TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Sergei Tsimbalist 121034IAPB

DETECTING, CLASSIFYING AND EXPLAINING IOT BOTNET ATTACKS USING DEEP LEARNING METHODS BASED ON NETWORK DATA

Bachelor's thesis

Supervisors: Sven Nõmm
PhD
Alejandro Guerra Manzanares
MSc

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Sergei Tsimbalist 121034IAPB

IOT BOTNETI RÜNNAKUTE AVASTAMINE, KLASSIFITSEERIMINE JA SELETAMINE SÜGAVÕPPEGA VÕRGUANDMETE PÕHJAL

bakalaureusetöö

Juhendajad: Sven Nõmm
PhD
Alejandro Guerra Manzanares
MSc

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Sergei Tsimbalist

06.01.2019

Abstract

The growing adoption of Internet-of-Things devices brings with it the increased participation of said devices in botnet attacks, and as such novel methods for IoT botnet attack detection are needed. This work demonstrates that deep learning models can be used to detect and classify IoT botnet attacks based on network data in a device agnostic way and that it can be more accurate than some more traditional machine learning methods, especially without feature selection. Furthermore, this works shows that the opaqueness of deep learning models can mitigated to some degree with Local Interpretable Model-Agnostic Explanations technique.

This thesis is written in English and is 31 pages long, including 5 chapters, 21 figures and 11 tables.

Annotatsioon

Asjade Interneti seadmete kasvav kasutuselevõtt toob kaasa nende seadmete suurema osalemise botneti rünnakutes, mistõttu on vaja uusi meetodeid IoT botneti rünnakute avastamiseks. See töö näitab, et sügava õppe mudeleid saab kasutada IoT botneti rünnakute avastamiseks ja klassifitseerimiseks võrguandmete põhjal seadme agnostilisel viisil ja et see võib olla täpsem, kui mõned traditsioonilisemad masinõppemeetodid, eriti ilma tunnuste valikuta. Lisaks näitab see, et sügava õppe mudelite läbipaistmatus võib mõningal määral leevendada Local Interpretable Model-Agnostic Explanations meetodiga.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 31 leheküljel, 5 peatükki, 21 joonist, 11 tabelit.

List of abbreviations and terms

ACK	Handshake protocol based botnet attack
C&C	*Command and Control*, botnet term
DDoS	*Distributed Denial of Service*, type of attack
DL	*Deep Learning,* machine learning subfield
IoT	*Internet of Things*, network of devices
k-NN	*k-Nearest Neighbors*, machine learning algorithm
LIME	*Local Interpretable Model-Agnostic Explanations*, data science method
LOF	*Local Outlier Factor*, machine learning algorithm
ML	*Machine Learning*, study of data and algorithms
MSE	*Mean Squared Error*, statistics term
P2P	*Peer to Peer*, mode of communication
SCAN	Scanning type of botnet attack
STD	*Standard Deviation*, statistics term
SYN	Handshake protocol based botnet attack
UDP	*User Datagram Protocol* based botnet attack
UDPPlain	*User Datagram Protocol* based botnet attack with modification

Table of contents

List of figures

List of tables

1 Introduction

1.1 Overview

In recent years the IoT (Internet-of-Things) market has experienced a dramatic growth, with number of IoT devices connected to the network reaching 7 billion in 2018 [1]. This creates a motivation for the malware industry to infect such devices and use them for malicious purposes. In 2017 the number of DDoS (Distributed Denial of Service) attacks has increased by 91% thanks to IoT botnets [2]. Such attacks are carried by out by a group of infected machines (bots), forming a botnet under the control of the attacker through a C&C (Command and Control) server, against some other entity in the network, such as a corporation or a government. For example, in 2016 one such botnet under the name of Mirai has attacked multiple internet companies like Krebs on Security, OVH and Dyn [3].

A typical botnet such as Mirai usually operates in multiple steps. Botnets spread starting with a scan phase in which they scan available networks for vulnerable devices. Access to these devices is then brute forced using some common credentials. In case of a successful access, the details of the compromised machine are sent to the attacker's server and the load phase begins, where malware is loaded and installed on a device. At this point device becomes a part of the botnet network and will be operated through a C&C server [3].

IoT itself means extending internet connection to devices and appliances that in the past operated offline. Connecting them to the internet allows for remote control and monitoring. IoT devices find their use in many different domains, both in consumer and enterprise markets. For example, the concept of Smart Home heavily relies on IoT by connecting thermometers, light bulbs, security cameras and other devices to the network [4]. This sheer variety of IoT devices and manufacturers contributes to the security challenges faced by the industry [4].

Traditional malware detection methods require a lot of manpower in order to analyze the threats and come up with detection rules [5], especially as malware industry continuously adapts and evades new countermeasures. In this regard machine learning gives a lot of promise for the task. One possible approach to the problem of IoT security against botnets is based on the application of machine learning to the network traffic data [5] [6].

1.2 Previous works

Earlier related research that deals with botnets does not yet refer to the IoT, perhaps as IoT concept has experienced tremendous growth only very recently. In 2011 a research has been done on detection of P2P (Peer-to-Peer) botnets during the pre-attack phase [7]. In 2013 a group of researchers in India again explored P2P botnet detection with neural networks [8]. Others tried different approaches, like not searching for particular patterns in overall traffic flow itself, but for example only in DNS queries [9] [10].

IoT botnet research comes prominently onto the scene more recently. In 2015 there was a proposal for packet inspection algorithm [11] that would have to be installed on the IoT device itself and would inspect the network packet payload. Later works, however, recognized that this host-based approach would be difficult to implement in real environment, due to the resource constraints and many differences between vendors. And so in 2018 a team from Ben-Gurion University at Negev described how botnet attack traffic can be detected based on network flow data using autoencoders, a particular type of neural network [12], in addition to that they also released a public dataset of Mirai and BASHLITE[1] botnet traffic data [13], that will be used in this work. A team at Aberdeen modeled IoT botnet attacks as a sequence using recurrent neural networks [14]. A couple of studies have been conducted at TalTech: one showed how to detect anomalous traffic using a combination of feature selection techniques together with unsupervised machine learning methods like Local Outlier Factor, One-class SVM and Isolation Forest [15], while another focused on the exploration of dimensionality reduction and methods like Decision Tree and k-Nearest Neighbors classification [16].

1.3 Novelty

Firstly, this work will fill the gap left by [12] where a separate model for each IoT device in the network has been created. The motivation is that training, deploying and maintaining separate models for each device could quickly become burdensome in big

[1] https://www.cyber.nj.gov/threat-profiles/botnet-variants/bashlite

networks. The datasets [13] for different devices will be combined into one and then autoencoder algorithm as described in [12] will be implemented for this dataset. This autoencoder will have similar size and number of layers as in [12] and their formula for the threshold will be used, albeit it will be modified – a parameter will be added to it, and an optimization is made for this parameter. Some details, such as activation function choice and optimization algorithm choice are this work's original contributions, as they were not described in detail in the referred work. The technical implementation (source code) is also completely independent from the referred work, as they didn't publicize it. Additionally, this work will employ a feature selection mechanism. This will allow for assessment of autoencoder method accuracy on smaller feature subsets and for making a fairer comparison on anomalous traffic detection with more traditional machine learning methods studied in [15].

Secondly, this work will propose and assess a deep feedforward neural network with softmax algorithm for attack type classification. Both classification of botnet type and classification of Mirai attack type will be done. This is all in contrast to [14], where a recurrent neural network was used, and only Mirai was studied. Moreover, this work will also employ feature selection for attack classification to assess neural network performance on varying number of features and make a comparison to more classical machine learning algorithms studied in [16].

Thirdly, this work will explore the topic of explaining the neural network predictions, that is usually not addressed by works studying deep learning methods. Neural networks suffer from what can be called a lack of interpretability of results, a problem not experienced by e.g. decision trees. As this can potentially stop a deep learning algorithm from being taken into a real-life use, this work will assess LIME (Local Interpretable Model-Agnostic Explanations) [17] technique, that can provide some interpretability to the results.

2 Research methods

For the purpose of our tasks we will employ two different architectures of neural networks: an autoencoder for anomaly detection and a feedforward network with softmax algorithm for classification. Fisher's score will be used to select the most important features and investigate algorithm performance on varying number of features. Data will be pre-processed using the normal scaling. Explanations for the neural network decision will be given by Local Interpretable Model-Agnostic Explanation method. Components studied in this work could in real life be combined as on Figure 1 to create a complete detection and classification pipeline.

On the technical side, code will be written in Python 3 with the use of modern frameworks for data handling and machine learning, such as Pandas, scikit-learn and Keras.

