Journal on
Artificial Intelligence

Tech Science Press

DOI: 10.32604/jai.2024.050563

ARTICLE

Deep Learning: A Theoretical Framework with Applications in Cyberattack Detection

Kaveh Heidary*

Department of Electrical Engineering and Computer Science, Alabama A&M University, Huntsville, AL 35803, USA

*Corresponding Author: Kaveh Heidary. Email: kaveh.heidary@aamu.edu

Received: 10 February 2024 Accepted: 28 May 2024 Published: 18 July 2024

ABSTRACT

This paper provides a detailed mathematical model governing the operation of feedforward neural networks (FFNN) and derives the backpropagation formulation utilized in the training process. Network protection systems must ensure secure access to the Internet, reliability of network services, consistency of applications, safeguarding of stored information, and data integrity while in transit across networks. The paper reports on the application of neural networks (NN) and deep learning (DL) analytics to the detection of network traffic anomalies, including network intrusions, and the timely prevention and mitigation of cyberattacks. Among the most prevalent cyber threats are R2L, U2L, probe, and distributed denial of service (DDoS), which disrupt normal network operations and interrupt vital services. Robust detection of the early stage of cyberattack phenomena and the consistent blockade of attack traffic including DDoS network packets comprise preventive measures that constitute effective means for cyber defense. The proposed system is an NN that utilizes a set of thirty-eight packet features for the real-time binary classification of network traffic. The NN system is trained with a dataset containing the packet attributes of a mix of normal and attack traffic. In this study, the KDD99 dataset, which was prepared by the MIT Lincoln Lab for the 1998 DARPA Intrusion Detection Evaluation Program, was used to train the NN and test its performance. It has been shown that an NN comprised of one or two hidden layers, with each layer containing a few neural nodes, can be trained to detect attack packets with concurrently high precision and recall.

KEYWORDS

Neural networks; backpropagation; classifier training; cybersecurity; packet classification; performance metrics

1 Introduction

Communication and control systems, including computers, sensor networks, and data centers, constitute the fundamental undergirding of modern society. The increasingly integrated global networks of sensors, actuators, computers, data centers, communication networks, software applications, and control systems, which are collectively labeled as information technology (IT) systems, handle the acquisition, production, storage, processing, and flow of sensitive and critical data related to personal, commercial, industrial, and government entities [1–3]. Among the myriad applications of IT systems are the control and monitoring of physical and industrial processes, including the electric power grid and virtually all other essential services. The industrial supervisory control and data

acquisition (SCADA) systems govern the operation of public and industrial systems that directly affect people's daily lives and the normal functioning of contemporary information-centric society [4]. The pervasive and growing interconnectivity of ubiquitous devices and systems, including sensors, data acquisition, storage, processing and communication modules, and actuators, amplify the complexity of IT systems [5–8]. Industrial control systems, including SCADA and other complex real-time sensing, communication, actuation, and control of web and application servers, databases, interfaces, switches, and routers, are distributed over wide geographic areas, and operate under incongruent standards, administrative regulations, and disparate jurisdictions [9].

The consistent availability, confidentiality, and integrity of the information that is generated, stored, accessed, processed, and transmitted through IT systems are fundamental and functional perquisites of modern society. As the world becomes increasingly interconnected and the industrial Internet of Things (IoT) proliferates globally, the production volume and flow speed of mission-critical data will accelerate. To deliver the expected value and services to their customers, clients, and constituents, commercial organizations, civil society, and governments will generate, store, access, transmit, update, and process vast quantities of data.

Bad actors, including criminal organizations and hostile governments, are constantly probing networks and exploiting the vulnerabilities of IT systems as a means to mount cyberattacks for financial gains, theft of intellectual property, espionage, and obtaining tactical and strategic advantages. Cyberattacks may involve theft, destruction, alteration, and encryption of data and processes, or disruption of normal services by unauthorized entities [10]. The increasing frequency, sophistication, and severity of cyberattacks demand the development of robust, agile, rapidly deployable, scalable, economical, low-power, and adaptive network intrusion detection systems.

A straightforward and effective cyber defense mechanism may involve arming the standard network packet capture and packet sniffer systems, which are routinely utilized for system administration and traffic analysis functions, with packet classifiers for the detection of malicious activity [11–13]. A binary packet classifier that is capable of real-time assignment of labels, namely benign (normal) or attack, to each packet constitutes an economical yet powerful first line of defense against various cyberattack modalities including DDoS and malware.

Packet classification is a standard process deployed on various network devices, including high-speed Internet routers and firewalls. Owing to the increasing diversity of network services, including data, voice, television, web hosting, live streaming, gaming, etc., packet classification to form proper packet flows, provide firewalls, and quality of service have become essential components of Internet routers. They are routinely used for packet filtering, traffic accounting, and other network services, including traffic shaping and service-aware routing, where packets are classified into differentiated traffic flows for providing application-specific quality of service [14,15]. The NN-based attack detector described in this paper is an additional packet classification layer that can be added to the existing routers' packet classification algorithms. The fundamental difference between the NN classifier described in this paper and the rule-base packet classifiers in [14,15] is their underlying approach to making classification decisions. The rule-based classifiers rely mainly on the predefined procedures and instructions which are crafted by domain experts to make classification decisions. The NN classifiers, on the other hand, automatically discover the intricate patterns and hidden features of the data without explicit external guidelines provided by the user. The NN classifier described in this paper is capable of learning complex patterns and nonlinear relationships from data which makes it more effective and flexible. The proposed NN detector has low storage requirements and can be readily deployed in static random-access memory (SRAM), which provides the detector with flexibility and scalability.

2 Background

The development and implementation of cyberattack detection tools and mitigation strategies have been active areas of research for several decades [9,11] and [14–17]. Classical cybersecurity solutions include the utilization of access control methods such as identity and access management (IAM) tools; multifactor authentication techniques such as passwords, token-based authentication, and biometrics; data encryption; antimalware detection; endpoint protection; and firewalls [17–19].

Motivated by the biological brain's remarkable ability to handle complex problems and information processing, researchers have dedicated several decades to developing artificial neural networks (ANNs) inspired by the structure and function of the brain [20–22]. Fundamentally, an NN is a massively parallel computational engine that is comprised of interconnected elementary analog processing nodes called neurons. The function of each node (neuron) is simply to compute the weighted sum of its various inputs and pass on the result through a non-linear activation function such as sigmoid [20–22]. The number of neural layers and nodes, the manner of interconnections among the nodes, and the weight coefficients assigned to the various connections in the network are the parameters that determine the network functionality. Although the initial inspirations responsible for the development of NN were primarily biological systems, they have proven their value for complex non-linear mapping as well as their practical applications to various problems in pattern recognition, signal processing, natural language processing, time-series prediction, and forecasting [19–23] and [24–26]. Because of their VLSI implementation, NN solutions are economical and can be deployed at scale [27]. Over the years, efficient learning algorithms have been devised to determine the NN weight coefficients [28].

In Section 3, the general framework of the feedforward multilayered perceptron is described. The computational complexity of the feedforward network is presented in Section 4. The process for computing the weights and biases of the feedforward multilayered network is explained in Section 5, where Sections 5.1 and 5.2 present, respectively, the input-output mapping function and the backpropagation algorithm. The training process used for the NN packet classifier is presented in Section 6. Section 7 describes the dataset used for training and testing the performance of the packet classifier. Test results are presented in Section 8. Conclusions and future research are presented in Section 9.

3 Feedforward Multilayered Perceptron Architecture

The binary classifier utilized here comprises a multi-layered perceptron (MLP), which is also called a feedforward neural network (FFNN), as shown in Fig. 1. In this section, we will briefly discuss the NN architecture and dynamics. The process of NN learning is achieved through gradient descent and backpropagation learning algorithms. The data processing progression from the network input to the output and the learning algorithm are succinctly reviewed in Section 4. The derivations and detailed formulations of the backpropagation algorithm are given in Section 5.

A typical FFNN is comprised of an input layer, an output layer, and one or more hidden layers. The input and output layers are the visible layers of the network, and all the other layers are called hidden layers. The reason for employing the term "hidden" for the layers sandwiched between the input and the output layers is that in an NN classifier, during the training phase, for each training vector at the input, the desired responses of the output layer neurons are known, whereas the desired outputs of the neurons in the hidden layers are not known a priori. Each layer is comprised of several neurons, which are also called nodes. In a fully connected network, each of the nodes (neurons) in a typical layer is connected to all the nodes in the next layer.

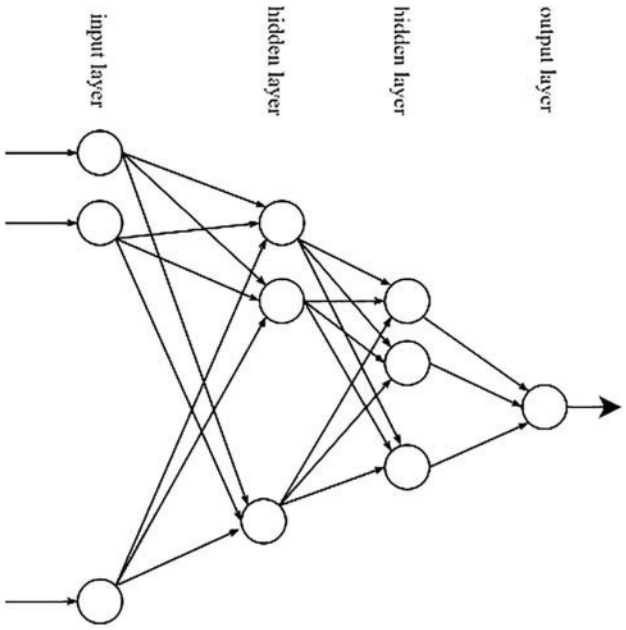

Figure 1: Fully connected feedforward neural network

The data processing steps progress from the input layer on the left to the output layer on the right, as shown in Fig. 1. The number of nodes in the input layer is dictated by the dimensionality of the input feature vectors, which the network is designed to process. The number of nodes in the output layer is determined by the functionality of the network. For example, a network that is intended for binary classification of input data has only one node at the output layer. Likewise, the number of nodes in the output layer of a NN classifier with K distinct classes is K. The number of hidden layers in the NN and the number of nodes in each of the hidden layers are application-dependent and are determined by standard optimization techniques or trial and error.

