Towards In-Network Machine Learning on Programmable Network Devices

Changgang Zheng

Jesus College

University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Hilary 2024

This thesis is dedicated to my parents, family, supervisor, colleagues, friends, and all those whose presence illuminates my life.

谨以此文献给我的父母，家人，老师，以及所有关心和帮助我的同学和朋友们。

ABSTRACT

Machine learning (ML) services have entered various aspects of our daily lives and industrial production, with most of these services being delivered over networks. The escalating demand for ML services has resulted in increased processing requirements and a growing volume of data. This poses significant challenges to traditional solutions, demanding higher-performance computers and increased network capacity. In-network computing is a promising and scalable solution, capable of processing packets in-band at the pace of data generation and facilitating early termination of traffic. However, using in-network computing for ML inference is a new research avenue. Network devices are designed for high-performance packet processing and forwarding, and their architecture is not intended for ML.

This research addresses the design challenges of in-network ML using a bottom-up approach. It proposes mapping techniques and deployment solutions that overcome existing limitations and enable the Internet to provide in-network ML services on programmable network devices, revolutionising the way we use the Internet for ML. The first part of the thesis focuses on mapping methodologies, which offer efficient mapping solutions for diverse ML models and hardware devices. Then, to support all aforementioned models and mappings, this thesis proposes a rapid prototyping framework, accommodating diverse programmable network devices and use cases. However, while in-network ML can be deployed, its inference accuracy falls short of that achieved by server-based solutions. To address this challenge, the third part of the thesis focuses on a hybrid deployment framework, utilising a large ML model in the backend to assist small in-network models. This hybrid deployment allows close to optimal inference performance while retaining most decisions within the network. The fourth part of the thesis approaches the same problem from a different perspective,

proposing a distributed deployment framework. By jointly utilising unused resources among programmable network devices distributed across the network, this solution further scales in-network ML models and computing functions. The last part of the thesis demonstrates the application of in-network ML in anomaly detection, IoT traffic classification, financial market prediction, and load balancing use cases, showing its potential as a practical service.

DECLARATION

This thesis is submitted to the Department of Engineering Science, University of Oxford, in fulfilment of the requirements for the degree of Doctor of Philosophy. This work took place at the University of Oxford, with a three-month external visit at the Yale University, United States of America. This thesis is entirely my work, and except where otherwise stated, describes my research.

郑长刚 Changgang Zheng

Changgang Zheng

Jesus College, Hilary 2024

ACKNOWLEDGEMENTS

This thesis would not have been possible without the help and support of so many individuals and institutes. Here, I want to express my sincere gratitude.

First and foremost I would like to thank my advisor, Professor Noa Zilberman, for her trust, guidance, help, and support, which gave me the chance to pursue my doctoral studies at the University of Oxford, do research with the best supervision, and finally have this thesis. Professor Zilberman showed me a rigorous research attitude, set high standards and requirements for me, facilitated my immersion into the research community, and taught me methodologies and approaches for conducting research. I greatly appreciate her mentorship, which helps me learn a lot.

I have been fortunate to be part of the Department of Engineering Science, University of Oxford, which provided me with this invaluable learning opportunity and an excellent working environment. I am also grateful to Jesus College, which served as my "home" at Oxford, for the warmth, kindness, and support from every member (particularly staff) of the college, especially during the COVID-19 pandemic. I would like to offer special thanks to my college advisor, Professor Stephen Morris, for his consistent communication and unwavering support throughout my doctoral study.

Furthermore, I want to express my sincere appreciation to my supervisor and VMware for funding my DPhil, alleviating any financial burdens, and allowing me to focus entirely on my research.

I would also like to thank Professor Leandros Tassiulas for providing me with the opportunity and funding for a three-month visiting PhD position at the Yale Institute for Network Science (YINS), Yale University. His guidance during this period was instrumental in my study.

I would like to extend my gratitude to all my colleagues and collaborators, particularly Xinpeng Hong, Benjamin Rienecker, Liam Perreault, Sawsan EL Zahr, Radostin Stoyanov, and others from Computing Infrastructure Group; visitors to our group, Mingyuan Zang and Masoud Hemmatpour; Shay Vargaftik and Yaniv Ben-Itzhak from VMware; and Haoyue Tang from Yale University.

I would also like to offer my special thanks to Professor Nick McKeown for his time in discussing and providing me with valuable insights related to the research and future development.

I wish to express my gratitude to all anonymous reviewers for their valuable comments and reviews. Additionally, I am thankful to Professor Steven Collins and Professor Justin P. Coon for their guidance and suggestions as assessors during the Transfer of Status (ToS) and Confirmation of Status (CoS) processes.

