Deep Learning Approaches for Network Traffic Classification In Software Defined Network

Semira Ahmed Abdu

A Thesis Submitted to the Department Computer Science and Engineering

College of Electrical Engineering and Computing

Presented In Partial Fulfillment of the requirement of Degree Master's

Computer Science and Engineering

Office of Graduate Study

Adama Science and Technology University

Sep,2025

Adama, Ethiopia

Deep Learning Approaches for Network Traffic Classification in Software Defined Network

Semira Ahmed Abdu

Advisor: Dr.Teklu Urgessa(PhD)

A Thesis Submitted to the Department of Computer Science and Engineering

College of Electrical Engineering and Computing

Presented in Partial Fulfillment of the requirement of Degree Master's Computer Science and Engineering

Office of Graduate Study

Adama Science and Technology University

Sept, 2025

Adama, Ethiopia

Declaration

I hereby declare that this Master's Thesis, titled "Deep Learning Approaches for Network Traffic Classification in Software Defined Networks," represents my original work and has not been submitted previously for the award of any academic degree, diploma, or certificate at this or any other institution. All sources of information and materials utilized in the preparation of this thesis have been appropriately acknowledged through citation.

Semira Ahmed Abdu _____ _____

Name of the Student Signature Date

Recommendation

I, the undersigned advisor of this thesis, hereby confirm that I have reviewed the revised version of the manuscript entitled "Deep Learning Approaches for Network Traffic Classification in Software Defined Networks" prepared under my supervision by Semira Ahmed Abdu. This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering (CSE). Based on my evaluation, I recommend that the revised version be submitted to the department through the appropriate official channels.

_____	_____	_____
Major Advisor	Signature	Date

Approval Sheet

I, the advisor of the thesis entitled “Deep Learning Approaches for Network Traffic Classification in Software Defined Network” developed by Semira Ahmed Abdu, hereby certify that the recommendation and suggestions made by the board of examiners are appropriately incorporated into the final version of the thesis

Major Advisor	Signature	Date

We, the undersigned, members of the Board of Examiners of the thesis by Semira Ahmed Abdu have read and evaluated the thesis entitled “Deep Learning Approaches for Network Traffic Classification in Software Defined Network” and examined the candidate during the open defense. This is, therefore, to certify that the thesis is accepted for the partial fulfillment of the requirement of the degree of Master of Science in Computer Science and Engineering (CSE).

Chairperson	Signature	Date
Internal Examiner	Signature	Date
External Examiner	Signature	Date

Final approval and acceptance of the thesis proposal is contingent upon submission of its final copy to the Office of Postgraduate Studies (OPGS) through the Department Graduate Council (DGC) and School Graduate Committee (SGC).

Department Head	Signature	Date
School Dean	Signature	Date
Office of Postgraduate Studies Dean	Signature	Date

ACKNOWLEGNMENT

First, I wish to express my sincere gratitude to Almighty God for providing me with the opportunity to undertake and complete this endeavor, and for granting me the strength and perseverance throughout this journey. I extend my profound appreciation to my advisor, Dr. Teklu Urgessa (PhD), for his invaluable guidance and support from the initial stages to the completion of this work. His insightful feedback and constant encouragement have been instrumental in shaping this thesis, and I am deeply grateful for his mentorship. My heartfelt thanks also go to my family, especially my husband, and to my friends, for their unwavering support and encouragement during both the challenging and rewarding moments of my studies at the University. I am also thankful to all the staff members of the ASTU Computer Science and Engineering Department for their advice, motivation, and guidance, which have been essential in introducing me to the world of academic research.

Table of Contents

List of Tables

List of Figures

List of Acronyms and Abbreviation

Adam	Adaptive Moment Estimation
AI	Artificial Intelligence
BERT	Bidirectional Encoder Representations from Transformers
CSV	Comma-Separated Values
DL	Deep Learning
DNN	Deep Neural Network
DNP	Distributed Network Protocol
DPI	Deep Packet Inspection
FTP	File Transfer Protocol
IDS	Intrusion Detection System
IoT	Internet of Things
IP	Internet Protocol
KNN	K-Nearest Neighbors
ISP	Internet Service Provider
LSTM	Long Short-Term Memory
NLP	Natural Language Processing
PCAP	Packet Capture
QoS	Quality of Service
ResNet	Residual Networks
RF	Random Forest
RNN	Recurrent Neural Network
SDN	Software-Defined Networking
SAE	Stacked Autoencoder
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VPN	Virtual Private Network

VAE	Variational Autoencoder
TLS	Transport Layer Security
UDP	User Datagram Protocol
VPN	Virtual Private Network
VAE	Variational Auto Encoder

Abstract

The exponential growth of network traffic demands advanced management solutions. Software-Defined Networking (SDN) addresses this need by providing a centralized control mechanism to measure, manage, and predict traffic. However, the volume of data processed by the SDN controller is substantial. Machine learning (ML) techniques have emerged as a viable method for analyzing this data. Accurate network traffic classification is vital for effective resource allocation, security, and granular network control. Traditional methods, including port-based analysis and deep packet inspection (DPI), have become less effective due to the computational demands of modern internet applications. This paper proposes a novel deep learning model for SDN environments designed to accurately classify a diverse range of traffic applications with high efficiency. The model demonstrated superior performance compared to conventional methods, achieving an accuracy of 92%, a precision of 92%, a recall of 88%, and an F1-score reflecting this high reliability. The findings suggest promising directions for future research to further enhance traffic classification in software-defined networks.

Keywords: Traffic classification, Software-defined networking, Artificial Intelligence, Deep Neural Network, Stacked Autoencoder, Random Forest

CHAPTER ONE
1.INTRODUCTION

1.1. Background

The landscape of modern networking is undergoing a profound transformation with the advent of Software Defined Networking (SDN), a groundbreaking architecture that fundamentally reimagines network design, management, and operation (Jiang et al., 2022). In contrast to traditional networks, where control and data forwarding are tightly integrated within physical devices like switches and routers, SDN introduces a paradigm shift by decoupling these functions. The control plane, which handles decision-making, routing policies, and network intelligence, is centralized within a software-based controller. This controller communicates with the underlying data plane through programmable interfaces. This separation not only simplifies management but also introduces unprecedented flexibility and scalability.

A core challenge in network management is traffic classification, which is the process of accurately identifying the nature of data flows traversing the network. Accurately classifying traffic whether it is web browsing, streaming media, voice calls, or malicious activities like malware and denial-of-service attacks is essential for ensuring optimal performance, security, and resource allocation. Traditional methods relied heavily on static signatures, port-based filtering, or statistical analysis; however, these approaches often prove inadequate, particularly when dealing with encrypted traffic or evolving protocols (Lotfollahi et al., 2018).

The recent surge in deep learning (DL), a subset of machine learning defined by neural networks with multiple layers that can perform automatic feature extraction, has demonstrated remarkable success in fields like image recognition, natural language processing, and speech analysis. Its application in network traffic classification is gaining momentum because DL models can learn complex, high-dimensional patterns directly from raw data, eliminating the need for manually engineered features. Within SDN frameworks, the centralized control architecture provides an ideal environment for deploying DL models, which can leverage a global view of the network to improve the accuracy and adaptability of traffic identification (Nazrul Hoque, 2024).

The rapid expansion of the internet, along with the proliferation of diverse communication devices and new technologies, has resulted in a massive increase in both the volume and complexity of network traffic (Reza et al., 2017). This growth introduces significant challenges in effectively measuring, analyzing, and managing traffic. The complexity is further compounded by the deployment of powerful hardware components routers, firewalls, servers, and switches that generate vast streams of data, rendering traditional traffic analysis methods insufficient. Consequently, network administrators face difficulties in optimizing traffic flow, detecting anomalies, and maintaining security in such dynamic environments.

To address these challenges, the concept of embedding intelligence into networks has gained prominence. The knowledge plane was proposed as an approach to integrate automation, reasoning, and learning capabilities into network systems, primarily through machine learning techniques. However, the distributed nature of traditional network architectures complicates the implementation of centralized intelligence. SDN's centralized control plane overcomes many of these issues by providing a unified, programmable interface where sophisticated algorithms particularly ML and DL can be employed to analyze network data, perform traffic classification, and make proactive decisions (Pu Zhao, 2021).

The control plane's visibility into the entire network state enables the development of models that can automatically learn features from raw traffic data, recognize patterns, and classify traffic types with high accuracy. This approach represents a significant improvement over conventional rule-based or port-based methods, especially in handling encrypted or non-standard traffic. Despite the scalability of SDN controllers, the volume of data they process can be overwhelming. This necessitates efficient algorithms like DL that can automatically distill meaningful features, thereby enhancing overall system performance and security. (X. Zhang, 2021).

1.1.1. Network Traffic Classification

Network Traffic Classification is the systematic process of assigning network flows to their respective application categories or protocol types. This is an essential step in managing network resources effectively and ensuring security. It forms the backbone of many operational tasks like Quality of Service (QoS) management, billing, intrusion detection, and anomaly prevention. As the diversity of applications and devices connected to the internet continues to grow exponentially, traditional classification techniques such as signature-based detection and

port filtering are increasingly insufficient. This is because they struggle to keep up with encrypted traffic, dynamic port usage, and sophisticated attack methods (Perera Jayasuriya Kuranage et al., 2020).

The significance of accurate traffic classification becomes even more pronounced within SDN environments. Here, centralized control enables the application of intelligent algorithms that utilize the network-wide view to better understand traffic patterns. This capability allows for more precise management and security policies, which facilitates a rapid response to threats and efficient resource utilization.

1.1.2. Deep Learning

Deep Learning (DL) enhances traditional ML by employing neural networks with multiple layers that can automatically learn hierarchical feature representations from raw data. This capacity to autonomously extract relevant features makes DL particularly suitable for complex, high-dimensional datasets encountered in network traffic analysis (Luca Gioachini, 2024).

1.2. Motivation of the study

The explosive growth of network traffic, driven by IoT, cloud computing, and multimedia applications, poses a significant challenge for traditional traffic analysis and security methods. Encrypted traffic, evolving protocols, and increasing attack sophistication demand smarter, adaptable solutions.

Deep learning's ability to automatically learn complex patterns from raw data, combined with SDN's centralized control architecture, creates an ideal environment for developing high-precision, scalable traffic classifiers. These models can adapt to network changes in real time, detect anomalies early, and improve security posture.

This motivation stems from the necessity to develop robust, intelligent traffic classification systems that can operate efficiently in large-scale, dynamic SDN environments, ultimately leading to more secure, manageable, and optimized networks.

1.3. Statement of the Problem

As networks continue to grow in size and complexity, traditional traffic classification methods often based on static signatures, port numbers, or statistical features are increasingly inadequate. They struggle with encrypted traffic, non-standard protocols, and real-time demands, which results in reduced accuracy and delayed responses.

The core challenge is to develop deep learning-based traffic classifiers that can handle the scale, diversity, and dynamism of modern SDN environments. These models must be capable of accurately identifying various traffic types, including benign and malicious flows, while using minimal manual feature engineering.

Although many traditional traffic classification methods were once effective, the widespread adoption of encryption techniques such as HTTPS, tunneling, and port-hopping has diminished their usefulness. These changes, combined with the challenges of class imbalance in network traffic and the scalability of deep learning models, have created significant gaps in achieving high-accuracy traffic classification.

Despite the promise of machine learning (ML) and deep learning (DL) techniques, key issues such as model robustness, feature extraction, and the ability to adapt to evolving traffic patterns remain largely unresolved. Consequently, the main problem lies in developing more efficient, adaptive, and scalable models that can handle the complexities of modern network traffic, particularly in encrypted environments, while also addressing class imbalance and data scarcity.

1.4. Research questions

1.5. RQ1. How can the optimal hyperparameters for deep learning models in SDN traffic classification be determined?

1.6. RQ2. What are the network traffic features that most significantly affect classification accuracy and efficiency?

1.7. RQ3. How well do different architectures for deep learning handle SDN traffic under contrasting conditions regarding accuracy?

1.8. Objective of the study

1.8.1. General Objective

1.8.2. The objective of this study is to analyze the effectiveness of DL algorithms in the classification of network traffic in SDN systems.

1.8.3. Specific objectives

- Identify the hyperparameters of deep learning (learning rate, batch size, activation functions, dropout rates, optimizers), tune them, and explore their impact on the classification accuracy, precision, recall, and F1-score.
- Investigate the impact of varying network traffic features (packet size, flow length, header information, protocol) on the performance of DL models and determine the most critical factors with regard to classification accuracy.
- Assess model performance using key metrics like accuracy, precision, recall, and F1-score.

.

1.9. Scope and Limitation of the study

1.9.1. Scope of the study

This study focuses on applying advanced deep learning architectures for efficient and accurate network traffic classification in Software Defined Networks (SDNs). The research tests a deep learning-based approach designed to enhance the speed and accuracy of traffic classification within SDNs. The objective is to achieve results that surpass the performance of traditional methods, including machine learning, port-based analysis, and deep packet inspection (DPI), particularly as network traffic volumes increase.

1.9.2. Limitation of the study

The widespread adoption of Machine Learning (ML) and Deep Learning (DL) for network traffic classification faces several significant challenges:

Data Availability and Quality: Supervised DL models require extensive, accurately labeled datasets for effective training. Acquiring such data for network traffic is particularly difficult, demanding substantial resources and domain expertise. The performance and general applicability of a proposed DL model are often constrained by the scarcity and quality of these labeled datasets.

Explain ability and Transparency: The "black box" nature of many DL models obscures the logic behind their predictions. This lack of interpretability makes it challenging to validate how traffic is classified, which is a critical concern in security-sensitive applications where understanding the decision-making process is essential.

Absence of Standardized Datasets: The field lacks universally accepted, representative public benchmarks for evaluation, unlike domains such as image processing with established datasets like ImageNet. This scarcity is primarily due to the sensitive nature of network data, which raises privacy issues, and the considerable effort required to create realistic and comprehensive datasets.

Ethical and Privacy Concerns: The use of DL for traffic classification introduces ethical questions related to data protection and user privacy. Any study in this area must proactively address these concerns and ensure compliance with relevant regulations, especially when handling potentially sensitive information.

Dynamic Data Nature: Network data is not static; it typically arrives as a continuous stream, which presents inherent analytical difficulties compared to fixed datasets.

Significance of the Study

Networks are fundamental to modern life, underpinning communication, finance, healthcare, and transportation. The increasing complexity of networks, driven by the growth of the Internet of Things (IoT) and connected devices, complicates traffic management and underscores the need for advanced classification techniques. The significance of integrating DL with SDN includes:

Enhanced Accuracy: DL techniques can achieve superior classification accuracy compared to traditional methods. This enables network administrators, operators, and service providers to gain better control over their networks, identify security threats more rapidly, and optimize overall performance. The scalable architecture of SDN, which separates the control and data planes, simplifies the deployment of these intelligent systems.

Scalability and Efficiency: DL algorithms can be trained on vast datasets and are capable of processing high volumes of data with both speed and precision. This makes DL particularly well-suited for managing traffic in large-scale SDN environments.

Adaptability: DL models offer greater flexibility than rigid, rule-based methods. They can learn directly from raw data and adapt to evolving network conditions and traffic patterns, capturing a level of complexity that traditional approaches cannot.

Improved Security: Securing network traffic is a paramount challenge in SDN. DL models can be trained to identify anomalies in traffic patterns, providing a powerful tool for detecting potential security breaches and threats.

Driving Innovation: As a rapidly advancing field of AI, DL has the potential to revolutionize network management. Applying these techniques to SDN allows researchers and engineers to develop novel solutions, leading to the creation of new tools and technologies that expand the capabilities of software-defined networks.

In summary, DL offers substantial advantages over traditional ML for traffic classification within SDN. Its strengths lie in automatic feature extraction, handling large-scale complex data, learning intricate patterns, and demonstrating continuous improvement, positioning it as a transformative technology for future networks.

CHAPTER TWO

2. LITERATURE REVIEW AND RELATED WORKS

2.1. The Role of Traffic Classification Techniques in Modern Networking

As network infrastructures evolve rapidly, the volume and variety of data traversing these systems have grown exponentially. Accurately identifying and categorizing this traffic is essential for maintaining security, managing resources efficiently, and ensuring optimal distribution. Traffic classification enables critical functions such as access control, content-based billing, anomaly detection, and malware identification, making it vital for seamless network operations.

