SN Computer Science (2022) 3:145
https://doi.org/10.1007/s42979-022-01031-1

ORIGINAL RESEARCH

Deep Neural Network Based Real-Time Intrusion Detection System

Sharuka Promodya Thirimanne[1] · Lasitha Jayawardana[1] · Lasith Yasakethu[1] · Pushpika Liyanaarachchi[2] · Chaminda Hewage[3]

Received: 31 July 2021 / Accepted: 10 January 2022
© The Author(s) 2022

Abstract
In recent years, due to the rapid growth in network technology, numerous types of intrusions have been uncovered that differ from the existing ones, and the conventional firewalls with specific rule sets and policies are incapable of identifying those intrusions in real-time. Therefore, that demands the requirement of a real-time intrusion detection system (RT-IDS). The ultimate purpose of this research is to construct an RT-IDS capable of identifying intrusions by analysing the inbound and outbound network data in real-time. The proposed system consists of a deep neural network (DNN) trained using 28 features of the NSL-KDD dataset. In addition, it contains the machine learning (ML) pipeline with sequential components for categorical data encoding and feature scaling, which is used before transmitting the real-time data to the trained DNN model to make predictions. Moreover, a real-time feature extractor, which is a C++ program that sniffs data from the real-time network traffic and derives relevant data related to the features of the NSL-KDD dataset using the sniffed data, is deployed between the gateway router and the local area network (LAN). Together with the trained DNN model, the ML pipeline is hosted in a server that can be accessed via a representational state transfer application programming interface (REST API). The DNN has revealed outstanding testing performance results achieving 81%, 96%, 70% and 81% for accuracy, precision, recall and f1-score accordingly. This research comprises a comprehensive technical explanation concerning the implementation and functionality of the complete system. Moreover, leveraging the extensive explanations provided in this paper, advanced IDSs capable of identifying modern intrusions can be constructed.

Keywords Deep neural network · Real-time intrusion detection system · Machine learning · Pipelines

This article is part of the topical collection "Cyber Security and Privacy in Communication Networks" guest edited by Rajiv Misra, R. K. Shyamsunder, Alexiei Dingli, Natalie Denk, Omer Rana, Alexander Pfeiffer, Ashok Patel and Nishtha Kesswani".

✉ Lasith Yasakethu
lasithy@sltc.ac.lk

Sharuka Promodya Thirimanne
sharukat@sltc.edu.lk

Lasitha Jayawardana
lasithaj@sltc.edu.lk

Pushpika Liyanaarachchi
klpushpika@eng.pdn.ac.lk

Chaminda Hewage
chewage@cardiffmet.ac.uk

[1] Faculty of Engineering, Sri Lanka Technological Campus, Padukka, Sri Lanka

[2] Faculty of Engineering, University of Peradeniya, Peradeniya, Sri Lanka

[3] Department of Computer Science, Cardiff Metropolitan University, Cardiff, UK

Introduction

The internet has become the most significant resource in this century since it has become incorporated into our daily lives, assisting us in a variety of ways; however, because of its extraordinary popularity and accessibility, networks in the corporate and personal sectors are exposed to a range of manual and machine-generated attacks. Even though firewalls are designed to secure networks, they are incapable of detecting intrusions in real-time. As a result, destructive cyber-attacks pose severe security difficulties, necessitating the need for adaptable and reliable intrusion detection systems (IDS) capable of monitoring policy violations, malicious activity, and unauthorized access in real-time. Intrusion detection can be done in higher efficacy by employing ML algorithms since those have pattern identification capability utilizing the statistical modelling concept based on the past data. Therefore, the ultimate objective of this research is to implement a fully functional ML based RT-IDS capable

Published online: 29 January 2022

of predicting whether an intrusion or not based on the information captured from the inbound data packet in real-time.

Many studies have been conducted to assess the performance of various ML algorithms trained on the KDD99, NSL-KDD [1], and USNW-NB15 [2] datasets. However, a limited number of studies have been published based on experiments to construct a RT-IDS. However, most of the existing studies on RT-IDSs lack descriptive technical explanations. Consequently, that deficiency is addressed in this research. Moreover, most of the research that reveals performance comparisons between various state-of-the-art ML algorithms was done utilizing the Weka tool. The originality of our research is that all the algorithms were constructed using industry-utilized frameworks and libraries to demonstrate the performance of ML in real-world applications. The DNN was selected as the ML algorithm for this experiment utilizing the conclusion of the previous research that we have done on comparative algorithm analysis for ML-based IDS using six ML algorithms: DNN, support vector machines (SVM), K-nearest neighbours (KNN), one-class SVM (OCSVM), K-means and expectation–maximization (EM) [3].

The research reveals the experiment carried out to create a RT-IDS using an ML algorithm. In this experiment, a DNN was trained using the NSL-KDD dataset, which was created using the KDD99 dataset to overcome the inherent flaws such as redundant records [4]. Moreover, a real-time feature extractor, which performs packet sniffing and feature extraction from inbound and outbound data packets, is established between the gateway router and the local network. An ML pipeline, which consists of sequential components for categorical feature encoding and feature scaling together with the trained DNN, was developed to perform real-time intrusion prediction. Furthermore, the real-time prediction system is hosted in a server connected to the local network via an application programming interface (API). This system is capable of predicting whether or not a network state represents an intrusion based on the data extracted by the data packets. In addition, this system's uniqueness is that the real-time prediction system is hosted in a server that can be accessed using an API, enabling both corporate and personal networks to utilize this system for preserving their network from external attacks.

Overall, this study has contributed by introducing a descriptive approach to the RT-IDS, which contains an ML pipeline together with a fully trained DNN utilizing the NSL-KDD dataset. Since the methodology and the performance of the RT-IDS have been discussed, advanced RT-IDSs with DNNs, trained using datasets with modern intrusion types can be developed quickly. Moreover, the intrusion prediction ML pipeline is hosted in a server; therefore, anyone can access this system and integrate it with their local network.

