Article

CiNeT: A Comparative Study of a CNN-Based Intrusion Detection System with TensorFlow and PyTorch for 5G and Beyond

Omesh A. Fernando [*] , Joseph Spring , Hannan Xiao

Posted Date: 27 August 2025

doi: 10.20944/preprints202508.1085.v2

Keywords: convolutional neural network (CNN); lossless encoding and decoding; network traffic to images; images to network traffic; intrusion detection system (IDS); computational complexity; TensorFlow; PyTorch; 5G; B5G

Article

CiNeT: A Comparative Study of a CNN-Based Intrusion Detection System with TensorFlow and PyTorch for 5G and Beyond

Omesh A. Fernando [1,*], Joseph Spring [1], and Hannan Xiao [2]

[1] Department of Computer Science, University of Hertfordshire, Hatfield AL10 9AB, UK
[2] Department of Informatics, King's College London, London WC2R 2LS, UK
[*] Correspondence: w.k.fernando@herts.ac.uk

Abstract

As 5G and beyond networks grow in heterogeneity, complexity, and scale, traditional Intrusion Detection Systems (IDS) struggle to maintain accurate and precise detection. Deep Learning (DL) provides a promising alternative detection method however, DL based IDS suffer from issues relating to interpretability, performance variability tied to framework dependency and high computational overhead which limit and question their deployment in real-world applications. In this study we introduce a novel new DL based IDS employing Convolutional Neural Networks (CNN) to identify different types of attack together with a bijective encoding-decoding pipeline that acts between network traffic features (such as IPv6, IPv4, MAC addresses and protocol data), and their RGB representations that facilitates our DL IDS in detecting spatial patterns without sacrificing fidelity. Following the application of our bijective pipeline from RGB images to their corresponding network traffic features the 'black-box'problem is resolved and digital forensic traceability enabled. Finally, we have evaluated our DL IDS on three datasets, UNSW NB-15, InSDN, and ToN_IoT, analysing the accuracy, GPU usage, memory utilisation, training, testing, and validation time. To summarise, in this study, we present a new CNN based IDS with an end-to-end pipeline between network traffic data and their RGB representation that offers high performance and transparency.

Keywords: convolutional neural network (CNN); lossless encoding and decoding; network traffic to images; images to network traffic; intrusion detection system (IDS); computational complexity; TensorFlow; PyTorch; 5G; B5G

1. Introduction

The rollout of 5th Generation (5G) following the 15th to 19th releases of the 3rd generation partnership project (3GPP) [1], and the rapid advancement of mobile communication technology toward 5G-Advanced [2,3], have provided the standardised, scalable, and reliable foundation necessary for the deployment of use cases at scale, with guaranteed performance, across heterogeneous environments, Connected and Automated Vehicles (CAVs), the Internet of Things (IoT), and Multi-Access Edge Computing (MEC) [4]. These use cases, which compliment primary use cases such as enhanced mobile broadband (eMBB), ultra-reliable low-latency communications (URLLC), and massive machine type communications (mMTC), have placed a heightened demand for secure networks requiring high privacy [5], reliability [6], and availability [7].

The exponential growth of connected devices and end users in mobile telecommunications with smart devices is expected to reach 7.95 billion by 2028 from 7.21 billion by the end 2024 [8], where mobile hand-held smart devices already generating 1.3 zettabytes of data in 2024 [9]. This exponential growth of users and traffic volume, raises an important question: *How do we protect the 5G and beyond from adversarial threats?*. The application of traditional Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), such as Firewalls, have proven to be ineffective in the face of diverse traffic patterns with exponential volumes [10]. The complexity of attacks, sophistication of Advanced Persistent Threats (APTs), and large-scale Distributed Denial of Service (DDoS) attacks, and the widespread availability of tools to conduct a sophisticated attack, continue to challenge existing security capabilities [11–13]. As per the research by [13] Machine Learning (ML) methods have proven to be an effective, faster and more accurate threat detection and prevention mechanism, compred to the traditional security solutions in the big-data era [14]. Despite its many advantages in cyber security, ML methods alone do possess drawbacks. Adversarial attacks and evasion [15], adaptability limitations, black box problem, lack of interpretability, [12,16], for example, can be construed as some drawbacks. Deep Learning (DL), a subset of ML, has attracted the attention of both academia and industry as a viable method to overcome drawbacks such as the above, and with its potential to autonomously grasp intricate patterns and connections within data [17]. Similarly, authors at [18], postulate that DL methods are increasingly becoming essential in cyber security solutions and deployments, in an attempt to address the fundamental limitations of ML based solutions.

Recent studies have explored the use of images based on network traffic to leverage the strong feature extraction and extrapolation capabilities offered through the use of CNN, and hence it has become a popular DL model for the intrusion detection problem. Popular methods include converting network traffic to grayscale [19], RGB images [20], and serialised images [21]. Higher accuracy generated by the use of RGB images as opposed to grayscale methods, motivated us to develop a new bijective method for image encoding and decoding to ensure that CNN approaches to the IDS problem remain interpretable by security experts, post detection. Unlike in serialised approaches where reconstruction of the original network data remains inaccessible.

Despite the popularity and the advances in the field of CNN based IDS's comparative studies involving computational efficiency and overhead remain unavailable, particularly, with respect to the use of TensorFlow and PyTorch frameworks. Differences in constructing computational graphs, hardware utilisation, optimisation strategies, and memory allocation and de-allocation, can significantly impact the model performance, training and testing time, and scalability [22]. Hence test the effectiveness of our approach, we employed three popular datasets used in the study of ML/DL for the intrusion detection problem. They are UNSW NB-15 [23], InSDN [24], and ToN_IoT [25].

This paper proposes CiNeT (Classify in Network Transformation), a novel CNN based IDS for 5G and beyond networks. CiNeT was developed to be integrated into the NeT2I (NEtwork Traffic to Images) and I2NeT (Images to NEtwork Traffic) pipeline [26], leveraging the PyTorch and TensorFlow frameworks. This process alleviates the "black-box" problem when ML/DL models have been applied and facilitates interpretability in detection and results [27].

We implement CiNeT and compare the two variants CiNeT-TF (developed using TensorFlow) and CiNeT-PT (developed using PyTorch). Our analysis and evaluation in terms of accuracy, loss, memory utilisation, GPU usage, training time, validation time, and architectural layers are present in Section 5. The experiments and collected results demonstrate that CiNeT-PT outperformed CiNeT-TF across all use cases for the above metrics. Furthermore, we analysed the robustness of NeT2I and I2NeT across theoretical and empirical complexities to test its application for systems requiring real-time detection.

1.1. Contribution

This paper makes the following contributions to the field of 5G and beyond security, by employing a CNN based IDS system. In particular, we have:

- An improved NeT2I and I2NeT implementation that encodes network traffic into RGB images and decodes the images back to network traffic in a bijective manner.
- Developed and evaluated a novel Plug-and-Play CiNeT detection algorithm. A CNN based IDS tailored for resource constrained edge devices for 5G and beyond.
- A comparative study of CiNeT between TensorFlow and PyTorch across varying architectural depths, in terms of performance trade-offs, resource usage, scalability, and speed of training and testing.

- Validation of results across UNSW NB-15, InSDN, and TON_IoT datasets.

1.2. Structure of the Paper

The remainder of this paper is organised as follows. Related work is discussed in Section 2. Proposed algorithms in this research are discussed at Section 3, outlining the evaluation metrics in Section 4 and programs used. Results and analysis for the proposed algorithms across datasets are presented in Section 5. The overall results obtained from our research are discussed in Section 6. Concluding remarks, observations, and future work are presented in Section 7.

2. Related Work

This section presents a review of the state-of-the art literature in the field of Network Intrusion Detection Systems (NIDS), datasets, deep learning frameworks, deep learning models, various model architectures, and finally, trends and advancements of image based representations for network traffic, which lays the foundation of our work.

The development, evaluation, and deployment of IDS based on ML models and the acquired accuracy rely heavily on the dataset. The quality and the realistic nature of network traffic pertaining in the dataset contributes towards the applicability and performance of such NIDS based on ML models. The UNSW NB-15 dataset [23], was introduced and designed to address the limitations and drawbacks found in the KDD Cup 99 dataset, which can be construed as outdated. UNSW NB-15 contained modern network attacks, attack vectors, and a balanced distribution between normal and malicious network traffic. Another dataset used in the context of researching ML models to the NIDS problem is the InSDN dataset [24], which is based on a Software Defined Network environment, providing a large scale collection of network traffic distributed amongst a multi-class classification of malicious and normal traffic. Finally, the recently collected ToN_IoT [25] dataset, has emerged as a comprehensive collection of network traffic in a multi class classification, and a benchmark dataset for training, testing, and evaluating ML models for NIDS problem. When conducted a cross evaluation amongst the three datasets, ToN_IoT dataset has shown a significant performance gain as opposed to UNSW NB-15 [28], and InSDN [29] datasets, highlighting its applicability and the quality of the data found within the collection. Hence, the ToN _IoT dataset may be considered a benchmark dataset for application ML models to advance research into NIDS's.

Deep Learning models have shown a superiority over ML models for the IDS problem, given their ability to reveal complex, hierarchical features and their representations from raw unprocessed data. Unlike ML models whose accuracy rely on accurate feature selection and engineering, DL models such as Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Artificial Neural Network (ANN), and Deep Neural Networks (DNN) for example, facilitate the extraction and identification patterns in network traffic, increasing their robustness to ever evolving network attacks. Researchers at [14] demonstrated that a DL based NIDS outperformed classical ML based NIDS, achieving higher accuracy and lower false alarms, in sophisticated multi-class attacks [30]. The research in [31] further emphasizes that the automatic feature extraction capability facilitated by DL models as a key contributing factor for their superior performance. These studies collectively confirm the validity of applying DL models to the NIDS problem, as opposed to using traditional ML models.

The choice of DL framework can significantly impact the development, training, evaluation, testing, and deployment of an IDS based on a DL model. TensorFlow and PyTorch are the most commonly used framework for deploying a model for DL tasks [32]. The extensive community support, ecosystem, and integration into cloud platforms such as Google Colab, Lambda Labs, and Jupyter Notebooks, for example, have contributed to these two frameworks becoming the de-facto standards in both academia and industry. TensorFlow, due to its static computational graph, offers high performance and a seamless capacity for integration and deployment. Conversely, PyTorch offers a dynamic computation graph that offers greater flexibility, ease of debugging, and faster prototyping have made this framework increasingly popular [2,32]. A recent survey [5] highlights the interchange between the two frameworks, PyTorch and TensorFlow, with the choice of framework having the potential to impact their application, particularly for real-time interference, model interpretability, and ease of development and deployment.

Long Short-Term Memory (LSTM) networks have been widely adopted and applied in NIDS's based on DL models. Their ability to model temporal dependencies, the capability to handle the multivariate, and sequential nature of network traffic has contributed to its wider adoption for intrusion detection problems. Respective research by [33–36] developed IDS's employing LSTM models and tested its accuracy for a variety of datasets. However, the LSTM models require a high resource availability and a computational complexity, which can constitute a bottleneck when applied in a real-time system. Also, the sequential nature of LSTM, hinders parallelisation, leading to longer training, validation, and testing periods. Furthermore, [33], has highlighted that LSTM's struggle with extensive dependencies in network traffic, and the reliance on hyper-parameter tuning, can contribute to overfitting and reduced model generalisation.

