2022/2023

FINAL YEAR PROJECT

Submitted in fulfillment of the requirements for the

ENGINEERING DEGREE FROM THE LEBANESE UNIVERSITY
FACULTY OF ENGINEERING – BRANCH 3

Major: Computer and Telecommunication Engineering

Prepared By:
KATIA HOURANI

EFFICIENT MALWARE DETECTION USING CONVOLUTION NEURAL NETWORK

Supervised by: Mr. Abed ELLATIF SAMHAT
Mr. Hussein KAZEM

Defended on Monday 11 September 2023 in front of the jury:

Dr. Haidar MOKDAD	President
Dr. Abed ELLATIF SAMHAT	Supervisor
Dr. Amjad EL HAJJAR	Member

Acknowledgment

First of all, I want to thank **ALLAH** for all the blessings, for leading me through trials and fears to success, for showing me the light even in times of great darkness, for giving me strength to go so far.

Before I begin this report, I'd like to express my gratitude to *Dr. Hassan Shraim*, the director of the engineering faculty at the Lebanese University, and *Dr. Youssef Harkous* chief of the Electrical Engineering department.

I extend my deepest gratitude to my supervisor, *Dr. Abed Al Latif Samhat,* for his invaluable guidance and unwavering support throughout the course of this project. *Dr. Samhat*'s profound expertise, insightful advice, and dedicated mentorship have been instrumental in shaping the trajectory of this research. His willingness to share knowledge, provide constructive feedback, and offer thoughtful suggestions has significantly enriched my understanding of the subject matter. I am truly fortunate to have had him as my supervisor. His commitment to fostering a productive learning environment, coupled with his exceptional guidance, has motivated me to consistently strive for excellence and push the boundaries of my capabilities.

I would also like to express my heartfelt gratitude to Eng. *Hussein Kazem* for his invaluable contributions to the successful completion of this project. His unwavering support, insightful guidance, and patient instruction have been instrumental in shaping the outcome of this endeavor.

I'd like to thank the jury members,…. for accepting to evaluate my project and for their time and effort.

And finally, I would like to thank my parents, family and friends for their unwavering support and encouragement throughout these past five years, which led us to the completion of this project.

Equally, I extend my appreciation to those who might not have shown immediate encouragement. Your challenges and doubts have been instrumental in shaping my determination and resilience. Your skepticism has pushed me to prove my capabilities and has contributed to the strong individual I've become today.

Katia Hourani

Abstract

Nowadays, detecting and categorizing malware present a remarkable challenge for developers and ethical hackers. The number of malicious file is increasing daily, making it more difficult for human resources to keep up with it, that's why, this project leverages the strength of Convolutional Neural Networks (CNNs) to discover an effective solution for malware detection and classification.

Multiple different neural network approaches have been attempted and documented over the years. In this report we implement and compare the results of different architectures over 2 different malware images datasets: MalIMG and MaleVIS.

The project is structured into two primary phases. The initial phase involves a comprehensive comparison of various established techniques. Subsequently, the second phase focuses on devising a new code that enhances the accuracy of malware detection. Python and its libraries constitute the implementation language of this project, with Google Collab serving as the platform for training and experimentation.

Key words: Malwares, malware detection, malware classification, convolutional neural network, accuracy, image processing, grayscale images.

Table Of Contents

List Of Tables

List Of Figures

List Of Symbols And Abbreviations

CNN	Convolutional Neural Network
DL	Deep Learning
AI	Artificial Intelligence
ML	Machine Learning
FC	Fully Connected Layer
ReLU	Rectified Linear Unit
RGB	Red, Green, Blue
SLP	Single-Layer Perceptron
MLP	Multi-Layer Perceptron
NN	Neural Network

Chapter 1: General Introduction

1. Introduction:

In the digital landscape of today, as technology advances and our dependence on computers and networks deepens, the threat of malware emerges as a significant concern. Malicious software, or malwares, present a modern-day challenge that disrupts the smooth functioning of digital ecosystems and compromises the integrity of data and systems.

In parallel with efforts to combat various challenges, the domain of cybersecurity struggles with the ever-evolving landscape of malware. Just as detecting and diagnosing complex issues is pivotal in different fields, the detection and classification of malware play a central role in safeguarding digital environments.

Traditional methods have been employed to detect and mitigate malware threats. However, with the rise of deep learning – characterized by neural network architectures with multiple layers – new possibilities have emerged.

This pursuit of enhancing cybersecurity through deep learning mirrors the exploration of advanced solutions in various domains.

Much like how advancements in various fields have transformed landscapes, the application of deep learning and neural networks holds the promise of revolutionizing our approach to malware detection and classification. This journey into the depths of advanced technology, guided by lessons from diverse domains, aims to reinforce our digital world against the pervasive and evolving threat of malware.

2. Objective:

The objective of this project is to efficiently detect malwares and classify them correctly using CNN models. The main contents of this work are as follows:

i. Review the methods used for malware detection,
ii. Analyze and compare the performance of each model in detecting malwares in both datasets,
iii. Studying the impact of modifications on the performance of these models,

iv. Finally, finding a novel model to have an optimal solution in detecting and classifying malwares.

3. Report Outline:

The remainder of this report is divided into five main chapter.

Chapter 2 presents a background about convolutional neural network; this chapter contains all the tools necessary to understand the rest of the report.

Chapter 3 present a background about malwares, their types and way of analyzing them, in addition to serval related works in order to go deeply into the main objective of this project.

Chapter 4 illustrates four different models presented in four different papers, with their performance over two datasets, MalIMG and MaleVIS.

Chapter 5 presents many modifications that we have applied over one of the models illustrated in chapter 4 with the lowest accuracy, as well as the impact of a combination of these modifications over the performance of the model in order to be able to optimize the classification and detection of malwares

Finally, chapter 6 concludes the report with a summary of our work and presents the outlines avenues for future work.

Chapter 2: Convolutional Neural Network

1. Introduction:

Deep learning, a branch of machine learning inspired by the brain's structure, has demonstrated remarkable performance in recent years. One of its prominent techniques is the Convolutional Neural Network (CNN), designed for image recognition tasks. CNNs have become a powerful tool extensively employed in image classification due to their hierarchical structure and efficient feature extraction capabilities.

In CNNs, images are initially converted into matrix format to be processed by computers. The system then learns to distinguish between different images and labels by analyzing the differences in the matrices during the training phase. Once trained, the CNN can make accurate predictions for new images based on these learned patterns.

By leveraging these operations effectively, CNNs achieve impressive results in image classification tasks, making them a dynamic and widely-used model in the field of computer vision.

2. Deep Learning:

Deep learning is a subfield of machine learning, while both fall under the broad category of artificial intelligence. With the aid of a neural network and deep learning, machines are now capable of making accurate decisions on their own.

Malware, with its insidious ability to infiltrate systems and wreak havoc, remains a significant threat in the digital realm. Traditional cybersecurity approaches often struggle to keep pace with the ever-evolving landscape of malicious software. As a result, researchers and practitioners have turned their attention to the potential of deep learning for malware detection.

Deep learning, a specialized branch of machine learning, aims to extract intricate patterns and representations from data using hierarchical architectures. This paradigm has captured significant attention in recent years for its remarkable performance in various domains, including cybersecurity.

The allure of deep learning for malware detection arises from several factors:

- ***Enhanced Processing Power:*** The advent of powerful Graphics Processing Units (GPUs) has significantly accelerated deep learning computations, making it feasible to analyze vast amounts of data quickly.
- ***Cost-Effectiveness:*** The reduced cost of computing hardware has made deep learning technology more accessible to researchers and organizations.
- ***Algorithmic Advancements:*** Continuous research and innovation in machine learning algorithms have led to improved deep learning models, offering higher accuracy and more robust malware detection capabilities.

By leveraging the power of deep learning, cybersecurity experts aim to create dynamic and intelligent systems that can autonomously identify and thwart malware threats. This proactive approach holds the potential to transform malware detection, enabling faster response times, enhanced accuracy, and a more resilient defense against the constantly evolving landscape of cyber threats.

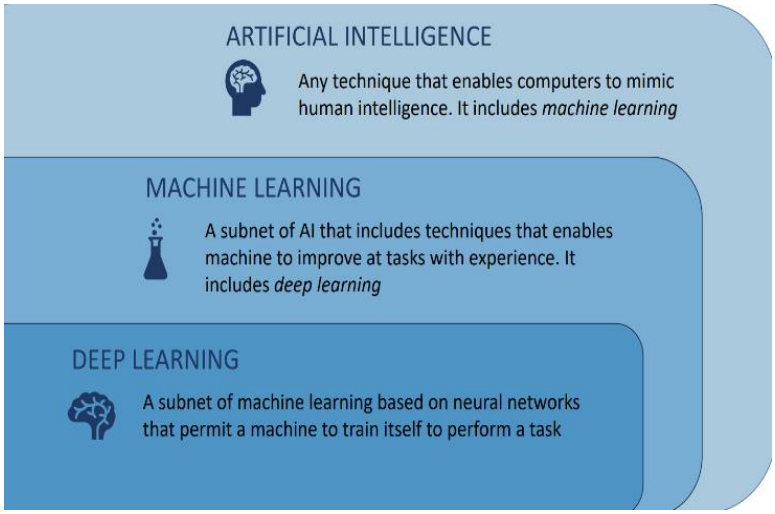

Figure 1: *Difference between AI, ML and DL*

3. Neural Network:

3.1 Definition:

Artificial neural networks are simple imitations of the neuron function in the human brain. An artificial neuron, known as a perceptron, is a mathematical and computational representation of a biological neuron. The formal neuron or perceptron generally has multiple inputs and one output, which correspond, respectively, to the synapses and the emergence cone of the biological neuron

(the starting point of the axon). In a biological neuron, dendrites control the amount of incoming flux, which can be replaced by weights in a perceptron. The core, responsible for producing a signal after stimulation, is referred to as the activation function, which transforms the input signal into another form. *Figure 2* illustrates the two neurons: biological and artificial, respectively, and *Table 1* shows the analogy between the two structures.

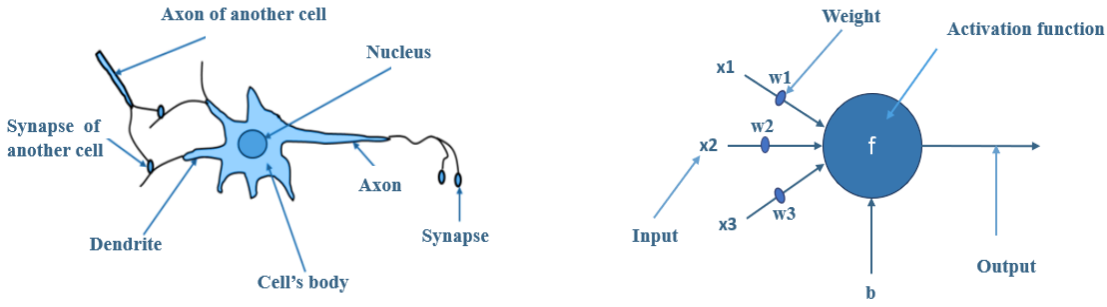

Figure 2: *Biological and artificial neurons*

Table 1: *The analogy between biological and artificial neurons.*

Biological neuron	Artificial neuron
Nucleus	Node
Dendrite	Weights
Synapse	Input
Axon	Output

Neural networks consist of:

- **Input layer:** takes the initial data into the system for further processing. Each input node is associated with a numerical value. It can take any real value.
- **Weights and biases:** Weight parameters represent the strength of the connection between units. Higher is the weight, stronger is the influence of the associated input neuron to decide the output. Bias plays the same as the intercept in a linear equation.
- **Activation function(s):** The activation function determines whether the neuron will fire or not. At its simplest, the activation function is a step function, but based on the scenario, different activation functions can be used.

• **Output :** is the final layer in the neural network where desired predictions are obtained

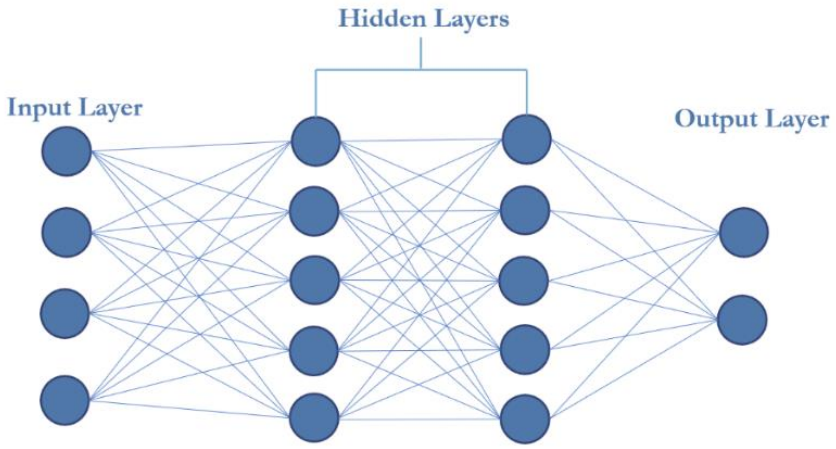

Figure 3: *Neural network architecture*

One of the remarkable features of neural networks is their ability to adapt to varying inputs, allowing them to produce optimal outcomes without the need for constant redesigning of the output criteria. Originating from the field of artificial intelligence, the concept of neural networks is rapidly gaining traction in the development of trading systems and other domains where pattern recognition and adaptation are crucial.

3.2 Single Layer Perceptron Model (SLP):

The simplest of the neural network models, SLP, is viewed as the beginning of artificial intelligence and provided inspiration in developing other neural network models and machine learning models. A single layer perceptron (SLP) is the simplest feed-forward network that can only classify linearly separable cases with a binary target (1, 0). The weights affect the output of the neuron. The aggregate values of the weights multiplied by the input are then summed within the neuron and then fed into an activation function, the standard function being the logistic function:

$$\boldsymbol{Input\ vector} \quad [x_0, x_1, x_2, \dots, x_0]^T \tag{2.01}$$

$$\boldsymbol{weights\ vectors} \quad [w_0, w_1, w_2, \dots, w_n] \tag{2.02}$$

The output of the function is given by:

$$\boldsymbol{y = f(x, w^T) = f\left(\sum_{i=0}^{i=n} x_i w_i\right)} \tag{2.03}$$

Generally, $x_0 = 1\ and\ w_0 = b$ where b is called the bias

The schema and equation of one neuron in a layer become:

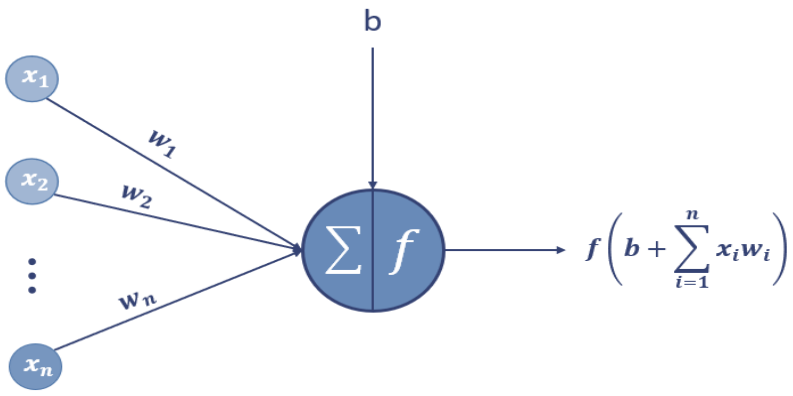

***Figure 4**: Single layer perceptron (SLP)*

When dealing with big datasets, the number of features is large (n is large), and so it is better to use a vector notation for the features and the weights, as follows:

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad w = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} \tag{2.04}$$

For consistency with formulas that will be use later, to multiply x and w, we will use matrix multiplication notation, and therefore we will write:

$$\boldsymbol{w^T x} = (\boldsymbol{w_1} \dots \boldsymbol{w_n}) \begin{pmatrix} \boldsymbol{x_1} \\ \vdots \\ \boldsymbol{x_n} \end{pmatrix} = \boldsymbol{w_1 x_1 + w_2 x_2 + \dots w_n x_n} \tag{2.05}$$

Where $\boldsymbol{w^T}$ indicates the transpose of $\boldsymbol{w}$.

$\boldsymbol{z}$ can then be written with this vector notation as:

$$\boldsymbol{z = w^T x} \tag{2.06}$$

And the neuron output $\hat{\boldsymbol{y}}$ **:**

$$\hat{\boldsymbol{y}} = \boldsymbol{f(z) = f(w^T x + b)} \tag{2.07}$$

In brief,

- $\hat{y}$ → the neuron output

- $f(z)$ → activation function applied to z
- w → weights
- b → bias

3.3 Multilayer Perceptron Model (MLP):

i. Forward Pass:

Very similar to SLP, the multilayer perceptron (MLP) model features multiple layers that are interconnected in such a way that they form a feed-forward neural network. Each neuron in one layer has directed connections to the neurons of a separate layer.

The *forward pass* is when each mini-batch is passed to the network's input layer, which just sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this. The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. [1]

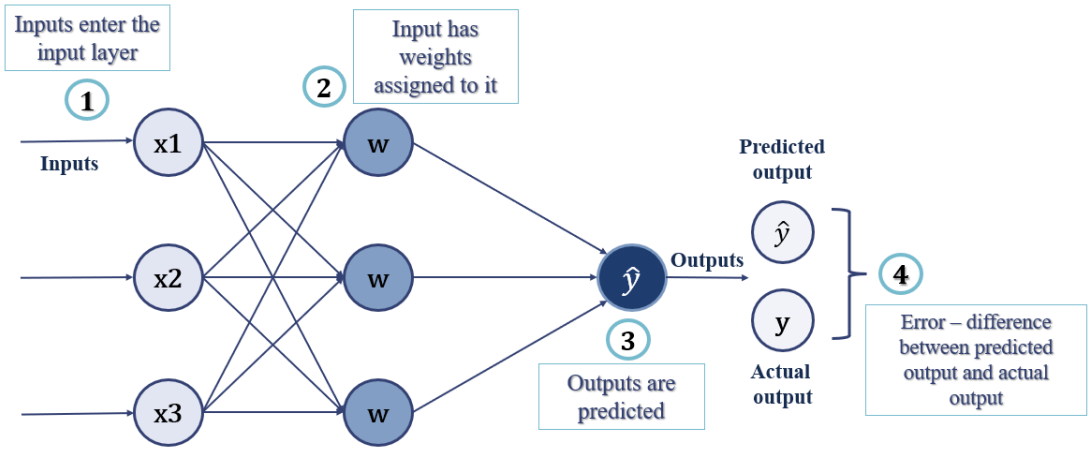

Figure 5: *Forward pass neural network*

To jump from one neuron to this is quite a big step. To build the model, we will have to work with matrix formalism, and therefore, we must get all the matrix dimensions right. First, here are some new notations.

- **L**: number of hidden layers, excluding the input layer, including the output layer
- ***nl*:** number of neurons in layer ***l*.**

Where, by convention, we define $\boldsymbol{n_0 = n_x}$. Each connection between two neurons will have its own weight. Let's indicate the weight between neuron $\boldsymbol{i}$ in layer $\boldsymbol{l}$ and neuron $\boldsymbol{j}$ in layer $\boldsymbol{l-1}$ with $\boldsymbol{w_{ij}}^{[l]}$.

- ***First hidden layer:***

The first two layers of a generic neural network, with the weights of the connections between the first neuron in the input layers and the others in the second layer. The weights and biases between the input layer and layer 1 can be written as a matrix, as follows:

$$\boldsymbol{w}^{[1]} = \begin{pmatrix} \boldsymbol{w_{11}}^{[1]} & \dots & \boldsymbol{w_{1n_x}}^{[1]} \\ \vdots & \ddots & \vdots \\ \boldsymbol{w_{n_1 1}}^{[1]} & \dots & \boldsymbol{w_{n_1 n_x}}^{[1]} \end{pmatrix} \qquad \boldsymbol{b}^{[1]} = \begin{pmatrix} \boldsymbol{b_1}^{[1]} \\ \vdots \\ \vdots \\ \boldsymbol{b_{n1}}^{[1]} \end{pmatrix} \qquad (2.08)$$

This means that our matrix $\boldsymbol{w}^{[1]}$ has dimensions $\boldsymbol{n_1 * n_x}$.

Of course, this can be generalized between any two layers $\boldsymbol{l}$ and $\boldsymbol{l-1}$, meaning that the weight matrix between two adjacent layers $\boldsymbol{l}$ and $\boldsymbol{l-1}$, indicated by $\boldsymbol{w}^{[1]}$, will have dimension $\boldsymbol{n_1 * n_{l-1}}$. By convention, $\boldsymbol{n_0 = n_x}$ is the number of input features (not the number of observations that we indicate with m).

$$\boldsymbol{w}^{[l]} = \begin{pmatrix} \boldsymbol{w_{11}}^{[l]} & \dots & \boldsymbol{w_{1n_{l-1}}}^{[l]} \\ \vdots & \ddots & \vdots \\ \boldsymbol{w_{n_l 1}}^{[l]} & \dots & \boldsymbol{w_{n_l n_{l-1}}}^{[l]} \end{pmatrix} \qquad \boldsymbol{b}^{[l]} = \begin{pmatrix} \boldsymbol{b_1}^{[l]} \\ \vdots \\ \vdots \\ \boldsymbol{b_{nl}}^{[l]} \end{pmatrix} \qquad (2.09)$$

Remember that each neuron that receives inputs will have its own bias, so when considering our two layers, $\boldsymbol{l}$ and $\boldsymbol{l-1}$, we will require $\boldsymbol{n_l}$ different values of b. We will indicate this matrix with $\boldsymbol{b}^{[l]}$, and it will have dimensions $\boldsymbol{n_l * 1}$.

