classification of the ransomware dataset, the Kulczynski distance gives an accuracy of 82%, while the Soergel distance gives an accuracy of 81%. Euclidean and Manhattan distances give an accuracy of 79% in this case.

Similarly, proximity measures such as Euclidean, Manhattan, Kulczynski, Cosine Similarity, Chebyshev, and Soergel distance give an accuracy of 94% when classifying the SWaT dataset. Among all the computations performed, the Chebyshev distance for binary classification of the ransomware dataset benefits the least from CUDA usage, with GPU computation being only 40.86 times faster than CPU computation. Conversely, the Cosine Similarity for classification of the SWaT dataset benefits the most from CUDA, with GPU computation being 237.05 times faster than CPU computation.

When dealing with binary classification of the ransomware dataset using the k-NN model, if accuracy is a high priority, the Kulczynski Distance or Soergel Distance as the proximity measure is recommended. For multi-class classification of the ransomware dataset using the k-NN model, if accuracy is a high priority, the Kulczynski Distance is recommended. For classification of the SWaT dataset using the k-NN model, if accuracy is a high priority, any of the five proximity measures can be used. However, if both accuracy and computational time are priorities, the Manhattan Distance is the better option.

The TUKNN has performed well for binary classification of the ransomware dataset and the SWaT dataset but has not performed as well for multi-class classification of the ransomware dataset. It has been observed that the TUKNN significantly benefits from CUDA's parallel computation in terms of reducing computational time.

Chapter 8

A Deep Learning Based Malware Detection Method

8.1 Introduction

Malware, designed to inflict undesirable or harmful effects on computer systems, poses a significant threat to their security. Typically categorized into various types such as worms, viruses, backdoors, Trojan horses, bots, rootkits, and spyware, malware often exhibits characteristics that span multiple categories. For instance, a worm may include a payload enabling the installation of a backdoor for remote access. Given the substantial loss and damage inflicted by malware, the focus on malware detection has become paramount in computer security.

The prevalent method for malware detection is the signature-based approach. This method employs a straightforward pattern-matching technique to identify malicious code, demonstrating high accuracy. The signature-based method, while widely adopted for its high accuracy, encounters limitations that necessitate exploration into alternative approaches for more robust malware detection. Notably, the sensitivity of signatures to slight variations in malicious code poses a significant drawback, making it susceptible to evasion through obfuscation technologies. This sensitivity also raises concerns about the adaptability of signature-based methods to evolving malware variants, particularly those modified to evade traditional detection mechanisms. Moreover, the reliance on known malware samples for signature creation introduces a challenge when confronted with previously unseen or modified instances of malicious code. As cyber adversaries continually employ sophisticated techniques to circumvent signature-based detec-

tion, a critical aspect in the realm of antivirus solutions becomes the development of methodologies that can effectively identify and thwart novel, unseen threats.

In response to these challenges, the field of intelligent malware detection has gained prominence. This paradigm shift involves the application of machine learning and data mining methods to enhance the capability of malware detection systems. Machine learning algorithms, by analyzing and learning patterns from a diverse training set containing both malicious and benign samples, exhibit a capacity to generalize and discriminate between the two. The utilization of machine learning in this context addresses the inherent limitations of signature-based methods, as it doesn't rely on explicit signatures but rather on learned patterns, enabling the identification of previously unseen or modified malware. In this field, different machine learning methods, such as the naive Bayes method, support vector machines (SVM), and decision trees, have been employed to detect unknown malicious executables. These studies show that machine learning methods are effective for detecting unknown malware. The integration of deep learning methodologies into malware detection systems has garnered significant attention. Among these methodologies, Convolutional Neural Networks (CNNs) have gained prominence for their ability to autonomously discern intricate patterns within data. Initially developed for image recognition, CNNs have demonstrated success in various domains, leading researchers to explore their potential in the landscape of malware detection. The promise of CNN-based deep learning in malware detection lies in its potential to enhance accuracy, reduce false positives, and adapt to the constantly evolving tactics of malware. By training on large datasets containing diverse malware samples, CNNs can effectively learn to identify commonalities and anomalies, enabling them to make informed decisions about the malicious code.

8.1.1 Motivation

The increasing complexity and sophistication of malware, have propelled the need for innovative and adaptive detection methods. Traditional approaches, such as signature-based detection, struggle to keep pace with the dynamic evolution of malware variants. In light of these challenges, the motivation to explore deep learning, particularly Convolutional Neural Networks (CNNs), for malware detection emerges as a compelling solution. CNNs, renowned for their ability to automatically learn intricate patterns from vast datasets, present a promising method to enhance detection accuracy, reduce false positives, and address the intricate

nature of malware.

8.1.2 Contribution

In this chapter, an exploration of Convolutional Neural Network (CNN) models is undertaken, with the specific objective of enhancing malware detection capabilities. The application of various CNN architectures is meticulously examined in the context of their effectiveness in identifying and classifying malware instances. Additionally, an exhaustive experimental study is conducted, encompassing a substantial number of malware samples.

8.2 Background

Deep learning, a subset of machine learning, represents a class of algorithms that utilize neural networks with multiple layers to model intricate patterns in data. This methodology has achieved substantial recognition due to its success across diverse fields such as computer vision, natural language processing, and more recently, cybersecurity, particularly in malware detection.

Convolutional Neural Networks (CNNs) are a specialized form of deep learning model highly effective for analyzing data with a grid-like structure, such as images. CNNs have gained widespread adoption in tasks including image classification, object detection, and image generation, largely due to their ability to autonomously and adaptively learn spatial hierarchies of features from input data.

