Fall 12-17-2020

Malware Classification Using LSTMs

Dennis Dang
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the [Artificial Intelligence and Robotics Commons](#), and the [Information Security Commons](#)

Recommended Citation

Dang, Dennis, "Malware Classification Using LSTMs" (2020). *Master's Projects*. 963.
DOI: <https://doi.org/10.31979/etd.c5r3-dsye>
https://scholarworks.sjsu.edu/etd_projects/963

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Malware Classification Using LSTMs

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Dennis Dang

December 2020

© 2020

Dennis Dang

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Malware Classification Using LSTMs

by

Dennis Dang

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2020

Dr. Mark Stamp Department of Computer Science

Dr. Fabio Di Troia Department of Computer Science

Dr. Thomas Austin Department of Computer Science

ABSTRACT

Malware Classification Using LSTMs

by Dennis Dang

Signature and anomaly based detection have long been quintessential techniques used in malware detection. However, these techniques have become increasingly ineffective as malware becomes more complex. Researchers have therefore turned to deep learning to construct better performing models. In this project, we create four different long-short term memory (LSTM) models and train each model to classify malware by family type. Our data consists of opcodes extracted from malware executables. We employ techniques used in natural language processing (NLP) such as word embedding and bidirection LSTMs (biLSTM). We also use convolutional neural networks (CNN). We found that our model consisting of word embedding, biLSTMs and CNN layers performed the best in classifying malware.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Mark Stamp for taking me on as his student, for his guidance, for helping me whenever I was stuck and for his great sense of humor. With his vast knowledge and expertise, he was always able to see things that I could not and always knew what to do to move the project forward whenever I ran into issues. I also want to thank him for teaching the machine learning course at SJSU that sparked my interest in machine learning.

I want to thank Dr. Fabio Di Troia for his help in providing malware data to use for this project and for helping me with data preprocessing. I've gotten the chance to know Dr. Fabio over the course of taking three courses that he has taught. He is a very friendly, knowledgeable and a down to earth person. I would like to thank him for all the help that he's given me. I also want to thank Dr. Thomas Austin for being on my committee. I got to know Dr. Austin when I enrolled in his CS252 Advanced Programming Language Principles class. Not only is he a great teacher but he is also very friendly and approachable.

And finally, I want to thank my family for all of their support, kindness, appreciation and understanding from when I started preschool to where I am now as a master's student. I would not have made it through these 20 tumultuous years of education without them. I want to make a special shout-out to my two year old twin nephews Ethan and Eric. They are the best, most silly and dorky nephews that any uncle can ask for.

TABLE OF CONTENTS

CHAPTER

1	Introduction	1
2	Background	4
2.1	Previous Work	4
2.2	Recurrent Neural Networks	5
2.3	The Vanishing and Exploding Gradient	7
2.4	Long Short-Term Memory	8
2.5	LSTM Training	12
2.6	Bidirectional LSTM	14
2.7	Word Embedding	15
2.8	Convolutional Neural Networks and Max Pooling	16
2.9	TensorFlow Layers	18
3	Dataset Preparation and Experimental Design	20
3.1	Hardware and Software Setup	22
3.2	Model Parameters	23
3.3	Training and Testing Methodologies	23
4	Experiments and Results	26
4.1	Using MLPs Only	26
4.2	LSTM Model Without Embedding	26
4.3	LSTM Model With Embedding	27
4.4	BiLSTM Model With Embedding	27

4.5	Embedding, BiLSTM and CNN Model	29
4.6	Comparison of Results	30
5	Conclusion and Future Work	35
	LIST OF REFERENCES	37
	APPENDIX	
	Confusion Matrices	40

LIST OF TABLES

1	Number of samples of malware files per family.	21
2	Relevant desktop PC hardware specifications.	22
3	Relevant software, operating system and Python packages.	23
4	The parameters and values for these parameters used in a grid search to determine the best possible combination of values to use in the models for this project.	23
5	The values of the parameters used in the models.	24
6	Groupings of each family for training.	25
7	Number of samples for training and testing	25
8	Experimental results for the model using just MLPs.	27
9	Experimental results for the model without embedding.	28
10	Experimental results for the model with embedding.	29
11	Experimental results for the model with embedding and bidirectional LSTM.	30
12	Experimental results for the model with embedding, biLSTM and CNN.	32

LIST OF FIGURES

1	Illustration comparing the differences between an RNN and a feed-forward network [1].	6
2	The structure of an unrolled RNN layer [2].	6
3	Unrolled structure of a vanilla RNN cell and an LSTM cell [3]. . .	8
4	The forget gate in a LSTM cell [3].	10
5	The input gate gate in a LSTM cell [3].	11
6	The output gate in a LSTM cell [3].	11
7	Updating the cell state [3].	12
8	The architecture of an unfolded bidirectional LSTM with three consecutive timesteps [4].	15
9	One-hot encoding vector representations of text.	15
10	Examples of word embedding vectors for various vectors and their 2D plots [5].	17
11	Sample opcode file.	21
12	Structural layout of the baseline model using just MLPs.	26
13	Structural layout of the LSTM model without embedding.	26
14	Structural layout of the LSTM model with embedding.	28
15	Structural layout of the LSTM model with embedding and bidirectional LSTM.	29
16	Structural layout of the model incorporating biLSTMs, embedding and a CNN layer where the 1DConv layer is a one dimensional convolutional layer and the MaxPool1D is a one dimensional max pooling layer.	31
17	Comparison of the average evaluation accuracy values for the four models used in this experiment for classifying malware families. .	31

A.18	Confusion matrix for the model using just MLPs only.	40
A.19	Confusion matrix for the model without embedding.	41
A.20	Confusion matrix for the model with an embedding layer.	42
A.21	Confusion matrix for the model with embedding and biLSTM layers.	43
A.22	Confusion matrix for the model with embedding, biLSTM and CNN and max pooling layers.	44

CHAPTER 1

Introduction

Malicious software (malware) are computer programs that are created to harm a computer, computer systems or a computer user [6]. Malware attacks can disrupt a person's or organization's day-to-day use of their computer systems, steal personal or confidential information, corrupt files or annoy users. Malware can be categorized into different families where the behavior of one malware from one particular family differs from that of another family. The authors in [7] and [8] for example, discuss the behavior of many different types of malware families.