Lastly, my heartfelt appreciation goes to my dear parents, whose support has been my strongest pillar, enabling me to pursue my studies without worry. They have shared my joys and supported me through moments of adversity. I am also thankful for the support and understanding from my large family.

This thesis signifies not just the end of my DPhil research but also the beginning of a new journey. As a close, I wish to share a poem with my friends and future myself, which has been a source of motivation throughout my doctoral journey.

老当益壮，宁移白首之心？ *Time should not alter the dream.*
穷且益坚，不坠青云之志。*Destitution should not wear away aspiration.*
酌贪泉而觉爽，*Remain calm, even when tempted.*
处涸辙以犹欢。*Stay positive, even in hardship.*
北海虽赊，扶摇可接；*The destination is far away, perseverance will take you there.*
东隅已逝，桑榆非晚。*The past is behind you, future is in your hands.*

Selected from *Preface to the Prince Teng's Pavilion* —- 王勃 Bo Wang (650–676).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AE autoencoder . . . 56
ALU arithmetic logic unit . . . 17
API application programming interface . . . 18
ASIC application-specific integrated circuit . . . 16
BMv2 behavioral model version 2 . . . 15
BNN binary neural network . . . 26
BT British Telecom . . . 7
CDF cumulative distribution function . . . 148
DDoS distributed denial of service . . . 23
DM direct-mapping . . . 36
DNN deep neural network . . . 98
DPU data processing unit . . . 16
dRMT disaggregated RMT . . . 14
DT decision tree . . . 25
EB encode-based . . . 36
ECMP equal-cost multi-path . . . 179
FPGA field programmable gate array . . . 2
HFT high-frequency trading . . . 175
IF isolation forest . . . 50
ILP integer linear programming . . . 130
INT in-network telemetry . . . 148
IoT Internet of things . . . 5
ISP Internet service provider . . . 147
KM *k*-Means . . . 27
KNN *k*-nearest neighbors . . . 51
LB lookup-based . . . 36
LOB limit order book . . . 176
LOC lines of code . . . 65
LPM longest prefix match . . . 48
MBO Market by Order . . . 175
ME micro-engine . . . 27
MITM man-in-the-middle . . . 172
ML machine learning . . . 1
MR MapReduce . . . 27
MTU maximum transmission unit . . . 109

CHAPTER 1

INTRODUCTION

With the exponential growth of data, increased computational capabilities, and continual refinement of algorithms, machine learning (ML) is experiencing a continuous enhancement of its capabilities. The services offered by ML have permeated various aspects of daily life and industrial production, playing pivotal roles in domains such as healthcare [13], financial services [56], and network management [31], thereby driving the development of societal digitisation and intelligence [118]. Behind these services are large amounts of servers distributed globally in cloud or data centres as well as the Internet that concurrently interconnect data, computing units, and users. Today, companies like Amazon, Google, Microsoft, Alibaba, and others, operate over ten thousand data centres [138], supporting an ML market of more than 26 billion USD [164]. Interconnected through the internet, these services provide predictions at rates as high as 1 million inferences per second with latency as low as milliseconds [229], constituting the current infrastructure of server (CPU&GPU)-based ML services.

Traditional server-based ML services are primarily provided in a pattern where data is transmitted from users to servers through the Internet for processing, and then sent back to users. This approach relies on the deployment of numerous servers or accelerators within centralised data centres, employing parallel processing on resource-rich servers to ease the computational requirements of end-users [82]. While this approach is effective for many applications, especially those not requiring volumetric data analysis and time-sensitive responses, it may not be suitable for all scenarios. Certain applications, particularly those with specific demands on throughput and latency, make server-based ML not cost-effective [201, 170]. The increasingly widespread application of ML services

and their large user base drive the demand for higher data rates and lower latency towards data centres. These demands can potentially reach terabits per second (Tbps) or the entire line rate [17, 96, 15] and are likely to grow further as digital transformation continues [179]. Even though accelerators such as GPUs can handle ML requests in parallel, the data still traverses network infrastructure, passes through PCI express (PCIe), and may involve the CPU on each server [82]. Processing this significant influx of data and requests presents challenges for the performance of traditional server-based ML and even potentially burdens the network [54].

