Henna: Hierarchical Machine Learning Inference in Programmable Switches

Aristide Tanyi-Jong Akem
aristide.akem@imdea.org
IMDEA Networks Institute
Universidad Carlos III de Madrid
Madrid, Spain

Beyza Bütün
beyza.butun@imdea.org
IMDEA Networks Institute
Universidad Carlos III de Madrid
Madrid, Spain

Michele Gucciardo
michele.gucciardo@imdea.org
IMDEA Networks Institute
Madrid, Spain

Marco Fiore
marco.fiore@imdea.org
IMDEA Networks Institute
Madrid, Spain

ABSTRACT

The recent proliferation of programmable network equipment has opened up new possibilities for embedding intelligence into the data plane. Deploying models directly in the data plane promises to achieve high throughput and low latency inference capabilities that cannot be attained with traditional closed loops involving control-plane operations. Recent efforts have paved the way for the integration of trained machine learning models in resource-constrained programmable switches, yet current solutions have significant limitations that translate into performance barriers when coping with complex inference tasks. In this paper, we present Henna, a first in-switch implementation of a hierarchical classification system. The concept underpinning our solution is that of splitting a difficult classification task into easier cascaded decisions, which can then be addressed with separated and resource-efficient tree-based classifiers. We propose a design of Henna that aligns with the internal organization of the Protocol Independent Switch Architecture (PISA), and integrates state-of-the-art strategies for mapping decision trees to switch hardware. We then implement Henna into a real testbed with off-the-shelf Intel Tofino programmable switches using the P4 language. Experiments with a complex 21-category classification task based on measurement data demonstrate how Henna improves the F1 score of an advanced single-stage model by 21%, while keeping usage of switch resources at 8% on average.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Programmable switch, machine learning, in-switch inference, P4

NativeNI '22, December 9, 2022, Roma, Italy

ACM Reference Format:
Aristide Tanyi-Jong Akem, Beyza Bütün, Michele Gucciardo, and Marco Fiore. 2022. Henna: Hierarchical Machine Learning Inference in Programmable Switches. In *Native Network Intelligence (NativeNI '22), December 9, 2022, Roma, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3565009.3569520

1 INTRODUCTION

The growing complexity and requirement for flexibility in modern network architectures necessitate network operations automation via *Network Intelligence* (NI) [1], where human intervention in network management is minimized or even eliminated where possible [2, 8]. Artificial intelligence (AI) models are viewed as the key enablers of NI in next-generation mobile networks [6, 18]. In particular, data-driven AI techniques based on machine learning (ML) tools have been especially successful at automating a number of tasks in network environments [4, 12, 14]. According to traditional realization of the software-defined network (SDN) paradigm, the NI models are implemented and run in programmable control planes [9]. However, control-plane models do not operate at line rate and hence cannot meet the extreme low latency requirements that characterize specific next-generation network functions [31].

The recent availability of commercial programmable data planes like Intel Tofino ASICs [11] or Netronome Network Processing Units (NPUs) [15], and of dedicated languages like P4 [3] opens new opportunities in terms of low-latency high-throughput inference in networks. Yet, programmable switches are highly constrained, with low available memory and limited support for mathematical operations [17, 19]. This rules out the possibility of training models in the switch, and limits the complexity of tasks that can be solved with ML models mapped onto the switch pipeline.

In this context, models based on decision trees (DTs) are presently those most suitable to in-switch operation, mainly because their logical structure makes them easier to map onto the PISA pipeline [5, 13, 26, 27, 29–31]. However, most of the solutions proposed to date have significant limitations which induce performance barriers when dealing with complex inference tasks. pForest [5] introduces a random forest (RF) mapping approach later replicated by Switchtree [13], where each level of a tree is mapped in a match and action table, with a final decision being made at tables encoding leaf nodes. This approach is not scalable since the tables of a

tree have to be accessed in the sequence of the levels they encode, thereby being directly bounded by the number of stages in the switch pipeline, and hence not suitable for handling complex tasks. NERDS [26] encodes single decision trees using a series of nested if/else conditional statements, which do not scale for even slightly complex tasks in the switch with limited resources. pHeavy [29] is a specialized machine learning-based heavy flow detector, which is not easily adaptable to other applications, does not support RFs and hence would not be directly applicable to difficult classification problems. Mousika [27] presents an encoding of decision trees by knowledge distillation which is very memory efficient. However, it only supports DTs and has only been validated with simple tasks. IIsy [28, 30] proposes an approach to map DTs into the switch pipeline by encoding feature thresholds over tree nodes using code words. The concept is extended in Planter [31, 32], by overlapping feature encoding over trees and implementing RF models. Planter represents the state-of-the-art in-switch implementation of RFs but, as we will see in our experiments, it shows its limits when it is used for complex inference tasks.

