CHAPTER 1

INTRODUCTION

1.1 Overview

The traffic through the network is heterogeneous and consists of flows from multiple applications and utilities. These applications are unique and will have their own requirements with respect to network parameters (e.g.: delay, jitter, etc). The quality and usability of these applications will be severely affected if these requirements are not satisfied. While meeting these requirements in a Local Area Network (LAN) with its huge bandwidth might be easy, but when considering WANs it usually is a challenge because of its bandwidth constraints.

Thus, traffic management on the WANs must exist in order to properly prioritize different applications across the limited WAN bandwidth and ensure that these requirements are met. In order to implement appropriate security policies, the network managers must have a detailed knowledge about applications and protocols. The user application may allow large delays or jitter, but the users might be very sensitive to long wait times. Managing network traffic requires a judicious balance of all these priorities.

Classification of traffic helps identify different applications and protocols that exist in a network. Different methods such as monitoring, discovery, control, and optimization can be performed on the identified traffic in order to improve the network performance. Typically, once the packets are classified (identified) as belonging to a particular application or protocol, they are marked or flagged. These markings or flags help the router determine appropriate service policies to be applied for those flows. All generic classification techniques based on Destination IP address, Source IP address, or IP protocol, etc. are limited in their ability as the inspection is limited to the IP header only. Similarly, classifying based on Layer 4 ports only is also limited [1]. The problem with this approach is that not all current applications use standard ports. Some applications even obfuscate themselves by using well the defined ports of other applications. Hence the Layer 4 port mechanism of application identification is not always reliable.

1.2 Self Learning Networks

Self Learning Networks (SLN) architecture is a solution that combines powerful analytics, using a wide set of machine learning technologies hosted on edge devices, along with advanced networking, to allow the network to become intelligent, adaptive, proactive, and predictive. The Cisco SLN architecture relies on the use of distributed, lightweight, yet complex analytic engines, referred to as Distributed Learning Agents (DLAs), implemented across the network. The DLA reports findings to the SLN Orchestrator, which provides orchestration and interaction among the distributed DLAs, as well as supports additional features to be developed in the future.

1.3 Deep Packet Inspection

Deep packet inspection (DPI) also called complete packet inspection is a form of computer network packet filtering that examines the data part (also the header) of a packet it passes an inspection point, searching for protocol noncompliance, viruses, spam, intrusions, or defined criteria to decide whether the packet may pass or if it needs to be routed to a different destination, or, for the purpose of collecting statistical information that functions at the Application layer of the OSI (Open Systems Interconnection model). Deep Packet Inspection (and filtering) enables advanced network management, user service, and security functions as well as internet data mining, eavesdropping, and internet censorship [5]. Although DPI has been used for Internet management for many years, some advocates of net neutrality fear that the technique may be used anticompetitive or to reduce the openness of the Internet.

1.4 Deep Learning

Deep learning (also known as deep structured learning or hierarchical learning) is part of a broader family of machine learning methods based on learning data representations, as opposed to task-specific algorithms. Deep learning architectures such as deep neural networks, deep belief networks and recurrent neural networks have been applied to fields including computer vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation. This deep learning helps us to solve the problem in layer-wise step by step along with enhancing the results compare to normal machine learning approaches.

1.5 Objective

Anomaly-based intrusion detection techniques are valuable methodology to detect both known as well as unknown/new attacks, so they can cope with the diversity of the attacks and the constantly changing nature of network attacks. There are many problems need to be considered in anomaly-based network intrusion detection system (NIDS), such as ability to adapt to dynamic network environments, unavailability of labelled data, false positive rate. We use Deep learning techniques to implement classification problem in network traffic dataset. These techniques show the sensitive power of generative models with good classification, capabilities to deduce part of its knowledge from incomplete data and the adaptability.

1.6 Motivation

In anomaly-based network intrusion detection, the system is trained with "normal" network traffic to generate Model. When the Model for the system is available, it is subsequently used to classify new events or objects or traffic as anomalous or not. To get an effective model, an expected characteristic is its ability to adapt to generalize its behaviour and cope with dynamic network environments. That means a self-learning system is needed. Another major issue in anomaly-based network intrusion detection is availability of labelled data for training and validation of models. Labels for normal behaviour are usually available, while labels for intrusions are not. Therefore, semi supervised and unsupervised anomaly detection techniques are preferred in this case. Therefore, motivated by problems above, we propose an anomaly-based NIDS using deep learning. With deep learning, we expect to tackle issues in anomaly-based network intrusion detection, such as high intrusion detection rate, ability to adapt to dynamic network environments, unavailability of labelled data. Deep learning a class of machine learning techniques, where many layers of information processing stages in hierarchical architectures are exploited for unsupervised feature learning and for pattern analysis or classification. Deep learning shows the power of generative models with high accuracy of classification.

CHAPTER 2
LITERATURE SURVEY

2.1 Anomaly-based Network Intrusion Detection System using Deep learning

Anomaly-based NIDS is an effective way to detect new attacks. However, its limitation is false positive when a normal behaviour is not in the normal network data set. Therefore, how to model properly all the normal network data, whether they share a common range of values for a set of Features, as well as correlation and sequencing between them. Implement and evaluate of the application of Deep Learning to anomaly-based NIDS. The results show that this system detects intrusion and classify attacks in four groups with high accuracy using two deep learning techniques include Stacked Auto encoder and Stacked RBM, show that the use of Stacked Auto encoder technique is better than that in RBM. The problem of the training time-consuming in SAE is much more than that in RBM due to much computation in SAE [1]. drawback in this approach was since streaming data involves noise hence normal auto encoder fails to handle noisy data this drawback is overcome by using stacked-denoising auto encoder in our approach.

2.2 Deep Packet: A Novel Approach for Encrypted Traffic Classification Using Deep Learning

Network traffic classification has become significantly important with rapid growth of current Internet network and online applications. There have been numerous studies on this topic have led to many different approaches. Most of these approaches use predefined features extracted by an expert in order to classify network traffic. In contrast, propose a deep learning based approach which integrates both feature extraction and classification phases into one system. This proposed scheme, called Deep Packet, It can handle both traffic characterization, in which the network traffic is categorized into major classes (e.g., FTP and P2P), and application identification in which identification of end-user applications (e.g., BitTorrent and Skype) is desired. Contrary to the most of current methods, Deep Packet can identify encrypted traffic and also distinguishes between VPN and non-VPN network traffic.

Deep Packet, a framework that automatically extracts features from network traffic using deep learning algorithms in order to classify traffic. It is the first traffic classification system using deep learning algorithms, namely, SAE and one dimensional CNN that can handle both application identification and traffic characterization tasks [2]. Their results showed that Deep Packet outperforms all of the similar works on the ISCX VPN-non VPN traffic dataset both in traffic classification and traffic characterization to the data they are able to characterize and classify the data up to 95 % accuracy this can be overcome in our model.

2.3 The Applications of Deep Learning on Traffic Identification

Generally speaking, most systems of network traffic identification are based on features. The features may be port numbers, static signatures, statistic characteristics, and so on. The difficulty of the traffic identification is to find the features in the flow data. The process is very time-consuming. Also, these approaches are invalid to unknown protocol. To solve these problems, they proposed a method that is based on neural network and deep learning – a hotspot of research in machine learning their results show that this approach works very well on the applications of feature learning, protocol identification and anomalous protocol detection. Auto encoder can be also used for feature selection. The process is very similar to ANN. However, there are two differences between them. First, ANN here is for finding discriminative features across many categories, while Auto encoder is for finding features with something in common in just one category [3]. This leads to opposite direction that we should pay attention to. If we use the former, the bigger the value in the better feature can be chosen. In Auto encoder, the smaller, the better, because we want to hold some stable information in a category. Second, In ANN model, labels are necessary while Auto encoder is not. That’s determined by their algorithmic essence.

