Do Switches Dream of Machine Learning? Toward In-Network Classification

Zhaoqi Xiong
University of Cambridge

Noa Zilberman
University of Cambridge
noa.zilberman@cl.cam.ac.uk

ABSTRACT

Machine learning is currently driving a technological and societal revolution. While programmable switches have been proven to be useful for in-network computing, machine learning within programmable switches had little success so far. Not using network devices for machine learning has a high toll, given the known power efficiency and performance benefits of processing within the network. In this paper, we explore the potential use of commodity programmable switches for in-network classification, by mapping trained machine learning models to match-action pipelines. We introduce IIsy, a software and hardware based prototype of our approach, and discuss the suitability of mapping to different targets. Our solution can be generalized to additional machine learning algorithms, using the methods presented in this work.

ACM Reference Format:
Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *ACM Workshop on Hot Topics in Networks (HotNets '19), November 13–15, 2019, Princeton, NJ, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3365609.3365864

1 INTRODUCTION

Machine learning (ML) is increasingly dominating digital aspects of our everyday life: from personalized online shopping, through social networks to finance and trading. The systems community is battling to support ML demands while facing an increasing number of barriers: from the end of Moore's law [33, 39] and Dennard's scaling [19] to the memory and dollar walls [42]. These challenges have driven innovation in hardware design for ML, including CPU optimization (e.g., [6, 53]), GPU (e.g., [14]) and FPGA (e.g., [18, 21, 51]) solutions, and specialized processing ASIC [12, 26]. All forms of ML acceleration receive considerable attention.

Networking has not escaped the ML trend, and ML is being used both for optimization and decision making (e.g., [13, 23, 52]). With the rise of programmable network devices, one would have expected that in-network ML would gain much traction. Not only has in-network computing demonstrated superior performance [24, 25], but it is also power efficient [50]. Still, in-network ML has eluded the networking community. Indeed, it has been shown that network devices can be used to improve distributed ML, for example, through in-network aggregation [45], but this was not an implementation of ML within a network device. Possibly the first steps toward in-network inference were in N2Net [47] and BaNaNa Split [43], which discussed the implementation of neural networks within programmable network devices.

In this paper we focus on the following question: given that ML training was already conducted, *can we deploy the trained model within programmable network devices?* By that, we focus on one specific aspect of ML, classification.

We concentrate our efforts on classification for ML inference that is not neural network based, and show that the trained models for packet classification seamlessly map to programmable data planes. The mapping enables traffic classification at line rate, and we find a balance between limited resources, line rate, and classification accuracy. Furthermore, we show that as long as the set of features is static, updates to classification models can be deployed through the control plane alone, without changes to the data plane.

In this paper, we make the following contributions:

- We demonstrate the mapping of four different trained machine learning algorithms to a match-action pipeline.
- We introduce software and hardware based prototypes of a framework that automatically maps trained models to a match-action pipeline.
- We demonstrate in-network classification within a hardware target and quantify resource requirements.

Our work is limited in scope. We do not explore all ML algorithms, or even the most popular ones, e.g., neural-networks are beyond the scope of this work. We do not claim any contribution in ML algorithms. Also, our approach is by design limited in accuracy and in the types of features it can extract. We believe our classification-focused contribution is the first

HotNets '19, November 13–15, 2019, Princeton, NJ, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7020-2/19/11...$15.00
https://doi.org/10.1145/3365609.3365864

step toward more complex, dedicated implementations of inference within network devices. Our code is available at [57].

1.1 Motivation

In-network ML, meaning running ML within network devices, is an exciting prospect for multiple reasons. First, switches offer very high performance. The latency through a switch is in the order of hundreds of nanoseconds per packet [3], while high-end ML accelerators operate at the scale of tens of microseconds to milliseconds per inference [26, 36]. The same accelerators achieve 10-90TOPS/sec [26], while a 16Tb/sec switch supports about fifteen billion packets per second [31]. However, OPS/s are not directly mapped to packet rate, e.g., NVidia Tesla V100 will infer 10K images/sec [36], while it is possible to transmit the same images dataset through a switch at a rate of over 10M images/sec. Network switches' power efficiency enables processing 10M's of packets per Watt[50], better than most accelerators, the absolute power consumption of a programmable switch [3] is lower than current-day alternatives [26].

Switches have another advantage: being ideally placed for some ML use-cases. The performance of distributed ML is bounded by time required to get data to and from nodes. If a switch can classify at the same rate that it carries packets to nodes in a distributed system, then it will equal or outperform any single node. Outside the data center, switches have further advantages. They can terminate data early, reducing the load on the network, and supporting scalability over time. Terminating data close to the edge saves power, reduces load on infrastructure and improves user experience thanks to reduced latency. Best of all, switches are already deployed within the network, and do not require installing additional hardware. As long as a switch can support both ML and networking operations, it will provide a solution cheaper than a system augmented with ML accelerators, and will free up cycles on CPUs running ML applications.