We note that other models, including Support Vector Machines (SVM), Naive Bayes, K-Means, XGBoost, or Isolation Forest have also been implemented in switches Planter [31]; however, none of them was found to offer better performance and higher scalability than DTs and RFs. Also, attempts at using binary neural networks to overcome the current performance barriers of RFs have been hampered by the limited support for mathematical operations and available memory of off-the-shelf programmable switches [21]; such neural network models have instead found applications in more flexible user plane environments, like smart network interface cards (smartNICs) [22] or FPGA-enhanced switching pipelines [24]. Here, it is worth noting that smartNICs are typically deployed at specific network appliances (*e.g.*, traffic classifiers or load balancers) in dedicated hosts within a data center: therefore, they grant line-rate inference at specific locations of the network only, and not at any point of the transport domain as enabled by in-switch solutions. And, attaching FPGAs to regular programmable switches has a clear and significant added deployment costs that hinder applying that approach at scale. Ultimately, we conclude that *there is still a large margin for improvement in solutions for relatively complex inference tasks that can be deployed pervasively within programmable switches.*

In this work, we make a step towards addressing the aforementioned gap and present a model for hierarchical in-switch machine learning, or Henna, which is a first in-switch implementation of a cascaded tree classifier. The main concept behind our solution is that a difficult classification task can be split into a sequence of easier ones, each of which can then be tackled using simpler, resource-efficient, and performance-improving tree-based classifiers. Therefore, Henna builds on the concept of hierarchical or multi-stage classification, which is known to alleviate imbalances in data and simplify classification tasks by exploiting the relationships between classes in earlier stages [10]. By developing Henna, we make the following contributions.

- We propose an in-switch implementation of a hierarchical two-stage classifier which breaks down a difficult classification task into simpler ones that are themselves easily handled by smaller classifiers.
- We exploit both ingress and egress processing to logically separate the two stages of Henna. This establishes our solution as the first in-switch inference model to exploit both parts of the pipeline, achieving a more efficient resource allocation.
- We implement Henna as well as a one-stage benchmark classifier in a commercial P4-programmable Intel Tofino switch and, unlike most previous works, we make our source code publicly[1] available to promote research in the area of in-network inference.
- We run experiments on an actual testbed and demonstrate how Henna improves performance with respect to the benchmark by a relative gain of up to 21% for a device identification task with 21 classes, which is a much more complex use case than those considered in previous works that targeted a maximum of 8 classes.

2 HENNA CASCADED TREE MODEL

Flat classification is the most straightforward approach to classification problems, where all the classes are inferred in one stage by a monolithic model. Although a one-stage strategy works well for problems where the classes are just a few or are naively told apart, it becomes less practical in tasks where the number of classes is large and the differences among them become more nuanced: indeed, in the latter situation, monolithic models tend to become extremely complex in order to meet the desired accuracy.

To address the accuracy-complexity trade-off, inherent hierarchical relationships between one or more classes in the problem can be exploited to simplify the classification task, by identifying class groups in earlier stages. The concept underpins hierarchical classification, where the task is broken down into a multiple stages in a hierarchy or directed acyclic graph, where easier distinctions are made at higher stages and the final stage identifies the actual class of the sample [20]. A multi-stage approach also helps to mitigate the problem of imbalance in datasets, by training local classifiers for groups of classes where each class is more likely to be better represented than in the whole dataset.

2.1 Motivation

Previous works have shown that the resource usage for in-switch machine learning models surges with model complexity [30, 31]. As the number of classes increases, the model size also grows until it hits a performance barrier determined by the amount of available resources in the switch. In these cases, a more complex model that could have yielded better accuracy is just not feasible for in-switch operation. Also, we remark that although next generations of programmable switches will arguably have more resources and capabilities, some constraints, like the maximum number of bits that can fit in a unit of ternary content-addressable memory (TCAM), will most likely remain and require techniques to overcome them.

In this scenario, hierarchical paradigms come to the rescue by splitting the target overall task into simpler ones that are themselves easier to handle; then, smaller classifiers can be trained to solve the sub-tasks, collectively yielding a better accuracy while being able to fit within the limited switch capabilities.

[1] Our python and P4 code is available at https://github.com/nds-group/Henna.