Fig. 1 shows an FFNN, where the nodes in the left column and the rightmost node comprise the visible layers of the network and denote, respectively, the input layer and the output layer. All the other layers between the input layer and the output layer are hidden layers. Two operations are performed at each node: (i) the weighted sum of the outputs of all the nodes in the preceding layer is computed at the node input and a bias is added to form the input signal; (ii) the node applies a nonlinear activation function to the signal at its input to generate the output signal.

4 Computational Complexity of the Feedforward Neural Network

In a fully connected FFNN, the input of a typical node is the weighted sum of the outputs of all the nodes in the preceding layer added to the bias factor of the node under consideration. The activation (output) of the neuron is expressed by Eq. (1).

$$y_i^{(k)} = f\left[\left(\sum_{j=1}^{N_{k-1}} w_{ij}^{(k)} y_j^{(k-1)}\right) + b_i^{(k)}\right]; 1 \le k \le K, 1 \le i \le N_k. \tag{1}$$

where k, i denote, respectively, the layer-index and the index of the neuron within the layer; $y_i^{(k)}$ denotes the output (activation) of the i^{th} neuron of the k^{th} layer; $w_{ij}^{(k)}$ denotes the weight factor associated with the connection between the j^{th} neuron of the $(k-1)^{th}$ layer and the i^{th} neuron of the k^{th} layer; $b_i^{(k)}$ denotes

the bias factor or threshold associated with the i^{th} neuron of the k^{th} layer; N_{k-1} denotes the number of nodes in the $(k-1)^{th}$ layer; K denotes the total number of layers; N_k denotes the number of nodes in the k^{th} layer; and f represents the non-linear neural activation function.

The input layer is considered the zeroth layer, and the outputs of the input layer nodes (neurons) are the same as the corresponding components of the input feature vector, which is to be processed by the network. The nodes of the input layer, the 0^{th}-layer, simply pass on the signals applied at their inputs to their respective outputs $y_i^{(0)} = x_i$. The neural activation function f can be any non-linear function. Examples of activation functions are defined below in Eqs. (2a)–(2f) and are plotted in Fig. 2.

$$\textit{unit step function}: f(x) = \begin{cases} 1; & x \geq 0 \\ 0; & x < 0 \end{cases}. \tag{2a}$$

$$\textit{signum function}: f(x) = \begin{cases} 1; & x \geq 0 \\ -1; & x < 0 \end{cases}. \tag{2b}$$

$$\textit{sigmoid function}: f(x) = \frac{1}{1+e^{-x}}. \tag{2c}$$

$$\textit{modified sigmoid function } [-1,1]: f(x) = \frac{1-e^{-x}}{1+e^{-x}}. \tag{2d}$$

$$\textit{hyperbolic tangent function}: f(x) = \frac{e^{2x}-1}{e^{2x}+1}. \tag{2e}$$

$$\textit{rectified linear unit function } ReLU: f(x) = \begin{cases} x; & x \geq 0 \\ 0; & x < 0 \end{cases}. \tag{2f}$$

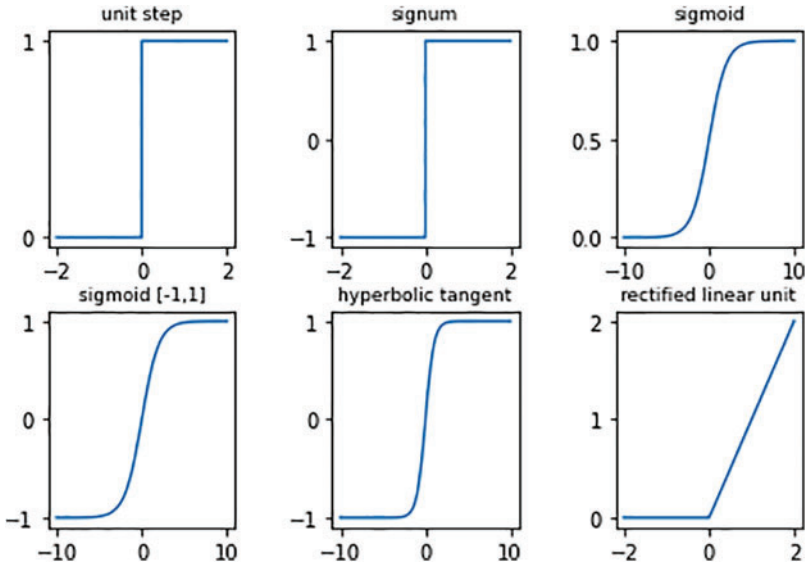

Figure 2: Typical non-linear activation functions

Fig. 3 shows a typical node in a typical layer, namely the i^{th} node in the k^{th} layer of the network and its connections to the nodes in the preceding layer, including the weights and the bias factor. The activation (output) of the node is expressed by Eq. (1) above. The meaning of the symbols in Fig. 3 is

explained right after Eq. (1). The matrix expression of Eq. (3) provides the relationship between the outputs of all the N_k nodes in the k^{th} layer of the network and the outputs of the N_{k-1} nodes of the $(k-1)^{th}$ layer.

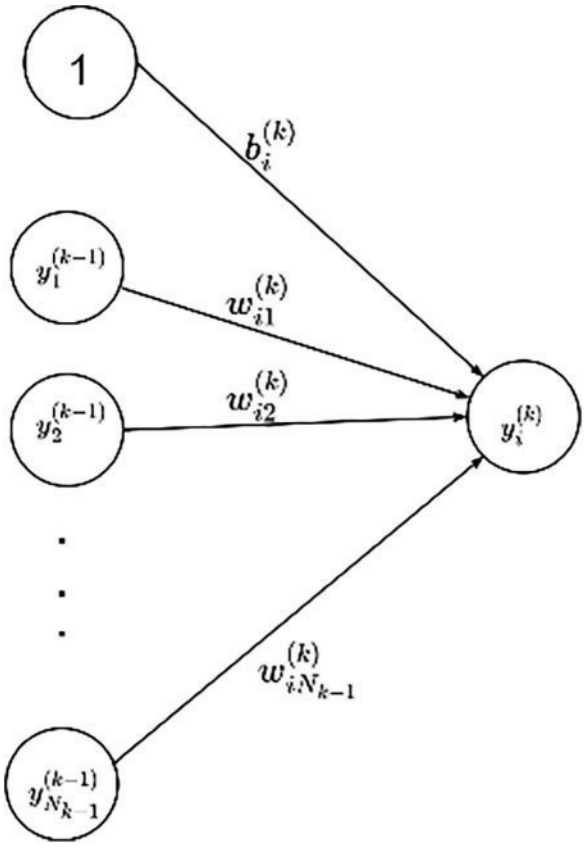

Figure 3: The input of the i^{th} neuron in the k^{th} layer of the feedforward network

In Eq. (3a) the superscript k refers to the layer-index in the network; f represents the neural activation function; $\mathbf{Y}^{(k)}$ is a column vector with length of (N_{k+1}) containing the outputs of all the nodes in the k^{th} layer prepended with one; $\mathbf{Y}^{(k-1)}$ is a column vector with length of $(N_{k-1}+1)$ containing the outputs of all the nodes in the $(k-1)^{th}$ layer prepended with one; and $\mathbf{W}^{(k)}$ is a matrix with dimensions $N_k \times (N_{k-1}+1)$ containing all the weight coefficients and bias factors associated with the connections to the inputs of the nodes in the k^{th} layer of the network.

$$\mathbf{Y}^{(k)} = f\left(\mathbf{W}^{(k)}\mathbf{Y}^{(k-1)}\right). \tag{3a}$$

$$\mathbf{Y}^{(k)} = \begin{bmatrix} 1 \\ y_1^{(k)} \\ \cdot \\ \cdot \\ \cdot \\ y_{N_k}^{(k)} \end{bmatrix}, \quad \mathbf{Y}^{(k-1)} = \begin{bmatrix} 1 \\ y_1^{(k-1)} \\ \cdot \\ \cdot \\ \cdot \\ y_{N_{k-1}}^{(k-1)} \end{bmatrix}. \tag{3b}$$

$$\mathbf{W}^{(k)} = \begin{bmatrix} b_1^{(k)} & w_{1,1}^{(k)} & w_{1,2}^{(k)} & \dots & w_{1,N_{k-1}}^{(k)} \\ & & & \vdots & \\ b_{N_k}^{(k)} & w_{N_k,1}^{(k)} & w_{N_k,2}^{(k)} & \dots & w_{N_k,N_{k-1}}^{(k)} \end{bmatrix}. \tag{3c}$$

In Eq. (3a), the activation function f is vectorized so that it operates on each component of its argument, which is a column vector. As seen from Eqs. (3a)–(3c), the NN processes the incoming data applied to the network input, which is also denoted as the zeroth layer, by a sequence of matrix multiplications followed by the nonlinear neural activation operations. Specifically, the input data represented as a column vector is multiplied with $\mathbf{W}^{(1)}$, which denotes the weight-bias matrix of the first hidden layer, and the result of the matrix multiplication is subsequently operated upon by the activation function f. The result of this operation is then multiplied with $\mathbf{W}^{(2)}$ and subsequently operated upon by f, and so on, until the last layer is reached, and the NN output is computed.

It is seen from above that the number of matrix multiplications required to compute the network response to the input data is equal to $(K + 1)$, where K represents the number of hidden layers. The number of algebraic operations and the computation of the neural activation functions required for the processing of a typical feature vector applied to the input of the NN are given below:

$$P_{mults.,adds.} = N_1 (D + 1) + \sum_{k=1}^{K} N_{k+1} (N_k + 1). \tag{4a}$$

$$P_f = \sum_{k=1}^{K+1} N_k. \tag{4b}$$

where $P_{mults.,adds.}$, P_f denote, respectively, the number of decimal additions-multiplications, and the number of activation function evaluations. The number of features (components) of the input vector is denoted as D; the number of hidden layers comprising the neural network is K; N_k denotes the number of nodes in the k^{th} layer of the network; and the subscript $K + 1$ denotes the output layer.

5 Multilayered Feedforward Neural Network Training

This section provides a detailed block diagram representation of the fully connected, multilayered FFNN. The complete mathematical formulations of the forward pass for modeling the network operations on the input data are given. The derivations of mathematical formulas used in the backpropagation operation are also given. Formulas for the computation of the loss (cost) function and its relationship to the network weights and biases are derived. The relationship between errors at the outputs of the neurons in consecutive layers of the network is formulated. The detailed exposition of the backpropagation algorithm, which is used for training the NN is discussed. The nomenclature employed in this section uses explicit symbols for the weights and biases, which are different from the notations in Section 3.

