Malware Prediction Using Deep Neural Network

By

Natasha Kiran

A thesis submitted to the faculty of Information Security Department, Military College of Signals, National University of Sciences and Technology, Islamabad, Pakistan, in partial fulfillment of the requirements for the degree of MS in Information Security.

JUNE 2021

ABSTRACT

This research focuses on Malware Prediction techniques using different machine learning algorithms on Microsoft Telemetry Dataset. The malware industry continues to be a well-organized, well-funded market dedicated to evading traditional security measures. Once a computer is infected by malware, criminals can hurt consumers and enterprises in many ways. This project focuses on developing techniques to predict if a machine will soon be hit with malware, using machine learning. Large- and small-scale IT companies are under constant threat of malware and large sums of money is spent to protect these systems, in addition to regular security audits of the systems, because once the systems are infected by malware, there is a serious threat to the businesses and their customers in many ways including data theft. For these reasons, this project is proposed with the goal of being able to predict any malware activity using machine learning techniques, to thwart the threats before they could harm the systems. We have trained models using three different Machine Learning algorithms to compare the most accurate and robust algorithm for malware prediction. This kind of research is also relevant to our national needs as now a days, more and more reliance is being put on IT systems all over the world including Pakistan, because they are efficient and reliable. These systems range from simple bus ticket management system to National data registration system and border management systems, which hold very critical and sensitive national information. The security and integrity of these systems are a paramount concern for the authorities and if a successful algorithm and system is developed which could effectively analyze and report any threat to these systems before they happen, it will be of immense importance and utility. With our research we will be able to deliver such a system that would be able to predict malicious activities including malware attacks and viruses in IT systems

CERTIFICATE

This is to certify that **NS Natasha Kiran** Student of **MSIS-16** Course Reg.No **00000206298** has completed his MS Thesis title **"Malware Prediction Using Deep Neural Network"** under my supervision. I have reviewed his final thesis copy and I am satisfied with his work.

Thesis Supervisor

(**Assistant Professor Mian M. Waseem Iqbal**)

Dated: ______Jun 2021

DECLARATION

I hereby declare that no portion of work presented in this thesis has been submitted in support of another award or qualification either at this institution or elsewhere.

DEDICATION

This thesis is dedicated to MY PARENTS, FAMILY, AND TEACHERS
for their love, endless support, and encouragement

ACKNOWLEDGEMENTS

I am grateful to ALLAH Almighty who has bestowed me with the strength and the passion to accomplish this thesis and I am thankful to Him for His mercy and benevolence. Without His consent, I could not have indulged myself in this task.

Table of Contents

1 Introduction

The malware industry continues to be a well-organized, well-funded market dedicated to evading traditional security measures. Once a computer is infected by malware, criminals can hurt consumers and enterprises in many ways. Large- and small-scale IT companies are under constant threat of malware and large sums of money is spent to protect these systems, in addition to regular security audits of the systems. Once the systems are infected by malware, there is a serious threat to the businesses and their customers in many ways including data theft.

After a malware attack, as a part of traditional pipeline of mitigation strategy, researchers manually try to dissect the malware files to look for any sign of code which can lead to malicious activity. On the other hand Anti-Virus programs rely on comparing the program code with an existing malware database and look for any similarities which can give away the presence of malicious activity. But code obfuscation is a very big flaw in static analysis, which is being exploited in zero-day attacks, in which attackers modify the code completely to evade static analysis.

A technique called dynamic, or behavior malware analysis involves the execution of malware in a controlled environment to look for any malicious footprints left by malware execution. Behavior malware analysis focuses on getting malware execution data to perform analysis rather than static files/codebase. As good as dynamic malware detection seems, Antivirus companies avoid using this technique for malware detection in the real world because of inherent problems which come with dynamic analysis time penalty. Dynamic analysis relies on post-execution data from malware which requires time for gathering data and then further time for processing information to look for malicious activities. Hence, in the real world, it is very cumbersome to wait for dynamic analysis, during that delayed time system can lose critical information to malware attack.

Many researchers are now more attracted towards the analysis of malware behavior data produced during live execution instead of post-execution. They setup live monitors to analyse traffic generated on the system to look for anomalies in network traffic like the size of packets sent, open ports through which data is being communicated, IP addresses talking to system etc. or systems call being made to the system by different programs, for example program making changes to the registry or accessing memory etc. But such anomaly-based analysis generates more false positives than actual numbers and some studies have suggested that attackers are now exploiting this technique to generate fake bulk traffic to disguise actual malicious activity.

A company named Virus Total that aggregates many antivirus products and online scan engines to check for viruses that the user's own antivirus may have missed, or to verify against any false positives, has reported that nearly one million new malwares have been seen everywhere in the recent years. With such huge numbers of new malware it is very difficult to do static or dynamic analysis on every new malware and getting results in real time, therefore we need to move towards more advanced ways to automatically detect and predict viruses in adequate time, based on different parameters.

Owing to the inherent flaws in static and dynamic analysis discussed above, we propose a novel idea of relying on machine telemetry data to predict chances of malware infection in system or network of systems, by using machine learning and deep neural networks. Our aim is to predict the chances of malware infection before they actually happen.

2 Literature Review

Numerous research work has been done in the field of malware analysis, which can be broadly divided into categories static analysis and dynamic analysis. Static analysis involves inspection of malware files before execution, analyzing code for network calls, API calls or any suspicious behavior. Whereas dynamic analysis, which is more advanced than static analysis, work on malware in execution or post execution looking at behavior or pattern of malware. I took systematic approach and reviewed research on malware prediction using different techniques and datasets.

2.1 Early-stage malware prediction using Recurrent Neural Networks

As good as dynamic malware detection may seem it has its drawback as well, one being that the time which is spent on collecting malware execution data for analyzing the behavior which can be drastic sometimes as virus can still effect system and more work would need to be done to undo the harm caused by malware. Matilda and Pete conducted a study on dynamic study using recurrent neural network where prediction is done during execution rather than waiting post execution. Analysis of Behavior snapshot of malware through deep neural network has improved the detection of malware in 5 seconds with 95% accuracy; a research by Matilda Rhodeb and Pete Burnap (2018) concluded that dynamic analysis of malware during execution is preferable.

2.2 Automatic Analysis of Malware Behavior Using Machine Learning

More and more researchers are moving toward dynamic analysis of malware rather than static because of fact that attackers can change signature of malwares to escape from static analysis. Rieck et al. [1] devises a new mechanism where he dynamically studies behavior of malware by devising their own dataset named Malheur. Dataset was devised by researchers by using behavioral analysis of binaries of malwares provided by different antimalware companies. Each malware was executed in sandbox and its pattern files were generated.

One of the most important factors in dynamic malware analysis is time required to process the malware so that enough data can be collected to be analyzed further for malware presence. And with the enormity of malware and more patterns to be observed it becomes successively difficult to detect malware in a timeframe that malware can be detected at earliest.

This research was conducted in 4 steps. First, Rieck et al. [1] and teams executed all the malwares in controlled environment to generate the system calls and changes malwares would be making to system. Once all the malwares were executed, behavior reports were generated for every malware containing key information like changes made to registry, network calls, data packets transported etc., these reports were then converted to vector space. Once all the malwares were represented in form of vectors different machine learning clustering techniques were applied to classify to which class malwares belong. At last successive analysis was done altering between clustering and classification.

Results show that by processing the behavior of malwares in clusters reduce the processing time and speed from 100 min to 25 min, and memory consumption was reduced from 5GB to 300 MBs. Hence Rieck et al. [1] successively proves that incremental behavior analysis increases efficiency in both time and memory space.

2.3 Classification of Malware System Call Sequences by Deep Learning

With every passing day malware detection is becoming one difficult task as attackers are devising novel ideas and techniques. Hence researchers are focusing on generic patters rather than actual code study. Every research is focused on different behavior of malware like the calls or changes to system.

Kolosnjaji et al. [2] focuses on call sequences made by malware to classify it using deep neural networks. This research utilizes deep learning in a very novel way. Firstly, call sequences made by malwares were analyzed with DNN. Then for classification of actual malwares were studied with RNNs and CNNs. Finally, the neural units were analyzed to predict performance of their work.

In this research LSTM was used in conjunction with CNNs, LSTM was used to sort importance and results show that using LSTM with CNN increased performance from 79% to 89%.

2.4 Malware Detection Using Process Behavior with Deep Neural Network

This research was conducted on malware detection using traffic data. Problem with such research is requirement of domain knowledge is very important. One cannot understand criticality of data if they cannot understand traffic dumps captured. This research was conducted using processed injected by malware after execution. Dataset was generated by executing malwares in Cuckoo Sandbox. After malware execution injected processes were tracked and logged. Tobiyama et al. [3] used RNN and CNN for this research, one very good point from this research is usage of LSTM with RNN, because RNNs are dependent on previous nodes output it faces error vanishing problem, but this problem is addressed by use of LSTM which only focuses on the meaningful information from previous node rather than whole.

For this research malware pattern files were generated by executing malwares in sandbox and collecting log files. Using RNN features were collected from log files, collected features were converting to image file to pass through CNN and lastly models were checked for validity. Results show that proposed model in this survey got 92% accuracy, but the dataset against which models were trained was very small hence accuracy of model could not be vouched for.

2.5 Analysis of Virtual Memory Access Patterns for Malware Detection Using Machine Learning

Malware detection can generally be divided into static and dynamic analysis, in static analysis malware is analyzed for any anomalies in code of malware looking for any existing known or suspicion signatures system or API calls, whereas dynamic analysis rely on the execution of malware so that its behavior can analyzed from its behavior. In this research, Xu et al. suggest a new way reliant relying on neither of previously suggesting techniques and proposing rather a new idea of malware detection with hardware analysis of malware memory access during its execution. But this technique requires strong understanding of binary execution, machine architecture and ways memory is accessed and managed during software execution lifecycle.

Xu et al. carried out analysis in iterations, whole execution of software was divided into different iterations and each iteration was analyzed. The result show that the way from where memory is accessed and how often largely tell a malicious behavior from legitimate software. Summary histogram was created for each iteration, and it was labelled as benign or malicious during software execution. Binary signature would be verified on after training model was trained, finally analyzing hardware memory execution, alarm would be generated on malware detection. Different models were tested for best performance, Near Forest gave best results with 99% accuracy.

2.6 Zero-Day Malware Detection

With rise of technology comes curse of increased attacks, even though Anti-Virus programs are becoming most robust and sophisticated but there always lurks zero-day attacks. Because new malwares keep evading signature-based methods and obfuscating their signatures. This study conducted by Gandotra et al. suggests combining both static and combing techniques to detect malwares, but shortcoming in this technique is enormity of data and time it takes to detect malware. Hence author suggests that classification should be done on selection of relevant data instead of whole. Research was conducted through WEKA classification technique.

Research was started by gathering malwares samples from virus Total database and collecting files from different operating systems. Second step was analysis of malware in sandbox environment through which attribute files were created, from which features were selected to be passed through classification model. But before that top feature were selected using IG method, top 7 features were selected to be passed to classification model. WEKA library was used to build classification model. Different classification models were used but random forest gave best results with 99% accuracy.

We have seen a lot of research being conducted in the field of dynamic malware analysis considering various parameters such as pattern, processes, API calls, network traffic, memory access etc. However, none of the research has been focused on machine telemetry data to look for malicious activity pattern and predict for malwares.

