DEEP LEARNING-BASED ANOMALY DETECTION FOR EDGE-LAYER DEVICES

By

Jonathan Hunter

Farah Kandah
Professor of Computer Science
(Chair)

Michael Ward
Professor of Computer Science
(Committee Member)

Anthony Skjellum
Professor of Computer Science
(Committee Member)

Donald R.Reising
Professor of Computer Science
(Committee Member)

DEEP LEARNING-BASED ANOMALY DETECTION FOR EDGE-LAYER DEVICES

By

Jonathan Hunter

A Thesis Submitted to the Faculty of the University of Tennessee at Chattanooga in Partial Fulfillment of the Requirements of the Degree of Master of Science: Computer Science

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

May 2022

ABSTRACT

This thesis work proposes a novel DL-based anomaly detection framework for IoT environments, employing higher-capacity embedded devices as a first line of defense for the IoT edge layer. In the proposed framework, embedded devices implement the DL anomaly detection engine at the network gateway and adapt to potential attacks by retraining on incoming network traffic. In order to test the feasibility of this framework, two neural network models, trained on variations of the CICIDS 2018 Intrusion Detection Data Set, are deployed and tested on the Raspberry Pi 4. Model performance metrics, including fit and evaluation time across various batch and data sizes, are compared alongside those of identical models running on higher-capacity devices. Device resource metrics of CPU and Memory usage are monitored for comparison across model variations, batch and data sizes.The potential benefit of retraining models at the edge is evaluated by comparing performance of models executing consistent retraining.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1 INTRODUCTION 1

1.1 Research Questions 3
1.2 Motivation 3
1.3 Organization 5

2 BACKGROUND INFORMATION 6

2.1 IoT and Embedded Devices 6
2.1.1 IoT System Architecture 7
2.2 Intrusion Detection System 7
2.2.1 Host-based Intrusion Detection 8
2.2.2 Network Flow-based Intrusion Detection 9
2.2.3 Anomaly Detection 10
2.3 Machine Learning 11
2.3.1 Deep Learning 11
2.3.2 Terminology 12
2.4 Principal Component Analysis 15

3 RELATED WORK 17

3.1 Machine Learning Intrusion Detection System for IoT 17
3.1.1 Novel Framework Solutions 18
3.1.2 Lightweight ML-based Intrusion Detection for IoT 18
3.1.3 Dimensionality Reduction Methods 20
3.2 Summary and Discussion 21

4 PROPOSED FRAMEWORK METHODOLOGY 23

4.1 Concept of Operations 23
4.2 Initial Model Training and Deployment 24
4.3 Network Flow Export and Collection 24
4.4 Flow Record Evaluation 25

4.5 Model Update . . . 25
4.5.1 Primary Model Update . . . 26
4.5.1.1 Anomaly Classification . . . 26
4.5.1.2 Model Performance Monitoring . . . 26
4.5.1.3 Model Retraining and Redeployment . . . 27
4.5.2 Secondary Model Update . . . 27

5 TESTING . . . 31

5.1 Devices Used . . . 31
5.2 Data and Preprocessing . . . 32
5.3 Scaling and Dimensionality Reduction . . . 33
5.4 Model Architecture Determination and Training . . . 35
5.5 Record Timestamp Generation . . . 36
5.6 Model Evaluation and Fit Testing . . . 38
5.6.1 Fit and Evaluation Time . . . 40
5.6.2 Fit and Evaluation Accuracy . . . 40
5.6.3 Device Resource Metrics . . . 41
5.6.4 Stress Simulation . . . 41

6 RESULTS AND DISCUSSION . . . 43

6.1 Results Processing and Comparison . . . 43
6.2 Device Average Task Speed Comparison and Analysis . . . 44
6.3 Effect of PCA, Batch Size on Embedded System Resources . . . 46
6.3.1 Memory . . . 47
6.3.2 CPU Load . . . 51
6.3.3 Stress Comparison . . . 54
6.4 Accuracy Analysis for Static and Dynamic Models . . . 57
6.5 Summary and Discussion . . . 61

7 CONCLUSION . . . 67

7.1 Closing Thoughts . . . 67
7.2 Future Work . . . 68

REFERENCES . . . 69

VITA . . . 73

LIST OF TABLES

5.1 Resulting best neural network architectures for IDS models 37

5.2 Values for Simulated Timestamp Generation 39

5.3 Average Records per Interval Value 39

6.1 Raspberry Pi batch-interval profiles for results analysis 43

6.2 Methods for results analysis 44

6.3 Average static, dynamic accuracy for both models 62

LIST OF FIGURES

2.1 IoT system architecture 8

2.2 Architecture of flow-based network monitoring, adapted from [1] 10

2.3 DNN feed-forward binary classifier: 10 principal components, 2 hidden layers, 16 neurons per layer 13

4.1 IDS Architectures 29

4.2 Proposed Framework Order of Operations 30

5.1 Amount of variation represented by each principal component of the 69-feature training data 34

6.1 Evaluation task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation) 46

6.2 Evaluation task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation) 47

6.3 Fit task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation) 48

6.4 Fit task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation) 49

6.5 Evaluation task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 on the Raspberry Pi 4 (models include 69-feature ReLU activation, 69-feature tanh activation, and 10-feature tanh activation) 50

6.6 Evaluation task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 on the Raspberry Pi 4 (models include 69-feature ReLU activation, 69-feature tanh activation, and 10-feature tanh activation) 51

6.7 Fit task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 on the Raspberry Pi 4 (models include 69-feature ReLU activation, 69-feature tanh activation, and 10-feature tanh activation) 52

6.8 Fit task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 on the Raspberry Pi 4 (models include 69-feature ReLU activation, 69-feature tanh activation, and 10-feature tanh activation) 53

6.9 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation alongside the same model running under stress) 54

6.10 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation alongside the same model running under stress) 55

6.11 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 10-feature tanh activation alongside the same model running under stress) 56

6.12 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 10-feature tanh activation alongside the same model running under stress) 57

6.13 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation, 69-feature tanh activation) 58

6.14 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation, 69-feature tanh activation) . . 59

6.15 CPU usage (in percent of 400 on 4 cores of Raspberry Pi 4) of fit and evaluation tasks for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation, 69-feature tanh activation, 10-feature tanh activation) 60

6.16 CPU usage (in percent of 400 on 4 cores of Raspberry Pi 4) of fit and evaluation tasks for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation, 69-feature tanh activation, 10-feature tanh activation) 61

6.17 Evaluation task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation, alongside both models running under stress) 62

6.18 Evaluation task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation, alongside both models running under stress) 63

6.19 Fit task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation, alongside both models running under stress) 64

6.20 Fit task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation, alongside both models running under stress) 65

6.21 Comparison of static and dynamic model accuracy across all batch sizes 66

CHAPTER 1

INTRODUCTION

Internet of Things (IoT) refers to a system of interrelated, interconnected physical devices that collect and transfer data over the Internet [2] [3]. IoT devices are generally limited in computational capability and so are often unable to incorporate or employ the various security mechanisms and protocols used by more powerful systems. In recent years, the vulnerabilities posed by IoT have made it a common target for malicious activity, and as a result, numerous IoT intrusion detection systems (IDS) have been proposed to handle attacks on the IoT ecosystem [2] [4].

As the nature of attacks targeting the IoT continues to evolve, it is vital for reliable intrusion detection systems to be autonomous and use contemporary techniques such as machine learning (ML) and deep learning (DL) to inform the IDS [2]. Even so, machine and deep learning-based intrusion detection is often regarded as an IoT security solution dependent on the incorporation of fog or cloud systems in the IoT system architecture [5–7]. The reasoning is simple: machine and deep learning algorithms require extensive computation, power, and storage capacity. Though what is defined as an IoT device can range broadly in terms of resource capacity and wireless capability, security solutions proposed for IoT often assume a level of capability inappropriate for executing complex machine learning algorithms in real-time.

However, methods to reduce the dimenesionality of training and testing data, including Principal Component Analysis (PCA), auto-encoders, and feature selection, are being proposed as a solution to implementing such systems in the IoT [8–11]. Such methods reduce the number of dimensions or features of the input data and, as a result, significantly reduce device memory requirements, training time, and processing.

The heterogeneity of IoT device capability, combined with dimensionality reduction methods and the nature of deep neural network models, present another option: pre-trained neural network-based intrusion detection on embedded devices. Unlike traditional machine learning algorithms, saved neural network models consist primarily of the architecture of the network and the weights of connections between neurons. The storage and processing of such information is beyond the capability of low-power wireless sensor nodes. But improving memory speed and processor architecture over the last several decades has resulted in small, cheap, commercially-available embedded devices with higher resource capacity. Regardless of whether or not this trend in technological advancement continues, IoT system deployments currently and will continue to have widely varying architectures and security needs, dependent upon their related enterprise and application.

The purpose of this research is to propose and evaluate an experimental IDS framework that implements its analysis engine at the edge layer, to serve as a first line of defense for IoT environments. As such, it is necessary to determine the feasibility of implementing deep neural network intrusion detection on limited-resource embedded devices, as well as the effect of dimensionality reduction on model performance and speed. To this end, we have trained two deep neural network models for binary classification using reduced versions and subsets of the Canadian Institute for Cybersecurity Intrusion Detection System dataset (CICIDS) 2018 [12]. These models will be deployed to devices of varying capacity, including the Raspberry Pi 4 8GB, XPS15-9570 personal computer with Intel Core i7-8750 CPU 2.20GHz and 16GB RAM, and University of Tennessee, Chattanooga (UTC) Simcenter's Firefly cluster, in order to examine and compare system and model performance metrics

1.1 Research Questions

In this thesis, we seek to address the following research questions:

- What is the trade-off of dimensionality reduction in terms of model performance, speed, and system resources?
- Is it possible for resource-limited embedded devices to employ DL network flow-based anomaly detection effectively?
- What are the optimal parameters, in terms of batch size and network flow collection rate, for implementing such systems on resource-limited devices?
- Further, can anomaly detection models, implemented on such devices, adapt to network traffic in a timely manner?

1.2 Motivation

Though lightweight classification algorithms are currently being proposed for securing the IoT [13–15], research surrounding the testing and comparison of their performance on embedded devices is still limited. In other efforts to circumvent the intensive computational demands of machine learning algorithms, proposed ML-based IDS frameworks often implement classifiers on more powerful remote systems in the cloud or fog layer [5–7].

In reviewing current literature surrounding these multi-layer framework proposals for implementing ML-based intrusion detection in the IoT, a number of observations were made:

1. Multi-layer framework proposals handle the computational cost of ML algorithms by implementing the classifier on servers outside the edge layer.
2. IDS implementations outside of the edge layer require that traffic anomalies go undetected until associated flow records reach the fog or cloud layer for analysis.

3. Though IDS implementation in the fog layer reduces bandwidth demands between the edge and cloud, the trade-off is an increase in traffic between fog and edge layer.
4. Dependency on fog servers demands sufficient infrastructure and cost, and may not be appropriate across all industries and applications that could make use of ML-based intrusion detection for IoT environments.
5. Retraining ML models at the edge would tailor anomaly detection to local network traffic, making the IDS more robust to attack.

In consideration of these observations alongside our research questions, the contributions of this thesis work are listed below:

- **Proposal of a novel IDS framework for IoT environment:** The proposed framework implements DL anomaly detection at the edge layer, refitting the model locally upon detection of an anomaly. In this way, edge-layer devices can serve as a first line of defense by adapting to incoming traffic in the interim between model updates.
- **Development and testing of DL models for intrusion detection:** Two models, trained on normalized and PCA-reduced datasets are deployed to devices of varying resource capacity in order to observe and compare performance metrics of average fit and evaluation time, and average evaluation accuracy.
- **Evaluation of the impact of deep learning models on embedded device resources:** Resource metrics, including CPU load and memory, are monitored during fit and evaluation of both models on the embedded device at various batch size and testing data interval values for both DL models. Comparison of these metrics provides further insight into the impact of DL on device resources.
- **Evaluation of the significance of single-epoch refit to model performance:** In order to ascertain the value of retraining models on new data at the edge layer, evaluation accuracy of

static (unchanging) and dynamic (refit at each interval) models is observed for comparison across various batch size and testing data interval values.

1.3 Organization

The remainder of this work is organized as follows: Chapter 2 covers background information and key concepts. Chapter 3 reviews related work previously conducted in the field. Chapter 4 outlines the concept of operations behind our proposed system framework. Chapter 5 details the stages of experimentation. Chapter 6 presents the results of testing and discusses their significance. Finally, Chapter 7 presents the conclusion of this work and discusses future research directions.

CHAPTER 2

BACKGROUND INFORMATION

2.1 IoT and Embedded Devices

Internet of Things (IoT) is the interconnection or convergence of various smart devices or 'things' that can communicate with one another, via the Internet [2, 4, 16]. More specifically, IoT devices can be defined as objects embedded with sensors, software, and in some cases, a pervasive user interface, connected to the Internet to relay and receive information to serve the broader purpose of making some aspect of industry or daily life simpler. Some examples of IoT devices include stand-alone embedded systems, such as the Raspberry Pi, Nvidia Jetson and Arduino products, low-power wireless sensor nodes for monitoring environmental conditions, as well as smart home appliances, such as smart fridges, smart tea-kettles, Google Home and Amazon Alexa. The 'things' in IoT can range broadly in terms of resource capacity and communication methods across a number of industries and applications.

This heterogeneity of IoT device capability, communication protocols and resources, in proportion to their range of applications, presents a number of challenges to be overcome in proposing appropriate security solutions. Inadequate security features in hardware, poorly designed software with a range of vulnerabilities, default passwords and other design errors have resulted in many IoT systems being much less secure than typical personal computer systems [17]. Further, the resource constraints of many IoT devices prevent them from utilizing existing security mechanisms, such as group-signature based authentication, homomorphic encryption, and public-key based solutions [18]. Many IoT devices do not even operate on the Internet Protocol (IP) as it introduces considerable overhead, instead relying simpler communication protocols and upstream

nodes to provide Internet access [17]. Studies of popular smart home devices have shown that due to differing standards and communication stacks, limited computational power and high number of interconnected devices, traditional security implementations are often not appropriate for IoT environments [16]. As the number of connected devices is projected to reach 75 billion by 2025 [19], research into novel techniques for securing the IoT is both critical and ongoing.

2.1.1 IoT System Architecture

IoT devices are connected to the broader Internet at the edge layer via gateway devices. As is shown in figure 2.1, IoT applications often function by collecting sensor data through these IoT devices, and forwarding that information to higher layers for processing. The cloud, which consists of powerful remote servers providing hosting services via virtual machines, is where most IoT data processing takes place. The fog takes advantage of existing Internet infrastructure to push some data processing geographically closer to the edge for more time-dependent IoT applications [17,20,21].

2.2 Intrusion Detection System

Intrusion Detection Systems (IDS) are mechanisms that monitor and analyze network traffic or device system calls in order to identify and protect against intrusions and potential malicious traffic that may threaten the confidentiality, availability, and integrity of information systems [4]. The two primary categries of IDS are Network-based (NIDS) and Host-based (HIDS), defined for their placement and detection mechanism. Where HIDS is designed to be implemented on a single system, NIDS captures and analyzes traffic across the network [4]. Typical IDS is composed of three distinct modules: network or host sensors, analysis engine, and reporting system [16]. Depending on the classification of IDS as network or host-based, sensors collect relevant information

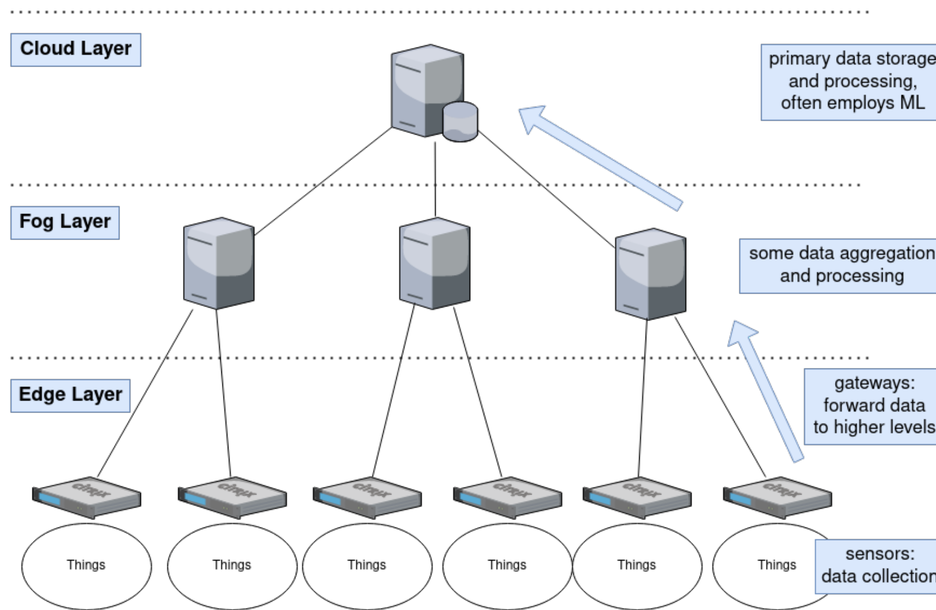


Figure 2.1 IoT system architecture

such as network traffic or host system calls and running processes. Information is then sent to the analysis engine, which investigates and detects intrusions in the current data. Upon detection of intrusion, the reporting system module alerts the network administrators [16].

2.2.1 Host-based Intrusion Detection

An HIDS exist on a singular device or 'host', monitoring for malicious activity that may be occurring within the system. It represents a more thorough approach to intrusion detection by analysing system calls, running processes, file-system changes and application logs, in addition to network traffic [16]. However HIDS can be computationally demanding and utilizes resources that may be otherwise allocated to performing standard system functions. NIDS is therefore the more common approach to securing IoT systems where device resources are limited [4, 6, 16].

2.2.2 Network Flow-based Intrusion Detection

Traditional NIDS is designed to perform deep packet inspection (DPI) in order to inspect the payload of every packet. Though DPI can provide greater insight into unusual behavior, the practical application of such a method is extremely difficult for high-speed connections. Network flow-based intrusion detection serves as a viable solution by analyzing flow records as opposed to packet content, significantly reducing the amount of data to process and network overhead. As such, routers from most major vendors come equipped with the capacity to perform flow accounting [1].

These network flow records, as defined in RFC 7011, consist of "a set of IP packets passing an observation point in the network during a certain time interval. All packets belonging to a particular flow have a set of common properties." Such common properties include source and destination addresses and port numbers [22].

