INFOCOM 2025 1571054592

DUNE: Distributed Inference in the User Plane

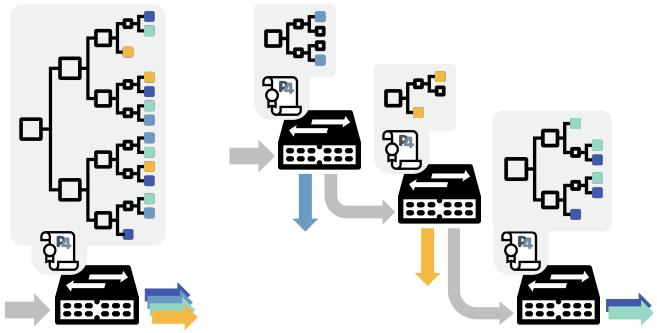
Beyza Bütün*†, David de Andres Hernandez*†, Michele Gucciardo‡, and Marco Fiore*
*IMDEA Networks Institute, Spain, †Universidad Carlos III de Madrid, Spain, ‡NEC Laboratories Europe, Spain
*{beyza.butun, david.deandres, marco.fiore}@imdea.org, ‡michele.gucciardo@neclab.eu

***Abstract*—The deployment of Machine Learning (ML) models in the user plane enables line-rate in-network inference, significantly reducing latency and improving the scalability of functions like traffic monitoring. Yet, integrating ML models into programmable network devices requires meeting stringent constraints in terms of memory resources and computing capabilities. Previous solutions have focused on implementing monolithic ML models within individual programmable network devices, which are limited by hardware constraints, especially while executing challenging classification use cases. In this paper, we propose DUNE, a novel framework that realizes for the first time a user plane inference that is distributed across the multiple devices that compose the programmable network. DUNE adopts fully automated approaches to (*i*) breaking large ML models into simpler sub-models that preserve inference accuracy while minimizing resource usage, (*ii*) designing the sub-models and their sequencing so as to enable an efficient distributed execution of joint packet- and flow-level inference. We implement DUNE using P4, deploy it in an experimental network with multiple industry-grade programmable switches, and run tests with real-world traffic measurements for two complex classification use cases. Our results demonstrate that DUNE not only reduces per-switch resource utilization with respect to legacy monolithic ML designs but also improves their inference accuracy by up to 7.5%.**

I. Introduction

User plane programmability is enabling innovation across a range of traditional network functions that include monitoring and telemetry, load balancing, caching, intrusion detection and verification, just to name a few among the many surveyed by recent reviews [1]–[3]. Among the many applications above, programmable network hardware has also paved the road for the deployment of Machine Learning (ML) models in the user plane for in-network inference. Indeed, ML models trained off-line can be implemented into programmable switches or smart Network Interface Cards (smartNICs), where they are applied to every transiting packet at line rate [4]. In-network ML allows performing inference tasks at the equipment forwarding capacity (*e.g.*, on Tbps traffic in modern switches) and with ultra-low latency (*e.g.*, in the order of tens of nanoseconds).

The advantages of user-plane inference over traditional control-plane ML are substantial, since the elimination of cross-plane interactions improves delays (cut down by orders of magnitude), scalability (eliminating overheads, *e.g.*, for traffic mirroring) and costs (removing the need for expensive dedicated hardware, *e.g.*, for Deep Packet Inspection). Yet, embedding ML models into programmable network equipment also entails remarkable challenges, due in particular to the severe limitations of the hardware in terms of compute capabilities and memory resources as well as to internal architectures that are purposed for forwarding and not for ML operations. Ultimately, these restrictions let user-plane ML pay a price in terms of inference accuracy as the task complexity grows [5].



(a) Monolithic (b) Distributed

Figure 1: Conceptual illustration of (a) traditional monolithic and (b) proposed distributed user-plane inference strategies.

A variety of solutions have been proposed in the past few years to adapt ML models to user plane environments, by carefully tailoring the design and mapping of the original ML architectures to the highly constrained programmable network hardware targets. Prior works have explored different ways to perform inference in switches [6] possibly augmented with external accelerators [7] and in smartNICs [8], using Decision Trees (DTs) [9], Random Forests (RFs) [10] or Neural Networks (NNs) [11] operating on features extracted at packet-level [12], flow-level [13] or at both levels jointly [5].

However, proposals in the literature have invariably sought to implement *monolithic* ML models into a single network equipment. We argue that this strategy is curbing the potential of ML-driven inference in network user planes that are inherently composed of tens or hundreds of switches and middleboxes. A much more flexible strategy is that of *distributing* the ML model across multiple network devices so that the inference task is performed over packets and flows as they traverse the network rather than at one specific point. Figure 1 illustrates this concept. In previous studies, the whole ML model (*e.g.*, a DT in the example) is coded as P4 programs and fitted into a single network device (*e.g.*, a switch in the example), abiding by its strict resource constraints. Our proposal is to *decompose* the operation of the original monolithic ML model into sub-models that (*i*) can be deployed as P4 programs into different devices and (*ii*) jointly execute the target inference task. As per the figure, this lets each device

solve a portion of the problem (*e.g.*, identifying a specific traffic class) and leave the rest to the downstream network.

While the concept is simple, its realization poses a number of challenges. How to best decompose the original ML model in order to preserve accuracy while keeping the sub-model resource usage under control, how to order the resulting sub-models, and how to ensure automation of such a decomposition process for any inference task are hard and open questions.

In this paper, we present DUNE, the first practical framework for the execution of distributed user plane inference in real-world programmable hardware. DUNE focuses on a specific type of inference, *i.e.*, classification, since this is the relevant task in the vast majority of network functions that include traffic classification, intrusion detection, application identification, spam filtering, bandwidth management, and congestion management. The design, implementation and evaluation of DUNE yield the following main contributions.

- We propose a novel framework to decompose large ML models trained to solve complex inference tasks into simpler sub-models that jointly perform the same operation. The framework is fully automated, can be applied to any class of input ML model, and solves original optimization problems in order to preserve the inference accuracy while minimizing the complexity of the sub-models.
- We introduce a design of the sub-models and of their interactions that allows executing joint packet- and flow-level inference in a distributed way across multiple programmable network devices.
- We implement DUNE in an experimental testbed with industry-grade programmable switches and test it with real-world measurement data, demonstrating its viability in practical scenarios.
- We show how distributed inference grants higher accuracy than the traditional single-device monolithic approach by up to 7.5%, while also using less resources per switch.

II. Related work

User-plane inference, also referred to as in-network ML, has been postulated as a viable paradigm only a few years ago [12], [14], [15], and has since attracted a growing attention by the research community. The field is now very active and even a recent first survey published this year [4] does not capture the substantial innovation that occurred in the past six months.

Solutions to deploy trained ML models in programmable network hardware have considered different targets, including network accelerators [7] and SmartNICs [8], yet programmable switches stay the favourite environment for user-plane inference and the focus of all researches summarized next.