Malware attacks are generally facilitated by the Internet. With the rise in the number of devices around us that are connected to the Internet, it has become more important than ever to keep our devices safe lest we risk loss of personal or confidential information [7]. Malware attacks can even be life threatening. In 2017, a ransomware¹ attack crippled many parts the United Kingdom's National Health Service (NHS) [9]. Computer systems containing data pertaining to the health and history of thousands of patients were targeted across dozens of hospitals in the UK. Hospitals were forced to pay a ransom to have their files unlocked or risk having them files corrupted and deleted forever. These attacks not only caused doctors and nurses to cancel some 19,000 appointments but also cost the NHS £92 million. Malware is clearly a challenge that warrants a significant research effort.

Malware detection techniques include signature based detection, anomaly based detection and machine learning based detection [6]. Signature based detection has long been the most popular approach to detecting malware. In signature based detection, each malware sample is first analyzed and a signature is extracted, which is then used

¹Ransomware is a type of malware that threatens to corrupt, delete, publish or block the victim's data unless a ransom is paid.

to identify the malware. If the signature is found in another sample, that sample is flagged as possible malware. However, various code obfuscation and code morphing techniques can easily thwart signature based detection mechanisms. In anomaly based detection, the system looks for activity that falls outside the “normal” range of the computer and flags such behavior as suspicious [10]. A major downside to anomaly based detection is the high rate of false positives. The drawbacks of signature and anomaly based detection techniques have motivated the rise of machine learning based detection.

Many classical machine learning algorithms have found success in detecting malware [11]. These algorithms include support vector machines (SVM), hidden markov models (HMM), random forest and naive Bayes, among many others. These models rely heavily on proper feature extraction from the dataset. Deep learning techniques have gained a lot of traction. Particularly, multilayer perceptrons (MLPs), convolutional neural networks (CNNs), extreme learning machines (ELM) have been used with great success [12]. Other techniques involving variants of recurrent neural networks (RNN) such as gated recurrent units (GRUs) and long-short term memories (LSTMs) have received far less attention in the literature [13].

In this research, we focus on using LSTMs to classify malware by family. We build on the work in [13] by combining various aspects of the methodologies employed in [14], [15] and [16]. Our dataset includes opcodes from malware belonging to 20 distinct families, and we use opcode sequences as our features. We consider five models, with each model being successively more complex. Our first model is the most basic consisting of only MLPs. This model serves as a baseline from which we compare our other LSTM models to. Our second model consists of only one LSTM layer. Our third model is an enhanced LSTM that includes an embedding layer, similar to the model considered in [13]. Our fourth model replaces the LSTM layer from our second

previous model with a biLSTM layer. Finally, our fifth model includes everything from our third model, plus an additional one-dimension CNN layer and a one-dimension max pooling layer. As far as we are aware, our fourth and fifth models have not previously been considered in the literature.

The remainder of this paper is organized as follows. Section 2 discusses relevant previous work and introduces the various deep learning techniques employed in this research. Section 3 covers the dataset, feature extraction, parameters, and so on. In Section 4, we present our experimental results. Finally, Section 5 concludes the paper, and we mention possible directions for future work.

CHAPTER 2

Background

2.1 Previous Work

The authors of [14] consider various models for malware classification. In one of these models, a two stage classifier is used— the first stage is either an LSTM or GRU which is used to derive features for a second stage classifier consisting of a single MLP layer. Another model uses a single stage classifier consisting of nine CNN layers. When trained and evaluated, both models achieved an about 80% accuracy.

In [15], the author proposes a novel deep learning architecture that includes both a CNN layer and an LSTM layer. This model is then trained on API call sequences. The CNN portion of the model consists of filters of increasing size, with the output of each filter fed into the LSTM layer. The output of the LSTM layer is used as input to a dropout layer, with a final fully connected layer for classification. The output of the dense layer is the model’s prediction for the given input. This model achieved an accuracy approaching 100%.

The authors of [16] consider a biLSTM based model to classify malware in a cloud based system. The model includes a CNN layer and is trained on system call sequences. The authors achieve an overall accuracy of approximately 90%. Interestingly, the authors also show that substituting the biLSTM for a regular LSTM layer resulted in worse accuracies in almost all cases.

The author in [13] classifies malware using an entirely different approach from the two papers mentioned above. The work in [13] is based on opcodes obtained from disassembled executables. This research also employs word embedding as a feature engineering step. Word embedding techniques are often used in natural language processing (NLP) applications. The result from word embedding are fed into an LSTM layer. For malware detection, this model attains an average AUC of 0.99, while for

classification, the model achieves an average AUC of 0.987.

Besides using LSTMs and its architectural variants, other machine learning algorithms have been used to classify malware. For instance, in [17], the authors classify malware using random forests. Their technique is a three stage process. First, malware binaries are visualized as images. This allows models to better identify patterns within the binaries which, in turn, allows models to overcome obfuscation tactics. While obfuscation introduces changes to the binary images, the general structure of the image is the same allowing models to identify the malware type. After converting the binaries to images, the next step in the authors’ technique is feature extraction from the images. Finally, random forest classifiers are applied on the resulting features. Using this technique, the authors’ were able to achieve a classification accuracy of over 98%.

In [18], the authors employ six different machine learning algorithms. The algorithms used are random forest, k-nearest neighbors (kNN), support vector machines (SVM), naive Bayes, MLPs and decision trees. The dataset that was used consisted of the malware signatures from approximately 5,800 malware samples from the Zeus, APT1, Crypto, Locker and shadowbrokers families. Using two different machine learning libraries, the authors were able to achieve an accuracy of 89.8%, 87.6%, 87.8%, 94.2%, 92.5% and 54.3% for the kNN, naive Bayes, MLP, random forest, decision tree and SVM algorithms, respectively.

2.2 Recurrent Neural Networks

In feed-forward neural networks¹, all training samples are treated independently of each other [19]. Consequently, feed-forward networks are impractical for cases where training samples depend on previous samples. Thus, a different type of architecture is needed in cases where “memory” is required, as when training on time series data.

¹Also called multilayer perceptrons (MLP).

Figure 1: Illustration comparing the differences between an RNN and a feed-forward network [1].

Recurrent neural networks (RNN) serve to add memory to the network [20]. As shown in Figure 1, the output in a RNN depends not only on the current input but also the input from the past, as indicated by a feedback loop. Whereas information only flows forward in a feed-forward network, information from the previous timesteps are available at each subsequent timestep in RNNs [21]. An unfolded structure of a RNN shown in Figure 2 shows a more complete picture of how data flows in RNNs [22].

Figure 2: The structure of an unrolled RNN layer [2].

