Deep Learning Model Analysis for Malware Prediction

Jalpa Deepak Patel
School of Computer Science
University of Windsor
Windsor, ON, Canada
patel2fu@uwindsor.ca

Sumisha Surendran
School of Computer Science
University of Windsor
Windsor, ON, Canada
suren111@uwindsor.ca

Abstract—In the current era of the internet, malware presents a severe and growing threat to security, making the detection of malware of extreme concern. Several machine learning algo-rithms are used for the automatic classification of malware in recent times. This task needs to be addressed by predicting if a computer will be corrupted with malware to avoid data loss. Deep learning is being used these days with improved performance and promising experimental results. Deep learning models are exhibited to work much better in the analysis of long sequences of system calls.

In this paper, we study how a deep learning architecture using the LSTM model can be designed for intelligent malware detection. A wide-ranging experimental study on an extensive sample collec-tion from “Microsoft Malware Prediction Dataset” is performed to compare various malware prediction methodologies. This paper aims in determining the chances of malware affecting a system by analyzing various Deep Learning models based on AUC efficiency metric.

I. Introduction

Malware is software that is destructive/compromised in nature, which affects the data or files. It includes files effected with viruses, trojans, ransomware and spyware. This malware is designed and developed by cyberattackers to cause damage, and it can even allow them to gain access to a network. There are diverse methods in which malware can enter systems - it can be as simple as from URL or some files in email. Whenever the source containing malicious code is clicked or opened, it executes and starts affecting the system.

As malware is malicious software, it can not only found on desktop but also mobiles. Malware on mobile devices can provide access to the camera, microphone or GPS. Malware enters mobile if the user downloads an unpublished and unverified application. Moreover, it can even enter from a simple test message or through emails. The cyber attacks keep expanding, which poses a severe security threat to the users and financial institutions [4]. Fig. 1 depicts the rise in malware recorded by AV-Test Institute. The same institute claimed that it records 350,000 malicious programs and potentially unwanted applications.

Not all types of malware are distinct; instead, the majority of the malware falls into the already existing family of malware [4] [8]. Fig. 2 shows that only a fraction of malware is new, and the remaining malware belongs to the family of existing mal-ware showing similar properties. It would follow comparable patterns of infecting computing gadgets like desktop, phones, tablets or Chromebooks. Hence, we work on developing a model that can foretell if a device would get affected by the virus or not.

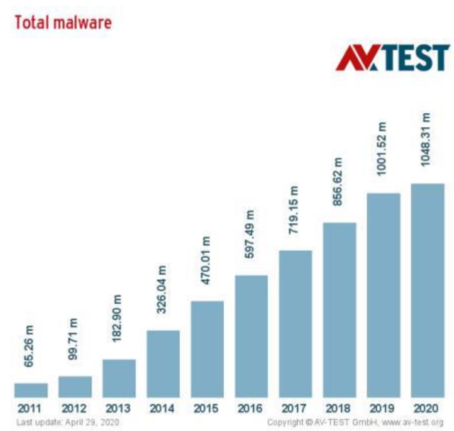

Fig. 1. Total malware statistics over past 10 years as shown in https://www.av-test.org/en/statistics/malware/

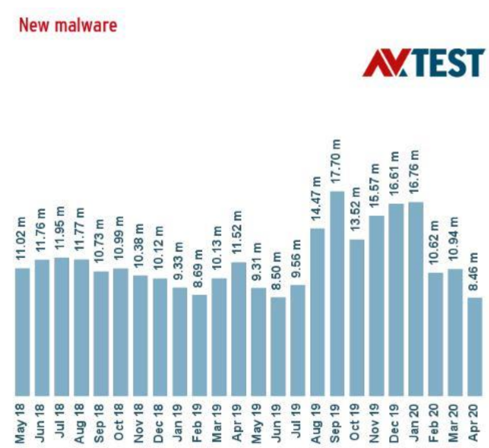

Fig. 2. Statistics of new malware over span of 2 years as shown in https://www.av-test.org/en/statistics/malware/

The task of predicting if the machine will be affected by the virus serves to be a crucial task as it can help users and organizations from data loss. Apart from data loss, we can also take timely action in order to protect our systems or networks from the data breach.