5.1 Forward Pass Operation of the Multilayered Perceptron

The block diagram representation of the FFNN is shown in Fig. 4, where the network consists of the input layer, the output layer, and l hidden layers. In a typical NN, the number of hidden layers is equal to or greater than one. The hidden layers are designated by their indices, one through l, where indices one and l denote, respectively, the first and the last hidden layers. The input layer is also called the zeroth layer, and the output layer is the $(l + 1)^{th}$ layer.

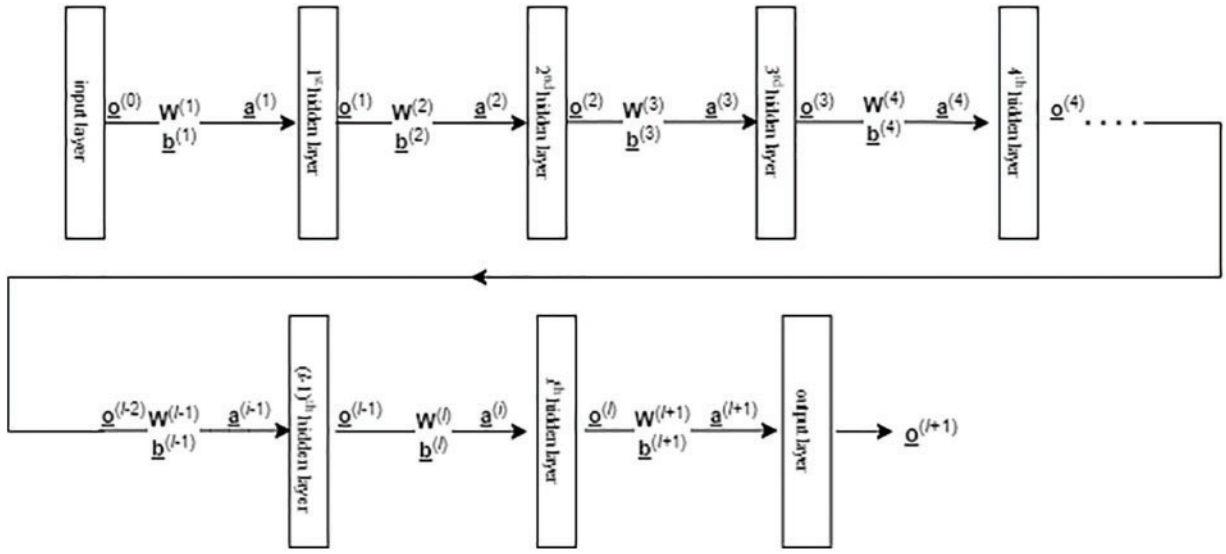

Figure 4: Block diagram representation of the multilayered feedforward neural network

Each of the hidden layers and the output layer, namely layers one through $(l + 1)$, consist of a set of neurons, which are also called nodes, where r_k denotes the number of neurons (nodes) in the k^{th} layer, including the hidden layers $1 \leq k \leq l$, and the output layer $k = l + 1$. The input layer has r_0 nodes, the outputs of which are the components of the feature vector representing the input data applied to the NN. Each layer is fully connected to the layer that immediately follows it. This means all the neurons (nodes) of the input layer as well as all the nodes of each hidden layer are connected to every neuron in the next layer. There are no connections between the neurons of the same layer. The neural connections are unidirectional, and signals propagate from neurons of each layer to the neurons of the next layer from left to right. In Fig. 4, column vectors are represented by underlined bold lowercase letters, matrices are represented by bold block letters, and the meaning of each symbol is explained in the following paragraphs.

The input and output of the k^{th} hidden layer, $1 \leq k \leq l$, are column vectors of length r_k, which are, respectively, designated as $\vec{\boldsymbol{a}}^{(k)}, \vec{\boldsymbol{o}}^{(k)}$ in the following analysis, and are denoted in Fig. 4 as $\underline{\mathbf{a}}^{(k)}$, $\underline{\mathbf{o}}^{(k)}$. Each element of the output vector at the k^{th} hidden layer is obtained from the corresponding element of the hidden layer's input vector operated upon by the neural activation function. The input layer of the block diagram of Fig. 4 at the upper-left, which is also called the zeroth layer, simply passes the vector applied to the input of the NN to the output of the zeroth layer. In all the other layers, however, two operations are performed. In each layer, the inputs of the neurons are computed as the weighted sum of the outputs of all the neurons in the preceding layer plus the bias vector. The output of each neuron is then computed by applying a nonlinear activation function, such as sigmoid or *ReLU*, to the input of the neuron.

$$\vec{\boldsymbol{a}}^{(k)} = \left[a_1^{(k)}, a_2^{(k)}, \ldots, a_{r_k}^{(k)}\right]^T, \vec{\boldsymbol{o}}^{(k)} = \left[o_1^{(k)}, o_2^{(k)}, \ldots, o_{r_k}^{(k)}\right]^T. \tag{5a}$$

$$\vec{\boldsymbol{o}}^{(k)} = \sigma\left(\vec{\boldsymbol{a}}^{(k)}\right), o_i^{(k)} = \sigma\left(a_i^{(k)}\right), 1 \leq i \leq r_k. \tag{5b}$$

where $\vec{\boldsymbol{a}}^{(k)}$, $\vec{\boldsymbol{o}}^{(k)}$ denote, respectively, the input and output vectors at the k^{th} layer in Fig. 4, the superscript T denotes matrix transpose, the subscripts refer to vector components, and σ denotes the neural

activation function, for example, the sigmoid function of Eq. (2c). In the block diagram of Fig. 4, a line connecting a pair of consecutive layers, for example, the $(k-1)^{th}$ and k^{th} layers, comprises $r_k \times r_{k-1}$ weighted connections, which is represented by the weight matrix $\mathbf{W}^{(k)}$. Where, r_{k-1}, r_k denote the number of neurons in two consecutive layers. The input vector of the k^{th} layer is obtained from the output vector of the $(k-1)^{th}$ layer by Eq. (6) below:

$$\vec{\boldsymbol{a}}^{(k)} = \mathbf{W}^{(k)}\vec{\boldsymbol{o}}^{(k-1)} + \vec{\boldsymbol{b}}^{(k)}; 1 \le k \le l+1. \tag{6a}$$

$$\vec{\boldsymbol{o}}^{(k-1)} = \left[o_1^{(k-1)}, o_2^{(k-1)}, \ldots, o_{r_{k-1}}^{(k-1)}\right]^T; 1 \le k \le l+1. \tag{6b}$$

$$\mathbf{W}^{(\boldsymbol{k})} = \begin{bmatrix} w_{1,1}^{(k)} & w_{1,2}^{(k)} & \ldots & w_{1,r_{k-1}}^{(k)} \\ w_{2,1}^{(k)} & w_{2,2}^{(k)} & \ldots & w_{2,r_{k-1}}^{(k)} \\ & & \vdots & \\ w_{r_k,1}^{(k)} & w_{r_k,2}^{(k)} & \ldots & w_{r_k,r_{k-1}}^{(k)} \end{bmatrix}, \vec{\boldsymbol{b}}^{(k)} = \left[b_1^{(k)}, b_2^{(k)}, \ldots, b_{r_k}^{(k)}\right]^T. \tag{6c}$$

where l is the number of hidden layers. In Eq. (6a), the terms on the right hand side, $\mathbf{W}^{(\boldsymbol{k})}$, $\vec{\boldsymbol{o}}^{(k-1)}$, $\vec{\boldsymbol{b}}^{(k)}$, are matrices with dimension of, respectively, $(r_k \times r_{k-1})$, $(r_{k-1} \times 1)$, $(r_k \times 1)$, and the term on the left hand side is a column matrix with $(r_k \times 1)$ dimensions, whose components are expressed below:

$$a_i^{(k)} = b_i^{(k)} + \sum_{j=1}^{r_{k-1}} w_{ij}^{(k)} o_j^{(k-1)}; 1 \le i \le r_k. \tag{7}$$

where $a_i^{(k)}$ denotes the i^{th} element of the input vector to the k^{th} layer of the network; $o_j^{(k-1)}$ is the j^{th} element of the vector at the output of the $(k-1)^{th}$ network layer; $w_{ij}^{(k)}$ denotes the value of the i^{th} row and j^{th} column of the weight matrix $\mathbf{W}^{(\boldsymbol{k})}$; $b_i^{(k)}$ is the i^{th} element of the bias matrix $\vec{\mathbf{b}}^{(\boldsymbol{k})}$; and r_{k-1}, r_k denote the number of neurons in the $(k-1)^{th}$ and k^{th} network layers. Here, the weight and the bias matrices are associated with the connections between the $(k-1)^{th}$ and k^{th} layers of the block diagram of Fig. 4. It is also noted that $a_i^{(k)}$, $o_j^{(k-1)}$ are, respectively, the signal at the input of the i^{th} neuron of the k^{th} layer, and the signal at the output of the j^{th} neuron of the $(k-1)^{th}$ layer. It is further noted that $w_{ij}^{(k)}$ is the weight factor associated with the connection from the j^{th} neuron of the $(k-1)^{th}$ layer to the i^{th} neuron of the k^{th} layer, and $b_i^{(k)}$ is the bias factor of the i^{th} neuron of the k^{th} layer.

