Malware Detection using Deep Learning

BSCS

Group Members:

Ahmad Shah
(05120171201026)

Shahid Ullah
(05120171201021)

Aniqa Nawaz
(05120171201007)

Supervisor:

Prof. Dr. M. Shamim Baig

Muslim Youth University
Japan Road, Islamabad

Department of Computer Science
Muslim Youth University, Islamabad

Malware Detection using Deep Learning

by

Ahmad Shah (05120171201026)
Shahid Ullah (05120171201021)
Aniqa Nawaz (05120171201007)

A thesis submitted in fulfilment of the requirement for the award of

Bachelor's Degree

in

Computer Science
(2017-2021)

Muslim Youth University
Japan Road, Islamabad

Supervisor's Signature: _____

Head of Department Signature: _____

Department of Computer Science
Muslim Youth University, Islamabad

UNDERTAKING

Use the following undertaking as it is. I certify that research work titled **"Malware Detection using Deep Learning"** is my own work. The work has not been presented elsewhere for assessment. Where material has been used from other sources, it has been properly acknowledgment/referred.

Submitted by:

Signature of Student	Signature of Student	Signature of Student
Ahmad Shah	Shahid Ullah	Aniqa Nawaz
05120171201026	05120171201021	05120171201007

ACKNOWLEDGEMENT

I am thankful to Allah almighty for the blessings in the successful completion of my project **"Malware Detection using Deep Learning"**. I have a great pleasure in acknowledging the help given by various individuals throughout the project work. This project is itself acknowledgement to the inspiration, drive and technical assistance contributed by many individuals.

I express my sincere and heartfelt gratitude to **Prof. Dr. M. Shamim Baig** Head of the department of Computer Science of Muslim Youth University, for being helpful and Co-operative during the period of the project.

I extent my sincere thanks to all the teaching and non-teaching staff for providing the necessary facilities and help. Without the support of anyone of them, this project would not have been a reality. I am also thankful to all my friends and to my parents for their incredible support given to me in every aspect.

Abstract:

Powerful and efficient mitigation of malware has been a protracted-lived endeavor inside the fact's protection network. The improvement of an anti-malware framework which can take a look at a difficult to understand malware is a prolific motion that may income numerous of the IT sectors. This venture aims at solving this problem by using applying Machine Learning strategies to first locate the nature of executable. After the affirmation of the maliciousness, a 2d-degree anti-malware system can efficaciously come across the corresponding malware family for the given malware X the usage of a mathematical generalization, $f: X \rightarrow Y$ This is achieved the usage of the electricity of Deep Learning. Using such models can be powerful in detecting zero-day attacks. The concept is to apply the pre-processed header details for education and trying out motive for first phase of detection, which can be extracted the use of Python programs. The next level in which the model predicts the same general end result. The model will employ Convolution Neural Network (CNN) version, Multilayer Perceptron (MLP) Model [14] and Long Short-Term Memory (LSTM) [15] version a variant of RNN. In this work we present malware location from raw byte sequences as a productive exploration area to the bigger deep learning community. Building a neural network for such an issue presents various difficulties that did not happened in tasks, such as image processing or NLP (Natural Language Processing). Specifically, we note that detection from raw bytes gives a sequence problem throughout 2,000,000 (2 million) time steps and a problem where batch normalization seems to frustrate the learning process. We present our underlying initial work in building an answer for tackle this issue, which has linear complexity dependence on the sequence length, and takes into consideration interpretable sub-regions of the binary to be distinguished. In doing as such we will talk about the many difficulties in building a neural organization to deal with information at this scale, and the strategies we used to work around them. Our main work is Established upon work and Concept of [1710.09435v1] Malware Detection by Eating a Whole EXE.

Contents

Chapter 1: Introduction:

Malware is a software advanced to damage or infiltrate a secured system without the valid proprietor's approval [4]. Malware is a vast definition for all the pc software or package that placed it to threats. Malware can be in reality categorized as either standalone software program or a record infector. The discovery of malicious software (malware) is a significant issue in cyber security, particularly as a greater amount of society becomes reliant upon computers. As of now, single occurrences of malware can cause a great many dollars in harms (Anderson et al. 2013). Hostile to infection items give some assurance against malware, yet are becoming progressively insufficient for the issue. Current anti-virus of infection innovations utilizes a mark-based methodology, where a mark is a bunch of physically created rules trying to distinguish a little group of malwares. These standards are for the most part explicit, and cannot normally perceive new malware regardless of whether it utilizes a similar usefulness. This approach is inadequate as most conditions will have one of a kind pairs that will have never been seen (Li et al. 2017) and millions of new malware tests are tracked down each day. The counter infection suppliers and industry specialists have perceived the impediments of marks for a long time (Spafford 2014). The need to foster procedures that sum up to new malware would make the assignment of malware identification an apparently ideal fit for AI, however there exist critical difficulties.

1.1: Basics of Malware Detection

Powerful and productive mitigation of malware has been a long-lived endeavor in the information security network. The improvement of an anti-malware framework, which can look at a difficult to understand malware, is a prolific movement that may profit several of the IT sectors [9]. Detection of malware can be carried out using Signature based either totally strategies or Anomaly primarily based strategies which is likewise called behavioral detection technique.

As discussed in [7] current malware analysis techniques either rely on static techniques, which essentially search for malware signature by looking into its code and other header data without actually, executing the malware functionality, or use dynamic techniques, where the suspicious executable is run in a controlled environment. However, there are drawbacks associated with both these techniques. Static techniques cannot detect zero-day attacks as discussed in [8], i.e. the malwares whose signature is not known therefore needs frequent updating of its knowledge base, requires immense knowledge of assembly language for reverse engineering the code and detecting any possible breaches in access restrictions. Dynamic approaches [10] may however detect zero-day attacks but carefully designed malwares may detect presence of sandboxed environment, and therefore may outsmart it by not exposing their full functionality, making it more tedious for the analyst to detect.

Brief evaluation has been given in **Table 1.1**.

Criterion	Signature Based Detection	Anomaly Based Detection
Detection basis	The code is analyzed for possible detection of signature corresponding to the known malware.	The system is trained with the behavior of the legitimate user. Deviation from this behavior is treated as malicious.
Accuracy	Highly accurate and reliable	May result in high false positive cases
System speed	Supports faster process of detection	It is slower and more complex than Signature based detection system.
System speed	Cannot detect zero-day attacks	Can also detect zero-day attacks.

Table 1.1: Comparison between Signature and Anomaly based detection techniques.

Therefore, this work will focus mainly on detecting malware and also catching the zero-day attacks with high accuracy. In order to make the results more reliable, the detection phase can deploy an Ensemble of 3 different Deep Learning Models and Algorithms namely Convolutional Neural Networks 1D (CNN-1D), Artificial Neural Networks (ANN / MLP) and Long Short-Term Memory (LSTM – an advanced variation of Recurrent Neural Networks [RNN]). This essentially provides speed of decision making about the nature of the executable.

1.2: What is Deep Learning?

Deep Learning Algorithms utilize Artificial Neural Networks as their principle structure. What separates them from different algorithms is that they don't need expert input during the feature design and designing stage. Neural Networks can gain proficiency with the attributes of the information.

Deep Learning algorithms take in the dataset and become familiar with its examples, they figure out how to represent the data with features they extract on their own. Then, at that point, they consolidate various representations of the dataset, every one recognizing a particular example or trademark, into a more theoretical, significant level portrayal of the dataset. This hands-off approach, absent a lot of human mediation in include plan and extraction, permits algorithms to adapt much faster to the data at hand.

1.3: Malware Growth Statistics

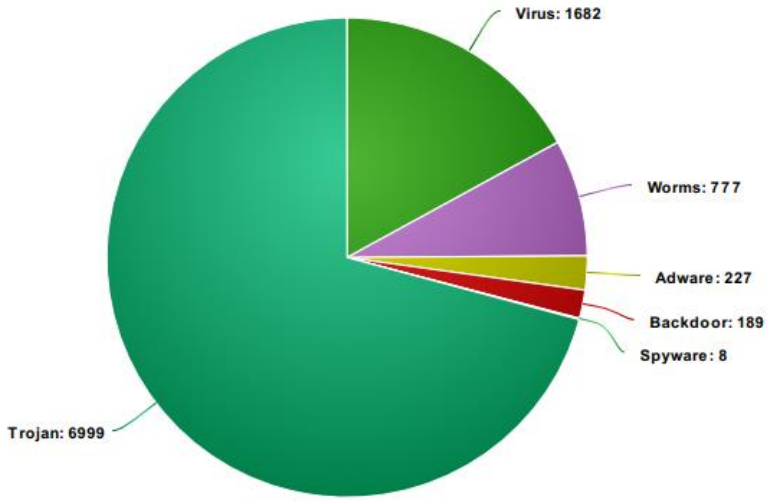
Over the last decade, malware has evolved tremendously with the widespread usage of internet. The growth has been exponential over all the platforms, with highest on Android and Windows. More than 357 million new malware samples have been recorded in the year 2016 by Kaspersky.

One main reason is the use of sophisticated obfuscation techniques that malware authors use to perform covert executions. Malware develops at a quick pace due to cutting edge malware acing the craft of avoidance. In this manner, customary antivirus engines think that it's hard to identify assaults in the principal stages. Malware is getting greater and greater. It powers development, advancement and urges noxious performers to effortlessly achieve their objectives.

The malware showcase developed from something that was tried and most likely utilized for the sake of entertainment, – with programmers making projects to perceive how they can access unapproved places and afterward concentrating on cash and going for taking individual information – into a more focused on assault vector.

In 2018, malware was considerably progressively nimble, and Gand-crab ransomware is an extraordinary model. It is a quickly developing malware that has been utilized and spread in floods of spam battles. While it achieved the rendition 4 as of now, this bit of malware was at first appropriated by means of endeavor packs which manhandles programming vulnerabilities found in frameworks.

The exponential development of malwares over the ongoing years can be credited to the way that malware is a beneficial business for pernicious creators. Profiting from malware has ended up being a triumphant choice for cybercriminals. As a rule, they pick rich and created nations, target expansive and effective associations, from where they can blackmail a great deal of cash and access their profitable information.



As indicated by the Telstra Security Report, the greater part of organizations who were casualties of a ransomware assault have paid the payoff and they would do it once more. Nearly 60 percent of ransomware exploited people in New Zealand and 55 percent in Indonesia paid the payoff, making it the most elevated for Asia. In Europe, 41 percent of respondent ransomware unfortunate

casualties paid up [11].

New discoveries from Check Point look into expressed that the quantity of worldwide associations influenced by crypto-mining malware dramatically increased from the second 50% of 2017 to the initial a half year of this current year, with digital crooks making an expected $2.5 billion in the course of recent months.

1.4: Problem/Motivation

The expanding instances of pernicious exercises over the web, where each new malware is more astute than the past, represents a genuine test to the malware investigators to distinguish and smother its effects. This gives a ceaseless capability of making security frameworks more astute. Machine Learning / Deep Learning approaches give a major lift in this regard. This work can help taking care of the issue of recognizing zero-day assaults by transformative infections, where the transformation motor itself changes structure, consequently making it unthinkable for signature-based discovery methods to see it. Which is the reason most of the network safety device merchants are moving towards Machine Learning and Deep Learning based methodologies. So, the future in these fields is assuredly splendid. Figure 1 shows the general malware development pattern in the course of the keep going ten years on different stages.

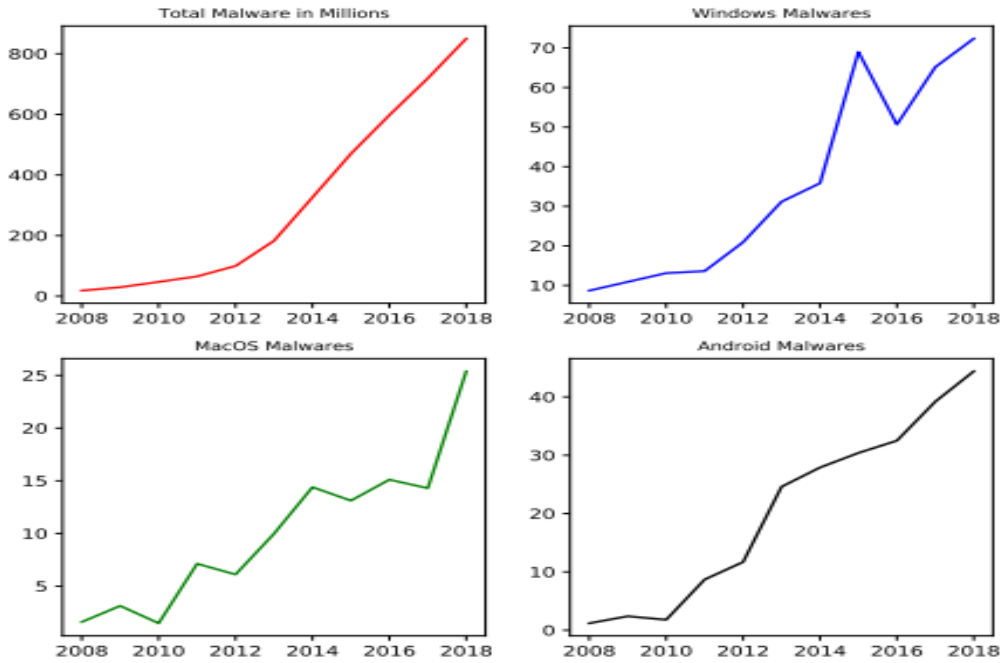


Figure 1.2: Malware growth over different platforms in the last decade

1.5: Objectives:

The proposed system is expected to perform the following tasks:

- Detect **zero-day** attacks as well as known malware attacks.
- **Generalize** the PE by classifying it into malwares and Benign Categories by studying its characteristics.

