**Winter Semester (2022-23)**

**Information Security Management (CSE3502)**

# Malware Detection using Artificial Intelligence

**J Component Review-3**

**Report**

By

Darsh Kumar(20BCE0301)

Mir Mohammed Zain(20BDS0211)

Khyati Pareek(20BDS0299)

Srishti Sinha(20BDS0329)

**Under the guidance of: Prof. Ruby D.**

## ABSTRACT

As we find ourselves immersed in the digital age, the specter of malware looms large over our technological landscape. With the increasing advancement and complexity of these malicious programs, the standard methods of detection have proven inefficient. As such, this project represents a bold foray into a new frontier of malware detection, leveraging the power of AI-driven techniques to combat these emerging threats. Our exploration delves into innovative methods of analysis, charting the impact of these techniques upon the ever-evolving digital landscape. By forging a path toward deep learning-based malware detection, we offer a glimpse into the future of cybersecurity. Drawing upon a veritable library of both malicious and benign files, we construct a visual lexicon of these programs, using them as the basis for our training data. By harnessing the incredible power of deep learning, our model is able to deftly distinguish between malware and non-malicious files, delivering a novel and efficient approach to detection. Foregoing the need for static or dynamic analysis, we are able to rapidly and accurately identify malware through the use of our unique methodology. This project, in its very essence, contains a major step forward in the ongoing fight against malicious software. It holds the promise of a safer, more secure digital realm, where threats are identified and neutralized with the utmost efficacy.

*Keywords: Malware detection, Neural networks, Deep Learning, Signature intrusion detection .*

## INTRODUCTION

Malware is a growing concern for individuals and organizations, with cyberattacks becoming more sophisticated and frequent. Traditional malware detection methods, such as signature-based detection, have been shown to be inadequate in detecting new or unknown variants of malware. To address this, researchers turned to convolutional neural networks as it is better for malware detection.

CNN is specifically designed for image classification and classification tasks. They use convolution, clustering, and activation functions to extract relevant features from an image. When used for image classification in a malware dataset, a convolutional layer is applied to detect edges or textures in the image, allowing the CNN to learn underlying patterns in the malware dataset. malicious software. The down-sampling features of the clustering layer reduce the computation required, and the activation function allows learned models to become increasingly complex.

Malware detection using the CNN model is a promising approach because it exploits the strengths of neural networks and signature-based detection. The CNN model can analyze a program's behavior and functionality, and then compare it with predefined signatures to detect known malware. This approach can help identify new or unknown malware variants by discriminating patterns in new entries. The proposed system visualizes the malware as an image and trains the classifier based on deep learning methods to maximize accuracy. This approach ensures improved performance, high classification accuracy, reduced computational costs, faster response to threats, and resilience to cloaking techniques. By representing malware as an image, it becomes easier to apply image processing techniques such as feature extraction and data enhancement.

The proposed method using CNN models and image processing provides a promising solution to handle the limitations of existing detection techniques. It provides a more efficient and accurate way to detect malware by visualizing it as an image and leveraging deep learning techniques to extract meaningful features. This approach provide stronger defenses against cyberattacks.

Malware, short for malware, is a major threat to information security. Hackers use malware to gain unauthorized access to computer systems, steal sensitive data, and damage computer networks. As a result, malware detection has become an important task for any organization that uses computers and networks. Lately, there is an increase in use of deep learning techniques for malware detection. One such approach is to use a convolutional neural network to detect malware based on visual features extracted from images of program behavior. CNN is specifically designed for image classification and analysis tasks. They are used to extract relevant features from an image by applying several convolution, clustering and activation functions. The convolution layer helps detect edges or textures in images, allowing the neural network to learn underlying patterns in malware datasets. The down-sampling features of the clustering layer reduce the computation required, and the activation function allows learned models to become increasingly complex.

Traditional malware detection techniques rely on signature-based detection systems that use predefined signatures or hashes to identify known malware. However, these approaches can be circumvented by new or unknown malware variants. Neural networks, on the other hand, can analyze the behavior and functionality of a program and compare them with predefined signatures. This approach leverages the strengths of neural networks and signature-based detection to distinguish patterns in new entries and detect known malware.

Malware detection is crucial for any organization that wants to protect its information systems against threats from malicious hackers. The prevalence of malware on the Internet has grown exponentially, making it essential to have an accurate classifier to identify and prevent malware attacks. Malware detection works as an early warning system for computer security related to malware and cyber attacks, preventing hackers from accessing your computer and preventing information from being stolen. violate.

Existing malware analysis technologies include the use of static and dynamic code analysis, which have their specific strengths. However, static analysis suffers from code obfuscation due to the need to decompress and decode the code, while dynamic analysis can ignore malicious behavior due to inappropriate virtual environments. Both approaches are computationally heavy and time consuming.

To overcome these limitations, the proposed system uses a new approach which is to first visualize the malware as an image and then train the classifier based on deep learning methods to Maximize accuracy. This approach ensures improved performance, high classification accuracy, reduced computational costs, faster response to threats, and resilience to cloaking techniques.

The CNN model is implemented by the suggested system using TensorFlow. The proposed method also makes use of fastai, a library that offers users high-level components that may be utilized swiftly and efficiently to produce the most cutting-edge outcomes in the ML domains. Additionally, it provides low-level components that may be combined and matched to produce new processes.

## LITERATURE SURVEY

| Sr. No. | Name of the transaction/journal/ conference | Major technologies used | Results/ Outcome | Drawbacks |
|---|---|---|---|---|
| [1] | Malware Images: Visualization and Automatic Classification (2017) | To categorize malware photos, the authors employed a deep-learning strategy based on CNNs. To visualize malware pictures and their related class labels, they created the Malware Image Grid (MIG) visualization approach. They evaluated the effectiveness of their suggested strategy using accuracy, precision, recall, the F1-score, and a confusion matrix. | Using a CNN-based classifier, the authors' studies on a sizable collection of malware photos yielded an accuracy of 97.86%. Additionally, they demonstrated how their suggested visualization method (MIG) might offer helpful insights into the properties of malware pictures and the classes to which they belong. | The quality of the virus photos, which may differ depending on the source and the technique used to acquire them, is one possible disadvantage of the suggested strategy. Another drawback is that because the suggested method is based on a pre-defined collection of classes, it might not be able to handle novel or undiscovered forms of malware. |
| [2] | Use of Data Visualization for Zero-Day Malware Detection (2018) | For machine learning system, the researchers employed a collection of malicious and good files. Each file's characteristics were generated by the algorithm and then shown as scatter plots and heatmaps. To find patterns | Both known and undiscovered malware files may be identified with high accuracy using the method. The visualisations made it easier to spot key malware-indicating | The method is computationally demanding, and creating the visualisations and analysing the data both need a sizable amount of computer power. The method depends on the availability of a |

| | | | | |
|---|---|---|---|---|
| | | that might differentiate between malicious and good files, the visualizations were examined. | patterns and characteristics, such high entropy and odd code structure. | dataset containing benign and malicious files for the machine learning algorithm's training, which might be difficult to come by in real life. |
| [3] | Malware Detection Based on Code Visualization and Two-Level Classification (2021) | A new malware detection system is proposed that utilizes a two-level Artificial Neural Network (ANN) combined with Ensemble Bagged Trees method for improved classification accuracy. | The ANN method is faster, up to 30 times faster, than the ensemble classification method and accuracy gets better with bigger and comprehensive datasets. This makes it a promising solution for implementation in antivirus software, | The continuously growing pool of available knowledge presents a challenge for the complexity and efficiency of methods, as it greatly affects their execution time. |
| [4] | An Evaluation of Image-based Malware Classification Using Machine Learning (2020) | This study includes the evaluation of multiple classifiers including k-NN, Naive Bayes, Support Vector Machine, and GIST and Convolutional Neural Network along with the Gabor wavelet-kNN method. | Among the test taken, k-NN had the highest accuracy on the datasets while Naive Bayes showed limitations in its performance for image-based malware classification. | K-NN relies on the Euclidean distance to determine the nearest neighbors, but when the number of pixels is too high it can lead to confusion rather than improving accuracy. |

|  |  |  |  |  |
|---|---|---|---|---|
| [5] | Visualized Malware Classification Based-on Convolutional Neural Network (2019) | CNN and image converter are the proposed malware classifier, which uses a combination of image processing and machine learning techniques to accurately identify and classify malicious code. | The study suggests that the proposed malware family classification model is effective in providing effective defense techniques. By quickly classifying malware families without executing malicious code or conducting in-depth code analysis, the model is well-suited for the fast-paced reality. | The proposed method relies on the use of (CNNs), which can become computationally heavy as number of malicious code categories increases. This means that higher hardware performance is required, making it challenging to use the method for large-scale analysis. |

*Table 1: Literature Survey*

[6] We examine artificial intelligence (AI) methods for identifying and preventing malware, underscoring the importance of looking to the future for solutions. Researchers benefit from this information as they investigate AI-powered malware detection and prevention.

CNN, RNN, LSTM, and autoencoders are only some of the deep learning techniques discussed in this article's [7] discussion of malware detection. Since LSTM and autoencoders have shown some success in Android malware detection, it is clear that more study of deep learning is necessary to make progress in this area.

Using picture modification and machine learning classifiers, [8] researchers present a computer vision-based method for malware detection. The method outperforms other methods by a wide margin, with an accuracy of 97.01 percent.

Using static analysis, dynamic analysis, and the EfficientNet B3 model for malware detection, [9] a two-layer hybrid malware detection system (2-MaD) is presented for IoT malware identification.

In [10] we present IoT-IDCS-CNN, an IoT attack detection and classification system based on convolutional neural networks. With the help of convolutional neural networks (CNNs), HPC, and parallel processing, it is able to get an accuracy of 99.3% in binary classification and 98.2% in multiclass classification.

To better understand the mechanics behind machine learning algorithms and their conclusions, [11] a framework is offered to analyse the generalizability and functionality of the MalConv architecture.

[12] For those just getting started with malware detection using deep learning, a survey of the literature reveals that convolutional neural networks (CNNs), long short-term memories (LSTMs), deep belief networks (DBNs), and autoencoders are some of the most popular methods.

Through malware visualisation, feature selection, and preprocessing, [13] researchers use deep learning techniques like CNNs and LSTM to create a highly accurate malware detection model.

[14] We present a location-based hashing, Apache Spark-based, big data and machine learning malware detection solution. The model improves detection speed while maintaining 99.8 percent accuracy.

