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

Kaiyi Zhang, *Member, IEEE*, Changgang Zheng, Nancy Samaan, *Member, IEEE*, Ahmed Karmouch, *Life Member, IEEE*, and Noa Zilberman, *Senior Member, IEEE*

***Abstract*—The increasing demand for real-time inference on high-volume network traffic has led to the rise of in-network machine learning, where programmable switches execute various models directly in the data-plane at line rate. Effective network management often involves multiple prediction tasks, such as predicting bit rate, flow size, or traffic class; however, existing solutions deploy separate models for each task, placing a significant burden on the data-plane and leading to substantial resource consumption when deploying multiple tasks. To address this limitation, we introduce MUTA, a novel in-network multi-task learning framework that enables concurrent inference of multiple tasks in the data-plane, without exhausting available resources. MUTA builds a multi-task neural network to share feature representations across tasks and introduces a data-plane mapping methodology to fit it within network switches. Additionally, MUTA enhances scalability by supporting distributed deployment, where different layers of a multi-task model can be offloaded across multiple switches. An orchestrator employs multi-objective optimization to determine optimal model placement in multi-path networks. MUTA is deployed on P4 hardware switches, and is shown to reduce memory requirements by $\times 10.5$, while at the same time improving accuracy by up to 9.14% using limited training data, compared with state-of-the-art single-task learning solutions.**

***Index Terms*—In-network computing; P4; multi-task learning; neural networks; programmable data-planes**

I. Introduction

AI-assisted network management schemes traditionally deploy learning models on either end-hosts or network control-planes [1]. However, this approach often results in long reaction times to network events and may incur substantial bandwidth consumption [2]. Programmable switches, along with domain-specific languages such as P4 [3], present a unique opportunity to execute machine learning (ML) inference algorithms directly within the data-plane. This capability facilitates learning-based network management analysis at line-rate, ensuring ultra-low latency response times without impacting network forwarding [4].

Prior in-network ML research has introduced numerous implementations of ML models in the data-plane. Tree-based models [5–12], in particular, are popular as their inference process can be effectively implemented using match-action tables. These implementations can carry out network management tasks such as attack detection [13], traffic classification [14], bot traffic detection [15], and heavy flow detection [16]. Additionally, some studies have investigated deploying neural networks on SmartNICs or network switches for traffic analysis [17–22] or leveraging these devices as accelerators to enhance neural network inference efficiency [23, 24].

However, the aforementioned 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 [25]. However, due to the limited resources of network switches, deploying individual models for each task can exhaust or even exceed all the switch resources [26]. 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, expensive, and difficult to scale, especially as network data volumes continue to grow. While automatic labeling schemes [27] 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) [28] 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

Kaiyi Zhang, Nancy Samaan and Ahmed Karmouch are with the School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, ON K1N 6N5, Canada (e-mail:kzhan122@uottawa.ca)

Changgang Zheng and Noa Zilberman are with the Department of Engineering Science, University of Oxford, OX1 3PJ Oxford, U.K.

tasks [28]. However, there are two challenges to implementing neural network 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 [29, 30], 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 [17, 18]; however, they suffer from precision degradation.

To address these challenges, this paper 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 (§VII-C). 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 (§VII-B) 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 modifications (§V). 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 (§VI). In summary, the main contributions of this paper are as follows:

- We introduce MUTA[1], 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 through 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 article is organized as follows; Section II discusses related work. Section III provides an overview of the proposed architecture while Section IV describes the proposed multi-task neural network and explains the adopted quantization scheme. Section V discusses the P4-based implementation details. Performance evaluation results are discussed in Section VII. Section VIII discusses some considerations and future research directions. Finally, Section IX concludes the paper.

II. Related Work

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

A. Programmable Data-Plane

The Protocol-Independent Switch Architecture (PISA) [35] enables data-plane programmability, empowering fast innovation of networking designs. Many current data-plane architectures [36–39] originate from, and are similar to, this general architecture. In a PISA pipeline, a data packet is first mapped into a packet header vector (PHV) by a parser. The PHV is then passed to a match-action pipeline for algorithm execution and data manipulation. The pipeline consists of match-action tables arranged in a sequence of logical stages. Match-action tables are fundamental units that lookup a value of key (e.g., a field in packet header) in a table, and map the resulting entry to a corresponding action. Finally, the processed PHV is assembled into a set of ordered headers and payload by the deparser. 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 operations, complex instructions like floating-point operation, matrix multiplication and loops are not supported. Furthermore, hardware switches are resource-constrained, with only tens of megabytes of memory and a restricted number of processing stages [4]. For example, Intel Tofino switch [40] has twelve processing stages and Mb-scale memory.

[1]A preliminary work appeared in part at the 2025 IEEE International Conference on High Performance Switching and Routing (HPSR) [31].

TABLE I: Comparison of advanced in-network neural network solutions.

Scheme	Model	Platform	Layer Split	Multi-path Distributed	Without Recirculation	Multi-task
N2Net [32]	binarized NN	RMT-like Switch	✘	✘	✓	✘
BaNaNa Split [23]	binarized NN	SmartNIC	✓	✘	✓	✘
N3IC [17]	binarized NN	SmartNIC	✘	✘	✓	✘
Qin et al.[18]	binarized NN	bmv2 (software)	✘	✘	✓	✘
NNSplit[33]	binarized NN	bmv2 (software)	✓	✘	✓	✘
MARTINI[34]	binarized NN	bmv2 (software)	✘	✘	✓	✓
BoS [22]	binarized RNN	Tofino (hardware)	✘	✘	✓	✘
INQ-MLT [19]	non-binarized NN	bmv2 (software)	✘	✘	✓	✘
IOI [30]	non-binarized NN	Modified ASIC	✘	✘	✓	✘
Taurus [29]	non-binarized NN	Modified ASIC	✘	✘	✓	✘
Razavi et al. [24]	non-binarized CNN	Tofino (hardware)	✓	✘	✘	✘
MUTA	non-binarized NN	Tofino (hardware)	✓	✓	✓	✓

B. 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 [41] and computer vision [42] and autonomous driving [43].

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 [28]. 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.