TABLE I. SUMMARY OF AVAILABLE TECHNIQUES

Sr.	Research	Focus	Technique	Summary
1	Early-stage malware prediction using recurrent neural networks	Dynamic Malware Detection	Recurrent Neural Network	Research focused on dynamic malware detection using recurrent neural network (RNNs) where prediction is done during execution rather than waiting post execution. Analysis of Behaviour snapshot of malware through deep neural network has improved the detection of malware in 5 seconds with 95% accuracy.
2	Automatic Analysis of Malware Behaviour Using Machine Learning	Malware Pattern	Clustering and Classification Techniques	Devised their own dataset named Malheur. Successive analysis was done altering between clustering and classification. Results show that by processing the behaviour

				of malwares in clusters reduce the processing time and speed.
3	Deep Learning for Classification of Malware System Call Sequences	Calls made by Malwares	Deep Neural Network	Research focuses on call sequences made by malware to classify it using deep neural networks. This research utilizes deep learning in a very novel way.
4	Malware Detection with Deep Neural Network Using Process Behavior	Process Behavior	RNN and CNN	Research conducted using traffic data generated by running systems. Using RNN features were collected from log files, collected features were converting to image file to pass through CNN and lastly models were checked for validity achieving 92% accuracy.
5	Automatic Malware Classification and New Malware Detection Using Machine Learning	Automatic Malware Detection	Decision Trees and Clustering.	Research done on huge dataset of malwares including Trojan, viruses, worms etc. On extracted features and decision tree techniques were applied to identify the malware family and unknown malwares were detected with clustering technique based on SNN.
6	Malware Detection Using Machine-Learning-Based Analysis of Virtual Memory Access Patterns	Virtual Memory Access Pattern	Near Forest Technique	Research suggests a new way of malware detection with hardware analysis of malware memory access during its execution. Different models were tested for best performance, Near Forest

				gave best results with 99% accuracy.
7	Zero-Day Malware Detection	Zero Day Attack	Random Forest Technique	Research suggests combining both static and combing techniques to detect malwares, author suggests that classification should be done on selection of relevant data instead of whole, random forest gave best results with 99% accuracy.

In all the research methods we have identified we have seen that dynamic methodology is more preferred way of detecting malwares rather than static methodologies, but dynamic malware detection has its limitations as well with time required to collect enough data to detect malwares.

In all the previously conducted research on malware detection main focus was on data from malwares like call sequences Kolosnjaji et al. [2], API calls , process Behavior Tobiyama et al. [3] , Virtual Memory Access Patterns Xu et al. [3] but none of the research was focused on data of machine on which malwares were executed .We are focusing our research and analysis on telemetry data of machines plus the threat reports generated by antivirus software to see for patterns and common features from machines infected by malwares , the analysis will be captured in form of machine learning model. Our focus will be on predicting chances of any machine getting infected by malware with help of our trained model so that sufficient steps can be taken ahead of time to stop the attack.

After going through all the research done previously in the field of malware detection, we decided to approach our training through different angles. First, almost all the aspects of malwares had been covered starting from analyzing actual malware files, logs generated by malwares, API made by malwares to system, memory access pattern of malwares, network traffic of systems running malwares. But we could not find any research conducted on telemetric data. So, we decided to train model through deep neural network and have it compared with different other techniques like LightGbm.

3 Proposed Methodology

In this research we will be using dataset provided by Microsoft Kaggle, specific business needs were considered while formulating the sample Dataset. Main consideration points were the running time of machines and user privacy. Malware Detection is ever growing research which is going forward with same pace of technological advancement , advancement in machine learning and data mining techniques and ever increasing sophistication which we are seeing in the attacks being launched but malware detection is becoming more complex with systems offline and online time , systems which are periodically getting patches for known vulnerabilities , systems which are getting advanced operating system versions and many more factors.

3.1 Data

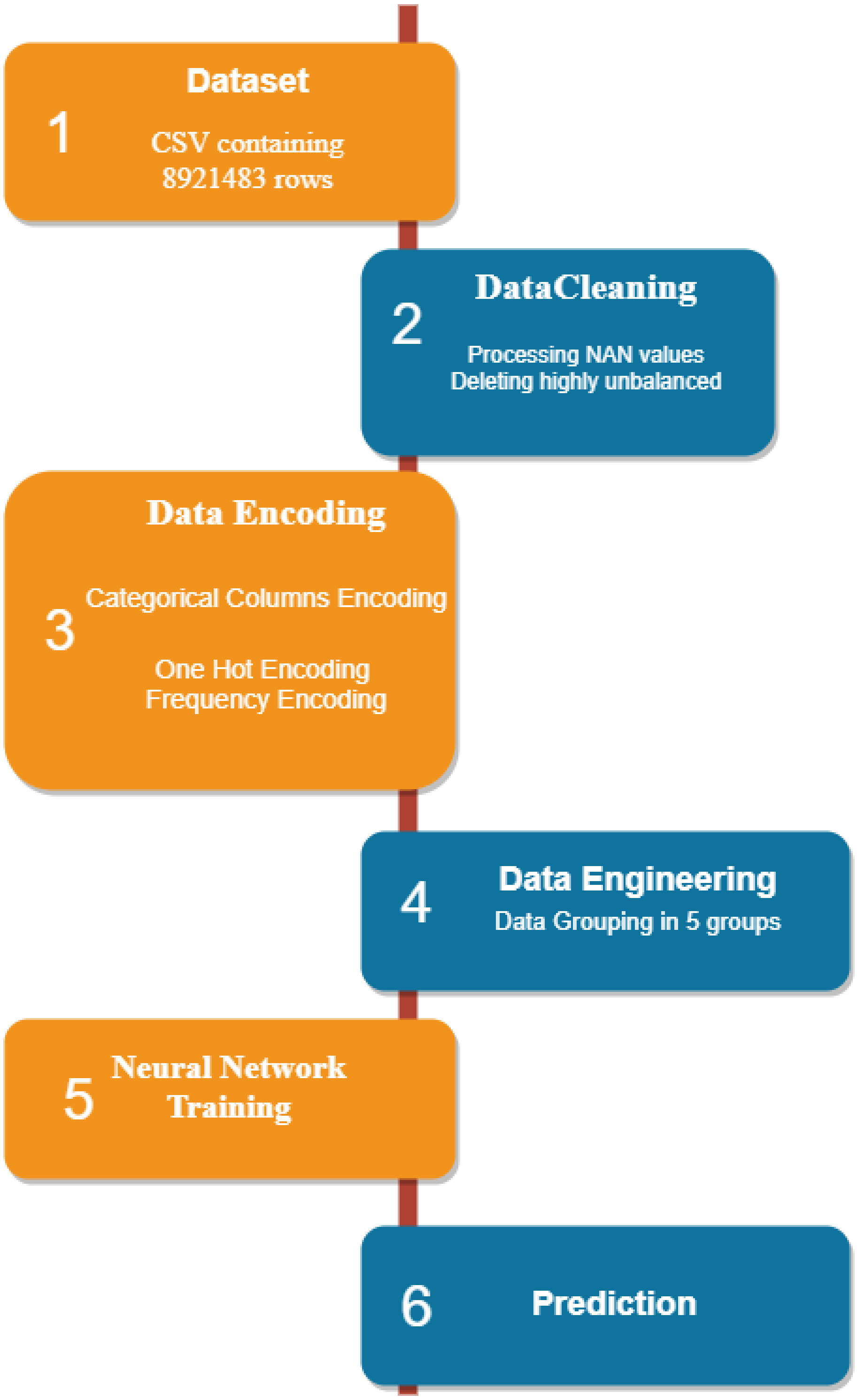
Each row in dataset can be uniquely identified by MachineIdentifier column, HasDetection will tell if malware was detected on machine or not. Methodology used to create dataset was crafted to meet certain constraints like to not to violate user privacy and time during which machines were running. Malware detection is fundamentally a time series problem, but more complications are added to it by certain factors like introduction of new systems, Machine's status either they are in use or not, Machines with security patches, machines with outdated OS versions etc. Dataset has total of 82 feature columns, few of the columns are explained in table 2 to give vague idea of data we are dealing with. We had total 8921483 records available to us for training.

TABLE II. DATASET COLUMNS DESCRIPTION

Column Name	Description
ProductName	Defender state information e.g., win8defender
CountryIdentifier	ID for the country the machine is in
OrganizationIdentifier	ID for the organization the machine belongs in, organization ID is mapped to both specific companies and broad industries
GeoNameIdentifier	ID for the geographic region a machine is in
LocaleEnglishNameIdentifier	English name of Locale ID of the current user
Platform	Calculates platform name (of OS related properties and processor properties)
IsProtected	This is a calculated field derived from the Spynet Report's AV Products field. Returns: a. TRUE if there is at least one active and up-to-date antivirus product running on this machine. b. FALSE if there is no active AV product on this machine, or if the AV is active, but is not receiving the latest updates. c. null if there are no Anti-Virus Products in the report. Returns: Whether a machine is protected.

3.2 **Research Flow Chart**

Our approach to this problem has been depicted in following flow chart, and each step is discussed in detail in the subsequent sections.



3.3 **Exploratory Data Analysis**

To better understand data interpolation, we plotted some graphs which are as follows. Following topics were investigated during this process.

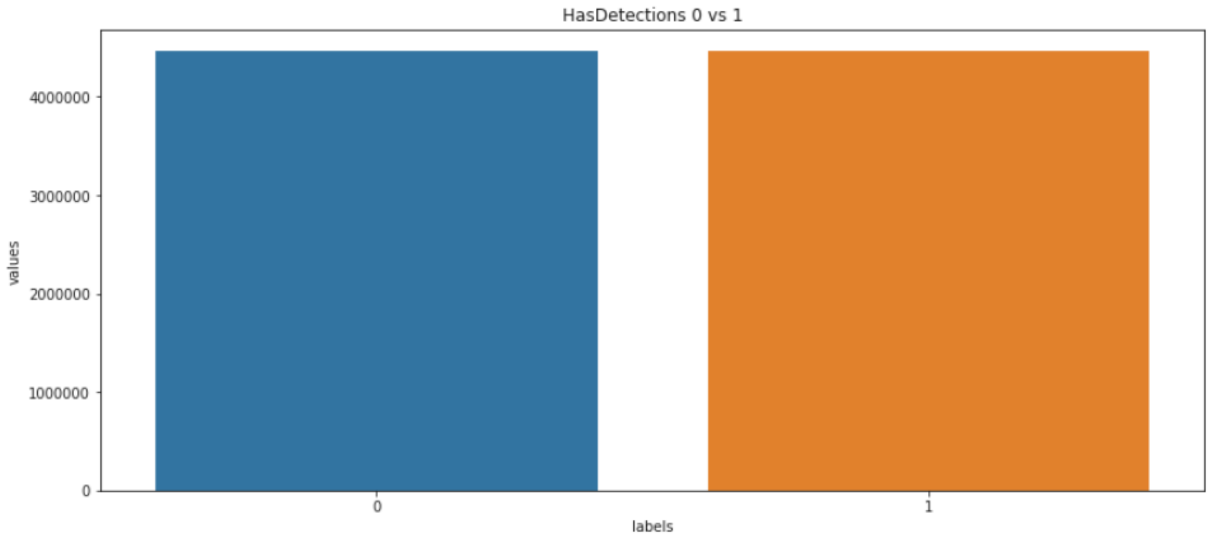
- Distribution of the target feature
- Frequency of the types of binaries, numerical and categorical in data
- Dimensions of each feature
- Frequency of each dimension
- Frequency of NaN-values

In addition, binary and numerical features were studied based on:

- Maximum value.
- Minimum value.
- Mean value.
- Standard deviation.
- Frequency of zero values

To answer basic question that is our data unbalanced with rest tour target column HasDetections. As we can see in Figure.1 that we are not facing any problem of data imbalance.

Fig. 1. Target column data distribution



As we stated earlier that this dataset is provided by Microsoft and it was crafted by collecting data from threat reports, Figure 2 is showing stats of which product generated how much data, we can see that most of the data was produced by wind8Defender.

