ACM DIGITAL LIBRARY
acm open
Check for updates
Latest updates: https://dl.acm.org/doi/10.1145/3718958.3750529

PDF Download
3718958.3750529.pdf
20 February 2026
Total Citations: 1
Total Downloads: 2836

Published: 08 September 2025

Citation in BibTeX format

SIGCOMM '25: ACM SIGCOMM 2025 Conference
September 8 - 11, 2025
Coimbra, Portugal

Conference Sponsors:
SIGCOMM

RESEARCH-ARTICLE

Pegasus: A Universal Framework for Scalable Deep Learning Inference on the Dataplane

YINCHAO ZHANG, Tsinghua University, Beijing, China
SU YAO, Tsinghua University, Beijing, China
YONG FENG, Tsinghua University, Beijing, China
KANG CHEN, Tsinghua University, Beijing, China
TONG LI, Renmin University of China, Beijing, China
ZHUOTAO LIU, Tsinghua University, Beijing, China
View all

Open Access Support provided by:
Tsinghua University
Renmin University of China
Beihang University
Beijing Institute of Technology

SIGCOMM '25: Proceedings of the ACM SIGCOMM 2025 Conference (September 2025)
https://doi.org/10.1145/3718958.3750529
ISBN: 9798400715242

Pegasus: A Universal Framework for Scalable Deep Learning Inference on the Dataplane

Yinchao Zhang[★] Su Yao[★] Yong Feng[★] Kang Chen[★†] Tong Li[♠] Zhuotao Liu[★‡]
Yi Zhao[◇] Lexuan Zhang[★] Xiangyu Gao[★] Feng Xiong[§] Qi Li[★] Ke Xu[★‡]
[★]Tsinghua University [‡]Zhongguancun Laboratory [♠]Renmin University of China [◇]Beijing Institute of Technology
[†]Beijing National Research Center for Information Science and Technology [§]Beihang University

Abstract

The paradigm of Intelligent DataPlane (IDP) embeds deep learning (DL) models on the network dataplane to enable intelligent traffic analysis at line-speed. However, the current use of the match-action table (MAT) abstraction on the dataplane is misaligned with DL inference, leading to several key limitations, including accuracy degradation, limited scale, and lack of generality. This paper proposes Pegasus to address these limitations. Pegasus translates DL operations into three dataplane-oriented primitives to achieve generality: Partition, Map, and SumReduce. Specifically, Partition "divides" high-dimensional features into multiple low-dimensional vectors, making them more suitable for the dataplane; Map "conquers" computations on the low-dimensional vectors in parallel with the technique of Fuzzy Matching, while SumReduce "combines" the computation results. Additionally, Pegasus employs Primitive Fusion to merge computations, improving scalability. Finally, Pegasus adopts full-precision weights with fixed-point activations to improve accuracy. Our implementation on a P4 switch demonstrates that Pegasus can effectively support various types of DL models, including Multi-Layer Perceptron (MLP), Recurrent Neural Network (RNN), Convolutional Neural Network (CNN), and AutoEncoder models on the dataplane. Meanwhile, Pegasus outperforms state-of-the-art approaches with an average accuracy improvement of up to 22.8%, along with up to 248× larger model size and 212× larger input scale.

CCS Concepts

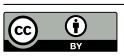
• **Networks → Programmable networks**; • **Computing methodologies → Learning paradigms**.

Keywords

Programmable Dataplane, Deep Learning, Intelligent Dataplane

ACM Reference Format:

Yinchao Zhang, Su Yao, Yong Feng, Kang Chen, Tong Li, Zhuotao Liu, Yi Zhao, Lexuan Zhang, Xiangyu Gao, Feng Xiong, Qi Li, Ke Xu. 2025. Pegasus: A Universal Framework for Scalable Deep Learning Inference on the Dataplane. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3718958.3750529



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGCOMM '25, Coimbra, Portugal
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1524-2/25/09
https://doi.org/10.1145/3718958.3750529

1 Introduction

In recent years, there has been a growing demand for Intelligent DataPlane (IDP), which leverages data-driven learning models to overcome the limitations of traditional rule-based approaches [21, 25, 39, 54, 64]. By utilizing high-performance programmable hardware [4, 53], IDP supports forwarding-native execution of learning models, enabling intelligent traffic analysis at line-speed, without affecting network throughput and latency. This empowers fine-grained traffic classification and malicious flow detection, essential for coping with increasingly complex network environments.

The core challenge in realizing IDP lies in the fact that switch dataplanes are primarily optimized for high-speed packet processing using the match-action table (MAT) abstraction [5], which inherently limits their ability to represent learning models. While many recent works [25, 60, 64] have explored tree-based models due to the similarity between their decision processes and the MAT abstraction, there remain scenarios that demand more expressive and versatile models. Consequently, the community has also explored to incorporating neural network (NN)-based models in IDP. However, the MAT abstraction on the dataplane lacks the flexibility to support complex computations such as multiplication and exponentiation, which are essential for DL.

To address this challenge, two main approaches have been proposed: computation simplification [33, 37, 61] and computation bypassing [49]. Computation simplification simplifies operations, for example, by binarizing the entire model. For instance, N3IC [38] replaces multiplication with binary XNOR and population count (popcnt) operations, directly implementing binary Multi-Layer Perceptron (MLP) within the MAT on SmartNICs. Computation bypassing avoids computation by storing input-output relationships on the dataplane, recording an enumerative mapping from input bit strings to output bit strings, as demonstrated in BoS [51].

However, both methodologies suffer from three key limitations:
Accuracy. Accuracy refers to how well a model accomplishes its task, measured by metrics such as precision, recall, or F1-score, depending on the specific traffic analysis tasks. Computation simplification, such as model binarization in N3IC [38], degrades accuracy due to the reduced numerical range. For example, N3IC may lead to accuracy degradation in VPN traffic classification tasks [51]. In contrast, computation bypassing through mapping can improve accuracy compared to computation simplification. However, the mapping has limitations in scale (see below). This limitation forces reductions in input precision or dimensionality, leading to a loss of critical information necessary for accurate predictions [22, 42, 65].
Scalability. Scalability represents the ability to perform DL inference at a larger scale, and this larger scale applies to both the input scale and the model size. Computation simplification encounters

Design	❶	❷	❸	❹	❺
Accuracy			✓		
Scalability		✓		✓	✓
Generality	✓				

Table 1: The goals of different designs in Pegasus.

Prior Works	Accuracy	Model size	Input scale
N3IC [38] (binary MLP)	22.8% ↑	248x ↑	29x ↑
BoS [51] (binary RNN)	17.9% ↑	237x ↑	212x ↑
Leo [25] (Decision Tree)	17.2% ↑	-	-

Table 2: Pegasus v.s. Prior Works.

difficulties in both the input scale and the model size. N3IC [38] cannot scale its throughput on the dataplane, such as in Barefoot Tofino architecture switches [8], due to additional limitations on binary operations in high-speed environments (e.g., shorter processing cycles allowing only one binary operation on one variable per MAT stage). Computation bypassing methods like BoS [51] suffer from limited input scale (e.g., a 21-bit input requires 2^{21} table entries, exceeding the capacity of the Barefoot Tofino 2 programmable switch [7]), resulting in poor scalability.

Generality. Generality refers to whether the system can support different DL operations, and thereby utilize these operations to perform inference for various models. The computation simplification strategy of N3IC is limited to Matrix Multiplication (MatMul) and fails to generalize to other DL layers, such as Batch Normalization (BN) and activation functions. BoS [51] also faces the generality problem. It processes a small number of inputs per time step for binary Recurrent Neural Network (RNN), making it unsuitable for other model types. This limitation conflicts with the current trend in networking to design specialized models for different tasks and to leverage larger input scales to capture more complex relationships [2, 14, 28, 45, 48]. The generalization issue restricts flexibility and limits the broader potential of IDP.

This paper proposes Pegasus to address the three limitations above with five tightly-coupled designs. ❶ For a wide range of model types, Pegasus translates operations (e.g., MatMul, BN and ReLU) within DL layers into three primitives: Partition, Map, and SumReduce. Specifically, ❷ Pegasus uses Partition to divide one-time operations on the entire input into multiple fine-grained computations on minimal input units, uses Map to retrieve precomputed results from mapping tables for each unit and uses SumReduce for aggregation to handle multi-input scenarios, which reduces table sizes. This method reduces the number of inputs that each table has to process. ❸ Therefore, Pegasus is able to adopt full-precision model weights (precomputed with full-precision parameters) while using fixed-point numbers for activation representations instead of binary numbers (since an 8-bit number query requires only 2^8 table entries). This leverages a wider numerical range, allowing the capture of more detailed and precise information crucial for accurate model predictions. Pegasus also adopts two additional methods to further optimize the primitive implementation. ❹ Divide the input into minimal units increases the number of lookups, placing considerable additional pressure on memory access bandwidth. To mitigate this overhead, we introduce Fuzzy Matching, which groups multiple units together and maps it to a corresponding table entry, enabling a single lookup to cover multiple units, effectively reducing memory access bandwidth consumption. ❺ Additionally, Pegasus adopts Primitive Fusion to merge multiple operations, thereby reducing the total number of tables. Table 1 shows how these designs contribute to addressing the three limitations of previous approaches.

Concretely, Pegasus introduces the following key innovations:

(i) A primitive-based framework that abstracts DL operations into dataplane-executable forms, enabling diverse models to run directly on commodity programmable switches without hardware modifications (See §4.1).

(ii) A Fuzzy Matching paradigm for table lookups on the dataplane, systematized and adapted in this work to enable compact and approximate indexing over feature vectors using match rules compatible with P4. This mechanism avoids exhaustive input-output mappings and significantly reduces storage and lookup overhead. In addition, by encoding inputs into compact Fuzzy Indexes, it increases fusion capacity by allowing more inputs to be processed within a single lookup, and enhances flow scalability by reducing per-flow state storage requirements (See §4.2).

(iii) A Primitive Fusion mechanism tailored to the dataplane, which—unlike prior fusion techniques on AI accelerator platforms that primarily reduce memory traffic—also reduces the number of executed operations, making it well-suited for resource-constrained environments (See §4.3).

Contributions. The major contribution of this paper is the design, implementation and evaluation of Pegasus[1], the first IDP design that supports multiple DL models on commodity programmable switches. Our implementation on the P4 switch demonstrates that Pegasus can effectively support various model types on the dataplane, including MLP, RNN, Convolutional Neural Network (CNN), and AutoEncoder. Experiments show that Pegasus can support 3840 bit input scale, 6083 Kb model size, and achieves an average classification accuracy of 97.3%. This represents up to 248× larger model size, 212× larger input scale, and up to 22.8% higher accuracy compared to state-of-the-art approaches (Table 2 gives a preview of Pegasus's benefits, see Table 5 for the full results).

