MUTA: Enabling Multi-Task Neural Network Inference in Programmable Data-Planes

Kaiyi Zhang[1], Changgang Zheng[2], Nancy Samaan[1], Ahmed Karmouch[1], Noa Zilberman[2]
[1]*School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Ontario, Canada*
{kzhan122, nsamaan, akarmouc}@uottawa.ca
[2]*Department of Engineering Science, University of Oxford, Oxford, United Kingdom*
{changgang.zheng, noa.zilberman}@eng.ox.ac.uk

***Abstract*—The need for real-time inference of large volumes of data led to the development of in-network machine learning. Programmable network switches can now execute various machine learning models in the data-plane at line rate. While a stream of data may require several prediction tasks, such as predicting bit rate, flow size, or traffic class, current solutions only support separate models for each task. This places a significant burden on the data-plane and leads to substantial resource consumption when deploying multiple tasks. To solve this problem, we introduce MUTA; a novel in-network multi-task learning solution. MUTA enables executing multiple inference tasks concurrently in the data-plane, without exhausting available resources. It introduces a data-plane mapping methodology to fit non-binarized multi-task neural networks within network switches. MUTA is deployed on P4-based hardware switches, and is shown to reduce memory requirements by $\times 10.5$ and improve 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; 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 leads to long reaction times to events and sometimes consumes significant bandwidth [2]. Programmable switches, using domain-specific languages like P4 [3], offer a unique opportunity to run machine learning (ML) inference algorithms directly within the data-plane, enabling learning-based network management analysis at line-rate, without affecting network forwarding and with ultra-low latency response times [4].

Prior in-network ML research has introduced numerous implementations of ML models in the data-plane, such as tree-based models [5, 6] and BNN models [7, 8]. These studies address only a single network management task at a time, using one deployed model. This approach requires the deployment of multiple independent models when multiple management tasks are needed. For example, ensuring Quality of Service (QoS) and optimizing network resource allocation involve numerous management tasks, such as traffic class prediction, bandwidth prediction, and duration prediction [9]. However, given the limited resources of network switches, deploying individual models for each task can exhaust or even exceed all the switch resources [6]. Additionally, some tasks are considered as *hard-to-label* tasks, which refer to tasks where accurately labeling collected traffic data is challenging (e.g., labeling traffic class is labor-intensive). Training models for such tasks with insufficient labeled data often results in poor model performance.

Multi-task learning (MTL) [10] 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 address multiple tasks, reducing switch resource consumption compared to deploying individual models for each task. Second, tasks with abundant, easily obtainable labels can supplement the training of *hard-to-label* tasks by contributing shared representations, thereby improving their accuracy. Thus, deploying MTL within the data-plane can be both resource efficient and enhance the performance of tasks with insufficient labeled data.

Neural network architectures are often used for MTL [10], building upon their ability to learn and share representations across multiple tasks. However, there are two challenges to implementing neural networks inference in the data-plane. First, the data-plane pipeline does not support complex operations required for neural network inference, such as matrix multiplication and floating-point operations. Second, the limited resources in the data-plane restrict the size of neural network models, making it difficult to deploy a large model 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 the neural network-based inference [11] or fully-binarized neural networks were used as alternatives [7, 8].

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 (§VI-C). Moreover, MUTA enhances accuracy in scenarios where specific tasks lack sufficient labeled data (§VI-B) by leveraging knowledge from related tasks through shared model parameters. Second, MUTA efficiently maps model layers and their associated weights to off-the-shelf programmable switches using a novel deployment methodology, enabling non-binarized MTL neural network inference in the data-plane without hardware modifications (§V). In summary, we make the following contributions:

- We introduce MUTA, an intelligent architecture that performs multiple management tasks using MTL models in the data-plane. MUTA generates a quantized MTL model suitable for deployment in the data-plane.
- We design a non-binarized multi-task neural network model that ensures efficient utilization of limited resources in data-planes and retains high accuracy when processing multiple tasks simultaneously.
- We present a novel mapping methodology for deploying the MTL model in the data-plane. To increase scalability, model's layers can be further distributed across multiple switches, with each layer's computations executed using match-action tables.

