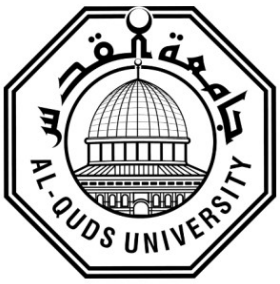
Deanship of Graduate Studies

Al-Quds University



Hybrid Anomaly Based Android Malware Detection Using Deep Neural Networks (HAMD-DNN)

Maher George Mousa Maria

M.Sc. Thesis

Jerusalem-Palestine

1445/2023

Hybrid Anomaly-Based Android Malware Detection Using Deep Neural Networks (HAMD-DNN)

By
Maher George Mousa Maria

B.Sc. in Computer Engineering, 2019,
Al-Quds University, East Jerusalem, Palestine

Supervisor: Dr. Rushdi A. Hamamreh

This Thesis was Submitted in Partial Fulfillment of the Requirements for the Master's Degree of Science in Electronics and Computer Engineering from the Faculty of Graduate Studies at Al-Quds University.

1445/2023

Al-Quds University

Deanship of Graduate Studies

Master of Electronics & Computer Engineering

Thesis Approval

Hybrid Anomaly-Based Android Malware Detection using Deep Neural Networks (HAMD-DNN)

Prepared By: Maher George Mousa Maria

Registration Number: 22012176

Master thesis submitted and accepted. Date: 29/08/2023
The names and signatures of the examining committee members are as follows:

Head of Committee:	**Dr. Rushdi Hamamreh**	**Signature:** ……..
Internal Examiner:	**Dr. Radwan Qasrawi**	**Signature:** RQ……..
External Examiner:	**Dr. Anas Melhem**	**Signature:** ……..

Jerusalem – Palestine

1445/2023

Dedication

I am dedicating this thesis to the soul of my paternal grandfather Mr. Mousa Maria, though he is no longer of this world, his memory continues to regulate my life. And, to my grandmother, whose love for me knew no bounds and who taught me the value of hard work.

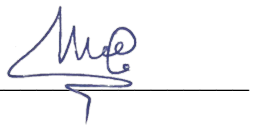
To my Parents, without whom none of my success would be possible. And, to my brother and sisters, for being a support system and the source of my motivation.

This thesis is a symbol of what we have accomplished as a family. This one is for US!

Declaration

I, Maher George Maria, declare that this thesis submitted for the master degree of Science in Electronics and Computer Engineering represents my own thesis in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated any idea/data/fact/source in my submission.

I also declare that this thesis or any part of it has not been submitted by me for the award of any degree, diploma title or recognition before.

Signed: 

Maher George Mousa Maria

Date: 29/08/2023

Acknowledgment

This thesis becomes a reality with the kind support and help of many individuals. I would like to acknowledge and give my warmest thanks to my supervisor Dr. Rushdi Hamamreh who made this work possible. His guidance and advice carried me through all the stages of writing my thesis.

I would also like to give special thanks for my family as a whole for their continuous support and understanding, when undertaking research and writing my thesis. Your prayer for me was what sustained me this far.

Finally, I would like to thank God, for letting me through all the difficulties. I have experienced your guidance day by day. You are the one who let me finish my degree. I will keep on trusting you for my future.

Abstract

A malware detection system for mobile devices contributes to the field of computer security. Cybersecurity is a major current problem mainly motivated by the growing number of malwares; data loss due to computer breaches cost a great loss. In addition ethical problems. Due to the popularity of smartphones and tablets, mobile devices are becoming the target of malware and cyberattacks. It is therefore essential to explore new ways to prevent, detect and counter cyberattacks. In these detection mechanisms, machine learning is used to create classifiers that determine whether an application is compromised. The advantage of a neural network is that it allows you to adapt to new situations. Therefore, we used this new technology to be able to identify types of malicious behavior and to be able to generalize it to future malicious programs.

The goal of this thesis is to propose a malware detection model on Android based on deep neural networks classification driven by sets of hybrid features. We reviewed and classified existing methods into two groups: the static methods which consist of examining the code of the mobile application and the dynamic methods which analyze the behavior of an application when it is running on a mobile terminal. Our goal is to use these two methods to take advantage of the both groups. To do this, we used the hybrid database "AMD" composed of 85 features. We are also conducted an experiment plan composed of hundreds of trainings in order to adjust the values of the hyperparameters improving the learning on this dataset as well as to select the most relevant remaining features, through this thesis, we work according to the most effective features from the AMD Dataset.

And to improve detection accuracy that have time-dependent frequencies such as attacks, three new input features (s_sessiontime, r_sessiontime, and sr_sessionime) are devised by aggregating the flows based on source, destination, and timestamp attributes using a time window of one minute. Also, after preprocessing the input features, the most

important 45 input features are selected. Moreover, the model's parameters are learned using many multiclass labeled flows from the AMD dataset. The hyperparameters of the model are optimized for best performance in terms of accuracy, recall, precision, and training time of the model.

The experimental results confirmed the high performance of the proposed model when tested from the "AMD" dataset. In addition, the optimal model architecture consists of one input layer, three hidden layers and one output layer. The model achieved an accuracy of 99.8 %, a false positive rate of less than 1%, and an area under the receiver operating characteristic curve (ROC-AUC) of 0.999. Also, the detection accuracy of the multiclass classifier is 99.6%

When the proposed model is compared with other recent models in literature, that was evaluated on similar datasets like" AMD", the experimental results show that the proposed model outperforms other models in terms of precision and recall.

Keywords: Machine learning, Cyberattacks, Classification, Android Malware Detection, AMD dataset, Deep learning, Neural Networks, Multiclass.

نظام كشف البرامج الخبيثة في نظام أندرويد باستخدام الشبكات العصبية العميقة.

اعداد: ماهر جورج ماريا

اشراف: د. رشدي حمامره

الملخص

تساهم أنظمة كشف البرمجيات الخبيثة (Malware detection systems) للأجهزة المحمولة في زيادة امن الأجهزة المحمولة الحاسوب. تشكل امن المعلومات الرقمية مشكلة رئيسية في الوقت الحالي تتدفع بشكل رئيسي لزيادة عدد البرمجيات الخبيثة، يكلف فقدان البيانات بسبب اختراقات الأجهزة المحمولة العديد من الدول خسارة كبيرة. وتنشأ مشاكل أخلاقية إذا تم الكشف عن المعلومات الشخصية للعملاء والمستخدمين. نظرًا لانتشار استخدام الهواتف الذكية والأجهزة المحمولة، لذا أصبحت الأجهزة المحمولة هدفًا للبرمجيات الخبيثة والهجمات السيبرانية. لذلك، من الضروري تطوير طرق جديدة لمنع واكتشاف ومواجهة الهجمات السيبرانية. في احدى هذه التقنيات، يُستخدم التعلم الآلي (Machine learning) لإنشاء أنظمة كاشفة تحدد ما إذا كان التطبيق خبيثاً أم لا. ميزة الشبكات العصبية العميق (DNN) هي أنها تسمح باتخاذ قرار بشأن البرامج غير المصنفات، على عكس الأنظمة التي تعتمد على قواعد بيانات ثابتة حيث تعتمد التكنولوجيا المقترحة في تحديد السلوك الخبيث لتعمم على الأنظمة التي تعتمد على القواعد الثابتة.

في هذه الرسالة تم اقتراح نموذج لكشف البرمجيات الخبيثة لأنظمة Android بناءً على تصنيف الشبكات العصبية العميقة المعتمد على مجموعات من الميزات(Features) . ويمكن تقسيمها إلى مجموعتين الطرق الثابتة(static) لفحص شفرة التطبيق المحمول اما الثانية الميزات الديناميكية(Dynamic) لتحليل سلوك التطبيق عند تشغيله على جهاز محمول. حيث تم اقتراح نموذج هجين(Hybrid) من هاتين الطريقتين للاستفادة من بعض المزايا التي تستخدم في الطريقتين التي تزيد من صحة الكشف (accuracy) ودقة الكشف (precision) في النموذج المقترح. للقيام بذلك، اخترنا استخدام قاعدة بيانات مختلطة تسمى "AMD" مكونة من 85 ميزة.

لتحسين الدقة في الكشف والتي تعتمد على ترددات تتغير مع الزمن مثل الهجمات، قمنا باستخراج ثلاث ميزات جديدة للإدخال هي s_sessiontime و r_sessiontimeو sr_sessionimeعن طريق الجمع بين الميزات بناءً على السمات المصدر والوجهة والطابع الزمني باستخدام نافذة زمنية دقيقة واحدة. كما تم اختيار السمات الأكثر أهمية بعد

معالجة ميزات الإدخال والتحقق منها. وايضا، سنقوم بتعليم النموذج العديد من المعلومات باستخدام عدد من الميزات ذات التصنيف المتعدد من قاعدة البيانات. "AMD" يتم تحسين الأوزان العالية للنموذج للحصول على أداء أفضل من حيث الدقة والاستدعاء والدقة ووقت التدريب.

تم تحقيق نتائج عالية للنموذج المقترح عند تطبيقه باستخدام قاعدة البيانات ."AMD" بالإضافة إلى ذلك، تتألف الهندسة المعمارية المثلى للنموذج من مدخل بطبقة واحدة ومخرج بطبقة واحدة وبينهم ثلاث طبقات مخفية. وقد حقق النموذج دقة بنسبة 99.8٪، مع معدل إيجابيات خاطئة أقل من 1٪، ومنطقة تحت منحنى السمت العملياتي للتشغيل-ROC) (AUCتبلغ 0.999. أيضًا، دقة الكشف للمصنف المتعدد تبلغ 99.6٪. وإذا تم مقارنة النموذج المقترح مع نماذج حديثة أخرى في الأدبيات، التي تم تقييمها على مجموعات بيانات مشابهة مثل"AMD" ، يظهر أن النموذج الخاص بنا يفوق النماذج الأخرى من حيث الدقة والاستدعاء.

Table of Content

List of Figures

List of Tables

List of Equations

Acronyms

AMD	Android Malware Dataset
APK	Application Package Kit
AUC	Area Under the Curve
CM	Confusion Matrix
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
CSV	Comma Separated Values
DNN	Deep Neural Networks
FN	False Negative
FP	False Positive
GPU	Graphical Processing Unit
HAMD	Hybrid Android Malware Detection
IoT	Internet of Things
IP	Internet Protocol
KNN	K Nearest Neighbor
MDI	Decrease in impurity
MLP	Multi-Layer Perceptron
RNN	Recurrent Neural Network
RNN	Recurrent Neural Networks
ROC	Receiver Operating characteristic curve
SVM	Support Vector Machines
TN	True Negative
TP	True Positive

Chapter One: Introduction

Chapter One: Introduction

1.1 Research Background

Mobile devices are becoming increasingly prevalent in our daily lives due to their popularity. And since these devices contain sensitive information, they are vulnerable to cyberattacks by malicious individuals. These attacks, such as the DDoS attack and the WannaCry ransomware, have become well-known examples of cyberattacks. Therefore, there is a need to investigate and improve malware detection techniques.

Many researchers have developed malware detection methods such as antivirus, static, dynamic, and hybrid methods. However, these methods have limitations in detecting newly installed malware on Android devices [4][5].

Many tools for detecting malware in Android systems can be classified as host-based, network-based, or signature-based detection which compares a feature to a database of known malware signatures. But by using anomaly detection we detect any unusual behavior compared to natural behavior. Thus, the proposed model HAMD-DNN is better in detecting malware.

Common models are classified by the detection approach signature-based classification that detect based on known patterns or signatures of malware. Our model is built on Anomaly-based which classify the detection according to its behavior and we combine that anomaly based technique with hybrid features which is a combination of static and dynamic features and using this large combination leads us to proposed a binary classification HAMD-DNN and multiclass HAMD-DNN models based on deep neural networks.

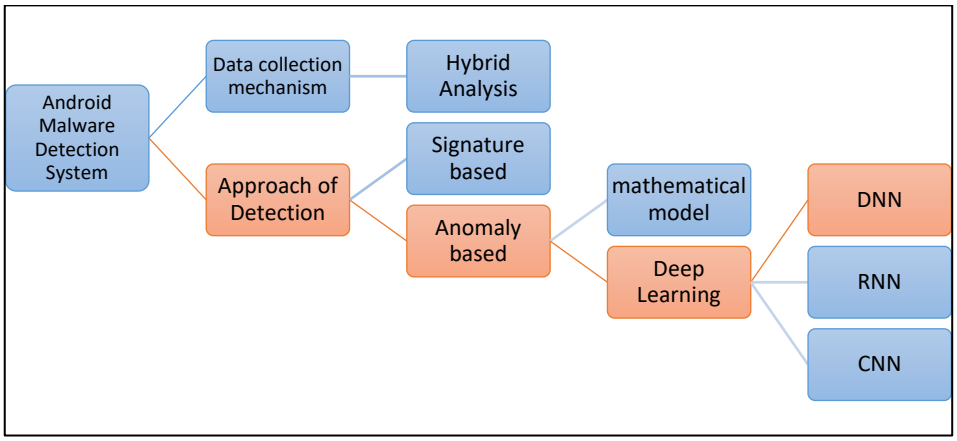


Figure 1 Android Malware Detection System Taxonomy

By machine learning process, we introduce a malware detection model that can be used to find a solution for the problem statement of detecting Android malware.

Anomaly-based malware detection model assigned to the hybrid feature dataset (AMD), with the aim of maximizing the detection of android malware and minimizing the false alarm rate.

This HAMD-DNN model goes through machine learning process, by defining objectives, data collection and preparation, building the model, evaluation, and prediction.

Our primary objective is to establish whether new samples provided to the proposed model HAMD-DNN are benign or malware, via machine learning algorithm [6][7].

This Thesis presents a novel approach to malware detection on Android devices using a hybrid anomaly-based machine learning model called HAMD-DNN, which is trained on the AMD dataset and uses various techniques to improve accuracy. This approach has the possibility to extensively expand the mobile devices security against malware attacks [3][8].

1.2 Thesis Problem Statement

The escalating prevalence of Android malware presents a critical challenge in maintaining the security and privacy of mobile device users. The inherent openness of the Android ecosystem, while fostering innovation, also provides a breeding ground for malicious applications. Conventional signature-based detection methods struggle to keep pace with the rapid evolution of malware, particularly in the face of polymorphic and obfuscated variants. This necessitates the development of a novel Android malware detection solution that transcends static signatures and adapts to the dynamic landscape of emerging threats. The goal is to create an advanced detection system capable of accurately identifying both known and previously unseen malware strains while minimizing false positives and negatives.

Two operating systems share around 98% of the Android market (Android architecture available in Appendix A.1) represents more than 80% of the mobile market while apple Inc.'s IOS represents a little less than 20% of the market according to Stat counters and Worldwide Quarterly Mobile Phone Tracker [10] .

In this context we seek to propose a model for the detection of malware for android devices. This operating system is open, unlike IOS, which offers more opportunities as well as a large community. In addition, the Android OS is used on different mobile devices. Therefore, research question is:

Can we develop a model that achieve a high malware detection with a minimum false rate?

1.3 Thesis Objectives

Android malware poses a significant threat to the security and privacy of mobile devices, as well as the sensitive information stored within them. Detecting and mitigating these malicious attacks is of utmost importance to ensure the privacy, reliability, and accessibility of Android devices. The continuous evolution of IT and the exponential growth in the number and sophistication of Android malware attacks have propelled the need for advanced techniques in the field of cybersecurity.

In this Thesis, the motivation lies in leveraging the power of deep neural networks (DNN) to develop an intelligent and automated Android malware detection system. By utilizing DNN, which has demonstrated remarkable capabilities in various domains, we aim to enhance the accuracy and effectiveness of detecting and classifying Android malware. This Thesis seeks to address the pressing need for robust and efficient solutions in the field of Android malware detection, contributing to the overall security and trustworthiness of mobile devices and their users [10] [12].

Through the development of an anomaly-based Android malware detection system, we seek to provide a proactive defense mechanism against the ever-evolving landscape of malware threats. By combining the extensive capabilities of DNN with the unique features of Android malware, this project aims to create a dependable and effective solution that can adapt and evolve to counter new and emerging threats. Ultimately, the successful implementation of this Thesis can significantly contribute to the advancement of cybersecurity in the realm of Android devices, safeguarding user data and preserving the integrity of mobile ecosystems [11] [12].

1.4 Thesis Questions

1. How can we prepare data to fit a deep learning model?
2. What's the best taxonomy for deep neural networks?
3. What are the hyperparameters that most affect modeling?
4. What is the performance metrices to test the model?
5. How efficient is the performance of the HAMD-DNN model based on the performance metrics?

1.5 Thesis Assumptions

This Thesis was conducted with the following assumptions:

- **Malware Detection System:** HAMD-DNN malware detection system for Android.
- **Algorithm**: Supervised deep neural networks.
- **Input:** Android application features.
- **Output:** In binary classifiers, the output is the decision, whether it is malware or benign.
- **Dataset**: Android malware dataset (AMD).

1.6 Thesis Contribution

- We developed a hybrid anomaly-based algorithm selected from static and dynamic features to detect Android Malware types.
- We developed a new binary and Multiclass classification models based on deep neural network (DNN) using a UpToDate Malware dataset (AMD)
- We developed a classifier based on RelU and SoftMax activation functions with Adam optimizer.

- We enhanced our model by adding new features to detect malwares that have time-dependent such as Android SMS Malware
- We tuned parameters like hidden layers, neurons per layer, learning rate, and batch size for optimal performance.
- We determine performance metrics (accuracy, recall, precision, and F1-score) on unseen instances of the AMD dataset.
- We provide a detailed methodology for obtaining and analyzing the detection results of HAMD.

1.7 Thesis Structure

The Thesis is structured into five chapters structured as follows:

Chapter 2: Malware Detection on Android Mobile Devices

This chapter conducts a comprehensive literature review of the field, focusing on various methods for scanning the Android operating system. we research into scientific research articles published in reputable scientific journals. We conclude this chapter by highlighting the limitations of the methods discussed. By analyzing these methods and recent publications, we lay the foundation for devising our own approach.

Chapter 3: Proposed Model: HAMD-DNN

This chapter introduces the architecture of the proposed HAMD-DNN model. We present a detailed explanation of the design and implementation of this model.

Chapter 4: Experimental Results

This chapter shows the experimental setup used for our Thesis, as well as the performance evaluation metrics employed. Later, the results of our experiments are presented and thoroughly analyzed.

Chapter 5: Discussion and Conclusion

This chapter gives an in-depth discussion of the results obtained. We draw conclusions based on our findings and insights from the research. Furthermore, we provide recommendations for future work and potential areas of research.

With this organized structure, we aim to present comprehensive and solid research that contributes to the existing knowledge in the field.

1.7 Threat Model

- Our model detects different types of malwares.
- Our model focused on detecting 3 types of malwares (Adware, Scareware, SMS malware) because these are new and unfamiliar malware types:
    - **Android Adware** is a program type of malware that displays unwanted ads on computers or mobile devices. It monitors browsing activity to bombard users with pop up ads and other advertisements.
    - **Android Scareware** is a cyberattack scam that hackers use to scare people into downloading malware, clicking on dangerous links, or visiting infected websites.
    - **SMS malware** is promoted by a text. The efforts of this harmful software are designed to breach and operate on a mobile device without the user's permission. Once on a device, the malware can then cause any number of detrimental effects.
- Our model was implemented using one input layer and 3 hidden layers, and have two types of output classification, the binary classification which uses the sigmoid as the activation function and the multi-classification which uses the SoftMax activation function.
- Our model is trained and tune to get the best accuracy and efficiency possible.
- Our model archives better malware detection compared to previous research.

1.8 Summary

We created a hybrid malware detection model for Android, utilizing two classification neural networks based on hybrid feature sets. Prior to this, the Thesis seeks to leverage the strengths of both static and dynamic methods. With the AMD hybrid dataset that consists of 355630 samples divided into benign and three types of malwares.

Chapter Two: Malware Detection on Android Mobile Devices

Chapter Two: Malware Detection on Android Mobile Devices

This chapter presents a literature review on the detection of malware targeting smartphones that use the Android OS. The first section of the literature review presents the different methods of analysis of the Android OS. The second section discusses the solutions offered by companies available on the Google Play Store as well as the solution offered by Google Inc. [12] and the third section focuses on the different approaches of this Thesis. Finally, the fourth and last sections will list the shortcomings of the solutions previously presented.

2.1 Android Operating System Scan Methods

2.1.1 Signature based Detection.

We will describe some techniques that are based on different signature approaches and methodologies for analyzing and detecting malware within the Android operating system:

A. Meta-Information Analysis

This technique is based on analyzing meta-information, such as data extracted from the Android Manifest file.xml. Machine learning algorithms are trained on these features to detect patterns that indicate potentially malicious behavior. The focus on system permissions in the Android Manifest file.xml is a key aspect of this method, as it provides insights into an app's capabilities and potential security risks. [13] [14].

$$y = w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b$$

Where:
$x_1, x_2, x_3, \ldots, x_n$ represent the meta-information features of an Android app.
$y = 1$ is the target label where $y = 1$ indicates malicious and $y = 0$ indicates benign.

$w_1, w_2, w_3, \ldots, w_n$ are the weights assigned to each feature.
$b = 1$ is the bias term.

B. Native Code Analysis

Native Code Analysis is centered around examining machine code (native code) that is directly executed by the processor. This technique focuses on the code that runs outside the virtual machine, such as Java bytecode. It aims to identify malicious behavior or threats within native code components of Android apps. The method leverages cross-platform semantic signatures and dynamic analysis to detect malware hidden within native code. [15] [17] [18].

The predicted probability that the app is malicious $P(y = 1)$ can be computed using the sigmoid function:

$$P(y = 1) = \frac{1}{1 + e^{-y}} \quad (1)$$

C. Clone Analysis

Clone Analysis is based on detecting and analyzing cloned or duplicated Android apps. The technique involves identifying similarities and differences between apps to differentiate genuine apps from clones that may contain added malicious code. It uses code representations, data removal, and hashing techniques to abstractly represent and compare code sections, effectively detecting even highly obfuscated clones. [14] [17].

$$y = X.W + b \quad (2)$$

You can then apply a threshold to classify the code fragments as clones $y = 1$ or non-clones $y = 0$.

D. Network Traffic Analysis

Network Traffic Analysis is built upon the examination of network traffic generated by Android apps. The method involves capturing and analyzing the packets of data exchanged between the device and external servers. It aims to identify patterns or anomalies in the network behavior of apps, which could indicate potential malware or privacy threats. This technique focuses on detecting unauthorized data transfers, tracking user behavior, and identifying apps through packet-level analysis [16].

The anomaly score S for a feature vector X is calculated as the sum of the entire changes between each feature value, a baseline value B (representing expected or normal behavior) [18]:

$$S = \sum_{i=1}^{n} |x_i - B_i| \quad (3)$$

2.1.2 Anomaly-Based Detection

Behavior-based detection, also known as anomaly-based detection, is a widely used method for identifying potential malware in the context of malware detection. This approach focuses on detecting deviations from normal behavior, making it capable of discovering previously unknown malware threats that traditional signature-based methods might miss [19] [20].

The process of anomaly-based detection starts by establishing a baseline of what is considered normal behavior for a system or application. This baseline is derived from analyzing legitimate activities and features associated with the system or application, including files accessed, network connections, processes executed, system calls, and other relevant indicators.

Once the baseline of normal behavior is set, any deviations or anomalies from it can be identified and flagged as potentially malicious. These anomalies could indicate the presence of malware or other malicious activities that have managed to bypass signature-based detection methods [19].

Anomaly-based detection employs various techniques to effectively identify and analyze deviations from normal behavior:

- **Statistical Analysis:** Statistical methods are used to model normal behavior patterns and detect statistical outliers, which can indicate potential malicious activities.
- **Machine Learning:** Machine learning algorithms are trained on known normal behavior patterns, enabling them to recognize and flag anomalies. This adaptive approach improves detection accuracy as the model learns from new data.
- **Heuristics:** Rule-based heuristics are applied to detect specific patterns or behaviors associated with malware, based on expert knowledge or known malicious patterns.
- **Data Mining:** Data mining techniques help discover patterns and correlations in large volumes of data, aiding in identifying anomalies and uncovering hidden relationships indicative of malware activity [20].

Anomaly-based detection is particularly effective in detecting emerging threats or new strains of malware, as it does not rely on predefined signature databases. However, it may produce false positives when legitimate activities deviate from the established baseline.

To enhance malware detection a combined approach that integrates both anomaly-based and hybrid features. By leveraging the strengths of both methods, these systems offer

comprehensive protection against malware and other cybersecurity threats. Figure 2 describes the basics of the anomaly-based techniques.

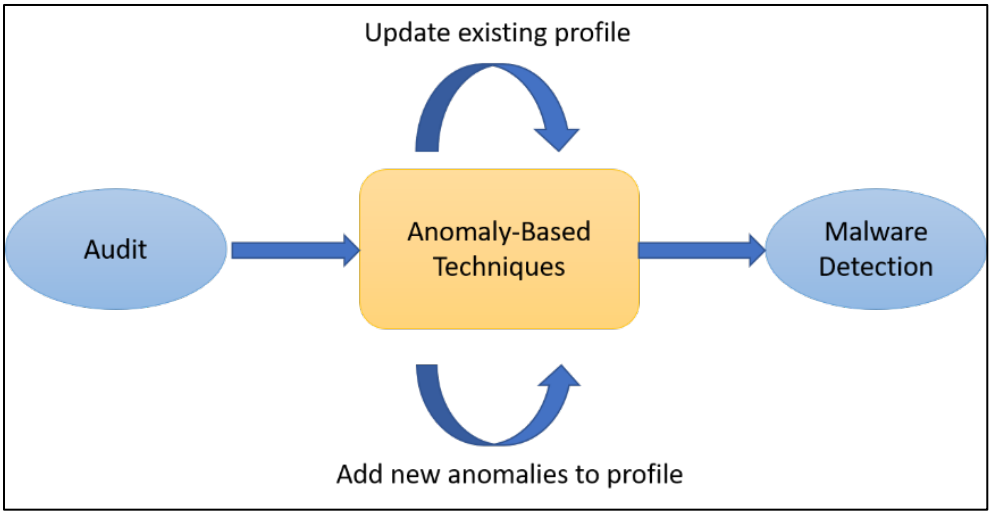


Figure 2: Anomaly-based technique of detection

2.1.3 Machine Learning

Researchers have recognized the complexity involved in developing an Android detection system using statistical and usual techniques, especially when dealing with large volumes of data. As a result, they have turned to the algorithms of machine learning to construct models that can efficiently classify benign and malware apps [21].

To create a machine learning model, a dataset is utilized and split into training and testing sets. The model is trained using the training set and then evaluated on the testing set. Popular frameworks like scikit-learn [22] and TensorFlow [23] are commonly used to manage the design of the machine-learning model.

Machine learning algorithms are categorized based on factors like data and label availability and the specific tasks they are designed to perform, as shown in Figure 3. This classification helps in selecting the most appropriate algorithm for the Android detection problem under consideration.

