Chapter 6: Detection of Malicious Webpages Using Deep Learning: Unstructured Data

6.1 Background

6.1.1 Inspiration

As seen in the previous chapter, the ever-increasing and evolving web platforms pose a daunting task for the security experts trying to outsmart the hackers and prevent any web-based attack. But, with 1.7 billion websites active on the Internet and hundreds getting added each minute, this is a complex and challenging task [151]. In the last chapter, we presented a deep learning model to detect malicious webpages using structured web data. This technique may face scalability challenges with the ever-evolving nature of new websites. To overcome such limitations in detecting malicious webpages, we have developed another deep learning model which uses unstructured web content.

We hypothesize that using unstructured data will allow deep learning models to scale better and help in unearthing new patterns that could not have been discovered using structured data.

Deep learning and Natural Language Processing (NLP) research have reported significant developments in the last decade. We attempt to leverage this potent combination to detect malicious websites more efficiently and effectively and handle ever-evolving web attacks better.

6.1.2 Webpages Analysis: Structured vs Unstructured Data

This chapter uses unstructured data as input to the deep learning models, extracting relevant information directly from raw web content, including JavaScript. The choice of unstructured web content over structured data as an input was discussed in the previous chapter and needs to be delved further into before we proceed ahead. Deep learning models have this unique ability to learn from either structured or unstructured data. However, while training unstructured data, we need to process it into coded vectors. On the other hand, structured data requires limited processing while training. Thus, unstructured data may be computationally more expensive to use vis-à-vis structured data. However, it has been observed that unstructured data in deep learning can unearth new patterns that complement the patterns obtained by analyzing structured data. With unstructured data, we get the added advantage of scalability (simplicity and modularity of the design facilitate scalability) and the ability to handle evolving web content (as data is not tied to any specific feature set, it can manage any new textual content). Not only this, our results in this chapter demonstrate that using unstructured data, we have been able to achieve better accuracy, precision, and recall as compared to structured data.

6.1.3 Overview

We propose a hybrid two-stage deep learning model. The first stage (Stage-I), which is unsupervised, comprises three layers of LSTM blocks (Autoencoders) for encoding the text. This is followed by the second stage (Stage-II), which is supervised, comprising four hidden layers of feed-forward supervised training. The output of three unsupervised layers in Stage-I is a fixed 20-dimensional vector that feeds into the Stage-II of supervised learning. In Stage-II, the four dense layers feed into the output layer, which gives the probability of webpage being malicious or benign. The proposed hybrid deep learning model could achieve a high classification accuracy of 99.89% with very low False Positives (FP) and False Negatives (FN) (specify the FP and FN rates also). The contribution of this work can be summarized as:

- Hybrid deep learning model for identifying malicious webpages with high accuracy, precision, and recall.

- The capability of detecting zero-day web attacks. In zero-day attacks, the vulnerabilities that have not yet been made public are utilized by hackers to penetrate systems. Security systems designed using deep learning detect malicious activities based on patterns. Thus they can detect zero attacks whereas, signature-based systems fail to do so.

- Scalability and ability to handle evolving content on the Internet.

The rest of the chapter is structured as follows:

Section 6.2 discusses related work. Section 6.3 describes the hybrid deep learning model. Section 6.4 analyzes the results. Section 6.5 explores the scope for future work, and Section 6.6 concludes with a discussion on the model's capabilities.

6.2 Related Work

The conventional ML algorithms used in Chapter 4 could not achieve accuracy higher than 99% and suffered from high false positives and negatives. The DNN based approach using structured data proposed in Chapter 5 improved the classification metrics significantly; however, it still left scope for further improvements, especially with regards precision and F1-score. Apart from the works carried out in the previous chapters and discussed before in the thesis, few more research papers need our consideration. Vinaya et al. extracted features for classifying malicious URLs using CNN, LSTM, and RNN [153]. While they explored feature extraction techniques for such a task, they did not provide an end-to-end solution. Wang et al. proposed an algorithm for expressing the similarity of web content with malicious pages using LSTM [154]. However, their model underperformed on recall and precision metrics. Shrivastava et al. used deep learning for web page classification; their model suffers from complexity, high FP, and FN [155]. It is evident from the literature that the use of unstructured data for webpage classification lies largely unexplored, and it is this gap which we intend to plug. The model proposed in the chapter also attempts to overcome the limitations of existing models while bridging the identified gap.

Before we present the proposed hybrid deep learning model, it is imperative to highlight advances in text analytics and NLP, which have helped our hybrid model achieve the desired accuracy. Contextualized text representation has scaled new heights recently. Using unsupervised deep learning, it has become possible to encode text effectively into vectors representing the context well. Accurate deep learning models can be built using such well-represented vectors, which can then be used for downstream tasks.

These models have been used extensively in review, feedback, sentiment analysis, etc. ELMo [156], Transformers [157], BERT [158], and Universal Sentence Encoder [159] are some of the recent and popular text embedding models. ELMo stands for 'Embedding from Language Models'. It is different from traditional word embedding, in which a token is assigned a representation based on the entire sentence. Thus, it gives a contextual representation of each word. It uses bidirectional LSTM trained on a large corpus of text. Transformers are a non-recurrent sequence-to-sequence encoder-decoder model that uses the attention mechanism. BERT (Bidirectional Encoder Representation from Transformers) gives a bidirectional representation of text, using both right and left context. Universal Sentence Encoder provides a mechanism for encoding sentences into embedding vectors of fixed length. For Stage-I unsupervised pre-training, we have developed a customized text encoder using the best capabilities of both ELMo and Universal Sentence Encoder.