- **The Detection** has to be done in a fairly quick time using Deep Learning based classification algorithm.

The objective of the phases would be to check accuracy and precision of 3 Deep Learning based classification algorithms. In addition, an Ensembled model will also be created to test whether it outweighs the conventional algorithms and Detects the Nature of PE File.

1.6: Workflow:

According to the implementation and Research objectives, the report will describe the work flow as below:

Step 1 – In our case, we have used Standard Malware Executables Dataset known as Ember 2018. It has pre-collected Benign and Malware Dataset Stored in Json files which can be further extracted to numpy arrays.

Step 2 - Using Numpy the Dataset is split into Train Set, Validation Set, and Testing set for the next step. And the Features set is made.

Step 3 - Using Various Deep Learning Models, the dataset and features set is passed and trained on certain models then the Best Weight and Model Structures of Relevant Models are saved for further evaluation and testing of PE File.

Step 4 – Saved Model and its Weight Parameters should be called to Evaluate the Testing Set and Cross Confirm the Accuracy and Precision Results.

Step 5 – The Last phase will be to Test the models and its trained parameters with PE input file. The Ember Feature Extractor will extract the PE File Info from the input file and the Test Results will be Predicted by the defined models to classify the file either Malicious or Benign.

1.7: Key Concepts:

The detection and analysis of malware has always been a challenging task with limitless scope of enhancements in techniques. The broad classification of detection techniques are the Static and Dynamic detection techniques. Performing dynamic and static analysis together helps identify the true capabilities and intent of the malware and can provide more robust technical indicators which might not be attainable by static or dynamic analysis alone.

What is Static Malware Analysis Technique?

Basic Static Analysis tests malware without running the real code or directions. It exploits distinctive devices and procedures to rapidly discover whether a record is malicious or amiable, give information about its usefulness and assembles the specialized markers to give straightforward marks [2]. This information assembled by means of fundamental static examination can incorporate MD5 checksums, filename, hashes, document measure, record type and acknowledgment by hostile to malware identification devices. When performing basic static analysis, diverse instruments and procedures are utilized to assemble however much information as could be expected about the malware. Most of the time, the initial step is to filter the malware to discover whether an antivirus will identify it as being malicious or not [8].

Criterion	Static Analysis	Dynamic Analysis
Central Idea	Done without executing the malware code	Done by executing the malware code in a controlled environment
Approach	Uses Signature based approach for malware detection	Uses Anomaly based approach for malware detection
Requirements	Involves file fingerprinting, reverse engineering a binary code, virus scanning, file unpacking and packer detection	Involves API and system calls, registry changes, instruction traces, memory writes and more.
Effectiveness	Ineffective if malware codes are more sophisticated in terms of mutation engine.	Mostly effective against all types of malwares since it analyzes them step-wise.

Table 1.7: Brief comparison between Static and Dynamic Analysis Technique

A more sophisticated advanced static detection technique involves reverse engineering the binary code into assembly code using a disassembler. This involves more complexity and the analyst must be well aware of the assembly language.

What is Dynamic Malware Analysis Technique?

Basic dynamic analysis [8] actually executes malware payload to monitor its behavior, understand its control flow and determine the indicators which can be used in signature detection. Technical indicators unmasked via basic dynamic analysis can include IP addresses, domain names, file path, registry keys and additional files located on the system.

Additionally, it can identify communication with command and control server for either downloading additional malware files or for deploying botnets into the network. Basic dynamic analysis can be understood as what most automated virtual environments or dynamic analysis engines perform nowadays. This form of analysis is often carried out in a sandboxed environment to keep the malware from actually harming the production machines, another advantage is that these sandboxes being virtual systems, can be conveniently rolled back to a safe state post completion of the analysis. Analyst can also make use of Debuggers such as **GDB** or **WinDbg** while executing a malware, to see its effect on the host machine.

Chapter 2: Literature Survey

There can be several theories which may be utilized as means of information regarding the maliciousness of a file or an executable. Some of them are mentioned below along with their setbacks where the scope of improvement exists.

2.1: Previous Works and Their Drawbacks:

Hyunjae Kang et. al. in [16] proposed improvement in permission based and API check based methods for malware detection in Android platform by incorporating creator's information. This cannot be always very reliable as it may lead to high false positive rates because all executables from the unknown creators cannot be malwares.

The work of Zarni Aung, Win Zaw in [17] is also based on machine learning approach, where the set of permissions that the executables and APIs require are treated as the feature vector of the subject. Based on these features, the ML based framework classified the file as goodware or malware. The drawback of this approach is that it misses most of the important parameters such as size of initialized and uninitialized data which changes in the case of obfuscated code, min-max resource entropy, count of invalid strings and version information. Because a legitimate executable almost always contains various legitimate strings, if this is not the case then there is a high chance that the creator has obfuscated the code and is therefore trying to perform some backdoor activity.

The work of Shafiq et. al. in [18] extracted 189 features from the PE headers and used different feature selection methods like Principal Component Analysis, Haar Wavelet Transformation and RFR removal to reduce the feature set and trained J48 decision tree to achieve the maximum accuracy of 97% and 0.5% of False Positive Rate. The experiment was conducted with a large dataset of more than 10,000 samples which on an average took 0.245 seconds to scan an unseen file.

The work of Wang et. al. in [20] proposed an SVM based discovery strategy for recognizing inconspicuous PE malware. Utilizing static examination, PE header passages were separated and the SVM classifier was prepared utilizing chosen highlights. Grouping model recognizes infections and worms with extensive precision however the location exactness is lower for trojans and indirect accesses.

Current malware evaluation techniques either rely upon static techniques, which basically search for malware signature by means of looking into its code and different header statistics without simply executing the malware capability, or use dynamic techniques, wherein the suspicious executable is run in controlled surroundings. However, there are drawbacks related to both these strategies. Static strategies cannot discover zero-day assaults as mentioned in, i.e. The malwares whose signature isn't always regarded therefore desires common updating of its knowledge base, calls for tremendous understanding of assembly language for opposite engineering the code and detecting any possible breaches in get entry to restrictions. Dynamic tactics may also but stumble on 0-day assaults but cautiously designed malwares may hit upon presence of sandboxed surroundings, and consequently may additionally outsmart it with the aid of not exposing their full functionality, making it more tedious for the analyst to come across.

Therefore, this work will focus especially on detecting malware and additionally catching the zero-day attacks with high accuracy. In order to make the results extra dependable, the detection segment can deploy an Ensemble of 3 special Deep Learning approaches particularly CNN, MLP, and LSTM approaches. This essentially provides speed of decision making about the nature of the executable. Further analysis provides in-depth knowledge of the malware family to which it belongs.

The Data reshaping for certain models is performed by using the Numpy function known as expand_dims. Python allows the users to split the dataset into random train and test data in the specified ratio, which is utilizable for cross validation, to avoid overfitting. Then, we use Various Deep learning approaches to find which will be the best approach for further detection.

2.2: Portable Executable Format:

The Portable Executable format is the standard document design for executables, object code and Dynamic Link Libraries (DLLs) utilized in 32-and 64-digit variants of Windows OS.

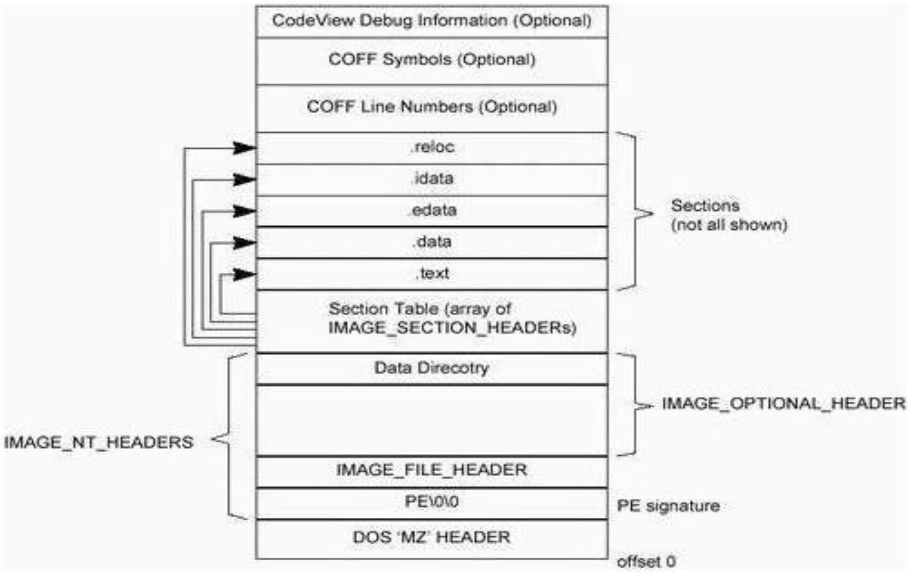


Figure 2.2.1: The 32-bit PE file structure. Creative commons image courtesy [3].

Ongoing examination demonstrates that compelling malware detection can be carried out in view of analyzing or dissecting portable executable (PE) file headers. Such examination normally depends on earlier information or prior knowledge of the header to separate pertinent elements and extract relevant features.

However, it is possible to think about the whole header overall, and utilize this straightforwardly to decide if the file is malware. In this examination, we gather an enormous and diverse malware data set. We then, at that point, investigate the adequacy of different Machine Learning Techniques based on PE headers to characterize the malware samples. We look at the accuracy and proficiency of every method considered. [5]

Comprehension of PE file format is not just needed for figuring out, however it is additionally expected for understanding the ideas of operating system. [5]

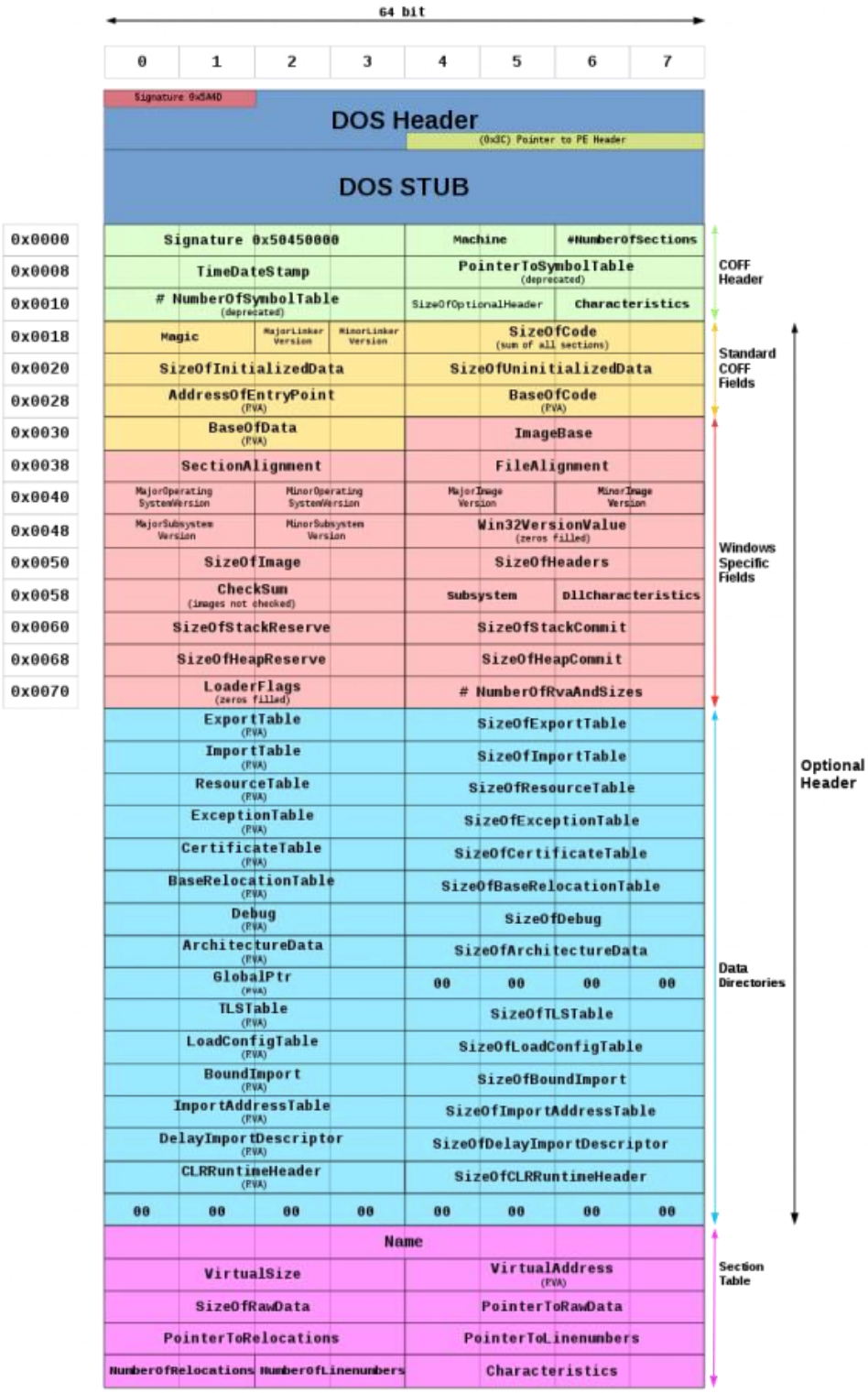


Figure 2.2.2: 64-bit PE File Structure Containing all Sections of a File

A PE is a file format created by Microsoft utilized for executables (.EXE, .SCR) and dynamic link libraries (.DLL). A PE file infector is a malware family that proliferates by affixing or wrapping malicious code into other PE files on an infected system. PE infectors are not especially mind boggling and can be identified by most antivirus products. Not with standing, this has not halted such malware

from spreading to OT networks where slight deviations in execution or framework conditions might bring about or result in adverse outcomes. [6]