Monolithic tree-based models. The vast majority of the literature proposes models based on tree structures, *i.e.*, DTs or RFs, that are implemented in individual switches. Early solutions deal with the efficient mapping of trees to the switch Application Specific Integrated Circuit (ASIC) [6], [16], [17], considering individual DTs [18]–[21] or special-purpose solutions tailored to one inference tasks [22], [23]. Subsequent works propose knowledge distillation approaches to cope with the limited switch resources [9], exploit both ingress and egress stages of the switch ASIC [24], provide support for inference based on flow-level features [13] or consider additional tree-based models [10]. Recent efforts achieve joint packet- and flow-level inference by automatically selecting the best option based on stateful information about each traffic flow [5], [25]. All these works invariably aim at deploying monolithic ML models that perform all inference operations, including feature extraction, state maintenance, tree traversal and consensus resolution, in a single programmable switch.

Monolithic neural networks. Neural architectures are significantly more involved than DTs or RFs, and embedding them in programmable switches has proven especially challenging. Still, after early attempts [26] several breakthroughs have led to workable implementations that rely upon custom activation functions [27] or Recurrent Neural Network (RNN) architectures adapted to the limited data plane stages [11]. Again, all these approaches propose to implement one NN in one switch.

Distributed ML. There exist several ways in which user-plane ML-driven inference can be distributed. In a first model, *inference is distributed across planes*, with part of the process run in a programmable network equipment and part in a more powerful control-plane server; several works have explored this direction [28]–[30] but all assume that user-plane operations happen in a same device.

A second notion of distribution is the *separation of feature extraction from the rest of the inference process*, with the former run in different devices than the latter; the position paper proposing this strategy [31] still considers the ML models to be monolithic and deployed in single switches.

The third acceptation is that of our interest, and intends to *distribute the operation of the ML model itself across multiple user-plane devices*, as depicted in Figure 1b. To the best of our knowledge, the only solution proposed to date in this direction is the very recent NetNN [32], which deploys a NN across a programmable network by implementing different layers of the neural architecture in subsequent switches. However, NetNN realizes a specific neural model with a precise sequence of convolutional and dense layers trained for one task, *i.e.*, intrusion detection; instead, we seek a general-purpose solution that can distribute any ML model across a given programmable network. Moreover, NetNN is only emulated with the bmv2 behavioral model that is known to simplify many of the limitations of the real-world ASICs, and may result in performance losses and a need for partial model re-design when porting the solution to actual hardware [33].

Dataplane Disaggregation. The concept of dataplane program disaggregation was introduced by Flightplan [34], a toolchain that splits a P4 program into subprograms that can be executed across devices. This approach allows for the combination of heterogeneous hardware to overcome the limitations of single-target execution and efficient resource usage. While Flightplan provides a framework for automated splitting, placement, and runtime support, the splitting task relies on a manual segmentation approach. Automating the splitting task is not straightforward nor easily generalizable, as each NF type

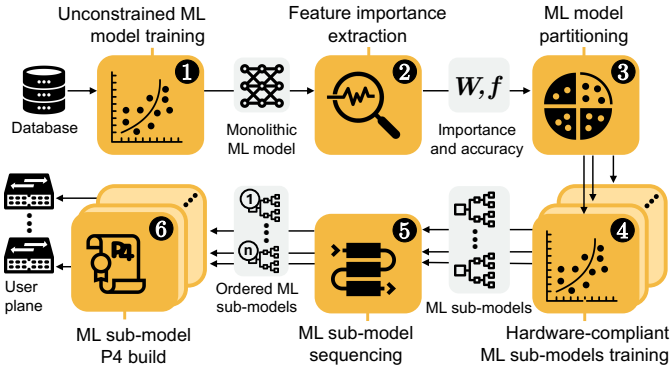


Figure 2: Overview of the DUNE workflow for the design of distributed ML compatible with programmable user planes.

might require specific segmentation procedures, constraints, and goals. More recently, DINC [35] has also addressed the challenges of distributed in-network computing, with special attention to routing through the network. Like Flightplan, DINC also relies on manual markers to identify the program segments, but unlike Flightplan which focuses on single path scenarios, DINC considers multiple paths in the network and aims to ensure that the service is provided for any set of paths without routing changes.

Progress beyond the state of the art. We present DUNE, a solution that can automatically disaggregate any ML model for classification so that it can be deployed over a programmable network. Our solution is practical as its design is tailored to real-world hardware and its operation is demonstrated with industry-grade programmable switches. We stress that DUNE is orthogonal and complementary to solutions like Flightplan [34] or DINC [35]; indeed DUNE produces the task *segmentation* that is an input to such solutions.

III. Distributed ML training with DUNE

As any other solution for user-plane ML [5]–[27], DUNE is composed by an off-line phase for the design and training of the ML model that is run in the control plane, and an on-line phase where the trained model deployed in the programmable network performs line-rate inference on the traffic. We present in this section the first phase, and in Section IV the second.

A. Workflow overview

The high-level workflow of DUNE during the off-line phase is outlined in Figure 2. Here, the goal of the framework is to generate a set of ML sub-models that (i) are compatible with the programmable network constraints and (ii) jointly execute the desired inference task. To this end, historical measurement data about the target inference task, suitably labeled for supervised learning, is initially fed to a first ML model training stage ①. At this stage, the goal is to train the most accurate ML model possible, ignoring constraints from the programmable network hardware, and the framework can accommodate any ML paradigm, *e.g.*, deep neural architectures, support vector machines or tree-based models, as expounded in Section III-B. The output of stage ① is a trained complex monolithic model that maximizes accuracy for the target traffic analysis task.

In stage ②, the monolithic ML model is examined so as to extract values that describe the importance of each input feature (*e.g.*, packet header fields or flow-level statistics) to estimate every model output (*e.g.*, a class of network traffic). Different tools can be used to that end as outlined in Section III-C, producing an importance matrix $\boldsymbol{W}$ that expresses the relevance of features in explaining output variables and an accuracy vector $\boldsymbol{f}$ of the inference quality for each class.

The information in $\boldsymbol{W}$ and $\boldsymbol{f}$ is used in stage ③ to compute a partitioning of the original inference task in to sub-tasks. Each sub-task employs a subset of the input features to estimate a specific disjoint subset of the output variables, and the partitioning aims at creating groups of variables that can be well explained by a compact set of features. To identify the best partitioning, DUNE hinges upon an original optimization problem solved via an efficient heuristic, as per Section III-D.

For each sub-tasks, stage ④ trains a dedicated ML sub-model that predicts the sub-task variables using the sub-task input features. In this case, the ML sub-models are designed abiding by the user-plane hardware constraints so as to ensure their compatibility with the target programmable network, as described in Section III-E.

As the ML sub-models tackle different portions of the whole inference task on the same network traffic, they must be run in sequence to execute the original function, similar to the chaining of Virtual Network Functions (VNFs). However, chaining ML sub-models propagates inference errors, which makes it critical to account for their (diverse) reciprocal accuracy when establishing an order of execution. As an intuitive example, it is critical to avoid deploying at the entry of the chain a ML model with a high rate of false positives: this leads to a large number of early misclassified flows, which are then never received by the downstream sub-models intended to infer the true category of such flows. As discussed in Section III-F, stage ⑤ of DUNE solves a dedicated optimization problem based on the reciprocal accuracy of the sub-models to identify their ordering yielding the highest inference performance.