6.3 Hybrid Deep Learning Model

The model proposed in this chapter uses LSTM (Long Short Term Memory is a type of RNN that connects previous information with the present data input) based Autoencoder (AE are used for unsupervised learning of codes from data, and it helps to represent data in reduced dimensions) for Stage-I unsupervised encoding, and Deep Neural Network (DNN) for Stage-II supervised classification. In this section, we propose and describe this model.

6.3.1 Hybrid Deep Learning Model

The design is a two-stage model, with stage-I being an unsupervised stacked LSTM autoencoder for encoding web content and JavaScript. This autoencoder is trained with random web content and JavaScript to encode data into fixed numerical codes (20-dimension vector). Stage-II of the model is a supervised DNN to classify webpages as malicious or benign. The trained encoder from Stage-I is utilized in Stage-II for encoding inputs (webpages) to the DNN.

The design of the hybrid deep model for classification is illustrated in **Figure 6.1**.

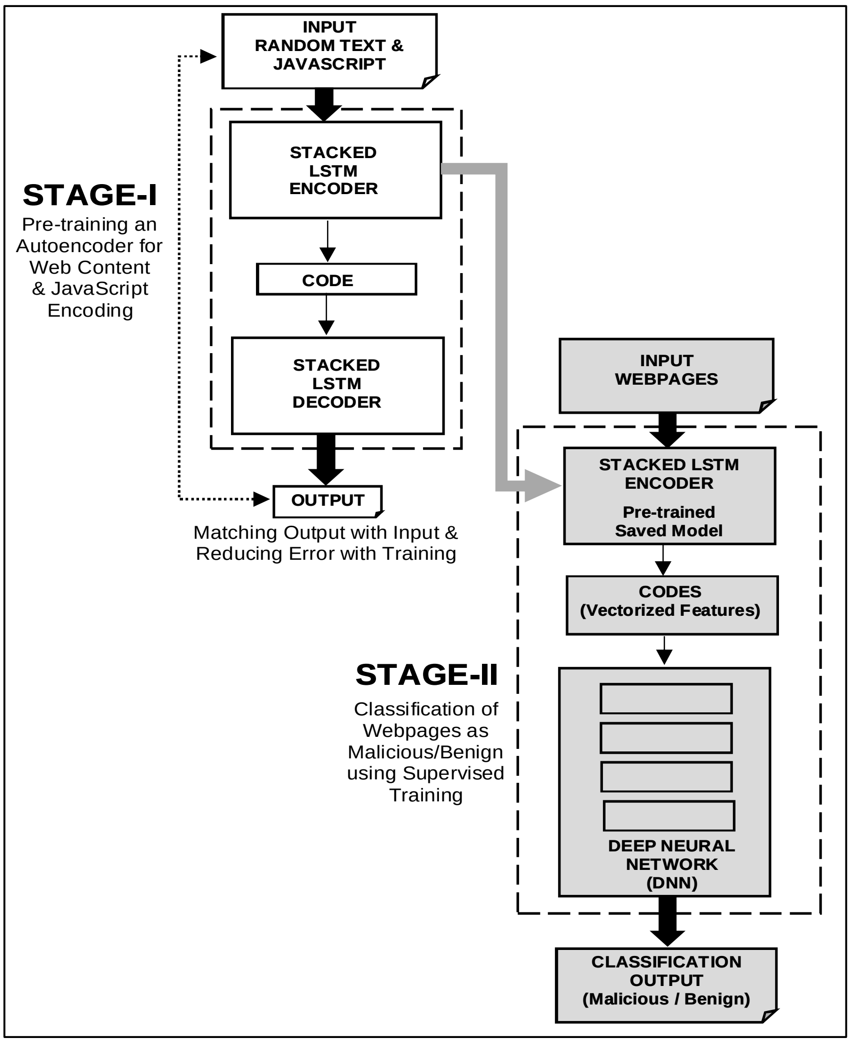

INPUT
RANDOM TEXT & JAVASCRIPT
STAGE-I
Pre-training an Autoencoder for Web Content & JavaScript Encoding
STACKED LSTM ENCODER
CODE
STACKED LSTM DECODER
OUTPUT
Matching Output with Input & Reducing Error with Training
INPUT WEBPAGES
STACKED LSTM ENCODER
Pre-trained Saved Model
CODES (Vectorized Features)
STAGE-II
Classification of Webpages as Malicious/Benign using Supervised Training
DEEP NEURAL NETWORK (DNN)
CLASSIFICATION OUTPUT (Malicious / Benign)

Figure 6.1: Hybrid Deep Learning Model

6.3.2 Stage-I: Unsupervised LSTM Autoencoder

Before understanding the Stage-I process, it is imperative to understand the need for a text encoder. Before the data is fed into the DNN model, it has to be represented as a vector. The accuracy of encoding textual data by encoder will determine the efficacy of the overall ML process. Hence, in NLP, it is essential to get a precise representation of text in numerical codes. Another question that arises is the necessity of pre-training a new encoder model for this work and why an existing generic encoder was not used (Generic encoder models can be found on sites like TensorFlow Hub [160]). Existing encoder models from previous research were trained on generic text, and they could not achieve the desired accuracy for our application. Web content, especially JavaScript, has a unique vocabulary not found in generic text encoders, and therefore, we designed and trained our bespoke encoder.

6.3.2.1 Dataset for Unsupervised Training

Before understanding the autoencoder design, let us understand its intended input which is a dataset of random web content and JavaScript from the Internet, consisting of 1 million samples. This dataset is different from the dataset used in Stage-II and is used exclusively for unsupervised pre-training of the autoencoder. This dataset is published online on Kaggle to facilitate further research [161].

