Chapter 5: Detection of Malicious Webpages Using Deep Learning: Structured Data

5.1 Background

In the last chapter, we discussed how malicious websites continue to be a threat on the Internet [125]. We also discussed how the detection of malicious websites evolved from static heuristics (signature-based detection [107][42]), dynamic heuristics (using honey clients, sandboxes, and browser emulators [126][50]), and then to ML. ML is the latest technology being explored for such web security tasks [127][72]. We had used conventional ML, wherein we used the C4.5, SVM, Naive Bayes and Random Forest classifiers with selected attributes to improve classification results. However, these conventional models had limitations regarding high false negatives (FN), low precision and F1-score, which were not suitable for commercial deployment by web security firms. So, we explored newer ML technologies that could surpass these results. In this chapter, we will explore deep learning models to further improve the classification results for the malicious webpage prediction problem.

In the past few years, deep learning has emerged as the most promising subfield of ML. It uses large neural networks to achieve good classification results in image and Natural Language Processing (NLP). We have used deep learning to overcome the limitations of previous ML approaches in webpage classification, including the one given in the last chapter. We have used a novel Deep Neural Network (DNN) model to detect and classify malicious webpages with better accuracy, precision, and recall. While deep learning can handle both structured and unstructured data, this chapter uses structured attributes to enable fast training and quick detection. This approach has given a high accuracy of 99.81% with very low FP and FN. While training a DNN model takes time, runtime on the test set is quick. The trained model takes a test sample (webpage) and classifies it in less than 264 μs in checking, including time for preprocessing the sample into a vector. This is an approximate time per

webpage without considering network delay. The exact time would vary based on the content of the webpage. Such high speed and accuracy make this technique suitable for deployment for web security solutions. The contribution of the work done in this chapter is summarized below:

- Deep learning based detection model for malicious webpages with high accuracy, precision, and recall.
- The model's fast detection capability enables its deployment on Web Browser platforms without deterioration of browsing experience.
- The capability of the proposed DNN model to detect Zero-Day attacks. This is achieved as the DNN model is able to detect more non-linear patterns compared to conventional ML techniques.

The remaining chapter is structured as follows. Related work is discussed in Section 5.2. Section 5.3 introduces the deep learning framework for malicious webpage detection. Section 5.4 discusses the results and analysis. Lastly, Section 5.5 concludes with a discussion on the utility of the model proposed and future work scope.

5.2 Related Work

Malicious web page detection approaches have evolved from static heuristics, dynamic honey client based detection to ML. Recently, with rapid advances in deep learning, it is being explored for solving web security tasks.

Earlier works using static heuristics include Cova [107] and Canali [42] et al., wherein they used signature-based detection techniques. Work utilizing dynamic heuristics include high interaction honey clients by Akiyama [128] and low interaction honey clients by Ikinci et al [50]. These approaches had limited capability of detecting new patterns.

Conventional ML approaches for malicious webpage detection have used classifiers like SVM, Random Forest, Decision Tree, etc. Eshete [127], Singh [72], Yoo [129] and Wang et al. [130] have used such techniques. However, their classification results could not surpass 99% accuracy and suffered from high false negatives or false positives.

While attempts have been made to use deep learning for malicious web page detection, research outcomes are confined mainly to restricted domains of Google labs [131] or cybersecurity and anti-virus firms with limited information shared in public. Apart from these organizational efforts, few research papers, as discussed below, have been published in this field, but they have been inadequate to address the problem statement holistically. Shrivastava et al. have used a deep learning framework for webpage classification [132]. However, the framework was complex and could not achieve satisfactory accuracy while keeping false positives and false negatives low. Compared to the model proposed in this chapter, their framework underperforms in all metrics, including time. Fang et al. proposed a deep learning based solution to detect cross side scripting (XSS) [152]; however, their solution remains confined to XSS attacks. Vinaya Kumar et al. have evaluated various LSTM, CNN, and RNN layers for feature extraction to classify malicious URLs [133]. Although, their work explored practical feature extraction techniques for such tasks, they failed to propose a suitable end-to-end solution for webpage classification. Wang et al. have proposed a LSTM bidirectional algorithm based on CNN and RNN for expressing the similarity of web content with a malicious page [134]. However, the model underperforms on precision and recall metrics.

Keeping related work in mind, the work presented in this chapter attempts to overcome existing limitations and gaps.

5.3 Deep Learning Framework for Detection of Malicious Webpages

This section proposes a deep learning framework for malicious webpage classification and describes its design and implementation.

