JOÃO ANDRÉ SIMIONI

AN ENERGY-EFFICIENT INTRUSION DETECTION OFFLOADING BASED ON DNN FOR EDGE COMPUTING

CURITIBA

2025

JOÃO ANDRÉ SIMIONI

AN ENERGY-EFFICIENT INTRUSION DETECTION OFFLOADING BASED ON DNN FOR EDGE COMPUTING

Dissertation presented to the Post-Graduate Program in Computer Science - PPGIa - of Pontifical Catholic University of Paraná, as a partial requirement for the attainment of the Master of Science degree in Information Systems.

Concentration Area: Computer Science

Advisor: PhD. Eduardo Kugler Viegas
Co-advisor: PhD. Altair Olivo Santin

CURITIBA

2025

Abstract

To improve the accuracy of Deep Neural Networks (DNNs) applied to Network Intrusion Detection Systems (NIDS) researchers often increase the complexity of their designed model. Given the processing limitations of resource-constrained devices, researchers have proposed offloading the NIDS task to the cloud. However, simultaneously ensuring energy efficiency and detection accuracy remains a challenge. This work proposes a DNN-based NIDS through early exits that operate following an energy-efficient edge-computing architecture implemented twofold. Firstly, a DNN-based NIDS is proposed, employing multi-objective optimization for efficient inference and computation offloading. It is designed to perform the classification task at the edge device and configured to proactively offload events to the cloud when additional processing capabilities are required. The insight is to utilize multi-objective optimization to identify the optimal balance between accuracy and energy efficiency in task offloading. Secondly, the final DNN branch performs classification with a reject option to ensure reliability when analyzing new network traffic patterns, while calibration adjusts the model's confidence values to enhance generalization. The rejection mechanism allows the model to accept only its most confident classifications enhancing the model's generalization capabilities. Experiments conducted with the proposal's prototype through a new intrusion dataset encompassing one-year-long network traffic with over 7TB of data attested to the proposal's feasibility. It can reduce the edge device's energy consumption and processing costs to only 1%, while maintaining accuracy. This is achieved while demanding the offloading of only 10% of network events to the cloud, optimizing resource utilization across both the edge and cloud infrastructures.

Keywords: *Intrusion Detection, Deep Learning, Early Exits, Energy-efficiency, Edge Computing.*

List of Images

Figure 1 - R_t population is formed by the combination of P_t and Q_t (with size 2N) and sorted by non-domination. Solutions from the best non-dominated sets ($F_1 - F_N$) are selected for the new population P_{t+1}, with size N. Selection, crossover and mutation are used to form Q_{t+1}. 22

Figure 2 - Graphical representation of genetic operations. 23

Figure 3 – Typical ML workflow. 24

Figure 4 - Basic ANN Sample 25

Figure 5 – Example of convolutional layer with a 3x3 kernel and stride 1. 27

Figure 6 - 3 filters applied to one 6x6 layer, with a 3x3 kernel and stride 1, resulting in 3 4x4 layers. 28

Figure 7 - A 2x2 pooling on an 8x8 layer results in a 4x4 layer. 28

Figure 8 - AlexNet diagram. 29

Figure 9 - *MAWIFlow* network flow distribution over the year. 48

Figure 10 - Accuracy trend over a year of commonly used classifiers without periodic model updates. Classifier is trained in January and evaluated in subsequent months without updates. 50

Figure 11 - Workflow of the proposed DNN-based NIDS through early exits for intrusion detection offloading in EC settings. DNN Branches are distributed from end device, edge, and cloud infrastructure. 52

Figure 12 - multi-objective optimization for DNN-based NIDS with Early Exits. Here, σ_i measures the average DNN inference energy consumption, and ϵ_i measures the error rate obtained using the selected acceptance thresholds $t^c{}_i$. 55

Figure 13 - Prototype implementation overview of the proposed model. 62

Figure 14 - multi-objective optimization for evaluated architectures. Processing time was measured on a Raspberry PI Model B. Error rate was measured on *MAWIFlow* Feb. and March. 65

Figure 15 - Reliability histograms for the 1st MobileNetV2 DNN branch on MAWIFlow January validation dataset. 66

Figure 16 - Accuracy and rejection performance of the model as time passes on MAWIFlow dataset. DNNs are trained on MAWIFlow January training dataset. Inference uses the chosen operation point and the proposed calibration scheme. 67

Figure 17 - F-Score performance improvement of the proposed scheme vs. the traditional approach. 68

Figure 18 - Average inference computational time and energy consumption per event at the *End Device* with the proposed model. *Traditional* denotes the execution of the entire DNN at the *End Device* without using the early exit approach. *Ours (No offload)* denotes using the proposed approach, with both exits processed at the *End Device*. *Ours (Offload)* denotes using the proposed approach, offloading the 2nd DNN branch to the Cloud Infrastructure. 69

Figure 19 - Evaluation of the trade-offs incurred by offloading from *End Device* to *Cloud Infrastructure* the event classification (for each event reaching 2nd DNN branch). *End Device* is deployed in Brazil south region. 71

Figure 20 - Trade-off on process time vs offload rate to Cloud Infrastructure. The offload rate must be defined based on the operator's needs and can be adjusted accordingly. 72

List of Tables

Table 1 - MobileNetV2 architecture [31] 30

Table 2 - Summary of related works 42

Table 3 - Selected features of chosen dataset [18]. 45

Table 4 - MAWIFlow dataset statistics. 48

Table 5 - Average event detection throughput (events / sec). 51

Table 6 - Average event inference time of the proposed model according to the deployed Cloud Infrastructure zone. 70

List of Equations

Eq. 1 - Typical metrics for NIDS evaluation 19

Eq. 2 - typical optimization problem 21

Eq. 3 - Cross Entropy Loss function 26

Eq. 4 - Binary Entropy Loss Function 26

Eq. 5 - Softmax function 26

Eq. 6 - Joint loss function 31

Eq. 7 - Rejector module 54

Eq. 8 - multi-objective optimization objective 1 - time 56

Eq. 9 - multi-objective optimization objective 2 - error 56

Eq. 10 - Average bin accuracy 58

Eq. 11 - Average bin confidence 58

Eq. 12 - Expected Confidence Level - ECE 58

Eq. 13 - Calibrated softmax function with temperature factor 59

List of Abbreviations and Acronyms

ANN	*Artificial Neural Networks*
CNN	*Convolutional Neural Network*
CPU	*Central Processing Unit*
CRT	*Class Related Thresholds*
FC	*Fully Connected Layer*
FN	*False Negative*
FP	*False Positive*
GPU	*Graphics Processing Unit*
IDS	*Intrusion Detection System*
ILSVRC	*ImageNet Large-Scale Visual Recognition Challenge*
KNN	*K - Nearest Neighbors*
LSTM	*Long Short-Term Memory*
MAX	*Maximum Value*
MIN	*Minimum Value*
NIDS	*Network-based Intrusion Detection System*
NSGA-II	*Non-dominated sorting genetic algorithm II*
RBM	*Restricted Boltzmann Machine*
RGB	*Red, Green Blue (color model)*
RQ	*Research Question*
SGD	*Stochastic Gradient Descent*
TP	*True Positive*
TN	*True Negative*

Summary

CHAPTER 1 INTRODUCTION ... 11

1.1. Motivation and Hypothesis ... 12
1.2. Objectives ... 14
1.3. Contributions ... 15
1.4. Publications ... 16
1.5. Organization of this Document ... 17

CHAPTER 2 BACKGROUND ... 18

2.1. Network-based Intrusion Detection for IoT ... 18
2.1.1. Signature Based ... 19
2.1.2. Behavior-based ... 20
2.2. Multi-Objective Optimization ... 21
2.3. Machine Learning ... 23
2.4. Neural Networks ... 24
2.4.1. CNN Architectures ... 27
2.4.1.1. AlexNet ... 29
2.4.1.2. MobileNetV2 ... 29
2.5. Early Exits ... 30
2.6. Classification with a Reject Option ... 31
2.7. Edge-Computing for NIDS ... 32
2.8. Conclusions ... 33

CHAPTER 3 RELATED WORK ... 34

3.1. Machine Learning for NIDS ... 34
3.2. Deep Learning for NIDS ... 35
3.3. Resource Optimization for NIDS on IoT ... 37
3.4. Multi objective for NIDS ... 38
3.5. Early Exits ... 39
3.6. Edge Offloading ... 41
3.7. Related work discussion ... 42
3.8. Conclusions ... 43

CHAPTER 4 PROBLEM STATEMENT ... 47

4.1. MAWIFlow 47
4.2. Traditional DNN performance 48
4.3. Conclusions 51
CHAPTER 5 METHODOLOGY 52
5.1. Proposed Model 52
5.2. Early Exits implementation 53
5.3. Multi-objective Optimization 55
5.4. Calibration 57
5.5. Conclusions 59
CHAPTER 6 RESULTS AND EVALUATION 61
6.1. Prototype 61
6.2. Model Building 63
6.3. Energy Consumption Measurement 63
6.4. DNN-based NIDS with Early Exits 64
6.5. Cloud Offloading 68
6.6. Conclusions 72
CHAPTER 7 CONCLUSION 74
BIBLIOGRAPHIC REFERENCES 76

Chapter 1

Introduction

The utilization of resource-constrained devices, such as those in the context of Internet of Things (IoT), has consistently risen in recent years. These devices typically consist of battery-powered embedded computing systems with restricted processing capabilities, often featuring network connectivity. Due to their widespread adoption, these devices have become prime targets for cyber attackers, with reports indicating a 40% surge in attack incidents over the past year alone [1].

To safeguard IoT devices, a common approach involves the use of Network-based Intrusion Detection System (NIDS), which monitors the device's network traffic for malicious footprints [2]. Over the past years, several techniques have been proposed to conduct such a task, wherein proposed approaches can usually be categorized as misuse-based or behavior-based schemes [3]. On the one hand, misuse-based approaches identify intrusion attempts by comparing them to a database of well-known malicious activities, limiting their ability to detect only previously known attacks. On the other hand, behavior-based techniques identify irregularities based on deviations from a previously established system profile, potentially detecting unseen attack types.

In response to this feature, several works have proposed new behavior-based techniques for intrusion detection, where approaches based on Deep Neural Networks (DNNs) frequently achieve the highest accuracy levels [4]. To accomplish such an objective, researchers, in their vast majority, escalate the parameters of their designed DNN architecture to pave the way for better accuracy. Besides increasing the inference computational costs, such an approach also renders them unsuitable for resource-constrained devices, given their limited memory and processing capabilities [5].

1.1. Motivation and Hypothesis

At deployment, the application of DNN-based intrusion detection entails forwarding input parameters throughout the entire network architecture until the output layer is reached. In this process, multiple spatial non-linear patterns and features are extracted, subsequently acting as indicators for the classification task conducted at the output layer. Therefore, regardless of the evaluated event's complexity, be it complex or simple, the decision can only be made once all indicators have been extracted, potentially leading to the inefficient use of computational resources [6]. Considering this, in recent years, several studies have explored the introduction of early exits in DNN architectures. Early exits add multiple termination points that split the network into branches, enabling the inference task to conclude prematurely if the current extracted patterns and features can reliably lead to a decision [7].

While early exits typically reduce the DNN inference computational cost with minimal impact on accuracy, they are not readily applicable to network intrusion detection, particularly on resource-constrained devices [8]. Unlike other domains, the behavior of network traffic is notably dynamic and continually evolves. The prior necessitates advanced DNN model generalization capabilities, while the latter can usually only be fixed through model updates [9]. Conversely, existing intrusion detection strategies based on early exits often fail to consider the tradeoffs in model generalization that arise from prematurely terminating the inference task. In addition, the classification unreliability resulting from an outdated DNN model due to changing network traffic behavior is generally assumed to be rectified through model updates. However, this procedure frequently involves an extended time frame, often spanning days or even weeks, requiring the deployed model to have a prolonged lifespan to ensure its reliability.

The effectiveness of early exits relies on the precise definition of thresholds that dictate the point at which an input should be allowed to exit the network [10]. These thresholds are pivotal because they balance the trade-off between computational resource usage and the accuracy of the output. As an input progresses deeper into the network, more computational resources are consumed due to the increased complexity and number

of parameters encountered in the later stages. Although this results in higher accuracy, as the later layers are capable of capturing more nuanced features and relationships in the data, it also means that unnecessary computation may be expended on inputs that could have been satisfactorily resolved at earlier points. Therefore, setting the right thresholds for early exits is a delicate task that requires careful consideration of the specific requirements of the task at hand, ensuring that the network maintains an optimal balance between efficiency and the quality of its predictions.

This trade-off between computational efficiency and accuracy is emblematic of a multi-objective optimization problem, where the improvement of one objective often comes at the cost of another [11]. In the context of neural networks with early exits, the dual objectives of minimizing computational resources and maximizing accuracy are inherently conflicting. Multi-objective optimization techniques are thus pivotal in navigating this complexity, as they enable the identification of a set of optimal solutions, known as Pareto optimal solutions, rather than a single definitive answer. These techniques facilitate the exploration of the trade-off curve, providing a spectrum of solutions that balance efficiency and accuracy in different measures. However, it is important to note that the selection among these solutions does not follow a one-size-fits-all approach. Instead, it requires the operator to make a decision based on the specific priorities and constraints of the application at hand. The choice among potential solutions ultimately depends on how the operator values the trade-offs between the variables being evaluated, highlighting the subjective nature of determining what constitutes the "best" solution in multi-objective optimization tasks.

As a way to reduce the processing requirements in the devices, Edge Computing (EC) has emerged as a solution, where the edge, or an intermediate infrastructure device, may cooperate with the cloud to conduct the processing task [12]. More specifically, intrusion detection can be achieved by offloading the DNN-based classification task from the edge device to the edge infrastructure or the cloud. Unfortunately, offloading all network traffic for classification purposes is not easily achievable. This is because network traffic must be classified with minimal processing delays, ensuring that identified attacks are blocked promptly [13]. Notwithstanding, offloading the network classification task can quickly exhaust the edge device’s bandwidth if all traffic is forwarded. In

practical terms, edge devices must reliably and autonomously identify which network traffic needs to be further assessed in the cloud [14]. Adequate NIDS task offloading can improve the overall IoT system's energy efficiency, as both edge and cloud infrastructure can benefit from lower resource usage.

1.2. Objectives

In the light of this, the main objective of this work is to present a new early exit DNN aiming for fast inference time while keeping intrusion detection reliability. To achieve it, this work aligns with the following specific objectives to trail the path:

In the light of this, the main objective of this work is to present a new DNN-based NIDS through early exits that operate following an energy-efficient edge computing architecture, implemented threefold:

Firstly, a DNN-based NIDS is conducted through early exits coped with a multi-objective optimization strategy for adequate classification offloading. The model implements the classification task at the edge and proactively offloads events to the cloud when additional processing capabilities are required. The insight is to leverage multi-objective optimization to find the optimal compromise between accuracy and energy efficiency due to cloud offloading.

Secondly, to ensure reliability with new network traffic behavior, classification is conducted with a reject option at the last DNN branch while also calibrating the model's confidence values. The rejection ensures that only highly confident classifications are accepted by the model, while the calibration improves the model generalization.

Thirdly, the proposed approach is implemented within an energy-efficient edge computing architecture encompassing edge devices and cloud infrastructure. The scheme enables reliable NIDS classification offloading from edge devices to cloud infrastructure while optimizing energy efficiency and classification latency.

To achieve it, this work aligns with the following specific objectives to trail the path:

- Adapt well known DNN models to use the early exit strategy on intrusion detection domain, building their models and creating the required set of tools to perform the training and evaluation process.
- Address the early exit DNN as a multi-objective optimization task, where one of the objectives is to minimize the inference time, while the other is to maximize the accuracy.
- Introduce a classification approach with a reject option at the final DNN branch, allowing the rejection of potentially unreliable intrusion classifications influenced by new network traffic behavior. This rejection option will integrate the multi-objective optimization task.
- Evaluate and compare the performance of the proposed model against the regular DNN models, without the early exit approach.
- Propose an offload model that can leverage edge and cloud computing, to distribute the processing resources.

As a result, this ensures classification reliability through the classification with a reject option while simultaneously reducing inference computational costs by employing early exits and Edge or Cloud offloading.

1.3. Contributions

In summary, the main contributions of this work are:

- An evaluation of widely used DNN-based NIDS concerning their accuracy, energy consumption and processing costs. The experiments reveal that current approaches demand unfeasible amounts of processing and energy while also experiencing a rapid degradation in accuracy over time.

- A new DNN-based NIDS with early exits that operates within an energy-efficient architecture. The proposed scheme can autonomously offload computation to cloud infrastructure when required while also dealing with changes in network traffic behavior.

- A proposal prototype that demonstrates our scheme feasibility under a variety of energy-efficient EC deployment settings. The scheme can reduce the edge device's energy consumption and processing costs to only 1%, while keeping or improving F1-Score by 0.02. This is achieved while demanding the offloading of only 10% of network events to the cloud, leading to resource optimization on both the edge and cloud infrastructures.

