





IIsy: Hybrid In-Network Classification Using Programmable Switches

Changgang Zheng , Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu ,
Shay Vargaftik , Yaniv Ben-Itzhak , Noa Zilberman , *Senior Member, IEEE*

Abstract—The soaring use of machine learning leads to increasing processing demands. As data volume keeps growing, providing classification services with good machine learning performance, high throughput, low latency, and minimal equipment overheads becomes a challenge. Offloading machine learning tasks to network switches can be a scalable solution to this problem, providing high throughput and low latency. However, network devices are resource constrained, and lack support for machine learning functionality. In this paper, we introduce IIsy - a novel mapping tool of machine learning classification models to off-the-shelf switches. Using an efficient encoding algorithm, IIsy enables fitting a range of classification models on switches, co-existing with standard switch functionality. To overcome resource constraints, IIsy adopts a hybrid approach for ensemble models, running a small model on a switch and a large model on the backend. The evaluation shows that IIsy achieves near-optimal classification results, within minimum resource overheads, and while reducing the load on the backend by 70% for data-intensive use cases.

Index Terms—In-network Computing; Machine Learning; P4; Programmable Data Planes; Software Defined Networks

I. INTRODUCTION

MACHINE Learning (ML) is increasingly applied to every aspect of our lives, leading to overwhelming processing requirements. Indeed, in data centers, ML has become a prominent workload [1]. To alleviate compute requirements and improve latency-sensitive applications’ performance, ML is pushed to the edge [2] and to end-user devices [3]. The performance requirements of ML have driven the development of a range of ML accelerators, including GPUs [4], FPGA [5], and custom ASICs [6]. While state-of-the-art accelerators can run trillions of operations per second, their network interface still limits their throughput. Network devices offer an untapped resource for scaling ML, and in particular – classification. The use of programmable network devices for in-network computing applications, such as caching [7], consensus [8] and network services [9], provides orders of magnitude throughput increase and latency reduction, combined with significant power savings [10].

Combining ML and networking is not a new trend [11], with most of the work focusing on ML on the end host. Newer

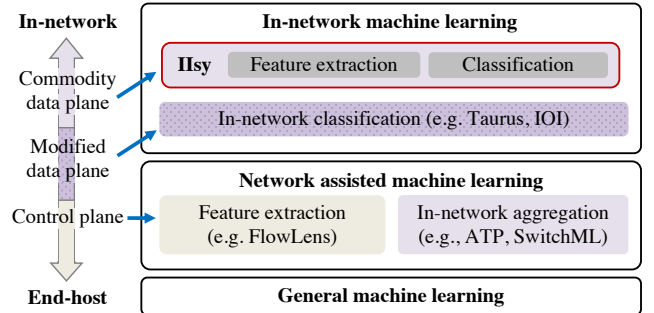


Figure 1: The positioning of IIsy

works had succeeded in creating *network-assisted machine learning*, as shown in Figure 1, using network switches either for aggregation [12], [13] or for feature extraction [14]. Despite these previous successes, running ML within network switches has proven hard to tackle. While ML accelerators usually focus on matrix multiplication [6], network switches do not support such operations. Several works have tried to address this limitation by modifying the data plane hardware or designing new hardware modules [15], [16]. These are experimental, not off-the-shelf solutions and cannot be easily and cheaply adopted.

A few attempts have been made to run ML models within the network (top of Figure 1), as further detailed in §X. Most of these works (e.g., [17], [18], [19], [20], [21] have done so using network interface cards (NICs), FPGA, or in a software environment, building upon the significant availability of resources and functionality, compared with switch-ASIC. However, as we convey in §II, such solutions lack the scalability and latency benefits of using switches. Attempts to implement ML-based classification on switch-ASIC supported only a limited number of models. These solutions also had limited size and performance (i.e., small model size with accuracy trade-off [22], [23], [24]) due to significant constraints on switch resources and functionality. To date, harnessing the power of network switch-ASIC for ML within off-the-shelf programmable switches (e.g., Tofino [25]) remains a challenge.

To attend to this problem, we present IIsy, *In-network Inference made easy*, supporting *off-the-shelf programmable switches* (e.g., Intel Tofino) to employ a range of ML classification methods. IIsy supports a range of tree-based classification models (Decision Tree (DT), Random Forest (RF), and XGBoost (XGB)) and classical models (Support Vector Machines (SVM), Naïve Bayes, and K-means), and is generalizable to other classification methods (§V). The support of

C. Zheng, R. Bensoussane, and N. Zilberman are with the Department of Engineering Science, University of Oxford, Oxford OX1 3PJ, UK.

A. Bernabeu (LS2N, CNRS, Ecole Centrale Nantes, Nantes Université, UMR 6004, France) while at the University of Oxford.

Z. Xiong, T. T Bui, and S. Kaupmees while at the University of Cambridge, Cambridge CB3 0FD, UK.

S. Vargaftik and Y. Ben-Itzhak are with VMware Research Group (by Broadcom), Tel Aviv 6997801, Israel.

ensemble models allows IIsy to overcome resource constraints through a hybrid deployment: running a small model on a switch and a large model at the backend. In this manner, IIsy achieves both high system-level performance (e.g., high throughput and low latency) and high ML performance (e.g., accuracy, F1 score).

The design of IIsy follows these guidelines:

1. **Off-the-shelf Switches.** Production-environment networks rely on programmable switch platforms and prioritize network functionality. Enabling in-network classification using existing equipment saves costs and resources and, importantly, offers platform maturity and availability without compromising functionality or performance. We describe several deployment scenarios in §II-B.
2. **Low-resource ML models.** ML models require complex mathematical operations, unsupported by switch-ASIC, or consume significant resources (e.g., Decision Tree based models). The scarcity of switch resources means that not every model mapping will be feasible. In addition, resources must be reserved for networking functionality. IIsy attends to these challenges by using lookup-based ML mapping algorithms, co-existing with standard switch functionality and parallelizing operations. We describe these in §V.
3. **ML performance.** For classification purposes, the same level of ML performance, e.g., accuracy and precision, as running on a CPU or a GPU is targeted. While this is highly desirable, it is sometimes practical to trade some accuracy for resources, e.g., saving half the memory resources while giving up 1% of accuracy. To address this challenge, IIsy supports hybrid deployments, offering competitive ML performance with low-resource in-network classification consumption, discussed in §III.
4. **Easy ML model updates.** As data changes over time and ML models need to be retrained, quick and easy deployment of updated ML models is sought. Moreover, deploying an updated classification model should be with minimal traffic disruption. To avoid disruption, IIsy supports real-time table updates using a use-case-specific (P4) program. This approach is addressed in §V.
5. **Feature extraction.** While packet-header features can be easily extracted, more complex features are needed to support many ML models. In PISA-style devices [26], this means using the parser to extract specific data from the packet and the match-action pipeline to turn this extracted data into a feature and store the information. Additionally, it is required to process data stored deep within the payload. These challenges are addressed in §VI.

In summary, this paper presents IIsy - a novel tool for mapping classification models to off-the-shelf programmable switches. IIsy is resource efficient, allows real-time ML model updates, and supports hybrid ML deployment. IIsy generates both data plane and control plane programs from the output of a common ML training framework and does not require modifications to tools, network devices, or protocols. In particular, the main contributions of this paper are:

- Introducing a mapping tool of ML classification models

to programmable network switches, supporting a range of classification methods, such as Decision Tree, Random Forest, XGBoost, SVM, Naïve Bayes, and K-means.

- Presenting an efficient mapping algorithm that is independent of the number of stages in the switch pipeline, which is critical for scaling ensemble models.
- Demonstrating feature extraction on packet, flow, aggregate, and file granularity.
- Demonstrating the use of hybrid deployments for in-network classification, consisting of a small model on a network switch and a large model over the hosts. IIsy achieves high ML performance while reducing the backend’s load and classification latency.

Our evaluation shows that IIsy supports ML-based classification within the network with minimal to no loss in accuracy (§VIII-C). Using 5 features, IIsy supports ensembles of up to 8 trees and tree depth of 10, representing more than a twofold improvement compared to related works [20], [23] (§VIII-E). Model size represents a trade-off, and we show that IIsy can support up to 50 features or 20 trees (§VIII-D). At the same time, all supported models run at line rate (6.4Tbps) with sub-microsecond latency (§VIII-F). Leveraging hybrid deployments, IIsy achieves near-optimal classification performance with over 70% of classification decisions within the switch (§VIII-G).

II. MOTIVATION

A. Benefits: the 3-Ls

The benefits of in-network classification using switches can be summarized as the 3-Ls: Location, Latency, and Load.

- **Location.** Any cloud-processed user-generated data goes through the network first. This means that any information that needs to be classified, is already processed by switches. Extending this processing to include classification is a natural step. Network switches are located at every point of the network (e.g., edge, data center, point of presence), providing early access to data as well as visibility into the aggregation of data sources. In addition, network switches are already part of the infrastructure carrying user data and do not need to be newly added. There are no cost or space overheads, beyond existing network requirements, unlike other accelerators (e.g., GPUs, middleboxes).
- **Latency.** The latency from a data-generating node to a processing node is always higher than the latency to any network device along the path between the node. Within a data center, every hop avoided through in-network classification saves hundreds of nanoseconds to microseconds [27]. In wide-area networks, propagation delay can be in the order of milliseconds, therefore, classifying next to the end-user or at the edge can significantly reduce latency. This is important for time-sensitive applications, such as financial transactions, industrial control [28], smart transportation systems, and latency-critical IoT applications [29]. As discussed later, automatically converting and loading trained ML models to (local and remote) network switches, can speed up

further the reaction to events in the network and shorten the time for detection and mitigation.

- **Load.** Network switches can process billions of transactions per second and do so while providing high power efficiency [10]. The rate of classification decisions by an end-host or an accelerator is bounded by the data rate of the attached network device. A fully realized in-network classification offers both the high throughput of a network switch and the reduction of the load on the backend. As shown in §VIII, in-network classification can significantly reduce the amount of traffic to servers, and that requires further processing. In some use cases, e.g., mitigating distributed denial of service (DDoS), dropping malicious traffic close to the source can dramatically reduce both network and server loads.

B. Deployments Scenarios

In-network classification is possible in different deployment scenarios, including (1) a native switch operation, (2) a switch acting as an endpoint accelerator, (3) smart NICs.

Switch/Router. A programmable switch used as a switch/router has some, or most, of its resources dedicated to networking operations, meaning that in-network classification needs to be resource efficient. Using in-network classification within a switch or a router does not require extra cost or space, and the power overheads are small [10]. The location of a switch/router affects its benefits; A switch/router very close to the user is most useful for data reduction, ultra-low latency applications, and mitigating the effects of distributed events (e.g., DDoS attacks). On the other hand, a switch within a data center can support more complex applications. For example, assuming that the switch is located after a load balancer, decrypted traffic can sometimes be assumed [30], enabling in-network classification in use-cases otherwise prohibited by traffic encryption.

Endpoint accelerator. The switch as an endpoint accelerator refers to using a switch purely for ML purposes. This model is already in use for some applications, such as load balancing [31]. Unlike other deployment scenarios, here the switch adds space, power, and cost overheads.