We evaluate the proposed solution using two use cases, showing that MTL can improve the accuracy of tasks with insufficient labels. The evaluation of MUTA on Intel Tofino switches shows that MUTA reduces memory usage by ×10.5 compared to single-task models, while maintaining line-rate throughput and sub-microsecond latency.

II. Related Work

A. Programmable Data-Plane

The Protocol-Independent Switch Architecture (PISA) [3] enables data-plane programmability, empowering fast innovation of networking designs. In a PISA pipeline, a 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. Finally, the processed PHV is assembled into a set of ordered headers and payload by the deparser.

While PISA supports simple operations like addition, shift and bitwise 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].

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 can use either hard parameter sharing, where some parameters of the deep learning models are shared among tasks while others are task-specific, or soft parameter sharing, where separate models for each task have their own parameters but are regularized to encourage similarity [10]. In this work, we adopt the hard parameter sharing approach as it is simple and efficient. 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 reduced memory footprint. Second, their resource efficiency is high, as they avoid repetitive features calculation in the shared layers.

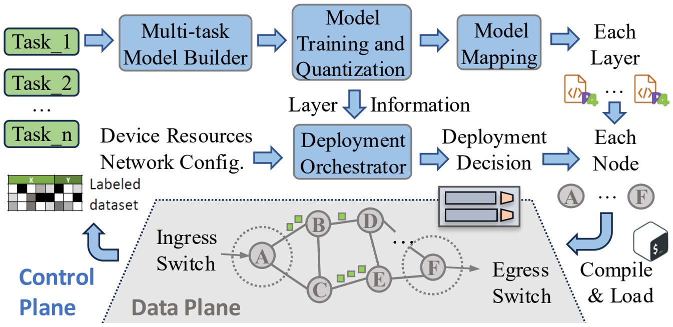


Fig. 1: MUTA architecture.

C. In-Network Neural Network Solutions

The implementation of Binary Neural Networks (BNNs) in the data-plane has been explored using commodity SmartNICs (e.g., N3IC [7]), and software switches (e.g., Qin et al. [8]). 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 [7, 8]. Following this approach, MARTINI [12] implements BNN-based MTL models in software switches. However, it has not been proven successful that these solutions can be effectively integrated into commercial switch Application-Specific Integrated Circuits (ASICs) while maintaining acceptable performance and scalability.

For higher precision in-network neural networks, INQ-MLT [13, 14] implements non-binarized neural networks for targets supporting multiplication operations (e.g., software switches [15]), and not for switch ASICs. Orthogonal to the above solution, Taurus [11] leveraged chip modification to introduce map-reduce support for neural network inference.

Our proposed work distinguishes itself from existing solutions by implementing a non-binarized MTL neural network in the data-plane and demonstrating its feasibility on a switch ASIC (Intel Tofino) without requiring any modifications to the hardware or memory architecture of existing switches.

III. An overview of MUTA

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. The control-plane periodically collects monitored traffic data from the data-plane and uses it to retrain the model.

Traffic data is labeled in the control-plane (e.g., manually) based on an application's objectives. The labeled data is used by the *multi-task model builder* to create appropriate models. After the model is built, the *model training and quantization* module generates a quantized MTL model, with parameters prepared for mapping the inference to the data-plane (§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

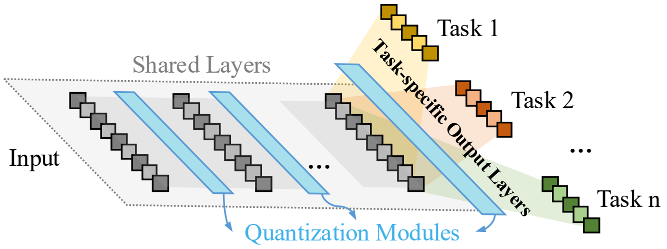


Fig. 2: Proposed multi-task learning architecture.