5.3.1 Understanding Deep Learning

Deep learning is a subfield of ML that has gained prominence in the last few years. It uses layered neurons to learn complex non-linear patterns in the data. It can be used in both supervised or unsupervised settings. Further it can handle both structured and unstructured data [135]. Deep Neural Networks (DNN) carry out hierarchical learning, with lower layers learning low-level features and higher layers progressively learning high-level features from them

[136]. According to the Universal Approximation Theorem, feed forward deep learning models can represent a nonlinear relationship in dataset better than shallow neural networks and other ML classification algorithms [137]. Typical deep learning techniques include DNN, Deep Belief Networks (DBN), Convolution Neural Networks (CNN), Auto Encoders (AE), Recurrent Neural Networks (RNN), etc.

In this chapter we use DNN with structured data input to classify webpages as malicious or benign.

5.3.2 Structured vs. Unstructured Data

Before describing the features and dataset used in this chapter, it is imperative to understand the choice of structured data as input to the DNN model vis-à-vis unstructured data. In a ML context, unstructured data refers to unorganized data like images, video, and text that does not follow a predefined format and is thus more challenging to process. To prepare unstructured data for learning, there is a need to carry out various transformations, encoding, and processing that are computationally expensive and time-consuming [135]. Thus, applications where speed and resources come at a premium, structured data is preferred over unstructured data for learning. Since this work is focused on quick detection of malicious webpages over millions of records, structured data is a preferred choice for this model. Subsequently, in the result section, this choice would be vindicated by the fast training and testing time exhibited by our model.

5.3.3 Dataset and Features

Refer the 25 attributes listed in **Table 4.1**, which were analyzed for their suitability in Chapter 4. Few amongst these attributes were selected and refined further in section 3.1.3 for the final webpages dataset created for ML tasks in the current and subsequent chapters. The final selected features of the webpages dataset, which have been used for DNN classification in this chapter, are listed in **Table 3.2**. Features F1-11, less F10, as per **Table 3.2** have been used for DNN classification in this chapter. For further details on the dataset refer to Chapter 3 (preliminary analysis and visualization) and Appendix A (pre-processing code).

5.3.4 Deep Learning Model

The deep learning model's design for the detection of malicious webpages in this research is shown below in **Figure 5.1**.

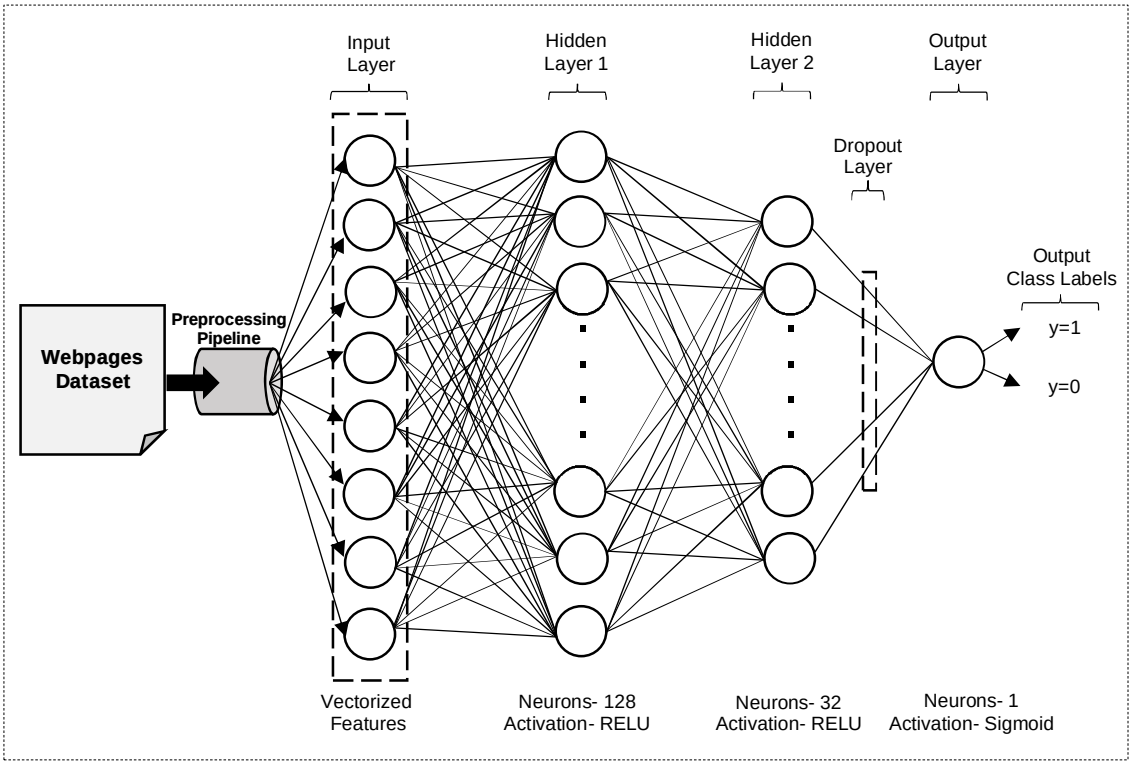

Figure 5.1: DNN Model for Malicious Webpage Classification

The dataset was preprocessed and fed to the Input Layer. The Input Layer carried out vectorization, as shown in **Table 5.1**.

