 4.1: Training day performance for the unsupervised CNN autoencoder

There were made multiple attempts to increase the performance by using contrastive learning, for instance[36]. Unfortunately, the attempt to use contrastive learning degraded the performance of the model. There were also attempts to use regularization and dropout, where both did not increase the performance, indicating that the benign and fraudulent traffic has many similarities.

### 4.4.2 Supervised CNN

The candidate hyperparameters of this model are the learning rate $\alpha \in \{$ 0.01, 0.001, 0.0005, 0.0001 $\}$ , the number of filters in the convolutional layers, $(\omega_1, \omega_2, \omega_3) \in \{$ (128,40,40), (64, 40, 20), (64, 8, 8), (20,4,2) $\}$, *scaler* $\in \{$ MinMax scaler, Standard scaler, Robust scaler $\}$, and activation function $f \in \{$ Sigmoid, ReLU, LeakyReLU $\}$. In the hyperparameter selection we also attempt to substitute the second convolutional layer with a MaxPooling layer.

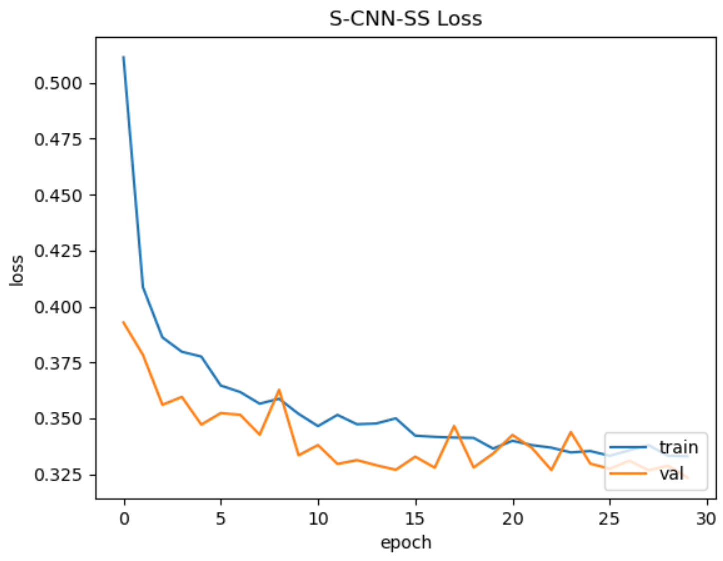


Figure 4.7: Training and validation loss for the optimal model

There are no discrepancies between the training and validation loss, so it seems like the optimized model did not overfit. After KFold cross validation, we end up with the following parameters: $\alpha = 0.001$ $f =$ *Standard* Scaler, $\omega_1 = 64$, $\omega_2 = 40$ and $\omega_3 = 20$, and $f =$ LeakyReLU.

| Attack Type | F1-Score | Precision | Recall | Accuracy |
|---|---|---|---|---|
| DrDoS NTP | 0.995 | 0.995 | 0.996 | 0.995 |
| DrDoS DNS | 0.820 | 0.817 | 0.825 | 0.821 |
| DrDoS LDAP | 0.267 | 0.186 | 0.791 | 0.488 |
| DrDoS MSSQL | 0.756 | 0.867 | 0.572 | 0.720 |
| DrDoS NetBIOS | 0.667 | 0.904 | 0.192 | 0.548 |
| DrDoS SNMP | 0.633 | 0.685 | 0.521 | 0.603 |
| DrDoS SSDP | 0.829 | 0.982 | 0.615 | 0.798 |
| DrDoS UDP | 0.807 | 0.946 | 0.603 | 0.774 |
| UDPLag | 0.804 | 0.782 | 0.836 | 0.809 |
| TFTP | 0.886 | 0.940 | 0.819 | 0.879 |

Table 4.2: Training day performance for the supervised CNN

The supervised CNN is very good at detecting attacks but needs further fine-tuning to capture the underlying structure of benign traffic.

### 4.4.3 Dense Autoencoder

The candidate hyperparameters of this model are $\alpha = \{0.01, 0.001, 0.0005, 0.0001\}$, the $d_1$ and $d_2$ pairs $\{$ $(120, 60), (60, 40), (40, 20), (20, 10)$, $s$ = {MinMax scaler, Standard scaler, Robust scaler}, $f$ = { Sigmoid, ReLU, LeakyReLU }, and how to decide the threshold.

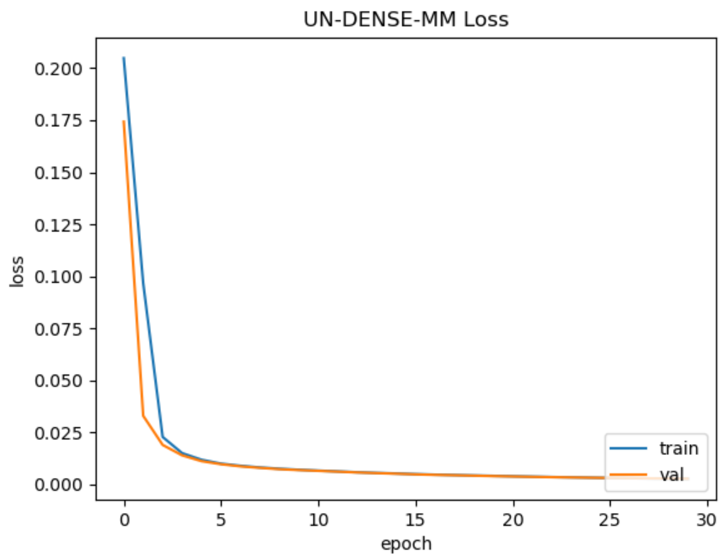


Figure 4.8: Training and validation loss for the optimal model

After KFold cross validation, we end up with the following parameters: $\alpha = 0.0001$ $f = MinMax$ Scaler, $d_1 = 20$, and $d_2 = 10$ and $f$ = LeakyReLU. We ended up using the interquartile range to determine the threshold.

| Attack Type | F1-Score | Precision | Recall | Accuracy |
|---|---|---|---|---|
| DrDoS NTP | 0.741 | 0.686 | 0.835 | 0.761 |
| DrDoS DNS | 0.474 | 0.351 | 0.870 | 0.611 |
| DrDoS LDAP | 0.286 | 0.186 | 0.884 | 0.535 |
| DrDoS MSSQL | 0.184 | 0.119 | 0.826 | 0.472 |
| DrDoS NetBIOS | 0.279 | 0.182 | 0.874 | 0.528 |
| DrDoS SNMP | 0.222 | 0.152 | 0.780 | 0.466 |
| DrDoS SSDP | 0.626 | 0.569 | 0.752 | 0.661 |
| DrDoS UDP | 0.649 | 0.521 | 0.914 | 0.718 |
| UDPLag | 0.038 | 0.023 | 0.823 | 0.423 |
| TFTP | 0.000 | 0.000 | 1.000 | 0.500 |

Table 4.3: Training day performance for the dense autoencoder

The dense autoencoder does not seem to overfit, and it converges. Like the CNN Autoencoder, an attempt was made to apply the contrastive learning technique[36] without yielding better results. The dense autoencoder is best at predicting benign flows and struggles to get a good detection rate.

### 4.4.4 Dense Supervised

The candidate hyperparameters of this model are $\alpha = \{0.01, 0.001, 0.0005, 0.0001\}$, the $d_1$ and $d_2$ pairs $\{(120, 60), (60, 40), (40, 20), (20, 10)\}$, $f = \{$MinMax scaler, Standard scaler, Robust scaler$\}$, and $f = \{$ Sigmoid, ReLU, LeakyReLU $\}$ By examining the training and validation loss during the training process, we see that the validation set follows the loss of the training set. We observe that the model does not overfit or underfit.

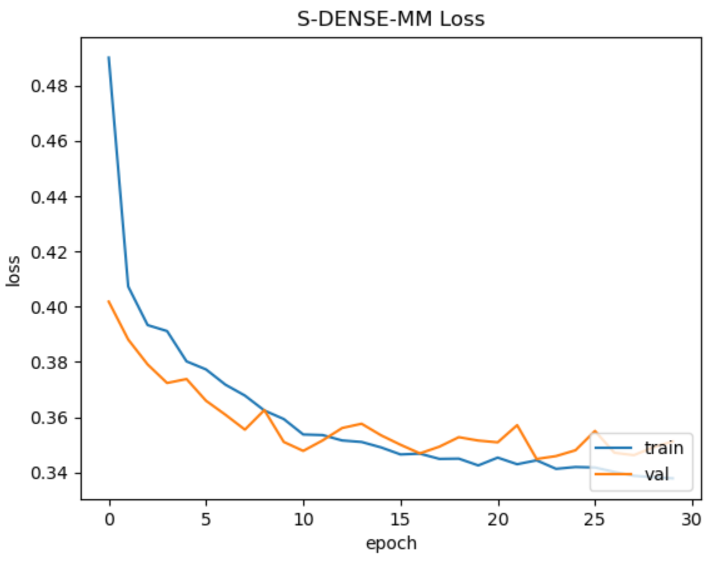


Figure 4.9: Training and validation loss for the optimal model

After KFold cross validation, we end up with the following parameters: $\alpha = 0.001$ $f = MinMax$ Scaler, $d_1 = 40$, and $d_2 = 20$, and $f =$ LeakyReLU.

| **Attack Type** | **F1-Score** | **Precision** | **Recall** | **Accuracy** |
|---|---|---|---|---|
| DrDoS NTP | 0.993 | 0.990 | 0.996 | 0.993 |
| DrDoS DNS | 0.818 | 0.813 | 0.827 | 0.820 |
| DrDoS LDAP | 0.267 | 0.186 | 0.791 | 0.488 |
| DrDoS MSSQL | 0.754 | 0.842 | 0.610 | 0.726 |
| DrDoS NetBIOS | 0.599 | 0.749 | 0.246 | 0.498 |
| DrDoS SNMP | 0.636 | 0.685 | 0.531 | 0.608 |
| DrDoS SSDP | 0.815 | 0.972 | 0.587 | 0.780 |
| DrDoS UDP | 0.803 | 0.938 | 0.603 | 0.770 |
| UDPLag | 0.804 | 0.782 | 0.837 | 0.810 |
| TFTP | 0.933 | 0.905 | 0.966 | 0.935 |

Table 4.4: Training day performance for the dense supervised model

The dense supervised model ended up performing below what to expect in an intrusion detection system. However, it got the best model score out of the four proposed models.

