Intelligent Data-Planes for Network Traffic Management

Kaiyi Zhang

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in
Electrical and Computer Engineering

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

© Kaiyi Zhang, Ottawa, Canada, 2025

Abstract

Network traffic management is increasingly complex due to the exponential growth of connected devices and bandwidth-intensive applications. As the volume of network traffic continues to rise, managing this data flow efficiently is a key challenge for modern communication networks. Traditional management techniques, which rely on static, protocol-driven methods, struggle to keep pace with the dynamic demands of today's networks. Emerging technologies like software-defined networking (SDN) and machine learning (ML) offer promising solutions by enabling more intelligent, real-time network management. However, in traditional ML-assisted SDN architectures, ML models are deployed in the control-plane, which relies on receiving relevant traffic information from the data-plane for analysis. This design poses a key limitation, as frequent control-plane/data-plane communication introduces latency that can delay time-sensitive services such as congestion mitigation and intrusion response. This thesis addresses this challenge by proposing a novel approach: introducing ML directly into the data-plane to enable real-time, autonomous decision-making, thus reducing the delays associated with traditional SDN architectures.

The primary objective of this research is to design and implement an intelligent data-plane with built-in ML inference, enabling real-time, local decisions and reducing reliance on the control-plane. First, we develop a quantization-aware ML toolbox that facilitates the training of ML models while simplifying their storage and execution within resource-limited data-planes. This approach ensures that quantized model inference can be effectively implemented in the data-plane, satisfying its operational and computational constraints. Second, to enable multi-phase decision-making within the data-plane, we design a confidence-based intrusion detection system that detects malicious flows at both early and later phases by leveraging the confidence level from early detection. Third, to support concurrent management tasks, we develop a novel in-network multi-task learning framework that performs simultaneous inference for multiple tasks in the data-plane. This approach is both resource-efficient and more accurate than single-task models by sharing feature representations among related tasks. Additionally, we enhance scalability by supporting distributed deployment, where different layers of a multi-task model can be offloaded across multiple switches. Finally, we address the challenge that offline trained models often struggle to adapt to dynamic network environments, where changing traffic patterns can degrade performance. We design an unsupervised drift detection mechanism in the data-plane that monitors distributional changes in traffic and triggers model updates when drift is detected. In addition, we present an in-network drift-aware traffic classification framework that not only classifies known traffic accurately but also identifies drifting samples that deviate from all known classes.

Acknowledgements

I am grateful for everything I have gained throughout this journey, and I have always believed that no achievement is accomplished alone. Just like the well-known butterfly effect, small actions can create meaningful change, and I am deeply thankful to everyone whose acts of kindness, support, and encouragement helped shape my path and brought me to where I am today.

First and foremost, I would like to express my deep and sincere thanks to my supervisor, Prof. Nancy Samaan, for her invaluable guidance and unwavering support. Prof. Samaan opened the door for me in 2019 when she accepted my transfer to a thesis-based master program and brought me into the world of networking research. Through every discussion, every challenge, every draft, every setback, and every milestone, Prof. Samaan guided me with patience, thoughtful insight, and constant encouragement. Prof. Samaan supported my growth at every stage of my academic path, gave me the freedom to explore ideas, and was always there when I needed direction. This thesis and this journey would not have been possible without her mentorship.

I would also like to thank my co-supervisor, Prof. Ahmed Karmouch, for his continuous help and support, both academically and personally. Our offices are close, so we met almost every weekday, and those daily interactions became an important part of my journey. He shared guidance not only in research but also in life, and I am deeply grateful for his help, mentorship, and company throughout these years.

My sincere appreciation also goes to my thesis committee members Prof. Marc St-Hilaire, Prof. Tet Yeap, Prof. Amiya Nayak, and Prof. Abdelhakim Hafid for examining my thesis and providing valuable feedback.

I would like to extend my heartfelt gratitude to Prof. Noa Zilberman for warmly hosting me in the Computing Infrastructure Group at the University of Oxford, and for her thoughtful guidance and constructive suggestions throughout my visit. I am also grateful for the opportunity to participate in group discussions and inspiring research activities. I feel truly fortunate to have met everyone in the group, and I would especially like to thank Dr. Changgang Zheng for the enjoyable collaboration.

I would also like to express my sincere thanks to Prof. Leandros Tassiulas for kindly hosting me in the SmartNets Lab at Yale University and for the valuable interactions, guidance, and support I received during my stay. I am equally grateful to the lab members for their help, kindness, and companionship. This experience was both inspiring and truly memorable.

I would also like to express my heartfelt thanks to my friends in Ottawa, Dr. Siwei Gu, Dr. Yulun Wu, Dr. Haitao Tian, Dr. Luming Fan, Dr. Yuan Wang, Dr. Haokun Yang, Dr. Le Qiao, Dr. Haiyang Wang, Bin Cheng, Juntong Yang, Tianhao Tao, Yuelang Huang, and many others, for their friendship and support throughout these years. The moments we shared, playing basketball and volleyball, lunches and dinners, gatherings and parties, game nights, watching games or shows, and our trips, are memories I will always treasure. My gratitude also goes to my lab mates, Zhuonan Huang and Taejoon Lee, for their help. I would also like to thank the friends I met during my master's studies, who supported me when I first arrived in Canada and helped me navigate life in a new country.

Finally, my deepest gratitude goes to my dear parents, whose unconditional love and support have carried me through every tough moment. They have always had my back, and their support for my journey to study abroad made everything possible.

Table of Contents

List of Tables ix

List of Figures x

Abbreviations xiii

1 Introduction 1

1.1 Motivation 1

1.2 Research Objectives 3

1.3 Contributions 4

2 Background 7

2.1 Software Defined Networking 7

2.2 Programmable Data-Planes 9

2.2.1 P4 Language 9

2.2.2 P4 Architectures 10

2.2.3 P4 Targets 11

2.2.4 Control-Plane and Data-Plane Interaction 12

2.3 Machine Learning 12

2.3.1 Supervised Learning 12

2.3.2 Unsupervised Learning 15

2.3.3 Reinforcement Learning 15

2.4 Traditional ML-based Network Management Schemes 15

2.5 Existing Solution for Intelligent Data-Planes 17

2.5.1 Tree-based Solutions 18

2.5.2 Neural Network-based Solutions 19

2.6 Multi-Task Learning 20

2.7 Conclusion 21

3 A Machine Learning-based Toolbox for P4 Programmable Data-Planes 22
3.1 Conceptual Overview . . . 22
3.2 Introduction . . . 24
3.3 Related Work . . . 26
3.3.1 Intelligent Data-Planes . . . 26
3.3.2 Intelligent Traffic Management . . . 27
3.4 Proposed Toolbox Overview . . . 29
3.4.1 Feature Extractor . . . 30
3.4.2 The ML Model Builder and Model Training . . . 31
3.5 Quantization Module . . . 32
3.5.1 Quantization and Dequantization Steps . . . 33
3.5.2 ML Quantization-Aware Training . . . 34
3.5.3 Deployment of the Quantized ML Model . . . 35
3.6 P4 Generation . . . 37
3.7 Theoretical Precision Analysis . . . 42
3.7.1 Notation . . . 42
3.7.2 Accuracy of Non-quantized Controller Models and Quantized IDP Models . . . 42
3.7.3 Quantization Analysis . . . 43
3.8 Performance Evaluation . . . 45
3.8.1 Dataset . . . 45
3.8.2 Experiment Setup . . . 45
3.8.3 Evaluation Metrics . . . 46
3.8.4 Classification Result . . . 47
3.8.5 INQ-MLT in Switches . . . 50
3.8.6 Precision Analysis Result . . . 52
3.9 Discussion . . . 53
3.10 Conclusion . . . 54

4 A Two-Stage Confidence-Based Intrusion Detection System in Programmable Data-Planes 55
4.1 Introduction . . . 55
4.2 Related Work . . . 57
4.3 Framework Overview . . . 57
4.3.1 Network Traffic Preprocessing . . . 59
4.3.2 The Convolutional Neural Network Design . . . 59
4.3.3 Training Scheme and Calibration Method . . . 60
4.3.4 Two-Stage Inference based on Reliability . . . 62
4.4 ML Model Mapping To P4 . . . 62
4.5 Performance Evaluation . . . 66
4.5.1 Dataset and Experimental Setting . . . 66
4.5.2 Results . . . 66
4.6 Conclusion . . . 68

5 Design, Implementation, and Deployment of Multi-Task Neural Networks in Programmable Data-Planes 69
5.1 Introduction 69
5.2 Related Work 72
5.2.1 Multi-Task Learning 72
5.2.2 Existing In-Network ML Solutions and Limitations 72
5.2.3 Design Challenges 75
5.3 An overview of MUTA 76
5.4 Multi-Task Model Training and Quantization 77
5.4.1 Model Architecture and Training 78
5.4.2 Quantization 78
5.5 Mapping Models to Switches 80
5.5.1 Data-Plane Mapping Methodology 80
5.5.2 Minimizing Stage Consumption 83
5.6 Deployment Orchestrator 84
5.6.1 Model Formulation 84
5.6.2 Constraints 85
5.6.3 Problem Formulation 86
5.7 Performance Evaluation 88
5.7.1 Use Cases 88
5.7.2 Multi-Task Model Performance 89
5.7.3 Hardware Resource Consumption 94
5.7.4 Latency and Throughput 96
5.7.5 Network-Wide Deployment Performance 97
5.8 Discussion 100
5.9 Conclusion 102

6 Towards Unsupervised Drift Detection in Programmable Data-Planes 103
6.1 Introduction 103
6.2 Related Work 104
6.3 In-Network Unsupervised Drift Detection 105
6.3.1 Method Overview 105
6.3.2 Feature Distribution 106
6.3.3 Similarity Measurement 107
6.4 P4 Implementation 108
6.4.1 Sliding Window Histogram 108
6.4.2 Distance Calculation 109
6.5 Performance Evaluation 110
6.5.1 Dataset 110
6.5.2 Experimental Setting 113
6.5.3 Results 113
6.6 Discussion 115
6.7 Conclusion 115

7 Towards In-Network Drift-Aware Traffic Classification 116
7.1 Introduction . . . 116
7.2 Related Work . . . 118
7.3 In-Network Drifting Samples Detection . . . 118
7.3.1 Triplet Network . . . 120
7.3.2 Distance-based Drifting Samples Detection . . . 122
7.4 P4 Implementation . . . 123
7.4.1 Encoder . . . 123
7.4.2 Distance Calculation . . . 124
7.5 Performance Evaluation . . . 124
7.5.1 Dataset . . . 124
7.5.2 Experimental Setting . . . 125
7.5.3 Results . . . 127
7.6 Conclusion . . . 130

8 Conclusion and Future Work 131
8.1 Conclusion . . . 131
8.2 Future Work . . . 132

List of Publications 134

References 135

List of Tables

3.1 The common used templates. 39

3.2 Features used for model training. 46

3.3 Classification performance for 1D-CNN. 48

4.1 Registers with their functionality 63

5.1 Comparison of advanced in-network neural network solutions. 73

5.2 Resource consumption for IIsy (DT): T1 - stalling prediction, T2 - startup delay prediction, T3 - resolution prediction, T4 - bitrate prediction. 95

5.3 Resource consumption for MUTA. 95

6.1 Injected concept drifts (mixed attacks) in CICIDS2017 dataset over time. . 110

7.1 The training scenarios (UNSW-IoT) 125

7.2 Drift detection results for UNSW-IoT dataset 126

7.3 Drift detection results for CICIDS2018 dataset 128

List of Figures

2.1 Main components in an SDN architecture. 8

2.2 Protocol independent switch architecture. 10

2.3 Structure of an MLP with 2 hidden layers. 13

3.1 The data-plane ML toolbox and its extensions across the thesis. 23

3.2 An overview of INQ-MLT toolbox. 29

3.3 Added quant nodes at the controller. 33

3.4 Packet processing pipeline. 37

3.5 Classification performance for MLP. 48

3.6 Comparison between BNN and 8-bit precision MLP. 49

3.7 The performance of INQ-MLT, controller-based ML and DT for anomaly detection. 50

3.8 Throughput comparison. 51

3.9 Comparison of quantization noises and error rates between standard training and QAT Schemes for MLP-1. 51

3.10 Comparison of quantization noises and error rates between standard training and QAT Schemes for MLP-2. 52

3.11 Comparison of quantization noises and error rates between standard training and QAT Schemes for 1D-CNN. 52

4.1 TSCIDS architecture. 58

4.2 Convolutional neural network structure. 59

4.3 Packet processing pipeline. 63

4.4 Reliability diagrams for CNN3. 65

4.5 Performance comparison for the calibrated and uncalibrated CNN3 under various threshold settings. 67

4.6 Performance comparison for only implementing CNN3, only implementing CNN5, and the two-stage scheme. 68

5.1 MUTA architecture. The control-plane is responsible for training and quantizing the model, generating the data-plane code, and determining the deployment strategy. The model layers are then distributed across multiple data-plane devices to cover intelligent services across the entire network. . 76

5.2 Proposed multi-task learning architecture. 78

5.3 Methodology for mapping layer computation to a match-action pipeline. The parser extracts the input vector from the packet header, followed by layer-wise inference executed through a sequence of match-action tables. The deparser then reconstructs the packet, embedding the output vector into the header. These intermediate layer values are forwarded to downstream switches, which use them as inputs for their assigned layers. The top-right corner illustrates parallel execution used to minimize stage consumption. . 79

5.4 Performance comparison between IIsy (DT) [1], single-task neural network (NN), and MUTA, using only 100 samples for label-limited tasks (resolution prediction and traffic class prediction) during training. 90

5.5 Performance comparison for label-limited tasks across different numbers of labeled training samples. 91

5.6 Performance comparison of the floating-point model (FP), quantized model without QAT (No QAT), and MUTA. 91

5.7 Impact of *hard-to-label* task loss weight on model performance. 92

5.8 Performance comparison between MUTA and control-plane ML schemes. (a) The control-plane ML baseline adopts the QoE prediction method presented in [2]. (b) The control-plane ML baseline follows the MTL scheme proposed in [3]. HL: Hidden Layer. 93

5.9 (a) Pipeline Relative Latency (R-Latency) on Tofino switches for tree models (IIsy), different layers in MUTA, and standalone switch.p4. (b) Throughput for different layers in MUTA on Tofino switches. 96

5.10 Network topology used for evaluation. 97

5.11 Network-wide deployment performance between MUTA and the greedy resource availability (GRA) baseline. 98

5.12 Relative objective weight of w (defined in (5.12)) influence on MUTA's deployment strategy. Tree topology: Depth 5 with 31 switches. 98

5.13 Relative objective weight of w (defined in (5.12)) influence on MUTA's deployment strategy. Fat-tree topology: 6 pods with 45 switches. 99

5.14 Impact of topology scale on solver execution time. 99

6.1 SPIDD architecture. 105

6.2 Impact of histogram bin count on drift detection performance. 111

6.3 Drift detection performance in mixed-attack scenarios. 112

6.4 Drift Detection Performance on UNSW-NB15. 114

7.1 The mapping from original feature space to latent space. In the latent space, the drifting sample is easier to identify due to its large distance from the in-distribution clusters. 117

7.2 IDAC architecture. 119

7.3 Triplet network architecture (left) and the triplet loss objective (right), which pulls the anchor closer to the positive and pushes it away from the negative. 122

7.4 Performance comparison of known traffic classification accuracy. 127

7.5 Boxplots of distances between testing samples and their nearest centroids for each class. Known traffic are classes 3, 6, and 9. 129

7.6 Latent space visualization without (left) and with (right) center loss. . . . 129

Abbreviations

AI Artificial Intelligence

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

BMv2 Behavioral Model version 2

BNN Binary Neural Network

CNN Convolutional Neural Network

CPU Central processing unit

DDoS Distributed Denial of Service

DL Deep Learning

DNN Deep Neural Network

DT Decision Tree

eBPF extended Berkeley Packet Filter

ECE Expected Calibration Error

FP Floating Point

GPU Graphics Processing Unit

IDP Intelligent Data-Plane

IDS Intrusion Detection System

ILP Integer Linear Programming

INT In-band Network Telemetry

IoT Internet of Things

MLP Multi-layer Perceptron

ML Machine Learning

MTL Multi-Task Learning

NIC Network Interface Card

NN Neural Network

P4 Programming Protocol-Independent Packet Processor

PHV Packet Header Vector

PISA Protocol-Independent Switch Architecture

PNA Programmable NIC Architecture

PSA Portable Switch Architecture

QAT Quantization-Aware Training

QoE Quality of Experience

QoS Quality of Service

RF Random Forest

RL Reinforcement Learning

RMT Reconfigurable Match Tables

SDN Software-Defined Network

SRAM Static Random-Access Memory

SVM Support Vector Machine

TCAM Ternary Content Addressable Memory

TNA Tofino Native Architecture

TVD Total Variation Distance

Chapter 1

Introduction

1.1 Motivation

Network traffic management is a fundamental aspect of modern communication networks. With the exponential increase in connected devices, ranging from smartphones and computers to Internet of thing (IoT) devices and industrial systems, managing the traffic flow of data has become more challenging and crucial. Network traffic management involves the practices, policies, and technologies used to ensure that data packets flow smoothly and efficiently between different nodes in a network, regardless of its size or complexity. This includes several key tasks, such as bandwidth allocation, load balancing, congestion control, and prioritizing certain types of traffic over others [4]. As networks continue to expand, the demand for more intelligent, automated, and real-time solutions for managing network traffic has led to the integration of advanced technologies such as software-defined networking (SDN) and machine learning (ML).

SDN has revolutionized how networks are designed, managed, and optimized for traffic flow [5]. Traditionally, networks were managed by embedded protocols that handled traffic flow statically, making real-time adjustments difficult. SDN, on the other hand, decouples the control-plane from the data-plane, allowing network administrators to manage and optimize traffic flow in real-time. This architecture allows the control-plane to make centralized decisions on traffic routing, resource allocation, and policy enforcement, thus enhancing the overall efficiency of network traffic management. By using programmable controllers, network operators can establish specific traffic rules for different types of data, ensuring that high-priority traffic, such as video conferencing or critical business applications, meets the required bandwidth and low-latency demands. SDN's programmability also allows for rapid scaling of network infrastructure to meet the needs of growing traf-

fic volumes, making it an ideal choice for large-scale enterprise networks and data center networks.

ML has emerged as a promising solution to address many of the challenges associated with network traffic management, particularly in the context of SDN [6]. By integrating ML algorithms into network operations, SDN controllers can make more informed, proactive decisions to optimize traffic flow, reduce latency, and predict network conditions. ML models can analyze large volumes of network traffic data, detect patterns, and predict future traffic trends [7]. This enables network administrators to optimize resources dynamically, anticipate congestion before it happens, and adjust routing strategies. For example, ML techniques such as supervised learning and reinforcement learning can be applied to forecast network traffic demands, allowing the SDN controller to make preemptive adjustments to traffic routing, thus minimizing delays and bottlenecks [8]. In particular, reinforcement learning has shown significant potential in enabling adaptive decision-making, where the controller learns from past actions and network responses to refine its traffic management strategies [9]. The integration of these intelligent techniques into network infrastructure not only enhances the efficiency of traffic management but also reduces the burden on the control-plane, as certain decisions can be offloaded to automated models.

While SDN and ML bring a new paradigm to network traffic management, they also introduce several challenges, particularly in the communication between the data-plane and the control-plane. In current ML-based network management schemes [10], the ML model is deployed on the control-plane, meaning that the data-plane must continuously send traffic data to the control-plane to request management decisions. The control-plane processes the data, executes the ML model, and returns the decision to the data-plane. This back-and-forth communication leads to non-negligible delays. For instance, in time-critical services, where rapid identification of network incidents is crucial, such delays can hinder the quick detection and response needed to prevent service disruptions and mitigate further impact.

To avoid such delays, an exciting question that arises in the context of network traffic management is whether we can bring ML directly to the data-plane. Traditionally, the data-plane has been responsible for forwarding packets based on pre-defined rules established by the control-plane. However, as networks evolve and real-time decisions become more critical, there is a growing interest in making the data-plane more intelligent and capable of handling certain traffic management tasks autonomously. This would reduce the frequent communication between the data and control-planes, potentially mitigating the delay issue faced by SDN architectures.

Programmable data-planes offer a transformative opportunity to bring intelligence di-

rectly to the data-plane, achieving learning-based traffic management at line-speed. Traditionally, the data-plane's role has been limited to simple packet forwarding based on static rules set by the control-plane. However, with the advent of programmability, data-planes can now perform more sophisticated tasks, including data aggregation, key-value store, and consensus protocols [11]. This shift reduces the reliance on the control-plane for every traffic management decision, thereby alleviating the communication overhead between the two planes. By incorporating ML models directly into the data-plane, networks can make local decisions in real time, such as detecting congestion [12] or identifying traffic anomalies [13], paving the way for more responsive, efficient, and scalable traffic management solutions to meet the growing complexity of modern networks.

1.2 Research Objectives

The scope of this thesis is to introduce ML into programmable data-planes for network traffic management. The key advantage of performing ML model inference directly on the data-plane is that it enables intelligent, real-time traffic analysis at line-speed (i.e., without reducing network forwarding speed) using data-driven models instead of predefined protocols [14]. This capability, referred to as intelligent data-plane (IDP), has the potential to transform various aspects of network design and management. The research objectives are outlined as follows:

Objective 1: Develop a ML-Based Management Framework for Real-Time Intelligent Decision-Making. This objective focuses on enabling low-latency, intelligent management decisions by shifting traffic analysis from the remote controller to the data-plane. By doing so, it eliminates the transmission delays associated with communication between the control-plane and the data-plane. The goal is to implement ML models that can perform real-time traffic analysis and decision-making directly within the data-plane.

Objective 2: Integrate ML Inference within the Data-Plane. Given the computational and memory constraints of data-planes, this objective aims to design techniques that enable ML inference despite these limitations. This includes addressing challenges like the absence of floating-point operations and the limited processing capabilities inherent to data-planes. The focus is on integrating ML models in a way that allows for efficient performance at line-speed, supporting tasks such as traffic analysis and anomaly detection without requiring hardware modifications.

Objective 3: Enable Multi-Phase Decision-Making within the Data-Plane. In many cases, different stages of a network flow carry varying amounts of information, making it critical to make decisions as early as possible. This objective aims to develop a

system that supports decision-making at multiple phases of a flow, balancing between early-stage decision-making and accuracy. The approach will incorporate mechanisms to adjust decision-making based on confidence levels, ensuring timely responses while minimizing resource usage.

Objective 4: Optimize Resource Utilization for Multiple Network Management Tasks. When managing multiple tasks simultaneously, deploying separate ML models for each task can lead to excessive resource consumption in the data-plane. This objective seeks to design a mechanism that enables a single ML model to manage multiple tasks efficiently, thereby conserving resources and optimizing data-plane performance.

Objective 5: Address Concept Drift. Concept drift refers to the phenomenon where the statistical properties of network traffic evolve over time, often due to new applications, emerging attack patterns, or changes in user behavior. Such drift can lead to a mismatch between the data seen during training and the current traffic, causing offline-trained models to lose effectiveness. This objective aims to design a mechanism that triggers ML model retraining when changes in traffic patterns are detected in the data-plane. Furthermore, it aims to identify and collect drifting samples for future retraining, while preserving high classification accuracy for in-distribution traffic.

1.3 Contributions

- First, we designed and implemented a general data-plane ML management toolbox called INQ-MLT, which enables the deployment of complex ML models within the data-plane. The toolbox integrates feature extraction, quantization-aware model training, automated P4 code generation, and line-rate model execution into a unified pipeline. By leveraging quantization techniques, INQ-MLT converts ML model parameters from floating-point representations to low-precision fixed integers, allowing them to be processed efficiently on programmable devices. The adopted quantization technique addresses the computational and memory constraints of data-plane devices, enabling line-speed inference and decision-making on network traffic. The proposed quantization-aware training process minimizes the loss of model accuracy during the transformation to integer operations, maintaining near-floating-point performance. The feasibility of INQ-MLT is demonstrated through multiple use cases, including anomaly detection and multi-label traffic classification, achieving a significant reduction in the traffic management decision-making process. This unified toolbox provides the foundation for all subsequent extensions. This work led to the publication of a conference paper [15], that was extended to a published journal paper [16].

- Second, building on this toolbox, to enable multi-phase decision-making within the data-plane, we developed a two-stage confidence-based intrusion detection system that improves the speed and accuracy of malicious flow detection directly in the data-plane. The system employs two convolutional neural networks (CNNs) trained with a customized transfer learning scheme: an early-stage CNN that processes the first few packets of a flow for rapid detection, and a later-stage CNN that refines the decision if the initial classification lacks sufficient confidence. This two-stage approach enables the system to make faster, more accurate decisions by balancing early detection with robust certainty analysis. A post-hoc calibration method is used to further improve the reliability of the predictions, ensuring that confidence scores align with prediction accuracy. This system is designed to be fully integrated into the P4-based programmable data-plane, ensuring efficient resource utilization while achieving low-latency intrusion detection. This work led to the publication of a conference paper [17].

- Third, to overcome the limitations of previous IDP designs that rely on independent models for independent tasks, we designed MUTA, an in-network multi-task learning (MTL) [18] solution. MUTA enables a multi-task neural network to perform multiple related tasks concurrently by sharing feature representations across tasks. This approach is both resource-efficient and more accurate than single-task models. Resource efficiency is achieved by sharing feature representations among related tasks, thereby eliminating redundant resource usage. Moreover, MUTA enhances accuracy in scenarios where specific tasks lack sufficient labeled data by leveraging knowledge from related tasks through shared model parameters. To accommodate the resource constraints of data-plane devices, we split the neural network model layer by layer and deploy layers across multiple switches in a distributed manner. The proposed network-wide deployment strategy ensures that the provided MTL service will cover the entire multi-path network, with the ability to adjust the trade-off between switch resource consumption and latency. This work led to the publication of a conference paper [19], that was extended to a published journal paper [20].

- Finally, the deployed ML model may become less effective due to concept drift, which refers to the phenomenon where the statistical properties of network traffic change over time—such as through the emergence of new applications, evolving attack patterns, or shifts in user behavior. As a result, the relationship between input features and predicted outcomes learned during training may no longer hold in the current environment. This mismatch between the training data and live traffic leads to a decline in the model's prediction accuracy and overall performance. Moreover, most

existing solutions focus primarily on classifying in-distribution traffic, which refers to traffic that matches previously observed classes, while neglecting the detection of out-of-distribution traffic that arises as a result of concept drift. To address this limitation, we first proposed an unsupervised drift detection method designed for the data-plane. This method continuously monitors the distribution of flow-level features using dual sliding windows and identifies concept drift by measuring the divergence between these distributions and comparing it to a predefined threshold, all without relying on labeled data. In addition, we developed IDAC, an in-network drift-aware traffic classification framework that supports both accurate classification of in-distribution traffic and detection of drifting samples. IDAC employs a triplet network to learn an encoder that maps input traffic features into a latent space where instances from the same class are embedded close together, forming compact clusters. Drifting samples are identified by their increased distance from known class centroids in this space. This line of research has led to the publication of both a workshop paper [21] and a conference paper [22].

The rest of this thesis proposal is organized as follows. Chapter 2 contains a background study on different subjects related to this work, especially SDN, programmable data-planes and ML. Chapter 3 presents a novel ML-based toolbox for programmable data-plane, which facilitates the execution of ML models directly within the data-plane. Chapter 4 introduces a novel intrusion detection system in the programmable data-plane, capable of detecting malicious flow at two stages of a flow. Chapter 5 introduces MUTA, a novel in-network multi-task learning framework that enables concurrent inference of multiple tasks in the data-plane. Chapter 6 introduces an unsupervised drift detection method tailored for the data-plane, enabling the timely triggering and notification of model updates in response to detected drift. Chapter 7 presents IDAC, an in-network drift-aware traffic classification framework that not only supports accurate classification of in-distribution traffic but also enables detection of drifting samples directly in the data-plane. Finally, Chapter 8 concludes this thesis and presents potential future research directions.

Chapter 2

Background

The purpose of this chapter is to provide a background of the technologies used in this thesis proposal and summarize related work to present state-of-the-art research and study. Several key technologies leveraged in this thesis proposal for intelligent data-plane design are introduced: software-defined networking (SDN), programmable data-plane and machine learning (ML).

2.1 Software Defined Networking

As networks grow in size and complexity, traditional network devices face limitations in providing efficient management and monitoring capabilities for network administrators. To overcome these challenges, SDN has emerged as a transformative approach to network architecture. In conventional network devices, such as switches and routers, the control-plane (which manages decision-making) and the data-plane (which handles traffic forwarding) are tightly coupled and hard-coded within each device. While device vendors offer interfaces for configuration, these options are often restrictive, providing limited flexibility for network administrators. As networks expand or new services arise, it becomes increasingly difficult to modify and apply configurations to each individual device across the network.

SDN addresses these challenges by decoupling the control-plane from the data-plane and centralizing the control functions in an external controller [5]. Figure 2.1 shows an overview of a generic SDN architecture. This separation enhances network flexibility and programmability by allowing administrators to dynamically configure network functions and services. The centralized controller provides a global view of the network, which significantly improves management and monitoring capabilities. Switches in the network

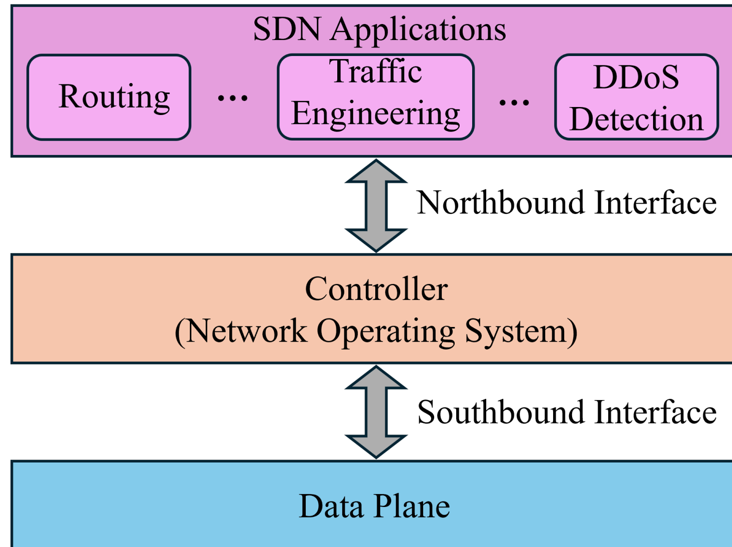


Figure 2.1: Main components in an SDN architecture.

become simple forwarding devices that are configured by the controller through a standardized API, known as the southbound interface. As a result, network flexibility is further enhanced by making the control-plane programmable. The controller functions as a network operating system, offering an abstract global view of the network to various SDN applications that run network management algorithms. Consequently, implementing new network behavior is simplified to updating the SDN application within the controller [23].

An SDN architecture typically consists of a controller and multiple network devices. The controller is responsible for centralized decision-making regarding traffic forwarding, while the data-plane, located in physical network devices, executes these decisions. The southbound interface (e.g., OpenFlow [24]) specifies both the logical structure of the packet-processing pipeline and the protocol used to configure it. The process is as follows: when traffic arrives at a network device, the data-plane checks the traffic against flow rules in a lookup table. If a match is found, the traffic is forwarded accordingly. If no match is found, the device sends the traffic information to the controller. The controller, leveraging its global view of the network, makes a decision and updates the flow tables on the device by installing new rules. These rules enable the data-plane to handle future traffic autonomously based on the updated policies.

The key advantage of this approach is that the controller can continuously collect information from all devices, allowing it to dynamically adjust traffic-forwarding policies. This leads to more flexible, agile, and efficient network operations, especially as the network scales. By simplifying network management and introducing programmability, SDN significantly enhances the ability to adapt to changing network demands.

2.2 Programmable Data-Planes

While SDN offers flexibility by allowing control and management of network traffic through the control-plane, its primary use has been limited to network control and application deployment. The data-plane, however, remains rigid and hardcoded, which becomes problematic when adapting to new protocols or emerging demands. SDN is fundamentally built around the Match-Action abstraction, which uses a predefined set of match fields and corresponding forwarding actions. To accommodate a wider range of protocols, the OpenFlow specification expanded from 12 header fields to 41 within five years after the introduction of OpenFlow 1.0 in 2009 [25]. Despite these extensions, the limitations persist—most hardware and software switches can only support a finite number of Match-Action pairs for standard network protocols.

Moreover, the design of network functions is tightly coupled to the capabilities of the switching hardware. Adding new functionalities to switch ASICs (Application-Specific Integrated Circuits) often requires coordination between network operators and equipment vendors to define requirements and develop the necessary support in hardware. This development cycle, from initial proposal to the commercial deployment of new protocols like VxLAN, can take years. Such long timelines slow down the ability of networks to evolve and meet the increasing demands of new services and use cases.

To address these challenges, the concept of the programmable data-plane was introduced to enhance the flexibility of data-plane functionality. It focuses on enabling more advanced programmability and customizable packet processing. In traditional and SDN-based switches, the data-plane handles switching and forwarding based on lookup tables with predefined Match-Action rules. The programmable data-plane introduces Reconfigurable Match Tables (RMT), allowing these tables to be dynamically reconfigured [26]. The Protocol Independent Switch Architecture (PISA) further expands this concept by offering flexible packet header parsing and packet processing capabilities along the processing pipeline [27]. To define and program this process, the P4 language [25] is employed, which enables network devices such as switches, routers, and network interface cards to be programmable, allowing them to manage packet processing and forwarding directly on the data-plane. The following sections provide further details on these concepts.

2.2.1 P4 Language

Programming Protocol-Independent Packet Processors (P4) [25] is a domain-specific language designed to enable the development of programmable data-planes in networking devices. P4 provides network administrators with a higher level of control and flexibility

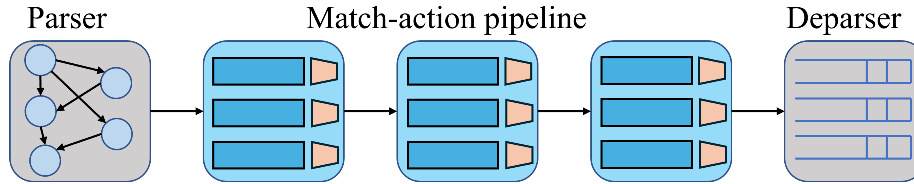


Figure 2.2: Protocol independent switch architecture.

by allowing them to define and customize how network traffic is parsed, processed, and forwarded. The P4 language is built with two main design goals: protocol independence and target independence. Protocol independence requires the network devices should not be tied to specific packet formats. Target independence enables P4 programs to be compiled and run on different hardware or software platforms. P4's structure includes blocks for packet parsing, ingress and egress processing, checksum verification, and packet reconstruction, making it a powerful tool for customizing and optimizing network behavior.

2.2.2 P4 Architectures

A programmable network device architecture defines the high-level structure of programmable network devices, and the interfaces between their major components. Most P4 architectures are derived from a basic abstract architecture PISA (Protocol Independent Switch Architecture) [27]. PISA is composed of three key elements: a parser, a deparser, and a match-action pipeline, as shown in Figure 2.2.

The parser operates as a state machine, extracting a sequence of fields from incoming packets, forming what is known as a Packet Header Vector (PHV). The PHV typically contains fields from standard packet headers (e.g., Ethernet, IP, VLAN, TCP/UDP) as well as intrinsic metadata (e.g., ingress and egress ports), and can also include custom, user-defined headers. The extracted PHV is then processed in a sequence of match-action stages, each using match-action tables. These tables serve as essential components that perform lookups based on a key value (e.g., a field from the packet header) and trigger corresponding actions for packet processing (e.g., algorithm execution and data manipulation). Each match-action stage allows a fixed number of operations, where a key is matched with a table entry, and a defined action is applied to the packet. After processing, the deparser reassembles the PHV fields with the original packet payload before the packet is forwarded. The parser, match-action pipeline, and deparser can be programmed to implement customized protocols.

While PISA supports simple operations like addition, shift, and bit-wise logic, it does not allow complex instructions such as floating-point arithmetic, matrix multiplication,

or iterative loops. These restrictions stem from the requirement to maintain deterministic, line-rate processing where each packet must be handled in a fixed number of cycles. Supporting heavy computations or unbounded control flow could break these latency guarantees, so PISA pipelines are limited to lightweight, highly parallelizable tasks, while more computationally intensive operations are typically offloaded to external processors or the control plane.

Various architectures are built on top of the PISA model, including both open-source and commercial solutions like the Portable Switch Architecture (PSA) [28], Programmable NIC Architecture (PNA) [29], v1model [30], and Tofino Native Architecture (TNA) [31].

2.2.3 P4 Targets

The P4 language is designed to support a wide range of packet processing targets, which can be broadly categorized into hardware and software targets.

P4 can be used to program a variety of hardware targets, such as switch-ASICs and SmartNICs. These devices can be customized to handle different traffic processing tasks. Notable examples of P4-programmable hardware switches include Intel Tofino and NVIDIA Spectrum. Programmable switch-ASICs introduce programmability into the switch pipeline while maintaining high performance, characterized by high throughput and low latency. Current switch-ASICs are capable of exceeding 50 Tbps throughput, processing tens of billions of packets per second with sub-microsecond latency. However, these hardware switches are resource constrained, typically providing only tens of megabytes of memory and a limited number of processing stages [32]; for instance, the Intel Tofino switch features twelve processing stages and Mb-scale memory [33].

A software switch is a network switch implemented as an application running on a standard CPU. Software switches, which are compatible with both x86 and ARM architectures, do not require specialized hardware. The most widely used P4 software switch in academia is Behavioral Model version 2 (BMv2) [34], which supports both the Simple Switch target based on the v1model architecture and the PSA switch based on the PSA architecture. However, software switches typically offer lower performance than hardware-based solutions.

The P4 model defines extern functions and objects accessible from the P4 code. Extern objects (e.g., meters, registers, and counters) are used to save the P4 program states, while extern functions can be used to implement complex operations (e.g., hash functions or encryption).

2.2.4 Control-Plane and Data-Plane Interaction

The control-plane and data-plane interact continuously during network operation to ensure real-time configuration and monitoring. In P4-enabled environments, the control-plane installs forwarding rules, retrieves statistics, and updates table entries through well-defined APIs such as P4Runtime [35], a standardized, gRPC-based interface that supports dynamic configuration without recompiling or halting the data-plane program.

This interaction typically follows a closed-loop process: the data-plane collects network telemetry (e.g., traffic counters, flow statistics, in-band telemetry) and forwards it to the control-plane, where global optimization algorithms or ML models compute new policies. The updated forwarding rules or actions are then pushed back to the data-plane for enforcement. Such a separation improves network flexibility and programmability, allowing rapid adaptation to changing traffic patterns or service requirements.

However, frequent communication between the planes introduces significant latency, particularly when ML inference resides exclusively in the control-plane. Time-sensitive applications, such as DDoS mitigation, congestion avoidance, or real-time QoS control, often require millisecond-level responses that centralized inference cannot always provide. Moreover, transmitting raw traffic features to remote servers for analysis imposes additional bandwidth and processing overhead. These limitations strongly motivate pushing ML inference closer to the data-plane itself, reducing round-trip delays while preserving centralized control for high-level policy decisions.

2.3 Machine Learning

ML plays an important role in optimizing network traffic management by enabling dynamic and data-driven decision-making. The selection of the appropriate ML model is crucial because it directly influences the performance and the complexity of the deployment. Broadly, ML models can be categorized into three main types: supervised learning, unsupervised learning, and reinforcement learning. Each of these paradigms serves different purposes and is chosen based on the network's goals, available data, and the expected balance between performance and computational overhead.

2.3.1 Supervised Learning

Supervised learning is the most widely used ML paradigm in network traffic management when labeled datasets are available. This approach involves training a model on a dataset

where both the input data and the corresponding correct outputs (labels) are known. The objective is for the model to learn the mapping between inputs and outputs so that it can predict the correct labels for unseen data.

Common supervised learning algorithms include decision trees (DTs), random forests (RFs), support vector machines (SVMs), and neural networks (NNs). These models have been effectively applied in traffic classification [36], heavy flow detection [37], and intrusion detection [13] in network environments. For example, classifiers such as RFs or SVMs are trained on traffic flows labeled with specific applications or user behaviors, allowing for real-time classification of network traffic [38].

The primary advantage of supervised learning is its ability to provide highly accurate predictions when sufficient labeled data is available. However, the process of labeling data is often time-consuming and labor-intensive, especially in dynamic network environments where new types of traffic or threats emerge regularly. Additionally, supervised models may struggle with generalization if trained on biased or incomplete datasets.

Deep Learning (DL) models, including Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, are categorized under supervised learning models as they rely on labeled datasets for training. These models demonstrate a high degree of proficiency in handling complex and high-dimensional data, offering advanced capabilities in various applications. However, their lack of interpretability, known as the "black box" problem, makes it difficult to understand the underlying mechanism of the model. Furthermore, the complexity of these networks can introduce additional overhead during deployment, posing challenges for practical implementation [39].

Multilayer Perceptron

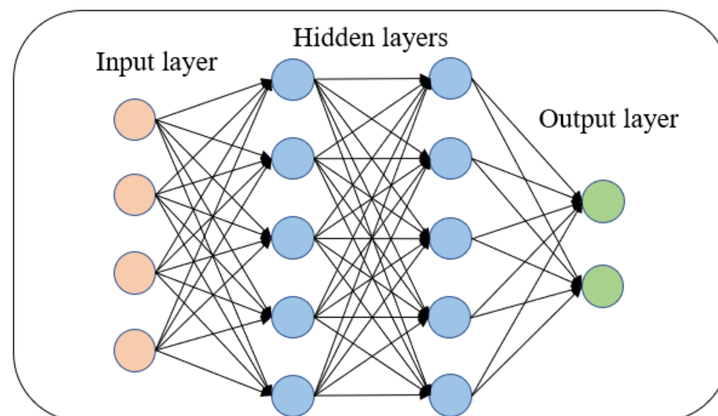


Figure 2.3: Structure of an MLP with 2 hidden layers.

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot) : R^m \rightarrow R^n$ by training on a dataset, where m is the number of dimensions for input, and n is the number of dimensions for output. An MLP with two hidden layers is shown in Figure 2.3. MLP can be considered as a subset of NNs, and only NNs with a sufficient number of hidden layers (usually more than one) can be regarded as 'deep' models, i.e., deep neural network (DNN).

For a given input vector $\mathbf{x}$, a standard operation of a MLP layer is given as follows:

$$\mathbf{y} = \sigma(\mathbf{W}\mathbf{x} + b) \tag{2.1}$$

Where $\mathbf{W}$ and b denotes the weights and bias. $\mathbf{y}$ is the output of the perceptron and $\sigma(\cdot)$ is the activation function, which is proposed to improve the nonlinearity of the model. Here we list some commonly used activation function [40]:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

Rectified Linear Unit (ReLU) [41],

$$\text{ReLU}(x) = \max(x, 0) \tag{2.3}$$

Additionally, the softmax function is always employed in the last layer to output the probabilities for a multi-label classification problem.

$$\text{softmax}\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=0}^{n-1} e^{x_j}} \tag{2.4}$$

where n is the number of dimensions for output.

Transfer Learning

Transfer learning is a ML technique where knowledge gained from solving one problem is applied to a different but related problem. The idea behind transfer learning is to leverage the patterns, features, or representations learned from a pre-trained model on a large dataset to improve performance on a smaller or less well-defined task. In traditional ML, models are typically trained from scratch, requiring a large amount of labeled data to learn the necessary features. However, transfer learning allows for the reuse of a pre-trained model's knowledge, reducing the need for extensive datasets and computational resources. This approach is especially beneficial in cases where labeled data is scarce or expensive

to obtain, such as in medical diagnosis or natural language processing. A typical transfer learning process involves fine-tuning a pre-trained model by adjusting its parameters to suit the specific characteristics of the new task. For instance, a model trained on image classification can be adapted to identify new objects with fewer training examples by leveraging the learned visual features like edges or shapes [42].

