Received XX Month, XXXX; revised XX Month, XXXX; accepted XX Month, XXXX; Date of publication XX Month, XXXX; date of current version XX Month, XXXX.

Digital Object Identifier 10.1109/TMLCN.2022.1234567

Robust and Lightweight Modeling of IoT Network Behaviors from Raw Traffic Packets

Aleksandar Pasquini[1], Rajesh Vasa[1], Irini Logothetis [1], Hassan Habibi Gharakheili[2], Alexander Chambers[3], and Minh Tran[3]

[1]A2I2, Deakin University, Geelong, VIC 3220, Australia
[2]School of Electrical Engineering and Telecommunications, University of New South Wales, Sydney, NSW 2052, Australia
[3]Information Sciences Division, Defence Science & Technology Group, Edinburgh, SA, Australia

Corresponding author: Aleksandar Pasquini (email: aleksandar.pasquini@deakin.edu.au).

This submission extends and builds upon our paper presented at the IEEE Globecom 2023 workshop on AI/ML for Edge/Fog Networks (A4E) [1].
This research is supported by the Commonwealth of Australia as represented by the Defence Science and Technology Group of the Department of Defence.

ABSTRACT

Machine Learning (ML)-based techniques are increasingly used for network management tasks, such as intrusion detection, application identification, or asset management. Recent studies show that neural network-based traffic analysis can achieve performance comparable to human feature-engineered ML pipelines. However, neural networks provide this performance at a higher computational cost and complexity, due to high-throughput traffic conditions necessitating specialized hardware for real-time operations. This paper presents lightweight models for encoding characteristics of Internet-of-Things (IoT) network packets. (1) We present two strategies to encode packets (regardless of their size, encryption, and protocol) to integer vectors: a shallow lightweight neural network and compression. With a public dataset containing about 8 million packets emitted by 22 IoT device types, we show the encoded packets can form complete (up to 80%) and homogeneous (up to 89%) clusters; (2) We demonstrate the efficacy of our generated encodings in the downstream classification task and quantify their computing costs. We train three multi-class models to predict the IoT class given network packets and show our models can achieve the same levels of accuracy (94%) as deep neural network embeddings but with computing costs up to 10 times lower; (3) We examine how the amount of packet data (headers and payload) can affect the prediction quality. We demonstrate how the choice of Internet Protocol (IP) payloads strikes a balance between prediction accuracy (99%) and cost. Along with the cost-efficacy of models, this capability can result in rapid and accurate predictions, meeting the requirements of network operators.

INDEX TERMS Feature Engineering, Packet Embedding, Network Behavior Characterization, IoT Devices

I. Introduction

THE proliferation of Internet-of-Things (IoT) devices has impacted the complexity of modern networks. As these devices become integral to both consumer and industrial environments, identifying and classifying them is essential for effective network management, security monitoring, and anomaly detection in the IoT domain [2].

Device classification allows network administrators to understand the types of devices connected to their networks, enabling tasks such as applying appropriate security policies, detecting unauthorized devices, and optimizing network resources. However, traditional device classification techniques rely on static models developed with manual feature engineering [3]–[6], where domain experts analyze network traffic to identify distinctive features. This approach is time intensive, requires specialized skills and knowledge about the network, and limits the adaptability of models to changing network behaviors [7].

To address these limitations, researchers have explored using traditional methods that do not leverage machine learning (ML) techniques. These approaches utilize signatures for classifying IoT network traffic, such as cloud services [8], accessed domain names [9], or Manufacturer Usage Description (MUD) profiles [10]. While these methods can be effective, they rely on network characteristics or require additional data that may not be readily available in all environments. This paper focuses on ML-based methods that use emitted packets from the device without the need for additional data.

Automated feature extraction methods, especially those leveraging neural networks, have gained prominence as a robust alternative to traditional techniques. Neural networks can automatically learn representations from raw data, capturing intrinsic patterns without human intervention [11]–[13]. These representations can be expressed as dense embeddings. The information content of embeddings is particularly advantageous in the field of network traffic inference, where embeddings have been used to achieve high accuracy (greater than 90%) in tasks such as device classification [14]–[23].

However, existing neural network–based approaches for device classification face two main limitations as they learn representations of packets. First, they may depend on specific types of network packets for input, such as Domain Name System (DNS) packets [19] or application-layer packets [21], which may not always be available, leading to delays or reduced applicability in diverse network settings. Second, they can involve computationally expensive preprocessing steps, such as transforming packets into two-dimensional (2D) grayscale or Red-Green-Blue (RGB) images [14]–[16], which increases resource consumption and thus hinders real-time analysis.

Given these limitations, there is a need for more computationally efficient techniques for IoT device classification that do not rely on specific packet types. This paper focuses on addressing these needs by making the following contributions:

Our **first** contribution establishes two lightweight packet transformation methodologies that accept any type of packet as input. The first method employs a one-dimensional convolutional neural network (1D CNN), which leverages the sequential nature and local patterns inherent in the raw byte values. The second method, inspired by Kolmogorov complexity [24], utilizes the Deflate compression algorithm [25] and Principal Component Analysis (PCA) [26] to reduce the entropy of the raw data. Our **second** contribution demonstrates that by using compression to represent a packet (compressed packet), the classifier performance is comparable to a neural network learnt representation of a packet (dense embedding) at one tenth of the computational cost. We benchmark both of our packet representation methods (compressed packet, dense embedding) with five other approaches by measuring training, inference and memory costs. We evaluate these representations using three IoT classifiers [3]–[5] for device classification, measuring and comparing accuracy and

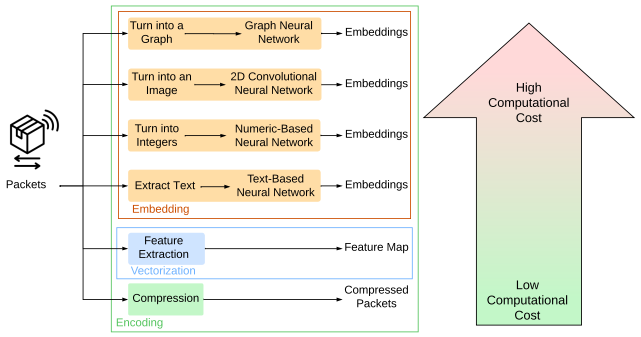


FIGURE 1. Packet information can be encoded, by various techniques, into usable inputs for machine learning models.

classification time. Our **third** contribution develops a trade-off framework that optimizes the use of varying amounts of packet information to maximize prediction quality relative to computational cost. Specifically, we evaluate the impact of different Transmission Control Protocol/Internet Protocol (TCP/IP) layer headers on accuracy by systematically removing them.

The structure of this paper is as follows: §II explains the relationship between encoding and embedding and provides an overview of prior studies in the field of packet embeddings. Our lightweight models are presented and intrinsically evaluated in §III, while the extrinsic device classification task results and cost analysis are shown in §IV. §V examines the trade-off between the amount of information gained from packets versus their computing and latency costs. Finally, the paper concludes in §VI.

II. Background and Related Work

A. Data Encoding for Learning Tasks

Machine Learning (ML) has been used for traffic classification tasks [7], [22], [27]–[29], yet these models seldom use packet bytes as input due to their heterogeneous nature. Instead, the bytes are first encoded. Encoding is the process of converting data from one form to another. This transformation does not have to include data compression. Fig. 1 shows some of the encoding methods and their qualitative costs.

Neural networks can be used to represent a packet. They create these representations by feeding data through its layers and adapting its internal parameters to minimize a predefined loss function. This loss function is crafted to incentivize the model to position similar items nearer to each other in the vector space, thereby capturing the underlying relationships and patterns within the data. These representations from neural networks are referred to as dense embeddings because they encode data as vectors in which every value is non-zero. Each neural network generates a different set of dense embeddings. However, not all network architectures are alike, as some offer a more accurate and cost-efficient set of embeddings than others.

Packets can also be represented by compression-based encoding, which reduces the data redundancy. However, this method does not organize packets with similar concepts closely within the representational space, making it less suitable for classification tasks. Compression algorithms cannot generally capture the subtle patterns needed for accurate concept learning and precise predictions [7].

B. Prior Research on Packet Encoding

Packet embeddings can be categorized based on the necessary packet transformations before executing the packet encoding process. There are four main types: Image, Graph, Text, and Byte. Image-based methods [7], [14], [15] involve converting the packet into an image, while graph-based models [7], [14], [15] transform the packet into a graph representation. Text-based approaches identify "words" in the packet, and then the "words" are embedded using Natural Language Processing (NLP) techniques. These transformations are needed to standardize the packets and utilize various neural network architectures effectively. These processes are computationally demanding due to the differing structures of packets compared to images, words, and graphs.

Additionally, each of these techniques has a different set of assumptions. Image-based models assume spatial locality and translational invariance, meaning that nearby pixels are related and patterns can appear anywhere within the image. Word-based models depend on sequential relationships and contextual dependencies inherent in language. Graph-based models assume data is interconnected through nodes and edges, capturing non-linear and non-sequential relationships.

However, packets do not meet these foundational assumptions and transforming the packet data into images, words, or graphs will not bring those necessary characteristics. They lack the spatial structure of images, the sequential context of language, and the relational connections of graphs. Packets are discrete data units with specific fields that may not have positional or contextual dependencies suitable for these models. Therefore, applying image-, word-, or graph-based neural network models to packet data is misaligned, leading to heightened computational costs for minimal accuracy improvements.

In contrast, the byte methods prove to be more efficient as they retain the input data in its original raw format. However, the efficacy of these methods remains largely unexplored beyond a limited number of protocols. In what follows, we briefly describe each group and highlight the distinguishing aspects of our approach. Table 1 provides a summary of works on traffic encoding methods (ours included), specifically highlighting the protocol of interest, the input data, preprocessing techniques, encoding strategies, downstream inference tasks, and the prediction types.

1) Image Based Techniques

Computer vision techniques have been found to be applicable in network security, specifically in understanding packet data [7], [14]–[16]. These methods leverage a 2D Convolutional Neural Network (2D CNN) to process packet payloads akin to image analysis. All the methods require preprocessing a certain number of bytes into a 2D image, as shown in the Input and Preprocessing columns of Table 1. However, there are different ways to do this preprocessing. For example, HAST-ID [14] one hot encodes the first n bytes of a packet and stacks the generated vectors to form a 2D image. This is similar to [16], where they stacked normalized feature vectors into a matrix. [7] generates 23×23 grayscale images by using the first 784 byte values as pixel values. Authors of [15] create an RGB image instead of a grayscale one. They do this by first dividing the packet into n segments and then forming a $m\times n$ packet matrix. They then rotate the matrix by 90 degrees for the second channel and by another 90 degrees for the third channel.

These methods show high performance (95-99% accuracy) for intrusion detection, but the packets are a sequence of bytes and do not have strong spatial characteristics like images. Packets are structured in a linear sequence, and their order holds information for network analysis, which these representations ignore. Utilizing a 2D CNN designed for grid-like data introduces complexity due to transformations not aligned with the inherent characteristics of packet sequences.

2) Graph Based Techniques

Graph-based methods involve turning packets into graphs, allowing the application of graph neural networks (as seen under the Preprocessing and Encoding Strategies columns in Table 1). This transformation can be done by associating nodes with IP addresses and creating an edge for each packet transmitted between a pair of nodes. For example, [23] models traffic as a bipartite graph where sender nodes are connected to destination TCP port nodes. Another method, presented by [22], transforms each packet into a graph by considering each byte in the packet's sequence as a distinct node. This approach ensures that the packet graph contains no more than 256 nodes, corresponding to the maximum number of unique byte values. The edges in this graph are represented using Pointwise Mutual Information to signify the connections between the packet's nodes.