[15] In this paper, we offer an Android malware detection approach based on autoencoders and five feature sets, and we show that this method can accurately identify malware.

[16] Using grey-scale image representation and autoencoder networks, we describe a novel malware detection strategy that achieves 96% accuracy and an F-score of roughly 96%.

With the help of AI planning, the authors of Paper [17] get to the bottom of the hypothesis exploration problem in malware detection. The authors frame the issue of hypothesis generation as one of AI planning, with time-dependent objectives and the associated costs of possible courses of action. In this study, we describe a strategy for integrating a modern planner into grounded research and establish the idea of "plausibility" for hypotheses in the context of flawed observations. Experimentally, the proposed method outperforms prior art by a wide margin.

The methods of SVM, KNN, LDA, LSTM, CNN-LSTM, and autoencoders, among others, are used in paper [18] to detect malicious Android-based assaults. SVM and LSTM models show great accuracy in identifying malware on Android applications, and the cybersecurity solution is assessed using two benchmark datasets.

In Paper [19], the authors present a methodology for improving the accuracy of malware detection systems that use machine learning. The study recommends employing adversarial training and producing adversarial malware pictures to strengthen classification models' resistance to attacks from malicious actors.

A literature review of deep learning and machine learning techniques for ransomware detection is presented in paper [20]. The study's goal is to find the holes in current methods and investigate how they could be patched. This study also explores how machine learning and deep learning can be used to identify zero-day threats and decipher ransomware's tactics.

In order to make malware detection more robust, paper [21] describes a method for creating adversarial malware images and retraining classification systems. The work improves the attack resilience of multiple categorization models by applying adversarial training.

In Paper [22], the authors use a variety of machine learning and deep learning algorithms to identify malicious attacks on Android mobile devices, including SVM, KNN, LDA, LSTM, CNN-LSTM, and autoencoders. The results show that the SVM, LSTM, and CNN-LSTM algorithms are efficient at detecting malware on Android.

This paper [23] examines the state of malware research, feature extraction methods, and classification/clustering algorithms utilised in advanced malware detection systems. Data mining approaches for malware detection are discussed, along with the difficulties and complications that arise from applying them, and predictions are made regarding the development of malware in the future.

Using kNN classification and a VP tree with a similarity hash, the authors of Paper [24] offer a new method for detecting malicious files. The method was designed to improve upon existing pattern-based antivirus and AI-based malware detection systems by speeding up the detection process and increasing the detection rate.

In order to improve the safety of Data Centres and Smart Cities, the authors of paper [25] introduce pAElla, an AI-driven edge computing and high-resolution power usage-based technique. The technique uses autoencoders and power spectral density measurements to detect malware in real time on an IoT-based monitoring system operating outside of the band. According to the findings, pAElla outperforms State-of-the-Art methods in both accuracy and malware detection breadth.

In paper [26], the authors propose combining permission and API calls with machine learning approaches to identify potentially malicious Android applications. The strategy is based on elementary static analysis and is universally applicable to mobile programmes.

Using both static and dynamic malware analysis techniques, the authors of Paper [27] provide a framework for AI-based ransomware detection and introduce a detection tool (AIRaD). Maximum accuracy was 99.54 percent, with a false-positive rate of just 0.005 percent.

In order to detect malware, the authors of paper [28] use LSTM to model traffic data from networks as a series of flows. Both new families of malware and malware inside 50 connection flows were detected with a high degree of accuracy.

In their paper, [29], the authors probe the viability of using LightGBM to categorise malware statically. The suggested method outperforms the standard log loss function on the EMBER dataset, with an AUC of 0.979, thanks to the use of a modified log loss function to optimise learning.

In paper [30], the authors offer a method for detecting ransomware using artificial intelligence and a combination of static and dynamic malware analysis. This research uses NLP, ARM, and ML classifiers to process characteristics extracted from dynamic link libraries (DLLs), functions (function calls), and assemblies. Using support vector machines and the Adaboost with J48 algorithms, the proposed model obtained a remarkable 99.54% accuracy with a false-positive rate of 0.005%.

# SYSTEM DESIGN

## Architecture

The proposed system consists of two major script files: cnn-model and exe-to-img, with an auxiliary script file: extract-exe. The main function of each script file are as follows:

1. **extract-exe:** It is an auxiliary script that copies exe files present in a directory over to a target directory. Its functioning is done in 3 major steps that loop over the directory recursively:
    a. Browse directory to get file list
    b. Iterate through file list to get executables list
    c. Copy executables in the list to target directory
2. **exe-to-img:** It is a main script that converts all executables into images of equal sizes that can be used as dataset for the CNN model. The steps of execution of the script are as follows:
    a. Browse directory to get file list
    b. For each executable in file list, execute the following steps:
        i. Get bytecode data of executable through standard python file handling functions
        ii. Re-structure data into groups of 3 bytes each to form pixel data
        iii. Re-shape grouped data into 512x512 dimension array. Ignore all overflowing bytes, and pad underflowing bytes with zeros
        iv. Create png image using standard python png library functions
        v. Save png image to target directory
3. **cnn-model:** It is the primary script that hosts the CNN model. There are three main environment variables that store the directory locations of the training dataset, testing dataset and saved model respectively. In order to make training and handling the model with ease, two types of execution are implemented:
    **a. Refresh mode:** The script ignores the existing model in the directory and overwrites it with a new untrained model. This mode is useful when experimenting with model architectures and dataset sizes. It is recommended to run the training option as the first operation to get accurate results during predictions.
    **b. Continue mode:** The script loads the existing model in the directory to “continue” the operation of the program as if no exit command was given in the middle. This mode is useful in dividing longer training sequences over separate execution passes, and in running the program to directly get prediction results.

    The script consists of a main menu that branches into 3 execution paradigms:

    **a. Train:** This option is used to train the CNN model. It follows these steps to complete one pass of training cycle:
        i. Load training dataset
        ii. Read number of epochs from the user

        iii. For each epoch, follow these steps:
            1. Iterate through the entire training dataset and execute the following steps for each image in dataset:
                a. Make a prediction for the image
                b. Compare result to actual values in the dataset
                c. Modify model weights according to error difference between predicted actual values
            2. Report epoch progress to user

b. **Predict:** This option is used to get a prediction from the CNN Model. The following main stages of the prediction cycle are completed in this option:
    i. Parsing testing dataset
    ii. Predicting on images present in the dataset
    iii. Comparing predictions with actual values in the dataset to get error points
    iv. Calculating performance metrics

c. **Edit Environment Variables:** There are three main environment variables used by the script file. This option is used to edit the variables to desired values. The variables are as follows:
    i. Training Dataset directory
    ii. Testing Dataset directory
    iii. Saved Model directory

The interactions between the scripts are shown in the following figure:

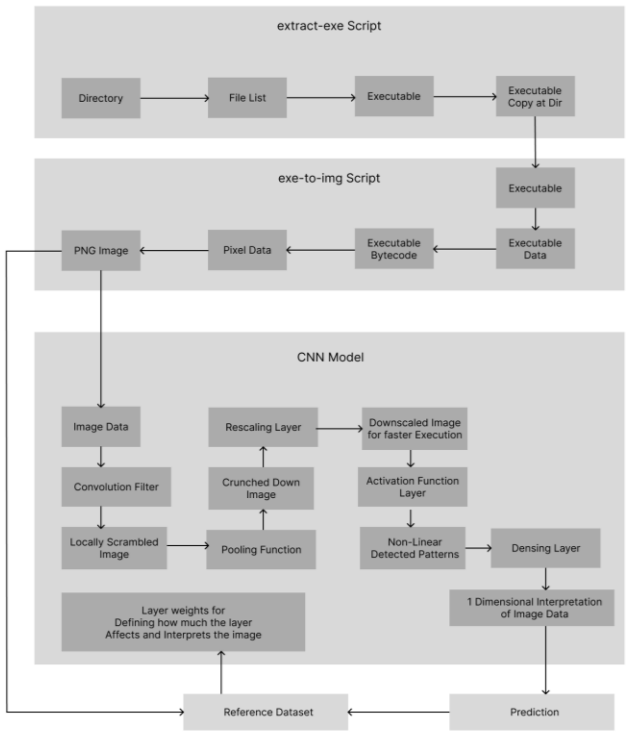


*Fig 1 : Architecture Diagram of the proposed model*

## Libraries used

1. **TensorFlow**: TensorFlow is a freely available software library that empowers developers to create and train machine learning models. It was originally created by Google and has found widespread use in diverse fields, ranging from image and speech recognition to natural language processing and robotics. It is popular both in research and industry for its ability to handle complex computations and its ease of use.
2. **Keras**: Keras is a comprehensive library that offers an array of pre-designed layers to developers, including convolutional layers suitable for processing images, recurrent layers ideal for sequential data, and dense layers for general-purpose computation. In addition, Keras provides an array of training tools, including various optimization algorithms, loss functions, and metrics, that developers can specify while building their models to achieve the desired training outcomes.
3. **ArgParse**: ArgParse is a python standard library used for parsing command line arguments and options. It is very useful in creating command line interfaces (CLI) that accept various options, arguments and sub-commands. In this project, ArgParse is used to give a CLI to exe-to-img, and specify startup mode for cnn-model.

## Control Flow

In our proposed workflow, the user first converts the executables in his specified directory to images using the exe-to-img script provided in our solution. Segregation of the executables into training, validation and testing dataset is made easier using the auxiliary extract-exe script, which copies the executable files in a user-defined directory to a target directory. The segregation of data using extract-exe script, if applied, needs to be applied before conversion to images. The finally segregated dataset is used by the cnn-model script that trains the CNN Model on the training and validation datasets. The user can verify the operation of the model through the cnn-model script's "Make Prediction" option.