In this work we focus only on classification. With the amount of user-generated data continuously increasing, the network becomes our *first line of defense* against the absurd scale of data being streamed, exceeding ten zetabytes per year [15]. The increased use of Internet of Things (IoT) is expected to lead to further data inflation, with much of this data subject to classification. While today middleboxes can still sustain the amount of processing required at the edge, they are unlikely to continue and scale as data demand grows [60]. But what if we could run classification within the network, before the data reaches its processing destination? We will not only reduce processing loads, but also respond earlier to events, terminating traffic close to the edge. Applications such as autonomous cars and automated factories are just a few of the latency critical applications [46] that will benefit.

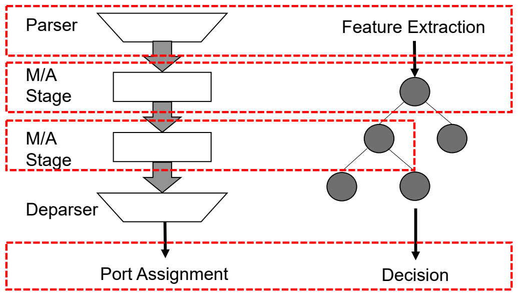

Figure 1: The similarity between a decision tree and a simple switch pipeline. Analogous components are circled. M/A indicates match-action. The pipeline's output can be more than just a port assignment.

Perhaps the most simple in-network classification example to consider is the Mirai Botnet, which used embedded and IoT devices to create a denial of service attack on selected targets [2]. Would it have been possible to stop the attack early on if edge devices had dropped all Mirai-related traffic based on the results of ML-based inference, rather than using "standard" access control lists? We discuss use-cases further in § 6.3 and §7.

2 SWITCH AS A CLASSIFICATION MACHINE

Commodity switches naturally act as classification machines. Consider the example of a standard layer 2 Ethernet switch, and while any switch-architecture applies, we maintain a mental model of a programmable data plane, such as used by P4 [8] or NPL [35].

As a new object (a packet) arrives, the first step is to extract the relevant features from it. In a switch, this resembles parsing the packet's header. Each header's field is, in fact, a feature, and the header parser is the features extractor.

The next step in a classification process is to apply the object's features to a trained ML model. In the case of a layer 2 Ethernet switch, this model takes the form of a non-binary decision tree, of one level. The feature used in the root's split is the destination MAC address. By accessing the MAC address table in the switch, the object's (packet) feature (destination MAC address) is finding the right branch. Once the appropriate branch is found, the object is assigned to a class, meaning the packet is assigned to an output port. We illustrate this similarity in Figure 1.

A layer 2 Ethernet switch can be represented also by more complex decision trees. One example would be checking that the source port is not identical to the destination port, and dropping the packet if the values are identical. In a decision tree, this will translate to adding another level, and an additional class (drop).

3 KEY INSIGHTS

One of the reasons machine learning algorithms have not been implemented within network devices to date, is the implementation complexity of the mathematical operations required. Implementing in switch hardware operations such as addition, xor or bit shifting is easy, but operations such as multiplication[1], polynomials or logarithms cannot be pipelined well, may add latency of affect throughput. However, while switches do not support operations such as $log(x)$ or $log(y)$, once $log(x)$ and $log(y)$ are known, operations such as $log(x \times y)$ can be easily done.

We do not try to implement mathematical operations within the switch. Instead, we build upon a common practice in programmable hardware (e.g., FPGA) and high performance computing: we use look up tables to store the results of calculations, or the equivalent of the results. Look up tables are a perfect fit to the match-action paradigm used by programmable switches.

In practice, implementation is not as easy as it sounds. Hardware switches have a finite amount of resources, and one cannot store an infinite size table to support all possible values. A solution we adopt in this work is not to store any potential value in the table, and be willing to lose some accuracy for the price of feasibility. We further save memory by storing classification results or codes (§5) rather than computation results, allowing us a more efficient use of ternary and longest prefix match (LPM) tables.

A second insight, already mentioned, is that switches are already set up as classification machines, with the parser acting as a features extraction module, and the match-action pipeline as a means to make the classification.

In many switch architectures only part of the packet goes through the programmable data plane, and the rest is buffered [8]. To process an entire packet, one solution is packet recirculation, with the packet (and features) being fragmented to header-size data units, and iteratively going through the pipeline. This approach degrades throughput, and requires adjustments to maintain metadata, but may still perform well in networks with low utilization or sufficient speed-up.

