In-Network Machine Learning Using Programmable Network Devices: A Survey

Changgang Zheng, Xinpeng Hong, Damu Ding, Shay Vargaftik, Yaniv Ben-Itzhak, Noa Zilberman, *Senior Member, IEEE*

***Abstract*—Machine learning is widely used to solve networking challenges, ranging from traffic classification and anomaly detection to network configuration. However, machine learning also requires significant processing and often increases the load on both networks and servers. The introduction of in-network computing, enabled by programmable network devices, has allowed to run applications within the network, providing higher throughput and lower latency. Soon after, in-network machine learning solutions started to emerge, enabling machine learning functionality within the network itself.**

This survey introduces the concept of in-network machine learning and provides a comprehensive taxonomy. The survey provides an introduction to the technology and explains the different types of machine learning solutions built upon programmable network devices. It explores the different types of machine learning models implemented within the network, and discusses related challenges and solutions. In-network machine learning can significantly benefit cloud computing and next-generation networks, and this survey concludes with a discussion of future trends.

***Index Terms*—In-network computing; Machine learning; P4; Programmable data planes; Software Defined Networks.**

I. Introduction

CLOUD and edge computing are becoming powerful, attending to the increasing flow of data from users to cloud-based services. The new generation of network-processing technology revolutionizes network infrastructure as we know it and supports the increasing demand for network-traffic forwarding and processing.

Recent network devices are no longer just high-performance, but also programmable. Switch-ASIC (e.g., [1, 2]), network interface cards (NICs) [3, 4, 5, 6], and FPGA-based network devices [7] use a domain-specific language, P4 [8], to define and customize network protocols directly in the data plane. This programmability enables executing advanced network functions, and improves resources' utilization [9, 10]. This brings new opportunities to offload computations and applications to network devices: computing entirely within a programmable network devices is called *In-network Computing*.

Changgang Zheng, Xinpeng Hong, Damu Ding, and Noa Zilberman are with the Computing Infrastructure Group, Department of Engineering Science, University of Oxford (e-mail: changgang.zheng, xinpeng.hong, damu.ding, noa.zilberman@eng.ox.ac.uk)

Shay Vargaftik, and Yaniv Ben-Itzhak are with the VMware (by Broadcom) Research Group (e-mail: shayv@vmware.com, yaniv.ben-itzhak@broadcom.com)

Machine learning (ML) was shown long ago to be useful for traffic classification [11, 12] and for network anomaly detection [13]. These network-oriented ML tasks are typically deployed on servers or middleboxes [14]. However, as data volume increases so do the processing demands from the devices running the ML-based tasks.

The popularity of ML for networking, and the rising packet-processing demands have led to the suggestion that running ML algorithms on programmable network devices can significantly improve ML performance in terms of throughput and latency [15, 16]. Furthermore, it can help reduce memory consumption and communication overheads [17]. Consequently, a wide range of ML algorithms have been implemented in different ways on multiple types of programmable network devices.

In this survey, we distinguish between three forms of ML execution: *General ML*, *Network-Assisted ML*, and *In-Network ML*. *General ML* refers to the case where both the ML model training and decision making are on the server side, including deployments on hardware accelerators such as GPU. *Network-Assisted ML* uses network devices primarily for model training acceleration (faster parameter updates) and better feature collection (more detailed features collected within the network), while the inference takes place on the end host. *In-Network ML* refers to complete ML processes, either training or inference, done entirely in the network. Commonly today this refers to *In-Network ML Inference*, where trained ML models are running on programmable network devices, and inference decisions are taken within the network device, as shown in Figure 1. Offloading ML to programmable network devices is not straightforward, as there are computational and hardware resource constraints [16, 18].

This survey paper provides a comprehensive exploration of in-network ML. It introduces its background and history, discusses development, methodologies, and implementation techniques, as well as challenges and their proposed solutions. While this paper does not assume prior knowledge, we refer

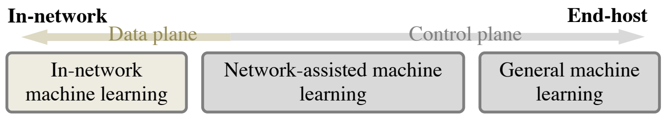


Fig. 1. The difference between general ML, network-assisted ML, and in-network ML (The arrow indicates where the decision is made).

the reader to previous surveys of ML in the networking domain [19, 20], and to previous surveys of programmable data planes [21, 22] for more extensive background, though neither discussed in-network ML.

In summary, this survey makes the following contributions:

- It clarifies important concepts in the field of ML and networking.
- It reviews the background for in-network ML and provides a comparison of target architectures and devices.
- It revisits research history, development, and taxonomy of in-network ML.
- It comprehensively reports the implementation details of in-network ML algorithms.
- It summarizes existing challenges for in-network ML and discusses potential solutions.

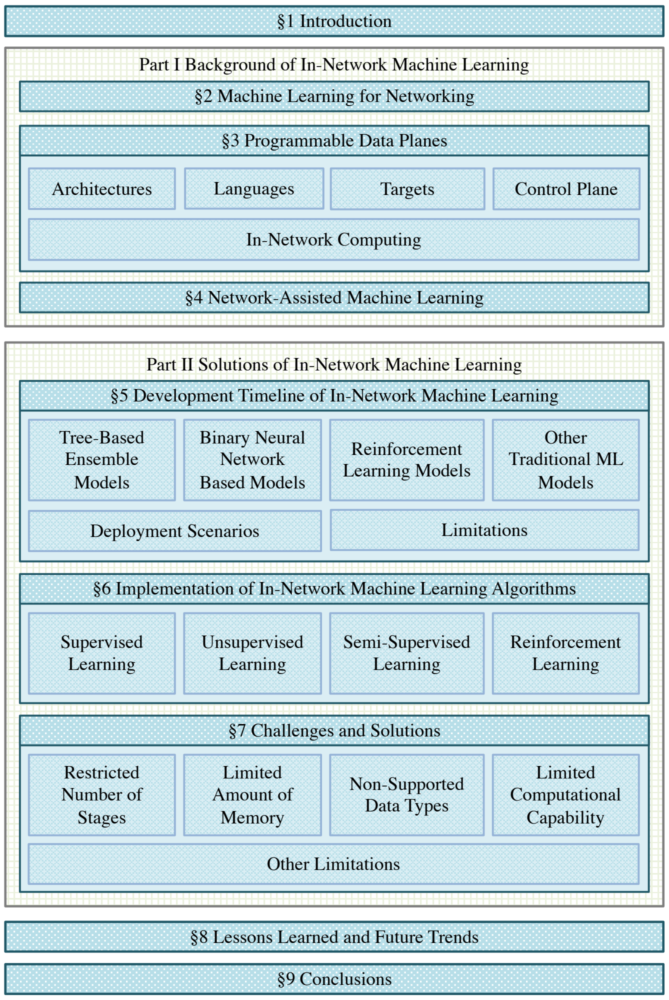

§1 Introduction
Part I Background of In-Network Machine Learning
§2 Machine Learning for Networking
§3 Programmable Data Planes
Architectures
Languages
Targets
Control Plane
In-Network Computing
§4 Network-Assisted Machine Learning
Part II Solutions of In-Network Machine Learning
§5 Development Timeline of In-Network Machine Learning
Tree-Based Ensemble Models
Binary Neural Network Based Models
Reinforcement Learning Models
Other Traditional ML Models
Deployment Scenarios
Limitations
§6 Implementation of In-Network Machine Learning Algorithms
Supervised Learning
Unsupervised Learning
Semi-Supervised Learning
Reinforcement Learning
§7 Challenges and Solutions
Restricted Number of Stages
Limited Amount of Memory
Non-Supported Data Types
Limited Computational Capability
Other Limitations
§8 Lessons Learned and Future Trends
§9 Conclusions

Fig. 2. Organization of the paper.

The rest of this survey is organized as shown in Figure 2. Section II introduces background of ML in the networking domain. Section III describes programmable data planes and existing targets. The concept of in-network computing and its applications for assisting ML are reported in Section IV. Section V, classifies in-network ML into several categories and reports their corresponding use-cases. Detailed implementations of typical in-network ML algorithms in programmable network devices are presented in Section VI. Section VII discusses the challenges of combining ML algorithms and programmable network devices, and introduces potential solutions. Section VIII summarizes lessons learned and discusses future trends of in-network ML. Finally, the survey is concluded in Section IX.

II. Machine Learning for Networking

A huge amount of data is flowing through today's networks. The volume will further grow [134, 135] with the introduction of new network technologies and the increased adoption of the Internet of Things (IoT), expected to reach billions of connected devices [19, 136]. The volume of data contributes to the development of powerful ML models aimed at improving the effectiveness of networks and networked-systems. ML is already used in many networking domains, such as traffic prediction, network analysis, and network automation. Table I presents nine application domains where ML is already used [19]: traffic prediction, traffic classification, traffic routing, congestion control, quality of service (QoS)/quality of experience (QoE) management, resource management, fault management, network security, and channel modeling. ML algorithms are used in these networking domains to perform classification, regression, and control tasks. They can be categorized into supervised learning (e.g., Naïve Bayes (NB) [137], Decision Tree (DT) [138], Support Vector Machine (SVM) [139], semi-supervised learning (e.g., Semi-supervised SVMs [140]), unsupervised learning (e.g., k-Nearest Neighbors (KNN) [141], Autoencoder (AE) [142]), and reinforcement learning (RL) (e.g., Q-learning [143], SARSA [144]) algorithms. Among these algorithms, as shown in Table I, SVM, DT, and Neural Network (NN) are the three most commonly used ML models for networking applications. Traffic Classification, QoS&E Management, and Network Security are the three popular networking use cases for adopting ML algorithms.

ML algorithms for networking use cases are dominantly deployed on servers or traditional accelerators (e.g., GPUs) [19, 145, 146, 147]. The introduction of programmable network devices, allowing the deployment of user programs within the data plane, has opened new venues for ML learning deployment. Programmable data planes enable high throughput and low latency packet processing within the network. However, these devices have inherent limitations compared to resource-rich environments like CPUs and GPUs. Moreover, the processing and programming logic differ significantly between these types of devices and require different frameworks. To gain a deeper understanding of in-network ML algorithms, we introduce programmable data planes in the following section.

III. Programmable Data Planes

Software-defined networking (SDN) [148, 149] decouples the operational planes of network devices, primarily separating the data plane from the control plane. While SDN supports network programmability, the initial focus was on changing forwarding and routing rules, with fixed or reconfigurable

TABLE I
SUMMARY OF ML WORKS IN NETWORKING DOMAIN. NOTE THAT THE WORKS IN COLUMN NEURAL NETWORK REFER TO ARTIFICIAL NEURAL NETWORKS (ANN) IN GENERAL WHILE THE WORKS IN COLUMNS BAYESIAN NETWORK AND RECURRENT NEURAL NETWORK FOCUS ON THE APPLICATIONS OF THESE TWO NETWORK ARCHITECTURES RESPECTIVELY. THIS TABLE SHOWS A SUBSET OF THE WORKS IN EACH AREA.

ML Algorithms	Support Vector Machines	Naïve Bayes	k-Nearest Neighbors	K-means	Decision Tree	Ensemble Trees	Neural Network	Bayesian Network	Recurrent Neural Network	Q-Learning
Traffic Prediction	[23]						[24][25] [26][27][28]		[29]	
Traffic Classification	[30][31][32] [33][34][35]	[36][37] [38][39][40]	[38][40] [41][42]	[43][44] [45][46][47]	[38][40] [39][30]	[36][30][48] [49][38][40][50]	[38][40][51]			
Traffic Routing							[52][53]			[54][55] [56][57]
Congestion Control		[58]	[59][60][61]		[59][60][61]	[59][60] [61][62]	[59][60] [61][62][63]			[64][65][66]
Resource Management	[67]						[68][69] [70][71][72]	[73][74][75]	[76][77][78]	[79][80] [81][82]
Fault Management	[83][84][85]			[86][87]	[83][84] [88][89]		[90][91] [86][92]	[93][94] [83][95][96]	[97]	
QoS & QoE Management	[98][99]	[98][99]	[98]		[98][99]	[98][100]	[101][98] [99][100]			[102][103]
Network Security	[104][105] [106][107]	[108]	[109]	[110]	[110][104] [111][105][108]	[104][112] [113][114][115]	[116][111] [117][107]	[115]	[118]	[119]
Channel Modeling	[120][121]		[122]		[123][124]	[122][125]	[126][127] [128][129]		[130][131]	[132][133]

data plane functionality [150]. The introduction of true programmability into the data plane [151], sometime later, has enabled changing the functionality of the data plane by users (e.g., network operators). Nowadays, network applications can be executed in an *in-network* manner, meaning that the program is implemented and executed within the data plane. In-network applications can adapt to increasing cloud network infrastructure demands, as they can potentially run at line rate, operating on all (or a selected subset of) incoming traffic. In this section, we present a brief overview of programmable network devices that function as the target devices for in-network ML and describe the workflow of programming them.

A. Protocol-Independent Switch Architecture

A programmable network device architecture defines the high level structure of a programmable network devices, and the interfaces between its major components. All current architectures are originated from and similar to the most common architecture called PISA (Protocol Independent Switch Architecture) [152]. Figure 3 shows PISA, the basic pipeline architecture for programmable data planes, developed based on the Reconfigurable Match Tables (RMT) model [151]. PISA allows network devices to customize packet processing within the data plane without hardware modifications, making them independent of vendor-provided, fixed sets of protocols.

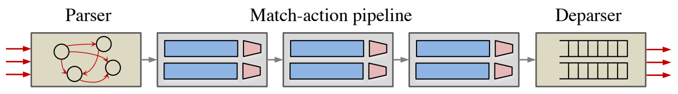


Fig. 3. Protocol-Independent Switch Architecture (PISA).

The PISA architecture has three building blocks: a parser, a deparser, and a match-action pipeline. The parser is a state machine that extracts a sequence of fields from a packet, called Packet Header Vector (PHV). The PHV usually contains fields from packet headers (e.g., Ethernet, IP, VLAN, TCP/UDP) and intrinsic metadata (e.g., the ingress and egress ports). The PHV can also include user-defined headers. The extracted vector is processed in a sequence of logical stages called match-action stages using match-action tables (M/A tables). M/A tables are fundamental units that lookup a key value (e.g., packet header) in a table, and map the resulting entry to a corresponding action to be taken on the processed packet. Each logical stage allows a fixed number of match-action operations, where a key (an input field from PHV or metadata) is looked up in a table, and a corresponding action is taken, enabling to process packets in a predetermined way. Finally, the deparser reconstructs PHV fields together with packet payload, before the packets are emitted.

There are several architectures built on top of PISA, including both open-source reference architectures and commercial solutions, such as Portable Switch Architecture (PSA, shown in Figure 4) [153], Programmable NIC Architecture (PNA, shown in Figure 7) [154], SimpleSumeSwitch [7], v1model (shown in Figure 5) [155], and Tofino Native Architecture (TNA, shown in Figure 4) [156]. Not all programmable data planes are PISA-based, for example, disaggregated RMT (dRMT) [157].

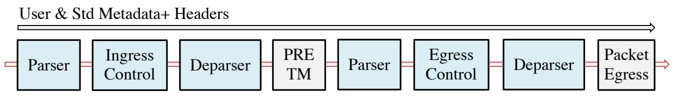


Fig. 4. PSA & TNA data plane architecture. PRE refers to the Packet Replication Engine and TM indicates Traffic Manager.

B. Data Plane Programming Language

Programming Protocol-Independent Packet Processors (P4) is a domain-specific language used for network packet processing [8]. P4 provides a flexible and programmable approach to customize packet parsing, matching, forwarding behaviors, and processing methodologies on network devices. A key feature of the P4 language is protocol independence, enabling support of a large number of protocols within a single device by change of a program. Furthermore, the P4 language supports flexible interaction between the programmable data plane and control plane, which enables coordination between the control logic and packet processing logic on network devices.

Programmable network devices are often programmed using P4, which does not have a single common compiler or development environment. Both commercial software development environments (SDE) from e.g., Intel, NVIDIA, and open-source solutions (e.g., BMv2) exist. The open source p4c [158] is the reference compiler, developed by the P4 community, but modified by vendors to their specific product. A P4 compiler takes a program written in P4 and compiles it into a binary that is executable on a specific target, based on a given data plane architecture. In general, the term P4 targets refers to the programmable network devices (e.g., specific switches or cards) and the term P4 architecture to the structure of the pipeline, defining the processing flow within a device. Currently $P4_{16}$ is the commonly used version of P4, and $P4_{14}$ is deprecated.

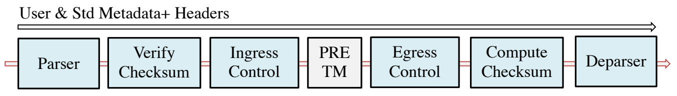


Fig. 5. v1model data plane architecture. PRE refers to the Packet Replication Engine and TM indicates Traffic Manager.

C. P4 Targets

The P4 language is designed to support a variety of packet processing targets, including smart network interface cards (SmartNICs), data processing units (DPUs), field programmable gate arrays (FPGAs), software switches, and hardware switch-ASICs.

P4-programmable hardware switches are available from multiple equipment vendors, such as Intel Tofino[1], NVIDIA Spectrum, and Cisco Silicon-One. DPU and SmartNIC vendors include NVIDIA BlueField, Intel IPU, AMD Pensando, Netronome NFP, and others. These vendors typically support a family of P4 programmable devices, across multiple generations.

In addition to hardware targets, there are multiple open-source implementations of P4-programmable software switch targets. Behavioral Model version 2 (BMv2) [159] is the most popular one and the reference P4 software switch. It supports multiple targets, including Simple Switch based on the v1model architecture and PSA switch based on PSA.

In the following, we discuss some commonly-used P4 platforms of different types:

1) Switch-ASIC: An Application-Specific Integrated Circuit (ASIC) is a chip designed for a particular purpose. Switch-ASICs are specialized devices designed specifically for packet forwarding, achieving high-throughput and low-latency switching. Programmable switch-ASICs introduce programmability into the switch pipeline, but do not compromise on performance. They are characterized by high throughput and low latency. Current switch-ASICs exceed 50Tbps and can process tens of billions of packets per second [160], with sub-microsecond latency.

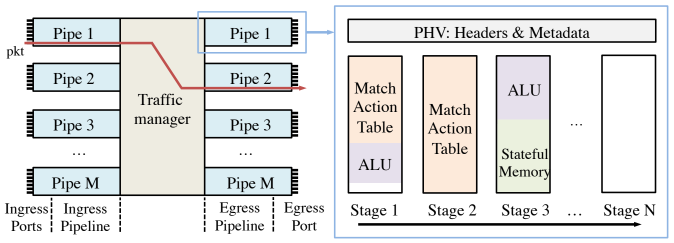


Fig. 6. Tofino-based switch data plane architecture.

Figure 6 shows as an example the architecture of Intel's Tofino switch-ASIC [156]. There are multiple *pipes* (each composed of an *ingress* and an *egress* pipeline), with multiple ports associated with each pipe. Pipes are analogues to CPU processing cores, and similar to them resources such as memory (e.g., CPU's L1 cache) and registers are not shared between pipes. Packets crossing between pipes happens only between the ingress and egress pipelines, typically through a shared buffer or a crossbar. This allows the pipes to guarantee throughput and latency performance, and prevents processing hazards and concurrent access issues. As a consequence, register, counter or meter values in pipe 1 (either ingress or egress pipeline) cannot be read by programs running in pipe 2. Incoming packets are processed in an ingress pipeline, before entering the traffic manager, and being processed in an egress pipeline. The egress pipeline is selected based on the packet's output port. Each pipeline contains a certain number of *stages*, which can execute operations such as *(i.)* using match-action to lookup keys in tables and take corresponding actions, *(ii.)* utilizing counters, meters, or registers, and *(iii.)* computing values using Arithmetic Logic Units (ALUs). Each switch-ASIC has limited hardware resources, which is the biggest challenge for offloading novel network functions into programmable targets as described in most existing works [161, 162, 163, 164]. Note that the integrated circuits of these devices contain billions of transistors, and that they are often limited by power and die size constraints. In fact, what is considered limited resources for ML is "just the right fit" for packet switching.

2) SmartNIC: A SmartNIC is an advanced type of a network interface card (NIC) that integrates processing capabilities, enabling the acceleration of network operations. NICs act as a connector between the network and the host, with incoming network traffic from the ports being sent to the CPU over PCI-Express (PCIe) bus. However, there is often a mismatch between the incoming network bandwidth, PCIe throughput, and CPU processing capacity. SmartNICs offload some of the processing from the CPU to the NIC in order to overcome this mismatch and free CPU cycles. SmartNICs are able to handle data rates of hundreds of Gbps, for example 2 ports of 100G. The main reference architecture for a programmable

[1]the development of this series was stopped in 2023

NIC is PNA [154], as shown in Figure 7. It consists of the main Parser, Pre Control, Main Control, and Main Deparser. Externs (specialized objects such as counters, meters, and registers) are included and can be used in the pipeline. SmartNICs tend to have more memory resources than a switch-ASIC, for example leveraging an external DRAM. Example SmartNIC targets include AMD Pensando [3], NVIDIA BlueField [4], and Netronome NFP [6].

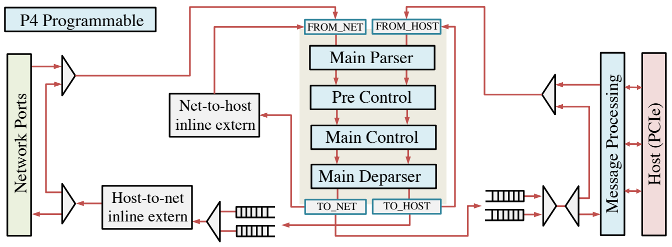


Fig. 7. Programmable NIC Architecture (PNA) version 0.5.