Smart NIC. Smart NICs, DPU (e.g., NVIDIA BlueField), and IPU (e.g., Intel Mount Evans) have been used in several recent in-network classification works [21], [17] due to their improved resource availability (memory, encryption modules) and flexibility of programming (e.g., FPGA or SoC based). While smart NICs benefit from the 3-Ls, it is to a lesser degree. Their throughput is lower than a switch (e.g., 400Gb/s vs 50Tb/s), end-to-end latency is higher than a switch (but lower than a CPU), and their location is further from the data source. As shown in previous works [10], there are still performance and power benefits, but sub-par to a switch ASIC.

The significant benefits of the 3-Ls lead us to focus on the *Native Switch* deployment scenario. IIsy also seamlessly supports the Endpoint Accelerator model. Our solution is applicable to smart NIC architectures, like the Portable NIC Architecture (PNA) [32], and was used for prototyping [33].

C. Limitations of In-Network Classification

While using in-network classification offers benefits such as the 3-Ls, it has some limitations. The constrained resources on programmable switches impose limitations on the size of models that can be deployed. This affects the accuracy of classification services. Even though IIsy scales in-network models beyond previous works, their size remains small compared with the very large models running on server-based setups. However, system performance demands for services, such as classification rate, are usually non-negotiable. Thereby, a solution that leverages the advantages of in-network machine learning while achieving classification performance at the server level is needed.

III. HYBRID DEPLOYMENT

The resource-constrained nature of network devices means that in-network ensemble models are smaller than full-grown ensemble models. Hence their ML performance may be sub-par. For such cases, we propose a **hybrid ML model** that can achieve close to optimum ML performance while still benefiting from the performance of in-network classification. A hybrid deployment employs a small in-network ML model on the network device and a large ML model over an endpoint. The idea of applying a hybrid model to in-network classification is inspired by techniques used in heterogeneous computing (e.g., ARM’s big.LITTLE) and from pure-ML hybrid deployments.

In many ML ensemble models, such as Random Forest and XGBoost, the ML model can provide a classification with a corresponding confidence level – the probability that the classification is correct [34]. In this paper, we adapt the ML concept of a hybrid deployment [35] by implementing a small in-network ensemble model (e.g., by limiting the number of trees in an ensemble or using a subset of features [36]), and running a large ML model at the backend.

The small models within the network are necessarily models where the training process can provide confidence scores for classification outputs, e.g. tree-based ensembles. The large model at the backend operates independently from the small models, and they do not need to be of the same type. The selection of the large model at the end-point primarily considers the processing ability of the backend system, as well as the ML performance of the large model. Advanced models, such as deep neural networks [37], and sophisticated training techniques, like adversarial training [38], can be utilized. However, due to their high performance for the studied use cases, and simplicity of analysis, we employ ensemble trees on both the small model inside the network and the large model on the backend system.

To cope with the lower ML performance of a small in-network ML model, classifications by the small model are considered valid only if their corresponding confidence is above a given (high) threshold. Invalid classifications by the small in-network ML model (i.e., confidence below the threshold) are forwarded for re-classification by the large ML model deployed at the backend. Confidence is a property of the model. The output confidence value is fixed for a given input.

Using a forest model as an example, the output confidence is the mean of the probabilities of selected output label for all the trees in the forest.

Previous pure ML works [35] have shown that most of the queries in a given dataset can be classified by a small ML model with a high confidence level. Hence, a hybrid ML deployment reduces both the classification latency and the load on the backend servers (by forwarding only “hard” queries for re-classification), as compared to a monolithic ML model deployment at the backend, where all queries are processed by the end-point. In §VIII, we validate these assumptions and demonstrate the benefits using two use cases from different domains, cyber-security, and finance.

IV. IISY ARCHITECTURE

Iisy automatically maps trained classification models to programmable network devices, and in particular to off-the-shelf switch ASIC. Iisy uses a common ML framework for training and converts its output to data plane code and control plane code.

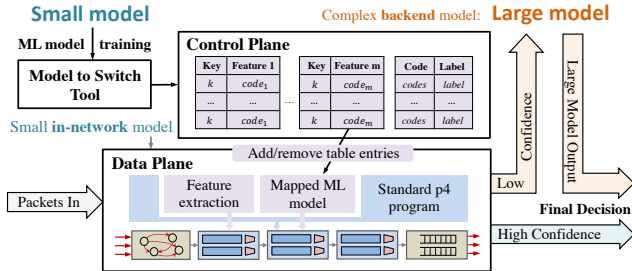


Figure 2: The high-level architecture of Iisy

The architecture of Iisy, shown in Figure 2, has four components: ML training, a mapping tool from a trained model to a target network device, a data plane implementation on a hardware target, and a control plane component for populating table entries. Additionally, the figure shows the integration with a server backend, for hybrid deployments.

Iisy uses host-based ML training based on standard ML frameworks, such as scikit-learn, and allows for model updates over time (see §VIII-I). Input datasets may be of different formats (e.g., csv, pcap traces), and the output is expected as a pickle file.

Iisy’s mapping tool takes the output of the ML framework, the trained model, and maps it to a switch-ASIC target (see §V). The tool generates two components: an implementation of the network switch data plane (P4 based), and the table entries loaded by the control plane.

The data plane components of Iisy target RMT [39] or PISA [26] based network switches, such as Intel’s Tofino and NetFPGA SUME [40]. It combines standard network switch functionality (provided by the user), and the in-network classification code generated by the mapping tool. In particular, this component of Iisy’s design provides resource-allocation optimizations.

The control plane component is responsible for the configurations and updates of the network device. As such, it combines the standard user-defined control plane and the table configurations generated by the mapping tool and required

for in-network classification. The control plane also supports runtime updates on the deployed classification model (§VIII-I).

Iisy’s support for hybrid deployment means that an additional data plane component is needed, which considers the confidence level of a classified transaction, and accordingly decides if to forward the transaction to its destination (high confidence) or route it to the backend for classification by the large model (low confidence). The confidence threshold is configurable, and confidence levels are programmed through the control plane. The type of the model on the switch and at the backend do not need to be identical, e.g., XGBoost on the switch and a neural network at the backend.

V. MAPPING MODELS TO SWITCHES

Mapping trained models to switches is challenging on multiple fronts. Not only the amount of on-switch memory is limited, but also the number of tables and stages that can be used. Additional constraints include limited mathematical operators, parsing states, metadata resources, and others.

To overcome these challenges, Iisy builds upon the following guidelines:

- Using lookup tables to implement mathematical operations, for example, multiplication and exponents.
- Optimizing on-chip resources, and reducing lookup tables size, by storing encoded results instead of explicit calculation results.
- Classification decisions and complex follow-up actions (e.g., in a hybrid deployment) use resource-efficient lookup tables instead of conditions or functions.
- Be willing to trade accuracy for resources. Users can further trade model size and ML performance.
- Breaking the dependency between model dimensions (e.g., features, trees, hyperplanes) and pipeline stages through design for parallelism.
- Optimizing the resource usage by sharing features lookup tables between models.

The methods for mapping different trained ML models to switches are described in Table I. Every method builds upon the Match-Action (M/A) pipeline [39], [26], [25] to map different, or more complex models. Specifically, in the M/A pipeline, a *key* (input value) is matched against a table, and the matched entry is associated with a result *action*. One intuitive method, shown in Table I (9), is suitable for all models. Using a single M/A table, this approach maps all input features to an output class. While this universal solution is simple, it typically leads to the explosion of table entries and stages, hindering its adoption in use cases with a large number of features or wide feature ranges. In the following, we introduce several other mapping methods, which are model-specific.

A. Decision Tree

The most basic functionality of network switches is packet classification, mapping incoming packets to output ports (i.e., classes in terms of ML). In a layer-2 Ethernet switch, the feature used for classification is the destination MAC address, and the MAC table is used to decide the output port – the classification’s result. This is analogous to a single-level

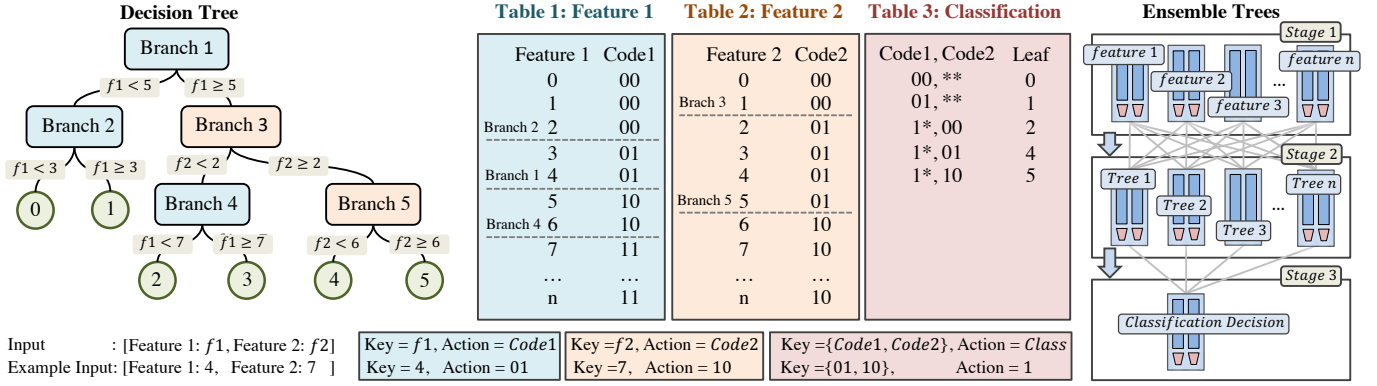


Figure 3: An example of a Decision Tree mapping, and the mapping to stages of an Ensemble Model.

decision tree. A more complex switch will consider also features such as VLAN tag, IPv4 address or flow size, creating a multi-level decision tree.

IIsy takes a more efficient approach for mapping decision trees to network devices, independent of tree depth. In the first step (equivalent to a switch stage), IIsy looks up in parallel with all input features. In the second step (equivalent to a switch stage), IIsy uses the results of the features' lookup to make a classification decision (Table I *Decision Tree*). The main idea behind this mapping is decomposing the path from the root of a tree to the leaf node into feature-based decisions, where each feature lookup indicates which path(s) is (are) taken. The classification decision lookup uses, as the lookup key, the integration of feature-based decisions.

Figure 3 shows a simple example of this mapping, using a decision tree with a depth of three, with six leaf nodes, and using two features. The color of each branch indicates the feature used, with three branches using feature 1 (f_1), and two branches using feature 2 (f_2). The branches using f_1 are mapped to Table 1, and the lookup key is the value of f_1 . The table contains 4 ranges, covering the potential outcomes of branches 1, 2, and 4. Each range is encoded as the resulting action. Similarly, Table 2 uses f_2 as the lookup key, with 3 ranges, corresponding to branches 3 and 5. The result of this table lookup is a second 2-bit code. While the tables are shown as an exact match, ternary match, longest prefix match, and range match implementations are possible. The third table, the classification table, uses as the lookup key the two resulting codes (actions) of Table 1 and Table 2. A match on this key results in a leaf node, the classification's result.

Table size analysis. Assume a tree with B branches using F features, where each feature f_i is w_i bits wide and used in b_i branches. The number of entries in a ternary feature table for f_i will be $O(b_i \times w_i)$, while an exact match table will contain all valid feature values. This number can end up small, as shown in §VIII-C, for example, if most values are mapped to a default entry, meaning that no memory is consumed.

