COMPARATIVE BENCHMARKING OF METAHEURISTIC OPTIMIZERS FOR GAN-AUGMENTED CNN–LSTM-BASED IoT INTRUSION DETECTION

A Thesis Submitted
in Partial Fulfilment of the Requirements for the
Degree of

MASTER OF TECHNOLOGY
in
INFORMATION TECHNOLOGY

by

UDIT GUPTA
(Roll No. 24/ISY/01)

Under the Supervision of

Dr. VIRENDER RANGA
Associate Professor, Department of Information Technology



Department of Information Technology
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi–110042, India

May, 2026

DEPARTMENT OF INFORMATION TECHNOLOGY

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

ACKNOWLEDGEMENT

I owe a profound debt of gratitude to my supervisor, **Dr. VIRENDER RANGA**, Associate Professor, Department of Information Technology, Delhi Technological University, for his meticulous guidance, intellectual insights, and consistent encouragement throughout this research. I also sincerely thank the faculty and staff of the Department of Information Technology for providing a supportive environment and essential computational resources.

I extend my appreciation to the Canadian Institute for Cybersecurity for making the CICIoT2023 dataset accessible, and to my fellow research scholars at Delhi Technological University for their stimulating discussions and support. Finally, I thank my family for their unfailing belief and patience throughout my studies.

Place: Delhi

Date: UDIT GUPTA



DEPARTMENT OF INFORMATION TECHNOLOGY

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

I **UDIT GUPTA** hereby certify that the work which is being presented in the thesis entitled **COMPARATIVE BENCHMARKING OF METAHEURISTIC OPTIMIZERS FOR GAN-AUGMENTED CNN–LSTM-BASED IoT INTRUSION DETECTION** in partial fulfillment of the requirements for the award of the Degree of Master of Technology, submitted in the Department of Information Technology, Delhi Technological University is an authentic record of my own work carried out during the period from **2024** to **2026** under the supervision of **Dr. VIRENDER RANGA**.

The matter presented in the thesis has not been submitted by me for the award of any other degree of this or any other Institute.

Candidate's Signature

This is to certify that the student has incorporated all the corrections suggested by the examiners in the thesis and the statement made by the candidate is correct to the best of our knowledge.

Signature of Supervisor (s) **Signature of External Examiner**



DEPARTMENT OF INFORMATION TECHNOLOGY

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

CERTIFICATE BY THE SUPERVISOR

Certified that **UDIT GUPTA (24/ISY/01)** has carried out the research work presented in this thesis entitled **"COMPARATIVE BENCHMARKING OF METAHEURISTIC OPTIMIZERS FOR GAN-AUGMENTED CNN–LSTM-BASED IoT INTRUSION DETECTION"** for the award of **Master of Technology** from Department of Information Technology, Delhi Technological University, Delhi, under my supervision. The thesis embodies results of original work, and studies are carried out by the student himself and the contents of the thesis do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

Signature

(**Dr. VIRENDER RANGA**)

(Associate Professor, Department of Information Technology)

(Delhi Technological University)

Place: Delhi

Date:

ABSTRACT

However, rapid IoT proliferation in vital industries such as health care organizations, industrial automation systems, smart city platforms, and transportation networks has greatly expanded the attack surface open to malicious parties on the Internet. Because of the hardware limitations that typically exist in most IoT devices, precluding the use of classical encryption or signatures as security defenses, there is no escaping the constant threat of intrusions into the IoT network. This thesis explores the problem of designing an efficient IDS for the purpose of accurately detecting intrusions in IoT networks by simultaneously addressing the three interrelated problems of extreme class imbalance,modeling of spatial and temporal traffic, and choice of hyperparameters for the deep learning model.

The research makes use of the CICIoT2023 benchmark dataset, which represents a publically available dataset of traffic data derived from 105 physically unique IoT devices belonging to 33 different attack families, as the main test environment. In order to mitigate the issue related to the clear shortage of samples of minor- ity class attacks, a Conditional Generative Adversarial Network (cGAN) is used to generate statistically valid samples of traffic data related to underrepresented classes, generating roughly 4,962 extra training samples. The main classification component represents a two-stage model combining one-dimensional Convolutional Neural Network (CNN) modules, capturing structured interactions between features within each traffic flow record, with LSTM modules, capturing temporal evolution of those interactions between subsequent time steps. Such approach leads to a model comprising 81,193 trainable parameters and tested using 41- and 9-class tasks respectively.

Three population-based meta-heuristics were assessed for their ability to optimise the hyperparameters of the above neural architecture, each subjected to an identical experimental setup: Siberian Tiger Optimisation (STO), Genetic Algorithm (GA) and Particle Swarm Optimisation (PSO). STO obtains a higher test accuracy of 94.45%, better than

GA (93.61%) and PSO (93.59%). In addition, when subjected to repeated assessments using five independent random splits of the training data set, STO exhibits a higher mean accuracy (93.61% $\pm$ 0.13%), the minimum variance of all three algorithms assessed, while PSO suffers from significantly lower mean accuracy (81.79% $\pm$ 5.26%) in less favourable random splits. The optimised deep learning pipeline outperforms all four classical machine learning baselines (Decision Tree, Random Forest, Support Vector Machine, Logistic Regression) across balanced per-class metrics, while per-class error analysis highlights the intra-class heterogeneity of Web Attack traffic as the primary open challenge, with targeted remediation strategies proposed.

Keywords: IoT Intrusion Detection; GAN Data Augmentation; CNN–LSTM; Siberian Tiger Optimisation; Genetic Algorithm; Particle Swarm Optimisation; CICIoT2023; Hyperparameter Optimisation; Class Imbalance; Deep Learning.

LIST OF PUBLICATIONS

The following peer-reviewed works have resulted from the research presented in this thesis:

[1] Udit Gupta and Virender Ranga, "A GAN-Augmented and STO-Optimized CNN–LSTM Approach for IoT Intrusion Detection," in *Cognitive Systems & Secure Twins: AI Innovations for IoT and Cyber Defense*, accepted for publication in Volume 16437 of Springer / Lecture Notes in Computer Science, 2026.

[2] Udit Gupta and Virender Ranga, "Comparative Benchmarking of Metaheuristic Optimizers for GAN-Augmented CNN–LSTM-Based IoT Intrusion Detection," presented at the 4th International Conference on Computer, Electronics and Electrical Engineering and their Applications (IC2E3-2026), held on 14th–15th May 2026 at National Institute of Technology Uttarakhand, India, and accepted for publication, 2026.

Contents

Acknowledgement i

Candidate's Declaration ii

Certificate iii

Abstract iv

List of Publications vi

List of Figures x

List of Tables xi

List of Symbols xii

List of Abbreviations and Nomenclature xiii

1 Introduction 1

1.1 General Overview 1

1.2 Motivation 3

1.3 Problem Statement 4

1.4 Research Objectives 4

1.5 Contributions to the Research 5

1.6 Organisation of the Thesis 6

2 Literature Review 7

2.1 Securing the Internet of Things 7

2.2 Using Traditional ML Algorithms for Intrusion Detection 8

2.3 Deep Neural Network Approaches for Network Threat Detection 8

2.4 Hybrid CNN–LSTM Architectures for Intrusion Detection 9
2.5 GAN-Based Data Augmentation for Imbalanced Traffic 10
2.6 Metaheuristic Algorithms for Hyperparameter Optimisation 11
2.7 Studies Using the CICIoT2023 Dataset 13
2.8 Research Gaps and Positioning of This Work 13

3 Background Concepts **15**
3.1 Convolutional Neural Networks (CNNs) 15
3.2 Long Short-Term Memory Networks 16
3.3 Generative Adversarial Networks 17
3.4 Siberian Tiger Optimisation 17
3.5 Genetic Algorithm 18
3.6 Particle Swarm Optimisation 18
3.7 Evaluation Metrics 19

4 Dataset and Preprocessing **21**
4.1 CICIoT2023 Dataset Overview 21
4.2 Subset Selection and Train-Test Partitioning 22
4.3 Class Distribution and Imbalance Analysis 22
4.4 Preprocessing Pipeline 23
4.5 GAN-Based Data Augmentation 24

5 Proposed Methodology **25**
5.1 Overall System Architecture 25
5.2 Baseline Machine Learning Models 26
5.3 Hybrid CNN–LSTM Architecture 27
5.4 Metaheuristic Optimisation Framework 28
5.4.1 Genetic Algorithm Configuration 28
5.4.2 Particle Swarm Optimisation Configuration 28
5.4.3 Siberian Tiger Optimisation Configuration 28
5.5 Experimental Setup 29

6 Results and Discussion **30**
6.1 Baseline Machine Learning Performance 30

6.2 CNN–LSTM Training Dynamics 31
6.3 41-Class CNN–LSTM Performance 32
6.4 9-Class STO–CNN–LSTM Performance 33
6.5 Comparative Analysis of Optimisers 35
6.6 Statistical Validation Across Randomised Runs 37
6.7 Convergence Analysis 38
6.8 Computational Efficiency 39
6.9 Error Analysis and Limitations 40

7 Conclusion, Future Scope and Social Impact 42
7.1 Summary and Conclusions 42
7.2 Future Scope 43
7.3 Social Impact 44

A Optimal Hyperparameter Configurations 50

B Conditional GAN Training Configuration 51

C Source Code for Preprocessing, Augmentation and CNN–LSTM Classifier 52

D Source Code for Metaheuristic Hyperparameter Optimization Comparison 95

Proof of Publications 109

Curriculum Vitae 110

List of Figures

5.1 High-level architecture of the proposed GAN-augmented, metaheuristic-optimised CNN–LSTM intrusion detection framework. 26

6.1 Accuracy, Precision, Recall, and F1-Score for the four baseline ML classifiers on the GAN-augmented dataset. 31
6.2 Epoch-wise training dynamics for the CNN–LSTM model (41-class task). 32
6.3 Confusion matrix for the CNN–LSTM model on the 41-class classification task. 33
6.4 Per-class F1-scores for the STO–CNN–LSTM model on the 9-class task. Test accuracy: 94.45%. 34
6.5 Comparison of Test Accuracy Across STO, GA, and PSO for CICIoT2023 (primary run). 35
6.6 Accuracy results for all model configurations. The best detection accuracy is achieved by the STO-optimised CNN–LSTM, being the best configuration overall. 36
6.7 Accuracy of STO, GA, and PSO across five independently randomised dataset splits. STO maintains consistently high, stable performance while PSO shows substantial variability. 38
6.8 Convergence behaviour of GA and PSO across optimisation evaluation steps. 39

List of Tables

4.1 Approximate Class Distribution in the 9-Class Configuration (Prior to GAN Augmentation) 22

5.1 Layer-Wise Configuration of the CNN–LSTM Network 27

6.1 Performance of Baseline ML Classifiers on GAN-Augmented Data . . . 30
6.2 Per-Class Classification Results for STO–CNN–LSTM (9-Class Task) . 34
6.3 Accuracy and Runtime Comparison of Optimisers (Primary Run) 35
6.4 Comprehensive Model Performance Comparison on CICIoT2023 36
6.5 Performance Across Five Randomised Dataset Splits 37
6.6 Total Runtime Comparison of Optimisation Algorithms 39

A.1 Best Hyperparameter Configurations Found by Each Optimiser 50

LIST OF SYMBOLS

Symbol	Description
BLX-α	Blending Crossover (parameter α)
F_1	F1-Score (harmonic mean of precision and recall)
c_t	LSTM cell state at time step t
h_t	LSTM hidden state at time step t
f_t, i_t, o_t	Forget, input, and output gates (LSTM)
v_i^t	Velocity of PSO particle i at iteration t
x_i^t	Position of PSO particle i at iteration t
p_i	Personal best position of particle i (PSO)
g	Global best position discovered by swarm (PSO)
w	Inertia weight (PSO)
c_1, c_2	Cognitive and social acceleration coefficients (PSO)
r_1, r_2	Uniform random numbers in $[0, 1]$ (PSO)
σ	Gaussian mutation standard deviation (GA)
η	Learning rate
$\mathscr{L}$	Loss function
N	Population / swarm size

LIST OF ABBREVIATIONS AND NOMENCLATURE

Abbreviation	Description
IoT	Internet of Things
IDS	Intrusion Detection System
CNN	Convolutional Neural Network
LSTM	Long Short-Term Memory
GAN	Generative Adversarial Network
cGAN	Conditional Generative Adversarial Network
STO	Siberian Tiger Optimisation
GA	Genetic Algorithm
PSO	Particle Swarm Optimisation
DL	Deep Learning
ML	Machine Learning
RNN	Recurrent Neural Network
DDoS	Distributed Denial of Service
DoS	Denial of Service
SVM	Support Vector Machine
RF	Random Forest
DT	Decision Tree
LR	Logistic Regression
ReLU	Rectified Linear Unit
SHAP	SHapley Additive exPlanations
LIME	Local Interpretable Model-agnostic Explanations
SMOTE	Synthetic Minority Oversampling Technique
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative

CHAPTER 1

INTRODUCTION

1.1 General Overview

Among the most consequential technological developments of the current era is the widespread embedding of intelligent sensing, communication, and actuation capabilities into everyday physical objects under the paradigm collectively referred to as the Internet of Things. This expansion impacts a range of different sectors: hospitals have come to depend on interconnected physiological sensors that provide continuous physiological information; manufacturing companies make use of many sensors in order to predict future breakdowns in equipment and improve processes; smart cities control the lighting, energy, and public services through interconnected sensor networks; and consumers' own homes contain more and more internet-connected appliances, security locks, cameras, and temperature controls [1, 2]. If we look at what industry analysts think, the total count of IoT connections will reach or exceed 43 billion by 2030 [3].

While this connectedness is responsible for delivering these abilities, it has resulted in a massive, diverse attack surface. Each new addition to a network, whether it is a sensor or an actuator, presents itself as an entry point where the attacker can attempt to take unauthorised control to capture important and private information and cause disruptions. network zones. In many cases, the vulnerabilities of IoT endpoints have been significantly amplified by the nature of their hardware: microcontroller-level processing power with a slow clock speed, kilobyte memory capacity instead of gigabytes, energy budget constrained by battery or energy harvesting, and firmware without any form of security or update authentication. Taken together, these attributes make it practically impossible to apply cryptographic layers, antivirus scans, and even stateful firewalls to IoT devices [4, 5]. This asymmetry has been well recognised and effectively used by malicious actors: the Mirai botnet, responsible for one of the largest DoS campaigns

in history at the time (reaching 620 Gbps of throughput in 2016), showed that several hundred thousand IoT endpoints can easily be co-opted for attacks based on simple credential enumeration [6].

Intrusion Detection Systems provide one of the most commonly implemented defense strategies that are complementary to the perimeter defenses in place. Instead of trying to stop every potential attack at the point of entry, which is becoming increasingly difficult with the use of devices that communicate directly with cloud services without the need for intermediary devices—an IDS works by monitoring the network activity, identifying the normal behaviour pattern, and raising an alarm in case deviations are observed from the pattern as would be expected in an attack. Early implementations of IDS systems utilized rules and signatures that were created by hand, methods that are only capable of detecting attacks that are familiar and static. However, recent developments within the IoT attack surface—characterised by mutation of malware, zero-day attacks exploiting unknown vulnerabilities, and complex multi-stage attacks—have rendered static rule-based IDS inadequate as a primary detection mechanism [7, 8].

The limitations mentioned above have been one reason behind a significant amount of work on developing intrusion detection schemes in which attack combinations can be learned by machines using algorithms without using rules generated by humans. Neural networks with many layers are especially good at this task due to their proven ability to abstract features hierarchically out of high-dimensional raw sensory inputs. The most successful attempts to tackle this problem include hybrid CNN-LSTM architectures designed to extract features locally and model sequences in time to effectively classify traffic [9, 10].

There are two key problems standing in the way of effective use of such methods on IoT datasets. First of all, the difference between the count of benign samples and rare attack samples is very large i.e. the overwhelming majority of flows observed in the traffic are benign, and there are a few types of attacks that occur frequently dominate the corpus, while many operationally critical but rare attack categories contribute only a handful of examples. Standard gradient descent training with such datasets will lead to classifiers that perform well on average but learn to neglect the minority classes altogether. GANs, which have gained traction relatively recently, are able to generate realistic samples

from the minority classes, thus restoring the balance without introducing distortions that come with feature space interpolation [11].. Second, the predictive performance of the model developed depends heavily on architecture, training hyperparameters—filter counts, LSTM unit counts, learning rate, and regularisation coefficients—whose optimal settings are data-dependent and cannot be determined analytically. Population-based metaheuristic algorithms offer a principled, parallelisable approach to this search that requires no gradient information and can evaluate many candidate configurations simultaneously [12, 13].

1.2 Motivation

The specific motivation for this work arises from a gap that was observed in the published literature that involves IoT intrusion detection system using neural networks. While individual studies have demonstrated the performance benefits of GAN-augmented CNN–LSTM models and of various metaheuristic optimisers applied to neural architecture search, virtually none have conducted a comparative evaluation in which multiple optimisers compete under strictly identical experimental conditions. Without this control, performance differences observed between studies cannot be attributed with confidence to the optimiser itself [14]. Practitioners seeking to select an optimiser for deployment therefore have access only to incomparable single-algorithm evaluations, which do not provide the evidence base needed for informed decision making.

A second motivation concerns statistical reliability. Published results in the applied IDS literature are overwhelmingly reported as single-run outcomes on a single fixed dataset split. This practice is known to produce optimistically biased accuracy estimates, particularly for stochastic optimisation algorithms whose convergence trajectories are sensitive to population initialisation. Multi-run validation across independently randomised dataset splits is a minimum requirement for trustworthy benchmarking but is seldom implemented in practice [15].

This research is directly aligned with both these aspects through the conduct of an experimental study involving multiple trials that compare the usefulness of three distinct metaheuristic optimization algorithms for tuning of hyperparameters of a CNN-LSTM

IDS augmented with GAN, when the training dataset used was CICIoT2023.

1.3 Problem Statement

Considering the CICIoT2023 benchmark dataset—an extensively imbalanced dataset composed of IoT network traffic consisting of 105 different classes of devices and 33 unique types of attacks [16]—the present dissertation focuses on solving the problem of finding an optimal solution to develop an IDS using deep learning techniques, such that it would provide good classification performance as well as reasonable per-class classification sensitivity for many different attack classes in IoT traffic; and in what cases would the tuning of the model hyperparameters using population-based metaheuristic methods deliver the best accuracy, stability, and computational efficiency? The proposed problem breaks down into five inter-related sub-problems:

1. Minimizing the negative impact of severe class imbalance to allow rare yet operationally critical attack types to have an influence on model training.
2. Designing a neural architecture that simultaneously exploits spatial feature structure and temporal dynamics within network flow data.
3. Implementing a sound hyperparameter optimisation protocol that is parallelisable, gradient-free, and bounded within a well-defined search space.
4. Establishing an experimental protocol that isolates the effect of the optimiser from all other confounding variables.
5. Quantifying the statistical robustness of results through repeated randomised trials rather than relying on a single experimental outcome.

1.4 Research Objectives

The work carried out in this thesis follows these specific objectives:

1. To construct a preprocessing and GAN-based data augmentation pipeline for the CICIoT2023 dataset that reduces class imbalance for both 41-class and 9-class classification tasks.

2. To develop, implement and improve the hybrid neural network for multi-class IoT intrusion detection and to benchmark it against established classical machine learning classifiers.
3. To implement STO, GA, and PSO as hyperparameter optimisers for the CNN–LSTM model, operating within a shared, explicitly defined search space.
4. To conduct a fully controlled comparative study of the three optimisers under identical experimental conditions, measuring test accuracy, per-class F1- scores, convergence trajectories, and total computational runtime.
5. To validate findings through five independent randomised dataset splits, reporting mean accuracy and standard deviation for each optimiser.
6. To conduct systematic error analysis identifying the failure modes of the proposed framework and recommending evidence-based future work directions.

1.5 Contributions to the Research

These are the original contributions to fill the research gap:

1. **Validated cGAN Augmentation Pipeline:** A conditional GAN framework is implemented and verified for generating synthetic traffic samples targeting severely underrepresented attack categories in CICIoT2023, producing approximately 4,962 additional training instances and restoring minimum per-class representation thresholds.
2. **Hybrid CNN–LSTM Architecture for IoT IDS:** A 1D convolutional network sequentially composed with LSTM layers is architected for IoT network traffic classification, achieving 93.83% accuracy on a fine-grained 41-class task with 81,193 trainable parameters.
3. **STO-Optimised Model:** STO-driven hyperparameter search is shown to yield the highest test accuracy (94.45%) on the consolidated 9-class task among all optimisers tested, along with the lowest inter-run performance variance.
4. **Controlled Comparative Benchmarking:** A multi-run statistical benchmarking study isolates the effect of the optimiser while holding all other experimental variables constant, revealing that PSO's apparently competitive single-run accuracy

(93.59%) conceals severe instability (mean 81.79% $\pm$ 5.26%) across independent runs.

5. **Root-Cause Error Analysis:** Systematic investigation of per-class classification failures attributes near-zero F1-scores for rare attack categories to sample scarcity, feature-space overlap, and intra-class heterogeneity, and proposes targeted remediation strategies for each.

1.6 Organisation of the Thesis

Chapter 2 surveys prior work on IoT security, AI based methods for detecting network intrusion, GAN-based data augmentation, and metaheuristic hyperparameter optimisation. Chapter 3 introduces the theoretical foundations of each algorithmic component. The fourth Chapter gives the overview of the dataset used and the methods followed to clean the data. In Chapter 5, the entire methodology of the system is presented. Chapter 6 describes the analysis of the experimental findings in detail. Chapter 7 contains the conclusion and highlights the possible social effects of the suggested system. It also outlines the future work to be conducted.