4 REALISTIC RESOURCE ALLOCATION

So far we have discussed the key insights enabling the of mapping classification algorithms to a match-action pipeline. In this section, we take a more realistic approach to in-network classification, considering commodity switches' limitations.

First, we note that switches are likely not to do only classification, but also, importantly, switching. It is thus expected that until we consider switching and other switch functionality an act of machine learning, that significant resources will be taken up by fundamental networking functionality.

Second, all the in-network classification solutions presented above share an important property: they don't require any externs, meaning **no target-specific functionality is required**. This pure match-action implementation enables porting between different targets, and means that a solution is not locked to a single platform.

We observe that today's programmable switches, support an order of 12 to 20 stages per pipeline, with multiple (e.g., four) pipelines per device [3, 34], setting an upper bound on the supported functionality. The tables' memory is likely to be in the order of hundreds of megabits [9], possibly divided across multiple pipelines. Last, a parser can extract only a limited number of headers, likely of the same order as the depth of the pipeline, therefore in some classification implementations the number of features will be of the same scale as the number of classes. On the other hand, the width of the features can be substantial, e.g. IPv6 address is 128 bit wide. It is unrealistic to expect tables of depth 2^{128}. Tables are therefore not expected to be deep proportionally to the key's width, but proportionally to the network's size (in a switch) or classification problem (for in-network classification).

In practical terms, using multiple features as the key to a table will not be easily feasible: silicon vendors have struggled to implement lookup tables for IPv6's 128b addresses, with current state-of-the-art memory depth reaching 300K-400K entries [4, 5][2], thus anything significantly (e.g., $> \times 10$) larger than that can be considered impractical. However, assuming 128b is a feasible key width opens up significant opportunities: as TCP source and destination port are represented by 16-bit each, flags are often just a few bits, and EtherType is 16-bit as well, multiple features can be concatenated into a single key without reaching the width of an IPv6 address.

We avoid setting bounds on the amount of resources required to implement different algorithms, as such bounds will bear little resemblance to reality. For example, if a key to a table is of width w, the depth of the table will be 2^w only if the table is direct-mapped, in which case it can be implemented as a simple memory and needs to store only the action. The use of exact match, LPM and range-type tables is intended specifically to avoid the table-depth requirement, yet it leads to an increased table-width (key width plus action width). We discuss this further in §6.3.

One way to increase the number of features (or classes) used in the classification is by concatenating multiple pipelines, where the output of one pipeline is feeding the input of the next pipeline. This approach will face two challenges. First, it will reduce the maximum throughput of the device, by a

[1]P4 supports multiplication, but allows the architecture to impose limitations, such as multiplying only by a power of two

[2]Some ASICs support >1M entries.

factor of the number of concatenated pipelines. Second, the metadata we use to carry information between stages is not shared between pipelines [3], and information may need to be embedded in an intermediate header.

5 FROM ML TO MATCH-ACTION

In this work, we adopt the P4 approach to programmable data planes [8], assuming a general pipeline model in the form of PISA or RMT [9]. We explore packet classification using four algorithms, both supervised and unsupervised: decision trees, K-means, SVM and Naïve Bayes. These algorithms are chosen because of the differences between them, and the results can be generalized. Neural-networks are beyond the scope of this work. Table 1 summarizes our approach.

5.1 Decision Trees

A decision tree can be intuitively mapped to a match-action pipeline. In every stage,a set of conditions are applied to a feature, and their results lead to different branches of the tree. As conditions are simple operations, they can be implemented in P4. This approach is wasteful, as the tree depth and conditions define the number of stages in the pipeline.

We propose a different approach, more suitable for match-action pipelines (Table 1.1[4]): the number of stages implemented in the pipeline equals the number of features used plus one. In every stage, we match one feature with all its potential values. The result (action) is the encoded into a metadata field, and indicates a branch taken in the tree. The last stage within the pipeline takes the coded fields of all features from the metadata bus, and maps (matches) the value to the resulting leaf node.

Decision trees can be implemented using *range*-type tables [37], but those are not available on many hardware targets. If the number of feature values is known and limited, using exact match tables instead may be feasible. Alternatively, ternary and LPM tables can be used, breaking a range into multiple entries, consequently increasing the resource consumption (compared to range-type tables), but providing a feasible path for usage.

5.2 SVM

A second supervised algorithm, support vector machine (SVM), uses hyperplanes to separate classes. The output of a training process takes the form of multiple equations, where each equation represents an hyperplane[5]:

$$\begin{cases} a_1x_1 + b_1x_2 + ...z_1x_n + d_1 = 0 \\ a_2x_1 + b_2x_2 + ...z_2x_n + d_2 = 0 \\ \qquad\qquad ... \\ a_mx_1 + b_kx_2 + ...z_mx_n + d_m = 0 \end{cases}$$

where n is the number of features, k is the number of classes and $m = k * (k - 1)/2$.