In this project, we compare the performance of our Deep Learning state-of-art models. In our models, we have embeddings for the input and send it to our models for training and prediction. We perform our testing of models on the "Microsoft Malware Prediction Dataset" [7]. We visualize the data and then generate the model from the gained insights. We even consider working on the Adversarial validation tech-nique, commonly used with Machine Learning Algorithms. We perform a comparative study of the two models and aim to progress our work on avoiding over-fitting of the models.

II. Literature Survey

In the connected world of computers, malicious code has turned out to be a ubiquitous and serious threat. Malicious code can penetrate hosts utilizing a variety of methods such as attacks against known software flaws, hidden functionality in regular programs, and social engineering. Given the damaging impact malicious code has on our cyber infrastructure, classifying malicious programs is a crucial objective. Discovering the occurrence of malicious code on a given host is a vital part of any security system. Detecting malicious software executables is made tricky by the continuous modifications created by miscreants in order to evade discovery by antivirus software. Interestingly, the first malware classifier proposed in the literature employed a neural network. Since this initial work, researchers have explored many different machine learn-ing models. Employing system calls as features have been used for malware classification and intrusion detection systems. In "Malware classification with recurrent networks" [9]Pascanu et al. takes a different approach to first learn a language model for the malware and benign files to construct the feature representation for each file in the training set. The authors propose either using a standard recurrent neural network (RNN) or an echo state network (ESN) as the language model. The structure of an ESN is similar to that of an RNN, but the weights are randomly initialized and are not trained. A logistic regression or multi-layer perceptron (MLP) classifier is then trained based on the feature representations output by the language model. In addition, Pascanu et al. propose using temporal max pooling for both recurrent language models to combine a long sequence of temporal features which helps improve the results. The best performing architecture is an ESN with temporal max pooling for generating the feature representation combined with a logistic regression classifier. Unfortunately, the authors found that the RNN failed to learn salient features of the files and has lower performance compared to the untrained ESN. The main aim of this work is to achieve a neural architecture that generates enhanced performance. We use recurrent neural network architectures that can better capture long-term dependencies than the standard RNN. In this study, we revisit the malware language model architecture to investigate whether these enhanced language models lead to improved results for malware classification. We demonstrate that the best performing system in this study uses an LSTM for the language model with temporal max pooling and a logistic regression classifier. We propose deep learning architecture for categorizing malware involving LSTM- and GRU-based language models and a character-level CNN. We show that the features learned from an LSTM language model help improve the performance compared to random-weight architectures, with the LSTM model and temporal max pooling outperforming other competing models.

III. Problem Statement

There are has been ongoing research conducted on classification of malware by using numerous deep learning techniques [5] [10]. These studies provide us with excellent models for classifying malware, and they achieve high precision for the same task. Even though researchers are working on the malware classification, there still awaits essential criteria to have been met, which is to predict how soon a system is going to be infected with malware. Classifying the malware is a relevant module, and the next steps ideally seem to detect the malware and to block it. Also, this task of predicting can further help in classifying the malware so we can block it with a relevant ant-virus mechanism. Microsoft was working on the same idea, and hence they had released anonymized data for a competition for building predictive models. The critical task addressed in this project is to determine the rate at which a system is affected by malicious software. Discussing this problem enables users to take preemptive measures to block the malware. As cybersecurity is an open challenge that we address more often, having developed a model that can intelligently predict and protect device security is a challenge in the digital era.

IV. Methodology

A. Long short-term memory

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. It can not only process single data points (such as images), but also entire sequences of data (such as speech or video). LSTMs help maintain the error that can be backpropagated through time and layers. By retaining a more constant error, they allow recurrent nets to continue to learn over many time steps usually over 1000, thereby opening a channel to link causes and effects remotely. This is one of the central challenges to machine learning and AI, since algorithms are frequently confronted by environments where reward signals are sparse and delayed, such as life itself.