3) FPGA: FPGA is a configurable integrated circuit that can be programmed to perform specific functions by loading a binary programming file, providing flexibility and reconfigurability for implementing custom hardware designs. FPGAs were early demonstration targets for P4-based network devices, with works such as P4FPGA [165]. Other example targets include NetFPGA [166] running P4→NetFPGA [7] and AMD Alveo running OpenNIC [167]. Both are based on existing FPGA boards and provide a framework able to compile P4 programs into a dedicated packet-processing module. FPGA-based programmable network devices reach data rates of hundreds of Gbps, lower than high-end switch-ASIC but higher than CPU-based targets. FPGA targets also allow users to design their own P4 architecture. For example P4→NetFPGA [7] uses the SimpleSumeSwitch architecture, which uses only a single pipeline, without separation to Ingress and Egress, while the traffic manager (called Output Queues module) is located after the P4 pipeline.

4) Software Switch: A software switch is a network switch running as an application on a CPU. Software switches are often considered virtual switches, and are implemented using SDN principles, separating the control plane and data plane functions. Software switches run on standard CPUs, with switches supporting both x86 and ARM architectures, and don't require specialized hardware. To overcome performance barriers of CPUs, kernel bypass is often used. For example, T4P4S compiles a P4 program using v1model architecture and loads it to a Network Hardware Abstraction Layer (NetHAL) API [168]. T4P4S loads the compiled program to a data plane that is accelerated by DPDK (or ODP). Similarly, the popular Open vSwitch (OVS) is able to compile P4 programs with PSA architecture to software targets like PSA-eBPF and P4 DPDK [169]. To explore P4 functionality, the P4 behavioral model, BMv2 is widely used. It supports v1model and PSA architectures. A hybrid target is P4Pi [170], which runs P4 on a Raspberry Pi. While P4Pi supports both T4P4S and BMv2, it also provides a low-cost hardware target that can be used for education and research. Software switches typically have lower performance than hardware targets.

D. Control Plane

Control plane manages the runtime behavior of P4 targets via an Application Programming Interface (API). The API is supported by a device driver or an equivalent software component. P4Runtime [171] is a common control plane specification that makes it possible to control or configure the data plane of a device running a P4 program. Figure 8 illustrates the main control plane operations. It facilitates runtime control of P4 entities (e.g., M/A tables, counters, meters), for example by adding and removing table entries. There is typically also a packet I/O mechanism for streaming packets to/from the control plane. Reconfiguration mechanisms allow the loading of P4 programs onto the target's data plane.

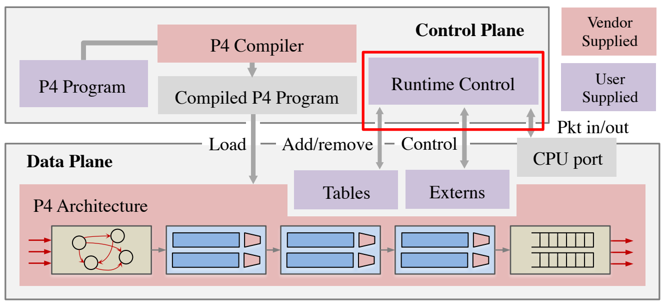


Fig. 8. Control Plane and Data Plane Interaction [171].

E. In-Network Computing

In-network computing refers to the offloading of programs or computation tasks to network devices, for example, programmable switches or SmartNICs. In-network computing takes advantage of network devices' high processing speeds and low overheads in physical space, energy, and cost, as they are already part of network infrastructure [10]. Realization of in-network computing allows networks to become part of available computing resources. It provides better integration of communication and computing resources when diverse application requirements need to be addressed [172]. Microsoft Azure highlighted the potential of in-network computing for telecommunication workloads [173], as it can efficiently process massive volumes of traffic directly within the network infrastructure. Their analysis identified cost efficiency, scalability and increased functionality compared with existing solutions. In response to what Microsoft identified as the main challenge for in-network computing, resource constraints, they have developed a resource elasticity solution [174].

In-network computing is implemented on any of the targets described in the previous section. It can be applied in various areas (e.g., caching, measurements, network services, and distributed systems). For example, NetCache [175] uses Switch ASIC to detect, index, cache, and serve hot key-value items in the data plane, providing significant throughput increase and latency reduction. P4xos [176] offloads a consensus protocol (Paxos) on programmable network devices (e.g., Switch ASIC, FPGA, and DPDK) and can effectively remove consensus as a bottleneck for distributed applications in data centers. NetChain [177] uses switch-ASIC to store data and process

TABLE II
SUMMARY OF NETWORK-ASSISTED ML ALGORITHMS. THE ALGORITHMS IN THE TOP PART OF THE TABLE FOCUS ON EXTRACTING INFORMATION IN THE DATA PLANE, TO ASSIST THE CLASSIFICATION EXECUTED ON THE END-HOST. THE ALGORITHMS IN THE BOTTOM PART OF THE TABLE FOCUS ON ACCELERATING HOST-BASED ML TRAINING PROCESS THROUGH IN-NETWORK AGGREGATION.

Scheme	Category	Algorithm	Language	Platform	ML Location[1]
SINET [178]	Defense Eavesdropping Attacks	Bayesian	P4	BMv2	Server (Controller)
FastFE [179]	Traffic Analysis	NN (Kitsune)	P4	Tofino	Server (Controller)
FlowLens [180]	Flow Classification	Multinomial NB, XGB, RF	P4	Tofino	Switch CPU
DPRO [181]	Routing Optimization	NN (RL)	P4	BMv2	Server (Controller)
ML-Pushback [182]	Defense Against DDoS	DT, Deep Learning	—	—	Server (Controller)
iSwitch [183]	In-Network Aggregation	Distributed RL	—	NetFPGA-SUME	—
SwitchAgg [184]	In-Network Aggregation	—	P4	NetFPGA and BMv2	—
SwitchML [185]	In-Network Aggregation	—	P4	Tofino	—
ATP [186]	In-Network Aggregation	Distributed DNN training	P4	Tofino	—
DAIET [9]	Data Aggregation	—	P4	BMv2	—

[1] The location of ML inference process.

queries in-band (within the data plane), which provides scale-free sub-RTT coordination in data centers.

IV. Network-Assisted Machine Learning

The successful introduction of in-network computing has led to attempts to apply in-network computing to ML applications. One type of effort focuses on accelerating ML frameworks using the network, while the frameworks are still deployed on CPUs and standard accelerators. We refer to these as *Network-Assisted ML*. The second type of effort focuses on offloading the actual ML operation, e.g., classification decision, to the network. We refer to these as *In-Network ML*.

This section mainly focuses on network-assisted ML, which has two main use cases: feature extraction and weight aggregation, as summarized in Table II.

Feature extraction. Due to the flexibility of programmable network devices and their potential for wide distribution on targeted networks, features can be extracted efficiently by these devices according to different use-case requirements and controllers' needs [187]. After extraction, they are sent to a controller or a defined ML server continuously for further processing. Features can vary from mean and standard values [179] of packet size to website fingerprinting [180]. Since they are extracted from inner networks, they are more representative and can provide more information and insights for ML use cases, compared to the features collected from the edge.

Weight aggregation. ML training can often be time-consuming. With the introduction of more powerful hardware accelerators, the bottleneck of ML training in data centers lies more in communication (weight aggregation from multiple different workers) compared to the training process, which can even cost around half of the whole training time. Conducting gradient aggregation on a switch (e.g. SwitchML [185], ATP [186]) instead of a specific aggregation server can reduce the communication overhead and thus speed up the training process.

V. Development Timeline of In-Network Machine Learning

Although in-network computing assists in the implementation of ML, in-network ML is different from network-assisted ML. Instead of deploying ML models on servers like network-assisted ML, in-network ML deploys them on the programmable data plane and mainly focuses on the inference process.

IIsy [188] highlighted three benefits of in-network ML: location, latency and load. Location, as network devices are already deployed within the network and any data for inference must go through the network. Latency, as in-network ML eliminates the latency introduced by CPU or GPU, and supports microsecond-scale inference decisions. Load, as any inference done on network devices saves processing cycles on a backend and can terminate traffic earlier, thereby reducing the volume of traffic deeper in the network.

Since 2017, researchers have tried to deploy ML algorithms directly in programmable network devices for classification. The work to date on in-network ML can be divided into four groups: 1) DT based models and tree-based ensemble models, 2) binary neural networks (BNN) based NN models, 3) RL models, and 4) other ML models. All in-network ML works are listed in Table III, and their development timeline is shown in Figure 9. There are three points worth highlighting: 1. The number of publications in this area has more than tripled since 2018. 2. Works in recent years tend to involve multiple ML models rather than a single model. 3. Several works went beyond model design and focused more on framework development (e.g., Planter [161], DINC [189], and Homunculus [190]).

While this section presents up-to-date works on in-network ML in terms of different types of ML models, sketching the skeleton of the in-network ML development timeline, the implementation of in-network ML solutions is illustrated in the next section in detail.

TABLE III
SUMMARY OF IN-NETWORK ML ALGORITHMS.

Scheme	Category	Algorithm	Language	Platform	ML location[1]
N2Net [191]	Packet Classification	BNN	P4	RMT-like Switch Pipeline	Data Plane
BaNaNa Split [15]	Classification	BNN	P4	SmartNIC	Data Plane
toNIC [192]	Data Classification	BNN	P4	Netronome SmartNIC	Data Plane
IIsy [16, 188]	Data/Packet/Flow Classification	DT, RF, XGB, KM, SVM, and NB	P4	NetFPGA-SUME, Tofino, BMv2	Data Plane
pForest [193]	Packet/Flow Classification	RF	P4	Tofino	Data Plane
SwitchTree [194]	Packet/Flow Classification	RF	P4	BMv2	Data Plane
Taurus [195, 196]	Packet/Flow Classification	M-RA[2]: DNN, SVM, KM, and LSTM	P4	Modified ASIC	Data Plane
Qiaofeng et al. [197]	Packet/Flow Classification	BNN (Federated learning)	P4/C	Netronome SmartNIC, and BMv2	Data Plane
N3IC [198, 199]	Data/Packet/Flow Classification	BNN	P4 P4	Netronome SmartNIC, NetFPGA, BMv2	Data Plane
BACKORDERS [200]	Packet/Flow Classification	RF	P4	BMv2	Data Plane
Planter [161, 201]	Data/Packet/Flow Classification	ET, SVM, NB, KM, KNN, PCA, AE, and BNN	P4	Tofino, Tofino2, BMv2, P4Pi	Data Plane
Bruno et al. [202]	Packet/Flow Classification	RF	P4	Netronome SmartNIC, and BMv2	Data Plane
IOI [203]	Classification	NN	P4	Modified ASIC	Data Plane
Clustreams [204]	Classification	k-NN Clustering	P4	Spectrum-3 switch	Data Plane
NetPixel [205, 206]	Image Classification	DT, CNN	P4	BMv2	Data Plane
pHeavy [17]	Flow Classification	DT	P4	BMv2, Tofino	Data Plane
OPaL [207, 208]	Control	Temporal-difference RL algorithms (SARSA)	P4	Netronome SmartNIC	Data Plane
QCMP [209]	Load Balancing/Control	Q-Learning	P4	Tofino, BMv2	Data Plane
Paolucci et al. [210]	DDoS Attack Detection	NN	P4	BMv2	Data Plane
INC [211]	Bot Detection, Botnet Inference	DT	P4	Tofino	Data Plane
Linnet [212]	Financial Prediction	NB, DT, RF, XGB	P4	BMv2	Data Plane
LOBIN [213]	Financial Prediction	KM, KNN, DT, RF, XGB	P4	BMv2, Tofino, Tofino2	Data Plane
P4Pir [214, 215, 216]	Runtime Model Update	DT, RF	P4	P4Pi	Data Plane
FLIP4 [217]	Runtime Model Update	Federated XGB	P4	P4Pi	Data Plane
MAP4 [218]	Packet/Flow Classification	DT, RF	P4	Netronome	Data Plane
Homunculus [190]	Parameter Tuning	DT, KM, SVM, and NN	P4	Modified ASIC	Data Plane
Mousika [219]	Data Classification & Prediction	DT	P4	Tofino	Data Plane
Mousikav2 [220]	Data Classification & Prediction	DT	P4	Tofino	Data Plane
Flowrest [221]	Flow Classification	RF	P4	Tofino	Data Plane
Ganesan et al. [222]	Packet Classification	DT	P4	BMv2	Data Plane
DINC [189]	Distributed Deployment	NB, SVM, DT, RF, XGB	P4	BMv2, Tofino	Data Plane

[1] The location of ML inference process.
[2] Map-Reduce Abstraction.

A. Tree-Based Ensemble Models

Deploying DT and tree-based ensemble models on programmable network devices was first proposed in 2019 [16, 193]. There are two main approaches, which are depth-based approach and encode-based approach.

The depth-based approach was proposed by pForest [193], and it encodes random forest (RF) into BMv2 and Intel Tofino in a hierarchical manner by using a match-action stage for each level in the tree. Evaluation of flow classification use case shows that the encoded data plane model can do inference at the line rate. This work also proposed an iterative training process to optimize tree models and used features. A similar work named SwitchTree [194] uses the same strategy to encode RF to BMv2. This work realizes in-network RF with a more complex feature extraction ability based on the UNSW dataset [223]. This method was then used in BACKORDERS [200] to do distributed denial-of-service (DDoS) attack detection on CICIDS2017 dataset. Compared to three previous works, Bruno et al [202], pHeavy [17], and MAP4 [218] use a more direct method with only $if()$ and $else()$ operations. Specifically, MAP4 argued not to choose other popular ML models. It encodes a simple and reasonably

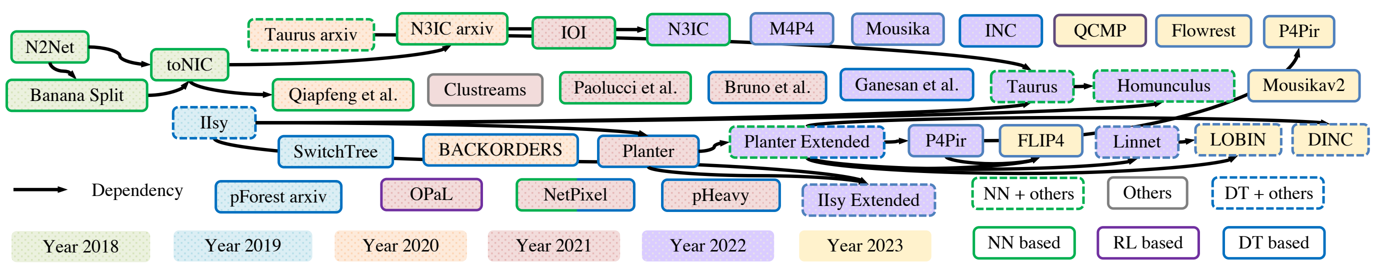


Fig. 9. Development History of In-network Machine Learning. All works can be found in Table III.

accurate DT model into a NIC and evaluates the encoded DT model by using intrusion detection and IoT classification use cases.

The encode-based approach is proposed by IIsy [16, 188] and Planter [161, 201]. This solution encodes each feature and uses an additional code-label table for the DT model, and it was tested on BMv2, Tofino, and FPGA. Planter extends the work from basic tree models to ensemble models, including RF, XGBoost (XGB), and isolation forest (IF), and it was tested by using anomaly detection on Tofino [201]. Planter also provides an automated in-network ML mapping framework for easy deployment of in-network ML algorithms. This mapping method has been applied by INC [211] and Flowrest [221] for bot detection and botnet inference. Based on Planter's framework, Linnet [212] applied this method to conduct future stock price movement prediction on a software switch and LOBIN [213] realized the similar functionality on resource-constrained commodity hardware switches. In an IoT use case, P4Pir [214, 215, 216] and FLIP4 [217] focused on tree models under Planter's mapping solution and explores how to do runtime model updates. FLIP4 [217] further combined the model to federated learning with diffusion noise to enable lightweight in-network ML deployment on the IoT edge while maintaining source data privacy. Homunculus [190], a parameter tuning framework that finds optimal data plane model parameters, also applied this method by using IIsy as a plugin. Other than IIsy's solution, encode-based DT can be simplified to only a decision table, which uses feature values as the input instead of mapping features into codes as the input. NetPixel applied this simplified encode-based DT algorithm to do image classification based on the BMv2 software switch [206] with the help of novel in-network image feature extraction techniques. pHeavy, Mousika, Mousikav2, and Ganesan et al. also mentioned this simplified method, by only using a decision table with range match, Longest Prefix Match (LPM), or ternary match, in their flow detection mechanism [17, 219, 220, 222]. Mousika focuses more on how to train a DT that best suits the data plane. It uses knowledge distillation to train binary DT, it then uses a similar method to map the tree by using ternary match.

B. BNN Based Models

Binary neural network (BNN) is a type of ML model first proposed in 2015 [224] and extended in the subsequent work [225]. It uses bit operations (XNOR and PopCount) to perform binary matrix multiplication efficiently and can provide a theoretical basis for deploying NN models in programmable network devices. XNOR-Net further develops the feed forward process of BNN and offers a concrete mathematical proof from weight binarization to input binarization [226]. N2Net provided a solution to implement the forward path of the BNN on a modern switching chip [191], proving the feasibility of mapping, providing an approach to leveraging the device parallelism, and offering a compiler to automatically generate the switching chip's configuration. However, this work did not discuss the deployment performance on commodity programmable network devices (though implemented on RMT [151]) and encountered the limitation of model size. The derivative work, Banana Split, extended the target devices of BNN from programmable switches in N2Net to SmartNICs, and proved that computing in programmable network devices can benefit end-host applications. When processing binarized NN models, Banana Split partitions the CNN, leverages the NN's layered-structure to be partly processed in the network, and explores when programmable network devices should be used as the co-processor of CPU [15]. Developed based on these efforts, toNIC realized binarized fully connected layers (the layers that connect each neuron to every neuron in the previous layer, facilitating comprehensive information exchange) on commodity SmartNICs, aiming at improving the efficiency and throughput of NN inference [192]. The result shows that the processing throughput can be improved by a factor of 10 with only a small NIC's resource overhead. However, the solution proposed in this work cannot function without servers and did not cover detailed evaluation in terms of throughput. The following work, N3IC, further proved that NN can be deployed on commercial programmable NICs to solve inference tasks which can reduce the cost of inference tasks required by packet monitoring applications. It implements the design on two different hardware NICs by using micro C and P4, and evaluates the design through three use cases e.g. traffic classification, anomaly detection, and network tomography. The result shows a 100x lower processing latency and 1.5x increase in throughput compared to CPU solutions. Similarly, Qiafeng et al. combined in-network BNN with federate learning to provide security service to multi-party edge device owners [197]. Evaluation results on two anomaly detection datasets show a multi-fold improvement in latency and communication overheads compared to other learning architectures without the help of in-network ML. By using a similar idea, NetPixel distributedly implements CNN on multiple software switches based on BMv2 and uses the generated model for image classification [205]. Based on previous research, Paolucci et al. [210] show a demo of NN on a software switch.

Instead of using existing network devices, Taurus [196] and Homunculus [190] added a custom hardware module to programmable network switches [195]. The added hardware can be based on a map-reduce abstraction introduced by Taurus using pipelined and SIMD parallelism for fast inference. It helps to realize several ML algorithms other than BNN and gains a processing speed that is three times faster than a server-based control plane. Similarly, IOI realized NN on programmable switches with the help of a plug-in transceiver module [203]. The module is designed specifically to perform non-linear operations such as matrix multiplication and non-linear activation.

In summary, there are two main technical approaches for realizing NN in programmable network devices. One is to use only the existing programmable network devices while the other is to add new modules to realize more difficult operations, such as matrix production and nonlinear operation, which requires area and power overheads.

C. *Reinforcement Learning Models*

Reinforcement learning (RL), as a key tool to online control systems in data-driven networks, was implemented on programmable network devices by a work named OPaL in 2021 [207, 208]. OPaL realized one-step temporal difference ML algorithm (e.g. SARSA) on Netronome SmartNIC. By using quantize fixed-point representations for values of actions and Tile-coding (a sparse-coding method for real-valued data [227]) based on *microengines* (MEs, a type of P4 extern plugin), the OPal can achieve significantly lower latency (more than 10 times lower) and throughput (around 3 times larger) compared to offline implementations. Different from OPaL, QCMP in 2023 [209] introduced two different approaches for in-network Q-learning implementation that do not rely on externs. The first method fully deployed the model inside the data plane, while the second solution employs the control plane to update the Q-table. QCMP's evaluation primarily focuses on the second approach, using traffic load balancing as a use case, and running on Tofino. Generally, there was so far limited research into in-band RL, which is a new research area in in-network ML and calls for further contributions.

D. *Other ML Models*

Several traditional ML algorithms, including SVM, NB, K-means (KM), and Principal Component Analysis (PCA) were implemented by IIsy [16, 188] and Planter [161]. To avoid complex operations, intermediate results of calculation are stored in M/A tables after training. IIsy made trade-offs between accuracy and feasibility, so as to realize the implementation of traditional ML models with complex operations on programmable network devices. In IIsy, more accurate algorithm implementation and less computation in network devices mean more memory consumption needed on network devices and more complex calculations needed for M/A table generation. IIsy's following work, Planter [161], generalized SVM, NB, KM, AE, and PCA as the Lookup Based Solution, and provided a general and modular workflow for similar models. All models supported by IIsy and Planter are able to run with a full line rate (6.4Tbps) and the same level of latency as an L2/L3 reference switch. Linnet [212] and LOBIN [213], by applying Planter, used NB, DT, RF, and XGB for future stock price prediction. Taurus [195] can also realize algorithms such as SVM and KM by using customized hardware based on map-reduce abstraction on programmable network devices. Its extension work Homunculus [196] did auto parameter tuning for embedded IIsy models as a plugin and also natively supports models in Taurus. Different from the previous three works, Clustreams [204] realized centroid-based clustering algorithms (e.g. KM) on programmable switches (Spectrum-3 Switch) by applying the combined quadtree [228] and ternary match-action tables (TCAM). Clustreams used quadtree to recursively divide the workspace and assign each workspace with a cluster where TCAM allows parallel searching of all quadrants during the inference process. The experiment result shows that Clustreams has a small overhead on the network latency and the switch throughput. The hybrid solution in Clustreams also contributes to reducing power consumption.

E. *Deployment Scenarios*

In-network ML can be used in different deployment scenarios, such as data center networks [188, 196], wide area networks (WAN) [201] and edge computing [214, 217]. Many of the works focus on the technology, yet it is natural to expect that DDoS mitigation will be deployed in WAN (dropping traffic close to the source), while latency sensitive use cases will be deployed at the data center or the edge (where computation time dominates over propagation time).