- ***Output Of Neurons:***

Now let's start considering the output of our neurons. First, we will consider the neuron of the first layer (remember that the input layer is by definition layer 0). We assume its output with $\boldsymbol{A}^{[1]}$ and assume that all neurons in layer $\boldsymbol{l}$ use the same activation function, which will be indicated by $\boldsymbol{g}^{[1]}$. And by that we obtain:

$$A^{[l]} = g^{[1]}\left(Z^{[1]}\right) = g^{[1]}\left(\sum_{i=0}^{i=n_x} w_{ij}{}^{[1]} x_i + b_j{}^{[1]}\right) \tag{2.10}$$

And now to the matrix notations:

$$Z^{[1]} = w^{[1]}x + b^{[1]} \tag{2.11}$$

$$Z^{[1]} = \begin{pmatrix} z_1{}^{[1]} \\ \vdots \\ \vdots \\ z_{n1}{}^{[1]} \end{pmatrix} \tag{2.12}$$

$$A^{[1]} = g^{[1]}\left(Z^{[1]}\right) = \begin{pmatrix} a_1{}^{[1]} \\ \vdots \\ \vdots \\ a_{n1}{}^{[1]} \end{pmatrix} \tag{2.13}$$

Where $Z^{[1]}$ will have dimensions of $n_1 * 1$.

We assume next that all neurons in layer l will use the same activation function that we will indicate with $g^{[l]}$. We can easily generalize the equation X for layer l.

$$Z^{[l]} = w^{[l]}A^{[l-1]} + b^{[l]} \tag{2.14}$$

$$Z^{[l]} = \begin{pmatrix} z_1{}^{[l]} \\ \vdots \\ \vdots \\ z_{nl}{}^{[l]} \end{pmatrix} \tag{2.15}$$

Our output matrix form will be:

$$A^{[l]} = g^{[l]}\left(Z^{[l]}\right) = \begin{pmatrix} a_1{}^{[l]} \\ \vdots \\ \vdots \\ a_{nl}{}^{[l]} \end{pmatrix} \tag{2.16}$$

$$\hat{Y} = A^{[L]} \tag{2.17}$$

A summary of the dimensions of all the matrices we have described so far:

- $w^{[l]}$ have dimension of $n_l * n_{l-1}$
- $b^{[l]}$ have dimension of $n_l * 1$

- $Z^{[l-1]}$ have dimension of $n_{l-1} * 1$
- $Z^{[l]}$ have dimension of $n_l * 1$
- $A^{[l]}$ have dimension of $n_l * 1$

But it is always important to remember that the value of a neural network is completely dependent on the quality of its training. Without abundant, diverse training data and an effective training procedure, the network will never "learn" how to classify input samples.

Multilayer Perceptron falls under the category of feedforward algorithms, because inputs are combined with the initial weights in a weighted sum and subjected to the activation function, just like in the Perceptron.

But the difference is that each linear combination is propagated to the next layer. Each layer is feeding the next one with the result of its computation and internal representation of the data. This goes all the way through the hidden layers to the output layer.

However, if the algorithm only computed the weighted sum in each neuron, propagated results to the output layer, and stopped there, it wouldn't be able to "learn" the weights that minimize the cost function. If the algorithm only computed one iteration, there would be no actual learning.

ii. Backward Pass:

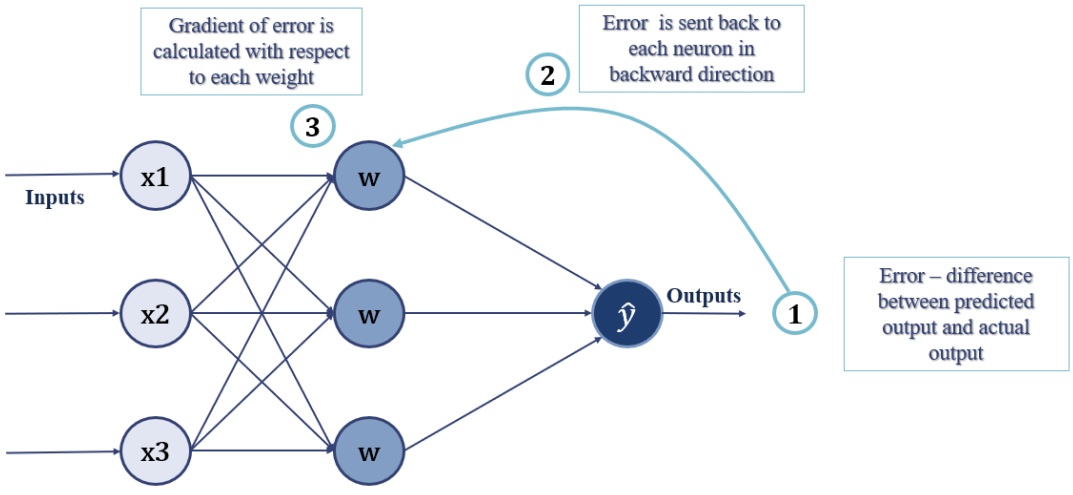

***Figure 6**: Backpropagation*

This is where *Backpropagation* comes into play. Backpropagation [1] [2] [3] in neural network is a short form for "backward propagation of errors". It is a standard method of training artificial

neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network.

The *Back propagation* algorithm in neural network computes the gradient of the loss function for a single weight by the chain rule. It efficiently computes one layer at a time, unlike a native direct computation. It computes the gradient, but it does not define how the gradient is used. Figure x below is a representation of how Backpropagation algorithm works:

1. Inputs X arrive through the preconnected path.

2. Input is modeled using real weights W. Usually, the initial weights are randomly selected.

3. Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.

4. Calculate the error in the outputs: Error= Actual Output – Desired Output

5. Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased. The process is repeated until the desired output is achieved.

4. Convolutional Neural Network (CNN):

4.1 Definition:

Convolutional Neural Networks (CNNs) have emerged as a revolutionary deep learning architecture that has significantly advanced the field of computer vision and image processing. CNNs are designed to automatically learn and extract hierarchical features from visual data, making them highly effective for tasks such as image classification, object detection, image segmentation, and more.

The fundamental building blocks of CNNs are layers, each with a specific role in the feature extraction process. These layers work cohesively to process the input data and gradually learn representations that capture essential patterns and structures present in the images. The hierarchical nature of these layers enables CNNs to learn complex features from raw pixel values, making them exceptionally adept at understanding the visual content of images.

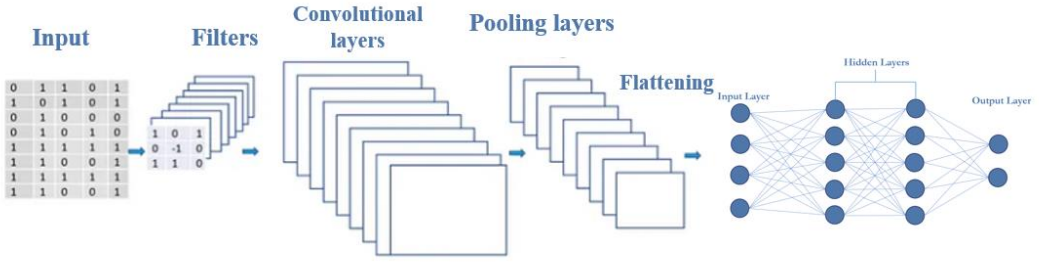

Figure 7: *Convolutional Neural Network (CNN)*

4.2 CNN Layers:

i. Convolutional Layer:

Convolutional layers also called "feature extractor layer " is the main part of CNN model.

First, a part of the input image is connected to the convolutional layer to perform the convolutional operation and then calculate the dot product between the respective field and the filter. The result of this operation is a single integer of the output volume[2]. Then we slide the filter over the next respective field of the same input image by a stride and do the same operation again. The figure below illustrates this operation.

It contains:

- **Filters:** the filters are essentially the neurons of the layer. They have both weighted input and generate an output value like a neuron.
- **Feature map:** it is the output of one filter applied to the previous layer. A given filter is drawn across the entire previous layer, moves one pixel at a time. Each position results in an activation of the neuron and the output are collected in a feature map.

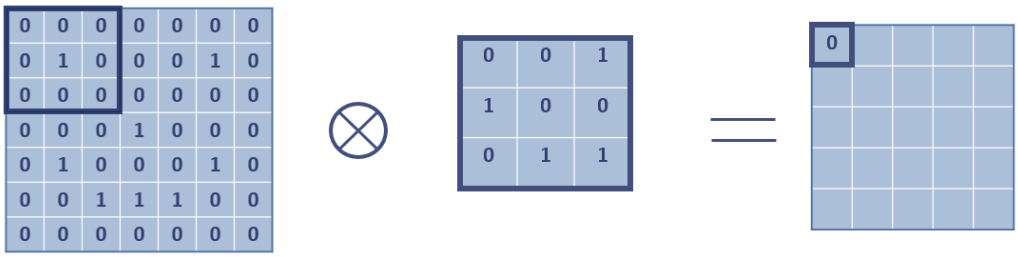

Figure 8: *First filter in convolutional layer*

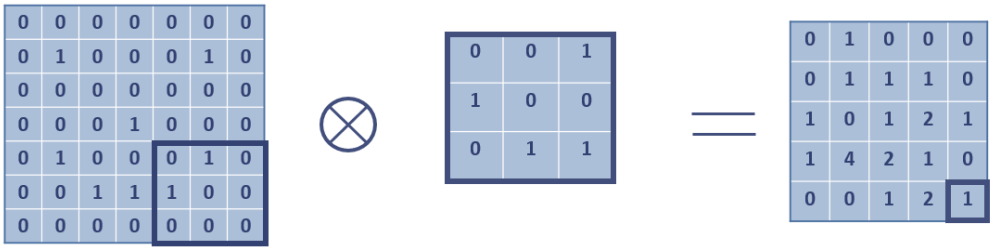

Figure 9: *Last filter in convolutional layer.*

ii. Pooling Layer:

Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network. The pooling layer summarizes the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarized features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations in the position of the features in the input image. There are different types of pooling layers:

- ***Max Pooling:***

Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.

- ***Min Pooling:***

In this type of pooling, the summary of the features in a region is represented by the minimum value in that region. It is mostly used when the image has a light background since min pooling will select darker pixels.

- ***Average Pooling:***

Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.

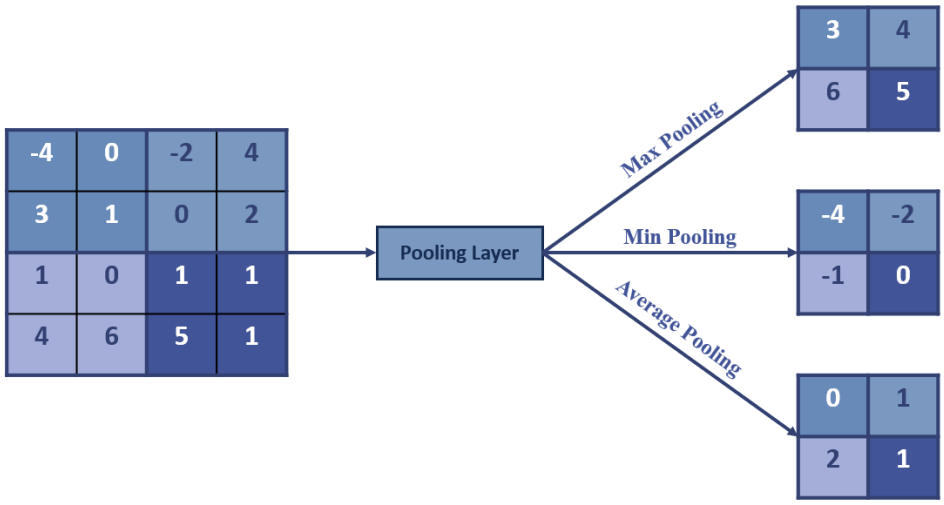

Figure 10: *Pooling layer*

iii. Flatten Layer:

It converts multidimensional output of a previous layer to a single dimensional vector. It is usually used before the fully connected layer in order to prepare the image to be classified.

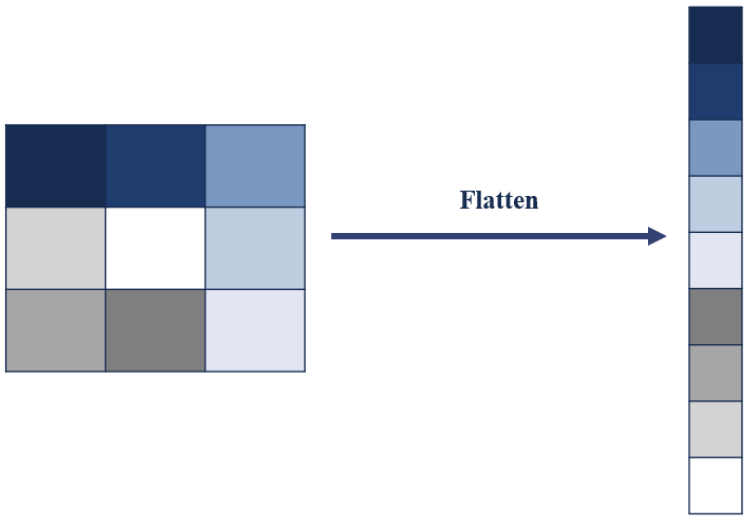

Figure 11: *Flatten layer*

iv. Fully Connected Layer:

Fully connected layers are the normal flat feedforward neural network layer. These layers may have a nonlinear activation function in order to output probabilities of class predictions. Fully

connected layers are used at the end of the network after feature extraction and consolidation have been performed by the convolutional and pooling layers. They are used to create final nonlinear combinations of features and for making predictions by the network.

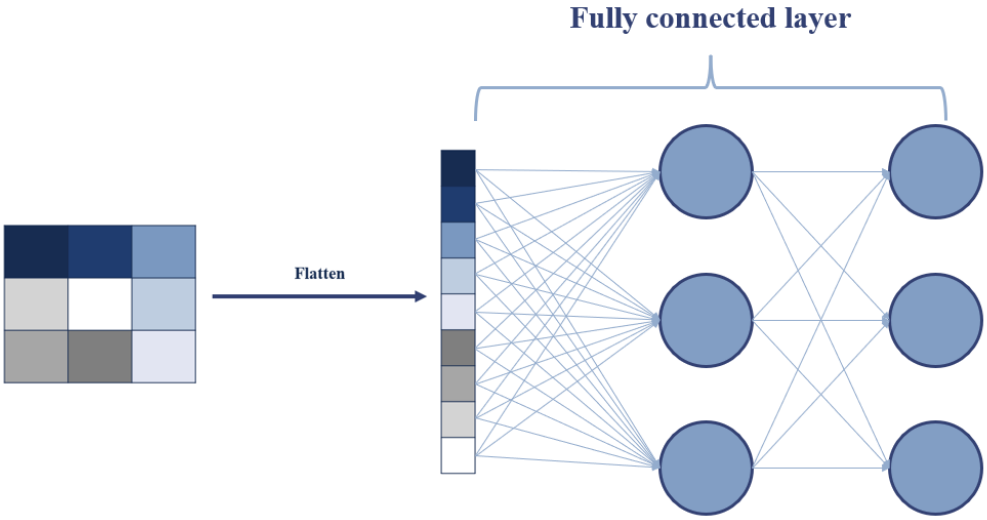

Figure 12: *Fully connected layer*

4.3 Activation Functions:

The goal of activation is to change the signal in a way that allows complicated transformations between the inputs to produce an output value. When the activation function is linear, the input does not change much from the output [1]. Therefore, the activation function must be nonlinear. Nonlinear functions are used to separate nonlinearly separable data. Below is explanation of the most popular activation functions.

i. Sigmoid Function:

The first purpose of sigmoid function is to reduce the input value and make it between 0 and 1. In addition to expressing the value as a probability, if the input value is a very large positive number, the function will convert that value into a probability of 1. Conversely, if the input value is a very large negative number, the function will convert that value into a probability of 0. On the other hand, the curve equation is such that, only small values really influence the variation of the output values.

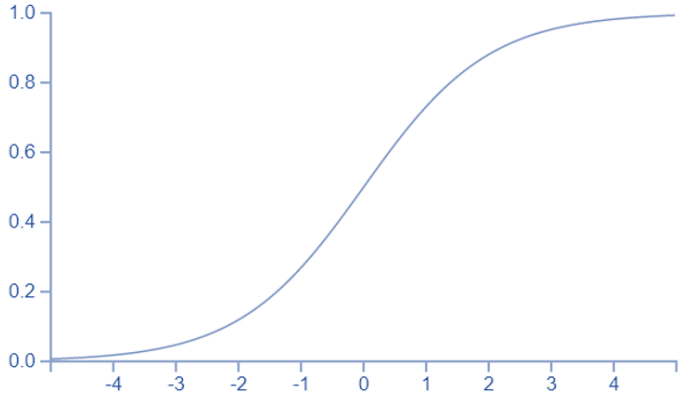

Figure 13: *Sigmoid activation function*

ii. Hyperbolic Tangent Tanh:

This function resembles the Sigmoid function, the difference is that the Tanh function produces a result between -1 and 1. The Tanh function is in general terms preferable to the Sigmoid function because it is centered on zero. Large negative inputs tend to -1 and large positive inputs tend to 1. In addition to that, the Tanh function has the same disadvantages as the sigmoid function.

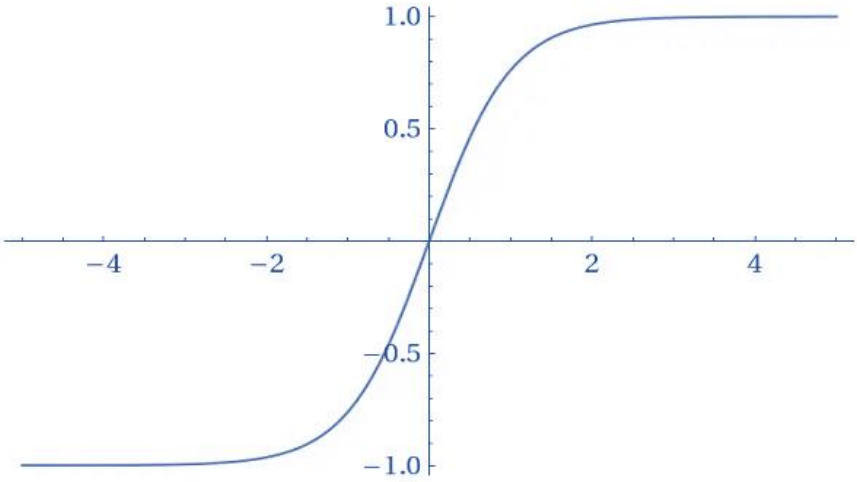

Figure 14: *Tanh activation function*

iii. Rectified Linear Unit (ReLU):

ReLU (Linear Rectification Unit) function solves the saturation problem of the two previous functions (Sigmoid and Tanh). This function is the most used. This activation function greatly increases the convergence of the network. But the ReLU function is not perfect. If the input value

is negative, the neuron remains inactive, so the weights are not updated and the network does not learn.

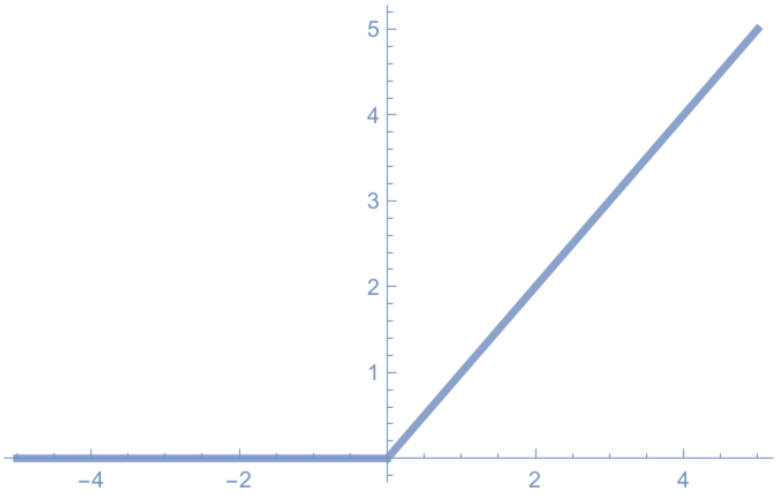

Figure 15: *ReLU activation function*

iv. Softmax:

Many multilayer neural networks end with a penultimate layer which generates real scores that are not conveniently scaled and can be difficult to use. Hence the softmax function is utilized to convert theses scores to a normalized probability distribution. Input values can be positive, negative, zero or greater than one. Softmax transforms these values into values between 0 and 1, so that they can be interpreted as probabilities. Softmax is defined as:

$$\boldsymbol{\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \; for \; i = 1, \dots, k \; and \; z = (z_1, \dots, z_k) \epsilon \; R} \qquad (2.18)$$

where N is the number of classes, $\boldsymbol{z}$ is the input vector, and $\boldsymbol{\sigma(z)_i}$ is the output class probability. This equation shows that the softmax function transforms a vector of k real values into a vector of k real values summing to 1. By softmax, if one of the inputs is small or negative, its probability will be small; otherwise, it will be large, but it will always remain between 0 and 1.

4.4 Performance Metrics:

i. Accuracy:

Accuracy is a metric that generally describes how the model performs across all classes. It is useful when all classes are of equal importance. It is calculated as the ratio between the number of correct predictions to the total number of predictions.