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. LSTMs were developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs. Relative insensitivity to gap length is an advantage of LSTM over RNNs, hidden Markov models and other sequence learning methods in numerous applications. [11]

Bidirectional Long Short-Term Memory : Single direction LSTMs suffer a weakness of not utilizing the contextual information from the future tokens. Bidirectional LSTM utilizes both the previous and future context by processing the sequence on two directions, and generate two independent sequences of LSTM output vectors. One processes the input sequence in the forward direction, while the other processes the input in the reverse direction. The output at each time step is the concatenation of the two output vectors from both directions, ie. ht = →ht k ←ht . The basic idea of bidirectional neural network is to present each training sequence forwards and backwards to two separate recurrent nets, both of which are connected to the same output layer. This means that for every point in each sequence, the BNN has complete, sequential information about all points before and after it. Also, because the network is free to use as much or as little of this context as necessary, there is no need to find a (task-dependent) time-window or target delay size. We use a bidirectional LSTM neural network to obtain a sentence-level context representation. Let lLS be an LSTM reading the words of a given sentence from left to right, and let rLS be a reverse one reading the words from right to left. Given a sentence w1:n, our 'shallow' bidirectional LSTM context representation for the target wi is defined as the following vector concatenation: biLS(w1:n, i) = lLS(l1:i1) rLS(rn:i+1) where l/r represent distinct left-to-right/right-toleft word embeddings of the sentence words.2 This definition is a bit different than standard bidirectional LSTM, as we do not feed the LSTMs with the target word itself (i.e. the word in position i). Next, we apply the following non-linear function on the concatenation of the left and right context representations:

MLP(x) = L2(ReLU(L1(x)))

where MLP stands for Multi Layer Perceptron, ReLU is the Rectified Linear Unit activation function, and

Li(x) = Wix + bi is a fully connected linear operation.Let c = (w1, ..., wi1, , wi+1, ..., wn) be the sentential context of the word in position i. We define context2vec's representation of c as: c = MLP(biLS(w1:n, i)). Next, we denote the embedding of a target word t as t. We use the same embedding dimensionality for target and sentential context representations. To learn target word and context representations, we use the word2vec negative sampling objective function

S = X t,c log (t · c) +X k i=1 log (ti · c) (1) where the summation goes over each word token t in the training corpus and its corresponding (single) sentential context c, and is the sigmoid function. t1, ..., tk are the negative samples, independently sampled from a smoothed version of the target words unigram distribution: p(t) (t) , such that 0 ¡ 1 is a smoothing factor, which increases the probability of rare words. [6]

V. Architecture

The proposed architecture can be explained in different steps like data gathering and cleaning, encoding the data, generating embeddings and model generation for training. The architecture is as shown in Fig. 3.

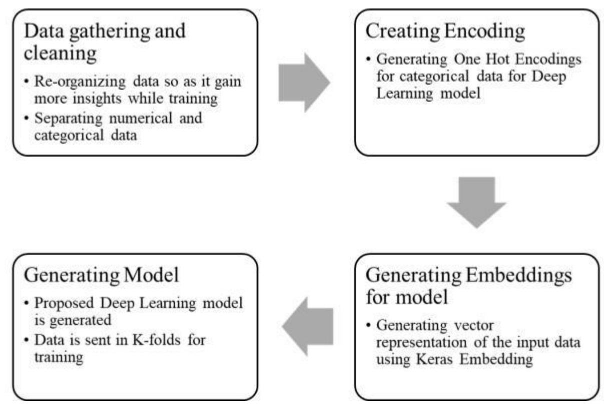

Fig. 3. Proposed Architecture

A. Data gathering and cleaning

The input data is real-time anonymized data that needs some preprocessing before it is used for further training. The data then re-organized into meaningful columns that thus are ready for training. The data consists of a lot of columns, which are version numbers, and it consists of special characters. We usually remove the special characters as a part of preprocessing. However, since this serves as a meaningful purpose, we need to rearrange it in a more understanding way.

Also, the data has different data types; we need to find some unity that makes our task for training easier.

B. Encoding Data

After the data categorization in numerical and categorical encodings, categorical data is One Hot Encoded. The numer-ical data entries are frequency encoded.We prefer One hot encoding because the famous Label encoding considers the higher categorical value as a superior input value. Further, one hot encoding is used for binarization and to not compute the mean and generate encodes.