Four different in-network ML deployment types are identified in [188]: native switch, endpoint accelerator, SmartNIC, and a hybrid deployment.

A native switch is the most common type of deployment, where a network switch is running in-network ML in parallel with its traditional networking functionality, such as packet forwarding and traffic management. This type of deployment is beneficial as the switch is already deployed, thus there is no additional cost or space requirement, and inference can happen as traffic passes through the network. The disadvantage of such deployments is that the co-location with networking functionality leaves fewer resources for in-network ML.

A programmable network devices can also act as a "pure" endpoint accelerator, where a dedicated network platform is used for the sole purpose of in-network ML. This concept is similar to traditional accelerators, such as GPUs, except that the network platform is network-attached rather than residing on a PCIe bus. While this deployment allows all the device's resources to be used for in-network ML, it adds cost, power, and space overheads. An endpoint device also adds an extra hop to the traffic compared with a native switch.

The deployment of in-network ML on SmartNICs, which includes also DPUs, makes it possible to provide in-network ML on incoming traffic to an end-host. This deployment scenario is not very different from a native switch, as the

ML model is co-located with native NIC functionality, yet a SmartNIC typically has more memory resources than a switch-ASIC. Another difference is that a SmartNIC has an order of magnitude (or more) lower throughput than a switch.

A hybrid deployment was suggested [188] as a means to overcome model size constraints. It suggests deploying a limited-size model on a network device (native switch, end-point accelerator, or SmartNIC) and a large model at the backend. When an inference decision on a network device has low confidence, the message is forwarded to the backend for inference using the large model. This approach enables the processing of a large portion of traffic within the network, and is especially useful for anomaly detection where most of the traffic is classified as benign with high confidence and is not sent to the backend.

F. Limitations

Researchers have attempted to effectively port ML models to programmable network devices. However, these attempts were preliminary and limited in practical application. Firstly, the model accuracy in previous works was compromised as the scalability of the solutions was limited and complex features are hard to extract on hardware targets, leading to the limitations of model size and complexity [16, 193]. While the ML performance was good, it was often less than running an equivalent inference task on a traditional ML platform. Secondly, only a few types of ML algorithms were deployed on network devices and were applied to limited tasks, meaning that the current deployment scheme is hard to extend to other use cases or motivate the deployment of other ML algorithms. Thirdly, the early works did not explore how the implementation of their in-network ML algorithms varies between hardware targets. Finally, use cases were mostly limited to networking applications.

Beyond general limitations, there are also specific limitations to the implementation of specific ML models:

- **DT and tree-based ensemble models:** Although tree-based models are considered the most mature in terms of implementation at the data plane level compared to other models, existing solutions have their limitations. The depth-based approach [193, 194] exhibits poor scalability on hardware targets and sensitivity to the number of input features. When feasible, it can support a large number of branches and leaf nodes while consuming a small amount of memory, but the overall stage consumption remains high. The encode-based approach [188, 201] can accommodate a large number of features and trees, with generally lower stage consumption. However, in scenarios where tree depth is large, the number of branches and leaf nodes increases, and can consume significant memory. Additionally, the aggregation of individual tree results to obtain a final output poses a challenge for tree models. Memory/stage consumption becomes substantial when dealing with a large number of possible output classes or a large number of trees. These issues represent the primary limitations hindering the realization of larger tree-based models.
- **BNN based models:** While previous works present solutions of implementing the forward path of BNN and proved the feasibility of mapping, it was not shown that these solutions fit on current commercial switch-ASIC with acceptable performance and scalability [198]. Also, some of the early works do not mention the performance matrix of their solutions compared to other end-host deployed ML benchmarks.
- **Reinforcement learning models:** Although some simple value-based RL algorithms (one-step temporal difference RL) can be implemented in the data plane, more complex value-based RL solutions and policy-based RL solutions are still missing. The current implementation takes advantage of microengines (MEs), the control plane, or a software switch. Realizing it on commercial hardware targets without use of externs or the control plane remains an open question.
- **Other ML models:** Some researchers have proposed preliminary general solutions and ideas for ML deployment, but only 11 ML algorithms have been tried. Whether more ML algorithms are feasible is still unknown, and performance testing is not perfect yet. In addition, the general relationship between the size of the ML model, total resource consumption, and implementation accuracy, is not well defined.

While there are many ML algorithms already mapped to programmable network devices, there are still many questions to be discussed in terms of model adaptability, model scalability, deployment techniques, and deployment strategy. The practical large-scale applications are restricted in these ways and need to be further studied.

VI. Implementation of In-Network Machine Learning Algorithms

This section presents the ML algorithms implemented so far in the data plane. The section discusses how these ML algorithms are realized, and how their authors overcame challenges in their implementation. As the number of ML algorithms mapped so far to the data plane is limited, this section also suggests related ML algorithms that have the potential to be mapped to programmable network devices using similar techniques.

The choice which ML models to port to the data plane isn't based just on popularity or usefulness. It is largely driven by the (limited) amount of resources required to support the model within the network. Most of the models also exhibit a linear complexity in their data plane implementation, enabling them to operate at line rate. Only a few implementations, noted below, use external logic or recirculation techniques, which degrade throughput in favor of more functionality.

Figure 10 shows three classification methods of ML algorithms, including learning paradigms, learning models, and learning tasks. The bottom part of the figure illustrates the ML algorithms that are discussed in this section and how they are classified using learning paradigms. In this figure, the algorithms with a red background have been implemented in the data plane implementation already. The algorithms in purple have not been implemented yet.

TABLE IV

THE SYMBOLIC PRESENTATION OF IN-NETWORK ML ALGORITHMS WITHIN MATCH-ACTION PIPELINE. *logic* REFERS TO ADDITION OPERATIONS AND CONDITIONS. EACH ALGORITHM TARGETS AN INFERENCE TASK WITH n INPUT FEATURES AND k LABELS. EACH ALGORITHM VARIATION IS EXPLAINED IN SECTION VI AND CAN BE COMPREHENDED IN CONJUNCTION WITH THE FIGURES AND EQUATIONS. THE BRACKETS INDICATE THE INDEX OF DIFFERENT IMPLEMENTATION APPROACHES FOR THE SAME ALGORITHMS.

No	Algorithm	Symbolic Presentation
1	SVM (1)	$n \times [key : x_i; action : map(w_1^i x_i), map(w_2^i x_i), \ldots, map(w_m^i x_i)] \rightarrow logic$
2	SVM (2)	$(k \times (k-1)/2) \times [key : x_1, x_2, \ldots x_n; action : vote] \rightarrow logic$
3	NN (1)	$N_{layers} \times (X_i = XNOR(x_1 + +x_2 + + \ldots + +x_{nodes}, W_i) \rightarrow x_i = Count(X_i) \rightarrow x_i = SIGN(x_i), \ldots)$
4	DT (1)	$n \times [key : x_i; action : code] \rightarrow 1 \times [key : codes; action : y]$
5	DT (2)	$N_{depth} \times [key : PreviousID, Direction; action : Find\_next\_level, Set\_class]$
6	ET (1)	$n \times [key : x_i; action : code_i] \rightarrow N_{tree} \times [key : code_1, \ldots, code_n; action : leaf] \rightarrow 1 \times [key : votes; action : y]\ or\ logic$
7	ET (2)	$N_{tree} \times N_{depth} \times [key : PreviousID, Direction; action : Find\_next\_level, Set\_class] \rightarrow 1 \times [key : votes; action : y]\ or\ logic$
8	NB (1)	$n \times k \times [key : x_i, y; Action : P(x_i \mid y)] \rightarrow logic$
9	NB (2)	$k \times [key : x_1, x_2, \ldots x_n; Action : P(x_1, x_2, \ldots, x_n \mid y)] \rightarrow logic$
10	KM (1)	$n \times k \times [key : x_i, y; Action : (x_i - c_i^j)^2] \rightarrow logic$
11	KM (2)	$k \times [key : x_1, x_2, \ldots, x_n; Action : D^j] \rightarrow logic$
12	KM (3)	$n \times [key : x_i; Action : d_i^1, d_i^2, \ldots, d_i^k] \rightarrow logic$
13	KM (4)	$[key : x_1, x_2, \ldots, x_n, ; Action : y]_{1st\ hot}\ if\ not\ hit \rightarrow \ldots\ if\ not\ hit \rightarrow [key : x_1, x_2, \ldots, x_n, ; Action : y]_{kth\ hot}$
14	TD Learning (1)	$Input : Reward, State \rightarrow Extern(Predict\_action, Update\_policy) \rightarrow Outpout : State, Action$

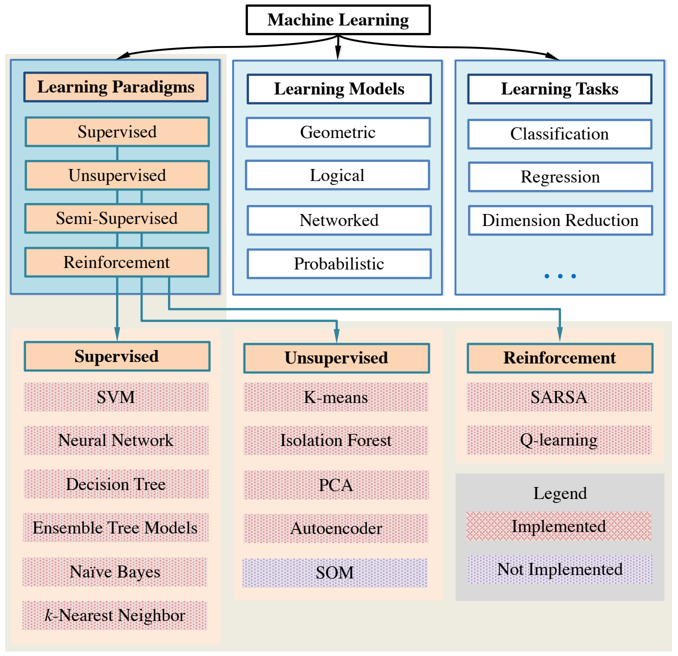


Fig. 10. Categorization of ML algorithms – The upper part of the graph includes three classification paradigms. The lower part of the graph generalizes the ML algorithms noted in this survey under the first classification paradigm.

Table IV shows a symbolic presentation of in-network ML algorithms. Some algorithms have multiple implementation approaches. Each algorithm targets an inference task with n input features and k labels. A detailed explanation of each implementation approach is shown next in the breakdown explanation of each algorithm.

A. *Supervised Learning*

Supervised learning, as a label-based learning method, is capable of building a model to make decisions for future events based on existing information (data) and annotated evaluations (labels) [229]. Datasets are input for training and their labels are used for supervising the model. The model is usually trained with labeled datasets to describe the mapping of the data space to the label space.

Supervised learning works in two phases, which are the training phase and the inference phase. In-network supervised learning offloads the inference phase in the data plane, and the training phase usually remains in the control plane due to the high operational complexity. It maps the trained model into a format that fits the M/A pipeline. Thus, the mapped model can label the data in the data plane based on the learned mapping.

Several supervised ML algorithms, including SVM, NN, DT, Ensemble Trees (ET) (e.g. RF, XGB, IF), NB, and k-Nearest Neighbor models will be further explained in this section.

*1) **Support Vector Machine (SVM):*** SVM is one of the typical classification algorithms, which performs well in solving nonlinear dataset problems and high dimensional pattern recognition problems [139] with a small sample size. The SVM model projects data into hyperspace and it aims at finding hyperplanes to perfectly divide the data. Each hyperplane separates the data into two sub-classes and keeps the data as far away from the hyperplane as possible. In the linear non-separable problem, SVM applies the kernel method to map data into high-dimensional feature space. For different datasets, different mapping patterns are required. The common kernel function includes linear, polynomial, Radial Based Function (RBF) [230], etc [231].

$$\begin{cases} w_1^1 x_1 + w_1^2 x_2 + \dots w_1^n x_n + d_1 = 0 \\ w_2^1 x_1 + w_2^2 x_2 + \dots w_2^n x_n + d_2 = 0 \\ \dots \\ w_m^1 x_1 + w_m^2 x_2 + \dots w_m^n x_n + d_m = 0 \end{cases} \tag{1}$$

For instance, Equation 1 shows how an SVM model is constructed by the linear kernel for a k classification task with n dimensional input data $X = \{x_1, x_2, \dots, x_n\}$ [232]. Each line of the equation represents a hyperplane as the border of two sub-classes. The model will generate m hyperplanes where $m = k(k-1)/2$. During the inference, SVM requires the support of floating-point values and negative numbers, multiplication operations, and comparison operations, which needed to be redesigned to fit programmable network devices with limited data types and limited mathematical operations.

To overcome the lack of floating point numbers, in-network SVM maps the needed values to a predefined range and approximates them to integers using $floor$, $ceil$, or $round$ operations [16, 233]. This method can tackle the challenge at the price of accuracy. To support multiplication, the mapped model uses M/A tables to store intermediate multiplication results, with the keys to the lookup being the operands.

Classified by the amount of memory required to store intermediate results, SVM has two mapping approaches [16, 161, 188], minimizing resource consumption on two different vectors. The first approach uses smaller tables, as shown in Table IV-1. This approach allocates a table to each feature. The action data of each feature table is a collection of intermediate results of the corresponding feature in the ith hyperplane ($map(w_1^i x_i)$, $map(w_2^i x_i)$, $\dots$, $map(w_m^i x_i)$). The mapping process here includes normalization, scaling, and data type transformation. Before the final decision, the model collects and adds all components to determine at which side of the hyperplane each input data point is located. For every input data $X = \{x_1, x_2, \dots, x_n\}$, being assigned to any side of a hyperplane means a vote for a specific class. The class with the largest number of votes will be output for each input data.

The second approach requires fewer computation resources, as shown in Table IV-2. This approach consumes $k(k-1)/2$ tables that store the result of all calculations. Each table determines the side (vote) of a hyperplane by using the set of features as the key. The second approach consumes more table entries compared to the first one but usually requires fewer calculations within the pipelines or stages, which can be useful when the range of feature values is narrow. To obtain the final decision, both methods need to compare the number of votes in each class. This can be done using either logic or M/A tables to get the final label.

*2) **Neural Network (NN):*** NN is a mathematical model which borrows the structure and function from biological neural networks [234]. The modern NN, as a statistical nonlinear modeling tool, stacks simple classifiers that operate in parallel to model the complex relationship between input and output from historical data. NN can effectively explore the patterns of data and extract complex input data features [235]. These simple classifiers, such as perceptrons, as shown in Figure 11, require floating-point input and weights to apply a list of mathematical operations such as multiplication, addition, activation functions (nonlinear), etc., to get the output, which is the major difficulties NN workflow faces for fitting in network devices with the limited amount of memory, limited data types, and limited mathematical operations.

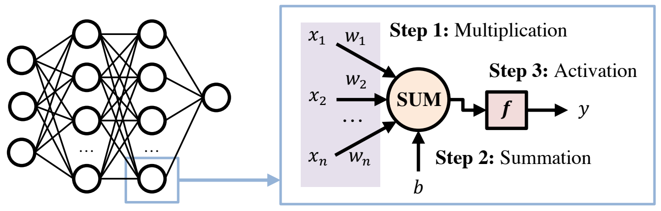


Fig. 11. A fully connected NN constructed by many perceptrons.

The first constraint, limited memory, can be solved by weight binarization. Binary inputs and weights can save up to 96% of the memory compared to using the floating-point [226] and help to deal with the multiplication operation that is not supported by several programmable network devices. The second constraint, limited data types, can be solved by round operation and binarization, which use integer or bit string to store weights and inputs instead of floating-point numbers. The third constraint, limited mathematical operations, can be solved by using XNOR operation to replace multiplication as a by-product of inputs and weights binarization and will be further proved in Section VII-D4.

With binary inputs and weights, the three key steps in the forwarding path of a perceptron in a neuron network are shown in Table IV-3 and Figure 11, which will be replaced by Step 1 (XNOR operation), Step 2 (Hamming weight), and Step 3 (comparison), as well as meet the constraints of programmable network devices [15, 191, 192, 197, 198].

While there are multiple types of Neural Networks, current programmable network devices implementations on existing devices include only Binary and Convolutional Neural Networks. More complex NN (e.g. DNN and LSTM) may become feasible by modifying switch ASIC (e.g. with additional MapReduce block [196]).

2.1. Binary Neural Network (BNN) is a type of artificial neural network (ANN) with binary weights and activations which facilitates the deployment of deep models on resource-limited devices [236].

BNN workflow with the data plane is shown in Figure 12, with arrows indicating data flow through the device. The workflow starts by extracting selected features from headers of incoming packets or local memory, such as meters, counters, and registers [15, 191, 192, 197, 198]. Then, it concatenates the features to bit strings as inputs. The weight of each neuron in the n layer is saved in a register as a bit string, and is read by the workflow. The device then executes an XNOR operation ($\overline{\oplus}$ in Figure 12) between the weight and the input bit string. The number of bits equal to one in each result will be counted by adapting the Hamming weight algorithm, and the model verifies if the number of bits equal to one is bigger or equal to half the length of the weights' bit string, as the sign operation. The verified result, a single bit, is stored in the least significant bit of the next layer's input bit string.

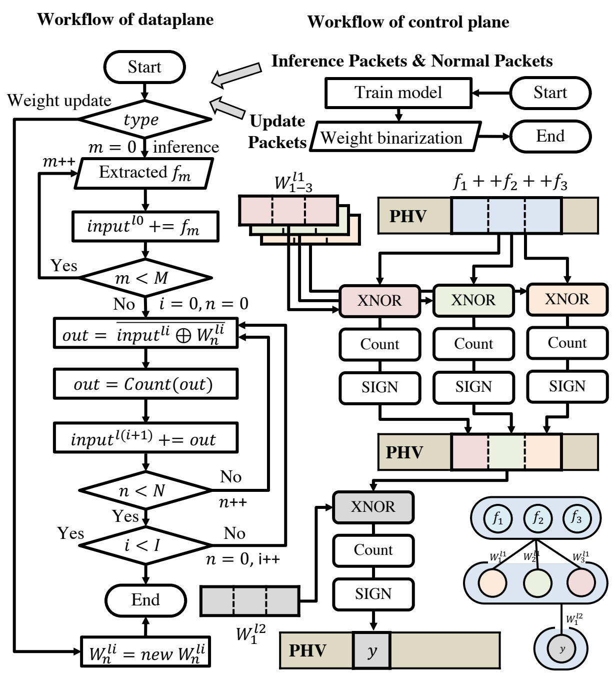


Fig. 12. Workflow of a typical BNN implementation within a programmable network devices. The left side of the figure depicts the NN inference process in the form of a flowchart, highlighting the specific operations required to implement it in the data layer. The top right corner shows the process of updating the network to the data plane. The right side of the figure displays a graphical representation of the implemented process in the data plane. A 4-neuron toy example is shown in the bottom right corner. Specifically, this network example requires three input features, three neurons in the middle layer, and one neuron in the output layer.

The workflow iterates previous steps for each layer until the last layer. There is a notation change as shown in Figure 12. Different from Figure 11, the weights in each layer equal the number of neurons in the next later. A simple toy example of a basic element of the workflow is shown in the bottom right corner of Figure 12.

The applied model and backpropagation method should ensure that the NN's weights are close to the range [-1, 1], which helps reduce the loss of information when applying the binarization technique. Benefiting from register-stored weights, BNN can apply the online update without stopping the device [197]. The trained binary weights will be packed and transferred to target data plane devices and trigger the weight update workflow.

2.2. Convolutional Neural Network (CNN) is a kind of feedforward NN with convolutional layers and a deep structure [237]. The convolutional operation enables representation learning and shift-invariant classification of input information according to its hierarchical structure. Currently, the convolutional layer has no data plane implementation, only the fully connected layer. The CNN model needs to be split into two parts in order to be accelerated on programmable network devices [198]. Take typical convolutional network structures as an example, the first part includes alternately stacked convolutional layers and pooling layers with a large number of parameters that need to be placed on local devices. In the second part, the fully connected layer, with a lower parameter size, can be converted to BNN and deployed into the network devices. The overhead of the model splitting process has been proved to be very little if the split is conducted carefully.

*3) **k-Nearest Neighbors (k-NN)**:* k-NN is an intuitive instance-based learning algorithm for supervised learning [238, 239]. The k-NN model first finds k closest training samples to the testing input in the feature space. Then, in a classification task, the k-NN categorizes the testing samples into the label with the largest number of closest training samples according to the majority voting rule, while calculating the mean value of k nearest neighbors as the output in a regression task.

$$y = \underset{c_j}{\operatorname{argmax}} \sum_{x^* \in N_k} I\left(y^* = c_j\right) \tag{2}$$

Equation 2 shows how k-NN is used for classification. In the equation, indicator function I equals 1 when $y == c_j$ and N_k is a set of x^* (x^* is the training sample with label y^*) that is included in the k nearest neighbors. In the forward process, the model calculates the distance of each stored data point to the input data to specify the set of neighbors. Generally, all the training datasets are stored in the model. However, in programmable network devices, it is hard to store such a large amount of data and find out which data are included in neighbors due to the limited memory and limited stages. Though it is difficult to realize this process directly during runtime calculation, the boundary of each output class can be calculated in advance and stored in a M/A table. The implementation of KNN in the data plane uses once again the value of features as the lookup keys of a M/A table, with the output being the classification result. This approach essentially performs a linear partitioning of the input feature space, such that each partition of the feature space is mapped to a class. The feature space slicing manner and table realization are similar to Clustreams' K-means solution (shown in Figure 16) [161, 204], and more information is provided in Section VI-B1, as the original source for this methodology.

*4) **Decision Tree (DT)**:* DT is one of the classical supervised learning algorithms, which can solve both classification and regression problems [240]. DT model has a tree-like structure that includes a root node, several internal nodes, and several leaf nodes [241]. The leaf node corresponds to the decision result, and the internal node (which can be named as branch) corresponds to the decision rule. When entering the DT, data starts from the root node, goes through the internal nodes, and heads for the corresponding branch according to the decision rules. Classification is completed when the data reaches the leaves. The tree structure of the DT equips it with a good application prospect in programmable network devices, especially when the calculation logic of a simple switch pipeline is similar to a tree structure [16]. During the inference, the DT does the comparison at the node iteratively to reach the next node or obtain the final output, which will be the main difficulty in making the DT workflow suitable for programmable network devices [16].

