POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Deploying Run-Time Adaptive Binarized Neural Network in Programmable Data Planes

Supervisors:
Alessio Sacco
Guido Marchetto
Flavio Esposito

Candidate:
Simone Geraci

Academic Year 2024-2025

Acknowledgements

I would like to express my deepest gratitude to all my family for their unwavering love, support, and encouragement throughout this journey. Your constant support, especially during difficult times, have been truly meaningful.

A special thanks goes to my Mum and Dad, whose have been with me since the beginning. You have always believed in me, and thanks to you, I have become the person I am. My love and gratitude for you come from the depths of my heart and it won't never vanish.

My heartfelt thanks go to my beloved Carola, whose patience, understanding, and constant motivation have been invaluable. You taught me what real love is and I will always be grateful to you for that.

Finally, I wish to thank Lorenzo, Giorgio, Giacomo, Mattia, Riccardo and all my dear friends — the old and new ones — for their companionship and support, which made this experience both enriching and memorable.

This thesis represents a part of myself, my story, and it closes the most important chapter of my life so far. I'm excited to see which doors life will open for me next, but above all, I'm grateful to share this journey with my family, the love of my life, and the friends who have profoundly shaped who I am.

Thank you.

Abstract

Switches and other network devices process data at wire rate, meaning they can handle packets at the maximum capacity of the data-link connection. Modern switches separate functionality into two layers: the control plane for slower, high-level decisions (e.g., forwarding tables) and the data plane, which is the hardware-accelerated path through which each packet is actually processed (e.g., port forwarding).

In recent years, with the rise of Programmable Data Planes (PDPs), a major research trend has explored how deep neural network (DNN) models can be leveraged to address long-standing network challenges (e.g., flow classification, anomaly detection) by deploying deep learning models within PDPs. However, deploying DNNs directly on PDPs is challenging due to limited memory and computational resources, the lack of support for neural network–oriented operations, and the need to maintain line-rate packet processing speed.

Our work introduces an innovative split-inference architecture that addresses key challenges found in existing in-network deep learning approaches. We focus on the anomaly detection use case, where the objective is to classify network flows as *benign* or *malicious* using flow statistics as features. We propose an inference framework that integrates two different DNNs: a Binarized Neural Network (BNN) deployed entirely in the data plane, and a more complex high-precision model operating in the control plane. The two models are linked through a fused training strategy based on Knowledge Distillation (KD). The quantized model is trained using both the ground truth and the full-precision model's predictions. In this way, we "guide" the binarized model to mimic the behavior of a deeper, denser network. Then, during the inference phase we selected critical samples based on an in-network confidence score and the most relevant flow features according to recent traffic; both of them fed an adaptive learning mechanism that continuously refines the in-switch model from the control plane. Our solution adapts dynamically to evolving conditions. This prevents accuracy degradation and facilitates long-term performance improvements in dynamic environments.

This thesis will provide comprehensive documentation of the key aspects of in-network machine learning, detailing all the implementation, architectural decisions, and obtained results.

We anticipate that the overall classification performance gap between BNNs and DNNs was not greater than a few percentage points in favor of the latter; however, BNNs outperformed DNNs in terms of CPU efficiency and memory consumption. The programmable switch architecture we targeted was Intel's Tofino ASIC, one of the fastest switches on the market. Achieving a complete forward pass in a

single packet traversal was not feasible; to address this limitation, we exploited the recirculation and mirroring mechanisms provided by the device. Our evaluations indicated that, in some cases, combining knowledge distillation with quantization-aware training led to faster convergence and improved accuracy. Under realistic and dynamic traffic conditions, our system demonstrated strong adaptability to distribution shifts, owing to the implemented refinement mechanism.

Starting from these results, the work can be extended in several directions. For example, expanding the bit width of the quantized neural network can effectively increase its performance.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

BNN Binarized Neural Network

NN Neural Network

FP Forward Pass

BP Backward Propagation

DNN Deep Neural Network

MLP Multi-Layer Perceptron

IDS Intrusion Detection System

ASIC Application-Specific Integrated Circuit

PISA Protocol Independent Switch Architecture

SHAP SHapley Additive exPlanations

QAT Quantization-Aware Training

TCAM Ternary Content-Addressable Memory

STE Straight Through Estimator

TNA Tofino Native Architecture

PHV Packet Header Vector

MAU Match-Action Unit

RMT Programmable Switch Architecure

GPU Graphical Processing Unit

TPU Tensor Processing Unit

IAT Inter Arrival Time

SVM Support Vector Machine

CNN Convolutional Neural Network

TTL Time To Live

SWAR SIMD Within a Register

SIMD Single Input Multiple Data

ALU Arithmetic Logic Unit

XAI EXplainable Artificial Intelligence

Chapter 1

Introduction

1.1 Motivation

Over the past decade, network infrastructures have undergone an unprecedented transformation in both scale and complexity. The ever-increasing demand for bandwidth, combined with the growth of connected devices and services, has led to a significant rise in network traffic diversity and volume. This evolution has also expanded the attack surface of modern networks, making security and adaptability fundamental requirements rather than optional features. Within this context, **Intrusion Detection Systems (IDS)** play a central role in safeguarding network infrastructures by identifying and mitigating abnormal or malicious traffic patterns before they can cause harm.

Traditional IDS solutions are typically deployed on external servers that receive and analyze mirrored traffic from switches and routers. Although this centralized design offers flexibility and computational power, it requires constant communication between the data plane and the IDS server, which introduces additional latency and may cause network congestion under heavy loads. As network speeds increase toward terabit-per-second levels, such centralized architectures struggle to maintain real-time performance, and their scalability becomes increasingly limited.

Recent advances in **programmable networking hardware**, such as the **Tofino ASIC** and the **P4 programming language**, have made it possible to embed custom logic directly into the data plane. These devices can execute user-defined operations at line rate, allowing packet inspection and even lightweight learning tasks to occur directly within the switch. This paradigm—known as **in-network computing**—brings computation closer to where data is generated, enabling faster and more responsive network functions. However, despite their impressive capabilities, programmable switches are constrained by hardware restrictions: operations like multiplication, division, loops, and floating-point arithmetic

are not supported, and available memory is limited to small, specialized hardware blocks.

These limitations make it difficult to deploy complex deep learning models directly in the data plane. As a result, existing intelligent systems continue to rely heavily on external control or cloud-based inference, which reintroduces latency and dependence on centralized resources. What we currently possess are fast, programmable networks capable of limited operations; what we aim for are **intelligent, self-adaptive networks** capable of learning and acting locally. Bridging this gap is the core motivation of this thesis.

To address this challenge, we explore how lightweight, quantized neural models can be integrated directly into the programmable data plane. The main objective is to achieve real-time classification of network traffic at line rate while retaining the ability to adapt to evolving patterns and threats. Achieving this balance between speed, adaptability, and resource efficiency is at the heart of this research.

1.2 Objective

The primary goal of this thesis is to design, implement, and evaluate a novel **split-inference framework** for adaptive intrusion detection within a programmable switch. The system distributes intelligence between the data plane and the control plane, combining high-speed execution with continuous learning. In the proposed approach, a **Binarized Neural Network (BNN)** is deployed within the **Tofino ASIC** data plane to perform classification at line rate using binary operations such as **XNOR** and **population count**. The control plane, on the other hand, operates as a supervisory entity responsible for training, retraining, and maintaining synchronization between model updates and hardware deployment.

To achieve these objectives, the work integrates several key elements:

- A P4-based feature extraction module that computes and encodes flow-level statistics within the switch;
- A hardware-compatible BNN inference engine designed to operate under Tofino's architectural constraints;
- A control-plane refinement loop based on a **teacher-student paradigm**, enabling periodic retraining and adaptation;
- A **confidence-based feedback mechanism** that quantifies prediction reliability and triggers retraining when needed.

To reduce model complexity and improve generalization, feature selection is performed using **SHapley Additive exPlanations (SHAP)**, which identifies

the most influential features contributing to accurate classification. In addition, **Quantization-Aware Training (QAT)** is employed to mitigate the impact of low-precision operations and ensure robust learning despite binarization.

The framework was evaluated on two well-known benchmark datasets for intrusion detection: **CICIDS2017** and **CIC-UNSW-NB15**. Experimental results demonstrate that the proposed architecture achieves strong predictive performance even under extreme quantization. The `Wide` BNN configuration achieved an F1-score of 0.952 on CICIDS2017, while the `Dense` model reached 0.943 on CIC-UNSW-NB15. Open source Tofino SDE further confirmed that the most complex model required less than 8% of total SRAM validating the feasibility of in-network inference on constrained devices.

Beyond these quantitative results, the work highlights a broader implication: the possibility of merging data-plane speed with adaptive intelligence. By embedding learning mechanisms directly within network hardware, we move closer to a new class of **self-optimizing networks** capable of reacting to changes in real time without centralized coordination. This paradigm has the potential to transform not only network security but also other domains, such as traffic engineering, quality-of-service management, and distributed telemetry.

In conclusion, this thesis provides both a conceptual and practical foundation for embedding neural inference within programmable switches. It demonstrates that deep learning principles can be adapted to operate under stringent hardware constraints without losing predictive power, paving the way for intelligent, autonomous network infrastructures.

Chapter 2

Background

Many concepts and entities play important roles in the development of this work, in the following chapter I highlights them. Having a prior knowledge on the following topics will ease the understanding of HYNN. Machine learning and advanced networking are the main areas of interest that are collaborating in the project development.

2.1 Binarized Neural Network

Deep Learning(DL) is an active branch of Machine Learning, precisely we will consider Neural Networks(NN) in the classification use case. The objective of a NN is to create a non-linear function representing the features space, that is used to yield a label, it is the class in which the sample is belonging to. The resulting function (i.e. model) must capture hidden patterns behind data, thus producing a generic decision making module which can correctly classify unseen data. In this way, the application can dynamically adopt different behaviors, based on the NN output, without using static concepts or logic. The process of creating the sudden function is called **training** phase. After that, the process of predicting the belonging class for unseen data is called **inference**.

DL is known to be resource demanding, especially during training; First, it requires a lot of fast-access memory to store and operate Neural Network weights. In the other hand, to achieve a reasonable speed and execution time, the device on which the NN is running needs some type of parallel-based processor to achieve full-layer execution in a single time-unit. Most DL models are operating `float` data type to achieve more precise model update during backpropagation; this guarantees a better final model.

Even if advanced AI application requires many computational resources, embedded systems or ASICs, whose are natively hardware constrained, could not remain behind in this field. In 2016, Courbariaux et al. published [2], the pioneering work about **Binarized Neural Networks(BNN)**. They introduce an extremely quantized NN using weights and activation constrained between $+1$ and -1 , hence 1-bit values.

The advantages introduced by this innovative work are two-fold. First, expensive FP operations are replaced by faster and more efficient bitwise operation. Second, weights dimension is reduced by a numbers of magnitude order. However, quantizing ML models can introduce counterbacks as well, model precision can be substantially lower than the FP counterparts, for instance.

We will discuss the disadvantages in the following subsections.

2.1.1 Forward Propagation

Forward propagation is the process of moving intermediate results from the input/first layer, through all the hidden units, arriving to the output/last layer. It is basically the process of gradually producing the result starting from the raw features and this step is shared between training and inference. The forwarding pass, in our context, is characterized by a binarization step added to the standard matrix multiplication. If the forward pass in a full-fledge NN is expressed:

$$Y^{(l)} = \sigma\left(W^{(l)} X^{(l-1)}\right)$$

the forward pass for a BNN is:

$$Y^{(l)} = \text{SIGN}\left(\text{popcount}\left(W^{(l)} \oplus x^{(l-1)}\right)\right)$$

Binarization functions

This is the binarization function suggested by [2]:

The previous function is the first and most used version of a binarization function, in the past years many variation of it were presented, each one with pros and cons.

$$\text{Sign}(x) = \begin{cases} +1, & \text{if } x \geq 0, \\ -1, & \text{otherwise.} \end{cases}$$

Between all of its variations, the binarization function proposed by XNOR-Net [3] has been appreciated by researchers and has the following formalization:

$$I \approx \alpha * \text{sign}(I) = \alpha * B_I$$
$$W \approx \beta * \text{sign}(W) = \beta * B_W$$

Where α and β are defined as follow:

$$\alpha = \frac{1}{n}\|I\|_{L_1}$$
$$\beta = \frac{1}{n}\|W\|_{L_1}$$

Rastegari et al., to reduce the quantization loss during the 32bit conversion to 1b added two scale factors α and β for weights and activations to the classical Sign function. We will discuss deeper the role of this function in the section

2.1.2 Backward Propagation

The backpropagation (BP) step is the fundamental process of learning of every NN. It is an iterative process that adjusts the weight parameters according to some multi-dimensional directions. A variant of the know Gradient Decent is used normally to train DNN, the Stochastic Gradient Descent, where the gradient of the loss function, with respect of the parameters is computed over a small batch, instead of all the samples; that for a computational resource motivation. Since the derivative of the loss must be computed, it has to be differentiable. However, in BNNs, propagating the error backward is not that simple; the *sign* function is not differentiable and in some parts of it, the gradient vanishes to 0.

2.1.3 Gradient Approximation

Nowadays, in order to successfully train a BNN, the *sign* function is applied to activations during forward propagation, but then, when the backpropagation needs to be performed, the Straight Through Estimator [1] is operated. Bengio et al. proposed it in 2012, and Courbariaux et al. were the first to integrate it in a BNN training context [2]. The function of STE is defined as follows:

$$\text{clip}(x, -1, 1) = \max(-1, \min(1, x)). \tag{2.1}$$

This estimator solves the derivative problem of the *sign* binarization function during BNN training, but another issue was noticed about this solution. Since the clipping operation makes the gradient vanishes for absolute input values greater than 1, researcher noticed how bad it affected the training, making it more instable and prone to stall in terms of accuract and learning. In practical scenarios, the Identity function is indeed, preferred; propagating gradients also in cases where absolute value of the gradients were outside the the range $[-1, +1]$.

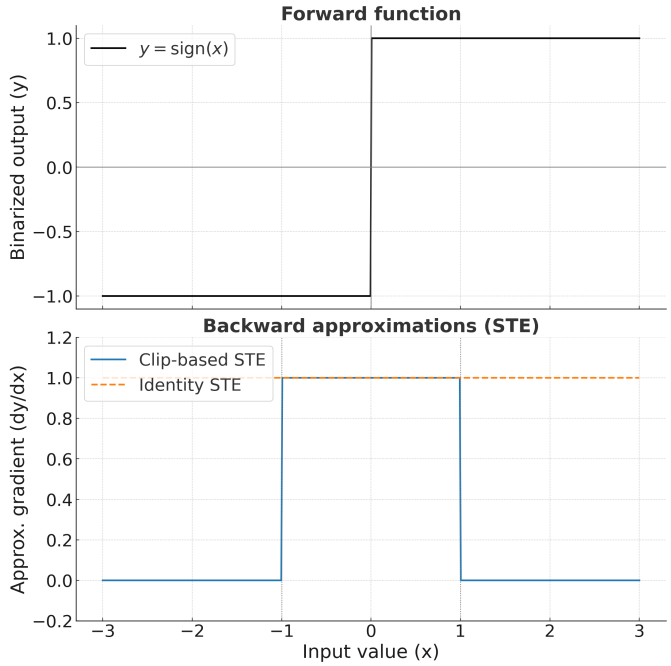


Figure 2.1. This plot shows the difference between the clipped and identity version of the Straight Through Estimator during backpropagation. The clipped version makes the gradients vanishes in ranges outside $[-1, +1]$, while the identity STE avoids these "dead zones", actually improving and stabilizing BNNs training.

2.2 Programmable Data Planes

The emergence of programmable data planes has radically changed the way researchers and practitioners conceive of computer networks. For decades, switching hardware was built around fixed-function ASICs that supported only a narrow set of standardized protocols. While efficient, this approach was fundamentally rigid, preventing operators from deploying new features or adapting the network to novel use cases without long hardware development cycles. Programmable Data Planes, in contrast, expose an interface where the behavior of the forwarding hardware can be defined through a high-level programming language, enabling networks to evolve at the speed of software while preserving line-rate performance.

2.2.1 Protocol-Independent Switch Architecture

The Protocol-Independent Switch Architecture (PISA) has become the canonical abstraction for programmable switches. In PISA, the data plane is structured

as a sequence of stages that can be reconfigured by the programmer. Incoming packets first traverse a parser, which extracts header fields based on a grammar defined in software rather than by fixed hardware logic. After parsing, packets enter a pipeline of match-action stages. Each stage compares selected header fields against tables that may be populated at runtime and, depending on the match, executes an associated action such as modifying headers, updating metadata, or changing forwarding decisions. Finally, the packet is reassembled by the deparser and transmitted on the output port.

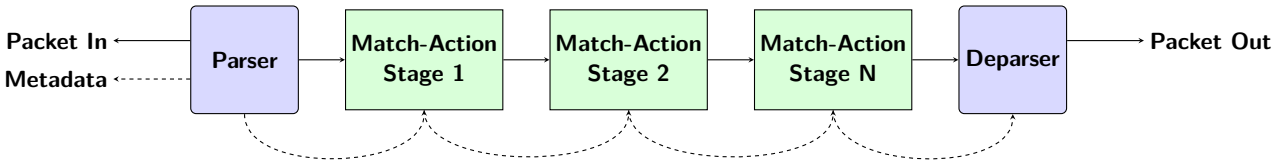


Figure 2.2. Protocol-Independent Switch Architecture (PISA). Packets traverse a programmable parser, a sequence of match-action stages, and a deparser, while metadata flows along the pipeline.