Finally, in the last stage ⑥, each ML sub-model is coded as a P4 program and injected into a programmable user-plane device. It is worth noting that the problem of identifying (*e.g.*, within a complex network topology) the exact devices where the sequence of ML sub-models shall be deployed goes beyond the scope of DUNE and can be addressed with solutions already present the literature. In fact, one can consider the ordered sub-models generated by DUNE as a chain of user-plane VNFs and apply one of the many algorithms proposed in the literature for VNF chaining optimization [36]; or, regard the output of our framework as a segmentation of an in-network computation task and employ a dedicated solution like DINC [35].

B. Unconstrained ML model training

Stage ① consists of the definition, hyper-parametrization and training of a single ML model that solves the desired inference task. As anticipated, this model is *unconstrained*,

i.e., is not limited by the programmable hardware memory or compute constraints, hence can be as complex as the control-plane ML Operations (MLops) resources allow. The exclusive objective of this unconstrained ML model is achieving the maximum accuracy possible in the target inference task.

While this stage of DUNE is general and can accommodate different ML paradigms, in our implementation we opt for using a large RF as the unconstrained model. The motivation is twofold: first, RFs are inherently more explainable than neural architectures, making the extraction of feature importance from a trained model much faster and more precise, as explained in Section III-C; second, for the challenging traffic classification use cases employed to evaluate the performance of DUNE and presented later in Section V, using NNs based on Mutit-Layer Perceptron (MLP) as the unconstrained ML models returned sensibly lower accuracy than that of RFs.

To design the unconstrained RF, we start from the RF model recently proposed by Jewel [5], a tree-based architecture that can perform packet-level inference (*i.e.*, using only features extracted from the header of the current packet, such as payload length) on the first packets of a flow and automatically switch to a more accurate flow-level inference (*i.e.*, using features that relate to all received packets in a same flow, such as the average inter-arrival time) as soon as enough packets in the flow have been seen to build reliable flow statistics. Note that we consider the monolithic version of this type of tree that is originally implemented in Jewel as a benchmark in the comparative evaluation in Section VI.

DUNE hyper-parametrizes the joint packet- and flow-level RF model by training it with an exhaustive grid search over the space of hyper-parameters, *i.e.*, the maximum number of trees, the maximum depth of each tree, and the rank of the packet for which the flow-level inference is triggered. The grid search is iterated so as to also identify the optimal set of input features that maximizes the unconstrained model accuracy: specifically, we start from a model trained with the full set of all available features, compute the Mean Decrease in Impurity (MDI) to remove the least relevant feature, re-run the grid search and iterate. In the end, we select the RF model that achieves the highest *macro F1* score (see Section V-C for a definition).

C. Feature importance extraction

Stage ② extracts the relationships between input features and output variables from the unconstrained model produced by the previous stage. It is worth highlighting that here we do not seek to rank the importance of the input features for the inference task as a whole, for which standard techniques such as the MDI used in stage ① exist. Instead, we need to quantify the explanatory power of each input feature for each individual output class, which is a sensibly harder problem.

DUNE takes advantage of the design choice of adopting an RF model in stage ① to extract feature importance in an efficient way. Specifically, it leverages the Per-Class Feature Importance (PCFI) method [37], which operates on tree structures and extends MDI by considering the paths (*i.e.*, sequences of nodes from the root node to a leaf that contains the inference decision) that compose in the trained RF. For each path in every RF tree, PCFI calculates the importance of a given feature as the total MDI at nodes that split their sub-trees based on that feature; it then associates such an MDI value to the class in the leaf at the end of the path. Once all paths have been processed, PCFI computes the final importance values for a feature-class pair as the mean of all MDI values of the input feature associated to the target class.

The PCFI approach is not the only one possible. For instance, the well-known SHapley Additive exPlanations (SHAP) [38] offer an alternative way to derive from a trained model how much each input feature contributes to the inference of an individual output variable. SHAP computes so-called Shapley values for each feature and produced inference output by evaluating all possible permutations of the features and determining the marginal contribution of the feature to the prediction as the difference caused by its absence. While SHAP has the advantage of being applicable to any ML model, suffers from poor exponential scalability in the model complexity [39] that makes it impractical in many cases.

In the case of tree-based models, a compute-efficient variant of SHAP exists: TreeSHAP [40] leverages the structure of trees to compute exact Shapley values in polynomial time. Yet, the complexity of PCFI and TreeSHAP is still very different. Let us denote by $\mathtt{t}$, $\mathtt{L}$, and $\mathtt{d}$ the number of trees, the maximum number of leaves in a tree, and the maximum depth of each tree in the unconstrained RF model, respectively: then PCFI has a complexity $O(\mathtt{t}\cdot\mathtt{L}\cdot\mathtt{d})$ whereas that of TreeSHAP is $O(\mathtt{t}\cdot\mathtt{L}\cdot\mathtt{d}^2)$. As tests with the reference use cases in Section V show similar accuracy in the feature importance returned by PCFI and TreeSHAP, we opt for the former in our implementation.

D. ML model partitioning

Stage ③ breaks down the original inference task into a series of smaller sub-tasks that jointly achieve the same goal. To this end, DUNE solves an original Set Partitioning Problem (SPP) to identify sub-tasks as disjoint groups of output classes that can be explained with high accuracy by a compact set of input features each. We next formalize SPP as an optimization problem and present the efficient heuristic used to solve it.

1) Formulation: Let $V = \{v_1, \ldots, v_m\}$ be the set of all m output variables (*i.e.*, classes) that must be explained with the set of r features in the inference task. Also, let us denote by $S = \{S_1, \ldots, S_n\}$ the set of all possible blocks (*i.e.*, class groups). $P \subset \{1, \ldots, n\}$ is a partition of V if and only if

$$\bigcup_{i \in P} S_i = V \quad \text{and} \quad S_i \cap S_j = \emptyset \quad \forall i, j \in P, i \neq j. \tag{1}$$

The goal of the SPP is to find the partition P that best balances inference accuracy and programmable network resource use. This translates into the two following requirements.

(i) *Compactness.* P shall be formed by the smallest possible number of blocks. This requirement stems from the fact that each block will be later mapped by our framework into a separate ML sub-model, and every added sub-model yields overhead (as it requires, *e.g.*, maintaining its

own flow state registers or mapping its ML logic to a user-plane device) and grows the number of network devices required to implement the full distributed solution.

(ii) *Effectiveness*. Each block in P shall maximize the accuracy of inference for the variables it includes by using a minimum number of features. The rationale is that accuracy must be preserved with limited feature sets, which later translate in less complex ML sub-models.

It is worth noting that the two requirements above entail a clear trade-off: the best accuracy using small feature sets is achieved with tiny blocks each dedicated to very few variables; yet, that also creates a large number of ML-models, which ultimately harms resource usage. The SPP thus aims at finding the best partition that solves such a trade-off.