[22] and [23] report an accuracy of 80-99% in intrusion detection and application classification, respectively. However, creating separate edges for every packet is computationally expensive, especially in high-throughput networks. For real-time tasks, packet graphs are impractical, leading many graph-based approaches to use flows instead.

3) Text Based Techniques

Inspired by their success in traditional text processing tasks, the efficacy of using NLP techniques for packets has been investigated. These approaches use NLP approaches, such

TABLE 1. Comparison of our work to previous traffic encoding methods.

Studies	Protocol Filter	Input	Preprocessing	Encoding Strategies	Downstream Task	Prediction Type
Image Based						
2D CNN [7]	All	784 bytes from a bidirectional flow	Transformation to grayscale image	2D CNN	Intrusion Detection	Multiclass
HAST-I [14]	All	N bytes from a bidirectional flow	Transformation to grayscale image	2D CNN	Intrusion Detection	Multiclass
HAST-II [14]	All	N bytes from M packets from a bidirectional flow	Transformation to grayscale image	2D CNN & LSTM	Intrusion Detection	Multiclass
SeNet-I [15]	ARP, IPv4	5 packets from an unidirectional flow	Transformation to RGB image	2D CNN	Intrusion Detection	Multiclass
2D CNN [16]	All	Headers of first N packets from an unidirectional flow	Transformation to grayscale image	2D CNN	Intrusion Detection	Multiclass
Text Based						
Payload Embeddings [17]	All	Payload from a single packet	One-hot encoding of bytes	Shallow Neural Network	Intrusion Detection	Binary
PAC-GPT [30]	ICMP & DNS	N packets from a bidirectional flows	Text summary	GPT-3	Packet Generation	Binary
Packet2Vec [27]	All	Entire header and payload of a single packet	Hexadecimal bigram of bytes	Skip-gram	Intrusion Detection	Binary
Pert [18]	Application layer	Payload of a single packet	Tokenisation of bigram byte strings	ALBERT	Application Classification	Multiclass
NorBert [19]	DNS	N packets	Extract domain names	Bert	Device Classification	Multiclass
PL-CNN [20]	All	Payloads of first N packets from unidirectional flows	Extract words from payload	Skip-gram	Intrusion Detection	Binary
PL-RNN [20]	All	Payloads of first N packets from unidirectional flows	Extract N characters from payload words	LSTM	Intrusion Detection	Binary
IoTminer [21]	HTTP, HTTPS, FTP, RTSP	Payload from a single packet	Extract words from payload	GloVe	Device Classification	Multiclass
Graph Based						
TCGNN [22]	Application layer	Payload from a single packet	Transformation to graph	GNN	Intrusion Detection	Multiclass
tGNN [23]	TCP	N packets from an unidirectional flow	Transformation to graph	Temporal GNN	Application Classification	Multiclass
Byte Based						
BLJAN [31]	Application layer	Payload from a single packet	Byte object to integers	BLJAN	Application Classification	Multiclass
1D CNN & FcNN [28]	TCP & UDP	Payload from a single packet	Byte object to normalized integers	1D CNN & FcNN	Intrusion Detection	Binary
PEAN [32]	TLS	N packets from a bidirectional flow	Byte object to hexadecimal integers	Transformer	Application Classification	Multiclass
Our Techniques						
Our 1D CNN	All	Entire header and payload of a single packet	Byte object to integers	1D CNN	Device Classification	Multiclass
Our Compression	All	Entire header and payload of a single packet	Byte object to integers	Deflate & PCA	Device Classification	Multiclass

as Word2Vec [33], GloVe [34] and BERT (Bidirectional Encoder Representations from Transformers) [35] to generate an embedding, with each approach employing a different generation method. For example, Word2Vec [33], the earliest method, utilizes shallow neural networks to learn vector representations of words through context-based training, either using Continuous Bag of Words (CBOW) or Skip-gram models. GloVe [34] extends this concept by aggregating global word-word co-occurrence matrices from a corpus and factorizing these matrices to yield embeddings. This approach integrates both global statistics and local context, effectively bridging the gap between global matrix factorization and local context window methods. BERT [35], the latest method, utilizes a transformer architecture with attention mechanisms to process texts bidirectionally, allowing for a more dynamic representation of word context within a sentence and an understanding of the interdependencies of words.

Some models require text as input, which limits the protocols that can be used. This limitation is detailed in the Protocol Filter column of Table 1. The naive method is to extract words from the packets and turn those words into embeddings. The main difference between methods is what words are extracted and what model is used for encoding the words to embeddings. The authors in [20] extract words from the payloads and then convert each word to a 20-dimensional vector via Skip-gram or Long Short Term Memory networks (LSTMs). Similarly, IoTminer [21] extracts words from application layer packets and uses GloVe to turn them into embeddings. PAC-GPT [30] converts network packets into text representations, which are then fed into GPT-3 to generate Python code for creating the packets.

Another method is to treat the bytes that make up a packet as words. Packet2Vec [27] converts each packet into a series of n-grams. This transformation effectively creates a sequence analogous to words in a text. This approach deliberately omits IP and port details to focus purely on content. Similarly, [17] extends the Skip-gram model found in Word2Vec and focuses on transforming payload sequences through word embeddings.

More recent methods leverage the BERT model. NorBert [19] extracts fully qualified domain names from DNS packets and tokenizes them by splitting the names according to their hierarchy level. These tokens are then inputted into a BERT-based model, which turns them into embeddings. Pert [18] does payload tokenization by taking pairs of byte values as basic character units to generate bigram strings. The authors then train a custom ALBERT model on the strings.

NLP-based approaches create embeddings that do not faithfully represent the packets' actual contents and purposes. NLP is designed to understand and interpret human languages where words have semantic and syntactic relationships. Network packets, conversely, contain structured binary data, where the significance of each bit or byte is contextually different from how words function in a sentence. By converting packets into n-grams and treating them as words, important contextual information is lost. For instance, certain byte sequences in packets, such as headers, checksums, or control flags, are critical for understanding the packet's purpose and integrity. Additionally, waiting for certain packets so that words can be extracted from them limits the effectiveness of the technique.

4) Byte Based Techniques

Byte-based techniques involve minimal packet transformations. The only preprocessing needed, as indicated in the Preprocessing column in Table 1, involves converting byte values into integers. These integer inputs can be used in many different types of models. BLJAN [31] utilizes an embedding module to process both the bytes and their labels into a joint space that captures their implicit correlations using a dual attention mechanism. Similarly, [28] employs a 1D CNN to generate their embeddings from the integer values. PEAN [32] utilizes packets in their hexadecimal representation. The authors train their transformer model by masking a portion of the integers, and the model attempts to regenerate them.

Despite requiring less preprocessing than the others mentioned earlier, these methods still have input limitations and use large and complex models. This becomes apparent when contrasted with our lightweight approach.

5) Our Novelty

In contrast to prior research, our methodology differentiates itself in three areas. The first aspect is that our models use a generalized input without the need to filter specific packet types or protocols. Certain filtering methods, particularly those requiring examination of packet layers (*e.g.*, application layer presence), can incur prohibitively high costs, especially in high-throughput networks. Filtering specific protocols may also hinder the inference of rich context about traffic behaviors. We will demonstrate that although some packets may not be very informative for the device classification task, our lightweight packet encoding methods can extract valuable information from the majority of packets. Secondly, using a single packet as input makes our inference system “stateless”, eliminating the complexity and overhead of maintaining expensive runtime states, such as flow records in alternative approaches. Our models can process each emitted packet independently, making predictions without requiring context. Our final novelty lies in utilizing compression to encode packets for a downstream classifier. To the best of our knowledge, there have been no prior attempts to do this. We showcase that models employing compression-encoded packets exhibit higher accuracy than those using human-engineered features but lower than those utilizing neural network embeddings. However, packet compression encodings are not learned, saving training time.

This paper expands and builds upon our preliminary work in [1]. Both the earlier work and this paper aim to classify IoT network packets. The human feature engineering pipeline created in our previous work serves as one of the baselines for this paper. Additionally, the three classifiers used in the previous study are again used for classification but have been modified to accept the 64-dimension inputs. There are three fundamental differences between these two papers.

- Our previous paper [1] utilized predefined human-driven feature engineering to extract features from IoT traffic data, where domain experts manually select and analyze relevant features. This contrasts with this paper, which employs model-driven, automated feature extraction using neural network-based methods to generate embeddings from raw traffic packets. This approach minimizes the need for manual intervention and enhances the efficiency and scalability of feature extraction by transforming raw data into compact numerical representations. These representations adapt to new data with minimal manual input.
- This paper focuses on metrics emphasizing computational efficiency and cost-effectiveness, introducing lightweight models designed to minimize computational overhead while maintaining high prediction accuracy. It demonstrates how these models can achieve accuracy levels comparable to more complex neural networks but with significantly reduced computing costs, making them well-suited for real-time applications. The study also examines trade-offs associated with using different types of packet data to balance prediction quality and computational costs. In contrast, our previous work [1] only measured the accuracy and reliability of ensemble methods.
- Our previous paper [1] examined the causes and implications of agreements and disagreements in predictions made by ensemble methods. In contrast, our current paper does not consider ensemble methods. Instead, it concentrates on developing efficient models for encoding and classifying IoT network packets. The goal is to achieve high accuracy with minimal computational costs rather than exploring the consistency of predictions across multiple classifiers.

III. Lightweight Transformation of Traffic Packets

In this section, we outline the architectures and preprocessing steps for the models and datasets we use in our experiments. This section concludes with an intrinsic evaluation of the embeddings and an ablation study of our methods. All models are built with Tensorflow 2.15 and Keras 3. More details can be found in our GitHub repository[1].

A. Packet Encoding Models

Our first method closely resembles the approach in [28], which trained a 1D CNN on byte values. However, we introduce three major enhancements. Firstly, we do not restrict inputs to packets with a specific transport layer protocol. Instead, we utilize every packet transmitted by a device, which enhances the flexibility of the embeddings in downstream tasks such as device classification. Secondly, our 1D CNN is more lightweight due to fewer layers and parameters (see Table 2). Finally, the embeddings produced

[1]https://github.com/AleksandarPasquini/EncodePacket

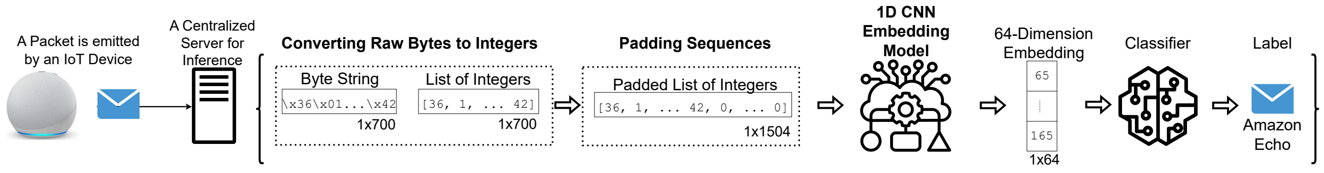


FIGURE 2. **Packets undergo two preprocessing steps before they are ready to be used as input for the 1D CNN. The bold steps are explained in more detail in Section III-A-1.**

by our models will enable multiclass predictions for device classification.