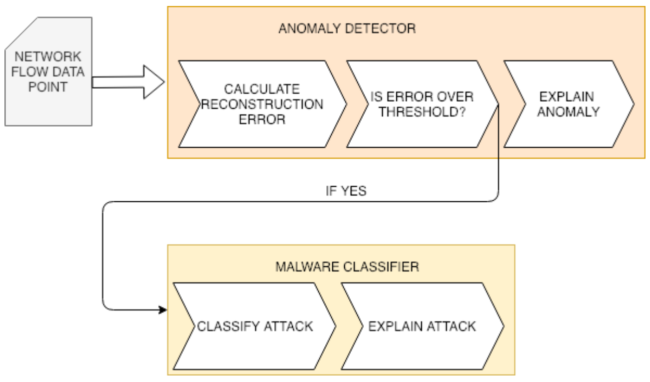


Figure 1. Potential attack detection, classification and explanation pipeline.

2.1 Data

The data was obtained experimentally in a laboratory environment by the research team at Ben-Gurion University of Negev [13]. In this environment different IoT devices were connected to an isolated network using both Wi-Fi and wired connections. Additionally, botnet components were installed in the network, such as C&C server. Then, using port mirroring at the internet switch, data was gathered using Wireshark for both normal traffic, when none of the devices were infected, and as well for malicious traffic, when devices were infected.

2.1.1 Description

The data comes from 9 IoT devices belonging to smart-home and security domains: doorbells, security cameras, thermostat, baby monitor. For this work all devices are combined into one dataset.

In total data consists of 115 features which are related to different information about the packets, aggregated in different ways.

The network statistics data that is included consists of: weight, or packet count; mean of packet size; variance of packet size; standard deviation of packet size; radius as root squared sum of two stream's variances; magnitude of two stream's means; covariance and Pearson's correlation coefficient.

Data is aggregated by:

- Source Host IP (*H* – feature name part in dataset file)
- Source MAC-IP (*MI*)
- Source and destination host IP (*HH*)
- Source and destination host and port (*HpHp*)
- Source and destination host traffic jitter (*HH_jit*)

Further, each statistic was calculated for different window sizes: 100ms, 500ms, 1.5sec, 10sec, and 1min using lambda decay parameter. More details about the dataset are available at the UCI website [13].

Two types of botnet were installed in the laboratory environment: BASHLITE and Mirai. These two are most notable botnet families in the world of Linux based IoT. Further, Mirai data consists of 5 attack classes: SYN, ACK, UDP, UDPPlain, SCAN. The dataset is described in greater detail in [12].

In total dataset consists of more than 5 million points.

Table 1. Data set size

Normal	**Mirai**	**BASHLITE**
555 932	3 668 402	1 032 056

For this work BASHLITE tcp.csv and udp.csv files were excluded, as their content appeared to be inconsistent.

2.1.2 Splitting the dataset

Dataset is heavily skewed towards attack data and attack data is dominated by Mirai data, it is unbalanced. During the experiments the data will be sampled to create more or less balanced classes.

For anomaly detection purposes, normal data will be split into 3 sets: train, optimize and test. Attack data will be added on testing stage and it will be equal to the normal test data set size.

For botnet classification purposes, as the data is heavily overrepresented with attack classes, but we also want to include the normal data to compare our classification to other results, the Mirai and BASHLITE datasets will be sampled to the size of normal data, and then our data set will be split into train and test subsets as 80:20, as is custom in the machine learning field. For Mirai attack type classification, from each of 5 classes 100000 points will be taken to total of 500000 and then split 80:20 for train and test.

2.1.3 Feature scaling

When training neural networks, it is considered to be the best practice to scale the data [18]. Otherwise different problems can be encountered such as degraded performance, unpredictable behavior of optimization algorithm and exploding gradient.

There are different ways to scale the data, some of them rely on knowing the maximum and minimum values for the distribution, and as the data is not in some predefined range like, e.g. pixel value data, the standardization method is chosen, that relies on computing the mean and variance of the train set.

This formula describes how to obtain scaled features (1).

$$\tilde{X}_i = \frac{X_i - \mu_i}{\sigma_i} \tag{1}$$

In the given formula X tilde is the dataset feature i after scaling, X is the dataset feature before scaling, μ is the mean of the training set feature and σ is the standard deviation of the training set feature. Scaling then is applied to each feature of the dataset independently. As a result, this should give us the features with mean of 0 and variance of 1.

2.1.4 Feature selection

Some features of the data may be more important than others for the tasks of anomaly detection or attack classification. This has been reflected by the works studying dimensionality reduction for botnet attack detection [15] [16].

By selecting a smaller number of features we will enable this work to be compared to other works and additionally demonstrate how neural network performance changes when applied to varying number of inputs.

The metric that we will use for measuring the importance of a feature will be Fisher's score [19, p. 290]. It is calculated using the following formula (2).

$$F = \frac{\sum_{j=1}^{k} p_j (\mu_j - \mu)^2}{\sum_{j=1}^{k} p_j \sigma_j^2} \tag{2}$$

This score can be thought of as measuring the ratio of the average interclass separation to the average intraclass separation. In the enumerator we sum for each class in the dataset the product of this class's proportion given by p_j with the squared difference of this class's mean μ_j and the feature's mean μ. While in the denominator we sum the product of the proportion of the class with its variance σ_j^2. Then it is possible to simply sort the features by this score in descending order and take only top N important features.

2.2 Anomaly detection

The task of detecting an attack can be framed as an anomaly or outlier detection problem. This is based on the assumption that the normal usage network traffic data and malware network data will differ in some dimensions, so that an algorithm will be able to make a distinction.

There are different methods for anomaly detection, some of them, which were used previously in other works for the same task, are Local Outlier Factor, Isolation Forest and One class SVM.

LOF algorithm for example uses the deviation of local density of a given data point with respect to its neighbors. The samples with considerably lower densities than neighbors are considered as outliers.

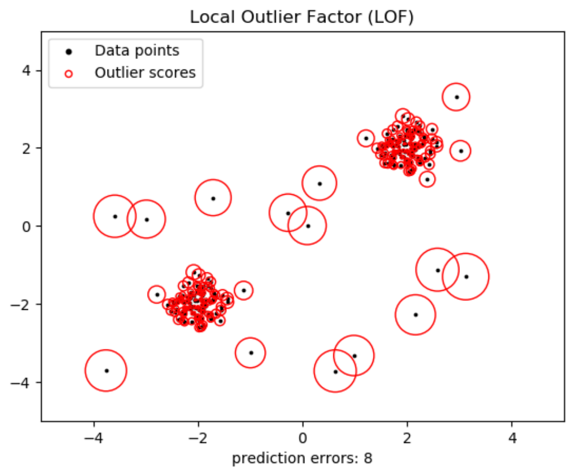


Figure 2. LOF plot[1]

Isolation forest works by splitting random features to isolate samples. The assumption is that outliers will need less partitions to be isolated.

One-class SVM learns from normal data its properties, and then on novel attack data it can predict it as not being normal.

2.2.1 Deep Autoencoder

One known method of how to perform anomaly detection using neural networks is to use the special architecture that is called autoencoder.

Autoencoders are not a recent invention, they were known as early as 1986 [20]. Until recently, however, the applications of autoencoders were mainly: representation learning, dimensionality reduction and denoising.

For the anomaly detection purposes, the autoencoders, in particular deep autoencoders, meaning having more than one hidden layer, have started gaining popularity only recently, perhaps due to the growth of interest in the field of DL as a whole. Examples of specific anomaly detection tasks solved by deep autoencoders include fraud detection, e.g. credit card fraud, or experiment data quality assessment, as was done CERN [21].

[1] https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_outlier_detection.html

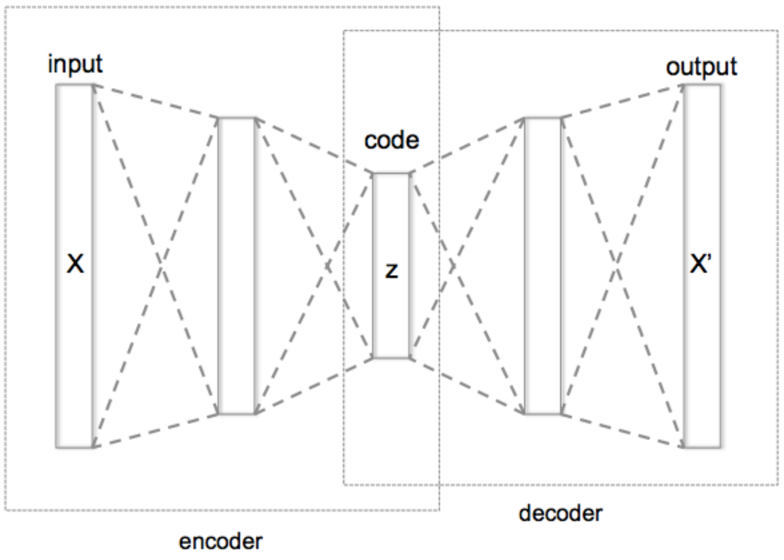


Figure 3. Autoencoder[1].