Accuracy is the fraction of predictions our model got right. Formally, accuracy has the following definition:

$$\boldsymbol{Accuracy = \frac{Number\ Of\ Correct\ Prediction}{Total\ Number\ Of\ Predictions}} \quad (2.19)$$

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

$$\boldsymbol{Accuracy = \frac{TP + TN}{TP + TN + FP + FN}} \quad (2.20)$$

Where:

TP = True Positives TN = True Negatives

FP = False Positives FN = False Negatives

ii. Loss Function:

At its core, a loss function is incredibly simple: It's a method of evaluating how well your algorithm models your dataset. If your predictions are totally off, your loss function will output a higher number. If they're pretty good, it'll output a lower number. As you change pieces of your algorithm to try and improve your model, your loss function will tell you if you're getting anywhere. Loss functions are related to model accuracy. In our case we have used *categorial cross entropy.*

What is categorial cross-entropy?

Categorical Cross Entropy is also known as Softmax Loss. It's a softmax activation plus a Cross-Entropy loss used for multiclass classification. Using this loss, we can train a Convolutional Neural Network to output a probability over the N classes for each image.

In multiclass classification, the raw outputs of the neural network are passed through the softmax activation, which then outputs a vector of predicted probabilities over the input classes.

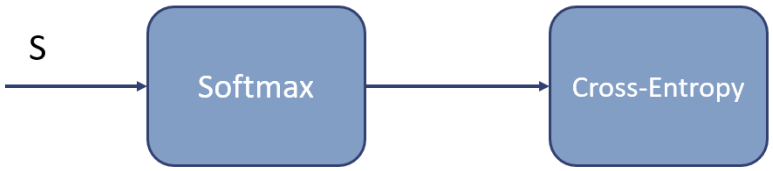

Figure 16: *Categorial cross-entropy*

iii. Learning Curve:

A learning curve is a plot of model learning performance over experience or time. They are a widely used diagnostic tool in machine learning for algorithms that learn from a training dataset incrementally. The model can be evaluated on the training dataset and on a holdout validation dataset after each update during training and plots of the measured performance can created to show learning curves.

Reviewing learning curves of models during training can be used to diagnose problems with learning, such as an underfit or overfit model, as well as whether the training and validation datasets are suitably representative [4].

The learning curve contains two types of plots, *the train learning curve* it gives an idea of how well the model is learning from the training dataset, and *the validation learning curve* that gives an idea of how well the model is generalizing from the validation dataset .

In our project we examined two types of learning curves, the accuracy and the loss learning curves. The diagnosis of the problems in CNN is more accurate to be detected from the loss learning curve, in the part bellow we will explain how overfit, underfit and good fit is detected from loss learning curves [5].

- ***Underfitting:***

It refers to a model that cannot learn the training dataset, and it is obtained when the model is not able to obtain a sufficiently low error value on the training set.

A plot of learning curves shows underfitting if:

- The training loss remains flat regardless of training.
- The training loss continues to decrease until the end of training.

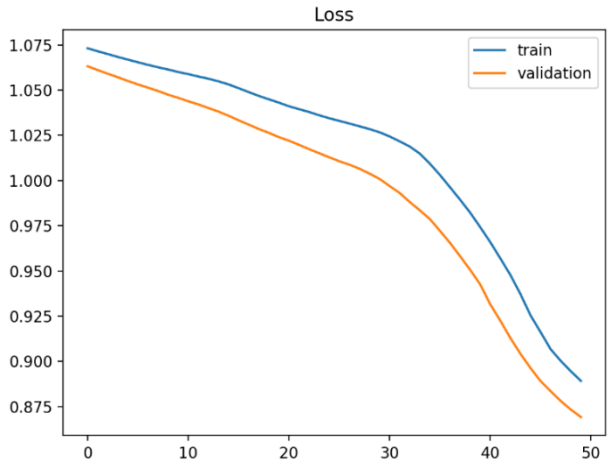

Figure 17: Loss learning curve showing an underfit model

- ***Overfitting:***

Overfitting refers to a model that has learned the training dataset too well, including the statistical noise or random fluctuations in the training dataset. The problem with overfitting, is that the more specialized the model becomes to training data, the less well it is able to generalize to new data, resulting in an increase in generalization error. This increase in generalization error can be measured by the performance of the model on the validation dataset.

A plot of learning curves shows overfitting if:

- The plot of training loss continues to decrease with experience.
- The plot of validation loss decreases to a point and begins increasing again.

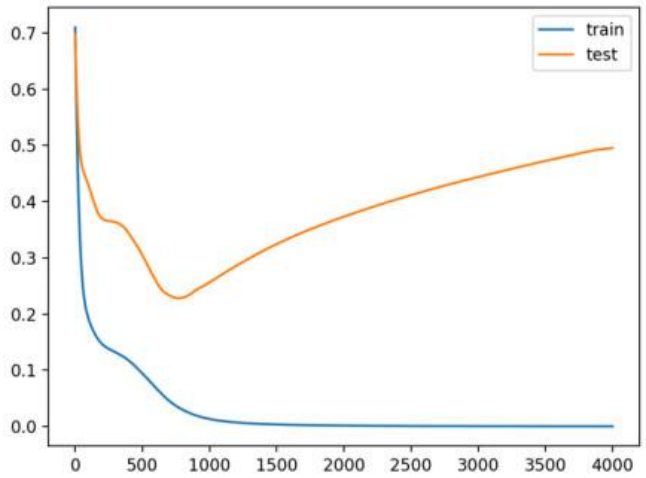

Figure 18: *Loss learning curve showing an overfit model*

- ***Good fit:***

A good fit is the goal of the learning algorithm and exists between an overfit and underfit model. It is identified by a training and validation loss that decreases to a point of stability with a minimal gap between the two final loss values.

A plot of learning curves shows a good fit if:

- The plot of training loss decreases to a point of stability.
- The plot of validation loss decreases to a point of stability and has a small gap with the training loss.

Continued training of a good fit will likely lead to an overfit.

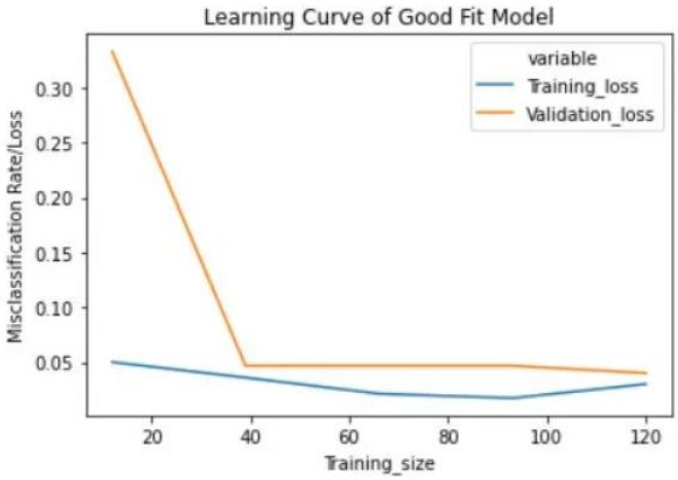

Figure 19: *Loss learning curve showing a good fit model*

5. Conclusion:

In conclusion, Convolutional Neural Networks (CNNs) have revolutionized computer vision by automating the process of learning and recognizing patterns in images. We've explored how CNNs excel in tasks like image classification and object detection, thanks to their unique ability to extract complex visual features.

As we wrap up our exploration of CNNs in computer vision, a new horizon emerges: the application of CNNs to tackle the escalating challenge of malware in our digital landscape. In the upcoming chapter, we'll delve into the exciting potential of CNNs in the realm of cybersecurity. By harnessing their pattern recognition prowess, we aim to develop innovative strategies for more accurate and adaptive malware detection, contributing to the ongoing fight against cyber threats.

Chapter 3: Malwares and Related Works

1. Introduction:

Malware is one of today's major Internet security threats. The amount of new malware has been continuously growing, and its threats are increasing rapidly. Malware, which is slipped into a victim's computer by hackers (attackers) through security vulnerabilities of the operating system or application software, can influence normal operation, collect sensitive information and steal superuser privileges in order to perform malicious actions.

Generally, mainstream malware includes malicious scripts, vulnerability exploits, back doors, worms, trojans, spywares, rootkits, etc., and combinations or variations of the above types as well. Traditionally, most malware detection systems are based on feature vectors, which contain essential. Mainstream malware feature extraction can be divided into two categories, static and dynamic [6], and can be detected using image processing algorithm.

In this chapter we will go deeper into the types of malwares and how we analyze it, we will go through the datasets used in our project and will also extend our explanation into some related work.

2. Types Of Malwares:

We will list some of the most common malware types and will define them:

2.1 Virus:

Possibly the most common type of malware, viruses attach their malicious code to clean code and wait for an unsuspecting user or an automated process to execute them. Like a biological virus, they can spread quickly and widely, causing damage to the core functionality of systems, corrupting files and locking users out of their computers. They are usually contained within an executable file.

2.2 Worms:

Worms get their name from the way they infect systems. Starting from one infected machine, they weave their way through the network, connecting to consecutive machines in order to

continue the spread of infection. This type of malware can infect entire networks of devices very quickly.

2.3 Spyware:

Spyware, as its name suggests, is designed to spy on what a user is doing. Hiding in the background on a computer, this type of malware will collect information without the user knowing, such as credit card details, passwords and other sensitive information.

2.4 Torjans:

Just like Greek soldiers hid in a giant horse to deliver their attack, this type of malware hides within or disguises itself as legitimate software. Acting discretely, it will breach security by creating backdoors that give other malware variants easy access.

2.5 Ransomware:

Also known as scareware, ransomware comes with a heavy price. Able to lockdown networks and lock out users until a ransom is paid, ransomware has targeted some of the biggest organizations in the world today with expensive results.

3. Malware Analysis:

Malware analysis involves mainly two techniques: static analysis and dynamic analysis.

Static analysis consists of examining the code or structure of a program without executing g it. This kind of analysis can confirm whether a file is malicious, provide information about its functionality and can also be used to produce a simple set of signatures. The most common static analysis approaches are:

- ***Finding sequences of characters or strings:*** Searching through the strings of a program is the simplest way to obtain hints about its functionality. For instance, you can find strings related to printed messages, URLs accessed by the program, the location of files modified by the executable and names of common Windows dynamic link libraries (DLLs).
- ***Analysis of the Portable Executable File Format:*** The Portable Executable(PE) file format is used by Windows executables, object code and DLLs. Among the information it includes, the most useful pieces of information are the linked libraries and functions as well as the metadata about the file included in the headers.

- ***Searching for packed/encrypted code:*** Malware writers usually use packing and encryption to make their files more difficult to analyze. Software programs that have been packed or encrypted usually contain very few strings and higher entropy compared to legitimate programs.
- ***Disassembling the program:*** recovering the symbolic representation from the machine code instructions [7].

Dynamic analysis involves executing the program and monitoring its behavior on the system. Unlike static analysis, dynamic analysis allows to observe the actual actions executed by the program. It is typically performed when static analysis has reached a dead end, either due to obfuscation and packing, or by having exhausted the available static analysis techniques. Some techniques are:

- ***Function Call Monitoring:*** The behavior of the program is analyzed by using the traces containing the sequence of functions invoked by the executable under analysis.
- ***Function Parameter Analysis:*** Consists of tracking the values of parameters and function return values.
- ***Information Flow Tracking:*** Analyze how a program processes data and how data is propagated through the system.
- ***Instruction Trace:*** Analysis of the complete sequence of machine instructions executed by the program.

Both methods have their own advantages and disadvantages. On the one hand, static analysis is faster but suffers from code obfuscation, techniques used by malware authors to conceal the malicious purpose of the program. On the other hand, code obfuscation techniques and polymorphic malware fails at dynamic analysis because it analyses the runtime behavior of a program by monitoring it while in execution [7].

4. Visualizing Malware As Grayscale Image:

The main benefit of visualizing a malicious executable as an image is that the different sections of a binary can be easily differentiated. Malware authors usually change a small part of the previously available code to produce new malware. If we represent malware as an image then these small changes can be easily tracked.

A given malware binary file is first read in a vector of 8-bits unsigned integers. Next, the binary value of each component of this vector is converted to its equivalent decimal value (for example the decimal value for [00000000] in binary is [0:black] and for [11111111] is [255:white]) which is then saved in a new decimal vector representative of the malware sample. At last, the resulting decimal vector is reshaped to a 2D matrix and visualized as a grayscale image. Selecting width and height of the 2D matrix (the spatial resolution of the image) mainly depends on the malware binary file size [8].

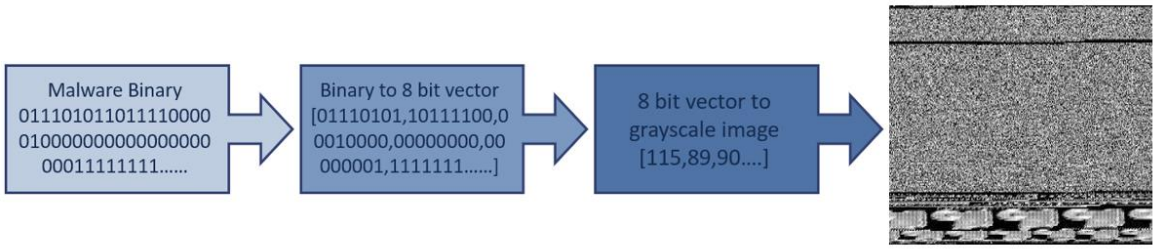

Figure 20: *Overview of malware visualization process*

Figure 21 provides some examples that support this observation

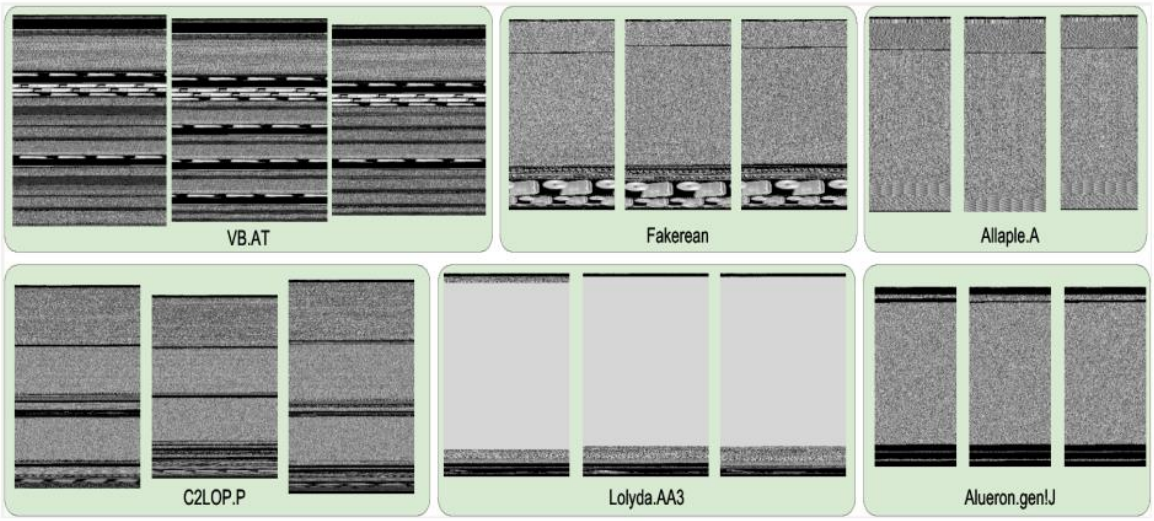

Figure 21: *Sample of malware grayscale images belonging to different families.*

In figure 22 is a sample of a code presented in [10] to transform binary malwares to grayscale images.

```
def convertAndSave(array,name):
    print('Processing '+name)
    if array.shape[1]!=16: #If not hexadecimal
        assert(False)
    b=int((array.shape[0]*16)**(0.5))
    b=2**(int(log(b)/log(2))+1)
    a=int(array.shape[0]*16/b)
    array=array[:a*b//16,:]
    array=np.reshape(array,(a,b))
    im = Image.fromarray(np.uint8(array))
    im.save(root+'\\'+name+'.png', "PNG")
    return im

#Get the list of files
files=os.listdir(root)
print('files : ',files)
#We will process files one by one.
for counter, name in enumerate(files):
        #We only process .bytes files from our folder.
        if '.bytes' != name[-6:]:
            continue
        f=open(root+'/'+name)
        array=[]
        for line in f:
            xx=line.split()
            if len(xx)!=17:
                continue
            array.append([int(i,16) if i!='??' else 0 for i in xx[1:] ])
        plt.imshow(convertAndSave(np.array(array),name))
        del array
        f.close()
```

Figure 22: *Code to convert binary malware to grayscale*

5. Related Work:

Many recent research efforts have focused on the field of malwares due to its serious threats to the privacy of our information and the potential compromise of various company operations. Previous work on malware classification can be broadly categorized into two groups: non-machine learning methods and machine learning-based methods. As the field of AI is constantly evolving, this project aims to enhance some of the latest research models to optimize their performance with different datasets.

Several machine learning algorithms have been proposed for the classification and detection of unknown codes, mainly falling into the categories of known and unknown malware families. In this section, we will discuss a few noteworthy research papers that employed Convolutional Neural Networks (CNNs) to detect and classify malwares. It's worth noting that many of these models will be explained in more detail later in this report.

In paper [10], the objective was to perform multi-classification of malware. *L. Nataraj* utilized CNNs to detect and classify malware, with the twist of transforming the malware from binary code into grayscale images (as depicted in Figure 19) to achieve this goal. This model achieved an accuracy of 95.1%.

In another approach, detailed in [11] by Daniel Gibert, the distinction between static and dynamic malware analysis was explained, along with the revolutionary impact of image processing methods on detecting and classifying malwares. This paper employed two datasets: the MalIMG dataset and the Microsoft Malware Classification Challenge dataset. Additionally, the impact of different input sizes on malware performance was specified.

Mahmoud Kalash, in [12], also employed two datasets—MalIMG and the Microsoft malware dataset. His model achieved an accuracy of 98.5% and 99.97%, respectively. Initially, M. Kalash introduced the contrast between recent non-Machine Learning Methods and Machine Learning Methods to emphasize the significance of Machine Learning in effectively detecting and classifying malwares into families.

In the work presented in [13], a model was implemented by Rajes Kumar and Riaz Ullah Khan to detect malwares and was trained using three types of datasets. These datasets include: one from Vision Research Lab, namely MalIMG, containing 9458 grayscale images extracted from an equal number of malware samples spanning 25 different malware families; the second dataset from the Microsoft Malware Classification Challenge, comprising 10868 binary files representing 9 distinct malware families; and a third benign dataset featuring 3000 diverse benign software samples. The authors comprehensively elucidated their approach, theoretical model, implementation details, and test results. They achieved an accuracy of 98% for the MalIMG dataset and 97.6% for the Microsoft Malware Classification Challenge Dataset.

6. Conclusion:

In this chapter, we introduce the main types of malwares and we also presented the three methods used to detect malwares: static, dynamic and image processing using grayscale images. We also declare and clarify some of the latest related work that applied to some of the most important malware datasets.

In the next chapter, we will explain each model presented in the previous work and we will check how each model will act with two of the best datasets: MalIMG and MaleVIS, in order to specify the model with the less performance and try to improve it in order to have the optimal model.

Chapter 4: Comparative Analysis of CNN Models for Malware Detection

1. Introduction:

Malicious software, malware, are a critical threats for computers and critical information. Traditional malware detection techniques, such as signature-based detection, are becoming increasingly ineffective as malware authors develop new ways to evade detection.

Convolutional neural networks (CNNs) have shown promise for malware detection due to their ability to learn complex patterns in data. In this chapter, we compare the performance of different CNN codes for malware detection from different papers. We evaluate the codes on two datasets of malware, and we will show that the codes accuracy in detecting malware.

We also discuss the limitations of the codes and the challenges of using CNNs for malware detection. We conclude that CNNs are a promising approach for malware detection, but more research is needed to improve the performance of the codes and to address the challenges of using CNNs for this task.

2. Datasets Used:

First let's discuss the datasets used.

This project is applied over two datasets: MalIMG and MaleVIS. We have chosen the first dataset because it was the dataset used in the related work that our research was based on, and the second dataset is chosen because these two datasets are the most important datasets in this field.

2.1 MalIMG Dataset:

This dataset has a total of 9,339 malware samples, represented as grayscale images. Each malware sample belongs to one of 25 malware families or classes.

In our experiments, we randomly selected 80% of malware samples for training and 20% for testing. At the end , we have 7471 malware sample for training and 1868 for testing.

2.2 MaleVIS Dataset:

This dataset has a total of 14,226 evenly distributed malware samples that are represented as RGB images. Each malware sample in the dataset belongs to one of the 25 malware families/classes. In our experiments, we randomly select 70% of malware samples in a family for training and the remaining 30% for testing. At the end, we have 12,803 malware samples for training and 1423 samples for testing.