2.3.2 Unsupervised Learning

Unsupervised learning models are employed when there are no labels in the dataset. These models operate by discovering patterns or structures in the data without any prior knowledge of labels. In network traffic management, unsupervised learning can facilitate tasks such as anomaly detection, where abnormal traffic patterns are detected without the need for explicit labeling of normal or abnormal behavior.

One common unsupervised learning technique is clustering, such as the k-means algorithm, which groups similar data points into clusters. In the context of network traffic, this method can be used to group traffic flows based on similarity, helping to identify new or previously unknown types of traffic. Dimensionality reduction techniques, such as Principal Component Analysis (PCA), are also used to visualize and simplify high-dimensional network data, making it easier to identify patterns [43].

2.3.3 Reinforcement Learning

Different from the aforementioned two types of ML methods, reinforcement learning (RL) can not only classify the traffic but also take reactions to the analysis decisions [9]. It is based on the idea of learning from the environment and making decisions/actions based on the rewards. In this case, it is possible to adapt and react dynamically to changing traffic conditions and new network activities.

2.4 Traditional ML-based Network Management Schemes

In recent years, ML has emerged as a powerful tool within communication technologies and network services due to its exceptional learning power. ML offers substantial potential in network monitoring and management, presenting advantages over traditional manual analysis and rule-based traffic management systems. For example, unlike static rule-based systems, ML algorithms can uncover hidden or complex patterns within vast amounts of

traffic data, enabling fine-grained traffic analysis. We summarize several network management tasks below:

- **Intrusion Detection**: ML-based traffic analysis solutions enhance network security by identifying and mitigating cyber threats that could compromise network availability, grant unauthorized access, or lead to the misuse of network resources. For instance, ML models are highly effective in detecting malicious activities, such as Distributed Denial of Service (DDoS) attacks [44]. Compared to traditional rule-based intrusion detection systems (IDS), ML-based solutions offer superior performance, including the ability to identify several variations of known attacks, thereby providing a more adaptive and robust defense [45].

- **Traffic Classification**: ML models are trained to build sophisticated traffic profiles by recognizing patterns in network behavior (e.g., flow features). This enables the identification of different traffic types (e.g., web, VoIP, gaming), applications (e.g., Skype, Youtube and Netflix), and the classification of connected devices across the network (e.g., IoT device identification [46]). The granular understanding of traffic types supports the optimization of network resource allocation and prioritization of critical services.

- **Heavy Flow Detection**: Also known as elephant flow detection. ML-based approaches have proven useful in identifying "elephant" (high-volume, long-lasting) and "mice" (low-volume, short-lived) flows, distinguishing between these types of traffic allows for more efficient load balancing and resource allocation, ensuring that high-priority or high-bandwidth flows receive appropriate attention to maintain optimal performance [37].

- **QoS and QoE Management**: ML-based analysis can proactively monitor and predict KPIs such as Quality of Service (QoS), latency, and jitter to identify potential congestion points in the network. By leveraging real-time predictions, these models optimize routing paths, leading to improved user experiences by reducing congestion, minimizing packet loss, and maximizing throughput. For the Quality of Experience (QoE) management, content providers like YouTube and Netflix must continuously adjust bandwidth provisioning and storage allocation, balancing operational costs with the QoS perceived by end users. For instance, ML can predict QoE metrics such as startup delay, bit-rate, resolution, and stalling [2]. QoE has become the de facto metric for guiding resource allocation decisions to ensure optimal service delivery [47].

With these advancements, many ML-based solutions have achieved promising results in terms of accuracy and efficiency in handling complex network management tasks. These improvements bring significant benefits for network administrators. For example, ML-based traffic classification enables differentiated QoS provisions, allowing for better service quality based on traffic type, while malware detection models can defend networks against emerging cyber threats promptly [48].

However, despite the potential of ML in network management, significant challenges remain in the practical deployment of these schemes. Most current ML-based schemes require high-performance servers equipped with substantial computational resources to support the intensive processing demands of ML models. More complex models, such as deep neural networks, may even require specialized hardware like Graphics Processing Units (GPUs) to accelerate computation [49]. These servers are typically deployed in the remote cloud or at the network edge. Traffic often needs to be redirected to specialized servers in the control plane for analysis, resulting in increased processing latency. The process of transmitting traffic to centralized or cloud-based ML servers can create bottlenecks for time-sensitive applications. Additionally, the added latency negatively impacts throughput, degrading the overall performance of the network.

While ML has unlocked new possibilities for network management, there is a need for research into more efficient ML deployment strategies. Solutions that bring ML computations closer to the network, such as deploying lightweight models directly on programmable network devices, could reduce latency and improve throughput without sacrificing the intelligence offered by ML-based analysis. In the following section, we will explore related work that integrates ML-based analysis within programmable data-planes.

2.5 Existing Solution for Intelligent Data-Planes

The advent of programmable data-planes and the development of programmable switching ASICs have given rise to a new research area known as in-network computing [11]. This approach integrates computing capabilities directly within the data-plane devices. Unlike traditional server-based ML schemes, where both training and inference are performed on remote servers or in the cloud, intelligent data-planes (IDPs) enable offloading the ML inference process to network devices. While the model training remains in the control-plane, the inference process, responsible for real-time decision-making, is executed directly within the network devices.

Although server-based ML deployments offer substantial computational resources, they introduce bottlenecks in network traffic analysis services due to the need for network data

to be sent from devices to centralized servers or cloud environments for processing. This results in additional transmission latency and overhead. In time-critical applications, such latency can prevent rapid security analysis and threat mitigation. In contrast, IDPs provide an opportunity to reduce this latency by leveraging programmable data-planes and P4 programs. They enable network devices to not only process packets but also execute ML-based decision-making in real time, performing intelligent packet manipulation before forwarding the packets to their next destination.

Previous research has explored the design and implementation of IDPs using various ML models. Tree-based models have been a popular choice due to their relatively low complexity and ease of integration with match/action tables. Additionally, the implementation of neural network-based models within IDPs has also been investigated.

2.5.1 Tree-based Solutions

Decision Trees (DTs) are widely used supervised learning algorithms with a tree-like structure, comprising a root node, internal nodes, and leaf nodes. The leaf nodes represent decision outcomes, while internal nodes (branches) contain decision rules. As data flows through a DT, it starts at the root and traverses the branches based on these rules, ultimately reaching a leaf node that provides the classification result.

The tree structure of a DT maps naturally to the operational logic of programmable network switches, where packet processing pipelines can incorporate a tree-like flow. During inference, the DT performs comparisons at each internal node, guiding the data to the appropriate branch and ultimately yielding the output. Three main strategies have been proposed for implementing DTs in the data-plane, as described below.

Hard-coded Approach

The approach described in [50] hard-codes the DT structure into P4 using *if-else* conditional statements, where model parameters are directly embedded in the code. While this method is straightforward, it has significant limitations: updates to model parameters require changes to the P4 code, followed by recompilation. Moreover, excessive conditional statements can exhaust the logic resources (e.g., stages) available in the data-plane. pHeavy [51] is a specialized heavy flow detection scheme based on this approach on the data-plane.

Depth-based Approach

The depth-based approach [52,53] offers more flexibility by leveraging match-action (M/A) tables to represent the DT structure. In this method, each stage of the pipeline corresponds to a layer of the tree, with table keys representing nodes at that layer. The action parameters store information such as feature values and thresholds. After a match is found, the action compares the feature value with the threshold and determines the next node based on the result. While this method introduces some sequential dependencies between M/A tables, which can limit scalability, it enables greater flexibility compared to the hard-coded approach.

Encode-based Approach

The encode-based approach [1, 54] diverges from the hierarchical structure of the DT, encoding each feature according to the split values at the tree's branches. Each branch is viewed as a segment of the feature space, and these segments are labeled with unique codes. A mapping table then associates these codes with the corresponding leaf nodes. This approach reduces the number of stages required by eliminating sequential dependencies between feature tables, as stages can be shared among different features. Planter [55] extends this idea, and provides an approach that maps random forests (RFs) by overlapping feature encoding over trees within match-action tables. To address the table entry explosion problem under ternary matching, NetBeacon [14] introduces a range marking mechanism to ensure each leaf node only consumes a single ternary entry in the model table.

2.5.2 Neural Network-based Solutions

Binary Neural Networks (BNNs) are a type of artificial neural network (ANN) designed to operate on resource-constrained devices by using binary weights and activations [56]. BNNs use bitwise operations (XNOR and PopCount) to efficiently perform matrix multiplications [57]. Each BNN neuron's weight is stored as a bit string in stateful registers, which can be dynamically read and written. During inference, bitwise XNOR operations are applied between the input bit string and each neuron's weight. The activation function is then computed by determining the Hamming weight of the XNOR output.

In addition to BNNs, more advanced approaches have been proposed for deploying neural networks on programmable network devices. For example, Taurus [58] introduces custom hardware based on the MapReduce abstraction to enable deep neural network processing. Similarly, IOI [59] uses a novel transceiver module to perform neural network

inference on programmable switches by handling linear operations like matrix multiplication in the optical domain. However, these approaches are not compatible with commodity switch ASICs and require specialized hardware modifications.

Despite showing feasibility, prior work faces critical limitations. BNNs trade accuracy for efficiency, making them unsuitable for complex or multi-phase decision-making tasks. Taurus and IOI deliver greater computational power but rely on hardware modifications, restricting their practicality on commodity switches. Furthermore, most existing solutions focus on single-task inference, lacking support for concurrent learning across multiple network management objectives. In contrast, our work introduces a high-precision neural network framework that uses quantization-aware training to maintain accuracy while remaining fully compatible with existing programmable data-plane targets. By enabling efficient in-network inference without requiring hardware changes, our approach addresses the accuracy and scalability limitations of prior solutions.

2.6 Multi-Task Learning

Multi-task learning (MTL) is a ML paradigm in which a single model is trained to perform multiple tasks simultaneously under the assumption that related tasks can benefit from shared knowledge and improve overall learning performance. This approach has been successfully applied in various domains, such as natural language processing [60], computer vision [61], and autonomous driving [62].

A widely used strategy in MTL is known as hard parameter sharing, where part of the model parameters, typically the early layers, are shared among all tasks while the later layers remain task-specific [63]. Compared with training separate models for each task, this parameter-sharing strategy greatly reduces the memory footprint because redundant parameters are eliminated and improves computational efficiency by avoiding repeated feature extraction in the shared layers. Beyond hard sharing, soft parameter sharing techniques have been introduced, in which each task maintains its own set of parameters but task relatedness is encouraged through constraints or regularization terms. This approach provides greater flexibility by allowing partial knowledge transfer while preserving the unique characteristics of individual tasks.

2.7 Conclusion

In this chapter, we examined key concepts in modern networking, focusing on SDN, programmable data-planes, and the integration of ML for enhanced network management. SDN introduces flexibility by decoupling the control and data-planes, allowing centralized control and dynamic traffic management. However, the static nature of the data-plane in traditional SDN systems limits adaptability. To address this, programmable data-planes, such as those enabled by the PISA and the P4 programming language, allow dynamic reconfiguration of packet processing. This provides better flexibility in handling evolving protocols and services. We also discussed the role of ML in optimizing network tasks like traffic classification and intrusion detection, with models such as DTs and neural networks offering data-driven decisions. Furthermore, the concept of in-network computing was introduced, where ML models are implemented directly within the programmable data-plane to perform real-time traffic analysis, minimizing latency by processing data at the data-plane level. Overall, the combination of SDN, programmable data-planes, and ML creates a robust framework for modern network management, offering improved scalability, flexibility, and intelligence. In the following chapter, we design an ML-based toolbox for programmable data-planes to further advance this goal.

Chapter 3

A Machine Learning-based Toolbox for P4 Programmable Data-Planes

3.1 Conceptual Overview

This chapter introduces a machine learning (ML) toolbox for P4 programmable data-planes, forming the foundation upon which the remaining contributions of this thesis are built. Modern programmable switches provide the capability to process packets at line rate while supporting user defined processing logic. Embedding ML inference into these devices, however, requires addressing constraints on memory, computation, and control plane interaction. The toolbox presented here consolidates a range of essential components, including feature extraction, training, quantization, model deployment, and in-network inference, into a unified processing pipeline that enables real time, data driven decision making directly within the data-plane. Instead of designing separate solutions for each network intelligence task, the toolbox establishes a common set of mechanisms that can be instantiated or extended depending on the specific requirements of the application.

While this chapter focuses on the design and implementation of the core toolbox, the same processing pipeline also serves as the backbone for several distinct research directions developed in the following chapters. Subsequent contributions extend it in different directions to address specific challenges in network intelligence, ranging from latency sensitive security analytics to adaptive learning under changing traffic conditions. Each chapter builds on the underlying primitives of packet feature extraction, quantized ML model execution, and decision making in the data-plane, while introducing additional modules or control logic tailored to its specific objectives. In this way, the toolbox provides a consistent substrate for early stage intrusion detection, multi-task learning across related

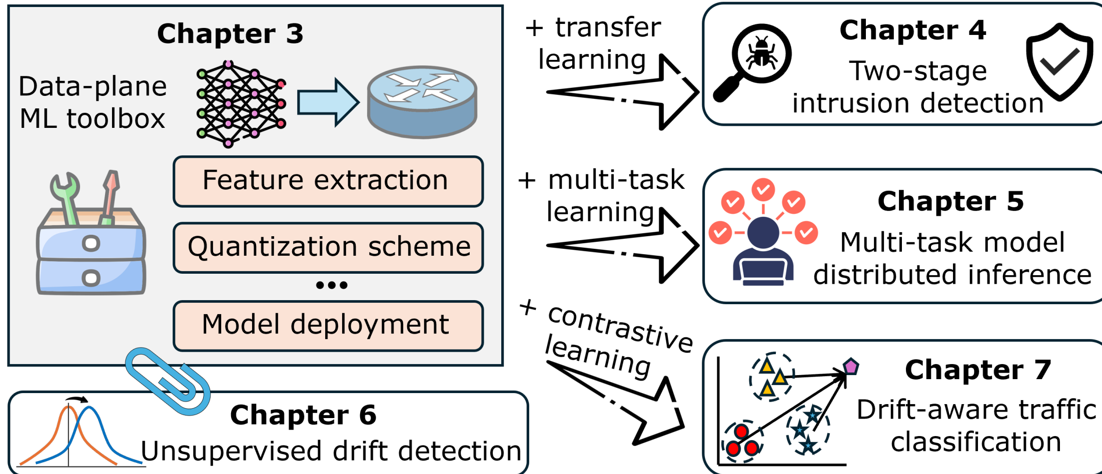


Figure 3.1: The data-plane ML toolbox and its extensions across the thesis.

network functions, unsupervised monitoring of traffic drift, and drift-aware classification under evolving network conditions. More concretely:

- Chapter 4 augments the toolbox with a two-stage decision-making mechanism for intrusion detection, combining early-phase inference with confidence-based refinement to achieve both speed and reliability.
- Chapter 5 builds on the unified toolbox to introduce multi-task learning, enabling concurrent inference across multiple network management tasks while optimizing resource usage across distributed switches.
- Chapter 6 integrates an unsupervised drift-monitoring module into the toolbox, continuously tracking distribution shifts in traffic features and triggering model update workflows when concept drift is detected.
- Chapter 7 extends the toolbox with drift-aware classification, combining classification accuracy with the ability to flag previously unseen or out-of-distribution traffic patterns.

Figure 3.1 depicts how the unified toolbox underlies these specialized contributions. The value of the design lies in its stable interfaces and reusable processing patterns: feature extraction logic, neural network quantization schemes, and template match-action tables for layer execution. These common interfaces allow later chapters to focus on algorithmic choices and policy logic while relying on a single, coherent pipeline for in-network execution across Chapters 4–7.

3.2 Introduction

The complexity of today's network management operations can be largely attributed to the heterogeneity and scale of the managed devices and served traffic. Network management operations include provisioning for strict performance and security guarantees, traffic monitoring and analysis [4], resource configuration [64], and fault management [65]. ML has demonstrated significant benefits in performing these management tasks [7]. These benefits can be attributed to the ability of ML models to analyze a large set of data at run-time to learn the current characteristics of the hosted traffic or to find optimal network and service configurations.

In current management architectures, these management tasks are mostly performed following the software-defined network (SDN) paradigm [66,67]. Due to the high computational and storage demands of ML models, their execution has been mostly limited to the control-plane. The latter then must continuously interact with the forwarding devices to collect monitoring data, train the model, and then communicate back the updated operational decisions. The result is a time-consuming control-loop that leads to non-negligible delays.

Meanwhile, the introduction of programmable data-plane languages such as P4 [25,68] has provided an efficient means for the control-plane to customize and adapt the forwarding behavior of packets as carried out by the data-plane. Commonly used P4 architectures such as the Protocol-Independent Switch Architecture (PISA) [27], the v1model [30] and the Portable Switch Architecture (PSA) [28] use a pipeline of match-action tables to modify the forwarding behaviour of flows at a very fine-grained level. Switches running P4 are also equipped with registers that can store per-flow states. Moreover, various protocols, such as P4-runtime [35,69] have been developed to allow data-plane reconfiguration at runtime.

The increasing popularity of Artificial Intelligence (AI) in various networking management tasks and the gradual adoption of P4 have paved the way for a new vision of intelligent data-planes (IDPs). IDPs enable intelligent traffic analysis at line-speed by deploying ML models on programmable devices [70]. Clearly, offloading ML models to the data-plane will eliminate the needed controller and data-plane communication resulting in a significant decrease in the decision-making latency. IDPs can, hence, effectively improve the management decision quality through real-time processing.

In recent years, several approaches have investigated the feasibility of realizing IDPs with focus on traffic classification [50], DDoS detection [17], heavy flow detection [51], and load balancing [71]. Several efforts have explored offloading of ML tools such as decision trees (DTs) [50, 51], random forests (RFs) [52], binary decision trees (BDTs) [48], K-

means [72], support vector machines (SVMs) [72], and binary neural networks (BNNs) [57, 73, 74] to the data-plane. In parallel, some other solutions introduced hardware modifications to the switch [58, 59] to host deep-learning models.

Unfortunately, implementing ML models that require increased computational capabilities at the data-plane without modifying the switch hardware remains an unaddressed challenge. Current programmable data-planes, such as those using P4, impose several computational restrictions. Particularly, they do not support floating-point operations. This limitation prohibits the direct implementation of neural deep learning based ML models. Even though existing work [75,76] has been proposed to use lookup-tables for floating-point arithmetic on switches, this can rapidly result in resource exhaustion when implementing large ML models. Similarly, the inability to perform division operations directly using P4 makes it difficult to store flow statistics (e.g., average packet size or its variance) [52].

In this chapter, we propose a novel in-network quantized ML toolbox (INQ-MLT). The toolbox is comprised of two components. The first is a quantization-aware ML model training module that executes at the control-plane. The second are quantized ML models that are integrated within the IDP pipeline. INQ-MLT addresses the aforementioned limitations by utilizing quantization techniques and control-data-plane collaboration while achieving efficient decision-making at line-speed. To this end, the main contributions of this chapter can be summarized as follows.

- We propose a novel toolbox, INQ-MLT, that facilitates the training of ML (e.g., neural and deep learning) models and simplifies the storage and execution of these ML models within the resource-limited IDPs.
- We develop an efficient quantization scheme that allows ML inference to be carried out using integer-only arithmetic on programmable devices. The model is then integrated and executed within the IDP pipeline while meeting its operational and computational constraints.
- To minimize the loss of accuracy resulting from quantization, we adopt a novel training mechanism, quantization-aware training, that forces the training of the model to compensate for any expected loss.
- We provide a rigorous theoretical analysis of how the precision of model parameters impacts the accuracy of model inference and derive the theoretical upper bound on the misclassification rate for the quantized model.

To evaluate the performance of INQ-MLT, we implement two ML models generated by the proposed toolbox on BMv2 software switches [34] using P4. We validate the toolbox

through the training and testing of a Multi-Layer Perceptron (MLP) for anomaly detection and a 1D-Convolutional Neural Network (CNN) for multi-label traffic classification use cases respectively.

The remainder of this chapter is organized as follows; Section 3.3 discusses related work. Section 3.4 provides an overview of the proposed toolbox while Section 3.5 describes the adopted quantization scheme and explains the quantized ML model execution steps. Section 3.6 discusses the P4-based implementation details. Section 3.7 presents the theoretical analysis of the quantization effect on the accuracy. Performance evaluation results are discussed in Section 3.8. Some considerations regarding the use of the toolbox are provided in Section 3.9. Finally, Section 3.10 concludes this chapter.

3.3 Related Work

In this section, we first review research efforts that have contributed to the development of intelligent data-planes. We then discuss existing methods for different intelligent traffic management functionalities.

3.3.1 Intelligent Data-Planes

Implementing ML models within network devices enables intelligent traffic analysis, with tree-based models often chosen for their low complexity and intuitive mapping to match-action tables. As also introduced in § 2.5.1, three main approaches exist for mapping such models into P4: the *hard-code* scheme, as in MAP4 [50] and pHeavy [51], which uses nested *if-else* chains; the *depth-based* scheme, used by pForest [52] and SwitchTree [53], where each match-action stage corresponds to one decision tree layer; and the *encode-based* scheme, adopted by IIsy [1,72] and extended by Planter [77], which encodes decisions per feature and combines them at the leaf node. To address memory and efficiency constraints, NetBeacon [14] mitigates ternary table entry growth, while Mousika [48] uses binary decision trees and knowledge distillation for compact and efficient deployment.

Implementing Binary Neural Networks (BNNs) on the data-plane has been explored by [73] and [57]. BNN compresses all the weights of a neural network into single bits and only requires simple operations such as XNOR and population count. Zhong et al. [59] propose IOI, a system providing deep neural network (DNN) inference on programmable switches. IOI plugs a novel transceiver module into programmable switches to perform linear operations such as matrix multiplication in the optical domain. Swamy et al. [58]

propose Taurus, a new data-plane architecture for per-packet ML. Taurus aims at implementing DNNs in the data-plane. They extend the PISA with a customized hardware MapReduce block, enabling the execution of DNNs. Swamy et al. [78] also propose Homunculus, a framework that enables network operators to specify their ML requirements in a declarative way and automatically generates efficient data-plane ML pipelines.

In general, the aforementioned schemes are limited to ML models (e.g., DTs) that can be directly mapped to simple rules within the data-plane match-action tables. Most mapping schemes impose structural limits on the maximum tree depth. For example, the *depth-based* mapping scheme used by pForest [52] and SwitchTree [53], uses a match-action stage for each level in the tree. Therefore, the tree depth is restricted by the available number of match-action stage in commercial switch ASICs. Some stages must be allocated for tasks such as calculating flow identifiers, maintaining register indices, updating stateful features, and implementing tree leaves, limiting the maximum tree depth [54]. pForest [52] claims that only relatively shallow trees of depth 4 can be implemented in a Tofino switch. Such tree depth limitation induce performance barriers when dealing with complex inference tasks.

We believe that exploring the applicability of utilizing complex ML models, which have already been utilized by the control-plane, at the data-plane level, can significantly enhance the performance of various network management operations while reducing the control-data plane communication delay. Finally, orthogonal to our presented solution is Taurus [58] and IOI [59] where the ASIC is modified to support DNN inference. In contrast to these approaches, we aim at offloading ML schemes at the data-plane without modifying the hardware/memory structure of current switches.

3.3.2 Intelligent Traffic Management

The complexity of network traffic patterns and the use of encrypted communications are driving the widespread adoption of traffic management based on ML [79]. The ML model customizes itself to the specific network environment by training on the dataset collected for that environment. As the network environment changes, more data is collected, and the model is retrained to reflect these changes, thereby helping to understand relationships between various events that network operators may not be aware of. For example, instead of just matching incoming flows against a statically known set of IP addresses, ML can learn the correlation between fine-grain features (e.g., real-time applications like online meeting often have short inter-arrival time; DDoS attack always involve small packet sizes) to make informed decisions in new and unseen scenarios [78].

Traffic Classification

Traffic classification has been applied to a wide range of network operation and management activities from QoS provisioning to performance monitoring as well as resource usage planning [38]. For example, an operator of an enterprise network may want to prioritize traffic for business-critical applications. Traditional port-based [80] and payload-based methods [81,82] become unreliable with the use of dynamically allocated ports and the ever-increasing security level. Therefore, many ML-based schemes have emerged. Liu et al. [83] propose FS-net, an end-to-end encrypted traffic classification model based on a multi-layer encoder-decoder structure. Shapira et al. [84] propose a CNN-based approach for encrypted traffic classification and application identification by transforming the packet sizes and packet arrival times of each flow into an image, a *FlowPic*. Huoh et al. [85] propose a multi-modal Graph Neural Network (GNN)-based approach for flow-based encrypted traffic classification, which maps traffic flows to graph representations. Wang et al. [86] propose TaTic, a two-phase early classification scheme, where the "easy flows" are quickly classified in the first phase by tree models, and more packets are used to classify the "hard flow" by using Temporal Convolutional Network (TCN). Yun et al. [87] propose a Transport Layer Security (TLS) traffic classification method, NeuTic, by designing a novel neural network equipped with a multi-kernel convolution and a sequence self-attention mechanism.

Network Security

Network security consists of protecting the network against cyber-threats that may compromise the network's availability. Doriguzzi-Corin et al. [88] propose Lucid, a CNN-based DDoS attack detection architecture. In Lucid, packet-level attributes were collected within a time-window and then used to generate a spatial data representation that is fed to a CNN. Vinayakumar et al. [89] employ an MLP to develop a flexible and effective intrusion detection system (IDS) for cyberattack detection. Li et al. [90] propose a transfer learning and ensemble learning-based IDS for Internet of Vehicles systems using CNNs. Longari et al. [91] propose CANnolo, an IDS based on Long Short-Term Memory (LSTM)-autoencoders to identify anomalies in controller area networks.

It is challenging to meet the throughput and latency requirements of modern networks while performing these ML-based traffic management tasks. In general, these aforementioned schemes deploy their models on control-plane servers. Moving the collected flow statistics from switches to the controller introduces a significant bottleneck for meeting the high throughput requirement. Running ML inference within the network data-plane would avoid data movements and address the challenge.

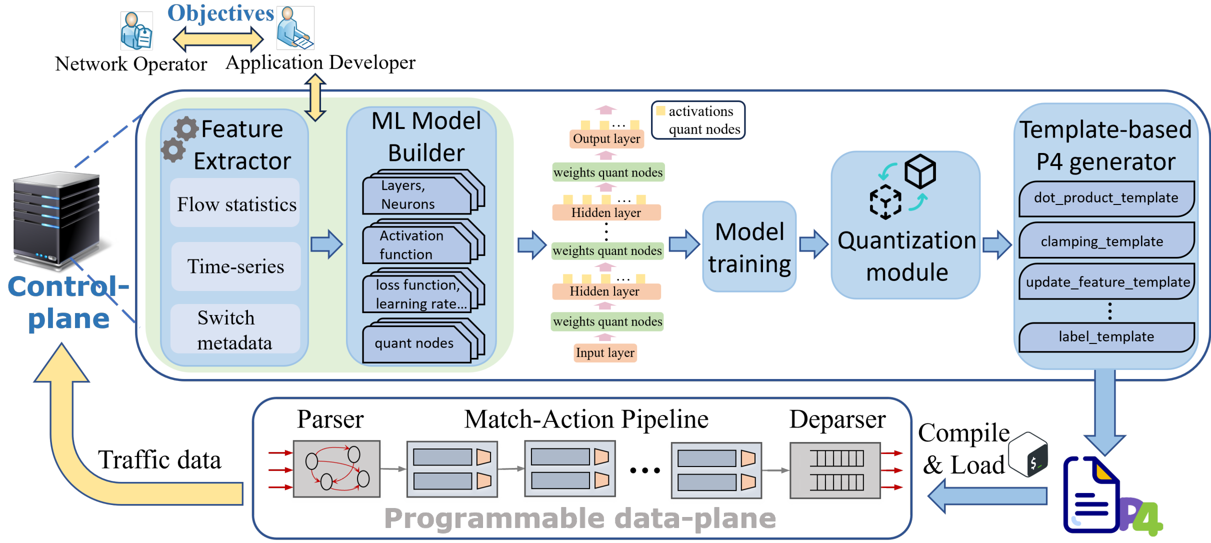


Figure 3.2: An overview of INQ-MLT toolbox.

3.4 Proposed Toolbox Overview

This section provides an overview of the proposed toolbox. As shown in Figure 3.2, the control-plane is responsible for building and training a ML model for network management applications. Network operators describe the desired application behavior to application developers and specify their application objectives (e.g., minimizing false positives in an anomaly detection model or maximizing the throughput of a traffic classification algorithm).

The trained model is then offloaded to the data-plane for various decision-making operations as part of the forwarding pipeline. The forwarding pipeline consists of a parser, multiple match-action tables and a deparser, enabling network devices to customize packet processing within the data-plane. The arriving packet first enters the parser for packet header parsing, then passes through multiple match-action tables for packet manipulation, and finally reaches the deparser for packet serialization. The controller collects the monitored traffic data from the data-plane to periodically retrain the ML model.

The collected traffic data is fed first into the feature extractor to obtain flow features, and application developers are charged with selecting suitable features. Based on the types of features, application developers can manipulate the ML model builder to create appropriate models. Once a floating-point model is obtained and trained, the quantization module generates a quantized ML model that employs P4-supported data types, thereby addressing the limitation of P4 not supporting floating-point numbers. The resulting quantized model utilizes integer weights and bit-oriented arithmetic operations. The mechanism

of the quantization module is introduced in Section 3.5.

The quantized ML model is then fed to the template-based P4 generation module which, in turn, produces the corresponding P4 code by assembling the predefined template of different operations. Finally, the generated P4 code representing the trained model is compiled and mapped to corresponding match-action tables in the IDP. We discuss our proposed packet processing pipeline in Section 3.6. Details pertaining to the remaining components of the toolbox are described in the following sections.

3.4.1 Feature Extractor

INQ-MLT identifies a flow $\mathcal{F}$ as a sequence of packets $[\mathcal{F}[i]$ for $i \in [0, 1, \ldots, |\mathcal{F}|-1]]$ having the same flow ID (e.g., source and destination IP addresses, source and destination ports, and transport-level protocol), where $\mathcal{F}[i]$ denotes the ith packet of the flow. A subflow is denoted as $\mathcal{F}[i : j]$ to represent the packet sequence $[\mathcal{F}[i], \mathcal{F}[i+1], \ldots, \mathcal{F}[j-1]]$ and consequently the first n packets of a flow are denoted by $\mathcal{F}[0 : n]$ or simply $\mathcal{F}[: n]$, where $n < |\mathcal{F}|$.

There are three different typical flow features employed in networking traffic management functionalities [92].

- **Standard flow statistics:** Standard statistical features include the mean, standard deviation, minimum, maximum, of packet length, inter-arrival time, TCP flag count, flow duration, etc. This type of feature is highly abstracted and normally picked by domain experts.
- **Time-series:** Tracking a fixed-size packet-level feature across the initial packets within a flow makes it possible to generate a dynamic-sized time-series feature that characterizes the flow. For example, the sizes of the packets in a flow are valid time-series features. This type of feature includes information about the relationship between two packets.
- **Raw bytes:** The rawest form of flow data, i.e., the actual flow bytes from packet headers and payloads.

Due to the depth limitation of the parser in parsing from the payload, using P4 to extract a sufficient number of useful bytes from the packet payload presents a challenging task. Additionally, deep examination of payloads using P4 requires clone and recirculation of the packet [93], which may cause a noticeable delay. Therefore, INQ-MLT only considers standard flow statistics and flow time-series features.

From a practical point of view, classifying a flow after it ends is not a useful task for many types of applications (e.g., intrusion detection). Therefore, to accurately identify an event as soon as possible, INQ-MLT only uses features based on the first n packets $\mathcal{F}[:n]$, where n is a parameter that depends on the application task (e.g., load balance) as well as the number of features. Once the feature type is selected, the next step is to select the relevant features and apply the necessary transformations.

In addition to extracting flow features, the extractor can also retrieve switch metadata (e.g., queue length, hop latency, and ingress timestamp) embedded by In-band Network Telemetry (INT) [94–96]. As a result, the switch state and the flow's entire history can be obtained to increase the power of the ML model.

3.4.2 The ML Model Builder and Model Training

Several factors affect the choice of ML models for network traffic management applications. The most important one is input features. Features directly affect not only the accuracy but input structure/dimension, which influences computational complexity. The choice of input feature and ML model are highly correlated.

Standard flow statistics: The number of statistical features, and consequently the input dimension, is often small. An MLP will be built when standard statistical features are selected.

Time series: A 1D-CNN will be constructed when time-series features are selected because 1D-CNN is capable of capturing temporal correlations between packets.

The application developer can assemble different pre-built neural network basic blocks (combinations of convolution kernels, fully connected layers, activation functions, and more) to design the optimal model for the task at hand.

Then, the generated neural network architecture will be ready for the subsequent training module. The model can be extended to suit the targeted network management operation (e.g., flow classification, flow size prediction, or DDoS prediction). Assume that the neural network consists of an input layer, N hidden layers, and an output layer. INQ-MLT normalizes each input into a zero-mean and unit-variance distribution. The output layer produces the desired outcome (e.g., flow predicted class). For the activation function at the output layer nodes, INQ-MLT utilizes softmax. The neural network uses back-propagation to learn the weights and biases of each layer. Training the model refers to finding values or weights on the links between the nodes of each consecutive layer that can be used in gradually calculating the correct output during model execution.

INQ-MLT inserts additional quantization nodes during the training process. For each link carrying the weight between two nodes, we add a *weights quant node*. Similarly, for each activation function output on each node for all the layers, we add an *activations quant node*. The design of these nodes will be discussed in Section 3.5.2.

To explain the purpose of these nodes, it is important to note that the deployed ML model at the IDP must satisfy several restrictions. IDPs cannot perform floating point operations and do not support matrix multiplication or looping operations. Hence, the weights of each layer of the ML model stored at the IDP are restricted only to integer representations. On the other hand, it is beneficial to use floating point representations of the features and node weights during training in order to increase the model accuracy. Hence, the training model at the control-plane trains and stores the model using floating-point values. The role of the quant nodes, then, is to train the model to compensate for any precision loss when the floating-point based model at the controller is transformed into a quantized ML model that can be stored at the IDP.

One of our design objectives is to keep the size of the ML model at the IDP small in order to reduce the switch memory consumption. Clearly, there is a trade-off between the model size (e.g., number of hidden layers) and the achieved accuracy, especially in some complex traffic management tasks. Hence, it is up to the developer to configure the desired input/output layers size as well as the number of hidden layers according to the selected network management operation while achieving the desired level of accuracy. Since the model is trained at the control-plane, these configurations can be reached after experimenting with different historical datasets. In our current implementation, we map the entire ML model to a single P4 program that is executed by a single forwarding device. We leave the idea of mapping larger ML models in a distributed manner at several switches as future work.

3.5 Quantization Module

The quantization module is employed to convert floating-point numerical model parameters and operations to lower-precision representations. Instead of adopting a 32-bit floating point format to represent weights and activations, the quantized ML model represents weights and activations using more compact formats (e.g., 8-bit integers) and in some cases binary values. Applying quantization to a trained model may introduce a perturbation to the trained model parameters. This can significantly reduce the model accuracy. We employ the quantization-aware training (QAT) technique [97] to address this limitation by inserting quantization nodes, as depicted in Figure 3.3. QAT simulates low-precision infer-

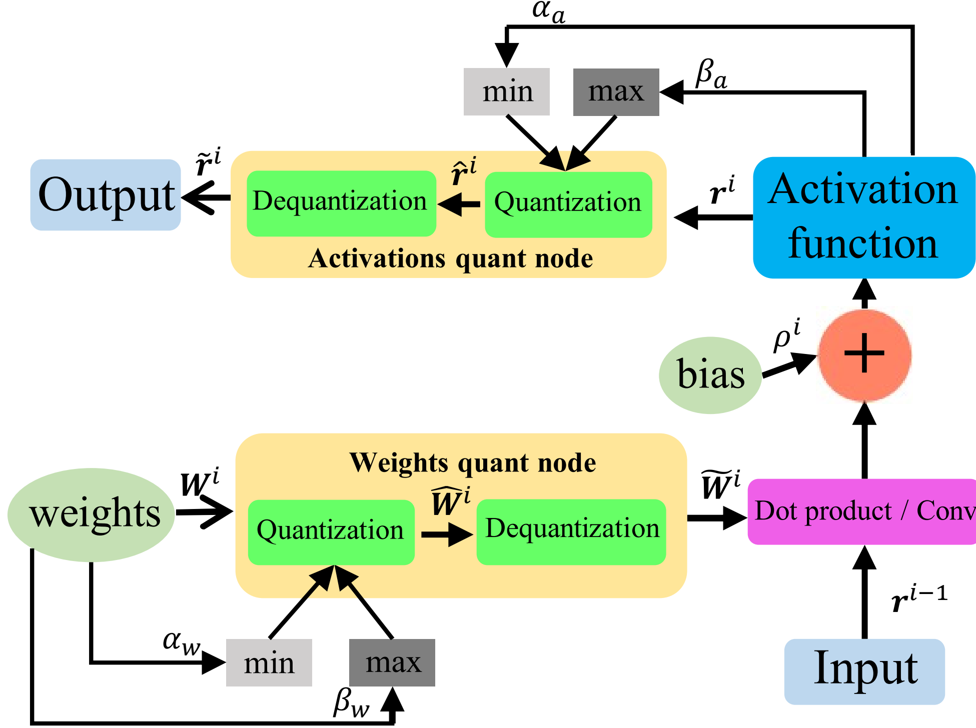


Figure 3.3: Added quant nodes at the controller.

ence time computation in the forward pass of the training process. This section describes how INQ-MLT quantizes the ML model and how the model can be employed.

3.5.1 Quantization and Dequantization Steps

At the controller, each quantization step maps a floating-point model parameter r within the continuous range $[R_{min}, R_{max}]$ to a finite integer value $\hat{r}$ between Q_{min} and Q_{max}, respectively. The range set $\{Q_{min}, \cdots, Q_{max}\}$ depends on the bit width B of the resulting integer $\hat{r}$ and the choice of whether it is a signed or an unsigned integer. More precisely,

$$(Q_{min}, Q_{max}) = \begin{cases} (0, 2^B - 1), & \hat{r} \text{ is unsigned} \\ (-2^{B-1}, 2^{B-1} - 1), & \hat{r} \text{ is signed} \end{cases} \tag{3.1}$$

For each model parameter, four quantization parameters must be defined. The first two are the values $\alpha \geq R_{min}$ and $\beta \leq R_{max}$, such that $[\alpha, \beta]$ denotes the clipping range of r, a bounded range that we are clipping the real values of r with.

Next, we define a real-valued scale S. The scale S specifies the quantization step, or the corresponding real-value distance between two consecutive integers. Using a constant value for S results in a uniform mapping. On the other hand, S can also be represented

as a function of the quantized weights resulting in a non-uniform mapping. In essence, a uniform quantization, selects S such that it divides a given range of real values r into an equal number of partitions,

$$S = \frac{\beta - \alpha}{2^B - 1} \tag{3.2}$$

The fourth parameter is an integer zero-point Z which is the quantization bias. The value of Z controls the mapping symmetry of different r values around its central value. Setting Z to zero results in a symmetric range that is efficient when the values that r takes are not skewed within $[\alpha, \beta]$.

Having defined those quantization constants, we can now proceed with the quantization process, where the quantized integer value $\hat{r}$ is obtained as follows,

$$\hat{r} = \text{clamp}(\left\lfloor \frac{r}{S} \right\rceil + Z; Q_{min}, Q_{max}) \tag{3.3}$$

where r is a real-valued input (activation output or node weight). $\lfloor \cdot \rceil$ is the round-to-nearest integer value operator and clamping is defined as:

$$\text{clamp}(\hat{r}; Q_{min}, Q_{max}) = \begin{cases} Q_{min}, & \hat{r} < Q_{min} \\ \hat{r}, & Q_{min} \leq \hat{r} \leq Q_{max} \\ Q_{max}, & \hat{r} > Q_{max} \end{cases} \tag{3.4}$$

In other words, any values of $\hat{r}$ that lie outside the range (Q_{min}, Q_{max}) will be clipped to its limits.

Dequantization recovers an approximation $\tilde{r}$ of the real value r using the quantized value $\hat{r}$ using:

$$\tilde{r} = S(\hat{r} - Z) \tag{3.5}$$

3.5.2 ML Quantization-Aware Training

As indicated before, the common approach to obtaining a quantized ML model is to design and train the model using floating point values and operations first. Then, this step is followed by quantizing only its weights. However, this may introduce a perturbation to the trained model parameters and result in a high-level of accuracy loss.

INQ-MLT circumvents this problem by adopting a different approach referred to as quantization-aware training (QAT). QAT recovers the quantization accuracy loss during training by inserting additional quantization (quant) nodes to the ML model during that phase as shown in Figure 3.3.

A quantization node is a sequence of quantization (3.3) and dequantization (3.5) operations stacked together. By adding those nodes, INQ-MLT introduces the quantization induced errors to the ML training phase. The model is then forced to learn as it is trained how to modify its weights in order to minimize its accuracy loss due to quantization. We note here that these additional nodes are only needed during the training phase and are not part of the ML model deployed to the IDP. Figure 3.3 illustrates how quantization nodes are added, respectively, for the quantized weights and activation functions outputs within the ML model. As shown, before the matrix of the weights of every layer is multiplied by the input to the layer, it passes through the *weights quant node* to be quantized and then dequantized. The output of the quant node is a new set of weights with some precision loss. The new weights matrix is multiplied by the input and a bias is added. Similarly, when an activation function is executed, its output is passed to an *activations quant node* before it is passed to the next layers. The mathematical details of the quantization and dequantization operations are explained in Eqn (3.3) and Eqn. (3.5). Hence the quant nodes ensure that the quantization precision loss is fed to the ML model as part of its training. QAT only simulates the lower precision behavior in the forward pass of the training process, while the back-propagation step remains unchanged.

After the training phase, the quantization process is performed on the pre-trained floating-point model based on the Eqn. (3.3). It is worth noting here that INQ-MLT treats the quantization ranges $[\alpha, \beta]$ differently for weights and activation function values. More precisely,

- For layer weights, the range is computed statically and is treated as a stored constant when the ML model is deployed to the IDP. It is sufficient for example to simply set $\alpha := \min r, \beta := \max r$.
- For activation functions outputs, (activations for short), the clipping range depends on the flow features, i.e., the input of the neural network. Therefore, the determination of the clipping range of activations quantization often requires a few batches of calibration data. The clipping range can be pre-calculated by running a series of calibration inputs to compute the typical range of activations. INQ-MLT can apply different metrics to find the best range.

3.5.3 Deployment of the Quantized ML Model

Once the model is trained, and all its weights are obtained by the control-plane, it is mapped to a corresponding P4 program. The IDP then executes the program using predefined

thresholds (e.g., every n packets of a flow or when a certain condition is satisfied). The outcome of the ML model then affects the processing of the packet as it passes within the IDP pipeline.

In this section, we demonstrate how the data-plane can store and execute the trained ML model given its constrained computing and memory resources. Assuming that the trained model has I Layers with N_i, $i = 1 \cdots I$, nodes at layer i. The weights of each layer i can be stored in the IDP as an $N_{i-1} \times N_i$ matrix of integer values. These values can be cached in the forwarding device memory or stored in its registers.

The triggering of each layer i, involves a multiplication operation between the output vector $\mathbf{r}^{i-1} = (r_1^{i-1}, \cdots, r_{N_{i-1}}^{i-1})$ of size N_{i-1} of the previous layer $i-1$ and the matrix of layer i weights, $\mathbf{W}^i$=$[w_{kl}^i]$ of size $N_{i-1} \times N_i$. Following this step by adding a bias $\boldsymbol{\rho}^i$ produces the vector $\mathbf{r}^i$= $\mathbf{r}^{i-1} \times \mathbf{W}^i + \boldsymbol{\rho}^i$. Here, $\boldsymbol{\rho}^i$ represents the bias value at layer i and $\mathbf{r}^i$ is the output of layer i and the ML model output for $i = I$.