The optimization problem can be formalized using a matrix representation. Let $\boldsymbol{s}_i \in \mathbb{Z}_2^m, i \in \{1, \ldots, n\}$ be an indicator vector designating which variables in V are part of a block S_i, *i.e.*, the k-th element of $\boldsymbol{s}_i$ is 1 if class k is in S_i and 0 otherwise. We then construct a matrix $\boldsymbol{A} \in \mathbb{Z}_2^{m \times n}$ with the $\boldsymbol{s}_i$ vectors as the n columns. By defining a vector of binary decision variables $\boldsymbol{x} \in \mathbb{Z}_2^n$ whose i-th element indicates if a block S_i is selected as part of the output partition P, we can write the constraints in (1) as $\boldsymbol{Ax} = \mathbf{1}$. The formal definition of the optimization problem is then

$$\max_{\boldsymbol{x}} \ \frac{1}{c(\boldsymbol{x})} \cdot \boldsymbol{g}^T \boldsymbol{x} \qquad \text{s.t.} \quad \boldsymbol{Ax} = \mathbf{1}, \tag{2}$$

where $c(\boldsymbol{x})$ captures requirement (i) above and denotes the cost of partitioning the inference task into the number of blocks defined by $\boldsymbol{x}$, whereas $\boldsymbol{g}^T\boldsymbol{x}$ models requirement (ii) as the accuracy achieved for the blocks in $\boldsymbol{x}$ with minimum features.

More precisely, the cost $c(\boldsymbol{x})$ is expressed as

$$c(\boldsymbol{x}) = \frac{m-1}{m - \mathbf{1}_n^T \boldsymbol{x}}, \tag{3}$$

where $\mathbf{1}_n \in \mathbb{Z}_2^n$ is a vector of 1's hence $\mathbf{1}^T\boldsymbol{x}$ is a scalar corresponding to the number of non-zero elements of $\boldsymbol{x}$, *i.e.*, the number of blocks in the selected partition. This entails a linearly decreasing advantage $1/c(\boldsymbol{x})$ as the number of blocks grows up to the total number of variables m.

As far as the second factor of the objective function in (2) is concerned, $\boldsymbol{g} \in \mathbb{R}^n$ is a vector of the gains g_i associated with all of the possible blocks S_i, which is expressed as

$$g_i = \Theta(\boldsymbol{s}_i) \cdot \Psi(\boldsymbol{s}_i), \quad \text{where} \tag{4}$$

$$\Theta(\boldsymbol{s}_i) \ = \ \max_{\boldsymbol{\phi}} \ \boldsymbol{s}_i^T \boldsymbol{W} \boldsymbol{\phi} - (\tfrac{1}{r}(\mathbf{1}_n^T \boldsymbol{s}_i)\mathbf{1}_r)^T \boldsymbol{\phi} \tag{5}$$

$$\Psi(\boldsymbol{s}_i) \ = \ 1/(1 + \max_{v_k | \boldsymbol{s}_i(k)=1} f_k - \min_{v_k | \boldsymbol{s}_i(k)=1} f_k). \tag{6}$$

In the expression in (5), the matrix $\boldsymbol{W} \in \mathbb{R}^{m \times r}$ contains the per-class feature importance values returned by stage ②, while $\boldsymbol{\phi} \in \mathbb{Z}_2^r$ is an array denoting a feature subset, *i.e.*, the h-th element is 1 if feature h is in $\boldsymbol{\phi}$, and 0 otherwise. Then, $\boldsymbol{s}_i^T \boldsymbol{W} \boldsymbol{\phi}$ is the cumulative importance retained by the selected features in $\boldsymbol{\phi}$ for classes in S_i, whereas $(\frac{1}{r}(\mathbf{1}_n^T \boldsymbol{s}_i)\mathbf{1}_r)^T \boldsymbol{\phi}$ is a penalty on the number of features used to achieve such importance.

In (6), f_k is the k-th element of array $\boldsymbol{f} \in \mathbb{R}^n$ provided by stage ② and denotes the accuracy (*e.g.*, the F1 score defined in Section V-C) achieved by the unconstrained ML model in inferring class v_k. The expression introduces a penalty proportional to the maximum difference in accuracy for two variables in the block S_i: high differences indicate inherently diverse complexity of the inference process for the corresponding classes, suggesting that they need ML sub-models of sensibly different size and are thus not good candidates for grouping.

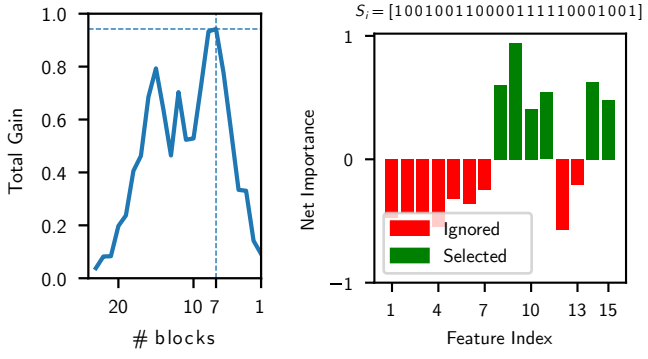
2) Solution: The SPP is known to be NP-hard even in its basic optimization problem formulation where the objective function is linear in the decision variables $\boldsymbol{x}$. Yet, in that case the SPP can be rendered as an Integer Linear Problem (ILP) and solved using standard solvers up to a reasonable size. Unfortunately, the objective function in (2) is non-linear, impeding such a simplification and making an optimal solution computationally impractical.

Algorithm 1 SPP greedy algorithm

```
1: level ← m, gain ← 0, (blocks) ← all single-element blocks
2: for i = m, ..., 2 do
3:     tuples ← COMBINATIONS(blocks, 2)
4:     for all tuple ∈ tuples do
5:         gains ← COMPUTEBLOCKGAIN(tuple)
6:     end for
7:     best tuple ← argmax(gains)
8:     for all block ∈ blocks do
9:         for all element ∈ best-tuple do
10:            if element ∈ block then
11:                REMOVE(element, blocks)
12:            end if
13:        end for
14:    end for
15:    ADD(best-tuple, blocks)
16:    gain ← COMPUTEPARTITIONGAIN(blocks)
17:    if gain > gain-max then
18:        level ← i
19:        gain-max ← gain
20:    end if
21: end for
```

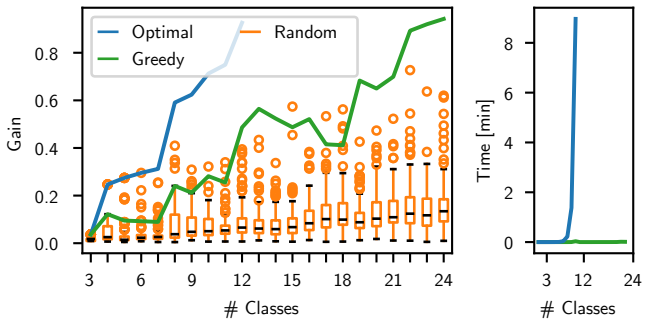
We propose a greedy approach, summarized in Algorithm 1, to approximate (2). The algorithm mimics an agglomerative clustering process where the gain in (4) is used as the *similarity metric* to merge at every iteration the two (blocks of) variables with maximum similarity. The algorithm performs $m-1$ iterations (line 2); at each iteration, it computes gains[1] among all current blocks (lines 4–6), identifies the pair of blocks with maximum gain (line 7), and updates the blocks data structure with the new merged block (lines 8–15). Each iteration returns a partition whose total gain is computed using (2) and possibly stored as the maximum gain achieved (lines 16–20). Such gains are not monotonic in general with respect to the iterations, but take different shapes depending on the target inference task. Figure 3a shows the evolution of the gain matching our objective function (2) over iterations, for one of the reference use cases we will present in Section V.