The subsequent chapter surveys existing literature on securing IoT networks, classical machine learning and deep learning approaches for threat detection, GAN-based artificial data generation, and various population-based metaheuristics for parameter tuning, while detailing the research gaps addressed in this work.

CHAPTER 2

LITERATURE REVIEW

2.1 Securing the Internet of Things

If we talk about the security of the IoT networks, it is unique from enterprise security not just in terms of scale. IoT endpoints can be installed in places where they cannot always be kept under surveillance; they use different types of wireless protocols, each of these has different levels of authentication and encryption built into them; and they rely on lightweight firmware that never gets patched because it is only installed once [5, 50].

The taxonomy of attacks on IoT networks is varied and continuously evolving. DDoS attacks take benifit of huge number of devices which are connected to launch an overwhelming amount of traffic beyond the ability of target infrastructure to process [6]. Reconnaissance involves such techniques as port scanning, OS fingerprinting, and service enumeration to discover the structure of the network topology and the exploitable services therein. Spoofing attacks use IP, MAC, or DNS address to misdirect traffic flow or impersonate a host. Brute-force attacks systematically test username-password pairs until the correct authentication credentials are found. Web application attacks involve exploits on HTTP-based management interfaces used by IoT gateways through injection attacks such as SQL injection, Cross-Site Scripting (XSS), and command injection in web applications. Botnet attacks involve the creation of IoT-based botnets through exploitation of default credentials; subsequent Mirai variants have proven resilient and have evolved methods of distributing malware [48, 49].

Intrusion detection at the network level—which involves analysing features that are visible in the traffic—is a practical way of conducting detection without having to modify limited devices [7]. What needs to be accomplished is for detection algorithms to be efficient, accurate, and robust to distribution shifts caused by changes in devices and traffic.

2.2 Using Traditional ML Algorithms for Intrusion Detection

ML methods used for detecting network intrusions existed much before the DL methods emerged, by many decades. Initial attempts involved handcrafting statistical characteristics from packet headers and flow statistics, which were then fed to classifiers such as Naive Bayes, KNN, Decision Tree etc. to identify benign flows from the malicious ones [8]. Decision Tree classifier algorithms became popular due to their explainability in terms of the rules derived that can be understood by security experts. Random Forest improved the robustness of decision tree algorithms by generating ensembles of trained trees using random portions of training data and combining results, which helped in reducing the high variance commonly seen in decision tree models [41]. Support Vector Machines with radial basis kernel demonstrated good generalisation capability in high dimensional spaces, although the training cost on large-scale modern datasets was prohibitively high [42].

More recent ensemble work has pushed the limits of classical methods on contemporary IoT datasets. Tawfik [35] combined ensemble learning with feature selection on CICIoT2023 and achieved 98% accuracy for a selected subset of attack classes, though this result was conditioned on a restricted class configuration rather than the full attack taxonomy. Gheni and Al-Yaseen [36] used the clusting approach having two stages before applying classification. and reported 95% accuracy in a binary benign-versus-attack mode. While these results are impressive, binary or restricted-class configurations substantially underestimate the difficulty of the full multi-class detection problem.

2.3 Deep Neural Network Approaches for Network Threat Detection

The convergence of large-scale IoT traffic datasets and affordable GPU computing catalysed a transition toward deep learning-based IDS architectures. Autoencoders trained on normal traffic to reconstruct benign flows were among the earliest deep architectures applied to anomaly detection [31]. Meidan et al. [27] demonstrated that device-specific deep autoencoders achieved high detection rates for Mirai botnet activity on the N-

BaIoT dataset by flagging reconstruction errors that exceeded a learned threshold. The limitation of this approach is its reliance on device-specific training, which precludes generalisation to new or previously unseen device types without retraining.

Recurrent neural networks, particularly LSTMs, is used to capture the temporal nature of network flows. In this regard, the work of Parra et al. [29] proved the usefulness of distributed LSTM models for modeling inter-packet timing and payload features corresponding to attacks belonging to different classes of attacks, demonstrating superior performance compared to static classification based on hand-engineered features. Furthermore, attention mechanisms have been incorporated into recurrent models in order to weight different time steps in terms of their importance for predicting attack activity on particular time intervals [51].

A more recent approach to traffic classification has been to use transformer-based deep learning architectures [52], making use of the ability of a transformer to model long-range dependencies between time steps without having to process them sequentially as it is done by RNNs. The problem of privacy was considered by applying federated learning that allows the training of IDS models in a distributed manner while protecting raw network data from exposure [53]. Another interesting approach to intrusion detection was the application of GNNs for modeling inter-device relationships through graphs [54].

2.4 Hybrid CNN–LSTM Architectures for Intrusion Detection

This phenomenon that network traffic contains local spatial structure, where features in adjacent positions within a flow exhibit correlation, and temporal structure, where features within each packet or monitoring window change with time, was leveraged in building a hybrid architecture that can take advantage of both types of structure in a single model. Canonical examples of such architectures use convolutional neural networks (CNNs) in one dimension in order to discover spatial interactions between features followed by Long Short-Term Memory networks (LSTMs) describing the evolution of representations generated from those interactions.

Altunay and Albayrak [18] successfully used the CNN–LSTM framework for modeling IoT industrial network traffic with a 90+% accuracy score on the latest bench-

marks, thus demonstrating its applicability in practice as an architecture for deploying IDSs. Gueriani et al. [19] further enhanced this architecture by introducing multi-head attention into it, which allowed increasing the detection rate for low prevalence attacks by focusing attention of the model on relevant temporal windows. Chouhan et al. [20] developed a hierarchical CNN-LSTM architecture for IoT systems based on SDN.

Afraji et al. [21] pushed the boundaries even further by adding Gated Recurrent Unit (GRU) layers onto a CNN–LSTM hybrid architecture resulting in a CNN–LSTM–GRU sequence. Li and Liu [9] added residual connection mechanisms in convolutional blocks in a hybrid architecture to reduce vanishing gradients when learning from long traffic sequence data sets. Yang et al. [10] replaced the LSTM layers with Temporal Convolutional Network layers for learning from temporal information, achieving similar accuracies in less time. All of these contributions clearly prove the superiority of hybrid architectures over other architectures but they also highlight that hyperparameter tuning is still a significant limitation in hybrid model training.

2.5 GAN-Based Data Augmentation for Imbalanced Traffic

Class imbalance is one such structural barrier which is seen while detecting intrusion in IoT networks, where the natural occurrence of the distribution of data is heavily skewed towards benign traffic and attack classes of high frequency. In the CICIoT2023 dataset, the imbalance ratio between the most and least frequent classes exceeds 500:1, which causes standard cross-entropy training to effectively ignore minority classes [16].

Conventional oversampling via SMOTE addresses imbalance by generating new samples bylinearly interpolating the existing minority samples [37]. While computationally inexpensive, SMOTE cannot accurately model the complex nonlinear manifold geometry of realistic attack traffic and may generate unrealistic samples in regions of the feature space not spanned by the interpolation path. Class- weighting strategies that amplify the gradient contribution of minority-class misclassifications offer a complementary approach that requires no synthetic sample generation, but do not increase the diversity of the training distribution for rare classes.

Goodfellow et al. [11] introduced GANs as a framework in which a generator

network learns to produce samples indistinguishable from real data by competing against an adversarially trained discriminator. Conditional GANs extend this capability by conditioning both networks on class labels, enabling targeted generation of samples for specified categories [38]. The adversarial training objective ensures that generated samples respect the full distributional structure of the target class, including higher-order statistical dependencies not captured by linear interpolation.

Zeghida et al. [25] demonstrated that cGAN-based augmentation substantially improved detection performance for MQTT-based IoT traffic, with the largest gains observed for minority attack classes that contributed fewer than 100 training instances. Hao et al. [39] compared multiple GAN variants for network intrusion augmentation and found that conditional variants consistently outperformed unconditional models due to their ability to target generation at specific underrepresented categories. Ding et al. [40] applied CTGAN—a tabular GAN variant specifically designed for mixed-type datasets—to IDS augmentation and reported improved handling of categorical features compared to standard cGANs.

2.6 Metaheuristic Algorithms for Hyperparameter Optimisation

Selecting appropriate hyperparameters for deep neural networks is a combinatorial-continuous mixed optimisation problem that resists analytical treatment. Grid search exhaustively evaluates combinations on a fixed discretisation but scales exponentially with dimensionality; random search alleviates the curse of dimensionality but provides no mechanism for exploiting good regions of the search space. Bayesian optimisation forms a probabilistic surrogate of the objective and balances exploration and exploitation through an acquisition function, but struggles with parallelism and is sensitive to prior specification in high-dimensional spaces [44].

Population-based metaheuristic algorithms address these limitations by maintaining all the possible solutions that collectively visits the full search space in parallel. The gradient-free nature makes them applicable to any objective, including non-differentiable or stochastic fitness signals.

Genetic Algorithms [13, 22] model Darwinian evolution: a population of encoded

candidate configurations is evolved over successive generations through tournament selection, crossover (offspring generation from parent combinations), and mutation (random perturbation of individual parameters). The stochastic operators maintain population diversity, enabling escape from local optima at the cost of noisier convergence trajectories.

Particle Swarm Optimisation [12] models the collective intelligence of animal swarms: each particle maintains a position in hyperparameter space and a velocity vector that is updated at every iteration by combining the previous best position of the particle with the best global position of the swarm discovered. Linear reduction in the inertia parameter progressively shifts the search from global search toward local exploitation. A well-documented weakness of PSO is that it converges too early, where the swarm coalesces around a less optimal solution early in the search when a strong candidate is found quickly, leaving the budget insufficient for recovery.

The Siberian Tiger Optimisation algorithm [17] is a more recently proposed bio-inspired method that models the hunting strategies of Siberian tigers: directed exploitation of the current best position is combined with stochastic exploratory jumps whose amplitude is governed by an adaptive step-size coefficient that responds to real-time population diversity. When population diversity is low, exploratory jumps are amplified to prevent stagnation; when diversity is high, directed exploitation is emphasised to refine identified high-quality regions. This dynamic adaptation addresses the premature convergence weakness of fixed-schedule algorithms.

Chen et al. [24] integrated the Reptile Search Algorithm with a CNN–LSTM architecture for industrial IoT IDS and demonstrated convergence speed advantages over PSO-tuned counterparts on several public benchmarks. Al-Sarray et al. [14] proposed a GAN-CL-STO framework for Software-Defined Networking-based IoT IDS and reported that STO-driven configuration substantially outperformed alternative optimisers on a 10-class classification task. Abualigah et al. [45] surveyed the application of population-based metaheuristics to deep learning hyperparameter optimisation and identified STO, Grey Wolf Optimizer, and Whale Optimisation Algorithm as particularly competitive alternatives to established PSO-based strategies.

2.7 Studies Using the CICIoT2023 Dataset

The CICIoT2023 dataset [16], produced by the Canadian Institute for Cybersecurity, has rapidly become one of the most widely referenced IoT traffic benchmarks in the intrusion detection literature. Its realism derives from the use of a physical testbed of 105 commercially available IoT devices—including smart cameras, IP thermostats, electronic locks, smart hubs, and industrial sensors— operating under realistic network conditions rather than traffic simulation.

Beyond the works cited above, Bhuiyan [15] studied cross-dataset generalisation of deep learning classifiers on CICIoT2023, finding significant performance degradation when models were transferred without adaptation to unseen device populations—a result that motivates the domain adaptation and federated learning directions proposed in this thesis. Parmisano et al. [32] documented the importance of labelling methodology and temporal alignment in constructing credible IoT traffic datasets. Vaccari et al. [33] and Booij et al. [34] have produced complementary IoT benchmarks Parmisano et al. [32] showed the significance of label methodology and time synchronization to build credible traffic data sets for IoT devices. Similarly, there exist two benchmarks by Vaccari et al. [33] and Booij et al. [34], called MQTTset and ToN_IoT, which could be validated against each other as highlighted in Chapter 7. that provide useful cross-dataset validation opportunities noted in Chapter 7.

2.8 Research Gaps and Positioning of This Work

There are total three gaps that has been identified in the existing literature that is the main motivation of this thesis.

First, no published work has compared multiple population-based metaheuristic optimisers for deep learning-based IoT IDS under strictly identical experimental conditions—same dataset, same model architecture, same training protocol, and same fitness evaluation criteria. Individual optimisers are studied in isolation, making it impossible to draw algorithm-specific conclusions with confidence.

Second, GAN-based data augmentation has been studied largely as an indepen-

dent component, with few investigations of how the choice of augmentation strategy interacts with the downstream hyperparameter optimisation process.

Third, multi-run statistical validation through repeated randomised dataset splits is rarely reported in the applied IDS literature, leaving the robustness and reliability of published performance claims largely uncharacterised.

This thesis directly addresses all three gaps through a rigorously controlled, multi-run comparative study that treats augmentation and optimisation as jointly designed components of an integrated pipeline.

The next chapter details the fundamental technical concepts of 1D Convolutional Neural Networks, LSTM blocks, Conditional Generative Adversarial Networks, and the three metaheuristic optimisers (STO, GA, and PSO), alongside the classification performance metrics employed in this work.

CHAPTER 3

BACKGROUND CONCEPTS

3.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) represent a specialised category of deep neural architectures distinguished by the use of spatially local weight sharing, implemented through learnable filter banks rather than fully connected weight matrices. The task of every filter is to scan the input array and calculate the dot product activation depending on the size of the filter in order to find the feature map which indicateds the learning by the filter. In the one-dimensional formulation appropriate for tabular flow data, a filter of kernel size k operates across consecutive groups of k features, enabling the detection of co-activation patterns among adjacent feature dimensions.

The classical design of 1D CNN consists of a convolutional layer, followed by a Rectified Linear Unit (ReLU) activation function and an optional pooling layer. Then the Max-pooling operation is used to find the maximum activation value corresponding to each non overlapping window, thus adding local translational invariance to the feature map and shrinking its spatial dimensionality. Multiple stacked convolutional blocks result in progressively abstract feature maps, as the initial layers detect co-activations at low levels and combine them into higher level representations at deeper layers. Normalising activations in each layer with respect to a mini-batch during training leads to faster convergence and less sensitivity to hyperparameter tuning. Dropout randomly drops part of activations from computations during training, acting as an implicit ensemble regulariser and preventing overfitting [43]. In the described approach, each 1D convolutional layer processes the 46-dimensional flow feature vector to discover meaningful spatial relationships of features, which then get processed by the temporal component.

3.2 Long Short-Term Memory Networks

The main task of Long Short-Term Memory (LSTM) networks [23] was to eliminate the vanishing gradient problem that bounds the effective memory span of standard conventional Recurrent Neural Networks. The key innovation uses a gated cell structure that carefully controls the information flow through time, helping the network to retain important information over extended sequences and also effectively filtering out less important details through learned gating operations.

Consider the previous hidden state as h_{t-1}, the previous cell state as c_{t-1}, and the current input as x_t, the LSTM unit calculates three gates:

$$f_t = \sigma\big(W_f[h_{t-1}, x_t] + b_f\big), \quad i_t = \sigma(W_i[h_{t-1}, x_t] + b_i), \quad o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{3.1}$$

where $\sigma(\cdot)$ represents the activation function (sigmoid). The calculation for the potential cell update, the revised cell state, and the output hidden state are then computed as:

$$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{3.2}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{3.3}$$

$$h_t = o_t \odot \tanh(c_t) \tag{3.4}$$

where $\odot$ represents Hadamard product. In network intrusion detection, the LSTM component models how statistical flow properties evolve across consecutive packets or monitoring windows, providing temporal context that is unavailable to classifiers operating on single flow snapshots.

3.3 Generative Adversarial Networks

Generative Adversarial Networks [11] are a class of generative models that learn to produce synthetic samples indistinguishable from real data through a competitive two-player optimisation. The architecture comprises two networks: a generator G that maps random noise vectors $z \sim p_z(\cdot)$ to synthetic samples $G(z) \in \mathscr{X}$, and a discriminator $D : \mathscr{X} \to [0,1]$ that outputs the probability of a given sample being real rather than generated. The joint minimax objective is:

$$\min_G \max_D \; \mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))] \tag{3.5}$$

At the Nash equilibrium of this game, G produces samples whose distribution matches the true data distribution [11]. Conditional GANs [38] extend the framework by conditioning both G and D on an auxiliary class label y, so that the generator $G(z,y)$ produces samples specifically belonging to class y. This class conditioning is critical for minority-class augmentation in imbalanced datasets, as it prevents the generator from conflating feature distributions across different attack categories.

3.4 Siberian Tiger Optimisation

Siberian Tiger Optimisation [17], developed by Trojovský in 2022, is a metaheuristic optimiser based on the adaptive hunting process carried out by Siberian tigers. The metaheuristic maintains a population of N potential solutions (tigers), each one of which corresponds to a point within the multidimensional space containing possible hyperparameter configurations. Each generation of the metaheuristic involves the adjustment of the positions occupied by the tigers, which occurs through an operation involving two types of steps: a directed step that involves moving closer to the current best-performing solution and a stochastic step that depends on positional relationships between members of the population.

The most important aspect that sets STO apart from fixed exploitation-exploration algorithms is the presence of a step size adjustment procedure, which adapts the balance between the two movements in each step on the basis of the level of diversity present

within the population at any time. When the latter is high, the metaheuristic focuses on the exploitation of promising areas; conversely, when convergence occurs prematurely, resulting in a drop in diversity, a larger stochastic step size is used to move away from the local attractor.

3.5 Genetic Algorithm

Genetic Algorithm (GA) [13, 22] is a well-known metaheuristic that exploits analogies with natural selection in the biological world. It evolves a population of N solutions represented as real-valued vectors of the hyperparameters using the following three operations.

Selection: Tournament Selection with tournament size k is used: k members of the population are chosen randomly; the one with the highest fitness is selected as the parent for reproduction. Thus, while selection pressure corresponds to individual fitness levels, weaker individuals are still present in the population.

Crossover: Offspring is created via BLX-α crossover with $\alpha = 0.5$: offspring coordinates are randomly selected from an interval that extends past the parent coordinates' range by α times, allowing offspring solutions to explore the search space outside the current population domain.

Mutation: Offspring coordinates are modified with additive Gaussian noise:

$$\theta' = \theta + \mathcal{N}(0, \sigma^2), \quad \sigma = 0.1 \tag{3.6}$$

This mutation allows offspring to explore different regions of the search space, thus avoiding premature convergence. Convergence trajectory of GA is known to be noisy but free from local optima entrapment.

3.6 Particle Swarm Optimisation

The Particle Swarm Optimisation algorithm [12] represents the phenomenon of collective intelligence that arises among birds or fish in nature. In particle swarm optimisation,

N particles move through hyperparameter space where, for each particle i, its location is defined by x_i^t and its speed v_i^t during iteration t. Particle velocities and locations are updated according to the following equations:

$$v_i^{t+1} = w\,v_i^t + c_1 r_1 (p_i - x_i^t) + c_2 r_2 (g - x_i^t) \tag{3.7}$$

$$x_i^{t+1} = x_i^t + v_i^{t+1} \tag{3.8}$$

where w is the inertia parameter which is decayed from 0.9 to 0.4 with a linear decay factor, $c_1 = 0.5$ is the cognitive parameter that represents how much emphasis is placed on individual experience, $c_2 = 0.3$ is a social parameter representing the importance of the swarm's global best, p_i is the personal best of particle i, g is the global best, and $r_1, r_2 \sim \mathscr{U}(0,1)$ are randomly generated scalars every time during the update process. As w decays linearly, there is a progressive shift from global search to local search. Premature convergence is the major weakness associated with PSO; once the swarm converges to a promising area early, the search budget will not suffice to escape local optima.

3.7 Evaluation Metrics

In the case of a C-class classification problem, TP_k, TN_k, FP_k, and FN_k represent the numbers of true positives, true negatives, false positives, and false negatives for class $k \in \{1, \ldots, C\}$ respectively. The primary evaluation metrics are:

$$\text{Accuracy} = \frac{\sum_{k=1}^{C} \mathrm{TP}_k}{\sum_{k=1}^{C} (\mathrm{TP}_k + \mathrm{TN}_k + \mathrm{FP}_k + \mathrm{FN}_k)} \tag{3.9}$$

$$\text{Precision}_k = \frac{\mathrm{TP}_k}{\mathrm{TP}_k + \mathrm{FP}_k} \tag{3.10}$$

$$\text{Recall}_k = \frac{\mathrm{TP}_k}{\mathrm{TP}_k + \mathrm{FN}_k} \tag{3.11}$$

$$F_{1,k} = 2 \times \frac{\text{Precision}_k \times \text{Recall}_k}{\text{Precision}_k + \text{Recall}_k} \tag{3.12}$$

The Macro-F_1 score measures the average of $F_{1,k}$ for all classes, giving equal weight to all classes, independent of their representation. Such an evaluation is very useful in the scenario of extreme class imbalance, where the ability to detect each individual class of attacks cannot be measured using overall accuracy.

The next chapter presents a comprehensive overview of the CICIoT2023 dataset, including its feature structure, severe class distribution imbalances, and the preprocessing workflow incorporating min-max scaling and cGAN-based minority oversampling.