To compute the response of the neural network to an input vector, we start at the input layer, and analyze the cascade effect of consecutive layers of the network on the input vector. The input layer (zeroth layer) passes through the applied vector to the output of the zeroth layer without doing anything to it. In the block diagram of Fig. 4, the output of the input layer, which is also called the zeroth layer, is set equal to the vector applied to the input of the neural network.

$$\vec{\boldsymbol{o}}^{(0)} = \vec{\boldsymbol{x}}. \tag{8}$$

where $\vec{\boldsymbol{x}}, \vec{\boldsymbol{o}}^{(0)}$ denote, respectively, the vector applied to the network and the output of the zeroth layer. The output of the zeroth layer (input layer) together with the weight and bias matrices of the first hidden layer are used to compute the input matrix of the 1st hidden layer. The output matrix of the 1st hidden layer is then computed from its input. The operations for computing the input and output matrices associated with the 1st hidden layer are given in Eqs. (9a), (9b).

$$\vec{\boldsymbol{a}}^{(1)} = \mathbf{W}^{(1)}\vec{\boldsymbol{o}}^{(0)} + \vec{\boldsymbol{b}}^{(1)} = \mathbf{W}^{(1)}\vec{\boldsymbol{x}} + \vec{\boldsymbol{b}}^{(1)}. \tag{9a}$$

$$\vec{\boldsymbol{o}}^{(1)} = \sigma\left(\vec{\boldsymbol{a}}^{(1)}\right) = \sigma\left(\mathbf{W}^{(1)}\vec{\boldsymbol{x}} + \vec{\boldsymbol{b}}^{(1)}\right). \tag{9b}$$

where, $\vec{a}^{(1)}$, $\vec{o}^{(1)}$ denote, respectively the input and the output of the 1st hidden layer. Using the output of the 1st hidden layer, the input of the 2nd hidden layer is computed which is then utilized to compute the output of that layer as shown by the expressions of Eqs. (10a), (10b).

$$\vec{a}^{(2)} = \mathbf{W}^{(2)}\vec{o}^{(1)} + \vec{b}^{(2)} = \mathbf{W}^{(2)}\left(\sigma\left(\mathbf{W}^{(1)}\vec{x} + \vec{b}^{(1)}\right)\right) + \vec{b}^{(2)}. \tag{10a}$$

$$\vec{o}^{(2)} = \sigma\left(\vec{a}^{(2)}\right) = \sigma\left(\mathbf{W}^{(2)}\left(\sigma\left(\mathbf{W}^{(1)}\vec{x} + \vec{b}^{(1)}\right)\right) + \vec{b}^{(2)}\right). \tag{10b}$$

Using the output of the 2nd hidden layer, the input of the 3rd hidden layer and subsequently its output are computed as shown in Eqs. (11a), (11b).

$$\vec{a}^{(3)} = \mathbf{W}^{(3)}\vec{o}^{(2)} + \vec{b}^{(3)} = \mathbf{W}^{(3)}\left(\sigma\left(\mathbf{W}^{(2)}\left(\sigma\left(\mathbf{W}^{(1)}\vec{x} + \vec{b}^{(1)}\right)\right) + \vec{b}^{(2)}\right)\right) + \vec{b}^{(3)}. \tag{11a}$$

$$\vec{o}^{(3)} = \sigma\left(\vec{a}^{(3)}\right) = \sigma\left(\mathbf{W}^{(3)}\left(\sigma\left(\mathbf{W}^{(2)}\left(\sigma\left(\mathbf{W}^{(1)}\vec{x} + \vec{b}^{(1)}\right)\right) + \vec{b}^{(2)}\right)\right) + \vec{b}^{(3)}\right). \tag{11b}$$

Continuing the iterative process of using the output of a layer to compute the input of the next layer and subsequently the layer output, one arrives at the expression for the output of $(l+1)^{\text{th}}$ layer which represents the response of the NN to the input $\vec{x}$ as shown in Eqs. (12a), (12b).

$$\vec{o}^{(l+1)} = \sigma\left(\mathbf{W}^{(l+1)}\sigma\left(\mathbf{W}^{(l)}\sigma\left(\mathbf{W}^{(l-1)}\sigma\left(\ldots\sigma\left(\mathbf{W}^{(1)}\vec{x} + \vec{b}^{(1)}\right) + \vec{b}^{(2)}\right)\ldots + \vec{b}^{(l-1)}\right) + \vec{b}^{(l)}\right) + \vec{b}^{(l+1)}\right). \tag{12a}$$

$$\widehat{\vec{y}} = \vec{o}^{(l+1)}. \tag{12b}$$

where $\widehat{\vec{y}}$ represents the response of the NN of Fig. 4 to the input $\vec{x}$. The matrices $\mathbf{W}^{(k)}$, $\vec{b}^{(k)}$ denote the weight and bias matrices associated with the k^{th} layer of the network, where $1 \leq k \leq l+1$, and l is the number of hidden layers which is equal to or greater than one. In the expression of Eq. (12a), the neural activation function σ is assumed to be the same across all the layers, which does not need to be the case. The activation function of each layer can be any one of the nonlinear functions given in Eqs. (2a)–(2f). In the experiments of Sections 6–8, the activation functions utilized across all the hidden layers are the *ReLU* function of Eq. (2f), and the activation function of the output layer is the sigmoid function of Eq. (2c).

Eq. (12a), which transforms the input vector $\vec{x}$ to the output vector $\vec{o}^{(l+1)}$, is a composition function, transforming the r_0 dimensional vectors at the input to the corresponding r_{l+1} dimensional vectors at the output of the network. The linear operations $\mathbf{W}^{(k)}\boldsymbol{o}^{(k-1)} + \vec{b}^{(k)}$ are affine transformations, which map hyperplanes with r_{k-1} dimensions in one space to hyperplanes with r_k dimensions in another space. If the nonlinear activation function is chosen to be *ReLU*, $\sigma\left(\mathbf{W}^{(k)}\boldsymbol{o}^{(k-1)} + \vec{b}^{(k)}\right)$ represents folding of the hyperplane in the second space, where each hyperplane has a single fold. Therefore, each layer of the network transforms the input of the layer through affine transformation and folding. The overall effect of the multilayered perceptron is creating continuous piecewise linear functions. The transformation is represented by folded hyperplanes. The number of folds in each hyperplane is equal to the sum of binomial factors $\sum_{q=0}^{r_0}\binom{l+1}{q}$, where l+1 is the number of layers and r_0 is input dimensionality.

The number of layers and the number of neurons in each of the layers of the network, which constitute the architecture of the multilayered FFNN, as well as the neural activation functions are design parameters which are application dependent. For example, in a NN designed for a classification application the number of neurons of the input layer equals the dimensionality of the feature vector representation of the input data to be classified. The number of neurons of the output layer is determined by the number of classes. For instance, a binary classifier has only one neuron at the

output layer. The number of hidden layers and the number of neurons in each hidden layer are set heuristically by the designer. The values of the weights and biases are computed by applying the multilayered feedforward perceptron learning algorithm which involves a training set comprised of a set of input feature vectors and the associated known classes.

5.2 Multilayered Perceptron Learning Algorithm

The training process of the neural network classifier involves the computation of the weight and bias matrices using a given training set. The training set is comprised of a finite set of training vectors and their corresponding classes. The number of nodes at the input layer r_0 in Fig. 4 is equal to the dimensionality of the training vector, and the number of nodes at the output layer r_{l+1} is dependent on the number of classes of the classification problem for which the network is designed. For example, consider a specific training vector $\vec{\boldsymbol{x}}$ that belongs to class $j(1 \leq j \leq r_{l+1})$ and is applied to the network input at the 0^{th} layer (input layer). Because it belongs to class j, the output of the j^{th} neuron at the output layer must be one, and the outputs of all the other neurons in that layer must be zero. This desired output, which is also called the target, represents the ground-truth, and is different from the actual output computed by the NN. The actual output of the network is computed by applying the feedforward process of Eqs. (12a), (12b) to the input vector. The difference between the target output and the computed (actual) output, which is called the error at the output layer, is subsequently used to update the weights and biases of the network using the gradient descent and backpropagation algorithms, which are detailed below.

The stochastic gradient descent (SGD) process involves applying one trainer at a time to the neural network and updating the network weights and biases accordingly. The trainers are applied one at a time, and the output is computed; the computed output is compared to the target output or ground-truth as specified by the training set; the difference between the computed output (actual output in response to the input) and the target output is used to update the weights and biases. This process is repeated for each trainer in the training set. The execution of the training process for all the vectors in the training set is called one training epoch. The training process is repeated for a user-prescribed number of training epochs, until all the trainers are classified correctly or until the relative changes in the weights and biases fall below the user-prescribed thresholds. The error, which is also called the loss or cost function, associated with one trainer is given by Eq. (13).

$$E = \vec{\boldsymbol{O}}^{(l+1)} - \vec{\boldsymbol{y}}. \tag{13}$$

where $\vec{\boldsymbol{O}}^{(l+1)}$ is the actual response of the NN in Fig. 4 to the trainer feature vector $\vec{\boldsymbol{x}}$ applied at the input of the network, and $\vec{\boldsymbol{y}}$ is the target output (ground-truth). The pair of vectors $\vec{\boldsymbol{x}}$, and $\vec{\boldsymbol{y}}$ comprise one training instant. The components of the response vector $\vec{\boldsymbol{O}}^{(l+1)}$ in the loss function expression of Eq. (13) are given below:

$$o_i^{(l+1)} = \sigma\left(a_i^{(l+1)}\right); a_i^{(l+1)} = b_i^{(l+1)} + \sum\nolimits_{j=1}^{r_l} w_{ij}^{(l+1)} o_j^{(l)}. \tag{14}$$

where $a_i^{(l+1)}$, $o_i^{(l+1)}$ denote, respectively, the input and the output of the i^{th} neuron of the $(l+1)^{\text{th}}$ layer; σ is the neural activation function; $b_i^{(l+1)}$, $w_{ij}^{(l+1)}$ denote, respectively, the bias of the i^{th} neuron of the $(l+1)^{\text{th}}$ layer and the weight coefficient associated with the connection from the j^{th} neuron of the l^{th} layer to the i^{th} neuron of the $(l+1)^{\text{th}}$ layer; $o_j^{(l)}$ is the output of the j^{th} neuron of the l^{th} layer; and r_l is the number of nodes in the l^{th} layer. It is noted that the $(l+1)^{\text{th}}$ layer is the output layer.

From Eq. (13), it is seen that the loss (cost) is affected by the difference between the actual responses of the output layer neurons $\left(o_i^{(l+1)}; 1 \leq i \leq r_{l+1}\right)$ and the target (desired) responses at the

output layer neuron ($y_i; 1 \leq i \leq r_{l+1}$), which are known a priori for each of the training vectors $\vec{\boldsymbol{x}}$ applied to the input of the network. The training process of the neural network involves the adjustment or updating of the weights and biases of the neural network in Fig. 4, namely, $w_{ij}^{(k)}$, $b_i^{(k)}$, $1 \leq k \leq l+1$, $1 \leq i \leq r_{k+1}$, $1 \leq j \leq r_k$, to minimize the cost function of Eq. (13). It is noted that the weights and biases affect the cost function (loss function) through affecting $o_i^{(l+1)}$ and they have no effect on y_i.