Following [98], let $\hat{\mathbf{W}}^i$=$[\hat{w}_{kl}^i]$ and $\hat{\mathbf{r}}^i$=$(\hat{r}_1^i, \cdots, \hat{r}_{N_i}^i)$ represent the quantized values of the matrix $\mathbf{W}^i$ and the vector $\mathbf{r}^i$, $i = 1 \cdots I$, respectively, as obtained using Eqn. (3.3).

From the definition of matrix multiplication, we have:

$$r_k^i = \sum_{l=1}^{N_{i-1}} r_l^{i-1} \times w_{kl}^i + \rho_k^i \tag{3.6}$$

Substituting for the real-values of the matrix and vectors with their corresponding quantized values, we obtain:

$$S_k^i \left(\hat{r}_k^i - Z_k^i\right) = \sum_{l=1}^{N_{i-1}} S_l^{i-1} \left(\hat{r}_l^{i-1} - Z_l^{i-1}\right) S_{w,kl}^i \left(\hat{w}_{kl}^i - Z_{w,kl}^i\right) + S_b^i \rho_k^i \tag{3.7}$$

By normalizing the input features and selecting a maximal common clipping range for the elements within each vector and matrix and substituting in (3.2), the scaling factors and zero-points can be set such that S_1^i= S_2^i=$\cdots$=S^i, Z_1^i= Z_2^i=$\cdots$=Z^i for every vector $\mathbf{r}^i$ and $S_{w,kl}^i$=S_w^i,$Z_{w,kl}^i$=Z_w^i, $\forall k, l$. We observe that the bias scale is the product of the scales of the weights and of the input activations, i.e., $S_b^i = S^{i-1} \times S_w^i$. We can then have:

$$\hat{r}_k^i = Z^i + M^i \left(\sum_{l=1}^{N_{i-1}} \left(\hat{r}_l^{i-1} - Z^{i-1}\right) \left(\hat{w}_{kl}^i - Z_w^i\right) + \rho_k^i \right) \tag{3.8}$$

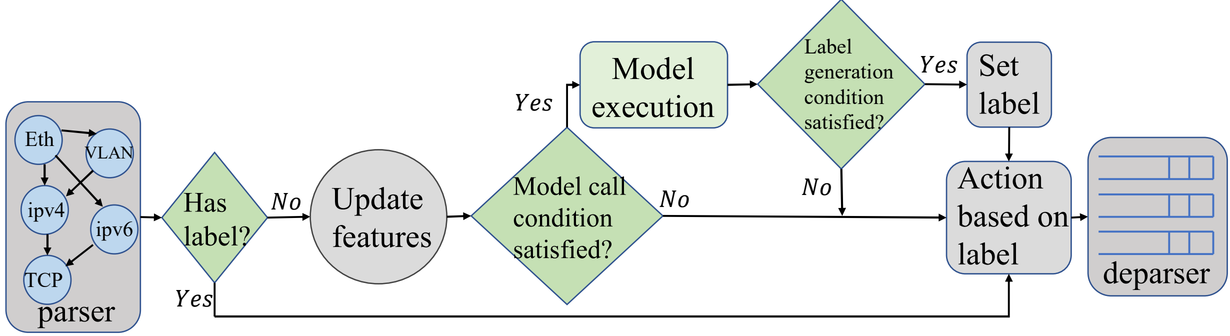


Figure 3.4: Packet processing pipeline.

where the multiplier M^i is defined as

$$M^i := \frac{S^{i-1} \times S_w^i}{S^i} \tag{3.9}$$

In Eqn. (3.8), the only non-integer value is that of the multiplier M^i, which can be calculated by the controller before deploying the model to the data-plane. Therefore, during the ML model execution process, we can consider these values as constants expressed and stored using fixed-point representations in the switch.

Clearly, the output of the last layer represents the final results of the ML model, which can be used as part of the data-plane packet pipeline as will be explained in Section 3.6.

3.6 P4 Generation

In this section, we describe the final step of how our proposed model is integrated within the switch P4 pipeline. The general processing pipeline of an IDP is illustrated in Figure 3.4 and is explained in the following sections.

Parser

For an arriving packet, it is first mapped into a Packet Header Vector (PHV) by the parser based on the requirement of models. The PHV contains different fields of the packet header, such as the flow ID and associated metadata (e.g., queue length upon packet arrival). Then the PHV is checked if incoming packets contain a label. If a label is present, INQ-MLT executes the corresponding action (e.g., drop the packet). In the case of traffic classification functionalities, the parser also contrasts the packet flow ID against a list of already classified or labeled flows using hash functions. If the packet flow ID is not found

in the list it is added. Once hashed, the corresponding stored statistics are retrieved and stored in the associated metadata.

Registers are used to store the aforementioned list of flows, their statistics, and the ML model output for their classification. For example, in the case of an ML model that is used for classifying heavy flows, the first cell in the register will store the flow ID and the last cell will store the label of the flow (e.g., for heavy flow identification, 0 for normal flow and 1 for heavy).

Initially, all cells in the registers are initialized to a default label. When the ML model is triggered to label a flow, the corresponding cell is modified with the new label. Hash functions in P4 are used to map flows to registers based on their IDs.

Update Features

All features and fields that are used by the ML model are also stored in registers. The P4 code that updates some features (e.g., average packet length) is limited to basic integer operations (e.g., addition, subtraction, and hash). To calculate standard statistic features, bit operations are used (e.g., we use right bit shift for divisions). To minimize the number of times these operations are executed, these values are only calculated upon triggering the ML model. For example, to obtain the average packet length, the register maintains the sum of the lengths of all packets as well as the number of packets. Before executing the ML model, the average is calculated once and fed to the model. To increase the accuracy of the calculations, it is preferred to choose the value of the number of packets as a power of two to maintain the accuracy of the division operation as it is carried out using bit-shift operations. For time-series features, rewrite the registers with the features of current packet when the feature of the previous packet is no longer required for model inference.

ML Model Execution

After updating features, INQ-MLT checks if the model execution conditions are satisfied. The execution of the ML model can be either partial or complete, depending on the model type. For example, an MLP is fully executed when a specific condition is met (e.g., the nth packet arrivals). In contrast, a 1D-CNN can be executed partially upon packet arrival by only executing a single elementwise multiplication. The intermediate results of this multiplication are stored in registers, and labels will only be generated when the label generation conditions are satisfied.

To simplify the mapping from the ML model to P4 code, we create a template library that contains various parameterized templates for commonly used operations. Each

Table 3.1: The common used templates.

Name	Parameters
dot_product_template	The weight matrix $\hat{\mathbf{W}}^i$; the zero-points Z^{i-1}, Z^i and Z_w^i; the multiplier M^i; the bias $\boldsymbol{\rho}^i$.
Conv1D_template	The kernel matrix $\hat{\mathbf{W}}^i$; the zero-points Z^{i-1}, Z^i and Z_w^i; the multiplier M^i; the bias $\boldsymbol{\rho}^i$.
normalize_quantize_template	The mean and variance of the training set.
clamping_template	The bit width applied and the choice of the use of signed or unsigned integer.
update_feature_template	The feature type (count, maximization, minimization or average).
label_template	The number of classes, i.e., how many different categories or classes.

template parameters are based on the ML model configuration or are calculated by the control-plane offline (e.g., the weights of each layer, the mean and variance used to normalize the input and the bit width applied). Table 3.1 shows some common used templates and their parameters. The P4 developer feeds these parameters to corresponding templates and assembles these template blocks into larger blocks or full packet pipelines. For example, the template of the operation of Eqn.(3.8) denoted *dot_product_template* is shown in Listing 3.1.

Listing 3.1 shows an example of the *dot_product_template* in layer i with two vectors $[\texttt{x1}, \texttt{x2}]$ and $[\hat{w}_{11}^i, \hat{w}_{21}^i]$.

```
1 meta.f1=(((x1-Z^{i-1})*W_{11}+(x2-Z^{i-1})*W_{21}+ρ_1^i)*m)>>e+Z^i;
```

Listing 3.1: P4 code fragment of a dot product operation based on Eqn.(3.8).

Here the multiplier M^i is expressed using its fixed-point representation $m \times 2^{-e}$, and 2^{-e} is performed by using the right bit shift operation >>. $W_{kl} = \hat{w}_{kl}^i - Z_w^i$ can be pre-computed since both $\hat{w}_{kl}^i$ and Z_w^i are constants. The calculation result is stored in `meta.f1`.

```
1 action i-th_Layer_calculation(){
2     int<32> x1=meta.prevLayerOutput1;
3     int<32> x2=meta.prevLayerOutput2;
4     // ...
5     dot_product_template[C1]();
6     dot_product_template[C2]();
7     /*similar code is omitted*/
8     dot_product_template[Cn]();}
```

Listing 3.2: P4 code fragment of an action for the vector-matrix multiplication of a neural network layer.

For the vector-matrix multiplication of an ML layer, we construct an action block which consists of multiple *dot_product_templates* with different parameter combinations, as shown in Listing 3.2, where `Cn` denotes the parameter combination of the nth column of the weight matrix $\hat{\mathbf{W}}^i$, the nth element of bias vector $\boldsymbol{\rho}^i$, the zero-points Z^{i-1}, Z^i and $\hat{Z}^i$ and the multiplier M^i.

We note that, for the result of each multiplication to be stored and fed to the next layer, they must be clamped using Eqn.(3.4), which involves conditional statements. Thus, we create *clamping_template*, whose parameter depends on the bit width used and whether the integer is signed or unsigned. Listing 3.3 shows an example of mapping an MLP model with one hidden layer by assembling the templates in an apply control block.

```
1  apply {
2       update_feature_template;
3       if(The ML call condition == True){
4           feature1.read();
5           feature2.read();
6           // Omit codes to extract other features
7           normalize_quantize_template;
8           firstLayer_calculation();
9           clamping_template(meta.1stLayerOutput1);
10          clamping_template(meta.1stLayerOutput2);
11          // ...
12          clamping_template(meta.1stLayerOutputn);
13          secondLayer_calculation();
14          meta.class=label_template(meta.output);
15      }}
```

Listing 3.3: P4 code fragment for mapping a MLP model with one hidden layer.

Before we check whether the ML model call condition is satisfied, we always need to update the flow features by using the *update_feature_template* (line 2 in Listing 3). Its parameter is the feature type, i.e, how do we update the features (e.g., count, maximization, minimization or average). Once the neural network is invoked, the values of the input features are fetched from the registers. Before feeding these values to the model, they are normalized. Division operations during normalization are carried out using bit shift operations. The values are then quantized to the integer range set depending on the bit width applied, and used as an input to the first layer in the ML model. *normalize_quantize_template* is used here. Consequently, each layer is triggered by executing the vector-matrix multiplication operation, using one layer at a time.

For the output layer, the classification result is obtained by using *label_template*, whose parameter relies on the number of classes. Recall that softmax is used as the final activation function; therefore, after obtaining the output vector, the label can be determined by identifying the node with the highest value through a straightforward comparison using *if* statements for binary classification. For a multi-class classification problem, it is more

scalable to realize the number comparison (i.e., the argmax operation) by designing a ternary matching table, as detailed in [99].

Set Label and Apply Action

Once the ML model is executed and the label generation condition is satisfied, its output is used to set the label of the packet flow. P4's match action tables are then used to find the suitable action for the packet belonging to the newly labeled flow. For example, if the packet is labeled as malicious, dropping the packet can be implemented by directly calling the P4 drop primitive or sending it to a deep packet inspection module [82]. Finally, at the end of the pipeline, the deparser assembles the packet data back into a stream of bytes for transmission before their departure.

Summary and Discussion

The P4 code is generated by assembling templates for commonly used operations. The P4 mapping of the vector-matrix multiplication consists of many *dot_product_templates*, and each contains multiple basic addition and multiplication operations. Clearly, the computational complexity increases with the growth of the size of the matrix, which increases the ML model inference latency, especially when we increase the model accuracy by using more neurons in each layer. To accelerate the ML model execution, we sketch two potential solutions and leave them as future work. The first solution is to explore the possibility of executing actions in parallel in P4. The second solution is to design an efficient approximate matrix multiplication algorithm [100] which reduces the problem to exact matrix multiplication in a lower-dimensional space.

It is also important to note that, to deploy our toolbox generated P4 programs, the programmable targets must provide direct support for multiplication operations. For those P4 targets that do not provide such support, we believe that with certain extensions, the P4 code generated by our toolbox can be adapted for use. Generally, most P4 targets support the shift and additions. Therefore, the multiplication operation can be decomposed into a number of shifts and additions, as detailed in [101]. Depending on the vector-matrix operations scale, the packets need to be recirculated to the pipeline multiple times to complete all the required operations. Even Taurus [58], which adds an FPGA-based implementation of a custom MapReduce block to programmable switches, enabling the support of vector-matrix multiplication. Our toolbox can adapt to Taurus switches by converting the generated P4 code to the corresponding Spatial language [102] on the FPGA.

Finally, replacing vector-matrix multiplication with table lookups can also be explored as a more general approach, as discussed in [103] and [104].

3.7 Theoretical Precision Analysis

In this section, we analyze how the precision of the quantized IDP model for activations and weights impact the IDP model inference accuracy theoretically. We employ the DNN precision analysis framework in [105].

3.7.1 Notation

Recall that the neural network trained by the controller for the traffic management task has an output layer with size N_I, and softmax is used as the output layer activation function. Therefore, to give the management decision, the neural network would typically have N_I numerical outputs $\{u_j\}_{j=1}^{N_I}$ and the decision would be $y = \arg\max_{j=1,\ldots,N_I} u_j$. Each numerical output is a function of the neural network's weights and activations:

$$u_j = f\left(\{a_h\}_{h\in\mathcal{A}}, \{w_h\}_{h\in\mathcal{W}}\right) \tag{3.10}$$

for $j = 1, \ldots, N_I$, where $\mathcal{A}$ and $\mathcal{W}$ are the index sets of all activations and weights in the network, respectively. a_h denotes the activation indexed by h and w_h denotes the weight indexed by h.

After applying the quantization module, the activations and weights are quantized to B_A and B_W bits, respectively, and the output u_j is corrupted by quantization noise ε_{u_j} so that:

$$u_j + \varepsilon_{u_j} = f\left(\{a_h + \varepsilon_{a_h}\}_{h\in\mathcal{A}}, \{w_h + \varepsilon_{w_h}\}_{h\in\mathcal{W}}\right) \tag{3.11}$$

where ε_{a_h} and ε_{w_h} are the quantization noise terms of the activation a_h and weight w_h, respectively. It is standard to assume that $\{\varepsilon_{a_h}\}_{h\in\mathcal{A}}$ are independent uniformly distributed random variables on $\left[-\frac{\Delta_A}{2}, \frac{\Delta_A}{2}\right]$, and $\{\varepsilon_{w_h}\}_{h\in\mathcal{W}}$ are independent uniformly distributed random variables on $\left[-\frac{\Delta_W}{2}, \frac{\Delta_W}{2}\right]$, with $\Delta_A = 2^{-(B_A-1)}$ and $\Delta_W = 2^{-(B_W-1)}$ [106].

3.7.2 Accuracy of Non-quantized Controller Models and Quantized IDP Models

For the non-quantized controller model and its quantized counterpart at the IDP, we define:

The non-quantized controller model error probability

$$p_{e,fl} = \Pr\{Y_{fl} \neq Y\}$$

where Y_{fl} is the output of the non-quantized controller model and Y is the true label.

The quantized IDP model error probability

$$p_{e,q} = \Pr\{Y_q \neq Y\}$$

where Y_q is the output of the quantized IDP model.

The mismatch probability between non-quantized controller models and quantized IDP models

$$p_m = \Pr\{Y_q \neq Y_{fl}\}$$

Proposition 1. *The upper bound for the error probability $p_{e,q}$ of the quantized IDP model is determined by the scenario in which there is no overlap between misclassified samples and samples whose predicted labels are in error due to quantization, as shown below:*

$$p_{e,q} \leq p_{e,fl} + p_m \tag{3.12}$$

$$\begin{aligned}
&\textit{Proof}: p_{e,q} = \Pr\{Y_q \neq Y\} \\
&= \Pr\{Y_q \neq Y, Y_q = Y_{fl}\} + \Pr\{Y_q \neq Y, Y_q \neq Y_{fl}\} \\
&= \Pr\{Y_{fl} \neq Y, Y_q = Y_{fl}\} + \Pr\{Y_q \neq Y, Y_q \neq Y_{fl}\} \\
&\leq p_{e,fl} + p_m.
\end{aligned}$$

From the above proposition, it can be seen that it requires estimating the mismatch probability to obtain an upper bound error probability of the quantized IDP model.

3.7.3 Quantization Analysis

In quantization noise analysis, it is standard to ignore the cross-products of quantization noise terms as their contribution is negligible. Therefore, using Taylor's theorem, the total quantization noise at the output of the quantized IDP model can be expressed as [105,107]:

$$\varepsilon_{u_i} = \sum_{h \in \mathcal{A}} \varepsilon_{a_h} \frac{\partial u_i}{\partial a_h} + \sum_{h \in \mathcal{W}} \varepsilon_{w_h} \frac{\partial u_i}{\partial w_h} \tag{3.13}$$

By evaluating the probability of any pair of outputs ($u_i < u_j$) that flipped due to quantization errors $\Pr\left(u_i + \varepsilon_{u_i} > u_j + \varepsilon_{u_j}\right)$, we can obtain an analytical upper bound on the mismatch probability p_m between the non-quantized controller and quantized IDP models.

Proposition 2. *Given B_A and B_W, the mismatch probability p_m between the non-quantized controller model and its quantized counterpart at the IDP is upper bounded as follows:*

$$p_m \leq \Delta_A^2 E_A + \Delta_W^2 E_W \tag{3.14}$$

where

$$E_A = \mathbb{E}\left[\sum_{\substack{j=1 \\ j \neq Y_{fl}}}^{N_I} \frac{\sum_{h \in \mathcal{A}} \left|\frac{\partial\left(u_j - u_{Y_{fl}}\right)}{\partial a_h}\right|^2}{24\left|u_j - u_{Y_{fl}}\right|^2}\right] \tag{3.15}$$

and

$$E_W = \mathbb{E}\left[\sum_{\substack{j=1 \\ j \neq Y_{fl}}}^{N_I} \frac{\sum_{h \in \mathcal{W}} \left|\frac{\partial\left(u_j - u_{Y_{fl}}\right)}{\partial w_h}\right|^2}{24\left|u_j - u_{Y_{fl}}\right|^2}\right] \tag{3.16}$$

E_A and E_W are the activation and weight quantization noises, respectively. The expectations $\mathbb{E}$ are taken over a random input and $\{a_h\}_{h \in \mathcal{A}}$, $\{u_j\}_{j=1}^{N_I}$, and Y_{fl} are thus random variables.

The detailed proof of this proposition can be found in [105].

The first term in (3.14) characterizes the impact of quantizing activations on the overall IDP model accuracy while the second characterizes that of weight quantization. We observe that the mismatch probability p_m increases as Δ_A^2 and Δ_W^2 becomes larger. This can be attributed to the fact that smaller precision results in more mismatch. Proposition 2 provides additional insight: the mismatch probability decreases exponentially with precision, because $\Delta_A = 2^{-(B_A - 1)}$ and $\Delta_W = 2^{-(B_W - 1)}$.

To obtain the quantities in the expectations in (3.15) and (3.16), it is necessary to perform one forward pass on an estimation set. During this forward pass, the numerical outputs are recorded, followed by one backward pass to probe all relevant derivatives. Consequently, (3.14) can be readily calculated. Then, we simply combine (3.12) with (3.14) to obtain an estimate of the accuracy of the quantized IDP model.

3.8 Performance Evaluation

We have implemented the performance of INQ-MLT on a network with BMv2 software switches [34]. We selected anomaly detection [108] and traffic classification [109] as the use-case scenarios for validating the performance of our proposed toolbox.

3.8.1 Dataset

Anomaly detection: The *CICIDS2017* [108] dataset is provided by the Canadian Institute for Cybersecurity, University of New Brunswick Canada. This dataset consists of data collected from several days of network activity. It contains both benign and malicious (e.g., flows representing DDoS attacks) flows. We select Tuesday, Wednesday, and Friday traces. We extract the flow features of the first eight packets to collect the needed input features vector for the ML model, which is then executed as part of the eighth packet's processing pipeline. The used dataset is highly unbalanced. Following [88], we utilize an under-sampling technique that randomly removes records from the dominant class to ensure unbiased learning and classification. We build MLPs to perform anomaly detection. MLP represents the basis for implementing artificial and deep neural networks and has been widely used as a benchmark in various network management tasks [110,111]. Several stateful features are used to train the MLP, as listed in Table 3.2.

Traffic classification: The *ISCX VPN-nonVPN* [109] dataset is utilized for the task of classifying traffic according to the function types. It contains six function types of network traces (email, chat, file transfer, P2P, VoIP, and streaming). We build a 1D-CNN to perform this traffic classification task. Three time-series features, i.e., packet length, inter-arrival time, and TCP window size, of the first eight packets, are used to train the 1D-CNN. These datasets are stored in PCAP file format which facilitates the use of dpkt [1] to extract the features.

3.8.2 Experiment Setup

For each use-case, the dataset is split into a training (60%), a validation (20%), and a test (20%) sets. The model training, validation, and quantization operations are performed by the control-plane using TensorFlow Lite [2]. We created a simple network topology consisting of two end-hosts connected by one forwarding device in the Mininet [3] environment. The

[1] https://github.com/kbandla/dpkt
[2] https://www.tensorflow.org/lite
[3] http://mininet.org/

Table 3.2: Features used for model training.

Model	Feature	Description
MLP	PSH	Cumulative number of occurrences of the TCP flag PSH.
	RST	Cumulative number of occurrences of the TCP flag RST.
	SYN	Cumulative number of occurrences of the TCP flag SYN.
	FIN	Cumulative number of occurrences of the TCP flag FIN.
	Max_Pkt_Len	Size of the largest packet.
	Mean_Pkt_Len	Average size of packets.
1D-CNN	Pkt_Len	The time-series of packet length.
	IAT	The time-series of inter-arrival time between two adjacent packets.
	TCP win_size	The time-series of TCP window size.

Linux network traffic tool, Tcpreplay [4], was used to replay the real network traces. *iperf* [5] is used to measure the throughput. The bit width B is set as 8.

MLP configuration: Two MLPs are developed: MLP-1 and MLP-2. Specifically, MLP-1 is composed of 1 hidden layer with 16 hidden units; whereas MLP-2 is composed of 2 hidden layers with 16 and 8 hidden units, respectively. Each hidden layer has a ReLU activation function.

CNN configuration: The 1D-CNN consists of 1 hidden layer with 16 kernels of size 2×3 and stride of 1, a flatten layer, and a fully-connected layer with 6 nodes.

3.8.3 Evaluation Metrics

For a specific traffic class i under analysis, classified test samples are divided into three categories:

- *True Positives for class i* (TP_i) - the number of flows that are correctly classified for $class_i$.
- *False Positives for class i* (FP_i) - the number of flows that are classified as belonging to $class_i$ but are not generated by $class_i$.
- *False Negatives for class i* (FN_i) - the number of flows that are classified as not belonging to $class_i$ but are indeed generated by $class_i$.

The performance metrics used for evaluating our model are accuracy, recall, precision, and F1 score.

[4]https://tcpreplay.appneta.com
[5]https://iperf.fr

Recall

It is defined as the rate of the flows correctly classified as $class_i$ in the total $class_i$ flows:

$$Recall_i = \frac{TP_i}{TP_i + FN_i}$$

Precision

It is defined as the rate of the real flows belonging to $class_i$ in all the flows classified as $class_i$:

$$Precision_i = \frac{TP_i}{TP_i + FP_i}$$

F1-score

The F1-score is the harmonic mean of precision and recall, and it measures the balance between precision and recall and is defined as:

$$F1\text{-}score_i = 2 \times \frac{Recall_i \times Precision_i}{Recall_i + Precision_i}$$

Accuracy

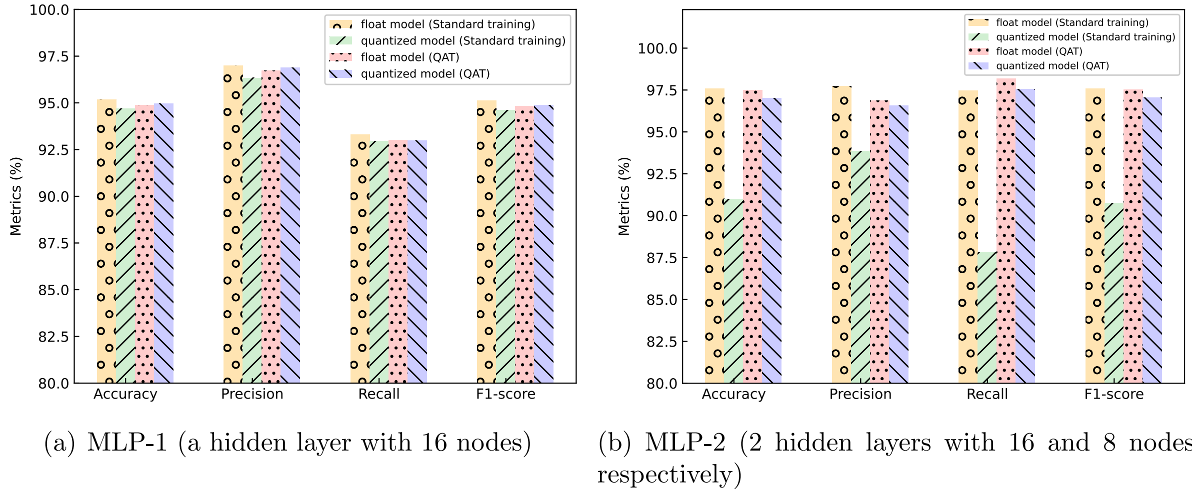
The overall accuracy metrics are used to assess the overall classification performance of the classifier, which is defined as the ratio of the total number of samples that are correctly classified by the model to the total number of samples:

$$Accuracy = \frac{\sum_{i \in \ classes} TP_i}{\sum_{i \in \ classes} (TP_i + FP_i)}$$

3.8.4 Classification Result

In our comparative analysis, we evaluate the proposed Quantization-Aware Training (QAT) scheme alongside the standard training scheme, where quantization nodes are not added during the training process.

For MLP-1, as depicted in Figure 3.5(a), it was observed that both the QAT scheme and the standard training scheme exhibited nearly identical levels of performance, with no significant difference in accuracy. In contrast, Figure 3.5(b) shows when assessing the performance of QAT for MLP-2, characterized by two hidden layers, notable distinctions



(a) MLP-1 (a hidden layer with 16 nodes)

(b) MLP-2 (2 hidden layers with 16 and 8 nodes respectively)

Figure 3.5: Classification performance for MLP.

emerged. The quantized model associated with the QAT scheme demonstrated a notably higher accuracy in comparison to the standard training scheme, wherein the latter exhibited a reduction in accuracy. This shows that QAT is more effective in improving the accuracy of deeper neural network architectures, such as MLP-2, compared to shallower ones like MLP-1. This is because the quantization error accumulates gradually with the increase of hidden layers, leading to accuracy loss.

Table 3.3: Classification performance for 1D-CNN.

	Overall Accuracy	Metric	Email	Chat	Streaming	Voip	File	P2P	Weighted Average
float model (standard training)	0.7230	Precision	0.7536	0.4583	0.7458	0.6816	0.6933	0.9176	0.7142
		Recall	0.8125	0.1964	0.4490	0.8085	0.8739	0.9176	0.7230
		F1-score	0.7820	0.2750	0.5605	0.7397	0.7732	0.9176	0.7040
quantized model (standard training)	0.6623	Precision	0.6974	0.4000	0.6429	0.5730	0.7407	0.9390	0.6651
		Recall	0.8281	0.1429	0.4592	0.8564	0.5042	0.9059	0.6623
		F1-score	0.7571	0.2105	0.5357	0.6866	0.6000	0.9222	0.6420
float model (QAT)	0.7230	Precision	0.8387	0.3721	0.6667	0.6912	0.7405	0.9176	0.7146
		Recall	0.8125	0.2857	0.4898	0.7979	0.8151	0.9176	0.7230
		F1-score	0.8254	0.3232	0.5647	0.7407	0.7760	0.9176	0.7145
quantized model (QAT)	0.7246	Precision	0.9123	0.3953	0.6712	0.6741	0.7422	0.9176	0.7203
		Recall	0.8125	0.3036	0.5000	0.8032	0.7983	0.9176	0.7246
		F1-score	0.8595	0.3434	0.5731	0.7330	0.7692	0.9176	0.7176

Table 3.3 presents similar results for the multi-label traffic classification tasks based on 1D-CNN and it shows the performance results on metrics for all 6 traffic function types.

For the QAT scheme-associated models, compared to the non-quantized float model, we note that the quantized model has almost no loss in performance metrics. This validates the effectiveness of QAT.

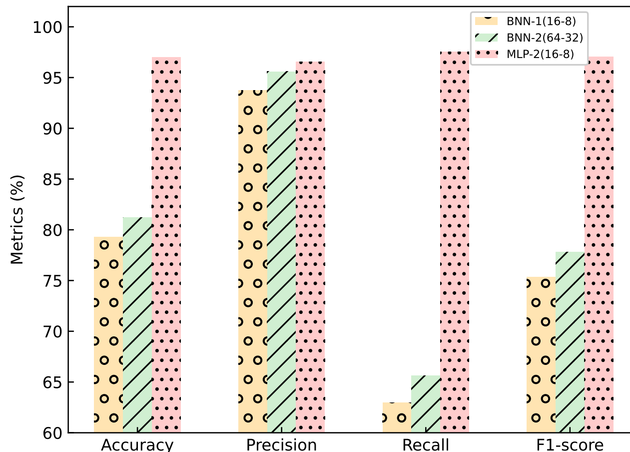


Figure 3.6: Comparison between BNN and 8-bit precision MLP.

We also conduct an evaluation to compare the classification performance of our quantized neural network with the binary neural network (BNN). As outline in [54], even simplified versions of BNNs are onerous to deploy into commodity programmable switches: a very basic BNN with two layers of 64 and 32 neurons already exhausts the resources of an Intel Tofino ASIC, while yielding poor inference results. Therefore, we train two BNNs for the anomaly detection task: BNN-1 with the identical architecture as our MLP-2 and BNN-2 conforming to the structure with 64 and 32 neurons referenced in [112]. The comparative analysis depicted in Figure 3.6 illustrates the performance difference between the two BNNs and the 8-bit precision MLP-2. As expected, the neural network with 8-bit precision exhibits higher accuracy compared to the two BNNs. BNN-2 employs a higher number of neurons, however, the enhancement in its classification performance is marginal. The distinctions between the BNN and the 8-bit precision neural network performance can be attributed to the following reasons. Specifically, an 8-bit precision allows for a wider range of weight values compared to binary weights, enhancing the model's expressiveness. Furthermore, binary weights (1-bit) inherently introduce quantization errors during training and inference, potentially leading to a loss of information. In contrast, the 8-bit precision offers a higher level of granularity, reducing quantization errors and preserving more information during computations.

3.8.5 INQ-MLT in Switches

The previous experiments are performed with Python. We now further evaluate our scheme on BMv2 software switches. We implement MLP-1 on the software switch. We compare the performance of our proposed architecture against that of a centralized controller-based ML model at runtime. In this model, the data-plane sends collected features to the controller, which executes the ML model, and then communicates back the classification decision to the data-plane. Figure 3.7 provides a comparison between the controller-based ML model and our P4 implementation of the ML model. As depicted, there is almost no accuracy difference.

Furthermore, we evaluate the classification performance of our approach in comparison to DTs with level-table mapping scheme adopted by [52, 53]. As claimed by pForest [52], only shallow trees of depth 4 can be implemented due to resource limitation. Therefore, we performed a comparison to determine the depth required for a DT to reach equivalent accuracy as a neural network with only one hidden layer. We can see from Figure 3.7, to achieve the same accuracy as MLP-1, for the DT with level-table mapping scheme, a depth of 6 is needed.

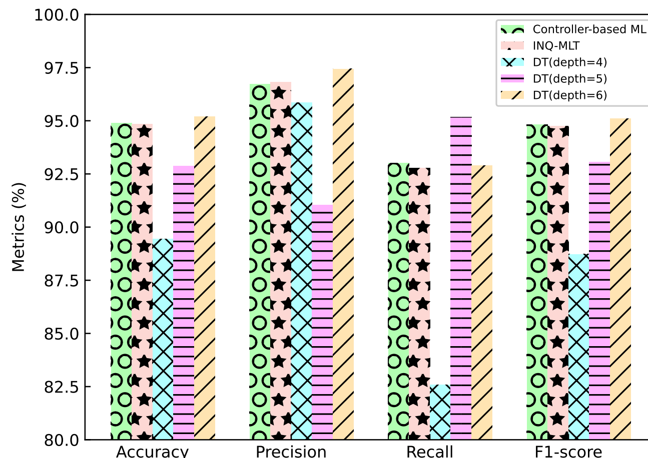


Figure 3.7: The performance of INQ-MLT, controller-based ML and DT for anomaly detection.

In terms of the throughput of the IDP pipeline, we first note that the ML inference delay in the controller consists of three parts: the transmission delay of sending features from the forwarding device to the controller, the ML execution delay in the controller and finally the transmission delay of returning the result back to the switch. Figure 3.8 plots the throughput of a scenario where every packet triggers the execution of the ML model for both the IDP of our INQ-MLT and that of the controller against various values of the one-way

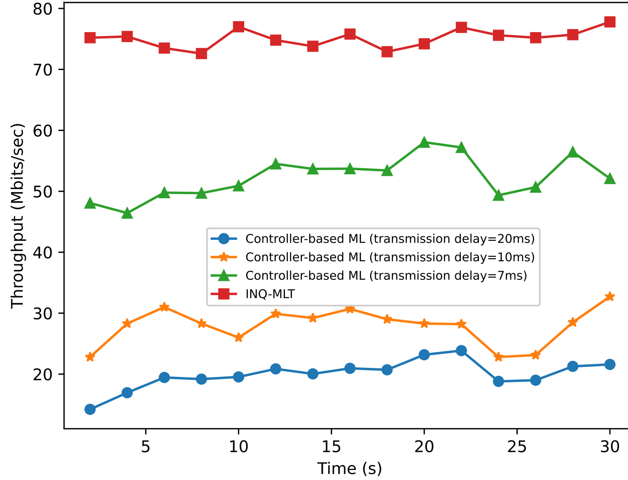
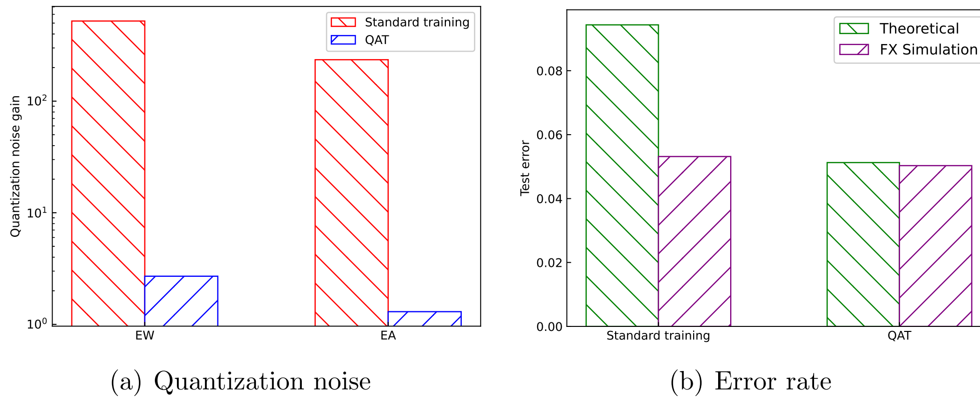


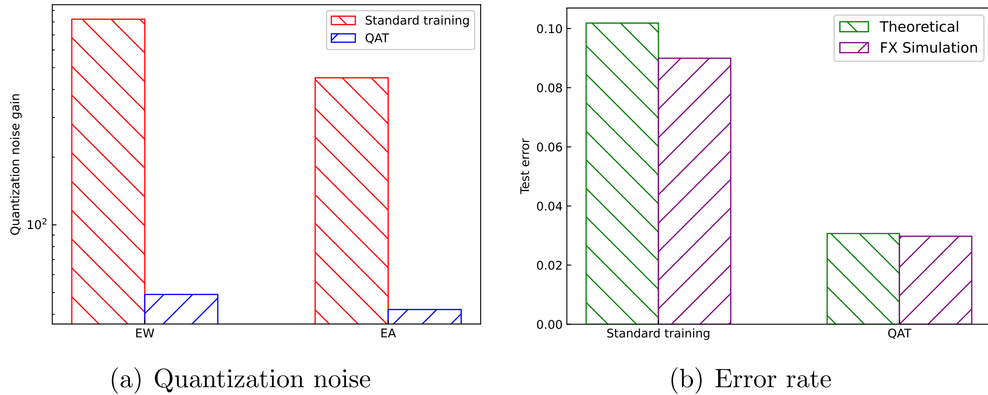
Figure 3.8: Throughput comparison.

transmission delay between the controller and the switch. The choice of transmission delay is based on [113]. We neglect the ML execution delay at the controller due to the controller's high computational capability. As depicted, the controller-based ML exhibits an increase in throughput as the transmission delay decreases, aligning with expectations. In contrast, our proposed approach, operating without causing any transmission, consistently achieves higher throughput compared to other three controller-based schemes. Since BMv2 was not developed as a production-grade switch, a better performance should be expected whilst running on a hardware switch. The plot demonstrates that INQ-MLT achieves a significant reduction in the traffic management decision-making process, especially when the transmission delay between the controller and the switch is high.



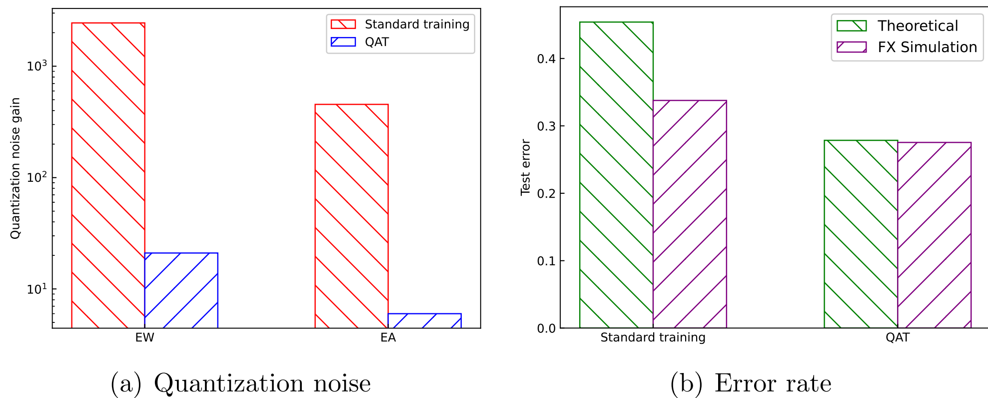
(a) Quantization noise (b) Error rate

Figure 3.9: Comparison of quantization noises and error rates between standard training and QAT Schemes for MLP-1.



(a) Quantization noise (b) Error rate

Figure 3.10: Comparison of quantization noises and error rates between standard training and QAT Schemes for MLP-2.



(a) Quantization noise (b) Error rate

Figure 3.11: Comparison of quantization noises and error rates between standard training and QAT Schemes for 1D-CNN.

3.8.6 Precision Analysis Result

Figures 3.9 to 3.11 show the quantization noises computed by (3.15) and (3.16) and the theoretical and simulation error rates, respectively. Both quantization noises E_A and E_W for standard training scheme are higher than the corresponding values for the QAT scheme, which means the impact of quantizing activations and weights on the overall accuracy is smaller for the QAT scheme compared to the standard training scheme. The theoretical bound successfully upper bound the error rate obtained through fixed-point simulations (FX simulation) for both schemes, which validates the analysis conducted in Section 3.7. The theoretical error rate is smaller for the QAT scheme compared to the standard training scheme, indicating that applying the QAT scheme offers a higher theoretical guarantee on the model's accuracy.

3.9 Discussion

In this section, we discuss some important considerations regarding the use of the toolbox.

Updating ML models during runtime: After an ML model is deployed, it is essential to update it periodically to adapt to changes in traffic patterns and maintain model performance. There are two possible updates scenarios: updating model weights and model modification (e.g., changing the number of selected flow features). Weights update can be realized by storing weights in reconfigurable memory (e.g., registers) and updating them atomically, which is analogous to updating values and without affecting the forwarding pipeline. Modifying the ML model by changing the flow features is expected to be an infrequent event but it remains to be a significant challenge to perform this modification at runtime. Existing solutions such as FlexCore [114] and IPSA [115] aims to achieve such P4 runtime programmability but they all require modifications to the existing switch architecture.

ML performance: Normally, a trade-off exists between model size, inference accuracy, and resource consumption. Increasing the number of features may require additional hidden layer neurons to capture the characteristics of these features, leading to a larger model and a higher computing resources. In some cases, while a large model can achieve higher accuracy, it may impact the line-rate execution of the P4-based target. Therefore, in certain tasks, it is acceptable to tolerate some accuracy loss to maintain the switches' processing speed at line-rate. Additionally, pruning [116] can be employed to minimize the computing resources needed for executing multiplication operations by removing parameter that do not impart the inference accuracy.

Feature Storage: Registers are the most widely used externs and are supported by most P4 targets. They are typically implemented in Static Random-Access Memory (SRAM) in certain P4 hardware [117]. Registers are used to maintain per-flow states and compute flow-level features. P4 uses hash functions to allocate the memory cells for each flow. Depending on the characteristics of distinct features, the width of memory cells employed for storage may vary; for instance, allocating 4 bits for the TCP flag and 16 bits for the packet size is sufficient. For the features used to train our MLP in the anomaly detection task, each flow requires 48 bits to store the necessary features. Thus, 1MB of memory can accommodate approximately 170k active flows. Unfortunately, the limited size of registers and the large number of flows may result in frequent hash collisions. By tracking the arrival time of the last packet and employing a timeout mechanism, it becomes possible to remove inactive flows information from the registers, hence significantly reducing collisions. Additionally, the model weights are also stored in registers, and the memory

required to store the weights of each layer depends on both the weight matrix and the applied bit width.

3.10 Conclusion

In this chapter, we presented INQ-MLT, an innovative toolbox designed to facilitate the training and deployment of machine learning models within intelligent data-planes. In INQ-MLT, the controller adopts a quantization-aware training mechanism to construct and train an ML model. The proposed mechanism compensates for the accuracy loss due to quantization by enforcing several quantization and de-quantization steps during the training process. Once trained, the weights and activations of the trained model are quantized and mapped from their floating point representations to more compact integer representations. A P4 program integrates the quantized model inference is compiled and deployed to the IDP. Simulation results demonstrated the feasibility of deploying the quantized models and showed that they can effectively classify ongoing flows at runtime after the first few packets.

In summary, this chapter introduced the unified data-plane ML toolbox, including feature extraction, quantization, model deployment, and inference execution. The subsequent chapters extend this toolbox with specialized modules for early intrusion detection (Chapter 4), multi-task learning (Chapter 5), unsupervised drift detection (Chapter 6), and drift-aware classification (Chapter 7). Together, these contributions transform the toolbox into a comprehensive platform for intelligent, adaptive, and resource-efficient data-plane ML.

In the next chapter, we leverage this toolbox to develop an intrusion detection system for the data-plane, capable of identifying malicious flows both in the early and later stages.

Chapter 4

A Two-Stage Confidence-Based Intrusion Detection System in Programmable Data-Planes

4.1 Introduction

The widespread expansion of computer networks and their new emerging applications have enabled attackers to launch various security attacks [118]. Intrusion Detection Systems (IDSs) play a crucial role in identifying and responding to potential cyber threats. Machine Learning (ML)-based approaches have increasingly been developed for enhancing IDSs. However, traditional ML-based IDS solutions typically deploy models at the control-plane, requiring frequent interactions between the data-plane and the controller for flow analysis. This scheme leads to decision delays that hinder timely detection of malicious traffic.