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 intermediate computation results are stored in the packet headers and passed sequentially to subsequent switches, enabling the network to perform layer-by-layer inference. The final prediction is obtained at the switch hosting the output layer (§V).

A *deployment orchestrator* is used to provide a recommended deployment of the generated P4 code of the MTL model across the entire network, supporting multi-path network typologies and ensuring full paths coverage. Distributed in-network computing techniques, such as those proposed in DINC [16], can be used to provide the deployment decisions.

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, because many network management tasks share underlying traffic characteristics and patterns (e.g., packet size distribution).

A. *Model Architecture and Training*

The overall architecture of our multi-task model is illustrated 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 MTL 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.

B. *Quantization*

As data-planes cannot perform floating-point (FP) operations, the weights 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 FP-based model to a quantized model which represents weights and activations using more compact format (e.g., 8-bit integers) [13].

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) [13]. 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 and mitigating the effect of quantization on overall accuracy.

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 mapping of a layer's computations to a set of match-action tables. 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.

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

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 output dimensions $(x_i w_{1i}, x_i w_{2i}, \cdots, x_i w_{mi})$ for all possible values of inputs x_i. The addition of bias can be integrated into any one of these tables, such as Table 1 in Fig.3. The looked-up intermediate values are then used for addition operations that generate output vector $\mathbf{y}$ (i.e., the element-wise sum of vectors from all match-action tables). For an input vector of size n, the switch utilizes n match-action tables to perform the vector-matrix multiplication. Once the output vector is obtained, the

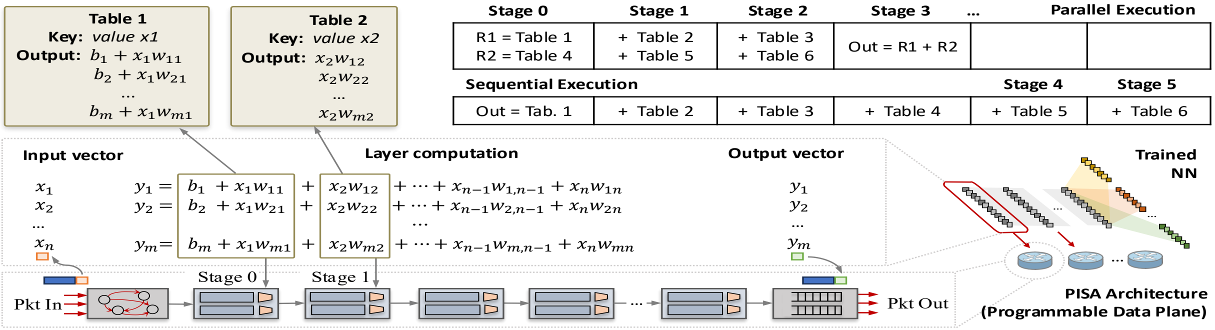


Fig. 3: Methodology for mapping layer computation to a match-action pipeline.

clamping function required by the quantization, implemented using *if-else* conditional statements, ensures each element's value falls within the bit-width range (e.g., `uint8` range is $[0, 255]$). The resulting output vector is written into the outgoing header, and serves as the input for 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) [17].

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., 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, using a total of 6 stages. 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. This parallel computation produces two intermediate results (R1 and R2). In the final stage, these two intermediate results are combined to produce the final output. Thus, a computation that required 6 stages in a sequential approach is now completed in just 4 stages. This not only saves stages, but also enhances the efficiency and reduces the latency of the computation process.