2.4 Comparison of Deep Learning Method to Traditional Methods Using for Network Intrusion Detection

Deep learning has gained prominence due to the potential it portends for machine learning. For this reason, deep learning techniques have been applied in many fields, such as recognizing some kinds of patterns or classification. Intrusion detection analyses got data from monitoring security events to get situation assessment of network. Lots of traditional machine learning method has been put forward to intrusion detection, but it is necessary to improve the detection performance and accuracy. The use of deep learning has in recent times gained prominence due to its effectiveness in evaluating network security [4].

Notably, traditional methods of network security are increasingly failing to the function due to increased processing of data. Nonetheless, deep learning has revolutionized the evaluation of challenges in network security similarly new methods of deep learning are gaining traction due demand for faster and efficient data assessment. Deep belief and deep coding techniques have enabled the analysis of large data sets and deeper system analysis respectively.

2.5 IEEE 802.11 Network Anomaly Detection and Attack Classification: A Deep Learning Approach

Despite the significant advancement in wireless technologies over the years, IEEE 802.11 still emerges as the de-facto standard to achieve the required short to medium range wireless device connectivity in anywhere from offices to homes. With it being ranked the highest among all deployed wireless technologies in terms of market adoption, vulnerability exploitation and attacks targeting it have also been commonly observed. IEEE 802.11 security has thus become a key concern over the years [5]. They analysed the threats and attacks targeting the IEEE 802.11 network and also identified the challenges of achieving accurate threat and attack classification, especially in situations where the attacks then proposed a solution based on anomaly detection and classification using a deep learning approach. The deep learning approach self-learns the features necessary to detect network anomalies and is able to perform attack classification accurately.

2.6 Overall Summary

Overall existing work summarizes that people are focused on only classification on different dataset not take care of noise in the real time traffic also achieved accuracy of up to 97% this two drawbacks are overcoming in our deep learning model by using stacked denoising auto encoder and some advanced optimizing technique to enhance the classification accuracy, finally testing our deep learning model can generalizes the anomalies in real time network traffic as well.

CHAPTER 3
SYSTEM REQUIREMENTS

3.1 Hardware Requirements

Hardware	Minimum Requirement
Processor	Pentium 4
RAM	4 GB
Hard-disk Space	5 GB
No of Cores	Dual core CPU

Table 3.1 Hardware Requirements

3.2 Software Requirements

Software	Minimum Requirement
Operating system	Windows 7 and above
MATLAB	Version above 2007
Weka	Version 3.8
Anaconda	Version 3.6.5

Table 3.2 Software Requirements

Table 3.1 shows the normal system specification along with this cloud based GPU using Floyd hub online services used to run this model Table 3.2 shows the software used in that matlab and weka tool used for dataset analysis and designed algorithm is implemented using keras and theano libraries solution is implemented in Jupyter notebook using python.

CHAPTER 4

SYSTEM DESIGN

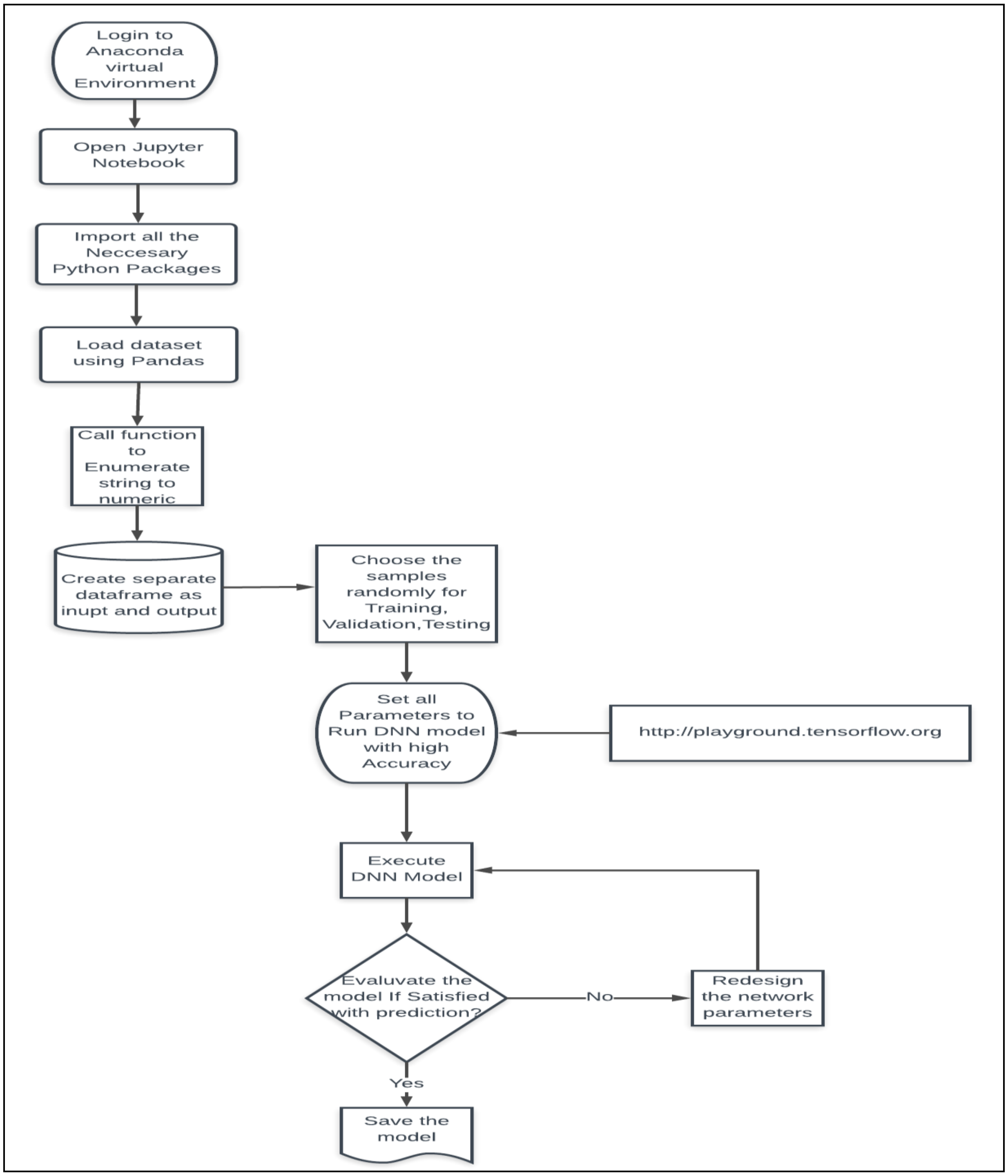

Login to Anaconda virtual Environment
Open Jupyter Notebook
Import all the Neccesary Python Packages
Load dataset using Pandas
Call function to Enumerate string to numeric
Create separate dataframe as inupt and output
Choose the samples randomly for Training, Validation,Testing
Set all Parameters to Run DNN model with high Accuracy
http://playground.tensorflow.org
Execute DNN Model
Evaluvate the model If Satisfied with prediction?
No
Redesign the network parameters
Yes
Save the model

Fig 4.1 DNN System Design Process

The problem of intrusion is gradually becoming nightmare for several organizations. To protect the valuable data of their clients, organizations implement security systems to detect and prevent security breaches. But since the intruders are using sophisticated techniques to penetrate the systems, even the highly reputed secured systems have become vulnerable now [5]. To deal with the current scenario, advanced levels of researches are required to be carried out to invent more sophisticated Intrusion Detection System (IDS). Above fig 4.1 shows the flow chart of DNN system design process. Created separate virtual environment in anaconda using following command.