Intrusion detection can increasingly benefit from programmable data-planes, which enable low-latency, in-network ML inference directly within switches. Building on the data-plane ML toolbox introduced in Chapter 3, we now develop a confidence-based intrusion detection system tailored for programmable switches. This chapter leverages the quantization-aware deployment pipeline described earlier but augments it with a multi-phase decision mechanism that balances early detection speed with high-confidence classification.

A key challenge in this design is determining when to trigger detection—at an early stage with limited information or after observing more packets from the same flow. By observing an extra packet in a given flow, the classification accuracy may either increase or

remain unchanged since additional information is provided [52]. Moreover, intelligent IDSs must not only be accurate but also should indicate when they are likely to be incorrect. More precisely, an ML-based anomaly detection model should not only make accurate predictions but also provide a quantification of prediction uncertainty [119]. Calibration refers to the process of ensuring a model's confidence aligns with its correctness. For a well-calibrated model, predictive scores, i.e., the level of confidence, should be indicative of the actual likelihood of correctness.

Convolutional Neural Networks (CNNs), a specific ML model, have grown in popularity in certain network service scenarios such as DDoS attack detection [88] and network traffic analysis [90]. These successes are attributed to the benefits of CNN with respect to reduced feature engineering and high detection accuracy. However, implementing CNN models in the programmable data-plane that enables fast detection and considers its trustworthiness is challenging.

In this chapter, we propose a novel in-network, **T**wo-**S**tage **C**onfidence-based **IDS** (TSCIDS), where the controller trains two context-dependent CNNs, mapped to the early and later phases of a flow, respectively. The flow is labeled in the early phase when the prediction confidence reaches a predefined threshold. To simplify the execution of the two CNNs, parameter sharing [42] is utilized in the training process, which not only saves memory but also accelerates the inference process. Our main contributions are summarized as follows.

- We propose a novel IDS framework, TSCIDS, that deploys two CNN models in the early and later phase of flows respectively within the data-plane. The early-stage CNN generates an initial label, while the later-stage CNN is triggered when this first labeling is not certain enough.
- To efficiently implement the proposed IDS in the data-plane, we propose a training scheme that utilizes transfer learning to train two CNNs sharing the same convolutional layer, allowing the later-stage CNN to leverage the early-stage CNN's hidden state, resulting in the reduction of memory usage and inference time.
- To improve the reliability of the trained model, we employ a post-hoc calibration method to enhance the reliability of the provided predictions.
- We design a packet processing pipeline to integrate the trained CNN models within data-planes.

The remainder of this chapter is organized as follows; Section 4.2 discusses related work. Section 4.3 provides an overview of the proposed architecture while Section 4.4 describes

the implementation details. Section 4.5 gives the simulation result. Finally, Section 4.6 concludes this chapter.

4.2 Related Work

Traditional ML algorithms such as Support Vector Machines (SVM), k-Nearest Neighbor (kNN), Random Forest (RF), and Naïve Bayes (NB) have been widely applied to IDSs with notable success [120]. In recent years, CNNs have also been extensively utilized for IDSs. Wang et al. [121] propose a malware traffic classification method using CNN by transforming the first 784 bytes in packets into images. Doriguzzi-Corin et al. [88] propose a CNN-based DDoS detection architecture with low processing overhead and attack detection time. Li et al. [90] propose a transfer learning and ensemble learning-based IDS for Internet of Vehicles systems using CNNs. CNNs have proven advantageous in IDSs for their ability to classify attacks by analyzing the relationships among features while requiring minimal data preprocessing. In general, these approaches cannot perform line-rate detection because their models are not deployable within the data-plane.

Recent research efforts have explored intelligent data-planes that offload ML inference tasks to programmable switches or SmartNICs, enabling real-time decision-making. Such frameworks [1, 48, 50] can be adapted for IDS scenarios, yet most existing works overlook the reliability of early predictions, particularly when only partial flow information is available. Our work extends these approaches by implementing CNN models in the intelligent data-plane to perform anomaly detection in different phases of a flow. Furthermore, the reliability of early detection results is considered due to the limited packet information available in the early phase.

4.3 Framework Overview

This section presents an overview of the proposed framework, illustrated in Figure 4.1. The control-plane plays a crucial role in building and training an intelligent IDS based on CNN models, which is subsequently offloaded to the data-plane. The data-plane utilizes the trained models to detect attack packets as part of the forwarding pipeline. The controller continuously collects monitored traffic data from the data-plane, enabling periodic retraining of the CNN models.

The traffic data is fed first into a traffic preprocessing module to transform the needed header information contained in the traffic flows into array-like data structures. To make

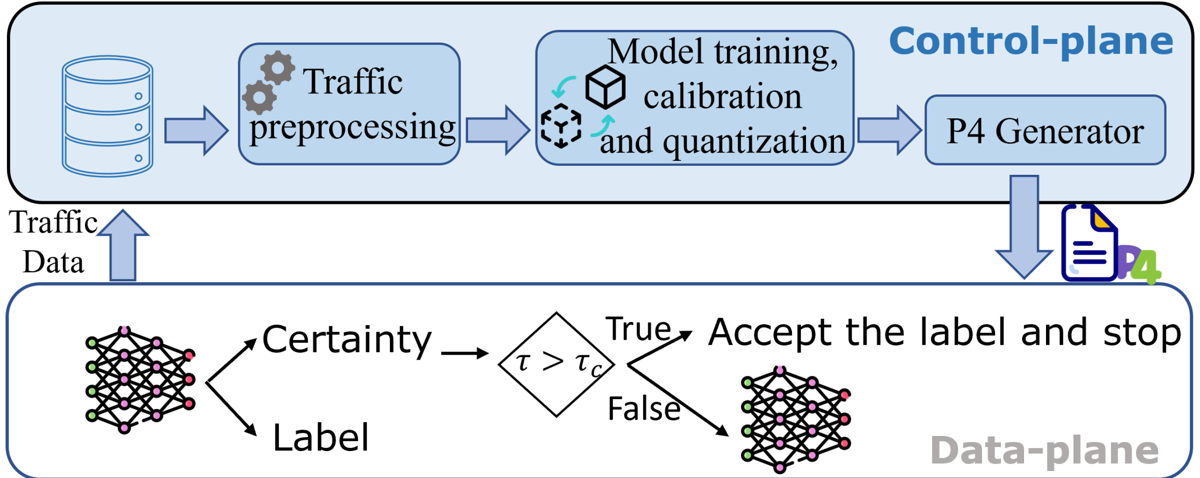


Figure 4.1: TSCIDS architecture.

the trained model provide a reliable prediction probability, after training the CNN models, a post-hoc calibration method is employed. To generate models that are suited for deployment in the data-plane, the quantization process is performed, which maps the weights and activation function values of the trained models from their floating point representations to integer representations. The details of the quantization module are introduced in our previous work [15]. The quantized CNN models are then fed to the P4 generator which produces the corresponding P4 program. Finally, the generated P4 program is compiled for a specific target.

When performing early detection with limited packet information, it is important to consider the reliability of the prediction. Therefore, early detection needs to predict not only whether a flow is benign or malicious but also provide predictive certainty τ. TSCIDS implements two CNNs in the data-plane, an early-stage CNN and a later-stage CNN. TSCIDS classifies the flow if the early-stage CNN gives a high certainty of the detection result. In other words, if the certainty τ exceeds a predefined threshold τ_c, the detection result will be accepted. Otherwise, to ensure high detection accuracy, the framework should wait for more packet information to execute the later-stage CNN.

Although we only present a two-stage detection scheme in this chapter, we envision that the detection process can be implemented at a more fine-grained level, i.e., setting different thresholds on multiple phases of a flow at multiple switches.

4.3.1 Network Traffic Preprocessing

We identify a flow $\mathcal{F}$ as a sequence of packets $[\mathcal{F}[i]$ for $i \in [0, 1, \ldots, |\mathcal{F}| - 1]]$ having the same flow ID (e.g., source and destination IP addresses, source and destination ports, and transport-level protocol), where $\mathcal{F}[i]$ denotes the ith packet of the flow. A subflow is denoted as $\mathcal{F}[i : j]$ to represent the packet sequence $[\mathcal{F}[i], \mathcal{F}[i + 1], \ldots, \mathcal{F}[j - 1]]$ and consequently the first n packets of a flow are denoted by $\mathcal{F}[0 : n]$ or simply $\mathcal{F}[: n]$, where $n < |\mathcal{F}|$. Sets of subflows are denoted as $\mathcal{D}[i : j] = \{\mathcal{F}_1[i : j], \mathcal{F}_2[i : j], \ldots \mathcal{F}_{|\mathcal{D}|}[i : j]\}$, where $|\mathcal{D}|$ denotes the size of this set.

We use packet-level features instead of flow-level features for two reasons. First, the flow-level features, such as average packet length, are highly abstracted, making it impossible to implement fine-grained operations (such as learning the relationship between two packets). Second, it is advantageous to use packet-level features to achieve real-time detection.

When the traffic is collected, we extract features from packet headers. Then each flow, identified by its flow ID, is transformed into a matrix $\mathcal{E}$ of size $n \times m$, where n is the number of packets required for the classification and m is the number of features. This preprocessing allows CNNs to learn the characteristics of malicious and benign traffic by sliding convolutional filters over such input to identify salient patterns.

4.3.2 The Convolutional Neural Network Design

The CNN consists of an input layer, a convolutional layer, a flatten layer, and an output layer. The network structure is shown in Figure 4.2.

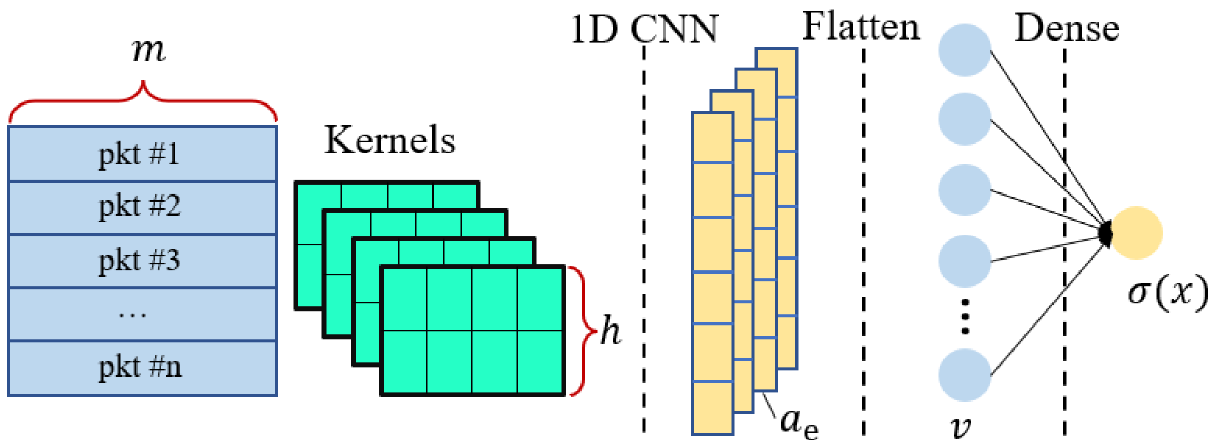


Figure 4.2: Convolutional neural network structure.

Input layer: After the traffic preprocessing module, each traffic flow is reshaped into a 2-D matrix of packet features. The first layer takes the 2-D matrix $\mathcal{E}$ of size $n \times m$ as input. The matrix $\mathcal{E}$ contains the first n individual packet vector, such that $\mathcal{E} = (pkt_1, ..., pkt_n)$ where pkt_n is the nth packet vector in flow, and the length of each packet vector is m.

Convolutional layer: A single convolutional layer with k filters of size $h \times m$, also known as kernels, operates on each input matrix. These filters convolve over $\mathcal{E}$ with a step size of 1 to extract and learn local features that contain valuable information for detecting malicious and benign flow. Each of the k filters generates an activation map a of size $(n - h + 1)$, such that $a_e = ReLU(\mathcal{E} * W_e^c + b_e^c)$, where $e = 1 \cdots k$, $\mathcal{E} * W_e^c$ denotes convolution of $\mathcal{E}$ and W_e^c, and W_e^c and b_e^c are the weight and bias parameters of the eth filter that are learned during the training stage. The rectified linear activation function $ReLU(x) = max\{0, x\}$ introduces non-linearity among the learned filters. All activation maps are stacked, creating an activation matrix A of size $(n - h + 1) \times k$, such that $A = [a_1 |...| a_k]$. We do not implement a pooling layer since P4 does not support maximum or minimum operations.

Flatten layer: This layer is used to convert the activation matrix A into a single one-dimensional vector v. This allows the subsequent fully connected layers to process the features and make classifications.

Output layer: v is fed into a fully-connected layer of the size $|v|$. The output layer has a sole node. The output x is generated after performing the dot product operation $x = v \cdot W^d + b^d$, where W^d and b^d are the weight and bias of the fully-connected layer. The output x is passed to the sigmoid function such that $\sigma(x) = 1/(1 + e^{-x})$. It returns a value between 0 and 1, representing the probability that the flow is malicious. If this probability is greater than 0.5, the flow is classified as malicious; otherwise, it is classified as benign.

4.3.3 Training Scheme and Calibration Method

Training

The early-stage CNN, denoted as CNN_α, is trained on the set of subflows $\mathcal{D}[0 : \alpha]$. In other words, CNN_α will make an initial prediction using the first α packets of a flow. The later-stage CNN that is trained with $\mathcal{D}[0 : \beta]$ is denoted as CNN_β, where $\beta > \alpha$.

In our training scheme, the weights are divided into two types: trainable and non-trainable weights. The first are the parameters that can be adjusted during the training process to improve the accuracy of the model. Non-trainable weights, on the other hand,

are fixed and do not change during the training process. For the training of early-stage CNN, all parameters are trainable, and back-propagation is applied to reduce the error. To build the later-stage CNN, we first initialize it by copying the convolutional layer of the early-stage CNN and adding a new fully-connected layer. The weights of the convolutional layer are set as non-trainable, and the weights of the fully-connected layer are set as trainable. We then train the later-stage CNN using the set of subflows $\mathcal{D}[0:\beta]$, freezing the weights of the convolutional layer and only allowing the weights of the fully-connected layer to get updated.

Calibration

The detection of malicious flow is a binary classification problem where label $\ell \in \{0,1\}$, i.e., malicious flow (positive class) and benign flow (negative class). The probability associated with the predicted class label should reflect its ground truth correctness likelihood. How reliable its prediction is depends on whether the classifier is calibrated or not. A non-calibrated classifier can produce predictive confidence that is excessively optimistic or pessimistic in relation to its decisions. On the other hand, a confidence-calibrated classifier ensures that the predicted probability of each sample, denoted by $\hat{p}$, is equal to $\Pr\{\ell = 1 \mid \hat{p}\}, \hat{p} \in [0,1]$. For example, given 100 predictions, each with confidence of 0.9, we expect that 90 should be correctly classified.

The calibration of the model can be visually represented by using reliability diagrams [119] as follows: partition the prediction probability into Φ equally-spaced bins and compute the relative-frequency of positive samples of each bin. Let B_ϕ be the set of evaluated samples such that the confidence associated with the predicted label falls within the interval $I_\phi \triangleq \left(\frac{\phi-1}{\Phi}; \frac{\phi}{\Phi}\right]$, where $\phi \in \{1, \cdots, \Phi\}$. The relative-frequency of positive samples $\operatorname{freq}(B_\phi)$ and confidence $\operatorname{conf}(B_\phi)$ of this bin can be computed by:

$$\operatorname{freq}(B_\phi) = \frac{1}{|B_\phi|} \sum_{\mu \in B_\phi} \mathbb{1}\left(\ell_{(\mu)} = 1\right) \tag{4.1}$$

$$\operatorname{conf}(B_\phi) = \frac{1}{|B_\phi|} \sum_{\mu \in B_\phi} \hat{p}_{(\mu)} \tag{4.2}$$

where $\ell_{(\mu)}$ is the true label for the μth sample, and $\hat{p}_{(\mu)}$ denotes the predicted confidence of the μth sample.

A concise metric of the deviation from perfect calibration is Expected Calibration Error (ECE) [119], defined as $\mathbb{E}_{\hat{p}}[\mid \Pr\{\ell = 1 \mid \hat{p}\} - \hat{p} \mid]$. This metric represents the expected

absolute deviation between the confidence and the confidence-conditional frequency of positive samples, which can be approximately calculated as:

$$\mathrm{ECE} \approx \sum_{\phi=1}^{\Phi} \frac{|B_\phi|}{N} \left|\mathrm{freq}\left(B_\phi\right) - \mathrm{conf}\left(B_\phi\right)\right| \tag{4.3}$$

where N is the total number of test samples.

Histogram Binning [122] is applied to calibrate the model. All the uncalibrated predictions of validation samples are divided into mutually exclusive Δ bins $\{B_\delta\}_{\delta=1}^{\Delta}$ according to a set of intervals $\{\theta_\delta\}_{\delta=1}^{\Delta+1}$ which uniformly partitions the probability interval $[0, 1]$. By allowing the value of Δ to be larger than the value of Φ used to draw reliability diagrams, a fine-grained calibration can be achieved. Each bin is assigned a confidence score η, which can be simply set to the relative frequency of positive samples $\mathrm{freq}\left(B_\delta\right)$ in each bin. If the uncalibrated confidence falls into bin B_δ, then the calibrated confidence is η_δ.

4.3.4 Two-Stage Inference based on Reliability

Because the output layer only has a single node with the sigmoid activation function, the output value is treated as the confidence score directly. TSCIDS requires the developer to provide a score threshold τ_c, where $0.5 \leq \tau_c \leq 1$, which considers the tradeoff between accuracy and detection speed. A higher threshold is chosen if the detection accuracy is the primary concern, while a lower threshold is chosen if the goal is to obtain the label as quickly as possible.

The early-stage model generates a potential label $\ell \in \{0, 1\}$ for the flow, along with its corresponding level of certainty τ, i.e., confidence score. If the confidence score τ is above τ_c, the label is accepted, memory freed, and the flow is no longer tracked. Otherwise, if τ is below the threshold τ_c, TSCIDS continues to track the flow and waits for additional packets. When the invocation condition of the later-stage model is met (i.e., the βth packet arrivals), it executes with additional packet information, and the resulting output label is accepted directly. We will elaborate further on the P4 implementation specifics of our CNN models in the subsequent section.

4.4 ML Model Mapping To P4

In this section, we explain how our proposed models are integrated within the P4 pipeline. The packet processing pipeline is illustrated in Figure 4.3. We define three types of regis-

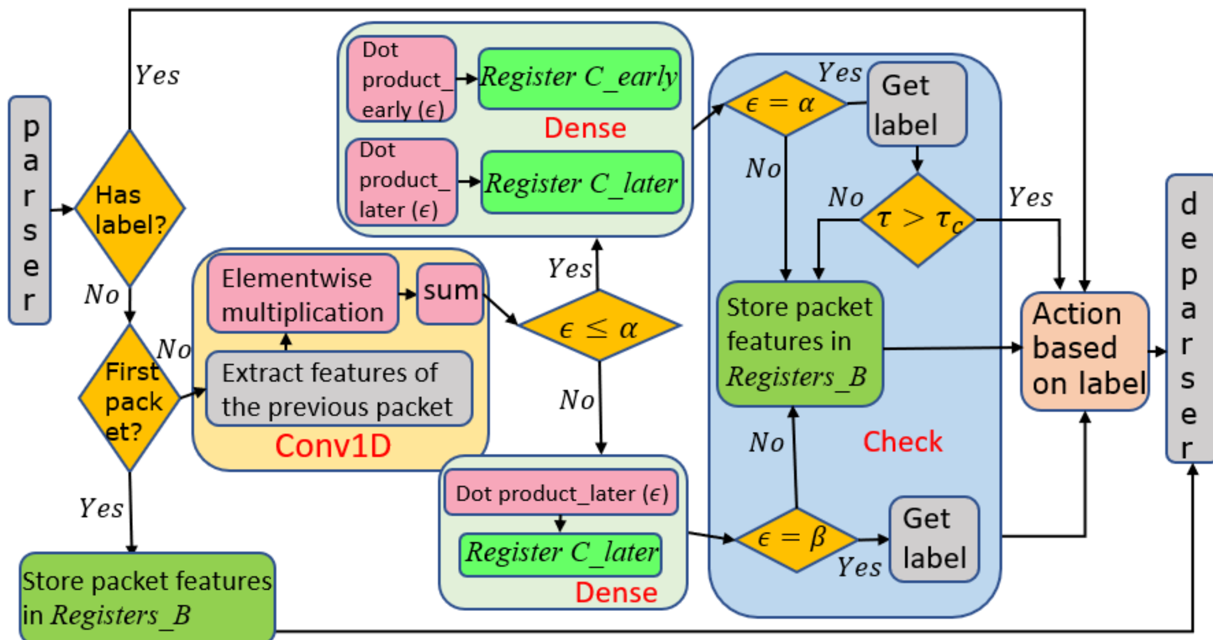


Figure 4.3: Packet processing pipeline.

Table 4.1: Registers with their functionality

Type	**Description**
Register_A	Counting the number of packets within individual flows.
Register_B	Storing crucial packet information pertaining to the preceding packet of each flow.
Register_C_early	Repository for the stages of results of early-stage CNN.
Register_C_later	Repository for the stages of results of later-stage CNN.

ters, each with specific functionality, as shown in Table 4.1. *Register_A* is implemented for the purpose of counting the number of packets within individual flows. Recall that filters convolve over $\mathcal{E}$ with a step of 1, each convolution operation is performed on a matrix that contains two adjacent packets. Therefore, *Register_B* are designated for storing crucial packet information pertaining to the preceding packet of each flow. Finally, *Register_C* serve as repositories for the stages of results and are directly utilized when generating the final output. *Register_C_early* and *Register_C_later* are responsible for storing the results of two CNNs in the early stage and later stage respectively.

Parser and Check Status

For an arriving packet, it is first mapped into a Packet Header Vector (PHV) by the parser. Then the PHV is passed to the check status module which examines incoming packets to determine if they contain a label. If a label is present, the system executes the corresponding action (e.g., drop the packet). Otherwise, the incoming packet is examined to ascertain whether it belongs to a new flow by checking *Register_A*. If it is the first packet of a new flow, the module records necessary packet fields and stores them in *Register_B*. Otherwise, the module retrieves the features of the preceding packet from *Register_B* and executes the Conv1D module for necessary computations.

Conv1D Implementaion

Prior to assigning labels to packets, all packets, except for the first packet of each flow, will undergo the CNN inference process. The CNN inference process comprises two distinct stages, namely the Conv1D stage and the dense layer stage. In the Conv1D stage, packet information from the preceding packet is extracted from *Register_B* and combined with the current packet information to form a matrix. Subsequently, the elementwise multiplication operation is performed between the matrix and the kernels. Then the resulting matrix is subjected to a summation operation to prepare for the dense layer stage.

Dense Layer Implementation

Before the packet enters the dense layer stage, which set of weights is utilized needs to be determined, i.e., to check if the number of received packets of a flow ϵ reaches the preset value α, where α is the required number for label generation of the early-stage CNN. If $\epsilon <= \alpha$, then two dot product operations between the Conv1D output and the weights of both the early-stage CNN's dense layer and the later-stage CNN's dense layer are performed respectively. Otherwise, if $\epsilon > \alpha$, the dot product operation using the dense layer weights of the later-stage CNN is performed.

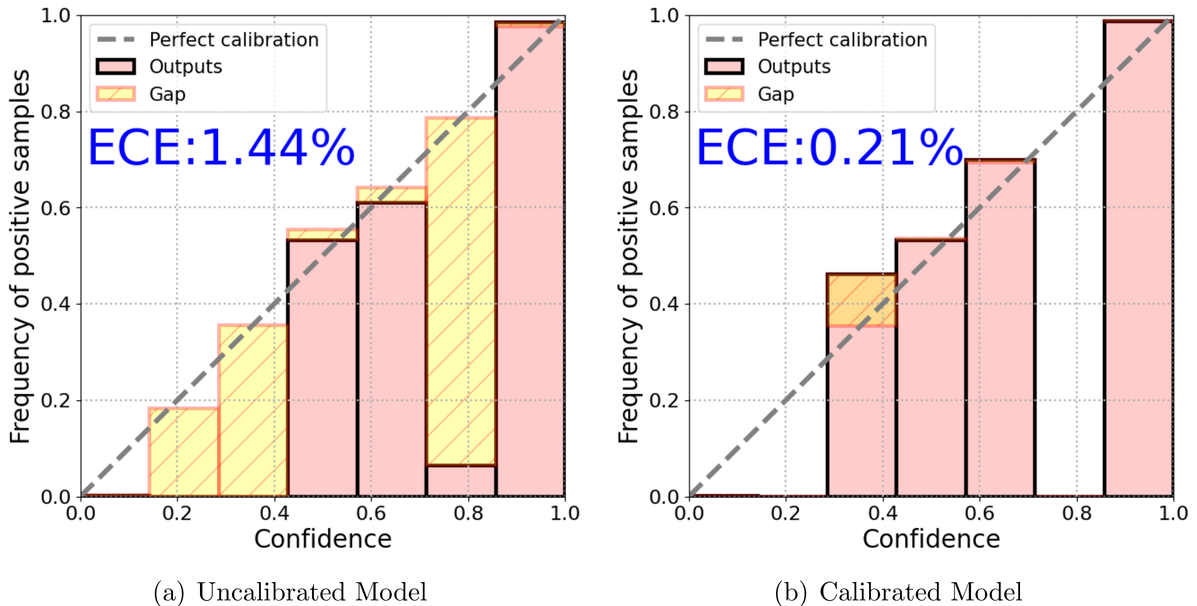
Only a part of the weight vector is used to execute the dot product operation. The parameters used in the dot product operation can be regarded as a function of the packet's position within the flow sequence, i.e., ϵ. Then the dot product results are updated in *Register_C_early* and *Register_C_later* respectively. *Register_C_later* is ready to be used if the later-stage CNN is called to make the decisions.

Check Module

In the early stage, if the number of received packets satisfies the required number for label generation, i.e., $\epsilon = \alpha$. The label is generated by evaluating the outcome stored in *Register_C_early*, followed by assessing whether the generated confidence score τ exceeds the threshold τ_c. Once the label is acceptable, it is stored in the metadata. Otherwise, the packet features are stored in *Register_B*, and the next packet of the same flow will enter the later stage. In the later stage, with the arrival of more packets and the number of received packets reaching the preset value β, i.e., the required number for later-stage label generation, the label is generated through an evaluation of the result stored in *Register_C_later*, and acceptance of this label is granted directly.

Action and Deparser

TSCIDS can apply arbitrary actions based on the label and certainty. If the packet is labeled as malicious, we can drop the packet or send it to a deep packet inspection module. Finally, for all normally forwarded packets, the deparser assembles the packet data back into a stream of bytes for transmission.



(a) Uncalibrated Model

(b) Calibrated Model

Figure 4.4: Reliability diagrams for CNN3.

4.5 Performance Evaluation

In this section, we first describe the dataset and simulation setting. Then we perform the calibration assessment. Finally, we discuss the experimental results.

4.5.1 Dataset and Experimental Setting

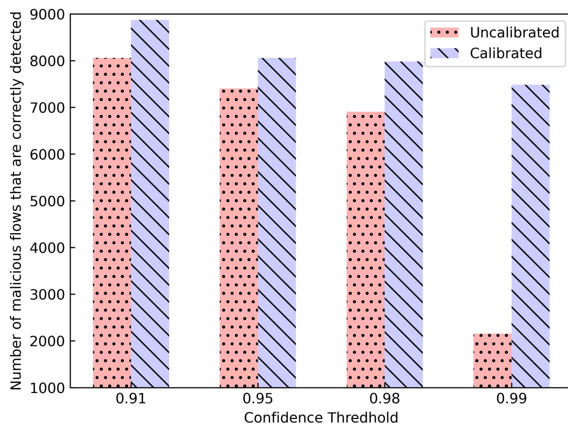
The CICIDS2017 [108] dataset consists of data collected from several days of network activity, including benign and common attack traffic. We select traces captured on three specific days: Tuesday, Wednesday, and Friday, which include Brute Force, DoS, and DDoS attacks. Given that the dataset is highly unbalanced, we follow [88] and employ an under-sampling technique to randomly remove records from the dominant class to ensure that our learning and classification processes were unbiased.

The CNN performance was evaluated using accuracy, recall, precision, and F1-score. The training, validation, and test sets are divided in a ratio of 6:2:2, respectively. The number of kernels is set to 16. Model training, validation, and quantization are performed using TensorFlow Lite. The early-stage CNN is trained based on the first 3 packets, denoted by CNN3. The later-stage model is trained based on the first 5 packets, denoted by CNN5. We use nine packet-level features to train the model: TCP flag (SYN, ACK, RST, FIN and PUSH), IP flag, packet length, TCP window size, and time to live.

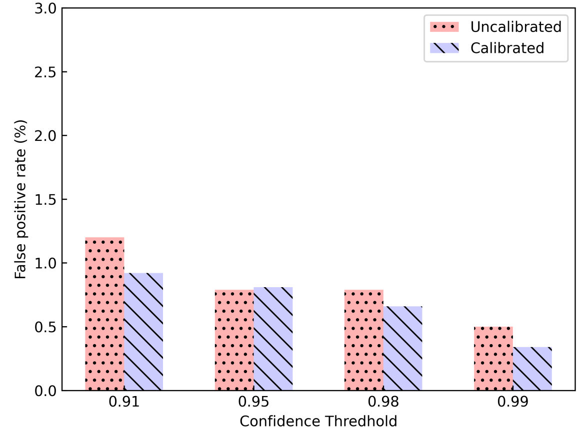
4.5.2 Results

Figure 4.4 shows the reliability diagrams and ECE error of the uncalibrated model and the calibrated model, where the prediction confidence is divided into 7 bins. The obtained diagram is compared with the $\Pr\{\ell = 1 \mid \hat{p}\} = \hat{p}$ identity line, represented by the dashed grey line. Gaps represent the mismatch between the prediction confidence and the frequency of positive samples. We can see that the calibration effect of the histogram binning is evident. Histogram binning achieves a $7\times$ ECE reduction with respect to the uncalibrated case.

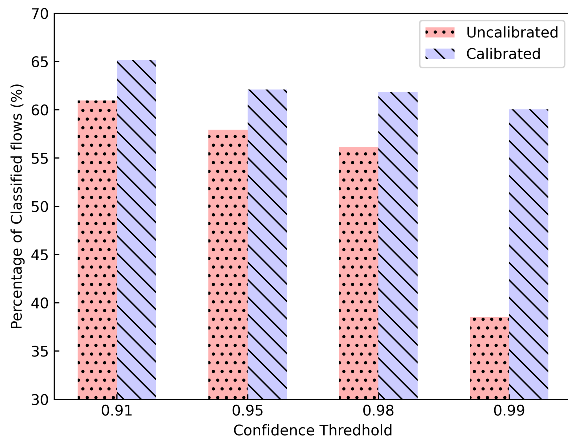
Figure 4.5 illustrates the performance comparison for both the calibrated and uncalibrated models under various threshold settings for CNN3. Specifically, Figure 4.5(a) gives the number of malicious flows that are correctly detected, and Figure 4.5(b) gives the false positive rate. Figure 4.5(c) shows the percentage of already classified flows, and Figure 4.5(d) gives the accuracy for these already classified flows (i.e., flows whose confidence reaches the threshold). Compared to the uncalibrated model, at the same threshold level, the calibrated model identifies a higher number of malicious flows while maintaining an



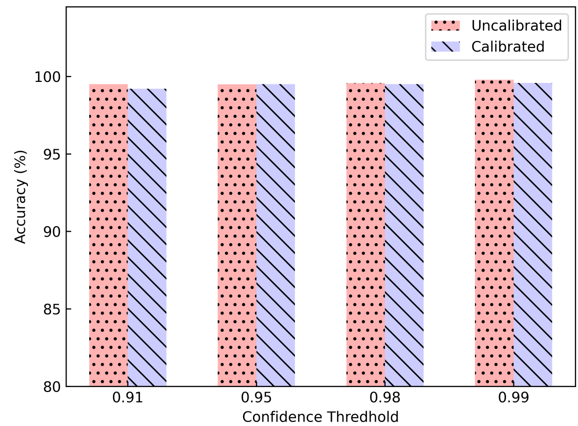
(a) Number of malicious flows that are correctly detected.



(b) False positive rate.



(c) Percentage of classified flows.



(d) Accuracy for those already classified flows.

Figure 4.5: Performance comparison for the calibrated and uncalibrated CNN3 under various threshold settings.

almost identical false positive rate, as evidenced by Figures 4.5(a) and 4.5(b). Similarly, Figures 4.5(c) and 4.5(d) show that compared to the uncalibrated model, the calibrated model classifies more flows with the same level of accuracy. This can be attributed to that the uncalibrated model is under-confident during this confidence range, whereas the calibration method rectifies this by adjusting the level of confidence of the model. From Figures 4.5(a) and 4.5(c), we also note that as the threshold value increases, the amount of traffic that can be classified in the early stage subsequently decreases.

Figure 4.6 provides a comparison of the performance of implementing only an early-stage CNN, only a later-stage CNN, and the proposed two-stage scheme. Compared to only implementing an early-stage CNN, using the two-stage scheme can improve the overall

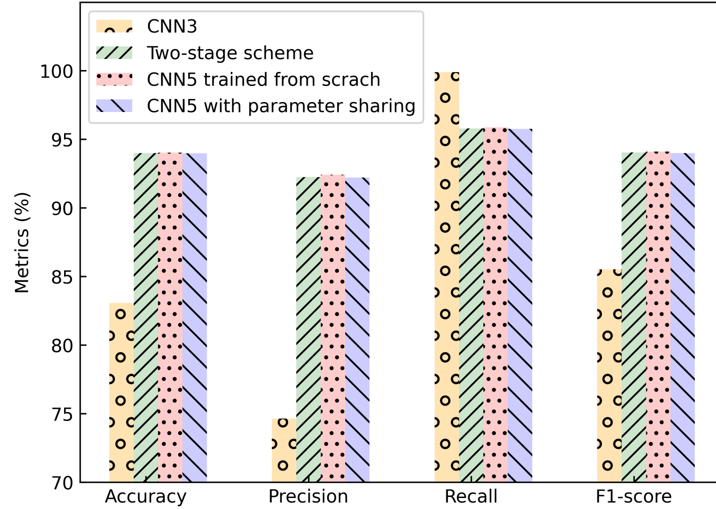


Figure 4.6: Performance comparison for only implementing CNN3, only implementing CNN5, and the two-stage scheme.

detection performance because flows that the early-stage CNN does not have enough confidence to identify are left for detection in the later stage. The inclusion of two additional packets leads to higher detection accuracy because they provide extra information. Comparing the two CNN5 models trained with different schemes, it is evident that employing parameter sharing resulted in almost no performance degradation. Compared to deploying two independent models, sharing convolutional layers can reduce the amount of memory needed to store model parameters, resulting in approximately half memory usage.

4.6 Conclusion

In this chapter, we propose TSCIDS which detects attack flows during the early or later phases of a flow by setting a confidence threshold. The controller constructs and trains two CNNs using a parameter sharing mechanism to save memory usage. The early-stage model is calibrated to adjust the prediction probabilities, ensuring it is neither over-confident nor under-confident. Experiment results show that more flows can be classified in the early stage by using the calibrated model.

Chapter 5

Design, Implementation, and Deployment of Multi-Task Neural Networks in Programmable Data-Planes

5.1 Introduction

Programmable devices and P4 [25] enable ML inference directly in the data-plane, offering line-rate analysis with ultra-low latency [32]. Both tree-based models [50, 52, 53, 72] and neural networks [57, 73, 99] deployed on SmartNICs or switches have been used to perform tasks such as attack detection [123], traffic classification [50], bot detection [124], and heavy flow analysis [51].

However, these studies address only a single network management task at a time, using one deployed model. Consequently, addressing multiple management tasks necessitates the deployment of multiple independent models. For example, ensuring Quality of Service (QoS) and optimizing network resource allocation require various management tasks, including traffic class prediction, bandwidth prediction, flow size prediction, and duration prediction [3]. However, due to the limited resources of network switches, deploying individual models for each task can exhaust or even exceed all the switch resources [1]. Additionally, some tasks are considered as *hard-to-label* tasks, which refer to tasks where accurately labeling collected traffic data is challenging. For example, labeling traffic classes is a labor-intensive process that requires domain experts to manually inspect and classify packets or flows based on observed patterns and behaviors. This approach is slow, expen-

sive, and difficult to scale, especially as network data volumes continue to grow. While automatic labeling schemes [125] exist, they typically rely on predefined rules or machine-generated patterns, which may fail to capture the complexity of real-world network traffic. These methods also struggle with ambiguous or mixed traffic types, leading to misclassifications and reduced model accuracy. Consequently, training models for such tasks with insufficient or low-quality labeled data often results in poor model performance.

Multi-task learning (MTL) [63] emerges as a promising solution to these issues. MTL enables the simultaneous execution of multiple related tasks by leveraging shared feature representations, providing two key advantages for in-network ML. First, a single multi-task model can replace multiple standalone models, significantly reducing the resource burden on network switches. Second, tasks with abundant and easily obtainable labels can supplement the training of *hard-to-label* tasks by contributing shared representations, thereby improving their accuracy. Thus, integrating MTL into the data-plane not only optimizes resource utilization but also improves the performance of tasks with insufficient labeled data.

Neural network architectures are often used to perform MTL tasks [63]. However, there are two challenges to implementing neural networks inference in the data-plane. First, the data-plane pipeline does not support complex operations required for neural network inference, such as matrix multiplication and floating-point operations. Second, the limited resources in the data-plane restrict the size of neural network models, making it difficult to deploy large models on a single switch. Given that neural network models used for MTL are usually deep, this is a notable challenge. As a result, hardware modifications were suggested to support neural network-based inference [58, 59], but these approaches restrict the direct utilization of existing switch ASICs. Fully-binarized neural networks, where binarization (or binary quantization) reduces weights and activations to binary values (typically ± 1), have been explored as an alternative [57, 73]; however, they suffer from precision degradation.

To address these challenges, this chapter proposes MUTA. First, MUTA builds an MTL neural network where tasks share multiple layers, rather than deploying multiple independent models supporting different management tasks. This approach is both resource-efficient and more accurate than single-task models (§5.7.3). Resource efficiency is achieved by sharing feature representations among related tasks, thereby eliminating redundant resource usage. Moreover, MUTA enhances accuracy in scenarios where specific tasks lack sufficient labeled data (§5.7.2) by leveraging knowledge from related tasks through shared model parameters. Second, MUTA efficiently maps model layers and associated weights to a set of off-the-shelf programmable switches using a novel deployment strategy, enabling non-binarized MTL neural network inference in the data-plane without hardware modifi-

cations (§5.5). The network-wide deployment strategy ensures that the provided service will cover the entire multi-path network, with the ability to adjust the trade-off between switch resource consumption and latency (§5.6). In summary, the main contributions of this chapter are as follows:

- We introduce MUTA, an intelligent architecture that performs multiple management tasks using MTL models in the programmable data-plane. MUTA generates a quantized MTL model suitable for deployment in the data-plane. To the best of our knowledge, this is the first work toward non-binarized multi-task model inference in the data-plane.
- We design and train a non-binarized multi-task neural network model that ensures efficient utilization of limited resources in data-planes while maintaining high accuracy when processing multiple tasks simultaneously.
- We present a novel mapping methodology for deploying the MTL model within the data-plane in a distributed manner. The model's layers can be allocated across multiple switches, and we design a novel implementation of the per-layer inference operation (i.e., vector-matrix multiplication) to enhance scalability and alleviate the resource burden on individual devices.
- To ensure that the MTL-based service does not affect existing network functions, and that correct service is provided regardless of the path taken thorough the network, we formulate the neural network layer placement problem in a multi-path network as an integer linear programming (ILP) problem and design a network-wide deployment strategy.

We evaluate the proposed solution using two use cases: video streaming quality of experience prediction and traffic characteristics prediction, showing that MTL can improve the accuracy of *hard-to-label* tasks with insufficient labels. The evaluation of MUTA on Intel Tofino switches shows that MUTA reduces memory usage by $\times 10.5$ compared to single-task models, while maintaining line-rate throughput and sub-microsecond latency. The proposed distributed deployment strategy is scalable and flexible, providing efficient distribution plans across different network scales and topologies without requiring changes to routing rules.

The remainder of this chapter is organized as follows; Section 5.2 discusses related work. Section 5.3 provides an overview of the proposed architecture while Section 5.4 describes the proposed multi-task neural network and explains the adopted quantization scheme.

Section 5.5 discusses the P4-based implementation details. Performance evaluation results are discussed in Section 5.7. Section 5.8 discusses some considerations and future research directions. Finally, Section 5.9 concludes the chapter.

5.2 Related Work

In this section, we review multi-task learning, existing in-network machine learning solutions, and highlight the key design challenges that motivate our approach.

5.2.1 Multi-Task Learning

Multi-task learning (MTL) is a machine learning training paradigm in which a shared model simultaneously learns multiple tasks under the assumption that the tasks are not completely independent and one can improve the learning of another. MTL has been successfully applied in various ML fields, including natural language processing [60] and computer vision [61] and autonomous driving [62].

MTL can be implemented using either hard parameter sharing or soft parameter sharing. In hard parameter sharing, a subset of parameters is shared across multiple tasks, while task-specific parameters are maintained separately. In contrast, soft parameter sharing employs independent models for each task, but their parameters are regularized to promote similarity and leverage commonalities among tasks [63]. In this work, we adopt the hard parameter sharing approach due to its simplicity and efficiency. Compared to the single-task case, where each individual task is solved separately by its own model, such multi-task models have several advantages. First, their inherent layer sharing leads to a substantially reduced memory footprint. Second, their resource efficiency is high, as they explicitly avoid repetitive features calculation in the shared layers.

5.2.2 Existing In-Network ML Solutions and Limitations

In-Network Tree-based Solutions

The research community has made substantial progress [14, 48, 50, 52–54, 72] in realizing tree-based inference models within programmable switches. Among the proposed methodologies, two advanced mapping schemes have been extensively explored: the depth-based mapping scheme and the encode-based mapping scheme. The depth-based mapping

Table 5.1: Comparison of advanced in-network neural network solutions.

Scheme	Model	Platform	Layer Split	Multi-path Distributed	Without Recirculation	Multi-task
N2Net [112]	binarized NN	RMT-like Switch	✘	✘	✓	✘
BaNaNa Split [126]	binarized NN	SmartNIC	✓	✘	✓	✘
N3IC [73]	binarized NN	SmartNIC	✘	✘	✓	✘
Qin et al. [57]	binarized NN	BMv2 (software)	✘	✘	✓	✘
NNSplit [74]	binarized NN	BMv2 (software)	✓	✘	✓	✘
MARTINI [127]	binarized NN	BMv2 (software)	✘	✘	✓	✓
BoS [99]	binarized RNN	Tofino (hardware)	✘	✘	✓	✘
INQ-MLT [16]	non-binarized NN	BMv2 (software)	✘	✘	✓	✘
IOI [59]	non-binarized NN	Modified ASIC	✘	✘	✓	✘
Taurus [58]	non-binarized NN	Modified ASIC	✘	✘	✓	✘
Razavi et al. [101]	non-binarized CNN	Tofino (hardware)	✓	✘	✘	✘
MUTA	non-binarized NN	Tofino (hardware)	✓	✓	✓	✓

schemes [50,52,53] follow a natural strategy, which involves mapping the hierarchical structure of the decision trees to the programmable switch pipeline. This requires at least one (and possibly more) stages per tree level. Consequently, tree depth is bottlenecked by the number of pipeline stages. The encode-based mapping scheme [1] overcomes this limitation by partitioning the input feature space and leveraging feature tables to encode individual feature values. The encoded feature space is then mapped to labels using a decision table. This scheme allows feature tables to share stages, significantly enhancing scalability and enabling the deployment of deeper and more complex trees.