The adaptation of Large Language Models (LLM) into various applications across industry and academia has opened an avenue for it to be considered with the intrusion detection problem. Research by [37,38] employed LLM's for exploring and analysing patterns in network traffic generating responses pertaining to network activity (malicious or non_malicious). Despite the adoption of LLMs' into the intrusion detection problem, similar to the LSTM models, resource requirements and computation remain extremely high [39]. Inference latency and the "black-box" nature of LLMs' create a bottleneck in their application to IDS's requiring real-time detection and the rationale for decisions.

Deep Neural Networks and Artificial Neural Networks have been foundational DL models for the intrusion detection problem. Research by [40–43] presented DL models tested across various benchmark datasets to support the usability of DL in NIDS's. Despite the popularity and the foundational work, these models face a tendency to overfit, which can lead to higher rates of false positive alarms. The extensive requirement for hyper-parameter tuning, the higher computational complexity, and the "black-box" problem create barriers to their deployment in real-time systems and resource constrained environments.

Convolutional Neural Networks (CNN) have become a popular choice for IDS's due to high accuracy, precision, and excellent generalisation properties [44]. This prompted us to explore CNN's as a viable option for the detection of malicious traffic in the 5G and beyond network infrastructure. Surveys conducted by [45,46] provides a comprehensive overview of the CNN based IDS's with various architectures, performance, and their applicability for intrusion detection problems. Respective research by [26,47–50] introduces various applications of CNN's to further solidify the usability of CNN's to the intrusion detection problem, emphasizing their effectiveness in handling high-dimensional network data. Compared to LSTM and LLM models, CNNs are computationally viable, with a lower cost and smaller memory footprint. CNNs also allow parallel processing, leading to faster training and suitable for inference times, unlike LSTM models. The ability to handle high dimensional data, shared weights, and parameter reduction through pooling avoids the overfitting problem unlike in ANN and DNN models. Finally, CNN's offer greater interpretability avoiding the "black-box" problem found in other models.

One of the greatest challenges in the application of CNN's to network security involves the process of encoding data into a form recognisable by the CNN. As CNNs accept images for training and testing, data collected from a dataset has to be converted into images. Existing research uses various methods to encode network traffic data into images that can be used to train a CNN algorithm. Research published in [19,51–56] employed grayscale images to represent and encode the desired features of network traffic for training and testing a CNN algorithm. However, representing network traffic by grayscale images can lead to a loss of information since modern network traffic, due to heterogeneity and complexity, can contain data that exceeds a pixel value in the grayscale (0-255).

Due to this limitation on grayscale images, authors in [21,57,58] and [20] employed mechanisms to encode network traffic as RGB images. Improvements in accuracy were observed [20,58], when employing RGB images in preference to grayscale images. Authors of [57] and [20], employed a tiled image approach along the x and y axis, for a given pixel length and width, whereas [21,58] generated serialised RGB images. Although, these images are capable of producing a higher accuracy than with grayscale images, RGB can represent a pixel value between 0 - 16,777,215, in a tiled image which places a high demand on CPU, memory and time of execution. Although, serialised images can be construed as computationally viable, but are not bijective with information being lost during conversion, between network traffic and images. This information loss, could potentially classify a CNN based IDS as a "black-box" solution. In this work, we propose an improved NeT2I and I2NeT bijective algorithm [26] for encoding and decoding of network traffic including IPv6 addresses, MAC addresses, IPv4 addresses, and other network features. We also present a novel detection algorithm CiNeT, a dynamic plug-and-play algorithm capable of detecting classes and determining related parameters.

3. Proposed Algorithm

In this section we present a novel encoding algorithm (NeT2I), used to represent network traffic in terms of RGB images and a corresponding decoding algorithm (I2NeT) to translate the RGB images back into network traffic. We include in this section a dynamic detection algorithm (CiNeT) developed using both TensorFlow and PyTorch which we use to identify observed traffic.

3.1. Encoding and Decoding Network Traffic (NeT2I-I2NeT Pipeline)

The secure, structured, and deterministic transformation of network data traffic into images is achieved through the integration of a NeT2I-I2NeT pipeline, initially introduced in [26]. The pipeline has been significantly developed since its first inception, to now include a comprehensive set of network features that accommodate for example IPv4 and IPv6 addresses, MAC addresses, flow duration, jitter, protocol identifiers, and packet load. These features form a multi-dimensional representation of network traffic and behaviour, for the detection of malicious attacks with advanced complexity.

The pseudo code presented in Algorithm 1 (Encoding Algorithm: NeT2I) describes our method for representing network traffic as PNG images. A generated image from the NeT2I, can be seen in Figure 1. The ability to handle IPv6 addresses is a significant new feature of this pipeline, driven by the urgent need for network security solutions to support the global transition to IPv6 [59]. The US Executive Order on furthering the Nation's Cybersecurity and Cyber readiness and the European Unions IPv6 action plan require all public internet assets to be IPv6 ready [60] by the end of 2025. In response to this, our NeT2I-I2NeT pipeline, provides a meticulous, lossless, and a deterministic method for encoding and decoding IPv6 addresses. To achieve this each 128-bit value is converted to its packed binary form. The 16-byte sequence is padded with two additional zero bytes to form an 18-byte array, compatible with a 3-byte RGB grouping, to form six consecutive RGB pixel tuples, with values ranging from 0 to 255. Each of these now corresponds to one horizontal stripe in the generated image. This approach ensures that each IPv6 address that is to be mapped and encoded to an RGB image, does so in a lossless manner, maintaining full address fidelity.

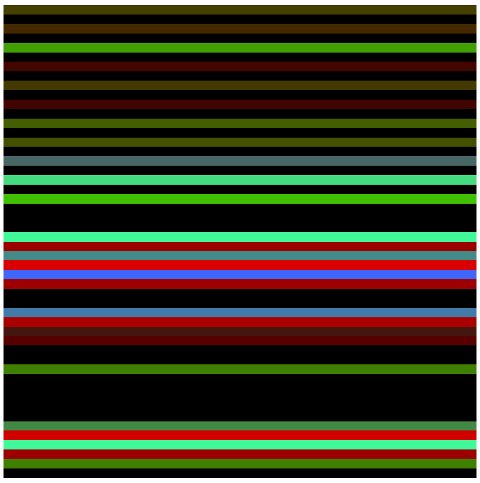


Figure 1. A Visual Representation of an Image Generated via NeT2i

During the encoding phase, NeT2I automatically generates a JSON companion file containing metadata, to ensure accurate and unambiguous data reconstruction. This file meticulously documents the structure of the network traffic in the input CSV file. The I2NeT decoder, as described in Algorithm 2 (Decoding Algorithm:I2NeT) reverses this process by reading the JSON metadata file by identifying six consecutive RGB pixels from the PNG image, with each pixel representing three of the 18-byte padded binary representation. The first 16 bytes are interpreted as the packed binary of the IPv6 address, allowing an accurate reconstruction of the original IPv6 address in a standardised colon separated hexadecimal format. In order to ensure robustness, at the decoding stage images generated from IPv6 addresses are tagged with an additional prefix(e.g., *ipv6_1022.png*). This ensures that in a scenario where the JSON file is not available, I2NeT can default to the appropriate path for decoding. The structured, type-aware, and error resilient approach ensures that I2NeT achieves high-fidelity for the recovery of IPv6 addresses.

In the NeT2I algorithm, floating point numbers are mapped to image data using an invertible binary serialisation approach that ensures both accuracy and integrity. Initially, a floating point number is converted to its 32-bit IEEE 754 [61] binary representation ensuring network byte-order consistency. This 4-byte floating point number is then represented in two RGB pixels. The first RGB pixel carries the first three bytes (RGB) and the second pixel encodes the fourth byte in its Red channel, setting the Green and Blue channel to zero. This deterministic mapping sequence allows the decoder in the I2NeT algorithm to accurately reconstruct the original floating-point number from the RGB representation, correctly reassembling the byte sequence in a loss-less manner. As in the case above, metadata is stored in the JSON file to aid at the decoding stage.

The I2NeT decoder captures and reconstructs floating-point numbers from images through a deterministic process that ensures accuracy, integrity, and fidelity. During the decoding phase, I2NeT extracts the RGB stripe from the horizontal bands in the image and processes each pair of pixels to recover and reconstruct the original 4-byte float structure. The original floating-point number is reconstructed by concatenating these four bytes accurately following the byte order. The I2NeT decoder follows type-aware reconstruction employing the metadata from the JSON file to determine fields which are floating-point numbers. In a scenario where the JSON file is not available, I2NeT uses a default generic decoding mechanism, ensuring a robust recovery. Similarly, I2NeT also follows bound checking, exception handling, and error resilience features to manage potential mismatches or corrupted data.

This end-to-end pipeline of NeT2I for encoding and I2NeT for decoding, creates a bijective transformation between structured data and corresponding image representation. The latest version of NeT2I can be found in the GitHub repository [62] and in the Python Package Index (PyPI) [63]. The latest version of I2NeT can be found in the GitHub repository [64] and in the Python Package Index (PyPI) [65]. By integrating support for modern network features such as IPv6 and floating-point numbers, the NeT2I-I2NeT pipeline provides a robust, scalable, future-proof, errorless, and a deterministic solution for securing network data transmission.