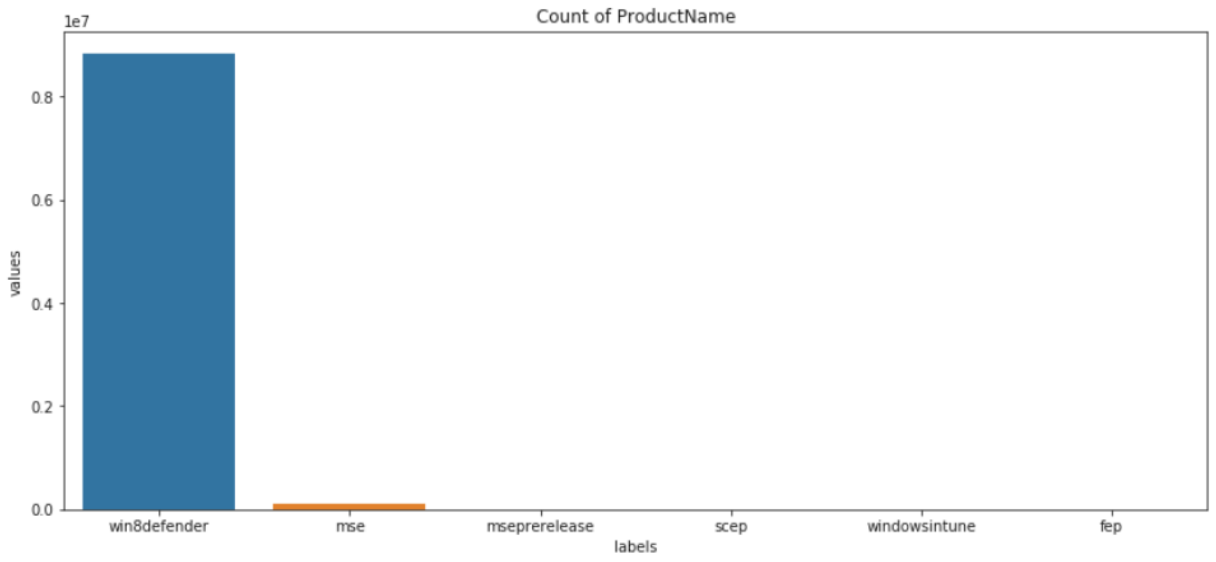


Fig. 2. Data Distribution of Product

Figure 3 shows data distribution for country identifier, in the data country identifier is shown is integer values but these values can easily be mapped to which country each number is representing.

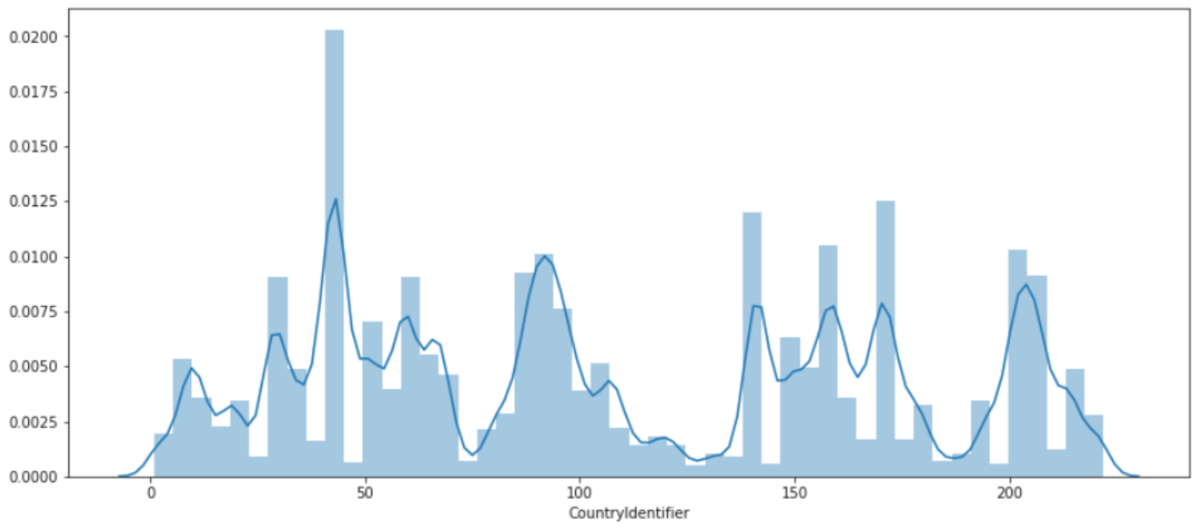


Fig. 3. Data distribution by Country Identifier

And we can see data distribution across different OS in Figure 4, explaining that which version of operating system has seen most virus reports.

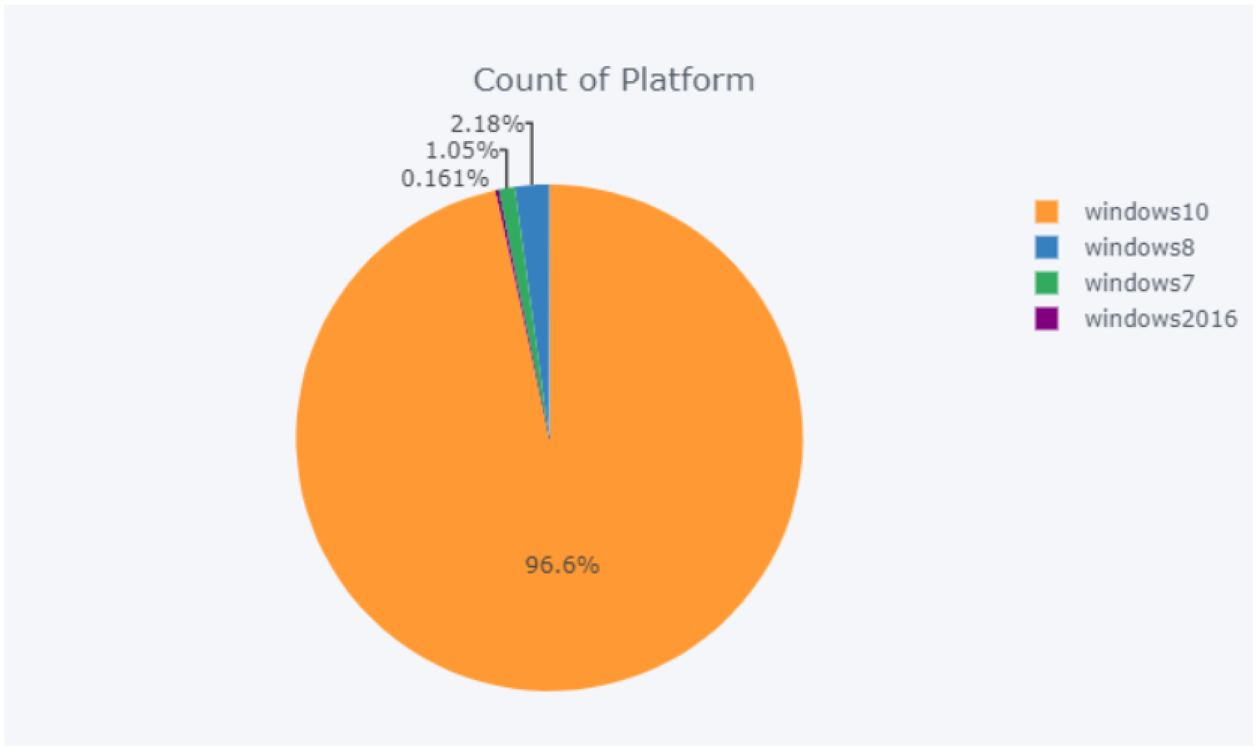


Fig. 4. Data distribution for platform count

And figure 5 is showing distribution of data across different processors, this could be very helpful in seeing that for which family of processor most of the malwares are launched.

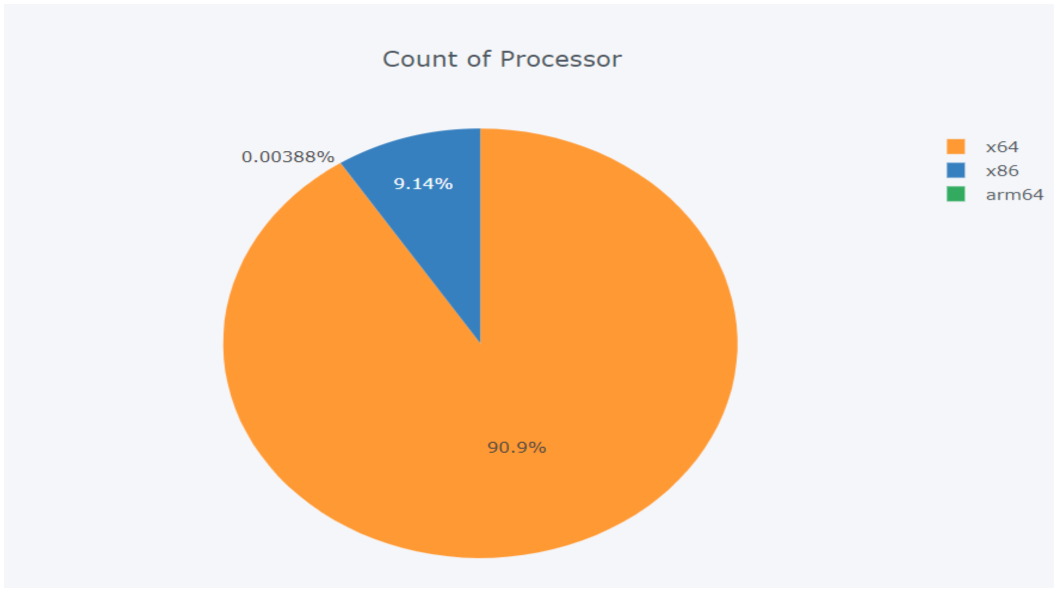


Fig. 5. Data distribution for type of processor

Now let us see how some of the columns has distribution for our target column HasDetection in Figure 6, we can see blue bar as HasDetection 1 and orange bar HasDetection 0, graph is plotted to see data distribution of each type of workstation against target column.

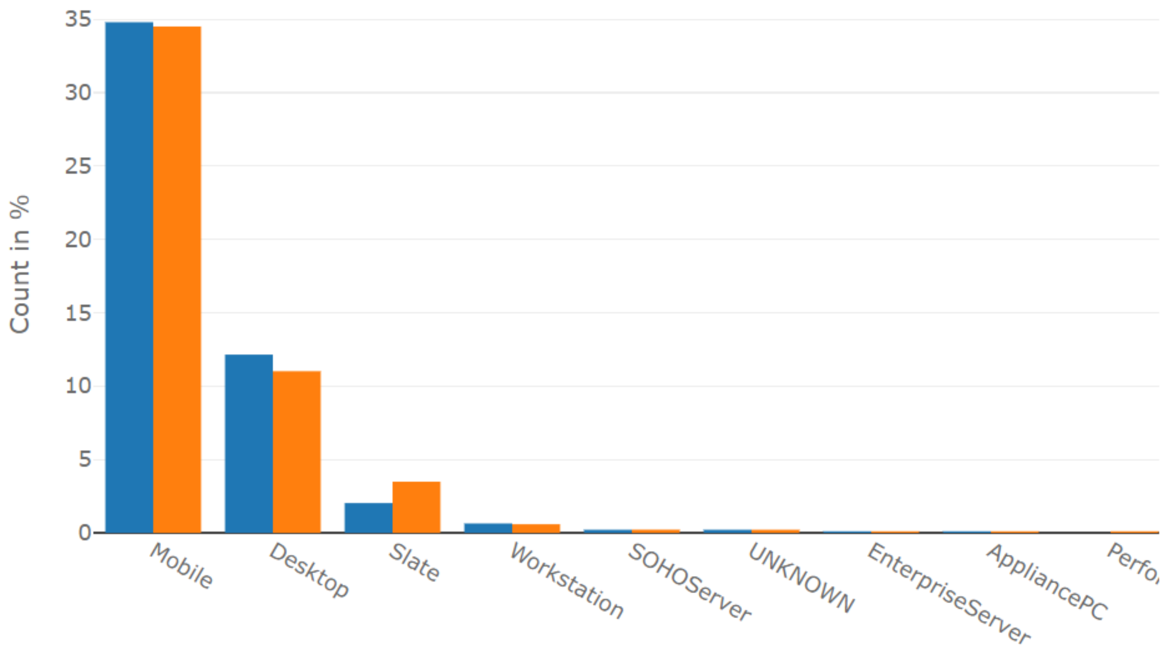


Fig. 6. Data distrubution for type of system.

Data exploration both manual and achieved through automation is very critical as it gives initial understaning of system , as ininitially data is provided as huge amount. Data exploration gives better understading of system then viewing data as unstructured flat files. After getting bettter understanding of provided dataset we started working on data cleaning as explained in next section.

