A HYBRID DL-BASED FRAMEWORK TO CLASSIFY MALWARE USING MEXICAN HAT WAVELET FUNCTION

By

AFRASIAB YOUNIS

NATIONAL UNIVERSITY OF MODERN LANGUAGES

ISLAMABAD

JANUARY, 2024

A Hybrid Dl-Based Framework to Classify Malware Using Mexican Hat Wavelet Function

By

Afrasiab Younis

BSCS, University of Engineering and Technology, Taxila, 2019

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In **Computer Science**

To

FACULTY OF ENGINEERING & COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

NATIONAL UNIUVERSITY OF MODERN LANGUAGES FACULTY OF ENGINEERIING & COMPUTER SCIENCE

THESIS AND DEFENSE APPROVAL FORM

The undersigned certify that they have read the following thesis, examined the defense, are satisfied with overall exam performance, and recommend the thesis to the Faculty of Engineering and Computer Sciences for acceptance.

Thesis Title: A Hybrid DL-Based Framework to Classify Malware using Mexican Hat Wavelet Function.

Submitted By: Afrasiab Younis **Registration #:** 58 MS/CS/S21

Master of Science in Computer Science
Degree Name in Full

Computer Science
Name of Discipline

Dr. Moeenuddin Tariq Research Supervisor	Signature of Research Supervisor
Mr. Farhad Research Co-Supervisor	Signature of Research Co-Supervisor
Dr. Sajjad Haider Head of Department (CS)	Signature of HoD (CS)
Dr, M. Noman Malik Name of Dean (FE&CS)	Signature of Dean (FE&CS)

January 8th, 2024

AUTHOR'S DECLARATION

I Afrasiab Younis

Son of Muhammad Aslam

Registration # MSCS-S21-118

Discipline Computer Science

Candidate of **Master of Science in Computer Science (MSCS)** at the National University of Modern Languages do hereby declare that the thesis **A Hybrid DL-Based Framework to Classify Malware using Mexican Hat Wavelet Function** submitted by me in partial fulfillment of MSCS degree, is my original work, and has not been submitted or published earlier. I also solemnly declare that it shall not, in future, be submitted by me for obtaining any other degree from this or any other university or institution. I also understand that if evidence of plagiarism is found in my thesis/dissertation at any stage, even after the award of a degree, the work may be cancelled and the degree revoked.

Signature of Candidate

Afrasiab Younis

Name of Candidate

January 8th,2024

Date

ABSTRACT

Title: A Hybrid DL-Based Framework to Classify Malware using Mexican Hat Wavelet Function.

Detecting and categorizing malware represents a substantial and demanding undertaking within the realm of information security and various other computer-related domains. Millions of malicious files are detected annually. The high volume is largely due to malware authors using mutations to evade detection, Malware variants are constantly evolving through the use of advanced obfuscation and packing methods, making detection and classification increasingly difficult. In order to efficiently examine and categorize a substantial volume of files, it becomes imperative to group them and ascertain their behavioral characteristics to classify them effectively. In recent, most malware classification techniques have been based on machine learning or deep learning models. These models work with the train and test. The models are trained with the features, for instance, opcode sequence, API calls, signature, etc. Recently, many deep learning techniques have been proposed for Alex Net Network, Resnet-50 Network, and Hybrid (AlexNet-Resnet-50). These models work well in terms of accuracy, Sensitivity, and so forth. However, these models are complex in nature and need high computational power. In order to adequately confront the difficulty presented by emerging malware variations, it becomes essential to employ alternative approaches, as conventional artificial intelligence and machine learning algorithms are no longer capable of identifying all intricate and constantly changing variants. A promising solution is deep learning, which differs from traditional machine learning. This study proposes a Mexican hat wavelet function that classifies malware variants through a hybrid deep learning model in this approach, malware samples undergo conversion into grayscale images before being fed into the DL system. Following the image acquisition section, the proposed method employs the convolution layers of the hybrid architecture to extract high-level malware features from the malware images with cloud-based architecture to decrease the computational intricacy, and neural network complexity to achieve higher accuracy. Upon subjecting the proposed method to testing using the MALIMG dataset, an accuracy of 99% was achieved. Similarly, when applied to the MALEVIS dataset, an accuracy of 97.12% was attained, outperforming the majority of machine learning-based methods employed for malware detection.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	-	Application Programming Interface
APT	-	Advanced Persistent Threats
BOFM	-	Behavior Oriented Feature Model
CNN	-	Convolutional Neural Network
CTPL	-	Computation Tree Predicate Logic
DL	-	Deep learning
FP	-	Frequent Pattern growth algorithm
Malware	-	Malicious software
MtNet	-	A Multi-Task Neural Network
MALIMG	-	Malicious Image
MALEVIS	-	Malware Evaluation with Vision
NLP	-	Natural language processing
RGB	-	Red, Green and Blue
ReLU	-	Rectified Linear Unit.
SCBM	-	Semantic-Centric Behavioral Model
SVM	-	Support Vector Machine
VGG-16	-	Visual Geometry Group 16
2D	-	Two-Dimensional

LIST OF SYMBOLS

Σ	-	Sigmoid
σ	-	Activation function
^	-	Logical conjunction

ACKNOWLEDGMENT

The authors would like to express their deep gratitude to all those who have contributed to this research. We would like to acknowledge the support and guidance provided by our supervisor Assistant Professor Dr Moeen Din Tariq and Co supervisor Sir Farhad, who has been a constant source of inspiration throughout the research process.

Finally, we would like to extend our sincere thanks to our families and friends, who provided us with unwavering support and encouragement. Their love and understanding have been a constant source of motivation during the journey of this research. Our heartfelt thanks go to our colleagues, for their invaluable contributions to the project and for creating a supportive research environment.

This research would not have been possible without the support and contributions of all those acknowledged here. We are deeply grateful for their help and generosity

DEDICATION

To my parents, who have always believed in me and encouraged me to follow my dreams. Your love and support have been my foundation, and I am forever grateful. To my advisor, who guided me with wisdom and patience throughout this journey. Your expertise and dedication to your students are an inspiration to me.

CHAPTER 1

INTRODUCTION

1.1 Overview

Information technology encompasses advanced rapidly and the growth of network technology has made our daily routines more comfortable. However, it also raises various security risks despite its exceptional services. Malware, an abbreviation for malicious software, encompasses any software intentionally crafted with the purpose of causing harm or disruption to computer systems. It encompasses a wide range of harmful software, this encompasses viruses, worms, trojans, ransomware, spyware, and other types of malicious software. Malware can be employed to illicitly acquire sensitive information, hold computer systems or data hostage, compromise the security of computer systems, and cause other types of harm. [1]–[3]. Malware has posed a threat to individuals and businesses since its emergence in the early 1970s. Since that time, the world has witnessed the release of hundreds of thousands of unique malware variants, all with the objective of causing as much disruption and harm as possible. There is no denying that the overall count of malware incidents has experienced a significant increase over time. The occurrence of malware attacks has been on the rise, with 69,277,289 unique malicious objects detected by antivirus company Kaspersky Lab in 2016 [47] and 670 million malware samples found by McAfee Labs in 2017 [48]. Additionally, Malwarebytes reported the detection of over 50 million cyber threats in both 2018 and 2019 [49,50]. As reported in the 2020 Trend Micro Cybersecurity Report, there were 119,000 cyberattacks occurring every minute [51]. Due to the wide variety of malware and the increasing frequency of these attacks, researchers have put forth various approaches to categorize malware.

1.2 Types of Malware

Malware Types: Here's a rundown of some of the most popular malware types and classifications.

1.2.1 Virus

A computer virus is a form of malicious software that reproduces itself by contaminating other computer programs and files. A computer virus can cause harm to the infected computer by altering or destroying data, stealing sensitive information, or using the infected computer to spread the virus to other computers. Some viruses are designed to cause damage to the computer system, while others are designed to steal personal information or launch attacks on other systems. Computer viruses can be spread through email attachments, downloads, or infected websites.

Key characteristics of computer viruses include:

i. Self-Replication: A computer virus has the ability to make copies of itself and attach those copies to other files or programs. This allows it to spread to other computers or files when infected files are shared or transferred.

ii. Malicious Intent: Computer viruses are typically created with malicious intent. They can cause various types of harm, including data loss, system crashes, unauthorized access, and theft of sensitive information.

iii. Concealment: Viruses often attempt to hide their presence to avoid detection by security software or users. They may employ techniques to evade antivirus scans and security measures.

iv. Trigger Conditions: Many viruses are designed to execute specific actions based on certain trigger conditions. For example, a virus might be programmed to activate on a certain date, when a specific file is opened, or when certain user actions are performed.

v. Spread Mechanisms: Computer viruses have the capacity to propagate through diverse avenues, including infected email attachments, files downloaded from the internet, compromised software, shared documents, and removable storage devices.

vi. Variants and Mutations: Like biological viruses, computer viruses can evolve and have different variants. New variants may have different behaviors or characteristics, making them harder to detect and combat.

vii. Payload: Viruses often carry a payload, which is the harmful action they perform once activated. The payload can vary, encompassing actions such as file deletion, unauthorized data retrieval, or the initiation of distributed denial-of-service (DDoS) attacks.

viii. Prevention and Protection: To safeguard against computer viruses, individuals can employ antivirus software, maintain current operating systems and software, exercise caution when downloading files or clicking on links, and adopt safe online practices.

1.2.2 Worm

A computer worm is a form of malicious software that autonomously propagates from one computer to another, frequently via a network, without necessitating any user intervention. Unlike viruses, which attach themselves to existing files, worms are standalone software that replicates and propagates on their own. They can cause harm to infected computers by consuming network bandwidth, slowing down or crashing the system, and sometimes opening security holes that can be exploited by other malicious software. Some computer worms are designed to steal sensitive information or use infected computers to launch attacks on other systems. Key characteristics of computer worms include:

i. Self-Propagation: Worms are capable of spreading themselves across networks and systems without requiring user intervention. They can do this by exploiting vulnerabilities in software or by using various communication methods.

ii. Autonomous Replication: Once a computer is infected with a worm, the worm can independently search for other vulnerable computers and attempt to infect them. This autonomous behavior can lead to rapid and widespread infections.

iii. Network Exploitation: Worms frequently take advantage of weaknesses in network services, operating systems, or software to illicitly gain entry to computers. These vulnerabilities serve as entry points for them to install themselves on the target systems.

iv. Payload: Worms might contain a harmful payload, which could encompass activities like file deletion, unauthorized data theft, establishment of remote access backdoors, or the initiation of distributed denial-of-service (DDoS) attacks.

v. Resource Consumption: Due to their self-replicating nature, worms can consume significant network and system resources as they spread. This can lead to degraded performance and network congestion.

vi. Email and Social Engineering: Some worms spread through email attachments, social media links, or instant messaging platforms. They may deceive users into opening infected files by disguising themselves as legitimate messages or files.

vii. Vulnerability Exploitation: Worms often target known vulnerabilities in software and systems. Patches and updates that address these vulnerabilities are crucial for preventing worm infections.

viii. Prevention and Mitigation: Protecting against worms involves maintaining up-to-date software and security patches, using firewalls and intrusion detection systems, implementing strong access controls, and using security software to detect and block malicious activity.

Worms can cause significant disruptions to computer networks, leading to financial losses, data breaches, and operational downtime. Therefore, practicing good cybersecurity hygiene and staying informed about the latest threats and vulnerabilities are essential for preventing worm infections and minimizing their impact.

1.2.3 Spyware

Computer spyware is a form of malicious software crafted to clandestinely monitor and collect information from an infected computer, all without the user's awareness or consent. Spyware has the capability to trace a user's online activity, purloin personal data like login details and credit card numbers, exhibit unwanted advertisements, and decelerate the computer's performance. Some spyware can even control the infected computer and use it for malicious purposes such as sending spam or participating in a network of infected computers used to launch attacks on other systems (known as a "botnet").

Key characteristics of computer spyware include:

i. Covert Installation: Spyware is often installed without the user's awareness or explicit consent. It may be bundled with seemingly legitimate software or hidden within downloads.

ii. Information Collection: Spyware monitors a user's actions on their computer or device, such as websites visited, search queries, keystrokes, login credentials, and more. This information is then transmitted to third parties for various purposes, including advertising, data profiling, and identity theft.

iii. Data Transmission: The data collected by spyware is sent to remote servers controlled by the individuals or groups behind the malware. This data may be used for malicious purposes or sold to advertisers, marketers, or other parties.

iv. Invasive Tracking: Spyware can track a user's online behavior in real-time, capturing personal and sensitive information as the user interacts with websites, applications, and online services.

v. Browser Hijacking: Some spyware variants alter a user's browser settings, redirecting them to unwanted websites or injecting ads and pop-ups into web pages.

vi. Slowdowns and System Issues: Spyware can consume system resources and slow down a computer's performance. It may also cause crashes and instability.

vii. Personal Privacy Violation: Spyware constitutes a serious invasion of privacy, as it captures and transmits personal and confidential data without the user's consent.

viii. Prevention and Removal: To prevent spyware infections, users should be cautious when downloading software, clicking on links, and opening attachments. Regularly updating software and using reputable antivirus and anti-spyware tools can help detect and remove spyware infections.

ix. Legal and Ethical Concerns: Spyware is often distributed illegally, as it invades user privacy and violates computer usage policies. Legal action can be taken against those responsible for creating and distributing spyware.

It's important to note that some software applications, such as parental control software and employee monitoring tools used within legal boundaries, share similarities with spyware in terms of monitoring user activities. However, these tools are typically installed with proper consent and for legitimate purposes, unlike malicious spyware that operates stealthily and with malicious intent.

1.2.4 Trojan

A computer Trojan is a type of malicious software that disguises itself as a legitimate program and is often distributed through email attachments, downloads, or compromised websites. In contrast to viruses and worms, Trojans do not replicate automatically; instead, they rely on deceiving users into voluntarily downloading and installing them. Once installed, Trojans can carry out a wide range of harmful activities, including stealing personal information, downloading additional malware, or providing remote control to an attacker over the compromised computer. Trojans can also establish a concealed entry point within the affected system, enabling an attacker to bypass standard security measures and gain unauthorized access to sensitive data. Furthermore, Trojans can be used to create a botnet, which is a network of compromised computers that can be employed for large-scale malicious operations like distributed denial of service (DDoS) attacks.

Key characteristics of Trojans include:

i. Disguised Nature: Trojans often masquerade as legitimate files, software, or content to trick users into executing or opening them. They may appear as harmless attachments, games, utility programs, or software updates.

ii. Unauthorized Actions: Once executed, a Trojan performs actions that the user did not intend or expect. These actions can include stealing sensitive information, creating backdoors for remote access, deleting files, spreading other malware, or conducting fraudulent activities.

iii. Remote Access: Trojans are crafted to grant unapproved remote access to the compromised system, enabling attackers to manipulate the infected computer, pilfer data, or initiate further attacks.

iv. Payloads: Trojans can carry a variety of payloads, which are the harmful actions they perform once executed. The specific payload depends on the intentions of the malware creator. Common payloads include data theft, data manipulation, and unauthorized system changes.

v. Social Engineering: Trojans often use social engineering techniques to manipulate users into executing them. This can include enticing subject lines in emails, fake download links, and deceptive advertising.

vi. Distribution: Trojans are often distributed through methods like email attachments, malicious links, infected websites, pirated software, and compromised downloads.

vii. Multiple Variants: Trojans come in various forms and serve different purposes, such as banking Trojans that target financial data, ransomware Trojans that encrypt files for ransom, and remote access Trojans (RATs) that provide unauthorized control over a system.

viii. Prevention and Protection: Users can protect themselves from Trojans by being cautious when downloading files, clicking on links, and opening email attachments. Regularly updating software, using strong passwords, and employing security software can help detect and prevent Trojan infections.

It's worth mentioning that the expression "Trojan horse" originates from Greek mythology, in which a wooden horse was employed to trick the city of Troy and gain entry to its fortifications. In the realm of computer security, a Trojan function in a similar manner by concealing its malicious purpose to infiltrate a computer system.

1.2.5 Ransomware

Computer ransomware is a category of malicious software that encrypts the files belonging to its target and demands a ransom in return for the decryption key. The dissemination of ransomware attacks frequently occurs through email attachments, compromised websites, or malicious advertisements. After computer gets compromised, the ransomware typically exhibits an on-screen message providing directions for remitting the ransom. Ransomware can cause significant harm to infected users, as the encrypted files can be permanently lost if the victim does not pay the ransom or does not have a backup of the files. In some cases, ransomware can also spread to other computers on the same network, leading to a widespread infection.

The Mexican Hat wavelet is a wavelet function that finds application in image processing and signal analysis. It is named after its shape, which resembles a Mexican sombrero. It is a type of second-generation wavelet and is used in various applications, including edge detection, noise reduction, and image compression. The Mexican Hat wavelet has good localization properties, meaning that it can accurately represent signals with sharp changes or edges. The Mexican hat wavelet is also known as the Ricker wavelet [55].

In deep learning, the Ricker wavelet is a commonly used wavelet function that can be used as a building block for constructing more complex wavelet functions. The Ricker wavelet is also known as the "Mexican hat wavelet" or the "Mexican hat function" due to its characteristic shape, which resembles a Mexican hat. This wavelet is used in various deep learning applications, including image processing, signal processing, and pattern recognition. The Ricker wavelet is particularly useful for detecting edges and detecting signals with specific frequencies in a signal or image. It can also be used as a feature extractor for deep neural networks, where the wavelet coefficients can be used as input to the network to improve its performance [55].

Key characteristics of ransomware include:

i. File Encryption: Ransomware employs robust encryption algorithms to encrypt the files of the victim, making the files unreadable and inaccessible without the decryption key.

ii. Ransom Demand: After encrypting the victim's files, the attacker presents a ransom demand, usually in cryptocurrency, in exchange for providing the decryption key. The ransom note typically includes instructions on how to make the payment and obtain the decryption key.

iii. Time Pressure: Ransomware attackers often impose a time limit on the ransom payment, threatening to permanently delete the decryption key if the ransom is not paid within the specified timeframe.

iv. Wide Distribution: Ransomware can be distributed through various methods, including malicious email attachments, compromised websites, malicious links, and exploit kits.

v. Variants and Families: There are different variants and families of ransomware, each with its own characteristics and methods. Some ransomware strains are more sophisticated than others.

vi. Impact: Ransomware attacks can have severe consequences, leading to data loss, operational downtime, financial losses, reputational damage, and legal repercussions.

vii. Publicized Cases: High-profile ransomware attacks have targeted individuals, businesses, hospitals, government agencies, and other institutions. Notable examples include the WannaCry and NotPetya attacks.

viii. Backup Importance: Regularly backing up important data is a key defense against ransomware. If an attack occurs, victims can restore their systems and data from a backup without paying the ransom.

ix. Reporting and Law Enforcement: Victims are encouraged to report ransomware attacks to law enforcement agencies and cybersecurity organizations. Paying the ransom does not ensure that the attacker will furnish the decryption key, and it may also encourage further attacks.

x. Prevention and Mitigation: Preventing ransomware involves maintaining up-to-date software, using strong and unique passwords, educating users about phishing and safe online practices, and using reputable antivirus and anti-malware solutions.

1.3 Malware Analysis

Malware analysis entails the procedure of scrutinizing malicious software (malware) to gain insights into its behavior and evaluate the consequences it has on a system. The objective of malware analysis is to uncover the fundamental code, functions, and techniques employed by the malware, and to ascertain its methods of infection and interaction with a system. This information can be used to develop countermeasures, such as patches, antivirus signatures, and intrusion detection rules, to protect against similar threats in the future. [6].

Malware analysis can be performed using various techniques, including static analysis, dynamic analysis, and reverse engineering. The process involves isolating the malware in a controlled environment, executing or disassembling it, and observing its behavior and interactions with the system. The outcomes of this analysis can be leveraged to establish a thorough comprehension of the malware, including its functionalities and the potential repercussions it may have on an organization [7].

Various forms of malware analysis exist, with Static and dynamic analysis emerging as the primary methods employed in malware analysis to scrutinize malware behavior and gain insight into its operations.

1.3.1 Static Analysis

The process of inspecting the code and structure of malware without engaging its execution, usually by disassembling or decompiling the binary code. This approach provides insight into the malware's structure and functionality, such as the target system, data it steals or the actions it performs. [8].

Below are several prevalent methods employed in static malware analysis:

i. File scanning and signature detection: This entails cross-referencing the malware code with a repository of recognized malware signatures in order to detect and categorize it [6].

ii. Disassembly and decompilation: This involves converting the binary code of the malware into a human-readable form, such as assembly code or a high-level programming language, for closer examination. [9].

iii. String and symbol analysis: This involve examining the strings and symbols in the malware code to understand its functionality and identify its behavior. [9].

iv. Code analysis: This involves examining the structure and functionality of the malware code to understand its behavior and identify any malicious code. [4].

v. Code emulation: This involves creating a virtual environment to simulate the malware's behavior without actually executing it on the system. [4].

1.3.2 Dynamic Analysis

The procedure of running malware within a controlled setting, For instance, in a virtual machine, to monitor its actions and interactions with the system. This approach provides real-time insight into the malware's actions and allows for the detection of malicious behavior that may not be visible in the code itself.

Here are some common techniques used in dynamic malware analysis:

i. Sandboxing: This involves executing the malware in a virtual environment, isolated from the host system, to observe its behavior. [9].

ii. Process monitoring: This involves monitoring the processes and system calls made by the malware during execution to understand its behavior and interactions with the system. [9].

iii. Network analysis: This involves monitoring the network traffic generated by the malware during execution to understand its behavior and potential impact on the network. [6].

iv. Memory analysis: This involves analyzing the memory of a system infected with malware to understand its behavior and identify malicious code in memory [6].

v. Behavioral analysis: This involves observing the behavior of the malware during execution to understand its behavior and interactions with the system [6].

Static analysis and dynamic analysis both serve crucial roles in the field of malware analysis and are frequently employed together to attain a thorough comprehension of the malware's actions and dormant impact.

1.4 Malware Analysis Using Deep Learning Techniques

Deep learning is emerging as a promising approach to address the constraints of existing methods for detecting and classifying malware. Deep learning, which falls under the umbrella of artificial intelligence and relies on artificial neural networks, has found widespread application in diverse domains such as image processing, computer vision, facial emotion recognition, human action recognition, and natural language processing. Despite its achievements in these domains, its adoption in the realm of cybersecurity, especially for malware detection [2], has been somewhat limited thus far. Various deep learning architectures, including deep neural networks, deep belief networks, recurrent neural networks, and convolutional neural networks, are harnessed to enhance model performance. Deep learning provides numerous benefits compared to conventional approaches, including the automatic generation of high-level features, handling unstructured data, managing large datasets, dimensionality reduction, accommodating various learning paradigms, and enhancing accuracy while lowering costs [36].

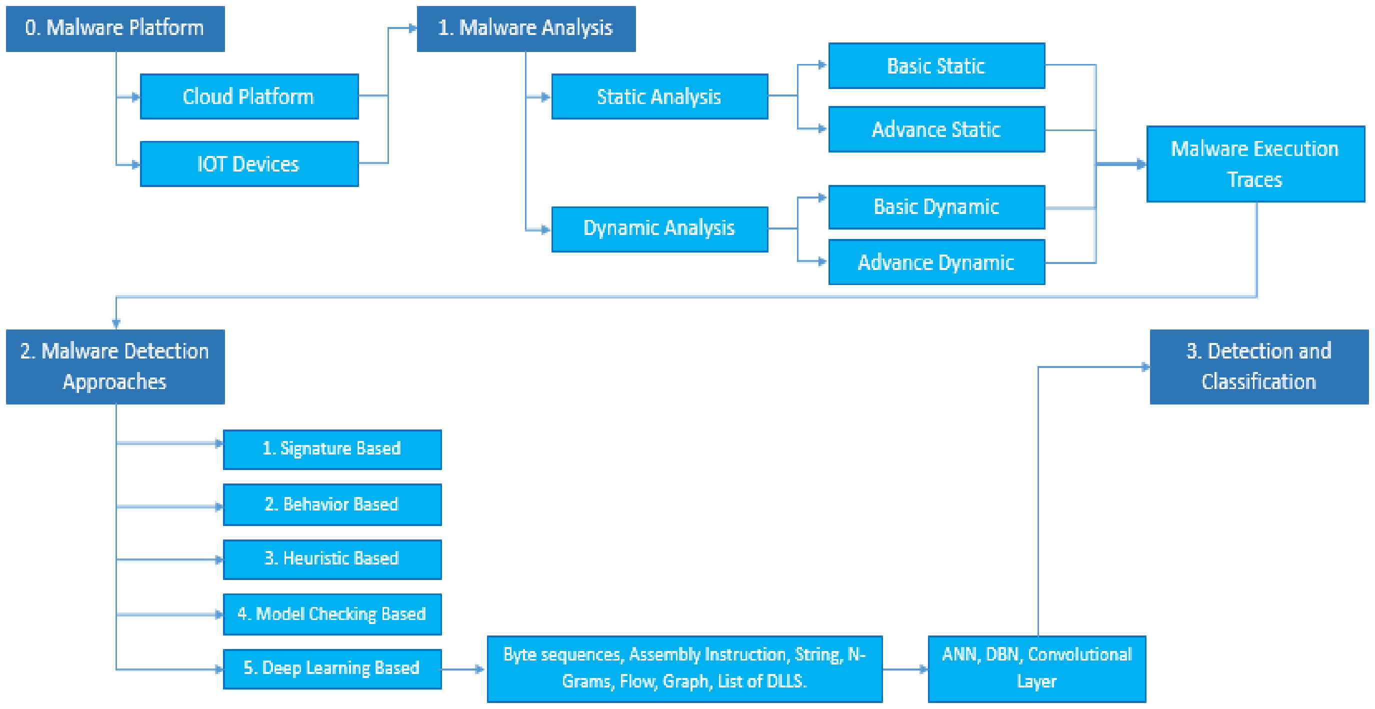


Figure 1.1 The process of analyzing, detecting, and categorizing malware.