An autoencoder can be described as consisting of two different networks: an encoder and a decoder. The job of the encoder is to receive the input X and encode it into a hidden state Z. The decoder then receives this hidden state Z and aims to reconstruct the original input as X' [22, pp. 502-504].

$$Z = e(X)$$
$$X' = d(Z)$$
$$X \approx X'$$

The optimization task then is to minimize the loss function, which in our case will be defined as a mean squared error between the original input and the reconstruction (3).

$$\mathcal{L}(X, X') = \frac{1}{n}\sum_{i=1}^{n}(X_i - X'_i)^2 \tag{3}$$

Now if we consider the fact that the dimensionality of hidden state Z is less than the dimensionality of input X, then this hidden state acts as a bottleneck, it forces the autoencoder to learn useful representation of the input dataset, so then it could successfully reconstruct them. We then expect that if we give the autoencoder a data point not from the normal distribution, but from attack dataset, it will fail to reconstruct

[1] https://en.wikipedia.org/wiki/Autoencoder#/media/File:Autoencoder_structure.png

it, meaning that the difference between the original and the reconstruction will be too high.

Additional construct that we will need then is the threshold for our error, such that if the error is higher that the given threshold, we consider the data point being an anomaly or an attack. In another work [12] the authors decided to use the threshold defined as follows (4).

$$\tau = \overline{MSE(X_{opt})} + STD\left(MSE\left(X_{opt}\right)\right) \tag{4}$$

However, we will try to find a more optimal value for our task by trying out different values of N with this formula (5).

$$\tau = \overline{MSE(X_{opt})} + N * STD\left(MSE\left(X_{opt}\right)\right) \tag{5}$$

Our goal will be to reduce the number of false positives, where a normal traffic is misclassified as an attack.

Internally our autoencoder will use 5 hidden layers of sizes 0.75, 0.5, 0.25, 0.5, 0.75 of the input feature vector size. That is in addition to input and output layers of sizes n respectively. The number of parameters in such model can be estimated as (6) where n is the number of input parameters and b is the number of bias[1] terms.

$$n * \frac{3}{4}n + \frac{3}{4}n * \frac{1}{2}n + \frac{1}{2}n * \frac{1}{4}n + \frac{1}{4}n * \frac{1}{2}n + \frac{1}{2}n * \frac{3}{4}n + \frac{3}{4}n * n + b = 2.5n^2 + b \tag{6}$$

We will use hyperbolic tangent as an activation function for our hidden unit neurons (7) as per literature recommendation [22, p. 195].

$$g(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{7}$$

Without a nonlinear function the result of the autoencoder becomes just a linear combination of features and as such loses power offered by nonlinear models.

[1] https://ml-cheatsheet.readthedocs.io/en/latest/nn_concepts.html#bias

2.3 Attack classification

Having solved the problem of attack detection using autoencoders we then would like to inspect the data to find out what type of attack is it exactly. As the dataset consists of two botnet types: Mirai and BASHLITE, there will be these two classes, plus, in order to compare our performance with earlier work using traditional ML methods, we will consider normal data as a third class. Also, we will try to classify different Mirai attack types: ACK, SCAN, SYN, UDP, UDPPlain.

In general, the task of classification appears to be the most popular application of machine learning and consequently there are lots of algorithms to solve it: SVMs, decision tress, random forests, k-nearest neighbors and others.

Apart from the neural network that will be applied in this work, of interest it to briefly introduce decision trees and k-nearest neighbors, as this is what we will compare our results to.

Decision Tree algorithm learns the partitions of feature values for given classes. Main advantage of the algorithm is that the results are easily interpretable. Sometimes however, decision tree learner can produce overly complicated and biased solution.

k-Nearest Neighbors algorithm works on principle that the class of the data point is decided by the classes of k nearest neighbors to that point.

2.3.1 Deep Neural Network

We will attempt to solve the problem of classification with a deep neural network with two hidden layers each with 8 neurons. Unfortunately, there are no theoretical rules yet that would guide the choice of a neural network size for particular tasks. Mostly the approach seems to be to just increase the number of layers or neurons until good accuracy is attained. Preliminary experiments showed that for our task of classification with 115 features and 3 classes and 5 classes, the proposed architecture should suffice.

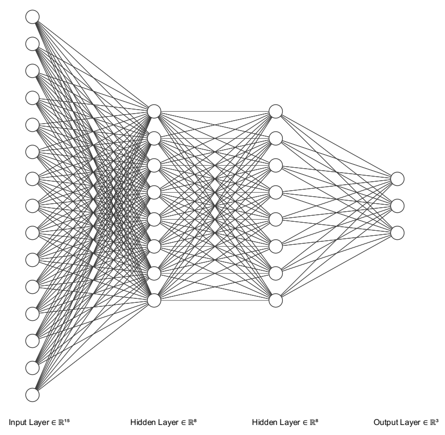

Input Layer $\in \mathbb{R}^{15}$
Hidden Layer $\in \mathbb{R}^{8}$
Hidden Layer $\in \mathbb{R}^{8}$
Output Layer $\in \mathbb{R}^{3}$

Figure 4. Neural network diagram.

Apart from the number of layers and neurons other important details to describe for proposed neural network are:

- Hyperbolic tangent activation function for hidden neurons
- Softmax function applied to the last layer
- Categorical cross-entropy as a loss function

Hyperbolic tangent activation function definition was already given previously (7).

The need for softmax arises from the fact that we are doing a classification, hence at the last layer we want to receive a probability vector with normalized probabilities for each class, and softmax (8) gives us exactly that [22, pp. 185-186].

$$softmax(x_j) = \frac{e^{x_j}}{\sum_{i=1}^{N} e^{x_i}} \tag{8}$$

Categorical cross-entropy [22, p. 178] is a standard choice for a loss function with multiple classes.

2.4 Evaluation metrics

For the task of anomaly detection, we can apply a two-class confusion matrix to evaluate our results as in Table 2.

Table 2. Confusion matrix for two classes.

	Predicted normal	**Predicted attack**
Actual normal	True Negative (TN)	False Positive (FP)
Actual attack	False Negative (FN)	True Positive (TP)

Accuracy can then be defined as (9).

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

And precision as (10).

$$PR = \frac{TP}{TP + FP} \tag{10}$$

False positive rate will be defined as (11).

$$FPR = \frac{FP}{FP + TN} \tag{11}$$

Similarly, accuracy can be computed for more than 2 classes. Also, for our deep learning models, we can also measure how long does the training take in seconds to achieve good accuracy and how complex the model is, meaning how many parameters does it contain and how much does it weight when saved to disk.

2.5 Local Interpretable Model-Agnostic Explanations

Both the autoencoder and the classification neural network suffer from the same problem of opaqueness. At least in theory, of course, one could inspect all the weights of neural network to understand how it made the decision that in made, but in practice the number of parameters is too high to make any meaningful inference from that.

As deep neural network becomes the algorithm of choice for many modern tasks, it becomes imperative to find a solution for interpretability. One possible approach, known as Local Interpretable Model-Agnostic Explanation [17] will be studied in this work.

This technique works by perturbing values of a particular instance that we want to explain and learning a local linear approximation for classification. As shown on the example Figure 5, where the bold red cross point is being explained, a linear model can be inferred by looking at instances that are near and their classes, and this line can be presented as an explanation.

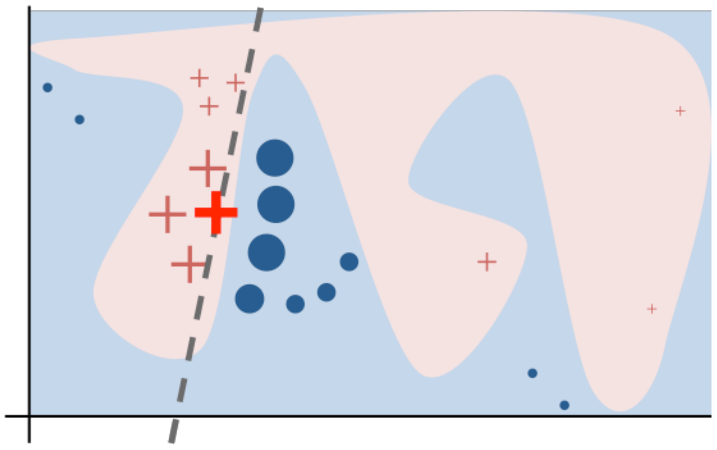


Figure 5. LIME plot[1].

2.6 Tools and technologies

The language of choice is *Python*[2] 3.6. Recently *Python* has been extremely popular with machine learning community thanks to many factors. The wealth of libraries and frameworks for data processing and data analysis makes working on ML projects quite a pleasant experience. Most of the experiments for this thesis were done in a form of a *Jupyter*[3] notebook. This can be more convenient than terminal programs, as *Jupyter* allows for better visualization and rerunning of different parts of an experiment. *Pandas*[4] library is used to handle the csv data for this work. *Pandas* data frames, backed by *Numpy*[5] arrays solve the problem of our not-so-small data (the size of the dataset is in Gigabytes). This work also relies on *scikit-learn*[6] library for the scaling and evaluation metrics functions. Neural network models are composed using *Keras*[7] 2.2 library, which in turn relies on *Tensorflow*[8] framework for all the computations.