Since model training is a one-time process, it is usually performed off-network before deploying the inference model. GPU-based computational resources can be utilized to expedite training, particularly with large volumes of traffic data. In contrast, real-time inference is far less computationally intensive and can be directly deployed into network switches or on a centralized CPU-based server.

Our second method turns a packet into a compressed object, leveraging compression as an automatic feature extractor. Compression algorithms are designed to eliminate redundant data, resulting in a more concise representation. This reduction in entropy enhances the visibility of the underlying patterns and structures in the data, reducing noise in downstream tasks. The authors of [36] employ the gzip compression algorithm in conjunction with a k-nearest-neighbor classifier for text classification. Their method achieves results comparable to deep learning methods on various in-distribution datasets and even outperforms encoder-only models like BERT on out-of-distribution datasets. Inspired by this concept, we explore the potential of compression for encoding packets. Compression algorithms offer the advantage of not requiring training and can operate efficiently on standard CPUs [25]. This approach is especially useful in environments with limited computational resources or scenarios where real-time processing of large network data is crucial.

Another advantage the compression method has over neural networks is that it does not require supervision. For each new classification task, supervised neural networks like 1D CNNs must be retrained on labels specific to that task; otherwise, the generated embeddings may not align with the classification objective. In contrast, the compression method is task-agnostic, producing the same feature vector regardless of the target task. Evaluating the generalizability of these methods to other classification tasks and analyzing retraining costs is beyond the scope of this paper.

We evaluate our methods against five techniques. Most of these techniques are byte-based, but we also include one technique from each of the image and text domains. To ensure consistency and fairness, we standardize the final vector length to a maximum of 64-dimensions across all techniques, except for the byte values technique and the human-engineered features, which serve as baseline methods. We selected 64-dimensions because the minimum Ethernet frame size is 64 bytes. Optimizing the embedding length is beyond the scope of this paper. In what follows, we will discuss the background of each encoding technique and detail how we incorporated them into our study.

1) Our Lightweight 1D CNN Model

We chose to utilize a 1D CNN because network packets are efficiently processed as sequences of bytes. By applying filters across the packet sequence, 1D CNNs can extract relevant features with minimal preprocessing. In the following sections, we outline the sequential steps involved in transforming raw packet data into an embedding. Fig. 2 visualizes the steps.

Converting Raw Bytes to Integers: The process begins by collecting the packet's byte object and converting each byte object into a feature vector. This transformation uses each byte integer value as a feature value, *e.g.,* a 700 byte long packet will be converted into a list of 700 integers.

The alternative approach involves using N-grams, where N denotes the number of consecutive integers grouped together as a single unit. For example, with N set to 3, the N-gram would consist of three consecutive integers processed as one feature. N-grams can represent more information in shorter sequences, potentially reducing computational complexity compared to individual integers. However, we did not adopt N-grams in our method as preliminary experiments showed they did not decrease the computational cost, mainly due to the larger vocabulary size involved.

Padding Sequences: The final step involves padding the sequences of integers to ensure uniform length, a requirement of the CNN architecture. Sequences are right-padded with zeros if they are shorter than the maximum length of 1504. In the example shown in Fig 2, the 700-length list is padded with zeros to reach a length of 1504. The maximum size of an Ethernet frame is typically 1518 bytes, which includes the Ethernet header (14 bytes) and trailer (4 bytes), leaving a payload (maximum transmission unit) of 1500 bytes. However, we ignore the Ethernet trailer as it does not provide information about the device that sent the packet. Additionally, the source MAC and IP addresses are removed, reducing the packet length by another 10 bytes. Removing source MAC and IP addresses from packet data is crucial to prevent overfitting and ensures the neural network model generalizes effectively across different packets of the same device type and across various networks with differing addressing configurations. Including these unique identifiers may lead the model to memorize specific addresses asso-

TABLE 2. Our lightweight 1D CNN architecture, where N is the batch size.

Input	$[2D]_{N\times 1504}$		
Layer	**Output Shape**	**# of Parameters**	**Hyperparameters**
Embedding	$[3D]_{N\times 1504\times 64}$	16384	$vocab\_size = 256, output\_dim = 64$
Conv1D	$[3D]_{N\times 1490\times 128}$	123008	$filters = 128, kernel\_size = 15$
GlobalMaxPooling1D	$[2D]_{N\times 128}$	0	
Dense	$[2D]_{N\times 64}$	8256	$units = 64$
Classification	$[2D]_{N\times 22}$	1430	$units = 22, activation = softmax$

ciated with device classes instead of learning underlying behavioral patterns. By excluding them, the model can focus on general device features, improving its ability to accurately classify unseen packets of known device classes based on behavior patterns rather than specific identifiers. While most packets require padding, we demonstrate that this does not affect downstream tasks because the models disregard the zeros. Our dataset does not include jumbo packets, which are approximately six times larger (9000 bytes payload), as they are typically managed only by specialized networks, thus validating the choice of 1504 bytes as a suitable length.

1D CNN Embedding Model: Our proposed 1D CNN architecture for generating packet embeddings comprises several key layers that work in tandem to transform the padded vector representations. This section explains the layers employed in the architecture, highlighting their roles and justification for their inclusion in the model. An overview can be seen in Table 2.

Embedding Layer: The Embedding layer serves as the initial component in our CNN architecture, transforming discrete tokens into fixed-sized dense vectors in a 64-dimensional embedding space. Each token in the input sequence, consisting of 1504 tokens, is represented by a unique dense vector, resulting in a 3D output shape of $N \times 1504 \times 64$, where "N" is the batch size. We set N to be 512. The embedding layer contains 16384 trainable parameters whereby each of the 256 possible token values (the vocabulary size of the byte values) is associated with a 64-dimensional vector. Thus, there are $256 \times 64 = 16384$ parameters, which are adjusted during training to enable the model to learn meaningful representations for the byte values.

Conv1D Layer: The Conv1D layer applies 128 one-dimensional convolutional filters to the embedding layer output sequence, which captures local patterns and relationships within the packet data. Each filter has a kernel size of 15, meaning it processes 15 adjacent elements at a time, allowing the model to detect various types of local features. Since the convolution operates without padding the sequence length is reduced as the kernel slides over the input. The output sequence length is determined using the formula: *Output Length = Input Length - Kernel Size + 1*. For this layer, the output length becomes 1490 $(1504 - 15 + 1)$, resulting in an output shape of $N \times 1490 \times 128$. This layer contains 123008 trainable parameters, with each of the 128 filters having 15×64 weights (one for each combination of the 15 input elements and 64 input channels from the embedding layer), plus a bias term per filter. Thus, the total number of parameters is calculated as $(15 \times 64 \times 128) + 128 = 123008$. These parameters are updated during training to help the model extract meaningful patterns from the data.

GlobalMaxPooling1D Layer: The GlobalMaxPooling1D layer applies a max-pooling operation across the entire length of each feature map produced by the Conv1D layer, extracting the most important information by selecting the maximum value from each of the 128 feature maps. This operation reduces the dimensionality of the data while retaining the most salient features. Specifically, it collapses the 1490×128 output from the Conv1D layer into a 128 vector for each sample by taking the maximum value across all 1490 time steps for each feature map. As a result, the output shape becomes $N \times 128$. This layer does not contain any trainable parameters since it performs a fixed mathematical operation without requiring weights to be learned during training.

Dense Layer: The Dense layer is a fully connected layer that transforms the 128-dimensional pooled features into a 64-dimensional space, capturing higher-level abstractions and interactions among the features extracted by the preceding layers. Therefore, the layer's output shape is $N \times 64$. There are 8256 trainable parameters in this layer because each of the 128 input units is connected to all 64 output units, requiring 128×64 weights plus one bias term for each of the 64 output units. Thus, totaling $(128 \times 64) + 64 = 8256$. These parameters enable the model to capture complex interactions in the data. The weights of this layer are extracted and used as embeddings. With 64 output dimensions, the downstream classifier has 64 features to learn from and make inferences. Fig. 2 shows the remaining process.

Classification (Dense Layer): The final Classification layer is a Dense layer that maps the 64-dimensional vectors to probabilities across 22 classes using a softmax activation function, enabling the model to generate probability distributions for class predictions. The output shape of this layer is $N \times 22$, as there are 22 classes. There are 1430 trainable parameters because each of the 64 input units connects to all 22 output units, requiring 64×22 weights, plus one bias term for each of the 22 output units ($(64 \times 22) + 22 = 1430$). Note that this layer is not involved in the generation of embeddings and is only used during training.

2) Our Lightweight Compression Model

Our alternative model utilizes compression, which can accommodate diverse data distributions and characteristics, making it applicable to a wider range of packet protocols without requiring manual adjustments. Compression generally falls into two main categories: lossless, which allows perfect reconstruction of the original data from the compressed file, and lossy, which discards some data permanently to achieve higher compression rates. Our compression strategy incorporated both approaches. The compression

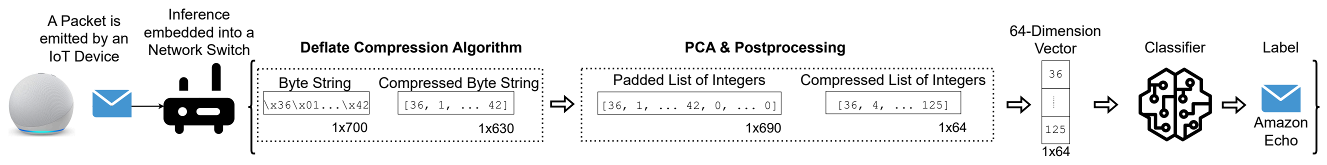


FIGURE 3. **Packets are compressed by both Deflate and PCA algorithms before they are ready to be used by the classifer. The bold steps are explained in more detail in Section III-A-2**

process is illustrated in Fig. 3. The following subsection describes our compression pipeline.

Deflate Compression Algorithm: Since packets are a byte object, the zlib library [37] functions can be directly applied without any preprocessing. We chose to employ the Deflate algorithm [25] for compression. Deflate is a widely used method for efficiently reducing data size during storage or transmission. It operates on data streams using two primary techniques: LZ77, which replaces repeated character sequences with references to earlier occurrences, and Huffman coding, which assigns shorter codes to frequently occurring symbols. Together, these techniques achieve high compression ratios while maintaining fast compression and decompression.

We also introduced several modifications to the standard Deflate compression algorithm. The zlib library adds some overhead to the compressed outputs. Specifically, There is a constant overhead of five bytes per 16 KB block of data, which represents 0.03% additional data. For inputs smaller than 16 KB, this overhead remains fixed at five bytes. Additionally, there is a one-time overhead of six bytes for the first block. In the worst-case scenario, where the input consists of just a single byte, this overhead can inflate the data size by 1100%. These zlib-specific overheads serve to provide metadata, ensuring data integrity, but are unnecessary for our purposes. Therefore, we also set the *wbits* parameter to -15, which removes the header and trailing checksum from the output.

We also set the compression level to the maximum level. These levels affect the balance between compression speed and the degree of compression achieved. Since the packets are all smaller than 1.5 KB, having higher levels of compression will not substantially slow down the process.

Even with the highest level of compression, the packets were only reduced in size by approximately 10%. This limited compression is due to the small size of packets and the lack of compressible information within them. While the deflate algorithm can theoretically achieve compress ratios up to 1:1032 in edge cases (like a file with all zeros), the average packet compression ratio was 1:1.08.