Algorithm 1 Encoding Algorithm: NeT2I

```
Input: input.csv
Output: image.png
Function encodeCSVToRGB(csv_path):
    Data: Loaded CSV data
    Data: Detected data types per column (Float, IPv4, IPv6, MAC, String)
    Data: Processed RGB pixel array
    /* Step 1:  Clean output directory and temporary files                    */
    cleanOutputDirectory()
    /* Step 2:  Split CSV into IPv4 and IPv6 datasets                         */
    ipv4_data, ipv6_data ← splitCSVByIPType(csv_path)
    /* Step 3:  Process each dataset separately                               */
    processDataset(ipv4_data, is_ipv6=False)  processDataset(ipv6_data, is_ipv6=True)
    /* Step 4:  Save type information for decoding                            */
    saveTypeInformation()
    /* Step 5:  Generate images from RGB pixel arrays                         */
    generateImages()
Function convertDataToRGB(row, types):
    Data: List of RGB pixel tuples per row
    rgb_row ← [ ]
    foreach value, dtype in zip(row, types) do
        switch dtype do
            case "IPv6 Address" do
                rgb_pixels ← convertIPv6ToRGB(value)
            case "IPv4 Address" do
                float_val ← convertIPv4ToFloat(value)
            case "MAC Address" do
                float_val ← convertMACToFloat(value)
            case "Float" do
                rgb1, rgb2 ← floatToTwoRGB(value)
            case "String" do
                hash_val ← hashStringToFloat(value)
            otherwise do
                Error: Unknown data type
            end
        end
    end
    return rgb_row
Function createImageFromRGB(rgb_row, image_id, prefix):
    initializeImageArray() /* Fill image with RGB pixel stripes               */
    rows_per_color ← calculateStripeHeight(rgb_row)  current_row ← 0
    foreach rgb in rgb_row do
        (r, g, b) ← normalizeRGB(rgb)  for i = 0 to rows_per_color do
            if current_row < image_size then
                fillRow(current_row, r, g, b)  current_row ← current_row + 1
            end
        end
    end
    /* Fill remaining rows with last color if needed                          */
    while current_row < image_size do
        fillRow(current_row, r, g, b)  current_row ← current_row + 1
    end
    return saveImage(image_id, prefix)
```

Algorithm 2 Decoding Algorithm:I2NeT

```
Input: input.png (directory or single image)
Output: output.csv
Function decodeImagesToCSV(image_path):
    /* Step 1: Auto-detect IP version and load type info */
    autoDetectIPVersion(image_path)
    /* Step 2: Extract RGB pixel values */
    rgb_values ← extractRGBFromImage(image_path)
    /* Step 3: Detect expected data structure */
    adaptive_type_info ← detectDataStructure(len(rgb_values), type_info)
    /* Step 4: Reconstruct structured values */
    decoded_values ← reconstructValues(rgb_values, adaptive_type_info)
    /* Step 5: Write to CSV */
    writeToCSV(decoded_values)
      return output.csv

Function extractRGBFromImage(mage_path):
    /* Extract RGB pixel values using stripe detection */
    img = openImage(image_path)
      array = convertToNumpyArray(img)
      stripe_colors = detectColorStripes(array)
      return stripe_colors

Function detectDataStructure(rgb_count, type_info):
    if type_info is available then
        calculateExpectedPixelCount(type_info)
        if rgb_count < expected_count then
            return truncated_types
        end
        else
            return type_info
        end
    end
    else
        /* Fallback: assume 2 pixels per float */
    end

Function reconstructValues(rgb_values, adaptive_type_info):
    for orig_type in original_types do
        if orig_type == "IPv6 Address" then
            ipv6 ← decodeIPv6From6Pixels(rgb_values)
            reconstructed.append(ipv6)
        end
        else if orig_type == "IPv4 Address" then
            ipv4 ← decodeIPv4From6Pixels(rgb_values)
            reconstructed.append(ipv4)
        end
        else if orig_type == "MAC Address" then
            mac ← formatMACFromChunks(chunk1, chunk2)
            reconstructed.append(mac)
        end
        else if orig_type == "Float" then
            float_val ← decodeFloatFrom2Pixels(rgb_values)
            reconstructed.append(float_val)
        end
        else if orig_type == "Integer" then
            int_val ← decodeIntFrom2Pixels(rgb_values)
            reconstructed.append(int_val)
        end
        else if orig_type == "String" then
            hash_val ← decodeHashFromPixel(rgb_values)
            reconstructed.append(str(hash_val))
        end
    end
    return reconstructed

Function writeToCSV(decoded_values):
    /* Write decoded values to CSV file */
    createNewCSVFile()
      for row in decoded_values do
        writeRowToCSV(row)
    end
    return output.csv
```

3.2. *Detection Algorithm (CiNeT)*

The CiNeT algorithm conducts a multi-class classification of network traffic by leveraging DL techniques. This algorithm has been developed and designed to integrate seamlessly with the NeT2I and I2NeT frameworks. CiNeT is implemented using both TensorFlow and PyTorch to enable cross-framework evaluation. This method allows for comparative analysis on model performance, training efficiency, accuracy, loss convergence, and computational complexity. The TensorFlow variant of CiNeT employs a CNN architecture with three convolutional blocks whereas the PyTorch variant employs four convolutional blocks, with batch normalisation and dropout for robustness. The model is trained on images generated via NeT2i and classification accuracy is validated by reconstructing predictions via I2NeT, ensuring a seamless encoding-classification-decoding pipeline of a full-fledged DL based IDS.

The CiNeT algorithm features an autonomous and an adaptive class detection mechanism that enables automation without requiring manual intervention or the specification of output classes. During the initialisation phase CiNeT scans the input directory and sub-directories within and passes the sub-directory names to dynamically construct a list of class labels and assign a unique index to each. The detected class count is used to configure the CNN, setting the output units to K, where K denotes the number of classes. This is used to select the appropriate loss function. For $K = 2$, binary cross-entropy is used and where $K > 2$, categorical cross-entropy is used.

In addition to autonomous class detection, CiNeT employs a data driven mechanism to select the optimal batch size. This decision is based on the total number of training samples. The algorithm determines the best candidate batch size from a predefined set of rules to reduce underutilisation and thereby improve training efficiency. Along with adaptive class detection and automated batch size selection, CiNeT is a highly scalable, efficient, and a plug-and-play classification system that integrates seamlessly with NeT2I and I2NeT for encoding and decoding.

3.2.1. TensorFlow

The CiNeT algorithm implemented within this ecosystem, implemented in TensorFlow, performs multi-class classification using a CNN architecture, employing three sequentially stacked convolutional blocks. The pseduo code can be found at Algorithm 3 (CiNeT-TF Algorithm). Each of these, comprises a Conv2D layer with ReLU activation, followed by max-pooling, and drop out for spatial downsampling and regularisation. The input layer accepts images in RGB format specifying the three colour channel and the size of 150 pixels ($150 \times 150 \times 3$). The size corresponds to the output image generated via NeT2I. The TensorFlow variant of CiNeT progressively extracts features through increasing filter depth of $32, 64$, and 128, respectively per layer. Regularisation is achieved by a 25% dropout upon each max-pooling operation, while the fully connected layer includes a 512-unit dense layer with a 50% drop out to ensure overfitting is mitigated.

CiNeT leverages Keras API for model creation, complication, training, and evaluation. RMSProp optimiser was utilised with a learning rate set for 1×10^{-4} with an adaptive loss function based on the class count (binary cross-entropy for two classes or categorical cross-entropy for multi-classes). Data augmentation was applied during training to improve generalisation and avoid overfitting with random rotation, shifting, shearing, zooming, and flipping across the horizontal plane. Performance validation was conducted using a test-set (15%), with training and validation accrued for 70% and 15% respectively. The confusion matrix consisted of Accuracy, F1-score, recall, and precision, these being generated using the results obtained from the test-set.

Algorithm 3 CiNeT-TF Algorithm

```
Input: Directory with class folders (e.g., data_A, data_B)
Output: Trained model (.h5), accuracy, confusion matrix
Function main():
    config ← splits=(0.7,0.15,0.15), epochs=100
      checkResources() /* Verify GPU and RAM                                   */
    class_names ← discoverClasses("/home/ubuntu/Images/")
    createDirs("training,validation,testing")
      for cls in class_names do
        src ← "data_" + cls
          splitData((src, f"training/cls", f"validation/cls", f"testing/cls"))
    end
    buildModel((num_classes))
      total ← countImages("training/")
      batch_size ← selectBatchSize(total)
      createLoaders(batch_size)
    history ← train(epochs=100)
      evaluation ← evaluateModel()
      plotData(targets, predictions)
      saveModel("model_saved")
      return success
Function discoverClasses(dir):
    classes ← [ ]  for item in listDir(dir) do
        if isDir(item) and startsWith(item,"data_") then
            classes.append(removePrefix(item,"data_"))
        end
    end
    return sorted(classes)
Function splitData(src, train, val, test):
    files ← shuffle(nonEmptyFiles(src))
      splitAndCopy(files, 0.7, 0.15, train, val, test)
Function buildModel(n):
    model ← Sequential()
      model.add(2DConvolutional Layer 1(32, (3,3), activation='relu', input_shape=(150,150,3)))
      model.add(MaxPooling2D(2,2))
      model.add(Dropout(0.25))
    model.add(2DConvolutional Layer 2(64, (3,3), activation='relu'))
      model.add(MaxPooling2D(2,2))
      model.add(Dropout(0.25))
    model.add(2DConvolutional Layer 3(128, (3,3), activation='relu'))
      model.add(MaxPooling2D(2,2))
      model.add(Dropout(0.25))
    model.add(Flatten())
      model.add(Dense(512, activation='relu'))
      model.add(Dropout(0.5))
      model.add(Dense(n, activation))
      if num_classes == 2 then
        activation = sigmoid
          loss_fn=binary_crossentropy
    else
        activation = softmax
          loss_fn = categorical_crossentropy
    end
    model.compile(RMSprop(lr=1e-4))
Function createLoaders(bs):
    trainGen ← ImageDataGenerator( Image_Augmentation)
      valTestGen ← ImageDataGenerator()
    trainGen ← trainGen.flow_from_directory("training/", bs, class_mode=auto)
      valGen ← valTestGen.flow_from_directory("validation/", bs, class_mode=auto)
      testGen ← valTestGen.flow_from_directory("testing/", bs, class_mode=auto)
Function train(epochs):
    /* Fit model using generators                                               */
    history ← model.fit( trainGen, epochs=epochs, validation_data=valGen, verbose=1 )
      return history
Function evaluateModel():
    /* Evaluate on test set                                                     */
    predictions ← model.predict(testGen)
      true_labels ← testGen.classes
    if num_classes == 2 then
        pred_labels ← (predictions > 0.5).astype(int)
    end
    else
        pred_labels ← argmax(predictions, axis=1)
    end
    acc ← accuracy(true_labels, pred_labels)
      return accuracy=acc, predictions=pred_labels, targets=true_labels
Function plotData(targets, preds):
    cm ← confusion_matrix(targets, preds)
Function saveModel(name):
    model.save('model')
```

3.2.2. PyTorch

The second variant of the CiNeT algorithm was implemented using PyTorch, performing multi-class classification leveraging CNN. The pseduo code can be found at Algorithm 4 (CiNeT-PT Algorithm). The optimal model architecture comprises of 4 sequentially stacked convolutional blocks, each consisting of a Conv2D layer, batch normalisation, ReLU activation, max-pooling, and dropout for regularisation. Similar to the TensorFlow variant, the input layer accepts RGB images of three channels$(150 \times 150 \times 3)$. The PyTorch variant of CiNeT progressively extracts features through increasing filter depths of $32, 64, 128$, and 256, respectively per layer. Regularisation is achieved by a 25% dropout upon each max-pooling operation, while the fully connected layer includes 512 neurons and a 50% drop out to ensure overfitting is mitigated, before the final classification layer.

RMSProp optimisation is employed with a learning rate of 1×10^{-4}, where the loss function is based on the number of classes. BCEWithLogitsLoss for binary classes and CrossEntropyLoss for multi-classes was chosen. Random rotation, horizontal flipping, shifting, shearing, and zooming under augmentation was used to increase input diversity and to reduce overfitting. Performance validation was conducted using a test-set (15%), with training and validation accruing for 70% and 15% respectively. The confusion matrix comprised of Accuracy, F1-score, recall, and precision was generated using the results collated from the test-set.