C. Generating Embeddings

We later create vector embeddings and serve it as input for our models—the embeddings generated using Kears Library.

The input embedding layer is initialized by random weights and generate embeddings for entire input data. This embedding layer is used for learning by the Deep Learning model.

D. Generate Model

We create three different models to compare the study. The deep learning models generated to determine the Area of the Curve(AUC) as the output.Aarchitecture of our proposed models is as follows:

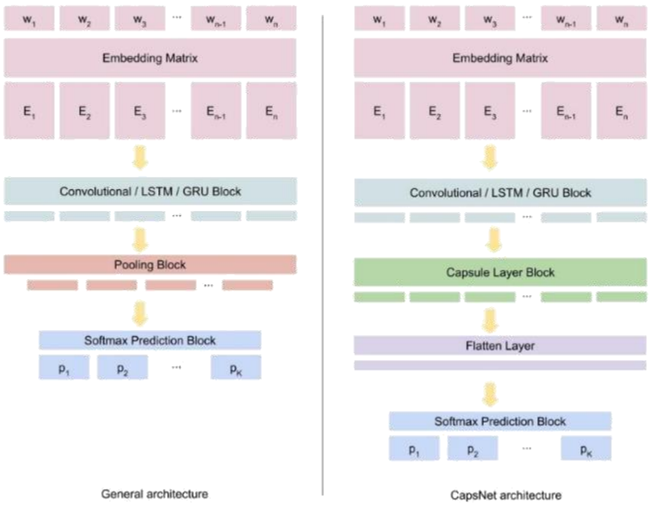

Fig. 4. Deep Learning Model with and without Keras Capsule layer

1) Model 1: This is a 3-layered Neural network with var-ious hyperparameters at different input layers like a dropout. We have two dropout layers after the first two neural network layers so that it can avoid overfitting. We set 40% Dropout for the two layers.

To normalize the activations of the previous layer at each batch, we use Batch Normalisation. We use the ReLU activa-tion function for the first two layers and a sigmoid activation function for the last output layer.

We use Adam for optimization and Decaying the learning rate. The cross-entropy method is used to determine the loss. We save the best model while iterating several times.

2) Model 2: For the second model, we propose a 4-layered Neural network. We design the input embedding layer, and as we know that the data set size is enormous, we add dropout to the embedding layer [3]. Later, to flatten the input, we add a Flatten Keras layer. The output of the learned embeddings is sent to the Deep Learning model. We add Dropout and Batch normalization to our input layer. We have two hidden layers. To both these layers, we apply a regularizer function followed by 20% and 10% dropout, respectively. We use ReLU as the activation function for the first three layers. We finally use Sigmoid activation for the output layer. The model loss is computed using a Cross-entropy loss function.

3) Model 3: We use an LSTM model with 5-layers. First, embeddings are generated for the input layers. Then it is sent as an input to a Bidirectional LSTM layer. We use a recurrent initializer for the weights of input. We then have a Keras Capsule layer, which is used for CapsNet networks [1].

This layer uses a non-linear transformation which is a Squashing function. This function is considered as an activation function. This is beneficial than a normal MaxPool layer because data routing is easier and faster in the Capsule layer. Finally, we have the output layer with a sigmoid activation function. Also, we use the NVIDIA CUDA Deep Neural Network Library to run our LSTM layer. This takes lesser time to execute than the normal LSTM layers [2]. Comparison of the models with and without Capsule Layer is shown in Fig. 4.

VI. Implementation

We implement the above-proposed models using the Mi-crosoft Malware Prediction Data set [7].

A. Data preprocessing

The data set [7] is huge and it consists of nearly 8.9 GB of training and testing data. The data consists of nearly 16.8M devices. Each row of the dataset has unique values of MachineIdentifier. HasDetections column is the label which gives information if the malware had been detected or not on a system. The training dataset has 8921483 rows and 83 columns. The dataset has information related to the hardware of the system and related to installed Anti-virus on the system. Anti-virus information and system information play a key role in determining if a machine is affected by malware. The data description is as show in in Fig. 5. And as it shows that there are rows with missing values which need to be removed. It also shows that there are a few rows that have binary input values like PuaMode. Also there are a few version numbers like OS version numbers. We need to reorganise the data so we can get the maximum insights. Also, computing columns that are not important will help us in saving some memory.