We also evaluated different compression algorithms, such as LZMA, which offers higher compression ratios than Deflate but at the cost of increased computational time. Our comparison revealed no significant difference between LZMA and Deflate in terms of compression ratios. Consequently, we chose Deflate as the compression algorithm for our pipeline. An example packet can be seen in Fig 3. When compressed by the Deflate algorithm, its size is reduced from 700 bytes to 630 bytes, making it ready for the next stage.

PCA & Postprocessing: After compressing the byte objects, they are converted into integer vectors, a necessary format for the classifiers. The size of all vectors is determined by the largest compressed packet. Fig. 3 provides a hypothetical example, demonstrating how the compressed byte string is converted into a list of integers and then padded to a length of 690, which corresponds to the size of the largest packet produced by the Deflate algorithm. Since this exceeds our standardized vector length of 64, we apply PCA to reduce it to 64-dimensions. After this compression, the vector is ready for downstream tasks.

B. Compared Encoding Techniques

The following section will discuss other encoding techniques from previous studies and detail how we incorporated them into our study:

1) Naive Encoding

We employ two baseline encoding methods. The first is the naive approach, which converts the packet's byte integer values into a list of values. To maintain consistency in length across all packets, we standardized the list lengths by determining the longest sequence length among all packets and padding shorter byte sequences with zeros to match the maximum length. While this technique is used for our first intrinsic experiment, the resulting list length causes the packet classifiers to require excessive time for training and inference.

Our second baseline method replaces the byte values approach with human-engineered features. We adopt the IoT-Sense [3], IoTDev [4], and IoTSentinel [5] feature vectors to represent the packets in our experiments. We utilized the feature extraction code provided by IoTDev to calculate the computing costs.

2) Word2Vec

Originally, Word2Vec [33] generates dense vector representations for words based on their contextual usage in a large corpus. This method can be adapted to packet data by treating each packet as a "sentence", and the individual bytes within the packets act as analogous "words", as demonstrated in [27].

The preprocessing follows a similar process to [27] with the byte objects converted into hexadecimal strings. The hex strings are then split into two-character words, and the model is trained on them. The packet vector is generated by averaging the word token vectors of the packet.

For our implementation, we used the Word2Vec implementation in gensim [38], configuring it with the following hyperparameters: a vector size of 64, a window size of 5 (the maximum distance between the current and predicted word within a sentence), and a minimum word count of 1 (considering all words that occur at least once in the dataset to prevent the exclusion of infrequent terms).

3) 2D CNN

As discussed in Section B, the choice of using a 2D CNN to embed individual packets is driven by its ability to effectively capture spatial relationships inherent in the data. By leveraging the two-dimensional structure formed by the bytes and their sequential arrangement, the network can learn hierarchical representations that capture both local patterns and global dependencies.

For preprocessing, the raw packet data undergoes conversion, where each packet is represented as a list of integers using a naive byte value approach. These lists are then padded to achieve uniformity, with a maximum sequence length set to 1504. Subsequently, the padded sequences are reshaped into a 2D array with dimensions (32×47), effectively creating a grayscale image suitable for input into the subsequent 2D CNN model.

We based our 2D CNN architecture on the work in [15]. Appendix Table 8 shows the layers used in the model. The embeddings utilized are extracted from the weights of the penultimate layer.

4) Transformer

Transformers have been applied to tasks such as traffic classification and generation [30], [39]. The transformer architecture excels in processing and contextualizing data, accommodating diverse input sequences, and handling complex correlations, which proves advantageous for converting packets into embeddings. Our transformer model draws inspiration from GPT-2, similar to work in [39]. Our work, however, focuses on individual packets rather than flows.

The preprocessing steps for the transformer involve creating a list of byte integer values and padding them for uniformity. These padded lists are then inputted into our own GPT-2-inspired architecture. This architecture can be seen in Table 9 in the Appendix section. The transformer decoder layers culminate in a pooling layer, from which the output is used as the embedding.

5) LSTM

Long Short-Term Memory (LSTM) networks have been employed for generating embeddings of individual packets due to their inherent ability to capture sequential patterns within the data [40]. The design of our LSTM architecture (Table 10 in the Appendix section) is inspired by an existing model [40]. Similar to the previous approaches, the weights from the penultimate layer are used as embeddings.

TABLE 3. Distribution of network packets across IoT device classes in the UNSW IoT dataset.

IoT Device Class	# Packets
Dropcam	2.1m
Samsung SmartCam	966k
Belkin Wemo Motion Sensor	749k
Amazon Echo	705k
Belkin Wemo Switch	612k
Insteon Camera	500k
Netatmo Welcome	369k
Withings Smart Baby Monitor	350k
Smart Things	290k
Withings Aura Smart Sleep Sensor	239k
TP-Link Day Night Cloud Camera	198k
HP Printer	166k
Netatmo Weather Station	130k
Triby Speaker	111k
LiFX Smart Bulb	88k
Nest Dropcam	76k
PIX-STAR Photo-Frame	42k
iHome	35k
TP-Link Smart Plug	25k
Withings Smart Scale	2985
NEST Protect Smoke Alarm	2317
Blipcare Blood Pressure Meter	131

The input for the LSTM comprises a network packet represented as a list of byte integer values. To ensure uniform processing, the maximum sequence length is determined among all byte lists and all lists are padded to that length.

C. Do Encoding Models Represent IoT Behavior?

Our first experiment aimed to determine the efficacy of the embedding generated by the models. We chose to intrinsically evaluate the quality of the packet embeddings. The objective was to verify that the embeddings accurately represent the underlying packet data, thereby enhancing the effectiveness of subsequent analysis tasks. Specifically, we expected embeddings from packets originating from the same type of device to exhibit similarities, while embeddings from packets of different device types should display distinct differences.

1) Our Packet Traces Data

For our experiments, we utilized the publicly available UNSW IoT dataset [41], a resource widely recognized and validated in various IoT-related research contexts [1], [4], [29]. This dataset contains network traffic captured as PCAP files from both IoT and non-IoT devices. The 22 IoT devices emitted 7.79 million packets (Table 3).

To prepare the UNSW dataset for analysis, we implemented three preprocessing steps. Firstly, we removed all packets not originating from IoT devices, including packets

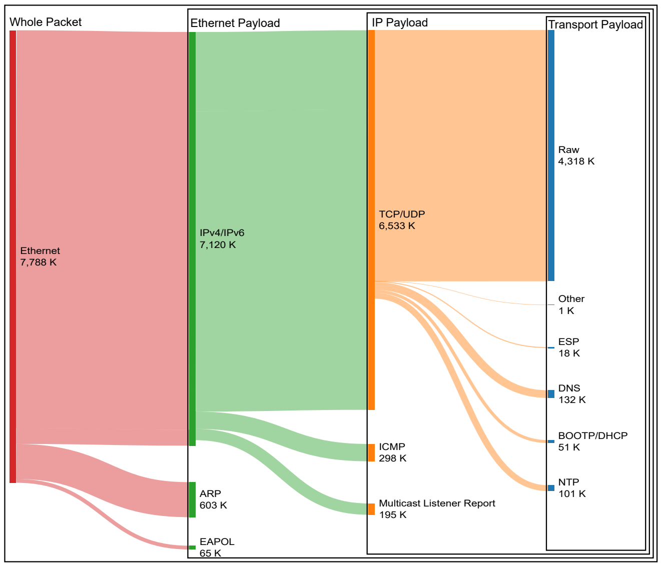


FIGURE 4. The breakdown of packet protocols within our dataset.

received by the devices, as they do not directly reflect device functionality. Secondly, we discarded packets without an Ethernet header as we needed the source MAC address to determine the ground truth. Finally, we stripped source MAC and IP addresses from the packets, where applicable, to prevent the classifiers from overfitting to these specific attributes. This process involved removing the IP addresses from both Address Resolution Protocols (ARP) and Internet Control Message Protocols version 6 (ICMPv6). We retained destination IP and MAC addresses, as they are crucial for understanding the communication endpoints and, consequently, the function of the IoT devices. We also chose not to perform any class balancing, as we wanted the datasets to reflect real-world network traffic conditions. However, in our final experiment in Section V, we also test the accuracy difference between including and excluding the destination IP. After these adjustments, the byte objects are ingested into the models.

We constructed four different datasets of packets, as shown in Fig. 4. The primary difference between these datasets lies in the removal of specific headers. The whole packet dataset (red column on the left of Fig. 4) includes all the information in a packet, whereas the IP payload dataset (the orange column) removes the Ethernet and IP headers from the packets. This approach allows us to assess the impact of selecting particular protocol payloads on the resulting embeddings in our final experiment.

We utilize the Scapy library for packet preprocessing, and the raw protocol (the long blue bar under the Transport Payload column in Fig. 4) is a byproduct of this process. The raw protocol is a default layer for data that does not fit into any of the more structured layers that Scapy can parse. This does not affect our experiments since we do not intend to strip any additional headers beyond the transport layer.

2) Efficacy of Packet Encodings

Before evaluating the methods, we first had to train the models to generate embeddings. A stratified split (70-30) was used to ensure both the training and test data maintained the same imbalanced packet distribution. We decided to train each model until they reached 95% accuracy on the training data, using a batch size of 512 for all the models. Next, we transformed the training data into the "seen" embedding dataset and the testing data into the "unseen" embedding dataset. These datasets were evaluated separately.

The embeddings were evaluated using a variety of metrics. The first metric we examined was cosine similarity. For each class, we computed the cosine similarity of 10,000 embeddings to calculate the average cosine similarity for that class. If a device had fewer than 10,000 packets, all of its instances were used. However, cosine similarity only evaluates how close the embeddings are to each other and does not test if embeddings from different classes are distinct from each other. Therefore, our second evaluation involved creating clusters and measuring different cluster metrics. The completeness, homogeneity and V-measure of the clusters were calculated. A perfect homogeneity score indicates that each cluster contains only members of a single class, while a perfect completeness score means all members of a given class are assigned to the same cluster. The V-measure is the harmonic mean of homogeneity and completeness. The clusters were formed by agglomerative clustering, with the number of clusters being set to the number of devices (22). A stratified dataset split was used, ensuring an equal number of embeddings from both the seen and unseen datasets were selected for clustering. Only 2% of the entire dataset was used to form these clusters. We also separately generated t-distributed Stochastic Neighbor Embedding (t-SNE) plots (Fig. 5) of 5000 embeddings from 7 devices to visualize the clusters in a two-dimensional space. These plots aid in data analysis and help visually communicate the data landscape.