2 Background and Motivation

Deep Learning. Deep Learning (DL) utilizes neural networks composed of multiple layers to model complex patterns in data [26]. The implementation of DL involves a variety of operations that process data through different types of layers, such as fully connected (FC), convolutional (Conv), activation (Act), normalization (Norm), pooling (Pool), recurrent (Rec), and embedding (Emb) layers.

In DL, each layer performs specific mathematical operations. For instance, FC layers compute weighted sums of inputs plus biases, enabling the network to capture linear relationships. Conv layers apply convolution operations to detect local patterns like edges in images. Rec layers handle sequential data by maintaining a hidden state that captures temporal dependencies. Activation functions like ReLU, Softmax, and tanh introduce nonlinearity, allowing the network to learn complex, non-linear relationships. Norm layers adjust the input distributions to subsequent layers, enhancing model stability and performance. Pooling layers reduce the spatial dimensions of data, decreasing computational load and controlling

[1]Available at https://github.com/afireswallow/Pegasus.

overfitting by summarizing features. Embedding layers transform discrete data into continuous vector spaces, which is particularly useful for capturing temporal features in time series data. All these operations involve intensive computations, such as Matrix Multiplications (MatMul), convolutions, and non-linear transformations.

Programmable Dataplane. The emerging programmable switches [5, 36, 53] offer flexible dataplane programmability, allowing developers to execute custom processing logic on each data packet. Many programmable switches today can be programmed using the P4 [4] language, a domain-specific language based on the match-action table (MAT) [5] abstraction. The MAT abstraction extracts fields from packet headers and matches them against flow tables, where matched entries specify the actions to be executed on the packets. While the MAT abstraction provides significant flexibility for designing network functions, its practical implementations often face critical limitations.

For instance, the Protocol-Independent Switch Architecture (PISA)—one of the most widely adopted implementations—supports only basic integer operations such as bitwise operations (e.g., NOR, XNOR), shifts, addition, and subtraction. It does not support floating-point numbers, multiplication, division, nor exponential operations—operations that are essential for DL inference computations. Secondly, the resources available for MAT on the dataplane are limited. For instance, on Barefoot Tofino 2, each pipeline only has 20 MAT stages, with each stage equipped with 10 Mb of SRAM, 0.5 Mb of TCAM, and a 1024-bit-wide Action Data Bus [7]. Given that DL involves numerous operations across multiple layers, the 20 MAT stages and 1024-bit bus make it difficult to meet the computational and data transfer demands.

Why DL on the dataplane? The increasing demand for real-time, intelligent network traffic analysis has created a need to deploy learning models directly on the dataplane switch [31, 35], enabling tasks like Intrusion Prevention systems (IPS) to analyze and block malicious traffic with terabit throughput and nanosecond-level latency. Traditional approaches [25, 60, 64] often rely on tree-based models, which are valued for their simplicity and interpretability. DL complement these methods by offering significant advantages in addressing certain unique challenges of networking: (1) The transmission of network data inherently exhibits temporal characteristics, and DL models, such as RNNs and 1D CNNs, are well-suited to capture temporal patterns, making them better fit network-specific tasks. (2) DL can extract features directly from raw packets, overcoming the difficulties of complex feature computation in the constrained dataplane environment [51]. (3) The networking field often lacks labeled data [63] and needs to address continually emerging new attacks [30], such as zero-day attacks [18, 52], making the unsupervised learning capabilities of DL an invaluable tool for adapting to these dynamic and unpredictable scenarios.

Motivation. DL inference typically involves highly compute intensive operations, which conflict with the flow table-centric dataplane. This requires developers to design DL inference implementations that better align with the dataplane MAT abstraction.

N3IC uses XNOR and population count (popcnt, counting the number of 1s in the binary representation) to replace the multiplication and addition operations in MatMul. This enables the implementation of a simple binary Multi-Layer Perceptron (MLP) on the computation-constrained dataplane. However, binarizing

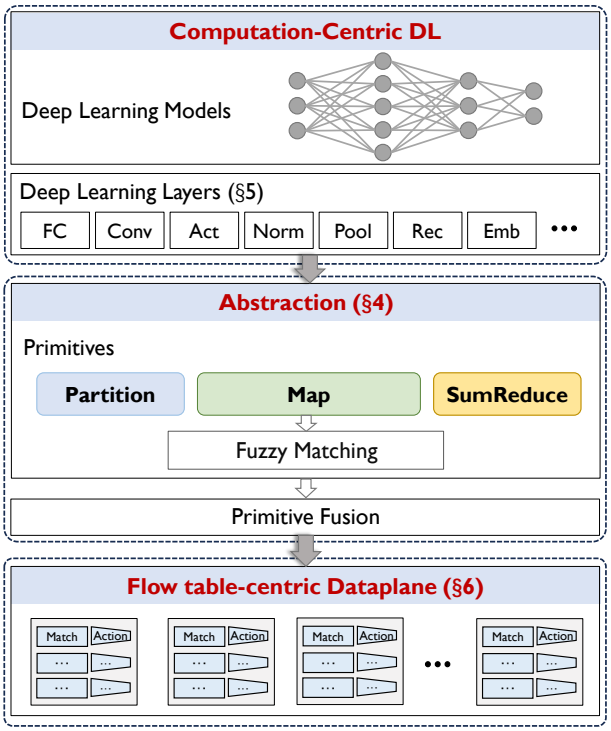


Figure 1: Pegasus Architecture.

the entire model reduces precision, leading to accuracy degradation. Moreover, this method does not support other DL operations, such as activation functions, limiting its generality. Finally, this approach has poor scalability, making it hard to fit within switch pipelines. For example, a 128-bit to 64-bit MatMul requires 64 XNOR and popcnt operations, with each popcnt taking up 14 switch stages [51].

In contrast, state-of-the-art BoS [51] bypasses all DL operations by looking up the mapping from input bit strings and output bit strings. This allows binary activations only at the input and output layers, while enabling full-precision computation within the model. This improves accuracy to some extent compared to N3IC. However, this method limits input scalability, requiring 2^n entries for an n-bit input, resulting in poor overall scalability. This limitation brings two additional issues. First, input binarization is required to increase dimensionality and boost accuracy. This binarization, however, reduces the numerical range of input data, leading to accuracy degradation, as evidenced by our experiments in §7.2. Second, the restricted input scale limits the method's generality, making it primarily suitable for models like Recurrent Neural Networks (RNN), where small inputs are processed at each time step.

We noticed that a recent work, Taurus [39], explores the design of a novel ASIC by incorporating additional hardware resources to enable DL inference. In this paper, We focus exclusively on implementing DL inference on commodity programmable switches.

3 Design Overview

3.1 Design Goals

We propose Pegasus to achieve higher accuracy, greater scalability, and generality in supporting various DL models. (1) Pegasus

Primitives	Expression
Partition	$\text{Partition}(X) = \{X_1, X_2, \ldots, X_k\}$
Map	$\text{Map}(\mathcal{F}, \{X_1, X_2, \ldots, X_k\}) = \{F_1(X_1), F_2(X_2), \ldots, F_k(X_k)\}$
SumReduce	$\text{SumReduce}(\{X_1, X_2, \ldots, X_k\}) = \sum_{i=1}^{k} X_i$

Table 3: Primitives in Pegasus. X is the input vector, X_i represents the i-th segment of X, and $\mathcal{F}$ is a set of functions including F_i.

introduces three primitives, including Partition, Map, and SumReduce, to decompose DL models into a sequence of primitives, achieving generality. (2) Pegasus uses Partition to divide input into segments, uses Map with Fuzzy Matching to retrieve precomputed results for each segment, and applies SumReduce to aggregate results through summation. Additionally, Pegasus employs Primitive Fusion to merge multiple primitives, reducing the number of operations. These methods enable Pegasus to efficiently handle larger model scales, achieving scalability. (3) Finally, Pegasus employs full-precision weights and fix-point activations to enhance model accuracy.

3.2 Pegasus Architecture

Figure 1 shows the hierarchical design of Pegasus. DL layers are composed of various DL operators, which are further converted into primitives for computation. The design of these primitives is dataplane-oriented and can be integrated tightly with the MAT abstraction. The lowest-level implementation of the primitives needs to satisfy the limitations of the specific programmable switch.

We analyzed common operations in DL (detailed in §5) and found that many functionalities can be realized through parallel data operations, necessitating the design of the Map primitives. DL often requires simple sum aggregation, and implementing this process on the dataplane is not complex. The SumReduce primitives are proposed. Finally, to enable data flow between primitives, the Partition primitive is needed. DL operators are then represented using these primitives. For example, consider a MatMul operation. We can use Partition to divide the input, apply Map to compute the product of each segment with the target matrix, and obtain the final result through SumReduce.

4 Pegasus Primitives

4.1 Primitives

Pegasus primitives fall into three categories: Partition, Map, and SumReduce, as illustrated in Table 3. The three primitives can be combined in varying quantities and orders to assemble various DL operators, enabling the construction of distinct DL models. Specifically, as the dataplane is better suited for handling multiple parallel small-scale computations rather than a single large-scale operation, the Partition primitives divide the multi-dimensional input vector into sub-vectors, reducing computational complexity. Map primitives execute specific functions (e.g., activation function and batch normalization) on each segment of inputs. Fuzzy matching (§4.2) efficiently supports multiple Map primitives with minimal storage resources and table lookups. SumReduce primitives perform element-wise summation on multiple vectors, resulting in an

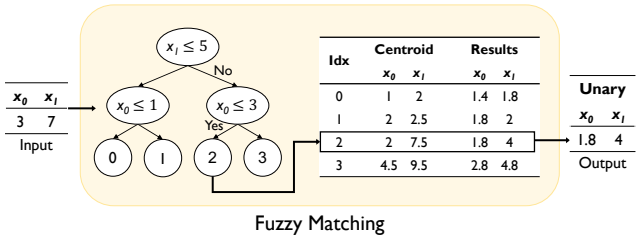


Figure 2: Implementation of a Map primitive: how input subvector $(3, 7)$ retrieves results $(1.8, 4)$ as the approximation of $f(X_i) = 0.4X_i + 1$.