6.3.2.2 Autoencoder Design

In Stage-I, a stacked LSTM autoencoder is trained to encode web content, and JavaScript found on webpages into a fixed 20-dimensional vector, as shown in **Figure 6.2**. Three layers of LSTM are used in the encoder and three in the decoder. The autoencoder is trained to reproduce its own input, and its process can be represented by Equations (6.1) and (6.2) given below.

$$\emptyset\ (Encoding\ Function): X -> \ Y \tag{6.1}$$

$$\theta\ (Decoding\ Function): Y -> \ X \tag{6.2}$$

The autoencoder is trained to reduce the difference between output and input as represented by Equation (6.3). The training is carried out using backpropagation and gradient descent algorithms. Backpropagation is an algorithm for computing error gradients using the chain rule of differentiation. At the same time, gradient descent is an optimization algorithm that minimizes loss function by iteratively moving in the direction of steepest descent.

$$Training:\ \underset{\emptyset,\theta}{argmin} \|X - \ (\emptyset o \theta)X\|^2 \tag{6.3}$$

Once the autoencoder is trained on the dataset, the encoder is detached from the decoder and is saved separately for Stage-II input encoding. This encoder model, which is pre-trained and saved, can accept a variable-length text (containing both web content and JavaScript code) and produce a reduced dimensional (summarized) 20-dimensional vector.

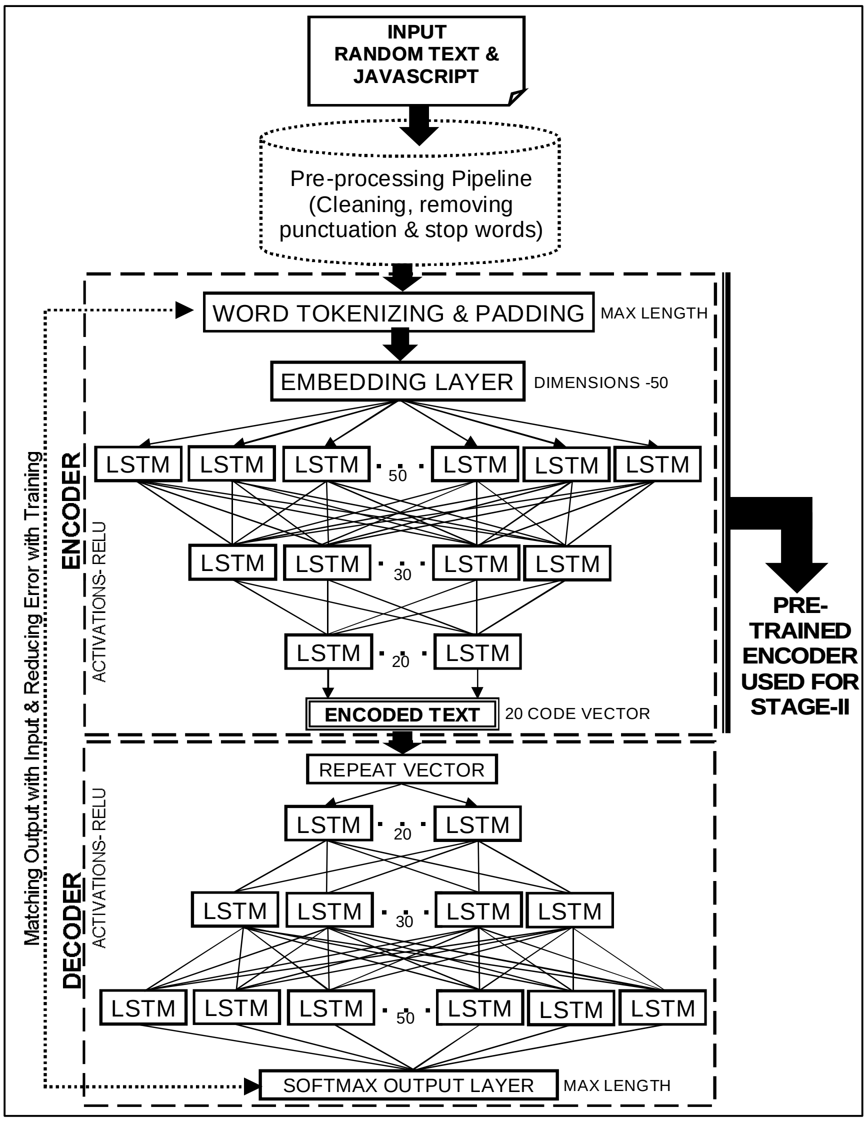

INPUT RANDOM TEXT & JAVASCRIPT
Pre-processing Pipeline (Cleaning, removing punctuation & stop words)
WORD TOKENIZING & PADDING
MAX LENGTH
EMBEDDING LAYER
DIMENSIONS -50
ENCODER
ACTIVATIONS- RELU
LSTM
50
30
20
ENCODED TEXT
20 CODE VECTOR
PRE-TRAINED ENCODER USED FOR STAGE-II
REPEAT VECTOR
DECODER
SOFTMAX OUTPUT LAYER
Matching Output with Input & Reducing Error with Training