1.5 Malware Feature Extraction Techniques

Malware feature extraction involves the identification and extraction of pertinent characteristics or attributes from malware samples, which can then be employed for their classification and detection. The extracted features can be either structural (e.g., opcode sequences, file header information) or behavioral (e.g., network activities, system calls). The following are some of the common malware feature extraction techniques [2], [10], [15]:

i. Opcode N-grams: It refers to a sequence of N instructions executed by the malware, where N can range from 1 to several hundred. These sequences are used as features to classify malware samples [2].

ii. File header information: File header information such as file type, entry point, and import table information can be used as features for malware classification. [10].

iii. System calls: System calls are low-level functions used by applications to interact with the operating system. Monitoring the system calls made by a program can provide valuable information about its behavior and can be used as feature for malware classification. [11].

iv. Network activities: Network activities such as IP addresses, port numbers, and protocol types can be used as features to classify malware samples [15].

v. API calls: API calls refer to the functions that applications use to interact with the operating system or other applications. Monitoring the API calls made by a program can provide valuable information about its behavior and can be used as feature for malware classification [10].

vi. Entropy: Entropy is a measure of randomness in a sequence of data. In malware analysis, entropy can be used as a feature to classify malware samples based on the randomness of their code. [10].

vii. N-grams are sequences of n symbols in the binary code of a malware sample. By counting the frequency of these sequences, unique features of the malware can be identified. [11].

viii. M-bag is a technique that combines N-grams with bag-of-words models. In this technique, the malware binary is divided into overlapping windows, and the frequency of N-grams within each window is calculated. The generated feature vectors can subsequently be employed for the classification of malware [12].

ix. K-tuple is a technique that focuses on the relationships between different parts of the malware binary code. In this technique, the binary code is divided into k-tuple sequences, and the frequency of these sequences is calculated. This frequency information can then be used as a feature to differentiate malware from benign software [13].

These represent some of the prevalent techniques for extracting features from malware, widely employed in malware analysis. The choice of features and the process of extracting them can greatly influence the precision and effectiveness of both malware classification as well as detection.

Malware feature extraction is the procedure of identifying and extracting pertinent data from a malware specimen, which can be utilized to distinguish it from legitimate software. N-gram technique, m-bag technique, and k-tuple technique are three common extracting relevant attributes techniques used in malware analysis.

1.5.1 N-Gram Technique, M-Bag Technique and K-Tuple Technique

The n-gram feature extraction technique is a commonly employed method for detecting and classifying malware. It utilizes both static and dynamic analysis attributes to generate features, employing consecutive system calls according to a defined n value, as described in reference [12]. As an illustration, when considering the sequence of system calls in the sample program, if they follow the order, P = (6, 7, 8, 9, 10), the 2-gram and 4-gram would be {(6, 7), (7, 8), (8, 9), (9, 10)} and {(6, 7, 8, 9), (7, 8, 9, 10)} respectively [10]. The tuple and bag techniques bear resemblance to n-grams, but there's a distinction: in the tuple approach, 'n' can have any spacing, whereas in the bag method, the emphasis lies more on frequencies rather than the sequence, as noted in reference [11]. While n-gram proves to be efficient, its performance diminishes due to the swift expansion of feature numbers. Additionally, various modified n-gram models have been suggested in the literature to extract malware characteristics, resulting in fewer features compared to traditional n-grams, as outlined in reference [13].

1.5.2 Graph-Based Techniques

The graph-based feature extraction technique is another popular method used in malware analysis. In this technique, the behavior of a malware sample is represented as a graph structure where nodes represent system calls, API functions, or other relevant behavior, and edges represent the relationships between these nodes [15]. Each graph is treated as a feature vector and used in machine learning algorithms for classification and detection purposes. This approach offers a more extensive portrayal of a malware sample's behavior, aiding in the capture of intricate relationships among the nodes. Moreover, it reduces the dimensionality of the feature space compared to the n-gram or bag of words approach. Moreover, graph-based representations exhibit resilience to minor code variations, rendering them well-suited for the detection of novel iterations of established malware [14].

1.5.3 Vision-Based Techniques

The vision-based method for extracting malware features entails the transformation of a malware sample into an image, followed by the utilization of computer vision algorithms to derive features from it. The malware binary is transformed into a grayscale image, with each bit being depicted as a pixel. The resulting image is then processed using computer vision techniques such as edge detection, thresholding, and feature extraction to generate features that represent the malware [16]. These characteristics can subsequently be applied to categorize the malware as either benign or malicious, and they can also serve to pinpoint resemblances and distinctions among various malware samples. This technique is particularly useful for detecting malware that is designed to evade detection by traditional signature-based methods. The visual representation of malware allows for the easy identification of patterns and features that can be used to distinguish between different malware families [17].

1.6 Malware Visualization

Visualizing malware as an image entails converting the malware's binary code into an image format for analysis and visualization. In this procedure, each byte of the malware code is interpreted as a pixel within the image. The resulting array is organized in a two-dimensional arrangement and displayed as a grayscale image, where values range from 0 (representing black) to 255 (representing white), as described in reference [16].

The key advantage of this visual approach lies in its ability to facilitate the differentiation of various segments within the binary code and offer a rapid means of detecting both similarities and distinctions among different malware specimens. Since malware creators often recycle portions of old code to produce new variants, reusing old malware can result in binaries that closely resemble each other. By representing malware as an image, it becomes feasible to identify even minor alterations while preserving the overall structure of samples belonging to the same malware family, as highlighted in reference [39].

1.7 Problem Statement

The spread of malware is a prevalent issue in computer networks and various applications. In recent times, classifying malware has become a challenging task, prompting researchers to propose numerous machines learning and deep learning techniques to address this problem. While deep learning techniques have shown encouraging results, they frequently involve intricate models. To tackle this challenge effectively, there is a growing need to develop models that strike a balance between high accuracy and low complexity. One potential solution lies in employing a cloud-based deep learning model with a reduced number of layers with the utilization of a Mexican hat-based activation function. In essence, the proposal suggests that by optimizing the deep learning model's architecture suggests that employing a cloud-based deep learning model with reduced layers and a specialized activation function like the Mexican hat approach can be an effective strategy to enhance accuracy in malware classification while simultaneously reducing computational demands and achieving cost-effectiveness.

1.8 Research Motivation

Machine learning methods are gaining popularity for classifying a wide range of malware types. However, a majority of the current machine learning techniques employed for malware classification rely on shallow learning algorithms, such as SVM. Recently, Convolutional Neural Networks (CNN), which are deep learning methods, have exhibited superior performance in contrast to conventional learning algorithms, particularly in tasks like image classification. Encouraged by this success, we introduce an innovative deep learning-based framework for malware classification.

1.9 Research Questions

i. What are the various deep learning techniques and major challenges to classify the malware?

ii. How to overcome the model complexity and enhance its efficiency?

iii. What methods can be employed to assess and contrast the effectiveness of the suggested malware classification system across various malware datasets over a cloud-based collaborative integrated development environment?

1.10 Research Objective

i. To analyze the various state of art Deep learning models that reduce model complexity and improve model accuracy.

ii. To develop a cloud-based deep learning model for malware classification that operates efficiently with limited cloud resources, exhibiting low model complexity and high accuracy.

iii. To assess the model's performance across various available datasets.

1.11 Thesis Organization

Rest of the thesis is structured as below:

Chapter 2 provides an extensive literature review that covers prior research papers. In the context of machine learning, the process of malware detection and classification entails using machine learning algorithms to identify and classify malicious software.

Chapter 3 provides us with the methodology, which encompasses a proposed framework for classifying malware, along with an in-depth exploration of the MALIMG and MALEVIS Datasets.

Chapter 4 provides us with implementation and evaluation, which offers a comprehensive understanding of the libraries, environment, our proposed model, and the results obtained from both datasets, i.e. MALIMG and MALEVIS. Additionally, it presents the results of all four-performance metrics.

Chapter 5 provides the final insights encompassing the entire thesis.

CHAPTER 2

LITERATURE REVIEW

2.1 Approaches for Detecting and Classifying Malware

In the field of machine learning, malware detection and classification involve employing machine learning algorithms to recognize and categorize malicious software. These algorithms undergo training using extensive datasets comprising both malware samples and benign software to grasp their distinctive attributes. In the detection phase, these algorithms assess newly encountered software to ascertain its malignancy by comparing it to established malware samples. In the classification phase, the algorithms categorize the detected malware into different types, such as viruses, Trojans, worms, spyware, and adware, based on its features and behavior. In comparison to conventional signature-based techniques, machine learning algorithms can enhance both the precision and speed of detecting and categorizing malware. There are several different approaches to achieving effective malware detection and classification, each with its own advantages and limitations. Some of the common approaches include:

2.1.1 Approach for Malware Detection and Classification Based on Signatures

The method of malware detection and classification based on signatures entails recognizing malware by matching its distinct attributes, such as file signature or hash value, with a repository of established malware profiles. If a file is identified as having a signature matching that of known malware, the signature-based system will label the file as malicious. The database of known malware is typically maintained by a security vendor and is updated regularly to keep up with the latest threats. Signature-based systems can detect known malware quickly and accurately, but they may not be able to detect new, unknown malware or malware that has been modified to evade detection. This occurs because the signature-based approach solely verifies precise matches with known malware and does not endeavor to assess the behavior or attributes of the malware in order to ascertain its malignancy.

A signature comprises a distinct collection of bits that serves to identify the structure of a program. It finds widespread application in malware detection [6], [9], [18]. The procedure commences by recognizing static attributes from executable files, subsequently forming signatures from these attributes and archiving them in a database. When there's a need to scrutinize a suspicious file, its signature is extracted and matched against known signatures within the database. Based on this comparison, the file is labeled as either malicious or benign. Respectively this approach is commonly referred to as signature-based malware detection and proves swift and efficient in identifying established malware. Nevertheless, it falls short when it comes to detecting novel or zero-day malware, as noted by Scott [19]. Signature-based malware detection presents certain limitations, including its inability to spot new variations, scalability challenges, and reliance on human intervention. Griffin et al. [20] have put forth an automated signature extraction technique that employs library identification methods and diversity-based heuristics to minimize erroneous identifications. Tang et al. [21] have elucidated a simplified regular expression signature method that employs bioinformatics techniques for detecting polymorphic worms. Additionally, Liu and Sandhu [22] have proposed a fingerprint-based signature generation approach geared toward detecting hardware-based malware through the creation of cryptographic hash-based signatures. The fundamental characteristics of signature-based malware detection and classification include:

i. Signature Creation: Cybersecurity experts analyze and reverse-engineer malware samples to identify distinct patterns or sequences of code that are characteristic of each malware variant. These patterns are known as signatures.

ii. Signature Database: A database containing signatures of known malware variants is maintained. A new file's signature is compared to the signatures in the database when it is scanned.

iii. Matching Process: The procedure entails matching the scanned file's or code's digital signature to signatures stored in the database. If a match is discovered, the file is classified as malicious based on the known malware strain and is then removed.

iv. Rapid Detection: Signature-based detection is fast and efficient for identifying malware that matches known signatures. It is particularly effective against well-established malware.

v. Virus Definition Updates: To stay effective, signature-based antivirus solutions require regular updates to their signature databases. New malware variants are constantly emerging, requiring antivirus software to have the latest signatures to detect them.

vi. Limitations: Signature-based detection has limitations. It is not effective against new or unknown malware that lacks a matching signature. Attackers can also use techniques like polymorphism (changing the code while preserving functionality) to evade signature-based detection.

vii. False Positives and Negatives: There is a risk of false positives (legitimate files misidentified as malware) and false negatives (malware that does not match any known signature) in signature-based detection.

viii. Known Malware: Signature-based detection is well-suited for identifying known malware that has been previously documented and analyzed.

ix. Efficiency: Signature-based detection is computationally efficient and can quickly scan large numbers of files for known malware.

x. Complementary Approaches: To get over signature-based detection's drawbacks, many modern antivirus solutions use a combination of approaches, including behavior-based analysis and machine learning, to provide more comprehensive protection against a broader range of threats.

While signature-based detection is a foundational method in malware detection and classification, it is most effective when used as part of a multi-layered security strategy that incorporates other approaches to handle new and evolving malware threats.

2.1.2 Approach to Detecting and Classifying Malware on Behavior-Based

The behavior-based method for malware detection and categorization leverages the actions of a program or system as a means to identify and classify malware. This strategy, often referred to as dynamic analysis, entails scrutinizing the program's behavior during its execution. Within this behavioral framework, the program operates within a controlled setting, such as a sandbox, where its actions are meticulously observed and assessed. By comparing the program's behavior to established patterns of malicious activity, it is possible to ascertain whether it exhibits malicious characteristics [2].

The primary benefit of the behavior-driven methodology lies in its ability to identify novel and unfamiliar malware since it does not depend on predetermined signatures or definitions. However, this approach can also be more time-consuming and resource-intensive compared to signature-based detection, as it requires the program to be executed. Additionally, behavior-based detection may not be able to detect malware that does not execute any malicious behavior during the analysis. Despite these limitations, the behavior-based malware analysis and classification approach is a critical component of modern malware detection and defense systems.

The behavior-based detection method entails the observation and scrutiny of a sample program's actions, and based on these observations, hence the program is categorized as either malicious or benign. This method has three components: collecting behaviors, developing characteristics, and using machine learning algorithms to determine if the program is harmful or safe [2].

The identification of behaviors entails the utilization of system calls, the application programming interface calls, and alterations in files, registries, and computer system network activity. Simply put, behaviors are analyzed by looking at the sequence or frequency of system calls and file-registry operations. Through the process of grouping these behaviors and constructing sequences, distinctive characteristics are identified. Despite changes in the program's source code, the program's behaviors will not change entirely. This allows for the detection of multiple variants of malicious software, as well as newly discovered malware. However, the primary drawback of behavior-based detection lies in the fact that malware might not manifest all its authentic behaviors when operating within secure environments like virtual machines and sandboxes. The graph model methodology for malware detection is described by Kolbitsch et al. in their work [23].

In a diagram, system calls are depicted as nodes, and transitions between them are denoted by edges. These connections in-between system calls are established by utilizing the output of one system call as the input for another. Resulting program diagram is subsequently compared to preexisting diagrams, and the examined sample is classified as either malware or benign based on this comparison. Any novel behaviors discovered during the analysis can also be incorporated into the diagram in a dynamic manner.

Lanzi et al. [12] introduced a system-focused behavioral model that considers the variation in how malicious and benign software interact with system resources, such as the directories, the files, the registries, and so on. This discrepancy in interaction patterns was leveraged to construct behavior sequences derived from system calls, which were subsequently employed to classify software as either malicious or benign. The approach they proposed made use of the n-gram technique; however, it encountered difficulties in effectively distinguishing between malicious and benign software due to the substantial number of behavior sequences generated using the n-gram technique.

Chandramohan et al. [10] introduced the BOFM (Behavior Oriented Feature Model) as an approach aimed at reducing the abundance of properties required for malware detection. This method involves the conversion of interconnected system calls into meaningful behaviors, which serve as the basis for establishing properties. Redundant features are subsequently removed, and a feature vector is crafted using the retained properties. ML algorithms are then applied to execute the classification process.

Singh et al. [24] developed a behavior-based method for detecting malware using multiple API system calls. This approach entails generating multiple API sequences through the utilization of depth-first search and n-grams, followed by the assessment of similarities between software using metrics like the Dice coefficient, the cosine coefficient, and the Tversky index. The resulting sequences are then classified using algorithms based on machine learning.

Aslan et al. [25] In their work, Aslan and colleagues (Aslan et al., 2019) introduced a malware detection model known as SCBM (Semantic-Centric Behavioral Model), which is rooted in behavior analysis. This method extracts semantically correlated features from scrutinized program samples, considering both system paths and behaviors to discern malicious behavioral patterns from benign ones. Notably, this proposed model generates a reduced number of features when compared to traditional methods like n-grams. Testing results indicate its effectiveness in addressing both known and unknown malware, as demonstrated by performance metrics including detection rate, false positive rate, f-score, and accuracy.

In reference to [26], a novel hybrid methodology is presented, which integrates dynamic analysis with 1. cyber threat intelligence, 2. machine learning, and 3. data forensics. This approach predicts IP reputation during the pre-acceptance phase by employing extensive data forensics techniques and distinguishes related zero-day attacks through behavioral analysis combined with a decision tree algorithm. The effectiveness of this approach is assessed using metrics such as f-measure, precision, and recall, and the findings demonstrate its performance to be on par with other prominent methods in the same domain.

In [27], A novel approach for malware detection named APTMalInsight is introduced. This approach identifies and recognizes Advanced Persistent Threats (APTs) through the use of an information and ontology knowledge framework. The method determines feature vectors for APT malware and is capable of detecting and grouping them with a high degree of accuracy.

Key components and characteristics of behavior-based malware detection and classification include:

i. Dynamic Analysis: Behavior-based detection involves analyzing malware in a controlled environment, often referred to as a sandbox. In this environment, the malware is executed and its behavior is monitored, recorded, and analyzed in real-time.

ii. Behavioral Patterns: Malware can exhibit various behaviors, such as making unauthorized changes to files, attempting to access sensitive data, communicating with command and control servers, or modifying system settings. By observing these behaviors, security analysts can identify malicious intent.

iii. Anomaly Detection: Behavior-based approaches focus on identifying anomalies or deviations from normal behavior. Malware often behaves differently from legitimate software, making it possible to detect unusual actions that may indicate a threat.

iv. Data Collection: During dynamic analysis, data is collected on the malware's actions, such as file system changes, network communication, memory usage, and system calls. This data is used to build a behavioral profile of the malware.

v. Feature Extraction: Significant attributes or traits are derived from the gathered data. These characteristics may encompass patterns of network traffic, API calls, system resource usage, and interactions with files and processes.

vi. Machine Learning: The extracted features can be leveraged with machine learning techniques, such as clustering and classification algorithms, to develop models capable of distinguishing between benign and malicious behavior.

vii. Classification: Malicious software can be categorized into various groups according to its behavior, including ransomware, spyware, trojans, and more. Behavior-based classification allows for more accurate identification of malware variants that may share similar traits.

viii. Real-Time Detection: Behavior-based detection can provide real-time protection by analyzing and detecting malicious behavior as it occurs. This proactive approach is valuable for identifying previously unknown threats.

ix. Evasion Techniques: Malware authors may use evasion techniques to avoid detection, such as delaying malicious actions or disguising their behavior. Behavior-based analysis is designed to detect these evasive tactics.

x. Limitations: Behavior-based detection proves its efficacy when dealing with novel and zero-day malware, it may have limitations when malware exhibits stealthy or sophisticated behavior. Additionally, false positives can occur if legitimate software behaves in unusual ways.

2.1.3 Approach to Malware Detection and Classification Based on Heuristics

Heuristic-based malware detection is a technique for recognizing malware by employing a predefined set of rules or patterns derived from prior knowledge or experience. This method operates on the premise that malware frequently adheres to identifiable and predictable patterns or behaviors, which can be employed to categorize the program as either malicious or benign.

Heuristics serve as a means to identify malware without relying on specific signatures or detailed information about a particular threat. This approach is particularly valuable for identifying novel or unfamiliar malware and can serve as a valuable complement to other malware detection methods, including signature-based and behavior-based approaches.

A heuristic-based malware detection and classification approach uses a combination of techniques and is driven by the experience. It relies on rules and machine learning methods to generate signatures using both string-based and behavior-related features, as explained in reference [28]. This method is widely employed for the identification of diverse malware types, including previously unencountered ones. The system is initially trained using specific features and subsequently employs test data to spot anomalies. While it exhibits a commendable success rate in identifying new malware, it is susceptible to a notable number of false positives and the false negatives, primarily attributable to optimization challenges.

A malware detection system was proposed by Ye et al.[29] introduced a malware detection system designed with the goal of identifying polymorphic and emerging malware strains that often elude conventional antivirus software. This system conducted an analysis of program Application Programming Interface sequences and generated pertinent rules through the utilization of the FP-growth algorithm. Subsequently, classification algorithms were employed to ascertain whether the program files were of a malicious or benign nature. The application of this system was primarily focused on Windows executable files. While the proposed approach demonstrated superior performance compared to certain antivirus scanners, it fell short of meeting the expectations for effectively detecting unknown malware.

In their paper [30], Canali et al. and his team introduced a behavior-based signature technique in which meaningful behavior patterns were crafted from combinations of system calls. These signatures were composed of elements that could encompass system calls, system calls along with their arguments, behaviors, or behavior groups with arguments. The authors organized these elements using n-gram, k-tuple, and m-bag models to formulate the signatures. They reported that their proposed approach efficiently identified malware.

Islam et al. devised a detection system that integrates elements from both static and dynamic analyses [31]. This system encompasses three distinct feature categories: 1. method lengths quantified in bytes, 2. printable string data, and 3. system calls along with their associated parameters. These features are amalgamated to create a feature vector, which is subsequently subjected to classification through machine learning algorithms. To address packed malware, a dynamic heuristic approach was introduced [32]. This technique initiates by computing the frequencies of API calls, identifying the API calls most strongly correlated with malware, and ultimately employing a Naive Bayes classifier and Levenshtein distance for the purposes of training and classification. The authors assert that their proposed method yields satisfactory outcomes across various variations of packed malware.

Prominent attributes of heuristics-based malware detection and categorization encompass:

i. Rule-Based Analysis: Heuristic techniques employ established rules or guidelines to detect potential malware, relying on known behaviors, characteristics, or patterns frequently linked to malicious software.

ii. Pattern Recognition: Heuristic analysis focuses on recognizing patterns, behaviors, or characteristics that are typical of malware. These patterns might include unauthorized changes to system files, attempts to access sensitive data, or communication with external servers.

iii. Behavioral Traits: Heuristic analysis can detect behaviors that deviate from normal or expected behavior. Malware often exhibits abnormal behavior, making it possible to identify actions that indicate a potential threat.

iv. Zero-Day Detection: Heuristic analysis is effective for identifying zero-day attacks, which are attacks that target vulnerabilities that are not yet known or patched by security updates.

v. Unseen Variants: Heuristic methods can detect previously unseen malware variants that do not have well-defined signatures. This makes them valuable for detecting new and evolving threats.

vi. Dynamic Analysis: Heuristic methodologies may incorporate dynamic analysis, where files are run within a controlled environment (sandbox) to observe their conduct. This can help to detect hidden or obfuscated malicious actions.

vii. Overcoming Evasion Tactics: Attackers may use tactics to evade detection, such as delaying malicious actions or employing obfuscation techniques. Heuristic analysis can help detect these evasion techniques.

viii. False Positives: Heuristic analysis can produce false positives (legitimate files flagged as malware) due to its reliance on predefined rules. Tuning the rules and algorithms can help reduce false positives.

ix. Complementary Approach: Heuristic analysis complements other malware detection approaches, such as signature-based detection and behavior-based analysis, to provide comprehensive protection against a wide range of threats.

x. Customization: Security solutions can be customized to apply specific heuristics based on the organization's needs and the types of threats they are most concerned about.

Heuristic-based analysis plays a crucial role in modern cybersecurity strategies. It allows security tools to detect and respond to new and emerging threats that may not be covered by existing signature databases. Nonetheless, achieving an equilibrium between sensitivity and the false positives is crucial to ensure effective protection without unnecessary disruptions to legitimate user activities.

2.1.4 Approach to Malware Detection and Classification Based on Model Checking

Model checking is a formal verification technique used to analyze the behavior of complex systems. It involves constructing a model of a system and verifying its properties against a set of predefined rules. This technique has been successfully applied in the field of software engineering for bug detection, however it can also find applications in the realm of malware detection and categorization. In this context, a system model is formulated., including the behavior of its software components and interactions with external entities. This model is then analyzed using a model checker, which searches for violations of predefined security rules. If any violations are found, it indicates the presence of malware in the system. Once the malware is detected, it can be further classified based on its behavior and properties. For example, it can be classified as a trojan, a worm, or a virus. This information can be used to identify the type of attack, the source of the malware, and the potential impact on the system [2].

One benefit of this method is its ability to offer a systematic and automated means of identifying and categorizing malware, mitigating the chances of encountering false positives and false negatives. Furthermore, it can be effectively employed for the detection of intricate and advanced malware that might pose challenges for conventional detection techniques.

However, this approach requires a high level of expertise in model checking, as well as a well-defined set of security rules. It also requires a large amount of computational resources, as the model checker has to analyze the entire system and its interactions. In summary, the model checking-based approach for malware detection and classification holds potential as a solution for addressing the escalating issue of malware. However, its practicality and efficiency for real-world systems necessitate additional research and development efforts, as indicated in reference [2].

Holzer et al. [33] introduced a verification system designed for the identification of malicious software. This system employs CTPL (computation tree predicate logic) as the specification language to define malicious behaviors and extracts a finite state model from disassembled executable files. The model controller subsequently assesses the specification, marking the sample as malicious if it is deemed correct, and as benign otherwise. This system exhibits the capability to identify malware within families that employ similar attack techniques, as well as novel malware variants displaying similar behaviors. The study concluded that the model-checking-based approach resulted in a more precise determination of malware's semantic attributes when compared to traditional detection methods, leading to a significant enhancement in the overall detection accuracy by as much as 70%.

Kinder et al. introduced a forward-looking approach to malware detection, employing the model-checking method [34]. This method has the capability to identify diverse types of computer worms without necessitating frequent signature updates. Control flow charts are extracted from executable files and subjected to automatic validation using specifications defined in a novel language called CTPL. Experimental results indicate that this method achieved a low rate of false positives while effectively detecting various forms of worms.

Critical features of malware detection and classification rooted in the model-checking approach encompass:

i. Formal Verification: Model checking involves formal methods from computer science and mathematics to rigorously analyze software behavior. It can be applied to executable code or abstract models of the software's behavior.

ii. Properties and Specifications: The model checking process involves defining properties or specifications that describe expected behaviors of legitimate software. These properties can include assertions about memory usage, data flow, system calls, and more.

iii. Automated Analysis: Model checking is an automated process that systematically explores all possible execution paths and behaviors of a program. It checks whether the program violates specified properties or requirements.

iv. Behavioral Analysis: Model checking can identify potential malware by detecting deviations from expected behaviors. If the program's behavior diverges from the specified properties, it may be flagged as suspicious or potentially malicious.

v. State Space Exploration: Model checking involves exploring the entire state space of a program to ensure that it adheres to the specified properties under all possible conditions. This exhaustive analysis can reveal hidden or subtle vulnerabilities and behaviors.

vi. Zero-Day Detection: Model checking-based approaches are capable of detecting zero-day vulnerabilities and attacks by analyzing program behaviors based on formal properties.

vii. Verification Logic: Formal verification techniques, such as temporal logic and automata theory, are used to specify the properties to be checked and to reason about the correctness of a program's behavior.

viii. Resource Consumption: Model checking-based methodologies can additionally scrutinize resource utilization patterns, including memory usage and execution duration, in order to pinpoint programs displaying irregular or malicious conduct.

ix. Challenges: Model checking can demand significant computational resources, particularly when applied to large and intricate software. Extending model checking to practical, real-world software necessitates a thoughtful assessment of the analysis's extent and thoroughness.

x. Complementary Approach: Model checking complements other malware detection techniques, such as heuristic analysis, behavior-based analysis, and machine learning, to provide comprehensive protection against a wide range of threats.