### 4.4.5 Base results

The performance of the supervised model was overall considerably better than the unsupervised ones. However, the unsupervised models did a better job at predicting benign flows. The results indicate that the models are not good enough to solely be used in an intrusion detection system due to inaccuracy and the number of false positives. By observing the results of the supervised models, it is easy to see that they fail to model benign traffic properly. When examining the unsupervised models, the same applies, but to a lesser degree. The autoencoder seems to fail in its task to find underlying structures in benign traffic. Consequently, the unsupervised models reconstruct both benign and fraudulent traffic equally poorly, making it hard to distinguish between them.

## 4.5 Modified attacks

### 4.5.1 Identifying controllable features

To experiment with modifications of attacks, we develop a sandboxing environment where we have to make some assumptions. We are unable to modify some parameters, and there are hard limits on packet size, idling before timeout, throughput, and how many flags we can send, for instance. To the best extent, we will try to abide by the theoretical boundaries. Below are some remarks on which columns are interesting to manipulate. See the appendix for more details about each feature in the dataset.

- Flow duration - Since we can delay / increase when the connection comes to a close this is a interesting parameter to manipulate

- Since we can delay/increase when the connection comes to a close, this is an interesting parameter to manipulate
- Backward(bwd) parameters are somewhat more difficult to analyze since each server may react differently, and thus we are unable to control it in the same way as the fwd parameters
- Flag counts can, to a smaller extent, be manipulated

### 4.5.2 How to modify an attack

It is infeasible to try all combinations of changes. The solution was to separate into three different experiments. The most important features across the models are used as a base when designing the experiments. Each model explanation from the day 2-attacks is located in **Appendix E**.

**Experiment number one** is about increasing the flow duration. However, all parameters that can be preserved are unchanged. The goal is to see if we can manage to increase the total data sent to the victim. We choose to modify flow duration because SHAP did not detect it as important in any of the models. It is also an uncomplicated parameter to manipulate.

**Experiment number two** is about increasing the number of packets sent while keeping the same amount of bytes per second. To achieve a higher number of packets and preserve throughput, we need to divide packets into a smaller size and send an increased amount of traffic with the smaller packets. The goal is to see if we can manage to maintain throughput while concealing the traffic.

**Experiment number three** is about maintaining throughput by decreasing packets sent and increasing packet size and all related parameters. The goal of this experiment is to test the reverse of experiment two.

| Feature Name | Feature dependencies |
|---|---|
| flow duration | total fwd packet, total bwd packets, total length of fwd packet, total length of bwd packet, fwd header length, bwd header length, Flow IAT Mean, Flow IAT Std, Fwd IAT Mean, Fwd IAT Std, ACK Flag Count |
| flags | none, except ack flag |
| flow packets/s | fwd packets/s, bwd packets/s, total fwd packets, total bwd packets, totlen fwd packets, totlen bwd packets, average packet size, Flow IAT Mean, Flow IAT Std, Fwd IAT Mean, Fwd IAT Std, ACK Flag Count |
| flow bytes/s | flow duration, average packet size, Packet Length Mean, Packet Length Std, Fwd Packet Length Mean, Fwd Packet Length Std |

Now that we have modeled some dependencies, we devise Algorithm 2 to utilize a feature and its dependencies to generate a new dataset with a $p$ percent increased dataset.

**Algorithm 2:** Modify dataset given a modificator p

```
Input: X - A dataset containing benign and fraudulent flows, col - What feature to
       permute, dependencies - Dependencies of the column, which is an array of
       tuple containing the dependency column and if it is negative proportional
       or proportional, p - how many percent modification to apply to the data
Output: X_modified - A modified version of the input data X
1 inverse_factor = $\frac{1}{1+p}$
2 X_modified ← deepcopy(X)
3 X_modified[:,col] = X[:, col]· (1+p)
4 for each (dep_col,prop) ∈ dependencies do
5     if dep_col is negative proportional then
6         X_modified[:, col] = X[:,col] · $inverse\_factor$
7     else
8         X_modified[:, col] = X[:,col] · $(1 + p)$
9 return X_modified
```

**Generating modified datasets** For the different experiments, we will apply changes from 1 to 100%, with a step size of 1%, generating 100 modified datasets. How the models react to the modified dataset will determine its robustness.

## 4.6 Performance on modified attacks

The data we will use for the modified attacks is from day 2 in the CiCDDoS2019 dataset. Day 2 contains the reflection attacks Portmap, NetBIOS, LDAP, MSSQL, UDP, and UDPLag. Before using the data set, we balance it to contain an equal amount of benign and fraudulent flows. For each model, we will evaluate the modified attacks against the base precision. The base precision is calculated before applying any modifications. For each attack, we generate a plot containing a threshold line equalling the base precision and a graph for each experiment conducted. The X-axis denotes the percentage change factor, and the Y-axis denotes the precision. If a point (X, Y) is placed below the threshold, the model was fooled by the modified data set. Note that the UDPLag contained very few samples and thus was omitted from the experiments.

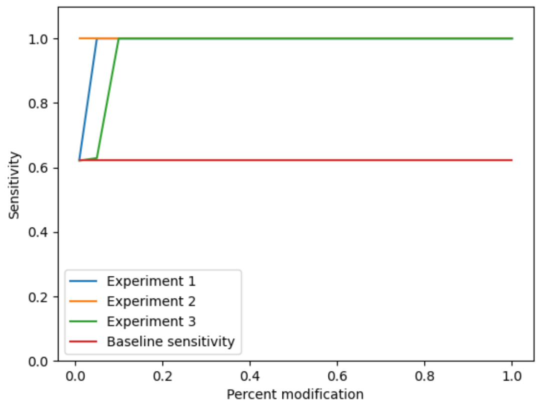
### 4.6.1 Unsupervised CNN Autoencoder

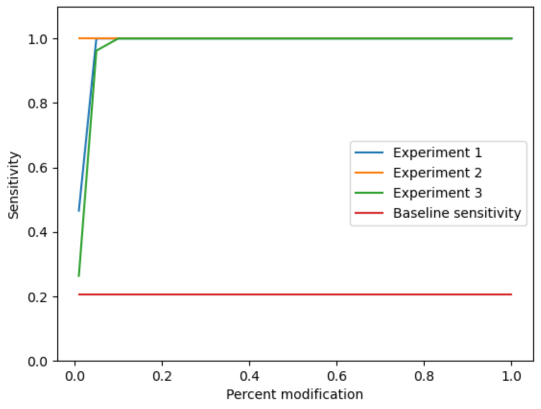
#### Base performance

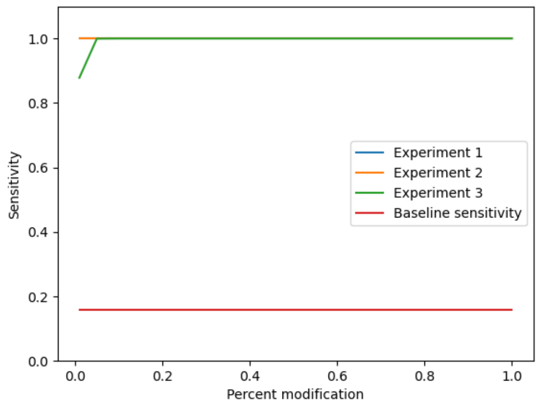
We start by measuring the base performance on the data set. In the table below, we observe a pretty mediocre result. However, the model does a decent job at predicting benign traffic. With the base performance as a threshold, we perform the modified attacks.

| Attack Type | F1-Score | Precision | Recall | Accuracy |
|---|---|---|---|---|
| Portmap | 0.094 | 0.057 | 0.844 | 0.451 |
| NetBIOS | 0.159 | 0.096 | 0.890 | 0.493 |
| LDAP | 0.120 | 0.072 | 0.873 | 0.473 |
| MSSQL | 0.708 | 0.621 | 0.866 | 0.743 |
| UDP | 0.354 | 0.243 | 0.870 | 0.556 |

Table 4.5: Base results for the unsuperivsed CNN autoencoder







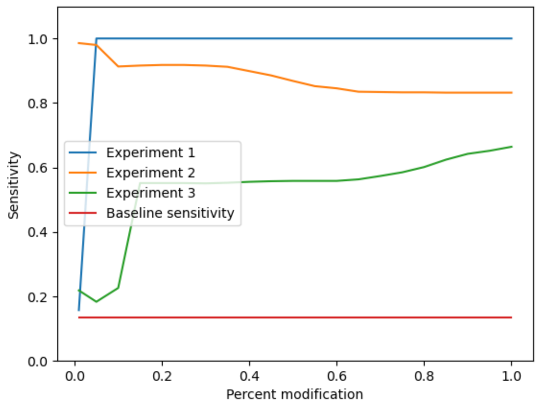


Figure 4.10: Modified attacks for the CNN Autoencoder. MSSQL top left, LDAP top right, NetBIOS bottom left and Portmap bottom right

**Experiment 1** There were not any noticeable results in this experiment. Even to the slightest increase in percent, the model instantly reacted and discovered the modified attacks.