This design is significant because it decouples the forwarding pipeline from specific protocols. Instead of being bound to Ethernet, IPv4, or TCP, the switch can be programmed to recognize arbitrary headers and behaviors. This protocol independence is what makes PISA versatile for research in new architectures, in-network computing, and advanced telemetry.

2.2.2 Data Plane Programming Language

To realize the potential of PISA, a dedicated programming language was needed. The result is P4, the Programming Protocol-Independent Packet Processors language. P4 allows developers to define custom headers, write parsers that extract those headers from raw packets, declare match-action tables, and specify the control flow that determines how a packet progresses through the pipeline.

The first version, P4_14, provided proof of concept but was closely tied to particular implementations. Its successor, P4_16, refined the model by emphasizing modularity and target-independence. In P4_16, the architecture is explicitly defined, so a single program can be compiled against different back ends such as the v1model for software switches, the PSA (Portable Switch Architecture), or TNA (Tofino Native Architecture). This separation of language and architecture enabled both academic and industrial adoption, making P4 the de facto standard for data plane programming.

```
parser start {
    extract(ethernet);
    return ingress;
}

control ingress {
    table forward {
        reads { ethernet.dstAddr : exact; }
        actions { set_port; drop; }
        size : 256;
    }

    action set_port(bit<9> port) { modify_field(metadata.egress_spec, port); }

    apply { forward.apply(); }
}
```

Listing 2.1. Concise BMv2 P4 snippet

In order to guarantee a certain speed standard, the P4 language has a very limited instruction set, both in quantity and functional sense. No support for loop-like construct and high-precision data type. In addition, there are no native implementation for multiplication or division; it is worth to mention that a barrel shifter is present and it allows to approximate the previous two computations.

2.2.3 Targets

A P4 program is not executed in the abstract: it must be compiled against a concrete target. Different targets expose different capabilities, reflecting trade-offs between speed, programmability, and deployment feasibility.

One of the most common research targets is BMv2, the behavioral model software switch. BMv2 implements the v1model architecture, a simplified representation of PISA suitable for prototyping. It runs on commodity CPUs, allowing developers to test P4 programs functionally even though it lacks hardware-level performance.

On the opposite end of the spectrum is Intel's Tofino ASIC, which implements the Tofino Native Architecture (TNA). Tofino is a fully programmable, production-grade switch capable of terabit throughput at line rate. The TNA architecture reflects the real hardware pipeline and provides developers with fine-grained control while enforcing constraints that ensure deterministic timing. Thanks to the high-performance hardware installed in this ASIC, Tofino makes a really good option to offload BNN inference, this is why we chose this target as foundation for our system.

Between these extremes are SmartNICs, which embed limited P4 programmability into network interface cards. While not as powerful as dedicated ASICs, SmartNICs allow certain packet processing tasks to be offloaded from the host CPU, making them attractive in cloud and edge environments.

2.2.4 Control Plane

The data plane alone cannot operate without a control plane. Whereas the data plane processes packets at line rate according to the logic defined in P4, the control plane is responsible for populating tables, updating rules, and reacting to network dynamics. In the P4 ecosystem, the main interface between these two components is P4Runtime, a standardized API that allows an external controller to install, modify, and remove entries from match-action tables. This separation preserves the performance of the data plane while ensuring flexibility: the same P4 program can be reused across different environments, while the control plane adapts it to the needs of specific applications.

2.3 Intel Tofino ASIC

Among all existing targets, Intel Barefoot's Tofino stands out as the most impactful. Tofino ASICs implement the PISA model while delivering multi-terabit performance. The pipeline consists of a programmable parser, a large sequence of match-action stages, and a deparser, each optimized to sustain line-rate throughput regardless of traffic patterns. Unlike software switches, Tofino's constraints are dictated by hardware realities: the number of tables, size of memories, and ordering of operations are limited. During this ASIC the development a huge effort was put in place to enhance the parallelization aspect of non-dependent operation; tables lookup, action execution, if not dependent on each other can be executed in the same stage without any further delay.

Packet Header Vector

In a Tofino switch, each packet's parsed header fields are collected into a fixed-size **Packet Header Vector (PHV)**, which serves as the operand space for match–action tables and ALU units. The PHV allocator is a compile-time tool that maps individual header fields (and metadata) into contiguous slices within the PHV, respecting hardware constraints on alignment, width, and access granularity. It groups fields that are often used together into the same 32B or 64B PHV blocks to minimize cross-block accesses and ensure that each stage's match and action pipelines can read/write their required slices in a single cycle. When a user-defined P4 program declares headers, metadata, and tables, the allocator analyzes all read/write patterns, co-locates dependent fields, and flags conflicts (e.g., rotated writes, overlapping slices) so that the final PHV layout maximizes throughput and avoids pipeline stalls.

Match-Action Unit

The packet is then processed by a **Match-Action Unit (MAU) pipeline**, composed of multiple **MAU stages**. Each MAU stage performs match-action operations where packet fields are matched against lookup tables, and corresponding actions are applied. The sequential nature of MAU stages enables complex packet processing logic to be built incrementally, while maintaining high throughput and low latency.

Feature	**Constraint**
MAU stages	12
SRAM	120 Mbit
TCAM	6.2 Mbit
PHV	4 Kb total size limit Limited fix-sized containers
Registers	Read or update once per packet RegisterAction must fit in one stage
Action	Simple comparisons allowed Cannot span over multiple stages
Table	Up to 1 sequential lookup per stage Up to 16 parallel lookup per stage One possible look-up per table

Table 2.1. The above table lists the main constraints we have complied during the development of the system. Each constraint is referred to a single Tofino pipeline.

With the now open-source open-p4studio, Barefoot/Intel enabled researchers to evaluate their ideas on production-grade SDE, when code structure allows it. This accessibility, coupled with the BFRuntime Interface for control-plane interaction, created a vibrant ecosystem of tools and projects that rely on programmable switches.

Chapter 3

Related Work

During the development of my work, I encountered numerous articles and previous projects from which I draw inspiration; in this section, I will highlight the core aspects about the approach they have chosen.

Since 2016, researchers have dreamed of a smart networking system, and to fulfill this goal, they have tried to place ML models inside the network. Thanks to evolving capabilities of network devices and programmability, trying to AI-enable them has never been more feasible.

Researchers have successfully integrated different ML models despite the computational capabilities of these devices. We can separate them into three main categories: **Decision Tree Ensemble Models** (DT), **Neural Network**(NN), and other models.

3.1 Neural Network-based works

Network devices hardly support the sophisticated and power-consuming operations needed to execute a full-fledged neural network; this is why the vast majority of ML inference implementations on the data plane are characterized by the presence of some compromises.

Binarized models

N2Net [14] extremely quantizes a NN to the point that a single bit is used to represent the weight of each neuron (e.g BNN). It combines a series of bitwise operations to compute the result of the layer. It does not present a complete implementation, nor the underlying architecture, and we only know that it is an RMT-like switch pipeline. I personally define it as the pioneer article of this research trend; the same approach and heuristics will drive many future works.

However, it does not discuss the performance aspect of the NN forward pass, nor the possible limitations of its approach in relation to the size of the model.

BaNaNa SPLIT [12] focused its analysis on the assumption that SmartNICs and other network-programmable devices can be successfully used as AI-accelerator components. The study highlights the limitations of CPU-based machines in the NN inference context. It shows that CPUs are efficient executors for `conv`, `pool`, and `norm` layers, but far less efficient when `fc` layers must be executed; when these types of blocks are processed, the degree of multi-programmability decreases and the system wastes many clock cycles. Its intuition came from this drawback; in fact, it offloads the `fc` computation to the SmartNIC. Before NN execution, BaNaNa SPLIT takes the full-fledged NN result before the `fc` layers and subjects it to a binarization stage. Then, the same methodology as [14] is followed to achieve the final result. However, the binarization process and communication between NIC and CPU noticeably increase the system's latency; as with [14], it does not provide precise benchmarks or evaluation sections.

Network devices rely on a very heterogeneous set of underlying architectures, each suitable for different tasks, which is why choosing the correct target for the selected use case is fundamental. N3IC [15] is one of the first papers to deploy a working BNN executor on different architectures. The authors demonstrate the feasibility of their idea on BMv2 (Software Switch), a very flexible and simplified version of a programmable switch. They decided to use a BNN to perform anomaly detection and traffic identification. They propose different BNN configurations, and with all of them they achieved a complete NN forward pass in a single packet processing. After that, they created a standalone compiler able to translate the definition of a BNN network into implementations that can be directly deployed on data-plane SmartNICs. They managed to support different targets using both micro-C and P4 languages.

Quantized models

The next section of models that will be discussed are the quantized models. In a standard scenario, where GPUs and TPUs can be used to infer NN results, a common weight size is 32b, even 64b. This generous dimension guarantees the maximum precision when gradient comes and small adjustment to weights are performed. However, many studies have shown that reduced weight dimension to a certain point decreases the overall NN accuracy of a small fraction; this is the assumption on which the following works are based.

INQ-MLT [18] quantizes weights and activations to a standard integer bit-width, natively supported by a variety of targets such as v1model, psa and tna. In this work, Kaiyi Zhang et al. proposed a novel approach named Quantization

Aware Training [10] where quantization nodes are added to the forward pass, simulating the quantization loss; in such a way the network is aware of the quantization process and can adjust its parameters, reducing the derived loss. In the proposed toolbox, they show two ML models, CNN and MLP, showing their superiority to BNN models mentioned in the previous subsection. In this proof of concept, they demonstrate that quantized ML models are worth to be explored in data plane configuration; however, their implementation remains too complex for the majority of hardware target, they proposed the toolbox for BMv2/v1model indeed. In fact, INQ-MLT explicitly mentions that a prior multiplication extern (support) is necessary of the deployment of the described toolbox; and such a support is all but common in data plane.

Following the footsteps of Kaiyi Zhang et al., Quark [19] proposed a framework for quantized CNN execution. In their analysis, they demonstrated that weights and activations with 7-bit width were optimal for storage and accuracy efficiency in the flow anomaly detection use case. Mai Zhang et al. adopted the Tofino/tna model to empower the entire project; they chose a hardware target, which allowed for a more precise and rigorous evaluation phase in the paper. Given the complexity of the ML model, a forward pass could not be achieved in a single switch traversal of a packet; more than 100 packet recirculations were needed to predict a unique flow classification. In addition, Quark supports domain shifting: changes in traffic behavior are addressed by constant feature tracking and retraining of the model.

In addition, some projects explored the distributed execution of more complex ML models. NetNN [11] implemented a complete data plane-based intrusion detection system. By mapping different parts of a DNN into a series of programmable switches, Kamran Razavi et al. showed the execution of full-fledged DNN architectures at line rate. They considered the entire raw packet as input features, avoiding complicated feature engineering on the data plane, which allowed them to use all the available resources for the DNN executor. Moreover, an enhanced version of the model was analyzed, taking into account the inter-arrival time (IAT) between packets belonging to the same flow as an additional feature. Their proposed work offered direct support for `conv` and `fc` layers. Finally, their work showed that DNN inference systems on a distributed data plane can achieve state-of-the-art performance while meeting real-time requirements.

Modern datacenter networks are often partitioned into ultra-fast data plane modules and control planes, where the latter accomplish complex data-driven management policies to update the former's parameters. Taurus [16] addressed the unavoidable delay between data and control plane communication by implementing an external FPGA-based module as an AI accelerator. The dedicated chip supports the MapReduce paradigm, often used in ML for vector-to-vector and

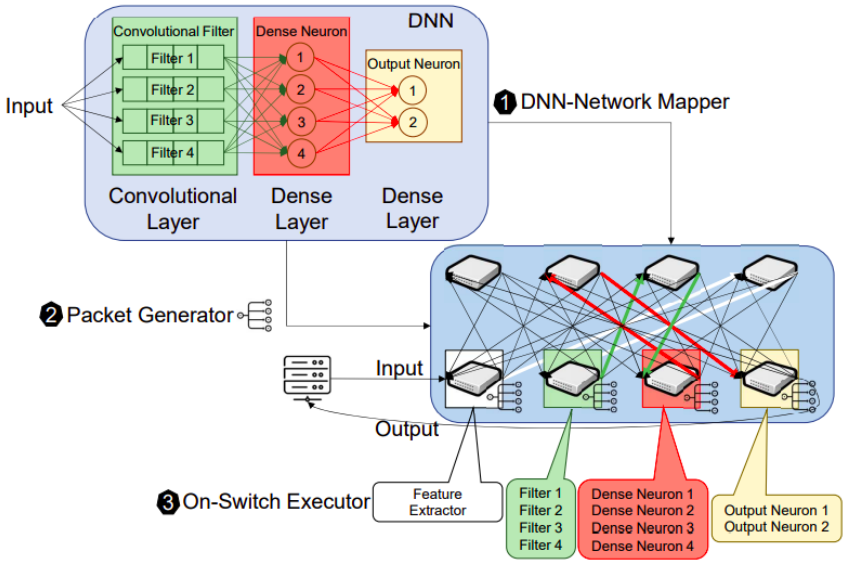


Figure 3.1. This NetNN overview shows clearly how distributed computation can map seamlessly to NN. The picture is taken from [11].

vector-to-scalar operations like dot products for NN neurons. Their implementation added a dedicated control block into P4, named MapReduce. By offloading the heaviest ML operations from the switch and exploiting the dedicated Taurus hardware, the latency with the control plane can be avoided while introducing only minimal delay compared to a full data plane approach.

3.2 Non Neural Network-based works

Since 2019, many works have deployed Decision Trees (DT) and tree based Ensemble models following two main approaches: encode-based and depth-based. As showed in 3.2, the encode-based uses a varying number of MAT depending on how many levels the tree has. A single MAT table is used for each level of the tree. The latter encodes each feature and uses an additional code-label table for the DT model; this is the case of IIsy [17] and the Planter [20].

IIsy is a framework for mapping traditional machine learning classifiers into programmable switches. The authors prototyped four algorithms on match-action pipelines: **decision trees**, where features are sequentially matched and encoded into metadata, reducing depth compared to naive stage-per-branch designs; **SVMs**, either by assigning votes per hyperplane through multiple tables or by aggregating feature-based vectors to approximate classification boundaries; **Naive Bayes**,

which replaces probability multiplications with lookup-based encodings or wide feature-to-class tables, trading precision for feasibility; and **K-means**, realized by storing per-feature distance vectors and summing them in the final stage to assign clusters. These designs emphasize lookup tables over arithmetic to fit hardware constraints. Implemented in both software (BMv2) and hardware (NetFPGA SUME), IIsy achieves full line rate and practical accuracy on IoT traffic classification, though limited by the number of features and classes that can be supported with available table depth and stage count.

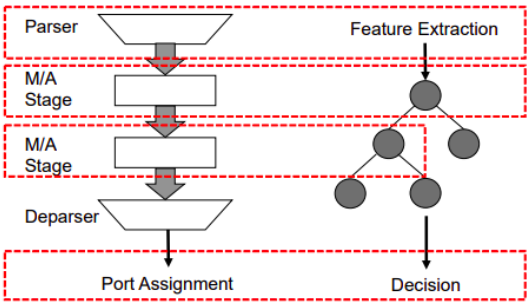


Figure 3.2. This scheme shows the similarity between MAT of a simple switch and a decision tree. The picture is taken from [17].

While IIsy focused only on traditional ML models, in 2024 Planter introduced a novel framework that supports all previously mapped models and extends coverage with adaptations for neural networks. Planter also supports a broad set of targets, including all major programmable architectures currently available on the market.

Together, these two papers highlight a clear trend: recent projects increasingly emphasize complete and extensible frameworks, rather than isolated algorithmic mappings or incremental optimizations.

Chapter 4

System

Modern Intrusion Detection Systems (IDS) for networks are typically based on separate external servers with dedicated computational resources. However, this well-known design requires constant communication between the core of the network and the IDS server, potentially leading to network overload. In addition, the time needed for this communication increases with the amount of traffic in the safeguarded network, causing potential delays in the IDS's ability to take protective measures against threats. Finally, the use of external servers represents a significant cost for large companies, and constant maintenance is required.