The structure of this paper is organized as follows. Related works are presented in "Related Work", and "Problem Statement" includes the problem statement. Moreover, "Background" contains the background, and subsequently, the system model of the experiment is in "System Model". "Methodology" explains the methodology, and the performance evaluation is addressed in "Simulations Results". "Performance Comparison" and "Future Works" comprise the discussion and the conclusion accordingly. Finally, the future works are stated in "Conclusion".

Related Work

This section contains a collection of recent studies and experiments on RT-IDS. However, majority of the studies on IDS have been done performing benchmarking on different ML algorithms using various datasets. Therefore, limited number of studies are available on RT-IDSs. Hayoung et al. have proposed a real-time intrusion and anomaly detection system based on self-organizing map (SOP). It classifies neurons as 'normal' or 'attacks", and once an attack is identified, it categorizes according to the relevant attack type. Moreover, they have used two subsets of the KDD99 dataset for training and testing purposes [5]. However, this work lacks descriptive technical explanations on real-time data capturing.

Sangkatsanee et al. have experimentally demonstrated that the decision tree (DT) technique outperforms Ripper Rule, back-propagation neural network (BPNN), Bayesian network (BN), Naïve Bayes (NB), and radial basis function neural network (RBF-NN). Moreover, the DT algorithm-based RT-IDS can classify inbound data as normal or attack with a detection rate higher than 98%. The DT algorithm was trained using the RLD09 (Reliability Lab Data 2009) dataset [6]. Furthermore, they have extracted 12 features and the information gain method has been used for feature selection. Post-processing methods for lowering the false alarm rate was used and have shown that the RT-IDS is efficient in detection rate and memory utilization and can categorize the incoming network data within 2 s [7].

A team of researchers in [8] have developed an RT-IDS capable of detecting intrusions for network traffic with a higher precision. It comprises four modules: network data acquisition, data pre-processing, convolutional neural network (CNN), and intrusion detection. The CNN was trained using the NSL-KDD dataset. In data pre-processing, one-hot encoding and feature normalisation were used for categorical data encoding and feature scaling accordingly. The RT-IDS can capture network data in using TCPDUMP, which uses the LIBPCAP library to capture data from the network layer. In addition, an open-source tool called Bro was used

to analyse and segment the inbound data packets based on predefined series of Bro rule scripts. This implementation provides real-time network monitoring capability to detect abnormal network behaviour.

Zhang et al. [9] have introduced a novel framework design, which consists of five modules: pre-processing, autoencoder (AE), database, classification, and feedback. The proposed framework was evaluated using the CIC-IDS2017 dataset. The sparse autoencoder (SAE) was used to handle dimensionality by eliminating unimportant features. The random forest (RF) was used as the main supervised classification algorithm in this framework. To make comparisons with previous work, the accuracy of binary classification and multiclass classification were utilized as experimental outcomes. Researchers obtained promising results in their evaluation, with an accuracy of 0.9992 for binary classification and 0.9990 for multiclass classification [9].

A group of researchers in [10] have proposed an intrusion detection approach based on deep SAE and self-taught learning. Through unsupervised learning, the deep SAE technique has been used to extract features effectively. The deep SAE trained on regression-related tasks was utilized to extract features from the NSL-KDD dataset. Even though the source task does not have similar data distribution as the target domain, both domains are related to each other by the time-series nature of input features and unpredictable behaviour. Finally, the NSL-KDD dataset features and the extracted features were fed as an input to train the SAE. They have experimentally proved that the SAE trained utilizing a combination of original and extracted features outperforms the SAE trained only on original features.

Karbir et al. [11] have proposed a network intrusion detection framework based on a Bayesian network using a wrapper approach. The proposed system eliminates irrelevant features using genetic algorithm feature selection techniques, and a Bayesian classifier is employed as the base classifier to identify attack types. The performance has been evaluated using the NSL-KDD dataset and has achieved an accuracy of 98.2653%, outperforming algorithms such as KNN, Boosted DT, Hidden NB, and Markov Chain [11].

Below table illustrates the summary of previous works related to intrusion detection (Table 1).

Problem Statement

The existence of numerous sorts of threats have necessitated a system capable of defending systems and networks. Although a firewall can accept, discard, or deny inbound data packets based on the ruleset, it is incapable of identifying intrusions. However, even though extensive research has been done to analyse the performance of various ML algorithms based on the existing datasets, a viable RT-IDS, which can identify intrusions, is not yet developed. The comparative analysis we performed between six ML algorithms to identify the optimum ML algorithm for an IDS during our previous study [3], the scarcity of studies related to RT-IDSs developed using deep learning approaches, which can make predictions by analysing network traffic in real-time, and the unavailability of a fully featured RT-IDS, which can be implemented in any system or used as a software-as-a-service (SaaS), stimulated the motivation for this research. This research aims to develop a DNN based RT-IDS utilising the NSL-KDD dataset to address the above challenge.

Background

This section contains the background information of the dataset, technologies, and methods used in this research: NSL-KDD dataset, DNN, ML pipeline, and packet sniffing.