3.4 Data cleaning

The Data Cleaning Process pre-processes the raw data sets and transform those for feature use. The goals of data cleaning step are as follows.

- Features with too many NaN-values are deleted.
- Features with highly unbalanced dimensions are deleted.
- Features are classified as binary, numeric or categoric.

3.4.1.1 NaN Or Missing Values

Occurrence of missing values represent messiness of data, reason of presence of missing or NaN values could be human error, software glitch or any other reason but they would skew our results I left in data. If we try to remove all such rows where we see missing values size of our dataset will drastically reduce one thing, we cannot afford in machine learning process, we cannot even replace it with 0 or 1 as for regression problems it could be misleading. So, for our data cleaning process we replaced numerical NaNs with mean and median for such columns which had any outliers in them. Whereas for categorical values we went with most frequent values.

Before cleaning up the data sets, the underlying features and characteristics were intensively studied with an EDA (Exploratory Data Analysis) Process as explained in previous chapter.

3.4.1.2 Deleting highly unbalanced Features

Training model on unbalance data will produce wrong predictions for instance if we train our model with data where 99% times x happens and only 1% y happens, trained model will predict x in most of times. So, it is important to either get rid of such columns or bias your data to get interesting results. As we can see some examples of unbalanced columns from our dataset in Figure 7 and Figure 8, we removed such columns before our training.

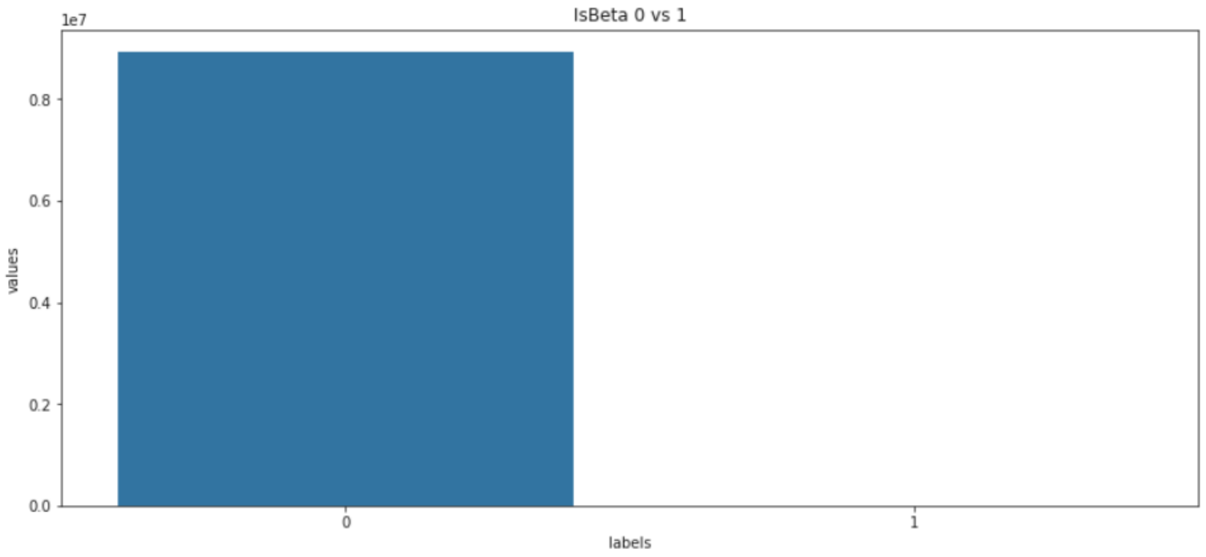


Fig. 7. Unbalanced IsBeta column

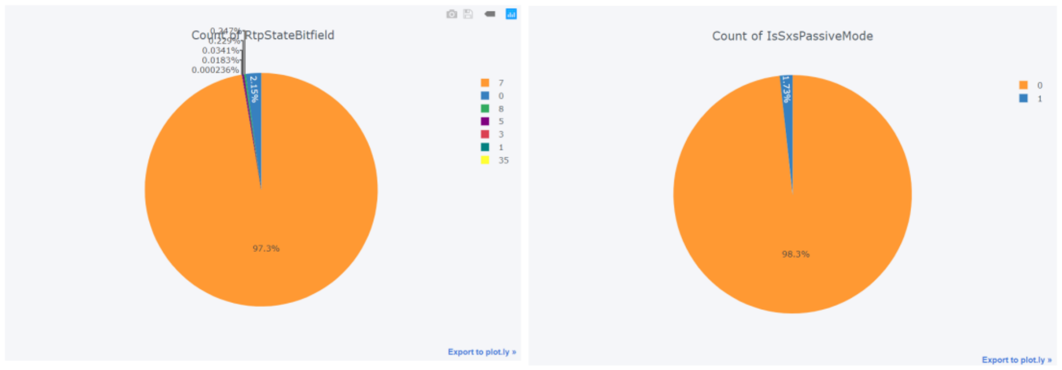


Fig. 8. UnBalanced RtpStateBitField, IsPassiveMode Column

3.5 **Data Encoding**

We have been provided with 82 columns in the dataset with total 8921483 entries. In the dataset we have numeric, categorical and timestamps, we will be using different techniques to encode data. Variables are encoded using following techniques.

- Selective One Hot Encoding
- Numeric Encoding
- Label Encoding
- Frequency Encoding

In total we have 43 categorical values which have total 211,562 entries.

3.5.1 Categorical Columns Encoding

As we know categorical values contain labels rather than numbers, many machine learning algorithms like decision trees can work with categorical data directly but for our research we need to convert all the categorical values to numeric for making our system more robust and efficient, because we do not have any ordinal relation in the categorical values.

3.5.2 One-Hot-Encoding

We will be using One-Hot-Encoding but we cannot possibly use 211,562 entries for encoding hence we will be using specialized selective technique to use only values which will be impacting on HasDetection. We are dividing our category variables in 2 categories Frequency Encoded Parameters FEM and One-Hot-Encoded parameters. Following 4 columns 'Engine version', 'Application Version', Antivirus Signature Version', 'Census OS Version' will get encoded through frequency encoded because we have ordinal values in these columns, they can be easily encoded but we will use Selective One hot encoding for 39 columns which does not have any ordinal relation with each other.

In selective One-Hot-Encoding we will ignore all the values which are deviating 0.5 less than 1 standard deviation for HasDetection. We will test following hypothesis for every value we have.

HP1 = Probability ('Has Malware'= 1 for Malware was Present) = 0.5

HP2 = Probability ('Has Malware'= 1 for Malware was Present) (is not equal) 0.5

For every value from each selected category, we will see how it is impacting on HasDetection rate, we are assuming that roughly 0.5 of the values has positive detection rate, then for every value we see if it deviates from our 0.5 standard deviation. If standard deviation is more or equal to 0.5, we will use a Boolean at its place otherwise we ignore it. As an example, let us take column 'Antivirus Product Installed', Value 1 has detection rate of 54.8% and Value 2 has Detection Rate of 39.6%, using Central limit theorem $z = 2(p(X)-0.5)\sqrt{n}$ their respective z values are

Z value for 1 = $2(0.54-0.5)\sqrt{6208893} = 199$

Z value for 2 = $2(0.39 - 0.5)\sqrt{2459008} = -344$

Hence, we will keep all values for version 1 and ignore value for 2. Similarly, For Column 'Country Identifier' we have total 222 unique identifiers. If we apply same Central Limit Theorem, we reduced 222 values to 115 values eliminating 107 values.

3.5.3 Frequency Encoding

Another way of Encoding data we have adapted in our research is Frequency Encoding, where we use the frequency of category as their label. Frequency Encoding helps machine learning training model to assign weights to labels with respect to target column. To optimize the code and model training we did some memory optimizations on variables assigning them to uint8, unint16 and uint32 depending on the highest values.

3.6 Data Engineering

For our Method 4.1(Deep neural network with embedding layer) after analysing data some new columns were introduced after mixing existing columns for instance Column AppVersion2 is engineered from AppVersion (e.g., 4.9.10586.0), in this column second number is taken for instance if AppVersion is 4.9.10586.0 we will take 9 as AppVersion2. This number indicates whether operating system has been updated and it is running latest version. All the categorical variables with cardinality greater than 10 are introduced as new variables, which increases model accuracy. For rest of our methods no new columns were added. In addition to all the encoding techniques we applied on categorical variables we defined above, new columns are introduced for all the columns with cardinality greater than 10, there new columns are introduced with the technique of Frequency encoding. With this technique 35 new variables are introduced.

Instead of passing data directly to network, all the variables are grouped in different pools depending on their relevance to each other, we have introduced 4 groups. All the variables are categorized in these 4 groups.

TABLE III. Groups with relevant columns

Group	Columns
Geographical Group	'CountryIdentifier','CityIdentifier','OrganizationIdentifier','GeoNameIdentifier', 'LocaleEnglishNameIdentifier','Census_OSInstallLanguageIdentifier','Census_OSUILocaleIdentifier','Wdft_RegionIdentifier'
Software &Virus Group	'DefaultBrowsersIdentifier','AVProductStatesIdentifier','AVProductsInstalled','AVProductsEnabled','IsProtected','SMode','IeVerIdentifier', 'SmartScreen','Firewall','Census_IsSecureBootEnabled', 'Census_IsWIMBootEnabled','Wdft_IsGamer','Census_OSWUAutoUpdateOptionsName','Census_GenuineStateName','AppVersion2'
Hardware Group	'Processor','Census_PowerPlatformRoleName','Census_MDC2FormFactor','Census_DeviceFamily','Census_ProcessorCoreCount','Census_ProcessorClass','Census_HasOpticalDiskDrive','TotalPhysicalRAM','Census_InternalBatteryType','Census_InternalPrimaryDiagonalDisplaySizeInInches''Census_InternalBatteryNumberOfCharges'
Name and Model Group	'Census_ProcessorModelIdentifier','Census_FirmwareManufacturerIdentifier','Census_FirmwareVersionIdentifier','Census_OEMModelIdentifier','Census_ProcessorManufacturerIdentifier'

4 Experimental Setup and Scenarios

After data cleaning and encoding we trained following 3 neural networks to compare their performance:

- Network training using LightGBM
- Network training using PyTorch Framework
- Network trained using TensorFlow Framework

4.1 Network Training Using LightGBM

As a first experiment we decided to run simple LGBM model on our cleaned and encoded data to see its performance. For model training we started with data preprocessing to get data into a state after which model can be trained for best result. For data cleaning we first cleared all the NaN values in data so that it will not skew the results. As we divided whole dataset in 3 groups i.e. categorical data, binary data, and numeric data, we applied different techniques for NaN removal. For categorical data we replaced such values with label Unknown, for numeric values we replaced it with -1 and for binary features we replaced values with most frequent values, and we did some basic label encoding. Afterwards LightGBM algorithm was applied with Bayesian Hyperparameter Optimization training using Cross Validation (3 folded) to return best fitted model, model was trained by passing parameters as explained in Table VII.

TABLE IV. LightGBM classifier parameters

Parameter	Value
boosting _type	Gbdt
learning_rate	0.0106
min_child_samples	295
num_leaves	160
reg_alpha	0.6321152748961743
reg_lambda	0.6313659622714517

subsample_for_bin	80000
Subsample	0.8202307264855064
colsample_bytree	0.6110437067662637
Estimators	12000

4.2 Important Feature Selection using LightGBM

For our second method we decided to take slightly different approach and play with feature and their impact on our target column "HasDetection" and then train important features using neural network.

4.2.1 Light Gradient Boost Machine to Find Important Features

LGBM is one of the best methods which produces a baseline model using subsample and it also highlights important features in dataset, features which are contributing most towards target column. First, we split our dataset in train and test for validation before passing it to LGBM. Afterwards we decided our parameters to train our model as shown in Table V.

TABLE I. LGBM FEATURE SELECTION PARAMETERS

Parameter	Value
boosting _type	gbdt
learning_rate	0.0106
min_child_samples	295

num_leaves	160
reg_alpha	0.6321152748961743
reg_lambda	0.6313659622714517
subsample_for_bin	80000
subsample	0.8202307264855064
colsample_bytree	0.6110437067662637
Estimators	5000

Defined parameters were passed in LGBM model with objective set as binary and we got Feature importance as displayed in Table VI, we are only mentioning top 10 columns here.