Historically, techniques like port-based analysis and deep packet inspection (DPI) served as the primary methods for traffic classification. However, the widespread adoption of encryption protocols such as HTTPS has diminished the effectiveness of these approaches. These traditional methods often rely on inspecting packet contents or port numbers, which are now commonly obscured or randomized. To overcome these limitations, researchers have increasingly turned to machine learning (ML) techniques that leverage pattern recognition and feature extraction for more effective traffic classification. These methods have demonstrated promising outcomes, particularly when enhanced with improved feature selection and refined classification models.

Despite these advancements, two significant challenges persist. First, network traffic data typically exhibits class imbalance, where certain applications or protocols generate substantially more traffic than others. This uneven distribution complicates the task of achieving accurate classification across all categories. Many ML models prioritize overall accuracy, which can result in the poor recognition of minority classes that are critical for security or policy enforcement. Acknowledging these issues, recent research explores the integration of deep learning (DL) with Software-Defined Networking (SDN) and the application of transfer learning to enhance model robustness and adaptability.

This review investigates the convergence of networking, machine learning, deep learning, and transfer learning, with a focus on their practical applications in traffic classification. A deeper understanding of these techniques can contribute to superior network performance, enhanced Quality of Service (QoS), and more resilient security measures.

2.1.1. Port-based Traffic Classification: A Classic Approach

One of the earliest and simplest techniques involves linking network traffic to known services or applications using common port numbers like HTTP on port 80 or HTTPS on port 443. By matching traffic flows to these registered ports, administrators could easily categorize data streams. This method's simplicity made it popular at first (Ibrahim et al., 2021), but it also had major limitations. As modern applications increasingly use techniques like tunneling, port hopping, and random port assignments, the reliability of port-based classification decreases. These methods, often employed to avoid detection or enhance privacy, create serious challenges and weaken the effectiveness of port-based approaches.

2.1.2. Payload-based Traffic Classification: Peering into Packet Content

Payload-based methods, known as Deep Packet Inspection (DPI), examine the actual data within packets. They focus on application-layer information to identify the specific service or application that generates the traffic (Siddiqi & Pak, 2020). These methods use predefined signatures or pattern matching, similar to digital fingerprints, to tell different protocols or applications apart. While they work well in controlled environments, DPI is less effective with encrypted traffic and can require a lot of computing power, which makes it less ideal for high-speed networks.

2.1.3. Harnessing Machine Learning and Deep Learning for Traffic Classification

The rise of ML and DL has changed how networks recognize and handle traffic. These methods can learn complex patterns in data on their own, resulting in more accurate and flexible classification systems. They create new opportunities in network optimization, security, and application management. However, using DL models requires large amounts of quality data and careful setup to avoid issues like false alarms or missed detections. These models need extensive datasets for training, and their complexity calls for thorough validation to prevent false positives that could cause unnecessary disruptions.

2.1.1.1. Improving QoS with ML and DL

Quality of Service (QoS) management ensures that important applications like video conferencing or VoIP receive priority bandwidth. DL improves this by classifying traffic more accurately. This allows for dynamic resource allocation based on current traffic patterns. It can also predict future network loads and adjust QoS settings as needed (Sivalingam, 2021; Dhillon & Haque, 2020; Zhang et al., 2022). This helps reduce congestion and latency.

• Dynamic bandwidth allocation: DL models analyze traffic to assign resources where they are needed most.

• **Predictive analytics**: Anticipating traffic surges allows for early adjustments, keeping service smooth.

• **Traffic management:** Finding bottlenecks and congestion points helps improve overall network performance.

2.1.1.2. Traffic Engineering: Fine-Tuning the Flow

Traffic engineering involves managing data flows to prevent bottlenecks and improve network use. DL algorithms help with real-time resource allocation and load balancing. They ensure network traffic is spread evenly and efficiently across multiple paths, reducing latency and preventing service loss (Qiu et al., 2022).

• **Resource allocation:** DL models learn traffic patterns to assign bandwidth on the fly.

• **Load balancing:** Distributing traffic smartly across multiple paths boosts throughput and resilience.

2.1.1.3. Application Identification: Recognizing Digital Footprints

Deep learning models can be trained to recognize distinct signatures of specific applications or services using flow characteristics, payload patterns, or even natural language descriptions. Methods like unsupervised learning can find unknown applications by grouping similar traffic patterns. Meanwhile, NLP models examine textual data related to applications for classification purposes (Trinh et al., 2020).

2.1.1.4. Fault Detection: Spotting the Anomalies

DL models are great at finding problems in large datasets. They examine system logs, sensor data, or network traffic to spot faults or failures early. This allows for quick maintenance and reduces downtime. The process involves training models on data from normal operations and

marking any deviations as potential issues (Ring et al., 2021).

2.1.1.5. User Identification: Securing Digital Identities

From facial recognition to voice authentication, DL techniques have improved user identification in terms of accuracy and efficiency. Neural networks analyze biometric features like fingerprints, iris patterns, and behavioral metrics such as typing rhythm or navigation patterns to verify identities quickly and securely (Ring et al., 2021; Pešek et al., 2022).

2.2. Applications of ML and DL in SDN Environments

Software-Defined Networking provides a flexible, programmable framework that allows ML and DL techniques to be easily combined to improve network intelligence. By training models on real-time traffic data, network administrators can classify flows, predict congestion, and detect threats. This approach leads to more resilient and efficient networks (Aouedi et al., 2020).

2.2.1. Deep Learning Meets SDN: A Symbiotic Relationship

In SDN, the central controller, like Open Daylight, serves as the command hub and manages network behavior. DL models are placed at key locations such as the network edge, data centers, or cloud platforms to analyze traffic in real-time. This setup allows for quick decision-making.

- **Edge deployment:** Enables immediate classification at the entry points and allows for instant QoS adjustments.
- **Data center deployment:** Helps monitor traffic specific to applications and optimizes resource use.
- **Cloud-based deployment:** Provides insights for scalable, global traffic management and user behavior analysis.The integration of DL with SDN creates a powerful ecosystem capable of responding adaptively to network demands, security threats, and changing traffic patterns.

2.2.2. DL-Driven Traffic Management: Enhancing Network Control

DL algorithms enhance the detail and precision of traffic classification in SDN. They allow for real-time anomaly detection, traffic prediction, and smart policy enforcement. These functions together boost network reliability and security.

2.2.3. Network Security

Despite the significant advantages of Software-Defined Networking (SDN), security remains a persistent challenge (Elsayed et al., 2020). Intrusion Detection Systems (IDS) are fundamental security tools, prized for their ability to identify novel attacks. These systems are designed to detect intrusions attempts to compromise a network's confidentiality, integrity, or availability, or to bypass its security mechanisms. By alerting administrators to such attempts, IDS enable proactive measures to prevent or mitigate potential damage (Guoyan Tian and Hongtao Du).

This paper explores the application of Deep Learning (DL) for anomaly detection in SDN traffic. The proposed method utilizes a Deep Belief Network (DBN) to model the complex relationships between network traffic features, aiming to identify anomalies with a high degree of accuracy (Liu & Lang, 2019).

In summary, the reviewed literature indicates that DL techniques hold considerable promise for traffic classification within SDN environments. These methods can reduce administrative overhead, enhance classification accuracy, and facilitate real-time, application-aware traffic management, leading to superior network control. Nevertheless, further research and testing are required to overcome existing challenges, including data scarcity, model complexity, and the inherent diversity of modern networks.

Related Works

Table 1. Literature reviews on DL in SDN network traffic classification

Ref.	Area of the study	Model used	Number of features and approaches Used.	Model Output	Accuracy	Limitations
M.M.Raiker,2020	Framework for the classification of application	SVM, NB	Fourteen traffic flow attributes	3 classes	NB: 96.79% SVM: 92.3%	No information provided on the size of the dataset used
H. A. H. Ibrahim, 2021	DL with traditional classification techniques	Deep CNN for classification	DT preselect relevant features	UNSW-NB15 dataset & 9 network attacks classes		Limited dataset and attack types
M. Zhang.2022	DNN-based traffic classification	SAE	Automatic by algorithm	10 classes	SAE: Training accuracy: 90% Testing: 79%	High computational cost, may not be suitable for real-time
A. Malik, R.2020	Deep-SDN	DNN	Automatic by algorithm	10 classes	96.00%	Limited number of features
Z.Fan.2017	ML-based Traffic Classification techniques	SVM and K-Means	30 flow features	8 classes	SVM: 98.7% K-M: 88.0%	The study considers limited number of features
J. Chen.2022	Combined novel deep neural network	CNN, RNN	Automatic by algorithm	6 applications	99.0%	Limited number of applications and protocols

2.1.2. Existing Studies and Future Directions

Current research demonstrates significant advances, with studies reporting traffic classification accuracy exceeding 96% using deep neural networks and successful anomaly detection through deep belief networks (Liu & Lang, 2019). However, several challenges persist, particularly in handling diverse real-world traffic conditions, minimizing computational demands, and ensuring model adaptability to evolving network environments.

CHAPTER THREE

3. MATERIALS AND METHODS

3.1. Architectural Framework for Traffic Classification in SDN

In creating a strong traffic classification system within the SDN framework, the design relies on separating the data plane from the control plane. You can think of the data plane as the highway that is only for sending packets through network hardware. The control plane works like a traffic coordinator, directing routes and enforcing rules through an external controller. Communication between these two areas happens smoothly using the Open Flow protocol, which ensures fast and reliable data exchange. To improve flexibility and lessen reliance on proprietary hardware vendors, open-source APIs are used. This allows developers to build and launch various network software solutions.

Integrating Deep Learning (DL) into this architecture improves its efficiency. By keeping a clear separation between data and control layers and allowing smooth communication through open protocols, DL boosts the system's ability to classify traffic accurately and flexibly. The open-source APIs provide the foundation for software customization, allowing for innovation without being limited by hardware constraints.

3.2. Method for Traffic Classification in SDN

Implementing a DL-based traffic classifier requires choosing the right tools, software, datasets, and hardware to achieve significant results in network management and security.

3.2.1. Network Traffic Dataset: The Foundation

The success of a Deep Learning (DL) model is fundamentally dependent on the quality of its training dataset. For effective network traffic classification in Software-Defined Networking (SDN), a high-quality dataset must comprehensively represent real-world network conditions. This involves capturing a diverse spectrum of applications, protocols, and network behaviors.

Data is typically gathered using packet capture tools or existing network monitoring infrastructure. A critical step is the careful labeling of this data into distinct categories, such as web browsing, video streaming, and voice over IP (VoIP). Including this variety ensures the model learns from accurate and representative traffic patterns, which is essential for robust performance when deployed in live environments.

3.2.2. Hardware Resources

The effective training and evaluation of deep learning models require substantial computational resources. The experimental setup for this project utilizes a high-performance computing environment to manage these intensive tasks. The core hardware is a Satellite S55t-A laptop, equipped with an Intel Core i7-4700MQ CPU operating at 2.40 GHz, 16 GB of RAM, and high-speed storage. This configuration provides the necessary processing power, memory capacity, and data throughput for efficient model development.

3.2.3. Software Ecosystem and Libraries

A robust software toolkit is essential for transforming raw data into actionable insights. The implementation leverages industry-standard frameworks and libraries. TensorFlow serves as the foundational platform for constructing and training complex neural networks, while Keras provides a high-level API to streamline the development process. For data manipulation and analysis, NumPy and Pandas are employed. The Scikit-learn library facilitates feature extraction and preprocessing, and Matplotlib and Seaborn are used for data visualization and result interpretation. This integrated software ecosystem supports the entire workflow, from initial data processing to the final deployment of the trained model.

Table 2. Hardware and Software Specifications

DL framework tools	Available interface
TensorFlow	Python
Keras (higher level library for TensorFlow)	Python
Anaconda Navigator	Anaconda3
Spyder IDE	Python
ML models	RF (Random Forest), SAE (Stacked Autoencoder), DNN (Deep Neural Network)

3.2. Data Collection and Preparation: Crafting a Reliable Dataset

The first step is to gather representative network traffic data that fits SDN environments. This includes using traffic monitoring tools like Wireshark or using existing datasets. It is important to make sure the data collected follows legal and ethical guidelines, including obtaining permissions and handling sensitive information responsibly.

The dataset should show a wide range of application behaviors, covering different protocols and network conditions. Since public labeled datasets are limited, collecting custom data from real or simulated SDN environments often becomes essential. One notable example is the Unicauca dataset, a large collection of network flows (https://huggingface.co/datasets/yyy999/Unicauca-dataset-April-June-2019-Network-flows) from a university campus in Colombia. It features over 2 million flows labeled across 114 application types, encrypted with TLS to reflect real-world traffic.

3.4. Preprocess Network Traffic Data

Convert the packet (in PCAP format) into a suitable format for training.

3.4.1. Input Preprocessing

The data type for each input layer is based on the data type of the column in train features for that input. Numeric inputs are combined and then scaled with a Normalization layer. For inputs that are not floats (categorical features), use a String Lookup layer, which takes in the string values and maps them to integers. Then use a Category Encoding layer. Use these layers to create a functional preprocessing model to generate preprocessed inputs.

You can then use this model to preprocess new data before feeding it into a trained model. A visual representation of the train_preprocessing model is created. A dictionary of features is made to hold arrays of feature values. This pipeline prepares input features using TensorFlow and Keras, making the data ready for neural network processing in machine learning and deep learning tasks. The input data will be in the correct format to run the neural network.

3.4.2. Feature Engineering and Reshaping

In traditional ML, feature engineering is an important step to find the right features that can describe and distinguish network traffic. These features include statistical, packet-level, or flow-level data. It is crucial to design features that capture key characteristics of the traffic, such as payload, packet size, flow length, or time between packets. Common techniques that help improve model performance, prevent overfitting, and enhance generalization include:

Feature Extraction: Extract relevant features from network traffic data. Common features include packet size, flow duration, and payload characteristics.

3.4.3. Feature Selection and Extraction

This method identifies the most useful features for a problem and removes unnecessary ones, reducing data size and model complexity. The process involves cleaning the dataset by removing incomplete, irrelevant, or incorrect data. This includes deleting duplicates, handling outliers, filling missing values, and removing packet data not relevant to the network traffic of interest.

To identify meaningful features, the proposed model uses Random Forest (RF). This is done by selecting features from network traffic datasets that represent real-world traffic and influence model performance.

3.4.4. Feature Scaling

This involves scaling data to a common range so that every feature has equal importance and is on the same scale. This common preprocessing step in machine learning improves performance and helps algorithms converge. It is important to calculate minimum and standard deviation from the training set to standardize test and validation sets consistently. This ensures all data transformations are uniform, keeping train, test, and validation data on the same scale, which is crucial for model performance. This process prevents bias toward features with larger scales and aids algorithm convergence during training.

Data Dimensional Reduction: Many features increase computational complexity, resource usage, and time consumption. To address this, identifying relevant features from large network traffic datasets is necessary, as it also improves neural network accuracy. The training data is converted into a 2D array where the first dimension represents the number of samples and the second dimension represents the number of features. Finally, the dataset is normalized and scaled to aid the convergence of the training process.

3.4.5. Categorical Encoding and Data Splitting

Categorical variables, such as protocol types, are converted into a numerical format using one-hot encoding. This technique transforms each category into a binary vector, where each bit indicates the presence or absence of a specific category. The resulting binary matrix is suitable for machine learning models that require numerical input.

The preprocessed dataset is partitioned into training (70%), validation (15%), and testing (15%) sets using stratified sampling. This approach preserves the distribution of classes across all subsets. The training set is used for model learning, the validation set for hyperparameter tuning, and the test set for final performance evaluation.

Summary of Preprocessing Pipeline:

Data Cleaning: Handle missing values, remove duplicates and outliers, filter irrelevant data.

Feature Selection: Utilize Random Forest to identify and retain impactful features.

Feature Scaling: Standardize numerical features using parameters derived from the training set.

Dimensionality Reduction: Apply techniques to reduce feature space while preserving critical information.