Model checking-based malware detection is particularly suitable for critical systems, where correctness and safety are of utmost importance. It is well-suited for verifying the behavior of software against formal specifications, enabling early detection of vulnerabilities and potential malware activities. However, its computational demands and complexity require careful consideration when applying it to practical applications.

2.2 Approach to Malware Detection and Classification Utilizing Deep Learning

n this strategy, deep learning algorithms, such as artificial neural networks, undergo training on an extensive dataset containing both malware and benign software. The aim is for these algorithms to learn the intrinsic patterns and distinguishing features that set them apart. Subsequently, the trained model can be employed to determine whether new, unobserved software should be categorized as malware or benign, contingent upon its resemblance to the acquired patterns. This method has exhibited promising outcomes, accurately identifying and classifying various forms of malware.

The deep learning-based detection method performs well and reduces feature dimensions effectively, However, it remains susceptible to evasion attacks [2]. Extending the number of hidden layers is a time-consuming process and provides only marginal performance gains. As of now, the adoption of deep learning in malware detection and classification is not widespread, necessitating further research to thoroughly assess its capabilities. The literature documents instances of deep learning-based approaches in malware detection, which are summarized as follows.

Saxe and Berlin [36] have put forward a concept for a malware detection system based on deep neural networks, which leverages two-dimensional software features. This system comprises three pivotal components: the extraction of four unique and complementary features from both malicious and non-malicious samples, the establishment of a deep neural network encompassing 1 input layer, 2 hidden layers, and 1 output layer, and computation of neural network outputs utilizing the calibrator score.

Huang and Stokes [37] described the MaNet architecture, a multitasking approach to malware classification, it trains on both malicious and benign samples using data acquired from dynamic analysis. This architecture utilizes the ReLU activation function, resulting in a reduction of epochs and error rate. Despite being noted as outperforming a traditional neural network architecture, the addition of an extra hidden layer did not enhance performance and the model was not immune to evasion attacks.

In a study by Ye et al. [38], a system was proposed to detect zero-day malware using heterogeneous deep learning. The system utilizes multi-layer constrained Boltzmann machines with associated memory and involves two phases: pre-training and fine-tuning. In the pre-training phase, features are acquired from both labeled and unlabeled files, enabling the identification of unique characteristics for each file. The fine-tuning phase employs supervised learning to differentiate between malware and benign files. The research demonstrated that this approach enhances performance when compared to conventional shallow learning methods.

Roseline and her team introduced an intelligent, vision-based approach to malware detection and classification [4]. This method employs a layered ensemble structure reminiscent of deep learning. Program executables undergo a transformation into 2D images, and the resulting patterns are harnessed for categorizing various malware variants into their respective classes. This classification process utilizes a deep forest approach coupled with sliding window scanning and cascading layering influenced by Convolutional Neural Networks (CNNs). Notably, the authors contend that their method does not necessitate backpropagation or hyperparameter tuning, and it demonstrated successful outcomes in the detection and classification of malware variants on the MALIMG and MALEVIS datasets.

In reference [5], a hybrid approach to malware classification is introduced, merging features derived from deep convolutional neural networks with fractal texture analysis based on segmentation. The authors of the paper employed pre-trained models like AlexNet and Inception-v3 to capture features from malware images, subsequently partitioning them into 25 classes. These extracted features underwent classification through a support vector machine, decision tree, and k-nearest neighbor algorithms. The outcomes reported in the paper indicate a notably high accuracy rate on the MALIMG dataset.

Essential attributes of malware detection and classification utilizing deep learning encompass:

i. Neural Network Architectures: Deep learning models utilize intricate neural network structures, including Convolutional Neural Networks (CNNs) for image analysis and Recurrent Neural Networks (RNNs) for sequence data. Moreover, more sophisticated architectures such as Long Short-Term Memory networks (LSTMs) and transformer models can also find application

ii. Feature Learning: Deep learning models acquire pertinent features and patterns directly from raw data, obviating the necessity for explicit feature engineering. This capability empowers the model to capture intricate relationships inherent in the data.

iii. Training Data: Deep learning models necessitate substantial volumes of labeled training data to acquire the ability to differentiate between benign and malicious behaviors. These data include features extracted from malware samples and legitimate software.

iv. Representation Learning: Deep learning models have the capacity to acquire hierarchical and abstract representations of data, empowering them to encompass nuanced and intricate attributes that could serve as indications of malware.

v. Behavioral Analysis: Deep learning models can analyze the behavior of files, programs, and network traffic to identify patterns that match known malware behaviors or deviations from normal activities.

vi. Image Analysis: Some deep learning models treat binary files as images and use techniques like CNNs to identify visual patterns within the file's binary representation.

vii. Sequence Analysis: For malware detection involving sequences of operations or system calls, RNNs and LSTMs can be employed to analyze the sequential behavior of programs.

viii. Transfer Learning: Transfer learning entails the utilization of pre-trained deep learning models trained on extensive datasets and adapting them for the specific task of malware detection. This approach proves beneficial in mitigating challenges related to limited data availability.

ix. Zero-Day Detection: Deep learning models can be effective at detecting new and unknown malware variants due to their ability to generalize from learned patterns.

x. Scalability: Deep learning models can scale to handle large and diverse datasets, allowing for effective detection and classification across a wide range of malware types.

xi. Challenges: Deep learning models may require significant computational resources for training and inference. Careful attention to hyperparameters, regularization, and data augmentation is essential.

xii. Interpretability: Deep learning models can be challenging to interpret, making it important to develop methods to understand their decision-making processes and provide explanations for their predictions.

The domain of malware detection and classification through deep learning is dynamically progressing, marked by continuous research and advancements directed at enhancing the precision, efficiency, and interpretability of deep learning models. It serves as a valuable complement to other malware detection methods, augmenting the overall efficacy of contemporary cybersecurity strategies.

2.2.1 Convolutional Neural Networks

Convolutional Neural Networks (abbreviated as ConvNets or CNNs) represent a category of deep, feed-forward artificial neural networks, predominantly employed in tasks such as image recognition, classification, and speech recognition. These networks encompass multiple layers of artificial neurons, including convolutional, pooling, and fully connected layers. Their inherent ability lies in learning hierarchical data representations by applying filters to raw input and gradually diminishing spatial dimensions. This architectural design empowers CNNs to autonomously and flexibly acquire spatial hierarchies of features from the input dataset, thereby reducing the necessity for manual feature extraction.

Its design is tailored for processing data organized in a grid-like manner, as is the case with images. This network employs multiple filters on the input data to acquire hierarchical representations or features from the input. The filters undergo learning through a process known as convolution, in which the network applies a small matrix of weights to local sections of the input data, producing a new transformed representation. Subsequently, pooling layers are employed to down-sample the feature maps and diminish spatial dimensions. Finally, the feature maps are input into a fully connected layer to make predictions grounded in the learned features. The distinctive architecture of CNNs enables them to autonomously learn valuable features directly from the data, alleviating the requirement for manual feature engineering. As a result, they have become a favored choice for tasks in computer vision and Natural Language Processing (NLP).

In a CNN, images are first converted into a matrix format, and the network uses these matrices to determine which image belongs to which label. During training, the network learns the effects of the differences between images on their corresponding labels. In the testing phase, it applies these learned relationships to predict the labels of new images. A CNN comprises three principal elements: 1. convolutional layer, 2. pooling layer, and 3. fully connected layer. The convolutional and pooling layers are in charge of the feature extraction procedure, whereas the fully connected layer is responsible for carrying out the classification task.

Here's a simple example to illustrate the convolution operation:

Suppose you have a 5x5 input matrix (representing an image) and a 3x3 filter matrix, also called a kernel, shown below:

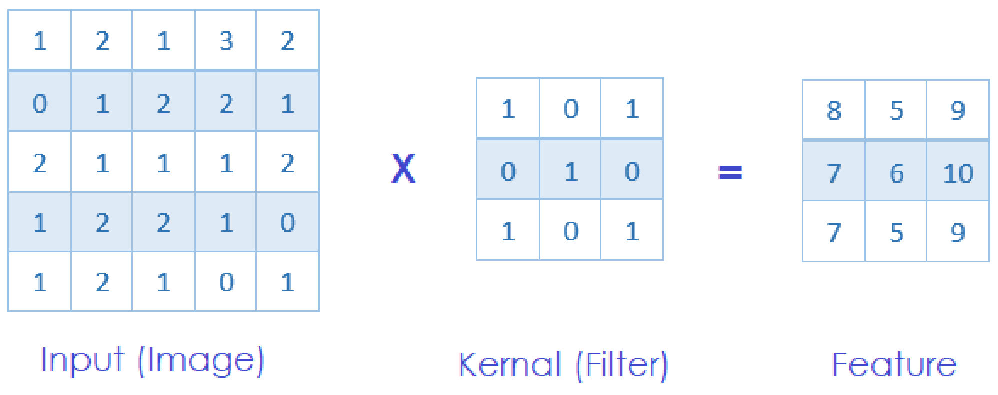


Figure 2.1 CO with Karnal 3 and Stride 2.

The convolution process entails moving the kernel across the input matrix, conducting element-wise multiplication for each overlapping region, and subsequently summing the outcomes. This process yields a fresh matrix, referred to as a feature map, with diminished dimensions. In the provided instance, the resultant feature map will have dimensions of 3x3, as depicted above.

The mathematical formula for the convolution operation as follows in **Equation 2.1**.

$$(f * g)(t) = \sum_{\tau = -\infty}^{\infty} f(\tau) g(t - \tau) \tag{2.1}$$

where f and g are the input and kernel matrices, respectively, and t is the index of the output matrix.

Key characteristics of Convolutional Neural Networks include:

i. Convolutional Layers: The core building blocks of CNNs are convolutional layers. These layers apply convolutional operations to input data using small filters (kernels) to extract local features and patterns. Convolutions empower the network to grasp spatial hierarchies of characteristics, including edges, textures, and shapes.

ii. Pooling Layers: Pooling layers decrease the spatial dimensions of the data while preserving crucial features. Widely employed pooling operations include max pooling and average pooling, which assist in reducing computational complexity and enhancing translation invariance.

iii. Feature Hierarchies: CNNs are constructed with multiple layers arranged sequentially. The initial layers are adept at detecting low-level features such as edges and corners, whereas the deeper layers specialize in recognizing high-level features like object components and entire objects.

iv. Learned Features: Unlike traditional image processing methods that use handcrafted features, CNNs automatically learn hierarchical features from data during training. This capability makes CNNs highly effective at tasks like object recognition.

v. Weight sharing is a fundamental concept in CNNs, where weights are shared among distinct input regions. This practice minimizes the parameter count and facilitates the network in generalizing across various portions of the input.

vi. Activation Functions: Activation functions, such as ReLU (Rectified Linear Unit), introduce non-linearity into the network, enabling it to learn complex relationships between features.

vii. Fully Connected Layer: CNNs often end with fully connected layers that combine learned features to make final predictions. These layers can be used for tasks like classification or regression.

viii. Transfer Learning: By fine-tuning pre-trained CNNs on more limited datasets following training on extensive ones like ImageNet, we can employ transfer learning. This approach harnesses the previously acquired features and has the potential to enhance performance.

ix. Data Augmentation: To enhance the diversity of training data and boost generalization, one can employ data augmentation methods such as image rotations, flips, and shifts.

Convolutional Neural Networks (CNNs) have played a pivotal role in achieving cutting-edge performance across various computer vision tasks. Their hierarchical structure and innate capability to autonomously learn pertinent features render them exceptionally well-suited for tasks involving pattern extraction from images or sequences. Although CNNs are frequently associated with image processing, analogous principles can be applied to other structured data types, such as speech and text.

In summary, the convolution layer within a Convolutional Neural Network (CNN) applies filters to the input data, facilitating the acquisition of local features while reducing the input's dimensions and transforming it into a new, condensed representation. Subsequently, the resulting feature map serves as input for the subsequent layer within the CNN. The primary objective of the convolution operation in a CNN is to discern and extract high-level features, such as edges, from the input image. A CNN is not confined to just one convolutional layer, and typically, the initial convolutional layer is responsible for identifying low-level features like edges, colors, gradient orientations, and so forth. As more layers are added to the network, its architecture becomes increasingly proficient at recognizing higher-level features.

2.2.2 Non-Linearity (ReLU)

ReLU, which is short for Rectified Linear Unit, serves as a commonly employed activation function within deep learning networks, particularly finding widespread use in Convolutional Neural Networks (CNNs).

The ReLU function is represented mathematically as in **Equation 2.2**.

$$f(x) = \max(0, x) \tag{2.2}$$

Here, x represents the input to the activation function, and f(x) denotes the resulting output. When x is positive, the output equals x, while for negative x values, the output becomes 0.

An illustration of how ReLU operates can be exemplified as follows: consider the input to the activation function as -2. When we apply the ReLU function to this input, the resulting output is 0, given that the input is negative. Conversely, if the input assumes a value of 5, the output corresponds to 5, owing to the input's positivity. This characteristic of ReLU enables neural networks to represent intricate associations between inputs and outputs, rendering it apt for addressing real-world challenges. Here’s another example to help understand the ReLU activation function. Suppose we have an input value x = -2. When we pass this input through the ReLU activation function, the output will be 0, because the ReLU function maps all negative values to 0, as illustrated in **Equation 2.3**.

$$x = -2$$
$$f(x) = \max(0, x) = \max(0, -2) = 0 \tag{2.3}$$

Now, let's consider another input value x = 5. When we pass this input through the ReLU activation function, the output will be 5, because the ReLU function maps all positive values to the same value as the input. As demonstrated in **Equation 2.4**.

$$x = 5$$
$$f(x) = \max(0, x) = \max(0, 5) = 5 \tag{2.4}$$

In this way, the ReLU activation function helps the network model complex relationships between inputs and outputs, making it suitable for solving real-world problems.

Key characteristics of the ReLU activation function:

i. Non-Linearity: ReLU introduces non-linearity by applying a thresholding operation. This allows neural networks to capture complex relationships and patterns in data that linear transformations cannot model.

ii. Sparse Activation: ReLU activations are sparse, meaning that they can quickly "activate" (output a non-zero value) for positive inputs and "deactivate" (output zero) for negative inputs. This sparsity can improve the network's efficiency and training speed.

iii. Vanishing Gradient: ReLU helps alleviate the vanishing gradient problem that can occur when using activation functions like sigmoid or tanh. In these functions, gradients can become very small for extreme input values, making learning difficult. ReLU gradients remain non-zero for positive inputs.

iv. Common Usage: ReLU finds extensive application in numerous deep learning structures, encompassing Convolutional Neural Networks (CNNs) and feedforward neural networks. Its popularity can be attributed to its straightforwardness and efficacy.

ReLU has some limitations that are as follow:

i. Dying ReLU Problem: In the case of specific negative inputs, the gradient of ReLU becomes zero, essentially causing the neuron's learning to halt during backpropagation. This issue is commonly referred to as the "dying ReLU" problem and can impede or decelerate the learning procedure.

ii. Leaky ReLU and Variants: To address the dying ReLU problem, variations of ReLU have been proposed, such as Leaky ReLU (allowing a small slope for negative inputs) and Parametric ReLU (making the slope trainable).

iii. Output Range: ReLU outputs only positive values or zeros, which can cause exploding activations when used in very deep networks. Techniques like batch normalization can help mitigate this issue.

2.2.3 Pooling Layer

The pooling layer is an integral element within Convolutional Neural Networks (CNNs) and serves the purpose of downsizing the feature maps produced by the convolution layer. The primary objective of pooling is to decrease the spatial dimensions of the feature maps while preserving crucial information essential for classification. Additionally, pooling contributes to the reduction of the network's computational complexity and guards against overfitting by confining the number of parameters.

In a Convolutional Neural Network (CNN), the pooling layer follows the convolutional layer and constitutes the second layer. Its principal objective is to diminish both the count of feature maps and the network's parameters through mathematical operations. In this specific instance, max-pooling was employed. During max-pooling, each feature map retains solely the maximum value within a defined matrix size, leading to a reduced quantity of output neurons. Subsequently, the network advances to the fully connected layer. Another intermediate layer that may find application is the dropout layer, which is instrumental in mitigating overfitting and divergence within the network.

The i-th layer of a Convolutional Neural Network (CNN) encompasses both the convolutional layer and the pooling layer, working in tandem. While the number of such layers can be expanded to capture intricate details within complex images, it necessitates greater computational resources. Following the conclusion of the feature extraction procedure, the resulting output is flattened and subsequently directed into a conventional neural network designed for classification. This concluding step enables the model to comprehend the input image's features and formulate predictions based on these features.

In CNNs, two prevalent pooling operations are frequently employed: max pooling and average pooling.

i. Max Pooling: Max pooling is the prevalent form of pooling operation. In this technique, a pooling window, typically sized as 2x2 or 3x3, traverses the input data. Within each window, the maximum value is chosen and transmitted to the subsequent layer. Max pooling effectively captures the most salient feature in the window and reduces the dimensionality of the data.

ii. Average Pooling: Average pooling bears similarities to max pooling, with the distinction being that it computes the average value within the pooling window instead of selecting the maximum. The utilization of average pooling serves to mitigate the likelihood of giving undue prominence to extreme values within the input data.

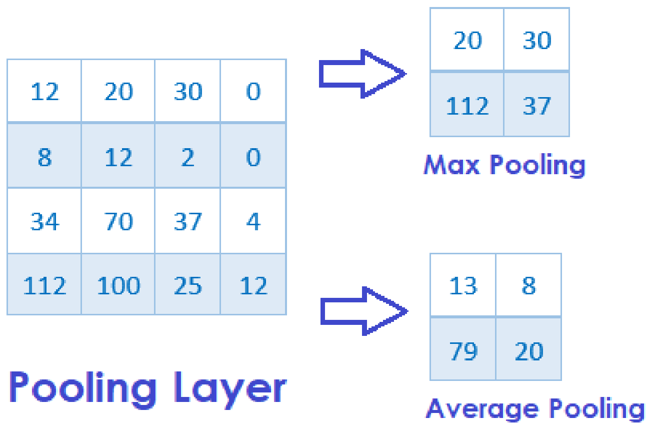


Figure 2.2 Pooling types.

There exist two primary types of pooling operations: max pooling and average pooling. In max pooling, the maximum value is chosen from a set of neighboring pixels within the feature map, while average pooling computes the average value. Max pooling is more commonly employed due to its lower sensitivity to input noise and variations.

Illustrating max pooling with an example, let's consider a 4x4 feature map produced by the convolution layer, as depicted earlier. Max pooling operates by partitioning the feature map into non-overlapping regions and selecting the maximum value from each region. For instance, when employing a 2x2 pooling window, the outcome of max pooling will yield a 2x2 feature map, as illustrated above. The mathematical formula for max pooling is given by **Equation 2.5**.

$$y_{i, j} = \max_{k=0}^{f_{h}-1} \max_{l=0}^{f_{w}-1} x_{s \cdot i + k, s \cdot j + l} \quad (2.5)$$

In this context, $x_{i, j}$ represents the value located at position (i, j) within the feature map, while f_{h} and f_{w} denote the height and width of the pooling window, respectively. Additionally, "s" represents the stride, which signifies the gap between the centers of consecutive pooling windows. Pooling layers offer several benefits in convolutional neural networks:

i. Dimensionality Reduction: Pooling layers are instrumental in diminishing the spatial dimensions of the input data, thereby contributing to the reduction of computational demands and memory requisites in the subsequent layers.

ii. Translation Invariance: Pooling captures the most important features in a local neighborhood, making the network less sensitive to small translations or shifts in the input data.

iii. Feature Robustness: By retaining only the most salient features, pooling layers help the network focus on higher-level patterns and reduce the impact of noise and minor variations.

iv. Regularization: Pooling operates as a form of regularization by mitigating overfitting, compelling the network to acquire more resilient and broadly applicable features.

In summary, the pooling layer diminishes the spatial dimensions of the feature maps generated by the convolutional layer, preserving solely the most crucial information for classification purposes. These compacted feature maps are subsequently employed as input for the ensuing layer within the CNN.

2.2.4 Fully connected layer

The last layer in a Convolutional Neural Network (CNN), also referred to as the classification layer, assumes the role of making the ultimate prediction grounded in the acquired features from the preceding layers. Specifically, the fully connected layer accepts the flattened output derived from the pooling layer and subjects it to a set of weights and biases. The outcome from the fully connected layer then undergoes an activation function, such as SoftMax, to yield the ultimate prediction. This prediction manifests as class scores corresponding to each potential class. These scores facilitate the determination of the class label with the highest score, ultimately serving as the CNN's definitive prediction.

The fully connected layer within a Convolutional Neural Network (CNN) accepts the flattened output from the preceding layer, typically stemming from a pooling layer. It subsequently applies a set of weights and biases to generate a collection of output activations. The mathematical expression for an individual activation within the fully connected layer can be formulated as indicated in **Equation 2.6**.

$$a = \sigma(W * x + b) \tag{2.6}$$

x denotes the input activations originating from the preceding layer, typically in the form of a flattened vector.
W stands for the weight matrix, serving as the intermediary that maps the input activations to the resultant output activations.

b symbolizes the bias vector, which is incorporated into the weighted input
σ signifies the activation function employed upon the weighted input, encompassing options like the ReLU or SoftMax function.
a represents the resulting activation yielded by the fully connected layer.

Here's an example to help understand the fully connected layer, suppose we have an input activations vector x = [x1, x2, x3] and a weight matrix W = [[w1, w2, w3], [w4, w5, w6], [w7, w8, w9]]. The bias vector b = [b1, b2, b3].

To calculate the output activations of the fully connected layer, we first need to calculate the weighted input as in **Equation 2.7**.

$$z = W * x + b = [[w1, w2, w3], [w4, w5, w6], [w7, w8, w9]] * [x1, x2, x3] + [b1, b2, b3] \quad (2.7)$$

Following that, we employ the activation function σ to the weighted input, yielding the output activations as in **Equation 2.8**.

$$a = \sigma(z) = [\sigma(z1), \sigma(z2), \sigma(z3)] \quad (2.8)$$

In this way, the fully connected layer uses the weights and biases to transform the input activations into a set of output activations that are then used for making predictions.

Key characteristics of fully connected layers include:

i. Neuron Connections: Each neuron within a fully connected layer establishes connections with every neuron in the preceding layer, and these connections possess associated weights. These weights undergo adaptation during the training phase, dictating the connection strengths among neurons.

ii. Feature Combination: Fully connected layers amalgamate the features garnered from the preceding layers to formulate more advanced predictions or classifications. Each neuron within the fully connected layer possesses the capacity to apprehend intricate relationships amid the input features.

iii. Output Units: The quantity of neurons contained within the fully connected layer dictates the number of generated output units. In classification tasks, these output units frequently correspond to distinct classes, whereas in regression tasks, they represent continuous values.

iv. Activation Function: Each neuron situated in the fully connected layer subjects the weighted summation of its inputs to an activation function. Commonly employed activation functions encompass ReLU, sigmoid, and tanh. The activation function introduces non-linearity into the network, endowing it with the capability to grasp intricate patterns.

v. Bias Term: Fully connected layers typically include a bias term for each neuron, which allows the network to learn offsets or shifts in the predictions.

vi. Flattening: Within CNNs, fully connected layers are typically preceded by convolutional and pooling layers. To facilitate the connection between these layers, the output from the last pooling layer undergoes flattening, transforming it into a 1D vector prior to entering the fully connected layer.

Fully connected layers assume the responsibility of delivering ultimate predictions or classifications based on the gleaned features. They excel at capturing intricate relationships among features, furnishing the network with the capacity to derive generalizations from the acquired patterns. The results generated by the fully connected layer find utility in tasks such as image classification, object recognition, sentiment analysis, and various others. Nevertheless, within deep neural networks, particularly those featuring a multitude of parameters, fully connected layers can potentially contribute to overfitting. Consequently, regularization techniques like dropout and L2 regularization are frequently employed to mitigate overfitting and enhance generalization performance

2.2.5 SoftMax Function

The SoftMax function serves as a commonly employed activation function, particularly in the output layer of a neural network, specifically designed for multi-class classification challenges. When provided with a vector of real numbers as input, the SoftMax function yields a vector of probabilities, which collectively sum up to 1. Its primary function is to transform the network's output into a probability distribution spanning multiple classes, thereby signifying the likelihood of each class being the correct solution. The SoftMax function is defined as in **Equation 2.9**.

S(y_i) = (e^(y_i)) / sum(e^(y_j)) for all j

The SoftMax function serves as a commonly employed activation function, particularly in the output layer of a neural network, specifically designed for multi-class classification challenges. When provided with a vector of real numbers as input, the SoftMax function yields a vector of probabilities, which collectively sum up to 1. Its primary function is to transform the network's output into a probability distribution spanning multiple classes, thereby signifying the likelihood of each class being the correct solution. Here's a simple example to help understand the SoftMax function:

Suppose you have a neural network that classifies images of MALIMG into three categories: Adialer.C, Agent.FYI, and C2LOP.P. The output layer of this network contains three neurons, one for each class. The output of the network for a given image might look like y = [2, 1, 0]. These values represent the confidence of the network that the image belongs to each class. However, these values are not probabilities and may not sum to 1. The Softmax function can be used to convert these values into probabilities as in **Equation 2.10**.

S(y_1) = e^(2) / (e^(2) + e^(1) + e^(0)) = 0.74
S(y_2) = e^(1) / (e^(2) + e^(1) + e^(0)) = 0.24 (2.10)
S(y_3) = e^(0) / (e^(2) + e^(1) + e^(0)) = 0.02

Now, the sum of these values equals 1, effectively indicating a probability distribution across the classes. The highest value within this distribution, which is 0.74 in this instance, signifies the network's predicted class, identified as Adialer.C in this particular scenario.