The emergence of programmable network devices offers a promising alternative, known as in-network ML [218], to complement existing server-based ML solutions and address the challenges of high throughput and low latency. In-network ML deploys ML algorithms on programmable network devices, where ML services can be processed at the speed of data generation and reach line rate. With this method, ML requests no longer need to undertake a long journey to the data centre, they can be processed at the network edge closer to the data source [72]. However, these devices were initially designed to enhance network flexibility, manageability, and adaptability rather than providing computation [29]. They have limited resources and programmability, with similar but heterogeneous architecture according to design characteristics, presenting difficulties in deploying ML algorithms. While several in-network ML algorithms have been proposed [218, 119, 185], they primarily focus on software targets, field programmable gate arrays (FPGAs), or smart network interface cards (SmartNICs), supporting only a limited range of models. These targets support a restricted data rate [80], exhibiting significant performance gaps when compared to commodity off-the-shelf programmable switches [201]. Moreover, limitations in models and corresponding mappings can restrict performance, es-

pecially when facing highly diverse and complex service demands. In addition, the absence of deployment tools makes it difficult to be applied in real applications.

This thesis focuses on the design, implementation, and application of in-network ML algorithms on programmable network devices. By proposing new and improving existing algorithm mapping solutions, this research proves the feasibility of deploying ML algorithms on resource-constrained commodity targets (Chapter 3). These proposed mappings can be readily integrated using the newly designed deployment frameworks (Chapter 4). To further scale the ML performance, this research also introduces deployment techniques and strategies aimed at improving system inference performance (Chapter 5) and model scalability (Chapter 6) of in-network ML algorithms. Through the application of these proposed solutions across various scenarios (Chapter 7), this research establishes that ML algorithms can be seamlessly offloaded onto programmable network devices while ensuring inference accuracy, model scalability, and overall system performance. These collective efforts in this thesis contribute to the transformation of in-network ML into a practical computing service.

1.1 Scope

The premise of this research is that programmable network devices can be included or already exist in the given network. Under this hypothesis, the primary research questions that I attempt to answer are:

1. How to make in-network ML feasible on programmable network devices?
2. How to simplify the implementation of in-network ML?
3. How to improve the performance of in-network ML systems?

4. How to scale in-network ML models?
5. Which applications can benefit from in-network ML?

Specifically, a wider range of feasible in-network ML models will provide more options to users and applications. Fast prototyping tools can trigger use cases' interest in adopting in-network ML algorithms. Better system design and deployment techniques can adapt larger ML models to in-network systems and thus gain better inference performance. Better accuracy and system performance will allow in-network ML to be applicable to more services. The sample in-network ML use cases will guide and motivate future use cases' realisation. To summarise, the overall research question is: *how to make in-network ML a practical service?*

1.2 Contributions

This thesis answers the previous 5 research questions, proposing solutions and frameworks to realise in-network ML on programmable network devices. These designs break through the traditional belief that network devices are not suitable for ML tasks, enabling the Internet to provide ML inference services with high throughput and low latency advantages, fundamentally changing the way ML is used in the Internet. The key contributions of this research are:

1. *Developing efficient in-network ML mapping methodologies.* This research proposes three effective methodologies for in-network ML mapping. Building upon these methodologies, this research proposes seven new in-network ML algorithm mappings, improves four prior mapping proposals, and supports four additional existing ML algorithm mappings.

2. *Design of a modular in-network ML implementation framework.* This research develops an automated framework, called Planter, for one-click mapping of in-network ML models for programmable network devices. The proposed modular framework supports a wide range of ML models, target devices, pipeline architectures, and use cases, enabling rapid prototyping of in-network ML models.

3. *Providing a high-performance in-network ML system.* This research proposes a hybrid deployment solution, running a small model on a switch and a large model on the backend. With the hybrid deployment, it achieves high system-level performance (e.g., high throughput and low latency) and close to optimal ML performance (e.g., accuracy, F1 score), overcoming the limitation of standalone deployment of in-network ML.

4. *Breaking the resource barrier among programmable network devices.* This research proposes a distributed in-network computing framework, named DINC, to overcome resource limitations of programmable network devices. DINC facilitates the joint control of multiple network devices, which decomposes larger in-network computing applications, including ML, into small segments and distributed deploys them on multiple programmable network devices.

5. *Applying in-network ML to different use cases.* The proposed solutions are applied to several widely used applications including anomaly detection, Internet of things (IoT) traffic classification, financial market prediction, and load balancing. These use cases demonstrate the practicality of in-network ML and the effectiveness of the proposed mapping methodologies and frameworks.

1.3 Thesis Outline

This thesis begins with a background chapter, followed by four chapters discussing in-network ML techniques and their implementation, one chapter discussing use cases, and finally, ending with a conclusion chapter. A brief overview of the chapters is as follows.