The architecture of flow-based intrusion detection, illustrated in Figure 2.2, incorporates two primary modules: a flow exporter and a flow collector. The flow exporter, otherwise known as the observation point, captures and creates flow records from observed traffic on the network by applying timestamps to packet header information. Flow exporter continually sends expired flow records to the collector, determining expired records via packet header FIN (finish) and RST (reset) flags, or timeout thresholds [1]. TCP (Transmission Control Protocol) flags FIN and RST both signify that the connection is ended between hosts, either intentionally (FIN), or potentially as the result of a faulty connection (RST). The inactive timeout threshold determines expiration of flows for which no new packets have appeared in the allotted time. Active timeout threshold determines the expiration of active flows after a much longer period has passed. Default inactive and active thresholds for Cisco NetFlow, for example, are 15 seconds and 30 minutes, respectively [1]. Upon receiving expired flow records, the flow collector performs network monitoring and analysis.

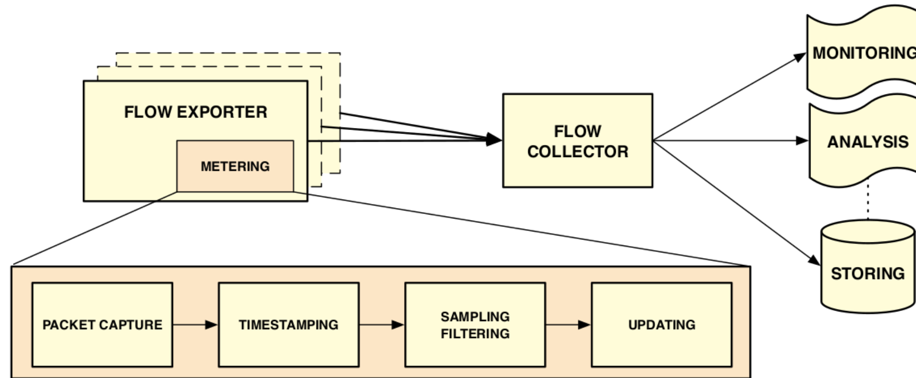


Figure 2.2 Architecture of flow-based network monitoring, adapted from [1]

2.2.3 Anomaly Detection

The two primary detection methods employed in IDS are signature-based and anomaly-based [4]. Signature-based detection approaches focus on matching network behavior to known attack signatures (specific traffic patterns or indicators of malware) stored in the IDS database. Anomaly-based IDS employs a detection strategy that compares current traffic patterns to a profile of normal behavior based on, and continually updated by, normal use network traffic data [2,4,16]. Anomalies are unusual behaviors in the network picked up by the classifier, located at the flow collector in the case of network flow-based intrusion detection systems. The principal benefit of anomaly-based detection, over signature-based, is in its potential to detect unknown, or zero-day attacks [14]. The issue with anomaly-based approaches, however, is the high false-positive rate that results from normal traffic behavior not matching the established pattern, as establishing the scope of all normal network behavior demands processing of a considerable amount of data. For this reason, researchers commonly employ the use of statistical and machine learning techniques in training anomaly-based IDS [16].

2.3 Machine Learning

In general terms, machine learning (ML) is the process of training a mathematical model to return accurate predictions from provided input, 'learning' primarily through optimization algorithms such as gradient descent, that seek to minimize the average 'cost' or 'loss' of the model. Cost is a function that represents the deviation between achieved and expected output. Though research into the field dates back to the late 1950s, recent exponential growth in computing power and data generation and collection has given machine learning algorithms greater significance to modern-day applications [23]. Machine learning models can be broken down into two major categories relating to the method by which the model is trained: supervised learning and unsupervised learning. In supervised learning, values relating the input and output relationship are available in the training data. In other words, training data is labelled. The goal is then to learn a function that best approximates the relationship between input and output, in order to make accurate predictions on future unlabelled input. In contrast, unsupervised learning techniques use unlabelled data in order to understand the underlying characteristics and relationships between observations [23, 24]. Supervised learning is the common approach for tackling classification tasks such as anomaly-based intrusion detection [23]. Establishing a profile of normal traffic requires that training data provide distinct examples of what can be considered 'normal' and 'abnormal' behavior, meaning each flow record of an IDS dataset used for training classifiers must be labelled appropriately by the provider.

2.3.1 Deep Learning

Deep learning is a category of machine learning models that seek to mimic our understanding of the biological neural networks of the human brain [13]. Though research into the potential of deep learning has existed for some time, the computational complexity and data requirements of training artificial neural networks has hindered the adoption and application of deep learning for

solving complex regression and classification tasks. In recent years, the development of advanced processing units and exponential increase in data collection have facilitated a resurgence in deep learning research [23, 24].

The basic structure of deep neural networks (DNNs), illustrated in Figure 2.3 consists of multiple hidden layers of neurons, or parallel elements connected via links and working in parallel to process data [13]. Neurons work by taking weighted input or 'activations' from the dataset or from output connections of previous layers, summing weighted activations together and employing an function to shape the summed value and output to neurons in the subsequent layer. Each connecting link between neurons carries the related weight value. The optimization of a neural network model is dependent on these weights and activation function. As such, training of DNNs consists of determining which weights and bias values best fit the model to the data according to resulting accuracy or loss [13, 23]. Referring to Figure 2.3 below, the example feed-forward DL model shown is trained on data consisting of 10 features or principal components, which correspond to the nodes of the input layer. The two subsequent hidden layers consist of 16 neurons each and are referred to as 'dense' layers, meaning each neuron is connected to every neuron of the previous and subsequent layers. The final output layer consists of a single neuron for binary classification. The output of this neuron determines whether or not the final result for input record is closer to 1 or 0, labelling it accordingly.

2.3.2 Terminology

Machine learning terminology used throughout this work will be clarified here:

- **Classifier:** In this work, the classifier refers to the model performing the classification task. Binary classification refers to potential model output of 0, 1 (benign or anomalous). Muilti-class classification spans a number of possible outputs equal to the input data labels (classes).

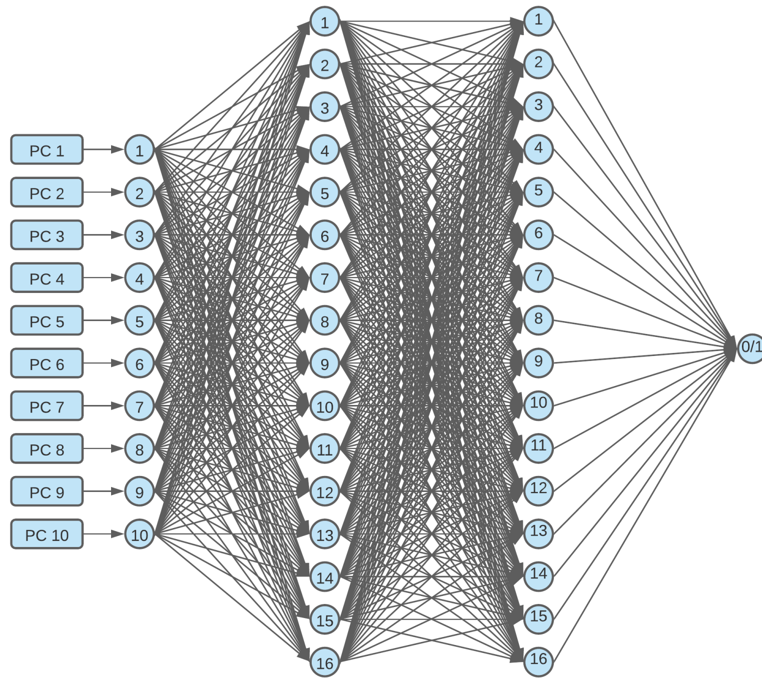


Figure 2.3 DNN feed-forward binary classifier: 10 principal components, 2 hidden layers, 16 neurons per layer

- **Model:** It is also important to distinguish the model from the underlying algorithm. The model is the result of training an algorithm on given data, making predictions on, or classifying, new input data.

- **Evaluate:** Evaluation, in reference to deep learning classifiers, is defined here as the process of a model performing classification over a number of records, labelling them as either anomalous or benign. In determining evaluation accuracy, the resulting labels are compared against those provided in the original data.

- **Fitting/Refitting:** Fitting refers to the process of training the model, or 'fitting' it to the given data. To refit is to retrain the model on new data.

- **Batch/Batch size:** Batch size is a variable provided to a machine learning model for training and evaluation. It represents the number of samples for which gradient descent is calculated

before updating model parameters. Higher batch sizes result in a more direct path to convergence (minimum loss), as more information is provided between parameter updates. The detriment to high batch size, in terms of limited-resource computing, is an increased demand in memory.

- **Optimizer:** Optimizer is the specific algorithm used to achieve the minimum loss of a DL model. Optimizers commonly employ some variation of gradient descent. Gradient descent is an algorithm which calculates the derivative, or 'slope' of the loss at a given point, to determine the direction (positive or negative) parameters must be adjusted by a learning rate variable to achieve lower loss. This process is repeated over a number of batches and epochs until it is determined that the model has reached its lowest potential loss value.

- **Epoch:** An epoch is a single pass of all training data through the neural network, through which weights are adjusted to reach minimum loss. Multiple epochs are generally necessary to optimize a DL model.

- **Features:** Features are the data that describe a record or sample. They can be visualized as the columns of a 2-dimensional array of samples. In deep learning, features are the input to the first layer of the model, meaning that reducing the number of features reduces the input and amount of data to be processed by the model.

- **Dimensionality:** The dimensionality of a dataset refers to the number of features per sample. Dimensionality reduction is the process of lowering this number of features. However, dimensionality and feature reduction are not necessarily equivalent. Where some feature reduction techniques use specific criteria to select a number of best features from those available, many dimensionality reduction techniques, such as PCA, generate an entirely new set of features based on the original data.

2.4 Principal Component Analysis

Principal Component Analysis (PCA) is a commonly-used linear dimensionality reduction technique which uses linear combinations to explain the variance - covariance of number of variables, the end result being a new set of features which cumulatively account for some amount of variation in the data [25]. More specifically, the PCA() function, imported from the Python scikit-learn decomposition library and used in testing, centers the input data with a mean 0 before employing Singular Value Decomposition (SVD) to project that data to a given dimensional space [26].

The fundamental concept of PCA dimensionality reduction is projection of a dataset X with n features or dimensions onto a subspace with k or fewer dimensions while retaining the majority of the variation of the original dataset [11]. The steps of this process are outlined below.

1. Features of the original dataset must be normalized and centered around a mean 0, with a standard deviation of 1. The method of normalization commonly used in data preprocessing for PCA is the min-max approach, demonstrated in Formula 2.1, in which X represents a feature of the dataset and X^i represents the i-th sample of that feature. max and min stand for the maximum and minimum values of the feature X in the data.

$$X^i = \frac{X^i - min}{max - min} \tag{2.1}$$

2. After features are centered around a mean zero, with standard deviation 1, the covariance matrix Cov_m is calculated according to Formula 2.2, where m represents the number of instances or records in the original dataset X and $X_(i)$ are the data points [11].

$$Cov_m = \frac{1}{m}\Sigma(X_(i))(X_(i))^T \tag{2.2}$$

3. The Eigen-vectors and corresponding Eigen-values of Cov_m are then calculated and Eigen-vectors are sorted by decreasing Eigen-values. In order to project to a subspace of dimension-

ality k, the first k Eigen-vectors (having the largest Eigen-values) are chosen to form matrix W [11].

4. Matrix W is then used to transform each sample onto the new subspace via Formula 2.3, in which X is an $n \times 1$ dimensional vector representing a single sample and y is the transformed $k \times 1$ dimensional vector representing the transformed sample in the new subspace of k dimensions [11].

$$y = W^T \times X \tag{2.3}$$

CHAPTER 3
RELATED WORK

Chapter 3 examines the related work that has already been conducted in the field. Section 3.1 explores the various methods by which researchers have proposed to implement machine learning intrusion detection in IoT security. Subsections 3.1.1 and 3.1.2 discuss framework solutions and comparative results of lightweight machine learning algorithms, respectively. Subsection 3.1.3 discusses the methods of dimensionality reduction used to reduce machine learning model training time and complexity and their varying effects on resulting model performance. Section 3.2 presents a summary and discussion of the literature.

3.1 Machine Learning Intrusion Detection System for IoT

Three major approaches to handling the computational requirements of implementing machine learning in IoT security have emerged in the research. Among these are novel approaches to the detection system architecture, employing the use of more powerful systems to take the brunt of the computational load involved; research and development of lighter-weight machine learning algorithms for use on more resource-limited devices; and feature and dimensionality reduction techniques to extract the most essential information from the data, reducing cost to evaluate without significantly impacting model performance.

3.1.1 Novel Framework Solutions

Common solutions to deep and machine learning-based intrusion detection for IoT systems incorporate the use of more powerful cloud and fog servers for both data storage and classification [5–7]. Samy et al. propose an attack detection framework implementing DNN models on fog-layer system clusters. The fog layer exists between the edge and cloud layers of typical IoT system architecture, comprising many connected domains containing virtualized servers, routers, simple data centers, and services [5]. Authors of [5] take advantage of this infrastructure to keep DL-based distributed IDS nearer the edge layer, providing lower latency, greater scalability, and high-speed response. The proposed architecture framework consists of IoT smart gateways at the edge layer, server clusters to implement the IDS in the fog layer, and more powerful cloud servers to run IDS performance monitoring and update.

Maharaja et al. propose a three-fold mechanism for securing IoT systems, employing VPN techniques for protecting the communication channels, ML-based traffic analysis for classification, and a challenge-response authentication mechanism to validate suspicious sources. In order to address the potential latency and response times of cloud-focused architectures, authors implement their decision-tree based traffic analysis and authentication control units in fog servers, offloading excess traffic to the cloud during peak volume periods. Testing found that the proposed framework achieved lower network cost and response times at higher traffic volumes than a pure-cloud implementation [7].

3.1.2 Lightweight ML-based Intrusion Detection for IoT

There has been some significant research into existing machine learning algorithms to determine which may serve as viable options for lightweight intrusion detection on resource-limited devices.

Authors of [27] assessed the performance of multiple ensemble and single ML classifiers on

CIDDS-001 [28], UNSW-NB15 [29], and NSL-KDD [30] intrusion detection datasets. Classifiers tested include Random Forest (RF), AdaBoost (AB), Gradient Boosted Machine (GBM), Extreme Gradient Boosting (XGB), Extremely Randomized Trees (ETC), Classification And Regression Trees (CART), and Multi-Layer Perceptron (MLP). Metrics for evaluation and validation include averages for accuracy, specificity, sensitivity and false positive rate. In addition, researchers observed and compared average response time of classifiers across each dataset when implemented on Raspberry Pi 3 Model B+, finding that while all classifiers perform similarly across all performance metrics, CART outperformed other models' accuracy by .01%, with the lowest false postive rate and average response time on the Raspberry Pi.

In another study, [31], authors used a portion of the CICIDS2017 dataset to compare Support Vector Machine (SVM) to DL methods for informing machine learning-based intrusion detection systems, specifically in detecting port scan attempts. They compare various neuron and hidden layer numbers for finding optimal conditions for their deep learning model and performed no feature extraction or dimensionality reduction on the dataset before running SVM, using all 80 of the original features. The study showed that deep learning methods outperformed SVM by a significant margin along accuracy, precision, recall, and F1 score performance metrics in detecting port scan attempts using this dataset. Though researchers provide the device specifications used in testing, there is no comparison of execution time in their table of performance metrics for each algorithm.

Additionally, lightweight neural network architectures are being proposed and tested. Authors of [15] proposed a lightweight 12-neuron recurrent Random Neural Network (RNN) for satisfactory performance with minimal computations and potential power consumption. They trained and tested their RNN on a Bot-IoT dataset using 1,177 samples from the dataset 10 "best" features, generated in [32], via coefficient and entropy techniques. Their results indicated that a lightweight RNN could serve as an appropriate solution to intrusion detection on resource-limited devices.

However, it should be noted that their experiment description made no mention of the specific devices used for testing, nor their relative resource capacity.

3.1.3 Dimensionality Reduction Methods

Various methods of dimensionality and feature reduction for the purpose of increasing the computational speed of training machine learning and deep learning-based intrusion detection systems have been explored in the literature.

The authors of [8] used gini importance, permutation importance, and drop-column importance to provide a detailed selection of the most important features for intrusion detection in the CICIDS2017 dataset, and found that permutation importance can be used to show 10 features represent approximately 99% of the cumulative importance in the data. Authors then tested the original dataset alongside a reduced dataset of only 10 features determined via permutation importance to compare feature reduction impact on performance metrics in attack detection, concluding that the difference in F1 score, false positive and false negative rates where negligible.

In [9], the authors compare Decision Tree (DT), RF, and SVM multiclass classifiers across the performance metrics of accuracy, precision, recall, and training time. Additionally, they use PCA to reduce the dimensionality of their dataset, testing the training time, training accuracy and testing accuracy of the RF model for different dimensions of: 3, 5, 10, 15, 20, 50, and 100. Authors used the IoT Network Intrusion Detection dataset from which they extracted 115 features categorized into 3 different classifications of attacks: Man in the Middle (MitM), Mirai, and Scan, alongside a minority of benign traffic. Researchers found that training the entire dataset at 15 features reduced using PCA optimized training time at approximately 1/6 the training time of the original dataset with minimal detraction to testing accuracy, though 10 features is optimal for both training time and accuracy across all subsets. Results of the study showed that RF identifies and classifies attacks and benign data at higher accuracy with a lower training time in all cases,

and that 10 principal components allow a significant reduction in computational resources without compromising the quality of the detection rate.

In [10], the authors compared the effect of PCA reduction on softmax regression and k-nearest neighbor classifiers using the KDD Cup 99 Data Set. Testing softmax regression on 3, 6, and 10 features. They found 10 to achieve the optimal detection rate, with KNN performing significantly slower at a negligible increase in accuracy. They concluded that, given computational complexity and speed, softmax regression on reduced dataset was the preferred algorithm for intrusion detection in IoT.

The authors of [11] compare PCA and auto-encoders as methods to reduce the dimensionality of the CICIDS2017 dataset. Similar to [8], the authors used PCA to reduce the dimensionality of the original dataset to 10 principal components. Using auto-encoders, they reduce the number of features to 56. They then tested the resulting reduced datasets using an RF classifier, finding that RF trained on the PCA-reduced dataset was able to achieve a maximum precision value of 99.6%. They conclude that PCA is the superior and faster method of dimensionality reduction.