Figure 6.2: Stage-I: Unsupervised Pre-training AE for Web Content & JS Encoding

6.3.3 Stage-II: Supervised Training

6.3.3.1 Dataset for Supervised Training

Stage-II model accepts unstructured web content as input. Dataset of unstructured web content was prepared by crawling and scraping websites on the Internet using our bespoke web crawler, MalCrawler [67]. Collected raw web content was cleaned to remove stop words, punctuations, HTML tags, etc., while JavaScript code in the web content was retained. The obfuscated JavaScript code, if any, was de-obfuscated using 'JavaScript Auto De-Obfuscator' (JSADO) [74] (De-obfuscation code has been published online [162]. JavaScript Obfuscation is the process of scrambling/ encrypting code with an aim to make it difficult for human understanding. Generally, it is used to protect intellectual property in software codes. However, occasionally, it is used by hackers to hide malwares.). The 'label', which gives the class of the webpage as malicious or benign, was added in the dataset using Google Safe Browsing API [61]. The cleaned dataset, comprising 1.564 million records, was packed into a CSV file with features as shown in **Table 6.1** (refer to F1 and F10 attributes in **Table 3.2** of Chapter 3).

Table 6.1: Attributes in Dataset File (DNN with Unstructured Data)

Attribute	Attribute Description
url	URL of the webpage {Datatype- Text}
content	Cleaned web content and JavaScript (Obfuscated JavaScript is de-obfuscated) {Datatype- Text}
label	Classification label categorizing webpage as malicious or benign {Datatype- Categorical}

Further details on the dataset, pre-processing, and visualization have been given in Chapter 3 and Appendix A of the thesis.

6.3.3.2 DNN Design

Stage-II carries out supervised training for webpage classification. The encoder model, which was pre-trained in Stage-I, is utilized in Stage-II for encoding webpages (input) using Transfer Learning (it is a process in which a trained and saved model is used for a second task). The output of this encoder is a fixed 20 code vector, which is further fed to four dense layers of DNN, as shown in **Figure 6.3**. Two Dropout layers [140] (it is a technique of randomly dropping nodes from training) were added in the model for regularization (it is the process of reducing overfitting). This DNN is a feed-forward network trained using 'Gradient Descent' with 'Binary Cross Entropy' as cost function (this function measures the performance of the ML model). The gradient descent algorithm works in two directions. In the forward pass, it computes errors, and in the backward pass, it computes error gradients using backpropagation and amends parameters accordingly [164]. RELU activation was used for all neurons in all hidden layers. In contrast, Sigmoid activation was used for the final DNN output '$\boldsymbol{y}$' ($y=0$ for malicious and $y=1$ for benign webpages).

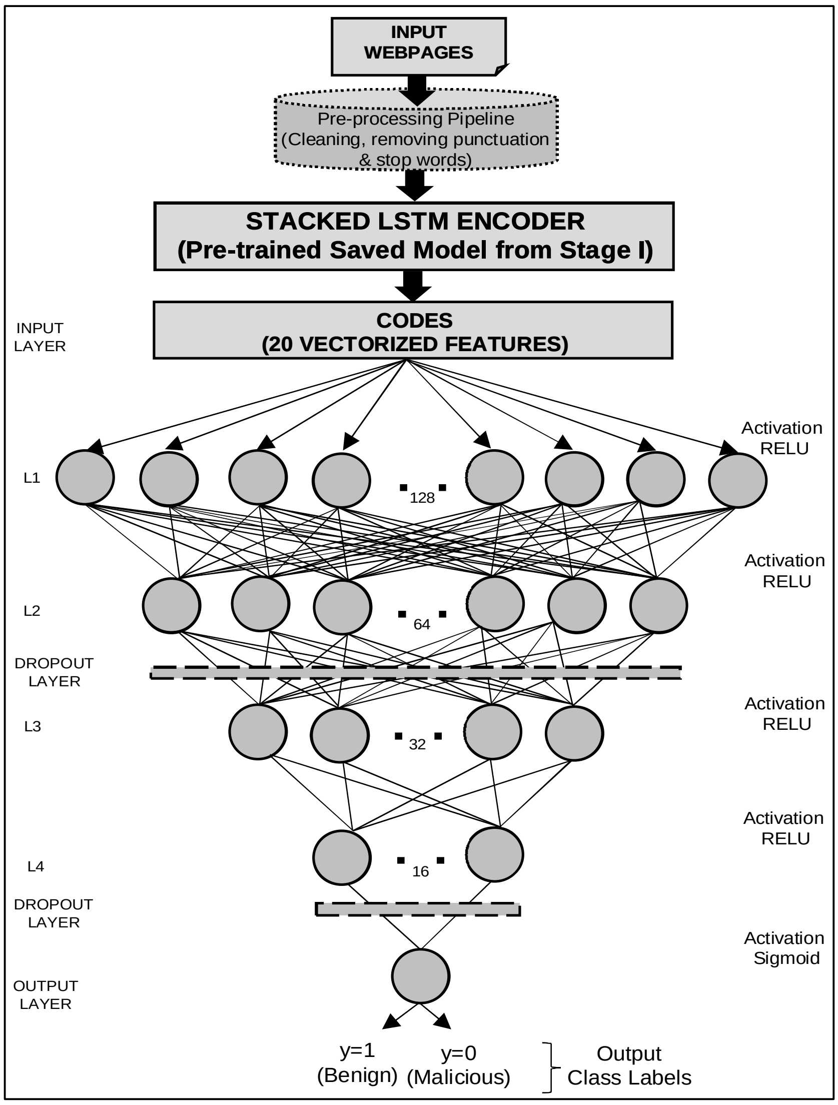


Figure 6.3: Stage-II: DNN Model for Classification

The process of training using gradient descent has already been described schematically in **Figure 5.2** in the previous chapter. Input to DNN model is represented by vector $\boldsymbol{X}= [x_1, x_2, \ldots, x_n]$. From the second layer to the fourth layer, inputs to neurons are activation outputs from the previous layer, e.g., inputs to l^{th} layer is given by vector $\boldsymbol{a^{l-1}} = [a_1^{l-1}, a_2^{l-1}, \ldots, a_i^{l-1}]$. Weight vector to j^{th} neuron in l^{th} layer is represented by $\boldsymbol{w_j^l} = [w_{j1}^l, w_{j2}^l, \ldots, w_{ji}^l]$, while bias for j^{th} neuron in l^{th} layer is b_j^l. As shown by **Figure 5.2** in the last chapter, Affine function z_j^l gives the weighted sum of inputs coming to the neuron.