One approach to SVM (Table 1.2) implements m tables, each table dedicated to a hyperplane, and indicating on which side of a hyperplane is a given input. The key used to access the match-action table is the set of features, and to avoid complex operations, the action is the "vote". A "vote" is a one-bit value mapped to the metadata bus that indicates if the input belongs within or outside a hyperplane. Once an input is matched against all m tables, all the "votes" (the sum of the metadata bus, across classes) are counted, and the class with the highest count of "votes" is the classification's result.

A second approach (Table 1.3) is to dedicate a table per feature, where the output of the table is a vector of the form $a_1x_1, a_2x_1, ...a_mx_1$. At the end of the match-action pipeline, the value of each hyperplane is calculated as the sum of all vectors, and a decision is taken. This approach requires smaller tables, but is limited: the values in the generated vectors have a limited accuracy (e.g., float cannot be represented), and may require a lot of bits. In addition, significant logic (sum operations) may be required at the end of the match-action pipeline.

5.3 Naïve Bayes

Our exploration of the supervised Naïve Bayes classifier [30] assumes a Gaussian distribution of independent features [20]. Related methods which may be more accurate for network traffic classification, such as kernel estimation [32], will follow similar implementation concepts. Under this assumption, the likelihood of feature x_i is expressed as:

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

And the classification rule will be:

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

Given a k classification problem, where n features are used, there are $k \times n$ pairs of (μ_y, σ_y). A naive implementation (Table 1.4) would use $k \times (n+1)$ match-action tables, to calculate for k classifications n probabilities, plus the product of all n calculated probabilities of that class. This process is not only wasteful, but is also hard to approximate in hardware when the probabilities are small.

A second approach (Table 1.5) uses one table per class, with all the features as the key. Instead of returning a float

[3]This is architecture and implementation dependent
[4]We use the notation Table 1.i to refer to entry i in Table 1.
[5]We indicate linearity for presentation only

	Classifier	A table per...	Key	Action	Last stage
1.	Decision Tree (1)	Feature	Feature's value	Feature's code word	Table, Decoding code words
2.	SVM (1)	Class (hyperplane)	All features	Vote	Logic/table, Votes counting
3.	SVM (2)	Feature	Feature's value	Calculated vector	Logic, hyperplanes calculation
4.	Naïve Bayes (1)	Class & feature	Feature's value	Probability	Logic, highest probability
5.	Naïve Bayes (2)	Class	All features	Probability	Logic, highest probability
6.	K-means (1)	Class & feature	Feature's value	Square distance	Logic, overall distance
7.	K-means (2)	Cluster	All features	Distance from core	Logic, distance comparison
8.	K-means (3)	Feature	Feature's value	Distance vectors	Logic, overall distance

Table 1: Different manners of implementing in-network classification within a match-action pipeline. *Logic* refers only to addition operations and conditions.

value as the classification's probability, the returned value is an integer value that symbolizes the probability. As long as similar values are used to symbolize probabilities across tables (different classifications), this approach yields accurate results. The downside here is the size of the required table: it uses a very wide key (a form of concatenation of all input features values), and its depth is proportional to this width.

5.4 K-means

K-means clustering is an example of unsupervised learning. For k classes it provides k centers of clusters, each composed of n coordinate values, one per feature. The distance of a given input x to a center of a cluster i is calculated as:

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

where x_1 to x_n are the values of the features. An input will be classified to the cluster to which it has the lowest distance.

For choosing a cluster based on shortest distance, it is sufficient to consider the square distances. One of the options (Table 1.7) is to use a table per cluster, with the key being all features. This approach requires less tables, compared with a table per c_j^i coordinate (Table 1.6), but instead uses much deeper and wider tables. A different approach (Table 1.8) uses one table per feature, and its action assigns to the metadata bus a set of distance values on a single axis, one per cluster. In this approach, the last stage both adds up the distance vectors and classifies to the smallest one.

Feasibility. Based on §4 and Table 1, our understanding of real-world switches indicates the feasibility and limitations of each implementation approach. Implementations 4[6] (Naïve Bayes) and 6 (K-means) will be both very limited. Even in a data-plane dedicated only to classification, it is not practical to use more than 4-5 features and 4-5 classes without exceeding the available number of stages, or alternatively, 2 classes and 10 features (and vice versa). Other methods provide more flexibility: supporting up to 20 classes or features. Classifiers 1 (Decision Tree), 3 (SVM) and 8 (K-means) will provide the best scalability, as a combination of number of tables, key's width and action's width.