[1]Gains calculated at a previous iteration are cached and not re-computed.



(a) Blocks selection (b) Feature selection

Figure 3: Greedy solution to the SPP in (2) in the UNSW use case. The plots show (a) the gain versus the number of blocks and (b) the feature selection for a given block S_i.



(a) Gain performance (b) Execution time

Figure 4: SPP algorithms comparison over the UNSW dataset.

The computation of the gain in (4) needs to be especially efficient since it is profusely used in Algorithm 1, hence we propose a computationally viable implementation with cost $O(r)$. The gain is a product of (6) and (5): while the factor in (6) is inexpensive, we decompose the calculation of (5) into r sub-problems leveraging its structure. In each of sub-problem, a feature is selected if the importance gain, *i.e.*, the minuend in (5), overcomes the uniform penalty, *i.e.*, the subtrahend in (5). For example, Figure 3b shows in green the features with a positive net importance gain for a given block s_i of the SPP for to the same inference task of Figure 3a.

Figure 4 provides a visualization of the accuracy-complexity trade-off of the greedy algorithm solving the SPP. We consider again the same inference task of Figure 3, but limiting it to a given number of classes, shown in the abscissa of the plots: when considering, *e.g.*, 9 classes, we randomly select 9 of the 24 classes that constitute the problem and run the greedy algorithm to identify a good partition of such 9 classes. Figure 4a shows that our proposed heuristic (green) performs substantially better than stochastic partitions (orange) with a gain that tends to increase as the complexity of the SPP grows. The optimal solution (blue) clearly outperforms the greedy solution, yet Figure 4b shows that it also has an exponentially growing execution time that does not allow solving the SPP for more than a handful of classes. Instead, the greedy algorithm has a reasonable computational cost $O(m^2)$ in Figure 4b.

E. Hardware-compliant ML sub-model training

Stage ④ trains one ML sub-model for each block in the partition P^* output above. Unlike the unconstrained model trained in stage ①, these sub-models are designed to comply with the constraints of the target programmable network devices. Each sub-model aims at classifying the variables (*i.e.*, classes) present in the corresponding block, plus one additional *others* category that gathers all classes that are part of the total inference problem but are not included in the current block.

Our implementation of DUNE relies on RFs to realize each ML sub-model. This choice is grounded in the significant success that RFs had as a reference ML paradigm for user-plane inference, as reported in Section II. The model design, hyper-parametrization and training are thus similar to those described in Section III-B and rely on a joint packet- and flow-level inference model. The difference is that the bit representation of the features, the maximum umber of trees and the maximum number of leaves per tree are now constrained to abide by the limitations of user-plane hardware. Indeed, the limited size of register entries, the small amount of memory, and the finite number of stages in programmable devices force restrictions that must be accounted for at design stage to ensure the compatibility of the ML models with the network equipment. Since we target a user plane composed of programmable switches, we adopt the techniques introduced in Flowrest [13] to (i) engineer features and (ii) limit tree dimension, thus ensuring that the design of the ML sub-models is fully compatible with the network hardware.

The best RF sub-model for each block is identified from the grid search over the (constrained) hyper-parameters by balancing accuracy and resource usage. Specifically, we select the RF that maximizes $\alpha \cdot \text{F1} + (1-\alpha) \cdot \text{U}_{\text{TCAM}}$, where F1 is the *macro F1* score defined in Section V-C, U_{TCAM} is the expected usage of Ternary Content-Addressable Memory (TCAM) in the switch, and α is a tunable parameter that we set to 0.5 in our experiments to fairly balance the two contributions.

F. ML sub-model sequencing

Stage ⑤ orders the ML sub-models so as to optimize the inference performance. The sub-model sequencing impacts the distributed inference accuracy due to the following aspects.

- Confusion between sub-models addressing different parts of the global inference. The problem arises due to false positives from an upstream sub-model that generate incorrect early decisions on flows that in fact belong to the target classes of downstream sub-models.
- Diverse accuracy of each sub-model. Sub-tasks have non-uniform complexity, and each sub-model has a different error rate that affects all downstream sub-models.

The goal of this stage is then to generate an ordering such that ML sub-models with a low rate of false positives and a high accuracy are deployed earlier in the sequence. DUNE solves a simple optimization problem to that end. The objective is

$$\min_{\boldsymbol{X}} \sum_{i,j \in P^*,\, i \neq j} p_{ji} \cdot (1 - f'_i) \cdot x_{ij}, \quad x_{ij} \in \boldsymbol{X} \tag{7}$$

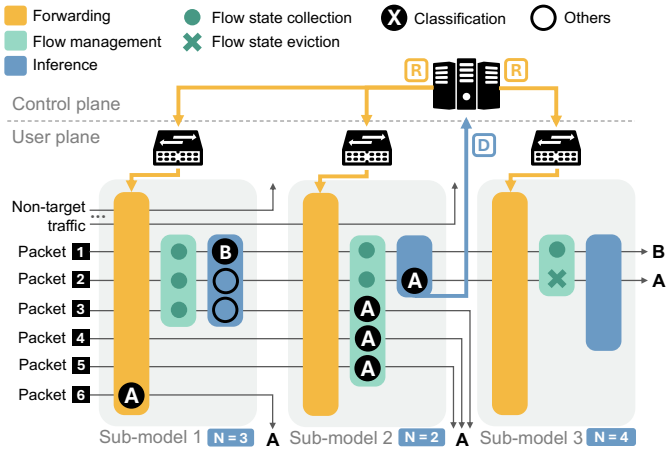


Figure 5: Toy example of the DUNE user plane operation with a distributed ML model over three switches performing inference on packets of one target flow.

where p_{ij} is the rate of false positives[2] that the ML sub-model of block S_i generates with respect to classes in block S_j of the partition P^*, f_i' is the *macro F1* score of the sub-model for block S_i trained in Stage ③, and x_{ij} are binary decision variables that take value one if the ML-model of block S_i precedes that of block S_j in the ordering.

In order to ensure that each ML sub-model is executed only once in a valid order, we define the two following constraints.

$$x_{ij} + x_{ji} = 1, \quad \forall i, j \in P^*, i \neq j, \tag{8}$$

$$u_i - u_j + x_{ij} \cdot \|P^*\| \leq \|P^*\| - 1 \quad \forall i, j \in P^*, i \neq j, \tag{9}$$

where u_i is an auxiliary variable associated with the block S_i used to enforce the order of deployed sub-models and $\|P^*\|$ is the total number of blocks. The problem in (7)–(9) is a version of the well-known Travelling Salesman Problem and can be formulated as an ILP and solved efficiently.

IV. DUNE USER-PLANE OPERATION

Once the preparation of the ordered ML sub-models is complete, each RF can be coded as a P4 program and deployed in the target programmable user plane for distributed inference, as per stage ⑥ in Figure 2. While the general framework we propose can be adapted to different user-plane configurations potentially combining diverse types of programmable network devices, the implementation of DUNE we produce and test in this paper is intended for a network of programmable switches.