Prominent characteristics of the softmax function include:

i. Probability Distribution: The softmax function transforms logits into a legitimate probability distribution encompassing multiple classes. Each value within the resulting vector denotes the likelihood of the input belonging to the corresponding class.

ii. Normalization: This function normalizes the logits by exponentiating them and subsequently dividing the results by the sum of the exponentiated values. This normalization procedure ensures that the resultant output probabilities always sum up to 1.

iii. Amplifying Differences: The softmax function amplifies the differences between logits. Logits with larger values will have larger exponentiated values, resulting in higher probabilities, while logits with smaller values will have lower probabilities.

iv. Numerical Stability: The exponential function can lead to numerical instability when dealing with very large or very small values. Techniques like subtracting the maximum value from the logits before applying softmax (known as the log-sum-exp trick) help mitigate this issue.

v. Multiclass Classification: The utilization of SoftMax in the output layer of a neural network is a common practice for tasks involving multi-class classification. Ordinarily, the class associated with the highest SoftMax probability is designated as the predicted class.

The SoftMax function assumes a pivotal role in both the training and deployment of neural networks for classification purposes. Throughout the training process, it collaborates with a loss function, such as cross-entropy, to calculate the loss and facilitate the adjustment of the network's weights via backpropagation. In the inference phase, the output produced by the SoftMax function furnishes the predicted class probabilities for the given input data.

2.3 Performance Metrics

We conducted Performance Metrics analysis using four criteria:

i. Accuracy.
ii. Sensitivity.
iii. Specificity.
iv. F-score.

2.3.1 Accuracy

Accuracy is a frequently employed metric for assessing the effectiveness of a classifier. In the context of Convolutional Neural Networks (CNNs), accuracy can be calculated as in **Equation 2.11**.

$$\text{Accuracy} = (TN + TP) / (TN + TP + FN + FP) \tag{2.11}$$

TN (True Negative) represents the count of instances correctly classified as negative. TP (True Positive) signifies the count of instances correctly classified as positive. FN (False Negative) represents the count of instances incorrectly classified as negative. FP (False Positive) represents the count of instances incorrectly classified as positive.

As an illustration, let's consider a scenario in which a CNN is trained to categorize a set of malware images. Among 1000 images, the CNN accurately identifies 900 as malware (TP) and

100 as non-malware (TN). However, it also incorrectly classifies 50 images as not being malware (FN) and 50 images as being malware (FP). In this case, the accuracy of the CNN as in **Equation 2.12**.

$$\text{Accuracy} = (100 + 900) / (100 + 900 + 50 + 50) = 1000 / 1200 = 83.33\% \qquad (2.12)$$

The accuracy metric provides a holistic view of the model's performance, but its comprehensiveness may vary in different scenarios. To assess the performance of a CNN more thoroughly, other metrics like precision, recall, F1-score, among others, can be employed.

2.3.2 Sensitivity (True Positive Rate or Recall)

Sensitivity, also referred to as True Positive Rate or Recall, stands as a critical performance metric in machine learning classification tasks. It gauges the model's proficiency in correctly identifying positive instances among all the actual positive instances present in the dataset. In essence, it assesses how well the model avoids false negatives. Mathematically, Sensitivity can be expressed as in **Equation 2.13**.

$$\text{Sensitivity} = (\text{True Positives}) / (\text{True Positives} + \text{False Negatives}) \qquad (2.13)$$

True Positives (TP) denote instances correctly classified as positive by the model, while False Negatives (FN) represent instances that are truly positive but are inaccurately categorized as negative by the model.

Sensitivity is particularly important in situations where the consequences of missing positive instances are significant. For example, in medical diagnosis, a false negative might mean failing to identify a disease in a patient who actually has it, which could have serious consequences.

A heightened sensitivity signifies that the model excels at capturing positive instances and mitigating false negatives, while a diminished sensitivity implies that the model overlooks a substantial number of positive instances.

2.3.3 Specificity (True Negative Rate)

Specificity, alternatively termed the True Negative Rate, constitutes another pivotal performance metric within machine learning classification tasks. This metric assesses the model's capability to accurately recognize negative instances among all the genuine negative instances within the dataset. In essence, it evaluates how adeptly the model avoids false positives. Mathematically, Specificity can be expressed as in **Equation 2.14**.

$$\text{Specificity} = (\text{True Negatives}) / (\text{True Negatives} + \text{False Positives}) \tag{2.14}$$

True Negatives (TN) denote instances correctly classified as negative by the model, while False Positives (FP) represent instances that are genuinely negative but are erroneously categorized as positive by the model.

Specificity is especially relevant in situations where correctly identifying negative instances is crucial. For example, in security systems, a false positive could lead to unnecessary alarms and disruptions. Elevated specificity signifies that the model excels at accurately identifying negative instances and mitigating false positives. Conversely, reduced specificity implies that the model is erroneously classifying a noteworthy number of negative instances as positive.

2.3.4 F-score (F1-Score)

The F-score, particularly the F1-Score, serves as a singular metric that amalgamates both precision and recall (or sensitivity) to furnish a well-rounded evaluation of a model's performance in binary classification tasks. It proves especially valuable when faced with class imbalance or the necessity to account for both false positives and false negatives.

The F1-Score represents the harmonic mean of precision and recall, as in **Equation 2.15**.

$$\text{F1-Score} = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall}) \tag{2.15}$$

Precision signifies the proportion of correct positive predictions relative to the total number of positive predictions generated by the model. It gauges the model's aptitude for avoiding false positives. Recall (Sensitivity) embodies the ratio of correct positive predictions to the complete count of actual positive instances within the dataset. It assesses the model's proficiency in averting false negatives. The F1-Score strikes an equilibrium between precision and recall. It considers both false positives and false negatives and proves particularly advantageous when seeking a compromise between these two forms of errors. A high F1-Score signals an effective equilibrium between precision and recall, while a lower F1-Score suggests an inequity between these two metrics.

2.4 Conclusion

The subsequent statements encapsulate the findings derived from the literature review.

Table 2.1: Conclusive literature review.

Serial	Title	Year	Dataset	Result	Remark
1	Robust intelligent malware detection using deep learning. [57]	2019	MALIMG	96%	Initial accuracy
			MALEVIS	86.29%	
2	Convolutional neural networks for malware classification. [60]	2016	MALIMG	95.33%	Improved accuracy
			MALEVIS	90.59%	
3	How to make attention mechanisms more practical in malware classification. [62]	2019	MALEVIS	91.31%	Improved accuracy for MALEVIS.
4	Detection of malicious code variants based on deep learning. [59]	2018	MALIMG	94.5%	Variation in accuracy
			MALEVIS	92.13%	
5	Binary malware image classification using machine learning with local binary pattern. [58]	2017	MALIMG	93.72%	Variation in accuracy
			MALEVIS	92.24%	
6	Malware classification using image representation. [61]	2019	MALIMG	96.08%	Variation in accuracy
			MALEVIS	93%	

7	A New Malware Classification Framework Based on Deep Learning Algorithms. [63]	2021	MALIMG	97.78%	Most appropriate
			MALEVIS	96.6%	

This chapter introduced the concept of deep learning, along with the associated notions of neural networks and Convolutional Neural Networks (CNNs). Particular attention was devoted to the typical layers employed in CNN implementations. Finally, the chapter delved into the metrics employed for the assessment of CNN models.

CHAPTER 3

METHODOLOGY

3.1 Visualizing malware as images

The process of converting malware code into images initiates by reading the binary file as an array of 8-bit unsigned integers. These binary values are subsequently transformed into decimal values, which are stored in a new decimal vector representing the malware sample. Finally, the resulting decimal vector undergoes reshaping into a 2D matrix, which is then visualized as a grayscale image. The dimensions of this 2D matrix, i.e., the spatial resolution of the image, are contingent on the size of the malware binary file. Additionally, this representation can detect zero-padding, which is employed for block alignment and to reduce executable entropy [39].

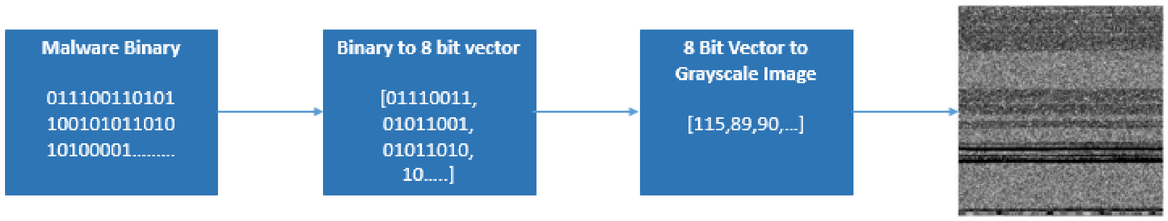


Figure 3.1: Binary codes to grayscale images.

Figure 3.1 illustrates the process of converting malware binary files into grayscale images. Initially, the binary file is read as a vector comprising 8-bit unsigned integers, and each element is converted into its decimal equivalent (e.g., [00000000] in binary corresponds to [0] in decimal, and [11111111] corresponds to [255]). These decimal values are then stored in a new vector representing the malware sample. Subsequently, this vector is reshaped into a 2D matrix, which is visualized as a grayscale image. The dimensions of the 2D matrix, or image resolution, are primarily determined by the binary file's size, following the resolution approach used by Nataraj et al. [16]. It has been observed that malware variants within the same family often exhibit similar textures (appearances), as demonstrated in Figure 3.2.

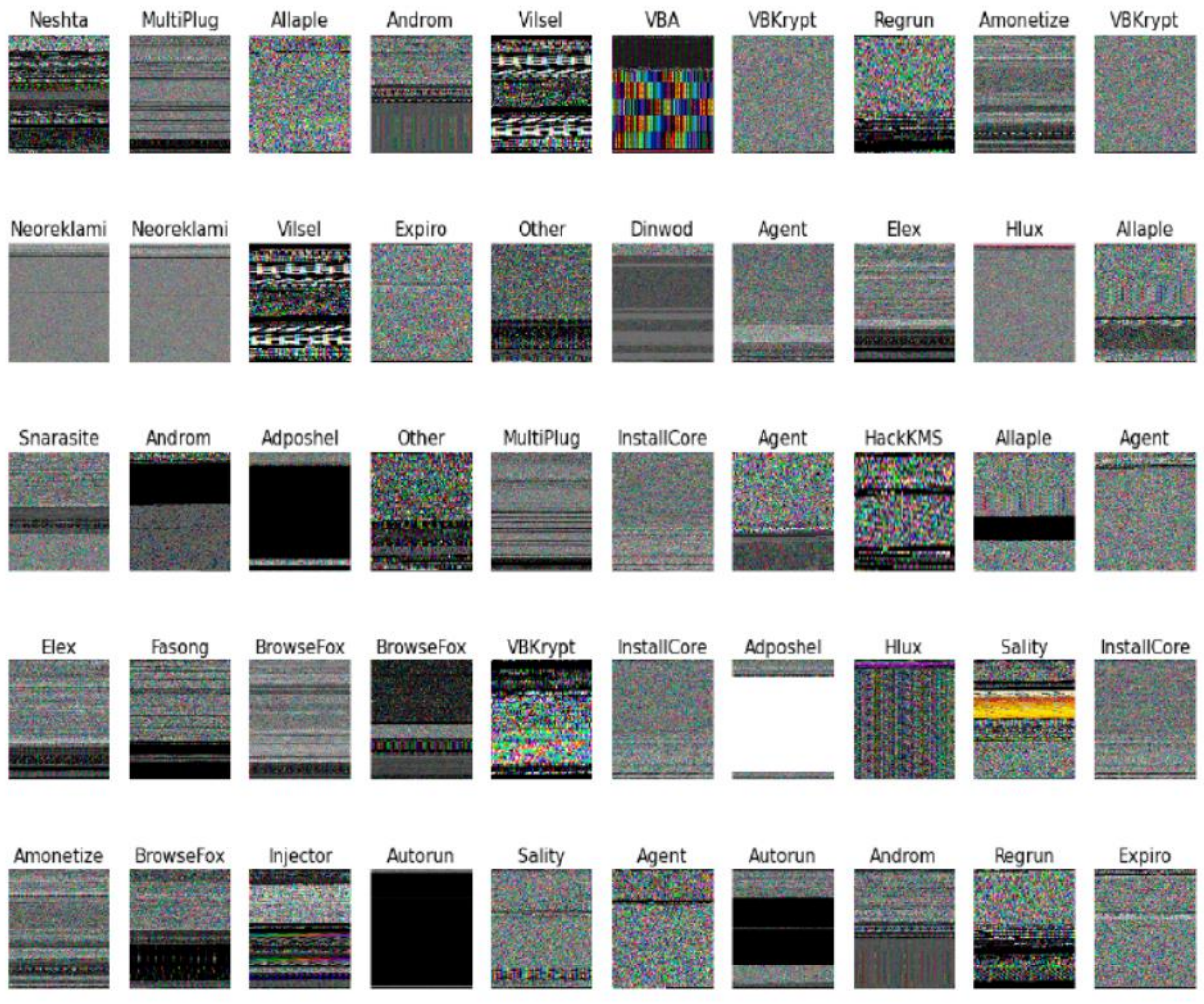


Figure 3.2: MALEVIS grayscale images.

Visualizing malware as images can be especially useful for large datasets where manual analysis may not be feasible, and can also help in reducing the dimensionality of the data for easier analysis. Nevertheless, it's crucial to emphasize that this approach does not replace conventional malware analysis methods. The accuracy of visual analysis should be considered in conjunction with other techniques to attain a comprehensive comprehension of malware and its behaviors.

3.2 Proposed Framework

The presented model presents an efficient structure for classifying malware by employing a hybrid deep neural network architecture. This framework comprises four main stages, as illustrated in Figure 1: data collection, the design of the neural network architecture, training, and evaluation. Moreover, Figure 2's system flowchart offers a more detailed breakdown of each of these stages.

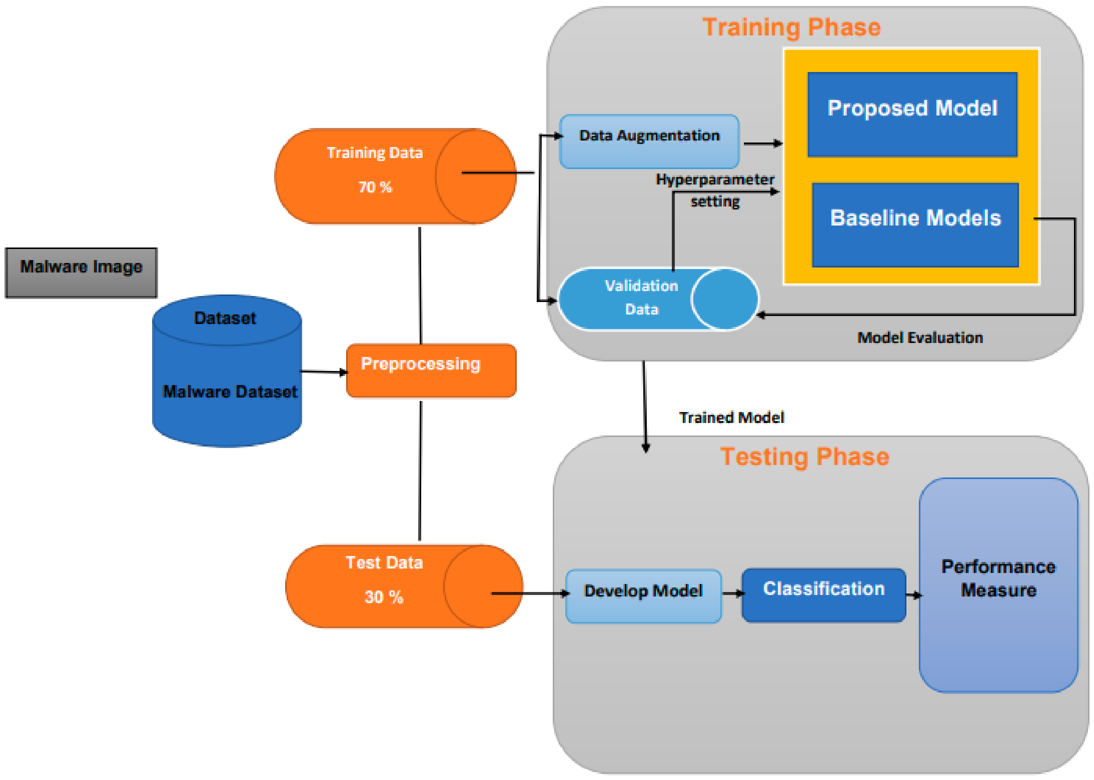


Figure 3.3: Proposed methodology for malware classification.

In Figure 3.4, the pre-training section showcases networks that have already undergone training and are currently being utilized as feature extractors. On the other hand, in the training section, the first three layers represent fully connected layers employed for the learning process. Lastly, the final layer signifies a SoftMax classifier, responsible for the classification process.

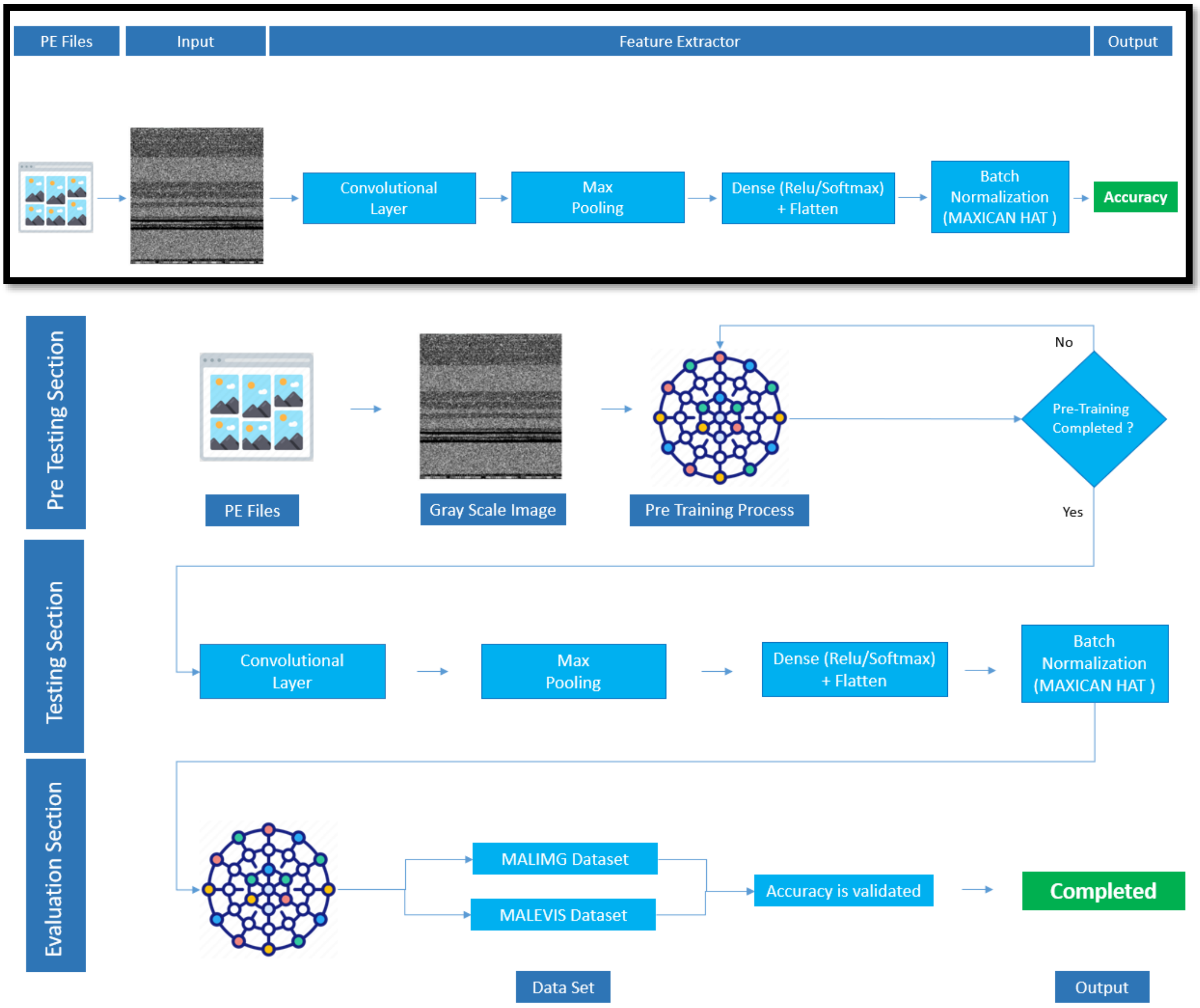


Figure 3.4: Flowchart of DL architecture for malware classification.

To commence, malware data is collected from various sources, including MALIMG [16] and MALEVIS [41]. Detailed explanations of these malware classification datasets will be provided in a subsequent section. Following data collection, the proposed deep neural network architecture is constructed.

At this stage, two pre-processing steps are carried out. Initially, an appropriate deep learning (DL) architecture is selected for performing malware classification tasks. This decision is based on preliminary experiments that have demonstrated the potential advantages of a hybrid module [42] in enhancing overall accuracy. Subsequently, transfer learning is explored to address specific challenges encountered during the classification process, such as time constraints and the large dataset size. In the context of transfer learning, the feature extraction step is initiated using pre-trained networks. Finally, a general classifier, such as a support vector machine or SoftMax, is employed for the classification task. This approach has been tailored to the proposed architecture as an effective means of overcoming the aforementioned challenges.

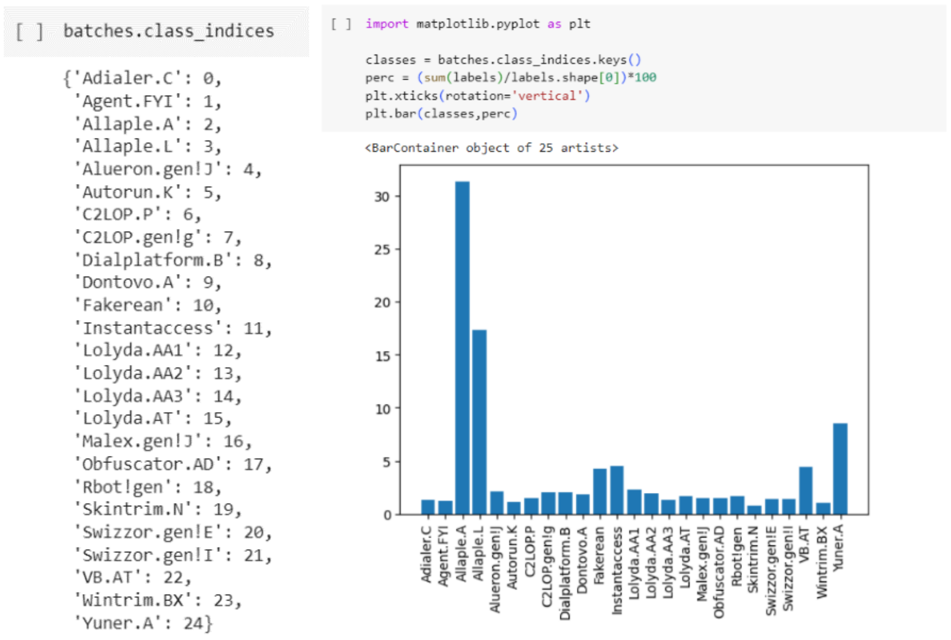
3.3 Data Sets

A dataset refers to a well-organized compilation of data utilized for the training, validation, or testing of machine learning models. Typically, datasets comprise a collection of instances or observations, wherein each instance encompasses several features or attributes. Additionally, these instances often include a target or label that the model endeavors to predict or classify. In our analysis, we employ the following two datasets:

i. MAILMG Dataset.
ii. MALEVIS Dataset.

3.3.1 MALIMG dataset

"MALIMG" is probably a shorthand for the "Malicious Image" dataset, which serves as a dataset for assessing the efficacy of image-based malware detection models. The MALIMG dataset encompasses a variety of malicious and benign image files, including icons and executables, frequently encountered within Windows operating systems. The objective is to train machine learning models capable of effectively discerning benign from malicious images by considering their visual characteristics and additional attributes [52], [53].



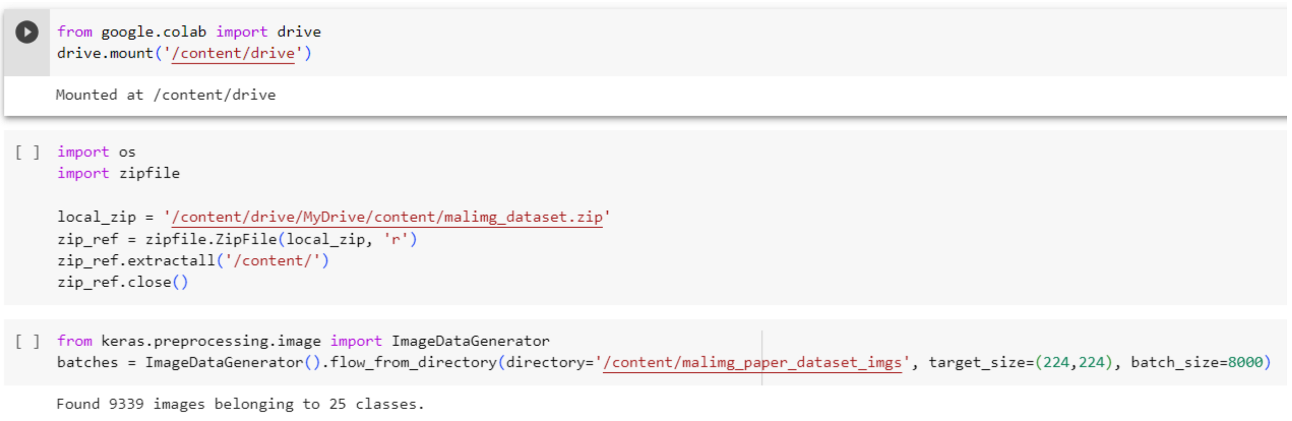
```
[ ]  batches.class_indices

{'Adialer.C': 0,
 'Agent.FYI': 1,
 'Allaple.A': 2,
 'Allaple.L': 3,
 'Alueron.gen!J': 4,
 'Autorun.K': 5,
 'C2LOP.P': 6,
 'C2LOP.gen!g': 7,
 'Dialplatform.B': 8,
 'Dontovo.A': 9,
 'Fakerean': 10,
 'Instantaccess': 11,
 'Lolyda.AA1': 12,
 'Lolyda.AA2': 13,
 'Lolyda.AA3': 14,
 'Lolyda.AT': 15,
 'Malex.gen!J': 16,
 'Obfuscator.AD': 17,
 'Rbot!gen': 18,
 'Skintrim.N': 19,
 'Swizzor.gen!E': 20,
 'Swizzor.gen!I': 21,
 'VB.AT': 22,
 'Wintrim.BX': 23,
 'Yuner.A': 24}
```

```
[ ]  import matplotlib.pyplot as plt

     classes = batches.class_indices.keys()
     perc = (sum(labels)/labels.shape[0])*100
     plt.xticks(rotation='vertical')
     plt.bar(classes,perc)

     <BarContainer object of 25 artists>
```

Figure 3.5: Names and percentage distribution of the MALIMG dataset.