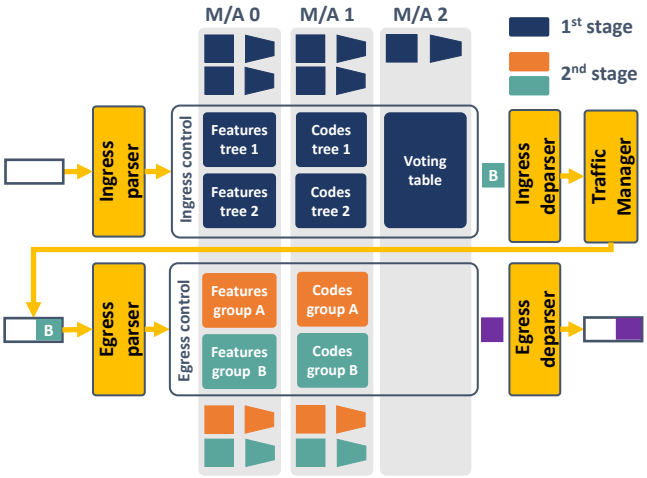


Figure 1: Overview of the Henna two-stage architecture, mapped onto the ingress and egress PISA pipeline.

All solutions currently proposed in the literature and listed in Section 1 adopt a monolithic approach that suffers from the issue above: by addressing the whole inference task via a single DT or RF, existing tree-based models do not scale well with the complexity of the problem, and their performance is restrained by the in-switch environment. To the best of our knowledge, Henna is the first model that does not rely on a single-stage architecture, and implements instead a hierarchical multi-stage operation for in-switch inference.

2.2 In-switch hierarchical inference design

Figure 1 provides an overview of the Henna workflow. We design our solution as a two-stage cascade of tree-based classifiers, which exploits both the ingress and egress packet processing pipelines of PISA to classify packets at high speed and with low latency. Specifically, (*i*) the ingress is mapped to a first-stage classifier, which tags each incoming packet according to a target group of classes; then, (*ii*) the egress implements a second-stage classifier, which identifies the actual class of the packet among those of the group. The main components of the system are illustrated in the figure and function as follows.

Ingress parser. Incoming packets go through the ingress parser which extracts header information from the headers and stores them in the Packet Header Vector (PHV). Such header information includes packet length, transport protocol ports, and flags, all of which serve as features for the subsequent ML-based inference.

Ingress control. This is the first part of the match-and-action (M/A) pipeline that a packet goes through upon exiting the parser. Once the packet arrives here with the PHV carrying the features as metadata, the first model in the Henna cascade is executed so that a class group is assigned to the packet by the first stage of the hierarchical classifier. We implement this first-stage model as an RF. The class group information is stored in a header field which will be then available to the egress pipeline. The packet is then reassembled at the ingress deparser and is sent to the egress via the traffic manager as shown in Figure 1.

Egress parser. As the packet arrives the egress, it is again parsed as in the ingress, and all the necessary header information, which now includes the classification result of the previous stage, are extracted and stored in the PHV.

Egress control. The class group determined by the first stage is checked and, based on its value, a specific second-stage model is selected for further classification of the packet. Generally, the tasks of these second-stage models are much simpler than that faced in the first stage; therefore, in our implementation[2], all second-stage models are implemented as DTs. The appropriate DT model is then run on the features gathered by the egress parser, and the output final class information is stored in a header field for downstream accuracy analysis. The processed packet is reassembled at the egress deparser and forwarded to the desired switch port.

Tree-based model mapping. To map the RF and DT models to the ingress and egress switch M/A pipeline, we employ the state-of-the-art mapping proposed in Planter [31]. This mapping strategy efficiently represents in a single feature table all the possible thresholds used on a same feature by the decision nodes of any tree, across multiple trees of an RF; this is a very compact representation, which is much more resource-efficient than, *e.g.*, mapping tree levels to M/A units [5]. In the control logic, the value of each feature for the current packet in the PHV is matched against the corresponding feature table, and the triggered action sets a binary code that encodes the decisions to be taken at each node using that feature. The paths to all leaf nodes in each tree can be then described as a specific sequence of feature-level codes, and are stored in dedicated code tables, one per tree: then, matching the concatenation of feature-level codes obtained for a specific packet against such code tables returns the per-tree class decision for the packet.

Voting table. Abiding by the original model [32], the code tables also report a certainty value indicating how accurate is the decision taken by the tree. In the case of RFs, the final decision is made by a majority vote over the classes output by all the trees in the model. When ties occur, the certainty values are checked, and the tied decision with the highest certainty is accepted. It is worth noting that we engineered our own version of the Planter mapping since at the time of writing no implementation was publicly available.