Algorithm 4 CiNeT-PT Algorithm

```
Input: Directory with class folders (e.g., data_A, data_B)
Output: Trained model (.pth), accuracy, confusion matrix
Function main():
    config ← splits=(0.7,0.15,0.15), epochs=100
       checkResources() /* Verify GPU and RAM                                    */
    class_names ← discoverClasses("/home/ubuntu/Images/")
    createDirs("training,validation,testing")
     for cls in class_names do
        src ← "data_" + cls splitData(src, f"training/cls", f"validation/cls", f"testing/cls")
    end
    buildModel(num_classes)
     total ← countImages("training/")
     batch_size ← selectBatchSize(total)
     createLoaders(batch_size)
    history ← [] for epoch ← 1 to 100 do
        train_loss, train_acc ← trainEpoch()
          val_loss, val_acc ← valEpoch()
    end
    test_acc, preds, targets ← evalTest()
     PlotData(targets, preds)
     saveModel("pytorch_model.pth")
     return success
Function discoverClasses(dir):
    classes ← [ ] for item in listDir(dir) do
        if isDir(item) and startsWith(item,"data_") then
            classes.append(removePrefix(item,"data_"))
        end
    end
    return sorted(classes)
Function splitData(src, train, val, test):
    files ← shuffle(nonEmptyFiles(src))
     splitAndCopy(files, 0.7, 0.15, train, val, test)
Function buildModel(n):
    2DConvolutional Layer 1 (32, (3,3))
     BatchNorm2d (32)
     MaxPooling2d (2,2)
     Dropout()
    2DConvolutional Layer 2(64, (3,3))
     BatchNorm2d (64)
     MaxPooling2d (2,2)
     Dropout()
    2DConvolutional Layer 3(128, (3,3))
     BatchNorm2d (128)
     MaxPooling2d (2,2)
     Dropout()
    2DConvolutional Layer 4 (256, (3,3))
     BatchNorm2d (256)
     MaxPooling2d (2,2)
     Dropout()
    Flatten ($256 \times 9 \times 9 = 20{,}736$ features),
     Fully Connected (20736 → 1024)
     Fully Connected (1024 → num_classes)
    if num_classes == 2 then
        criterion ← BCEWithLogitsLoss()
    else
        criterion ← CrossEntropyLoss()
    end
    optimizer ← RMSprop(model.parameters(), lr=1e-4)
    moveModelToDevice(model, device)
Function createLoaders(bs):
    trainT ← [Image_Augmentation]
     valT ← []
    trainLoader ← DataLoader("training/", trainT, bs, shuffle=True)
     valLoader ← DataLoader("validation/", valT, bs)
     testLoader ← DataLoader("testing/", valT, bs)
Function trainEpoch():
    foreach (data,target) in trainLoader do
        output ← model(data)
    end
    return loss/N, acc%
Function valEpoch():
    foreach (data,target) in valLoader do
        output ← model(data.to(device))
    end
    return loss/len, correct/total
Function evalTest():
    foreach (data,target) in testLoader do
        pred ← argmax(model(data))
    end
    return accuracy, pred
Function saveModel(path):
    torch.save('model')
```

4. Evaluation Metrics for the Algorithms

4.1. Workflow and Datasets

Among publicly available datasets, InSDN contains the most recently collected data for malicious and non-malicious traffic. The dataset contains multi-classes and has been collated on a Software Defined Networking environment, with 80 features, 18% of the dataset consists of malicious network data. The ToN_IoT dataset is another comprehensive dataset consisting of multi-class traffic along with malicious and non-malicious traffic. The dataset contains 44 features with approximately 15% of the dataset classified as malicious. Finally, the UNSW-NB 15 dataset contains 49 features with 12% of the available traffic in the dataset corresponding to malicious traffic. The workflow of the proposed algorithms and their applications on the datasets is shown in Figure 2. CSV files extrapolated from the datasets and used as input to the NeT2I algorithm generating PNG images based on the network traffic. The generated PNG images are then input into the CiNeT algorithm. Following detection by the CiNeT algorithm, the I2NeT algorithm is used to decode the images back to a CSV file in order to evaluate the accuracy of our detection algorithm. The following Table 1 shows the selected features for each dataset, through Recursive Feature Elimination with Random Forest due to its ability to handle data with high-dimensions and its ability to handle non-linear complex interactions [66].

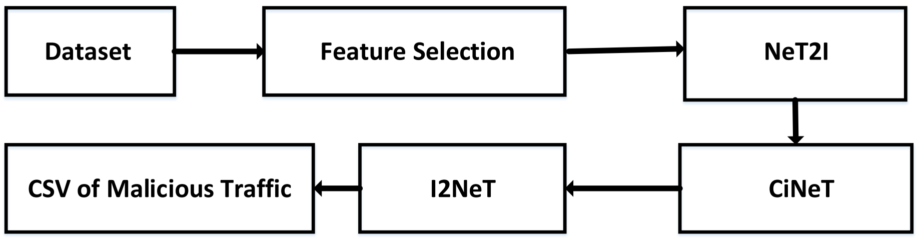


Figure 2. Workflow of the Proposed Algorithms.

Table 1. Features Selected from each Dataset.

Dataset	Selected Features
In-SDN	f2, f3, f4, f5, f6, f8, f9, f10, f11, f12, f13, f15, f19, f21, f22, f23, f24, f27, f31, f68, f78
UNSW NB-15	f1, f2, f4, f5, f6, f7, f10, f11, f16, f17, f18, f19, f21, f24, f25, f26, f27, f28, f38, f40, f45
ToN IoT	f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f16, f17, f18, f19, f20, f21

4.2. Theoretical and Empirical Performance Analysis

The NeT2I and I2NeT algorithms were evaluated with respect to execution time, CPU usage, memory utilisation, and time complexity in terms of the Big-O notation, supporting both theoretical and empirical performance analysis.

4.3. Theoretical Analysis

For our theoretical evaluation of the computational complexity, we employ the Big-O notation framework for NeT2I, I2NeT, and CiNeT algorithms. As per the study of [67], the application of a theoretical evaluation using this framework, provides a characterisation of the algorithm as a function of the input size. The encoding to detection to decoding processes within the proposed pipeline involves a deterministic transformation that encompasses problems that can be solved in polynomial time. This efficiency is critical for both desk approaches and real-time applications, requiring predictable and scalable performance is essential [67].

4.4. Empirical Computational Complexity

As discussed in [67], identifying accurate empirical measurements of an algorithm's execution time and memory utilisation is important and essential, as theoretical complexity alone may not reflect accurate performance due to memory allocation, de-allocation, system-level overheads, and I/O operations. To accommodate this, we collect and collate execution time, CPU usage, memory utilisation, and GPU utilisation.

4.4.1. Execution Time

For measuring the time of execution, we employed the Python `time` library. Variation in execution time can affect performance and efficiency, particularly important when applying the algorithm to an intrusion detection problem based in an environment with low resources. To ensure reliability average execution time is presented in later section with each experiment being repeated over multiple runs ($n = 10$), to mitigate caching and Just-In-Time (JIT) compilation effects [68].

4.4.2. CPU Usage

To monitor CPU usage, Python `psutil` library was employed. The library was utilised to monitor and sample CPU usage at one-second intervals during the execution of NeT2I and I2NeT. To ensure statistical reliability, the average of multiple runs ($n = 10$) were recorded. The experiments were conducted in an environment with the Global Interpreter Lock, which restricts code to be executed as a single-thread. This restriction is important to ensure that the proposed algorithms can be executed in environments with low-processing power [69] such as Multi-Access Edge Computing (MEC) nodes.

4.4.3. Memory Utilisation

As discussed in the work presented by [70], memory allocation in a modern operating system, is inherently imprecise compared to CPU usage or execution time. This is due to over-allocation, caching, and ineffective garbage collection. To monitor the memory consumption across NeT2I, I2NeT, and CiNeT, we employed the Python `memory-profiler` library. Computer systems often over-allocate memory to minimise the allocation and deallocation frequency, and garbage collection does not occur instantaneously. Due to this non-deterministic behaviour, averages of multiple independent runs $(n = 10)$ were recorded, to ensure a more reliable and a comparable indicator.

4.4.4. GPU Utilisation

CiNeT algorithm was designed to leverage GPU acceleration while NeT2I and I2NeT were designed to be CPU bound. To determine the GPU compute usage, and memory allocation, we employed the `nvidia-smi` and CUDA interface. The experiments were conducted in a controlled environment with access to a single GPU to emulate an edge-launched intelligent IDS. AS highlighted in [39], DL executions exhibit suboptimal GPU utilisation despite resource allocation and availability. This leads to inefficient compute usage and prolonged execution times. By measuring utilisation, we ensure that CiNeT achieves high computational throughput and effectively leverages the parallel processing capabilities of the GPU. GPU utilisation is a key indicator for algorithmic efficiency. High and sustained utilisation reflects minimal idle time and effective kernel execution. Low and erratic usage may indicate under utilisation, poor resource management and orchestration, despite high accuracy. Therefore, GPU monitoring provides a valuable and necessary statistic collected over multiple independent runs $(n = 10)$, to ensure performance gains are not achieved at the cost of excessive hardware dependency or energy consumption.

5. Results and Analysis

We collected data on a UVT_Cloud deployment running Ubuntu 22.04 LTS, with a single CPU, 8GB of RAM and 20 GB of HDD space for the NeT2I and I2NeT algorithms. The CiNeT algorithm was trained, validated, and tested using an NVIDIA H100 GPU (96 GB VRAM) within a system based on the ARM64 architecture. 225,000 images representing network traffic that belong to various malicious and non-malicious classes were selected with images being subsequently grouped following the ratio of 70:15:15, for training, validation, and testing, respectively.

5.1. Encoded Images

The Figure 3 represents images generated from distinct lines of network traffic in the input CSV. The image consists of one dimensional horizontal lines with a variable x value and a fixed y value. Each line in the generated PNG, encompasses a network feature such as source IP, destination IP, MAC address, protocol, source port, destination port, packet length, jitter, duration, load. Each file name consists a prefix (ipv4, ipv6) to aid the I2NeT algorithm distinguish and apply the correct decoding strategy.

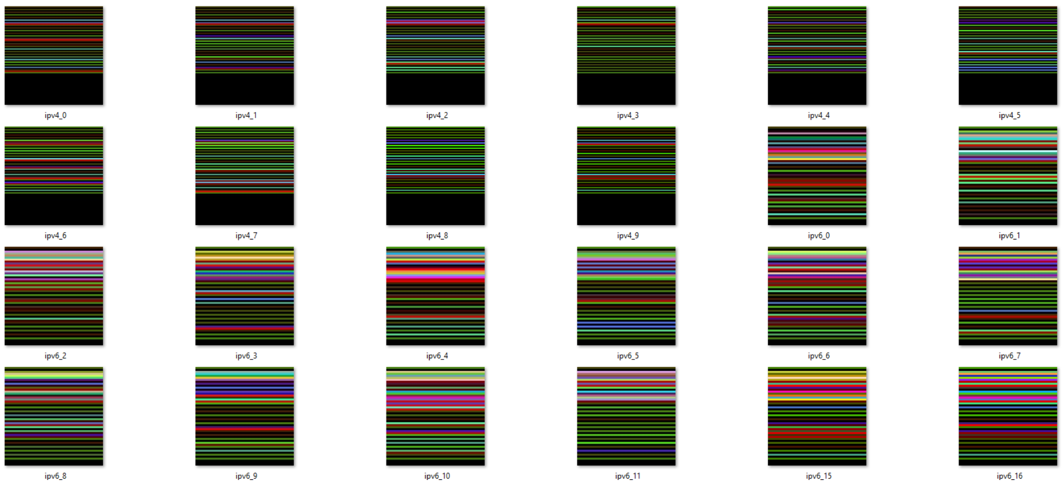


Figure 3. Images Generated from NeT2I.