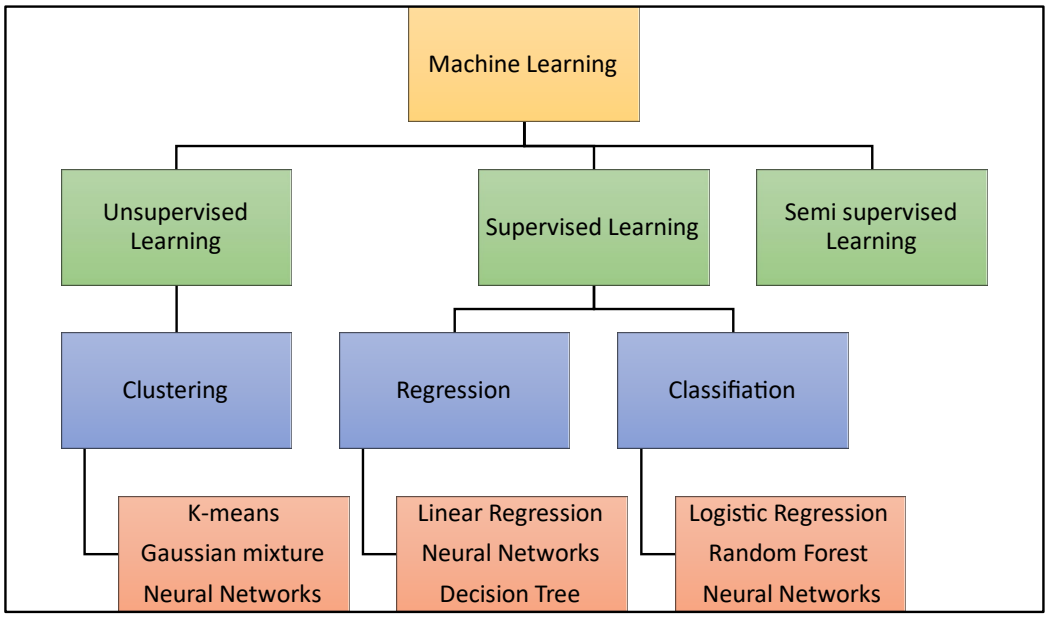


Figure 3: Machine Learning Algorithms Taxonomy

Machine learning taxonomy refers to the categorization and classification of various concepts, techniques, and approaches within the field of machine learning. This taxonomy helps provide a structured framework for understanding the different aspects of machine learning and how they relate to each other. Here's a high-level overview of the machine learning taxonomy:

1. **Supervised Learning:** This machine learning paradigm involves training a model or function that establishes a relationship between the input and the output (X, Y), as formulated in equation 1.

$$\boldsymbol{Y = f(X)} \tag{4}$$

In supervised learning, the objective is to predict the output ($\hat{Y}$) from the input (X) with least error. To achieve this, a labeled dataset is needed, which consists of examples from a specific domain. Each example in the dataset includes input variables or features and the corresponding output variable or target.

Supervised machine learning algorithms learn the model parameters from the labeled dataset by minimizing the loss between the true labels and the predicted targets. In the context of Android malware detection, the input variables are the Hybrid features extracted from the Android Package Kit (APK). The algorithm uses these features to predict whether an APK is malicious or benign, based on the labeled examples in the dataset.

2. **Semi-supervised Learning**: This machine learning concept brings together both supervised and unsupervised learning. It is utilized when the available dataset contains only a controlled amount of labeled data but a substantial amount of unlabeled data. The algorithm leverages the labeled data for supervised learning tasks and utilizes the unlabeled data for unsupervised learning tasks to improve overall performance.
3. **Unsupervised Learning:** In this machine learning technique, the dataset consists only of unlabeled data, meaning there are no corresponding output provided. The algorithm's objective is to classify structures, relationships, patterns or within the data without explicit guidance on what to look for. It is often used for tasks such as clustering, anomaly detection, and dimensionality reduction.

	From input x, output
Unsupervised	Summary **z**
Semi-Supervised	Prediction **y**
Supervised	Action **a** to maximize reward **r**

Figure 4: Machine Learning Based on Dataset Labels Taxonomy

Machine learning algorithms can be categorized into several main types based on their functionality and the tasks they perform. Here are the main categories of machine learning algorithms:

1. **Regression**: This algorithm is used to train a model in supervised learning that predicts a non-stop value of a target variable. [24].
2. **Classification**: This algorithm is used to train a model in supervised learning that classifies data into different categories or classes. The classifier can be a binary classifier, accomplished of classifying between two classes. As in malware detection, a binary classifier model can classify between benign and malware applications [20].
3. **Clustering**: In this task, data is split into groups or clusters based on some relationship between characteristics. [25] [24].

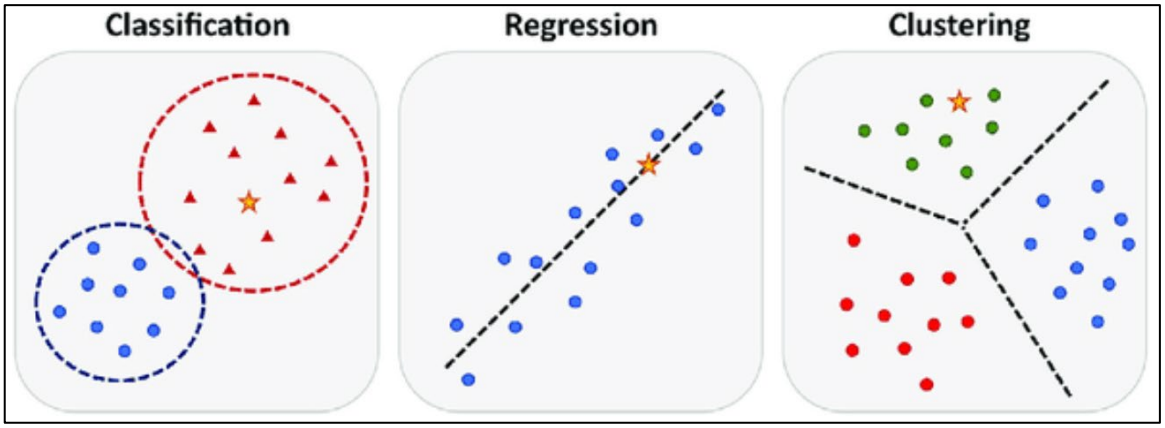


Figure 5: Machine Learning Algorithms

Classification algorithms are a type of supervised learning algorithms that assign input data points to predefined classes or categories,

a) **K Nearest Neighbor (KNN):** K Nearest Neighbor categorizes a sample based on the classes of its k nearest neighbors. If the majority of neighbors belong to the same class, then the sample is assigned to that class with a high probability [7]. The algorithm employs a distance system of measurement to classify samples. [26].

Nearest Neighbor – Distance Measures

Given the feature vector X, and calculating distance between X and all examples in the training dataset using a distance metric (e.g., Euclidean distance):

$$\text{distane}(X, X_i) = \sqrt{\sum_{j=1}^{n} (x_i - x_{ij})^2} \quad (7)$$

Support Vector Machine (SVM): Support Vector Machine is a classification algorithm that finds a linear decision boundary separating data points of one class from those of the other class. For linearly separable data, the SVM identifies the hyperplane with the largest margin between classes. When dealing with data that isn't linearly separable, a loss function imposes penalties on misclassified points that fall on the incorrect side of the hyperplane. Support Vector Machines (SVMs) employ kernel transformations to convert non-linearly separable data into higher-dimensional space, enabling the identification of a linear decision boundary [7] [8]. The equation describing the hyperplane in a linear SVM can be expressed as:

$$w.X + b = 0 \quad (8)$$

Where w is the weight vector.
X is the feature vector
b is the bias term.

b) **Artificial Neural Networks (ANNs):** Artificial neural networks form a subset of machine learning algorithms that draw inspiration from biological neural networks. They adopt a feedforward structure, comprising an input layer, one or multiple hidden layers, as well as an output layer. Figure 6 illustrates the presence of neurons within these hidden layers. These neurons establish connections with the preceding and subsequent layers via links, each assigned a specific weight. Within each layer, neurons gather inputs from the previous layer, apply activation functions, and transmit results to the subsequent layer [25]. Thanks to their capacity to grasp intricate patterns within data, artificial neural networks prove adept at tasks encompassing classification, regression, and pattern recognition [24].

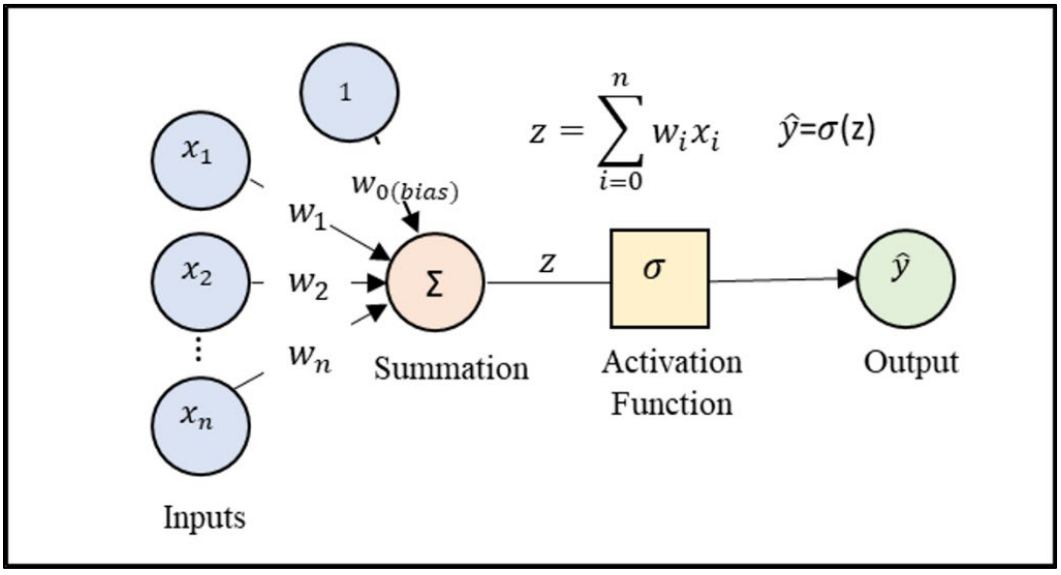


Figure 6: Artificial Neuron Architecture

2.1.4 Deep Learning

Deep learning constitutes a subset of machine learning techniques rooted in artificial neural networks. Models known as Deep Neural Networks (DNNs) employ neural networks featuring two or more hidden layers, facilitating the application of nonlinear modifications to input data. With each passage through a hidden layer, the data undergoes a progressive extraction of higher-level feature representations.

Within supervised learning, the DNN model undertakes the task of mapping input feature vectors (x) to corresponding labels (y) by navigating through intermediate transformations [25]-[27]. This process empowers the model to discern intricate data patterns and relationships, rendering deep learning remarkably potent for a wide array of assignments, such as image recognition, natural language processing, and speech recognition. Nevertheless, the most prevalent forms of DNN include:

1) **Feedforward Deep Neural Network (DNN):** This pertains to a subtype of deep learning algorithms where data moves from the input layer through hidden layers to the output layer without feedback connections, as depicted in Figure 7. DNNs are often used to establish clear representations of input data, especially when features are fixed in length. They find widespread use in applications like image and speech recognition, as well as natural language processing, owing to their knack for learning intricate patterns and representations from data.

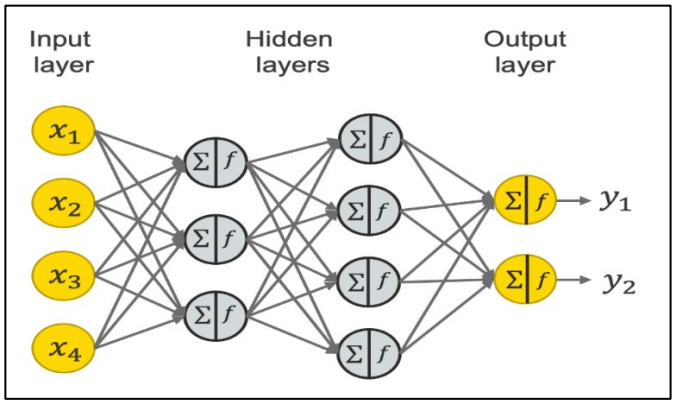


Figure 7: Feedforward Deep Neural Network Architecture

2) **Recurrent Neural Network (RNN):** RNN is a potent architecture built to process sequential data. It is well-suited for handling two-dimensional inputs of different sizes, like images or videos. The network's strength lies in capturing the stochastic nature of data through feedback connections between its hidden layers, shown in Figure 8. These feedback connections allow the network to model intricate patterns and dependencies within the sequential data, making it highly effective in tasks involving image recognition, video analysis, natural language processing, and other domains that require sequential data processing.

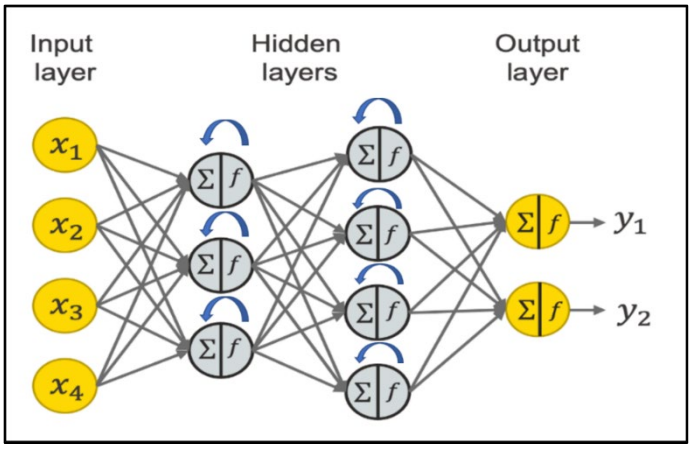


Figure 8: Recurrent Neural Network Architecture

3) **Convolutional Neural Network (CNN):** CNN is designed for classification tasks that excels at capturing patterns in two-dimensional or variable-sized inputs, such as images or videos. Relevant filters are applied to learn spatial and temporal associations in the input features. The convolutional layer performs a matrix multiplication between a filter and the input to achieve this.

CNN is notably effective when tackling problems involving a multitude of input variables, as it minimizes the parameters requiring learning in contrast to feedforward deep neural networks [25], [28]. The structure of a convolutional neural network's architecture is illustrated in Figure 9.

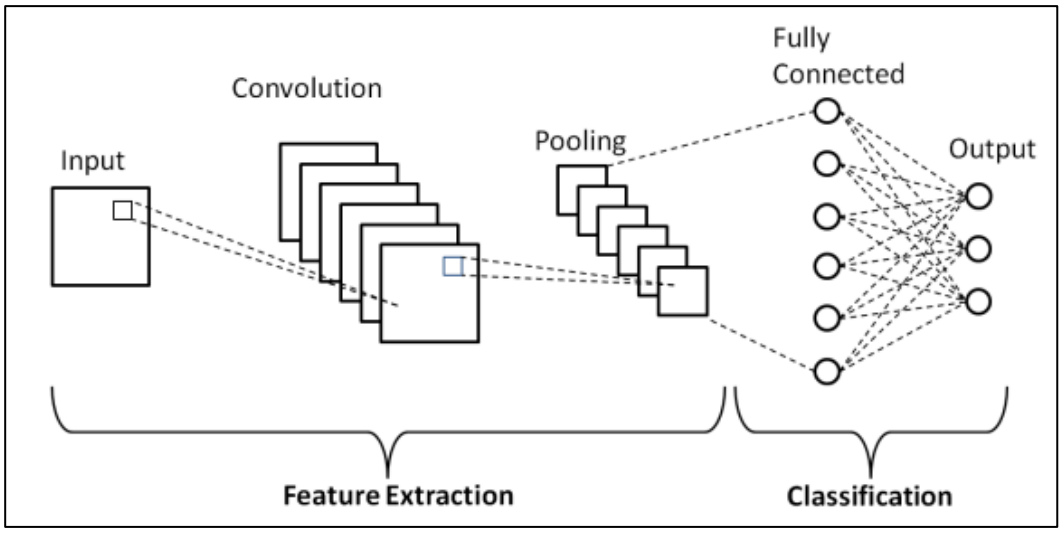


Figure 9: Architecture of Convolutional Neural Network

I chose to implement a feedforward Deep Neural Network (DNN) for Android malware detection due to its suitability for the specific characteristics of the data and the task at hand. Unlike Recurrent Neural Networks (RNNs), which excel at processing sequences and capturing temporal dependencies, the nature of the static and dynamic features in Android apps doesn't necessarily require modeling sequential patterns. Since Android malware detection is more about identifying complex relationships and patterns within feature vectors, the feedforward architecture of a DNN is well-suited for this purpose.

Convolutional Neural Networks (CNNs), on the other hand, are primarily designed for processing grid-like data such as images. While CNNs can be adapted for feature extraction, the multidimensional nature of their convolutional layers might not align with the one-dimensional nature of feature vectors used in our case. A DNN, with its fully connected layers, can effectively handle the linear relationships among various features extracted from Android apps.

By opting for a DNN, I am capitalizing on its ability to learn intricate feature interactions in a holistic manner. This is particularly valuable for malware detection, as malicious behaviors can often be nuanced and require capturing complex combinations of features. Additionally, a DNN's architecture offers flexibility in terms of depth and hidden units, allowing it to learn hierarchical representations of the data, which is essential when dealing with diverse and evolving malware patterns. Overall, the choice of a feedforward DNN aligns well with the nature of the Android malware detection problem, focusing on capturing intricate patterns within feature vectors rather than sequential or grid-like data.

2.2 Activation Functions

1. **Well-Known Activation Functions:**

   a. Linear Activation (Identity Function):

   - Equation: $f(x) = x$
   - Range: $-\infty, +\infty$
   - Limitation: Linear activations can lead to poor learning in deep networks since they can only model linear relationships. Malware detection often requires capturing complex non-linear patterns.

   b. Step Function:

   - Equation: $f(x) = x \begin{cases} 1, & if\ x \geq 0 \\ 0, & if\ x \leq 0 \end{cases}$
   - Limitation: Step function is not continuous and not differentiable, making it unsuitable for gradient-based optimization methods like backpropagation used in training neural networks

c. Sigmoid Function:

- Equation:$\sigma(x) = \frac{1}{1+e^{-x}}$
- Range: (0,1)
- Use: used for binary classification models, where the result represents the probability of the positive class (malware).
- Pros: Well-suited for binary classification tasks, smooth gradient, interpretable probabilities.
- Cons: Saturates and causes gradients to vanish for large inputs (vanishing gradient problem).

d. ReLU (Rectified Linear Unit):

- Equation: $f(x) = \max(0, x)$
- Range: $0, +\infty$
- Use: Commonly used in hidden layers for both binary and multi-class classification tasks.
- Pros: Avoids vanishing gradient problem, computationally efficient, encourages sparse activations, learns quickly.
- Cons: May cause "dying ReLU" problem (units always output 0), not centered around zero (can lead to optimization challenges).

e. SoftMax Function:

- Equation: $SoftMax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$
- Range: (0,1)

- Use: Typically assigned to the output layer of multi-class classification models to produce class probabilities.
- Pros: Converts raw scores into probability distribution, suitable for multi-class problems, handles multiple classes well.
- Cons: Sensitive to outliers, can lead to large gradients during training.

2. **Reasons for Choosing Common Activation Functions:**
   a. **Sigmoid**: Well-suited for binary classification tasks like malware detection where output needs to be a probability (range between 0 and 1).
   b. **ReLU**: Addresses the vanishing gradient problem and accelerates convergence by allowing gradients to flow more freely in the network, also it's simple and computationally efficient, making it suitable for deep networks.
   c. **SoftMax**: Transforms raw scores into a probability distribution over multiple classes, making it suitable for multi-class classification tasks such as detecting different types of malware.

When designing a malware detection model, it's important to consider factors such as network architecture, data characteristics, computational efficiency, and the specific goals of the detection task. The choice of activation function should align with these considerations to ensure effective learning and accurate predictions.

2.3 Thesis Approaches and Tools

The below subsections represent an overview of the malware detection solutions offered by the academic community, demonstrate the dataset used for this Thesis, and describe the static, dynamic analysis beside the hybrid analysis approaches highlighting their advantages and disadvantages.

2.3.1 Overview

This section presents an overview that covers two key aspects. Firstly, it provides a comprehensive overview of the proposed solutions, offering a broad and inclusive understanding of their nature and potential impact. Secondly, it provides a more specific view of the machine learning based solution. These aspects allow for both a high-level understanding and a focused insight into the approaches being considered.

1. General Overview

The Taxonomy Comparison of Android Software Security Analysis Methods, issued in October 2016 by Sadeghi [34] are a good start in the study of mobile malware detection techniques. Literature review articles are excellent for giving an overview of the domain since they do not stop at specific detection methods. First, it should be noted that this Thesis is only to analyze malware for Android OS. Because the majority market share of smartphones using this system (80%), phones that are not smart are not considered since most malware attacks smartphones. The use of Android is also preferred by researchers because it is an open OS allowing more possibilities. The authors of the taxonomy deal with both malware detection methods and vulnerability detection methods. There are many tables whose categories are built by reading articles and previous literature reviews and taxonomies. The articles studied are classified according to the threats considered (information leakage, spoofing, elevation of privilege, etc.), according to the specifications

of the problem (deployment layer of the solution, finesse of the threat, objects studied in the system, etc.), according to the approaches (static, dynamic or hybrid analysis) and the additional methods (machine learning, formal analysis) and level of automation. However, methods based on dynamic analysis are again grouped according to the level of examination (apps, kernel, VM) and the generation of application inputs, while static methods can be based on different data structures, can represent the code in various ways and have varying analysis sensitivity [34].

The comprehensive review and analysis of malicious smartphone applications M. Talal et al. [35] aims to provide an overview of malware detection analyses by probing and taxonomizing the literature and modern security solutions available that are proposed by different researchers. A categorization of the available security solutions was thus made based on their structures. Four distinct categories have been established, namely 1) revisions and studies, 2) security solutions for mobile devices, 3) smartphones security surveys and classification, 4) grouping and classification. The tables present detailed information on techniques for detecting malicious software based on artificial intelligence or not. Thus, for the thirty articles whose detection is based on artificial intelligence, we can find the classification technique used (Classification or Clustering), machine learning algorithms, datasets (standard, collected by the authors) and many others. Then, several questions and personal gaps of researchers are focused on the recommendations for several participants (researchers, users, developers and finally companies) to enhance research in each respective area [35].

2. Overview of Machine Learning-Based Solutions

A. Souri and R. Hosseini [36] is a review of the literature on malware detection approaches using learning. The studied articles were classified into two main categories. The first category is the one that groups signature-based approaches and the second category groups behavior-based approaches together. Experimental results show that 65% of these detection approaches used the dynamic analysis method while signature-based approaches are more oriented towards static detection methods. [36].

All methods based on supervised learning, regardless of the algorithms used, need datasets with features that come from malware and benign applications to train their models.

2.3.2 Android Malware Datasets Overview

Detection tools on Android based on machine learning require the formation of datasets whose samples are labeled as malicious or benign. Using these datasets are essential to create reliable methods for detecting and classifying malware. To this end, different Android sample datasets have been made public in recent years.

The Genome Project [37] was launched in 2012 containing 1260 malware, but was discontinued in 2015.

Drebin [38] contains 5560 malware from 179 different families as well as benign applications extracted from different sources. Drebin is a well-known database used by many researchers to test the accuracy of their own detection methods but is too old because the malware was collected from 2010 till 2012.

Contagio [39] which releases new malware applications, This dataset covers 24553 samples collected between year 2010 till year 2016 from 71 malware groups.

The Koodous Portal [41] offers a large set of malware samples solely for research purposes, including in this case a feature report.

- 55535003 applications as of March 2020.

The AndroZoo project [42] offers a large data set holding more than 5700000 data samples being analyzed with different antivirus engines.

- 10502608 applications as of March 2020.

UGR'16: A New Dataset for the Evaluation of Network Based IDSs, which leads to a near path of study.

AMD [3] is a database containing a complete set of data on the dynamic and static features of Android applications obtained through the AndroPyTool feature extractor, each application has a total of 85 features. The features of this dataset make it suitable for use as a reference dataset for the scientific community to test different algorithms and techniques. In order to make the AMD dataset easy to use, all data is provided in json and csv formats and is publicly available on Aida under an international Creative Commons Attribution-Non-Commercial-Share Alike 4.0 license.

The dataset comprises four distinct labels: Android Adware, Android Scareware, Android SMS Malware, and Benign. It encompasses a total of 355,630 entries or instances, accompanied by 85 columns. The data exhibits the subsequent label distribution:

a. Android Adware 147443
b. Android Scareware 117082
c. Android SMS Malware 67397
d. Benign 23708

SandDroid [41] also presents a dataset of features extracted from malware samples, including static and dynamic analyses. If we compare it to AMD, SandDroid only includes malware samples and the feature set isn't as extensive.

2.3.3 Static Analysis Approaches

Static analysis is a method used in software security and malware detection that involves examining the code of a program without executing it. In the context of Android malware detection, static analysis focuses on analyzing the application's code, resources, permissions, and other characteristics to identify potential security issues and malicious behavior [44], and mainly this static approach uses a static dataset, and we can present some of these features as follow:

$$\mathbf{D_s = [PM, AC, MI, CC, UL, RF, DI, SU, ASC, CFA, DCL, MA]} \quad \mathbf{5}$$

Where: Permissions: PM
API Calls: AC
Manifest Information: MI
Code Complexity: CC
Used Libraries: UL
Resource Files: RF
Declared Intents and Broadcast Receivers: DI
Strings and URLs: SU
App Signing Certificate: ASC
Control Flow Analysis: CFA
Dynamic Code Loading: DCL
Metadata Analysis: MA

The static analysis approach can be described as follows:

1. **Code Inspection**: Static analysis involves inspecting the source code or bytecode of an Android application. The analysis looks for known patterns of malicious code, such as specific API calls or command sequences associated with malware.

2. **Signature-Based Detection**: Static analysis can use signature-based detection, where the application's code is compared to a database of known malware signatures. If a match is found, the application is flagged as potentially malicious.
3. **Permission Analysis**: Static analysis reviews the permissions requested by the application. It checks if the requested permissions are excessive or unrelated to the app's legitimate functionality, which could indicate malicious behavior.
4. **API Call Analysis**: By analyzing the API calls used by the application, static analysis can identify potentially harmful actions, such as sending SMS messages, making unauthorized network connections, or accessing sensitive data.
5. **Obfuscation Detection**: Static analysis can detect code obfuscation techniques used by malware to hide its true intentions or evade detection. It tries to deobfuscate the code and understand its actual functionality.

Here are the disadvantages of using only static analysis for Android malware detection, explained in simpler points:

- **Misses Sneaky Behavior:** Static analysis doesn't catch apps that act differently when running, letting some sneaky apps slip by.
- **Gets Confused:** Sometimes, it thinks harmless apps are harmful, causing confusion and trust issues.
- **New Threats Get Through**: It's not good at finding brand-new threats that don't match known patterns.
- **Struggles with Complex Apps**: For apps with lots of special tricks or secret code, it can't figure them out very well.

- **Doesn't Show All Tricks**: It doesn't tell us how bad apps change behavior in different situations, which we need to know.

To be good at finding malware apps, we used both static and dynamic ways together. This helps us find more problems and be more accurate.