NSL-KDD Dataset

The dataset availability for intrusion detection is rare because most datasets cannot be shared due to various security and privacy concerns. The NSL-KDD dataset, on the other hand, provides open access to the entire dataset and was developed to overcome the inherent problems of the KDD99 dataset, which was developed based on the data captured in DARPA'98 [1]. Even though KDD99 has been used in many research studies,

Table 1 Summary of previous related works

Ref	Methods	Dataset	Achievement
[5]	SOP	KDD99	Misclassification rate of 0.05 with 0.5 s prediction time
[6]	DT	RLD09	Detection rate higher than 98% with 2 s prediction time
[8]	CNN	NSL-KDD	Intrusion detection for network data streams with higher detection precision
[9]	RF	CICIDS-2017	Accuracy of 0.9992 for binary classification and 0.9990 for multiclass classification
[10]	SAE	NSL-KDD	SAE trained using both original and extracted features outperforms SAE trained only on original features
[11]	Bayesian network	NSL-KDD	An accuracy of 98.2653% outperforming KNN, Boosted DT, Hidden NB, and Markov Chain

there are several advantages when using the NSL-KDD dataset. The ML classifier will not be biased towards classes with frequent records due to the elimination of duplicate data. Since the selected record count from each difficulty-level group is inversely proportional to the percentage of records in the KDD99 dataset, the classification rates of various ML algorithms vary, allowing the accuracy of multiple learning approaches effective. Moreover, the test set duplicate records were totally removed and, even though NSL-KDD is substantially smaller than KDD99, the number of records in the training sets is adequate to train an ML algorithm.

There are a few disadvantages when using the NSL-KDD dataset, such as inadequate documentation outlining the calculation mechanisms used to derive the features and containing obsolete data. Therefore, this dataset demonstrates less productivity while designing a modern commercial-level application. The NSL-KDD dataset is 52.3 MB in size and includes two separate datasets for training and testing. The table below shows the number of records in each dataset, as well as the number of records associated with each attack type (Table 2).

Furthermore, UNSW-NB15 and CICDS2017 are two other datasets available for intrusion detection. However, the UNSW-NB15 dataset contains a considerable number of duplicate records, and the elimination of duplicate records reduces the number of records available for training. Moreover, the CICDS2017 dataset suffers from a class imbalance problem, which leads to biasing the ML model towards the majority class.

Deep Neural Network

The artificial neural network (ANN) is concept developed based on the biology of the human brain [12]. Because neural network (NN) can generate any decision boundary classification in feature space, they can operate as nonlinear discriminating functions [13]. In recent years, the use of DNN in the domain of intrusion detection has been a prominent research focus, and it is an effective method that emerged from the shallow neural network (SNN). DNN is superior at modelling or abstracting representations and can simulate exceedingly complicated models. DNN has enormous potential for achieving effective data representation to build useful solutions. The above-mentioned facts and the comparative analysis carried out between six ML algorithms, classified under supervised, semi-supervised, and unsupervised learning in our previous study [3] led us to employ DNN for the proposed method.

The DNN produces outputs based on the weights applied to the connections and the related activation functions of the neurons and it is made up of numerous processing layers [14]. The proposed approach trains the DNN using the NSL-KDD dataset, resulting in higher classification accuracy.

Machine Learning Pipeline

Manual data transformation prior to training ML algorithm is ineffective and impractical for real-time commercial-level applications. ML workflow of data transformation and correlating the data into the model can be automated using the ML pipelines. The efficiency and the simplicity of building ML models will be increased by utilizing ML pipelines since the redundant tasks associated with the workflow will be eliminated. ML pipeline is an aggregate of five essential tasks associated with the ML workflow namely, data ingestion, cleaning, pre-processing, model validation and deployment. Since ML pipelines are not one-way and iterative behavioural capabilities of those, improve the performance scores of the ML algorithms [15].

Packet Sniffing

Packet sniffing is a technique for intercepting data packets as they travel across a network. Because data passes via the network in the form of packets, packet sniffing tools can swiftly capture the data packets. Packet sniffing applications are known as packet sniffers, and they can read packets that pass through the network layer of the TCP/IP layer. The packet sniffing applications are divided into two categories based on their intended use. Commercial packet sniffers are used by network administrators to monitor and validate network traffic, whereas underground packet sniffers are used by individuals' who sniff other people's personal and sensitive data for personal benefit. Packet sniffing tools are commonly used for monitoring network traffic, troubleshooting communication issues, assessing network performance, extracting usernames, and identifying network intruders [16].

System Model

The system model of the research is shown in Fig. 1. The functional block diagram illustrates the overall flow of the entire system. The Linux environment is installed inline between the organization's network and the gateway router.

Table 2 Details of normal and attack data

	Total	Normal	DoS	Probe	U2R	R2L
Train	125,973	67,343	45,927	11,656	52	995
Test	22,544	9711	7458	2421	200	2654

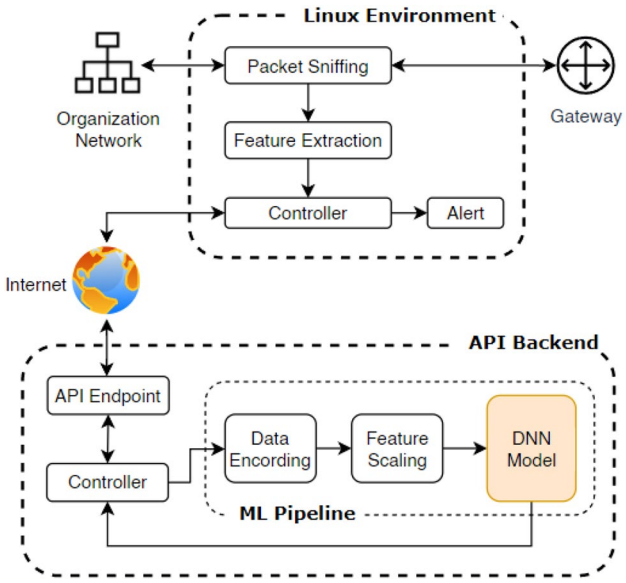


Fig. 1 Functional block diagram

The network traffic flowing through the Linux environment will then be sniffed by the packet sniffing method. The feature extraction then extracts the features from the data that was sniffed by the packet sniffing technique. Then the Linux environment's controller arranges the extracted data as a feature array and sends it as a hypertext transfer protocol (HTTP) request via the internet to the API endpoint of the API backend. The API backend controller extracts the data from the HTTP request and feeds it into the ML pipeline. The ML pipeline consists of two components: categorical data encoding, which transforms categorical data into numerical values, and feature scaling, which scales the entire dataset to a standard scale. Once the data pre-processing is completed, the pre-processed data is fed into the trained DNN model. Following that, the API backend controller returns the prediction result as an HTTP response for the corresponding HTTP request. Finally, the Linux environment controller alerts the network administrator if the HTTP response contains an anomaly.