The high-level operation of DUNE in the user plane is illustrated in a simple example in Figure 5, where a sequence of three ML sub-models are deployed into three switches. Each switch is programmed so as to perform three sets of operations on each transiting packet: (*i*) *forwarding* is the baseline function of the switch and consists in redirecting incoming traffic to its next hop or to the ML processing, depending on whether the flow is a target for inference or not; (*ii*) *flow management* is only applied to flows for which inference needs to be run, and stores stateful information about the flow-level features and situation (*e.g.*, classified or not) of every target flow; (*iii*) *inference* is the actual joint packet- and flow-level RF operation that classifies each packet using its packet-level features and, for the N-th packet that triggers flow-level inference, flow-level features as well. We implement these three sets of operations as thoroughly described in [5], and adopt the well-known Planter strategy [16] to map efficiently RFs to the Match-Action Units (MAU) of each switch ASIC.

Once the network is programmed, the operation proceeds as illustrated in Figure 5. Traffic that is not a target for inference is normally forwarded by the entry switch according to its legacy forwarding table. When the first packet of a flow that has to be classified is received by the entry switch (*e.g.*, packet 1 in the figure, whose flow we assume to belong to class A), it is forwarded to the local flow management stage, where flow-level information is collected and stored, and then moved to the inference stage. Since the first ML sub-model in this example performs flow-level inference on packet N=3, packet 1 of the flow is classified using packet-level features only, and in this specific example, is misclassified as belonging to class B. The packet is then moved along the distributed ML pipeline, and traverses the two remaining switches: in each case, it is identified as a packet to be classified, flow-level information is collected, but the packet skips the inference stage since it has already been tagged with a class, although incorrectly.

When packet 2 arrives, it follows a similar path. However, let us assume that this time the packet-level inference run by the first switch does not tag packet 2 with a class, but marks it as *others*, *i.e.*, belonging to a class pertaining to a downstream ML sub-model as discussed in Section III-E. Hence, packet 2 is forwarded to the second switch, which updates its internal flow-level state with information from this packet. In the example, the second ML sub-model is, in fact, responsible for inferring packets and flow belonging to class A; also, the flow-level inference point in this model is N=2. The sub-model tags packet 2 as pertaining to class A, and updates its local flow management to automatically associate all following packets for the same flow to class A. Note that packet 2 is still forwarded to the third switch, which lets it evict the associated flow-level data from the flow management registers. The classification of the flow also triggers a digest message D from the second switch to the control plane; this lets the centralized controller inject new forwarding rules R in all switches so that the forwarding stage takes care of redirecting the traffic according to the inference result (as it happens to packet 6 that is received after the rule update), and all flow-level registers can be freed.

It is worth noting that simple sub-models can be implemented in the same switch, in which case they share flow management registers and any common features. We open-source our code [41] for full reproducibility of the solution and complete transparency of our DUNE user-plane implementation.

[2]Since each ML sub-model is trained on the entire set of classes, it is possible to calculate the false positives across blocks. For instance, for the sub-model associated to block S_i we can compute the rate of flows belonging to the classes in block S_j that such a model misclassifies as its own.

V. Experimental setup

A. Programmable network testbed

The network testbed where we test DUNE comprises 2 servers and 3 switches. The servers are equipped with AMD EPYC 24-core processors running at 2.8GHz, with 128GB of RAM, and QSFP28 interfaces. The switches are Edgecore Wedge 100BF-QS programmable switches, featuring Intel Tofino BFN-T10-032Q chipsets and 32 100GbE QSFP28 ports. All switches run the Open Network Linux (ONL) operating system, and the Intel Software Development Environment (SDE) version 9.7.0 used for compiling P4 programs. During the experiments, our solution, DUNE, is deployed in a distributed fashion across the three switches, with one server dedicated to implementing the control plane. The functionality of the control plane is to host controller that binds to the Barefoot Runtime Interface (BRI) of the switch and performs the initial setup of the switch, activating ports, loading the trained ML models as table entries via a P4 program, and getting the digest from the user plane to collect classification results and update registers and tables throughout the experiment. Another server is used to inject traffic into the switches. The experimental testbed operates as a comprehensive 100-Gbps platform.

B. Inference use cases

We select two complex classification tasks in device identification and attack detection that are based on publicly available real-world measurement data to show how DUNE performs across different and demanding use cases.

UNSW-IoT [42] is a device identification use case based on a traffic measurement data collected from 26 Internet of Things (IoT) and several non-IoT devices. Measurements are conducted in a living lab emulating a smart environment over a period of 6 months. In order to detect which of the 26 devices that traffic flows belong to, we train ML models with 15 days of data and test with one day of data.

ToN-IoT [43] is an attack detection use case where benign and 9 types of cyberattacks are generated within a representative medium-scale testbed. This testbed comprises several IoT and industrial IoT (IIoT) devices, along with various non-IoT devices. For our analysis, we use 75% of the data for training and the remaining 25% for testing. Despite the dataset containing one benign and nine types of cyberattacks, three of the target classes with quite lower samples in the test set. Hence, we exclude these classes during training, resulting in the classification of seven classes instead of the original ten.

C. Inference accuracy metrics

We assess the performance of DUNE and selected benchmarks using F1 score metric, which is commonly used for evaluating classification solutions. This metric is calculated based on three key measures: true positives (TP), false positives (FP), and false negatives (FN) as F1=2TP/(2TP+FP+FN). We compute the F1 score by averaging with three different methods: (i) *micro* average, which sums the TP, FP, and FN across all classes before calculating the metric; (ii) *macro* average, which is the mean of the F1 scores of each class; and (iii) *weighted* average, which weights the F1 scores of each class by the number of samples in the dataset. For a fair assessment of the performance of the different models, we use a *flow-level* metric that operates on packets but removes the bias caused by long flows [44]: the score is calculated with packet-level TP, FP, and FN where each packet has a weight $w = 1/l$, being l the length of the flow the packet belongs to.

D. Benchmarks

We benchmark DUNE against four state-of-the-art in-switch classification solutions: Mousika [9], Flowrest [13], Netbeacon [25], and Jewel [5]. All use monolithic ML models.

Mousika is a packet-level (PL) classifier leveraging knowledge distillation. In this approach, a Random Forest (RF) or Decision Tree (DT) is trained and distilled into a single Binarized Decision Tree (BDT). We implement Mousika using the publicly available source code [45] and use our model analysis approach to run a grid search over the hyperparameters and PL features, which are later binarized and used to train the teacher RF or DT model. As the BDT student model uses a compact binary-tree representation, we need not limit its size.

Flowrest is a resource-aware, tree-based solution for flow-level (FL) inference. Thanks to its flow management strategy, it allows tracking and classifying flows directly in the data plane. Yet, during the flow feature collection phase, it is unable to classify packets; thus, it is not suitable for classifying traffic with predominantly short-lived flows. The Flowrest instance we use is based on the available source code [46].

Netbeacon is a tree-based solution capable of performing PL classification for short flows and FL classification otherwise. To this end, Netbeacon deploys 3 sets of models: an XGBoost model to predict flow sizes; a PL classifier for short flows, collided flows, and pre-FL-inference packets; and FL classifiers with variable inference points—depending on the flow length— to classify long flows. Depending on the flow-size classification result, FL features are computed and stored—for long flows—or not. Our implementation is based on the publicly available source code [47].