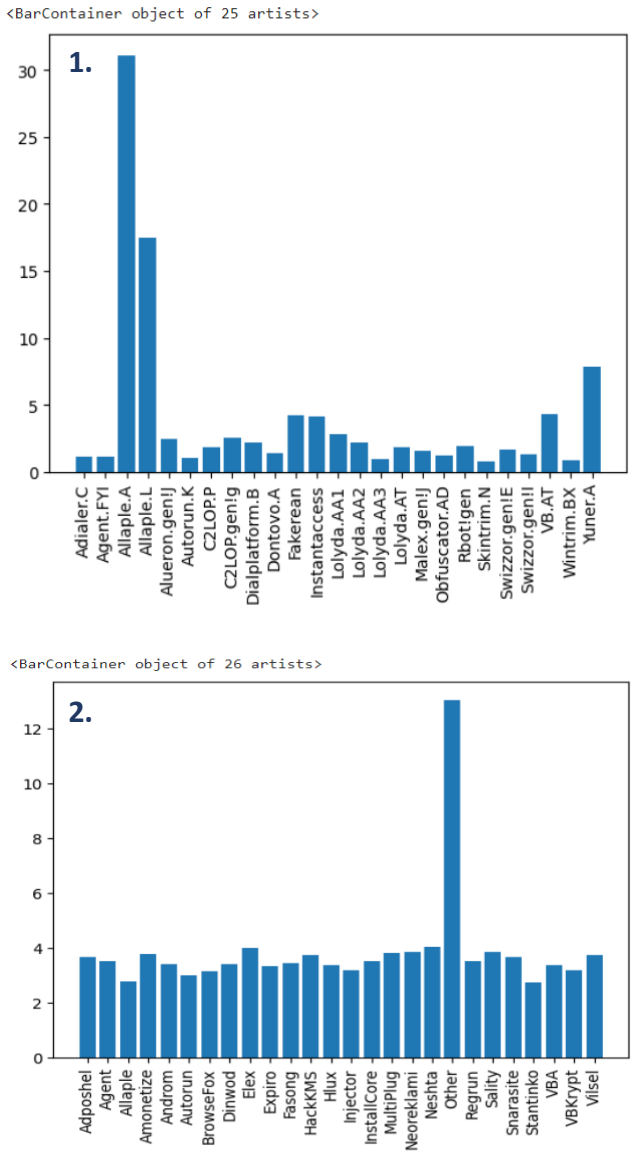

Figure 23: *Distribution of malware families in the 1.MalIMG and 2.MaleVIS dataset*

These datasets contain different families of malwares presented with their types in the two tables below:

Table 2: *Table showing the number, name and types of different families in both datasets*

MALIMG DATASET

Number	Family Name	Type
0	Adialer.C	Dialer
1	Agent.FYI	Backdoor
2	Allaple.A	Worm
3	Allaple.L	Worm
4	Alueron.gen!J	Torjan
5	Autorun.K	Worm:AutoIT
6	C2LOP.P	Torjan
7	C2LOP.gen!g	Torjan
8	Dialplatform.B	Dialer
9	Dontovo.A	TDownloader
10	Fakerean	Roque
11	Instantaccess	Dialer
12	Lolyda.AA1	PWS
13	Lolyda.AA2	PWS
14	Lolyda.AA3	PWS
15	Lolyda.AT	PWS
16	Malex.gen!J	Torjan
17	Obfuscator.AD	TDownloader
18	Rbot!gen	Backdoor
19	Skintrim.N	Torjan
20	Swizzor.gen!E	TDownloader
21	Swizzor.gen!I	TDownloader
22	VB.AT	Worm
23	Wintrim.BX	TDownloader
24	Yuner.A	Worm

MALEVIS DATASET

Number	Family Name	Type
0	Adposhel	Adware
1	Agent	Torjan
2	Allaple	Worm
3	Amonetize	Adware
4	Androm	Backdoor
5	Autorun	Worm
6	BrowseFox	Adware
7	Dinwod	Torjan
8	Elex	Torjan
9	Expiro	Virus
10	Fasong	Worm
11	HackKMS	Torjan
12	Hlux	Worm
13	Injector	Torjan
14	InstallCore	Adware
15	MultiPlug	Adware
16	Neoreklami	Adware
17	Neshta	Virus
18	Other	Torjan
19	Regrun	Backdoor
20	Sality	Virus
21	Snarasite	Torjan
22	Stantinko	Backdoor
23	VBA	Torjan
24	VBKrypt	Torjan
25	Vilsel	Torjan

3. Parameters Chosen:

For sure, training deep learning model is time consuming and requires powerful resources and high computational power, therefore we used google Collab which provides the users with a free GPU memory to fasten the work and achieve good results.

In this project, we intend to combine the models into one Google Collab code, and for that there is some parameters that should be unified for all codes. In the table below, we provide these parameters for each datasets:

Table 3: *Table showing the unified parameters*

	MalIMG	MaleVIS
Input Image Size	**(224,224)**	**(224,224)**
Batch Size	**2500**	**2500**
Dataset split	**20% for testing , 80% for training**	**30% for testing , 70% for training**

4. Codes Comparison:

Our main interest in this project is to find a model that has an optimal solution in detecting and classifying malwares. In order to achieve this goal, we've begun by comparing different model in order to spot the one with the worst performance and try to enhance its performance by monitoring the impact of some modification on the performance

4.1 Description of codes:

i. CNN-1:

This code is presented in [14], it represents the basic CNN model, this model will be labeled CNN-1 in the rest of this report. The main goal of this model is to have a quick and easy way to process malwares and to classify them correctly.

The dataset is represented as grayscale images in order to classify the malware based on their similarity in the grayscale image.

The structure of the model is the following:

- Convolutional Layer: 30 filters , (3*3) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size,
- Convolutional Layer: 15 filters , (3*3) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size,
- DropOut Layer: Dropping 25% of neurons,
- Flatten Layer
- Dense /Fully Connected Layer: 128 neurons, ReLU activation function,
- DropOut Layer: Dropping 50% of neurons,
- Dense /Fully Connected Layer: 50 neurons, Softmax activation function,
- Dense /Fully Connected Layer: N classes neurons, Softmax activation function.

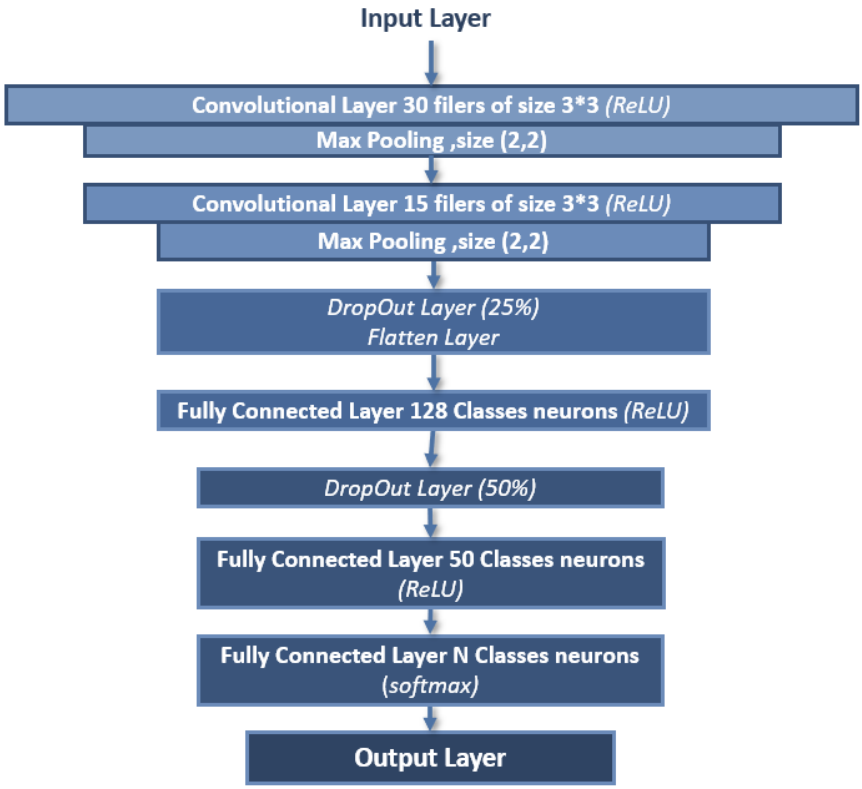

Figure 24: *Representation of CNN-1 model*

ii. CNN-2:

This code is presented in paper [11], this model will be labeled CNN-2 toward this report.

The structure of the model is the following:

- Convolutional Layer: 50 filters , (5*5) kernel size, with ReLU activation function,

- Max Pooling Layer: (2*2) pool size, Stride 1,
- Normalization Layer,
- Convolutional Layer: 70 filters , (5*5*50) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Normalization Layer,
- Convolutional Layer: 70 filters , (5*5*70) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Normalization Layer,
- Flatten Layer
- Dense /Fully Connected Layer: 256 neurons, ReLU activation function,
- Dense /Fully Connected Layer: N classes neurons, Softmax activation function.

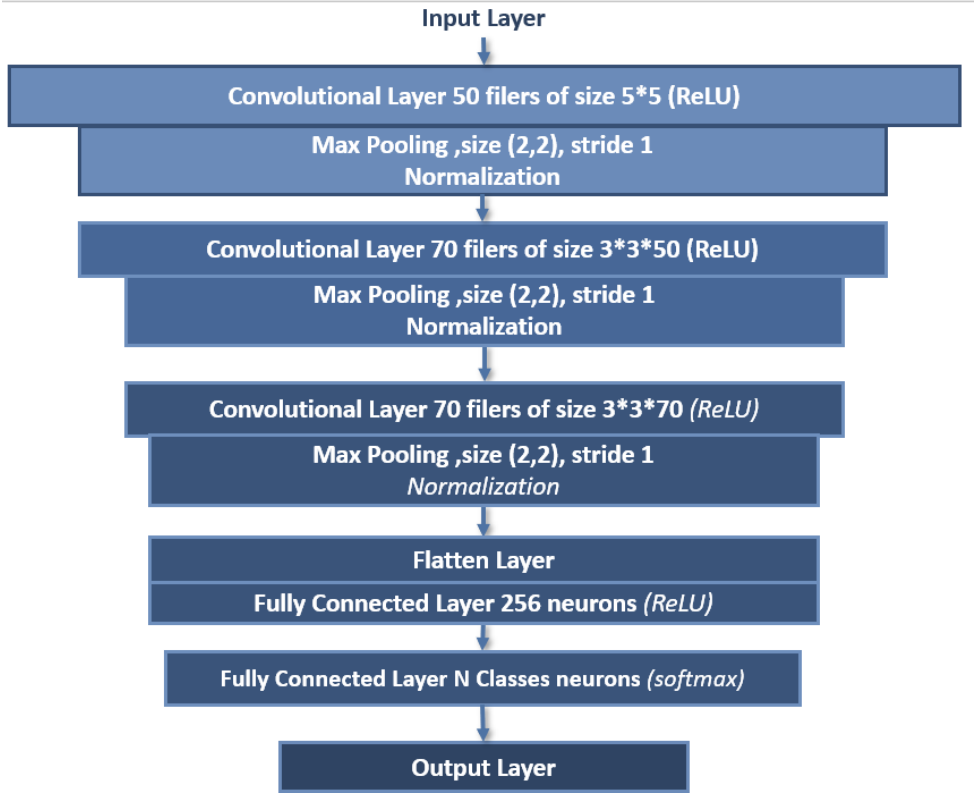

Figure 25: *Representation of CNN-2 model*

iii. CNN-3:

This code is presented in paper [12], this model will be labeled CNN-3 toward this report.

The structure of the model is the following:

- Convolutional Layer: 64 filters , (5*5) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Convolutional Layer: 128 filters , (3*3) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Convolutional Layer: 256 filters , (3*3) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Convolutional Layer: 512 filters , (3*3) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Convolutional Layer: 512 filters , (3*3) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Flatten Layer,
- Dense /Fully Connected Layer: 4096 neurons, ReLU activation function,
- Dense /Fully Connected Layer: 4096 neurons, ReLU activation function,
- Dense /Fully Connected Layer: N classes neurons, Softmax activation function

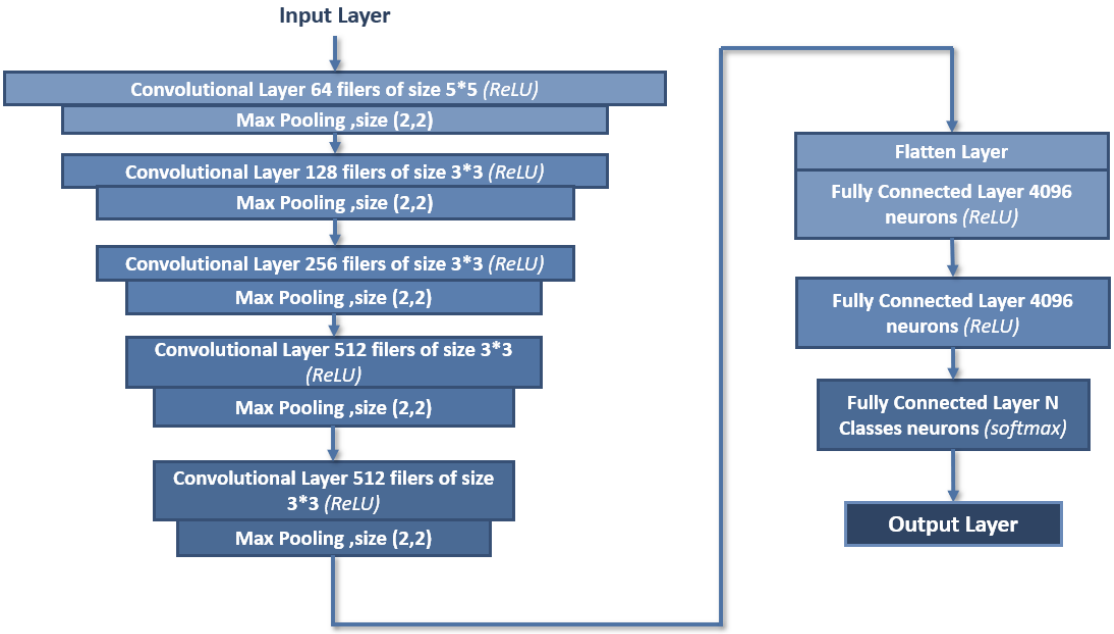

Figure 26: *Representation of CNN-3 model*

iv. CNN-4

This code is presented in paper [13], this model will be labeled CNN-4 toward this report.

The structure of the model is the following:

- Convolutional Layer: 32 filters , (3*3) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Convolutional Layer: 32 filters , (3*3) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Convolutional Layer: 64 filters , (3*3) kernel size, with ReLU activation function,
- Max Pooling Layer: (2*2) pool size, Stride 1,
- Flatten Layer,
- Dense /Fully Connected Layer: 256 neurons, ReLU activation function,
- Dense /Fully Connected Layer: N classes neurons, Softmax activation function

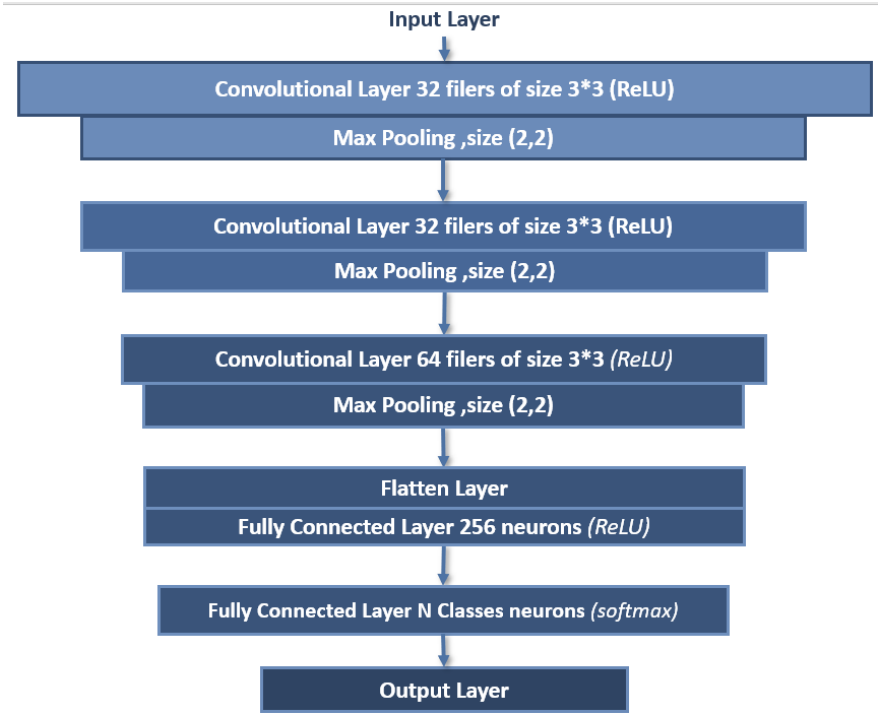

Figure 27: *Representation of CNN-4 model*

4.2 Results of MalIMG:

We will present in this section the results obtained after training and validating the MalIMG dataset with the models mentioned above.

The input image size is (224,224), the training and validation is done by with a batch size of 2500. And we have chosen 20% of the data for validation 80% for testing.

First, in the table below we will present the number of epochs, accuracy and loss of each model:

Table 4: *Number of epochs, accuracy an loss in the four models for MalIMG*

	Number Of Epochs	Accuracy	Loss
CNN-1	100	96.6	0.1802
CNN-2	100	95.8	0.2773
CNN-3	100	96.8	0.1589
CNN-4	100	97.5	0.2252

Next, we will list the accuracy and loss learning curve representing the difference between the variation of testing and validation in the four different models.