**Experiment 2** Again, there were no significant results.

**Experiment 3** Again, none of the attacks that were modified managed to bypass the autoencoder model.

**Discussion** The CNN autoencoder was very robust against manipulated attacks. However, the detection rate of unmodified attacks was already pretty low, and it allowed plenty of fraudulent traffic to flow through, which is not ideal for an intrusion detection system. If we were to tune the model further and achieve better results on the unmodified attacks, it could be a decent candidate model for an IDS.

### 4.6.2 Supervised CNN

Supervised CNN outperformed the CNN autoencoder in the unmodified attacks section. While it does perform better than the CNN autoencoder on the new unseen data, it is still not good enough.

| Attack Type | F1-Score | Precision | Recall | Accuracy |
|---|---|---|---|---|
| Portmap | 0.099 | 0.052 | 0.988 | 0.520 |
| NetBIOS | 0.192 | 0.134 | 0.745 | 0.440 |
| LDAP | 0.744 | 0.878 | 0.517 | 0.698 |
| MSSQL | 0.899 | 0.958 | 0.827 | 0.893 |
| UDP | 0.045 | 0.027 | 0.818 | 0.423 |

Table 4.6: Base results for the supervised CNN

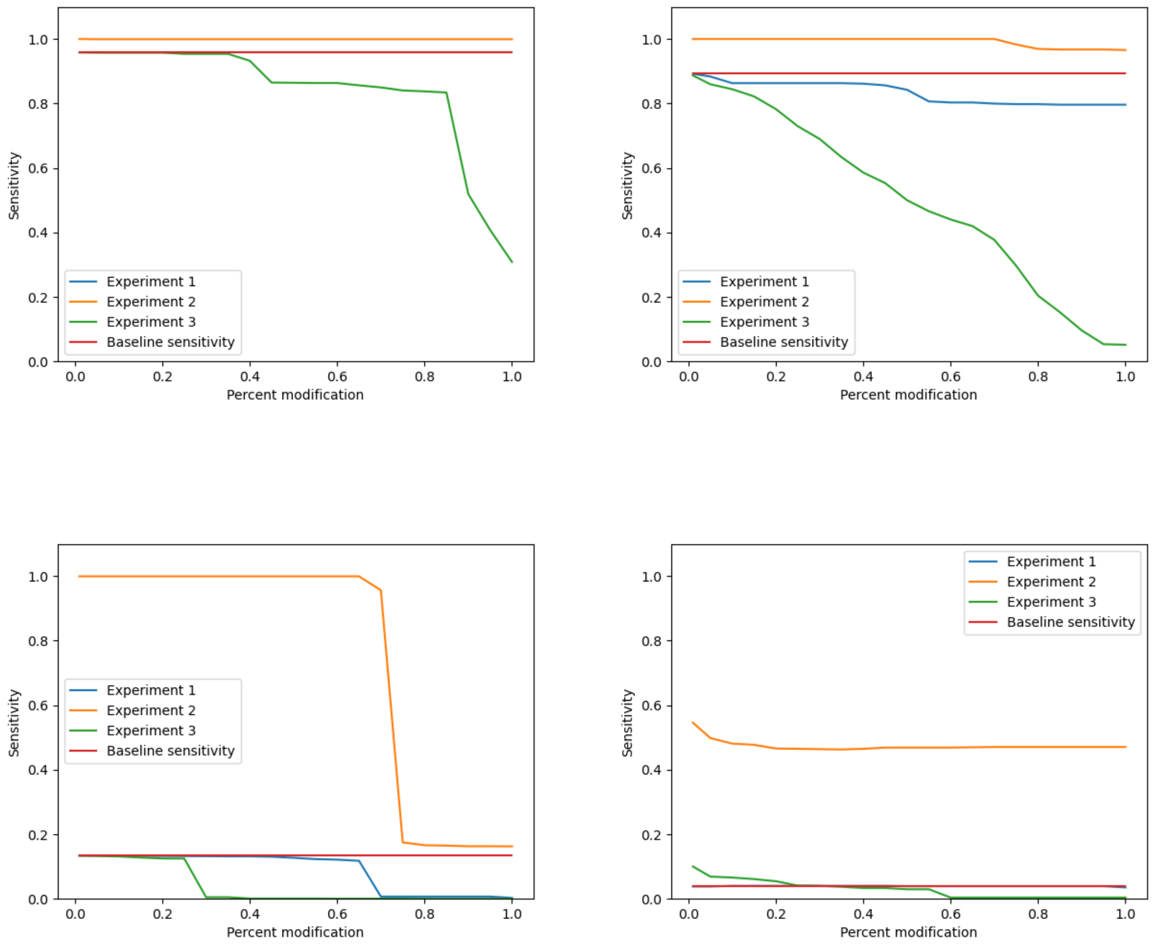


Figure 4.11: Modified attacks for the supervised CNN. MSSQL top left, LDAP top right, NetBIOS bottom left and Portmap bottom right

**Experiment 1** The attacks were successful, and we managed to double the throughput and conceal a greater portion of the flows from the IDS in the LDAP and NetBIOS attacks. The magnitude of the attacks was vastly increased.

**Experiment 2** This experiment did not yield any results.

**Experiment 3** By observing the graphs, the attack was immensely successful. The modification managed to conceal almost every flow from the IDS, thus dramatically increasing the throughput

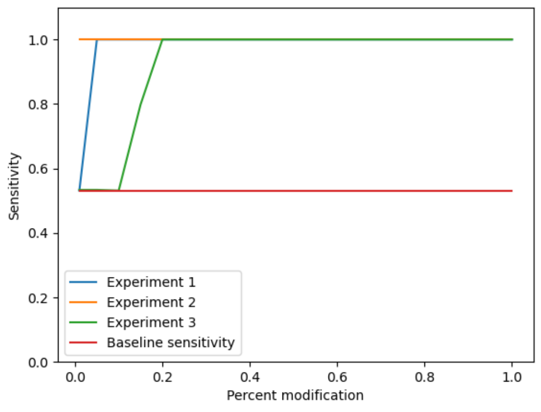
**Discussion** For every experiment apart from number 2, the more we modified the traffic both in increasing the throughput and preserving the throughput, we managed to conceal fraudulent flows. Based only on the pure, unmodified results alone, one would think this model performs well. In reality, the decision boundaries of the model are severely flawed. That we managed to bypass the model is a significant result because it means we can gain an insight from the SHAP values to fool a model. There could be several reasons why it responds so poorly to the modifications. What is certain is that it fails to recognize combinations of parameters that should be inferred as an attack.

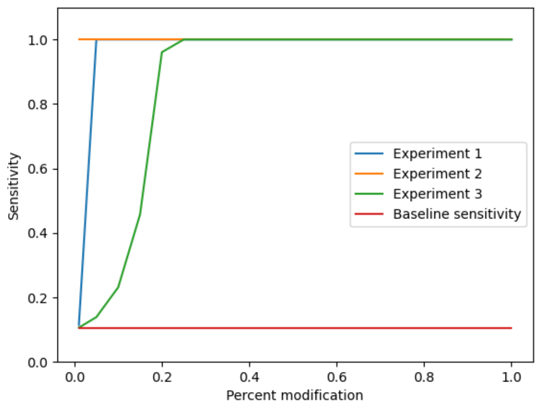
## 4.6.3 Dense Autoencoder

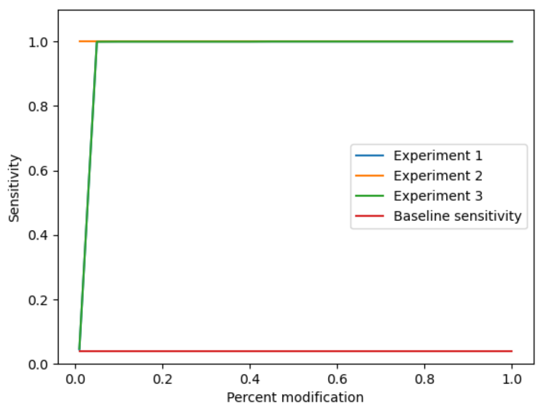
Let's take on the next model and get some base results for the unmodified attacks. Like the Unsupervised CNN it has rather poor performance, but decent results on benign traffic.

| Attack Type | F1-Score | Precision | Recall | Accuracy |
|---|---|---|---|---|
| Portmap | 0.096 | 0.058 | 0.848 | 0.453 |
| NetBIOS | 0.097 | 0.052 | 0.971 | 0.512 |
| LDAP | 0.176 | 0.108 | 0.880 | 0.494 |
| MSSQL | 0.062 | 0.035 | 0.920 | 0.477 |
| UDP | 0.391 | 0.288 | 0.814 | 0.551 |

Table 4.7: Base results for the dense autoencoder







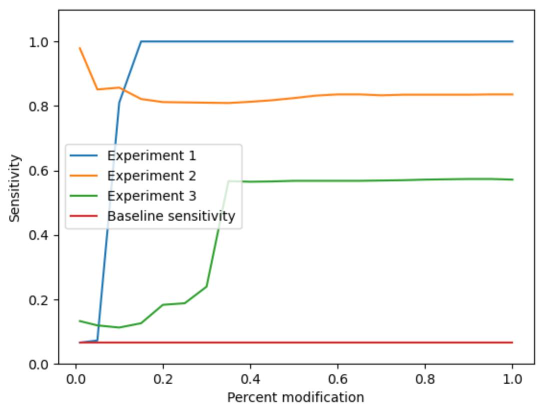


Figure 4.12: Modified attacks for the Dense autoencoder. MSSQL top left, LDAP top right, NetBIOS bottom let and Portmap bottom right

**Experiment 1** There were no considerable results.

**Experiment 2** There were no considerable results.

**Experiment 3** There were no considerable results.

**Discussion** From the graphs above, we observe that the model is robust against modification attacks. Experiment three came closest to yielding any results. However, any modification to the dataset made the detection rate better, which could be due to a wide range of reasons. It seems like the attacking flows are not different enough from the benign traffic to make them go above the reconstruction threshold. Nevertheless, when the fraudulent flows are modified, they get far enough from regular flows to get recognized instantly.