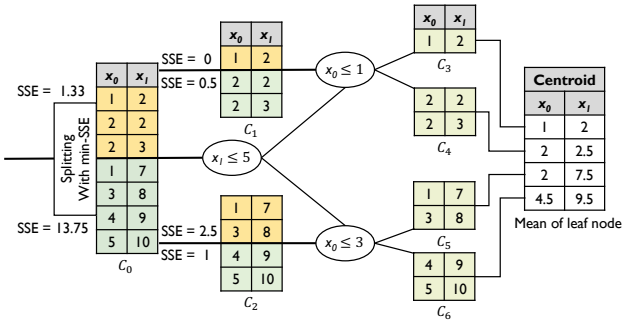


Figure 3: An example of obtaining cluster parameters and centroids from the training dataset in Pegasus.

aggregated vector. These primitives are simple enough to be implemented using the MAT abstraction. More importantly, Pegasus employs primitive fusion (§4.3) to reduce resource overhead and improving the scalability.

4.2 Fuzzy Matching

Instead of retrieving results through exhaustive and non-scalable input-output mapping table lookups, Fuzzy Matching groups multiple input units into a vector and executes a feature-threshold-based search on the vector, following ideas from prior work on table-based quantization [3].

Fuzzy Indexing. Specifically, a clustering tree is constructed where each node contains a specific feature (one dimension in the vector) and its corresponding threshold. The input vector is mapped to the index of a leaf node through simple comparison operations, where each leaf node corresponds to a precomputed centroid (i.e., cluster center) representing the approximate value for data in that region. Compared to traditional distance-based clustering methods, this approach can be easily implemented on the constrained dataplane. Figure 2 shows an example for the input ($x_0 = 3, x_1 = 7$). Based on the conditions $x_1 > 5$ and $x_0 \le 3$, the input is mapped to centroid index (Fuzzy Index) 2. This index corresponds to the precomputed centroid $(2, 7.5)$. After applying Map $f(X_i) = 0.4X_i + 1$, the approximate results are $(1.8, 4)$. This approach leverages the continuity of DL operators (e.g., MatMul and BN), where the operator $f(x)$ remains relatively stable within a small range of input x, allowing minor variations in the input without significantly affecting the output [67].

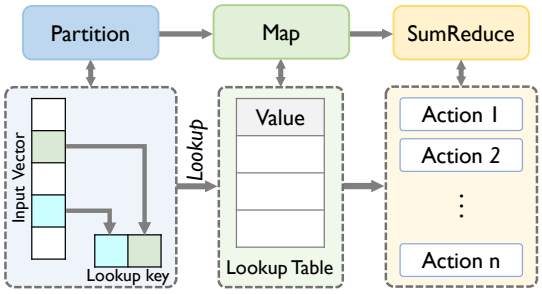


Figure 4: Correspondence between the MAT abstraction and primitives.

Parameter Learning. Based on the independent and identically distributed (i.i.d.) assumption of DL [43, 44], we can learn the parameters (including features and thresholds at the non-leaf nodes, and centroids at the leaf nodes) from the training set for inference. We adopt a greedy clustering strategy, starting with all training data as a single cluster C_0 at the root. At each step, we split the current cluster into two sub-clusters by selecting the optimal feature dimension and threshold that minimize the total Sum of Squared Errors (SSE), thereby enhancing intra-cluster similarity. For example, as shown in Figure 3, cluster C_0 is split along feature x_1 at threshold 5, forming two sub-clusters assigned to the left and right child nodes. This process continues recursively until the tree reaches the target size. Although the greedy strategy does not guarantee a global optimum, it provides a near-optimal split, suitable for efficient dataplane implementation. The centroid of each cluster is computed as the mean vector of its feature dimensions. For instance, the centroid of cluster C_6 (4.5, 9.5) is the average of (4, 9) and (5, 10).

Benefits of Fuzzy Matching. Compared to storing precomputed input-output mappings for each input unit on the dataplane, Fuzzy Matching offers four key advantages:

- Storage Efficiency: traditional methods suffer from exponential storage growth as the number or bit-width of operands increases. For example, a binary operation (e.g., Hadamard product) or two 8-bit inputs requires 2^{16} table entries. Fuzzy matching avoids storing all possible input-output pairs, drastically reducing storage overhead and enhancing scalability, albeit with a slight accuracy drop of approximately 1% (see §7.5).
- Lookup Reduction: Fuzzy Matching enables a single table lookup to cover multiple input units, substantially reducing the number of table lookups and improving memory access bandwidth utilization.
- Primitive Fusion: Fuzzy Matching significantly enhances the capability of Advanced Primitive Fusion (see §4.3).
- Flow Scalability: Fuzzy Matching supports concurrent flow scalability by storing Fuzzy Indexes of per-flow features instead of raw data (see §7.3).

4.3 Primitive Fusion

In many systems, fusion is employed to optimize resource utilization [6, 19, 62]. These techniques, commonly applied on hardware accelerators such as GPUs and TPUs, fuse multiple operators into a single execution kernel to reduce intermediate memory access and

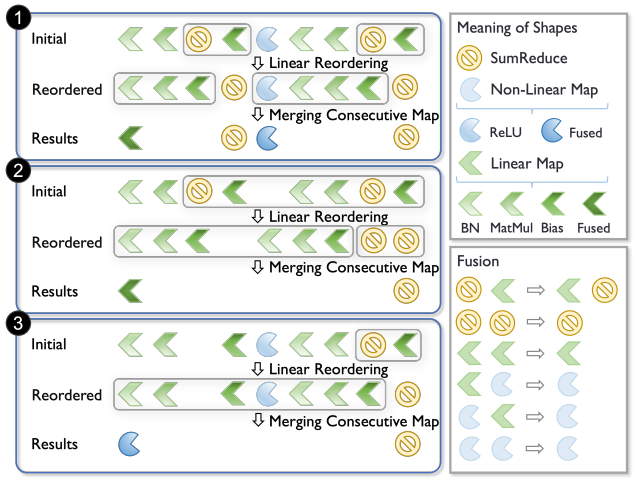


Figure 5: Primitive Fusion techniques (based on an MLP model): ❶ Basic Primitive Fusion; ❷ Advanced Primitive Fusion with Removal of Nonlinear Mappings; ❸ Advanced Primitive Fusion with Reduction of SumReduce.

kernel launch overhead. However, they do not reduce the number of computations, as each operator in the fused kernel still executes in full. In contrast, we further optimize our primitive implementation through Primitive Fusion, which focuses on compressing multiple operations into a single table lookup. This not only eliminates intermediate memory traffic but also reduces the number of operations executed, making it particularly effective for resource-constrained dataplane environments.

As shown in Figure 4, an MAT firstly extracts specific fields from the input vector for Partition, and then performs table lookups to retrieve precomputed results of Map primitives, followed by executing corresponding Actions on these results, such as SumReduce primitives. This process aligns with $\text{SumReduce}\big(\text{Map}(\mathcal{F}, \text{Partition}(X))\big)$, where X denotes the input vector, and $\mathcal{F}$ represents a set of functions that are applied individually to each partitioned group.

Basic Primitive Fusion. We propose a general approach to fuse primitives **without modifying the model architecture**. Specifically, we introduce two simple techniques to realize this approach:

(1) Linear Reordering. If a SumReduce is followed by a Map whose function f satisfies the linearity property $f(a+b) = f(a) + f(b)$, we can swap the order of SumReduce and Map. This preserves correctness because applying f on each partition and then summing is equivalent to summing first and then applying f, provided that f is linear.

(2) Merging Consecutive Map Primitives. Because each Map function applies independently to each partition, consecutive Map operations can be merged into a single Map.

By leveraging these techniques, Pegasus can fuse complex sequences of primitives without altering the underlying model, thereby enabling more efficient inference pipelines. For example, consider an MLP with two hidden layers, where each hidden layer includes: (1) a BN layer that applies an element-wise linear transform ($\gamma \cdot \frac{x-\mu}{\sigma} + \beta$), (2) a FC layer that performs MatMul plus bias

addition, (3) a ReLU activation defined as $\max(0, x)$. The layout of the MLP is shown in the "initial" state of Figure 5 ❶. By leveraging basic primitive fusion, we are able to compress seven table lookups into just two (Fused Maps and Fused Non-Linear Maps in the "results" state), thereby eliminating five lookup operations and significantly reducing computational overhead. As a result, the entire MLP can now be executed through this compact two-step lookup process.

Advanced Primitive Fusion. To further reduce the overhead associated with table lookups, we propose **two modifications to the model architecture**. As illustrated in Figure 5, the key to achieving deeper fusion lies in addressing the nonlinear mappings and SumReduce primitives.

(1) Removal of Nonlinear Mappings. As shown in Figure 5 ❷, by removing the nonlinear mapping (i.e., the ReLU activation) from the two-layer MLP, the entire computation—originally composed of multiple BN and FC layers—collapses into a single linear transformation. This enables the model to be compressed into a single table lookup, regardless of the number of intermediate operations. However, while this approach is highly efficient, purely linear models often struggle to capture complex patterns and relationships within the data, potentially leading to a significant drop in accuracy.

(2) Preservation of Nonlinearity via SumReduce Reduction. To preserve nonlinearity in the model while still enabling advanced fusion, we propose a second approach. As shown in Figure 5 ❸, by retaining only the final SumReduce operation and removing all others, the model can fuse nonlinear transformations within each partition into a single table lookup. In the MLP example, this corresponds to performing the entire MLP within each feature group, followed by a final SumReduce to produce the model output. This method, similar to Neural Additive Models (NAM) [1], can effectively capture complex and nonlinear relationships within each segment. This method benefits from Fuzzy Matching, which allows for more data within each segment. The outputs from each partition are then aggregated through SumReduce, allowing for a straightforward yet comprehensive integration of global information while maintaining the independence of individual sub-models.

4.4 Mapping Optimization

Primitive Fusion allows us to cluster inputs only before the fused large operators, replacing the original inputs with centroids to reduce the pressure on the dataplane. However, this approach **inevitably introduces approximation errors**. To ensure the mapping table more accurately aligns with the model's actual output, we employ **backpropagation** to **dynamically adjust** the stored centroids and cluster parameters, making it closer to the ideal performance.

Backpropagation. Pegasus first trains an initial model on the training dataset to generate cluster parameters and centroids in the mapping table. Subsequently, Pegasus constructs mapping tables and performs centroid assignment within the model using the technique from Zhang [56]. This method simulates the decision process of a decision tree via matrix operations, where the decision logic is functionally equivalent to our centroid assignment. Rather than relying on the original non-differentiable centroid assignment, we leverage the differentiable matrix operations to apply backpropagation, fine-tuning the cluster parameters and centroids to better align with the model's output while minimizing performance degradation.