In contrast, in-network inference systems offer microsecond-scale decisions capabilities, enabling immediate reaction to events. For example, an in-network DNN could identify a malicious packet or a congestive flow and trigger mitigation on the very next switch hop, something not feasible with cloud or CPU-based analytics that operate at millisecond or higher timescales. This ultra-low latency processing not only improves responsiveness but also reduces network load, as fewer packets need to be mirrored to external analyzers. Our system overcomes the limitations

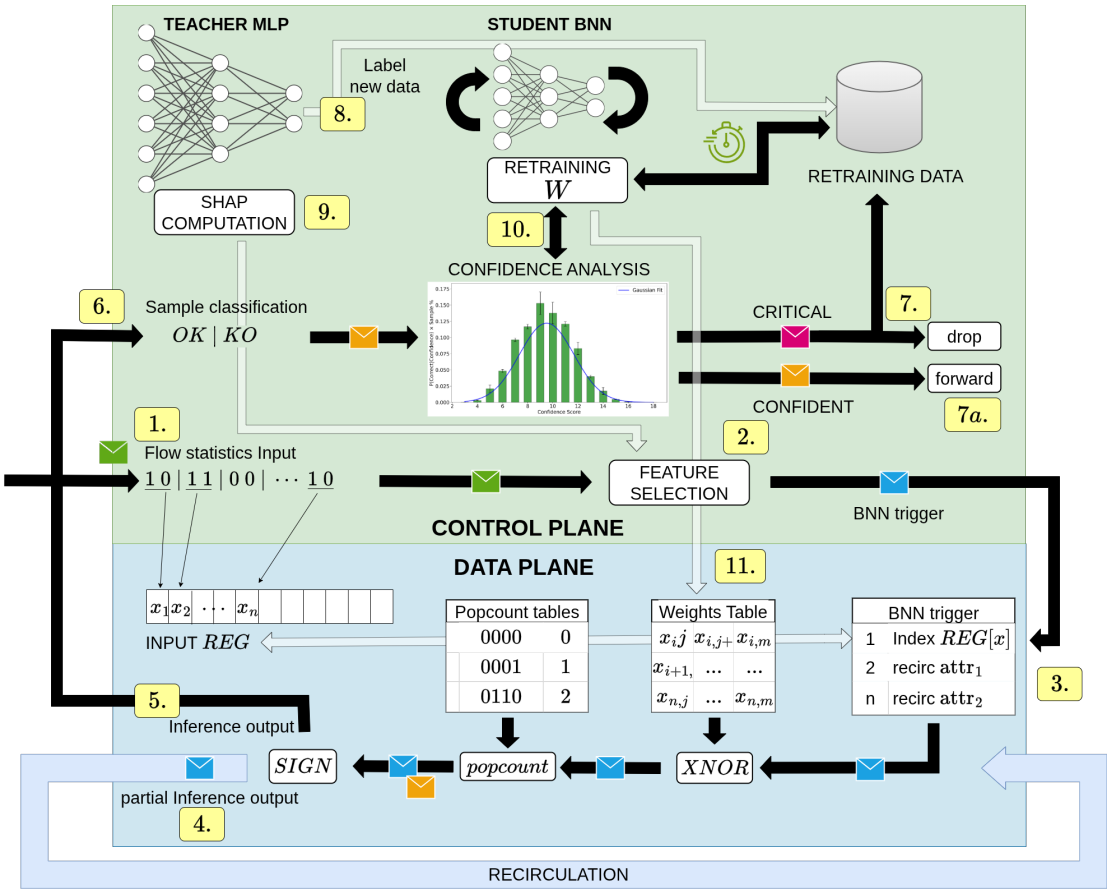
of traditional approaches by deploying a dynamic all-in-one framework entirely within a programmable switch.

4.1 System Overview

The system is composed of two neural network instances: a DP-resident BNN and a CP-resident DNN.
After a fixed number of packets belonging to the same flow traverse the switch, the latter is marked as *mature* and becomes ready for classification. At this point, the BNN in the DP infers its class and sends the predicted label, together with the

confidence score, to the CP. The confidence score is a metric indication representing how sure is the network about the prediction.

Figure 4.1. This figure illustrates the complete workflow of our adaptive intrusion detection framework. Network flow statistics are first extracted in the data plane and passed through the BNN executor, where bitwise XNOR, popcount and *SIGN* perform real-time inference. The classification results and confidence scores are then analyzed in the control plane to determine whether samples are confidently classified or marked as critical for retraining. Critical samples are then labeled by the teacher MLP, whose computes SHAP-based feature importance, and guides the student BNN during retraining, enabling the system to continuously adapt to evolving network behaviors.



When the flow classification is *high-confident*, the corresponding flow is considered handled, with actions dictated by the inference outcome(e.g., drop or quarantine for attacks, forwarding for benign traffic). If the confidence score is below the threshold the control CPU records the *critical* flow as fine-tuning material during

the BNN retraining phase.

For each refinement process, the best $n \in \{133,128,98\}$ bit-features (depending on the deployed BNN variant) are selected and used for retraining.

Thanks to the data plane and control plane co-design we are able to take advantage of both components. Exploiting the control plane CPU, we can tailor extremely quantized Neural Networks and their associated features to suit the anomaly detection use case. Moreover, by using the provided API, we interact with the data plane through a robust and reliable protocol, allowing us to deploy the just-trained model efficiently.

The benefits of leveraging the DP processing pipeline to execute the BNN inference phase are two-fold. First, the ultra-fast per-packet processing time of the switch is highly suitable for offloading the inference phase of our neural network models, basically at wire speed. Second, by analyzing the flows as they naturally traverse the switch en route to their final destination, we are able to gather up-to-date flow statistics necessary both for fast and reliable classification and for the model refinement phase.

We define flow as a group of packets that traverse the network and have the shares fundamental properties as shown in the Table 4.1.

Field	**Description**
Source IP address	The IP address of the device that initiated the communication.
Source port	The port number on the source device used for the communication.
Destination IP address	The IP address of the device intended to receive the communication.
Destination port	The port number on the destination device designated to receive the data.
Protocol	The communication protocol used for the data transfer (e.g., TCP, UDP).

Table 4.1. All the packet traversing the network that shared these attributes are considered a flow

4.1.1 Datasets

In the following two subsection I will presents the key aspects of the datasets used to evaluate our system. Even if multi-class categorization are present for

UNSW-NB15 and CICIDS2017, we focused on binary anomaly detection, taking into account just the label for **bening** and **malicious** flows.

CICIDS2017

The CICIDS2017 dataset [13], produced by the Canadian Institute for Cybersecurity, offers another comprehensive benchmark designed to reflect real-world network traffic. It was captured over five days of operation in a testbed that simulated both legitimate user behavior and diverse attack vectors, including DDoS, brute force, botnets, and infiltration attempts. Each flow is annotated with 80 statistical features covering packet- and flow-level characteristics. CICIDS2017 is particularly valued for its temporal realism and traffic diversity, which allow a robust evaluation of intrusion detection approaches under dynamic and evolving conditions.

UNSW-NB15

The UNSW-NB15 dataset [7–9] is a widely used benchmark for evaluating intrusion detection systems. It was generated at the Australian Centre for Cyber Security using the IXIA PerfectStorm tool, which combines modern normal traffic with synthetic attack scenarios. The dataset contains nine categories of malicious activity, including exploits, DoS, reconnaissance, and backdoors, along with a rich set of flow-based features. Compared to earlier datasets such as KDD'99 or NSL-KDD, UNSW-NB15 provides more realistic traffic patterns and a better balance between benign and malicious samples, making it suitable for training and validating machine learning models under contemporary network conditions.

CIC-UNSW-NB15 [6], an augmented version of the original UNSW-NB15, has been generated analyzing the original pcap files with the same tool used by CICIDS2017 and, in fact, the two datasets share the same features. For UNSW-NB15 We will use the augmented version for our evaluation test in order to ensure consistency between features and features values.

4.1.2 Deep Learning Models

Nowadays, with the arising popularity of ML framework, a variety of DL models are available and easy to deploy; we could opt for some complex DL network architecture to ensure the highest possible performance, however, we were constrained by the control plane resources and by the data plane custom architecture. In addition to that, spending a large amount of time training the network can cause a late adaption to the dynamic network traffic, causing a mismatch between the data distribution where we trained the BNN and the live data we will infer on.

These were the reasons why we chose the well-know Multilayer Perceptron(MLP) for both control plane and data plane resident NNs architecture. It is one of the pioneer NNs used in the classification task, and is widely used today as last module in more complex networks, often named `flatten`. Unlike some other popular architectures, Convolutional Neural Network (CNN) for instance, MLPs lack of capacity to capture locality patterns behind the analyzed data. However, since our training data are in tabular format, this is not threatening our scope.

4.2 Control Plane

The orchestrator of our system is represented by the control CPU, it is in perpetual communication with the data plane; in such manner, it is able to install certain rules and apply specific policies in order to adapt in the best possible way to the actual network scenarios. More specifically, a number of tasks are under its control. First, receiving network statistics, inference results — together with the confidence score —from the data plane pipelines. Second, keep track of critical samples and use them for future model refinement. Third, trigger the refinement phase for the binarized NN. And finally, it processes weights and input flow in order to make them suitable for the data plane deployment.

4.2.1 Model training

In this section, I will explain in detail the NN training methods I adopted and, in addition, I will make a brief introduction about the main concepts and logic behind the ML frameworks I operated.

Quantization Framework — Brevitas

All our quantized models are built with Brevitas [4], a PyTorch-integrated framework for FPGA-oriented NN quantization. Brevitas implements a flexible uniform-affine quantization scheme with trainable scale factors and per-tensor (or per-channel) clipping thresholds, using the Straight-Through Estimator (STE) for backpropagation. Instead of relying solely on SIGN-based binarization (which maps values to ± 1 with no intermediate precision), Brevitas allows arbitrary bit-widths for both weights and activations, supports symmetric or asymmetric ranges, and can optionally learn the quantization step size during training. This richer setup preserves more of the original dynamic range—improving accuracy on more complex tasks—while still yielding hardware-friendly integer representations for deployment.

Brevitas, indeed, allows for generic quantization training approaches, independent from the chosen quantization strength. We focused on BNN training, and to fullfill that objective, we implemented a custom binary quantizer by subclassing Brevitas's Quantizer interface. This BinaryQuantizer forces both weights and activations to $-1, +1$ during the forward pass, while still leveraging the STE for gradient updates.

Quantization Aware Training

Quantization-aware training (QAT) in Brevitas seamlessly integrates quantization operations into the forward pass, so that both weights and activations are "aware"

of their limited precision during training. By replacing standard layers with their quantized counterparts and specifying bit-widths, scale constraints, and quantizer classes, Brevitas simulates the effects of low-precision inference while still performing full-precision gradient updates via the STE. Since quantization is not differentiable, the STE is used to approximate gradients and enable backpropagation. This approach allows the model to adapt its parameters around quantization noise and non-idealities, typically yielding significantly higher accuracy at deployment compared to post-training quantization alone.

```
class CommonBinQuant(ExtendedInjector):
    quant_type = QuantType.BINARY
    bit_width_impl_type = BitWidthImplType.CONST
    scaling_impl_type = ScalingImplType.CONST
    restrict_scaling_type = RestrictValueType.FP
    zero_point_impl = ZeroZeroPoint
    float_to_int_impl_type = FloatToIntImplType.ROUND
    scaling_per_output_channel = False
    narrow_range = True
    signed = True
    bit_width = 1

class CommonBinWeightQuant(CommonBinQuant, WeightQuantSolver):
    scaling_const = 1

class CommonBinActQuant(CommonBinQuant, ActQuantSolver):
    min_val = -1.0
    max_val = 1.0
    scaling_const = 1
```

Figure 4.2. The above code block show how a Brevitas Quantizer is declared. Particularly important attributes are `bit_width`, `min/max_val` and `scaling_cost`. They define respectively, min/max values for weights and activation and the corresponding bit width; the `scaling_cost` refer to the affine quantization that Brevitas supports, however we are not using it.

Confidence Score Implementation in Binary Neural Networks

The confidence score represents a quantitative measure of the certainty showed by a BNN about its classifications. Unlike traditional neural networks that operate with continuous activations, BNNs employ binarized weights $\mathbf{W} \in \{-1, +1\}$

and activations $\mathbf{a} \in \{-1, +1\}$, which fundamentally alters the interpretation of intermediate layer outputs as confidence indicators.

When multi-class classification task is involved, full-fledge NNs are often using a `Softmax` layer — defined as 4.1 — at the end of inference pipeline in order to enable the corresponding output to a probabilistic interpretation.

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{4.1}$$

However — even if we are treating out binary classification task as multi-class classification with only two classes — we could not use the aforementioned layer to compute confidences since binary values $\{0, 1\}$ are not intrinsically suitable for probabilistic interpretation. In our configuration the possibles BNN outputs span over $2^{\text{output\_layer}} = 2^2 = \{00, 01, 10, 11\}$.
Rather than `Softmax`, we exploited the number of active neuron units in the penultimate BNN hidden layer as confidence score. By gathering statistics on neurons activation on the validation set, we can helpful insights about the BNN classification trend.

Mathematical Formulation
In a BNN with architecture $f : R^d \to R^c$ where d represents the input dimensionality and c the number of classes, the confidence score $C(\mathbf{x})$ for input $\mathbf{x}$ is computed as:

$$C(\mathbf{x}) = \sum_{i=1}^{h} \max(0, a_i^{(L-1)}) \tag{4.2}$$

where $a_i^{(L-1)}$ represents the i-th activation of the penultimate hidden layer, and h denotes the number of hidden units in that layer. The $\max(0, a_i^{(L-1)})$ converts the bipolar activations $\{-1, +1\}$ to the corresponding $\{0, 1\}$, counting the number of "active" neurons.

Confidence histogram and Weighted Accuracy
The practical evaluation of confidence scores involves analyzing their distribution across validation samples and computing weighted accuracy metrics. For a given confidence value c, we define the weighted accuracy contribution as:

$$W(c) = A(c) \times P(c) \tag{4.3}$$

where $A(c)$ represents the accuracy of samples with confidence score c, and $P(c)$ denotes the percentage of samples exhibiting that confidence level:

$$A(c) = \frac{\sum_{i:C(\mathbf{x}_i)=c} I[\hat{y}_i = y_i]}{\sum_{i:C(\mathbf{x}_i)=c} 1} \tag{4.4}$$

$$P(c) = \frac{\sum_{i:C(\mathbf{x}_i)=c} 1}{N} \tag{4.5}$$

where $I[\cdot]$ is the indicator function, $\hat{y}_i$ and y_i are the predicted and true labels respectively, and N is the total number of validation samples.

This weighted formulation provides a practical interpretation: samples with higher confidence scores should ideally demonstrate higher accuracy, while the percentage weighting ensures that confidence values affecting more samples contribute proportionally more to the overall model assessment. The confidence histogram visualization plots these weighted values $W(c)$ against confidence scores, revealing the relationship between network certainty and prediction reliability across the sample distribution.

Thanks to this mechanism we are able to understand whether a sample is hard to classify for the current model. During the successive refinement phase, we will select the retraining sample looking at the corresponding confidence score.

4.2.2 Model deployment

Upon training, the model has to be deployed within the data plane environment. We need to extract the binarized weights from the Binarized model, transform weights from the bipolar configuration to the binary one — $\{-1,1\} \rightarrow \{0,1\}$ — and finally, load them withing the Tofino ASIC's dedicated tables.

The controller organizes the upload of weights and inputs into Tofino registers in a way that matches the parallelism and stage constraints of the hardware. Instead of loading full vectors directly, both weights and inputs are divided into smaller fixed-size slices with a suitable size depending on the BNN architecture. These slices are grouped into batches, so that each hardware stage can process a subset of neurons in parallel without exceeding the register width or pipeline capacity 4.3. The weight batches are stored in dedicated tables, indexed by their batch identifiers, ensuring that each group of slices is associated with the correct neurons. Input vectors are split following the same principle and written into registers so that their alignment matches the weight organization. This batching strategy allows the execution pipeline to perform consistent parallel XNOR and popcount operations across neurons while keeping register usage predictable and hardware-friendly.

```
# push weight into weight table l0
for j in range(num_neuron_batches):
    for ix in range(weight_batches_no):
        base = self.parallel_neurons_cap * j

        w1 = w_mx[ix]

        (w0,  w1,  w2,  w3, w4, w5, w6) = w1[base:
        ↪  base+self.parallel_neurons_cap]

        self.bfrt_l0_weights.add_with_get_weights(
            f"{ix}",            # key "weight_batch"
            f"{j}",             # key "neuron_batch"
            f"0b{w0}",          # bit<14> nr1_w
            f"0b{w1}",          # bit<14> nr2_w
            f"0b{w2}",          # bit<14> nr3_w
            f"0b{w3}",          # bit<14> nr4_w
            f"0b{w4}",          # bit<14> nr5_w
            f"0b{w5}",          # bit<14> nr6_w
            f"0b{w6}"           # bit<14> nr7_w
        )
```

Figure 4.3. Upon weights extraction, the binarized parameters are pushed within the BNN pipeline through the `bfrt_l0_weights.add_with_get_weights` API call. They are pushed by batches, a slice of weights is loaded per neurons batch, that will guarantee parallel neuron execution in dataplane.

4.2.3 BNN Weights and Inputs Deployment

The deployment of trained BNN model onto the Tofino ASIC requires a controller mechanism that manages both weight distribution and real-time input processing through the P4 dataplane. Our controller architecture, operates as a bridge between the trained PyTorch-like models and the hardware-accelerated BNN executor running on the Tofino switch.