A step-by-step detailed process is as follows:

1. Extract exe files using the extract-exe script. Command:
   python3 extract-exe.py

2. Convert exe files to images using the exe-to-img script. Command:
   python3 exe-to-img --folder [Source Directory] --target [Target Directory] [--benign | --malware]

3. Run cnn-model script
   python3 model.py [--refresh | --continue]
   The following options can be selected in cnn-model main menu in no particular order:

   a. Train model using train model option, execute it first if running in refresh mode
   b. Test model performance using make prediction option
   c. Edit environment variables if images are stored in locations differing from defaults:
      i. Model location: ./.model/
      ii. Training and validation dataset location: ../Generated Images/

iii. Testing dataset location: ../Equalised Images/

The flow of control can be visualised using the following figure:

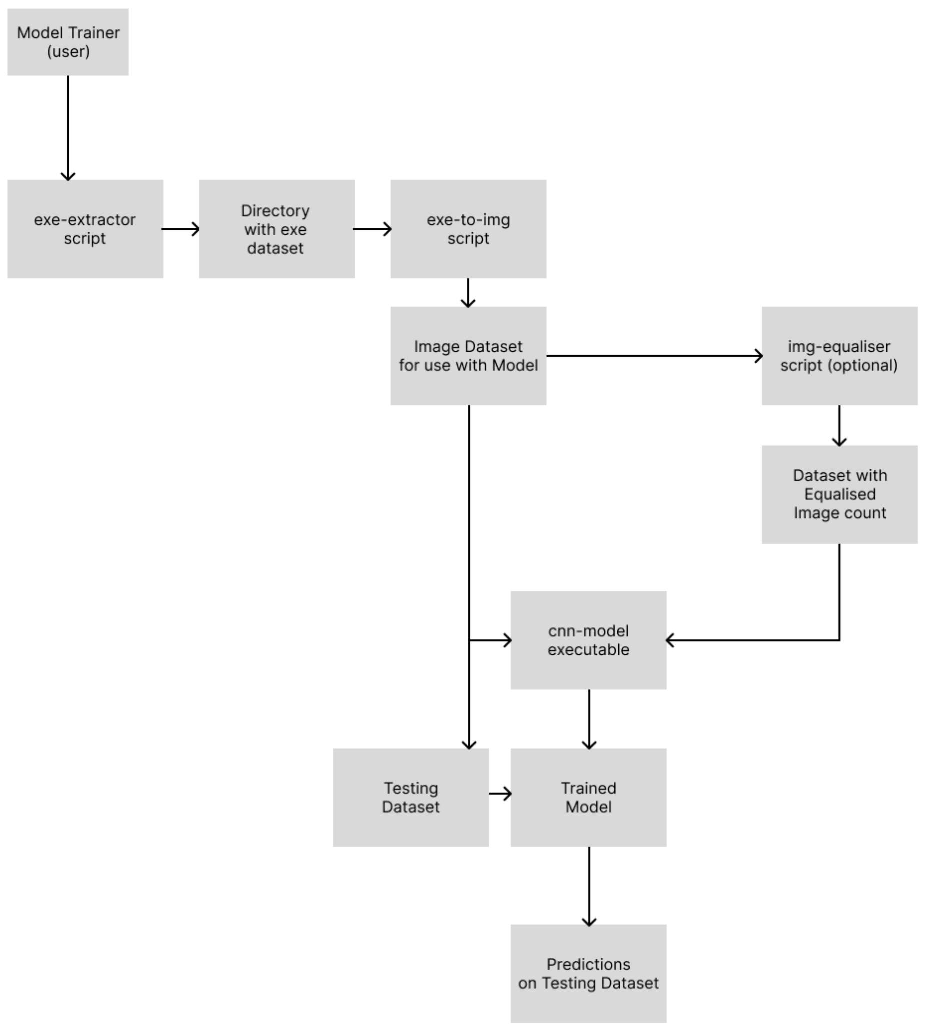
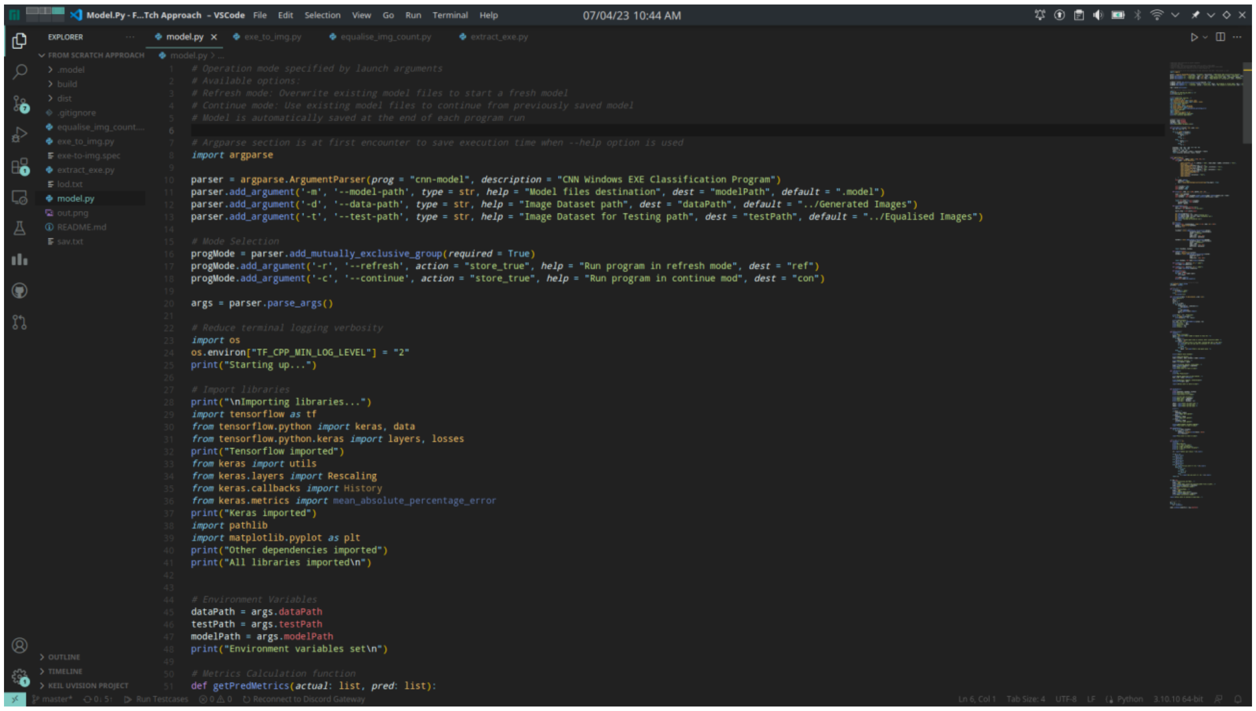

Model Trainer (user)
exe-extractor script
Directory with exe dataset
exe-to-img script
Image Dataset for use with Model
img-equaliser script (optional)
Dataset with Equalised Image count
cnn-model executable
Testing Dataset
Trained Model
Predictions on Testing Dataset