The number of entries in the classification table depends on the depth and shape of the tree. The worst case is when the branches are evenly divided between all features, so features require the same number of (multiple) bits for the code. The

best case is where $F - 1$ features are used only once in branches, requiring a single-bit code, and the last feature is used $B - (F - 1)$ times. This can be written as:

$$2^{\lceil \log_2 \frac{B}{F} \rceil^F} \geq \text{Entries} \geq 2^{B-1 + \lceil \log_2(B-(F-1)) \rceil} \quad (1)$$

B. Ensemble Tree-Based Methods

Ensemble methods improve ML prediction results by combining multiple learning models [41]. We consider two tree-based types of ensemble methods: Bagging and Boosting. Ensemble models require three logical pipeline stages: the first to lookup features, the second to lookup in parallel the independent decisions of all the models, and the third to make a classification decision based on the decisions of the ensemble. While features are looked up only once, similar to DT (§V-A), here a feature's lookup generates multiple results — one per model.

Bagging. In bagging methods, multiple learners are used, and each learner has an equal weight (single vote) in the final decision. Each learner is trained using a different sample with replacement of the training data. We use Random Forest [42], which is built from multiple decision trees, as an example of mapping a bagging model to a network device.

The mapping of a model to a switch is oblivious to differences between bagging models in terms of training-related parameters such as sample selection and training method. The mapping is influenced only by constraints of the training outcome, e.g., selected features, number, and depth of trees.

While each decision tree in a Random Forest can be independently mapped to a device, as in §V-A, this is inefficient. For example, a Random Forest of ten trees, each using the same five features, will require fifty feature tables and eleven decision actions (one decision per tree, plus one classification for the entire forest).

IIsy significantly reduces resource requirements by sharing feature tables between trees. This means that for the previous example, IIsy will require just five feature tables instead of fifty. The result of each lookup in a feature's table is a series of action codes (as defined in §V-A), one per tree. Trees can be pruned to create action codes of feasible length. This mapping is not free; the number of entries in each feature table will increase to cover all models' branches, as well as the size of metadata required to carry action codes.

No.	Classifier	A Table per...	Key	Action	Last stage
1.	Decision Tree (1)	Feature	Feature's value	Feature's code word	Table, Decoding code words to class
2.	Decision Tree (2)	Class	All features	Vote	Logic/table, votes counting
3.	SVM (1)	Feature	Feature's value	Calculated vector	Logic, hyperplanes calculation and voting
4.	SVM (2)	Hyperplane	All features	Vote	Logic/table, votes counting
5.	Naïve Bayes (1)	Feature	Feature's value	Probability	Logic, selecting highest probability
6.	Naïve Bayes (2)	Class	All features	Probability	Logic, Probability comparison
7.	K-means (1)	Feature	Feature's value	Distance vectors	Logic, selecting smallest overall distance
8.	K-means (2)	Cluster	All features	Distance from cores	Logic, Distance comparison
9.	General Model	Model	All features	Label	Table

Table I: Different manners of implementing in-network classification within a match-action pipeline. Logic refers only to addition operations and conditions. Random Forest and XGBoost are an extension of Decision Tree.

As noted in §V-A, IIsy requires a table per tree to turn an encoded path into the decision of a tree. The classification result of the entire ensemble is based on the collective classification results of all trees in the ensemble and is implemented as a table. The decision table takes as the lookup key the classification results of all the trees and provides the classification result as an output.

Boosting. Boosting [43] and bagging methods are different, as the ensemble is built by training new learners to focus on misclassifications by previous learners. Gradient boosting, such as XGBoost, is often built from an ensemble of decision trees, where a small decision tree (e.g., with 8-32 terminal nodes) is added at each iteration and scaled by a constant factor. Then, a new tree is grown to reduce the loss (according to the loss function) of the previous trees. In boosting, new trees are trained with a focus on previous misclassifications. The decision is based on the weighted outcome of each tree.

Despite the differences in training, the mapping of a generated XGBoost model is mostly identical to a Random Forest (§V-B), using a table per feature, and a table per tree. The main difference is that leaf nodes are weighted. The weighting can be applied either when constructing the tree table, which is typically more resource-efficient, or at the decision stage. Specifically, the weight in XGBoost is converted into an equivalent codes. Different to “one vote per tree” in bagging, in boosting trees have different weights in the final vote. The weight codes are used to represent the number of votes from each tree. In the final stage, it is effective (§VIII) to aggregate the weighted results across all trees, either by summation or using a lookup table. Directly votes’ summation and comparing the summed votes for each class is easy, but more stages and logic resources. On the other hand, a lookup table enables mapping of weight codes from each tree to the corresponding output class (Figure 3), requiring only one pipeline stage but costs more memory (table entries).

C. Classical models: SVM, K-means, and Naïve Bayes

Classical classification algorithms can be mapped using approaches similar to ensemble models. In IIsy’s prototype, these are applied to SVM, K-means, and Naïve Bayes.

While previous models were indifferent to data types, classical models are sensitive to it. In IIsy, integers and fixed-point numbers are seamlessly supported. Fixed point numbers are either normalized (e.g., multiplied by a factor to achieve

a natural number) or quantized. The use of quantization is common for all data types as a means of table compression, where the number of bins is relative to the number of bits in the lookup’s result (action).

The first classical model’s mapping approach uses a lookup table per feature, and the results are encoded. For example, it might indicate a normalized value. If the result of a lookup is a code, the second stage in the pipeline will use a lookup table with the codes of all features as the key, similar to decision trees. If the result of a lookup is a value, then the last stage in the pipeline will operate on all values, typically adding them up and comparing the results across classes (for example Table I SVM (1)).

A second mapping approach, which is not always feasible, holds a table per class or class indicator. For example, in Table I SVM (2), there will be a table per hyperplane. The lookup keys are the values of all the features. The result of the lookup will be an indicator, such as if the entry belongs within or outside the hyperplane (for SVM) or the distance from a center of a cluster (for example, Table I K-means (2)). This approach is theoretically applicable to a wide range of algorithms, including *Decision Tree* (2) in Table I.

Based on these two approaches, IIsy provides mapping solutions for SVM, K-means, and Naïve Bayes:

SVM. Support Vector Machines (SVM) uses hyperplanes to separate between classes, where the output of the training stage is the equations of the hyperplanes, such as:

$$\begin{cases} a_1x_1 + b_1x_2 + \dots z_1x_n + d_1 = 0 \\ \dots \\ a_mx_1 + b_kx_2 + \dots z_mx_n + d_m = 0 \end{cases}$$

where x_i is the value of feature i , n is the number of features, k is the number of classes and $m = k * (k - 1)/2$. There are two ways to map SVM to a network device. First, to hold a table per feature, and second, to hold a table per hyperplane.

A table per feature means that the key to a table is the feature’s value (Table I SVM (1)), and the output of the lookup is a vector of calculated values $a_i \times x_i$, where x_i is the value of the feature (potentially normalized or quantized). The value of an SVM hyperplane, separating two classes, is calculated as the sum of vectors from all feature tables, and a decision is taken. This can be optimized by adding up the features in each pipeline stage.

A table per hyperplane means that m lookup tables are used, one per hyperplane, and the outcome of the lookup indicates on which side of a hyperplane is a given input (Table I SVM (2)). The key to a table is a set of features, and the action is a “vote”. A “vote” is a one-bit value mapped to the metadata bus that indicates if the input belongs within or outside a hyperplane. The “votes” from all m tables are counted in the last stage, and the class with the highest count of “votes” is the classification result.

The table per hyperplane approach is feasible only when the concatenation of all features does not lead to a too-wide key (same when using each feature code as a separate key). If the features used are, for example, source and destination port, protocol, and some IP flags, the key will be relatively small, and the solution will be feasible. Theoretically, the use of all features can yield the classification within a single table. However, this table is likely to be very large and less resource-efficient than distributing across a few smaller tables.

The main advantage of the table per hyperplane approach is that there is no unintentional loss of accuracy; the output of the table is a vote, not a value. It is possible to purposely lose some accuracy, e.g., if one wants to reduce the number of table entries by merging multiple ranges of different “votes” into a single entry (e.g., if keys 0-1111 and 1113-32767 are mapped to class 1, and key 1112 to class 2). In contrast, a solution using a table per feature may lose some accuracy, as the result of a lookup is a calculated value (and not a code), which has an accuracy limited by its number of bits. The final classification decision may not be affected by the loss of accuracy in calculations along the pipe, but this is not guaranteed.

Using a table per feature will be favored in some cases, e.g., if working with eight features, each of eight-bit, so each table is only (and at most) 256 entries deep, and features can be looked up in parallel. The table per hyperplane equivalent will be multiple (m) tables of a 64-bit key (or eight eight-bit keys).

K-means. An example of unsupervised learning mapped to a network device uses K-means clustering. In K-means, k classes are represented by k centers of clusters, with each center defined by n coordinate values, one per feature. A data point will be mapped to a class based on its nearest center of a cluster. The distance from cluster i is denoted by:

$$D_i = \sqrt{(x_1 - c_1^i)^2 + (x_2 - c_2^i)^2 + \dots + (x_n - c_n^i)^2}$$

where x_1 to x_n are the values of the data point’s features. Obviously, to find the nearest cluster, it is sufficient to consider the square distances. As in previous examples, there are two ways to map the model to a network device.

One option is using a table per feature (Table I K-means (1)), with the lookup’s result of table i being a vector of $\{(x_i - c_1^i)^2, (x_i - c_2^i)^2, \dots, (x_i - c_n^i)^2\}$. Here, the last stage will need to sum up all D_i and find the smallest one¹. As before, the challenges here are the accuracy of the calculation and the required width of the metadata bus.

The second approach uses a table per class (Table I K-means (2)), with multiple keys of all features (a feature per key) or one key constructed by the concatenation of all features (presenting a challenge of key width). The result of each such table lookup is the distance of the data point from the center of the cluster. As proposed above, this distance can be represented by an integer value across all tables, allowing an easy comparison and selection at the last stage.

Naïve Bayes. For a Naïve Bayes classifier [44], we assume a Gaussian distribution of independent features [45]. Similar concepts apply to related methods, such as kernel estimation [11]. Under this assumption, the likelihood of feature x_i is expressed as:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

And the classification rule is:

$$\hat{y} = \arg \max_y P(y) \prod_{i=1}^n P(x_i|y) \quad (2)$$

If there are n features and k classes, there are $k \times n$ pairs of (μ_y, σ_y) .

A mapping based on a table per feature is possible but can be both inefficient and inaccurate. Here, the result of each feature-value lookup will be a vector of probabilities (Table I Naïve Bayes (1)). As the number of bits per vector is limited, there will be some accuracy loss. Even if the target allows for any vector length and a fixed point notation is used, the amount of metadata that needs to be carried between stages will be larger than other solutions, and depending on the number of features and classes, exceeding allowed resources. In this approach, each class will require a table at the end of the pipeline to calculate its overall probability, bringing the overall number of tables required to $O(n+k)$. Unless there is a compromise on accuracy, the number of entries in each such table will be large, as the keys are all probabilities per class (or the concatenation of all probabilities). Finally, at the end of the pipeline, a comparison is required to find $\max_y P(y)$.