### 4.6.4 Dense Supervised

Similar to the Supervised CNN, the dense supervised model outperforms the two unsupervised approaches. From the results, we observe that detecting DDoS attacks is challenging.

| Attack Type | F1-Score | Precision | Recall | Accuracy |
|---|---|---|---|---|
| Portmap | 0.186 | 0.104 | 0.989 | 0.546 |
| NetBIOS | 0.190 | 0.131 | 0.750 | 0.441 |
| LDAP | 0.764 | 0.921 | 0.509 | 0.715 |
| MSSQL | 0.899 | 0.958 | 0.825 | 0.892 |
| UDP | 0.030 | 0.018 | 0.816 | 0.417 |

Table 4.8: Base results for the dense supervised model

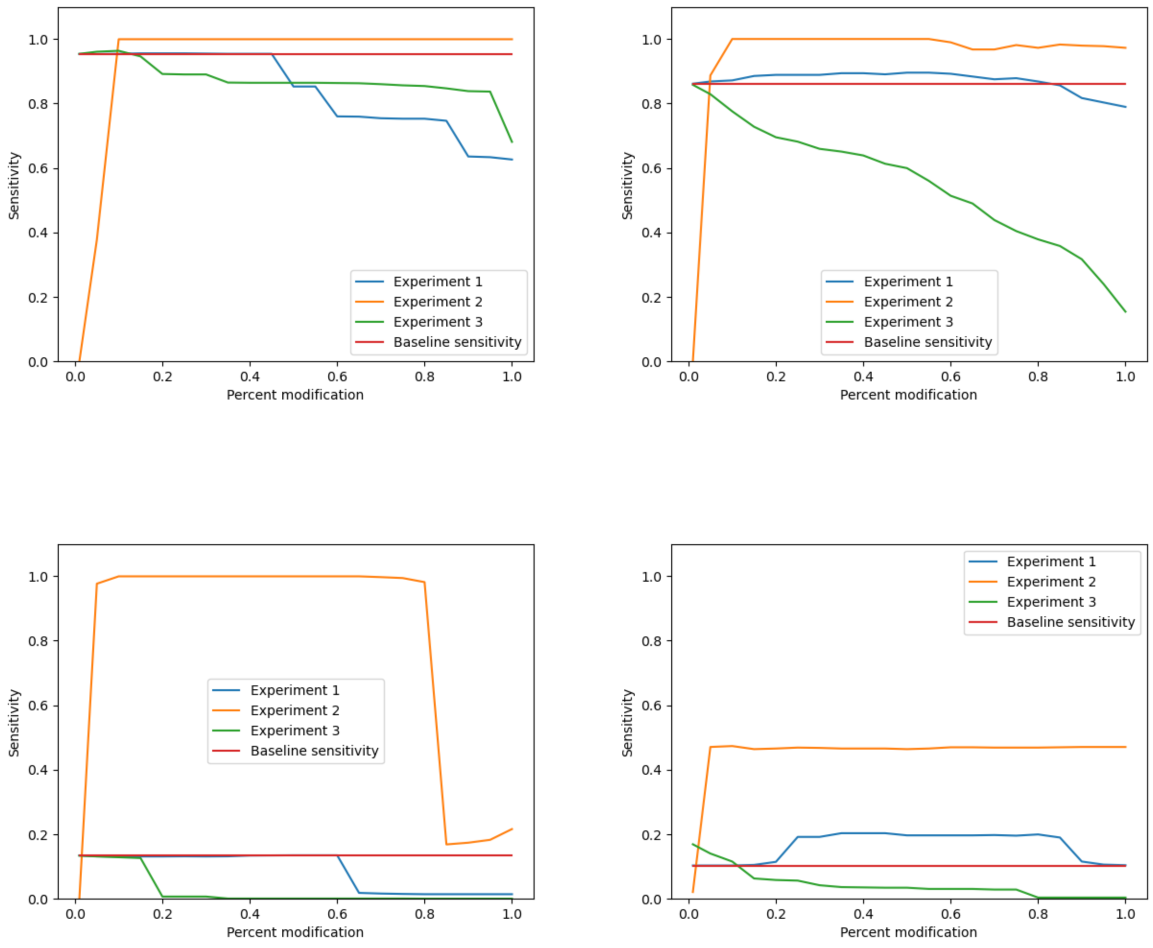


Figure 4.13: MSSQL top left, LDAP top right, NetBIOS bottom let and Portmap bottom right

**Experiment 1** In this experiment, the attempt to modify traffic was successful in all the attacks. In the MSSQL and NetBIOS attacks, we managed to conceal a significant portion of the traffic.

**Experiment 2** There were no considerable results.

**Experiment 3** Lastly, we observe that experiment 3 was very successful. We managed to bypass a considerable amount of flows.

**Discussion** The dense supervised model had one of the best base results and outperformed the unsupervised models. However, the proposed model was highly vulnerable to modification attacks—even more than the supervised CNN. Like the supervised CNN, the decision boundaries have to be severely flawed. In Section 4.4.4 we could see that the dense supervised model struggled in some of the attacks to detect benign flows. The modification attack further highlighted this issue. It is hard to single out why the model performs so incredibly poorly on the modified attacks. There was, unfortunately, no time to research this issue further. Hence, we leave it as future work.

# Chapter 5

# Conclusion

## 5.1 Summary

The primary aim of this thesis was to figure out if we can utilize SHAP values as a base for generating more sophisticated attacks. The main source of data is the University of New Brunswick's DDoS dataset. It has two days worth of attack data, where we used the first day to select and optimize four candidate models. After examining the test performance, the supervised models outperformed the unsupervised model by quite a lot. Although much effort was put in to find high-performing models to use in an IDS, the performance of our models did not have a sufficiently high detection rate to be used in an IDS alone. For each attack on the second day, we recorded the performance per model and determined each of the models most important features. When modifying an attack to bypass a particular model, we apply changes to each feature recognized as important. For the unsupervised models, this approach did not produce any meaningful results. However, for supervised models, we managed to both conceal fraudulent traffic and double the throughput. The supervised models who would be selected as the main model for the IDS were completely disarmed. Hence, utilizing the SHAP framework to modify attacks and bypass intrusion detection systems appears to be possible.

## 5.2 Future work

While modifying attacks worked for supervised models, we did not manage to fool the unsupervised models. Future work could be to test the robustness of unsupervised models further. Additionally, the unsupervised models' performance was not good enough. We need to tweak the unsupervised models to detect a more significant portion of the attacks. Another important future work is setting up a testbed and executing actual modified attacks, confirming our findings. Further, one could try to research defense mechanisms for the supervised IDS to withstand the modified attacks. It is also interesting to attempt the modification attack on another IDS, for instance, Bro[37]. Since our approach to fool intrusion detection systems showed promise, further work could be to translate the attack to other fields, such as credit card fraud detection.

# Appendix

## A: CiCFlowmeter feature descriptions

| **Feature Name** | **Description** |
|---|---|
| Protocol | The protocol of the flow |
| Flow duration | Duration of the flow in Microsecond |
| total Fwd Packet | Total packets in the forward direction |
| total Bwd packets | Total packets in the backward direction |
| total Length of Fwd Packet | Total size of packet in forward direction |
| total Length of Bwd Packet | Total size of packet in backward direction |
| Fwd Packet Length Min | Minimum size of packet in forward direction |
| Fwd Packet Length Max | Maximum size of packet in forward direction |
| Fwd Packet Length Mean | Mean size of packet in forward direction |
| Fwd Packet Length Std | Standard deviation size of packet in forward direction |
| Bwd Packet Length Min | Minimum size of packet in backward direction |
| Bwd Packet Length Max | Maximum size of packet in backward direction |
| Bwd Packet Length Mean | Mean size of packet in backward direction |
| Bwd Packet Length Std | Standard deviation size of packet in backward direction |
| Flow Bytes/s | Number of flow bytes per second |
| Flow Packets/s | Number of flow packets per second |
| Flow IAT Mean | Mean time between two packets sent in the flow |
| Flow IAT Std | Standard deviation time between two packets sent in the flow |
| Flow IAT Max | Maximum time between two packets sent in the flow |
| Flow IAT Min | Minimum time between two packets sent in the flow |
| Fwd IAT Min | Minimum time between two packets sent in the forward direction |
| Fwd IAT Max | Maximum time between two packets sent in the forward direction |
| Fwd IAT Mean | Mean time between two packets sent in the forward direction |