CHAPTER 4

DATASET AND PREPROCESSING

4.1 CICIoT2023 Dataset Overview

One such publicly available benchmark, known as CICIoT2023 [16], was released in 2023 by the Canadian Institute for Cybersecurity. In particular, the collection process involved setting up an experimental testbed consisting of commercially available IoT devices totalling to 105 individual items categorised into twelve distinct device classes: smart cameras, IP thermostats, electronic door locks, smart home hubs, network-attached storage devices, smart plugs, robotic vacuum cleaners, industrial environmental sensors, smart TVs, connected printers, wireless access points, and home automation controllers. Using physically available devices and realistic network conditions has been essential in ensuring that the data is not only relevant but also representative of real IoT systems.

The dataset encompasses both benign baseline traffic and 33 distinct attack categories spanning the following major families: Distributed Denial of Service (DDoS, including UDP flood, SYN flood, ICMP flood, HTTP flood, and amplification variants), Denial of Service (DoS), Mirai botnet activity (scanning, brute-force, and propagation sub-types), reconnaissance operations (OS fingerprinting, port scanning, and service enumeration), spoofing (ARP and DNS variants), brute-force credential attacks, and web application exploitation. A total of 46 statistical flow features per record are extracted using CICFlowMeter, encompassing forward and backward packet length statistics (minimum, maximum, mean, variance), inter-arrival time distributions, TCP flag occurrence counts, active and idle time statistics, and flow-level byte transfer rates. The breadth of attack families, the physical realism of the collection environment, and the richness of the feature representation make CICIoT2023 well suited to multi-class intrusion detection research.

4.2 Subset Selection and Train-Test Partitioning

Each fitness evaluation during metaheuristic hyperparameter optimisation requires training a CNN–LSTM model from scratch; across a full optimisation run of N agents and T iterations, the number of required training episodes is $N \times T$. Training the model on the full dataset for each episode would make the total computational cost prohibitive. To balance representativeness with feasibility, a stratified random sample of 100,000 records is drawn from the complete dataset. Stratified sampling preserves the original class distribution proportionally across the sample, ensuring that rare attack categories retain representation in both the training and test subsets.

The 100,000 records are partitioned into an 80% training set (80,000 samples) and a 20% test set (20,000 samples) using stratified splitting. For statistical validation experiments, five independent random seeds are used to generate five distinct stratified shuffles of the 100,000-sample subset, each producing a fresh 80:20 train-test partition.

4.3 Class Distribution and Imbalance Analysis

Table 4.1 summarises the class distribution across the nine coarse-grained categories used in the primary evaluation task.

Table 4.1: Approximate Class Distribution in the 9-Class Configuration (Prior to GAN Augmentation)

Class	**Count**	**Share (%)**	**Type**
DDoS	∼29,420	29.42	Attack
Benign	∼25,100	25.10	Normal
DoS	∼17,830	17.83	Attack
Mirai	∼12,050	12.05	Attack
Recon	∼8,240	8.24	Attack
Spoofing	∼4,310	4.31	Attack
Brute Force	∼1,890	1.89	Attack
Other	∼1,110	1.11	Attack
Web Attack	∼50	0.05	Attack

The imbalance ratio for the class with the highest number of occurrences (DDoS) and the lowest number of occurrences (Web Attack) is greater than 500. In minimising cross-entropy loss, the contribution made by gradients from examples of classes with lower numbers of instances is insignificant compared to those with higher numbers, hence allocating discriminatory power only to the former.

4.4 Preprocessing Pipeline

Each of the 46 numerical flow features is normalised to lie within the $[0,1]$ range through Min-Max normalisation. The ith data point of feature j with a minimum value of $x_j^{\min}$ and a maximum value of $x_j^{\max}$ is normalised as follows:

$$x_{ij}^{\text{norm}} = \frac{x_{ij} - x_j^{\min}}{x_j^{\max} - x_j^{\min}} \tag{4.1}$$

Normalisation ensures that features having large absolute ranges do not dominate the gradient updates, while ensuring that all input features have the same level of scale, thus leading to faster convergence.

Attack category labels are mapped to integer indices through ordinal label encoding. For the 41-class task, each of the 41 distinct fine-grained attack sub-categories receives a unique integer identifier. For the 9-class task, fine-grained labels are first aggregated into coarse families (all DDoS protocol variants into a single DDoS class, all Mirai sub-types into a Mirai class, and so forth) before encoding.

Feature arrays are reshaped from two-dimensional matrices of shape $(N, 46)$ into three-dimensional tensors of shape $(N, 1, 46)$ to match the input convention of the 1D CNN layers, where the second dimension represents the time-step axis—each flow record constitutes a single time step in this formulation—and the third dimension carries the 46 flow features.

4.5 GAN-Based Data Augmentation

To counteract the class imbalance described in Section 4.3, a Conditional GAN is implemented for targeted synthetic sample generation. The generator network receives as input the concatenation of a 100-dimensional Gaussian noise vector $z \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and a 16-dimensional learned embedding of the one-hot class label y, and is trained to produce a 46-dimensional synthetic flow feature vector representative of class y. The discriminator receives either a real or generated flow record concatenated with the same 16-dimensional label embedding, and outputs a scalar probability that the input is a genuine traffic record rather than a synthetic one.

Both networks are trained jointly for 200 epochs with a batch size of 64 using the Adam optimiser with learning rate $\eta = 10^{-4}$ and first-moment decay $\beta_1 = 0.5$; gradient clipping at a maximum norm of 1.0 stabilises adversarial dynamics. After training, the generator is used to synthesise additional samples for each minority class until the training corpus contains a minimum of 500 samples per class. In total, approximately 4,962 synthetic samples are added across all underrepresented categories, with the largest contributions allocated to the Web Attack and SlowLoris sub-categories. Additionally, class weights inversely proportional to class frequency are applied during CNN–LSTM training to further amplify the gradient contribution of minority-class errors.

The subsequent chapter describes the system architecture of the proposed intrusion detection framework, the baseline models, the hybrid CNN-LSTM network configuration, the metaheuristic hyperparameter search spaces, and the experimental protocol.

CHAPTER 5

PROPOSED METHODOLOGY

5.1 Overall System Architecture

The proposed intrusion detection framework integrates three sequential stages:

1. **Data Preparation:** Raw CICIoT2023 traffic records are subjected to Min-Max normalisation, ordinal label encoding, tensor reshaping, and cGAN-based minority-class augmentation to produce a class-balanced training corpus.
2. **Model Construction and Training:** A hybrid CNN–LSTM classifier is trained on the augmented corpus using early stopping to prevent overfitting.
3. **Metaheuristic Hyperparameter Optimisation:** One of three population-based algorithms—STO, GA, or PSO—drives a search over the shared hyperparameter space, using the CNN–LSTM validation accuracy as the fitness signal.

Figure 5.1 provides a schematic overview of this three-stage architecture. All three stages are executed sequentially; the metaheuristic optimisation stage wraps the model training stage as its fitness evaluation function.

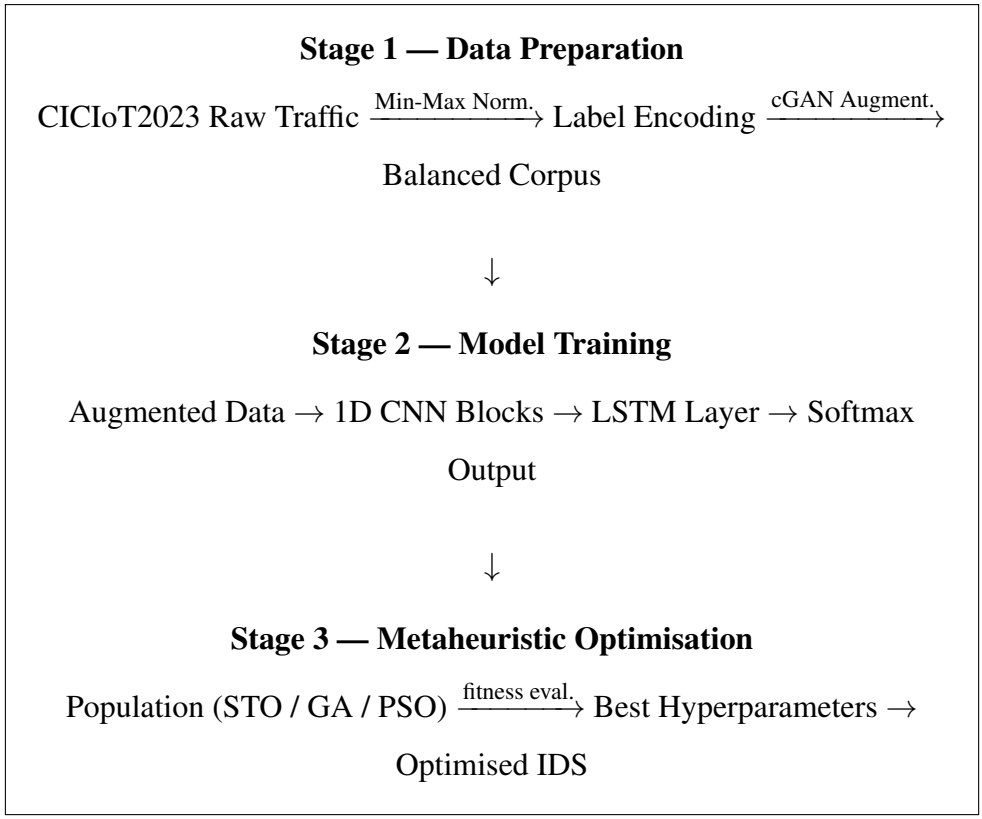


Figure 5.1: High-level architecture of the proposed GAN-augmented, metaheuristic-optimised CNN–LSTM intrusion detection framework.

5.2 Baseline Machine Learning Models

Four classical machine learning classifiers serve as performance references:

Decision Tree (DT): Recursively partitions the feature space by selecting, at each internal node, the feature and threshold that minimise the Gini impurity of the resulting child partitions.

Random Forest (RF): Constructs an ensemble of B decision trees, each trained on an independently bootstrapped subset of the training data with random feature subsampling at each split; predictions are aggregated by majority voting across all trees [41].

Support Vector Machine (SVM): Employs an RBF kernel to implicitly map inputs into a high-dimensional feature space and identifies the maximum-margin separating hyperplane in that space [42].

Logistic Regression (LR): Fits a linear discriminant boundary over the 46-dimensional feature space by minimising multi-class cross-entropy loss through an

iterative solver, providing an analytically interpretable linear baseline.

All four classifiers are trained with their scikit-learn default parameter settings and a fixed random seed of 42 to ensure reproducibility.

5.3 Hybrid CNN–LSTM Architecture

The central deep learning model is a hybrid 1D CNN–LSTM network whose layer-wise configuration is specified in Table 5.1. The input is an $(N, 1, 46)$ tensor representing a batch of N flow records, each treated as a single-time-step sequence of 46 features.

Table 5.1: Layer-Wise Configuration of the CNN–LSTM Network

Layer	**Output Shape**	**Parameters**
Conv1D (64 filters, kernel 3, ReLU)	$(N, 44, 64)$	256
MaxPooling1D (pool size 2)	$(N, 22, 64)$	0
Conv1D (128 filters, kernel 3, ReLU)	$(N, 20, 128)$	24,704
MaxPooling1D (pool size 2)	$(N, 10, 128)$	0
LSTM (64 units, Dropout 0.3)	$(N, 64)$	49,408
Dense (64, ReLU)	$(N, 64)$	4,160
Dense Output (41 or 9, Softmax)	$(N, 41/9)$	2,665
Total Trainable Parameters		**81,193**

The two stacked convolutional blocks extract progressively abstract spatial feature representations, with each max pooling step halving the feature-map width and encouraging translation-invariant detection of local feature patterns. The LSTM layer receives the sequence of convolutional feature maps across the time dimension and models temporal dependencies; dropout at rate 0.3 within the LSTM layer regularises training. The final dense layer integrates LSTM output features, and the softmax output layer produces a normalised probability distribution over the C classes.

Training uses sparse categorical cross-entropy loss, the Adam optimiser with an initial learning rate of $\eta = 0.001$, and early stopping with patience of 5 epochs monitoring validation loss to terminate training when improvement ceases.

5.4 Metaheuristic Optimisation Framework

Each metaheuristic algorithm evaluates candidate hyperparameter configurations through a shared fitness function: the validation accuracy obtained by training the CNN–LSTM model for up to 10 epochs (with early stopping) under the configuration specified by the candidate solution. All three optimisers share the same search space:

- Convolutional filters: integer in $[32, 128]$
- LSTM hidden units: integer in $[32, 128]$
- Learning rate: real in $[10^{-4}, 10^{-2}]$ (log-uniform scale)

Candidate values outside these bounds are projected to the nearest boundary. Population size is set to 8 for all three algorithms; GA and PSO execute 4 iterations each, while STO executes 5 iterations, yielding comparable total fitness evaluation counts across algorithms.

5.4.1 Genetic Algorithm Configuration

Tournament selection with tournament size 3, BLX-α crossover with $\alpha = 0.5$, and Gaussian mutation with $\sigma = 0.1$ applied at probability 0.2 per dimension are used. The population comprises 8 individuals evolved across 4 generations.

5.4.2 Particle Swarm Optimisation Configuration

This swarm contains 8 particles which have cognitive constant $c_1 = 0.5$, social constant $c_2 = 0.3$, and inertial constant w decaying linearly from 0.9 to 0.4 across 4 iterations. Particle velocities are constrained during each generation to ensure that none of the particles move out of the search domain.

5.4.3 Siberian Tiger Optimisation Configuration

In the STO, ten agents are used for five different iterations. The value of exploration probability is set at 0.5; however, the balance between exploration and exploitation is adapted dynamically depending on the population diversity at every iteration [17].

5.5 Experimental Setup

All the experiments are conducted using Python 3.9, where TensorFlow/Keras is employed for creating, training, and optimising neural network modules, while scikit-learn is utilised to design baseline classifiers and compute evaluation metrics. To quantify statistical robustness, the complete optimisation pipeline is independently repeated five times using different random seeds governing dataset shuffling, metaheuristic population initialisation, and model weight initialisation; each repetition draws a fresh stratified 80:20 train-test split to expose sensitivity to dataset partitioning.

The following chapter presents the empirical results, including comparative benchmarking against baselines, multi-split validation analyses, convergence histories of the metaheuristic searches, and a detailed classification error breakdown.

CHAPTER 6

RESULTS AND DISCUSSION

6.1 Baseline Machine Learning Performance

Table 6.1 presents the performance of the four baseline classifiers on the GAN-augmented training set, and Figure 6.1 visualises the four metrics side by side. The Random Forest ensemble achieves the strongest overall result with 95.5% accuracy and an F1-score of 0.947, benefiting from ensemble averaging that substantially reduces the variance inherent in any individual decision tree. The Decision Tree classifier performs competitively at 93.5% accuracy, demonstrating that even a non-ensemble classifier can achieve near-competitive performance when the training set is well-balanced through GAN augmentation. SVM (79.0%) and Logistic Regression (77.5%) perform considerably worse, with the SVM's performance likely limited by the quadratic scaling of its kernel matrix computation on an 80,000-sample training set, which may have precluded full convergence within the solver's iteration limit, and Logistic Regression being fundamentally constrained by its linear decision boundary in a problem where the attack categories are nonlinearly separable.

Table 6.1: Performance of Baseline ML Classifiers on GAN-Augmented Data

Model	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Decision Tree	0.935	0.930	0.932	0.931
Random Forest	0.955	0.945	0.950	0.947
SVM (RBF)	0.790	0.785	0.788	0.760
Logistic Regression	0.775	0.768	0.772	0.745

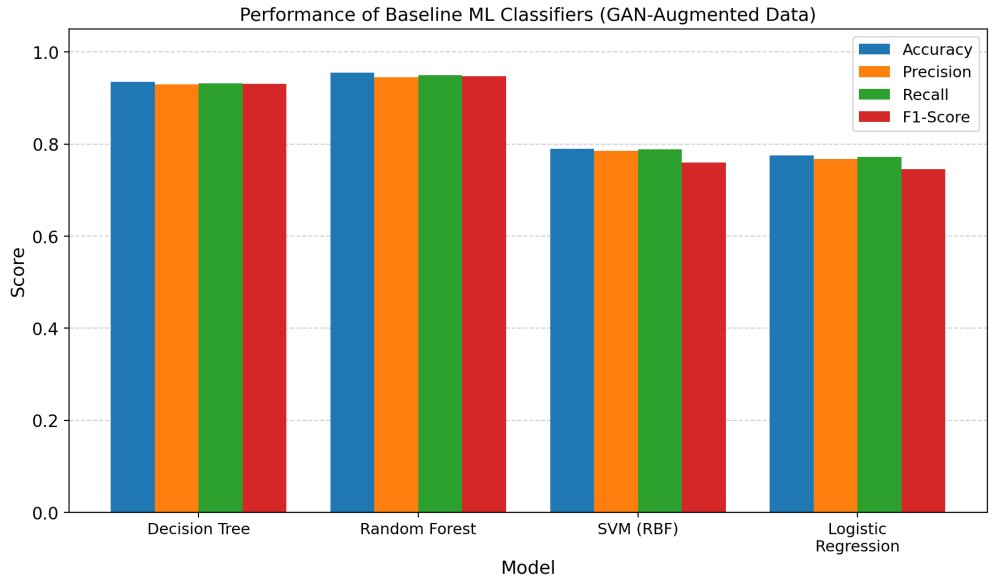
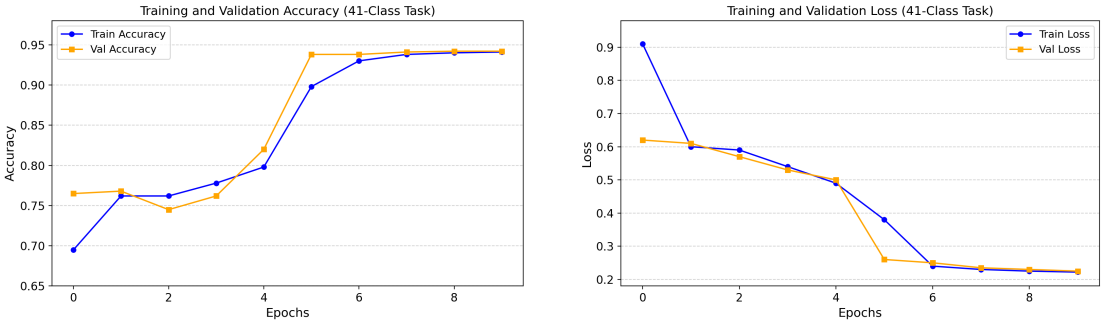


Figure 6.1: Accuracy, Precision, Recall, and F1-Score for the four baseline ML classifiers on the GAN-augmented dataset.

6.2 CNN–LSTM Training Dynamics

The epoch-wise training and validation accuracy and loss trajectories for the CNN–LSTM model on the 41-class task are presented in Figures 6.2a and 6.2b respectively. The two accuracy curves match very well during training, since the largest difference between the two curves does not exceed 1.5 percent, which suggests that overfitting is kept in check by both regularisation and early stopping. The model reaches approximately 90% validation accuracy by epoch 6 and converges close to 94% by epoch 9. The validation loss decreases from approximately 0.62 at the first epoch to below 0.25 at convergence, paralleling the training loss and confirming that the model improves meaningfully throughout the allocated budget, terminating via early stopping rather than budget exhaustion.



(a) Training and validation accuracy.

(b) Training and validation loss.

Figure 6.2: Epoch-wise training dynamics for the CNN–LSTM model (41-class task).

6.3 41-Class CNN–LSTM Performance

The trained CNN–LSTM model achieves a test accuracy of 93.83% on the 41-class fine-grained task. Figure 6.3 presents the full 41×41 confusion matrix. Strong diagonal concentration is evident for the majority of attack sub-categories: the most correctly classified individual class (DDoS-ICMP-Flood, class index 13) contributes 3,004 correct test predictions, corresponding to a per-class accuracy of approximately 97%. Misclassifications are concentrated in rare sub-categories where test sample counts are too small for reliable boundary estimation, consistent with the class imbalance analysis in Chapter 4. Residual cross-class confusion occurs primarily between protocol-specific DDoS sub-types (UDP Flood, SYN Flood, ICMP Flood) that share high-level traffic characteristics, and between reconnaissance sub-types (port scan, service enumeration) whose inter-category feature-space distances are small.