The minimization of the cost function is achieved through the gradient descent process. The formulation of the gradient of the cost function involves derivation of the partial derivatives of the responses of the output-layer neurons $o_i^{(l+1)}$ with respect to the network's weights and biases. The expressions for the partial derivatives of the cost function with respect to the output layer weights and biases are obtained by applying chain rule as shown below:

$$\frac{\partial E}{\partial w_{ij}^{(l+1)}} = \frac{\partial E}{\partial o_i^{(l+1)}} \frac{\partial o_i^{(l+1)}}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial w_{ij}^{(l+1)}}, \frac{\partial E}{\partial b_i^{(l+1)}} = \frac{\partial E}{\partial o_i^{(l+1)}} \frac{\partial o_i^{(l+1)}}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial b_i^{(l+1)}}. \tag{15}$$

The partial derivatives on the right-hand sides of the expressions of Eq. (15) are given by Eqs. (16a)–(16c).

$$\frac{\partial E}{\partial o_i^{(l+1)}} = o_i^{(l+1)} - y_i. \tag{16a}$$

$$\frac{\partial o_i^{(l+1)}}{\partial a_i^{(l+1)}} = \sigma\left(a_i^{(l+1)}\right)\left(1 - \sigma\left(a_i^{(l+1)}\right)\right) = o_i^{(l+1)}\left(1 - o_i^{(l+1)}\right). \tag{16b}$$

$$\frac{\partial a_i^{(l+1)}}{\partial w_{ij}^{(l+1)}} = o_j^{(l)}, \frac{\partial a_i^{(l+1)}}{\partial b_i^{(l+1)}} = 1. \tag{16c}$$

The neural activation function is assumed to be the sigmoid of Eq. (2c), and its derivative is given in Eq. (17).

$$\sigma(x) = \frac{1}{1+e^{-x}}, \frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x)). \tag{17}$$

Substituting from Eq. (16) into Eq. (15) and using Eq. (17), one arrives at the expressions for the partial derivatives of the cost function with respect to the weights and biases of the output layer as shown in Eqs. (18a), (18b).

$$\frac{\partial E}{\partial w_{ij}^{(l+1)}} = \left(o_i^{(l+1)} - y_i\right) o_i^{(l+1)}\left(1 - o_i^{(l+1)}\right) o_j^{(l)}. \tag{18a}$$

$$\frac{\partial E}{\partial b_i^{(l+1)}} = \left(o_i^{(l+1)} - y_i\right) o_i^{(l+1)}\left(1 - o_i^{(l+1)}\right). \tag{18b}$$

where $o_i^{(l+1)}$ is the computed output of the i^{th} neuron of the $(l+1)^{\text{th}}$ layer; $o_j^{(l)}$ is the computed output of the j^{th} neuron of the l^{th} layer; and y_i is the target response (desired response) of the i^{th} neuron at the output layer. As noted earlier, the target response y_i is known a priori as components of the response vector to the training input $(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}})$. These outputs, namely $o_i^{(l+1)}$, $o_j^{(l)}$, however, are computed after passing the training vector $\vec{\boldsymbol{x}}$ through the feedforward network in Fig. 4. The partial derivatives of Eqs. (18a), (18b) are expressed in terms of a new parameter $\delta_i^{(l+1)}$ as shown by Eqs. (19a)–(19c).

$$\delta_i^{(l+1)} = o_i^{(l+1)}\left(1 - o_i^{(l+1)}\right)\left(o_i^{(l+1)} - y_i\right). \tag{19a}$$

$$\frac{\partial E}{\partial w_{ij}^{(l+1)}} = \delta_i^{(l+1)} o_j^{(l)}. \tag{19b}$$

$$\frac{\partial E}{\partial b_i^{(l+1)}} = \delta_i^{(l+1)}. \tag{19c}$$

Using gradient descent, the procedures for updating the weights and biases of the output layer are expressed by Eqs. (20a), (20b).

$$w_{ij}^{(l+1)} \leftarrow w_{ij}^{(l+1)} - \gamma \delta_i^{(l+1)} o_j^{(l)}. \tag{20a}$$

$$b_i^{(l+1)} \leftarrow b_i^{(l+1)} - \gamma \delta_i^{(l+1)}. \tag{20b}$$

where the parameter γ is a user-prescribed small number, i.e., $\gamma = 0.1$, called the learning rate. Eqs. (20a), (20b) provide a procedure for updating the output layer weights and biases after computing the response vector $\vec{\hat{y}} = \vec{o}^{(l+1)}$ of the network to the training input $\vec{x}$.

The derivations of the expressions of Eqs. (20a), (20b) for updating of the weights and biases of the output layer are straightforward, because the error terms at the outputs of the neurons of the output layer are known. As seen in Eq. (19a), $\delta_i^{(l+1)}$ can be computed directly, because y_i, which denotes the desired (target) output of the i^{th} neuron in the output layer, is known from the training set. This, however, is not the case for the hidden layers. The desired outputs of the neurons in the hidden layers are not known directly, which is the reason they are called hidden layers.

The error terms at the outputs of the neurons of the hidden layers are not directly known, because unlike the output layer, for which the target values of the outputs of the neurons are known a priori, the target values of the outputs of the neurons of the hidden layers are not known. To determine the partial derivatives of the cost function with respect to the weights and biases of the hidden layers, one must first determine the errors at the outputs of the neurons of the hidden layers. This is where the concept of back propagation originates. Knowing the errors at the outputs of the neurons of the output layer, the errors are propagated backward to compute the errors at the outputs of the neurons of the hidden layers, one layer at a time.

The backpropagation process is illustrated by formulating the partial derivative of the cost function of Eq. (13) with respect to a typical weight factor of the l^{th} layer, namely $w_{jp}^{(l)}$. As seen in Fig. 5, $w_{jp}^{(l)}$ denotes the weight coefficient associated with the connection between p^{th} neuron of the $(l\text{-}1)^{th}$ layer and the j^{th} neuron of the l^{th} layer. It is noted that $w_{jp}^{(l)}$, although affecting only one neuron in the l^{th} layer, namely the j^{th} neuron, it affects all the neurons in the output layer. This is because the j^{th} node of the l^{th} layer is connected to all the nodes of the output layer. Applying chain rule, the partial derivative of the coast function with respect to $w_{jp}^{(l)}$ is obtained as below:

$$\frac{\partial E}{\partial w_{jp}^{(l)}} = \sum_{i=1}^{r_{l+1}} \frac{\partial E}{\partial o_i^{(l+1)}} \frac{\partial o_i^{(l+1)}}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial o_j^{(l)}} \frac{\partial o_j^{(l)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{jp}^{(l)}}. \tag{21a}$$

$$\frac{\partial E}{\partial o_i^{(l+1)}} = o_i^{(l+1)} - y_i. \tag{21b}$$

$$\frac{\partial o_i^{(l+1)}}{\partial a_i^{(l+1)}} = o_i^{(l+1)}\left(1 - o_i^{(l+1)}\right). \tag{21c}$$

$$\frac{\partial a_i^{(l+1)}}{\partial o_j^{(l)}} = w_{ij}^{(l+1)}. \tag{21d}$$

$$\frac{\partial o_j^{(l)}}{\partial a_j^{(l)}} = o_j^{(l)}\left(1 - o_j^{(l)}\right). \tag{21e}$$

$$\frac{\partial a_j^{(l)}}{w_{jp}^{(l)}} = o_p^{(l-1)}. \tag{21f}$$

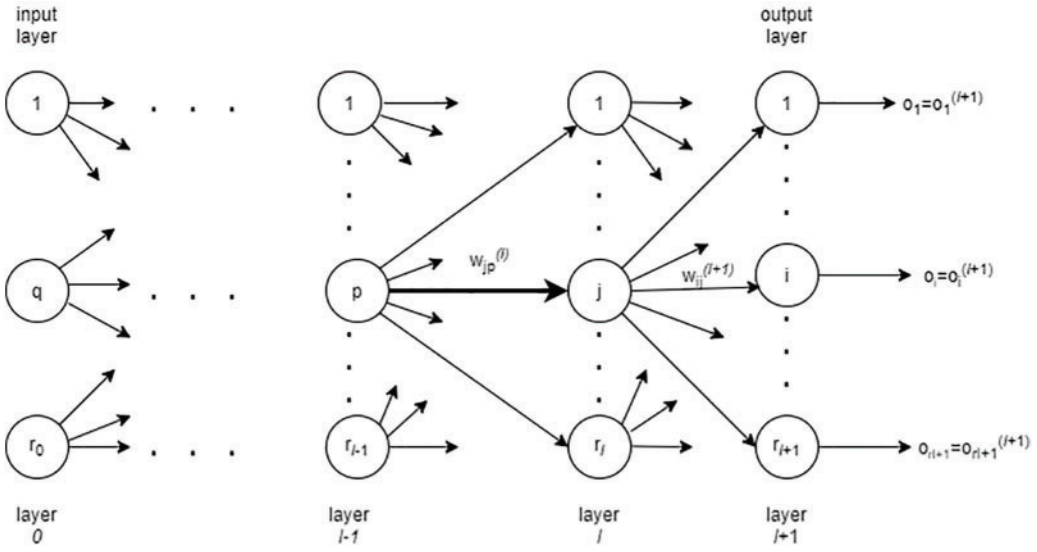

Figure 5: Illustration of backpropagation

Substituting Eqs. (21b)–(21f) in Eq. (21a) and rearranging terms one obtains the following:

$$\frac{\partial E}{\partial w_{jp}^{(l)}} = o_j^{(l)}\left(1 - o_j^{(l)}\right) o_p^{(l-1)} \sum_{i=1}^{r_{l+1}} \left[o_i^{(l+1)}\left(1 - o_i^{(l+1)}\right)\left(o_i^{(l+1)} - y_i\right)\right] w_{ij}^{(l+1)}. \tag{22}$$

Substituting from Eq. (19a) in Eq. (22), and rearranging terms leads to the following:

$$\frac{\partial E}{\partial w_{jp}^{(l)}} = \left[o_j^{(l)}\left(1 - o_j^{(l)}\right) \sum_{i=1}^{r_{l+1}} w_{ij}^{(l+1)} \delta_i^{(l+1)}\right] o_p^{(l-1)}. \tag{23}$$

The bracketed term in Eq. (23) is called the error at the output of the j^{th} neuron of the l^{th} layer, as given below:

$$\delta_j^{(l)} = o_j^{(l)}\left(1 - o_j^{(l)}\right) \sum_{i=1}^{r_{l+1}} w_{ij}^{(l+1)} \delta_i^{(l+1)}. \tag{24}$$

Eq. (24) shows how the error at the output of any layer can be obtained in terms of the error at the output of the next layer. This equation describes the backpropagation process, where the known error at the output of the network is propagated backward, one layer at a time, to obtain the error at the outputs of all the other layers including the hidden layers and the input layer. After computing all the $\delta_j^{(k)}$ terms for a typical layer, the $\frac{\partial E}{\partial w_{jp}^{(k-1)}}$ terms at preceding layer are computed, and this process

starts at the output layer and proceeds all the way to input layer, one layer at a time. Following the computation of all the partial derivatives (gradients) the change values for all the weights and biases are recorded as expressed by Eqs. (25a)–(25c).

$$\delta_j^{(k)} = o_j^{(k)}\left(1 - o_j^{(k)}\right)\sum_{i=1}^{r_{k+1}} w_{ij}^{(k+1)}\delta_i^{(k+1)};\ 1 \le k \le l,\ 1 \le j \le r_k. \tag{25a}$$

$$w_{ij}^{(k)} \leftarrow w_{ij}^{(k)} - \gamma\delta_i^{(k)}o_j^{(k-1)}. \tag{25b}$$

$$b_i^{(k)} \leftarrow b_i^{(k)} - \gamma\delta_i^{(k)}. \tag{25c}$$

6 Training the Neural Network

The values of the NN weights and biases are initially set randomly using a random number generator and a probability distribution function such as a normal distribution with zero mean and unit standard deviation. The training set is utilized to compute the NN weights and biases through the process of training the network, as described below.