TABLE II. Top 10 Features

	Feature	Importance	Normalized	Cumulative
0	DisplaySizeInInches	57048	0.071758	0.071758
1	monitor_dims	44264	0.055678	0.127436
2	AVProductsInstalled	39925	0.050220	0.177657
3	AVProductIdentifier	35066	0.044108	0.221765
4	CountryIdentifier	34878	0.043872	0.265636
5	AvSigVersion	29084	0.036584	0.302220
6	OSInstallTypeName	28141	0.035397	0.337618

7	RegionIdentifier	24468	0.030777	0.368395
8	AppVersion	22664	0.028508	0.396903
9	LocaleEnglishNameId	21664	0.027250	0.424153

After analyzing the feature importance, we trained our model with different selection of column as explained in next section.

4.3 Deep Neural Network Training using PyTorch

After feature selection using Light GBM we then trained our neural network using pyTorch framework. The network architecture of our model is given below

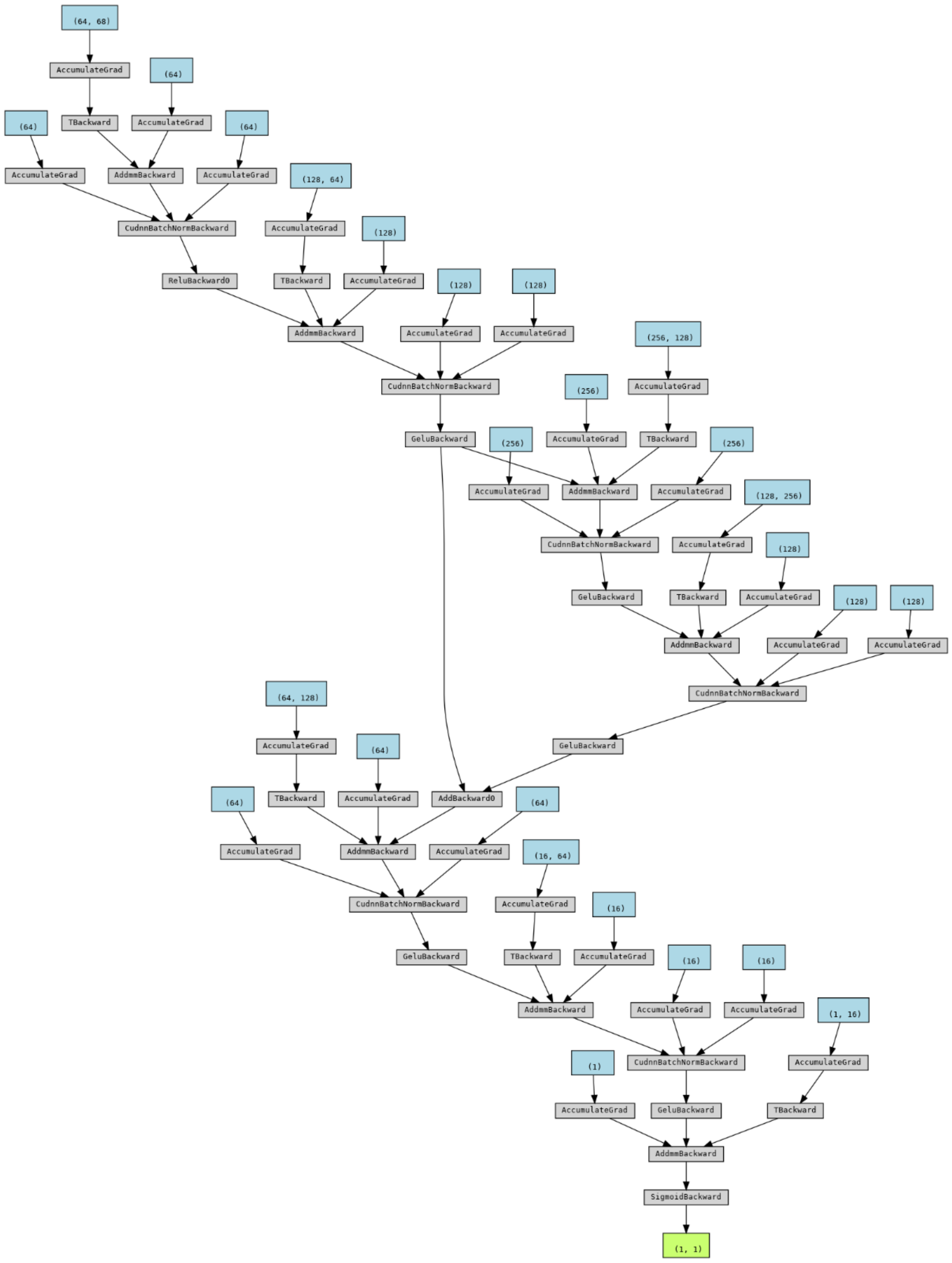


Fig. 9. Model Network Architecture

4.3.1 Block Summary

TABLE III. Summary of all the layers in network

Layer (type)	Output Shape	Param #
Linear-1	[-1, 0, 64]	4,416
BatchNorm1d-2	[-1, 0, 64]	128
ReLU-3	[-1, 0, 64]	0
Linear-4	[-1, 0, 128]	8,320
Dropout-5	[-1, 0, 128]	0
BatchNorm1d-6	[-1, 0, 128]	256
GELU-7	[-1, 0, 128]	0
Linear-8	[-1, 0, 256]	33,024
Dropout-9	[-1, 0, 256]	0
BatchNorm1d-10	[-1, 0, 256]	512
GELU-11	[-1, 0, 256]	0
Linear-12	[-1, 0, 128]	32,896
Dropout-13	[-1, 0, 128]	0
BatchNorm1d-14	[-1, 0, 128]	256
GELU-15	[-1, 0, 128]	0
Linear-16	[-1, 0, 64]	8,256
Dropout-17	[-1, 0, 64]	0
BatchNorm1d-18	[-1, 0, 64]	128
GELU-19	[-1, 0, 64]	0
Linear-20	[-1, 0, 16]	1,040
Dropout-21	[-1, 0, 16]	0
BatchNorm1d-22	[-1, 0, 16]	32
GELU-23	[-1, 0, 16]	0
Linear-24	[-1, 0, 1]	17
Sigmoid-25	[-1, 0, 1]	0
Total params	89,281	
Trainable params	89,281	
Non-trainable params	0	

Brief explanation of our network layers is as follows.

4.3.2 Batch Normalization

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.

This slows down the training by requiring lower learning rates and careful parameter initialization and makes it notoriously hard to train models with saturating nonlinearities. We address the problem by normalizing layer inputs by making normalization a part of the model architecture and performing the normalization for each training mini-batch. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ\gammaγ and β\betaβ are learnable parameter vectors of size C (where C is the input size). By default, the elements of γ\gammaγ are set to 1 and the elements of β\betaβ are set to 0. The standard-deviation is calculated via the biased estimator, equivalent to torch.var(input, unbiased=False).

Also, by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default momentum of 0.1.

4.3.3 **RELU**

Applies the rectified linear unit function element-wise:

$$\mathrm{ReLU}(x) = (x)^+ = \max(0, x)$$

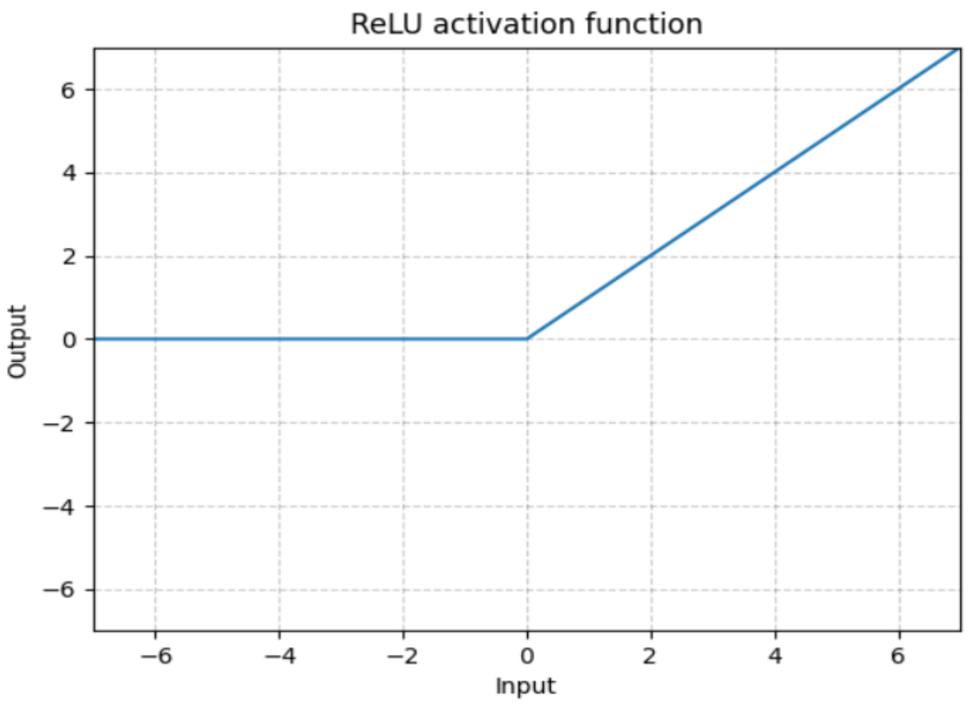


Fig. 10. ReLU Layer

4.3.4 **Dropout**

During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.

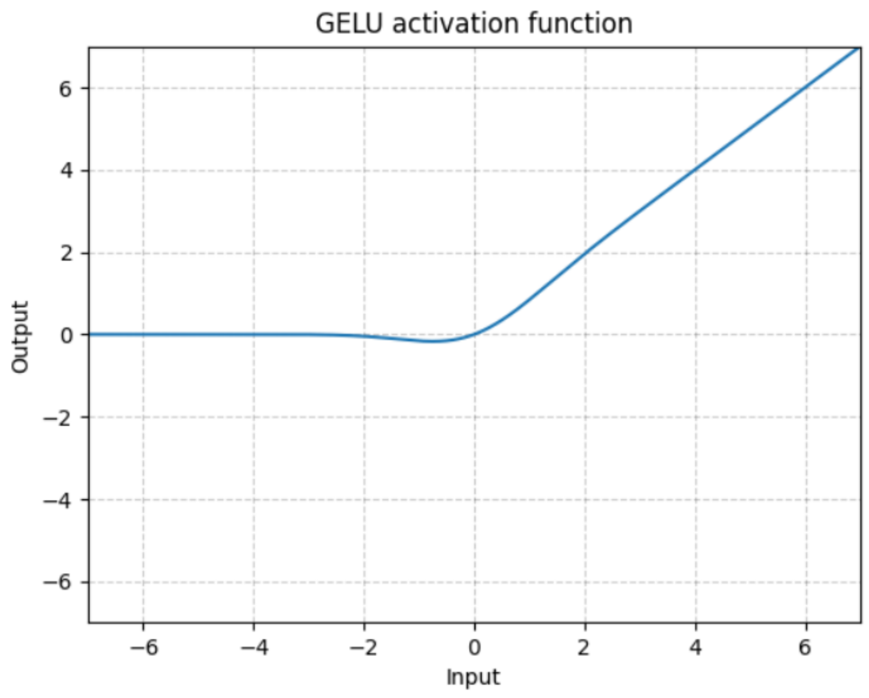
Furthermore, the outputs are scaled by a factor of $\frac{1}{1-p}$ during training. This means that during evaluation the module simply computes an identity function.

4.3.5 **GELU**

Applies the Gaussian Error Linear Units function:

$$\mathrm{GELU}(x) = x * \Phi(x)$$

where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.