However, current tree-based solutions require separate tree models to be deployed for different tasks, which significantly increases resource consumption within the data-plane. As each branch in a tree model is formed based on features relevant to a specific task, it is difficult to share branches or nodes across different tasks. This structural rigidity means that tree models do not naturally support the sharing of information between tasks. Additionally, tree-based models struggle with tasks that have limited training data.

In-Network Neural Network Solutions

Table 5.1 summarizes existing in-network neural network schemes. The implementation of Binary Neural Networks (BNNs) in the data-plane has been explored using commodity SmartNICs (e.g., N3IC [73] and BaNaNa Split [126]), and software switches BMv2 (e.g., Qin et al. [57]). These works binarize both the weights and the activations of a Multi-Layer Perceptron (MLP) model. The forward propagation in fully-connected layers is then executed using XNOR operations and customized population count (`popcnt`) operations [57, 73]. Following this approach, MARTINI [127] implements BNN-based MTL models in software switches. However, it has not been proven that these solutions can

be effectively integrated into commercial switch Application-Specific Integrated Circuits (ASICs) while maintaining acceptable performance and scalability.

Instead of full model binarization, BoS [99] enables the use of recurrent neural network (RNN) in the data-plane by only performing binarization on activation functions. They avoid direct computations of the layer forward propagation by replacing it with a table lookup. It realizes equivalent layer forward propagation by recording a mapping from input to output bit strings in a match-action table.

As a further step toward higher precision in-network neural networks, INQ-MLT [15,16] introduces an in-network quantized ML toolbox designed to generate non-binarized neural networks for data-plane deployment. However, the solution is suitable only for targets supporting multiplication operations (e.g., software switches [34]), and not for switch ASICs. Razavi et al. [101] implement a quantized convolutional neural network (CNN) on Tofino2 switches [128] for an image classification task. Their approach decomposes each multiplication into multiple shift operations, necessitating a significant amount of recirculation. This approach results in a substantial throughput reduction and increased latency.

Orthogonal to the above solutions, Taurus [58] proposes modified switches, using custom hardware based on the MapReduce abstraction, supporting deep neural networks. Similarly, IOI [59] implements neural network inference on programmable switches by plugging a novel transceiver module. This module is designed to perform linear operations such as matrix multiplication in the optical domain. Both solutions are not applicable to commodity switch ASICs.

Neural networks are inherently suitable for MTL due to their ability to learn and share representations across multiple tasks. Neural networks utilize shared parameters within their layered architecture, enabling the extraction and sharing of useful features between tasks. This shared representation facilitates better generalization and allows the model to make efficient use of the available data from all tasks. By leveraging the shared representation, tasks with abundant labeled data can significantly enhance the performance of tasks with insufficient labeled data through shared learning [3]. Furthermore, neural networks can scale to handle large and intricate network management tasks, whereas decision trees can become computationally expensive and difficult to manage as the complexity of the tasks grows.

Although the concurrent work MARTINI [127] also explores similar MTL idea in the data-plane, our approach, MUTA, differs in the following key aspects: First, while MARTINI employs BNNs in software switches, MUTA uses higher precision neural networks and introduces a PISA-friendly mapping methodology, making it applicable in hardware switches. Second, while MARTINI focuses primarily on the resource efficiency of MTL,

MUTA additionally demonstrates the advantage of improved accuracy for *hard-to-label* tasks by leveraging shared representations. Third, MUTA supports MTL services across the entire multi-path network, providing broader coverage and scalability.

5.2.3 Design Challenges

Model Inference on unmodified switch ASIC: Programmable switch ASICs, such as Intel Tofino [33], lack support for complex operations essential for neural network inference, including matrix multiplication and floating-point arithmetic. Existing approaches to address these limitations often involve either hardware modifications [58,59] or extensive recirculation [101]. Hardware modifications, while enabling advanced operations, restrict the direct utilization of existing switch ASIC. To perform multiplication without hardware changes, one can decompose each multiplication into a number of bit shifts and addition, but this consumes excessive stage resources (e.g., an 8-bit multiplication requires 4 stages). When the pipeline cannot fit the entire inference model, packets must be cloned and recirculated within the pipeline multiple times. This approach significantly degrades throughput and latency, making it unsuitable for real-time applications. MUTA overcomes these constraints by introducing a distributed neural network mapping methodology, along with a novel layer-wise inference implementation, enabling neural network execution on unmodified switch ASICs while maintaining line-rate performance without the need for hardware modifications or excessive recirculation.

Distributed Deployment: Deploying MTL model to a single switch has a performance curb, as the resources of a single switch are limited and cannot accommodate very large models. MUTA applies a distributed processing approach to support large models, inspired by server-based distributed inference [129]. While the idea of distributing a neural network's layers in the data-plane is not new [74], two significant gaps remain: first, distribution across resource constrained switch-ASICs is significantly different to using resource-unlimited software switches. Second, unlike server-based distributed inference, where dedicated nodes are used, in network-based inference there are a lot of potential paths of packets through the network. This means that correct execution of all model's layers needs to be guaranteed, and it has to be done without changing routing rules as this may lead, e.g., to congestion on certain routes. MUTA solves both challenges, designing a deployment strategy that ensures MTL model's services correctly cover an entire network, and demonstrating it on a switch ASIC (Intel Tofino).

Our Design Goal: To develop a practical in-network MTL framework that leverages a shared neural network model to perform multiple prediction tasks efficiently, the design

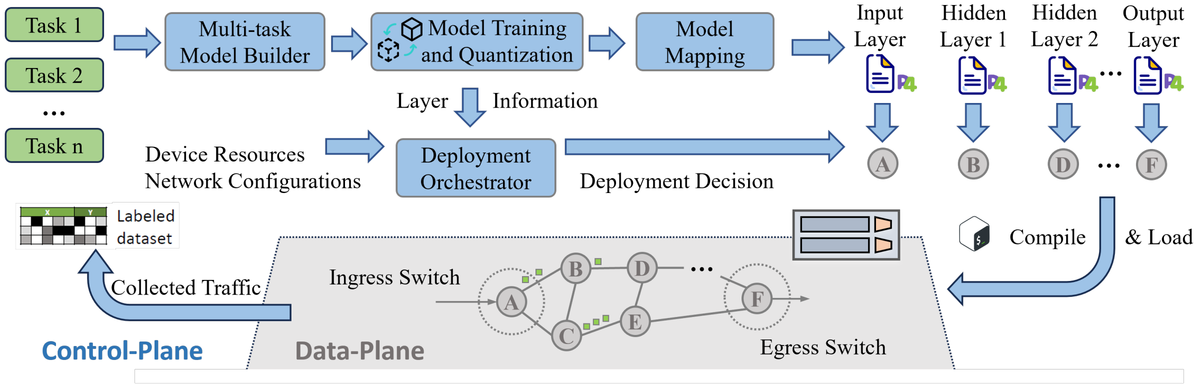


Figure 5.1: MUTA architecture. The control-plane is responsible for training and quantizing the model, generating the data-plane code, and determining the deployment strategy. The model layers are then distributed across multiple data-plane devices to cover intelligent services across the entire network.

must address the challenges of resource constraints, scalability, and deployment across programmable switches while ensuring high accuracy and maintaining line-rate performance.

5.3 An overview of MUTA

This section provides an overview of the proposed architecture. MUTA combines control-plane and data-plane components. As shown in Figure 5.1, the control-plane is responsible for building and training a multi-task neural network model for network management applications. The trained model is offloaded to the data-plane in a distributed manner. The control-plane periodically collects monitored traffic data from the data-plane and uses it to retrain the model.

Based on the application's objectives or the requirements of the network operator, a set of network management tasks is first defined. Collected raw traffic data is labeled in the control-plane (e.g., manually) to reflect these defined tasks. The relationships among tasks are then analyzed to determine their interdependencies and potential for shared learning. The labeled data is used by the *multi-task model builder* to create appropriate models. After the multi-task model is built, the *model training and quantization* module generates a quantized MTL model, with parameters prepared for mapping the model inference to data-plane program (§5.4).

Once the quantized MTL model is obtained, it is fed into the *model mapping* module to generate the data-plane P4 code. The module first splits the model layer by layer, extracting the weights from each layer, and recording the dependencies between layers.

Subsequently, it produces P4 code for each layer, mapping the model inference to match-action tables in accordance with the extracted weights. The feature extraction process, which may be either stateful or stateless, can be implemented as a standalone P4 program, and we do not prescribe a specific feature extraction mechanism. The extracted features can then be transmitted in-band together with the data packets [130]. During model inference, intermediate computation results from each layer are stored in the packet headers and passed sequentially to subsequent switches, enabling the network to execute inference in a layer-by-layer manner. The final prediction is obtained at the switch hosting the model's output layer (§5.5) and is then cached in registers for reuse; it can either trigger local actions (e.g., shaping, prioritization) or be written into the packet header for downstream processing.

A *deployment orchestrator* is used to provide a recommended deployment of the generated P4 code of the MTL model across the entire network, supporting complex multi-path network typologies and ensuring full paths coverage. It matches the resource requirements of each layer, as standalone programs (e.g., a minimum of 10 MB of memory), with the resource constraints of the target switch (e.g., 20 MB of available memory). The orchestrator analyzes layers' information and obtains their resource requirements and dependencies (e.g., layers must be completed in order). The control-plane provides the network topology and routing table, identifying all possible paths and the resources available on each switch. The orchestrator formulates an integer linear programming problem and produces a deployment strategy (§5.6).

5.4 Multi-Task Model Training and Quantization

To concurrently execute multiple network management tasks, we adopt a structured approach that leverages shared feature representations to construct a multi-task model. Typically, to train a single task, the learning model learns its own feature representations of the input data through hidden layers. Each network management task, such as traffic prediction or anomaly detection, extracts unique feature representations for its specific requirements. However, because many network management tasks share underlying traffic characteristics and patterns (e.g., packet size distribution) [131], it is feasible to learn a unified feature representation. By employing a multi-task learning framework, it is possible to train a shared model that captures these shared features, enabling more efficient and generalized learning across tasks. This shared representation not only enhances the model's ability to generalize [63] but also reduces the computational overhead associated with training separate models for each task.

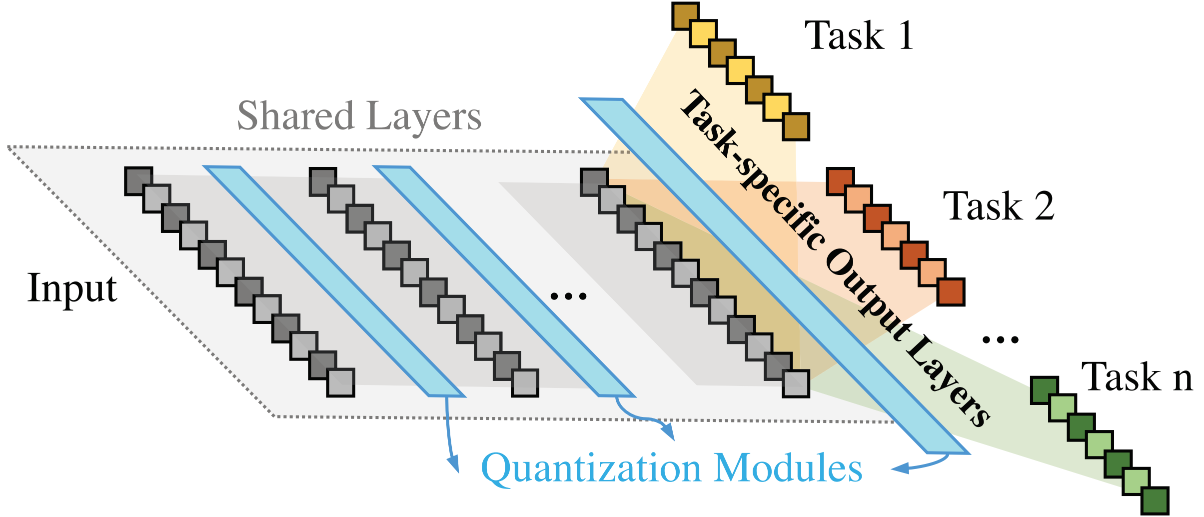


Figure 5.2: Proposed multi-task learning architecture.

5.4.1 Model Architecture and Training

The overall architecture of our multi-task model is shown in Figure 5.2. The initial layers of the multi-task neural network share common feature representations and are jointly used to execute different tasks. For the output layer, each task has its own dedicated task-specific layer, which uses the shared representation to produce task-specific outputs. Suppose we aim to train a neural network to simultaneously perform N management tasks. For each task $i \in \{1, 2, \cdots, N\}$, there is an associated loss function $\mathcal{L}_i$ and a task-specific output y_i. The objective of the multi-task learning approach can be formulated as:

$$\arg\min_{\theta} \sum_{i=1}^{N} \lambda_i \mathcal{L}_i(y_i, \hat{y}_i) \tag{5.1}$$

where $\hat{y}_i$ denotes the true label for task i. λ_i denotes the weight assigned to the loss of task i, indicating the relative importance of the task. The model parameters θ (i.e., weights and bias) are iteratively updated by back-propagation to minimize the loss function, using a combined direction derived from the gradients of each task. This joint training approach avoids the need to train separate models from scratch for each task, reduces the total number of model parameters, and thus reduces computational overhead.

5.4.2 Quantization

As data-planes cannot perform floating-point operations, the weights of each layer of the MTL model are restricted to fixed-point representations when stored in the data-plane.

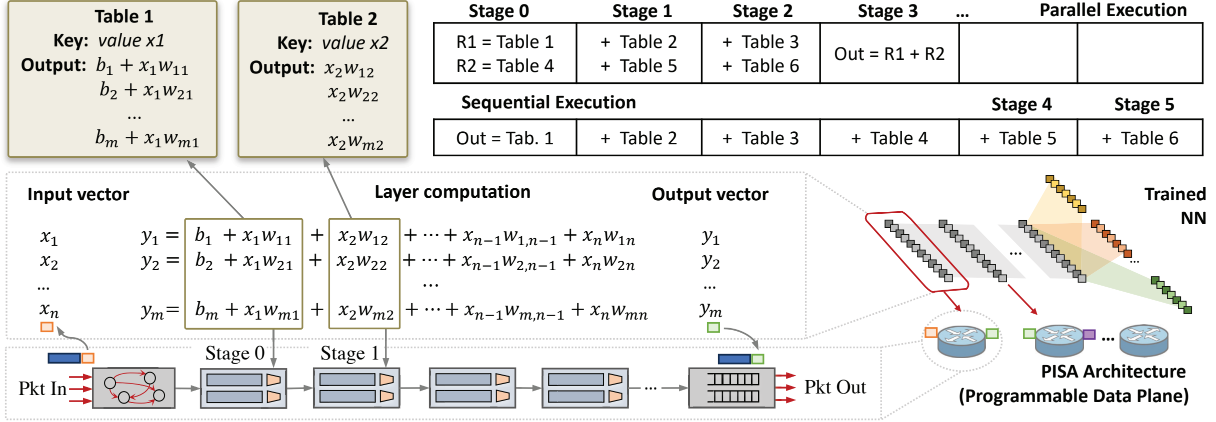


Figure 5.3: Methodology for mapping layer computation to a match-action pipeline. The parser extracts the input vector from the packet header, followed by layer-wise inference executed through a sequence of match-action tables. The deparser then reconstructs the packet, embedding the output vector into the header. These intermediate layer values are forwarded to downstream switches, which use them as inputs for their assigned layers. The top-right corner illustrates parallel execution used to minimize stage consumption.

Therefore, we employ a quantization technique to transform the floating-point based model to a quantized model which represents weights and activations using more compact format (e.g., 8-bit integers) [16]. A floating-point model parameter r is mapped to a quantized value q by defining three quantization parameters: the real-valued scale S, the zero-point Z, and the bit-width B. The scale S specifies the quantization step, or the corresponding real-value distance between two consecutive integers. The zero-point Z is an integer that ensures that real zero is quantized without error. The quantized integer q is obtained as follows,

$$q = \text{clamp}(\left\lfloor \frac{r}{S} \right\rceil + Z; Q_{min}, Q_{max}) \tag{5.2}$$

where $\lfloor \cdot \rceil$ is the round-to-nearest integer value operator. The function $\text{clamp}(q; Q_{min}, Q_{max}) = \min(\max(q, Q_{min}), Q_{max})$ ensures that the quantized value q stays within the clipping range $[Q_{min}, Q_{max}]$, which is determined by the bit-width B.

Applying quantization to a trained model may introduce a perturbation to the trained model parameters, significantly reducing the model accuracy. To mitigate this, we employ quantization-aware training (QAT) [98]. As depicted in Figure 5.2, we add quantization nodes, which are sequences of quantization and de-quantization operations stacked together. This process simulates low-precision inference time computation in the forward pass of the training process, thereby introducing the quantization induced errors to the training phase. The model is forced to learn as it is trained how to modify its weights in order to minimize its accuracy loss due to quantization. Importantly, these additional

nodes are only needed during the training phase and are not part of the inference.

After the model is trained by the control-plane and its quantized weights are obtained, each layer is mapped to corresponding match-action tables as part of the packet forwarding pipeline, as detailed in Section 5.5.

5.5 Mapping Models to Switches

In this section, we describe the implementation of the MTL model within the programmable data-plane. Deploying the entire model within a single switch limits scalability, especially for deeper models, so we decompose the model into individual layers and distribute computations across multiple switches. Each layer is implemented as a P4 program following PISA and assigned to a switch. Figure 5.3 illustrates the encoding and mapping of a layer's computations to a set of match-action tables, as explained next. Multiple layers can also be assigned to a single switch if the layer size is small. Intermediate layer results are then forwarded to subsequent switches, enabling layer-by-layer inference of the entire model.

5.5.1 Data-Plane Mapping Methodology

Layer Inference in a Single Switch

The computations within a neural network layer require multiple multiplication and addition operations. Given that switch ASICs do not inherently support multiplication operations, we replace these operations using match-action tables. These tables are used to store precomputed mappings between input values and the corresponding intermediate results, effectively replacing multiplications with table lookups.

```
1   apply {
2       action ac_input1(int<32>z11, int<32>z21,...,int<32>z81){
3           meta.R1_h1 = meta.R1_h1 + z11;
4           meta.R1_h2 = meta.R1_h2 + z21;
5           ...
6           meta.R1_h8 = meta.R1_h8 + z81;}
7       action ac_input2(int<32>z12, int<32>z22,...,int<32>z82){
8           meta.R1_h1 = meta.R1_h1 + z12;
9           meta.R1_h2 = meta.R1_h2 + z22;
10          ...
11          meta.R1_h8 = meta.R1_h8 + z82;}
12      action ac_input3(int<32>z13, int<32>z23,...,int<32>z83){
13          meta.R1_h1 = meta.R1_h1 + z13;
14          meta.R1_h2 = meta.R1_h2 + z23;
15          ...
16          meta.R1_h8 = meta.R1_h8 + z83;}
17      action ac_input4(int<32>z14, int<32>z24,...,int<32>z84){
```

```
18          meta.R2_h1 = meta.R2_h1 + z14;
19          meta.R2_h2 = meta.R2_h2 + z24;
20          ...
21          meta.R2_h8 = meta.R2_h8 + z84;}
22      action ac_input5(int<32>z15, int<32>z25,...,int<32>z85){
23          meta.R2_h1 = meta.R2_h1 + z15;
24          meta.R2_h2 = meta.R2_h2 + z25;
25          ...
26          meta.R2_h8 = meta.R2_h8 + z85;}
27      action ac_input6(int<32>z16, int<32>z26,...,int<32>z86){
28          meta.R2_h1 = meta.R2_h1 + z16;
29          meta.R2_h2 = meta.R2_h2 + z26;
30          ...
31          meta.R2_h8 = meta.R2_h8 + z86;}
32      table tb_input1 {
33          key={hdr.input1:exact;}
34          actions = {ac_input1;}
35          size=256;} // Stage 0
36      table tb_input4 {
37          key={hdr.input4:exact;}
38          actions = {ac_input4;}
39          size=256;} // Stage 0
40      table tb_input2 {...} // Stage 1
41      table tb_input5 {...} // Stage 1
42      table tb_input3 {...} // Stage 2
43      table tb_input6 {...} // Stage 2
44      // Stage 3
45      meta.output1 =  meta.R1_h1 + meta.R2_h1
46      meta.output2 =  meta.R1_h2 + meta.R2_h2
47      ...
48      meta.output8 =  meta.R1_h8 + meta.R2_h8
49  }
```

Listing 5.1: P4 code fragment demonstrating vector-matrix multiplication between an input vector of size 6 and a 6 × 8 layer weight matrix. For example, the parameters $(z_{12}, z_{22}, ..., z_{82})$ in the action *ac_input*2 correspond to the precomputed outputs $(x_2w_{12}, x_2w_{22}, \cdots, x_2w_{82})$ in Table 2 of Figure 5.3.

For example, the triggering of each layer, requires a vector-matrix multiplication operation between the input vector $\mathbf{x} = (x_1, \cdots, x_n)$ and the layer weight matrix $\mathbf{W} = [w_{mn}]$ of size $n \times m$, followed by adding the bias vector $\boldsymbol{b} = (b_1, \cdots, b_m)$, resulting in the output vector $\mathbf{y} = \mathbf{xW} + \boldsymbol{b}$. However, directly using a single match-action table to enumerate all possible combinations of inputs would result in an impractically large table, making implementation on a single switch infeasible. Therefore, we employ smaller match-action tables, dedicating one table to each input variable. Listing 5.1 provides a P4 code fragment illustrating vector-matrix multiplication for an input vector of size 6 and a weight matrix of dimensions 6 × 8.

For an input x_i, the training process provides bias and weights that are constant during the inference process. A small match-action table is then used to store the precomputed output dimensions $(x_iw_{1i}, x_iw_{2i}, \cdots, x_iw_{mi})$ for all possible values of inputs x_i. This allows x_i to act as the key in the match-action table for retrieving the corresponding parameters

used in the action function, thereby eliminating the need for multiplication operations. For example, the parameters $(z_{12}, z_{22}, ..., z_{82})$ in the action *ac_input*2 (as shown in Listing 5.1) correspond to the precomputed outputs $(x_2w_{12}, x_2w_{22}, \cdots, x_2w_{82})$ in Table 2 of Figure 5.3. The addition of bias can be integrated into any one of these tables, such as Table 1 in Figure 5.3.

The looked-up intermediate values are then used for addition operations that generates output vector $\mathbf{y}$ (i.e., the element-wise sum of vectors from all match-action tables). For an input vector of size n, the switch utilizes n match-action tables to perform the vector-matrix multiplication required for the layer inference. Once the vector $\mathbf{y}$ is obtained, a non-linear activation function is applied, expressed as $\mathbf{y}' = g(\mathbf{y})$, and realized through the clamping mechanism defined in Eqn. (5.2). For instance, when using ReLU, the operation can be represented as $\mathbf{y}' = \text{clamp}(\mathbf{y}'; Z, Q_{\max})$, where Z is the zero-point. This clamping, implemented using *if–else* conditional statements, guarantees that each element remains within the bit-width range (e.g., `uint8` values in $[0, 255]$) [98], with negative intermediate values clipped to the quantized zero-point. The resulting output vector $\mathbf{y}'$ is then written into the outgoing header and passed as input to the next switch.

Complete Model Execution Across Switches

Once a switch completes its assigned layer computation, it encapsulates the results in packet headers and forwards them to the next switch. The subsequent switch parses the headers, retrieves the intermediate data, and uses it as input for its assigned layer computations, enabling scalable deployment of MTL models across the data-plane.

When a packet arrives at the switch deploying the output layer, the final prediction result is generated after applying the activation function. Binary classification using a sigmoid activation function can obtain the label by comparing the output value to the quantized value corresponding to 0.5 using conditional statements. For a multi-class classification problem, using a ternary matching table provides better scalability for numbers comparison (i.e., the argmax operation) [99].

Regarding the final classification result, the switch hosting the output layer stores the decision in registers indexed by flow keys so that subsequent packets of the same flow can directly reuse the cached result without recomputation. This decision can then be leveraged in two ways. First, it can be written into the packet header and forwarded in-band to downstream switches or end-hosts for further use. Second, the local switch that generates the result can directly use it to trigger immediate actions (e.g., traffic shaping, prioritization, or filtering). In scenarios with early flow classification, the result is stored

in registers and reused for subsequent packets of the same flow. This dual capability ensures that results are available both for network-wide services and for local, real-time decision-making.

5.5.2 Minimizing Stage Consumption

The above description illustrates the concept of the process as a sequence of computations. However, directly implementing this in the pipeline can be highly inefficient and potentially unfeasible; Sequential dependencies between operations lead to a series of stages used on the switch, where each match-action table consumes a processing stage within the pipeline and metadata (stored in the PHV and initialized per packet) is used to pass shared information between stages. This sequential approach is wasteful, leading to an excessive number of processing stages dependent on the number of inputs (e.g., the number of features in the first layer). To overcome this constraint, we minimize stage consumption through parallel execution, as illustrated in the upper right corner of Figure 5.3.

As a simple example, assume an input vector of size 6. In a traditional sequential execution, the elements of the input vector are processed one after the other, leading to a total of 6 stages used. In contrast, a parallel execution allows to look up inputs in two or more tables in the same stage. This is achieved by dividing the input vector into two (or more) parts and processing them simultaneously. In this example, the first three elements (Table 1, Table 2, and Table 3) and the last three elements (Table 4, Table 5, and Table 6) of the input vector are processed in parallel in the first three stage (line 38-49 in Listing 5.1). This parallel computation produces two intermediate results (R1 and R2). In the subsequent stage, these two intermediate results are combined to produce the final output (line 51-54 in Listing 5.1). Thus, a computation that originally required 6 stages in a sequential approach is now completed in just 4 stages. The choice of number of lookups per stage is further discussed in §5.7.3. This method not only saves stages, but also enhances the efficiency and reduces the latency of the computation process.

Automated P4 Code Generation: As shown in Listing 5.1, the P4 code follows a highly regular structure. To facilitate efficient P4 code generation for each layer, we develop a template library containing parameterized templates for common operations. The parameters are determined by the ML model configuration or precomputed by the control-plane. MUTA offers three key parameters: the number of input nodes, the number of output nodes, and the level of parallel execution. By specifying these parameters, MUTA can automatically generate the corresponding data-plane code.

After the data-plane code is generated, the system must address two crucial distributed

deployment requirements to ensure the efficient and effective operation of the MTL models. First, the deployment must ensure the correctness and integrity of model execution. Second, it must ensure that the services provided by the model cover the entire network while utilizing as few resources as possible. These two considerations are addressed by the deployment orchestrator, which is explained in Section 5.6.

5.6 Deployment Orchestrator

To effectively distribute the layers of the MTL model across multiple switches, several requirements need to be met. First, the deployment must not affect the functionality of the network and should not require changes to routing rules. Second, the model's correct order of execution must be maintained. Third, the MTL-based service needs to cover the entire network (i.e., maintain its functionality for any set of paths). To this end, we formulate the layer-to-switch placement problem as an integer linear programming (ILP) problem and define a deployment strategy.

5.6.1 Model Formulation

Following [132], we consider a network comprising multiple programmable switches across a topology with various paths. The MTL model inference can be distributed among multiple switches by splitting the model layer by layer. Placing these layers across multiple switches is an optimization problem. Our goal is to minimize resource consumption, computation delay, and duplicated deployed layers, without impacting the network's original routing rules.

Network model

A network with $|\mathcal{S}|$ programmable switches can be represented by $(\mathcal{S}, \mathcal{P})$, where $\mathcal{S} := \{s_1, \cdots, s_{|\mathcal{S}|}\}$ denotes the set of switches. $\mathcal{P} := \{p_1, \cdots, p_{|\mathcal{P}|}\}$ denotes the set of available paths in the network. Each path $p \in \mathcal{P}$, is an ordered set of size l_p, i.e., $p = \{s_p^1, \cdots, s_p^{l_p}\}$. The chain $s_p^1 \to \cdots \to s_p^{l_p}$ represents a path from an ingress switch s_p^1 to an egress switch $s_p^{l_p}$, where l_p is the total number of switches in path p.

Resource model

Let $\mathcal{R} := \{\lambda_1, \cdots, \lambda_{|R|}\}$ be the set of resource type in the programmable switches (e.g., memory and stage). We use Q_s^λ to denote the available resource type $\lambda \in \mathcal{R}$ on switch

$s \in \mathcal{S}$.

Neural Network model

We assume the MTL model can be split into $|\mathcal{K}|$ layers. Let $\mathcal{K} := \{k_1, \cdots, k_{|\mathcal{K}|}\}$ be the set of model layers. These layers can be deployed into several switches to distribute the inference task.

Deployment Decision

Let $X_{k \to s} \in \{0, 1\}, \forall k \in \mathcal{K}, s \in \mathcal{S}$ be the deployment decision, where $X_{k \to s} = 1$ indicates layer k is deployed on switch s. If $X_{k \to s} = 1$, by executing layer k, switch s will use $\mathcal{O}_k^\lambda$ units of resource type $\lambda \in \mathcal{R}$.

The goal is to design a deployment strategy, i.e., $\{X_{k \to s}\}$, that can meet the correctness and integrity of full model execution while minimizing resource consumption and execution latency on the programmable switches.

The set of resource types $\mathcal{R}$ can include various elements such as memory, pipeline stages, header space, or compute cycles, depending on the target architecture. These resource types can be adapted based on the capabilities and limitations of the underlying platform. To compute per-layer resource consumption $\mathcal{O}_k^\lambda$, we employ a compiler-assisted profiling approach. Specifically, each layer is compiled separately using the P4 software development environment (Intel Barefoot SDE for Tofino) to obtain accurate metrics such as memory footprint and stage occupancy. These values are subsequently incorporated into the resource constraints defined in the following formulation.

5.6.2 Constraints

Dependency

For the MTL model, all $|\mathcal{K}|$ layers have to be completed in order among each path. For every path, any layers k should appear at least once before next layer $k+1$. Mathematically, if layer $k+1$ is deployed on switch s_p^e, i.e., the e-th switch of the p-th path, the deployment decision variable $X_{k+1 \to s_p^e} = 1$ and layer k has to be deployed on at least one node in set

$\{s_p^1, \cdots, s_p^{e-1}\}$, i.e.,

$$\sum_{u=1}^{e-1} X_{k \to s_p^u} \geq X_{k+1 \to s_p^e}, \forall p \in \mathcal{P}, \forall e \in \{2, \cdots, l_p\} \\ 1 \leq k \leq |\mathcal{K}| - 1 \tag{5.3}$$

Integrity

All the layers should be executed on each path to satisfy the integrity of the MTL model. Therefore, on every path p, every layer $k \in \mathcal{K}$ should appear at least once, i.e.,

$$\sum_{u=1}^{l_p} X_{k \to s_p^u} \geq 1, \forall p \in \mathcal{P}, \forall k \in \mathcal{K} \tag{5.4}$$

Resource Constraints

The available resources on each switch s must be sufficient for all deployed layers. Therefore,

$$\sum_{k=1}^{|\mathcal{K}|} \mathcal{O}_k^\lambda X_{k \to s} \leq Q_s^\lambda, \forall s \in \mathcal{S}, \forall \lambda \in \mathcal{R} \tag{5.5}$$

5.6.3 Problem Formulation

Resource Consumption

Let $\Psi_{C,\lambda}$ be the total resource cost in the network. Recall that $\mathcal{O}_k^\lambda$ is the resource type λ overhead if layer k is deployed. Therefore, $\Psi_{C,\lambda}$ can be computed as follows:

$$\Psi_{C,\lambda} = \sum_{s=1}^{|\mathcal{S}|} \sum_{k=1}^{|\mathcal{K}|} \mathcal{O}_k^\lambda X_{k \to s} \tag{5.6}$$

Latency (Number of hops)

Assume that the transmission delay on each path is fixed. We then focus on minimizing the time required to complete the MTL program, which is proportional to the number of hops. The execution latency on each path $p \in \mathcal{P}$ can be computed by checking the index

of the switch where the output layer is executed on path p. The execution latency $\Psi_{L,p}$ on path p can be computed by:

$$\Psi_{L,p} = \sum_{v=1}^{l_p} v X_{|\mathcal{K}| \to s_p^v} H\left(1 - \sum_{u=1}^{v} X_{|\mathcal{K}| \to s_p^u}\right) \tag{5.7}$$

where $H(x)$ denotes the unit step function, defined as $H(x) = 1$ if $x \geq 0$ and $H(x) = 0$ if $x < 0$, to ensure that only the first switch of the deployment output layer is considered. However, the step function introduces non-linearity into the objective function, which can significantly increase the complexity of the problem, especially in large-scale networks. To tackle this issue and simplify the problem, we linearize the problem by introducing an auxiliary binary variable $Z_{p,v} \in \{0, 1\}, \forall p \in \mathcal{P}, \forall v \in \{1, \cdots, l_p\}$, represent the output layer execution indicator, where $Z_{p,v} = 1$ indicates that the v-th switch on path p is the first to execute the output layer $|\mathcal{K}|$. The auxiliary variable $Z_{p,v}$ helps identify the correct position for executing the output layer along each path.

Therefore, Using $Z_{p,v}$, we can rewrite the latency (5.7) as below:

$$\Psi_{L,p} = \sum_{v=1}^{l_p} v \cdot Z_{p,v} \tag{5.8}$$

To ensure correctness, $Z_{p,v}$ is subject to the following constraints:

$$Z_{p,v} \leq X_{|\mathcal{K}| \to s_p^v}, \forall p \in \mathcal{P}, \forall v \in \{1, \cdots, l_p\} \tag{5.9}$$

$$\sum_{u=1}^{v-1} X_{|\mathcal{K}| \to s_p^u} + Z_{p,v} \leq 1, \forall p \in \mathcal{P}, \forall v \in \{2, \cdots, l_p\} \tag{5.10}$$

$$\sum_{v=1}^{l_p} Z_{p,v} = 1, \forall p \in \mathcal{P} \tag{5.11}$$

Constraint (5.9) ensures $Z_{p,v}$ can only be 1 if the final layer $|\mathcal{K}|$ is deployed on s_p^v. Constraint (5.10) ensures $Z_{p,v}$ is 1 only if none of the earlier switches on the path $\{s_p^1, \cdots, s_p^{v-1}\}$ has deployed the final layer. Constraint (5.11) ensures that the output layer $|\mathcal{K}|$ is executed at exactly one position along each path.

Integer Linear Programming Problem

The ultimate objective function is a weighted linear combination of the execution latency of all paths and the resource consumption. Hence, we can formulate the integer linear programming problem as follows:

$$\begin{aligned} \min \quad & w_C \sum_{\lambda=1}^{|\mathcal{R}|} \Psi_{C,\lambda} + w_L \sum_{p=1}^{|\mathcal{P}|} \Psi_{L,p} \\ \text{s.t.} \quad & (5.3), (5.4), (5.5), (5.9), (5.10), (5.11) \end{aligned} \tag{5.12}$$

where w_C, $w_L \in \mathbb{R}^+$ are the weights of resource consumption and execution latency, respectively. The weights of latency and resource consumption depend on the specific use case. For example, in an anomaly detection scenario, minimizing detection latency may be prioritized over resource consumption, as quickly identifying anomalies can be critical. Additionally, other objective functions, such as fairness, can also be incorporated.

Solution

The problem described above falls under the category of standard Integer Linear Programming (ILP). Several well-established ILP solvers, such as HiGHS [133] and CPLEX [134], can be employed to obtain an optimal solution. It is acceptable to use these solvers directly if the computational time required by these solvers remains within a practical and tractable range.

5.7 Performance Evaluation

We have evaluated the performance of MUTA on a network with Intel Tofino switches. We selected video streaming Quality of Experience (QoE) prediction [2] and traffic characteristics prediction [3] as the use-case scenarios for validating the performance of our proposed architecture.

5.7.1 Use Cases

Video Streaming QoE: Traffic patterns can be utilized to infer the Quality of Experience (QoE) for video streaming applications. Predicting QoE directly in the data-plane enables faster content delivery and real-time adaptation for video traffic [135]. We use the dataset

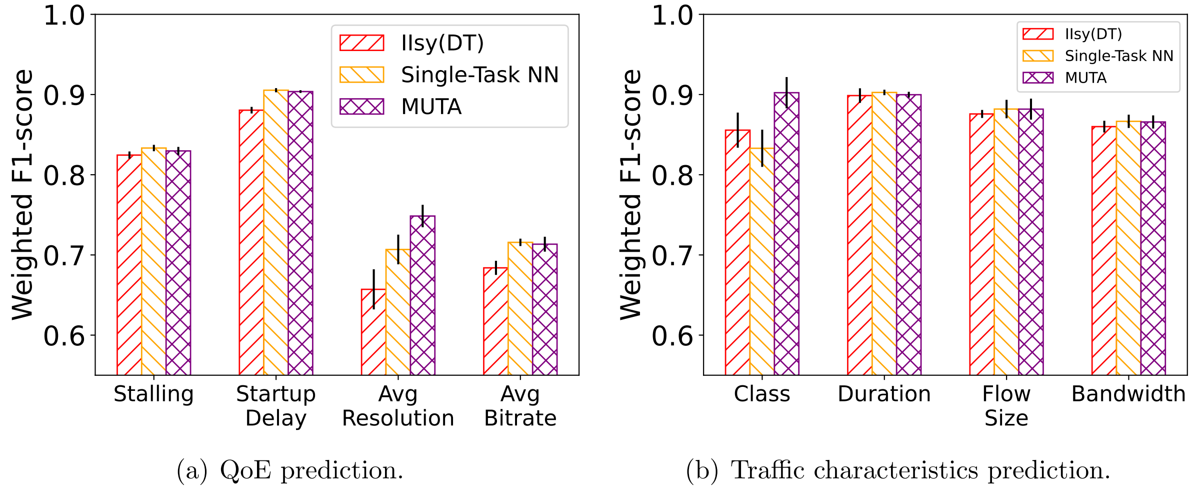
provided by [2] to tackle four tasks, i.e., startup delay, video resolution, video bit-rate prediction, and re-buffering occurrence. It contains the traffic of more than 40000 video sessions labelled with ground truth information obtained at the client side. This dataset applies a simple binary classification into high ($\geq$ 700p) or low average resolution, existing (true) or non-existing (false) stalling, short (< 5 s) or long startup delay, and high ($\geq$ 500 kbps) or low average bit-rate. The dataset consists of 109 flow-level features. However, not all features can be measured on switch ASICs (e.g., skewness and kurtosis). Thus, we only select switch-compatible features for our evaluation. We rank switch-compatible features according to the ANOVA scores [136] and use the top 7 features. Resolution prediction is considered as the *hard-to-label* task in this use case.

Network Traffic Characteristics: Accurate prediction of traffic characteristics in the data-plane is crucial for efficient routing and load balancing. We use QUIC dataset [137] captured at University of California at Davis. It contains QUIC traffic of 5 Google services: Google Docs (1251 flows), Google Drive (1664 flows), Google Music (622 flows), YouTube (1107 flows), Google Search (1945 flows). We tackle four prediction tasks, i.e., bandwidth, duration, flow size, and traffic class prediction tasks. We perform the four tasks by only observing the first few packets, not the entire flow. We formulate the bandwidth and duration prediction problem as a multi-class classification task by dividing the bandwidth and duration values into five classes based on [3]. For flow size prediction, we classify the flows that belong to the top 20% as elephant flows, while the other flows are mice flows. The dataset contains time-series features such as packet length, relative time, and direction. We extract per-flow statistics (max, min, mean) over windows of the first 8, 16, 32, and 64 packets. Features from all window sizes are retained, and inference is triggered after the 64th packet using the complete feature set. Traffic class prediction is considered the *hard-to-label* task in this use case.

5.7.2 Multi-Task Model Performance

Setting and Training

The model employed for QoE prediction includes two hidden layers, each containing 8 nodes. The model used for traffic characteristics prediction has a slightly larger architecture, consisting of two hidden layers with 14 nodes each, to handle the complexity of the multi-class classification task. Both models use ReLU activation for hidden layers. The output layer is task-specific: softmax for multi-class classification in traffic characteristics prediction and sigmoid for binary classification in QoE prediction. During training, we multiply the input of task-specific layer to a mask vector to prevent back-propagation from



(a) QoE prediction.

(b) Traffic characteristics prediction.

Figure 5.4: Performance comparison between IIsy (DT) [1], single-task neural network (NN), and MUTA, using only 100 samples for label-limited tasks (resolution prediction and traffic class prediction) during training.

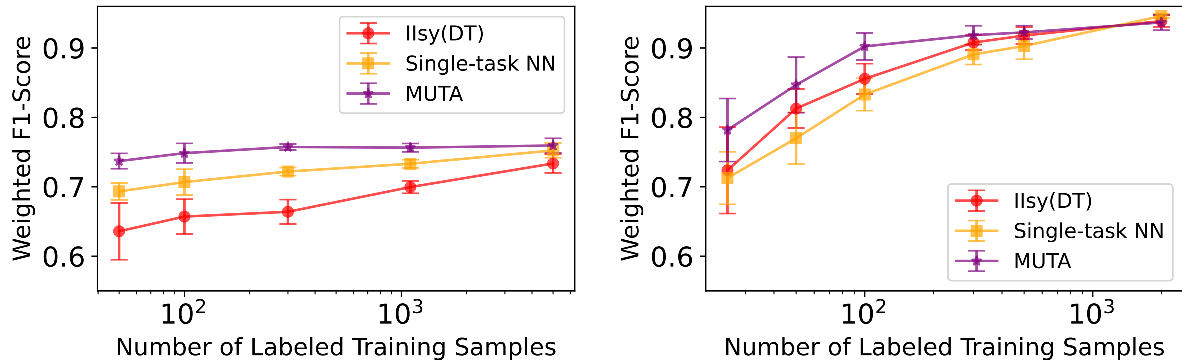
this task for data samples that do not have a label. The depth of the decision tree model is set to 6 for all tasks.

The model training, validation, and quantization operations are performed by the control-plane using TensorFlow Lite [1]. For each use-case, the dataset is split into a training (80%), and a test (20%) sets. To assess model performance, the weighted F1-score is employed, as it offers a more comprehensive evaluation than basic classification accuracy. Particularly in scenarios involving class imbalance or unequal misclassification costs, the F1-score captures the trade-off between precision and recall more effectively. This preference for performance metrics aligns with previous research in this field [1,58,73]. All results are reported on the test set, and the performance is checked using 5-fold cross-validation.

Results

As illustrated in Figure 5.4, MUTA outperforms decision trees (DTs) and single-task NNs for *hard-to-label* tasks in both use-cases, where only 100 labeled samples are available for training. For instance, in the resolution prediction task, MUTA improves the F1-score by 4.17% compared to single-task NNs and by 9.14% compared to DTs. The large amount of data available for the other three tasks improves the training process by allowing the model parameters to be trained with such abundant data. There is no significant performance difference between single-task models and MUTA for the other three tasks

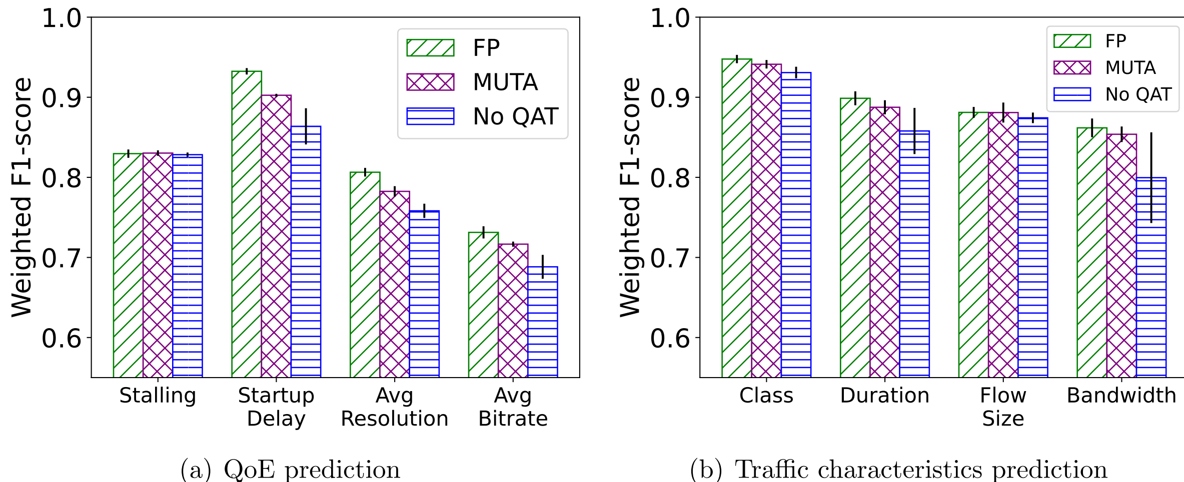
[1] https://www.tensorflow.org/lite



(a) Resolution prediction task in QoE prediction.