Conda create –n env_name python=x.x anaconda

Env_name=sln and x.x is python version 3.6.4 then activate your virtual environment using

Conda activate sln

In anaconda prompt go to the specific folder where you installed all python packages then open jupyter notebook the proceed with DNN model.

The deep learning framework can be considered as a special kind of NNs with many (hidden) layers. Nowadays, with rapid growth of computational power and the availability of graphical processing units (GPUs), training deep NNs have become more plausible. Therefore, the researchers from different scientific fields using deep learning framework in their respective area of research.

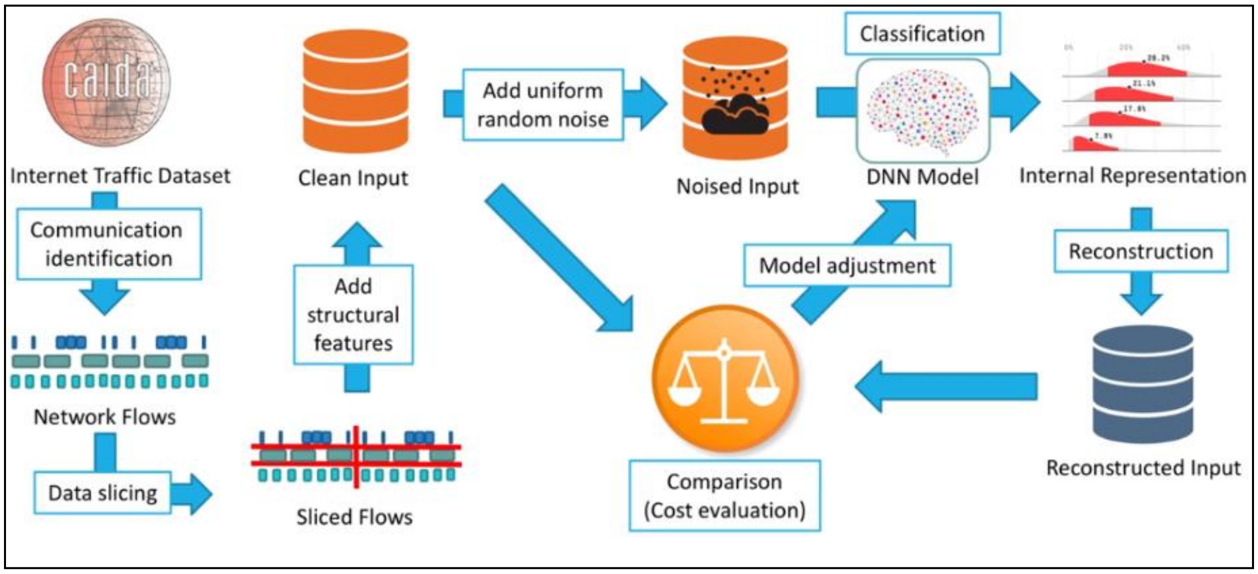


Figure4.2 Model of deep learning process

The principle of deep learning is to compute hierarchical features or representations of the observational data, where the higher-level features or factors are defamed from lower level ones. Fig 4.2 shows the model of deep learning process in Deep learning techniques aim to learn a good feature representation from a large amount of labelled data, so the model can be pre-trained in a completely supervised fashion [7]. The very limited labelled data can then be used to only slightly fame-tune the model for a specific task in the supervised classification. The strategy of layer-wise unsupervised training followed by supervised fame-tuning allows efficient training of deep networks and gives promising results for many challenging learning problems, substantially improving upon the current [8]. Initializing weights in an area where is near a good local minimum make increase to internal distributed representations that are high level abstractions of the input, thus it brings a better generalization.

CHAPTER 5

IMPLEMENTATION

5.1 Data Analysis

The UNSW-NB15 data set description

The raw network packets of the UNSW-NB 15 data set was created by the IXIA Perfect Storm tool in the Cyber Range Lab of the Australian Centre for Cyber Security (ACCS) for generating a hybrid of real modern normal activities and synthetic contemporary attack behaviors.

Tcpdump tool is utilized to capture 100 GB of the raw traffic (e.g., Pcap files). This data set has nine types of attacks, namely, Fuzzers, Analysis, Backdoors, DoS, Exploits, Reconnaissance, Shellcode and Worms. The Argus, Bro-IDS tools are used and twelve algorithms are developed to generate totally 45 features with the class label.

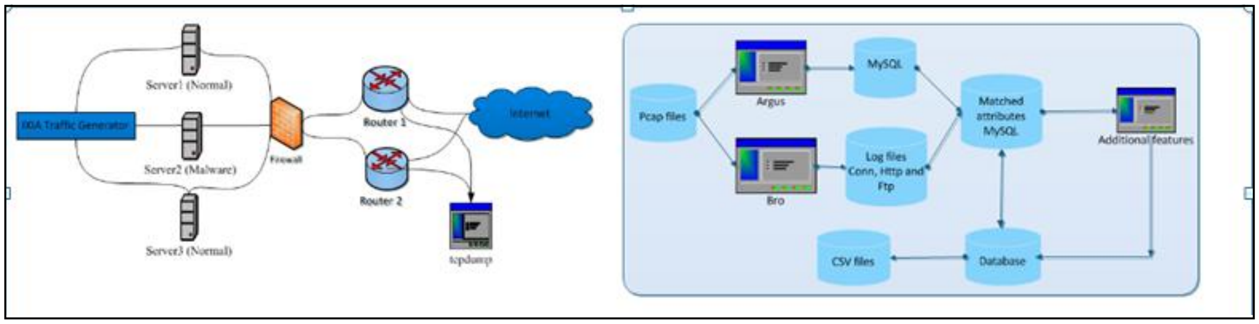


Figure 5.1 Testbed configured feature creation

The number of records in the training set is 175,341 records and the testing set is 82,332 records from the different types, attack and normal. Figure 5.1 shows the testbed configuration data set and the method of the feature creation of the UNSW-NB15 dataset showed in table 5.1, respectively.

For our analysis of dataset considered only 1000 samples of the dataset and It contains output class labels of size 243 normal,331 exploits,150 Fuzzers,124 Dos,114 Reconiassance,15 shellcode,14 Analysis,6 Backdoor,2 worms.

No.	Name	Type	Description
1	Proto	Nominal	Transaction protocol
2	State	Nominal	Indicates to the state and its dependent protocol, e.g. ACC, CLO, CON, ECO, ECR, FIN, INT, MAS, PAR, REQ, RST, TST, TXD, URH, URN, and (-) (if not used state)
3	Dur	Float	Record total duration
4	Sbytes	Integer	Source to destination transaction bytes
5	Dbytes	Integer	Destination to source transaction bytes
6	Sttl	Integer	Source to destination time to live value
7	Dttl	Integer	Destination to source time to live value
8	Sloss	Integer	Source packets retransmitted or dropped
9	Dloss	Integer	Destination packets retransmitted or dropped

10	Service	Nominal	http, ftp, smtp, ssh, dns, ftp-data ,irc and (-) if not much used service
11	Sload	Float	Source bits per second
12	Dload	Float	Destination bits per second
13	Spkts	Integer	Source to destination packet count
14	Dpkts	Integer	Destination to source packet count
15	Swin	Integer	Source TCP window advertisement value
16	Dwin	Integer	Destination TCP window advertisement value
17	Stcpb	Integer	Source TCP base sequence number
18	Dtcpb	Integer	Destination TCP base sequence number
19	Smeansz	Integer	Mean of the ?ow packet size transmitted by the src
20	Dmeansz	Integer	Mean of the ?ow packet size transmitted by the dst