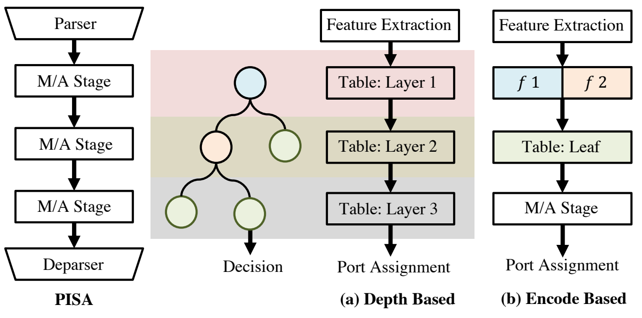


Fig. 13. The difference in mapping a DT to a match-action pipeline [201] between (a) Depth-based approach and (b) Encode-based approach.

There are two main DT mapping strategies to the data plane: a depth-based approach and an encode-based approach. i) Depth-based approach [193, 194, 202], as shown in Figure 13 (a), hierarchically maps the tree model into the M/A pipeline (as shown in Table IV-5, using N_{depth} tables). The model is executed within the pipeline layer by layer, until reaching the leaf node. The model starts by extracting the required features from incoming data. In each layer of the DT, the model compares a feature's value and a threshold. The next layer uses the node ID and the comparison result to extract the node information for the current layer. This approach consumes little memory, as the number of nodes in the tree is limited, but requires a lot more stages there is a dependency between the tree's depth and the number of stages. A simple version of the depth-based approach uses if and $else$ statements instead of M/A tables, which is intuitive but requires more lines of code [202] and does not save stages.
ii) Encode-based approach [16, 161, 188, 201], as shown in Figure 13 (b), breaks the hierarchical structure of the tree model. It encodes each feature according to the split value in each branch of the tree. Consequently, each branch can be treated as the result of one-time slicing on feature space. After slicing, each feature range is labeled with a code. A code-to-leaf mapping (M/A table) is used to map the combined codes of all features to the leaf node. This method uses a relatively small number of stages because feature tables have no dependency and can share stages (as shown in Table IV-4, requiring n parallel feature tables and one decision table). However, it usually requires more memory to implement the map operation. A simplified version of the encode-based approach uses only one leaf (decision) table for classification. Instead of a mapped code, the table uses feature values as input, and range-match is used for labeling [17, 205, 206]. This solution can save stages but consumes more table entries when the crafted tree model is complex, potentially exceeding the amount of memory available on a programmable network devices [161].

5) ***Ensemble Tree Models (ET):*** ET models construct a more powerful classifier than a single DT model with the help of several base DT models [242]. The combination of base models can statistically and computationally benefit the classification performance of the model. There are two main

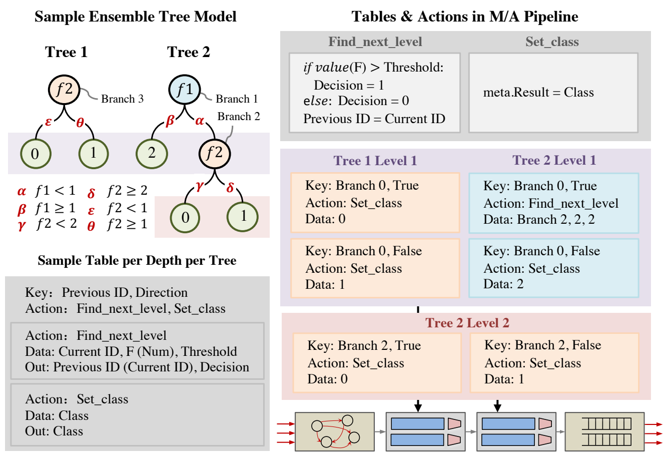


Fig. 14. Workflow of a typical ET model (depth-based approach, also shown in Table IV-6). The left bottom and top right sides of the figure show the two types of M/A tables required for this mapping approach, namely Sample Table per Depth per Tree (when the next node is still a branch) and Tables & Actions in M/A Pipeline (when the next node is a leaf node). The bottom right corner of the figure displays the mapping result of the example model in the data plane, which is divided into two layers. The first layer consists of one Sample Table per Depth per Tree (branch node f_2) and three Tables & Actions in M/A Pipeline tables (leaf nodes 0, 1, 2), while the second layer consists of two Tables & Actions in M/A Pipeline tables (leaf nodes 0, 1). A 2-tree toy example is shown in the top left corner. The example model includes two input features, f_1 and f_2, and three possible outputs: 0, 1, and 2.

ensemble techniques: bagging and boosting. Within the data plane, the main difference between these two techniques is in the final stage logic used to assign the label [201]. Before the final stage, the data plane implementation is very similar for both types of ensemble tree models.

Similar to the mapping of a DT, the implementation of the ensemble model uses the extended depth-based approach [193] and the encode-based approach [161]. They are shown in Table IV-6 and IV-7. The depth-based approach, as shown in Figure 14, can share stages when executing different trees at the same depth. For an N_{tree} trees ensemble model with maximum depth N_{depth}, up to $N_{tree} * N_{depth}$ tables are needed. Among them, M/A tables of the same layer's depth can share stages. The encode-based approach, as shown in Figure 15, shares stages for every feature table and every tree table. For a N_{tree} trees ensemble model constructed by using n features, this approach requires n feature tables and N_{tree} tree (code) tables, where each feature table can share the same stage within the pipeline and tree tables can share the other stage, meaning that only three logical stages are needed: features lookup, tree-result lookup and label assignment.

5.1. Bagging: The bagging technique, also named bootstrap aggregating [243], is a parallel learning sampling technique that trains a series of independent, homogeneous models in parallel, and uses the aggregate output of each model according to some strategy. Different trees are trained on different data. RF is one of the typical models generated by the Bagging technique. In RF every tree model has the same importance. In the classification problem, the base model (DT) votes to get the final result. In the regression problem, the mean of base models is the result. Implementation wise, both the depth-

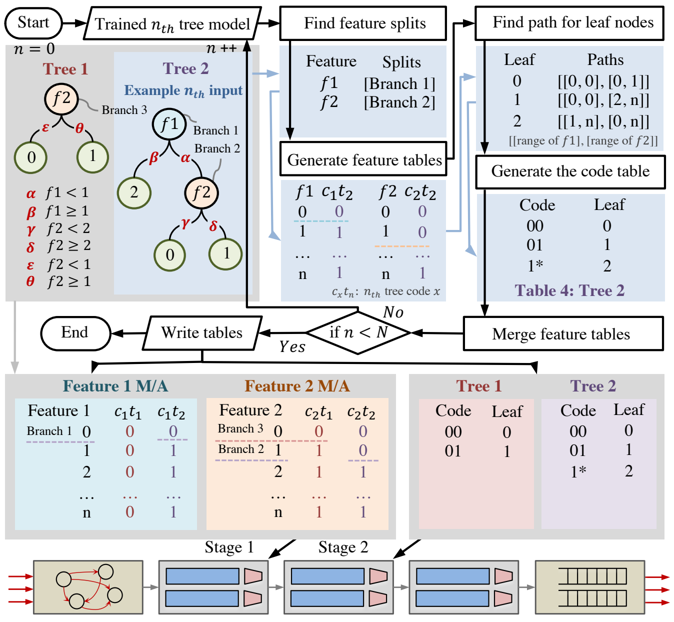


Fig. 15. Workflow of a typical ET [201] (encode based approach, also shown in Table IV-7). The flowchart in the figure demonstrates the specific process of mapping the model to the data plane using the encode-based method, including four main steps: finding feature splits, generating feature tables, finding paths for leaf nodes, and generating the code table. The resulting tables are shown at the bottom of the figure, including two Feature Tables, two Tree Tables, and a Decision Table (or if-else logic) that is not displayed in the figure. A 2-tree toy example is shown in the top left corner. This is the same as the tree model shown in Figure 14.

based approach and encode-based approach can be used to realize RF. Each tree is implemented individually as in DT, most efficiently by parallelizing different trees (as explained above). In the final logic, the model can use either a M/A table to map from votes to class or use logic to compare each class' vote [201].

5.2. Boosting: Boosting is a serial learning sampling technique, which samples data under the distribution based on the learning results of the last iteration [244]. Each subsequent tree is trained to estimate the errors of the previous trees. To take XGBoost (XGB) [245] as an example that currently has two ways to implement the model within the network [161, 201]. In the first implementation method, all trees add up all the probabilities that belong to a specific class to form the overall prediction for that class. The final probability can be calculated in a similar way to the vote calculation in the bagging technique. This method could suffer from an accuracy loss as floating-point numbers may not be supported by the programmable network devices, therefore the probability is approximated and mapped to a new integer space. Another approach encodes the probabilities in each tree [161] and then uses a table to map all the encoded probabilities to the final label. This method is rigid and does not exhibit accuracy loss.

*6) **Naïve Bayes (NB)**:* As a statistical classification method, NB is the general term for algorithms based on Bayes' theorem [246] shown in Equation 3, which is one of the simple classical Bayes models.

$$P(x_i|y) = \frac{P(y|x_i)P(x_i)}{P(y)} \tag{3}$$

Based on the posterior probability introduced by Bayes' theorem, NB tests the probability of each set of inputs with each label, under the assumption that each input feature is independent. As shown in Equation 4, the label with the highest probability is the prediction result.

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i \mid y) \tag{4}$$

In general, calculating the conditional probability of a certain class given a set of input features requires multiplying all the conditional probabilities for each input feature corresponding to a particular output class with the posterior probability of the class, as shown in Equation 4. The concatenated multiplication operation and the floating-point values used in the equation make it hard to implement NB on programmable network devices, where multiplication and floating point numbers are not natively supported.

To implement NB in the data plane despite these two limitations, a model mapping uses M/A tables and trades resources for accuracy [16, 161, 188]. A Naïve Bayes implementation uses M/A tables to store probabilities and the results of multiplying probabilities. For example, using input feature x_i and class y as a key to the table results in the value $P(x_i|y)$. To handle the lack of floating point numbers, the results are cast to integers (multiplied by 10^n, the required accuracy) or fixed-point numbers.

Two mapping approaches of NB use the above methodology [16]. The first approach, shown in Table IV-8, uses nk (or $n(k+1)$) M/A tables, one table for each conditional probability. The lookup keys to each table are the value of a feature, and the action is the conditional probability for a given class ($P(x_i|y)$). The probability of class y, $P(y)$ is stored either in a M/A table or in a register. Another option is to hold $P(y)$ as a constant in the P4 program. To multiply the conditional probabilities of the different features, another table can be used per label. The final classification uses conditions to pick the high-probability label, though a M/A table can be used for the same purpose.

An optimization to this approach was proposed in Planter [161], using the $log()$ operation to avoid multiplication [161], using addition instead in the decision stage.

The second approach shown in Table IV-9, moves the burden of multiplication from the network devices to the compilation stage [16]. This approach uses k M/A tables, one for each class. The key to each table is the collection of all input features, and the output action is the probability of a given class. The probabilities of all classes are compared in the last stage to determine the label. The calculation of all the conditional probabilities is done in the compilation stage when all possible values (quantized) are generated to populate the tables. As a result, the tables generated by the second approach tend to be significantly larger than in the first approach, but the number of tables is smaller.

B. Unsupervised Learning

Unsupervised learning is another type of ML algorithm used to explore hidden patterns in unlabeled data [229]. Also as a statistical method, unsupervised learning algorithms can use unmarked input data to explore data structures and patterns automatically without explicit supervision.

In-network unsupervised learning algorithms transfer the learned data structure into M/A pipeline and apply it to new incoming data mainly for clustering or dimension reduction tasks. In-network unsupervised learning does not learn in the data plane but applies ML models trained by unsupervised learning methods to the data plane.

Common unsupervised learning methods include KM, Isolation Forest (IF), PCA, Self-Organizing Map (SOM), AE, etc.

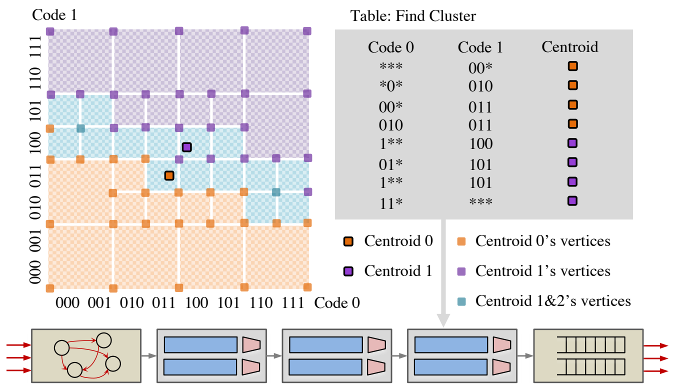
*1) **K-means**:* KM algorithm is an iterative method that aims to divide the dataset into k clusters in which each observation is assigned to the cluster with the nearest mean. The distances between data points from the same cluster need to be as short as possible while observations from different clusters are kept as far as possible [247].

$$D_i = \sqrt{\left(x_1 - c_1^i\right)^2 + \left(x_2 - c_2^i\right)^2 + .. \left(x_n - c_n^i\right)^2} \tag{5}$$

Equation 5 shows how to calculate the distance between each set of input to each cluster, where $X = \{x_1, x_2, \dots, x_n\}$ is the input and $C^i = \{c_1^i, c_2^i, \dots, c_n^i\}$ is the centroids of cluster i. The complex mathematical operations, including square and square root, make it challenging to fit on programmable network devices.

To overcome these difficulties, there are three similar implementation approaches, which can be classified by the overhead of calculations and the number of table entries. The first approach, as shown in Table IV-10, requires the most calculations and the smallest number of table entries [16, 161, 188]. In the implementation of this solution, for each table, the input key is the feature value and cluster-ID, and the action data is the square distance of the feature to that cluster. In the decision logic, the total distance is calculated by summation. The cluster-ID with the closest distance (decided through comparison) is the output class. Note that root square operation is not required as the comparison is of squared distances. The second approach requires the minimum computation and the maximum number of table entries, as shown in Table IV-11, where n M/A tables use the input features sets, $x_1, \dots x_n$, as key, the distance to each label cluster as action data. The final logic conducts comparison to determine the output label. The main difference from the first approach is that the first approach is unidimensional while the second approach is multi-dimensional per table. The third approach is a compromise between the first two approaches. As shown in Table IV-12, this approach uses n M/A tables. In each table, the key is the feature value and the action data is a collection of squared distances of that feature between each cluster (unidimensional). The final logic is similar to the first approach.

As shown in Table IV-13, there is another different approach for offloading centroid-based unsupervised learning algorithms with the help of Quadtree (which applies Ternary match) [161, 204]. Before analyzing its detailed implementation, one needs to understand the nature of this type of an algorithm. KM and other centroid-based unsupervised learning methods do classification based on the distances between the input data point and each centroid, which means that boundaries dividing each class can be found in feature space. This fourth method iteratively divides the feature space into small pieces until all the input values that lie in it belong to a specific class. As shown in Figure 16, for a two-dimensional input toy example, this method uses fine-grained squares to describe the boundaries and uses coarse-grained squares to represent spaces inside. How detailed it depicts boundaries depends on the trade-off between accuracy and memory overhead. After the feature space is well split, this method stores the information of each piece in the TCAM table. There are two methods to index the feature space. The first one, as shown in Figure 16, uses consecutive codes to index (encode) each input feature after assigning each block to a class. This method is intuitive. Although Exact-to-TCAM is a classical network problem, it is not easy to find the most suitable and efficient way to conduct the transfer. In comparison, the second one uses the Quadtree index purely, as shown in Clustreams' work [204]. In this method, the feature space is easier to be transferred to the TCAM table but requires pre-processing before the packet is sent to programmable network devices.



Code 0	Code 1	Centroid
***	00*	🟧
0	010	🟧
00*	011	🟧
010	011	🟧
1**	100	🟪
01*	101	🟪
1**	101	🟪
11*	***	🟪

Fig. 16. Workflow of a centroid-based (K-means) unsupervised learning [204].

In many cases, the input is biased and not evenly distributed in feature space, which creates some "hot zones". For a higher processing speed, the algorithm can split the previously large TCAM table into multiple smaller ones according to how hot the area is. If the input hits a "hot zone", it will be classified and the algorithm will skip the following TCAM tables. Otherwise, the input will be moved to the next table with the less hot area until it hits a table [204]. Note that this approach is useful only on targets where stages in the pipeline can be skipped.

*2) **Principal Component Analysis (PCA)**:* PCA is a statistical method that is commonly utilized to reduce the dimensionality of large data sets while preserving most of the data variation [248]. It computes the cumulative projection of each component on each data point onto new components to conduct dimension reduction. Planter [161] is the first work that offloads this model into the data plane.

The forward process of the PCA algorithm requires matrix multiplication and matrix division operation, which can be interpreted as $Y = WX$. For the input with n original dimensions, the new dimension y_j can be calculated by $y_j = w_j^1 x_1 + w_j^2 x_2 + \dots w_j^n x_n, \{j \in \mathbb{Z} \mid 1 \leq j \leq k\}$, where k is the total number of new dimensions. Under this condition, Planter's implementation uses M/A tables to store the intermediate result of the multiplication operation on programmable network devices with limited mathematical operations, in a manner that is very similar to the SVM solution (Section VI-A1). Specifically, for each input feature x_i, there will be a M/A table with input x_i and output $w_1 x_i, w_2 x_i, \dots, w_k x_i$, storing intermediate results. After all intermediate results are extracted, pipeline logic performs multiplication operations to calculate the value of new dimensions $y_1, y_2, \dots, y_k$. In this method, the resource overhead is the function of the range and the number of input features. When there are too many input features or the range of features is wide, it would be hard to realize it on programmable network devices with limited memory.

*3) **Self-Organizing Map (SOM):*** SOM is a special type of unsupervised ANN model trained with a competitive learning algorithm, which is intended for the purpose of dimensionality reduction [249]. It typically generates a two-dimensional representation from multidimensional inputs while conserving the topological space in the meantime. A SOM forms a map for the input distribution where samples are grouped based on the similarity between one another. Multidimensional data can be visualized and interpreted with this technique.

In a SOM, one or two-dimensional space is evenly filled in by neurons before training, which means the number of trained neurons is large. Thus, in the forward process, there is a significant overhead of computation and comparison as the model needs to compare the distance of input between every two neurons. This makes it hard to implement a SOM on programmable network devices with its limited resources as the comparison operation has a high dependency and potentially requires many stages. In general, the structure of SOM looks a lot like NN, while an implemented functionality will resemble PCA and Autoencoder. To the best of our knowledge, the implementation of SOM in the data plane has not been realized.

*4) **Autoencoder:*** Autoencoder is typically used to perform the task of representation learning through data encoding and decoding as a family of unsupervised ANN architectures [250]. By design, an autoencoder is capable of learning efficient compression of a set of data to obtain a knowledge representation of the original input, as well as subsequent reconstruction based on the representation. In general, it discovers the underlying structure of data and can be applied for various purposes, such as dimensionality reduction, feature extraction, image denoising, and anomaly detection.

$$X_{new} = XW + Bias \tag{6}$$

The autoencoder model usually has a funnel-like structure. For different use cases, the depth of the encoder network and the decoder network varies. The full-size autoencoder is usually hard to be implemented due to limited stages and mathematical operations. However, a model with only a low number of hidden layers is potentially realizable [161]. Equation 6 shows the forward path of a one-layer autoencoder, which shares the same logic as Equation 1 ($WX + B = 0$). Its structure is very similar to PCA with similar limitations. In the pipeline, biases represented as B are stored either within a M/A table or a register. Following this, for each input feature, a M/A table is utilized, taking the input x_i and producing a collection of intermediary outcomes $w_1 x_i, w_2 x_i, \dots, w_k x_i$ (where k denotes the number of encoded nodes). Subsequently, while reading these intermediate results from the M/A tables, simultaneous operations are conducted to compute values ranging from y_1 to y_k.

*5) **Isolation Forest (IF):*** IF is an unsupervised learning algorithm based on DTs and designed for anomaly detection which is capable of identifying anomalies rather than normal observations [251]. IF isolates the outliers by randomly selecting features and randomly sub-sampling data. In trees, the data points with shorter paths have a higher probability of being anomalies as they are easier to be separated. The general method for offloading IF to programmable network devices [161, 188, 201] is very similar to tree-based ensemble models (Section VI-A5) and both depth-based and encode-based approaches can be used to realize the IF. The operational mechanism inherent to this model is predicated on the principle that a node of greater depth signifies a correspondingly diminished probability of being classified as an anomaly. The main difference between IF and previous RF implementations resides in the outcome of the table that represents leaf nodes. In IF, instead of the tree's vote, the table's output value is the depth of the leaf node. Subsequently, the final decision table or logic is based on the aggregation of these values across all trees to output the label. Given that trees in IF usually exhibit substantial depths and a high number of leaf nodes, quantization can be employed to constrain the depth values of these leaf nodes. This serves to delimit decision table size and thereby avoid entry explosion. Trimming all the nodes above a threshold can save the data plane resources. Still, it is still hard to conduct resource control for in-network IF for two reasons: 1) IF usually trains the models without any constraints, which makes the trained model itself complex. 2) IF counts the number of branches it passes along the route. As a result, the number of possible values on leaf nodes is large, making the decision table large.

C. Semi-Supervised Learning

Semi-supervised learning is an approach situated between unsupervised learning and supervised learning. It refers to solutions for the training cases with only a small portion of labeled data and a large amount of unlabeled data [252]. In general, semi-supervised learning algorithms are needed when the collection of labeled data is time-consuming and labor-intensive.

Semi-supervised learning algorithms can be divided into self-training, graph-based semi-supervised learning, semi-supervised SVM, etc. To date, no in-network computing work has explored this learning paradigm.

Semi-supervised learning takes advantage of training with a low portion of labeled data and may use the aforementioned offloading techniques to deploy the trained algorithm to programmable network devices. Generally, semi-supervised learning is a key concept in the training phase, which can greatly help in-network ML algorithms to train in the case of large data volume. Semi-supervised learning is stable with high efficiency, but sometimes it suffers from low accuracy.