8.2.1 Neural Networks

A neural network comprises interconnected layers of nodes (neurons), where each connection between neurons has an associated weight, and each neuron applies a non-linear activation function to its input. The basic architecture of a neural network includes an input layer, one or more hidden layers, and an output layer. The effectiveness of deep learning stems from the depth of these networks, i.e., the number of hidden layers.

The learning process in neural networks involves adjusting the weights using a method called backpropagation, which minimizes the error between the

predicted output and the actual target. The error is typically measured using a loss function, such as mean squared error (MSE) for regression tasks or cross-entropy for classification tasks.

Mathematically, the output of a single neuron can be expressed as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{8.1}$$

where:

- y is the output of the neuron,
- f is the activation function,
- w_i are the weights,
- x_i are the input features, and
- b is the bias term.

The figure below illustrates a simple neural network architecture with an input layer, two hidden layers, and an output layer. This diagram demonstrates the flow of information through the network, where each layer applies a transformation to the input data.

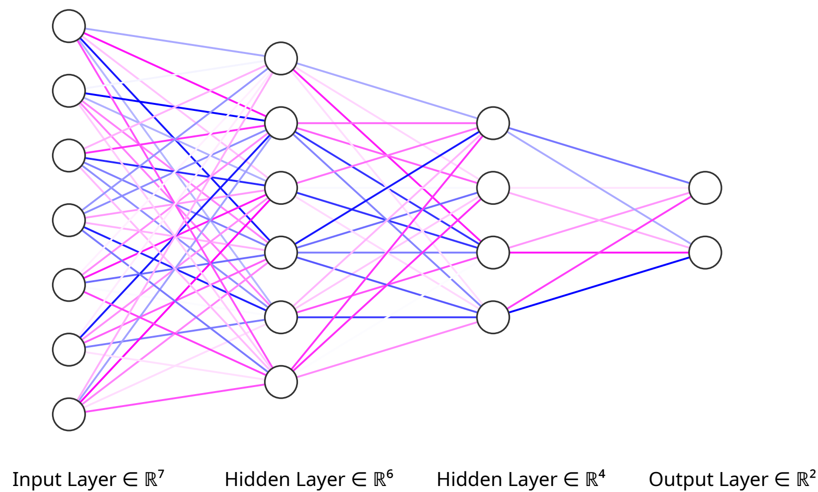

Figure 8-1: Neural Network Architecture

8.2.2 Convolutional Neural Networks (CNNs)

CNNs are specialized neural networks designed to handle data with a grid-like structure, making them highly suitable for image data. They are composed of several types of layers: convolutional layers, pooling layers, and fully connected layers.

Convolutional Layers The convolutional layer is the core building block of a CNN. It applies a set of learnable filters (kernels) to the input image to produce feature maps. Each filter slides (convolves) across the input image, performing an element-wise multiplication and summation operation, which can be mathematically represented as:

$$(I * K)(x, y) = \sum_i \sum_j I(x + i, y + j)K(i, j) \tag{8.2}$$

where:

- I is the input image,
- K is the convolutional kernel,
- $*$ denotes the convolution operation,
- (x, y) are the coordinates of the output feature map.

The result is a set of feature maps that highlight various aspects of the input image, such as edges or textures.

Activation Functions Following the convolution operation, an activation function (e.g., ReLU) is applied to introduce non-linearity into the model. The Rectified Linear Unit (ReLU) function is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{8.3}$$

Pooling Layers Pooling layers reduce the spatial dimensions (width and height) of the feature maps, thereby decreasing the computational load and controlling

overfitting. The most common type of pooling is max pooling, which selects the maximum value from a patch of the feature map. This can be expressed as:

$$y = \max(x_1, x_2, \ldots, x_n) \tag{8.4}$$

Fully Connected Layers After several convolutional and pooling layers, the high-level reasoning in the neural network is done via fully connected layers. These layers take the flattened output of the last pooling layer and produce the final output, typically through a softmax function for classification tasks:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{8.5}$$

where:

- z_i are the input scores to the softmax function.

Training CNNs Training a CNN involves feeding the input data through the network, computing the loss, and using backpropagation to adjust the weights. This process iterates over multiple epochs until the network's performance converges to an acceptable level. Optimizers like Stochastic Gradient Descent (SGD) or Adam are commonly used to update the weights during training.

Example CNN Architecture The following figure illustrates a detailed CNN architecture, showcasing the layers and their configurations. This architecture consists of several convolutional layers followed by pooling layers, culminating in fully connected layers. Each layer extracts increasingly complex features from the input data, eventually enabling accurate classification or prediction tasks.

This specific configuration includes:

- **Input Layer**: The initial layer that receives the raw input data.
- **Convolutional Layers**: Multiple layers (e.g., conv1, conv2, conv3, conv4, conv5) with increasing depth and complexity, responsible for feature extraction.

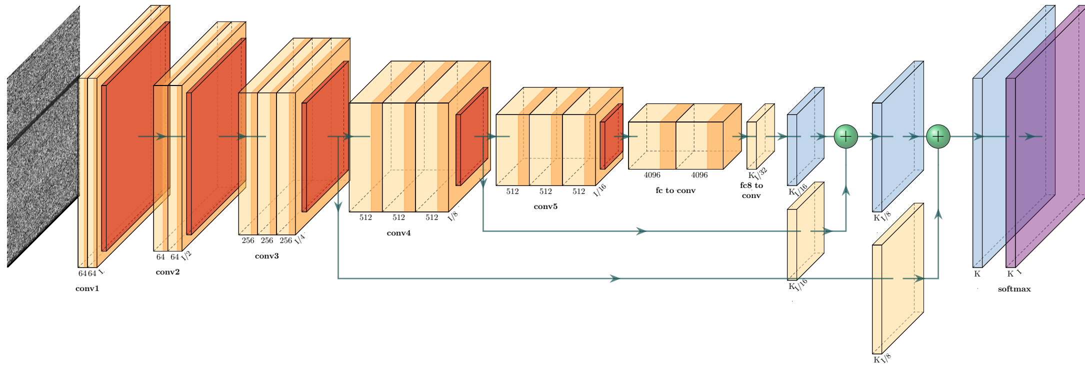

Figure 8-2: Convolutional Neural Network (CNN) Architecture

- **Fully Connected Layers**: Layers that combine features extracted by convolutional layers, including fc to conv and fc8 to conv.