In Figure 2, x_t is the input at timestep t and is associated with U which is a

weight matrix. The output at timestep t is represented by o_t has a corresponding weight matrix V . The memory of the network (also called the “hidden state”) at timestep t is represented by s_t and is calculated from the previous timestep. The weight matrix for s_t is symbolized by W . To calculate s_t , we use the current input and previous hidden state in the function $s_t = f(Ux_t + Ws_{t-1})$ where f is some kind of nonlinear activation function such as tanh or ReLU.

2.3 The Vanishing and Exploding Gradient

RNNs perform well with short term dependencies but struggle with longer term dependency. The wider the gap between the contextual information and the place that it is needed, the worse the model performs. This long-term dependency problem occurs due to the “vanishing gradient” problem [23]. During training, RNNs calculate gradients to update the weights of the model. Due to the architecture of the model, calculating these gradients involve multiplying together many values less than one [23]. The longer sequence of data, the more terms get added to these products. As a result, these values become so small that the gradients quickly approach zero. Since the change in the model’s weights depend on the magnitude of the gradients, a small gradient leads to either slow learning or no learning whatsoever. These small gradient values are therefore said to be “vanishing”.

The vanishing gradient occurs when either the sigmoid or hyperbolic tangent activation functions is used. The opposite behavior called the “exploding gradient” occurs while using the ReLU activation function [24]. Rather than multiplying together small numbers, the exploding gradient occurs when the model multiplies together large values. With enough terms, this product can grow exponentially fast thereby making it impossible for the RNN model to learn the temporal relationship between distant events.

2.4 Long Short-Term Memory

To overcome the vanishing and exploding gradient issues, complex gated RNN architectures have been developed. The best known and most widely used of these is long short-term memory (LSTM) models. LSTMs address the issue of long-term dependency by, in effect, decoupling the memory from the output of the network and ensuring that additive updates are done to the memory, rather than multiplicative updates. With additive updates, the gradient is more stable.

Figure 3: Unrolled structure of a vanilla RNN cell and an LSTM cell [3].

Other than ensuring a more stable gradient, there are more architectural differences between RNNs and LSTMs. Figure 3 illustrates the differences between the internal structures of a RNN cell and a LSTM cell. LSTMs maintain two hidden

states that are carried along from one timestep to the next. The hidden state h_{t-1} consists of a modified form of the output from the previous timestep $t - 1$ to the next timestep t . The cell state is represented by c_{t-1} . It is the model's memory and carries information over a long period of time of the inputs from earlier timesteps. If the model encounters information that is deemed important, it is added to the cell state. Otherwise, information that is not deemed important will be deleted. This is necessary as carrying too much information from the start to the end of the input data sequence can cause poor performance.

Other than these states, LSTM models consist of three gates. These are the forget gate f_t , the input gate i_t and the output gate o_t [3]. The input gate determines what kind of new information is to be added to the overall memory, the forget gate determines what kind of information to forget and the output gate determines what kind of values the cell will pass to the next timestep. These gates are further explained as follows.

1. **Forget Gate:** The forget gate f_t determines what information to remove from the model's memory. Removing certain information is vital to the model since the model will perform poorly if it tries to remember everything. The value of this gate is determined by calculating

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad (1)$$

Where W_f and U_f are the weights on x_t and h_{t-1} for the forget gate respectively and b_f is a constant and called a bias parameter. Since f_t is a sigmoid function, the output will be a value between 0 and 1. A value of 0 indicates completely forgetting a learned value while a value of 1 indicates completely keeping the value. Figure 4 shows where the forget gate resides in the model.

Figure 4: The forget gate in a LSTM cell [3].

2. **Input Gate:** The input gate i_t determines if new data should be added to the model's memory and consists of a sigmoid function and a tanh function. The tanh portion creates candidate values to add to memory while the sigmoid portion determines which of these candidate values should be added and to what extend each value should be committed to memory. Symbolically, these two portions are expressed as

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad (2)$$

$$\tilde{C}_t = \tanh(W_{\tilde{C}} x_t + U_{\tilde{C}} h_{t-1} + b_{\tilde{C}}) \quad (3)$$

Similarly as before, W_i and U_i are weights on x_t and h_{t-1} for the input gate respectively and b_i is a bias constant. The candidate values is represented as $\tilde{C}_t$. Figure 5 shows where the input gate resides in the model

3. **Output Gate:** The output gate o_t determines the value of the output of the model at the current timestep. Just like the input gate, this gate also consists of a sigmoid portion and a tanh portion. The sigmoid portion determines what parts of the memory will be in the output. In the tanh portion, tanh is applied to the cell state to push the values between -1 and 1 . Finally, the final output is a product of the sigmoid and tanh portions which represents a filtered version

Figure 5: The input gate gate in a LSTM cell [3].

of the cell state. These two portions are symbolically expressed as

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad (4)$$

$$h_t = o_t \tanh(C_t) \quad (5)$$

Where W_o and U_o are weights on x_t and h_{t-1} and b_o is the bias constant. The output of the model h_t for timestep t is the hidden state for the next timestep $t + 1$. Figure 6 shows where the output gate resides in the model.

Figure 6: The output gate in a LSTM cell [3].

In between calculating the values for the input and output gates, the model updates its memory. Figure 7 shows where updating the cell state occurs in the model. This process is done in two steps. First, portions of the memory that are no longer important are forgotten. This is done by multiplying the cell state from the previous

timestep C_{t-1} with the forget gate f_t . Afterwards, the new values are added to the memory using the input gate i_t and the candidate values $\tilde{C}_t$. Symbolically, the new cell state for the current timestep C_t is expressed as

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t \quad (6)$$

Figure 7: Updating the cell state [3].

2.5 LSTM Training

Training a LSTM model consists of a performing a forward pass followed by a backwards pass. In the forward pass, the model calculates the values of the forget, input, candidate and output gates as described in equations 1, 2, 3 and 4 respectively [3]. These values along with the input at the current timestep are then used to calculate the hidden state, cell state and output for the current timestep [25]. These calculations are repeated for every timestep. The forward pass is complete once the error of the output is calculated using the known output and a specified loss function.

Once the forward pass is done, we do a backwards pass where the weights and biases of the model are updated based on the error calculated in the previous steps. The backwards pass is thus where the model does the learning.