C. Existing In-Network ML Solutions and Limitations

1) In-Network Tree-based Solutions: The research community has made substantial progress [5–12] in realizing tree-based inference models within programmable switches. Among the proposed methodologies, two advanced mapping schemes have been extensively explored: the hierarchical mapping scheme and the feature-encoding mapping scheme. Hierarchical mapping schemes [7–9] follows a natural strategy, which involves mapping the hierarchical structure of 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 feature-encoding mapping scheme [26] 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. Beyond single-switch deployment, DUNE [44] further extends scalability by distributing tree-based models across multiple switches.

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.

2) In-Network Neural Network Solutions: Table I 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 [17] and BaNaNa Split [23]), and software switches bmv2 (e.g., Qin et al. [18]). 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[17, 18]. Following this approach, MARTINI [34] 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 [22] 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 [19, 20] 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 [36]), and not for switch ASICs. Razavi et al. [24] implement a quantized convolutional neural network (CNN) on Tofino2 switches [45] 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 [29] proposes modified switches, using custom hardware based on the MapReduce abstraction, supporting deep neural networks. Similarly, IOI [30] 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 [25]. 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 [34] 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.

D. Design Challenges

Model Inference on unmodified switch ASIC: Programmable switch ASICs, such as Intel Tofino [40], 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 [29, 30] or extensive recirculation [24]. 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 ceiling, 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 [46]. While the idea of distributing a neural network's layers in the data-plane is not new [33], two significant gaps remain: first, distribution across resource-constrained switch-ASICs is significantly different from 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 must address the challenges of resource constraints, scalability, and deployment across programmable switches while ensuring high accuracy and maintaining line-rate performance.

III. An overview of MUTA

This section provides an overview of the proposed architecture. MUTA combines control-plane and data-plane components. As shown in Fig. 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. 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 (§IV).

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

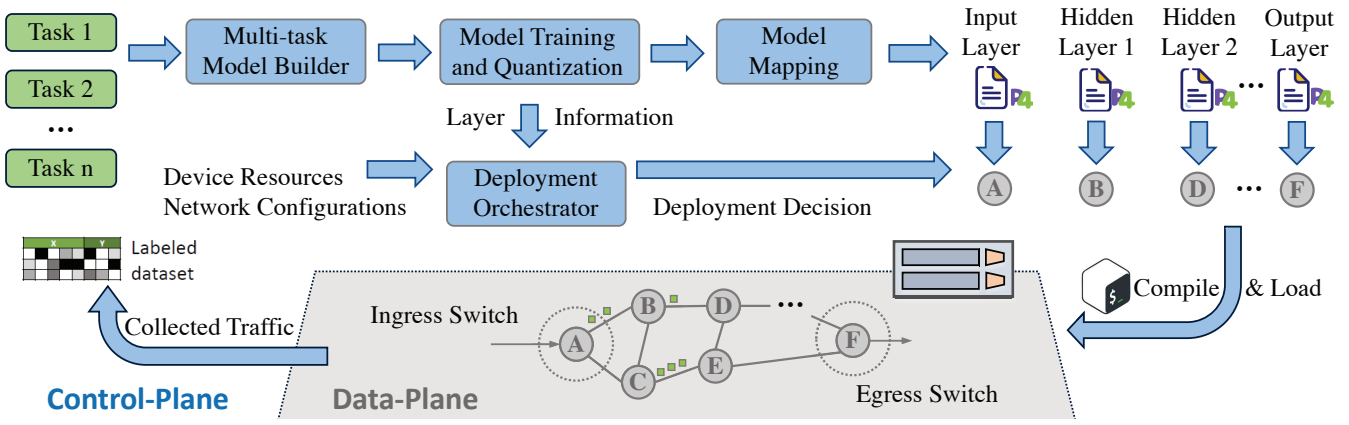


Fig. 1: MUTA architecture. The control-plane is responsible for training and quantizing the model, generating the data-plane code, and determining the deployment strategy. The model layers are then distributed across multiple data-plane devices to cover intelligent services across the entire network.

implemented as a standalone P4 program, and we do not prescribe a specific feature extraction mechanism. Extracted features can then be transmitted in-band together with the data packets [47]. 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 (§V) 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 topologies 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 (§VI).

IV. 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) [48], 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

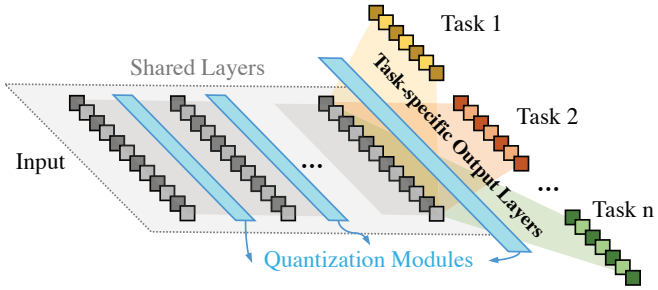


Fig. 2: Proposed multi-task learning architecture.

generalized learning across tasks. This shared representation not only enhances the model's ability to generalize [28] but also reduces the computational overhead associated with training separate models for each task.

A. Model Architecture and Training

The overall architecture of our multi-task model is shown in Fig. 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:

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

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.

B. 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. 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) [19]. 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,

$$q = \text{clamp}(\left\lfloor \frac{r}{S} \right\rceil + Z; Q_{min}, Q_{max}) \tag{2}$$

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) [49]. As depicted in Fig. 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 its corresponding match-action tables as part of the packet forwarding pipeline, as detailed in Section V.