1.4. Publications

- SIMIONI, João André; VIEGAS, Eduardo Kugler; SANTIN, Altair Olivo; MATOS, Everton de; An Energy-efficient Intrusion Detection Offloading Based on DNN for Edge Computing; IEEE Internet of Things Journal 2025. DOI: https://doi.org/10.1109/JIOT.2025.3544060

- SIMIONI, João André; VIEGAS, Eduardo Kugler; SANTIN, Altair Olivo; HORCHULHACK, Pedro. An Early Exit Deep Neural Network for Fast Inference Intrusion Detection. In: The 40th ACM/SIGAPP Symposium on Applied Computing (SAC), Sicily, Italy, 2025.

- HORCHULHACK, Pedro; VIEGAS, Eduardo Kugler; SANTIN, Altair Olivo; SIMIONI, João André. Fortalecendo a Segurança de Redes: Um Olhar Profundo na Detecção de Intrusões com CNN Baseada em Imagens e Aprendizado por Transferência. In: Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, 2024, Brasil. Anais do XLII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2024). p. 449. DOI: https://doi.org/10.5753/sbrc.2024.1420

- SIMIONI, João André; VIEGAS, Eduardo Kugler; SANTIN, Altair Olivo; HORCHULHACK, Pedro. Detecção de Intrusão Através de Redes Neurais

Profundas com Saídas Antecipadas para Inferência Rápida e Confiável. In: SIMPÓSIO BRASILEIRO DE SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS (SBSEG), 24., 2024, São José dos Campos/SP. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2024. p. 242-255. DOI: https://doi.org/10.5753/sbseg.2024.241485.

- HORCHULHACK, Pedro; VIEGAS, Eduardo Kugler; SANTIN, Altair Olivo; SIMIONI, João A.. Network-based Intrusion Detection Through Image-based CNN and Transfer Learning. In: 2024 International Wireless Communications and Mobile Computing (IWCMC), 2024, Ayia Napa. 2024 International Wireless Communications and Mobile Computing (IWCMC), 2024. p. 386. DOI: https://doi.org/10.1109/IWCMC61514.2024.10592364

Software registry obtained:

- Certificado de Registro de Programa de Computador, process Nº: BR512024004346-2, with the title: *Uma ferramenta para offloading de detecção de intrusão com eficiência energética baseado em DNN para computação em nuvem de borda* and publication date of August 1st, 2024.

1.5. Organization of this Document

The remainder of this document is organized as follows: Chapter 2 presents the background required to understand and implement the proposed work. Chapter 3 presents and discusses existing works in literature that address different aspects of the elements used. Chapter 4 presents the problem statement, showing the performance based on the traditional approaches. Chapter 5 presents the methodology used for the current state of the work. Chapter 6 presents the results of the proposed model and finally chapter 7 presents this work conclusions.

Chapter 2
Background

This section outlines the key concepts and background necessary for understanding the problem addressed by this work. It begins with an exploration of Intrusion Detection Systems (IDSs) - their definitions, common implementations, and their significance in the field. Following this, the discussion moves to optimization problems, with a particular focus on those involving multiple objectives, setting the stage for the methodologies employed in the study. The narrative then transitions to a brief overview of machine learning (ML), culminating in an introduction to Neural Networks (NN) and the specific models that have been utilized in the research. Finally, the text details the two innovative techniques employed in the proposed model: Early Exits and Classification with a Rejection option, elucidating their functions and contributions to the solution.

2.1. Network-based Intrusion Detection for IoT

The objective of an Intrusion Detection System (IDS) is to detect unauthorized access attempts to a device [15]. These attempts are usually called *attacks*. Depending on where the system acts, they can be categorized as HIDS (host-based intrusion detection systems), NIDS (network-based intrusion detection systems) or HIDS (hybrid intrusion detection systems). When the processed data comes directly from the device (host), such as application logs and memory inspection, the IDS is classified as host based. If, on the other hand, network traffic inspection is used, the IDS is classified as network based. Approaches that use both techniques are named HIDS.

Besides the location of the data, the systems can be further classified depending on the technique utilized to analyze the collected information [16]. The two main techniques used are: *signature-based* and *behavior-based*. Other terms are commonly

seen in literature to refer to the same techniques. Signature-based is also known as misuse-based and the term anomaly-based is also widely used in literature to refer to the behavior-based technique.

When evaluating the effectiveness of these alternatives, some typical measurements are used:

- True Positive (TP): number of attack samples correctly classified as an attack.
- True Negative (TN): number of normal samples correctly classified as normal.
- False Positive (FP): number of normal samples incorrectly classified as an attack.
- False Negative (FN): number of attack samples incorrectly classified as normal.

Further, F-Measure is measured according to the harmonic mean of precision and recall values while considering attack samples as positive and normal samples as negative, as shown in Eq. 1.

Eq. 1 - Typical metrics for NIDS evaluation

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

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

$$F - Measure = 2\ x\ \frac{Precision\ x\ Recall}{Precision + Recall}$$

2.1.1. Signature Based

Signature-based NIDSs rely on pre-defined, well-known traffic sequences or patterns to detect an attack. These sequences or patterns are called *signatures*. An example of such a signature is a sequence of packets requesting to open a connection to many different ports, which could characterize a port scanning action. Another example is a multiple attempt to login with different passwords, which could be identified as a brute force attack to try to figure out credentials for a user.

The main drawback of this approach is that, while very effective to detect well-known attacks, it behaves poorly against new or not known attacks [17], being especially vulnerable to newly discovered attack forms and highly dependent on system updates.

2.1.2. Behavior-based

Behavior-based NIDSs, on the other hand, use a different approach. It compares the traffic data against the normal operation traffic, looking for anomalies on the behavior, which are flagged as being a potential attack. They have been extensively used by network operators for monitoring the device's communication and detecting malicious activities [2]. In general, these tools implement Machine Learning (ML) or Deep Learning (DL) concepts [3], following a four sequential module implementation, encompassing Data Acquisition, Feature Extraction, Classification, and Alert. The first module is responsible for the real-time collection of network packets from a specified Network Interface Card (NIC). The behavior of the collected data is extracted by the Feature Extraction module, which typically compiles a feature vector summarizing the communication between network entities within a specified time window. The resulting behavioral vector serves as input to the Classification module, which employs a previously trained model to classify events as either normal or attack. Lastly, the Alert module signals events classified as attack.

DL techniques are usually implemented using Deep Neural Networks (DNNs) and their use for network traffic classification requires the prior execution of the training task [18]. The behavior of a training dataset is extracted during the training phase. Thus, it should ideally consist of millions of labeled network samples representative of what is expected in a production environment. The resulting model's accuracy is assessed during the testing phase, and the measured accuracy rates are expected to indicate its performance in a production deployment.

IoT devices are often characterized by their resource constraints and limited processing power, and these can pose challenges when using complex neural networks, as the inference process can be time consuming making the NIDS ineffective. Also,

energy consumption is a common challenge with these devices, since, for many usage types, they are placed in remote locations and depend on batteries to operate. Memory and bandwidth limitations may also impose additional challenges to NIDS that use Neural Networks, if they need to update their models. Models may become outdated and unable to adapt to new attack vectors or changes in network patterns, putting IoT ecosystems at risk.

2.2. Multi-Objective Optimization

An optimization problem is a mathematical or computational problem where the goal is to find the best solution from a set of feasible solutions. The *best* solution is determined based on certain criteria, which are defined by an objective function. The objective function quantifies the performance or quality of a solution, and the optimization process aims to either maximize or minimize this function.

Formally, an optimization problem can be defined as follows:

Eq. 2 - typical optimization problem

$$Minimize\ or\ Maximize: f(x)$$
$$x \in X$$

Where $f(x)$ represents the objective function that needs to be optimized, x the decision variables and X the set of possible values for x. The optimization problem may be subject of some constraints, which would limit the set X.

In traditional optimization problems, there is usually a single objective to be maximized or minimized, however, in many real-world scenarios, there are multiple criteria or goals that need to be considered, and these objectives may conflict with each other [11].

In the set of solutions, some of them are said to be dominated by another one, when another solution is equal or better than the current one in all objectives. If the solution is not dominated by another one it is called *Pareto optimal*, and it means that it cannot be improved in any objective without compromising another one. The final step in the multi-objective optimization task is to find a Pareto optimal set, composed only by

Pareto optimal options. In the Pareto optimal set, no solution can be said to be better than the other, without considering external factors. So, it depends on the system's operator to identify which solution to adopt from the set.

One popular algorithm to solve multi-objective optimization problems is the non-dominated sorting genetic algorithm-II (NSGA-II) [19]. It is an extension on the original NSGA [20] algorithm to improve its efficiency. The key features of the algorithm are a Non-Dominated Sorting approach to categorize the solutions based on the dominance relationship, Crowding Distance, to maintain the diversity of the population to keep them spread over the Pareto set. Figure 1 shows the procedure of NSGA-II [19].

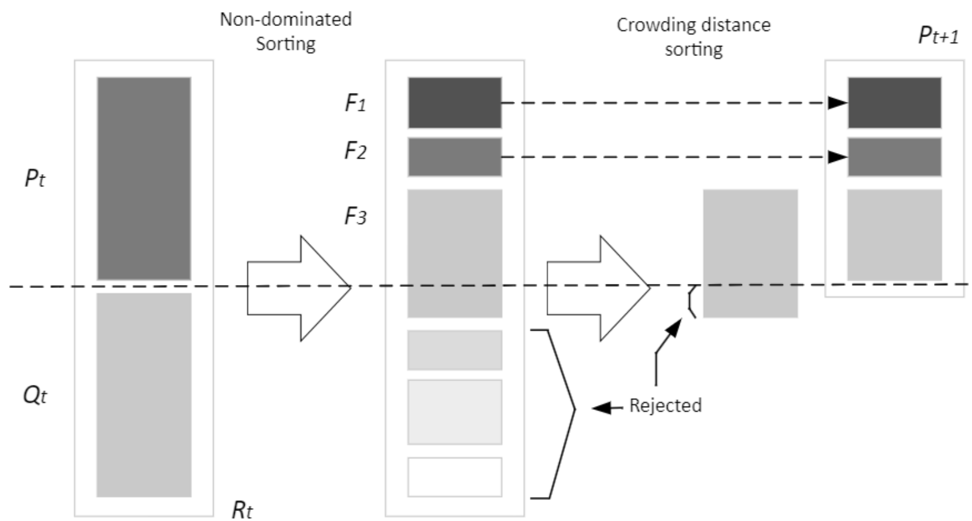


Figure 1 - R_t population is formed by the combination of P_t and Q_t (with size 2N) and sorted by non-domination. Solutions from the best non-dominated sets (F_1 – F_N) are selected for the new population P_{t+1}, with size N. Selection, crossover and mutation are used to form Q_{t+1}.
Source: adapted from [19]

Selection, crossover, and mutation are genetic operators to evolve the population between the generations [21]. Selection chooses individuals from the current population based on some criteria to form generation. Mutation introduces small random changes to selected individuals, promoting diversity in the population. Lastly, crossover combines genetic material from two parent individuals to create offspring, mimicking the recombination of genetic material in biological reproduction. Figure 2 shows a graphic representation of these genetic operators.

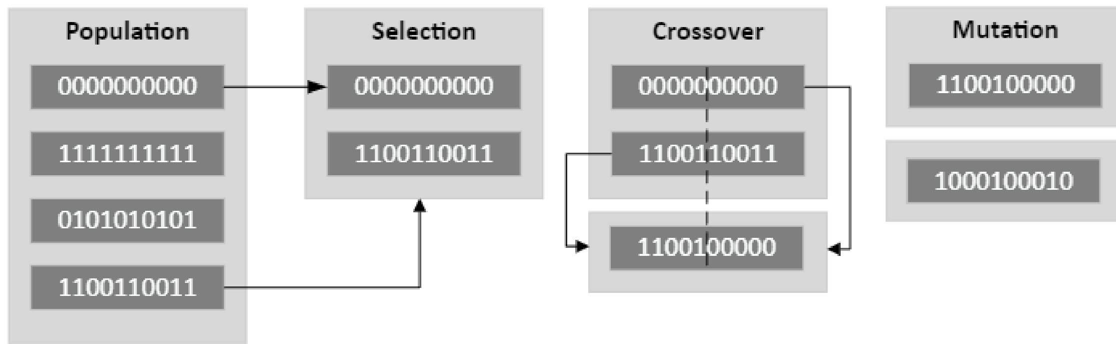


Figure 2 - Graphical representation of genetic operations.

2.3. Machine Learning

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) that focuses on the development of algorithms and computational models that enable systems to learn from data and improve their performance in specific tasks without being explicitly programmed. The learning process in ML involves identifying patterns and generalizing from data, allowing systems to make predictions, decisions, or automate tasks based on past experiences [22].

ML can be broadly categorized into two main types [23]: supervised learning and unsupervised learning. The main difference between these two classes is that supervised learning requires the training data to have a label on each of its samples, and the ML model is trained to recognize the labels based on the remaining characteristics of the sample. For classification problems, where there is a limited set of labels, the term class is used for the labels. The remaining characteristics are called features.

In a traditional supervised ML problem, the initial step is to extract the features from the evaluated population, followed by labeling the samples with the corresponding classes. This set of labeled samples is called a dataset and is used to train the ML model. Usually, the full dataset is split into two groups. One is used to train the model, while the other is used to evaluate the performance of the trained model.

Once the trained model is ready, it can be used to classify new, not known samples, in a process called inference. One important aspect of the inference process is that the data to be fed into the trained model must have the same set of features that was

used to build and train the model. This poses an additional challenge, mainly for NIDSs, since the process of converting network streams into a feature set is time and resource consuming [24]. Figure 3 show a summary of this process and considers an additional step of monitoring the inference output on the trained model. The inferred data can be fed into the model to allow further improvements in the model.

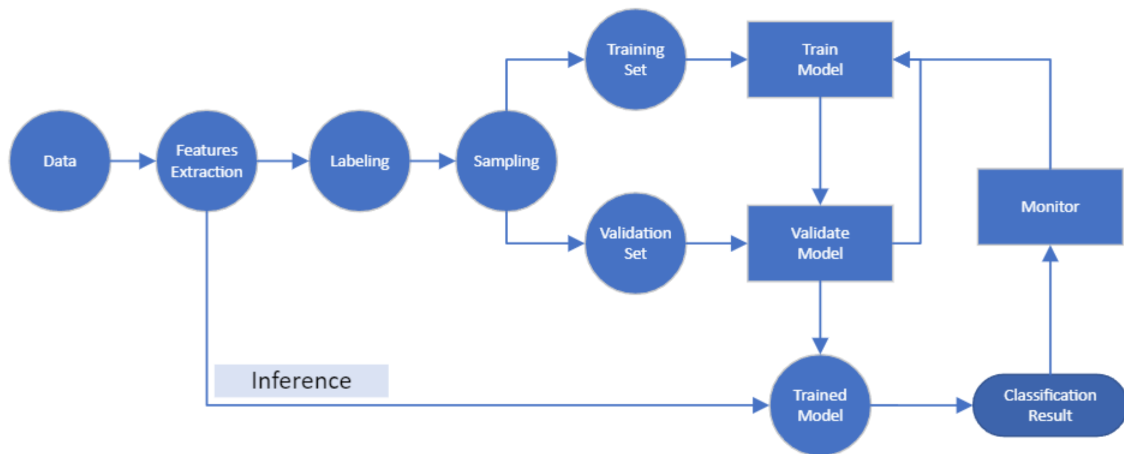


Figure 3 – Typical ML workflow.

2.4. Neural Networks

Neural Networks (NN) or Artificial Neural Networks (ANNs) are a class of machine learning models inspired by the structure and functioning of the human brain. They consist of interconnected nodes, called neurons, organized into layers. A basic representation of an ANN can be seen in Figure 4.

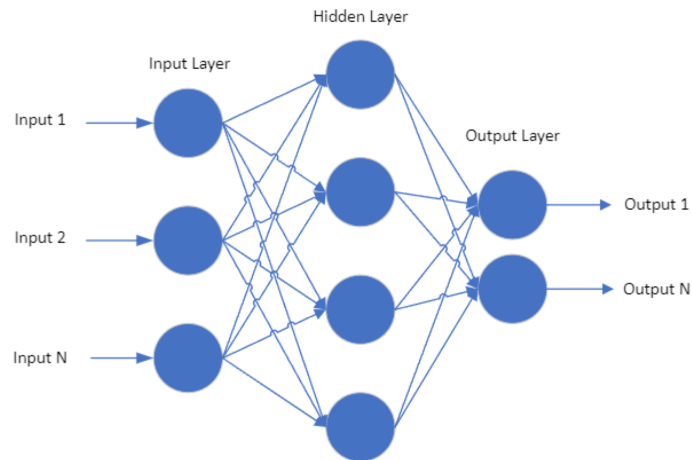


Figure 4 - Basic ANN Sample

The input layer is composed of as many inputs as the number of features from the dataset used. The output layer, for a classification problem, has as many neurons as the number of classes present on the dataset. The number of intermediate layers is a project decision and will impact on the model size, complexity and performance, and ANNs with many intermediate layers are also known as Deep Neural Networks (DNNs). Many ANNs architectures are the subject of studies in literature, where the most common are the Recurring Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) [25].