4.3.6 **Sigmoid**

Applies the element-wise function:

$$\mathrm{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$

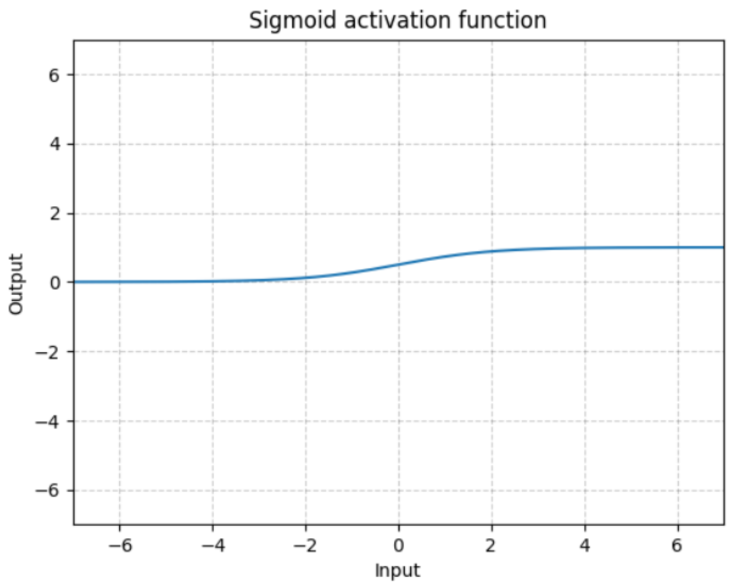


Fig. 11. Sigmoid Activation Layer

4.4 Deep Neural Network with Embedding Layer

For our final experiment we used TensorFlow to train our deep neural network, we passed all the groups, as explained in Data engineering section, through embedding layer before sending it to Neural network.

4.4.1 Embedding Layer

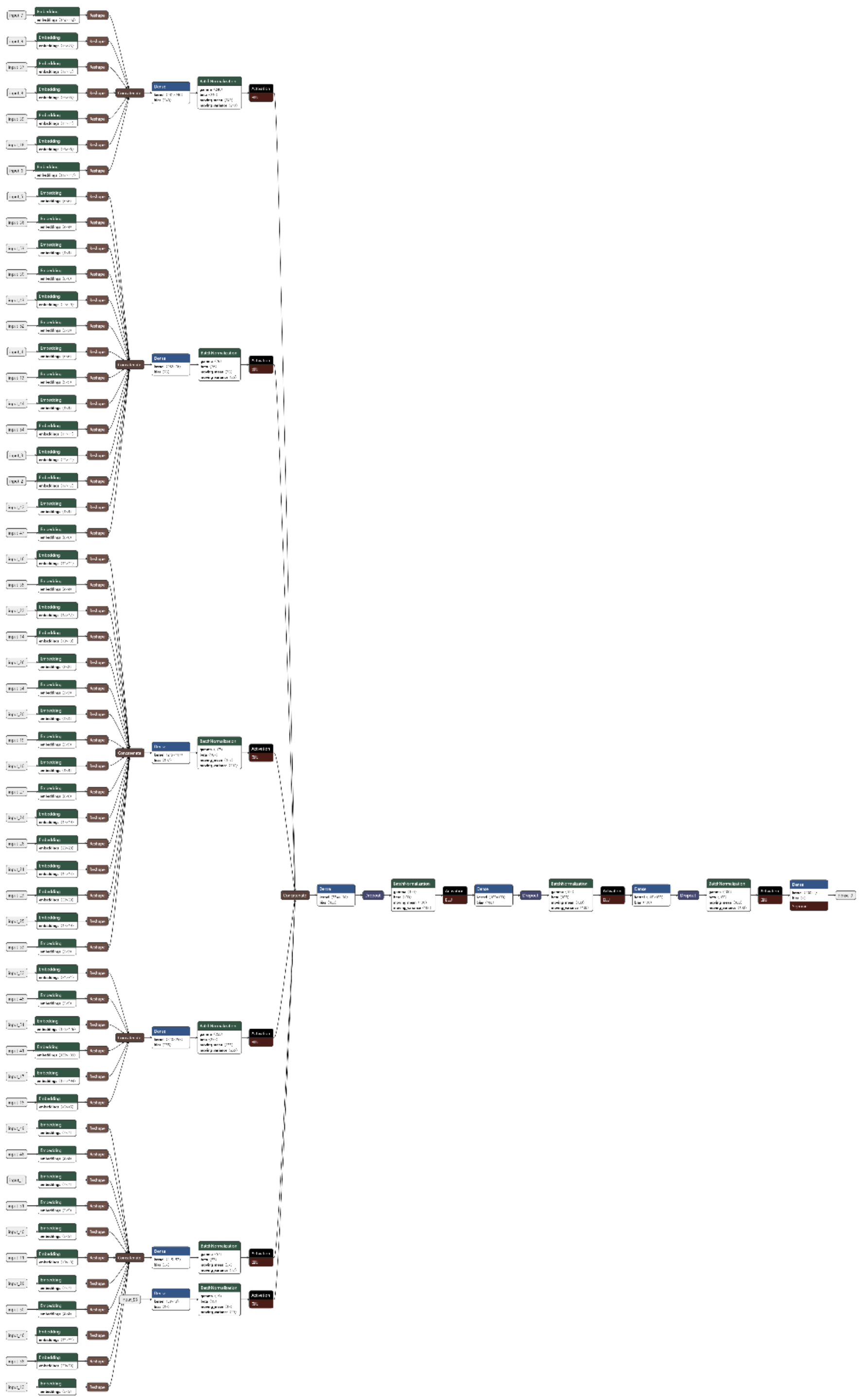


Fig. 12. Embedding Layer concatenating into Group

All the variables, statistically encoded labels, as discussed previously are fed to embedding layer with configuration of 1:1, in 1:1 ratio one input to node will generate one output. Output from first embedded layer is grouped together based on our grouping criteria. These grouped variables are then feed into common dense layer with input output ration of 2:1. Introduction of this dense layer will squeeze variables and it will represent variables in reduced dimensional space. As our dense layer is only using identity activation, our dense layer is behaving like PCA. We are passing following number of variables in every group.

TABLE V. Groups with variable counts

Group Name	Variables count
Geographical	8
Hardware	18
Name-Model	6
Software	15
Misc.	12
Timestamp	33

Embedding layer mimics one hot encoding for category variables. For instance, if we have got 200 unique values and we pass them from one hot encoding, we will get 100 Boolean values back. In embedding layer, it would be like sending 100 units to 100 output 100:100. Let us take another example from our implementation, we are first passing 3 inputs to embedding layer and getting output of 3 making it 3:3 embedding layer, in second instance we are seeing embedding layer of 6:3, we are sending input of 6 to embedding layer and getting back output of 3 making it like PCA.

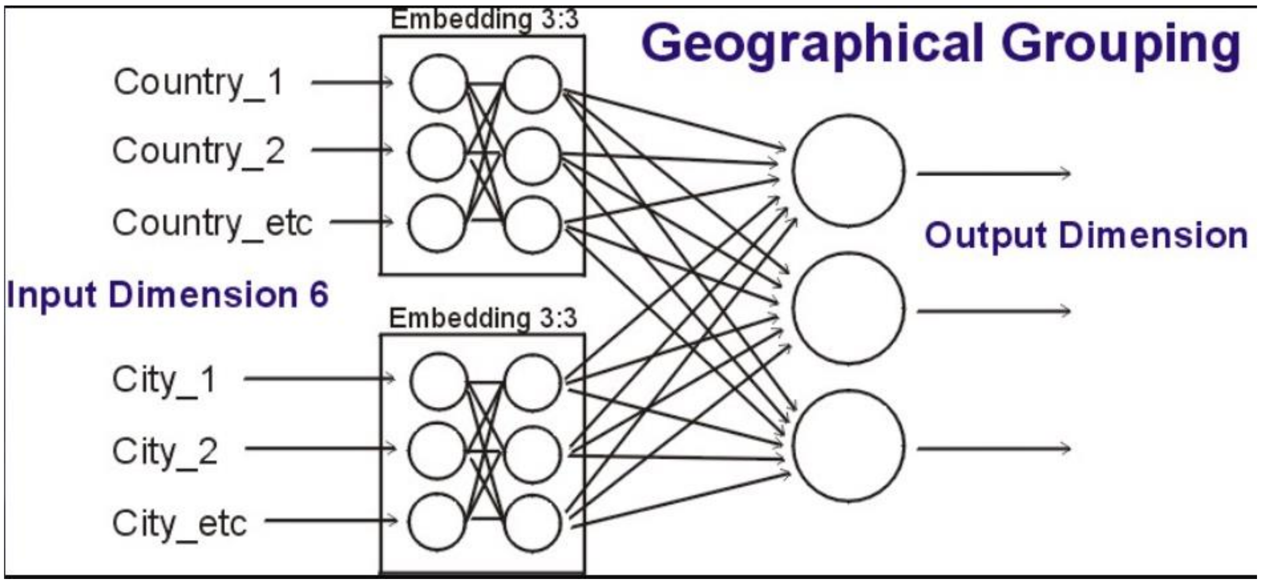


Fig. 13. Embedding layer with 6:3 ratio.

4.4.2 **Network Training**

The training network architecture diagram is as follows:

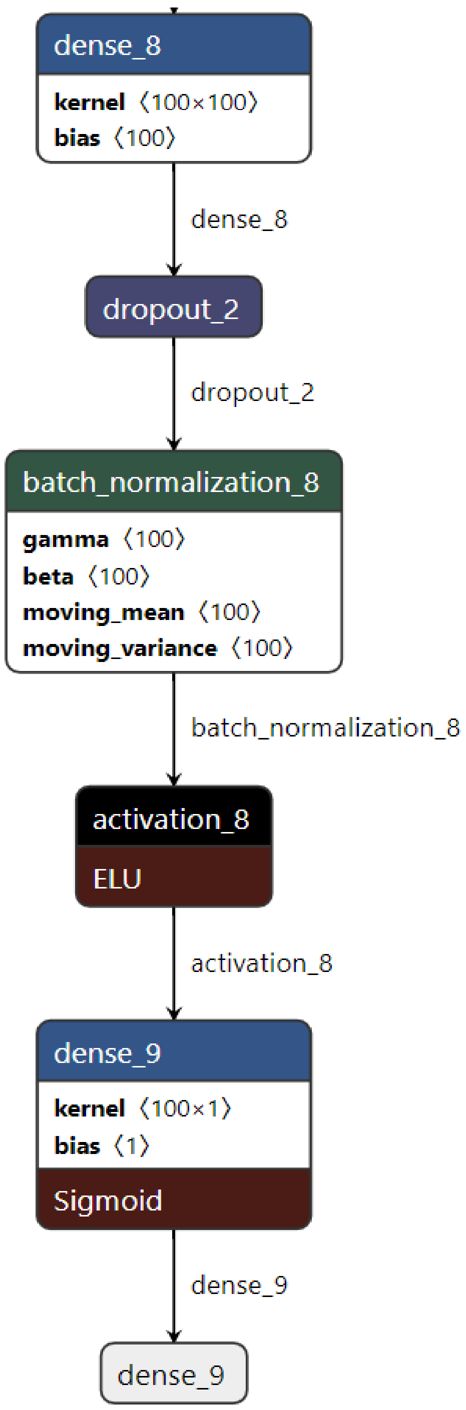


Fig. 14. Network Diagram of trained model

TABLE VI. BLOCK SUMMARY OF LAYERS IN NETWORK

Layer (type)	Configuration
Dense	Kernel [756x100] Bias 100
Dropout	
Batch Normalization	Gamma 100 Beta 100 Moving mean 100 Moving Variance 100
Activation ELU	
Dense	Kernel [100x100] Bias 100
Dropout	
Batch Normalization	Gamma 100 Beta 100 Moving mean 100 Moving Variance 100
Activation ELU	
Dense	Kernel [100x100] Bias 100
Dropout	
Batch Normalization	Gamma 100 Beta 100 Moving mean 100 Moving Variance 100
Activation ELU	
Dense	Kernel [100x1] Bias 100
Sigmoid	

Block summary of all the layers in network are explained in Table V, we used these layers with mention parameters, each layer is explained below in detail.