Methodology

The implementation of this research was conducted under eight methodological steps: data pre-processing, DNN implementation, ML pipeline development, API endpoint development and documentation, ML integration, API deployment, network configuration and feature extraction. ML Development.

Table 3 Selected attributes for training the DNN

No	Feature Name	No	Feature Name
1	duration	15	srv_rerror_rate
2	protocol_type	16	same_srv_rate
3	service	17	diff_srv_rate
4	flag	18	srv_diff_host_rate
5	src_bytes	19	dst_host_count
6	dst_bytes	20	dst_host_srv_count
7	land	21	dst_host_same_srv_rate
8	wrong_fragment	22	dst_host_diff_srv_rate
9	urgent	23	dst_host_same_src_port_rate
10	count	24	dst_host_srv_diff_host_rate
11	srv_count	25	dst_host_serror_rate
12	serror_rate	26	dst_host_srv_serror_rate
13	srv_serror_rate	27	dst_host_rerror_rate
14	rerror_rate	28	dst_host_srv_rerror_rate

Data Pre-processing

Data pre-processing is the initial step that should be performed before feeding the data into the ML model. The tasks are feature selection, categorical data encoding and feature scaling.

Feature selection: The NSL-KDD dataset's 41 attributes are classified into three categories. They are basic, content, and traffic features. Without inspecting the payload, the basic features can be derived from the packet headers. The time interval is used to calculate traffic features. Domain expertise is required, however, to assess the payload of the packet to derive content features [10]. Furthermore, the NSL-KDD dataset authors have not explicitly stated how to derive the content features from the packets. Due to the difficulty of deriving features from the payload in real-time, the DNN model was trained using the remaining 28 features while excluding the 13 content features. The below table reveals the selected attributes for training the ML algorithm (Table 3).

Categorical data encoding The one-hot encoding (OHE) was adopted to perform categorical data encoding since ML algorithms achieve best performance when numerical values are used. Instead of integer encoding, OHE was used because if the nominal categorical data were encoded using integer encoding, an ordered numerical list would be created, which would mislead the ML algorithms by assigning irrelevant importance to the values based on their magnitude. The shortcoming of the OHE is that it creates a new column for each category, resulting the "curse of dimensionality." Therefore, the categories with the lowest frequency were combined into a single category. Table 4 shows the category count before and after grouping.

Table 4 NSL-KDD, before and after category reduction

Name	No. categories before	No. categories after
Protocol_type	3	3
Service	70	25
Flag	11	11

Table 5 Neuron count in each layer

Layer no	Neurons
1	64
2	160
3	352
4	320
5	448
6	384
7	192
8	224
9–16	32

After the category reduction phase is completed, OHE is undertaken for categorical features using the 'OneHotEncoder' function in the Scikit-Learn library.

Feature scaling the feature scaling concludes the data pre-processing, and it is employed to transform the numerical values of the complete dataset to a standard scale. Furthermore, the Standardization method is a scaling mechanism capable of rescaling the attributes to zero mean and the distribution with unit standard deviation. Equation 1 demonstrates the standardization equation, which was used for feature scaling.

$$X_{\text{standardized}} = \frac{X - \text{mean}(X)}{\text{standard deviation}(X)}. \quad (1)$$

For feature scaling, the 'StandardScaler' function of the Scikit-Learn [17] library was used.

Deep Neural Network Implementation

The DNN was built using the Keras, which is an open-source software library that is used for developing ANN. The DNN consists of 16 layers (excluding output layer) with different number of neurons in each hidden layer. Table 5 depicts the number of neurons associated with each hidden layer.

Several hyperparameters are associated in DNNs, which should be predetermined, that have a direct impact on the performance of the final model, such as the number of hidden layers, the number of neurons, activation function, weights initializer, bias initializer, learning rate, regularisation coefficient, and the optimizer. In the DNN model, the input layer and all the hidden layers were activated using ReLU (Rectified Linear Unit) function. The Eq. 2 depicts the ReLU activation function and it is a piecewise linear function and when the input is positive it directly output the input, otherwise, the output will be zero. The nodes which are activated using this function is referred as a rectified linear activation unit [18].

$$\text{Re}LU(x) = \max(0, x). \quad (2)$$

The output layer was activated using the Sigmoid function, which can map any real value to the range (0,1). This function converts the output of the DNN network into a probability score. The Eq. 3 depicts the equation of the Sigmoid function.

$$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}. \quad (3)$$

The initialization of the weights and the biases is crucial since improper initialization may lead to gradient exploding or vanishing phenomena. Thus, when the initialization is too large, it leads to exploding gradients, while too small initialization leads to vanishing gradients. In order to avoid the above phenomenon, the activations should have zero mean, and the variance should be constant across every layer [19]. Therefore, the weights of the layers which were activated using the *ReLU* [20] function were initialized using *He Uniform* initializer, and the output layer was initialized using the *Glorot Uniform* initializer [21]. Moreover, the TensorFlow-based Keras initializer functions called *HeUniform* and *GlorotUniform* functions were used for weight initialization. The bias initialization of all the layers was performed using the Zero initializer.