V. 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. Fig. 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.

A. Data-Plane Mapping Methodology

1) 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.

```
apply {
    action ac_input1(int<32>z11, int<32>z21,...,int<32>z81){
        meta.R1_h1 = meta.R1_h1 + z11;
        meta.R1_h2 = meta.R1_h2 + z21;
        ...
        meta.R1_h8 = meta.R1_h8 + z81;}
    action ac_input2(int<32>z12, int<32>z22,...,int<32>z82){
        meta.R1_h1 = meta.R1_h1 + z12;
        meta.R1_h2 = meta.R1_h2 + z22;
        ...
        meta.R1_h8 = meta.R1_h8 + z82;}
    action ac_input3(int<32>z13, int<32>z23,...,int<32>z83){
        meta.R1_h1 = meta.R1_h1 + z13;
        meta.R1_h2 = meta.R1_h2 + z23;
        ...
        meta.R1_h8 = meta.R1_h8 + z83;}
    action ac_input4(int<32>z14, int<32>z24,...,int<32>z84){
        meta.R2_h1 = meta.R2_h1 + z14;
        meta.R2_h2 = meta.R2_h2 + z24;
        ...
        meta.R2_h8 = meta.R2_h8 + z84;}
    action ac_input5(int<32>z15, int<32>z25,...,int<32>z85){
        meta.R2_h1 = meta.R2_h1 + z15;
        meta.R2_h2 = meta.R2_h2 + z25;
        ...
        meta.R2_h8 = meta.R2_h8 + z85;}
    action ac_input6(int<32>z16, int<32>z26,...,int<32>z86){
        meta.R2_h1 = meta.R2_h1 + z16;
        meta.R2_h2 = meta.R2_h2 + z26;
        ...
        meta.R2_h8 = meta.R2_h8 + z86;}
    table tb_input1 {
        key={hdr.input1:exact;}
        actions = {ac_input1;}
        size=256;} // Stage 0
    table tb_input4 {
        key={hdr.input4:exact;}
        actions = {ac_input4;}
        size=256;} // Stage 0
    table tb_input2 {...} // Stage 1
    table tb_input5 {...} // Stage 1
    table tb_input3 {...} // Stage 2
    table tb_input6 {...} // Stage 2
    // Stage 3
    meta.output1 =  meta.R1_h1 + meta.R2_h1
    meta.output2 =  meta.R1_h2 + meta.R2_h2
    ...
    meta.output8 =  meta.R1_h8 + meta.R2_h8
}
```

Listing 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_input2 correspond to the precomputed outputs $(x_2w_{12}, x_2w_{22}, \cdots, x_2w_{82})$ in Table 2 of Fig. 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 $\mathbf{y= xW + b}$. 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 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

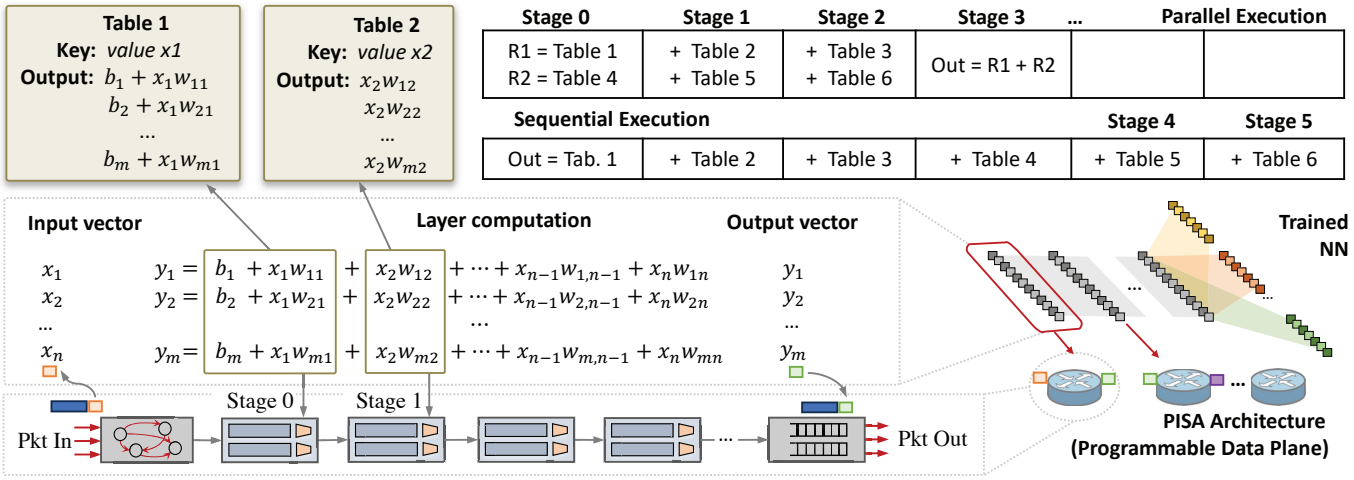


Fig. 3: Methodology for mapping layer computation to a match-action pipeline. The parser extracts the input vector from the packet header, followed by layer-wise inference executed through a sequence of match-action tables. The deparser then reconstructs the packet, embedding the output vector into the 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.

operations. For example, the parameters $(z_{12}, z_{22}, ..., z_{82})$ in the action *ac_input*2 (as shown in Listing 1) correspond to the precomputed outputs $(x_2w_{12}, x_2w_{22}, \cdots, x_2w_{82})$ in Table 2 of Fig. 3. The addition of bias can be integrated into any one of these tables, such as Table1 in Fig.3.

The looked-up intermediate values are then used for addition operations that generates 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. (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 $[0, 255]$) [49], 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.

2) 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) [22].

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.

B. 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 Fig. 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 stages (line 38-49 in Listing 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 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 VII-C3. 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 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 VI.

VI. 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.

A. Model Formulation

Following [50], 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.

1) 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 $s_p^1 \rightarrow \cdots \rightarrow s_p^{l_p}$ 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.