Categorical Encoding: Convert categorical variables into binary representations via one-hot encoding.

Data Splitting: Partition data into training, validation, and test sets with stratified sampling.

Use the training set for training DL model and use validation set for performance evaluation of the model.

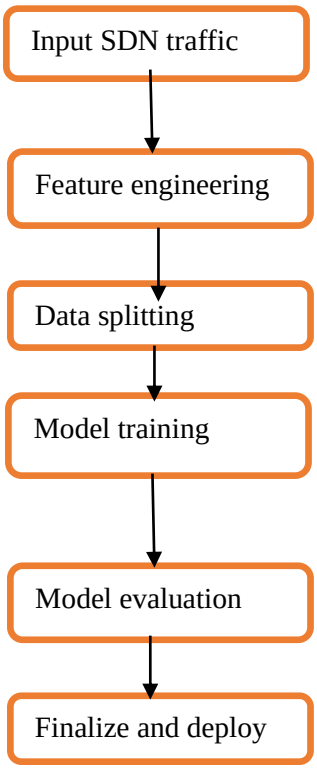


Figure 1 Data processing life cycle in DL-SDN integration

3.3. Model Development

Artificial Intelligence (AI) methodologies have catalyzed substantial progress across multiple network traffic management domains, including traffic engineering, intrusion detection, resource allocation, protocol analysis, and traffic synthesis (S. Soleymanpour et al., 2020). Nevertheless, contemporary approaches frequently depend on highly specialized models designed for particular applications. This strategy commonly encounters obstacles including limited datasets for specialized tasks, challenges in model training with sparse data, and the substantial development expenditures required for creating multiple task-specific solutions.

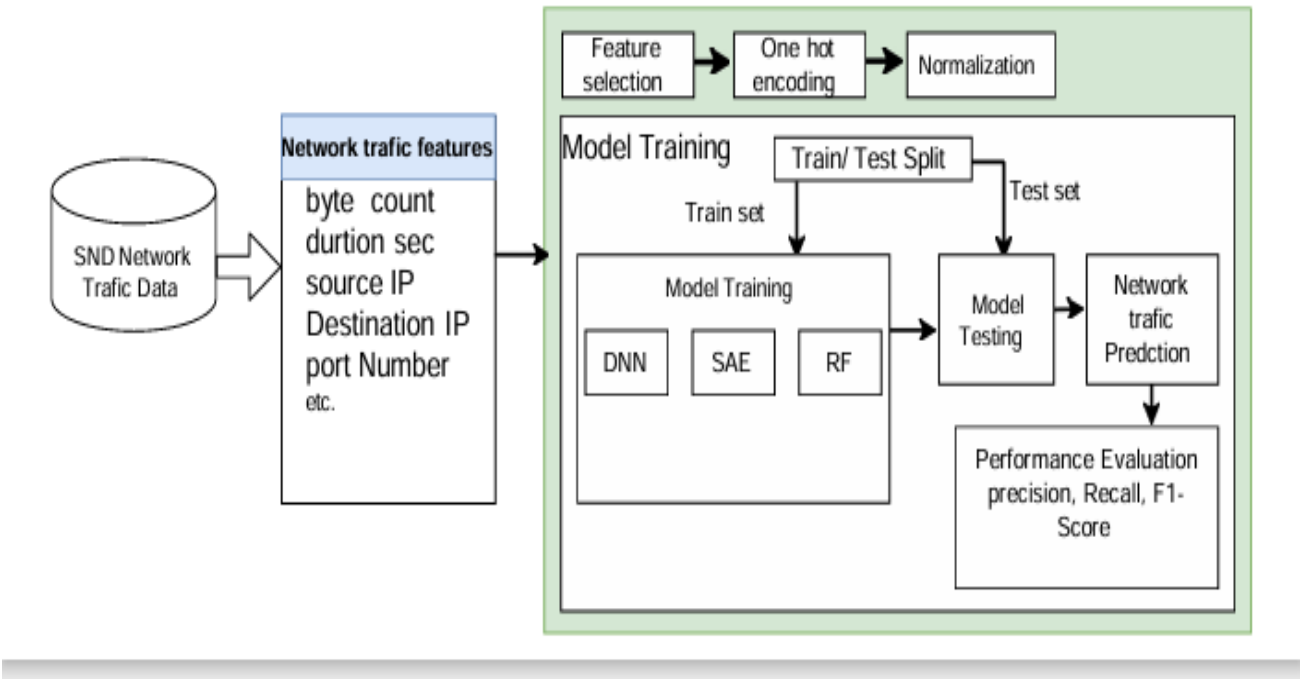


Figure 2 Diverse traffic classification tasks with different implementation

3.3.1. The Proposed Model Architecture

After systematic hyperparameter optimization, the final model architecture was determined. This process selected the optimal configuration for the number and kind of layers, activation functions, optimizers, batch size, dropout rates, and learning rate. The model is a feedforward neural network, characterized by its sequential data flow without cyclical connections between nodes. It is built using TensorFlow, a widely-used machine learning framework.

The model's deep learning structure includes these core elements:

Input Layer: This is the entry point for the training data into the model. Its dimensions are defined by the number of features in the dataset.

Hidden Layers: These are the primary computational layers situated between the input and output. Each layer contains neurons that apply weights to the inputs and process them through an activation function to produce an output. The model uses the ReLU (Rectified Linear Unit) activation function, which outputs the input value directly if it is positive, and outputs zero otherwise.

Normalization Layer: A normalization step is applied after each dense layer. This layer standardizes the outputs, scaling the features to improve the stability and performance of the subsequent dense layer.

Model Evaluation

Model evaluation quantifies a model's effectiveness in generating predictions from input data. This process is critical for machine learning and other predictive models, as it determines whether a model is sufficiently accurate and reliable for practical use. The deep learning model's performance will be assessed using metrics including accuracy, precision, recall, and the F1 score. Its results will also be compared against those from standard techniques. Evaluation involves analyzing the model's performance on a validation set with these metrics and a confusion matrix, then comparing the results to identify patterns and trends. The specific metrics applied to the proposed model are detailed below.

3.5.1.1. Accuracy

Accuracy metrics for deep learning are measures that calculate how well an algorithm performs in correctly predicting outcomes. These metrics are essential because they enable developers and researchers to determine if their models are functioning correctly and to make necessary improvements. Accuracy is the most frequently used metric for classification models. It is calculated as the number of correctly classified instances divided by the total number of instances. However, accuracy alone is not always a robust indicator of performance, particularly when dealing with imbalanced datasets where the number of samples in each class varies significantly. The formula for calculating accuracy is provided below:

$$\text{Accuracy} = (\text{TP} + \text{TN})/(\text{TP} + \text{TN} + \text{FP} + \text{FN}),$$

Where:

TP = True Positive

TN =True Negative

FP = False Positive

FN = False Negative

3.3.1.1. Precision

Precision is a metric that assesses a model's ability to avoid false positives. It measures the ratio of correctly identified positive instances (true positives) to all instances the model labeled as positive (including false positives). A high precision score indicates that when the model predicts a positive outcome, it is highly reliable. Precision is calculated with the following formula:

$$Precision = TP/(TP + FP)$$

3.3.1.2. Recall

Recall, also known as sensitivity, evaluates a model's capability to identify all relevant positive instances within a dataset. It measures the proportion of actual positives that were correctly identified. This metric is crucial for understanding if a model is effectively capturing all members of a target class. Recall is determined by the formula:

$$\text{Recall} = \text{TP}(\text{TP} + \text{FN})$$

3.3.1.3. F1 score

The F1-score provides a single metric that balances the trade-offs between precision and recall. It is the harmonic mean of the two, offering a more comprehensive view of a model's accuracy than either metric alone, especially with imbalanced datasets. An F1-score of 1 represents perfect precision and recall, while a score of 0 indicates the opposite. The formula for the F1-score is:

$$\text{F\_1 score} = (2 * \text{Precision} * \text{Recall})/(\text{Precision} + \text{Recall})$$

3.3.1.4. ROC curve

The Receiver Operating Characteristic (ROC) curve is a graphical plot used for binary classification models. It illustrates the diagnostic ability of a classifier by plotting the True Positive Rate (Recall) against the False Positive Rate at various classification thresholds. The Area Under the ROC Curve (AUC) summarizes the model's overall ability to distinguish between classes.

Key Interpretation of AUC-ROC values:

AUC = 1.0: Represents a perfect classifier.

AUC = 0.5: Suggests a model with no discriminative ability, equivalent to random guessing.

0.5 < AUC < 1.0: Indicates a model with predictive power, where values closer to 1.0 signify stronger performance.

.

3.3.1.5. Confusion Matrix

A confusion matrix is a tool used to evaluate the performance of a classification model. It presents a table that compares the model's predictions with the actual values. The matrix is structured around four key outcomes:

True Positives (TP): Instances where the model correctly predicts the positive class.

False Positives (FP): Instances where the model incorrectly predicts the positive class (the actual class is negative).

True Negatives (TN): Instances where the model correctly predicts the negative class.

False Negatives (FN): Instances where the model incorrectly predicts the negative class (the actual class is positive).

By analyzing the distribution of these outcomes, one can identify the specific types of errors a model makes. For instance, a high number of false positives indicates a problem with precision, while a high number of false negatives indicates a problem with recall. This makes the confusion matrix an effective method for diagnosing a model's strengths and weaknesses.

3.3.1.6. Area under curve

The Area Under the Curve (AUC) is a metric for evaluating the performance of classification algorithms. It represents the probability that a randomly selected positive instance will be ranked higher than a randomly selected negative instance by the model. This metric provides an aggregate measure of performance across all possible classification thresholds.

Calculation for Multi-Class Scenarios:

In multi-class classification, the AUC is computed for each class individually using a one-versus-rest approach, where each class is treated as the positive class in turn.

Aggregation Methods:

The class-specific AUC scores can be combined using these methods:

Micro-average: Calculates a global AUC by combining the true positives, false positives, true negatives, and false negatives from all classes.

Macro-average: Computes the AUC for each class independently and then averages these scores.

Weighted-average: Calculates the macro-average but weights each class's contribution based on its number of instances.

Interpretation:

A higher AUC value indicates better model performance. An AUC of 0.5 indicates that the model has no discriminative power, equivalent to random guessing. An AUC of 1.0 represents perfect classification ability.

3.3.1.7. Probability Distribution of Prediction

The probability distribution of predictions facilitates balancing the trade-off between recall and precision, enabling model predictions to be both accurate and practical. This approach helps determine the model's confidence level in its predictions and supports decision-making based on these outputs. It is particularly valuable in multi-label classification scenarios, where the complexity of data may lead to a single input being associated with multiple classes.

3.5.1.9. Resource Utilization

Profiling tools track computational demands, including CPU usage and training duration, which are critical for real-world deployment. These assessments guide iterative improvements, enhancing the model's robustness and reliability.

Training Time: Training time refers to the period required for a deep learning model to learn from a dataset and distinguish network traffic patterns. This duration depends on several factors, including dataset size, model complexity, and hardware capabilities. Training step time is a key consideration in model development and evaluation as it

impacts both the speed and efficiency of the learning process. Shorter training times generally indicate faster learning and lower computational requirements. However, very short step times achieved through small batch sizes may compromise the stability and accuracy of gradient updates. Consequently, a balance must be struck between training efficiency and model quality. The proposed model incorporates specific practices to optimize training duration.

CPU Usage: CPU and memory consumption are measured during both training and inference phases. A specialized profiling tool provides detailed monitoring of computational resource allocation.

Classification Report: A classification report summarizes model performance for each class using precision, recall, F1-score, and support metrics. This detailed breakdown helps identify which classes the model classifies effectively and where performance deficiencies exist.

3.6. Deep Learning SDN Integration

Software-Defined Networking (SDN) offers a programmable architecture that enables dynamic control of network behavior. Deep learning models can classify network traffic using multiple features. After training, these models are integrated with SDN controllers to enable real-time traffic classification. This integration supports the development and testing of DL-based solutions for software-defined networks, advancing capabilities in network management, security, and resource allocation within SDN infrastructures.

The following pseudo-algorithm demonstrates the integration of a deep learning model into an SDN environment for traffic categorization. The trained model identifies different traffic types, enabling the SDN controller to manage network operations more effectively.

Algorithm: SDN_Deep Learning_Traffic_Classification

Input: Network traffic data

Output: Traffic classification results

1: Initialize the SDN environment

2: Deploy SDN controllers and configure network topology

3: Collect network traffic data

4: Preprocess the traffic data for deep learning

Feature extraction and Normalization

5: Split the preprocessed data into training and testing sets

6: Define the deep learning model architecture

Input layer with size equal to the number of features

Hidden layers

Output layer with size equal to the number of traffic classes

7: Train the DL model using the training set

Choose a loss function and an optimizer

Validate the model on the testing set

8: Integrate the trained model with the SDN controller

Develop a module for the controller to input traffic data into the model

Use the model's predictions to classify network traffic

9: Define network policies based on traffic classification

Quality of Service rules

Security measures (firewall rules)

10: Monitor network traffic in real-time using SDN switches

11: Classify incoming traffic using the deep learning model

12: Dynamically apply network policies based on classification

End Algorithm

SDN controller follows the following measures based on network traffic classification results from a deep learning model.

Flow Classification:

The deep learning model classifies network traffic into different categories (e.g., video streaming, web browsing, VoIP).

Each flow is labeled based on its predicted application or service.

Traffic Policy Mapping: The SDN controller maps the flow labels to predefined traffic policies.

Policies define how specific types of traffic should be handled (e.g., prioritization, bandwidth allocation).

Policy Enforcement:

The controller enforces the defined policies across the network.

It instructs switches and routers on how to handle traffic based on the flow labels.

Dynamic Path Selection:

Using the deep learning results, the controller dynamically selects optimal paths for different traffic types.

It considers factors like latency, congestion, and available bandwidth.

Quality of Service QoS Configuration:

The controller configures QoS parameters for each flow category.

It ensures that critical applications receive the necessary resources.

Adaptive Traffic Steering:

Based on real-time traffic conditions, the controller adjusts flow paths.

It reroutes traffic away from congested links or faulty devices.

Security and Anomaly Detection:

The controller collaborates with security modules.

It identifies anomalies (e.g., DDoS attacks) based on traffic patterns.

Monitoring and Reporting:

The controller continuously monitors traffic behavior.

It generates reports on network performance, compliance, and policy violation

CHAPTER FOUR

4. RESULTS AND DISCUSION

This research implements a comprehensive network traffic classification system through a multi-stage analytical pipeline. The methodology encompasses several critical phases: initial data preprocessing to clean and normalize raw network data, followed by feature selection to identify the most discriminative attributes. Subsequent stages involve hyperparameter optimization and the selection of an appropriate deep learning architecture. The final implementation enables accurate multilabel classification of network traffic into distinct application categories.

4.2. Overview of the Unicauca-April-June-2019 Network Flows Dataset

The research utilizes the Unicauca-April-June-2019 Network Flows Dataset as its foundational training and validation resource. This comprehensive dataset contains approximately 2 million records, each representing detailed network flow characteristics with 209 distinct attributes. The dataset's richness provides a robust foundation for developing accurate traffic classification models.

Key features include:

- Timestamp (ts): When the traffic occurred.
- Flow duration: How long each connection lasted.
- Source and Destination IPs (src_ip, dst_ip): The addresses involved.
- Ports (src_port, dst_port): Identifiers for applications or services.
- Protocol (proto): Such as TCP or UDP.
- Packet count and total bytes: Volume metrics.
- Packet size: Average size of individual packets.

Aside from raw data, each record receives labels to indicate the type of service whether it's streaming, browsing, or file transfer making this dataset perfect for multilevel traffic classification. It's publicly available via platforms like Hugging Face, GitHub, and Kaggle,

hence a resource for researchers and experts alike.

4.3. Data Processing & Analysis: Preparing Data for Deep Learning

These procedures are essential for enabling the deep learning model to learn from high-quality data, which is a critical requirement for the accurate classification of network traffic in an SDN environment. Each step facilitates the model's ability to identify and learn underlying network traffic patterns. The process begins with preprocessing the collected dataset, which involves the removal of unwanted traffic entries, elimination of duplicate records, and encoding of numerical and categorical features. Data cleaning, feature extraction, normalization, and other necessary transformations were performed using tools including Scikit-learn and Pandas.