In the DNN, stochastic gradient descent (SGD) was used as the optimizer with a learning rate of 0.001. The binary cross-entropy was used as the loss function, and it is capable of estimating the loss of the model, which weights of the DNN can be updated accordingly to reduce the loss on the subsequent evaluation. Moreover, it is used for binary classification problems where the target values are in the set {0,1}. The DNN model was trained after executing it for 100 epochs, and the cross-validation technique was used to identify the performance of the model for untrained data. In addition, it is possible to identify whether the model is overfitting or underfitting by analysing the training and cross-validation accuracy curves. Therefore, to avoid overfitting, the 'early stopping' method was utilized. Hyperparameter tuning is required to enhance the performance of the ML model. Furthermore, the *Keras Tuner* library was used to tune the hyperparameters: the number of hidden layers, the number of neurons in each hidden layer, regularization coefficient, and learning rate.

Once the training is being done, the DNN machine learning model is saved in a JavaScript Object Notation (JSON)

file, which is a text format for data storage and transportation; moreover, the weights of the DNN are saved in a hierarchical data format 5 file (H5), which saves data in the hierarchical data format (HDF). Even though the ML model is saved to retrieve back whenever predictions are being made, the predicting process will be interrupted due to a dimension mismatch incurred while ingesting the data into the ML model. Therefore, data should be pre-processed in the exact same format in which the data pre-processing is done in the training stage. Consequently, the state of the 'ColumnTransformer' is also saved in a PICKLE file, which converts a Python object into a character stream and saves it on the disk.

Machine Learning Pipeline Development

The developed ML pipeline predominantly includes two sequential components: column transformer and the trained DNN model. Moreover, the column transformer is an aggregate of one-hot encoder and the Standard Scaler, which are used for categorical data encoding and feature scaling accordingly. The below figure illustrates the architecture of the ML pipeline built for real-time data transformation and prediction.

Before initiating the real-time prediction process, the pre-trained DNN model and the saved *Column Transformer* files are ingested into the ML pipeline as sequential components. Firstly, the real-time data extracted from the inbound traffic will be fed into the ML pipeline, and then the 'ColumnTransformer' performs OHE on three predetermined columns. Following that, feature scaling is performed using the *Standard Scaler* on all columns with decimal values. Upon the completion of data pre-processing, those data will be fed to the trained DNN, and the predictions will be made based on experience (Fig. 2).

When using a pipeline, there are various benefits. The pipeline used to train the ML model can be used to pre-process the test dataset and test the trained model. Moreover, the pipeline can process continuous streams of network traffic data in real-time.

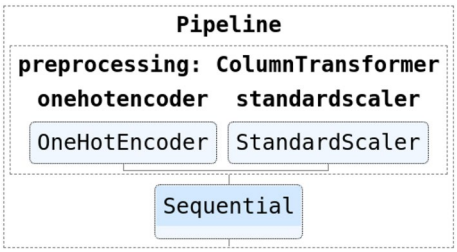


Fig. 2 Pipeline architecture

API Development

API Endpoint Development and Documentation

The API serves as the backend of the real-time prediction system and was constructed with Flask [22]. Furthermore, the basis for utilizing Flask is that it does not require any specialized tools or libraries; hence, it is regarded as a microframework. One end of the communication channel is known as the API endpoint. The backend server is one of the RESTful API's endpoints, which is identifiable by the URL: host/API/V2. In this system, an HTTP request will be generated using the Get () method, containing a comma-separated string comprising a subset of the features space of the NSL-KDD dataset. When the URL is called back along with the query, a response will be returned indicating whether or not there is an anomaly via the API endpoint. API documentation is a technical explanation that offers instructions on how to use and interact with an API, such as procedures for calling the API, the format of the returning response from the API, and different response formats dependent on the error type. The Swagger documentation framework was used to produce the API documentation, which is viewable via the "host/." URL.

ML Integration

A microservice is launched with Flask, and all routing pathways were configured. To run the ML pipeline, three files must be added to the webserver. As a result, in the ML integration section, the JSON file of the ML model, the H5 file containing the DNN weights, and the PICKLE file of the ColumnTransformer including the categorical data encoder and the feature scaler were imported using Flask. In addition, the Python file containing the ML pipeline code is loaded at the same time. The trained ML model is in standby mode whenever the Flask server is active, and when an API request is made, an immediate response is transmitted to the client via the API endpoint.

API Deployment

The deployment phase of the Flask application required a production level server at the end of its development. As a result, the Gunicorn server [23], a Python Web Server Gateway Interface (WSGI), was used. Furthermore, for portability between platforms, the Gunicorn and Flask applications were containerised using Docker technology. However, to provide accessibility during the development phase, the Nginx web server was used, and it was built into

a single container. Following that, the Nginx container and the container containing the Gunicorn and Flask application were combined into a single docker file. Consequently, this system can be executed in any environment using Docker technology. Finally, the docker file containing both containers was executed on a Linux server.

Real-Time Feature Extraction

Network Configuration

The initial stage of implementing real-time feature extraction is the network configuration. A Linux workstation with two network interfaces was installed between the gateway router and the LAN inline to the data connection to sniff the inbound and outbound data packets. Furthermore, by using the Linux machine configurations the two network interfaces were bridged virtually, then it will capture inbound and outbound data packets that flow via the Linux workstation utilising the packet sniffing mechanism developed in the C++ programming language.

Feature Extraction

The Packet Sniffing method was utilized for feature extraction, and it is a low-resource consuming application that can be deployed on any machine on a network, which consists of two or more network interfaces. C++ programming language was used to code the packet capturing system using the LIBPCAP package. Moreover, LIBPCAP is a library that provides a high-level API for capturing network traffic. Then the C++ application captures the packet as a string buffer. In addition, the packet analysis function was developed to process packets with some protocols such as ICMP, TCP, and UDP.

Cost Analysis

The performance of DL algorithms improves as the amount of data increases. However, as the amount of data increases, the performance of most conventional ML methods decreases. When employing DL, the advantage of performance improvement can be used for complex problems. However, it requires a very large volume of data (> 100,000) to outperform many ML algorithms. Furthermore, the proposed DL approach necessitates a significant amount of processing power when compared to other conventional ML algorithms, and the training time is significantly longer. The hardware and software utilized for training the ML algorithms are depicted in the below table (Table 6).