Table 5.1: Feature (Input) layer- DNN with Structured Data

#	**Features Name**	**Transformation Carried Out**
F1, F2, F8, F9	url_vect, url_len, js_len, js_obf_len	Normalized and fed as a numerical column.
F4, F5	geo_loc, tld	Converted to Hashed Categorical Columns, with bucket size equal to the number of unique values in each.
F6, F7	who_is, https	One Hot coded and fed as Categorical Column.
F11	label	Converted into a single class label column with binary value 0/1.

The vectorized and normalized values given above are condensed into a dense feature layer and fed to the next layer. The next layer is a hidden layer comprising 128 fully connected neurons. The output of this layer is fed to the third layer, which too is a hidden layer of 32 fully connected neurons. RELU (Rectified Linear Unit), which is one of the activation functions used in deep learning, was used for both the hidden layers. The choice of RELU for hidden layers was based on its faster convergence during training. The third layer feeds its output to a fully connected single neuron layer with a Sigmoid activation function. The sigmoid function is another popular activation function used for binary classification problems that gives a binary 0/1 output for each of the two class labels defined in the F11 feature. Between the third and output layer, a dropout layer was introduced with a dropout rate set at 10%. Dropout is a technique of randomly excluding few nodes from update cycles. 10% dropout from a layer of 32 nodes means that three nodes are dropped in each update cycle. The use of this technique gave a regularized model and overcame overfitting [140]. The summary of tunable parameters in the generated DNN model is given in **Table 5.2**.

Table 5.2: Layer-wise Tunable Parameters (DNN with Structured Data)

Layer#	**Layer (Type)**	**Output Shape**	**Param#**
1	Input	Multiple	-
2	Dense (Hidden 1)	128	184192 (1438 x 128 Weights + 128 bias)
3	Dense (Hidden 2)	32	4128 (128 x 32 Weights + 32 bias)
-	Dropout	-	-
4	Output	1	33 (32 Weights + 1 bias)
	Total Params: 188,353 Trainable Params: 188,353 Non-trainable Params: 0		

This DNN model is a feed forward network that is trained using 'Gradient Descent'. Gradient Descent is an optimization algorithm that minimizes the cost/loss function by moving in the direction of steepest descent, thereby finding the minima of cost/loss function. 'Binary Cross Entropy' has been used as the loss function in this work. In this model, an extension of 'Gradient Descent' algorithm known as 'Adam Optimization' is used due to its better

performance. Adam is a variant of the Stochastic Gradient Descent algorithm which uses an adaptive learning rate. Its name is derived from **Ada**ptive **m**oment estimation [141]. The Gradient Descent algorithm works in two steps - the forward and backward pass. In the forward pass, it computes errors using current parameters. In backward pass, it computes gradients using partial derivatives and amends parameters accordingly [142].

At any neuron in layer $\boldsymbol{l}$ in this DNN model, calculations can be shown as given in Fig.2. Where, if $l = 1$ (i.e., the input layer), inputs are depicted by vector **X**= [x_1, x_2, …. , x_n]; for $l > 1$, inputs to the neuron are the activation outputs from previous layer $l - 1$ and are depicted by the vector $\boldsymbol{a^{l-1}} = [a_1^{l-1}, a_2^{l-1}, \ldots, a_i^{l-1}]$ .The vector of weights from layer $l - 1$ to j^{th} Neuron in l^{th} layer is depicted by $\boldsymbol{w_j^l} = [w_{j1}^l, w_{j2}^l, \ldots, w_{ji}^l]$ and for the complete l^{th} layer by,

$$\boldsymbol{w^l} = \begin{bmatrix} w_{11}^l & w_{12}^l & ..w_{1i}^l \\ w_{21}^l & w_{22}^l & ..w_{2i}^l \\ & \cdots & \\ w_{j1}^l & w_{j2}^l & ..w_{ji}^l \end{bmatrix}$$

Bias for j^{th} Neuron in l^{th} layer is depicted by b_j^l and for complete l^{th} layer by $\boldsymbol{b^l} = [b_1^l, b_2^l, \ldots, b_j^l]$. Similarly, vectors **W** and **b** can be shown as matrices combining $\boldsymbol{w^l}$ and $\boldsymbol{b^l}$ for all layers.