The deployment process begins loading the final trained model weights inyo the BNN hardware implementation. This is possible by accessing the model's `quant_tensor` object from Brevitas model 4.4, containing already weights in a binarized format, suitable for successive deployment.

```python
def extract_binary_weights(model: torch.nn.Module):
    binary_weights = []
    for name, module in model.named_modules():
        if isinstance(module, qnn.QuantLinear):
            qt: torch.Tensor = module.weight_quant(module.weight)
            bw = qt.int().detach().cpu().numpy()
            binary_weights.append((name, bw.tolist()))

    return binary_weights
```

Figure 4.4. The code block above is responsible of extracting binarized weights from the BNN model. This is the first step to achieve the BNN deployment on data plane.

The weight deployment leverages the `bfrt` interface to configure the Tofino's match-action tables with the binarized network parameters. Each layer's weights are transformed from the PyTorch state dictionary format into the specific binary encoding required by the P4 program, where weights are represented as single bits (+1/-1 values converted to 0/1). The controller establishes communication channels through virtual Ethernet interfaces to coordinate between the feature extraction CPU interface and the BNN execution engine.

For real-time input processing, the controller implements a packet-based inference pipeline where network traffic features are extracted, binarized according to the SHAP-selected feature set, and formatted into `BNNFeaturesHeader` packets for transmission to the Tofino dataplane. The `FEATURE_EXTRACTOR_CPU_INTF` receives raw network flows, applies the same binarization logic used during training (reducing from 168 to the selected subset of features), and forwards the binary feature vectors to the BNN executor. The controller maintains synchronization between concurrent active flows (`CONCURRENT_ACTIVE_FLOWS`) and handles the bidirectional communication required for retrieving inference results from the hardware-accelerated BNN, enabling sub-microsecond anomaly detection directly within the network dataplane.

4.2.4 Model refinement

In dynamic network environments, evolving traffic patterns often lead to performance degradation in static models. To preserve accuracy and adaptability, our framework integrates a **model refinement** mechanism based on a **teacher–student**

paradigm, where a stable MLP supervises the BNN through confidence-guided retraining. When the system detects low-confidence predictions, uncertain samples are labeled by the teacher and used to update the BNN weights. This iterative process enables the model to adapt to distribution shifts while maintaining efficient, hardware-friendly inference.

SHapley Additive exPlanations-based Feature Selection

SHAP [5] values represent a principled framework for interpreting machine learning models through the lens of cooperative game theory. They attribute each feature a "Shapley value" quantifying its marginal contribution to the model's prediction by considering all possible feature combinations. This approach ensures fairness and consistency in feature attribution, addressing the limitations of traditional feature importance measures. SHAP values provide both global interpretability—by summarizing overall feature influence—and local interpretability, explaining individual predictions. Applicable to a wide range of model architectures, including tree ensembles, neural networks, and linear models, SHAP offers a unified, mathematically rigorous method for understanding complex predictive systems. By enhancing transparency and accountability, SHAP has become a foundational tool in the field of explainable artificial intelligence (XAI).

When a refinement phase is triggered, we optimize the feature set for deployment on the Tofino ASIC's BNN executor. We implement a SHapley Additive exPlanations (SHAP) based feature selection methodology specifically designed for binarized network traffic datasets. Our approach begins with the full 168-feature — the standard statistics size gathered from the statistics pipeline — binarized dataset, where each feature represents a single bit extracted from network flow characteristics. The SHAP analysis is performed using trained neural network models (both BNN and MLP architectures) to compute feature importance scores through Shapley values, which quantify each feature's contribution to the model's prediction output.

The feature selection process operates in two phases: first, we train baseline models on the complete 168-bit feature space and apply SHAP explainability to rank features by their mean absolute SHAP importance values. Subsequently, we select the top-ranked features to create reduced feature sets suitable for the hardware constraints of the Tofino BNN executor. This reduction is critical as the P4-programmable dataplane has limited computational resources and register space for real-time inference. The selected features maintain the binary nature of the original dataset, ensuring compatibility with the BNN's binary operations while significantly reducing the computational overhead from 168 to a more manageable number of features (typically 92-133 features, depending on the architecture and

SHAP source model).

This SHAP-guided dimensionality reduction preserves the most discriminative network flow characteristics for anomaly detection while enabling efficient deployment in network hardware.

The resulting models, trained on SHAP-selected features, demonstrate comparable or improved performance compared to their full-feature counterparts, validating the effectiveness of our feature selection strategy for hardware-accelerated network intrusion detection systems.

Retraining

The retraining phase serves as a fundamental adaptation mechanism aimed at mitigating performance degradation under evolving data distributions. When a model encounters inputs that diverge from its original training domain, its predictive confidence tends to decrease, signaling a potential loss of generalization. These out-of-distribution samples — or "critical samples" — for which the model exhibits uncertain predictions—are systematically collected throughout the evaluation process, creating a repository of challenging cases that expose the model's limitations under distributional change. When enough critical cases are gathered, typically after sufficient exposure to the shifted distribution, the BNN model undergo s complete reinitialization and retraining using a combination of the original training data and the accumulated critical samples. Crucially, the SHAP feature selection process is recalculated at this juncture using the MLP teacher model, which demonstrates superior robustness to domain shift compared to the BNN models. This strategic choice leverages the MLP's stable performance characteristics to identify features that remain informative across both domains, ensuring that the retrained BNN with SHAP features benefits from feature selections that are less susceptible to the distributional changes that have degraded the original models' performance. The retraining process thus creates an adaptive learning cycle where model weaknesses inform targeted improvements through both sample augmentation and refined feature selection.

4.3 Data Plane

The data plane implements the real-time packet processing logic of the proposed framework directly on the Tofino ASIC. It is divided into two main components: a **P4-based feature extraction pipeline**, responsible for collecting and encoding flow-level statistics, and a **BNN executor pipeline**, which performs in-network inference through bitwise operations and controlled packet recirculation. Together, these modules enable fully hardware-embedded anomaly detection at line rate.

4.3.1 P4-based Feature Extraction Pipeline Implementation

The Tofino feature extraction pipeline, implemented in `feature_extractor.p4`, operates as a high-performance stateful packet processing system that computes network flow statistics directly within the switch dataplane at line rate. The pipeline processes incoming packets through a series of specialized control modules including flow hashing (`FlowHashing`), packet counting (`PacketsCounter`), TTL analysis (`TTL`), packet type classification (`PacketType`), inter-arrival time computation (`IAT`), and byte statistics (`Bytes`). Each flow is tracked using a hash-based indexing system that maintains state across multiple packets until the flow reaches maturity (defined by `BIDIRECTIONAL_FLOW_MATURE_TIME`), at which point the extracted features are packaged into a BNN header and forwarded to the neural network executor.

The complete feature vector comprises 168 binary features that capture comprehensive network flow characteristics extracted during the stateful processing phase. While basic counters, flags, and discrete measurements are computed directly within the P4 dataplane's match-action pipeline, more complex statistical features requiring floating-point arithmetic—such as mean packet sizes (`smean`, `dmean`) and derived statistical measures—are computed in the control plane before being binarized and integrated into the final feature vector. This hybrid approach leverages the Tofino's line-rate processing capabilities for simple aggregations while offloading computationally intensive operations to the control CPU, ensuring that the overall system maintains sub-microsecond processing latency for real-time intrusion detection.

The pipeline's architecture ensures that feature extraction occurs transparently within the network forwarding process, with minimal impact on packet forwarding latency. When a flow reaches maturity, the system activates the BNN header validation, triggers packet clonation through egress mirroring (`set_mirror()`), and

formats the extracted features for consumption by the BNN executor. This design enables the deployment of machine learning-based intrusion detection systems directly within the network infrastructure, providing real-time anomaly detection capabilities with hardware-accelerated performance characteristics essential for high-speed network security applications.

4.3.2 Binarized Neural Network Executor Pipeline

On data plane side, deployed within a Tofino ASIC model, the BNN pipeline is running, allowing classifications to be inferred at wire-rate. In this subsection, I will explain how we implemented the executor using the native operations offered by the ASIC device.

Our work propose three BNN variants, with different neural units counts and hidden layer number, exploring different scenarios, resource consumptions and performances. We will refer to them with the following names and architectures: `Tiny` $\leftarrow [98, 21, 2]$, `Dense` $\leftarrow [133, 43, 2]$ and `Wide` $\leftarrow [128, 32, 8, 2]$. Since they basically are Binarized Multi Layer Perceptron, I recall that each of the sudden layers is a `fc`.

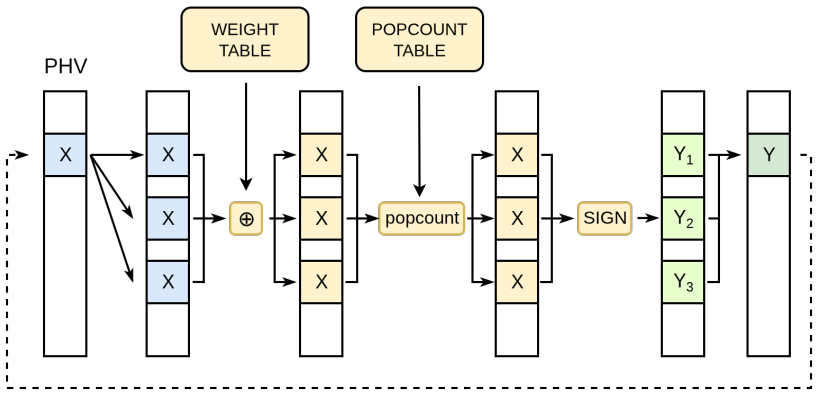


Figure 4.5. The above image represents the forward pass logic overview. Every inference step is showed where each input features are replicated, processed, and then aggregated to produce the output that will feed the next layer's input vector.

Forward-pass on Tofino

The forward-pass (FP) logic uses a combination of bit-wise operations and packet recirculation to iteratively produce the BNN prediction. Upon flow maturity in achieved, inference is triggered by the reception of an Ethernet packet with a prior defined *ether_type* and a custom nested header(**BNNHdr**). The latter contains

all the necessary fields needed by the data plane to store and compute partial and final results.

```
header bnn_pkt {
    bit<8> layer_no;
    bit<8> l0_out_1;
    ...
    bit<8> l0_out_4;
    bit<8> l1_out;
    bit<16> input_offset;
    bit<16> input_offset_cp;
    bit<8> pop_recirc;
    bit<8> nrs_recirc;
    popcount_t pop1;
    ...
    popcount_t pop7;
}
```

Listing 4.1. In the BNN tiny variant, this is the BNNHdr definition.

Copy step

Upon trigger, the forward-pass starts and input features are replicated once per neuron in the current layer, enabling fully parallel per-neuron computation in a single pipeline stage. In the proposed three BNNs variants, `Dense` and `Tiny` are 7-parallel-neurons enabled, `Wide` computes 4 neurons results at a time instead. This is due to the increased hidden layers number, Tofino ASIC model limitation obliged us to reduce that parameter in order to successfully compile the P4 program.

XNOR as Dot Product

Once the BNN input has been replicated and all the available parallel neurons are ready, the weights are retrieved from the corresponding table and combined with the input features via the bitwise $XNOR$ ($\oplus$) operator.

During preprocessing, all weights equal to -1 are converted to 0 before being loaded into the data plane. This conversion is necessary for representing signed binary values in a way that is compatible with the XNOR operation. Table 4.2 illustrates why the conversion is required: it ensures that the XNOR can act as a direct replacement for multiplication in neural networks where weights are constrained to the bipolar set $[-1, +1]$.

In practice, this mapping leverages the fact that binary multiplication and logical equivalence (XNOR) yield the same outcome when 1 encodes $+1$ and 0 encodes -1. Thus, the XNOR operator can efficiently simulate a dot product between the input vector and the weight vector.

Bin activations	Bin weights	XNOR
$-1\,(0)$	$-1\,(0)$	$+1\,(1)$
$-1\,(0)$	$+1\,(1)$	$-1\,(0)$
$+1\,(1)$	$-1\,(0)$	$-1\,(0)$
$+1\,(1)$	$+1\,(1)$	$+1\,(1)$

Table 4.2. The table shows the relation between the multiplication and XNOR result with $\{1,0(-1)\}$ input, since Tofino supports only positive integers.

Popcount operation

Next, the **popcount** operation is computed to measure how many neurons from the previous calculation contribute positively to the classification. Network ASIC hardware does not natively support this operator, and two main approaches are commonly used to address this limitation.

One approach is the technique called **SWAR** (or SIMD Within a Register), where the result is obtained by applying a sequence of parallel operations within a single variable. Even if SWAR focuses on small and parallel computations (4.2), it does not adapt well to hardware constraints. Every register assignment (assignments to x in 4.2) depends on the previous one. Each sequential assignment or computation is mapped to an independent stage in the hardware pipeline, and the number of required stages to complete the execution would be unsustainable. Furthermore, the lack of flexibility in reusing intermediate results makes this solution inefficient when scaling to wider architectures.

An alternative approach is to use **prefilled popcount tables**. All x-bit permutations are mapped to the corresponding popcount value. The parameter x must be chosen as a divisor of the weights' width, which ensures efficiency in the recirculation logic (discussed in Section 4.3.3). The exact value of x depends on the underlying BNN architecture: for `Tiny` and `Dense`, $x = 14$, while for `Wide`, $x = 16$. Depending on the chosen variant, a popcount table consumes between 3.25% (14K entries) and 13% (64K entries) of the available SRAM in the stage where it is placed. In addition, once prefilled, the tables can be reused without further computation overhead, which makes them appealing in scenarios where memory availability is less critical than processing speed.

Considering that our ASIC model platform supports multiple parallel lookups (up to 16 per stage) and that only a fraction of the available SRAM is consumed, we believe this method is more sustainable than SWAR. In practice, this strategy balances memory usage against computational overhead, leading to a design that better matches the throughput requirements of modern BNN inference.

```
def popcount32(uint32_t x):
    // m* are statics masks
    x = (x & m1 ) + ((x >>  1) & m1 );
    x = (x & m2 ) + ((x >>  2) & m2 );
    x = (x & m4 ) + ((x >>  4) & m4 );
    x = (x & m8 ) + ((x >>  8) & m8 );
    x = (x & m16) + ((x >> 16) & m16);
    return x;
```

Listing 4.2. Popcount algorithm on 32bit string using SWAR. Every assignment to x is depending on its previous value and that leads to an high stages consumption.

Aggregation step

To obtain binary activations, the XNOR operation is first performed between the inputs and the corresponding binary weights. The raw results of these XNOR computations are then passed through the SIGN function, which binarizes the outcome and introduces the necessary non-linearity for the network. In this way, each neuron produces a 1-bit output rather than a integer-valued activation. Finally, the 1-bit outputs from all neurons in the current layer are concatenated to form a compact binary vector, which serves as the input to the next layer of the network.

4.3.3 Recirculation logic

Complete and entire forward pass in a single packet traversal is not feasible within our choosen ASIC, recalling the stages number constraint. To overcome the issue we propose a recirculation protocol that splits the inference phase in a series of modular partial processing. Each recirculation advance the current result to a certain grade, until we reach the final outcome.

The provided hardware ports can assume different usages, the vast majority operates as normal communication links with external environment, however, a small range can be used as a recirculation port. That means we can forward the processed packet to them and it will be fed again in the Ingress pipeline, ready to repeat the processing.

This section will target the `Dense` BNN, other variants can be easily adapted changing the loop-unrolling length, the input vector length and the division factor in the loop definition; some mention to different variants are present to ease the explanation.

Algorithm 1 BNN inference algorithm for `Dense`.
L is the number of layers.
N_L is the number of neurons of the layer l.
a_l is the output of layer l.
$W_{l,n,k,i}$ is the partial weight on index i for layer l, neuron group n and weight batch k

Require: a vector of 133-bit inputs a_0, the binary weights W.
Ensure: the MLP output a_L.

```
for l = 1 to L do
    {Neurons-wise recirculation}
    for n = 1 to (N_L)//7 do
        for k = len(W_{l,n}//14) to 1 do
            {Weight-wise recirculation}
            a_l ← a_l|| (Sign((a_{l-1,k} ⊕ W_{l,n,k,1})) ≪ 7(n-1) +6
            a_l ← a_l|| (Sign((a_{l-1,k} ⊕ W_{l,n,k,2})) ≪ 7(n-1) +5
            ...
            a_l ← a_l|| (Sign((a_{l-1,k} ⊕ W_{l,n,k,3})) ≪ 7(n-1) +1
            a_l ← a_l|| (Sign((a_{l-1,k} ⊕ W_{l,n,k,4})) ≪ 7(n-1)
        end for
    end for
end for
```

Neurons batch

Parallel execution of neurons within the same layer is a key characteristic of neural networks, as it enables faster forward (and backpropagation) computation. However, the number of available PHV containers is limited. After a thorough analysis—taking into account all headers, weights, and intermediate results stored in the PHV throughout the pipeline—we determined that the maximum feasible number of parallel neurons is four for `Wide variant` and seven for the others. This parallelization will result in an inner loop-unrolling of the Algorithm 1. Different grades of unrolling are achieved depending of the chosen BNN variant ; for instance, the `Dense` variant will be composed by seven unrolling section.

Weights batch

Additionally, our popcount computation method, described in Section 4.3.2, presents two limitations:

- It cannot compute the population count for binary strings longer than 16 bits.

- Seven (or four) popcount tables have to be instanced, given that just one lookup is possible for each table per packet processing. However, since all of them make independent lookups, they can be easily parallelized without consuming more than one MAU stage.