The basic flow is that, in each neuron, the inputs receive a weight and are summed up. This sum is then applied to the neuron activation function, and the output is forwarded to the next neuron. The ANN learning process gradually sets these weights, through each training cycle. In each training cycle, the output is compared to the expected output of the sample, using a loss function, and its results are applied back to the network using a process called backpropagation. The objective of training a neural network is to minimize this loss function, thereby improving the model's prediction accuracy. One commonly used loss function, especially for classification problems, is known as *Cross-Entropy Loss*. It measures the classification performance of a model with outputs which have a probability, measured between 0 and 1. For a multi-class classification problem with C classes, the general formula is:

Eq. 3 - Cross Entropy Loss function

$$Cross\ Entropy\ Loss = -\sum_{i=1}^{C} y_i \log(p_i)$$

where $y_i = 1$ if i is the actual class or 0, otherwise, and p_i is the predicted probability that the instance belongs to class i. Specifically for binary classifications, which is the case for this work, the equation can be summarized as:

Eq. 4 - Binary Entropy Loss Function

$$Binary\ Entropy\ Loss = -(y\ log(p) + (1-y)log(1-p))$$

with y being the true class label (0 or 1) and p the predicted probability that the instance belongs to class 1.

There are special layers which try to solve specific problems found in DNNs models, such as dropout layers, which randomly drops (sets to zero) a fraction of the inputs, to improve the model generalization, pooling layers, which down sample the spatial dimensions of the input volume and SoftMax layers [26]. SoftMax layers are a specific type particularly used in the output layer of a classification model. They are commonly employed to convert the raw output scores (also known as logits) produced by the model into probabilities. The SoftMax function is often used in multi-class classification problems. It takes a vector of real numbers as input and transforms them into a probability distribution. The function is defined as follows for an input vector z of length K:

Eq. 5 - Softmax function

$$Softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where:

- $Softmax(z)_i$ is the i-th element of the output vector after applying the softmax function.
- e is the base of the natural logarithm.
- z_i is the i-th element of the input vector.

- $\sum_{j=1}^{K} e^{z_j}$ is the sum of exponentiated values across all elements of the input vector.

The softmax function essentially normalizes the input vector, ensuring that the output values fall within the range (0, 1) and sum to 1. These normalized values can be interpreted as probabilities. The higher the score for a particular class, the higher the probability assigned to that class.

2.4.1. *CNN Architectures*

CNNs are a specific class of DNNs that uses convolutional layers for processing structured grid data, such as images. They use three basic ideas: local receptive fields, shared weights, and pooling [26]. Instead of connecting all the neurons of one layer to the next layer, convolutional layers will slide a filter, known as kernel, over the input data, capturing local patterns of the data, which refers to the local receptive field idea.

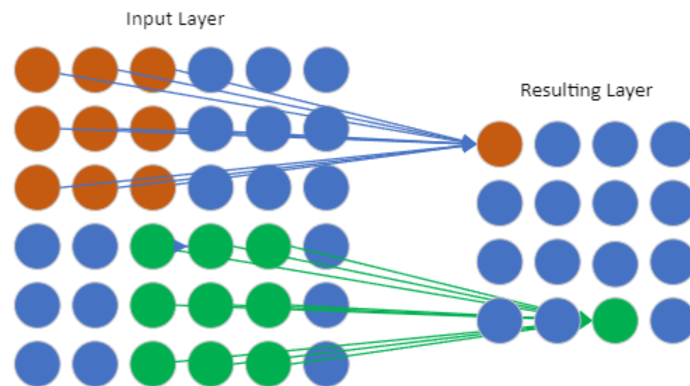


Figure 5 – Example of convolutional layer with a 3x3 kernel and stride 1.

The shared weights refer to the principle that the weights applied for the calculation of the resultant neuron in the next layer are the same for every group in the input layer. Figure 5 shows an example of a 6x6 input layer, using a 3x3 kernel and stride 1. The stride refers to the step size used when sliding the kernel over the input data. Many filters can be used, generating a set of resulting layers, as shown in Figure 6.

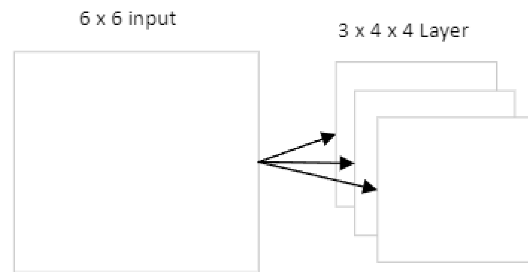


Figure 6 - 3 filters applied to one 6x6 layer, with a 3x3 kernel and stride 1, resulting in 3 4x4 layers.

The pooling layers downsample the dimensions of the input data, reducing its resolution. Common pooling operations include max pooling and average pooling, which retain the most vital information while reducing computational complexity. As an example, pooling from an 8x8 layer using 2x2 samples would generate a 4x4 resulting layer, as demonstrated in Figure 7.

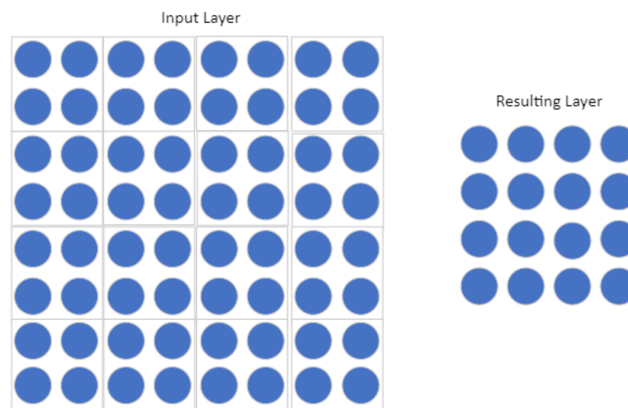


Figure 7 - A 2x2 pooling on an 8x8 layer results in a 4x4 layer.

After the sets of convolutional and pooling layers, a CNN typically have one or more fully connected layers that combine the learned features for final decision-making. These layers are typically present in the later stages of the network. That can be coupled with a softmax layer to evaluate the probability of each output class in a classification problem.

Over the years many CNN models have been proposed [27] [28], and for the purpose of this work two well-known models are used. AlexNet, which won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 [29] and MobilenetV2 [30], developed in 2018 as an evolution of MobileNet, designed for mobile and edge devices.

2.4.1.1. *AlexNet*

AlexNet CNN became famous after winning the ILSVRC in 2012 [29]. It is composed of five convolutional layers, each one using a Rectified Linear Unit activation function, followed by a max pooling layer. It expects a square input (256x256) split into three components (RGB). After the convolutional layers, it uses three fully connected layers, where the first two use a dropout layer to reduce the overfitting. The last layer is fed to a 1000-way softmax layer to match the 1000 classes from the dataset. Figure 8 presents the general AlexNet diagram as originally proposed.

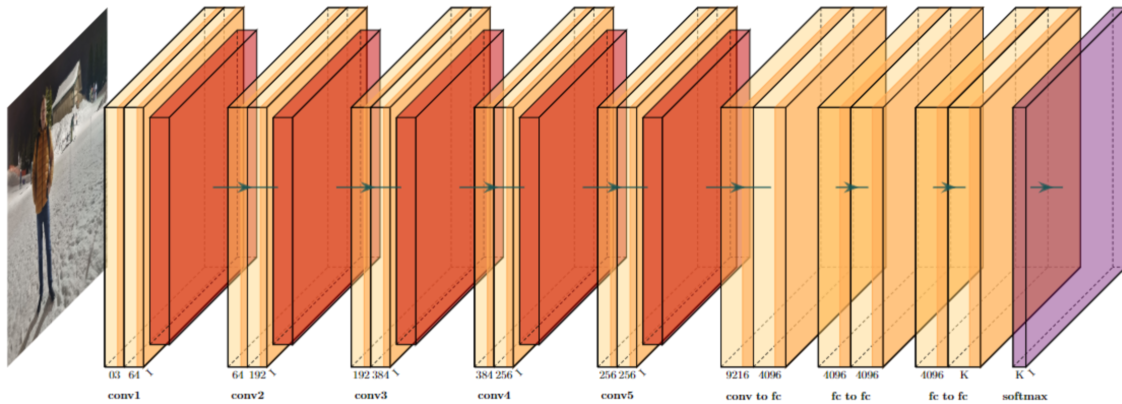


Figure 8 - AlexNet diagram.

To achieve the results reported, additional features were added to the process, such as: data augmentation to increase the number of samples in the training set; GPU parallelization to increase training speed and allow the model to fit the GPU memory available in 2012; and local response normalization in the two first layers to increase the model generalization.

2.4.1.2. *MobileNetV2*

MobileNetV2 [30] is a CNN architecture particularly designed for deployment on resource-constrained devices. Introduced by Google researchers in 2018, it uses a combination of inverted residuals, linear bottlenecks, and depth-wise separable convolutions to optimize computational efficiency. The inverted residuals consist of a lightweight depth-wise separable convolution followed by a linear bottleneck layer,

facilitating efficient information flow. Shortcut connections aids in mitigating the vanishing gradient problem during training.

At the end of the network, MobileNetV2 uses a global average pooling instead of traditional fully connected layer, reducing the number of parameters and making it more adequate for deployment on mobile and edge devices. The presented model is composed of one convolutional layer at the beginning, followed by a sequence of bottleneck structures, each composed of multiple layers. After this sequence, a convolutional layer with 1280 filters is applied and captured using an adaptative average pool layer, with a new convolutional layer to the number of classes. Table 1 show the structure of a MobileNetV2 CNN, where *t* is the expansion factor, *c* the number of output channels, *n* the number repetitions of that line, and *s* the stride of the initial layer.

Table 1 - MobileNetV2 architecture [30]

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

2.5. Early Exits

Over the past years, the accuracy of proposed DNN-based intrusion detection techniques has consistently improved [4]. To pave the way for more accurate detection, researchers usually escalate the number of DNN parameters. Consequently, implemented solutions frequently demand impractical inference computational costs, hindering their deployment on resource-constrained devices. Early exits are designed to tackle this challenge by introducing side branches that enable the premature termination of network inference [7]. Each side branch typically consists of fully connected layers that classify

the input at the current layer. If an input sample has high confidence at a branch, it can exit from that branch without traversing the entire network, effectively reducing the depth of the network that a portion of the samples need to traverse.

To conduct the DNN training procedure with early exits, researchers often resort to a joint training rationale [3]. Let *N* be the number of exit branches, and $y^{\sim i}$ be the classification output of each branch i on a given event with label *y*. We can compute the joint loss function as a weighted sum of losses of each branch through the following equation:

Eq. 6 - Joint loss function

$$\mathcal{L}_{joint}\left(y^{\sim i}, y^{i}\right) = \sum_{i=1}^{N} w_i \, \mathcal{L}\left(y^{\sim i}, y^{i}\right)$$

where $\mathcal{L}_{joint}$ is the joint loss function, $\mathcal{L}$ the loss function, and w_i the branch i weight. Here, w_i can be used to fine-tune the accuracies at each branch, such as allowing for a preference towards more accurate earlier branches, resulting in a lower overall computational cost.

At the test phase, the network inference task traverses the input event until it reaches the first branch, prematurely stopping the inference based on whether the classification threshold surpasses a given confidence threshold *t*. Typically, the acceptance threshold is established based on the operator's judgment, considering the desired tradeoff between accuracy and average inference time. If the final branch is reached, the event's class is determined based on the decision made at the final branch.

2.6. Classification with a Reject Option

In classification tasks, a rejection refers to the ability of a classifier to abstain from deciding on a particular sample or instance. Instead of assigning the sample to one of the predefined classes, the classifier may choose to reject it. This is particularly useful when the classifier is uncertain about the correct classification or when the pattern falls into a region where classification confidence is low. One common approach is to set a

confidence threshold, and if the classifier's confidence in its decision is below that threshold, it rejects the pattern. Patterns that are rejected may then be subjected to further analysis or handled by more sophisticated procedures, such as manual inspection or alternative classifiers [31].

While setting a single confident threshold is a common approach, Fumera et al. [31] explores the use of Class-Related Thresholds (CRT) rule as an alternative. This rule suggests using multiple reject thresholds for different data classes to achieve an optimal error-reject trade-off, even when posteriori probabilities are affected by errors. The CRT rule determines whether a pattern should be rejected or accepted based on class-specific thresholds.

2.7. Edge-Computing for NIDS

Edge computing (EC) aims to enable resource-constrained devices, such as those from IoT, to offload their tasks to edge-cloud infrastructure, paving the way for more processing resources [12]. In this context, EC should address the energy management challenges created by task offloading, such as additional network usage, task dispatch, and adequate management [32]. Offloading security-related tasks can greatly benefit from EC, given the limited processing capabilities of targeted devices that attackers can explore to circumvent detection [33]. A typical security application involves network traffic analysis for intrusion detection purposes. However, forwarding the entire network traffic for offloading the analysis task is not readily applicable in EC settings, as it can easily exhaust the device’s network bandwidth. Therefore, approaches that can effectively determine when analyzed network traffic require additional processing capabilities are essential.

Splitting NIDS-related tasks from resource-constrained devices to edge-cloud infrastructures is still a challenge in the literature [34]. Finding the optimal compromise between accuracy, energy efficiency, and classification latency poses a significant challenge to designed schemes. This is because offloading a neural network branch from edge to cloud also requires sending the features extracted by the prior branch. As a result,

the benefits of energy efficiency and accuracy can often only be achieved with significant tradeoffs in network communication.

2.8. Conclusions

This chapter briefly presented the background used for this work, describing what are IDSs and their usual types, the goal of multi-objective optimization and how it is implemented in this work. Foundations on ML and typical Neural Networks evaluated and the concepts of Early Exits and classification with a rejector. Lately, the purpose of Edge Computing in NIDS is discussed, highlighting the tradeoffs between processing resources and networking overhead.

Chapter 3
Related Work

This section aims to present available works on the literature related to different implementation of NIDS, on more traditional ML techniques and deep learning, using neural networks. It will also investigate works related to resource optimization focused on IoT devices and discuss some early exits implementations. Finally, some works related to cloud offload will be evaluated.

3.1. Machine Learning for NIDS

While deep learning techniques are many times preferred over traditional machine learning ones [35], since they can reach higher accuracy rates, they usually come at the cost of higher resource requirements. This leaves room for works that use traditional ML techniques. Verma et at. [36] present a comprehensive study on Machine Learning Based IDSs for IoT networks, exploring the efficacy of various classifiers in detecting anomalies and attacks in IoT traffic. Leveraging datasets like CIDDS-001, UNSW-NB15, and NSL-KDD, they evaluate classifiers such as Random Forest (RF), AdaBoost (AB), Gradient Boosting Machine (GBM), XGBoost (XGB), and Extra Trees Classifier (ETC). Utilizing hold-out and 10-fold cross-validation, they analyze performance metrics like accuracy, specificity, sensitivity, false positive rate (FPR), and area under the curve (AUC). Their findings suggest RF outperforms others in accuracy and specificity, while XGB excels in AUC. They conduct statistical tests to validate performance differences among classifiers, revealing significant variations across datasets and evaluation methods. Additionally, they assess model building time (MBT) to consider resource constraints, highlighting the importance of classifier response time, particularly in IoT applications, where computational resources are limited. Although resource consumption is mentioned, no studies have been made to reduce it. Also, dynamic behavior change is not addressed in the work.

Shukla [37] introduces ML-IDS, a machine learning-based IDS designed to detect wormhole attacks in Internet of Things (IoT) networks. The proposed ML-IDS consists of three approaches: K-means clustering based IDS (KM-IDS), decision tree-based IDS (DT-IDS), and a hybrid approach combining both. KM-IDS divides the network into safe zones and detects attacks when nodes from different zones attempt to connect. DT-IDS trains on known network topologies to set a threshold for permissible node distances and detects attacks based on this threshold. Hybrid-IDS combines both approaches to improve accuracy by filtering false positives. Experimental results demonstrate high detection for varying network sizes and topologies, with the hybrid approach achieving the best balance between detection rate and false positives. The work does not address resource consumption or the dynamic nature of network traffic.

Manihiro et al. [38] discuss the security challenges associated with the increased number of IoT devices, particularly in terms of network attacks such as denial-of-service (DoS), man-in-the-middle (MITM), and scanning attacks. To address these challenges, the paper proposes a machine learning-based anomaly detection approach that leverages a hybrid feature selection engine to identify the most relevant features for attack detection, coupled with the Random Forest algorithm for classifying traffic as normal or anomalous. The method is evaluated using the IoTID20 dataset, showing high accuracy in detecting the three attacks evaluated. This approach is highlighted for its effectiveness in enhancing IoT network security by accurately identifying and mitigating potential threats with a low false-positive rate. There is no mention to the resource consumption concerns or evaluation of dynamic behavior.

3.2. Deep Learning for NIDS

Network-based intrusion detection through DNN-based approaches has been a popular research topic in the literature over the past years [2]. In general, the proposed schemes often prioritize improved detection accuracies while overlooking their practical applicability. Li Ma et al. [4] discusses the challenges faced by current IoT intrusion detection methods and proposes a novel CNN-based model to address these challenges, particularly focusing on DDoS attacks in the IoT scenario. The proposed model integrates

a multilayer convolution feature fusion mechanism and a novel loss function based on symmetric logarithmic loss. Extensive experiments conducted on the NSLKDD dataset demonstrate that the proposed approach outperforms traditional methods, achieving higher accuracy with lower false alarm rates. Future work will involve further optimizing the network structure, extending the model for multivariate classification tasks, and investigating detection systems for other typical IoT application attacks. Although their approach enhances classification accuracy on an outdated dataset, it tends to overlook the associated inference costs and classification reliability issues.