2.3.4 Dynamic Analysis Approaches

Dynamic analysis is a method used in software security and malware detection that involves running the application in a controlled environment and observing its behavior during execution. In the context of Android malware detection, dynamic analysis focuses on monitoring the application's actions, interactions with the system, network communications, and other runtime behavior to identify potentially malicious activities [45], and mainly this dynamic approach uses a dynamic dataset with dynamic features that could be presented as follow:

$$\mathbf{D_c = [NT, FA, PA, ACS, RC, SU, ….]} \qquad \mathbf{(6)}$$

Where: Network Traffic: NT
File Access: FA
Process Activity: PA
API Call Sequence: ACS
Resource Consumption: RC
Sensor Usage: SU

The dynamic analysis approach involves observing an application's behavior as it runs, offering insights that static analysis alone might miss. It tracks real-time events like API calls, file operations, and network requests. Detecting malware activities is a primary goal, spotting unauthorized data access and suspicious network communication. This analysis is carried out in a controlled environment, a sandbox, ensuring the host system remains unaffected. Emulated Android devices are used for testing, enabling observation of hardware interactions. Sometimes, specific inputs or conditions are used to provoke

potential malware behavior, aiding in understanding app responses. Evasion techniques are unveiled, such as delayed malicious actions or conditional behavior. By profiling normal behavior, it distinguishes between legitimate and suspicious apps. Continuous monitoring throughout execution detects dynamically changing or time-based malicious activities.

However, dynamic features come with certain disadvantages. Dynamic analysis can be resource-intensive and time-consuming, as it requires running the app in a controlled environment, leading to slower analysis speeds. Some malwares may detect the presence of dynamic analysis tools or sandboxes, allowing them to alter their behavior to evade detection. Moreover, the sheer volume of data generated during dynamic analysis can make it challenging to process and extract meaningful insights, requiring sophisticated data handling and analysis techniques. Despite these challenges, dynamic analysis remains a crucial tool in the arsenal of Android malware detection, helping to uncover new and evolving threats that static analysis might overlook.

2.3.5 Hybrid Analysis Approaches

Hybrid analysis approaches for Android malware detection offer several advantages over single-method detection techniques. These advantages arise from their ability to combine the strengths of different analysis methods, which leads to improve accuracy, adaptability, and overall effectiveness in detecting a wide range of Android malware [46], as this Hybrid approaches deals with a combination of static and hybrid features and covers all their disadvantages, the hybrid dataset could be represented as follow:

$$\boldsymbol{D_h = D_s \quad \cap \quad D_c}$$

And the complete dataset features that we select are shown in Appendix B, Table 1 Details of the 45 Features in the AMD Dataset

Here's a description of some of the key advantages of hybrid analysis:

1. **Enhanced Detection Accuracy**: By combining multiple detection techniques, hybrid analysis can identify malware more accurately. Each method contributes its own unique insights, allowing the system to catch malware that might be missed by individual techniques. For example, static analysis can identify known malware patterns, while dynamic analysis can capture previously unseen malware behavior.
2. **Increased Robustness**: Hybrid analysis makes the system more resilient to evasion techniques used by sophisticated malware. Attackers may employ various tactics to evade detection, such as obfuscation and anti-analysis techniques. By combining static and dynamic analysis, the system can counter these evasion attempts more effectively.
3. **Comprehensive Coverage**: Different detection methods target different aspects of malware behavior. Hybrid analysis covers a broader spectrum of characteristics, including code signatures, behavior patterns, and permission usage, providing a more comprehensive approach to malware detection.
4. **Adaptability to New Threats**: Android malware is constantly evolving, with new variants and zero-day threats emerging regularly. Hybrid analysis can adapt to these changes more effectively because it can incorporate updates from multiple sources and leverage machine learning to learn from new samples and behaviors.
5. **Reduced False Positives/Negatives**: Single-method analysis may result in false positives (misidentifying benign apps as malware) or false negatives (failing to detect actual malware). Hybrid analysis, by cross-verifying results from multiple

techniques, helps minimize these errors and improves the overall reliability of the detection process.

6. **Resource Optimization:** Hybrid approaches can optimize resource usage. For example, less resource-intensive static analysis can be performed first to quickly rule out obvious benign applications. Then, more detailed dynamic analysis can be applied to the remaining suspicious apps, reducing the workload for resource-intensive techniques.
7. **Scalability**: Hybrid analysis can be designed to scale well with large datasets and a high volume of Android applications. By leveraging cloud-based analysis and ensemble methods, the system can handle a vast number of samples efficiently.
8. **Continuous Learning**: Hybrid analysis systems often employ machine learning algorithms to continuously improve their detection capabilities. As new malware samples are collected and analyzed, the system can learn from them and adapt to new threats over time.

Overall, hybrid analysis approaches for Android malware detection represent a powerful and adaptive defense mechanism against the ever-evolving landscape of Android threats. By combining the strengths of various techniques, these approaches offer a more comprehensive, accurate, and future-proof solution to protect users from the growing range of Android malware [46].

2.3 State-of-The-Art HAMD-DNN Models

Lately, a lot of research has been proposed to accurately identify malware in Android devices. The main attributes of these research are:

1. Anomaly based malware detection for Android which detects real time malware and better than signature-based Android malware detection, but there isn't any research that combines these benefits with the hybrid features analysis.
2. Deep neural networks (DNNs) outperform classical machine learning techniques when dealing with large datasets. This advantage is due to the deep structure of DNNs, However, it's essential to consider the specific requirements of each problem and choose the most suitable algorithm accordingly, as classical approaches may still be useful in certain scenarios, especially with limited data or when interpretability is crucial. [27].
3. Among the datasets extensively employed in research, prominent examples include Drebin, and M0Droid. Android malware datasets exhibit variations in terms of their sizes, dates, encompassed malware types, timestamps, and feature counts. Since machine learning relies heavily on datasets for learning model parameters and evaluating performance, the selection of a suitable dataset holds significance in ensuring the model's generalizability within actual network environments [38], [40], [41].

Table 2: State-of-the-Art HAMD Models

Work	Algorithms	Dataset	Accuracy
Droiddetector.[47] (2016)	ML& DL	CICMalDroid	89.7%
MMA.[38] (2019)	ML	Omnidroid	90.7%
DroidCat.[48] (2018)	Supervised ML	Drebin	97%
MiMaLo. [49] (2020)	KNN, SVM	MiMaLo	89.79%, 87.5%

ANASTASIA. [39] (2016)	DNN	M0Droid	97%
Droidmat. [40] (2012)	KNN	Contagio	91%
CTML. [41] (2022)	ML	AMD	99%

Utilizing deep neural networks, I have designed an Android malware detection model. While there exist numerous projects and research initiatives centered around comparable classifiers, the distinctiveness of our proposed model lies in the fusion of anomaly-based and hybrid feature analysis approaches. The outcome is a model that achieves an optimal level of detection accuracy while minimizing the occurrence of false alarms.

To substantiate the efficacy of our approach, a comprehensive comparison was undertaken, encompassing two distinct research benchmarks. The first, CTML, employs a dataset identical to ours but is exclusively based on signatures. The second, MiMalo, employs a similar dataset but exclusively relies on anomaly-based techniques without the incorporation of hybrid features. Within our research, I successfully demonstrated that the amalgamation of these techniques culminates in the development of a hybrid anomaly-based Android malware detection model, employing deep neural networks. This novel integration serves as the foundation for the title of our thesis, "Hybrid Anomaly-Based Android Malware Detection Model Using Deep Neural Networks."

All these comparisons are meticulously detailed within the results chapter of our thesis. Additionally, to provide a visual representation of these comparative outcomes, a comprehensive figure has been devised. This illustrative aid encapsulates the essence of the various models and their respective performance, effectively enhancing the clarity of our findings and underscoring the significance of our research in advancing Android malware detection methodologies.

2.4 Summary

In this chapter, we present a background about the Android operation system detections methods and enterprise solutions. In addition, we give a brief review for Android anomaly-based detection, machine learning and deep learning.

Finally, we show the state-of-the-art HAMD models. One of main properties of recent works is that deep learning is mostly used to detect malware on Android mobile devices. Many projects, models and published surveys recommended using deep learning since Deep learning is accomplished of modeling difficult issues.

In our proposed model we used the neural network to implement two model classifiers the first on is the binary classifier using the Sigmoid activation function, and the second one it the multiclass classification that gives us the exact malware type as output using the SoftMax activation function, and in common of both classifiers hidden layers the activation function is the ReLU activation function.

Furthermore, we found that the best path to go with regarding the previous works is the deep learning anomaly based for hybrid dataset features from Android malware dataset (AMD).

Chapter Three: Proposed Model: HAMD-DNN

Chapter Three: Proposed Model: HAMD-DNN

In this chapter, the first section introduces the architecture of HAMD-DNN, the second section shows the design and the implementation, while the third section presents the multiclass classification of our model.

3.1 HAMD-DNN Architecture

The HAMD-DNN model system contains three main mechanisms: data collection, data preparation, and classification as represented in Figure 10.

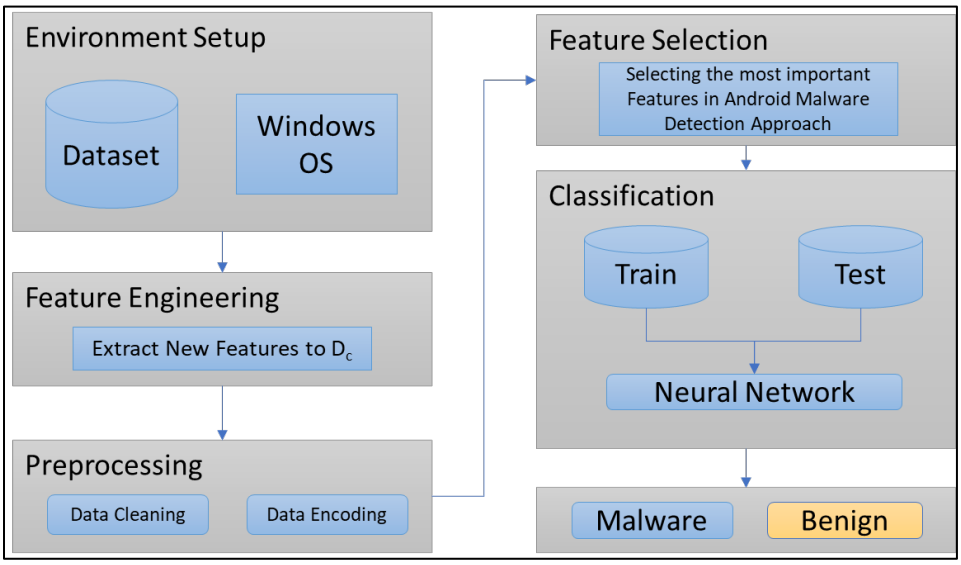


Figure 10: HAMD-DNN Architecture

This model is designed and implemented on windows OS as an work space environmental with an input of the new features a process of collecting the data, to a step of improving the efficiency of the dataset samples we are going to combine some features to create a new features and extract them to the dataset, the next step is to start the data preprocessing and preparing including the cleaning and encoding and scaling, then for the

last step before classification we are going to select them most important feature to work in.

3.1 Data Collection

In this thesis we are going to propose a hybrid malware detection model for Android. To do this, we chose to use the AMD [3] Dataset composed of 85 Hybrid features. A combination of static and dynamic features are employed. Each sample in the dataset is labeled as either benign or malware, enabling supervised learning approaches for subsequent model training and evaluation. The resulting customized Android malware dataset consists of 355630 samples and 85 selected features, providing a balanced representation of benign and malicious applications. The dataset is designed to enhance the performance and accuracy of Android malware detection models, as it focuses on the most informative features for this specific task. Each $\mathbf{j}^{th}$ recorded in the dataset correspond to the features. features are counted as $\boldsymbol{X_{cj}}$ for the input of the model. But deep learning models can handle only numeric data which should be prepared first. Also, $\boldsymbol{y_j}$ represents the label of each record, as formulated in equation 3.

$$\mathbf{Dc_j} = [\boldsymbol{x_{1j}}, \boldsymbol{x_{2j}}, \boldsymbol{x_{3j}}, \ldots\ldots, \boldsymbol{x_{bj}}, \mathbf{y_j}] \tag{9}$$

Where $\mathbf{Dc_j}$ is the dataset, $\boldsymbol{x_{bj}}$ represents the feature $\boldsymbol{b}$ in the $\boldsymbol{jth}$ iteration of dataset $\mathbf{Dc}$, $\mathbf{Y_j}$ is the last label record.

Figure 11 shows some a sample records from the Android malware dataset. The file is available in comma separated value (CSV) format.

Source IP	Source Port	Destination IP	Destination Port	Protocol	Timestamp	Flow Duration	Fwd Packets	Backward Packets	Length of Fwd	Length of Bwd	Packet Length	Label
10.42.0.151	43477	99.75.88.79	39698	17	15/6/2017 2:12	119888	1	1	18	26	18	Android_Adware
10.42.0.211	60774	99.239.230.180	58452	17	26/6/2017 12:43	540866	5	5	90	130	18	Android_Scareware
10.42.0.151	43477	99.199.0.77	62146	17	23/6/2017 1:14	144533	1	1	18	26	18	Android_Scareware
10.42.0.211	60774	98.29.23.144	8854	17	13/6/2017 8:35	86075	1	1	18	26	18	Android_Adware
10.42.0.151	43477	98.251.4.55	38811	17	15/6/2017 4:12	81190	1	1	18	26	18	Benign
10.42.0.151	43477	98.237.215.7	10700	17	30/6/2017 9:49	118734	1	1	18	26	18	Android_Scareware
10.42.0.151	43477	98.237.215.7	10700	17	30/6/2017 9:49	118734	1	1	18	26	18	Android_SMS_Malware
10.42.0.211	60774	98.206.108.200	13401	17	14/6/2017 11:40	575482	1	1	18	26	18	Android_Adware
10.42.0.211	60774	98.193.217.188	32582	17	11/7/2017 11:09	768741	5	5	90	130	18	Android_SMS_Malware
10.42.0.211	60774	98.173.234.99	51136	17	27/6/2017 3:39	129286	1	1	18	26	18	Android_Scareware
10.42.0.211	60774	98.160.231.183	43147	17	15/6/2017 5:18	141887	1	1	18	26	18	Android_Adware
10.42.0.211	48889	98.139.225.43	443	6	13/6/2017 11:55	15438974	9	9	823	4331	421	Android_Adware
10.42.0.211	48889	98.139.225.43	443	6	13/6/2017 11:55	902031	3	0	0	0	0	Android_Adware
10.42.0.211	58853	98.139.225.43	443	6	13/6/2017 4:47	16093884	9	10	918	4245	516	Benign
10.42.0.211	58853	98.139.225.43	443	6	13/6/2017 4:47	5050086	3	0	0	0	0	Benign
10.42.0.211	59040	98.139.225.43	443	6	13/6/2017 4:54	15600370	9	9	1163	4227	517	Android_SMS_Malware

Figure 11: Sample of records in Android Malware dataset

Android Malware Dataset is used to execute tests and determine the performance of the models. It's a medium dataset. As many Android Datasets consists of a total of static features and dynamic features. We chose the AMD (Android Malware Dataset) which contains the Android features related to the field of Android malware.

This dataset incorporates a combination of feature types, as we aim to capture a more accurate representation of Android malware features.

To facilitate supervised learning approaches for subsequent model training and evaluation, we assigned labels to each sample in the dataset, in order to classify them as either benign or malicious. This labeling enables the development of robust models for Android malware detection.

The Dataset AMD (Android malware dataset) encompasses a carefully selected 86 features that were deemed to be highly informative for the task of detecting and distinguishing between benign and malicious applications. By including a balanced representation of both types of applications, we sought to provide a realistic and representative dataset for training and evaluating malware detection models [3].

The primary goal of using this dataset is to enhance the performance and the accuracy of Android malware detection models. While focusing on the most informative

features specific to this task, we aim to provide researchers and practitioners with a valuable resource for advancing the field of Android security.

Finally, the AMD Dataset gives us the ability to create a binary classification model as well as multiclass classification model, as it gives us four label distributions for the multiclass: Android Adware, Android Scareware, Android SMS Malware and Benign.

Table 3: Malware Types of Classification in Multiclass

Malware type	Class
Adware	anomaly-based
Scareware	signature-based and anomaly-based
SMS Malware	anomaly-based

In Android systems, malware occurrences are typically more prevalent than benign instances, leading to imbalanced class proportions in the dataset. Handling imbalanced data is a challenge in machine learning. One approach to address this issue is either by under-sampling the majority class or by oversampling the minority class. In this context, the dataset records labeled as malware have been decreased to create a balanced AMD dataset. We will display the resulting number of samples and class labels in the balanced dataset.

Table 4: Android Malware Dataset Features Classification

Class Label	Number of Samples	Percentage
Benign	23700	50.0%
Adware	7900	16.6%
Scareware	7900	16.6%
SMS Malware	7900	16.6%

3.2 Data Preparation

Data Preparation of the AMD dataset goes into different steps as shown in Algorithm 2, after the data is loaded data cleaning starts to clean the dataset from duplicate, empty and corrupted samples, continuing with feature replacement, data discretization and scaling ending with the feature selecting of the most important feature for the view of android malware detection model.

Algorithm 2 Preparing Android Malware D dataset

Input: Selected subsets of AMD dataset $Dc \subset D_{AMD} = [Xc_{ij}, Yc_j]$

$with\ features\ as\ Xc_{ij} \rightarrow (b \times j)$

$Yc_j\ represents\ the\ label\ of\ the\ class \rightarrow (1 \times j)$

j: $records\ number, b$: $features\ number$

Output: Preprocessed AMD dataset

$Dp = D_{train} \cup D_{tt} = [Xp_{ij}, Yp_{ij}]$

$using\ a\ representation\ composed\ of\ preprocessed\ and\ carefully\ chosen\ features$

$Xp_{ij} \rightarrow (m \times j)$

$Yp_{ij}\ is\ the\ label\ of\ the\ sample \rightarrow (k \times j)$

- $k\ is\ the\ classes\ total\ number$

3.2.1 Feature Engineering

This involves the creation of new features derived from the existing ones. These newly created features can join to the dataset or replaced with some feature [52]. Additionally, some of the existing features may require transformation, like scaling, facilitating pattern recognition by machine learning algorithms [26]. For instance, in the case of Support Vector Classifier (SVC), a kernel function can be applied to map the original non-linear observations into a higher-dimensional space, where they become separable [25].

The significance of feature engineering lies in its ability to enhance the discriminatory power of machine learning classifiers, thus leading to improved prediction accuracy of the model. In scenarios like the AMD dataset, it becomes crucial to engineer new features that can capture the temporal representation.

Feature engineering is a critical step in the data preparation phase of any machine learning or data analysis project. Feature Engineering involves creating new features from the existing raw data that can potentially improve the performance of the models and provide more meaningful insights. In the context of the AMD dataset and detecting temporal patterns in malware attacks, three new features have been proposed which are s_sessiontime, r_sessiontime, and sr_sessiontime.

s_sessiontime: This feature represents the number of flows sent by a specific source IP address within a one-minute time window. It helps capture the outgoing flow frequency from different source IP addresses within short time intervals.

r_sessiontime: This feature represents the number of flows received by a specific destination IP address within a one-minute time window. It helps capture the incoming flow frequency to different destination IP addresses within short time intervals.

sr_sessiontime: This feature represents the number of flows sent and received by the same IP address within a one-minute time window. It helps capture the self-

communication patterns of a specific IP address, which might be relevant for detecting certain types of cyberattacks or malware synchronization.

These new features aim to provide temporal representations of the data by aggregating network flows within one-minute intervals. This time window has been chosen based on research indicating that it strikes a balance between capturing relevant data and avoiding excessive traffic capture. A shorter time window might not capture enough data, while a longer time window could lead to unnecessary and computationally expensive captures.

In the context of malware synchronization patterns for Android, based on research the use of a one-minute aggregation window width has been found to be sufficient for capturing relevant data while avoiding excessive traffic capture.

By using a one-minute time window, the system aggregates data and calculates the frequency of flows within that time interval. This allows researchers or security analysts to analyze the patterns of malware synchronization in Android devices effectively. A shorter time window might not provide enough data to detect the synchronization patterns accurately, while a longer time window could lead to capturing excessive network traffic, which may not be necessary for the analysis.

By striking a balance with the one-minute time window, the thesis aims to provide enough granularity to capture important synchronization patterns of malware while minimizing the amount of captured traffic, making it a practical and efficient approach for the evaluation of data in this context.

3.2.2 Data preprocessing

Regarding data preprocessing, the following steps have been applied:

a. Data Cleaning

This step involves searching for and handling missing values and duplicate instances in the dataset. Data cleaning ensures that the data is of high quality and reduces the chances of introducing bias or errors in the analysis.

b. Data Encoding

Categorical features are encoded into numerical representations so that deep learning algorithms can process them. In this case, the feature has been encoded to four Booleans. The source port and destination ports have been encoded into 51 Boolean features based on the port value, separating them into private or registered ports.

Index	Malware
0	Adware
1	Scareware
2	SMS Malware
3	Benign

One-Hot Code

Index	Adware	Adware	SMS Malware	Benign
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

$$T_{Encode} = f\left(\frac{T_k+1}{2}\text{x}65535\right), \qquad \text{T}_{Encode} = f\left(\frac{T_k+T_{max}}{2T_{max}}\text{x}65535\right) \tag{10}$$

3.2.3 Feature Replacement

The original source and destination address features have been changed with the newly engineered features (s_sessiontime, r_sessiontime, and sr_sessiontime). This transformation allows the model to capture the time-based patterns related to source and destination IP addresses more effectively.

3.2.4 Data Discretization

Data discretization, also known as binning, is a data preprocessing technique that involves converting continuous data into discrete intervals or bins. This process is useful for several reasons like dealing with continuous data, handling noise and interpretability.

In this AMD dataset, three continuous features have been discretized, they are: timestamp (ts), number of packets (pkt), and number of bytes (byt) are discretized by different techniques

- **Uniform Binning:** This method involves dividing the range of a continuous feature into equal-width intervals or bins. For example, if td ranges from 0 to 1000, and we want 5 bins, each bin would cover a range of 200 units (0-200, 201-400, 401-600, 601-800, 801-1000).
- **Quantile Binning**: In this approach, data is divided into bins based on quantiles. For instance, if we want 5 quantiles, each bin would contain an equal number of data points, and the range of values in each bin would be determined by the quantiles.

3.2.5 Scaling Data

After data discretization and encoding, the next step in the data preprocessing pipeline is data scaling. Data scaling is the process of standardizing or normalizing numerical features to bring them to a common scale. Scaling is crucial for many machine learning algorithms because it helps prevent features with larger magnitudes from dominating the model's training process.

Each input in the dataset is scaled alone. Commonly, using two scaling methods:

- **MinMaxScaler:** This method scales a value to a [0 1] range, as shown in eq. 4.

$$Xscaled = \frac{(X - minX)}{(maxX - minX)} \tag{11}$$

- **The Standard Scaler:** transforms input variables with a Gaussian distribution to have a unit standard deviation and zero mean.

After preprocessing the features for each sample, specifically within the realm of binary classification, the class labels are transformed into a solitary Boolean output variable. This variable assumes a value of 0 when the class label corresponds to “benign” and a value of 1 when the class label corresponds to “malware”.

3.2.6 Feature Selection

To identify the most informative features for the machine learning model, we used Mean Decrease in Impurity (MDI).

This feature selection approach operates as a filter method, separate from the deep learning classification model. Furthermore, the newly chosen feature count, denoted as '*m*', is smaller than the initial preprocessed features '*p*'. As a result, the modified input feature vector prepared for processing using DL, the model is depicted as shown in eq. 12.

$$\boldsymbol{X_{ij}} = [\boldsymbol{x_{1j}}, \boldsymbol{x_{2j}}, \boldsymbol{x_{3j}}, \ldots\ldots, \boldsymbol{x_{mj}}]^{\boldsymbol{T}} \tag{12}$$

Where $\boldsymbol{j = 1, 2, \ldots, n}$ number of dataset statements and $\boldsymbol{m}$ is total number of features.

The genuine class labels can be expressed as shown in equation 13 for the context of binary classification.

$$\boldsymbol{Y_{pj}} = \boldsymbol{y_j} \tag{13}$$

Figure 12 displays the highest-ranking numerical features from the AMD dataset. The higher the MDI value for a feature, the more important it is in predicting the target variable.

By using only, the most important features, the researchers aimed to create more efficient and accurate deep learning models while reducing the potential noise and irrelevant information that less important features might introduce. This feature selection step helps optimize the model's performance, minimize overfitting, and make the model more interpretable and manageable [56].

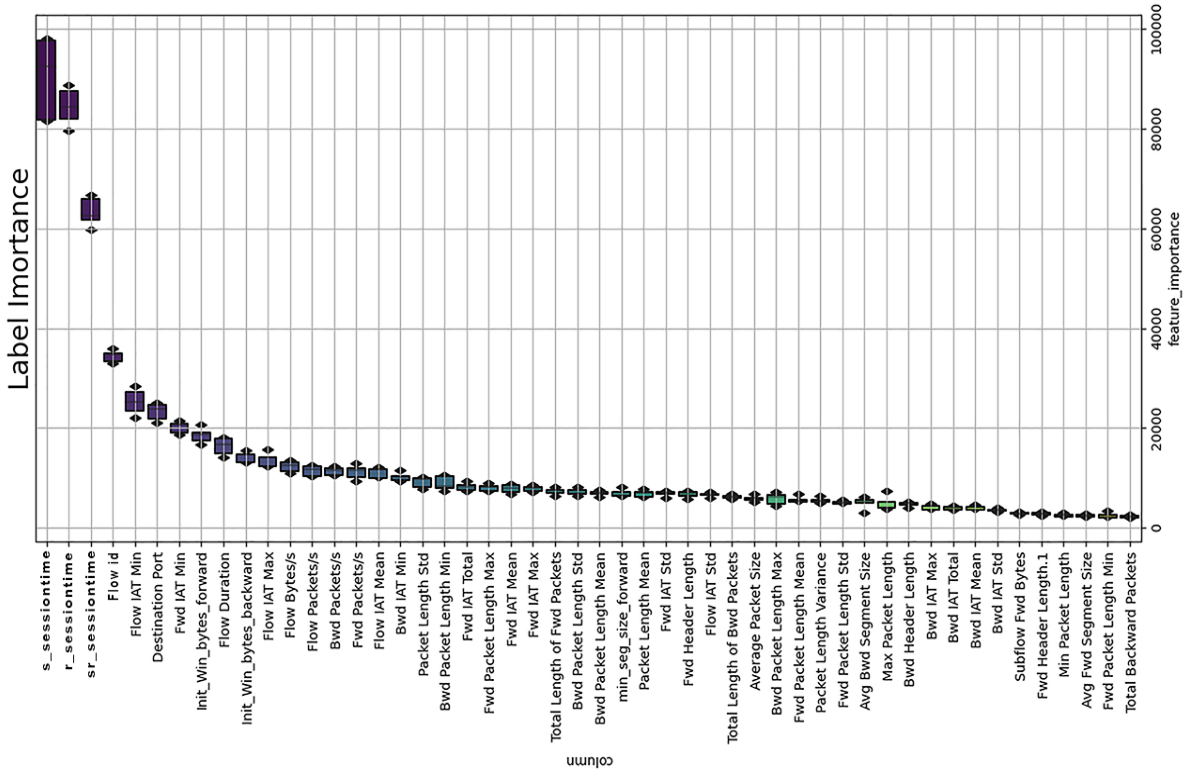


Figure 12: The Most important Features labels

3.3 HAMD-DNN Modeling

This section will represent that modeling process of the proposed model HAMD-DNN after the data collection and preparation, we will provide the design, training and evaluation for our proposed models (Binary and Multiclass model), figure below show again the modeling and framework of the HAMD-DNN models.