| | |
|---|---|
| Fwd IAT Std | Standard deviation time between two packets sent in the forward direction |
| Fwd IAT Total | Total time between two packets sent in the forward direction |
| Bwd IAT Min | Minimum time between two packets sent in the backward direction |
| Bwd IAT Max | Maximum time between two packets sent in the backward direction |
| Bwd IAT Mean | Mean time between two packets sent in the backward direction |
| Bwd IAT Std | Standard deviation time between two packets sent in the backward direction |
| Bwd IAT Total | Total time between two packets sent in the backward direction |
| Fwd PSH flags | Number of times the PSH flag was set in packets travelling in the forward direction (0 for UDP) |
| Bwd PSH Flags | Number of times the PSH flag was set in packets travelling in the backward direction (0 for UDP) |
| Fwd URG Flags | Number of times the URG flag was set in packets travelling in the forward direction (0 for UDP) |
| Bwd URG Flags | Number of times the URG flag was set in packets travelling in the backward direction (0 for UDP) |
| Fwd Header Length | Total bytes used for headers in the forward direction |
| Bwd Header Length | Total bytes used for headers in the backward direction |
| FWD Packets/s | Number of forward packets per second |
| Bwd Packets/s | Number of backward packets per second |
| Packet Length Min | Minimum length of a packet |
| Packet Length Max | Maximum length of a packet |
| Packet Length Mean | Mean length of a packet |
| Packet Length Std | Standard deviation length of a packet |
| Packet Length Variance | Variance length of a packet |
| FIN Flag Count | Number of packets with FIN |
| SYN Flag Count | Number of packets with SYN |
| RST Flag Count | Number of packets with RST |
| PSH Flag Count | Number of packets with PUSH |
| ACK Flag Count | Number of packets with ACK |
| URG Flag Count | Number of packets with URG |
| CWR Flag Count | Number of packets with CWR |
| ECE Flag Count | Number of packets with ECE |
| Down/Up Ratio | Download and upload ratio |
| Average Packet Size | Average size of packet |
| Fwd Segment Size Avg | Average size observed in the forward direction |

| | |
|---|---|
| Bwd Segment Size Avg | Average number of bytes bulk rate in the backward direction |
| Fwd Bytes/Bulk Avg | Average number of bytes bulk rate in the forward direction |
| Fwd Packet/Bulk Avg | Average number of packets bulk rate in the forward direction |
| Fwd Bulk Rate Avg | Average number of bulk rate in the forward direction |
| Bwd Bytes/Bulk Avg | Average number of bytes bulk rate in the backward direction |
| Bwd Packet/Bulk Avg | Average number of packets bulk rate in the backward direction |
| Bwd Bulk Rate Avg | Average number of bulk rate in the backward direction |
| Subflow Fwd Packets | The average number of packets in a sub flow in the forward direction |
| Subflow Fwd Bytes | The average number of bytes in a sub flow in the forward direction |
| Subflow Bwd Packets | The average number of packets in a sub flow in the backward direction |
| Subflow Bwd Bytes | The average number of bytes in a sub flow in the backward direction |
| Fwd Init Win bytes | The total number of bytes sent in initial window in the forward direction |
| Bwd Init Win bytes | The total number of bytes sent in initial window in the backward direction |
| Fwd Act Data Pkts | Count of packets with at least 1 byte of TCP data payload in the forward direction |
| Fwd Seg Size Min | Minimum segment size observed in the forward direction |
| Active Min | Minimum time a flow was active before becoming idle |
| Active Mean | Mean time a flow was active before becoming idle |
| Active Max | Maximum time a flow was active before becoming idle |
| Active Std | Standard deviation time a flow was active before becoming idle |
| Idle Min | Minimum time a flow was idle before becoming active |
| Idle Mean | Mean time a flow was idle before becoming active |
| Idle Max | Maximum time a flow was idle before becoming active |
| Idle Std | Standard deviation time a flow was idle before becoming active |

# B: Variable definitions

| **Variable** | **Description** |
|---|---|
| $X$ | Some data set consisting of $n$ entries |
| $X_i$ | Entry $i$ in the data set X consisting of k features |
| $X'_i$ | An estimation of $X_i$ |
| $X_{min}$ | A vector consisting of the minimum value for each column |
| $X_{max}$ | A vector consisting of the maximum value for each column |
| $y$ | Some vector with corresponding labels to some data set X |
| $y_i$ | The $i$'th label corresponding to entry $X_i$ |
| $y'_i$ | A label prediction of $X_i$ with the true value $y_i$ |
| $x$ | Some data point in some data set consisting of k features |
| $x_i$ | Feature $i$ of the data point $x$ |
| $f$ | Some activation function $f$ like ReLU, Sigmoid or LeakyReLU |
| $z$ | Some neuron activation value |
| $\xi$ | A surrogate model trained on some instance $x$ |
| $\mathcal{L}$ | Loss function |
| $m$ | Some machine learning model |
| $g$ | An explanation model for some instance $x$ |
| $G$ | Family of all possible explanations |
| $w^x$ | Weight between sampled data and original data in the surrogate model training process |
| $\phi$ | The Shapley values for all features |
| $\phi_i$ | The Shapley values for the feature $i$ |
| $M$ | The maximum coalition size |
| $z'$ | The coalition vector. $z' \in \{0, 1\}^M$ |
| $\beta_i$ | The ith slope in a linear regressor |
| $\epsilon_i$ | The irreducible error of the ith datapoint |
| $\gamma, \alpha, \epsilon$ | Optimizer parameters |
| $w^{(t)}, b^{(t)}$ | The biases and weights of the t'th iteration |

# C: Attempted models

## Supervised models

- Convolution neural network[2]
- Dense neural network [3]
- XGBoost[38]
- Random forest [39]

## Unsupervised models

- Generative Adversarial Network(GAN)[35]
- AnoGAN[40]
- LSTM Autoencoders [33]
- Convolution-Deconvolution Neural Network [41]
- Dense Autoencoders [3]
- Isolation forest [42]

## D: Figure sources

- Figure 3.3 - Underfitting and Overfitting.
  Available from: https://subscription.packtpub.com/book/data/9781838556334/7/ch07lvl1sec82/underfitting-and-overfitting
- Figure 3.5 - Linear regression
  Available from: https://community.cloudera.com/t5/Community-Articles/Understanding-Linear-Regression/ta-p/281391
- Figure 3.8: 2D Convolution block. Available from: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/2d-convolution-block
- Figure 3.9: Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Illustration-of-Max-Pooling-and-Average-Pooling-Figure-2-above-shows-an-example-of-max_fig2_333593451
- Figure 3.10 - Transposed Convolution Demystified
  Available from: https://towardsdatascience.com/transposed-convolution-demystified-84ca81b4baba

Throughout the thesis we created figures with draw.io and alexlenail.me/NN-SVG/AlexNet.html to visualize the proposed models.