- **Output Layer**: The final layer, typically utilizing a softmax function to generate the classification output.

This detailed architecture demonstrates how CNNs process and learn from input data, progressively capturing and refining features through multiple layers to achieve high-performance outcomes in tasks such as malware detection.

In summary, CNNs' capability to automatically learn hierarchical feature representations makes them exceptionally powerful for image-like data, leading to their adoption in various domains, including malware detection, where binary or hexadecimal representations of files can be treated as images for analysis.

8.3 Problem statement

The problem to be addressed involves the development of an effective and efficient malware classification system utilizing Convolutional Neural Networks (CNNs). Specifically, the task is to design a model capable of accurately distinguishing between benign and malicious files within a given dataset of malware. The detection of malware is defined as a classification problem. For a given malware dataset, the objective is to identify the class label y (goodware or malware) with high precision based on a given set of training instances x_i^{train}. The output is a binary label $y \in \{0, 1\}$, indicating malware or goodware, respectively. The aim is to achieve high detection accuracy (i.e., minimum false alarm rate) for any given input test instance in identifying its class label.

8.4 The Proposed Method

The proposed model for the detection of malware is illustrated in Figure 8-3. The model aims to classify a given data into two categories, malware, and goodware with minimum false alarms.

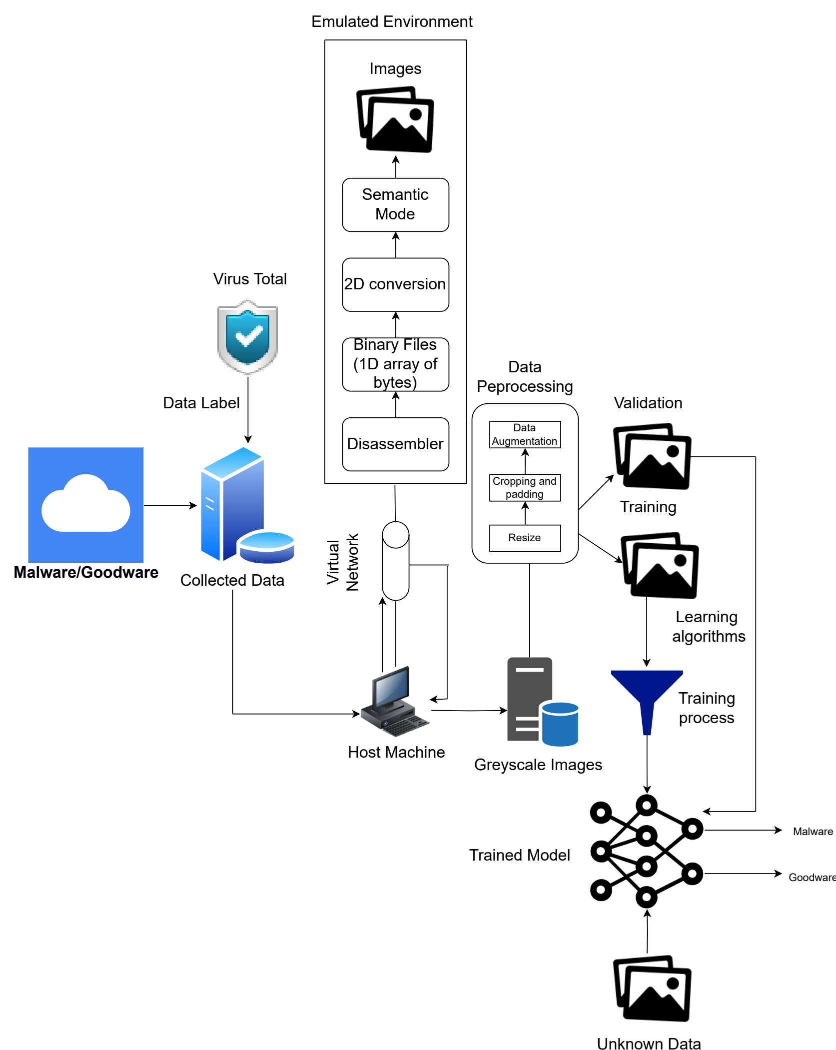

Figure 8-3: Overview of the proposed method

8.4.1 Feature Data Generation

To convert a binary file into an image, we utilize the bytes sequence representing the binary as the pixel values of a grayscale PNG image. A web-based tool is developed to encode any binary file into a lossless PNG format, ensuring fidelity

in the conversion process. This approach allows for the visualization of binary data as images, facilitating interpretation and potential further analysis while preserving the integrity of the original data. The screenshot of the web-based tool used for this conversion process is shown in Figure 8-5. This tool was instrumental in creating a diverse and robust dataset for training the CNN models.

The conversion process begins by determining the length of the binary data, measured in bytes. The binary data is then transformed into a numerical array, with each byte represented as an unsigned 8-bit integer. A special indicator is added to the beginning of the array to mark the length of the data. The necessary dimensions to create a square image are calculated based on the square root of the data length, with any decimal values rounded up. Padding is computed, and zeros are appended to the array to ensure it fits neatly into the square image. The array is then reshaped into a two-dimensional grid, forming the grayscale image.