DL Layers	DL Operators
Emb	Embedding Lookup
FC	Matrix Multiplication (Weighted Aggregation) Bias Addition (Element-wise Transformation)
Conv	Convolution (Weighted Aggregation)
Act	ReLU, tanh (Element-wise Transformation) Softmax (Multi-Input Operation)
Norm	Batch Normalization (Element-wise Transformation) Layer Normalization (Multi-Input Operation)
Pool	Pooling (Multi-Input Operation)
Rec	Matrix Multiplication (Weighted Aggregation) Bias Addition (Element-wise Transformation) tanh, Sigmoid (Element-wise Transformation) Hadmard (Element-wise Transformation)

Table 4: Operators in DL layers.

Adaptive Fixed-Point Quantization. During the inference process, the fixed-point positions of inputs and outputs can differ, especially when there are significant differences in numerical ranges (e.g., input range [-100, 100] versus output range [0, 5]). Some inference hardware employs Post-Training Static Quantization [32], which pre-defines the fixed-point positions for each layer's weights and activations based on known numerical ranges. This method helps maximize register bit-width utilization and improve numerical precision during inference.

In Pegasus, since the mapping table stores operations at full precision, we only need to perform fixed-point quantization on the final outputs before the SumReduce primitive. We pre-calculate the fixed-point positions and store the corresponding outputs in a mapping table. This approach allows Map primitives to handle inputs and outputs with different fixed-point positions, enhancing precision, particularly when there is a mismatch in numerical ranges. By optimizing in this manner, Pegasus flexibly processes data across varying ranges without sacrificing computational accuracy.

5 Deep Learning Operators

In deep learning (DL), layers are the building blocks of neural networks, each designed to perform specific transformations on the input. DL layers are typically constructed from a set of DL operators, as outlined in Table 4, which maps layers to their corresponding operators. In this section, we explain how Pegasus primitives can be used to implement these DL operations. All references to DL layers are focused on the inference phase.

• **Embedding Lookup.** Embedding Lookup is commonly used in embedding layers during inference, mapping discrete input indices to dense vectors. It can be viewed as an indexing function $f(x) = E[x]$, efficiently implemented using the Map primitive.

• **Element-wise Transformation.** Element-wise Transformation refers to operations performed independently on each element of the input, making it naturally suitable for implementation using the Map primitive. During inference, most parameters, such as weights, biases, and other model parameters, are known in advance. These can be treated as constants, part of the function rather than inputs, reducing the computational overhead during the mapping process.

• **Weighted Aggregation.** Weighted Aggregation is the most computationally intensive operation in DL [10, 17, 40], generating output by performing element-wise multiplication between input elements and their corresponding weights, followed by summing the results. This operation can be partitioned into multiple parts, with each part processed using the Map primitive to obtain the corresponding result vectors. The final output is then obtained by applying the SumReduce primitive to aggregate these partial results.

• **Multi-Input Operation.** Multi-Input Operation refers to computations where an element's output depends on multiple input elements. These inputs may be too numerous to fit into a single partition due to combinatorial explosion. There are two common ways to implement this operation. The first method uses the Map primitive to process each partition, then apply the SumReduce primitive to aggregate their influence on the output, followed by Map primitives to operate on the aggregated result and produce the final output. For example, Softmax (defined as $\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$) involves a Map primitive to exponentiate each element e^{x_i}, followed by a SumReduce primitive to sum these values $\sum e^{x_i}$, and a final Map primitive to normalize each element by this sum $\frac{e^{x_i}}{\sum_j e^{x_j}}$. The second method uses consecutive Map primitives to progressively compute operations between multiple elements. For instance, average pooling requires several Map primitives to iteratively compute the average value, yielding the final result.

6 Implementation

Pegasus is generalizable to commodity programmable switches, such as PISA-based [4] and Trio-based [53] switches, which support the P4 language. This generalizability stems from Pegasus's reliance solely on comparisons, table lookups, and additions. To demonstrate its practicality, we have implemented Pegasus on the PISA switch.

While the core logic of Pegasus maps well to P4-compatible programmable switches, deploying it on real hardware still involves several architectural constraints and engineering considerations. In this section, we first describe the platform-specific design and implementation of Pegasus on PISA switches, addressing hardware constraints such as limited arithmetic and matching capabilities (§6.1). We then introduce a syntax-level abstraction, Pegasus Syntax, which allows users to define models without dealing with low-level P4 programming (§6.2). Finally, we introduce the DL models we implemented to evaluate Pegasus (§6.3).

6.1 PISA-specific Design and Implementation

Fixed-point Aggregation. Pegasus uses the Map to generate result vectors, and aggregates them via SumReduce. However, PISA switches do not support native aggregation operations, and each pipeline stage can perform at most one addition per element. As a result, Pegasus must immediately accumulate each element of the result vector into the final output during the Map phase. Each stage, however, supports only a limited number of 8-bit, 16-bit, and 32-bit fields along with their corresponding table-related ALUs. As shown in Figure 6 ❶, 8-bit fields are **insufficient** for 8-bit fixed-point addition, as this can lead to severe and unpredictable **overflow** issues, significantly impacting performance. This necessitates using

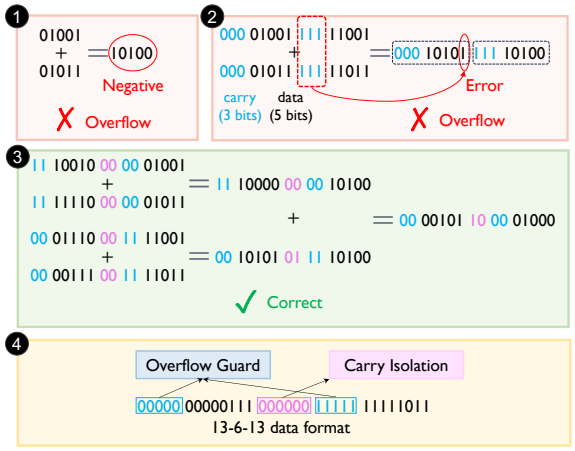


Figure 6: ❶ ❷ show issues with implementing fixed-point aggregation on the dataplane; ❸ shows the correct approach, and ❹ presents our proposed data format.

larger bit-width fields (e.g., 16-bit) for addition, with higher bits as an Overflow Guard to manage overflow effectively. However, the limited number of 16-bit ALUs makes large-vector aggregation challenging. To improve efficiency, we adopt 32-bit fields in specific scenarios to aggregate two 8-bit dimensions concurrently.

Yet, directly packing two 16-bit values into a 32-bit field can still cause overflow in the lower half, potentially corrupting the upper value (see Figure 6 ❷). To address this, as shown in Figure 6 ❹, we adopt a 13-6-13 layout, where each 8-bit value is extended to 13 bits with overflow guard bits, and a 6-bit isolation region is inserted in between to block inter-field overflow. This format safely supports the accumulation of at least 32 signed values per field and remains robust under highly skewed inputs.

Fuzzy Matching Implementation. In Pegasus, the input traverses the clustering tree to obtain the Fuzzy Index. This process requires a multi-level comparator, which is not natively supported by PISA-based switches. To address this, we use the numerical range of values to represent the leaf nodes of the clustering tree. enabling efficient implementation via range matching. This also allows us to mask unused bits, making the design compatible with our 13-6-13 fixed-point format. To efficiently convert these ranges into ternary rules, we introduce the Consecutive Range Coding (CRC) algorithm [64], which enables the effective transformation of numerical ranges into ternary rules.

Despite this, additional optimization is needed to reduce the cost of Fuzzy Indexing in hardware. In software, a 4-bit index is a typical choice for partitioned sub-vectors, but such fine-grained representation leads to excessive TCAM and transfer overhead. We therefore design simple strategies based on original sub-vector size to reducing TCAM usage and data transfer cost.

(1) One-dimensional Partitioning. When applying the CRC method, splitting 15 thresholds within an 8-bit number typically yields no more than 32 ternary rules. Allocating an entire TCAM query resource for such a small rule set would be inefficient. Instead, we increase the partition granularity to two dimensions, which allows us to encode the combinations using only 1024 entries, thereby

```
struct InputVec_t {
    bit<8>  input_dim0;
    …
    bit<8>  input_dim7;
}; /* Definition of OutputVec_t is eliminated. */
struct ig_metadata_t {
    InputVec_t input_vec;
    OutputVec_t output_vec;
};
ig_metadata_t meta;

meta.output_vec = SumReduce(
    Map(
      Partition(meta.input_vec, dim = 2, stride = 2),
      clustering_depth = 4,
      CNN_dimension = 3,
      CNN_kernel = cnn_kernel,
      CNN_stride = cnn_stride
    )
);
```

Figure 7: Pegasus Syntax.

halving both the number of TCAM queries and the data transfer volume, while the actual TCAM slot usage remains comparable.

(2) Multi-dimensional Partitioning. Using the CRC method to encode two elements into 16 or fewer fuzzy regions does not significantly reduce TCAM overhead compared to composing two one-dimensional Fuzzy indices [64]. As a result, unlike One-dimensional Partitioning, we cannot directly combine multiple partitions into a single TCAM query. Instead, we encode each pair of 8-bit values into a 4-bit Fuzzy Index, and aggregate three such indices for an SRAM-based lookup. This design reduces data transfer cost at the expense of modest SRAM consumption.

6.2 Pegasus Syntax

To facilitate the implementation of various DL models on the Pegasus framework, we have designed a specialized syntax called Pegasus Syntax. Figure 7 illustrates the proposed syntax, which provides a high-level abstraction for defining and configuring DL models. To support the translation of Pegasus Syntax into P4 language, we developed a translation tool. This tool significantly reduces programming complexity, allowing developers to focus on high-level logic design without delving into the intricacies of low-level P4 code.

Specifically, our Pegasus Syntax maintains a consistent form with the primitives. In Partition phase, input data and its partitioning rules are explicitly specified. The partitioned data in each segment is used to perform Map operations. In the Map phase, we define the depth of the clustering tree and a series of CNN parameters to determine the output dimensions for each group of inputs. The translator automatically calculates the output dimensions based on these parameters. This design is motivated by the fact that certain operations, such as the convolution process in CNNs, are partially connected. Reducing the output dimensions of the Map primitives can effectively minimize table resource overhead. The specific allocation of hardware resources is automatically handled by the translator.

6.3 Implemented Neural Networks

We implemented the following six representative DL models[2] within Pegasus, all of which utilize the Fuzzy Matching and Basic Primitive Fusion. Additionally, we applied the Advanced Primitive Fusion technique in CNN-M, CNN-L and AutoEncoder to enable larger model scales with lower overhead, achieving improved accuracy.