Chapter 2: Background and Related Work. Chapter 2 provides the background and related work for the thesis. It covers an introduction to programmable network devices, defines the scope of in-network ML, and conducts a literature review of in-network ML.

Chapter 3: In-network Machine Learning Mapping. Chapter 3 is dedicated to the mapping of ML models to programmable network devices. It presents three principal methodologies for in-network ML mapping, leading to the development of seven new in-network ML inference algorithm mappings, the improvement of four existing mappings, and the implementation details of four other existing mappings. The evaluation in this chapter analyses the theoretical memory (table entry) and stage consumption of models realised using these mapping methodologies. A detailed evaluation is included in the following chapter.

Chapter 4: Automated Deployment Framework. Chapter 4 introduces a framework named Planter for rapid deployment of in-network ML algorithms. Planter facilitates the selection of ML models, architectures, targets, datasets, and use cases, and automatically generates, compiles, loads, and runs the mapped ML models on the selected target. With the help of the Planter framework, this chapter compares all existing in-network ML solutions and demonstrates the benefits of the proposed approaches in terms of resource utilisation and inference performance. Additionally, the evaluation reports the scalability of proposed and

existing in-network ML algorithms.

Chapter 5: Hybrid In-network Machine Learning. Chapter 5 introduces a hybrid in-network ML deployment. First, the chapter shows the challenges for achieving close-to-optimum performance on resource-constrained commodity hardware. Then it shows that by employing a small in-network ML model on the network device and large ML models over the backend server, an in-network ML system can achieve close to optimal inference accuracy. Finally, this chapter reports the accuracy and performance (throughput & latency) of hybrid deployment using ensemble models (e.g., random forest (RF) and XGBoost (XGB)) in use cases such as anomaly detection and financial market prediction.

Chapter 6: Distributed In-network Computing. Chapter 6 focuses on distributed in-network computing (DINC). First, the chapter states the motivation for distributed deployment and explains routing requirements in a network. Next, the chapter introduces an integer linear programming model for deployment optimisation on multi-path networks with resource-constrained network devices. Then, the chapter details the design of a modular DINC framework for distributed in-network computing. Finally, the chapter reports the results of distributed deploying seven in-network computing programs (e.g. five ML algorithms, one caching, and one load balancing) on two practical networks (Folded-Clos data centre and British Telecom (BT) wide area network (WAN)) with five different setups. Evaluations show that DINC is an important first step towards the efficient utilisation of data plane resources through distributed in-network computing.

Chapter 7: In-network Machine Learning Applications. Using the solutions presented in Chapters 3 - 6, Chapter 7 shows our exploration of in-network ML use cases, including anomaly detection, IoT traffic classification, financial market

prediction, and load balancing. This chapter also discusses the characteristics of in-network ML use cases, with a particular emphasis on how to optimally apply in-network ML.

Chapter 8: Conclusion. This final chapter summarises the contributions of this research, discussing its benefits and limitations, and pointing out future directions of in-network ML research.

1.4 Publications List

The following publications comprise the core of this thesis, which focuses on the theory and implementation of in-network ML algorithms. I am the primary contributor to these works, with guidance and support from my supervisor, Noa Zilberman. Authors marked with "$*$" contributed equally to the design, implementation, evaluation, and manuscript. Other collaborators mainly assisted with the evaluation and manuscript preparation.

1. **Changgang Zheng**, Hong X, Ding D, Vargaftik S, Ben-Itzhak Y, Zilberman N. **In-Network Machine Learning Using Programmable Network Devices: A Survey**. *IEEE Communications Surveys & Tutorials*, 2023.

   This paper provides a survey of in-network ML, which is the basis of Chapter 2.

2. **Changgang Zheng**, Zang M, Hong X, Perreault L, Bensoussane R, Vargaftik S, Ben-Itzhak Y, Zilberman N. **Planter: Rapid Prototyping of In-Network Machine Learning Inference**. *arXiv preprint* [245], 2022. ACM SIGCOMM Computer Communication Review (CCR) [246], 2024.

This paper proposes three mapping methodologies and mapping solutions of more ML models, which are the basis of Chapter 3. The automated deployment framework (Planter) is the basis of Chapter 4.

3. **Changgang Zheng**, Xiong Z, Bui TT, Kaupmees S, Bensoussane R, Bernabeu A, Vargaftik S, Ben-Itzhak Y, Zilberman N. **IIsy: Hybrid In-Network Classification Using Programmable Switches**. *arXiv preprint* [243], 2022. *IEEE/ACM Transactions on Networking* [244], 2024.