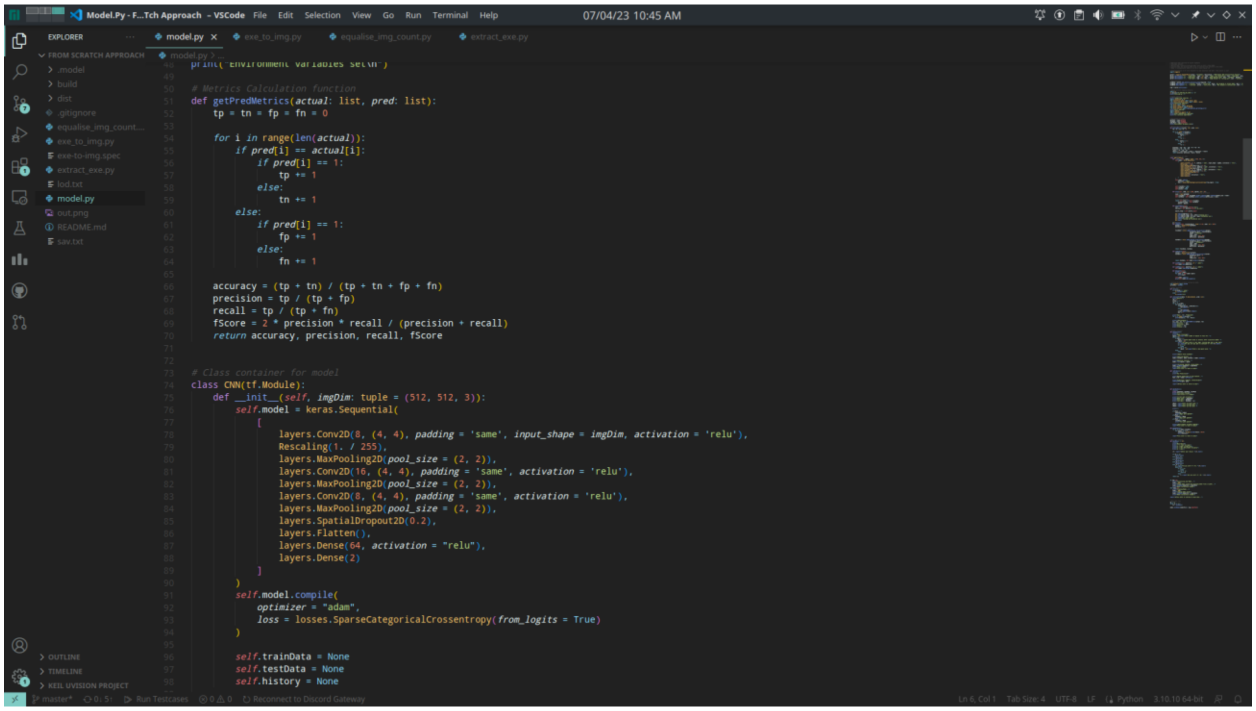
*Fig 2 : Control Flow*

# IMPLEMENTATION

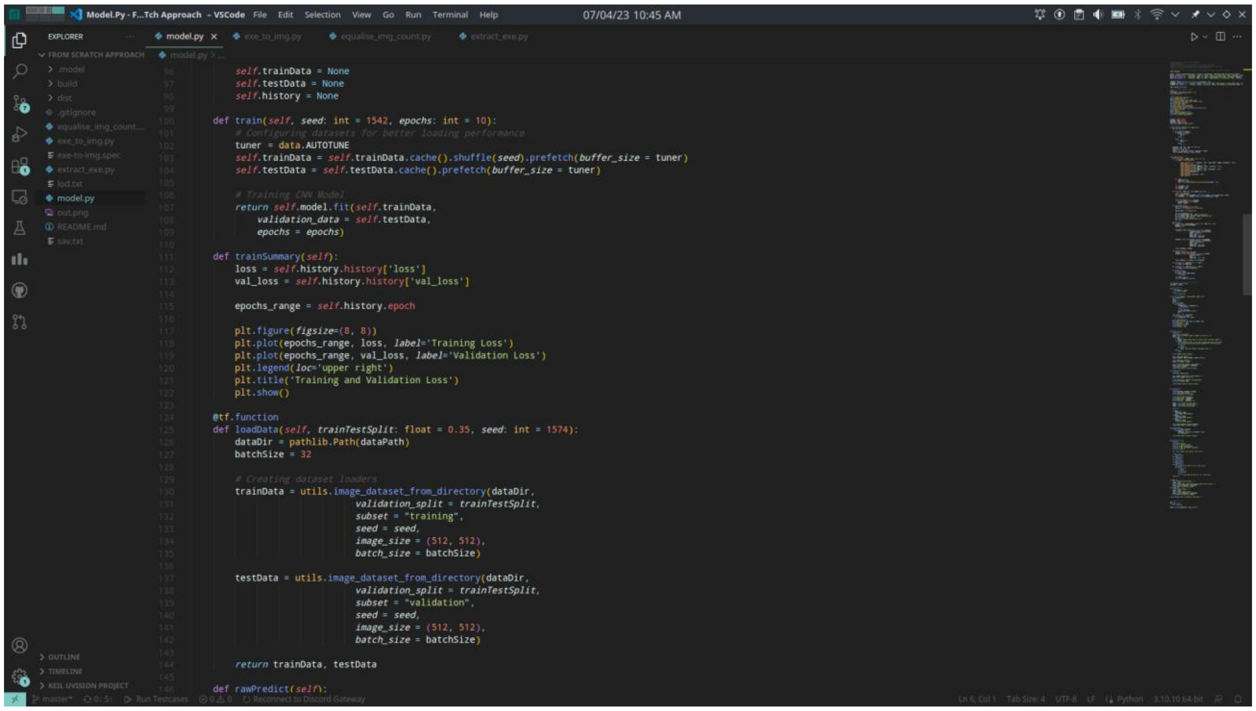
## cnn-model



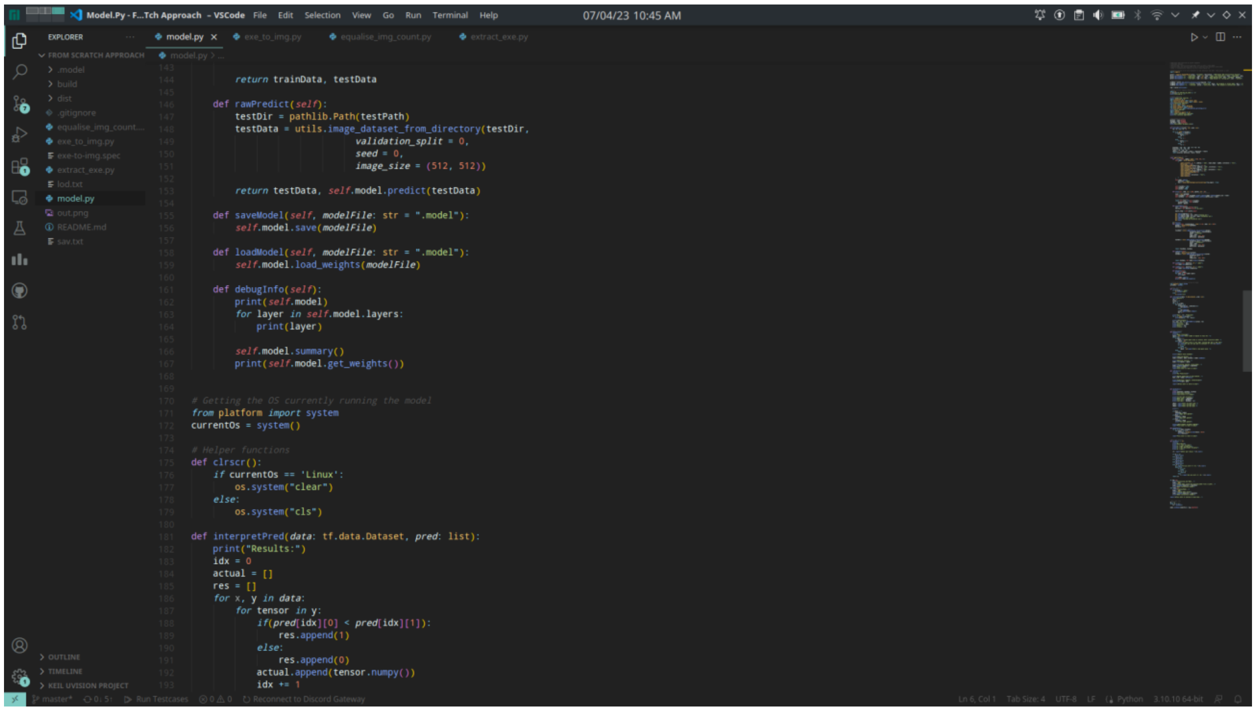
```python
# Operation mode specified by launch arguments
# Available options:
# Refresh mode: Overwrite existing model files to start a fresh model
# Continue mode: Use existing model files to continue from previously saved model
# Model is automatically saved at the end of each program run

# Argparse section is at first encounter to save execution time when --help option is used
import argparse

parser = argparse.ArgumentParser(prog = "cnn-model", description = "CNN Windows EXE Classification Program")
parser.add_argument('-m', '--model-path', type = str, help = "Model files destination", dest = "modelPath", default = ".model")
parser.add_argument('-d', '--data-path', type = str, help = "Image Dataset path", dest = "dataPath", default = "../Generated Images")
parser.add_argument('-t', '--test-path', type = str, help = "Image Dataset for Testing path", dest = "testPath", default = "../Equalised Images")

# Mode Selection
progMode = parser.add_mutually_exclusive_group(required = True)
progMode.add_argument('-r', '--refresh', action = "store_true", help = "Run program in refresh mode", dest = "ref")
progMode.add_argument('-c', '--continue', action = "store_true", help = "Run program in continue mod", dest = "con")

args = parser.parse_args()

# Reduce terminal logging verbosity
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"
print("Starting up...")

# Import libraries
print("\nImporting libraries...")
import tensorflow as tf
from tensorflow.python import keras, data
from tensorflow.python.keras import layers, losses
print("Tensorflow imported")
from keras import utils
from keras.layers import Rescaling
from keras.callbacks import History
from keras.metrics import mean_absolute_percentage_error
print("Keras imported")
import pathlib
import matplotlib.pyplot as plt
print("Other dependencies imported")
print("All libraries imported\n")


# Environment Variables
dataPath = args.dataPath
testPath = args.testPath
modelPath = args.modelPath
print("Environment variables set\n")

# Metrics Calculation function
def getPredMetrics(actual: list, pred: list):
```



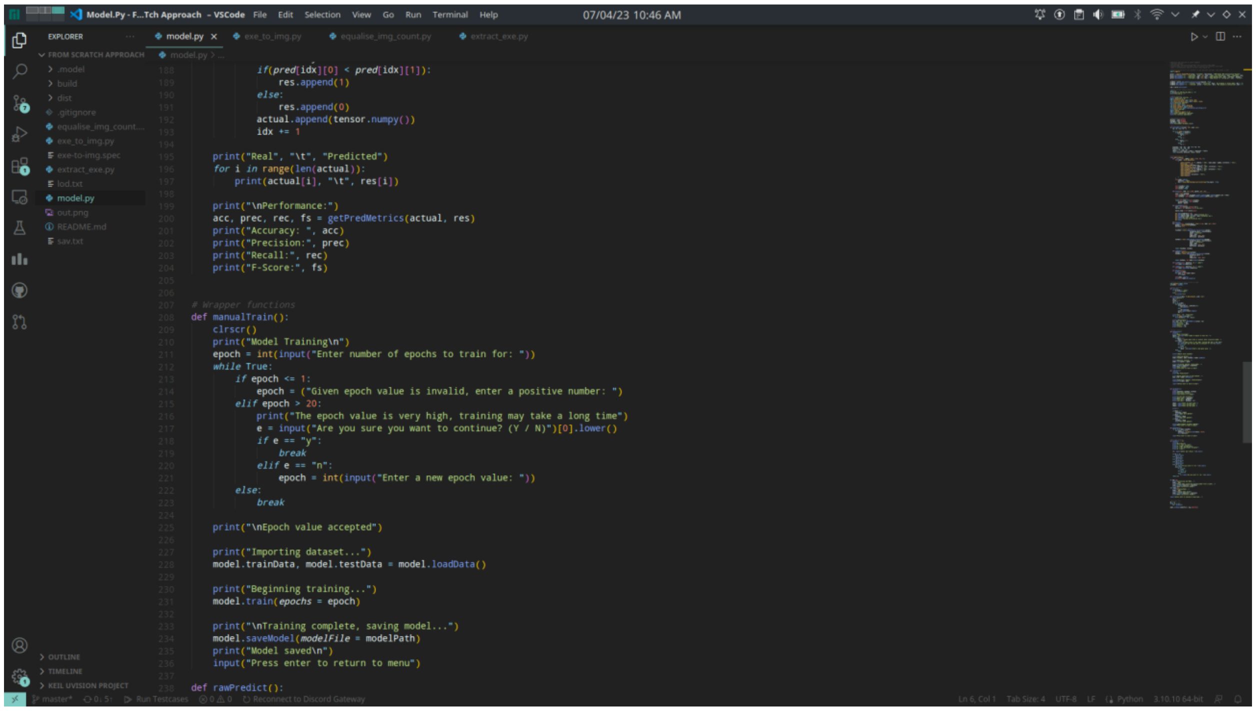
```python
print("Environment variables set\n")

# Metrics Calculation function
def getPredMetrics(actual: list, pred: list):
    tp = tn = fp = fn = 0

    for i in range(len(actual)):
        if pred[i] == actual[i]:
            if pred[i] == 1:
                tp += 1
            else:
                tn += 1
        else:
            if pred[i] == 1:
                fp += 1
            else:
                fn += 1

    accuracy = (tp + tn) / (tp + tn + fp + fn)
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    fScore = 2 * precision * recall / (precision + recall)
    return accuracy, precision, recall, fScore


# Class container for model
class CNN(tf.Module):
    def __init__(self, imgDim: tuple = (512, 512, 3)):
        self.model = keras.Sequential(
            [
                layers.Conv2D(8, (4, 4), padding = 'same', input_shape = imgDim, activation = 'relu'),
                Rescaling(1. / 255),
                layers.MaxPooling2D(pool_size = (2, 2)),
                layers.Conv2D(16, (4, 4), padding = 'same', activation = 'relu'),
                layers.MaxPooling2D(pool_size = (2, 2)),
                layers.Conv2D(8, (4, 4), padding = 'same', activation = 'relu'),
                layers.MaxPooling2D(pool_size = (2, 2)),
                layers.SpatialDropout2D(0.2),
                layers.Flatten(),
                layers.Dense(64, activation = "relu"),
                layers.Dense(2)
            ]
        )
        self.model.compile(
            optimizer = "adam",
            loss = losses.SparseCategoricalCrossentropy(from_logits = True)
        )

        self.trainData = None
        self.testData = None
        self.history = None
```