The MALIMG datasets serve as a means to assess the resilience and adaptability of various machine learning models, including deep learning models, when it comes to detecting malware in image formats. These datasets establish a standard for appraising the performance of diverse algorithms, aiding researchers in pinpointing areas for enhancing malware detection [16].

The MALIMG dataset encompasses a total of 9,339 malware samples, divided into 25 distinct classes, each represented as grayscale images. These samples are distributed among different malware families or classes, each with varying quantities of samples. In experimental setups, 90% of the malware samples within each family are randomly selected for training purposes, while the remaining 10% are reserved for testing. This results in a training set comprising 8,394 samples and a testing set containing 945 samples [54].



```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

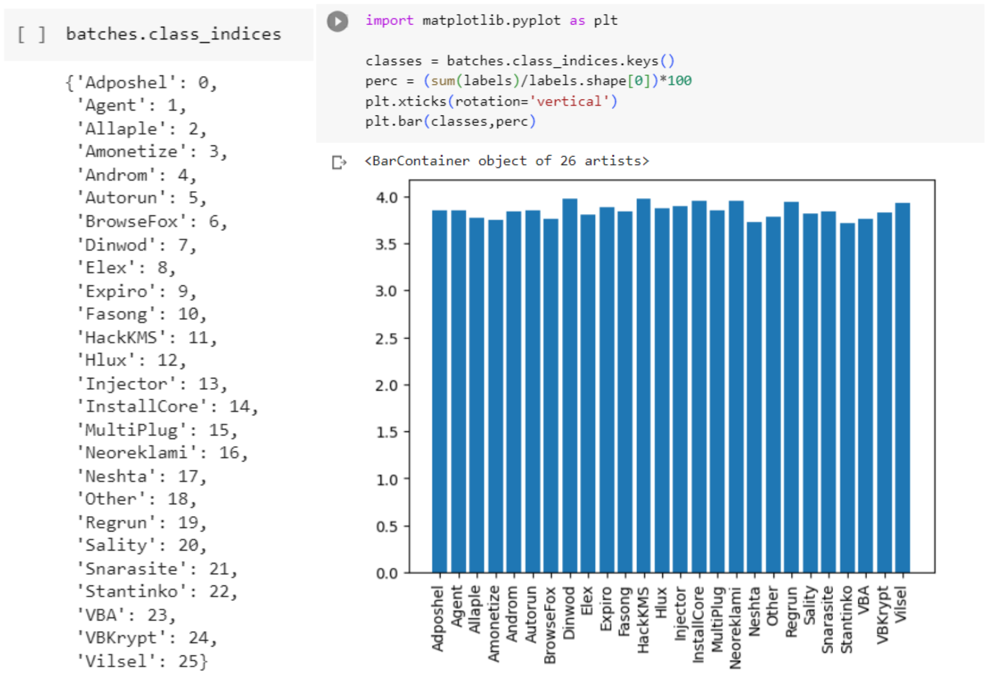
```
[ ] import os
    import zipfile

    local_zip = '/content/drive/MyDrive/content/malimg_dataset.zip'
    zip_ref = zipfile.ZipFile(local_zip, 'r')
    zip_ref.extractall('/content/')
    zip_ref.close()
```

```
[ ] from keras.preprocessing.image import ImageDataGenerator
    batches = ImageDataGenerator().flow_from_directory(directory='/content/malimg_paper_dataset_imgs', target_size=(224,224), batch_size=8000)

    Found 9339 images belonging to 25 classes.
```

Figure 3.6: Import the MALIMG dataset in Web IDE.

3.3.2 MALEVIS Dataset

The proposed method's performance evaluation was conducted using the MALEVIS (Malware Evaluation with Vision) dataset. This dataset comprises 14,226 RGB byte images categorized into 26 classes, encompassing 25 distinct malware categories and 1 benign software class. The dataset was partitioned into 9,100 samples for training and 5,126 samples for testing, with an even distribution of 350 images per class [44]. The malware classes included in the dataset encompass Adposhel, Agent-fyi, Allaple.A, Amonetize, Androm, AutoRun-PU, BrowseFox, Dinwod! rfn, Elex, Expiro-H, Fasong, HackKMS.A, Hlux!IK, Injector, InstallCore.C, MultiPlug, Neorekla-mi, Neshta, Regrun.A, Sality, Snarasite.D!tr, Stantinko, VBA/Hilium.A, VBKrypt, and Vilsel [45].

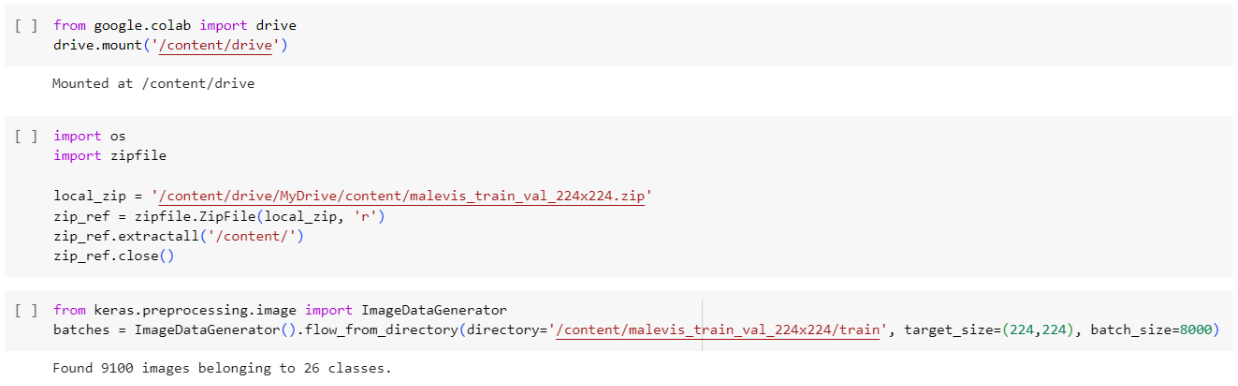


```
[ ]  batches.class_indices

{'Adposhel': 0,
 'Agent': 1,
 'Allaple': 2,
 'Amonetize': 3,
 'Androm': 4,
 'Autorun': 5,
 'BrowseFox': 6,
 'Dinwod': 7,
 'Elex': 8,
 'Expiro': 9,
 'Fasong': 10,
 'HackKMS': 11,
 'Hlux': 12,
 'Injector': 13,
 'InstallCore': 14,
 'MultiPlug': 15,
 'Neoreklami': 16,
 'Neshta': 17,
 'Other': 18,
 'Regrun': 19,
 'Sality': 20,
 'Snarasite': 21,
 'Stantinko': 22,
 'VBA': 23,
 'VBKrypt': 24,
 'Vilsel': 25}
```

```
import matplotlib.pyplot as plt

classes = batches.class_indices.keys()
perc = (sum(labels)/labels.shape[0])*100
plt.xticks(rotation='vertical')
plt.bar(classes,perc)

<BarContainer object of 26 artists>
```

Figure 3.7: Names and percentage distribution of the MALEVIS dataset.

To assess the performance of the proposed method, a dataset of byte images was generated, consisting of 26 distinct classes, comprising 25 malware categories and one legitimate class. The dataset creation process involved extracting binary images from malware files obtained from Comodo Inc. These binary images were then converted into 3-channel RGB format using the bin2png script developed by Sultanik. Subsequently, the vertically elongated images were resized to two different square resolutions: 224x224 pixels and 300x300 pixels [44].



```
[ ]  from google.colab import drive
     drive.mount('/content/drive')

     Mounted at /content/drive
```

```
[ ]  import os
     import zipfile

     local_zip = '/content/drive/MyDrive/content/malevis_train_val_224x224.zip'
     zip_ref = zipfile.ZipFile(local_zip, 'r')
     zip_ref.extractall('/content/')
     zip_ref.close()
```

```
[ ]  from keras.preprocessing.image import ImageDataGenerator
     batches = ImageDataGenerator().flow_from_directory(directory='/content/malevis_train_val_224x224/train', target_size=(224,224), batch_size=8000)

     Found 9100 images belonging to 26 classes.
```

Figure 3.8: Import the MALEVIS dataset in Web IDE.

CHAPTER 4

IMPLEMENTATION AND EVALUATIONS

4.1 Overview

This section offers a detailed overview of the implementation, experimental accuracy, and assessment of the proposed deep neural network model. In our implementation, we opted to utilize the Google Colab notebooks platform, which provides substantial virtual RAM, storage capacity, and GPU acceleration. The experience of working with Google Colab notebooks closely resembles that of working with the popular Kaggle Notebooks platform.

We decided to use the TensorFlow and Keras API for our implementation due to its ease of use and availability of extensive documentation and support from Google. Keras is an open-source neural network library that serves as an interface for TensorFlow, which is a free and open-source machine learning library primarily designed for deep neural network training and inference.

The process started by uploading the MALIMG and MALEVIS dataset zip files to Google Drive. This allowed us to access the data from the zipfiles in the notebook. Next, the Keras preprocessing library was used to extract the images and labels automatically. The datasets were divided into training and testing sets using the scikit-learn package. The training set was employed to train the neural network, adjusting its weights and biases. On the other hand, the testing set was utilized to assess the CNN's accuracy. Following the training and testing phases, accuracy and confusion matrices were visualized using Python's Matplotlib and Seaborn packages. The initial learning rate was set at 0.001, with a reduction by a factor of 10 after every 20 epochs. Weight decay and momentum were configured to 0.0005 and 0.9, respectively. The parameters within the

M-CNN network were initialized using the VGG-16 [56] weights. Random shuffling of the training data was performed in each epoch. Model evaluation was based on accuracy, representing the proportion of correctly labeled malware samples in the test data.

4.1.1 TensorFlow

TensorFlow is a robust open-source software library tailored for large-scale numerical computations, particularly in the realm of machine learning. In the context of malware data analysis, TensorFlow serves as a valuable tool for constructing machine learning models capable of analyzing malware samples and discerning between malicious and non-malicious files. This is accomplished by developing deep neural networks, which undergo training on a dataset comprising both malware and non-malware samples. TensorFlow provides a flexible platform for training, validating, and deploying these models, allowing researchers and practitioners to perform sophisticated malware analysis and classification.

Key features of TensorFlow include:

i. Graph Computation: TensorFlow employs a dataflow graph as a means to depict mathematical computations. This approach empowers users to delineate intricate computational processes and models using a symbolic representation that operates at a higher level.

ii. Automatic Differentiation: TensorFlow includes an automatic differentiation feature, which is crucial for training neural networks using gradient descent optimization algorithms.

iii. Flexible Architecture: It supports both CPUs and GPUs for efficient numerical computations, which are crucial for training large neural networks.

iv. Abstraction Levels: TensorFlow offers different levels of abstraction, from high-level APIs like Keras (a neural network API) to lower-level APIs that allow users to have more control over model design.

v. Wide Range of Libraries: TensorFlow provides various libraries and tools for building different types of machine learning models, including image and text analysis, reinforcement learning, and more.

vi. Model Deployment: TensorFlow models can be deployed on various platforms, including cloud services, mobile devices, browsers, and embedded systems.

vii. Community and Ecosystem: TensorFlow boasts a thriving and engaged community actively involved in its advancement. This vibrant community provides an abundance of resources, tutorials, and pre-trained models, simplifying the initiation of diverse machine-learning endeavors.

viii. TensorBoard: TensorFlow includes a visualization tool called TensorBoard that helps users visualize and monitor training progress, model architectures, and other relevant information.

ix. Support for Various Neural Network Architectures: TensorFlow supports building various neural network architectures, from simple feedforward networks to more complex convolutional networks, recurrent networks, and more.

TensorFlow has made substantial contributions to the progression of machine learning and deep learning, enhancing accessibility and efficiency for researchers, developers, and data scientists in the creation and deployment of advanced machine learning models.

4.1.2 Keras

Keras is an open-source high-level software library that offers a Python interface for artificial neural networks. Its purpose is to streamline the construction and training of deep learning models by supplying a user-friendly and high-level API. Keras serves as a user-friendly interface for the underlying TensorFlow library, delivering a more intuitive and accessible means for users to create, train, and assess neural networks. Some notable characteristics of Keras encompass:

i. User-Friendly API: Keras offers a straightforward and comprehensible interface for creating and training neural networks. It enables users to specify models with just a few lines of code, ensuring accessibility for individuals ranging from newcomers to seasoned experts in deep learning.

ii. Modularity: Keras emphasizes modularity, allowing users to build complex neural network architectures by combining pre-defined building blocks called "layers." These layers can be stacked and interconnected to create different architectures.

iii. Flexibility: While Keras provides high-level abstractions, it also offers a degree of flexibility that enables users to customize their models by specifying layer configurations, activation functions, optimization algorithms, loss functions, and more.

iv. Support for Various Backends: Initially, Keras supported multiple deep learning libraries as backends, including TensorFlow, Theano, and Microsoft Cognitive Toolkit (CNTK). However, starting with TensorFlow 2.0, Keras is tightly integrated with TensorFlow, making TensorFlow the default backend.

v. Multi-Layer Perceptrons (MLPs) and More: Keras offers extensive support for various neural network architectures, encompassing straightforward feedforward networks (multi-layer perceptrons), convolutional neural networks (CNNs), recurrent neural networks (RNNs), and amalgamations of these architectural styles.

vi. Model Visualization: Keras includes tools like TensorBoard integration that help users visualize their model architectures, training progress, and other metrics.

vii. Transfer Learning and Pre-Trained Models: Keras simplifies the process of transfer learning by providing access to pre-trained models through its applications module. This capability is especially valuable for harnessing models that have been trained on expansive datasets to address particular tasks.

viii. Community and Resources: Keras has a strong community of users and contributors, which has led to a wealth of tutorials, guides, and resources available for both beginners and experienced users.

To sum it up, Keras offers an accessible and flexible means of developing, training, and deploying deep learning models. Its emphasis on ease and modularity has made it a favored option among researchers, developers, and data scientists, enabling them to explore neural networks without delving into intricate low-level intricacies.

4.1.3 NumPy

NumPy, an abbreviation for Numerical Python, stands as a foundational library within the Python programming language, and it enjoys extensive application in machine learning and scientific computing. Its core functionality revolves around enabling the manipulation of large, multi-dimensional arrays and matrices, bolstered by an array of high-level mathematical operations tailored for efficient use with these arrays. NumPy serves as a pivotal cornerstone for numerous machine learning libraries and frameworks.

Here are some key aspects of NumPy in machine learning:

i. Multi-Dimensional Arrays: NumPy introduces the numpy.ndarray data type, which allows you to create arrays of various dimensions (e.g., 1D, 2D, 3D) to store and manipulate data efficiently. These arrays possess homogeneity, indicating that all their elements share the same data type.

ii. Efficient Element-Wise Operations: NumPy offers an extensive collection of mathematical and logical functions that can be applied to arrays in an element-wise manner. This makes it easy to perform operations like addition, subtraction, multiplication, and more on entire arrays at once, which is crucial for machine learning computations.

iii. Broadcasting: NumPy incorporates broadcasting, enabling operations on arrays with different shapes and sizes without requiring explicit looping, thus enhancing code simplicity and performance.

iv. Integration with Machine Learning Libraries: Many machine learning libraries and frameworks, including TensorFlow and PyTorch, are built on top of NumPy or have NumPy-like interfaces. This makes it easy to seamlessly integrate NumPy arrays into machine learning workflows.

v. Random Number Generation: NumPy provides a random number generation module (numpy.random) that is commonly used for tasks like data augmentation, initialization of model parameters, and generating random samples for statistical experiments.

vi. Linear Algebra Operations: NumPy encompasses an extensive array of linear algebra functions, including matrix multiplication, eigenvalue decomposition, and singular value decomposition, which are crucial components of numerous machine learning algorithms, particularly those rooted in linear algebra.

vii. Data Manipulation and Preprocessing: NumPy is instrumental in data preprocessing tasks, such as scaling, normalization, and reshaping of data, which are often required before feeding data into machine learning models.

viii. Performance Optimization: NumPy is implemented in C and Fortran, which makes it highly optimized for numerical computations. This results in efficient code execution, a crucial factor in machine learning applications, especially for large datasets and complex models.

In conclusion, NumPy stands as a foundational library in machine learning, offering indispensable tools for the efficient handling of numerical data. Its versatile array of operations and mathematical functions are essential for tasks such as data manipulation, model training, and evaluation within the realm of machine learning.

4.1.4 Seaborn

Seaborn, a Python data visualization library frequently employed in machine learning and data analysis, serves as an invaluable tool. Built upon Matplotlib, another widely-used data visualization library, Seaborn offers a more user-friendly interface for crafting informative and visually appealing statistical graphics. Its utility shines when expeditiously exploring data, visualizing variable relationships, and generating informative presentations and reports within the context of machine learning.

Seaborn boasts the following key attributes and applications in the realm of machine learning:

i. Statistical Visualization: Seaborn excels at creating statistical visualizations that help you understand the distribution of data, identify patterns, and visualize relationships between variables. It supplies functions for producing a diverse range of plots, encompassing histograms, scatter plots, bar plots, box plots, violin plots, and may more.

ii. Integration with Pandas: Seamlessly integrating with Pandas DataFrames, Seaborn simplifies the task of working with and visualizing data stored in Pandas data structures.

iii. Automatic Estimation and Aggregation: Seaborn often automatically computes and visualizes summary statistics (e.g., means, medians, confidence intervals) within plots, simplifying the process of exploring and understanding data.

iv. Color Palettes: Seaborn offers a variety of aesthetically pleasing color palettes, making it easy to create visually appealing visualizations. These palettes are particularly useful when distinguishing between categories or groups of data.

v. Facet Grids: Seaborn provides tools for creating facet grids, which allow you to create multiple plots or facets, each showing a subset of your data based on specific variables. This is useful for visualizing relationships within different subsets of your dataset.

vi. Regression Analysis: Seaborn includes functions for visualizing and exploring linear and non-linear relationships between variables, making it handy for regression analysis in machine learning.

vii. Categorical Data Visualization: Seaborn specializes in visualizing categorical data. It offers functions for creating count plots, bar plots, and categorical scatter plots that are useful for exploring relationships in categorical data.

viii. Time Series Visualization: Seaborn supports time series data visualization, making it useful for analyzing and plotting time-based datasets in machine learning tasks.

ix. Customization: While Seaborn offers sensible defaults for numerous plot styles, it also affords extensive customization options for plots, encompassing colors, styles, labels, and beyond.

In summary, Seaborn serves as a valuable asset within the machine learning workflow. It streamlines the process of visualizing and comprehending data, a pivotal step in data preprocessing, feature engineering, and model evaluation. It helps data scientists and machine learning practitioners create informative and visually appealing visualizations quickly and effectively.

4.1.5 Scikit-learn

Scikit-learn, commonly denoted as "sklearn," stands as a well-liked open-source machine learning library tailored for the Python programming language. Renowned for its versatility, it ranks among the most extensively employed libraries for an array of machine learning endeavors. Its applications span classification, regression, clustering, dimensionality reduction, model selection, and data preprocessing. This robust library is constructed atop other Python libraries like NumPy, SciPy, and matplotlib, furnishing a uniform and approachable API for seamless interaction with a multitude of machine learning algorithms.

Here are some key aspects of scikit-learn:

i. Scikit-learn encompasses an extensive repertoire of machine learning algorithms, encompassing both supervised and unsupervised learning paradigms. Among the array of algorithms at its disposal are Supervised learning algorithms, which comprise Support Vector Machines, Decision Trees, Random Forests, k-Nearest Neighbors, Linear and Logistic Regression, and more. In the realm of Unsupervised learning, Scikit-learn boasts algorithms such as Clustering (including K-Means and Hierarchical clustering) and Dimensionality Reduction (featuring PCA and t-SNE), along with Gaussian Mixture Models, and various others.

ii. Data Preprocessing: Scikit-learn provides a suite of data preprocessing tools, encompassing functions for tasks like scaling, normalization, imputation of missing values, feature selection, and feature extraction.

iii. Model Evaluation: The library supplies functions for assessing machine learning models through metrics such as accuracy, precision, recall, F1-score, ROC AUC, and others. It also supports techniques like cross-validation to ensure robust model assessment.

iv. Hyperparameter Tuning: Scikit-learn includes tools for hyperparameter tuning through methods like grid search and randomized search, making it easier to find the best hyperparameters for your models.

v. Pipeline: Scikit-learn allows you to build data processing and modeling pipelines, ensuring that data preprocessing and model training are handled in a structured and reproducible manner.

vi. Feature Engineering: The library offers various feature engineering techniques, such as one-hot encoding, label encoding, and text vectorization, to prepare data for machine learning models.

vii. Seamless Integration with Other Libraries: Scikit-learn smoothly integrates with other well-known Python libraries like NumPy, Pandas, and Matplotlib, facilitating a seamless workflow for data manipulation and visualization.

viii. Ease of Use: Scikit-learn is renowned for its straightforward and uniform API, ensuring accessibility for both novice and seasoned machine learning professionals.

ix. User Community and Documentation: Scikit-learn boasts a sizable and engaged user community, complemented by comprehensive documentation, tutorials, and practical examples to assist users in initiating their journey and grasping the library's potential.

Overall, scikit-learn is a versatile and essential tool in the machine learning ecosystem. It simplifies the implementation of various machine learning algorithms and provides a standardized interface for building, evaluating, and deploying machine learning models. Whether you are a novice or an expert in machine learning, scikit-learn is a valuable resource for your data science and machine learning projects.

4.1.6 Matplotlib.pyplot

Matplotlib.pyplot is a Python library that facilitates the generation of static, animated, and interactive visual representations across a diverse array of formats, including line plots, bar charts, scatter plots, histograms, and numerous others. It is part of the larger Matplotlib library, which is a comprehensive data visualization library for Python. pyplot is a subpackage within Matplotlib that provides a simple and convenient interface for creating and customizing plots.

Here are the key components and features of matplotlib.pyplot:

i. Plotting Functions: Pyplot offers an assortment of plotting functions that streamline the process of generating a wide array of plots, all achievable with just a few lines of code. For example, you can create line plots using plt.plot(), scatter plots with plt.scatter(), bar charts with plt.bar(), and histograms with plt.hist().

ii. Customization: You can customize every aspect of your plots, including titles, labels, axis limits, colors, markers, line styles, and more. Pyplot offers a range of functions, such as plt.title(), plt.xlabel(), plt.ylabel(), plt.legend(), plt.grid(), and more, to incorporate labels and annotations into your plots.

iii. Subplots: Using the plt.subplots() function, you can arrange multiple plots within the same figure, organizing them in a grid-like manner. This is useful for comparing different aspects of your data in a single figure.

iv. Save and Export: Matplotlib offers the flexibility to save your plots in multiple file formats, such as PNG, PDF, SVG, and others, using the plt.savefig() function.

v. Interactive Features: While pyplot is primarily used for static plots, you can combine it with other libraries like Jupyter Notebook and interactive widgets to create interactive visualizations for exploring data.

vi. Integration with NumPy: Matplotlib smoothly integrates with NumPy arrays, simplifying the process of plotting data stored within NumPy arrays.

vii. Support for LaTeX: You can use LaTeX for rendering text and mathematical expressions in titles, labels, and annotations, which is particularly useful for creating scientific plots.

4.1.7 Google Collaboratory

Google Collaboratory, commonly referred to as Colab, is a freely accessible Jupyter Notebook environment provided by Google. It allows users to write and execute code, edit documents, and run interactive web applications. Colab provides a cloud-based platform with free access to GPUs and TPUs, making it an ideal environment for conducting large-scale data experiments and machine learning projects. Additionally, Colab notebooks can be stored and shared on Google Drive, allowing users to collaborate on projects in real-time and share accuracy with others. Google Colab provides access to a virtual machine with a GPU and a CPU. The amount of RAM, GPU, and CPU cores provided by Colab varies depending on the type of runtime being used. Currently, the standard runtime provided by Colab offers 12 GB of RAM, an V100 GPU.

The high-RAM runtime provides 25 GB of RAM while the A100 GPU. Key features of Google Colab include:

i. Colab offers complimentary access to Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs), both of which are hardware accelerators that greatly enhance the training speed of deep learning models..

ii. Colab operates entirely in the cloud, eliminating concerns about the capabilities of your local machine and allowing you to write and execute code on powerful remote servers.

iii. Jupyter Notebook Integration: Colab uses the Jupyter Notebook interface, allowing you to write and run code in cells. This notebook-style interface makes it easy to write, execute, and visualize code step by step.

iv. Colab comes equipped with popular libraries for data science and machine learning, including TensorFlow, PyTorch, pandas, NumPy, and others, already installed. This eliminates the need for setup and configuration, saving you time and effort.

v. Easy Data Sharing: You can upload and store datasets directly in your Colab environment. Additionally, you can easily share your Colab notebooks with others for collaboration or educational purposes.

vi. Google Drive Integration: Colab notebooks are saved in your Google Drive, making it easy to organize and access your work from various devices.

vii. Code Snippets and Markdown: Colab supports Markdown cells, allowing you to document your code and experiments. You can also include images, hyperlinks, and formatted text to make your notebooks more informative.

viii. Interactive Visualizations: You can create interactive plots and visualizations using libraries like Matplotlib and Seaborn. These visualizations help you better understand your data and experiment results.

ix. Easy Setup and Environment Management: Since Colab is cloud-based, you don't need to worry about installing and managing libraries and dependencies. Everything is handled by Google's infrastructure.

x. Time-Limited Sessions: Colab sessions are time-limited, meaning that after a certain period of inactivity, your environment might be disconnected. However, you can always reconnect and continue your work.

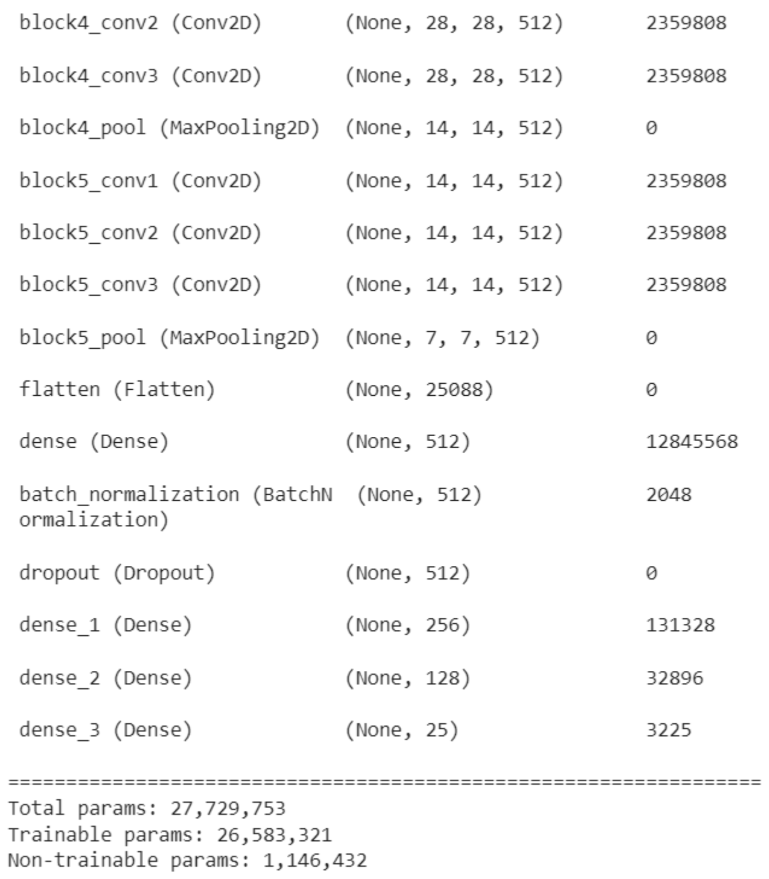
In summary, Google Colab is a powerful and user-friendly tool for writing, executing, and sharing Python code, especially for data science, machine learning, and research tasks. Its combination of cloud resources, pre-installed libraries, and collaboration features makes it an excellent choice for both beginners and experienced developers.

4.1.8 Our Proposed Model

We have reduced the complexity of our deep learning model by simplifying the architecture of our Convolutional Neural Network (CNN). Previously models consisted of many Convolution layers, Max-pooling layers, Dense layers, and SoftMax layers for classification. The previous architecture resulted in a high number of trainable weights and biases, this hinders the model's ability to effectively generalize to unfamiliar data.
To reduce the complexity, we have made the following changes, Here's the breakdown of the layers. VGG16 base layers (pre-trained). These layers are not trainable, except for the last few.

i. It consists a flatten layer.
ii. A Dense layer containing 512 units and utilizing the 'relu' activation function.
iii. Batch Normalization layer.
iv. Dropout layer with dropout rate 0.5.
v. A Dense layer consisting of 256 units and using the 'relu' activation.
vi. A Dense layer comprising 128 units with the 'relu' activation.
vii. The final Dense layer with 26 units in its output (assuming it's a classification task) and activation 'SoftMax'.

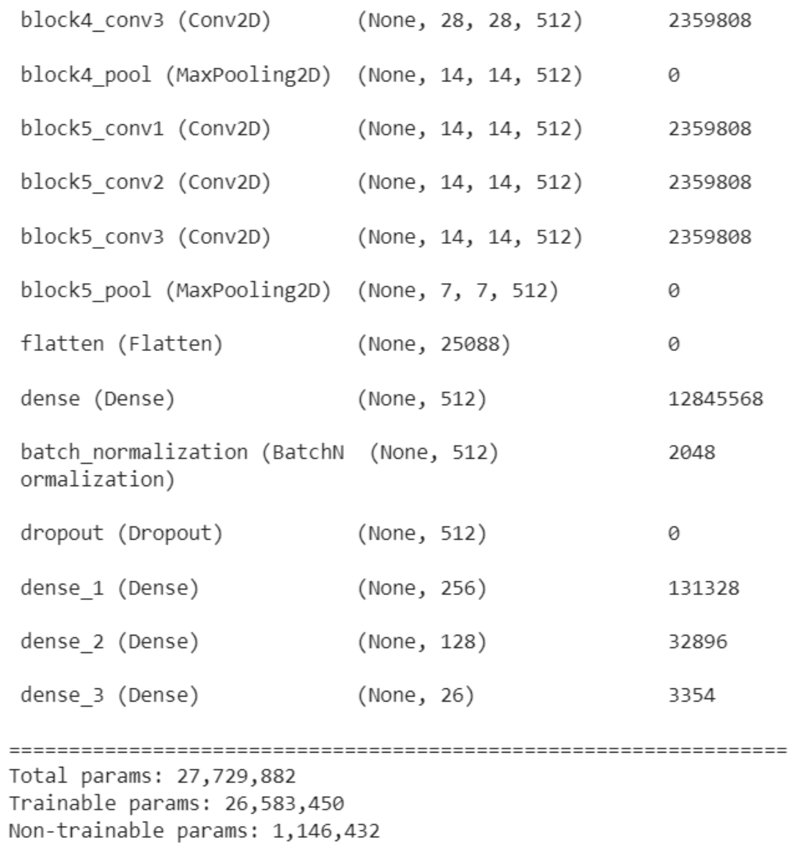
The cumulative count of layers in the model, encompassing both the foundational and additional layers, would be the sum of these layers. The count includes all types of layers (Conv2D, MaxPooling2D, Dense, etc.). The new architecture accuracy in a reduced number of trainable weights and biases, improving the generalization performance of the model. The following is a summary of the layers of the CNN when applied to the MALIMG dataset, along with the number of trainable weights and biases (params) in each layer:



```
 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 512)               12845568

 batch_normalization (BatchN  (None, 512)              2048
 ormalization)

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 256)               131328

 dense_2 (Dense)             (None, 128)               32896

 dense_3 (Dense)             (None, 25)                3225

=================================================================
Total params: 27,729,753
Trainable params: 26,583,321
Non-trainable params: 1,146,432
```

Figure 4.1: Number of layers trainable weights and biases (params) utilized for MALIMG dataset.

A breakdown of the CNN applied to the MALEVIS dataset is provided below, including the number of trainable weights and biases (params) in each layer:



```
 block4_conv3 (Conv2D)      (None, 28, 28, 512)      2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)     0

 block5_conv1 (Conv2D)      (None, 14, 14, 512)      2359808

 block5_conv2 (Conv2D)      (None, 14, 14, 512)      2359808

 block5_conv3 (Conv2D)      (None, 14, 14, 512)      2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)       0

 flatten (Flatten)          (None, 25088)            0

 dense (Dense)              (None, 512)              12845568

 batch_normalization (BatchN  (None, 512)            2048
 ormalization)

 dropout (Dropout)          (None, 512)              0

 dense_1 (Dense)            (None, 256)              131328

 dense_2 (Dense)            (None, 128)              32896

 dense_3 (Dense)            (None, 26)               3354

=================================================================
Total params: 27,729,882
Trainable params: 26,583,450
Non-trainable params: 1,146,432
```

Figure 4.2: Number of layers trainable weights and biases (params) utilized for MALEVIS dataset.

With these changes, our model has a reduced complexity, making it easier to train and improving its generalization performance on new data.

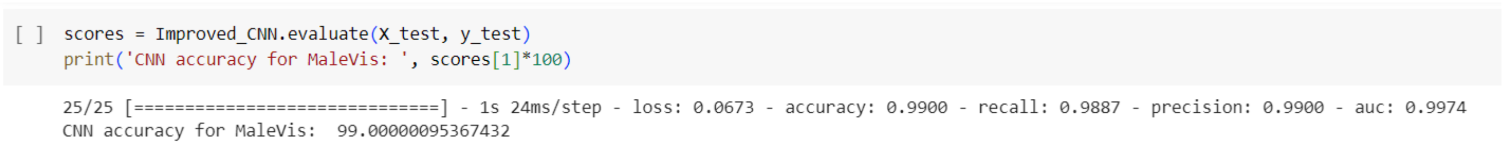
4.2 Accuracy

The table reveals that the CNN achieves an average validation accuracy of 99% on the MALIMG dataset, while on the MALEVIS dataset, it achieves a validation accuracy of 97.12%.

Table 4.1: Accuracy of MALIMG and MALEVIS datasets.

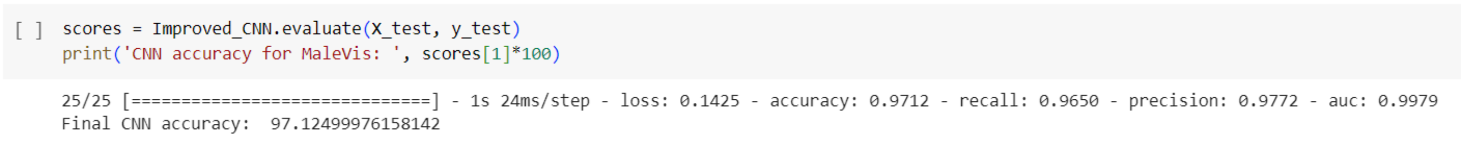
Dataset	Accuracy
MALIMG	99%
MALEVIS	97.12%

The following figures illustrate the evolution of accuracy for both the training and validation sets at each epoch in the most recent training run..



```
[ ] scores = Improved_CNN.evaluate(X_test, y_test)
    print('CNN accuracy for MaleVis: ', scores[1]*100)

    25/25 [==============================] - 1s 24ms/step - loss: 0.0673 - accuracy: 0.9900 - recall: 0.9887 - precision: 0.9900 - auc: 0.9974
    CNN accuracy for MaleVis:  99.00000095367432
```

Figure 4.3: Accuracy for MALIMG Dataset.



```
[ ] scores = Improved_CNN.evaluate(X_test, y_test)
    print('CNN accuracy for MaleVis: ', scores[1]*100)

    25/25 [==============================] - 1s 24ms/step - loss: 0.1425 - accuracy: 0.9712 - recall: 0.9650 - precision: 0.9772 - auc: 0.9979
    Final CNN accuracy:  97.12499976158142
```

Figure 4.4: Accuracy for MalVis Dataset.

4.2.1 MALIMG Dataset

Here's a breakdown of the code:

i. Importing the Required Library: The code begins by importing the matplotlib.pyplot module as plt. This module is a part of the Matplotlib library, which is commonly used for creating visualizations in Python.

ii. Extracting Training History: The variables acc, val_acc, loss, and val_loss are utilized to store the 1. training accuracy, 2. validation accuracy, 3. training loss, and 4. validation loss values, correspondingly. These values are likely obtained from the history object after training a machine learning model. The history object typically contains metrics recorded at each epoch during training.

iii. Defining Epochs: The epochs variable is defined as a range representing the number of epochs. This will serve as the label for the x-axis of the plot..

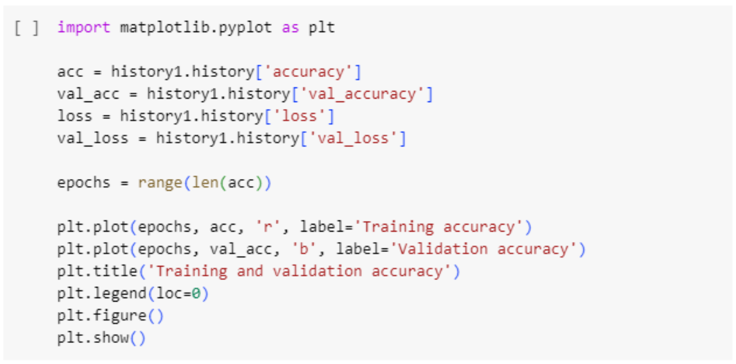
iv. Plotting Training Accuracy and the Validation Accuracy: The code uses plt.plot() to create two lines on the same plot. One line represents the training accuracy (acc) and is displayed in red ('r'). The other line represents the validation accuracy (val_acc) and is displayed in blue ('b').

v. Setting Title and Legend: The title of the plot is set using plt.title(). In this instance, it is configured as "Training and validation accuracy." The plt.legend() function introduces a legend to the plot, facilitating the differentiation between the training and validation accuracy lines. The loc=0 argument specifies that the legend should be placed in the "best" location on the plot.

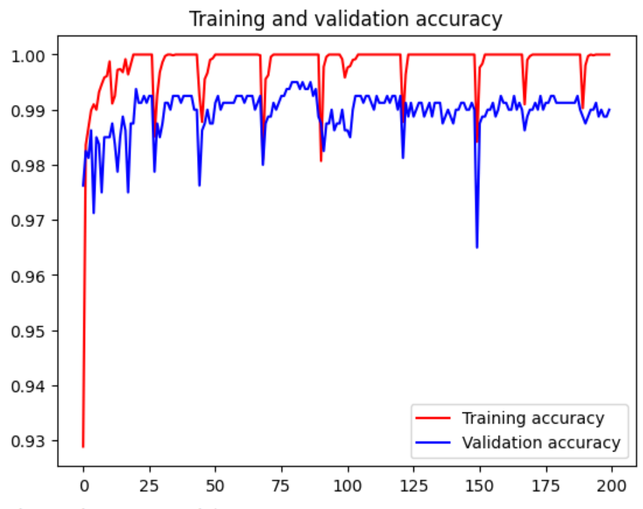
vi. Creating a New Figure: plt.figure() creates a new figure, which means that subsequent plots will be drawn on a new canvas. This is typically used when you want to create multiple separate plots in a single script or code block.

vii. Displaying the Plot: Finally, plt.show() is used to display the plot on the screen. This command is essential for rendering the visualizations you've created using Matplotlib.

In summary, this code generates a plot that shows the progression of training and validation accuracy over epochs. It aids in evaluating the performance of your model on both the training and validation datasets, and in identifying indications of either overfitting or generalization issues.



```
[ ]  import matplotlib.pyplot as plt

     acc = history1.history['accuracy']
     val_acc = history1.history['val_accuracy']
     loss = history1.history['loss']
     val_loss = history1.history['val_loss']

     epochs = range(len(acc))

     plt.plot(epochs, acc, 'r', label='Training accuracy')
     plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
     plt.title('Training and validation accuracy')
     plt.legend(loc=0)
     plt.figure()
     plt.show()
```



```
<Figure size 640x480 with 0 Axes>
```

Figure 4.5: Graphs depicting the growth of Training accuracy (in red) and Validation accuracy (in blue) as the number of epochs increases on the MALIMG dataset.

4.2.2 MALEVIS Dataset

The code snippet creates a depiction of a machine-learning model's training and validation accuracy over several epochs using the matplotlib library. Let's break down the code step by step:

i. Importing the Required Library: The code starts by importing the matplotlib.pyplot module as plt. This module is used for creating various types of plots and visualizations in Python.

ii. Extracting Training History Metrics: The variables acc, val_acc, loss, and val_loss is used to store specific metrics from the history1 object. Typically, metrics that were captured while a machine learning model was being trained can be found in the history1 object. Specifically:
   a. acc: Training accuracy values recorded over different epochs.
   b. val_acc: Validation accuracy values recorded over different epochs.
   c. loss: Training loss values recorded over different epochs.
   d. val_loss: Validation loss values recorded over different epochs.

iii. Defining Epochs: The variable epochs is created as a range that spans the length of the acc list. This range is used to represent total number of epochs plotted on x-axis.

iv. Plotting Training and Validation Accuracy: The code uses the plt.plot() function to create two lines on the same plot:
   a. The first line represents the training accuracy (acc) and is drawn in red ('r').
   b. The second line represents the validation accuracy (val_acc) and is drawn in blue ('b').

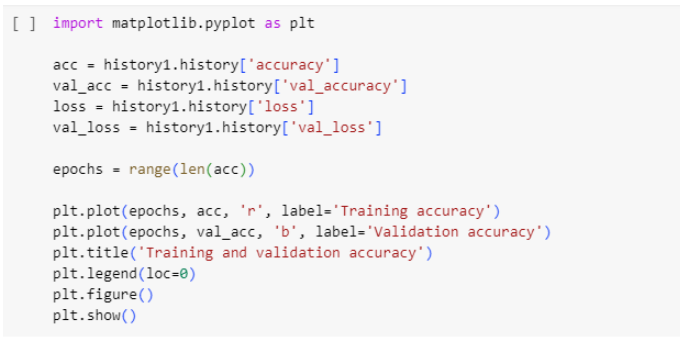
v. Setting Title and Legend:
   a. plt.title() sets the title of the plot to "Training and validation accuracy".

   b. plt.legend(loc=0) adds a legend to plot, indicating that which line correlate to training accuracy and which communicate with the validation accuracy. The loc=0 argument specifies that the legend should be placed in the "best" location on the plot.

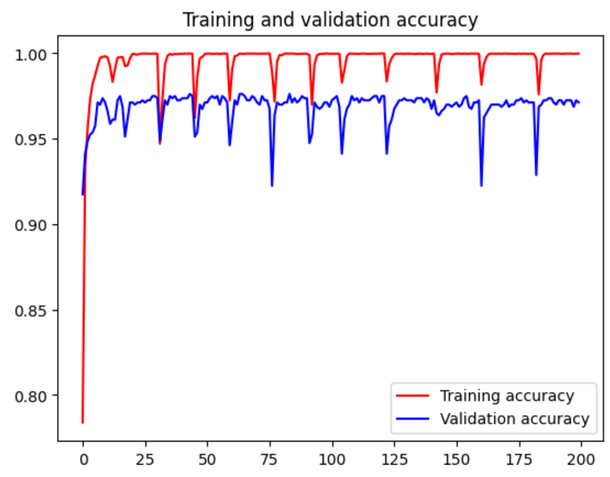
vi. Creating a New Figure: plt.figure() is used to create a new figure. This is useful if you want to create multiple separate plots in the same script or code block.

vii. Displaying the Plot: Finally, plt.show() is called to display the plot on the screen.

In summary, this code generates a line plot that visualizes, progression of training and validation accuracy across epochs. You may learn more about how effectively your model is learning from the training data and whether it is overfitting or generalizing to new data by comparing these two curves. The visualizations help you make informed decisions about model training and adjustments.



```
[ ] import matplotlib.pyplot as plt

    acc = history1.history['accuracy']
    val_acc = history1.history['val_accuracy']
    loss = history1.history['loss']
    val_loss = history1.history['val_loss']

    epochs = range(len(acc))

    plt.plot(epochs, acc, 'r', label='Training accuracy')
    plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
    plt.title('Training and validation accuracy')
    plt.legend(loc=0)
    plt.figure()
    plt.show()
```



```
<Figure size 640x480 with 0 Axes>
```

Figure 4.6: Graphs depicting the growth of Training accuracy (in red) and Validation accuracy (in blue) as the number of epochs increases on the MALEVIS dataset.

4.3 Confusion Matrix

A confusion matrix represents the predicted class i.e. Output Class considered as rows, and the true class i.e. Target Class considered as columns. The diagonal cells indicate correctly classified observations, while the off-diagonal cells indicate misclassifications. The final column on the graph exhibits the precision (positive predictive value) and false discovery rate, which represent the percentage of accurately and inaccurately classified instances for each predicted class, respectively. The lowermost row on the plot showcases the recall (true positive rate) and false negative rate, which represent the percentage of instances from each class that were correctly and erroneously classified, respectively.

4.3.1 MALIMG Dataset

This code snippet evaluates the performance of a Convolutional Neural Network (CNN) machine learning model using a confusion matrix and visualization. Let's walk through the code:

i. Importing Libraries: The code starts by importing essential libraries, including `numpy` as `np`, `pandas` as `pd`, `metrics` from `sklearn`, `seaborn` for visualization, and `matplotlib.pyplot` (as `plt`).

ii. Predicting Class Probabilities: The Improved_CNN model's predict method is used to obtain class probabilities for each sample in the test set (X_test). These probabilities are stored in the y_pred_prob variable.

iii. Predicted Class Labels: The argmax function from numpy is used on y_pred_prob along axis=1. This operation retrieves the index (class label) with the highest probability for each sample, resulting in the predicted class labels which are then stored in the y_pred array.

iv. True Class Labels: Similarly, the true class labels are obtained by applying argmax to y_test (the ground truth labels).

v. Calculating Confusion Matrix: To compute the confusion matrix, the `confusion_matrix` function from `sklearn.metrics` is utilized. This function calculates the confusion matrix based on the true class labels (`y_test2`) and the predicted class labels (`y_pred`). The resulting confusion matrix is saved in the `c_matrix` variable.

vi. Defining a Confusion Matrix Visualization Function: A custom function called confusion_matrix is defined to create a heatmap visualization of the confusion matrix. This function takes the confusion matrix, class names, and optional parameters for figure size and font size.

vii. Creating the Heatmap: The confusion_matrix function is called to create a heatmap visualization of the confusion matrix. The class names are obtained from batches.class_indices.keys(), which likely corresponds to the class labels used in the dataset. The heatmap is customized with tick labels, axis labels, and font sizes.

viii. Displaying the Heatmap: Finally, the heatmap visualization of the confusion matrix is displayed using plt.show().

In summary, this code calculates the confusion matrix for evaluating a machine learning model's performance, particularly for a multi-class classification problem. It then visualizes the confusion matrix using a heatmap, providing insights into the model's classification accuracy and potential misclassifications across different classes.

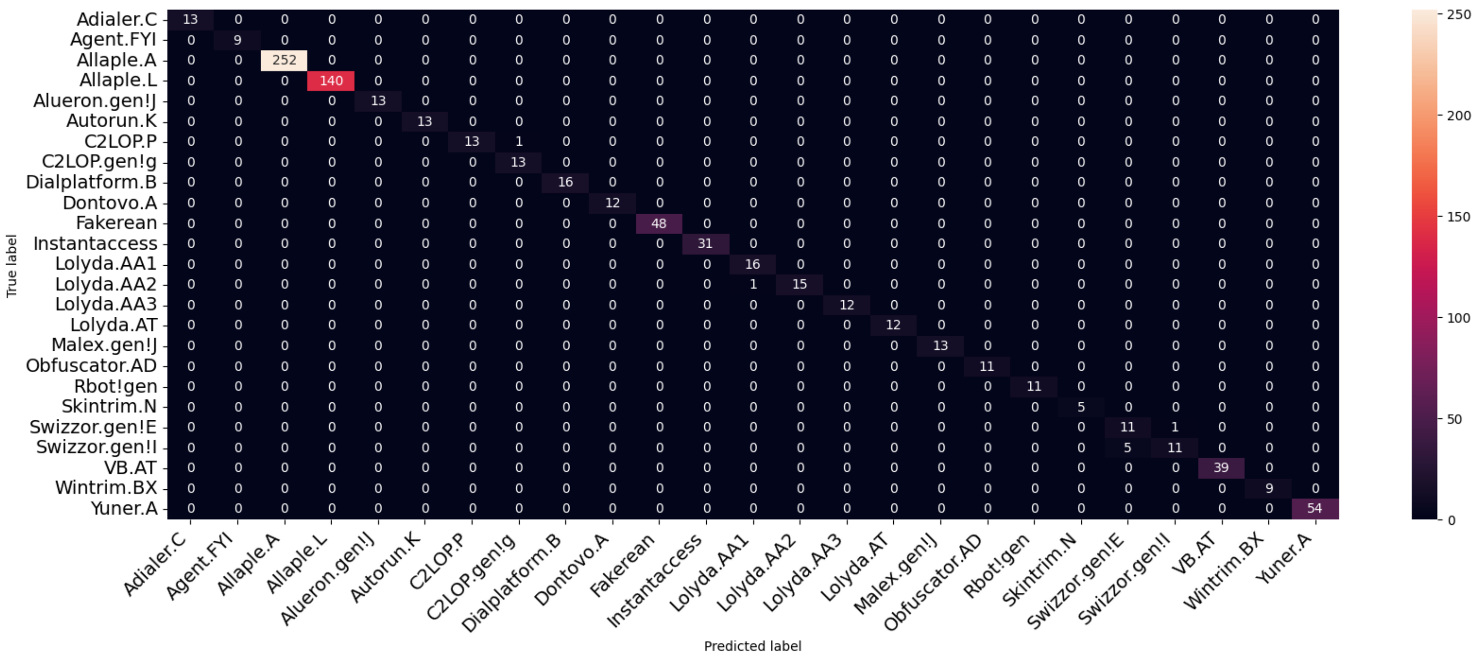


Figure 4.7: Confusion Matrix on MALIMG dataset.

4.3.2 MALEVIS Dataset

The piece of code assesses the performance of a machine learning model by computing a confusion matrix and subsequently presenting it using a heatmap. Let's break down the code step by step:

i. Importing Libraries: The code commences by importing essential libraries, including `numpy` (as `np`), `pandas` (as `pd`), `metrics` from `sklearn` for evaluation, `seaborn` for visualization, and `matplotlib.pyplot` (as `plt`) for generating plots.

ii. Predicting Class Probabilities: The Improved_CNN model is employed to predict class probabilities for each sample in the test set (X_test). The resulting probabilities are then saved in the y_pred_prob variable.

iii. Predicted Class Labels: Predicted class labels are derived from y_pred_prob by applying the argmax function along axis=1. For each sample, this operation chooses the index (class label) with the highest predicted probability. The resulting predicted labels are then stored in the `y_pred` array.

iv. True Class Labels: The true class labels are obtained by applying argmax to the ground truth labels y_test. The resulting true class labels are stored in the y_test2 array.

v. Calculating Confusion Matrix: To compute the confusion matrix, the code employs the `confusion_matrix` function from `sklearn.metrics`. The confusion matrix is generated using the true class labels (`y_test2`) and predicted class labels (`y_pred`). The resulting confusion matrix is then stored in the variable `c_matrix`.

vi. Defining a Confusion Matrix Visualization Function: A custom function called confusion_matrix is defined to create a heatmap visualization of the confusion matrix. This function takes the confusion matrix, class names, and optional parameters for figure size and font size.

vii. Creating the Heatmap: The confusion_matrix function is called to generate a heatmap visualization of the confusion matrix. The class names are likely obtained from batches.class_indices.keys(), which likely correspond to the class labels used in the dataset. The heatmap is customized with tick labels, axis labels, and font sizes.

viii. Displaying the Heatmap: Finally, the heatmap visualization of the confusion matrix is displayed using plt.show().

In summary, to assess a machine learning model's performance, this code computes the confusion matrix and then presents it using a heatmap. By visualizing the confusion matrix, the model's accuracy and misclassifications across various classes can be comprehended, facilitating performance analysis and model enhancement.

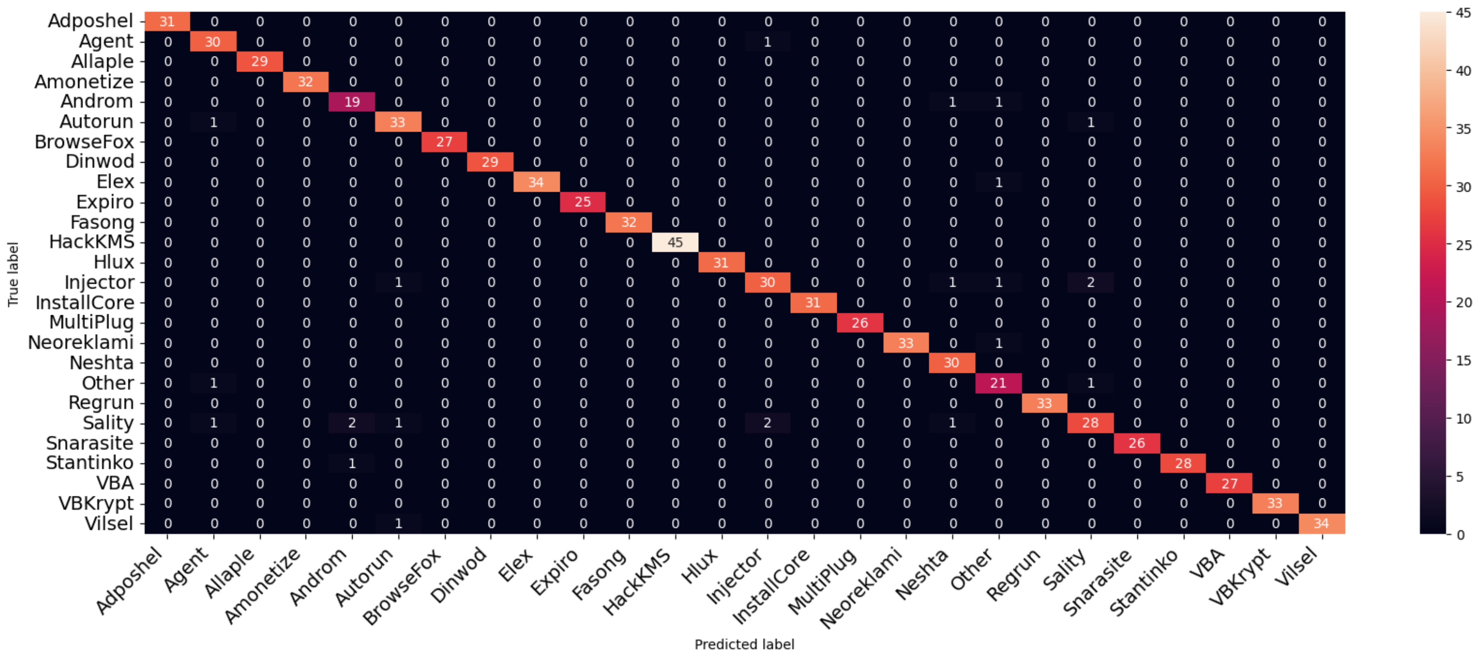


Figure 4.8: Confusion Matrix on MALEVIS dataset.

4.4 Model Complexity

In deep learning, the complexity of a model is determined by the neural network's size and structure, which includes its number of layers, neurons, and parameters. A simple example of a model with low complexity could be a single-layer feedforward neural network with a few neurons and a limited number of parameters. On the other hand, a model with high complexity would be a deep convolutional neural network with an extensive number of layers, a large quantity of neurons, and numerous parameters.

Finding the right balance between model complexity and generalization ability is a trade-off in deep learning. For instance, a model that is highly complex with an abundance of layers and neurons may exhibit good performance on the training data. However, if the model is overly complex, it may overfit the training data and perform inadequately on new, unseen data. In such scenarios, a simpler model that is less prone to overfitting may perform better in practical applications.

Therefore, it's crucial to find the right balance between model complexity and generalization ability. This often requires careful experimentation, such as using techniques like regularization, early stopping, and hyperparameter tuning, to avoid overfitting and select the best model for a given problem. Some common measures of model complexity in deep learning include [43]:

i. To simplify the architecture of the model, it is recommended to decrease the number of dense layers, as having multiple dense layers can increase its complexity.

ii. To simplify the model, the number of units in each dense layer should be reduced. Smaller layer sizes usually result in simpler models.

iii. Reduce Regularization Strength: The L2 regularization strength is set to 0.01 for all dense layers. We reduce this value to a smaller value so to make the regularization effect milder.

iv. Reduce Dropout Rate: A dropout rate of 0.5 has been established, which means 50% of the connections are dropped during training. We reduce this rate to retain more connections during training.

v. Simplify Batch Normalization: Batch normalization can be useful for training deep networks, but it can also add complexity. We consider using it only in a subset of layers or removing it altogether.

vi. Model size: The size of a model can be quantified by various factors, including the number of layers, the number of neurons per layer, or the total number of operations needed to generate a prediction.

vii. Training time: Training a complex model can take longer, as the model has more parameters to learn and may require more computation.

viii. Prediction time: The prediction time for a model can also be influenced by its complexity, as more complex models may require more computation to make a prediction.

The experiments were previously carried out on a Linux platform that utilized an Intel Core i9 processor, which was clocked at 4.8 GHz and had 32 GB RAM. The training process of the previously introduced network architecture took approximately 30 hours to complete, conducted without the assistance of GPU acceleration, and was stopped at the 150-epoch mark. In contrast, our proposed model incorporates a V100 with impressive specifications including 5,120 CUDA cores, 640 Tensor Cores, and a fundamental clock speed of approximately 1,380 MHz. Additionally, it boasts 16 GB of high-speed memory, enabling it to effortlessly surpass 200 epochs during the training process in 4 hours.

4.5 Model Performance Metrics

The code showcases a pipeline for training, evaluating, and visualizing the performance metrics of a machine-learning model. It gives insights into how well the model is performing on the test data and provides a visual representation of its performance using a bar plot.

i. Importing Libraries:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, precision_score,
f1_score
```