21	trans_depth	Integer	Represents the pipelined depth into the connection of http request/response transaction
22	res_bdy_len	Integer	Actual uncompressed content size of the data transferred from the server's http service.
23	Sjit	Float	Source jitter (mSec)
24	Djit	Float	Destination jitter (mSec)
25	Stime	Timestamp	record start time
26	Ltime	Timestamp	record last time
27	Sintpkt	Float	Source interpacket arrival time (mSec)
28	Dintpkt	Float	Destination interpacket arrival time (mSec)
29	Tcprtt	Float	TCP connection setup round-trip time, the sum of 'synack' and 'ackdat'.
30	Synack	Float	TCP connection setup time, the time between the SYN and

			the SYN_ACK packets.
31	Ackdat	Float	TCP connection setup time, the time between the SYN_ACK and the ACK packets.
32	is_sm_ips_ports	Binary	If source (1) and destination (3)IP addresses equal and port numbers (2)(4) equal then, this variable takes value 1 else 0
33	ct_state_ttl	Integer	No. for each state (6) according to specific range of values for source/destination time to live (10) (11).
34	ct_flw_http_mthd	Integer	No. of flows that has methods such as Get and Post in http service.
35	is_ftp_login	Binary	If the ftp session is accessed by user and password then 1 else 0.
36	ct_ftp_cmd	Integer	No of flows that has a

			command in ftp session.
37	ct_srv_src	Integer	No. of connections that contain the same service (14) and source address (1) in 100 connections according to the last time (26).
38	ct_srv_dst	Integer	No. of connections that contain the same service (14) and destination address (3) in 100 connections according to the last time (26).
39	ct_dst_ltm	Integer	No. of connections of the same destination address (3) in 100 connections according to the last time (26).
40	ct_src_ ltm	integer	No. of connections of the same source address (1) in 100 connections according to the last time (26).
41	ct_src_dport_ltm	Integer	No of connections of the same source address (1) and the

			destination port (4) in 100 connections according to the last time (26).
42	ct_dst_sport_ltm	Integer	No of connections of the same destination address (3) and the source port (2) in 100 connections according to the last time (26).
43	ct_dst_src_ltm	Integer	No of connections of the same source (1) and the destination (3) address in in 100 connections according to the last time (26).
44	attack_cat	nominal	The name of each attack category. In this data set , nine categories e.g. Fuzzers, Analysis, Backdoors, DoS Exploits, Generic, Reconnaissance, Shellcode and Worms
45	Label	Binary	0 for normal and 1 for attack records

Table 5.1 Dataset Information

5.2 Data Representation

Changing the values of nominal type to numeric according to it

0) **Normal** –no attack in the packet

1) **Reconnaissance** - is a type of computer attack in which an intruder engages with the targeted system to gather information about vulnerabilities. The word *reconnaissance* is borrowed from its military use, where it refers to a mission into enemy territory to obtain information. In a computer security context, reconnaissance is usually a preliminary step toward a further attack seeking to exploit the target system. The attacker often uses port scanning, for example, to discover any vulnerable ports

2) **Backdoors**- A backdoor is a means to access a computer system or encrypted data that bypasses the system's customary security mechanisms. A developer may create a backdoor so that an application or operating system can be accessed for troubleshooting or other purposes. However, attackers often use backdoors that they detect or install themselves as part of an exploit. In some cases, a worm or virus is designed to take advantage of a backdoor created by an earlier attack.

3) **Dos** - An exploit (from the English verb *to exploit*, meaning "to use something to one's own advantage") is a piece of software, a chunk of data, or a sequence of commands that takes advantage of a bug or vulnerability to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized). Such behavior frequently includes things like gaining control of a computer system, allowing privilege escalation, or a denial-of-service (DoS or related DDoS) attack.

4) **Exploits** - An exploit is a piece of software, a chunk of data, or a sequence of commands that takes advantage of a bug or vulnerability to exploit the system.

5) **Analysis** –type of attack on data segment of the packet.

6) **Fuzzers** - Fuzz testing (fuzzing) is a quality assurance technique used to discover coding errors and security loopholes in software, operating systems or networks. It involves

inputting massive amounts of random data, called fuzz, to the test subject in an attempt to make it crash.

7) **Worms** - A computer worm is self-replicating malware that duplicates itself to spread to uninfected computers. Worms often use parts of an operating system that are automatic and invisible to the user. It is common for worms to be noticed only when their uncontrolled replication consumes system resources, slowing or halting other tasks.

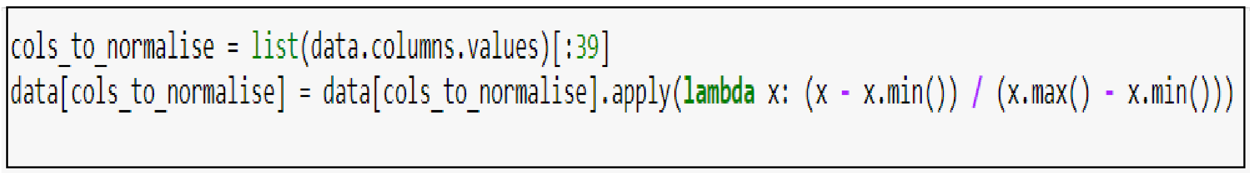
8) **ShellCode-** In hacking, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called shellcode. Because the function of a payload is not limited to merely spawning a shell, some have suggested that the name shellcode is insufficient. However, attempts at replacing the term have not gained wide acceptance. Shellcode is commonly written in machine code.

5.3 Data Normalization

Data Normalization is a method used to standardize the range of independent variables or features of data. In data processing, it is also known as feature scaling and is generally performed during the data preprocessing step. Since the range of values of raw data varies widely, in some machine learning algorithms, objective functions will not work properly without normalization.

Min-max normalization is a normalization strategy which linearly transforms x to y.

Y = (X-min)/(max-min) where min and max are the minimum and maximum values in that particular vector, where X is the set of observed values in that vector. It can be easily seen that when x=min, then y=0, and When x=max, then y=1. This means, the minimum value in X is mapped to 0 and the maximum value in X is mapped to 1. So, the entire ranges of values of X from min to max are mapped to the range 0 to 1.



```
cols_to_normalise = list(data.columns.values)[:39]
data[cols_to_normalise] = data[cols_to_normalise].apply(lambda x: (x - x.min()) / (x.max() - x.min()))
```

5.4 Feature Selection

Feature selection is the process of removing features from the original data set that are irrelevant with respect to the task that is to be performed. So not only the execution time of the classifier that processes the data reduces but also accuracy increases because irrelevant or redundant features can include noisy data affecting the classification accuracy negatively. Researchers knew artificial neural network as a universal function approximations and from the very beginning it was known that multiple number nonlinear transformations smoothen out non-convexity of any mapping or learning. But the problem was how to train very deep network, especially when the dimension of the input space is very high. So, concept of feature space has come, where manually researchers reduced the number of input dimension and smoothen out some non-convexity through feature space.