For a given input, there is a final output value of the model $\hat{y}_t$ along with what the value that the model should have produced y_t for the given input. Using these

values, we can specify an error function as a function of both values as

$$E(\hat{y}_t, y_t)$$

The exact function for $E(\hat{y}_t, y_t)$ varies greatly based on the application of the model. A plethora of loss functions are discussed in detail in [26]. Once an error function is known, the LSTM model updates its weights and biases by calculating the gradient of the error function with respect to all weights and biases used in the model [27]. For timestep t , we will have an error E_t . The gradient of E_t is

$$\nabla E_t = \left(\frac{\partial E_t}{\partial W}, \frac{\partial E_t}{\partial U}, \frac{\partial E_t}{\partial b} \right)$$

$$W = (W_i, W_f, W_{\tilde{c}}, W_o)$$

$$U = (U_i, U_f, U_{\tilde{c}}, U_o)$$

$$b = (b_i, b_f, b_{\tilde{c}}, b_o)$$

These calculations must be done at every timestep [27]. Thus for T timesteps, we have

$$\begin{aligned} \frac{\partial E}{\partial W} &= \sum_{t=1}^T \frac{\partial E_t}{\partial W} \\ \frac{\partial E}{\partial U} &= \sum_{t=1}^T \frac{\partial E_t}{\partial U} \\ \frac{\partial E}{\partial b} &= \sum_{t=1}^T \frac{\partial E_t}{\partial b} \end{aligned}$$

Once the gradients are calculated, the weights and biases of the model are updated using

$$\begin{aligned} W &= W - \alpha \frac{\partial E}{\partial W} \\ U &= U - \alpha \frac{\partial E}{\partial U} \\ b &= b - \alpha \frac{\partial E}{\partial b} \end{aligned}$$

Where α is the learning rate of the model. Once this calculation is done, the backward pass is complete. These steps are repeated for all input data into the model.

2.6 Bidirectional LSTM

BiLSTM models are an expansion of LSTMs that process a sequence of data in both forward and backward directions in two separate LSTM layers [28]. The forward layer processes the data in the same way as a standard LSTM, while the backward layer processes the same data but in reverse order [29]. Figure 8 shows an unfolded architecture of a biLSTM with three consecutive timesteps. At every timestep, there are two hidden states values. The hidden state values in the forward layer $\vec{h}$ are calculated from timestep t_0 to t_{n-1} in the forward direction where n is the total number of timesteps for a given sequence of data. Correspondingly, the hidden state values in the backward layer $\overleftarrow{h}$ are calculated in reverse order from t_{n-1} to t_0 . A final output y_t is calculated at each timestep that combines together both of these hidden states. This is represented as

$$y_t = f(\vec{h}_t, \overleftarrow{h}_t)$$

Where f is a function that can be a summation, multiplication, concatenation or an average function [26]. The type of function used will depend on the nature of the problem and will generally vary from one application the next.

The consequence of combining the forward and backward layers is that the outputs at every timestep will include information not only from future timesteps but also from timesteps in the past. This kind of model is useful in applications such as machine translation. In machine translation, it is often necessary to know the words that occur later on in the sentence in order to produce an accurate translation. Normal LSTM models would not work in this application because these models can only make use of information from previous timesteps. BiLSTMs would be the model

Figure 8: The architecture of an unfolded bidirectional LSTM with three consecutive timesteps [4].

to use if the output at any timestep depends on future and past timesteps.

2.7 Word Embedding

One of the challenges in NLP is converting words into numbers for machine learning models. One-hot encoding is a technique that solves this problem. In one-hot encoding, words are represented by one-dimensional vectors of zeroes at all indices except at a certain index in which the value at that index is one [30]. The location of this designated index varies from one word to the next. Additionally, the length of the vector grows by one for every new word encountered in the dataset. Examples of one-hot encoding vectors for various European places are shown in Figure 9.

	Rome	Paris						n^{th} word
Rome	=	[1, 0, 0, 0, 0, 0, ...]						0]
Paris	=	[0, 1, 0, 0, 0, 0, ...]						0]
Italy	=	[0, 0, 1, 0, 0, 0, ...]						0]
France	=	[0, 0, 0, 1, 0, 0, ...]						0]

Figure 9: One-hot encoding vector representations of text.

While one-hot encoding allows text to be represented numerically in a systematic way, there are many disadvantages. One major disadvantage is that one-hot encoding does not efficiently use memory. This is because only the index of one value within the vector distinguishes one word from the next. The size of these vectors can grow tremendously for datasets where there are many unique words and if there are too many words, there may be inadequate memory to store these one-hot encoded vectors that consist mainly of zeroes. Another drawback is in the inability to represent words of similar meaning in a similar way. It has been found that similar words having similar representations greatly increases the model's ability to learn words and language [31].

An alternative to one-hot encoding is word embedding. In word embedding, words are represented as vectors of floats [31]. Unlike one-hot encoding, the size of this vector does not grow for each additional new word encountered in the dataset. Once an initial size is set, these vectors remain fixed at that size. Additionally, word embedding allows similar representation of similar words. If two words are similar in meaning, the values in the corresponding vectors of each word are close in value which allows models to learn words easier and faster. Examples of vectors for various words are shown in Figure 10.

Word embedding layers are neural networks that are trained with the rest of the model. With an adequately large dataset, word embedding layers can learn which words are similar in meaning through context and can therefore learn the vector representations of these words that the rest of the machine learning model can use.

2.8 Convolutional Neural Networks and Max Pooling

Convolutional neural networks (CNNs) are designed to efficiently deal with local structure [32]. CNNs are often used with images, but the technique can excel in any

Figure 10: Examples of word embedding vectors for various vectors and their 2D plots [5].

application where local structure dominates.

The hidden layers within a CNN act as filters where each filter specializes in detecting a certain feature within the data, while deeper layers detect progressively more abstract features. For example, when training on images, first layer filters might detect vertical and horizontal lines, the final layer might be able to distinguish between images of, say, dogs and cats.

Within every CNN layer of a model, the number of filters and size of each filter must be specified. These filters are matrices with relatively small dimensions. In two-dimensions, we can specify a filter $F = \{f_{ij}\}$ with dimensions $n \times m$. We can also specify a two-dimensional matrix $A = \{a_{ij}\}$ of dimensions $N \times M$ representing an image. To apply the filter F to the image A , we perform a convolving operation

$$c_{ij} = \sum_{k=0}^{n-1} \sum_{l=0}^{m-1} f_{kl} \cdot a_{i+k, j+l}$$

Where $i = 0, 1, \dots, N - n$ and $j = 0, 1, \dots, M - m$. The elements of the smaller

matrix C is represented by c_{ij} with dimensions $(N - n) \times (M - m)$ that is the result of applying the filter F to image A . If the CNN model contains more layers, than this output matrix C becomes the input for the next layer. This operation is repeated for all CNN layers.