MLP-B. MLP is well-suited for handling high-dimensional data, making it particularly effective in processing statistical features. We implemented a basic MLP model (MLP-B) that operates on statical features, including flow-level and packet-level features. However, It's hard to extract effective statistical features for MLP on the dataplane. For example, calculating averages is challenging on programmable switches, while using cumulative sums can lead to overfitting to large flows. To ensure fairness, we only use the maximum and minimum packet lengths and inter-packet delays (IPD) as flow-level information. Our MLP-B consists of three hidden layers, each comprising a sequence of Batch Normalization, a FC layer, and a ReLU activation function.

RNN-B. RNNs are well-suited for capturing temporal dependencies in sequential data, making them particularly effective for handling time-series features. Our implementation is based on the windowed binary RNN design in BoS [51], which processes multiple time steps on the switch to capture sequential dependencies without requiring hidden state write-backs. It classifies packets based on the sequence of packet lengths and inter-packet delays (IPD). The RNN-B model consists of an Emb layer, a tanh activation function, and multiple FC layers.

CNN-B, CNN-M, and CNN-L. The one-dimensional CNN demonstrates unique advantages in processing windowed sequence data [24]. We implemented three CNN models: CNN-B (basic), CNN-M (medium), and CNN-L (large), with increasing model complexity and scalability. CNN-B serves as the baseline model, employing only the Basic Primitive Fusion technique. It uses packet length and IPD sequences as input features. CNN-M extends CNN-B by incorporating Advanced Primitive Fusion, enabling larger model scales with lower overhead. CNN-L builds on CNN-M by further leveraging Advanced Primitive Fusion to support even larger model sizes and input scales. This enables CNN-L to extract 60 raw bytes from each packet as a raw packet sequence. Specifically, CNN-L supplements the packet-level features used in the MLP-B model with additional bytes extracted from the packet payload (i.e., the content beyond the TCP header), padding the input to a fixed size of 60 bytes. This feature construction is inspired by recent work in traffic analysis using large language models [46, 58], which emphasizes the semantic value of raw packet content. All three models are based on the textcnn architecture proposed by Zhang et al. [57], consisting of multiple Conv layers, FC layers, Pool layers, and ReLU activation functions.

AutoEncoder. Autoencoders are effective for unsupervised anomaly detection by learning compact representations and reconstructing input data. Our implementation uses mean absolute error (MAE) to calculate reconstruction error, which is then used to determine whether a flow is anomalous. The Autoencoder model consists of an

[2] In addition to the six DL models mentioned in the text, their variants can also be implemented using these methods. Note that large models, such as Transformers, cannot be supported due to resource limitations.

Emb layer and multiple FC layers for encoding and decoding. Each FC layer is preceded by a Batch Normalization layer and followed by a ReLU activation function.

7 Evaluation

Our evaluation addresses the following questions: (i) Whether Pegasus can achieve higher accuracy and generality in supporting a variety of DL models? (§7.2). (ii) How scalable is Pegasus across model size, input scale, and the number of simultaneous flows? (§7.3). (iii) Whether the unsupervised models implemented by Pegasus can effectively defend against real-world unknown attack traffic? (§7.4). (iv) What advantages DL model implementations on the dataplane offer compared to those on the control plane? (§7.5).

7.1 Experiment Setup

Testbed Setup. We implemented Pegasus using P4 [4] on a Barefoot Tofino 2 programmable switch [7], connected to two Linux servers. One server replays pcap files via tcpreplay, while the other server receives packets from the programmable switch.

Traffic Classification Datasets. We use three publicly available and widely used traffic classification datasets, which are also utilized in BoS [51]: (i) PeerRush [34]: This dataset contains traffic generated by P2P applications, categorized into three classes (eMule, uTorrent, and Vuze). (ii) CICIOT2022 (CICIOT) [9]: This dataset contains traffic collected from IoT devices in different working states, categorized into three classes: Power, Idle, and Interact. (iii) ISCXVPN2016 (ISCXVPN) [16]: This dataset consists of VPN-encrypted network traffic, categorized into seven classes (Email, Chat, Streaming, FTP, VoIP, P2P). For each dataset, we selected 75% of the flows (identified by five-tuple) from each class to train the DL models, 10% for validation, and 15% for testing.

Baselines. We implemented N3IC [38], Leo [25], and BoS [51], using the largest model configurations specified in their respective papers. Among these, Leo and BoS were deployed on the switch, while N3IC was evaluated through software simulation because the largest models in their papers could not be implemented on the switch. It is important to note that our evaluation focuses solely on the accuracy of the models themselves. We did not employ common optimization techniques, such as those in BoS, which enhance accuracy by aggregating predictions from multiple packets within a flow and offloading hard-to-classify cases to the control plane via the Integrated Model Inference System (IMIS).

Metrics. Consistent with prior works [51, 64], we use packet-level macro-accuracy, defined as the average F1-score across different classes, to evaluate model accuracy. Unless otherwise specified, all accuracy measurements in the evaluation refer to macro-accuracy. Additionally, we report the overall Precision (PR) and Recall (RC) to provide a comprehensive evaluation of the models.

7.2 Accuracy Comparison and Analysis

In this section, we compare the accuracy of Leo, N3IC, and MLP-B using the same statical features; BoS, RNN-B and CNN-B using the same raw packet sequences. These models serve to demonstrate that even a direct implementation of prior architectures using our method can already superior accuracy compared to the original designs. Detailed analysis of CNN-M and CNN-L, which leverage Advanced Primitive Fusion to improve scalability and enhance model accuracy, is deferred to §7.3. Table 5 summarizes the input scale, model size (i.e., the total size of parameters involved in computation, excluding embedding layers), and classification accuracy for all evaluated models.

Statical Features. As shown in Table 5, MLP-B achieves better accuracy than N3IC, with improvements ranging from 5.8% to 11.9%, despite the two models having similar sizes. This illustrates the accuracy degradation caused by the full-model binarization in N3IC, particularly in the absence of Norm and Act layers. In contrast, Pegasus uses full-precision model weights and fixed-point activations to enhance accuracy. These design choices allow Pegasus to maintain its accuracy advantage even under the Fuzzy Matching-induced errors (See §7.5 for accuracy degradation compared to CPU/GPU-based execution).

Compared to Leo, Pegasus achieved a 7.3% accuracy improvement on the CICIOT dataset. This advantage may stem from the complex relationships among CICIOT features, which an MLP model of this scale can capture more effectively than tree-based approaches. However, the improvement remains modest on the PeerRush and ISCXVPN tasks, with only an average 1.0% gain. This is consistent with the fact that decision trees perform well on statistical features. Nevertheless, DL models excel at processing raw packet sequences, a capability we will demonstrate through the CNN-L implementation (see 7.3).

Raw Packet Sequence. As shown in Table 5, despite using full-precision model weights, the BoS still exhibits a 4.1% to 7.1% lower accuracy than RNN-B across the three traffic classification tasks. This confirms the impact of input and output binarization on model accuracy.

Additionally, the CNN-B model demonstrates comparable accuracy to the RNN-B model but performs slightly lower, with an average gap of 0.6%. This may be attributed to RNN's superior ability to capture sequential dependencies under the same model size.

7.3 Scalability Evaluation

MLP models are constrained by the switch's limited ability to extract complex statistical features, while RNN models face implementation challenges on the switch due to the requirement for sequential execution over multiple time steps. Given these limitations, we select CNN models to evaluate the impact of scalability on classification accuracy, as their accuracy is significantly affected by model size.

Model Scale Scalability. As shown in Table 5, as the model size increases, CNN-M achieves accuracy improvements of 1.5% to 2.6% over CNN-B, while outperforming RNN-B by 1.2% to 1.6%. This improvement is not proportional to model size, as larger models face diminishing returns due to feature saturation and dataset complexity limits. Nevertheless, CNN-M achieves these gains with lower overhead compared to CNN-B (see §7.4). By leveraging Advanced Primitive Fusion, CNN-M significantly reduces the number of tables, optimizing resource utilization.

To further improve traffic classification accuracy, we expanded the feature set and model size. With this enhancement, CNN-L demonstrates exceptional accuracy, achieving 99.66%, 93.80%, and

Method	Input Scale (b)	Model Size (Kb)	PeerRush			CICIOT			ISCXVPN		
			PR	RC	F1	PR	RC	F1	PR	RC	F1
Leo [25] (Decision Tree)	128	-	0.8720	0.8776	0.8728	0.7910	0.8072	0.7848	0.7338	0.7797	0.7475
N3IC [38] (binary MLP)	128	24.4	0.8217	0.8308	0.8241	0.7855	0.7877	0.7745	0.6688	0.6521	0.6388
MLP-B	128	34.3	0.8823	0.8826	0.8823	0.8555	0.8615	0.8581	0.7676	0.7552	0.7574
BoS [51] (binary RNN)	18	25.6	0.8677	0.8696	0.8678	0.8311	0.8253	0.8276	0.7033	0.7089	0.6907
RNN-B	128	10.9	0.9083	0.9100	0.9090	0.8707	0.8708	0.8707	0.7848	0.7658	0.7617
CNN-B	128	11.4	0.9051	0.9069	0.9057	0.8861	0.8657	0.8659	0.7706	0.7600	0.7520
CNN-M	128	974	0.9201	0.9220	0.9207	0.8821	0.8839	0.8829	0.7942	0.7897	0.7780
CNN-L	**3840**	**6083**	**0.9967**	**0.9966**	**0.9966**	**0.9391**	**0.9377**	**0.9380**	**0.9868**	**0.9877**	**0.9872**

Table 5: Comparison of classification accuracy across different methods.

98.72% on the PeerRush, CICIOT, and ISCXVPN datasets, respectively. This represents an improvement of 7.2% to 23.5% over CNN-B, and average accuracy gains of 17.2%, 22.8% and 17.9% over Leo, N3IC, and BoS, respectively.

Additionally, CNN-L has a model size of 6083Kb, which is 248x and 237x larger than those of N3IC (24.4Kb) and BoS (25.6Kb), respectively (see §7.4). CNN-L also supports input sizes of 3840 bits, representing a 29x increase over N3IC (128 bits) and a 212x increase over BoS (18 bits). Traditional methods struggle to support such large input sizes for two main reasons: (a) PISA switches only have 4096-bit Packet Header Vector (PHV), making it difficult to handle such large-scale features while supporting basic functionalities. CNN-L benefits from the design of primitives. Specifically, CNN-L uses Partition to divide the input, distributing the inference process across each packet within the window. Each packet only processes 480 bits of features, allowing CNN-L to successfully implement. (b) Excessive per-flow register usage can impact the number of concurrent flows that can be supported (see below). This demonstrates Pegasus's exceptional scalability.