5.5 Stacked Denoising Auto-Encoder

An autoencoder takes an input $\mathbf{x} \in [0,1]^d$ and first maps it (with an *encoder)* to a hidden representation $\mathbf{y} \in [0,1]^{d'}$ through a deterministic mapping, e.g.:

$$\mathbf{y} = s(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Where s is a non-linearity such as the sigmoid. The latent representation $\mathbf{y}$, or code is then mapped back (with a *decoder)* into a reconstruction $\mathbf{z}$ of the same shape as $\mathbf{x}$. The mapping happens through a similar transformation, e.g.:

$$\mathbf{z} = s(\mathbf{W}'\mathbf{y} + \mathbf{b}')$$

(Here, the prime symbol does not indicate matrix transposition.) $\mathbf{z}$ should be seen as a prediction of $\mathbf{x}$, given the code $\mathbf{y}$. Optionally, the weight matrix $\mathbf{W}'$ of the reverse mapping may be constrained to be the transpose of the forward mapping: $\mathbf{W}' = \mathbf{W}^T$. This is referred to as *tied weights*. The parameters of this model (namely $\mathbf{W}$, $\mathbf{b}$, $\mathbf{b}'$ and, if one doesn't use tied weights, also $\mathbf{W}'$) are optimized such that the average reconstruction error is minimized.

The reconstruction error can be measured in many ways, depending on the appropriate distributional assumptions on the input given the code. The traditional *squared error* $L(\mathbf{x}\mathbf{z}) = ||\mathbf{x} - \mathbf{z}||^2$, can be used. If the input is interpreted as either bit vectors or vectors of bit probabilities, *cross-entropy* of the reconstruction can be used:

$$\boxed{L_H(\mathbf{x}, \mathbf{z}) = -\sum_{k=1}^{d} [\mathbf{x}_k \log \mathbf{z}_k + (1 - \mathbf{x}_k) \log(1 - \mathbf{z}_k)]}$$

The hope is that the code $\mathbf{y}$ is a *distributed* representation that captures the coordinates along the main factors of variation in the data. This is similar to the way the projection on principal components would capture the main factors of variation in the data. Indeed, if there is one linear hidden layer (the *code)* and the mean squared error criterion is used to train the network, then the k hidden units learn to project the input in the span of the first k principal components of the data. If the hidden layer is non-linear, the auto-encoder behaves differently from PCA, with the ability to capture multi-modal aspects of the input distribution. The departure from PCA becomes even more important when we consider *stacking multiple encoders* (and their corresponding decoders) when building a deep auto-encoder[6] .

Because $\mathbf{y}$ is viewed as a lossy compression of $\mathbf{x}$, it cannot be a good (small-loss) compression for all $\mathbf{x}$. Optimization makes it a good compression for training examples, and hopefully for other inputs as well, but not for arbitrary inputs. That is the sense in which an auto-encoder generalizes: it gives low reconstruction error on test examples from the same distribution as the training examples, but generally high reconstruction error on samples randomly chosen from the input space.

The idea behind denoising autoencoders is simple. In order to force the hidden layer to discover more robust features and prevent it from simply learning the identity, we train the autoencoder to reconstruct the input from a corrupted version of it.

The denoising auto-encoder is a stochastic version of the auto-encoder. Intuitively, a denoising auto-encoder does two things: try to encode the input (preserve the information about the input), and try to undo the effect of a corruption process stochastically applied to the input of the auto-encoder. The latter can only be done by capturing the statistical dependencies between the inputs. The denoising auto-encoder can be understood from different perspectives the manifold learning perspective, stochastic operator perspective, bottom-up – information theoretic perspective, top-down – generative model perspective.

Denoising autoencoders can be stacked to form a deep network by feeding the latent representation (output code) of the denoising autoencoder found on the layer below as input to the current layer. The unsupervised pre-training of such architecture is done one layer at a time. Each layer is trained as a denoising autoencoder by minimizing the error in reconstructing its input (which is the output code of the previous layer). Once the first k layers are trained, we can train the $k+1$-th layer because we can now compute the code or latent representation from the layer below.

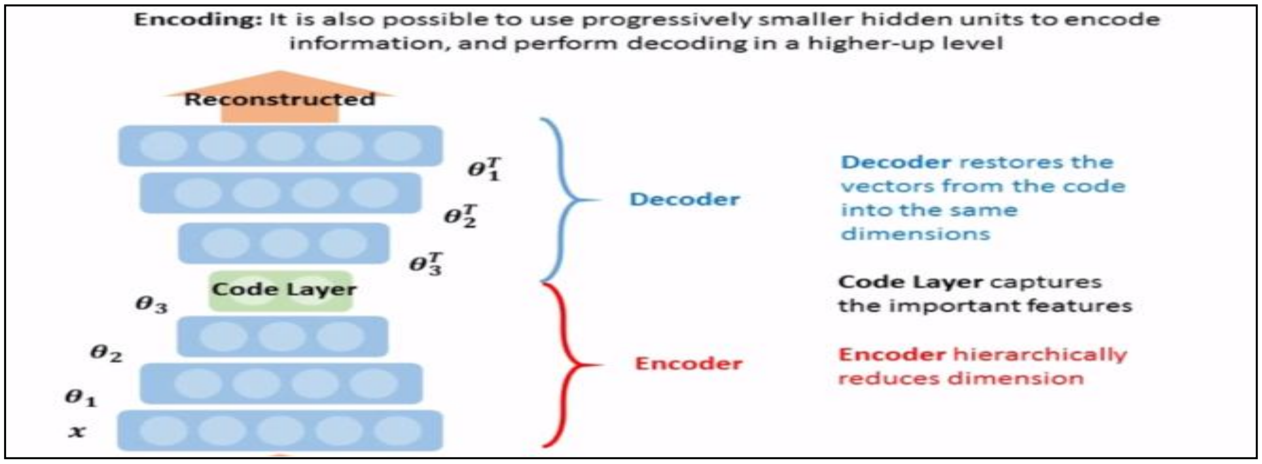

Encoding: It is also possible to use progressively smaller hidden units to encode information, and perform decoding in a higher-up level
Reconstructed
θ_1^T
θ_2^T
θ_3^T
Code Layer
θ_3
θ_2
θ_1
x
Decoder
Encoder
Decoder restores the vectors from the code into the same dimensions
Code Layer captures the important features
Encoder hierarchically reduces dimension

Figure 5.2 Stacked auto encoder

Once all layers are pre-trained, the network goes through a second stage of training called fine-tuning. Here we consider supervised fine-tuning where we want to minimize prediction error on a supervised task. For this, we first add a logistic regression layer on top of the network (more precisely on the output code of the output layer). We then train the entire network as we would train a multilayer perceptron. At this point, we only consider the encoding parts of each auto-encoder. This stage is supervised, since now we use the target class during training.

5.6 Deep Neural Network Model

In this model uses a high level API for building deep neural network, all these API are in tflearn package of tensorflow and TFLearn introduces a High-Level API that makes neural network building and training fast and easy. This API is intuitive and fully compatible with Tensorflow. Layers are a core feature of TFLearn. TFLearn brings "layers" that represent an abstract set of operations to make building neural networks more convenient.

5.6.1 Input Layer

Syntax:

```
tflearn.layers.core.input_data (shape=None,placeholder=None,dtype=tf.float32,
data_preprocessing=None, data_augmentation=None, name='InputData')
```

This layer is used for inputting (aka. feeding) data to a network. A TensorFlow placeholder will be used if it is supplied, otherwise a new placeholder will be created with the given shape.

Arguments

- **shape**: list of int. An array or tuple representing input data shape. It is required if no placeholder is provided. First element should be 'None' (representing batch size), if not provided, it will be added automatically.
- **placeholder**: A Placeholder to use for feeding this layer (optional). If not specified, a placeholder will be automatically created. You can retrieve that placeholder through graph key: 'INPUTS', or the 'placeholder' attribute of this function's returned tensor.
- **dtype**: tf.type, Placeholder data type (optional). Default: float32.
- **data_preprocessing**: A DataPreprocessing subclass object to manage real-time data pre-processing when training and predicting (such as zero center data, std normalization...).
- **data_augmentation**: DataAugmentation. A DataAugmentation subclass object to manage real-time data augmentation while training (such as random image crop, random image flip, random sequence reverse...).
- **name**: str. A name for this layer (optional).