Stages allocation. As a final remark, we highlight that the RF and DT models could be both implemented in the ingress pipeline, without using egress resources. However, this approach forces the trees in the two stages to line up in series in the ingress processing, contending for the same M/A resources and thereby limiting the number of tree models within a same M/A unit. Allocating the models of the two stages to the ingress and egress pipeline, respectively, avoids the problem and allows trees in both stages to share the resources of the M/A units, ultimately leading to more efficient use of the scarce switch resources.

2.3 Offline model preparation

Due to the limited capabilities of the switch, all ML steps preliminary to inference must be executed offline [19]. These include feature extraction, model training and validation, as well as the

[2]Extending our implementation of Henna to support RF models also in the second stage is straightforward, but was not required for the use case we considered in our performance evaluation, presented in Section 3.2.

generation of all feature and code tables that allow implementing the tree-based models in PISA, as described in Section 2.2. We detail these steps below.

Feature extraction. We compute stateless packet-level features in the training data using Tshark [7], which extracts header information of packets from historical pcap traces.

Model selection. Feature selection and a grid search on the model hyperparameters are performed with the help of Scikit-Learn libraries [16]. Specifically, we perform the feature selection by training the RF model in the first stage and DT models in the second stage with all extracted features and then ranking the features according to their importance as expressed by the Mean Decrease in Impurity (MDI). We subsequently find the subset of features that achieve the best performance by adding the ranked features one by one and training the models with those only. We repeat the feature selection process jointly across a traditional grid search on the maximum number of trees in the RF model and the maximum depth of each tree, so as to identify the best hyperparameters.

Tree pruning. The switch can only support a limited size of the sequence of feature-level codes used to represent a path within a tree. This inherently limits the number of leaves that can be supported by the hardware. We thus use pruning to limit the number of leaves of the tree to fit the hardware constraints. We do so by tuning the `max_leaf_nodes` parameter of the `RandomForestClassifier()` in Scikit-Learn.

Table generation. From the best trained model, we generate feature and code table entries abiding by the operation in Section 2.2. For each DT, we collect all the thresholds of individual features and build feature tables; then, for each leaf, we trace and encode the path to the leaf in a binary string, which is stored into the code table. For the RFs, we simply loop over all trees in each step and combine all thresholds of the same feature in the same table to have a single feature table for all the trees.

3 EXPERIMENTAL SETUP

We implement Henna as a P4 program, which can be compiled and executed in PISA-compliant programmable switches. We run experiments in a real-world testbed using a device classification use case of much higher complexity than inference tasks considered to date to evaluate user-plane ML models.

3.1 Testbed setup

We implement our models in a rackmount testbed comprising 3 Edgecore Wedge100BF-QS programmable switches equipped with an Intel Tofino BFN-T10-032Q ASIC and 32 100GbE QSFP28 ports each. The testbed is completed by 2 servers with Intel 8-core Xeon processors at 2GHz, 48GB of RAM and QSFP28 interfaces, which are connected to the switches. The experimental platform thereby represents a production-grade full-100Gbps network, and is depicted in Figure 2. Each switch runs Open Network Linux (ONL) and the Intel Software Development Environment (SDE) version 9.7.0. We build a Barefoot Runtime Interface (BRI)-based Python controller that performs the initial configuration of the switch at start up, sets up ports, and injects the table entries encoding the machine learning models. We use 100Gbps connections to send captured traffic through the switch from one server to another, by replaying

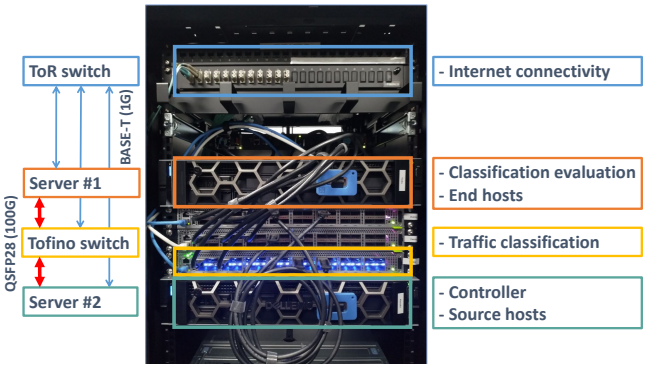


Figure 2: Photo of the experimental testbed, annotated with the hardware elements (left) and their role (right).

pcap traces via Tcpreplay[25]. The captured traffic on the receiving server is then analyzed for classification statistics.