To address constraints described in Sections 4.3.3 and 4.3.3, we introduced a recirculation mechanism that effectively overcomes these limitations.
Algorithm 1 implements a multi-layer BNN inference entirely in the Tofino data plane by interleaving two forms of *recirculation*: **neuron-wise** and **weight-wise**. Let L be the total number of layers, and N_l the number of neurons in layer l. The input to layer l is a 133-bit vector a_{l-1}, and the full binary weight matrix for layer l is partitioned into batches $W_{l,n,k}$ of 14 bits each, where $n \in [1, \ldots, N_l/7]$ indexes groups of four neurons, and $k \in [1, \ldots, |W_{l,n}|/14]$ indexes successive 14-bit segments of those four neurons' weight vectors.

- ***Neuron-wise recirculation*** (outer loop): for each neuron group n, we process seven neurons in parallel. We replicate a_{l-1} once per group n so that each pipeline instance computes the four SIGN outputs.
- ***Weight-wise recirculation*** (inner loop): within a group, the 133-bit weight vector for those four neurons is too wide to XOR in one pass. Instead, we "stream" it 14 bits at a time. On each pass k, we extract the next 14-bit chunk $W_{l,n,k}$, perform bit-wise XOR with the corresponding 16 bits of a_{l-1}, look up its popcount via a preloaded SRAM table, threshold it into a 1-bit SIGN result, and shift it into the correct 4-bit output slot (at offsets $(n-1) \times 7 + \{6,5,4,3,2,1,0\}$). After each inner iteration, the packet is recirculated back through the same ingress pipeline to handle the next weight chunk.

Once all neuron groups n are done, the $7 \times N_l/7 = N_l$ bits of a_l have been assembled (i.e., layer output), and the pipeline moves on to layer $l+1$.

This double-recirculation approach trades *time* (multiple passes through the ingress pipeline) for *space*: by slicing both the weight vectors and the neuron outputs into manageable 14-bit and 4-bit pieces, we avoid exceeding Tofino's per-stage ALU and PHV-allocation limits, yet still achieve fully parallel per-neuron inference across the entire network.

4.3.4 Input Layer

As explained in the FP introduction 4.3.2, the trigger packet does not contains the input features, but rather just the needed fields for computations. This absence of input is due to the **Parser** constraints of Tofino, it is not able to handle such long

sequence other than the required processing field. We will retrieve input batches directly from **Tofino Registers** — hence, a persistent array-like blocks within the data plane where each block is accessible just once per packet processing. — and the control CPU is in charge of loading input batches before sending the trigger packet. Each weight recirculation will have its corresponding input batch to be retrieved and XNOR-ed with.

Multiple flows are managed through an input_offset_index passed by the control CPU, indicating the starting index from which the input flow can be analyzed.

This is valid just for the input layer, from the first hidden unit the needed activations are carried out by the custom header.

Chapter 5

Results

This chapter presents the experimental evaluation of the proposed run-time adaptive intrusion detection framework built on top of programmable data planes. The goal of this analysis is to validate the feasibility, accuracy, and efficiency of deploying BNN inference directly within a Tofino ASIC/TNA target, as well as to assess how quantization-aware learning, SHAP-based feature selection, and confidence-guided retraining contribute to system adaptability under dynamic traffic conditions.

We begin by examining the training performance of the proposed architectures — `Tiny`, `Dense`, and `Wide` — each representing a different balance between computational cost and learning capacity. Through comparisons across two major cybersecurity datasets (CICIDS2017 and CIC-UNSW-NB15), we analyze how SHAP-based feature selection enhances the predictive accuracy and generalization capability of the quantized models. We also investigate the influence of architectural depth and width on F1-score, precision, and recall, providing insights into the trade-offs between accuracy and hardware efficiency.

The second part of the chapter focuses on the **inference performance** of the in-network BNN executors. Here, we quantify both latency and resource utilization in the programmable pipeline, demonstrating that even under the architectural constraints of a real ASIC, our models sustain acceptable delay. We further analyze the impact of model size and recirculation depth on execution time, confirming that pipeline traversal frequency is the primary latency factor.

Finally, we explore the framework's **adaptive behavior** under distribution shift. Using the confidence-based retraining mechanism, the system dynamically detects performance degradation and selectively retrains on newly labeled traffic samples provided by the teacher MLP. This experiment emulates a realistic evolving network scenario where traffic patterns change over time, highlighting the framework's ability to recover performance and maintain stability through online refinement.

Overall, this chapter provides both empirical and architectural evidence that **quantized neural inference within a programmable switch** is not only feasible but also effective. The presented results demonstrate how lightweight BNN models, supported by SHAP-driven feature selection and adaptive retraining, can form the foundation for intelligent, self-improving network monitoring directly in the data plane.

5.1 SHAP Feature Selection Comparison: MLP vs BNN Base Models

Base models have the same input dimensionality as the CICIDS2017 and UNSWNB15 datasets, hence 168 bit/features. MLP and BNN base models are used as targets on which SHAP values are computed to order bit features by importance. Even if BNN base model is not realistically used in the final framework, it is important as evaluation study since it helps us in the understanding of feature contributionsby a real BNN, whose deeply different from a full-fledge model.

These plots reveal how individual features contribute to model predictions. Each dot represents a single sample, with the horizontal position showing the feature's impact (SHAP value) on the final decision. Features are ordered by overall importance from top to bottom. The color coding is straightforward: red dots indicate high feature values(1) , blue dots show low values(-1), and the horizontal spread tells us how consistently a feature behaves across different samples.

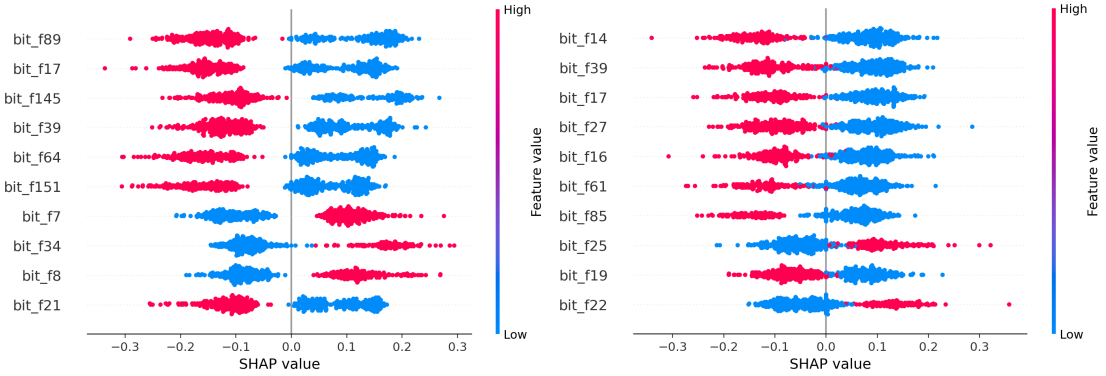


Figure 5.1. Top 10 SHAP features computed on Tiny BNN over the CIC-UNSW-NB15 dataset. This BNN model casts a wide net, identifying numerous features with substantial variation in their contributions.

When examining the feature selection patterns, a clear divergence emerges between the two approaches. As showed in 5.1, the BNN teacher model demonstrates a broader feature selection strategy, identifying numerous features with substantial variation in their contributions. Features like bit_f89 and bit_f17 show particularly wide SHAP distributions, suggesting these features play different roles depending on the specific sample being analyzed.

The MLP approach reveals a more focused trend. Here, fewer features dominate the selection process, but their contributions appear more concentrated and

consistent across samples as Figure 5.2 shows. Features such as bit_f149 and bit_f148 emerge as clear priorities in the MLP-derived selection, yet they show less prominence in the BNN's feature ranking. This suggests the MLP has identified features that provide precise, targeted signals rather than those with broad distributional importance.

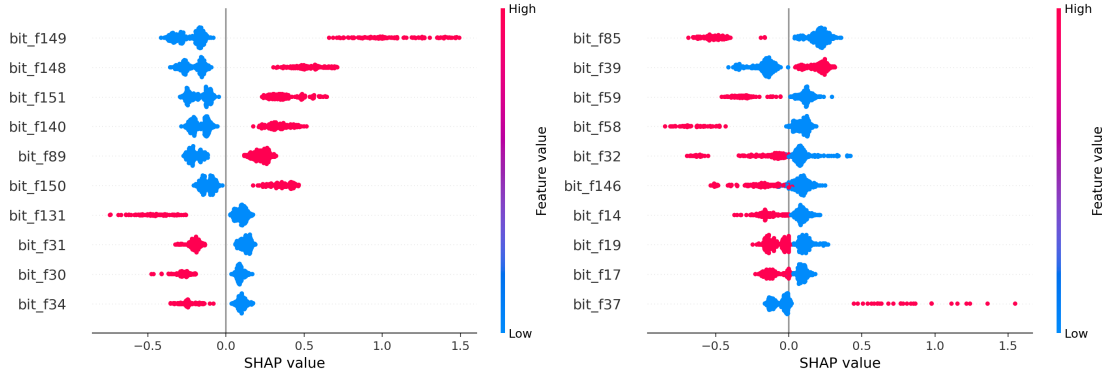


Figure 5.2. Top 10 SHAP features computed on Tiny architecture over the CIC-UNSW-NB15 dataset. This MLP model casts a narrow net, identifying features with minimal variation in their contributions.

SHAP values tendences: BNN vs MLP: These differences reflect the distinct characteristics of each architecture's feature learning capacity. The BNN's wider network structure enables it to distribute feature importance across multiple pathways, leading to the selection of features whose contributions vary significantly across different sample contexts. The MLP's more constrained architecture focuses its learning capacity on fewer, more critical features, gravitating toward those that offer concentrated predictive value. This focused approach appears to guide the model toward more selective feature identification that prioritizes precision over breadth in feature utilization.

Further SHAP-based beeswarm plots are available in Appendix B.

5.2 Training on SHAPed features

A central aspect of our training methodology is the application of SHAP-based feature selection, which identifies the most informative input bits within the 168-dimensional dataset.

Each BNN configuration demonstrates a specific trade-off between complexity, accuracy, and generalization. As the network width and depth increase, precision and F1-score typically improve, albeit with diminishing returns.

5.2.1 Architecture-Wise Comparison

Model	Dataset	SHAP	Accuracy	Precision	Recall	F1-Score
MLP base	CICIDS2017	–	0.989	0.992	0.981	0.987
	CIC-UNSW-NB15	–	0.964	0.904	0.996	0.948
BNN base	CICIDS2017	–	0.950	0.944	0.936	0.940
	CIC-UNSW-NB15	–	0.947	0.883	0.968	0.923
TF BNN	CICIDS2017	BNN	0.954	0.974	0.914	0.944
	CIC-UNSW-NB15	base	0.955	0.886	0.992	0.936
TF BNN	CICIDS2017	MLP	0.956	0.996	0.897	0.943
	CIC-UNSW-NB15	base	0.952	0.886	0.981	0.931

Table 5.1. Detailed performance comparison for `Tiny` architecture [168/98,21,2]. SHAP feature selection improves model performance for both MLP- and BNN-based masks. **Tiny achieves the highest F1-score on CIC-UNSW-NB15 (0.936)**.

Tiny architecture. Table 5.1 illustrates how SHAP enhances the performance of the lightweight `Tiny` model. The MLP-derived SHAP mask yields the best results overall, achieving an F1-score of 0.944 on CICIDS2017 and 0.936 on CIC-UNSW-NB15. Compared to the base BNN, these results show an improvement of approximately 1.5% in F1-score, confirming the effectiveness of SHAP in guiding feature relevance. The slightly higher precision (0.996) compared to recall (0.897) on CICIDS2017 indicates that the model makes fewer false positives, though it remains somewhat conservative in attack detection.

Dense architecture. The `Dense` model [168/132,21,2] expands the hidden representation compared to `Tiny`, resulting in stronger balance between recall and precision.

As seen in Table 5.2, the `Dense` configuration provides balanced results with F1-scores of 0.943 on CICIDS2017 and 0.927 on CIC-UNSW-NB15, demonstrating consistent robustness across datasets. The MLP-based SHAP mask achieves the best overall performance, but the BNN-derived mask performs comparably, suggesting that both approaches capture largely overlapping informative features. This confirms that as model capacity increases, dependence on the teacher model for SHAP importance diminishes.

Model	Dataset	SHAP	Accuracy	Precision	Recall	F1-Score
MLP base	CICIDS2017	–	0.990	0.991	0.984	0.988
	CIC-UNSW-NB15	–	0.965	0.905	0.997	0.949
BNN base	CICIDS2017	–	0.957	0.965	0.931	0.948
	CIC-UNSW-NB15	–	0.949	0.889	0.967	0.926
TF BNN	CICIDS2017	BNN	0.953	0.954	0.933	0.943
	CIC-UNSW-NB15	base	0.948	0.891	0.959	0.924
TF BNN	CICIDS2017	MLP	0.953	0.953	0.932	0.943
	CIC-UNSW-NB15	base	0.950	0.889	0.969	0.927

Table 5.2. Detailed performance comparison for `Dense` architecture [168/132,21,2]. SHAP-based training enhances recall and stability across datasets.

Wide architecture. The `Wide` model [168/128,32,8,2] introduces greater representational capacity by adding a second hidden layer and wider neuron groups, allowing more complex feature interactions.

Model	Dataset	SHAP	Accuracy	Precision	Recall	F1-Score
MLP base	CICIDS2017	–	0.990	0.991	0.984	0.988
	CIC-UNSW-NB15	–	0.965	0.906	0.997	0.949
BNN base	CICIDS2017	–	0.956	0.991	0.903	0.945
	CIC-UNSW-NB15	–	0.944	0.868	0.979	0.920
TF BNN	CICIDS2017	BNN	0.961	0.990	0.916	0.952
	CIC-UNSW-NB15	base	0.952	0.888	0.979	0.931
TF BNN	CICIDS2017	MLP	0.961	0.991	0.914	0.951
	CIC-UNSW-NB15	base	0.952	0.884	0.985	0.932

Table 5.3. Detailed performance comparison for `Wide` architecture [168/128,32,8,2]. **Wide achieves the best F1-score on CICIDS2017 (0.952)**.

From Table 5.3, it is evident that the `Wide` model delivers the best overall classification performance, achieving an F1-score of 0.952 on CICIDS2017 and 0.932 on CIC-UNSW-NB15. Precision and recall remain well balanced, confirming that widening the architecture enhances discriminative power without overfitting. However, this improvement comes at the expense of higher hardware cost and delay, as we will discuss in Section 5.4. Consequently, `Wide` demonstrates the upper bound of attainable accuracy, while smaller models achieve near-optimal

results at significantly lower complexity.

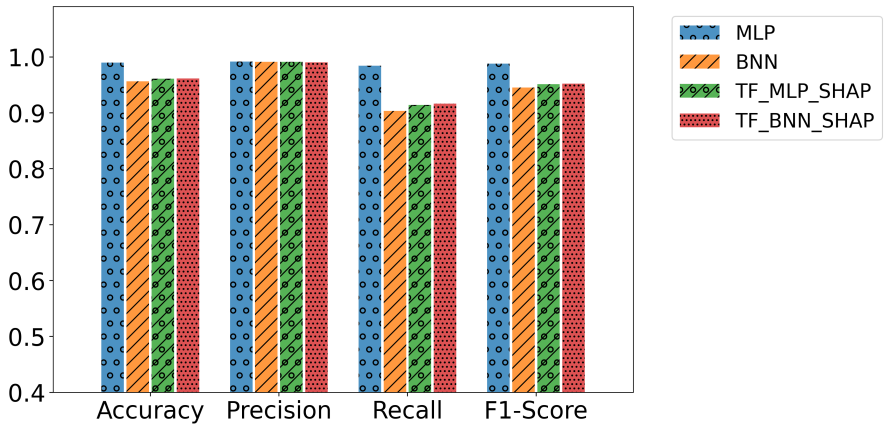


Figure 5.3. The above image shows the evaluation performance of the `Wide` architecture over the CICIDS2017 under the same Training configurations. Feature selection by SHAP not only allowed to reduce the input dimensionality by a factor minimum factor of 20%, but also achieved a consistent performance improvement in every metrics. MLP and BNN are the base models without feature selection, TF_*_SHAP are the BNNs models trainedon best SHAP features computed respectively from MLP and BNN base models.

For a more detailed view of the SHAP-derived training across all architectures, — including accuracy, loss and confidence score behavior — the complete set of plots is provided in Appendix A.

5.3 Retraining Evaluation under Distribution Shift

The retraining phase evaluation was conducted on the `Tiny` and `Dense` architectures, as these models are better suited for deployment on the Tofino ASIC. Their reduced recirculation depth and higher degree of parallelization make them more compatible with the hardware's pipeline constraints, ensuring efficient and stable execution. Moreover, the confidence score mechanism performs more reliably on these architectures, since the larger number of neurons in the penultimate layer provides a smoother and more statistically meaningful distribution of activation-based confidence values.