[1] https://raw.githubusercontent.com/marcotcr/lime/master/doc/images/lime.png

[2] https://www.python.org/

[3] https://jupyter.org/

[4] https://pandas.pydata.org/

[5] https://www.numpy.org/

[6] https://scikit-learn.org/stable/

[7] https://keras.io/

[8] https://www.tensorflow.org/

3 Results

The results of the experiments are listed in this chapter. Experiments were run multiple times to ensure their validity. Random seed was set where possible to ensure that results don't vary between different reruns. All the experiments were run on a computer with 2,6 GHz Intel Core i7 CPU.

3.1 Feature scoring

Fisher's score was calculated (Table 3) on training sets for each data subset used respectively for attack detection, botnet type classification and Mirai attack type classification. Results in later sections where only a subset of features is used, are based on these rankings. Code for the score calculation procedure can be found in Appendix 1.

Table 3. Fisher's scores for different data subsets.

2 class (normal, attack)		**3 class (2 botnets + 1 normal)**		**5 classes of Mirai attacks**	
Feature	**F Sc**	**Feature**	**F Sc**	**Feature**	**F Sc**
MI_dir_L0.1_weight	3.34	MI_dir_L3_weight	1.96	MI_dir_L0.01_var	43.75
H_L0.1_weight	3.34	H_L3_weight	1.96	H_L0.01_var	43.75
MI_dir_L1_weight	3.18	MI_dir_L5_weight	1.93	MI_dir_L0.1_var	41.43
H_L1_weight	3.18	H_L5_weight	1.93	H_L0.1_var	41.43
MI_dir_L3_weight	3.01	MI_dir_L1_weight	1.87	MI_dir_L0.01_mean	30.05
H_L3_weight	3.01	H_L1_weight	1.87	H_L0.01_mean	30.05
MI_dir_L5_weight	2.86	MI_dir_L0.1_weight	1.70	MI_dir_L0.1_mean	27.03
H_L5_weight	2.86	H_L0.1_weight	1.70	H_L0.1_mean	27.03
MI_dir_L0.01_weight	1.65	MI_dir_L0.01_weight	1.43	MI_dir_L1_var	19.62
H_L0.01_weight	1.65	H_L0.01_weight	1.43	H_L1_variance	19.62

3.2 Attack detection

The autoencoder as described in section 2.2.1 was created (Appendix 2) and trained on all 115 features. Default Keras optimization hyperparameters were used. Training was limited to 100 epochs with additional condition of early stopping (Figure 6), using functionality provided by Keras. This ensures that the model is trained only until the score on validation set is not getting worse – this in turn helps to avoid overfitting the training set.

```
es = EarlyStopping(monitor='val_loss', patience=5)
```

Figure 6. Keras early stopping.

Adam [23] gradient descent optimization algorithm was chosen, as it should give noticeable improvements in training times compared to more basic stochastic gradient descent.

The code for training procedure is found in Appendix 4.

The model trained in total for 23 epochs, taking about 25 seconds for each epoch, totaling in 584 seconds of training time. Figure 8 demonstrates the training curve. Loss is defined as MSE between original and predicted values.

Training set had 185310 samples and optimization (validation) set 185311 samples.

The created model contained 33900 parameters (weights) and when saved, the file was of size 471 kB. Figure 7 shows the created model’s architecture.

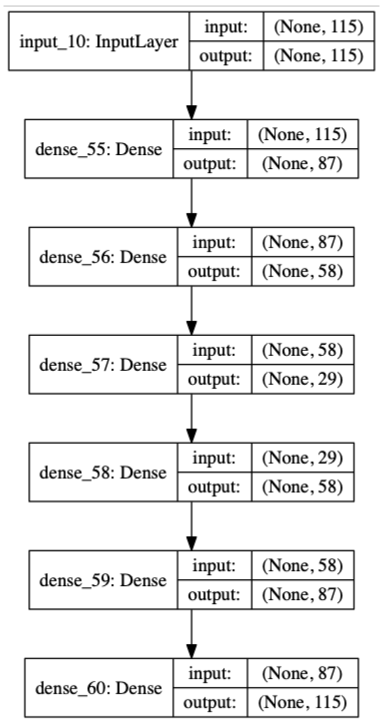


Figure 7. Keras Autoencoder architecture.

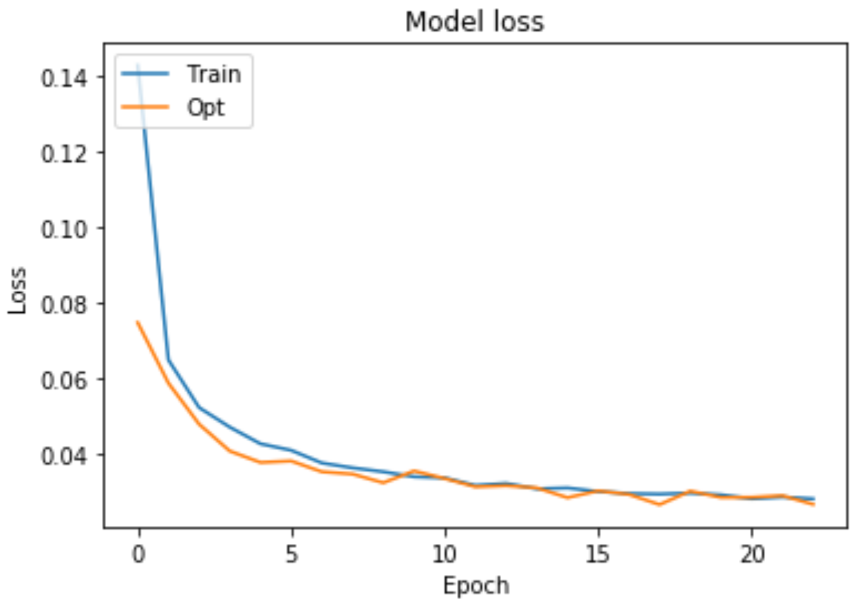


Figure 8. Autoencoder training curve.

Values of integers from 1 to 10 were tried for the threshold equation (5) parameter N on the optimization dataset. The results are presented in the Table 4. Threshold with N value of 10 showed the best accuracy and lowest numbers of false positives so it was chosen for subsequent operations on the test set.

Table 4. Results with different N values for threshold.

N	**Accuracy**	**False positives**	**False negatives**
1	0.9894	3911	23
2	0.9934	2203	25
3	0.9961	1396	28
4	0.9973	978	35
5	0.9979	744	38
6	0.9983	587	41
7	0.9986	467	43
8	0.9988	387	45
9	0.9990	292	51
10	0.9992	245	52

With model trained and threshold selected it is now possible to do an evaluation on the test set. Achieved accuracy is 0.9991 and precision is 0.9985. A more detailed breakdown is given by a confusion matrix Figure 9. False positive rate is thus 0.0015.

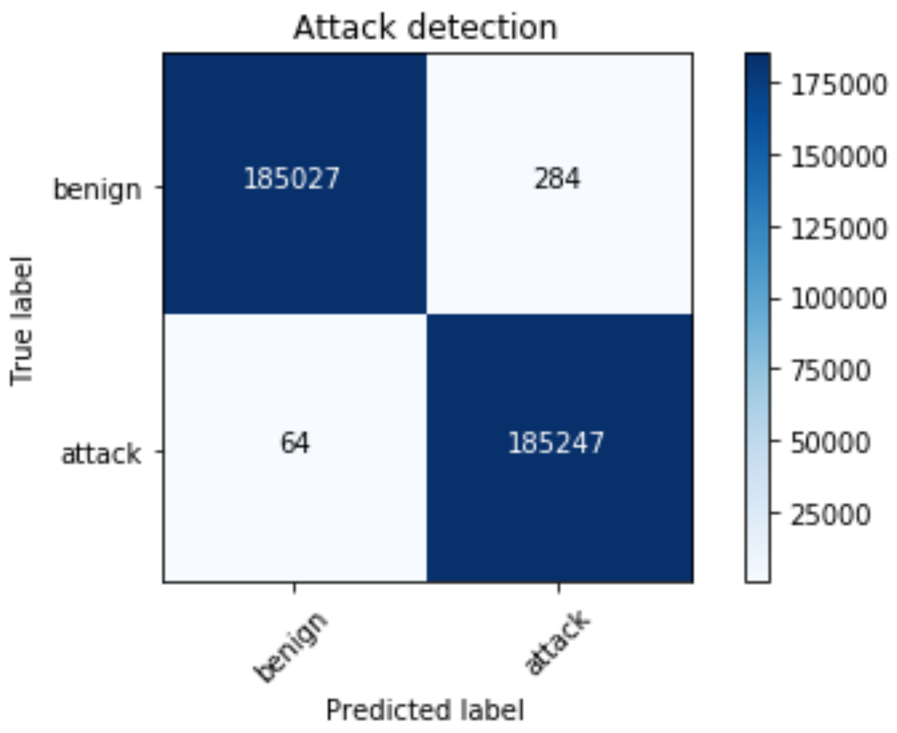


Figure 9. Attack detection on test set.