3) Similarity Results

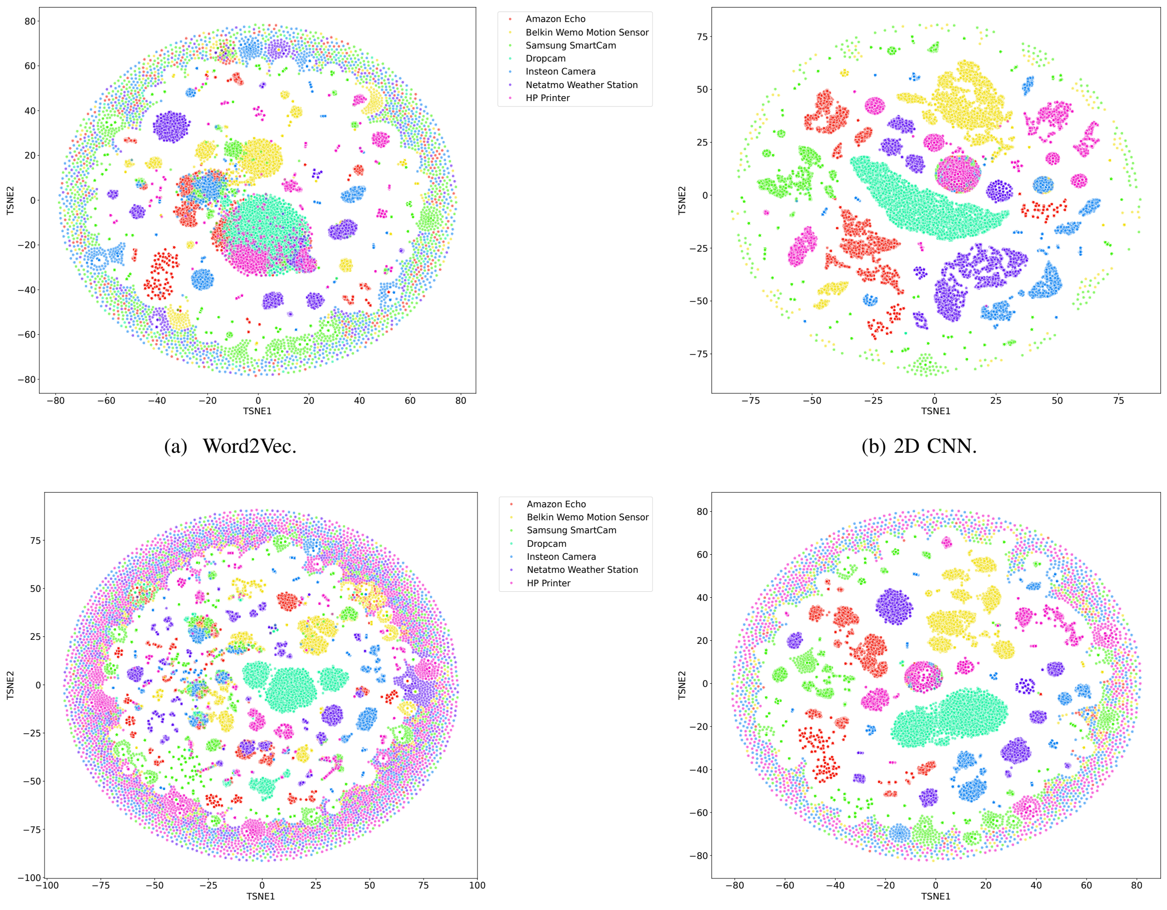
The metrics for each encoding method are presented in Table 4. A visualization of the embedding clusters produced by four representative encoding methods can be seen in Fig. 5. Further visualizations can be seen in our GitHub repository[2].

The byte values and compression methods produced the second-worst and worst cosine similarity scores, respectively. This is because byte values of objects do not inherently represent vectors in a meaningful metric space. These are low-level representations and lack the contextual or structural information needed to determine the similarity of the content or functionality of these objects. Also, cosine similarity assumes a linear space where dimensions are independent and equally significant, which is not the case with network packets. Different fields in a packet play different roles. For instance, destination IP addresses and port numbers have more significant implications for similarity than other

[2]https://github.com/AleksandarPasquini/EncodePacket

TABLE 4. Summary of encoding cluster evaluation metrics.

Metric	Bytes Values	Compression		1D CNN		Word2Vec		2D CNN		LSTM		Transformer	
	Baseline	Seen	Unseen	Seen	Unseen	Seen	Unseen	Seen	Unseen	Seen	Unseen	Seen	Unseen
Cosine Similarity	0.52±0.10	0.33±0.16	0.33±0.16	0.71±0.11	0.71±0.12	0.64±0.13	0.64±0.19	0.75±0.11	0.75±0.12	**0.81±0.11**	**0.81±0.12**	0.75±0.13	0.74±0.14
Completeness	0.47	0.40	0.40	0.80	0.80	0.47	0.47	0.70	0.71	**0.85**	**0.85**	0.82	0.81
Homogeneity	0.50	0.43	0.43	0.89	0.89	0.48	0.48	0.81	0.83	**0.92**	**0.92**	0.90	0.90
V-measure	0.49	0.41	0.42	0.84	0.84	0.47	0.48	0.75	0.76	**0.88**	**0.88**	0.86	0.86



(a) Word2Vec. (b) 2D CNN.

(c) Compression. (d) 1D CNN.

FIGURE 5. t-distributed Stochastic Neighbour Embedding (t-SNE) of the encoding methods. The perplexity was set to 30, and 1000 iterations were performed.

fields. This poor performance also extends to the cluster analysis, with both techniques exhibiting poor completeness and homogeneity scores. These results indicate that these methods are likely to underperform in downstream tasks.

Word2Vec also exhibits poor performance because the assumptions of natural language processing do not hold for network packets. For example, the distributional hypothesis, which suggests that words appearing in similar contexts have similar meanings, cannot be applied to packets. This is reflected in the poor cosine similarity and V-measure scores.

The remaining models perform well, with none achieving a cosine similarity lower than 0.70 or a V-measure score lower than 0.75. Additionally, there is a maximum of a 0.01 difference in the V-measure between the seen and unseen embeddings, indicating that the models can generalize well to unseen data. The t-SNE projections show the 2D CNN embeddings creating the best clusters (Fig. 5b). Most of the clusters are homogeneous except for the one in the center. However, LSTM achieves the highest metrics, suggesting that the LSTM embeddings will yield the highest accuracy in the downstream classification task.

4) 1D CNN Ablation

We also conducted an ablation study to assess the impact and importance of individual components or variables in the 1D CNN. By selectively disabling or modifying these components one at a time, we could determine their individual contributions to the model's performance. Each ablation was

TABLE 5. The impact of ablation scenarios on the cosine similarity of seen and unseen instances, as well as on training time.

Ablation Category	Ablation Scenario	Seen Similarity	Unseen Similarity	Average Training Time per Batch
Baseline: kernel size=15, embeddings dimension=64 # filters=128, activation function=None, batch size=512		**0.74**	**0.74**	**18ms**
Kernel Size	↓ decreasing to 3	0.67	0.67	17ms
	↓ decreasing to 5	0.70	0.70	17 ms
	↓ decreasing to 7	0.71	0.70	17 ms
	↓ decreasing to 9	0.71	0.71	17 ms
	↓ decreasing to 12	0.73	0.72	18 ms
	↑ increasing to 18	0.75	0.74	19 ms
	↑ increasing to 21	0.73	0.73	19 ms
Embeddings Dimension	↓ decreasing to 32	0.73	0.73	17 ms
	↑ increasing to 128	0.73	0.72	20 ms
Number of Filters	↓ decreasing to 96	0.74	0.73	17 ms
	↑ increasing to 256	0.76	0.75	21 ms
Activation Function	+ adding Relu to convolutional layer	0.73	0.73	18 ms
	+ adding Sigmoid to convolutional layer	0.74	0.74	18 ms
	+ adding Swish to convolutional layer	0.72	0.71	19 ms
1D Convolution Assumption	- removing convolutional layer	0.52	0.51	14 ms

trained with the same amount of resources. The results can be seen in Table 5.

The first component we modified was the kernel size. The number of parameters grows linearly with kernel size, so smaller kernel sizes are generally preferred. With a kernel size of 5, the number of parameters is 67,158, which is half of the number for a kernel size of 15. However, this reduction comes at a cost: a 5% decrease in cosine similarity. It also reduces the training batch time by 1 millisecond, saving approximately 10 seconds per epoch. We believe that this time saving was not worth the decrease in performance. Even with a kernel size of 9, which maintained a batch training time of 17ms, the performance loss was too significant to justify. At a kernel size of 12, there was a decrease in performance without any time savings. Therefore, decreasing the kernel size does not improve the overall model.

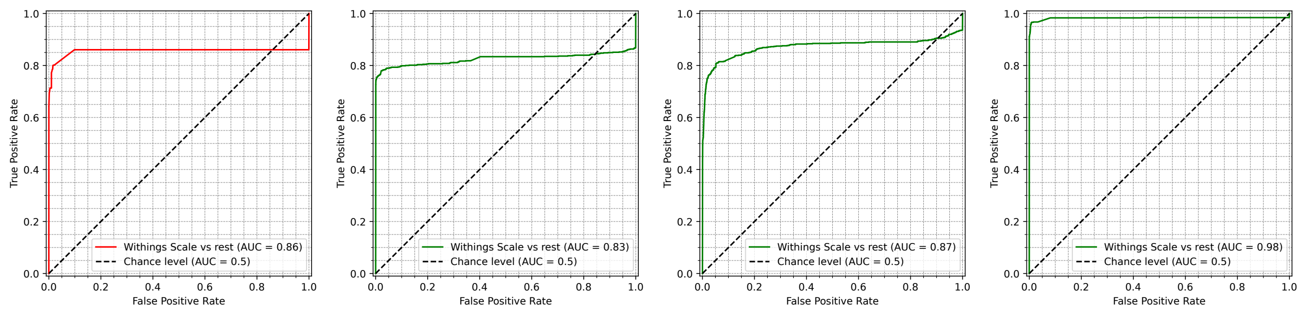
We also increased the kernel size to 18 and 21 to see if there was an increase in the cosine similarity. Our results show, at best, a 0.01 increase but generally no change, and the training time also increases by 1 ms per step. This suggests that a kernel size of 15 is appropriate for our 1D CNN.

Changing the number of embedding dimensions, a hyperparameter for the first layer, impacted the training time. Decreasing the number of dimensions to 32 reduced the time to 17 ms, while increasing the dimensions to 128 raised the time to 20 ms. However, both changes resulted in a 0.01 decrease in cosine similarity. Neither of these ablations improved the model.

The filters in the convolutional layer are important for extracting features from the data. More filters should extract more features, which can lead to more similar embeddings. This is reflected in an increase in cosine similarity to 0.76, but this comes at the cost of an additional 3 ms in training time. We decided not to increase the number of filters, as the 0.02 performance increase was not worth the extra time. Similarly, decreasing the filters reduced the performance by 0.01 and decreased the time by 1 ms. Again, the small changes did not improve the model.

Next, we evaluated three activation functions—Rectified Linear Units (ReLU), Sigmoid, and Swish—within the convolutional layer. ReLU ($\max(0, x)$), Sigmoid ($1/(1 + e^{-x})$) and Swish ($x/(1+e^{-x})$) should empower the neural network to identify a broader spectrum of non-linear patterns within the data, thus providing the resulting embeddings with more information. After testing the model with each activation function, the results did not change or decrease. Additionally, the training time increased by 2 ms when adding the Swish function. This could be because convolutional layers already perform a kind of element-wise non-linear operation through the convolution process. This inherent non-linearity identification might have been sufficient, and adding an extra activation function did not alter the model's ability to capture more complex patterns.

Finally, we removed the 1D convolution layer to test our assumption regarding its necessity for generating packet embeddings. If the performance remains relatively consistent or only experiences a minor degradation, it would suggest that the 1D convolution layer might indeed be dispensable for this task, indicating that a simpler architecture suffices. However, the observed 0.22 decrease in results indicates that the convolution layer provides important information to the embeddings, thereby validating our assumption.



(a) Human-engineered features. (b) Word2Vec embeddings. (c) Compression. (d) 1D-CNN embeddings.

FIGURE 6. The ROC curves for a representative class, Withings Smart Scale, when using (a) human-engineered features, (b) Word2Vec embeddings, (c) compression, and (d) 1D-CNN embeddings as input for the IoTSense classifier [3].