$$z_j^l = \sum_i w_{ji}^l a_i^{l-1} + b_j \tag{6.4}$$

For input layer, the equation is represented as,

$$z_j^1 = \sum\nolimits_n w_i^1 x_i + b_j \tag{6.5}$$

Affine function feeds into the activation function, which for layer one to four ($l1 - l4$) is a RELU as represented by Equation (6.6), and for output layer ($l5$) is a Sigmoid depicted by Equation (6.7).

$$a_j^l = max\left(0, z_j^l\right) \tag{6.6}$$

$$y = a_j^l = \frac{1}{\left(1 + e^{-z_j^l}\right)} \tag{6.7}$$

During training, tuning of parameters **W** and **b** is carried out. In the forward pass, the error is computed by comparing predicted output y and target output t using binary cross entropy loss function E.

$$E = \frac{-1}{n}\sum_{c=1}^{n}[t_c \, log(y_c) + (1 - t_c) \, log(1 - y_c)] \tag{6.8}$$

Thereafter, backpropagation [165] is carried out, wherein error contribution of each parameter $w_{ji}^l$and b_j^l in the network towards total loss **E** is computed through gradients. Using the chain rule of differentiation, partial derivatives are calculated successively backward from output to input. In this model, error gradient with respect to w in last layer ($l5$) is,

$$\frac{\partial E}{\partial w_{kj}^5} = \frac{\partial E}{\partial y_c} \frac{\partial y_c}{\partial z_k^5} \frac{\partial z_k^5}{\partial w_{kj}^5} \tag{6.9}$$

Solving partial derivative using Equation (6.8),

$$\frac{\partial E}{\partial y_c} = \frac{-t_c}{y_c} + \frac{1 - t_c}{1 - y_c} = \frac{y_c - t_c}{y_c(1 - y_c)} \tag{6.10}$$

Solving partial derivative using Equation (6.7),

$$\frac{\partial y_c}{\partial z_k^5} = \frac{e^{-z_k^5}}{\left(1 + e^{-z_k^5}\right)^2} = y_c(1 - y_c) \tag{6.11}$$

Solving partial derivative using Equation (6.4),

$$\frac{\partial z_k^5}{\partial w_{kj}^5} = a_j^4 \tag{6.12}$$

Using Equations (6.9) to (6.12),

$$\frac{\partial E}{\partial w_{kj}^5} = (y_c - t_c)a_j^4 \tag{6.13}$$

Similarly, chain rule is used to compute error gradient with respect to b in last layer,

$$\frac{\partial E}{\partial b_k^5} = \frac{\partial E}{\partial y_c}\frac{\partial y_c}{\partial z_k^5}\frac{\partial z_k^5}{\partial b_k^5} \tag{6.14}$$

Computing partial derivative using Equation (6.4),

$$\frac{\partial z_k^5}{\partial b_k^5} = 1 \tag{6.15}$$

Using Equations (6.10), (6.11), (6.14) and (6.15),

$$\frac{\partial E}{\partial b_k^5} = (y_c - t_c) \tag{6.16}$$

Equations (6.13) and (6.16) give us error gradients with respect to parameters in last layer (only a single neuron exists in last layer, thus, $k = 1$). Computation of error gradients with respect to parameters in lower layers requires recursive application of chain rule as part of back propagation algorithm. For the third layer ($l4$), error gradient with respect to w is given as,

$$\frac{\partial E}{\partial w_{ji}^4} = \frac{\partial E}{\partial y_c}\frac{\partial y_c}{\partial z_k^5}\frac{\partial z_k^5}{\partial a_j^4}\frac{\partial a_j^4}{\partial z_j^4}\frac{\partial z_j^4}{\partial w_{ji}^4} \tag{6.17}$$

The first two partial derivatives have been computed in Equations (6.10) and (6.11), and the third is calculated using Equation (6.4),

$$\frac{\partial z_k^5}{\partial a_j^4} = w_{kj}^5 \tag{6.18}$$

Solving partial derivative of RELU activation in layer 3 using Equation (6.6),

$$\frac{\partial a_j^4}{\partial z_j^4} = \begin{cases} 1 \; when \; z_j^4 > 0 \\ 0 \; when \; z_j^4 \leq 0 \end{cases} \tag{6.19}$$

Using Equation (6.4),

$$\frac{\partial z_j^4}{\partial w_{ji}^4} = a_i^3 \tag{6.20}$$

Using Equations (6.10), (6.11) and (6.17) to (6.20),

$$\frac{\partial E}{\partial w_{ji}^4} = \begin{cases} (y_c - t_c) w_{kj}^5 a_i^3 & when\, z_j^4 > 0 \\ 0 & when\, z_j^4 \leq 0 \end{cases} \tag{6.21}$$