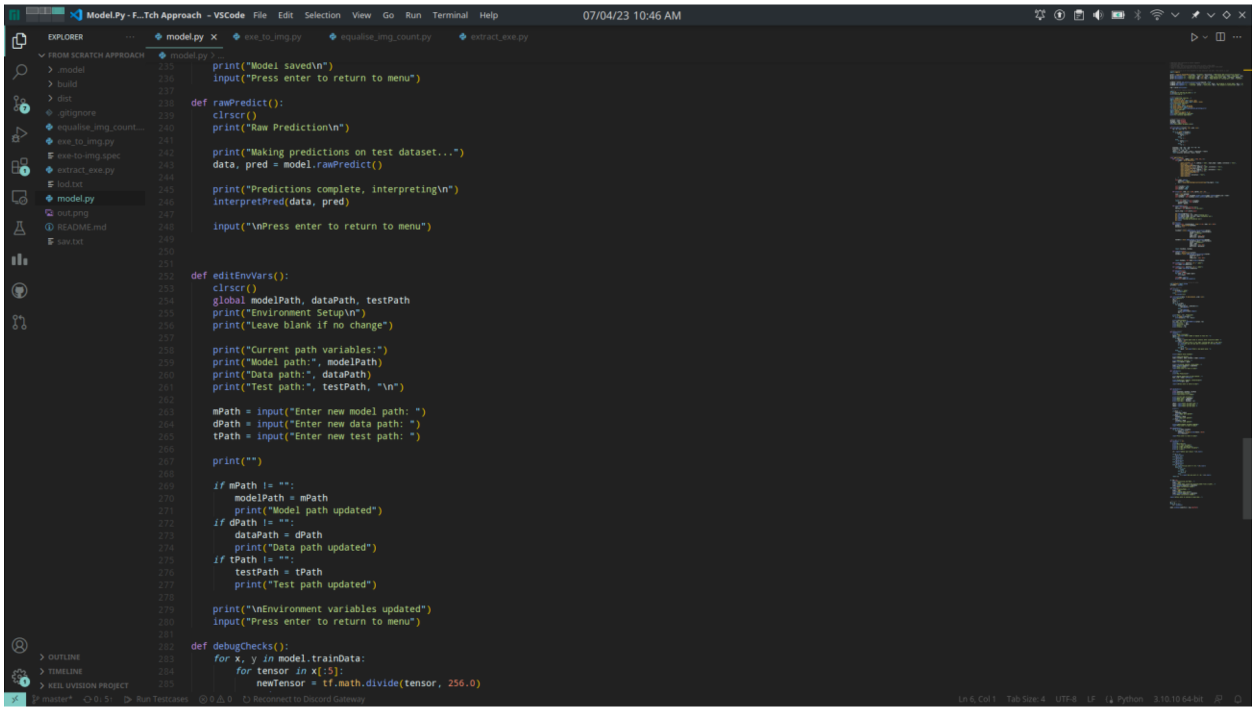
```python
        self.trainData = None
        self.testData = None
        self.history = None

    def train(self, seed: int = 1542, epochs: int = 10):
        # Configuring datasets for better loading performance
        tuner = data.AUTOTUNE
        self.trainData = self.trainData.cache().shuffle(seed).prefetch(buffer_size = tuner)
        self.testData = self.testData.cache().prefetch(buffer_size = tuner)

        # Training CNN Model
        return self.model.fit(self.trainData,
            validation_data = self.testData,
            epochs = epochs)

    def trainSummary(self):
        loss = self.history.history['loss']
        val_loss = self.history.history['val_loss']

        epochs_range = self.history.epoch

        plt.figure(figsize=(8, 8))
        plt.plot(epochs_range, loss, label='Training Loss')
        plt.plot(epochs_range, val_loss, label='Validation Loss')
        plt.legend(loc='upper right')
        plt.title('Training and Validation Loss')
        plt.show()

    @tf.function
    def loadData(self, trainTestSplit: float = 0.35, seed: int = 1574):
        dataDir = pathlib.Path(dataPath)
        batchSize = 32

        # Creating dataset loaders
        trainData = utils.image_dataset_from_directory(dataDir,
                              validation_split = trainTestSplit,
                              subset = "training",
                              seed = seed,
                              image_size = (512, 512),
                              batch_size = batchSize)

        testData = utils.image_dataset_from_directory(dataDir,
                              validation_split = trainTestSplit,
                              subset = "validation",
                              seed = seed,
                              image_size = (512, 512),
                              batch_size = batchSize)

        return trainData, testData

    def rawPredict(self):
```



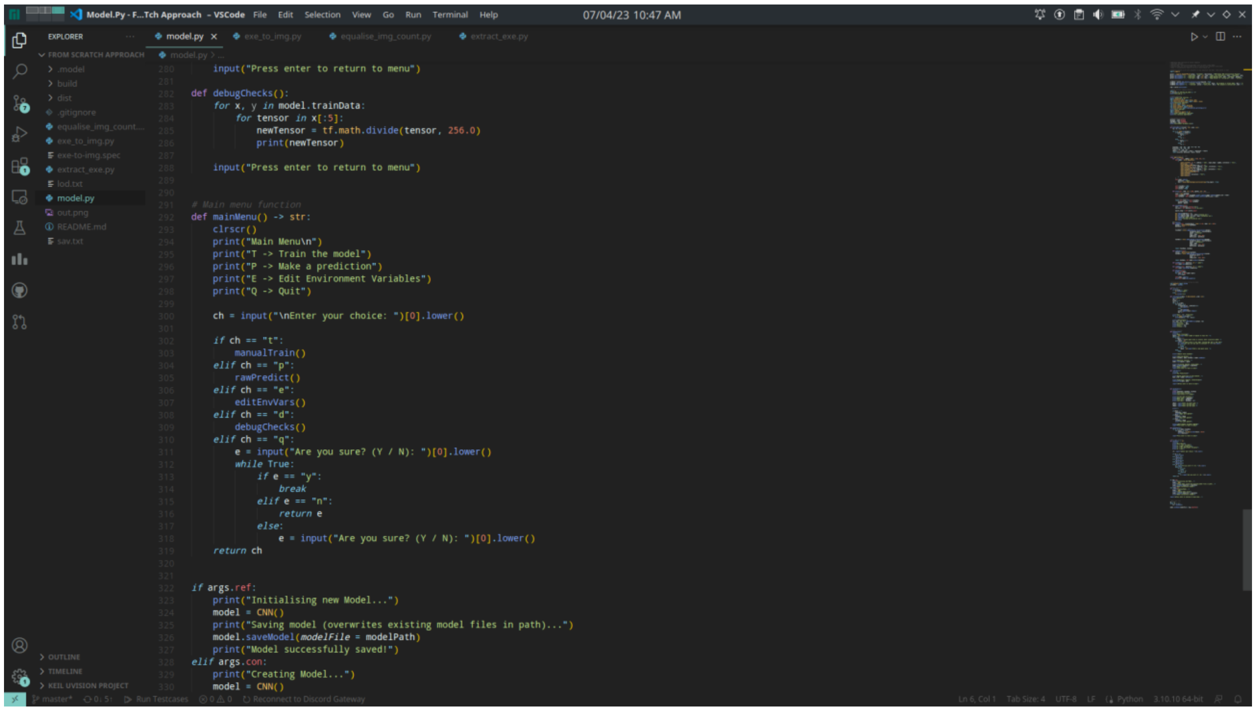
```python
        return trainData, testData

    def rawPredict(self):
        testDir = pathlib.Path(testPath)
        testData = utils.image_dataset_from_directory(testDir,
                              validation_split = 0,
                              seed = 0,
                              image_size = (512, 512))

        return testData, self.model.predict(testData)

    def saveModel(self, modelFile: str = ".model"):
        self.model.save(modelFile)

    def loadModel(self, modelFile: str = ".model"):
        self.model.load_weights(modelFile)

    def debugInfo(self):
        print(self.model)
        for layer in self.model.layers:
            print(layer)

        self.model.summary()
        print(self.model.get_weights())


# Getting the OS currently running the model
from platform import system
currentOs = system()

# Helper functions
def clrscr():
    if currentOs == 'Linux':
        os.system("clear")
    else:
        os.system("cls")

def interpretPred(data: tf.data.Dataset, pred: list):
    print("Results:")
    idx = 0
    actual = []
    res = []
    for x, y in data:
        for tensor in y:
            if(pred[idx][0] < pred[idx][1]):
                res.append(1)
            else:
                res.append(0)
            actual.append(tensor.numpy())
            idx += 1
```