Another accuracy-focused approach was proposed by J. Zhang et al. [39], which introduces a novel approach to network intrusion detection using Bayesian Convolutional Neural Networks (BCNN) and an ensemble-based detection scheme to enhance performance. Evaluation on NSL-KDD and UNSW-NB15 datasets demonstrates that the proposed BCNN-IDS outperforms conventional CNN-IDS and SVM-IDS, achieving higher detection rates and lower false alarm rates. This model also implements a rejection feature, named *drop-rate*, and by adjusting a threshold parameter, the BCNN-IDS achieves optimal trade-offs between detection rate, false alarm rate, and drop rate. Albeit the accuracy benefits, the utilization of multiple DNNs hampers their deployment on resource-constrained devices.

M. Ge et al. [5] explored the application of deep learning techniques, specifically customized feed-forward neural networks (FNN), to classify network traffic as either normal or malicious. Using the BoT-IoT dataset, they compared the results of the FNN models with a Support Vector Classifier (SVC), for multiclass and binary classification, showing the superiority of the FNN models in terms of both accuracy and runtime efficiency, further solidifying the applicability of deep learning techniques in network security. Their proposal surpassed traditional shallow based methods but has not applied the model to real IoT devices.

J. Zhang et al. [40] discuss an innovative autonomous model update scheme for Deep Learning-based network traffic classifiers (NTCs). The scheme addresses the challenges of updating classifiers with new applications without a pre-existing dataset. It involves a discriminator comprising a statistical filter and a convolutional neural network-

based binary classifier to autonomously filter out and build a dataset for the new application from active network traffic. The scheme utilizes transfer learning to update the existing classifier with this newly created dataset, ensuring the classifier remains effective in identifying both existing and new applications. Evaluation on the ISCX VPN-nonVPN dataset demonstrated the scheme's success in filtering new application packets, building a relevant training dataset, and updating the classifier efficiently through transfer learning. The strategy maintains high classification accuracy and adapts to new network applications, considering the dynamic behavior of network traffic, but does not evaluate the resource consumption of the approach.

3.3. Resource Optimization for NIDS on IoT

In recent years, recognizing the impracticality of applying these methods on resource-constrained devices, some research has shifted focus from accuracy to improving inference computational costs [3]. Tekin et al. [41] investigate various deployment approaches for ML-based intrusion detection systems (IDS) in Smart Home Systems (SHSs). They explore cloud, edge, and IoT device-based deployment strategies, as well as conventional and TinyML approaches for ML inference. Through experiments using datasets of IoT network traffic captured in a Distributed Smart Space Orchestration System (DS2OS) environment, they evaluate the performance and energy consumption of popular ML algorithms including Logistic Regression (LR), k-Nearest Neighbors (k-NN), Decision Trees (DT), Random Forests (RF), Naive Bayes (NB), and ANNs. Results show that while Naive Bayes requires less training time, it sacrifices some accuracy, whereas k-NN incurs high energy consumption during inference, making it less suitable for resource-constrained IoT devices. Decision Trees exhibits the best performance in terms of inference time on IoT end devices, while Random Forests demonstrate good accuracy and energy efficiency during training. The study highlights the importance of considering both accuracy and energy consumption when deploying ML-based IDS in SHSs, providing valuable insights for real-world implementation, and identifying areas for further research. While it does not consider CNNs in the comparison table, the paper

is relevant to demonstrate the concerns about energy consumption by NIDSs in IoT devices.

Hoang et al. [42] explore the significance and challenges of implementing IDSs in IoT environments, given the exponential growth of IoT applications and the increasing sophistication of cyberattacks. With the aim of addressing these challenges, the authors propose a lightweight DNN-based IDS model designed for deployment on IoT gateway devices. Leveraging techniques such as Principal Component Analysis (PCA) for dimensionality reduction, the proposed model demonstrates superior performance compared to existing CNN-based models, achieving high accuracy while minimizing computational resources and training time. Experimental results, conducted on the IoT23 dataset and its reduced subset, validate the effectiveness of the proposed model in detecting various types of attacks with precision and efficiency suitable for edge computing environments. The proposal, on the other hand, relies on a lightweight model and PCA for resources consumption reduction and doesn't consider the dynamic behavior of network traffic.

Wang et al. [43] proposes the application of MobileNetV2, a lightweight DNN implementation, for intrusion detection. The authors compare their results with many other models, showing that the proposed solution has not only improved metrics such as accuracy and recall, but also achieved these results in less time. Tests were conducted using the Car-Hacking and CICIDS2017 datasets and hyper parameter optimization was used to set the (Transfer Learn) TL base parameters. There are no mentions from the authors to the evolving behavior of network traffic.

3.4. Multi objective for NIDS

In literature there are different uses of multi-objective optimization tasks. Sharma et al. [44] summarizes some common objectives which IDS for IoT devices try to optimize: Classification accuracy, Detection rate/recall, Precision, False alarm/error rate, Specificity, Number of features, Response time, Memory usage and Category Detection, as well as the algorithms commonly used for this task.

Roopak et al. [45] introduces an IDS aimed at combating DDoS attacks within IoT networks by leveraging a hybrid approach that combines deep learning techniques with multi-objective optimization methods. The proposed IDS utilizes a fusion of Jumping Gene adapted NSGA-II for efficient data dimensionality reduction and a combination of CNN and Long Short-Term Memory (LSTM) networks for accurate attack classification. This hybrid model was tested on the CICIDS2017 dataset using High-Performance Computing (HPC) resources, achieving a high rate and a significant reduction in training time. There are no discussions on the resource consumption of the model, only on the increased accuracy and no mention to the dynamic behavior of the network traffic.

Asgharzadeh et al [46] introduces a hybrid method combining a convolutional neural network (IoTFECNN) designed for feature extraction with a binary multi-objective enhanced Capuchin Search Algorithm (BMECapSA) for feature selection. This combination, referred to as CNN-BMECapSA-RF, is tested on two datasets, NSL-KDD and TON-IoT, showing superior classification results compared to existing methods. No mention to resources consumption optimization was made on this work.

3.5. Early Exits

While the application of early exits has been extensively researched in various domains [8], its adoption in intrusion detection is still in its early stages. One of the first papers to address the Early Exit concept was presented by Teerapittayanon et al. [10], where they discuss the challenges posed by the increasing depth of neural networks, leading to higher latency and energy consumption during inference despite improvements in classification accuracy. To address this issue, the authors propose BranchyNet, a neural network architecture that incorporates early exit branches to allow certain test samples to exit the network early, leveraging the observation that many samples can be accurately classified at earlier stages. BranchyNet is trained using a joint optimization approach and utilizes exit criteria based on the entropy of classification results. Experimental results on various convolutional neural network architectures, including LeNet, AlexNet, and ResNet, demonstrate that BranchyNet significantly reduces inference time and energy

consumption while maintaining high accuracy, providing 2x-6x speedup on both CPU and GPU. Additionally, the authors discuss hyperparameter sensitivity, tuning of entropy thresholds, effects of branch structure, and cache optimization strategies to further improve the efficiency of BranchyNet.

En Li et al. [6] proposes a framework devised to facilitate low-latency edge intelligence by orchestrating collaborative DNN co-inference between mobile devices and edge servers, called Edgent, which incorporates two specialized configurators tailored for static and dynamic network environments, aiming to optimize DNN inference latency under varying conditions. In the static environment, Edgent employs regression models to predict layer-wise inference times and trains branchy DNN models to enable early-exit mechanisms, ensuring efficient utilization of computational resources. The model's efficacy was evaluated using Raspberry Pi devices and desktop PCs, with experiments conducted using synthetic and real-world bandwidth datasets. Results demonstrate its ability to balance between inference accuracy and latency across diverse network scenarios, leveraging computational resources effectively at the network edge. Their approach notably enhances inference time, yet it tends to neglect the influence of network traffic behavior changes on the reliability of their solution.

A similar approach was proposed by W. Seifeddine et al. [7], which discusses the challenges posed by the increasing size of deep learning models, leading to computational inefficiencies in both training and inference phases. To address this, the authors propose Dynamic Hierarchical Neural Network Offloading (DHN^{2}O), a method aimed at improving model execution in embedded IoT systems by leveraging edge or cloud servers. DHN^{2}O combines early exit strategies, where computation can halt at intermediate layers, with offloading decisions to more capable devices, thus optimizing resource utilization. Using reinforcement learning, specifically Deep Q-Networks (DQN), the method dynamically selects actions (stay, exit, or offload) at each potential exit point based on factors like energy consumption and accuracy. Experimental results demonstrate the effectiveness of DHN^{2}O in achieving efficient inference while respecting computational constraints, with potential applications in various real-world scenarios. Although the authors can decrease inference computational costs, they overlook how the non-stationary behavior of network traffic can affect their scheme.

3.6. Edge Offloading

Splitting the network using Early-Exits opens up the floor to offloading part of the inference work to other devices, such as edge computing servers located near the IoT devices, or even farther away, to cloud computing infrastructures, such as AWS or IBM cloud. Some papers discuss strategies behind the offloading strategy, and they are discussed in this section. Colocrese et al. [47] proposes a model-distributed inference with early-exit (MDI-Exit) framework for image classification in edge networks. MDI-Exit splits the deep neural network (DNN) model into many parts, where each part can be processed in a different worker, in a sequential process. The framework also allows for early exiting from the DNN model if the target accuracy is reached. MDI-Exit adaptively determines early-exit and offloading policies as well as data admission at the source, based on the data arrival rate (images / sec). They also have a multi-objective optimization problem, related to data rate vs. accuracy, but didn't evaluate this using any known technique, deciding only to present comparison results with fixed data rate or fixed accuracy.

Angelucci et al. [48] combines early exiting (EE) with edge computing to trade-off accuracy with inference latency. The article focuses on the support of applications for connected and automated driving. The authors model the complex interactions between EE and edge computing using a Markov Decision Process (MDP). They then formulate an optimization problem to select the inference strategy that maximizes the average task accuracy. The optimal policy can be derived by mapping the MDP into a linear program. Unlike our proposed problem, the authors use a different strategy. The end device uses a different, simpler DNN, and EE is used in the more complex Edge DNN. Offloading might not occur depending on the current network quality.

Bajpai et al. [49] present an innovative unified approach that merges early exits and split computing. The authors determine the 'splitting layer,' the optimal depth in the DNN for edge device computations, and whether to infer on the edge device or offload to the cloud for inference considering accuracy, computational efficiency, and communication costs. They introduce I-SplitEE, an online unsupervised algorithm that adapts to diverse environmental distortions. Unfortunately, they didn't evaluate any real

data information with their model, but, instead, they simulated the performance tradeoffs by using different offloading costs.

3.7. Related work discussion

A summary of the articles presented in this section is displayed in Table 2, where, for each paper, it is specified which ML technique was used, the application targeted by the model, the datasets used, if an IoT scenario or device was used, if a resource optimization technique was employed and if the data behavior change in time was taken in consideration, which is a specific characteristic of network traffic.

Table 2 - Summary of related works

Paper	Application	ML Technique	Dataset	IoT Target	Resource Optimization Technique	Behavior Change Consideration
Verma et at. [36]	IDS	RF AB GBM XGB ETC	CIDDS-001 UNSW-NB15 NSL-KDD	No	None	No
Shukla [37]	Wormhole Detection	KM DT	Not mentioned	Not mentioned	N/A	No
Manihiro et al. [38]	DoS, MITM and Scanning Detection	RF	IoTID20	No	N/A	No
Li Ma et al. 2020 [4]	DDoS detection	CNN	NSL-KDD	IoT Gateway	N/A	No
Zhang et al. 2020 [39]	IDS	CNN	NSL-KDD; UNSW-NB15	No	N/A	No
M. Ge et al. [5]	IDS	FNN	BoT–IoT	No	Just Comparison	No

J. Zhang et al. [40]	NTC	CNN	ISCX VPN-nonVPN	No	N/A	Yes
Tekin et al. [41]	IDS	LR k-NN DT RF NB ANN	DS2OS	Raspberry Pi 4 Azure IoT Kit	Just Comparison	No
Hoang et al. [42]	IDS	CNN	IoT23	Edge Gateway	Lightweight Model / PCA	No
Y. Wang et al. [43]	IDS	CNN	Car-Hacking CICIDS2017	ICVs	Optimized CNN (MobileNetV2)	No
Roopak et al. [45]	DDoS Detection	CNN LSTM	CICIDS2017	N/A	N/A	No
Asgharzadeh et al [46]	IDS	CNN	NSL-KDD TON-IoT	N/A	N/A	No
Teerapittayanon et al. [10]	Image Recognition	CNN	MNIST, CIFAR10	N/A	Early Exits	No
En Li et al. [6]	Image Recognition	CNN	CIFAR10	RPi 3	Early Exits + Edge Offload	No
W. Seifeddine et al. [7]	Image Recognition	CNN	CIFAR10	N/A	Early Exits + Edge Offload	No
Colocrese et al. [47]	Image Recognition	DNN	CIFAR10	Jetson NANO	Early Exits + Edge Offload	No
Angelucci et al. [48]	Image Recognition	DNN	CIFAR10	N/A	DNN + Edge Offload	No
Bajpai et al. [49]	Image Recognition	DNN	Caltech-256 CIFAR10	N/A	DNN + Edge Offload	No

3.8. Conclusions

This section reviewed a collection of published papers to determine the methodology for the current work.

The initial goal was to demonstrate that Machine Learning and Deep Learning are prevalent in behavior-based NIDS implementations. The focus of most research is on

enhancing system performance in terms of accuracy, false negatives, and false positives, often at the expense of overlooking computational resource requirements.

The review then turned to studies where resource consumption was emphasized, particularly within the NIDS domain. Some studies introduced more efficient solutions, while others merely compared outcomes. Further analysis involved papers on multi-objective optimization to grasp the employed techniques and their objectives within this framework. Additionally, the examination included papers on Early Exits, though their primary focus was on Image Recognition.

Despite some studies mentioning the IoT scenario as a basis for their research, just a few of them actually utilized IoT devices in their experimentation. Remarkably, only a single paper addressed the Dynamic Behavior of input, a key aspect of Network Traffic.

Table 2 organizes the articles by:

- Paper: The author(s) and reference.
- Application: The intended application of the proposed solution.
- ML Technique: The machine learning techniques assessed or used.
- Dataset: The datasets applied in model evaluation.
- IoT Target: The usage of actual IoT devices in results evaluation.
- Resource Optimization Technique: The application of methods to minimize model resource requirements.
- Behavior Change Consideration: The consideration of input's dynamic behavior, typical of network traffic.

This work proposes a synthesis of these varied concepts by applying Deep Learning for NIDS, incorporating Early Exits to lessen resource consumption, and integrating a rejector mechanism. The strategic adjustment of early exit and rejector thresholds is aimed at finding an optimal operational balance, thereby enhancing the model's longevity in the face of dynamic traffic behavior, granting its accuracy while reducing the resource requirements.

Table 3 - Selected features of chosen dataset [50].