Similarly, error gradient with respect to b in the fourth layer is given by,

$$\frac{\partial E}{\partial b_j^4} = \frac{\partial E}{\partial y_c} \frac{\partial y_c}{\partial z_k^5} \frac{\partial z_k^5}{\partial a_j^4} \frac{\partial a_j^4}{\partial z_j^4} \frac{\partial z_j^4}{\partial b_j^4} \tag{6.22}$$

Using Equation (6.4),

$$\frac{\partial z_j^4}{\partial b_j^4} = 1 \tag{6.23}$$

Using Equations (6.10), (6.11), (6.17) to (6.19), (6.22) and (6.23),

$$\frac{\partial E}{\partial b_j^4} = \begin{cases} (y_c - t_c) w_{kj}^5 & when\, z_j^4 > 0 \\ 0 & when\, z_j^4 \leq 0 \end{cases} \tag{6.24}$$

Moving further backward, up the layers, gradients with respect to parameters of layer 3 are computed recursively based on derivatives of layer 4, and so on. Since layers 1 to 3 use RELU activation like layer 4, equations can be built up similarly as shown above and thus are not discussed further. This backpropagation, as described above, is carried out for all input samples. Thereafter, loss function **E** is minimized over all **n** input samples **X**, using a variant of gradient descent called 'Adam'. Adam algorithm uses partial derivatives computed during backpropagation to find out global minima of loss function ***E*** with respect to **W** and **b**. This successive tweaking of **W** and **b** leads to minimization of loss function over inputs **X**, expressed mathematically as,

$$argmin_{\boldsymbol{W},\boldsymbol{b}}\, \boldsymbol{E} \tag{6.25}$$

W and **b** are tweaked in batches using a learning rate $\boldsymbol{\alpha}$, which signifies step-size during descent. For this DNN model, $\alpha = 1e^{-3}$ was found suitable.

$$\boldsymbol{W} = \boldsymbol{W} - \alpha \frac{\boldsymbol{\partial E}}{\boldsymbol{\partial W}} \tag{6.26}$$

$$\boldsymbol{b} = \boldsymbol{b} - \alpha \frac{\boldsymbol{\partial E}}{\boldsymbol{\partial b}} \tag{6.27}$$

6.3.3.3 Class Imbalance in Data

The issue of class imbalance has already been discussed in the thesis in Chapter 3 (refer to **Figure 3.2**) and in section 5.3.5 of the previous chapter. So, like in the previous chapter, we have used modified weights for this DNN model to handle class imbalance.

$$wt_0 = \frac{total}{(neg \times 2)} \tag{6.28}$$

$$wt_1 = \frac{total}{(pos \times 2)} \tag{6.29}$$

{wt_0- Weight of Negative Class, wt_1- Weight of Positive Class, total- Total Number of Samples, neg- Number of Negative Samples, pos- Number of Positive Samples}

6.3.3.4 Convergence in Training

While training a large dataset with imbalanced classes, convergence time to reach minima of the cost function is high. To reduce convergence time, we have used 'Initial Bias' [166] based on class weights as given in Equation (6.30).

$$bias_0 = log_e\left(\frac{pos}{neg}\right) \tag{6.30}$$

Applying the above equation, initial loss decreased manifold and thus resulted in faster convergence.

6.3.3.5 Tuning of Hyperparameters

In deep learning, the correct selection of Hyperparameters (these are values that control the machine learning process) is essential to train an accurate model. These suitable values are generally arrived at by persistent trial and error. In our experimentation, we used the Grid Search [167] technique to find appropriate values of Hyperparameters.

6.3.4 Experimental Setup

The hybrid model has been implemented using Python libraries - TensorFlow [147] (it is an open-source ML platform in Python that Google released), Keras [148] (it is a deep learning library in Python), NumPy (it is a

Python library that provides functions for vector calculus) and Skikit-learn [168] (it is a ML library in Python and in this model it was used for tuning hyperparameters using grid search). For the generation of result graphs, TensorBoard (it provides visualization of TensorFlow results) and Seaborn (a data visualization library based on Matplotlib) were used. Code for Stage-II was written to run on GPU (it was used as computations involved matrix multiplications, which is more efficient on GPU vis-à-vis CPU). In contrast, for Stage-I, Tensor Processing Unit (TPU) [169]was used to meet large processing requirements of stacked LSTMs. TPUs are Application Specific Integrated Circuits (ASIC) developed by Google for high-speed deep learning. However, with minor modifications, this code can be run on either CPU, GPU, or TPU. Code for complete setup as listed below (written in separate Jupyter Notebooks) is published on Kaggle to facilitate reproducibility and further research.

- Code for Stage-I pre-training using stacked LSTM encoders [170].
- Code for Stage-II classification using DNN [171].
- Code for tuning of Hyperparameters [172].

Results of training, validation, and testing of this hybrid model are given in the subsequent section.

6.4 Results and Analysis

6.4.1 Stage-I (Pre-training of Autoencoder)

The Autoencoder in Stage-I was trained on a dataset comprising 1 million random samples of web content and JavaScript (Note: Dataset used in Stage-I for training the autoencoder differs from Stage-II). Autoencoder was trained to 100 Epochs, with successive reduction of loss as plotted in **Figure 6.4**. Once trained, the autoencoder could achieve an accuracy of 98.83%. After training, the encoder was detached and saved for Stage-II input encoding. This encoder outperformed other generic text encoders in encoding web content and JavaScript.