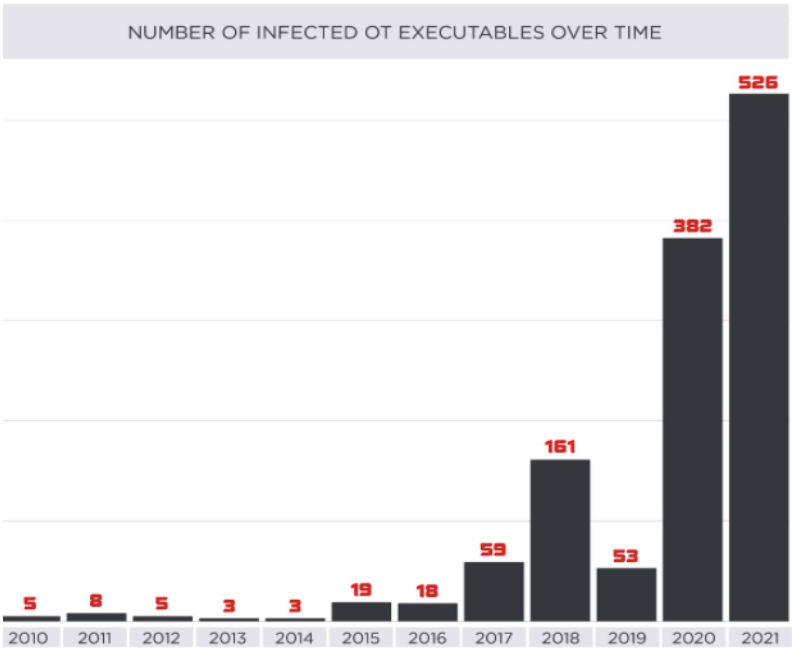


Figure 2.2.3: shows the vertical pattern of infected OEM OT executables among 2010 and mid-2021.

2.3 The PE file headers and Sections

The PE document format is an information structure consisting of the data vital for the Windows OS loader to deal with the wrapped executable code. About each record with executable code that is stacked by Windows is in the PE document format, however some heritage document groups do show up on uncommon event in malware.

PE documents start with a header that incorporates information related to the section code, application type, dynamic space required by the program and library functions necessary.

Knowing how the library code is connected is basic for comprehension of malware on the grounds that the information which can be found in the PE document header relies upon how the library code has been connected. [2] The common sections in the PE file are mentioned as follows:

.txt – The .text section consists of the instructions the CPU is expected to execute. Other sectors store supporting data and information. For the most part, this is the section which contains the main function from where the execution starts, it also contains the pointer to the files and tables to be loaded.

.rdata – This section stores the read-only data used by the program.

.data – The .data section contains the program's global information, which is not scope dependent to the various functions of the program. Thus, each method can use the data stored in global scope.

Executable	Description
.text	Contains the executable code
.rdata	Contains read-only data. Global scope of access.
.data	Data with global access to read and write.
.idata	Not always present. Stores import details.
.edata	Not always present. Stores details of exported data via functions
.pdata	Stores information related to handling exceptions. Only for 64- bit exe.
.rsrc	Lists the necessary resources to the exe.
.reloc	Helps relocating the library files.

Table 2.3: Sections of a PE file for a Windows executable [2]

.rsrc – The .rsrc section holds the information of the resources utilized by the executable that are not regarded as its features, for example, images, symbols, strings, and optional headers. String values can be saved either in the .rsrc, or in the fundamental program, however they are frequently put away in the .rsrc section for multi-language support.

2.4 Sections Header in PE

The most valuable data originates from the section header. These headers portray each segment of a PE document. The compiler for the most part makes and names the areas of an executable, and the client has little command over these names. Thus, the segments are typically reliable from one executable to other. For instance: The Size of Raw Data indicates how huge the area is on disk, Virtual Size shows how much space is apportioned to the segment amid the load process. These two qualities ought to generally be equivalent, since information should take up the same amount of room on the disk as it does in memory. Little contrasts are ordinary, and are because of contrasts between arrangement in memory and on disk.

Field	Information Obtained
Imports	Foreign library functions exploited by malware.
Exports	Malware functions to be invoked by other libraries or programs.
Time Date Stamp	Compilation date and time.
Sections	Count of file sections and their in-memory sizes.
Subsystem	Indicates whether the program is a command-line or GUI Application.
Resources	Other data structures utilized by the malware.

Table 2.4: PE Field Sections Information for Basic Analysis

The section sizes can be helpful in identifying packed exe. For instance, if the Virtual Size is a lot bigger than the Size of Raw Data, at that point this is generally a sign of jumbled code, especially if the **.text** section is bigger in memory than on disk. The summary of the PE headers is provided in the table 2.3.1. Therefore, it can be said that the PE header contains the useful information which can be utilized for malware analysis. The features of PE header are therefore served as feature set for the Phase 1 as discussed later.

Chapter 3: Methodology:

The proposed plan is to divide the entire work into two phases [13]. The **First phase** actually gets Ember Dataset 2018 Feature version 2 and Train three Models on this Dataset. Usually, cross-validation also takes place in this stage. The **Second Phase** will be for taking Input PE File and extracting all the usual PE File Structure Data converting from Raw Bytes to Features Data. Calling a saved model as per choice to run the prediction for PE file whether its malicious or non-malicious.

Proposed hypothesis

Given an executable, the system should be able to detect whether it is malicious or not malicious for best possible counter-measure designing.

3.1 Dataset:

A dataset is, basically a collection of data pieces that can be treated by a computer as a solitary unit for prediction and expectation purposes. This implies that the information gathered ought to be made uniform and justifiable for a machine that doesn't see information the same way as humans do.

3.2 Ember Dataset:

The EMBER dataset is a collection of elements from PE records that fill in as a benchmark dataset for scientists. The EMBER2017 dataset contained elements from 1.1 million PE records filtered in or before 2017 and the EMBER2018 dataset contains highlights from 1 million PE documents examined in or before 2018. This archive makes it simple to reproducibly prepare the benchmark models, expand that gave highlight set, or characterize new PE documents with the benchmark models. [12]

```
"sha256": "000185977be72c8b007ac347b73ceb1ba3e5e4dae4fe98d4f2ea92250f7f580e",
"appeared": "2017-01",
"label": -1,
"general": {
  "file_size": 33334,
  "vsize": 45056,
  "has_debug": 0,
  "exports": 0,
  "imports": 41,
  "has_relocations": 1,
  "has_resources": 0,
  "has_signature": 0,
  "has_tls": 0,
  "symbols": 0
},
"header": {
  "coff": {
    "timestamp": 1365446976,
    "machine": "I386",
    "characteristics": [ "LARGE_ADDRESS_AWARE", ..., "EXECUTABLE_IMAGE" ]
  },
  "optional": {
    "subsystem": "WINDOWS_CUI",
    "dll_characteristics": [ "DYNAMIC_BASE", ..., "TERMINAL_SERVER_AWARE" ],
    "magic": "PE32",
    "major_image_version": 1,
    "minor_image_version": 2,
    "major_linker_version": 11,
    "minor_linker_version": 0,
    "major_operating_system_version": 6,
    "minor_operating_system_version": 0,
    "major_subsystem_version": 6,
    "minor_subsystem_version": 0,
    "sizeof_code": 3584,
    "sizeof_headers": 1024,
    "sizeof_heap_commit": 4096
  }
},
"imports": {
  "KERNEL32.dll": [ "GetTickCount" ],
  ...
},
"exports": []
"section": {
  "entry": ".text",
  "sections": [
    {
      "name": ".text",
      "size": 3584,
      "entropy": 6.368472139761825,
      "vsize": 3270,
      "props": [ "CNT_CODE", "MEM_EXECUTE", "MEM_READ"]
    },
    ...
  ]
},
"histogram": [ 3818, 155, ..., 377 ],
"byteentropy": [0, 0, ... 2943 ],
"strings": {
  "numstrings": 170,
  "avlength": 8.170588235294117,
  "printabledist": [ 15, ... 6 ],
  "printables": 1389,
  "entropy": 6.259255409240723,
  "paths": 0,
  "urls": 0,
  "registry": 0,
  "MZ": 1
},
}
```


Figure 3.2: Raw features extracted from a single PE file.

The EMBER dataset comprises of 8 groups of raw features that incorporate both parsed features and histograms and counts of strings. In what follows, we make a differentiation between raw features (the dataset provided) and model features (or vectorized features) derived from the dataset size utilized for preparing a model, addressing a mathematical rundown of the crude highlights, wherein strings, imported names, sent out names, and so on, are caught utilizing the feature hashing trick. [12]

This dataset is of 1.1 Million where we divided it as 1 lac for training and 2 lacs for testing as per requirement. The dataset has 2351 features extracted for Feature Processing.

3.3 Phase 1: Training:

For Training phase, the dataset details of the executables in Json Object Files are extracted using ember own's functions known as create_vectorized_features() and put in the Numpy Arrays .dat file as dataset to save and train the classifiers & models. The dataset is sliced into 6 individual formats for Training, Validation and Testing. Then, we call our Models to train on them. We save the models data and its weights results for further evaluation and testing.

After extracting all the necessary information, the following tasks are performed sequentially:

- Select the relevant features by splitting the data into X and Y Coordinates and Train, Validation set and Test set to avoid overfitting.

```
Loading data...
(200000, 2351)
(100000, 2351)
Train/Val/Test: 144000/16001/100000
time: 1min 15s (started: 2022-02-03 09:12:40 +00:00)
```

Figure 3.3: Training, Validation and Testing Set Divided

- Calling the Model to be trained on this dataset to measure accuracies and other results.
- Validation testing is done in fitting the data run over every epochs and best training model data and weights accuracy as per parameter val_accuracy is saved during training.
- At the end, it gives the approximate Accuracy result of the data on specified Model.

3.4 Phase 2: Testing Input File:

For testing detection phase, the header details of one executable are extracted using ember's own feature extractor known as feature_extractor() and the scaled data as per model scaling is called in the code. There are several available tools which allow us to do this job. It is achieved by PE reader, Python PEfile library, or Ember has its own Feature Extraction PE File Library. After extracting the Raw Bytes,

After extracting all the necessary information, the following tasks are performed sequentially:

- Load the saved model data and its scaler data to avoid overfit the raw bytes extracted.
- Call the model to perform prediction on Raw Features Extracted from the PE File.

Chapter 4: Design/Architecture and Algorithms Details:

4.1: Multilayer Perceptron/Artificial Neural Networks:

The basic building block of a neural network is a neuron. You might have known about neurons in the brain. We realize that the human brain is comprised of 10s of billions of neurons associated together in a network. A biological neuron is a sort of cell that gets multiple inputs through parts called dendrites. Some kind of processing takes place inside the neuron to conclude whether it ought to "fire" - and that implies convey an output signal along its axon. More often than not, the axon is associated with a dendrite of another neuron, making up the network.

An artificial neuron is an exceptionally worked or simplified model of a biological neuron. It comprises of multiple ((k)) input values. Each input value ((f_i)) is multiplied by a different weight ((w_i)). These scaled input values are then summed up together and we add on a constant called the bias ((b)). Finally, a function (usually a nonlinear function) is applied to this summed value and that provides the single output value of the neuron.

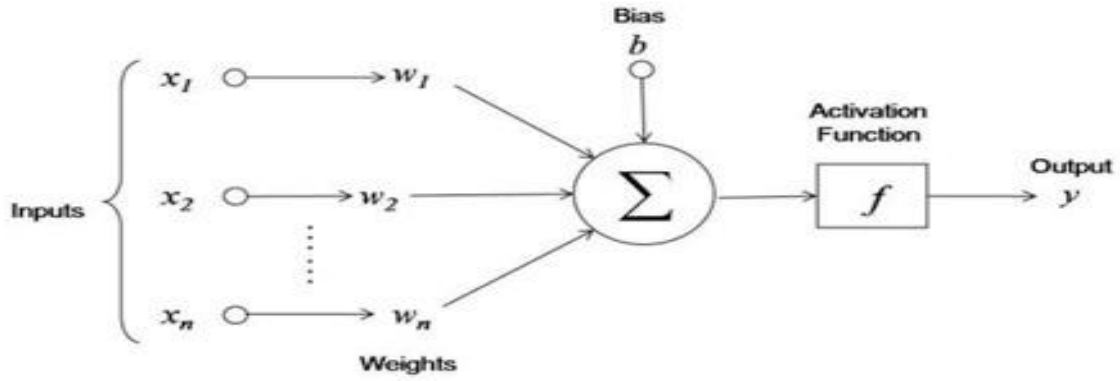


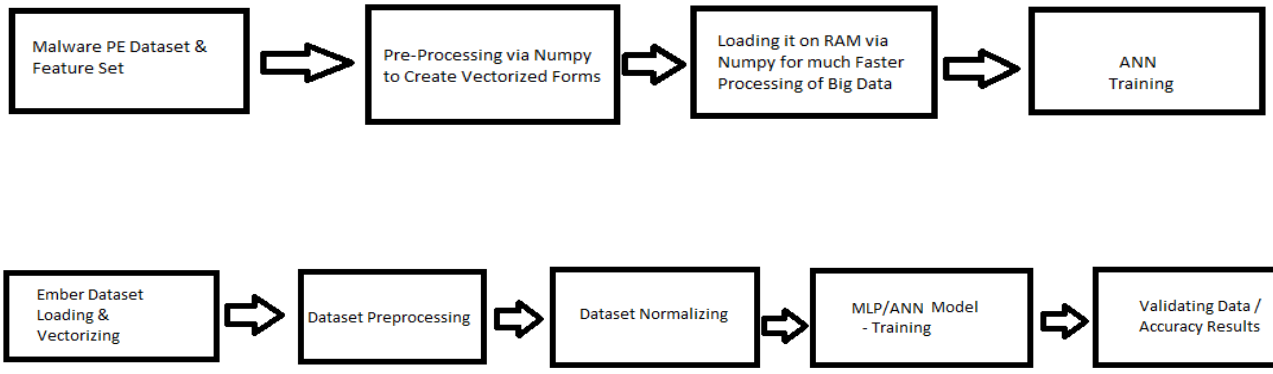
Figure 4.1: Basic Working of ANN

An ANN or Multi-layer Perceptron has input and output layers, and at least one hidden layer with numerous neurons stacked together. And keeping in mind that in the Perceptron the neuron should have an activation function that forces a limit, as ReLU or sigmoid.