Also, the test data set has 7853253 rows and 82 columns. It would not have the HasDetections column as we are using it for testing.

We perform various visualization on the dataset. We gain insights from the data set about the counts of the OS type and the chasis type of the system. These are important values in determining what type of machine is an easy target for the malware. Fig. 5 and Fig. 6 explain the detection count based on the OS edition and the counts of touch and non touch devices.

B. Experiments

We perform the experiments of Model 1 and Model 2 on the complete dataset, on a system that has i7 Intel Hexa core pro-cessor and 16GM RAM with 500 GB SSD. The same system supports a 6GB NVIDIA GTX 1650 graphic processor. The LSTM model requires GPU to run as we use NVIDIA CUDA library for our model. The GPU based model was crashing on the system with above mention configuration. Hence, we reduced the dataset size and performed our experiments. We even then failed to perform our experiments hence, we ran it on Kaggle kernel with reduced dataset using the GPU facility provided by Kaggle.

We execute our Model 1 without embeddings for 20 epochs and record the AUC for the model. Model 2 is executed with K-Fold validation where we use 5-fold validation for the model. The model records the best validation loss for every fold and saves that. It stops execution if there is no change

	Feature	Unique_values	Percentage of missing values	Percentage of values in the biggest category	type
28	PuaMode	2	99.974119	99.974119	category
41	Census_ProcessorClass	3	99.589407	99.589407	category
8	DefaultBrowsersIdentifier	1730	95.141637	95.141637	float16
68	Census_IsFlightingInternal	2	83.04403	83.04403	float16
52	Census_InternalBatteryType	78	71.046809	71.046809	category
71	Census_ThresholdOptIn	2	63.524472	63.524472	float16
75	Census_IsWIMBootEnabled	2	63.439038	63.439038	float16
31	SmartScreen	21	35.610795	48.379658	category
15	OrganizationIdentifier	49	30.841487	47.037662	float16
29	SMode	2	6.027686	93.928812	float16
14	CityIdentifier	107366	3.647477	3.647477	float32
80	Wdft_IsGamer	2	3.401352	69.205344	float16
81	Wdft_RegionIdentifier	15	3.401352	20.177195	float16
53	Census_InternalBatteryNumberOfCharges	41087	3.012448	56.643094	float32
72	Census_FirmwareManufacturerIdentifier	712	2.054109	30.253692	float16
69	Census_IsFlightsDisabled	2	1.799286	98.199728	float16
73	Census_FirmwareVersionIdentifier	50494	1.794915	1.794915	float32
37	Census_OEMModelIdentifier	175365	1.145919	3.416271	float32
36	Census_OEMNameIdentifier	2564	1.070203	14.428946	float16
32	Firewall	2	1.023933	96.856251	float16
46	Census_TotalPhysicalRAM	3446	0.902686	45.894971	float32
79	Census_IsAlwaysOnAlwaysConnectedCapable	2	0.799676	93.50432	float16
62	Census_OSInstallLanguageIdentifier	39	0.673475	35.636026	float16
30	IeVerIdentifier	303	0.660137	43.55601	float16
42	Census_PrimaryDiskTotalCapacity	5735	0.594251	31.850422	float32
44	Census_SystemVolumeTotalCapacity	536848	0.594094	0.594094	float32