Based on the specifications mentioned above, the proposed DNN consumed approximately 30 min to complete the training process. In addition, the hyperparameter tuning process of the proposed DNN was time-consuming due to the larger number of hyperparameters associated with DNN.

Table 6 Hardware specifications and utilized libraries

CPU	Intel(R) Xeon(R) CPU @ 2.30 GHz
Memory	12.00 GB
Platform	Google Colab
Libraries	Pandas, TensorFlow, Keras, Scikit
Language	Python

Simulations Results

The DNN model's performance was evaluated using the accuracy, loss, precision, recall, f1-score, and confusion matrix (CM) together with the curves illustrated below. The below Figs. 3, 4, 5 and 6 were obtained using the TensorBoard, the TensorFlow visualization toolkit to demonstrate the behaviour of the accuracy, loss, precision, and recall of

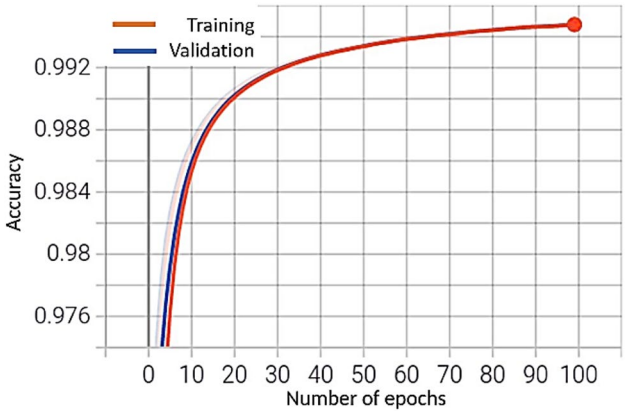


Fig. 3 Accuracy curves of training and validation set

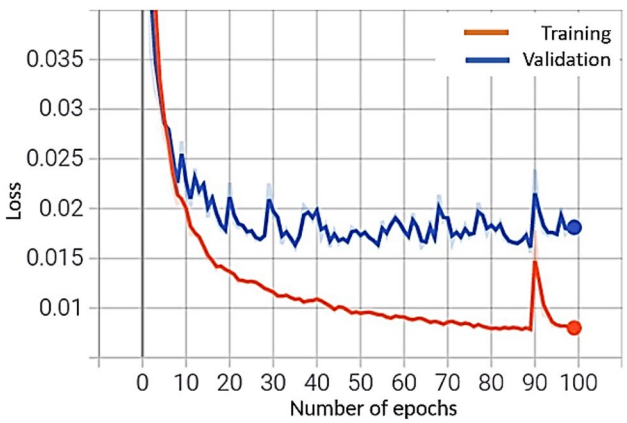


Fig. 4 Loss curves of training and validation set

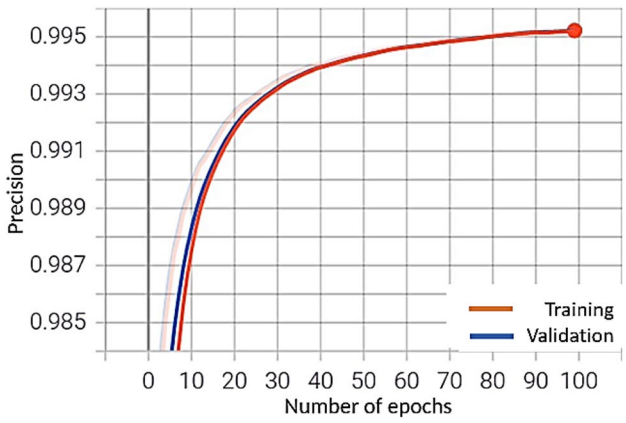


Fig. 5 Precision curves of training and validation set

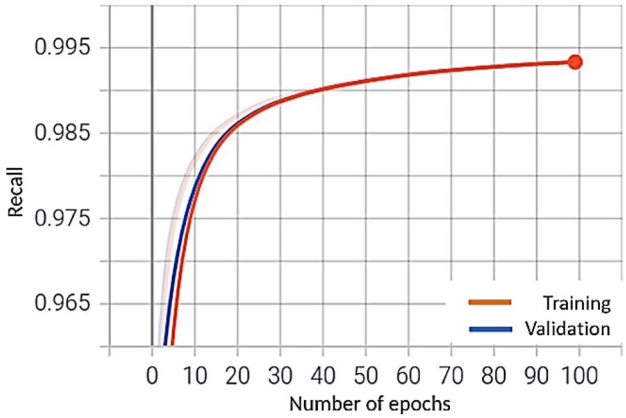


Fig. 6 Recall curves of training and validation set

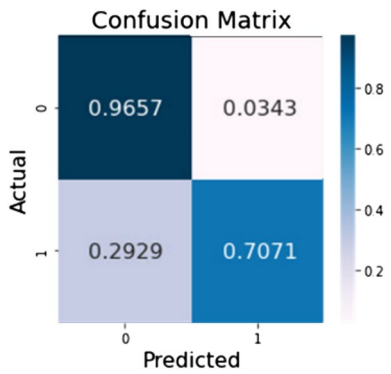


Fig. 7 Normalized confusion matrix of testing dataset

the training and cross-validation sets with the number of epochs (Fig. 7).

Precision, recall, and F1-score depends on the number of predicted true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN). These performance indicators are particularly effective when analysing the performance when the class distribution is skewed.

- TP: predicts 1 and the actual class is 1.
- FP: predicts 1 and the actual class is 0.
- TN: predicts 0 and the actual class is 0.
- FN: predicts 0 and the actual class is 1.