2) Resource model: Let $\mathcal{R} := \{\lambda_1, \cdots, \lambda_{|R|}\}$ be the set of resource types 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 $s \in \mathcal{S}$.

3) 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.

4) Deployment Decision: Let $X_{k\rightarrow s} \in \{0, 1\}, \forall k \in \mathcal{K}, s \in \mathcal{S}$ be the deployment decision, where $X_{k\rightarrow s} = 1$ indicates layer k is deployed on switch s. If $X_{k\rightarrow s} = 1$, by executing layer k, switch s will use O_k^λ units of resource type $\lambda \in \mathcal{R}$.

The goal is to design a deployment strategy, i.e., $\{X_{k\rightarrow s}\}$, 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 O_k^λ, 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.

B. Constraints

1) 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 $X_{k+1\rightarrow s_p^e} = 1$ and layer k has to be deployed on at least one node in set $\{s_p^1, \cdots, s_p^{e-1}\}$, i.e.,

$$\sum_{u=1}^{e-1} X_{k\rightarrow s_p^u} \geq X_{k+1\rightarrow s_p^e}, \forall p \in \mathcal{P}, \forall e \in \{2, \cdots, l_p\} \quad 1 \leq k \leq |\mathcal{K}| - 1 \tag{3}$$

2) 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.,

$$\sum_{u=1}^{l_p} X_{k\rightarrow s_p^u} \geq 1, \forall p \in \mathcal{P} \tag{4}$$

3) Resource Constraints: The available resources on each switch s must be sufficient for all deployed layers. Therefore,

$$\sum_{k=1}^{|\mathcal{K}|} O_k^\lambda X_{k\rightarrow s} \leq Q_s^\lambda, \forall s \in \mathcal{S}, \forall \lambda \in \mathcal{R} \tag{5}$$

C. Problem Formulation

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

$$\Psi_{C,\lambda} = \sum_{s=1}^{|\mathcal{S}|} \sum_{k=1}^{|\mathcal{K}|} O_k^\lambda X_{k\to s} \tag{6}$$

2) 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:

$$\Psi_{L,p} = \sum_{v=1}^{l_p} v X_{|\mathcal{K}|\to s_p^v} H\left(1 - \sum_{u=1}^{v} X_{|\mathcal{K}|\to s_p^u}\right) \tag{7}$$

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 (7) as below:

$$\Psi_{L,p} = \sum_{v=1}^{l_p} v \cdot Z_{p,v} \tag{8}$$

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

$$Z_{p,v} \leq X_{|\mathcal{K}|\to s_p^v}, \forall p \in \mathcal{P}, \forall v \in \{1,\cdots,l_p\} \tag{9}$$

$$\sum_{u=1}^{v-1} X_{|\mathcal{K}|\to s_p^u} + Z_{p,v} \leq 1, \forall p \in \mathcal{P}, \forall v \in \{2,\cdots,l_p\} \tag{10}$$

$$\sum_{v=1}^{l_p} Z_{p,v} = 1, \forall p \in \mathcal{P} \tag{11}$$

Constraint (9) ensures $Z_{p,v}$ can only be 1 if the final layer $|\mathcal{K}|$ is deployed on s_p^v. Constraint (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 (11) ensures that the output layer $|\mathcal{K}|$ is executed at exactly one position along each path.

3) 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:

$$\begin{aligned} \min \quad & w_C \sum_{\lambda=1}^{|\mathcal{R}|} \Psi_{C,\lambda} + w_L \sum_{p=1}^{|\mathcal{P}|} \Psi_{L,p} \\ \text{s.t.} \quad & (3),(4),(5),(9),(10),(11) \end{aligned} \tag{12}$$

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.

4) Solution: The problem described above falls under the category of standard Integer Linear Programming (ILP). Several well-established ILP solvers, such as HiGHS [51] and CPLEX [52], 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.

VII. 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 [53] and traffic characteristics prediction [25] as the use-case scenarios for validating the performance of our proposed architecture.

A. 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 [54]. We use the dataset provided by [53] 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 labeled 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 69 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 [55] 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 [56] 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 [25]. 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.

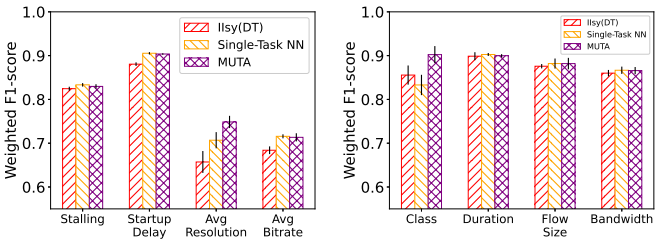
B. Multi-Task Model Performance

1) 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 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 [2]. 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 [17, 26, 29]. All results are reported on the test set, and the performance is checked using 5-fold cross-validation.

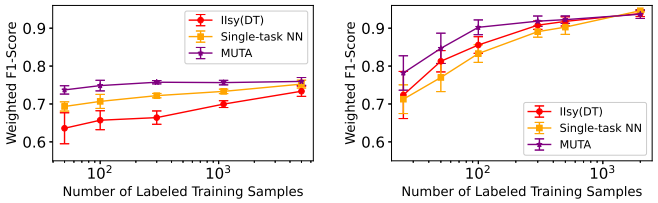
2) Results: As illustrated in Fig. 4, MUTA outperforms decision trees (DTs) and single-task NNs for *hard-to-label* tasks in both use-cases, where only 100 labeled samples are available for training. For instance, in the resolution prediction task, MUTA improves the F1-score by 4.17% compared to single-task 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 because there are abundant training

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



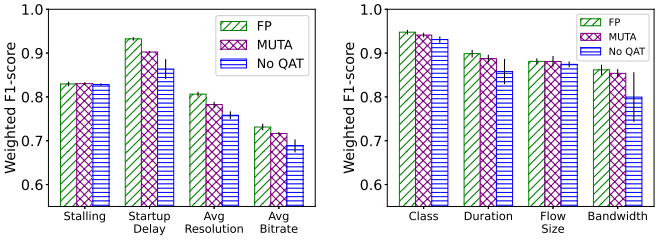
(a) QoE prediction (b) Traffic characteristics prediction

Fig. 4: Performance comparison between IIsy (DT) [26], single-task neural network (NN), and MUTA, using only 100 samples for label-limited tasks (resolution prediction and traffic class prediction) during training.