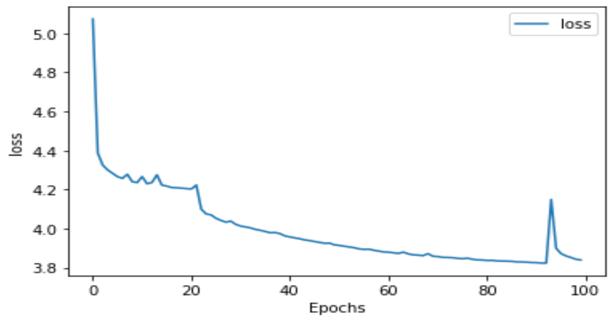


Figure 6.4: Loss for Stage-I Unsupervised Training

6.4.2 Stage-II (Classification of Webpages)

6.4.2.1 Training and Validation Results

Stage-II model was trained on 0.9 million samples, validated on 0.1 million samples, and evaluated over a test dataset of 0.564 million samples. The DNN network was trained over 100 epochs with early stopping (with patience set to 50). The accuracy over the epochs (during training and validation) is plotted in **Figure 6.5**. It may be noted that the 'Early Stopping Algorithm' stops the training early once metrics have stabilized (Patience decides the stabilization point), and it restores the best metrics reached before exiting the session.

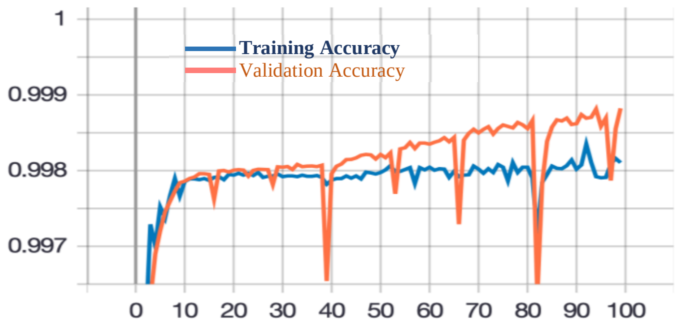


Figure 6.5: Training and Validation Accuracy

It can be seen from **Figure 6.5** that training accuracy lagged validation accuracy. This behavior is attributed to 20% dropout implemented for regularization during training (20% dropout will randomly switch off $\frac{1}{20^{th}}$ neurons, thereby reducing training performance). Since dropout is used only for training and not for validation and testing, training accuracy lags validation and test accuracy. Similarly, Binary Cross Entropy validation loss over the training Epochs is plotted in **Figure 6.6**.

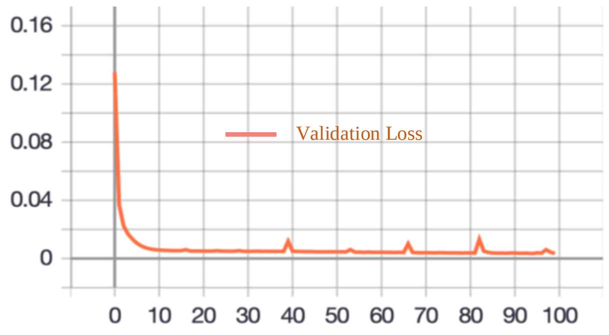


Figure 6.6: Validation Loss

The previous section had highlighted the role of initial bias in faster convergence during training (refer Equation (6.30)). The plot of training with and without initial bias is shown below in **Figure 6.7**. The plot clearly brings out that loss dropped rapidly with initial bias, and after that converged faster.

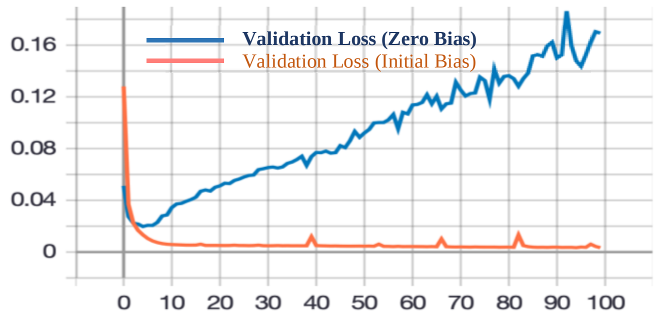


Figure 6.7: Impact of Initial Bias

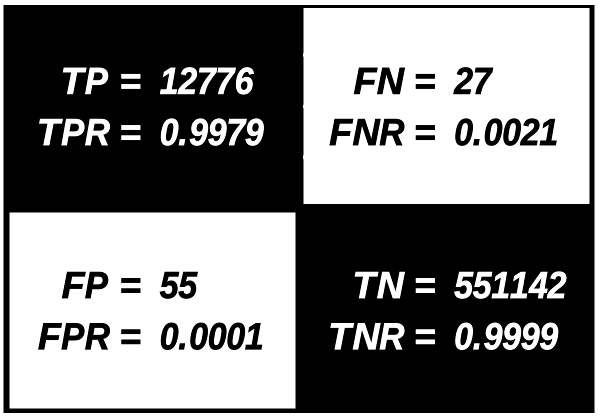
6.4.2.2 Test Evaluation Results

The trained model was evaluated on a test dataset that comprised 0.367 million samples. The result of this evaluation is shown below in **Table 6.2**. An accuracy of 99.89% during evaluation vindicates the effectiveness of this hybrid model in the classification of malicious webpages. This notion is further reinforced by high precision, recall and F1-score.

Table 6.2: Evaluation on Test Dataset (Hybrid DNN with Unstructured Data)

Metric	Value
Accuracy	0.9989
Recall (TPR, Sensitivity)	0.9979
Specificity (TNR, Selectivity)	0.9999
Precision (PPV)	0.9957
NPV	1.0000
F1 Score	0.9968
*Note: Positive class represents the 'Malicious label'.	