# E: SHAP explanations for the day 2 attacks

## Supervised CNN

### LDAP

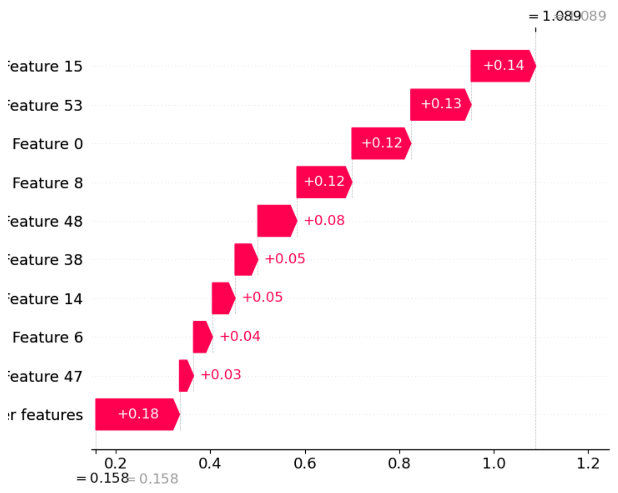


Figure 5.1: LDAP Explanation

### MSSQL

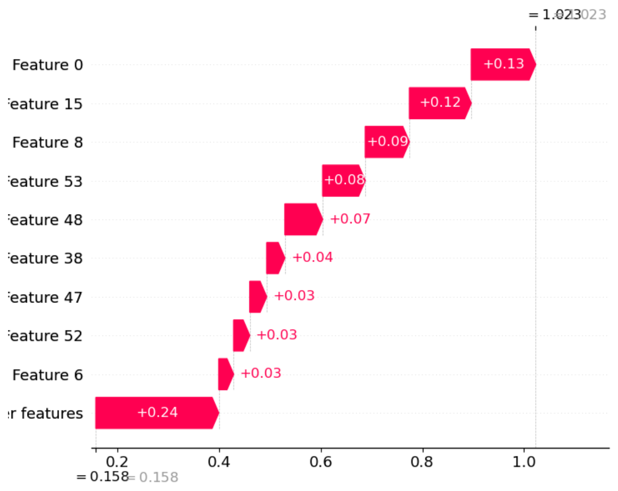


Figure 5.2: MSSQL Explanation

## Portmap

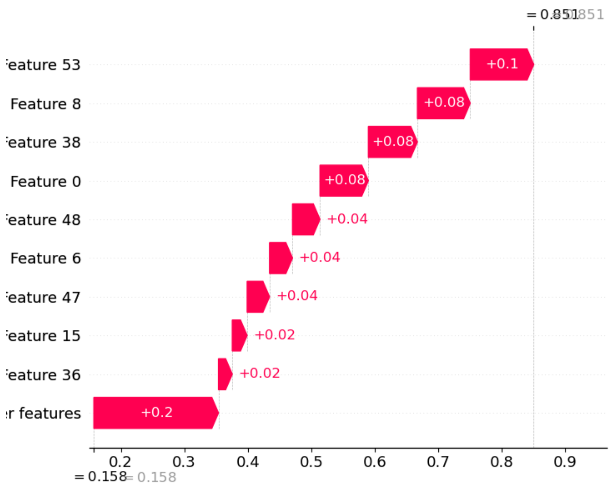


Figure 5.3: Portmap Explanation

## UDP

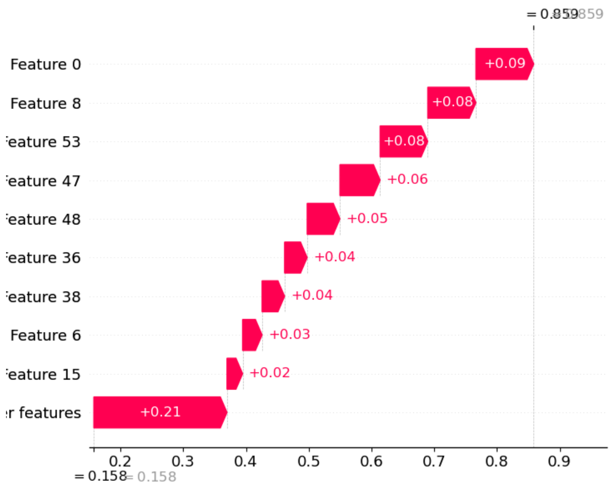


Figure 5.4: LDAP Explanation

## Supervised dense neural network

### LDAP

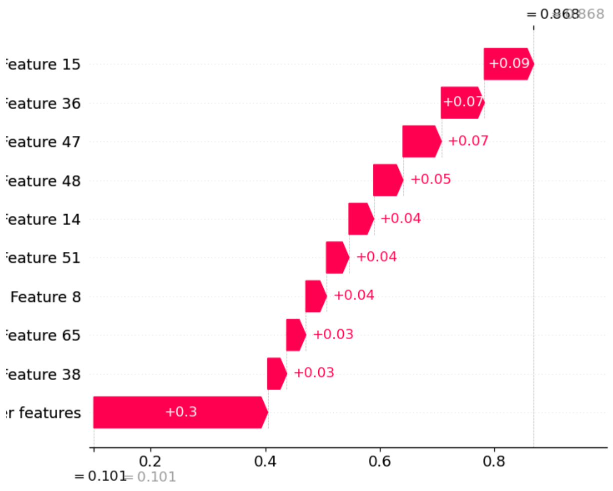


Figure 5.5: LDAP Explanation

### MSSQL

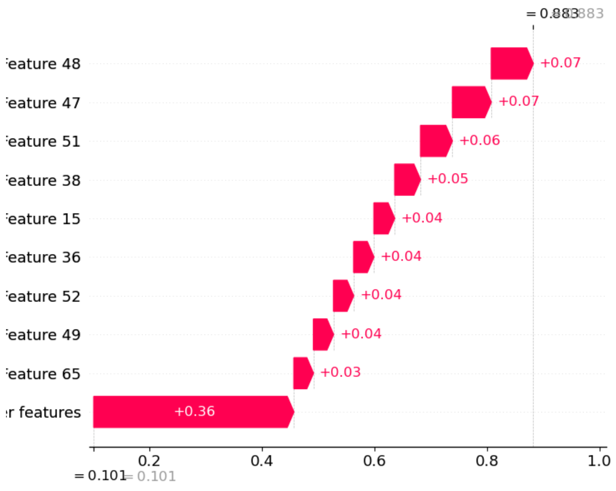


Figure 5.6: MSSQL Explanation

## Portmap

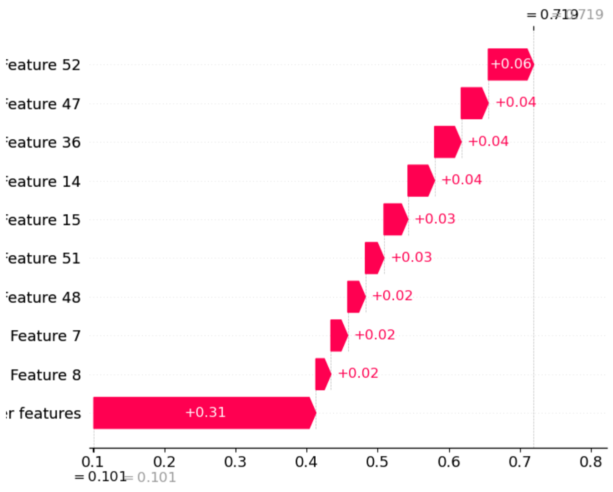


Figure 5.7: Portmap Explanation

## UDP

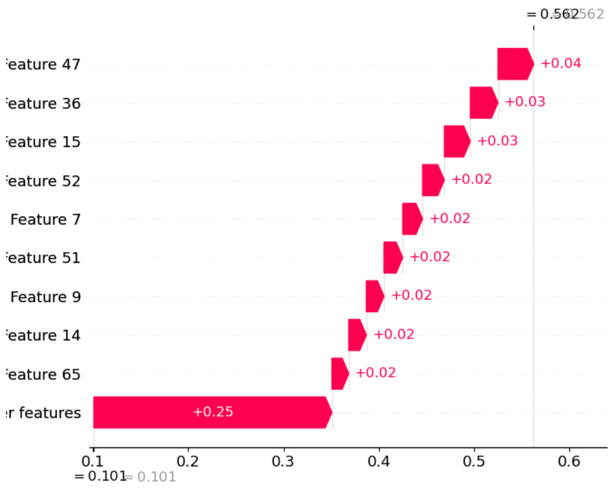


Figure 5.8: LDAP Explanation

# Unsupervised CNN autoencoder

## LDAP

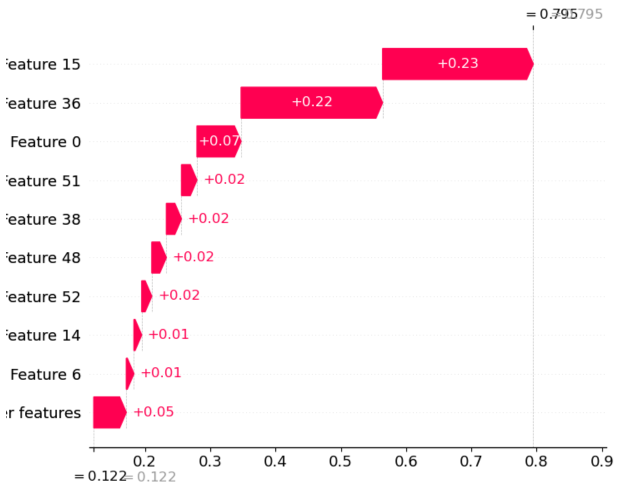


Figure 5.9: LDAP Explanation

## MSSQL

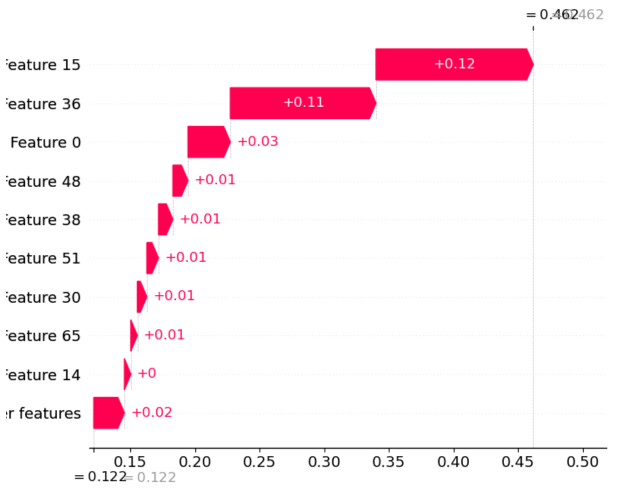


Figure 5.10: MSSQL Explanation

## Portmap

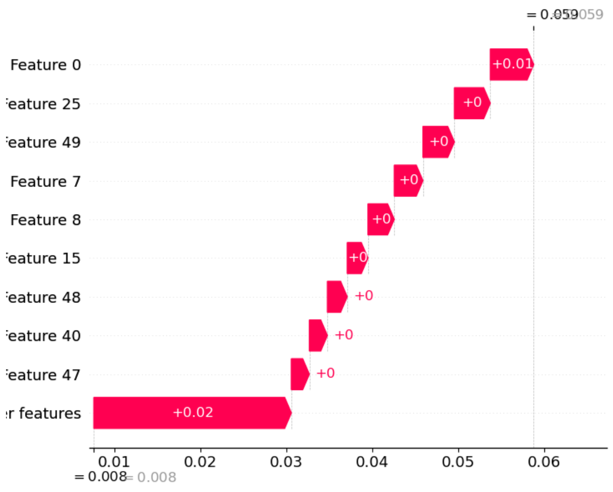


Figure 5.11: Portmap Explanation

## UDP

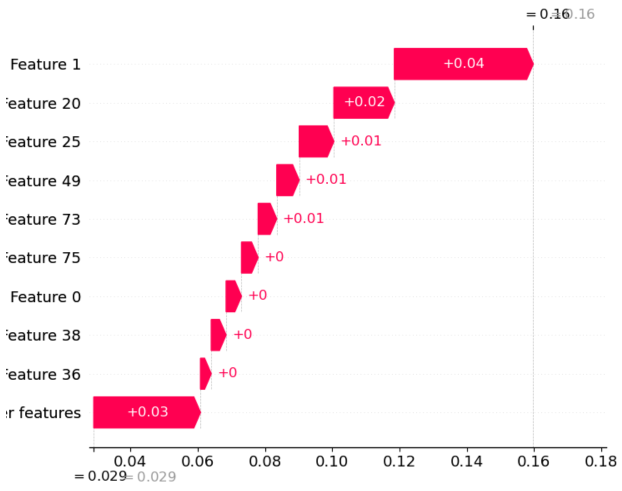


Figure 5.12: LDAP Explanation

## Unsupervised dense autoencoder

### LDAP

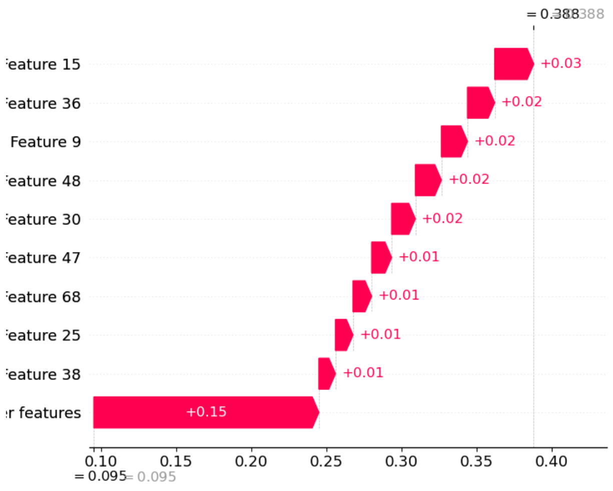


Figure 5.13: LDAP Explanation

### MSSQL

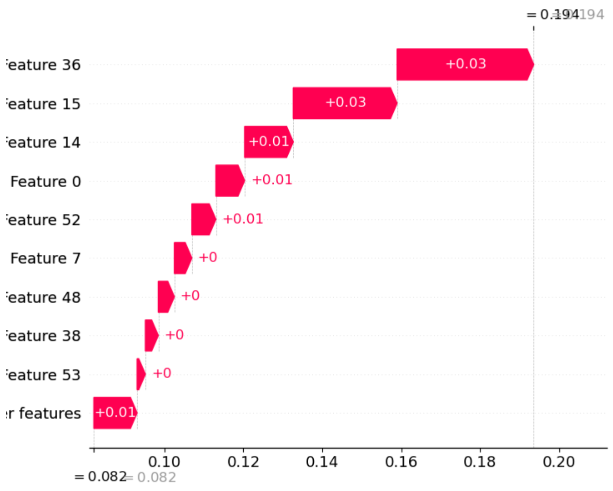


Figure 5.14: MSSQL Explanation

## Portmap

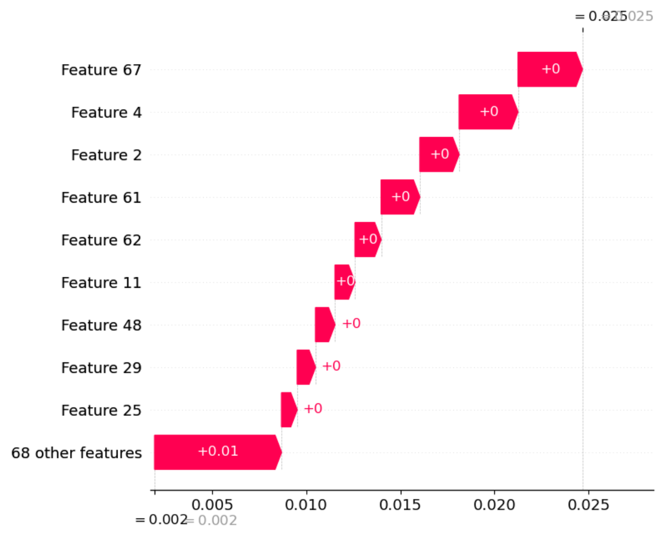


Figure 5.15: Portmap Explanation

## UDP

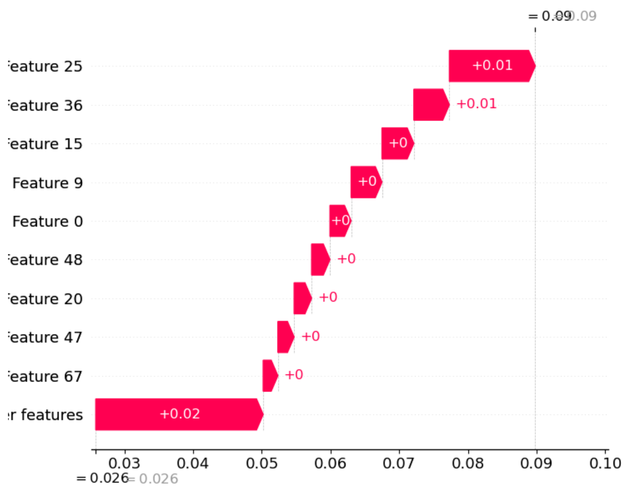


Figure 5.16: LDAP Explanation

## Unsupervised dense autoencoder

### LDAP

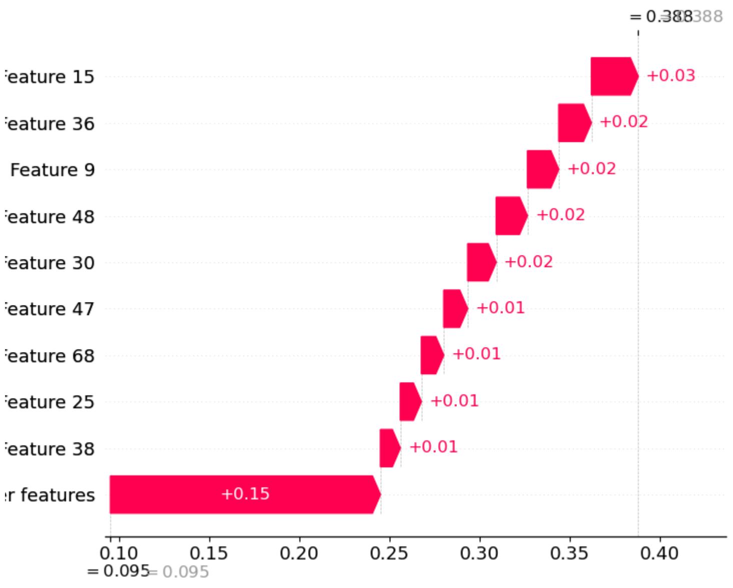


Figure 5.17: LDAP Explanation

## MSSQL

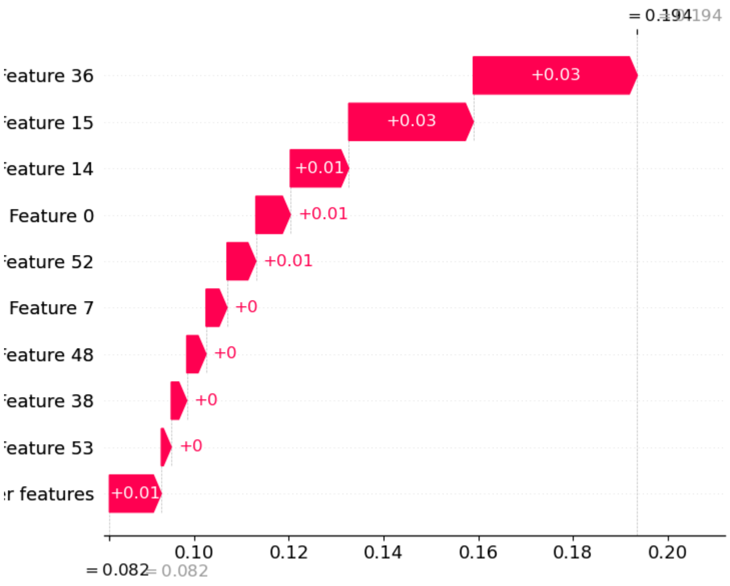


Figure 5.18: MSSQL Explanation

## Portmap

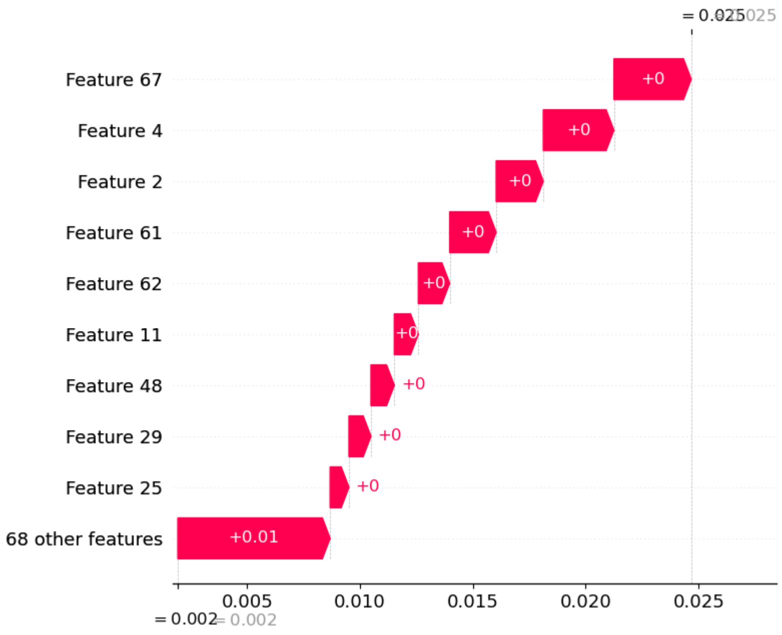


Figure 5.19: Portmap Explanation

## UDP

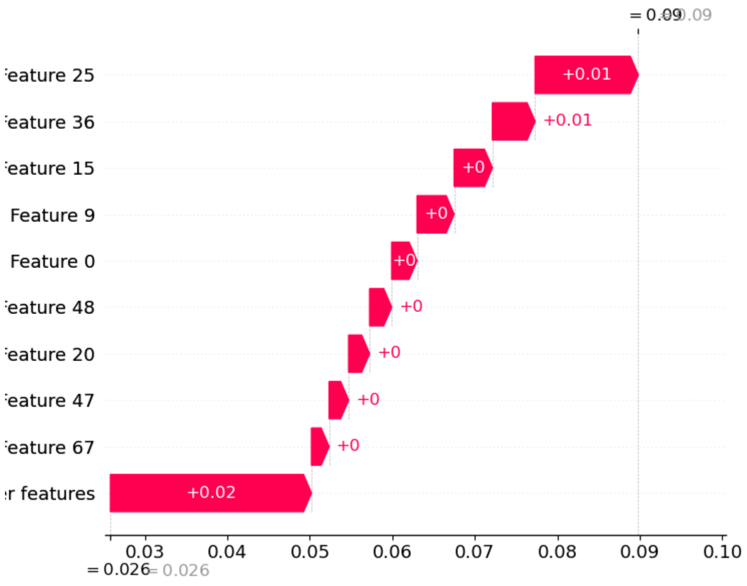


Figure 5.20: LDAP Explanation

# References

[1] Cisco. *Cisco Annual Internet Report (2018–2023) White Paper*. 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[2] Jiyeon Kim et al. "CNN-Based Network Intrusion Detection against Denial-of-Service Attacks." In: *Electronics* (2020), p. 916. DOI: 10.3390/electronics9060916.

[3] Fahimeh Farahnakian and Jukka Heikkonen. "A deep auto-encoder based approach for intrusion detection system." In: Feb. 2018, pp. 178–183. DOI: 10.23919/ICACT.2018.8323688.

[4] Amjad Al-Tobi and Ishbel Duncan. "KDD 1999 generation faults: a review and analysis." In: *Journal of Cyber Security Technology* 2 (Sept. 2018), pp. 1–37. DOI: 10.1080/23742917.2018.1518061.

[5] I. Sharafaldin et al. "Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy." In: *2019 International Carnahan Conference on Security Technology (ICCST)*. 2019, pp. 1–8. DOI: 10.1109/CCST.2019.8888419.

[6] Ansam Khraisat et al. "Survey of intrusion detection systems: techniques, datasets and challenges." In: *Cybersecurity* 2 (Dec. 2019). DOI: 10.1186/s42400-019-0038-7.