Number of Concurrent Flows Supported. Storing per-flow features on the switch requires the use of its stateful SRAM resources. Supporting more per-flow features reduces the number of concurrent flows that can be managed, which is why previous works did not support features at the same scale as CNN-L on the dataplane.

However, Pegasus achieves this at a low cost. For scenarios requiring a larger number of per-flow features, such as CNN-L, Pegasus executes an expressive convolutional neural network directly on the switch to compress 60 raw per-packet features (480 bits) into 2 compact features. As shown in Figure 8, increasing the feature width from 2 to 4 brings negligible accuracy improvement, which justifies our compact design choice. This compressed representation is further reduced through Fuzzy Matching, which maps the features into 4-bit Fuzzy Indexes stored in the mapping table (this technique is broadly used across our models to minimize per-flow register usage; see Table 6). These two steps together significantly reduce per-flow storage overhead. We defer the detailed implementation of CNN-L to Section §A.

Figure 8 summarizes how classification accuracy varies with the per-flow storage overhead, where the X-axis corresponds to the required SRAM overhead to support 1 M flows for different per-flow storage sizes. The CNN-L model uses 48 bits per flow, including 16 bits for the previous packet timestamp (used for IPD calculation) and 4 bits for the Fuzzy Index extracted from each packet (for a window size of 8, the features of 7 packets need to be stored). Additionally, a 28-bit version of CNN-L removes the IPD feature, while a 72-bit version extracts 8 bits for the Fuzzy Index (corresponding to 4 compressed features) from each packet [3]. Even with 28 bits of per-flow storage, the model achieves classification accuracies of 99.1%, 92.9%, and 97.2% on the PeerRush, CICIOT, and ISCXVPN datasets, respectively. Compared to Leo, N3IC, and BoS, it improves the average accuracy by 16.2%, 21.8%, and 16.9%, respectively. Moreover, the 28-bit per-flow storage usage is significantly lower than BoS's 72-bit usage and the 80-bit usage of Leo and N3IC (see §7.4).

7.4 Unsupervised Malicious Traffic Detection Evaluation

Previous works have predominantly focused on leveraging learning models to classify traffic on the dataplane under scenarios with abundant labeled data. However, in real-world networks, attacks often come from unknown traffic, such as zero-day attacks. It is unrealistic to anticipate such attack traffic in advance and train supervised models accordingly. Detecting unknown traffic through unsupervised models is challenging as it requires extracting multiple complex features from the traffic, reconstructing the original inputs using large model structures, and determining whether the traffic is malicious based on reconstruction errors [30, 41]. This complexity has prevented prior works from addressing this area on the dataplane.

In this section, we validate that the AutoEncoder implemented by Pegasus can utilize a large model structure to extract features from raw packet sequences (packet length and IPD) and identify unknown attack traffic by calculating reconstruction errors using MAE. Specifically, the model leverages knowledge learned in the Emb layer during traffic classification tasks to capture relationships and features from raw packet sequences. These features are reconstructed through the encoder and decoder.

Datasets. We use the AutoEncoder to reconstruct traffic on the training sets of the PeerRush, CICIOT, and ISCXVPN datasets. To evaluate the model's ability to detect unknown attacks, we inject two representative malicious traffic at a 1:4 mixture of attack-to-benign traffic into the testing set: (a) Malware Attack, including Cridex, Geodo, Htbot, Neris, and Virut, sourced from USTC-TFC2016 [47]. (b) DoS Attack, utilizing SSDP Reflection Flood traffic, collected from Kitsune [30].

[3] In fact, since PISA switches do not support 4-bit registers, we actually used 4 8-bit registers to replace the 7 4-bit registers.

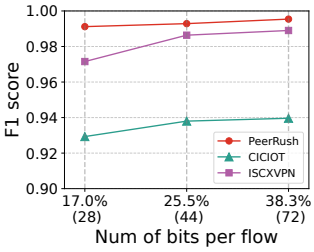
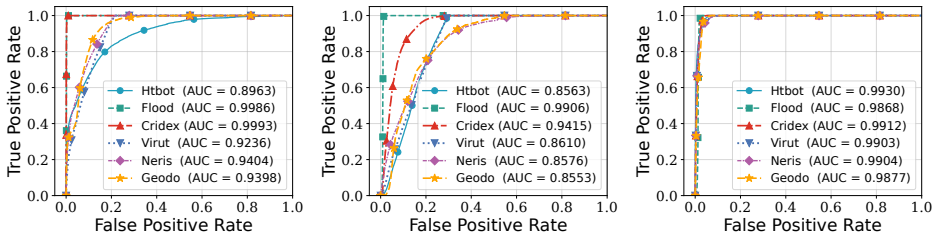


Figure 8: Impact of per-flow storage usage on classification accuracy.



(a) PeerRush (b) CICIOT (c) ISCXVPN

Figure 9: ROC curves across different datasets.

Metrics. We use AUC (AURoC, Area Under the Receiver Operating Characteristic Curve) as metrics, as these are commonly used in existing studies [11, 13, 20, 66]. AUC measures the model's ability to distinguish between normal and malicious traffic.

Results. As shown in Figure 9, the AutoEncoder achieves average AUCs of 95.0%, 89.4%, and 99.0% on the PeerRush, CICIOT, and ISCXVPN datasets, respectively, across different types of malicious traffic. This demonstrates that the AutoEncoder can effectively distinguish normal traffic from anomalous traffic when only normal traffic is available during training. In practical deployments, programmable switches can dynamically adjust response strategies based on the MAE value and its abnormal fluctuations. For instance, they can enforce traffic rate limits or send real-time alerts to administrators, enabling the system to handle potential malicious traffic or attacks more efficiently. This system can be integrated into a broader security analysis framework to fully leverage its capabilities in detecting anomalous traffic and enabling timely, adaptive responses.

Hardware Resource Utilization. Unlike the accuracy evaluation, we implemented moderately sized versions of BoS (with a hidden size of 8) and Leo (with 1024 nodes) to assess resource overhead, as models like BoS are inherently designed for small-scale scenarios.

We report the stateful per-flow bit usage, stateless SRAM and TCAM overhead, and Action Data Bus utilization for implementing different methods on the switch in Table 6. In Pegasus, TCAMs are used to retrieve the Fuzzy Index, the SRAMs are used to store mapping tables, and the Action Data Bus are used to transfer data fetched from SRAM/TCAM.

Compared to CNN-B, CNN-M has a larger model size but lower resource overhead. This is primarily due to the fusion of all intermediate-layer operations through Advanced Primitive Fusion, which improves resource utilization. This effect is even more pronounced in CNN-L. Despite having a model size of 6083Kb, CNN-L only occupies 7.12% of SRAM and 13.33% of TCAM. The majority of the model parameters are fused, so they do not occupy storage resources during inference.

Compared to BoS and Leo, CNN-L has a larger resource overhead, which is understandable given its larger model scale and higher numerical precision. However, this drawback is alleviated as the number of concurrent flows increases, because CNN-L has a lower per-flow register usage. As shown in Figure 8, when supporting 1M concurrent flows, the 28-bit per-flow storage version of CNN-L saves 21.3% of SRAM overhead compared to the 72-bit requirement, thereby significantly mitigating this drawback.

Models	Stateful bits/flow	SRAM	TCAM	Bus
Leo	80	2.44%	21.67%	3.55%
BoS	72	2.81%	0%	0.74%
MLP-B	80	7.75%	12.92%	29.45%
RNN-B	240	7.38%	23.33%	33.36%
CNN-B	72	5.56%	7.08%	13.16%
CNN-M	72	3.50%	6.67%	3.98%
CNN-L	44	7.12%	13.33%	7.11%
AutoEncoder	240	5.06%	7.92%	7.23%

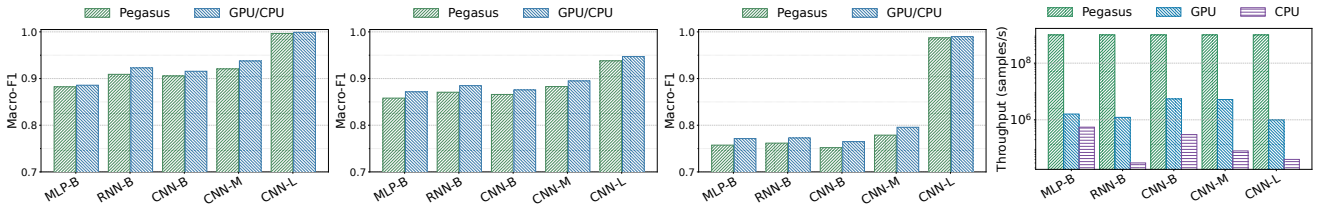
Table 6: Hardware resource utilization for different methods.

7.5 Compare With Control Plane DL

Pegasus uses Fuzzy Matching instead of precise computation to perform DL inference, which inevitably reduces model accuracy. To evaluate the impact of Pegasus on accuracy and throughput improvements, we implemented full-precision DL inference on the edge using an Intel Xeon E5-2699 v4 CPU and four Tesla V100 GPUs. Since the PISA pipeline on programmable switches ensures that any program compiled for it can run at line-rate, the size of the DL model does not affect dataplane throughput. To maximize control plane throughput, we pre-loaded features into CPU memory and GPU VRAM, using multi-threading to fully utilize all CPU cores and four GPUs, minimizing communication overhead. The accuracy and throughput comparison results are shown in the Figure 10.

The results indicate that Pegasus results in an average reduction of 1.08% in model accuracy, ranging from 0.2% to 1.7%. Notably, the CNN-L model, which features richer inputs and higher model capacity, experiences below-average accuracy loss (0.3%, 0.2%, and 0.9% across three datasets). This encourages us to fully leverage Pegasus's potential in designing more powerful models, rather than limiting its use to simple, small-scale neural networks.

However, it increases throughput by over 3800× and 600× compared to CPU and GPU, respectively. This throughput improvement represents the idealized capacity of these devices. In real-world conditions, although the gap may be narrower, the throughput gains would still be significant. Given the substantial throughput improvement, the reduction in accuracy can be considered acceptable.



(a) PeerRush (b) CICIOT (c) VPN (d) Throughput

Figure 10: (a–c) Comparison of classification accuracy for different models implemented on the programmable switch v.s. CPU/GPU across various datasets. (d) Throughput comparison for models on the programmable switch v.s. CPU/GPU.