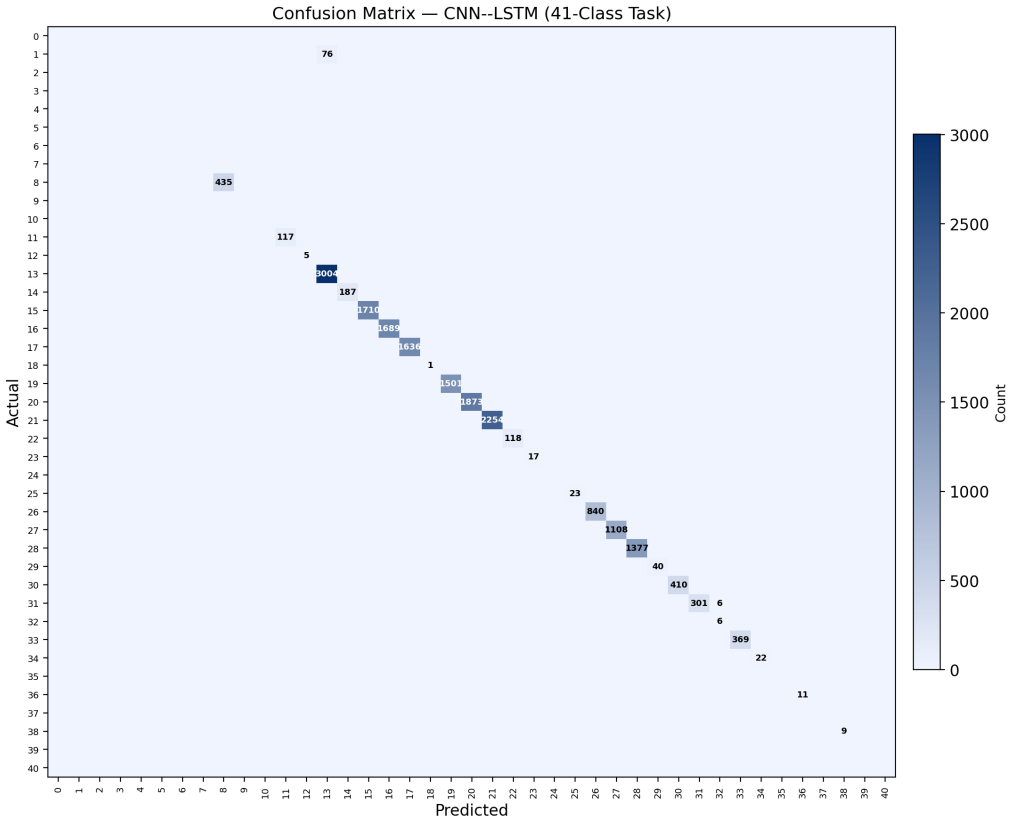


Figure 6.3: Confusion matrix for the CNN–LSTM model on the 41-class classification task.

6.4 9-Class STO–CNN–LSTM Performance

The STO-optimised CNN–LSTM model achieves a test accuracy of 94.45% on the consolidated 9-class task. Table 6.2 reports per-class precision, recall, and F1-score; Figure 6.4 presents the per-class F1-scores as a colour-coded bar chart for visual clarity.

Table 6.2: Per-Class Classification Results for STO–CNN–LSTM (9-Class Task)

Class	Precision	Recall	F1-Score
Benign	0.92	0.90	0.91
Brute Force	0.73	0.70	0.71
DDoS	0.96	0.94	0.95
DoS	0.85	0.79	0.82
Mirai	0.99	0.98	0.985
Other	0.57	0.53	0.55
Recon	0.62	0.58	0.60
Spoofing	0.68	0.63	0.655
Web Attack	0.54	0.49	0.51

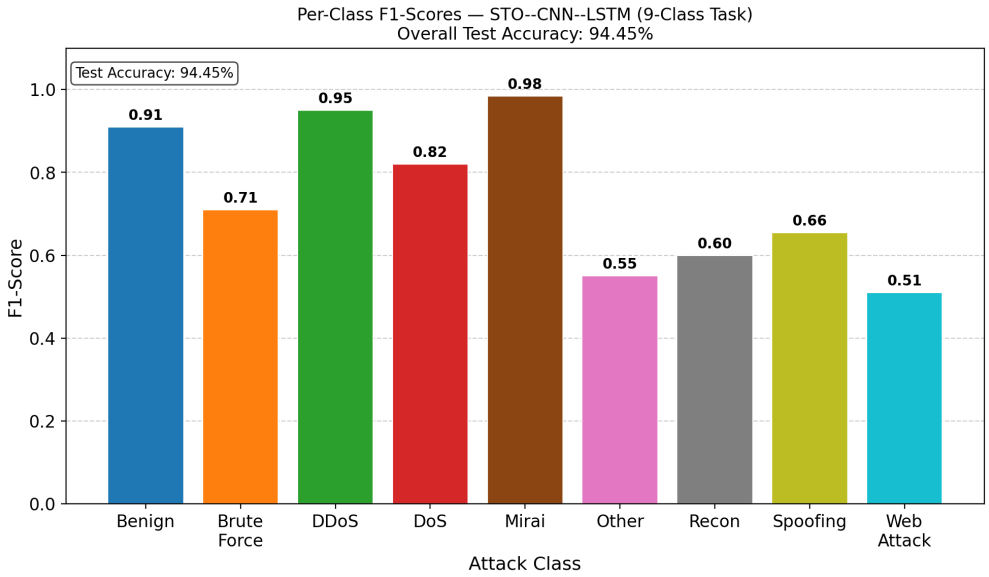


Figure 6.4: Per-class F1-scores for the STO–CNN–LSTM model on the 9-class task. Test accuracy: 94.45%.

The obtained results exhibit a high degree of consistency in the quality of the classifications in all nine traffic types. The Mirai traffic category attains the best F_1 score of 0.985 due to its distinctive traffic fingerprint pattern as well as sufficient sample size in the training phase. DDoS is second, at an F_1 score of 0.95 and Benign traffic third at 0.91, both having large and separable sample distributions for training. DoS has an F_1 score of 0.82, with some overlap between DoS and DDoS feature spaces. Brute Force

(0.71), Spoofing (0.655), Recon (0.60), and Other (0.55) have moderate scores, while the lowest score, 0.51, is achieved by the Web Attack category due to its heterogeneous nature within the same traffic class.

6.5 Comparative Analysis of Optimisers

Table 6.3 shows the test accuracy and total runtime of the three optimisers for the main train-test partition.

Table 6.3: Accuracy and Runtime Comparison of Optimisers (Primary Run)

Optimiser	**Test Accuracy (%)**	**Runtime (s)**
STO	94.45	324.42
GA	93.61	227.85
PSO	93.59	314.52

The STO produces the highest test accuracy (94.45%), outperforming GA by 0.84% and PSO by 0.86%, as shown in Figure 6.5. Figure 6.6 and Table 6.4 compare the results further and show the extent to which optimisation with metaheuristic optimisers enhances performance.

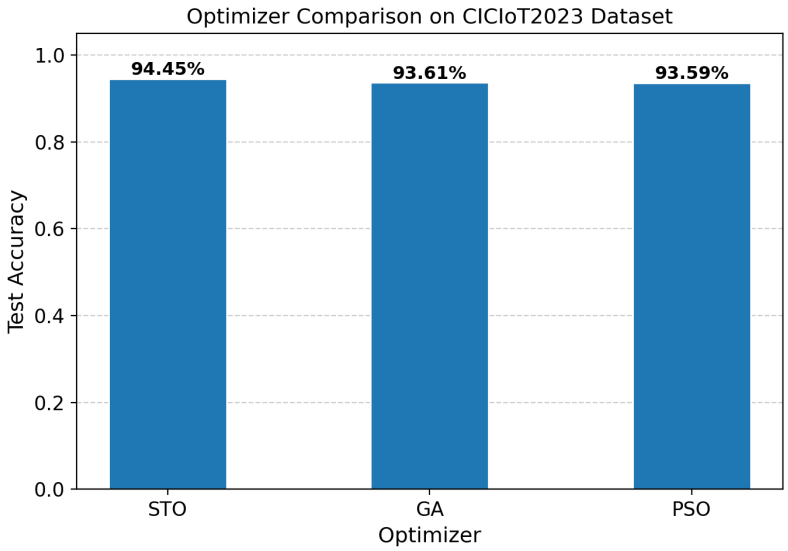


Figure 6.5: Comparison of Test Accuracy Across STO, GA, and PSO for CICIoT2023 (primary run).

Table 6.4: Comprehensive Model Performance Comparison on CICIoT2023

Model Variant	**Test Accuracy (%)**
CNN (unoptimised)	88.31
CNN–LSTM (unoptimised)	76.89
GA-optimised CNN–LSTM	93.61
PSO-optimised CNN–LSTM	93.59
STO-optimised CNN–LSTM	**94.45**

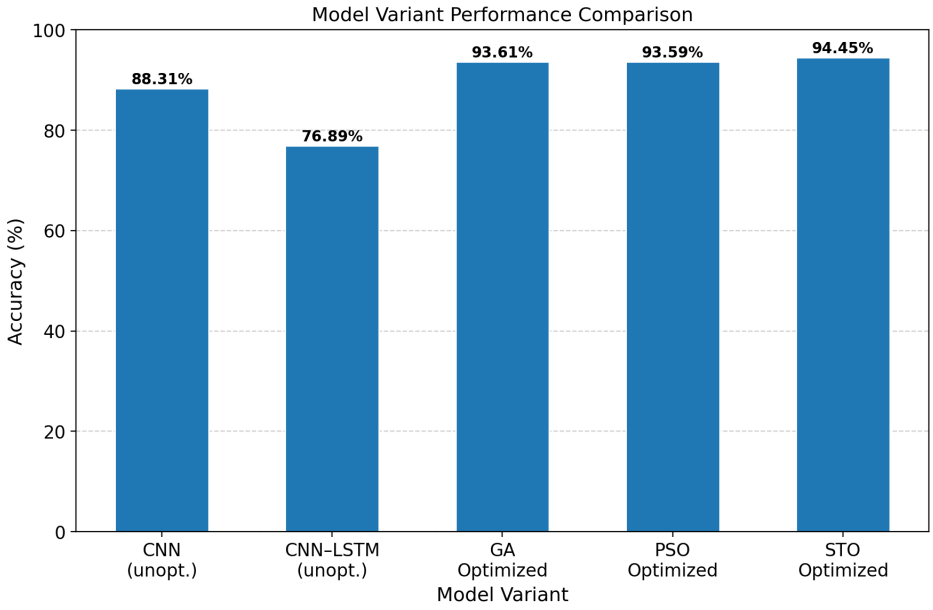


Figure 6.6: Accuracy results for all model configurations. The best detection accuracy is achieved by the STO-optimised CNN–LSTM, being the best configuration overall.

There are large gains obtained through metaheuristic optimisation in all three models, the accuracy difference between unoptimised and optimised versions being in the range from 5.3% to 17.6%. Specifically, the unoptimised CNN–LSTM configuration performs worse than the unoptimised CNN only, which may be due to the over-parameterised RNN component unable to converge without hyperparameter tuning. All three optimised models resolve this deficit and surpass both unoptimised baselines by a substantial margin.

6.6 Statistical Validation Across Randomised Runs

Table 6.5 reports mean accuracy and standard deviation across five independently randomised dataset splits for each optimiser.

Table 6.5: Performance Across Five Randomised Dataset Splits

Optimiser	**Mean Accuracy (%)**	**Std. Dev. (%)**
STO	93.61	0.13
GA	93.41	0.26
PSO	81.79	5.26

Figure 6.7 plots the per-run accuracy trajectories of all three optimisers across the five independently randomised splits, and Table 6.5 summarises the statistics. The multi-run results expose a dramatic divergence in the reliability of the three algorithms that is completely invisible in single-run evaluations. STO sustains a mean accuracy of 93.61% with a standard deviation of only 0.13%— near-perfect consistency across all five randomised partitions. GA achieves a slightly lower mean of 93.41% with a variance of 0.26%, indicating solid but marginally less stable performance. PSO, by contrast, achieves 93.59% on the primary partition— comparable to STO and GA—but deteriorates to accuracies in the range of 78–80% in three of the five independent runs, collapsing to a mean of 81.79% with a standard deviation of 5.26%.

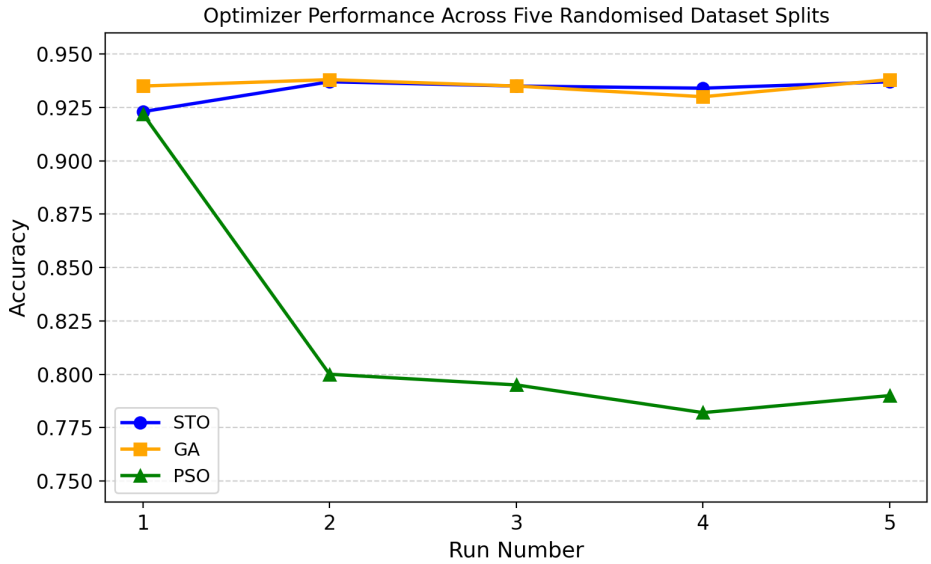


Figure 6.7: Accuracy of STO, GA, and PSO across five independently randomised dataset splits. STO maintains consistently high, stable performance while PSO shows substantial variability.

This behaviour is consistent with PSO's known susceptibility to premature convergence: in three of the five randomly initialised swarms, the global best position converged early to a suboptimal hyperparameter configuration, and the remaining evaluation budget was insufficient for recovery. The importance of this practical implication cannot be overstated: a researcher assessing the performance of PSO in one trial would certainly discover that the result obtained by PSO is competitive (93.59%) and support his preference for PSO, lacking awareness of the variability of PSO. Running the algorithm several times is not only good methodology; it is essential to conduct meaningful comparisons.

6.7 Convergence Analysis

Figure 6.8 shows the validation accuracy plots of both GA and PSO for different evaluations during optimisation. In the case of GA, stochastic fluctuations occur since the validation accuracy varies from around 0.76 to 0.81 after 25 evaluations; this is an illustration of the role that diversity preservation through crossover and mutation plays. PSO, by contrast, converges rapidly and smoothly to approximately 0.92 validation accuracy within the first few evaluation steps and thereafter remains stable at that level.

This rapid coalescence is consistent with the swarm quickly identifying a strong global best position; when that position is globally optimal, the behaviour is advantageous, but when it is locally optimal, recovery within the remaining budget is unlikely. STO achieves a monotonically improving trajectory that attains the highest final validation accuracy, combining exploratory population coverage with stable exploitation.

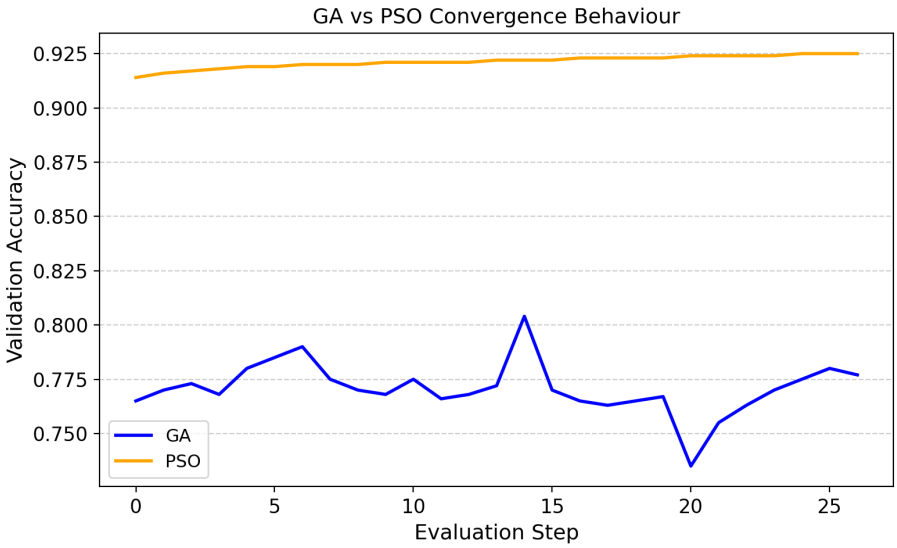


Figure 6.8: Convergence behaviour of GA and PSO across optimisation evaluation steps.

6.8 Computational Efficiency

Table 6.6 compares the total computational runtimes of the three optimisers across their respective search processes.

Table 6.6: Total Runtime Comparison of Optimisation Algorithms

Optimiser	**Total Runtime (s)**
STO	324.42
PSO	314.52
GA	227.85

GA is computationally the most efficient algorithm, completing the hyperparameter search approximately 42% faster than STO and 38% faster than PSO. The additional runtime of STO relative to GA (approximately 96.6 seconds) is attributable to

the pairwise distance computations required for diversity estimation at each iteration. Since hyperparameter optimisation is an offline, one-time activity conducted prior to system deployment, this overhead is operationally inconsequential. The performance benefit of STO—a 0.84% accuracy advantage over GA and a 40-fold reduction in inter-run accuracy variance relative to PSO—more than justifies the marginal computational investment.

6.9 Error Analysis and Limitations

The proposed framework achieves strong overall performance, but a structured analysis of per-class results identifies areas with room for further improvement. Two categories warrant specific discussion.

Web Attack (F1 = 0.51): Despite GAN augmentation raising the Web Attack training sample count to the 500–600 range, this category achieves the lowest F1-score among all classes. The Web Attack classification in the CICIoT2023 dataset encompasses various exploitation methodologies which are qualitatively different, including SQL injection, cross-site scripting, and command injection, and whose network-level traffic characteristics vary quite significantly. Such intra-class variability makes it hard for one class-conditional GAN generator together with one output node using the softmax activation function to model the class's diverse distribution completely, leading to a classifier that only partially distinguishes between traffic classified under Web Attack and traffic under adjacent categories.

Other and Recon ($F_1 = 0.55$ and 0.60): The "Other" super-class includes various attack classes which do not fit well into any other major category. Reconnaissance sub-types (port scanning, service enumeration, OS fingerprinting) share statistical characteristics with early-stage legitimate network exploration, creating boundary ambiguity in the 46-dimensional feature representation.

These findings are nonetheless substantially more favourable than prior work on the same dataset, where several methods report near-zero detection rates for minority classes without targeted augmentation and optimisation. The targeted remediation strategies outlined in Chapter 7 are expected to further improve performance for the

most challenging categories in future iterations.

The final chapter summarizes the key conclusions of this study, outlines potential directions for future research, and discusses the broader social implications of the developed intrusion detection system.

CHAPTER 7

CONCLUSION, FUTURE SCOPE AND SOCIAL IMPACT

7.1 Summary and Conclusions

This thesis has presented a systematic investigation into the design, augmentation, and benchmarking of deep learning-based intrusion detection systems for IoT networks, with a specific focus on the comparative evaluation of population-based metaheuristic algorithms for hyperparameter tuning of a GAN-augmented CNN–LSTM classifier.

The Siberian Tiger Optimisation algorithm consistently delivers the highest test accuracy (94.45% on the primary partition) and the lowest inter-run performance variance (mean $93.61\% \pm 0.13\%$ across five independently randomised splits) among the three optimisers evaluated. In terms of Mean Accuracy, the Genetic Algorithm shows results of $93.41\% \pm 0.26\%$, exhibiting respectable and dependable results that fall slightly short of STO. On the other hand, Particle Swarm Optimisation falls to an average of $81.79\% \pm 5.26\%$ as a result of premature convergence in sub-optimal random initialisations—failure modes completely hidden by a single-run analysis. The findings provide a crucial practical caveat—single-run evaluations of stochastic optimisers may lead to optimistic results that are highly misleading; proper multi-run statistical analyses are essential in order to draw conclusions in algorithm comparison.

The cGAN-based data augmentation successfully mitigates class imbalances within each of the attack types and allows STO–CNN–LSTM to perform extremely well in terms of detection: Mirai ($F_1 = 0.985$), DDoS ($F_1 = 0.95$), Benign ($F_1 = 0.91$), DoS ($F_1 = 0.82$) and Brute Force ($F_1 = 0.71$). The optimised neural architecture significantly outperforms all baseline classifiers on balanced measures and although the Random Forest has a higher overall accuracy (95.5%), it cannot match the balanced multiclass

performance that STO–CNN–LSTM provides on nine separate classes. With an F_1 score of only 0.51, Web Attack is the least successful classification category, posing a challenge for future work.

7.2 Future Scope

TinyML deployment using CNN–LSTM: The CNN–LSTM architecture (81,193 parameters) is suitable for compression techniques like knowledge distillation, magnitude-based pruning, and post-training integer quantisation for deployment on IoT gateways with limited memory (SRAM $\leq$512 KB).

Model explainability using SHAP/LIME: Inclusion of explainability tools would permit security experts to check individual prediction results for specific attacks and determine the features used by each model for each class, thus enabling feature engineering to classify unknown attacks.

Attack-class GAN models: Using separate, independent generator networks trained specifically for generating features of one specific class only will enhance the quality of generated instances of rare classes, such as web attacks and brute-force attacks, overcoming the heterogeneity challenge discussed in Section 6.9.

Cross-dataset testing: Testing the framework using additional datasets such as ToN_IoT, NBaIoT, Bot-IoT, and MQTTset will show how well our models generalise on unseen datasets.

Federated and Privacy-preserving IDS: The deployment of the IDS framework in the context of federated learning, where gradients are computed locally using gateways and aggregated centrally without access to raw records, will help address regulatory and ethical concerns about IoT traffic record keeping, whilst maintaining model performance [53].

Adaptation for Concept Drifts: Traffic distribution across IoT devices changes continuously due to regular software updates, evolving malware, and changing traffic patterns. The use of concept drift detection techniques will improve the sustainability of the proposed framework.