Field	Description	Field	Description
minimumInterArrivalTime	Minimum packet inter-arrival time for all packets of the flow (considering both directions).	quartileFirstDataWire_a_b	First quartile of bytes in (Ethernet) packet
quartileFirstInterArrivalTime	First quartile inter-arrival time	medianDataWire_a_b	Median of bytes in (Ethernet) packet
medianInterArrivalTime	Median inter-arrival time	avgDataWire_a_b	Mean of bytes in (Ethernet) packet
avgInterArrivalTime	Mean inter-arrival time	quartileThirdDataWire_a_b	Third quartile of bytes in (Ethernet) packet
quartileThirdInterArrivalTime	Third quartile packet inter-arrival time	maximumDataWire_a_b	Maximum of bytes in (Ethernet) packet
maximumInterArrivalTime	Maximum packet inter-arrival time	varianceDataWire_a_b	Variance of bytes in (Ethernet) packet
varianceInterArrivalTime	Variance in packet inter-arrival time	minimumDataWire_b_a	Minimum number of bytes in (Ethernet) packet (server→client)
minimumInterArrivalTime_a_b	Minimum of packet inter-arrival time (client→server)	quartileFirstDataWire_b_a	First quartile of bytes in (Ethernet) packet
quartileFirstInterArrivalTime_a_b	First quartile of packet inter-arrival time	medianDataWire_b_a	Median of bytes in (Ethernet) packet
medianInterArrivalTime_a_b	Median of packet inter-arrival time	avgDataWire_b_a	Mean of bytes in (Ethernet) packet
avgInterArrivalTime_a_b	Mean of packet inter-arrival time	quartileThirdDataWire_b_a	Third quartile of bytes in (Ethernet) packet
quartileThirdInterArrivalTime_a_b	Third quartile of packet inter-arrival time	maximumDataWire_b_a	Maximum of bytes in (Ethernet) packet
maximumInterArrivalTime_a_b	Maximum of packet inter-arrival time	varianceDataWire_b_a	Variance of bytes in (Ethernet) packet
varianceInterArrivalTime_a_b	Variance of packet inter-arrival time	total_packets_a_b	The total number of packets seen (client→server).
minimumInterArrivalTime_b_a	Minimum of packet inter-arrival time (server→client)	total_packets_b_a	” (server→client)
quartileFirstInterArrivalTime_b_a	First quartile of packet inter-arrival time	ack_pkts_sent_a_b	The total number of ack packets seen (TCP segments seen with the ACK bit set) (client→server).
medianInterArrivalTime_b_a	Median of packet inter-arrival time	ack_pkts_sent_b_a	” (server→client)
avgInterArrivalTime_b_a	Mean of packet inter-arrival time	pure_acks_sent_a_b	The total number of ack packets seen that were not piggy-backed with data (just the TCP header and no TCP data payload) and did not have any of the SYN/FIN/RST flags set (client→server)
quartileThirdInterArrivalTime_b_a	Third quartile of packet inter-arrival time	pure_acks_sent_b_a	” (server→client)
maximumInterArrivalTime_b_a	Maximum of packet inter-arrival time	pushed_pkts_sent_a_b	The count of all the packets seen with the PUSH bit set in the TCP header. (client→server)
varianceInterArrivalTime_b_a	Variance of packet inter-arrival time	pushed_pkts_sent_b_a	” (server→client)

Field	Description	Field	Description
minimumDataWire	Minimum of bytes in (Ethernet) packet, using the size of the packet on the wire.	syn_pkts_sent_a_b	The count of all the packets seen with the SYN bits set in the TCP header respectively (client→server)
quartileFirstDataWire	First quartile of bytes in (Ethernet) packet	syn_pkts_sent_b_a	The count of all the packets seen with the SYN bits set in the TCP header respectively (server→client)
medianDataWire	Median of bytes in (Ethernet) packet	fin_pkts_sent_a_b	The count of all the packets seen with the FIN bits set in the TCP header respectively (client→server)
avgDataWire	Mean of bytes in (Ethernet) packet	fin_pkts_sent_b_a	The count of all the packets seen with the FIN bits set in the TCP header respectively (server→client)
quartileThirdDataWire	Third quartile of bytes in (Ethernet) packet	urgent_pkts_sent_a_b	The total number of packets with the URG bit turned on in the TCP header. (client→server)
maximumDataWire	Maximum of bytes in (Ethernet) packet	urgent_pkts_sent_b_a	” (server→client)
varianceDataWire	Variance of bytes in (Ethernet) packet	throughput_a_b	The average throughput calculated as the unique bytes sent divided by the elapsed time i.e., the value reported in the unique bytes sent field divided by the elapsed time (the time difference between the capture of the first and last packets in the direction). (client→server)
minimumDataWire_a_b	Minimum number of bytes in (Ethernet) packet (client→server)	throughput_b_a	” (server→client)

Chapter 4
Problem Statement

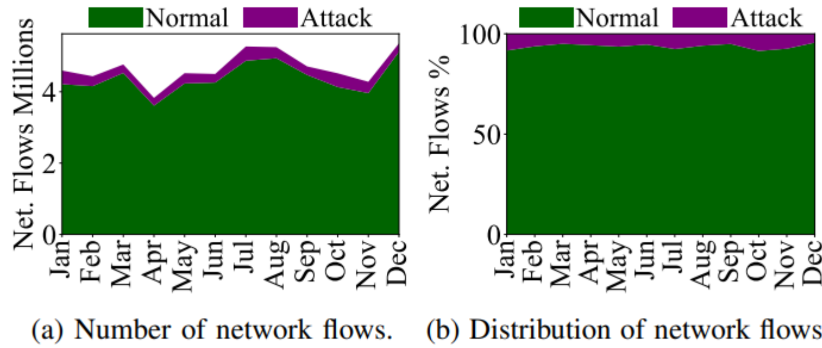
In this section a deeper analysis of the performance of DNN techniques in relation to accuracy and processing requirements when confronted with changes in network traffic behavior. More specifically, initially the dataset utilized in the experiments conducted in this work is introduced, followed by an assessment of the performance of DNN techniques on this dataset.

4.1. MAWIFlow

Current datasets in the literature often assume a static behavior of network traffic [51]. Consequently, designed schemes can typically achieve high detection accuracies during the testing phase but may perform poorly when deployed in production.

To ensure a realistic evaluation, MAWIFlow dataset is used, a publicly available intrusion dataset containing real, valid, and labeled network traffic from production environments spanning an extended period. To achieve these characteristics, the dataset is built upon MAWI [52] working group traffic archive. It includes network traffic from MAWI samplepoint-F, a transit link between Japan and the USA, collected in 15-second intervals daily. Network data is collected daily, resulting in a network PCAP file for each day over the evaluation period, totaling over 7TB of data and encompassing more than 70 billion network flows. For this research, the network data collected throughout the entire year 2016 was employed. This collected data is organized into network flows based on the hosts and services involved in each communication. Each network flow represents a 15-second segment of client/service and server/service data, which is subsequently summarized into an associated feature set. For the purpose of this work, 58 features were extracted from Moore [50] work and are listed in Table 3. To assign labels, this study employs the MAWILab [9] unsupervised machine learning

algorithms, which identify network anomalies subsequently labeled as attacks in the dataset. Figure 9 show the number of flows and the class distribution of the dataset over the year.



(a) Number of network flows. (b) Distribution of network flows.

Figure 9 - *MAWIFlow* network flow distribution over the year.

Table 4 - MAWIFlow dataset statistics.

Property	**Value**
Average Daily Network Packets	105 Millions
Average Daily Network Flows	9 Millions
Average Daily Throughput	610 Mbps
Average Daily Anomalous Flows	1.8 Millions
Average Daily Dataset Size	19.7 GB
Total Network Packets	27.72 Billions
Total Network Flows	6.14 Billions
Total Dataset Size	7.1 TB

4.2. Traditional DNN performance

The evaluation aims to answer two Research Questions (RQs):

- ***(RQ1)*** What is the intrusion detection performance of widely used DNN techniques?
- ***(RQ2)*** What are the computational costs of evaluated techniques?

The performance of two widely used CNN architectures is evaluated, namely AlexNet and MobileNetV2. CNNs are fundamentally designed to process image data, which naturally comes in multi-dimensional arrays (e.g., a typical color image has three layers corresponding to RGB values). However, datasets in fields such as signal

processing, time series analysis, or even certain types of image data might not fit this multi-layered format. This is the case for the Moore's work [50] in the MAWI dataset used, which provides a one-dimensional dataset with 58 features.

The first step in adapting the one-dimensional dataset for CNN usage involves reshaping the data from a 58-feature vector into a two-dimensional 8x8 format, essentially converting a flat array into a matrix. This transformation is akin to preparing a grayscale image from raw pixel values. The choice of an 8x8 matrix requires the addition of six dummy features (filled with zeros) to match the needed dimensions, ensuring that the dataset aligns with the input structure of the CNNs without introducing any artificial bias or significant information. This preprocessing is done during the dataset reading phase, illustrating how data preprocessing can significantly impact the compatibility of data with complex models.

For the second adaptation, considering both AlexNet and MobileNetV2 are inherently designed to process three-channel (RGB) images, the models were modified to accept a single channel (8x8x1) input. This modification is crucial because it allows the networks to process the adapted dataset without requiring the standard three-dimensional input. Additionally, MobileNetV2 employs a 2D adaptive average pooling layer to downscale the input to a 48x48x1 format. As the dataset is smaller than that, this layer has the opposite effect, effectively enlarging the data while maintaining its essential features. This technique was mirrored in the AlexNet model, by adding this adaptation layer, expanding the input size to feed into its layers. These adaptations not only demonstrate the flexibility of CNN architectures but also highlight the creativity required to apply deep learning techniques across various data types beyond their original design intentions.

The DNNs were trained using Adam optimizer, running for 1,000 training epochs. Categorical cross-entropy was used as the loss function. The learning rate was 0.001 with a learning rate scheduler that stops training if there is no improvement in the validation accuracy over 50 epochs. These models were implemented using PyTorch API version 2.1.0. The classifiers were evaluated in terms of True Positive and True Negative rates.

The TP refers to the ratio of intrusion events that are correctly classified as intrusions, while TN denotes the ratio of normal events that are correctly classified.

The initial experiment is designed to address *RQ1* and assess the classification performance of the chosen intrusion detection techniques on the MAWIFlow dataset when it encounters changes in network traffic behavior over time. To achieve this goal, the selected DNN architectures was trained using the MAWIFlow data from January. The model's performance was evaluated over the remaining year without periodic model updates. Figure 10 displays the monthly accuracies of the chosen DNN architectures. There is a significant decline in classification accuracy over time. As an example, AlexNet, as shown in Figure 10a, experiences a decrease in its TP rate of up to 25% when compared to the training period (Jan. vs. Nov.). Evaluated intrusion detection techniques struggle to address the evolving network traffic patterns over time.

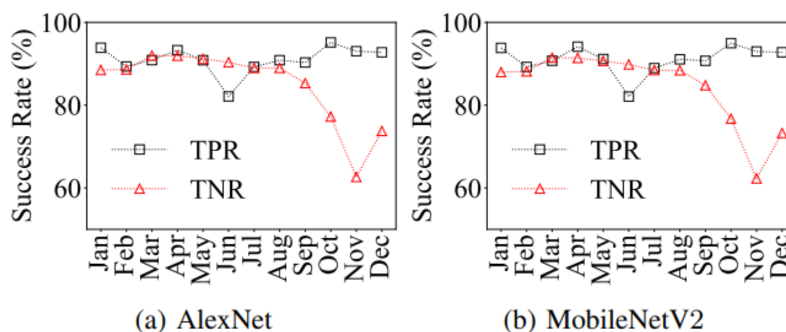


Figure 10 - Accuracy trend over a year of commonly used classifiers without periodic model updates. Classifier is trained in January and evaluated in subsequent months without updates.

The second experiment is designed to address *RQ2* and examine the computational costs associated with the selected techniques. The average inference computational costs were assessed in two environments, namely Desktop and Raspberry. The first is equipped with a 16-core Intel Xeon E5-2640 v3 CPU, 32 GBs of memory, and a Nvidia Tesla T4 GPU running on top of Ubuntu Linux 22.04. The latter is a Raspberry Pi 3 Model B, with a 4-core Broadcom CPU and 1 GB of memory running on top of Raspberry Pi OS with kernel version 6.1. Table 5 shows the average inference throughput of each evaluated DNN on the selected platforms. It is possible to note a significant decrease in the detection throughput on a resource-constrained device. In this scenario, an average throughput of ≈ 7.3 events per second proves insufficient for

handling the thousands of network events that a device may encounter when deployed on a real network. Hence, in addition to addressing changes in network traffic behavior over time, proposed schemes must also be capable of accomplishing this task while placing minimal demands on processing resources.

Table 5 - Average event detection throughput (events / sec).

DNN	System		
	Raspberry	Desktop	
		CPU	GPU
AlexNet	7.36	247.34	17,609
MobileNetV2	7.93	509.58	3,419

4.3. Conclusions

The preliminary outcomes of utilizing the AlexNet and MobileNetV2 DNNs on this dataset highlight both the potential and challenges of applying advanced machine learning models to the dynamic realm of network traffic analysis. These results validate the hypothesis that, with the correct preprocessing and model adaptation, DNNs can be used for datasets far from the original design specifications, such as one-dimensional feature vectors from network traffic. However, as anticipated, the inherent variability and dynamic nature of network traffic pose significant challenges to maintaining model accuracy over time.

Moreover, the computational demands of running such complex models on resource-constrained devices, exemplified by the Raspberry Pi, highlight a significant barrier to their practical deployment in production environments. The extended processing time not only diminishes the feasibility of using these DNNs in real-time applications but also limits their scalability and efficiency in environments where rapid data processing is paramount. This challenge accentuates the importance of optimizing model architectures and exploring innovative solutions to reduce computational requirements without sacrificing performance.

Chapter 5

Methodology

5.1. Proposed Model

To address the aforementioned challenges, this work proposes a new DNN-based NIDS through early exits that operate following an energy-efficient EC rationale. The workflow of our proposed scheme is illustrated in Figure 11, and is implemented in three main stages.

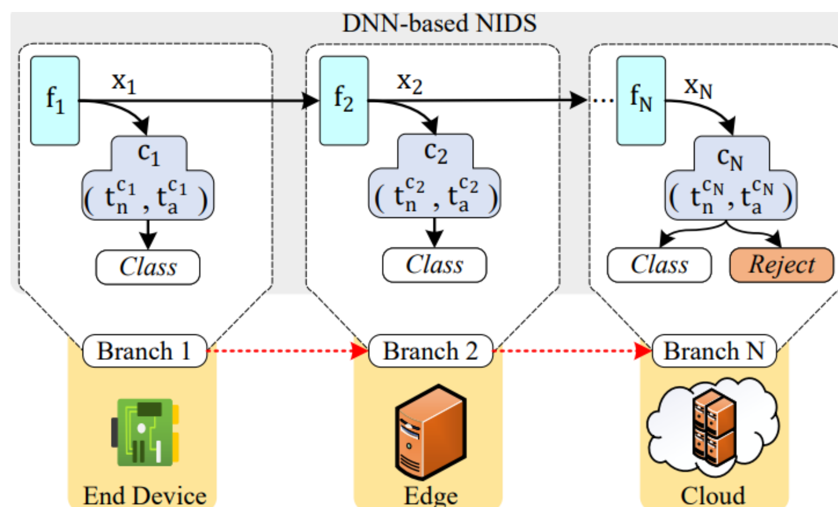


Figure 11 - Workflow of the proposed DNN-based NIDS through early exits for intrusion detection offloading in EC settings. DNN Branches are distributed from end device, edge, and cloud infrastructure.

First, intrusion detection is conducted using a DNN model with early exits. The insight is to leverage early exits to distribute the DNN-based inference NIDS task within an energy-efficient EC architecture. Consequently, the intrusion detection task can be split between the *End Device*, *Edge*, and *Cloud* infrastructure based on the classification difficulty of the evaluated event. This approach aims to offload computation only for a subset of events requiring additional processing capabilities, thereby simultaneously

decreasing the *End Device*'s energy consumption and reducing the *Cloud* infrastructure's resource usage. Second, to find the optimal tradeoff for each DNN branch (Figure 11, Branch 1 to N), a multi-objective optimization task is conducted. The aim is to adequately assess the accuracy and energy consumption tradeoffs when offloading intrusion detection tasks among the EC entities. Third, to improve the model's generalization capabilities for adequately addressing the non-stationary behavior of network traffic, the classification task is performed with model calibration and a reject option. On the one hand, model calibration ensures that the model's confidence values accurately reflect their reliability, enabling a reliable assessment of classification confidence. On the other hand, the reject option in the final model branch (Figure 11, Branch N) allows the model to reject potential misclassifications at the *Cloud* infrastructure. As a result, the proposed model can reduce the energy consumption of *End Devices* and the resource usage of the *Cloud* infrastructure, improve the model's generalization capabilities, and ultimately pave the way for an effective energy-efficient EC.

The next subsections further describe the implementation of the proposed model, including the modules that implement it.

5.2. Early Exits implementation

Current DNN-based NIDSs demand computational processing capabilities that are unfeasible for implementation on resource-constrained devices (see Table 5). To address this, edge offloading for DNNs can be performed based on the input's complexity by introducing early exits. This is because conducting the side branches on the same *End Device*, as often assumed in the literature, can still pose challenges related to energy consumption and processing requirements for events reaching the final branches. To address these challenges, the proposed model distributes the deployment of DNN branches between *End Device*, *Edge*, and *Cloud* infrastructure according to the current evaluated event, as illustrated in Figure 11.

Consider an intrusion detection task where $x \in \mathbb{R}^D$ denotes a D-dimensional feature vector input and $y \in \{n, a\}$, where n denotes a *normal* event, and a denotes *attack* labeled

events. The proposed model aims at learning a DNN that can model the probabilistic predictive distribution $p(y|x)$ over ground truth labels. Following a typical early exit framework, during the model learning process, N classifiers $c_{\{0,\ldots,N\}}$ are introduced for a given set of intermediate DNN layers $f_{\{0,\ldots,N\}}$. Namely, let c_i be the i-th classifier that receives the x_i feature vector output by the f_i DNN layer, the DNN will prematurely end inference if the c_i classification confidence level surpasses the classification threshold t^{c_i} for *normal*-classified ($t^{c}{}_{n^i}$), or *attack*-classified ($t^{c}{}_{a^i}$) events, as illustrated in Figure 11.

Here, the classification threshold t^{c_i}, denotes a tuple where $t^{c}{}_{n^i}$ establishes the acceptance threshold for *normal* events, and $t^{c}{}_{a^i}$ the acceptance threshold for *attack* events. As a result, relying on lower t^{c_i} for a given classifier c_i can increase the number of accepted events, whereas using higher t^{c_i} will lead to a higher acceptance rate on branch i. The acceptance thresholds should be defined based on the operator's needs, as they directly impact energy consumption and processing costs.