4.3.1. Handle Missing Values and Infinite Values

This portion of the code cleans the DataFrame by eliminating rows containing any null values and removing columns that are entirely empty. This ensures the data is devoid of null entries that could negatively impact the training process.

```
text
sdn_train = sdn_train.dropna()
with pd.option_context('mode.use_inf_as_na', True):
    sdn_train.dropna(inplace=True)
```

This cleaning process guarantees that the DataFrame contains no rows with missing or infinite values, a necessary step since such values can disrupt the classification model's operation. The code removes rows with null values, including those with infinity values, from the sdn_train DataFrame and applies the changes directly to it.

4.3.2. Remove Duplicate Values

The subsequent task involves identifying and removing duplicate values and inconsistent columns that would have no beneficial impact on the classification task's performance.

```
text
# Remove duplicate rows
sdn_train = sdn_train_ds.drop_duplicates()
```

The drop_duplicates() function call returns a copy of the sdn_train_ds DataFrame with duplicate rows removed. It identifies duplicates by comparing all columns in each row.

Original Dataset: 2,704,839 rows × 50 columns

After Deduplication: 2,704,829 rows × 50 columns

Duplicates Removed: 10 rows (0.00037% of original data)

4.3.1. Handling less relevant features

In this phase, several attributes (columns) are identified as less significant or redundant for the classification task. Removing these columns reduces the dataset's dimensionality and can enhance the performance of the classification models by focusing on more relevant attributes.

text

```
sdn_train = sdn_train.drop(['flow_key', 'src_ip', 'dst_ip'], axis=1)
```

The removal of 'flow_key', 'src_ip', and 'dst_ip' columns results in a training dataset with the revised shape = (2704829, 47).

4.3.4. One-Hot Encoding of Categorical Features

This section converts the categorical 'application_protocol' and 'web_service' columns into a numerical format compatible with deep learning algorithms by creating separate binary columns for each category.

text

```
# Encode categorical features

ohe = OneHotEncoder(sparse=False)

protocol_ohe = ohe.fit_transform(sdn_train['application_protocol'].values.reshape(-1, 1))

protocol_cols = ohe.get_feature_names_out(['application_protocol'])

protocol_df = pd.DataFrame(protocol_ohe, columns=protocol_cols)

# Concatenate the one-hot encoded protocol columns to the original dataframe

sdn_train = pd.concat([sdn_train, protocol_df], axis=1)
```

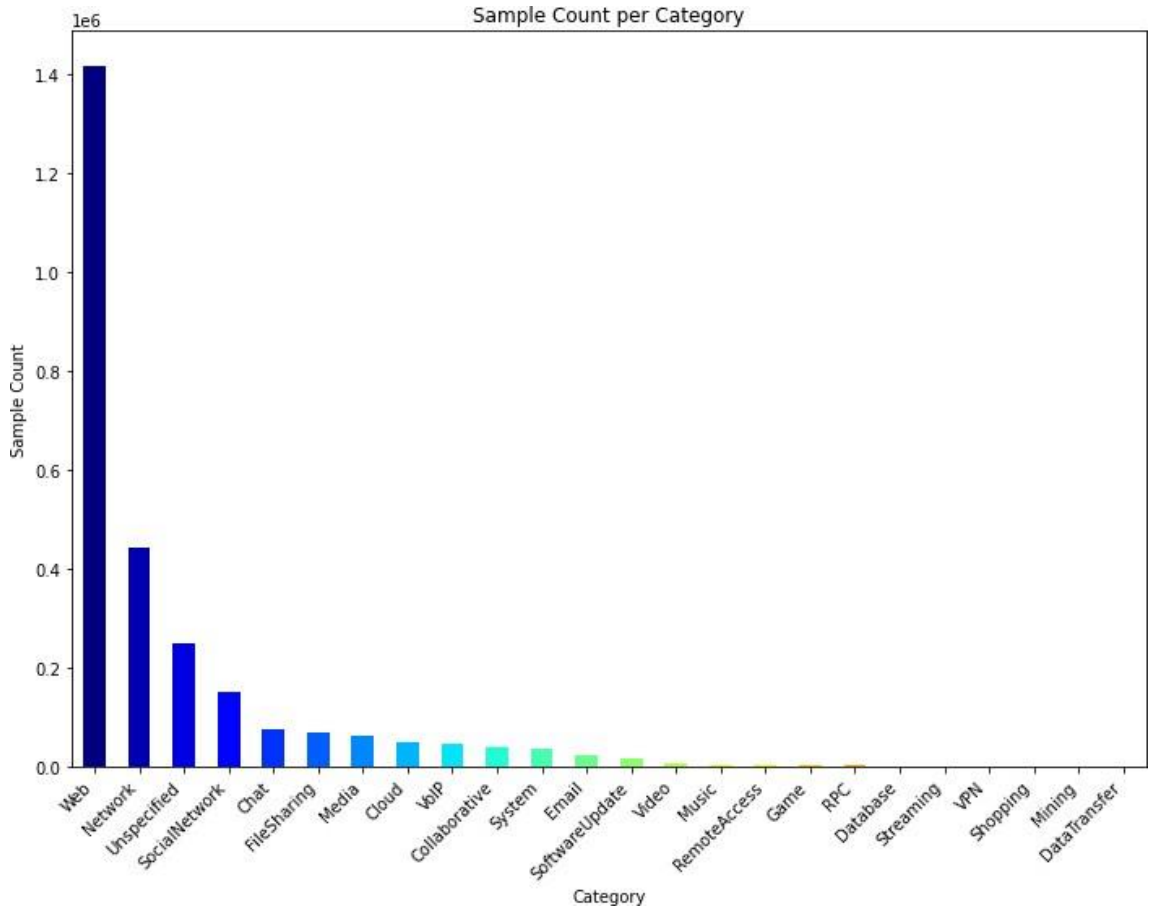
Drop the original protocol column

sdn_train = sdn_train.drop('application_protocol', axis=1)

This process is repeated for the 'web_service' column. The categorical columns are converted into numerical values and then concatenated with the original dataset. After completing these steps, the final dataset shape becomes = (2704829, 210).

Data Visualization:

The graph below represents the distribution of samples across different categories in the training dataset.



Each bar in the visualization represents a specific category, with its height indicating the number of samples belonging to that category. This graphical representation helps identify which categories are dominant and which have fewer instances in the dataset.

text

train_features = sdn_train.copy()

A copy of the sdn_train DataFrame is created to form a separate DataFrame containing only the feature variables.

text

train_labels = train_features.pop('category')

The 'category' column is extracted from the train_features DataFrame and stored separately as train_labels, effectively separating the input features from the target variable for model training.

text

print(sdn_train['category'].value_counts())

This command displays the distribution of samples across different categories in the target variable, showing the count of instances for each class.

Value Counts of Target Variable: print(sdn_train['category'].value_counts())

Table.1.Sample Traffic features

R/N	Column Name	Description
1	src_ip_numeric	Source device IP address in numeric format
2	src_port	Source device port number
3	dst_port	Destination device port number
4	Proto	Communication protocol used
5	pktTotalCount	Total number of packets in the flow
6	octetTotalCount	Total number of bytes in the flow
7	min_ps	Minimum packet size in the flow
8	max_ps	Maximum packet size in the flow
9	avg_ps	Average packet size in the flow
10	std_dev_ps	Standard deviation of packet sizes in the flow
11	FlowStart	Flow start timestamp
12	FlowEnd	Flow end timestamp
13	flowDuration	Total duration of the flow
14	f_flowStart	Forward flow start timestamp
15	f_flowEnd	Forward flow end timestamp
16	f_flowDuration	Duration of the forward flow

19	f_avg_piat	Average inter-arrival time between packets in the forward flow direction
20	b_std_dev_piat	Standard deviation of inter-arrival time between packets in the backward flow direction
21	Flow End Reason	Reason for flow termination
22	category	Classification category of the flow
23	application_protocol	Application layer protocol used for communication
24	web_service	Specific web service utilized for communication

The network traffic dataset contains the following traffic categories:

Table 2 Traffic category lists (target)

R/N	Traffic Category	Description
1	Web	Traffic generated by web browsers, servers, and applications. Typically unicast, this category encompasses common internet activities like web browsing, email, and file transfers.
2	Network	Traffic originating from network infrastructure devices including routers, switches, firewalls, and VPNs. Used for routing, switching, security, and administrative communication.
3	Unspecified	Traffic that cannot be classified into other categories. May result from data collection errors, classification issues, or unknown/emerging applications not recognized by network systems.
4	Social-network	Traffic from social media platforms, typically unicast or multicast. Used for content sharing, messaging, and social interactions.
5	Chat	Communication traffic from messaging applications, usually unicast. Includes voice/video calls, text messaging, file sharing, and group discussions.
6	Download-File Transfer-FileSharing	Data transfer traffic for downloading and file sharing, which can be unicast or multicast depending on packet addressing.
7	Media	Traffic from streaming services (e.g., Netflix), video conferencing platforms (e.g., Zoom), and online gaming environments.

/N	Traffic Category	Description
7	Media	Traffic from streaming platforms (Netflix), video conferencing (Zoom), online gaming (Steam), music services (Spotify), and radio broadcasts.
8	Cloud	Traffic generated by cloud computing (AWS), storage (Google Drive), backup services, and cloud gaming platforms.
9	VoIP	Voice over Internet Protocol traffic from services like Skype, WhatsApp Voice, and Google Voice.
10	Collaborative	Traffic from collaborative tools including Google Docs, Microsoft Office 365, Dropbox Paper, and Slack.
11	System	Traffic from system administration, monitoring tools, diagnostic utilities, and security applications.
12	Email	Traffic associated with email services such as Gmail, Outlook, and Yahoo Mail.
13	Software Update	Traffic generated by software update services including Windows Update and macOS update mechanisms.
14	Video	Traffic from video-specific services including streaming platforms, video conferencing, and online gaming.
15	Music	Audio-related traffic, typically unicast or multicast, for online music listening, podcast downloads, and radio streaming.
16	Remote Access	Traffic from remote desktop and access applications like TeamViewer, AnyDesk, and LogMeIn.
17	Game	Traffic generated by gaming platforms such as Steam, Epic Games Store, and Origin.
18	RPC	Traffic from Remote Procedure Call services including Google Cloud Functions, AWS Lambda, and Azure Functions.
19	Database	Traffic related to database operations for data storage, querying, updates, and online backups.
20	Streaming	Traffic from live and on-demand streaming services like

		YouTube, Twitch, and Facebook Live.

		(VPN) applications and services such as NordVPN, ExpressVPN, ProtonVPN etc.
22	Shopping	Shopping traffic is usually unicast or multicast depending on the source and destination addresses of the packets.
23	Mining	Traffic from mining applications and services, such as Bitcoin mining, Ethereum mining, etc. Mining traffic can be used for various purposes such as generating cryptocurrency, validating transactions, securing networks, etc.
24	Data Transfer	This category includes traffic from data transfer applications and services, such as FTPS (FTP over SSL/TLS), P2P (Peer-to-Peer), BitTorrent (BT), etc.

The following section outlines a reusable input data preprocessing model designed for retraining with various models. The training features dictionary contains the input features, which are processed according to their data types.

Feature Processing: For numeric features (float32), the data is added to numeric inputs.For non-numeric features (object), additional processing is required.

Normalization and Concatenation:

Numeric features are combined into a single tensor (X) and normalized using a normalization layer. For each non-numeric feature:

A string lookup layer (lookup) is created with the unique vocabulary

A one-hot encoding layer (one_hot) processes the lookup results

The processed non-numeric input is added to preprocessed_inputs

All preprocessed inputs (both numeric and one-hot encoded) are concatenated into a single tensor (preprocessed_inputs_cat).

Preprocessing Model Creation:

A Keras model (train_preprocessing) is defined with input features and concatenated preprocessed inputs. This model maps raw input features to preprocessed features. A features dictionary (features_dict) is created with example features, and the preprocessing model is applied to the training data (features_dictionary). This reusable model eliminates the need for repeated data processing tasks.

4.3.5. Feature Selection

This step extracts meaningful features from network traffic data to serve as input for the deep learning model. The goal is to identify a subset of features most relevant for the classification task.

A Random Forest Classifier is employed for feature selection based on importance scores. Random forests create multiple decision trees during training and aggregate their predictions. After training on features (x) and target (y), the model estimates each feature's importance relative to the target variable.

The implementation: Separates target and independent variables: x (feature matrix) and y (target vector). Trains the RandomForestClassifier model

Extracts feature importance scores

Identifies the least significant feature

Removes the least important feature from the dataset

Text

'x' is feature matrix and 'y' is the target vector x = train features y = train labels

Train the model

model = RandomForestClassifier() model.fit(x, y)

Get feature importances

importances = model.feature_importances_

print("Feature importances:", importances)

Sort feature importances and get indices of least important features

indices_to_remove = importances.argsort()[:1]

```
print("Indices to remove:", indices_to_remove)

# Print names of least important features

print("Names of least important features:", [train_labels[i] for i in indices_to_remove])

# Remove least important features from dataset

X = np.delete(x, indices_to_remove, axis=1)
```

The np.delete function removes the column at the specified indices_to_remove from the feature matrix x. The axis=1 parameter indicates that a column (feature) should be deleted, not a row. This operation produces a new feature matrix X with the least important feature excluded, which may enhance model performance if that feature was noisy or irrelevant. This procedure can be repeated iteratively to remove multiple unimportant features. Essentially, this process identifies the most relevant features that impact model performance.

Following the feature selection phase, the training dataset has a final shape of (2704829, 208). Among these features, the top 20 most important features impacting the efficiency of machine learning models including Random Forest (RF), Stacked Autoencoder (SAE), and Deep Neural Network (DNN) are described below:

dst_port: The destination port number used by the receiver, crucial for identifying the intended service type for the traffic.

max_ps: The maximum packet size observed in the traffic, which can indicate the volume and type of data being transmitted.

f_max_ps: The maximum packet size observed in the forward direction (from source to destination).

web_service_DNP3: A binary feature indicating the detection of the Distributed Network Protocol 3, commonly used in industrial, power grid, and utility communications.

b_max_ps: The maximum packet size observed in the backward direction (from destination to source).

min_ps: The minimum packet size observed in the traffic.

avg_ps: The average packet size across the entire traffic flow, providing insight into the consistency of the data flow.

application_protocol_TLS: A binary feature indicating the presence of Transport Layer Security, used for secure communication.

std_dev_ps: The standard deviation of packet sizes, reflecting the variability in packet sizes within the traffic.

web_service_Unencrypted_Jabber: A binary feature indicating the use of the Jabber/XMPP protocol without encryption.

src_port: The source port number used by the sender, which can help identify specific applications or services.

web_service_FTP_DATA: A binary feature indicating the use of the File Transfer Protocol for data transfer.

src_ip_numeric: The numeric representation of the source IP address, useful for geolocation or network topology analysis.

f_flowStart: The timestamp marking the start of the flow in the forward direction.

flowEnd: The timestamp indicating when the flow ended.

These features are essential for RF, SAE, and DNN classification models as they help in understanding the behavior and nature of network traffic, which is fundamental for accurate traffic classification. Understanding these features facilitates easier interpretation of the model's decisions and can lead to performance improvements. The following figure illustrates the 20 most important network traffic features that potentially affect the classification performance of the proposed model, compared to baseline models (RF and SAE).