The above figure illustrates the confusion matrix obtained using the test dataset of the NSL-KDD. The Eqs. 4–7 below shows the equations of the accuracy, precision, recall and F1-score.

$$\text{Accuracy} = \frac{\text{TP+TN}}{\text{total samples}}, \tag{4}$$

$$\text{Precision } (P) = \frac{\text{TP}}{\text{TP+FP}}, \tag{5}$$

$$\text{Recall } (R) = \frac{\text{TP}}{\text{TP+FN}}, \tag{6}$$

$$\text{F1 - score} = 2 \times \frac{P \times R}{P + R}. \tag{7}$$

Precisions determines the number of positive predictions that are truly positive. Moreover, recall calculates the proportion of positive predictions created employing the positive instances in the dataset. The F1-score is derived by computing the weighted average between precision and recall, and it may be used to seek a balance between precision and recall [24]. The performance of the model is exceptional when the F1-score is greater. Tables 7 and 8 shows the training and test results for accuracy, precision, recall, and f1-score obtained using the NSL-KDD test dataset.

According to the accuracy, loss, precision, and recall graphs in Figs. 3, 4, 5 and 6, the trained DNN model does not overfit or underfit since both curves in each graph have shown almost similar values without any considerable

Table 7 Training set performance results

	Training set results
Accuracy	0.9948
Precision	0.9952
Recall	0.9934
F1-score	0.9942

Table 8 Testing set performance results

	Test set results
Accuracy	0.8187
Precision	0.9645
Recall	0.7071
F1-score	0.8159

Table 9 Performance comparison using test set accuracy, precision, recall and f1-score

Algorithm	Accuracy	Precision	Recall	F1-score
KNN	0.7908	0.9584	0.6136	0.7481
SVM	0.7397	0.9643	0.5568	0.7059
OCSVM	0.7959	0.9600	0.5429	0.6935
K-Means	0.7328	0.9576	0.5369	0.6880
Proposed DNN	0.8187	0.9645	0.7071	0.8159

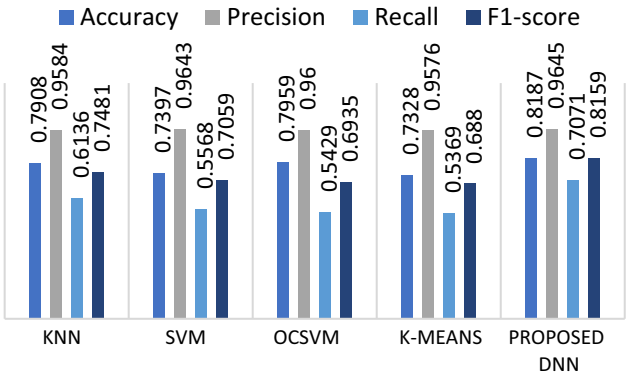


Fig. 8 Performance comparison graph of ML algorithms

differences. In addition, the normalized confusion matrix obtained using the test dataset has shown satisfactory results by achieving higher values for TPs and TNs. Finally, the accuracy, precision, recall, and f1-score calculated using the Eqs. 4–7 have revealed that the trained DNN performs optimally for unseen data by getting higher performance results as shown in Table 8. Consequently, the trained DNN is optimal for a RT-IDS.

Performance Comparison

Table 9 shows a comparison of the proposed DNN model performance with different ML classifiers for binary classification, which were trained using 28 features of the NSL-KDD dataset.

The below graph illustrates the performance of different ML classifiers for easy understanding.

According to Table 9 and Fig. 8, the proposed DNN has shown promising results by outperforming all the other algorithms in terms of accuracy, precision, recall, and f1-score. Other algorithms have shown poor recall values, which leads to an increment in the number of false alarms. Since the f1-score is the weighted average between precision and recall, it was considered the dominant performance metric during evaluation. The f1-score of the DNN is significantly higher compared to others. Consequently, DNN is the optimum algorithm for the RT-IDS.

Furthermore, the proposed RT-IDS has several advantages. By implementing the RT-IDS near the gateway router, network protection for the entire organization can be gained, and by deploying it on a single host, personal network protection can be obtained. Moreover, since the trained DNN, along with the ML pipeline installed in the backend, is containerized using Docker technology, the deployment of the proposed is easier.

Future Works

Several constraints were encountered during the implementation of the RT-IDS. One of the most major constraints has been the unavailability of a rich dataset that contains modern intrusion types and portrays current network traffic patterns. Furthermore, a lack of information about the methods employed for deriving features during the development of the NSL-KDD dataset has restricted the number of features that can be extracted from the network traffic. In addition, there are two disadvantages in the proposed system. Firstly, training the DNN using the NSL-KDD dataset, which contains outdated intrusion types and network traffic patterns, has hampered the deployment of RT-IDS as a contemporary real-world application. Moreover, testing result for the recall is lower compared to precision since the number of normal class records is greater than intrusions. As a result, the proposed system exhibits some bias towards normal data and has a modest tendency to generate false alarms. Consequently, the future scope of the project is aimed at developing a dataset that represents current network traffic patterns together with employing the anomaly detection technique to identify intrusions and integrating it with an automated system to block intrusions.

Conclusion

This research presents a descriptive technical information about RT-IDS based on DNN ML algorithm. It can capture real-time network traffic and identify destructive intrusions and it is hosted in a web server to provide accessibility for personal and corporate sector networks to employ it to their networks via a RESTful API. Since the real-time feature extraction module is containerized, it is effortless to integrate into any system. The proposed system's usability and efficacy have been boosted by its ease of implementation and remote accessibility. The proposed system is extremely advantageous for instantly detecting intrusions by analysing inbound and outbound network traffic. This system outputs descriptive information regarding intrusion data packets,

easing network administrators' decision-making for appropriate measures. Furthermore, because sufficient API documentation is available, even users with limited programming skills can utilize the proposed system. The construction of the DNN, which is trained using the NSL-KDD dataset is systematically discussed together with the technical implementation and the simulation results on both training and testing aspects. Moreover, the techniques and procedures employed for real-time feature extraction from the inbound and outbound network traffic are clearly mentioned. In addition, how the ML prediction pipeline is hosted in a web server is descriptively discussed in this paper. The observed results of the DNN training and testing have showed exceptional training results and satisfactory results in the testing stage with a precision of 96%. Finally, our research work has contributed by presenting a fully functional RT-IDS which can be practically implemented as an extra layer of the network protection.