3.2 Summary and Discussion

Among dimensionality reduction methods, principal component analysis stands out in its capacity to significantly reduce computational resource requirements for building the model without compromising resulting model performance metrics [9–11]. The current literature surrounding novel framework approaches to implementing ML-based intrusion detection in the IoT is limited in applicability. In other words, reliance upon fog systems for intrusion detection may not prove appropriate across all industries and applications that can benefit from ML-based intrusion detection for IoT environments. Additionally, such frameworks suffer from a number of drawbacks including increased traffic between the edge and fog layers, and anomalous traffic remaining undetected until reaching the fog or cloud layer. Research comparing lightweight machine learning algorithms

and dimensionality reduction techniques often limit performance metrics to the accuracy and precision of the resulting model, without consideration of the time, complexity, and resource usage in practical application [14, 15, 27, 31]. As such, further research into additional IDS frameworks, performance metrics and resource demands across multiple devices is necessary.

CHAPTER 4

PROPOSED FRAMEWORK METHODOLOGY

Chapter 4 outlines the concept of operations behind the proposed deep learning intrusion detection system. Section 4.1 provides a brief overview of the concept of operations and the remaining sections examine each stage in greater detail.

4.1 Concept of Operations

Figure 4.1 illustrates some differences between the proposed architecture (a) and those presented in the literature (b), adapted from [5]. Both frameworks rely upon more powerful remote servers for further analysis of attack detection information and IDS performance monitoring and updates (represented by the red and blue arrows, respectively). However, where previous frameworks relied upon fog layer servers to implement the traffic analysis module, the proposed architecture performs anomaly detection at the edge layer by implementing the neural network classifier at the gateway. Further detail regarding the order of operations for the proposed framework is provided in Figure 4.2.

The concept of operations for the proposed framework consists of 4 primary phases:

1. **Initial Model Training and Deployment:** The initial deep learning model is trained on the remote server to be deployed to edge-layer gateways throughout the network for anomaly detection.

2. **Network Flow Export and Collection:** Gateways generate flow records of the network traf-

fic for analysis, normalizing and reducing data through PCA for processing.

3. **Flow Record Evaluation:** Gateways employ deep learning model to evaluate and label incoming flow record data as anomalous or benign, sending subsequently labelled flow records to the remote server upon detection of an anomaly.

4. **Model Update:** Gateways periodically refit the model to local network traffic, to strengthen anomaly detection classifier. Upon receiving flow records from gateways, the remote server determines attacks, then updates and redeploys the initial model as necessary.

4.2 Initial Model Training and Deployment

This phase, represented in the first box of Figure 4.2, will require the collection of data. For the purpose of binary classification anomaly detection, training data must include sufficient number of labelled records, being diverse and robust in attack range and benign traffic patterns. Though intrusion Detection datasets or data generation methods used for initial model training may vary, it is pertinent that finalized training datasets include identical features to the normalized and reduced flow records later evaluated by the flow collector unit. Training of the initial model will comprise multiple epochs, parameters tuning as described in Section 2.3, and train/test split methods to optimize model performance.

4.3 Network Flow Export and Collection

In reference to terminology provided in Section 2.2, the network flow exporter and network flow collector units describe separate roles both performed by the IoT gateway. In this phase, represented in the second box of Figure 4.2, the flow exporter unit will observe network traffic and generate timestamped flow records, using flow accounting software such as CICFlowmeter used in [12]. The first timeout interval here represents the exporter's inactive and active flow

record timeout settings discussed in Section 2.2. The exporter unit will send expired records to the collector unit for network monitoring and analysis. The flow collector unit of the IoT gateway will continually aggregate, normalize and reduce data received from the flow exporter, in order to be consistent with the format of data determined in the initial model training phase.

4.4 Flow Record Evaluation

In this phase, represented in the third box of Figure 4.2, the gateway IDS will evaluate data to determine anomalous traffic using the resulting model from the training and deployment phase, classifying flow records as either benign or anomalous (potentially malicious). Detection of an anomaly in the current batch of flow records will trigger sending of all saved labelled flow records to the remote server for anomaly classification, initial model performance evaluation and possible retraining. In the event that no anomalies are detected in the current batch of evaluated flow records, the model may be refit on all labelled records saved on the device. The second time interval of Figure 4.2 here represents a timeout to determine if enough labelled data has aggregated to refit the model without degrading resulting model performance metrics.

4.5 Model Update

Two processes are distinguished for updating the anomaly detection classifier. One instance of model refitting will occur at the remote server, triggered by reception of anomalous-labelled flow records from a gateway in the network. The resulting model is considered the primary, that will be distributed to all edge layer gateways when training is complete. The other instance is the secondary, or local model refit, which occurs at the gateway between primary model updates.

4.5.1 Primary Model Update

The process of updating the primary anomaly detection classifier, shown in final box of Figure 4.2, is divided into 3 stages: anomaly classification, model performance monitoring and evaluation, and model retraining and redeployment.

4.5.1.1 Anomaly Classification

Flow records received by the remote server will be evaluated using a separate multi-class classifier to provide further insight regarding potential malicious traffic. Anomalous flow records that do not match attack types may be reclassified as benign. The binary classifier model would then be retrained on relabelled data and redeployed to gateways, as described in the model retraining and redeployment stage, in order to facilitate the introduction of new technologies and benign traffic patterns to the network. In addition to alerting systems administrators, attack identification by the multi-class classifier triggers primary model performance evaluation and update. Additional triggers for performance evaluation may include scheduled updates, discovery of new exploits, or high ratio of anomalous to benign-labelled aggregated records.

4.5.1.2 Model Performance Monitoring

In the performance monitoring stage, the primary binary classifier will evaluate the newly aggregated labelled flow records, monitoring model performance metrics of accuracy, precision, false positive and negative rates. Should the evaluation indicate a drop in these metrics from those achieved in the initial training phase, retraining of the primary model will be triggered. In the case that performance metrics achieved in this evaluation are equal to or greater than those achieved in initial model training, no further action will be taken, as the IDS is functioning properly.

4.5.1.3 Model Retraining and Redeployment

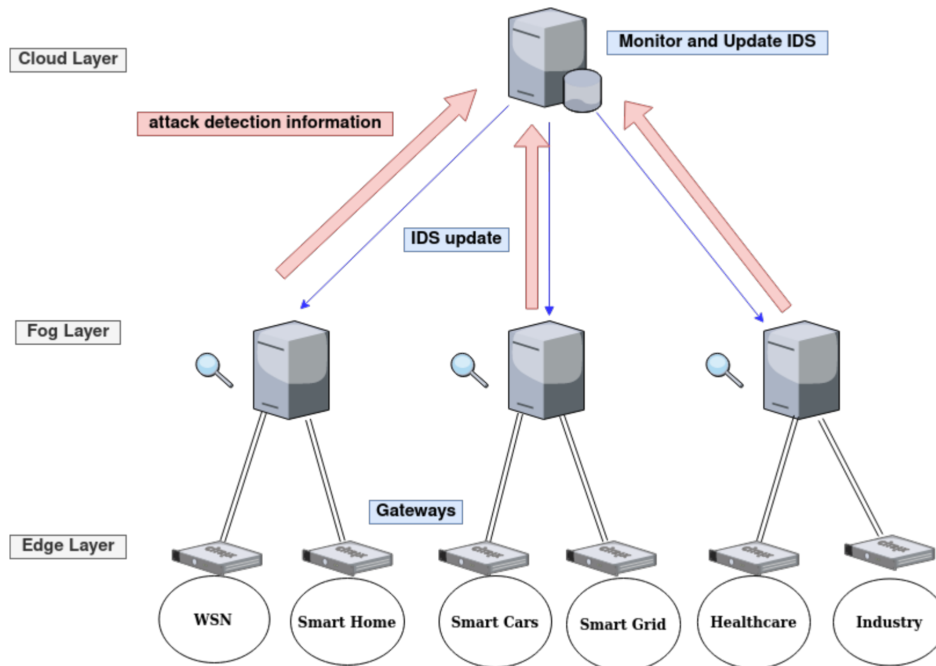
Unlike secondary model refit, retraining the primary model comprises some elements of initial model training and deployment, including multiple epochs, parameters tuning, with train/test split and early stopping mechanisms to optimize model performance without overfitting to the new data. Optimal performance metrics of precision, F1 score, false positive and false negative rates will be prioritized. Achieving greater performance metrics in training than those of the initial model will trigger model save and redeployment to gateways throughout the network.

4.5.2 Secondary Model Update

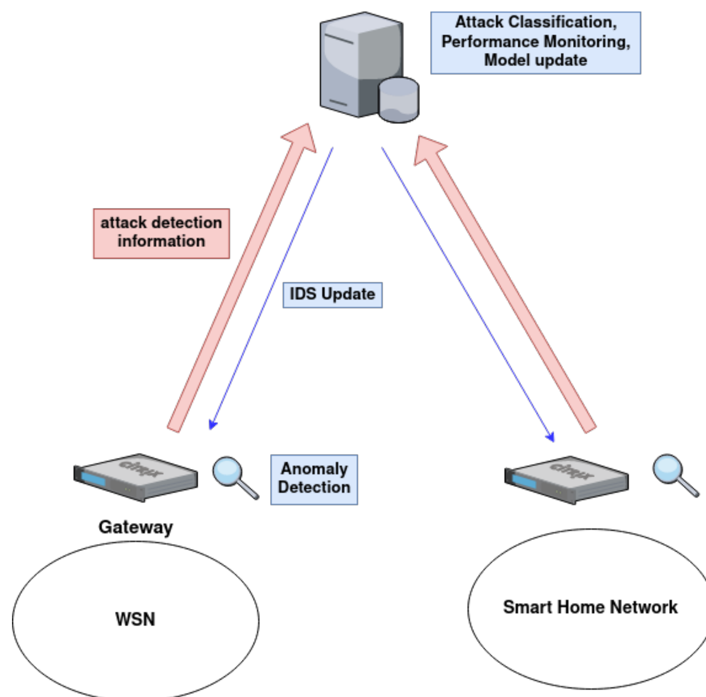
Secondary model update, shown in the third box of Figure 4.2, occurs intermittently at edge layer gateways to further normalize the benign traffic patterns of the local network. In secondary model update, the binary classifier is simply refit on saved labelled records at the gateway. The end result of performing this local refit is a model that is gradually customized at each gateway between primary model updates, making it more robust to traffic that does not align with the patterns established by local network devices.

Additional epochs and performance monitoring are avoided in order to prevent the retraining process from interfering with other intrusion detection operations on the resource-limited gateway device. An issue with this method of single-epoch refit is the potential for model performance to degrade. As such, the purpose of the second timeout interval, discussed in Section 4.3, is to allow a sufficient amount of labelled data to accumulate before refitting. A higher timeout interval value will result in more data accumulated, and more consistently high performance. However, should the model degrade regardless, a higher timeout value will also result in a longer period during which the model is performing sub-optimally. Additionally, performing a refit task over larger amounts of data may take significantly longer, utilizing time and resources that may be better allocated to other gateway operations. Thus a range of timeout interval values will be examined in

testing to compare their resulting time to fit, resources used, and average evaluation accuracy upon refit.



(a) Previous



(b) Proposed

Figure 4.1 IDS Architectures

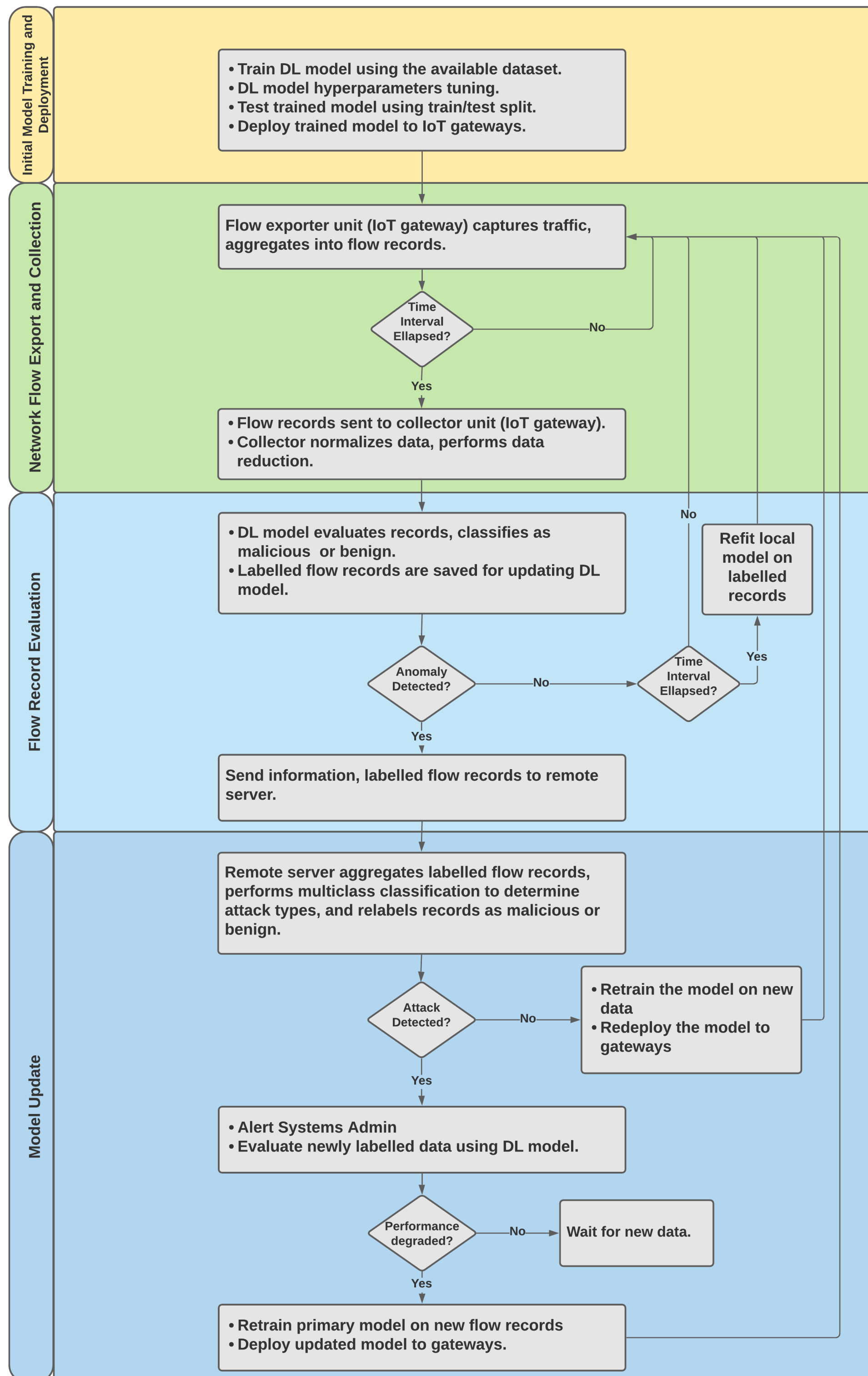


Figure 4.2 Proposed Framework Order of Operations

CHAPTER 5
TESTING

This chapter details the steps involved in the data preparation and testing processes. Section 5.1 discusses the devices of varying capacity used in testing. Section 5.2 covers the initial stages of data collection and preprocessing for future testing. Section 5.3 reviews the scaling and dimensionality reduction methods used to generate two distinct datasets and models for comparison. Section 5.4 details the methods by which initial neural network architectures were determined and resulting models were trained. Section 5.5 details the process of generating record timestamps for flow export interval simulation. Section 5.6 outlines the method by which performance metrics were collected in testing.

5.1 Devices Used

In order to examine the potential for gateway devices to implement deep learning network flow-based intrusion detection, the Raspberry Pi 4 was chosen for its ease of use, low price and commercial availability. Though the Raspberry Pi 4 with 8GB RAM borders the edge of what may currently be considered IoT, the 64-bit Raspbian OS, in beta at the time of writing, is critical to the operation of later versions of TensorFlow used in testing.

Fit and evaluate times and accuracy were also monitored on an an XPS15-9570 personal computer with Intel Core i7-8750 CPU 2.20GHz and 16GB using CPU-enabled TensorFlow, and the University of Tennessee Chattanooga's (UTC) SimCenter Firefly cluster, which consists of 4 compute nodes based on the Intel Xeon Gold 6148 CPU (each node equipped with four Nvidia

Tesla 32GB V100s) [33], in order to establish a baseline representing architectures more heavily reliant upon cloud and fog-based solutions for traffic evaluation. Additionally, UTC's Firefly cluster was used to expedite the processes of determining best neural network architecture for both models as well as the potential benefit of continuous single-epoch refit to model performance.

All programming for this thesis work was done in Python, employing Scikit-learn [34], TensorFlow [35], and Keras [36] libraries for model development and testing.

5.2 Data and Preprocessing

The data used for training and testing neural network models is drawn from the CICIDS 2018 Intrusion Detection Dataset, a series of CSVs converted from network traffic files collected over the course of a month [12]. Each file is composed of 79 distinct features and together account for 7 different attack scenarios including 15 distinctly labelled attack types across 15,831,819 samples [12]. This dataset was chosen for its consideration and inclusion of the critical characteristics of a complete and efficient intrusion detection dataset: diversity of attacks, anonymity, available protocols, complete network traffic, complete network interaction, feature set, labeled data samples, heterogeneity, and metadata [37]. Additionally, it includes a vast distribution of records among a variety of attack types, and its 79 features allow for a greater difference to be examined between standard and reduced-dimension datasets.

The processing of this data comprised the removal and subsequent addition of a number of features, random sampling of specific attack types, data normalization using min-max feature scaling, and dimensionality reduction via Principal Component Analysis.

It was determined by [8] that eight features of the CICIDS 2017 Dataset, which CICFlowmeter collects by default, are flags set to zero across the entire dataset and thus have no impact on model results. After confirming this consistency with CICIDS 2018, these features were removed. In addition, all categorical features from the original dataset were initially removed, as categorical

features are unfit for feature scaling and PCA reduction, and encoding these features would add time and processing to models, counterproductive to the goal of determining feasibility for embedded devices. However, after testing models trained with and without the "destination port" feature, it was found that models which incorporated destination ports into their training data achieved consistently higher accuracy, and were thus used exclusively in later testing.

Attack types for training and testing data were chosen for the number of samples included in the original dataset, so as to have a sufficient number of records of each attack type for training the model. Those including fewer than 10,000 records were removed. Of the 10 remaining attack types, 5,500 records were randomly sampled, distributing 1,000 records to training and the remainder to the testing dataset. Two exceptions include DoS-Goldeneye attack records, only 30,000 of which were included in testing, and DoS-Slowloris attack records, of which the 10,000 randomly sampled were distributed evenly across training and testing data. Intent behind sampling was to provide a generally even distribution of as many attack types as possible.

These data were then concatenated with 100,000 and 400,000 records of randomly sampled benign-labelled data to provide even distribution of benign and malicious data, emphasizing the importance of the accuracy metric in binary classification. The resulting training and testing datasets, consisting of 69 features and comprising 200,000 and 800,000 records, respectively, were relabelled '0' and '1', with '0' to indicate benign records and '1' for malicious.