IV. Efficacy of Packet Embeddings in Predicting IoT Classes

We now evaluate the practical utility of our embeddings in real-world network analysis tasks by using them as features in IoT device classifiers. To do this, we adapt three established IoT device classification techniques [3]–[5], modifying them to classify single packets instead of using packet aggregation. Details of this adaptation process are available in our previous work [1]. Despite adapting these three techniques to operate with embeddings and compressed data, we maintained the same hyperparameters across the experiments to ensure consistent conditions.

We excluded the byte-value method due to its impractical size and instead used the human-engineered features as the baseline for comparison. The embeddings are again split into 70% (the seen dataset from the previous experiment) and 30% (the unseen dataset). Table 6 reports the average accuracy, precision, recall and AUC of these experiments.

Additionally, we quantify the memory cost, the training and inference times and computing costs. To track these metrics during training and generation, we used CodeCarbon 2.3.4 [42][3]. All evaluations were performed on a 64-bit Ubuntu 5.4.0 server equipped with 20 AMD EPYC 7742 64-core processors, 1008 GB RAM, and 4 NVIDIA A100s GPUs. It is worth noting that the Word2Vec, compression and human feature engineering methods did not make use of the GPUs. CodeCarbon measured the energy usage of the GPUs and RAM, while providing an estimated CPU usage based on the processors' thermal design power. With the collected information, we perform a cost-benefit analysis of the techniques.

A. Performance

Table 6 provides an overview of the performance of embeddings generated by each method across the three classifiers. We use macro-average to ensure that each class contributes equally to the final evaluation metric. Calculating metrics for each class independently and then averaging them prevents majority classes, like Dropcam, from disproportionately influencing the results. The results indicate that human feature engineering ranks as the second worst-performing method. This highlights the ability of neural networks to produce useful embeddings that outperform those created by human experts.

TABLE 6. Macro-average performance across the three IoT classifiers.

Method	Accuracy	Precision	Recall	AUC
Compression	0.91	0.88	0.79	0.97
1D CNN	**0.94**	**0.95**	**0.88**	**0.98**
Human Features	0.88	0.87	0.78	0.96
Word2Vec	0.87	0.86	0.70	0.96
2D CNN	**0.94**	**0.95**	0.87	**0.98**
LSTM	**0.94**	0.94	0.87	**0.98**
Transformer	**0.94**	**0.95**	**0.88**	**0.98**

Fig. 6 illustrates the Receiver Operating Characteristic (ROC) curves for a representative class, Withings Smart Scale, across different methods. At a false positive rate of 0, the human-engineered features exhibit the second lowest true positive rate, while the compression method performs the worst. This again shows that the neural network models generate more informative features than those developed by humans, especially at the threshold of 0 false positives.

Surprisingly, compression does not have the worst AUC or accuracy; that distinction goes to Word2Vec. The poor clustering observed in the first experiment has resulted in subpar performance in this experiment for Word2Vec. However, the compression method, which initially exhibited worse clustering performance than Word2Vec, now outperforms it across all metrics. This suggests that features defining clusters may not align with those defining class boundaries, leading to ineffective clustering but effective classification.

The other embedding methods showed similar performance, all resulting in trends resembling the one shown in Fig. 6d. This highlights their ability to extract information from the raw data. Although the LSTM method was expected to achieve the highest metrics according to the first

[3]https://github.com/mlco2/codecarbon/tree/v2.4.1

TABLE 7. Costs of encoding methods when using the whole packet as input (normalized for 1000 input packets).

Method	Training Time (Until 95% accuracy)	Training Cost (Until 95% accuracy)	Embedding Generation Time (Per 10000 inputs)	Inference Time Embedding Generation + Classification (Per 10000 inputs)	Inference Cost (Per 10000 inputs)	Model Memory Cost
Compression	**0**	**0**	4.32s	4.38s	0.6 Wh	**0**
1D CNN	445s	121.9 Wh	**2.00**s	**2.07**s	**0.4** Wh	1.71 MB
Human Features	**0**	**0**	53.68s	53.76s	7.3 Wh	**0**
Word2Vec	138s	18.8 Wh	87.73s	87.83s	12.0 Wh	0.09 MB
2D CNN	1145s	261.3 Wh	2.37s	2.43s	0.5 Wh	36.94 MB
LSTM	6013s	1563.0 Wh	30.64s	30.70s	6.6 Wh	4.87 MB
Transformer	49436s	12430.0 Wh	4.39s	4.46s	1.2 Wh	2.31 MB

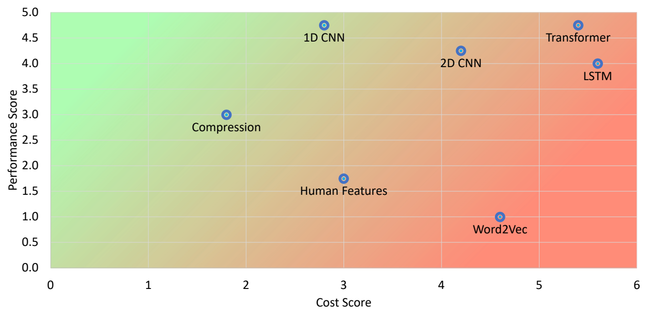


FIGURE 7. Cost versus benefit of various techniques.

experiment, all neural networks, except Word2Vec, produced similar results. This suggests a performance ceiling, where approximately 6% of the packets are difficult to classify correctly. These 6% of packets are likely low-entropy packets, such as those from the network time protocol (NTP). Once the source IP and MAC addresses are removed from an NTP packet, very little distinguishes an NTP packet sent by one device from another.

B. Costs

All of the neural networks, except Word2Vec, achieved over 90% accuracy, but the computational cost of achieving that accuracy varied significantly. These differences can be seen in Table 7. The inherent complexity of the neural networks led to more demanding computational requirements during both the training and inference stages. In contrast, compression and human feature engineering techniques are more resource-efficient due to their reduced algorithmic complexity. For example, the LSTM model is 7.5 times slower in generation time than the compression method. Additionally, the compression and human-engineered feature methods do not require model training.

Regarding energy costs, generally, the longer the model takes to train and generate embeddings, the more power it will consume. However, using the A100 GPUs can reduce execution time at the cost of higher power consumption. This is evident with the compression method taking approximately the same time to generate embeddings as the transformer, but the transformer uses more power. The slower time for compression is due to selecting the highest level of compression for the packets.

The memory costs for each technique depend on the number of parameters required by the architecture. Interestingly, execution times are not correlated with the model sizes; rather, the types of transformations being performed have a more significant impact. For example, the 2D CNN model has 13 times more trainable parameters than the transformer model, yet the transformer is 43 times slower during training. However, higher memory costs do lead to increased energy consumption. The 1D and 2D CNN models both have similar embedding generation times, but the larger 2D CNN model consumes approximately 25% more energy.

C. Cost vs Benefit Analysis

To determine the most cost-effective model, we ranked the models based on their performance (Table 6) and costs (Table 7). Each column of Table 7 was used to rank the models except for the embedding generation time, which is already accounted for in the inference time. Models with the highest performance or lowest cost received a score of 1, while those with the lowest performance or highest costs received a score of 7, given there are seven methods.

For example, the 1D CNN model received performance scores of [4,5,6,4] (based on four performance metrics in Table 6) and cost scores of [4,4,1,1,4] (based on five cost metrics in Table 7). These scores were then averaged and plotted in Fig. 7. The 1D CNN model is represented at the score coordinates (cost, performance) = (2.8, 4.75). Note that regions closer to the top left of Fig. 7, highlighted in green, are more desirable than regions closer to the bottom right, highlighted in red.

From Fig. 7, two techniques stand out: Compression and the 1D CNN. These techniques offer the best performance relative to their cost. However, our metrics do not capture the entire picture. Traditional human feature extraction is often a time-consuming process, and previous studies did not report the time taken to identify those features. In contrast, both the neural networks and compression pipelines streamline this process by autonomously discerning information from the raw data. Therefore, these methods are preferable to human feature engineering.

A network engineer should choose compression or 1D CNN based on the available resources. If resources are limited, compression is the recommended choice. If resources are ample, 1D CNN is the better option.

To fully understand the suitability of our models, a larger-scale experiment involving hundreds of devices should be conducted. It is possible that more devices require bigger

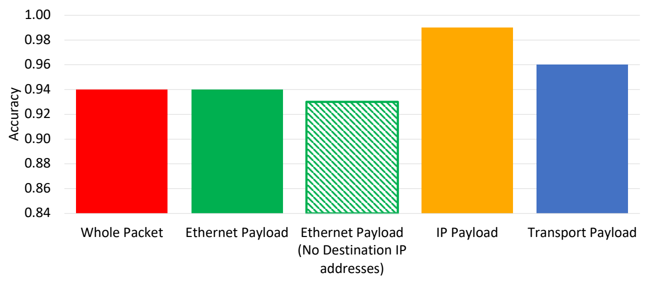


FIGURE 8. The average accuracy of the three IoT classifiers with different input datasets. Each input dataset went through the 1D CNN pipeline. Error bars are omitted due to a minimal standard deviation of 0.001

models to distinguish the types, and compression alone may not suffice. Another approach could be to classify devices based on their function rather than their make/model. This would reduce the number of classes needed and potentially make the models more robust to out-of-distribution devices.

V. Balancing Packet Data Against Prediction Quality

We used the whole packet as the input throughout our previous experiments, allowing the models to process any packet. However, there are low information packets, and using only parts of the packet might achieve similar results without the extra processing cost. The optimal input should have a smaller size, achieve high accuracy (or another output metric), and be widely available in the network. A smaller size is desirable as it reduces both the training and inference costs. Widespread availability [6] in the network also means that the data can be extracted without long delays. Additionally, achieving 100% accuracy on 20% of all network packets is less advantageous than achieving 75% accuracy on 100% of the packets.

We examine four different types of inputs (Whole Packet, Ethernet Payload, IP Payload, Transport Payload) and evaluate their size, availability and accuracy. We recognize that the sub-datasets exclude packets found in the parent datasets; for example, EAPOL packets are not included in the IP Payload dataset. As shown in Fig. 4, our dataset is organized into four subsets with specific relationships. We opted to include all valid packets in the parent datasets to streamline processing and avoid runtime filtering. In this experiment, we also created a version of the Ethernet Payload sub-dataset with the destination IP address removed from packet headers.

These five datasets are processed through the 1D CNN pipeline and subsequently used to train the three multiclass classifiers. Fig. 8 shows the average accuracy of the classifiers trained on each sub-dataset. Additionally, we illustrate in Fig. 9 how frequently five representative devices supply input packet data in a 1-hour window. In what follows, we discuss our experimental results.

1) Whole Packet

Using the whole packet as input for an IoT device classifier provides a comprehensive view as it contains all the possible information except for the source addresses. This approach leads to relatively accurate classification for 94% of packets (as indicated by the red bar in Fig. 8), but it also has the maximum input size of 1504 (the maximum transmission unit in our dataset). If a packet size is less than 1504 bytes, then it needs to be padded, introducing redundant information. This issue is exacerbated by protocols like the Address Resolution Protocol (ARP), which lack distinguishing device features, especially when source IP and MAC addresses are removed from packets. Such packets become unclassifiable, adding unnecessary computational overhead. That said, using the entire packet avoids assumptions about specific packet protocols and does not rely on dynamic conditional filters, enabling it to accept any packet as input. This approach can expedite the classification process for less frequently active devices, such as the Pixtar photo frame and Blipcare blood pressure meter (shown in the two bottom rows in Fig. 9).