D. *Reinforcement Learning (RL)*

RL constructs a strategy with the purpose of controlling agents to maximize the expectation of cumulative rewards. The RL strategy is learned from the interaction between agents and the environment. This type of an algorithm is inspired by the behaviorism theory. In psychology, this theory explains how organisms gradually form the most rewarding habitual behaviors under stimulation or punishments from the environment [253].

RL can be divided into value-based, policy-based, and actor-critic. Two traditional and common value-based RL algorithms were the focus of in-network ML research: Q-learning and SARSA. OPaL [208] focused on implementing them in the data plane, as explained next. More advanced RL algorithms, such as Deep Q-Networks (DQN) [254] and Deep Deterministic Policy Gradient (DDPG) [255], take advantage of the capabilities of deep neural networks (DNN) to address complex problems involving a large action space. However, despite attempts to implement DNNs within the network in N3IC [199], the immense complexity of these networks, and therefore high resource consumption, has thus limited the success of deploying deep RL within the data plane so far.

*1) **SARSA**:* SARSA is an on-policy RL algorithm and learns the value of the policy executed by the agent [144]. There are two essential but complex operations in this algorithm: ε-greedy and updating a Q-table. For ε-greedy, the challenges are realizing the greedy policy and randomly selecting the action within the data plane, as these either require externs or significant resources (e.g., many stages). In a Q-table, as shown in Equation 7, the equation requires three multiplication operations, and the second half of the equation has a dependency, making it hard to fit on programmable network devices with limited stages. Meanwhile, for a value-based method, defining the value (reward) of each action is tricky, particularly when it comes to balancing trade-offs between memory and accuracy, as the intermediate results stored in the M/A table should be optimized to minimize accuracy loss.

$$Q(S, A) \leftarrow (1-\alpha) Q(S, A)+\alpha\left[R(S, a)+\gamma Q\left(S^{\prime}, a^{\prime}\right)\right] \quad (7)$$

While it is challenging to implement SARSA on some programmable ASIC, targets such as Netronome SmartNIC (Agilio LX) [207] enable it through externs. The Agilio LX architecture is different from the architecture of e.g., Intel Tofino, used in many previous works. As a SmartNIC, it supports more functions but has a lower data rate compared to switches. Agilio LX has programmable MEs (microengines), and packet processing throughput is increased through MEs parallelism. This paper takes SARSA as an example of one-step temporal-difference (TD) RL, as shown in Figure 17 (Table IV-14 also shows the generalized workflow of TD learning). During ingress or egress processing, the environment reward and current state information are extracted and provided to an extern for in-band RL [207, 208]. Such extern implements the RL cores using multiple MEs inside and the storage component constructed by different types of memory. The input information first does a Q-table lookup for the value of action by using Tile Coding. This coding method adds up all the coarse-grained tiles to represent one fine-grained Q-table. Each coarse-grained tile can be executed in each Minion (thread) in parallel. Then, the model selects the action (based on the action data and the greedy policy) and sends it to the output directly. Meanwhile, the model updates the Q-table based on the previous state value and action if the policy needs to be updated, which can contribute to accelerating the whole process.

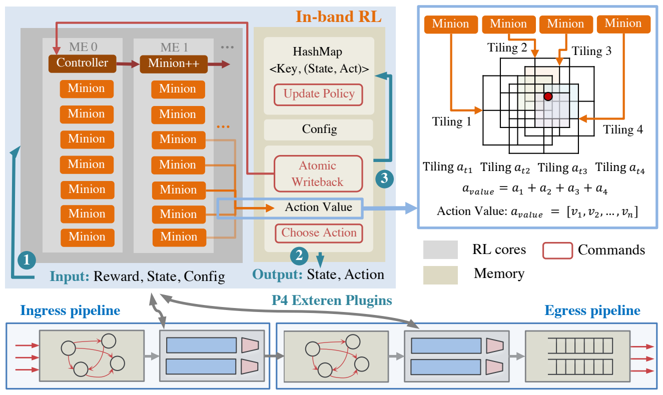


Fig. 17. Workflow of One-Step Temporal-Difference RL [207].

This implementation approach of one-step temporal-difference RL takes advantage of P4 extern plugins in SoC- or NPU-based SmartNICs. In the M/A pipeline, the RL function is called using an extern to implement fixed point values of actions and tile-coded policies.

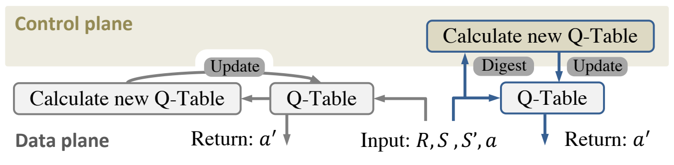
*2) **Q-learning**:* Q-learning is an off-policy RL algorithm. It uses the greedy approach to learn the value of an action given a particular state and seeks to find the optimal action-selection policy using a value function called the Q-function [256]. Q-learning learns the value of the optimal policy without being influenced by the actions of agents.

There are similarities between SARSA and Q-learning. They are both model-free and value-based methods, which means they do not require model knowledge and they update the value function based on temporal difference learning. The approach Q-learning uses to update a Q-table is shown in Equation 8. It is similar to SARSA's equation, but requires an additional max operation.

$$Q(S, A) \leftarrow (1-\alpha)Q(S, A) + \alpha \left[R(S, a) + \gamma \max_a Q(S', a)\right] \tag{8}$$

Consequently, the difficulties in Q-learning's data plane implementation are similar to that of SARSA's implementation, meaning how to update a Q-Table and how to realize ε-greedy operation [208]. This similarity means it can also be implemented by using ME externs, like SARSA's implementation, as reported in Section VI-D1.

The main difference between Q-learning and SARSA is the order of Q-table updating and action execution. The "decision first, update later" update order of Q-learning makes it well-suited for execution in the data plane [209]. Figure 18 illustrates two additional data plane implementation methods for Q-learning.



(a) Q-learning fully on Data Plane (b) Q-learning partially on Data Plane

Fig. 18. Workflow of QCMP based in-network RL [209].

In the first approach, shown in Figure 18 (a), the Q-table is stored within the data plane using registers [209], with the state serving as the key and the concatenated Q-values of all actions as the value. To determine the action, the input packet must provide the current state S', allowing the selection of the action with the highest value using an ϵ-greedy policy and a random number from the packet. For Q-table updates, the previous state S, action a, and reward $R(S, a)$ can either be input from the packet or stored in the data plane using registers. Equation 8 is applied, replacing multiplication with tables that store intermediate results.

The second method (Figure 18 (b)), transfers the complex Q-value calculation and Q-table update process to the control plane [209]. Consequently, the Q-table can be stored in the M/A table within the data plane.

In terms of implementation, the first method is more resource-heavy, especially in stages, and better suited for software switches, while the second method is compatible with commodity hardware (such as Tofino switch-ASIC) but exhibits slower response times to changes in the environment due to the control plane's latency. It therefore fits better applications that require occasional or periodic updates, without precluding acting on every packet.

VII. Challenges and Solutions

While offloading ML models to programmable network devices, many common challenges have been identified by existing works, such as limited amounts of memory, restricted number of stages, non-natively supported data types, and weak computational capability. This is because ML models usually have a relatively high implementation complexity but programmable network devices have very limited hardware resources. In the following subsections, we will list these challenges and report possible solutions.

The many different implementations of in-network ML algorithms face several common problems and challenges. These challenges range from resource constraints, through limited data types to weak computational capabilities. The application-specific, constrained nature of network device architectures stands in contrast to the increasingly large and complex ML models. The following section discusses the common challenges in mapping ML models to the data plane, and their solutions.

The two central types of constraints that have been addressed to date are limited amount of resources (e.g., memory, number of processing stages) and lack of functionality (e.g., support of data types or mathematical operations).

A. Restricted Number of Stages

Unlike CPUs, where instructions go through the pipeline and the data is in the memory, in PISA-based devices the data goes through the pipeline, and the instructions are in the memory (M/A tables). To maintain line rate, packets go through the pipeline only once, therefore the number of stages in the pipeline is directly related to the number of serial operations that can be done on a packet. This number of stages, especially in switch-ASIC, is limited (e.g., 12 stages per ingress or egress pipe in Tofino [257], and 20 stages per pipe in Tofino 2 [258]), leading to a narrow range of supported functionalities [16]. We classify the stage limitations into two categories: actions with dependency and tables with dependency, as explained next.

*1) **Actions with Dependencies:*** Theoretically, complex computations can be enabled in programmable network devices but require several steps of action, and in most cases, these actions have dependencies. For example, multiplication can be realized through repeated addition. However, the limited number of stages within the pipeline is prohibitive for such solutions. Moreover, the lack of loops constrains it further. Therefore, using the minimum stages for different complex computations within programmable network devices is an open challenge. The most common existing solution is using M/A tables to store the results of complex calculations [16]. This approach is inspired by the use of lookup tables in FPGAs to realize different operations.

```
bit<64> m1  = 0x5555555555555555;
bit<64> m2  = 0x3333333333333333;
bit<64> m4  = 0x0f0f0f0f0f0f0f0f;
bit<64> m8  = 0x00ff00ff00ff00ff;
bit<64> m16 = 0x0000ffff0000ffff;
bit<64> m32 = 0x00000000ffffffff;
/* This PopCount is for 64-bits' input */
action PopCount(bit<64> bitInput){
    bit<64> x = (bit<64>) bitInput;
    x = (x & m1 ) + ((x >>  1) & m1 );
    x = (x & m2 ) + ((x >>  2) & m2 );
    x = (x & m4 ) + ((x >>  4) & m4 );
    x = (x & m8 ) + ((x >>  8) & m8 );
    x = (x & m16) + ((x >> 16) & m16);
    x = (x & m32) + ((x >> 32) & m32);
}
```

Code Example 1. PopCount code example under action format.

For example, Hamming weight (PopCount) shown in Code Example 1 is a method to calculate the number of ones in a 64-bits string. It utilizes iterative right-shifting and bitwise & operations to examine the least significant bit of each shifted position, incrementing a counter accordingly, in which 24 arithmetic operations are needed (Line 10-15). Due to the high dependency of these calculations, at least 18 pipeline stages are required to enable this operation, which is more than the maximum number of stages for the Tofino Switch and some other programmable network devices. Instead, M/A tables can replace these complex actions and require a much smaller number of stages. Code Example 2 shows how to use M/A tables to lookup the number of ones in a 64-bit string: we split a 64 bits string into four 16 bits substrings, then we adopt four 16 bits PopCount tables to lookup the number of ones in each substring. Finally, the result is the sum of ones in four substrings. The PopCount using M/A tables only consumes three stages (compared with 18 stages using the first code) and 4 SRAM-based tables of 64K entries.

```
1   action PopCount(bit<8> count) {
2       meta.popcount_result = count;
3   }
4   table popcount {
5       /* This is a 16-bit key */
6       key = { meta.popcount_input : exact; }
7       actions = { PopCount; }
8       const entries = {
9           0 : PopCount(0);
10          1 : PopCount(1);
11          2 : PopCount(1);
12          3 : PopCount(2);
13          4 : PopCount(1);
14          5 : PopCount(2);
15          ...
16      }
17      size = 65536; /* 2^16 */
18      /* This is required too */
19      const default_action = PopCount(0);
20  }
```

Code Example 2. PopCount code example under table format. PopCount code example under table format. Line 9-15 show five sample table entries. When the input value *meta.popcount_input* as the input for example equals 3 (...011) the output will be 2 based on the result hardcoded in line 12.

Thus, M/A tables are suited for executing complex operations in programmable network devices without consuming many pipeline stages, and enable calculating multiple operations within a single table.

*2) **Tables with Dependencies:*** Intuitively, M/A tables may have dependencies, and most of the dependencies are logically needed and cannot be changed. For example, in ML models like SVM and Ensemble models [16, 161], some tables have a dependency on each other where the typical examples are *Vote Tables*.

```
1   /* Tables have dependencies */
2   action do_vote1(bit<8> vote) { // vote = 1 or 0
3       meta.count_vote = meta.count_vote + vote;
4   }
5   action do_vote2(bit<8> vote) { // vote = 1 or 0
6       meta.count_vote = meta.count_vote + vote;
7   }
8   /* This helps to break the dependencies */
9   action do_vote1(bit<8> vote) { // vote = 1 or 0
10      meta.vote1 = vote;
11  }
12  action do_vote2(bit<8> vote) { // vote = 1 or 0
13      meta.vote2 = vote;
14  }
15  table vote_table1 {
16      key = { meta.input1 : exact; }
17      actions = { do_vote1; }
18      size = n;
19  }
20  table vote_table2 {
21      key = { meta.input2 : exact; }
22      actions = { do_vote2; }
23  }
24  /* Other method to calculate */
25  vote_table.apply(); // key1 = meta.vote1, key2 =
       meta.vote2, action data == meta.vote1 + meta.
       vote2 (pre calculated)
```

Code Example 3. Code example of tables with dependency due to technical reasons.

As shown in Code Example 3 from Line 1 to Line 7, each table calls an action, and the operations inside each action are dependent. In order to prevent the dependency of tables and save the stages consumed by arithmetic operations, it is needed to: (i.) move logical operations out of actions under each table (Line 8-14) (ii.) use M/A tables, which can ignore the dependency of these operations, to calculate the results (Line 24-25). Even though the tables without a dependency can reduce stage usage, the number of tables in each stage is still limited, and developers should not use too many tables for computation within the data plane.

B. Limited Amount of Memory

Unlike end-host devices like servers, programmable network devices are fast but have a comparatively small amount of memory that contains only a few tens of MB of SRAM and TCAM. Efficiently using precious space to store needed information is an open problem that networking researchers have been exploring for decades in the context of routing tables [259]. This subsection includes two possible problems as well as the solutions that can help reduce memory consumption of in-network computations as well as ML algorithms.

*1) **Inefficient Mapping of inputs in Match-action tables:*** Many arithmetic operations in ML algorithms are not natively supported in programmable network devices. As noted above, M/A tables are commonly used to look up the results of these complex operations (e.g., predict a label from a set of features): considering that $f(\cdot)$ is the lookup table and x_i is an input of lookup table (e.g., a feature in the ML algorithm), the output of arithmetic operations y can be presented as $y = f(x_1, x_2, \cdots, x_n)$, where n is the number of features in a ML algorithm [16].

$$N_{entries} = r_1 \times r_2 \times \cdots \times r_n = \prod_{i=1}^{n} r_i \tag{9}$$

Equation 9 shows the number of table entries required in an n inputs operation. r_i represents the range of inputs x_i: being x_i^{min} the possible minimum value of x_i and x_i^{max} the possible maximum, $r_i = x_i^{max} - x_i^{min} + 1$. In case the range r_i for each feature x_i is large, the total number of table entries will be also large and may exceed table size limit. It is possible to reduce the dimension of table entries by using intermediate inputs. For instance, instead of directly setting the features as inputs, using pre-computed posterior probability $\mathbb{P}(x_i|y)$ as inputs can significantly decrease the number of table entries and so the memory. In short, the approach can be formulated as follows: $y = f(mid_1, mid_2, \cdots, mid_n)$, where $mid_i = \text{g}(x_i)$ and $g(\cdot)$ is a function that can map x_i to a narrow range. Therefore, as shown in Equation 10, the total number of used table entries is $\sum_{i=1}^{n} r_i + \prod_{i=1}^{n} r_i^{mid}$, where $\{mid_i \mid mid_i^{min} \leq mid_i \leq mid_i^{max}\}$ and $r_i^{mid} = mid_i^{max} - mid_i^{min} + 1$.

$$N_{entries} = \sum_{i=1}^{n} r_i + \prod_{i=1}^{n} r_{mid}^{i} \tag{10}$$

Since the range of each input's intermediate result (i.e., r_{mid}^{i}) is usually very narrow, the total number of table entries $N_{entries}$ is significantly reduced.

*2) **Excessive Exact-Match Table Entries**:* A large number of table entries are required in many ML classification use cases, such as in the decision table in DT, the vote-to-class table in RF, and the lookup table in SVM that stores intermediate results of complex arithmetic operations. Exact-Match tables match a key with a table entry of an identical value. Even if an Exact-Match Table is used appropriately to look up the computational results, the number of table entries is usually very large and may exceed hardware resources. To solve this problem, the model can use three traditional alternative matching approaches to map multiple entries in the same scheme to a single output, meaning Longest Prefix Match (LPM) [260], Ternary Match [16, 261], and Range Match [219]. This can help significantly reduce the number of table entries, however not every set of input features and output actions easily maps to one of the three.

TABLE V
A COMPARISON OF FOUR DIFFERENT APPROACHES TO REDUCE EXACT TABLE ENTRIES (EACH TABLE ENTRY: KEY - LABEL).

ID	Exact	LPM	Ternary	Range	Default[1]
1	00001 - 1	00*** - 0	*000* - 1	[1, 1] - 1	00001 - 1
2	00010 - 0	01*** - 1	*0*1* - 0	[2, 3] - 0	00100 - 2
3	00011 - 0	001** - 2	*010* - 2	[4, 5] - 2	00101 - 2
4	00100 - 2	0011* - 0	*1*** - 1	[6, 7] - 0	01000 - 1
5	00101 - 2	0000* - 1		[8, 8] - 1	
6	00110 - 0				
7	00111 - 0				
8	01000 - 1				

[1] Using default action to replace the hottest class. In this case, label 0 is stored as default.

Table V shows an example of using different matches to save memory resources: while exact match needs 8 table entries according to the label in the ML model, LPM requires 5 entries and Ternary match consumes only 4 entries. To elaborate, in LPM the table entry with the longest matching prefix is selected. For instance, the input 00001 matches both 00*** in ID 1 and 0000* in ID 5. The output will still be 1 because the matching length of 0000* in ID 5 is the longest. The ternary match is determined by $key\&mask == input\&mask$. When multiple rows are hit, the one with the highest order (ID) will be chosen as the output. For example, if the input is 01111, *1*** in row ID 4 will be matched because 01111 & 01000 == 01000 & 01000, resulting in an output of 1 (*1*** is an abbreviation for key 01000 and mask 01000, which only focuses on the fourth bit value). Range match is straightforward, where the output is the interval of table entries that includes the input value. This illustrates how when there is a large number of input features and a determined range of the label, Range/LPM/Ternary-based approaches may be more effective than Exact Match. In general, the choice of a matching approach depends on the characteristics of the ML model and features space. In our example, Ternary match uses the smallest number of entries, but in other cases, LPM or Range-match may have better performance [161].

While Exact Match often uses SRAM, Range-based matches use TCAM (or SRAM-based algorithmic TCAM). Effectively using TCAM for ML use cases is an interesting research topic that can build upon past work on efficient TCAMs usage, trying to optimize for TCAM's high power dissipation [262], compression potential [263], and compression complexity [264]. While past work focused on matching in forwarding and routing tables (e.g., [263, 264]), ML features space is richer and may not exhibit the same spatial and temporal properties that were leveraged in past work, creating new research opportunities.

In P4, a key that is missing from the M/A table is assigned a *default* action. In Table V, the Default column refers to the case where default is assigned label 0 as the action. For instance, if the input is 00111, the output (under the Default column) will be 0. Consequently, a ML model can use the default action to remove the most frequent label in the table. That is, the action with the largest number of table entries can be configured as the default action and table size can be significantly reduced.

C. Non-Supported Data Types

The support of some data types is missing in programmable data planes for ML. The most common missing type is floating-point, frequently used in ML implementation. We mainly discuss alternative ways of implementing floating points and negative numbers in this section.

*1) **Floating-point Number**:* Most ML algorithms are developed using floating point numbers, which are not natively supported in P4-capable programmable data planes. However, floating point values are not the only feasible way of numerical presentation for real numbers [185]. Take the training process of DNN as an example, there are several solutions including block floating-point representation [265, 266], fixed-point quantization, and quantization after mapping using an adaptive scaling factor [185]. For quantization methods, dithering for floating-point numbers can be applied to reduce quantization error by introducing random noise [267, 268, 269, 270, 271]). Here, we provide an example of the most commonly used method, the quantization after mapping. For an input such as -0.25, the output will be 48 when the $factor$ is 10 and $shift$ is 5 ($round((value + shift) * factor)$). Although this result has some accuracy loss, the specific accuracy loss rate can be controlled by adjusting the $factor$, which applies to most use cases.

*2) **Negative Number:*** Storing and computing using negative numbers is an inevitable challenge when implementing ML algorithms. Suppose we have a binary value of 0110, representing the decimal number 6 in unsigned binary notation. If we use a signed bit at the leftmost position, the same binary value can represent both positive and negative numbers. In this case, the signed bit is 0, indicating a positive number. However, if we change the signed bit to 1, the same binary value now represents a negative number. To distinguish between positive and negative numbers, we need to use the most significant bit as a sign bit. However, this approach sacrifices the range of values (e.g., only use 63 bits instead of 64 bits can be used to represent a number) and is not enough for unifying positive and negative numbers in computations. An alternative solution is to use two's complement representation [272]: the two's complement of a positive number and 0 is the number itself, whereas the two's complement of a negative number can be calculated by reversing the bits of its corresponding positive number and adding 1. For example, the two's complement of the decimal number 6 is 0110 while the two's complement of the decimal number -6 is 1010, which is obtained by reversing the bits of 0110 and adding 1. In this way, one can represent both positive and negative numbers using the same number of bits, and any computation containing negative numbers in programmable network devices is feasible. Two's complement is also often used in digital electronics to replace a subtraction operation. Note that in many cases the handling or conversion of negative numbers can be done in the compilation stage, where table entries are generated, and is not required as a data plane operation.

D. Limited Computational Capability

To assure high packet processing speed, programmable network devices only support a limited number of basic arithmetic and logical operations, such as addition (+), subtraction (−), leftshift (<<), right shift (>>), xor (∧) and if-else condition. However, some additional operations, like the comparison of two variables, division, multiplication, frequency count, and matrix multiplication, are inevitable when implementing ML algorithms. In this subsection, we identify the limitations and report potential solutions for them.