8 Discussion and Related Work

Data-Driven Traffic Analysis. Researchers have proposed various methods for intelligent traffic analysis [15, 29], such as encrypted traffic classification, website fingerprinting and malicious traffic detection. There is a growing recognition of the benefits of performing traffic analysis at line-rate. NetBeacon [64] and Leo [25] leverage decision trees in the dataplane to enable IDP. However, like N3IC [38], these methods face the challenge of extracting complex features directly on the dataplane. BoS [51] addresses this limitation by using DL to automatically extract features, supporting IDP without manual feature engineering on the dataplane. Building on this foundation, Pegasus further enhances the capability of executing deep learning inference within the dataplane.

Hardware Dependency. Taurus [39], Trio [53], Trident [23] have explored adding computational capabilities to the dataplane to support IDP. However, achieving line-rate computation is often prohibitively expensive and difficult to integrate with the fundamental operations of the dataplane, such as table lookups and packet forwarding. In contrast, Pegasus is specifically designed to align with the flow-table-centric architecture of the dataplane. Operations like multi-level comparisons and fixed-point addition, which Pegasus relies on, can be more efficiently implemented on other dataplane devices. In fact, the majority of our overhead is caused by the limitation of the Barefoot Tofino architecture. We believe that with lightweight hardware adjustments, Pegasus could enable more advanced capabilities for IDP.

Deployment in Real-World Environments. Pegasus is designed to implement DL models on the dataplane, allowing users to balance accuracy and resource overhead based on their specific requirements. Beyond its standalone utility, our approach can also be combined with other analysis techniques to achieve better overall effectiveness. While such integrated deployments can help reduce overall resource consumption, addressing this challenge is beyond the scope of this work and is left for future investigation. Additionally, IDP may encounter issues with limited flow registers, particularly in extreme conditions. This limitation is inevitable in scenarios requiring stateful functionalities [12, 27, 50]. For such challenges, prior works such as AIFO [55] and P4LRU [59] offer more relevant solutions. These methods can store flow characteristics for large flows and utilizing packet features to identify small flows, ultimately achieving higher classification accuracy.

Practical Deployment of Fuzzy Matching. As a core mechanism in Pegasus, Fuzzy Matching may face efficiency issues in practical deployment, especially when inputs deviate significantly from the training set. However, such deviations violate the i.i.d. assumption in machine learning and often indicate a shift in traffic behavior—potentially due to emerging malicious activities. In such cases, our unsupervised model plays a key role in detecting abnormal patterns. Meanwhile, the control plane can enhance robustness by continuously collecting evolving flow features and updating the model to adapt to distributional drift. Although training Fuzzy Matching takes longer than conventional models—due to the construction of Fuzzy Matching and quantization optimization—this cost arises only at deployment or during major shifts, and can be amortized over long-term use. In practice, retraining is infrequent and incurs minimal overhead.

9 Conclusion

The limited computational resources of programmable switches are not the root cause hindering intelligence realization; rather, it is the ineffective use of the MAT abstraction that becomes the real obstacle. Simple but useful, Pegasus expresses DL models using dataplane-friendly primitives, enabling implementation on commodity programmable switches without requiring additional complex computational resources. The primary goal of Pegasus is to address the accuracy, scalability, and generality limitations of prior IDP designs. Experimental results demonstrate Pegasus's advantages in realizing intelligent models. It serves as a viable option for line-rate DL inference and offers an alternative to the growing trend of continuously adding line-rate computational resources to the dataplane.

Ethics: *This work does not raise any ethical issues.*

Acknowledgement

We thank the anonymous reviewers and our shepherd Vishal Shrivastav for their insightful comments and suggestions which help improve this paper. We also thank Yuhang Li from Tsinghua University, Xiangsheng Zeng and Fu Song, interns from Tsinghua University, for conducting the experiments. This work was supported in part by the National Science Foundation for Distinguished Young Scholars of China under No. 62425201, the National Natural Science Foundation of China under U22B2031, No. 62472240, No. 62394322, No. 62472036, No. 62202258, No. 62202473, No. 62441230, No. 62132011, No. 62472247, the Beijing National Research Center for Information Science and Technology under Grant BNR2025RC01010. Ke Xu (xuke@tsinghua.edu.cn) and Su Yao (yaosu@tsinghua.edu.cn) are the corresponding authors.

References

[1] Rishabh Agarwal, Levi Melnick, Nicholas Frosst, Xuezhou Zhang, Ben Lengerich, Rich Caruana, and Geoffrey E Hinton. 2021. Neural additive models: Interpretable machine learning with neural nets. *Advances in neural information processing systems* 34 (2021), 4699–4711.

[2] Abd AlRhman AlQiam, Yuanjun Yao, Zhaodong Wang, Satyajeet Singh Ahuja, Ying Zhang, Sanjay G Rao, Bruno Ribeiro, and Mohit Tawarmalani. 2024. Transferable Neural WAN TE for Changing Topologies. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 86–102.

[3] Davis Blalock and John Guttag. 2021. Multiplying matrices without multiplying. In *International Conference on Machine Learning*. PMLR, 992–1004.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[7] Intel Corporation. 2024. Barefoot Tofino 2. https://www.intel.cn/content/www/cn/zh/products/details/network-io/intelligent-fabric-processors/tofino-2.html.

[8] Intel Corporation. 2024. Barefoot Tofino Series. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

[9] Sajjad Dadkhah, Hassan Mahdikhani, Priscilla Kyei Danso, Alireza Zohourian, Kevin Anh Truong, and Ali A Ghorbani. 2022. Towards the development of a realistic multidimensional IoT profiling dataset. In *2022 19th Annual International Conference on Privacy, Security & Trust (PST)*. IEEE, 1–11.

[10] Jeff Dean, David Patterson, and Cliff Young. 2018. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro* 38, 2 (2018), 21–29.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[12] Yong Feng, Zhikang Chen, Haoyu Song, Yinchao Zhang, Hanyi Zhou, Ruoyu Sun, Wenkuo Dong, Peng Lu, Shuxin Liu, Chuwen Zhang, et al. 2024. Empower programmable pipeline for advanced stateful packet processing. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 491–508.

[13] Chuanpu Fu, Qi Li, Meng Shen, and Ke Xu. 2024. Detecting tunneled flooding traffic via deep semantic analysis of packet length patterns. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3659–3673.

[14] Massimo Gallo, Alessandro Finamore, Gwendal Simon, and Dario Rossi. 2020. Real-time deep learning based traffic analytics. In *Proceedings of the SIGCOMM'20 Poster and Demo Sessions*. 76–78.

[15] Xiangyu Gao, Tong Li, Yinchao Zhang, Ziqiang Wang, Xiangsheng Zeng, Su Yao, and Ke Xu. 2025. FENIX: Enabling In-Network DNN Inference with FPGA-Enhanced Programmable Switches. *arXiv preprint arXiv:2507.14891* (2025).

[16] Gerard Drapper Gil, Arash Habibi Lashkari, Mohammad Mamun, and Ali A Ghorbani. 2016. Characterization of encrypted and VPN traffic using time-related features. In *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP 2016)*. SciTePress Setúbal, Portugal, 407–414.

[17] Ian Goodfellow. 2016. Deep learning.

[18] Dongqi Han, Zhiliang Wang, Wenqi Chen, Kai Wang, Rui Yu, Su Wang, Han Zhang, Zhihua Wang, Minghui Jin, Jiahai Yang, et al. 2023. Anomaly Detection in the Open World: Normality Shift Detection, Explanation, and Adaptation.. In *NDSS*.

[19] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[20] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. 2021. New directions in automated traffic analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3366–3383.

[21] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 161–176.

[22] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2018. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research* 18, 187 (2018), 1–30.

[23] Broadcom Inc. 2024. Trident 5 / BCM78800 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800.

[24] Alon Jacovi, Oren Sar Shalom, and Yoav Goldberg. 2015. Understanding convolutional neural networks for text classification. *arXiv preprint arXiv:1809.08037* (2015).

[25] Syed Usman Jafri, Sanjay Rao, Vishal Shrivastav, and Mohit Tawarmalani. 2024. Leo: Online {ML-based} Traffic Classification at {Multi-Terabit} Line Rate. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1573–1591.

[26] John D Kelleher. 2019. *Deep learning*. MIT press.

[27] Alberto Lerner, Davide Zoni, Paolo Costa, and Gianni Antichi. 2024. Rethinking the Switch Architecture for Stateful In-network Computing. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. 273–281.

[28] Chenning Li, Arash Nasr-Esfahany, Kevin Zhao, Kimia Noorbakhsh, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. 2024. m3: Accurate Flow-Level Performance Estimation using Machine Learning. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 813–827.

[29] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication*. 256–269.

[30] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. 2018. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089* (2018).

[31] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. 2023. A Deep Learning Perspective on Network Routing. *arXiv preprint arXiv:2303.00735* (2023).

[32] PyTorch. 2024. Post Training Quantization (PTQ) — Torch-TensorRT. https://pytorch.org/TensorRT/tutorials/ptq.html. Accessed: 2024-09-18.

[33] Qiaofeng Qin, Konstantinos Poularakis, Kin K Leung, and Leandros Tassiulas. 2020. Line-speed and scalable intrusion detection at the network edge via federated learning. In *2020 IFIP networking conference (Networking)*. IEEE, 352–360.

[34] Babak Rahbarinia, Roberto Perdisci, Andrea Lanzi, and Kang Li. 2013. Peerrush: Mining for unwanted p2p traffic. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 10th International Conference, DIMVA 2013, Berlin, Germany, July 18-19, 2013. Proceedings 10*. Springer, 62–82.

[35] Michael Seufert, Katharina Dietz, Nikolas Wehner, Stefan Geißler, Joshua Schüler, Manuel Wolz, Andreas Hotho, Pedro Casas, Tobias Hoßfeld, and Anja Feldmann. 2024. Marina: Realizing ML-Driven Real-Time Network Traffic Monitoring at Terabit Scale. *IEEE Transactions on Network and Service Management* (2024).

[36] Vishal Shrivastav. 2022. Programmable multi-dimensional table filters for line rate network functions. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 649–662.

[37] Giuseppe Siracusano and Roberto Bifulco. 2018. In-network neural networks. *arXiv preprint arXiv:1801.05731* (2018).

[38] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. 2022. Re-architecting traffic analysis with neural network interface cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 513–533.

[39] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: a data plane architecture for per-packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1099–1114.