[7] Maonan Wang et al. "An Explainable Machine Learning Framework for Intrusion Detection Systems." In: *IEEE Access* 8 (2020), pp. 73127–73141. DOI: 10.1109/ACCESS.2020.2988359.

[8] Marek Majkowski. *Reflections on reflection (attacks)*. 2017. URL: https://blog.cloudflare.com/reflections-on-reflections/.

[9] Liron Segal. *Old Protocols, New Exploits: LDAP Unwittingly Serves DDoS Amplification Attacks*. 2017. URL: https://www.f5.com/labs/articles/threat-intelligence/old-protocols-new-exploits-ldap-unwittingly-serves-ddos-amplification-attacks-22609.

[10] Help Net Security. *New DDoS attacks misuse NetBIOS name server, RPC portmap, and Sentinel licensing servers*. 2015. URL: https://www.helpnetsecurity.com/2015/10/29/new-ddos-attacks-misuse-netbios-name-server-rpc-portmap-and-sentinel-licensing-servers/.

[11] Imperva. *NTP Amplification.* URL: https://www.imperva.com/learn/ddos/ntp-amplification/.

[12] Imperva. *SNMP Reflection / Amplification.* URL: https://www.imperva.com/learn/ddos/snmp-reflection/.

[13] SecurityWeek News. *DDoS Attacks Abuse TFTP for Reflection and Amplification.* 2016. URL: https://www.securityweek.com/ddos-attacks-abuse-tftp-reflection-and-amplification.

[14] David Silver et al. "Mastering the game of Go without human knowledge." In: *Nature* 550 (Oct. 2017), pp. 354–359. DOI: 10.1038/nature24270.

[15] Engineering Statistics Handbook. *What are outliers in the data?* URL: https://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm.

[16] Qiong Liu and Ying Wu. "Supervised Learning." In: (Jan. 2012). DOI: 10.1007/978-1-4419-1428-6_451.

[17] Memoona Khanam et al. "A Survey on Unsupervised Machine Learning Algorithms for Automation, Classification and Maintenance." In: *International Journal of Computer Applications* 119 (June 2015), pp. 34–39. DOI: 10.5120/21131-4058.

[18] Yale University. *Multiple Linear Regression.* URL: http://www.stat.yale.edu/Courses/1997-98/101/linmult.htm.

[19] Chigozie Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning.* 2018. arXiv: 1811.03378 [cs.LG].

[20] J. Wu. "Introduction to Convolutional Neural Networks." In: 2017.

[21] Kasun Vimukthi Jayalath. *Feedforward and Backpropagation Mathematics Behind a Simple Artificial Neural Network.* 2020. URL: https://medium.com/analytics-vidhya/feedforward-and-backpropagation-mathematics-behind-a-simple-artificial-neural-network-fd3f3ae15e3b.

[22] Sebastian Ruder. "An overview of gradient descent optimization algorithms." In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[23] Dor Bank, Noam Koenigstein, and Raja Giryes. *Autoencoders.* 2021. arXiv: 2003.05991 [cs.LG].

[24] Mengnan Du, Ninghao Liu, and Xia Hu. "Techniques for Interpretable Machine Learning." In: *CoRR* abs/1808.00033 (2018). arXiv: 1808.00033. URL: http://arxiv.org/abs/1808.00033.

[25] Scott Lundberg and Su-In Lee. "A unified approach to interpreting model predictions." In: *CoRR* abs/1705.07874 (2017). arXiv: 1705.07874. URL: http://arxiv.org/abs/1705.07874.