For an NN classifier, the training set comprises a finite set of tuples, where each tuple consists of a feature vector and the corresponding class label comprising one training object. For example, the training set of a binary classifier comprises a set of feature vectors and the respective class labels, where each label can be either zero or one. The training vectors are applied to the NN classifier, one trainer at a time. The computed class label, which is the actual NN response to the feature vector representation of the training object, is compared to the true class label of the trainer. The true class label of the input feature vector, which is also called the ground truth, is the desired or target response.

According to the terminology in Section 5.2, the feature vector representation of the training object is denoted as $\vec{\boldsymbol{x}}$, which is applied to the NN input. The computed response (actual response) of the NN to the training object is denoted as $\vec{\boldsymbol{o}}^{(l+1)}$. The desired (target) response is the true class label of the training object and is denoted as $\vec{\boldsymbol{y}}$. The pair of vectors comprising the feature vector of the training object and the respective class ($\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$) comprise one training instant. For a binary classifier, the output layer of the NN has only one neuron, and the computed response is a scaler. The desired (target) response or ground truth is a binary number, zero or one, which is the true class label of $\vec{\boldsymbol{x}}$.

The process of updating the weights and biases of a typical binary classifier proceeds as follows. The training object feature vector $\vec{\boldsymbol{x}}$ is applied to the NN, and the scalar response of the network $o^{(l+1)}$ is computed. The computed (actual) response $o^{(l+1)}$ of the network and the true class label of $\vec{\boldsymbol{x}}$, which is the binary number y are compared. Eqs. (19a), (20a), (20b) are used to update the weights and biases associated with the last layer, which is the $(l+1)^{\text{th}}$ layer of the NN. The error term at the output of the $(l+1)^{\text{th}}$ layer, $\delta^{(l+1)}$ which is computed in accordance with Eq. (19a) is backpropagated, one layer at a time, to compute the error vectors $\vec{\boldsymbol{\delta}}^{(k)}$ at the output of every other layer using Eq. (25a). Eqs. (25b), (25c) are subsequently used to update the weights and biases at every layer. The procedure of computing the error vector at the output of the k^{th} layer, namely, $\vec{\boldsymbol{\delta}}^{(k)}$ and updating of the weights and biases of that layer $\vec{\mathbf{W}}^{(k)}, \vec{\mathbf{b}}^{(k)}$ is done one layer at a time, starting from the output layer, and proceeding to the left. The weight and bias updating procedure outlined above is repeated for every training object in the training set. Each time, one trainer is applied to the NN, the weights and biases are updated, and the trainer is set aside. Then the next trainer is applied, the NN weights and biases are updated, and so on, until all the trainers have been utilized and the trainer set is empty. This constitutes one training epoch. The procedure is repeated for a user-prescribed number of training epochs, for example one hundred. At the end of each epoch, the order of trainers in the training set is shuffled randomly. The

method of training described above is called stochastic gradient descent, and is the method used to obtain the experimental results reported in Section 7.

There are several other alternative training methods. In batch training, all the training elements are applied to the classifier, one trainer at a time as is done in stochastic gradient descent. After applying each trainer to the network, the actual network response to the trainer input is computed and is compared to the true class of the object as was done before. The backpropagation algorithm is used to compute and record the prescribed changes for all the weights and biases of the network without making any updates to the values of the weights and biases. After applying the entire training set to the NN and recording the prescribed changes of the weights and biases for each input, the change records across all trainers are combined and the weights and biases are updated at once at the end of the epoch. This process is repeated for a prescribed number of epochs.

The minibatch training method contains elements from both the stochastic training and the batch training. Here, multiple subsets of the training set each comprising a small number of trainers, for example one hundred trainers, are chosen randomly. Each randomly chosen subset is called a minibatch. Each of the randomly chosen minibatches are used to train the NN using the batch training method described above. There are several alternative ways to do minibatch training. In minibatch training without replacement, after a minibatch is utilized to train the NN, the trainers in the minibatch are removed from the training set, and the next minibatch is chosen from the remaining training set. In minibatch training with replacement, after a minibatch is utilized to train the NN, the trainers in the minibatch are put back into the training set, the set is reshuffled and the next minibatch is chosen and the training process continues.

7 Dataset

The multilayered feedforward neural network described in Section 4 is utilized as a classifier for cybersecurity applications. The NN is designed as a binary classifier to label the network traffic packets as either normal or an attack. The binary classifier is trained and tested using the KDD99 dataset, which has been widely utilized in cybersecurity studies. The KDD dataset is comprised of packet header data for 805,051 packets and is divided into two subsets, namely the training set and the test set. The training set comprises 494,022 packets, including 97,278 normal packets and 396,744 attack packets. The test set comprises 311,029 packets, including 60,593 normal packets and 250,436 attack packets. The attack packets in the training and test sets include four attack types: denial of service (DoS), probe attack, remote to local attack (R2L), and user to root attack (U2R). Each of the header packets in the training and test sets has forty-two attributes, including the packet label. Thirty-eight of the attributes are numerical, and three are categorical. In the work reported here, all the packets associated with the two main subsets of the KDD99 dataset, namely the training set and the test set, were combined into one dataset. The three non-numerical features of each packet header were dropped, leaving a dataset with 805,051 packets, where each packet has thirty-eight features and one binary label of normal or attack. The data set was then partitioned into two separate sets in accordance with the packets' binary labels. The normal packets were all grouped together into the normal set, which contains 157,871 packets. Likewise, the attack packets were grouped together into the attack set, which contains 647,180 packets.

In the experiments reported in Section 8, the input applied to the classifier is a typical network traffic packet header. Each input is represented by its feature vector, which has thirty-eight dimensions. Each input can have one of two possible class labels, namely normal or attack. Therefore, the NN binary classifier has thirty-eight nodes in the input layer and one node in the output layer. The

number of hidden layers and the number of nodes in each hidden layer are design parameters, that are determined through experimentation by trial and error heuristically. The training and test sets that are used to train the binary classifier and evaluate its performance were chosen from the normal and attack sets described in the previous paragraph.

8 Test Results

The packet classifier used in the experiments of this section is a multilayered feedforward neural network comprising an input layer with thirty-eight nodes, an output layer with one node, and l hidden layers. The hidden layers are numbered from one through l, and the number of nodes in each hidden layer is denoted as r_k, where $1 \leq k \leq l$ denotes the index of the hidden layer as shown in the block diagram of Fig. 4.

The NN training process involves a training set comprising *normal* and *attack* packets, which are randomly chosen from the *normal* and *attack* sets described in Section 7. In all of the experiments presented here, the number of *normal* and *attack* packets in the training set of each experiment are equal. Following the training process, the NN is tested by evaluating its performance using a test set. The test set also comprises *normal* and *attack* packets, which are randomly selected from the *normal* and *attack* sets of Section 7, after the removal of the training packets from each set. In all the experiments presented here, the number of *normal* and *attack* packets in the test set of each experiment are equal. Each of the packets of the test set is applied to the trained classifier, and the classifier computes the class label for each test packet. The computed class of each test packet is compared to the respective true class. If the true class of the input packet (ground truth) is *attack* and the computed class is also *attack*, this is called true-positive (TP). If the true class of the input packet is *attack* but the computed class is *normal*, this is called false-negative (FN). If the true class of the input packet is *normal* and the computed class is also *normal*, this is called true-negative (TN). If the true class of the input packet is *normal* but the computed class is *attack*, this is called false-positive (FP). These definitions are further illustrated by Eqs. (26a), (26b) and Table 1.

$$\mathrm{TP} + \mathrm{FN} = \text{total number of } \boldsymbol{attack} \text{ packets in the test set} \tag{26a}$$

$$\mathrm{TN} + \mathrm{FP} = \text{total number of } \boldsymbol{normal} \text{ packets in the test set} \tag{26b}$$

Table 1: Definitions of classifier performance parameters

		Computed class (assigned label)	
		Normal packet	Attack packet
Ground-truth or true class	Normal packet	**TN**	**FP**
	Attack packet	**FN**	**TP**

The true-positive rate (TPR) and the true-negative rate (TNR) denote, respectively, the percentage of *attack* and *normal* packets in the test set that are correctly labeled by the NN classifier. The false-positive rate (FPR) and the false-negative rate (FNR) denote, respectively, the percentage of *normal* and *attack* packets that are misclassified. The additional classifier performance parameters that are used in this report include classifier precision, recall, and F-score. The classifier precision denotes the true-positive rate (TPR) divided by the total number of test packets that are classified as *attack* packets by the classifier. The classifier recall is the true-positive rate (TPR) divided by the total number of *attack* packets in the test set. The classifier F-score is the harmonic mean of its precision and recall.

$$precision = \frac{TPR}{TPR + FPR}. \tag{27a}$$

$$recall = \frac{TPR}{TPR + FNR}. \tag{27b}$$

$$F - score = \frac{2 \times precision \times recall}{precision + recall}. \tag{27c}$$

The plots of Fig. 6 show the effect of the number of trainers on the classifier performance as measured by the five metrics defined above. Here, the classifier, in addition to the input and output layers, is comprised of one hidden layer with ten nodes. The number of test packets was set at two thousand, equally divided between *normal* and *attack* packets. None of the training packets were included in the test set, and the NN was trained using ten epochs with a batch size of one, namely, stochastic gradient descent training. For each setting of the number of trainers, the experiment was repeated twenty-five times, and the performance results were averaged across all the trials of the experiment.