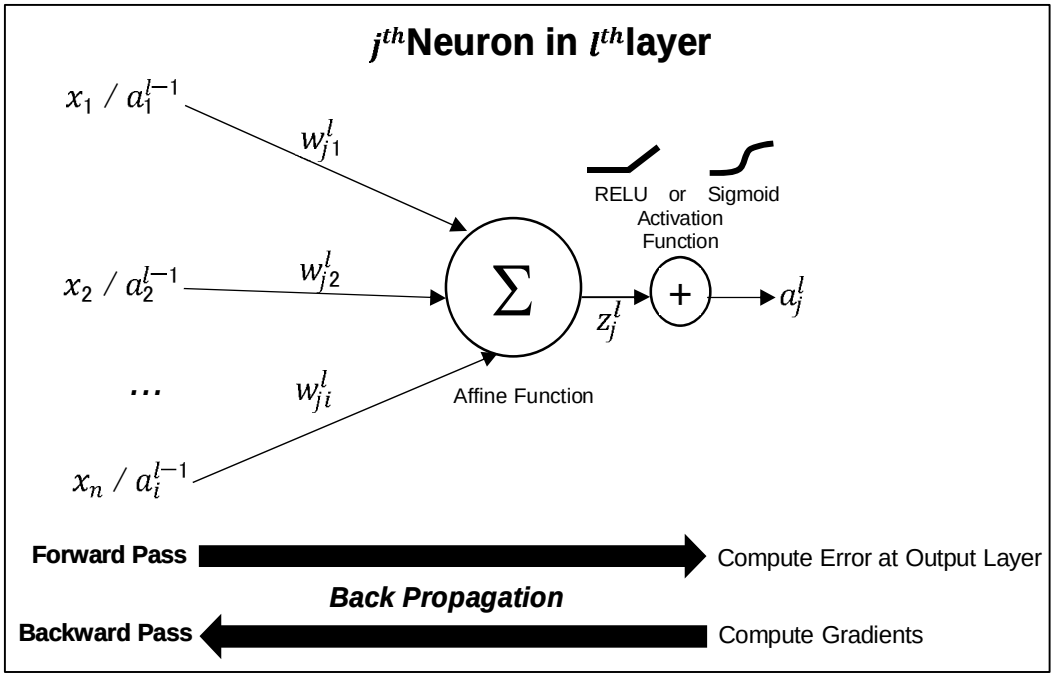

Figure 5.2: A Single Computation Unit in DNN

As seen in **Figure 5.2**, the Affine function z_j^l computes the weighted sum of all inputs coming to the neuron.

$$z_j^l = \sum_i w_{ji}^l a_i^{l-1} + b_j \quad (5.1)$$

At the input layer, this equation changes to,

$$z_j^1 = \sum_n w_i^1 x_i + b_j \quad (5.2)$$

Equations (5.1) and (5.2) are shown with vectors as,

$$\boldsymbol{z^l = w^l . a^{l-1} + b} \quad (5.1a)$$

$$\boldsymbol{z^1 = w^1 . X + b} \quad (5.2a)$$

Affine function feeds into the activation function, which for layer-2 and layer-3 ($l = 2,3$) is a RELU as represented by equation (5.3). Activation function for the output layer ($l = 4$) is a Sigmoid as represented by equation (5.4).

$$a_j^l = max(0, z_j^l) \quad (5.3)$$

$$y = a_j^l = \frac{1}{(1+e^{-z_j^l})} \quad (5.4)$$

As shown in **Figure 5.2**, many such neural units form a neural layer, and many such layers are stacked together to form the DNN. The output of the activation function in the final layer is the DNN output, $\boldsymbol{y}$. The training comprises actions to find those values of parameters **W** and **b** that lead to the correct predicted output $\boldsymbol{y}$. To achieve this, the 'Binary Cross Entropy' loss function **E** compares values of final output $\boldsymbol{y}$ and target output **t** as shown in equation (5.5). This completes the forward pass or the feed-forward in the DNN.

$$E = \frac{-1}{n} \sum_{c=1}^{n} [t_c \, log(y_c) + (1 - t_c) \, log(1 - y_c)] \quad (5.5)$$

After that, backward pass or backpropagation is carried out, wherein error contribution of each parameter w_{jl}^l and b_j^l in the network towards total loss **E** is computed through gradients. Using the chain rule of differentiation, the partial derivates are calculated successively backward from output to input. In this model, the error gradient for w in the last layer ($l = 4$) is given as,

$$\frac{\partial E}{\partial w_{kj}^4} = \frac{\partial E}{\partial y_c}\frac{\partial y_c}{\partial z_k^4}\frac{\partial z_k^4}{\partial w_{kj}^4} \quad (5.6)$$

Solving partial derivate using equation (5.5),

$$\frac{\partial E}{\partial y_c} = \frac{-t_c}{y_c} + \frac{1-t_c}{1-y_c} = \frac{y_c - t_c}{y_c(1-y_c)} \quad (5.7)$$