Additionally, autoencoder models were trained on reduced number of features for the purposes of their comparison. Some modifications in the training procedure were introduced, such as the maximum number of epochs was limited to 50. Also, the value for threshold parameter N was used that was selected earlier from Table 4. Figure 10 shows how accuracy depends on the number of input features in such model. While Figure 11 demonstrates how number of parameters in the autoencoder grows as a function of a number of input features.

Table 5 makes a comparison between more traditional ML methods for anomaly detection and the autoencoder implemented in this work. It should be noted, that the specific features used by each method are varied, as such this comparison is not strict.

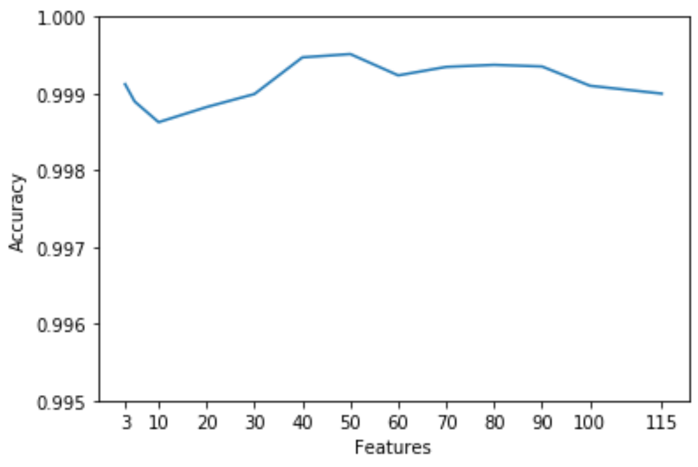


Figure 10. Autoencoder accuracy for different # of features.

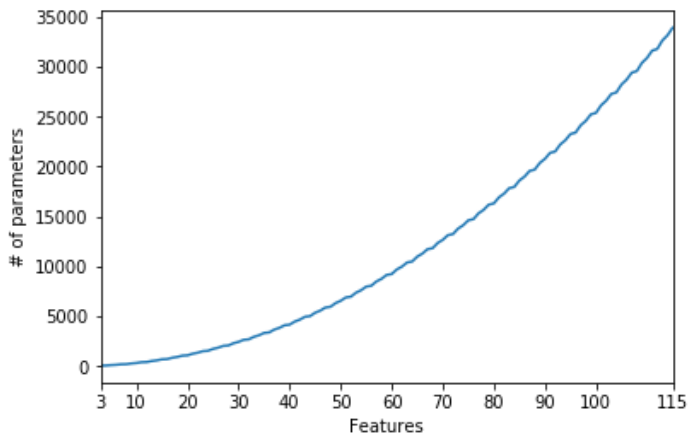
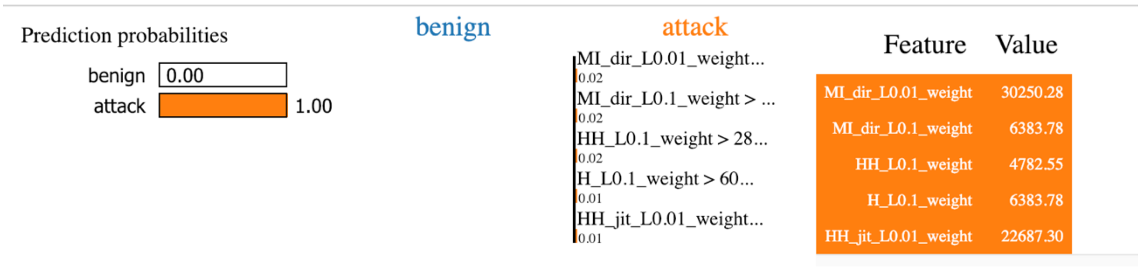


Figure 11. Autoencoder # of parameters for different # of features.

Table 5. Attack detection performance comparison.

Number of features	**Accuracy**		
	Deep Autoencoder	**Entropy SVM [15]**	**Variance Isolation Forest [15]**
3	0.9991	0.8337	0.6229
5	0.9989	0.6765	0.6055
10	0.9986	0.5050	0.9220

Figure 12 presents a LIME explanation for a detected attack. While Table 6 presents the boundaries used by this decision.



Feature	Value
MI_dir_L0.01_weight	30250.28
MI_dir_L0.1_weight	6383.78
HH_L0.1_weight	4782.55
H_L0.1_weight	6383.78
HH_jit_L0.01_weight	22687.30

Figure 12. LIME explanation for attack detection.

Table 6. LIME explanation boundaries for attack detection.

Boundary	**Importance**
MI_dir_L0.01_weight > 27982.34	0.02
MI_dir_L0.1_weight > 6079.16	0.02
HH_L0.1_weight > 2849.41	0.02
H_L0.1_weight > 6079.16	0.01
HH_jit_L0.01_weight > 12113.47	0.01

3.3 Attack classification

Two types of deep learning classifier models were trained to differentiate between different botnet types (Mirai, BASHLITE and also non-botnet) and between different Mirai botnet attack types (SCAN, ACK, SYN, UDP, UDPPlain). Additionally, these types of classifiers were trained on varying number of features to assess their performance.

3.3.1 Botnet type classification

For botnet type classification a neural network was implemented as specified in section 2.3.1 using function from Appendix 3. As with autoencoder, amount of training epochs was limited to 100 for practical purposes, and default Keras hyperparameters were used. Training code can be found in Appendix 5.

Resulting model based on all 115 features contained 1027 parameters (weights). Training took only 11 epochs with early stopping. Figure 13 shows the learning curve for this model. Total training time amounted to 689 seconds with about 62 seconds per epoch. Saved model file weighted only 41kB.

On the test set of size 333560, the accuracy of 0.9997 was achieved.

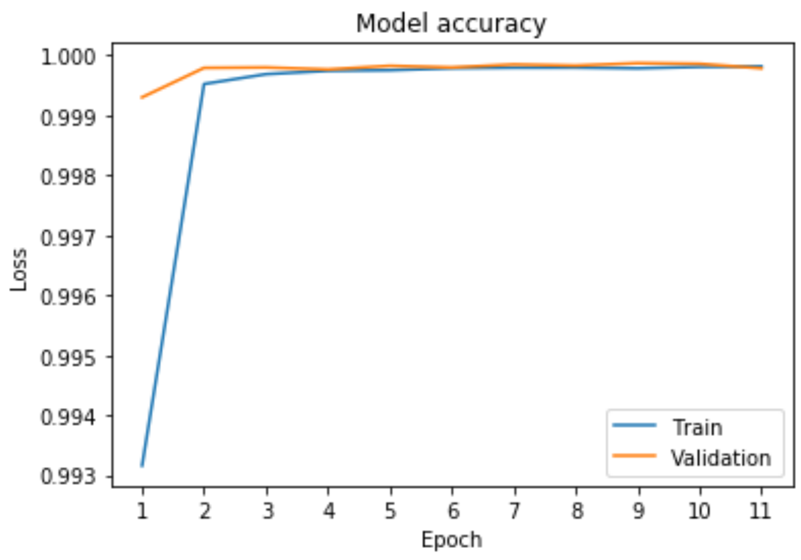


Figure 13. Botnet classification learning curve.

Figure 14 gives a complete test set classification breakdown in the form a confusion matrix.

For the purpose of comparison with other ML methods, model was retrained on 2, 3 and 10 features selected by their Fisher's score and the comparison is shown in Table 7. This classifier was also retrained on selected numbers of features from 2 to 115 to show how the accuracy changes with growing number of features, the result is on Figure 15.

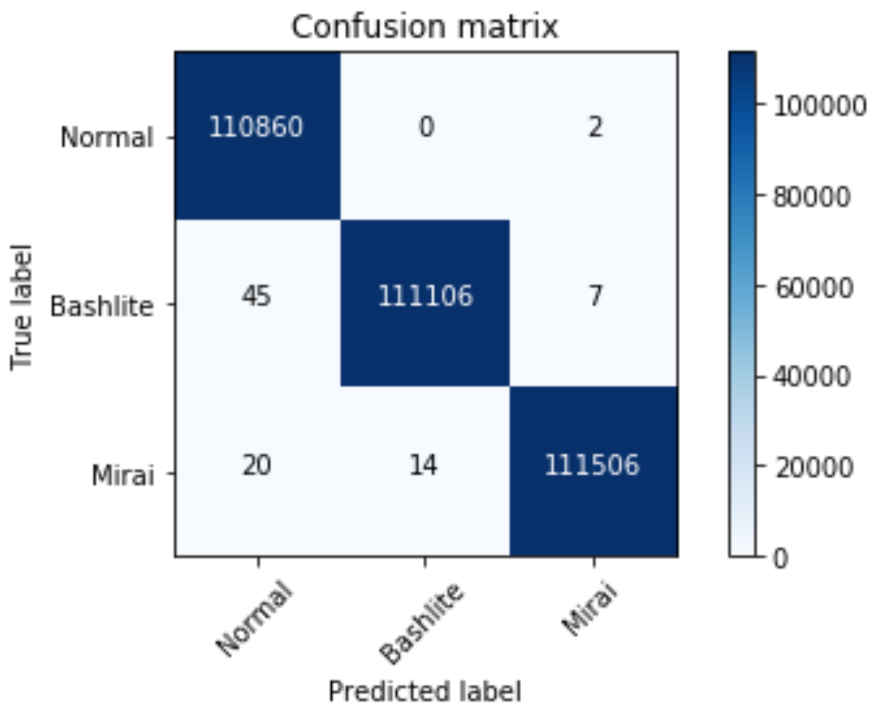


Figure 14. Botnet classification confusion matrix.