In grayscale images, each pixel's value represents its brightness, typically depicted as a single number. The standard pixel format for grayscale images is the byte image, where the brightness value is stored as an 8-bit integer, allowing for a range of values from 0 to 255. Conventionally, zero signifies black, while 255 signifies white, with intermediate values representing various shades of gray corresponding to different brightness levels. Finally, the resulting image file is saved in PNG format, facilitating the translation of binary data into an image format for further analysis, as shown in Figure 8-4. The algorithm for grayscale image generation is shown in Algorithm 3.

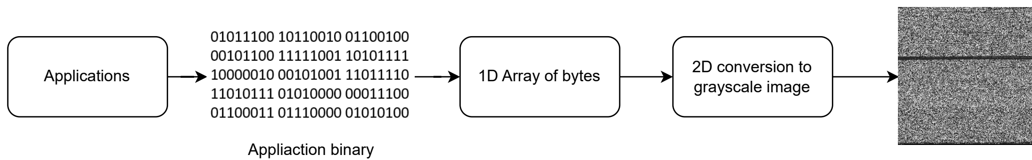

Figure 8-4: Feature Data Generation

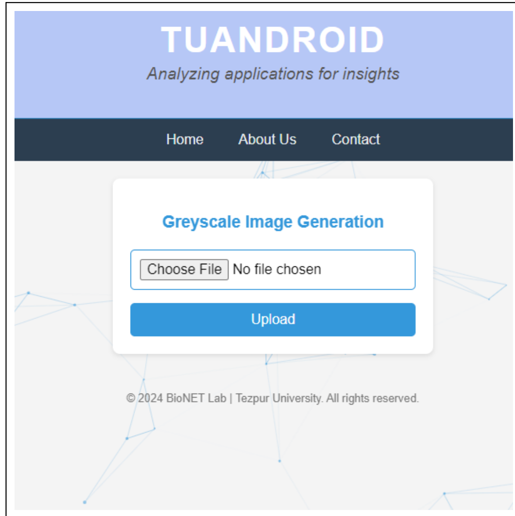

Figure 8-5: Web-based tool for binary to image conversion

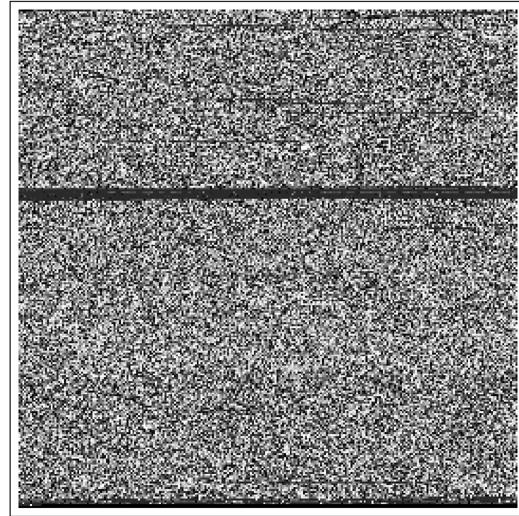

Figure 8-6: Example of Airpush malware visualized as image

Algorithm 3: Generate Grayscale Images from Binary Files

Input: List of Binary files
Output: Grayscale images
initialization;
$input_files \leftarrow$ list all files ;
Function `create_GrayImage`(*input_file, output_dir, imgFormat, imageSize, resampling_filter*):
  $file_content \leftarrow$ read binary file $input_file$;
  $data \leftarrow$ convert $file_content$ to byte array of uint8;
  $image_size \leftarrow$ ceil(sqrt(length(data)));
  $padded_data \leftarrow$ pad $data$ with zeros to length $image_size^2$;
  $image_array \leftarrow$ reshape $padded_data$ to 2D array of size $(image_size, image_size)$;
  $image \leftarrow$ convert $image_array$ to grayscale image;
  $resized_image \leftarrow$ resize $image$ to $(imageSize, imageSize)$ using $resampling_filter$;
  $output_file_path \leftarrow$ generate output path in $output_dir$;
  save $resized_image$ to $output_file_path$;
foreach *file in input_files* **do**
  $output_dir \leftarrow$ generate output directory;
  `create_GrayImage`(*file, output_dir, imgFormat, imageSize, resampling_filter*);
end

The total time complexity of the grayscale image generation algorithm is $O(n + m^2)$, where n represents the size of the input file and m is the dimension

of the output image. Significant steps in the algorithm include resizing the image and saving it. The algorithm processes the input file data linearly, contributing $O(n)$ to the complexity, while the image handling operations, such as resizing, contribute $O(m^2)$. This quadratic time complexity is still more efficient compared to dynamic analysis methods, which involve computationally intensive operations like runtime behavior monitoring and result in much higher time complexity. The grayscale image generation algorithm, therefore, provides a balance between processing efficiency and computational cost, making it suitable for malware analysis tasks.

8.4.2 Data Preprocessing

Data preprocessing is a crucial step before feeding the data into a Convolutional Neural Network (CNN) model, as it significantly enhances model accuracy. One of the primary preprocessing steps is ensuring that all images are of the same dimension. In this case, the images in the dataset are resized to 256x256 pixels. During each epoch, these images are then randomly cropped to the full height or width to ensure consistency in input size. For image resizing, bilinear non-adaptive interpolation is employed, which helps maintain the quality of the images while changing their dimensions. Following resizing, a variety of data augmentation techniques are applied to generate transformed versions of the original data, while preserving their class labels. These techniques include:

- **Rotation**: Rotating the images by a certain angle.
- **Flipping**: Horizontally and/or vertically flipping the images.
- **Perspective Warping**: Applying slight perspective transformations to the images.
- **Brightness Changes**: Adjusting the brightness levels of the images.
- **Contrast Changes**: Modifying the contrast of the images.
- **Noise Addition**: Adding random noise to the images to simulate real-world variability.
- **Resolution Adjustment**: Changing the resolution of the images.