[6]Numbers indicate entry number in Table 1.

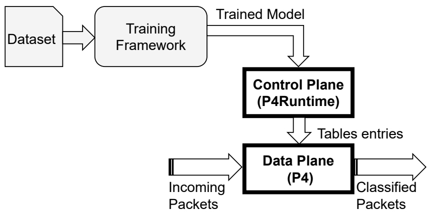

Figure 2: High level architecture of IIsy. IIsy's components are in white. External components are in grey.

6 PROTOTYPE AND EVALUATION

We implement **IIsy**, **I**n-network **I**nference made ea**sy**, as a prototype of our observations. Our framework includes a software-based implementation, demonstrating the ability to automatically map classification algorithms to network devices, and a hardware-based implementation, exploring resource requirements and performance.

IIsy has three components. First, a machine learning training environment. Second, a programmable data plane within a network device, and third, a control plane used for mapping the trained algorithm to the network device. The framework is illustrated in Figure 2.

We use Scikit-learn [40] as a training environment. While the input to the training can be any dataset, we use labelled packet traces as our input, to illustrate realistic traffic scenarios. Scikit-learn is selected for ease of use, and be replaced by other training environments, as long as their outputs can be converted to a text format matching our control plane.

6.1 Software-based prototype

Our software-based prototype implements the data plane in P4 using the v1model architecture. P4Runtime [38] is used for the control plane. We use bmv2 and mininet as for testing.

We write a P4 program per use-case. A use-case refers to a pair of parameters: the machine learning algorithm used, and the expected network traffic. The network traffic dictates the design of the data-plane's parser (i.e., which headers need to be parsed) and the machine learning algorithm defines the match-action design that needs to be implemented.

A python script is used to generate the control plane. We take the output of the ML training stage, and convert the parameters to table-writes to the match-action pipeline, using P4runtime. This stage is, despite its simplicity, the most important stage: it enables us to change the network devices' operation, and implement different classification rules without changing the P4 program, as long as the type of machine learning model and the set of features used do not change.

6.2 Hardware-based prototype

IIsy's hardware prototype is implemented over NetFPGA SUME [59] using the P4→NetFPGA workflow [22]. Currently, P4→NetFPGA does not support P4runtime, and we implement the control plane configurations using the P4→NetFPGA control plane interface. Our hardware implementation mainly explores the feasibility of porting to hardware targets and the classification rate. We use a P4 program that is very similar to the software-based prototype, but with minor hardware-target alterations: range-type tables are replaced by exact-match or ternary tables, and the syntax is adapted to the P4→NetFPGA workflow requirements. Another difference from our software implementation is that the architecture used by P4→NetFPGA is SimpleSumeSwitch [22].

For the performance evaluation we use OSNT, an open source network tester [1] for traffic generation at line rate ($4\times10G$), and for latency measurements. As OSNT can replay limited size packet traces, functional testing using large trace files is done using tcpreplay over a standard X520 NIC.

6.3 Example: IoT Traffic

IoT is a driving use-case for in-network classification, as noted in §1. We use pcap traces of IoT devices released by Sivanathan et al. [48] as our dataset. Our goal is to demonstrate using IIsy to classify incoming traffic by device-type.

We divide the monitored devices to five classes: static smart-home devices (e.g., power plug), sensors (e.g., weather sensor), audio (e.g., smart assistants), video (e.g., security camera), and "others". We pick classes that can be mapped to different quality of service groups: from high bandwidth (video) to best effort ("others" class). A set of 11 features is selected for the evaluation, such as EtherType, IP protocol and flags and TCP ports. All the features are directly extracted from the packet's header. We do not use identifiable information such as MAC or IP address, which may both skew our results (as the devices have fixed addresses), and as we wish to show that classification can be deployed independently of address-based match-action practices.

Table 2 summarizes the dataset's properties. For six of the features, only a small number of values exists in the dataset, meaning that very small tables, or even registers, may suffice to hold their computed values. A table holding packet sizes will still fit within a standard lookup table [56]. Using exact match tables for TCP and UDP port numbers is feasible, but comes at a high cost on FPGA targets: each such table will consume close to 2Mb of memory [56], and may not allow to meet timing constraints. For this reason, we use ternary tables, allowing to match over a range of values.