5.3 Scaling and Dimensionality Reduction

Data normalization, and more specifically, min-max scaling, is a commonly-practiced step in preprocessing data for machine learning and PCA as the range of data can vary widely across features [10, 11]. Min-max feature scaling normalizes the data to a range of [0,1] by subtracting the value of the feature minimum from the current element, and dividing that number by the feature range (maximum value - minimum value). Features were scaled by this method prior to the

application of PCA, in which data was centered with a mean of zero. Figure 5.1 shows the explained variance ratio, or percentage of variance represented, for each principal component of the transformed 69-feature training dataset.

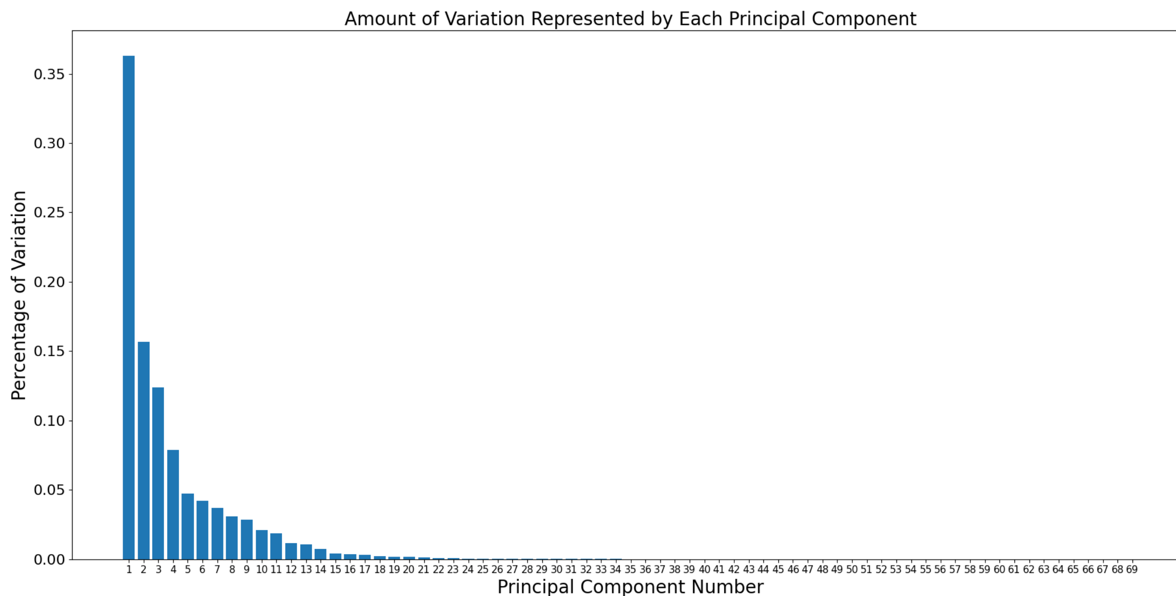


Figure 5.1 Amount of variation represented by each principal component of the 69-feature training data

The final dataset consists of 10 principal components, cumulatively accounting for approximately 93% of the variation in data. This number of principal components, previously shown to maintain high performance metrics for models trained on CICIDS 2017, was chosen to achieve a high Reduction Ratio of 10:69, while still accounting for a high percentage of variation in the input data [10, 11].

5.4 Model Architecture Determination and Training

For comparison of time to fit and resulting accuracy, two models were trained on variations of the 200,000-sample dataset. The first model training and testing data consists of the 69 features maintained from the original dataset and normalized using min-max feature scaling, and the second model, 10 principal components derived from the first variation. All models incorporate a sequential, feed-forward architecture, with an input layer matching the features of its training dataset, and an output layer of a single neuron. For the model trained on 69 features, Rectified Linear Unit (ReLU) was used as the activation function for its relative computational simplicity and resulting speed [38]. For the model trained on 10 principal components, hyberbolic tangent (tanh) activation function was used to ensure prediction consistency for negative input values [39]. The formulas for ReLU and and tanh activation are shown in Formula 5.1, and Formula 5.2 respectively, where x represents the input data and $f(x)$ the output at each neuron.

$$f(x) = max(x, 0) \tag{5.1}$$

$$f(x) = tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{5.2}$$

Number of hidden layers and number of neurons per hidden layer for final models were determined via an iterative program, using a train-test split ratio of 80% to 20% to train neural networks of varying architectures to a maximum of 10 epochs, with early stopping mechanisms prioritizing the maximum validation accuracy. The number of hidden layers and neurons per layer ranged from 1 to 6 and 5 to 100, respectively. Sequential feed-forward architecture, number of epochs, hidden layer and neuron ranges were chosen to expedite the process of determining architecture with highest validation accuracy, while limiting the size and complexity of the resulting model, so as to still be appropriate for embedded devices. The resulting models for 69 full and 10 PCA-reduced datasets, summarized in Figure 5.1, achieve initial validation accuracy of 95.40%

and 94.29%, and consist of five hidden layers each and 81 and 86 neurons per layer, respectively. Each row of the tables below represents a layer of the model. The first column of each table shows the name or label of the layer and its type. All layers for both models are of the 'dense' type, described in Chapter 2. The second column of each table shows the number of neurons that comprise each layer, 81 for hidden layers of the 69-feature model, and 86 for those of the PCA-reduced model. The final column of each table provides the number of trainable parameters (weights and biases) associated with that layer, corresponding to the number of input and output connections at each neuron of the layer.

Subsequent model creation and training incorporated these determined best architecture hidden layer and neuron numbers with a train-test split ratio of 80% to 20% on the same 200,000-sample training datasets. Maximum number of training epochs was set to 1,000, with early stopping mechanisms prioritizing model validation accuracy to avoid overfitting the input data. Resulting trained models for 69-feature and 10-principal component datasets achieved validation accuracy of 95.61% and 95.25%, respectively.

To accurately discern the affects of PCA reduction on model task completion time and system resources on the Raspberry Pi 4, a third model, trained on the 69-feature dataset and using the same architecture and tanh activation function of the PCA-reduced model was generated. This model achieved a validation accuracy of 95.88% in training.

5.5 Record Timestamp Generation

Simulation of network traffic influx for a network flow-based IDS demands the generation of timestamps for records to imitate the number of records captured between instances of refitting the model. Record timestamps were provided in the original dataset, but removed for the purpose of nomalizing features and deriving principal components. Additionally the concatenation of data spanning multiple days, and subsequent sampling and randomization of data for training and

Table 5.1 Resulting best neural network architectures for IDS models

Layer (type)	Output Shape	Param #
dense_1800 (Dense)	(None, 69)	4830
dense_1801 (Dense)	(None, 81)	5670
dense_1802 (Dense)	(None, 81)	6642
dense_1803 (Dense)	(None, 81)	6642
dense_1804 (Dense)	(None, 81)	6642
dense_1805 (Dense)	(None, 81)	6642
Output (Dense)	(None, 1)	82

Total params: 37,150
Trainable params: 37,150
Non-trainable params: 0
Accuracy: .9540

(a) Standard, 69-features model architecture

Layer (type)	Output Shape	Param #
dense_1830 (Dense)	(None, 10)	110
dense_1831 (Dense)	(None, 86)	946
dense_1832 (Dense)	(None, 86)	7482
dense_1833 (Dense)	(None, 86)	7482
dense_1834 (Dense)	(None, 86)	7482
dense_1835 (Dense)	(None, 86)	7482
Output (Dense)	(None, 1)	87

Total params: 31,071
Trainable params: 31,071
Non-trainable params: 0
Accuracy: .9429

(b) Reduced, 10-principal components model architecture

testing dataset generation would render the attempted reintroduction of the original timestamps meaningless. As such, new timestamps were generated for the 800,000-sample testing datasets to approximate the traffic influx during each of the 10 attacks included in the final testing data.

The authors of [12] have included attack start and end times in their description of CICIDS 2018. The number of seconds per attack were calculated and summed in instances of attack types spanning multiple distinct attacks. This number was then multiplied by the ratio of records of the corresponding attack type included in the final testing data to records of the same attack type included in the original concatenated CSV files, then divided by the number records of that attack type included in the final testing data. The resulting value was then incremented across its corresponding number of records in the final testing data. Table 5.2 shows the values calculated at each step of simulated timestamp generation. Four attacks types which included fewer than 10,000 records in the original training data, were excluded from sampling and timestamp generation.

As benign data is dispersed between attacks across all provided CSV files for CICIDS 2018, the rate of benign traffic was simulated using the purely benign traffic CSV file included in CICIDS 2017 by dividing the approximate number of seconds spanning the traffic capture by the number of included flow records. The final testing dataset timestamps span approximately 10.52 hours, beginning at 0 and ending at 37888.0763354195. Table 5.3 shows the average number of records, over which fit and evaluation tasks are executed for each interval value tested.

5.6 Model Evaluation and Fit Testing

The following subsections detail the processes by which comparable metrics were drawn in system and model performance testing. Changing variables to be compared include device, binary classifier (being either 69 feature or 10 principal component), batch size and interval value, and CPU stress. Metrics for variable comparison include fit and evaluation time, evaluation accuracy, CPU load percentage and memory. The final subsection outlines the method by which CPU stress

Table 5.2 Values for Simulated Timestamp Generation

Attack Type	# of Records	Samples in Training	Samples in Testing	% of Data Used	Attack Time (min)	Attack Time * % used (sec)	Time to Increment (sec)
ftp bruteforce	186186	10000	45000	0.24169	97	1406.66	0.03126
ssh bruteforce	180929	10000	45000	0.24872	90	1343.07	0.02985
DoS-goldeneye	40022	10000	30000	0.74959	43	1933.94	0.06446
DoS-slowloris	10583	5000	5000	0.47246	41	1162.24	0.23245
DoS-slowhttptest	135134	10000	45000	0.333	56	1118.89	0.02486
DoS-hulk	444706	10000	45000	0.10119	34	206.43	0.00459
Web-bruteforce	561						
Web-xss	218						
Web-sql	86						
bot	275435	10000	45000	0.16338	174	1705.67	0.0379
infiltration	155558	10000	45000	0.28928	385	6682.39	0.1485
DDoS LOIC-HT	554442	10000	45000	0.08116	65	316.53	0.00703
DDoS LOIC-UD	1629						
DDoS HOIC	660664	150000	50000	0.07568	60	272.45	0.00545
benign	13185666	100000	400000	0.03034			0.05435
totals	15831819	200000	800000				

Table 5.3 Average Records per Interval Value

Interval Period (seconds)	Average # of Records Fit/Evaluated
5	105.57
10	211.14
30	633.41
60	1265.82
120	2531.64
180	3791.46
240	5063.28
300	6299.2
600	12499.98

simulation was implemented in testing. In order to simulate various inactive and active threshold flow exporter settings, interval variables representing 5, 10, 15, 30, 60, 120, 180, 240, 300, and 600 seconds were tested. These same interval values were used in testing fit time and evaluation accuracy to simulate the second timeout interval of the proposed framework.

5.6.1 Fit and Evaluation Time

Trained models and final testing datasets were deployed to each of the three systems for testing, alongside a number of Python scripts to test model fit and evaluation times at various interval and batch sizes. Fit and evaluation times were approximated by calling the performance counter function of the Python Time library before and after the Keras fit or evaluate, subtracting the former value from the latter. The performance counter function returns the value, in fractional seconds, of a system's performance counter to measure a short duration [40]. Fit and evaluation times across both models and all interval and batch size values were drawn from each of the three devices for speed comparison.

In addition, the 69-feature model utilizing tanh activation was deployed to the Raspberry Pi 4, to compare fit and evaluation times to those of the PCA-reduced model.

5.6.2 Fit and Evaluation Accuracy

Accuracy of a machine learning model is calculated as the ratio of correctly predicted samples (True Positive + True Negative) to all predictions (True Positive + True Negative + False Positive + False Negative). In order to observe and quantify the performance trade-off of dimensionality reduction, fit and evaluation validation accuracy were calculated at each interval for all interval and batch size values. An issue to be observed in refitting locally, without a performance monitoring trigger or additional epochs prioritizing model performance, is the potential for accuracy to degrade upon refit. For this reason, both refit and evaluation accuracy of the initial model was monitored at every interval in order to compare their respective averages. Higher average refit accuracy would indicate performing local refit in this way has merit. To ensure consistency between model fit accuracy and validation accuracy upon fit, additional tests were run on the Firefly cluster, pulling evaluation accuracy from a static (unchanging) version of the initial model, alongside evaluation accuracy from before and after refitting a dynamic (changing) model, which was

saved upon refit. To prevent redundant refitting in these additional tests, 50 distinct but initially identical dynamic models were generated for testing each interval value and batch size, for both normalized and reduced datasets.

5.6.3 Device Resource Metrics

In order to observe and compare resource demands of both models at varying batch size and interval values for the Raspberry Pi 4, the ‘dstat’ application was run simultaneously with each Python script, collecting computational load and RAM at single-second intervals. ‘Dstat’ is a command-line application for monitoring system resource statistics [41]. CPU usage was calculated by combining user and system usage percentages for each core. System usage represents the amount of CPU time used by the kernel for low-level processes. User usage percentage represents the time used by higher-level processes in the user space, such as applications. To establish a baseline value for this method of load calculation, ‘dstat’ was run on the Raspberry Pi 4 over a 60-second period while the system was otherwise idle. Average CPU utilization over this period was 3.212862069% (of 400). Over the same idle period average memory utilization was established at 274.538389 MB.

5.6.4 Stress Simulation

Determining the feasibility of embedded devices as deep learning intrusion detection-enabled flow collector units demands the comparison of fit and evaluation speed and performance metrics as additional load or stress is placed on the device. To this end, ‘stress-ng’, a terminal-based application for stress testing computer systems [42], was employed to run simultaneously during additional testing for every Python script. ‘Stress-ng’ was enabled to run on all 4 cores of the Rasperry Pi’s ARM Cortex-A72 processor simultaneously to maximize CPU load at 400%. It

is expected that stress testing will show a significant increase in fit and evaluation times, where any changes in model fit and evaluation accuracy will remain statistically insignificant. Nevertheless, all resource and model performance metrics were drawn for comparison to the initial tests.

CHAPTER 6

RESULTS AND DISCUSSION

Chapter 6 presents an analysis and discussion of experiment results. Section 6.1 provides an overview of processed results and methodology for data interpretation. Section 6.2 examines the affect of batch size and PCA dimensionality reduction on device memory and CPU utilization. Section 6.3, presents an analysis of the accuracy of static and dynamic models to ascertain the potential benefit of local model refitting in the proposed framework. Section 6.4 presents a summary and discussion of the results.

6.1 Results Processing and Comparison

Table 6.1 Raspberry Pi batch-interval profiles for results analysis

Features	Interval	Batch Size	Avg # Records	Avg Eval Time	Avg Eval RAM	Avg Eval Load	Avg Fit Time	Avg Fit RAM	Avg Fit Load
69	5	16	105.57	0.18	1052.28	102.78	0.20275	1078.89	104.41
69	5	32	105.57	0.1718	1047.89	102.17	0.18481	1075.91	102.99
69	5	64	105.57	0.17061	1051.1	102.19	0.17947	1077.92	103.08
69	5	128	105.57	0.16614	1055.62	102.46	0.17584	1076.99	102.58
69	5	256	105.57	0.16655	1054.16	102.43	0.17448	1078.2	102.5
69	10	16	211.14	0.19583	1050.67	102.82	0.23898	1043.04	105.42

Resource metrics were averaged alongside number of records collected at each interval, as well as fit and evaluation times, to provide a comparable profile for each batch and interval size for each model on the Raspberry Pi 4. Examples of such record profiles are provided in Table 6.1. In addition, batch-interval profiles including fit and evaluation time and accuracy averages were generated for each interval value and batch size on the XPS15 and Firefly cluster. Upon discovery of a testing error, a number of batch sizes for 15-second intervals, all results pertaining to this interval value were removed. Table 6.2 presents how comparison of these finalized data provide insight into initial research questions.

Table 6.2 Methods for results analysis

Question to be resolved	Metric for comparison
How does the Pi 4 compare to the XPS15, firefly in terms of evaluation speed, fit speed?	Compare average evaluation and fit time across all models, devices.
What is the trade-off of PCA, in terms of model performance?	Compare average evaluation time, accuracy, fit time across all models, devices.
What is the trade-off of PCA, in terms of device resources?	Compare average RAM usage, CPU load during evaluation and fit across all models on Pi 4.
How does batch size affect static and dynamic model accuracy?	Compare average static and dynamic model accuracy across all models, batch sizes.
What is the trade-off of batch size, in terms of device resources?	Compare average RAM usage, CPU load during evaluation and fit across all batch sizes on Pi 4.

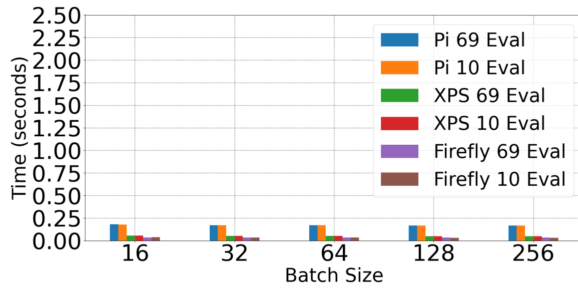
6.2 Device Average Task Speed Comparison and Analysis

Aside from the obvious significant reductions and increases in task completion time in relation to device capacity and amount of data processed, respectively, a number of trends are

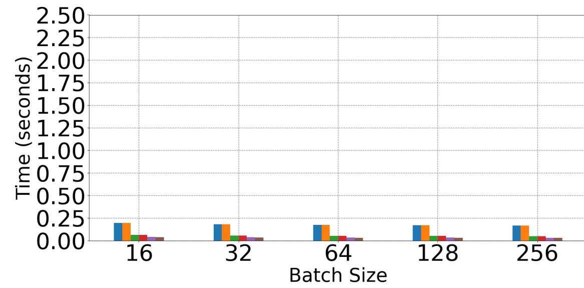
observed from the results data that conform to the understanding of DL evaluation and training operation established in Chapter 2. First, in comparing time values across Figure 6.1 through Figure 6.8, we find that greater deviation in task completion time exist between batch sizes at higher interval values. The simple explanation is that increases in the amount of data to be processed places greater significance on the amount of data processed in a single step. Second, fit times are consistently higher than evaluate times and greater deviation exists between batch sizes. This is also expected as the training process requires observation and adjustment of neural network parameters. Additionally, higher batch sizes mean fewer instances of parameter adjustment across a single epoch.