3.2 Inference use case

In order to explore the limits of in-switch inference, we target a much more complex use case than those typically considered in the related literature. Specifically, we use the publicly available UNSW-IoT traces [23], which report 20 days of captured traffic in pcap files, and contain measurement data for traffic flows generated by a wide range of IoT and non-IoT devices. The inference objective is to classify packets transiting in the switch by tagging them with one of 21 possible source devices. To this end, we use packet header information as input to the ML model: the features considered comprise TCP flags (ACK, SYN, PUSH, ECE, RESET, FIN), TCP/UDP source and destination ports, and packet length. We remark that the largest classification task tacked in related works involved 8 classes for solutions only tested in emulated environments, and just 6 classes for models deployed in actual hardware.

In Henna, we organize the 21 classes into 5 class groups: Switches and Plugs (Belkin Wemo switch, iHome, TP-Link Smart plug, and Light Bulbs LiFX Smart Bulb); Sensors (Withings Aura smart sleep sensor, Belkin wemo motion sensor, and NEST Protect smoke alarm); Video Devices (Withings Smart Baby Monitor, Insteon Camera, TP-Link Day Night Cloud camera, Samsung SmartCam, Dropcam, and Netatmo Welcome); Appliances (PIX-STAR Photo-frame, Amazon Echo, Triby Speaker, Netatmo weather station, Withings Smart scale, and Smart Things); and, Computers (Laptop, and MacBook). The first stage of our hierarchical model aims at identifying the group via a RF, whereas in the second stage 5 dedicated DTs return the actual class given the group. Based on the hyperparameter tuning procedure described in Section 2.3, we use a RF model with 3 trees of maximum depth 10 in the first stage, and DT models with variable depth from 4 to 10, depending on the target group, in the second stage. We train Henna on 15 days of data and test on 1 day.

3.3 Benchmark and metrics

We compare Henna to a state-of-the-art single-stage model, *i.e.*, a monolithic RF using our implementation of Planter [31]. This benchmark aims at classifying each of the 21 target devices at once.

To this end, it is configured with 3 trees of maximum depth of 10; note that 10 is the maximum depth considered for DT or RF in-switch implementations in the literature as deeper trees are considered impractical [5, 31], whereas increasing the number of trees in the RF did not result in any performance gain, as shown in Figure 3 for the metrics presented hereinafter. We consider this model representative of most of the previous solutions for in-switch inference, which all use a single-stage approach and so experience the same problems that Henna seeks to address. We deploy the single-stage benchmark in our Tofino switch testbed with P4.

The quality of the classification result returned by both Henna and the benchmark is evaluated using standard performance metrics, *i.e.*, precision, recall and F1 score. These metrics are defined based on four key measures of a classification problem, *i.e.*, true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN), as follows.

- **Precision** captures the fraction of positive predictions that truly belong to that class, as TP/(TP + FP).
- **Recall** measures the quantity of positive samples that are predicted as positive, as TP/(TP + FN).
- **F1 score** is an harmonic mean of recall and precision that is widely used to compare model performances, and is computed as 2TP/(2TP + FP + FN).

For every metric, the final value is averaged over all classes in two ways, which we introduce next.

- **Macro average** is the unweighted mean of the metrics computed for each individual class; it expresses the model performance in a scenario where all classes have the same importance, *i.e.*, without the bias due to the representation of each class in the data.
- **Weighted average** is the mean of the metrics computed for each class, weighted using the number of samples of each class in the data; it represents the classification quality for a generic packet in the (possibly biased) target data.

4 RESULTS AND DISCUSSION

We evaluate Henna and the one-stage benchmark in terms of the metrics in Section 3.3, as well as by looking at their switch resource usage, as reported next.

4.1 Classification accuracy

The per-class performance of Henna against the benchmark in terms of precision, recall and F1 score is summarized in Table 1 and Table 2. Our solution improves the single-stage approach in all metrics, with relative gains of over 21% and 8% in terms of macro and weighted F1 scores, respectively.

A detailed breakdown of the classification performance of the 21 devices is shown in Figure 4. The improved accuracy of Henna is very consistent across all target classes, in terms of all metrics. The per-class precision of our solution is typically on par or better than that of the single-stage benchmark, with a couple of exceptions. Where Henna shines is in terms of increased recall, *i.e.*, in limiting the number of false negatives with respect to the benchmark. The F1 scores reflect the trends above: overall, Henna improves the compound metric in all classes with just one exception, where it still achieves very close performance to that of the one-stage model.