Jewel is a joint packet- and flow-level inference solution. As the previously discussed benchmarks, it is based on tree models. However, unlike NetBeacon, which uses multiple models, Jewel uses a single model for both PL and FL classification. Packets in a new flow are initially classified based on stateless PL features but are used to collect FL statistics. Once a predetermined n-th packet is received, FL features are used for inference. Unlike NetBeacon, Jewel employs a unified RF for both PL and FL inference: during the PL phase, the FL features are assigned constant values, and upon the arrival of the n-th packet, the model switches to FL inference since FL features become available. The FL decision on the n-th packet is then applied to all subsequent packets of the flow. We implement Jewel using the publicly available code [48].

VI. Evaluation

Accuracy. We begin by comparing the performance of distributed inference executed by DUNE with the benchmarks

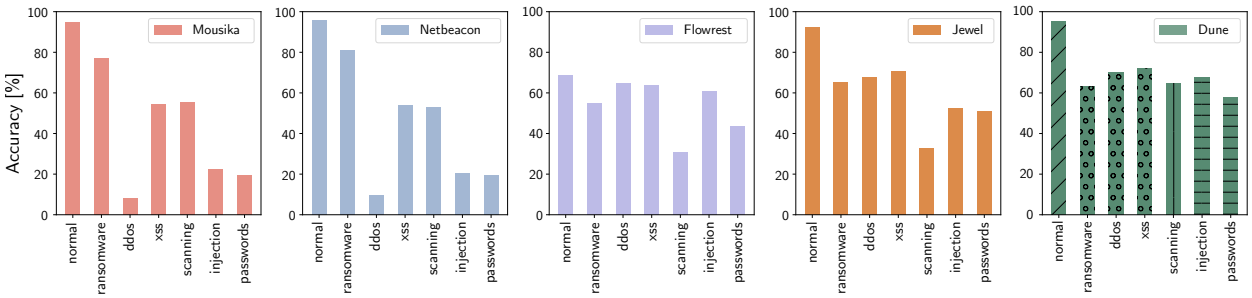


Figure 6: Accuracy per-class obtained by DUNE and the benchmarks in ToN-IoT.

Dataset	F1-Score	Mousika	Flowrest	Netbeacon	Jewel	Dune
UNSW	Macro	64.921%	40.100%	46.057%	65.718%	**70.263%**
	Micro	83.568%	41.918%	58.192%	79.132%	**84.291%**
	Weighted	**83.96%**	40.614%	59.278%	79.776%	83.296%
ToN-IoT	Macro	47.099%	55.367%	47.468%	61.718%	**67.541%**
	Micro	42.335%	62,038%	42.380%	67.045%	**73.037%**
	Weighted	37.826%	61.723%	38.024%	65.153%	**72.647%**

Table I: Performance of DUNE versus the selected benchmarks.

		Benchmarks				Dune		
Dataset	Resource	Mousika	Flowrest	Netbeacon	Jewel	SW1	SW2	SW3
UNSW	TCAM	27.80%	14.90%	36.10%	15.60%	9.40%	8.70%	8.70%
	SRAM	0.70%	10.10%	13.50%	12.40%	17.00%	13.00%	16.10%
ToN-IoT	TCAM	1.40%	14.20%	20.50%	5.60%	2.40%	5.60%	8.00%
	SRAM	5.50%	14.70%	19.50%	15.10%	12.70%	16.40%	15.90%

Table II: TCAM and SRAM usage across all models.

performing monolithic inference across three different metrics. As shown in Table I, DUNE consistently outperforms all monolithic solutions across all metrics in both use cases, with the exception of the Weighted F1 score in the UNSW dataset, where the PL solution Mousika performs slightly better. The performance improvement of DUNE, compared to the second-best model in both use cases, ranges from 0.7% to 7.5% in all metrics, with an average gain of 5.2% in the Macro F1 score. It is typically 20% or more over the worst benchmark.

Figure 6 details the accuracy per class for the different solutions. Mousika and NetBeacon exhibit a significant imbalance across classes, whereas NetBeacon and Jewel have more balanced performance yet still encounter challenges in accurately classifying certain classes compared to others. DUNE, on the other hand, shows a balanced accuracy, except for the class representing benign traffic, which is best classified by all the solutions. DUNE achieves a more homogeneous accuracy across all classes by breaking down the main inference task into dedicated sub-models, as represented with different hatches, ensuring that each class receives adequate attention.

Resource usage. Due to the constraints of programmable network hardware, evaluating resource usage is essential. Table II shows the SRAM and TCAM resource usage of all the benchmarks and DUNE in both use cases. We show the resource consumption of all the switches in which the sub-models are deployed. In UNSW, we implement two sub-models per switch, while in ToN-IoT, we implement two models in the last switch and one in the other two. In terms of SRAM consumption, Mousika is the most efficient as it does not require flow-level management. There is no general gain or loss in the consumption across switches hosting distributed models compared to FL and hybrid solutions. However, given the fact that multiple models are typically deployed per switch, the SRAM consumption of individual sub-models is lower than that of monolithic models, except for Mousika. The reason is that we utilize only one register if multiple sub-models use the same FL features in each switch. Regarding TCAM, the scarcest resource in programmable switches, DUNE shows, in UNSW, lower consumption across all switches compared to other benchmarks, despite deploying two sub-models per switch. In ToN-IoT, besides Mousika, which consumes less but brings a lower score, the TCAM usage in the switches running a single model is lower than the monolithic solutions.

Dataset	No inference	Mousika	Flowrest	Netbeacon	Jewel	Dune
UNSW	852.54	871.64	966.72	989.67	981.47	1137,70
ToN-IoT	852.54	869.18	1001.97	1004.43	1015.08	1183,61

Table III: Latency in *ns* of all models across three switches.

Latency. Finally, we calculate the end-to-end latency of the benchmarks and DUNE in all use cases across three switches, in Table VI. The *no inference* case is a baseline where packets are simply forwarded by the switches. For monolithic models, packets passing through the first switch incur both forwarding and inference latency, while packets traversing subsequent switches are subject only to forwarding latency. In DUNE, all switches combine the latency of forwarding and inference. The results show how DUNE realizes line-rate inference with a negligible added delay of around 100 ns per switch with respect to a pure forwarding operation. This delay is on par with that of solutions with a flow-management functionality.

VII. Conclusions

We propose DUNE, the first framework that automatically distributes user-plane classification models across multiple programmable network devices. Extensive experiments show that DUNE yields higher accuracy than traditional single-device approaches, with lower resource usage and comparable delay. The authors have provided public access to their code at [41].

Acknowledgments

This research was supported by the SNS JU and the European Union's Horizon Europe research and innovation program under Grant Agreement No. 101139270 (ORIGAMI). M. Fiore is Talent Attraction fellow (2023-5A/TIC-28944) and B. Bütün predoctoral fellow (PIPF-2022/COM-24867), both being co-financed by Comunidad de Madrid.

REFERENCES

[1] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *ArXiv*, vol. abs/2101.10632, 2021.

[2] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021.