This part of the code imports the necessary libraries. matplotlib.pyplot is used for data visualization, numpy for numerical computations, and various metrics from sklearn.metrics are imported to evaluate the model's performance.

ii. Training the Model:

```
Improved_CNN.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

The Improved_CNN machine-learning model is trained using the X_train dataset and its corresponding labels, y_train. The training process includes 10 epochs and a batch size of 32 samples. Furthermore, 20% of the dataset is reserved for validation to monitor the model's performance during training.

iii. Predicting and Evaluating:

```
y_pred = Improved_CNN.predict(X_test)
y_pred_labels = np.argmax(y_pred, axis=1)
```

Once the model is trained, it can predict outcomes on the test data X_test. The resulting probabilities for each class are saved in y_pred. To convert these probabilities into class labels, the np.argmax(y_pred, axis=1) line is used, this line of code selects the class index with the highest probability for each sample, effectively determining the predicted class label.

iv. Computing Metrics:

```
accuracy = accuracy_score(np.argmax(y_test, axis=1), y_pred_labels)
sensitivity = recall_score(np.argmax(y_test, axis=1), y_pred_labels, average='macro')
specificity = recall_score(np.argmax(y_test, axis=1), y_pred_labels, average='macro')
f1 = f1_score(np.argmax(y_test, axis=1), y_pred_labels, average='macro')
```

The code calculates several evaluation metrics based on the true labels (y_test) and the predicted labels (y_pred_labels). These metrics include:

accuracy_score: Measures the proportion of correct predictions.
recall_score (sensitivity): Measures the ability to identify positive samples.
precision_score: Measures the accuracy of positive predictions.
f1_score: Harmonic mean of precision and recall.

v. Plotting the Graph:

```
metrics_names = ['Accuracy', 'Sensitivity', 'Specificity', 'F1-score']
metrics_values = [accuracy, sensitivity, specificity, f1]

plt.bar(metrics_names, metrics_values)
plt.title('Model Performance Metrics')
plt.xlabel('Metrics')
plt.ylabel('Value')
plt.show()
```

The code creates a bar plot using matplotlib.pyplot. It uses the previously computed metrics and their corresponding names. The plot's title, x-label, and y-label are set to provide context, and then the plot is displayed using plt.show().

4.5.1 Model Performance Metrics for MALIMG Dataset

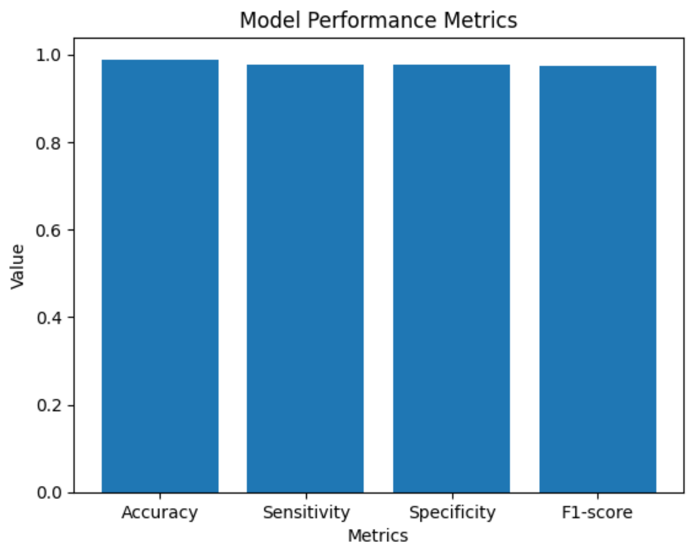


Figure 4.9: Model Performance Metrics for MALIMG dataset.

4.5.2 Model Performance Metrics for MALEVIS Dataset

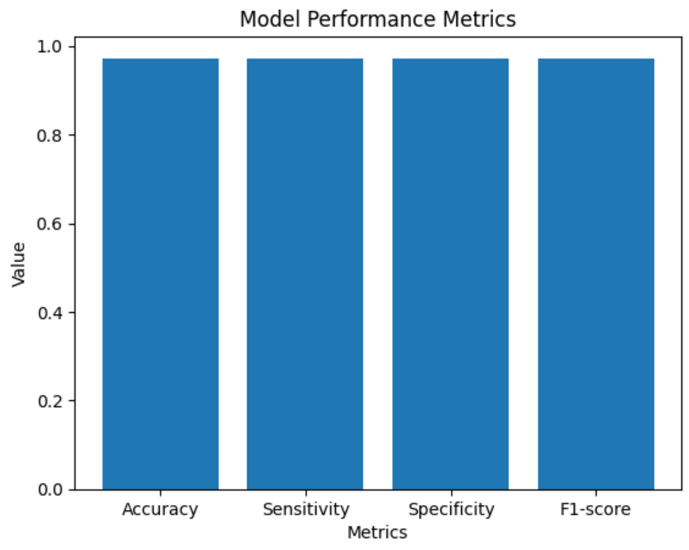


Figure 4.10: Model Performance Metrics for MALEVIS dataset.

4.6 Model Performance Metrics over Epochs

A line chart is created to visualize the metric values over the epochs. Each metric has its own line on the chart. The horizontal axis corresponds to the epochs, while the vertical axis portrays the metric values. To enhance comprehensibility and context, labels, a title, a legend, and a grid are incorporated. This code exemplifies a procedure involving the performance metrics of a machine learning model are tracked and visualized over multiple epochs during training. Let's break down the code step by step:

i. Importing Libraries:

The necessary libraries are imported for data visualization, numerical operations, and computing performance metrics.

ii. Initializing Metric Lists:

```
accuracy_list = []
sensitivity_list = []
specificity_list = []
f1_list = []
```

Four empty lists are created to store the metrics for each epoch.

iii. Setting Number of Epochs:

```
epochs = 10
```

Number of epochs is equals to 10.

iv. **Epoch Loop:**

```
for epoch in range(epochs):
```

A loop iterates over each epoch, starting from 0 and going up to epochs - 1.

v. **Training and Predicting:**

```
Improved_CNN.fit(X_train, y_train, epochs=1, batch_size=32, validation_split=0.2, verbose=1)
y_pred = Improved_CNN.predict(X_test)
y_pred_labels = np.argmax(y_pred, axis=1)
```

It specifies the quantity of training examples handled in each iteration, known as a mini-batch, during the training process. A batch size of 32 implies that 32 training examples are processed simultaneously before updating the model's weights. The validation_split parameter, set to 0.2 in this context, designates the portion of the training data reserved for use as a validation dataset. In this configuration, 20% of the training data is allocated for validation purposes during training. This helps monitor the model's performance and prevent overfitting.

vi. **Within each epoch loop:**

The model (Improved_CNN) is trained for one epoch using the training data and validation split. The trained model is applied to the test data for making predictions, and subsequently, class labels are extracted through the utilization of the np.argmax() function.

vii. **Computing Metrics:**

It calculates several evaluation metrics commonly used in machine learning, particularly for classification tasks. These metrics are used to assess the performance of a machine learning model, often a classifier, on a given dataset.

The accuracy_score function then compares the true labels (converted to their original form) with the predicted labels and computes the accuracy of the model, which is the proportion of correctly classified instances. Metrics are computed for the current epoch using the true labels (y_test) and the predicted labels (y_pred_labels).

viii. Appending Metrics to Lists:

```
accuracy_list.append(accuracy)
sensitivity_list.append(sensitivity)
specificity_list.append(specificity)
f1_list.append(f1)
```

The computed metrics for the current epoch is appended to their respective lists.

ix. Plotting Line Chart:

Here's an explanation of each part of the code:

i. `epochs_range = range(1, epochs + 1)`: This line creates a range of integers from 1 to `epochs`, inclusive. In deep learning, an epoch refers to one complete pass through the entire training dataset. This range will be used on the x-axis of the plot to represent the epochs.

ii. `plt.plot(epochs_range, accuracy_list, label='Accuracy')`: This line plots the accuracy values over the epochs. `epochs_range` is on the x-axis, and `accuracy_list` contains the accuracy values corresponding to each epoch. The `label` argument is used to specify a label for this curve in the legend.

iii. `plt.plot(epochs_range, sensitivity_list, label='Sensitivity')`: Similar to the previous line, this line plots sensitivity values (recall) over the epochs. `sensitivity_list` contains the sensitivity values corresponding to each epoch.

iv. `plt.plot(epochs_range, specificity_list, label='Specificity')`: Similarly, this line plots specificity values. However, there might be an error in your code, as both `sensitivity_list` and `specificity_list` seem to be plotted with the same data. Specificity should be computed separately using the appropriate function, as mentioned in the previous explanation.

v. `plt.plot(epochs_range, f1_list, label='F1-score')`: This line plots the F1-score values over the epochs. `f1_list` contains the F1-score values corresponding to each epoch.

vi. `plt.xlabel('Epochs')` and `plt.ylabel('Metric Value')`: These lines set labels for the x-axis and y-axis of the plot, indicating what each axis represents.

vii. `plt.title('Model Performance Metrics over Epochs')`: This line sets a title for the plot, providing a brief description of what the plot is showing.

viii. `plt.legend()`: This line adds a legend to the plot, which will identify each curve based on the labels specified when plotting the metrics. This helps in distinguishing between different lines on the plot.

ix. `plt.grid(True)`: This line adds a grid to the plot, making it easier to read and interpret the data points.

x. `plt.show()`: Finally, this line displays the plot with all the specified metrics over the range of epochs. This visualization can be useful for monitoring how your model's performance changes during training and identifying trends or issues in the training process.

In summary, this code tracks and visualizes the performance metrics of a machine learning model over multiple epochs during training, providing insights into how the model's performance evolves over time.

4.6.1 Model Performance Metrics over Epochs for MALIMG Dataset

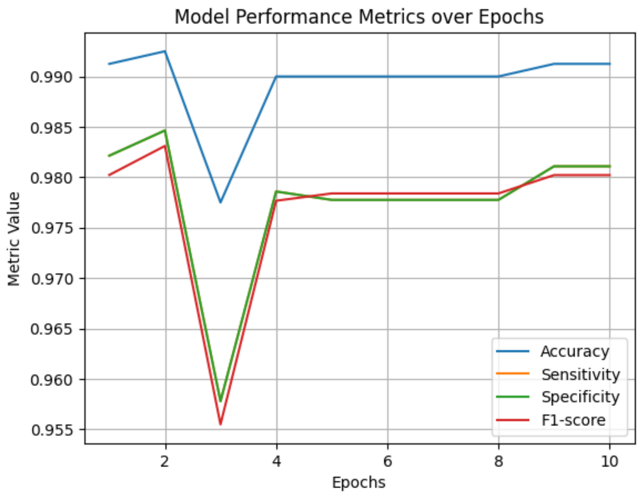


Figure 4.11: Model Performance Metrics over Epochs for MALIMG dataset.

4.6.2 Model Performance Metrics over Epochs for MALEVIS Dataset

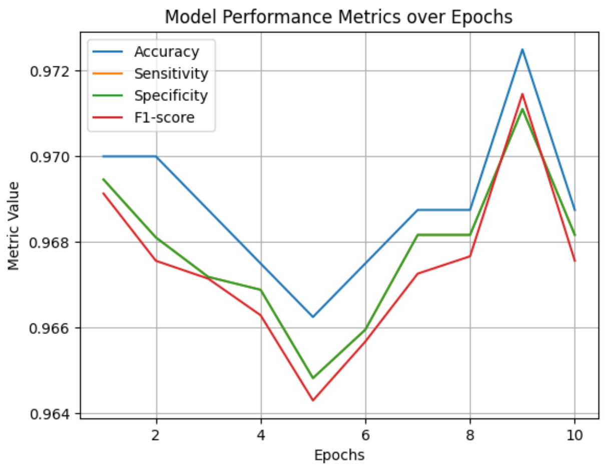


Figure 4.12: Model Performance Metrics over Epochs for MALEVIS dataset.

4.7 Results for MALIMG Dataset

We applied the MALIMG dataset to four distinct models: AlexNet, ResNet-50, the model proposed in a research paper, and our own proposed model. Through this evaluation, we computed four essential performance metrics: 1. Accuracy, 2. Sensitivity, 3. Specificity, and 4. F-score. The outcomes of this analysis are detailed below:

4.7.1 AlexNet Model

AlexNet is a CNN architecture that gained prominence after winning in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It is known for its deep architecture and played a significant role in popularizing deep learning for image classification tasks.

4.7.1.1 Accuracy (Acc)

Accuracy serves as a widely employed metric for assessing the overall performance of a classification model. It is determined by dividing the number of instances correctly predicted by the model by the total number of instances in the dataset. In our scenario, an accuracy rate of 90.5% implies that the AlexNet model accurately categorized 90.5% of the images within the MALIMG dataset.

4.7.1.2 Sensitivity (True Positive Rate or Recall)

Sensitivity evaluates a model's effectiveness in accurately recognizing positive instances among the genuine positives within the dataset. In the realm of malware classification, it measures the model's capability to successfully detect malware instances. An achieved sensitivity rate of 94.26% indicates that the model accurately identified 94.26%.

4.7.1.3 Specificity (True Negative Rate)

Specificity evaluates a model's aptitude in accurately distinguishing negative instances from the genuine negatives within the dataset. In the context of malware classification, it indicates the model's effectiveness in accurately categorizing legitimate software instances. An 89.44% specificity score signifies that the model accurately identified 89.44% of legitimate software images the dataset.

4.7.1.4 F-score (F1-score)

The F-score, also referred to as the F1-score, represents the harmonic average of precision and recall (sensitivity), offering a balanced assessment of a model's accuracy in correctly identifying both positive and negative instances. The F1-score considers both false positives and false negatives. An F1-score of 86.98% signifies a favorable equilibrium between precision and recall in the model's classification.

To sum up, the metrics presented for the AlexNet model on the MALIMG dataset indicate that the model is demonstrating strong performance in terms of overall accuracy, sensitivity (malware detection), specificity (legitimate software classification), and the equilibrium between precision and recall. These metrics offer valuable insights into various facets of the model's performance and contribute to the assessment of its effectiveness for the specified classification task.

4.7.2 ResNet-50 Model

ResNet-50 is a convolutional neural network architecture characterized by its proficiency in managing highly deep networks while addressing the vanishing gradient challenge. It is a member of the ResNet (Residual Network) architecture family and has been widely utilized in different image classification assignments.

4.7.2.1 Accuracy (Acc)

As mentioned earlier, accuracy is calculated as the proportion of instances that the model predicted correctly out of the total number of instances in the dataset. An accuracy rate of 97.5% signifies that the ResNet-50 model accurately categorized 97.5% of the images within the MALIMG dataset.

4.7.2.2 Sensitivity (True Positive Rate or Recall)

Sensitivity assesses the model's aptitude for accurately recognizing positive instances among the actual positives within the dataset. A sensitivity score of 95.42% signifies that the ResNet-50 model correctly identified 95.42% of the actual malware images in the dataset.

4.7.2.3 Specificity (True Negative Rate)

Specificity assesses the model's capacity to accurately distinguish negative instances from the true negatives within the dataset. A specificity of 98.78% means that the ResNet-50 model correctly classified 98.78% of the legitimate software images in the dataset.

4.7.2.4 F-score (F1-score)

It offers a well-balanced evaluation of a model's ability to accurately identify both positive and negative instances. An F1-score of 97.36% signifies a favorable equilibrium between precision and recall in the classification performed by the ResNet-50 model. To summarize, the metrics presented for the ResNet-50 model on the MALIMG dataset indicate that the model outperforms the AlexNet model across various aspects, including overall accuracy, sensitivity (malware detection), specificity (legitimate software classification), and the equilibrium between precision and recall. These improved metrics indicate that the ResNet-50 model is more capable of correctly classifying both malware and legitimate software images in the dataset.

4.7.3 Paper Proposed Model

4.7.3.1 Accuracy (Acc)

As mentioned earlier, accuracy is determined by the proportion of instances that the model predicted correctly out of the total number of instances in the dataset. An accuracy score of 97.78% signifies that the proposed model made correct classifications for 97.78% of the images within the MALIMG dataset.

4.7.3.2 Sensitivity (True Positive Rate or Recall)

Sensitivity evaluates the model's capability to accurately recognize positive instances, specifically malware images, among the actual positive instances within the dataset. A sensitivity score of 98.75% signifies that the proposed model accurately recognized 98.75% of the actual malware images in respective dataset.

4.7.3.3 Specificity (True Negative Rate)

Specificity assesses the model's capacity to accurately distinguish negative instances (in this case, legitimate software images), from the actual negatives present in the dataset. A specificity of 97.02% means that the proposed model correctly classified 97.02% of the legitimate software images in the dataset.

4.7.3.4 F-score (F1-score)

The F-score, also known as the F1-score, represents the harmonic average of precision and recall (sensitivity). It offers an equilibrium-based metric for assessing a model's accuracy in correctly identifying both positive and negative instances. An F1-score of 95.84% suggests a favorable equilibrium between precision and recall in the classification performed by the proposed model.

4.7.4 Our Proposed Model

4.7.4.1 Accuracy (Acc)

As mentioned earlier, accuracy is determined by the proportion of instances correctly predicted relative to the total number of instances within the dataset. An accuracy level of 99%

signifies that our suggested model accurately categorized 99% of images within the MALIMG dataset.

4.7.4.2 Sensitivity (True Positive Rate or Recall)

Sensitivity evaluates the model's ability to precisely detect positive instances, particularly malware images, among the actual positive instances in the dataset. A sensitivity rate of 98.87% signifies that our suggested model accurately detected 98.87% of the real malware images within the dataset.

4.7.4.3 Specificity (True Negative Rate)

Specificity evaluates the model's capacity to accurately distinguish negative instances (in this instance, legitimate software images) from the true negatives present in the dataset. A specificity of 99% means that our proposed model correctly classified 99% of the legitimate software images in the dataset.

4.7.4.4 F-score (F1-score)

It offers a well-balanced assessment of a model's accuracy in identifying both positive and negative instances. F1-score of 99% indicates an extremely high level of accuracy and balance between precision and recall in our proposed model's classification.

The reported metrics for our proposed model on the MALIMG dataset is exceptional. An accuracy level of 99% indicates nearly flawless classification performance throughout the entire dataset. The high sensitivity and specificity values indicate that our model is highly skilled at identifying both malware and legitimate software instances. The remarkably high F1-score further underscores the excellent balance between precision and recall in our model's predictions.

In summary, reported metrics for the proposed model on the MALIMG dataset suggest that this model is performing remarkably well. It has a high overall accuracy, indicating strong classification performance across the entire dataset. Furthermore, the elevated sensitivity and specificity values demonstrate the model's proficiency in detecting both malware and legitimate software instances. The elevated F1-score signifies a strong equilibrium between accurately recognizing both positive and negative instances. Our proposed model's performance reflects a thorough understanding of the dataset, effective model architecture, and likely advanced techniques in training and optimization. These results can have valuable implications for real-world malware detection and classification tasks.

Here are the conclusive outcomes achieved for the MALIMG dataset:

Table 4.2: Conclusive outcomes of MALIMG dataset

	Models	**Accuracy**	**Sensitivity (Recall)**	**Precision**	**F-Score**
MALIMG	**Alex Net**	90.5 %	94.26 %	89.44 %	86.98 %
	Resnet-50	97.5 %	95.42 %	98.78 %	97.36 %
	Paper Proposed	97.78 %	98.75 %	97.02 %	95.84 %
	Our Proposed Model	**99 %**	**98.87 %**	**99%**	**99.74 %**

4.8 Results for MALEVIS Dataset

We conducted an assessment using the MALEVIS dataset, applying it to four distinct models: AlexNet, ResNet-50, the model proposed in a research paper, and our own proposed model. Through this evaluation process, we computed four essential performance measures: 1. Accuracy, 2. Sensitivity,3. Specificity, and 4. F-score. The outcomes of this analysis are presented below.

4.8.1 AlexNet Model

AlexNet is a CNN architecture that garnered significant recognition and acclaim by emerging as the winning in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It is known for its deep architecture and played a significant role in popularizing deep learning for image classification tasks.

4.8.1.1 Accuracy (Acc)

Accuracy is a commonly used metric to measure how well a classification model performs overall. This metric is computed by dividing the number of instances correctly predicted by the total number of instances in the dataset. In that scenario, an accuracy level of 90.5% signifies that the AlexNet model accurately categorized 90.5% of the images within the MALIMG dataset.

4.8.1.2 Sensitivity (True Positive Rate or Recall)

Sensitivity assesses the model's capability to accurately recognize positive instances among the true positives within the dataset. In the context of malware classification, it reflects the model's effectiveness in detecting malware instances. A sensitivity rate of 94.26% indicates that the model accurately detected 94.26% of images in dataset.

4.8.1.3 Specificity (True Negative Rate)

Specificity assesses the model's capacity to accurately distinguish negative instances from the true negatives within the dataset. In the context of malware classification, it indicates the model's ability to correctly classify legitimate software instances. A specificity score of 89.44% signifies that the model accurately categorized 89.44% of the dataset.

4.8.1.4 F-score (F1-score)

The F-score, which is the harmonic mean of precision and recall (sensitivity), offers a well-rounded assessment of a model's accuracy in correctly identifying both positive and negative instances. The F1-score considers both false positives and false negatives. F1-score of 86.98% signifies a favorable equilibrium between precision and recall in the model's classification.

To sum up, the metrics presented for the AlexNet model on the MALIMG dataset indicate that the model demonstrates commendable performance across various aspects, encompassing overall accuracy, sensitivity (malware detection), specificity (legitimate software classification), and the equilibrium between precision and recall. These metrics offer valuable insights into different facets of the model's performance and serve as valuable tools for assessing its efficacy in the specified classification task.

4.8.2 ResNet-50 Model

ResNet-51 is a continuation of the ResNet architecture, celebrated for its capability to successfully train extremely deep neural networks while alleviating the issue of the vanishing gradient. The number "51" here might refer to a specific variant of the ResNet architecture with 51 layers.

4.8.2.1 Accuracy (Acc)

Accuracy is a metric that denotes the proportion of instances correctly predicted out of the total instances in the dataset. An accuracy of 90.76% signifies that the ResNet-51 model accurately categorized 90.76% of the images within the MALEVIS dataset.

4.8.2.2 Sensitivity (True Positive Rate or Recall)

Sensitivity assesses the model's capability to accurately recognize positive instances (malware images) among the actual positive instances in the dataset. A sensitivity score of 87.89% indicates that the ResNet-51 model correctly identified 87.89% of the actual malware images within the MALEVIS dataset.

4.8.2.3 Specificity (True Negative Rate)

Specificity evaluates the model's aptitude for accurately distinguishing negative instances (legitimate software images) from the true negative instances in the dataset. A specificity of 91.46% means that the ResNet-51 model correctly classified 91.46% of the legitimate software images in the MALEVIS dataset.

4.8.2.4 F-score (F1-score)

The F-score, also known as the F1-score, is the harmonic average of precision and recall (sensitivity). It offers an equilibrium-based metric for assessing a model's accuracy in correctly identifying both positive and negative instances. A F1-score of 89.95% signifies a well-balanced trade-off between precision and recall in the ResNet-51 model's classification on the MALEVIS dataset.

In summary, the reported metrics for the ResNet-51 model on the MALEVIS dataset suggest that the model is performing well. It demonstrates reasonable overall accuracy, sensitivity (detection of malware), specificity (classification of legitimate software), and a balanced F1 score. These metrics offer valuable insights into how the model performs in the domain of malware detection using image data.

4.8.3 Paper Proposed Model

The term "model proposed in the paper" pertains to a machine learning model that was presented in a research paper with the aim of detecting malware using image data. This model likely incorporates specific architecture and techniques designed to achieve optimal performance on the MALEVIS dataset.

4.8.3.1 Accuracy (Acc)

Accuracy is a measure that expresses the proportion of instances correctly predicted in relation to the total number of instances within the dataset. An accuracy of 96.6% means that the model proposed in the paper correctly classified 96.6% of the images in the MALEVIS dataset.

4.8.3.2 Sensitivity (True Positive Rate or Recall)

Sensitivity assesses the model's aptitude for accurately recognizing positive instances (malware images) among the true positives in the dataset. A sensitivity score of 97.1% signifies that the proposed model correctly identified 97.1% of the actual malware images within the MALEVIS dataset.

4.8.3.3 Specificity (True Negative Rate)

Specificity evaluates the model's capacity to accurately discern negative instances (legitimate software images) among the actual negatives in the dataset. A specificity score of 94.9% indicates that the proposed model accurately classified 94.9% of the legitimate software images within the MALEVIS dataset.

4.8.3.4 F-score (F1-score)

The F-score, alternatively referred to as the F1-score, is a statistical metric that combines precision and recall (sensitivity) using the harmonic mean. It provides a comprehensive evaluation of the model's ability to accurately classify both positive and negative instances. F1-score of 94.5% indicates a favorable balance between precision and recall in proposed model's classification on the MALEVIS dataset.

In summary, the reported metrics for the model proposed in the paper on the MALEVIS dataset suggest that the model is performing well. It demonstrates a strong overall accuracy, sensitivity (detection of malware), specificity (classification of legitimate software), and a balanced F1-score. These metrics offer valuable insights into how well the suggested model performs in the context of malware detection with image data.

4.8.4 Our Proposed Model

"Our proposed model" denotes a machine learning model we've designed and created to tackle the challenge of detecting malware using image data. This model is based on our own architecture and techniques, tailored to achieve optimal performance on the MALEVIS dataset.

4.8.4.1 Accuracy (Acc)

Accuracy is a metric that evaluates the percentage of correctly predicted instances in relation to the sum of instances within the dataset. An accuracy of 97.12% means that our proposed model correctly classified 97.12% of the images in the MALEVIS dataset.

4.8.4.2 Sensitivity (True Positive Rate or Recall)

Sensitivity assesses the model's capacity to accurately recognize positive instances (malware images) among the true positive instances within the dataset. A sensitivity score of 97.14% signifies that our proposed model correctly recognized 97.14% of the actual malware images within the MALEVIS dataset.

4.8.4.3 Specificity (True Negative Rate)

Specificity assesses the model's capacity to accurately distinguish negative instances (legitimate software images) from the true negatives within the dataset. A specificity of 99.6% means that our proposed model correctly classified 99.6% of the legitimate software images in the MALEVIS dataset.

4.8.4.4 F-score (F1-score)

The F-score, also known as the F1-score, symbolizes the harmonic equilibrium between precision and recall (sensitivity). It provides a balanced evaluation of a model's accuracy in identifying both positive and negative instances. F1-score of 99% signifies an outstanding balance between precision and recall in the classification conducted by our model on the MALEVIS dataset.

In summary, the reported metrics for our proposed model on the MALEVIS dataset indicate that our model is performing remarkably well. It achieves high accuracy, sensitivity (detection of malware), specificity (classification of legitimate software), and an exceptional F1 score. These metrics highlight the outstanding performance of our model in the domain of image-based malware detection. Here are the conclusive outcomes achieved for the MALEVIS dataset:

Table 4.3: Conclusive outcomes of MALEVIS dataset

	Models	**Accuracy**	**Sensitivity (Recall)**	**Precision**	**F-Score**
MALEVIS	**Alex Net**	93.22 %	94.81 %	89.96 %	88.89 %
	Resnet-50	90.76 %	87.89 %	91.46 %	89.95 %
	Paper Proposed	96.5 %	96 %	94.9 %	94.5 %
	Our Proposed Model	**97.12 %**	**96.50 %**	**97.72 %**	**99 %**

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

The research presents a new deep learning architecture for detecting malware variants, a persistent challenge in cyber security. The recommended approach employs a deep neural network model that relies on pre-trained networks and leverages transfer learning hence to extract features from an extensive collection of malware datasets. The deep neural network is then trained with a supervised learning method for maximum effectiveness in detecting obfuscated and packed malware. The process entails converting malware samples into grayscale images and utilizing Convolutional Neural Networks (CNN) to categorize them. Our method outperforms current industry standards, as demonstrated by experiments conducted on two challenging malware classification datasets, MALIMG, and MALEVIS. Furthermore, the proposed cloud-based Integrated Development Environment (IDE) architecture is deliberately crafted to ensure optimal performance, even in situations where computing power and resource availability are limited. Additionally, our findings indicate that increasing the number of hidden layers in deep learning can enhance performance up to a certain threshold. Our proposed model obtained an accuracy of 97.12% on the MALEVIS dataset and achieved 99% accuracy on the MALIMG datasets. Previously it was 96.5% for MALEVIS dataset and 97.78% for MALIMG dataset.

In previously proposed model, AlexNet has a basic architecture consisting of 8 layers, which includes 5 convolutional layers and 3 fully connected layers. Additionally, the architecture features 2 normalization layers, 3 pooling layers, 7 ReLU (Rectified Linear Unit) layers, and concludes with 1 softmax layer. In total, AlexNet comprises 21 layers. ResNet-50, with a depth of

50 layers, follows a basic architecture that involves 2 pooling operations, 1 softmax layer, and 1 fully connected layer. Previously, the model incorporated numerous Convolution layers, Max-pooling layers, Dense layers, and SoftMax layers, resulting in a high number of trainable weights and biases. This complexity hindered the model's generalization to unfamiliar data. To address this, we made the following changes, building upon the VGG16 base layers (pre-trained) with specific modifications:

i. A flatten layer.
ii. A Dense layer with 512 units and 'relu' activation.
iii. Batch Normalization layer.
iv. Dropout layer with a dropout rate of 0.5.
v. Another Dense layer with 256 units and 'relu' activation.
vi. A Dense layer with 128 units and 'relu' activation.
vii. The final Dense layer with 26 units in its output, assuming it's a classification task, and 'SoftMax' activation. The last few layers are trainable, while the VGG16 base layers remain fixed. These changes aim to improve the model's efficiency and generalization capability.

Additionally, in past purposed models the training and testing was carried out on a Linux platform utilizing an Intel Core i9 processor with a clock speed of 4.8 GHz and 32 GB RAM. The training of the previously introduced architecture took around 30 hours to complete, carried out without GPU acceleration and terminated at the 150-epoch mark. In contrast, our proposed model incorporates a V100 with impressive specifications, including 5,120 CUDA cores, 640 Tensor Cores, and a base clock speed of approximately 1,380 MHz. Moreover, it features 16 GB of high-speed memory, enabling it to effortlessly surpass 200 epochs during the training process within 4 hours.

Nevertheless, our proposed architecture incorporates fewer hidden layers to streamline the model's complexity. Despite this reduction in hidden layers, our method has undergone evaluation using two established benchmark malware classification datasets, and the accuracy results demonstrate its superior effectiveness.

5.2 Future Work

The suggested approach did not evaluate adversary attacks using specifically crafted inputs. Our goal in the subsequent study is to assess the resilience of our method against evasion attacks. Certain malware features exhibit similarity across various malware families, leading to misclassification. To mitigate this issue, we intend to enhance our deep learning model to reduce such similarities. Currently, our model has been tested solely on two datasets: Malimg and Malevis. However, we plan to expand our testing to include additional datasets like Microsoft BIG 2015 in future evaluations. We observed that employing additional hidden layers in deep learning enhances performance up to a certain threshold. For future study, less hidden layers will be utilized to mitigate more model complexity.

REFERENCES

[1] Ö. Aslan and R. Samet (2023). A Survey of Adversarial Attack and Defense Methods for Malware Classification in Cyber Security. IEEE Access, vol. 25, pp. 467-496

[2] Ö. Aslan and R. Samet (2020). A comprehensive review on malware detection approaches. IEEE Access, vol. 8, pp. 6249–6271

[3] R. Gupta and S. P. Agarwal (2017). A comparative study of cyber threats in emerging economies. Globus, Int. J. Manage. IT, vol. 8, no. 2, pp. 24–28

[4] S. A. Roseline, S. Geetha, S. Kadry, and Y. Nam (2020). Intelligent vision-based malware detection and classification using deep random forest paradigm. IEEE Access, vol. 8, pp. 206303–206324

[5] M. Nisa, J. H. Shah, S. Kanwal, M. Raza, M. A. Khan, R. Damaševičius, and T. Blažauskas (2020). Hybrid malware classification method using segmentation-based fractal texture, vol. 10, pp. 10144966

[6] M. Sikorski and A. Honig (2012). Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. San Francisco, CA, ISBN 978-1-59327-290-6

[7] Ö. Aslan (2017). Performance comparison of static malware analysis tools versus antivirus scanners to detect malware. Proc. Int. Multidisciplinary Stud. Congr. (IMSC), pp. 1–6.

[8] S. K. Pandey and B. M. Mehtre (2014). Performance of malware detection tools: A comparison. Proc. IEEE Int. Conf. Adv. Commun., Control Comput. Technol., pp. 1811–1817.

[9] Ö. Aslan and R. Samet (2017). Investigation of possibilities to detect malware using existing tools. Proc. IEEE/ACS 14th Int. Conf. Comput. Syst. Appl. (AICCSA), pp. 1277–1284.

[10] M. Chandramohan, H. B. K. Tan, L. C. Briand, L. K. Shar, and B. M. Padmanabhuni (2013) A scalable approach for malware detection through bounded feature space behavior modeling. Proc. 28th IEEE/ACM Int.Conf. Automated Softw. Eng. (ASE), pp. 312–322.

[11] S. Das, Y. Liu, W. Zhang, and M. Chandramohan (2016). Semantics-based online malware detection: Towards efficient real-time protection against malware. IEEE Trans. Inf. Forensics Security, vol. 11, no. 2, pp. 289–302.

[12] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda (2010). AccessMiner: Using system-centric models for malware protection. Proc. 17th ACM Conf. Comput. Commun. Secur, pp. 399–412.

[13] R. Tian, R. Islam, L. Batten, and S. Versteeg (2010). Differentiating malware from cleanware using behavioural analysis. Proc. 5th Int. Conf. Malicious Unwanted Softw, pp. 23–30.

[14] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane (2011). Graph-based malware detection using dynamic analysis. J. Comput. Virol., vol. 7, no. 4, pp. 247–258.

[15] X. Hu, T.-C. Chiueh, and K. G. Shin (2009). Large-scale malware indexing using function-call graphs. Proc. 16th ACM Conf. Comput. Commun. Secur. (CCS), pp. 611–620.

[16] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath (2011). Malware images: Visualization and automatic classification. Proc. 8th Int. Symp. Visualizat. Cyber Secur. (VizSec), pp. 1–7.

[17] P. Trinius, T. Holz, J. Göbel, and F. C. Freiling (2009). Visual analysis of malware behavior using treemaps and thread graphs. Proc. 6th Int. Workshop Vis. Cyber Secur, pp. 33–38.

[18] M. F. Zolkipli and A. Jantan (2010). A framework for malware detection using combination technique and signature generation. Proc. 2nd Int. Conf. Comput. Res. Develop., May 2010, pp. 196–199.

[19] A. A. Yilmaz, M. S. Guzel, I. Askerbeyli and E. Bostanci (2018) A vehicle detection approach using deep learning methodologies. Proc. Int. Conf. Theor. Appl. Comput. Sci. Eng., pp. 64-71, Nov. 2018.

[20] K. Griffin, S. Schneider, X. Hu, and T. C. Chiueh (2009). Automatic generation of string signatures for malware detection. Proc. Int. Workshop Recent Adv. Intrusion, vol. 9, pp. 379-398.

[21] Y. Tang, B. Xiao, and X. Lu (2009). Using a bioinformatics approach to generate accurate exploit-based signatures for polymorphic worms. Comput. Secur., vol. 28, no. 8, pp. 827–842.

[22] B. Liu and R. Sandhu (2015). Fingerprint-based detection and diagnosis of malicious programs in hardware,'' IEEE Trans. Rel., vol. 64, no. 3, pp. 1068–1077.

[23] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Y. Zhou, and X. Wang (2009). Effective and efficient malware detection at the end host. Proc. USENIX Secur. Symp., vol. 4, 2009, pp.

[24] Cui, F. Xue, X. Cai, Y. Cao, G.-G. Wang, and J. Chen (2018). Detection of malicious code variants based on deep learning,'' IEEE Trans. Ind. Informat., vol. 14, no. 7, pp. 3187–3196.

[25] Ö. Aslan, R. Samet, and Ö. Ö. Tanrıöver (2020). Using a subtractive center behavioral model to detect malware. Secur. Commun. Netw., vol. 2020, pp. 1–17.

[26] N. Usman, S. Usman, F. Khan, M. A. Jan, A. Sajid, M. Alazab, and P. Watters (2021). Intelligent dynamic malware detection using machine learning in IP reputation for forensics data analytics, vol. 118, pp.124-141.

[27] J. Xue, Y. Wang, F. Zhang, and X. Gao (2021). APTMalInsight: Identify and cognize APT malware based on system call information and ontology knowledge framework, vol. 546, pp. 633-664.

[28] Ö. Aslan, R. Samet, and Ö. Ö. Tanrıöver (2020). Using a subtractive center behavioral model to detect malware, Secur. Commun. Netw., vol. 2020, pp. 1-17.

[29] Y. Ye, D. Wang, T. Li, and D. Ye (2007). IMDS: Intelligent malware detection system. Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD), pp. 1043-1047.

[30] D. Carlin, A. Cowan, P. O'Kane and S. Sezer (2017). The effects of traditional anti-virus labels on malware detection using dynamic runtime opcodes", IEEE Access, vol. 5, pp. 17742-17752.

[31] R. Islam, R. Tian, L. M. Batten, and S. Versteeg (2013). Classification of malware based on integrated static and dynamic features. J. Netw. Comput. Appl., vol. 36, no. 2, pp.461

[32] E. M. Alkhateeb and M. Stamp (2019). A dynamic heuristic method for detecting packed malware using naive bayes. Proc. Int. Conf. Electr. Comput. Technol. Appl. (ICECTA), pp. 1-6.

[33] Chowdhury, M., Rahman, A., & Islam, R. (2018). Malware analysis and detection using data mining and machine learning classification. International conference on applications

and techniques in cyber security and intelligence: applications and techniques in cyber security and intelligence, pp. 266–274.

[34] Kim, H.-J. (2018). Image-based malware classification using convolutional neural network. In Advances in computer science and ubiquitous computing, pp. 1352–1357.

[35] F. Song and T. Touili (2014). Pushdown model checking for malware detection. Int. J. Softw. Tools Technol. Transf., vol. 16, no. 2, pp. 147–173.

[36] Rehman, Z.-U., Khan, S. N., Muhammad, K., Lee, J. W., Lv, Z., Baik, S. W., Shah, P. A., Awan, K., & Mehmood, I. (2018). Machine learning assisted signature and heuristicbased detection of malwares in Android devices. Computers and Electrical Engineering, vol. 69, pp. 828–841.

[37] W. Huang and J. W. Stokes (2016). MtNet: A multi-task neural network for dynamic malware classification. Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment vol. 96, pp. 236–211.

[38] Y. Ye, L. Chen, S. Hou, W. Hardy, and X. Li (2018). DeepAM: A heterogeneous deep learning framework for intelligent malware detection. Knowl. Inf. Syst., vol. 54, no. 2, pp. 265–285.

[39] S. Venkatraman, M. Alazab, and R. Vinayakumar (2019). A hybrid deep learning image-based analysis for effective malware detection. J. Inf. Secur. Appl., vol. 47, pp. 377–389.

[40] S. Akarsh, K. Simran, P. Poornachandran, V. K. Menon and K. P. Soman (2019). Deep Learning Framework and Visualization for Malware Classification", 2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS), pp. 1059-1063.

[41] U. Tayyab, F. Khan, M. Durad, A. Khan and Y. Lee (2022). A Survey of the Recent Trends in Deep Learning Based Malware Detection", Electronics (MDPI) Journal of Cybersecurity and Privacy, pp. 800-829.

[42] H. Lallie, L. Shepherd, J. Nurse, A. Erola, G. Epiphaniou, C. Maple, et al (2021). Cyber Security in the Age of COVID-19: A Timeline and Analysis of Cyber-crime and Cyber-attacks During the Pandemic. Computers & Security, vol. 105, pp. 1-20.

[43] Bozkir, A.S.; Cankaya, A.O.; Aydos, M. (2019). Utilization and comparision of convolutional neural networks in malware recognition. Proceedings of the 2019 27th Signal Processing and Communications Applications Conference (SIU), Sivas, Turkey, pp. 1–4.

[44] Patil, S.; Varadarajan, V.; Walimbe, D.; Gulechha, S.; Shenoy, S.; Raina, A.; Kotecha, K. (2021). Improving the Robustness of AI-Based Malware Detection Using Adversarial Machine Learning. Algorithms, vol. 14, pp. 297.

[45] Hammad, B.T.; Jamil, N.; Ahmed, I.T.; Zain, Z.M.; Basheer, S. Robust (2022). Malware Family Classification Using Effective Features and Classifiers. Appl. Sci, vol. 12, pp. 7877.

[46] S. Sriram, R. Vinayakumar, V. Sowmya, M. Alazab, and K. P. Soman (2020). Multi-scale learning based malware variant detection using spatial pyramid pooling network,'' in Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS), pp. 740–745.

[47] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An and H. Ye (2018). Significant permission identification for machine-learning-based Android malware detection, IEEE Trans. Ind. Informat., vol. 14, pp. 3216-3225.

[48] Gao, X.; Hu, H.; Shan, C.; Liu, B.; Niu, Z.; Xie, H. (2020). Malware classification for the cloud via semi-supervised transfer learning. J. Inf. Secur. Appl. 2020, vol. 55, pp 102661.

[49] L. Liu, B.-S. Wang, B. Yu and Q.-X. Zhong (2017). Automatic malware classification and new malware detection using machine learning", Frontiers Inf. Technol. Electron. Eng., vol. 18, no. 9, pp. 1336-1347.

[50] W. Han, J. Xue, Y. Wang, F. Zhang, and X. Gao (2021). APTMalInsight: Identify and cognize APT malware based on system call information and ontology knowledge framework, Inf. Sci., vol. 546, pp. 633–664.

[51] Z. Shan and X. Wang (2014). Growing grapes in your computer to defend against malware. IEEE Trans. Inf. Forensics Security, vol. 9, no. 2, pp. 196–207.

[52] Qi, P.; Wang, W.; Zhu, L.; Ng, S.K. (2021). Unsupervised domain adaptation for static malware detection based on gradient boosting trees. Proceedings of the 30th ACM International Conference on Information & Knowledge Management (CIKM '21), Queensland, Australia, pp. 1457–1466.

[53] Verma, V.; Muttoo, S.K.; Singh, V.B. (2002). Multiclass malware classification via first- and second-order texture statistics. Comput. Secur. 2002, vol. 97, pp. 101-123.

[54] Zaheer Masood, Khalid Majeed, Raza Samar, Muhammad Asif Zahoor Raja (2017). Design of Mexican Hat Wavelet neural networks for solving Bratu type nonlinear systems. Neurocomputing Volume 221, 19 January 2017, Pages 1-14

[55] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, and S. Venkatr (2019). Robust intelligent malware detection using deep learning. IEEE Access, vol. 7, pp. 46717–46738, 2019.

[56] J.-S. Luo and D. C.-T. Lo (2017). Binary malware image classification using machine learning with local binary pattern. Proc. IEEE Int. Conf. Big Data (Big Data), Dec. 2017, pp. 4664–4667.

[57] Z. Cui, F. Xue, X. Cai, Y. Cao, G.-G. Wang, and J. Chen (2018). Detection of malicious code variants based on deep learning. IEEE Trans. Ind. Informat., vol. 14, no. 7, pp. 3187–3196, Jul. 2018, doi: 10.1109/tii.2018.2822680.

[58] D. Gibert (2016). Convolutional neural networks for malware classification. M.S. thesis, Univ. Rovira Virgili, Tarragona, Spain, Oct. 2016.

[59] A. Singh, A. Handa, N. Kumar, and S. K. Shukla (2019). Malware classification using image representation. Proc. Int. Symp. Cyber Secur. Cryptogr. Mach. Learn. Cham, Switzerland: Springer, Jun. 2019, pp. 75–92.

[60] X. Ma, S. Guo, H. Li, Z. Pan, J. Qiu, Y. Ding, and F. Chen (2019). How to make attention mechanisms more practical in malware classification. IEEE Access, vol. 7, pp. 155270–155280, 2019, doi: 10.1109/access.2019.2948358.

[61] Ömer Aslan, Abdullah Asim Yilmaz (2021). A New Malware Classification Framework Based on Deep Learning Algorithms. IEEE Digital Object Identifier 10.1109/ACCESS.2021.30895, vol.9.

[62] BENCHADI. DJAFER YAHIA M , BOJAN BATALO, AND KAZUHIRO FUKUI (2023). Efficient Malware Analysis Using Subspace-Based Methods on Representative Image Patterns . IEEE VOLUME 11, 2023

[63] Jaafar M. Alghazo, David M. Feinauer, Sherif E. Abdelhamid (2023). Exploring Automatic Malware Detection through Deep Learning Models . Department of Computer and Information Sciences, Virginia Military Institute, Lexington, Virginia, USA

[64] Olorunjube James Falana , Adesina Simon Sodiya , Saidat Adebukola Onashoga , Biodun Surajudeen Badmus (2022). Mal-Detect: An intelligent visualization approach for malware detection. Science Direct Journal of King Saud University - Computer and Information Sciences Journal of King Saud University - Volume 34, Issue 5, May 2022, Pages 1968-1983

[65] Ahmed Bensaoud, Jugal Kalita (2022). Deep multi-task learning for malware image classification . Science Direct Journal of Information Security and Applications Volume 64, February 2022, 103057