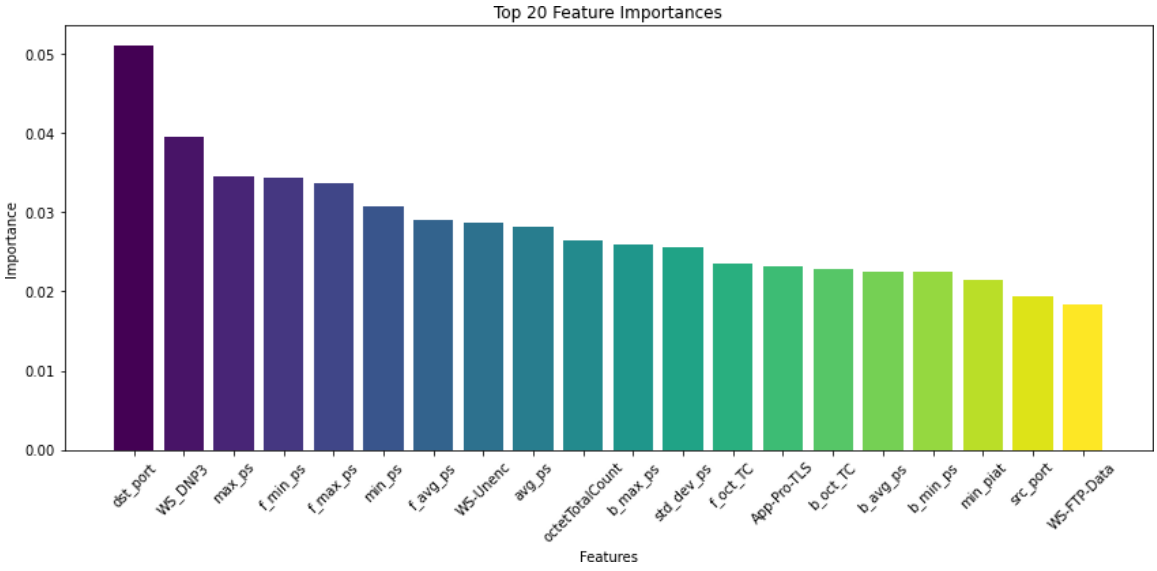


Figure 3 the top 20 most important features

4.4. Data splitting

The dataset is partitioned into training, validation, and test subsets, with the validation split being stratified according to the training labels. This procedure prepares data for deep learning by dividing it into distinct sets for model training, hyperparameter tuning, and final evaluation. This approach enables model training, parameter optimization, and assessment on unseen data. The validation set facilitates model tuning without involving the test set, which is reserved exclusively for final testing.

text

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=0)

This code segment splits the features (X) and labels (y) into training (X_train, y_train) and testing (X_test, y_test) sets. The test_size=0.15 parameter allocates 15% of the data to the testing set, while the remaining 85% is used for training. The random_state=0 parameter ensures reproducible results across multiple executions.

text

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.15, stratify=y_train, random_state=0)

This further divides the initial training set into a refined training set (X_train, y_train) and a validation set (X_val, y_val). The test_size=0.15 parameter allocates 15% of the original training data to the validation set. The stratify=y_train parameter maintains the same class distribution in the validation set as in the training set, ensuring a representative sample.

4.4.1. One-Hot Encoding

Categorical values (network traffic types) are converted into binary vectors. This representation is essential for multi-label classification problems where the model needs to output probability scores for each class. One-hot vectors serve as the standard representation for classification tasks in neural networks.

4.4.2. Label Encoding

This technique assigns unique numerical values to each categorical value. For instance, with

three traffic types, the first type might be encoded as 0, the second as 1, and the third as 2. This encoding method is commonly employed in classification problems.

text

```
#label encoding
from sklearn.preprocessing import LabelEncoder
label_enc = LabelEncoder()
label_enc.fit(y)
y = label_enc.fit_transform(y)
classes = label_enc.classes_
```

4.4.3. Categorical Encoding

This approach replaces categorical values with the mean of the target variable for that specific category. It is particularly useful when dealing with numerous categories that have frequent occurrences.

text

```
#Categorical Features
label_enc = LabelEncoder()
label_enc.fit(y)
y = label_enc.fit_transform(y)
```

The to_categorical function generates a one-hot encoded representation of labels, which is standard practice for handling categorical data in neural networks. The num_classes parameter specifies the total number of classes, ensuring the binary matrix has columns corresponding to the class count.

text

```
classes = label_enc.classes_
num_classes = len(np.unique(y_train))
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
y_val = tf.keras.utils.to_categorical(y_val, num_classes)
```

4.4.4. Feature Scaling

Feature scaling, specifically standardization, is applied separately to the training, testing, and validation datasets. This process calculates the mean and standard deviation for each dataset and standardizes the values by subtracting the mean and dividing by the standard deviation.

text

```
X_train = (X_train - np.mean(X_train)) / np.std(X_train)
X_test = (X_test - np.mean(X_test)) / np.std(X_test)
X_val = (X_val - np.mean(X_val)) / np.std(X_val)
```

4.1.1.1. Reshape the Features

text

```
# Reshape the features to match the input shape
input_shape = (X_train.shape[1],)
X_train = X_train.reshape(-1, *input_shape)
X_test = X_test.reshape(-1, *input_shape)
X_val = X_val.reshape(-1, *input_shape)
```

4.5. Define the Proposed Model

This model processes CSV files containing network traffic data, where each row represents a traffic example and each column represents a feature. After training, the model can classify new network traffic data into one of 24 categories based on learned patterns. The model's accuracy depends on data quality and quantity, architectural design, and training hyperparameters.

4.5.1. Hyperparameter Tuning

The necessary libraries, including TensorFlow and its dependencies, are imported. Optimal parameters are selected through a specialized model developed for hyperparameter tuning. The following code demonstrates the hyperparameter tuning process:

text

```
# Define a function to create the model with hyperparameters
```

```
def build_model(hp):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Dense(hp.Int('units_input', min_value=32, max_value=512, step=32), activation='relu', input_dim=208))
    # ..... (rest of code) .....
    return model

# Create a tuner
tuner = RandomSearch(
    build_model,
    objective='val_acc',
    max_trials=10,
    executions_per_trial=1,
    directory='my_dir',
    project_name='hparam_tuning'
)

# Perform hyperparameter tuning
tuner.search(X_train, y_train, epochs=5, validation_data=(X_val, y_val))

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the optimal hyperparameters and train it
model = build_model(best_hps)
```

The deep learning model is loaded with tuned parameters and trained on a multilabel dataset, specifically a DNN architecture. The model uses a sigmoid activation function in the output layer, appropriate for multilabel datasets, and is compiled with a binary_crossentropy loss function. The model structure comprises:

Sequential Model: A linear stack of layers where each layer processes one input tensor to produce one output tensor.

Dense Layers: Fully connected neural network layers where each layer's output is computed through weighted matrix multiplication, bias addition, and activation function application.

ReLU Activation: Applied in hidden layers to introduce non-linearity, enabling complex pattern learning. The function outputs the input directly if positive, otherwise outputs zero.

Sigmoid Output Layer: Contains 24 units corresponding to the data classes, generating probability distributions across all classes.

The model is compiled with the Adam optimizer (learning rate=0.001), known for efficiency in deep learning models. The loss function is binary_crossentropy, suitable for multilabel classification. Performance metrics include accuracy, F1 score, precision, and recall, all crucial for evaluating classification model performance.

The table below compares the optimized parameters for the proposed model.

:

Table 3.Parameters of the proposed model

Layer Type	Neurons	Activation function	Input/Output Shape	Dropout Rate	Description
Dense	512	ReLU	Input: 208	-	Fully connected layer
Batch Normalization	-	-	Output: 512	-	Normalization layer
Dropout	-	-	Output: 512	0.1	Dropout layer
Dense	256	ReLU	Output: 256	-	Fully connected
Batch Normalization	-	-	Output: 256	-	Normalization layer
Dropout	-	-	Output: 256	0.1	Dropout
Dense	128	ReLU	Output: 128	-	Fully connected l
Batch Normalization	-	-	Output: 128	-	Normalization
Dropout	-	-	Output: 128	0.1	Dropout
Dense	64	ReLU	Output: 64	-	Fully connected
Batch Normalization	-	-	Output: 64	-	Normalization
Dropout	-	-	Output: 64	0.1	Dropout
Dense	32	ReLU	Output: 32	-	Fully connected
Batch Normalization	-	-	Output: 32	-	Normalization layer
Dropout	-	-	Output: 32	0.1	Dropout
Dense	16	ReLU	Output: 16	-	Fully connected
Batch Normalization	-	-	Output: 16	-	Normalization

Dropout	-	-	Output: 16	0.1	Dropout
Dense (Output Layer)	24	Sigmoid	Output: 1	-	Binary classification layer

Dropout layers are incorporated into the model architecture to mitigate overfitting. During training, these layers randomly disable a specified fraction of input units (10% in this case) at each weight update cycle. This randomization prevents the network from becoming overly dependent on specific neurons and encourages more robust feature learning.

Optimizer: The Adam optimizer is employed, which is widely utilized in deep learning implementations. It uses adaptive learning rate methods to compute individual learning rates for different parameters.

Learning Rate: The default learning rate of the Adam optimizer, 0.001, is used.

Loss Function: Binary cross-entropy is applied. This loss function is suitable for multi-label classification tasks where each instance can belong to only one class.

Epochs: The model is trained for 100 epochs. One epoch represents a complete pass through the entire training dataset.

Metrics: Accuracy is used as the evaluation metric, which calculates the proportion of correctly classified instances.

Table 6 shows the summary of a SAE model the number of parameters, layer types, the output shape it produces, which has been trained on the same data as the proposed data.

Table 4.Parameters of the SAE model

Layer (Type)	Output Shape	# Parameters
Input_3 (Input layer)	(None, 208)	0
Dense_12 (dense)	(None, 128)	26,752
Dense_13 (dense)	(None, 64)	8,256
Dense_14 (dense)	(None, 32)	2,080
Dense_15 (dense)	(None, 16)	528
Dense_16 (dense)	(None, 8)	136
Dense_17 (dense)	(None, 16)	144
Dense_18 (dense)	(None, 32)	544
Dense_19 (dense)	(None, 64)	2,112
Dense_20 (dense)	(None, 128)	8,320
Dense_21 (dense)	(None, 24)	3,096

4.4.1. Training the Classification Model

The processed data from the previous phase is used to train three distinct models: Random Forest (RF), Stacked Autoencoder (SAE), and the proposed model. The following section verifies their effectiveness in classifying network traffic into various categories and detecting different types of network traffic based on the patterns and features learned from the data.

Train the RF model:

text

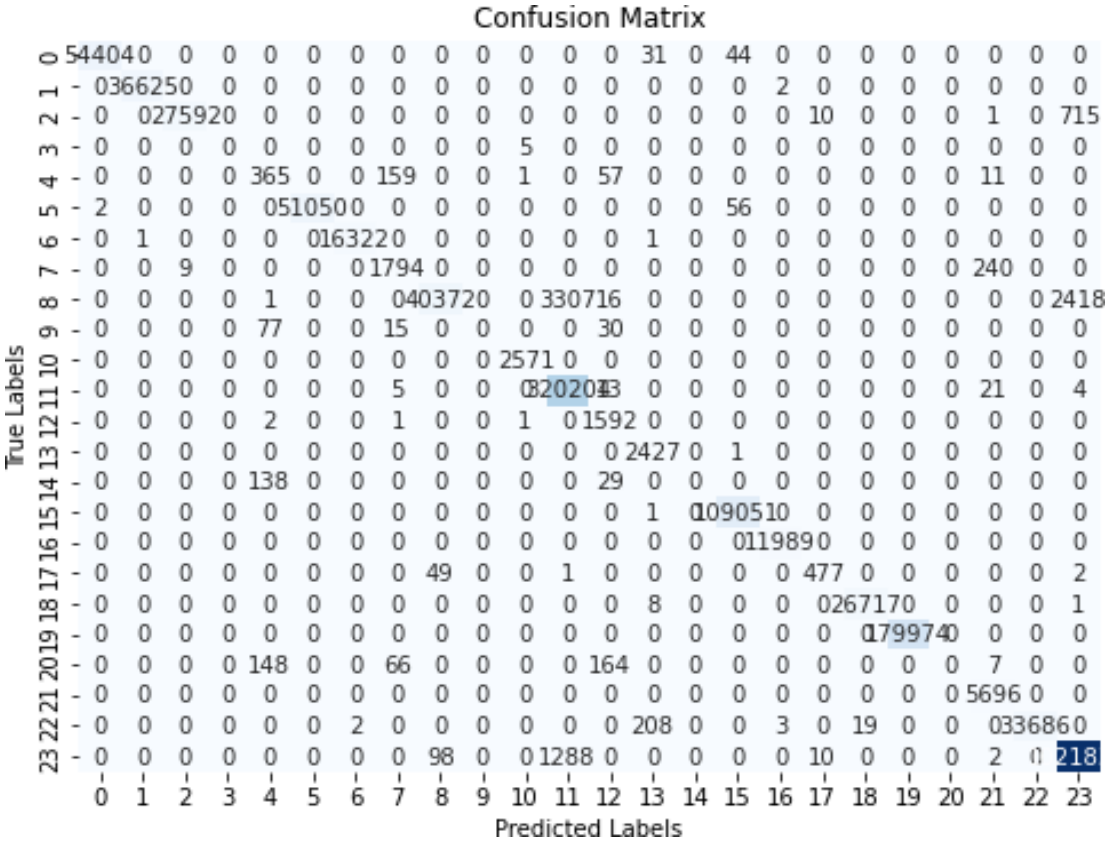
rfc = RandomForestClassifier()

rfc.fit(X_train, y_train)

Train the SAE and the proposed DNN model:

text

history = model.fit(X_train, y_train, batch_size=128, epochs=100, validation_data=(X_val, y_val))



4.5. Model Evaluation

The performance metrics from the deep learning model indicate that the proposed model is performing quite well.

Accuracy (acc): The accuracy exceeds 90%, indicating that the model correctly predicts the target class for the majority of instances.

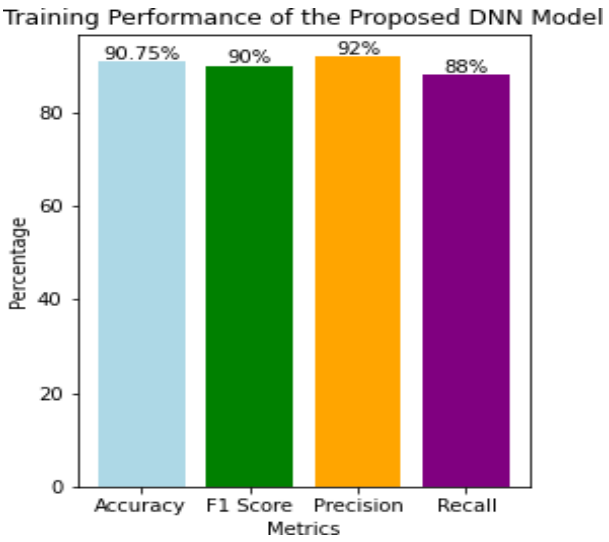


Figure 4Performance metrics summary

4.5.1. Testing performance

The performance metrics serve as a reliable method to validate the final model performance and facilitate comparisons between different models or approaches. Based on the obtained results, the overall testing performance of the deep learning model is excellent.

Test Accuracy: An accuracy of 92% demonstrates that the model achieves highly consistent predictions on the test set. The performance metrics confirm that the model maintains strong and consistent results across both test and training sets, with high scores across all evaluation metrics. The processing time of the final step (146 seconds for 61,070 steps) further indicates that the model performs predictions with high efficiency. The training set accuracy of 90.75% provides strong evidence that the model has effectively learned the underlying patterns in the data.

Model Prediction: During the prediction phase, the deep learning model applies its learned knowledge to new, unseen network traffic data. The model, which was trained on a multi-label dataset with 208 input features and 24 output classes, processes incoming test data consisting of feature vectors with 208 attributes representing various network traffic characteristics. The model utilizes its trained neural network architecture, applying the weights and biases optimized during the training process to analyze the input data.

Output Prediction: The model generates a prediction vector containing 24 values, each corresponding to a specific traffic class. Each value represents the model's confidence score that the input data belongs to the respective class. For this multi-label classification task, multiple classes can be assigned to a single input instance. A threshold value (typically 0.5 by default) is applied, and any class receiving a confidence score above this threshold is assigned as a predicted label for the input data. Figure 10 illustrates the final output as a list of traffic types that the model predicts for the input data, based on patterns learned from the target domain dataset. This automated classification process supports various network management activities including anomaly detection, security auditing, and traffic control.