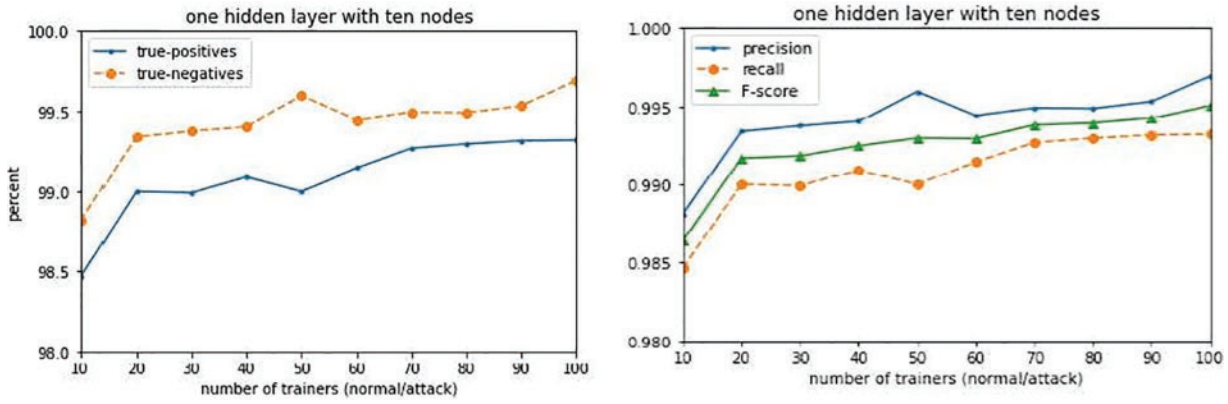

Figure 6: Effect of number of trainers on performance of classifier with one hidden layer

The box plots of Fig. 7 show the distributions of the true-positive rate and the true-negative rate across twenty-five instantiations of the experiment. The plots of Figs. 6 and 7 show that as the number of normal and attack trainers increases from ten to one hundred, the classifier performance improves as expected. This experiment shows that an FFNN classifier with one hidden layer comprising ten nodes, trained with two hundred packets equally divided between *normal* and *attack*, leads to precision exceeding 99.7%, which is a remarkable achievement. The plots of Fig. 7 also show that as the number of trainers increases, the statistical dispersion of the classifier performance parameters, namely, TPR and TNR, tightens across different instantiations of the experiment.

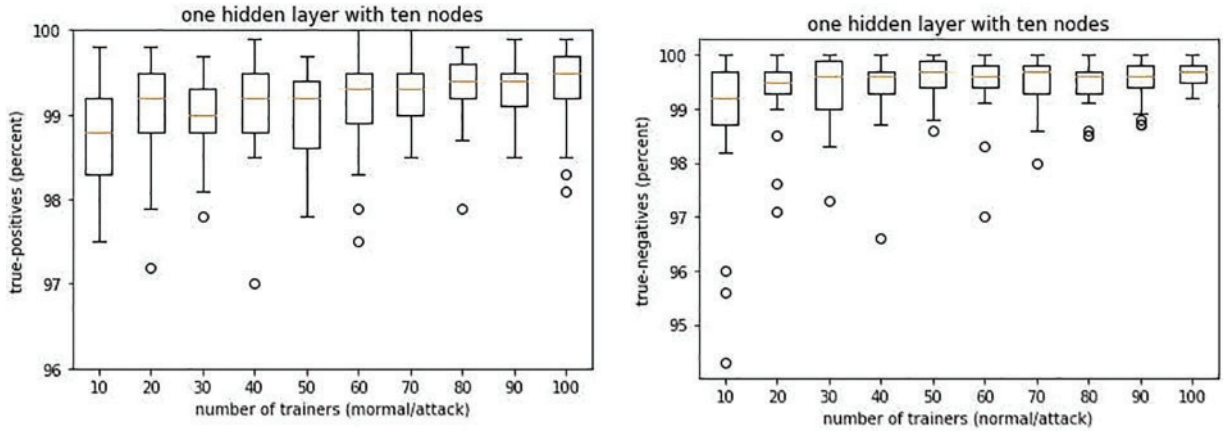

Figure 7: Effect of the number of trainers on distributions of TPR and TNR of the classifier with one hidden layer

The plots of Fig. 8 show the effect of the number of trainers on the performance of classifiers with different numbers of hidden layers. In these experiments, three different FFNN classifiers, namely, with one, two, and three hidden layers, were trained, and the performance of each trained classifier was assessed. Each classifier has ten nodes in each of its hidden layers. The numbers of normal and attack packets used in the training sets of different classifiers varied from one hundred to one thousand. As expected, as the number of trainers increases, the classifier's performance improves. The plot in the fourth quadrant of Fig. 8 shows that increasing the number of hidden layers does not appreciably improve the classifier performance, as measured by the F-score. The plots of Figs. 9 and 10 show the effect of the number of nodes in the hidden layer on the performance of a classifier with one hidden layer. The number of trainers from normal and attack classes was fixed at twenty, and two thousand test packets were equally divided between normal and attack packets. For each setting of the number of nodes, the experiment was repeated one hundred times, and the performance results were averaged across all the instantiations of the experiment. It is seen from Fig. 9 that performance, as measured by the F-score of the classifier, improves as the number of nodes increases from one to ten, where it reaches a plateau. Increasing the number of nodes beyond ten does not have an appreciable effect on the performance, as measured by the F-score. The plots of Fig. 10 show the distributions of true-positive and true-negative rates across all the one hundred instantiations of the experiment. As the number of nodes increases, there seems to be a tightening of the true negative rate, as shown by the box plot on the right of Fig. 10. This shows that the greater number of nodes improves the performance, as measured by the worst-performing classifier in the one hundred instantiations of the classifier in this experiment.

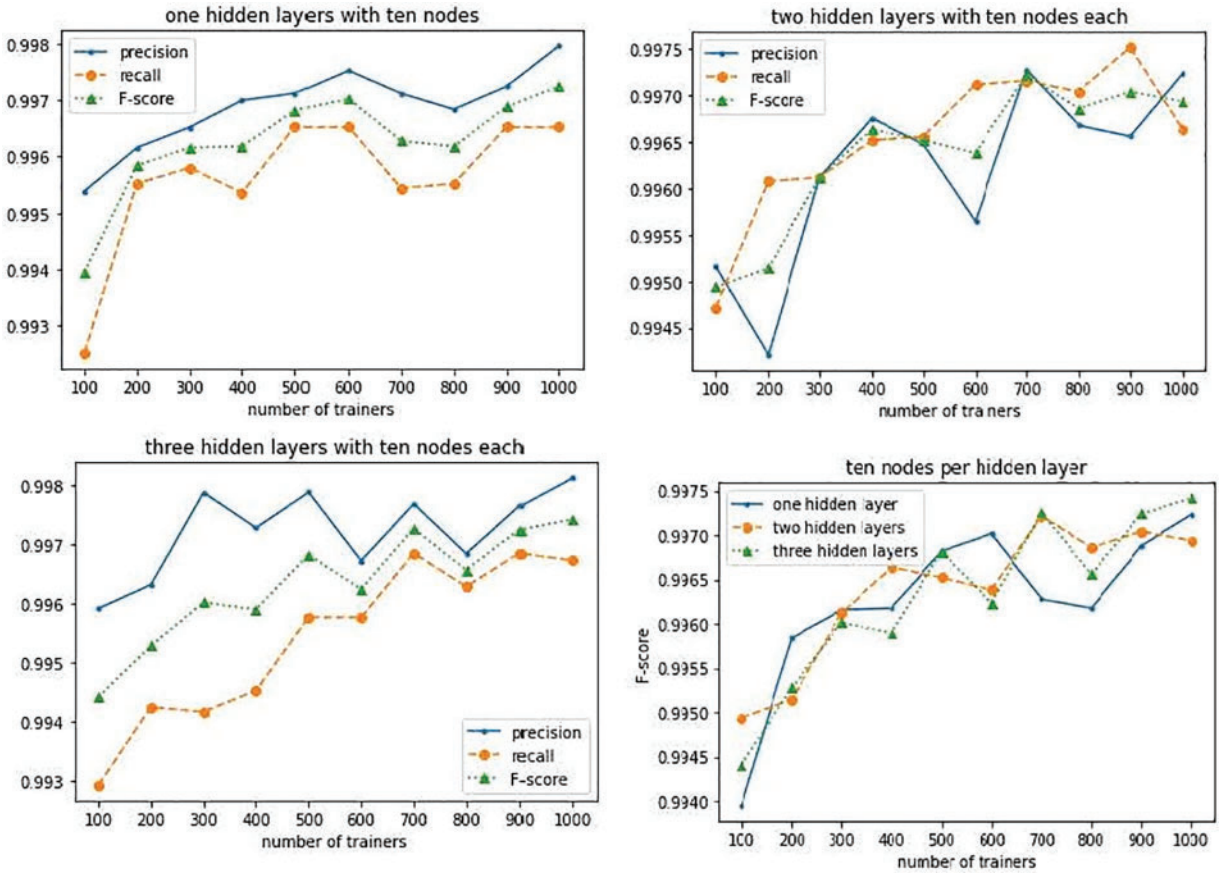

Figure 8: Effect of number of trainers on the performance of classifiers with different number of hidden layers