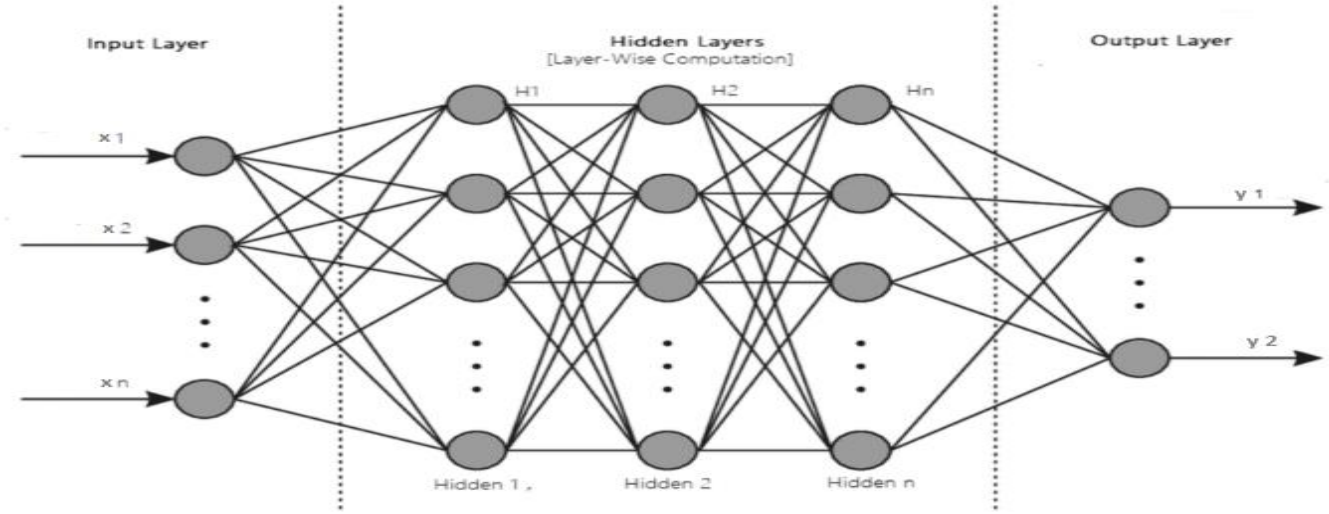
The base architecture of deep learning, which is also known as the feed-forward artificial neural network, is called a multilayer perceptron (MLP). A typical MLP is a fully connected network consisting of an input layer, one or more hidden layers, and an output layer, as shown in Fig 4.1. Each node in one layer connects to each node in the following layer at a certain weight. MLP utilizes the "Backpropagation" technique, the most "fundamental building block" in a neural network, to adjust the weight values internally while building the model. MLP is sensitive to scaling features and allows a variety of hyperparameters to be tuned, such as the number of hidden layers, neurons, and iterations, which can result in a computationally costly model.

This type of neural networks is one of the simplest variants of neural networks as they pass information in one direction through various input nodes, until it makes it to the output node.

4.1.1: Flow Chart of ANN/MLP:



4.1.2: Architecture Diagram of ANN:



4.1.3: Multilayer Perceptron/ANN Architecture:

Input Layer: Input Shape: (2351,1), Activation: ReLU

Hidden Layer: 64 Neurons, Activation: ReLU

Hidden Layer: 32 Neurons, Activation: ReLU

Hidden Layer: 16 Neurons, Activation: ReLU

Hidden Layer: 8 Neurons, Activation: ReLU

Hidden Layer: 4 Neurons, Activation: ReLU

Output Layer: 2 Neurons, Activation: Sigmoid

In this case, we have 2351 features and 5 hidden layers. The first hidden layer comprises of 64 neurons and then 32, 16, 8, and 4 respectively. Then we have 2 output neurons. The output has 2 classes with sigmoid activation function where it contains the accuracy of malicious and benign data from among the data we provided it as input.

The output of one layer serves as an input to the next layer. First the datasets are converted into data objects files and split as per requirement.

After the dataset is split, its then transformed and loaded as a numpy array. Its then further split into Train, Validation data for Validation and testing remains same. The scaler Transform method cleans scale the dataset to align everything and transform the data into its required form for neural networks and a dump file of dataset is saved as scaler.pkl - a pickle file which will be further used in testing and evaluation phases as well.

Then it goes through a Net of Neural Networks, by which the Neural Networks in our case its MLP learns through given parameters in the scale of the dataset and cross validate as well on the validation data for training.

After the Training has been done, on every epoch if the latter is better than before the model structure, data and weights are being saved as the best one. Then the evaluation part takes place and the scaler.pkl file is called for scaler dataset to measure across dimensionality and reassure the data size and the best model (user choice) is called which was saved before. Then it will validate the model weights on Testing data for prediction

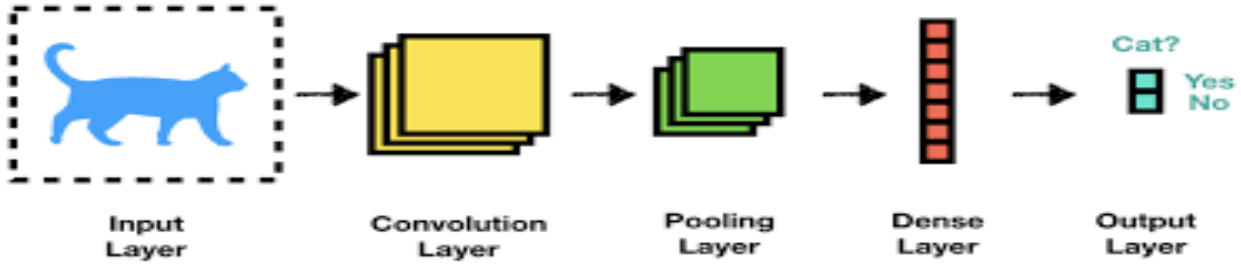
After this, Testing takes place. We Take an input file pass it, scaler.pkl file for specific neural networks and saved model. Then it extracts raw bytes features from the PE file through ember own function feature_extractor() and further matched with the nature either of benign or malicious. Whichever matches it, it will get the label and score of benign or malicious Thus, testing of a PE file has been done.

4.2 Convolutional Neural Networks(1D):

The architecture of a convolutional neural network is a multi-layered feed-forward neural network, made by stacking many hidden layers on top of each other in sequence. It is this sequential design that allows convolutional neural networks to learn hierarchical features.

CNN works well for identifying simple patterns within your data which will then be used to form more complex patterns within higher layers. A 1D CNN is very effective when you expect to derive interesting features from shorter (fixed-length) segments of the overall data set and where the location of the feature within the segment is not of high relevance.

This is how it works:



A convolutional neural network (CNN) is a specific form of synthetic neural community that uses perceptron, a device gaining knowledge of unit algorithm, for supervised studying, to examine records. CNNs apply to picture processing, herbal language processing and different types of cognitive obligations.

A convolutional neural community is likewise referred to as a ConvNet. Like other types of artificial neural networks, a convolutional neural network has an input layer, an output layer and numerous hidden layers. Some of these layers are convolutional, the usage of a mathematical model to pass on consequences to successive layers. This simulates some of the movements inside the human visual cortex.

CNNs are an essential example of deep studying, where a extra sophisticated version pushes the evolution of synthetic intelligence by means of imparting systems that simulate exceptional kinds of biological human brain interest.

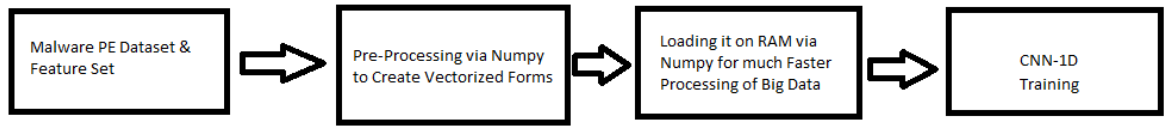
Technically, Deep learning CNN fashions to teach and check, each input image will pass it through a sequence of convolution layers with filters (Kernels), Pooling, fully related layers (FC) and observe Softmax characteristic to categorize an object with probabilistic values among zero and 1. The below discern is a whole go with the flow of CNN to process an enter photo and classifies the items based totally on values.

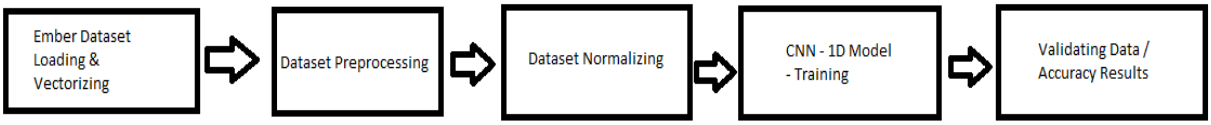
Most Common Convolutional Neural Network types are:

- CNN-1D (1 Dimensional)
- CNN-2D (2 Dimensional)

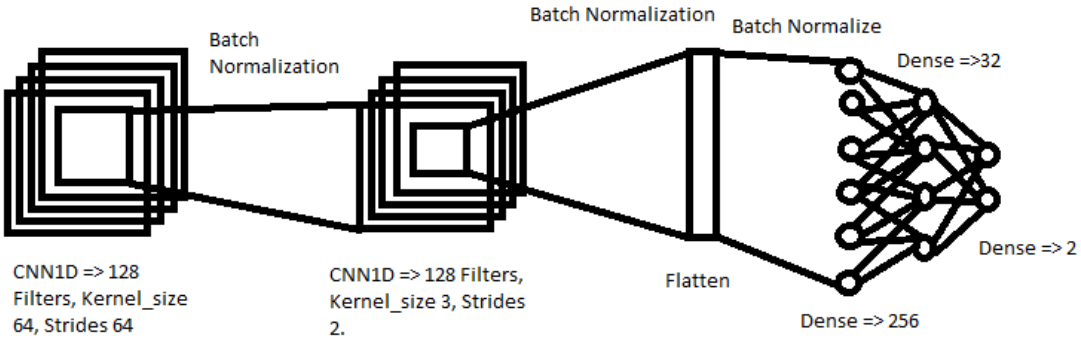
We will be using in our Case Convolutional Neural Networks 1-Dimensional.

4.2.1: CNN Flow chart:





4.2.2: CNN Architecture Diagram:



4.2.3: CNN Architecture:

We will be using CNN-1D for our Computational Model. Our Architecture will be as Follows:

CNN1D Layer => 128 Filters => Kernel_size 64 => Strides 64 => Activation ReLU

BatchNormalization

CNN1D Layer => 128 Filters => Kernel_size 3 => Strides 2 => Activation ReLU

BatchNormalization

Flattening Layer

Dense Layer => 256 Neural Nets => Activation ReLU

BatchNormalization

Dense Layer => 32 Neural Nets => Activation ReLU

BatchNormalization

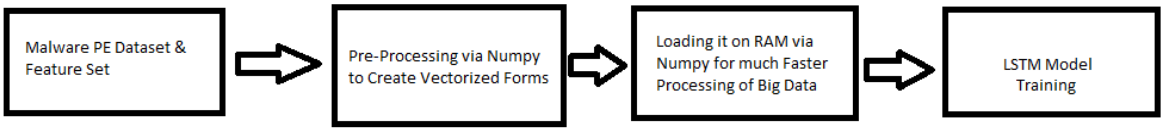
Dense Layer => 2 Neural Nets => Activation ReLU

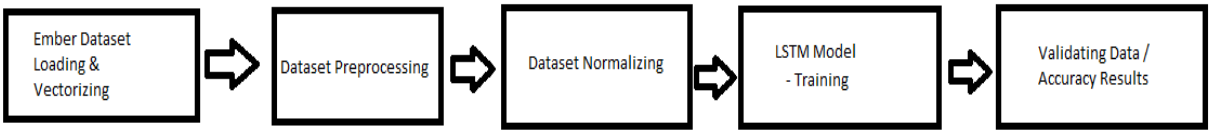
Our Approach may be by the use of Convolutional Neural Networks – 1 Dimensional Computational version for our deployment and task.

4.3: Long Short-Term Memory:

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. ... LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series.