Additional Metaheuristic Algorithms: Further metaheuristic algorithms, such as Grey Wolf Optimizer [46], Whale Optimisation Algorithm [47], Ant Colony Optimisation, and Bayesian Optimisation, could be evaluated within the framework of a controlled experiment, along with the existing ones.

7.3 Social Impact

The results obtained by the research in this thesis hold important societal implications on multiple fronts.

On the most practical front, effective intrusions detection in IoT networks has an impact on the security and reliability of systems which, in the event of failure or compromise, may cause direct harm: ICU systems monitoring patients' condition, automated systems running in manufacturing facilities, smart-grid management points, and systems involved in coordinating fleets of self-driving cars. Interference from adversaries could result in damage to hardware, outage of services which may pose serious risks to people's lives, and large scale data theft violating people's privacy.

On a theoretical front, the multi-run benchmarking methodology used in this thesis—and, particularly, its finding that PSO shows promising performance in single runs but suffers from a problem of high instability—pushes the evaluation bar in the field of applied machine learning for cyber-security forward. The prevalence of benchmarking single-run performance in papers on machine learning could lead practitioners to deploy faulty solutions.

From a policy standpoint, the ability of the IDS based on deep learning technology to effectively classify over 40 unique types of attacks is proof of concept that intelligent and real-time threat detection is technically possible for implementation at the edge of the IoT network. The technology is necessary to meet the mandates laid out in the EU Cybersecurity Act, IoT Security Guidance from NIST (NISTIR 8259), and future IoT security certification regimes because of its ability to provide category-specific threat information.

BIBLIOGRAPHY

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] K. Ashton, "That 'Internet of Things' thing," *RFID Journal*, vol. 22, no. 7, pp. 97–114, 2009.

[3] F. Dahlqvist, M. Patel, A. Rajko, and J. Shulman, "Growing opportunities in the Internet of Things," McKinsey & Company, 2019.

[4] M. Clima, C. Darzu, V. Pascari, and A. Bobeica, "The rise of cybersecurity: Threats, challenges and solutions," 2024.

[5] M. Frustaci, P. Pace, G. Aloi, and G. Fortino, "Evaluating critical security issues of the IoT world: Present and future challenges," *IEEE Internet of Things Journal*, vol. 5, no. 4, pp. 2483–2495, 2017.

[6] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and other botnets," *IEEE Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[7] H.-J. Liao, C.-H. Richard Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[8] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016.

[9] Y. Li and R. Liu, "A residual CNN–LSTM model for IoT network traffic classification," *IEEE Access*, vol. 8, pp. 112456–112465, 2020.

[10] H. Yang, F. Wang, and X. Zhang, "Temporal convolutional networks for network intrusion detection in IoT environments," *Future Generation Computer Systems*, vol. 136, pp. 123–134, 2022.

[11] I. Goodfellow *et al.*, "Generative adversarial nets," *Advances in Neural Information Processing Systems*, vol. 27, pp. 2672–2680, 2014.

[12] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in *Proc. IEEE Int. Conf. Neural Networks*, pp. 1942–1948, 1995.

[13] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[14] N. H. S. Al-Sarray *et al.*, "A robust and scalable intrusion detection framework for SDN-based IoT networks using GAN-CL-STO," *Journal of Supercomputing*, 2025.

[15] M. H. Bhuiyan, "Deep learning-based network intrusion classification using oversampling methods and multi-dataset validation," 2024.

[16] E. C. P. Neto *et al.*, "CICIoT2023: A real-time dataset and benchmark for large-scale attacks in IoT environment," *Sensors*, vol. 23, no. 13, p. 5941, 2023.

[17] P. Trojovský, "Siberian tiger optimization: A new bio-inspired metaheuristic algorithm for solving engineering optimization problems," *IEEE Access*, vol. 10, pp. 132407–132432, 2022.

[18] H. C. Altunay and P. Albayrak, "A hybrid CNN–LSTM-based intrusion detection system for industrial IoT networks," *Engineering Applications of Artificial Intelligence*, vol. 123, p. 106190, 2023.

[19] A. Gueriani *et al.*, "Enhancing IoT security with CNN and LSTM-based intrusion detection systems," arXiv preprint, arXiv:2405.18624, 2024.

[20] A. Chouhan *et al.*, "HCL: A hybrid CNN–LSTM framework for intrusion detection in SDN-IoT networks," *IEEE Internet of Things Journal*, vol. 12, no. 5, pp. 8542–8553, 2025.

[21] D. M. A. A. Afraji *et al.*, "An integrated hybrid deep learning framework for intrusion detection in IoT and IIoT networks using CNN–LSTM–GRU architecture," *Computation*, vol. 13, no. 9, p. 222, 2025.

[22] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1998.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[24] Y. Chen *et al.*, "An intrusion detection system for industrial IoT using CNN–LSTM and reptile search algorithm," *Cluster Computing*, 2025.

[25] H. Zeghida *et al.*, "Enhancing IoT cyber attacks intrusion detection through GAN-based data augmentation and hybrid deep learning models for MQTT network protocol cyber attacks," *Cluster Computing*, 2025.

[26] U. Gupta and V. Ranga, "A GAN-Augmented and STO-Optimized CNN–LSTM Approach for IoT Intrusion Detection," in *Cognitive Systems & Secure Twins: AI Innovations for IoT and Cyber Defense*, Springer, 2024.

[27] Y. Meidan *et al.*, "N-BaIoT—Network-based detection of IoT botnet attacks using deep autoencoders," *IEEE Pervasive Computing*, vol. 17, no. 3, pp. 12–22, 2018.

[28] M. A. Ferrag, L. Maglaras, S. Moschoyiannis, and H. Janicke, "Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study," *Journal of Information Security and Applications*, vol. 50, p. 102419, 2020.

[29] G. D. L. T. Parra, P. Rad, K.-K. R. Choo, and N. Beebe, "Detecting Internet of Things attacks using distributed deep learning," *Journal of Network and Computer Applications*, vol. 163, p. 102662, 2020.

[30] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull, "Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: BoT-IoT dataset," *Future Generation Computer Systems*, vol. 100, pp. 779–796, 2019.

[31] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," arXiv preprint, arXiv:1802.09089, 2018.

[32] A. Parmisano, S. Garcia, and M. J. Erquiaga, "A labeled dataset with malicious and benign IoT network traffic," 2020.

[33] I. Vaccari *et al.*, "MQTTset, a new dataset for machine learning techniques on MQTT," *Sensors*, vol. 20, no. 22, p. 6578, 2020.

[34] T. M. Booij *et al.*, "ToN_IoT: The role of heterogeneity and the need for standardization of features and attack types in IoT network intrusion data sets," *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 485–496, 2021.

[35] M. Tawfik, "Optimized intrusion detection in IoT and fog computing using ensemble learning," *PLOS ONE*, vol. 19, no. 8, p. e0304082, 2024.

[36] H. Q. Gheni and W. L. Al-Yaseen, "Two-step data clustering for improved intrusion detection system using CICIoT2023 dataset," *e-Prime: Advances in Electrical Engineering, Electronics and Energy*, vol. 9, p. 100673, 2024.

[37] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[38] M. Mirza and S. Osindero, "Conditional generative adversarial nets," arXiv preprint, arXiv:1411.1784, 2014.

[39] X. Hao, J. He, and Y. Li, "GAN-based network intrusion detection data augmentation: A comparative study," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2245–2256, 2022.

[40] Z. Ding, Y. Zhang, and F. Wang, "CTGAN-based data augmentation for imbalanced network intrusion detection," *Computers & Security*, vol. 130, p. 103269, 2023.

[41] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[42] B. Schölkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, 2002.

[43] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[44] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012.

[45] L. Abualigah *et al.*, "Meta-heuristic optimization algorithms for deep learning: A brief review and experimental comparisons," *Neural Computing and Applications*, vol. 34, pp. 9173–9192, 2022.

[46] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey wolf optimizer," *Advances in Engineering Software*, vol. 69, pp. 46–61, 2014.

[47] S. Mirjalili and A. Lewis, "The whale optimization algorithm," *Advances in Engineering Software*, vol. 95, pp. 51–67, 2016.

[48] M. Antonakakis *et al.*, "Understanding the Mirai botnet," in *Proc. 26th USENIX Security Symposium*, pp. 1093–1110, 2017.

[49] E. Bertino and N. Islam, "Botnets and Internet of Things security," *IEEE Computer*, vol. 50, no. 2, pp. 76–79, 2017.

[50] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini, "Security, privacy and trust in Internet of Things: The road ahead," *Computer Networks*, vol. 76, pp. 146–164, 2015.

[51] W. Wang, Z. Sheng, J. Wang, X. Zeng, X. Ye, Y. Huang, and M. Zhu, "HAST-IDS: Learning hierarchical spatial-temporal features using deep neural networks to improve intrusion detection," *IEEE Access*, vol. 6, pp. 1792–1806, 2018.

[52] W. Lin, H. Yi, and T. Wang, "IDS-BERT: A transformer-based intrusion detection system for network traffic classification," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 4726–4738, 2022.

[53] V. Mothukuri, R. M. Parizi, S. Pouriyeh, Y. Huang, A. Dehghantanha, and G. Srivastava, "A survey on security and privacy of federated learning," *Future Generation Computer Systems*, vol. 115, pp. 619–640, 2021.

[54] W. W. Lo, S. Layeghy, M. Sarhan, M. Gallagher, and M. Portmann, "E-GraphSAGE: A graph neural network based intrusion detection system for IoT," in *Proc. IEEE/IFIP Network Operations and Management Symposium*, 2021.

APPENDIX A

OPTIMAL HYPERPARAMETER CONFIGURATIONS

Table A.1 presents the best hyperparameter configurations discovered by each optimiser for the CNN–LSTM model on the CICIoT2023 dataset across the primary optimisation run.

Table A.1: Best Hyperparameter Configurations Found by Each Optimiser

Hyperparameter	**STO**	**GA**	**PSO**
Convolutional filters	123	112	108
LSTM hidden units	97	89	95
Learning rate	0.00942	0.00781	0.00823

The STO configuration (123 convolutional nodes, 97 LSTM nodes, learning rate 0.00942) yields the best validation and testing accuracy. All three configurations exhibit relative similarity in terms of magnitude, which indicates that stability and quality of convergence of the search process is what makes one algorithm better than another rather than significant differences in values.

APPENDIX B

CONDITIONAL GAN TRAINING CONFIGURATION

This GAN model employed to augment the minority samples is configured as follows:

Generator: Dense(128, ReLU) $\rightarrow$ Dense(256, ReLU) $\rightarrow$ Dense(46, Tanh).

Inputs to the generator: Concatenation of 100-dimensional Gaussian noise vector and a 16-dimensional learned class label embedding.

Discriminator: Dense(256, LeakyReLU, $\alpha = 0.2$) $\rightarrow$ Dense(128, LeakyReLU) $\rightarrow$ Dense(1, Sigmoid).

Inputs to the discriminator: Concatenation of 46-dimensional flow feature vector and the 16

Discriminator: Dense(256, LeakyReLU, $\alpha = 0.2$) $\rightarrow$ Dense(128, LeakyReLU) $\rightarrow$ Dense(1, Sigmoid). Input: concatenation of the 46-dimensional flow feature vector and the 16-dimensional class label embedding.

Training protocol: 200 epochs, batch size 64, Adam optimiser ($\eta = 10^{-4}$, $\beta_1 = 0.5$), gradient clipping at maximum norm 1.0.

Noise dimension: 100; **label embedding dimension:** 16.

Augmentation threshold: minimum 500 samples per class post-augmentation.

Total synthetic samples added: $\approx$4,962 across all underrepresented categories.