2) Ethernet Payload

The Ethernet payload provides classifiers with nearly identical information as the whole packet, excluding the Ethernet header. The solid green bar in Fig. 8 reveals that the Ethernet payload achieves equivalent accuracy to the whole packet, suggesting this specific header contributes little significant information. We note that removing the Ethernet header results in only a minor 8 byte reduction in total size. Importantly, the availability of the Ethernet payload is the same as the whole packet within our dataset because all packets include an Ethernet header. This uniformity may differ in other network contexts. Choosing the Ethernet payload for network administrators offers a preferable balance, delivering comparable accuracy and availability at a slightly lower computational cost.

3) IP Payload

Focusing solely on the IP payload emphasizes the actual data exchanged between devices and their corresponding endpoints, potentially retaining the distinctive patterns while eliminating the irrelevant ones. This sub-dataset excludes approximately 10% of all packets, mainly ARP packets, which are likely to be incorrectly classified. Therefore, the availability of this scenario is 90% of the Whole Packet scenario. This sub-dataset achieves 99% accuracy, shown by the yellow bar in Fig. 8. Removal of both the IP and Ethernet headers reduces the total packet size by only about 2%. Since all inputs must be padded to the maximum size, removing these headers only gives a reduction of 24 bytes (The 4-byte IP source address and 6-byte MAC source address are already removed).

4) Transport Payload

The transport payload sub-dataset contains 40% fewer packets than the original dataset of whole packets. A significant portion of those packets are involved in TCP handshakes and do not carry payload data, leaving us with approximately 4.6 million usable packets. In Fig. 9, we observe that packets with transport payloads (blue dots) are less frequent compared to other packets, particularly noticeable

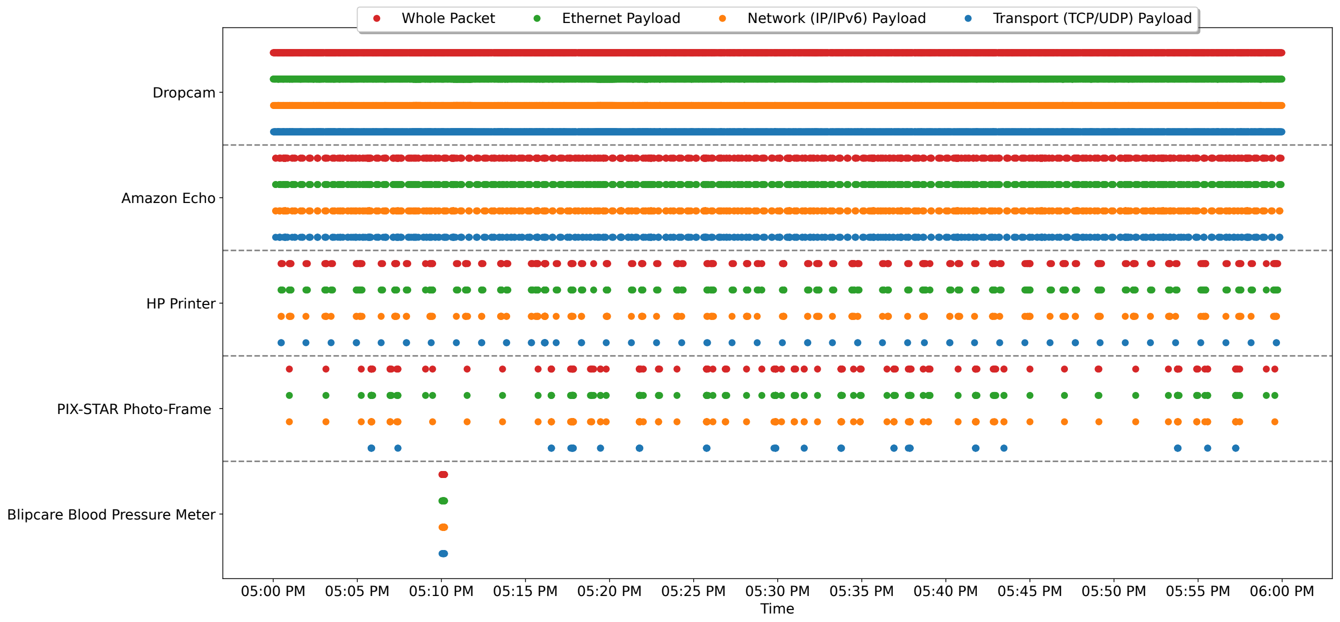


FIGURE 9. The frequency and availability of various input packets emitted by five representative devices in our dataset within a one-hour window.

in the traffic from the HP printer and Pixtar photo frame. Limiting the inputs to the packet payloads has been done in many studies [17], [18], [20]. The data patterns within the payloads are expected to strongly indicate the type of device generating the traffic. However, this approach achieved only 96% accuracy (shown by the blue bar in Fig. 8), which is lower than that achieved using the IP payloads. This suggests that the transport headers do contribute valuable information for device classification. Furthermore, the reduction in size is minimal, approximately 8 bytes (UDP header)[4], similar to the IP payload scenario. Thus, relying on transport payload is less favorable due to lower availability and accuracy compared to IP packets, despite a marginal decrease in the input size.

Discussion of Input Analysis

While primarily qualitative, our analysis above can be extended into a formal optimization framework. Constructing a realistic objective function requires capturing three aspects: computational costs, performance metrics, and input availability with normalization factors. Together, these elements can help network administrators make informed decisions. For administrators with sufficient resources, accuracy should be prioritized over cost. Conversely, if resources are limited, cost should be given more weight. Another area for further exploration is examining the advantages and disadvantages of only the IP header or the first N bytes of a packet payload, where N is a configurable parameter.

Alternative methods for quantifying the costs and availability of the inputs could be developed. Using size as a proxy for computational costs may not be ideal, as not all packets contain useful information. Also, one may prefer the mean arrival time of each input to be a better measure of packet availability. We note that packet availability varies across different networks. Some devices might rarely use certain protocols, while others use them frequently. Understanding this ratio would enable a more accurate model of how quickly all devices can be classified correctly.

VI. Conclusion

This paper developed two novel approaches for classifying network behaviors from IoT devices using packet embeddings: one based on a 1D CNN architecture and the other on compression techniques. By applying these methods to real traffic traces, we demonstrated that automatically generated embeddings can achieve up to 6% higher classification accuracy than features identified by expert humans. We performed a cost-benefit analysis comparing the benefits of neural network models to simpler algorithms. While neural networks achieved a 94% performance metric, this came with increased computational costs. Our compression and 1D CNN models provided the best balance between performance and cost. Finally, we examined the types of packet inputs and their impact on availability, input size, and accuracy. We found that using the IP payload of packets yields the highest accuracy, but approximately 10% of packets do not contain an IP layer. Therefore, network administrators should use the whole packet as input until an IP packet is available.

[4]When the maximum transmission unit (MTU) is used, the payload of UDP packets is larger than that of TCP because UDP headers are only 8 bytes compared to at least 20 bytes for TCP. Therefore, all inputs in the Transport Payload scenario will be padded to the maximum size, which corresponds to the largest UDP payload possible.

Our compression embedder model uses fewer computing resources than the 1D CNN embedder model, making it a more efficient choice for embedding directly into network switches. However, due to its lower predictive accuracy, network operators may choose to configure switches to dynamically forward packets from specific MAC addresses to a central server running the 1D CNN-based model for more accurate classification. Although the 1D CNN model demands greater computational power, it offers higher classification accuracy, making it well-suited for deployment on an appropriately configured central server.

Acknowledgment

This research is supported by the Commonwealth of Australia as represented by the Defence Science and Technology Group of the Department of Defence.