48	Census_InternalPrimaryDiagonalDisplaySizeInInches	785	0.52832	34.158346	float16
49	Census_InternalPrimaryDisplayResolutionHorizontal	2050	0.526661	50.608895	float16
50	Census_InternalPrimaryDisplayResolutionVertical	1552	0.526661	55.748814	float16
40	Census_ProcessorModelIdentifier	2583	0.46341	3.242555	float16
39	Census_ProcessorManufacturerIdentifier	7	0.463073	87.870122	float16
38	Census_ProcessorCoreCount	45	0.462995	60.866484	float16
9	AVProductStatesIdentifier	28970	0.405998	65.28696	float32
10	AVProductsInstalled	8	0.405998	69.594853	float16
11	AVProductsEnabled	6	0.405998	97.002942	float16
26	IsProtected	2	0.404014	94.180329	float16
6	RtpStateBitfield	7	0.362249	96.973642	float16
76	Census_IsVirtualDevice	2	0.178816	99.118499	float16
43	Census_PrimaryDiskTypeName	4	0.143967	65.087878	category
33	UacLuaenable	11	0.121482	99.271803	float32
47	Census_ChassisTypeName	52	0.006983	58.833402	category
16	GeoNameIdentifier	292	0.002387	17.171237	float16
51	Census_PowerPlatformRoleName	10	0.000616	69.30359	category
24	OsBuildLab	663	0.000235	41.004382	category
61	Census_OSInstallTypeName	9	0	29.233223	category
60	Census_OSSkuName	30	0	38.89341	category
64	Census_OSWUAutoUpdateOptionsName	6	0	44.325557	category
63	Census_OSUILocaleIdentifier	147	0	35.541445	int16
65	Census_IsPortableOperatingSystem	2	0	99.94548	int8
78	Census_IsPenCapable	2	0	96.192909	int8
58	Census_OSBuildRevision	285	0	15.845269	int32
66	Census_GenuineStateName	5	0	88.299187	category
67	Census_ActivationChannel	6	0	52.991067	category
77	Census_IsTouchEnabled	2	0	87.445686	int8
70	Census_FlightRing	10	0	93.65796	category
74	Census_IsSecureBootEnabled	2	0	51.39771	int8
59	Census_OSEdition	33	0	38.894778	category
0	MachineIdentifier	8921483	0	0.000011	category
57	Census_OSBuildNumber	165	0	44.935141	int16
20	OsVer	58	0	96.761323	category
2	EngineVersion	70	0	43.098967	category
3	AppVersion	110	0	57.605042	category
4	AvSigVersion	8531	0	1.146861	category
5	IsBeta	2	0	99.999249	int8
7	IsSxsPassiveMode	2	0	98.266622	int8
12	HasTpm	2	0	98.797106	int8
13	CountryIdentifier	222	0	4.451861	int16
17	LocaleEnglishNameIdentifier	252	0	23.477991	int8
18	Platform	4	0	96.606304	category
19	Processor	3	0	90.853001	category
21	OsBuild	76	0	43.888679	int16
56	Census_OSBranch	32	0	44.938246	category
22	OsSuite	14	0	62.328886	int16
23	OsPlatformSubRelease	9	0	43.888735	category
25	SkuEdition	8	0	61.80969	category
27	AutoSampleOptIn	2	0	99.997108	int8
34	Census_MDC2FormFactor	13	0	64.152103	category
35	Census_DeviceFamily	3	0	99.838256	category
1	ProductName	6	0	98.935569	category
45	Census_HasOpticalDiskDrive	2	0	92.281272	int8
54	Census_OSVersion	469	0	15.845202	category
55	Census_OSArchitecture	3	0	90.858045	category
82	HasDetections	2	0	50.020731	int8

Fig. 5. Data description of the Dataset

in validation loss of the model. Each iteration executes for 15 epoochs. MOdel 3 which is the GPU based LSTM model is trained for 20 epochs and AUC is then determined. We train this model on 100000 records which is 1/8th size of the origianl dataset.