The effect of Principal Component Analysis on task completion time is minimal for both fitting and evaluating models. In fact, time-to-complete for the reduced dataset is increased in some instances of evaluation and fitting. A likely explanation for the minimal affect of PCA is the tanh activation function utilized in the reduced model. The increased computation complexity of the tanh function over ReLU is compounded across each neuron. Figure 6.5 and Figure 6.6 confirm this explanation, as the 69-feature ReLU and tanh models are compared, with time-to-evaluate consistently higher for the tanh model.

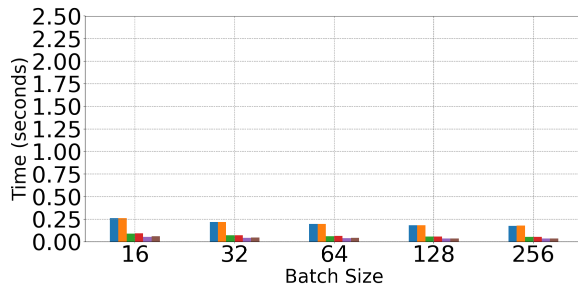
Time-to-fit is more consistent with expectations, as distinctions between reduced and standard models are slightly more pronounced, and reduced model fitting is consistently faster at lower batch sizes. This is to be expected as PCA is shown to reduce model training time [9, 11]. However, as batch size increases, so to does time-to-fit for the reduced model in relation to that of the standard model, surpassing it in some instances. Further, Figure 6.7 and Figure 6.8 show higher fit times for the 69-feature ReLU model than those of the tanh at batch size 16. At this batch size, only 16 records of the testing data interval are fed through the model before adjusting parameters, making the number of trainable parameters significantly more important to fit time.



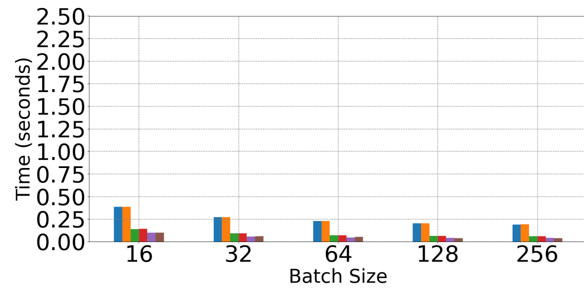
(a) 5 Seconds



(b) 10 Seconds



(c) 30 Seconds

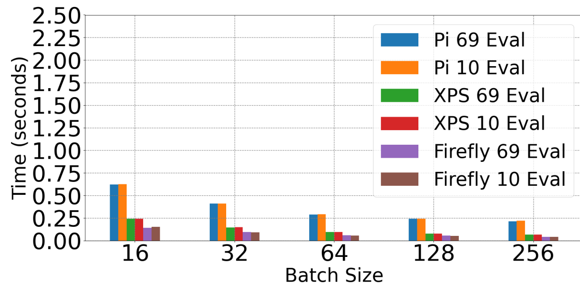


(d) 60 Seconds

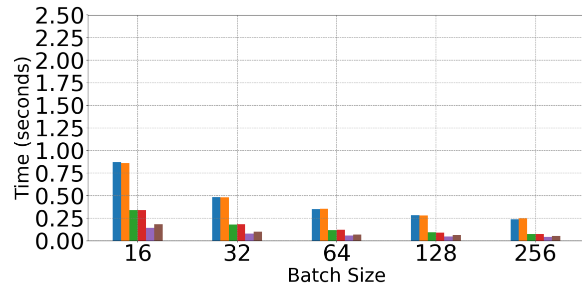
Figure 6.1 Evaluation task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation)

6.3 Effect of PCA, Batch Size on Embedded System Resources

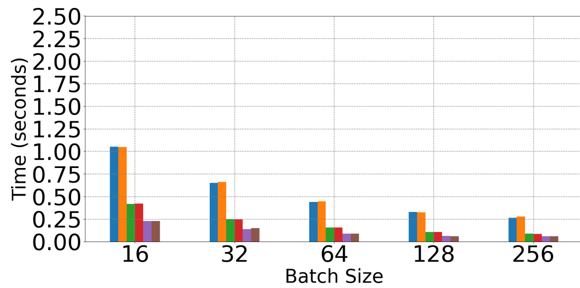
Further insight into possible explanations for the limited impact of PCA on fit and evaluation speed may be gained through observation of device resources during task execution. Section 6.3.1 and Section 6.3.2 examine the relationships of model reduction, batch size and interval values, to system memory and CPU usage, respectively. Section 6.3.3 discusses observations made from comparison of normal execution metrics to those monitored under stress.



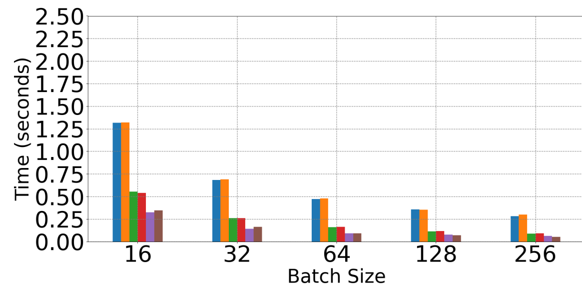
(a) 120 Seconds



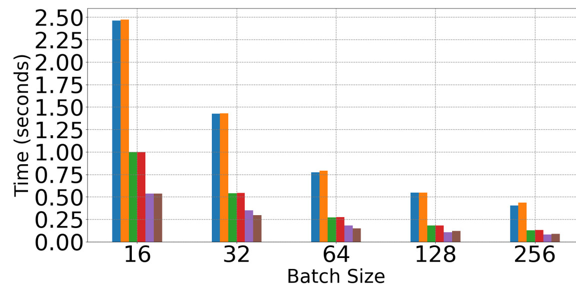
(b) 180 Seconds



(c) 240 Seconds



(d) 300 Seconds

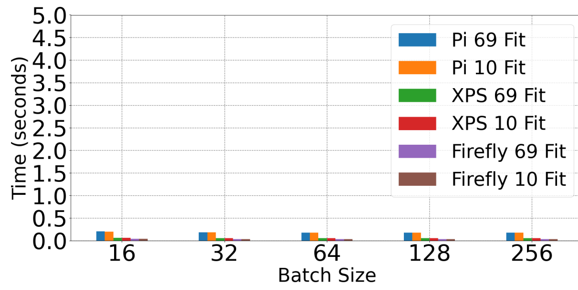


(e) 600 Seconds

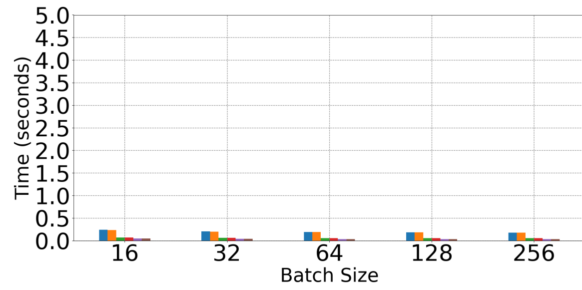
Figure 6.2 Evaluation task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation)

6.3.1 Memory

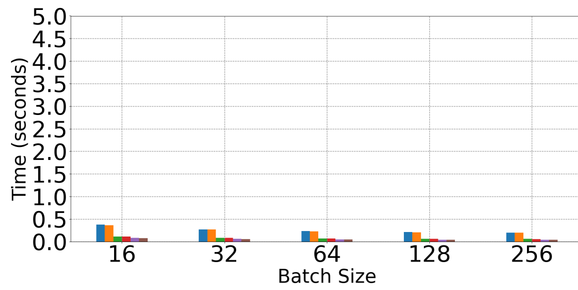
As is shown in Figure 6.9, Figure 6.12, PCA significantly reduces fit and evaluation RAM usage at a consistent percentage. Where standard model usage maintains above one 1,000 MB



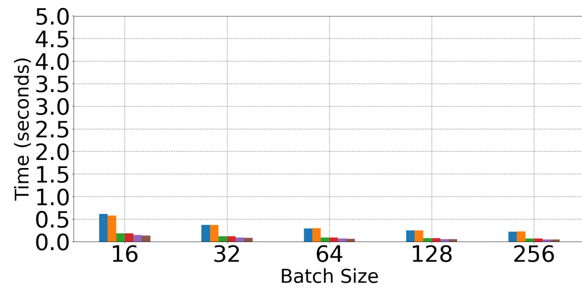
(a) 5 Seconds



(b) 10 Seconds



(c) 30 Seconds

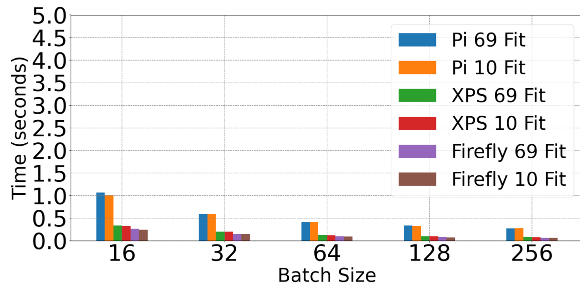


(d) 60 Seconds

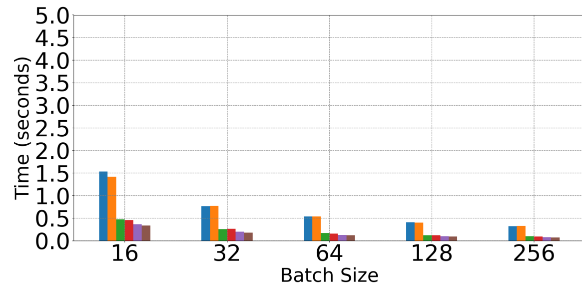
Figure 6.3 Fit task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation)

of RAM across lower interval values of 300 and 600 seconds, just under at higher intervals, the reduced model usage stays approximately 600 MB throughout. Considering the established idle of 275 MB alongside the greater range of signed values in the reduced dataset, this ratio is compatible with expectations.

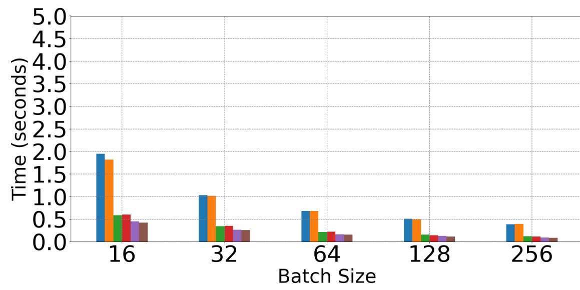
Figure 6.13 and Figure 6.14 compare the RAM usage of fit and evaluation tasks for 69-feature ReLU and tanh models. Here it can be seen that tahn model RAM is slightly higher for fit and evaluation tasks, a result of differing architectures. The tanh model includes an additional 5 neurons per hidden layer and with these, the associated trainable parameters. In other words, there are simply more values to store and update for the tanh model. It is also significant to note that RAM usage appears to trend slightly lower at much higher interval sizes. Interval value does



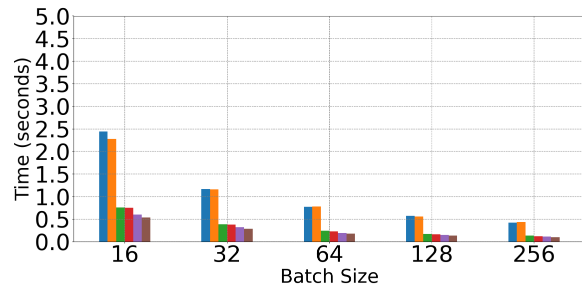
(a) 120 Seconds



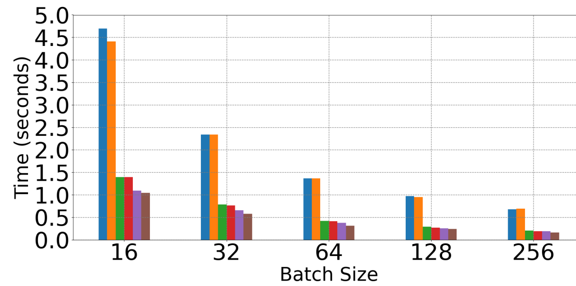
(b) 180 Seconds



(c) 240 Seconds



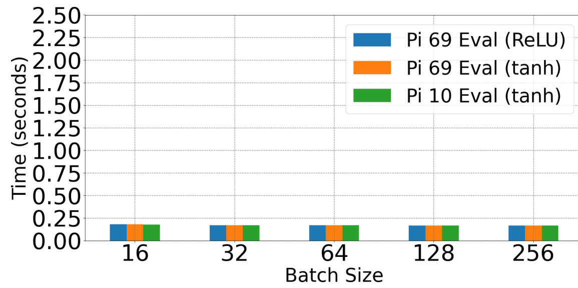
(d) 300 Seconds



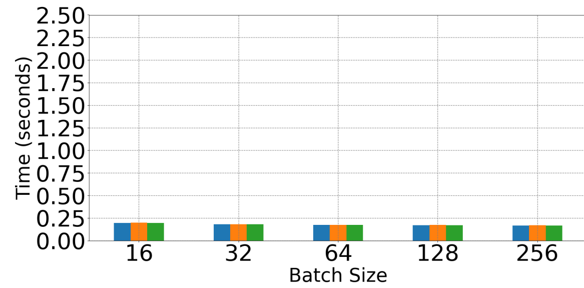
(e) 600 Seconds

Figure 6.4 Fit task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation)

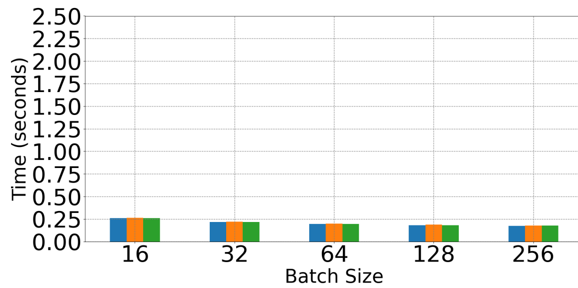
not increase the amount of data processed simultaneously (represented by batch size), but rather the number of times simultaneous data processing must occur, depending on the batch size. In other words, the number of records in the interval period, divided by the batch size, is roughly



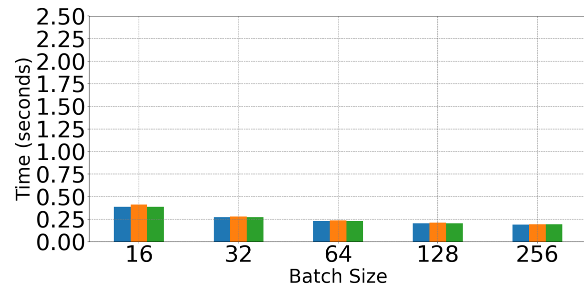
(a) 5 Seconds



(b) 10 Seconds



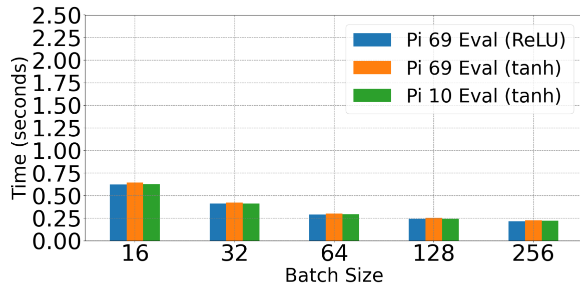
(c) 30 Seconds



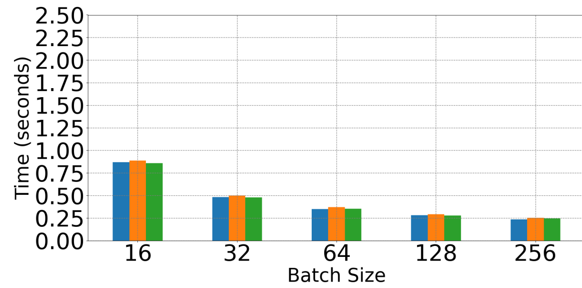
(d) 60 Seconds

Figure 6.5 Evaluation task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 on the Raspberry Pi 4 (models include 69-feature ReLU activation, 69-feature tanh activation, and 10-feature tanh activation)

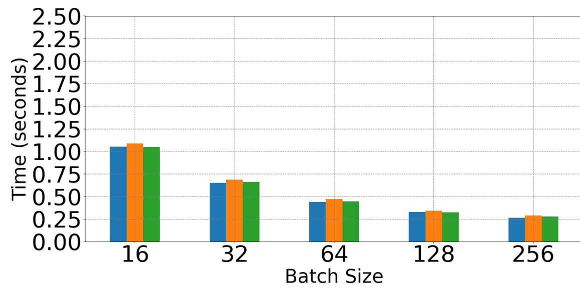
equivalent to the number of times a data processing task occurs over a single epoch. Thus, batch size, as mentioned in Chapter 2, should be a greater contributing factor in determining the feasibility of performing such tasks on lower-memory systems. Even so, Figure 6.9 through Figure 6.12 show that the deviation of memory usage across batch sizes is not significant for either model, as deviation across batch sizes is in the order of tens of MB.