4.3.1: Flow Chart:

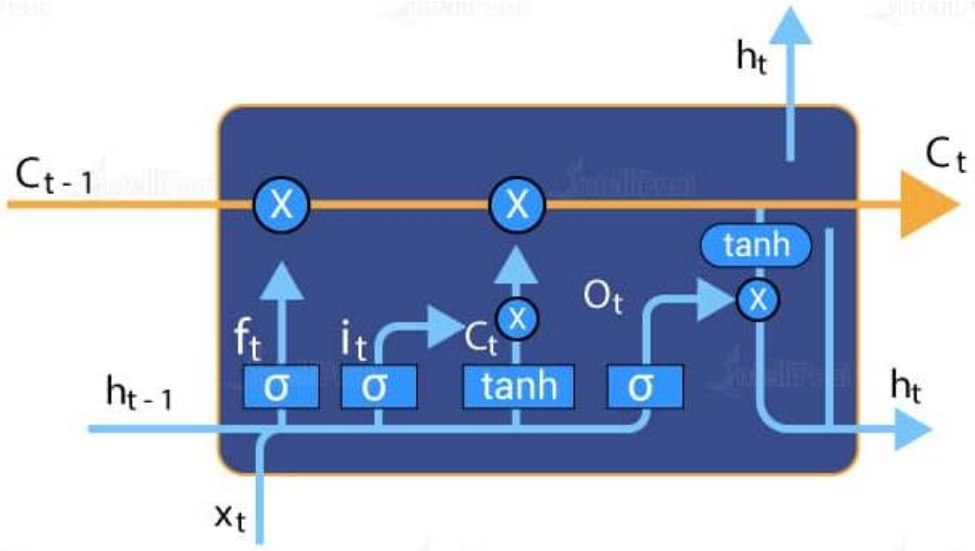




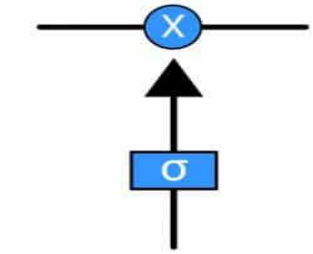
LSTM stands for long short-term memory networks, used in the field of Deep Learning. It is a variety of recurrent neural networks (RNNs) that are capable of learning long-term dependencies, especially in sequence prediction problems. LSTM has feedback connections, i.e., it is capable of processing the entire sequence of data, apart from single data points such as images. This finds application in speech recognition, machine translation, etc. LSTM is a special kind of RNN, which shows outstanding performance on a large variety of problems.

The Logic Behind LSTM

The central role of an LSTM model is held by a memory cell known as a 'cell state' that maintains its state over time. The cell state is the horizontal line that runs through the top of the below diagram. It can be visualized as a conveyor belt through which information just flows, unchanged.

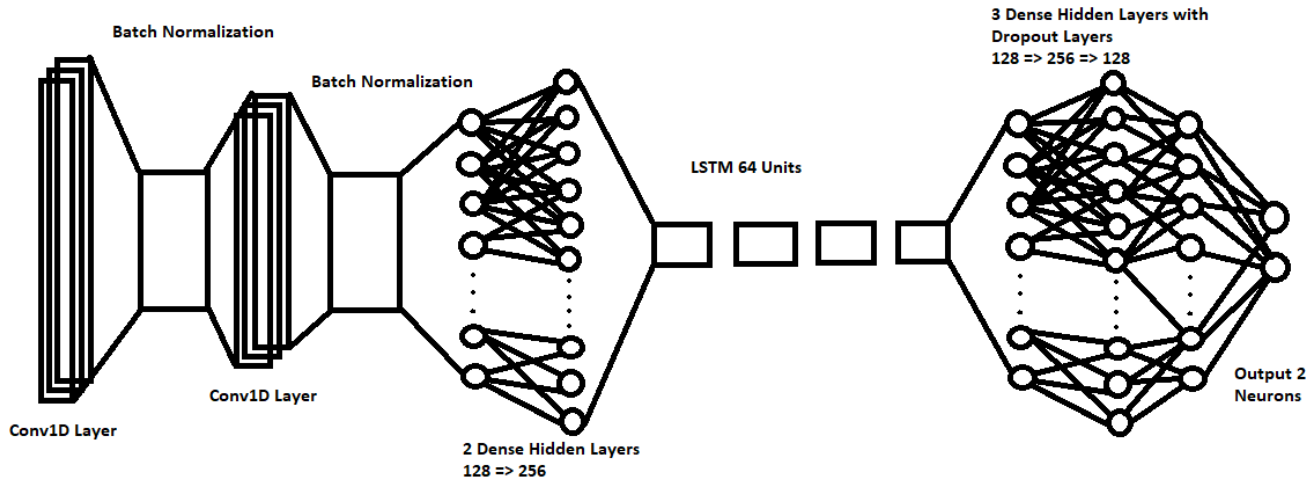


Information can be added to or removed from the cell state in LSTM and is regulated by gates. These gates optionally let the information flow in and out of the cell. It contains a pointwise multiplication operation and a sigmoid neural net layer that assist the mechanism.



The sigmoid layer gives out numbers between zero and one, where zero means ‘nothing should be let through,’ and one means ‘everything should be let through.

4.3.2: LSTM Architecture Diagram:



4.3.3: LSTM Architecture:

We will be using CNN-1D for our Computational Model. Our Architecture will be as Follows:

CNN1D Layer => 128 Filters => Kernel_size 64 => Strides 64 => Activation ReLU

BatchNormalization

CNN1D Layer => 128 Filters => Kernel_size 3 => Strides 2 => Activation ReLU

BatchNormalization

Flattening Layer

Dense Layer => 128 Neural Nets => Activation ReLU

BatchNormalization

Dense Layer => 256 Neural Nets => Activation ReLU

BatchNormalization

LSTM Layer => 64 Units => dropout 0.2

BatchNormalization

Dense Layer => 128 Neural Nets => Activation ReLU

Dense Layer => 256 Neural Nets => Activation ReLU

Dropout 0.2

Dense Layer => 128 Neural Nets => Activation ReLU

Dropout 0.1

Dense Layer => 2 Neural Nets => Activation Sigmoid

Our Approach may be by the use of Convolutional Neural Networks – 1 Dimensional with LSTM was due to increasing the Performance of Training. As CNN-1D will perform the Feature set extraction which makes it easier for LSTM to sequentially learn and train.

Chapter 5: Implementation Phase:

5.1: Tools/Environment:

We will start by describing the Environment in which we will explain how we will be deploying our Detection System using Deep Learning.

We will be using for Environment:

- Python 3+
- Ember 0.10.0 (Feature Files)
- Tensorflow 2
- Lief 0.10.1

The Hardware Minimum Requirements are:

- CPU: Intel Core Xeon 3.10 GHz
- RAM: 16-64 GB DDR3
- GPU: Nvidia GTX 960Ti
- HardDrive: 15 GB for Dataset Minimum

The Implementation & methodology will be explained as below in steps from Dataset Loading to Preprocessing, then to Training and Analysis.

The coding directory should be as below:

- Project
    - emberdataset/
        - the dataset should be placed here
    - emberfiles/
        - __init__.py
        - features.py
        - create_data.py
    - cnn_model/
        - training.py
        - evaluation.py
        - testing.py
    - ann_model/
        - training.py
        - evaluation.py
        - testing.py
    - lstm_model/
        - training.py
        - evaluation.py
        - testing.py

The implementation will be discussed below thoroughly to explain the major steps of how the Models are to be implemented.

5.2: CNN Model:

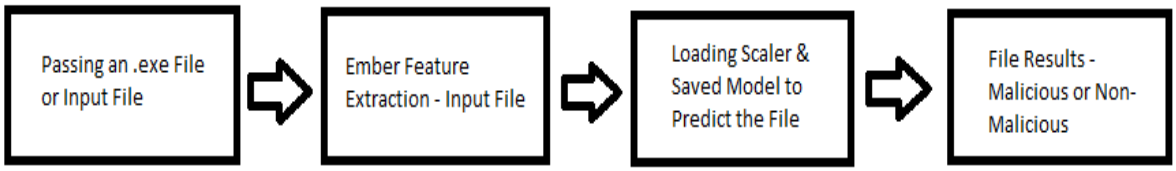
We will be First Extracting the Dataset of Ember 2018 Feature model 2 as described inside the Paper for the provision of this dataset. Extracting it into vectors and .dat file for the advent of Data Objects. We also can convert it into CSV Files but for quicker get right of entry to and overall performance, we chose the default method of Data Objects. [13]

After the Creating the Data Objects. We will building up our Neural Net Architecture which incorporates 2 Conv1D Layers, with BatchNormalization() on each step and knocking down it for Two Dense Sequential Layers. [13]

After the dataset extraction part. We will Train the Neural Nets with Visualization of Graph to examine the Accuracy Results.

For Cross-validation, we will be evaluating through the best performing saved data of model to evaluate and Measure the Accuracy through Graph Visualization.

Testing phase of .exe files will take place after Evaluation. The process will be passing an input file to the python code. Using Ember Feature Extractor, the features will be further matched and compared with the results of current features of the file. A condition will be placed for analyzing the nature of file and labelling through either Malicious Features or Non-Malicious Features.



5.2.1: CNN Implementation:

First, we will Extract the Dataset and turn it into Vectorized form by running create_data.py file and then Save it using Ember Features File calling a function:

```
ember.create_vectorized_features( path_to_dir , scale = 1. )
```

It will take time to create vectorized data objects files. After Creating, we will then Read them to Access the Dataset and save it for Training.

```
ember.read_vectorized_features( path_to_data_objects , scale = 1.)
```

In training.py file, before passing the dataset into Model, we will restructure the data into Training, Testing, and Validation Data. These below codes will be used for Data Cleaning (Just keeping supervised data) and splitting the dataset as well into train, validation set, and test set.

```
idx = int((1. - split) * X_train.shape[0])
X_val, y_val = X_train[idx:], y_train[idx:]
X_train, y_train = X_train[:idx], y_train[:idx]


X_test = X_test[y_test != -1]
y_test = y_test[y_test != -1]
```

After the transformation using scaler_transform, we will scale the data to Normalize it.

```
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
if not os.path.exists(save_dir):
    os.mkdir(save_dir)
```

```
with open(os.path.join(save_dir, 'scaler.pkl'), 'wb') as f:
    pkl.dump(scaler, f)
```

Then, The Data should be Transformed into 3-Dimensional Data formation using Numpy Function .expand_dims(Training / Testing / Validation Data Nodes / axis = -1)

```
X_train = np.expand_dims(X_train, axis=-1)
X_val = np.expand_dims(X_val, axis=-1)
X_test = np.expand_dims(X_test, axis=-1)
```

Our Model will be training on Training Features & Testing Features with Validation Data which will also Save the Best Performing Epochs Result. After, the Training we can re-assure the Results by Visualized Graphs that must be shown or saved in any Directory (if needed).

Model Structure will be as below to be implemented by Keras modelling:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv1d_2 (Conv1D)            (None, 36, 128)           8320

 batch_normalization_4 (Batc  (None, 36, 128)          512
 hNormalization)

 conv1d_3 (Conv1D)            (None, 17, 128)           49280

 batch_normalization_5 (Batc  (None, 17, 128)          512
 hNormalization)

 flatten_1 (Flatten)          (None, 2176)              0

 dense_3 (Dense)              (None, 256)               557312

 batch_normalization_6 (Batc  (None, 256)              1024
 hNormalization)

 dense_4 (Dense)              (None, 32)                8224

 batch_normalization_7 (Batc  (None, 32)               128
 hNormalization)

 dense_5 (Dense)              (None, 2)                 66

=================================================================
Total params: 625,378
Trainable params: 624,290
Non-trainable params: 1,088
_________________________________________________________________
Batch size: 64
Epochs: 10
Learning rate: 0.001
Weight decay: 0.0005
```

This above Image shows the actual implemented architecture of CNN-1D.

After the Training, we will have the Best Models Data & Weights being saved in a Directory by making historical checkpoints method in keras for model saving. We can then Further call the model.h5 file for our evaluation in evaluation.py file and after assuring the evaluation by passing the Test Features for Accuracy measurements.

We will be Testing the Neural Nets we made by Passing an Actual .exe File.

For this process, we will first pass the input file to testing.py file which we have created in the model directory.

```
input_file = "filename.exe"
```

Then we call the CNN-1D model which is saved for the best accuracy got while training on dataset. The model should be loaded and the model.summary() should be called for assuring the model architecture.

```
model_path = "modelpath/cnn_model.h5"

model = load_model(model_path)

model.summary()
```

Features of the File will be Extracted from the Ember Features Code and by following method.

```
extractor = ember.features.PEFeatureExtractor()

with open(input_file, 'rb') as f:

    raw_bytes = f.read()

feature = np.array(extractor.feature_vector(raw_bytes), dtype=np.float32)
```

And then after the extraction of Features of input file. The scaler.pkl file which was also saved for data scaling during the training phases for the model should be called to scale this data again for the same model.

```
scaler_path = "model_path/scaler.pkl"

with open(scaler_path, 'rb') as f:

    scaler = pkl.load(f)
```

Features set should be reshaped for CNN and LSTM both because these two models requires Three Dimensional Inputs.

```
features = scaler.transform(features)

features = np.expand_dims(features, axis=-1)
```

Prediction on File Data will take Place. Either it will Detect it as Benign or Malware.