The second approach is to use one table per class, with all the features as the key, and with the result being the probability of that class (Table I Naïve Bayes (2)). The disadvantage is the size of the required table: it uses a very wide key (a form of a concatenation of all input feature values) or too many keys (each feature used as one key), and its depth is proportional to this unless a compromise is made for accuracy. The resulting probability does not need to be presented as a fraction, and an integer value can be used that symbolizes the probability. As long as the same notation is used across all tables, the final comparison of $P(y)$ and the classification result will be correct.

Table size analysis. Our experience shows that the first approach, single-feature-as-a-key, shown in Table I (1, 3, 5, 7) provides (relatively) shallow tables, proportional to the number of classes. Assume a use case with n input

¹Values can also be summed up in each stage.

features, with range $(0, f_{max}^i]$ for a given feature i . Single-feature-as-a-key requires $\sum_{i=0}^n f_{max}^i$ table entries and $n + k$ logical stages, where k is the number of stages required for the final logic (usually 1-2 stages). Models using this approach may experience accuracy loss and higher stage consumption, as explored in §VIII-C. The second approach, all-features-as-a-key, shown in Table I (2, 4, 6, 8), is feasible only when the use of multiple features allows for a feasible key size and table depth. Under the same use case assumption with m classes (e.g., hyperplane in SVM), this approach requires $m \prod_{i=0}^n f_{max}^i$ table entries with $m + k$ logical stages. It provides higher accuracy results and requires fewer operations at the last stage, but costs more memory (§VIII-C).

Clearly, no single solution fits all use cases. The approach fits some classification, regression, and clustering models. Iterative models are less suitable, though they may be feasible, e.g., using recirculation. Importantly, IIsy takes care that the type of feature or its range will not affect the accuracy of the classification (e.g., through normalization).

D. Training and Mapping

The aforementioned ML algorithms are trained offline on a server using established frameworks like *scikit-learn* [46]. The trained models are then mapped to the data plane, before being deployed. IIsy maps the trained model into three distinct components: 1. data plane code, 2. M/A table entries, and 3. control plane code. The data plane code is in P4 using the mappings previously described (Table I). The features of the trained model are extracted in the parser. A template is used for every mapping, where the number of M/A tables and logical operations, as well as lookup keys are taken from the trained model and its configured hyperparameters. M/A table entries are directly derived from model parameters and weights. Control plane code is used to load M/A table entries into the data plane model. As mapping implementation depends on model selection and mapping approach, each mapping is using a template. Further automation work is described in [47]. IIsy does not guarantee that a model would fit on a switch, as some models are too large.

E. Retraining and Updates

ML models often need to be retrained, e.g., due to data skew, and the resulting classification model needs to be updated. IIsy enables such updates using only table updates, without changes to the deployed program.

A IIsy-generated P4 program depends on a set of user definitions: the features that need to be extracted (not necessarily used by the ML model), the type of the model, and constraints on the model (e.g., number of trees). While retraining will result in a different ML model, as long as the definitions above are kept the P4 program will not change. Changes to actions in the features table, and in different code-to-classification entries in the tree and decision tables (as in Figure 3) manifest as table entries changes rather than changes to the P4 code. These can be loaded through table updates, a common management operation. Changes to

hyperparameter definitions requires generating new P4 code and is not supported in runtime.

In a hybrid deployment, traffic can be directed to the backend during updates, to avoid misclassification. Data for retraining can be collected through sampling and using in-network telemetry [9], and will be affected by the location of a switch (e.g., edge vs data center), similar to [15].

VI. FEATURE EXTRACTION

Network devices are designed to extract headers from packets. However, the research community has already gone beyond packet headers for applications ranging from telemetry [9] to in-network computing [10]. In IIsy, the features extracted are selected based on ML importance score. The number of features is constrained by the number of stages in the parser [39] and memory, and scales to tens of features (§VIII-D).

Packet level features. Extracting packet-level features is native to network devices. Packet header extraction is done in the parser, and features are stateless. Such features include, for example, protocol type or source and destination port number. Packet level features also refer to features that describe the packet, such as packet size, switch source port, or timestamp

Flow level features. Flow-level features, such as flow size and flow duration, are stateful. Information is collected and stored across multiple packets [48]. IIsy supports two types of flow-level features: counted features (e.g., flow size, packets count), and time-related features (e.g., flow’s start time, inter-packet gap).

Aggregate level features. Aggregate level features consider a group of flows, the aggregation of traffic (e.g., from/to port X) or the network as a whole. Examples of features useful for ML purposes include traffic volume from a group of subnets, inter-arrival time toward a specific application or a histogram of source and destination ports [49]. Implementing aggregate-level features is mostly similar to flow-level features, however, additional operations may be required, such as mapping flow identifiers to an aggregated-feature identifier.

Supporting file-level features. Supporting features extraction from files is more complex than in previous cases. We distinguish between four stages of file processing:

1. Start of a file. Where the file header needs to be processed, and initial resources need to be assigned. This is similar to a start of a flow but with a more complex parsing of the header.
2. Looking into the file’s payload. If packet size exceeds the width of the programmable data plane’s bus, then it may require recirculation (target dependent).
3. Examining payload across packets. As a file is likely to be broken across many packets, extracting features from a file means that contents at the end of a previous packet

need to be stitched with the contents at the head of the next packet.

4. End of file. This is similar to the end of a flow and may allow to free up some resources.

In IIsy it is feasible to extract data from a subset of file types, not from all file types. Text-based files, such as `txt`, `xml`, `html`, and `csv`, are straightforward to process. File types that use many objects (e.g., `docx`, `pdf`), have a complex file structure (e.g., `mp3`, `png`), or are composed of many elements (e.g., videos composed of frames) are very hard to process due to the complex structure and required resources. Extracting a feature from the `jpeg` file, such as the average value or the value of a certain pixel, is possible. However, extracting more complex features is typically beyond the resource budget [50].

IIsy’s file processing makes a few assumptions. First, we assume that files are not encrypted (as in some deployment scenarios described in §II-B). Second, while both TCP and UDP are feasible, no packet reordering is currently supported, e.g., a direct-attached network device. Third, we assume file-type-specific feature extraction. Last, privacy and legal rights to process the data need to be addressed by the operator. Going beyond these assumptions is future work.

IIsy’s contribution focuses on two file processing challenges: looking within the packet’s payload and examining the payload across packets. For deep payload inspection, we observe that feature extraction is not limited by the number of consecutive bytes in a header, but by Packet Header Vector (PHV) bit-width. It is possible to skip bytes within the packet in order to extract information, without wasting PHV resources. This enables IIsy to support line-rate features extraction as long as the total size (in Bytes) of extracted features is less than PHV size. To extract more features, a packet requires recirculation, with processed bytes being stripped from the packet. As recirculation has negative performance implications, limiting the number of features may be preferred over performance loss.

Examining payload across packets requires saving features data, done using registers. When a feature is split across two packets, the first part is stored in a register, and then matched to the second part either using recirculation (matching both parts of the feature in the right pipeline stage) or in a second pipeline, such as the egress or folded pipeline.

There is a trade-off in functionality, performance, and resource efficiency when applied to specific file-level use cases. As previous works have suggested [24], an easy getaway is to manipulate the file sent at the host’s side before entering the network so that some challenges can be avoided.

VII. IMPLEMENTATION

IIsy’s framework uses four components (ML trainer/converter, data plane implementation, control plane component, and backend large model), as described in §IV. In this section, we describe the implementation of our prototype.

The prototype’s ML training framework is based on *scikit-learn* [51]. The implementation enables fast development and prototyping of different models and, in particular, the hybrid

approach. The training of the hybrid models used *scikit-learn* 0.24.1 and *XGBoost* 1.3.3, running over a `c4.8xlarge` AWS EC2 instance with 36 vCPUs and 60 GB RAM running Ubuntu 16.04 LTS.

The switch implementation runs on two platforms: Intel’s Barefoot Tofino (ASIC), and NetFPGA-SUME [40] (FPGA). All the models are mapped to both targets, except for boosting, which targets only Tofino. The NetFPGA implementation enabled exploring the limits of feature extraction. This includes complex stateful features (§VI), such as jitter, inter-arrival time, and data rate. On Tofino, packet-level, flow and aggregate features are supported, with a further focus on files. Data is extracted from text files, both with fixed and unknown feature sizes (e.g., words separated by delimiters). The prototype supports features of up to 15 ASCII characters (32-bit). In addition, it supports features split between packets and features implemented deep within the packet (§VIII-A2).

The data plane and the control plane are auto-generated, using python scripts and a configuration file. A user defines in a configuration file design constraints, such as the maximum number of trees, and the tool takes the output of the training stage (pickle file) and uses it to generate both the data plane (P4 files) and the control plane (table entries). Further information is provided in [47].

The system test environment uses $64 \times 100G$ ports Barefoot Tofino. P4-NetFPGA [52] is used for FPGA development. Four servers with 100G NVIDIA ConnectX-5 NICs are used to send and receive traffic from the switch. To test full throughput, we use a snake configuration, where traffic is looped from each port to the following one, enabling traffic across all 64 ports, which is a common practice [8]. As a baseline, we measure 6.4Tbps on the switch when running simple forwarding.

VIII. EVALUATION

In this section, we evaluate in-network classification for feasibility, performance, resource consumption and ML performance. For brevity, this section focuses on Intel Tofino, and further details of NetFPGA are provided in [33], [53].

A. Use cases

Our evaluation is driven by two use cases: network anomaly detection using the UNSW-NB15 dataset [54], and time sensitive financial market prediction using the Jane Street Market Prediction dataset [55]. For each of these use cases, described below, we explore the classification performance of a standalone switch, as well as part of a hybrid model V-B.

1) *Anomaly detection - Reducing backend resource consumption*: Anomaly detection, such as intrusion detection and prevention, is typically done at the backend and can consume significant compute or acceleration resources [56]. All network traffic toward certain application servers needs to be examined, and malicious traffic needs to be filtered. Our goal is to provide a scalable solution, whereby normal traffic is admitted by the switch, and anomaly traffic is either dropped in the switch or sent to the backend (in a hybrid mode). In the hybrid mode evaluation, any traffic that is classified as anomalous

or with low confidence is sent to the backend for deeper inspection. In this manner, the switch does not block (drop) legitimate traffic and offloads significant processing from the backend, as most traffic is normal. The dataset used, UNSW-NB15 [54], contains a mix of normal traffic and different types of attacks. This use-case is focused on *load* benefits, where in-network classification saves resources compared with host-based solutions while also scaling with the network’s bandwidth.

From ML perspective, Random Forest is the most suitable for this use-case, as it offers low variance in its classifications. This leads to a more predictable fraction of the traffic that is correctly classified as normal (unless the traffic distribution changes dramatically – which requires retraining the model). Other ML models are evaluated for feasibility purposes.

Our learning uses 80% of the data for training and 20% for testing. The model running on the backend is using a Random Forest of 200 trees (estimators) and 10,000 leaf nodes (at most), and all the features in the dataset.

2) *Financial market prediction - Reducing latency*: Low latency financial transactions, such as algorithmic trading, are very sensitive to latency. For top 10% financial traders, the decision latency is less than 42 microseconds [57] from a passive order to an active transaction.