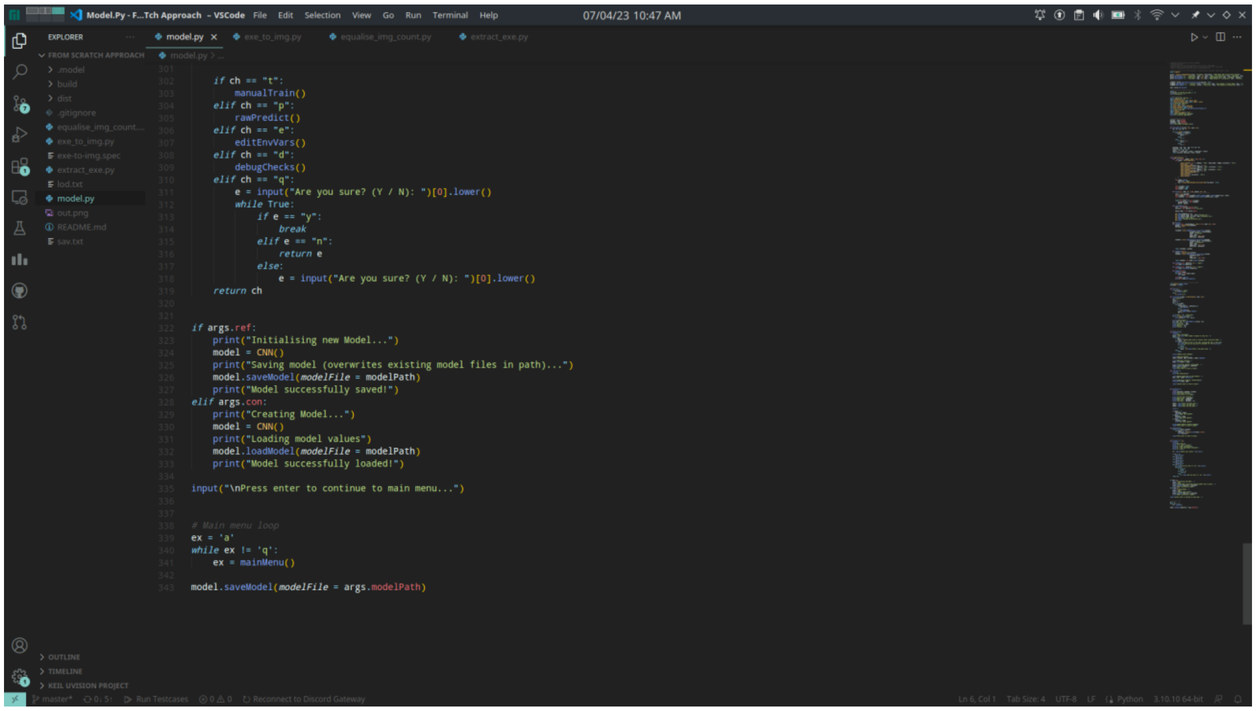
```python
            if(pred[idx][0] < pred[idx][1]):
                res.append(1)
            else:
                res.append(0)
            actual.append(tensor.numpy())
            idx += 1

    print("Real", "\t", "Predicted")
    for i in range(len(actual)):
        print(actual[i], "\t", res[i])

    print("\nPerformance:")
    acc, prec, rec, fs = getPredMetrics(actual, res)
    print("Accuracy: ", acc)
    print("Precision:", prec)
    print("Recall:", rec)
    print("F-Score:", fs)


# Wrapper functions
def manualTrain():
    clrscr()
    print("Model Training\n")
    epoch = int(input("Enter number of epochs to train for: "))
    while True:
        if epoch <= 1:
            epoch = ("Given epoch value is invalid, enter a positive number: ")
        elif epoch > 20:
            print("The epoch value is very high, training may take a long time")
            e = input("Are you sure you want to continue? (Y / N)")[0].lower()
            if e == "y":
                break
            elif e == "n":
                epoch = int(input("Enter a new epoch value: "))
        else:
            break

    print("\nEpoch value accepted")

    print("Importing dataset...")
    model.trainData, model.testData = model.loadData()

    print("Beginning training...")
    model.train(epochs = epoch)

    print("\nTraining complete, saving model...")
    model.saveModel(modelFile = modelPath)
    print("Model saved\n")
    input("Press enter to return to menu")

def rawPredict():
```



```python
    print("Model saved\n")
    input("Press enter to return to menu")

def rawPredict():
    clrscr()
    print("Raw Prediction\n")

    print("Making predictions on test dataset...")
    data, pred = model.rawPredict()

    print("Predictions complete, interpreting\n")
    interpretPred(data, pred)

    input("\nPress enter to return to menu")



def editEnvVars():
    clrscr()
    global modelPath, dataPath, testPath
    print("Environment Setup\n")
    print("Leave blank if no change")

    print("Current path variables:")
    print("Model path:", modelPath)
    print("Data path:", dataPath)
    print("Test path:", testPath, "\n")

    mPath = input("Enter new model path: ")
    dPath = input("Enter new data path: ")
    tPath = input("Enter new test path: ")

    print("")

    if mPath != "":
        modelPath = mPath
        print("Model path updated")
    if dPath != "":
        dataPath = dPath
        print("Data path updated")
    if tPath != "":
        testPath = tPath
        print("Test path updated")

    print("\nEnvironment variables updated")
    input("Press enter to return to menu")

def debugChecks():
    for x, y in model.trainData:
        for tensor in x[:5]:
            newTensor = tf.math.divide(tensor, 256.0)
```



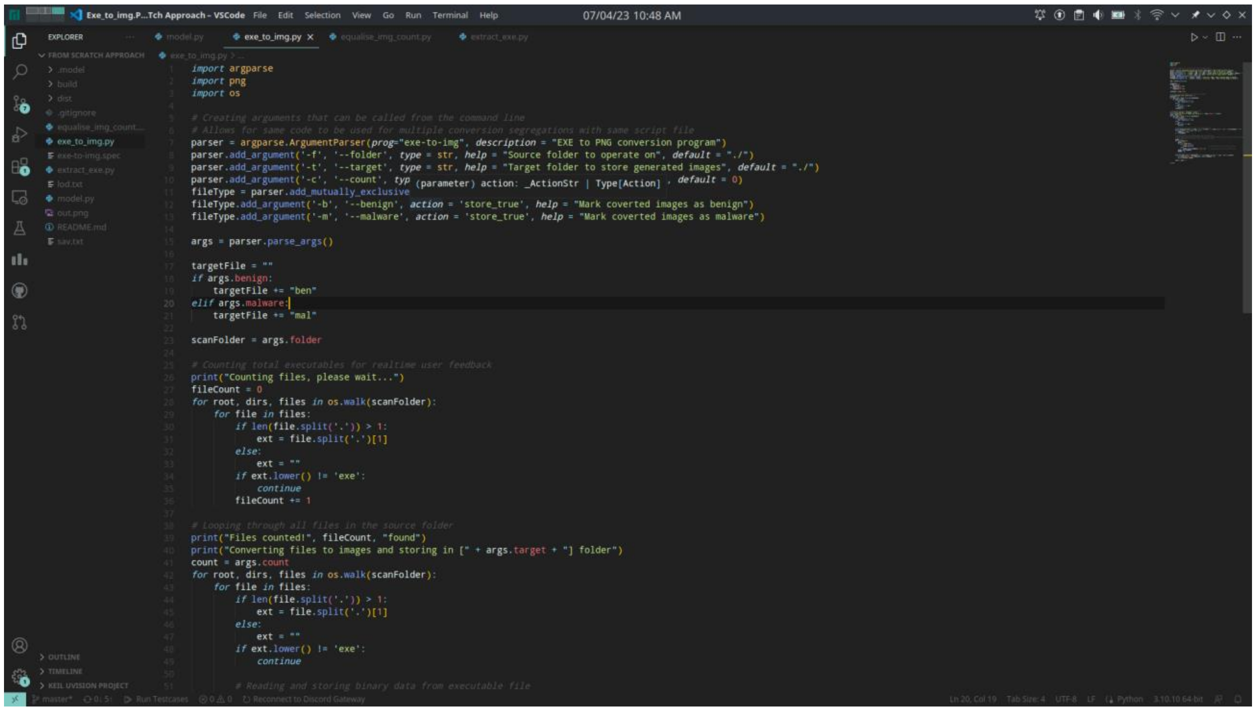
```
    input("Press enter to return to menu")

def debugChecks():
    for x, y in model.trainData:
        for tensor in x[:5]:
            newTensor = tf.math.divide(tensor, 256.0)
            print(newTensor)

    input("Press enter to return to menu")


# Main menu function
def mainMenu() -> str:
    clrscr()
    print("Main Menu\n")
    print("T -> Train the model")
    print("P -> Make a prediction")
    print("E -> Edit Environment Variables")
    print("Q -> Quit")

    ch = input("\nEnter your choice: ")[0].lower()

    if ch == "t":
        manualTrain()
    elif ch == "p":
        rawPredict()
    elif ch == "e":
        editEnvVars()
    elif ch == "d":
        debugChecks()
    elif ch == "q":
        e = input("Are you sure? (Y / N): ")[0].lower()
        while True:
            if e == "y":
                break
            elif e == "n":
                return e
            else:
                e = input("Are you sure? (Y / N): ")[0].lower()
    return ch


if args.ref:
    print("Initialising new Model...")
    model = CNN()
    print("Saving model (overwrites existing model files in path)...")
    model.saveModel(modelFile = modelPath)
    print("Model successfully saved!")
elif args.con:
    print("Creating Model...")
    model = CNN()
```



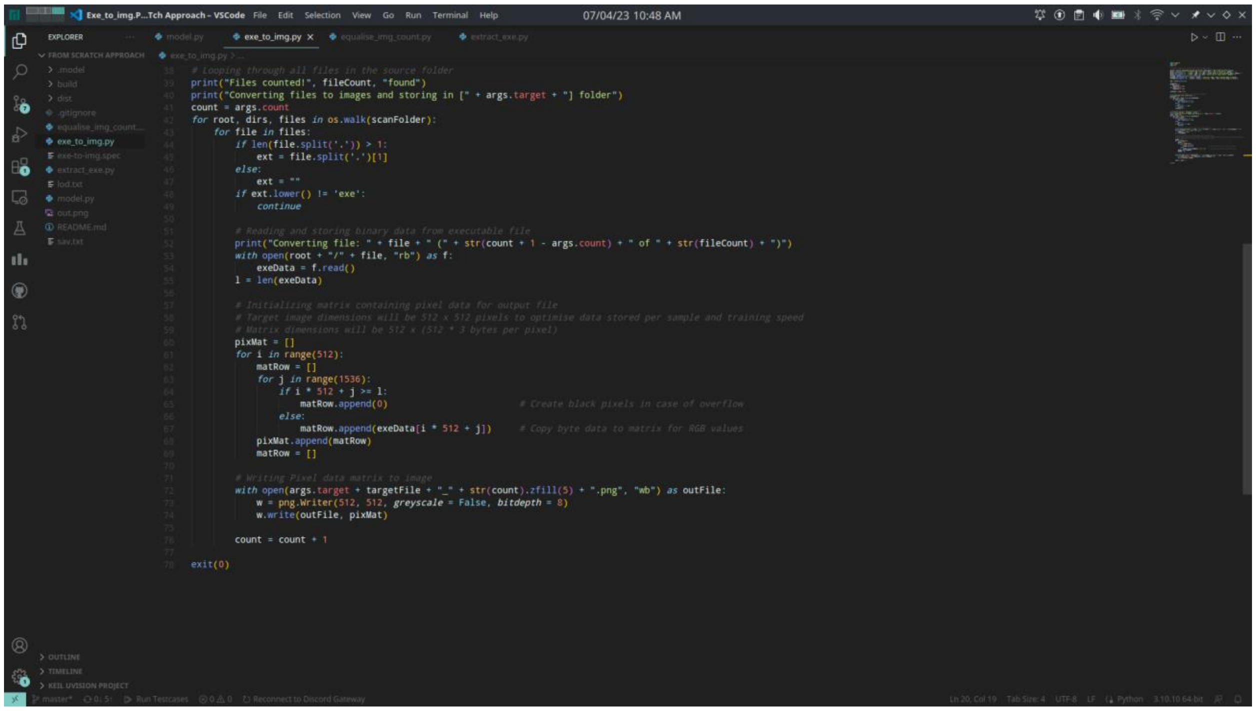
```

    if ch == "t":
        manualTrain()
    elif ch == "p":
        rawPredict()
    elif ch == "e":
        editEnvVars()
    elif ch == "d":
        debugChecks()
    elif ch == "q":
        e = input("Are you sure? (Y / N): ")[0].lower()
        while True:
            if e == "y":
                break
            elif e == "n":
                return e
            else:
                e = input("Are you sure? (Y / N): ")[0].lower()
    return ch


if args.ref:
    print("Initialising new Model...")
    model = CNN()
    print("Saving model (overwrites existing model files in path)...")
    model.saveModel(modelFile = modelPath)
    print("Model successfully saved!")
elif args.con:
    print("Creating Model...")
    model = CNN()
```

```
    print("Loading model values")
    model.loadModel(modelFile = modelPath)
    print("Model successfully loaded!")

input("\nPress enter to continue to main menu...")


# Main menu loop
ex = 'a'
while ex != 'q':
    ex = mainMenu()

model.saveModel(modelFile = args.modelPath)
```