```
score = model.predict(features)[0]

if score[-1] < args.threshold:

  print('Score: %.5f -> Benign' % score[-1])

else:

  print('Score: %.5f -> Malicous' % score[-1])
```

5.2.2: Results:

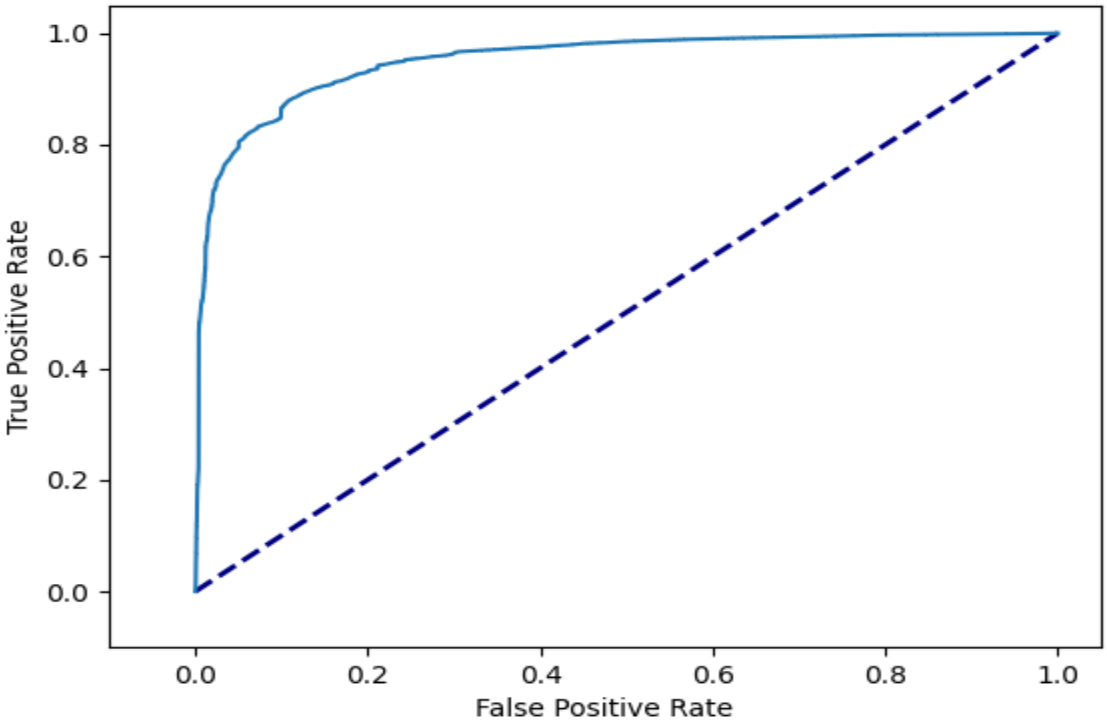
Giving batch size and epochs number to train our neural network:

```
Batch size: 64
Epochs: 15
Learning rate: 1e-05
Weight decay: 0.0005
2022-02-03 22:51:54.607201: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Epoch 1/15
2250/2250 [==============================] - 79s 34ms/step - loss: 0.6306 - accuracy: 0.8660 - val_loss: 0.5379 - val_accuracy: 0.9156

Epoch 00001: val_accuracy improved from -inf to 0.91557, saving model to .\ember\cnn_model.001.h5
Epoch 2/15
2250/2250 [==============================] - 73s 33ms/step - loss: 0.5047 - accuracy: 0.9250 - val_loss: 0.4929 - val_accuracy: 0.9323

Epoch 00002: val_accuracy improved from 0.91557 to 0.93225, saving model to .\ember\cnn_model.002.h5
Epoch 3/15
2250/2250 [==============================] - 69s 31ms/step - loss: 0.4652 - accuracy: 0.9404 - val_loss: 0.4681 - val_accuracy: 0.9413

Epoch 00003: val_accuracy improved from 0.93225 to 0.94125, saving model to .\ember\cnn_model.003.h5
Epoch 4/15
2250/2250 [==============================] - 69s 30ms/step - loss: 0.4404 - accuracy: 0.9482 - val_loss: 0.4463 - val_accuracy: 0.9473

Epoch 00004: val_accuracy improved from 0.94125 to 0.94725, saving model to .\ember\cnn_model.004.h5
Epoch 5/15
2250/2250 [==============================] - 68s 30ms/step - loss: 0.4209 - accuracy: 0.9540 - val_loss: 0.4333 - val_accuracy: 0.9503
```

After completion of epochs if the training accuracy increase from the recent saved model then it will save it in new file otherwise it'll show us that the accuracy did not improve.

```
Epoch 00012: val_accuracy improved from 0.95950 to 0.96131, saving model to .\ember\cnn_model.012.h5
Epoch 13/15
2250/2250 [==============================] - 68s 30ms/step - loss: 0.3280 - accuracy: 0.9757 - val_loss: 0.3748 - val_accuracy: 0.9614

Epoch 00013: val_accuracy improved from 0.96131 to 0.96144, saving model to .\ember\cnn_model.013.h5
Epoch 14/15
2250/2250 [==============================] - 67s 30ms/step - loss: 0.3190 - accuracy: 0.9775 - val_loss: 0.3679 - val_accuracy: 0.9628

Epoch 00014: val_accuracy improved from 0.96144 to 0.96275, saving model to .\ember\cnn_model.014.h5
Epoch 15/15
2250/2250 [==============================] - 68s 30ms/step - loss: 0.3117 - accuracy: 0.9785 - val_loss: 0.3653 - val_accuracy: 0.9626

Epoch 00015: val_accuracy did not improve from 0.96275
```

For Cross-validation, we will be evaluating through the best performing saved data of model to evaluate and Measure the Accuracy through Graph Visualization.

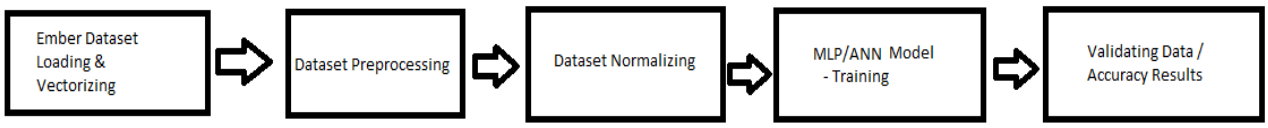


5.3: ANN Model:

We will be First Extracting the Dataset of Ember 2018 Feature model 2 as described inside the Paper for the provision of this dataset. Extracting it into vectors and .dat file for the advent of Data Objects. We also can convert it into CSV Files but for quicker get right of entry to and overall performance, we chose the default method of Data Objects. [13]

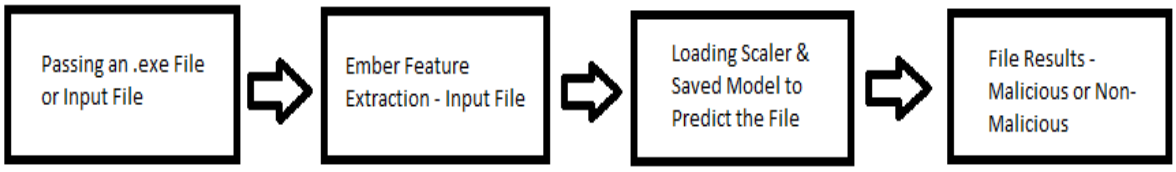
After the Creating the Data Objects. We will building up our Neural Net Architecture which incorporates 5 hidden Layers consisting of Input Layer => Dense Layer of 64 Neurons followed by Batch Normalization Layer => then Dense 32 and Batch Normalization Layer => Dense 16 and Batch Normalization Layer => Dense 8 and Batch Normalization Layer => Dense 4 and the last one as an Output Layer which has 2 Neurons.

After the dataset extraction part. We will Train the Neural Nets with Visualization of Graph to examine the Accuracy Results.



For Cross-validation, we will be evaluating through the best performing saved data of model to evaluate and Measure the Accuracy through Graph Visualization.

Testing phase of .exe files will take place after Evaluation. The process will be passing an input file to the python code. Using Ember Feature Extractor, the features will be further matched and compared with the results of current features of the file. A condition will be placed for analyzing the nature of file and labelling through either Malicious Features or Non-Malicious Features.



5.3.1: ANN Implementation:

First, we will Extract the Dataset and turn it into Vectorized form by running create_data.py file and then Save it using Ember Features File calling a function:

```
ember.create_vectorized_features( path_to_dir , scale = 1. )
```

It will take time to create vectorized data objects files. After Creating, we will then Read them to Access the Dataset and save it for Training.

```
ember.read_vectorized_features( path_to_data_objects , scale = 1.)
```

In training.py file, before passing the dataset into Model, we will restructure the data into Training, Testing, and Validation Data. These below codes will be used for Data Cleaning (Just keeping supervised data) and splitting the dataset as well into train, validation set, and test set.

```
idx = int((1. - split) * X_train.shape[0])
X_val, y_val = X_train[idx:], y_train[idx:]
X_train, y_train = X_train[:idx], y_train[:idx]


X_test = X_test[y_test != -1]
y_test = y_test[y_test != -1]
```

After the transformation using scaler_transform, we will scale the data to Normalize it.

```
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
if not os.path.exists(save_dir):
    os.mkdir(save_dir)
with open(os.path.join(save_dir, 'scaler.pkl'), 'wb') as f:
    pkl.dump(scaler, f)
```

Then, the data dimensionality is skipped in this part for MLP because the Data it requires is in 2-Dimensional Form. As the data is in 2-Dimesions Form, then there is no need to reshape the dimensions.

Our Model will be training on Training Features & Testing Features with Validation Data which will also Save the Best Performing Epochs Result. After, the Training we can re-assure the Results by Visualized Graphs that must be shown or saved in any Directory

Model Structure will be as below to be implemented by Keras modelling:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                150528
_________________________________________________________________
batch_normalization (BatchNo (None, 64)                256
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
batch_normalization_1 (Batch (None, 32)                128
_________________________________________________________________
dense_2 (Dense)              (None, 16)                528
_________________________________________________________________
batch_normalization_2 (Batch (None, 16)                64
_________________________________________________________________
dense_3 (Dense)              (None, 8)                 136
_________________________________________________________________
batch_normalization_3 (Batch (None, 8)                 32
_________________________________________________________________
dense_4 (Dense)              (None, 4)                 36
_________________________________________________________________
dense_5 (Dense)              (None, 2)                 10
=================================================================
Total params: 153,798
```

This above figure shows the actual implemented architecture of ANN/MLP.

After the Training, we will have the Best Models Data & Weights being saved in a Directory by making historical checkpoints method in keras for model saving. We can then Further call the model.h5 file for our evaluation in evaluation.py file and after assuring the evaluation by passing the Test Features for Accuracy measurements.

We will be Testing the Neural Nets we made by Passing an Actual .exe File.

For this process, we will first pass the input file to testing.py file which we have created in the model directory.

input_file = "filename.exe"

Then we call the ANN/MLP model which is saved for the best accuracy got while training on dataset. The model should be loaded and the model.summary() should be called for assuring the model architecture.

model_path = "modelpath/ann_model.h5"

model = load_model(model_path)

```
model.summary()
```

Features of the File will be Extracted from the Ember Features Code and by following method.

```
extractor = ember.features.PEFeatureExtractor()
with open(input_file, 'rb') as f:
    raw_bytes = f.read()
feature = np.array(extractor.feature_vector(raw_bytes), dtype=np.float32)
```

And then after the extraction of Features of input file. The scaler.pkl file which was also saved for data scaling during the training phases for the model should be called to scale this data again for the same model.

```
scaler_path = "model_path/scaler.pkl"
with open(scaler_path, 'rb') as f:
    scaler = pkl.load(f)
```

Features set should not be reshaped as it is done for CNN and LSTM because ANN/MLP require 2-Dimensional input. Hence, dimensionality reshaping is skipped.

```
features = scaler.transform(features)
```

Prediction on File Data will take Place. Either it will Detect it as Benign or Malware.

```
score = model.predict(features)[0]
if score[-1] < args.threshold:
  print('Score: %.5f -> Benign' % score[-1])
else:
  print('Score: %.5f -> Malicous' % score[-1])
```

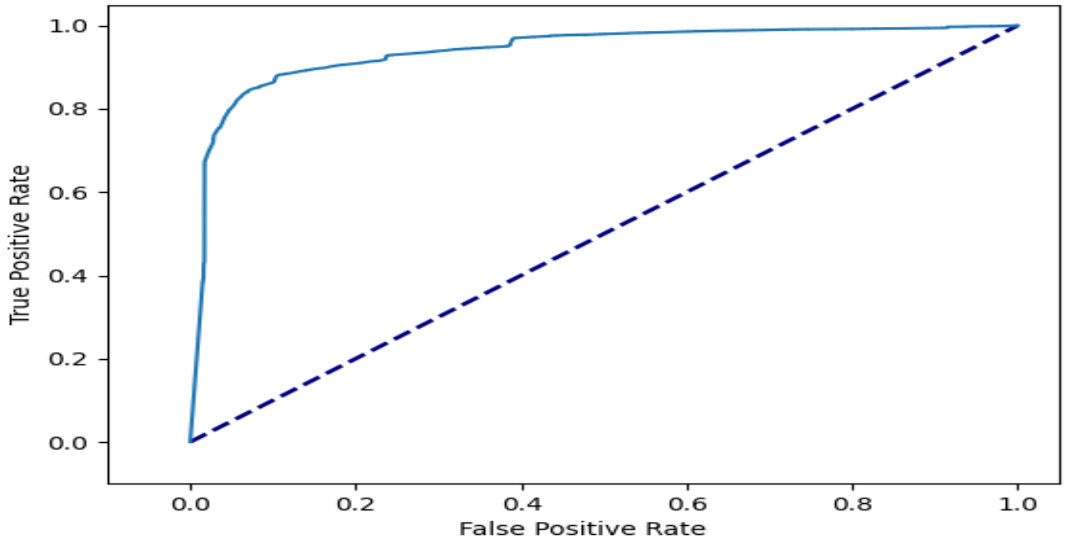
5.3.2: ANN Results:

Giving batch size and epochs number to train our neural network:

```
Epoch 00005: val_accuracy did not improve from 0.95100
Epoch 6/10
2250/2250 [==============================] - 8s 4ms/step - loss: 0.1370 - accuracy: 0.9612 - val_loss: 0.1518 - val_accuracy: 0.9548

Epoch 00006: val_accuracy improved from 0.95100 to 0.95475, saving model to ann_model\ann_model.006.h5
Epoch 7/10
2250/2250 [==============================] - 8s 4ms/step - loss: 0.1186 - accuracy: 0.9654 - val_loss: 0.1405 - val_accuracy: 0.9584

Epoch 00007: val_accuracy improved from 0.95475 to 0.95844, saving model to ann_model\ann_model.007.h5
Epoch 8/10
2250/2250 [==============================] - 9s 4ms/step - loss: 0.1074 - accuracy: 0.9683 - val_loss: 0.1350 - val_accuracy: 0.9593

Epoch 00008: val_accuracy improved from 0.95844 to 0.95932, saving model to ann_model\ann_model.008.h5
Epoch 9/10
2250/2250 [==============================] - 9s 4ms/step - loss: 0.0973 - accuracy: 0.9700 - val_loss: 0.1317 - val_accuracy: 0.9580

Epoch 00009: val_accuracy did not improve from 0.95932
Epoch 10/10
2250/2250 [==============================] - 8s 4ms/step - loss: 0.0879 - accuracy: 0.9723 - val_loss: 0.1291 - val_accuracy: 0.9599

Epoch 00010: val_accuracy improved from 0.95932 to 0.95988, saving model to ann_model\ann_model.010.h5
```

This is the accuracy getting after training our model and it is saving to the model file ann_model.010.h5.