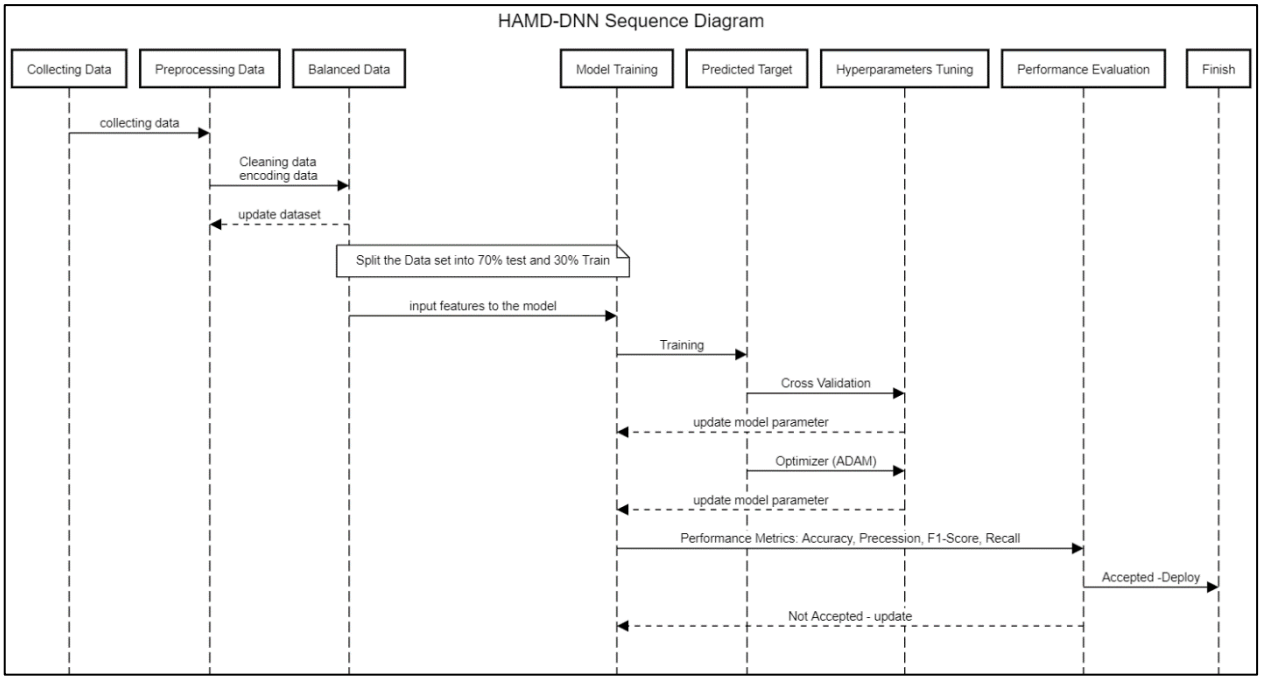


Figure 13: Proposed stages for HAMD-DNN

And algorithm 1: shows the stages of designing and applying our model.

Algorithm 1 Approach for applying HAMD-DNN

Inputs: Samples Extraction
Outputs: The Expected Record (0 Benign, 1 Malware)

Stage 1. Data Collection
Stage 2. Data Preparation
Stage 3. Split the dataset into train set and test set.
Stage 4. Designing the HAMD-DNN
Stage 5. Train HAMD-DNN model on training set.
Stage 6. Calculate the parameters of the model using the testing set.
Stage 7. Tuning the model parameters.
Stage 8. implement

3.2.1 HAMD-DNN Model Training and Validation

To calculate the parameters, HAMD is trained on known samples from the dataset. Through the training process, the model frequently adjusts its parameters to minimize the difference between the predicteds outputs and the actual labels in the training data. This is typically done using optimization algorithms like SGD, which update the parameters based on the gradients of the loss function with respect to the model's predictions. The training process continues until the model achieves satisfactory performance on the training data, effectively learning to make accurate predictions for the given task. Once the training is complete, we will be ready to run it on new and unseen data [57].

A. Train-Test Split

In order to ensure that a machine learning model can make accurate predictions on new, unseen data, it is customary to divide the dataset into training and test sets. The training set is utilized to train the model, while the test set serves to assess the model's predictive accuracy. For this particular thesis, the dataset was divided into 70% for training and validation, and 30% for testing purposes.

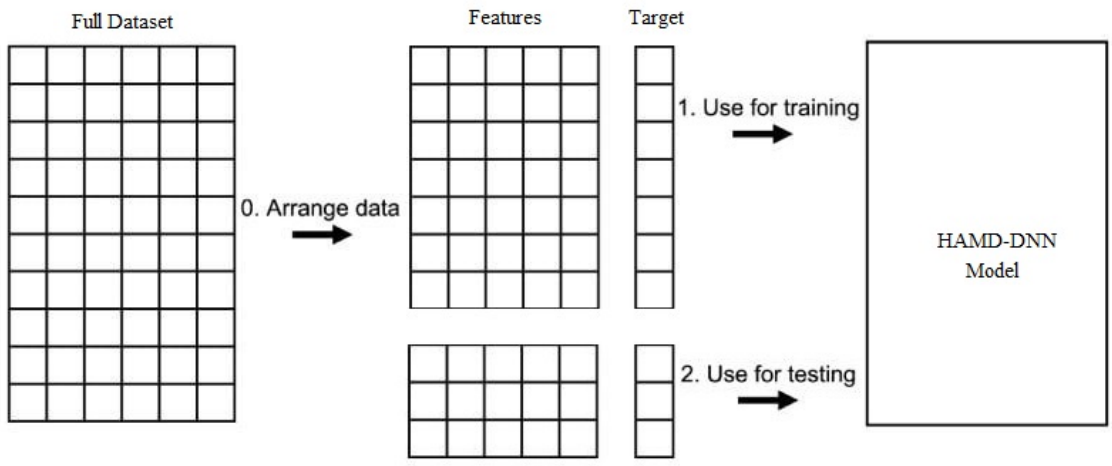


Figure 14: Train-Test Split Framework

This division of data is analogous to the challenge of detecting zero-day attacks, where the model is confronted with previously unseen labels. Consequently, this estimation allows us to gauge how effectively the model can handle novel anomaly network flows

Table 5: Illustrates the distribution of samples in the dataset

Class Label	Samples	Training Samples	Testing Samples
Benign	23708	16,596	7112
Android_SMS_Malware	67397	47178	20219
Android_Scareware	117082	81957	35125
Android_Adware	147443	103210	44233
Total Samples	355630	234941	120689

B. Cross Validation

To enhance the ability of a binary classification model to perform well on new, unseen data, the training set is divided into two parts: 70% for training the model and 30% for justification purposes. The classifier is then trained on the training set and its performance is evaluated on the validation set. During this process, after each epoch, the validation loss is monitored. If the val-loss does not increase, a counter is increased until it reaches the predefined limit, at a point to prevent overfitting the early stopping will be applied.

For multiclass model classification, a 10-fold cross validation approach is employed. This partitioning method ensures that the proportion of observations in each class remains the same in each fold. By doing so, the model is exposed to various subsets of the data, which helps reduce bias compared to other methods. Typically, k is set to 5 or 10, meaning the dataset is divided into k subsets. During each iteration, the model is trained on k-1 subsets and validated on the remaining subset. This process is repeated k times to ensure the model is exposed to all minor classes, particularly in imbalanced datasets where

class distribution is uneven. The goal of this sampling method is to mitigate overfitting and improve the model's ability to generalize when it encounters unseen data.

Within our thesis, a stratified 10-fold cross-validation methodology is implemented for the multiclass classifier. This approach ensures equitable representation of each class in both training and testing datasets, maintaining balanced proportions. For a visual illustration of the 5-fold cross-validation process, refer to Figure 15.

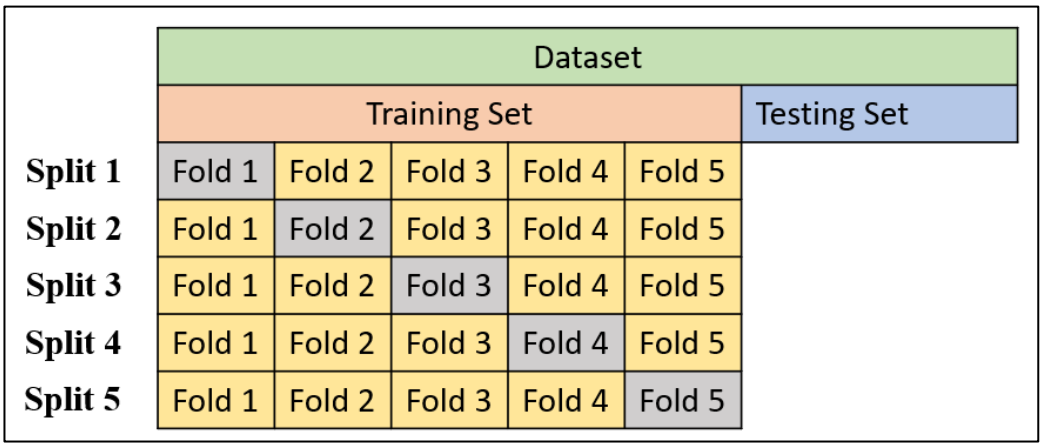


Figure 15: 5-Fold Cross-Validation Process

C. Loss Function

Once the DNN architecture is established, the subsequent step involves learning the model parameters from the labeled dataset. This learning process aims to align predicted outputs with actual labels. The training procedure encompasses feeding batches of data samples into the model during the forward pass, thereby computing the output. Following this, the disparity between the true labels and predicted results is evaluated using a loss function.

The primary objective of training is to minimize this loss function, thereby iteratively refining the model parameters to enhance prediction accuracy. This iterative cycle persists until the model attains a level of performance deemed satisfactory on the training data.

In the context of binary classifiers, the cross-entropy loss function, denoted by equation 14, is commonly employed.

$$\boldsymbol{Loss_{CE}} = -\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\mathbf{j=1}}^{\boldsymbol{N}} \boldsymbol{y_j} \cdot \mathbf{log}\left(\boldsymbol{p}(\boldsymbol{y_j})\right) + \left(\mathbf{1} - \boldsymbol{y_j}\right) \cdot \mathbf{log}\left(\mathbf{1} - \boldsymbol{p}(\boldsymbol{y_j})\right) \quad (14)$$

where $\boldsymbol{y_j}$ is 1 for malware and 0 for benign, $\boldsymbol{p}(\boldsymbol{y_j})$ is the predicted probability of the observation ($\boldsymbol{j}$) where N stands for the quantity of instances within the batch.

For the multiclass classifier, the categorical cross-entropy loss function is employed, as depicted in equation 15.

$$\mathbf{Loss_{CCE}} = -\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{j=1}}^{\boldsymbol{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{k}} \boldsymbol{y_{ji}} \cdot \mathbf{log}\left(\boldsymbol{p}(\boldsymbol{y_{ji}})\right) \quad (15)$$

where $\boldsymbol{y_{ji}}$ is the label for class ($\boldsymbol{i}$) and iteration ($\boldsymbol{j}$).

$\boldsymbol{N}$ is number of iterations in the batch, $\boldsymbol{p}(\boldsymbol{y_{ji}})$ is the expectation number of class ($\boldsymbol{i}$) and iteration ($\boldsymbol{j}$).

D. Optimizer: Adam

optimizing neural network model training and validation leads to the minimize the error or loss function during the training process. Neural networks learn by adjusting their weights and biases based on the error between the predicted outputs and the actual targets (ground truth). The optimizer is responsible for finding the optimal values for these weights and biases, thereby improving the model's performance in making accurate predictions on new, unseen data.

The algorithm of ADAM has been briefly mentioned below:

Stage 1: while w_t do not converges.
do{
Stage 2: Determine gradient $g_t = \frac{\partial f(x,w)}{\partial w}$
Stage 3: Determine $p_t = m_1 \cdot p_{t-1} + (1 - m_1) \cdot g_t$
Stage 4: Determine $q_t - m_2 \cdot q_{t-1} + (1 - m_2) \cdot g_t^2$
Stage 5: Determine $\hat{p}_{t=} \; p_t/(1 - m_1^t)$
Stage 6: Determine $\hat{q}_{t=} \; pq_t/(1 - m_2^t)$
Stage 7: Renew $w_t = w_{t-1} - \alpha . \hat{p}_t/(\sqrt{\hat{q}_t} + \in)$
}
Stage 8: end and return w_t

The optimizer performs the following key tasks:

- Minimizing errors: The optimizer helps to minimize the error between predicted and true values by adjusting the model's parameters during training.
- Gradient Descent: The optimizer uses gradient descent to find the best values for the model's parameters by following the direction of steepest descent on the loss surface.
- Learning Rate: The optimizer uses a learning rate to control how big the parameter updates should be in each step of the training.

- Minimization of Loss Function: The optimizer's goal is to find the set of model parameters that minimize the loss function, leading to improve model performance.

During validation, the trained model's fixed parameters are evaluated on new data to assess how well it generalizes to unseen examples.

In order to cover all the above tasks, Adam is the recommended optimizer as it's the most popular and efficient optimizer, adapting learning rates for different parameters and performing well in various neural networks models.

E. Regularization

To ensure optimal performance of the algorithm on unknown data, while preventing overfitting, regularization strategies are employed. These techniques introduce penalty terms to the cost function, reducing the weights and preventing extreme values, such as in L1 and L2 regularization. Among these strategies, dropout is the most commonly utilized in deep learning.

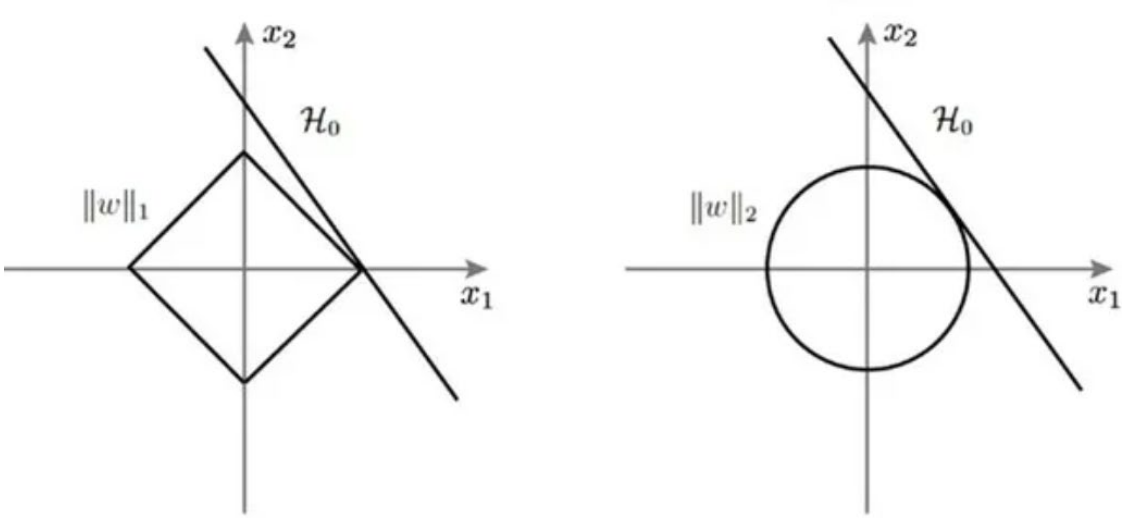


Figure 16: L1 - L2 Regression Curves

The dropout strategy involves randomly deactivating a fixed percentage of neurons in a network layer during a single gradient step. By doing so, the model is forced to rely on different subsets of neurons, promoting robustness and preventing over-reliance on specific features during training. The dropout rate, a hyperparameter, determines the percentage of neurons deactivating and can be assigned in a range of 10% to 50%.

Here, a dropout rate of 20% is applied to the second hidden layer and the third hidden layer, effectively encouraging diverse neuron activations and contributing to improved model generalization.

F. The Proposed Models Training

The binary deep neural network model is trained for 100 epochs using early stopping, a batch size of 100, and the cross-entropy loss function. The optimizer utilized is Adam, with a learning rate set to 0.001.

For the training and validation process, 70% of the dataset is used, while the remaining 30% is reserved for testing the model's performance on unseen data as presented in figure 17.

```
In [24]: result = model.fit(x=x_train,
                            y=y_train,
                            batch_size=batch_size,
                            epochs=max_epochs,
                            verbose=1,
                            #callbacks=[early_stopping],
                            validation_split=0.2)

Epoch 1/20
640/640 [==============================] - 7s 6ms/step - loss: 0.0787 - accuracy: 0.9701 - val_loss: 0.0051 - val_accuracy: 0.9
988
Epoch 2/20
640/640 [==============================] - 3s 5ms/step - loss: 0.0071 - accuracy: 0.9979 - val_loss: 0.0051 - val_accuracy: 0.9
977
Epoch 3/20
640/640 [==============================] - 3s 5ms/step - loss: 0.0024 - accuracy: 0.9992 - val_loss: 0.0023 - val_accuracy: 0.9
995
Epoch 4/20
640/640 [==============================] - 4s 6ms/step - loss: 0.0037 - accuracy: 0.9988 - val_loss: 0.0034 - val_accuracy: 0.9
994
Epoch 5/20
640/640 [==============================] - 4s 6ms/step - loss: 0.0025 - accuracy: 0.9991 - val_loss: 0.0021 - val_accuracy: 0.9
995
Epoch 6/20
640/640 [==============================] - 4s 6ms/step - loss: 0.0027 - accuracy: 0.9993 - val_loss: 0.0085 - val_accuracy: 0.9
967
```

Figure 17: Code sample of Proposed Model Training

The detailed sequence of steps for fitting the HAMD-DNN model is illustrated in Figure 18 as a flowchart.

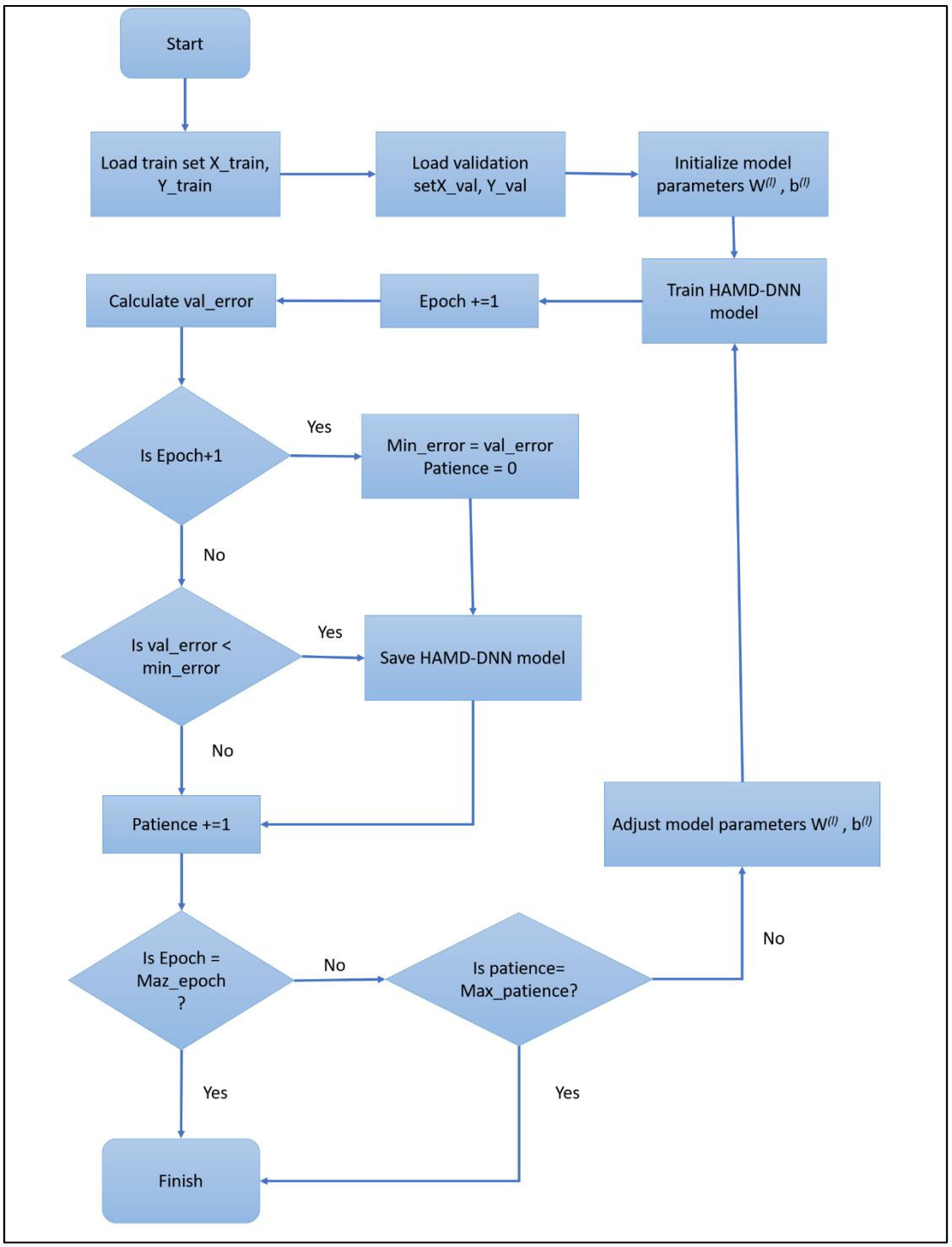


Figure 18: Flow chart of HAMD-DNN Model Training

3.2.2 HAMD-DNN Model Testing

Once we complete training, it's tested using previously unknown samples. By comparing the model's predicted values with the true labels, performance metrics are calculated to evaluate how well the model performs. If the test error, also known as generalization error, is higher than the training error, this indicates that the model is overfitting the training data, which means that it is excessively tailored to the specific characteristics of the training set and may not generalize well to new, unseen data.

The process flow of training and testing the proposed HAMD-DNN models is illustrated in Figure 20 in the next pages.

3.2.3 HAMD-DNN Model Hyperparameters Tuning

Prior to creating the architecture of HAMD models, several design decisions need to be considered, as illustrated in Figure 19.

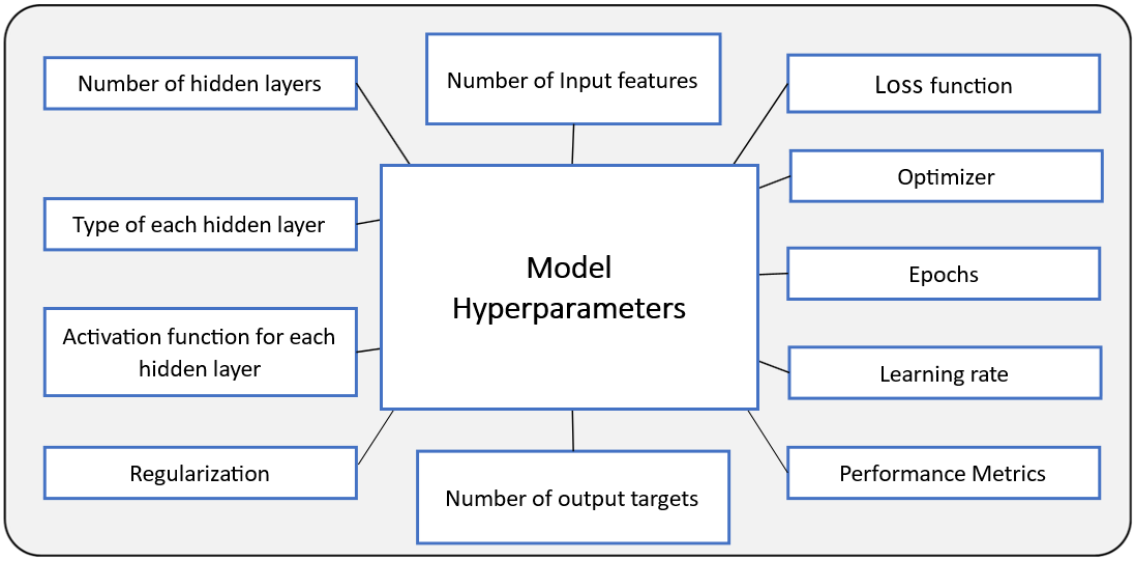


Figure 19: HAMD-DNN model's hyperparameters

Hyperparameters play a crucial role in the performance and generalization ability of a deep learning model. Hyperparameters are set before the training process and cannot be directly learned from the data like regular model parameters. Hyperparameter tuning, also known as hyperparameter optimization, is the process of finding the best set of hyperparameters for a given model and dataset. Some common hyperparameter tuning strategies are:

- Grid Search: Grid search involves defining a grid of hyperparameter values and exhaustively searching through all possible combinations of these values. It evaluates the model's performance for each combination and selects the one with the best results.
- Random Search: Random search involves randomly sampling hyperparameter values from predefined ranges. Unlike grid search, it does not search exhaustively, making it more efficient when dealing with a large number of hyperparameters.
- Learning Rate Schedulers: Instead of manually setting the learning rate, learning rate schedulers adjust it during the training based on predefined rules or heuristics. Common schedulers include step decay, exponential decay, and cyclic learning rates.
- Early Stopping: Early stopping is a technique where the training process is stopped once the model's performance on a validation set starts to degrade. This prevents overfitting and helps in finding the optimal number of training epochs.
- Cross-Validation: Cross-validation is a method to estimate a model's performance by dividing the dataset into multiple subsets and using each subset for validation while training on the others. It helps in obtaining more reliable estimates of hyperparameter performance.

  The selected hyperparameters for the proposed model are listed in Table 5.

Table 6: Selected Hyperparameters of The Model

Hyperparameter	Value
Optimizer	Adam (learning rate = 0.001)
Loss	Cross Entropy
Weight initialization	uniform
Regularization	Dropout 20% in 2-3 hidden layer
Epochs	100
Batch size	100

In binary classification, ReLU activation function is applied to the hidden layers, and the Adam optimizer is used with the default learning rate of 0.001. For binary classification, the sigmoid activation function is used. To prevent overfitting, the early stopping regularization technique is implemented, which stops training when updates no longer improve the model's performance on the validation set. Consequently, the optimal number of epochs is determined by the early stopping technique, ensuring an efficient and well-performing model.

Effective hyperparameter tuning can significantly improve a model's performance, making it crucial for achieving state-of-the-art results in deep learning tasks.

In this thesis, hyperparameters are defined based on literature recommendations. The ReLU activation function is used in hidden layers, and the Adam optimizer is employed. For binary classification, the Sigmoid activation function is utilized in the output layer. As for the Multiclass classification SoftMax activation function is used, the early stopping regularization technique is employed to halt training when parameter updates no longer improve the validation set, determining the optimal number of epochs.

To find the optimal learning rate, six values [0.0005, 0.001, 0.005, 0.01, 0.05, 0.1] are being tested in training the HAMD-DNN model with 50,000 samples and evaluating performance with 300000 samples. The best detection accuracy is achieved with a learning rate of 0.001 for the Adam optimizer, which is subsequently selected for training the model.