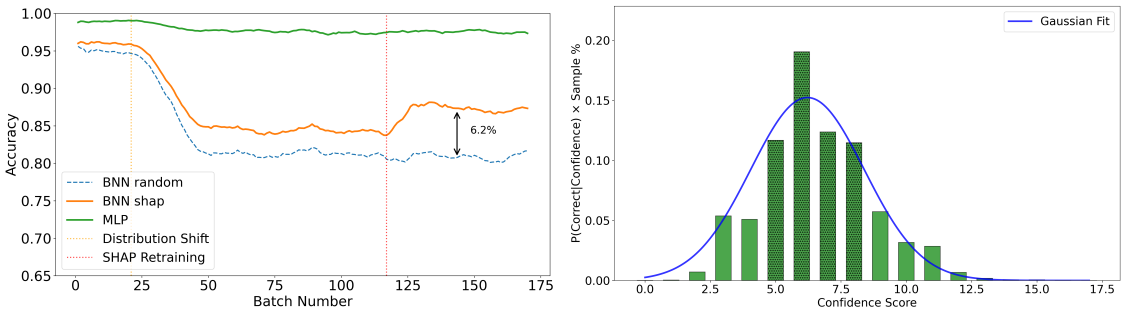


Figure 5.4. Distribution shift evaluation for the `Tiny` architecture. **Left:** Accuracy trend across batches as CIC-UNSW-NB15 samples are gradually introduced into CICIDS2017 traffic. The vertical dashed lines indicate the *distribution shift* and subsequent *retraining* points. After retraining, the model presents a 6.2% accuracy gap vs the random BNN, demonstrating improved adaptability to evolving data. **Right:** Confidence score distribution fitted with a Gaussian curve, showing that prediction correctness correlates with higher confidence values, validating the reliability of the confidence-based retraining trigger. In addition, the latter shows that certain confidence score are correctly recurring, we will use the higher 80% percentile confidence score value to select the "confident" scores; In that plot, the darker bins on the right represent the chosen "confident" score.

To assess the adaptability of the proposed system under dynamic network conditions, we evaluated the `Tiny` and `Dense` BNN architectures in a controlled domain-shift experiment. The goal was to simulate a realistic scenario where the network traffic gradually changes over time. Initially, both models operated on pure CICIDS2017 batches, representing a stable environment. After a defined *distribution shift* point, CIC-UNSW-NB15 samples were progressively introduced into the batch stream, increasing linearly until they accounted for 40% of the traffic. When the confidence monitoring module detected a consistent decline

in reliability, the teacher MLP triggered the retraining phase by labeling low-confidence samples, aggregating them with the initial training set, and initiating a new refinement cycle.

As shown in Fig. 5.4, the `Tiny` model experienced a gradual degradation as the distribution drift intensified. Its accuracy decreased from 0.960 on pure CICIDS2017 data to 0.906 under moderate shift, reaching a minimum of 0.856 at the highest level of contamination. After the retraining phase, accuracy stabilized around 0.874, with a peak of 0.900, corresponding to a recovery of about 6.2%. Despite its minimal structure, the model proved capable of adapting to the new data distribution, outperforming the randomized baseline that suffered a larger total degradation of 13.8%. These results indicate that SHAP-based feature selection significantly improves generalization, allowing even compact architectures to maintain robustness with limited resources.

The `Dense` model, reported in Fig. 5.5, exhibited a more stable trend during the same experiment. Starting from a baseline accuracy of 0.957, it dropped to 0.903 during the shift and reached 0.848 at maximum drift intensity. Following retraining, accuracy recovered to an average of 0.891 and peaked at 0.907, corresponding to a 3.9% improvement with respect to the randomized model. The confidence score distribution shown in Fig. 5.5 confirms a clear relationship between prediction correctness and confidence, validating the reliability of the confidence-guided sample selection strategy. Compared to its random-feature baseline, which experienced an 11.6% drop, the SHAP-enhanced `Dense` architecture demonstrated stronger resilience and faster recovery.

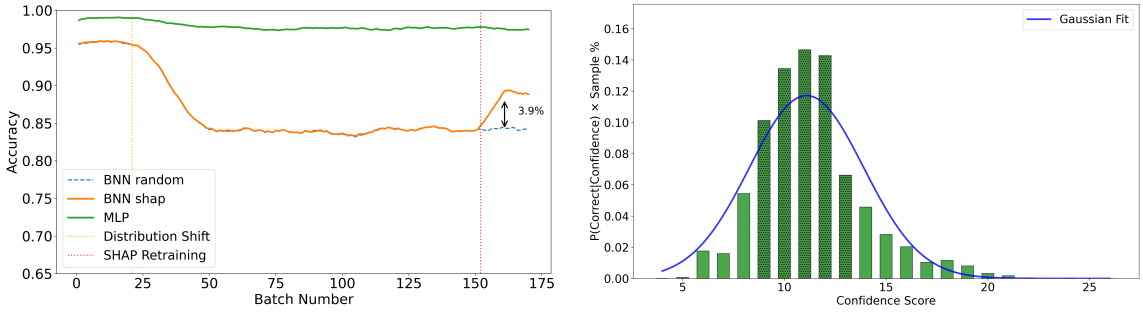


Figure 5.5.

Overall, these results highlight the effectiveness of the proposed confidence-based retraining approach in mitigating the effects of distribution shift. While the `Dense` model provides higher stability and smoother adaptation, the `Tiny` model achieves comparable recovery with substantially fewer resources. The combination

of SHAP-guided feature selection and teacher–student refinement enables quantized models to remain adaptable and efficient under evolving traffic patterns, validating the framework's potential for real-time deployment within programmable data planes.

5.4 Performance Evaluation of BNN Executors

In this section, we evaluate the runtime behavior and hardware efficiency of the three proposed Binarized Neural Network (BNN) executors — `Tiny`, `Dense`, and `Wide` — deployed within the Tofino ASIC. Our analysis focuses on two key performance aspects: inference delay and resource utilization. The objective is to understand how architectural variations and recirculation strategies impact both latency and hardware footprint, thereby guiding design trade-offs for in-network intelligence at line rate.

5.4.1 Inference Delay Analysis

The inference delay measures the time required for a single input packet to complete the neural inference process within the data plane. Delay is normalized against a baseline P4 program implementing a simple L3 switch, used as a reference for comparison.

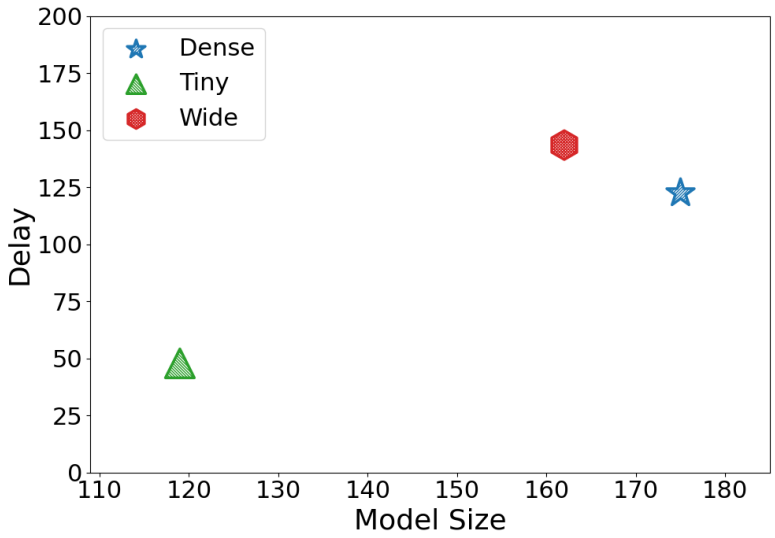


Figure 5.6. Relative inference delay of BNN architectures on the Tofino ASIC. The X-axis represents the model dimension in terms of neuron count, while the Y-axis reports delay relative to a baseline L3 switch. The goal of this analysis is to quantify how model size and recirculation depth affect inference latency. Although the `Tiny` architecture requires about 50× the unit time of a simple L3 switch, this overhead remains acceptable given the high-speed per-packet processing capabilities of the data plane. The results confirm the hypothesis that larger models incur greater delay due to increased pipeline traversals.

Figure 5.6 illustrates the relative inference delay for the three architectures.

The `Tiny` variant achieves the lowest delay, benefiting from its compact structure and reduced number of recirculation passes. In contrast, the `Dense` model introduces moderate delay, balancing neuron parallelism and computational depth through efficient stage utilization and seven-way parallel neuron execution. The `Wide` architecture, which increases layer count but reduces neuron parallelism to four, exhibits the highest delay, as it requires more recirculations to complete a full inference pass.

Overall, the results highlight a clear trade-off: as model size and layer depth increase, inference delay grows proportionally to the number of required pipeline traversals. Pipeline traversal frequency thus emerges as the dominant latency factor. For completeness, the `feature_extractor.p4` module — which performs only flow feature extraction without any recirculation — operates at approximately $2\times$ the unit time of a simple L3 program, confirming that recirculation overhead is the principal contributor to inference delay.

5.4.2 Resource Utilization Analysis

To complement the delay study, we now analyze how each architecture impacts hardware resource usage on the Tofino ASIC. This evaluation provides insights into the trade-off between model complexity and available memory resources, which directly influence scalability and deployability under line-rate constraints.

Table 5.4. Hardware resource utilization per packet traversal for each BNN executor.

Model	**SRAM [%]**	**MaxStageSRAM [%]**	**Recirc**
`Tiny`	4.7	35	25
`Dense`	4.7	35	64
`Wide`	7.6	65	69

Table 5.4 summarizes resource utilization for a single packet traversal within the programmable data plane. The results reveal important distinctions in memory occupancy and recirculation demand across the three architectures.

The `Wide` model consumes noticeably more SRAM, primarily due to its 16-bit prefilled popcount lookup tables, which are used to emulate neuron activation counting. In contrast, both the `Tiny` and `Dense` models employ 14-bit tables, significantly reducing their memory footprint. This design difference reflects the `Wide` architecture's need to accommodate larger intermediate computations, resulting from its increased layer width.

The `Tiny` and `Dense` models share the same fundamental pipeline structure and memory allocation strategy, but differ in their recirculation behavior. The `Dense`

executor requires 64 recirculations compared to the `Tiny` model's 25, highlighting how deeper network topologies directly translate into additional processing rounds. Each recirculation corresponds to an additional packet traversal through the match-action pipeline, increasing both latency and aggregate resource consumption.

The `MaxStageSRAM` metric indicates the highest memory occupancy reached in any single pipeline stage. The `Wide` model approaches 65% of available memory bandwidth, whereas `Tiny` and `Dense` remain around 35%, leaving sufficient headroom for additional tasks or simultaneous packet processing. This consideration becomes critical for real-time network operations, where multiple packets may require concurrent processing.

It is important to note that all values reported in Table 5.4 refer to a single packet traversal. The total inference cost scales roughly linearly with the number of recirculation cycles, making the `Tiny` model approximately 2.5 times more efficient than the `Wide` model in aggregate resource consumption. TCAM utilization remains negligible (0.3%) for all architectures, since the BNN design exclusively relies on exact matches, consistent with the deterministic and quantized nature of binary weight operations.

In summary, the performance characterization demonstrates that lightweight BNN configurations such as `Tiny` are the most suitable candidates for real-time inference in programmable switches. They achieve a favorable balance between latency, memory footprint, and architectural simplicity, while still delivering meaningful classification capability at line rate.

Chapter 6

Conclusion

This thesis has explored the integration of adaptive deep learning mechanisms within programmable network devices, focusing on the implementation of Binarized Neural Networks (BNNs) on the Intel Tofino ASIC. Through the proposed system, it has been demonstrated that it is feasible to perform neural inference directly within the data plane while low delay and minimal hardware footprint. The presented architecture merges the responsiveness of line-rate packet processing with the intelligence of adaptive learning, forming the basis for a new generation of self-optimizing network systems.

The work successfully bridges the gap between two traditionally separate domains: machine learning and high-performance network programmability. By combining a P4-based feature extraction pipeline, a hardware-constrained BNN executor, and a control-plane refinement loop based on confidence-driven feedback, this framework provides an autonomous mechanism capable of identifying, learning, and reacting to evolving traffic patterns. The proposed retraining logic, triggered by confidence degradation, proved effective under simulated distribution shifts, allowing both `Tiny` and `Dense` models to recover accuracy after exposure to unseen traffic. Notably, SHAP-guided feature selection enhanced generalization, reducing model size while preserving discriminative power.

From a hardware perspective, the system efficiently utilized Tofino's limited on-chip memory, with even the largest model (`Wide`) consuming less than 8% of total SRAM. The trade-off analysis across `Tiny`, `Dense`, and `Wide` configurations revealed clear relationships between architectural complexity, inference delay, and resource consumption. These insights enable informed design choices for future in-network learning deployments, balancing accuracy, latency, and scalability.

Beyond its immediate results, this work highlights a broader paradigm: the convergence of learning and forwarding within the same physical plane. The ability to adapt models dynamically at line rate opens the door to distributed intelligence in networking—where the network itself contributes to understanding and managing

traffic in real time, without reliance on external compute resources.

It is important to note that the two datasets used in this work, CICIDS2017 and CIC-UNSW-NB15, share similar feature spaces and comparable traffic characteristics. While this similarity facilitates controlled experimentation and model comparison, it may also limit the diversity of behaviors observed during evaluation. In a real network environment, traffic dynamics are likely to be more heterogeneous, encompassing unseen attack patterns, protocol variations, and device-specific behaviors. Consequently, future validation should focus on deploying the framework in operational scenarios to assess its robustness under truly unpredictable conditions and to better understand how the retraining and confidence mechanisms respond to complex, non-stationary traffic distributions.

Future Work

Several research directions naturally emerge from this foundation. First, while binarization enables unprecedented hardware efficiency, future explorations into **multi-bit quantization** on Tofino (e.g., 2- to 4-bit precision) could provide a more flexible compromise between accuracy and speed. Frameworks such as Brevitas already offer support for variable quantization, suggesting that partial precision models can be trained and mapped efficiently to PISA architectures. This evolution could allow finer gradient representation and more expressive feature interactions without exceeding hardware constraints.

Second, the **confidence-driven retraining mechanism** can be extended toward a more integrated and decentralized logic. Instead of reporting every inference result to the control plane, the data plane itself could act as a preliminary filter—*transmitting results only when confidence drops below a defined threshold*. Such a mechanism would offload the control channel, reduce communication overhead, and enable selective refinement of uncertain samples. This shift of partial intelligence toward the data plane would represent a step closer to fully autonomous, in-network learning systems.

Third, future developments should explore **alternative neural architectures** suited for in-network deployment. While Multilayer Perceptrons have proven effective for tabular flow features, other topologies—such as lightweight Convolutional or Graph Neural Networks—could better capture structural or temporal dependencies within packet sequences. Hybrid or hierarchical models distributed across multiple switches might also enhance scalability and cooperative learning in large-scale topologies.

Finally, integrating this work into real-time, multi-switch environments would enable large-scale validation and pave the way for **fully distributed, self-adaptive**

network intelligence. Such systems could not only detect and mitigate anomalies but also orchestrate resources, manage quality-of-service, and even predict network conditions—achieving the long-envisioned goal of a truly cognitive data plane.

In summary, this thesis establishes a concrete proof of concept that deep learning can coexist with the strict constraints of programmable switches. By embedding intelligence where data resides, we move toward networks that are not merely programmable but perceptive—capable of learning from the traffic they forward. The path forward lies in enriching this foundation with multi-bit precision, on-chip confidence reasoning, and diverse architectures, transforming today's programmable pipelines into tomorrow's intelligent network fabrics.

Appendix A — Extended Training Plots

This appendix presents the complete set of training-related plots for the three Binarized Neural Network (BNN) architectures — `Tiny`, `Dense`, and `Wide`. While the main body of the thesis discusses the overall training performance and key metrics, the following figures provide a more detailed view of each model's optimization dynamics and predictive behavior throughout the training process.

For each architecture, we report the evolution of the **training and validation loss**, the **accuracy** progression over epochs, and the corresponding **confidence score distribution** derived from active neuron statistics in the penultimate layer. These plots help visualize the convergence patterns achieved through Quantization-Aware Training (QAT) and highlight how model capacity influences stability and generalization. In particular, they allow for the identification of overfitting tendencies and fluctuations in prediction confidence, which are more pronounced in smaller networks due to their limited representational flexibility.

In addition to per-model plots, we include a series of **comparison figures** contrasting the training behaviors of `Tiny`, `Dense`, and `Wide` architectures under identical training conditions.

Overall, the plots reported in this appendix serve as a visual complement to the quantitative analyses presented in the Results chapter, providing a more comprehensive understanding of the learning dynamics and reliability of the proposed in-network BNN models.