[26] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier." In: *CoRR* abs/1602.04938 (2016). arXiv: 1602.04938. URL: http://arxiv.org/abs/1602.04938.

[27] L. S. Shapley. "A value for n-person games." In: *Contributions to Theory, Games, vol. 2, no. 28* (1953), pp. 307–317.

[28] Ane Blazquez-Garcia et al. "A review on outlier/anomaly detection in time series data." In: *CoRR* abs/2002.04236 (2020). arXiv: `2002.04236`. URL: `https://arxiv.org/abs/2002.04236`.

[29] Mavuto Mukaka. "Statistics Corner: A guide to appropriate use of Correlation coefficient in medical research." In: *Malawi medical journal : the journal of Medical Association of Malawi* 24 (Sept. 2012), pp. 69–71.

[30] Ajinkya More. *Survey of resampling techniques for improving classification performance in unbalanced datasets.* 2016. arXiv: `1608.06048 [stat.AP]`.

[31] Neustar. *DDoS attacks: Big rise in threats to overload business networks.* 2021. URL: `https://www.home.neustar/resources/whitepapers/cyber-threats-and-trends-pandemic-style`.

[32] Arash Habibi Lashkari. "CICFlowmeter-V4.0 (formerly known as ISCXFlowMeter) is a network traffic Bi-flow generator and analyser for anomaly detection. https://github.com/ISCX/CICFlowMeter." In: Aug. 2018. DOI: `10.13140/RG.2.2.13827.20003`.

[33] Mahmoud Elsayed et al. "Network Anomaly Detection Using LSTM Based Autoencoder." In: Nov. 2020.

[34] Joao Paulo Abreu Maranhao et al. "Error-Robust Distributed Denial of Service Attack Detection Based on an Average Common Feature Extraction Technique." In: *Sensors* 20.20 (2020). ISSN: 1424-8220. DOI: `10.3390/s20205845`. URL: `https://www.mdpi.com/1424-8220/20/20/5845`.

[35] Ian J. Goodfellow et al. *Generative Adversarial Networks.* 2014. arXiv: `1406.2661 [stat.ML]`.

[36] Ting Chen et al. "A Simple Framework for Contrastive Learning of Visual Representations." In: *CoRR* abs/2002.05709 (2020). arXiv: `2002.05709`. URL: `https://arxiv.org/abs/2002.05709`.

[37] Robin Sommer. "Bro: An Open Source Network Intrusion Detection System." In: Jan. 2003, pp. 273–288.

[38] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System." In: *CoRR* abs/1603.02754 (2016). arXiv: `1603.02754`. URL: `http://arxiv.org/abs/1603.02754`.

[39] Leo Breiman. "Random Forests." In: *Mach. Learn.* 45.1 (Oct. 2001), pp. 5–32. ISSN: 0885-6125. DOI: `10.1023/A:1010933404324`. URL: `https://doi.org/10.1023/A:1010933404324`.

[40] Thomas Schlegl et al. *Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery.* 2017. arXiv: `1703.05921 [cs.CV]`.

[41] Ido Cohen et al. “DeepBrain: Functional Representation of Neural In-Situ Hybridization Images for Gene Ontology Classification Using Deep Convolutional Autoencoders.” In: Nov. 2017, pp. 287–296. ISBN: 978-3-319-68611-0. DOI: 10.1007/978-3-319-68612-7_33.

[42] Fei Tony Liu, Kai Ting, and Zhi-Hua Zhou. “Isolation Forest.” In: Jan. 2009, pp. 413–422. DOI: 10.1109/ICDM.2008.17.