Data augmentation is particularly beneficial when the available data is limited. It increases the variability in the images and acts as a regularizer at

the dataset level, which helps prevent overfitting and improves the generalization ability of the model.

8.4.3 Selection of CNN architectures

In the realm of Convolutional Neural Networks (CNNs), several models such as ResNet, DenseNet, AlexNet, and others have been introduced to handle large volumes of data efficiently. However, the performance of these models can vary significantly depending on the availability and quality of the data, as well as the number of samples. Therefore, to determine the effectiveness of different CNN architectures in the context of malware detection, eight CNN models were considered in this experimental research. The performance of these models was evaluated based on accuracy, precision, recall, and F1-score.

This study includes CNN architectures from four prominent categories: ResNet [99], DenseNet [100], AlexNet [101], and VGG [102]. Specifically, the architectures considered are ResNet18, ResNet34, ResNet50, AlexNet, VGG, DenseNet121, DenseNet161, and DenseNet169. By comparing these models, the aim is to identify the most effective architecture for detecting malware. Table 8.1 provides a comparative overview of various deep learning models in terms of their computational efficiencies.

Table 8.1: Comparison of Model Computational Efficiencies

Model	Inference Time	Training Time	No. of Parameters	Memory
ResNet18	Fast	Moderate	Low	Low
ResNet34	Moderate	Moderate	Medium	Medium
ResNet50	Moderate	High	High	High
DenseNet121	Moderate	High	Medium	High
DenseNet161	Slow	Very High	Very High	Very High
DenseNet169	Moderate	High	High	High
AlexNet	Fast	Short	Moderate	Moderate
VGG	Slow	Long	Very High	Very High

8.4.4 Transfer Learning and Fine-tuning Approaches

Training an entire Convolutional Neural Network (CNN) from scratch is challenging because it requires a relatively large dataset and is very time-consuming due to the need for significant computational and memory resources. Therefore, we

utilize the benefits of transfer learning. In transfer learning, a pre-trained model that has already been trained on a very large dataset from a different domain is fine-tuned for another domain. Figure 8-7 provides an overview of the transfer learning approach.

Generally, the first few layers of a CNN encode low-level features that are common across most computer vision tasks. The later layers, before the final layer, learn more complex features. In transfer learning, these layers are retained and adjusted as needed to solve a specific task. The final fully connected layer of the pre-trained CNN model is replaced with a new fully connected layer that has as many neurons as the number of classes in the new target application. In our approach, we replace the fully connected layer of the pre-trained model with a new fully connected layer tailored to our problem. The last layer contains two neurons, representing the application's two target classes: malware and goodware.

The random weights of this newly introduced layer need to be appropriately adjusted. To maintain the layer weights from the pre-trained network, the network optimizer adjusts the weights of the newly added layer while freezing the previous layers. To determine an optimal range for the learning rate without extensive experimentation, we use a technique known as cyclical learning rate before training. Initially, with all previous layers frozen, we train the newly inserted layer for one epoch to adjust its random weights. Subsequently, the layers are unfrozen, and the entire network is trained for the specified number of epochs.

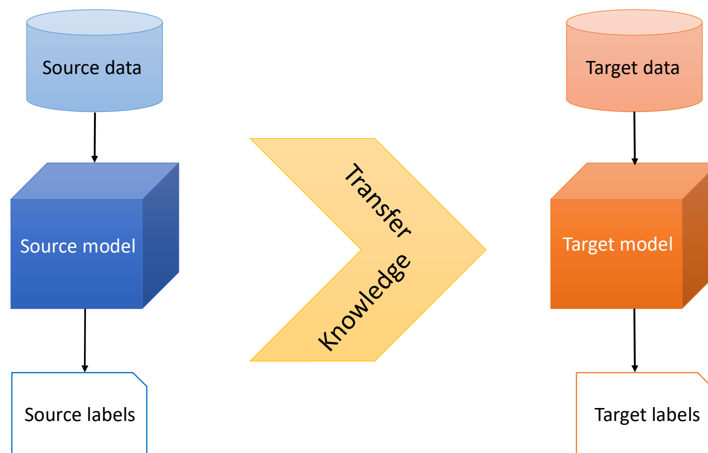

Figure 8-7: Illustration of transfer learning

8.5 Experimental Results

We implement all deep learning models using PyTorch 1.9.0. The experiments are performed on a Dell Precision 7810 Tower with 2x Intel Xeon E5-2600 v3 consisting of 8 cores, 64GB RAM, and NVIDIA Tesla K80 GPU with 12GB VRAM.

8.5.1 Datasets

This section provides a summary of the datasets used in evaluating the proposed method and presents the experimental results. The evaluation is performed using two different datasets. The first dataset is the MALNET-IMAGE TINY dataset [103], which consists of a total of 61,201 images for training, 8,743 images for validation, and 17,486 images for testing.

The second dataset TUANDROMD-X was generated using a specific process detailed in Section 9.4.1. To construct this dataset, 20000 samples of raw malware binaries are obtained from [34]. Additionally, we curated a set of the top 1,000 Android applications from Google Play, which served as representative examples of benign applications. The dataset is categorized into 72 distinct classes, with 71 classes dedicated to various types of malware and one class for goodware. This approach allowed us to create a dataset that encompasses a diverse range of data points, including both malicious and benign software instances. The detailed dataset statistics are presented in Table 8.2. To provide a visual representation of the data distribution, Figure 8-8 illustrates the overall class distribution, while Figure 8-9 shows the top 10 categories within the dataset.