Feature	Unique Values	Class	Num. Packets
Packet Size	1467	Static devices	1,485,147
Ether Type	6	Sensors	372,789
IPv4 Protocol	5	Audio	817,292
IPv4 Flags	4	Video	3,668,170
IPv6 Next	8	Other	17,472,330
IPv6 Options	2		
TCP Src Port	65536		
TCP Dst Port	65536		
TCP Flags	14		
UDP Src Port	43977		
UDP Dst Port	43393		

Table 2: Selected properties of the IoT training dataset

Our choice of eleven features will fit devices such as Barefoot Tofino, where using a table per feature, and one decision table, equals the number of stages in the pipeline [3, 34]. We expect future work to explore more complex features, as well as a larger number of them (See §7), and the implementation feasibility over NetFPGA platform. Exploring optimal fitting of rules to tables is also beyond the scope of this paper, and was extensively studied (e.g., [10, 11, 27]).

We train our models using Scikit-learn, and also obtain model's statistics. We validate the classification based on mapping to ports. Our goal is that the switch's classification output will match the model's classification result.

We implement all four models in IIsy, one approach per model, and evaluate functionality and resource consumption. Our performance test measures throughput and latency, and is conducted for the decision tree implementation.

A summary of the resources required to implement the models on the NetFPGA platform [59] is provided in Table 3. Utilization figures refer to the Virtex-7 690T FPGA used on the board, providing a relative comparison between the different methods. We use small tables of 64 entries, except for the last (decision) table, which uses exact match and is set to the number of possible options. Tables of 512 entries fit on the FPGA, but fail to close timing at 200MHz.

A look at the small size tables provides interesting insights. For example, for the decision tree, between two and seven match ranges are required per feature, and those fit into the tables consuming no more than 47 entries, a significant saving from 64K potential values (e.g., TCP port). In contrast, models that use multiple features as a key to the table are much harder to map to table entries, and require reordering

Model	# tables	Logic Util.	Memory Util.
Reference Switch	3	15%	33%
Decision Tree	6	27%	40%
SVM (1)	11	34%	53%
Naïve Bayes (2)	5	30%	44%
K-means	5	30%	44%

Table 3: Resource utilization of in-network classification implementations on NetFPGA-SUME.

of bits between features (interleaving most significant bits first, and least significant last) to enable matching across ranges. As can be expected, 64 entries are not sufficient for a match without loss of accuracy.

The most accurate implementation uses a decision tree. A trained model with a tree depth of 11 achieves an accuracy of 0.94, with similar precision, recall and F1-score. Reducing the tree depth decreases the prediction's accuracy by 1%-2% with every level. On NetFPGA we implement a pipeline with just five levels, with accuracy and F1-score of approximately 0.85. Consequently, only five features are required. As a reminder, our goal is not to find an optimal traffic classification model, but to conduct classification that is as accurate as the trained model. The accuracy of the implementation is evaluated by replaying the dataset's pcap traces and checking that packets arrive at the ports expected by the classification. Our classification is identical to the prediction of the trained model. We further evaluate the performance of the implementation, using OSNT, and verify that we reach full line rate. The latency of our design (which is toolchain-version dependent) is $2.62\mu s$ ($\pm 30ns$), on a par with reference (non-ML) P4$\rightarrow$NetFPGA designs with a similar number of stages.

7 DISCUSSION

In-Network Computing: In-network classification is a class of in-network computing. While in this work we take in-network computing for granted, following several high-profile works (e.g., [24, 25, 44]), this research area is still considered controversial and immature [7, 41]. One important challenge is that in-network computing consumes resources otherwise required for networking purposes. Using a switch as a network-attached accelerator, will maintain the throughput benefits without sharing resources, but will require power and space that come almost for free when the computing is done as part of a packet's network traversal.

Feature Extraction: In our prototype we mostly used features extracted from packet headers. Examples such as cache queries [50] or DNS requests [55] can be similarly extracted. Features such as queue size may be available through the pipeline's metadata bus, but are architecture specific. Extracting features that require state, such as flow size, is possible [29, 49] but requires using e.g., counters or externs, and may be target-specific. Many ML models require complex features that it may not be possible to extract on a switch.

Performance and Scalability: Our implementation uses only match-action tables, without complex operations. Consequently, on hardware targets, the performance of IIsy will be similar to the platform's packet processing rate. While IIsy scales in throughput, it may not scale with features-number or values-per-feature. This is a property that varies between hardware targets (§4). The solution that we offer trades classification's precision for resources, where classes that are expected to have lower precision are tagged for further processing by a host, similar to [54].

Switch ASIC: Porting IIsy to commercial switches is left to future work. Discussions with switch vendors indicate that it is feasible, and likely to achieve line rate performance.

Use Cases: The most expected use cases of IIsy are network traffic related, such as traffic classification [32], as IIsy proposes a means to handle traffic-volume scalability challenges [16]. Related use cases include traffic filtering and mitigation of distributed denial of service attacks. Congestion control is another likely use case, with features such as queue size readily available on some hardware targets.