As stated in Section 3.1, the NeT2I converts floating-point numbers to RGB pixel values without loss, using the IEEE 754 standardisation. For example, for a floating point value 3.14159, the Python function `struct.pack('!f',3.14159)` would be used to generate the IEEE 754 binary representation $[64, 9, 33, 25]$. The respective binary representation would then be split into 2 RGB pixel values $((64, 9, 33), (25, 0, 0))$, which would then be employed at the image generation phase in NeT2I.

When encoding an IPv4 address, NeT2I splits the IP address into 4 octets. Each integer octet is treated as a float number, and mapped to the respective RGB channel. For instance, for the IP address 192.168.1.100,

- `struct.pack('!f',192.0)` $\rightarrow$IEEE 754 $[66, 192, 0, 0]$ $\rightarrow$RGB$((66, 192, 0), (0, 0, 0))$
- `struct.pack('!f',168.0)` $\rightarrow$IEEE 754 $[66, 160, 0, 0]$ $\rightarrow$RGB$((66, 160, 0), (0, 0, 0))$
- `struct.pack('!f',1.0)` $\rightarrow$IEEE 754 $[63, 128, 0, 0]$ $\rightarrow$RGB$((63, 128, 0), (0, 0, 0))$
- `struct.pack('!f',100.0)` $\rightarrow$IEEE 754 $[66, 100, 0, 0]$ $\rightarrow$RGB$((66, 100, 0), (0, 0, 0))$

The generated RGB pixel values are employed in the image generation.

MAC addresses are handled by converting the hexadecimal string into a 48-bit long integer. This integer value is split into two 24-bit chunks, which is converted to a float and mapped using the two RGB pixel method. For example, for a MAC address $00 : 1A : 2B : 3C : 4D : 5E$, we proceed as follows:

- $00 : 1A : 2B : 3C : 4D : 5E$ $\rightarrow$$001A2B$ and $3C4D5E$
- $001A2B$ $\rightarrow$67019
- $3C4D5E$ $\rightarrow$3951966
- `struct.pack('!f', 67019.0)` $\rightarrow$IEEE 754 $[72, 131, 128, 0]$ $\rightarrow$RGB $((72, 131, 128), (0, 0, 0))$
- `struct.pack('!f', 3951966.0)` $\rightarrow$IEEE 754 $[87, 102, 128, 0]$ $\rightarrow$RGB $((87, 102, 128), (0, 0, 0))$

The generated RGB pixel values from the MAC address are employed in image generation.

Finally, IPv6 addresses are mapped to their 128-bit long representation, which is converted to 16 bytes with two additional zeros appended to the end, making an 18 byte long array. For example, for the IP address $2001 : 0db8 : 85a3 : 0000 : 0000 : 8a2e : 0370 : 7334$, we have:

- `IPv6Adress.packed(2001:0db8:85a3:0000:0000:8a2e:0370:7334)` $\rightarrow$ $[32, 1, 13, 184, 133, 163, 0, 0, 0, 0, 0, 0, 138, 46, 3, 112]$
- Appending $[0, 0]$ $\rightarrow [32, 1, 13, 184, 133, 163, 0, 0, 0, 0, 0, 0, 138, 46, 3, 112, 0, 0] \rightarrow$ RGB $((32, 1, 13), (184, 133, 163), (0, 0, 0), (0, 0, 0), (138, 46, 3), (112, 0, 0))$

Following the above process, network features are converted to RGB pixel values and these are then used for image generation without loss.

5.2. Computational Complexity

In Table 2, we present the following averaged times: execution, CPU usage, and memory utilisation for the NeT2I and I2NeT algorithms across three datasets. In Table 3 the computational complexity for the two variants of the CiNeT algorithm are found, averaged across the three datasets. Evaluation for the CiNeT algorithm is carried out in terms of time complexity, GPU usage, and memory utilisation.

Table 2. Computational Complexity of NeT2I and I2NeT algorithms.

Algorithm	Dataset	Number of Images	Total Execution Time	Execution Time Per Image	CPU Usage	Memory Utilisation
NeT2I	InSDN	215,000	99s	0.00046s	100%	18%
	UNSW NB 15	215,000	100s	0.000465s	100%	19%
	TON-IoT	215,000	100s	0.000465s	100%	19%
I2NeT	InSDN	215,000	103s	0.00047s	100%	20%
	UNSW NB 15	215,000	102s	0.000474s	100%	20%
	TON-IoT	215,000	101s	0.000469s	100%	20%

Table 3. Computational Complexity for the CiNeT algorithms.

Algorithm	Layers	Training Time	Validation Time	Testing Time	GPU Usage	Memory Utilisation
CiNeT - TF	1 Layer	12.35 hrs	1.49 hrs	25s	98.2%	26.9%
	2 Layers	12.11 hrs	2.01 hrs	43s	99.9%	27.5%
	3 Layers	13.25 hrs	2.11 hrs	1.24 mins	99.9%	27.7%
	4 Layers	15.1 hrs	2.35 hrs	2.15 mins	99.9%	29.5%
	5 Layers	17.45 hrs	3.05 hrs	3.30 mins	99.9%	30%
CiNeT - PT	1 Layer	5.1 hrs	3.19 hrs	2.01 mins	5.1%	8.4%
	2 Layers	5.24 hrs	3.29 hrs	2.09 mins	8.9%	9.4%
	3 Layers	5.38 hrs	3.31 hrs	2.11 mins	13.2%	10.6%
	4 Layers	6.01 hrs	3.42 hrs	2.13 mins	14.8%	11%
	5 Layers	6.22 hrs	3.45 hrs	2.20 mins	15.8%	12.1%

5.2.1. Execution Time

We formulated execution time per image, based on the total execution time, since our task of image creation from network traffic is an Aperiodic Task [71]. From the collated results, the new NeT2I and I2NeT algorithms are consistent with our previous study [26].

5.2.2. CPU Usage

Given the global interpreter lock and single threaded execution of Python code, we observed that the algorithms NeT2I, and I2NeT used 100% of the CPU resources. However, due to their light weight and deterministic logic, NeT2I and I2NeT algorithms utilised the CPU for a smaller window as seen in the total execution time in Table 2, compared to [57], when it was evaluated against NeT2I in [26], releasing the CPU for other tasks. This efficient used resources and allows for a rapid completion, thus freeing the CPU for other processes in an MEC environment. High CPU usage during active execution is a hallmark of an efficient algorithm design, as it indicates reduced and minimal idle time, illustrating an efficient use of available computational resources [70].

5.2.3. Memory Utilisation

NeT2I utilised 19% of memory on average across the three datasets, consistent with the findings at [26]. This low memory footprint is attributed to the algorithms deterministic approach and the absence of large intermediate data structures. The light-weight encoding operations such as binary serialisation, struct packing, and pixel generation, and mapping makes NeT2I ideal to launch in a resource-constrained environment. The higher utilisation of 20% by I2NeT can be attributed towards the overhead generated by image loading and the initial processing of NumPy arrays for RGB extraction. Although a marginal increase in memory utilisation was recorded, memory leaks or unbounded copies were not recorded, making the decoding I2NeT suitable for resource-constrained edge devices.

5.2.4. GPU Utilisation

During the training, validation, and testing phases, data relating to GPU usage was collected. The data can be seen in Table 3. The observed results and usage patterns revealed a dramatic divergence. The variant developed using the TensorFlow framework, namely CiNeT-TF exhibited a high GPU usage consistently with an average of 99.9% across its application to the three datasets (for layers 1 to 5). This demonstrates a highly efficient execution in which the GPU is nearly saturated during the training and validation process, minimizing idle time and maximum throughput. However, this high utilisation attributes to a cost, where frequent out-of-memory (OOM) errors occurring, when employing larger batch sizes within deeper architectures (4 to 5 layers), indicating a resource boundary. The same has been observed in the works of Gao *et al.* [39], where utilisation constituted a constraint of computation, forcing reduced batch sizes or model complexities.

In contrast, the CiNeT developed using the PyTorch framework (CiNeT -PT) exhibited a low GPU utilisation across its application to the three datasets and all applications (layers 1 to 5). While this may constitutes a performance deficiency, it reflects the fundamental difference between frameworks, where reproducibility and stability are crucial over optimal hardware usage [22]. Unlike the TensorFlow variant of CiNeT, operating at the edge of GPU memory which resulted in OOM errors, the PyTorch variant prevents resource contention and system instability. As stated in the work of Sencan *et al.* [72], such inefficiencies are common in real-world workloads where GPU usage remains low despite high resource availability. Thus suggesting that underutilisation of resources, computation, or memory can be construed as a deliberate design choice to enhance fault tolerance in production environments where crash avoidance is crucial and predictable behaviour sought, as opposed to unpredictability and OOM errors due to saturation [73,74]. Table 3 also shows the memory utilisation of CiNeT algorithm across the two variants with varying architectural depths. Initial observations, reveal that the iterations of CiNeT-TF have consumed more memory, ranging from 26.9% to 30% of the system's total memory allocation, compared to the CiNeT-PT variants, utilising only 8.4% to 12.1%. The disparity in memory utilisation is attributed to the memory management strategy of the underlying framework, where TensorFlow tend to reserve larger memory blocks at the start of execution for graph construction and data handling, contributing to a higher memory utilisation. Conversely, PyTorch employs a more dynamic on-demand memory allocation process, where memory is allocated as tensors during the training phase, and when these tensors are out of scope, the allotted memory is released back to the system through efficient and prompt reference counting and garbage collection mechanisms. This process employed by PyTorch led to a more conservative memory footprint in contrast to the memory footprint created by TensorFlow [32].

The, CiNeT-PT variant can therefore be viewed as a failure-averse deployment for the intrusion detection problem, with a priority for stability and energy efficiency. As stated by [75], peak utilisation of resources, memory, and computation is largely proportional to higher energy consumption, and by operating at a lower GPU usage, CiNeT-PT avoids resource saturation, and power demands, whilst maintaining an enhanced fault tolerance, resulting in a reliable and a sustainable choice for real-time security applications.

5.3. Theoretical Analysis (Big-O Notation)

The theoretical analysis of NeT2I, I2NeT, and CiNeT is conducted by employing the Big-O notation. Big-O notation is used to characterise an algorithm as a function of time and space with respect to the input [76].