Table 8.2: Dataset Statistics

Characteristic	**Count**
Total Instances	21,000
Malware Instances	20,000
Goodware Instances	1,000
Total Classes	72
Malware Classes	71
Goodware Class	1

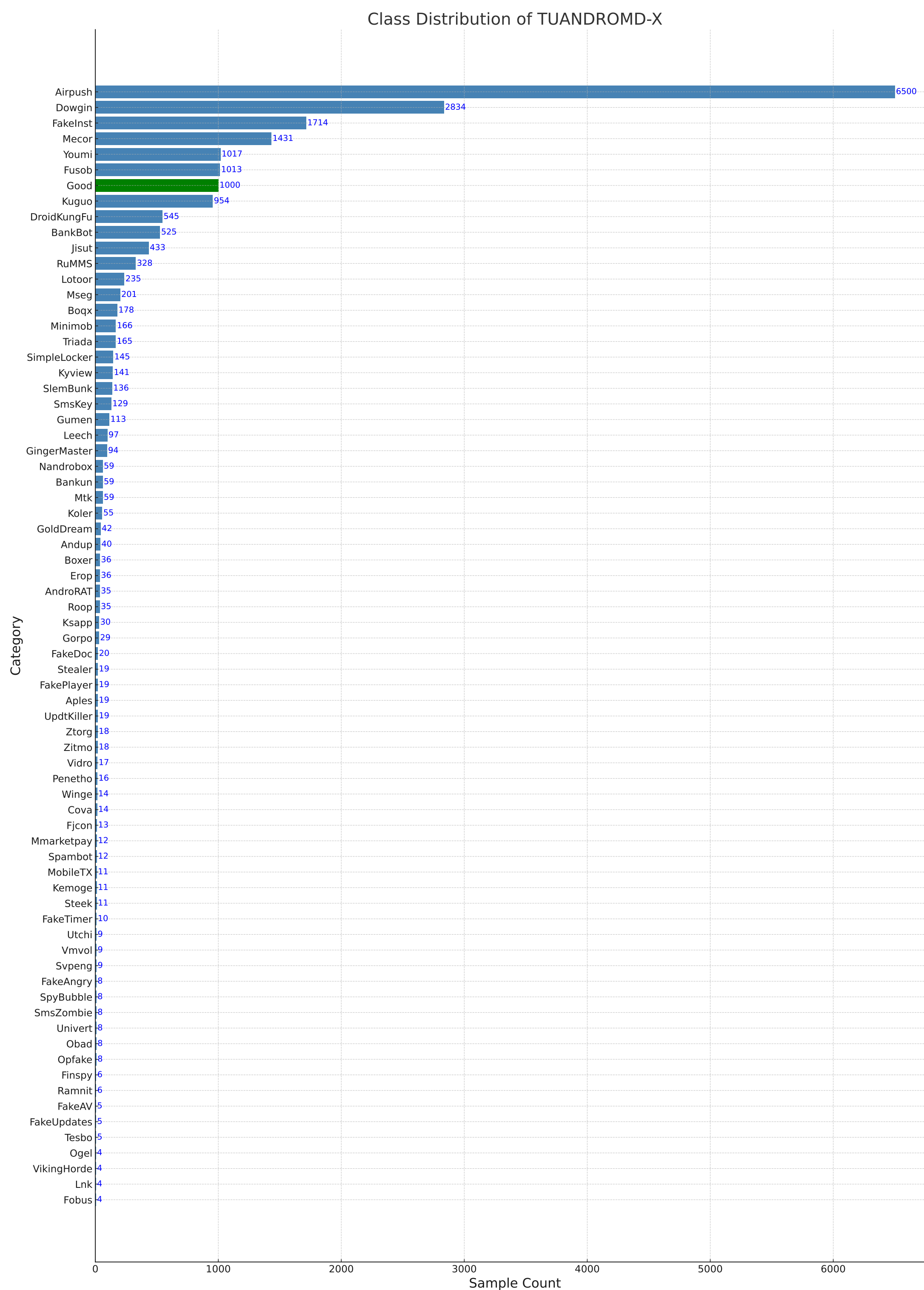

Figure 8-8: Class Distribution of TUANDROMD-X

8.5.2 Results

The training phase of the neural network focused particularly on tuning the weights of the newly added layers. Optimization for these weights was performed using the Adam optimizer and the cross-entropy loss function. Precise hyper-