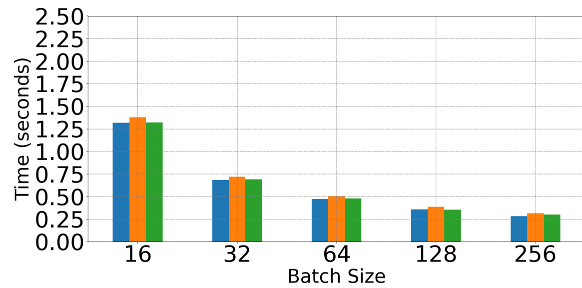
(a) 120 Seconds



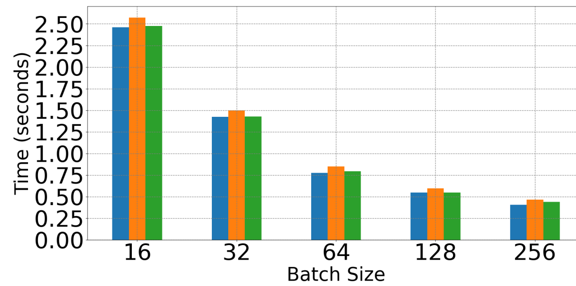
(b) 180 Seconds



(c) 240 Seconds



(d) 300 Seconds

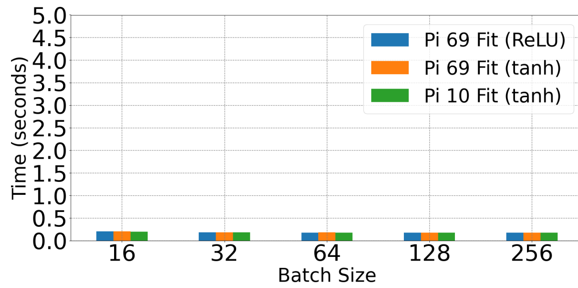


(e) 600 Seconds

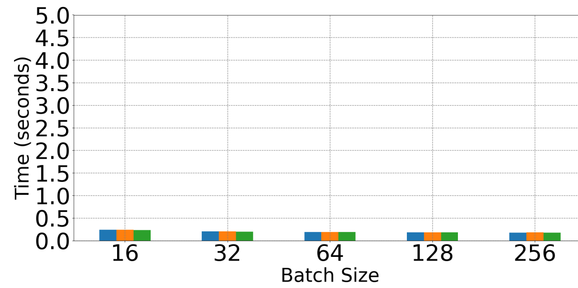
Figure 6.6 Evaluation task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 on the Raspberry Pi 4 (models include 69-feature ReLU activation, 69-feature tanh activation, and 10-feature tanh activation)

6.3.2 CPU Load

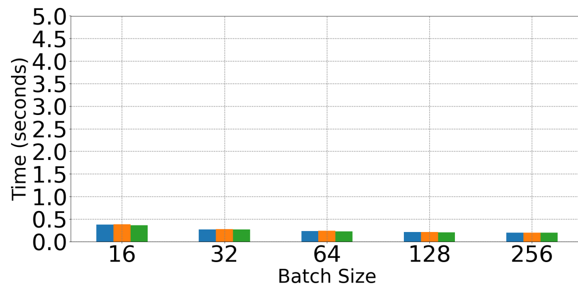
In examining computational load values presented in Figure 6.15 and Figure 6.16, a number of trends emerge in relation to model dimensionality, task performed, batch size and interval



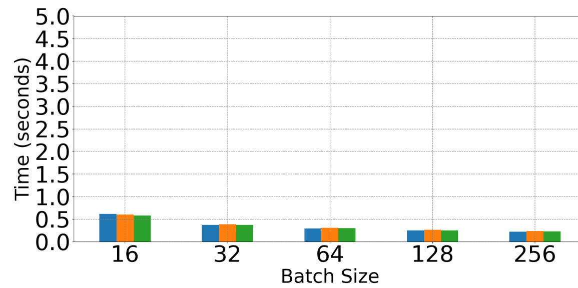
(a) 5 Seconds



(b) 10 Seconds



(c) 30 Seconds

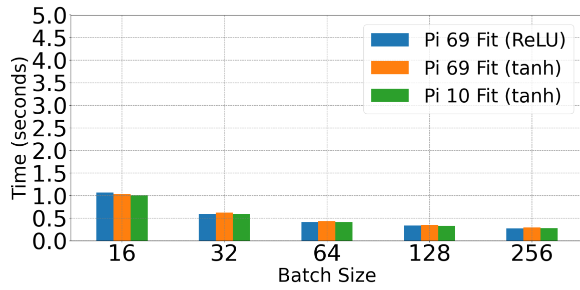


(d) 60 Seconds

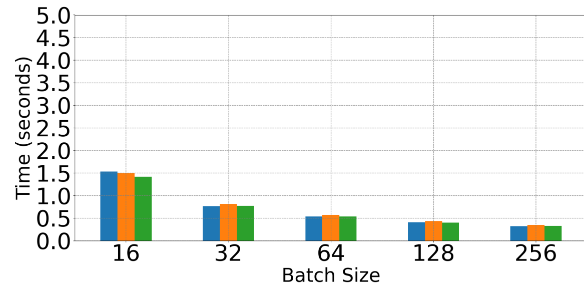
Figure 6.7 Fit task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 on the Raspberry Pi 4 (models include 69-feature ReLU activation, 69-feature tanh activation, and 10-feature tanh activation)

values. The first trend observed is that CPU usage is higher over periods of fitting than those of evaluation. The simple explanation relates to the computational cost of calculating and adjusting model parameters upon processing every batch, an aspect of training that is not present in model evaluation.

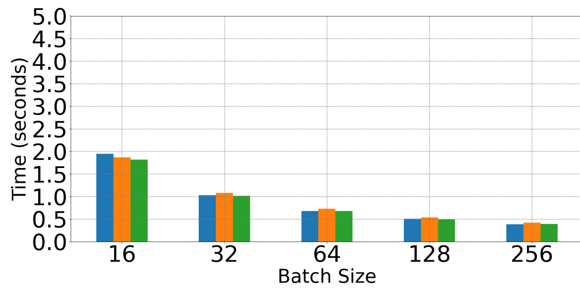
Another, more immediately apparent trend, is that CPU usage, while performing fit and evaluate tasks over the PCA reduced model, is consistently higher than that of the standard models, deviation between these values increasing at higher intervals. Two possible explanations relate to the differences in activation function and input data of these models: First, the primary contributing factor is resulting in higher CPU usage among tanh models is the increased computational complexity of the tanh activation function over ReLU. Multiplying this complexity increase across all



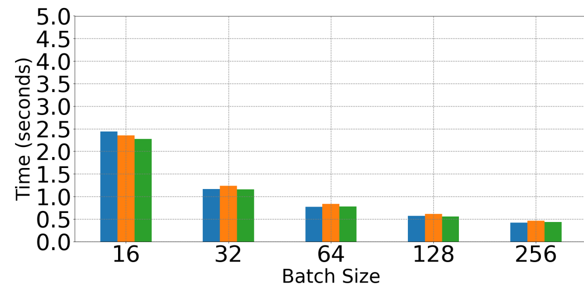
(a) 120 Seconds



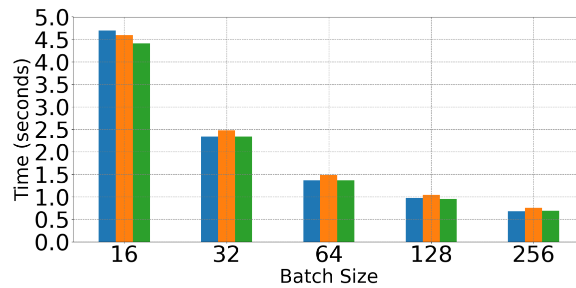
(b) 180 Seconds



(c) 240 Seconds



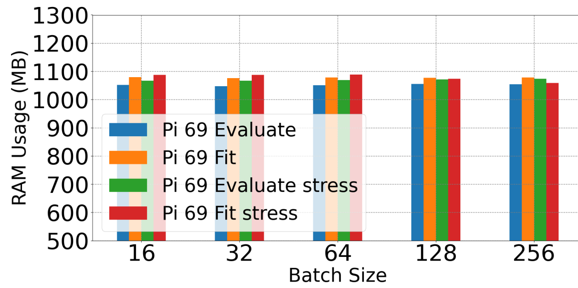
(d) 300 Seconds



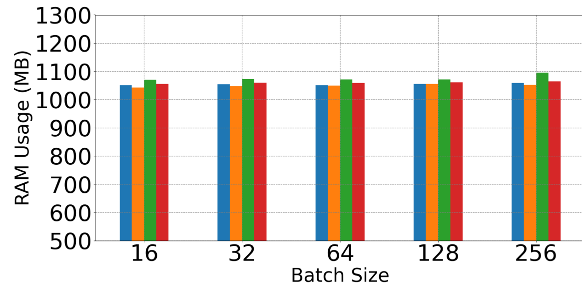
(e) 600 Seconds

Figure 6.8 Fit task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 on the Raspberry Pi 4 (models include 69-feature ReLU activation, 69-feature tanh activation, and 10-feature tanh activation)

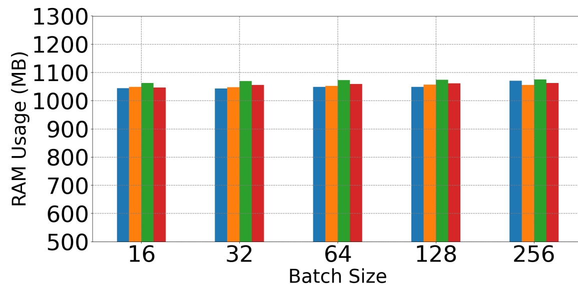
activation occurring over a batch of predictions increases CPU load. However, the PCA-reduced model maintains a higher load for fit and evaluation tasks than its 69-feature tanh counterpart. Aside from the changes in architecture resulting from a lower number of features, the distinction



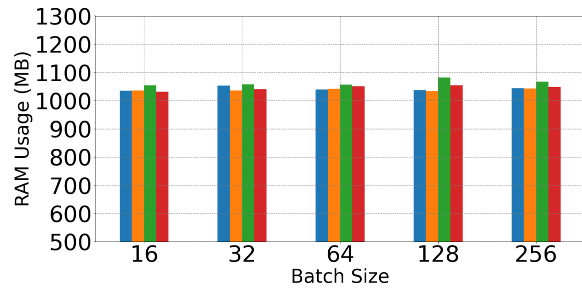
(a) 5 Seconds



(b) 10 Seconds



(c) 30 Seconds



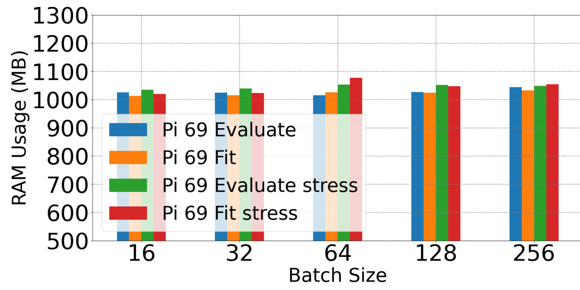
(d) 60 Seconds

Figure 6.9 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation alongside the same model running under stress)

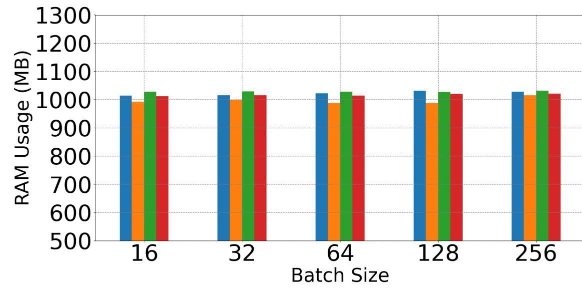
between these models is the nature of the input testing data. Where the values of the 69-feature input data range from 0 to 1 across all features, principal components evaluated by the reduced model have a higher range of both positive and negative values centered about a mean of 0. The computation of this larger range of signed values may add additional load to the model.

6.3.3 Stress Comparison

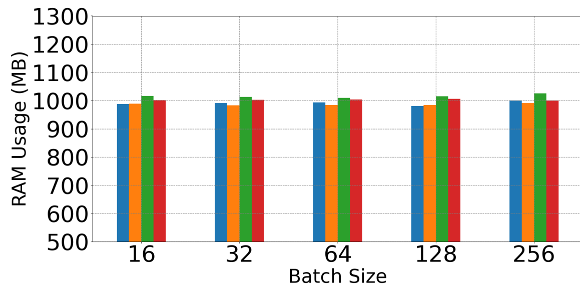
Though stressed device memory usage, represented in Figure 6.9 through Figure 6.12, is consistently higher than that of the device running normally, the margin of difference is in the order of tens of MB and may be explained by the memory cost of running the stress-ng application. CPU



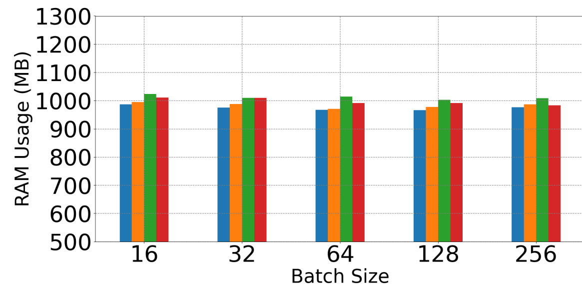
(a) 120 Seconds



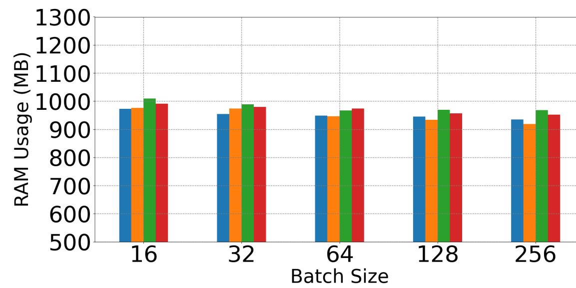
(b) 180 Seconds



(c) 240 Seconds



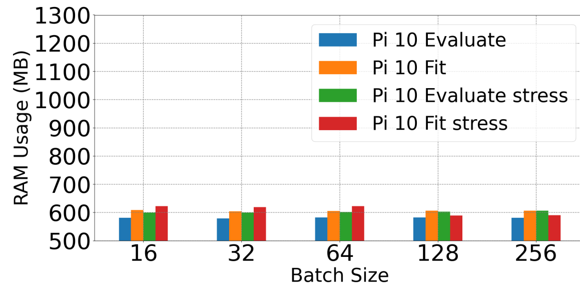
(d) 300 Seconds



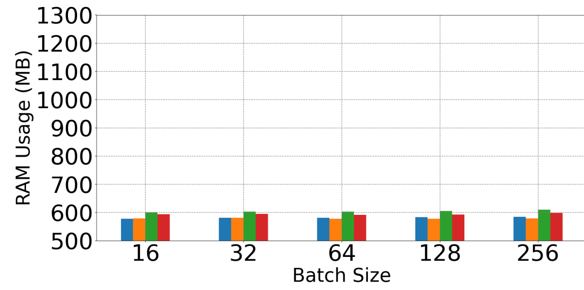
(e) 600 Seconds

Figure 6.10 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation alongside the same model running under stress)

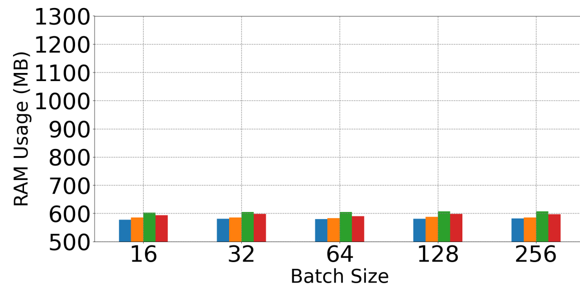
load for tasks under stress was not considered for analysis, as stress-ng seeks to maximize load across all cores for its duration. Thus little insight would be gained from comparison of this metric across models, tasks, batch and interval sizes under stress.



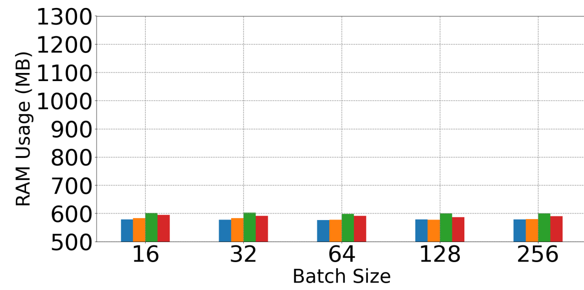
(a) 5 Seconds



(b) 10 Seconds



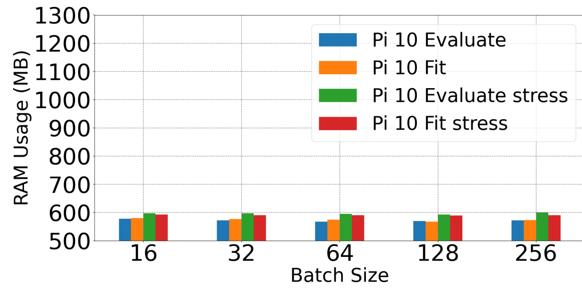
(c) 30 Seconds



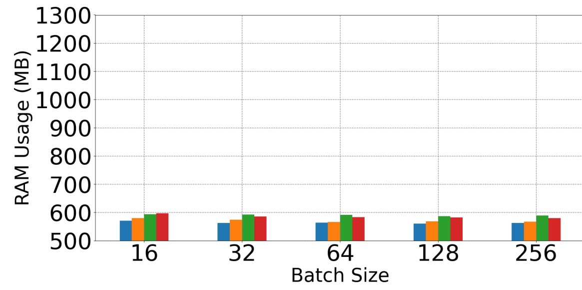
(d) 60 Seconds

Figure 6.11 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 10-feature tanh activation alongside the same model running under stress)

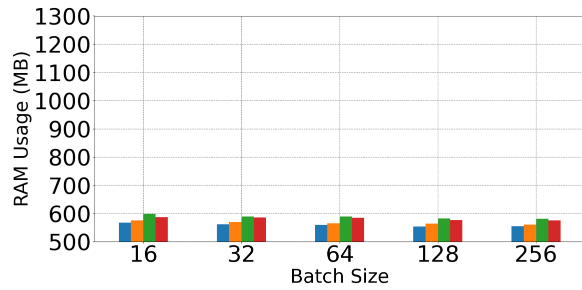
Figure 6.17 through Figure 6.20 compare the standard (ReLU) and reduced model evaluation and fit times while running the 'stress-ng' application. It is made clear in both figures that sharing the CPU with the 'stress-ng' application increases task completion time across all intervals and batch sizes, more than doubling it in some instances. Additionally, trends observed in the task speed comparison of Section 6.2 remain consistent for the device running under stress. The ratio of task-completion-time to time-interval is at its highest in fitting and evaluating 5 seconds of data at batch size 16. It is worth noting that both fit and evaluation at this interval and batch size are completed in under 1 second even under stress, meaning it is feasible for the Raspberry Pi 4 to perform evaluation at a five-second flow record expiration timeout setting, even under stress, at a batch size of 16.