4.4.2.1 **Dense Layer**

As we explained earlier that we divided all our features into different groups and passed all the groups through Embedding layer which gave us representation of data with more concise and related features, output from all the groups was fed into first dense layer of network.

4.4.2.2 Dropout

We passed output of Dense layer to Dropout Layer with rate 0.2.

4.4.2.3 Batch Normalization

After Dropout we introduced layer of batch normalization with configurations explained in Table V

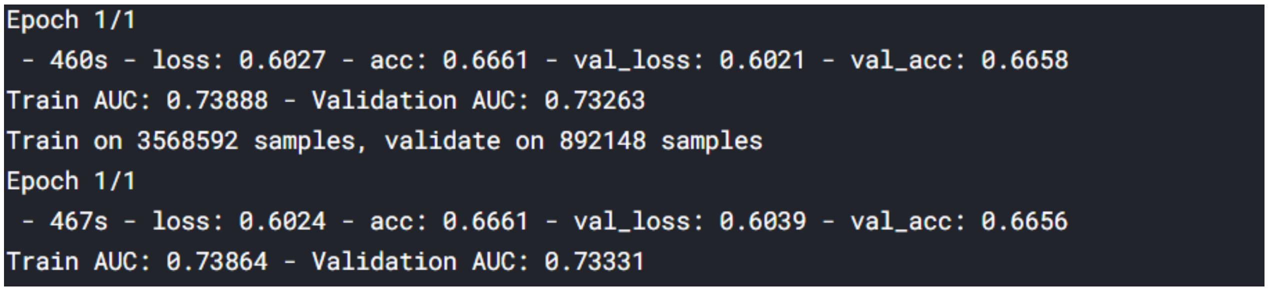
4.4.2.4 Activation ELU

Batch Normalization output was fed into ELU layer.

4.4.2.5 Sigmoid

At the end Sigmoid layer was introduced.

We trained our network on system with 32gb ram and 8gb GPU. Result of last training epoch are shared below. We achieved accuracy of 73%



```
Epoch 1/1
 - 460s - loss: 0.6027 - acc: 0.6661 - val_loss: 0.6021 - val_acc: 0.6658
Train AUC: 0.73888 - Validation AUC: 0.73263
Train on 3568592 samples, validate on 892148 samples
Epoch 1/1
 - 467s - loss: 0.6024 - acc: 0.6661 - val_loss: 0.6039 - val_acc: 0.6656
Train AUC: 0.73864 - Validation AUC: 0.73331
```

Fig. 15. Training Result of last Epochs.

5 Result Analysis

5.1 Experiment Number 1

The block diagram of our first experiment shows the end to end process involved in this method. As explained earlier, we started with cleaning the data for any NaN values, then we trained our model using Bayesian Hyperparameter Optimization as a training criteria with LightDBM.

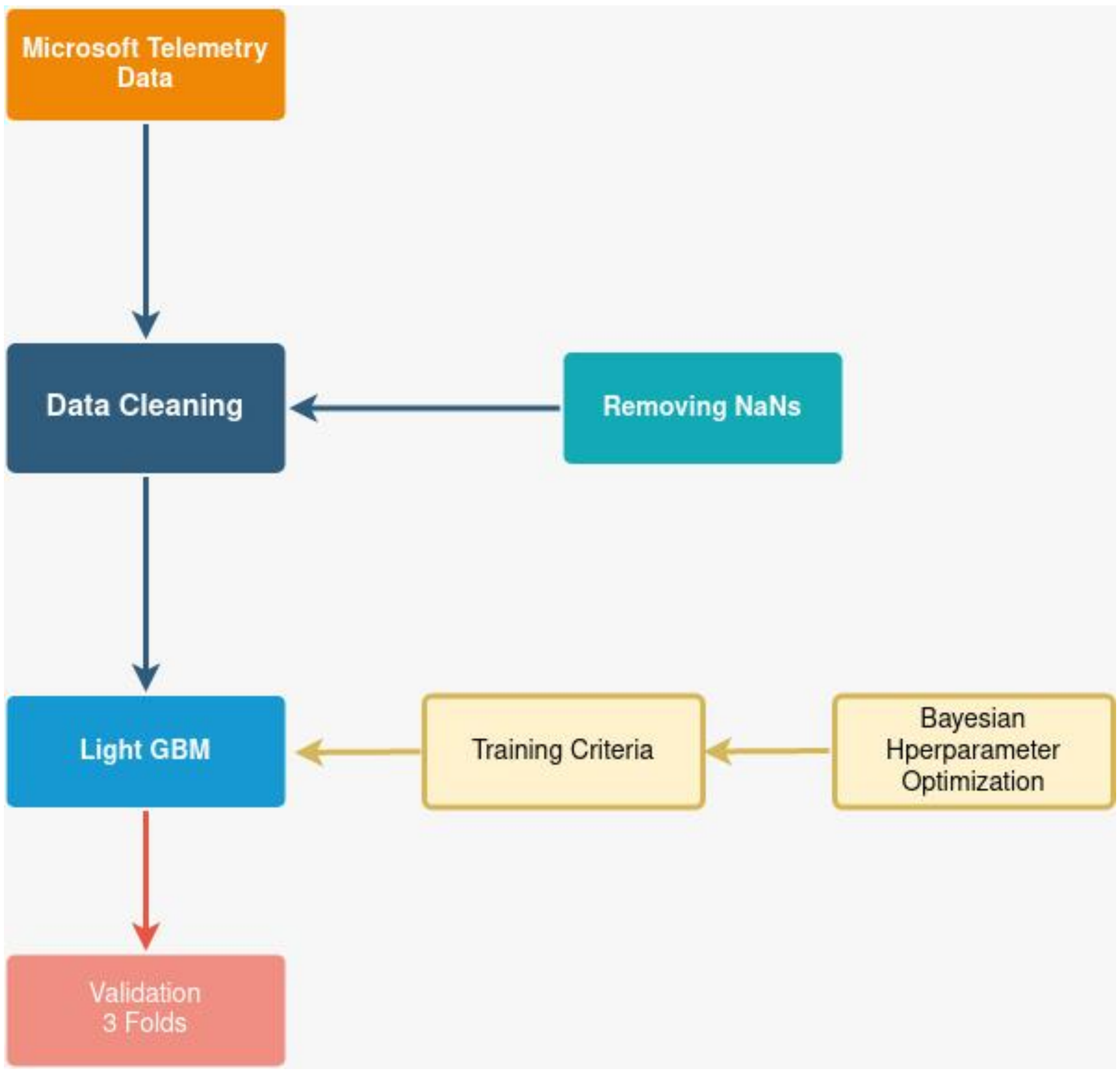


Fig. 16 Block Diagram Experiment 1

5.2 Experiment Number 2

For the second experiment we started with already cleaned data from experiment 1, and used LightGBM for selecting important features of the data rather than training the model. Afterwards we fed the data to deep learning training algorithm using PyTorch framework. The type of neural network training in which we already have target labels and tune our training algorithm to minimize loss in predicting the correct label is called supervised learning. The block diagram of the end to end process is shown below.

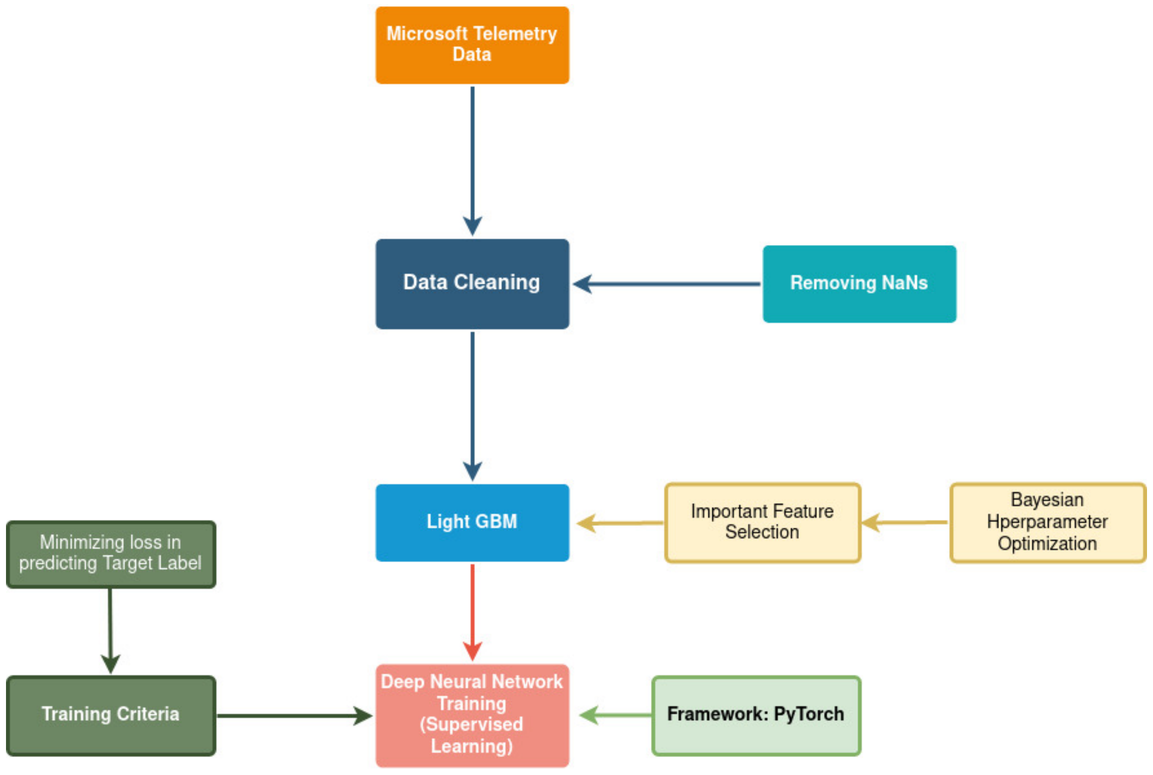

Microsoft Telemetry Data
Data Cleaning
Removing NaNs
Light GBM
Important Feature Selection
Bayesian Hperparameter Optimization
Minimizing loss in predicting Target Label
Training Criteria
Deep Neural Network Training (Supervised Learning)
Framework: PyTorch

Fig. 17 Block Diagram Experiment 2

5.3 Experiment Number 3

For our final experiment we again started with cleaned data from experiment 1, after which we manually did data engineering and grouping based on likeness of data wihh eachother. Finally we trained the deep neural network model by using Tensorflow framework. Once again as we knew the result column so supervised learning process was focused on minimizing loss in predicting the right result over several epocs. The block diagram showing end to end process is shown below.