4.2 Resource Usage

It is important to determine if and to what extent the enhanced accuracy of Henna comes at a cost in terms of increased resource usage, processing latency, or power consumption. To collect all these statistics, we employ the Intel P4 Insight tool[3], which provides a detailed analysis of the compiled P4 programs in the target Intel Tofino ASICs, and of their mapping to hardware resources. The results are summarized in Table 3.

On average, Henna consumes just an added 3.40% of the total available resources with respect to the single-stage benchmark, for an overall 8.50% utilization of the hardware capacity. When compared to the standard P4 program for core L2/L3 switching, popularly known as `switch.p4`, Henna consumes just about 22% of what such a baseline program requires. These figures indicate that Henna leaves sufficient room in the switch to coexist with other legacy switching functions; this is especially true when considering that Henna would share M/A stages with legacy switching functions upon compilation, and that it would reuse constructs (*e.g.*, header information) already created by legacy functions within limited resources like PHV containers.

We also note how the one-stage model consumes 8 M/A units to implement one RF, whereas Henna only requires 2 additional M/A units to implement both a similarly sized RF plus 5 additional DT models. The reason is that our solution exploits for the first time also the egress pipeline, which lets trees of different stages coexist in the same M/A stages and overall leads to more efficient usage of the PISA structure.

With regard to latency, Table 3 shows that the inference time of Henna is much smaller than the packet processing delay of legacy forwarding functions via `switch.p4`; specifically, latency is 43.40% and 62.68% of that of `switch.p4` in ingress and egress, respectively. These figures are slightly higher than those for the one-stage benchmark, but well within the limit for line-rate operation as classification occurs much faster than packet forwarding.

Power consumption is also relatively low. In the ingress, Henna consumes less than the benchmark, and just 15.56% of the power required by `switch.p4`. At the egress, power needs are less than half of those of the legacy forwarding.

Figure 5 and Figure 6 offer a closer look at the switch resource usage. The figures detail the percent consumption of the total switch capacity by the one-stage model and Henna, for each type of resource. We separate the results in two plots to favor readability across consumption at different scales for two groups of resource types. While usage of most resources is fairly limited, in-switch inference tends to absorb specific types of memory: (*i*) the ternary content-addressable memory (TCAM) and the Ternary Match Input Xbar (TM Xbar) that are used for matching concatenated feature-level codes that include wildcards; and, (*ii*) the PHV used to store the stateless features used for classification through the PISA pipeline. In these cases, Henna can consume 30% to 45% of the switch resources. We remark however that these resources are not fully used in legacy forwarding operations, hence the utilization for inference is still compatible with normal functions. Moreover, the high consumption is an inherent limitation of the Planter model mapping

[3]https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

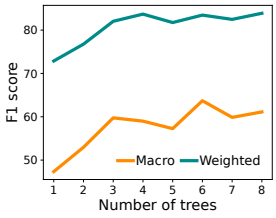


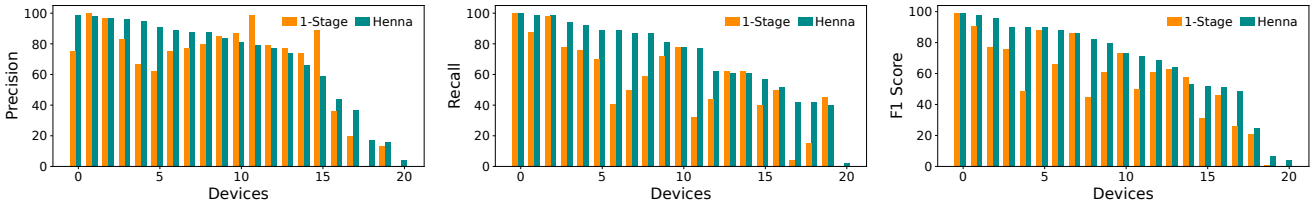
Figure 3: F1 score of single-stage RF model versus its size in trees.

Table 1: Macro scores of the three considered metrics, for the single-stage benchmark and Henna, with absolute and relative gains of our proposed solution.

Metric	1-Stage	Henna		
		Value	Gain	
			Absolute	Relative
Precision	65.38%	70.50%	5.12%	7.83%
Recall	55.50%	70.95%	15.45%	27.84%
F1 score	55.54%	67.50%	11.95%	21.52%

Table 2: Weighted scores of the three considered metrics, for the single-stage benchmark and Henna, with absolute and relative gains of our proposed solution.