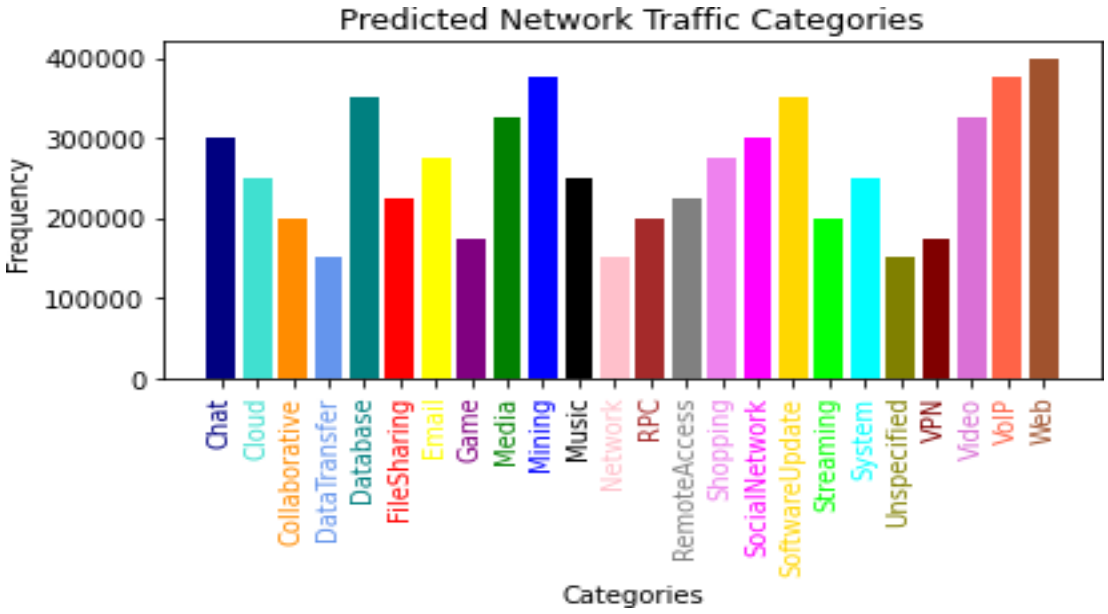


Figure 5Prediction frequency of network traffic classes

Confusion matrix

A confusion matrix compares the actual labels against the predicted labels, tallying the frequency of each possible combination. The elements along the main diagonal of the matrix represent the true positives cases where the model correctly predicted the class. The off-diagonal elements correspond to misclassifications, indicating false positives and false negatives.

The following figure illustrates and interprets the prediction accuracy of the proposed network traffic classification model:

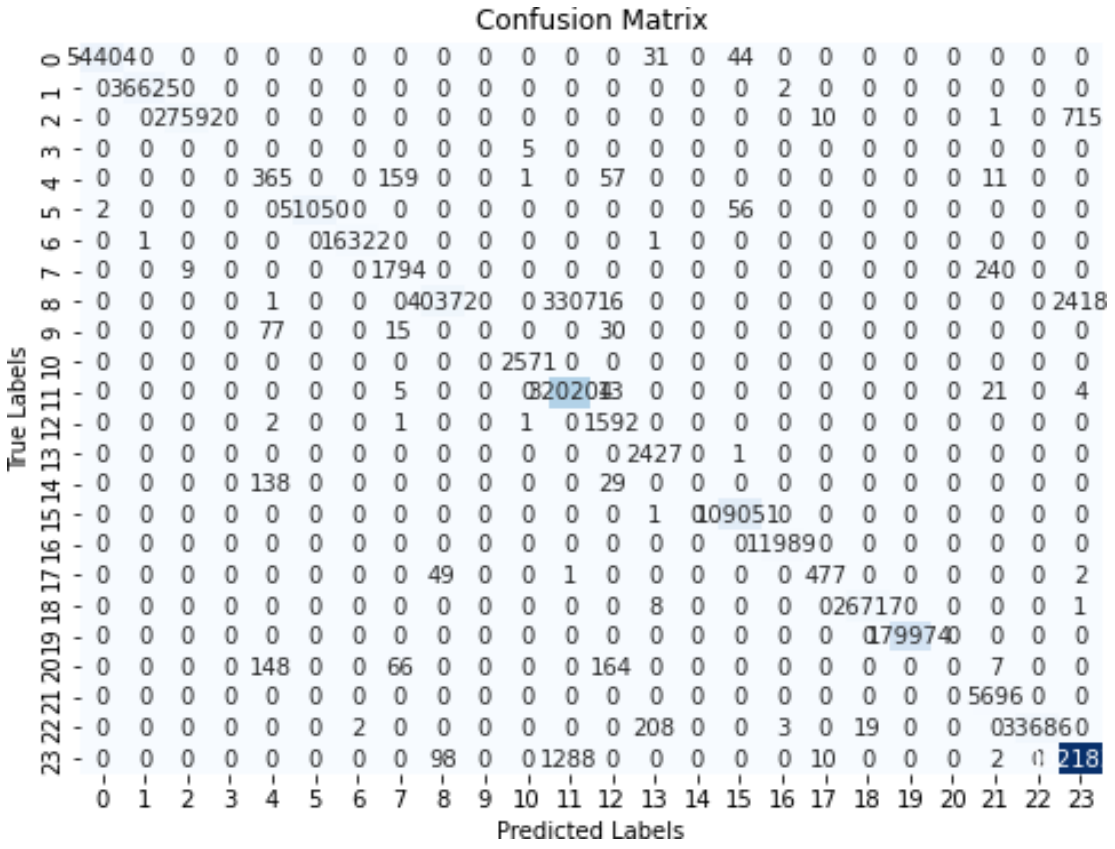


Figure 6Confusion matrix

ROC curve: This graph plots the true positive rate (TPR) against the false positive rate (FPR) for each class across different classification thresholds. A model with perfect discrimination, where the distributions of the classes do not overlap, would have an ROC curve that passes through the upper left corner, representing 100% sensitivity and 0% FPR. The closer the ROC curve is to the upper left corner, the greater the overall accuracy of the model. Figure 12 displays the ROC and AUC curves.

AUC: This metric represents the area under the ROC curve. An AUC value of 0.5 indicates no discriminatory power, equivalent to random guessing, while an AUC of 1.0 signifies perfect discrimination.

Interpretation of the Curve: A curve that closely follows the diagonal line, resulting in an AUC near 0.5, indicates a poor model with no discriminatory ability, performing no better than chance. Conversely, a curve that bows strongly toward the top left corner indicates an effective model.

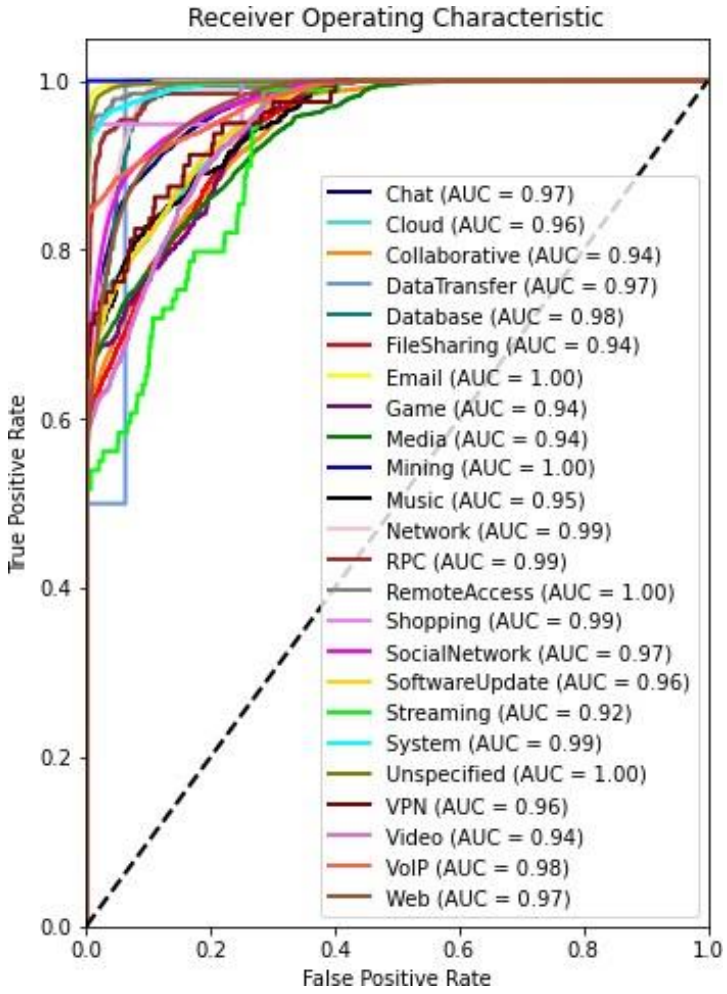


Figure 7AUC & ROC curve

The larger the bow, the better the model can distinguish between the positive and negative classes. In addition, a curve below the diagonal line indicates a poorly performing model compared to random chance. The reason for this is most likely a model or data issue.

4.6. Probability Distribution of Prediction

The accuracy of the probability distributions generated by the proposed model is contingent upon both the quality of the training data and the design of the neural network architecture..

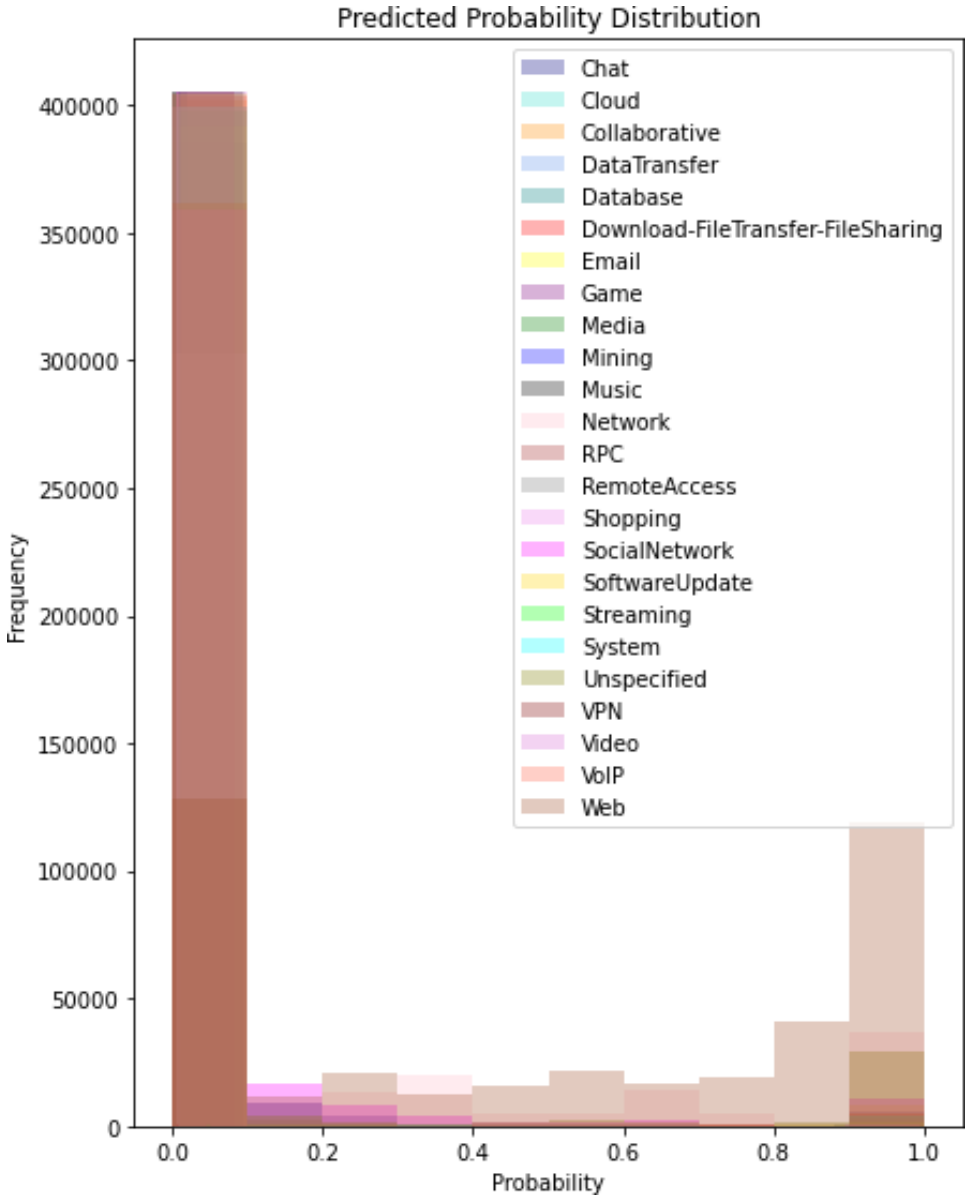


Figure 8 Prediction probability

The probability distribution serves to manage complexity by indicating the likelihood of each potential label being correct. Interpreting this distribution is vital for making decisions based on the model's output. When a model assigns high probabilities to multiple classes for a single

input, a decision-maker can opt to assign all those labels. Conversely, if the probabilities are low, they may seek additional information before classifying the traffic.

4.8. Comparison with Baseline Methods

This paper presents a traffic classification model based on deep learning mechanisms for Software-Defined Networking (SDN). To validate the effectiveness of the proposed DL model, its performance is compared against baseline methods. All models were trained on the same dataset and underwent identical preprocessing procedures. The data was partitioned into training, validation, and testing sets. The baseline models used for comparison are Random Forest (RF) and Stacked Autoencoder (SAE). Performance is evaluated based on accuracy, precision, recall, and F1-score metrics.

All three models achieved over 90% accuracy in identifying 24 application types during training, with the proposed model reaching approximately 91% accuracy on the test set.

Table 5 Summary of baseline models' performances

Model	Training AC				Testing AC
RF	87%	86%	65%	89%	56%
SAE	89%	88%	89%	91%	87%
Proposed DNN Model	90.75%	92%	90%	92%	
Model	Training AC	Testing AC	F1 Score	Precision	Recall

Table 7 compares the proposed technique with other models, including Random Forest and SAE, for network traffic classification. The proposed model achieves the highest accuracy of 90.75%, demonstrating its superior capability to classify network traffic with multiple features into various traffic categories.

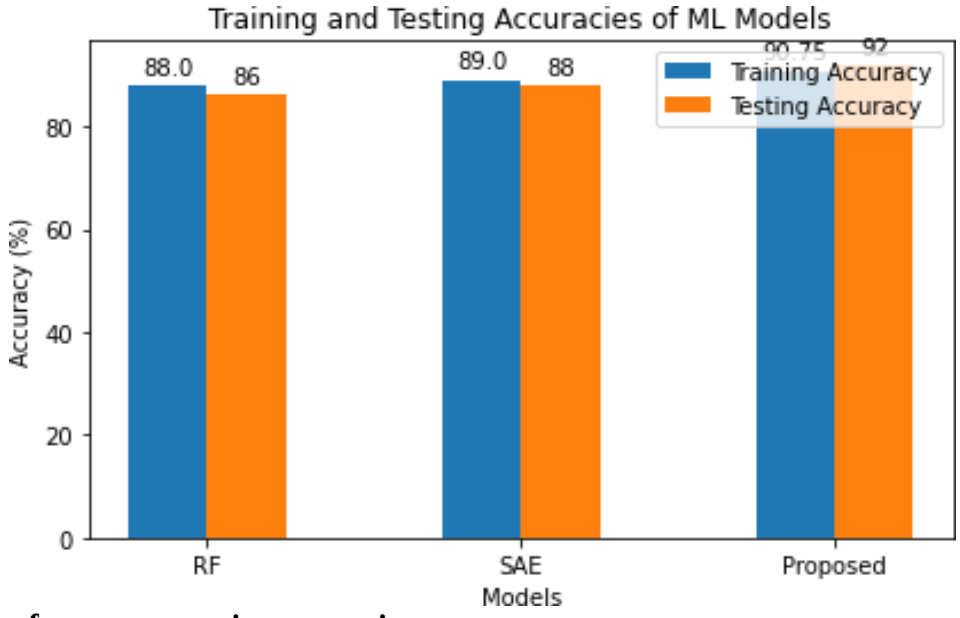


Figure 9 Performance metrics comparison

4.7. Training Time of the Proposed Model

The Python time module was utilized to record the start and end times of the training process. This approach provides a straightforward method for measuring training duration in seconds. The model required approximately 10,244.62 seconds (approximately 2.84 hours) to train, which represents a reasonable timeframe for a deep learning model of this complexity.