[3] S. Kaur, K. Kumar, and N. Aggarwal, "A review on p4-programmable data planes: Architecture, research efforts, and future directions," *Computer Communications*, vol. 170, pp. 109–129, 2021.

[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," *IEEE Communications Surveys & Tutorials*, vol. 26, no. 2, pp. 1171–1200, 2024.

[5] A. T.-J. Akem, B. Bütün, M. Gucciardo, and M. Fiore, "Jewel: Resource-efficient joint packet and flow level inference in programmable switches," in *IEEE INFOCOM 2024*, 2024, pp. 1–10.

[6] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pForest: In-network inference with random forests," *CoRR*, vol. abs/1909.05680, 2019.

[7] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, "Taurus: A data plane architecture for per-packet ml," *ASPLOS*, 2022.

[8] G. Siracusano, S. Galea, D. Sanvito, M. Malekzadeh, H. Haddadi, G. Antichi, and R. Bifulco, "Re-architecting traffic analysis with neural network interface cards," in *NSDI*. Renton, WA: USENIX, Apr. 2022.

[9] G. Xie, Q. Li, G. Duan, J. Lin, Y. Dong, Y. Jiang, D. Zhao, and Y. Yang, "Empowering in-network classification in programmable switches by binary decision tree and knowledge distillation," *IEEE/ACM Trans. Netw.*, 2023.

[10] 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," *ACM SIGCOMM Computer Communication Review*, 2024.

[11] 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 NSDI*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 419–440. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/yan

[12] 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*, ser. HotNets '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–33.

[13] A. T.-J. Akem, M. Gucciardo, and M. Fiore, "Flowrest: Practical flow-level inference in programmable switches with random forests," in *IEEE INFOCOM 2023*, 2023.

[14] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *HotNets'17*. NY, USA: ACM, 2017.

[15] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" *NetCompute '18*, 2018.

[16] C. Zheng and N. Zilberman, "Planter: Seeding trees within switches," in *SIGCOMM '21*. NY, USA: ACM, 2021, p. 12–14.

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

[18] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello, "Programmable switches for in-networking classification," in *IEEE INFOCOM 2021*, pp. 1–10.

[19] G. Xie, Q. Li, C. Cui, P. Zhu, D. Zhao, W. Shi, Z. Qi, Y. Jiang, and X. Xiao, "Soter: Deep learning enhanced in-network attack detection based on programmable switches," in *SRDS*, 2022.

[20] H. Siddique, M. Neves, C. Kuzniar, and I. Haque, "Towards network-accelerated ML-based distributed computer vision systems," in *IEEE ICPADS*, 2021, pp. 122–129.

[21] K. Friday, E. Bou-Harb, and J. Crichigno, "A learning methodology for line-rate ransomware mitigation with p4 switches," in *Network and System Security*. Springer Nature Switzerland, 2022, pp. 120–139.

[22] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "pHeavy: Predicting heavy flows in the programmable data plane," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4353–4364, 2021.

[23] K. Friday, E. Kfoury, E. Bou-Harb, and J. Crichigno, "INC: In-network classification of botnet propagation at line rate," in *Computer Security – ESORICS 2022*. Springer International Publishing, 2022, pp. 551–569.

[24] A. T.-J. Akem, B. Bütün, M. Gucciardo, and M. Fiore, "Henna: Hierarchical machine learning inference in programmable switches," in *NativeNI 22*. ACM, 2022, p. 1–7.

[25] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, "An efficient design of intelligent network data plane," in *USENIX symposium on security*, 2023.

[26] G. Siracusano and R. Bifulco, "In-network neural networks," *CoRR*, vol. abs/1801.05731, 2018.

[27] Z. Zhao, Z. Li, Z. Song, F. Zhang, and B. Chen, "Rids: Towards advanced ids via rnn model and programmable switches co-designed approaches," in *IEEE INFOCOM 2024*, 2024, pp. 1–10.

[28] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, G. Estrada, K. Ma'ruf, F. Coras, V. Ermagan, H. Latapie, C. Cassar, J. Evans, F. Maino, J. Walrand, and A. Cabellos, "Knowledge-defined networking," *SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 3, p. 2–10, sep 2017.

[29] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling distributed machine learning with In-Network aggregation," in *18th NSDI*. USENIX, Apr. 2021, pp. 785–808.

[30] M. Seufert, K. Dietz, N. Wehner, S. Geißler, J. Schüler, M. Wolz, A. Hotho, P. Casas, T. Hoßfeld, and A. Feldmann, "Marina: Realizing ml-driven real-time network traffic monitoring at terabit scale," *IEEE Transactions on Network and Service Management*, vol. 21, no. 3, pp. 2773–2790, 2024.

[31] L. Bracciale, T. Swamy, M. Shahbaz, P. Loreti, S. Salsano, and H. Elbakoury, "The case for native multi-node in-network machine learning," in *NativeNI 22*, ser. NativeNi '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 8–13.

[32] K. Razavi, S. D. Fard, G. Karlos, V. Nigade, M. Mühlhäuser, and L. Wang, "Netnn: Neural intrusion detection system in programmable networks," in *IEEE ISCC*, 2024.

[33] H. Kim, X. Chen, J. Brassil, and J. Rexford, "Experience-driven research on programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 1, p. 10–17, 2021.

[34] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo, "Flightplan: Dataplane disaggregation and placement for p4 programs," pp. 571–592. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/sultana

[35] C. Zheng, H. Tang, M. Zang, X. Hong, A. Feng, L. Tassiulas, and N. Zilberman, "Dinc: Toward distributed in-network computing," *Proc. ACM Netw.*, vol. 1, no. CoNEXT3, nov 2023.

[36] F. Schardong, I. Nunes, and A. Schaeffer-Filho, "Nfv resource allocation: a systematic review and taxonomy of vnf forwarding graph embedding," *Computer Networks*, vol. 185, p. 107726, 2021.

[37] E. Cramer and G. Prevedello, "Per-Class Feature Importance," Aug. 2020.

[38] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[39] S. Arora and B. Barak, *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[40] S. M. Lundberg, G. G. Erion, and S. Lee, "Consistent individualized feature attribution for tree ensembles," *CoRR*, vol. abs/1802.03888, 2018.

[41] B. Bütün et al., "Dune," https://github.com/nds-group/Dune.

[42] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying IoT devices in smart environments using network traffic characteristics," *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, 2019.

[43] A. Alsaedi, N. Moustafa, Z. Tari, A. Mahmood, and A. Anwar, "TON_IoT telemetry dataset: A new generation dataset of IoT and IIoT for data-driven intrusion detection systems," 2020.

[44] M. Gucciardo, B. Bütün, A. T.-J. Akem, and M. Fiore, "Evaluating the impact of flow length on the performance of in-switch inference solutions," in *CNERT 24*. IEEE INFOCOM 2024.

[45] G. Xie et al., "Mousika," https://github.com/xgr19/Mousika.

[46] A.T-J. Akem et al., "Flowrest," https://github.com/nds-group/Flowrest.

[47] G. Zhou et al., "Netbeacon," https://github.com/IDP-code/NetBeacon.

[48] A.T-J. Akem et al., "Jewel," https://github.com/nds-group/Jewel.