In contrast to traditional early exit strategies, to further increase the model's generalization, we incorporate a reject option at the last DNN branch to reliably handle the classification decision (Figure 11, *Branch N*). The model accepts or rejects the classification at the final branch based on its associated confidence value $\hat{p}$. To achieve such a goal, the module's implementation is coped with a rejection *rej* function, as determined by the following equation:

Eq. 7 - Rejector module

$$rej(\hat{p}, t^{C_N}) \begin{cases} \phi & if\ \hat{p} \leq t^{C_N} \\ \hat{p} & otherwise \end{cases}$$

where $\emptyset$ denotes events likely to be incorrect decisions the DNN model final branch performs, and t^{C_N} the acceptance thresholds at the final model branch. As a result, the *rej* module suppresses unreliable classification as measured by their associated classification confidence values at the final DNN branch (Figure 11, *Branch N*). Similarly, the rejection threshold should be determined based on the operator's judgment. A higher rejection threshold will enhance system reliability but result in a higher proportion of rejected events. Conversely, a lower threshold will accept more events but expose the system to unreliable classifications.

Therefore, by relying on the proposed early exit approach, the NIDS inference task can be split between the *End Device*, *Edge*, and *Cloud*. When offloading a given branch i, the network tradeoff only involves sending the output of the current DNN layer f_i, namely vector x_i, as opposed to the traditional approach where the entire network traffic must be transmitted. Thus, our proposed model enables the adequate NIDS inference splitting into an EC setting.

5.3. Multi-objective Optimization

Network-based intrusion detection for resource-constrained devices should be performed with minimal processing requirements while maintaining classification accuracy and reliability. To address such a challenge, the model building is conducted as a multi-objective optimization task, as illustrated in Figure 12.

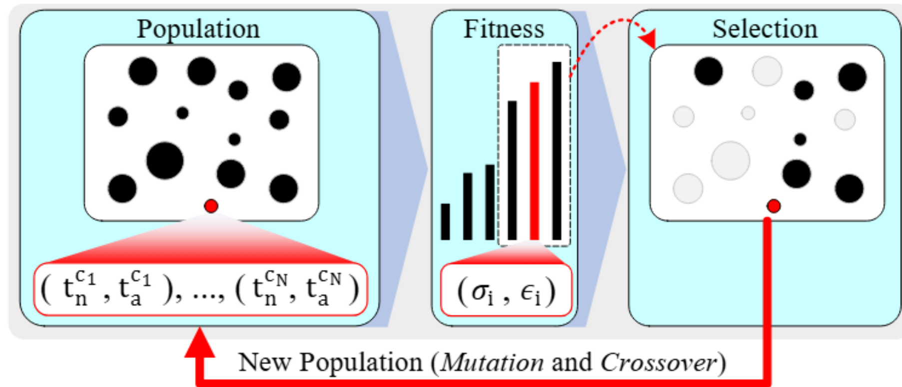


Figure 12 - multi-objective optimization for DNN-based NIDS with Early Exits. Here, σ_i measures the average DNN inference energy consumption, and ϵ_i measures the error rate obtained using the selected acceptance thresholds $t^c{}_i$.

Considering a DNN model h implemented with N branches, coupled with a Rejector module at the final branch (see Section 5.2), the goal of the multi-objective optimization is to find $N-1$ associated t_b branches thresholds, along with a t_{rej} Rejector threshold. Hence, two objectives are considered based on the system's processing costs and accuracy. It is assumed that the first objective is a direct result of accepting an additional number of events at earlier branches, whereas the second objective relates to increasing the rejection rate, which, in turn, raises processing costs due to more events

reaching the final branch. Therefore, the multi-objective can be conducted by solving the following equation.

Eq. 8 - multi-objective optimization objective 1 - time

$$\underset{t_{rej},\{t_b^1,\dots,t_b^{N-1}\}}{\arg min}\ time(h(D, t_{rej}, \{t_b^1, \dots, t_b^{N-1}\}))$$

And

Eq. 9 - multi-objective optimization objective 2 - error

$$\underset{t_{rej},\{t_b^1,\dots,t_b^{N-1}\}}{\arg min}\ error(h(D, t_{rej}, \{t_b^1, \dots, t_b^{N-1}\}))$$

where h denotes the DNN model with multiple branches, coupled with the Rejector (see Figure 11). Here, the time measures the model h computational inference time on a given dataset D when using a rejection threshold t_{rej} and a set of t_b branches thresholds. In turn, the error measures the error rate with the same set of thresholds. As a result, the proposed scheme aims to identify the optimal system thresholds that enhance computational inference time while minimizing error rates.

Therefore, in terms of overheads, the computational requirements for solving the multi-objective optimization process are only necessary during the training phase, which occurs off the end device. During this phase, the optimization process is performed to compute the trade-offs between error rate and energy consumption for the classification branch thresholds.

Once trained, the system operates with the predefined thresholds. Additionally, to address dynamic network environments, the network operator has the flexibility to periodically reevaluate and adjust the classification thresholds without requiring model updates. This allows for the rejection of a greater number of events in response to changing conditions, thus giving the network operator more time to perform model training tasks as needed.

5.4. Calibration

The network traffic behavior is highly dynamic and changes over time. This non-stationary behavior poses a challenge for current DNN-based NIDSs, as they often struggle to generalize network traffic behavior adequately (see Figure 10). In this context, a key assumption of this model involving early exits revolves around assessing the classification confidence of the model's branches to terminate inference prematurely. Therefore, the model's predicted confidence vector should reflect the ground-truth probabilities of the correctness of the model.

However, DNNs are known to produce both overconfident and underconfident classifications for input events. For example, if the model predicts a probability of 0.7 for a given class, it is expected that out of 100 predictions, approximately 70% of those are correct. If the percentage of correct predictions is below 70%, the model is overconfident, whereas if it is above 70%, it is underconfident. Therefore, to reliably deploy DNN based NIDSs with early exits, it is essential to ensure that their confidence levels accurately reflect their expected accuracy.

To address such a challenge, the proposed scheme conducts the confidence calibration of the designed DNN model. In practice, each DNN branch confidence output is calibrated to ensure their sampled confidence can function as their expected accuracy. Given this goal and recognizing that achieving perfect model calibration is not feasible because the model confidence is a continuous random variable, the model calibration is conducted through empirical approximation.

The objective is to calibrate each classifier c_i at every branch i, such that their output classification confidence $\hat{p}$ reflects their accuracy. To estimate the expected accuracy from a finite number of samples with a continuous random variable, the predictions are grouped into M interval bins, each of size $1/M$, and compute the accuracy over each bin. Each bin separates the classification confidence into finite intervals. For example, assuming $M = 10$, the first bin will contain all events with classification

confidence values ranging from 0.0 to up 0.1, whereas the second bin will contain those ranging from 0.1 up to 0.2.

Let B_m be the samples whose classification confidence $\hat{p}$ falls within the bin interval m, the accuracy of B_m can be computed as:

Eq. 10 - Average bin accuracy

$$acc(Bm) = \frac{1}{|\,Bm\,|} \sum_{i\,\in B_m} 1(\hat{y}_i = y_i)$$

where $\hat{y}_i$ is the predicted label, and y_i the true label. Then, the average confidence can be computed within the bin interval B_m according to the following equation:

Eq. 11 - Average bin confidence

$$conf(Bm) = \frac{1}{|\,Bm\,|} \sum_{i\,\in B_m} \hat{p}_i$$

where $\hat{p}_i$ is the classifier confidence for event i. Thus, a perfectly calibrated model will have $acc(B_m) = conf(B_m)$ for every m ∈ {1,...,M}. Finally, the calibration correctness can be measured through Expected Calibration Error (ECE) approach, as follows:

Eq. 12 - Expected Confidence Level - ECE

$$ECE = \sum_{m=1}^{M} \frac{|\,Bm\,|}{n} \,|\, acc(Bm) - \; conf(Bm) \,|$$

where n is the number of samples. As a result, the difference between acc (Eq. 10) and conf (Eq. 11) denotes the calibration gap for a given bin. The ECE can be used as an empirical metric of model calibration, where values close to zero denote a perfectly calibrated model.

The proposal uses the ECE metric to calibrate the post-training model. To achieve this, the temperature scaling technique was relied on. Each DNN branch classifier outputs a vector called logits, which is then passed through a softmax function to obtain the class probabilities. The temperature scaling goal is to find a temperature vector T that can be

used to divide the DNN logits such that it minimizes the resulting model ECE. Hence, temperature vector T can be used during the inference phase as follows:

Eq. 13 - Calibrated softmax function with temperature factor

$$softmax(z_i) = \frac{e^{z_i/T}}{\sum_j e^{z_i/T}}$$

where z_i is the sample *i* logit, and *T* the temperature vector. Temperature scaling does not affect the resulting model's accuracy, as *T* does not change the maximum of the softmax function. Therefore, it is possible to conduct simple parameter search to find a *T* vector that can reduce ECE. The resulting model will then use *T* to calibrate the classification confidence for later use to measure classification correctness.

5.5. Conclusions

The proposed model aims to facilitate the offloading of DNN-based NIDS inference while complying with energy efficient EC requirements. To achieve this goal, the application of early exits was relied on, where model branches can be deployed across *End Device*, *Edge*, and *Cloud* infrastructures (Figure 11). To ensure the model can reliably handle the dynamic behavior of network traffic, a reject option was incorporated at the final model branch executed in the cloud, which aims at identifying unreliable decisions over time. In addition, the threshold finding was formulated as a multi-objective optimization task to determine the optimal tradeoffs between energy consumption and error rate. Finally, the resulting model confidence values were calibrated to effectively generalize network traffic behavior while employing early exits, ensuring a reliable indication of model correctness. As a result, the proposed model facilitates the implementation of DNN-based NIDS on resource constrained devices. This approach enables efficient offloading while decreasing energy consumption at the End Device and lowering the resulting system error rate.

In addition, the multi-objective optimization model allows the network operator to define the desired acceptance rate for the utilized DNN branches. This is because the rejection rate should be carefully determined based on the operator's specific needs. A

high rejection rate may lead to a substantial number of events being offloaded into the cloud environment, potentially increasing costs associated with storage and processing, such as for later model update purposes. Conversely, accepting more samples on the first DNN branch at the end device could result in a higher error rate. In practice, the model provides flexibility for the network operator to adjust the operation points dynamically based on current network conditions. This approach allows for gradually increasing rejection rates as the deployed model ages, ensuring system reliability until the model update process is performed.

Chapter 6

Results and Evaluation

The proposal evaluation aims at answering the following RQs:

- (***RQ3***): How does the proposed multi-objective optimization improve system performance?
- (***RQ4***): Does the proposed model improve classification reliability?
- (***RQ5***): What are the system tradeoffs when implemented in an EC architecture?

The subsequent subsections provide further details about the implementation of the model and its performance.

6.1. Prototype

A proposal prototype was implemented in a distributed environment as illustrated in Figure 13. It considers the implementation of multiple End Devices executed on an Edge Infrastructure and a single Cloud Infrastructure for executing the offloaded task.

Each End Device executes the DNN-based NIDS pipeline with the proposed model (see Figure 11). To achieve this goal, it would continuously collect the network packets from a given monitored NIC through a capture library, such as Scapy API. The behavior of the collected packets could then be extracted using a manipulation library such as numpy, compounding a feature set with the features listed on Table 3. A single early exit was implemented on the chosen DNNs, where the first branch is executed at the End Device, and the second branch at the Cloud Infrastructure. The extracted feature set is used as input to the first DNN branch, implemented through Pytorch API v.2.1.0 (Figure 13, Class. Branch 1). The associated event label is used as input to the Alert module, if its classification confidence level surpasses acceptance threshold (Figure 11,

tc1). Otherwise, the output of the intermediate DNN layer at the first branch is offloaded to the cloud (Figure 13, Offload).

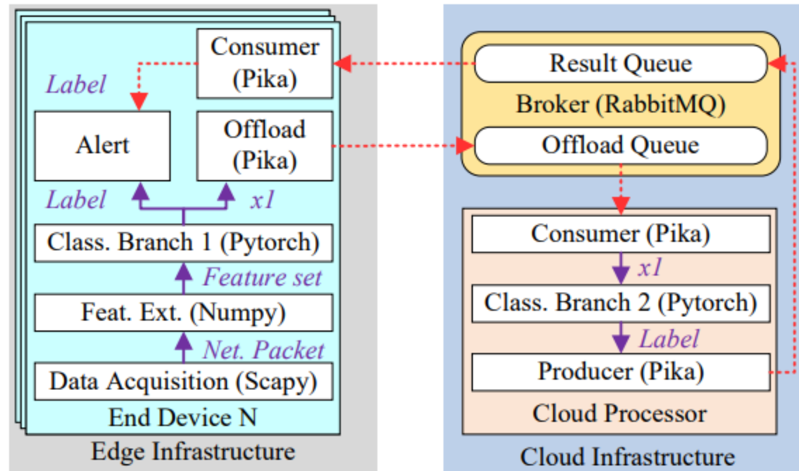


Figure 13 - Prototype implementation overview of the proposed model.

The communication channel between the Edge Infrastructure and the Cloud Infrastructure was implemented through a RabbitMQ broker. To this end, the prototype relies on two queues, namely Offload and Result. The first sends the first branch intermediate output from the End Device to the cloud, whereas the latter sends back the associated Label from the second branch as executed at the cloud. The sending and reading of the RabbitMQ messages were implemented through Pika API v.1.3.2.

At the Cloud Infrastructure, the RabbitMQ broker was deployed with the second DNN branch. To achieve this objective, a Cloud Processor module is executed, which continuously consumes the Offload Queue messages and feeds them as input to the *Class. Branch 2* module. The module executes the second (final) DNN branch and forwards the resulting Label to the Producer module, which produces a message to the Result RabbitMQ queue. Finally, the End Device reads the generated message through a Consumer module and forwards it to the Alert module.

To generate a resource-constrained device behavior, the End Device was implemented through a Raspberry Pi 3 Model B, with a 4-core Broadcom CPU and 1 GB of memory running on top of Raspberry Pi OS with kernel version 6.1. The Cloud Processor module was implemented through a dedicated Virtual Machine (VM) on top of IBM Cloud Computing. The VM was equipped with 2 virtual CPU cores, 8 GB of memory running on top of Ubuntu OS v. 22.04.

The Cloud Infrastructure was executed through multiple zones to adequately measure the resulting tradeoffs of the proposed scheme.

6.2. Model Building

The proposed model evaluation was done using the same DNN architectures evaluated previously in Chapter 5. To achieve such a goal, a single early exit component was introduced on each chosen DNN, as follows:

- ***AlexNet***. A classifier is introduced between the 1st and 2nd convolutional layers. The classifier flattens the 1st convolutional layer output by applying a fully connected layer with 1,600 input neurons followed by a 2-neurons output.
- ***MobileNet***. A classifier is introduced after the 2nd bottleneck layer. The classifier flattens the preceding layer output, followed by a 0.2 dropout layer and a fully connected layer with 1,280 input followed by a 2-neurons output.

Therefore, each evaluated DNN architecture consists of two branches encompassing the added layers from the first branch, while the last branch comprises the traditional DNN output (Figure 13, Class. Branch 1 and 2). The modified models are trained through the joint loss function with categorical loss for each branch, with a 1.0 branch layer weight w. The learning rate was set at 0.001. These models were implemented using PyTorch API version 2.1.0.

6.3. Energy Consumption Measurement

To measure the system's energy consumption, a current-voltage power ratio consumption meter was used, connected to the electrical outlet that powers the Raspberry Pi 3. In practice, firstly the idle power consumption is measured and then the difference in energy consumption computed when performing the inference task (using the proposed early exit) on the Raspberry Pi. It is important to note that the inference task energy consumption also includes the offloading to the cloud when the second DNN branch is required (see Figure 13). Therefore, the measurement replicates a realistic setting where

a portion of events are classified at the End Device using the 1st branch, while the remaining events are classified using the 2nd branch executed at the Cloud Infrastructure.

6.4. DNN-based NIDS with Early Exits

The first experiment aims to answer *RQ3* and investigate how the proposed multi-objective optimization can improve the system performance. To achieve this goal, the aim is to find the optimal tradeoff between energy consumption and error rate according to the used classification thresholds (t^c). In practice, the classification thresholds were adjusted for each selected DNN to account for the application of two exits. Therefore, the multi-objective optimization aims at finding the optimal classification thresholds for the first branch at the *End Device* (t_a^{c1}, t_n^{c1}), and the second branch at the *Cloud Infrastructure* (t_a^{c2}, t_n^{c2}).

To achieve this goal, the scheme was implemented with a multi-objective optimization task using the Non-dominated Sorting Genetic Algorithm (NSGA-II) [19] algorithm implemented on top of pymoo API. The NSGA-II uses a 100-population size, 1000 generations, a crossover of 0.9, and a mutation probability of 0.1. The multi-objective feature selection aims to decrease energy consumption (σ) and error rate (ϵ). The energy consumption was computed through the prototype, using the measurement approach described previously. The selected DNNs are trained using the *MAWIFlow* January training dataset, while the objectives were measured through the January validation dataset. The resulting model's performance is measured through the testing dataset.