In some programmable network devices, comparing two variables directly in an if-else condition does not have a primitive, which may be troublesome when implementing ML models with an explicit comparison. As shown in Code Example 4, to compare two variables, the implicit comparison is needed by computing the difference of variables out of the if-else condition and then comparing the difference with 0 [161].

```
/* This is the comparison of two variables */
action do_subtract(){sub = meta.a - meta.b}
/* Logic part */
do_subtract()
if ( sub < 0 ){...}
```

Code Example 4. Code example of how to do comparison between two integers

As shown in Code Example 4, when subtraction is needed, one should subtract outside the condition and isolate the comparison into a separate action.

*1) **Multiplication:*** Multiplication is a basic arithmetic operation frequently used in ML algorithms. Multiplication can be computed using either M/A tables or bit shifting when one of the multiplication factors is a power of 2. In case M/A tables are used, the output returns the product of the multiplied operands.

$$Product = Factor_1 << log_2(Factor_2) \tag{11}$$

Equation 11 shows how to calculate the product of two multipliers if any of the two multipliers is a power of 2. This method can also be extended to more general cases. For instance, if a $Factor_2$ is known as a constant 6, the product is $Factor_1 * 6$, which can be decomposed to $Factor_1 * 2 + Factor_1 * 4$. Then the result is retrieved by computing $Factor_1 << 1 + Factor_1 << 2$ (a different decomposition is possible with subtractions but is sometimes less efficient in the hardware). The value on the right side of the left shift needs to be pre-stored in P4 code.

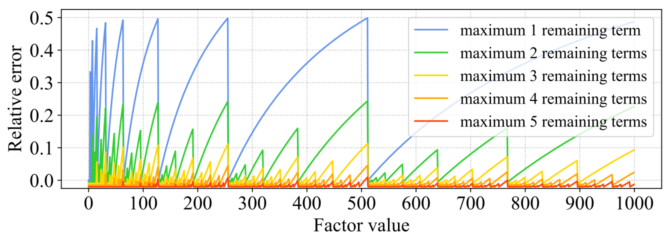


Fig. 19. The variation of relative error with the value of a factor that is decomposed under different limitations to the maximum number of remaining terms.

However, only a limited number of decomposition terms can be considered since too many terms would exceed the number of stages in a programmable network devices. Therefore, we present the variation of relative error with the value of a factor that is decomposed under different limitations to maximize the number of terms from our experimental results. As shown in Figure 19, multiplication results get more accurate when more decomposition terms are retained. More importantly, it proves that choosing the larger number as the divided factor will obtain more accurate results.

*2) **Division:*** Division an elementary but necessary arithmetic operations in ML models. However, since division requires too many sub-binary calculations that slow the packet processing, P4 does not natively support it. Similar to multiplication, such an operation can be replaced by a M/A table and bit shifting [273]. If the dividend or divisor is a constant, the result of the division can be looked up in a M/A table with stored results. However, in case both components are variables, the size of the required table exponentially increases. This makes M/A tables less effective when computing the division of two variables. As an alternative solution, bit shift is more suitable and efficient when the divisor is a constant and a power of 2.

$$Quotient = Dividend >> log_2(Divisor) \tag{12}$$

Equation 12 shows that the quotient can be obtained easily by right shifting $log_2(Divisor)$ bits. Division should be not frequently used since it usually consumes too many hardware resources.

3) ***Frequency:*** Frequency (e.g., packet and flow count) is needed as a high-level input feature of ML models. However, when the number of inputs is large, counting frequency requires large memory and stage consumption. *Sketches* are an efficient way often used in networking to approximate frequency considering memory constraints, in which the two most commonly used are Count Sketch and Count-min Sketch.

Count Sketch is composed of d rows with size K, and it uses two sets of hash functions $\{h_1, \ldots, h_d\}$ and $\{g_1, \ldots, g_d\}$ [274]. Hash functions h_i are used to map n inputs (e.g., a field extracted from packet header) into a lower dimension space K (K is much smaller than n), and hash functions g_i are the values (either -1 or 1) to update the counters, where $h_i : X \mapsto \{0, K\}$ and $g_i : X \mapsto \{-1, 1\}$. For any incoming input, the sketch updates the corresponding counter with index h_i in each row, which can be formulated as $C_{i,h_i(x)} \leftarrow C_{i,h_i(x)} + g_i(x)$. When querying frequency, the estimated result is the median value of each row's counter multiplied by $g_i(x)$ as shown in Equation 13.

$$median\{g_1(x)C_{1,h_1(x)}, \ldots, g_d(x)C_{d,h_d(x)}\}. \tag{13}$$

Count-min sketch is a simplified version of count sketch. Instead of using hash function g, it uses add one in the update counter (h_i) process and uses Equation 14 to evaluate the result.

$$min\{C_{1,h_1(x)}, \ldots, C_{d,h_d(x)}\}. \tag{14}$$

Compared to Count-min Sketch, Count Sketch has a better accuracy and memory trade-off. However, Count-min sketch only needs one hash function and is easier to be implemented in programmable network devices. The choice of sketches depends on the different requirements of ML algorithms.

4) ***Matrix Multiplication:*** Many ML algorithms, especially neural network-based algorithms, require matrix multiplication. However, as noted before, multiplication operations and floating-point values may not be supported. The most intuitive solution for matrix multiplication is to use M/A tables to store the intermediate results of the calculation, which has been widely used in programmable switches with a limited number of stages [16, 161, 188, 203]. Another less-common solution, initially introduced by BNN, uses mathematical operations and newly defined data types to approximate the multiplication result [226]. This approach requires more stages but less memory, which is feasible as a solution in SmartNICs and software switches.

Considering BNN as an example, the model optimizes and simplifies the matrix multiplication in a fully connected neural network to an input matrix and a weight matrix, where each matrix is a binary matrix. The product of the two matrices is the closest approximation to the neural network. It has been proved that matrix multiplication can be computed by solving an optimization problem as reported in [226]. More specifically, the task is divided into two parts. First, the weights are replaced with binary numbers. Shown in Figure 11 Step 1, the input and weight are of size n, and $\mathbf{X}, \mathbf{W} \in \mathbb{R}^{1\times n}$. Suppose that there is a binary metric $\mathbf{B} \in \{+1, -1\}^{n\times 1}$ and a scaling factor $\alpha \in \mathbb{R}^+$ such that the composed weight, $\mathbf{X}^T\mathbf{W} \approx \mathbf{X}^T\alpha\mathbf{B}$, should be as close to the original weight as possible.

$$\alpha^*, \mathbf{B}^* = \underset{\alpha,\mathbf{B}}{\text{argmin}} \|\mathbf{W} - \alpha\mathbf{B}\|^2 \tag{15}$$

The optimization can be solved by expanding the Equation 15. The optimized binary weight is $\mathbf{B}^* = \text{sign}(\mathbf{W})$ and the scaling factor is $\alpha^* = \frac{1}{n}\|\mathbf{W}\|_{\ell 1}$.

Second, binary the input $\mathbf{X}$ into $\mathbf{H} \in \{+1, -1\}^n$ with a scaling factor $\beta \in \mathbb{R}^+$. The constructed inputs and the weights need to be very similar to each other $\mathbf{X}^\top\mathbf{W} \approx \beta\mathbf{H}^\top\alpha\mathbf{B}$. Then, suppose $\mathbf{Y}_i = \mathbf{X}_i\mathbf{W}_i$, $\mathbf{C}_i = \mathbf{H}_i\mathbf{B}_i$, $\gamma = \beta\alpha$ and get the following optimization.

$$\gamma^*, \mathbf{C}^* = \underset{\gamma,\mathbf{C}}{\text{argmin}} \|\mathbf{X} \odot \mathbf{W} - \beta\alpha\mathbf{H} \odot \mathbf{B}\| = \underset{\gamma,\mathbf{C}}{\text{argmin}} \|\mathbf{Y} - \gamma\mathbf{C}\| \tag{16}$$

The form of this optimization is similar to Equation 15 using a similar method and result of binary weight. The optimized multiplication of the constructed input and weights is $\mathbf{C}^* = \text{sign}(\mathbf{Y}) = \text{sign}(\mathbf{X}) \odot \text{sign}(\mathbf{W})$, where $\odot$ indicates element-wise product, and the optimized multiplication of scaling factor is $\gamma^* = \frac{1}{n}\|\mathbf{X}\|_{\ell_1}\frac{1}{n}\|\mathbf{W}\|_{\ell 1} = \beta^*\alpha^*$. Then, the output of step 2 in Figure 11 is shown in the Equation 17.

$$\mathbf{X}^T\mathbf{W} = \sum_{i=1}^{n} \mathbf{C}_i^* \approx \beta^*\alpha^* \sum_{i=1}^{n} (\text{sign}(\mathbf{X}) \odot \text{sign}(\mathbf{W}))_i \tag{17}$$

When two values $m, n \in \{-1, 1\}$, $m\overline{\oplus}n$ will have most similar pattern to the $m \odot n$. Thus, replace $\odot$ by $\overline{\oplus}$. $\beta^*\alpha^*$ can be ignored due to the multiplication operation. The multiplication of all inputs and weights and the summation can be simplified to Equation 18.

$$\mathbf{X}^T\mathbf{W} \approx \sum_{i=1}^{n} \overline{(\text{sign}(\mathbf{X}) \oplus \text{sign}(\mathbf{W}))_i} \tag{18}$$

In this Equation, the result will remain the same when inputs and weights change to the bit format $\mathbf{X}, \mathbf{W} \in \{0, 1\}^n$. Therefore, $\mathbf{X}^T\mathbf{W}$ calculation is finally transformed to the form with only $\overline{\oplus}$ (XNOR) and $\sum$ (summation, a sequence of additions) operations, which can be implemented in the data plane.

E. Other Limitations

In addition to the aforementioned limitations, there are still many unsolved challenges related to in-network ML algorithms, including but not limited to:

- **Updating ML models during runtime:** When a model is deployed, due to the appearance of new data, change of environment, and drift of concepts, an existing model may no longer be able to predict outcomes accurately. Models require timely updates to keep their ML performance. In-network ML update is not easy as server-based ML, and it is challenging to conduct a hitless update process with minor to no influence on normal packet forwarding functionality.
- **In-network ML when traffic is encrypted:** In general, many applications and protocols encrypt the data payload, which includes any user data that is being transmitted. In host-based ML, the traffic is decrypted before it is sent for inference. In-network ML may not be able to deal with encrypted data, though it can still take advantage of unencrypted protocol headers or metadata. Some new devices, such as SmartNICs, include encryption/decryption accelerator cores that can allow overcoming the problem, yet there is no solution for switch-ASIC.
- **Increasing the scalability of ML models supported by programmable network devices:** Current in-network ML still supports only limited model sizes due to resource constraints on programmable network devices. This can limit potential applications of in-network ML on multiple use cases. Potential solutions for this are all challenging to realize, whether designing a new target with more resources, upgrading the ML algorithm with less resource consumption, or scaling the model by deploying it on multiple devices [189].

These open questions are left for the community to explore in the future.

VIII. Lessons Learned and Future Trends

A. Lessons Learned

This survey covers the definition of in-network ML, clarifies the scope, describes its evolution and background, and reports state-of-art research and solutions in P4-capable programmable network devices.

Intuitively, in-network ML is an emerging research field that has already attracted a lot of interest. Research to date has mainly focused on offloading and mapping ML techniques to programmable data planes. Many ML models and their enhancements have been proposed and implemented. As network devices are resource constrained, the size of the models implemented was limited. Despite that, existing works have proven that high ML performance can be achieved for some use cases, providing inference in the network with a good trade-off between model size and accuracy. This makes in-network ML a promising research direction.

The applications of in-network ML have so far focused on traditional networking use cases, and in particular security applications such as anomaly detection and DDoS detection and mitigation [17, 194, 200, 201, 202, 210, 217, 218]. While these are mostly volumetric in nature, and benefit from the high packet processing rate of network devices, a different direction is to benefit from latency reduction, such as in financial use cases [201, 213]. To support these use cases, feature extractions for in-network ML have used both stateful and stateless features. A challenge in expanding the adoption of in-network ML is the limited availability of automated tools and frameworks. Only a few works developed frameworks to ease and accelerate the adoption of in-network ML. These include automating in-network ML generation (Planter [161]), parameter tuning (Homunculus [190]), and distributed deployment (DINC [189]).

Though the ML performance and overhead of each in-network ML model are compared in Planter [161], there is currently no unified benchmark for resource consumption in programmable network devices and ML classification performance, which makes it difficult to compare different works and balance models' performance against its resource consumption.

B. Future Trends

There are several foreseeable research directions for in-network ML.

- **Runtime programmability:** Currently, every time the ML model in the data plane is changed or the model's parameters require an update, the developers have to recompile the program and execute new compiled code in the network device. The problem can be divided to two parts: changing the value of a used parameter, and changing the features being extracted and used. The first may be solved by storing parameters in re-configurable memory (e.g., tables) and atomically updating them, in a manner similar to routing rules and without affecting traffic. The second, feature extraction, is more challenging as P4 manipulation may be unavoidable. While solutions to P4 runtime programmability have been suggested [275], none were applied to in-network ML or can easily be adapted.
- **Large-scale ML models:** Applying ML to network-related problems requires processing large amounts of data from the network, matching traffic rates. Traditional ML approaches can't handle such traffic volumes at real time, making in-network ML a favorable solution. However, to support a wide range of use cases with high inference performance, it is required to support large-scale ML models. Mapping large ML models to network devices, overcoming their resource constraints, requires new offloading approaches. One possible solution is decomposing a large-scale model into multiple components and distributing them into programmable network devices, as done with some P4 programs [189, 276]. Then, a coordinator (e.g., an SDN controller) can be used to retrieve the results from multiple devices and make a final decision.
- **Benchmarks and metrics:** A variety of ML models have been mapped to a range of programmable network devices. To date, in-network ML publications have been use case focused, or model and hardware specific, thereby failing to comprehensively showcase algorithm performance from a varied perspective. Consequently, it is hard to fairly compare different solutions. To solve

this problem, it is necessary to create a benchmark kit for a unified quantitative assessment, as done for other ML solutions [277]. Such a benchmark will track and compare the performance of different in-network ML algorithms across platforms, as well as assess the state-of-the-art in a particular task or application.

- **More in-network ML use cases:** Many ML models have been proven feasible and reliable in the use cases that are described in existing works, especially network anomaly detection. However, the range of in-network ML use cases is still very narrow, leading to limited adoption in practice. Applying in-network ML beyond the networking domain, to fields such as Natural Language Processing and Computer Vision, is worth exploring. In-network ML can potentially be applied also to 6G, providing a faster and more robust network, and to smart cities empowering IoT devices with low-latency ML services. At a certain point, the emergence of new use cases for in-network ML will drive ML improvements and give rise to novel ideas.

In-network ML is increasingly recognized as a viable computational paradigm for ML-based applications, especially in the networking domain [209, 215]. While not all ML-based applications will benefit from in-network ML, highly data intensive use cases and those requiring ultra-low latency are prime candidates. As the deployment methodologies of in-network ML continue to evolve, the optimal exploitation of this new technology remains subject for ongoing discussions. In the short term, in-network ML is not expected to replace established ML accelerators, such as GPUs. It complements existing computing methods and serves use cases with specific requirements. Identifying use cases outside the networking domain remains the focus of future research.

IX. Conclusions

Offloading computing tasks from end servers into programmable network devices is an ideal way to reduce the workload of CPU and accelerate the data processing in modern computing infrastructure. This paper provided an overview of in-network ML using P4-capable programmable network devices. We reviewed the background, history, and recent development of in-network ML. In particular, we presented different types of ML algorithms in programmable data planes in detail as well as how the implementation challenges are dealt with. Finally, we summarized the lessons that we learned and provided our insights on the future trends of in-network ML. As programmable network devices becomes more powerful, this topic has the potential to be developed in the future.

Acknowledgments

This paper complies with all applicable ethical standards of the authors' home institution. This work was partly funded by VMware, Innovate UK (project 10056403) in the context of the SmartEdge EU project (grant agreement No. 101092908), the Leverhulme Trust (ECF-2016-289), and the Isaac Newton Trust. We acknowledge support from Intel. We would like to thank the anonymous reviewers, Riyad Bensoussane and Sawsan EL-Zahr for their valuable feedback, and Radostin Stoyanov for their contribution to early work of this paper. 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] "Barefoot Tofino," https://www.barefootnetworks.com/products/brief-tofino/ [Online; accessed November 2023].

[2] "Spectrum SN4000 Open Ethernet Switches," Website, https://www.nvidia.com/en-us/networking/ethernet-switching/spectrum-sn4000/ [Online; accessed November 2023].

[3] "AMD Pensando™ Infrastructure Accelerators," Website, https://www.amd.com/en/accelerators/pensando [Online; accessed November 2023].

[4] "NVIDIA BlueField Data Processing Units," Website, https://www.nvidia.com/en-us/networking/products/data-processing-unit/ [Online; accessed November 2023].

[5] "Broadcom Stingray SmartNIC Accelerates Baidu Cloud Services," https://www.broadcom.com/company/news/product-releases/53106 [Online; accessed November 2023].

[6] "Netronome Flow Processor Product Brief," Website, https://www.netronome.com/media/documents/PB_NFP-6000-7-20.pdf [Online; accessed November 2023].

[7] S. Ibanez, G. Brebner, N. McKeown, and N. Zilbermann, "The P4→NetFPGA Workflow for Line-Rate Packet Processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 1–9.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming Protocol-independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[9] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network Computation Is A Dumb Idea Whose Time Has Come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 150–156.

[10] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, "The Case for In-network Computing on Demand," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[11] A. W. Moore and D. Zuev, "Internet Traffic Classification Using Bayesian Analysis Techniques," in *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2005, pp. 50–60.

[12] A. Dainotti, A. Pescape, and K. C. Claffy, "Issues and Future Directions in Traffic Classification," *IEEE network*, vol. 26, no. 1, pp. 35–40, 2012.

[13] D. K. Bhattacharyya and J. K. Kalita, *Network Anomaly Detection: A Machine Learning Perspective*. Crc Press, 2013.

[14] R. Doshi, N. Apthorpe, and N. Feamster, "Machine Learning DDoS Detection for Consumer Internet of Things Devices," in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 29–35.

[15] D. Sanvito, G. Siracusano, and R. Bifulco, "Can The Network Be The AI Accelerator?" in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018, pp. 20–25.

[16] Z. Xiong and N. Zilberman, "Do Switches Dream of Machine Learning? Toward In-network Classification," in *Proceedings of the 18th ACM workshop on hot topics in networks*, 2019, pp. 25–33.

[17] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "pHeavy: Predicting Heavy Flows in The Programmable Data Plane," *IEEE Transactions on Network and Service Management*, 2021.

[18] T. C. Silva and L. Zhao, *Machine Learning in Complex Networks*. Springer, 2016, vol. 1.

[19] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications And Research Opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, pp. 1–99, 2018.

[20] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine Learning for Networking: Workflow, Advances and Opportunities," *Ieee Network*, vol. 32, no. 2, pp. 92–99, 2017.

[21] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research," *arXiv preprint arXiv:2101.10632*, 2021.

[22] R. Bifulco and G. Rétvári, "A Survey on The Programmable Data Plane: Abstractions, Architectures, and Open Problems," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–7.

[23] A. Y. Nikravesh, S. A. Ajila, C.-H. Lung, and W. Ding, "Mobile Network Traffic Prediction Using MLP, MLPWD, And SVM," in *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 2016, pp. 402–409.

[24] A. Eswaradass, X.-H. Sun, and M. Wu, "Network Bandwidth Predictor (NBP): A System for Online Network Performance Forecasting," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, vol. 1. IEEE, 2006, pp. 4–pp.

[25] S. Chabaa, A. Zeroual, J. Antari *et al.*, "Identification And Prediction of Internet Traffic Using Artificial Neural Networks," *Journal of Intelligent Learning Systems and Applications*, vol. 2, no. 03, p. 147, 2010.

[26] Y. Zhu, G. Zhang, and J. Qiu, "Network Traffic Prediction Based on Particle Swarm BP Neural Network," *J. Networks*, vol. 8, no. 11, pp. 2685–2691, 2013.

[27] Y. Li, H. Liu, W. Yang, D. Hu, and W. Xu, "Inter-data-center Network Traffic Prediction with Elephant Flows," in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2016, pp. 206–213.

[28] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin, "Online Flow Size Prediction for Improved Network Routing," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. IEEE, 2016, pp. 1–6.

[29] Z. Chen, J. Wen, and Y. Geng, "Predicting Future Traffic Using Hidden Markov Models," in *2016 IEEE 24th international conference on network protocols (ICNP)*. IEEE, 2016, pp. 1–6.

[30] R. Alshammari and A. N. Zincir-Heywood, "Machine Learning Based Encrypted Traffic Classification: Identifying SSH and Skype," in *2009 IEEE symposium on computational intelligence for security and defense applications*. IEEE, 2009, pp. 1–8.

[31] A. Finamore, M. Mellia, M. Meo, and D. Rossi, "Kiss: Stochastic Packet Inspection Classifier for UDP Traffic," *IEEE/ACM Transactions on Networking*, vol. 18, no. 5, pp. 1505–1515, 2010.

[32] D. Schatzmann, W. Mühlbauer, T. Spyropoulos, and X. Dimitropoulos, "Digging into HTTPS: Flow-based Classification of Webmail Traffic," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pp. 322–327.

[33] P. Bermolen, M. Mellia, M. Meo, D. Rossi, and S. Valenti, "Abacus: Accurate Behavioral Classification of P2P-TV Traffic," *Computer Networks*, vol. 55, no. 6, pp. 1394–1411, 2011.

[34] A. Este, F. Gringoli, and L. Salgarelli, "Support Vector Machines for TCP Traffic Classification," *Computer Networks*, vol. 53, no. 14, pp. 2476–2490, 2009.

[35] A. F. Esteves, P. R. In, M. Pereira *et al.*, "On-line Detection of Encrypted Traffic Generated by Mesh-based Peer-to-peer Live Streaming Applications: The Case of GoalBit," in *2011 IEEE 10th International Symposium on Network Computing and Applications*. IEEE, 2011, pp. 223–228.

[36] P. Haffner, S. Sen, O. Spatscheck, and D. Wang, "ACAS: Automated Construction of Application Signatures," in *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, 2005, pp. 197–202.