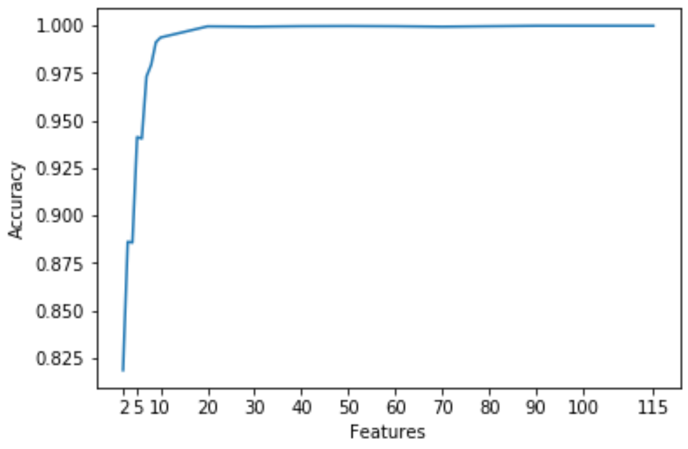


Figure 15. Botnet classification accuracy with different # of features.

Table 7. Botnet classification accuracy comparison.

Number of features	**Accuracy**		
	Deep Neural Network	**Decision Tree [16]**	**k-NN [16]**
2	0.8188	0.9843	0.9805
3	0.8663	0.9851	0.9724
10	0.9933	0.9897	0.9497

A random Mirai botnet class data point was then selected and its explanation by LIME is presented on Figure 16. LIME boundaries are additionally specified in Table 8.

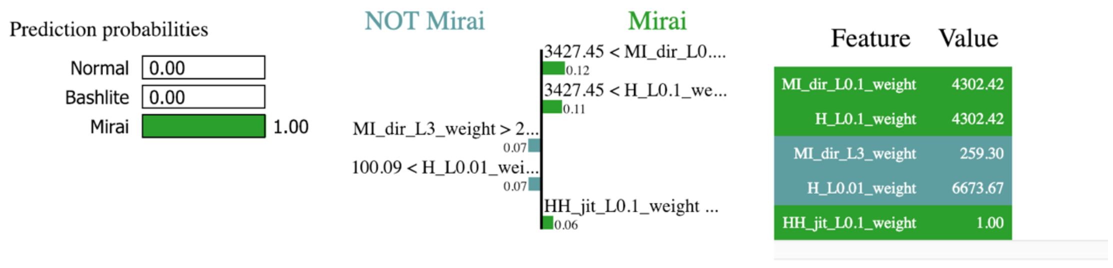


Figure 16. LIME explanation for botnet classification.

Table 8. LIME explanation boundaries for Mirai botnet classification.

Boundary	**Importance**
3427.45 < MI_dir_L0.1_weight <= 6431.98	0.12
3427.45 < H_L0.1_weight <= 6431.98	0.11
MI_dir_L3_weight > 252.04	-0.07
100.09 < H_L0.01_weight <= 11161.10	-0.07
HH_jit_L0.1_weight <= 1.21	0.06

3.3.2 Mirai attack type classification

For Mirai attack type classification the neural network was created as specified in section 2.3.1 using function from Appendix 3. Again, default hyperparameters provided by Keras were used during training, and Adam optimization algorithm was employed to speed up training.

With early stopping mechanism, training took 17 epochs, in total taking 258 seconds, so about 15 seconds for each epoch. Training set size was 400000. Training curve is presented on Figure 17. Model consisted of 1045 parameters and weighted 41 kB.

On test set of size 100000 accuracy of 0.9984 was achieved. A detailed confusion matrix is shown on Figure 18.

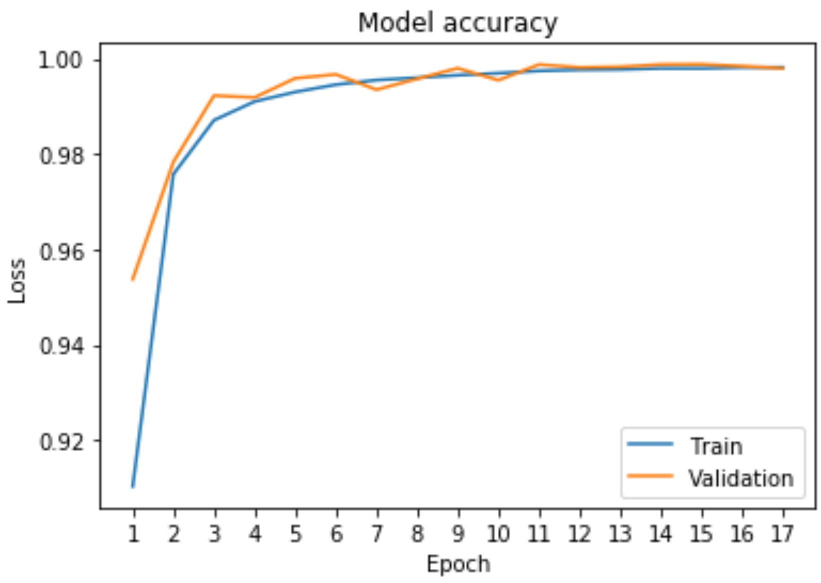


Figure 17. Mirai attack classification learning curve.

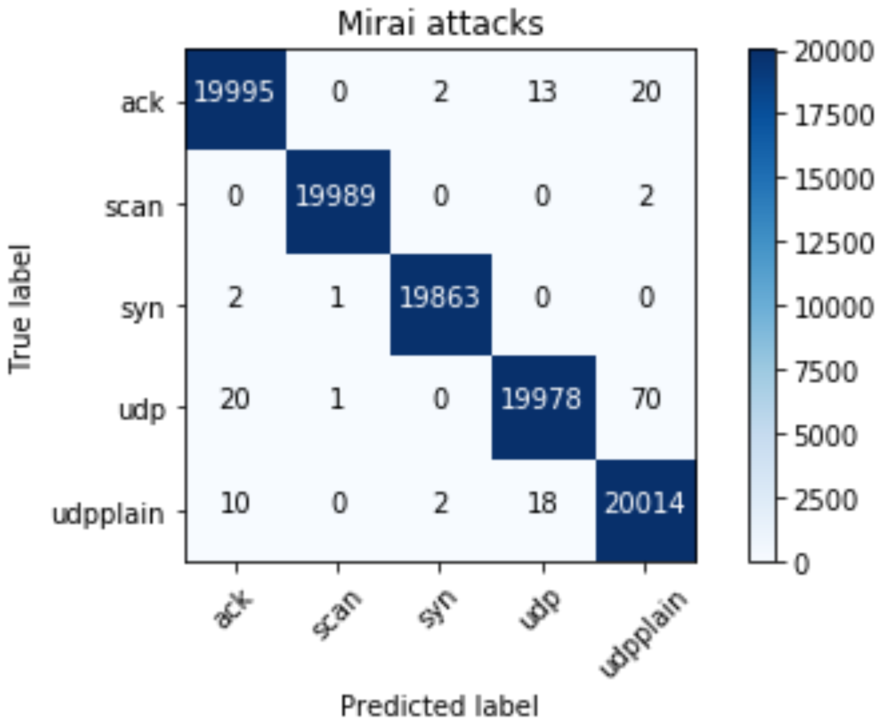


Figure 18. Mirai attack classification confusion matrix.

For comparison purposes model was also recreated and retrained on 2, 3, 4, 5, and 10 features selected by Fisher’s score. The results are demonstrated in Table 9. Figure 19 shows how accuracy changes from on sets of features from 2 to 115.

Table 9. Mirai attack classification accuracy for different # of features.

Number of features	Accuracy
2	0.6235
3	0.6294
4	0.6284
5	0.9783
10	0.9788

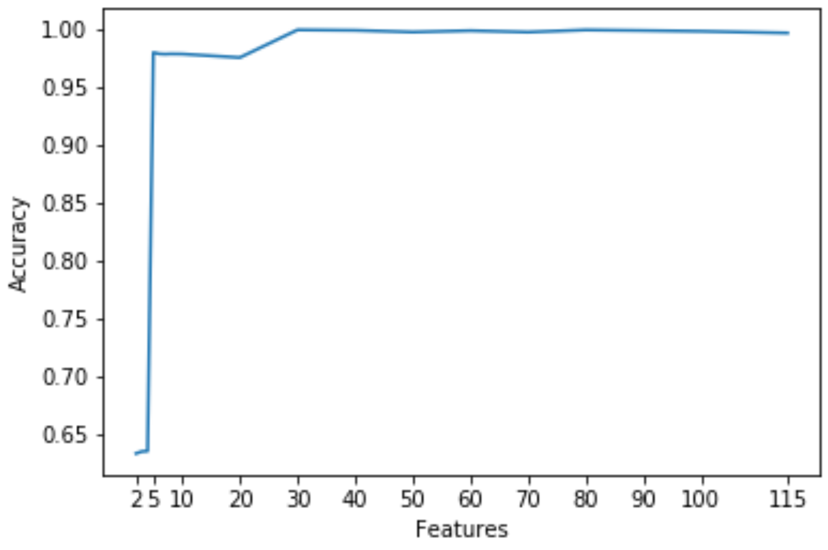


Figure 19. Mirai attack classification accuracy for different # of features.