(b) Traffic class prediction task in traffic characteristics prediction.

Figure 5.5: Performance comparison for label-limited tasks across different numbers of labeled training samples.



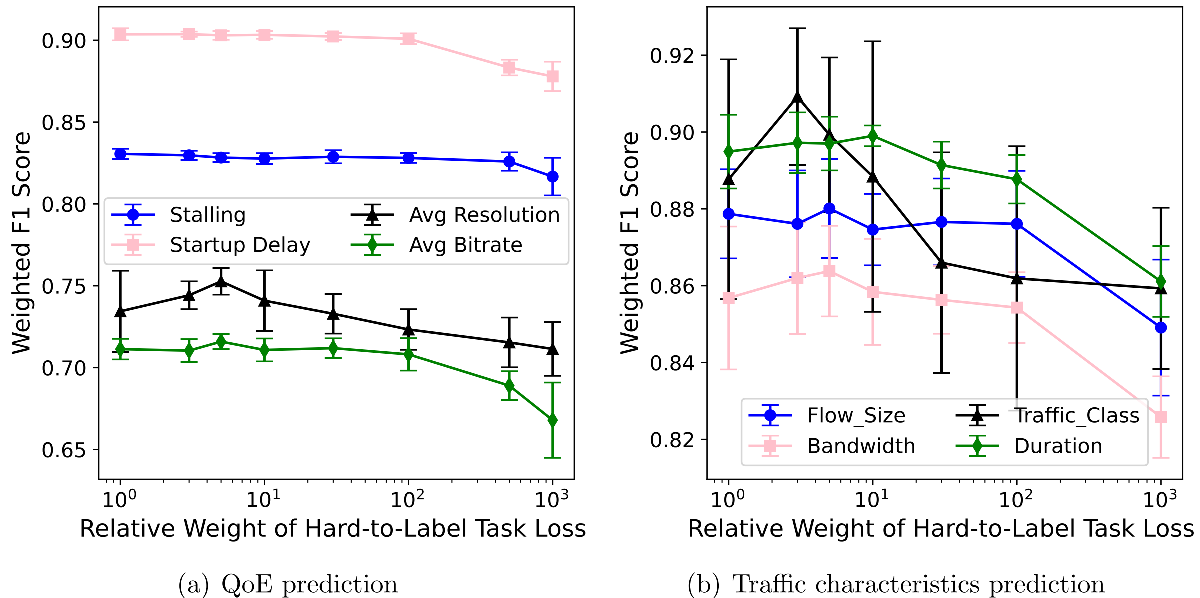
(a) QoE prediction

(b) Traffic characteristics prediction

Figure 5.6: Performance comparison of the floating-point model (FP), quantized model without QAT (No QAT), and MUTA.

because there are abundant training data for these tasks. Single-task models tend to perform poorly when training data is limited, as insufficient supervision increases the risk of underfitting and reduces the model's ability to generalize to unseen instances. This result demonstrates that MUTA can improve the performance of *hard-to-label* tasks without affecting the performance of other tasks.

Figure 5.5 illustrates the performance of three schemes across different numbers of labeled training samples for *hard-to-label* tasks. As shown, MUTA consistently outperforms both DT and single-task NN schemes when the number of available labeled samples is limited. For the resolution prediction task, MUTA with only 100 labeled samples achieves



(a) QoE prediction

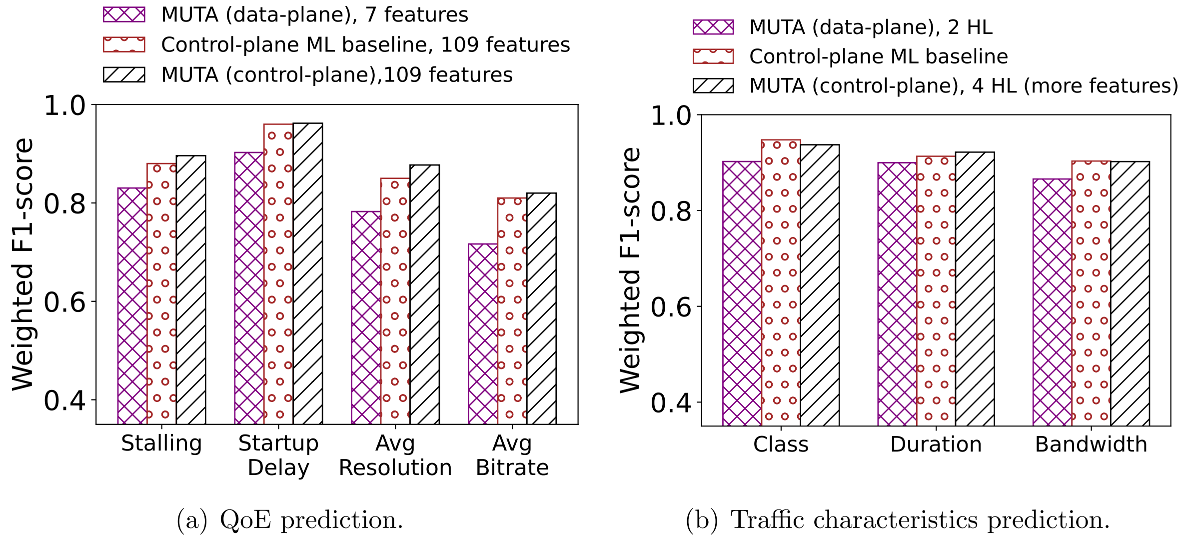
(b) Traffic characteristics prediction

Figure 5.7: Impact of *hard-to-label* task loss weight on model performance.

almost the same performance as single-task models with more than 5000 labeled samples. This is attributed to MUTA's ability to reduce the need on labeled data for *hard-to-label* tasks. By learning shared representations, MUTA effectively transfers knowledge across tasks, thereby improving the performance of tasks with limited labels. As the number of labeled samples increases, the performance gap between the methods decreases. Theoretical conditions under which multi-task models outperform single-task models are discussed in [138], [139] and [140].

Figure 5.6 presents the effect of quantization on accuracy loss for MUTA compared to quantized models without Quantization-Aware Training (QAT), using floating-point models as the baseline. All three schemes have the identical structure. For both use cases, the quantized model without QAT suffers from significant performance loss due to the perturbation of trained parameters during quantization, resulting in severe accuracy degradation. Using QAT, MUTA demonstrates a much smaller performance degradation, highlighting QAT's effectiveness in mitigating accuracy loss during the quantization process.

Figure 5.7 presents the performance of the four tasks under varying loss function weights assigned to the *hard-to-label* task. Intuitively, when the training samples for the *hard-to-label* task are fewer compared to other tasks, the shared parameters of the MTL model are predominantly influenced by tasks with abundant data during training. Increasing the weight of the *hard-to-label* task's loss function can help increase its influence on the training process. As shown in Figure 5.7, increasing this weight initially enhances the performance of the *hard-to-label* task until a maximum is reached. Further increasing



(a) QoE prediction.

(b) Traffic characteristics prediction.

Figure 5.8: Performance comparison between MUTA and control-plane ML schemes. (a) The control-plane ML baseline adopts the QoE prediction method presented in [2]. (b) The control-plane ML baseline follows the MTL scheme proposed in [3]. HL: Hidden Layer.

the relative weight causes performance degradation of all tasks. This degradation can be attributed to the model overfitting to the limited training data available for the *hard-to-label* task, causing the shared parameters to become skewed toward patterns in this task. As the shared parameters are used for all tasks, this bias negatively impacts their performance as well. Additionally, excessively large gradient updates for the *hard-to-label* task introduce instability, making it difficult for the model to converge effectively during training. Therefore, selecting an appropriate loss weight for the *hard-to-label* task is crucial to achieve optimal performance across all tasks.

Compare with control-plane ML: Figure 5.8 presents a comparison between MUTA and traditional control-plane ML schemes. For QoE prediction, the control-plane ML baseline is based on the methodology introduced by Seufert and Orsolic [2], who benchmark various classical ML algorithms and find that a random forest model trained on the complete set of 109 input features offers the best performance. As shown in Figure 5.8 (a), while the data-plane model exhibits slightly lower accuracy in this scenario, the gap is largely attributable to its use of only seven input features due to hardware constraints. When MUTA is configured to leverage the complete set of 109 input features for training a large MTL model in the control-plane, it achieves an average performance improvement of 1.5% compared to the baseline in [2]. For traffic characteristics prediction, we adopt a representative control-plane MTL scheme proposed by Rezaei and Liu [3], which employs a deep 1D-CNN architecture with over 10 layers and utilizes fine-grained time-series features. Their model addresses three specific tasks: bandwidth, duration, and traffic class

prediction. Accordingly, we restrict our comparison to these three tasks. As illustrated in Figure 5.8 (b), despite using coarse-grained features (statistical features) and significantly smaller models (only two hidden layers), the data-plane model still has a competitive performance, especially given the substantial computational and storage resource disparity between the control-plane and data-plane. When MUTA is configured to train a larger model with more features in the control-plane, it achieves nearly the same accuracy as the scheme proposed in [3], while using fewer layers.

To further enhance MUTA's performance, a hybrid approach similar to that proposed in [1] can be considered. For example, initial traffic classification can be performed on switches using the MTL model at line rate, and only traffic samples with low classification confidence are forwarded to a server for inference using a more sophisticated model. This hybrid strategy reduces latency and server load while improving overall classification performance.

5.7.3 Hardware Resource Consumption

Setting and Metrics

We implement the model using $P4_{16}$ targeting Tofino Native Architecture (TNA) [31] used in Intel Tofino switch ASIC. All P4 code was compiled using version 9.13.2 of Intel Barefoot SDE. For the resource consumption, we mainly focus on the following three aspects: 1) Program resources, i.e., the number of stages, and table entries; 2) Memory resources, i.e., the percentage of used SRAM and TCAM; 3) The metadata used to execute action functions. The results reported are based on the QoE prediction use case. MUTA is compared to two advanced tree-based solutions, i.e., the feature-encoding solution (e.g., IIsy [1]) (other schemes such as Flowrest [54] and NetBeacon [14] are all derived from or closely related to IIsy) and the direct mapping solution (e.g., SwitchTree [53] and pforest [52]). These tree models are generated using Planter [77]. However, every tree model generated using the direct mapping solution failed to fit due to extremely high pipeline-stage consumption. Consequently, we report results only for tree models generated using the feature-encoding solution (i.e., IIsy [1]).

Results

Table 5.2 presents the resource consumption of IIsy for each individual task as well as the cumulative consumption for all four tasks combined. Similarly, Table 5.3 details the resource utilization of MUTA across each layer of the neural network, along with the

Table 5.2: Resource consumption for IIsy (DT): T1 - stalling prediction, T2 - startup delay prediction, T3 - resolution prediction, T4 - bitrate prediction.

	T1	T2	T3	T4	Total
SRAM(%)	23.23	56.67	89.48	28.44	197.82
TCAM(%)	2.431	2.431	2.431	2.431	9.724
Stages	4	8	12	5	29
Table Entries	421874	940243	1490240	526208	3378565
Metadata (bytes)	19	35	51	23	128

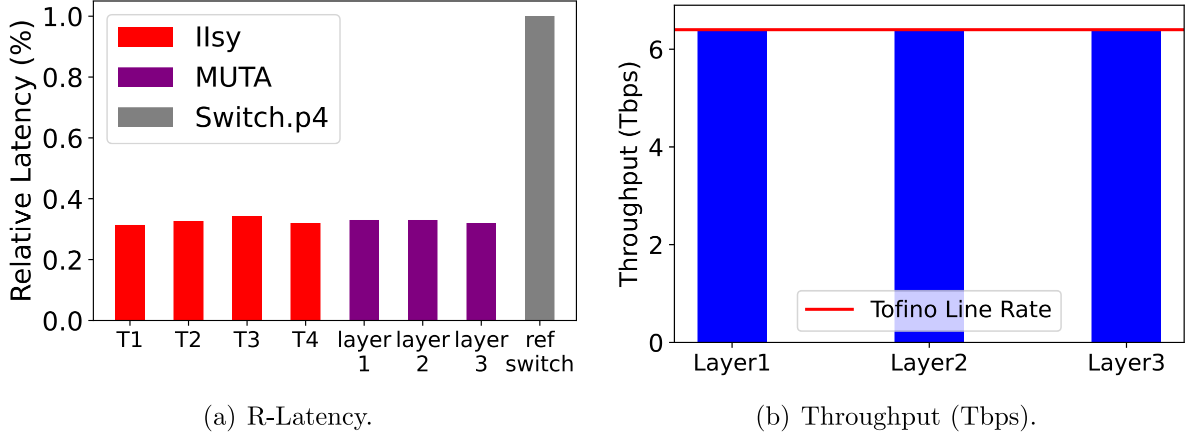
Table 5.3: Resource consumption for MUTA.

	Layer1	Layer2	Layer3	Total
SRAM(%)	2.178	10.00	6.667	18.845
TCAM(%)	6.250	0	0	6.250
Stages	6	6	5	17
Table Entries	1560	2048	2048	5656
Metadata (bytes)	260	292	128	680

total consumption for the entire model. It is important to note that TCAM is only utilized in the first layer of the MTL model, due to the range-based match type used in the match-action table at this layer. In contrast, subsequent layers employ exact match tables exclusively. Compared to IIsy, MUTA consumes significantly lower memory resources, especially for SRAM, reducing usage from 197.82% to 18.845%. Moreover, MUTA reduces stage consumption, requiring only 17 stages to execute all tasks, fitting within a Tofino2 switch (20 stages available) [141] or use the proposed distributed execution across multiple Tofino switches (12 stages available). In contrast, IIsy needs 29 stages to complete four tasks. However, MUTA incurs 5.3 times the metadata usage due to the parallel execution of multiple match-action tables. These results indicate that, at the cost of increased consumption of metadata, MUTA demonstrates improved memory and stage efficiency relative to IIsy.

Trade-off between stage and metadata

There is a trade-off between the number of used stages and metadata at varying levels of parallelization. Using more metadata allows for more table lookups per stage, which leads to higher parallelization, thereby saving more stages. Conversely, lower levels of parallelization, or the absence thereof, result in a greater number of stages. The decision regarding this trade-off depends on the specific scenario.



(a) R-Latency.

(b) Throughput (Tbps).

Figure 5.9: (a) Pipeline Relative Latency (R-Latency) on Tofino switches for tree models (IIsy), different layers in MUTA, and standalone switch.p4. (b) Throughput for different layers in MUTA on Tofino switches.

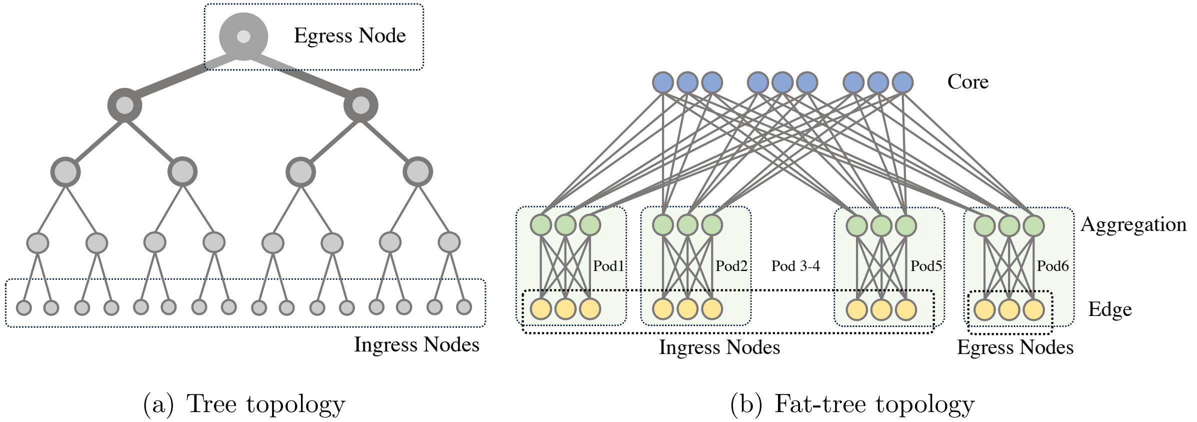
5.7.4 Latency and Throughput

Setting

The latency of Tofino is under non-disclosure agreement (NDA), therefore we report our measurements of pipeline latency of each layer relative to *switch.p4*, an L2/L3 reference switch program for Tofino, including 10 network functions such as load balancing, tunneling, firewall, and statistics. MUTA's relative pipeline latency is computed based on data reported by SDE. In the throughput test, the Tofino switch with bf-sde-9.5.0 runs each layer of the model with snake configuration. Packets are sent to the switch by a server using DPDK 20.11 via a 100G NIC with both (i) collected network traffic traces and (ii) synthetic traffic.

Results

As shown in Figure 5.9 (a), all of MUTA layers have a lower latency than the reference *switch.p4*. The latency for all layers is less than 33% of *switch.p4*. This illustrates that even under resource constraints, MUTA still can achieve comparable latency (at the sub-microsecond level) to simple packet switching. As shown in Figure 5.9 (b), all layers are able to achieve a full line-rate of 6.4Tbps.



(a) Tree topology

(b) Fat-tree topology

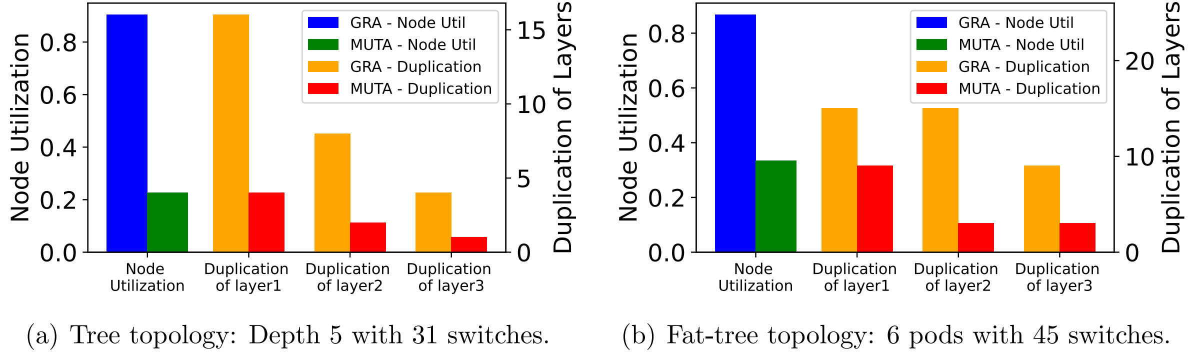
Figure 5.10: Network topology used for evaluation.

5.7.5 Network-Wide Deployment Performance

Simulation Setting and Metrics

We use the IBM ILOG CPLEX optimization solver [134] to solve the optimization problem in Section 5.6. The computations were conducted on a PC with Intel Core i7-9750H processor @ 2.60 GHz cpu and 16 GB of RAM. We evaluate the deployment orchestrator using two network topologies, as illustrated in Figure 5.10. The first topology, shown in Figure 5.10 (a), is a tree structure with a depth of 5, comprising 31 switches and 640 servers. In this configuration, the root switch is considered as the egress node, while the leaf switches function as ingress nodes, reflecting a hierarchical and centralized traffic flow. The second topology, depicted in Figure 5.10 (b), is a fat-tree architecture with 6 pods, consisting of 45 switches and 600 servers, and is commonly used in data center networks [142]. In this setup, the edge switches within the rightmost pod are configured as egress nodes, while all other edge switches act as ingress nodes. We use two metrics to evaluate the efficiency of our orchestrator in such topology with multiple paths. 1. Node utilization: The percentage of used nodes compared to the total available nodes. 2. Layer Duplication: the number of duplication for different layers across all paths.

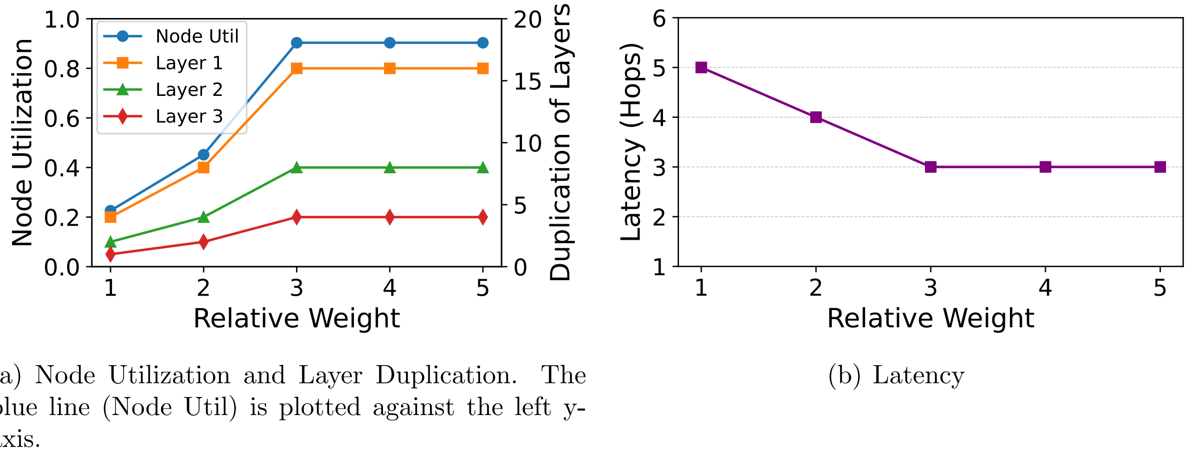
We compare our deployment orchestrator with a baseline method called greedy resource availability (GRA). In this baseline, each layer is deployed on the first switch along the path that has sufficient available resources to accommodate it. The process continues sequentially for the subsequent layers until all layers are deployed.



(a) Tree topology: Depth 5 with 31 switches.

(b) Fat-tree topology: 6 pods with 45 switches.

Figure 5.11: Network-wide deployment performance between MUTA and the greedy resource availability (GRA) baseline.



(a) Node Utilization and Layer Duplication. The blue line (Node Util) is plotted against the left y-axis.

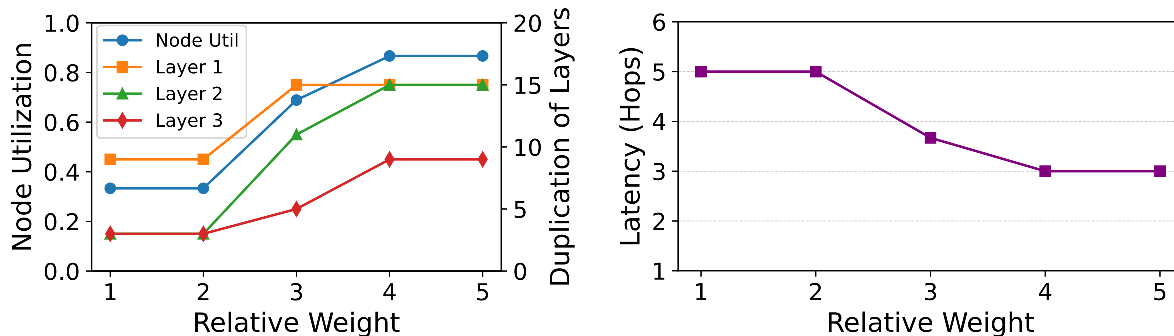
(b) Latency

Figure 5.12: Relative objective weight of w (defined in (5.12)) influence on MUTA's deployment strategy. Tree topology: Depth 5 with 31 switches.

Results

Figure 5.11 shows the comparison between GRA and MUTA in a tree topology with a depth of 5 and a fat-tree topology with 6 pods. In both topologies, compared to the GRA deployment strategy, MUTA has lower node utilization. This means MUTA has higher resource efficiency because it can cover the MTL-based service with fewer nodes. At the same time, MUTA significantly reduces duplication across layers. In the tree topology, MUTA reduces duplication in Layer 1, Layer 2, and Layer 3 by 75% compared to GRA. In the fat-tree topology, MUTA achieves a 40% reduction in Layer 1 (from 15 duplications to 9), an 80% reduction in Layer 2 (from 15 duplications to 3), and a 66.7% reduction in Layer 3 (from 9 duplications to 3). These reductions saves the memory and stage resources.

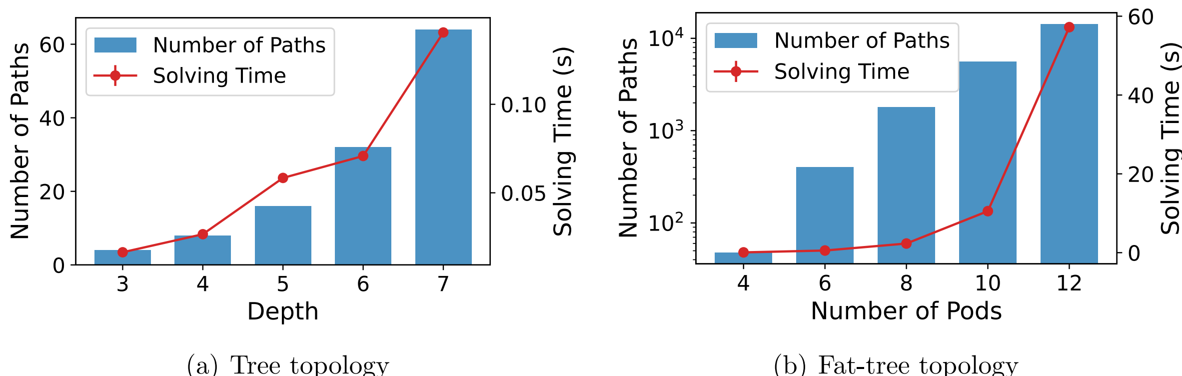
Figures 5.12 and 5.13 illustrate the tradeoff between the orchestrator's impact on la-



(a) Node Utilization and Layer Duplication. The blue line (Node Util) is plotted against the left y-axis.

(b) Latency

Figure 5.13: Relative objective weight of w (defined in (5.12)) influence on MUTA's deployment strategy. Fat-tree topology: 6 pods with 45 switches.



(a) Tree topology

(b) Fat-tree topology

Figure 5.14: Impact of topology scale on solver execution time.

tency and resource efficiency as the relative weights of latency and other objectives are varied. In both topologies, increasing the relative weight of latency results in a reduction in the number of hops. To ensure functionality, the provided strategy requires more nodes and additional duplicated layers. These findings highlight MUTA's flexibility in adapting deployment strategies based on the defined objective function and its associated weights.

Figure. 5.14 illustrates the solving time as the scale of the topology increases. As the topology expands, the number of available paths grows, leading to an increase in computational time. For tree topology, the optimal deployment decision can be determined in under half a second. For fat-tree topology, the number of path grows more significantly with the number of pods, exceeding 10,000 paths when the number of pods reaches 12. However, the solving time is still under one minute. The computation time for obtaining optimal deployment decisions using IBM ILOG CPLEX solver [134], which handles ILP

formulation introduced in Section 5.6, remains efficient and well within acceptable limits. Therefore, we do not consider any heuristic method for this optimization problem.

5.8 Discussion

In this section, we discuss some key considerations for using MUTA.

Model update: After an MTL model is deployed, it is essential to periodically update it, adapting to changes in traffic patterns. There are two possible update scenarios: updating model weights and model structure modification (e.g., changing the number of nodes). Updating model weights can be done at runtime, as it only requires entry updates in match-action tables, and these can be done atomically without affecting the forwarding pipeline. Modifying the model structure requires stopping traffic during the update. The implementation of model updates after retraining is envisioned as a future enhancement, building on our previous continuous learning work P4Pir [123] and drift detection work SPIDD [21].

Deployment updates: Changes in network topology (e.g., adding or removing switches) or in model structure (e.g., increasing the number of layers), require rerunning the orchestrator to find the optimal deployment decision and updating the affected switches accordingly. Just like routing tables need to be updated when the network changes, the deployment update can be carried out as part of that process.

Feature management: MUTA is agnostic to whether features are derived at the packet-level or at the flow-level, whether through early flow classification using the first few packets or full-flow classification by observing the entire flow. Packet-level features can be extracted directly from packet headers with minimal overhead, whereas flow-level features require maintaining state across packets using registers in the data-plane. This design introduces hash collisions when multiple flows map to the same register entry. Prior work, such as Flowrest [54], addresses this challenge by employing timeout-based eviction policies to manage per-flow state. These techniques complement our approach and can be seamlessly integrated into MUTA.

Scalability: The scalability of MUTA is improved by distributing MTL model layers across multiple switches. This strategy enables effective management of the computational load and facilitates model expansion as necessary. The maximum number of model layers depends on the number of switches available on a given path. In terms of layer size, using Tofino, each switch can handle a layer of 16x16. Tofino 2 can manage larger layers, as it supports more stages and memory resources than the Tofino chip we utilize.

In-band transmission overhead: MUTA transmits intermediate layer outputs in-band using packet headers to enable distributed inference across switches. In our implementation, the size of these intermediate results is minimal due to the use of 8-bit quantization and compact layer dimensions (e.g., 8–16 nodes), requiring only a few tens of bytes per packet. This overhead remains well below the Ethernet MTU of 1500 bytes. However, scaling to larger models or supporting a higher number of tasks could increase the header size beyond the MTU. In such cases, enabling jumbo frames [143] or implementing fragmentation and reassembly mechanisms in the data-plane would be necessary. These strategies represent an important consideration for future deployments of more complex models.

Task Grouping: Training all tasks together in a single model may not always be optimal, as the model might fail to learn a shared representation that can generalize to all objectives. To address this, one can analyze inter-task affinity [144] to determine which tasks should be grouped and trained together. Inter-task affinity captures how much a task's gradient update helps or hurts another task's loss, allowing the identification of task groupings that are more likely to reduce each other's losses during training. This task grouping can be formulated as an optimization problem, where the objective is to maximize model performance (e.g., the sum of the accuracy of each task.) while considering data-plane resource limits as a constraint.

Resource optimization: For neural networks, there is a trade-off between performance and complexity, characterized by parameters such as depth (number of layers) and width (number of nodes per layer). Increased complexity (i.e., a deeper or wider network) typically results in higher resource consumption. Consequently, it is sometimes pragmatic to trade off a bit of accuracy to reduce complexity. For instance, saving half the resources while only losing 1% of accuracy. Furthermore, techniques such as pruning [116] can be applied to reduce the resources required for vector-matrix multiplication operations by eliminating parameters that do not significantly impact inference accuracy.

Use cases: While MUTA is primarily designed for network management tasks, it is also applicable to other MTL-based applications, such as in-network financial market prediction for high-frequency trading [145] (e.g., forecasting future stock price movements and volatility across different periods).

Generalization: MUTA is generic in the sense that all its core designs are adaptable. The mapping methodology uses match-action tables, a common data-plane primitive, making MUTA potentially deployable on other types of programmable data-planes. While MUTA has been demonstrated on Tofino switches, alternative platforms such as Xsight Labs X2 [146] and Cisco Silicon One [147] also support similar capabilities and could serve

as viable deployment targets. The deployment orchestrator is scalable and flexible, and can support other in-network computing tasks (e.g., other resource-heavy in-network ML tasks [77]). As long as a task can be divided into smaller sub-tasks, the dependencies between sub-tasks can be established, and the resources required for sub-tasks can be estimated.

Potential enhancement: While MUTA has been evaluated on use cases involving four tasks, future work will explore using one MTL model for as many network management tasks as possible. This involves a deep analysis of the relationships and potential synergies among various tasks to determine which can be effectively learned together within a shared model architecture. This requires the collection of a comprehensive, high-quality dataset that captures the diverse nature of these tasks and their interdependencies. Additionally, we aim to achieve more fine-grained distributed execution by splitting layers into smaller parts to maximize resource utilization in the programmable data-plane.

5.9 Conclusion

In this chapter, we introduced a novel in-network solution for multi-task learning (MTL). Given multiple network management tasks, MUTA demonstrates enhanced performance for tasks with limited labeled data. The architecture effectively maps MTL model layers into match-action tables and deploys these layers in a distributed manner in programmable switches, while ensuring the MTL-based service covers the entire network. Experimental results indicate that MUTA runs at line-rate, efficiently utilizes switch resources, and optimizes layer-to-switch placement in multi-path networks.

Chapter 6

Towards Unsupervised Drift Detection in Programmable Data-Planes

6.1 Introduction

As programmable data-planes continue to integrate increasingly sophisticated machine learning (ML) capabilities, most intelligent data-plane solutions [14, 48, 73] are trained only once using pre-collected traffic traces and are expected to perform well without further retraining. While they perform well in stable environments, their performance may degrade when network conditions change over time. This degradation occurs because the underlying patterns of traffic evolve due to factors such as traffic fluctuations or the emergence of new traffic types. This shift in network behavior, known as concept drift [148], is a common challenge in real-time ML applications (e.g., security [149]) and often requires adaptive strategies to maintain accuracy.

To sustain in-network ML performance, one common approach is to periodically retrain the model using newly observed traffic data [16,58,123]. However, this method is inefficient because network behavior changes unpredictably. Fixed retraining schedules may either be too frequent, wasting resources (CPU/GPU cycles and network bandwidth), or too infrequent, causing performance degradation. A more effective solution is to retrain the model only when concept drift is detected. This strategy avoids unnecessary retraining, reducing overhead while ensuring the model is updated only when meaningful changes in traffic behavior occur. Existing methods typically address this challenge by monitoring model accuracy and initiating retraining when a decline is observed [150, 151]. However,

these supervised approaches rely on the immediate availability of ground truth labels, which are often difficult to obtain or only available with a delay.

In this work, we propose SPIDD, an unsupervised concept drift detection framework designed for in-network deployment. SPIDD does not require labeled data and operates entirely within the data-plane. It monitors flow feature distributions over two non-overlapping time windows, representing historical and recent traffic, and quantifies their dissimilarity using a similarity metric. These distributions are recorded in P4 registers, and a drift is flagged when the dissimilarity between windows exceeds a predefined threshold. Upon detection, model retraining is triggered to restore classification performance. In summary, the key contributions of this chapter are as follows:

- We present a framework, SPIDD, for **S**ustaining **P**erformance of **I**n-network ML through adaptive retraining triggered by unsupervised **D**rift **D**etection.
- We design and implement a concept drift detection mechanism that compares historical and recent traffic distributions directly within the data-plane using P4.
- Simulation results show that SPIDD accurately detects concept drift in dynamic network environments.

6.2 Related Work

Concept drift detection techniques are typically divided into two main categories: error-rate- and data-distribution-based approaches [152]. Error-rate-based drift detection monitors the online performance of a deployed ML model. When a statistically significant increase in error rate is observed, a retraining process is triggered. In contrast, data-distribution-based methods assess drift by comparing the statistical distribution of input features over time. These approaches use distance metrics to evaluate the dissimilarity between historical and recent data. If the difference exceeds a threshold, a drift is detected and the model is updated accordingly.

In the context of in-network ML, most existing efforts have followed the error-rate-based approach. Xavier et al. [150] implements supervised in-network drift detection directly in the data-plane by tracking model accuracy. However, this method fundamentally depends on obtaining immediate ground truth labels for incoming packets within the data-plane—an unrealistic requirement in real-world deployments where ground truth labels are unavailable in the data-plane. In contrast, CARAVAN [151] tracks model degradation using an accuracy proxy in the control-plane, leveraging a large language model for packet labeling.

While more feasible, CARAVAN requires continuous packet sampling from the data-plane, introducing additional overhead.

Our approach differs from these prior works by adopting a data-distribution-based strategy that operates without labels and entirely within the data-plane.

6.3 In-Network Unsupervised Drift Detection

This section presents SPIDD, a novel unsupervised drift detection solution that operates entirely in the data-plane and enables the timely triggering and notification of model updates in response to detected drift.

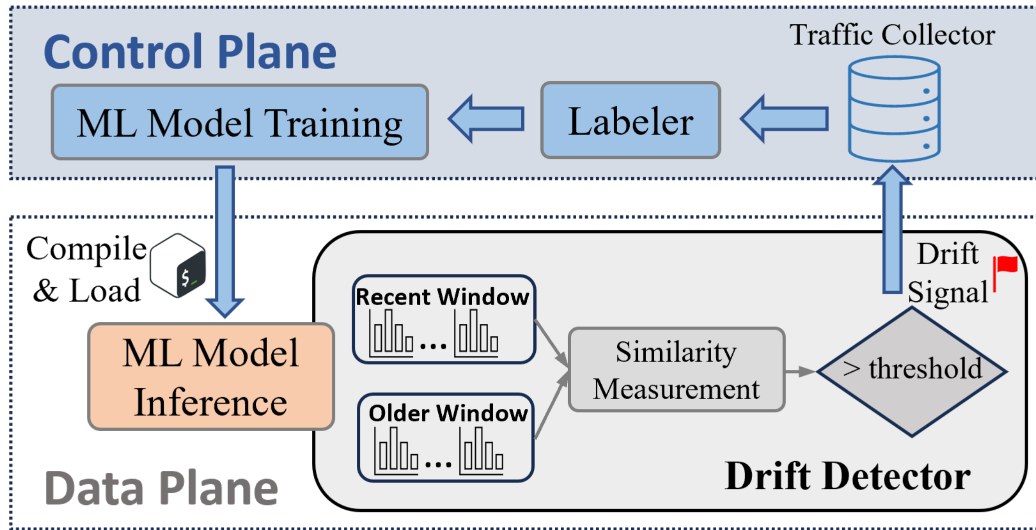


Figure 6.1: SPIDD architecture.

6.3.1 Method Overview

Figure 6.1 illustrates the high-level architecture of SPIDD. The system follows the software-defined networking (SDN) paradigm and is composed of two main components: the data-plane and the control-plane. The data-plane is responsible for performing both ML inference and unsupervised drift detection in real time. The control-plane is notified when a drift is detected and is responsible for managing model retraining.

At the core of SPIDD's drift detection mechanism are two sliding windows: one capturing historical traffic and another collecting recent traffic. These windows are used to track the evolving distribution of traffic features over time. As new packets arrive, their

features are recorded into the current window. Once the recent window reaches its predefined capacity, the system computes the similarity between the feature distributions in the historical and recent windows.

If this distance exceeds a predefined threshold, the system reports the drift to the control-plane and triggers a model retraining. The recent window is then designated as the old window, while a new empty recent window begins recording feature distributions of newly incoming traffic.

To quantify the similarity, the data-plane calculates the total variation distance (TVD) between the two distributions. If this distance exceeds a predefined threshold, a drift is flagged. The system then sends a drift signal to the control-plane, which initiates the retraining pipeline. At this point, the recent window is designated to become the new historical window, and a fresh recent window begins collecting traffic statistics from incoming flows.

Upon receiving a drift notification, the control-plane begins collecting samples from the recent traffic window and stores them in a new training dataset. A labeling module (e.g., using heuristics rule-set) assigns ground truth labels to the collected samples. Once labeled, the data is used to retrain the ML model. The newly trained model is then offloaded to the data-plane to restore and maintain high inference accuracy. The detailed steps of the in-network drift detection process are outlined in Algorithm 1. While we mention how the control-plane might respond upon receiving a drift notification, such as collecting samples, labeling, and retraining, these steps are included only to contextualize how SPIDD could fit into a broader system. The design and evaluation of these downstream components are beyond the scope of this work.

6.3.2 Feature Distribution

Traffic features are measurable characteristics of network or user activity, such as packet size, inter-arrival time, and flag count. To track the distribution of traffic features in the data-plane, SPIDD uses a histogram-based approach. Each monitored feature is discretized into a fixed number of bins, with each bin representing a specific range of feature values. The level of discretization, referred to as granularity (i.e., the number of bins per feature), is a tunable parameter that balances detection sensitivity with resource efficiency. Higher granularity (more bins) facilitates the detection of fine-grained distributional changes but incurs increased memory and computational overhead. Conversely, lower granularity (fewer bins) reduces resource consumption at the expense of reduced sensitivity to minor shifts. Then, the bin edges are computed by dividing the observed value range of each feature

into equal-width intervals.

The system maintains a count of how many times a feature value falls into each bin over a given time window, maintaining these counts in P4 register arrays. As packets are processed, their feature values are mapped to the corresponding bins, and the associated counters are incremented. This approach allows the system to build an approximate representation of the empirical distribution of traffic features using only basic arithmetic operations, making it suitable for in-network execution. By comparing the histograms from the historical and recent windows, SPIDD can detect shifts in traffic patterns that signal concept drift.

6.3.3 Similarity Measurement

Several metrics can be used to compare distributions and detect drift, including Kullback–Leibler (KL) divergence and Hellinger distance [153]. However, these metrics rely on computationally expensive operations such as logarithms, square roots and divisions, which are not natively supported by P4 and therefore unsuitable for in-network use.

SPIDD employs Total Variation Distance (TVD) [154] as the similarity metric. TVD is well-suited for data-plane implementation because it only requires basic arithmetic and absolute value operations—both of which are supported in P4. Formally, given two distribution P and Q over the sample space Ω, their total variation distance, also known as the statistical difference is defined as:

$$TVD(P,Q) = \frac{1}{2}\sum_{x\in\Omega}|P(x) - Q(x)| \tag{6.1}$$

In practice, we omit the constant factor $\frac{1}{2}$ as it does not affect the threshold-based drift detection. In SPIDD, each traffic feature has its own histogram, and the TVD is computed for each feature individually. These distances are then summed across all features to produce a single total TVD value (line 11 in Algorithm 1). This total TVD provides a simple yet effective measure of how much the recent traffic distribution deviates from historical patterns.

Although SPIDD computes the TVD for each feature independently and aggregates the results to produce a total drift score, this approach is primarily designed for coarse-grained drift detection. It implicitly assumes that all features contribute equally to the overall drift, which may not reflect the underlying semantics of traffic patterns. For example, a substantial deviation in a single feature and minor deviations across multiple features may yield the same total TVD values, despite potentially indicating different types of

Algorithm 1 SPIDD: Unsupervised Drift Detection

Initialization: Bin edges, TVD threshold θ, window size N

```
1:  Initialize counters for hist_old and hist_recent to 0
2:  count ← 0
3:  for each incoming packet do
4:      for each feature f do
5:          Extract feature value v_f
6:          Determine bin index b_f from v_f using bin edges
7:          Increment hist_recent[f][b_f] by 1
8:      end for
9:      count ← count + 1
10:     if count ≥ N then
11:         TVD ← Σ_f Σ_i |hist_recent[f][i] − hist_old[f][i]|
12:         if TVD > θ then
13:             Report drift to control-plane
14:         end if
15:         hist_old ← hist_recent
16:         Reset hist_recent to 0
17:         count ← 0
18:     end if
19: end for
```

distributional change. Such uniform weighting can obscure the relative informativeness and volatility of individual features, particularly in contexts where certain attributes are more indicative of significant shifts in network behavior. To support more fine-grained and semantically meaningful drift analysis, future work may consider assigning different weights to different features based on their historical volatility.

6.4 P4 Implementation

In this section, we describe how SPIDD is implemented within the P4 data-plane pipeline.

6.4.1 Sliding Window Histogram

P4 registers are stateful memory structures available in the data-plane that allow programmable switches to maintain and update persistent values across packets. SPIDD leverages these registers to maintain histograms for traffic feature distributions. Each monitored feature is associated with a dedicated register array, where each entry (or bin) corresponds to a predefined value range. As packets are processed, relevant features are

extracted, and the appropriate bin index is determined. The corresponding bin counter is then incremented, allowing the feature distribution to be updated over time.