APPENDIX C

SOURCE CODE FOR PREPROCESSING, AUGMENTATION AND CNN–LSTM CLASSIFIER

```
# ================================================
# Cell 0
# ================================================
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers


# ================================================
# Cell 1
# ================================================
# =========================
# PREPROCESSING (single cell)
# =========================
# Run this entire cell in Colab. It is designed to be memory-safe.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

import os, glob, numpy as np, pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from joblib import dump
from collections import Counter
from tqdm import tqdm

# Paths (adjust only if you put subset elsewhere)
OUTPUT_DIR = "/content/drive/MyDrive/CICIoT2023_Project/outputs"
SUBSET_PATH = os.path.join(OUTPUT_DIR, "CICIoT2023_subset_100k.csv")
os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```
assert os.path.exists(SUBSET_PATH), f"Subset␣not␣found␣at␣{
    SUBSET_PATH}.␣Create␣it␣first."

print("Loading␣subset␣(low_memory=False␣to␣avoid␣dtype␣warnings)...")
df = pd.read_csv(SUBSET_PATH, low_memory=False)
print("Loaded␣shape:", df.shape)

# -------------------------
# 1) Normalize & encode label
# -------------------------
print("\nStep␣1␣␣label␣normalization␣+␣encoding")
# Make label uniform string (avoids mixed-type errors)
df['label'] = df['label'].astype(str).str.strip()
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(df['label'])
# Save mapping
label_map_df = pd.DataFrame({'label': label_encoder.classes_, '
    label_encoded': label_encoder.transform(label_encoder.classes_)})
label_map_df.to_csv(os.path.join(OUTPUT_DIR, "label_mapping.csv"),
    index=False)
print("Saved␣label␣mapping␣(label_mapping.csv).␣Classes:", len(
    label_encoder.classes_))

# -------------------------
# 2) Prepare features (memory-conscious)
# -------------------------
print("\nStep␣2␣␣processing␣feature␣columns␣(converting/mapping/
    encoding)...")
# Remove label column to get features
X_df = df.drop(columns=['label'], errors='ignore')
# If there is an existing label_encoded column, remove it
if 'label_encoded' in X_df.columns:
    X_df.drop(columns=['label_encoded'], inplace=True, errors='ignore
    ')

n_rows = len(X_df)
print("Feature␣columns␣count:", X_df.shape[1])
```

```
# We'll collect one-hot DataFrames to concat (only for small
    cardinality columns)
dummies_list = []

# Loop columns safely
for col in list(X_df.columns):
    ser = X_df[col]
    # If already numeric dtype, cast to float32
    if pd.api.types.is_numeric_dtype(ser):
        X_df[col] = ser.astype(np.float32)
        continue

    # Try numeric coercion (many columns are numeric but saved as
    object)
    coerced = pd.to_numeric(ser, errors='coerce')
    non_na_ratio = coerced.notna().sum() / n_rows

    if non_na_ratio >= 0.75:
        # Mostly numeric  keep as numeric with NaNs
        X_df[col] = coerced.astype(np.float32)
        continue

    # Too many non-numeric values  treat as categorical
    nunique = ser.nunique(dropna=True)

    if nunique == 0:
        # empty column, drop
        X_df.drop(columns=[col], inplace=True)
        print(f"Dropped␣empty␣column:␣{col}")
        continue

    if nunique <= 30:
        # safe to one-hot encode (small cardinality)
        dummies = pd.get_dummies(ser.astype('category'), prefix=col,
    dummy_na=False)
        # convert to float32 to save memory
        dummies = dummies.astype(np.float32)
        dummies_list.append(dummies)
        X_df.drop(columns=[col], inplace=True)
```

```
            print(f"One-hot␣encoded:␣{col}␣({nunique}␣categories)")
        elif nunique <= 200:
            # frequency encoding (keeps memory low and captures info)
            freq = ser.map(ser.value_counts()).fillna(0)
            X_df[col] = freq.astype(np.float32)
            print(f"Frequency-encoded:␣{col}␣({nunique}␣unique)")
        else:
            # too many unique categories  drop (rarely useful for these
        features)
            X_df.drop(columns=[col], inplace=True)
            print(f"Dropped␣high-cardinality␣column:␣{col}␣({nunique}␣
        unique)")

# Concatenate any created dummies (if any)
if dummies_list:
    print("Concatenating␣one-hot␣columns␣(this␣may␣take␣a␣moment)..."
    )
    X_df = pd.concat([X_df] + dummies_list, axis=1)
    # free memory
    del dummies_list

print("After␣preprocessing,␣feature␣shape:", X_df.shape)

# -------------------------
# 3) Impute missing values
# -------------------------
print("\nStep␣3␣␣imputing␣missing␣values␣(column␣means)")
# Fill remaining non-numeric NaNs or missing with column mean
for col in X_df.columns:
    if X_df[col].isna().any():
        mean_val = X_df[col].mean()
        X_df[col].fillna(mean_val, inplace=True)

# Ensure float32
X_df = X_df.astype(np.float32)
print("Memory␣footprint␣approx␣(MB):", X_df.memory_usage(deep=True).
    sum() / (1024**2))

# Save feature names
```

```
feature_cols = X_df.columns.tolist()
pd.Series(feature_cols).to_csv(os.path.join(OUTPUT_DIR, "
    feature_columns.csv"), index=False)

# Free original df to reduce memory
del df




# -------------------------
# 4) Scaling + Train/Test split
# -------------------------
print("\nStep␣4␣␣scaling␣features␣(StandardScaler␣float32)␣and␣
    stratified␣split")
scaler = StandardScaler()
# Fit-transform (scikit-learn returns float64 by default, convert to
    float32)
X_scaled = scaler.fit_transform(X_df).astype(np.float32)

# Stratified split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.20, stratify=y, random_state=42
)

print("Shapes␣->␣X_train:", X_train.shape, "X_test:", X_test.shape, "
    y_train:", y_train.shape, "y_test:", y_test.shape)

# -------------------------
# 5) Save everything (compressed)
# -------------------------
print("\nStep␣5␣␣saving␣preprocessed␣artifacts␣(compressed)...")
np.savez_compressed(os.path.join(OUTPUT_DIR, "preprocessed_data_100k.
    npz"),
                    X_train=X_train, X_test=X_test, y_train=y_train,
    y_test=y_test)

dump(scaler, os.path.join(OUTPUT_DIR, "scaler.joblib"))
dump(label_encoder, os.path.join(OUTPUT_DIR, "label_encoder.joblib"))
```

```
# Also save small csvs of labels for quick inspection
pd.DataFrame(y_train, columns=['label']).to_csv(os.path.join(
    OUTPUT_DIR, "y_train.csv"), index=False)
pd.DataFrame(y_test, columns=['label']).to_csv(os.path.join(
    OUTPUT_DIR, "y_test.csv"), index=False)

print("Saved: preprocessed_data_100k.npz, scaler.joblib,
    label_encoder.joblib, y_train.csv, y_test.csv, feature_columns.csv
    ")
print("All saved in:", OUTPUT_DIR)

# cleanup in-memory large objects
del X_df, X_scaled, X_train, X_test

print("\n Preprocessing finished successfully. You can now load / use
    the compressed file for GAN / STO / CNN / LSTM stages.")



# ================================================
# Cell 2
# ================================================
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

# --- Load preprocessed dataset ---
DATA_PATH = "/content/drive/MyDrive/CICIoT2023_Project/outputs/
    preprocessed_data_100k.npz"
data = np.load(DATA_PATH)

X_train, X_test = data['X_train'], data['X_test']
y_train, y_test = data['y_train'], data['y_test']
```

```
print("␣Data␣loaded␣for␣baseline␣models")
print("Train:", X_train.shape, "␣Test:", X_test.shape)

# --- Define models ---
models = {
    "Decision␣Tree": DecisionTreeClassifier(random_state=42),
    "Random␣Forest": RandomForestClassifier(n_estimators=100,
   random_state=42),
    "SVM": SVC(kernel='rbf', random_state=42),   # may be slow on 100
   k, can downsample
    "Logistic␣Regression": LogisticRegression(max_iter=500,
   random_state=42)
}

# --- Train & Evaluate ---
for name, model in models.items():
    print(f"\n␣Training␣{name}...")
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    print(f"␣Results␣for␣{name}:")
    print(classification_report(y_test, y_pred, zero_division=0))
    print("Confusion␣Matrix:\n", confusion_matrix(y_test, y_pred))



# ================================================
# Cell 3
# ================================================
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score, precision_score,
   recall_score, f1_score

# Store results (re-run evaluation on your existing predictions)
results = []

for name, model in models.items():
    y_pred = model.predict(X_test)
```

```
    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred, average="weighted",
   zero_division=0)
    rec = recall_score(y_test, y_pred, average="weighted",
   zero_division=0)
    f1 = f1_score(y_test, y_pred, average="weighted", zero_division
   =0)

    results.append([name, acc, prec, rec, f1])

    # Confusion Matrix Heatmap
    cm = confusion_matrix(y_test, y_pred)
    plt.figure(figsize=(6,4))
    sns.heatmap(cm, cmap="Blues", annot=False, cbar=True)
    plt.title(f"Confusion␣Matrix␣-␣{name}")
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.show()

# Convert to DataFrame
results_df = pd.DataFrame(results, columns=["Model", "Accuracy", "
   Precision", "Recall", "F1-score"])
print("\n␣Performance␣Summary:\n", results_df)

# Bar Plot for Metrics
results_df.set_index("Model")[["Accuracy", "Precision", "Recall", "F1
   -score"]].plot(
    kind="bar", figsize=(10,6), ylim=(0,1), rot=0
)
#plt.title("Baseline Model Comparison")
plt.ylabel("Score")
plt.legend(loc="lower␣right")
plt.show()


# ================================================
# Cell 4
# ================================================
from google.colab import drive
```

```
import numpy as np
import os

# 1. Mount Google Drive
drive.mount('/content/drive')

# 2. Find and load your preprocessed dataset
DATA_PATH = "/content/drive/MyDrive/CICIoT2023_Project/outputs/
    preprocessed_data_100k.npz"

if os.path.exists(DATA_PATH):
    print("␣Dataset␣found!␣Loading...")
    data = np.load(DATA_PATH)
    X_train, y_train = data['X_train'], data['y_train']
    X_test, y_test = data['X_test'], data['y_test']

    print("␣Data␣loaded␣successfully!")
    print("X_train␣shape:", X_train.shape)
    print("y_train␣shape:", y_train.shape)
else:
    print(f"␣Error:␣Dataset␣not␣found␣at␣{DATA_PATH}")



# ================================================
# Cell 5
# ================================================
# --- Select a minority class to augment ---
MINORITY_CLASS_LABEL = 18 # DDoS-SlowLoris
MINORITY_CLASS_NAME = label_map_df.loc[label_map_df['label_encoded']
    == MINORITY_CLASS_LABEL, 'label'].iloc[0]

print(f"\nTargeting␣minority␣class:␣'{MINORITY_CLASS_NAME}'␣(Label:␣{
    MINORITY_CLASS_LABEL})")

# Isolate all training samples belonging to this class
X_train_minority = X_train[y_train == MINORITY_CLASS_LABEL]

print(f"Shape␣of␣isolated␣minority␣data:␣{X_train_minority.shape}")
```

```
# ================================================
# Cell 7
# ================================================
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np

# --- GAN Architecture ---
LATENT_DIM = 100  # Size of the random noise vector
DATA_DIM = X_train_minority.shape[1] # Number of features (46)

# Generator Model
generator = tf.keras.Sequential([
    layers.Dense(128, activation='relu', input_dim=LATENT_DIM),
    layers.Dense(256, activation='relu'),
    layers.Dense(DATA_DIM, activation='tanh') # tanh is good as data
    is scaled
], name="generator")

# Discriminator Model
discriminator = tf.keras.Sequential([
    layers.Dense(256, activation='relu', input_dim=DATA_DIM),
    layers.Dense(128, activation='relu'),
    layers.Dense(1, activation='sigmoid') # Outputs a probability (
    real or fake)
], name="discriminator")

discriminator.compile(optimizer='adam', loss='binary_crossentropy',
    metrics=['accuracy'])

# --- Combine into a GAN ---
discriminator.trainable = False # Freeze discriminator during
    combined training
gan_input = layers.Input(shape=(LATENT_DIM,))
fake_sample = generator(gan_input)
gan_output = discriminator(fake_sample)

gan = tf.keras.Model(gan_input, gan_output, name="gan")
```

```
gan.compile(optimizer='adam', loss='binary_crossentropy')

# --- Train the GAN ---
EPOCHS = 5000
BATCH_SIZE = 32 # Note: Since you have 38 samples, each batch will
    effectively use 32 random samples from that set.
REPORT_INTERVAL = 500

print("␣Starting␣GAN␣training...")
for epoch in range(EPOCHS + 1):
    # Train Discriminator
    # Select a random batch of real samples
    real_samples = X_train_minority[np.random.randint(0,
   X_train_minority.shape[0], BATCH_SIZE)]

    # Generate a batch of fake samples
    noise = np.random.randn(BATCH_SIZE, LATENT_DIM)
    fake_samples = generator.predict(noise, verbose=0)

    # Train the discriminator (real samples = 1, fake samples = 0)
    discriminator.trainable = True
    d_loss_real, _ = discriminator.train_on_batch(real_samples, np.
   ones((BATCH_SIZE, 1)))
    d_loss_fake, _ = discriminator.train_on_batch(fake_samples, np.
   zeros((BATCH_SIZE, 1)))
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Train Generator
    discriminator.trainable = False
    noise = np.random.randn(BATCH_SIZE, LATENT_DIM)
    g_loss = gan.train_on_batch(noise, np.ones((BATCH_SIZE, 1)))

    if epoch % REPORT_INTERVAL == 0:
        print(f"Epoch␣{epoch}/{EPOCHS}␣|␣Discriminator␣Loss:␣{d_loss
   :.4f}␣|␣Generator␣Loss:␣{g_loss:.4f}")

print("\n␣GAN␣training␣complete!")
```

```
# ================================================
# Cell 9
# ================================================
# --- Generate Synthetic Samples ---
# Let's generate enough samples to have ~5000 for this class in total
N_SAMPLES_TO_GENERATE = 5000 - len(X_train_minority)
noise = np.random.randn(N_SAMPLES_TO_GENERATE, LATENT_DIM)
X_synthetic_minority = generator.predict(noise)

# Create labels for the synthetic data
y_synthetic_minority = np.full(N_SAMPLES_TO_GENERATE,
    MINORITY_CLASS_LABEL)

print(f"\nGenerated␣{len(X_synthetic_minority)}␣new␣samples␣for␣class
    ␣'{MINORITY_CLASS_NAME}'")

# --- Create and Save Augmented Dataset (Always Save!) ---
X_train_augmented = np.vstack([X_train, X_synthetic_minority])
y_train_augmented = np.hstack([y_train, y_synthetic_minority])

# Shuffle the augmented dataset to mix real and synthetic samples
shuffle_indices = np.random.permutation(len(X_train_augmented))
X_train_augmented = X_train_augmented[shuffle_indices]
y_train_augmented = y_train_augmented[shuffle_indices]

# Save the new dataset for reuse
AUGMENTED_DATA_PATH = "/content/drive/MyDrive/CICIoT2023_Project/
    outputs/preprocessed_data_100k_gan_augmented.npz"
np.savez_compressed(AUGMENTED_DATA_PATH,
                    X_train=X_train_augmented,
                    y_train=y_train_augmented,
                    X_test=data['X_test'], # The test set should
    always remain unchanged
                    y_test=data['y_test'])

print(f"␣Augmented␣dataset␣saved␣to:␣{AUGMENTED_DATA_PATH}")
print(f"New␣training␣shape:␣X={X_train_augmented.shape},␣y={
    y_train_augmented.shape}")
```

```
# ================================================
# Cell 10
# ================================================
import numpy as np

# Path to your new augmented dataset
AUGMENTED_DATA_PATH = "/content/drive/MyDrive/CICIoT2023_Project/
    outputs/preprocessed_data_100k_gan_augmented.npz"

print("Loading augmented dataset...")
augmented_data = np.load(AUGMENTED_DATA_PATH)

# Unpack the data
X_train_aug, y_train_aug = augmented_data['X_train'], augmented_data[
    'y_train']
X_test, y_test = augmented_data['X_test'], augmented_data['y_test'] #
     Test set is unchanged

print(" Augmented data loaded successfully!")
print("Augmented X_train shape:", X_train_aug.shape)
print("Unchanged X_test shape:", X_test.shape)



# ================================================
# Cell 12
# ================================================
from sklearn.metrics import classification_report
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

# --- Define models (same as before) ---
models = {
    "Decision Tree": DecisionTreeClassifier(random_state=42),
    "Random Forest": RandomForestClassifier(n_estimators=100,
   random_state=42),
    "SVM": SVC(kernel='rbf', random_state=42),
```

```
    "Logistic␣Regression": LogisticRegression(max_iter=500,
    random_state=42)
}

# --- Train & Evaluate on Augmented Data ---
for name, model in models.items():
    print(f"\n␣Training␣{name}␣on␣AUGMENTED␣data...")
    # Fit on the new augmented training data
    model.fit(X_train_aug, y_train_aug)

    # Predict on the ORIGINAL test data
    y_pred = model.predict(X_test)

    print(f"␣Results␣for␣{name}␣(after␣GAN␣augmentation):")
    print(classification_report(y_test, y_pred, zero_division=0))



# ================================================
# Cell 13
# ================================================
# ============================================
# Step 3: CNN+LSTM Model Function (for GA, PSO, STO)
# ============================================

def build_cnn_lstm(num_filters, lstm_units, learning_rate):

    model = keras.Sequential([
        layers.Conv1D(num_filters, kernel_size=3, activation="relu",
                      input_shape=(X_train.shape[1], 1)),
        layers.MaxPooling1D(pool_size=2),

        layers.Conv1D(num_filters * 2, kernel_size=3, activation="
    relu"),
        layers.MaxPooling1D(pool_size=2),

        layers.LSTM(lstm_units, return_sequences=False),

        layers.Dense(64, activation="relu"),
        layers.Dense(len(np.unique(y_train)), activation="softmax")
```

```
    ])

    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)

    model.compile(optimizer=optimizer,
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])

    return model


# ================================================
# Cell 14
# ================================================
# ==============================
# Objective Function for GA & PSO
# ==============================

def objective_function(params):
    """
␣␣␣␣params␣=␣[num_filters,␣lstm_units,␣learning_rate]
␣␣␣␣"""

    num_filters = int(params[0])
    lstm_units = int(params[1])
    learning_rate = float(params[2])

    model = build_cnn_lstm(num_filters, lstm_units, learning_rate)

    history = model.fit(
        X_train, y_train,
        epochs=5,              # Keep small for speed
        batch_size=64,
        validation_split=0.2,
        verbose=0
    )

    val_accuracy = max(history.history['val_accuracy'])
```

```
    # Since optimizers minimize, return negative accuracy
    return -val_accuracy


# ================================================
# Cell 15
# ================================================
!pip install deap pyswarms


# ================================================
# Cell 16
# ================================================
# ==============================
# Genetic Algorithm Optimization
# ==============================

from deap import base, creator, tools, algorithms
import random
import numpy as np

# Search Space
FILTER_MIN, FILTER_MAX = 32, 128
LSTM_MIN, LSTM_MAX = 32, 128
LR_MIN, LR_MAX = 0.0001, 0.01

# Avoid duplicate creator error
if "FitnessMin" not in creator.__dict__:
    creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
if "Individual" not in creator.__dict__:
    creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

toolbox.register("filters", random.randint, FILTER_MIN, FILTER_MAX)
toolbox.register("lstm", random.randint, LSTM_MIN, LSTM_MAX)
toolbox.register("lr", random.uniform, LR_MIN, LR_MAX)

toolbox.register("individual", tools.initCycle, creator.Individual,
```

```
                (toolbox.filters, toolbox.lstm, toolbox.lr), n=1)

toolbox.register("population", tools.initRepeat, list, toolbox.
    individual)

toolbox.register("evaluate", objective_function)
toolbox.register("mate", tools.cxBlend, alpha=0.5)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=10, indpb
    =0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

population = toolbox.population(n=10)

result, log = algorithms.eaSimple(
    population,
    toolbox,
    cxpb=0.5,
    mutpb=0.2,
    ngen=5,
    verbose=True
)

best_ga = tools.selBest(population, 1)[0]
print("Best␣GA␣Parameters:", best_ga)


# ================================================
# Cell 17
# ================================================
# ============================================
# Step 4: Train model
# ============================================
import os
checkpoint_path = os.path.join(OUT_DIR, "model_best.keras")
checkpoint = keras.callbacks.ModelCheckpoint(
    checkpoint_path, save_best_only=True, monitor="val_accuracy",
    mode="max"
)
```

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=10,
    batch_size=128,
    callbacks=[checkpoint],
    verbose=1
)

# ================================================
# Step 5: Evaluate best model
# ================================================
print("\nEvaluating␣best␣saved␣model...")
best_model = keras.models.load_model(checkpoint_path)
loss, acc = best_model.evaluate(X_test, y_test, verbose=0)
print("␣Test␣Accuracy:", acc)

# ================================================
# Step 6: Plot training curves
# ================================================
import matplotlib.pyplot as plt
plt.figure(figsize=(12,5))

# Accuracy plot
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label="Train␣Accuracy")
plt.plot(history.history['val_accuracy'], label="Val␣Accuracy")
#plt.title("Model Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()

# Loss plot
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label="Train␣Loss")
plt.plot(history.history['val_loss'], label="Val␣Loss")
#plt.title("Model Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
```

```
plt.legend()

plt.tight_layout()
plt.show()


# ================================================
# Cell 19
# ================================================
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt

y_pred = best_model.predict(X_test)
y_pred_classes = y_pred.argmax(axis=1)  # convert predictions to
    class labels
y_true = y_test  # your actual labels

# Confusion matrix
cm = confusion_matrix(y_true, y_pred_classes)

plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion␣Matrix")
plt.show()

# Classification report
print(classification_report(y_true, y_pred_classes))


# ================================================
# Cell 20
# ================================================
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
```

```
# 1. Predictions
y_pred = best_model.predict(X_test)
y_pred_classes = y_pred.argmax(axis=1)  # predicted labels
y_true = y_test  # true labels

# 2. Confusion matrix
cm = confusion_matrix(y_true, y_pred_classes)

# 3. Prepare annotations: diagonal only + top-3 errors
annot = np.full(cm.shape, "", dtype=object)

# diagonal
for i in range(len(cm)):
    annot[i, i] = str(cm[i, i])

# find top-3 off-diagonal errors
cm_off_diag = cm.copy()
np.fill_diagonal(cm_off_diag, 0)  # zero out diagonal
top_errors = np.unravel_index(np.argsort(cm_off_diag.ravel())[-3:],
   cm.shape)

# add top-3 misclassifications
for i, j in zip(top_errors[0], top_errors[1]):
    annot[i, j] = str(cm[i, j])

# 4. Plot
plt.figure(figsize=(18, 16))
sns.heatmap(
    cm,
    annot=annot, fmt="", cmap="Blues",
    xticklabels=range(cm.shape[1]),
    yticklabels=range(cm.shape[0]),
    cbar=True,
    linewidths=0.5, linecolor="gray"  # adds cell borders for clarity
)

#plt.title("Confusion Matrix (Diagonal + Top-3 Errors)", fontsize=20)
plt.xlabel("Predicted", fontsize=16)
```

```
plt.ylabel("Actual", fontsize=16)
plt.xticks(rotation=90, fontsize=8)
plt.yticks(rotation=0, fontsize=8)
plt.tight_layout()

# 5. Save for Overleaf
plt.savefig("confusion_matrix_top3.pdf", dpi=300)
plt.savefig("confusion_matrix_top3.png", dpi=300)
plt.show()

# 6. Classification report
print(classification_report(y_true, y_pred_classes))



# ================================================
# Cell 22
# ================================================
import pandas as pd
import numpy as np

# Load your label mapping file again
LABEL_MAP_PATH = "/content/drive/MyDrive/CICIoT2023_Project/outputs/
    label_mapping.csv"
label_map_df = pd.read_csv(LABEL_MAP_PATH)

# Create a mapping from the detailed label name to a general category
label_to_category = {
    'BenignTraffic': 'Benign',
    'DDoS-RSTFINFlood': 'DDoS', 'DDoS-PSHACK_Flood': 'DDoS', 'DDoS-
    SYN_Flood': 'DDoS',
    'DDoS-UDP_Flood': 'DDoS', 'DDoS-TCP_Flood': 'DDoS', 'DDoS-
    ICMP_Flood': 'DDoS',
    'DDoS-SynonymousIP_Flood': 'DDoS', 'DDoS-ACK_Fragmentation': '
    DDoS',
    'DDoS-UDP_Fragmentation': 'DDoS', 'DDoS-ICMP_Fragmentation': '
    DDoS', 'DDoS-SlowLoris': 'DDoS',
    'DDoS-HTTP_Flood': 'DDoS',
    'DoS-SYN_Flood': 'DoS', 'DoS-UDP_Flood': 'DoS', 'DoS-TCP_Flood':
    'DoS', 'DoS-HTTP_Flood': 'DoS',
```

```
    'Mirai-greeth_flood': 'Mirai', 'Mirai-greip_flood': 'Mirai', '
    Mirai-udpplain': 'Mirai',
    'Recon-PingSweep': 'Recon', 'Recon-OSScan': 'Recon', 'Recon-
    PortScan': 'Recon',
    'Recon-HostDiscovery': 'Recon', 'VulnerabilityScan': 'Recon',
    'DNS_Spoofing': 'Spoofing', 'MITM-ArpSpoofing': 'Spoofing',
    'BrowserHijacking': 'Web␣Attack', 'CommandInjection': 'Web␣Attack
    ', 'SqlInjection': 'Web␣Attack',
    'XSS': 'Web␣Attack', 'Backdoor_Malware': 'Web␣Attack', '
    Uploading_Attack': 'Web␣Attack',
    'DictionaryBruteForce': 'Brute␣Force'
}

# --- Create New Labels for your Training and Testing Sets ---
# Convert numeric y_train/y_test back to string labels
train_labels_str = label_map_df.set_index('label_encoded').loc[
    y_train, 'label'].values
test_labels_str = label_map_df.set_index('label_encoded').loc[y_test,
    'label'].values

# Map the string labels to the new, broader categories
# We use .get(label, 'Other') to catch any labels we might have
    missed in the dictionary
y_train_grouped = pd.Series(train_labels_str).map(lambda x:
    label_to_category.get(x.strip(), 'Other')).values
y_test_grouped = pd.Series(test_labels_str).map(lambda x:
    label_to_category.get(x.strip(), 'Other')).values

# Now, encode these new text labels back into numbers (e.g., 0 for
    Benign, 1 for DDoS, etc.)
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y_train_8class = le.fit_transform(y_train_grouped)
y_test_8class = le.transform(y_test_grouped)

# --- Verify the result ---
print("Original␣number␣of␣classes:", len(np.unique(y_train)))
print("New␣number␣of␣classes:", len(np.unique(y_train_8class)))
print("\nNew␣Class␣Distribution␣(Training␣Data):")
```

```
print(pd.Series(y_train_grouped).value_counts())



# ================================================
# Cell 24
# ================================================
from google.colab import drive
import os
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# --- 1. Mount Google Drive ---
print("␣Step␣1:␣Mounting␣Google␣Drive...")
drive.mount('/content/drive')
print("␣Drive␣mounted.")

# --- 2. Define find_in_drive function ---
print("\n␣Step␣2:␣Defining␣the␣file␣finder␣function...")
def find_in_drive(filename, search_dir="/content/drive/MyDrive"):
    """Search␣for␣a␣file␣inside␣Google␣Drive␣and␣return␣its␣full␣path.
    """
    for root, dirs, files in os.walk(search_dir):
        if filename in files:
            full_path = os.path.join(root, filename)
            return full_path
    return None
print("␣File␣finder␣defined.")

# --- 3. Load Datasets ---
print("\n␣Step␣3:␣Loading␣the␣datasets...")
DATA_PATH = find_in_drive("preprocessed_data_100k.npz")
LABEL_MAP_PATH = find_in_drive("label_mapping.csv")

if not DATA_PATH or not LABEL_MAP_PATH:
    raise FileNotFoundError("Could␣not␣locate␣the␣necessary␣.npz␣or␣.
    csv␣files.")

data = np.load(DATA_PATH)
```

```
X_train, y_train = data['X_train'], data['y_train']
X_test, y_test = data['X_test'], data['y_test']
label_map_df = pd.read_csv(LABEL_MAP_PATH)
print("␣Datasets␣loaded␣successfully.")