5.3.1. Theoretical Complexity for NeT2I

In evaluating NeT2I, Let n denote the number of rows in the input csv, d the number of features, and p the dimension of the output image.

- Reading the CSV file is $O(n \cdot d)$, as each of rows must be scanned and passed across columns.
- The nested loop which iterates over each row and feature, results $O(n \cdot d)$ iterations.
  - Within the loop, each data entry is encoded using $O(1)$
  - Therefore, processing one row is $O(d)$ and for n rows, $O(n \cdot d)$
- Image generation of p by p pixels, where p is the number of RGB stripes, will result in p^2
- As the pixels are of a fixed size and doesn't scale with the number of rows or features, $O(p^2) = O(1)$

Therefore, we can state that the total time complexity ($T(n,d)$)

$$T(n,d) = O(n \cdot d) + O(1) = O(n \cdot d) \tag{1}$$

For the space complexity $S(n,d)$, we can state that

- The input as per the above $O(n \cdot d)$
- The output image, as per the above $O(p^2) = O(1)$

Hence,

$$S(n,d) = O(n \cdot d) \tag{2}$$

Therefore, we can state that NeT2I algorithm belong to the class of polynomial-time bounded by $O(n \cdot d)$, making it suitable for real-time applications.

5.3.2. Theoretical Complexity for I2NeT

Similarly, applying the Big-O notation for I2NeT, let m denote the number of images, p the number of RGB pixel stripes, and d the number of features,

- Discovery of images with $O(m)$ and sorting m files $O(mlogm)$
- Loading the JSON file as $O(1)$
- Image decoding and RGB extraction, involves iterating over p rows in the image, resulting $O(p)$
- Employing the JSON file, calculation of the pixel count, with d as the number of features, which is constant, resulting a $O(1)$
- Value reconstruction similar to value encoding is $O(1)$
- For one image $T(1,p) = O(p)$ with m images, $T(m,p) = O(m \cdot p)$. With $O(mlogm)$ cost being smaller, total time complexity can be stated using

$$T(m,p) = O(m \cdot p) \tag{3}$$

With regards to the space complexity of I2NeT,

- The list of images $O(m)$
- Extracting RGB pixel data and the reconstruction for a row $O(p)$
- Hence, the final CSV $O(m \cdot p)$

Therefore, we can state that the space complexity ($S(m,p)$)

$$S(m,p) = O(m \cdot p) \tag{4}$$

Similar to NeT2I, the decoding algorithm I2NeT algorithm also belong to the class of polynomial-time, bounded by $O(m \cdot p)$, As stated by [76], polynomial time is regarded as tractable or efficiently solvable problems.

5.3.3. Theoretical Complexity for CiNeT

When applying the theoretical complexity for CiNeT, training, validation, and testing phases of the algorithm must be taken into consideration. Let E denotes the epochs, N_{train} the number of training samples, N_{val} the validation samples, and F the number of floating-point operations for a single forward pass through the network. This theoretical application is embedded into the standard methodology for training, validating, and testing a DL model [39,77].

- For each sample, the model conducts a forward pass through convolutional layers and fully connected layers. As the number of operations is determined by the model architecture and the operations per sample is constant, the forward pass is $O(F)$
- Similarly, the back propagation can be construed as being approximately proportional to the above, hence it is also $O(F)$
- The optimizer updates the weights and this can be $O(P)$, where P is the number of parameters.
- Above steps are repeated for each sample resulting N_{train} times per epoch, with the loop being repeated E times.

Therefore, we can state the total training time (T_{train}) is

$$T_{train} = O(E \cdot N_{train} \cdot (F + F + P)) = O(E \cdot N_{train} \cdot F) \tag{5}$$

During the validation and the testing phase, the model requires a forward pass $O(F)$, for each N, where N_{val} for validation and N_{test} for testing, resulting a $T_{inference}$

$$T_{inference} = O(N \cdot F) \tag{6}$$

For the space complexity S, we can state

$$S = O(P + B \cdot F_{activation}) \tag{7}$$

where P is the total number of parameters, batch size B, and the number of floating-point values in the activation map $F_{activation}$ created during the forward pass.

5.4. Training and Validation

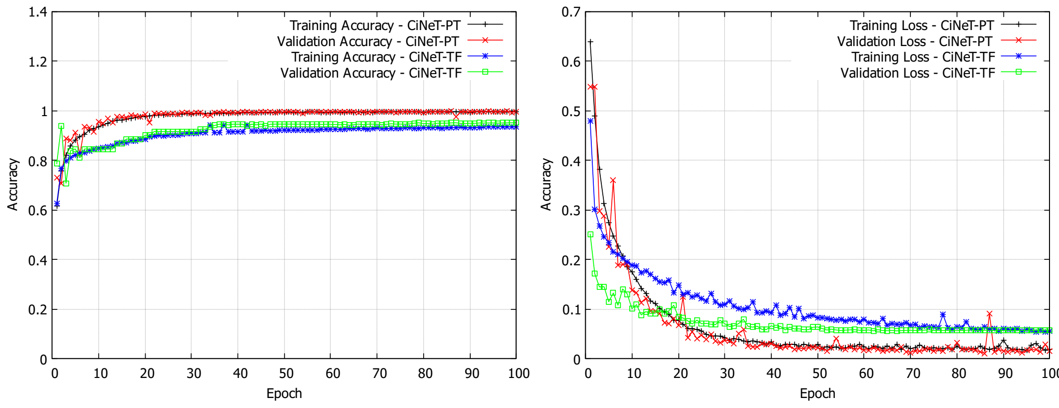
Table 4 presents the validation accuracy acquired during the training and validation phases of the CiNeT algorithm across the three datasets, employing both variants. The collated results demonstrate that CiNeT-PT outperformed CiNeT-TF across all datasets and architectures, with the highest validation accuracy reaching 99.2%. Conversely, CiNeT-TF achieved the highest validation accuracy of 97.1%. CiNeT-PT achieved the highest validation accuracy when the architecture contained 4 layers, whereas the CiNeT-TF achieved the highest validation accuracy with 3 layers. This finding corroborate the initial work of [20], where the accuracy of their model developed in TensorFlow degraded accuracy and precision when the number of layers exceeded three, suggesting architectural depth and sensitivity to accuracy.

Table 4. Validation Accuracy of CiNeT Variants.

Dataset	**CiNeT - TF**					**CiNeT - PT**				
	1L	**2L**	**3L**	**4L**	**5L**	**1L**	**2L**	**3L**	**4L**	**5L**
InSDN	94.3	95.8	97.1	96.5	95.9	96.7	97.8	98.4	99.1	98.6
UNSW-NB15	93.6	95.0	96.8	96.0	95.3	95.9	97.1	98.0	98.9	98.3
ToN-IoT	94.8	96.0	97.2	96.7	96.1	97.0	98.0	98.7	99.2	98.8

As evident by the Figure 4a,b, the CiNeT-PT (4 Layer) model acting on the ToN_IoT dataset exhibits a steady increase in training and validation accuracy, suggesting effective learning with minimal overfitting. Conversely, the CiNeT-TF (3 Layer) model acting on ToN_IoT shows fluctuations in both accuracy and loss, with the validation curve displaying a slower convergence, suggesting potential numerical issues during training.

Albeit marginal, this performance gap suggests that CiNeT-PT implementation benefitted from the efficient training process, better gradient handling, consistent weight initialisation, efficient memory usage (as shown in Table 3), numerical instabilities, as suggested by the authors at [78,79]. These studies further comment that TensorFlow models are prone to silent bugs and incorrect gradient computations that can degrade model accuracy. In contrast, PyTorch's dynamic computation graph and programming model, offer greater transparency and control during the training and validation stages, enabling a reliable and more accurate model with a deeper architecture, making it more suitable for complex multi-class intrusion detection problem.



(**a**) Training and Validation Accuracy (**b**) Training and Validation Loss

Figure 4. Training and Validation Data from the TON_IoT dataset.

5.5. *Evaluation of Detection*

The performance of the two CiNeT algorithms was evaluated using the results shown in Tables 5 and 6. Table 5 presents the accuracy acquired during class-wise tests for both CiNeT-TF(3 Layer) and CiNeT-PT (4 Layer) for each of the three datasets, with Table 6 providing a more granular analysis of the best model (CiNeT-PT(4 Layer)), presenting the accuracy (Acc), precision (Prec), Recall (Rec), and F1-score (F1) for each traffic class.

Table 5. Test Accuracy of CiNeT Variants Across Datasets and Traffic Classes.

Traffic Class	**InSDN**		**UNSW-NB15**		**ToN-IoT**	
	CiNeT-TF	**CiNeT-PT**	**CiNeT-TF**	**CiNeT-PT**	**CiNeT-TF**	**CiNeT-PT**
	(3L)	(4L)	(3L)	(4L)	(3L)	(4L)
Normal	98.5	99.0	96.9	97.5	99.1	99.4
DDoS	98.7	99.3	97.2	98.1	99.0	99.5
DoS	97.5	98.4	95.8	96.9	98.1	98.8
Reconnaissance	96.3	97.5	94.6	95.7	97.0	97.8
Exploits	95.1	96.4	93.2	94.5	95.9	96.7
Backdoor	94.0	95.2	91.8	93.0	94.8	95.6

Table 6. Confusion Matrix for CiNeT-PT (4 Layer).

Traffic Class	**InSDN**				**UNSW-NB15**				**ToN-IoT**			
	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1
Normal	99.0	98.7	99.2	98.9	97.5	97.1	97.8	97.4	99.4	99.2	99.5	99.3
DDoS	99.3	99.1	99.4	99.2	98.1	97.8	98.3	98.0	99.5	99.3	99.6	99.4
DoS	98.4	98.0	98.7	98.3	96.9	96.5	97.2	96.8	98.8	98.5	99.0	98.7
Reconnaissance	97.5	97.0	97.9	97.4	95.7	95.2	96.0	95.6	97.8	97.4	98.1	97.7
Exploits	96.4	95.9	96.8	96.3	94.5	94.0	94.9	94.4	96.7	96.3	97.0	96.6
Backdoor	95.2	94.7	95.6	95.1	93.0	92.5	93.4	92.9	95.6	95.2	95.9	95.5

Further to the results found in Table 4, Table 5 confirms superior detection for the CiNeT-PT (4 Layer) algorithm. For the ToN-IoT dataset (a highly sought after dataset with diverse and realistic data), CiNeT-PT achieved an exceptional accuracy of 99.4% for normal traffic and 99.5% for DDoS traffic, suggesting an improvement from 99.1% and 99% for the CiNeT-TF algorithm. The performance variation is observable for each of the cases investigated.

As shown in the Table 6, the CiNeT-PT variant achieves an F1-score of 99.4%, 99.3% for precision, and 99.6% for recall, over DDoS traffic, suggesting that the model performed with high precision and accuracy, suggesting a lower false alarms rate. Similarly, for other attack types such as backdoor, exploits, DoS, and reconnaissance the CiNeT-PT variant maintained high accuracy and precision, demonstrating the robust application of CiNeT-PT for detecting a wide range of attacks. Achieving an accuracy of 100% would signify an overfitting of data in neural networks, the achieved accuracy can be considered as having achieved cross-validation in our methodology due to augmentation, regularisation, and increase of varied training data [80].