To efficiently determine the bin index for each feature value, SPIDD utilizes a match-action table configured with range-based matching. Each table entry specifies a minimum and maximum value, defining an interval that corresponds to a particular histogram bin. As shown in Listing 1, when a packet arrives, the *feature1_update* table matches the value of *meta.feature1* against a defined range to determine which bin it belongs to. The associated action, *set_feature1_bin*, is then invoked to increment the appropriate counter in the register array.

```
register<bit<32>>(NUMBER_OF_BINS) feature1;
action set_feature1_bin(bit<32> bin_index) {
    bit<32> c1;
    feature1.read(c1, bin_index);
    feature1.write(bin_index, c1+1);
}
table feature1_update {
    key = {
        meta.feature1: range; // feature value
    }
    actions = {
        set_feature1_bin;
    }
    size = NUMBER_OF_BINS;
}
```

Listing 6.1: P4 code fragment that implements feature distribution updates.

6.4.2 Distance Calculation

As described in §6.3.3, TVD requires calculating the absolute difference between corresponding bins in two histograms and summing the results across all bins. Since P4 does not natively support absolute value operations, we emulate absolute value computation using ternary conditional expressions. Specifically, for any two values a and b, the expression $|a - b|$ is computed as:

$$a - b > 0 \; ? \; a - b : b - a \tag{6.2}$$

This operation is performed across all corresponding bins of the recent and historical histograms.

Table 6.1: Injected concept drifts (mixed attacks) in CICIDS2017 dataset over time.

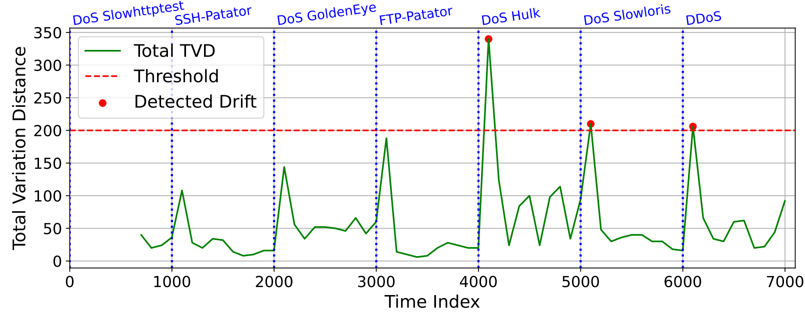
Time Interval	Injected Mixed Attacks	
	inter-category	intra-category
t0-t2000	DoS Slowhttptest+ SSH-Patator	DDoS+ SSH-Patator
t2001-t4000	FTP-Patator+ DoS Hulk	DoS GoldenEye+ DoS Hulk
t4001-t6000	SSH-Patator+ DoS GoldenEye	DoS Slowloris+ DoS Slowhttptest
t6001-t8000	DDoS+ DoS Slowloris	SSH-Patator+ FTP-Patator
t8001-t10000	SSH-Patator+ DoS Hulk	DoS Slowloris+ DoS Hulk
t10001-t12000	FTP-Patator+ DoS GoldenEye	DoS GoldenEye+ DoS Slowhttptest
t12001-t14000	DDoS+ DoS Slowhttptest	DDoS+ DoS Hulk

6.5 Performance Evaluation

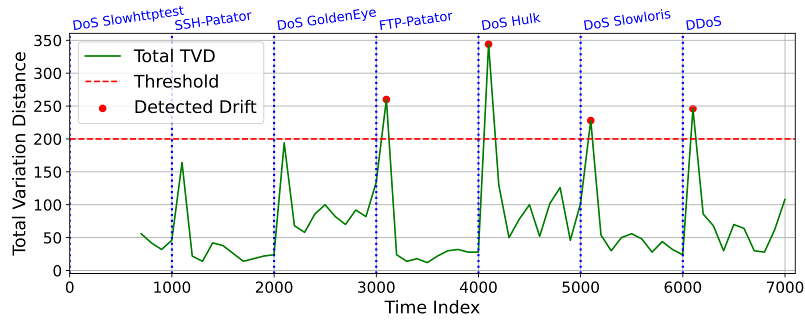
In this section, we present experimental results to evaluate the effectiveness of our unsupervised drift detection framework in a dynamic network environment.

6.5.1 Dataset

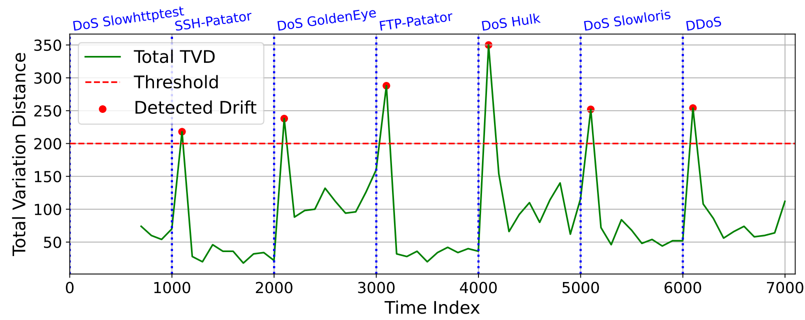
We use network intrusion detection as the primary use case to validate SPIDD. The objective is to classify network flows based on whether they exhibit malicious behavior. The detection system must adapt to emerging attacks that were not present in the original training data. For this purpose, we use the CICIDS2017 dataset [108], evaluating both single-attack and mixed-attack scenarios. The single-attack setting follows the drift configuration provided by [151], which includes seven concept drifts introduced over time through the injection of new attack traffic. To further assess robustness, we design two mixed-attack scenarios. The first scenario introduces inter-category diversity by combining DoS and non-DoS attack types, thereby simulating a wider spectrum of threat categories. The second scenario captures intra-category variability by mixing attacks with similar behavioral patterns, such as multiple DoS variants or brute-force attacks like SSH-Patator and FTP-Patator. An overview of the mixed-attack configuration is provided in Table 6.1. Additionally, we also evaluate the generality of SPIDD using the UNSW-NB15 dataset [155], which includes a different set of attack types.



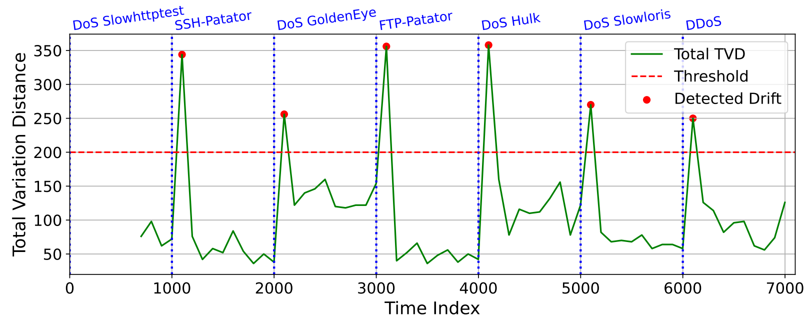
(a) 3 bins



(b) 5 bins

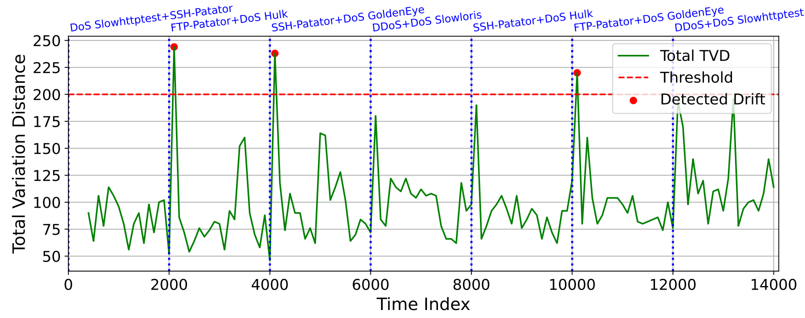


(c) 8 bins

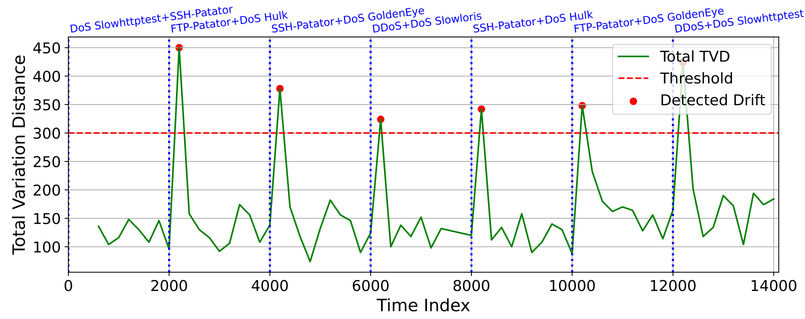


(d) 12 bins

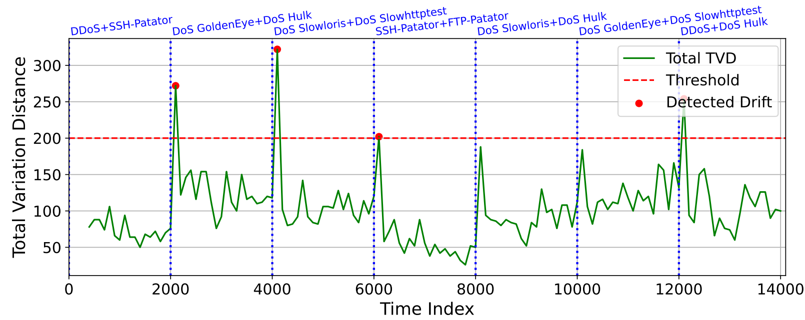
Figure 6.2: Impact of histogram bin count on drift detection performance.



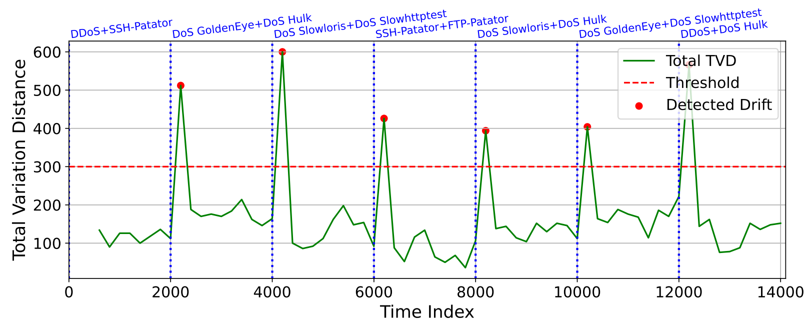
(a) Inter-category Scenario; window size: 100



(b) Inter-category Scenario; window size: 200



(c) Intra-category Scenario; window size: 100



(d) Intra-category Scenario; window size: 200

Figure 6.3: Drift detection performance in mixed-attack scenarios.

6.5.2 Experimental Setting

SPIDD is implemented in BMv2 software switch using P4 with v1model architecture. For CICIDS2017 dataset, we use seven traffic features: flow IAT min, flow IAT max, and flow IAT mean (minimum, maximum, and average inter-arrival time in microseconds); packet length min, packet length max, and packet length mean (minimum, maximum, and average packet length); and flow duration (in microseconds).

The UNSW-NB15 dataset [155] includes a different set of attack types, such as Reconnaissance, Exploits, Generic, Fuzzers, and Backdoor. We use seven features to track the traffic distribution: smean (mean of the packet size transmitted by the src), dmean (mean of the packet size transmitted by the dst), sttl (source to destination time to live value), dttl (destination to source time to live value), dur (record total duration), ct_src_ltm (no. of connections of the same source address in 100 connections according to the last time), ct_dst_ltm (no. of connections of the same destination address in 100 connections according to the last time).

6.5.3 Results

Figure 6.2 illustrates the impact of the number of histogram bins on SPIDD's drift detection performance in the single-attack setting. Each subfigure shows the TVD over time, using a fixed threshold of 200 and a window size of 100, with different bin counts applied to each histogram. Red dots indicate the time points at which concept drift was detected. When using only 3 bins, the detector fails to identify several drift points due to the coarse granularity of the histogram, which smooths out important distributional changes. With 5 bins, detection improves, but two drift points remain undetected. At 8 bins, all injected drifts are accurately detected, indicating that this bin count provides a good balance between sensitivity and granularity. With 12 bins, the detector not only detects all drift points but also shows more pronounced deviations from the threshold line, suggesting increased robustness and clearer separation between normal fluctuations and true drift events. These results demonstrate that the number of bins plays a critical role in the effectiveness of distribution-based drift detection. Too few bins result in under-sensitivity, while too many bins may introduce computational overhead without significant performance gains beyond a certain point.

To evaluate the robustness of our approach under more complex traffic patterns, we design two mixed-attack scenarios, as summarized in Table 6.1. For both scenarios, we employ 12 histogram bins to capture feature distributions. As shown in Figure 6.3, when the window size is set to 100, the detector fails to identify some of the injected drift

points. Moreover, the TVD values exhibit significant fluctuation between drift intervals, particularly in the inter-category scenario, making it more difficult to distinguish between normal variation and true distributional shifts. In contrast, increasing the window size to 200 enables SPIDD to detect all drift points accurately, while the TVD curve becomes more stable between drift events. The reduced detection performance with a window size of 100 in mixed-attack scenarios, compared to single-attack settings, can be attributed to the higher variability and distributional overlap introduced by mixing multiple attack types. A smaller window collects fewer samples, which may not be representative enough to capture the composite distributional shifts caused by such heterogeneous attack behaviors. Consequently, the computed histogram may fluctuate due to statistical noise, obscuring meaningful changes and leading to missed detections. In contrast, a larger window (e.g., size 200) aggregates more samples, providing a more stable and reliable estimate of the underlying distribution. This allows SPIDD to distinguish genuine drifts from random fluctuations more effectively, especially in scenarios where the attack-induced changes are subtle or occur gradually due to mixed behaviors.

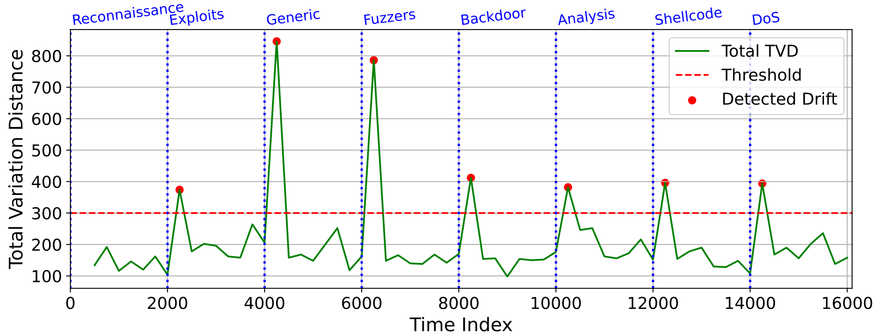


Figure 6.4: Drift Detection Performance on UNSW-NB15.

For UNSW-NB15 dataset, we configure the drift detector with a window size of 250 and employ 12 histogram bins to capture the distribution of features. As shown in Figure 6.4, SPIDD successfully detects all drift events across the evaluation timeline.

Memory Overhead: In our BMv2 testing environment, we estimate that the register array requires less than 0.5KB to store feature distributions, assuming 7 features and 8 bins per feature with 32-bit counters. For bin mapping, we use match-action tables with one table per feature and a total of 56 range entries. While BMv2 does not reflect hardware memory constraints, the small number of entries indicates that the drift detector design remains lightweight and practical for deployment on other resource-constrained targets.

6.6 Discussion

Adaptability to Gradual Drift: While our evaluation focuses on scenarios involving sudden drift, it is important to consider how the proposed method performs under gradual drift conditions, where the data distribution changes incrementally over time. Since SPIDD compares traffic distributions across sliding windows, it has the potential to capture gradual shifts if the window size and detection threshold are appropriately configured.

Drift Detector Update: As traffic patterns evolve, it is necessary to update the bin configurations to maintain detection accuracy. Bin edges can be updated at runtime because the system uses match-action tables with range-type keys to locate the bins. Updating the bin edges only requires modifying the table entries, which can be done atomically without any switch downtime. However, changing the number of bins in a distribution is more disruptive, as it requires modifications to the P4 program itself. In such cases, traffic must be temporarily halted during the update process.

6.7 Conclusion

In this chapter, we presented an unsupervised drift detection method for data-planes. Our approach is model-agnostic, as it operates by tracking changes in the distribution of traffic features. We utilize total variation distance to quantify distributional shifts and identify potential drift events. Experimental evaluations on both single-attack and mixed-attack scenarios demonstrate the effectiveness of our approach in accurately detecting concept drift. Future work will explore the detection of additional drift types, including gradual and incremental drift, and to integrate SPIDD into a complete pipeline with automated labeling, retraining, and redeployment to sustain long term in-network ML performance.

Chapter 7

Towards In-Network Drift-Aware Traffic Classification

7.1 Introduction

Existing intelligent data-plane approaches [50,58,72] typically operate under a closed-world assumption, where the distribution of traffic observed during deployment is expected to closely match that of the training data. In other words, they are designed to classify only previously known traffic classes, referred to as in-distribution samples, that appeared in the original training set. In practice, network environments are highly dynamic and subject to continuous changes. This results in concept drift, which refers to the emergence of previously unseen traffic classes or substantial shifts in the behavior of known traffic patterns, neither of which is represented in the original training set [148]. As a result, the predictive performance and reliability of the deployed machine learning (ML) models can degrade significantly over time.

The previous chapter addressed this issue through unsupervised drift detection, which triggers model retraining whenever significant shifts in traffic distribution are detected. Typically, this involves collecting new traffic samples, labeling them, and updating the model offline before redeployment. However, retraining effectiveness hinges on the timely collection of diverse and representative samples. Delayed collection or insufficient coverage of emerging traffic patterns often results in models that fail to generalize to the new environment. Existing sampling-based strategies [151] further limit retraining quality by capturing only a small subset of drifting samples. Moreover, most approaches lack mechanisms for explicitly identifying drifting samples in real time, resulting in a disconnect between drift manifestation and model adaptation [1, 73]. This highlights a critical gap:

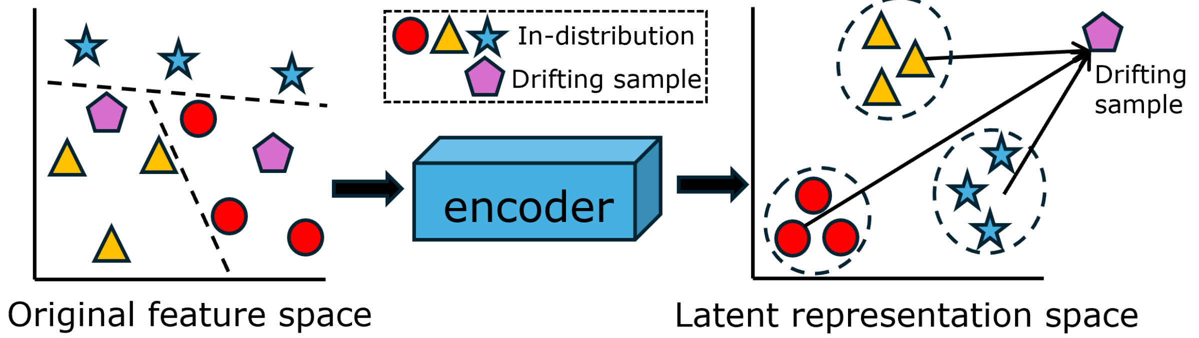


Figure 7.1: The mapping from original feature space to latent space. In the latent space, the drifting sample is easier to identify due to its large distance from the in-distribution clusters.

between the moment new or shifted traffic patterns emerge and the point at which retraining is initiated, there is limited in-network intelligence to flag and collect such drifting samples.

To bridge this gap, we propose IDAC, an **I**n-network **D**rift-**A**ware **C**lassification framework that simultaneously enables the classification of in-distribution samples and the detection of concept drift samples directly within the data-plane. By enabling real-time identification of traffic instances that deviate from the training distribution, IDAC supports selective logging, adaptive mitigation, and efficient sample collection for future retraining. IDAC enhances existing in-network learning-based functions by enhancing their robustness in open-world network environments. IDAC adopts a distance-based detection strategy using a triplet ML network to learn a similarity function over labeled training data. As shown in Figure 7.1, this model maps high-dimensional traffic instances into a significantly lower-dimensional latent representation space where samples from the same class are embedded closer together, while those from different classes are more distinctly separated. In this latent space, IDAC computes centroids for each known class and measures the distance of incoming traffic samples to these centroids. Samples that fall significantly far from all known centroids are flagged as potential concept drift instances, whereas samples within acceptable proximity are classified based on their nearest centroid. In summary, the key contributions of this chapter are as follows:

- We introduce IDAC, an in-network drift-aware traffic classification framework that not only supports accurate classification of in-distribution traffic but also enables detection of drifting samples directly in the data-plane.
- We develop a distance-based drift detection approach that employs a triplet network trained with triplet loss and center loss to construct a compact and discriminative

latent space, facilitating both robust classification and effective drift identification.

- IDAC is data-plane friendly and can be efficiently deployed in programmable switches, enabling real-time monitoring in dynamic network environments.

We evaluate IDAC on two use cases and demonstrate its effectiveness in detecting drifting samples without compromising the classification accuracy of in-distribution data.

7.2 Related Work

Recent efforts have explored the deployment of various ML models in programmable data-planes, including decision trees [1], random forests [52], multi-layer perceptrons [19], binary neural networks [73], and recurrent neural networks [99], targeting both software and hardware switches. These models have been used to support tasks such as attack detection [58], traffic classification [50], and IoT device identification [72]. These models perform inference via match-action rules and can be updated on the fly without incurring switch downtime. However, existing solutions primarily focus on in-distribution inference and do not support the real-time detection of drifting samples within the data-plane.

Prior approaches for detecting drifting samples involve monitoring the prediction confidence of the original classifier [156]. The intuition is that samples with low confidence scores may deviate from the training distribution and thus represent potential drift. However, prediction confidence is a probability, which assume a closed-world setting where all possible classes are known a priori. As a result, drifting samples may still be assigned high confidence scores for incorrect classes, as evidenced by prior work [157, 158]. Conversely, a sample may have low confidence simply because it lies near a decision boundary between known classes, not because it is out-of-distribution. These limitations significantly undermine the reliability of confidence-based drift detection. In contrast, we adopt a distance-based detection approach that evaluates the similarity of incoming samples to known class distributions in a learned latent space, offering a more robust mechanism for identifying drift.

7.3 In-Network Drifting Samples Detection

Detecting drifting samples requires the construction of an effective distance function capable of distinguishing such samples from those belonging to the known distribution. To achieve this, we leverage contrastive learning [159] to learn a discriminative representation

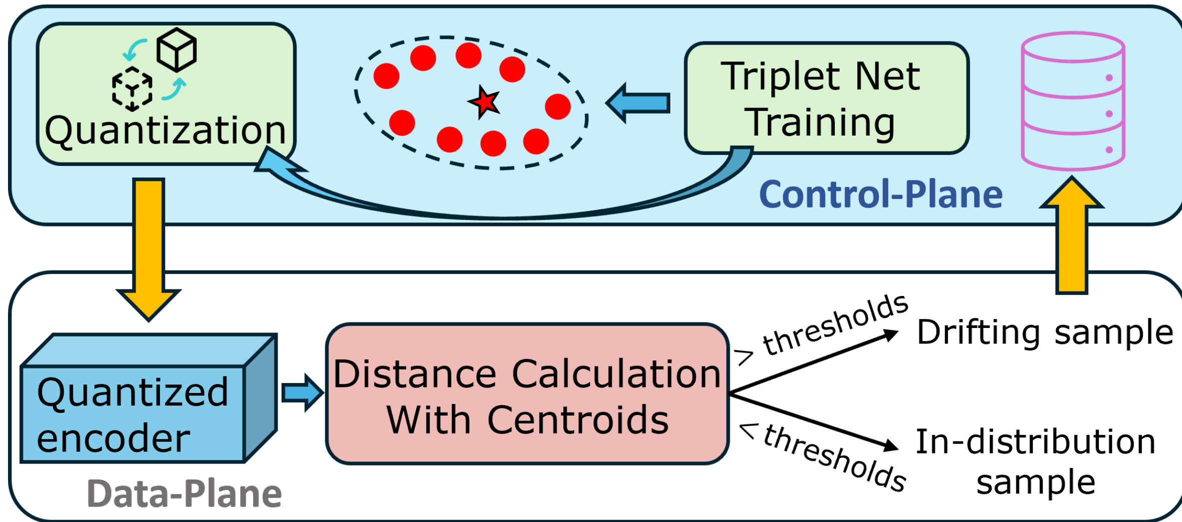


Figure 7.2: IDAC architecture.

of the training data. Contrastive learning utilizes the existing class labels to train a model that embeds samples into a latent space where semantic similarities and dissimilarities are preserved. As illustrated in Figure 7.1, given high-dimensional input traffic features, the contrastive learning framework learns to project samples into a low-dimensional latent space such that intra-class samples are mapped closer together, while inter-class samples are pushed further apart. This structure enables the use of distance metrics in the latent space to quantify the degree of deviation between incoming samples and known class distributions. Samples exhibiting a large distance from all known class centroids are flagged as potential drifting instances. To realize this approach, we employ a triplet neural network [160], as detailed below.

The overall architecture of IDAC is presented in Figure 7.2. Built on the software-defined networking (SDN) paradigm, IDAC consists of two main components: the control-plane and the data-plane. The control-plane component of IDAC is responsible for training the triplet network, quantizing one encoder, computing class centroids, and generating match-action tables for deployment in the data-plane. The data-plane component of IDAC performs real-time inference by mapping incoming traffic features into the latent space using the quantized encoder, followed by distance-based drift detection. When a drifting sample is identified, its information is forwarded to the control-plane. Upon receiving the reported drifting sample, the control-plane labels and stores it in the data collector for next retraining.

7.3.1 Triplet Network

To learn an effective distance function and enable drift detection, we adopt a triplet network architecture. As illustrated in Figure 7.3, it consists of three identical encoders with shared parameters. Each encoder processes one element of a input and transforms it into a fixed-length feature representation. Each input $\mathbf{x} \in \mathbb{R}^p$ represents a traffic sample (e.g., a flow) characterized by p features. The encoder function $f(\cdot)$ is implemented using several fully-connected layers that map $\mathbf{x}$ from the original feature space to a low-dimensional latent representation $\mathbf{z} = f(\mathbf{x}) \in \mathbb{R}^q$, where $q \ll p$. The shared-weight configuration ensures that the three encoders learn the same mapping function, enabling the model to assess the similarity between input samples based on their respective embeddings.

Suppose the training set consists of N classes, where each class is denoted by $i \in \{1, 2, \cdots, N\}$. Before training the triplet network, we first construct a batch of triplets, each consisting of three traffic samples: $(\mathbf{x}_a, \mathbf{x}^+, \mathbf{x}^-)$. Here, $\mathbf{x}_a$ is referred to as the anchor, which serves as the reference point for measuring similarity and dissimilarity. The sample $\mathbf{x}^+$ is a positive sample that belongs to the same class as the anchor, while $\mathbf{x}^-$ is a negative sample from a different class than the anchor. These inputs are independently processed by a shared-weight encoder $f(\cdot)$, producing corresponding embeddings $\mathbf{z}_a = f(\mathbf{x}_a)$, $\mathbf{z}^+ = f(\mathbf{x}^+)$, and $\mathbf{z}^- = f(\mathbf{x}^-)$. During training, a triplet loss function [161] is designed to encourage the network to learn $f(\cdot)$ that can bring samples of the same class closer together in the latent space while pushing samples from different classes farther apart. Specifically, it enforces that the distance between the anchor and the positive sample should be smaller than the distance between the anchor and the negative sample by at least a predefined margin $m > 0$:

$$\mathcal{L}_{\text{triplet}} = \mathbb{E}_{(\mathbf{x}_a, \mathbf{x}^+, \mathbf{x}^-)} \left[\max \left(0, S(\mathbf{z}_a, \mathbf{z}^+) - S(\mathbf{z}_a, \mathbf{z}^-) + m \right) \right] \tag{7.1}$$

where $S(\cdot, \cdot)$ represents the distance between samples. The expectation operator $\mathbb{E}_{(\mathbf{x}_a, \mathbf{x}^+, \mathbf{x}^-)}[\cdot]$ indicates that the loss is averaged over all triplets in the training batch. Intuitively, if the distance between the anchor and the negative is already larger than that of the positive by at least m, the triplet is considered "satisfied" and contributes no loss. Otherwise, the network is penalized, and its parameters are updated to reinforce the desired margin.

While triplet loss effectively enforces inter-class separation by ensuring that embeddings of dissimilar classes are pushed apart, it does not explicitly constrain the spatial distribution of samples within the same class. As a result, intra-class variance may remain high, leading to dispersed clusters. To address this limitation, we incorporate a center loss, which penalizes the distance between sample embeddings and their corresponding class

centers in the latent space. Center loss promotes intra-class compactness, encouraging embeddings of the same class to cluster tightly around a learned prototype, defined as:

$$\mathcal{L}_{\text{center}} = \sum_{i=1}^{N} \frac{1}{|B_i|} \sum_{\mathbf{x} \in B_i} S(\mathbf{z}, \mathbf{z}_c^i) \tag{7.2}$$

where B_i denotes a subset of batch training samples belonging to class i, and $\mathbf{z}_c^i$ denotes the center of class i in the latent space. The overall training objective is defined as a weighted sum of the triplet and center losses:

$$\mathcal{L} = (1-\lambda)\mathcal{L}_{\text{triplet}} + \lambda\mathcal{L}_{\text{center}} \tag{7.3}$$

where $\lambda > 0$ controls the contribution of center loss.

To ensure smooth convergence, we update each center using an exponential moving average based on the current mini-batch statistics [162]. Specifically, for each class $i \in \{1, 2, \cdots, N\}$, we compute the mean embedding $\bar{\mathbf{z}}^i$ of all samples belonging to class i within the mini-batch. The corresponding center $\mathbf{z}_c^i$ is then updated as:

$$\mathbf{z}_c^i \leftarrow (1-\alpha) \cdot \mathbf{z}_c^i + \alpha \cdot \bar{\mathbf{z}}^i \tag{7.4}$$

where $\alpha \in (0, 1)$ is a smoothing factor that controls the update rate.

In principle, the distance function $S(\cdot, \cdot)$ can be instantiated using any differentiable metric, such as cosine similarity, Mahalanobis distance, or Euclidean distance. In this work, we adopt the squared Euclidean distance, i.e., $S(\mathbf{z}_a, \mathbf{z}^+) = \|\mathbf{z}_a - \mathbf{z}^+\|_2^2$. This choice is motivated by its computational simplicity and alignment with the constraints of programmable data-planes. In particular, P4 does not support complex mathematical operations such as square roots, but the squared Euclidean distance requires only basic arithmetic operations, making it feasible for efficient in-network execution.

After training, we obtain the floating-point (FP) encoder model, where both the weights and activations are represented using FP values. As data-planes cannot perform FP operations, the weights of the encoder are restricted to fixed-point representations when stored in the data-plane. Therefore, we employ post-training quantization [97] to transform the FP-based model to a quantized model which represents weights and activations using more compact format (e.g., 8-bit integers) [16].

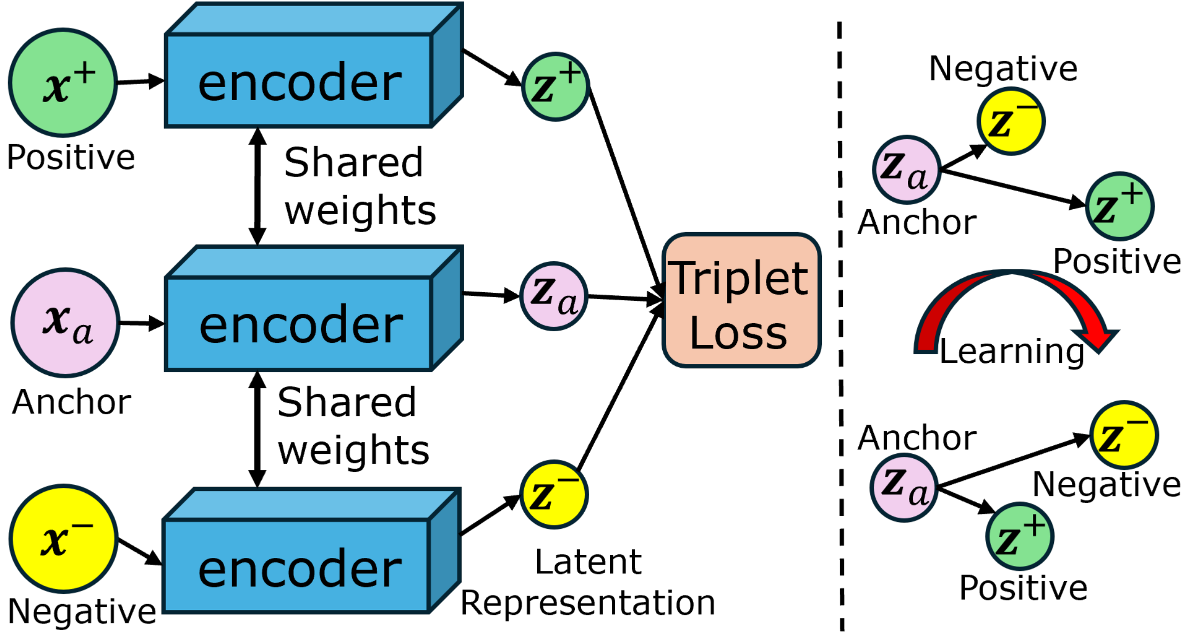


Figure 7.3: Triplet network architecture (left) and the triplet loss objective (right), which pulls the anchor closer to the positive and pushes it away from the negative.

7.3.2 Distance-based Drifting Samples Detection

After training the triplet network and quantizing one of its encoders, we employ the quantized encoder to perform drifting sample detection. We first use the quantized encoder to map all the training samples into the latent space. For each class i, we obtain the class center $\mathbf{z}_c^i$. Given a testing sample $\mathbf{x}_t$, we use the encoder to map it to the latent space representation $\mathbf{z}_t$. Then, we calculate the squared Euclidean distance between the testing sample and each of the centroids $\mathbf{z}_c^i$ as: $||\mathbf{z}_t - \mathbf{z}_c^i||_2^2$. Based on these distances, we assess whether the testing sample lies within the expected distribution of any known class. If the sample is far from all class centroids, it is considered a candidate drifting sample. To determine whether a sample is drifting, we establish a threshold τ_i for each class by computing the β-percentile (e.g., $\beta = 0.99$) of the squared Euclidean distances between the training samples and their corresponding centroid. A testing sample $\mathbf{x}_t$ is flagged as drifting if $||\mathbf{z}_t - \mathbf{z}_c^i||_2^2 > \tau_i$ for all $i = 1, 2, \cdots, N$.

Since square root operations are not natively supported in P4, we use squared Euclidean distances both when computing distances and thresholds. This avoids runtime complexity while preserving the relative structure of the latent space, ensuring that drift detection remains effective and compatible with programmable data-plane environments.

7.4 P4 Implementation

In this section, we describe the implementation of the detection logic in the data-plane using P4 language.

7.4.1 Encoder

The incoming packet first passes through the parser, which extracts traffic features from the packet headers to serve as input to the encoder. The encoder is essentially a multi-layer perceptron (MLP), and its implementation has been investigated across various programmable data-plane targets. For targets that support arithmetic operations like multiplication (e.g., software switches), its implementation has been demonstrated in the INQ-MLT toolbox [16]. In contrast, switch ASICs lack native support for multiplication, so match-action tables are used to emulate these operations by storing precomputed mappings between input values and intermediate results, as shown in MUTA [19]. The latent embedding produced by the encoder is subsequently used to compute the distance to class centroids.

```
z1,z2,z3 = MLP(x1,x2,...,xp); // encoder inference
action ac_z1(bit<32> y11, bit<32> y12){
    meta.dist1 = y11;
    meta.dist2 = y12;}
action ac_z2(bit<32> y21, bit<32> y22){
    meta.dist1 = meta.dist1 + y21;
    meta.dist2 = meta.dist2 + y22;}
action ac_z3(bit<32> y31, bit<32> y32){
    meta.dist1 = meta.dist1 + y31;
    meta.dist2 = meta.dist2 + y32;}
table tb_z1{
    key = {meta.z1: exact;}
    actions = {ac_z1;}
    size = 256;}
table tb_z2{
    key = {meta.z2: exact;}
    actions = {ac_z2;}
    size = 256;}
table tb_z3{
    key = {meta.z3: exact;}
    actions = {ac_z3;}
    size = 256;}
if meta.dist1 > thres1 && meta.dist2 > thres2{
    report to control-plane;
}
else {
    meta.class = argmax(); //find the nearest centroid
}
```

Listing 7.1: P4 code fragment illustrating detection logic for a three-dimensional latent vector with two class centroids.

7.4.2 Distance Calculation

Listing 7.1 presents the implementation logic for distance computation and drift detection within the data-plane. As an illustrative example, we consider a latent space representation $\mathbf{z} = (z_1, z_2, z_3)$. Each match action table is used to process the kth element of the latent vector, denoted as z_k. These tables store precomputed squared difference values, specifically $(z_k - c_{k1})^2$, $(z_k - c_{k2})^2$, ..., $(z_k - c_{kN})^2$, for all quantized values of z_k, where N is the number of centroids. For example, when z_1 is used as the key, the corresponding match-action table retrieves precomputed values y_{11} and y_{12} (line 2), which represent the squared differences $(z_1 - c_{11})^2$ and $(z_1 - c_{12})^2$, respectively, thereby avoiding the need for multiplication operations during runtime. This procedure is repeated for the remaining elements z_2 and z_3. The retrieved partial results are then accumulated using addition operations to compute the total squared Euclidean distance. If the computed distance to any class centroid exceeds its associated threshold (line 23), a corresponding response is triggered, such as reporting the sample to the control-plane.

If a sample is not identified as a drifting instance, its class is assigned based on the nearest class centroid, determined through an argmax operation. This argmax computation can be implemented using either nested if-else logic or a ternary matching table [99], the latter offering improved scalability in practical deployment.

7.5 Performance Evaluation

7.5.1 Dataset

IoT Traffic Classification: The objective is to classify each network flow with its corresponding IoT device type while also identifying traffic from previously unseen devices. The classification outcomes provide early insights into the nature of traffic (e.g., application type or data type), thereby allowing network operators to make informed decisions tailored to specific devices, applications, or data categories. Such early-stage classification can be leveraged to enhance quality of service (QoS) and quality of experience (QoE). For instance, traffic from IoT video cameras may indicate a live video session, which requires routing through less congested paths to reduce latency and retransmissions, both of which are essential for maintaining high video quality. Following prior work [73, 151], we use the UNSW-IoT dataset [163], defining ten device categories: nine specific types (iot1–iot9), such as cameras, sensors, and home assistants, and one additional class (iot0) for miscellaneous devices (e.g., smartphones and laptops). For clarity, each category is assigned a

Table 7.1: The training scenarios (UNSW-IoT)

Known	Scenario1	Scenario2	Scenario3
	iot1:dropcam	iot2:amazon echo	iot3:smart things
	iot4:baby monitor	iot5:insteon monitor	iot6:samsung smartcam
	iot7:triby speaker	iot8:motion censor	iot9:smart sleep sensor

unique index. As shown in Table 7.1, we evaluate three scenarios, each including three known IoT categories, with the rest treated as unknown traffic.

Intrusion Detection: The objective is to perform binary classification of network flows as either Benign or Malicious, while simultaneously identifying unknown attack traffic not seen during training. Following prior work [148], we utilize the CICIDS2018 dataset [108], which includes a diverse range of network traces generated by both normal activity and various attack types. In our setup, malicious traffic comprises multiple known attack categories. We also define three scenarios, as shown in Table 7.3, each consisting of benign traffic and two specific attack types used for training. All remaining attack types are treated as unknown and reserved for testing.

7.5.2 Experimental Setting

The model training and quantization operations are performed by the control-plane using TensorFlow Lite.[1]. The detection logic is implemented in BMv2 software switches using P4 with v1model architecture. The known traffic is divided into training and testing subsets with a 60:40 split. The final testing set includes both the testing portion of known traffic and the previously unseen (unknown) traffic. We only use the training data to train the triplet Network and calculate the centroids. For the drifting detection, the positive samples are samples in the unseen family in the testing set. The negative samples are the rest of the testing samples from the known families. We calculate three evaluation metrics: precision, recall, and F1 score. Precision measures the ratio of true unseen-family samples out of the inspected samples. Recall measures the ratio of unseen-family samples that are successfully discovered by the detection module out of all the unseen-family samples. F1-score is the harmonic mean of precision and recall. In addition to these, we report the classification accuracy on known traffic and an overall metric known as the purity rate [164], which captures the combined effectiveness of known-class classification and unknown-class detection. The purity rate is defined as follows:

[1]https://www.tensorflow.org/lite

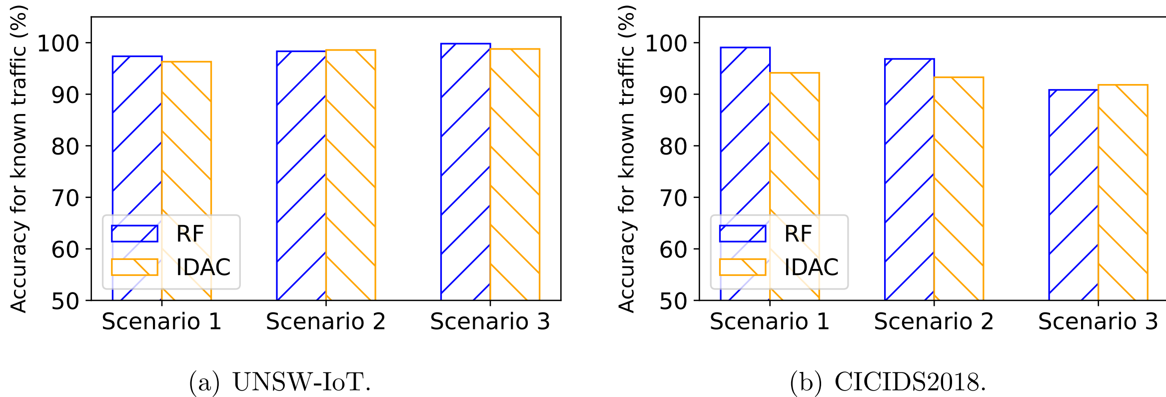
Table 7.2: Drift detection results for UNSW-IoT dataset

		Scenario1							
		RF				IDAC			
		Prec	Recall	F1	Purity	Prec	Recall	F1	Purity
unknown devices	iot0	95.99	75.80	84.71	87.56	94.87	81.33	87.58	**89.50**
	iot2	92.47	38.87	54.73	70.77	95.26	88.47	91.74	**92.74**
	iot3	89.86	28.07	42.77	65.86	95.75	99.20	97.45	**97.62**
	iot5	89.46	26.87	41.32	65.32	95.77	99.73	97.71	**97.86**
	iot6	94.07	50.20	65.46	**75.92**	89.08	35.90	51.18	68.85
	iot8	96.89	98.53	97.70	97.89	95.75	99.17	97.43	**97.61**
	iot9	88.90	25.37	39.47	64.64	95.54	94.30	94.92	**95.39**
		Scenario2							
		RF				IDAC			
		Prec	Recall	F1	Purity	Prec	Recall	F1	Purity
unknown devices	iot0	97.99	90.93	94.33	94.97	98.32	95.40	96.84	**97.14**
	iot1	92.73	23.80	37.88	64.45	98.34	96.97	97.65	**97.85**
	iot3	98.17	99.90	99.03	99.05	98.39	100.0	99.19	**99.23**
	iot4	97.64	77.13	86.19	88.70	98.39	100.0	99.19	**99.23**
	iot6	98.17	100.0	99.08	**99.09**	98.34	96.67	97.50	97.71
	iot7	98.16	99.80	98.98	99.00	98.39	99.97	99.17	**99.21**
	iot9	98.10	96.27	97.18	97.39	98.38	99.47	98.92	**98.98**
		Scenario3							
		RF				IDAC			
		Prec	Recall	F1	Purity	Prec	Recall	F1	Purity
unknown devices	iot0	99.67	69.00	81.54	85.80	98.56	95.87	97.20	**97.45**
	iot1	99.77	99.77	99.77	**99.79**	98.62	100.0	99.30	99.33
	iot2	98.03	11.60	20.75	59.71	98.52	93.10	95.73	**96.20**
	iot4	77.42	08.00	01.58	54.80	98.56	95.87	97.20	**97.45**
	iot5	99.33	34.53	51.25	70.14	98.62	100.0	99.30	**98.33**
	iot7	98.89	20.73	34.28	63.86	98.59	97.80	98.19	**98.33**
	iot8	99.77	99.77	99.77	**99.79**	98.61	99.53	99.07	99.12

$$\text{Purity} = \frac{N_{\text{known}}^{\text{correct}} + N_{\text{unknown}}^{\text{correct}}}{\#\text{Total testing samples}} \tag{7.5}$$

where $N_{\text{known}}^{\text{correct}}$ denotes the number of correctly classified known samples, and $N_{\text{unknown}}^{\text{correct}}$ denotes the number of correctly identified unknown samples.