[37] J. Zhang, C. Chen, Y. Xiang, W. Zhou, and Y. Xiang, "Internet Traffic Classification by Aggregating Correlated Naive Bayes Predictions," *IEEE transactions on information forensics and security*, vol. 8, no. 1, pp. 5–15, 2012.

[38] A. Dainotti, A. Pescapé, and C. Sansone, "Early Classification of Network Traffic Through Multi-classification," in *International Workshop on Traffic Monitoring and Analysis*. Springer, 2011, pp. 122–135.

[39] T. T. Nguyen, G. Armitage, P. Branch, and S. Zander, "Timely And Continuous Machine-learning-based Classification for Interactive IP Traffic," *IEEE/ACM Transactions On Networking*, vol. 20, no. 6, pp. 1880–1894, 2012.

[40] W. De Donato, A. Pescapé, and A. Dainotti, "Traffic

Identification Engine: An Open Platform for Traffic Classification," *IEEE Network*, vol. 28, no. 2, pp. 56–64, 2014.

[41] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield, "Class-of-service Mapping for QoS: A Statistical Signature-based Approach to IP Traffic Classification," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004, pp. 135–148.

[42] L. He, C. Xu, and Y. Luo, "VTC: Machine Learning Based Traffic Classification As A Virtual Network Function," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, 2016, pp. 53–56.

[43] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust Network Traffic Classification," *IEEE/ACM transactions on networking*, vol. 23, no. 4, pp. 1257–1270, 2014.

[44] Y. Liu, W. Li, and Y. Li, "Network Traffic Classification Using K-means Clustering," in *Second international multi-symposiums on computer and computational sciences (IMSCCS 2007)*. IEEE, 2007, pp. 360–365.

[45] J. Erman, A. Mahanti, M. Arlitt, and C. Williamson, "Identifying And Discriminating Between Web And Peer-to-peer Traffic in The Network Core," in *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 883–892.

[46] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian, "Traffic Classification on The Fly," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 2, pp. 23–26, 2006.

[47] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson, "Offline/Realtime Traffic Classification Using Semi-supervised Learning," *Performance Evaluation*, vol. 64, no. 9-12, pp. 1194–1213, 2007.

[48] W. Li and A. W. Moore, "A Machine Learning Approach for Efficient Traffic Classification," in *2007 15th International symposium on modeling, analysis, and simulation of computer and telecommunication systems*. IEEE, 2007, pp. 310–317.

[49] Y. Jin, N. Duffield, J. Erman, P. Haffner, S. Sen, and Z.-L. Zhang, "A Modular Machine Learning System for Flow-level Traffic Classification in Large Networks," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 6, no. 1, pp. 1–34, 2012.

[50] M. Elnawawy, A. Sagahyroon, and T. Shanableh, "FPGA-based Network Traffic Classification Using Machine Learning," *IEEE Access*, vol. 8, pp. 175 637–175 650, 2020.

[51] K. Hara and K. Shiomoto, "Intrusion Detection System Using Semi-Supervised Learning with Adversarial Auto-encoder," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–8.

[52] D. M. Casas-Velasco, O. M. C. Rendon, and N. L. da Fonseca, "DRSIR: A Deep Reinforcement Learning Approach for Routing in Software-Defined Networking," *IEEE Transactions on Network and Service Management*, 2021.

[53] D. M. Casas-Velasco, O. M. C. Rendon, and N. L. da Fonseca, "Intelligent Routing Based on Reinforcement Learning for Software-Defined Networking," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 870–881, 2020.

[54] A. Forster and A. L. Murphy, "FROMS: Feedback Routing for Optimizing Multiple Sinks in WSN with Reinforcement Learning," in *2007 3rd international conference on intelligent sensors, sensor networks and information*. IEEE, 2007, pp. 371–376.

[55] R. Arroyo-Valles, R. Alaiz-Rodriguez, A. Guerrero-Curieses, and J. Cid-Sueiro, "Q-Probabilistic Routing in Wireless Sensor Networks," in *2007 3rd International Conference on Intelligent Sensors, Sensor Networks and Information*. IEEE, 2007, pp. 1–6.

[56] T. Hu and Y. Fei, "QELAR: A Machine-Learning-Based Adaptive Routing Protocol for Energy-Efficient And Lifetime-Extended Underwater Sensor Networks," *IEEE Transactions on Mobile Computing*, vol. 9, no. 6, pp. 796–809, 2010.

[57] A. A. Bhorkar, M. Naghshvar, T. Javidi, and B. D. Rao, "Adaptive Opportunistic Routing for Wireless Ad Hoc Networks," *IEEE/ACM Transactions On Networking*, vol. 20, no. 1, pp. 243–256, 2011.

[58] N. Fonseca and M. Crovella, "Bayesian Packet Loss Detection for TCP," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 3. IEEE, 2005, pp. 1826–1837.

[59] I. El Khayat, P. Geurts, and G. Leduc, "Improving TCP in Wireless Networks with an Adaptive Machine-Learnt Classifier of Packet Loss Causes," in *International Conference on Research in Networking*. Springer, 2005, pp. 549–560.

[60] El Khayat, Ibtissam and Geurts, Pierre and Leduc, Guy, "Enhancement of TCP over Wired/Wireless Networks with Packet Loss Classifiers Inferred by Supervised Learning," *Wireless Networks*, vol. 16, no. 2, pp. 273–290, 2010.

[61] P. Geurts, I. El Khayat, and G. Leduc, "A Machine Learning Approach to Improve Congestion Control over Wireless Computer Networks," in *Fourth IEEE International Conference on Data Mining (ICDM'04)*. IEEE, 2004, pp. 383–386.

[62] I. El Khayat, P. Geurts, and G. Leduc, "Machine-learnt versus Analytical Models of TCP Throughput," *Computer Networks*, vol. 51, no. 10, pp. 2631–2644, 2007.

[63] M. F. Zhani, H. Elbiaze, and F. Kamoun, "α_ SNFAQM: An Active Queue Management Mechanism Using Neurofuzzy Prediction," in *2007 12th IEEE Symposium on Computers and Communications*. IEEE, 2007, pp. 381–386.

[64] S. Masoumzadeh, G. Taghizadeh, K. Meshgi, and S. Shiry, "Deep Blue: A Fuzzy Q-Learning Enhanced Active Queue Management Scheme," in *2009 International Conference on Adaptive and Intelligent Systems*. IEEE, 2009, pp. 43–48.

[65] H. Jiang, Y. Luo, Q. Zhang, M. Yin, and C. Wu, "TCP-Gvegas with Prediction And Adaptation in Multi-hop Ad Hoc Networks," *Wireless Networks*, vol. 23, no. 5, pp. 1535–1548, 2017.

[66] A. P. Silva, K. Obraczka, S. Burleigh, and C. M. Hirata, "Smart Congestion Control for Delay-and Disruption Tolerant Networks," in *2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 2016, pp. 1–9.

[67] D. Vassis, A. Kampouraki, P. Belsis, and C. Skourlas, "Admission Control of Video Sessions over Ad Hoc Networks Using Neural Classifiers," in *2014 IEEE Military Communications Conference*. IEEE, 2014, pp. 1015–1020.

[68] A. Hiramatsu, "ATM Communications Network Control by Neural Network," *detail*, vol. 1, p. 260, 1989.

[69] N. Baldo, P. Dini, and J. Nin-Guerrero, "User-driven Call Admission Control for VoIP over WLAN with A Neural Network Based Cognitive Engine," in *2010 2nd International Workshop on Cognitive Information Processing*. IEEE, 2010, pp. 52–56.

[70] N. Baldo and M. Zorzi, "Learning And Adaptation in Cognitive Radios Using Neural Networks," in *2008 5th IEEE Consumer Communications and Networking Conference*. IEEE, 2008, pp. 998–1003.

[71] B. Bojovic, N. Baldo, J. Nin-Guerrero, and P. Dini, "A Supervised Learning Approach to Cognitive Access Point Selection," in *2011 IEEE GLOBECOM Workshops (GC Wkshps)*. IEEE, 2011, pp. 1100–1105.

[72] D. Liu, Y. Zhang, and H. Zhang, "A Self-learning Call Admission Control Scheme for CDMA Cellular Networks," *IEEE transactions on neural networks*, vol. 16, no. 5, pp. 1219–1228, 2005.

[73] B. Bojović, G. Quer, N. Baldo, and R. R. Rao, "Bayesian And Neural Network Schemes for Call Admission Control in LTE Systems," in *2013 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2013, pp. 1246–1252.

[74] G. Quer, N. Baldo, and M. Zorzi, "Cognitive Call Admission Control for VoIP over IEEE 802.11 Using Bayesian Networks," in *2011 IEEE Global Telecommunications Conference-GLOBECOM 2011*. IEEE, 2011, pp. 1–6.

[75] R. Shi, J. Zhang, W. Chu, Q. Bao, X. Jin, C. Gong, Q. Zhu, C. Yu, and S. Rosenberg, "MDP and Machine Learning-Based Cost-Optimization of Dynamic Resource Allocation for Network Function Virtualization," in *2015 IEEE International Conference on Services Computing*. IEEE, 2015, pp. 65–73.

[76] A. Blenk, P. Kalmbach, P. Van Der Smagt, and W. Kellerer, "Boost Online Virtual Network Embedding: Using Neural Networks for Admission Control," in *2016 12th International Conference on Network and Service Management (CNSM)*. IEEE, 2016, pp. 10–18.

[77] A. Adeel, H. Larijani, A. Javed, and A. Ahmadinia, "Critical Analysis of Learning Algorithms in Random Neural Network Based Cognitive Engine for LTE Systems," in *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*. IEEE, 2015, pp. 1–5.

[78] A. Testolin, M. Zanforlin, M. D. F. De Grazia, D. Munaretto, A. Zanella, M. Zorzi, and M. Zorzi, "A Machine Learning Approach to QoE-based Video Admission Control And Resource Allocation in Wireless Systems," in *2014 13th Annual Mediterranean Ad Hoc Networking Workshop (MED-HOC-NET)*. IEEE, 2014, pp. 31–38.

[79] S. Mignanti, A. Di Giorgio, and V. Suraci, "A Model Based RL Admission Control Algorithm for Next Generation Networks," in *2009 Eighth International Conference on Networks*. IEEE, 2009, pp. 191–196.

[80] H. Tong and T. X. Brown, "Adaptive Call Admission Control Under Quality of Service Constraints: A Reinforcement Learning Solution," *IEEE Journal on selected Areas in Communications*, vol. 18, no. 2, pp. 209–221, 2000.

[81] J. Wang and Y. Qiu, "A New Call Admission Control Strategy for LTE Femtocell Networks," in *2nd international conference on advances in computer science and engineering*, 2013, pp. 334–8.

[82] R. Mijumbi, J.-L. Gorricho, J. Serrat, M. Claeys, F. De Turck, and S. Latré, "Design and Evaluation of Learning Algorithms for Dynamic Resource Management in Virtual Networks," in *2014 IEEE network operations and management symposium (NOMS)*. IEEE, 2014, pp. 1–9.

[83] Y. Wang, M. Martonosi, and L.-S. Peh, "Predicting Link Quality Using Supervised Learning in Wireless Sensor Networks," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 11, no. 3, pp. 71–83, 2007.

[84] A. Pellegrini, P. Di Sanzo, and D. R. Avresky, "A Machine Learning-Based Framework for Building Application Failure Prediction Models," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 1072–1081.

[85] Z. Wang, M. Zhang, D. Wang, C. Song, M. Liu, J. Li, L. Lou, and Z. Liu, "Failure Prediction Using Machine Learning And Time Series in Optical Network," *Optics Express*, vol. 25, no. 16, pp. 18 553–18 565, 2017.

[86] U. S. Hashmi, A. Darbandi, and A. Imran, "Enabling Proactive Self-healing by Data Mining Network Failure Logs," in *2017 international conference on computing, networking and communications (ICNC)*. IEEE, 2017, pp. 511–517.

[87] K. Qader, M. Adda, and M. Al-Kasassbeh, "Comparative Analysis of Clustering Techniques in Network Traffic Faults Classification," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 5, no. 4, pp. 6551–6563, 2017.

[88] E. Kiciman and A. Fox, "Detecting Application-Level Failures in Component-based Internet Services," *IEEE transactions on neural networks*, vol. 16, no. 5, pp. 1027–1041, 2005.

[89] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure Diagnosis Using Decision Trees," in *International Conference on Autonomic Computing,*

2004. Proceedings. IEEE, 2004, pp. 36–43.

[90] A. Snow, P. Rastogi, and G. Weckman, “Assessing dependability of wireless networks using neural networks,” in *MILCOM 2005-2005 IEEE Military Communications Conference.* IEEE, 2005, pp. 2809–2815.

[91] J. S. Baras, M. Ball, S. Gupta, P. Viswanathan, and P. Shah, “Automated Network Fault Management,” in *MILCOM 97 MILCOM 97 Proceedings*, vol. 3. IEEE, 1997, pp. 1244–1250.

[92] Y. Kumar, H. Farooq, and A. Imran, “Fault Prediction And Reliability Analysis in A Real Cellular Network,” in *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC).* IEEE, 2017, pp. 1090–1095.

[93] C. S. Hood and C. Ji, “Proactive Network-fault Detection [Telecommunications],” *IEEE Transactions on reliability*, vol. 46, no. 3, pp. 333–341, 1997.

[94] P. Kogeda and J. I. Agbinya, “Prediction of Faults in Cellular Networks Using Bayesian Network Model,” in *International conference on Wireless Broadband and Ultra Wideband Communication.* UTS ePress, 2006.

[95] M. Ruiz, F. Fresi, A. P. Vela, G. Meloni, N. Sambo, F. Cugini, L. Poti, L. Velasco, and P. Castoldi, “Service-triggered Failure Identification/Localization Through Monitoring of Multiple Parameters,” in *ECOC 2016; 42nd European Conference on Optical Communication.* VDE, 2016, pp. 1–3.

[96] R. M. Khanafer, B. Solana, J. Triola, R. Barco, L. Moltsen, Z. Altman, and P. Lazaro, “Automated Diagnosis for UMTS Networks Using Bayesian Network Approach,” *IEEE Transactions on vehicular technology*, vol. 57, no. 4, pp. 2451–2461, 2008.

[97] A. I. Moustapha and R. R. Selmic, “Wireless Sensor Network Modeling Using Modified Recurrent Neural Networks: Application to Fault Detection,” *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 5, pp. 981–988, 2008.

[98] M. S. Mushtaq, B. Augustin, and A. Mellouk, “Empirical Study Based on Machine Learning Approach to Assess The QoS/QoE Correlation,” in *2012 17th European Conference on Networks and Optical Communications.* IEEE, 2012, pp. 1–7.

[99] P. Charonyktakis, M. Plakia, I. Tsamardinos, and M. Papadopouli, “On User-Centric Modular QoE Prediction for VoIP Based on Machine-Learning Algorithms,” *IEEE Transactions on mobile computing*, vol. 15, no. 6, pp. 1443–1456, 2015.

[100] E. Demirbilek and J.-C. Grégoire, “Machine Learning–Based Parametric Audiovisual Quality Prediction Models for Real-Time Communications,” *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 13, no. 2, pp. 1–25, 2017.

[101] V. A. Machado, C. N. Silva, R. S. Oliveira, A. M. Melo, M. Silva, C. R. Francês, J. C. Costa, N. L. Vijaykumar, and C. M. Hirata, “A New Proposal to Provide Estimation of QoS And QoE over WiMAX Networks: An Approach Based on Computational Intelligence And Discrete-event Simulation,” in *2011 IEEE Third Latin-American Conference on Communications.* IEEE, 2011, pp. 1–6.

[102] M. Claeys, S. Latré, J. Famaey, T. Wu, W. Van Leekwijck, and F. De Turck, “Design of A Q-Learning-based Client Quality Selection Algorithm for HTTP Adaptive Video Streaming,” in *Adaptive and Learning Agents Workshop, part of AAMAS2013 (ALA-2013)*, 2013, pp. 30–37.

[103] M. Claeys, S. Latre, J. Famaey, and F. De Turck, “Design And Evaluation of A Self-learning HTTP Adaptive Video Streaming Client,” *IEEE communications letters*, vol. 18, no. 4, pp. 716–719, 2014.

[104] M. Panda, A. Abraham, and M. R. Patra, “A Hybrid Intelligent Approach for Network Intrusion Detection,” *Procedia Engineering*, vol. 30, pp. 1–9, 2012.

[105] S. Peddabachigari, A. Abraham, C. Grosan, and J. Thomas, “Modeling Intrusion Detection System Using Hybrid Intelligent Systems,” *Journal of network and computer applications*, vol. 30, no. 1, pp. 114–132, 2007.

[106] D. S. Kim, H.-N. Nguyen, and J. S. Park, “Genetic Algorithm to Improve SVM Based Network Intrusion Detection System,” in *19th International Conference on Advanced Information Networking and Applications (AINA’05) Volume 1 (AINA papers)*, vol. 2. IEEE, 2005, pp. 155–158.

[107] S. Mukkamala, G. Janoski, and A. Sung, “Intrusion Detection Using Neural Networks and Support Vector Machines,” in *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN’02 (Cat. No. 02CH37290)*, vol. 2. IEEE, 2002, pp. 1702–1707.

[108] N. B. Amor, S. Benferhat, and Z. Elouedi, “Naive Bayes vs Decision Trees in Intrusion Detection Systems,” in *Proceedings of the 2004 ACM symposium on Applied computing*, 2004, pp. 420–424.

[109] Y. Li and L. Guo, “An Active Learning Based TCM-KNN Algorithm for Supervised Network Intrusion Detection,” *Computers & security*, vol. 26, no. 7-8, pp. 459–467, 2007.

[110] A. P. Muniyandi, R. Rajeswari, and R. Rajaram, “Network Anomaly Detection by Cascading K-Means Clustering and C4.5 Decision Tree Algorithm,” *Procedia Engineering*, vol. 30, pp. 174–182, 2012.

[111] Z.-S. Pan, S.-C. Chen, G.-B. Hu, and D.-Q. Zhang, “Hybrid Neural Network and C4.5 for Misuse Detection,” in *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 03EX693)*, vol. 4. IEEE, 2003, pp. 2463–2467.

[112] W. Hu, W. Hu, and S. Maybank, “AdaBoost-Based Algorithm for Network Intrusion Detection,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 38, no. 2, pp. 577–583, 2008.

[113] J. Zhang and M. Zulkernine, “Anomaly Based Network Intrusion Detection with Unsupervised Outlier Detection,” in *2006 IEEE International Conference on Communications*, vol. 5. IEEE, 2006, pp. 2388–2393.

[114] B. Pfahringer, “Winning the KDD99 Classification Cup: Bagged Boosting,” *ACM SIGKDD Explorations*

Newsletter, vol. 1, no. 2, pp. 65–66, 2000.

[115] S. Chebrolu, A. Abraham, and J. P. Thomas, "Feature Deduction and Ensemble Design of Intrusion Detection Systems," *Computers & security*, vol. 24, no. 4, pp. 295–307, 2005.

[116] J. Cannady, "Artificial Neural Networks for Misuse Detection," in *Proceedings of the 1998 National Information Systems Security Conference (NISSC'98)*, 1998, pp. 443–456.

[117] M. Moradi and M. Zulkernine, "A Neural Network Based System for Intrusion Detection and Classification of Attacks," in *Proceedings of the IEEE international conference on advances in intelligent systems-theory and applications*. IEEE Lux-embourg-Kirchberg, Luxembourg, 2004, pp. 15–18.

[118] J. Kim, J. Kim, H. L. T. Thu, and H. Kim, "Long Short Term Memory Recurrent Neural Network Classifier for Intrusion Detection," in *2016 International Conference on Platform Technology and Service (PlatCon)*. IEEE, 2016, pp. 1–5.

[119] A. Servin and D. Kudenko, "Multi-Agent Reinforcement Learning for Intrusion Detection: A Case Study and Evaluation," in *German Conference on Multiagent System Technologies*. Springer, 2008, pp. 159–170.

[120] M. Sánchez-Fernández, M. de Prado-Cumplido, J. Arenas-García, and F. Pérez-Cruz, "SVM Multiregression for Nonlinear Channel Estimation in Multiple-input Multiple-output Systems," *IEEE transactions on signal processing*, vol. 52, no. 8, pp. 2298–2307, 2004.

[121] P. Haffner, G. Tur, and J. H. Wright, "Optimizing SVMs for Complex Call Classification," in *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03).*, vol. 1. IEEE, 2003, pp. I–I.

[122] Y. Zhang, J. Wen, G. Yang, Z. He, and X. Luo, "Air-to-air Path Loss Prediction Based on Machine Learning Methods in Urban Environments," *Wireless Communications and Mobile Computing*, vol. 2018, 2018.

[123] D. F. F. Baptista, R. Ferrari, and R. Attux, "Channel Equalization Based on Decision Trees," *Journal of Communication and Information Systems*, vol. 35, no. 1, pp. 150–161, 2020.

[124] S. Luan, Y. Gao, W. Chen, N. Yu, and Z. Zhang, "Automatic Modulation Classification: Decision Tree Based on Error Entropy and Global-Local Feature-Coupling Network Under Mixed Noise and Fading Channels," *IEEE Wireless Communications Letters*, vol. 11, no. 8, pp. 1703–1707, 2022.

[125] Y. Wu, Y. Wang, J. Huang, C.-X. Wang, and C. Huang, "A Weighted Random Forest Based Positioning Algorithm for 6G Indoor Communications," in *2022 IEEE 96th Vehicular Technology Conference (VTC2022-Fall)*. IEEE, 2022, pp. 1–6.

[126] T. O'Shea, K. Karra, and T. C. Clancy, "Learning Approximate Neural Estimators for Wireless Channel State Information," in *2017 IEEE 27th international workshop on machine learning for signal processing (MLSP)*. IEEE, 2017, pp. 1–7.

[127] J. Huang, C.-X. Wang, L. Bai, J. Sun, Y. Yang, J. Li, O. Tirkkonen, and M.-T. Zhou, "A Big Data Enabled Channel Model for 5G Wireless Communication Systems," *IEEE Transactions on Big Data*, vol. 6, no. 2, pp. 211–222, 2018.