A random UDP type attack point was selected to create a LIME explanation. Figure 20, Table 10 and Table 11 show this explanation and as classifier was not absolutely certain in this case, additionally explanation for ACK class which had 30% probability assigned to it is also explained.

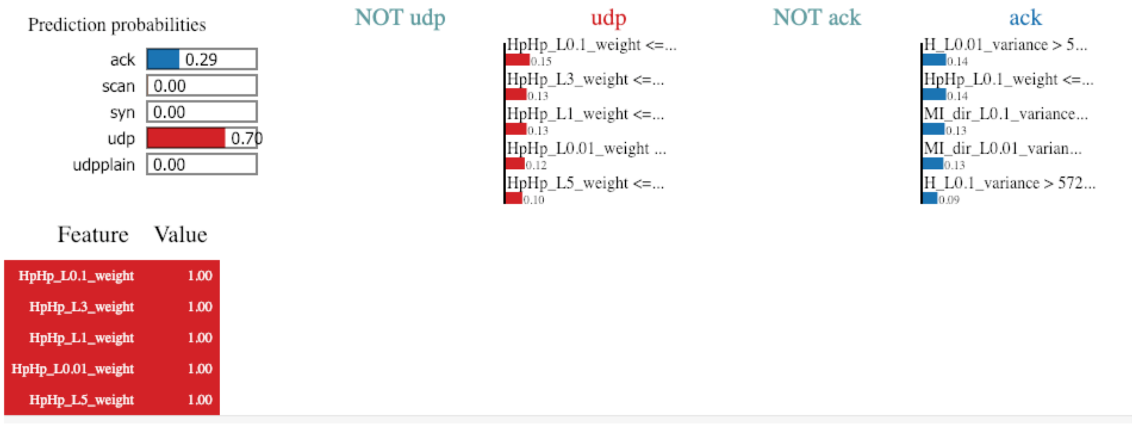


Figure 20. LIME explanation for Mirai attack type UDP.

Table 10. LIME explanation boundaries for correct UDP class.

Boundary	**Importance**
HpHp_L0.1_weight <= 1.00	0.15
HpHp_L3_weight <= 1.00	0.13
HpHp_L1_weight <= 1.00	0.13
HpHp_L0.01_weight <= 1.00	0.12
HpHp_L5_weight <= 1.00	0.10

Table 11. LIME explanation boundaries for wrong ACK class.

Boundary	**Importance**
H_L0.01_variance > 57736.45	0.14
HpHp_L0.1_weight <= 1.00	0.14
MI_dir_L0.1_variance > 57211.65	0.13
MI_dir_L0.01_variance > 57736.45	0.13
H_L0.1_variance > 57211.65	0.09

4 Analysis and discussion

Overall the results of the experiments showed that very good results can be achieved in the application of deep learning methods to the problems of IoT botnet attack detection and classification.

4.1 Detecting attacks

The attack detection with deep autoencoder showed outstanding results with accuracy of 0.9991 and false positive rate of 0.0015. As such there does not appear to be a lot of room for improvement in this regard.

One of the goals of this work was to show, that the anomaly detection with autoencoders can be applied to the data coming from multiple different IoT devices, this is in contrast to [12], where an autoencoder was trained for each IoT device in the network. The results suggest that this goal can be considered accomplished, as the mean false positive rate of 0.007±0.01 that was reported there is quite comparable to ours 0.0015. Is should be noted that our approach had an advantage of higher threshold by the equation (5) with N=10. In fact, difference in the number of false positives between N=1 and N=10 as shown by Table 4 is almost 16 times, even if both numbers are small in relation to the total number of negatives.

Another interesting aspect of the studied approach is that the accuracy of the autoencoder seems to stay relatively constant with varying number of features as shown in Figure 10. The explanation can probably be given by the plot in Figure 21 which shows that there is a very noticeable difference in data distributions based on the number of packets for normal and attack data. This is not surprising, considering that DDoS attacks are based on large amounts of packets beings sent in order to deny a service. What is interesting however, is that the autoencoder is able to effectively filter out the unimportant data, as the accuracy does not drop with increasing number of features, compared to more traditional methods, some of which apparently have problems with increased dimensions like SVMs as per Table 5.

It has to be noted, however, that to do this feature analysis using Fisher's score, one has to be in possession of attack data. And as such it would not possible to do this scoring having only normal data, whereas the whole idea of framing the problem of attack detection as anomaly detection is that the model can be trained suing only normal data.

If such a system were to be implemented in real life, with just one autoencoder for a whole (sub)network, and not separate models for each IoT device in the network, then the size of the autoencoder model, which increased quadratically with the number of features used, as shown by equation (6) and also visually demonstrated by Figure 11, does not seem as a significant concern.

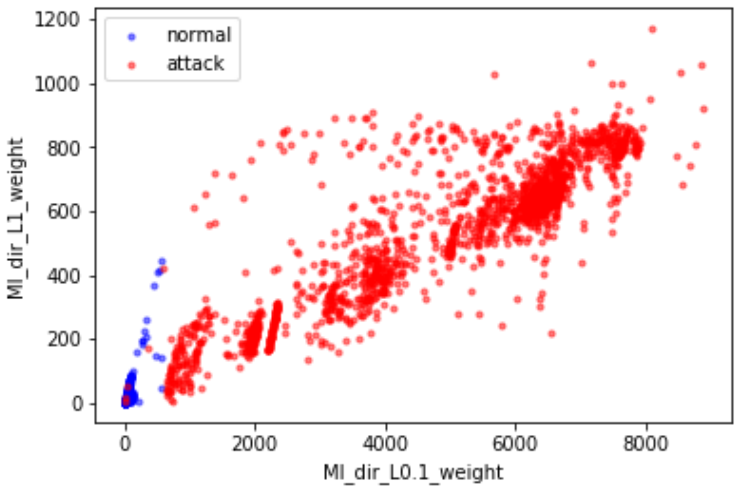


Figure 21. Plot for 2 most important attack detection features.

The result of the LIME technique applied to an attack data point, as presented in Figure 12 and Table 6, seems to be intuitively correct, in regard that most important conditions for this decision are identified as very high number of packets in the statistic.

4.2 Classifying attacks

Both botnet type classification and Mirai attack type classification showed very good results on the whole 115 feature set with accuracies over 0.99. However, on a very small feature sets the results we not as great, as shown in Table 7 and Table 9, even being substantially lower on 2 and 3 features than those achieved in [16] with more traditional ML. This could however, be attributed to different feature selection strategies.

Interestingly, for Mirai attack type classification on different feature sets (Table 9) the performance dramatically improved after including 5th feature from Table 3, which corresponded to a mean of the packet size during last minute time frame, while 4 features used before referred only to the variance within different time frames, which perhaps suggests that when selecting small number of features, diverse types should be included.

Not surprisingly, two most confused Mirai classes are UDP and UDPPlain with 88 data points confused between them, as seen on Figure 18. Both of them constitute a variation of a flooding attack using the same network protocol, namely User Datagram Protocol, and UDPPlain appears to be modified to not include payload [24].

In a similar vein as with attack detection, it is noticeable, that model accuracy does not degrade with increasing number of features, reflecting the models' ability to disregard unimportant data. In contrast, for example, k-Nearest Neighbors algorithm appeared to have a degraded in accuracy as can be seen in Table 7 when features increased from 2 to 3 and to 10.

LIME explanations for Mirai UDP attack type in Table 10, Table 11, Figure 20 present an interesting case. Firstly, the classifier is not absolutely certain that it is, in fact, a UDP attack, its predictions are split 70% for UDP and 30% for ACK type attack. Most important boundaries in favor of UDP are packet counts grouped by source and destination ports. According to research, a distinctive feature of UDP flood attack is port randomization [24], which means that in the statistics grouped by port it is quite possible to get only one packet for each combination of ports, which is in accordance with the data. But why is classifier not completely certain? The explanation for 30% probability class ACK points out the packet size variance. Out of 5 Mirai attack types present in the dataset, only UDP, UDPPlain and ACK can send the payload, according to research [24], with UDPPlain usually being used without payload to optimize for packets-per-second. And so, it seems that the classifier's relative uncertainly can be explained by the fact that both UDP and ACK attacks can have high payload variance.