- ***CNN-1:***

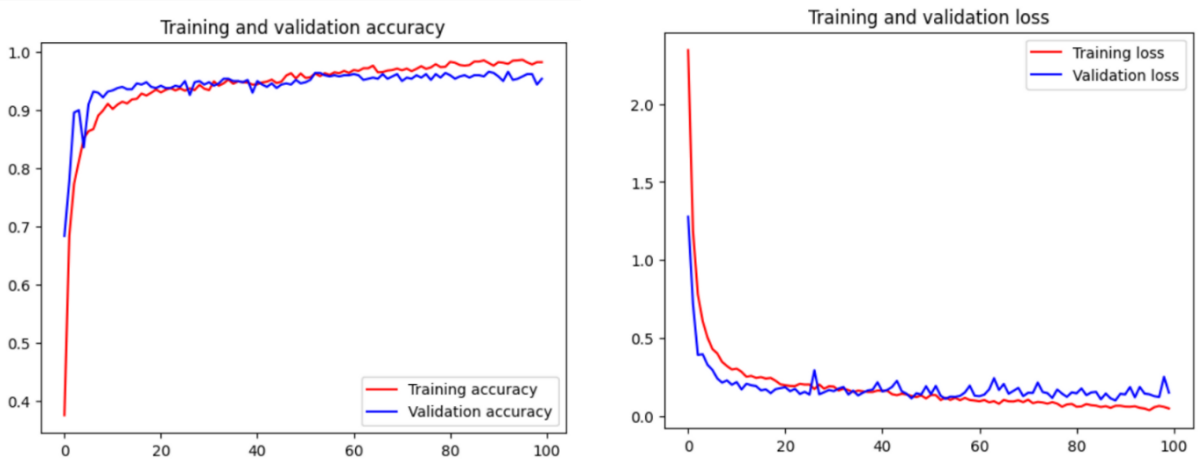

Figure 28: *The accuracy and loss learning curve for MalIMG dataset- CNN-1*

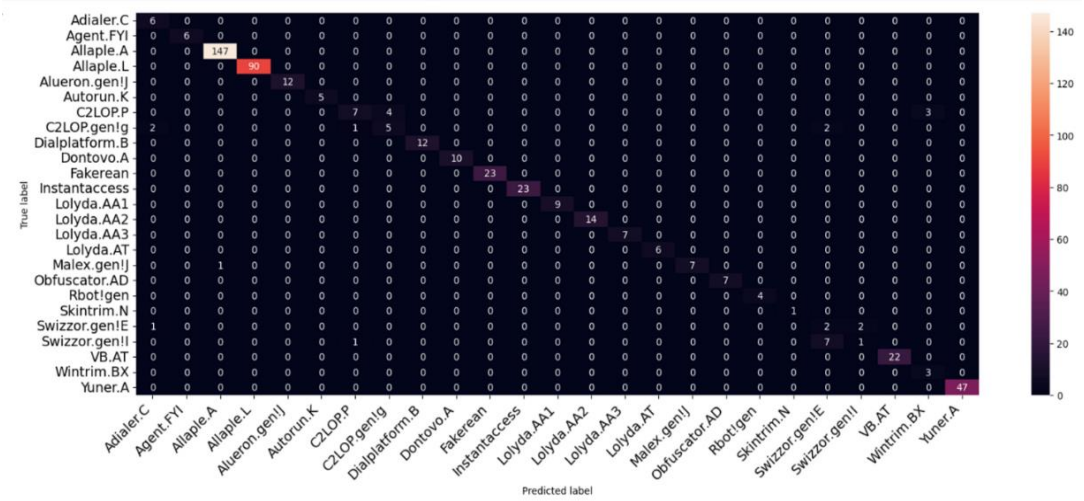

Figure 29: *CNN-1 confusion matrix of MalIMG dataset*

- ***CNN-2:***

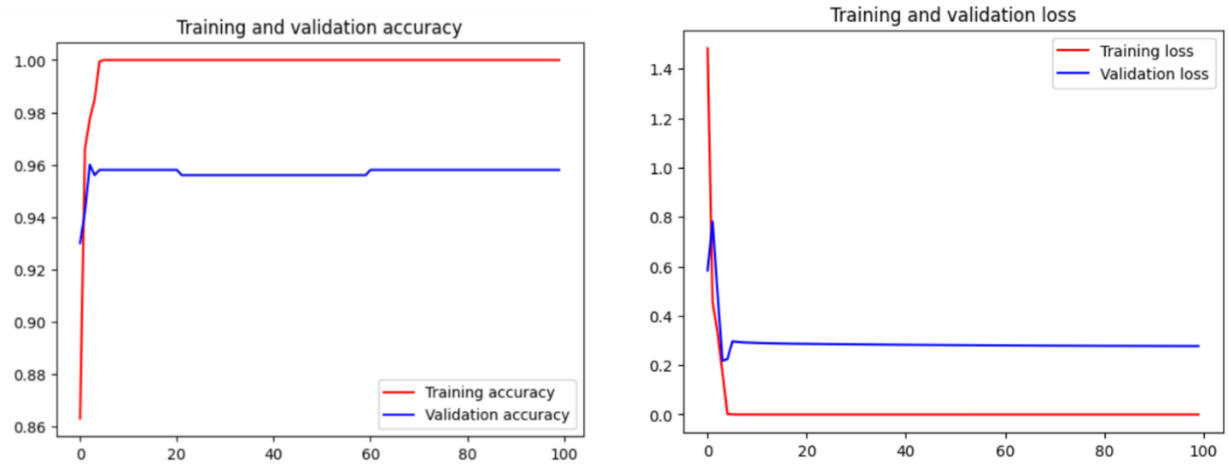

Figure 30: *The accuracy and loss learning curve for MalIMG dataset- CNN-2*

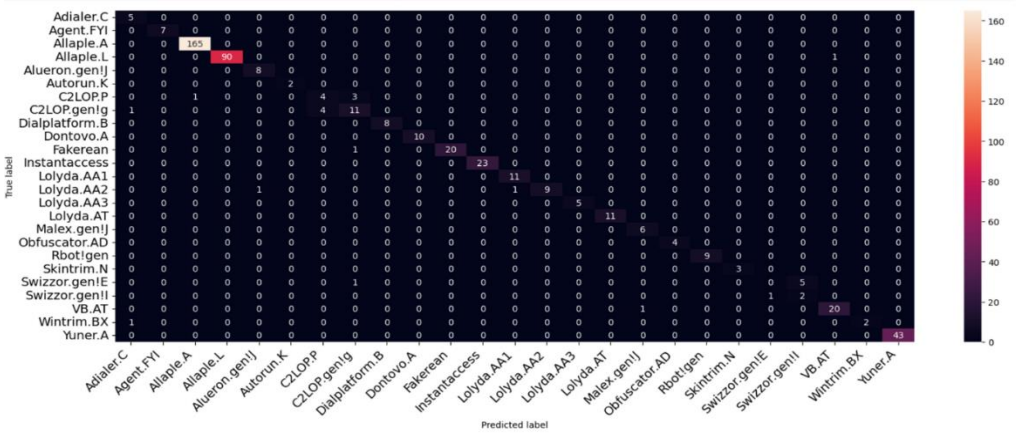

Figure 31: *CNN-2 confusion matrix of MalIMG dataset*

- ***CNN-3:***

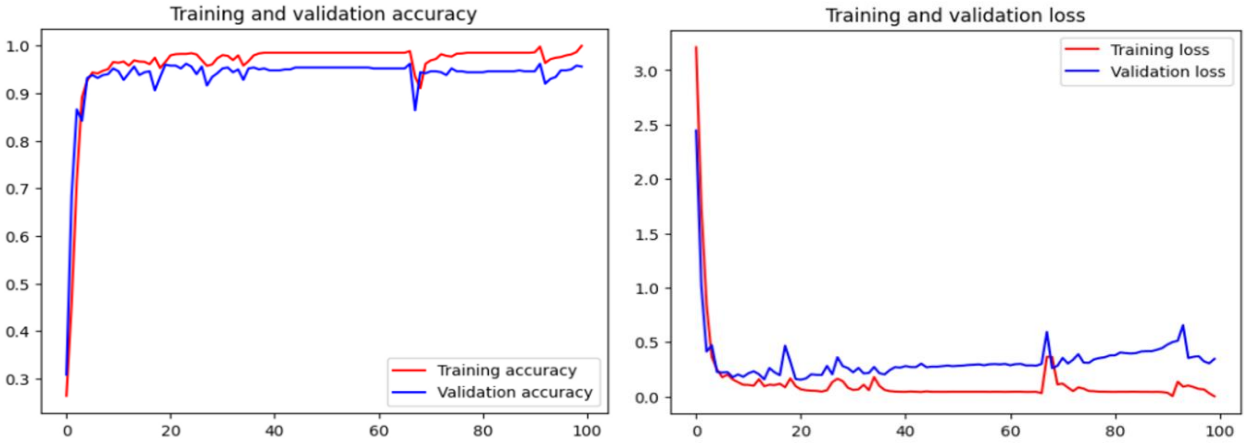

Figure 32: *The accuracy and loss learning curve for MalIMG dataset- CNN-3*

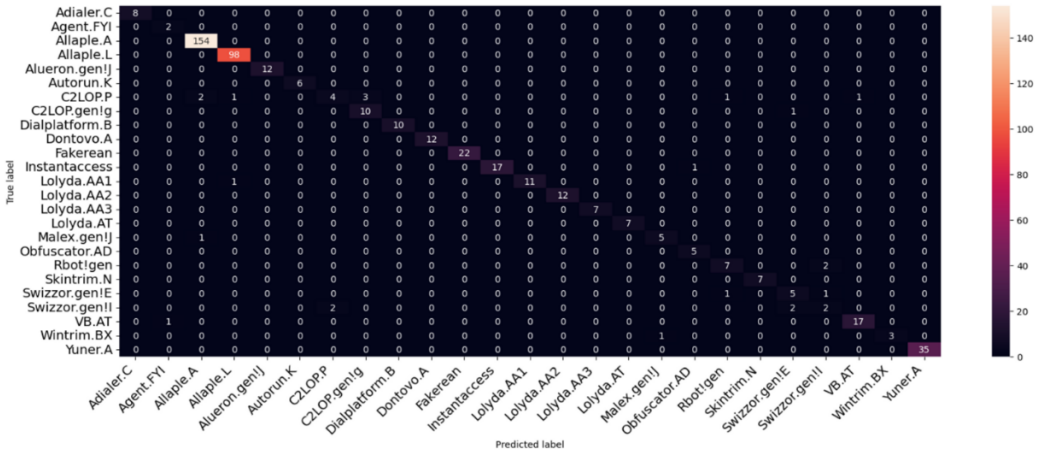

Figure 33*: CNN-3 confusion matrix of MalIMG dataset*

- ***CNN-4:***

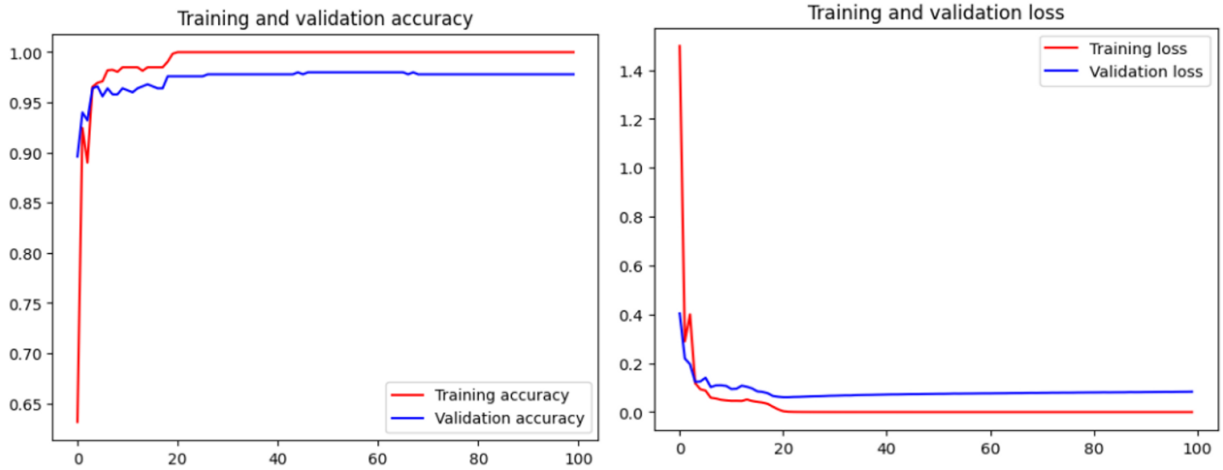

Figure 34: *The accuracy and loss learning curve for MalIMG dataset- CNN-4*

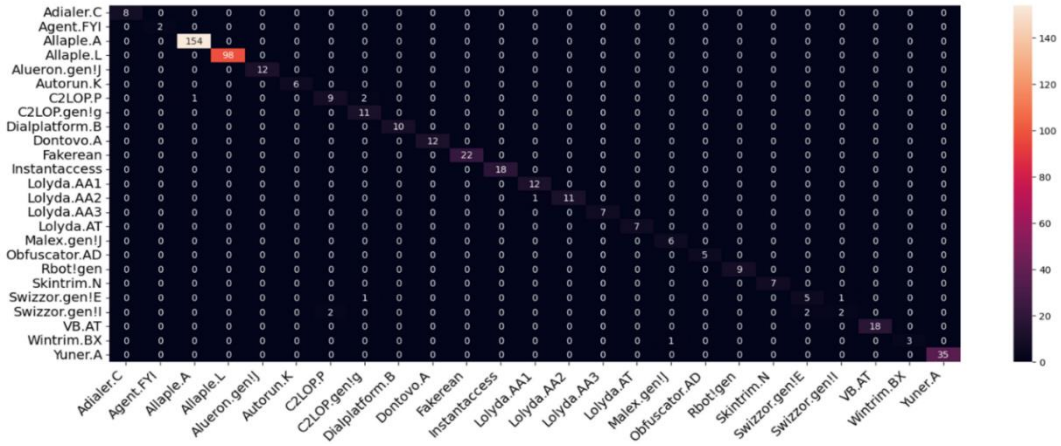

***Figure 35**: CNN-4 confusion matrix of MalIMG dataset*

4.3 Results of MaleVIS:

We will present in this section the results obtained after training and validating the MaleVIS dataset with the models mentioned above.

The input image size is (224,224), the training and validation is done by with a batch size of 2500. And we have chosen 10% of the data for validation 90% for testing.

First, in the table below we will present the number of epochs, accuracy and loss of each model:

***Table 5:** Number of epochs, accuracy an loss in the four models for MaleVIS*

	Number Of Epochs	Accuracy	Loss
CNN-1	100	86.4	1.4311
CNN-2	100	82.4	1.9713
CNN-3	100	84.80	1.3712
CNN-4	100	87.2	1.1527

Now, we will list the accuracy and loss learning curve representing the difference between the variation of testing and validation in the four different models.