In algorithmic trading, a data feed from the stock market provides live information using an unencrypted protocol, such as NASDAQ ITCH. Typically, a large backend is used to provide real-time classification for all market transactions. In this use case, the switch can identify and tag high-priority transactions, while other transactions are sent to the backend for fine-grain classification. The tagged high-priority and high-confidence transactions can be forwarded to a different server for immediate execution. Moreover, tagged queries can be prioritized over dedicated link(s), avoiding congestion. Assigning time-sensitive high-priority and high-confidence transactions to a special fast processing path may bring significant financial benefits with low resource consumption. Any misclassified high-priority transactions will simply undergo the regular classification path. This is an example of *latency* benefits for time-sensitive applications, while the change to the backend’s load is small.

To demonstrate this use-case, the Jane Street Market Prediction dataset [55] is used. Each entry in the dataset contains 130 anonymized features, representing real market data, and two output values (‘weight’ and ‘resp’) representing the trade’s return. using these two output values, we label the transactions by recommended actions: ‘Strong sell or buy’, and ‘Sell/Hold/Buy’. The transactions are typically a feed of individual trade instructions from the stock exchange. The Jane Street dataset is a recent and open information available from a trading company, presenting pre-processed transactions.

Our goal is to minimize the latency experienced by transactions marked as “strong buy or sell” (accounts for $\approx 13.1\%$ of the total transactions). All incoming transactions are assumed to go through the switch, so any classification by the switch has an additive latency of close to zero². All low-confidence

Model	SVM	Bayes	KMeans	DT	RF	XGB
Tables	6	6	4	5	11	11
Memory	5.37%	9.22%	9.12 %	1.11%	1.89%	6.68%
Stages	8	8	7	2	3	4
Latency	30.37%	31.11%	23.33%	28.52%	35.56%	35.93%
Accuracy	92.14%	87.92%	52.41%	88.69%	88.91%	88.88%
Acc. sbytes	91.78%	86.93%	87.36%	97.04%	97.05%	96.83%

Table II: Anomaly Detection - Latency on Tofino relative to switch.p4 reference program. The row named accuracy (Acc.) sbytes shows the IIsy performance of using flow-level features ‘sbytes’.

Model	SVM	Bayes	KMeans	DT	RF	XGB
Tables	6	6	4	5	11	11
Memory	1.15%	1.15%	1.04%	1.11%	2.00%	6.68%
Stages	8	7	6	2	3	3
Latency	30.37%	23.33%	30.00%	27.78%	34.81%	34.81%
Accuracy	72.08%	71.92%	70.35%	72.43%	72.44%	72.47%

Table III: Financial Transactions - Latency on Tofino relative to switch.p4 reference program.

packets are forwarded to the backend.

In terms of ML performance, while we evaluate with different models, the preference for this use case is XGBoost, commonly used in financial applications as boosting offers a controlled bias that is more suitable for identifying minorities.

Our learning uses 80% of the dataset for training and 20% for testing. The model running on the backend is using all 130 features, with XGBoost of 100 trees (estimators) and a maximum depth of 8 (XGBoost trees tend to be shallow).

B. Feature Extraction

In the anomaly detection use case, the Tofino implementation supports packet-level features (e.g., source and destination port, protocol, service, and ports equivalence) and flow-level features (e.g., duration, flow size in bytes, and packets in each direction). Flow level features sometimes improve the quality of the prediction, but cost two stages: to hash the flow ID, and to update a register holding the value of the feature (e.g. flow size). Choosing between the two options requires weighting also other considerations, such as if flow ID is needed for “standard” networking purposes. Our resource consumption evaluation (Table II) uses source and destination port, protocol, and service features, and the study of hybrid deployment (Table IV) uses in addition the feature `is_sm_ips_ports` (same source and destination) and the stateful feature “source bytes”(sbytes). By swapping feature ‘service’ with flow-level feature ‘sbytes’, the effect of flow-level features on accuracy is shown in Table II accuracy (Acc.) of sbytes row.

The Jane Street dataset contains 130 numerical features, which we evaluate twice: using packets containing numerical values, and in a csv format. For ease of exploration, the dataset is reformatted as columns of eight characters, yet other IIsy implementations are not of fixed size or known delimiter location. The features ranked most important and used are features number 42, 43, 45, 124, and 126. Both numerical and csv

²Use of L1 switches, e.g., Cisco Nexus 3550 is a different scenario.

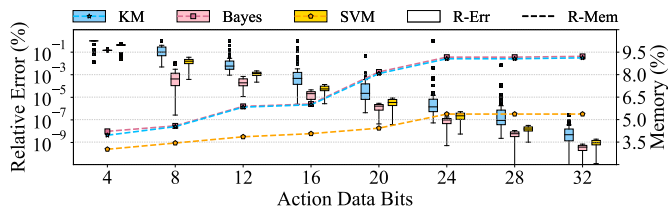


Figure 4: Calculation error in SVM, Bayes and K-means.

format processing use these features, thereby demonstrating feature extraction from deep within the packet, successfully extracting without recirculation.

As financial transactions are typically a feed of individual trade instructions (§VIII-A2), and the size of an entry in the Jane street dataset, with 130 columns, barely fits within an MTU packet (1522B), each transaction is sent individually.

C. Resource Consumption

IIsy aims to maximize the performance of ML prediction, while still fitting the design within the switch ingress pipeline. This section explores the resource requirements of different mapping methods.

We start by comparing two types of mapping methods of classical ML models: using a single feature as a key to a table and using all features as a key to a table. Both methods are evaluated on two datasets, Iris dataset (a small dataset of flower classification) [58] and UNSW dataset (a larger dataset related to anomaly detection) [54]. For the small Iris dataset, the single-feature-as-a-key approach (Table I (1, 3, 5, 7)) requires around 220 table entries and 2-8 stages while the all-features-as-a-key approach (Table I (2, 4, 6, 8)) requires around 6×10^6 table entries and 1-3 stages. In UNSW anomaly detection use case, the single-feature-as-a-key approach requires around 2×10^5 table entries and 2-9 stages. The all-features-as-a-key approach requires an infeasible number of table entries (about 4×10^{13}), which results in the explosion in table entries and stages, as a stage can support a limited number of table entries). Consequently, even though the all-features-as-a-key approach requires less stages on small datasets, it is infeasible for both anomaly detection and finance use cases. Therefore, in the following evaluations, the single-feature-as-a-key mapping approach in Table I (1, 3, 5, 7) is applied to each model.

Table II and Table III summarize the resource consumption of anomaly detection and financial transactions, respectively. The tables show, for each model, the size of the model that fits within Tofino’s ingress pipeline using the features noted above. The ensemble models use a small model of 6 trees with a depth of 4 (As in §VIII-G, see §VIII-D for scalability).

In the table, the memory indicates the proportion of overall utilization, while the latency is assessed relative to Tofino’s switch.p4 reference design (due to NDA). *switch.p4* is an L2/L3 switch program for Tofino, commonly used as a reference design, including 10 network functions such as load balancing, tunneling, firewall, and statistics.

As the results show, the memory requirements are low in comparison with switch.p4. For anomaly detection, all the models consume less than 9.3% of the memory, with DT and RF requiring less than 1.9%. In the financial use case, all the

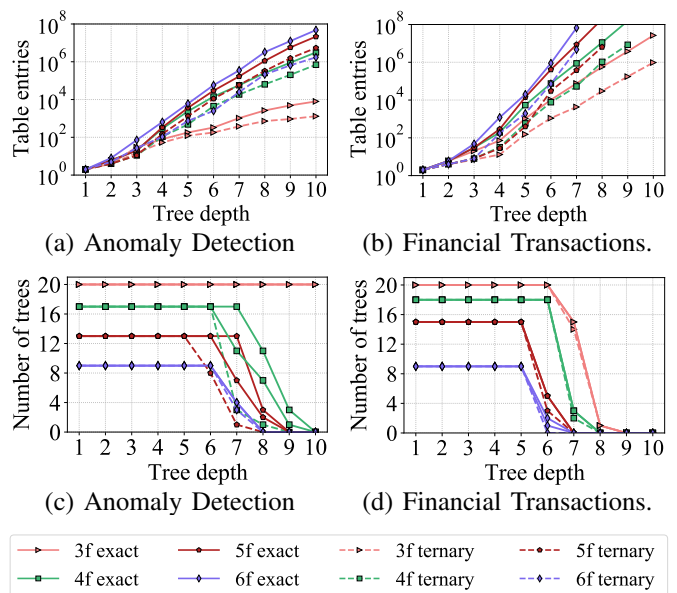


Figure 5: Ensemble scaling of table entries (a,b) and maximum number of trees (c,d) with tree depth and features.

models require less than 6.7% of memory. This demonstrates the efficiency of our mapping algorithm. As shown, a decision tree requires only 2 stages, and an ensemble requires 3-4 stages, smaller than the number of trees.

The implementation of classical ML models may introduce an error. This error is studied for both calculation error and classification error. The calculation error, shown in Figure 4 for the anomaly detection use case, is the relative error of a result calculated on a switch (e.g., hyperplane equation in SVM), compared with the same equation calculated on a server. While this error is small (less than 0.001%), the more important result is the misclassification error due to calculation error: zero for SVM and K-means, and 0.00003% for Naïve Bayes when using action width of 16 bit. This error is due to extremely low probabilities, and can be eliminated by encoding the results of Naïve Bayes calculations, rather than normalizing values. As shown in Figure 4, increasing the number of bits in an action has a minor effect on memory consumption, but can significantly reduce calculation errors.

D. Scalability

The size of a model fitting within a switch depends on the type of model, the dataset, and its features. This is demonstrated in Figure 5 (a) & (b), showing how memory requirements of a decision tree scale with the number of features and the depth of the tree, using exact match or ternary feature tables. In the finance use case, all the features are similar, and adding features increases memory requirements in a roughly consistent manner. In the anomaly detection use case, features vary significantly in their memory requirement. For example, the protocol type requires significantly fewer entries than the source or destination port. Consequently, the anomaly detection use case requires less memory than the finance use case for the same model size. As Figures 5 (c) & (d) show, using up to 6 features, one can fit up to 20 trees.

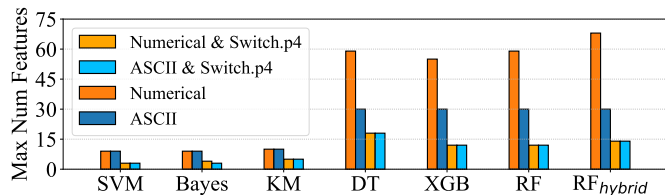


Figure 6: The maximum number of features that can fit on a switch in the financial transactions use case.

Increasing tree depth means that fewer trees can fit within the switch, due to the size of the decision table.

To explore the maximum number of features that can be supported, four types of implementations are evaluated: in-network classification using numerical features, in-network classification using ASCII (from `CSV`) features, in-network classification integrated with `switch.p4` and using numerical features, and in-network classification integrated with `switch.p4` and using ASCII features. This is applied to the financial use case, supporting both types of features.