For Cross-validation, we will be evaluating through the best performing saved data of model to evaluate and Measure the Accuracy through Graph Visualization.



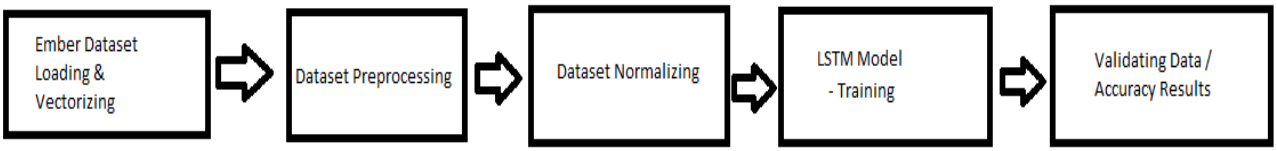
5.4: LSTM Model Implementation:

We will be First Extracting the Dataset of Ember 2018 Feature model 2 as described inside the Paper for the provision of this dataset. Extracting it into vectors and .dat file for the advent of Data Objects. We also can convert it into CSV Files but for quicker get right of entry to and overall performance, we chose the default method of Data Objects. [13]

After the Creating the Data Objects. We will building up our Neural Net Architecture which incorporates 2 Conv1D Layers with 128 Filters, 64 kernel Size and Strides 64 and the second layer with 3 kernel size and 2 Strides , with BatchNormalization() on each step => Followed by Dense layer of 256 Neurons and Batch Normalization Layer => Followed by 64 Units LSTM Cells with Dropout Layer of rate 0.2 (20% Dropout) => then attached with Dense Layer of 128 Neurons => Dense 256 followed by dropout layer of rate 0.2 (20% Dropout) and => Dense Layer of 128 Neurons again with Dropout 0.1 rate (10% Dropout) => All coming the Dense layer of 2 Neurons for the output layer.

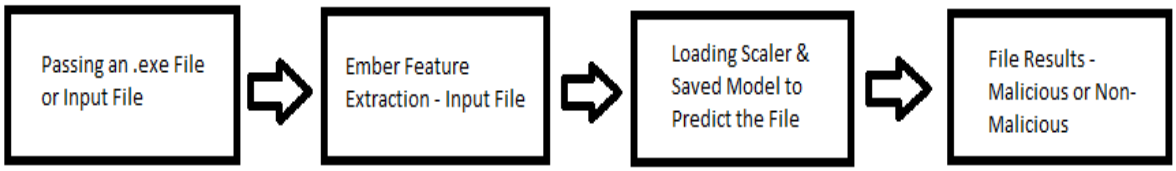
The Convolutional Layers and Dense Layers were added just for the Extraction of Features set from the Dataset which makes the LSTM Cells Sequential Learning easier to train.

After the dataset extraction part. We will Train the Neural Nets with Visualization of Graph to examine the Accuracy Results.



For Cross-validation, we will be evaluating through the best performing saved data of model to evaluate and Measure the Accuracy through Graph Visualization.

Testing phase of .exe files will take place after Evaluation. The process will be passing an input file to the python code. Using Ember Feature Extractor, the features will be further matched and compared with the results of current features of the file. A condition will be placed for analyzing the nature of file and labelling through either Malicious Features or Non-Malicious Features.



5.4.1: LSTM Implementation:

First, we will Extract the Dataset and turn it into Vectorized form by running create_data.py file and then Save it using Ember Features File calling a function:

```
ember.create_vectorized_features( path_to_dir , scale = 1. )
```

It will take time to create vectorized data objects files. After Creating, we will then Read them to Access the Dataset and save it for Training.

```
ember.read_vectorized_features( path_to_data_objects , scale = 1.)
```

In training.py file, before passing the dataset into Model, we will restructure the data into Training, Testing, and Validation Data. These below codes will be used for Data Cleaning (Just keeping supervised data) and splitting the dataset as well into train, validation set, and test set.

```
idx = int((1. - split) * X_train.shape[0])
X_val, y_val = X_train[idx:], y_train[idx:]
X_train, y_train = X_train[:idx], y_train[:idx]


X_test = X_test[y_test != -1]
y_test = y_test[y_test != -1]
```

After the transformation using scaler_transform, we will scale the data to Normalize it.

```
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
if not os.path.exists(save_dir):
    os.mkdir(save_dir)
```

```
with open(os.path.join(save_dir, 'scaler.pkl'), 'wb') as f:
    pkl.dump(scaler, f)
```

Then, The Data should be Transformed into 3-Dimensional Data formation using NumPy Function .expand_dims(Training / Testing / Validation Data Nodes / axis = -1)

```
X_train = np.expand_dims(X_train, axis=-1)
X_val = np.expand_dims(X_val, axis=-1)
X_test = np.expand_dims(X_test, axis=-1)
```

Our LSTM Model will be training on Training Features & Testing Features with Validation Data which will also Save the Best Performing Epochs Result. After, the Training we can re-assure the Results by Visualized Graphs that must be shown or saved in any Directory.

Model Structure will be as below to be implemented by Keras modelling:

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv1d_4 (Conv1D)            (None, 36, 128)           8320

 batch_normalization_7 (Batc  (None, 36, 128)          512
 hNormalization)

 conv1d_5 (Conv1D)            (None, 17, 128)           49280

 batch_normalization_8 (Batc  (None, 17, 128)          512
 hNormalization)

 dense_6 (Dense)              (None, 17, 256)           33024

 batch_normalization_9 (Batc  (None, 17, 256)          1024
 hNormalization)

 lstm_1 (LSTM)                (None, 64)                82176

 batch_normalization_10 (Bat  (None, 64)               256
 chNormalization)

 dense_7 (Dense)              (None, 128)               8320

 dense_8 (Dense)              (None, 256)               33024

 dropout_4 (Dropout)          (None, 256)               0

 dense_9 (Dense)              (None, 128)               32896

 dropout_5 (Dropout)          (None, 128)               0

 dense_10 (Dense)             (None, 2)                 258

=================================================================
Total params: 249,602
Trainable params: 248,450
Non-trainable params: 1,152
```

This above Image shows the actual implemented architecture of LSTM.

After the Training, we will have the Best Models Data & Weights being saved in a Directory by making historical checkpoints method in keras for model saving. We can then Further call the model.h5 file for our evaluation in evaluation.py file and after assuring the evaluation by passing the Test Features for Accuracy measurements.

We will be Testing the Neural Nets we made by Passing an Actual .exe File.

For this process, we will first pass the input file to testing.py file which we have created in the model directory.

```
input_file = "filename.exe"
```

Then we call the LSTM model which is saved for the best accuracy got while training on dataset. The model should be loaded and the model.summary() should be called for assuring the model architecture.

```
model_path = "modelpath/lstm_model.h5"

model = load_model(model_path)

model.summary()
```

Features of the File will be Extracted from the Ember Features Code and by following method.

```
extractor = ember.features.PEFeatureExtractor()

with open(input_file, 'rb') as f:

    raw_bytes = f.read()

feature = np.array(extractor.feature_vector(raw_bytes), dtype=np.float32)
```

And then after the extraction of Features of input file. The scaler.pkl file which was also saved for data scaling during the training phases for the model should be called to scale this data again for the same model.

```
scaler_path = "model_path/scaler.pkl"

with open(scaler_path, 'rb') as f:

    scaler = pkl.load(f)
```

Features set should be reshaped for CNN and LSTM both because these two models require Three Dimensional Inputs.

```
features = scaler.transform(features)

features = np.expand_dims(features, axis=-1)
```

Prediction on File Data will take Place. Either it will Detect it as Benign or Malware.

```
score = model.predict(features)[0]
if score[-1] < args.threshold:
   print('Score: %.5f -> Benign' % score[-1])
else:
   print('Score: %.5f -> Malicous' % score[-1])
```

5.4.2:Results:

Giving batch size and epochs number to train our neural network:

```
Batch size: 256
Epochs: 5
Learning rate: 0.0001
Weight decay: 0.0005
2022-02-04 09:06:45.699053: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Epoch 1/5
563/563 [==============================] - 457s 808ms/step - loss: 0.7816 - accuracy: 0.5966 - val_loss: 0.7399 - val_accuracy: 0.4164

Epoch 00001: val_accuracy improved from -inf to 0.41641, saving model to .\lstm_dir\lstm_model.001.h5
Epoch 2/5
563/563 [==============================] - 455s 809ms/step - loss: 0.6880 - accuracy: 0.6118 - val_loss: 0.7051 - val_accuracy: 0.4331

Epoch 00002: val_accuracy improved from 0.41641 to 0.43310, saving model to .\lstm_dir\lstm_model.002.h5
Epoch 3/5
563/563 [==============================] - 454s 807ms/step - loss: 0.6675 - accuracy: 0.6198 - val_loss: 0.6635 - val_accuracy: 0.6143

Epoch 00003: val_accuracy improved from 0.43310 to 0.61434, saving model to .\lstm_dir\lstm_model.003.h5
Epoch 4/5
563/563 [==============================] - 471s 836ms/step - loss: 0.6591 - accuracy: 0.6217 - val_loss: 0.6569 - val_accuracy: 0.6152

Epoch 00004: val_accuracy improved from 0.61434 to 0.61521, saving model to .\lstm_dir\lstm_model.004.h5
Epoch 5/5
563/563 [==============================] - 457s 812ms/step - loss: 0.6553 - accuracy: 0.6222 - val_loss: 0.6533 - val_accuracy: 0.6152

Epoch 00005: val_accuracy did not improve from 0.61521
```

After completion of epochs if the training accuracy increase from the recent saved model then it will save it in new file otherwise it'll show us that the accuracy did not improve.