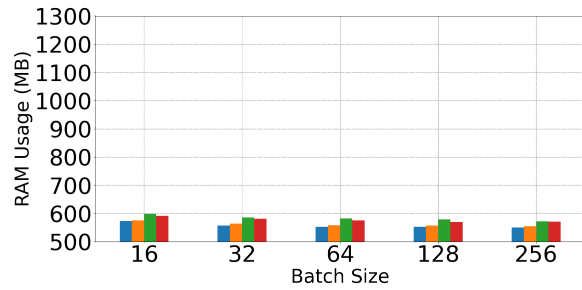
(a) 120 Seconds



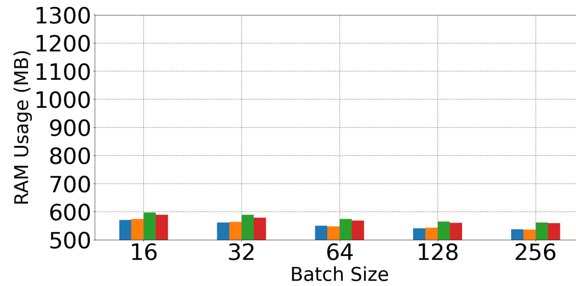
(b) 180 Seconds



(c) 240 Seconds



(d) 300 Seconds

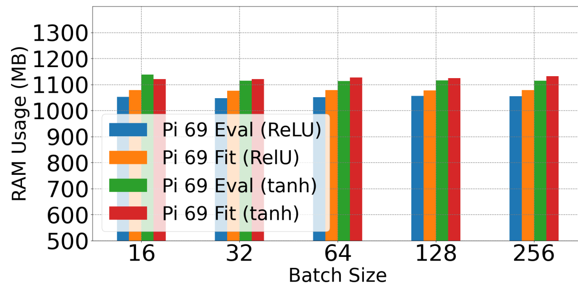


(e) 600 Seconds

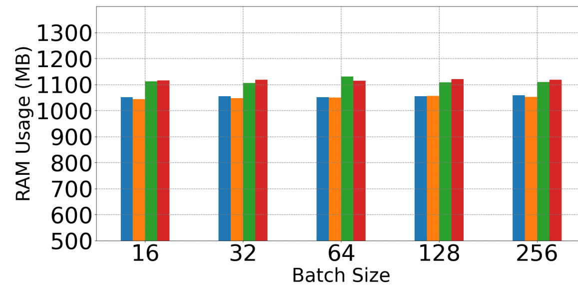
Figure 6.12 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 10-feature tanh activation alongside the same model running under stress)

6.4 Accuracy Analysis for Static and Dynamic Models

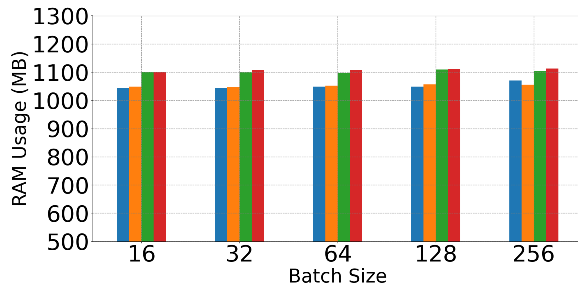
Figure 6.21 shows average validation accuracy achieved across 92 distinct models. Two static models, being the same described in Section 5.4 and used throughout testing, performed



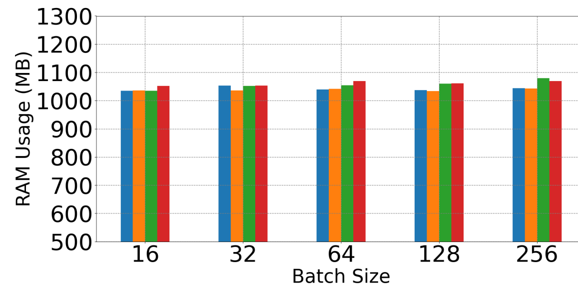
(a) 5 Seconds



(b) 10 Seconds



(c) 30 Seconds

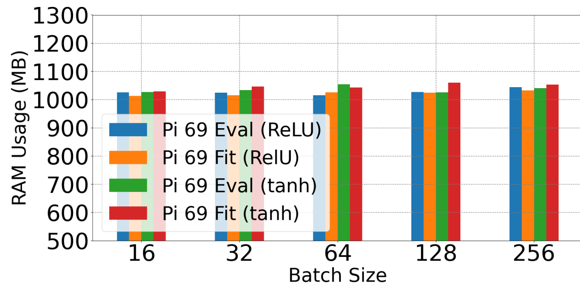


(d) 60 Seconds

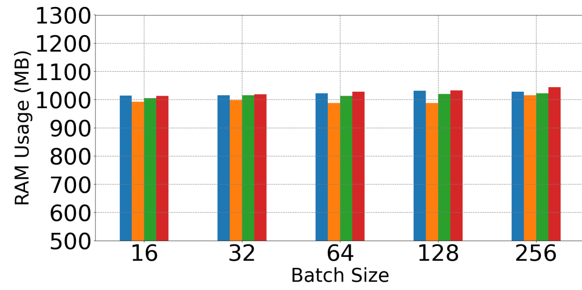
Figure 6.13 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation, 69-feature tanh activation)

evaluation across all represented batch size and interval values. The remaining 90, initially identical to static models, were refit and saved after each evaluation to simulate adaptation to continuous anomalous traffic. Distinct models for each interval-batch pair ensures no model retrains on the same data twice.

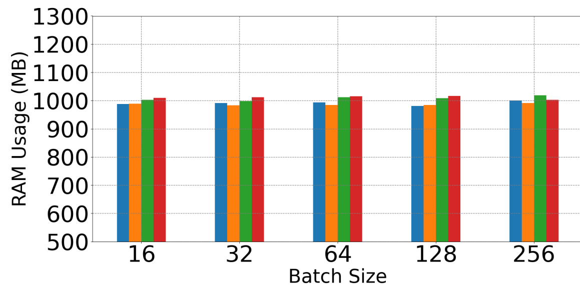
Results show static accuracy for both standard and reduced to be identical across all batch sizes of a given interval to be identical, as would be expected. Though batch size is a considerable parameter in training, no parameters are adjusted in evaluation. Additionally, standard deviation in accuracy across all interval values for both static models were in the order of 5 decimal places. Despite principal components representing a cumulative 93% of the variation in training data, static accuracy for the reduced model was considerably lower than that of the standard, so much as to



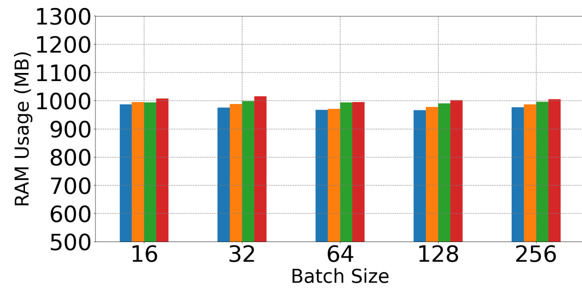
(a) 120 Seconds



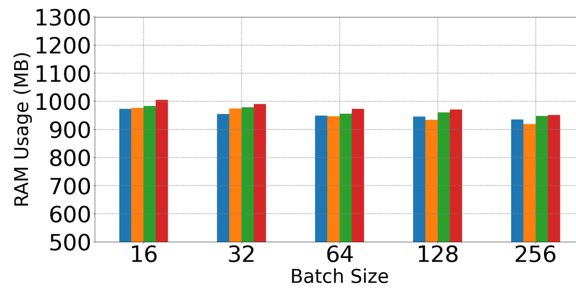
(b) 180 Seconds



(c) 240 Seconds



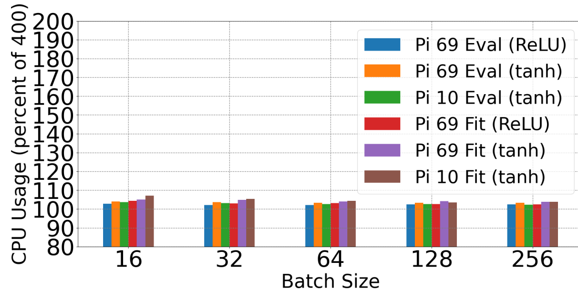
(d) 300 Seconds



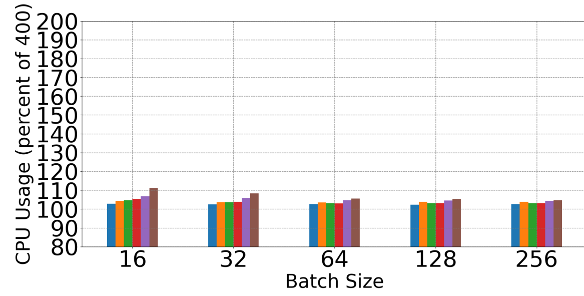
(e) 600 Seconds

Figure 6.14 Memory usage (in MB on Raspberry Pi 4) of fit and evaluation tasks for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation, 69-feature tanh activation)

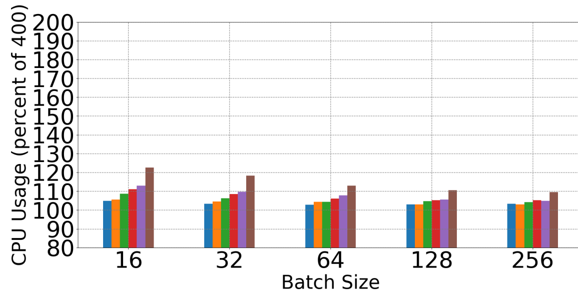
be impractical for detecting anomalies. It should be noted that models were trained on a minority (20%) of the overall data.



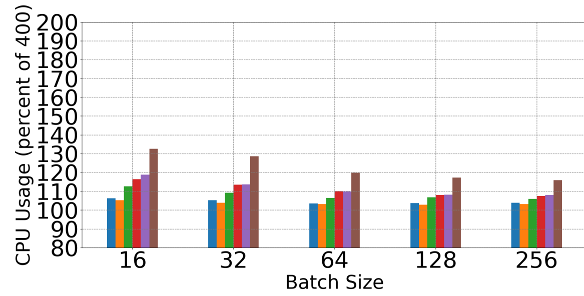
(a) 5 Seconds



(b) 10 Seconds



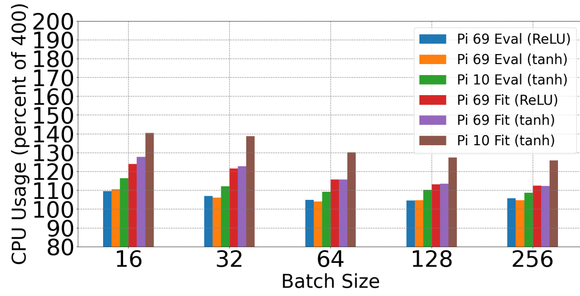
(c) 30 Seconds



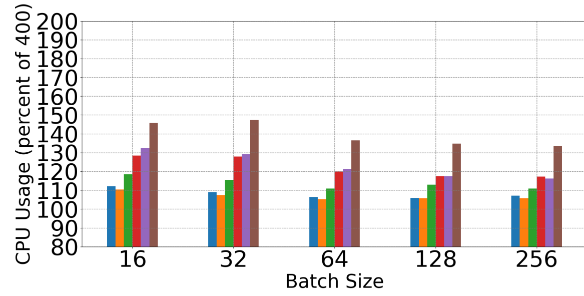
(d) 60 Seconds

Figure 6.15 CPU usage (in percent of 400 on 4 cores of Raspberry Pi 4) of fit and evaluation tasks for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation, 69-feature tanh activation, 10-feature tanh activation)

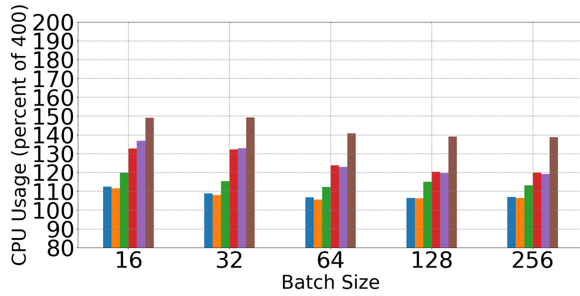
Further examination of accuracy averages, provided in Table 6.3, show that average accuracy of our 69-feature dynamic models steadily improves as batch size increases, surpassing average static model accuracy at 256 samples. This conforms to the understanding of batch size in model training provided in Chapter 2. Higher batch sizes represent a greater number of samples over which gradient descent is calculated before adjusting tensor weights, meaning faster convergence. Though average accuracy of the dynamic reduced models is less predictable in this regard, Table 6.3 show it to be higher than that of the standard static model across most batch sizes.



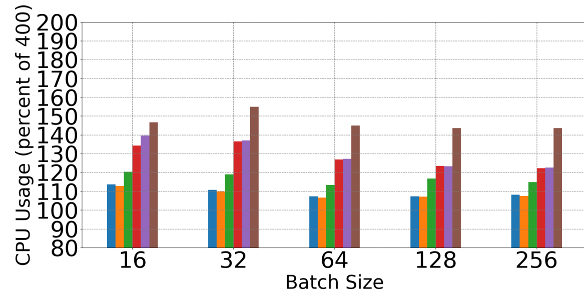
(a) 120 Seconds



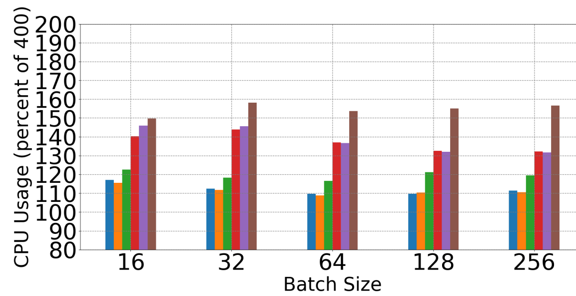
(b) 180 Seconds



(c) 240 Seconds



(d) 300 Seconds

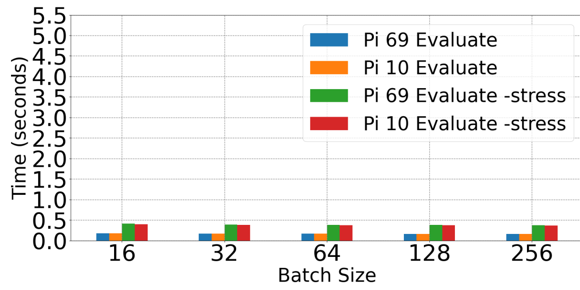


(e) 600 Seconds

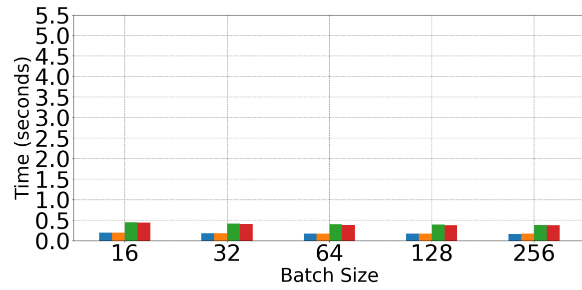
Figure 6.16 CPU usage (in percent of 400 on 4 cores of Raspberry Pi 4) of fit and evaluation tasks for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation, 69-feature tanh activation, 10-feature tanh activation)

6.5 Summary and Discussion

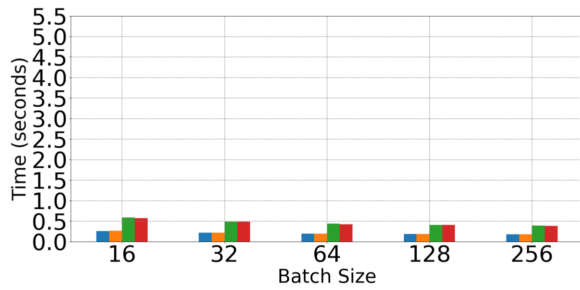
Results of testing indicate network edge DL-based anomaly detection via the proposed framework is feasible for embedded devices such as the Raspberry Pi 4. While fit and evaluate



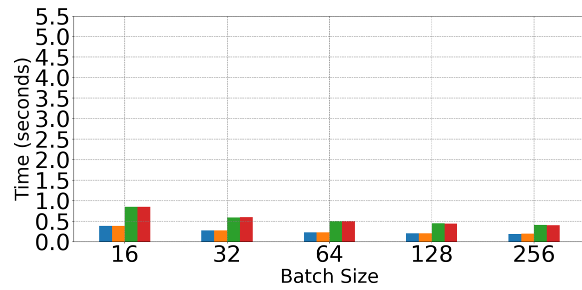
(a) 5 Seconds



(b) 10 Seconds



(c) 30 Seconds

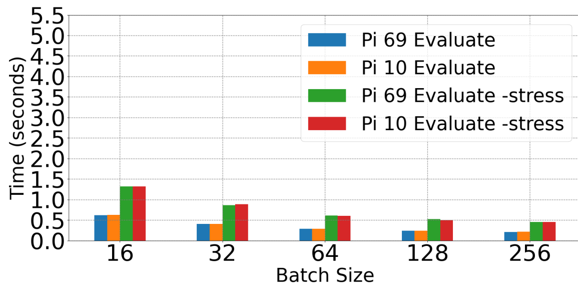


(d) 60 Seconds

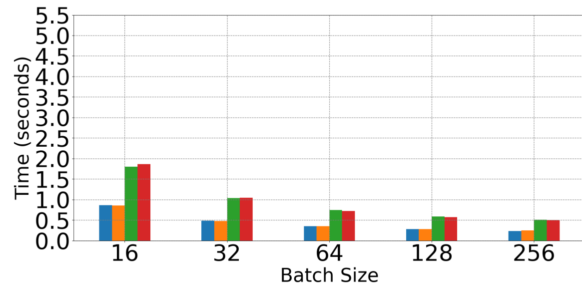
Figure 6.17 Evaluation task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation, alongside both models running under stress)

Table 6.3 Average static, dynamic accuracy for both models

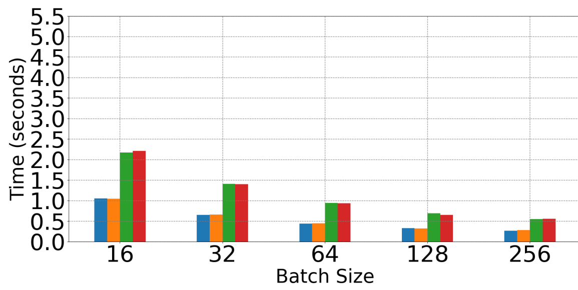
Batch Size	Static 69	Dynamic 69	Static 10	Dynamic 10
16		0.84079		0.91169
32		0.88867		0.91664
64	0.92226	0.91852	0.5659	0.92631
128		0.92274		0.92481
256		0.9309		0.92273