Figure 6 shows the maximum number of features feasible under the four variations. Tree-based models can fit more features compared to classic models due to stage sharing. For example, decision tree and random forest can fit up to 59 numerical features or 30 ASCII features (due to PHV size), while XGBoost fits 55 numerical features and 30 ASCII. Classical models support 9-10 numerical or ASCII features and are limited by the number of stages required for logical operations. The integration with `switch.p4` limits the resources available for feature tables, leading to 12-18 features allowed for tree-based models.

Table IV explores the effect of ensemble size on ML performance, showing both native switch deployment and hybrid-deployment. The results are compared to fully-grown ensemble models running on a backend (§VIII-A). As the table shows, the size of a model has a limited effect on its ML performance (except for small RF in anomaly detection), and negligible effect in a hybrid deployment. Furthermore, the results of the hybrid deployment are almost identical to the baseline, showing that a hybrid deployment using a small model allows for high ML performance and little resources. We report [47] similar quality of ML performance for medium-size models also with other attack-detection datasets [59], [60], [61] and other UNSW-based research [62].

E. State-of-the-art Comparison

We compare IIsy’s lookup-based mapping with two state-of-the-art in-network classification works, Clustreams [24] and SwitchTree [20], under the finance use case. Clustreams was originally implemented on Spectrum 3, and uses a tree data structure that encodes a 2 dimensional workspace. SwitchTree was originally implemented on bmv2 and uses a pipeline stage for every level of a tree.

As shown in Figure 7, the IIsy’s K-means algorithm requires significantly less memory than Clustreams, for the same accuracy, where model depth is the parameter used to adjust Clustream’s accuracy. Figure 7 (b) shows that IIsy’s DT implementation requires 8 stages less than SwitchTree, with

Anomaly Detection, Random Forest, confidence threshold 0.7

	Small	Medium	Large	Baseline
Features	4	5	6	25
Trees	6	10	14	200
Max Depth	4	5	6	—
Accuracy	97.05	97.17	97.78	99.51
Precision	98.06	98.12	98.60	99.67
Recall	88.55	89.04	91.36	99.75
F1 score	92.60	92.94	94.58	98.88
Hybrid Accuracy	98.58	98.94	99.31	—
Hybrid F1	96.64	97.53	98.41	—

Financial Market Prediction, XGBoost, confidence threshold 0.7

	Small	Medium	Large	Baseline
Features	4	5	6	130
Trees	6	10	14	200
Max Depth	4	5	6	—
Accuracy	72.48	72.65	73.73	77.34
Precision	68.48	68.76	70.05	74.43
Recall	66.51	65.69	68.09	72.76
F1 score	67.16	65.51	68.78	73.43
Hybrid Accuracy	77.31	77.30	77.26	—
Hybrid F1	73.41	73.43	73.40	—

Table IV: Ensemble models scalability and ML performance.

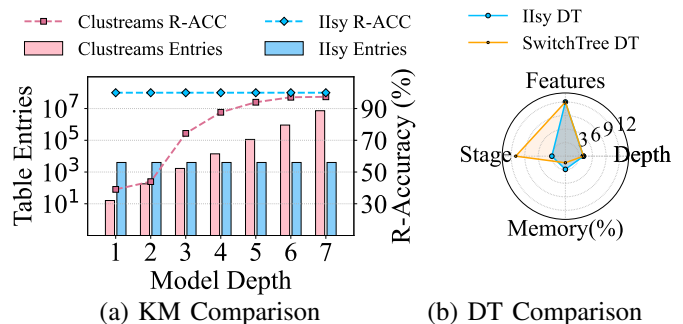
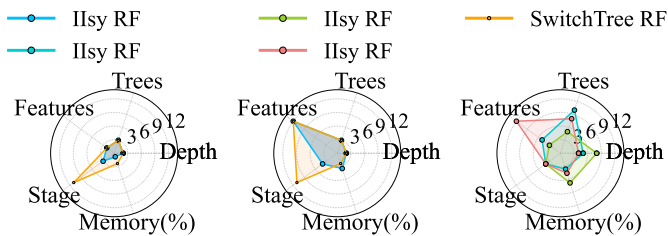


Figure 7: Comparison of (a) Table entries and Relative accuracy (R-ACC) in IIsy and Clustreams KM. (b) Stages and Memory in IIsy and SwitchTree DT. The scale (3, 6, 9, 12) applies to all evaluation parameters (Features, Stages, Depth, Memory(%)).

only 3% memory overhead. Using a Random Forest and for a small model size (3 trees, depth of 2), as shown in Figure 8 (a), IIsy requires only 3 stages and 0.82% memory, compared with 11 stages and 2.4% memory in SwitchTree. The maximum size of a SwitchTree RF model fitting on Tofino (Figure 8 (b)) is again 3 trees and a depth of 2, but using 12 features. For the same model, IIsy requires 1.5% more memory but uses only 4 stages, 7 stages less than SwitchTree. Figure 8 (c) shows combinations of hyperparameters settings of IIsy, all too big for SwitchTree, and their resource consumption, including (blue) 10 trees, 5 features, depth of 5 consume 4 stages and 3.7% memory. (green) 5 trees of depth 8 and 3 features require 4 stages and 6.9% memory, and (red) 8 trees with a depth of 4 and 12 features use 4 stages and 4.6% memory.

IIsy outperforms other reported scalability results of ensemble models (See §X). pForest [23] reported a maximum depth of 4 on Tofino. NetBeacon [63] used two trees of depth 9 or a single tree of depth 10. In comparison, IIsy fits on Tofino



(a) Small Model (b) Large Model (c) Large IIsy

Figure 8: Comparison of Stages and Memory in IIsy RF and SwitchTree RF, for (a) small model (b) maximum SwitchTree (c) Maximum IIsy. The scale (3, 6, 9, 12) applies to all evaluation parameters (Trees, Features, Stages, Depth, Memory(%)).

model depth of up to 15. The maximum number of trees currently supported is 20, with up to 55 features, though there is a trade-off between these parameters as explored above.

F. Throughput and Latency

IIsy is designed for line rate operation, without recirculations or packet drops. All evaluated programs meet Tofino’s timing for a minimum packet size (128 bytes). Our throughput and latency tests use UNSW’s pcap traces [54], and the Jane Street dataset converted to pcap and sent over UDP. The throughput is compared with a simple layer 2 forwarding program. In all cases, there are no packet drops on any of the 64 switch ports. The performance is compared with the classification of the datasets on a server using scikit-learn, without the additional host packet processing overheads (which favors the host performance). Note that ensemble models are commonly classified on CPUs [64].

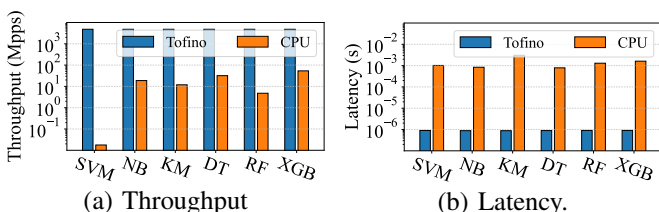


Figure 9: Throughput (anomaly detection use case) and latency (finance use case) of ML algorithms on Tofino and CPU.

As shown in Figure 9 (a), the switch implementation achieves $25\times$ (XGBoost) to $80,000\times$ (SVM) throughput improvement compared with classifying on a CPU. The latency of classifying on the switch, as shown in Figure 9 (b), is $\times 10^2$ to $\times 10^3$ better than on a CPU. Compared with the latency of current financial trading systems [57] IIsy can save at least an order of magnitude in latency.

For reference, Tables II and III report the latency of the use cases relative to the reference switch.p4 program. This latency is sub-microsecond.

G. ML performance

In this section, we explore IIsy’s ML performance, with a focus on ensemble models in a hybrid deployment. Although SVM and Naïve Bayes achieve an accuracy of 0.88–0.92, this is as the anomaly detection dataset is biased, with most of

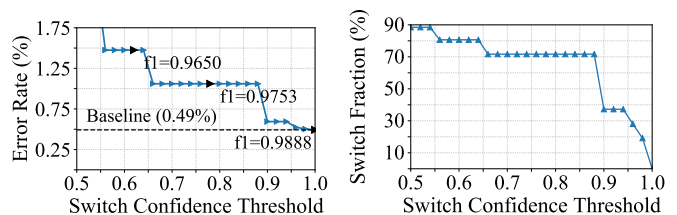


Figure 10: Anomaly Detection in a hybrid deployment (Random Forest) - fraction of traffic handled by the switch and misclassification rate.

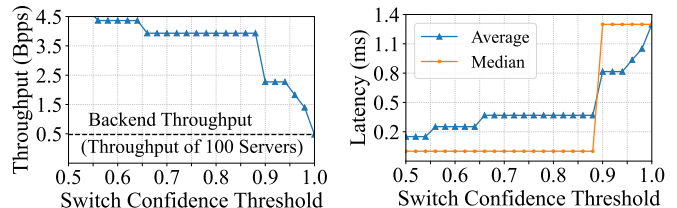


Figure 11: Throughput and latency of hybrid deployment in Anomaly Detection use case (same setup as in Figure 10). Backend uses 100 servers.

the traffic benign. Using ensemble tree models, we correctly identify anomalies.

The baseline for ML performance comparison is the full ensemble model running on backend servers. We implement on the switch a small model (Table IV), that classifies a subset of the traffic, and forwards to the backend all low-confidence or anomalous traffic. A confidence level is set in the switch to determine the threshold for on-switch classification.

Figure 10 shows for the anomaly detection use case using Random Forest, the fraction of traffic offloaded by the switch and the corresponding misclassification rate, as a function of the switch classification confidence threshold. The baseline results is a misclassification rate of 0.49% and an F1 score of 0.9888. In comparison, with a confidence threshold of 0.7, 84.5% of the traffic is handled by the switch, achieving a misclassification rate of 1.03% and F1 score of 0.976. These improve as the confidence threshold increases, but the fraction of traffic handled by the switch decreases. For the same scenario, Figure 11 shows the throughput and latency of hybrid deployment, where the throughput follows the fraction of offloaded while latency is the opposite. Switch’s and server’s performance match the results in §VIII-F, and the backend uses 100 servers.