```
Epoch 00008: val_accuracy did not improve from 0.98564
168/168 [==============================] - 4s 26ms/step - loss: 0.0699 - accuracy: 0.9871 - val_loss: 0.0808 - val_accuracy: 0.9831
Epoch 9/10
166/168 [============================>.] - ETA: 0s - loss: 0.0668 - accuracy: 0.9872
Epoch 00009: val_accuracy did not improve from 0.98564
168/168 [==============================] - 4s 27ms/step - loss: 0.0669 - accuracy: 0.9872 - val_loss: 0.0736 - val_accuracy: 0.9848
Epoch 10/10
167/168 [============================>.] - ETA: 0s - loss: 0.0638 - accuracy: 0.9878
Epoch 00010: val_accuracy did not improve from 0.98564
168/168 [==============================] - 5s 27ms/step - loss: 0.0639 - accuracy: 0.9877 - val_loss: 0.0724 - val_accuracy: 0.9853
```

For Cross-validation, we will be evaluating through the best performing saved data of model to evaluate and Measure the Accuracy through Graph Visualization.

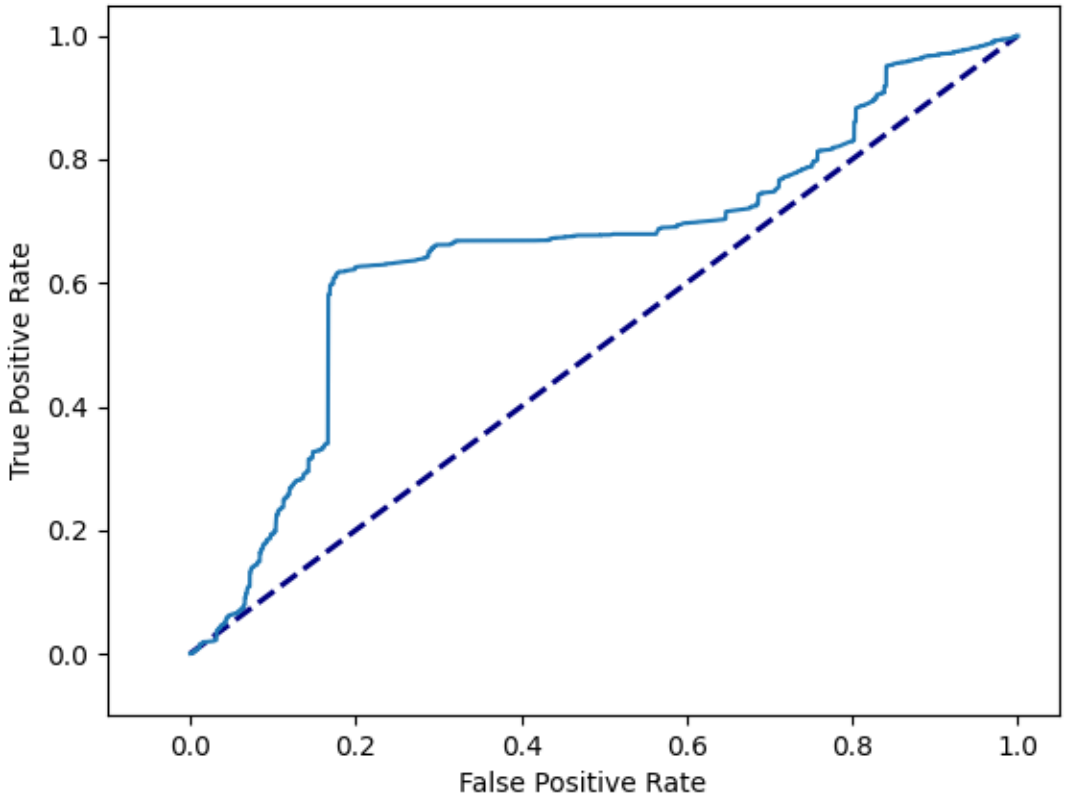


Figure of ROC Curve - LSTM

Chapter 6: Result Analysis:

6.1: ANN/MLP

ANN Results of checking different files (malicious/benign):

Checking Bittorrent.exe

```
Total params: 153,798
Trainable params: 153,558
Non-trainable params: 240

_________________________________________________________________
2022-02-04 07:58:45.728589: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Score: 0.81661 -> Malicous
```

Checking Setupchipset.exe

```
Total params: 153,798
Trainable params: 153,558
Non-trainable params: 240

_________________________________________________________________
2022-02-04 08:00:59.148288: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Score: 0.02410 -> Benign
```

ANN/MLP CPU and GPU time taking difference:

```
Epoch 00010: val_accuracy did not improve from 0.95957
2250/2250 [==============================] - 15s 6ms/step - loss: 0.0604 - accuracy: 0.9760 - val_loss: 0.1246 - val_accuracy: 0.9591
time: 3min 24s (started: 2022-02-03 08:41:55 +00:00)
```

CPU takes 3 min 24 seconds.

```
Epoch 00010: val_accuracy improved from 0.95925 to 0.96175, saving model to /content/drive/MyDrive/FYP2/dataset/ember2018/ann_dir/ann_model_gpu.010.h5
2250/2250 [==============================] - 15s 7ms/step - loss: 0.0573 - accuracy: 0.9777 - val_loss: 0.1227 - val_accuracy: 0.9618
time: 2min 39s (started: 2022-02-03 08:46:52 +00:00)
```

GPU takes 2 min 39 seconds.

```
Epoch 00010: val_accuracy improved from 0.98714 to 0.98821, saving model to /content/drive/MyDrive/FYP2/ember20
168/168 [==============================] - 4s 25ms/step - loss: 0.0230 - accuracy: 0.9925 - val_loss: 0.0477 -
time: 1min 17s (started: 2022-02-13 22:00:27 +00:00)
```

TPU takes 1 min 17 seconds.

6.2: CNN-1D:

CNN Results of checking different files (malicious/benign):

Checking rufus.exe:

```
PS C:\Users\Shahid Khan\PycharmProjects\pythonProject4> python .\test.py --input_file '.\exe files\rufus.exe' --scaler_path .\ember\scaler.pkl --model_path .\ember\cn
n_model.014.h5
2022-02-04 07:53:32.801467: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlerror: cudart64_110.d
ll not found
2022-02-04 07:53:32.801586: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
Example: .\exe files\rufus.exe
Model path: .\ember\cnn_model.014.h5
Loading model from .\ember\cnn_model.014.h5
2022-02-04 07:53:37.169245: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'nvcuda.dll'; dlerror: nvcuda.dll not found

2022-02-04 07:53:37.169771: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2022-02-04 07:53:37.176582: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retrieving CUDA diagnostic information for host: Shahid
2022-02-04 07:53:37.177087: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostname: Shahid
2022-02-04 07:53:37.178079: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN)
to use the following CPU instructions in performance-critical operations:  AVX AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-02-04 07:53:37.649077: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Score: 0.08736 -> Malicous
```

Checking ZoomInstaller.exe:

```
PS C:\Users\Shahid Khan\PycharmProjects\pythonProject4> python .\test.py --input_file '.\exe files\ZoomInstaller.exe' --scaler_path .\ember\scaler.pkl --model_path .\
ember\cnn_model.014.h5
2022-02-04 07:56:03.905888: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlerror: cudart64_110.d
ll not found
2022-02-04 07:56:03.906056: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
Example: .\exe files\ZoomInstaller.exe
Model path: .\ember\cnn_model.014.h5
Loading model from .\ember\cnn_model.014.h5
2022-02-04 07:56:11.418523: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'nvcuda.dll'; dlerror: nvcuda.dll not found

2022-02-04 07:56:11.418810: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2022-02-04 07:56:11.422781: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retrieving CUDA diagnostic information for host: Shahid
2022-02-04 07:56:11.423175: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostname: Shahid
2022-02-04 07:56:11.424005: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN)
to use the following CPU instructions in performance-critical operations:  AVX AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-02-04 07:56:11.791210: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Score: 0.00074 -> Benign
```

CNN CPU and GPU time taking difference:

```
Epoch 00010: val_accuracy did not improve from 0.99890
5621/5621 [==============================] - 174s 31ms/step - loss: 0.0088 - accuracy: 0.9989 - val_loss: 0.0089 - val_accuracy: 0.9989
time: 35min 27s (started: 2022-02-03 06:34:40 +00:00)
```

CPU takes 35 min 27 seconds.

```
Epoch 00010: val_accuracy improved from 0.95444 to 0.95544, saving model to /content/drive/MyDrive/FYP2/dataset/ember2018/cnn_dir/cnn_model_gpu.010.h5
2250/2250 [==============================] - 22s 10ms/step - loss: 0.1662 - accuracy: 0.9594 - val_loss: 0.2020 - val_accuracy: 0.9554
time: 4min 26s (started: 2022-02-03 07:18:58 +00:00)
```

GPU takes 4 min 26 seconds.

```
Epoch 00010: val_accuracy did not improve from 0.98590
168/168 [==============================] - 4s 23ms/step - loss: 0.0770 - accuracy: 0.9889 - val_loss: 0.0985 -
time: 1min 14s (started: 2022-02-13 21:53:03 +00:00)
```

TPU takes 1 min 14 seconds.

6.3: LSTM:

LSTM Results of checking different files (malicious/benign):

Checking Bittorrent.exe

```
------------------------------------------------------------
2022-02-04 07:58:45.728589: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Score: 0.81661 -> Malicous
```

Checking ZoomInstaller.exe:

```
------------------------------------------------------------
2022-02-04 08:00:59.148288: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
Score: 0.02410 -> Benign
```

LSTM CPU, GPU, TPU time taking difference:

For LSTM the Batch Size was Increased for Training the Neural Networks faster.

```
Epoch 00010: val_accuracy improved from 0.98623 to 0.98649, saving model to /content/drive/MyDrive/FYP2/ember2018/
168/168 [==============================] - 118s 703ms/step - loss: 0.0619 - accuracy: 0.9880 - val_loss: 0.0672 -
time: 21min 1s (started: 2022-02-13 21:06:06 +00:00)
```

CPU takes 21 min 1 second.

```
Epoch 00010: val_accuracy did not improve from 0.98612
168/168 [==============================] - 10s 62ms/step - loss: 0.0612 - accuracy: 0.9886 - val_loss: 0.0697 -
time: 2min 27s (started: 2022-02-13 21:29:14 +00:00)
```

GPU takes 2 min 27 seconds.

```
Epoch 00010: val_accuracy did not improve from 0.98564
168/168 [==============================] - 5s 27ms/step - loss: 0.0639 - accuracy: 0.9877 - val_loss: 0.0724 -
time: 1min 25s (started: 2022-02-13 21:42:24 +00:00)
```

TPU takes 1 min 25 seconds.

6.4: Time Comparison:

This is the time comparison between all models and its working environment for the better performance to be measured upon.

Model	CPU	GPU	TPU	Epochs	Batch Size
ANN/MLP	3 min 24 seconds	2 min 39 seconds.	1 min 17 seconds	10	64
CNN-1D	35 min 27 Seconds	4 min 26 Seconds	1 min 14 seconds	10	64
LSTM	21 min 1 Second	2 min 27 seconds	1 min 25 seconds	10	1000

Table 6.4: Time Comparison for CPU, GPU, and TPU

6.5: Results Comparison (Average):

Model	Accuracy	Validation Accuracy	Test Files	Training Epochs	Training Batch Size
ANN/MLP	**99.25%**	**98.82%**	**9/10**	**10**	**64**
CNN-1D	**98.89%**	**98.52%**	**8/10**	**10**	**64**
LSTM	**98.77%**	**98.56%**	**8/10**	**10**	**1000**

Table 6.5: Table of Accuracy Results

Conclusion/Future Work:

Thus, by using Standard Dataset of PE Malwares, we have induced three methodologies to detect malware in an input file. The training phase has been done on three deep learning models CNN-1D, MLP, and LSTM (Variation of RNN).
The Training Accuracy Results are as follows:

1- CNN => 98% Accuracy
2- ANN/MLP => 99% Accuracy
3- LSTM => 98% Accuracy
4- Then, Testing of File inputs takes place as with CNN Saved model. 8/10 results are positive, as with ANN, 9/10 results are positive and with LSTM due to Long times of Training, improved architecture and accurate learning results 8/10 were positive.

Future work will be to improve the Neural Network Architecture on these Three Models to improve more Accuracy to get more Positive Results. It can also include improving these models to be more hybrid and faster with Inception Model, VGG16 and Resnet101. The reference code will be given in my github account: https://github.com/shaxtex98/.

References:

[1] [1710.09435v1] Malware Detection by Eating a Whole EXE (arxiv.org), 25 Oct, 2017
https://arxiv.org/abs/1710.09435v1

[2] M. Sikorski and A. Honig, Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. San Francisco, CA, USA: No Starch Press, 1st ed., 2012.

[3] R. Perdisci, A. Lanzi, and W. Lee, "Classification of packed executables for accurate computer virus detection," Pattern recognition letters, vol. 29, no. 14, pp. 1941–1946, 2008.

[4] S. William, Computer security: Principles and practice. Pearson Education India, 2008.

[5] PE Header Analysis for Malware Detection (sjsu.edu) September, 2018.
https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1624&context=etd_projects

[6] Portable Executable File Infecting Malware Is Increasingly Found in OT Networks, Ken Proska, Corey Hildebrandt, Daniel Kappellmann Zafra, Nathan Brubaker, Oct 27, 2021
https://www.mandiant.com/resources/pe-file-infecting-malware-ot

[7] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," Journal of Computer Virology and Hacking Techniques, vol. 13, pp. 1–12, 2015.

[8] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 421–430, Dec 2007.

[9] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," Journal of Information Security, vol. 05, pp. 56–64, 01 2014.

[10] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," Journal in Computer Virology, vol. 2, no. 1, pp. 67–77, 2006.

[11] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on, pp. 297–300, IEEE, 2010.

[12] [1804.04637] EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models (arxiv.org) April 16, 2018
https://arxiv.org/abs/1804.04637

[13] 2- Malware Detection using Deep Learning Approach – Dataset & Flow Diagrams

https://codexlearner.com/artificial-intelligence-library/deep-learning/2-malware-detection-using-deep-
learning-approach-dataset-flow-diagrams/

[14] Multilayer Perceptron Explained with a Real-Life Example and Python Code: Sentiment Analysis – Sep 21, 2021
https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

[15] Li-Ion Batteries Parameter Estimation with Tiny Neural Networks Embedded on Intelligent IoT Microcontrollers, Giulia Crocioni, Danilo Pau, Jean-Michel Delorme, Giambattista Gruosso, July 2020
https://www.researchgate.net/publication/342686148_Li-Ion_Batteries_Parameter_Estimation_With_Tiny_Neural_Networks_Embedded_on_Intelligent_IoT_Microcontrollers

[16] H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," International Journal of Distributed Sensor Networks, vol. 11, no. 6, p. 479174, 2015.

[17] Z. Aung and W. Zaw, "Permission-based android malware detection," International Journal of Scientific & Technology Research, vol. 2, no. 3, pp. 228–234, 2013.

[18] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "Pe-miner: Mining structural information to detect malicious executables in realtime," in International Workshop on Recent Advances in Intrusion Detection, pp. 121–141, Springer, 2009.

[19] GitHub – tamnguyenvan/malnet: Malware Detection using Convolutional Neural Networks

[20] T.-Y. Wang, C.-H. Wu, and C.-C. Hsieh, "Detecting unknown malicious executables using portable executable headers," in INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on, pp. 278–284, IEEE, 2009.