The meaning of metrics used in **Table 6.2** has already been given in **Table 4.4**. The confusion matrix is given in **Figure 6.8**. The matrix clearly highlights negligible FNR and FPR during the evaluation of the test dataset.



TP = 12776
TPR = 0.9979
	FN = 27
	FNR = 0.0021

FP = 55
FPR = 0.0001
	TN = 551142
	TNR = 0.9999

Figure 6.8: Confusion Matrix

AUC-ROC (Area Under Curve- Receiver Operating Characteristic) metric gives the probability of ranking a random positive sample vis-a-vis a random negative sample. The AUC graph is plotted in **Figure 6.9**. The high AUC value shows that this hybrid model can clearly distinguish the two classes with negligible errors.

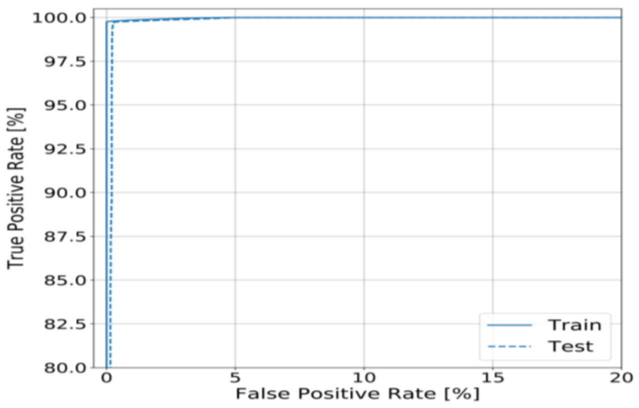


Figure 6.9: AUC-ROC Graph

6.4.3 Utilization of Computational Resources

Analysis of computational resources utilized is an important consideration factor in ML. As discussed earlier, the Stage-I model is designed using LSTMs; thus, its complexity is $O(n)$. For Stage-I, BERT [158] model was also considered based on its good performance in generic text encoders; however, it was not used due to its high computational requirements (complexity was $O(n^2)$) Stage-II model is a feed-forward DNN, whose complexity is $O(n)$. The complete hybrid model took 665.55 mins to train over 100 Epochs (312.25 mins for Stage-I and 353.30 mins for Stage-II). The test time per sample was 997 μs (Stage-I code was run on Kaggle with a TPU v3.8: 8 Core, 16 GB RAM. Stage II code was run on Kaggle Server with configuration: CPU- Intel Xenon 2.2Ghz, GPU- Nvida Tesla P100, RAM- 16GB.). However, if we consider pre-processing time per sample (e.g., time for preparing a vectorized sample from each webpage visited by the web browser), then the total time per sample is 1.3 ms (this is an approximate value and the total time will vary based on the size of webpage and network delays.).

6.4.4 Analysis and Inference

If we analyze the training and validation accuracy, it emerges that the model is adequately trained without overfitting. The model has given an accuracy of 99.89%, which surpasses all existing models for malicious webpage classification. High accuracy has been achieved while keeping very low FP, FN, and high precision and recall metrics. These values substantiate the capability of the proposed hybrid model in detecting malicious webpages precisely.

6.5 Future Work

With regards to scope for future work, the work presented in this chapter offers various options, as listed below:

- Stage-I model used a stacked LSTM network with fixed 20 code output. We had kept feature output limited to 20 to avoid overloading of computational resources. However, higher encoding outputs (64, 128, etc.) may be explored if adequate computational resources are available. Theoretically, higher code output should improve accuracy further. Also, few other designs apart from LSTM, like Transformers [157], BERT [158], etc., may be explored to improve results.
- This model can be deployed as a web browser plugin to detect malicious webpages. Also, 'Tensorflow.js' [150] may be used to train and run this model directly from the browser.

6.6 Deployability of Solutions Proposed

Solutions in part I and II of the thesis were proposed keeping in mind the current requirements in web security. These solutions have been designed and developed to be deployed without further significant development efforts.

MalCrawler: MalCrawler has been used extensively in the thesis for crawling and gathering webpages. MalCrawler, which is a Java-based application, has run for months on the Internet without interruptions. The crawler is ready to be deployed for any focused crawling task related to web security and can be utilized by Internet search and cybersecurity firms/organizations.

Trained ML Models for Web Security Predictions: The thesis has offered various ML models for webpage prediction, each with its strength and weakness. For e.g., DNN based model with structured data input proposed in Chapter 5 gives a good time response albeit with reduced accuracy. The hybrid DNN model presented in this chapter offers good prediction accuracy, although with reduced time response. These models can be chosen based on the deployment requirements and be integrated into existing solutions or deployed as standalone applications. For e.g., malicious webpage prediction models can be integrated into a browser using a plugin. Such a browser plugin with a trained ML model can be developed with a small development effort. Furthermore, these ML models can be used for creating newer solutions using the concept of Transfer Learning [228]. In Transfer Learning, a pre-trained model is used as the starting point for a new model.

6.7 Conclusions

In this chapter, we introduced a hybrid deep learning model to classify webpages as malicious or benign. The model used a two-stage hybrid design. Stage-I was a LSTM Autoencoder pre-trained to vectorize input web content and JavaScript into fixed 20-dimensional output. The trained encoder from Stage-I was used to encode webpages at the input of Stage-II. The stage-II model was a four-layer supervised DNN trained to classify webpages as benign or malicious. The results achieved in this research have outperformed previous studies in various metrics like accuracy, FP, FN, precision, and recall. Therefore, this solution is suitable for various web security requirements that demand satisfactory performance over standard evaluation metrics.