While not strictly required, pooling layers can be applied in between CNN layers. These layers reduce the dimensionality, thereby reducing the computational load. Pooling can also reduce noise and potentially improve performance. In max pooling, we specify a window size and only the maximum value within each (non-overlapping) window is retained.

2.9 TensorFlow Layers

TensorFlow models are created by adding various layers in sequence. What distinguishes one model from another is the type of layers used and the parameter passed into the constructors of each layer. A short description of each layer is provided below [33].

- **Input Layer:** The first layer and entry point into a neural network
- **Dropout Layer:** Adds noise to the network during training by randomly severing the number of connections between neurons from one layer to the next. In doing so, overfitting is reduced allowing models to better generalize. This has the effect of increasing model accuracy during evaluation.
- **LSTM Layer:** Implements a single LSTM layer with all of the algorithms required for forward and backward propagation.
- **Bidirectional Layer:** A wrapper layer that allows RNN layers to implement bidirectional models. Rather than implementing two separate RNN layers for the forwards and backwards direction and concatenating the results, the bidirectional wrapper layer does all of this in one layer.

- **Dense Layer:** Implements a single fully connected vanilla neural network layer.
- **Embedding Layer:** Responsible for mapping positive integers to vectors of floating point values.
- **Conv1D Layer:** Implements the convolutional neural network layer in one dimension.
- **MaxPooling1D Layer:** Implements the max pooling operation in one dimension.

CHAPTER 3

Dataset Preparation and Experimental Design

The dataset used in this project was acquired from [8] and from [34]. Our dataset consists of binary files from 20 distinct malware families. The names of the malware families and the number of samples per family is shown in Table 1.

To extract features from our dataset, we first disassemble every executable file and extracted mnemonic opcode sequences. Afterwards, we perform a frequency analysis on all opcodes. The results from this frequency analysis is used to sort opcodes in order of decreasing frequency. Next, we create an opcode to integer mapping where each opcode is assigned a unique integer. Finally, we use this mapping to convert each opcode mnemonic into integers.

We retain the 30 most frequent opcodes, with all other opcodes grouped into a single “other” category. The top 30 opcodes represent about 80% all opcodes in our dataset.

File disassembling consists of obtaining the opcode instructions for the binary file. This was done by importing the files into a Ubuntu virtual machine and using the `objdump` Linux command. To display the opcodes, the flag `-d` was used. This flag indicates that only the machine instructions portion of the binary file is to be disassembled; other sections such as headers are ignored. The result of running this command is piped through various `sed` and `cut` commands in order to isolate the opcodes. After running these commands for a single executable file, we obtain the executable file’s opcodes. An example of a file’s opcodes written to a text file is shown in Figure 11.

After disassembling, the next step is to create a lookup table that maps a specific opcode to an integer. This is necessary because the model cannot directly work with an opcode string; models can only work with numbers. To do this, a frequency analysis

Malware Family	Number of Samples
Adload	1,044
Agent	817
Alureon	1,327
Bho	1,159
CeeInject	886
Cycbot	1,029
DelfInject	1,097
Fakerean	1,063
Hotbar	1,476
Lolyda	915
Obfuscator	1,331
Onlinegames	1,284
Rbot	817
Renos	1,309
Starpag	1,084
Vobfus	924
Vundo	1,784
Winwebsec	3,651
Zbot	1,785
Zeroaccess	1,119

Table 1: Number of samples of malware files per family.

Figure 11: Sample opcode file.

is first done where the most frequent opcodes that are present throughout the dataset appear at the beginning of a list. Afterwards, the first 30 opcodes are assigned a

unique incrementing integer starting at 1. All other opcodes past the first 30 opcodes are assigned the number 31. A dictionary is then created using this data where the keys are the opcode string and the values are the associated integer assigned to each opcode. Using this dictionary as a lookup table, the opcodes from the text files are converted into numeric values that can be used in a model.

The models used in this project require all input data to be of the same length. To accomplish this, we experimented with various opcode sequence lengths, as discussed below. Of course, truncating the opcode sequence results in a loss of information, but using a short sequence improves efficiency, and we have obtained strong results with relatively short feature vectors.

3.1 Hardware and Software Setup

The models used in this project were run on a PC desktop. The specifications of this machine is shown in Table 2. Software, operating system and Python packages used are shown in Table 3.

Hardware	Feature	Details
CPU	Brand and Model	Intel i7-8700
	Base Clock Speed	3.2 GHz
	# Core	6
	# Threads	12
GPU	Chipset	NVIDIA GeForce GTX 1070 Ti
	Video Memory	8GB GDDR5
	Memory Speed	1683 MHz
	Cuda Cores	2432
DRAM	Brand and Model	G. Skill TridentZ RGB Series
	Amount	$2 \times 8\text{GB} = 16\text{GB}$
	Speed	3200MHz
Motherboard	Brand and Model	MSI Z370 SLI Plus LGA 1151

Table 2: Relevant desktop PC hardware specifications.

Software	Version Details
OS	Windows 10 Pro
Python	3.8.3
Jupyter Notebook	6.1.4
Numpy	1.18.5
Scikit Learn	0.23.2
Tensorflow-GPU	2.3.1
CUDA Toolkit	10.1
cuDNN SDK	7.6
NVIDIA GPU Drivers	431.36
Oracle VM VirtualBox	6.0.10
VM OS	Ubuntu 18.04.5 LTS

Table 3: Relevant software, operating system and Python packages.

3.2 Model Parameters

Deep learning models generally have many parameters. For each of our models, we performed a grid search over all combinations of the values listed in Table 4. All models were trained and evaluated on the same dataset. For every model evaluated, the accuracy was determined and the parameters for the model with highest accuracy were generally selected. In a few cases where accuracy differences were deemed insignificant, we selected parameters so that training times were reduced. Table 5 shows the values of the parameters selected.

Parameter	Values Used
Opcode Lengths	[2000, 4000, 6000, 8000, 10000]
LSTM Units	[16, 32, 64, 128, 256]
Embedding Vector Lengths	[16, 32, 64, 128, 256]
Dropout Amount	[0.1, 0.2, 0.3, 0.4]

Table 4: The parameters and values for these parameters used in a grid search to determine the best possible combination of values to use in the models for this project.