In our deep Neural Network we are feeding 42 features for DNN model so that it keeps placeholder of size 42 in the Input Layer. It specified as below

input_layer = tflearn.input_data(shape=[None,42])

5.6.2 Hidden Layer

Syntax:

```
tflearn.layers.core.fully_connected (incoming, n_units, activation='linear', bias=True, weights_init='truncated_normal', bias_init='zeros', regularizer=None, weight_decay=0.001, trainable=True, restore=True, reuse=False, scope=None, name='FullyConnected')
```

It is fully connected layer it accepts the inputs from the input layer

Arguments

- **incoming**: Tensor. Incoming (2+)D Tensor.
- **n_units**: int, number of units for this layer.
- **activation**: str (name) or function (returning a Tensor). Activation applied to this layer (see tflearn.activations). Default: 'linear'.

- **bias**: bool. If True, a bias is used.
- **weights_init**: str (name) or Tensor. Weights initialization. (see tflearn.initializations) Default: 'truncated_normal'.
- **bias_init**: str (name) or Tensor. Bias initialization. (see tflearn.initializations) Default: 'zeros'.
- **regularizer**: str (name) or Tensor. Add a regularizer to this layer weights (see tflearn.regularizers). Default: None.
- **weight_decay**: float. Regularizer decay parameter. Default: 0.001.
- **trainable**: bool. If True, weights will be trainable.
- **restore**: bool. If True, this layer weights will be restored when loading a model.
- **reuse**: bool. If True and 'scope' is provided, this layer variables will be reused (shared).
- **scope**: str. Define this layer scope (optional). A scope can be used to share variables between layers. Note that scope will override name.
- **name**: A name for this layer (optional). Default: 'FullyConnected'.

```
dense1 = tflearn.fully_connected(input_layer, 64, activation='tanh',
                regularizer='L2', weight_decay=0.001)
```

In our classification problem of UNSW network traffic dataset using 64 number of neurons in the hidden layer and tan hyperbolic function as activation function with Regularization techniques used to address over-fitting and feature selection problems L2 regularizer is also called as Ridge Regression. It adds "*squared magnitude*" of coefficient as penalty term to the loss function. This technique works very well to avoid over-fitting issue.

When training neural networks, it is common to use "weight decay," where after each update, the weights are multiplied by a factor slightly less than 1. This prevents the weights from growing too large, and can be seen as gradient descent on a quadratic regularization term. During training regularization term is added to the network's loss to compute the back prop gradient. The weight decay value determines how dominant this regularization term will be in the gradient computation.

5.6.3 Dropout Layer

Syntax:

```
tflearn.layers.core.dropout (incoming, keep_prob, noise_shape=None, name='Dropout')
```

This layer Outputs the input element scaled up by 1 / keep_prob. The scaling is so that the expected sum is unchanged.

Arguments

- incoming : A Tensor. The incoming tensor.
- keep_prob : A float representing the probability that each element is kept.
- noise_shape : A 1-D Tensor of type int32, representing the shape for randomly generated keep/drop flags.
- name : A name for this layer (optional).

In our DNN model taking the inputs from the dense layer and Kept 0.8 as the probability that each element is kept in the whole dataset. As shown below

dropout1 = tflearn.dropout(dense1, 0.8)

5.6.4 Adam Optimizer

The Adam optimization algorithm is an extension to stochastic gradient descent, Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training.

But in Adam optimizer learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. It computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.

Adam Configuration Parameters

- **alpha**. Also referred to as the learning rate or step size. The proportion that weights are updated (e.g. 0.001). Larger values (e.g. 0.3) results in faster initial learning before the rate is updated. Smaller values (e.g. 1.0E-5) slow learning right down during training
- **beta1**. The exponential decay rate for the first moment estimates (e.g. 0.9).

- **beta2**. The exponential decay rate for the second-moment estimates (e.g. 0.999). This value should be set close to 1.0 on problems with a sparse gradient (e.g. NLP and computer vision problems).
- **epsilon**. Is a very small number to prevent any division by zero in the implementation (e.g. 10E-8).

Compare to other optimizers like stochastic gradient descent, RMSprop, Momentum ,etc.
Adam optimizer is working fine for our dataset used it as shown below
sgd = tflearn.Adam(learning_rate=0.01, beta1=0.99)

5.6.5 Output Layer

Softmax function or normalized exponential function is a generalization of the logistic function that squashes a K-dimensional vector z of arbitrary real value to a K dimensional vector of real values.where each entry in the range (0,1) and all the entries are add up to 1.
Function is given by:

$$\sigma : \mathbb{R}^K \to \left\{ \sigma \in \mathbb{R}^K | \sigma_i > 0, \sum_{i=1}^{K} \sigma_i = 1 \right\}$$

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

The softmax function is used in various multiclass classification methods and this can be seen as the composition of K linear functions as shown $\mathbf{x} \mapsto \mathbf{x}^\mathsf{T}\mathbf{w}_1, \ldots, \mathbf{x} \mapsto \mathbf{x}^\mathsf{T}\mathbf{w}_K$
The operation is equivalent to applying a linear operator defined by w to vectors x, thus transforming the original, probably highly-dimensional, input to vectors in a *K*-dimensional space outputs. It used as shown below
softmax = tflearn.fully_connected(dropout3,2, activation='softmax')

CHAPTER 6 TESTING

Testing systems that don't always return the same answers require new approaches. This is especially true when testing systems whose responses adapt to what they have learned from previous training. Software testing, in theory, is a fairly straightforward activity. For every input, there should be a defined and known output. We enter values, make selections, or navigate an application and compare the actual result with the expected one. But there is a type of software where having a defined output is no longer the case. Actually, two types. One is machine learning systems; the second is predictive analytics. Most Deep learning systems are based on neural networks. A neural network is a set of layered algorithms whose variables can be adjusted via a learning process. The learning process involves using known data inputs to create outputs that are then compared with known results.

Selected features are applied to three different classifier to measure the accuracy of classification as follows:-

6.1 Bagging Classifier

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

```
BaggingClassifier(base_estimator=None, bootstrap=True,
bootstrap_features=False, max_features=1.0, max_samples=1.0,
n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
verbose=0, warm_start=False)
```

Arguments:

base_estimator : object or None, optional (default=None)

The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.

bootstrap : boolean, optional (default=True)

Whether samples are drawn with replacement.

bootstrap_features : boolean, optional (default=False)

Whether features are drawn with replacement.

max_features : int or float, optional (default=1.0)

The number of features to draw from X to train each base estimator.

- If int, then draw max_features features.
- If float, then draw max_features * X.shape[1] features.

max_samples : int or float, optional (default=1.0)

The number of samples to draw from X to train each base estimator.

- If int, then draw max_samples samples.
- If float, then draw max_samples * X.shape[0] samples.

n_estimators : int, optional (default=10)

The number of base estimators in the ensemble.

n_jobs : int, optional (default=1)

The number of jobs to run in parallel for both fit and predict. If -1, then the number of jobs is set to the number of cores.

oob_score : bool

Whether to use out-of-bag samples to estimate the generalization error.

random_state : int, RandomState instance or None, optional (default=None)

If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

verbose : int, optional (default=0)

Controls the verbosity of the building process.

warm_start : bool, optional (default=False)

When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new ensemble.