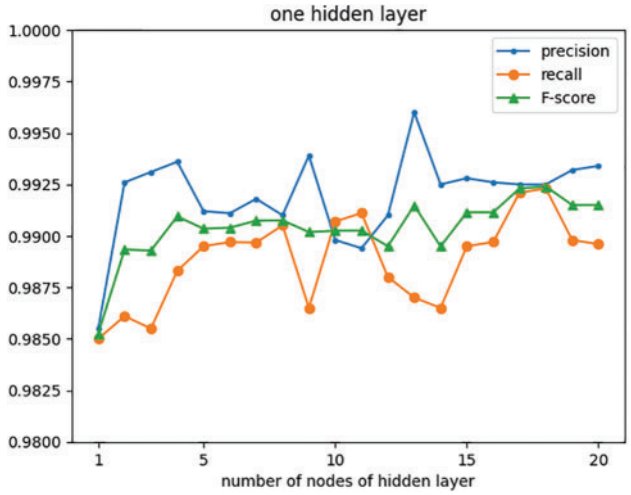

Figure 9: Effect of number of nodes of hidden layer on classifier performance

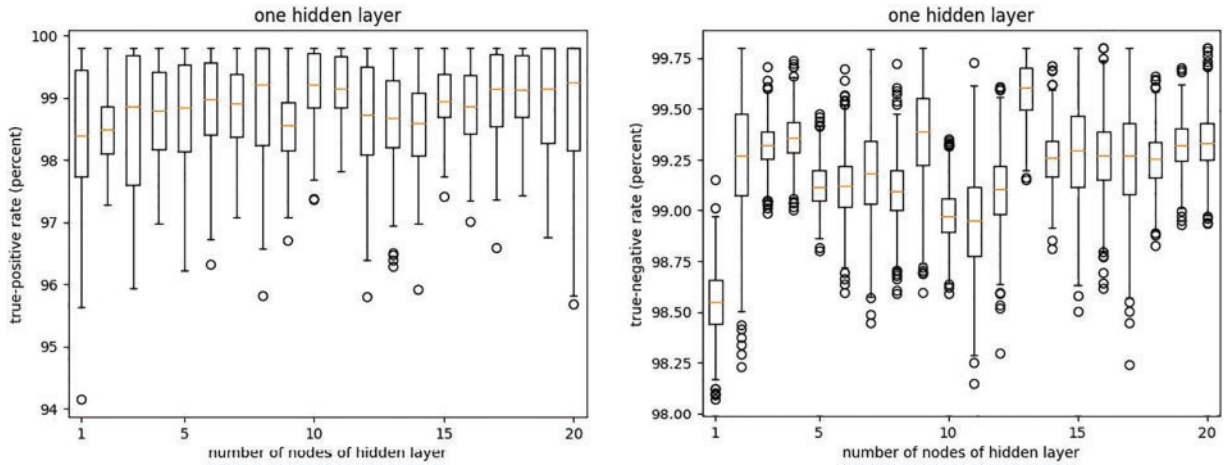

Figure 10: Effect of number of nodes of hidden layer on TPR and TNR distributions

This study compared the performance of three classifiers: a binary neural network (NN) with one hidden layer of ten nodes, a k-nearest neighbors (KNN) classifier with three neighbors, and a support vector classifier (SVC) with a linear kernel. All three machine learning algorithms were implemented using the Scikit-learn library. The classifiers were trained on the same data and evaluated on an identical test set consisting of 1000 packet headers from each class in the KDD dataset. For each setting of the number of trainers each experiment was repeated 100 times using randomly selected trainers, and the average performance across all repetitions is reported in Table 2. Among the three classifiers compared, the neural network (NN) demonstrated the highest performance for binary classification of network traffic packets, exceeding the k-nearest neighbor (KNN) and support vector classifier (SVC).

Table 2: Comparison of classification accuracy

		Number of trainers from each class				
		100	200	400	800	1000
Classifier type	NN	0.994	0.994	0.995	0.996	0.998
	SV	0.875	0.896	0.936	0.947	0.959
	KNN	0.912	0.928	0.925	0.936	0.965

9 Conclusions

This paper establishes the mathematical underpinnings of the fully connected feedforward neural network (FFNN), commonly known as the multi-layer perceptron (MLP), and elucidates the intricate formulations of the backpropagation and gradient descent algorithms employed for adjusting network weights and biases throughout the training phase. Comprehensive derivations of the mathematical formulas delineating the forward propagation from the input to the output of the neural network (NN) are presented, alongside an analysis of its computational complexity. The NN is applied to conduct a binary classification of network traffic utilizing a well-known open-source benchmark dataset. The paper evaluates the impact of different parameters, including network configurations such as the

number of hidden layers and the number of nodes within each hidden layer, as well as the number of trainers, on the performance of the classifier. The statistical distributions of the true-positive and true-negative rates of the classifier are analyzed across various experimental setups, which involve random selection of training and testing elements under different scenarios including variations in the number of hidden layers and nodes. Compared to the k-nearest neighbor (KNN) and support vector classifier (SVC), the study demonstrates that a neural network (NN) classifier achieves superior accuracy in binary classification of network traffic packets.

Acknowledgement: None.

Funding Statement: Kaveh Heidary's research project was partially funded by Quantum Research International Inc. through Contract QRI-SC-20-105. https://www.quantum-intl.com/.

Availability of Data and Materials: The network traffic data used in this paper is open-source and can be downloaded from the following source: https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html (accessed on 08/04/2024).

Conflicts of Interest: The author declares that they have no conflicts of interest to report regarding the present study.

References

[1] M. G. Solomon and D. Kim, "Evolution of communication technologies," in *Fundamentals of Communications and Networking*, 3rd ed. Burlington MA, USA: Jones & Bartlett Learning, 2022, pp. 1–26.

[2] B. A. Forouzan, "Network layer: Delivery, forwarding, and routing," in *Data Communication and Networking with TCP/IP Protocol Suite*, 6th ed. New York, NY, USA: McGraw Hill, 2022, pp. 647–699.

[3] P. Baltzan and A. Phillips, "Databases and data warehouses," in *Essentials of Business Driven Information Systems*, 5th ed. New York, NY, USA: McGraw Hill, 2018, pp. 169–208.

[4] National Institute of Standards and Technology (NIST), *Guide to Operational Technology (OT) Security*, NIST SP 800-82, Rev 3, Sep. 2023. Accessed: Apr. 5, 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r3.pdf

[5] "The White House National cybersecurity strategy," Mar. 2023. Accessed: Apr. 8, 2024. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2023/03/National-Cybersecurity-Strategy-2023.pdf

[6] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010. doi: 10.1016/j.comnet.2010.05.010.

[7] M. Ammar, G. Russello, and B. Crispo, "Internet of things: A survey on the security of IoT frameworks," *J. Inf. Secur. Appl.*, vol. 38, pp. 8–27, Feb. 2018. doi: 10.1016/j.jisa.2017.11.002.

[8] L. Chen, S. Tang, V. Balasubramanian, J. Xia, F. Zhou and L. Fan, "Physical-layer security based mobile edge computing for emerging cyber physical systems," *Comput. Commun.*, vol. 194, pp. 180–188, Oct. 2022. doi: 10.1016/j.comcom.2022.07.037.

[9] R. W. Lucky and J. Eisenberg, "National Academies Renewing U.S. telecommunications research," in *National Research Council of the National Academies, Committee on Telecommunications Research and Development*. Washington, DC, 2006. Accessed: Nov. 10, 2023. [Online]. Available: http://www.nap.edu/catalog/11711/renewing-us-telecommunications-research

[10] Y. Li and Q. Liu, "A comprehensive review study of cyber-attacks and cyber security: Emerging trends and recent developments," *Energy Rep.*, vol. 7, pp. 8176–8186, Nov. 2021. doi: 10.1016/j.egyr.2021.08.126.

[11] S. Ansari, S. G. Rajeev, and H. S. Chandrashekar, "Packet sniffing: A brief introduction," *IEEE Potentials*, vol. 21, no. 5, pp. 17–19, 2003. doi: 10.1109/MP.2002.1166620.

[12] M. A. Qadeer, A. Iqbal, M. Zaheed, and M. R. Siddiqi, "Network traffic analysis and intrusion detection using packet sniffer," in *2010 Second Int. Conf. Commun. Softw. Netw.*, Singapore, 2010, pp. 313–317. doi: 10.1109/ICCSN.2010.104.

[13] K. E. Hemsly and R. E. Fisher, "History of industrial control system cyber incidents," in *Idaho National Laboratory Report*, US Department of Energy, Dec. 2018.

[14] P. Gupta and V. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, pp. 24–32, 2001. Accessed: Jan. 15, 2024. [Online]. Available: https://cse.sc.edu/~srihari/reflib/GuptaIN01.pdf

[15] C. L. Hsieh, N. Weng, and W. Wei, "Scalable many-field packet classification for traffic steering in SDN switches," *IEEE Trans. Netw. Serv. Manag.*, vol. 16, no. 1, pp. 348–361, Mar. 2019. doi: 10.1109/TNSM.2018.2869403.

[16] A. S. Qureshi, A. Khan, N. Shamin, and M. H. Durad, "Intrusion detection using deep sparse auto-encoder and self-taught learning," *Neural Comput. Appl.*, vol. 32, no. 8, pp. 3135–3147, Apr. 2020. doi: 10.1007/s00521-019-04152-6.

[17] D. Ding, Q. L. Han, Y. Xiang, A. Ge, and X. M. Zhang, "A survey on security control and attack detection for industrial cyber-physical systems," *Neurocomputing*, vol. 275, pp. 1674–1683, Jan. 2018. doi: 10.1016/j.neucom.2017.10.009.

[18] R. Alguliyev, Y. Imamverdiyev, and L. Sukhostat, "Cyber-physical systems and their security issues," *Comput. Ind.*, vol. 100, pp. 212–223, Sep. 2018. doi: 10.1016/j.compind.2018.04.017.

[19] V. P. Janeja, "Understanding sources of cybersecurity data," in *Data Analytics for Cybersecurity*. Cambridge, UK: Cambridge University Press, 2022, pp. 14–27.

[20] I. H. Sarker, A. S. M. Kayes, S. Badsha, H. Alqahtani, P. Watters and A. Ng, "Cybersecurity data science: An overview from machine learning perspective," *J. Big Data*, vol. 7, no. 41, pp. 1–29, Jul. 2020.

[21] I. Goodfellows, Y. Bengio, and A. Courville, "Machine learning basics," in *Deep Learning*. Boston, MA, USA: MIT Press, 2016, pp. 98–155.

[22] C. C. Aggarwall, "Deep learning: Principles and learning algorithms," in *Neural Network and Deep Learning: A Textbook*, 2nd ed. Yorktown Heights, NY, USA: Springer, 2021, pp. 119–162.

[23] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for Boltzmann machines," *Cogn. Sci.*, vol. 9, pp. 147–169, Jan. 1985. doi: 10.1207/s15516709cog0901_7.

[24] Y. Bengio, "Deep learning of representations: Looking forward," in *Proc. First Int. Conf. Stat. Lang. Speech Process.*, Berlin, Heidelberg, Springer, May 2013, vol. 7978. doi: 10.1007/978-3-642-39593-2_1.

[25] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015. doi: 10.1038/nature14539.

[26] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 7586, pp. 504–507, Jul. 2006. doi: 10.1126/science.1127647.

[27] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, "VLSI implementation of deep neural networks using integral stochastic computing," *IEEE Trans. Very Large Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2688–2699, Oct. 2017. doi: 10.1109/TVLSI.2017.2654298.

[28] L. Alzubaidi *et al.*, "Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions," *J. Big Data*, vol. 8, no. 53, pp. 1–74, Mar. 2021. doi: 10.1186/s40537-021-00444-8.