8 RELATED WORK AND CONCLUSION

Recent years have seen a surge in research within the intersection of ML and networking. This ranged from network traffic classification [17, 32, 58] to using ML for scheduling and congestion control [23, 52]. ML frameworks are being accelerated using network devices [18, 44], either as parameters servers or to aggregate and multicast traffic [28, 45].

The implementation of inference within network devices is still in its infancy. N2Net [47] and BaNaNa Split [43] have demonstrated implementations of binary neural networks within network devices and analyzed processing and communications overheads. Li [28] proposed an implementation of reinforced learning within a switch, but used a bespoke acceleration module. This paper is complementary to these works, discussing non-neural networks ML algorithms.

In this paper, we have introduced IIsy, a framework for in-network classification. We have mapped both supervised and unsupervised algorithms to a match-action pipeline, and discussed the applicability of such implementations. Our prototypes are implemented both in software and hardware, and achieve full line rate classifying real world traces. This is but the first step in implementing ML within network devices, and In-network training is the next big challenge.

ACKNOWLEDGMENTS

We thank the anonymous HotNets reviewers and our shepherd, Manya Ghobadi, for their valuable feedback. This project was partially funded by the Leverhulme Trust (ECF-2016-289) and the Isaac Newton Trust.

REFERENCES

[1] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, et al. 2014. OSNT: Open source network tester. *IEEE Network Magazine* 28, 5 (2014), 6–12.

[2] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the Mirai botnet. In *26th USENIX Security Symposium*. 1093–1110.

[3] Arista. 2018. *Arista 7170 Multi-function Programmable Networking*. White Paper, https://www.arista.com/assets/data/pdf/Whitepapers/7170_White_Paper.pdf.

[4] Arista. 2019. *7060X4 Series 100/200/400G Data Center Switches*. https://www.arista.com/assets/data/pdf/Datasheets/7060X4-Datasheet.pdf.

[5] Arista. 2019. *7800R3 Series Data Center Switch Router*. https://www.arista.com/assets/data/pdf/Datasheets/7800R3-Data-Sheet.pdf.

[6] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. 2017. An in-depth performance characterization of CPU-and GPU-based DNN training on modern architectures. In *Proceedings of the Machine Learning on HPC Environments*. ACM.

[7] Theophilus A Benson. 2019. In-Network Compute: Considered Armed and Dangerous. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 216–224.

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

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM '13)*. ACM, New York, NY, USA, 99–110.

[10] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yacov Hel-Or. 2018. Encoding short ranges in TCAM without expansion: Efficient algorithm and applications. *IEEE/ACM Transactions on Networking* 26, 2 (2018), 835–850.

[11] Anat Bremler-Barr and Danny Hendler. 2010. Space-efficient TCAM-based classification using gray coding. *IEEE Trans. Comput.* 61, 1 (2010), 18–30.

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[13] Sandeep P Chinchali, Eyal Cidon, Evgenya Pergament, Tianshu Chu, and Sachin Katti. 2018. Neural networks meet physical networks: Distributed inference between edge devices and the cloud. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. ACM, 50–56.

[14] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: performance and programmability. *IEEE Micro* 38, 2 (2018), 42–52.

[15] Cisco. 2018. *Cisco Global Cloud Index: Forecast and Methodology, 2016–2021*. "https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html"

[16] Alberto Dainotti, Antonio Pescape, and Kimberly C Claffy. 2012. Issues and future directions in traffic classification. *IEEE network* 26, 1 (2012), 35–40.

[17] Jeffrey Erman, Martin Arlitt, and Anirban Mahanti. 2006. Traffic classification using clustering algorithms. In *Proceedings of the 2006 SIGCOMM workshop on Mining network data*. ACM, 281–286.

[18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 1–14.

[19] David J Frank, Robert H Dennard, Edward Nowak, Paul M Solomon, Yuan Taur, and Hen-Sum Philip Wong. 2001. Device scaling limits of Si MOSFETs and their application dependencies". *Proc. IEEE* 89 (2001), 259–288.

[20] David J Hand and Keming Yu. 2001. Idiot's Bayes—not so stupid after all? *International statistical review* 69, 3 (2001), 385–398.

[21] Zhenhao He, David Sidler, Zsolt István, and Gustavo Alonso. 2018. A flexible K-means operator for hybrid databases. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 368–3683.

[22] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilbermann. 2019. The P4→NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 1–9.

[23] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *International Conference on Machine Learning*. 3050–3059.

[24] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. 35–49.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.

[26] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. IEEE, 1–12.

[27] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. 2005. Algorithms for advanced packet classification with ternary CAMs. In *ACM SIGCOMM Computer Communication Review*, Vol. 35. ACM, 193–204.