However, a smaller learning rate results in slightly longer training times compared to larger learning rates.

The batch size is another hyperparameter investigated using the same training samples, with batch sizes of [32, 64, 128, 256, 512, 1024, 2048]. The best performance is observed with a batch size of 512. Furthermore, larger batch sizes show slightly shorter training times compared to smaller batch sizes.

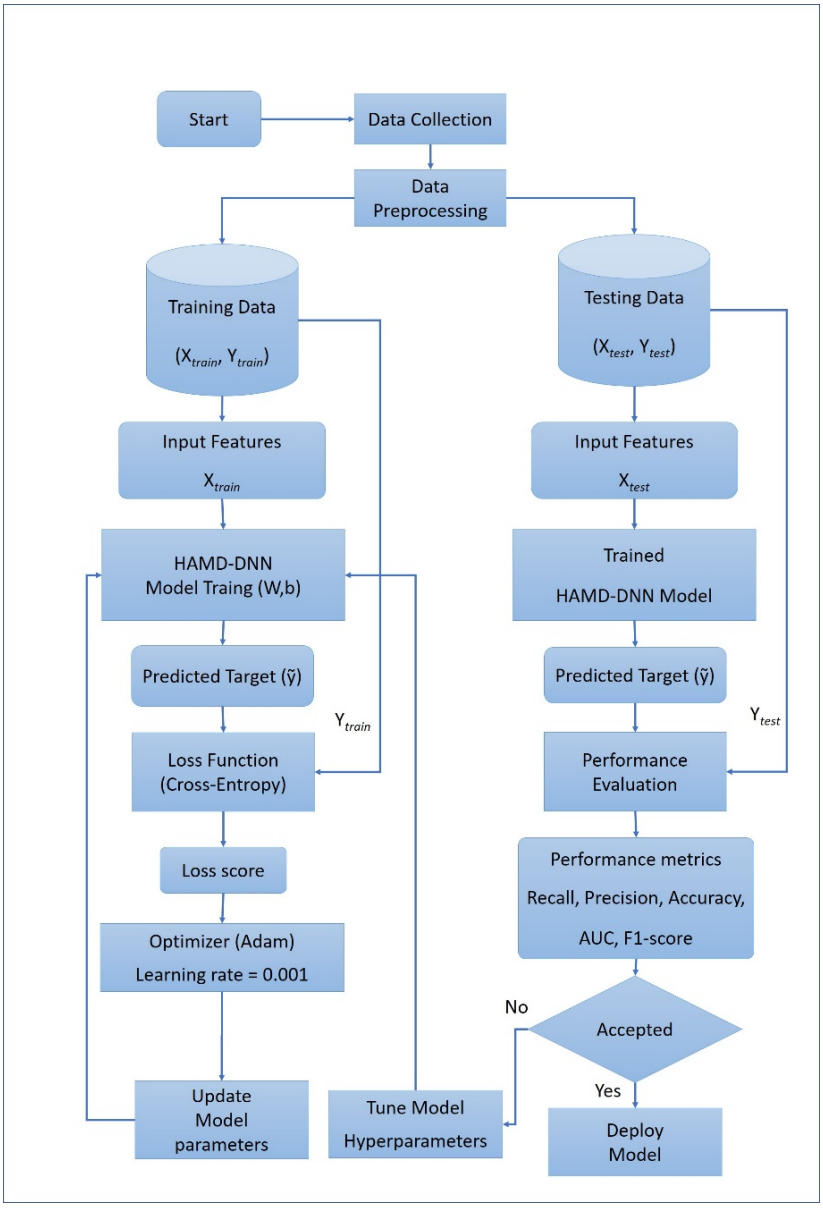


Figure 20: HAMD-DNN model training and testing process flow diagram

3.4 The Proposed HAMD-DNN Models

3.3.1 Deep Neural Network Classifier

Feedforward neural networks constitute a deep learning technique featuring an input layer, an output layer, and a minimum of two hidden layers. In the context of classification tasks, a feedforward network establishes a deterministic connection between the input feature vector X and the actual label Y. This connection can be represented as an approximating function, as demonstrated in equation 16.

$$\boldsymbol{Y} \approx \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{b}) \tag{16}$$

where $\boldsymbol{X}$ is the input vector of features, $\boldsymbol{Y}$ is the true class label, $\boldsymbol{W}$ is the weight matrix, and $\boldsymbol{b}$ is the bias vector. $\boldsymbol{W}$ and $\boldsymbol{b}$ are the model parameters that should be learned from the labeled observations. In a feedforward model information flows from the input through the intermediate computations in the hidden layers, and finally to the output $\boldsymbol{Y}$ without feedback connections [53].

The architecture of the proposed binary HAMD-DNN model is depicted in Figure 21 consists of one input layer, multiple hidden layers, and one output layer with sigmoid activation function.

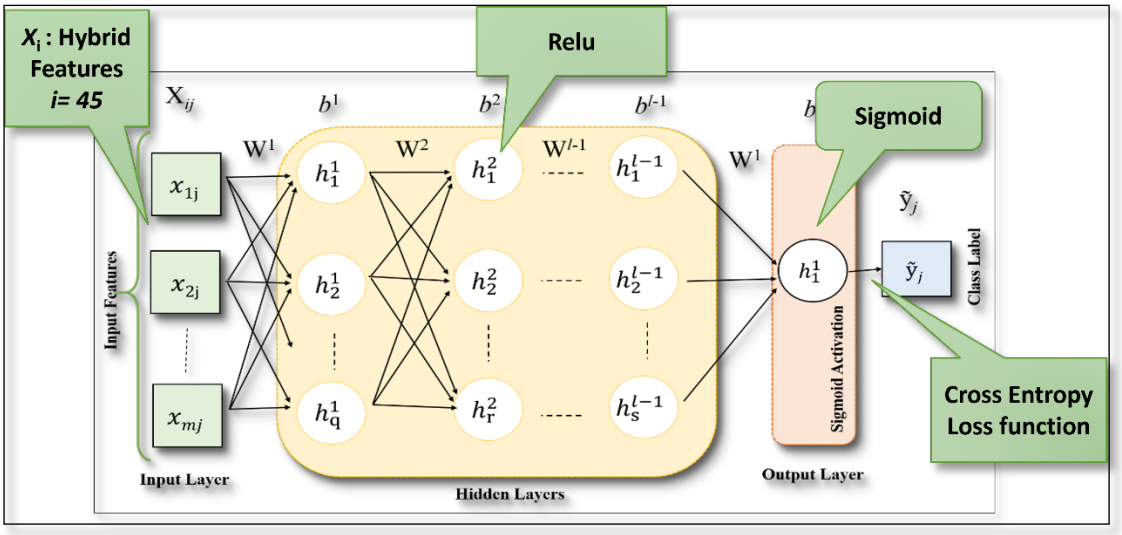


Figure 21: The proposed binary HAMD-DNN classifier architecture

Input Layer:

In a deep neural network, the input layer receives preprocessed and normalized input variables denoted as X_p. The dimensionality of the input layer is determined by the count of preprocessed and selected features. For example, if 'm' features are chosen following data preprocessing, the input feature vector for the 'jth' observation can be expressed as outlined in equation 17.

$$\boldsymbol{X_{ij}} = [\boldsymbol{x_{1j}}, \boldsymbol{x_{2j}}, \boldsymbol{x_{3j}}, \ldots\ldots, \boldsymbol{x_{mj}}]^{\boldsymbol{T}} \tag{17}$$

Hidden Layers:

Within a fully connected deep neural network, every hidden layer comprises neurons, also known as hidden units. Each neuron establishes connections with all nodes in the preceding layer and all nodes in the subsequent layer through links. These links are characterized by weights denoted as '$\boldsymbol{w}$,' while each neuron possesses an associated bias denoted as '$\boldsymbol{b}$.' The output of the initial neuron within the primary hidden layer for the '$\boldsymbol{j^{th}}$' observation is defined by equation 18.

$$\mathbf{h_{1j}^{1}} = \boldsymbol{\varphi}\left(\sum_{\mathbf{i=1}}^{\boldsymbol{m}} \boldsymbol{x_{ij}} \boldsymbol{w_{i1}} + \boldsymbol{b_1}\right) \tag{18}$$

where $\{\boldsymbol{x_{1j}}, \ldots, \boldsymbol{x_{mj}}\}$ is the input features vector of the $\boldsymbol{jth}$ observation. and $\{\boldsymbol{w_{11}}, \ldots\ldots, \boldsymbol{w_{m1}}\}$ are the weights for the links between input variables and first hidden unit, $\boldsymbol{b_1}$ is the bias of the first hidden unit, $\boldsymbol{\varphi}$ is the activation function and $\mathbf{h_{1j}^{1}}$ is the output of the first hidden unit.

Determining the model's hyperparameters, such as the count of hidden layers (representing the model's depth) and the quantity of neurons within each hidden layer (reflecting the model's width), relies on factors such as the number of input features, the complexity of the problem at hand, and the volume of training examples accessible.

To mathematically characterize the structure of a deep neural network, the weights within each layer are denoted as a matrix $\boldsymbol{W}$, and the biases for each layer are expressed as a vector $\boldsymbol{b}$. These weights and biases constitute the model's learnable parameters, which are acquired through the assimilation of labeled observations or instances. Additionally, an activation function is applied to each layer to nonlinearly transform the linear output of individual neurons. For the hidden layers, the activation function employed is the rectified linear unit (ReLU), as indicated in equation 19. This function activates neurons based on their output: when the output is below zero, the neurons remain inactive [55].

$$\boldsymbol{ReLU}(\boldsymbol{x}) = \mathbf{max}(\mathbf{0}, \boldsymbol{x}) \tag{19}$$

The output vector of the initial hidden layer encompasses the outputs of all hidden units within that layer, represented by equation 20.

$$\boldsymbol{H}^{(\mathbf{1})} = \mathbf{ReLU}(\boldsymbol{W}^{(\mathbf{1})\boldsymbol{T}}\mathbf{X} + \boldsymbol{b}^{(\mathbf{1})}) \tag{20}$$

Where: $\boldsymbol{q}$ is number of hidden units in first layer, $\boldsymbol{m}$ is number of input features.

$$\boldsymbol{H}^{(1)} = \begin{bmatrix} \boldsymbol{h}_1^{(1)} \\ \boldsymbol{h}_2^{(1)} \\ \vdots \\ \boldsymbol{h}_q^{(1)} \end{bmatrix},\ \boldsymbol{W}^{(1)T} = \begin{bmatrix} \boldsymbol{w}_{11}^{(1)} & \boldsymbol{w}_{21}^{(1)} & \dots & \boldsymbol{w}_{m1}^{(1)} \\ \boldsymbol{w}_{12}^{(1)} & \boldsymbol{w}_{22}^{(1)} & \dots & \boldsymbol{w}_{m2}^{(1)} \\ \vdots & \ddots & \ddots & \vdots \\ \boldsymbol{w}_{1q}^{(1)} & \boldsymbol{w}_{2q}^{(1)} & \dots & \boldsymbol{w}_{mq}^{(1)} \end{bmatrix},\ \boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_{1j} \\ \boldsymbol{x}_{2j} \\ \vdots \\ \boldsymbol{x}_{mj} \end{bmatrix}, \boldsymbol{b}^{(1)} = \begin{bmatrix} \boldsymbol{b}_1^{(1)} \\ \boldsymbol{b}_2^{(1)} \\ \vdots \\ \boldsymbol{b}_q^{(1)} \end{bmatrix}$$

In general, the output calculation for each hidden layer $\boldsymbol{H}^{(\boldsymbol{l})}$ is mathematically given as in equation 21.

$$\boldsymbol{H}^{(l)} = \boldsymbol{ReLU}(\boldsymbol{W}^{(l)T}\boldsymbol{H}^{(l-1)} + \boldsymbol{b}^{(l)}) \tag{21}$$

Where:

$(\boldsymbol{l})$ is the hidden layer number $\mathbf{1}, \dots, \boldsymbol{l}$.

$\boldsymbol{W}^{(l)T}$ is the weights matrix of the hidden layer.

$\boldsymbol{b}^{(l)}$ is the bias vector of the hidden layer.

$\boldsymbol{h}^{\mathbf{0}} = \boldsymbol{X}_{\boldsymbol{p}}$ the input features vector.

Therefore, the configuration of a deep neural network can be described as a sequence of composite functions, which can be expressed using equation 22.

$$\boldsymbol{H^{(l)} = H^{(l-1)}(H^{(l-2)}(\cdots H^{(1)}(X)))} \quad (22)$$

Output Layer:

For binary classification tasks, the ultimate fully connected layer within a deep learning model functions as the output layer. In this layer, only a single neuron is necessary, and the sigmoid activation function is employed to convert the output into a probability value spanning the range of 0 to 1, as depicted in equation 23.

$$\boldsymbol{sigmoid(x) = \frac{1}{1 + e^{-x}}} \quad (23)$$

Following the application of the sigmoid activation function to the output of the last hidden layer neuron, the resultant predicted score falls within the interval of 0 to 1. Subsequently, the determination of the predicted class label hinges on the utilization of a threshold in conjunction with the predicted score, as outlined in equation 24.

$$\hat{\boldsymbol{y}} = \begin{cases} \boldsymbol{0}, & \boldsymbol{\sigma(h_1^l) \leq threshold} \\ \boldsymbol{1}, & \boldsymbol{\sigma(h_1^l) > threshold} \end{cases} \quad (24)$$

Where $\boldsymbol{\sigma(h_1^l)}$ is the estimated score, 0 represents the normal class, and 1 represents the anomaly class.

3.3.2 Binary HAMD Model Setup

For binary classification, the objective is to classify data into two classes: Benign represented as 0's and the malware represented as 1's in the dataset. The proposed binary classifier is a deep Multi-Layer Perceptron (MLP) consisting of an input layer, three hidden layers, and an output layer. The input shape is 45, and since the 45 is the most effective features chosen for model construction.

The architecture of the MLP is as follows:

- The first hidden layer has 90 units with a Rectified Linear Unit (ReLU) activation function.
- The second dense layer has 60 units with ReLU activation, and a dropout regularization of 20% is applied to prevent overfitting.
- The third hidden layer consists of 45 units with ReLU activation.
- The output layer has one hidden unit with a sigmoid activation function.

The binary classifier's output, denoted as $\hat{y}$, predicts 1 if an anomaly is detected and 0 if the connection represents normal traffic. Figure 22 illustrates the design of the HAMD-DNN model. For binary classification, the sigmoid activation function is utilized to transform the output values between 0 and 1, providing the final classification decision.

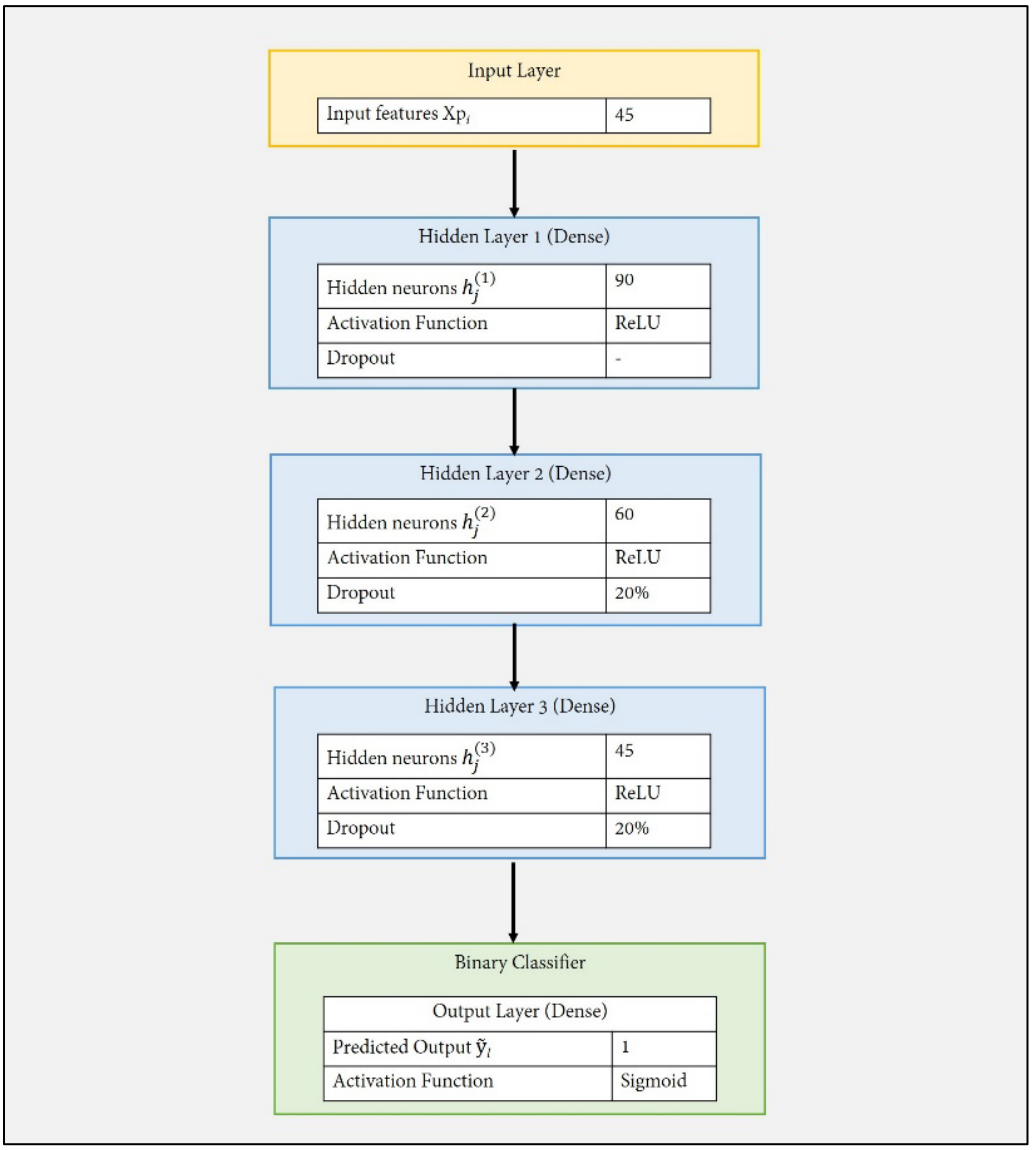


Figure 22: Design and Setup of HAMD-DNN Model

3.3.3 Multiclass HAMD-DNN Model

Android Malware Detection is used to analyze malicious activities in applications, anomaly-based HAMD are being used to build models of normal behavior of applications so that any behavior that deviates from normal behavior is considered as malware. However, HAMD generates a high false alarm rate when new patterns of normal behavior are detected and also it fails to detect attack patterns that are similar to normal patterns.

To solve the above problems, we propose in this thesis a model called Hybrid anomaly-based Android Malware Detection system using feedforward Deep Neural Networks (HAMD-DNN). The model takes the collected features as input and after the data is prepared, the deep learning algorithm classifies each application as either malware or benign. We trained the model now on a multiclass labeled feature from the AMD dataset to improve the model's detection accuracy and minimize false positive rate. Furthermore, new features were extracted to capture the temporal representation from the sequence of features flows using source IP and destination IP and timestamps of the flows. In addition, collected features are transformed and encoded, and important features are selected.

For a multiclass classifier, the output is a binary class label if the result is benign and multiclass label if the result is malware with an output of the malware type, which may include Android_Adware, Android_Scareware, or Android_SMS_Malware as shown in table 7.

The output of the proposed multiclass HAMD-DNN is a vector of dimension equals to number of classes as in equation 25.

$$\hat{\boldsymbol{y}} = [\hat{\boldsymbol{y}}_{\boldsymbol{1}}, \hat{\boldsymbol{y}}_{\boldsymbol{2}}, \dots, \hat{\boldsymbol{y}}_{\boldsymbol{k}}] \tag{25}$$

Where k is the number of classes (benign and malware classes), and each output is an estimated score between 0 and 1.

The neuron of the largest probability is 1 and others are zeros. For example, the background class in our model is represented as [1 0 0 0 0 0 0].

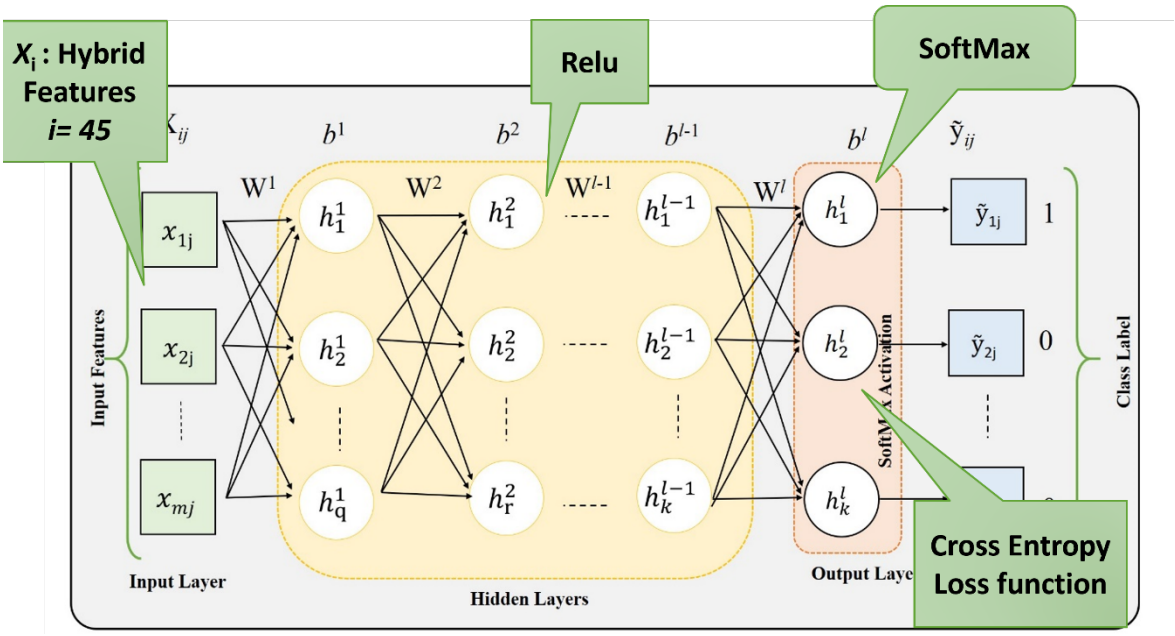


Figure 23: Architecture of Multiclass HAMD-DNN Classifier

Table 7: Model Outputs Comparison

	Number of Outputs	Output Classes
Outputs for Binary Classifier	2	Benign Malware
Outputs for Multiclass Classifier	4	Benign Android_Adware, Android_Scareware, Android_SMS_Malware

The SoftMax activation function is employed in multiclass classification scenarios. The technique of early stopping, a form of regularization, is utilized to halt training once updates to parameters cease to yield enhancements on a validation set. Consequently, the determination of the ideal number of epochs is governed by the early stopping method, elucidated comprehensively in Figure 19.

For multi classification, the proposed model is designed to classify the behavior into four categories. The normal connections are classified as benign and the unique labels for malware are Android_Adware, Android_Scareware, Android_SMS_Malware.

Figure 20 illustrates a deep feed-forward neural multiclass classifier, comprising an input layer, three hidden layers, and an output layer. The input layer has a shape of 45, as the model is constructed using the 45 most effective features. The first hidden layer consists of 90 units and utilizes the Rectified Linear Unit (ReLU) activation function. In the second dense layer, which contains 60 units, ReLU is also used as the activation function, and a dropout regularization of 20% is implemented to prevent overfitting. The third hidden layer is comprised of 45 units with ReLU activation. Finally, the output layer encompasses seven neurons and employs the SoftMax activation function.

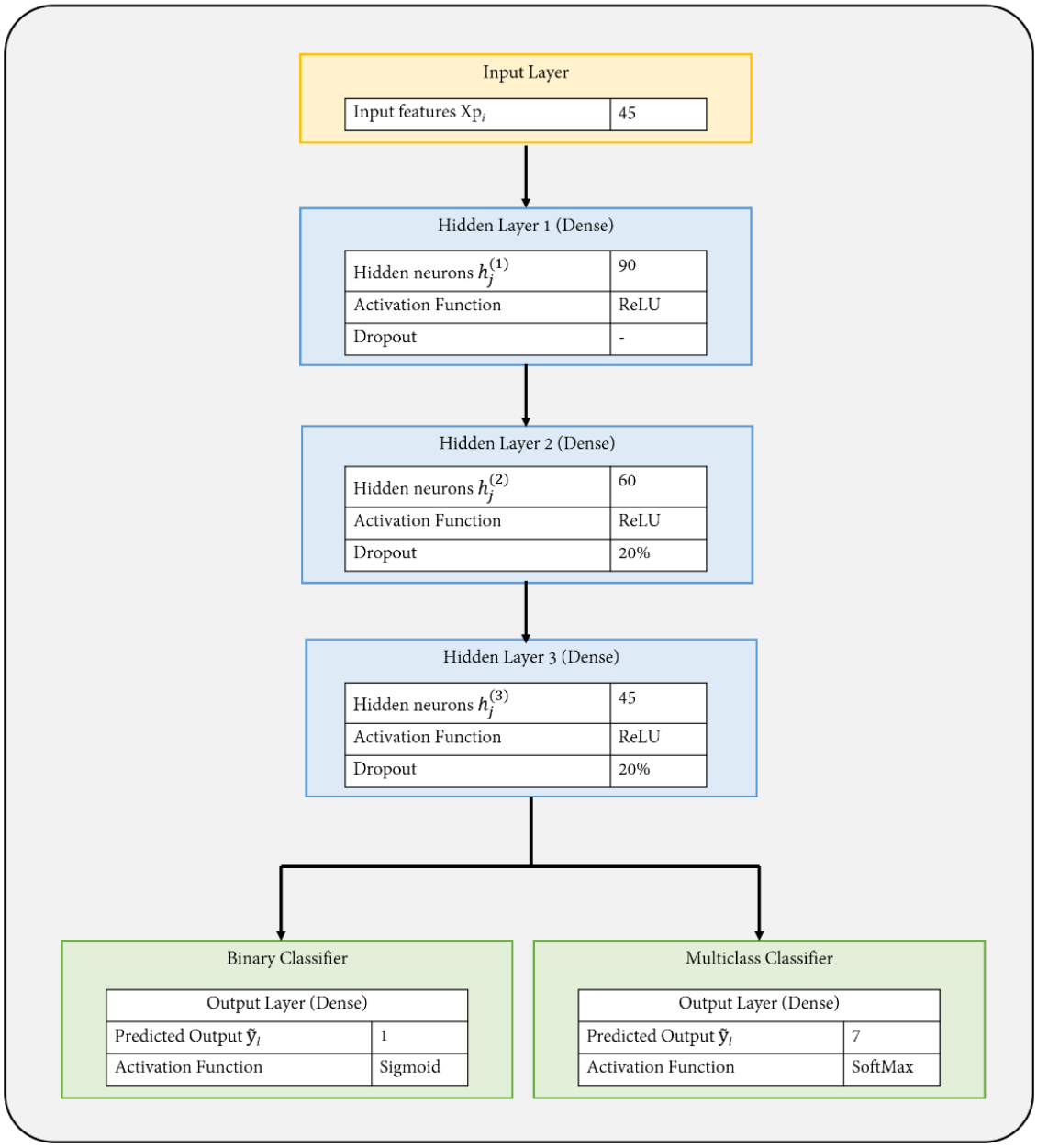


Figure 24: Design and setup of HAMD-DNN model

The multiclass deep neural network (DNN) model undergoes training for 15 epochs using a batch size of 100. The categorical cross-entropy loss function is employed along with the Adam optimizer having a learning rate of 0.001. Both training and 10-fold cross-validation are conducted on 70% of the dataset, while the remaining 30% is reserved for testing. Below is a Python code example, Figure 21 demonstrating how to fit the multiclass model using stratified 10-fold cross-validation.

```
import numpy as np
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score
from keras.optimizers import Adam

# Define your multiclass DNN model
def create_model():
    model = Sequential()
    model.add(Dense(64, input_dim=X.shape[1], activation='relu'))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy',
optimizer=Adam(lr=0.001), metrics=['accuracy'])
    return model

# Convert class labels to one-hot encoded vectors
num_classes = np.max(y) + 1
y_encoded = keras.utils.to_categorical(y, num_classes)

# Initialize stratified k-fold cross-validation
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

# Lists to store accuracy values for each fold
accuracy_scores = []

# Perform 10-fold cross-validation
for train_index, test_index in kfold.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y_encoded[train_index], y_encoded[test_index]

    # Create the model
    model = create_model()

    # Train the model
    model.fit(X_train, y_train, epochs=100, batch_size=100, verbose=0)

    # Evaluate the model on the test data
    y_pred = model.predict_classes(X_test)
    accuracy = accuracy_score(np.argmax(y_test, axis=1), y_pred)
    accuracy_scores.append(accuracy)

# Calculate the mean and standard deviation of the accuracy scores
mean_accuracy = np.mean(accuracy_scores)
std_accuracy = np.std(accuracy_scores)

print("Mean Accuracy: {:.2f}%".format(mean_accuracy * 100))
print("Standard Deviation: {:.2f}".format(std_accuracy))
```

3.5 Summary

This chapter introduced the key components involved in constructing models for HAMD using deep learning. The process starts with the extraction, transformation, selection, preprocessing, and scaling of features from the labeled dataset. Subsequently, the deep neural network learns model parameters from the preprocessed data during training. After training, the model's performance is evaluated, and hyperparameters are tuned to enhance the models' effectiveness.