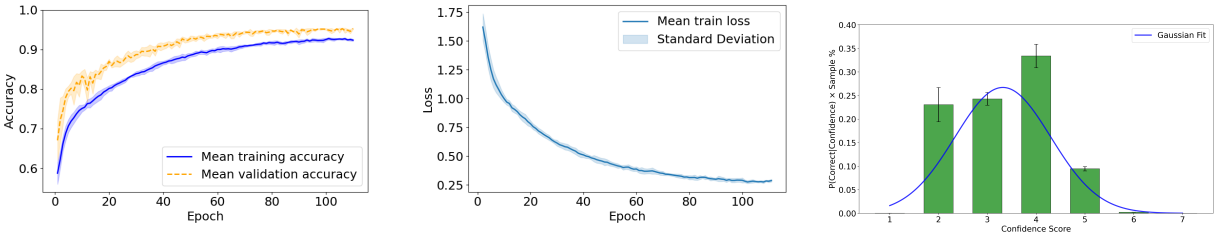


Figure 1. BNN accuracy and loss convergence during from-BNN-SHAP training for`Wide` over CIC-UNSW-NB15.Cross-validation confidence score trend on the right

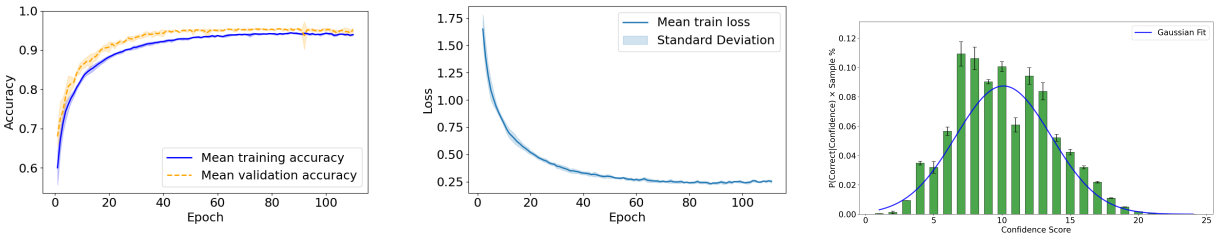


Figure 2. BNN accuracy and loss convergence during from-BNN-SHAP training for `Dense` over CIC-UNSW-NB15.Cross-validation confidence score trend on the right

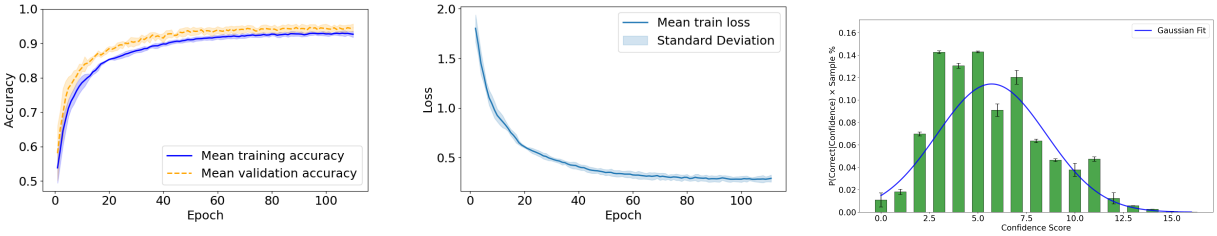


Figure 3. BNN accuracy and loss convergence during from-BNN-SHAP training for `Tiny` over CIC-UNSW-NB15.Cross-validation confidence score trend on the right

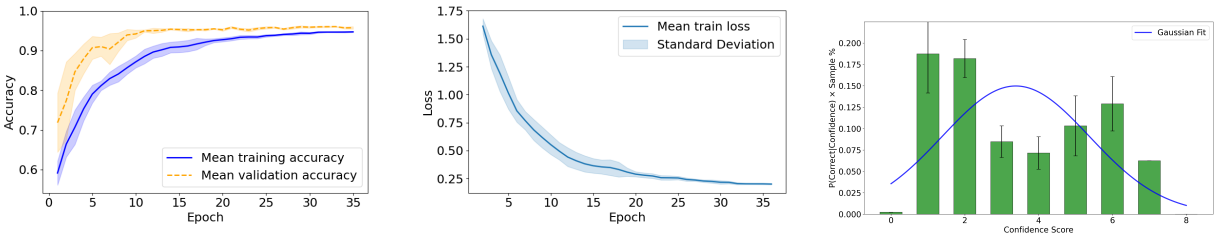


Figure 4. BNN accuracy and loss convergence during from-BNN-SHAP training for `Wide` over CICIDS2017.Cross-validation confidence score trend on the right

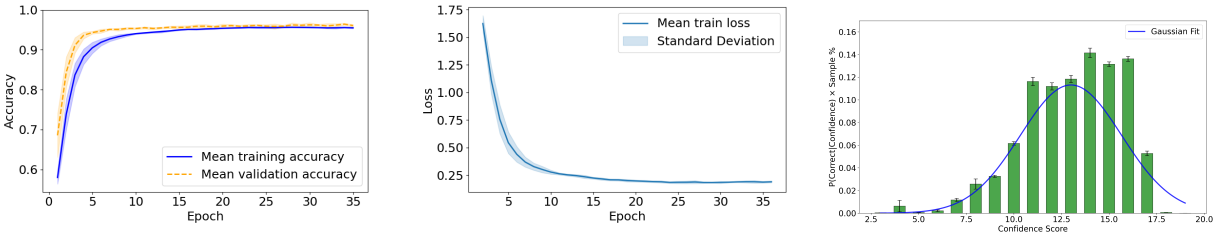


Figure 5. BNN accuracy and loss convergence during from-BNN-SHAP training for `Dense` over CICIDS2017.Cross-validation confidence score trend on the right

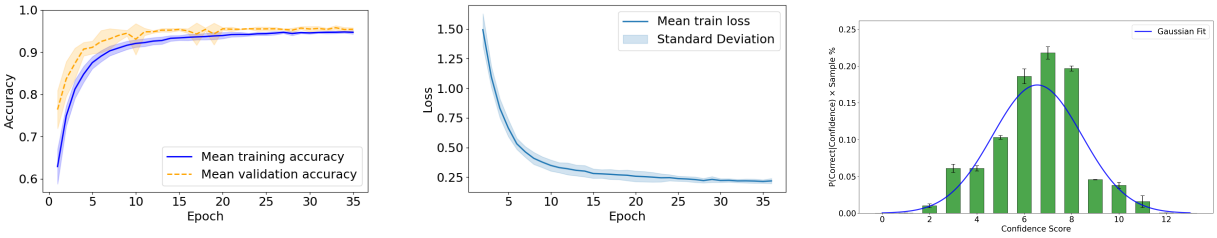


Figure 6. BNN accuracy and loss convergence during from-BNN-SHAP training for `Tiny` over CICIDS2017.Cross-validation confidence score trend on the right

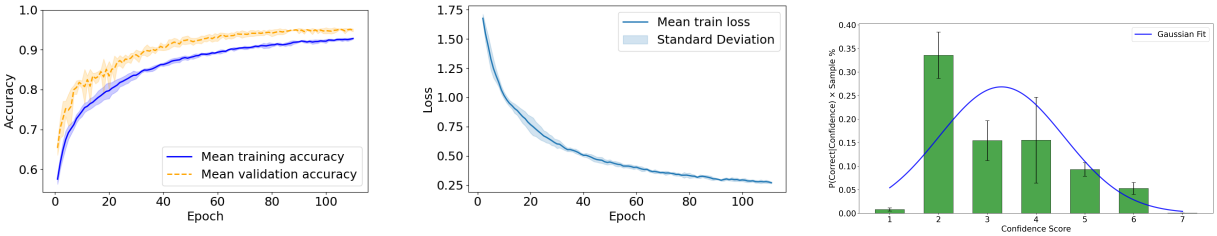


Figure 7. BNN accuracy and loss convergence during from-MLP-SHAP training for `Wide` over CIC-UNSW-NB15.Cross-validation confidence score trend on the right

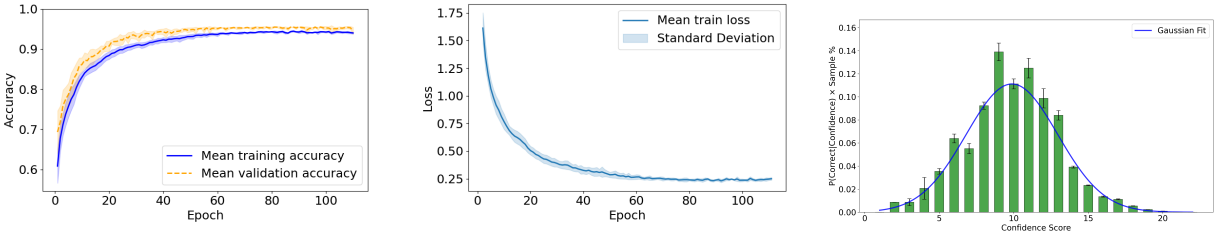


Figure 8. BNN accuracy and loss convergence during from-MLP-SHAP training for `Dense` over CIC-UNSW-NB15.Cross-validation confidence score trend on the right

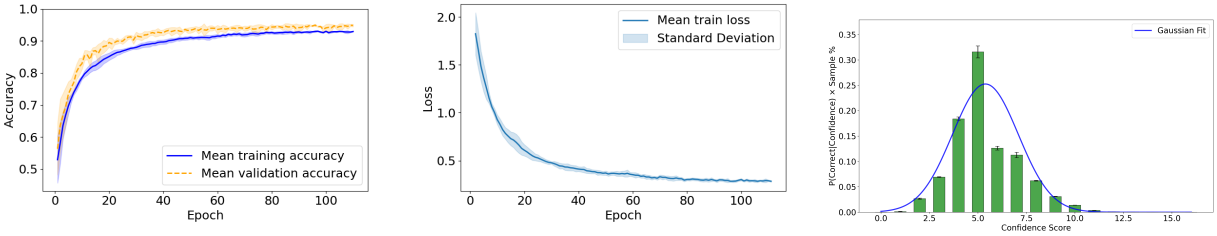


Figure 9. BNN accuracy and loss convergence during from-MLP-SHAP training for `Tiny` over CIC-UNSW-NB15.Cross-validation confidence score trend on the right

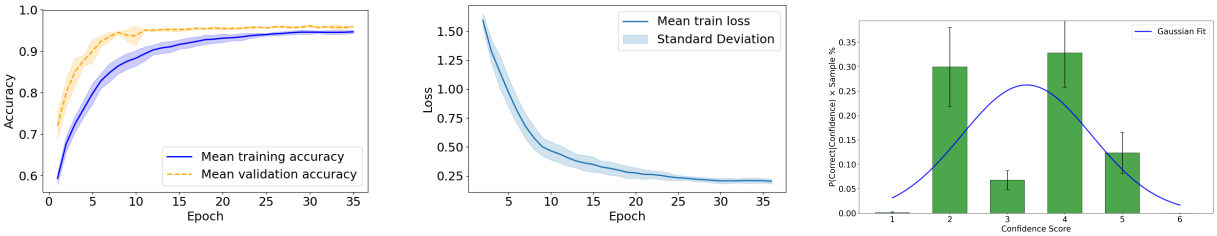


Figure 10. BNN accuracy and loss convergence during from-MLP-SHAP training for `Wide` over CICIDS2017.Cross-validation confidence score trend on the right

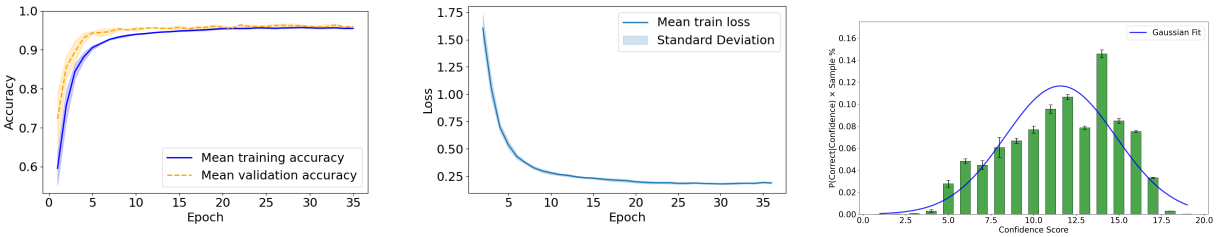


Figure 11. BNN accuracy and loss convergence during from-MLP-SHAP training for `Dense` over CICIDS2017.Cross-validation confidence score trend on the right

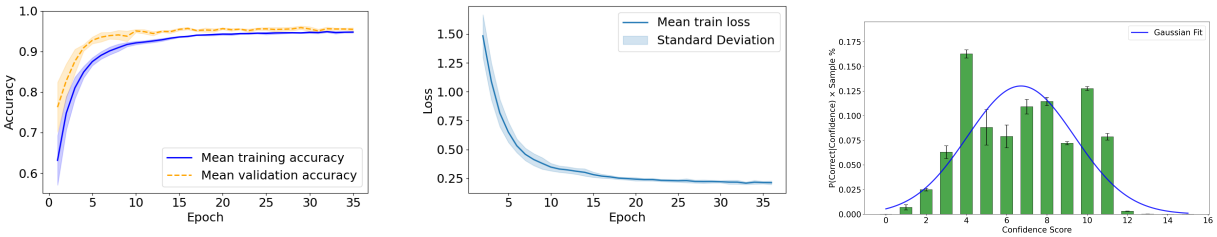


Figure 12. BNN accuracy and loss convergence during from-MLP-SHAP training for `Tiny` over CICIDS2017.Cross-validation confidence score trend on the right

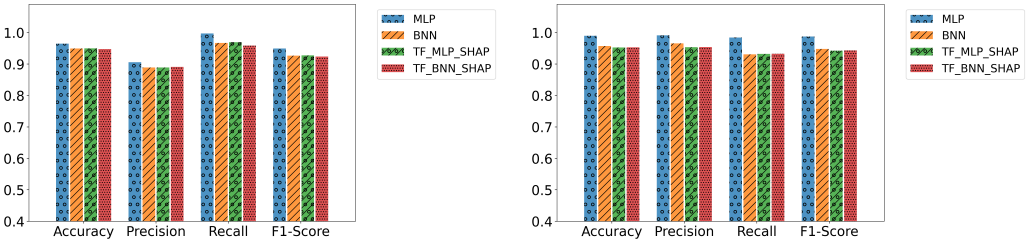


Figure 13. SHAP comparison for Dense Training: **Right.** evaluation over the CICIDS2017 dataset.**Left.** evaluation over the CIC-UNSW-NB15 dataset

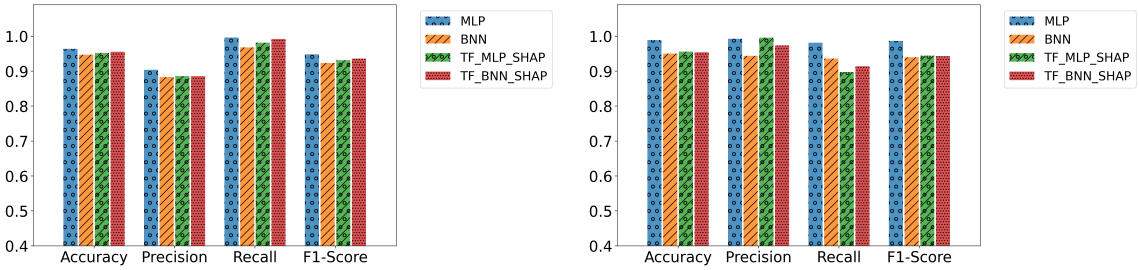


Figure 14. SHAP comparison for Tiny Training: **Right.** evaluation over the CICIDS2017 dataset.**Left.** evaluation over the CIC-UNSW-NB15 dataset

Appendix B — SHAP Feature Importance Plots

This appendix reports the complete set of SHAP beeswarm plots generated during the feature selection analysis for the `Tiny`, `Dense`, and `Wide` architectures. Each figure illustrates the contribution and variability of the most influential input features within the CICIDS2017 and CIC-UNSW-NB15 datasets. These visualizations complement the quantitative results discussed in Chapter **??**, providing a more detailed understanding of how individual bits influence model predictions. By comparing the distributions across architectures and teacher models (MLP- and BNN-based SHAP masks), we can observe how feature relevance evolves with network capacity and dataset characteristics, confirming the effectiveness of SHAP-based selection in guiding compact yet discriminative BNN training.

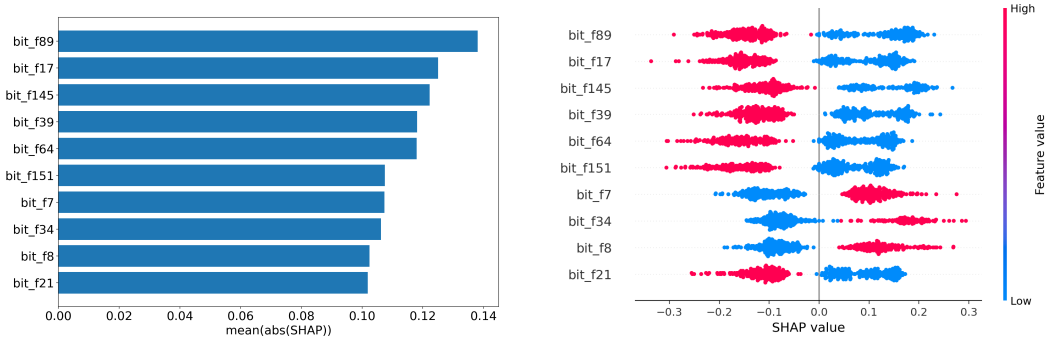


Figure 15. Features importance ranking using SHAP for `Tiny` BNN base model over CIC-UNSW-NB15.