- ***CNN-1:***

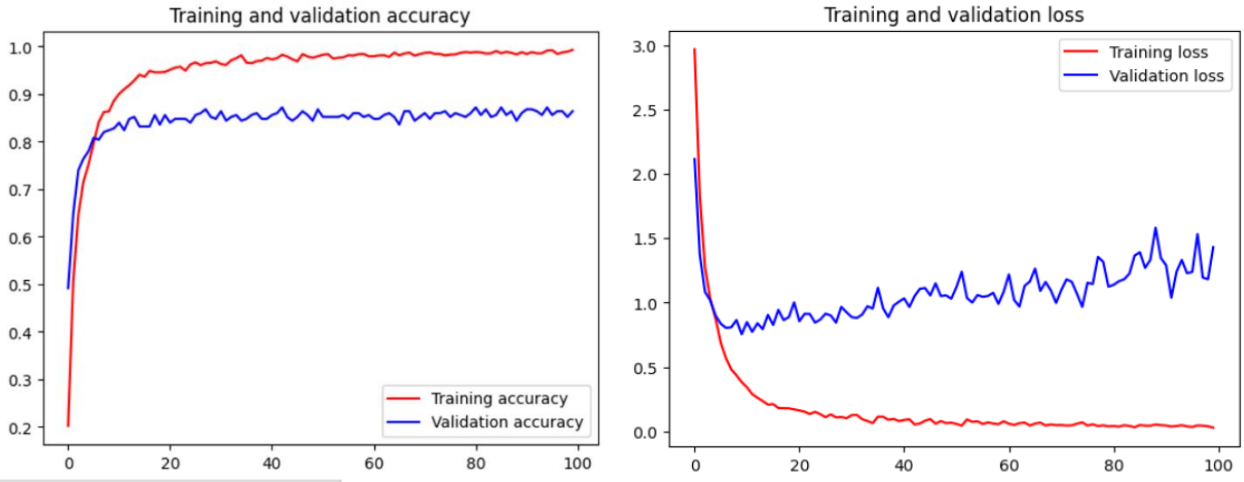

Figure 36: *The accuracy and loss learning curve for MaleVIS dataset- CNN-1*

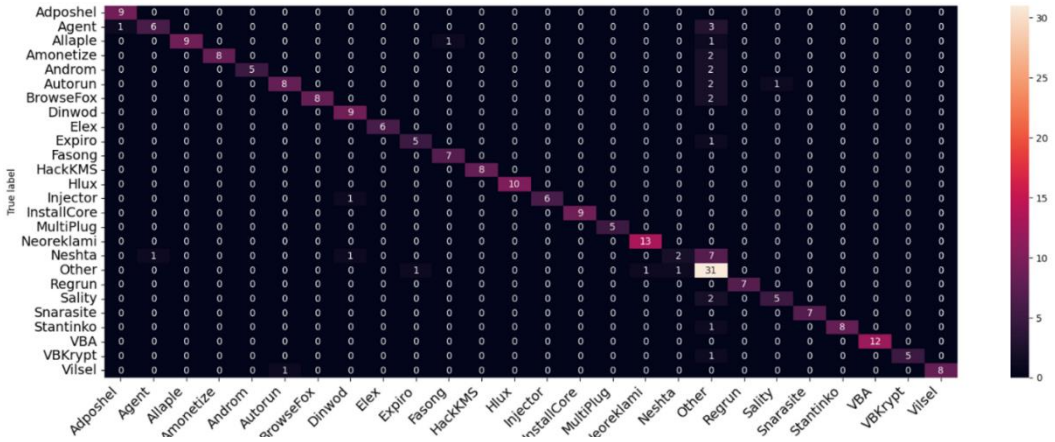

Figure 37: *CNN-1 confusion matrix of MaleVIS dataset*

- ***CNN-2:***

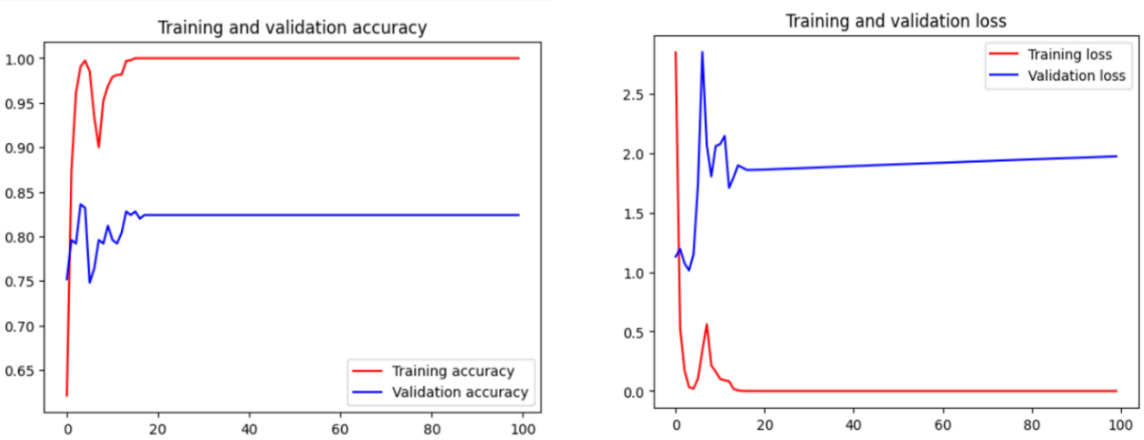

Figure 38: *The accuracy and loss learning curve for MaleVIS dataset- CNN-2*

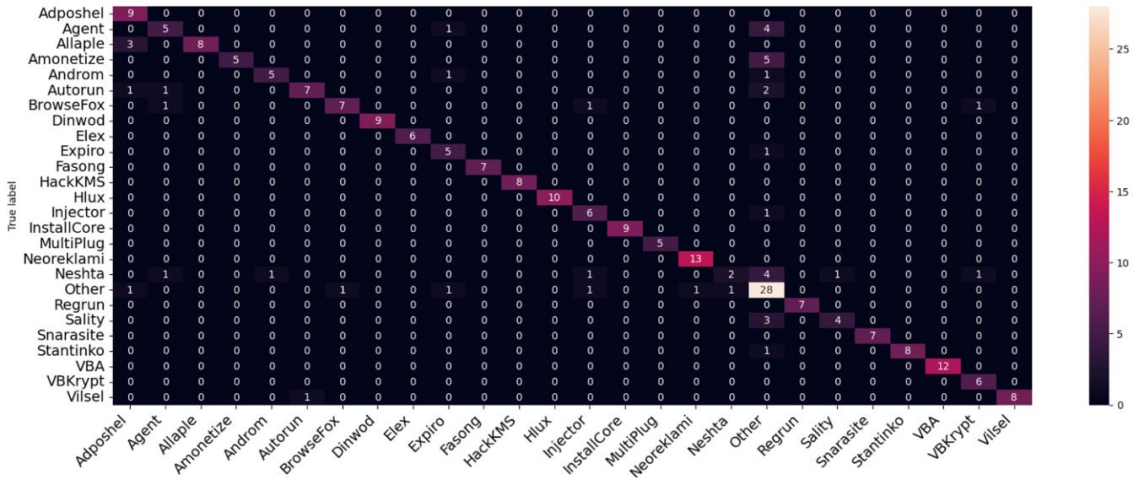

Figure 39: *CNN-2 confusion matrix of MaleVIS dataset*

- ***CNN-3:***

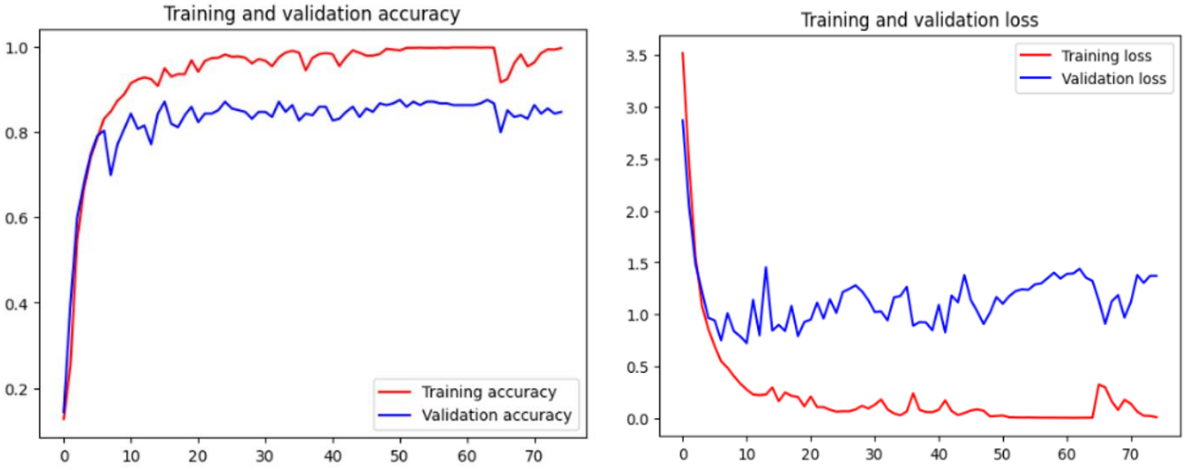

Figure 40*: The accuracy and loss learning curve for MaleVIS dataset- CNN-3*

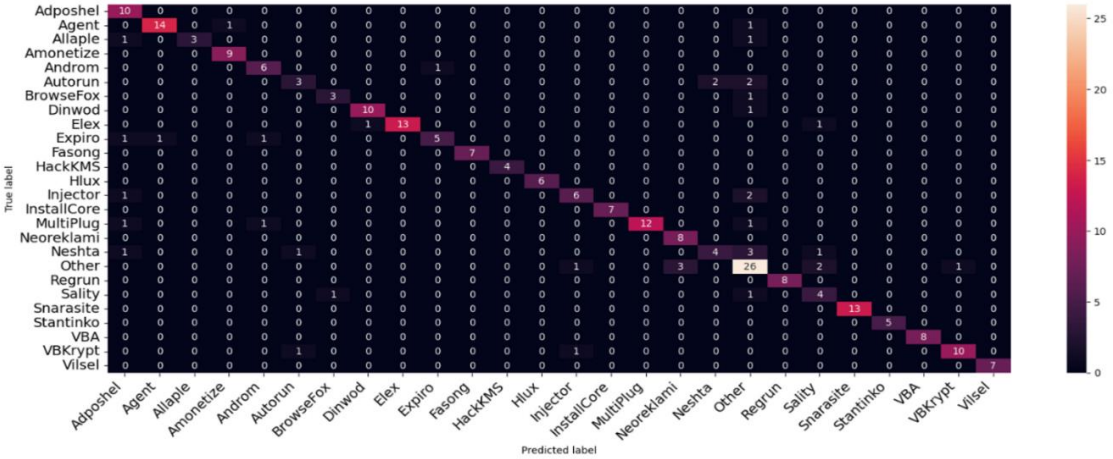

Figure 41: *CNN-3 confusion matrix of MaleVIS dataset*

- ***CNN-4:***

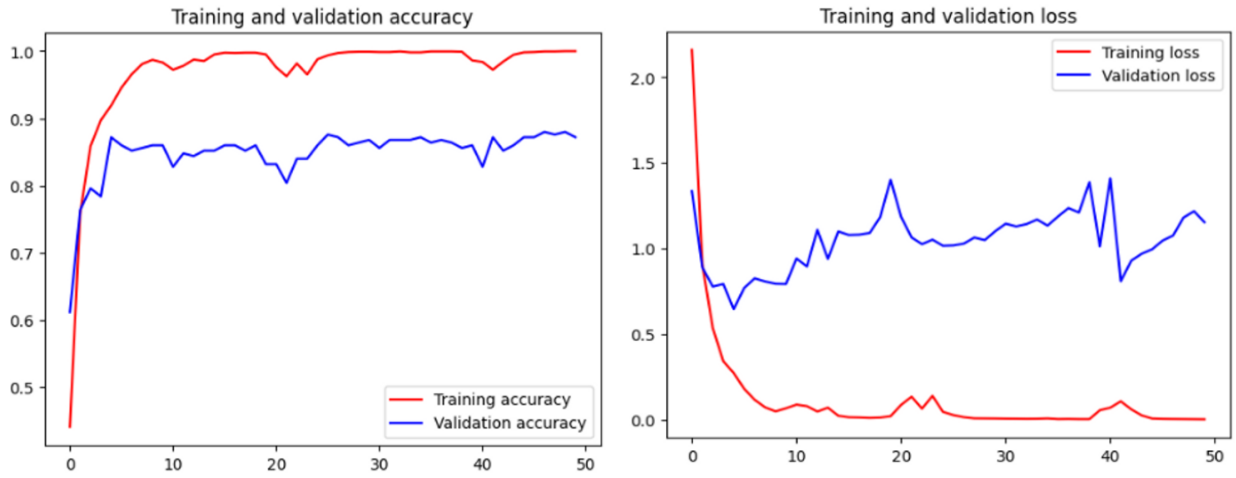

Figure 42: *The accuracy and loss learning curve for MaleVIS dataset- CNN-4*

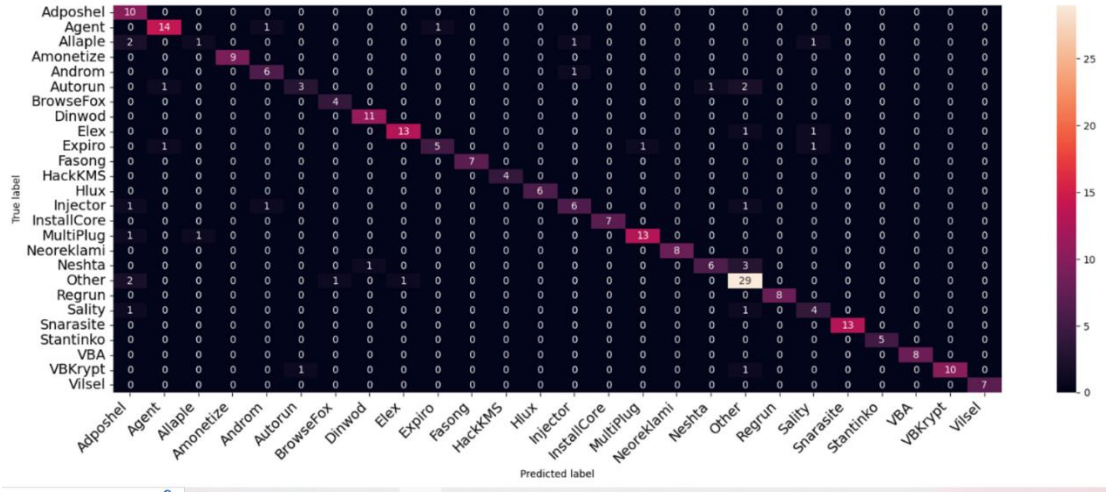

Figure 43: *CNN-4 confusion matrix of MaleVIS dataset*

5. Conclusion:

In this chapter, we presented some of the malware detection and classification models. By observing the performance of these models over the two datasets MalIMG and MaleVIS, we can conclude that these models work best with MalIMG dataset and that in MaleVIS occurs an overfit diagnosed from the learning curve, we will discuss it later in this report. We also can observe that CNN-2 is the model that has the lower performance in our case, and for that we have choose it to be the model to be modified later in Chapter 5 in order to optimize its performance.

Chapter 5: Implementation of CNN-2 Model's Modifications

1. Introduction:

Our project's main goal is to understand how changes made to a malware classification model affect its ability to detect and categorize malware. The ultimate aim is to develop a model that performs well across various datasets.

In Chapter 4, we observed that out of the four models tested on the MalIMG and MaleVIS datasets, CNN-2 performed the poorest. For this reason, we selected CNN-2 as the model to enhance through various modifications in the upcoming chapter. This will help us analyze the impact of these changes on its performance.

First in this chapter we will present the impact of each of the modifications chosen as well as its result over MalIMG dataset (we will not apply them over MaleVIS because the models with this dataset are overfitting) and we will also try to combine serval modifications together in order to design our own model that presents the best performance over various datasets.

2. Modifications Made:

Before going through the results and the impact over the model of CNN-2, let's first dive into each modification and understand them clearly

2.1 Modification 1:Substituting Layer with Batch Normalization:

This modification will be called Mod1 later in this report.

Batch normalization (also known as batch norm) is a method used to make training of artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling [15]. Note that this type of normalization is not effective with small batch size

Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch (see *figure 44*). This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

In layer normalization, all neurons in a particular layer effectively have the same distribution across all features for a given input.

Normalizing across all features but for each of the inputs to a specific layer removes the dependence on batches. This makes layer normalization well suited for sequence models such as transformers and recurrent neural network (RNN) that were popular in the pre-transformer era.

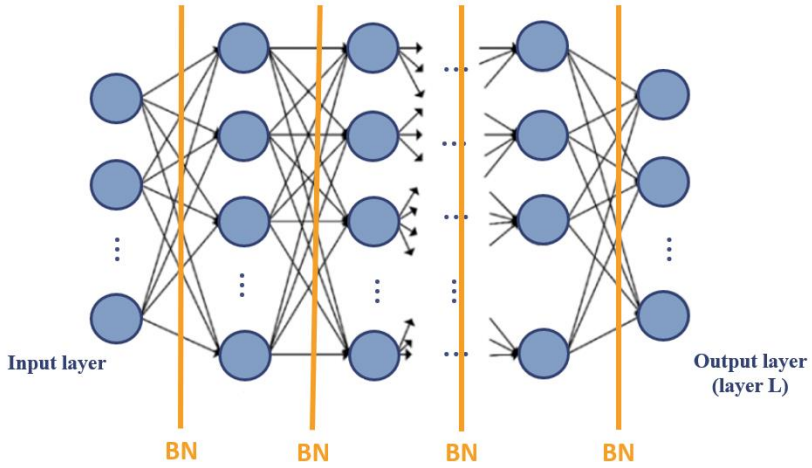

Figure 44: *Batch normalization*

The table below shows the different between batch normalization and layer normalization:

Table 6: *Table showing the difference between batch and layer normalization*

Batch Normalization	Layer Normalization
Normalize each feature independently across the mini-batch	Normalizes each of the inputs in the batch independently across all features.
Dependent on batch size, it's not effective for small batch sizes.	Independent of the batch size, so it can be applied to batches with smaller sizes as well.
Requires different processing at training and inference times	Is done along the length of input to a specific layer, the same set of operations can be used at both training and inference times.

2.2 *Modification 2: Adding Convo Layer:*

This modification will be called Mod2 later in this report.

By adding a convolutional layer, we increase the model complexity which allows it to capture more intricate patterns in the data. It also helps it learn a deeper hierarchy of features which will

lead to higher level representation of the data. But this modification can lead to overfitting especially if the dataset is not large enough.

2.3 Modification 3: Substituting ReLU With Sigmoid Activation Function:

This modification will be called Mod3 later in this report.

Our main objective is to analyze the effects of different activation functions on the performance of our malware classification model. Specifically, we compare the use of the sigmoid activation function with the widely used ReLU (Rectified Linear Unit) activation function. Our objective is to understand how the choice of activation functions influences convergence speed, generalization, and classification accuracy.

The explanation of these two activation functions is presented in Chapter 2, and the discussion of the impact will be presented in the next section.

2.4 Modification 4: No Normalization:

This modification will be called Mod4 later in this report.

The normalization layers (such as LayerNormalization) in a neural network play a role in standardizing the inputs to each layer, which can aid in the training process and potentially improve model convergence and generalization.

In our original model architecture, we have used LayerNormalization after each MaxPooling2D layer. If we were to remove these normalization layers and keep the rest of the architecture unchanged, we will have some impact over the performance of our model that we will discuss later.

2.5 Modification 5: Variation Of Learning Rate:

This modification will be called Mod5 later in this report.

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

The learning rate may be the most important hyperparameter when configuring your neural network. Therefore, it is vital to know how to investigate the effects of the learning rate on model performance and to build an intuition about the dynamics of the learning rate on model behavior.

If you set a high learning rate, then your system will rapidly adapt to new data, but it will also tend to quickly forget the old data (you don't want a spam filter to flag only the latest kinds of spam it was shown). Conversely, if you set a low learning rate, the system will have more inertia; that is, it will learn more slowly, but it will also be less sensitive to noise in the new data or to sequences of nonrepresentative data points (outliers) [1].

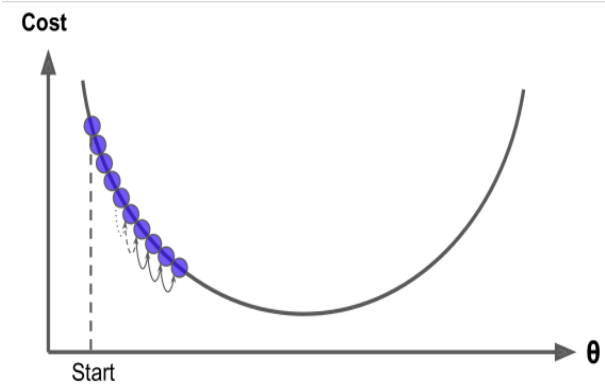

Figure 45: *Learning rate too small*

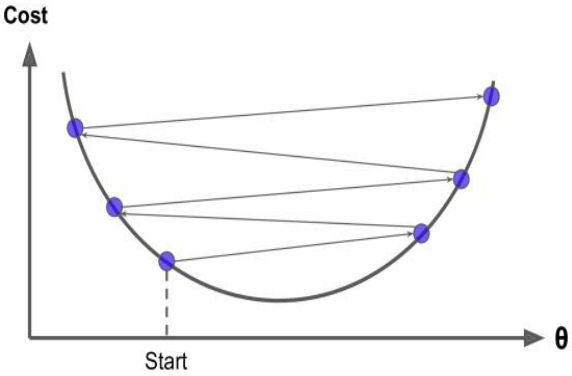

Figure 46: *Learning rate too large*

In this section we have choose serval learning rate and visualized the performance of the model with each of them in order to choose the optimal learning rate in our case. The range of learning rate chosen is between 0.001 and 0.01.

2.7 Combination Of Modifications:

We choose in this part to combine some of the modifications above in order to also see the impact on our model performance, so we have chosen two types of modifications, the first is some combinations of modifications applied to the Mod 2 (batch normalization), the second one is applying the same set of modifications in addition to Mod 4 (No Normalization).

We will go through each one of these combinations later in the *Results and discussion section.*

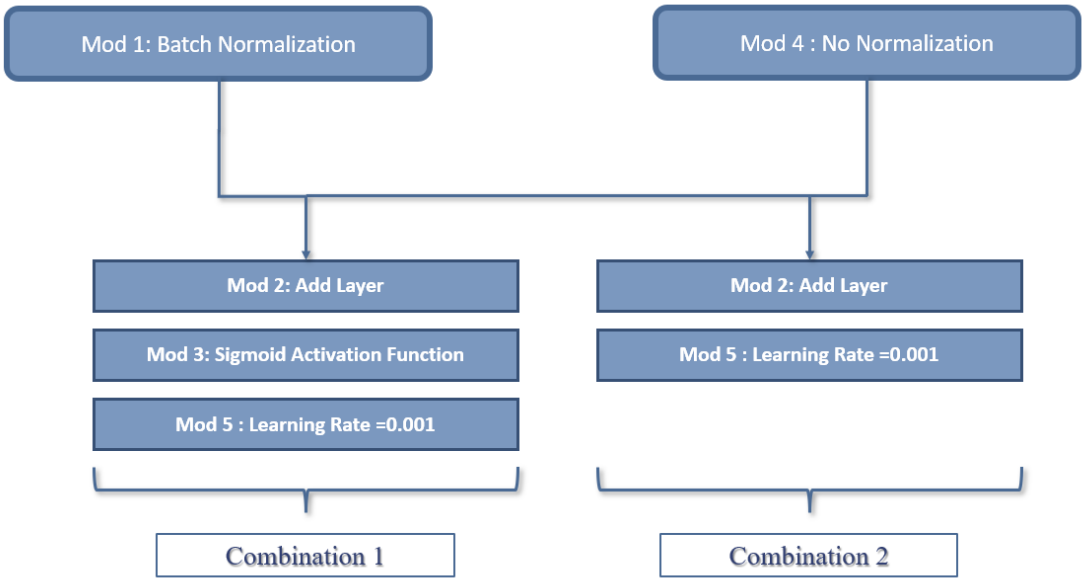

Figure 47: Graph representing the combination of modifications applied

3. Results And Discussion:

- ***Original CNN-2 results:***

The original results of CNN-2 are presented in Chapter 4 *(section 4.1-ii)* we occurred an accuracy of **95.8%** and a loss of **0.2773.**

3.1 Mod 1: Substituting Layer with Batch Normalization:

i. Results:

This model is trained within 100 epochs and occurred an accuracy of **96.8%** and a loss of **0.2505.**

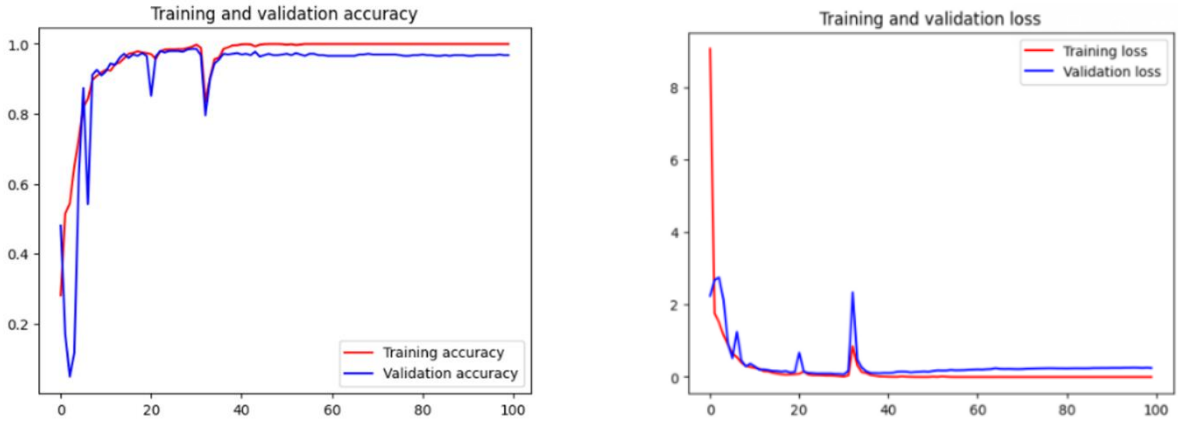

Figure 48: *Learning curve for Mod1 with MalIMG*

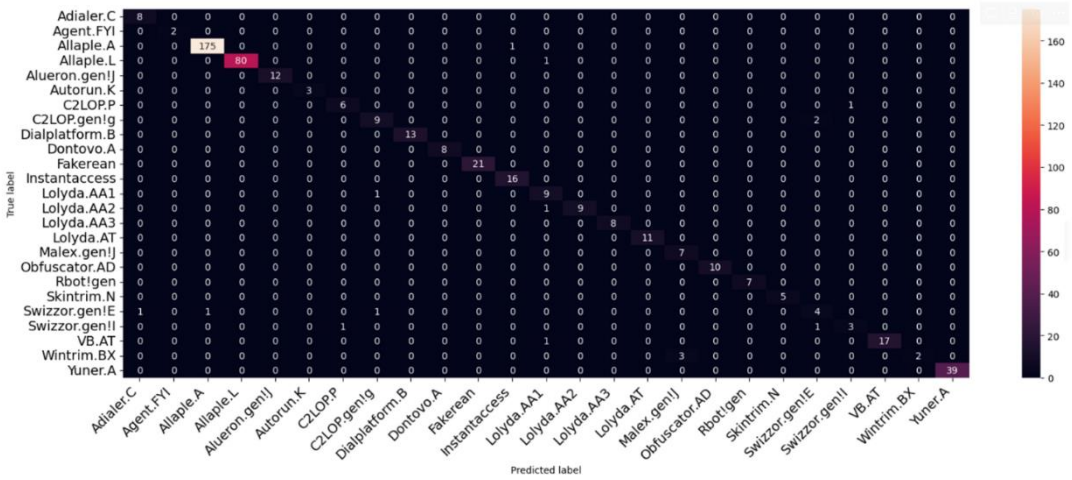

Figure 49: *Confusion matrix for Mod1*

ii. Discussion:

The introduction of batch normalization in place of layer normalization in our malware detection CNN model yielded a noteworthy improvement in accuracy from **95.8% to 96.8%;** however, an intriguing aspect of this transition was the irregular spikes observed in the loss and accuracy learning curves. These spikes, while initially confusing, can be attributed to several factors inherent in neural network training dynamics.

Firstly, batch normalization has the potential to render the model more sensitive to hyperparameters, such as learning rate and weight initialization, which if not optimally tuned, can result in brief periods of oscillation in the curves.

Furthermore, the inherent complexity of the deep CNN architecture, combined with occasional data outliers or noise, can lead to infrequent fluctuations during training. Additionally, the nature

of batch normalization, which relies on mini-batch statistics, can lead to variability in the normalization process, occasionally resulting in temporary spikes.

While these spikes are a valid area of concern, it is essential to recognize that their impact is transient and does not overshadow the overall positive trend of the model's performance. These observations underscore the confusion of neural network training and highlight the importance of sensible hyperparameter tuning and regularization techniques to minimize these short-lived instabilities.

3.2 *Mod 2: Adding Convo Layer:*

This model is trained over 100 epochs with an accuracy of **97.04%** and a loss of **0.2652.**

i. Results:

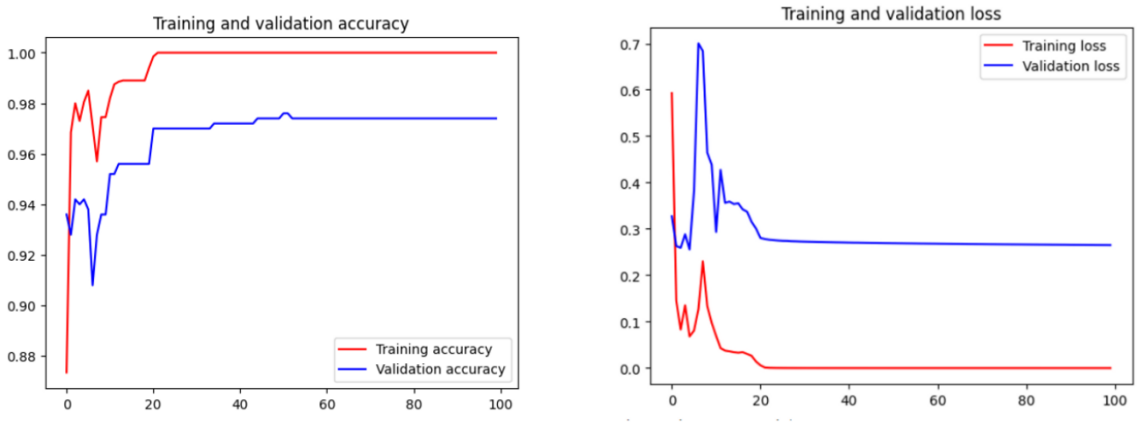

Figure 50: *Learning curve for Mod 2 with MalIMG*

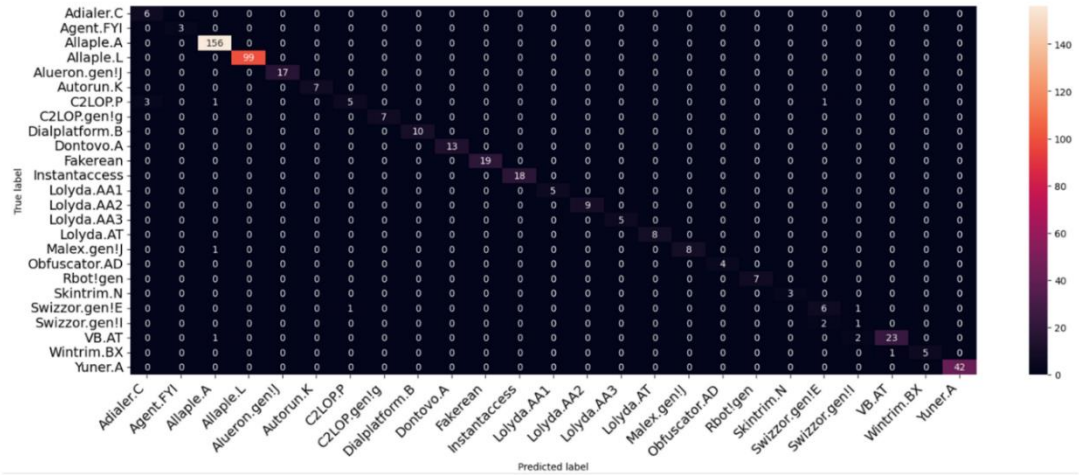

Figure 51: *Confusion metrics for Mod 2*

ii. Discussion:

The introduction of a convolutional layer with max pooling to the original CNN model has led to a significant accuracy boost, elevating the model's performance from 95.8% to 97.04%.

This improvement demonstrates the positive impact of architectural modifications on model efficacy. However, it's noteworthy that a minor instance of overfitting is observed towards the end of training. The overfitting could potentially be attributed to the increased complexity introduced by the new convolutional layer and max pooling. As the model gains more layers, it becomes more capable of capturing intricate patterns within the training data, potentially including noise or outlier information.

Reducing this slight overfitting could involve strategies such as introducing stronger regularization techniques or augmenting the dataset to ensure a balanced representation of features. Despite this small challenge, the overall accuracy enhancement underlines the model's capacity to learn complex patterns.