VI. Performance Evaluation

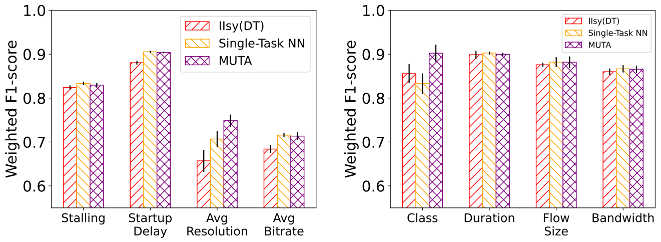
A. Use Cases

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. We use the dataset provided by [18] to tackle four tasks, i.e., video resolution, re-buffering occurrence, startup delay, and video bit-rate prediction. 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. 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 [19] captured at University of California at Davis. It contains QUIC traffic of 5 Google services: Google Docs, Google Drive, Google Music, YouTube, Google Search. We tackle four tasks, i.e., bandwidth, duration, flow size, and traffic class prediction tasks. 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 [9]. For flow size prediction, we classify the flows that belong to the top 20% as elephant flows, while the other flows are mice. Traffic class prediction is considered the *hard-to-label* task.

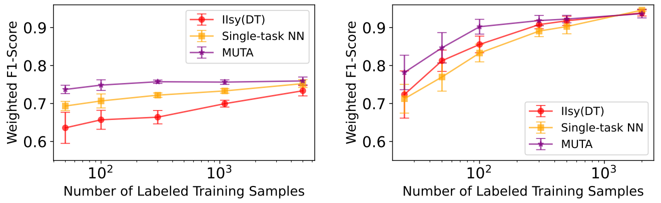
B. Multi-Task Model Performance

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



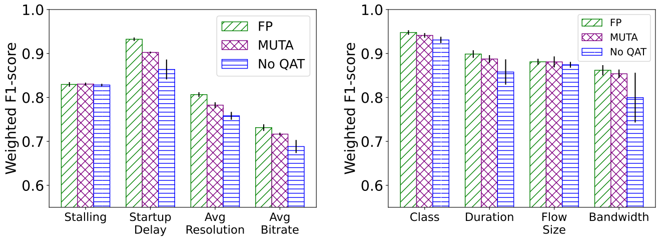
(a) QoE prediction (b) Traffic characteristics prediction

Fig. 4: Performance comparison between IIsy (DT), single-task neural network (NN), and MUTA, using only 100 samples for *hard-to-label* 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 hard-to-label 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.

multi-class classification task. 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 all decision tree models for the two use cases is set to 6.

The model training, validation, and quantization operations are performed by the control-plane using TensorFlow Lite. For each use-case, the dataset is split into training (80%) and test (20%) sets. The weighted F1-score is used to evaluate the performance. 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 data for these tasks. 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.

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.

C. Hardware Resource Consumption

1) Setting and Metrics: We implement the MTL model using $P4_{16}$ targeting Tofino Native Architecture (TNA) used in Intel Tofino switch ASIC. For resource consumption, we 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 are reported for the QoE prediction use case. MUTA is compared to two advanced tree-based solutions, i.e., the feature-encoding solution (e.g., IIsy [6]) and the direct mapping solution (e.g., pForest [5]). However, the tree models using direct mapping failed to compile due to extremely high pipeline-stage consumption. Consequently, we report results only for tree models using the feature-encoding solution (i.e., IIsy [6]).

2) Results: Table I presents the resource consumption of IIsy for each individual task, and the cumulative consumption for all four tasks combined. Similarly, Table II details the resource utilization of MUTA across each layer of the neural network. Importantly, TCAM is utilized only in the first layer of the MTL model, for the range-based match type used in the match-action table at this layer. Subsequent layers employ exact match tables exclusively. Compared to IIsy, MUTA consumes significantly lower memory resources, especially for SRAM, reduced from 197.82% to 18.845% utilization. Moreover, MUTA reduces stage consumption, requiring only 17 stages to execute all tasks, fitting within a Tofino2 switch (20 stages available) or use the proposed distributed execution

TABLE I: 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
Stage	4	8	12	5	29
Table Entries	421874	940243	1490240	526208	3378565
Metadata (bytes)	19	35	51	23	128

TABLE II: Resource Consumption for MUTA.

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

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 metadata consumption, MUTA improves memory and stage efficiency relative to IIsy.