(a) Resolution prediction task in QoE prediction (b) Traffic class prediction task in traffic characteristics prediction

Fig. 5: Performance comparison for label-limited tasks across different numbers of labeled training samples.

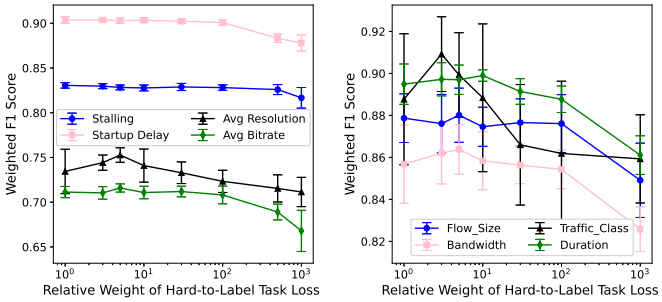


(a) QoE prediction (b) Traffic characteristics prediction

Fig. 6: Performance comparison of the floating-point model (FP), quantized model without QAT (No QAT), and MUTA.

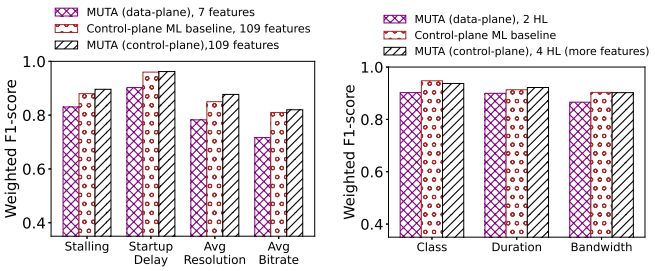
data for these tasks. Single-task models tend to perform poorly when training data is limited, as insufficient supervision increases the risk of underfitting and reduces the model's ability to generalize to unseen instances. This result demonstrates that MUTA can improve the performance of *hard-to-label*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

Fig. 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 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



(a) QoE prediction (b) Traffic characteristics prediction

Fig. 7: Impact of *hard-to-label* task loss weight on model performance.



(a) QoE prediction (b) Traffic characteristics prediction

Fig. 8: Performance comparison between MUTA and control-plane ML schemes. (a) The control-plane ML baseline adopts the QoE prediction method presented in [53]. (b) The control-plane ML baseline follows the MTL scheme proposed in [25]. HL: Hidden Layer.

conditions under which multi-task models outperform single-task models are discussed in [57], [58] and [59].

Fig. 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.

Fig. 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 Fig. 7, increasing this weight initially enhances the performance of the *hard-to-label* task until a maximum is reached. Further increasing 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: Fig. 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 [53], 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 Fig. 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 [53]. For traffic characteristics prediction, we adopt a representative control-plane MTL scheme proposed by Rezaei and Liu [25], 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 Fig. 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 [25], while using fewer layers.

To further enhance MUTA's performance, a hybrid approach similar to that proposed in [26] 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.

C. Hardware Resource Consumption

1) Setting and Metrics: We implement the model using $P4_{16}$ targeting Tofino Native Architecture (TNA) [37] 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 [26]) (other schemes such as Flowrest [10] and NetBeacon [12] are all derived from or closely related to IIsy) and the hierarchical

TABLE II: 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 III: 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

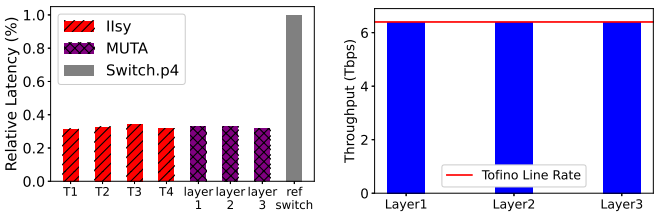
mapping solution (e.g., SwitchTree [8] and pforest [9]). These tree models are generated using Planter [60]. However, every tree model generated using the hierarchical mapping solution failed to fit due to extremely high pipeline-stage consumption. Consequently, we report results only for tree models generated using the feature-encoding solution (i.e., IIsy [26]).

2) Results: Table II presents the resource consumption of IIsy for each individual task as well as the cumulative consumption for all four tasks combined. Similarly, Table III details the resource utilization of MUTA across each layer of the neural network, along with the 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) [61] 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.

3) 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.

D. Latency and Throughput

1) 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



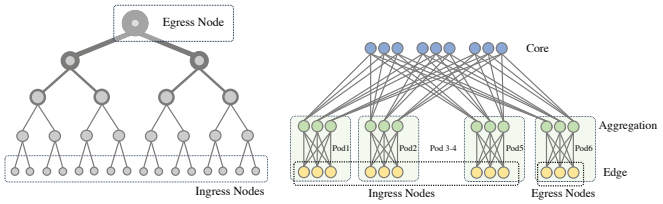
(a) R-Latency. (b) Throughput (Tbps).

Fig. 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.

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.

2) Results: As shown in Fig. 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 Fig. 9 (b), all layers are able to achieve a full line-rate of 6.4Tbps.

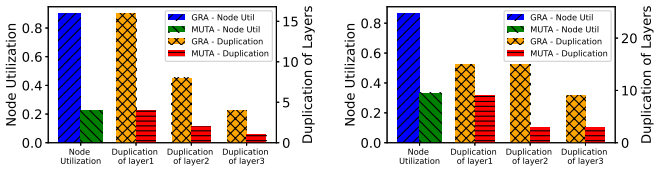
E. Network-Wide Deployment Performance



(a) Tree topology (b) Fat-tree topology

Fig. 10: Network topology used for evaluation.