## exe-to-img

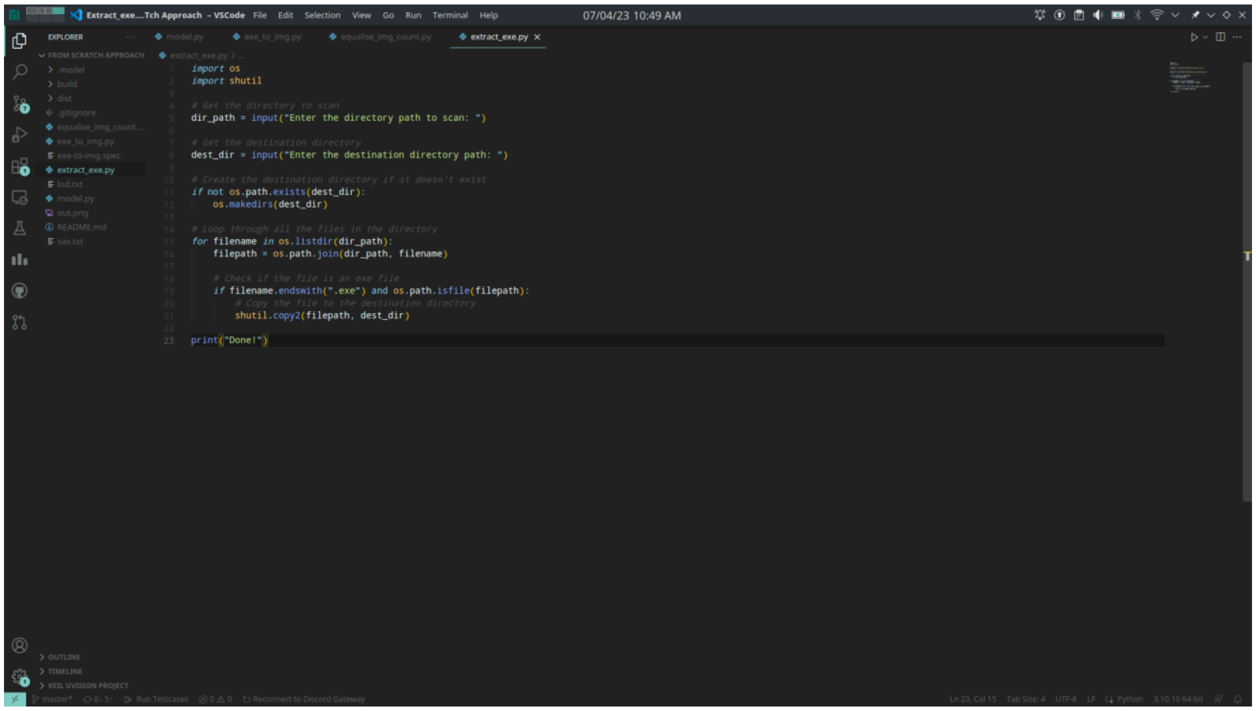


```python
import argparse
import png
import os

# Creating arguments that can be called from the command line
# Allows for same code to be used for multiple conversion segregations with same script file
parser = argparse.ArgumentParser(prog="exe-to-img", description = "EXE to PNG conversion program")
parser.add_argument('-f', '--folder', type = str, help = "Source folder to operate on", default = "./")
parser.add_argument('-t', '--target', type = str, help = "Target folder to store generated images", default = "./")
parser.add_argument('-c', '--count', typ (parameter) action: _ActionStr | Type[Action] , default = 0)
fileType = parser.add_mutually_exclusive
fileType.add_argument('-b', '--benign', action = 'store_true', help = "Mark coverted images as benign")
fileType.add_argument('-m', '--malware', action = 'store_true', help = "Mark coverted images as malware")

args = parser.parse_args()

targetFile = ""
if args.benign:
    targetFile += "ben"
elif args.malware:
    targetFile += "mal"

scanFolder = args.folder

# Counting total executables for realtime user feedback
print("Counting files, please wait...")
fileCount = 0
for root, dirs, files in os.walk(scanFolder):
    for file in files:
        if len(file.split('.')) > 1:
            ext = file.split('.')[1]
        else:
            ext = ""
        if ext.lower() != 'exe':
            continue
        fileCount += 1

# Looping through all files in the source folder
print("Files counted!", fileCount, "found")
print("Converting files to images and storing in [" + args.target + "] folder")
count = args.count
for root, dirs, files in os.walk(scanFolder):
    for file in files:
        if len(file.split('.')) > 1:
            ext = file.split('.')[1]
        else:
            ext = ""
        if ext.lower() != 'exe':
            continue

        # Reading and storing binary data from executable file
```



```python
# Looping through all files in the source folder
print("Files counted!", fileCount, "found")
print("Converting files to images and storing in [" + args.target + "] folder")
count = args.count
for root, dirs, files in os.walk(scanFolder):
    for file in files:
        if len(file.split('.')) > 1:
            ext = file.split('.')[1]
        else:
            ext = ""
        if ext.lower() != 'exe':
            continue

        # Reading and storing binary data from executable file
        print("Converting file: " + file + " (" + str(count + 1 - args.count) + " of " + str(fileCount) + ")")
        with open(root + "/" + file, "rb") as f:
            exeData = f.read()
        l = len(exeData)

        # Initializing matrix containing pixel data for output file
        # Target image dimensions will be 512 x 512 pixels to optimise data stored per sample and training speed
        # Matrix dimensions will be 512 x (512 * 3 bytes per pixel)
        pixMat = []
        for i in range(512):
            matRow = []
            for j in range(1536):
                if i * 512 + j >= l:
                    matRow.append(0)                    # Create black pixels in case of overflow
                else:
                    matRow.append(exeData[i * 512 + j])     # Copy byte data to matrix for RGB values
            pixMat.append(matRow)
            matRow = []

        # Writing Pixel data matrix to image
        with open(args.target + targetFile + "_" + str(count).zfill(5) + ".png", "wb") as outFile:
            w = png.Writer(512, 512, greyscale = False, bitdepth = 8)
            w.write(outFile, pixMat)

        count = count + 1

exit(0)
```

## extract-exe



```python
import os
import shutil

# Get the directory to scan
dir_path = input("Enter the directory path to scan: ")

# Get the destination directory
dest_dir = input("Enter the destination directory path: ")

# Create the destination directory if it doesn't exist
if not os.path.exists(dest_dir):
    os.makedirs(dest_dir)

# Loop through all the files in the directory
for filename in os.listdir(dir_path):
    filepath = os.path.join(dir_path, filename)

    # Check if the file is an exe file
    if filename.endswith(".exe") and os.path.isfile(filepath):
        # Copy the file to the destination directory
        shutil.copy2(filepath, dest_dir)

print("Done!")
```

## RESULTS AND DISCUSSION

### Our Model Performance

| **Measure** | **Value** |
|---|---|
| Accuracy | 0.952 |
| Precision | 0.981 |
| Recall | 0.976 |
| F-Score | 0.978 |

*Table 2(a): Model Performance when Using 32x32x3 resolution images*

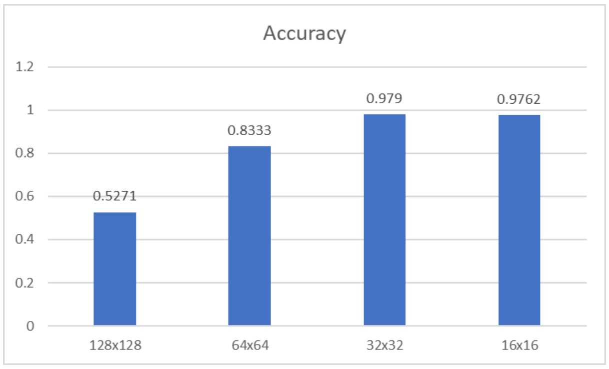
| **Measure** | **Value** |
|---|---|
| Accuracy | 0.472 |
| Precision | 0.472 |
| Recall | 0.265 |
| F-Score | 0.339 |

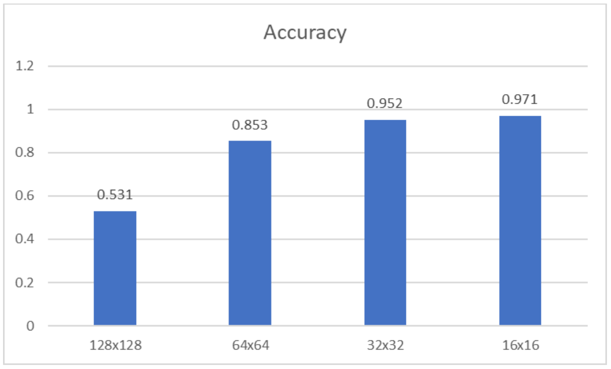
*Table 2(b): Model Performance when using 512x512x3 resolution images*

### Comparison with Other Models

| Model | Accuracy (256x256 Images) | Accuracy (32x32 Images) |
|---|---|---|
| k-NN | 0.527 | 0.979 |
| Naive Bayes | NaN | 0.955 |
| Support Vector Model | NaN | 0.974 |
| Reference CNN | NaN | 0.962 |
| CNN | 0.531 | 0.976 |

*Table 3: Model Comparison with Reference k-NN Model*





*(a)* *(b)*

*Figure 3: Model Performance (a) Reference Model, (b) Our Model*

In the base paper referred for the project, the k-Nearest Neighbours algorithm had very high accuracy of 97.9% when the supplied images of 32x32 pixels resolution were used. This is very high performance, but the major trade-off is that only the first 1024 bytes. When images of 128x128 pixels resolution (implying the first 16 kilobytes are read) were used, the accuracy fell down to 52.71%. This implies that in the modern age, where executables are present in sizes of range 10 to 100 megabytes, malware will be easier to pass through when packaged within larger executable files.