Given that most existing in-network ML approaches employ random forests (RF) as the classification model [1, 52], we adopt RF as our baseline method. RFs provide class confidence estimates by averaging the predicted class probabilities across all decision trees



(a) UNSW-IoT.

(b) CICIDS2018.

Figure 7.4: Performance comparison of known traffic classification accuracy.

in the ensemble. Each tree computes class probabilities based on the proportion of training samples of each class within the corresponding leaf node. In our setup, we define a confidence threshold of 0.8: samples with confidence scores below this threshold are considered potential drifting samples; otherwise, accept the label.

7.5.3 Results

Tables 7.2 and 7.3 demonstrate that the proposed IDAC method consistently outperforms the RF baseline in detecting drifting traffic across both the UNSW-IoT and CICIDS2018 datasets. Specifically, IDAC achieves higher recall and F1 scores, indicating superior sensitivity and overall effectiveness in identifying previously unseen samples. For the UNSW-IoT dataset, Figure 7.4 further shows that IDAC maintains almost the same classification accuracy on known traffic, which contributes to a higher overall purity rate.

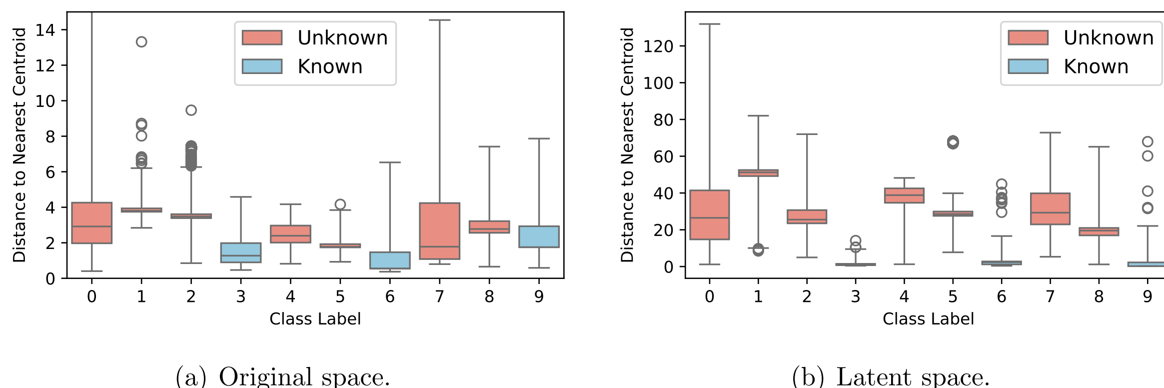
For the CICIDS2018 dataset, IDAC demonstrates substantially stronger drift detection capability compared to RF. In Scenario1, for instance, the RF model fails to detect any unknown samples due to its overconfident predictions on unfamiliar inputs. While IDAC significantly improves recall and F1 scores, its precision on CICIDS2018 is relatively lower than on UNSW-IoT. This performance gap can be attributed to the high variability of benign traffic patterns in CICIDS2018, which makes it challenging for the model to learn a compact and discriminative representation of benign behavior. As a result, some benign samples are misclassified as unknown. Moreover, Figure 7.4 shows that IDAC incurs an accuracy drop of approximately $3\%-4\%$ on known traffic in Scenarios1 and 2. Nonetheless, it is acceptable to sacrifice a small amount of accuracy in exchange for improved sensitivity to drifting samples, which is critical in dynamic and evolving network environments.

To further demonstrate the advantage of IDAC, we present a box plot analysis in Fig-

Table 7.3: Drift detection results for CICIDS2018 dataset

		Scenario1:Benign - FTP-BruteForce - SSH-BruteForce							
		RF				IDAC			
		Prec	Recall	F1	Purity	Prec	Recall	F1	Purity
unknown attacks	Bot	00.00	00.00	00.00	54.04	88.03	51.09	64.66	**74.58**
	DD-hoic	00.89	00.01	00.02	54.04	91.68	76.59	83.46	**86.17**
	DD-loic	00.00	00.00	00.00	54.04	85.61	41.33	55.75	**70.14**
	D-eye	00.00	00.00	00.00	54.04	85.12	39.75	54.19	**69.42**
	D-hulk	03.05	00.04	00.07	54.05	93.20	95.22	94.20	**94.64**
	D-http	09.37	00.12	00.23	**54.09**	00.00	00.00	00.00	51.36
	Infiler	90.37	10.44	18.71	58.78	79.02	26.17	39.31	**63.25**
		Scenario2:Benign - Bot - DDoS-HOIC							
		RF				IDAC			
		Prec	Recall	F1	Purity	Prec	Recall	F1	Purity
unknown attacks	DD-loic	93.11	50.30	65.31	75.69	91.33	84.68	87.88	**89.38**
	D-eye	80.12	15.01	25.28	59.65	92.04	92.95	92.49	**93.14**
	D-hulk	96.08	91.20	93.57	94.28	92.56	99.99	96.13	**96.34**
	D-http	00.00	00.00	00.00	52.83	92.56	100.0	96.14	**96.34**
	FTP-BF	00.00	00.00	00.00	52.83	92.56	100.0	96.13	**96.34**
	Infiler	61.14	05.86	10.69	55.49	75.56	24.86	37.41	**62.19**
	SSH-BF	00.20	00.01	00.01	52.83	86.42	51.15	64.27	**74.14**
		Scenario3:Benign - DDoS-LOIC-HTTP - DoS-GoldenEye							
		RF				IDAC			
		Prec	Recall	F1	Purity	Prec	Recall	F1	Purity
unknown attacks	Bot	00.86	00.10	00.17	49.59	84.99	46.45	60.07	**71.20**
	DD-hoic	90.09	99.91	94.75	94.96	92.42	99.95	96.04	**95.52**
	D-hulk	90.02	99.08	94.33	**94.59**	92.06	95.06	93.54	93.29
	D-http	00.00	00.00	00.00	49.55	92.42	100.0	96.06	**95.54**
	FTP-BF	00.00	00.00	00.00	49.55	92.42	100.0	96.06	**95.54**
	Infiler	28.96	04.48	07.76	51.58	72.52	21.63	33.32	**59.92**
	SSH-BF	81.96	49.94	62.06	72.25	92.08	95.24	93.63	**93.38**

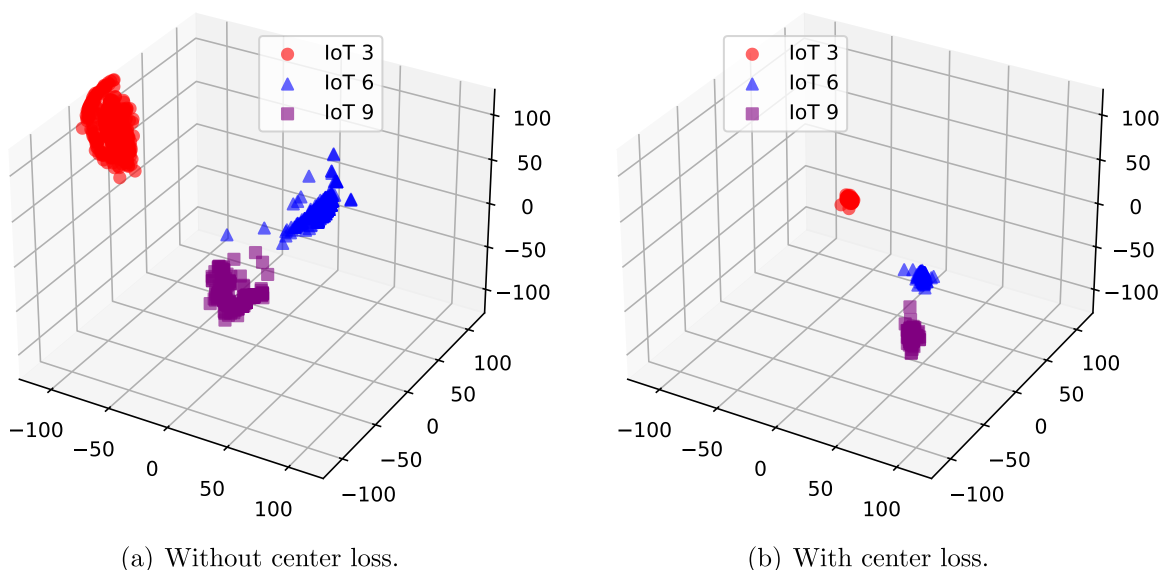
ure 7.5 using Scenario3 of the UNSW-IoT dataset. For each testing sample, we measure the distance to its nearest class centroid in both the original feature space and the latent space learned by IDAC. The results reveal that drifting (unknown) and non-drifting (known) samples exhibit significant overlap in the original space, making them difficult to distinguish. In contrast, the latent space learned through contrastive learning achieves a more distinctive separation between the two groups. Specifically, known samples exhibit tightly clustered distances (i.e., lower spread), while unknown samples show larger and more dispersed distance distributions. This improved separation is attributed to the contrastive



(a) Original space.

(b) Latent space.

Figure 7.5: Boxplots of distances between testing samples and their nearest centroids for each class. Known traffic are classes 3, 6, and 9.



(a) Without center loss.

(b) With center loss.

Figure 7.6: Latent space visualization without (left) and with (right) center loss.

learning objective, which encourages the model to pull samples from the same class closer together while pushing samples from different classes further apart, thereby enhancing the detectability of previously unseen traffic families.

We further investigate the role of center loss in structuring the latent space. Figure 7.6 presents a comparative visualization of latent embeddings obtained with and without the center loss objective. When center loss is applied, samples from the same class form more compact clusters in the latent space. This tighter clustering enhances the model's ability to distinguish known classes from unseen traffic patterns, as drifting samples are more likely to lie far from any class center.

7.6 Conclusion

In this chapter, we demonstrate the feasibility of integrating drift-aware classification into intelligent data-planes. By training a triplet network with triplet loss and center loss, we obtain a latent representation that encourages compact clustering of in-distribution samples. This structure enables effective detection of drifted samples based on their distance from class centroids. Evaluations across two use cases confirm that our method achieves more reliable drifting sample detection than confidence-based baselines, while preserving classification accuracy on known traffic. These results highlight the importance of incorporating drift detection mechanisms into future intelligent data-plane designs.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis presents a set of contributions aimed at enabling intelligent, real-time network management in the programmable data-plane. First, we introduced INQ-MLT, an innovative toolbox designed to facilitate the training and deployment of machine learning (ML) models within intelligent data-planes. INQ-MLT adopts quantization-aware training to mitigate accuracy loss by simulating quantization during training; after training, weights and activations are converted from floating-point to compact integer representations. Second, we developed a two-stage, confidence-based intrusion detection system that combines early- and late-stage CNNs to improve both detection speed and reliability. Post-hoc calibration is applied to align confidence with accuracy, while a parameter sharing mechanism is applied to reduce data-plane resource usage. Third, we proposed MUTA, an in-network multi-task learning framework that improves resource efficiency and predictive accuracy when deploying multiple management tasks. MUTA builds a multi-task neural network and allows the sharing of feature representations across related tasks. It supports distributed inference by splitting the model layer by layer, while ensuring full-network coverage and maintaining inference integrity. Fourth, we designed an unsupervised drift detection mechanism that monitors feature distributions over time using dual sliding windows and identifies concept drift without requiring labeled data, ensuring sustained ML model performance in dynamic environments. Finally, we designed IDAC, an in-network drift-aware traffic classification framework that not only supports accurate classification of in-distribution traffic but also enables detection of drifting samples directly in the data-plane. These contributions collectively demonstrate the feasibility and effectiveness of integrating advanced ML capabilities into the programmable data-plane, enabling real-time, adaptive, and resource-efficient network traffic management.

8.2 Future Work

Based on the results of this thesis, there are several possible directions for future research.

Use Cases. This research focuses on intelligent data-planes (IDPs) for network traffic management, such as traffic classification and intrusion detection. However, its potential extends far beyond this specific domain. Although IDPs have begun to see adoption in areas such as finance and smart grid systems, their current range of use cases remains relatively limited. Future work can explore broader applications, particularly in latency-sensitive domains such as e-health and autonomous vehicles. Additionally, IDPs hold promise for enhancing the performance of emerging communication infrastructures, including next-generation cellular (6G) and satellite networks.

Trust and Security. Although in-network ML has shown significant promise in cybersecurity applications, its own security and trustworthiness have received limited attention. Future research should focus on improving the interpretability of in-network ML models, developing resilient mechanisms to mitigate both operational failures and malicious threats, and designing robust strategies for handling failure modes within resource-constrained data-plane environments. Additionally, adversarial robustness remains a critical yet underexplored challenge. Ensuring that in-network ML models can withstand adversarial inputs is essential for maintaining reliable operation in adversarial settings. Advancing these directions will be key to enhancing the trust, reliability, and practical deployment of intelligent in-network systems.

Hardware Deveplopment. Most existing works are built on programmable switches, SmartNICs, and FPGAs, with prototypes often evaluated using software targets such as BMv2 and T4P4S [165]. However, hardware limitations, including restricted compute and memory resources, constrain the complexity of deployable machine learning models. The recent discontinuation of Intel Tofino raises concerns about the long-term support for high-performance programmable switches. Looking ahead, designing future programmable devices with enhanced capabilities, such as native support for low-precision machine learning operations, without compromising network performance, remains a key challenge. Enhancing hardware functionality can significantly expand the range, scalability, and effectiveness of in-network ML models.

eBPF Integration. Beyond P4, other network programmability paradigms such as eBPF (extended Berkeley Packet Filter) [166] can also play a vital role in enhancing the capabilities of intelligent data-planes. eBPF provides greater flexibility by allowing the execution of sandboxed programs with bounded execution time directly within the operating system kernel. This enables seamless interaction with various kernel subsystems, includ-

ing the network stack. While eBPF is not confined to networking alone, it has recently gained significant popularity in data-plane applications, particularly for high-performance and low-latency packet processing. In terms of programmability, eBPF is more expressive than P4, as it supports advanced constructs such as bounded loops and tail call chaining (allowing up to 32 interconnected programs). These capabilities make it suitable for implementing complex logic, including custom parsers and regular expression matching. Recent studies [167] have utilized eBPF to extract rich, flow-level features that are challenging to compute efficiently in P4 due to hardware and memory constraints. These features include temporal and bidirectional flow statistics computed over sliding time windows, such as standard deviation, directional covariance, and jitter (i.e., variation in packet inter-arrival times). The extracted features are embedded into packet headers via in-band telemetry and delivered to programmable switches where the ML inference models are deployed. Beyond feature extraction, the broader potential of eBPF in augmenting intelligent data-plane functionality remains a promising avenue for future exploration.

List of Publications

Conferences Publication:

- **Kaiyi Zhang**, Nancy Samaan, Ahmed Karmouch and Leandros Tassiulas. Towards In-Network Drift-Aware Traffic Classification. *IEEE Conference on Network Functions Virtualization and Software-Defined Networking (NFV-SDN)*, 2025.
- **Kaiyi Zhang**, Nancy Samaan, Ahmed Karmouch and Leandros Tassiulas. Towards Unsupervised Drift Detection in Programmable Data-Planes. *ACM CoNEXT Workshop on In-Network Computing and AI for Distributed Systems (INCAS)*, 2025.
- **Kaiyi Zhang**, Changgang Zheng, Nancy Samaan, Ahmed Karmouch and Noa Zilberman. MUTA: Enabling Multi-Task Neural Network Inference in Programmable Data-Planes. *IEEE International Conference on High Performance Switching and Routing (HPSR)*, 2025. [**Best Paper Award**]
- **Kaiyi Zhang**, Nancy Samaan and Ahmed Karmouch. A Two-Stage Confidence-Based Intrusion Detection System in Programmable Data-Planes. *IEEE Global Communications Conference (GLOBECOM)*, 2023.
- **Kaiyi Zhang**, Nancy Samaan and Ahmed Karmouch. An Intelligent Data-Plane with a Quantized ML Model for Traffic Management. *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2023.

Journals Publication:

- **Kaiyi Zhang**, Changgang Zheng, Nancy Samaan, Ahmed Karmouch and Noa Zilberman. Design, Implementation, and Deployment of Multi-Task Neural Networks in Programmable Data-Planes. *IEEE Transactions on Network and Service Management (TNSM)*, 2025.
- **Kaiyi Zhang**, Nancy Samaan and Ahmed Karmouch. A Machine Learning-based Toolbox for P4 Programmable Data-Planes. *IEEE Transactions on Network and Service Management (TNSM)*, 2024.

References

[1] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Iisy: Hybrid in-network classification using programmable switches. *IEEE/ACM Transactions on Networking*, 32(3):2555–2570, 2024.

[2] Michael Seufert and Irena Orsolic. Improving the transfer of machine learning-based video qoe estimation across diverse networks. *IEEE Transactions on Network and Service Management*, 21(3):2824–2836, 2023.

[3] Shahbaz Rezaei and Xin Liu. Multitask learning for network traffic classification. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2020.

[4] Mahmoud Abbasi, Amin Shahraki, and Amir Taherkordi. Deep learning for network traffic monitoring and analysis (ntma): A survey. *Computer Communications*, 170:19–41, 2021.

[5] Kamal Benzekki, Abdeslam El Fergougui, and Abdelbaki Elbelrhiti Elalaoui. Software-defined networking (sdn): a survey. *Security and communication networks*, 9(18):5803–5833, 2016.

[6] Junfeng Xie, F Richard Yu, Tao Huang, Renchao Xie, Jiang Liu, Chenmeng Wang, and Yunjie Liu. A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges. *IEEE Communications Surveys & Tutorials*, 21(1):393–430, 2018.

[7] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):1–99, 2018.

[8] Gabriel O Ferreira, Chiara Ravazzi, Fabrizio Dabbene, Giuseppe C Calafiore, and Marco Fiore. Forecasting network traffic: A survey and tutorial with open-source comparative evaluation. *IEEE Access*, 11:6018–6044, 2023.

[9] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(4):3133–3174, 2019.

[10] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine learning for networking: Workflow, advances and opportunities. *IEEE Network*, 32(2):92–99, 2017.

[11] Somayeh Kianpisheh and Tarik Taleb. A survey on in-network computing: Programmable data plane and technology specific applications. *IEEE Communications Surveys & Tutorials*, 25(1):701–761, 2022.

[12] Huiling Jiang, Qing Li, Yong Jiang, GengBiao Shen, Richard Sinnott, Chen Tian, and Mingwei Xu. When machine learning meets congestion control: A survey and comparison. *Computer Networks*, 192:108033, 2021.

[13] Kamran Shaukat, Suhuai Luo, Vijay Varadharajan, Ibrahim A Hameed, and Min Xu. A survey on machine learning techniques for cyber security in the last decade. *IEEE Access*, 8:222310–222354, 2020.

[14] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. An efficient design of intelligent network data plane. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6203–6220, 2023.

[15] Kaiyi Zhang, Nancy Samaan, and Ahmed Karmouch. An intelligent data-plane with a quantized ml model for traffic management. In *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2023.

[16] 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.

[17] Kaiyi Zhang, Nancy Samaan, and Ahmed Karmouch. A two-stage confidence-based intrusion detection system in programmable data-planes. In *GLOBECOM 2023-2023 IEEE Global Communications Conference*, pages 6850–6855. IEEE, 2023.

[18] Michael Crawshaw. Multi-task learning with deep neural networks: A survey. *arXiv preprint arXiv:2009.09796*, 2020.

[19] Kaiyi Zhang, Changgang Zheng, Nancy Samaan, Ahmed Karmouch, and Noa Zilberman. MUTA: Enabling Multi-Task Neural Network Inference in Programmable Data-Planes. In *2025 IEEE 26th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. IEEE, 2025.

[20] Kaiyi Zhang, Changgang Zheng, Nancy Samaan, Ahmed Karmouch, and Noa Zilberman. Design, implementation, and deployment of multi-task neural networks in programmable data-planes. *IEEE Transactions on Network and Service Management*, 2025.

[21] Kaiyi Zhang, Nancy Samaan, Ahmed Karmouch, and Leandros Tassiulas. Towards Unsupervised Drift Detection in Programmable Data-Planes. In *Proceedings of the ACM CoNEXT Workshop on In-Network Computing and AI for Distributed Systems (INCAS '25)*, pages 14–20, 2025.

[22] Kaiyi Zhang, Nancy Samaan, Ahmed Karmouch, and Leandros Tassiulas. Towards In-Network Drift-Aware Traffic Classification. In *2025 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–6. IEEE, 2025.

[23] Celio Trois, Marcos D Del Fabro, Luis CE de Bona, and Magnos Martinello. A survey on sdn programming languages: Toward a taxonomy. *IEEE Communications Surveys & Tutorials*, 18(4):2687–2712, 2016.

[24] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

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

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

[27] Nick McKeown. Pisa: Protocol independent switch architecture. In *P4 Workshop*, pages 1–22, 2015.

[28] P4-16 portable switch architecture (psa). https://p4.org/wp-content/uploads/sites/53/p4-spec/docs/PSA-v1.2.html. Accessed: 2025-11-27.

[29] P4 portable nic architecture (pna). https://p4.org/wp-content/uploads/sites/53/p4-spec/docs/PNA-v0.7.html. Accessed: 2025-11-27.

[30] v1model. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4. Accessed: 2024-01-14.

[31] Intel barefoot networks. p4-16 intel tofino native architecture. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf. Accessed: 2023-07-06.

[32] Changgang Zheng, Xinpeng Hong, Damu Ding, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. In-network machine learning using programmable network devices: A survey. *IEEE Communications Surveys & Tutorials*, 26(2):1171–1200, 2023.

[33] Vladimir Gurevich and Andy Fingerhut. P4-16 programming for intel tofino using intel p4 studio. In *2021 P4 Workshop*, 2021.

[34] Behavioral model version 2. https://github.com/p4lang/behavioral-model. Accessed: 2022-05-22.

[35] P4runtime specification. version 1.4.1. the p4.org api working group. https://p4.org/wp-content/uploads/sites/53/2024/10/P4Runtime-Spec-v1.4.1.html. Accessed: 2025-11-15.

[36] Fannia Pacheco, Ernesto Exposito, Mathieu Gineste, Cedric Baudoin, and Jose Aguilar. Towards the deployment of machine learning solutions in network traffic classification: A systematic survey. *IEEE Communications Surveys & Tutorials*, 21(2):1988–2014, 2018.

[37] Ling Xia Liao, Han-Chieh Chao, and Mu-Yen Chen. Intelligently modeling, detecting, and scheduling elephant flows in software defined energy cloud: A survey. *Journal of Parallel and Distributed Computing*, 146:64–78, 2020.

[38] Shahbaz Rezaei and Xin Liu. Deep learning for encrypted traffic classification: An overview. *IEEE Communications Magazine*, 57(5):76–81, 2019.

[39] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges. *IEEE Transactions on Network and Service Management*, 16(2):445–458, 2019.

[40] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(3):2224–2287, 2019.

[41] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[42] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

[43] D Arivudainambi, Varun Kumar KA, P Visu, et al. Malware traffic classification using principal component analysis and artificial neural network for extreme surveillance. *Computer Communications*, 147:50–57, 2019.

[44] Rohan Doshi, Noah Apthorpe, and Nick Feamster. Machine learning ddos detection for consumer internet of things devices. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 29–35. IEEE, 2018.

[45] Andrés Chartuni and José Márquez. Multi-classifier of ddos attacks in computer networks built on neural networks. *Applied Sciences*, 11(22):10609, 2021.

[46] Yair Meidan, Michael Bohadana, Asaf Shabtai, Juan David Guarnizo, Martín Ochoa, Nils Ole Tippenhauer, and Yuval Elovici. Profiliot: A machine learning approach for iot device identification based on network traffic analysis. In *Proceedings of the symposium on applied computing*, pages 506–509, 2017.

[47] Huaizheng Zhang, Linsen Dong, Guanyu Gao, Han Hu, Yonggang Wen, and Kyle Guan. Deepqoe: A multimodal learning framework for video quality of experience (qoe) prediction. *IEEE Transactions on Multimedia*, 22(12):3210–3223, 2020.

[48] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1938–1947. IEEE, 2022.

[49] Sparsh Mittal and Shraiysh Vaishay. A survey of techniques for optimizing deep learning on gpus. *Journal of Systems Architecture*, 99:101635, 2019.

[50] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. Map4: A pragmatic framework for in-network machine learning traffic classification. *IEEE Transactions on Network and Service Management*, 19(4):4176–4188, 2022.

[51] Xiaoquan Zhang, Lin Cui, Fung Po Tso, and Weijia Jia. pheavy: Predicting heavy flows in the programmable data plane. *IEEE Transactions on Network and Service Management*, 18(4):4353–4364, 2021.

[52] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680*, 2019.

[53] Jong-Hyouk Lee and Kamal Singh. Switchtree: in-network computing and traffic analyses with random forests. *Neural Computing and Applications*, pages 1–12, 2020.

[54] Aristide Tanyi-Jong Akem, Michele Gucciardo, and Marco Fiore. Flowrest: Practical flow-level inference in programmable switches with random forests. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2023.

[55] Changgang Zheng and Noa Zilberman. Planter: seeding trees within switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*, pages 12–14, 2021.

[56] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *Pattern Recognition*, 105:107281, 2020.

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

[58] 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*, pages 1099–1114, 2022.

[59] Zhizhen Zhong, Weiyang Wang, Manya Ghobadi, Alexander Sludds, Ryan Hamerly, Liane Bernstein, and Dirk Englund. Ioi: In-network optical inference. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Optical Systems*, pages 18–22, 2021.

[60] Shijie Chen, Yu Zhang, and Qiang Yang. Multi-task learning in natural language processing: An overview. *ACM Computing Surveys*, 56(12):1–32, 2024.

[61] Simon Vandenhende, Stamatios Georgoulis, Wouter Van Gansbeke, Marc Proesmans, Dengxin Dai, and Luc Van Gool. Multi-task learning for dense prediction tasks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(7):3614–3633, 2021.

[62] Keishi Ishihara, Anssi Kanervisto, Jun Miura, and Ville Hautamaki. Multi-task learning with attention for end-to-end autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2902–2911, 2021.

[63] Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

[64] Patrick Poullie, Thomas Bocek, and Burkhard Stiller. A survey of the state-of-the-art in fair multi-resource allocations for data centers. *IEEE Transactions on Network and Service Management*, 15(1):169–183, 2017.

[65] Yinbo Yu, Xing Li, Xue Leng, Libin Song, Kai Bu, Yan Chen, Jianfeng Yang, Liang Zhang, Kang Cheng, and Xin Xiao. Fault management in software-defined networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(1):349–392, 2018.

[66] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[67] You-Chiun Wang and Siang-Yu You. An efficient route management framework for load balance and overhead reduction in sdn-based data center networks. *IEEE Transactions on Network and Service Management*, 15(4):1422–1434, 2018.

[68] Athanasios Liatifis, Panagiotis Sarigiannidis, Vasileios Argyriou, and Thomas Lagkas. Advancing sdn from openflow to p4: A survey. *ACM Computing Surveys*, 55(9):1–37, 2023.

[69] Tomasz Osiński, Halina Tarasiuk, Paul Chaignon, and Mateusz Kossakowski. A runtime-enabled p4 extension to the open vswitch packet processing pipeline. *IEEE Transactions on Network and Service Management*, 18(3):2832–2845, 2021.

[70] Ricardo Parizotto, Bruno Loureiro Coelho, Diego Cardoso Nunes, Israat Haque, and Alberto Schaeffer-Filho. Offloading machine learning to programmable data planes: A systematic survey. *ACM Computing Surveys*, 56(1):1–34, 2023.

[71] Changgang Zheng, Benjamin Rienecker, and Noa Zilberman. Qcmp: Load balancing via in-network reinforcement learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, pages 35–40, 2023.

[72] 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*, pages 25–33, 2019.

[73] 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, 2022.

[74] David Chunhu Li, Muhamad Rizka Maulana, and Li-Der Chou. Nnsplit-søren: Supporting the model implementation of large neural networks in a programmable data plane. *Computer Networks*, 222:109537, 2023.

[75] Matthews Jose, Kahina Lazri, Jérôme François, and Olivier Festor. Stateful inrec: Stateful in-network real number computation with recursive functions. *IEEE Transactions on Network and Service Management*, 20(1):830–845, 2023.

[76] Penglai Cui, Heng Pan, Zhenyu Li, Penghao Zhang, Tianhao Miao, Jianer Zhou, Hongtao Guan, and Gaogang Xie. Enabling in-network floating-point arithmetic for efficient computation offloading. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4918–4934, 2022.

[77] 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*, 54(1):2–21, 2024.

[78] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. Homunculus: Auto-generating efficient data-plane ml pipelines for datacenter networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 329–342, 2023.

[79] Eva Papadogiannaki and Sotiris Ioannidis. A survey on encrypted network traffic analysis applications, techniques, and countermeasures. *ACM Computing Surveys*, 54(6):1–35, 2021.

[80] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008.

[81] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas CK Hui. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials*, 18(4):2991–3029, 2016.

[82] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. Deepmatch: practical deep packet inspection in the data plane using network processors. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 336–350, 2020.

[83] Chang Liu, Longtao He, Gang Xiong, Zigang Cao, and Zhen Li. Fs-net: A flow sequence network for encrypted traffic classification. In *IEEE INFOCOM 2019-IEEE Conference On Computer Communications*, pages 1171–1179. IEEE, 2019.

[84] Tal Shapira and Yuval Shavitt. Flowpic: A generic representation for encrypted traffic classification and applications identification. *IEEE Transactions on Network and Service Management*, 18(2):1218–1232, 2021.

[85] Ting-Li Huoh, Yan Luo, Peilong Li, and Tong Zhang. Flow-based encrypted network traffic classification with graph neural networks. *IEEE Transactions on Network and Service Management*, 20(2):1224–1237, 2022.

[86] Yipeng Wang, Huijie He, Yingxu Lai, and Alex X Liu. A two-phase approach to fast and accurate classification of encrypted traffic. *IEEE/ACM Transactions on Networking*, 31(3):1071–1086, 2022.

[87] Xiaochun Yun, Yipeng Wang, Yongzheng Zhang, Chen Zhao, and Zijian Zhao. Encrypted tls traffic classification on cloud platforms. *IEEE/ACM Transactions on Networking*, 31(1):164–177, 2023.

[88] Roberto Doriguzzi-Corin, Stuart Millar, Sandra Scott-Hayward, Jesus Martinez-del Rincon, and Domenico Siracusa. Lucid: A practical, lightweight deep learning solution for ddos attack detection. *IEEE Transactions on Network and Service Management*, 17(2):876–889, 2020.

[89] Ravi Vinayakumar, Mamoun Alazab, KP Soman, Prabaharan Poornachandran, Ameer Al-Nemrat, and Sitalakshmi Venkatraman. Deep learning approach for intelligent intrusion detection system. *IEEE Access*, 7:41525–41550, 2019.

[90] Li Yang and Abdallah Shami. A transfer learning and optimized cnn based intrusion detection system for internet of vehicles. In *ICC 2022-IEEE International Conference on Communications*, pages 2774–2779. IEEE, 2022.

[91] Stefano Longari, Daniel Humberto Nova Valcarcel, Mattia Zago, Michele Carminati, and Stefano Zanero. Cannolo: An anomaly detection system based on lstm autoencoders for controller area network. *IEEE Transactions on Network and Service Management*, 18(2):1913–1924, 2020.

[92] Iman Akbari, Mohammad A Salahuddin, Leni Ven, Noura Limam, Raouf Boutaba, Bertrand Mathieu, Stephanie Moteau, and Stephane Tuffin. A look behind the curtain: Traffic classification in an increasingly encrypted web. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(1):1–26, 2021.

[93] Sahil Gupta, Devashish Gosain, Minseok Kwon, and Hrishikesh B Acharya. Deep4r: Deep packet inspection in p4 using packet recirculation. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[94] Lizhuang Tan, Wei Su, Wei Zhang, Jianhui Lv, Zhenyi Zhang, Jingying Miao, Xiaoxi Liu, and Na Li. In-band network telemetry: A survey. *Computer Networks*, 186:107763, 2021.

[95] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 662–680, 2020.

[96] Shaofei Tang, Deyun Li, Bin Niu, Jianquan Peng, and Zuqing Zhu. Sel-int: A runtime-programmable selective in-band network telemetry system. *IEEE Transactions on Network and Service Management*, 17(2):708–721, 2019.

[97] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*, pages 291–326. Chapman and Hall/CRC, 2022.

[98] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[99] Jinzhu Yan, Haotian Xu, Zhuotao Liu, Qi Li, Ke Xu, Mingwei Xu, and Jianping Wu. Brain-on-switch: Towards advanced intelligent network data plane via nn-driven traffic analysis at line-speed. *arXiv preprint arXiv:2403.11090*, 2024.

[100] Mina Ghashami, Edo Liberty, Jeff M Phillips, and David P Woodruff. Frequent directions: Simple and deterministic matrix sketching. *SIAM Journal on Computing*, 45(5):1762–1792, 2016.

[101] Kamran Razavi, George Karlos, Vinod Nigade, Max Mühlhäuser, and Lin Wang. Distributed dnn serving in the network data plane. In *Proceedings of the 5th International Workshop on P4 in Europe*, pages 67–70, 2022.

[102] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

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

[104] Yongkweon Jeon, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Jeongin Yun, and Dongsoo Lee. Biqgemm: matrix multiplication with lookup table for binary-coding-based quantized dnns. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[105] Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. Analytical guarantees on numerical precision of deep neural networks. In *International Conference on Machine Learning*, pages 3007–3016. PMLR, 2017.

[106] Christos Caraiscos and Bede Liu. A roundoff error analysis of the lms adaptive algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(1):34–41, 1984.

[107] Jordan L Holt and J-N Hwang. Finite precision error analysis of neural network hardware implementations. *IEEE Transactions on Computers*, 42(3):281–290, 1993.

[108] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp*, 1:108–116, 2018.

[109] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. Characterization of encrypted and vpn traffic using time-related. In *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP)*, pages 407–414, 2016.

[110] Ons Aouedi, Kandaraj Piamrat, and Benoît Parrein. Ensemble-based deep learning model for network traffic classification. *IEEE Transactions on Network and Service Management*, 19(4):4124–4135, 2022.

[111] Mario Di Mauro, Giovanni Galatro, and Antonio Liotta. Experimental review of neural-based approaches for network intrusion management. *IEEE Transactions on Network and Service Management*, 17(4):2480–2495, 2020.

[112] Giuseppe Siracusano and Roberto Bifulco. In-network neural networks. *arXiv preprint arXiv:1801.05731*, 2018.

[113] Abdunasser Alowa and Thomas Fevens. Towards minimum inter-controller delay time in software defined networking. *Procedia Computer Science*, 175:395–402, 2020.

[114] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 651–665, 2022.

[115] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 635–649, 2022.

[116] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

[117] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, 212:103561, 2023.

[118] Jan Lansky, Saqib Ali, Mokhtar Mohammadi, Mohammed Kamal Majeed, Sarkhel H Taher Karim, Shima Rashidi, Mehdi Hosseinzadeh, and Amir Masoud Rahmani. Deep learning-based intrusion detection systems: a systematic review. *IEEE Access*, 9:101574–101599, 2021.

[119] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International conference on machine learning*, pages 1321–1330. PMLR, 2017.

[120] Anna L Buczak and Erhan Guven. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys & Tutorials*, 18(2):1153–1176, 2015.

[121] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. Malware traffic classification using convolutional neural network for representation learning. In *2017 International conference on information networking (ICOIN)*, pages 712–717. IEEE, 2017.

[122] Bianca Zadrozny and Charles Elkan. Obtaining calibrated probability estimates from decision trees and naive bayesian classifiers. In *Icml*, volume 1, pages 609–616, 2001.

[123] Mingyuan Zang, Changgang Zheng, Lars Dittmann, and Noa Zilberman. Toward continuous threat defense: in-network traffic analysis for iot gateways. *IEEE Internet of Things Journal*, 11(6):9244–9257, 2023.

[124] Masoud Hemmatpour, Changgang Zheng, and Noa Zilberman. E-commerce bot traffic: In-network impact, detection, and mitigation. In *2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, pages 179–185. IEEE, 2024.

[125] Jorge Luis Guerra, Carlos Catania, and Eduardo Veas. Datasets are not enough: Challenges in labeling network traffic. *Computers & Security*, 120:102810, 2022.

[126] 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*, pages 20–25, 2018.

[127] Seongyeon Yoon, Heewon Kim, Hyeonjae Jeong, Chanbin Bae, Haeun Kim, and Sangheon Pack. Multi-task aware resource efficient traffic classification via in-network inference. In *Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing*, pages 69–74, 2024.

[128] Anurag Agrawal and Changhoon Kim. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–32. IEEE Computer Society, 2020.

[129] Thaha Mohammed, Carlee Joe-Wong, Rohit Babbar, and Mario Di Francesco. Distributed inference acceleration with adaptive dnn partitioning and offloading. In

IEEE INFOCOM 2020-IEEE Conference on Computer Communications, pages 854–863. IEEE, 2020.

[130] Lorenzo Bracciale, Tushar Swamy, Muhammad Shahbaz, Pierpaolo Loreti, Stefano Salsano, and Hesham Elbakoury. The case for native multi-node in-network machine learning. In *Proceedings of the 1st International Workshop on Native Network Intelligence*, pages 8–13, 2022.

[131] Christoph Hardegen, Benedikt Pfülb, Sebastian Rieger, and Alexander Gepperth. Predicting network flow characteristics using deep learning and real-world network traffic. *IEEE Transactions on Network and Service Management*, 17(4):2662–2676, 2020.

[132] Changgang Zheng, Haoyue Tang, Mingyuan Zang, Xinpeng Hong, Aosong Feng, Leandros Tassiulas, and Noa Zilberman. DINC: Toward distributed in-network computing. *Proceedings of the ACM on Networking*, 1(CoNEXT3):1–25, 2023.

[133] Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.

[134] Ibm ilog cplex optimization studio. https://www.ibm.com/products/ilog-cplex-optimization-studio. Accessed: 2025-02-04.

[135] Francisco Germano Vogt, Fabricio Eduardo Rodriguez Cesen, Ariel Góes De Castro, Suneet Kumar Singh, Marcelo Caggiani Luizelli, Christian Esteve Rothenberg, and Gianni Antichi. Video streaming qoe meets programmable data planes: The case of in-network qoe for 360° vr. *IEEE Network*, 39(2):176–183, 2024.

[136] Rupert G Miller Jr. *Beyond ANOVA: basics of applied statistics*. CRC press, 1997.

[137] Shahbaz Rezaei and Xin Liu. How to achieve high classification accuracy with just a few labels: A semi-supervised approach using sampled packets. *arXiv preprint arXiv:1812.09761*, 2018.

[138] Andreas Maurer. Bounds for linear multi-task learning. *The Journal of Machine Learning Research*, 7:117–139, 2006.

[139] Shai Ben-David and Reba Schuller. Exploiting task relatedness for multiple task learning. In *Learning Theory and Kernel Machines: 16th Annual Conference on Learning Theory and 7th Kernel Workshop, COLT/Kernel 2003, Washington, DC, USA, August 24-27, 2003. Proceedings*, pages 567–580. Springer, 2003.

[140] Yu Zhang. Multi-task learning and algorithmic stability. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2015.

[141] Intel tofino 2 12.8 tbps, 20 stage, 4 pipelines. https://www.intel.com/content/www/us/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/specifications.html. Accessed: 2025-01-24.

[142] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.

[143] Ethernet Alliance and Blaine Kohl. Ethernet jumbo frames, 2009.

[144] Chris Fifty, Ehsan Amid, Zhe Zhao, Tianhe Yu, Rohan Anil, and Chelsea Finn. Efficiently identifying task groupings for multi-task learning. *Advances in Neural Information Processing Systems*, 34:27503–27516, 2021.

[145] Xinpeng Hong, Changgang Zheng, Stefan Zohren, and Noa Zilberman. LOBIN: In-network Machine Learning for Limit Order Books. In *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*, pages 159–166. IEEE, 2023.

[146] Xsight labs x2. https://xsightlabs.com/. Accessed: 2025-05-02.

[147] Cisco silicon one. https://www.cisco.com/site/us/en/products/networking/silicon-one/index.html. Accessed: 2025-05-02.

[148] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. {CADE}: Detecting and explaining concept drift samples for security applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2327–2344, 2021.

[149] Giuseppina Andresini, Feargus Pendlebury, Fabio Pierazzi, Corrado Loglisci, Annalisa Appice, and Lorenzo Cavallaro. Insomnia: Towards concept-drift robustness in network intrusion detection. In *Proceedings of the 14th ACM workshop on artificial intelligence and security*, pages 111–122, 2021.

[150] Bruno Missi Xavier, Magnos Martinello, Celio Trois, Brenno M Alenca, and Ricardo A Rios. Fast learning enabled by in-network drift detection. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, pages 129–134, 2024.

[151] Qizheng Zhang, Ali Imran, Enkeleda Bardhi, Tushar Swamy, Nathan Zhang, Muhammad Shahbaz, and Kunle Olukotun. Caravan: Practical online learning of {In-Network}{ML} models with labeling agents. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 325–345, 2024.

[152] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, 2018.

[153] Igor Goldenberg and Geoffrey I Webb. Survey of distance measures for quantifying concept drift and shift in numeric data. *Knowledge and Information Systems*, 60(2):591–615, 2019.

[154] Weiming Feng, Liqiang Liu, and Tianren Liu. On deterministically approximating total variation distance. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1766–1791. SIAM, 2024.

[155] 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. IEEE, 2015.

[156] Jielun Zhang, Fuhao Li, Feng Ye, and Hongyu Wu. Autonomous unknown-application filtering and labeling for dl-based traffic classifier update. In *IEEE INFOCOM 2020-IEEE conference on computer communications*, pages 397–405. IEEE, 2020.

[157] Dan Hendrycks and Kevin Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *ICLR*, 2017.

[158] Heinrich Jiang, Been Kim, Melody Guan, and Maya Gupta. To trust or not to trust a classifier. *Advances in neural information processing systems*, 31, 2018.

[159] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *IEEE computer society conference on computer vision and pattern recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

[160] Elad Hoffer and Nir Ailon. Deep metric learning using triplet network. In *International workshop on similarity-based pattern recognition*, pages 84–92. Springer, 2015.

[161] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[162] Yandong Wen, Kaipeng Zhang, Zhifeng Li, and Yu Qiao. A discriminative feature learning approach for deep face recognition. In *European conference on computer vision*, pages 499–515. Springer, 2016.

[163] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying iot devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2018.

[164] Yutong Chen, Zhen Li, Junzheng Shi, Gaopeng Gou, Chang Liu, and Gang Xiong. Not afraid of the unseen: a siamese network based scheme for unknown traffic discovery. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7. IEEE, 2020.

[165] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. T4p4s: A target-independent compiler for protocol-independent packet processors. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[166] Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io//. Accessed: 2025-07-27.

[167] Giulio Sidoretti, Lorenzo Bracciale, Stefano Salsano, Hesham Elbakoury, and Pierpaolo Loreti. Dida: Distributed in-network intelligent data plane for machine learning applications. *IEEE Transactions on Network and Service Management*, 22(3):2564–2579, 2025.