Metric	1-Stage	Henna		
		Value	Gain	
			Absolute	Relative
Precision	84.50%	89.07%	4.57%	5.41%
Recall	77.25%	83.91%	6.66%	8.62%
F1 score	78.95%	85.52%	6.57%	8.32%



(a) Precision **(b) Recall** **(c) F1 score**

Figure 4: Comparison of Henna metric scores against the one-stage benchmark across the 21 target classes.

Table 3: Summary of the resource usage of the models. Power consumption and latency are expressed in % of those of switch.p4.

Resource	1-Stage	Henna
Overall (w.r.t. total)	5.10%	8.50%
Overall (w.r.t. switch.p4)	13.42%	22.27%
Match-Action units	8	10
Latency at ingress	35.42%	43.40%
Latency at egress	59.15%	62.68%
Power consumption at ingress	20.73%	15.56%
Power consumption at egress	–	42.29%

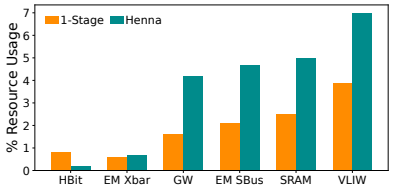


Figure 5: Breakdown of the resource usage as a % of the total available in the switch, for the least consumed resources.

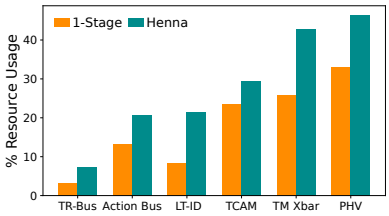


Figure 6: Breakdown of the resource usage as a % of the total available in the switch, for the most consumed resources.

strategy, rather than of the hierarchical approach: indeed, the problem affects the single-stage benchmark as well, which only spares 5%–10% of them with respect to Henna.

5 CONCLUSION

We presented Henna, a two-stage tree-based in-switch packet classifier. Results obtained in a real-world experimental platform with a sizeable use case show that a hierarchical solution improves classification performance while keeping resource usage under control.

We note that Henna is a proof of concept, which only implements a two-stage hierarchy and operates on simple packet-level features, and we plan to work on removing these limitations in the future. To this end, multiple directions are possible. A straightforward one is exploring use cases with multi-stage hierarchies that go beyond the two currently considered; this can be achieved by compressing multiple stages in both the ingress and egress pipelines, as well as by exploiting the multiple pipelines present in high-end programmable switches. A different and orthogonal option is changing the classification target of the models from packets to flows, and employing flow-level features for improved model performance; yet, this would require maintaining stateful registers to store flows and stateful features, which can be a challenge in constrained switch environments. Another interesting perspective is extending Henna beyond a single switch and distribute different stages or parts of a complex model or ensemble of models over different switches: this would enable distributed inference in the network where different nodes can contribute to a final and possibly more accurate classification result. Lastly, given that the resource usage footprint of Henna in terms of critical resources like PHV and TCAM is fairly high (owing to the Planter RF mapping scheme), the exploration of more efficient model mapping techniques is another research direction that will contribute to making RF to switch mappings more efficient and hence reduce the resource footprint of Henna.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement no. 101017109 "DAEMON", which supported the work of M. Gucciardo, and the Marie Skłodowska-Curie grant agreement no. 860239 "BANYAN", which supported the work of A.T.-J. Akem. This work was also funded by the CHIST-ERA grant no. CHIST-ERA-20-SICT-001 "ECOMOME", via grant PCI2022-133013 of Agencia Estatal de Investigación, which supported the work of B. Bütün.

REFERENCES

[1] Albert Banchs, Marco Fiore, Andres Garcia-Saavedra, and Marco Gramaglia. 2021. Network Intelligence in 6G: Challenges and Opportunities *(MobiArch '21)*. ACM, NY, USA, 7–12.

[2] Dario Bega, Marco Gramaglia, Marco Fiore, Albert Banchs, and Xavier Costa-Perez. 2020. AZTEC: Anticipatory Capacity Allocation for Zero-Touch Network Slicing. In *IEEE INFOCOM 2020*.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (jul 2014), 87–95.

[4] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada Solano, and Oscar Maurício Caicedo Rendón. 2018. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *J. Internet Serv. Appl.* 9, 1 (2018), 16:1–16:99.