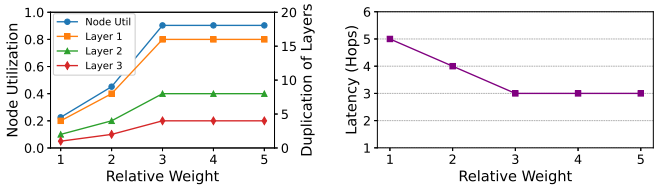
1) Simulation Setting and Metrics: We use the IBM ILOG CPLEX optimization solver [52] to solve the optimization problem in Section VI. The computations were conducted on a PC with Intel Core i7-9750H processor @ 2.60 GHz cpu and 16 GB of RAM. We evaluate the deployment orchestrator using two network topologies, as illustrated in Fig. 10. The first topology, shown in Fig. 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 Fig. 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 [62]. 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



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

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

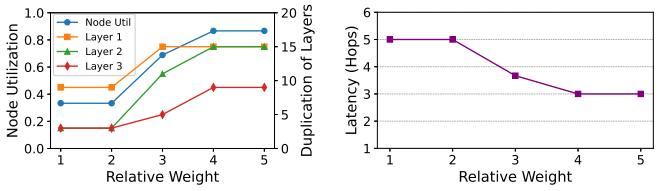
Fig. 11: Network-wide deployment performance between MUTA and the greedy resource availability (GRA) baseline.



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

(b) Latency

Fig. 12: Relative objective weight of w (defined in (12)) influence on MUTA's deployment strategy. Tree topology: Depth 5 with 31 switches.



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

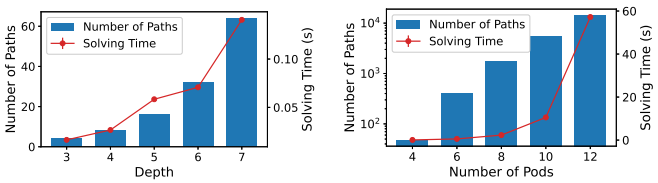
(b) Latency

Fig. 13: Relative objective weight of w (defined in (12)) influence on MUTA's deployment strategy. Fat-tree topology: 6 pods with 45 switches.

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.

2) Results: Fig. 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



(a) Tree topology

(b) Fat-tree topology

Fig. 14: Impact of topology scale on solver execution time.

(from 9 duplications to 3). These reductions saves the memory and stage resources.

Figs. 12 and 13 illustrate the tradeoff between the orchestrator's impact on latency 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.

Fig. 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 [52], which handles ILP formulation introduced in Section VI, remains efficient and well within acceptable limits. Therefore, we do not consider any heuristic method for this optimization problem.

VIII. 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 [13] and drift detection work SPIDD [63].

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 [10], 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 [64] 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 [65] 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 [66] 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 [67] (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 [68] and Cisco Silicon One [69] 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 [60]). 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.

IX. Conclusion

In this paper, 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.

X. Acknowledgments

This research was supported in part by EU Horizon SmartEdge (101092908, Innovate UK 10056403), VMWare, and the Natural Sciences and Engineering Research Council of Canada (NSERC). For the purpose of Open Access, the author has applied a CC BY public copyright license to any Author Accepted Manuscript (AAM) version arising from this submission.

REFERENCES

[1] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo. "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities". In: *Journal of Internet Services and Applications* 9.1 (2018), pp. 1–99.

[2] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu. "A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges". In: *IEEE Communications Surveys & Tutorials* 21.1 (2018), pp. 393–430.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[4] C. Zheng, X. Hong, D. Ding, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman. "In-network machine learning using programmable network devices: A survey". In: *IEEE Communications Surveys & Tutorials* 26.2 (2023), pp. 1171–1200.

[5] Z. Xiong and N. Zilberman. "Do switches dream of machine learning? toward in-network classification". In: *Proceedings of the 18th ACM workshop on hot topics in networks*. 2019, pp. 25–33.

[6] C. Zheng and N. Zilberman. "Planter: seeding trees within switches". In: *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. ACM New York, NY, 2021, pp. 12–14.

[7] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello. "Map4: A pragmatic framework for in-network machine learning traffic classification". In: *IEEE Transactions on Network and Service Management* 19.4 (2022), pp. 4176–4188.

[8] J.-H. Lee and K. Singh. "Switchtree: in-network computing and traffic analyses with random forests". In: *Neural Computing and Applications* (2020), pp. 1–12.

[9] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever. "pforest: In-network inference with random forests". In: *arXiv preprint arXiv:1909.05680* (2019).

[10] A. T.-J. Akem, M. Gucciardo, and M. Fiore. "Flowrest: Practical flow-level inference in programmable switches with random forests". In: *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE. 2023, pp. 1–10.

[11] G. Xie, Q. Li, Y. Dong, G. Duan, Y. Jiang, and J. Duan. "Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation". In: *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE. 2022, pp. 1938–1947.

[12] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu. "An efficient design of intelligent network data plane". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 6203–6220.

[13] M. Zang, C. Zheng, L. Dittmann, and N. Zilberman. "Toward continuous threat defense: in-network traffic analysis for iot gateways". In: *IEEE Internet of Things Journal* 11.6 (2023), pp. 9244–9257.

[14] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello. "Programmable switches for in-networking classification". In: *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE. 2021, pp. 1–10.

[15] M. Hemmatpour, C. Zheng, and N. Zilberman. "E-commerce bot traffic: In-network impact, detection, and mitigation". In: *2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN)*. IEEE. 2024, pp. 179–185.

[16] X. Zhang, L. Cui, F. P. Tso, and W. Jia. "pHeavy: Predicting heavy flows in the programmable data plane". In: *IEEE Transactions on Network and Service Management* 18.4 (2021), pp. 4353–4364.