In comparison, the CNN used by the base paper had a lower accuracy of 96.29% when 32x32 pixel resolution images are used, compared to the k-NN itself. This discrepancy can be attributed to several factors including the number of epochs and the structure of the neural network itself. Where the CNN gained its advantage was in getting test results, with improvements by a factor of as much as 15.

Coming to the performance metrics of our CNN model, we see that an accuracy of 47.2% was attained when 512x512 pixel images were used. The difference in images is not just in the pixel size, but information packed within a pixel itself. The images used in the base paper used one pixel to store one byte of data, while our images can store 3 bytes of data in each pixel. This implies that our model achieved an accuracy score which was lowered by approximately 5% while processing 48 times more data in each image.

## Training Times

| Model | 128x128 Images | 64x64 Images | 32x32 Images |
|---|---|---|---|
| Reference k-NN | NaN | NaN | 0.33 s |
| Reference CNN (10 epochs) | NaN | NaN | 88.49 s |
| Our CNN (CPU run, 10 epochs) | 840 s | 520 s | 195 s |
| Our CNN (GPU Accelerated, 10 epochs) | 79.8 s | 49.4 s | 24.7 |

*Table 4: Model Training Time Comparison*

The models specified by the base paper were lacking in training times required for images of resolutions 128x128 pixels and 64x64 pixels, and did not specify the model specifics and code for us to actively run and test the training times. However, it is clearly visible that the increased complexity of our CNN model, demonstrated by a 2-fold increase in CPU bound training times of similarly sized image datasets, was sharply reduced by implementing our proposed approach using NVidia CUDA framework and Tensorflow Keras libraries.

This increased performance from CPU bound training to GPU accelerated training is evident in larger image sizes, with improvements of up to 1052.63%. The faster GPU execution of the model training algorithm can allow for higher epoch counts and model complexities to be used

## CONCLUSION AND FUTURE SCOPE

In conclusion, using image processing with CNN models for malware detection offers a promising solution to the challenges faced by traditional signature- and behavior-based detection methods. system must face. By converting malware into an image representation and using deep learning techniques to extract relevant features, this method has shown significant improvements in accuracy, speed, and accuracy. resilience to cloaking techniques.

In addition, the flexibility and scalability of CNN models enables the detection of many different types of malware, including previously unknown variants. As the use of machine learning in cybersecurity continues to grow, the potential for integrating image processing with CNN models in malware detection is enormous.

The scope for future research in this area is enormous. One potential direction is to explore the use of transformation and aggregation learning methods to further improve model accuracy. Transfer learning can leverage pre-trained CNN models on large data sets to learn features relevant to malware detection, while ensemble methods can combine multiple models. to improve overall performance.

Additionally, investigating the use of explainable AI (XAI) techniques can improve the interpretability of the model, allowing security professionals to better understand how the model performs predictions and capable of identifying new features that can help detect malware.

In summary, the use of image processing with CNN models in malware detection has shown significant improvements in accuracy, speed, and resiliency. Through continuous research and development, this method has the potential to be a powerful tool in the fight against malware and cyber threats.

## REFERENCES

[1] Faruk, M. J. H., Shahriar, H., Valero, M., Barsha, F. L., Sobhan, S., Khan, M. A., ... & Wu, F. (2021, December). Malware detection and prevention using artificial intelligence techniques. In 2021 IEEE International Conference on Big Data (Big Data) (pp. 5369-5377). IEEE.

[2] Majid, A. A. M., Alshaibi, A. J., Kostyuchenko, E., & Shelupanov, A. (2021). A review of artificial intelligence based malware detection using deep learning. Materials Today: Proceedings.

[3] Qasem Abu Al-Haija and Saleh Zein-Sabatto (2020, December)An Ecient Deep-Learning-Based Detection and Classification System for Cyber-Attacks in IoT Communication Networks 10.3390/9122152

[4] Baek, S., Jeon, J., Jeong, B., & Jeong, Y. S. (2021). Two-stage hybrid malware detection using deep learning. Human-centric Computing and Information Sciences, 11(27), 10-22967.

[5] Syed Shakir Hameed Shah ,Abd Rahim Ahmad , Norziana Jamil and Atta ur Rehman Khan(2022, July) Memory Forensics-Based Malware Detection Using Computer Vision and Machine Learning 11, 2579.

[6] Bose, S., Barao, T., & Liu, X. (2020, July). Explaining ai for malware detection: Analysis of mechanisms of malconv. In 2020 International Joint Conference on Neural Networks (IJCNN) (pp. 1-8). IEEE.

[7] Rahman Ali , Asmat Ali,Farkhund Iqbal,Mohammed Hussain ,and Farhan Ullah (2022,September)Deep Learning Methods for Malware and Intrusion Detection: A Systematic Literature Review 10.1155/2022/2959222

[8] Huang, L. C., Chang, C. H., & Hwang, M. S. (2020). Research on malware detection and classification based on artificial intelligence. International Journal of Network Security, 22(5), 717-727.

[9] Manish Kumar (2021, October) Scalable malware detection system using big data and distributed machine learning approach. Springer-Verlag GmbH Germany. doi.org/10.1007/s00500-021-06492-9(

[10] Abdelmonim Naway, Yuancheng Li (2019, Jan) Android Malware Detection Using Autoencoder doi.org/10.48550/arXiv.1901.07315

[11] Xing, X., Jin, X., Elahi, H., Jiang, H., & Wang, G. (2022). A malware detection approach using autoencoder in deep learning. IEEE Access, 10, 25696-25706.

[12] Sohrabi, S., Udrea, O., & Riabov, A. (2013, June). Hypothesis exploration for malware detection using planning. In Twenty-Seventh AAAI Conference on Artificial Intelligence.

[13] Alkahtani, H., & Aldhyani, T. H. (2022). Artificial intelligence algorithms for malware detection in android-operated mobile devices. Sensors, 22(6), 2268.

[14] Patil S, Varadarajan V, Walimbe D, Gulechha S, Shenoy S, Raina A, Kotecha K. Improving the Robustness of AI-Based Malware Detection Using Adversarial Machine Learning. Algorithms. 2021; 14(10):297.

[15] Fernando DW, Komninos N, Chen T. A Study on the Evolution of Ransomware Detection Using Machine Learning and Deep Learning Techniques. IoT. 2020; 1(2):551-604

[16] Patil, S., Varadarajan, V., Walimbe, D., Gulechha, S., Shenoy, S., Raina, A., & Kotecha, K. (2021). Improving the robustness of AI-based malware detection using adversarial machine learning. Algorithms, 14(10), 297.

[17] Alkahtani H, Aldhyani THH. Artificial Intelligence Algorithms for Malware Detection in Android-Operated Mobile Devices. Sensors. 2022; 22(6):2268

[18] Ye, Y., Li, T., Adjeroh, D., & Iyengar, S. S. (2017). A survey on malware detection using data mining techniques. ACM Computing Surveys (CSUR), 50(3), 1-40.

[19] Choi S. Combined kNN Classification and Hierarchical Similarity Hash for Fast Malware Detection. Applied Sciences. 2020; 10(15):5173.

[20] Libri, A., Bartolini, A., & Benini, L. (2020). pAElla: Edge AI-based real-time malware detection in data centers. IEEE Internet of Things Journal, 7(10), 9589-9599.

[21] Peiravian, N., & Zhu, X. (2013, November). Machine learning for android malware detection using permission and api calls. In 2013 IEEE 25th international conference on tools with artificial intelligence (pp. 300-305). IEEE.

[22] Poudyal, S., & Dasgupta, D. (2020, December). Ai-powered ransomware detection framework. In 2020 IEEE Symposium Series on Computational Intelligence (SSCI) (pp. 1154-1161). IEEE.

[23] Tanjie Wang, Yueshen Xu, Xinkui Zhao, Zhiping Jiang, Rui Li, Android malware detection via efficient application programming interface call sequences extraction and machine learning classifiers, IET Software, 10.1049/sfw2.12083.

[24] Gao, Y., Hasegawa, H., Yamaguchi, Y., & Shimada, H. (2022). Malware detection using LightGBM with a custom logistic loss function. IEEE Access, 10, 47792-47804.

[25] S. Poudyal and D. Dasgupta, "AI-Powered Ransomware Detection Framework," 2020 IEEE Symposium Series on Computational Intelligence (SSCI), Canberra, ACT, Australia, 2020, pp. 1154-1161, doi: 10.1109/SSCI47803.2020.9308387.

[26] Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. (2017, July). Malware images: visualization and automatic classification. In Proceedings of the 8th international symposium on visualization for cyber security (pp. 1-7).

[27] Venkatraman, S., & Alazab, M. (2018). Use of data visualisation for zero-day malware detection. Security and Communication Networks, 2018, 1-13.

[28] Moussas, V., & Andreatos, A. (2021). Malware detection based on code visualization and two-level classification. Information, 12(3), 118.

[29] Son, T. T., Lee, C., Le-Minh, H., Aslam, N., Raza, M., & Long, N. Q. (2020). An evaluation of image-based malware classification using machine learning. In Advances in Computational Collective Intelligence: 12th International Conference, ICCCI 2020, Da Nang, Vietnam, November 30–December 3, 2020, Proceedings 12 (pp. 125-138). Springer International Publishing.

[30] Seok, S., & Kim, H. (2019). Visualized malware classification based-on convolutional neural network. Journal of The Korea Institute of Information Security & Cryptology, 26(1), 197-208.