6. Discussion

In the context of this research, we present a comprehensive performance evaluation of the CiNeT algorithm, a novel DL algorithm capable of automated attack class detections a new IDS that operates on images encoded and decoded via a NeT2I-I2NeT pipeline. Our work has improved and extended the pipeline through which we can encode a broader range of network features, including IPv6 and floating point numbers without loss, as RGB images, allowing for a rigorous comparative analysis of the CiNeT algorithm deployed using TensorFlow and PyTorch. The evaluation focuses not only on critical performance such as accuracy, prediction, recall, and F1-score but also theoretical and empirical evaluation over its computational complexity, and resource utilisation, providing a holistic view.

The computational complexity of CiNeT algorithms reveal a fundamental trade-off between resource usage and model robustness. The TensorFlow variant achieving peak GPU usage demonstrated hardware efficiency, but fragility, and error prone behaviour during training and validation. Whereas the PyTorch variant exhibited low GPU usage demonstrated exceptional stability, robustness, and consistent performance. This observation highlighted an important insight, high GPU usage may constrain performance, efficiency, and sustainability. The CiNeT-PT 4 Layer model therefore, produces a superior performance with prioritised stability, reproducibility, and sustainability, without compromising accuracy of detection.

By careful observation, it is evident that the CiNeT-PT variant has outperformed the CiNeT-TF variant. As discussed previously, this performance gain can be attributed to the stable and efficient training process, gradient handling, weight initialisation, and resource usage. As the CiNeT-PT (4 Layer) model was able to detect with high precision and minimal false positives, we can state that the model is able to generalise exceptionally well, as an IDS.

Finally, across the three datasets, the ToN-IoT dataset outperformed both the UNSW-NB15 and InSDN datasets. This can be attributed to the dataset quality and diversity of traffic when evaluating intrusion detection problems. This aligns with recent studies that have suggested ToN-IoT is better suited for IDS's equipped with DL models. Following our research, we can state that the CiNeT-PT (4 layer) variant represents a significant advancement in the field of network security.

7. Conclusions and Future Work

In this paper, we presented CiNeT, a novel CNN based IDS capable of detecting multiple classes of malicious traffic, leveraging both PyTorch and TensorFlow. Alongside CiNeT, we also introduced an advanced pipeline of the NeT2I and I2NeT capable of conducting a bijective encoding - decoding process, that can leverage the CiNeT detection algorithm. Thus allowing the utilisation of CNN for spatial pattern recognition in network flows, maintaining full traceability from detection to packet level information, providing a step towards intrusion prevention. This method solves the “black-box" problem commonly associated with DL applications in cyber security [18].

Two variants of the CiNeT algorithms were evaluated across three datasets, UNSW NB-15, InSDN, and TON_IoT, with a focus on multi-class classification in intrusion detection. Our results demonstrated that CiNeT-PT (4 Layer) achieved superior accuracy of 99.5%, outperforming the CiNeT-TF architectures. CiNeT-PT outperformed CiNeT-TF in training time, validation time, RAM utilisation, and GPU usage, making the application of CiNeT-PT suitable for systems requiring edge detection [81].

Currently, we are working towards the deployment of CiNeT in our 5G testbed, extending the research at [81], integrating control and data plane programmability using technologies such as Software Defined Networking and Programming Protocol independent Packet Processing [82], to realise a real-time CNN based NIDS. This work draws upon the foundational framework of Real-Time Deep Learning based NIDS (RTDL-NIDS) [83], where NeT2I-CNN-I2NeT pipeline was successfully implemented and evaluated within a 5G-Multi-Access Edge Computing (MEC) mobile telecommunication testbed. This work demonstrates that the aforementioned pipeline enables intrusion detection in real-time as opposed to a ‘desk-approach’. The ongoing research is aimed at implementing the CiNeT-PT (4 Layer) within a 5G testbed, paving the way for an intelligent and automated security implementation.

Author Contributions: Conceptualization, O.F. J.S. and H.X.; methodology, O.F. J.S. and H.X.; software, O.F.; validation, O.F., J.S. and H.X.; formal analysis, O.F. J.S. and H.X.; investigation, O.F. J.S. and H.X.; resources, O.F.; data curation, O.F. J.S. and H.X.; writing—original draft preparation, O.F. J.S. and H.X.; writing—review and editing, O.F. J.S. and H.X.; visualization, O.F; supervision, J.S. and H.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding

Data Availability Statement: The original contributions presented in this study are included in the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Parkvall, S.; Dahlman, E.; Furuskar, A.; Frenne, M. NR: The new 5G radio access technology. *IEEE Communications Standards Magazine* **2018**, *1*, 24–30.
2. Lin, X. The bridge toward 6G: 5G-Advanced evolution in 3GPP Release I9. *IEEE Communications Standards Magazine* **2025**, *9*, 28–35.
3. Chen, W.; Lin, X.; Lee, J.; Toskala, A.; Sun, S.; Chiasserini, C.F.; Liu, L. 5G-advanced toward 6G: Past, present, and future. *IEEE journal on selected areas in communications* **2023**, *41*, 1592–1619.
4. Lin, X. An overview of 5G advanced evolution in 3GPP release 18. *IEEE Communications Standards Magazine* **2022**, *6*, 77–83.
5. Eleftherakis, S.; Giustiniano, D.; Kourtellis, N. SoK: Evaluating 5G-Advanced Protocols Against Legacy and Emerging Privacy and Security Attacks. In Proceedings of the 18th ACM Conference on Security and Privacy in Wireless and Mobile Networks, 2025, pp. 196–210.
6. Michaelides, S.; Lenz, S.; Vogt, T.; Henze, M. Secure integration of 5G in industrial networks: State of the art, challenges and opportunities. *Future Generation Computer Systems* **2025**, *166*, 107645.
7. Bodenhausen, J.; Sorgatz, C.; Vogt, T.; Grafflage, K.; Rötzel, S.; Rademacher, M.; Henze, M. Securing wireless communication in critical infrastructure: challenges and opportunities. In Proceedings of the International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services. Springer, 2023, pp. 333–352.
8. Taylor, P. Mobile phone subscriptions worldwide 2024, 2025.
9. Ericsson. Mobile Data Traffic Forecast – Ericsson Mobility Report, 2024. Accessed: 2025-04-22.
10. Wang, Y.; Wang, J. Research on Computer Network Big Data Security Defense System Based on Support Vector Machine and Deep Learning. In Proceedings of the 2025 IEEE International Conference on Electronics, Energy Systems and Power Engineering (EESPE). IEEE, 2025, pp. 386–392.
11. Ma, J.; Li, S. The construction method of computer network security defense system based on multisource big data. *Scientific Programming* **2022**, *2022*, 7300977.

12. Apruzzese, G.; Laskov, P.; Montes de Oca, E.; Mallouli, W.; Brdalo Rapa, L.; Grammatopoulos, A.V.; Di Franco, F. The role of machine learning in cybersecurity. *Digital Threats: Research and Practice* **2023**, *4*, 1–38.
13. Ozkan-Okay, M.; Akin, E.; Aslan, Ö.; Kosunalp, S.; Iliev, T.; Stoyanov, I.; Beloev, I. A comprehensive survey: Evaluating the efficiency of artificial intelligence and machine learning techniques on cyber security solutions. *IEEe Access* **2024**, *12*, 12229–12256.
14. Ali, M.L.; Thakur, K.; Schmeelk, S.; Debello, J.; Dragos, D. Deep Learning vs. Machine Learning for Intrusion Detection in Computer Networks: A Comparative Study. *Applied Sciences* **2025**, *15*, 1903.
15. Rosenberg, I.; Shabtai, A.; Elovici, Y.; Rokach, L. Adversarial machine learning attacks and defense methods in the cyber security domain. *ACM Computing Surveys (CSUR)* **2021**, *54*, 1–36.
16. Kaloudi, N.; Li, J. The ai-based cyber threat landscape: A survey. *ACM Computing Surveys (CSUR)* **2020**, *53*, 1–34.
17. Chauhan, N.K.; Singh, K. A review on conventional machine learning vs deep learning. In Proceedings of the 2018 International conference on computing, power and communication technologies (GUCON). IEEE, 2018, pp. 347–352.
18. Charmet, F.; Tanuwidjaja, H.C.; Ayoubi, S.; Gimenez, P.F.; Han, Y.; Jmila, H.; Blanc, G.; Takahashi, T.; Zhang, Z. Explainable artificial intelligence for cybersecurity: a literature survey. *Annals of Telecommunications* **2022**, *77*, 789–812.
19. Al-Turaiki, I.; Altwaijry, N. A convolutional neural network for improved anomaly-based network intrusion detection. *Big Data* **2021**, *9*, 233–252.
20. Kim, J.; Kim, J.; Kim, H.; Shim, M.; Choi, E. CNN-based network intrusion detection against denial-of-service attacks. *Electronics* **2020**, *9*, 916.
21. Nazarri, M.N.A.A.; Yusof, M.H.M.; Almohammedi, A.A. Generating network intrusion image through IGTD algorithm for CNN classification. In Proceedings of the 2023 3rd International Conference on Computing and Information Technology (ICCIT). IEEE, 2023, pp. 172–177.
22. Shahriari, M.; Ramler, R.; Fischer, L. How do deep-learning framework versions affect the reproducibility of neural network models? *Machine Learning and Knowledge Extraction* **2022**, *4*, 888–911.
23. Moustafa, N.; Slay, J. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In Proceedings of the 2015 Military Communications and Information Systems Conference (MilCIS), 2015, pp. 1–6.
24. Elsayed, M.S.; Le-Khac, N.A.; Jurcut, A.D. InSDN: A novel SDN intrusion dataset. *IEEE Access* **2020**, *8*, 165263–165284.
25. Moustafa, N. The ton_iot datasets.
26. Fernando, O.A.; Xiao, H.; Spring, J. New Algorithms for the Detection of Malicious Traffic in 5G-MEC. In Proceedings of the 2023 IEEE Wireless Communications and Networking Conference (WCNC), 26–29 March 2023, Glasgow, Scotland, UK. IEEE, 2023.
27. Adadi, A.; Berrada, M. Peeking inside the black-box: a survey on explainable artificial intelligence (XAI). *IEEE access* **2018**, *6*, 52138–52160.
28. Tareq, I.; Elbagoury, B.M.; El-Regaily, S.; El-Horbaty, E.S.M. Analysis of ton-iot, unw-nb15, and edge-iiot datasets using dl in cybersecurity for iot. *Applied Sciences* **2022**, *12*, 9572.
29. Kolhar, M.; Aldossary, S.M. A deep learning approach for securing IoT infrastructure with emphasis on smart vertical networks. *Designs* **2023**, *7*, 139.
30. Yin, C.; Zhu, Y.; Fei, J.; He, X. A deep learning approach for intrusion detection using recurrent neural networks. *IEEE Access* **2017**, *5*, 21954–21961.
31. Shaheen, F.; Verma, B.; Asafuddoula, M. Impact of automatic feature extraction in deep learning architecture. In Proceedings of the 2016 International conference on digital image computing: techniques and applications (DICTA). IEEE, 2016, pp. 1–8.
32. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* **2019**, *32*.
33. Dash, N.; Chakravarty, S.; Rath, A.K.; Giri, N.C.; AboRas, K.M.; Gowtham, N. An optimized LSTM-based deep learning model for anomaly network intrusion detection. *Scientific Reports* **2025**, *15*, 1554.
34. Thakkar, A.; Kikani, N.; Geddam, R. Fusion of linear and non-linear dimensionality reduction techniques for feature reduction in LSTM-based Intrusion Detection System. *Applied Soft Computing* **2024**, *154*, 111378.