Random Forest is one of the most popular and most powerful machine learning algorithms. It is a type of ensemble machine learning algorithm called Bootstrap Aggregation or bagging classifier is shown below.

```
from sklearn.ensemble import BaggingClassifier

model = BaggingClassifier()

print(model)

start = time.time()

model.fit(X_train, y_train)

end = time.time()

print ('Training in %.2f'%(end-start))

y_pred = model.predict(X_test)

print (confusion_matrix(y_true=y_test, y_pred=y_pred))

print(classification_report(y_pred,y_test))
```

Output:

This algorithm Trained in 0.14 sec

	precision	recall	f1-score	support
0	1.00	1.00	1.00	49
1	0.65	0.94	0.77	16
2	0.00	0.00	0.00	0
3	0.24	0.24	0.24	25
4	0.77	0.67	0.72	76
5	0.00	0.00	0.00	2
6	0.93	0.93	0.93	30
7	0.00	0.00	0.00	1
8	0.33	1.00	0.50	1
avg / total	0.76	0.75	0.75	200

Table 6.1 Classification report of Bagging Classifier

In the above table 6.1 shows the classification report of Bagging Classifier.

Precision

It gives what proportion of positive identifications was actually correct and it calculated using below formula

$$\boxed{\text{Precision} = \frac{TP}{TP + FP}}$$

Where TP –True Positive, FP-False Positive

Recall

It gives what proportion of actual positives was identified correctly and it calculated using below formula

$$\text{Recall} = \frac{TP}{TP + FN}$$

Where TP-True Positive, FN-False Negative

F1 Score

The F_1 score is the harmonic average of the Precision and Recall, where an F_1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.It calculated as follows.

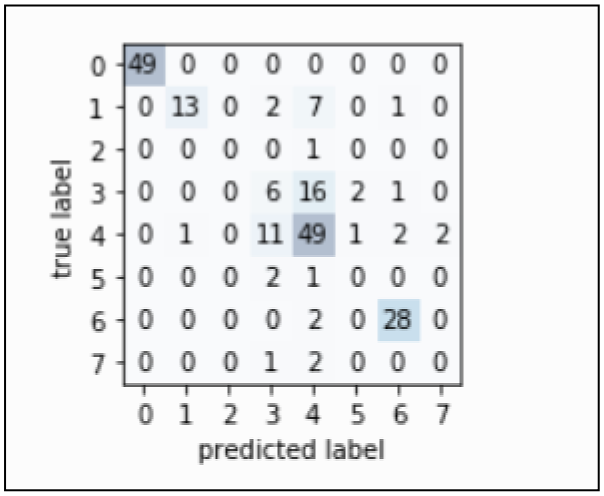
$$F_1 = \frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Accuracy

Accuracy is one metric for evaluating classification models. Informally, accuracy is the fraction of predictions our model got right. Formally, accuracy has the following definition:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

The below fig 6.1 shows the confusion matrix of Bagging classifier.



	0	1	2	3	4	5	6	7
0	49	0	0	0	0	0	0	0
1	0	13	0	2	7	0	1	0
2	0	0	0	0	1	0	0	0
3	0	0	0	6	16	2	1	0
4	0	1	0	11	49	1	2	2
5	0	0	0	2	1	0	0	0
6	0	0	0	0	2	0	28	0
7	0	0	0	1	2	0	0	0

Fig 6.1 Bagging Classifier Confusion matrix

This Algorithm gives the overall Accuracy =0.725

6.2 Adaboost Classifier

An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

```
AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,

learning_rate=1.0, n_estimators=50, random_state=None)
```

Arguments:

base_estimator : object, optional (default=DecisionTreeClassifier)

The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper classes_ and n_classes_ attributes.

n_estimators : integer, optional (default=50)

The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

learning_rate : float, optional (default=1.)

Learning rate shrinks the contribution of each classifier by learning_rate. There is a trade-off between learning_rate and n_estimators.

algorithm : {‘SAMME’, ‘SAMME.R’}, optional (default=’SAMME.R’)

If ‘SAMME.R’ then use the SAMME.R real boosting algorithm. base_estimator must support calculation of class probabilities. If ‘SAMME’ then use the SAMME discrete boosting algorithm. The SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations.

random_state : int, RandomState instance or None, optional (default=None)

If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

Below code shows the Adaboost classifier.

```
from sklearn.ensemble import AdaBoostClassifier

model = AdaBoostClassifier()

print (model)

start = time.time()

model.fit(X_train, y_train)

end = time.time()

print ('Training in %.2f'%(end-start))

y_pred = model.predict(X_test)

con=confusion_matrix(y_target=y_test, y_predicted=y_pred)

print (classification_report(y_pred,y_test))

fig, ax = plot_confusion_matrix(conf_mat=con)

plt.show()

print("Accuracy=%s"% accuracy_score(y_test,y_pred))
```

Output:

This algorithm Trained in 0.31 sec

	precision	recall	f1-score	support
0	1.00	1.00	1.00	49
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	1.00	0.44	0.61	151
5	0.00	0.00	0.00	0
6	0.00	0.00	0.00	0
8	0.00	0.00	0.00	0
avg / total	1.00	0.57	0.70	200

Table 6.2 Classification report of Adaboost Classifier

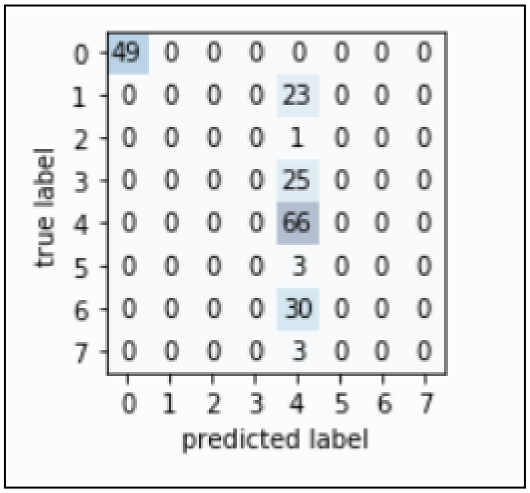


Fig 6.2 Adaboost Classifier Confusion matrix

This Algorithm gives the overall Accuracy=0.575

6.3 XGB Classifier

XGB (Xtreme Gradient Boosting) belongs to a family of boosting algorithms that convert weak learners into strong learners. A weak learner is one which is slightly better than random guessing. Boosting is a sequential process; i.e., trees are grown using the information from a previously grown tree one after the other. This process slowly learns from data and tries to improve its prediction in subsequent iterations.

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
n_jobs=1, nthread=None, objective='multi:softprob', random_state=0,
reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
silent=True, subsample=1)
```

Arguments:

base_score:

The initial prediction score of all instances, global bias.

booster: string

Specify which booster to use: gbtree, gblinear or dart.

colsample_bylevel : *float*

Subsample ratio of columns for each split, in each level.

colsample_bytree : *float*

Subsample ratio of columns when constructing each tree.

gamma : *float*

Minimum loss reduction required to make a further partition on a leaf node of the tree.

learning_rate : *float*

Boosting learning rate (xgb's "eta")

max_delta_step : *int*

Maximum delta step we allow each tree's weight estimation to be.

max_depth : *int*

Maximum tree depth for base learners.

max_depth : *int*

Maximum tree depth for base learners.

missing : *float, optional*

Value in the data which needs to be present as a missing value. If None, defaults to np.nan.

n_estimators : *int*

Number of boosted trees to fit.

n_jobs : *int*

Number of parallel threads used to run xgboost. (replaces nthread)

nthread : ***int***

Number of parallel threads used to run xgboost. (Deprecated, please use n_jobs)

objective : ***string or callable***

Specify the learning task and the corresponding learning objective or a custom objective function to be used (see note below).

random_state : ***int***

Random number seed. (replaces seed)

reg_alpha : ***float (xgb's alpha)***

L1 regularization term on weights

reg_lambda : ***float (xgb's lambda)***

L2 regularization term on weights

scale_pos_weight : ***float***

Balancing of positive and negative weights.

seed : ***int***

Random number seed. (Deprecated, please use random_state)

silent : ***boolean***

Whether to print messages while running boosting.

subsample : ***float***

Subsample ratio of the training instance.

These parameters are used to define XGB classifier as shown below.