3.3 Training and Testing Methodologies

The dataset was sorted in ascending order based on the number of training samples per family. The dataset was then partitioned into four groups of five families

Parameter	Value
Batch Size	32
Maximum Number of Epochs	100
Percentage of Data to be Used in Testing	15%
Number of Unique Opcodes Used	30
Opcode Sequence Length	2,000
Dropout Amount	30%
Number of LSTM Units	16
Number of Dense Units	50
Embedding Vector Length	128
CNN Kernal Size	3
Number of CNN Filters	128
Max Pooling Size	2

Table 5: The values of the parameters used in the models.

each, where the first group consisted of families with the most malware samples, while the last group consisted of families with the least samples. The dataset was partitioned in this manner in order to allow the models to have the highest accuracy possible when classifying each group of families. The more samples there are for a family, the easier it should be for the models to learn that particular and tell it apart from other families. And so by grouping the families in this order, we should be able to see the highest accuracy possible for each model.

The models were trained on the first group of 5 families, then the second group of 10 (i.e., the first and second groups of 5), then the third group of 15, and finally on all families together. With each additional group, the difficulty of classifying malware by family increased. Table 6 shows the families that constitute each group. Table 7 shows the number of training and testing samples for each additional group.

The initial values of the weights of the LSTM are randomly selected and the embedding and dense layers are randomly initialized each time the models are trained,

and hence the model accuracy will likely vary each time a model is trained. Therefore, we train each model type on each grouping of malware families five times. At the start of every training run, the dataset is shuffled before being split into training and testing sets. The average of these five values are used to compare the model types.

Group Number	Total Number of Files	Malware Families
1	10,005	Winwebsec Zbot Vundo Hotbar Renos
2	6,220	Obfuscator Alureon Onlinegames Bho Zeroaccess
3	5,317	Delfinject Startpage Fakerean Adload Cycbot
4	4,359	Vobfus Lolyda Ceeinject Agent Rbot

Table 6: Groupings of each family for training.

Groups	Families	Samples	
		Training	Testing
1	5	8,480	1,472
1,2	10	13,760	2,400
1,2,3	15	18,272	3,200
1,2,3,4	20	21,984	3,872

Table 7: Number of samples for training and testing

CHAPTER 4

Experiments and Results

In this section, we give experimental results for each of the four model types tested. We conclude this section with a comparison of the different models.

4.1 Using MLPs Only

The structural layout of our first model using only MLPs is given in Figure 12. Note that in this model, no LSTMs were used. The MLP layers are represented by dense layers. The first dense layer learns the features of each input while the second dense layer is the classifier. The experimental results for this model appear in Table 8. For five families, this model performs quite well with average accuracy of 83.56%. However, the accuracy drops significantly when more families are added.

Figure 12: Structural layout of the baseline model using just MLPs.

4.2 LSTM Model Without Embedding

The structural layout of our most basic LSTM model is given in Figure 13. Note that the model consists of four types layers, namely, an input layer, dropout layers, an LSTM layer, and a dense layer. The experimental results are shown in Table 9. This model struggles with classifying just five families, with an accuracy value of 55.73%. The accuracy drops as more families are classified with the largest drop in accuracy occurring from classifying five families to ten. Clearly, a more sophisticated model is required.

Figure 13: Structural layout of the LSTM model without embedding.

<i>Model Using Just MLPs</i>		
Number of Unique Families to Classify	Accuracy Per Experiment (%)	Overall Average Accuracy (%)
5	81.95	83.56
	84.08	
	82.41	
	85.14	
	84.21	
10	56.31	57.50
	56.81	
	59.40	
	61.50	
	53.48	
15	49.27	51.22
	53.18	
	54.82	
	54.54	
	44.31	
20	53.83	50.48
	45.68	
	52.92	
	46.87	
	53.08	

Table 8: Experimental results for the model using just MLPs.

4.3 LSTM Model With Embedding

In this model, we add an embedding layer to our basic LSTM, as illustrated in Figure 14. Note that the embedding layer is between the input and LSTM layer. The experimental results are shown in Table 10. With the addition of this layer, we see a significant improvement in the accuracy, with an average result of 74.66% with five families, but the accuracy drops dramatically when 10 or more families are considered.

4.4 BiLSTM Model With Embedding

The structural layout of this model shown in Figure 15. The only difference from our previous model is that the uni-directional LSTM layer has been replaced with a biLSTM layer. The experimental results for this model are given in Table 10. From the results, we can see that a biLSTM is far more powerful than an LSTM

<i>Model Without Embedding</i>		
Number of Unique Families to Classify	Accuracy Per Experiment (%)	Overall Average Accuracy (%)
5	63.91	55.73
	48.44	
	63.65	
	42.94	
	60.73	
10	40.50	39.28
	36.96	
	41.46	
	43.75	
	33.71	
15	32.65	34.47
	35.56	
	32.34	
	35.06	
	36.46	
20	34.25	30.55
	27.74	
	30.42	
	30.45	
	29.88	

Table 9: Experimental results for the model without embedding.

Figure 14: Structural layout of the LSTM model with embedding.

in this context, as the accuracy has improved significantly. In fact, the accuracy when classifying 20 families with this biLSTM model is nearly as good as the 5-family accuracy for the previous model.

<i>Model With Embedding</i>		
Number of Unique Families to Classify	Accuracy Per Experiment (%)	Overall Average Accuracy (%)
5	76.09	74.66
	73.64	
	73.17	
	76.90	
	73.51	
10	54.46	54.89
	56.96	
	55.67	
	54.71	
	52.90	
15	54.28	53.36
	51.97	
	50.28	
	53.22	
	57.03	
20	51.11	49.66
	52.12	
	51.60	
	45.82	
	47.65	

Table 10: Experimental results for the model with embedding.

Figure 15: Structural layout of the LSTM model with embedding and bidirectional LSTM.

4.5 Embedding, BiLSTM and CNN Model

The structural layout of this model appears Figure 16. Note that this model includes all of the layers as the previous model with the addition of a one-dimension

<i>Model With Embedding and Bidirectional LSTM</i>		
Number of Unique Families to Classify	Accuracy Per Experiment (%)	Overall Average Accuracy (%)
5	89.47	89.66
	90.83	
	89.95	
	85.94	
	92.12	
10	79.58	79.30
	79.54	
	78.13	
	78.79	
	80.46	
15	76.13	75.50
	76.13	
	76.66	
	76.28	
	72.31	
20	73.71	73.36
	74.74	
	69.53	
	74.10	
	74.72	

Table 11: Experimental results for the model with embedding and bidirectional LSTM.

convolutional layer and a max pooling layer. The experimental results for this case are shown in Table 12. The addition of these CNN layers improves accuracy, and the improvement is most significant as more families are considered—even for 20 families, we obtained a very respectable 81.06% average accuracy

4.6 Comparison of Results

A bar graph of the average accuracies for each model is shown in Figure 17. As noted above, the MLP only model performs remarkably well for five families. It even outperforms the LSTM model with embedding. For more than five families, however, performance decreases rapidly.