[40] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[41] Ruming Tang, Zheng Yang, Zeyan Li, Weibin Meng, Haixin Wang, Qi Li, Yongqian Sun, Dan Pei, Tao Wei, Yanfei Xu, et al. 2020. Zerowall: Detecting zero-day web attacks through encoder-decoder recurrent neural networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2479–2488.

[42] Laurens Van Der Maaten, Eric O Postma, H Jaap Van Den Herik, et al. 2009. Dimensionality reduction: A comparative review. *Journal of Machine Learning Research* 10, 66-71 (2009), 13.

[43] Vladimir Naumovich Vapnik, Vlamimir Vapnik, et al. 1998. Statistical learning theory. (1998).

[44] Ulrike Von Luxburg and Bernhard Schölkopf. 2011. Statistical learning theory: Models, concepts, and results. In *Handbook of the History of Logic*. Vol. 10. Elsevier, 651–706.

[45] Jiazhao Wang, Wenchao Jiang, Ruofeng Liu, Bin Hu, Demin Gao, and Shuai Wang. 2024. {NN-Defined} Modulator: Reconfigurable and Portable Software Modulator on {IoT} Gateways. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 775–789.

[46] Tongze Wang, Xiaohui Xie, Wenduo Wang, Chuyi Wang, Youjian Zhao, and Yong Cui. 2024. Netmamba: Efficient network traffic classification via pre-training unidirectional mamba. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*. IEEE, 1–11.

[47] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. 2017. Malware traffic classification using convolutional neural network for representation learning. In *2017 International conference on information networking (ICOIN)*.

IEEE, 712–717.
[48] Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. 2024. NetLLM: Adapting Large Language Models for Networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 661–678.
[49] Zhaoqi Xiong and Noa Zilberman. 2019. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*. 25–33.
[50] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, et al. 2023. Clickinc: In-network computing as a service in heterogeneous programmable data-center networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 798–815.
[51] Jinzhu Yan, Haotian Xu, Zhuotao Liu, Qi Li, Ke Xu, Mingwei Xu, and Jianping Wu. 2024. {Brain-on-Switch}: Towards Advanced Intelligent Network Data Plane via {NN-Driven} Traffic Analysis at {Line-Speed}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 419–440.
[52] Liyan Yang, Yubo Song, Shang Gao, Aiqun Hu, and Bin Xiao. 2022. Griffin: Real-time network intrusion detection system via ensemble of autoencoder in SDN. *IEEE Transactions on Network and Service Management* 19, 3 (2022), 2269–2281.
[53] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using trio: juniper networks' programmable chipset-for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 633–648.
[54] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. 2022. Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 458–472.
[55] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 179–193.
[56] Jinxiong Zhang. 2021. Yet Another Representation of Binary Decision Trees: A Mathematical Demonstration. *arXiv preprint arXiv:2101.07077* (2021).
[57] Ye Zhang and Byron Wallace. 2015. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820* (2015).
[58] Ruijie Zhao, Mingwei Zhan, Xianwen Deng, Yanhao Wang, Yijun Wang, Guan Gui, and Zhi Xue. 2023. Yet another traffic classifier: A masked autoencoder based traffic transformer with multi-level flow representation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5420–5427.
[59] Yikai Zhao, Wenrui Liu, Fenghao Dong, Tong Yang, Yuanpeng Li, Kaicheng Yang, Zirui Liu, Zhengyi Jia, and Yongqiang Yang. 2023. P4LRU: towards an LRU cache entirely in programmable data plane. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 967–980.
[60] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. IIsy: Practical in-network classification. *arXiv preprint arXiv:2205.08243* (2022).
[61] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. Automating in-network machine learning. *arXiv preprint arXiv:2205.08824* (2022).
[62] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, et al. 2023. {EINNET}: Optimizing tensor programs with {Derivation-Based} transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 739–755.
[63] Guangmeng Zhou, Xiongwen Guo, Zhuotao Liu, Tong Li, Qi Li, and Ke Xu. 2024. TrafficFormer: An Efficient Pre-trained Model for Traffic Data. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 102–102.
[64] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. 2023. An efficient design of intelligent network data plane. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6203–6220.
[65] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).
[66] Shitong Zhu, Shasha Li, Zhongjie Wang, Xun Chen, Zhiyun Qian, Srikanth V Krishnamurthy, Kevin S Chan, and Ananthram Swami. 2020. You do (not) belong here: detecting DPI evasion attacks with context learning. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 183–197.
[67] Eric R Ziegel. 2003. The elements of statistical learning.

Appendix

Appendices are supporting material that has not been peer-reviewed.

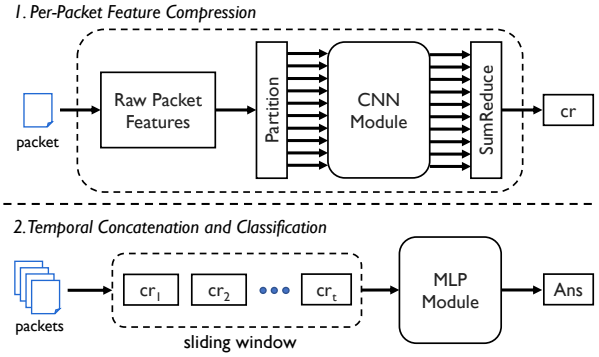


Figure 11: CNN-L model architecture.

A Model Realization

In this segment, we supplement the implementation details of the CNN-L model, which is the most complex and accurate classification model we have deployed on the dataplane.

A.1 CNN-L Model Architecture

Figure 11 illustrates our data-plane-friendly CNN-L model architecture. The overall design consists of two key components: Per-Packet Feature Compression and Temporal Concatenation and Classification, which we describe in turn.

(1) Per-Packet Feature Compression. Ideally, each packet (represented by a 480-bit feature vector) would be directly processed by the CNN Module, which includes multiple inter-segment SumReduce primitives, to generate a compressed representation. However, such designs are inefficient to implement on switch hardware, as the SumReduce primitive significantly increases processing complexity (see §4.3).

To address this, we eliminate all SumReduce primitives inside the CNN Module, following the design principle of Advanced Primitive Fusion. Functionally, this is equivalent to first applying a Partition primitive to divide each packet into multiple feature segments, and then processing each segment independently through the CNN Module to produce a compressed representation of the segment. Finally, a lightweight SumReduce is applied across these outputs to generate the final compressed representation of the packet.

It is worth noting that to better capture dependencies among segments, feature-level shuffling or stacking additional intermediate layers could be introduced in the future. However, in this work, we adopt a simple, lightweight architecture that trades a small amount of accuracy for significantly improved hardware resource efficiency.

(2) Temporal Concatenation and Classification. Finally, we collect the compressed representations of the most recent 8 packets using a sliding window. These vectors are concatenated into a single 16-dimensional input and fed into an MLP Module, which produces the final classification result.

A.2 On-Switch CNN-L Implementation

We implement the CNN-L model prototype on a Barefoot Tofino 2 programmable switch. The processing pipeline is divided into four phases, each corresponding to a key functional phase:

<table>
<tr><th>Stage</th><th>Flow State</th><th>Module</th><th>Primitives</th><th>Stage</th><th>Flow State</th><th>Module</th><th>Primitives</th></tr>
<tr><td>0</td><td>calculate flow hash</td><td rowspan="8">CNN Module</td><td rowspan="7">Map:
get Fuzzy Index;
retrieve Map results;
accumulate during Map</td><td>8</td><td></td><td></td><td>Map: get Fuzzy Index</td></tr>
<tr><td>1</td><td>get flow index</td><td>9</td><td>Fuzzy Index formatting</td><td></td><td></td></tr>
<tr><td>2</td><td>get packet count, last timestamp</td><td>10</td><td>cr window update</td><td></td><td></td></tr>
<tr><td>3</td><td>caculate IPD, window position</td><td>11</td><td>cr window dispatch</td><td></td><td></td></tr>
<tr><td>4</td><td></td><td>12</td><td></td><td rowspan="3">MLP Module</td><td>Map</td></tr>
<tr><td>5</td><td></td><td>13</td><td></td><td rowspan="2">SumReduce</td></tr>
<tr><td>6</td><td></td><td>14</td><td></td></tr>
<tr><td>7</td><td></td><td>SumReduce</td><td>…</td><td></td><td></td></tr>
</table>

Figure 12: The breakdown of CNN-L implementation on a Barefoot Tofino 2 programmable switch.

(1) Flow State Acquisition. In stage 0-3, the switch computes and retrieves necessary per-flow metadata, including the flow index, packet count, and the timestamp of the last packet. Based on this information, we calculate the inter-packet delay (IPD) and determine the correct position in the sliding window where the current compressed representation (cr) should be stored.

(2) CNN Module. Each 60-dimensional input feature vector is divided into 10 segments using the Partition primitive, with each segment containing 6 dimensions. For each segment, a Fuzzy Index is extracted from every pair of features; three such indices are then combined to form a 12-bit key, which is used to retrieve Map results from on-chip SRAM. To reduce stage consumption, we perform accumulation during the Map phase by executing the addition in the Action block immediately after the table lookup. This avoids the need for a separate SumReduce primitive, as each stage supports only one addition per element.

Notably, the Map process can begin at stage 0, as most of the raw input features reside in the packet payload and do not depend on previously fetched flow state. Finally, we perform a lightweight SumReduce in a single stage to complete the CNN Module, since most of the aggregation has already been distributed across the Map stages.

(3) CR Concatenation and Window Management. To minimize register usage, we store the Fuzzy Index—serving as a logical equivalent of the cr—instead of the full compressed vector. In CNN-L, each Fuzzy Index is 4 bits and must be formatted before storage to ensure proper placement in the upper or lower half of an 8-bit register. The formatted index is then written into the cr window. We retrieve all stored cr, reorder them temporally, and feed them into the MLP module.

(4) MLP Module. The MLP module receives the 8 crs from the window, partitions them into four pairs, and applies the Map primitive to each. A two-level SumReduce (i.e., $\log_2 4$ stages) is then used to aggregate the results and produce the final classification output.

Overall, the CNN-L model requires only 15 pipeline stages, demonstrating the efficiency of Pegasus. On Barefoot Tofino 1 and Tofino 2, this leaves 8 and 25 stages respectively for downstream packet processing based on model outputs (considering both ingress and egress pipelines).

Noted that, in our hardware resource utilization report (see Table 6), we only include the cost associated with the operations listed under the Primitives column. This is because the resource consumption of flow state depends on the number of concurrent flows, which is analyzed separately in our scalability evaluation (§7.3).