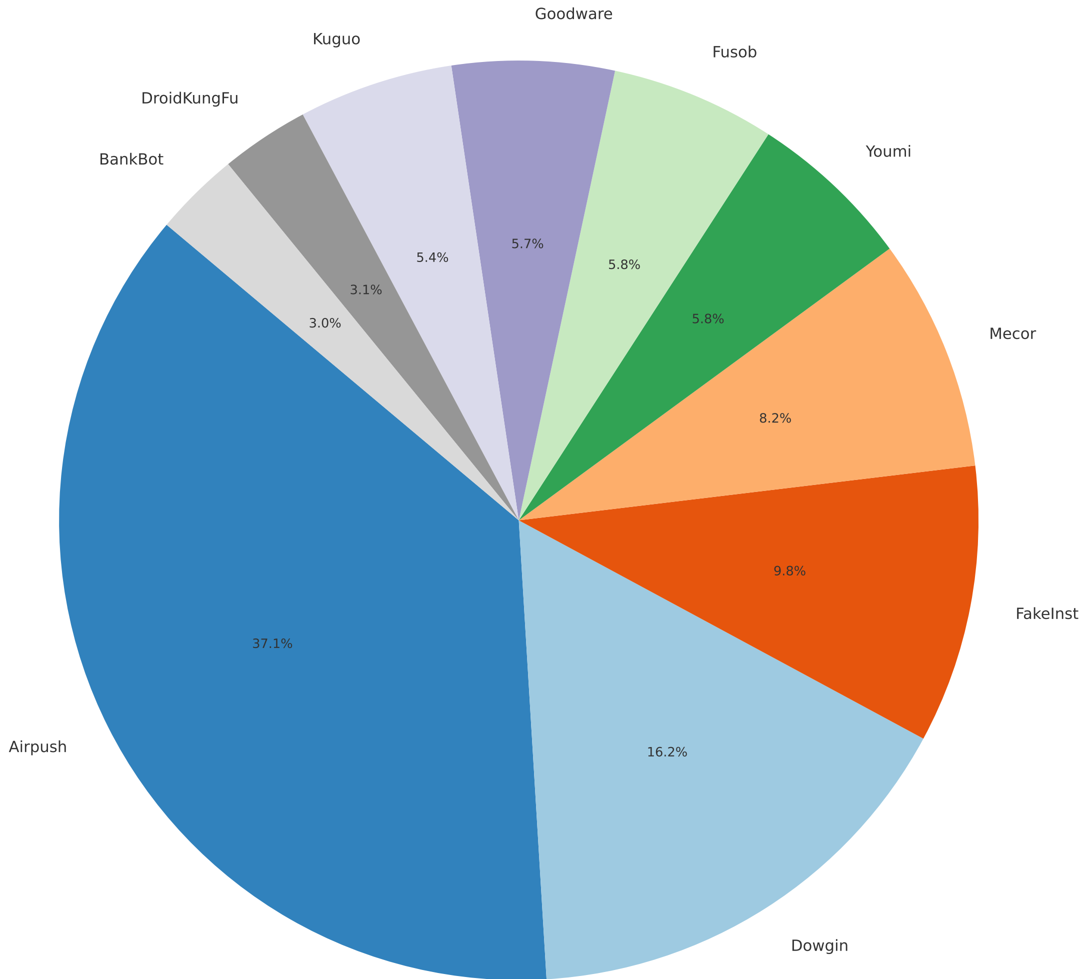

Figure 8-9: Top 10 categories of TUANDROMD-X

parameters were used for configuring the Adam optimizer, including $\beta_1 = 0.98$, $\beta_2 = 0.979$, and $\epsilon = 1 \times 10^{-8}$. These hyperparameter values were obtained through an extensive grid search to ensure optimal performance.

For the optimization of the learning trajectory, a Cyclical learning rate [104] policy targeting a learning rate value of 0.003 was adopted. Consistency across all tests was ensured by standardizing batch sizes to 64. This methodological consistency guarantees fairness and reliability in comparing model performances. The training spanned over 50 epochs, during which the model weights showing the highest validation accuracy were retained.

The performance metrics reported in Table 8.3 for Dataset 1 highlight the efficacy of various deep learning models in malware detection. This table presents an overview of the Accuracy, Precision, Recall, and F1 Score for different models including ResNet18, ResNet34, ResNet50, DenseNet121, DenseNet161,

DenseNet169, AlexNet, and VGG. Among the models evaluated, ResNet50 demonstrates superior performance with an accuracy of 0.963, a precision of 0.943, a recall of 0.948, and an F1 score of 0.954. DenseNet169 also performs well, with an accuracy of 0.960 and the highest F1 score of 0.958, indicating its robustness in handling this dataset. Other models like ResNet18 and ResNet34 also perform very well, with accuracy values of 0.956 and 0.955 only slightly lower than ResNet50 and DenseNet169. DenseNet121 and DenseNet161 show consistent results with accuracy scores of 0.953 and 0.952, respectively. AlexNet and VGG, while performing lower relative to ResNet and DenseNet variants, achieved accuracy scores of 0.936 and 0.944, respectively.

Table 8.4 presents the performance metrics of various deep learning models evaluated on Dataset 2. As shown in Table 8.3, ResNet50 achieves the highest accuracy of 0.945, along with a precision of 0.89, recall of 0.85, and an F1 score of 0.87. DenseNet models (DenseNet161, DenseNet121, and DenseNet169) also exhibit commendable results (Table 8.3). DenseNet161 achieves an accuracy of 0.931, along with a precision of 0.90, a recall of 0.83, and an F1 score of 0.86. DenseNet121 and DenseNet169 show comparable performance, with accuracy scores of 0.924 and 0.919, respectively. Additionally, ResNet18 and ResNet34 perform reasonably, with accuracy scores of 0.912 and 0.897, respectively.

Table 8.3: Performance Metrics of Various Models on Dataset 1

Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**
ResNet18	0.956	0.938	0.965	0.942
ResNet34	0.955	0.932	0.976	0.937
ResNet50	0.963	0.943	0.948	0.954
DenseNet121	0.953	0.951	0.970	0.949
DenseNet161	0.952	0.9489	0.967	0.946
DenseNet169	0.960	0.950	0.962	0.958
AlexNet	0.936	0.927	0.931	0.923
VGG	0.944	0.938	0.947	0.935