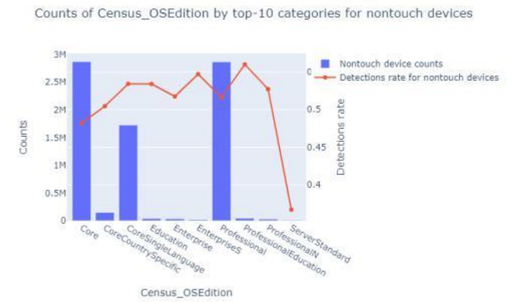

Fig. 6. Count of OS edition for top-10 Non-touch devices

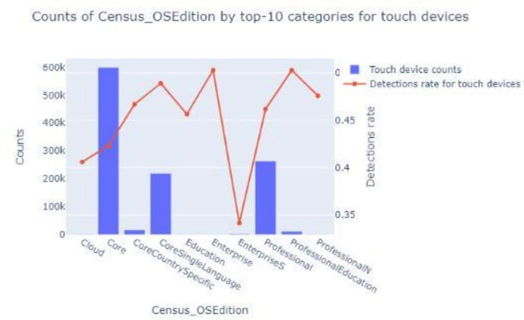

Fig. 7. Count of OS edition for top-10 touch devices

VII. Results

We determine the efficiency of our model by AUC metric. We chose this as a metric because of the skewed dataset and to not overfit any class. AUC is used to determine the hit rate to the false alarm rate and this is useful metric for prediciting if a machine could be affected with malware. Captured below in Fig. 8. are the results from our experiments :

	Model 1 (3- layer NN without embeddings)	Model 2 (4- layer NN with embeddings)	Model 3 (5-layer CuDNNLSTM model)
AUC	70.5035%	71.272%	73%

Fig. 8. AUC comparison of proposed models

As seen in the above table we don't have much difference in the AUC results. The model with embeddings does perform better than the model without embeddings. Also, overfitting seemed to be an easy target for the dataset and hence adding the hyperparameters to avoid overfitting served an essential task for all the models. We identify that the whole dataset could not be used for the LSTM model and only a small fraction was used to determine the achieved results. The reason was because of the lack of available computational resources. Hence, we can say that LSTM is a better model with the highest AUC out of the 3 models.

VIII. Conclusion and Future Works

The proposed Deep Learning models do not receive very high AUC and that can be improved further. We notice that lack of computational capacity effected the execution of our proposed Model-3. We further propose that we can achieve better reults if we can train our dataset with Adversarial validation. We tried to achieve the validation results, but could not input it to the Deep Learning model. Hence, we further try to achieve better AUC values and also explore Adversarial validation for our models.

Acknowledgment

We would like to thank Dr. Alioune Ngom for provid-ing us with all the resources and reference materials to help understand the concepts of neural networks and deep learning. Also, for sharing thesis work of William Briguglio ”Interpreting Machine Learning Malware Detectors Which Leverage N-gram Analysis”. This paper helped us a lot in understanding more about malware detection and other cyber attack detection. This project helped us to understand more about methods and models that can further improve the overall performance of deep learning models in malware detection compared with traditional learning methods.

References

[1] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil DB Bruce, Yang Wang, and Farkhund Iqbal. Malware classification with deep convolutional neural networks. In 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pages 1–5. IEEE, 2018.

[2] Yan Lu, Jonathan Graham, and Jiang Li. Deep learning based malware classification using deep residual network. In Proceedings of the 2019 Modeling, Simulation and Visualization Student Capstone Conference, 2019.

[3] kaggle. CapsNet, 2018. https://www.kaggle.com/fizzbuzz/beginner-s-guide-to-capsule-networks.

[4] kaggle. LSTM, 2018. https://www.kaggle.com/bgeier/word2vec-keras-ltsm-capsule-try/notebook.

[5] kaggle. NN-Embeddings, 2018. https://www.kaggle.com/cdeotte/neural-network-malware-0-67.

[6] Oren Melamud, Jacob Goldberger, and Ido Dagan. context2vec: Learn-ing generic context embedding with bidirectional lstm. In Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning, pages 51–61, 2016.

[7] microsoft. Dataset, 2019. https://www.kaggle.com/c/microsoft-malware-prediction/data.

[8] Lakshmanan Nataraj, S Karthikeyan, and BS Manjunath. Sattva: Sparsity inspired classification of malware variants. In Proceedings of the 3rd ACM Workshop on Information Hiding and Multimedia Security, pages 135–140, 2015.

[9] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 1916–1920. IEEE, 2015.

[10] Muhammad Furqan Rafique, Muhammad Ali, Aqsa Saeed Qureshi, Asifullah Khan, and Anwar Majid Mirza. Malware classification using deep learning based feature extraction and wrapper based feature selection technique. arXiv preprint arXiv:1910.10958, 2019.

[11] Wikipedia contributors. Long short-term memory — Wikipedia, the free encyclopedia, 2020. [Online; accessed 1-May-2020].