Declarations

Conflict of interest The authors declare that they have no conflict of interest. Some of the datasets used and the code generated during the current study are available from the corresponding author on reasonable request.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Tavallaee M, Bagheri E, Lu W, Ghorbani AA. A detailed analysis of the KDD CUP 99 data set. In: Proceedings of the 2009 IEEE symposium on computational intelligence in security and defense applications (CISDA), Ottawa, ON, Canada, July 2009. https://doi.org/10.1109/CISDA.2009.5356528.
2. Nour M, Jill S. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In: Proceedings of the military communications and information systems conference, Australia, November 2015. https://doi.org/10.1109/MilCIS.2015.7348942.
3. Thirimanne S, Jayawardana L, Liyanaarachchi P, Yasakethu L. Comparativqe algorithm analysis for machine learning based intrusion detection system. In: Proceedings of the 10th international conference on information and automation for sustainability (ICIAfS), Negombo, Sri Lanka, August 2021. https://doi.org/10.1109/ICIAfS52090.2021.9605814.
4. Rajesh T, Deepa P. A survey of intrusion detection models based on NSL-KDD data set. In: Proceedings of the 5th HCT information technology trends (ITT), Dubai, United Arab Emirates, November 2018. https://doi.org/10.1109/CTIT.2018.8649498.
5. Hayoung O, Kijoon C. Real-time intrusion detection system based on self-organized maps and feature correlations. In: Proceedings of the 3rd international conference on convergence and hybrid information technology, Korea (South), November 2008. https://doi.org/10.1109/ICCIT.2008.362.
6. Komviriyavut T, Sangkatsanee P, Wattanapongsa-korn N, Charnsripinyo C. Network intrusion detection and classification with Decision Tree and rule based approaches. In: Proceedings of the 9th international symposium on communications and information technology, Korea, September 2009. https://doi.org/10.1109/ISCIT.2009.5341005.
7. Sangkatsanee P, Wattanapongsakorn N, Charnsripinyo C. Practical real-time intrusion detection using machine learning approaches. Comput Commun. 2011;34(18):2227–35. https://doi.org/10.1016/j.comcom.2011.07.001.
8. Hui W, Zijian C, Bo H. A network intrusion detection system based on convolutional neural network. J Intell Fuzzy Syst. 2020;38:7623–37. https://doi.org/10.3233/JIFS-179833.
9. Zhang C, Chen Y, Meng Y, Ruan F, Chen R, Li Y, Yang Y. A novel framework design of network intrusion detection based on machine learning techniques. Secur Commun Netw. 2021. https://doi.org/10.1155/2021/6610675.
10. Qureshi AS, Khan A, Shamim N, Durad MH. Intrusion detection using deep sparse auto-encoder and self-taught learning. Neural Comput Appl. 2020. https://doi.org/10.1007/s00521-019-04152-6.
11. Kabir MR, Onik AR, Samad T. A network intrusion detection framework based on bayesian network using wrapper approach. Int J Comput Appl. 2017. https://doi.org/10.5120/ijca2017913992.
12. Jukic S, Saracevic M, Subasi A, Kevric J. Comparison of ensemble machine learning methods for automated classification of focal and non-focal epileptic EEG signals. Mathematics. 2020;8(9):1481. https://doi.org/10.3390/math8091481.
13. Tang H, Cao Z. Machine learning-based intrusion detection algorithms. J Comput Inf Syst. 2009;5:1825–31.
14. Poonam S, Akansha S. Era of deep neural networks: a review. In: 8th international conference on computing, communication and networking technologies, July 2017. https://doi.org/10.1109/ICCCNT.2017.8203938.
15. Algorithmia. What an ML pipeline is and why it's important. 2020. https://algorithmia.com/blog/ml-pipeline. Accessed 7 June 2021.
16. Ansari S, Rajeev S, Chandrashekar H. Packet sniffing: a brief introduction. IEEE Potent. 2003;21(5):17–9. https://doi.org/10.1109/MP.2002.1166620.
17. Scikit-Learn. Sklearn Preprocessing StandardScaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html. Accessed 19 July 2021.
18. Brownlee J. A gentle introduction to the rectified linear unit (ReLU), machine learning mastery. 2019. https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks. Accessed 3 June 2021.
19. Katanforoosh K, Kunin D. Initializing neural networks. deeplearning.ai. 2019. https://www.deeplearning.ai/ai-notes/initialization/#II. Accessed 5 June 2021.
20. Keras. Layer activation functions. 2021. https://keras.io/api/layers/activations/#relu-function. Accessed 19 July 2021.
21. Keras. Layer weight initializers. 2021. https://keras.io/api/layers/initializers/. Accessed 19 July 2021.
22. Hunt-Walker N. An introduction to the Flask Python web app framework. Opensource. 2018. https://opensource.com/article/18/4/flask. Accessed 18 July 2021.

23. Gunicorn. Gunicorn—WSGI server. https://docs.gunicorn.org/en/stable/. Accessed 27 July 2021.
24. Yiqun Z, Pengcheng M, Qian G. Multiple classification models based student's phobia prediction study. In: Proceedings of the 3rd IEEE international conference on robotic computing (IRC), February 2019. https://doi.org/10.1109/IRC.2019.00109.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.