[17] G. Siracusano, S. Galea, D. Sanvito, M. Malekzadeh, G. Antichi, P. Costa, H. Haddadi, and R. Bifulco. "Re-architecting traffic analysis with neural network interface cards". In: *19th USENIX symposium on networked systems design and implementation (NSDI 22)*. 2022, pp. 513–533.

[18] Q. Qin, K. Poularakis, K. K. Leung, and L. Tassiulas. "Line-speed and scalable intrusion detection at the network edge via federated learning". In: *2020 IFIP Networking Conference (Networking)*. IEEE. 2020, pp. 352–360.

[19] K. Zhang, N. Samaan, and A. Karmouch. "A machine learning-based toolbox for p4 programmable data-planes". In: *IEEE Transactions on Network and Service Management* 21.4 (2024), pp. 4450–4465.

[20] K. Zhang, N. Samaan, and A. Karmouch. "An intelligent data-plane with a quantized ml model for traffic management". In: *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2023, pp. 1–9.

[21] K. Zhang, N. Samaan, and A. Karmouch. "A Two-Stage Confidence-Based Intrusion Detection System in Programmable Data-Planes". In: *GLOBECOM 2023-2023 IEEE Global Communications Conference*. IEEE. 2023, pp. 6850–6855.

[22] J. Yan, H. Xu, Z. Liu, Q. Li, K. Xu, M. Xu, and J. Wu. "Brain-on-Switch: Towards Advanced Intelligent Network Data Plane via NN-Driven Traffic Analysis at Line-Speed". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 419–440.

[23] D. Sanvito, G. Siracusano, and R. Bifulco. "Can the network be the AI accelerator?" In: *Proceedings of the 2018 Morning Workshop on In-Network Computing*. 2018, pp. 20–25.

[24] K. Razavi, G. Karlos, V. Nigade, M. Mühlhäuser, and L. Wang. "Distributed DNN serving in the network data plane". In: *Proceedings of the 5th International Workshop on P4 in Europe*. 2022, pp. 67–70.

[25] S. Rezaei and X. Liu. "Multitask learning for network traffic classification". In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2020, pp. 1–9.

[26] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman. "IIsy: Hybrid in-network classification using programmable switches". In: *IEEE/ACM Transactions on Networking* 32.3 (2024), pp. 2555–2570.

[27] J. L. Guerra, C. Catania, and E. Veas. "Datasets are not enough: Challenges in labeling network traffic". In: *Computers & Security* 120 (2022), p. 102810.

[28] S. Ruder. "An overview of multi-task learning in deep neural networks". In: *arXiv preprint arXiv:1706.05098* (2017).

[29] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. 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*. 2022, pp. 1099–1114.

[30] Z. Zhong, W. Wang, M. Ghobadi, A. Sludds, R. Hamerly, L. Bernstein, and D. Englund. "IOI: In-network optical inference". In: *Proceedings of the ACM SIGCOMM 2021 Workshop on Optical Systems*. 2021, pp. 18–22.

[31] K. Zhang, C. Zheng, N. Samaan, A. Karmouch, and N. 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)*. IEEE. 2025, pp. 1–6.

[32] G. Siracusano and R. Bifulco. "In-network neural networks". In: *arXiv preprint arXiv:1801.05731* (2018).

[33] D. C. Li, M. R. Maulana, and L.-D. Chou. "NNSplit-SØREN: Supporting the model implementation of large neural networks in a programmable data plane". In: *Computer Networks* 222 (2023), p. 109537.

[34] S. Yoon, H. Kim, H. Jeong, C. Bae, H. Kim, and S. Pack. "Multi-task Aware Resource Efficient Traffic Classification via in-Network Inference". In: *Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing*. 2024, pp. 69–74.

[35] N. McKeown. "PISA: Protocol Independent Switch Architecture". In: *P4 Workshop*. 2015, pp. 1–22.

[36] *The Reference P4 Software Switch*. https://github.com/p4lang/behavioral-model. Accessed: 2024-07-22.

[37] *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.

[38] *P4 Portable NIC Architecture (PNA)*. https://p4.org/p4-spec/docs/pna-working-draft-html-version.html. Accessed: 2025-01-21.

[39] *P4-16 Portable Switch Architecture (PSA)*. https://p4.org/p4-spec/docs/PSA.html. Accessed: 2025-01-21.

[40] V. Gurevich and A. Fingerhut. "P4-16 Programming for Intel Tofino Using Intel P4 Studio". In: *2021 P4 Workshop*. 2021.

[41] S. Chen, Y. Zhang, and Q. Yang. "Multi-task learning in natural language processing: An overview". In: *ACM Computing Surveys* (2021).

[42] S. Vandenhende, S. Georgoulis, W. Van Gansbeke, M. Proesmans, D. Dai, and L. Van Gool. "Multi-task learning for dense prediction tasks: A survey". In: *IEEE transactions on pattern analysis and machine intelligence* 44.7 (2021), pp. 3614–3633.

[43] K. Ishihara, A. Kanervisto, J. Miura, and V. 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*. 2021, pp. 2902–2911.

[44] B. Bütün, D. D. A. Hernandez, M. Gucciardo, and M. Fiore. "Dune: Distributed inference in the user plane". In: *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*. IEEE. 2025, pp. 1–10.

[45] A. Agrawal and C. Kim. "Intel tofino2–a 12.9 tbps p4-programmable ethernet switch". In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society. 2020, pp. 1–32.

[46] T. Mohammed, C. Joe-Wong, R. Babbar, and M. Di Francesco. "Distributed inference acceleration with adaptive DNN partitioning and offloading". In: *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE. 2020, pp. 854–863.

[47] L. Bracciale, T. Swamy, M. Shahbaz, P. Loreti, S. Salsano, and H. Elbakoury. "The case for native multi-node in-network machine learning". In: *Proceedings of the 1st International Workshop on Native Network Intelligence*. 2022, pp. 8–13.