Solving partial derivative using equation (5.4),

$$\frac{\partial y_c}{\partial z_k^4} = \frac{e^{-z_k^4}}{\left(1+e^{-z_k^4}\right)^2} = y_c(1-y_c) \quad (5.8)$$

Solving partial derivative using equation (5.1),

$$\frac{\partial z_k^4}{\partial w_{kj}^4} = a_j^3 \quad (5.9)$$

Using equations (5.6) to (5.9),

$$\frac{\partial E}{\partial w_{kj}^4} = (y_c - t_c)a_j^3 \quad (5.10)$$

Similarly, using the chain rule to compute error gradient for b in the last layer,

$$\frac{\partial E}{\partial b_k^4} = \frac{\partial E}{\partial y_c}\frac{\partial y_c}{\partial z_k^4}\frac{\partial z_k^4}{\partial b_k^4} \quad (5.11)$$

Computing partial derivative using equation (5.1),

$$\frac{\partial z_k^4}{\partial b_k^4} = 1 \quad (5.12)$$

Using equations (5.7), (5.8), (5.11), and (5.12),

$$\frac{\partial E}{\partial b_k^4} = (y_c - t_c) \quad (5.13)$$

Equations (5.10) and (5.13) give us error gradients concerning parameters in the last layer (Note: The final layer has only a single neuron with Sigmoid activation. Thus, k=1.). Computation of error gradients for lower layers' parameters requires recursive application of chain rule as part of the backpropagation algorithm. For the third layer ($l = 3$), the error gradient with respect to w is given below.

$$\frac{\partial E}{\partial w_{ji}^3} = \frac{\partial E}{\partial y_c}\frac{\partial y_c}{\partial z_k^4}\frac{\partial z_k^4}{\partial a_j^3}\frac{\partial a_j^3}{\partial z_j^3}\frac{\partial z_j^3}{\partial w_{ji}^3} \quad (5.14)$$

The first two partial derivatives have been computed in equation (5.7) and (5.8), the third is calculated using equation (5.1),

$$\frac{\partial z_k^4}{\partial a_j^3} = w_{kj}^4 \tag{5.15}$$

Solving partial derivate of RELU activation of layer 3 using equation (5.3),

$$\frac{\partial a_j^3}{\partial z_j^3} = \begin{cases} 1 \; when \; z_j^3 > 0 \\ 0 \; when \; z_j^3 \leq 0 \end{cases} \tag{5.16}$$

Using equation (5.1),

$$\frac{\partial z_j^3}{\partial w_{ji}^3} = a_i^2 \tag{5.17}$$

Using equations (5.7), (5.8), and (5.14) to (5.17),

$$\frac{\partial E}{\partial w_{ji}^3} = \begin{cases} (y_c - t_c) w_{kj}^4 a_i^2 \; when \; z_j^3 > 0 \\ 0 \qquad\qquad\qquad when \; z_j^3 \leq 0 \end{cases} \tag{5.18}$$

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

$$\frac{\partial E}{\partial b_j^3} = \frac{\partial E}{\partial y_c} \frac{\partial y_c}{\partial z_k^4} \frac{\partial z_k^4}{\partial a_j^3} \frac{\partial a_j^3}{\partial z_j^3} \frac{\partial z_j^3}{\partial b_j^3} \tag{5.19}$$

Using equation (5.1),

$$\frac{\partial z_j^3}{\partial b_j^3} = 1 \tag{5.20}$$

Using equations (5.7), (5.8), (5.14) to (5.16), (5.19), and (5.20),

$$\frac{\partial E}{\partial b_j^3} = \begin{cases} (y_c - t_c) w_{kj}^4 \; when \; z_j^3 > 0 \\ 0 \qquad\qquad when \; z_j^3 \leq 0 \end{cases} \tag{5.21}$$

Moving further backward in the DNN, gradients with respect to parameters of layer-2 are computed recursively based on the derivates of layer-3. Since layer 2 uses RELU activation like layer 3, 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. After that, the loss function **E** is minimized over all **n** input samples of **X**, using gradient descent. 'Adam' algorithm was used with a mini-batch size

of 2048 random samples. Adam algorithm uses the partial derivates computed during the backpropagation to determine the global minima of the loss function $\boldsymbol{E}$ with respect to **W** and **b**. This ultimately leads to a successive tweaking of **W** and **b** to minimize the loss function for all **X** inputs. This process can be denoted mathematically by equation (5.22).

$$argmin_{\boldsymbol{W},\boldsymbol{b}}\boldsymbol{E} \tag{5.22}$$

W and **b** are tweaked, and their new values are computed in each mini-batch using equations (5.23) and (5.24). Here in the equations, $\boldsymbol{\alpha}$ is the learning rate, and it signifies the step size during descent. For this model, $\alpha = 1e^{-3}$ was found to be optimum.