This chapter provides a comprehensive outline of the methodology employed to develop and train HAMD (Android Malware Detection) models using the Android Malware benchmark dataset. To address memory and time limitations, a carefully chosen subset was utilized from the extensive dataset. The training and validation phases were conducted on 70% of this selected subset, with the remaining 30% reserved for testing on unseen data. Before commencing the training process, the data underwent thorough preparation, and essential features were carefully selected. Additionally, the model's hyperparameters were fine-tuned to maximize the efficacy of the proposed models.

As we expose the two models classification, we found that the multiclass classification is better than the binary as it shows that exact malware type ether adware, SMS malware or scareware, compared with the binary classification which gives an output of ether malware or benign only.

Chapter Four: Experimental Results of the HAMD Model

Chapter Four: Experimental Results of the HAMD Model

This chapter presents the experimental setup, the performance metrics used to evaluate the proposed models, and the experimental results of the HAMD models.

4.1 Experimental Setup

Resources used to develop the HAMD-DNN Model

- Anaconda Open-source environment with Python 3.10.9.
- Keras, the deep learning library.
- TensorFlow 2.3 [23].

The experiments runs on a HP ProBook laptop computer Core i7 CPU and 16 GB RAM. Anaconda with free Jupyter notebook environment.

4.2 Performance Metrics

To evaluate the performance of a classifier several metrics are used for model classification and multiclass classification models.

Detection rate, also known as Recall or True Positive Rate (TPR), measures the proportion of true positive predictions out of the total actual positive instances in the dataset. It is a fundamental performance metric in binary classification tasks, especially when the focus is on identifying positive cases correctly.

Detection Rate (Recall/TPR) is calculated using the following equation:

$$\mathbf{recall} = \frac{\mathbf{TP}}{\mathbf{TP} + \mathbf{FN}} \tag{26}$$

Where,

True Positive (TP): The number of instances correctly predicted as malware.

False Negative (FN): The number of instances incorrectly predicted as benign.

False Positive (FP): The number of instances incorrectly predicted as malware when they are benign.

True Negative (TN): The number of instances correctly predicted as benign.

The False Positive Rate (FPR) performance metric used in binary classification tasks. It measures the proportion of negative instances that are incorrectly classified as positive by the model. In other words, it quantifies how often the model makes false positive predictions out of the total actual negative instances.

The False Positive Rate is calculated using the following equation:

$$\mathbf{FPR} = \frac{\mathbf{FP}}{\mathbf{TN} + \mathbf{FP}} \tag{27}$$

Accuracy is a widely used metric to evaluate classification models. It calculates the percentage of correctly classified instances out of the total instances in the dataset.

$$\mathbf{Accuracy} = \frac{(\mathbf{TP} + \mathbf{TN})}{(\mathbf{TP} + \mathbf{TN} + \mathbf{FP} + \mathbf{FN})} \tag{28}$$

Precision measures the proportion of true positive predictions out of the total positive predictions made by the model. It is useful when the cost of false positives is high.

$$\mathbf{Precision} = \frac{\mathbf{TP}}{\mathbf{TP} + \mathbf{FP}} \tag{29}$$

The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall when both are important. It is especially useful when there is an imbalance between the classes.

$$\mathbf{F1 - score} = \frac{\mathbf{2 * precision * recall}}{\mathbf{precision + recall}} \tag{30}$$

AUC (Area Under the Curve) and ROC (Receiver Operating Characteristic) curve are performance evaluation metrics commonly used in binary classification tasks. They are particularly useful when dealing with imbalanced datasets or when it's essential to understand a model's performance at various classification thresholds.

The ROC curve is a graphical representation of the True Positive Rate (Recall) plotted against the False Positive Rate (FPR) at various classification thresholds.

The x-axis represents the FPR (1 - Specificity), and the y-axis represents the TPR (Sensitivity or Recall).

The curve is created by changing the classification threshold of the model and calculating the corresponding TPR and FPR values.

The ROC curve visually shows how the model's performance changes as you adjust the threshold for classifying positive and negative instances.

A perfect classifier's ROC curve would pass through the top-left corner (TPR = 1, FPR = 0), while a random classifier's curve would follow the diagonal line (45-degree line).

AUC-ROC (Area Under the ROC Curve) quantifies the overall performance of the classifier, where an AUC value of 1 represents a perfect classifier, and an AUC of 0.5 indicates a random classifier.

AUC is a scalar value representing the area under the ROC curve.

It ranges from 0 to 1, where 0.5 corresponds to random guessing, and 1 indicates a perfect classifier.

AUC provides a single value to summarize the classifier's performance across all possible classification thresholds.

A higher AUC suggests a better classifier, as it means the model can distinguish between positive and negative instances more effectively across different thresholds.
In order to understand the competence of the results of each experiment, the general analysis of the AUC is as follows

- If AUC = 0.5: The model is no better than random guessing.
- If 0.5 < AUC < 1: The model is better than random, and a higher AUC indicates better discrimination power.
- If AUC = 1: The model perfectly distinguishes between positive and negative instances.

The ROC curve and AUC are especially useful when the data is imbalanced, i.e., one class heavily outnumbers the other, as they provide insights into the classifier's performance across different operating points. However, it's crucial to consider the specific context and domain when interpreting ROC and AUC values, as the evaluation goals may vary depending on the application.

The confusion matrix is a table that summarizes the performance of a classification model by showing the counts of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions. It is a crucial tool for evaluating the performance of a classification algorithm, especially in binary classification tasks, as illustrated in Table 7.

Table 8: Confusion matrix

		True class	
		Attack	Normal
predicted class	Malware	TP	FP
	Benign	FN	TN

By analyzing the values in the confusion matrix, you can gain insights into how well the classification model is performed in terms of its accuracy, precision, recall, and other performance metrics.

4.3 Experimental Results and Analysis

The subsequent sections will show and examine the experimental findings of both Binary and Multiclass HAMD models.

4.3.1 Binary HAMD Classifier Results

There were four experiments conducted for the HAMD-DNN binary model architecture:

Experiment #1: One input layer, four hidden layers with [360, 180, 90, 45] hidden units, ReLU activation function for each layer, and one output layer with one neuron and sigmoid activation function.

The overall architecture can be summarized as follows:

Input Layer -> 360 neurons (ReLU activation) -> 180 neurons (ReLU activation) -> 90 neurons (ReLU activation) -> 45 neurons (ReLU activation) -> 1 neuron (Sigmoid activation) -> Output

Experiment #2: One input layer, three hidden layers with [90, 60, 45] hidden units, ReLU activation function for each layer, and one output layer with one neuron and sigmoid activation function.

The overall architecture can be summarized as follows:

Input Layer -> 180 neurons (ReLU activation) -> 90 neurons (ReLU activation) -> 45 neurons (ReLU activation) -> 1 neuron (Sigmoid activation) -> Output

Experiment #3: One input layer, two hidden layers with [90, 45] hidden units, ReLU activation function for each layer, and one output layer with one neuron and sigmoid activation function.

The overall architecture can be summarized as follows:

Input Layer -> 90 neurons (ReLU activation) -> 45 neurons (ReLU activation) -> 1 neuron (Sigmoid activation) -> Output

Experiment #4: One input layer, one hidden layer with [45] hidden units, ReLU activation function, and one output layer with one neuron and sigmoid activation function.

The overall architecture can be summarized as follows:

Input Layer -> 45 neurons (ReLU activation) -> 1 neuron (Sigmoid activation) -> Output

Among these experiments, the best accuracy was achieved in experiment # 2, which used three hidden layers with output shapes [90, 60, 45] and one output layer with one neuron and sigmoid activation function. Therefore, the proposed model architecture selected for HAMD is based on experiment #2.

The HAMD-DNN model was trained using 80% of the train set and cross-validated with the remaining 20% of the train set. To assess the model's performance, it was evaluated on an unseen test set. State-of-the-art metrics were utilized for evaluation and comparison.

Table 8 and Figure 25 displays the test results of the HAMD-DNN model when different DNN architectures were experimented with. The best-performing proposed model consisted of three layers, comprising one input layer, three hidden layers, and one output layer, with hidden units of (90, 60, 45) in each respective layer. This model achieved an impressive detection accuracy of 99.9%.

Table 9: Performance Evaluation Metrics Score for HAMD-DNN using 45 Input Features

Architecture	Accuracy	Precision	Recall	F1 Score
Four Hidden Layers	99.7688	99.7688	99.7678	99.7678
Three Hidden Layers	99.8329	99.8329	99.8329	99.8329
Two Hidden layers	99.7788	99.7788	99.7788	99.7788
One Hidden layers	99.6317	99.6317	99.6317	99.6317

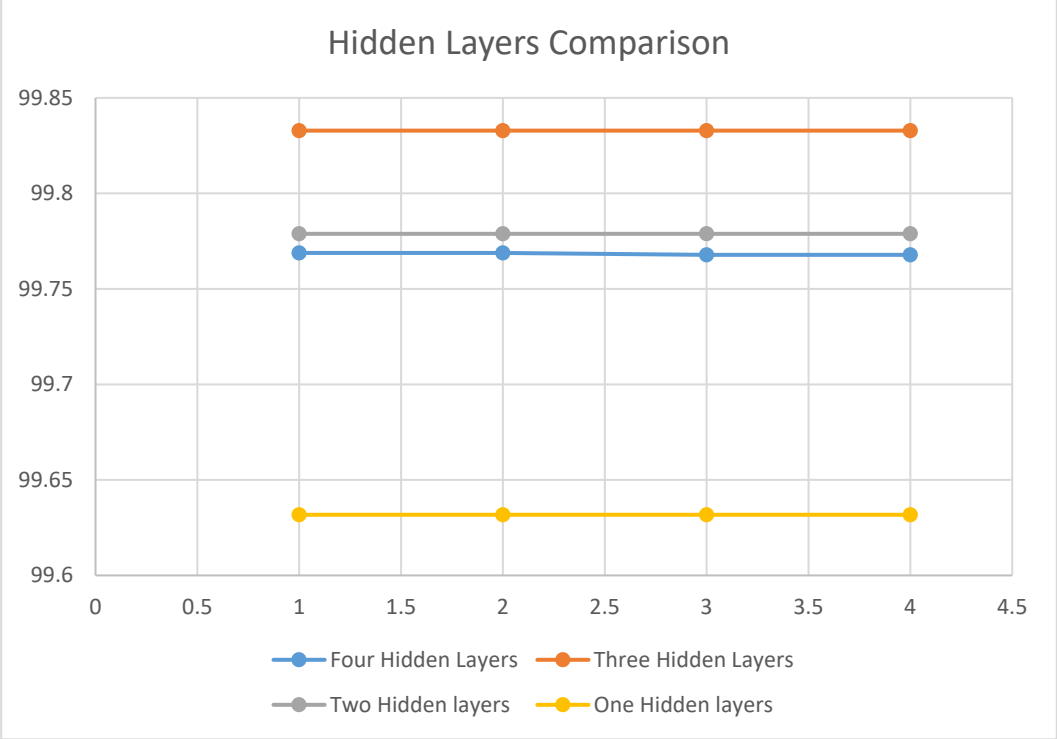


Figure 25: Hidden Layers Comparison

Figure 26 shows the training and validation loss. While training the model for 100 epochs, the loss converges to approximately 0.002.

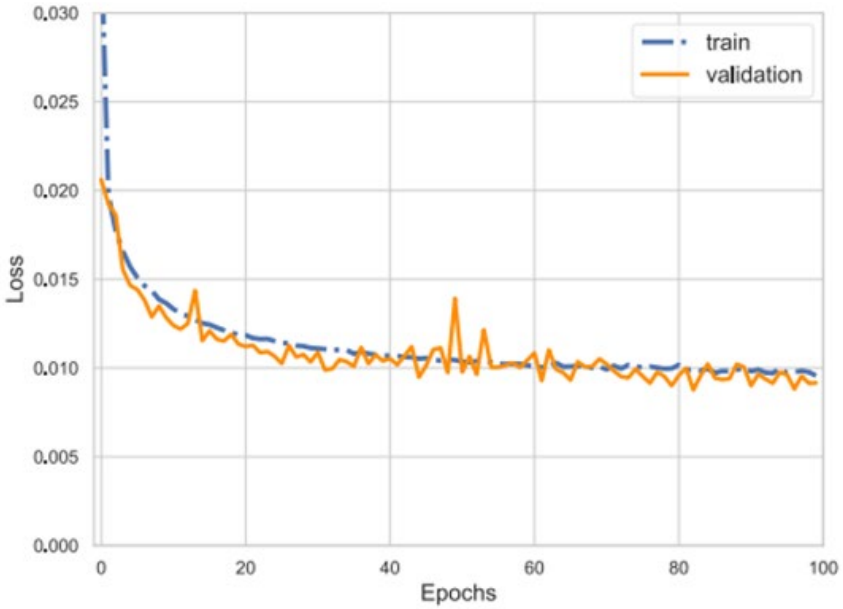


Figure 26: Training and Validation Cross-Entropy Loss Curve for HAMD-DNN

Figure 27 shows the accuracy of the model during training and validation of the model for 100 epochs. The model accuracy converges to around 99.94 %.

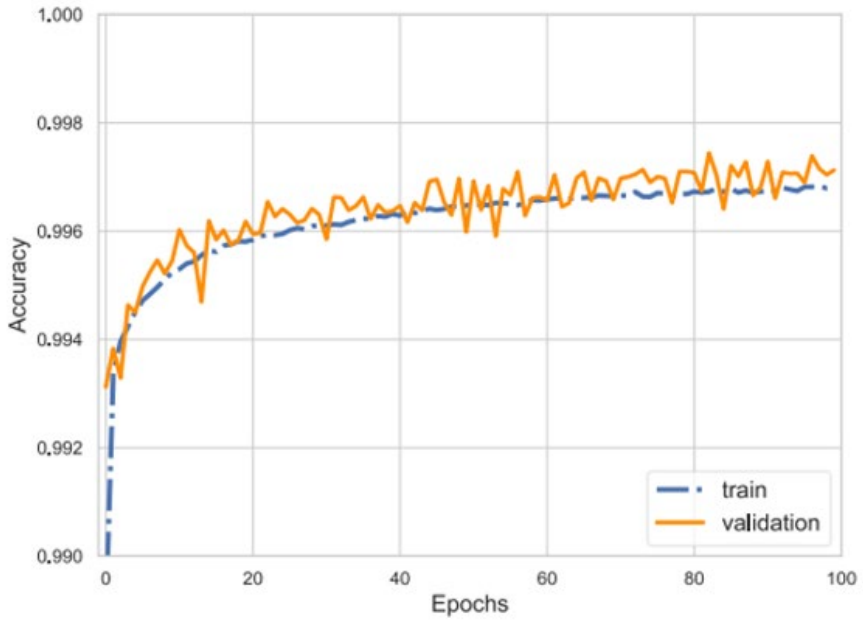


Figure 27: Training and Validation Accuracy Curve for HAMD-DNN

Table 10 summarizes the results in a percentage of the performance evaluation metrics for the binary deep learning classifier when trained on 45 features, 28 features, and 20 features for each class. The highest F1-score is 99.844% for the model when trained on 45 input features.

Table 10: Performance Evaluation Metrics Score for Different Number of Input Features

Binary HAMD-DNN	Accuracy	Precision	Recall	F1 Score
15 features	93.592	94.346	93.592	94.589
28 features	98.766	99.512	98.766	98.766
45 features	**99.844**	**99.844**	**99.844**	**99.844**

Figure 28 illustrates that HAMD-DNN achieves consistent performance metrics between 96% to 97% when trained using only 20 basic features. Moreover, there is a notable enhancement of approximately 2% to 2.5% in all performance metrics when the model is trained on 28 uncorrelated features. However, using 45 selected features results in only a marginal increase of about 0.4% to 0.5% in performance.

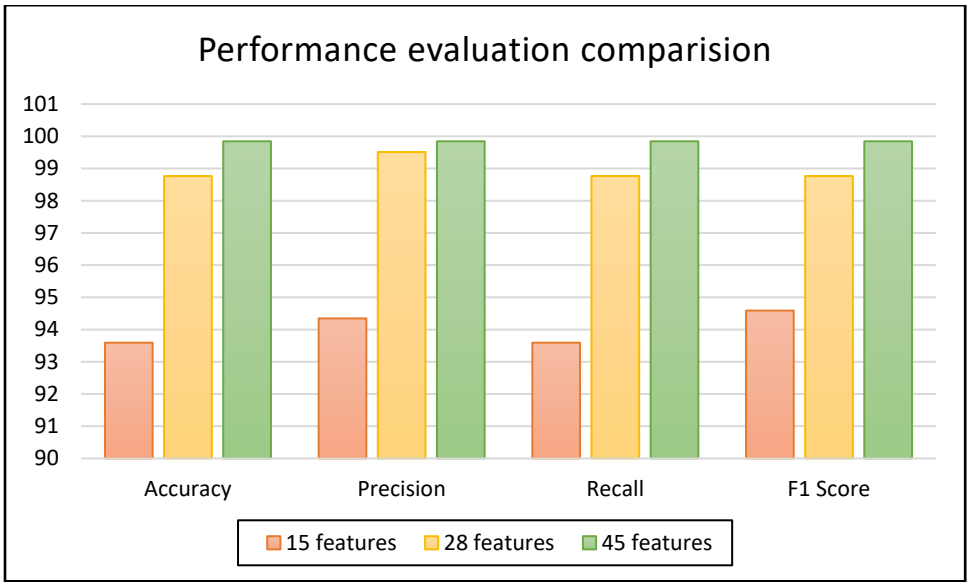


Figure 28: Performance Evaluation Metrics for Binary HAMD-DNN Classifier for Different Number of Features

Figure 29 shows the confusion matrices for the proposed HAMD-DNN models when trained on 20, 28, 45 of input features respectively and tested on unseen data of the derived dataset.

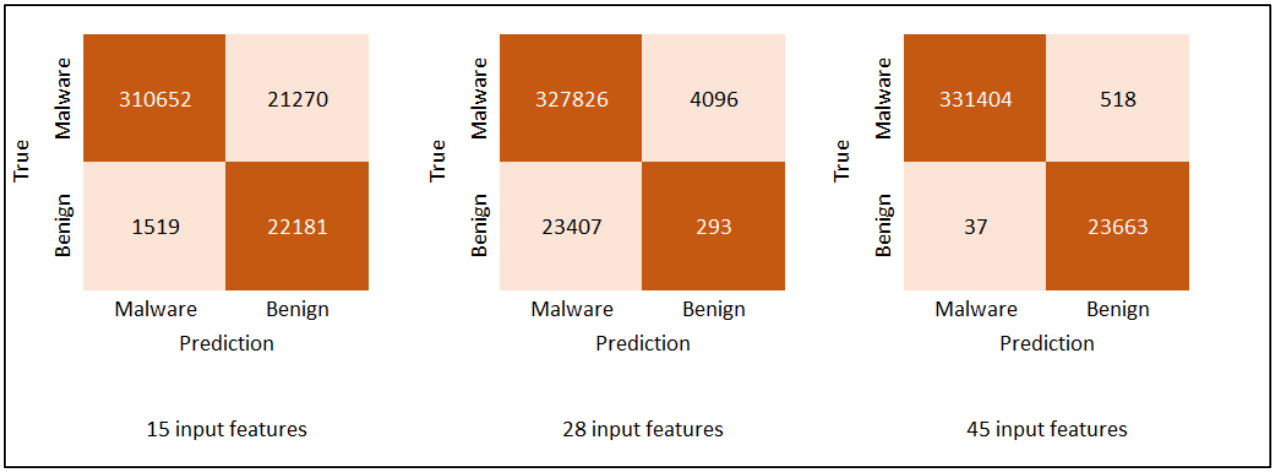


Figure 29: Confusion Matrices for Binary HAMD-DNN

When using the minimal number of features, the classifier had the highest number of incorrect detections, particularly in the form of false negatives. However, the detection accuracy showed a remarkable improvement when the classifier was trained with the most effective 45 features. Similarly, false positives were observed, where normal connections were incorrectly classified as malicious, but this issue was alleviated to some extent by incorporating the 45 features. Notably, employing the 45 features instead of 28 features resulted in a modest yet notable enhancement in both precision and recall, underscoring the significance of selecting more informative features for training the classifier.

It appears that the AUC values for the model are as follows:

AUC = 0.9999 when trained with 45 features.

AUC = 0.9914 when trained with 28 features.

AUC = 0.9831 when trained with 15 features.

Since a higher AUC indicates better separability, the model achieves its highest AUC which is almost equal 1 when trained with 45 effective features. This suggests that the model has optimal separability and performs excellently when trained with these 45 features.

We plot the curves by using the following code:

```
# Example data
y_true = [1, 0, 1, 1, 0, 0, 1, 0, 1, 0]
y_probs = [0.9, 0.7, 0.6, 0.8, 0.4, 0.3, 0.95, 0.2, 0.85, 0.1]

# Calculate the ROC curve
fpr, tpr, thresholds = roc_curve(y_true, y_probs)

# Calculate the AUC (Area Under the Curve)
roc_auc = auc(fpr, tpr)

# Plot the ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()
```

Below, figures 30, 31, 32 shows the ROC curves that we draw using python for the three different AUC values of 15, 28, 45 features

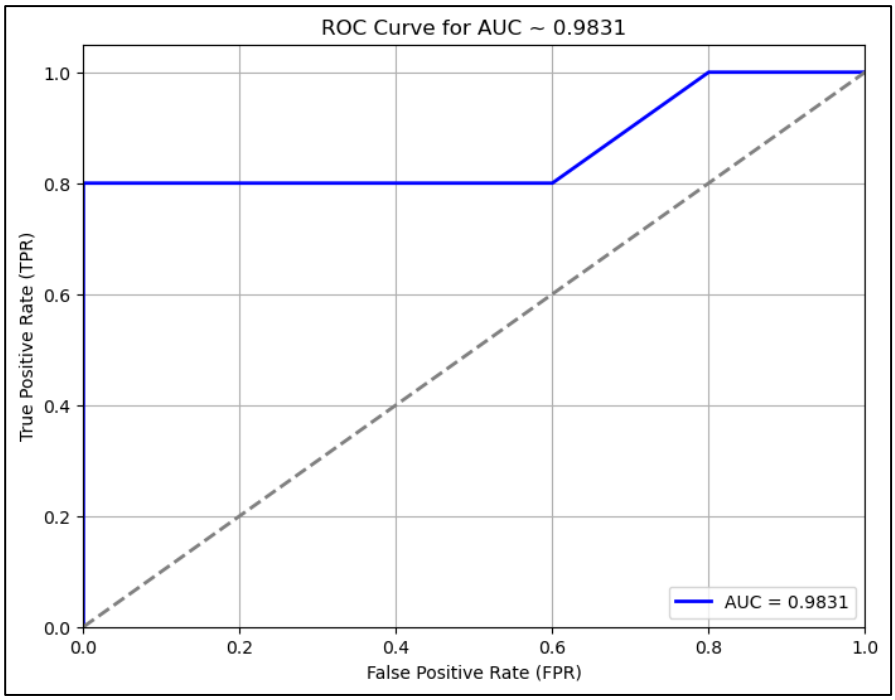


Figure 30: ROC Curve for 15 Features

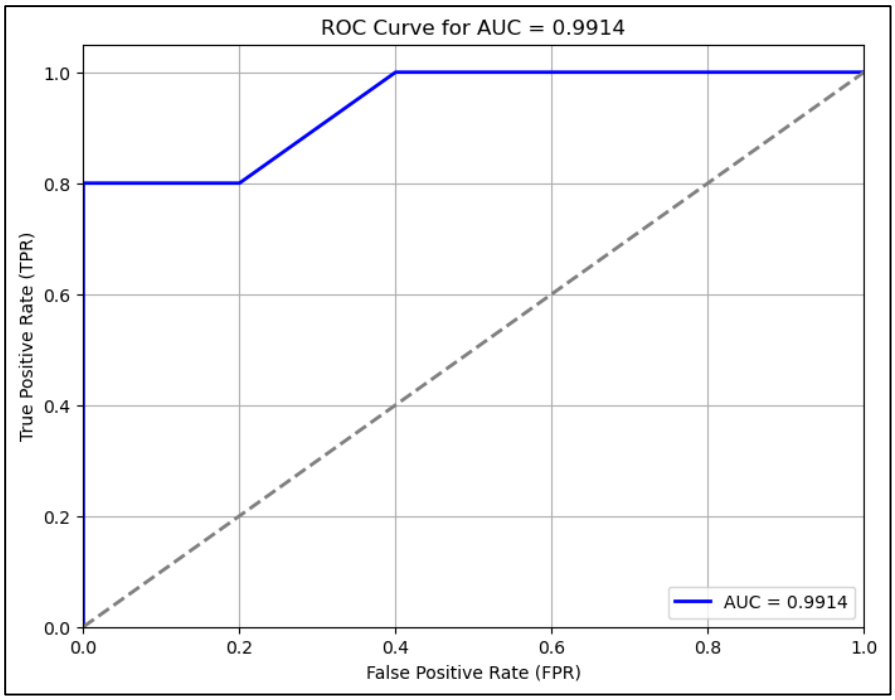


Figure 31: ROC Curve for 28 Features

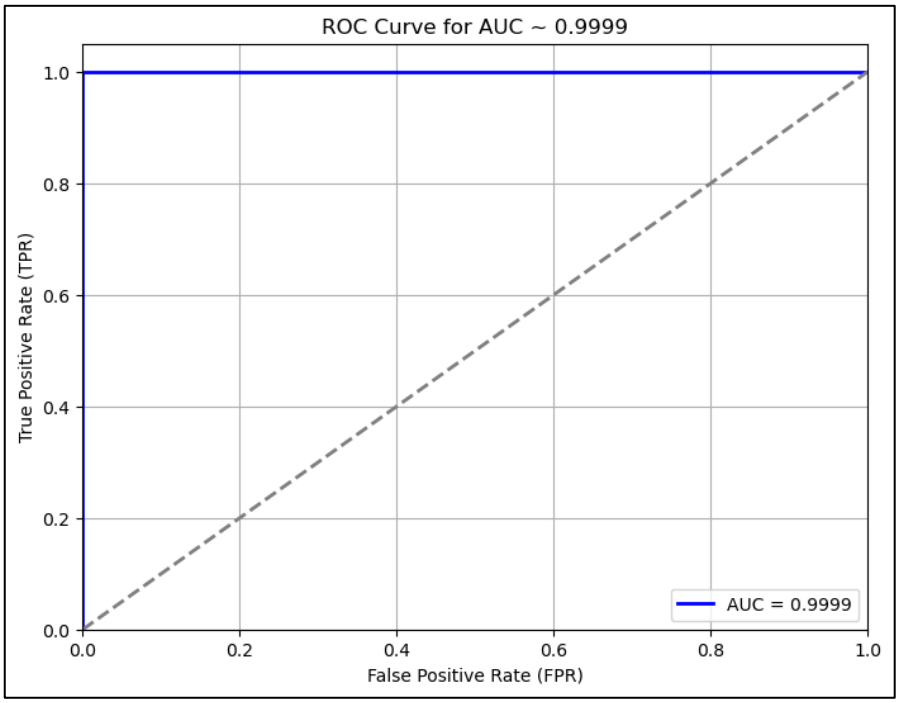


Figure 32: ROC Curve for 45 Features

As we see from the curves the best curve is for the 45 features which gives the highest AUC, for the ROC curve it is clear that if we compare the curve of the

4.3.2 Multiclass HAMD Classifier Results

The Multiclass HAMD-DNN model was trained using the train set. To tackle class imbalance and avoid favoring majority classes, a 10-fold stratified cross-validation approach was applied. The model's performance and reliability were then evaluated on unseen data using state-of-the-art metrics, which were used to assess and compare the achieved outcomes.