The basic LSTM model also performs poorly, even with five families. This suggests using only integer representation of opcode mnemonics is not enough to accurately

Figure 16: Structural layout of the model incorporating biLSTMs, embedding and a CNN layer where the 1DConv layer is a one dimensional convolutional layer and the MaxPool1D is a one dimensional max pooling layer.

Figure 17: Comparison of the average evaluation accuracy values for the four models used in this experiment for classifying malware families.

<i>Model With Embedding, BiLSTM and CNN</i>		
Number of Unique Families to Classify	Accuracy Per Experiment (%)	Overall Average Accuracy (%)
5	93.00	94.32
	96.33	
	92.73	
	94.70	
	94.32	
10	90.42	87.38
	90.29	
	81.29	
	89.58	
	85.29	
15	87.69	86.91
	87.56	
	82.59	
	87.31	
	89.41	
20	83.29	81.06
	76.34	
	80.60	
	82.18	
	82.88	

Table 12: Experimental results for the model with embedding, biLSTM and CNN.

classify malware. More layers in the model is needed.

The addition of an embedding layer dramatically increases the accuracy. This is not surprising given that previous work has shown that embedding layers can greatly improve the accuracy of machine learning models when applied to opcode sequences [35].

The substitution of a uni-directional LSTM layer with a bidirectional LSTM layer improves upon the results even further. It seems that the largest increase in accuracy between the embedding models with a uni-directional LSTM layer and the embedding models with a biLSTM layer occurs when classifying more than five families. This model performs quite well even with the most difficult task of classifying 20 families where the average accuracy is slightly less than 75%. BiLSTMs and word embedding

are often used together in NLP applications. However, their use in malware research appears to be very uncommon to this point in time. Our models indicate that there is much to be gained by considering both the forward and backward opcode sequence. Perhaps the models are able to better distinguish families based on which patterns appear more frequently by considering both the forward and backward sequences.

Finally, the addition of a one-dimensional CNN layer to the biLSTM and embedding layers gives the best performance among the five models studied in this research. Compared to the model without a CNN layer, the addition of this layer seems to have greater impact to performance when classifying more than five families. At an average of 81 percent for classifying 20 families, this model performs remarkably well. A possible explanation for why this model performs so well is that in addition to the benefits that come from having an embedding and biLSTM layers, a CNN layer helps the model by providing a different perspective on the opcode sequences. Specifically, the CNNs focus on local structure whereas the biLSTMs are focused on overall characteristics. The interplay between local and global structure has the potential to provide the best of both, which we have combined together into a single model. The addition of a max pooling layer serves to reduce noise and help the model focus on just the most important features further highlighting the crucial aspects of the local structure that the CNN highlights.

Confusion matrices for each model appear in the Appendix in Figure A.18 through Figure A.22. These matrices show how often families are classified incorrectly and show precisely where these misclassifications occur. For example, considering our best model, we can see that it was able to classify 10 families with an accuracy of over 90%. In contrast, three families are badly misclassified, namely, Obfuscator, Agent and Rbot with only 44%, 31% and 61% classified correctly, respectively.

There are a few key takeaways from these confusion matrices. First, we that there

were four families that were consistently classified well even for the worst performing model. These families are Winwebsec, Bho, Adload and Startpage. In the case of Winwebsec, we had the most number of samples from this family from Table 1. We can conclude that the high number of samples allowed each model to learn this family well contributing to its high classification rate. Since we did not have as many samples for the other families, however, we can conclude that there were perhaps a unique patterns or characteristics in the sequence of opcodes that allowed the models to learn each of these families well.

Secondly, from above, we can see that despite the high accuracy of our best performing model, Obfuscator, Agent and Rbot were badly classified. For Obfuscator and Rbot, we see a relatively high misclassification rate for the other family. The model misclassified Obfuscator as Rbot 16% of the time while Rbot was misclassified as Obfuscator 13% of the time. Perhaps there is too much similarity in the opcode sequences between these two families for the model to tell them apart. With only 817 Rbot samples, we can surmise that perhaps with more Rbot samples, the model can learn to better distinguish these two families.

We can also see that Agent was misclassified as Startpage 29% of the time. However, Startpage was only misclassified as Agent 2% of the time. As with Rbot, Agent only had 817 samples. Perhaps more Agent samples will allow for a higher classification rate.

CHAPTER 5

Conclusion and Future Work

In this project, we found that malware classification using only MLPs is only feasible for a small number of families. However, for a greater number of families, we found that long-short term memory (LSTM) models out performed MLP models. Although using just a single LSTM layer alone yields poor results, we found that incorporating techniques from natural language processing (NLP), specifically word embedding and bidirectional LSTMs (biLSTM) greatly improves performance. We also discovered that that we could obtain even better performance by including a convolutional neural network (CNN) layer in our model. Our best model was able to classify samples from 20 different malware families with an average accuracy in excess of 81%. We conjecture that the interplay between the long-term memory of the biLSTM and the local structure found by the CNN are the key to obtaining this strong performance.

For future work, more classification experiments using more samples of families that performed poorly on our best models would be of interest. This would reveal if the cause of high misclassification was the result of having a low number of samples or misclassification resulted from some inherent opcode sequence structure that the model was not able to distinguish. More can be done into investigating why applying NLP techniques are so effective in classifying malware. The addition of an embedding layer, greatly improved our model's overall accuracy. Other techniques can be considered. For example, we might apply principle component analysis (PCA) to reduce the dimensionality of the weights obtained from the embedding layer. This could reveal how each opcode is being represented in vector space. Additionally, experiments involving different word embedding algorithms (e.g., GloVe) would be worthwhile. Another area for possible work would be to create a model using only

MLPs and CNNs and compare the synergy between this model with the synergy between our biLSTM and CNN model. Finally, further research into the possible benefits of combining LSTMs and CNNs in this problem domain would be of great interest.