3.3 Mod 3: Substituting ReLU With Sigmoid Activation Function:

i. Results:

This model is trained over 200 epochs with an accuracy of **95%** and a loss of **0.1792.**

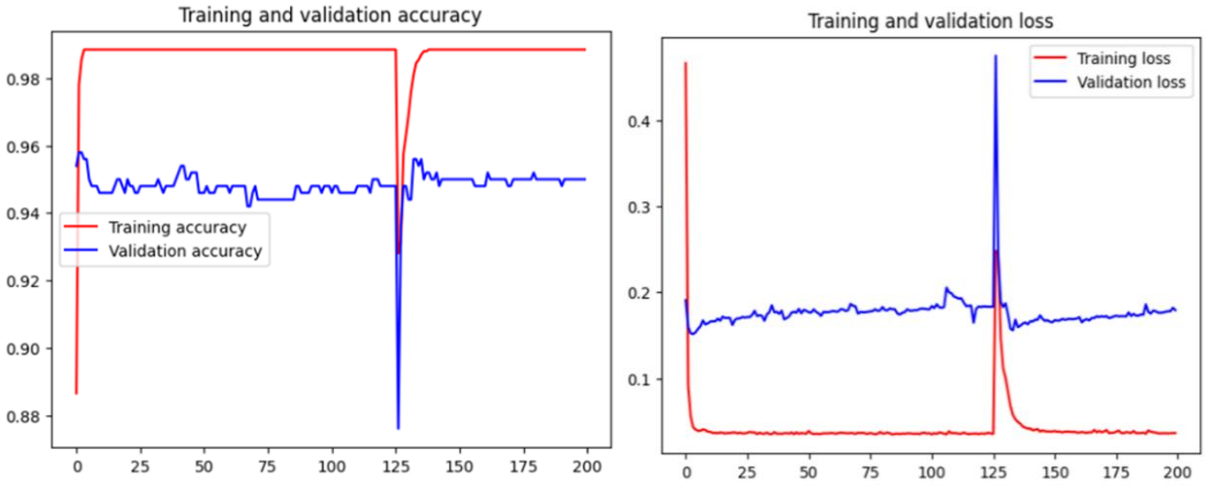

Figure 52: *Learning curve for Mod 3 with MalIMG*

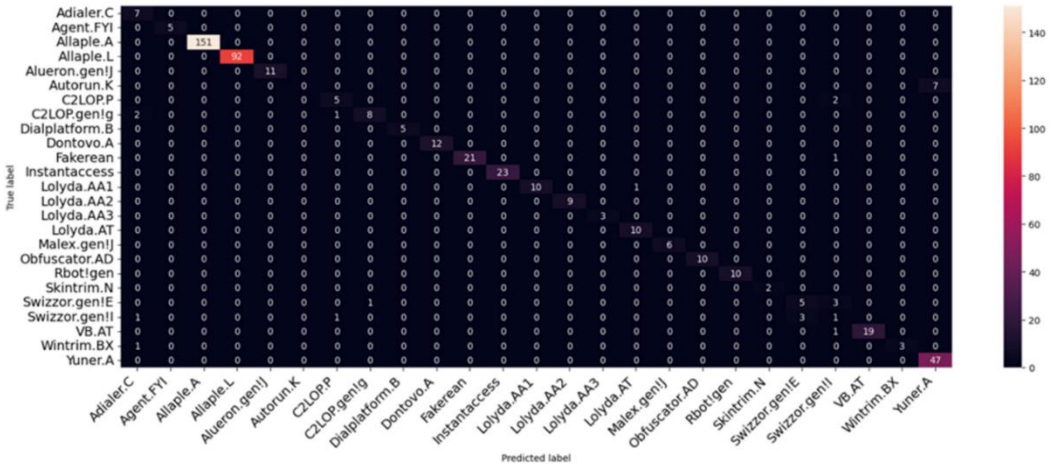

Figure 53: *Confusion metrics for Mod 3*

ii. Discussion:

The modification of the original CNN model, where all ReLU activation functions were replaced with sigmoid activations, led to an accuracy of 95% and a loss of 0.1792. This alteration in activation functions can notably impact how the model processes information and captures features. Despite the accuracy being slightly lower than the initial model's 95.8%, it's still a commendable result. The observed loss value indicates that the model is able to provide accurate predictions with relatively low uncertainty.

On analyzing the learning curves, a consistent and stable training accuracy of 99% is evident, suggesting the model has effectively learned the training data. The validation accuracy, ranging between 94% and 95%, with some fluctuations, could be attributed to the model's sensitivity to the validation dataset's variability. The isolated spike in the accuracy learning curve around epoch 125 could be a momentary perturbation in the training process, possibly due to a sudden shift in the data distribution or a localized optimization challenge.

While this spike could raise concern, the overall trend of stability in the training process supports the model's robustness. This experimentation showcases the importance of activation functions in influencing a model's learning dynamics, as well as the potential trade-offs between accuracy and interpretability.

3.4 Mod 4: No Normalization:

i. Results:

This model is trained over 100 epochs with an accuracy of **96.4%** and a loss of **0.3456.**

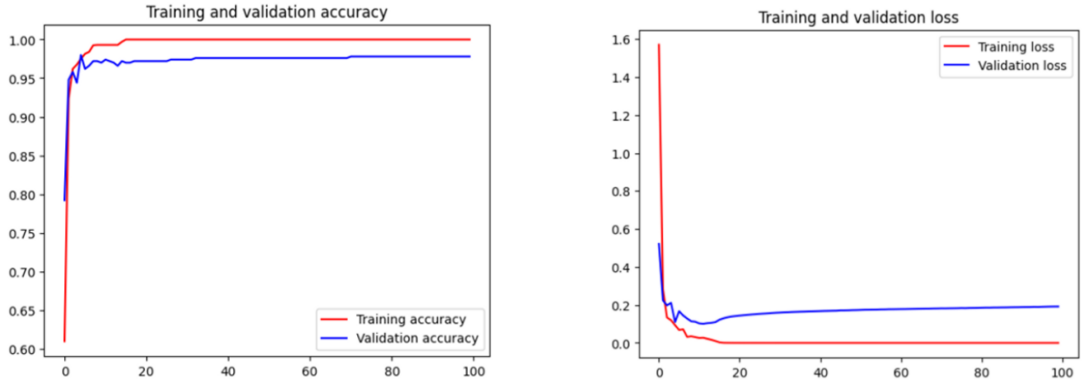

Figure 54: *Learning curve for Mod 4 with MalIMG*

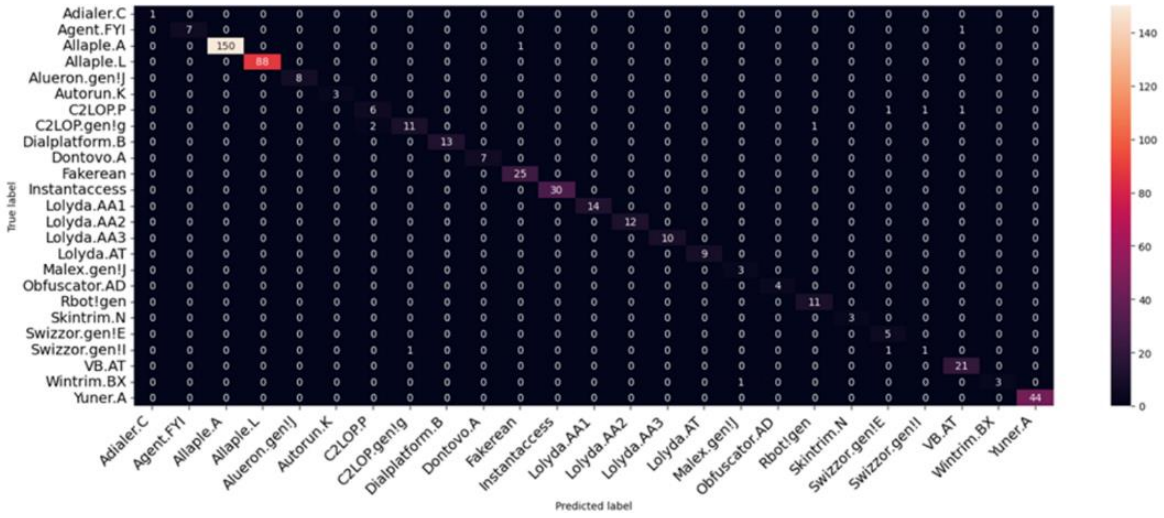

Figure 55: *Confusion metrics for Mod 4*

ii. Discussion:

The removal of normalization from the modified CNN model led to an accuracy of 96.4% and a loss of 0.3456. This modification sheds light on the significance of normalization techniques in training stability and convergence. Despite the achieved accuracy being promising at 96.4%, attention should be directed to the loss value of 0.3456, which indicates a moderate level of uncertainty in the model's predictions.

Turning to the learning curve analysis, the training accuracy curve holding a stable value of 1 from epoch 20 to 100 is indicative of potential overfitting. This suggests that the model might be fitting the training data almost perfectly but could struggle to generalize to new data points. The validation accuracy ranging between 95% and 96%, with stability from epochs 80 to 100, suggests a possible plateau where the model's learning capacity for validation data reaches a limit.

Regarding the loss learning curve, the minimal training loss, nearly approaching 0, signifies that the model is closely fitting the training data. However, the slight elevation in the validation loss suggests that the model's performance on unseen data might not be as strong.

This iteration of the model underscores the role of normalization in maintaining training stability and facilitating better generalization.

3.6 Mod 5: Variation Of Learning Rate:

i. Results:

The main objective of this modification is to visualize the performance of our model with the variation of learning rate in a range between 0.001 and 0.01. The table below presents the values of accuracy and loss with each learning rate.

Table 7: *Accuracy and loss values with Mod 5*

Learning Rate	Accuracy	Loss
0.001	95.2	1.5
0.002	95.6	0.1589
0.003	96	0.1614
0.004	95	0.2034
0.005	92.4	0.3056

Learning Rate	Accuracy	Loss
0.006	95.6	0.2265
0.007	93.8	0.2881
0.008	90.4	0.2131
0.009	82.4	0.6671

And the graph below represents the variation of accuracy and loss with the variation of learning rate values

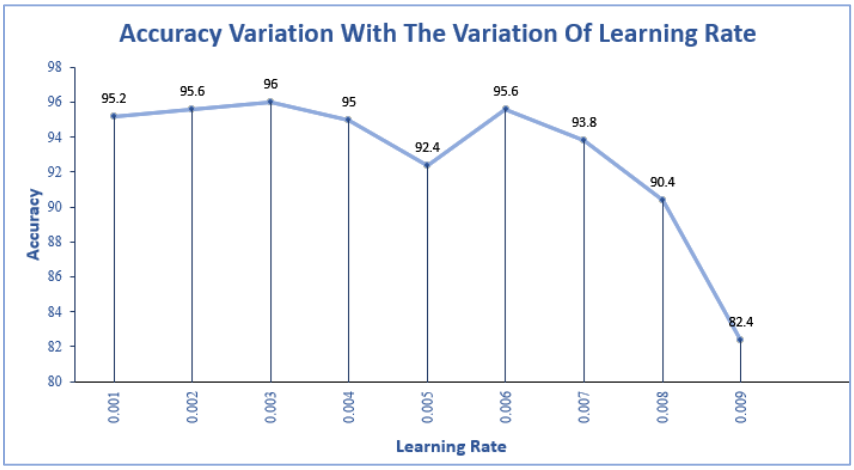

Figure 56: *Graph representing the variation of accuracy with the variation of learning rate*

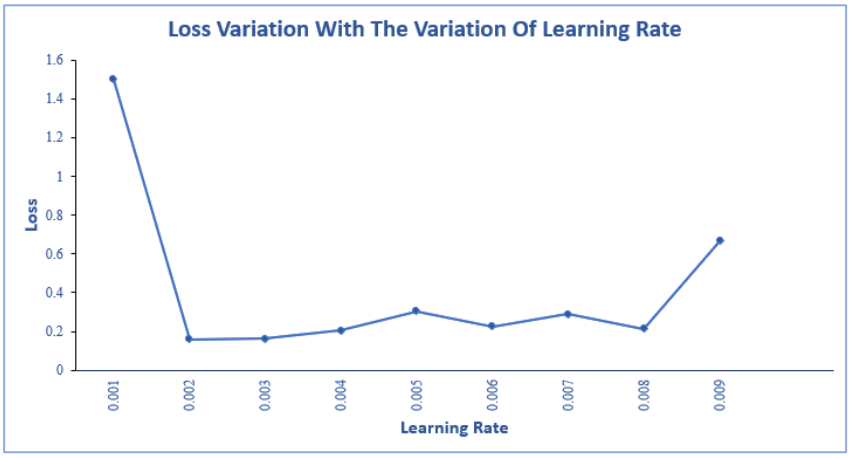

Figure 57: *Graph representing the variation of loss with the variation of learning rate*

ii. Discussion:

The systematic exploration of various learning rates in conjunction with the Adam optimizer provided valuable insights into the impact of this hyperparameter on the training process of the CNN model. Learning rate plays a pivotal role in determining the step size taken towards minimizing the loss function during optimization.

The results of this experiment reveal a spectrum of outcomes based on different learning rate values. A learning rate that is too small, exemplified by 0.001, fosters a more cautious convergence, resulting in an accuracy of 95.2% and a loss of 1.5. On the other hand, larger learning rates such as 0.005 or 0.009, propel the optimization process rapidly, leading to accuracy values of 92.4% and 82.4% respectively, accompanied by less desirable losses of 0.3056 and 0.6671.

The optimal learning rate, in this case, emerges as 0.003, striking a harmonious balance between convergence speed and stability. This configuration yielded an accuracy of 96% and a loss of 0.1614, highlighting the capacity of the model to learn effectively and converge to a desirable solution. The experiment emphasizes the importance of selecting an appropriate learning rate to guide the optimization process, demonstrating how a well-chosen learning rate can significantly impact the model's performance and convergence behavior.

3.7 Combination Of Modifications:

As mentioned earlier, we are going to mix and match some of the modifications that we have discussed to see how they will affect the model’s performance.

In addition; we seek to evaluate the effects of both batch normalization and the absence of normalization on the model's performance .To address this, we have created two combinations for testing listed in the table below, along with their corresponding labels as referenced in this report.

We have chosen two combinations:

- Combination 1: Mod 2 (add layers) +Mod 3(sigmoid activation function) +Mod 5 (learning rate= 0.001)
- Combination 2: Mod 2 (add layers) +Mod 5 (learning rate=0.001)

Table 8: *Combinations labels*

	Combinations Chosen	Labels
Mod1: Batch Normalization	Combination 1	$C_{1,1}$
	Combination 2	$C_{1.2}$
Mod4: No Normalization	Combination 1	$C_{2.1}$
	Combination 2	$C_{2.2}$

- ***$C_{1,1}$:***

  i. Results of $C_{1,1}$:

As mentioned above this combination includes batch normalization with adding convo layer as well as a learning rate of 0.001.

This model is trained over 100 epochs with an accuracy of **97.8%** and a loss of **0.273.**

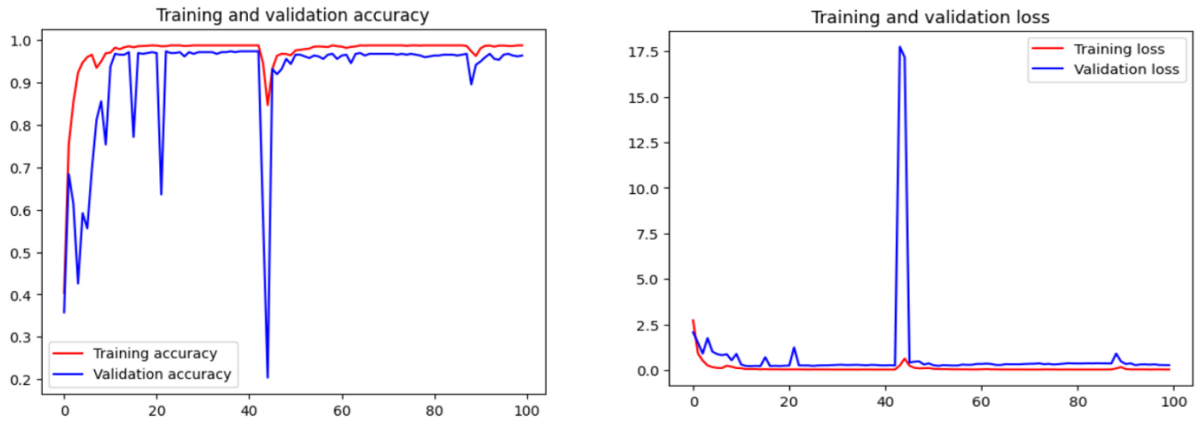

Figure 58: *Learning curve for C 1.1 with MalIMG*

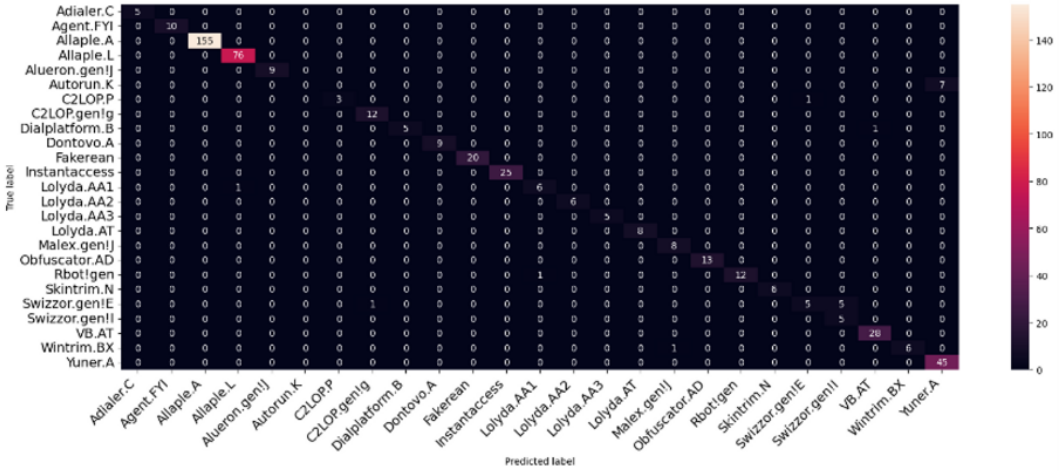

Figure 59*: Confusion metrics of C 1.1*

ii. Discussion:

Through a comprehensive integration of multiple modifications within a single model, we've achieved notable advancements in $C_{1.1}$. By strategically implementing a convolutional layer alongside max pooling, substituting the ReLU activation function with sigmoid, specifying a learning rate of 0.001, and adopting batch normalization, our efforts have culminated in a significant performance boost. The model's accuracy now stands at an impressive 97.8%, accompanied by a low loss of 0.273.

However, amidst this remarkable progress, it's important to address the occasional spikes observed in the learning curve, as discussed previously. These spikes, which could be attributed to brief instabilities in specific epochs, serve as reminders of the intricacies of neural network training dynamics.

Despite these fluctuations, the collective influence of our modifications has led to substantial improvements, underscoring the potential effects when incorporating multiple changes within a single model configuration. These results emphasize the holistic nature of model optimization and the necessity of both considering the achieved accuracy and monitoring the learning curve behaviors.

- ***C 1,2 :***

i. Results of C 1,2:

As mentioned above this combination includes batch normalization with adding convo layer, sigmoid activation function instead of ReLU as well as a learning rate of 0.001.

This model is trained over 100 epochs with an accuracy of **96.6%** and a loss of **0.1343.**

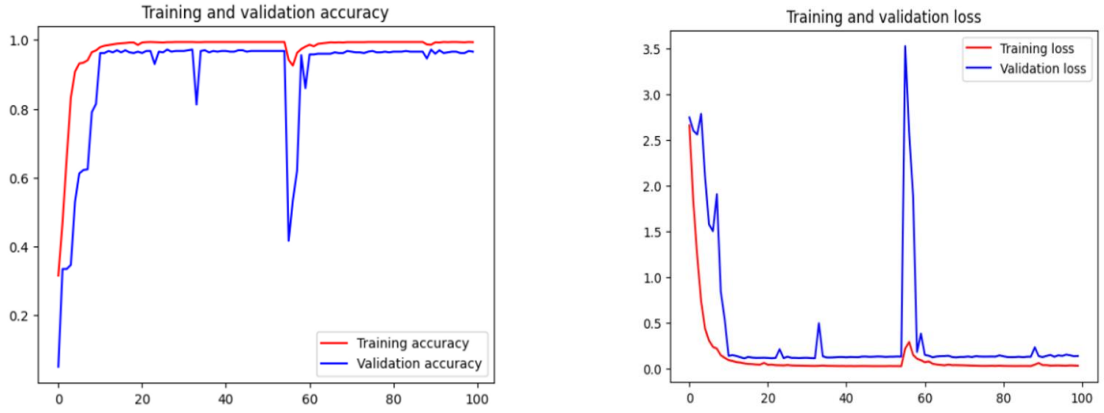

Figure 60: *Learning curve for C 1.2 with MalIMG*

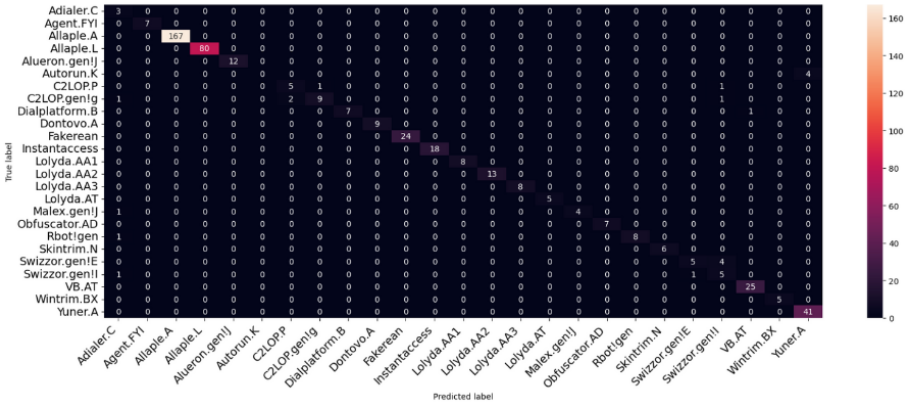

Figure 63: *Confusion metrics of C 1.2*

ii. Discussion:

Incorporating batch normalization, a convolutional layer, sigmoid activation, and a meticulously selected learning rate of 0.001, we have achieved substantial improvements. After 100 epochs of training, our model now boasts an accuracy of 96.6% and a remarkably low loss of 0.1343.