Figure 14a shows the Pareto curve of the proposed multi-objective optimization approach. Energy consumption is normalized according to the baseline with the traditional approach without using the proposed early exit strategy (Figure 10). It is possible to observe a direct tradeoff between the proposal error rate and the resulting energy consumption at the End Device. Increasing the number of events accepted at the 1st branch increases the resulting system error rate and reduces energy consumption. For example, the multi-objective optimization can achieve a ≈ 7% error rate while demanding

only ≈ 14% of the energy consumption compared to the traditional approach. In addition, energy consumption can be reduced to as little as 1% if a 10% error rate is tolerated.

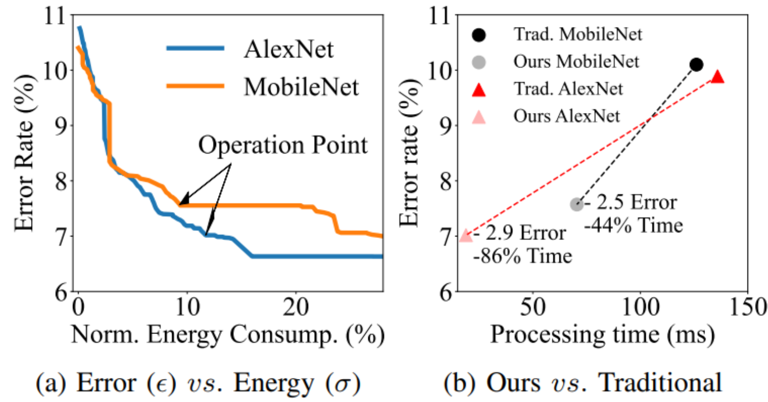


Figure 14 - multi-objective optimization for evaluated architectures. Processing time was measured on a Raspberry PI Model B. Error rate was measured on *MAWIFlow* Feb. and March.

An additional investigation was performed on how the proposed multi-objective optimization can enhance system performance. In this case, as the operating point must align with the operator's requirements, an average operating point was selected for the remaining evaluations (Figure 14a, Operation Point).

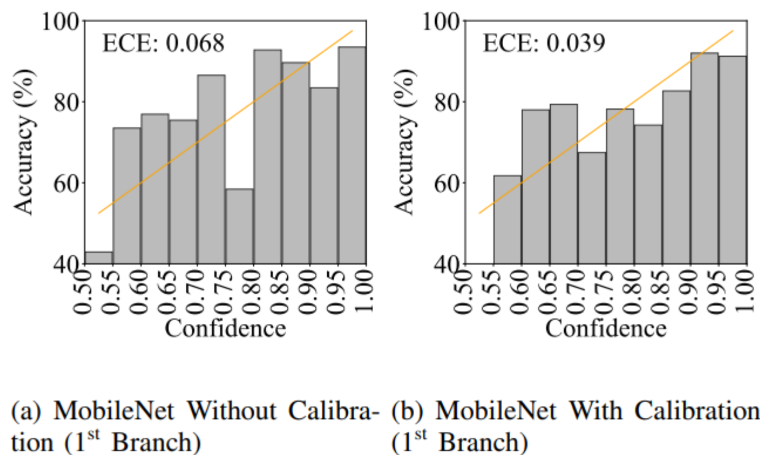
Figure 14b compares the proposed scheme's error and required processing resources with the chosen operation point vs. the traditional approach without applying early exits. It is possible to observe a significant improvement in the required processing resources while still achieving a reduction in the error rate. In practice, the scheme can reduce average required processing resources (and thus, energy consumption) at the End Device in 86% and 44% while decreasing the error rate in 2.9% and 2.5% compared to traditional approaches for the AlexNet and MobileNetV2 DNNs, respectively.

As a result, the application of early exits can pave the way for energy-efficient EC, as the proposed model significantly reduces the energy consumption of resource-constrained devices. Moreover, this improvement in energy consumption is achieved without compromising the resulting model's error rate.

Making use of the selected operation points (Figure 14a), and before applying it through the entire *MAWIFlow* year, the proposed model calibration approach was conducted. To achieve this objective, a simple heuristic-driven search was applied for each DNN branch to find the temperature *T* values for each confidence bin that improves

the branch ECE value. In practice, a 10-sized bin was considered for each branch (1st, and 2nd), and vary the temperature T from 0.0 to 1.0 in 0.00001 intervals. The temperature T value was used for each bin, which improves the resulting branch ECE.

Figure 15 shows the obtained reliability histogram for the 1st MobileNetV2 DNN branch on *MAWIFlow* January validation dataset, with vs. without the proposed confidence calibration scheme. It is possible to observe that the proposed calibration model can approximate the classification confidence values for each bin to reflect the expected model accuracy. As an example, the proposal reduces the ECE on the 1st MobileNetV2 branch by 43% and when compared to its non-calibrated counterpart. Similarly, when both branches are considered, the calibration approach reduces the obtained ECE values by 38% and 40% for the AlexNet and MobileNetV2, respectively. Therefore, by adjusting the model's classification confidence values, they can reliably be used for early exits and the proposed rejection approach.



(a) MobileNet Without Calibration (1st Branch) (b) MobileNet With Calibration (1st Branch)

Figure 15 - Reliability histograms for the 1st MobileNetV2 DNN branch on MAWIFlow January validation dataset.

To answer *RQ5*, the proposed model is applied throughout the entire *MAWIFlow* year. To achieve this objective, it was used the chosen operation point obtained through the multi-objective optimization approach (Figure 14a). In addition, the resulting model is passed through the confidence calibration procedure as shown in Figure 15. Similarly, the model is trained using the *MAWIFlow* January training dataset and evaluated as time passes without model updates.

Figure 16 shows the model's accuracy and rejection performance on the MAWIFlow dataset. Notably, accuracy also degrades but not as significantly as observed with the traditional approaches (Figure 16 vs. 2). It is important to note that the proposal achieves the accuracy benefit in two ways. First, through the application of the proposed early exit technique, which offloads to the Cloud Infrastructure a subset of events that require additional processing capabilities. Second, through the classification with a reject option at the 2nd DNN branch, which identifies and suppresses potential misclassifications at the cloud, thereby preventing false alerts. This characteristic can be observed as time passes with the resulting model's rejection rate (Fig. 8, Rejection Rate). On average, throughout the entire MAWIFlow year, the proposal rejected 8% and 6% of events for AlexNet and MobileNetV2, respectively. Recalling that the operating point must be chosen according to the operator's needs. In this case, it would be possible to further decrease the rejection rate, if required by the operator, by selecting an operating point with a higher error rate (Figure 14a).

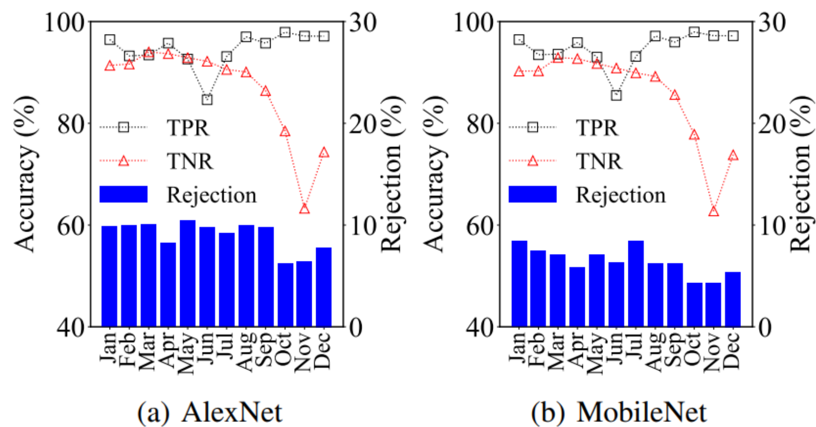


Figure 16 - Accuracy and rejection performance of the model as time passes on MAWIFlow dataset. DNNs are trained on MAWIFlow January training dataset. Inference uses the chosen operation point and the proposed calibration scheme.

A further investigation of the accuracy performance of the model was conducted, when compared to the traditional approaches. Figure 17 shows the monthly F1-Score improvement of the selected DNN architectures with vs. without the proposal. The proposed scheme improves the F1-Score by an average of 0.02 for both selected architectures. As a result, the proposed scheme significantly reduces energy consumption for the inference task on the End Device while keeping or even improving classification accuracy. This improvement is achieved by applying early exits as implemented by the

proposed model, which can lead to trade-offs due to communication overhead with the cloud environment.

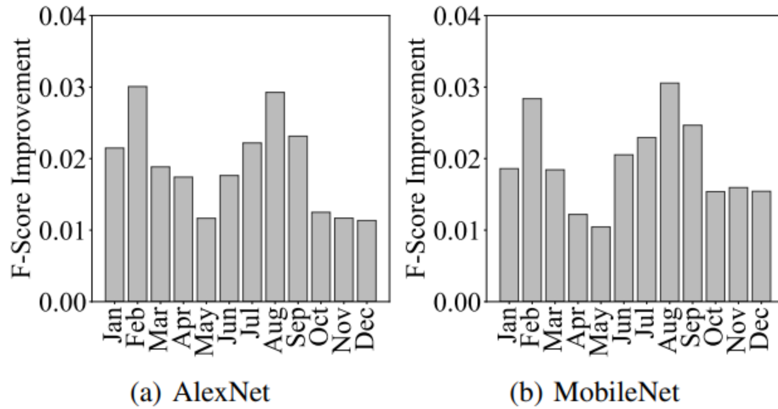


Figure 17 - F-Score performance improvement of the proposed scheme vs. the traditional approach.

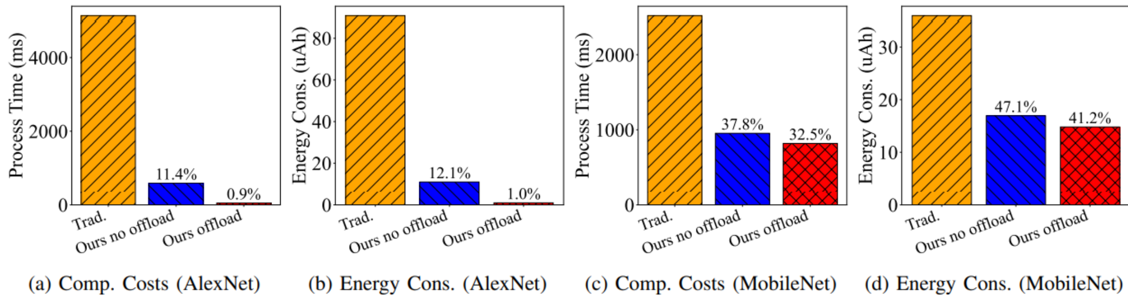
6.5. Cloud Offloading

To answer *RQ5* we analyze the performance of the proposal prototype (Figure 13) in a distributed cloud environment. To achieve this objective, the End Device is deployed in the state of Parana, in the southern region of Brazil, while the Cloud Infrastructure is deployed in two different IBM Cloud Services zones, namely Brazil South and Central US. The first deployment assesses the performance of the scheme with fewer network hops to the cloud, while the second evaluates it with a higher number of network hops, which can lead to increased classification offloading latency.

The first experiment investigates the computational processing benefits to the End Device achieved by offloading the inference task to the cloud. To achieve this objective, the proposed prototype was implemented using the previously selected operation points (see Figure 14) and offload the subset of events to the Cloud Infrastructure based on the specified acceptance thresholds. For this evaluation, the average event processing time was measured at the End Device, which includes both the execution of the 1st branch and the cloud offloading when required.

Figure 18a and Figure 18c show the average processing time per event with the proposed model vs. the traditional approach. In this case, the *Traditional* denotes the execution of the device's traditional DNN architecture without early exits. It is possible

to note that the proposed model substantially reduces the processing time at the *End Device* when compared to the traditional approaches. In practice, the scheme reduces processing costs to only 11% and 37% for the AlexNet and MobileNetV2, respectively, when both exits are executed at the *End Device (Ours (No Offload)).* In addition, if the 2nd DNN branch is offloaded to the cloud, the average event processing time is decreased to as little as 1% for AlexNet and to 32% for MobileNetV2. This substantial difference between the selected DNN architectures is caused by the location where the 1st branch is introduced. While the 1st AlexNet branch is introduced after the first set of convolutional layers, the 1st MobileNet branch is introduced only after the 2nd bottleneck layer, thereby requiring additional processing before inference can terminate prematurely. Notwithstanding, given that approximately 90% of events are classified at the 1st DNN branch, which requires less processing, offloading a subset of events to the *Cloud Infrastructure* can reduce processing time compared to the traditional approach.



(a) Comp. Costs (AlexNet) (b) Energy Cons. (AlexNet) (c) Comp. Costs (MobileNet) (d) Energy Cons. (MobileNet)

Figure 18 - Average inference computational time and energy consumption per event at the *End Device* with the proposed model. *Traditional* denotes the execution of the entire DNN at the *End Device* without using the early exit approach. *Ours (No offload)* denotes using the proposed approach, with both exits processed at the *End Device*. *Ours (Offload)* denotes using the proposed approach, offloading the 2nd DNN branch to the Cloud Infrastructure.

The benefits achieved at the *End Device* with the proposed scheme are further investigated by measuring the average energy consumption per event, using the measurement technique. Figure 18b, and Figure 18d show the average energy consumption of the proposed model vs. the traditional on-device approach. Similarly, the energy consumption is substantially decreased compared to the execution of the entire inference process at the *End Device*. In this case, the energy consumption is decreased to only 12% and 47% if no offloading is used for AlexNet and MobileNet, respectively. Conversely, if a cloud offloading strategy is used, energy consumption is reduced to 1%

and 41% for AlexNet and MobileNet, respectively. Consequently, the proposed model significantly reduces processing costs and energy consumption at the *End Device*.

The second experiment investigates the trade-offs when offloading events to the *Cloud Infrastructure*. To this end, the average inference time per event was assessed across different deployed *Cloud Infrastructure* zones. Table 6 shows the average event inference time of the proposed scheme vs. the traditional on-device approach. In practice, the proposed approach conducts event inference by an average of 28 and 223 milliseconds when deployed in *Brazil* (closer to the *End Device*), while the traditional approach demands an average of 1385 and 669 milliseconds, a substantial reduction of 97% and 66% for the AlexNet and MobileNetV2.

Table 6 - Average event inference time of the proposed model according to the deployed Cloud Infrastructure zone.

DNN	Approach	Deploy Zone	Avg Event Inf. Time (ms)		
			Branch 1	Branch 2	Total
AlexNet	Ours	Brazil	18.08	10.28	28.36
		US	18.08	46.18	64.26
	Local	Device	20.24	1365.70	1385.94
MobileNet	Ours	Brazil	217.13	5.97	223.09
		US	217.13	32.6	249.73
	Local	Device	236.89	432.56	669.45

The deployment zone of the Cloud Infrastructure can impact the average event inference time when operating under real-time conditions.

This observation holds true when comparing different deployment zones throughout the evaluation over the entire *MAWIFlow* dataset (Table 6, *Deploy Zone*). In this case, the deployment zone can increase the average inference time by 35 and 26 milliseconds for the AlexNet and MobileNetV2, respectively. This impact shows that the deployment zone, measured by *Brazil* vs. *US* deployment, incurs a difference of event inference time of 11% for MobileNet, reaching 56% for AlexNet. As a result, as fewer events are offloaded into the cloud, and the 1st DNN branch demands fewer processing costs, the overall resulting average event inference time is also decreased. This characteristic suggests that the network operator should take into consideration the physical location of the *End Devices* when choosing to deploy the *Cloud Infrastructure*, as it can impact the execution time of the inference, and this is also architecture dependent.

The third experiment assesses the network trade-offs caused by offloading the outputs of the 1st DNN branch to the Cloud Infrastructure. Recalling that for every event offloaded to the cloud the prototype demands the publishing of the 1st DNN branch output to a RabbitMQ topic (see Figure 13). Figure 19a shows the network usage for each classification event offloaded to the cloud. In practice, the prototype requires ≈ 7.2 KB, and ≈ 14.5 KB for offloading each event from *End Device* to the *Cloud Infrastructure* for the AlexNet and MobileNetv2 DNNs respectively. It is important to note that this trade-off occurs only for that subset of events reaching the 2nd DNN branch. In this case, the chosen operation point (highlighted in Figure 14a) offloads only ≈ 10% and ≈ 8% of events on the entire dataset for the AlexNet and MobileNet DNNs respectively. In addition, each network flow (event) comprises ≈ 11.6 network packets in *MAWIFlow* dataset (Table 4). Consequently, the proposal substantially decreases the network overhead compared to the traditional offloading strategy, wherein all network packets must be offloaded to the cloud.