$$\boldsymbol{W} = \boldsymbol{W} - \alpha\frac{\boldsymbol{\partial E}}{\boldsymbol{\partial W}} \tag{5.23}$$

$$\boldsymbol{b} = \boldsymbol{b} - \alpha\frac{\boldsymbol{\partial E}}{\boldsymbol{\partial b}} \tag{5.24}$$

5.3.5 Handling Class Imbalance

It is pertinent to note that the number of malicious webpages on the Internet is just a very small percentage compared to benign webpages. This disproportion is also visible in the dataset that was prepared for this research (refer to **Figure 3.2** in Chapter 3). Out of the 1.2 million samples in the training dataset, 97.73% were positive (benign), and only 2.27% were negative (malicious). The class imbalance creates a bias towards the majority class, thereby resulting in an inaccurate trained model [143]. Thus, there was a need to address this imbalance.

Techniques that are generally used for handling class imbalance are discussed below:

- **Over Sampling:** The dataset is oversampled to create additional samples of the minority-class [144]. ADASYN and SMOTE are some oversampling algorithms that are suitable for this task (Note: ADASYN (Adaptive Synthetic) and SMOTE (Synthetic Minority Oversampling Technique) are algorithms for generating synthetic samples of minority class based on existing minority observations.).

- **Under Sampling:** The majority-class samples are reduced to bring down imbalance [144]. However, this technique can lead to a drastic reduction of samples and is thus prone to error.

- **Modifying the Weights of Loss Function:** The loss function too can be modified to handle the imbalance by giving higher weight to the minority class vis-a-vis the majority class [145]. These weights are different from parameter w of DNN model discussed in the previous subsection. These are weightages given to classes in the loss function and are denoted by wt.

Modification of weights in the loss function was used as it gave better results than the other two techniques. The modified weights for the binary cross entropy loss function used in this model were computed using the formula below.

$$wt_0 = \frac{total}{(neg \times 2)} \quad (5.25)$$

$$wt_1 = \frac{total}{(pos \times 2)} \quad (5.26)$$

where,

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

These weights were fed to the 'Binary Cross Entropy' loss function during the model's training.

Apart from the techniques discussed, two more tricks were used to handle this imbalanced data. Firstly, data was handled in a large batch size of 2048 samples, which ensured that the minority-class was adequately represented in each batch. Secondly, correct initial bias was given to the model at the start of training to ensure faster convergence. The correct bias was derived using equation (5.27) given below [146].

$$bias_0 = log_e(pos/neg) \quad (5.27)$$

5.3.6 Implementation

The DNN model proposed in the last two subsections was implemented using TensorFlow and Keras libraries. TensorFlow is an open-source ML platform in Python that was released by Google [147]. Keras [148] is a deep learning library in Python that can run on top of TensorFlow. Few functionalities that were not part of the standard library were coded in Python using NumPy library. NumPy is a Python library that provides functions for vector calculus. For generating results and analysis graphs, TensorBoard and Seaborn libraries are used. TensorBoard library provides storage, retrieval, visualization of machine learning results produced using TensorFlow. Seaborn is a Python data visualization library based on Matplotlib. The code is written to run on CPUs. However, with minor modification, the code can run on GPUs and thereby further improve its time performance. The code is published online on Kaggle [149] to support further research.

5.4 Results and Analysis

The model proposed in the previous section was trained with 1.0 million samples. For validation, a separate dataset of 0.2 million samples and for testing a different test dataset of 0.35 million samples were used.

5.4.1 Training and Validation Results

The DNN model was trained over 40 Epochs with Early Stopping (with patience set to 20). The accuracy in each epoch during training and validation is plotted below in **Figure 5.3**. The dark red line represents training accuracy, while the light red line depicts validation accuracy. While the training was set to 40 epochs, it is seen that it was stopped at 25 by the early stopping algorithm when the accuracy stabilized.