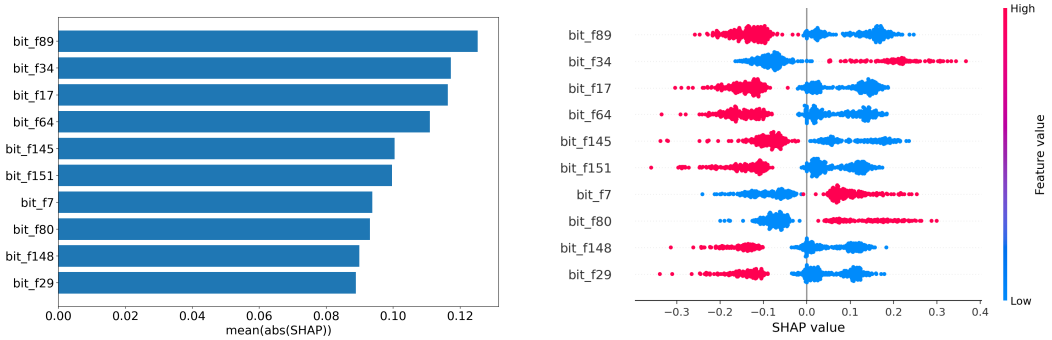


Figure 16. Features importance ranking using SHAP for `Dense` BNN base model over CIC-UNSW-NB15.

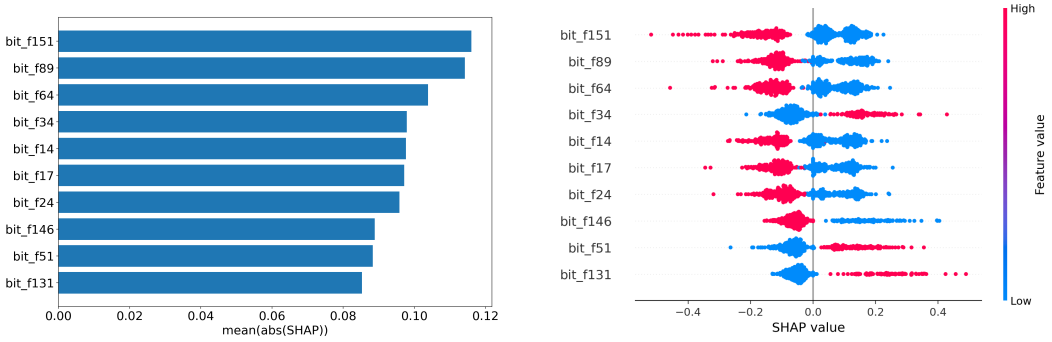


Figure 17. Features importance ranking using SHAP for `Wide` BNN base model over CIC-UNSW-NB15.

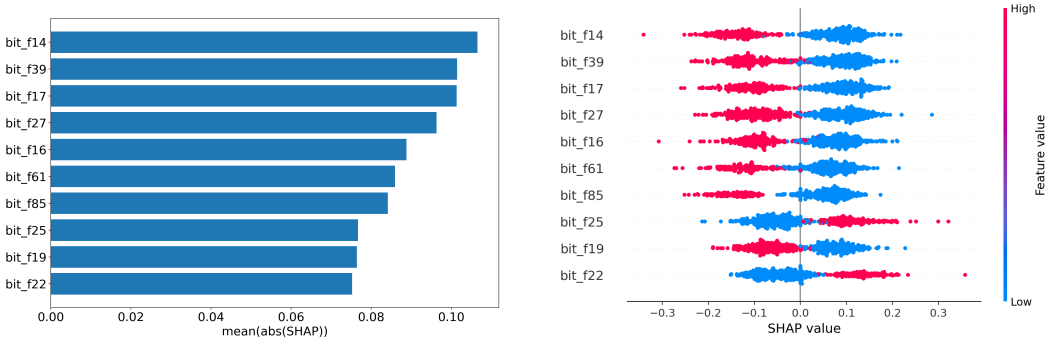


Figure 18. Features importance ranking using SHAP for `Tiny` BNN base model over CICIDS2017.

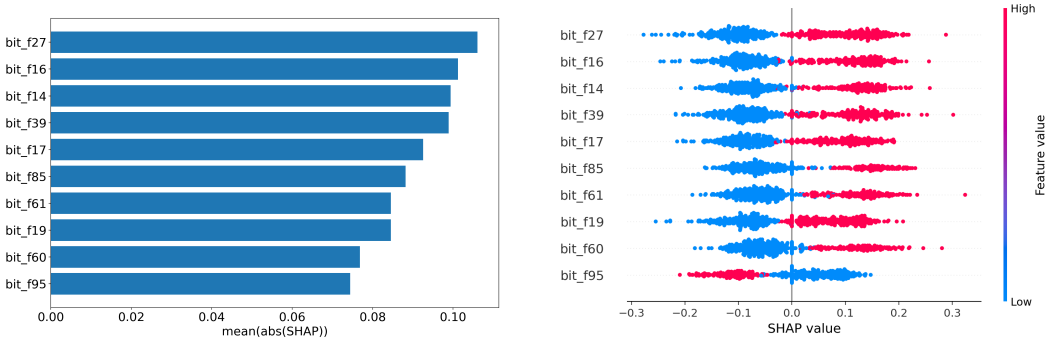


Figure 19. Features importance ranking using SHAP for `Dense` BNN base model CICIDS2017.

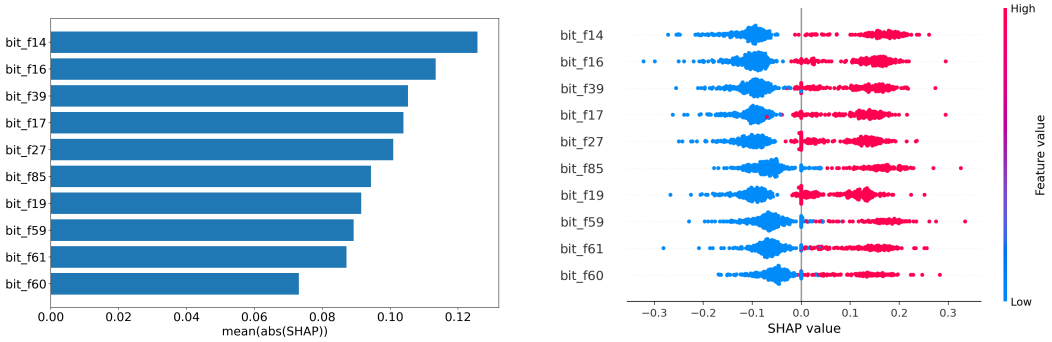


Figure 20. Features importance ranking using SHAP for `Wide` BNN base model CICIDS2017.

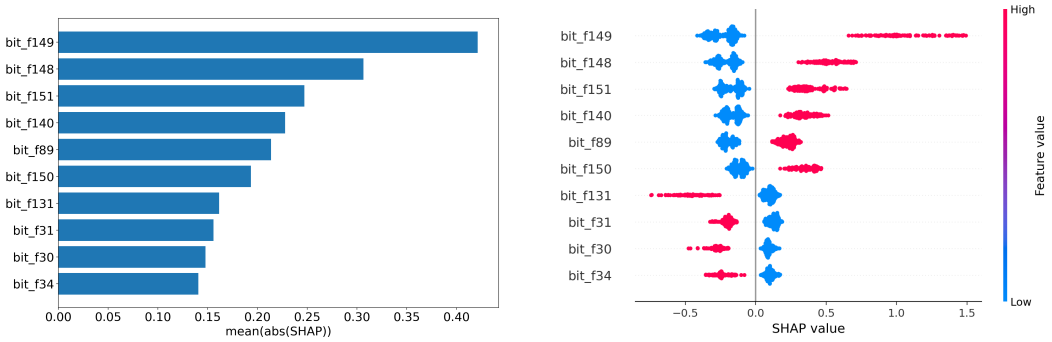


Figure 21. Features importance ranking using SHAP for `Tiny` MLP base model over CIC-UNSW-NB15.

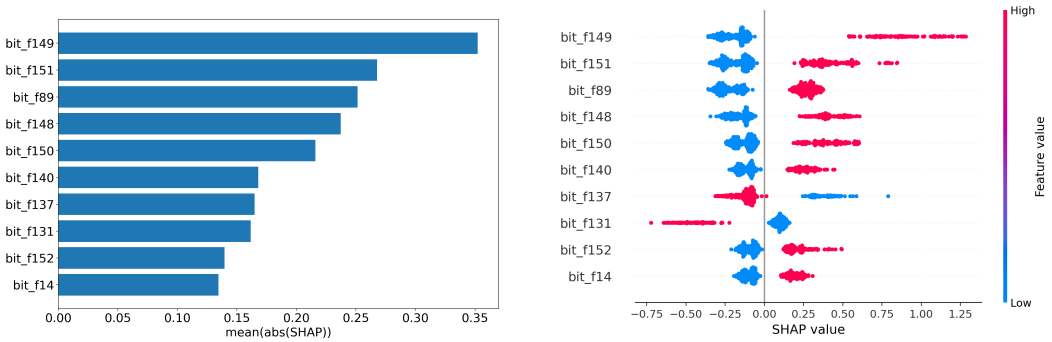


Figure 22. Features importance ranking using SHAP for `Dense` MLP base model over CIC-UNSW-NB15.

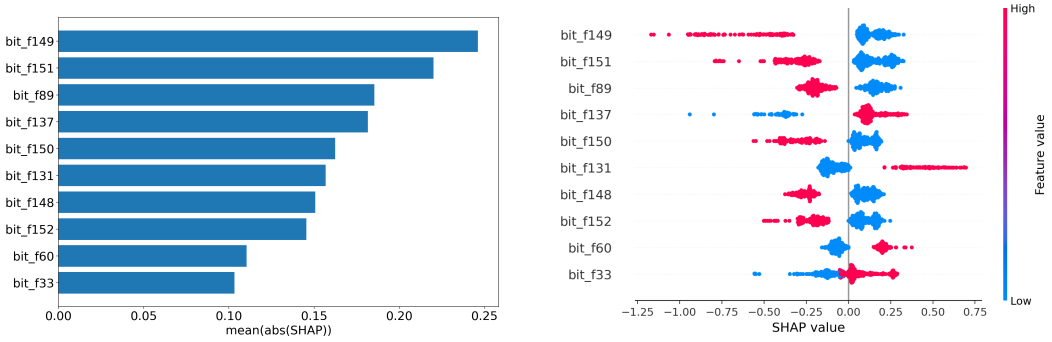


Figure 23. Features importance ranking using SHAP for `Wide` MLP base model over CIC-UNSW-NB15.

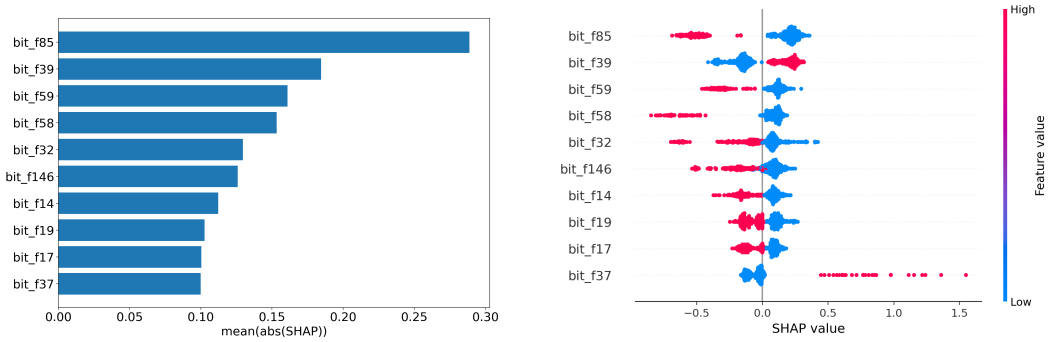


Figure 24. Features importance ranking using SHAP for `Tiny` MLP base model over CICIDS2017.

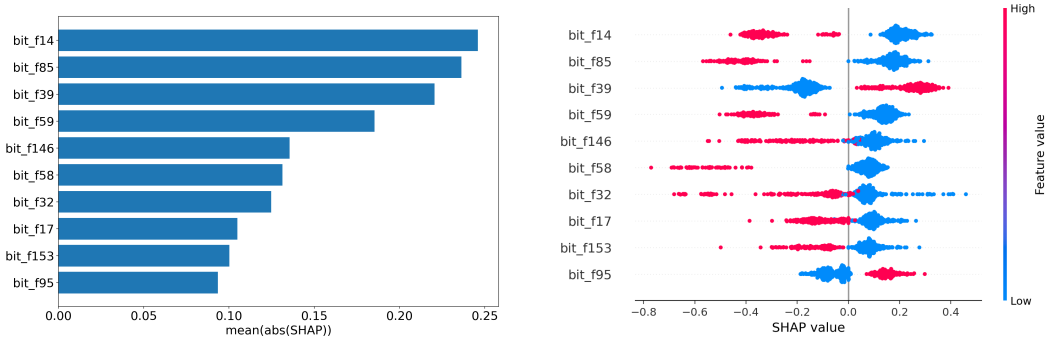


Figure 25. Features importance ranking using SHAP for `Dense` MLP base model CICIDS2017.

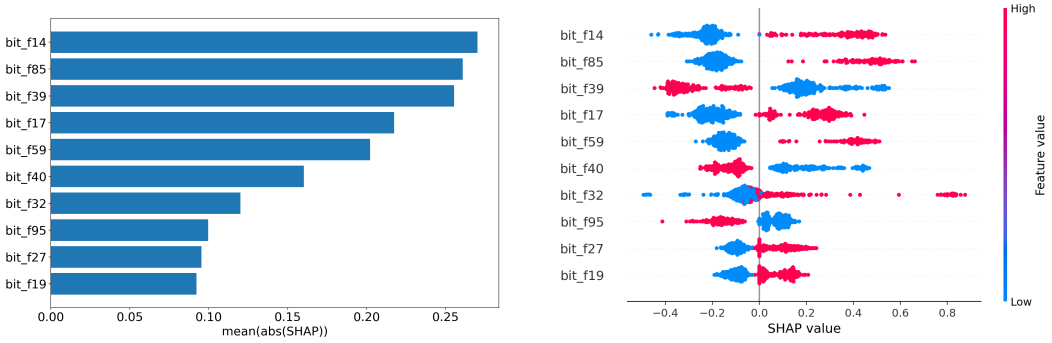


Figure 26. Features importance ranking using SHAP for `Wide` MLP base model CICIDS2017.

Bibliography

[1] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432, 2013.

[2] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[3] Suvrima Datta, Aditya Kotha, U. Venkanna, and K. Mallikharjuna Rao. Xnetiot: An extreme quantized neural network architecture for iot environment using p4. *IEEE Transactions on Network and Service Management*, 21(5):5756–5767, 2024.

[4] Giuseppe Franco, Alessandro Pappalardo, and Nicholas J Fraser. Xilinx/brevitas, 2025.

[5] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *CoRR*, abs/1705.07874, 2017.

[6] Hesamodin Mohammadian, Arash Habibi Lashkari, and Ali A. Ghorbani. Poisoning and evasion: Deep learning-based nids under adversarial attacks. In *2024 21st Annual International Conference on Privacy, Security and Trust (PST)*, pages 1–9, 2024.

[7] Nour Moustafa and Jill Slay. Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set). In *2015 Military Communications and Information Systems Conference (MilCIS)*, pages 1–6, 2015.

[8] Nour Moustafa and Jill Slay. The evaluation of network anomaly detection systems: Statistical analysis of the unsw-nb15 data set and the comparison with the kdd99 data set. *Information Security Journal: A Global Perspective*, 25(1-3):18–31, 2016.

[9] Nour Moustafa, Jill Slay, and Gideon Creech. Novel geometric area analysis technique for anomaly detection using trapezoidal area estimation on large-scale networks. *IEEE Transactions on Big Data*, 5(4):481–494, 2019.

[10] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network

quantization. *CoRR*, abs/2106.08295, 2021.

[11] Kamran Razavi, Shayan Davari Fard, George Karlos, Vinod Nigade, Max Mühlhäuser, and Lin Wang. Netnn: Neural intrusion detection system in programmable networks, 2024.

[12] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the network be the ai accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, pages 20–25, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *International Conference on Information Systems Security and Privacy*, 2018.

[14] Giuseppe Siracusano and Roberto Bifulco. In-network neural networks. *CoRR*, abs/1801.05731, 2018.

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

[16] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 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*, ASPLOS '22, page 1099–1114, New York, NY, USA, 2022. Association for Computing Machinery.

[17] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 25–33, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Kaiyi Zhang, Nancy Samaan, and Ahmed Karmouch. A machine learning-based toolbox for p4 programmable data-planes. *IEEE Transactions on Network and Service Management*, 21(4):4450–4465, 2024.

[19] Mai Zhang, Lin Cui, Xiaoquan Zhang, Fung Po Tso, Zhang Zhen, Yuhui Deng, and Zhetao Li. Quark: Implementing convolutional neural networks entirely on programmable data plane. In *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*, pages 1–10, 2025.

[20] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Liam Perreault, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Planter: Rapid Prototyping of In-Network Machine Learning Inference. *ACM SIGCOMM Computer Communication Review*, 2024.