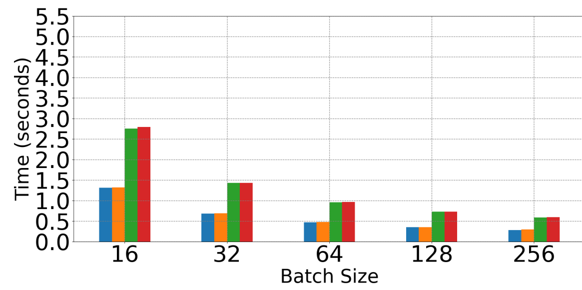
(a) 120 Seconds



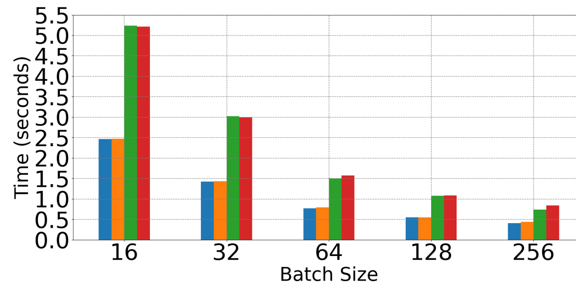
(b) 180 Seconds



(c) 240 Seconds



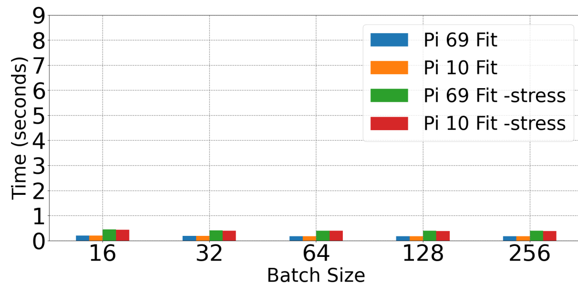
(d) 300 Seconds



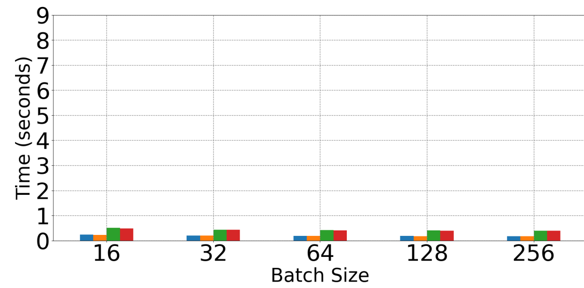
(e) 600 Seconds

Figure 6.18 Evaluation task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation, alongside both models running under stress)

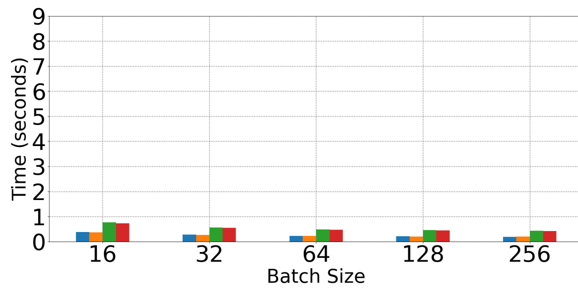
times are considerably higher than those of of the XPS15 and Firefly cluster, an average of only .2 seconds is required for the Raspberry Pi to retrain models over 5 seconds of aggregated data, representing the highest task-to-interval ratio of 4%. In addition, resource metrics collected show



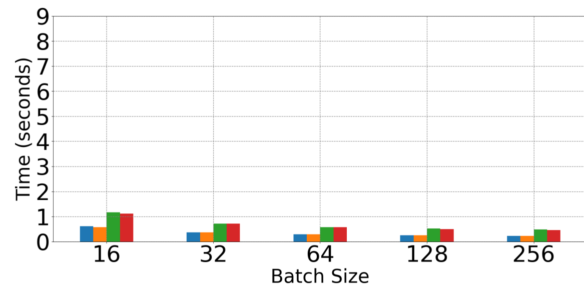
(a) 5 Seconds



(b) 10 Seconds



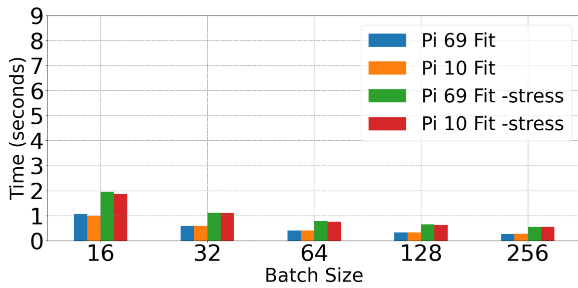
(c) 30 Seconds



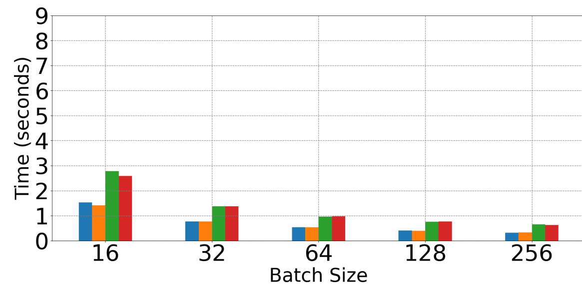
(d) 60 Seconds

Figure 6.19 Fit task completion time (in seconds) for 5, 10, 30, and 60-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation, alongside both models running under stress)

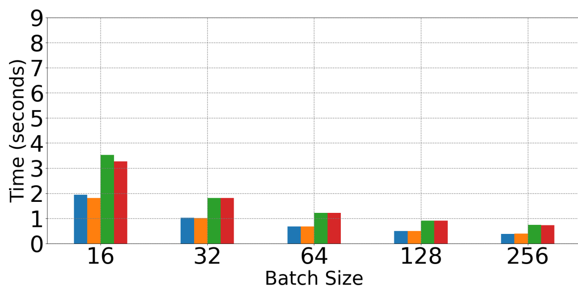
that though dimensionality reduction significantly reduces memory usage, it has negligible impact on processing requirements and resulting time required to process data. Rather, choice of neural network architecture and activation functions are of greater importance to these metrics. Finally performance testing of static and dynamic models show improved average accuracy from performing single epoch refit at higher batch sizes. As resource monitoring has indicated batch size has little impact on system memory usage, batch size of 256 is both achievable and preferable for edge network anomaly detection.



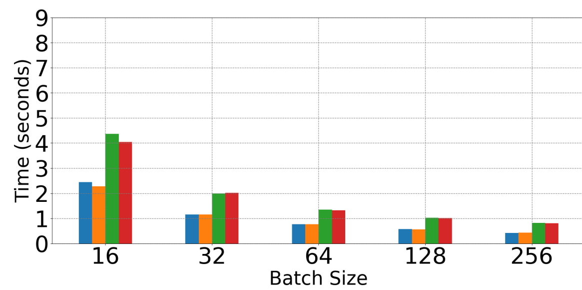
(a) 120 Seconds



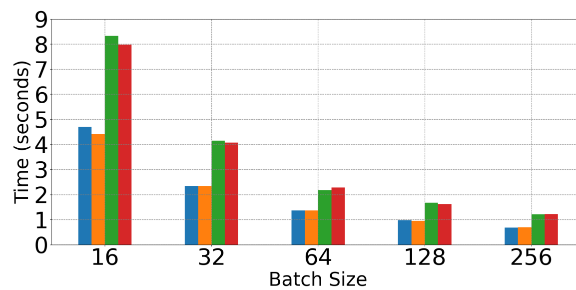
(b) 180 Seconds



(c) 240 Seconds



(d) 300 Seconds



(e) 600 Seconds

Figure 6.20 Fit task completion time (in seconds) for 120, 180, 240, 300, and 600-second intervals of testing data across batch sizes 16, 32, 64, 128, and 256 (models include 69-feature ReLU activation and 10-feature tanh activation, alongside both models running under stress)

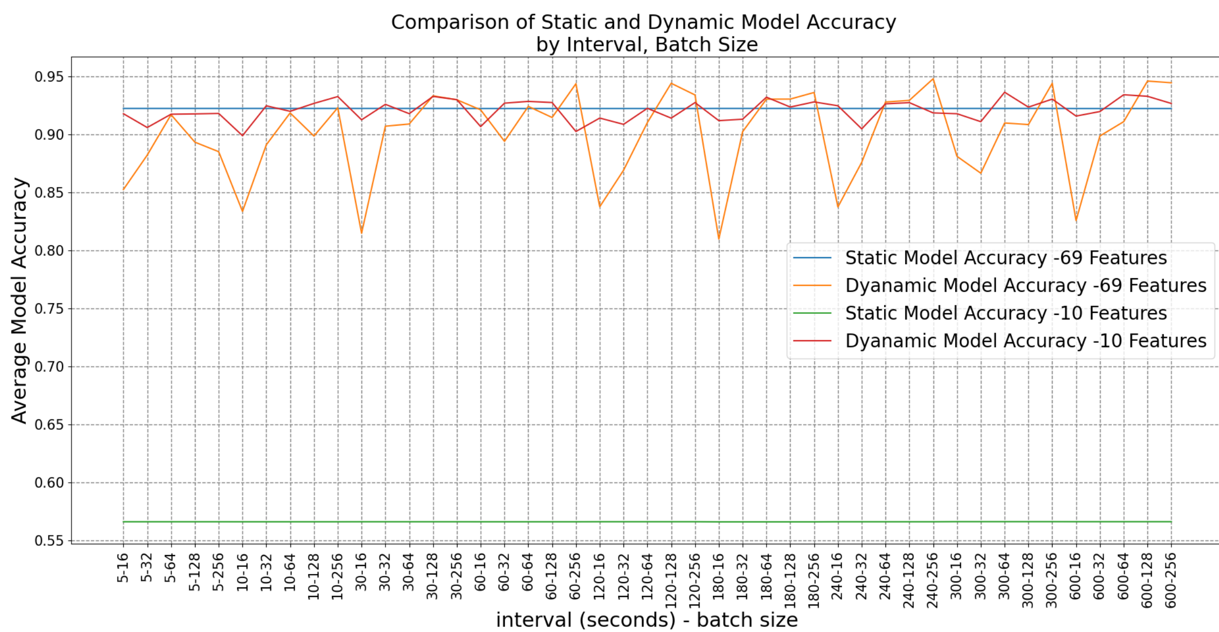


Figure 6.21 Comparison of static and dynamic model accuracy across all batch sizes

CHAPTER 7
CONCLUSION

7.1 Closing Thoughts

In this work, we have proposed a novel framework for implementing DL-based anomaly detection at the edge layer. It relies on embedded devices for performing network monitoring and classification and adapts to potential malicious traffic by retraining upon detection of anomalies. To determine the feasibility of the framework, we trained two neural network models of varying dimenionality for binary classification, and monitored model performance and resource demands on the Raspberry Pi 4. In addition, validation accuracy of both models was tested upon consistent refitting to various intervals of testing data. In the experiments conducted, the trade-off of PCA dimensionality reduction in terms of model accuracy and speed is often negligible, in some cases even detrimental, though accuracy upon refit was found to be more consistent than in other models. In terms of system resources, memory usage of the reduced model was roughly half of its counterpart for a dimensionality reduction ratio 10:69. Additionally, the highest batch size of 256 and higher timeout intervals for refitting are shown to be achievable and to facilitate consistently higher model accuracy upon refit. Further, refitting upon the lowest tested inactive record timeout value requires only 4% of the amount of time taken to accumulate data. Together, these results indicate that it is not only feasible of resource-limited devices to employ DL network flow-based anomaly detection effectively, but that anomaly detection models, implemented on such devices can adapt to network traffic in a timely manner.

7.2 Future Work

Though it has been shown that the Raspberry Pi 4 can utilize more complex DL models for anomaly detection effectively, further evaluation of the effect of PCA reduction on model and device resource metrics is necessary. Additionally, while testing has shown batch size to have insignificant impact on device memory usage, a deeper examination into its affect on paging, L1, L2 cache memory should be conducted. Future testing will emphasize identical neural network architecture, additional model performance metrics, and feature a range of dimensionality reduction techniques. Furthermore, though performing single-epoch refit can improve the accuracy of the current model, it is not clear that a higher-accuracy classifier would benefit. As the current ratio of fit-time to interval period permits, future testing will seek to determine a practical limit for multi-epoch training in real time.

REFERENCES

[1] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of ip flow-based intrusion detection," *IEEE Communications Surveys Tutorials*, vol. 12, no. 3, p. 343â356, 2010.

[2] A. Khraisat and A. Alazab, "A critical review of intrusion detection systems in the internet of things: techniques, deployment strategy, validation strategy, attacks, public datasets and challenges," *Cybersecurity*, vol. 4, no. 1, 2021.

[3] F. Kandah, J. Cancelleri, D. Reising, A. Altarawneh, and A. Skjellum, "A hardware-software codesign approach to identity, trust, and resilience for iot/cps at scale," in *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2019, pp. 1125–1134.

[4] M. F. Elrawy, A. I. Awad, and H. F. Hamed, "Intrusion detection systems for iot-based smart environments: a survey," *Journal of Cloud Computing*, vol. 7, no. 1, 2018.

[5] A. Samy, H. Yu, and H. Zhang, "Fog-based attack detection framework for internet of things using deep learning," *IEEE Access*, vol. 8, p. 74571â74585, 2020.

[6] J. Sicato, S. K. Singh, S. Rathore, and J. Park, "A comprehensive analyses of intrusion detection system for iot environment," *Journal of Information Processing Systems*, vol. 16, pp. 975–990, 09 2020.

[7] R. Maharaja, P. Iyer, and Z. Ye, "A hybrid fog-cloud approach for securing the internet of things," *Cluster Computing*, vol. 23, no. 2, p. 451â459, 2019.

[8] B. Reis, E. Maia, and I. PraÃ§a, "Selection and performance analysis of cicids2017 features importance," *Foundations and Practice of Security*, p. 56â71, 2020.

[9] M. G. Desai, Y. Shi, and K. Suo, "Iot bonet and network intrusion detection using dimensionality reduction and supervised machine learning," *2020 11th IEEE Annual Ubiquitous Computing, Electronics amp; Mobile Communication Conference (UEMCON)*, 2020.

[10] S. Zhao, W. Li, T. Zia, and A. Y. Zomaya, "A dimension reduction model and classifier for anomaly-based intrusion detection in internet of things," *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, 2017.

[11] R. Abdulhammed, H. Musafer, A. Alessa, M. Faezipour, and A. Abuzneid, "Features dimensionality reduction approaches for machine learning based network intrusion detection," *Electronics*, vol. 8, no. 3, p. 322, 2019.

[12] I. Sharafaldin, A. Habibi Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," *Proceedings of the 4th International Conference on Information Systems Security and Privacy*, 2018.

[13] Y. N. Soe, Y. Feng, P. I. Santosa, R. Hartanto, and K. Sakurai, "Machine learning-based iot-botnet attack detection with sequential architecture," *Sensors*, vol. 20, no. 16, p. 4372, 2020.

[14] S. Fenanir, F. Semchedine, and A. Baadache, "A machine learning-based lightweight intrusion detection system for the internet of things," *Revue dIntelligence Artificielle*, vol. 33, no. 3, p. 203â211, 2019.

[15] K. Filus, J. DomaÅska, and E. Gelenbe, "Random neural network for lightweight attack detection in the iot," *Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, p. 79â91, 2021.

[16] B. B. ZarpelÃ£o, R. S. Miani, C. T. Kawakani, and S. C. d. Alvarenga, "A survey of intrusion detection in internet of things," *Journal of Network and Computer Applications*, Feb 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804517300802

[17] D. SERPANOS, *INTERNET-OF-THINGS IOT SYSTEMS: architectures, algorithms, methodologies*. SPRINGER, 2019.

[18] K. Sha, T. A. Yang, W. Wei, and S. Davari, "A survey of edge computing-based designs for iot security," *Digital Communications and Networks*, vol. 6, no. 2, p. 195â202, 2020.

[19] L. S. Vailshery, "Number of iot devices 2015-2025," Nov 2016. [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[20] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC 12*, 2012.

[21] Li, Guo, Ma, Mao, and Guan, "Online workload allocation via fog-fog-cloud cooperation to reduce iot task service delay," *Sensors*, vol. 19, no. 18, p. 3830, 2019.

[22] "Rfc 7011." [Online]. Available: https://www.tech-invite.com/y70/tinv-ietf-rfc-7011.html

[23] C. Gambella, B. Ghaddar, and J. Naoum-Sawaya, "Optimization problems for machine learning: A survey," *European Journal of Operational Research*, vol. 290, no. 3, pp. 807–828, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037722172030758X

[24] A. Ghumro, A. Kanwal, I. Memon, and I. Simming, "A review of mitigation of attacks in iot using deep learning models," vol. 18, pp. 36–42, 01 2021.

[25] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, 2016.

[26] "sklearn.decomposition.pca." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[27] A. Verma and V. Ranga, "Machine learning based intrusion detection systems for iot applications," Nov 2019. [Online]. Available: https://link.springer.com/article/10.1007/s11277-019-06986-8

[28] M. Ring, S. Wunderlich, D. GrÃ¼dl, D. Landes, and A. Hotho, "Creation of flow-based data sets for intrusion detection," *Journal of Information Warfare*, vol. 16, pp. 40–53, 2017.

[29] N. Moustafa and J. Slay, "Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set)," in *2015 Military Communications and Information Systems Conference (MilCIS)*, 2015, pp. 1–6.

[30] "Nsl-kdd dataset." [Online]. Available: https://www.unb.ca/cic/datasets/nsl.html

[31] D. Aksu and M. Ali Aydin, "Detecting port scan attempts with comparative analysis of deep learning and support vector machine algorithms," *2018 International Congress on Big Data, Deep Learning and Fighting Cyber Terrorism (IBIGDELFT)*, 2018.

[32] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull, "Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset," *Future Generation Computer Systems*, vol. 100, p. 779â796, 2019.

[33] [Online]. Available: https://wiki.simcenter.utc.edu/doku.php/clusters:firefly

[34] "learn." [Online]. Available: https://scikit-learn.org/stable/

[35] "All symbols in tensorflow 2 : Tensorflow core v2.7.0." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/all_symbols

[36] "Module: tf.keras : Tensorflow core v2.7.0." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras

[37] A. Thakkar and R. Lohiya, "A review of the advancement in intrusion detection datasets," *Procedia Computer Science*, vol. 167, p. 636â645, 2020.

[38] Placzek, Stanislaw and Placzek, Aleksander, "Learning algorithm analysis for deep neural network with relu activation functions," *ITM Web Conf.*, vol. 19, p. 01009, 2018. [Online]. Available: https://doi.org/10.1051/itmconf/20181901009

[39] X. Wang, Y. Qin, Y. Wang, S. Xiang, and H. Chen, "Reltanh: An activation function with vanishing gradient resistance for sae-based dnns and its application to rotating machinery fault diagnosis," *Neurocomputing*, vol. 363, pp. 88–98, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231219309464

[40] “time - time access and conversions.” [Online]. Available: https://docs.python.org/3/library/time.html

[41] [Online]. Available: https://linux.die.net/man/1/dstat

[42] [Online]. Available: https://wiki.ubuntu.com/Kernel/Reference/stress-ng

VITA

Jonathan Hunter was born and raised in Little Rock, Arkansas. He graduated college from the University of Arkansas in 2014 with a Bachelor's Degree in English. After moving to Chattanooga in 2018, Jonathan entered the Master's program for Computer Science at the University of Tennessee, Chattanooga. While pursuing his Master's, Jonathan conducted research under Dr. Farah Kandah. His interests include cybersecurity, machine learning, and embedded systems. Jonathan graduated from UTC in May 2022 with a Master of Science Degree in Computer Science with an emphasis in Cybersecurity.