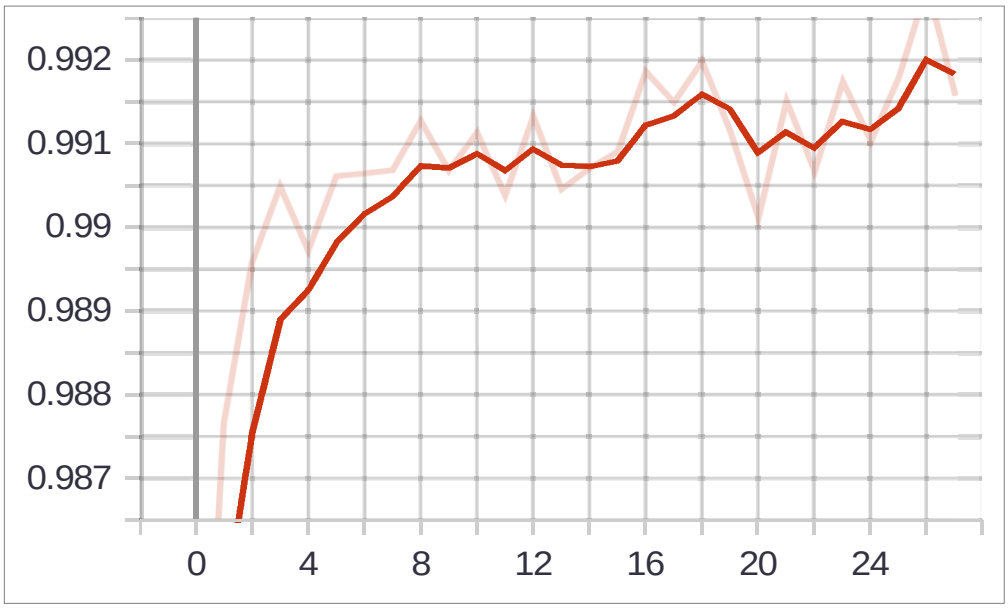

Figure 5.3: Training and Validation Accuracy

Similarly, the binary cross entropy loss during the validation and training is shown below in **Figure 5.4**.

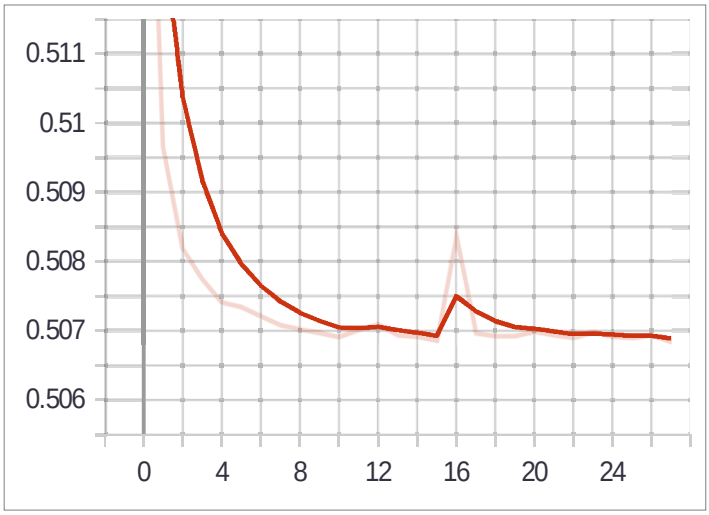

Figure 5.4: Training and Validation Loss

It can be seen from **Figures 5.3** and **5.4** that the training accuracy had lagged behind validation accuracy, and validation loss has lagged behind training loss. This behavior is attributable to the 10% dropout that was implemented during training for regularization (10% dropout will randomly switch off $\frac{1}{10^{th}}$ neurons in layer 3, thereby reducing training performance). Since dropout is not used during validation and testing, training performance always lags validation and test performance in such regularized models.

In the previous section, we had discussed the initial bias used in this model as per equation (5.27). The modified initial bias resulted in faster convergence, as can be seen in **Figure 5.5**. The graph shows that the loss dropped rapidly within few epochs with the use of initial bias.

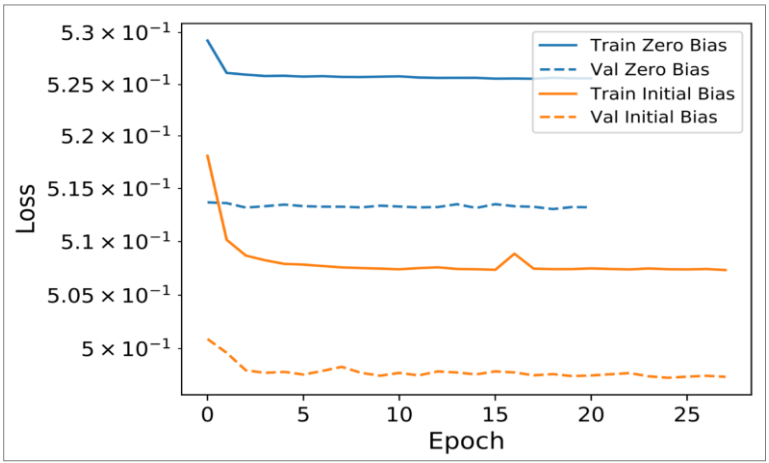

Figure 5.5: Impact of Initial Bias

5.4.2 Evaluation on Test Dataset

The test dataset's evaluation gave results as shown in **Table 5.3**. The total number of samples in the test dataset were 0.564 million.