REFERENCES

[1] A. Pasquini *et al.*, “Exploring the Reliability of IoT Packet Classifiers: An Experimental Study,” in *Proc. IEEE Globecom Workshops*, Kuala Lumpur, Malaysia, Dec 2023.
[2] I. Lee, “Internet of Things (IoT) Cybersecurity: Literature Review and IoT Cyber Risk Management,” *Future internet*, vol. 12, no. 9, p. 157, Sep 2020.
[3] B. Bezawada *et al.*, “Behavioral Fingerprinting of IoT Devices,” in *Proc. ACM ASHES*, Toronto, Canada, Oct 2018.
[4] K. Kostas *et al.*, “IoTDevID: A Behavior-Based Device Identification Method for the IoT,” *IEEE IoT Journal*, vol. 9, no. 23, pp. 23 741–23 749, Jul 2022.
[5] M. Miettinen *et al.*, “IoT SENTINEL: Automated Device-Type Identification for Security Enforcement in IoT,” in *Proc. IEEE ICDS*, Atlanta, USA, Jul 2017.
[6] A. Pashamokhtari *et al.*, “Efficient IoT Traffic Inference: From Multi-View Classification to Progressive Monitoring,” *ACM Trans. Internet Things*, vol. 5, no. 1, pp. 1–30, Dec 2023.
[7] M. Hamidouche *et al.*, “Enhancing IoT Security via Automatic Network Traffic Analysis: The Transition from Machine Learning to Deep Learning,” *arXiv preprint arXiv:2312.00034*, Nov 2023.
[8] A. Pashamokhtari, N. Okui, Y. Miyake, M. Nakahara, and H. Habibi Gharakheili, “Combining Stochastic and Deterministic Modeling of IPFIX Records to Infer Connected IoT Devices in Residential ISP Networks,” *IEEE Internet of Things Journal*, vol. 10, no. 6, pp. 5128–5145, Nov 2022.
[9] S. J. Saidi, A. M. Mandalari, R. Kolcun, H. Haddadi, D. J. Dubois, D. Choffnes, G. Smaragdakis, and A. Feldmann, “A Haystack Full of Needles: Scalable Detection of IoT Devices in the Wild,” in *Proc ACM IMC*, Oct 2020.
[10] A. Hamza, D. Ranathunga, H. Habibi Gharakheili, T. A. Benson, M. Roughan, and V. Sivaraman, “Verifying and Monitoring IoTs Network Behavior Using MUD Profiles,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 1, pp. 1–18, May 2020.
[11] Y. Xian, T. Lorenz, B. Schiele, and Z. Akata, “Feature Generating Networks for Zero-Shot Learning,” in *Proc. IEEE CVPR*, Salt Lake City, USA, Jun 2018.
[12] X. Xu *et al.*, “Semantic Embedding Space for Zero-shot Action Recognition,” in *Proc. IEEE ICIP*, Quebec, Canada, Sep 2015.
[13] C. Cheng *et al.*, “Meta-Adapter: An Online Few-shot Learner for Vision-Language Model,” *Proc. NeurIPS*, Dec 2023.
[14] W. Wang *et al.*, “HAST-IDS: Learning Hierarchical Spatial-temporal Features using Deep Neural Networks to Improve Intrusion Detection,” *IEEE access*, vol. 6, pp. 1792–1806, Dec 2017.
[15] Y. A. Farrukh *et al.*, “Senet-i: An Approach for Detecting Network Intrusions Through Serialized Network Traffic Images,” *Engineering Applications of Artificial Intelligence*, vol. 126, no. 4, p. 107169, Sep 2023.
[16] E. Paolini *et al.*, “Real-Time Network Packet Classification Exploiting Computer Vision Architectures,” *IEEE Open Journal of the Communications Society*, vol. 5, pp. 1155–1166, Feb 2024.
[17] M. Hassan *et al.*, “Intrusion Detection Using Payload Embeddings,” *IEEE Access*, vol. 10, pp. 4015–4030, Dec 2021.
[18] H. Y. He *et al.*, “PERT: Payload Encoding Representation from Transformer for Encrypted Traffic Classification,” in *Proc. IEEE ITU Kaleidoscope*. Ha Noi, Vietnam: IEEE, Dec 2020.
[19] F. Le *et al.*, “NorBERT: NetwOrk Representations Through BERT for Network Analysis & Management,” in *Proc. MASCOTS*, Nice, France, Oct 2022.
[20] H. Liu *et al.*, “CNN and RNN Based Payload Classification Methods for Attack Detection,” *Knowledge-Based Systems*, vol. 163, pp. 332–341, Jan 2019.
[21] Y. Zhang *et al.*, “IoTminer: Semantic Information Extraction in the Packet Payloads,” in *Proc. IEEE GLOBECOM*, Rio de Janeiro, Brazil, Jan 2022.
[22] G. Hu *et al.*, “TCGNN: Packet-grained Network Traffic Classification via Graph Neural Networks,” *Engineering Applications of Artificial Intelligence*, vol. 123, no. 3, p. 106531, Aug 2023.
[23] L. Gioacchini *et al.*, “Exploring Temporal GNN Embeddings for Darknet Traffic Analysis,” in *Proc. ACM GNNet*, New York, USA, Dec 2023. [Online]. Available: https://doi.org/10.1145/3630049.3630175
[24] C. S. Wallace and D. L. Dowe, “Minimum Message Length and Kolmogorov Complexity,” *The Computer Journal*, vol. 42, no. 4, pp. 270–283, Jan 1999.
[25] P. Deutsch, “Rfc1951: Deflate Compressed Data Format Specification Version 1.3,” USA, 1996.
[26] H. Abdi and L. J. Williams, “Principal Component Analysis,” *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, Jul 2010.
[27] E. L. Goodman *et al.*, “Packet2Vec: Utilizing Word2Vec for Feature Extraction in Packet Data,” *arXiv preprint arXiv:2004.14477*, Apr 2020.
[28] D. A. Bierbrauer *et al.*, “Transfer Learning for Raw Network Traffic Detection,” *Expert Systems with Applications*, vol. 211, p. 118641, Jan 2023.
[29] A. Sivanathan *et al.*, “Detecting Behavioral Change of IoT devices using Clustering-Based Network Traffic Modeling,” *IEEE IoT Journal*, vol. 7, no. 8, pp. 7295–7309, Mar 2020.
[30] D. K. Kholgh and P. Kostakos, “PAC-GPT: A Novel Approach to Generating Synthetic Network Traffic with GPT-3,” *IEEE Access*, vol. 11, pp. 114 936–114 951, Oct 2023.
[31] K. Mao *et al.*, “Byte-Label Joint Attention Learning for Packet-grained Network Traffic Classification,” in *Proc. IEEE/ACM IWQOS*, Tokyo, Japan, Jun 2021.
[32] P. Lin *et al.*, “A Novel Multimodal Deep Learning Framework for Encrypted Traffic Classification,” *IEEE/ACM Transactions on Networking*, vol. 31, no. 3, pp. 1369–1384, Jun 2023.
[33] T. Mikolov *et al.*, “Efficient Estimation of Word Representations in Vector Space,” *arXiv preprint arXiv:1301.3781*, Sep 2013.
[34] J. Pennington *et al.*, “Glove: Global Vectors for Word Representation,” in *Proc. ACL EMNLP*, Doha, Qatar, Oct 2014.
[35] J. Devlin *et al.*, “Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *arXiv preprint arXiv:1810.04805*, May 2018.
[36] Z. Jiang *et al.*, ““Low-Resource” Text Classification: A Parameter-Free Classification Method with Compressors,” in *Proc. Association for Computational Linguistics*, Toronto, Canada, Jul 2023, pp. 6810–6828.
[37] J.-l. Gailly and M. Adler, “Zlib Compression Library,” 2004.
[38] R. Řehůřek and P. Sojka, “Software Framework for Topic Modelling with Large Corpora,” in *Proc. LREC Workshop on New Challenges for NLP Frameworks*, Valletta, Malta, May 2010.
[39] R. Bikmukhamedov and A. Nadeev, “Generative Transformer Framework for Network Traffic Generation and Classification,” *T-Comm*, vol. 14, no. 11, pp. 64–71, Oct 2020.
[40] R.-H. Hwang *et al.*, “An LSTM-Based Deep Learning Approach for Classifying Malicious Traffic at the Packet Level,” *Applied Sciences*, vol. 9, no. 16, Aug 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/16/3414
[41] A. Sivanathan *et al.*, “Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics,” *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, pp. 1745–1759, Aug 2019.
[42] B. Courty *et al.*, “mlco2/codecarbon: v2.4.1,” May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11171501

Appendix

TABLE 8. 2D CNN architecture, where N is the batch size.

Input	$[4D]_{N\times32\times47\times1}$		
Layer	**Output Shape**	**# of Parameters**	**Hyperparameters**
BatchNormalization	$[4D]_{N\times32\times47\times1}$	4	
Relu Activation	$[4D]_{N\times32\times47\times1}$	0	
Conv2D	$[4D]_{N\times32\times47\times16}$	160	$filters = 16, kernel\_size = (3,3)$ $padding = same$
BatchNormalization	$[4D]_{N\times32\times47\times16}$	64	
Relu Activation	$[4D]_{N\times32\times47\times16}$	0	
Conv2D	$[4D]_{N\times32\times47\times16}$	2320	$filters = 16, kernel\_size = (3,3)$ $padding = same$
BatchNormalization	$[4D]_{N\times32\times47\times16}$	64	
Relu Activation	$[4D]_{N\times32\times47\times16}$	0	
Conv2D	$[4D]_{N\times32\times47\times32}$	4640	$filters = 32, kernel\_size = (3,3)$ $padding = same$
BatchNormalization	$[4D]_{N\times32\times47\times32}$	128	
Relu Activation	$[4D]_{N\times32\times47\times32}$	0	
Conv2D	$[4D]_{N\times32\times47\times32}$	9248	$filters = 32, kernel\_size = (3,3)$ $padding = same$
BatchNormalization	$[4D]_{N\times32\times47\times32}$	128	
Relu Activation	$[4D]_{N\times32\times47\times32}$	0	
Conv2D	$[4D]_{N\times32\times47\times64}$	18496	$filters = 64, kernel\_size = (3,3)$ $padding = same$
BatchNormalization	$[4D]_{N\times32\times47\times64}$	256	
Relu Activation	$[4D]_{N\times32\times47\times64}$	0	
Conv2D	$[4D]_{N\times16\times24\times64}$	36928	$filters = 64, kernel\_size = (3,3)$ $padding = same, strides = (2,2)$
Flatten	$[2D]_{N\times24576}$	0	
Dense	$[2D]_{N\times128}$	3145856	$units = 128, activation = relu$
Dropout	$[2D]_{N\times128}$	0	$rate = 0.2$
Dense	$[2D]_{N\times64}$	8256	$units = 64, activation = relu$
Dropout	$[2D]_{N\times64}$	0	$rate = 0.2$
Classification	$[2D]_{N\times22}$	1430	$units = 22, activation = softmax$

Note: The padding type is called "SAME" because it ensures that the output size remains the same as the input size.

TABLE 9. Transformer Architecture, where N is the batch size

Input	$[2D]_{N\times1504}$		
Layer	**Output Shape**	**# of Parameters**	**Hyperparameters**
TokenAndPositionEmbedding	$[3D]_{N\times1504\times64}$	112640	$vocab\_size = 256$ $sequence\_length = 1504$ $embedding\_dim = 64$
TransformerDecoder	$[3D]_{N\times1504\times64}$	33472	$intermediate\_dim = 128$ $num\_heads = 2, dropout = 0.2$ $normalize\_first = True$
TransformerDecoder	$[3D]_{N\times1504\times64}$	29344	$intermediate\_dim = 96$ $num\_heads = 2, dropout = 0.2$ $normalize\_first = True$
TransformerDecoder	$[3D]_{N\times1504\times64}$	25216	$intermediate\_dim = 64$ $num\_heads = 2, dropout = 0.2$ $normalize\_first = True$
LayerNormalization	$[3D]_{N\times1504\times64}$	128	
GlobalMaxPooling1D	$[2D]_{N\times64}$	0	
Classification	$[2D]_{N\times22}$	1430	$units = 22, activation = softmax$

TABLE 10. LSTM architecture, where N is the batch size

Input	$[2D]_{N\times1504}$		
Layer	**Output Shape**	**# of Parameters**	**Hyperparameters**
Embedding	$[3D]_{N\times1504\times64}$	16384	$vocab\_size = 256, output\_dim = 64$
Bidirectional LSTM	$[3D]_{N\times1504\times256}$	197632	$units = 128, return\_sequences = True$
Dropout	$[3D]_{N\times1504\times256}$	0	$rate = 0.2$
Bidirectional LSTM	$[3D]_{N\times1504\times128}$	164352	$units = 64, return\_sequences = True$
Dropout	$[3D]_{N\times1504\times128}$	0	$rate = 0.2$
Bidirectional LSTM	$[2D]_{N\times64}$	41216	$units = 32, return\_sequences = False$
Dropout	$[2D]_{N\times64}$	0	$rate = 0.2$
Dense	$[2D]_{N\times64}$	4160	$units = 64$
Classification	$[2D]_{N\times22}$	1430	$units = 22, activation = softmax$

Aleksandar Pasquini received his Bachelor of Science degree in 2019 and his Master of Information Technology degree in 2022, both from the University of Melbourne. He is currently pursuing a Ph.D. in Computer Science at Deakin University. His research interests include network behavior characterization, machine learning in networks, and the Internet of Things (IoT).

Rajesh Vasa received his BAppSc. (1997) and PhD in Software Engineering (2010) from Swinburne University of Technology. He is currently the Head of Translational Research & Development at Deakin Applied AI Institute, Deakin University; and holds a Chair in Software and Technology Innovation. His research interests include evolution of complex systems, robustness of AI, construction agentic systems and quantification of failure in AI models.

Irini Logothetis (MIEAust CPEng NER APEC Engineer IntPE(Aus)) received her BCCompSc(Hons) BEng(Elect)(Hons) in 2005 and her Ph.D. in Engineering in 2021. She is a Senior Research Fellow at the Applied Artificial Intelligence Institute (A2I2) at Deakin University. Her research is in knowledge reasoning and representation in sustainability for social value.

Hassan Habibi Gharakheili received his B.Sc. and M.Sc. degrees in Electrical Engineering from the Sharif University of Technology in Tehran, Iran, in 2001 and 2004, respectively, and his Ph.D. in Electrical Engineering and Telecommunications from the University of New South Wales (UNSW) in Sydney, Australia in 2015. He is currently a Senior Lecturer at UNSW Sydney. His research interests include programmable networks, learning-based networked systems, and data analytics in computer systems.

Alexander Chambers received his PhD in High-Performance Computational Physics from the University of Adelaide in 2018. He is currently a cybersecurity researcher at Defence Science Technology Group, Australia. His research interests include data science applications to cybersecurity, particularly network analysis and security operations.

Minh Tran received his PhD in Computer Systems Engineering from the University of South Australia in 2011. He is currently a cybersecurity researcher at the Defence Science and Technology Group, Australia. His research interests are in the areas of cyber terrain mapping and user/entity behavioural analytics.