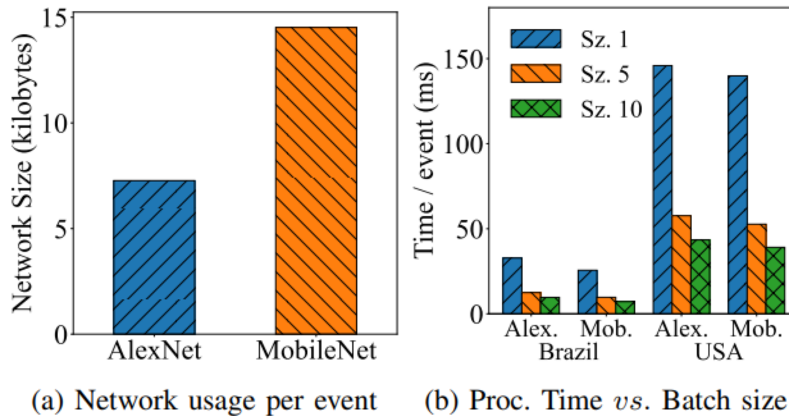


Figure 19 - Evaluation of the trade-offs incurred by offloading from *End Device* to *Cloud Infrastructure* the event classification (for each event reaching 2nd DNN branch). *End Device* is deployed in Brazil south region.

Finally, it was also evaluated how the proposal's average event processing time in the cloud can be further decreased when events are offloaded. To this end, a scenario wherein events are offloaded following a batch-oriented rationale was considered. In essence, the computation is offloaded when a batch of *N* events are available, reducing the average network communication time for publishing the RabbitMQ messages and conducting batch-based inference through Pytorch API at the *Cloud Infrastructure*. Figure 19b shows the impact on the average event processing time when a batch-oriented

approach is used. In this case, the prototype significantly reduces the processing time by up to 70% and 72% for the AlexNet and MobileNetv2 DNNs, respectively. As a result, despite the trade-offs on network usage and processing delays caused by the proposal offloading, the event average energy consumption and processing time was significantly reduced.

6.6. Conclusions

The proposal aimed at simultaneously improving the energy efficiency and processing costs of DNN-based NIDSs through an early-exit implementation rationale for edge computing. This was achieved by processing a subset of events at the End Device, thus relieving the *Cloud Infrastructure* when no additional processing capabilities are required. For the subset of events where classification cannot be reliably conducted, the *End Device* requests assistance from the *Cloud Infrastructure*, leading to a reduction in the processing footprint on both entities.

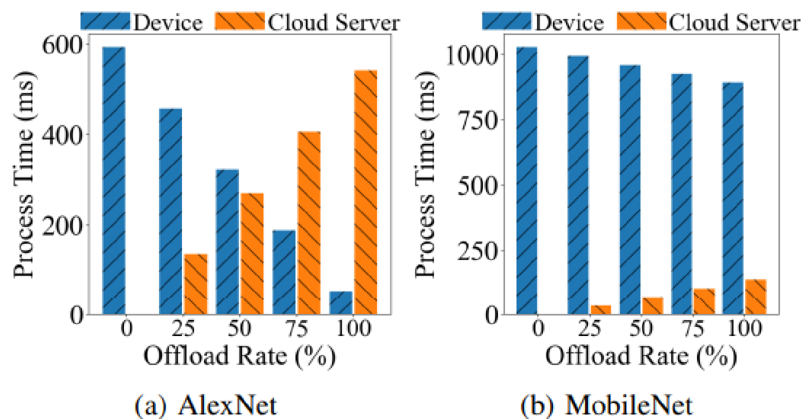


Figure 20 - Trade-off on process time vs offload rate to Cloud Infrastructure. The offload rate must be defined based on the operator’s needs and can be adjusted accordingly.

Thanks to the early exit strategy, this cooperation between the *End Device* and the *Cloud Infrastructure* can be adjusted based on the operator’s needs. For example, the *End Device* can execute both branches locally when sufficient processing capabilities and energy are available. Conversely, it can offload all computation to the *Cloud Infrastructure*, when necessary, thereby relieving the *End Device* of processing demands. Figure 20 shows the process time trade-off according to the adopted offload rate to the

Cloud Infrastructure. The scheme enables the dynamic adjustment of offload rate according to the current *End Device* capabilities. In addition, these benefits are achieved with minimal network communication overheads (Figure 19a). Consequently, it is demonstrated that a strategy integrating the *End Device*, which locally conducts inference for a subset of events, with the *Cloud Environment*, which provides additional processing capabilities, can effectively reduce the processing footprint of the overall system.

Chapter 7
Conclusion

This work investigated the challenges of intrusion detection in resource-constrained IoT devices and proposed a novel approach to enhance efficiency and reliability. The study focused on mitigating the limitations of traditional Deep Neural Networks (DNNs), which often prove computationally expensive for deployment on such devices. To address this, the research introduced a DNN model with early exits, enabling faster inference and reduced energy consumption. The proposed model incorporated a reject option at the final DNN branch to enhance classification reliability. This mechanism allowed the system to abstain from making decisions on uncertain or challenging events, thereby minimizing the risk of misclassifications. Furthermore, a multi-objective optimization approach was employed to determine the optimal balance between energy consumption and accuracy. This ensured that the system could effectively leverage early exits while maintaining a high level of performance.

An evaluation of the proposed model was conducted using the MAWIFlow dataset, a comprehensive collection of real-world network traffic. The results demonstrated significant improvements in processing time and energy consumption compared to traditional DNNs. Notably, the introduction of early exits did not compromise classification accuracy; instead, it led to an improvement in the F1-Score. This highlights the effectiveness of the proposed approach in achieving both efficiency and reliability. The deployment of the proposed model in an edge computing environment was also explored. This demonstrated the potential for further enhancing efficiency by distributing the computational workload between the resource-constrained device and the cloud.

Finally, as future work, we plan to optimize the location of introduced DNN early exits to reduce network usage further. In addition, we aim to leverage rejected events to

incrementally adjust deployed DNN branches, thereby addressing the nonstationary behavior of network traffic.

Bibliographic References

[1] "Mid-Year Update: 2023 SonicWall Cyber Threat Report, August, 2023 (accessed October 5, 2023).," [Online]. Available: https://www.sonicwall.com/medialibrary/en/white-paper/mid-year2023-cyber-threat-report.pdf.

[2] B. Molina-Coronado, U. Mori, A. Mendiburu and J. Miguel-Alonso, "Survey of network intrusion detection methods from the perspective of the knowledge discovery in databases process," *IEEE Transactions on Network and Service Management, vol. 17,* p. 2451–2479, 12 2020.

[3] Al-Garadi, M. Ali, A. Mohamed, A. K. Al-Ali, X. Du, I. Ali and M. Guizani, "A Survey of Machine and Deep Learning Methods for Internet of Things (IoT) Security," *IEEE Communications Surveys & Tutorials 22, no. 3,* 2020.

[4] L. Ma, Y. Chai, L. Cui, D. Ma, Y. Fu and A. Xiao, "A Deep Learning-Based DDoS Detection Framework for Internet of Things," *2020 IEEE International Conference on Communications (ICC),* 2020.

[5] M. Ge, N. F. Syed, X. Fu, Z. Baig and A. Robles-Kelly, "Towards a Deep Learning-Driven Intrusion Detection Approach for Internet of Things," *Computer Networks 186,* 02 2021.

[6] E. Li, L. Zeng, Z. Zhou and X. Chen, "Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing," *IEEE Transactions on Wireless Communications 19, no. 1,* p. 447–457, 01 2020.

[7] W. Seifeddine, C. Adjih and a. N. Achir, "Dynamic hierarchical neural network offloading in IoT edge networks," *10th IFIP International Conference on*

Performance Evaluation and Modeling in Wireless and Wired Networks (PEMWN) IEEE, 09 2021.

[8] Laskaridis, Stefanos, A. Kouris and N. D. Lane, "Adaptive Inference through Early-Exit Networks: Design, Challenges and Directions.," *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning, Virtual WI USA: ACM,* 2021.

[9] R. Fontugne, P. Borgnat, P. Abry and K. Fukuda, "MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking," *Proceedings of the 6th International Conference,* 2010.

[10] S. Teerapittayanon, B. McDanel and H. Kung, "BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks," *23rd International Conference on Pattern Recognition (ICPR),* p. 2464–2469, 2016.

[11] A. Konak, D. W. Coit and A. E. Smith, "Multi-Objective Optimization Using Genetic Algorithms: A Tutorial," *Reliability Engineering & System Safety 91, no. 9,* 09 2006.

[12] L. Kong, J. H. J. Tan, G. Chen, S. Wang, X. Jin, P. Zeng, M. Khan and S. K. Das, "Edge-computing-driven internet of things: A survey," *ACM Computing Surveys,* pp. 1-41, Dec 2022.

[13] J. Liu, Y. Gao and F. Hu, "A fast network intrusion detection system using adaptive synthetic oversampling and lightgbm," *Computers & Security,* p. 102289, 2021.

[14] C. Feng, P. Han, X. Zhang, B. Yang, Y. Liu and L. Guo, "Computation offloading in mobile edge computing networks: A survey," *Journal of Network and Computer Applications,* p. 103366, Jun 2022.

[15] A. A. Gendreau and M. Moorman, "Survey of Intrusion Detection Systems towards an End to End Secure Internet of Things.," *IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud),* pp. 84-90, 2016.

[16] H.-Y. Kwon, T. Kim and M.-K. Lee, "Advanced Intrusion Detection Combining Signature-Based and Behavior-Based Detection Methods," *Electronics 11, no. 6,* p. 867, 9 03 2022.

[17] E. Benkhelifa, T. Welsh and W. Hamouda, "A Critical Review of Practices and Challenges in Intrusion Detection Systems for IoT: Toward Universal and Resilient Systems," *IEEE Communications Surveys & Tutorials 20, no. 4,* p. 3496–3509, 2018.

[18] S. Gamage and J. Samarabandu, "Deep Learning Methods in Network Intrusion Detection: A Survey and an Objective Comparison," *Journal of Network and Computer Applications 169,* 11 2020.

[19] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation 6, no. 2,* pp. 182-197, 04 2002.

[20] N. Srinivas and K. Deb, "Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms," *Evolutionary Computation 2, no. 3,* pp. 221-248, 09 1994.

[21] R. A. M. Filho and S. R. Vergilio, "A Multi-Objective Test Data Generation Approach for Mutation Testing of Feature Models," *Journal of Software Engineering Research and Development 4,* 12 2016.

[22] F. Hussain, R. Hussain, S. A. Hassan and E. Hossain, "Machine Learning in IoT Security: Current Solutions and Future Challenges," *IEEE Communications Surveys & Tutorials 22, no. 3,* p. 1686–1721, 2020.

[23] M. W. Berry, A. Mohamed and B. W. Yap, Supervised and Unsupervised Learning for Data Science, Springer International Publishing, 2020.

[24] D. Nguyen, G. Memik, S. Memik and A. Choudhary, "Real-Time Feature Extraction for High Speed Networks," *In International Conference on Field Programmable Logic and Applications,* p. 438–443, 2005.

[25] D. A. Bashar, "SURVEY ON EVOLVING DEEP LEARNING NEURAL NETWORK ARCHITECTURES," *Journal of Artificial Intelligence and Capsule Networks 2019, no. 2,* pp. 73-82, 04 12 2019.

[26] M. A. Nielsen, Neural Networks and Deep Learning, Determination Press, 2015.

[27] Z. Li, F. Liu, W. Yang, S. Peng and J. Zhou, "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects," *IEEE Transactions on Neural Networks and Learning Systems 33, no. 12,* p. 6999–7019, 12 2022.

[28] Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike and M. S. Nasrin, *The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches,* arXiv, 2018.

[29] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Communications of the ACM 60, no. 6,* pp. 84-90, 24 05 2017.

[30] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *IEEE/CVF Conference on Computer Vision and Pattern Recognition,* p. 4510–4520, 2018.

[31] G. Fumera, F. Roli and G. Giacinto, "Reject Option with Multiple Thresholds," *Pattern Recognition 33,* p. 2099–2101, 12 2000.

[32] L. She, W. Wang, J. Wang, Z. Lin and Y. Zeng, "Progressive supervised pedestrian detection algorithm for green edge–cloud computing," *Computer Communications, vol. 224,* pp. 16-28, Aug 2024.

[33] Z. Ning, X. Kong, F. Xia, W. Hou and X. Wang, "Green and sustainable cloud of things: Enabling collaborative edge computing," *IEEE Communications Magazine, vol. 57, no. 1,* pp. 72-78, Jan 2019.

[34] A. Maia, A. Boutouchent, Y. Kardjadja, M. Gherari, E. G. Soyak, M. Saqib, K. Boussekar, I. Cilbir, S. Habibi, S. O. Ali, W. Ajib, H. Elbiaze, O. Erc¸etin, Y. Ghamri-Doudane and R. Glitho, "A survey on integrated computing, caching, and

communication in the cloud-toedge continuum," *Computer Communications, vol. 219,* pp. 128-152, Apr 2024.

[35] A. Thakkar and R. Lohiya, "A Review on Machine Learning and Deep Learning Perspectives of IDS for IoT: Recent Updates, Security Issues, and Challenges," *Archives of Computational Methods in Engineering 28, no. 4,* p. 3211–3243, 06 2021.

[36] A. Verma and V. Ranga, "Machine Learning Based Intrusion Detection Systems for IoT Applications," *Wireless Personal Communications 111, no. 4,* p. 2287–2310, 04 2020.

[37] P. Shukla, "ML-IDS: A Machine Learning Approach to Detect Wormhole Attacks in Internet of Things," *Intelligent Systems Conference (IntelliSys),* p. 234–240, 2017.

[38] P. Maniriho, E. Niyigaba, Z. Bizimana, V. Twiringiyimana, L. J. Mahoro and T. Ahmad, "Anomaly-Based Intrusion Detection Approach for IoT Networks Using Machine Learning," *International Conference on Computer Engineering, Network, and Intelligent Multimedia (CENIM),* pp. 303-308, 2020.

[39] J. Zhang, F. Li and F. Ye, "An Ensemble-Based Network Intrusion Detection Scheme with Bayesian Deep Learning," *ICC 2020 - 2020 IEEE International Conference on Communications (ICC),* 2020.

[40] J. Zhang, F. Li, H. Wu and F. Ye, "Autonomous Model Update Scheme for Deep Learning Based Network Traffic Classifiers," *IEEE Global Communications Conference (GLOBECOM),* 2019.

[41] N. Tekin, A. Acar, A. Aris, A. S. Uluagac and V. C. Gungor, "Energy Consumption of On-Device Machine Learning Models for IoT Intrusion Detection," *Internet of Things 21,* 04 2024.

[42] T.-M. Hoang, T.-A. Pham, V.-V. Do, V.-N. Nguyen and M.-H. Nguyen, "A Lightweight DNN-Based IDS for Detecting IoT Cyberattacks in Edge

Computing," *International Conference on Advanced Technologies for Communications (ATC),* p. 136–140, 2022.

[43] Y. Wang, G. Qin, M. Zou, Y. Liang, G. Wang, K. Wang, Y. Feng and Z. Zhang, "A Lightweight Intrusion Detection System for Internet of Vehicles Based on Transfer Learning and MobileNetV2 with Hyper-Parameter Optimization," *Multimedia Tools and Applications,* 07 2023.

[44] S. Sharma, V. Kumar and K. Dutta, "“Multi-Objective Optimization Algorithms for Intrusion Detection in IoT Networks: A Systematic Review," *Internet of Things and Cyber-Physical Systems,* 02 2024.

[45] M. Roopak, G. Y. Tian and J. Chambers, "An Intrusion Detection System Against DDoS Attacks in IoT Networks," *10th Annual Computing and Communication Workshop and Conference (CCWC),* p. 562–567, 2020.

[46] H. Asgharzadeh, A. Ghaffari, M. Masdari and F. S. Gharehchopogh, "Anomaly-Based Intrusion Detection System in the Internet of Things Using a Convolutional Neural Network and Multi-Objective Enhanced Capuchin Search Algorithm," *Journal of Parallel and Distributed Computing 175,* pp. 1-21, 05 2023.

[47] M. Colocrese, E. Koyuncu and H. Seferoglu, "Early-Exit Meets Model-Distributed Inference at Edge Networks," *2024 IEEE 30th International Symposium on Local and Metropolitan Area Networks (LANMAN),* pp. 39-44, Aug 2024.

[48] S. Angelucci, R. Valentini, M. Levorato, F. Santucci and C. F. Chiasserini, "Edge Computing with Early Exiting for Adaptive Inference in Mobile Autonomous Systems," *ICC 2024 - IEEE International Conference on Communications,* pp. 2980-2985, 2024.

[49] D. J. Bajpai, A. Jaiswal and M. K. Hanawal, "I-SplitEE: Image Classification in Split Computing DNNs with Early Exits," *ICC 2024 - IEEE International Conference on Communications,* pp. 2658-2663, 2024.

[50] A. Moore, "Discriminators for use in flow-based classification," Dept. Comput. Sci., Univ. London, London, UK, 2005.

[51] V. R. G. Engelen and W. Joosen, "Troubleshooting an intrusion detection dataset: the cicids2017 case study," *2021 IEEE Security and Privacy Workshops (SPW),* May 2021.

[52] MAWI, "MAWI Working Group Traffic Archive - Samplepoint F," 2023. [Online]. Available: https://mawi.wide.ad.jp/mawi/.

[53] R. Narmeen, P. Mach, Z. Becvar and I. Ahmad, "Joint Exit Selection and Offloading Decision for Applications Based on Deep Neural Networks.," *IEEE Internet of Things Journal 11, no. 23,* Jan 2024.