Figure 12 presents the effect of confidence threshold on the ML performance of financial market prediction. Figure 12 (a) shows that the baseline achieves an error rate of 0.231. In comparison, the hybrid model achieves an error rate of 0.271 with a confidence threshold of 0.5. Increasing the confidence level to 0.7 reduces the error rate to 0.236. However, there is a trade-off here, shown in Figure 12 (b): with a threshold of 0.6, 72.91% of transactions are classified by the switch, whereas at 0.7 confidence, 50.07% of the transactions are classified by the switch. To put these results in context, consider Figure 12 (c), which shows the error rate for classifications done by the switch compared with the error rate for the same transactions if done by the host. As the graph shows, transactions that achieve low confidence (below 0.8) on the switch, are less likely to be

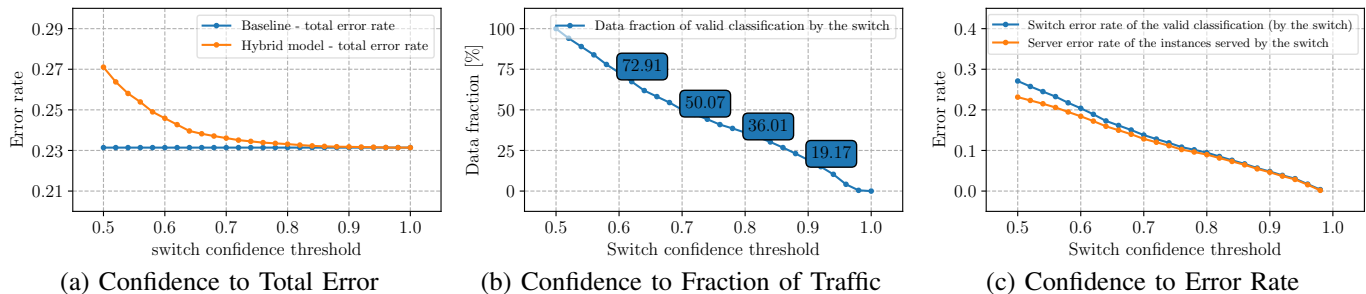


Figure 12: Financial transactions in a hybrid deployment (XGBoost) - error rate and fraction of traffic handled by the switch.

misclassified by the full-grown model running on the server. In fact, starting 0.8 confidence threshold (where 36.01% of decisions are being served by the switch model) the error rate difference between the server and the switch is very small, and some traders may even find that the error difference at 0.7 is still small enough to provide higher transactions rate for 50.07% of the transactions.

H. Optimizations

An easy resource optimization for ensemble models is reducing the number of trees or their depth in the training stage, thereby reducing ML performance. As demonstrated in Table IV, in a hybrid deployment, there is a minor benefit to using *Medium* or *Large* switch models over a *Small* model. This can further be tweaked by changing the confidence threshold, and is configurable.

Memory resources can be saved by quantizing or dropping entries, i.e. consciously mapping more entries in the features tables to the same code. This reduces the size of all tables but may lead to a loss of ML performance. In an experiment using the finance use case and longest prefix match (lpm) tables, we find that dropping entries covering ranges with a single value has a negligible effect on ML performance on a stand-alone switch, but memory requirements are reduced by over 20%. However, dropping entries with 2 consecutive values (one masked bit) leads to significant accuracy loss, and less than 10% memory savings.

I. ML Model Updates

For run-time updates (§V-E), we measure the update time of our ML ensemble models on a switch. The update time varies based on the size of the model: from 50ms for a small model, to several seconds for a large one (Table IV). We address the effect of ML model updates in a follow-up work [65].

IX. DISCUSSION

Generalization. The focus of this paper is on the methodology of mapping ML models to network devices. IIsy’s mapping solution is designed for RMT-based [39] programmable network devices, which have been the focus of the community’s research (e.g., [7], [16], [15]), and in particular P4 practitioners. IIsy was tested on multiple architectures, such as v1model, SimpleSumeSwitch [52] and TNA [25]. As IIsy uses a simple data plane, with complexity mainly in the algorithms

mapping from the trained models to table entries, porting between targets is straight forward. It requires syntax changes in the P4 code generator and a script generating control-plane commands, but not to the mapping tool. While this paper focuses on six different mapped models, the mapping can be easily extended to more models, details are provided in [47].

Benefits. A lesson of this work is that despite resource constraints, network switches can serve as important classification components in hybrid deployments. Saving microseconds (or more) of latency in time-sensitive applications and reducing the load on backend servers by tens of percent, without adding new hardware to the infrastructure. While classification cannot be added to a fully utilized switch, our results show that the resource overheads of in-network classification are minimal.

Scope. This paper focuses on the methodology of mapping trained ML models to network devices. The work does not seek to improve the quality of training ML models, nor to contribute to a specific use case. Applying the methodology to certain applications, such as congestion control, is beyond the scope of the paper. While the methodology offered in this work cannot be directly applied to neural network models, our choice of ensemble models is primarily as they provide the best results for the example use cases.

Hybrid Deployment. In hybrid classification scenarios, different settings can be used to determine which packets should be forwarded to the backend for additional processing. One intuitive choice is to consider only confidence level, as applied in the financial market prediction use case, achieving high accuracy across all labels. Alternatively, there are other options, such as forwarding only the traffic with low confidence under a specific class. This approach is used in this paper in the anomaly detection use case, where the focus is on identifying and mitigating attacks. In this case, only packets classified as normal with low confidence are subjected to further scrutiny, while all packets labeled malicious are dropped as a precaution. The specific setting can be adjusted based on use case requirements.

Model Update. To address data drift and the dynamic network environment, IIsy-supported models can be updated while ensuring uninterrupted normal traffic flow through the control plane. We address the effect of ML model updates in a follow-up work [65].

Limitations. Some of the limitations discussed in this work, e.g., the number of tables or features, are the property of the target platform and will change on a different platform. For

Project	Target	Models	Const.
BaNaN [22]	RMT, NIC	BNN	P
N3IC [17]	NIC, FPGA	BNN	P
Qin [18]	bmv2, NIC	BNN	✗
pForest [23]	bmv2, ASIC	RF	✓
SwitchTree [20]	bmv2	RF	✗
NERDS [21]	bmv2, NIC	RF	P
NetBeacon [63]	ASIC	RF, XGB	✓
IOI [16]	Modified ASIC	NN	—
Taurus [15]	Modified ASIC	DNN, SVM, KM, LTSM	P
iSwitch [19]	FPGA	RL	P
Clustreams [24]	ASIC	KM	✗
Iisy	ASIC, FPGA	SVM, KM, NB, DT, RF, XGB	✓

Table V: A comparison of in-network classification solutions. Legend: Const. - Resource constrained. NN - Neural Network. BNN/DNN - Binary/Deep NN. RF - Random Forest. NB - Naïve Bayes. KM- K-means. XGB - XGBoost. P - Partial.

example, NetFPGA is mostly limited by memory and logic resources, while on Tofino, memory and logic resources are rarely limiting us, and we are limited by different constraints, such as the number of stages.

X. RELATED WORK

The application of ML to network traffic, especially for traffic classification, has been of interest for a long time (e.g., [11], [66]). Using ML for scheduling and congestion control (e.g., [67]) was also studied. The focus of these works has been on using ML over traditional computing platforms.

The challenges of ML have led researchers to explore new approaches to resource-constrained ML, using devices of limited resources [68], [69]). This work focuses on network switches, using a drastically different pipeline architecture and with much higher processing rates.

The first to propose in-network classification was Sanvito *et al.* [22]. As shown in Table V, several works have implemented ML on a software switch [22], [20], [21], [70], a NIC [21] an FPGA [17], [19], or a DPU [71]. These targets have more resources and significantly lower throughput than a switch.

Clustreams [24] used a tree data structure to implement K-means, successfully clustering 80%-90% of objects. pForest [23] was implemented on Tofino, using a methodology similar to SwitchTree [20], with limited scalability (e.g., depth of 4). Our baseline comparison with SwitchTree is indicative of pForest.

Several works proposed modifying switches for ML purposes, adding dedicated new modules [19], [15], [16]. Iisy’s use of off-the-shelf switches is complementary to these works.

Iisy’s early work [33] was used by many follow up projects, and in particular Planter [47], DINC [72], NetBeacon [63], and Homunculus [73] which automate the deployment process and hyperparameter search, respectively.

An orthogonal thread of research uses programmable switches to accelerate ML frameworks. These works focus on parameter servers and in-network aggregation [13], [12]

in the training stage, rather than on classification, operating under different sets of requirements.

Using programmable network devices for anomaly detection was explored both at the host side [56] and within switch-ASIC (e.g., [74]). Our work is orthogonal to these non-ML based efforts. Several studies [75], such as Planter [47], SwitchTree [20], NERDS [21], and NetBeacon [63], focused on in-network anomaly classification, and referenced Iisy [33]. Iisy outperforms these later works, as discussed in VIII. P4Pir [65] is a follow up work which focuses on updating ML models for anomaly detection on IoT gateways.

ML for financial transactions has been widely researched, with XGBoost and SVM often used [76]. Acceleration of financial transactions has mostly focused on the backend, e.g. using FPGA [77]. A related programmable switches project is the publish-subscribe system [78] for the NASDAQ Market data feed filter and router. Based upon Iisy, Linnet [79] and LOBIN [80] use in-network ML for market prediction using limit order books.

XI. CONCLUSION

This paper presented Iisy, mapping trained ML models to programmable switches for in-network classification. Using a hybrid deployment, Iisy reduces the load on the backend, achieves high throughput, low latency, and high ML performance, while coexisting with standard switch functionality.

Acknowledgments This paper complies with all applicable ethical standards of the authors’ home institution. This work was partly funded by VMware, EU Horizon SMARTEDGE (101092908, UKRI 10056403), Leverhulme Trust (ECF-2016-289) and Isaac Newton Trust. We acknowledge support from Intel. 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] K. Hazelwood, S. Bird, D. Brooks *et al.*, “Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective,” in *IEEE HPCA*, 2018, pp. 620–629.
- [2] S. Cass, “Taking AI to the edge: Google’s TPU now comes in a maker-friendly package,” *IEEE Spectrum*, vol. 56, no. 5, pp. 16–17, 2019.
- [3] A. Boroumand, S. Ghose, Y. Kim *et al.*, “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks,” in *ASPLOS*, 2018, pp. 316–331.
- [4] A. A. Awan, H. Subramoni, and D. K. Panda, “An In-depth Performance Characterization of CPU- and GPU-based DNN Training on Modern Architectures,” in *ACM MLHPC*, 2017.
- [5] M. Blott, T. B. Preußer, N. J. Fraser *et al.*, “FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks,” *ACM TRETS*, 2018.
- [6] N. P. Jouppi, C. Young, N. Patil *et al.*, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” in *IEEE ISCA*, 2017, pp. 1–12.
- [7] X. Jin, X. Li, H. Zhang *et al.*, “NetCache: Balancing Key-Value Stores with Fast In-Network Caching,” in *ACM SOSP*, 2017, pp. 121–136.
- [8] H. T. Dang, P. Bressana, H. Wang *et al.*, “P4xos: Consensus as a Network Service,” *IEEE/ACM TON*, vol. 28, no. 4, pp. 1726–1738, 2020.
- [9] C. Kim, A. Sivaraman, N. Katta *et al.*, “In-band Network Telemetry via Programmable Dataplanes,” in *ACM SIGCOMM*, 2015.
- [10] Y. Tokusashi, H. T. Dang, F. Pedone *et al.*, “The Case For In-Network Computing On Demand,” in *EuroSys*, 2019.
- [11] A. W. Moore and D. Zuev, “Internet Traffic Classification Using Bayesian Analysis Techniques,” in *SIGMETRICS PER*, 2005.
- [12] A. Sapio, M. Canini, C.-Y. Ho *et al.*, “Scaling Distributed Machine Learning with In-Network Aggregation,” in *NSDI*, 2021.