LIST OF REFERENCES

- [1] “Recurrent neural network RNN,” <https://docs.paperspace.com/machine-learning/wiki/recurrent-neural-network-rnn>.
- [2] D. Britz, “Recurrent neural networks tutorial, part 1 – introduction to rnns,” <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>, 2015.
- [3] C. Olah, “Understanding LSTM networks,” <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>, 2015.
- [4] Z. Cui, R. Ke, Z. Pu, and Y. Wang, “Stacked bidirectional and unidirectional lstm recurrent neural network for forecasting network-wide traffic state with missing values,” *Transportation Research Part C: Emerging Technologies*, vol. 118, p. 102674, 09 2020.
- [5] H. Gautam, “Word embedding: Basics,” <https://medium.com/@hari4om/word-embedding-d816f643140>, 2020.
- [6] R. Tahir, “A study on malware and malware detection techniques,” *International Journal of Education and Management Engineering*, vol. 8, no. 2, pp. 20–30, 2018.
- [7] S. Choudhary and A. Sharma, “Malware detection & classification using machine learning,” in *2020 International Conference on Emerging Trends in Communication, Control and Computing*, ser. ICONC3, 2020, pp. 1–4.
- [8] P. Prajapati and M. Stamp, “An empirical analysis of image-based learning techniques for malware classification,” in *Malware Analysis using Artificial Intelligence and Deep Learning*, M. Stamp, M. Alazab, and A. Shalaginov, Eds. Springer, 2021.
- [9] O. Williams, “The WannaCry ransomware attack left the NHS with a £73m IT bill,” <https://tech.newstatesman.com/security/cost-wannacry-ransomware-attack-nhs>, 2018.
- [10] A. Mujumdar, G. Masiwal, and D. B. Meshram, “Analysis of signature-based and behavior-based anti-malware approaches,” *International Journal of Advanced Research in Computer Engineering and Technology*, vol. 2, no. 6, 2013.
- [11] M. Sewak, S. K. Sahay, and H. Rathore, “Comparison of deep learning and the classical machine learning algorithm for the malware detection,” in *19th*

- IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, ser. SNPD, 2018, pp. 293--296.
- [12] M. Jain, W. Andreopoulos, and M. Stamp, "Convolutional neural networks and extreme learning machines for malware classification," *Journal of Computer Virology and Hacking Techniques*, vol. 16, no. 3, pp. 229--244, 2020.
 - [13] R. Lu, "Malware detection with LSTM using opcode language," <https://arxiv.org/abs/1906.04593>, 2019.
 - [14] B. Athiwaratkun and J. W. Stokes, "Malware classification with LSTM and GRU language models and a character-level cnn," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing*, ser. ICASSP, 2017, pp. 2482--2486.
 - [15] J. Zhang, "Deepmal: A CNN-LSTM model for malware detection based on dynamic semantic behaviours," in *2020 International Conference on Computer Information and Big Data Applications*, ser. CIBDA, 2020, pp. 313--316.
 - [16] P. Mishra, K. Khurana, S. Gupta, and M. K. Sharma, "Vmanalyzer: Malware semantic analysis using integrated CNN and bi-directional LSTM for detecting VM-level attacks in cloud," in *2019 Twelfth International Conference on Contemporary Computing*, ser. IC3, 2019, pp. 1--6.
 - [17] S. Gupta, P. Bansal, and S. Kumar, "Ulbp-rf: A hybrid approach for malware image classification," in *2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, 2018, pp. 115--119.
 - [18] G. Mahajan, B. Saini, and S. Anand, "Malware classification using machine learning algorithms and tools," in *2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*, 2019, pp. 1--8.
 - [19] M. Stamp, *Introduction to Machine Learning with Applications in Information Security*, 1st ed. Chapman & Hall/CRC, 2017.
 - [20] T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur, "Extensions of recurrent neural network language model," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing*, ser. ICASSP, 2011, pp. 5528--5531.
 - [21] N. Chowdhury and M. A. kashem, "A comparative analysis of feed-forward neural network recurrent neural network to detect intrusion," in *2008 International Conference on Electrical and Computer Engineering*, 2008, pp. 488--492.

- [22] D. Britz, “Recurrent neural networks tutorial, introduction,” <https://www.kdnuggets.com/2015/10/recurrent-neural-networks-tutorial.html>, 2015.
- [23] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in *International Conference on Machine Learning*, 2013, pp. 1310–1318.
- [24] J. Brownlee, “A gentle introduction to exploding gradients in neural networks,” <https://machinelearningmastery.com/exploding-gradients-in-neural-networks/>, 2019.
- [25] D. Thakur, “LSTM and its equations,” <https://medium.com/@divyanshu132/lstm-and-its-equations-5ee9246d04af>, 2018.
- [26] “Keras: Losses,” <https://keras.io/api/losses/>, 2020.
- [27] C. Kouridi, “Deriving the backpropagation equations for a LSTM,” <https://christinakouridi.blog/2019/06/19/backpropagation-lstm/>, 2019.
- [28] Z. Cui, R. Ke, Z. Pu, and Y. Wang, “Deep bidirectional and unidirectional LSTM recurrent neural network for network-wide traffic speed prediction,” <https://arxiv.org/abs/1801.02143>, 2018.
- [29] N. Tavakoli, “Modeling genome data using bidirectional LSTM,” in *2019 IEEE 43rd Annual Computer Software and Applications Conference*, ser. COMPSAC, vol. 2. IEEE, 2019, pp. 183–188.
- [30] D. Karani, “Introduction to word embedding and word2vec,” <https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa>, 2018.
- [31] J. Brownlee, “What are word embeddings for text?” <https://machinelearningmastery.com/what-are-word-embeddings/>, 2019.
- [32] M. Stamp, “Alphabet soup of deep learning topics,” <https://www.cs.sjsu.edu/~stamp/RUA/alpha.pdf>, 2019.
- [33] “Tensorflow core v.2.3.0 api,” https://www.tensorflow.org/api_docs/python/tf, 2020.
- [34] A. Nappa, M. Z. Rafique, and J. Caballero, “The MALICIA dataset: Identification and analysis of drive-by download operations,” *International Journal of Information Security*, vol. 14, no. 1, pp. 15–33, 2015.
- [35] A. Chandak, W. Lee, and M. Stamp, “A comparison of word2vec, hmm2vec, and pca2vec for malware classification,” in *Malware Analysis using Artificial Intelligence and Deep Learning*, M. Stamp, M. Alazab, and A. Shalaginov, Eds. Springer, 2021.

APPENDIX

Confusion Matrices

Figure A.18: Confusion matrix for the model using just MLPs only.

Figure A.19: Confusion matrix for the model without embedding.

Figure A.20: Confusion matrix for the model with an embedding layer.

Figure A.21: Confusion matrix for the model with embedding and biLSTM layers.

Figure A.22: Confusion matrix for the model with embedding, biLSTM and CNN and max pooling layers.