35. Bukhari, S.M.S.; Zafar, M.H.; Abou Houran, M.; Moosavi, S.K.R.; Mansoor, M.; Muaaz, M.; Sanfilippo, F. Secure and privacy-preserving intrusion detection in wireless sensor networks: Federated learning with SCNN-Bi-LSTM for enhanced reliability. *Ad Hoc Networks* **2024**, *155*, 103407.
36. Hossain, M.D.; Inoue, H.; Ochiai, H.; Fall, D.; Kadobayashi, Y. LSTM-based intrusion detection system for in-vehicle can bus communications. *Ieee Access* **2020**, *8*, 185489–185502.
37. G. Lira, O.; Marroquin, A.; To, M.A. Harnessing the advanced capabilities of llm for adaptive intrusion detection systems. In Proceedings of the International Conference on Advanced Information Networking and Applications. Springer, 2024, pp. 453–464.
38. Adjewa, F.; Esseghir, M.; Merghem-Boulahia, L.; Kacfah, C. Llm-based continuous intrusion detection framework for next-gen networks. In Proceedings of the 2025 International Wireless Communications and Mobile Computing (IWCMC). IEEE, 2025, pp. 1198–1203.
39. Gao, Y.; He, Y.; Li, X.; Zhao, B.; Lin, H.; Liang, Y.; Zhong, J.; Zhang, H.; Wang, J.; Zeng, Y.; et al. An empirical study on low gpu utilization of deep learning jobs. In Proceedings of the Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
40. Amini, M.; Asemian, G.; Kantarci, B.; Ellement, C.; Erol-Kantarci, M. Deep Fusion Intelligence: Enhancing 5G Security Against Over-the-Air Attacks. *IEEE Transactions on Machine Learning in Communications and Networking* **2025**.
41. Rajabi, S.; Asgari, S.; Jamali, S.; Fotohi, R. An intrusion detection system using the artificial neural network-based approach and firefly algorithm. *Wireless Personal Communications* **2024**, *137*, 2409–2440.
42. Alzubi, O.A.; Alzubi, J.A.; Qiqieh, I.; Al-Zoubi, A. An IoT intrusion detection approach based on salp swarm and artificial neural network. *International Journal of Network Management* **2025**, *35*, e2296.
43. Azzaoui, H.; Boukhamla, A.Z.E.; Perazzo, P.; Alazab, M.; Ravi, V. A lightweight cooperative intrusion detection system for rpl-based iot. *Wireless Personal Communications* **2024**, *134*, 2235–2258.
44. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *nature* **2015**, *521*, 436–444.
45. Mohammadpour, L.; Ling, T.C.; Liew, C.S.; Aryanfar, A. A survey of CNN-based network intrusion detection. *Applied Sciences* **2022**, *12*, 8162.
46. Elouardi, S.; Motii, A.; Jouhari, M.; Amadou, A.N.H.; Hedabou, M. A survey on Hybrid-CNN and LLMs for intrusion detection systems: Recent IoT datasets. *IEEE Access* **2024**.
47. Wang, L.H.; Dai, Q.; Du, T.; Chen, L.f. Lightweight intrusion detection model based on CNN and knowledge distillation. *Applied Soft Computing* **2024**, *165*, 112118.
48. Abed, R.A.; Hamza, E.K.; Humaidi, A.J. A modified CNN-IDS model for enhancing the efficacy of intrusion detection system. *Measurement: Sensors* **2024**, *35*, 101299.
49. El-Ghamry, A.; Darwish, A.; Hassanien, A.E. An optimized CNN-based intrusion detection system for reducing risks in smart farming. *Internet of Things* **2023**, *22*, 100709.
50. Yang, L.; Shami, A. A transfer learning and optimized CNN based intrusion detection system for Internet of Vehicles. In Proceedings of the ICC 2022-IEEE International Conference on Communications. IEEE, 2022, pp. 2774–2779.
51. Kim, I.; Chung, T.M. Malicious-Traffic Classification Using Deep Learning with Packet Bytes and Arrival Time. In Proceedings of the International Conference on Future Data and Security Engineering. Springer, 2020, pp. 345–356.
52. Wang, Z.; Ghaleb, F.A.; Zainal, A.; Siraj, M.M.; Lu, X. An efficient intrusion detection model based on convolutional spiking neural network. *Scientific Reports* **2024**, *14*, 7054.
53. Yue, C.; Wang, L.; Wang, D.; Duo, R.; Nie, X. An ensemble intrusion detection method for train Ethernet consist network based on CNN and RNN. *IEEE Access* **2021**, *9*, 59527–59539.
54. Liu, Y.; Kang, J.; Li, Y.; Ji, B. A Network Intrusion Detection Method Based on CNN and CBAM. In Proceedings of the IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2021, pp. 1–6.
55. Ding, Y.; Zhai, Y. Intrusion detection system for NSL-KDD dataset using convolutional neural networks. In Proceedings of the Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence, 2018, pp. 81–85.
56. Shapira, T.; Shavitt, Y. Flowpic: Encrypted internet traffic classification is as easy as image recognition. In Proceedings of the IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2019, pp. 680–687.
57. Janabi, A.H.; Kanakis, T.; Johnson, M. Convolutional Neural Network Based Algorithm for Early Warning Proactive System Security in Software Defined Networks. *IEEE Access* **2022**, *10*, 14301–14310.

58. Farrukh, Y.A.; Wali, S.; Khan, I.; Bastian, N.D. Senet-i: An approach for detecting network intrusions through serialized network traffic images. *Engineering Applications of Artificial Intelligence* **2023**, *126*, 107169.
59. Zilberman, A.; Dvir, A.; Stulman, A. IPv6 Routing Protocol for Low-Power and Lossy Networks Security Vulnerabilities and Mitigation Techniques: A Survey. *ACM Computing Surveys* **2025**, *57*, 1–77.
60. Team, I. IPv4 to IPv6 migration: Complete Enterprise Guide for 2025.
61. Riedy, J.; Demmel, J. Augmented arithmetic operations proposed for IEEE-754 2018. In Proceedings of the 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH). IEEE, 2018, pp. 45–52.
62. Fernando, O. OMESHF/NeT2I: Net2I (network to image) is a python package for converting network traffic data into image representations.
63. Fernando, O. net2i.
64. Fernando, O. OMESHF/I2NeT: I2NeT (image to network) is the reverse companion to net2i. It decodes RGB images created from network traffic data back into structured tabular form.
65. Fernando, O. i2net.
66. Darst, B.F.; Malecki, K.C.; Engelman, C.D. Using recursive feature elimination in random forest to account for correlated variables in high dimensional data. *BMC genetics* **2018**, *19*, 65.
67. Sedgewick, R.; Wayne, K. *Algorithms: Part I*; Addison-Wesley Professional, 2014.
68. Blackburn, S.M.; Garner, R.; Hoffmann, C.; Khang, A.M.; McKinley, K.S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S.Z.; et al. The DaCapo benchmarks: Java benchmarking development and analysis. In Proceedings of the Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2006, pp. 169–190.
69. Barbarossa, S.; Sardellitti, S.; Ceci, E. Joint Communications and Computation: A Survey on Mobile Edge Computing. *IEEE Signal Processing Magazine* **2018**, *35*, 36–58. https://doi.org/10.1109/MSP.2018.2856178.
70. Gorelick, M.; Ozsvald, I. *High Performance Python: Practical Performant Programming for Humans*; O'Reilly Media, 2020.
71. Buttazzo, G.C. *Hard real-time computing systems: predictable scheduling algorithms and applications*; Vol. 24, Springer Science & Business Media, 2011.
72. Sencan, E.; Kulkarni, D.; Coskun, A.; Konate, K. Analyzing GPU Utilization in HPC Workloads: Insights from Large-Scale Systems. In Proceedings of the Proceedings of the Practice and Experience in Advanced Research Computing (PEARC '25), New York, NY, USA, 2025; pp. 1–10. https://doi.org/10.1145/3708035.3736010.
73. Ganguly, D.; Mofrad, M.H.; Znati, T.; Melhem, R.; Lange, J.R. Harvesting Underutilized Resources to Improve Responsiveness and Tolerance to Crash and Silent Faults for Data-Intensive Applications. In Proceedings of the 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017, pp. 536–543. https://doi.org/10.1109/CLOUD.2017.74.
74. Ananthanarayanan, G.; Ghodsi, A.; Shenker, S.; Stoica, I. Why let resources idle? Aggressive cloning of jobs with Dolly. In Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12), 2012.
75. Govindan, S.; Sivasubramaniam, A.; Urgaonkar, B. Benefits and limitations of tapping into stored energy for datacenters. In Proceedings of the Proceedings of the 38th annual international symposium on Computer architecture, 2011, pp. 341–352.
76. Papadimitriou, C.H. Computational complexity. In *Encyclopedia of computer science*; 2003; pp. 260–265.
77. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press, 2016.
78. Tambon, F.; Nikanjam, A.; An, L.; Khomh, F.; Antoniol, G. Silent Bugs in Deep Learning Frameworks: An Empirical Study of Keras and TensorFlow. *arXiv preprint arXiv:2112.13314* **2021**.
79. Zhang, Y.; Chen, Y.; Cheung, S.C.; Xiong, Y.; Zhang, L. An Empirical Study on TensorFlow Program Bugs. In Proceedings of the Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2018, p. 129–140.
80. Zhang, H.; Zhang, L.; Jiang, Y. Overfitting and underfitting analysis for deep learning based end-to-end communication systems. In Proceedings of the 2019 11th International Conference on Wireless Communications and Signal Processing (WCSP). IEEE, 2019, pp. 1–6.
81. Fernando, O.A.; Xiao, H.; Spring, J. Developing a Testbed with P4 to Generate Datasets for the Analysis of 5G-MEC Security. In Proceedings of the 2022 IEEE Wireless Communications and Networking Conference (WCNC). IEEE, 2022, pp. 2256–2261.

82. Fernando, O.A.; Xiao, H.; Spring, J.; Che, X. A Performance Evaluation for Software Defined Networks with P4. *Network* **2025**, *5*, 21.
83. Fernando, O.A. Real-Time Application of Deep Learning to Intrusion Detection in 5G-Multi-Access Edge Computing **2024**.