```
from xgboost import XGBClassifier
model1 = XGBClassifier()
start = time.time()
model1.fit(X_train, y_train)
print (model1)
end = time.time()
print ('Training in %.2f sec'%(end-start))
y_pred = model1.predict(X_test)
multiclass=confusion_matrix(y_target=y_test, y_predicted=y_pred)
print (classification_report(y_pred,y_test))
fig, ax = plot_confusion_matrix(conf_mat=multiclass)
plt.show()
print("Accuracy=%s"% accuracy_score(y_test,y_pred))
```

Output:

This algorithm Trained in 1.65 sec

	Precision	recall	f1-score	support
0	1.00	1.00	1.00	49
1	0.57	1.00	0.72	13
2	0.00	0.00	0.00	1
3	0.36	0.43	0.39	21
4	0.82	0.64	0.72	84
5	0.00	0.00	0.00	2
6	0.93	1.00	0.97	28
8	0.67	1.00	0.80	2
avg / total	0.80	0.78	0.78	200

Table 6.3 Classification report of XGB Classifier

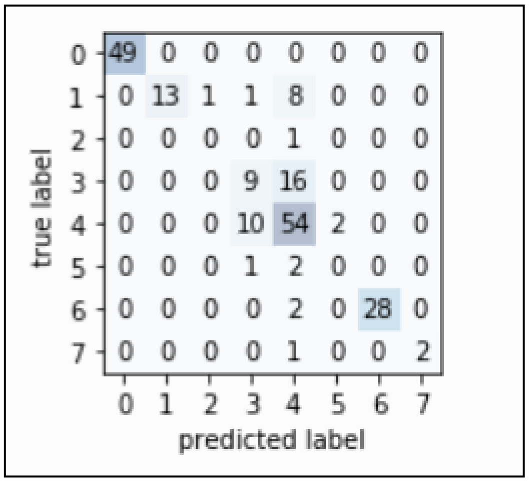


Fig 6.3 XGB Classifier Confusion matrix

This Algorithm gives the overall Accuracy=0.775

CHAPTER 7

RESULTS AND SCREEN SHOTS

A deep neural network (DNN) is an artificial neural network (ANN) with multiple hidden layers between the input and output layers. DNNs can model complex non-linear relationships. DNN architectures generate compositional models where the object is expressed as a layered composition of primitives. The extra layers enable composition of features from lower layers, potentially modeling complex data with fewer units than a similarly performing shallow network. Deep architectures include many variants of a few basic approaches. Each architecture has found success in specific domains. It is not always possible to compare the performance of multiple architectures, unless they have been evaluated on the same data sets. DNNs are typically feed forward networks in which data flows from the input layer to the output layer without looping back.

In this we divided dataset of size 1000 samples to Training set as 500 samples, Validation set as 250 samples and Testing set as 250 samples.

Evaluated this model using as shown below

model.evaluate(X=test_dataset, Y=test_labels)

This DNN model gives the 86% as the overall accuracy of classification. Below figure shows the confusion matrix of this model prediction.

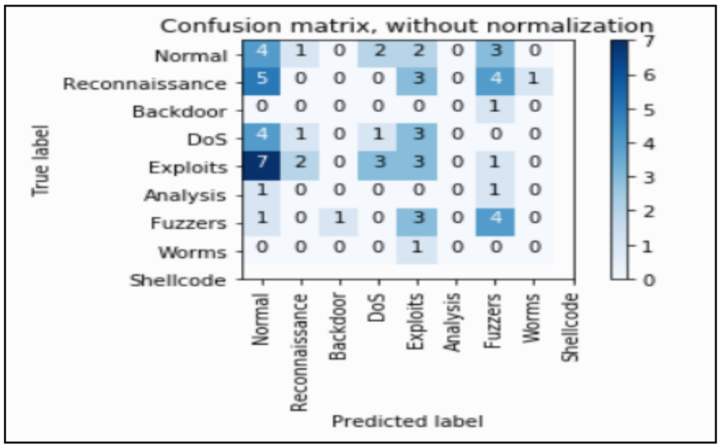


Fig 7.1 DNN Classifier Confusion matrix

CHAPTER 8

CONCLUSION AND FUTURE ENHANCEMENTS

8.1 CONCLUSION

The computing world has a lot to gain from neural networks. Their ability to learn by example makes them very flexible and powerful. Furthermore there is no need to devise an algorithm in order to perform a specific task; i.e. there is no need to understand the internal mechanisms of that task. They are also very well suited for real time systems because of their fast response and computational times which are due to their parallel architecture.

The classification results were slightly better in the three layer neural network. From practical point of view experimental results suggested that there is more to do in the IDS based on ANN. The cause of an intrusion detection system is to invention a potential intruder as possible as. An approach for a neural network based intrusion detection system motivated to classify the normal and attack packets further classifies to what type of attack by using DNN model of core layer. “Network Traffic classification using Deep Learning” simplifies the manual bootstrapping of network traffic labels, once provided to the system and it able to classify with high accuracy.

8.2 FUTURE ENCHANCEMENT

In the future enhancing the classification accuracy of our model Deep Neural Networks and enhancing the problem statement much more broader so that problem can be solved effectively in Deep Neural Networks once in the traffic it identified that traffic leads to attack then classifying that traffic leads to what type of attack and how we can overcome with automatic network troubleshooting.
In future, planning to implement a real-time NIDS for real time dynamic network traffic using deep learning technique. Additionally, making feature learning on raw network traffic headers instead of derived features using raw headers can be another high impact research in this area.

BIBLIOGRAPHY/REFERENCES

[1] Van, Nguyen Thanh, Tran Ngoc Thinh, and Le Thanh Sach. "An anomaly-based network intrusion detection system using Deep learning." *System Science and Engineering (ICSSE), 2017 International Conference on*. IEEE, 2017.

[2] Lotfollahi, Mohammad, et al. "Deep Packet: A Novel Approach For Encrypted Traffic Classification Using Deep Learning." *arXiv preprint arXiv:1709.02656* (2017).

[3] Wang, Zhanyi. "The applications of deep learning on traffic identification." *BlackHat USA* (2015).

[4] Dong, Bo, and Xue Wang. "Comparison deep learning method to traditional methods using for network intrusion detection." *Communication Software and Networks (ICCSN), 2016 8th IEEE International Conference on*. IEEE, 2016.

[5] Thing, Vrizlynn LL. "IEEE 802.11 Network Anomaly Detection and Attack Classification: A Deep Learning Approach." *Wireless Communications and Networking Conference (WCNC), 2017 IEEE*.

[6] Huang, Wenhao, et al. "Deep architecture for traffic flow prediction: deep belief networks with multitask learning." *IEEE Transactions on Intelligent Transportation Systems 15.5 (2014): 2191-2201*.

[7] Taimur Bakhshi and Bogdan Ghita "Research Article on Internet Traffic Classification: A Two phased Machine Learning Approach" Hindawi Publishing Corporation Journal of Computer Networks and Communications Volume 2016, Article ID 2048302.

[8] Zhang, Jun, et al. "Network traffic classification using correlation information." *IEEE Transactions on Parallel and Distributed systems* 24.1 (2013): 104-117.