- [13] C. Lao, Y. Le, K. Mahajan *et al.*, “Atp: In-network aggregation for multi-tenant learning,” in *NSDI*, 2021.
- [14] D. Barradas, N. Santos, L. Rodrigues *et al.*, “FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications,” in *NDSS*, 2021.
- [15] T. Swamy, A. Rucker, M. Shahbaz *et al.*, “Taurus: a data plane architecture for per-packet ML,” in *ASPLOS*, 2022, pp. 1099–1114.
- [16] Z. Zhong, W. Wang, M. Ghobadi *et al.*, “IOI: In-network Optical Inference,” in *SIGCOMM OptSys*, 2021.
- [17] G. Siracusano, S. Galea, D. Sanvito *et al.*, “Re-architecting Traffic Analysis with Neural Network Interface Cards,” in *NSDI*, 2022.
- [18] Q. Qin, K. Poullarakis, K. K. Leung *et al.*, “Line-Speed and Scalable Intrusion Detection at the Network Edge via Federated Learning,” in *IFIP Networking*, 2020.
- [19] Y. Li, I.-J. Liu, Y. Yuan *et al.*, “Accelerating Distributed Reinforcement Learning with In-switch Computing,” in *ISCA*, 2019, pp. 279–291.
- [20] J. H. Lee and K. Singh, “SwitchTree: In-network Computing and Traffic Analyses with Random Forests,” *Neural Computing and Applications*.
- [21] B. M. Xavier, R. S. Guimarães, G. Comarela *et al.*, “Programmable Switches for in-Networking Classification,” in *WKSHPs CCC*, 2021.
- [22] D. Sanvito, G. Siracusano, and R. Bifulco, “Can the Network be the AI Accelerator?” in *NetCompute*, 2018.
- [23] C. Busse-Grawitz, R. Meier, A. Dietmüller *et al.*, “pForest: In-Network Inference with Random Forests,” *arXiv:1909.05680*, 2019.
- [24] R. Friedman, O. Goaz, and O. Rottenstreich, “Clustreams: Data Plane Clustering,” in *SOSR*, 2021, pp. 101–107.
- [25] Intel, “P4_16 Tofino Native Architecture Application Note - Public Version,” 2021. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch-Document.pdf.
- [26] P. Bosshart, D. Daly, G. Gibb *et al.*, “P4: Programming Protocol-Independent Packet Processors,” *SIGCOMM CCR*, 2014.
- [27] N. Zilberman, M. Grosvenor, D. A. Popescu *et al.*, “Where has my time gone?” in *PAM*. Springer, 2017, pp. 201–214.
- [28] S. Laki, C. Györgyi, J. Pető *et al.*, “In-Network Velocity Control of Industrial Robot Arms,” in *NSDI*, 2022, pp. 995–1009.
- [29] P. Schulz, M. Matthe, H. Klessig *et al.*, “Latency Critical IoT Applications in 5G: Perspective on the Design of Radio Interface and Network Architecture,” *IEEE Communications Magazine*, 2017.
- [30] AWS, “New – TLS Termination for Network Load Balancers,” 01 2019. [Online]. Available: <https://aws.amazon.com/blogs/aws/new-tls-termination-for-network-load-balancers/>
- [31] Businesswire, *InsidePacket extends SONiC use cases, enabling new edge-cloud network services*, 03 2020.
- [32] G. Brebner, “Extending the Range of P4 Programmability,” in *Keynote in the EuroP4*, 2018.
- [33] Z. Xiong and N. Zilberman, “Do Switches Dream of Machine Learning? Toward In-Network Classification,” in *HotNets*, 2019, pp. 25–33.
- [34] B. Zadrozny and C. Elkan, “Transforming Classifier Scores into Accurate Multiclass Probability Estimates,” in *SIGKDD*, 2002, pp. 694–699.
- [35] S. Vargaftik and Y. Ben-Itzhak, “Efficient Multiclass Classification with Duet,” in *EuroMLSys*, 2022, pp. 10–19.
- [36] J. Tang, S. Alelyani, and H. Liu, “Feature Selection for Classification: A Review,” *Data classification: Algorithms and applications*, p. 37, 2014.
- [37] K. He, X. Zhang, S. Ren *et al.*, “Deep Residual Learning for Image Recognition,” in *CVPR*, 2016, pp. 770–778.
- [38] Y. Ganin, E. Ustinova, H. Ajakan *et al.*, “Domain-Adversarial Training of Neural Networks,” *JMLR*, vol. 17, no. 1, pp. 2096–2030, 2016.
- [39] P. Bosshart, G. Gibb, H.-S. Kim *et al.*, “Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN,” in *SIGCOMM*, 2013, pp. 99–110.
- [40] N. Zilberman, Y. Audzevich, G. Covington *et al.*, “NetFPGA SUME: Toward 100 Gbps as Research Commodity,” *IEEE Micro*, 2014.
- [41] Z. Zhou, *Ensemble Methods: Foundations and Algorithms*. CRC press.
- [42] L. Breiman, “Random Forests,” *Springer Machine learning*, 2001.
- [43] R. E. Schapire and Y. Freund, “Boosting: Foundations and Algorithms,” *Kybernetes*, 2013.
- [44] M. E. Maron, “Automatic Indexing: An Experimental Inquiry,” *Journal of the ACM*, vol. 8, no. 3, pp. 404–417, 1961.
- [45] D. J. Hand and K. Yu, “Idiot’s Bayes-Not So Stupid After All?” *International statistical review*, vol. 69, no. 3, pp. 385–398, 2001.
- [46] F. Pedregosa, G. Varoquaux, A. Gramfort *et al.*, “Scikit-learn: Machine learning in python,” *JMLR*, vol. 12, no. Oct, pp. 2825–2830, 2011.
- [47] C. Zheng, M. Zang, X. Hong *et al.*, “Automating In-Network Machine Learning,” 2022.
- [48] V. Sivaraman, S. Narayana, O. Rottenstreich *et al.*, “Heavy-Hitter Detection Entirely in the Data Plane,” in *ACM SOSR*, 2017, pp. 164–176.
- [49] S. Sen and J. Wang, “Analyzing peer-to-peer traffic across large networks,” in *ACM SIGCOMM Workshop on IM*, 2002, pp. 137–150.
- [50] R. Glebbe, J. Krude, I. Kunze *et al.*, “Towards Executing Computer Vision Functionality on Programmable Network Devices,” in *ENCP*.
- [51] F. Pedregosa, G. Varoquaux, A. Gramfort *et al.*, “Scikit-learn: Machine Learning in Python,” *JMLR*, vol. 12, pp. 2825–2830, 2011.
- [52] S. Ibanez, G. Brebner, N. McKeown *et al.*, “The P4→NetFPGA Workflow for Line-Rate Packet Processing,” in *ACM FPGA*, 2019, pp. 1–9.
- [53] C. Zheng, Z. Xiong, T. T. Bui *et al.*, “Illy: Practical In-Network Classification,” 2022.
- [54] N. Moustafa and J. Slay, “UNSW-NB15: a comprehensive data set for network intrusion detection systems,” in *IEEE MilCIS*, 2015.
- [55] J. S. Group. (2020) Jane Street Market Prediction. <https://www.kaggle.com/c/jane-street-market-prediction>. [Online; accessed May 2022].
- [56] Z. Zhao, H. Sadok, N. Atre *et al.*, “Achieving 100Gbps Intrusion Prevention on a Single Server,” in *OSDI*, 2020.
- [57] M. Baron, J. Brogaard, B. Hagströmer *et al.*, “Risk and Return in High-Frequency Trading,” *JFQA*, vol. 54, no. 3, pp. 993–1024, 2019.
- [58] R. A. Fisher, “The Use of Multiple Measurements in Taxonomic Problems,” *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.
- [59] E. Chatzoglou, G. Kambourakis, and C. Kolias, “Empirical Evaluation of Attacks Against IEEE 802.11 Enterprise Networks: The AWID3 Dataset,” *IEEE Access*, vol. 9, pp. 34 188–34 205, 2021.
- [60] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, “Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization,” *ICISSP*, vol. 1, pp. 108–116, 2018.
- [61] S. Stolfo, W. Fan, W. Lee *et al.*, “Cost-based Modeling for Fraud and Intrusion Detection: Results from the JAM Project,” in *DISCEX*, 2000.
- [62] I. Alrashdi, A. Alqazzaz, E. Aloufi *et al.*, “AD-IoT: Anomaly Detection of IoT Cyberattacks in Smart City Using Machine Learning,” in *IEEE CCWC*, 2019, pp. 0305–0310.
- [63] G. Zhou, Z. Liu, C. Fu *et al.*, “An Efficient Design of Intelligent Network Data Plane,” in *USENIX Security*, 2023.
- [64] H. Jiang, Z. He, G. Ye *et al.*, “Network Intrusion Detection Based on PSO-Xgboost Model,” *IEEE Access*, vol. 8, pp. 58 392–58 401, 2020.
- [65] M. Zang, C. Zheng, L. Dittmann *et al.*, “Towards Continuous Threat Defense: In-Network Traffic Analysis for IoT Gateways,” in *IOTJ*, 2023.
- [66] A. Dainotti, A. Pescapé, and K. C. Claffy, “Issues and Future Directions in Traffic Classification,” *IEEE Network*, vol. 26, no. 1, pp. 35–40, 2012.
- [67] V. Dukić, S. A. Jyothi, B. Karlaš *et al.*, “Is advance knowledge of flow sizes a plausible assumption?” in *NSDI*, 2019, pp. 565–580.
- [68] S. Wang, T. Tuor, T. Salonidis *et al.*, “Adaptive Federated Learning in Resource Constrained Edge Computing Systems,” *IEEE JSAC*, 2019.
- [69] R. Sanchez-Iborra and A. F. Skarmeta, “TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities,” *IEEE Circuits and Systems Magazine*, 2020.
- [70] M. Zang, C. Zheng, T. Koziak *et al.*, “Federated learning-based in-network traffic analysis on IoT edge,” 2023.
- [71] M. Hemmatpour, C. Zheng, and N. Zilberman, “E-Commerce Bot Traffic: In-Network Impact, Detection, and Mitigation,” in *ICIN*, 2024.
- [72] C. Zheng, H. Tang, M. Zang *et al.*, “DINC: Toward Distributed In-Network Computing,” in *Proceedings of ACM CoNEXT’23*, 2023.
- [73] T. Swamy, A. Zulfiqar, L. Nardi *et al.*, “Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks,” in *ASPLOS*, 2023, pp. 329–342.
- [74] Z. Liu, H. Namkung, G. Nikolaidis *et al.*, “Jaquen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches,” in *USENIX Security*, 2021.
- [75] C. Zheng, X. Hong, D. Ding *et al.*, “In-Network Machine Learning Using Programmable Network Devices: A Survey,” *IEEE COMST*, 2023.
- [76] A. Tsantekidis, N. Passalis, A. Tefas *et al.*, “Using Deep Learning to Detect Price Change Indications in Financial Markets,” in *IEEE EUSIPCO*, 2017, pp. 2511–2515.
- [77] R. Velu, *Algorithmic Trading and Quantitative Strategies*. CRC Press.
- [78] T. Jepsen, A. Fattaholmanan, M. Moshref *et al.*, “Forwarding and Routing with Packet Subscriptions,” in *CoNEXT*, 2020, pp. 282–294.
- [79] X. Hong, C. Zheng, S. Zohren *et al.*, “Linnet: Limit Order Books Within Switches,” in *SIGCOMM’22 Poster*, 2022, pp. 37–39.
- [80] X. Hong, C. Zheng, S. Zohren *et al.*, “LOBIN: In-Network Machine Learning for Limit Order Books,” in *IEEE HPSR*, 2023.