Table 11 presents the performance evaluation results for the multiclass classifier HAMD-DNN. It indicates that the F1-score achieves its peak when 45 features are utilized, surpassing all other attack classes. Additionally, the SMS Malware demonstrates the highest separability among the malware types. On the other hand, Scareware attacks exhibit the lowest detection rate when only 15 input features are employed.

Table 11: Performance Evaluation for Multiclass Classier HAMD-DNN

Number of Features	Performance Metric	Benign	Adware	Scareware	SMS Malware
15	Precision	95.012974	90.359183	95.869487	99.980
	Recall	98.07581	94.692782	41.416835	100
	F1-Score	96.519877	92.475452	57.844686	99.979
28	Precision	100.06	99.347573	97.465444	99.404
	Recall	100.059	99.819856	91.58892	99.93
	F1-Score	100.04999	99.582714	94.434627	99.673
45	Precision	99.607729	99.861881	97.97575	100
	Recall	99.87689	99.885896	94.753818	99.999
	F1-Score	99.741809	99.873888	96.337768	99.999

Figure 33 shows the precision for each class at varying numbers of input features. Precision signifies the classifier's tendency to overestimate one class over another, leading to false positives. Therefore, higher precision implies a lower false positive rate. The false positive rate decreased when the number of input features increased from 15 to 28 and then to 45. Additionally, the figure displays the recall for each class at different numbers of input features.

The bar chart indicates a high detection rate for all malware labels when using 15, 28, and 45 input features. However, the detection rate for SMS malware notably improved with the addition of extra input features. Furthermore, the recall slightly improved as the number of input features increased from 28 to 45. The best recall for all classes were observed when 45 features were used as input for the model. A drop-in recall indicates that the model either missed or underestimated a positive class, leading to false negatives.

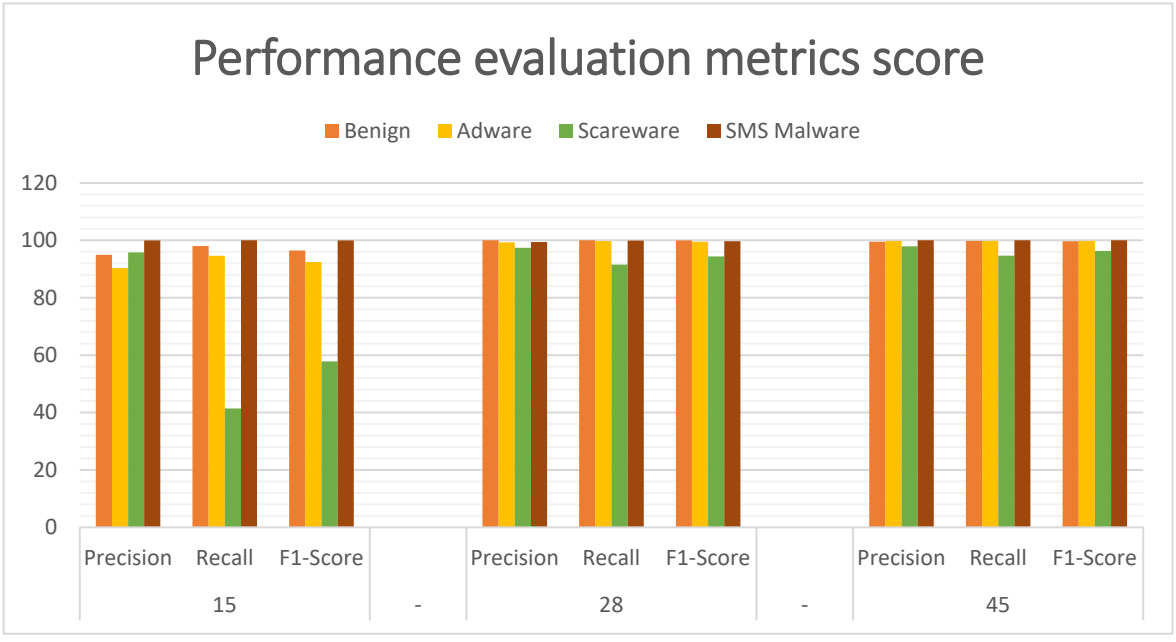


Figure 33: Performance Evaluation Metrics Score

Figures 34 shows the confusion matrix for the multiclass HAMD-DMM classifier when 45 features are utilized. The diagonal of the matrix represents the number of samples correctly predicted with their true labels. The intensity of the color corresponds to the number of samples.

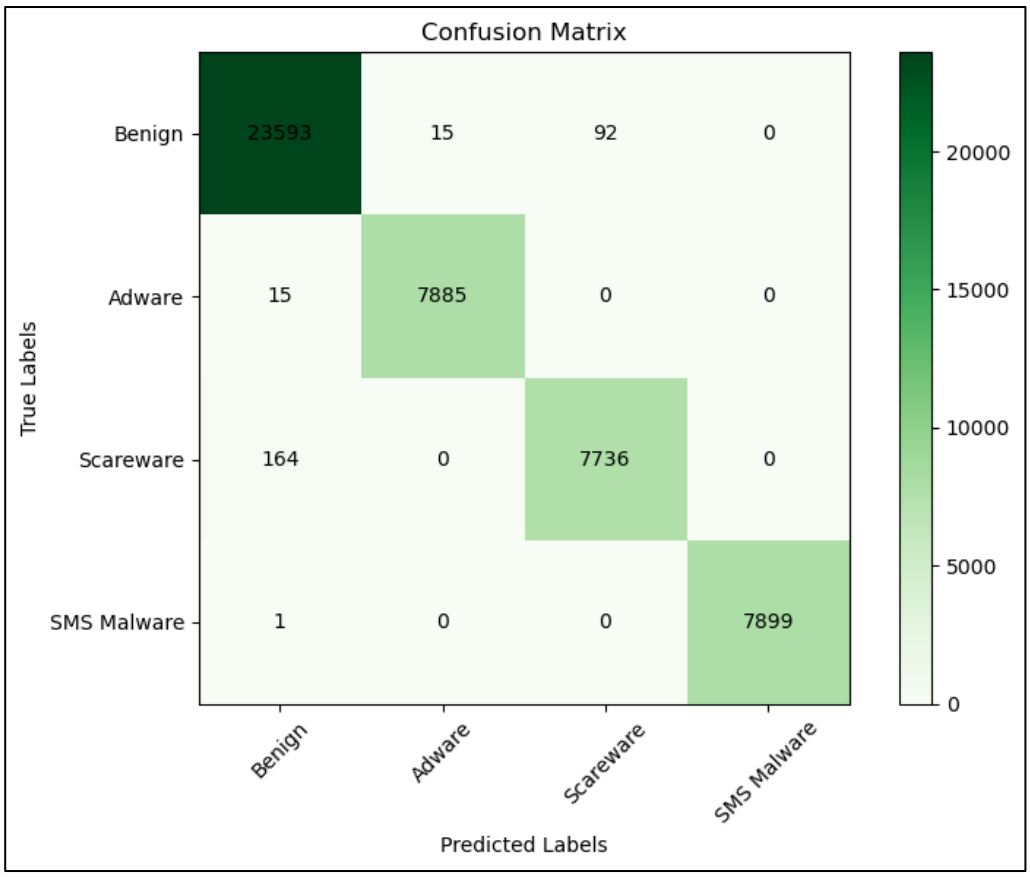


Figure 34: Confusion Matrix for Multiclass -HAMD-DNN (45 features)

Upon analyzing the confusion matrix, it is evident that some misclassifications occur. Specifically, there were 164 benign samples misclassified as scareware (false positives), and 92 scareware samples were misclassified as benign (false negatives). This indicates that scareware is not entirely separable and presents challenges for accurate classification.

Furthermore, 15 benign samples were incorrectly classified as adware, and 15 adware samples were wrongly labeled as benign. However, it is important to note that the other classes show a higher degree of separability and achieve relatively accurate predictions.

For the performance comparison with state-of-the-art models, we evaluate the effectiveness and efficiency of our proposed approach on the AMD dataset for anomaly-based Android malware detection systems. It is worth noting that there are no previous projects reported in the literature that utilize the same dataset for binary or multiclass classification. Most of the existing models have relied on external tools and servers for their evaluations. Therefore, our thesis presents a novel and unique contribution, as it directly utilizes the AMD dataset for the evaluation of both binary and multiclass classification tasks, setting it apart from the traditional approaches that rely on external resources.

4.3.3 Comparison with State-of-the-art Models

In table 12 it compares our model with other similar existing models [33] using machine learning.

Table 12: Comparison with State-Of-Arts

Model	Algorithm	Metric	Benign	Adware	Scareware	SMS Malware
MiMaLo	ML	Precision	88.5177	91.71834	94.31946	93.21864
		Recall	90.61812	74.91498	88.41768	92.5185
		F1-Score	92.11842	82.41648	92.5185	92.81856
HAMD-DNN	DNN	Precision	99.607729	99.861881	97.97575	100
		Recall	99.87689	99.885896	94.753818	99.999
		F1-Score	99.741809	99.873888	96.337768	99.999
CTML	ML	Precision	91.173231	94.46989	97.149044	96.015199
		Recall	93.336664	77.162429	91.07021	95.294055
		F1-Score	94.881973	84.888974	95.294055	95.603117

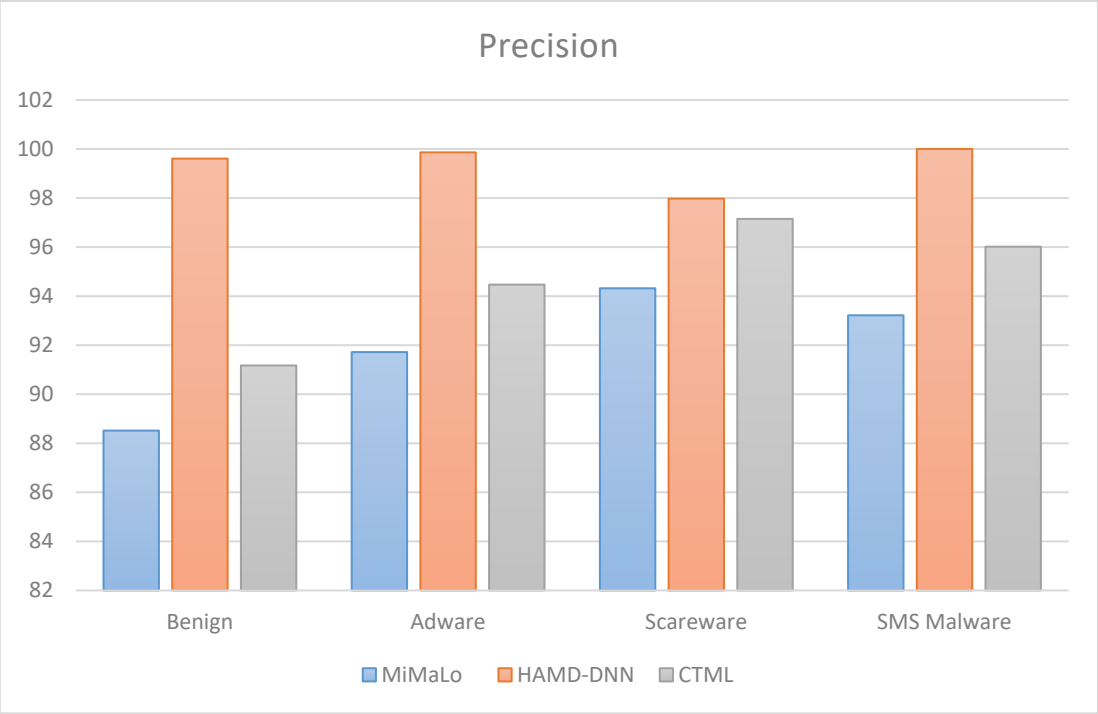


Figure 35: Comparison with State-Of-Art

As we can see in the precision our model has reached an impressive values of performance evaluation.

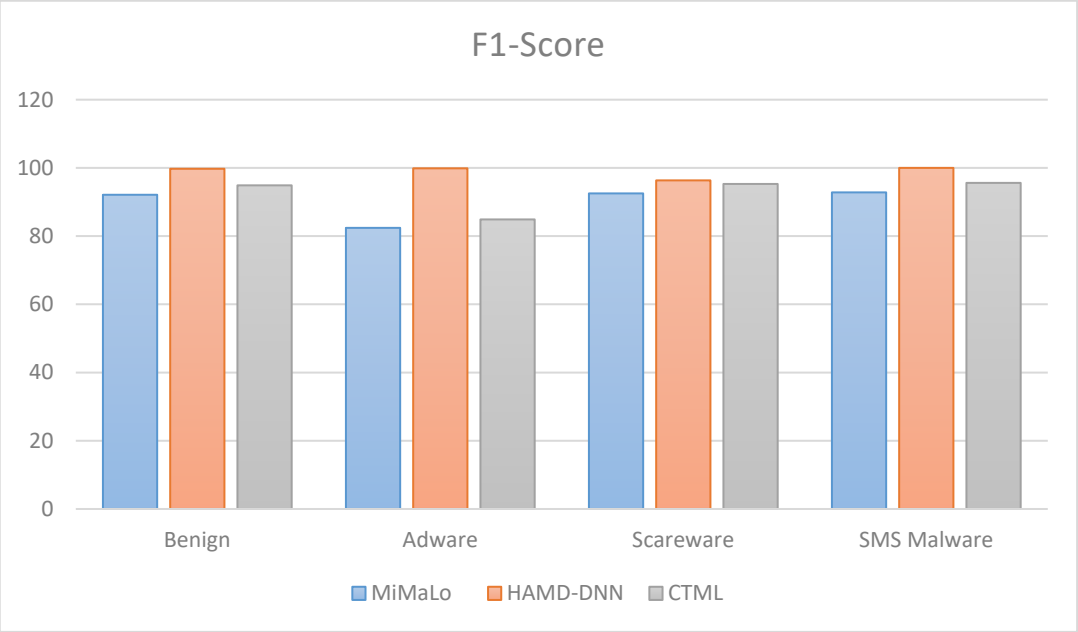


Figure 36: Comparison with State-Of-Art

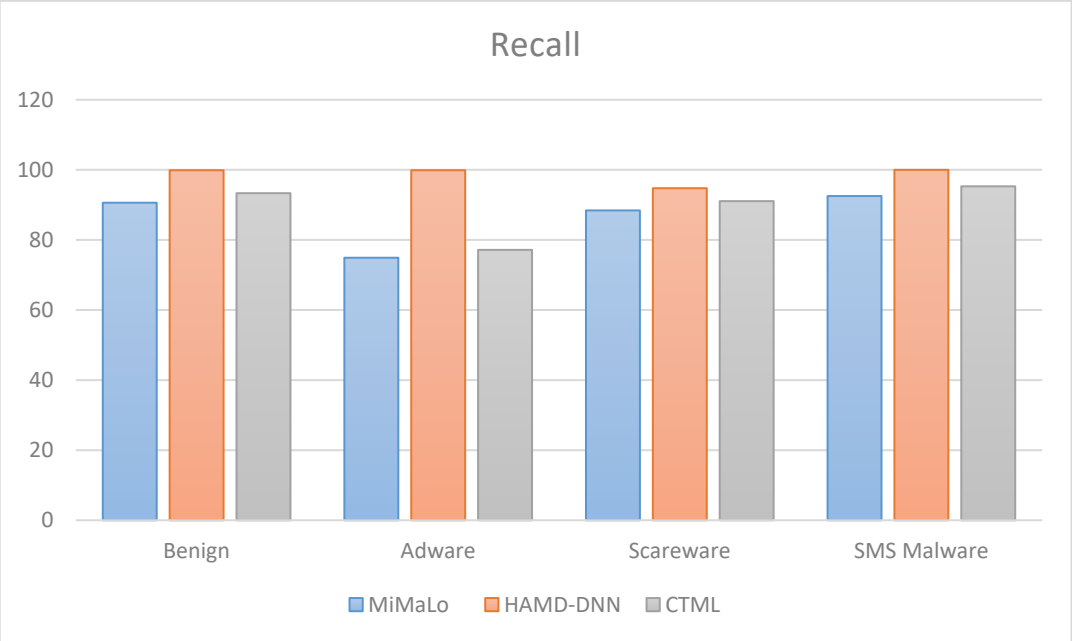


Figure 37: Comparison with State-Of-Art

Using special techniques, especially the hybrid combination of features and selection has reached us to a better performance matrix compared with other previous projects that uses similar techniques.

4.4 Summary

This project proposes innovative Binary and Multiclass HAMD-DNN models, designed to enhance classification accuracy and effectively detect various malware types.

Through rigorous experimentation and analysis, the project identifies the optimal network architecture for the Binary HAMD-DNN model. It is determined that employing three hidden layers with 90, 60, and 45 neurons, respectively, leads to the best performance in terms of network flow classification. Moreover, the model's effectiveness is further enhanced by selecting the 45 most important features, which significantly contribute to improved accuracy.

The proposed Binary HAMD-DNN model showcases remarkable results in classifying malware types. With a staggering classification accuracy of up to 99.8%. The Receiver Operating Characteristic Area Under the Curve (ROC-AUC) score reaches an outstanding value of 0.99, confirming the model's robustness and its ability to distinguish between different classes.

In addition to model classification, the thesis introduces the Multiclass HAMD-DNN model specifically designed for malware detection. This model performs exceptional accuracy and efficiency in identifying different types of malwares. It achieves impressive F1-scores of 99.682% for classifying benign samples, 99.814% for detecting Adware, 96.280% for detecting Scareware, and nearly perfect F1-score in classifying SMS Malware.

The performance of the proposed HAMD-DNN models is compared to other existing models in the field. The results demonstrate that the Binary HAMD-DNN model outperforms alternative approaches in terms of F1-score, precision, and recall metrics.

Furthermore, when benchmarked against related works, both Binary and Multiclass HAMD-DNN models showcase superior accuracy and effectiveness in classifying network flows and detecting malware.

So, this project presents the Binary and Multiclass HAMD-DNN models, offering a powerful solution for Android features classification and malware detection. Through the optimal network architecture and feature selection, the proposed models achieve exceptional accuracy and performance. The results indicate that the proposed models are highly effective in various real-world scenarios and hold great promise in strengthening network security and improving malware detection systems. The advancements made in this project contribute to the ever-evolving field of cybersecurity and open up new possibilities for enhanced network protection.

And as we saw in the previous section and according to the comparison with other model, we saw clearly that our model is more stable and efficient and achieve a better result than many other models with our special and new dataset used, as shown in figure 35 and table 12.

Chapter Five: Discussion and Conclusion

Chapter Five: Discussion and Conclusion

5.1 Discussion of The Results

Based on the experimental results, the following conclusions can be drawn:

Imbalanced dataset

The HAMD model is specifically designed for Android malware detection, utilizing a hybrid approach to capture and learn normal and anomalous patterns from data by leveraging examples. To ensure effective training and minimize errors, it is vital to provide an adequate number of learning examples for each pattern or class. However, when dealing with imbalanced datasets, where certain classes are underrepresented, the deep learning algorithm may develop a bias towards the majority classes, resulting in lower detection accuracy for the minority classes.

To address this challenge, the approach involves two strategies. In the first strategy, the model undertakes under-sampling of background traffic, reducing its influence, while simultaneously increasing the sample size for the underrepresented attack classes. This helps to balance the dataset and ensure equal representation of various classes.

In the second strategy, to improve generalization and minimize errors, the model is trained and validated using the stratified 10-fold cross-validation technique, which takes into account all samples from the minority classes. This technique partitions the data into ten subsets, ensuring that each subset maintains the same proportion of classes as the original dataset, thereby mitigating any potential bias and enhancing the model's overall performance.

By implementing these strategies, the HAMD model aims to optimize its detection capabilities and foster a more robust and accurate approach to identifying Android malware in real-world scenarios.

Feature Engineering

Feature engineering is a vital step in the classification model, entailing the mapping of input features to class labels or a target. This process revolves around extracting pertinent input features from raw data, employing a range of techniques to optimize model performance. The significance of these selected features holds a profound impact on the model's accuracy. As demonstrated in Figure 14, training the model on more relevant features results in remarkable improvements across all performance metrics. This emphasizes the critical role of feature engineering in enhancing the overall effectiveness and precision of the classification model.

Feature Selection

The choice of input features significantly influences the model's performance and capacity. Selecting relevant and uncorrelated features can lead to improved detection accuracy while simultaneously reducing the total number of model parameters, resulting in a less complex model. As depicted in Table 12, we observed a noticeable enhancement in performance when using 28 features compared to the initial 20 basic features. Furthermore, there was a slight yet discernible performance improvement when the model utilized 45 features instead of 28. These results underscore the importance of thoughtful feature selection in optimizing the model's efficiency and effectiveness.

Hyperparameters Tuning

Optimal hyperparameter selection plays a crucial role in enhancing both the model's training and generalization performance. As illustrated in Table 10, adjusting hyperparameters such as the number of hidden layers and hidden units directly impacts the model's performance and complexity. It is vital to tune these hyperparameters to align with the complexity of the specific problem, the number of input features, and the availability of sufficient training examples, especially for minority classes.

Automated techniques like grid search and random search can be employed to systematically search for the most suitable hyperparameters. Alternatively, manual tuning based on heuristics and relevant previous research can be applied. Whichever approach is chosen, finding the optimal hyperparameters is essential for maximizing the model's efficacy and ensuring it is well-suited to the unique requirements of the classification task at hand.

Performance metrics

The use of cutting-edge performance metrics is of utmost importance for diagnosing issues, enhancing overall performance, and enabling effective comparative analysis. Visualizing performance can be achieved through metrics.

By leveraging these performance metrics, practitioners can pinpoint areas for improvement and optimize the model's overall performance. The insights gained from these metrics are instrumental in refining the model's effectiveness and ensuring its suitability for the specific classification task at hand.

5.2 Conclusion

This research researched into the realm of anomaly-based Android malware detection systems (HAMD-DNN) and explored the potential of deep learning algorithms in constructing predictive models for this purpose. By leveraging Android features extracted from APKs, these models were designed to distinguish between benign and malicious applications in binary classification, while also identifying different types of malware in multiclass classification. The primary objective was to assess the efficacy of deep learning algorithms, considering the challenges posed by low detection accuracy, frequent false alarms, and the difficulty in detecting unknown malware observed in prior research.

The experimental results yielded promising outcomes, as both binary and multiclass HAMD-DNN models showed remarkable performances based on state-of-the-art evaluation metrics. The HAMD model was strong and dependable since it was based on deep learning, however, it demands careful consideration of the various key factors. Foremost, the supervised feedforward deep network necessitates extensive training on a sufficiently large and labeled dataset, ensuring its quality and accuracy to facilitate optimal model performance and generalization. Overcoming class imbalance in the dataset can be addressed effectively by employing sampling strategies and k-fold stratified cross-validation techniques, which provide an equitable representation of all classes.

Data preparation emerged as a crucial aspect significantly impacting the training and generalization errors of deep learning-based models. Through effective feature engineering, it becomes possible to extract or transform features from raw data, unveiling temporal or spatial associations vital for improved malware detection. Furthermore, given

deep learning algorithms' ability to process only numerical data, preprocessing input features plays a pivotal role in ensuring successful model execution. By selecting uncorrelated and essential features, the model's performance and efficiency can be enhanced, while also reducing the model's complexity.

Tuning the hyperparameters of the model emerged as another pivotal consideration to minimize both training and generalization errors. While data-driven model parameters are learned from the dataset, setting hyperparameters in advance significantly influences model performance. Employing techniques such as grid search and random search assists in identifying optimal hyperparameters, and in specific scenarios, heuristic approaches and existing literature may prove useful in manually fine-tuning the model.

In conclusion, deep learning algorithms emerge as formidable tools for constructing predictive models, capable of learning intricate patterns from data and effectively approximating complex functions that map inputs to outputs. When applied within the Android security context to model hybrid anomaly-based Android malware detection systems (HAMD), these algorithms exhibit exceptional classification performance, accurately detecting and recognizing various patterns of anomalies within known and unknown network traffic using benchmark datasets. The insights gained from this research provide valuable guidance in developing efficient and reliable HAMD systems based on deep learning, ushering in new possibilities for advancing Android malware detection and ensuring enhanced cybersecurity in the mobile ecosystem

5.3 Recommendations and Future Work

To expand upon this thesis and pave the way for further advancements in Android malware detection, several areas of exploration and improvement can be pursued in future research:

1. **Big Data Platform for Android Malware**: To facilitate the integration of deep learning HAMD models into real-world applications, the creation of a comprehensive big data platform dedicated to Android malware is essential. Such a platform would help overcome memory and computational constraints, enabling more extensive and sophisticated analyses of Android malware behavior.