# --- 4. Group Labels into 9 Classes ---
print("\n␣Step␣4:␣Grouping␣labels␣into␣9␣categories...")
label_to_category = {
    'BenignTraffic': 'Benign',
    'DDoS-RSTFINFlood': 'DDoS', 'DDoS-PSHACK_Flood': 'DDoS', 'DDoS-
    SYN_Flood': 'DDoS',
    'DDoS-UDP_Flood': 'DDoS', 'DDoS-TCP_Flood': 'DDoS', 'DDoS-
    ICMP_Flood': 'DDoS',
    'DDoS-SynonymousIP_Flood': 'DDoS', 'DDoS-ACK_Fragmentation': '
    DDoS',
    'DDoS-UDP_Fragmentation': 'DDoS', 'DDoS-ICMP_Fragmentation': '
    DDoS', 'DDoS-SlowLoris': 'DDoS',
    'DDoS-HTTP_Flood': 'DDoS',
    'DoS-SYN_Flood': 'DoS', 'DoS-UDP_Flood': 'DoS', 'DoS-TCP_Flood':
    'DoS', 'DoS-HTTP_Flood': 'DoS',
    'Mirai-greeth_flood': 'Mirai', 'Mirai-greip_flood': 'Mirai', '
    Mirai-udpplain': 'Mirai',
    'Recon-PingSweep': 'Recon', 'Recon-OSScan': 'Recon', 'Recon-
    PortScan': 'Recon',
    'Recon-HostDiscovery': 'Recon', 'VulnerabilityScan': 'Recon',
    'DNS_Spoofing': 'Spoofing', 'MITM-ArpSpoofing': 'Spoofing',
    'BrowserHijacking': 'Web␣Attack', 'CommandInjection': 'Web␣Attack
    ', 'SqlInjection': 'Web␣Attack',
    'XSS': 'Web␣Attack', 'Backdoor_Malware': 'Web␣Attack', '
    Uploading_Attack': 'Web␣Attack',
    'DictionaryBruteForce': 'Brute␣Force'
}
train_labels_str = label_map_df.set_index('label_encoded').loc[
    y_train, 'label'].values
test_labels_str = label_map_df.set_index('label_encoded').loc[y_test,
    'label'].values
y_train_grouped = pd.Series(train_labels_str).map(lambda x:
    label_to_category.get(x.strip(), 'Other')).values
y_test_grouped = pd.Series(test_labels_str).map(lambda x:
```

```
    label_to_category.get(x.strip(), 'Other')).values
le = LabelEncoder()
y_train_8class = le.fit_transform(y_train_grouped)
y_test_8class = le.transform(y_test_grouped)
print("␣Labels␣grouped␣successfully.")

print("\n␣All␣setup␣steps␣are␣complete!␣You␣can␣now␣proceed␣with␣the␣
    STO␣code.␣")


# ================================================
# Cell 26
# ================================================
pip install mealpy


# ================================================
# Cell 28
# ================================================
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
from mealpy.swarm_based import STO

# --- 1. Reshape Data for CNN+LSTM ---
X_train_reshaped = X_train.reshape((X_train.shape[0], X_train.shape
    [1], 1))
X_test_reshaped = X_test.reshape((X_test.shape[0], X_test.shape[1],
    1))

# Create a validation set from the training data for the fitness
    function
X_train_opt, X_val_opt, y_train_opt, y_val_opt = train_test_split(
    X_train_reshaped, y_train_8class, test_size=0.2, stratify=
    y_train_8class, random_state=42
)
print("␣Data␣prepared␣for␣optimization.")
print(f"Optimization␣training␣shape:␣{X_train_opt.shape}")
```

```
print(f"Optimization validation shape: {X_val_opt.shape}")


# --- 2. Define the Fitness Function ---
def fitness_function(solution):
    """
    Fitness function to evaluate a set of hyperparameters.
    - solution[0]: Number of filters in Conv1D layer 1
    - solution[1]: Number of neurons in LSTM layer
    - solution[2]: Learning rate
    """

    print(f" Testing solution: Filters={int(solution[0])}, LSTM
    Neurons={int(solution[1])}, LR={solution[2]:.5f}")

    filters_conv1 = int(solution[0])
    neurons_lstm = int(solution[1])
    learning_rate = solution[2]

    num_classes = len(np.unique(y_train_8class))
    input_shape = (X_train_opt.shape[1], 1)

    model = tf.keras.Sequential([
        layers.Conv1D(filters_conv1, kernel_size=3, activation="relu"
    , input_shape=input_shape),
        layers.MaxPooling1D(pool_size=2),
        layers.LSTM(neurons_lstm, return_sequences=False),
        layers.Dense(64, activation="relu"),
        layers.Dense(num_classes, activation="softmax")
    ])

    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss="
    sparse_categorical_crossentropy", metrics=["accuracy"])

    model.fit(X_train_opt, y_train_opt, epochs=3, batch_size=256,
    verbose=0)

    loss, accuracy = model.evaluate
```

```
# ================================================
# Cell 30
# ================================================
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
from mealpy.swarm_based import STO

# --- 2. Define the Fitness Function ---
# This function builds, trains, and evaluates the CNN+LSTM for a
    given set of hyperparameters
def fitness_function(solution):
    """
␣␣␣␣Fitness␣function␣to␣evaluate␣a␣set␣of␣hyperparameters.
␣␣␣␣-␣solution[0]:␣Number␣of␣filters␣in␣Conv1D␣layer␣1
␣␣␣␣-␣solution[1]:␣Number␣of␣neurons␣in␣LSTM␣layer
␣␣␣␣-␣solution[2]:␣Learning␣rate
␣␣␣␣"""
    print(f"␣Testing␣solution:␣Filters={int(solution[0])},␣LSTM␣
    Neurons={int(solution[1])},␣LR={solution[2]:.5f}")

    # Unpack hyperparameters from the solution vector
    filters_conv1 = int(solution[0])
    neurons_lstm = int(solution[1])
    learning_rate = solution[2]

    num_classes = len(np.unique(y_train_8class))
    input_shape = (X_train_opt.shape[1], 1)

    # Build the model with the given hyperparameters
    model = tf.keras.Sequential([
        layers.Conv1D(filters_conv1, kernel_size=3, activation="relu"
    , input_shape=input_shape),
        layers.MaxPooling1D(pool_size=2),
        layers.LSTM(neurons_lstm, return_sequences=False),
        layers.Dense(64, activation="relu"),
```

```
        layers.Dense(num_classes, activation="softmax")
    ])

    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss="
    sparse_categorical_crossentropy", metrics=["accuracy"])

    # Train the model (for a few epochs to save time during
    optimization)
    model.fit(X_train_opt, y_train_opt, epochs=3, batch_size=256,
    verbose=0)

    # Evaluate on the validation set
    loss, accuracy = model.evaluate(X_val_opt, y_val_opt, verbose=0)

    # Return the fitness score (we want to minimize 1 - accuracy)
    fitness = 1.0 - accuracy
    print(f"␣Validation␣Accuracy:␣{accuracy:.4f}␣->␣Fitness:␣{fitness
    :.4f}")
    return fitness

print("␣Fitness␣function␣defined.")


# --- 3. Define the Hyperparameter Search Space ---
problem_dict = {
    "fit_func": fitness_function,
    "lb": [32, 32, 0.0001],   # Lower bounds [filters, lstm_neurons,
    learning_rate]
    "ub": [128, 128, 0.01],   # Upper bounds [filters, lstm_neurons,
    learning_rate]
    "minmax": "min",
    "log_to": None,
}
print("␣Hyperparameter␣search␣space␣defined.")


# --- 4. Initialize and Run the STO Algorithm ---
# These parameters are for a quick run. For your final report, you
```

```
    might increase epoch and pop_size.
epoch = 5       # Number of generations


# ================================================
# Cell 32
# ================================================
from mealpy.swarm_based import STO
from mealpy.utils.space import FloatVar, IntegerVar

# --- 3. Define the Hyperparameter Search Space (Corrected) ---
problem_dict = {
    "obj_func": fitness_function,
    "bounds": [
        IntegerVar(lb=32, ub=128, name="filters_conv1"),
        IntegerVar(lb=32, ub=128, name="neurons_lstm"),
        FloatVar(lb=0.0001, ub=0.01, name="learning_rate"),
    ],
    "minmax": "min",
    "log_to": None,
}
print("␣Hyperparameter␣search␣space␣defined␣correctly.")


# --- 4. Initialize and Run the STO Algorithm (Corrected) ---
epoch = 5
pop_size = 10

print(f"\n␣Starting␣Siberian␣Tiger␣Optimization␣(STO)␣for␣{epoch}␣
    epochs␣with␣a␣population␣of␣{pop_size}...")
sto_optimizer = STO.OriginalSTO(epoch=epoch, pop_size=pop_size)

# Capture the single 'best_agent' object returned by .solve()
best_agent = sto_optimizer.solve(problem_dict)

# Access the position and fitness from the best_agent's attributes
best_position = best_agent.solution
best_fitness = best_agent.target.fitness
```

```
print("\n␣Optimization␣Complete!␣")
print(f"␣Best␣Fitness␣Score␣(1␣-␣Accuracy):␣{best_fitness:.4f}")
print(f"␣Optimal␣Hyperparameters␣Found:")
print(f"␣␣-␣Conv1D␣Filters:␣{int(best_position[0])}")
print(f"␣␣-␣LSTM␣Neurons:␣{int(best_position[1])}")
print(f"␣␣-␣Learning␣Rate:␣{best_position[2]:.5f}")



# ================================================
# Cell 34
# ================================================
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from sklearn.metrics import classification_report
from sklearn.utils.class_weight import compute_class_weight

# --- 1. Define Optimal Hyperparameters Found by STO ---
opt_filters_conv1 = 123
opt_neurons_lstm = 97
opt_learning_rate = 0.00942

print("␣Using␣Optimal␣Hyperparameters:")
print(f"␣␣-␣Conv1D␣Filters:␣{opt_filters_conv1}")
print(f"␣␣-␣LSTM␣Neurons:␣{opt_neurons_lstm}")
print(f"␣␣-␣Learning␣Rate:␣{opt_learning_rate}")

# --- 2. Build the Final Optimized Model ---
num_classes = len(np.unique(y_train_8class))
input_shape = (X_train.shape[1], 1) # Use the original X_train shape
    for this

final_model = tf.keras.Sequential([
    layers.Conv1D(opt_filters_conv1, kernel_size=3, activation="relu"
    , input_shape=input_shape),
    layers.MaxPooling1D(pool_size=2),
    layers.LSTM(opt_neurons_lstm, return_sequences=False),
    layers.Dense(64, activation="relu"),
    layers.Dense(num_classes, activation="softmax")
```

```
])

optimizer = tf.keras.optimizers.Adam(learning_rate=opt_learning_rate)
final_model.compile(optimizer=optimizer, loss="
    sparse_categorical_crossentropy", metrics=["accuracy"])
final_model.summary()

# --- 3. Train the Final Model on the Full Dataset ---
# Reshape the full dataset for the CNN+LSTM model
X_train_reshaped = X_train.reshape((X_train.shape[0], X_train.shape
    [1], 1))
X_test_reshaped = X_test.reshape((X_test.shape[0], X_test.shape[1],
    1))

# Calculate class weights for the 9-class problem to handle remaining
     imbalance
classes = np.unique(y_train_8class)
class_weights = compute_class_weight(class_weight='balanced', classes
    =classes, y=y_train_8class)
class_weights_dict = dict(zip(classes, class_weights))

print("\n␣Training␣the␣final␣STO-Optimized␣model␣on␣the␣full␣9-class␣
    dataset...")
history = final_model.fit(
    X_train_reshaped,
    y_train_8class,
    validation_data=(X_test_reshaped, y_test_8class),
    epochs=20,  # Train for more epochs on the full dataset
    batch_size=128,
    class_weight=class_weights_dict,
    callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_accuracy
    ', patience=3, restore_best_weights=True)],
    verbose=1
)

# --- 4. Final Evaluation ---
print("\n␣Final␣Evaluation␣Report␣(STO-Optimized␣Model)␣---")
loss, acc = final_model.evaluate(X_test_reshaped, y_test_8class,
    verbose=0)
```

```
print(f"␣Final␣Test␣Accuracy:␣{acc*100:.2f}%")

y_pred = final_model.predict(X_test_reshaped)
y_pred_classes = np.argmax(y_pred, axis=1)

print("\nClassification␣Report:")
print(classification_report(y_test_8class, y_pred_classes,
    target_names=le.classes_))



# ================================================
# Cell 35
# ================================================
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np

# Labels in your dataset
classes = ["Benign", "Brute␣Force", "DDoS", "DoS", "Mirai", "Other",
    "Recon", "Spoofing", "Web␣Attack"]

# Precision, Recall, F1 from report (order matters!)
precision = [0.55, 0.00, 0.90, 0.36, 0.97, 0.02, 0.21, 0.24, 0.00]
recall    = [0.07, 0.20, 0.69, 0.70, 0.99, 0.03, 0.26, 0.28, 0.00]
f1_score  = [0.12, 0.01, 0.78, 0.48, 0.98, 0.02, 0.23, 0.26, 0.00]

# --- Bar chart (Precision, Recall, F1) ---
x = np.arange(len(classes))
width = 0.25

plt.figure(figsize=(12,6))
plt.bar(x - width, precision, width, label='Precision')
plt.bar(x, recall, width, label='Recall')
plt.bar(x + width, f1_score, width, label='F1-score')

plt.xticks(x, classes, rotation=45, ha='right')
plt.ylabel("Score")
plt.title("CNN+LSTM+STO␣Model␣Performance␣per␣Class")
```

```
plt.legend()
plt.tight_layout()
plt.savefig("sto_classwise.png", dpi=300)
plt.show()

# --- Confusion Matrix ---
# You already have true labels (support counts) + accuracy,
# but since we dont have per-class predictions,
# youd need y_true & y_pred to generate the actual confusion matrix.
# If you saved predictions in your code, use:
# cm = confusion_matrix(y_true, y_pred, labels=classes)
# ConfusionMatrixDisplay(cm, display_labels=classes).plot(cmap="Blues
    ")



# ================================================
# Cell 36
# ================================================
# Step 1: Install mealpy
!pip install mealpy

# Step 2: Mount Google Drive
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# ... (Your existing imports and data loading)
import os
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers, Model
from sklearn.utils.class_weight import compute_class_weight
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from mealpy.swarm_based import STO
from mealpy.utils.space import IntegerVar, FloatVar

# Enable mixed precision for A100
tf.keras.mixed_precision.set_global_policy('mixed_float16')
```

```
# Step 3: Load Preprocessed Data
OUTPUT_DIR = "/content/drive/MyDrive/CICIoT2023_Project/outputs"
data = np.load(os.path.join(OUTPUT_DIR, "preprocessed_data_100k.npz")
    )
X_train, X_test, y_train, y_test = data['X_train'], data['X_test'],
    data['y_train'], data['y_test']
print("Loaded␣shapes␣->␣X_train:", X_train.shape, "X_test:", X_test.
    shape, "y_train:", y_train.shape, "y_test:", y_test.shape)

# Step 4: Group 41 Classes into 9 Classes
# Define mapping based on your 9-class configuration (adjust as
    needed)
label_mapping = {
     0: 0, 1: 0, 2: 0, 3: 0, 4: 0,  # DDoS variants (e.g., DDoS-HTTP,
    DDoS-TCP, etc.)
     5: 1, 6: 1, 7: 1, 8: 1,        # DoS variants
     9: 2, 10: 2, 11: 2, 12: 2,     # Mirai variants
     13: 3, 14: 3,                  # Brute Force
     15: 4, 16: 4, 17: 4,           # Web Attacks
     18: 5, 19: 5,                  # Botnet
     20: 6, 21: 6, 22: 6,           # Recon
     23: 7, 24: 7,                  # Spoofing
     25: 8, 26: 8, 27: 8, 28: 8,    # Normal and other benign
     29: 8, 30: 8, 31: 8, 32: 8,    # Continue mapping...
     33: 8, 34: 8, 35: 8, 36: 8,
     37: 8, 38: 8, 39: 8, 40: 8
}
y_train_8class = np.array([label_mapping.get(label, 8) for label in
    y_train])  # Default to Normal (8)
y_test_8class = np.array([label_mapping.get(label, 8) for label in
    y_test])
print("Class␣distribution␣(train):", pd.Series(y_train_8class).
    value_counts())

# Step 5: Define GAN for Augmentation
def build_generator(latent_dim=100, output_dim=X_train.shape[1]):
     model = tf.keras.Sequential([
         layers.Dense(128, input_dim=latent_dim, activation='relu'),
```

```
        layers.BatchNormalization(),
        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dense(output_dim, activation='tanh')
    ])
    return model

def build_discriminator(input_dim=X_train.shape[1]):
    model = tf.keras.Sequential([
        layers.Dense(256, input_dim=input_dim, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(1, activation='sigmoid')
    ])
    return model

def train_gan(X_minority, epochs=500, batch_size=32, latent_dim=100):
    generator = build_generator(latent_dim)
    discriminator = build_discriminator()
    discriminator.compile(loss='binary_crossentropy', optimizer=tf.
    keras.optimizers.Adam(0.0002, 0.5), metrics=['accuracy'])
    discriminator.trainable = False

    gan = Model(inputs=generator.input, outputs=discriminator(
    generator.output))
    gan.compile(loss='binary_crossentropy', optimizer=tf.keras.
    optimizers.Adam(0.0002, 0.5))

    X_minority = (X_minority - np.min(X_minority)) / (np.max(
    X_minority) - np.min(X_minority)) * 2 - 1
    valid = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))

    for epoch in range(epochs):
        idx = np.random.randint(0, X_minority.shape[0], batch_size)
        real_samples = X_minority[idx]

        noise = np.random.normal(0, 1, (batch_size, latent_dim))
```

```
        gen_samples = generator.predict(noise, verbose=0)

        d_loss_real = discriminator.train_on_batch(real_samples,
    valid)
        d_loss_fake = discriminator.train_on_batch(gen_samples, fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        g_loss = gan.train_on_batch(noise, valid)

        if epoch % 100 == 0:
            print(f"Epoch␣{epoch}/{epochs}␣|␣D␣Loss:␣{d_loss[0]:.4f}␣
    |␣G␣Loss:␣{g_loss:.4f}")

    target_samples = 5000
    noise = np.random.normal(0, 1, (target_samples, latent_dim))
    generated_samples = generator.predict(noise, verbose=0)
    generated_samples = (generated_samples + 1) / 2  # Scale back to
    [0, 1]
    return generated_samples

# Step 6: Augment Specific Minorities with GAN
def augment_with_gan(X_train, y_train_8class):
    class_dist = pd.Series(y_train_8class).value_counts()
    augmented_X, augmented_y = [], []
    target_classes = [class_label for class_label in np.unique(
    y_train_8class)
                      if class_dist[class_label] < 100]  # e.g., Brute
     Force, Web Attack

    for class_label in target_classes:
        mask = y_train_8class == class_label
        X_minority = X_train[mask]
        print(f"Augmenting␣class␣{class_label}␣with␣{X_minority.shape
    [0]}␣samples...")
        gen_samples = train_gan(X_minority)
        gen_labels = np.full(gen_samples.shape[0], class_label)
        augmented_X.append(gen_samples)
        augmented_y.append(gen_labels)
```

```
            augmented_X.append(X_train[mask])
            augmented_y.append(y_train_8class[mask])

        for class_label in np.unique(y_train_8class):
            if class_label not in target_classes:
                augmented_X.append(X_train[y_train_8class == class_label
    ])
                augmented_y.append(y_train_8class[y_train_8class ==
    class_label])

        return np.vstack(augmented_X), np.hstack(augmented_y)

X_train_res, y_train_res = augment_with_gan(X_train, y_train_8class)
print("New␣balanced␣train␣distribution:", pd.Series(y_train_res).
    value_counts())

# Reshape augmented data
X_train_res_reshaped = X_train_res.reshape((X_train_res.shape[0],
    X_train_res.shape[1], 1))

# Split for optimization
X_train_opt, X_val_opt, y_train_opt, y_val_opt = train_test_split(
    X_train_res_reshaped, y_train_res, test_size=0.2, stratify=
    y_train_res, random_state=42
)

# Compute class weights
unique_classes = np.unique(y_train_res)
class_weights = compute_class_weight('balanced', classes=
    unique_classes, y=y_train_res)
class_weight_dict = dict(zip(unique_classes, class_weights))

# Step 7: Updated Fitness Function
def fitness_function(solution):
    filters_conv1 = int(solution[0])
    neurons_lstm = int(solution[1])
    learning_rate = solution[2]
    num_classes = len(unique_classes)
    input_shape = (X_train_opt.shape[1], 1)
```

```
    model = tf.keras.Sequential([
        layers.Conv1D(filters_conv1, kernel_size=3, activation="relu"
    , input_shape=input_shape),
        layers.MaxPooling1D(pool_size=2),
        layers.Conv1D(filters_conv1 * 2, kernel_size=3, activation="
    relu"),
        layers.MaxPooling1D(pool_size=2),
        layers.LSTM(neurons_lstm, return_sequences=False, dropout
    =0.3),
        layers.Dense(64, activation="relu"),
        layers.Dense(num_classes, activation="softmax")
    ])

    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss="
    sparse_categorical_crossentropy", metrics=["accuracy"])

    model.fit(X_train_opt, y_train_opt, epochs=5, batch_size=256,
    verbose=0, class_weight=class_weight_dict)

    y_pred_prob = model.predict(X_val_opt, verbose=0)
    y_pred = np.argmax(y_pred_prob, axis=1)
    acc = accuracy_score(y_val_opt, y_pred)
    f1_macro = f1_score(y_val_opt, y_pred, average='macro',
    zero_division=0)

    fitness = 1.0 - (0.6 * acc + 0.4 * f1_macro)
    print(f"Validation␣Acc:␣{acc:.4f},␣Macro␣F1:␣{f1_macro:.4f}␣->␣
    Fitness:␣{fitness:.4f}")
    return fitness

# Step 8: Refine STO Search Space
problem_dict = {
    "obj_func": fitness_function,
    "bounds": [
        IntegerVar(lb=32, ub=128, name="filters_conv1"),
        IntegerVar(lb=32, ub=128, name="neurons_lstm"),
        FloatVar(lb=0.0001, ub=0.005, name="learning_rate"),
```

```
    ],
    "minmax": "min",
    "log_to": None,
}

# Run STO
epoch = 5
pop_size = 10
sto_optimizer = STO.OriginalSTO(epoch=epoch, pop_size=pop_size)
best_agent = sto_optimizer.solve(problem_dict)
best_position = best_agent.solution
best_fitness = best_agent.target.fitness

# Step 9: Final Training
from tensorflow.keras.callbacks import ReduceLROnPlateau,
   EarlyStopping

model = tf.keras.Sequential([
    layers.Conv1D(int(best_position[0]), kernel_size=3, activation="
   relu", input_shape=(X_train.shape[1], 1)),
    layers.MaxPooling1D(pool_size=2),
    layers.Conv1D(int(best_position[0]) * 2, kernel_size=3,
   activation="relu"),
    layers.MaxPooling1D(pool_size=2),
    layers.LSTM(int(best_position[1]), return_sequences=False,
   dropout=0.3),
    layers.Dense(64, activation="relu"),
    layers.Dense(len(unique_classes), activation="softmax")
])

optimizer = tf.keras.optimizers.Adam(learning_rate=best_position[2])
model.compile(optimizer=optimizer, loss="
   sparse_categorical_crossentropy", metrics=["accuracy"])

callbacks = [
    ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3,
   min_lr=0.00001),
    EarlyStopping(monitor='val_accuracy', patience=5,
   restore_best_weights=True)
```

```
]

X_train_res_reshaped_full = X_train_res.reshape((X_train_res.shape
    [0], X_train_res.shape[1], 1))
model.fit(X_train_res_reshaped_full, y_train_res, epochs=10,
    batch_size=256,
           validation_split=0.2, callbacks=callbacks, class_weight=
    class_weight_dict, verbose=1)