Table 5.3: Evaluation on Test Dataset (DNN with Structured Data)

Metrics	Value
Accuracy	0.9981
Recall (TPR, Sensitivity)	0.9970
Specificity (TNR, Selectivity)	0.9990
Precision (PPV)	0.9586
NPV	0.9999
F1 Score	0.9774
*Note: Positive class represents the 'Malicious label'.	

The meaning of metrics used in **Table 5.3** has already been given in **Table 4.4**. Confusion Matrix that gives us the distribution of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) is given below in **Figure 5.6**. The confusion matrix clearly highlights negligible FN and FP during the test dataset evaluation.

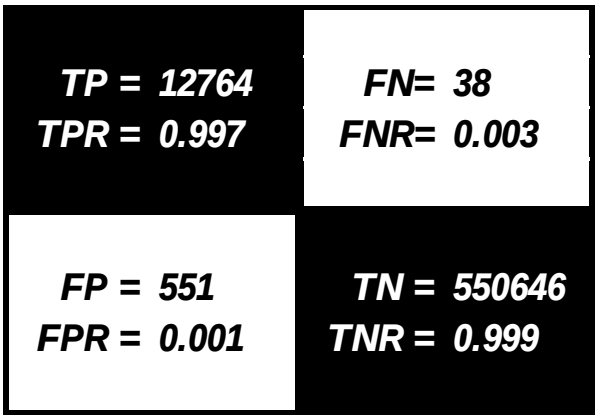

Figure 5.6: Confusion Matrix

AUC-ROC metric denotes the area under the Receiver Operating Characteristic curve. This metric gives the probability of ranking a random positive sample vis-a-vis a random negative sample. The AUC graph is given below in **Figure 5.7**. The high AUC value (0.9967) shows that the DNN model is capable of distinguishing classes with negligible errors.

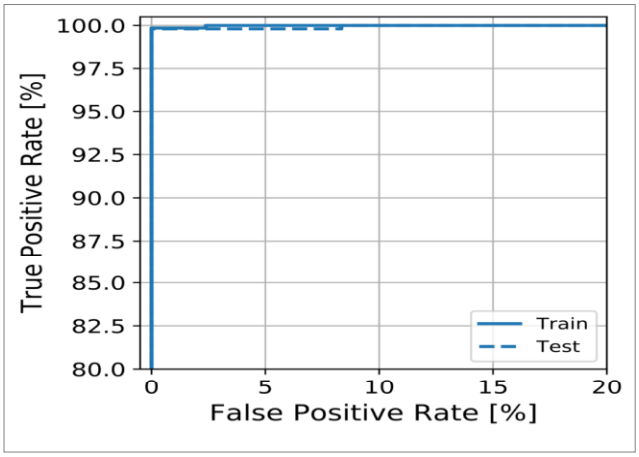

Figure 5.7: AUC-ROC Graph

5.4.3 Execution Time and Computational Resource

Time and computational resources used are essential factors in gauging any machine learning solution. Training time for 1 million records over 40 Epochs was 175.96 mins, which is impressive keeping in mind the high training time requirements of deep learning. This is the time taken on a 2.5Ghz Intel i5 CPU, without using any GPU, parallelization, or distributed computing. If these techniques are used, training time will come down further. What is essential is that the testing time per sample is less compared to other ML models [130][133]. The test time per sample is 14 . If we consider the time for preprocessing each sample (for example, preparing one sample from each

webpage being visited by a browser), we get a total time of 264 μs. Total time will vary based on the size of the webpage. Further, network delay has not been factored in this calculation. Such fast response in testing makes this trained DNN model suitable for deployment on Internet Browsers or other platforms for real-time maliciousness check.

5.4.4 Analysis

Analysis of training and validation results clearly show that the model was well trained without overfitting. The trained model gave a high accuracy of 99.81%, which surpasses the results of other machine learning models used until now to detect malicious webpages. The negligible False Negatives and False Positives, high Precision and Recall values substantiate the model's capability to produce accurate decisions with minimal false alarms.

5.5 Conclusions

This work provides a functional interdisciplinary approach to use deep learning in the field of web security. While other ML techniques have been used to detect malicious webpages, the use of deep learning in this field has been largely unexplored. It was seen that deep learning performs better than earlier ML models in the detection of malicious webpages. The deep learning model has outperformed previous models not only in accuracy, precision, and recall, but also in test response time. The performance of this model makes it suitable for real-time web security solutions on the Internet.

With respect to scope for further work, this DNN model may be deployed on a browser like Google Chrome or Mozilla Firefox using a plugin to provide real-time detection of malicious webpages while browsing. Also, the use of 'Tensorflow.js' [150] can be explored for training and running this model directly on the web browser. Lastly, we used structured data for deep learning. It would be interesting to know how the accuracy and response time changes with the use of unstructured data as input to the deep learning model, for example, feeding the web content directly to the DNN. We explore this in the next chapter.