[5] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. 2019. pForest: In-Network Inference with Random Forests. *CoRR* abs/1909.05680 (2019). arXiv:1909.05680

[6] Daniel Camps-Mur, Anastasius Gavras, Mir Ghoraishi, Halid Hrasnica, Alexandros Kaloxylos, Markos Anastasopoulos, Anna Tzanakaki, Gokul Srinivasan, Kiril Antevski, Jorge Baranda, Koen Schepper, Claudio Casetti, Carla Chiasserini, Andres Garcia-Saavedra, Carlos Guimares, Koteswararao Kondepu, Xi Li, Lina Magoula, Marco Malinverno, and Tezcan Cogalan. 2021. AI and ML – Enablers for Beyond 5G Networks. *5G PPP Technology Board* (2021).

[7] Gerald Combs. 1998. Tshark. *Wireshark* (1998). https://www.wireshark.org/docs/man-pages/tshark.html

[8] Nick Feamster and Jennifer Rexford. 2017. Why (and How) Networks Should Run Themselves. *CoRR* (2017). arXiv:1710.11583

[9] Anteneh A. Gebremariam, Muhammad Usman, and Marwa Qaraqe. 2019. Applications of Artificial Intelligence and Machine Learning in the Area of SDN and NFV: A Survey. *SSD 2019* (2019), 545–549.

[10] Hanxian He, Kourosh Khoshelham, and Clive Fraser. 2017. A two-step classification approach to distinguishing similar objects in mobile LIDAR point clouds. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* IV-2/W4, 67–74.

[11] Intel. 2016. Tofino Programmable Ethernet Switch ASIC. (2016). https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html

[12] Marios Evangelos Kanakis, Ramin Khalili, and Lin Wang. 2022. Machine Learning for Computer Systems and Networking: A Survey. *ACM Comput. Surv.* (Feb 2022).

[13] Jong-Hyouk Lee and Kamal Preet Singh. 2020. SwitchTree: in-network computing and traffic analyses with Random Forests. *Neural Computing and Applications* (2020), 1–12.

[14] Yingchi Mao, Andri Pranolo, Leonel Hernandez, Aji Prasetya Wibawa, and Zalik Nuryana. 2022. Artificial intelligence in mobile communication: A Survey. *IOP Conference Series: Materials Science and Engineering* 1212, 1 (jan 2022), 012046.

[15] Netronome. 2016. Netronome Agilio SmartNICs. (2016). https://www.netronome.com/products/smartnic/overview/

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011).

[17] Dan R. K. Ports and Jacob Nelson. 2019. When Should The Network Be The Computer? *(HotOS '19)*. ACM, NY, USA, 209–215.

[18] Dario Rossi and Liang Zhang. 2022. Landing AI on Networks: An equipment vendor viewpoint on Autonomous Driving Networks. *Huawei Technologies* (2022).

[19] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *HotNets'17* (Palo Alto, CA, USA). ACM, NY, USA.

[20] Carlos Nascimento Silla and Alex Alves Freitas. 2010. A survey of hierarchical classification across different application domains. *Data Mining and Knowledge Discovery* 22 (2010), 31–72.

[21] Giuseppe Siracusano and Roberto Bifulco. 2018. In-network Neural Networks. *CoRR* (2018). arXiv:1801.05731

[22] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. 2022. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *NSDI*. USENIX, Renton, WA.

[23] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. 2019. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing* 18, 8 (2019).

[24] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: A Data Plane Architecture for Per-Packet ML. *ASPLOS* (2022).

[25] Aaron Turner and Fred Klassen. 2013. Tcpreplay. https://tcpreplay.appneta.com/

[26] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. 2021. Programmable Switches for in-Networking Classification. In *IEEE INFOCOM 2021*.

[27] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. 2022. Mousika: Enable General In-Network Intelligence in Programmable Switches by Knowledge Distillation. In *IEEE INFOCOM 2022*. 1938–1947.

[28] Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *HotNets 2019* (Princeton, NJ, USA). ACM, NY, USA, 25–33.

[29] Xiaoquan Zhang, Lin Cui, Fung Po Tso, and Weijia Jia. 2021. pHeavy: Predicting Heavy Flows in the Programmable Data Plane. *IEEE Transactions on Network and Service Management* 18, 4 (2021), 4353–4364.

[30] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. IIsy: Practical In-Network Classification. *arXiv* (2022).

[31] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. 2022. Automating In-Network Machine Learning. *arXiv* (2022).

[32] Changgang Zheng and Noa Zilberman. 2021. Planter: Seeding Trees within Switches. In *SIGCOMM '21* (Princeton, NJ, USA). ACM, NY, USA, 12–14.