[28] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement Learning with In-switch Computing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. 279–291.

[29] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the ACM SIGCOMM Conference*. ACM, 101–114.

[30] Melvin Earl Maron. 1961. Automatic indexing: an experimental inquiry. *J. ACM* 8, 3 (1961), 404–417.

[31] Mellanox. 2019. *Mellanox Quantum HDR Switch Silicon*. "https://www.mellanox.com/related-docs/prod_silicon/PB_Quantum_HDR_Switch_Silicon.pdf"

[32] Andrew W Moore and Denis Zuev. 2005. Internet traffic classification using bayesian analysis techniques. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 33. ACM, 50–60.

[33] G. E. Moore. 1965. Cramming More Components onto Integrated Circuits. *Electronics* 38 (April 1965), 114–117.

[34] Timothy Prickett Morgan. 2018. *Programmable Networks Get A Bigger Foot In The Datacenter Door*. The Next Platform.

[35] NPLang.org 2019. *NPL Specification*. NPLang.org. Rev. 1.3.

[36] Nvidia. 2018. *Nvidia AI inference platform performance study*. Technical Overview, https://www.nvidia.com/content/dam/en-zz/Solutions/data-center/gated-resources/inference-technical-overview.pdf.

[37] P4 Language Consortium 2018. *P4_16 Language Specification*. P4 Language Consortium. Rev. 1.1.0.

[38] P4 Language Consortium 2019. *P4Runtime Specification*. P4 Language Consortium. Rev. 1.0.0.

[39] David A Patterson and John L Hennessy. 2013. *Computer organization and design: the hardware/software interface*. Newnes.

[40] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[41] Dan RK Ports and Jacob Nelson. 2019. When Should The Network Be The Computer?. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 209–215.

[42] Partha Ranganathan. 2019. *End of Moore's law and how an introverted computer architect learned to love networking*. Keynote.

[43] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. 2018. Can the Network be the AI Accelerator?. In *Proceedings of the 2018 Workshop on In-Network Computing*. ACM, 20–25.

[44] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 150–156.

[45] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling Distributed Machine Learning with In-Network Aggregation. *arXiv preprint arXiv:1903.06701* (2019).

[46] Philipp Schulz, Maximilian Matthe, Henrik Klessig, Meryem Simsek, Gerhard Fettweis, Junaid Ansari, Shehzad Ali Ashraf, Bjoern Almeroth, Jens Voigt, Ines Riedel, et al. 2017. Latency critical IoT applications in 5G: Perspective on the design of radio interface and network architecture. *IEEE Communications Magazine* 55, 2 (2017), 70–78.

[47] Giuseppe Siracusano and Roberto Bifulco. 2018. In-network neural networks. *arXiv preprint arXiv:1801.05731* (2018).

[48] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. 2018. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing* (2018).

[49] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. ACM, 164–176.

[50] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 21.

[51] Da Tong, Yun Rock Qu, and Viktor K Prasanna. 2017. Accelerating decision tree based traffic classification on FPGA and multicore Platforms. *IEEE Transactions on Parallel and Distributed Systems* 28, 11 (2017), 3046–3059.

[52] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. 2019. Is advance knowledge of flow sizes a plausible assumption?. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 565–580.

[53] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. (2011).

[54] Shay Vargaftik, Isaac Keslassy, and Yaniv Ben-Itzhak. 2019. RADE: Resource-Efficient Supervised Anomaly Detection Using Decision Tree-Based Ensemble Methods. *arXiv preprint arXiv:1909.11877* (2019).

[55] Jackson Woodruff, Murali Ramanujam, and Noa Zilberman. 2019. P4DNS: In-Network DNS. In *Proceedings of the 2nd P4 Workshop in Europe*.

[56] Xilinx. 2019. *Exact Match Binary CAM Search IP for SDNet*. Smart-CORE IP Product Guide, PG189 (v1.0) https://www.xilinx.com/support/documentation/ip_documentation/cam/pg189-cam.pdf.

[57] Zhaoqi Xiong and Noa Zilberman. 2019. *IIsy Repository*. https://github.com/cucl-srg/IIsy/.

[58] Jun Zhang, Xiao Chen, Yang Xiang, Wanlei Zhou, and Jie Wu. 2015. Robust network traffic classification. *IEEE/ACM Transactions on Networking (TON)* 23, 4 (2015), 1257–1270.

[59] Noa Zilberman, Yury Audzevich, G.Adam Covington, and Andrew W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro* 34, 5 (September 2014), 32–41.

[60] Noa Zilberman, Andrew W Moore, and Jon A Crowcroft. 2016. From photons to big-data applications: terminating terabits. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374, 2062 (2016), 2014.0445.