This combination effectively leverages the strengths of each modification to enhance the overall model performance. While the learning curve, despite displaying intermittent spikes which were expected with batch normalization, appears well-behaved, it's intriguing to note the behavior of the loss learning curve. Starting with relatively high values for both training and validation, the loss curve then descends to around 0.25 before transitioning into a minor oscillation. This phenomenon could be attributed to the combined effect of the architectural adjustments and the chosen learning rate.

The model initially grapples with convergence as it adapts to new techniques but stabilizes into a more optimal state as training progresses. These findings underscore the symbiotic nature of these changes, demonstrating their collective potency in enhancing our model's performance, with the learning curves reflecting the dynamic interplay between optimization strategies, training dynamics, and the underlying data distribution."

- ***C $_{2,1}$:***

i. Results of C $_{2,1}$:

As mentioned above this combination includes No normalization with adding convo layer as well as a learning rate of 0.001.

This model is trained over 50 epochs with an accuracy of **97.6%** and a loss of **0.2360.**

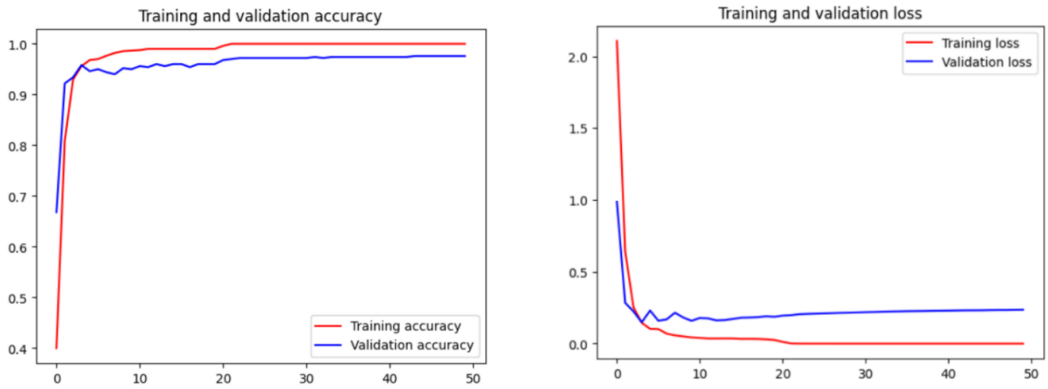

Figure 61:Learning curve for C 2.1 with MalIMG

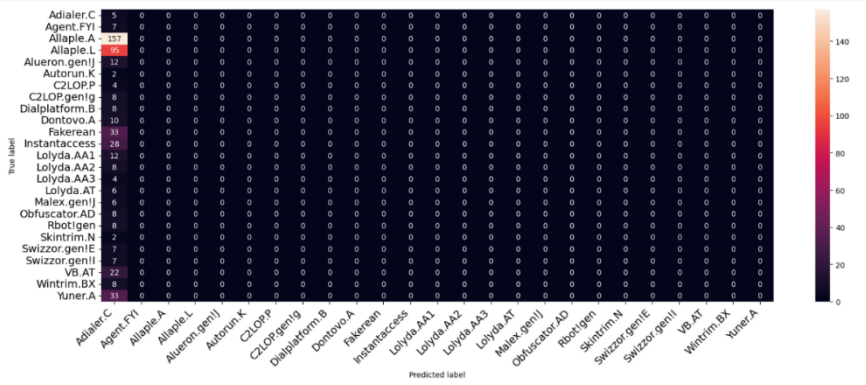

Figure 62: *Confusion metrics for C 2.1*

ii. Discussion:

In this combination we have incorporate a convolutional layer with max pooling, substitute the ReLU activation with sigmoid, and set a learning rate of 0.001 has yielded an accuracy of 97.6% and a loss of 0.2360.

This combined approach has showcased the impact of architectural and hyperparameter adjustments on model performance. Notably, the learning curve reveals intriguing trends. In terms of accuracy, the curve demonstrates initial stability, maintaining a validation accuracy within the range of 96-97% and a training accuracy of 100% for the first 20 epochs. This stability reflects the quick learning of the model on the training data. However, in terms of loss, the curve tells a different story. From epochs 20 to 50, the training loss nears zero, suggesting a strong fit to the training data, while the validation loss experiences a slight elevation from 0.1 to around 0.2 and continues to rise. This divergence between training and validation loss could point to overfitting, indicating that while the model excels in memorizing training data, it struggles to generalize to new, unseen data.

Despite this potential overfitting concern, the overall outcome underscores the effectiveness of the collective modifications in enhancing accuracy and demonstrating the intricate interplay between architectural choices, learning dynamics, and the inherent complexities of the dataset.

- ***C $_{2,2}$:***

i. Results of C $_{2,2}$:

As mentioned above this combination includes No normalization with adding convo layer, using sigmoid as activation function, as well as a learning rate of 0.001.

This model is trained over 50 epochs with an accuracy of **29%** and a loss of **2.555.**

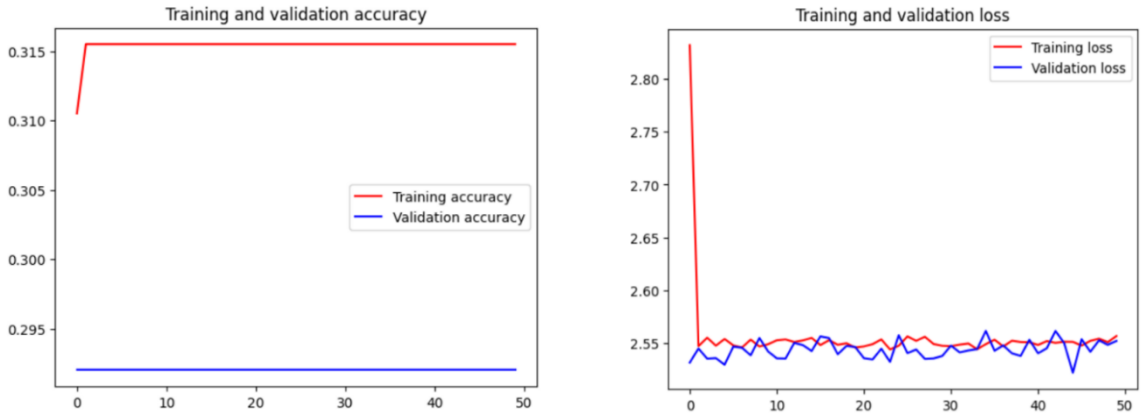

Figure 63: *Learning curve for C 2.2 with MalIMG*

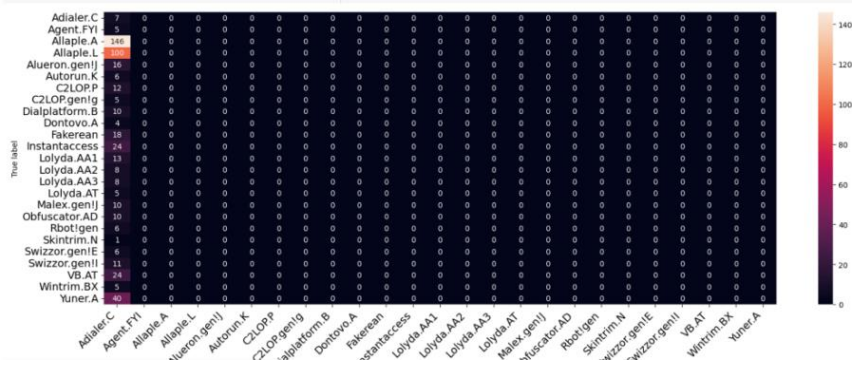

Figure 64: *Confusion metrics of C 2.2*

ii. Discussion:

The experimentation with different modifications within the model has yielded results that warrant examination. The incorporation of a convolutional layer with max pooling, along with the specified learning rate of 0.001 and the absence of normalization, has culminated in an accuracy of 29% and a considerably high loss of 2.555.

The learning curve provides insight into the challenges faced by this model iteration. In terms of accuracy, the curve displays a lack of progress, with both training and validation accuracies stagnant at around 31% after an initial rise. This lack of significant improvement could point to a misalignment between the model's complexity and the dataset's intricacies.

The loss learning curve reinforces this notion, as it oscillates around a higher loss value of 2.55 for both training and validation, indicating a model that struggles to optimize effectively. These outcomes suggest that the introduction of the convolutional layer without proper normalization might have led to suboptimal convergence.

The observed discrepancies between accuracy and loss behaviors, along with the lack of convergence, underscore the delicate balance required when modifying neural network architectures, emphasizing the significance of strategies like normalization to facilitate stable and effective training dynamics.

3.8 Discussion Of MaleVIS Results:

We will now discuss the result observed for the dataset MaleVIS in *Chapter 4 section 4.3.*

The exploration of four different models on the MaleVIS dataset has revealed a recurring challenge of overfitting, which warrants careful consideration and remediation.

Overfitting occurs when a model learns to perform exceedingly well on the training data, but struggles to generalize effectively to new, unseen data. This phenomenon is likely due to a mismatch between the complexity of the model and the size and diversity of the dataset. The overfitting can be attributed to the models capturing not only the genuine patterns in the data but also noise or outliers, leading to suboptimal performance on validation or testing data.

To mitigate overfitting, several strategies can be employed. One approach involves introducing regularization techniques, such as dropout or L2 regularization, which discourage the model from relying too heavily on any single feature or neuron. Additionally, augmenting the dataset or using data augmentation techniques can enhance the model's exposure to various scenarios, promoting better generalization.

Experimenting with different model architectures, adjusting hyperparameters like learning rate, and ensuring proper validation techniques like cross-validation can also aid in addressing overfitting. As you navigate these challenges, it's prudent to carefully analyze your model's learning curves, loss trends, and validation performance to strike the right balance between model complexity and data characteristics, ultimately ensuring a more robust and generalized solution.

4. Conclusion:

In this chapter, we visualized the results of the modifications made to the model CNN-2 as well as the performance of this model with every modification and with a combinations of ones. We have also discussed the results obtained with the second dataset, MaleVIS. In the next chapter we will present first the future work that would be done with this project and we will also present our conclusion for this project.

General Conclusion And Future Work:

In this project, we embarked on the journey of enhancing malware detection through Convolutional Neural Networks (CNNs). Through a systematic exploration of four models inspired by different research papers, we dived into the complication of CNN architecture and its application in the realm of malware identification. After careful evaluation, we selected a model that exhibited lower accuracy and subsequently devised a series of six modifications aimed at improving its performance. And we also presented four combinations of these modifications in order to find the best model in detecting and classifying malwares. By applying these modifications, we aimed to harness the power of convolutional layers, activation functions, learning rates, and normalization techniques. Our experiments on the MalIMG dataset, commonly used in prior studies, provided valuable insights into the dynamics of each modification's impact on the model's accuracy, loss, and learning curve behaviors.

While our endeavor has yielded significant progress, there remain avenues for future exploration and refinement. The transition to the MaleVIS dataset revealed the challenge of overfitting, a phenomenon that warrants dedicated attention. Future work will focus on implementing strategies to combat overfitting, including the incorporation of regularization techniques, data augmentation, and architectural adjustments to strike a harmonious balance between complexity and generalization. Additionally, we envisage extending our efforts to real-world scenarios by transforming hexadecimal values into grayscale images and subjecting them to our model. This will enable us to validate the model's efficacy in real-time applications, reflecting its potential impact in actual cybersecurity settings.

References

[1] "Hands–On Machine Learning with Scikit–Learn and TensorFlow 2e (PDFDrive).pdf."

[2] "Goodfellow et al. - 2016 - Deep learning.pdf."

[3] "Adrian Rosebrock - Deep Learning for Computer Vision with Python. 1,Starter Bundle-PyImageSearch (2017) 1.pdf."

[4] M. J. Anzanello and F. S. Fogliatto, "Learning curve models and applications: Literature review and research directions," *Int. J. Ind. Ergon.*, vol. 41, no. 5, pp. 573–583, Sep. 2011, doi: 10.1016/j.ergon.2011.05.001.

[5] "Diagnosing Model Performance with Learning Curves." https://rstudio-conf-2020.github.io/dl-keras-tf/notebooks/learning-curve-diagnostics.nb.html (accessed Aug. 20, 2023).

[6] "Ni et al. - 2018 - Malware identification using visualization images .pdf."

[7] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Using convolutional neural networks for classification of malware represented as images," *J. Comput. Virol. Hacking Tech.*, vol. 15, no. 1, pp. 15–28, Mar. 2019, doi: 10.1007/s11416-018-0323-0.

[8] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang, and F. Iqbal, "Malware Classification with Deep Convolutional Neural Networks," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, Paris: IEEE, Feb. 2018, pp. 1–5. doi: 10.1109/NTMS.2018.8328749.

[9] H. Mallet, "Malware Classification using Convolutional Neural Networks — Step by Step Tutorial," *Medium*, May 28, 2020. https://towardsdatascience.com/malware-classification-using-convolutional-neural-networks-step-by-step-tutorial-a3e8d97122f (accessed Aug. 13, 2023).

[10] "Nataraj et al. - 2011 - Malware images visualization and automatic classi.pdf."

[11] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Using convolutional neural networks for classification of malware represented as images," *J. Comput. Virol. Hacking Tech.*, vol. 15, no. 1, pp. 15–28, Mar. 2019, doi: 10.1007/s11416-018-0323-0.

[12] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang, and F. Iqbal, "Malware Classification with Deep Convolutional Neural Networks," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, Paris: IEEE, Feb. 2018, pp. 1–5. doi: 10.1109/NTMS.2018.8328749.

[13] R. Kumar and R. U. Khan, "Opcode and Gray Scale Techniques for Classification of Malware Binaries".

[14] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, Pittsburgh Pennsylvania USA: ACM, Jul. 2011, pp. 1–7. doi: 10.1145/2016904.2016908.

[15] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." arXiv, Mar. 02, 2015. Accessed: Aug. 20, 2023. [Online]. Available: http://arxiv.org/abs/1502.03167

[16] A. Damodaran, F. Di Troia, V. A. Corrado, T. H. Austin, and M. Stamp, "A Comparison of Static, Dynamic, and Hybrid Analysis for Malware Detection," *J. Comput. Virol. Hacking Tech.*, vol. 13, no. 1, pp. 1–12, Feb. 2017, doi: 10.1007/s11416-015-0261-z.

[17] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, "Backpropagation and the brain," *Nat. Rev. Neurosci.*, vol. 21, no. 6, pp. 335–346, Jun. 2020, doi: 10.1038/s41583-020-0277-3.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. in Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016.

[19] Z. Cui, F. Xue, X. Cai, Y. Cao, G. Wang, and J. Chen, "Detection of Malicious Code Variants Based on Deep Learning," *IEEE Trans. Ind. Inform.*, vol. 14, no. 7, pp. 3187–3196, Jul. 2018, doi: 10.1109/TII.2018.2822680.

[20] M. S. Akhtar and T. Feng, "Detection of Malware by Deep Learning as CNN-LSTM Machine Learning Techniques in Real Time," *Symmetry*, vol. 14, no. 11, p. 2308, Nov. 2022, doi: 10.3390/sym14112308.

[21] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization." arXiv, Jul. 21, 2016. Accessed: Aug. 20, 2023. [Online]. Available: http://arxiv.org/abs/1607.06450

[22] H. Rathore, S. Agarwal, S. K. Sahay, and M. Sewak, "Malware Detection using Machine Learning and Deep Learning," 2018, pp. 402–411. doi: 10.1007/978-3-030-04780-1_28.

[23] K. He and D.-S. Kim, "Malware Detection with Malware Images using Deep Learning Techniques," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, Rotorua, New Zealand: IEEE, Aug. 2019, pp. 95–102. doi: 10.1109/TrustCom/BigDataSE.2019.00022.

[24] S. Ni, Q. Qian, and R. Zhang, "Malware identification using visualization images and deep learning," *Comput. Secur.*, vol. 77, pp. 871–885, Aug. 2018, doi: 10.1016/j.cose.2018.04.005.

[25] H. Zhang, L. Feng, X. Zhang, Y. Yang, and J. Li, "Necessary conditions for convergence of CNNs and initialization of convolution kernels," *Digit. Signal Process.*, vol. 123, p. 103397, Apr. 2022, doi: 10.1016/j.dsp.2022.103397.

[26] J. Brownlee, "Overfitting and Underfitting With Machine Learning Algorithms," *MachineLearningMastery.com*, Mar. 20, 2016. https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/ (accessed Aug. 20, 2023).

[27] "(PDF) Detection of Malware by Deep Learning as CNN-LSTM Machine Learning Techniques in Real Time." https://www.researchgate.net/publication/365140194_Detection_of_Malware_by_Deep_Learning_as_CNN-LSTM_Machine_Learning_Techniques_in_Real_Time (accessed Mar. 10, 2023).

[28] "Robust Intelligent Malware Detection Using Deep Learning | IEEE Journals & Magazine | IEEE Xplore." https://ieeexplore.ieee.org/document/8681127 (accessed Mar. 10, 2023).

[29] O. Aslan and A. A. Yilmaz, "A New Malware Classification Framework Based on Deep Learning Algorithms," *IEEE Access*, vol. 9, pp. 87936–87951, 2021, doi: 10.1109/ACCESS.2021.3089586.

[16] [3][17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29]

Appendix: Code's Implementation

All the needed information for the modifications and combinations applied are presented in this report, you can apply it on the original code of CNN-2 model.

Also note that the same models was applied to both datasets MalIMG and MaleVIS.

1. CNN-1 Model:

```
import tensorFlow as tf
import keras
from keras import *
from keras.layers import *
print(tf.config.list_physical_devices('GPU'))

def malware_model1():
    Malware_model = Sequential()
    Malware_model.add(Conv2D(30, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=(224,224,3)))

    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(Conv2D(15, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))

    Malware_model.add(Dropout(0.25))
    Malware_model.add(Flatten())
    Malware_model.add(Dense(128, activation='relu'))
    Malware_model.add(Dropout(0.5))
    Malware_model.add(Dense(50, activation='relu'))
    Malware_model.add(Dense(25, activation='softmax'))
    Malware_model.compile(loss='categorical_crossentropy', optimizer
= 'adam', metrics=['accuracy'])
    return Malware_model


CNN1=malware_model1()
CNN1.summary()
```

2. CNN-2 Model:

```
import tensorflow as tf
import keras
from keras import *
from keras.layers import *
print(tf.config.list_physical_devices('GPU'))

def malware_model2():
    Malware_model = Sequential()
    Malware_model.add(Conv2D(50, kernel_size=(5, 5),
                     activation='relu',
                     input_shape=(224,224,3)))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(LayerNormalization(axis=1))

    Malware_model.add(Conv2D(70, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(LayerNormalization(axis=1))

    Malware_model.add(Conv2D(70, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(LayerNormalization(axis=1))

    Malware_model.add(Flatten())
    Malware_model.add(Dense(256, activation='relu'))
    Malware_model.add(Dense(25, activation='softmax'))
    Malware_model.compile(loss='categorical_crossentropy', optimizer
= 'adam', metrics=['accuracy'])
    return Malware_model
CNN2=malware_model2()
CNN2.summary()
```

3. CNN-3 Model:

```
import tensorflow as tf
import keras
from keras import *
from keras.layers import *
print(tf.config.list_physical_devices('GPU'))

def malware_model3():
```

```
    Malware_model = Sequential()
    Malware_model.add(Conv2D(64, kernel_size=(5, 5),
                     activation='relu', input_shape=(224,224,3)))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))

    Malware_model.add(Conv2D(128, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))

    Malware_model.add(Conv2D(256, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))

    Malware_model.add(Conv2D(512, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))

    Malware_model.add(Conv2D(512, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))
    Malware_model.add(Flatten())
    Malware_model.add(Dense(4096, activation='relu'))
    Malware_model.add(Dense(4096, activation='relu'))
    Malware_model.add(Dense(25, activation='softmax'))
    Malware_model.compile(loss='categorical_crossentropy', optimizer
= 'adam', metrics=['accuracy'])
    return Malware_model


CNN3=malware_model3()
CNN3.summary()
```

4. CNN-4 Model:

```
import tensorflow as tf
import keras
from keras import *
from keras.layers import *
print(tf.config.list_physical_devices('GPU'))

def malware_model4():
    Malware_model = Sequential()
    Malware_model.add(Conv2D(32, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=(224,224,3)))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))

    Malware_model.add(Conv2D(32, (3, 3), activation='relu'))
```

```
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))

    Malware_model.add(Conv2D(64, (3, 3), activation='relu'))
    Malware_model.add(MaxPooling2D(pool_size=(2, 2)))

    Malware_model.add(Flatten())
    Malware_model.add(Dense(256, activation='relu'))
    Malware_model.add(Dense(25, activation='softmax'))
    Malware_model.compile(loss='categorical_crossentropy', optimizer
= 'adam', metrics=['accuracy'])
    return Malware_model


CNN4=malware_model4()
CNN4.summary()
```