D. Latency and Throughput

We report our measurements of pipeline latency of each layer relative to *switch.p4*, an L2/L3 reference switch program for Tofino, as the latency of Tofino is under NDA. MUTA's relative pipeline latency is computed based on data reported by SDE. As shown in Fig. 7 (a), 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. 7 (b), all layers can achieve a full line-rate of 6.4Tbps.

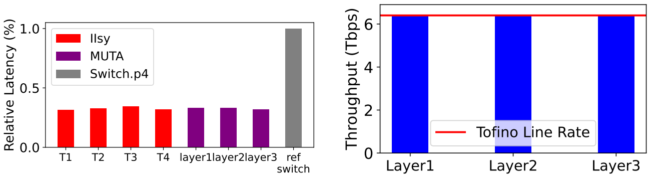
VII. Discussion

Model update: After an MTL model is deployed, it is essential to periodically update it, adapting to changes in traffic patterns. There are two update scenarios: updating model weights and model structure modification. 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.

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. Tofino2 can manage larger layers, as it supports more stages and memory resources than the Tofino chip we utilize.

VIII. Conclusion

In this paper, we introduced MUTA, a novel in-network solution for multi-task learning (MTL). MUTA enables the efficient execution of multiple network management tasks using a shared model. The proposed mapping scheme effectively maps MTL model layers into match-action tables and achieves complete model execution in the data-plane. Experimental results demonstrate that MUTA enhances performance for tasks with limited labeled data, while maintaining resource efficiency and operating at line-rate.



(a) R-Latency.

(b) Throughput (Tbps).

Fig. 7: (a) Pipeline Relative Latency (R-Latency) on Tofino for DTs, different layers in MUTA, and standalone switch.p4. (b) Throughput for each layer.

IX. Acknowledgments

This research was supported in part by EU Horizon Smart-Edge (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 et al. "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] M. Wang et al. "Machine learning for networking: Workflow, advances and opportunities". In: *Ieee Network* 32.2 (2017), pp. 92–99.

[3] P. Bosshart et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM CCR* 44.3 (2014), pp. 87–95.

[4] C. Zheng et al. "In-network Machine Learning Using Programmable Network Devices: A Survey". In: *IEEE Commun. Surveys & Tuts* (2023).

[5] C. Busse-Grawitz et al. "pforest: In-network inference with random forests". In: *arXiv preprint arXiv:1909.05680* (2019).

[6] C. Zheng et al. "IIsy: Hybrid In-Network Classification Using Programmable Switches". In: *IEEE/ACM ToN* (2024).

[7] G. Siracusano et al. "Re-architecting traffic analysis with neural network interface cards". In: *USENIX NSDI*. 2022, pp. 513–533.

[8] Q. Qin et al. "Line-speed and scalable intrusion detection at the network edge via federated learning". In: *IFIP Networking*. 2020, pp. 352–360.

[9] S. Rezaei and X. Liu. "Multitask learning for network traffic classification". In: *2020 29th ICCCN*. IEEE. 2020, pp. 1–9.

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

[11] T. Swamy et al. "Taurus: a data plane architecture for per-packet ML". In: *ACM ASPLOS*. 2022, pp. 1099–1114.

[12] S. Yoon et al. "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.

[13] K. Zhang et al. "A Machine Learning-Based Toolbox for P4 Programmable Data-Planes". In: *IEEE TNSM* (2024).

[14] K. Zhang et al. "An intelligent data-plane with a quantized ml model for traffic management". In: *NOMS 2023*. IEEE. 2023, pp. 1–9.

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

[16] C. Zheng et al. "DINC: Toward distributed in-network computing". In: *Proceedings of the ACM on Networking* 1.CoNEXT3 (2023), pp. 1–25.

[17] J. Yan et al. "Brain-on-Switch: Towards Advanced Intelligent Network Data Plane via NN-Driven Traffic Analysis at Line-Speed". In: *USENIX NSDI*. 2024, pp. 419–440.

[18] M. Seufert and I. Orsolic. "Improving the Transfer of Machine Learning-Based Video QoE Estimation Across Diverse Networks". In: *IEEE TNSM* (2023).

[19] 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).