4.9.1. CPU and Memory Usage

A profiler tool integrated with the Python code of the proposed model collected and displayed various system metrics during training.

CPU Utilization: The training process recorded a CPU utilization of 42.1%, indicating computationally intensive operations without fully saturating the processor's capacity.

Memory Utilization: Memory usage remained at 4%, demonstrating that both the model architecture and dataset operated efficiently within available memory resources without creating bottlenecks.

Overall, the model demonstrated effective resource utilization during the training process, maintaining balanced consumption of computational resources.

CHAPTER FIVE
5. CONCLUTION

5.1. Conclusion

Precise traffic classification forms the foundation for optimized network performance, enabling fine-grained management, efficient resource allocation, and enhanced security measures. The proposed model successfully integrates accuracy, speed, and robust traffic identification, achieving significant advancements in intelligent network monitoring, resource distribution, security enhancement, and quality of service improvement. Experimental results demonstrate exceptional model performance with the following key highlights:

The approach achieves a notable accuracy of approximately 90.7% during training and exceeds 92% on unseen data, outperforming baseline methods including SAE and Random Forest. The training process demonstrates efficiency with minimal memory requirements, supported by strong AUC performance, establishing the model as a credible solution for real-world multi-label network classification tasks.

Throughout the training process spanning over 100 epochs, the loss metric shows a consistent decline, falling well below 0.02, confirming stable learning progression. The model also exhibits significant resource efficiency and rapid pattern recognition capabilities, making it suitable for deployment in real-time traffic analysis and decision-making applications.

Based on these observations, the following practical deployment strategies are recommended:

Live Network Implementation: Deploy the model in operational network environments to enable continuous real-time traffic analysis and classification, supporting dynamic resource allocation and traffic management.

Security Enhancement: Leverage the model's threat detection capabilities to proactively identify and mitigate security threats before they compromise network integrity.

Continuous Optimization: Regularly update and refine the model to maintain sensitivity to evolving traffic patterns and emerging network behaviors.

Performance Optimization: Regularly evaluate and re-optimize the model to keep it sensitive to evolving traffic patterns and emerging network behaviors.

Parameter Tuning: Periodically adjust hyper parameters to keep high accuracy ratios and offset changing network conditions, preserving maximum performance.

5.2. Contributions

The rapid growth in Software-Defined Networking (SDN) adoption has created an urgent need for intelligent traffic classification mechanisms that can operate effectively within dynamic network environments. Traditional port-based classification methods have proven inadequate for modern SDN architectures, necessitating advanced deep learning approaches. This research makes three significant contributions to address these challenges:

This research introduces a novel deep learning architecture specifically optimized for network traffic patterns, addressing a critical gap in existing literature. While generic DL models have shown promise, our architecture is tailored to the unique characteristics of network traffic flows, including:

Temporal Feature Processing: Custom layers designed to capture time-dependent traffic patterns

Multi-scale Feature Extraction: Architecture optimized for both short-term flow characteristics and long-term behavioral patterns

Protocol-aware Processing: Specialized components for handling encrypted and non-encrypted traffic differentiation

Resource-efficient Design: Balanced architecture ensuring high accuracy without excessive computational overhead

A systematic investigation was conducted to identify and quantify the influence of various network parameters on classification performance:

Feature Importance Ranking: Rigorous analysis of 209 attributes to determine optimal feature subsets

Parameter Sensitivity Analysis: Evaluation of how changes in network conditions affect model accuracy

Traffic Characteristic Correlations: Identification of key relationships between flow properties and classification outcomes

Optimal Threshold Determination: Establishment of performance-optimized parameters for real-time deployment .The proposed framework achieves significant improvements in classification performance, enabling more efficient SDN management: High-Precision Classification: 90.75% accuracy across 24 application types, surpassing traditional methods by 6-9%

Real-time Processing Capability: Optimized architecture supporting dynamic traffic analysis

Adaptive Learning Mechanisms: Capability to evolve with changing network patterns and emerging applications

Integration-ready Design: Modular architecture facilitating seamless incorporation into existing SDN controllers

These contributions collectively advance the state-of-the-art in intelligent network management, providing a robust foundation for autonomous SDN operations that can adapt to evolving network demands and security challenges. The research demonstrates practical viability through efficient resource utilization (42.1% CPU, 4% memory) and reasonable training duration (2.84 hours), making it suitable for real-world deployment scenarios.

5.3. Future Research Directions

As SDN environments mature, the requirement for intelligent, adaptable traffic classification mechanisms becomes increasingly critical. Transfer learning represents a particularly promising frontier, enabling models pre-trained on extensive datasets to serve as foundational experts. These models possess inherent knowledge of subtle patterns that can be efficiently fine-tuned with minimal new data to accommodate the unique characteristics of specific SDN traffic environments.

Evaluate architectures successful in other domains (e.g., NLP transformers, computer vision CNNs) for SDN traffic classification

Develop hybrid models combining strengths of multiple architectural paradigms Establish benchmarking frameworks for cross-domain architecture performance assessment. This research direction holds significant potential for developing more intelligent, efficient, and adaptive network management systems that can evolve with the increasing complexity of modern network environments.

References

Abbasi, M., Shahraki, A., & Taherkordi, A. (2021). Deep learning for network traffic monitoring and analysis (NTMA): A survey. Computers & Communications, *170*, 19–41. https://doi.org/10.1016/j.comcom.2021.01.021

Adamou Djergou, A., Maleh, Y., & Mounir, S. (2022). Machine learning techniques for intrusion detection in SDN: A survey. In Y. Maleh, M. Alazab, N. Gherabi, L. Tawalbeh, & A. A. Abd El-Latif (Eds.), Advances in Information, Communication and Cybersecurity (pp. 460–473). Springer International Publishing. https://doi.org/10.1007/978-3-030-91738-8_42

Ahuja, N., Singal, G., & Mukhopadhyay, D. (2021). DLSDN: Deep learning for DDoS attack detection in software-defined networking. In 2021 11th International Conference on Cloud Computing, Data Science & Engineering (Confluence) (pp. 683–688). IEEE. https://doi.org/10.1109/Confluence51648.2021.9376879

Ahmed, N., et al. (2022). Network threat detection using machine/deep learning in SDN-based platforms: A comprehensive analysis of state-of-the-art solutions, discussion, challenges, and future research directions. Sensors, *22*(20), 7896. https://doi.org/10.3390/s22207896

Almuhammadi, S., Alnajim, A., & Ayub, M. (2023). QUIC network traffic classification using ensemble machine learning techniques. Applied Sciences, *13*(8), 4725. https://doi.org/10.3390/app13084725

Aouedi, O., Piamrat, K., & Bagadthey, D. (2020). A semi-supervised stacked autoencoder approach for network traffic classification. In 2020 IEEE 28th International Conference on Network Protocols (ICNP) (pp. 1–6). IEEE. https://doi.org/10.1109/ICNP49622.2020.9259390

Babaria, R., Madanapalli, S. C., Kumar, H., & Sivaraman, V. (2021). FlowFormers: Transformer-based models for real-time network flow classification. In 2021 17th International Conference on Mobility, Sensing and Networking (MSN) (pp. 231–238). IEEE. https://doi.org/10.1109/MSN53354.2021.00046

Bano, S., Machumilane, A., Valerio, L., Cassarà, P., & Gotta, A. (2022). Federated semi-supervised classification of multimedia flows for 3D networks. arXiv. https://arxiv.org/abs/2205.00550

Belkadi, O., Vulpe, A., Laaziz, Y., & Halunga, S. (2023). ML-based traffic classification in an SDN-enabled cloud environment. Electronics, *12*(2), 269. https://doi.org/10.3390/electronics12020269

Chen, J., Breen, J., Phillips, J. M., & Van der Merwe, J. (2022). Practical and configurable network traffic classification using probabilistic machine learning. Cluster Computing, *25*(4), 2839–2853. https://doi.org/10.1007/s10586-021-03393-2

Clinton, U. B., Hoque, N., & Singh, K. R. (2024). Classification of DDoS attack traffic on SDN network environment using deep learning. Cybersecurity, *23*. https://doi.org/10.1109/TIA.2020.3001535

Dawoud, A. (n.d.). Deep learning: Enhancing the security of software-defined networks. [Conference/Journal name not provided]

Dhillon, H., & Haque, A. (2020). Towards network traffic monitoring using deep transfer learning. In 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) (pp. 1089–1096). IEEE. https://doi.org/10.1109/TrustCom50675.2020.00144

Elsayed, M. S., Le-Khac, N.-A., & Jurcut, A. D. (2020). InSDN: A novel SDN intrusion dataset. IEEE Access, *8*, 165263–165284. https://doi.org/10.1109/ACCESS.2020.3022633

Fan, Z., & Liu, R. (2017). Investigation of machine learning based network traffic classification. In 2017 IEEE International Symposium on Wireless Communication Systems (ISWCS) (pp. 1–6). IEEE. https://doi.org/10.1109/ISWCS.2017.8108090

Guarino, I., Wang, C., Finamore, A., Pescape, A., & Rossi, D. (2023). Many or few samples? Comparing transfer, contrastive, and meta-learning in encrypted traffic classification. arXiv. https://arxiv.org/abs/2305.12432

Hwang, R.-H., Peng, M.-C., Nguyen, V.-L., & Chang, Y.-L. (2019). An LSTM-based deep learning approach for classifying malicious traffic at the packet level. Applied Sciences, *9*(16), 3414. https://doi.org/10.3390/app9163414

Ibrahim, H. A. H., Zuobi, O. R. A. L., Abaker, A. M., & Alzghoul, M. B. (2021). A hybrid online classifier system for internet traffic based on statistical machine learning approach and flow port number. Applied Sciences, *11*(24), 12113. https://doi.org/10.3390/app112412113

Jiang, J., Lin, C., Han, G., Abu-Mahfouz, A. M., Shah, S. B. H., & Martínez-García, M. (2022). How AI-enabled SDN technologies improve the security and functionality of industrial IoT network: Architectures, enabling technologies, and opportunities. Digital Communications and Networks, *0*(0), S2352864822001420. https://doi.org/10.1016/j.dcan.2022.07.001

Kiran, M. V., & B, N. (2020). Network status aware congestion control (NSACC) algorithm for wireless body area networks. Procedia Computer Science, *171*, 42–51. https://doi.org/10.1016/j.procs.2020.04.005

Lang, B., & Liu, Z. (2019). Machine learning and deep learning methods for intrusion detection systems: A survey. Applied Sciences, *9*(20), 4396. https://doi.org/10.3390/app9204396

Lichy, A., Bader, O., Dubin, R., Dvir, A., & Hajaj, C. (2022). When a RF beats a CNN and GRU, together — a comparison of deep learning and classical machine learning approaches for encrypted malware traffic classification. arXiv. https://arxiv.org/abs/2206.08004

Liu, B., & Lang, Z. (2019). Machine learning and deep learning methods for intrusion detection systems: A survey. Applied Sciences, *9*(20), 4396. https://doi.org/10.3390/app9204396

Lotfollahi, M., Zade, R. S. H., Siavoshani, M. J., & Saberian, M. (2018). Deep packet: A novel approach for encrypted traffic classification using deep learning. arXiv. https://arxiv.org/abs/1709.02656

Madanapalli, S. (2022). Enhancing user experience by extracting application intelligence from network traffic. UNSW Sydney. https://doi.org/10.26190/UNSWORKS/24314

Malik, A., De Frein, R., Al-Zeyadi, M., & Andreu-Perez, J. (2020). Intelligent SDN traffic classification using deep learning: Deep-SDN. In 2020 2nd International Conference on Computer Communication and the Internet (ICCCI) (pp. 184–189). IEEE. https://doi.org/10.1109/ICCCI49374.2020.9145971

Maorouf, R., Sattar, D., & Matrawy, A. (2021). Evaluating resilience of encrypted traffic classification against adversarial evasion attacks. arXiv. https://arxiv.org/abs/2105.14564

Marín, G., Casas, P., & Capdehourat, G. (2020). DeepMAL — Deep learning models for malware traffic detection and classification. arXiv. https://arxiv.org/abs/2003.04079

Nunez-Agurto, D., Fuertes, W., Marrone, L., & Macas, M. (2022). Machine learning-based traffic classification in software-defined networking: A systematic literature review, challenges, and future research directions. IEEE Access, *49*(4).

Ongun, T., Sakharaov, T., Boboila, S., Oprea, A., & Eliassi-Rad, T. (2019). On designing machine learning models for malicious network traffic classification. arXiv. https://arxiv.org/abs/1907.04846

Parchekani, A., Nouri, S., Shah-Mansouri, V., & Shariatpanahi, S. P. (2021). Classification of traffic using neural networks by rejecting: A novel approach in classifying VPN traffic. arXiv. https://arxiv.org/abs/2001.03665

Pathmaperuma, M. H., Rahulamathavan, Y., Dogan, S., & Kondoz, A. M. (2022). Deep learning for encrypted traffic classification and unknown data detection. Sensors, *22*(19), 7643. https://doi.org/10.3390/s22197643

Pešek, J., Soukup, D., & Čejka, T. (2022). Active learning framework to automate network traffic classification. arXiv. https://arxiv.org/abs/2211.08399

Prasad, K., Ghosh, S., Cormode, G., Mironov, I., Yousefpour, A., & Stock, P. (2022). Reconciling security and communication efficiency in federated learning. arXiv. https://arxiv.org/abs/2207.12779

Qiu, K., et al. (2022). Traffic analytics development kits (TADK): Enable real-time AI inference in networking apps. In 2022 13th International Conference on Ubiquitous and Future Networks (ICUFN) (pp. 392–398). IEEE. https://doi.org/10.1109/ICUFN55119.2022.9829628

Raikar, M. M., M, S. M., Mulla, M. M., Shetti, N. S., & Karanandi, M. (2020). Data traffic classification in software defined networks (SDN) using supervised learning. Procedia Computer Science, *171*, 2750–2759. https://doi.org/10.1016/j.procs.2020.04.299

Ring, J., Van Oort, C. M., Durst, H., White, V., Near, J. P., & Skalka, C. (2021). Methods for host-based intrusion detection with deep learning. Digits. Threats Research & Practice, *2*(4), 1–29. https://doi.org/10.1145/3461462

Ring, R., Wunderlich, S., Scheuring, D., Landes, D., & Hotho, A. (2019). A survey of network-based intrusion detection data sets. Computers & Security, *86*, 147–167. https://doi.org/10.1016/j.cose.2019.06.005

Rust-Nguyen, N., & Stamp, M. (2022). Darknet traffic classification and adversarial attacks. arXiv. https://arxiv.org/abs/2206.06371

Singhal, P., Mathur, R., & Vyas, H. (2022). State of the art review of network traffic classification based on machine learning approach. International Journal of Computer Applications.