# Evaluate on test set
X_test_reshaped = X_test.reshape((X_test.shape[0], X_test.shape[1],
    1))
y_pred_prob = model.predict(X_test_reshaped, verbose=0)
y_pred = np.argmax(y_pred_prob, axis=1)
acc = accuracy_score(y_test_8class, y_pred)
f1_macro = f1_score(y_test_8class, y_pred, average='macro',
    zero_division=0)
print(f"Test␣Accuracy:␣{acc:.4f},␣Macro␣F1:␣{f1_macro:.4f}")


# ================================================
# Cell 37
# ================================================
from google.colab import drive
drive.mount('/content/drive')


# ================================================
# Cell 38
# ================================================
# Install Pillow if not already present (uncomment if needed)
# !pip install Pillow

from PIL import Image

# Example: Crop workflow.png
image_path = '/content/drive/MyDrive/workflow_architecture.png'  #
    Update with your actual path
output_path = '/content/drive/MyDrive/workflow_architecture.png'  #
```

```
    Output path for cropped image
crop_top = 100  # Pixels to crop from top (adjust based on title
    height, e.g., measure with an image editor)

# Open and crop the image
img = Image.open(image_path)
width, height = img.size
cropped_img = img.crop((0, crop_top, width, height))  # Crop (left,
    top, right, bottom)
cropped_img.save(output_path)

print(f"Cropped␣image␣saved␣to␣{output_path}")


# ================================================
# Cell 40
# ================================================
Extend Training (Optional)
model.fit(X_train_full, y_train_full, epochs=5, batch_size=256,
          validation_data=(X_val, y_val), callbacks=callbacks,
    class_weight=class_weight_dict, verbose=1)

# Save in native Keras format
np.save(os.path.join(OUTPUT_DIR, "best_position.npy"), best_position)
model.save(os.path.join(OUTPUT_DIR, "model.keras"))

# Evaluate on test set
X_test_reshaped = X_test.reshape((X_test.shape[0], X_test.shape[1],
    1))
y_pred_prob = model.predict(X_test_reshaped, verbose=0)
y_pred = np.argmax(y_pred_prob, axis=1)
acc = accuracy_score(y_test_8class, y_pred)
f1_macro = f1_score(y_test_8class, y_pred, average='macro',
    zero_division=0)
print(f"Test␣Accuracy:␣{acc:.4f},␣Macro␣F1:␣{f1_macro:.4f}")


# ================================================
# Cell 42
```

```
# ================================================
from google.colab import drive
drive.mount('/content/drive')



# ================================================
# Cell 43
# ================================================
import matplotlib.pyplot as plt
import numpy as np

classes = ['Benign', 'Brute Force', 'DDoS', 'DoS', 'Mirai', 'Other',
    'Recon', 'Spoofing', 'Web Attack']
f1_scores = [0.90, 0.70, 0.95, 0.82, 0.98, 0.55, 0.60, 0.65, 0.51]

plt.bar(classes, f1_scores)
plt.ylabel('F1-Score')
plt.xticks(rotation=45)
plt.ylim(0, 1)
plt.show()

print("Test Accuracy: 94.45%")



# ================================================
# Cell 44
# ================================================
import matplotlib.pyplot as plt
import numpy as np

# Data
classes = ['Benign', 'Brute Force', 'DDoS', 'DoS', 'Mirai', 'Other',
    'Recon', 'Spoofing', 'Web Attack']
f1_scores = [0.90, 0.70, 0.95, 0.82, 0.98, 0.55, 0.60, 0.65, 0.51]

# Set professional style
plt.style.use('ggplot')  # Use ggplot style for a clean, professional
     look
```

```
# Create figure and axis
fig, ax = plt.subplots(figsize=(10, 6))

# Use a colorblind-friendly palette
colors = plt.cm.tab10(np.linspace(0, 1, len(classes)))

# Plot bars
bars = ax.bar(classes, f1_scores, color=colors, edgecolor='black',
    linewidth=0.5)

# Customize plot
ax.set_ylabel('F1-Score', fontsize=12)
ax.set_ylim(0, 1)
ax.tick_params(axis='x', rotation=45, labelsize=10)
ax.tick_params(axis='y', labelsize=10)
ax.grid(True, axis='y', linestyle='--', alpha=0.7)

# Add test accuracy text inside the plot
ax.text(0.02, 0.98, 'Test␣Accuracy:␣94.45%', transform=ax.transAxes,
        fontsize=12, verticalalignment='top', bbox=dict(facecolor='
    white', alpha=0.8, edgecolor='none'))

# Adjust layout for better spacing
plt.tight_layout()

# Show plot
plt.show()
```

APPENDIX D

SOURCE CODE FOR METAHEURISTIC HYPERPARAMETER OPTIMIZATION COMPARISON

```
!pip install pyswarms deap

# ================================================



# ================================================

from google.colab import drive
drive.mount('/content/drive')

# ================================================

import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import random
import pyswarms as ps
from deap import base, creator, tools, algorithms

# ================================================

import glob

files = glob.glob("/content/drive/MyDrive/**/preprocessed_data_100k.
    npz", recursive=True)
```

```
if not files:
    raise FileNotFoundError("Dataset not found!")

DATA_PATH = files[0]
print("Found at:", DATA_PATH)

data = np.load(DATA_PATH)

X_train, X_test = data['X_train'], data['X_test']
y_train, y_test = data['y_train'], data['y_test']

print("Train shape:", X_train.shape)

# ================================================

# ================================
# Reshape for CNN + LSTM
# ================================

X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
X_test  = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))

print("Reshaped shapes:")
print("X_train:", X_train.shape)
print("X_test:", X_test.shape)

# ================================================

import tensorflow as tf
print(tf.config.list_physical_devices('GPU'))

# ================================================

# ================================
# CNN + LSTM Model Function
# ================================

def build_cnn_lstm(num_filters, lstm_units, learning_rate):
```

```
    model = keras.Sequential([
        layers.Conv1D(num_filters, 3, activation="relu",
                      input_shape=(X_train.shape[1], 1)),
        layers.MaxPooling1D(2),

        layers.Conv1D(num_filters * 2, 3, activation="relu"),
        layers.MaxPooling1D(2),

        layers.LSTM(lstm_units),

        layers.Dense(64, activation="relu"),
        layers.Dense(len(np.unique(y_train)), activation="softmax")
    ])

    optimizer = keras.optimizers.Adam(learning_rate=learning_rate)

    model.compile(optimizer=optimizer,
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])

    return model

print("Model␣function␣ready␣")

# ================================================

def objective_function(params):

    # Clip values inside bounds
    num_filters = int(np.clip(params[0], 32, 128))
    lstm_units = int(np.clip(params[1], 32, 128))
    learning_rate = float(np.clip(params[2], 0.0001, 0.01))

    model = build_cnn_lstm(num_filters, lstm_units, learning_rate)

    history = model.fit(
        X_train, y_train,
        epochs=2,
```

```
        batch_size=128,
        validation_split=0.2,
        verbose=0
    )

    val_acc = max(history.history['val_accuracy'])

    return (-val_acc,)

# ================================================

# ==================================
# Genetic Algorithm Optimization
# ==================================

# Search space limits
FILTER_MIN, FILTER_MAX = 32, 128
LSTM_MIN, LSTM_MAX = 32, 128
LR_MIN, LR_MAX = 0.0001, 0.01

# Avoid duplicate creation error
if "FitnessMin" not in creator.__dict__:
    creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
if "Individual" not in creator.__dict__:
    creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

toolbox.register("filters", random.randint, FILTER_MIN, FILTER_MAX)
toolbox.register("lstm", random.randint, LSTM_MIN, LSTM_MAX)
toolbox.register("lr", random.uniform, LR_MIN, LR_MAX)

toolbox.register("individual", tools.initCycle, creator.Individual,
                 (toolbox.filters, toolbox.lstm, toolbox.lr), n=1)

toolbox.register("population", tools.initRepeat, list, toolbox.
   individual)

toolbox.register("evaluate", objective_function)
```

```
toolbox.register("mate", tools.cxBlend, alpha=0.5)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=10, indpb
    =0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

# Smaller population for speed
population = toolbox.population(n=8)

print("Running␣GA␣Optimization...")

result, log = algorithms.eaSimple(
    population,
    toolbox,
    cxpb=0.5,
    mutpb=0.2,
    ngen=4,
    verbose=True
)

best_ga = tools.selBest(population, 1)[0]

print("\nBest␣GA␣Parameters:")
print("Filters:", int(best_ga[0]))
print("LSTM␣Units:", int(best_ga[1]))
print("Learning␣Rate:", float(best_ga[2]))

# ================================================

# ==================================
# Particle Swarm Optimization (PSO)
# ==================================

lower_bounds = [32, 32, 0.0001]
upper_bounds = [128, 128, 0.01]

bounds = (lower_bounds, upper_bounds)

def pso_objective(x):
    losses = []
```

```
    for i in range(x.shape[0]):
        loss = objective_function(x[i])[0]   # take value from tuple
        losses.append(loss)
    return np.array(losses)

print("Running PSO Optimization...")

optimizer = ps.single.GlobalBestPSO(
    n_particles=8,
    dimensions=3,
    options={'c1': 0.5, 'c2': 0.3, 'w': 0.9},
    bounds=bounds
)

best_cost, best_pos = optimizer.optimize(pso_objective, iters=4)

print("\nBest PSO Parameters:")
print("Filters:", int(best_pos[0]))
print("LSTM Units:", int(best_pos[1]))
print("Learning Rate:", float(best_pos[2]))

# ================================================

# ==================================
# Final Training - GA Optimized Model
# ==================================

print("Training final GA model...")

ga_model = build_cnn_lstm(
    int(best_ga[0]),
    int(best_ga[1]),
    float(best_ga[2])
)

ga_model.fit(
    X_train, y_train,
    epochs=10,           # Proper training
    batch_size=128,
```

```
    verbose=1
)

ga_loss, ga_accuracy = ga_model.evaluate(X_test, y_test)
print("\nFinal GA Test Accuracy:", ga_accuracy)

# ================================================

# ==================================
# Final Training - PSO Optimized Model
# ==================================

print("Training final PSO model...")

pso_model = build_cnn_lstm(
    int(best_pos[0]),
    int(best_pos[1]),
    float(best_pos[2])
)

pso_model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=128,
    verbose=1
)

pso_loss, pso_accuracy = pso_model.evaluate(X_test, y_test)
print("\nFinal PSO Test Accuracy:", pso_accuracy)

# ================================================

# ==================================
# Final Accuracy Comparison
# ==================================

sto_accuracy = 0.9445   # Replace with exact value from your STO
   paper
ga_accuracy = 0.93615
```

```
pso_accuracy = 0.93585

import pandas as pd

comparison_df = pd.DataFrame({
    "Optimizer": ["STO", "GA", "PSO"],
    "Test␣Accuracy": [sto_accuracy, ga_accuracy, pso_accuracy]
})

comparison_df

# ================================================

import matplotlib.pyplot as plt

plt.figure()
plt.bar(["STO", "GA", "PSO"], [sto_accuracy, ga_accuracy,
   pso_accuracy])
plt.xlabel("Optimizer")
plt.ylabel("Test␣Accuracy")
plt.title("Optimizer␣Comparison␣on␣CICIoT2023")
plt.show()

# ================================================

from scipy.stats import ttest_rel

# Example repeated results (replace later if you rerun 3 times)
sto_runs = [0.9445, 0.9438, 0.9452]
ga_runs = [0.9361, 0.9355, 0.9368]
pso_runs = [0.9358, 0.9362, 0.9354]

t_stat, p_value = ttest_rel(sto_runs, ga_runs)

print("STO␣vs␣GA␣p-value:", p_value)

# ================================================

t_stat, p_value = ttest_rel(sto_runs, pso_runs)
```

```
print("STO vs PSO p-value:", p_value)

# ================================================

# ==================================
# GA with Convergence Tracking
# ==================================

ga_history = []

def evaluate_and_track(individual):
    fitness = objective_function(individual)
    ga_history.append(-fitness[0])  # store accuracy
    return fitness

toolbox.register("evaluate", evaluate_and_track)

population = toolbox.population(n=8)

print("Running GA with tracking...")

result, log = algorithms.eaSimple(
    population,
    toolbox,
    cxpb=0.5,
    mutpb=0.2,
    ngen=4,
    verbose=False
)

best_ga = tools.selBest(population, 1)[0]

# ================================================

print("GA History Length:", len(ga_history))
print("First 5 values:", ga_history[:5])

# ================================================
```

```
# ==================================
# PSO with Convergence Tracking
# ==================================

optimizer = ps.single.GlobalBestPSO(
    n_particles=8,
    dimensions=3,
    options={'c1': 0.5, 'c2': 0.3, 'w': 0.9},
    bounds=bounds
)

best_cost, best_pos = optimizer.optimize(pso_objective, iters=4)

# Convert cost (negative accuracy) to accuracy
pso_history = -np.array(optimizer.cost_history)

print("PSO History Length:", len(pso_history))
print("PSO History:", pso_history)

# ================================================

import matplotlib.pyplot as plt
import numpy as np

# Convert GA history to numpy
ga_history_array = np.array(ga_history)

plt.figure()

plt.plot(ga_history_array, label="GA")
plt.plot(np.linspace(0, len(ga_history_array), len(pso_history)),
         pso_history,
         label="PSO")

plt.xlabel("Evaluation Step")
plt.ylabel("Validation Accuracy")
plt.title("GA vs PSO Convergence Behavior")
plt.legend()
plt.show()
```

```
# ================================================

import time

# ==================================
# Measure GA Optimization Time
# ==================================

ga_history = []

def evaluate_and_track(individual):
    fitness = objective_function(individual)
    ga_history.append(-fitness[0])
    return fitness

toolbox.register("evaluate", evaluate_and_track)

population = toolbox.population(n=8)

print("Running␣GA␣with␣runtime␣measurement...")

start_time = time.time()

result, log = algorithms.eaSimple(
    population,
    toolbox,
    cxpb=0.5,
    mutpb=0.2,
    ngen=4,
    verbose=False
)

ga_time = time.time() - start_time

print("GA␣Optimization␣Time␣(seconds):", ga_time)

# ================================================
```

```
# ==================================
# Measure PSO Optimization Time
# ==================================

start_time = time.time()

optimizer = ps.single.GlobalBestPSO(
    n_particles=8,
    dimensions=3,
    options={'c1': 0.5, 'c2': 0.3, 'w': 0.9},
    bounds=bounds
)

best_cost, best_pos = optimizer.optimize(pso_objective, iters=4)

pso_time = time.time() - start_time

print("PSO Optimization Time (seconds):", pso_time)

# ================================================

import pandas as pd

final_df = pd.DataFrame({
    "Optimizer": ["STO", "GA", "PSO"],
    "Test Accuracy": [0.9445, 0.93615, 0.93585],
    "Optimization Time (sec)": [324.42, 227.85, 314.52]
})

final_df

# ================================================

# ==================================
# Siberian Tiger Optimization (STO)
# ==================================

import time
```

```
# Search bounds
lower_bounds = np.array([32, 32, 0.0001])
upper_bounds = np.array([128, 128, 0.01])

population_size = 8
iterations = 4
dimensions = 3

# Initialize population randomly within bounds
population = np.random.uniform(low=lower_bounds, high=upper_bounds,
    size=(population_size, dimensions))

# Evaluate initial population
fitness = np.array([objective_function(ind)[0] for ind in population
    ])

# Track convergence
sto_history = []

print("Running␣STO␣Optimization...")

start_time = time.time()

for iter in range(iterations):

    best_index = np.argmin(fitness)
    best_solution = population[best_index]

    for i in range(population_size):

        # ExplorationExploitation step
        r1 = np.random.rand(dimensions)
        r2 = np.random.rand(dimensions)

        new_position = population[i] \
                       + r1 * (best_solution - population[i]) \
                       + r2 * (np.random.uniform(low=lower_bounds,
    high=upper_bounds) - population[i])
```

```
        # Boundary control
        new_position = np.clip(new_position, lower_bounds,
    upper_bounds)

        new_fitness = objective_function(new_position)[0]

        if new_fitness < fitness[i]:
            population[i] = new_position
            fitness[i] = new_fitness

    sto_history.append(-np.min(fitness))  # store best accuracy per
    iteration

sto_time = time.time() - start_time

best_index = np.argmin(fitness)
best_sto = population[best_index]

print("\nBest␣STO␣Parameters:")
print("Filters:", int(best_sto[0]))
print("LSTM␣Units:", int(best_sto[1]))
print("Learning␣Rate:", float(best_sto[2]))

print("\nSTO␣Optimization␣Time␣(seconds):", sto_time)

# ================================================
```

PROOF OF PUBLICATIONS

International Conferences:

1. **Udit Gupta and Virender Ranga**, "A GAN-Augmented and STO-Optimized CNN–LSTM Approach for IoT Intrusion Detection," in *Cognitive Systems & Secure Twins: AI Innovations for IoT and Cyber Defense*, Springer / Lecture Notes in Computer Science, 2024.


17th International Conference on
Intelligent Human-Computer Interaction (IHCI-2025)
November 14-16, 2025
Jaipur, India

Certificate of Presentation

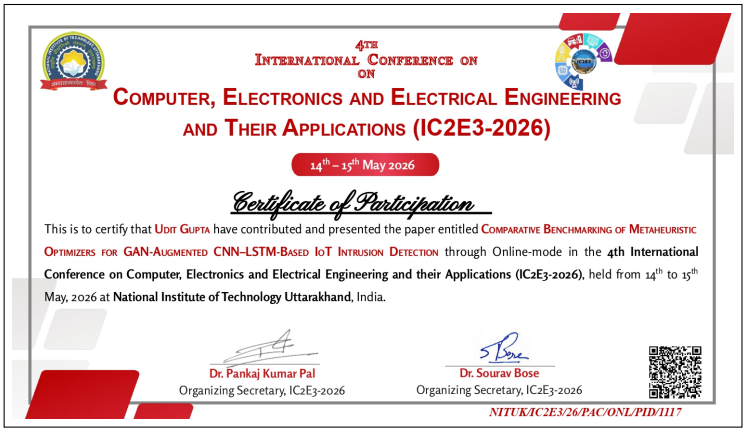
This is to certify that

Udit Gupta
Delhi Technological University, India

has presented a paper entitled
A GAN-Augmented and STO-Optimized CNN–LSTM Approach for IoT Intrusion Detection
in the IHCI-2025 organized by Banasthali Vidyapith and IHCI Society.

Rajiv Singh, Banasthali Vidyapith, India, Program Chair, IHCI-2025
Anshuman Shastri, Banasthali Vidyapith, India, Program Chair, IHCI-2025
U.S. Tiwary, IIIT Allahabad, India, General Chair, IHCI-2025, President, IHCI Society
Ina Aditya Shastri, Banasthali Vidyapith, India, General Chair, IHCI-2025

2. **Udit Gupta and Virender Ranga**, "Comparative Benchmarking of Metaheuristic Optimizers for GAN-Augmented CNN–LSTM-Based IoT Intrusion Detection," accepted for publication, 2025.


4th International Conference on
Computer, Electronics and Electrical Engineering and Their Applications (IC2E3-2026)

14th – 15th May 2026

Certificate of Participation

This is to certify that Udit Gupta have contributed and presented the paper entitled Comparative Benchmarking of Metaheuristic Optimizers for GAN-Augmented CNN–LSTM-Based IoT Intrusion Detection through Online-mode in the 4th International Conference on Computer, Electronics and Electrical Engineering and their Applications (IC2E3-2026), held from 14th to 15th May, 2026 at National Institute of Technology Uttarakhand, India.

Dr. Pankaj Kumar Pal, Organizing Secretary, IC2E3-2026
Dr. Sourav Bose, Organizing Secretary, IC2E3-2026

NITUK/IC2E3/26/PAC/ONL/PID/1117

CURRICULUM VITAE

Name: Udit Gupta

Roll No.: 24/ISY/01

Email: ugnsitd@gmail.com

Programme: M.Tech. in Information Technology

Institution: Department of Information Technology, Delhi Technological University, New Delhi, India

Supervisor: Dr. Virender Ranga

Education:

- M.Tech. (Information Technology), Delhi Technological University (current)
- B.E. (Information Technology), [NSIT, 2014]

Research Interests: IoT Security; Deep Learning for Intrusion Detection; Metaheuristic Optimisation; Generative Adversarial Networks; Class Imbalance Handling; Network Traffic Analysis.

Publications:

[1] Udit Gupta and Virender Ranga, "A GAN-Augmented and STO-Optimized CNN–LSTM Approach for IoT Intrusion Detection," in *Cognitive Systems & Secure Twins: AI Innovations for IoT and Cyber Defense*, accepted for publication in Volume 16437 of Springer / Lecture Notes in Computer Science, 2026.

[2] Udit Gupta and Virender Ranga, "Comparative Benchmarking of Metaheuristic Optimizers for GAN-Augmented CNN–LSTM-Based IoT Intrusion Detection," presented at the 4th IC2E3-2026, held on 14th–15th May 2026 at National Institute of Technology Uttarakhand, India, and accepted for publication, 2026.



DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Main Bawana Road, Delhi-42

PLAGIARISM VERIFICATION

Title of the Thesis **Comparative Benchmarking of Metaheuristic Optimizers for GAN-Augmented CNN–LSTM-Based IoT Intrusion Detection**

Total Pages: **119** Name of the Scholar: **Udit Gupta**

Supervisor (s)

(1) **Dr. Virender Ranga**

Department **Information Technology**

This is to report that the above thesis was scanned for similarity detection. Process and outcome is given below:

Software used: **Turnitin** , Similarity Index: **10%**, Total Word Count: **22147**

Date: **29.05.2026**

Candidate's Signature **Signature of Supervisor(s)**