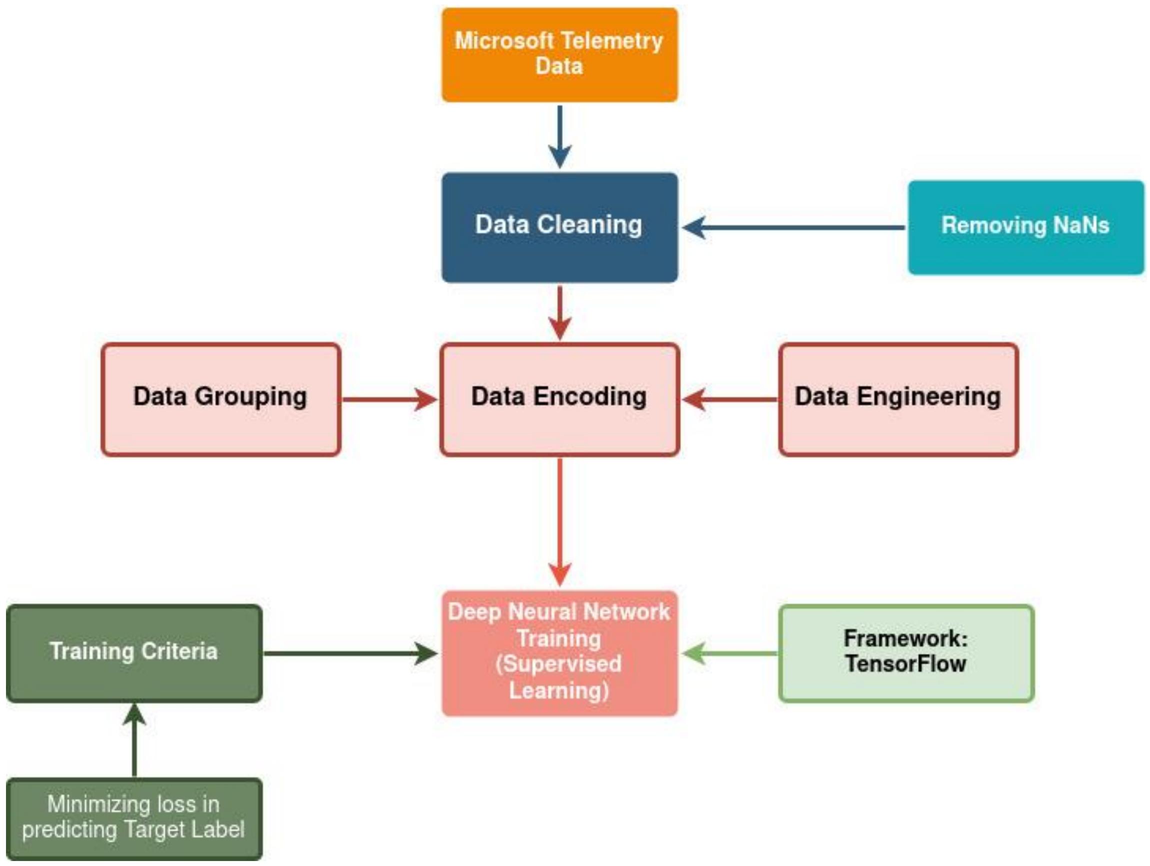


Fig. 18 Block Diagram Experiment 3

5.4 Comparison

The comparison of training using 3 different methodologies is presented in the following table. We achieved best performance through network trained using LGBM encoding and PyTorch framework.

TABLE VII. COMPARISON RESULTS

Experiment	LightGBM	Deep Neural Network with LGBM encoding	Deep Neural Network with Embedding Layer
Framework	LGBM	PyTorch	Tensorflow
Data Preprocessing	• Data Cleaning ○ Removing NaNs	• Data Cleaning ○ Removing NaNs • Data Encoding ○ Using Light GBM	• Data Cleaning ○ Removing NaNs • Data Encoding • Data Grouping
Training Criteria	Bayesian Hyperparameter Optimization	Minimizing Loss in predicting the Target Label	Minimizing Loss in predicting the Target Label
Accuracy	66%	74%	73%
Comments	Since we only did data cleaning and then trained on data, for this reason this model has less accuracy	For this experiment we did data encoding before training the network, and by tuning different parameters like number of layers and loss criteria (Adam) we achieved maximum accuracy	For our final experiment we took a step further in data engineering and grouped similar data columns before training using TensorFlow framework and achieved slightly less accuracy than our second experiment

6 Future Work and Discussion

For improving the accuracy of neural networks further experiments can be performed by tuning different training parameters like learning rate, loss function, depth of network and number of training epochs etc. but these experiments need a considerable amount of resources (GPU, RAM and CPU). As an example, our PyTorch network took 4 days to train using 12GB Nvidia 2080Ti GPU, 16 GB Ram and 4 CPUs.

In order to utilize this research in practical scenarios, we are also working on a software utility to predict chances of Malware threats in an organization or network setup by selecting different inputs from the application and predicting results live by doing inference from the trained network.

7 References

[1] Matilda Rhode, Pete Burnap, Kevin Jones,Early-stage malware prediction using recurrent neural networks,Computers & Security,Volume 77,2018,Pages 578-594,ISSN 0167-4048.

[2] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," Journal of Computer Security, vol. 19, no. 4, pp. 639–668, 2011. 2.

[3] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in Australasian Joint Conference on Artificial Intelligence. Springer, 2016, pp. 137–149.

[4] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual, vol. 2. IEEE, 2016, pp. 577–582.

[5] Xu, S. Ray, P. Subramanyan, and S. Malik, "Malware detection using machine learning based analysis of virtual memory access patterns," in Proceedings of the Conference on Design, Automation & Test in Europe. European Design and Automation Association, 2017, pp. 169– 174.

[6] Sanjay Sharma, C. Rama Krishna and Sanjay K. Sahay. "Detection of AdvancedMalware by Machine Learning Techniques". Proceedings of SoCTA 2017.

[7] Kateryna Chumachenko. "Machine Learning Methods for Malware Detection and Classification". 2017.

[8] E. Gandotra, D. Bansal, and S. Sofat, "Zero-day malware detection," in Embedded Computing and System Design (ISED), 2016 Sixth International Symposium on. IEEE, 2016, pp. 171–175.

8 Appendices

8.1 LightGBM with Deep Neural network Training Code

```
# pytorch mlp for binary classification
from numpy import vstack
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torch import Tensor
from torch.nn import Linear
from torch.nn import ReLU
from torch.nn import GELU
from torch.nn import Dropout
from torch.nn import Sigmoid
from torch.nn import Module
from torch.optim import SGD, Adam
from torch.nn import BCELoss
from torch.nn import BatchNorm1d
from torch.nn.init import kaiming_uniform_
from torch.nn.init import xavier_uniform_
from tqdm import tqdm
import numpy as np
# dataset definition
import dask.dataframe as dd
import pandas as pd
```

```
print('Training neural network, NK Net')
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        print('Reading CSV data')
        # load the csv file as a dataframe
        #df = read_csv(path, header=None)
        df = pd.read_csv(path) #.head(n=10000)
        # store the inputs and outputs
        #self.X = df.values[1:, 1:-1]
        #     AVProductsInstalled,     AVProductsEnabled,     IsProtected,     Firewall,
IeVerIdentifier

        select_range = list(range(1,70))
        select_range.remove(55)

        self.X = df.values[:, select_range]

        self.y = df.values[:, -2]
        # ensure input data is floats
        self.X = self.X.astype('float32')
        self.y = self.y.astype('float32')
        # label encode target and ensure the values are floats
        self.y = LabelEncoder().fit_transform(self.y)
        #self.y = self.y.astype('float32')
        self.y = self.y.reshape((len(self.y), 1))

    # number of rows in the dataset
    def __len__(self):
        return len(self.X)
    # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # get indexes for train and test rows
    def get_splits(self, n_test=0.33):
        # determine sizes
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calculate the split
        return random_split(self, [train_size, test_size])

# model definition


class MLP(Module):
    # define model elements
    def __init__(self, n_inputs):
        super(MLP, self).__init__()

        # input to first hidden layer
        self.hidden1 = Linear(n_inputs, 64)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu')
        self.bn1 = BatchNorm1d(64)
        self.act1 = ReLU()
        # second hidden layer
        self.hidden2 = Linear(64, 128)
        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
        self.do2 = Dropout(0.2)
        self.bn2 = BatchNorm1d(128)
        self.act2 = GELU()
```

```
         # third hidden layer
        self.hidden3 = Linear(128, 256)
        kaiming_uniform_(self.hidden3.weight, nonlinearity='relu')
        self.do3 = Dropout(0.2)
        self.bn3 = BatchNorm1d(256)
        self.act3 = GELU()
          # fourth hidden layer
        self.hidden4 = Linear(256, 128)
        kaiming_uniform_(self.hidden4.weight, nonlinearity='relu')
        self.do4 = Dropout(0.2)
        self.bn4 = BatchNorm1d(128)
        self.act4 = GELU()
          # fifth hidden layer
        self.hidden5 = Linear(128, 64)
        kaiming_uniform_(self.hidden5.weight, nonlinearity='relu')
        self.do5 = Dropout(0.2)
        self.bn5 = BatchNorm1d(64)
        self.act5 = GELU()
        # sixth hidden layer
        self.hidden6 = Linear(64, 16)
        kaiming_uniform_(self.hidden6.weight, nonlinearity='relu')
        self.do6 = Dropout(0.2)
        self.bn6 = BatchNorm1d(16)
        self.act6 = GELU()
        # seventh hidden layer and output
        self.hidden7 = Linear(16, 1)
        xavier_uniform_(self.hidden7.weight)
        self.act7 = Sigmoid()

    # forward propagate input
    def forward(self, X):
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
X = X.to(device)



# input to first hidden layer
X = self.hidden1(X)
X = self.bn1(X)
X = self.act1(X)

# second hidden layer
X = self.hidden2(X)
X = self.do2(X)
X = self.bn2(X)
X = self.act2(X)
identity = X

# third hidden layer
X = self.hidden3(X)
X = self.do3(X)
X = self.bn3(X)
X = self.act3(X)

# fourth hidden layer
X = self.hidden4(X)
X = self.do4(X)
X = self.bn4(X)
X = self.act4(X)

X += identity

# fifth hidden layer
```

```
        X = self.hidden5(X)
        X = self.do5(X)
        X = self.bn5(X)
        X = self.act5(X)

        # sixth hidden layer
        X = self.hidden6(X)
        X = self.do6(X)
        X = self.bn6(X)
        X = self.act6(X)

        # seventh hidden layer and output
        X = self.hidden7(X)
        X = self.act7(X)



        return X

# prepare the dataset



def prepare_data(path):
    # load the dataset
    dataset = CSVDataset(path)
    # calculate split
    train, test = dataset.get_splits()

    # prepare data loaders
    train_dl = DataLoader(train, batch_size=32, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl
```

```
# train the model


def train_model(train_dl, model):
    # define the optimization
    num_epochs = 200
    criterion = BCELoss()
    #optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    optimizer = Adam(model.parameters(), lr=0.05)
    # enumerate epochs
    for epoch in tqdm(range(num_epochs)):
        # enumerate mini batches
        for i, (inputs, targets) in enumerate(train_dl):
            device = torch.device(
                "cuda:0" if torch.cuda.is_available() else "cpu")
            inputs = inputs.to(device)
            targets = targets.to(device)
            # clear the gradients
            optimizer.zero_grad()
            # compute the model output
            yhat = model(inputs.type(torch.FloatTensor))
            # calculate loss
            loss = criterion(yhat.to(device), targets.type(torch.FloatTensor).to(device))
            # credit assignment
            loss.backward()
            # update model weights
            optimizer.step()
        print("Epoch {}/{}, Loss: {:.3f}".format(epoch+1,num_epochs, loss.item()))

# evaluate the model
```

```
def evaluate_model(test_dl, model):
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        inputs = inputs.to(device)
        yhat = model(inputs)
        # retrieve numpy array
        yhat = yhat.cpu().detach().numpy()
        actual = targets.numpy()
        actual = actual.reshape((len(actual), 1))
        # round to class values
        yhat = yhat.round()
        # store
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate accuracy
    acc = accuracy_score(actuals, predictions)
    return acc

# make a class prediction for one row of data



def predict(row, model):
    # convert row to data
    row = Tensor([row])
    # make prediction
    yhat = model(row)
    # retrieve numpy array
    yhat = yhat.cpu().detach().numpy()
    return yhat


# prepare the data
path                                                                                    =
'/vol/research/facer2vm_dev/people/junaid/MalwareThesisLightGBM/data/train_encod
ed_selective_full.csv'
train_dl, test_dl = prepare_data(path)
print(len(train_dl.dataset), len(test_dl.dataset))
# define the network
print('Defining model')
model = MLP(68)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)

# train_dl.to(device)
# test_dl.to(device)
# train the model
print('Starting training')
train_model(train_dl, model)
# save model
PATH                                                                                    =
'/vol/research/facer2vm_dev/people/junaid/MalwareThesisLightGBM/nk_net.pth'
torch.save(model.state_dict(), PATH)
print('Fininshed training, model saved at {}'.format(PATH))
# evaluate the model
acc = evaluate_model(test_dl, model)
print('Accuracy: %.3f' % acc)
# make a single prediction (expect class=1)
#     AVProductsInstalled,     AVProductsEnabled,     IsProtected,     Firewall,
IeVerIdentifier
```

```
row = np.random.uniform(low=0.0, high=13.3, size=(68,))
model.eval()
yhat = predict(row, model)
print('Predicted: %.3f (class=%d)' % (yhat, yhat.round()))
```