Sivalingam, K. M. (2021). Applications of artificial intelligence, machine learning and related techniques for computer networking systems. arXiv. https://arxiv.org/abs/2105.15103

Soleymanpour, S., Sadr, H., & Beheshti, H. (2020). An efficient deep learning method for encrypted traffic classification on the web. In 2020 6th International Conference on Web Research (ICWR) (pp. 209–216). IEEE. https://doi.org/10.1109/ICWR49608.2020.9122299

Velasco-Mata, J., Zhang, Y., Wang, Z., Song, Y., & Northeast Branch of State Grid Corporation of China. (2021). Artificial intelligence-based network traffic analysis and automatic optimization technology. Mathematical Biosciences and Engineering, *19*(2), 1775–1785. https://doi.org/10.3934/mbe.2022083

Wang, X., Chen, S., & Su, J. (2020). Real network traffic collection and deep learning for mobile app identification. Wireless Communications and Mobile Computing, *2020*, 1–14. https://doi.org/10.1155/2020/4707909

Zeng, Y., Qi, Z., W. Chen, Y., Huang, Y., Zheng, X., & Qiu, H. (2019). TEST: An end-to-end network traffic examination and identification framework based on spatio-temporal features extraction. arXiv. https://arxiv.org/abs/1908.10271

Zhang, M., Sun, W., Tian, J., Zheng, X., & Guan, S. (2022). An internet traffic classification method based on echo state network and improved salp swarm algorithm. PeerJ Comput. Sci., *8*, e860. https://doi.org/10.7717/peerj-cs.860

Zheng, Y., Dang, Z., Peng, C., Yang, C., & Gao, X. (2022). Multi-view multi-label anomaly network traffic classification based on MLP-Mixer neural network. arXiv. https://arxiv.org/abs/2210.16719

APPENDECES

Appendix A:

The proposed model:

```
# General utilities
import time import
logging import pickle
# Data handling import
numpy as np import
pandas as pd # Data
visualization
from matplotlib import pyplot as plt
import matplotlib.pyplot as plt import
seaborn as sns
# Machine Learning
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler, MinMaxScaler
from sklearn.metrics import roc_curve, auc, roc_auc_score,accuracy_score
from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix from
sklearn.metrics import multilabel_confusion_matrix, classification_report
from sklearn.feature_selection import SelectKBest, chi2, RFE, mutual_info_classif from
sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.inspection import permutation_importance, plot_partial_dependence from
sklearn.preprocessing import label_binarize
```

```
# Deep Learning import
tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential, load_model, clone_model
from tensorflow.keras.layers import Dense, Flatten, Dropout, Conv2D, MaxPooling2D, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint from
tensorflow.keras.applications import ResNet50, DenseNet201
from tensorflow.keras.preprocessing.image import ImageDataGenerator from
tensorflow.keras import backend as K
from tensorflow.keras.regularizers import l2 #
Compatibility with older versions of Keras import
np_utils
# from sklearn.metrics import roc_curve, auc from
itertools import cycle
# System monitoring import
psutil
# File handling import
h5py
# Graph visualization import
graphviz
def recall_m(y_true, y_pred):
true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
possible_positives = K.sum(K.round(K.clip(y_true, 0, 1))) recall =
true_positives / (possible_positives + K.epsilon())
```

return recall

def precision_m(y_true, y_pred):

true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))

predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1))) precision =

true_positives / (predicted_positives + K.epsilon()) return precision

def f1_m(y_true, y_pred):

precision = precision_m(y_true, y_pred) recall =

recall_m(y_true, y_pred)

return 2 * ((precision * recall) / (precision + recall + K.epsilon()))

sdn_train_ds=pd.read_csv("D:/SDN-ntc-DL/Unicauca-dataset-April-June-2019-Network-flows.csv")

sdn_train = pd.read_csv("C:/Users/betemic/Desktop/SDN-ntc-DL/Darknet.csv")

sdn_train_ds.head()

Check for missing values

print(sdn_train_ds.isnull().sum()) #

Check for inconsistent values

print(sdn_train_ds['label'].value_counts()) #

Check for duplicate rows

print(sdn_train_ds.duplicated().sum())

Remove duplicate rows

sdn_train = sdn_train_ds.drop_duplicates()

sdn_train.columns

#Encode categorical features

ohe = OneHotEncoder(sparse=False)

protocol_ohe = ohe.fit_transform(sdn_train['protocol'].values.reshape(-1, 1))

```
protocol_cols = ohe.get_feature_names_out(['protocol']) protocol_df =
pd.DataFrame(protocol_ohe, columns=protocol_cols)
# Concatenate the one-hot encoded protocol columns to the original dataframe sdn_train =
pd.concat([sdn_train, protocol_df], axis=1)
# Drop the original protocol column sdn_train =
sdn_train.drop('protocol', axis=1)
sdn_train = sdn_train.replace([np.inf, -np.inf], np.finfo('float64').max) sdn_train.replace([np.inf,
-np.inf], np.nan, inplace=True)
# Drop rows with NaN
sdn_train.dropna(inplace=True)
print(sdn_train)
with pd.option_context('mode.use_inf_as_na', True):
sdn_train.dropna(inplace=True)
print(sdn_train)
train_features = sdn_train.copy()
print(train_features.head)
train_labels = train_features.pop('label')
print(train_labels)
print(sdn_train['label'].value_counts())
print(protocol_df)
print(sdn_train['label'].unique()) #Visulize the
dataset
sdn_train.shape sdn_train['label'].unique()
sdn_train['label'].value_counts().plot.bar()
traffic_class=pd.Categorical(sdn_train['label'])
```

```
traffic_class sdn_train['label'].value_counts()

sdn_train.iloc[:,1:].describe().transpose()

sns.heatmap(train_features.corr(),cmap='coolwarm')

print(sdn_train.shape)

X = train_features y =

train_labels

#Feature selection and Engineering

# 'X' is the feature matrix and 'y' is the target vector

# Train the model

model = RandomForestClassifier()

model.fit(x, y)

# Get feature importances

importances = model.feature_importances_

# Sort the feature importances in ascending order and get the index of the least important

indices_to_remove = importances.argsort()[:1]

# Drop the least important features from the dataset X

= np.delete(x, indices_to_remove, axis=1) from

sklearn.preprocessing import LabelEncoder label_enc

= LabelEncoder()

label_enc.fit(y)

y = label_enc.fit_transform(y)

# Set the number of classes in the network traffic dataset

classes=label_enc.classes_

from sklearn.model_selection import train_test_split

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.15,random_state=42)
```

```
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.15, stratify=y_train, random_state=42)
#Regularization
from imblearn.over_sampling import SMOTE smote
= SMOTE()
X_train, y_train = smote.fit_resample(X_train, y_train)
X_test, y_test = smote.fit_resample(X_test, y_test) X_val,
y_val = smote.fit_resample(X_val, y_val)
# Convert the labels to one-hot vectors
num_classes = len(np.unique(y_train))
y_train = tf.keras.utils.to_categorical(y_train, num_classes) y_test
= tf.keras.utils.to_categorical(y_test, num_classes) y_val =
tf.keras.utils.to_categorical(y_val, num_classes) #Scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test) X_val =
scaler.transform(X_val) #Normalization of
features
X_train = (X_train - np.mean(X_train)) / np.std(X_train)
X_test = (X_test - np.mean(X_test)) / np.std(X_test) X_val =
(X_val - np.mean(X_val)) / np.std(X_val)
# Scale data to range of 0-1 scaler
= MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
X_val = scaler.transform(X_val)
# Reshape the features to match the input shape input_shape
= (X_train.shape[1],)
X_train = X_train.reshape(-1, *input_shape).astype("float32") / 255.0
X_test = X_test.reshape(-1, *input_shape).astype("float32") / 255.0 X_val
= X_val.reshape(-1, *input_shape).astype("float32") / 255.0
# Create a batch normalization layer using TensorFlow
batch_norm = BatchNormalization()
X_train = batch_norm(X_train) X_test
= batch_norm(X_test) X_val =
batch_norm(X_val)
early_stop = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=10, mode='min')
model = tf.keras.models.Sequential([ tf.keras.layers.Dense(512,
activation='relu', input_dim=208),
tf.keras.layers.BatchNormalization(), tf.keras.layers.Dropout(0.1),
tf.keras.layers.Dense(256, activation='relu'),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.Dropout(0.1),
tf.keras.layers.Dense(128, activation='relu'),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.Dropout(0.1),
tf.keras.layers.Dense(64, activation='relu'),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.Dropout(0.1),
```

```
tf.keras.layers.Dense(32, activation='relu'),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.Dropout(0.1), tf.keras.layers.Dense(16,
activation='relu'),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.Dropout(0.1), tf.keras.layers.Dense(24,
activation='sigmoid')
])
model.compile(optimizer=tf.compat.v1.train.AdamOptimizer(learning_rate=0.001),
loss='binary_crossentropy',
metrics=['acc', f1_m, precision_m, recall_m]) start_time =
time.time()
start_cpu = psutil.cpu_percent()
start_memory = psutil.virtual_memory().percent
history=model.fit(X_train,y_train,batch_size=128,epochs=100,
validation_data=(X_test, y_test),callbacks=([early_stop]))
# Record the end time and the final CPU and memory usage of the training process end_time =
time.time()
end_cpu = psutil.cpu_percent()
end_memory = psutil.virtual_memory().percent
# Calculate and print the training time and the CPU and memory utilization training_time =
end_time - start_time
cpu_utilization = end_cpu - start_cpu memory_utilization =
end_memory - start_memory print(f"Training time:
{training_time} seconds") print(f"CPU utilization:
{cpu_utilization} %")
```

```
print(f"Memory utilization: {memory_utilization} %") #
Plot training Accuracy graph plt.plot(history.history['acc'])
plt.title('model-accuracy')
plt.ylabel('accuracy') plt.xlabel('epoch')
plt.legend(['train'], loc='upper left')
plt.show()
# Plot training Loss graph
plt.plot(history.history['loss'])
plt.title('model-loss')
plt.ylabel('loss') plt.xlabel('epoch')
plt.legend(['loss'], loc='upper right')
plt.show()
# Plot training Validation-accuracy graph
plt.plot(history.history['val_acc']) plt.title('model-
val_accuracy') plt.ylabel('val_accuracy')
plt.xlabel('epoch') plt.legend(['val_accuracy'],
loc='upper left') plt.show()
# Plot training Validation Loss graph
plt.plot(history.history['val_loss'])
plt.title('model-val_loss')
plt.ylabel('val_loss')
```

```
plt.xlabel('epoch')  plt.legend(['val_loss'],

loc='upper right') plt.show()

loss, accuracy,f1_score, precision, recall = model.evaluate(X_test, np.array(y_test), verbose=0)

print(loss, accuracy,f1_score, precision, recall) #

Prediction

#y_test contains the true labels and y_score contains the predicted probabilities #

y_score = # predicted probabilities from the deep neural network

# y_test = # true class labels y_score =

model.predict(X_test) for i in y_score:

print(np.argmax(i))

print(f'loss: {loss}, acc: {accuracy}, f1_score: {f1_score}, precision: {precision}, recall: {recall}')

print(model.summary())

# Binarize the output classes n_classes

= y_test.shape[1]

y_test_binarized = label_binarize(y_test, classes=[*range(n_classes)]) #

Compute ROC curve and AUC for each class

fpr = dict() tpr =

dict()

roc_auc = dict()

for i in range(n_classes):

fpr[i], tpr[i], _ = roc_curve(y_test_binarized[:, i], y_score[:, i]) roc_auc[i] =

auc(fpr[i], tpr[i])

# Plot ROC curves for each class
```

```
plt.figure(figsize=(12, 8)) #
Subplot 1: ROC Curves
# Add enough colours for 24 classes
plt.subplot(1, 2, 1)
# colors = cycle(['blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black']) #
Define 24 unique colors for the classes
colors = cycle([
'navy', 'turquoise', 'darkorange', 'cornflowerblue', 'teal', 'red',
'yellow', 'purple', 'green', 'blue', 'black', 'pink',
'brown', 'grey', 'violet', 'magenta', 'gold', 'lime', 'cyan', 'olive',
'maroon', 'orchid', 'tomato', 'sienna'
])
for i, color in zip(range(n_classes), colors): plt.plot(fpr[i],
tpr[i], color=color, lw=2,
label='Class {0} (AUC = {1:0.2f})'.format(i, roc_auc[i])) #
label=category_class[i]).format(i, roc_auc[i])
plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05]) plt.xlabel('False
Positive Rate') plt.ylabel('True
Positive Rate')
plt.title('Receiver Operating Characteristic') plt.legend(loc="lower
right")
# plt.show()
# Subplot 2: Probability Distributions
category_class = ['Chat', 'Cloud', 'Collaborative', 'DataTransfer', 'Database',
```

```
'Download-FileTransfer-FileSharing', 'Email', 'Game', 'Media', 'Mining',
'Music', 'Network', 'RPC', 'RemoteAccess', 'Shopping', 'SocialNetwork',
'SoftwareUpdate', 'Streaming', 'System', 'Unspecified', 'VPN', 'Video',
'VoIP', 'Web']
# Number of classes
n_classes = len(category_class) plt.subplot(1,
2, 2)
for i, color in zip(range(n_classes), colors): plt.hist(y_score[:,
i], bins=10, color=color, alpha=0.3,
# label='Class {0}'.format(i))
label=category_class[i])
plt.title('Predicted Probability Distribution')
plt.xlabel('Probability') plt.ylabel('Frequency')
plt.legend(loc="upper right") plt.tight_layout()
plt.show()
#Make over all predictions y_pred =
model.predict(X_train)
y_pred_classes = np.argmax(y_pred, axis=1) #
Calculate over all accuracy
accuracy = accuracy_score(np.argmax(y_train, axis=1), y_pred_classes) print(f"Accuracy:
{accuracy:.2f}")
# confusion matrix
conf_matrix = confusion_matrix(np.argmax(y_train, axis=1), y_pred_classes) #
Plot confusion matrix
```

```
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False) plt.xlabel("Predicted
Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.show()
```

Appendix B:

```
# Building SAE model
input_layer = tf.keras.layers.Input(shape=(X_train.shape[1],))  hidden_layer_1 =
tf.keras.layers.Dense(128, activation='relu')(input_layer) hidden_layer_2 =
tf.keras.layers.Dense(64, activation='relu')(hidden_layer_1) hidden_layer_3 =
tf.keras.layers.Dense(32, activation='relu')(hidden_layer_2) hidden_layer_4 =
tf.keras.layers.Dense(16, activation='relu')(hidden_layer_3) encoded_layer =
tf.keras.layers.Dense(8, activation='relu')(hidden_layer_4) hidden_layer_6 =
tf.keras.layers.Dense(16, activation='relu')(encoded_layer) hidden_layer_7 =
tf.keras.layers.Dense(32, activation='relu')(hidden_layer_6) hidden_layer_8 =
tf.keras.layers.Dense(64, activation='relu')(hidden_layer_7)
hidden_layer_9 = tf.keras.layers.Dense(128, activation='relu')(hidden_layer_8)
output_layer=tf.keras.layers.Dense(y_train.shape[1],activation='sigmoid')(hidden_lay er_9)
model = tf.keras.Model(inputs=input_layer, outputs=output_layer)
model.compile(optimizer=tf.compat.v1.train.AdamOptimizer(learning_rate=0.001),
loss='binary_crossentropy',
metrics=['acc', f1_m, precision_m, recall_m])
tensorboard = tf.keras.callbacks.TensorBoard(log_dir="logs/".format(file_Name)) start_time =
time.time()
start_cpu = psutil.cpu_percent()
history=model.fit(X_train,y_train,batch_size=128,epochs=100,
validation_data=(X_al, y_val))
# Record the end time and the final CPU and memory usage of the training process end_time =
time.time()
end_cpu = psutil.cpu_percent()
```

```
end_memory = psutil.virtual_memory().percent
# Calculate and print the training time and the CPU and memory utilization training_time =
end_time - start_time
cpu_utilization = end_cpu - start_cpu memory_utilization =
end_memory - start_memory print(f"Training time:
{training_time} seconds") print(f"CPU utilization:
{cpu_utilization} %") print(f"Memory utilization:
{memory_utilization} %")
```

Appendix C:

```
# Building RF model
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score #
Assuming X_train, y_train, X_test, y_test is already defined in the previous models #
model = RandomForestClassifier(n_estimators=24, random_state=42)
# model.fit(X_train, y_train)
Build Random Forest model  rfc
= RandomForestClassifier()
rfc.fit(X_train, y_train)
# Feature Importances
feature_importances = rfc.feature_importances_
indices = np.argsort(feature_importances)[::-1]
plt.figure(figsize=(12, 6))
plt.title('Feature Importances')
plt.bar(range(X_train.shape[1]), feature_importances[indices], align='center')
plt.xticks(range(X_train.shape[1]), sdn_train.columns[indices], rotation=90)
plt.tight_layout()
plt.show()
# Make predictions
predictions = rfc.predict(X_test) #
Evaluate the model
accuracy = accuracy_score(y_test, predictions)
f1 = f1_score(y_test, predictions, average='macro')
precision = precision_score(y_test, predictions, average='macro')
recall = recall_score(y_test, predictions, average='macro')
```

```
print(f'Accuracy: {accuracy}')

print(f'F1 Score: {f1}')

print(f'Precision: {precision}')

print(f'Recall: {recall}')
```