[128] H. He, C.-K. Wen, S. Jin, and G. Y. Li, "Deep Learning-based Channel Estimation for Beamspace mmWave Massive MIMO Systems," *IEEE Wireless Communications Letters*, vol. 7, no. 5, pp. 852–855, 2018.

[129] T. O'shea and J. Hoydis, "An Introduction to Deep Learning for the Physical Layer," *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, no. 4, pp. 563–575, 2017.

[130] G. Kechriotis, E. Zervas, and E. S. Manolakos, "Using Recurrent Neural Networks for Adaptive Communication Channel Equalization," *IEEE transactions on Neural Networks*, vol. 5, no. 2, pp. 267–278, 1994.

[131] D.-C. Park and T.-K. J. Jeong, "Complex-bilinear Recurrent Neural Network for Equalization of A Digital Satellite Channel," *IEEE Transactions on Neural Networks*, vol. 13, no. 3, pp. 711–725, 2002.

[132] M. Ghavamzadeh, S. Mannor, J. Pineau, A. Tamar *et al.*, "Bayesian Reinforcement Learning: A Survey," *Foundations and Trends® in Machine Learning*, vol. 8, no. 5-6, pp. 359–483, 2015.

[133] S. M. Aldossari and K.-C. Chen, "Machine Learning for Wireless Communication Channel Modeling: An Overview," *Wireless Personal Communications*, vol. 106, pp. 41–70, 2019.

[134] U. Cisco, "Cisco Annual Internet Report (2018–2023) White Paper," *Online](accessed March 26, 2021) https://www. cisco. com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/whitepaper-c11-741490. html*, 2020.

[135] D. Gutierrez, "The Intelligent Use Of Big Data On An Industrial Scale," 2017.

[136] R. Meulen, "Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016," *Gartner Letzte Aktualisierung*, vol. 7, p. 2017, 2017.

[137] P. Domingos and M. Pazzani, "On the Optimality of the Simple Bayesian Classifier under Zero-One Loss," *Machine learning*, vol. 29, no. 2, pp. 103–130, 1997.

[138] J. N. Morgan and J. A. Sonquist, "Problems in the Analysis of Survey Data, and a Proposal," *Journal of the American statistical association*, vol. 58, no. 302, pp. 415–434, 1963.

[139] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[140] K. Bennett and A. Demiriz, "Semi-Supervised Support Vector Machines," *Advances in Neural Information processing systems*, vol. 11, 1998.

[141] E. Fix and J. L. Hodges, "Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties," *International Statistical Review/Revue Internationale de Statistique*, vol. 57, no. 3, pp. 238–247, 1989.

[142] M. A. Kramer, "Nonlinear Principal Component Analysis Using Autoassociative Neural Networks ," *AIChE journal*, vol. 37, no. 2, pp. 233–243, 1991.

[143] C. J. C. H. Watkins, "Learning from Delayed Rewards," 1989.

[144] G. A. Rummery and M. Niranjan, *On-Line Q-Learning Using Connectionist Systems*. Citeseer, 1994, vol. 37.

[145] M. A. Alsheikh, S. Lin, D. Niyato, and H.-P. Tan, "Machine Learning in Wireless Sensor Networks: Algorithms, Strategies, and Applications," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 1996–2018, 2014.

[146] P. V. Klaine, M. A. Imran, O. Onireti, and R. D. Souza, "A Survey of Machine Learning Techniques Applied to Self-Organizing Cellular Networks," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2392–2431, 2017.

[147] A. Aljuhani, "Machine Learning Approaches for Combating Distributed Denial of Service Attacks in Modern Networking Environments," *IEEE Access*, vol. 9, pp. 42 236–42 264, 2021.

[148] N. McKeown, "Software-Defined Networking," *28th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, vol. 17, no. 2, pp. 30–32, 2009.

[149] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN: An Intellectual History of Programmable Networks," *SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[150] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore, "Reconfigurable Network Systems and Software-Defined Networking," *Proceedings of the IEEE*, vol. 103, no. 7, pp. 1102–1124, 2015.

[151] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[152] N. McKeown, *PISA: Protocol Independent Switch Architecture*, 2015, P4 Workshop.

[153] P4 Language Consortium, "P4_16 Portable Switch Architecture (PSA)," 2018.

[154] "P4 Portable NIC Architecture (PNA)," May 2021, https://p4.org/p4-spec/docs/PNA.html [Online; accessed November 2023].

[155] "Version 1.0 Switch Architecture Model," Website, https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4 [Online; accessed November 2023].

[156] "Open Tofino," https://github.com/barefootnetworks/Open-Tofino [Online; accessed November 2023].

[157] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy *et al.*, "dRMT: Disaggregated Programmable Switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 1–14.

[158] "p4c," https://github.com/p4lang/p4c [Online; accessed November 2023].

[159] "The Reference P4 Software Switch," https://github.com/p4lang/behavioral-model [Online; accessed November 2023].

[160] T. P. Morgan, *Spectrum-4 Ethernet Leaps To 800G With Nvidia Circuits*, The Next Platform, 2022, https://www.nextplatform.com/2022/04/01/spectrum-4-ethernet-leaps-to-800-gb-sec-with-nvidia-circuits/ [Online, accessed 2023].

[161] C. Zheng, M. Zang, X. Hong, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Automating In-Network Machine Learning," *arXiv preprint arXiv:2205.08824*, 2022.

[162] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "PINT: Probabilistic In-Band Network Telemetry," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 662–680.

[163] P. Wintermeyer, M. Apostolaki, A. Dietmüller, and L. Vanbever, "P^{2}GO: P4 Profile-Guided Optimizations," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020, pp. 146–152.

[164] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi *et al.*, "Flow Event Telemetry on Programmable Data Plane," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 76–89.

[165] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A Rapid Prototyping Framework for P4," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 122–135.

[166] N. Zilberman, Y. Audzevich, G. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, September 2014.

[167] G. Brebner, "Expanding The P4 Universe," Website, https://opennetworking.org/wp-content/uploads/2022/05/Expanding-the-P4-universe.pdf [Online; accessed November 2023].

[168] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[169] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The Design and Implementation of Open vSwitch," in *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, 2015, pp. 117–130.

[170] S. Laki, R. Stoyanov, D. Kis, R. Soulé, P. Vörös, and N. Zilberman, "P4Pi: P4 on Raspberry Pi for Networking Education," *ACM SIGCOMM Computer*

Communication Review, vol. 51, no. 3, pp. 17–21, 2021.

[171] "P4runtime," https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html [Online; accessed November 2023].

[172] Y. Tokusashi, H. Matsutani, and N. Zilberman, "LaKe: The Power of In-Network Computing," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2018, pp. 1–8.

[173] V. Bahl, "Unlocking the potential of in-network computing for telecommunication workloads," Website, https://azure.microsoft.com/en-us/blog/unlocking-the-potential-of-in-network-computing-for-telecommunication-workloads/ [Online; accessed November 2023].

[174] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "Tea: Enabling state-intensive network functions on programmable switches," in *Proceedings of the 2020 ACM SIGCOMM Conference*, 2020, pp. 90–106.

[175] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 121–136.

[176] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé, "P4xos: Consensus as a Network Service," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1726–1738, 2020.

[177] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-Free Sub-RTT Coordination," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 35–49.

[178] M. Liu, D. Gao, G. Liu, J. He, L. Jin, C. Zhou, and F. Yang, "Learning Based Adaptive Network Immune Mechanism to Defense Eavesdropping Attacks," *IEEE Access*, vol. 7, pp. 182 814–182 826, 2019.

[179] J. Bai, M. Zhang, G. Li, C. Liu, M. Xu, and H. Hu, "FastFE: Accelerating ML-based Traffic Analysis with Programmable Switches," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020, pp. 1–7.

[180] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. Ramos, and A. Madeira, "FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications."

[181] Q. Li, J. Zhang, T. Pan, T. Huang, and Y. Liu, "Data-driven Routing Optimization based on Programmable Data Plane," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2020, pp. 1–9.

[182] Y. Mi and A. Wang, "ML-pushback: Machine learning based pushback defense against DDoS," in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, 2019, pp. 80–81.

[183] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating Distributed Reinforcement learning with In-Switch Computing," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 279–291.

[184] F. Yang, Z. Wang, X. Ma, G. Yuan, and X. An, "SwitchAgg: A Further Step Towards In-Network Computing," in *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 2019, pp. 36–45.

[185] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling Distributed Machine Learning with In-Network Aggregation," *arXiv preprint arXiv:1903.06701*, 2019.

[186] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. M. Swift, "ATP: In-network Aggregation for Multi-tenant Learning," in *USENIX NSDI*, 2021, pp. 741–761.

[187] F. Musumeci, A. C. Fidanci, F. Paolucci, F. Cugini, and M. Tornatore, "Machine-Learning-Enabled DDoS Attacks Detection in P4 Programmable Networks," *Journal of Network and Systems Management*, vol. 30, pp. 1–27, 2022.

[188] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "IIsy: Practical In-Network Classification," *arXiv preprint arXiv:2205.08243*, 2022.

[189] C. Zheng, H. Tang, M. Zang, X. Hong, A. Feng, L. Tassiulas, and N. Zilberman, "DINC: Toward Distributed In-Network Computing," in *Proceedings of ACM CoNEXT'23*, 2023.

[190] T. Swamy, A. Zulfiqar, L. Nardi, M. Shahbaz, and K. Olukotun, "Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 329–342.

[191] G. Siracusano and R. Bifulco, "In-network Neural Networks," *arXiv preprint arXiv:1801.05731*, 2018.

[192] G. Siracusano, D. Sanvito, S. Galea, and R. Bifulco, "Deep Learning Inference on Commodity Network Interface Cards," in *Proc. Workshop Syst. ML Open Source Softw. NeurIPS*, 2018, pp. 1–8.

[193] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pForest: In-Network Inference with Random Forests," *arXiv preprint arXiv:1909.05680*, 2019.

[194] J.-H. Lee and K. Singh, "SwitchTree: In-network Computing and Traffic Analyses with Random Forests," *Neural Computing and Applications*, pp. 1–12, 2020.

[195] T. Swamy, A. Rucker, M. Shahbaz, and K. Olukotun, "Taurus: An Intelligent Data Plane," *arXiv preprint arXiv:2002.08987*, 2020.

[196] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, "Taurus: A Data Plane Architecture for

Per-Packet ML," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1099–1114.

[197] Q. Qin, K. Poularakis, K. K. Leung, and L. Tassiulas, "Line-Speed and Scalable Intrusion Detection at the Network Edge via Federated Learning," in *2020 IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 352–360.

[198] G. Siracusano, S. Galea, D. Sanvito, M. Malekzadeh, H. Haddadi, G. Antichi, and R. Bifulco, "Running Neural Networks on the NIC," *arXiv preprint arXiv:2009.02353*, 2020.

[199] G. Siracusano, S. Galea, D. Sanvito, M. Malekzadeh, G. Antichi, P. Costa, H. Haddadi, and R. Bifulco, "Re-architecting Traffic Analysis with Neural Network Interface Cards," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 513–533.

[200] B. L. Coelho, "Detecting DoS Attacks Utilizing Random Forests in Programmable Data Planes," 2020.

[201] C. Zheng and N. Zilberman, "Planter: Seeding Trees Within Switches," in *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*, 2021, pp. 12–14.

[202] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello, "Programmable Switches for In-Networking Classification," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[203] Z. Zhong, W. Wang, M. Ghobadi, A. Sludds, R. Hamerly, L. Bernstein, and D. Englund, "IOI: In-network Optical Inference," 2021.

[204] R. Friedman, O. Goaz, and O. Rottenstreich, "Clustreams: Data Plane Clustering," in *ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21)*, 2021.

[205] H. Siddique, "Towards In-Network Image Classification for Latency-Critical IoT Applications," 2021.

[206] H. Siddique, M. Neves, C. Kuzniar, and I. Haque, "Towards Network-accelerated ML-based Distributed Computer Vision Systems," 2021.

[207] K. A. Simpson and D. P. Pezaros, "Online RL in the Programmable Dataplane with OPaL," in *17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21)*, 2021.

[208] Simpson, Kyle A and Pezaros, Dimitrios P, "Revisiting the Classics: Online RL in the Programmable Dataplane," pp. 1–10, 2022.

[209] C. Zheng, B. Rienecker, and N. Zilberman, "QCMP: Load Balancing via In-Network Reinforcement Learning," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, 2023.

[210] F. Paolucci, L. De Marinis, P. Castoldi, and F. Cugini, "Demonstration of P4 Neural Network Switch," in *2021 Optical Fiber Communications Conference and Exhibition (OFC)*. IEEE, 2021, pp. 1–3.

[211] K. Friday, E. Kfoury, E. Bou-Harb, and J. Crichigno, "INC: In-Network Classification of Botnet Propagation at Line Rate," in *European Symposium on Research in Computer Security*. Springer, 2022, pp. 551–569.

[212] X. Hong, C. Zheng, S. Zohren, and N. Zilberman, "Linnet: Limit Order Books Within Switches," in *Proceedings of the SIGCOMM'22 Poster and Demo Sessions*, 2022, pp. 37–39.

[213] X. Hong, C. Zheng, S. Zohren, and N. Zilberman, "LOBIN: In-Network Machine Learning for Limit Order Books," in *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2023, pp. 159–166.

[214] M. Zang, C. Zheng, R. Stoyanov, L. Dittmann, and N. Zilberman, "P4Pir: In-Network Analysis for Smart IoT Gateways," in *Proceedings of the SIGCOMM'22 Poster and Demo Sessions*, 2022, pp. 46–48.

[215] M. Zang, C. Zheng, L. Dittmann, and N. Zilberman, "Towards Continuous Threat Defense: In-Network Traffic Analysis for IoT Gateways," *IEEE Internet of Things Journal*, 2023.

[216] M. Zang, C. Zheng, L. Dittmann, and N. Zilberman, "Advanced threat defense with in-network traffic analysis for iot gateways," *MobiUK*, 2023.

[217] M. Zang, C. Zheng, T. Koziak, N. Zilberman, and L. Dittmann, "Federated Learning-Based In-Network Traffic Analysis on IoT Edge," *Security for IoT Networks and Devices in 6G (Sec4IoT), IFIP Networking*, 2023.

[218] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello, "MAP4: A Pragmatic Framework for In-Network Machine Learning Traffic Classification," *IEEE Transactions on Network and Service Management*, 2022.

[219] G. Xie, Q. Li, Y. Dong, G. Duan, Y. Jiang, and J. Duan, "Mousika: Enable General In-Network Intelligence in Programmable Switches by Knowledge Distillation," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 1938–1947.

[220] G. Xie, Q. Li, G. Duan, J. Lin, Y. Dong, Y. Jiang, D. Zhao, and Y. Yang, "Empowering In-Network Classification in Programmable Switches by Binary Decision Tree and Knowledge Distillation," *IEEE/ACM Transactions on Networking*, 2023.

[221] A. T.-J. Akem, M. Gucciardo, M. Fiore *et al.*, "Flowrest: Practical Flow-Level Inference in Programmable Switches with Random Forests," in *IEEE International Conference on Computer Communications*, 2023.

[222] A. Ganesan and K. Sarac, "Attack Detection and Mitigation using Intelligent Data Planes in SDNs," in *GLOBECOM 2022-2022 IEEE Global Communications Conference*. IEEE, 2022, pp. 1–6.

[223] N. Moustafa. (2015) The UNSW-NB15 Dataset. https://research.unsw.edu.au/projects/unsw-nb15-dataset. [Online; accessed November 2023].

[224] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[225] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[226] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[227] M. Hamid, "Tile-Coding: An Efficient Sparse-Coding Method for Real-Valued Data," Website, https://github.com/criteo-research/tf-tile/blob/master/doc/Tile-Coding-An-Efficient-Sparse-Coding-Method-for-Real-Valued-Data.md [Online; accessed November 2023].

[228] H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.

[229] K. Das and R. N. Behera, "A Survey on Machine Learning: Concept, Algorithms and Applications," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 5, no. 2, pp. 1301–1309, 2017.

[230] Y.-W. Chang, C.-J. Hsieh, K.-W. Chang, M. Ringgaard, and C.-J. Lin, "Training and Testing Low-degree Polynomial Data Mappings via Linear SVM," *Journal of Machine Learning Research*, vol. 11, no. 4, 2010.

[231] A. Patle and D. S. Chouhan, "SVM Kernel Functions for Classification," in *2013 International Conference on Advances in Technology and Engineering (ICATE)*. IEEE, 2013, pp. 1–9.

[232] S. Suthaharan, "Support Vector Machine," in *Machine learning models and algorithms for big data classification*. Springer, 2016, pp. 207–235.

[233] Â. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading Real-time DDoS Attack Detection to Programmable Data Planes," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2019, pp. 19–27.

[234] S.-C. Wang, "Artificial Neural Network," in *Interdisciplinary computing in java programming*. Springer, 2003, pp. 81–100.

[235] M. Anthony, P. L. Bartlett, and P. L. Bartlett, *Neural Network Learning: Theoretical Foundations*. cambridge university press Cambridge, 1999, vol. 9.

[236] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary Neural Networks: A Survey," *Pattern Recognition*, vol. 105, p. 107281, 2020.

[237] Y. LeCun, Y. Bengio *et al.*, "Convolutional Networks for Images, Speech, and Time-Series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[238] E. Fix, *Discriminatory analysis: nonparametric discrimination, consistency properties*. USAF school of Aviation Medicine, 1985, vol. 1.

[239] N. S. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[240] A. J. Myles, R. N. Feudale, Y. Liu, N. A. Woody, and S. D. Brown, "An Introduction to Decision Tree Modeling," *Journal of Chemometrics: A Journal of the Chemometrics Society*, vol. 18, no. 6, pp. 275–285, 2004.

[241] S. R. Safavian and D. Landgrebe, "A Survey of Decision Tree Classifier Methodology," *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[242] D. Opitz and R. Maclin, "Popular Ensemble Methods: An Empirical Study," *Journal of artificial intelligence research*, vol. 11, pp. 169–198, 1999.

[243] T.-H. Lee, A. Ullah, and R. Wang, "Bootstrap Aggregating and Random Forest," in *Macroeconomic Forecasting in the Era of Big Data*. Springer, 2020, pp. 389–429.

[244] R. E. Schapire, "A Brief Introduction to Boosting," in *Ijcai*, vol. 99. Citeseer, 1999, pp. 1401–1406.

[245] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[246] F. Bayes, "An Essay Towards Solving A Problem in The Doctrine of Chances," *Biometrika*, vol. 45, no. 3-4, pp. 296–315, 1958.

[247] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[248] S. Wold, K. Esbensen, and P. Geladi, "Principal Component Analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[249] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[250] A. Ng *et al.*, "Sparse autoencoder," *CS294A Lecture notes*, vol. 72, no. 2011, pp. 1–19, 2011.

[251] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation Forest," in *2008 eighth ieee international conference on data mining*. IEEE, 2008, pp. 413–422.

[252] X. Zhu and A. B. Goldberg, "Introduction to Semi-Supervised Learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 3, no. 1, pp. 1–130, 2009.

[253] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[254] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv preprint arXiv:1312.5602*, 2013.

[255] J. Liu, F. Gao, and X. Luo, "Survey of Deep Reinforcement Learning Based on Value Function and Policy Gradient," *Chinese Journal of Computers*, vol. 42, no. 6, pp. 1406–1438, 2019.

[256] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[257] "APS Networks 100GbE Barefoot Tofino-based Network Switch," Website, https://www.opencompute.org/products/85/aps-netwo

rks-100gbe-barefoot-tofino-based-network-switch [Online; accessed November 2023].

[258] "Intel® Tofino 2 12.8 Tbps, 20 stage, 4 pipelines," Website, https://www.intel.com/content/www/us/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/specifications.html#tofino [Online; accessed November 2023].

[259] M. Rifai, N. Huin, C. Caillouet, F. Giroire, D. Lopez-Pacheco, J. Moulierac, and G. Urvoy-Keller, "Too Many SDN Rules? Compress Them with MINNIE," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–7.

[260] M. Kobayashi, T. Murase, and A. Kuriyama, "A Longest Prefix Match Search Engine for Multi-Gigabit IP Processing," in *2000 IEEE international conference on communications. ICC 2000. Global convergence through communications. Conference record*, vol. 3. IEEE, 2000, pp. 1360–1364.

[261] V. Rios and G. Varghese, "MashUp: Scaling TCAM-based IP Lookup to Larger Databases by Tiling Trees," *arXiv preprint arXiv:2204.09813*, 2022.

[262] K. Kannan and S. Banerjee, "Compact TCAM: Flow Entry Compaction in TCAM for Power Aware SDN," in *International conference on distributed computing and networking*. Springer, 2013, pp. 439–444.

[263] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," in *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.* IEEE, 2004, pp. 174–183.

[264] A. Bremler-Barr and D. Hendler, "Space-Efficient TCAM-Based Classification Using Gray Coding," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 18–30, 2010.

[265] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training DNNs with Hybrid Block Floating Point," *arXiv preprint arXiv:1804.01526*, 2018.

[266] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof *et al.*, "Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks," *arXiv preprint arXiv:1711.02213*, 2017.

[267] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "signSGD: Compressed Optimisation for Non-Convex Problems," in *International Conference on Machine Learning*. PMLR, 2018, pp. 560–569.

[268] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep Gradient Compression: Reducing The Communication Bandwidth for Distributed Training," *arXiv preprint arXiv:1712.01887*, 2017.

[269] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-Bit Stochastic Gradient Descent and its Application to Data-Parallel Distributed Training of Speech DNNs," in *Fifteenth Annual Conference of the International Speech Communication Association*. Citeseer, 2014.

[270] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning," *arXiv preprint arXiv:1705.07878*, 2017.

[271] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[272] J. Von Neumann, "First Draft of A Report on The EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[273] J. N. Mitchell, "Computer Multiplication and Division Using Binary Logarithms," *IRE Transactions on Electronic Computers*, no. 4, pp. 512–517, 1962.

[274] G. Cormode, "Count-Min Sketch." 2009.

[275] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen, "Runtime Programmable Switches," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 651–665.

[276] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo, "Flightplan: Dataplane Disaggregation and Placement for P4 Programs," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 571–592.

[277] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "MLPerf Inference Benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.