4.3 Implementation in real environment

Using elements studied in this work, it could be possible to assemble a complete detection, classification and explanation pipeline as suggested by Figure 1. There would be however, many practical considerations.

The anomaly detection part of the pipeline should be easier to implement in practice, as there would be no need to obtain and maintain a dataset for different botnets and their attacks. It would still require sampling traffic from an IoT network that is not yet infected, of course, which could be a problem in and on itself. Further, it is not clear, how the method performance would change if the network itself is changed, for example: new devices are added. If it would be a device of the same type as the one we already have in the network, e.g. one more security camera, but from another manufacturer, perhaps the data distribution will not change much, but this is just speculation. As dataset actually contains multiple devices from similar categories, this could be something worth exploring.

The theoretically good false positive rate of 0.0015 in practice would still mean that more than 1 in 1000 normal data points would be falsely labeled as an attack. If a data point is generated every 100 milliseconds, that would give us a false alarm every 100 seconds, and this is not practical. One data point by itself, however, could hardly be considered an attack, where a continuous stream of packets is usually sent. To that regard, an approach suggested by [12] could be implemented, where an alarm only goes off if there is a majority of attack data points within a window of some specified size.

While the attack detection mechanism based on autoencoder could be developed locally, maybe by an engineer maintaining the IoT network, the botnet classifier presents a challenge in the form of data set requirements. Ideally, this is a task for a cybersecurity firm or a research university team, that specializes in IoT botnets and collects and maintains data sets needed to train an effective classifier that would recognize a wide range of different botnets and attack types.

5 Conclusions

Deep learning methods demonstrated good IoT botnet attack detection and classification accuracy. Moreover, these methods can work well with varying number of features and generally their performance does not suffer from extra features, meaning that in real world environment they offer the possibility of using all existing data features.

It was also shown that there may not be a need to create different models for each device in the network, and that one model trained on data from all devices can be just as good.

The lack of interpretability, one of the neural network limitations, can be successfully addressed using LIME, as was demonstrated by producing feature value boundaries which upon closer inspection agreed with common sense.

References

[1] K. L. Lueth, "State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating," [Online]. Available: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/. [Accessed 6 January 2019].

[2] A. D. Rayome, "DDoS attacks increased 91% in 2017 thanks to IoT," [Online]. Available: https://www.techrepublic.com/article/ddos-attacks-increased-91-in-2017-thanks-to-iot/. [Accessed 6 January 2019].

[3] M. Antonakakis, "Understanding the Mirai Botnet," in *Proceedings of the 26th USENIX Security Symposium*, Vancouver, 2017.

[4] M. Rouse, "IoT devices (internet of things devices)," [Online]. Available: https://internetofthingsagenda.techtarget.com/definition/IoT-device. [Accessed 6 January 2019].

[5] Z. Bazrafshan, et al, "A survey on heuristic malware detection techniques," in *2013 5th Conference on Information and Knowledge Technology (IKT)*, 2013.

[6] M. Stevanovic and J. M. Pedersen, "On the Use of Machine Learning for Identifying Botnet Network Traffic," *Journal of Cyber Security,* vol. 4, pp. 1-32, 2016.

[7] S. Saad, et al, "Detecting P2P botnets through network behavior analysis and machine learning," in *2011 Ninth Annual International Conference on Privacy, Security and Trust*, Montreal, 2011.

[8] S. C. Guntuku, P. Narang and C. Hota, "Real-time Peer-to-Peer Botnet Detection Framework based on Bayesian Regularized Neural Network".

[9] X. D. Hoang and Q. C. Nguyen, "Botnet Detection Based On Machine Learning Techniques Using DNS Query Data," *Future Internet,* vol. 10, no. 43, 2018.

[10] A. Kamal, et al, "A survey of botnet detection based on DNS," *Neural Computing and Applications,* 2015.

[11] D. H. Summerville, K. M. Zach and Y. Chen, "Ultra-lightweight deep packet anomaly detection for Internet of Things devices," in *IPCCC*, Nanjing, 2015.

[12] Y. Meidan, et al, "N-BaIoT: Network-based Detection of IoT Botnet Attacks Using Deep Autoencoders," *IEEE PERVASIVE COMPUTING,* vol. 18, no. 9, 2018.

[13] Y. Meidan, et al , "iot botnet dataset," [Online]. Available: https://archive.ics.uci.edu/ml/datasets/detection_of_IoT_botnet_attacks_N_BaIoT. [Accessed 27 December 2018].

[14] C. D. McDermott, F. Majdani and A. V. Petrovski, "Botnet Detection in the Internet of Things using Deep Learning Approaches," in *IJCNN*, Rio De Janeiro, 2018.

[15] S. Nõmm and H. Bahsi, "Unsupervised Anomaly Based Botnet Detection in IoT Networks," in *ICMLA*, 2018.

[16] H. Bahsi, S. Nõmm and F. B. L. Torre, "Dimensionality Reduction for Machine Learning Based IoT Botnet Detection," in *ICARCV*, Singapore, 2018.

[17] M. T. Ribeiro, S. Singh and C. Guestrin, "“Why Should I Trust You?” Explaining the Predictions of Any Classifier," in *KDD*, 2016, San Francisco.

[18] A. Ng, "Gradient Descent in Practice I - Feature Scaling," Coursera, [Online]. Available: https://www.coursera.org/lecture/machine-learning/gradient-descent-in-practice-i-feature-scaling-xx3Da. [Accessed 6 January 2019].

[19] C. C. Aggarwal, Data Mining: The Textbook, Springer, 2015.

[20] D. Rumelhart, G. Hinton and R. Williams, "Learning internal representations by error propagation.," in *Parallel Distributed Processing*, vol. 1, Cambridge, MA, MIT Press, 1986, p. 336.

[21] A. A. Pol, "Anomaly detection using Deep Autoencoders for the assessment of the quality of the data acquired by the CMS experiment," in *23rd International Conference on Computing in High Energy and Nuclear Physics*, Sofia, 2018.

[22] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016.

[23] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference for Learning Representations*, San Diego, 2015.

[24] C. Seaman, "Threat Advisory: Mirai Botnet," [Online]. Available: https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/akamai-mirai-botnet-threat-advisory.pdf. [Accessed 6 January 2019].

Appendix 1 – Fisher’s score calculation procedure

```
scored = []
indices = {}
shps = {}
for cl in classes:
    indices[cl] = df_fish['class'] == cl
    shps[cl] =  df_fish[indices[cl]].shape[0]

for col in df_fish.columns:
    if col == 'class':
        continue
    num = 0
    den = 0
    m = df_fish[col].mean()

    for cl in classes:
        num += (shps[cl] / df_fish.shape[0]) * (m -
df_fish[indices[cl]][col].mean())**2
        den += (shps[cl] / df_fish.shape[0]) *
df_fish[indices[cl]][col].var()
    score = {'feature': col, 'score': num / den}
    scored.append(score)
```

Appendix 2 – Autoencoder model creation function

```
def create_model(input_dim):
    inp = Input(shape=(input_dim,))
    encoder = Dense(int(math.ceil(0.75 * input_dim)), activation="tanh")(inp)
    encoder = Dense(int(math.ceil(0.5 * input_dim)),
activation="tanh")(encoder)
    encoder = Dense(int(math.ceil(0.25 * input_dim)),
activation="tanh")(encoder)
    decoder = Dense(int(math.ceil(0.5 * input_dim)),
activation="tanh")(encoder)
    decoder = Dense(int(math.ceil(0.75 * input_dim)),
activation="tanh")(decoder)
    decoder = Dense(input_dim)(decoder)
    return Model(inp, decoder)
```

Appendix 3 – Deep Neural Network model creation function

```
def create_model(input_dim, hidden_layer_size, num_of_classes):
    model = Sequential()
    model.add(Dense(hidden_layer_size, activation="tanh",
input_shape=(input_dim,)))
    model.add(Dense(hidden_layer_size, activation="tanh"))
    model.add(Dense(num_of_classes))
    model.add(Activation('softmax'))
    return model
```

Appendix 4 – Autoencoder training

```
    model = create_model(top_n_features)
    model.compile(loss="mean_squared_error",
                    optimizer="adam")
    cp = ModelCheckpoint(filepath=f"anomaly/anomaly{top_n_features}.h5",
                                 monitor='val_loss',
                               save_best_only=True,
                               verbose=0)

    es = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)

    epochs = 100
    history = model.fit(X_train, X_train,
                    epochs=epochs,
                    validation_data=(X_opt, X_opt),
                    verbose=1,
                    callbacks=[cp, es])
```

Appendix 5 – Classifier training

```
    model = create_model(top_n_features, 8, 3)
    model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
    cp = ModelCheckpoint(filepath=f'./models/model_{top_n_features}.h5',
                               save_best_only=True,
                               verbose=0)
    es = EarlyStopping(patience=5, monitor='val_acc',
restore_best_weights=True)
    epochs = 100

    history = model.fit(X_train, y_train,
                    epochs=epochs,
                    validation_split=0.2,
                    verbose=1,
                    callbacks=[cp, es])
```