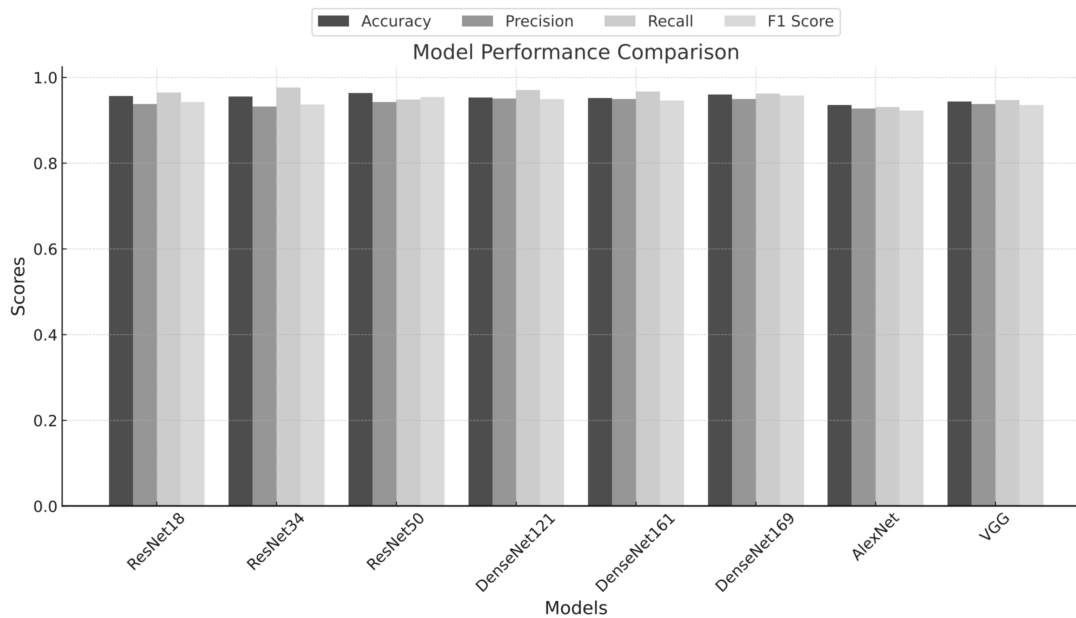

Figure 8-10: Model Performance Comparison on Dataset 1

Table 8.4: Performance Metrics of Various Models on Dataset 2

Model	Accuracy	Precision	Recall	F1 Score
ResNet18	0.912	0.88	0.84	0.86
ResNet34	0.897	0.91	0.81	0.85
ResNet50	0.965	0.89	0.85	0.87
DenseNet121	0.924	0.88	0.84	0.86
DenseNet161	0.931	0.90	0.83	0.86
DenseNet169	0.919	0.89	0.84	0.86
AlexNet	0.908	0.89	0.83	0.86
VGG	0.895	0.91	0.82	0.86

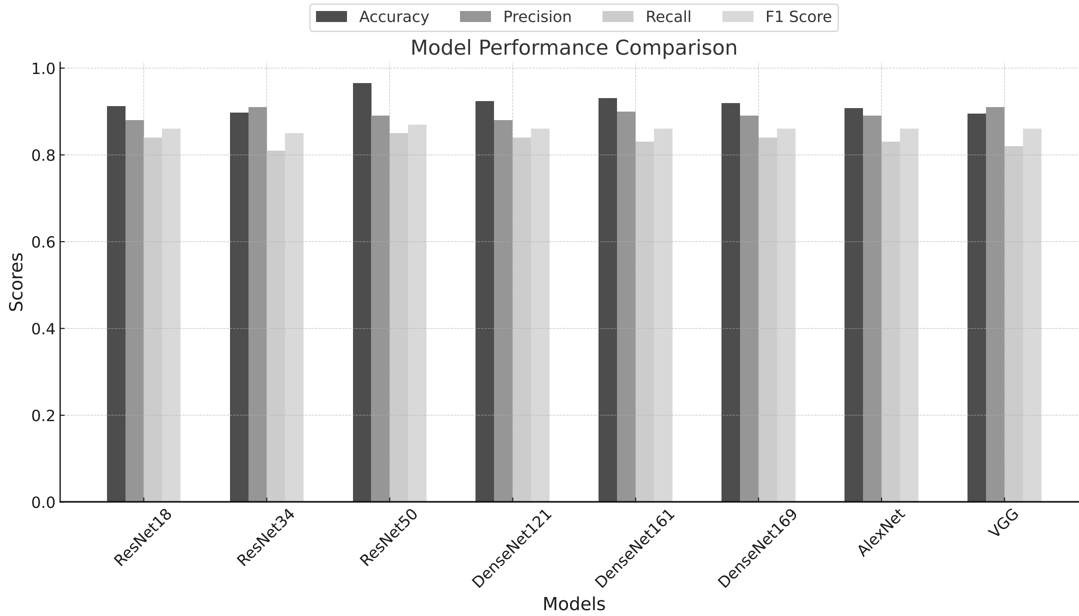

Figure 8-11: Model Performance Comparison on Dataset 2

8.6 Discussion

This study shows the experimentation with Convolutional Neural Network (CNN) models for malware detection. The CNN models demonstrated a notable capability to capture intricate patterns and characteristics from a varied set of malware samples. The achieved accuracy highlights their effectiveness in differentiating between benign and malicious files. Moreover, the exploration of hyperparameter values within the CNN architectures allowed for a nuanced understanding of the impact of these choices on model performance. The fine-tuning of parameters, such as kernel sizes, stride lengths, and filter numbers, proved instrumental in optimizing the accuracy and efficiency of the malware detection models.

This study explores the use of Convolutional Neural Network (CNN) models for malware detection, demonstrating their notable capability to capture intricate patterns and characteristics from a diverse set of malware samples. The high accuracy achieved by these models underscores their effectiveness in distinguishing between benign and malicious files. Additionally, the comprehensive exploration of hyperparameter values within the CNN architectures provided valuable insights into how these choices impact model performance. Fine-tuning parameters such as kernel sizes, stride lengths, and the number of filters proved instrumental in optimizing both the accuracy and efficiency of the malware detection models.