[48] C. Hardegen, B. Pfülb, S. Rieger, and A. Gepperth. "Predicting network flow characteristics using deep learning and real-world network traffic". In: *IEEE Transactions on Network and Service Management* 17.4 (2020), pp. 2662–2676.

[49] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. 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*. 2018, pp. 2704–2713.

[50] C. Zheng, H. Tang, M. Zang, X. Hong, A. Feng, L. Tassiulas, and N. Zilberman. "DINC: Toward distributed in-network computing". In: *Proceedings of the ACM on Networking* 1.CoNEXT3 (2023), pp. 1–25.

[51] Q. Huangfu and J. J. Hall. "Parallelizing the dual revised simplex method". In: *Mathematical Programming Computation* 10.1 (2018), pp. 119–142.

[52] *IBM ILOG CPLEX Optimization Studio*. https://www.ibm.com/products/ilog-cplex-optimization-studio. Accessed: 2025-02-04.

[53] M. Seufert and I. Orsolic. "Improving the Transfer of Machine Learning-Based Video QoE Estimation Across Diverse Networks". In: *IEEE Transactions on Network and Service Management* (2023).

[54] F. G. Vogt, F. E. R. Cesen, A. G. De Castro, S. K. Singh, M. C. Luizelli, C. E. Rothenberg, and G. Antichi. "Video Streaming QoE Meets Programmable Data Planes: The Case of In-Network QoE for 360° VR". In: *IEEE Network* (2024).

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

[56] S. Rezaei and X. Liu. "How to achieve high classification accuracy with just a few labels: A semi-supervised approach using sampled packets". In: *arXiv preprint arXiv:1812.09761* (2018).

[57] A. Maurer. "Bounds for linear multi-task learning". In: *The Journal of Machine Learning Research* 7 (2006), pp. 117–139.

[58] S. Ben-David and R. 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*. Springer. 2003, pp. 567–580.

[59] Y. Zhang. "Multi-task learning and algorithmic stability". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 29. 1. 2015.

[60] C. Zheng, M. Zang, X. Hong, L. Perreault, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman. "Planter: Rapid prototyping of in-network machine learning inference". In: *ACM SIGCOMM Computer Communication Review* 54.1 (2024), pp. 2–21.

[61] *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.

[62] M. Al-Fares, A. Loukissas, and A. Vahdat. "A scalable, commodity data center network architecture". In: *ACM SIGCOMM computer communication review* 38.4 (2008), pp. 63–74.

[63] K. Zhang, N. Samaan, A. Karmouch, and L. 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)*. 2025.

[64] E. Alliance and B. Kohl. *Ethernet jumbo frames*. 2009.

[65] C. Fifty, E. Amid, Z. Zhao, T. Yu, R. Anil, and C. Finn. "Efficiently identifying task groupings for multi-task learning". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 27503–27516.

[66] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang. "Pruning and quantization for deep neural network acceleration: A survey". In: *Neurocomputing* 461 (2021), pp. 370–403.

[67] X. Hong, C. Zheng, S. Zohren, and N. Zilberman. "LOBIN: In-network Machine Learning for Limit Order Books". In: *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*. IEEE. 2023, pp. 159–166.

[68] *Xsight Labs X2*. https://xsightlabs.com/. Accessed: 2025-05-02.

[69] *Cisco Silicon One*. https://www.cisco.com/site/us/en/products/networking/silicon-one/index.html. Accessed: 2025-05-02.

Kaiyi Zhang (Member, IEEE) received the B.Eng. degree from the China University of Petroleum, Qingdao, China, in 2018, and the M.Sc. degree in systems science and the Ph.D. degree in electrical and computer engineering from the University of Ottawa, Ottawa, ON, Canada, in 2020 and 2025, respectively. His research interests include software defined networks, programmable data-planes, and machine learning. He is the recipient of the Best Paper Award at the IEEE HPSR 2025.

Changgang Zheng received the DPhil (Ph.D.) degree in Engineering Science from the University of Oxford, Oxford, U.K., and B.Eng. degrees in Electronic and Electrical Engineering (First-Class Honors) from the University of Glasgow, Glasgow, U.K., and in Communication Engineering from the University of Electronic Science and Technology of China, Chengdu, China. His research interests include networking, machine learning, and their intersection.

Nancy Samaan (Member, IEEE) received the B.Sc. and M.Sc. degrees from the Department of Computer Science, Alexandria University, Egypt, and the Ph.D. degree in computer science from the University of Ottawa, Canada, in 2007. She is currently a Professor with the School of Electrical Engineering and Computer Science, University of Ottawa. Her current research interests include software defined networks, programmable data-planes, information centric networks, network resource management and autonomic communications.

Ahmed Karmouch (Life Member, IEEE) received the M.S. and Ph.D. degrees in computer science from the University of Paul Sabatier, Toulouse, France, in 1976 and 1979, respectively. He is a Professor with the School of Electrical Engineering and Computer Science, University of Ottawa. He also held an Industrial Research Chair with the Ottawa Carleton Research Institute and Natural Sciences and Engineering Research Council. He has been the Director of the Ottawa Carleton Institute for Electrical and Computer Engineering. He is involved in several projects with industry and government laboratories in Canada and Europe. His current research interests are in, named data networks, software defined networks, cloud computing, In-network computing, and programmable data-planes. He organized several conferences and workshops; and edited several books. He served as a Guest Editor for IEEE Communications Magazine and Computer Communications.

Noa Zilberman (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from Tel Aviv University, Tel Aviv, Israel. She is currently a Professor with the University of Oxford, Oxford, U.K. Prior to joining Oxford, she was a Fellow and an Affiliated Lecturer with the University of Cambridge, Cambridge, U.K. Her research interests include computing infrastructure, programmable hardware, and networking.