2. **Semi-Supervised Learning**: The process of creating labeled malware detection datasets in the real world can be time-consuming and resource intensive. To address this challenge, incorporating semi-supervised learning approaches is recommended. By allowing models to learn from both labeled and unlabeled data, the efficiency and scalability of Android malware detection systems can be significantly improved.

3. **Sequence-Based Detection with RNN and LSTM**: To enhance the detection of unknown attacks and sophisticated malwares, adopting sequence-based models like Recurrent Neural Networks (RNN) or Long Short-Term Memory (LSTM) algorithms can be highly beneficial. By capturing the temporal dependencies and patterns in the sequence of actions, these models can better discern complex and evolving malware behaviors.

4. **Addressing Imbalanced Datasets**: Android malware detection datasets often suffer from class imbalance, with malwares being minority classes compared to normal behavior samples. To overcome this issue and enhance the generalization of deep

learning models, adversarial algorithms like Generative Adversarial Networks (GANs) and autoencoders can be employed. These algorithms can model the probability distribution of minority classes and synthesize additional samples, effectively increasing the representation of malwares in the dataset.

5. **Explainable Deep Learning Models**: As the adoption of deep learning in security-critical domains like malware detection grows, it becomes imperative to focus on interpretability and explainability. Future research should explore techniques to make deep learning models more transparent and understandable, providing insights into how the models arrive at their decisions.

6. **Transfer Learning and Domain Adaptation**: Leveraging transfer learning and domain adaptation techniques can be advantageous when dealing with limited labeled data from a different domain or source. By utilizing knowledge gained from related tasks or datasets, these methods can enhance the performance of Android malware detection models.

7. **Real-Time Detection and Deployment**: Investigating real-time detection methods and strategies for deploying deep learning-based HAMD systems on mobile devices would be crucial for ensuring rapid response to emerging threats and preserving user privacy.

In conclusion, the above recommendations offer valuable directions for future research, aiming to advance the field of Android malware detection using deep learning. By addressing challenges related to data, class imbalance, and model transparency, these endeavors can contribute to building more efficient, robust, and reliable HAMD systems, ultimately bolstering cybersecurity measures in the mobile ecosystem.

References

[1] Malware detection in mobile environments based on Autoencoders and API-images. (2019, November 12). https://doi.org/10.1016/j.jpdc.2019.11.001

[2] Lee, S., Jung, W., Lee, W., Oh, H. G., & Kim, E. T. (2022). Android malware dataset construction methodology to minimize bias–variance tradeoff. *ICT Express*, *8*(3), 444–462. https://doi.org/10.1016/j.icte.2021.10.001

[3] Android Malware Detection. (n.d.). Retrieved from https:///datasets/subhajournal/Android-malware-detection

[4] Aboaoja, F. A., Zainal, A., Ghaleb, F. A., Saleh Al-rimy, B. A., Elfadil Eisa, T. A., & Hassan Elnour, A. A. (2022, August 25). Malware Detection Issues, Challenges, and Future Directions: A Survey. https://doi.org/10.3390/app12178482

[5] Sihwail, R., Omar, K., & Zainol Ariffin, K. A. (2018). A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis. *International Journal on Advanced Science, Engineering and Information Technology*, *8*(4–2), 1662. https://doi.org/10.18517/ijaseit.8.4-2.6827

[6] Akpan, U. I., & Starkey, A. (2021). Review of classification algorithms with changing inter-class distances. *Machine Learning With Applications*, *4*, 100031. https://doi.org/10.1016/j.mlwa.2021.100031

[7] Akhtar, M. S., & Feng, T. (2023). Evaluation of Machine Learning Algorithms for Malware Detection. *Sensors*, *23*(2), 946. https://doi.org/10.3390/s23020946

[8] Akhtar, M. S., & Feng, T. (2022). Malware Analysis and Detection Using Machine Learning Algorithms. *Symmetry*, *14*(11), 2304. https://doi.org/10.3390/sym14112304

[9] A survey on various mobile malware attacks and security characteristics . (2017). *International Journal of Latest Trends in Engineering and Technology*. https://doi.org/10.21172/1.82.060

[10] US Smartphone Market Share: By Quarter (Updated on May 16, 2023). (2023, May 16). Retrieved from https://www.counterpointresearch.com/us-market-smartphone-share/

[11] Urmila, T. (2022). Machine learning-based malware detection on Android devices using behavioral features. *Materials Today: Proceedings*, *62*, 4659–4664. https://doi.org/10.1016/j.matpr.2022.03.121

[12] Android Malware Detection And Prevention. (2017). *International Journal of Recent Trends in Engineering and Research*, *3*(2), 213–217. https://doi.org/10.23883/ijrter.2017.3028.0uhbl

[13] Ke, Q., & Cheng, Y. (2015). Applications of meta-analysis to library and information science research: Content analysis. *Library & Information Science Research*, *37*(4), 370–382. https://doi.org/10.1016/j.lisr.2015.05.004

[14] ANDROID PROJECTS ON ANDROID ATTACK APPLICATION. (2020). *Journal of Xidian University*, *14*(5). https://doi.org/10.37896/jxu14.5/233

[15] Kim, M., Han, Y., Cho, M., Park, C., & Kim, S. W. (2015). DEX2C: Translation of Dalvik Bytecodes into C Code and its Interface in a Dalvik VM. *IEIE Transactions on Smart Processing and Computing*, *4*(3), 169–172. https://doi.org/10.5573/ieiespc.2015.4.3.169

[16] N., S. (2020). IoT based Child Protection Using Android App. *International Journal of Psychosocial Rehabilitation*, *24*(5), 5726–5733. https://doi.org/10.37200/ijpr/v24i5/pr2020280

[17] Su, X., Zhang, D., Li, W., & Wang, X. (2015). AndroGenerator: An automated and configurable Android app network traffic generation system. *Security and Communication Networks*, *8*(18), 4273–4288. https://doi.org/10.1002/sec.1341

[18] Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., & Liu, Y. (2022). Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Transactions on Software Engineering*, *48*(10), 4181–4213. https://doi.org/10.1109/tse.2021.3114381

[19] Panagiotou, P., Mengidis, N., Tsikrika, T., Vrochidis, S., & Kompatsiaris, I. (2021). Host-based Intrusion Detection Using Signature-based and AI-driven Anomaly Detection Methods. *Information & Security: An International Journal*, *50*, 37–48. https://doi.org/10.11610/isij.5016

[20] Tremblay, M. C., Dutta, K., & Vandermeer, D. (2010). Using Data Mining Techniques to Discover Bias Patterns in Missing Data. *Journal of Data and Information Quality*, *2*(1), 1–19. https://doi.org/10.1145/1805286.1805288

[21] Graph Approach for Android malware detection using machine learning techniques. (2021). *Humanitarian and Natural Sciences Journal*, *2*(11). https://doi.org/10.53796/hnsj21115

[22] Jurczyk, T. (2021). Clustering with Scikit-Learn in Python. *Programming Historian*, (10). https://doi.org/10.46430/phen0094

[23] Kashyap, A. K., Srivastava, H., Yadav, D., Verma, A. N. S., & Shahi, A. (2023). Object Detection Using Tensorflow Lite. *International Journal of Research Publication and Reviews*, *4*(5), 3093–3097. https://doi.org/10.55248/gengpi.4.523.40694

[24] Lipor, J., & Balzano, L. (2020). Clustering quality metrics for subspace clustering. *Pattern Recognition*, *104*, 107328. https://doi.org/10.1016/j.patcog.2020.107328

[25] Chen, X., Zheng, H., Wang, H., & Yan, T. (2022). Can machine learning algorithms perform better than multiple linear regression in predicting nitrogen excretion from lactating dairy cows. *Scientific Reports*, *12*(1). https://doi.org/10.1038/s41598-022-16490-y

[26] Tastemir, B., Malikova, F., & Aitbayeva, R. (2022). RF MACHINE LEARNING TECHNIQUE FOR EMAIL SPAM FILTERING. *SERIES PHYSICO-MATHEMATICAL*, *2*(342), 130–141. https://doi.org/10.32014/2022.2518-1726.134

[27] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

[28] Franco, N. R., Fresca, S., Manzoni, A., & Zunino, P. (2023). Approximation bounds for convolutional neural networks in operator learning. *Neural Networks*, *161*, 129–141. https://doi.org/10.1016/j.neunet.2023.01.029

[29] GOOGLE PLAY STORE BASED APPLICATIONS FOR SEARCH RANK FRAUD AND MALWARE DETECTION. (2020). *Journal of Critical Reviews*, *7*(04). https://doi.org/10.31838/jcr.07.04.163

[30] Mutelo, S., & Iyawa, G. (2023). Mobile Apps for Smart Cities: A Systematic Review of Google Play Store and Apple App Store. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.4332847

[31] Mobile malware goes straight for the money, says Kaspersky. (2015). *Network Security*, *2015*(5), 1–2. https://doi.org/10.1016/s1353-4858(15)30032-5

[32] Google Android security exploit made public by researcher. (2010). *Infosecurity*, *7*(6), 7. https://doi.org/10.1016/s1754-4548(10)70099-1

[33] G. (2022, December 7). Build software better, together. Retrieved from https://github.com

[34] Sadeghi, A., Bagheri, H., Garcia, J., & Malek, S. (2017). A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software. *IEEE Transactions on Software Engineering*, *43*(6), 492–530. https://doi.org/10.1109/tse.2016.2615307

[35] Talal, M., Zaidan, A. A., Zaidan, B. B., Albahri, O. S., Alsalem, M. A., Albahri, A. S., . . . Alaa, M. (2019). Comprehensive review and analysis of anti-malware apps for smartphones. *Telecommunication Systems*, *72*(2), 285–337. https://doi.org/10.1007/s11235-019-00575-7

[36] Aslan, O., & Samet, R. (2020). A Comprehensive Review on Malware Detection Approaches. *IEEE Access*, *8*, 6249–6271. https://doi.org/10.1109/access.2019.2963724

[37] Zou, S., Zhang, J., & Lin, X. (2014). An effective behavior-based Android malware detection system. *Security and Communication Networks*, *8*(12), 2079–2089. https://doi.org/10.1002/sec.1155

[38] Ünver, H. M., & Bakour, K. (2020, June 29). Android malware detection based on image-based features and machine learning techniques - SN Applied Sciences. https://doi.org/10.1007/s42452-020-3132-2

[39] T. (n.d.). Android-Malware-Datasets/README.md at master · traceflight/Android-Malware-Datasets. Retrieved from https://github.com/traceflight/Android-Malware-Datasets/blob/master/README.md

[40] VirusTotal. (n.d.). Retrieved from https://www.virustotal.com/gui/home/upload

[41] Singh, A. (2020). Malicious and Benign Webpages Dataset. *Data in Brief*, *32*, 106304. https://doi.org/10.1016/j.dib.2020.106304

[42] Martín, A., Lara-Cabrera, R., & Camacho, D. (2019). Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset. *Information Fusion*, *52*, 128–142. https://doi.org/10.1016/j.inffus.2018.12.006

[43] Sharma, T., & Rattan, D. (2021). Malicious application detection in Android — A systematic literature review. *Computer Science Review*, *40*, 100373. https://doi.org/10.1016/j.cosrev.2021.100373

[44] Akremi, A. (2021). Software Security Static Analysis False Alerts Handling Approaches. *International Journal of Advanced Computer Science and Applications*, *12*(11). https://doi.org/10.14569/ijacsa.2021.0121180

[45] Gamao, A. O. (2018). Malware Analysis on Android Apps: A Permission-Based Approach. *Social Science and Humanities Journal*, *2*(10). https://doi.org/10.18535/sshj/v2i10.109

[46] KABAKUŞ, A. T. (2021). Hybroid: A Novel Hybrid Android Malware Detection Framework. *Erzincan Üniversitesi Fen Bilimleri Enstitüsü Dergisi*, *14*(1), 331–356. https://doi.org/10.18185/erzifbed.806683

[47] Yuan, Z., Lu, Y., & Xue, Y. (2016). Droiddetector: Android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, *21*(1), 114–123. https://doi.org/10.1109/tst.2016.7399288

[48] Cai, H., Meng, N., Ryder, B., & Yao, D. (2019). DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling. *IEEE Transactions on Information Forensics and Security*, *14*(6), 1455–1470. https://doi.org/10.1109/tifs.2018.2879302

[49] Sriyanto, B. Sahrin, S., Mohd. Faizal, A., Suryana, N., & Suhendra, A. (2022). MiMaLo: Advanced Normalization Method for Mobile Malware Detection. *International Journal of Modern Education and Computer Science*, *14*(5), 24–33. https://doi.org/10.5815/ijmecs.2022.05.03

[50] Agrawal, P., & Trivedi, B. (2020). Unstructured Data Collection from APK files for Malware Detection. *International Journal of Computer Applications*, *176*(28), 42–45. https://doi.org/10.5120/ijca2020920308

[51] Njeri, N. R. (2022). Data Preparation For Machine Learning Modelling. *International Journal of Computer Applications Technology and Research*, *11*(06), 231–235. https://doi.org/10.7753/ijcatr1106.1008

[52] Soe Myint Myat. (2019). Feature Extraction using Hybrid Analysis for Android Malware Detection Framework. *International Journal of Engineering Research And*, *V8*(06). https://doi.org/10.17577/ijertv8is060691

[53] Salah, A., Shalabi, E., & Khedr, W. (2020). A Lightweight Android Malware Classifier Using Novel Feature Selection Methods. *Symmetry*, *12*(5), 858. https://doi.org/10.3390/sym12050858

[54] A, L., & M, S. (2022). Android Malware Detection using Multilayer Autoencoder and R F. *International Journal of Engineering Trends and Technology*, *70*(11), 249–257. https://doi.org/10.14445/22315381/ijett-v70i11p227

[55] Android Malware Detection And Prevention. (2017). *International Journal of Recent Trends in Engineering and Research*, *3*(2), 213–217. https://doi.org/10.23883/ijrter.2017.3028.0uhbl

[56] Android Malware Visualize Importance. (n.d.). Retrieved from https://www.kaggle.com/code/stpeteishii/Android-malware-visualize-importance

[57] Yang, Z. P., & Zhao, Y. (2024). Hybrid SGD algorithms to solve stochastic composite optimization problems with application in sparse portfolio selection problems. *Journal of Computational and Applied Mathematics*, *436*, 115425. https://doi.org/10.1016/j.cam.2023.115425

Appendix A

A.1 Android Architecture

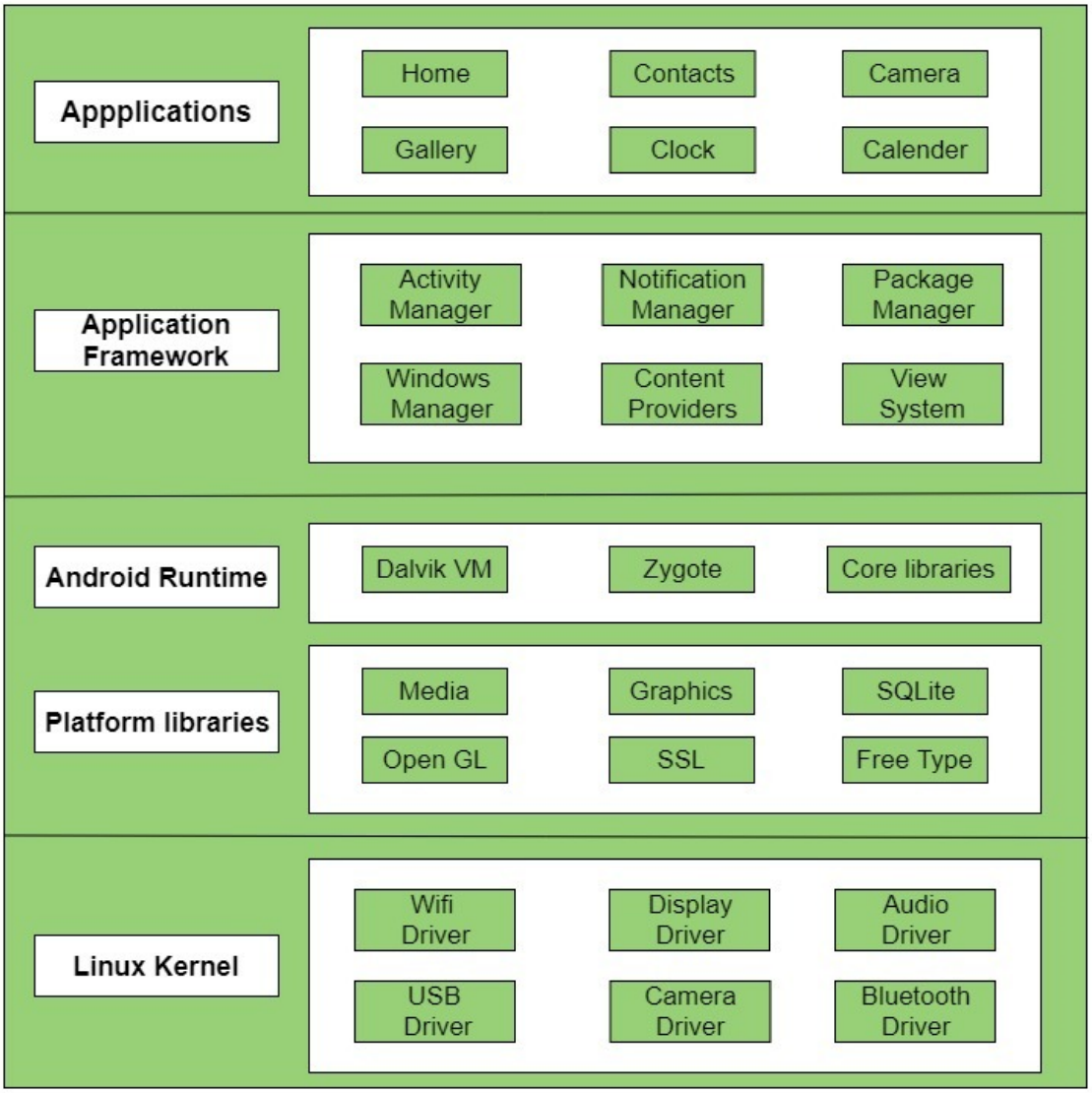


Figure 38: Android Architecture

A.2 Sample Python Code

The python source code for building, training, and evaluating both binary and multiclass HAMD models using UGR'16 datasets is organized as follows:

```
# Build HAMD-DNN Model
#After Importing libraries
# Model Creation Function, with Keras Classifier

def create_model():

  model = Sequential()
  model.add(Dense(90, input_dim=45, kernel_initializer='uniform',
activation='relu'))
  model.add(Dense(60, kernel_initializer='uniform', activation='relu'))
  model.add(Dropout(0.2))
  model.add(Dense(45, kernel_initializer='uniform', activation='relu'))
  model.add(Dropout(0.2))
  model.add(Dense(1, kernel_initializer='uniform', activation='sigmoid'))

  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
      loss=tf.keras.losses.BinaryCrossentropy(),
      metrics=[tf.keras.metrics.BinaryAccuracy(),
          tf.keras.metrics.FalseNegatives(),
          tf.keras.metrics.FalsePositives(),
          tf.keras.metrics.Precision(),
          tf.keras.metrics.Recall(),
          tf.keras.metrics.AUC()])
  return model
```

Appendix B

AMD Preprocessed Features

The details of selected input features used to train the HAMD-DNN

Table 13: Details of the 45 Features in the AMD Dataset

No.	Feature name	Type
1	Flow id	continuous
2	s_sessiontime	continuous
3	r_ sessiontime	continuous
4	sr_sessiontime	continuous
5	Timestamp	boolean
6	Flow Duration	boolean
7	Total Fwd Packets	boolean
8	Total Backward Packets	boolean
9	Total Length of Fwd Packets	boolean
10	Total Length of Bwd Packets	boolean
11	Fwd Packet Length Max	boolean
12	Fwd Packet Length Min	boolean
13	Flow Bytes/s	boolean
14	Flow Packets/s	boolean
15	Flow IAT Std	boolean
16	Flow IAT Max	boolean
17	Flow IAT Min	boolean
18	Bwd IAT Total	boolean
19	Bwd IAT Mean	boolean
20	Bwd IAT Std	boolean
21	Fwd PSH Flags	boolean
22	Bwd PSH Flags	boolean
23	Fwd URG Flags	boolean
24	Bwd URG Flags	boolean
25	Fwd Header Length	boolean
26	Bwd Header Length	boolean
27	FIN Flag Count	boolean
28	SYN Flag Count	boolean
29	RST Flag Count	boolean
30	PSH Flag Count	boolean
31	ACK Flag Count	boolean
32	URG Flag Count	boolean
33	CWE Flag Count	boolean
34	ECE Flag Count	boolean

35	Down/Up Ratio	boolean
36	Subflow Fwd Packets	boolean
37	Subflow Bwd Packets	boolean
38	Init_Win_bytes_forward	boolean
39	Init_Win_bytes_backward	boolean
40	act_data_pkt_fwd	boolean
41	min_seg_size_forward	boolean
42	Active Mean	boolean
43	Active Std	boolean
44	Idle Mean	boolean
45	Idle Std	boolean

Table 14: Target Classes of Multiclass HAMD

No.	Class Label	Type
1	Benign	boolean
2	Adware	boolean
3	Scareware	boolean
4	SMS Malware	boolean

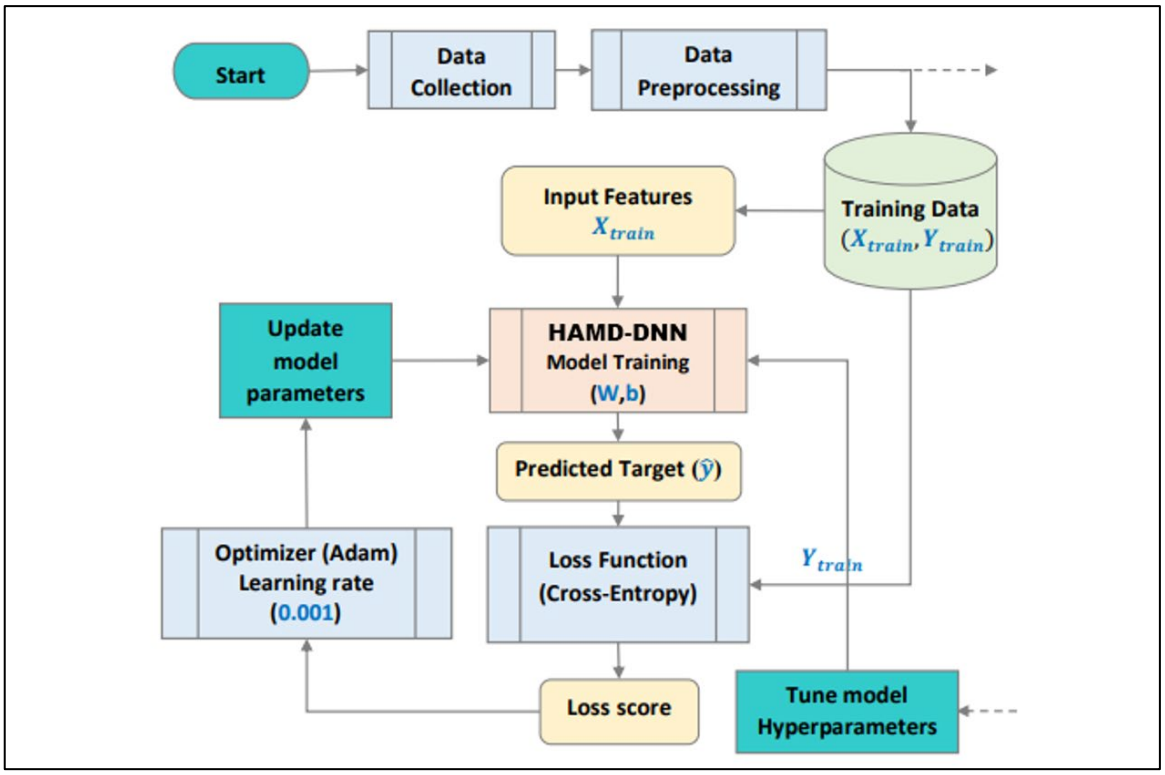


Figure 39: Testing of the HAMD_DNN Model

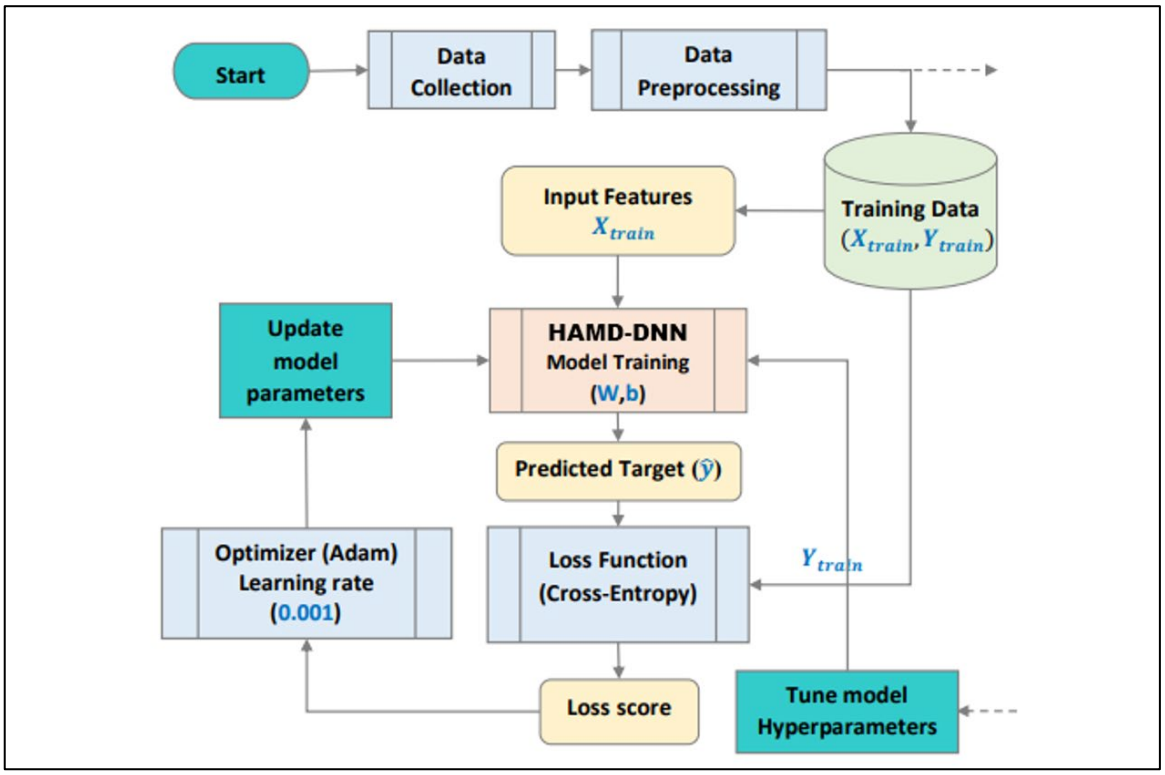


Figure 40: Training of HAMD-DNN model