   This paper sets the basis for practical in-network ML, which is the key component in Chapter 3. The hybrid in-network ML framework (IIsy) is the basis of Chapter 5.

4. **Changgang Zheng**, Zilberman N. **Planter: Seeding Trees Within Switches**. In *Proceedings of ACM SIGCOMM Poster and Demo Sessions*, 2021.

   This extended abstract proposes solutions to map ensemble tree models, which is a key component in Chapter 3.

5. **Changgang Zheng**, Tang H, Zang M, Feng A, Hong X, Tassiulas L, Zilberman N. **DINC: Toward Distributed In-network Computing**. In *Proceedings of the ACM on Networking & CoNEXT '23*, 2023.

   This paper proposes a distributed in-network computing framework (DINC), which is the basis to Chapter 6.

6. **Changgang Zheng**$*$, Rienecker B$*$, and Zilberman N. **QCMP: Load Balancing via In-network Reinforcement Learning**. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing (FIRA)*, 2023.

   This paper proposes in-network reinforcement learning (RL) for load balancing, which is a component of Chapters 3 & 7.

The following publications are related to the use cases of in-network ML algorithms, which apply the outcome of previous papers. For these publications, I contributed to the conception and validation of the ideas, the research of prior work, the framework design, the implementation and running of experiments, and the writing of the manuscript.

7. Zang M, **Changgang Zheng**, Dittmann L, Zilberman N. **Towards Continuous Threat Defense: In-Network Traffic Analysis for IoT Gateways**. *IEEE Internet of Things Journal (IoTJ)*, Volume: 11 Issue: 6, 2023.

   This paper proposes an in-network ML application with runtime update ability for IoT traffic analysis, which is a component of Chapter 7.

8. Zang M, **Changgang Zheng**, Stoyanov R, Dittmann L, Zilberman N. **P4Pir: In-network Analysis for Smart IoT Gateways** (early work of Publication 7). In *Proceedings of ACM SIGCOMM Poster and Demo Sessions*, 2022.

   This extended abstract proposes an in-network ML application for IoT traffic analysis, which is a component of Chapter 7.

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

   This paper proposes a federated in-network ML application for IoT traffic analysis, which is a component of Chapter 7.

10. Hong X, **Changgang Zheng**, Zohren S, Zilberman N. **Accelerating Machine Learning for Trading Using Programmable Switches**. In *27th European Conference on Artificial Intelligence (ECAI)*, 2024.

    This paper proposes in-network ML solutions for financial market prediction with hybrid deployment, which is a component of Chapter 7.

11. Hong X, **Changgang Zheng**, Zohren S, Zilberman N. **LOBIN: In-Network Machine Learning for Limit Order Books**. In *IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*, 2023.

    This paper proposes in-network ML solutions for financial market prediction based on commodity switches, which is a component of Chapter 7.

12. Hong X, **Changgang Zheng**, Zohren S, Zilberman N. **Linnet: Limit Order Books Within Switches** (early work of Publication 9). In *Proceedings of ACM SIGCOMM Poster and Demo Sessions*, 2022.

    This paper proposes in-network ML solutions for financial market prediction, which is a component of Chapter 7.

13. Hong X, **Changgang Zheng**, Zilberman N. **In-Network Machine Learning for Real-Time Transaction Fraud Detection**. In *27th European Conference on Artificial Intelligence (ECAI)*, 2024.

    This paper proposes in-network ML solutions for detecting transaction fraud, which is a component of Chapter 7.

14. Hemmatpour M, **Changgang Zheng**, Zilberman N. **E-Commerce Bot Traffic: In-Network Impact, Detection, and Mitigation**. In *Proceedings of 27th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, 2024.

    This paper proposes in-network ML solutions for detecting e-commerce bot traffic, which is a component of Chapter 7.

CHAPTER 2

BACKGROUND AND RELATED WORK

In recent years, the emergence of software-defined networking (SDN) and programmable network devices has triggered offloading of server-based applications onto network devices [109]. These offloaded applications leverage the distinct deployment location and capabilities of processing in-network, which can enhance service performance while reducing the load on the network. ML is one of the popular server-based applications and has gained widespread adoption across various fields such as networking, finance, and healthcare, among others [13, 56, 31]. The continuous emergence of diverse businesses and escalating service demands pose challenges to traditional server-based ML systems [77, 82, 17]. In-network ML, as a prospective solution, involves deploying ML algorithms closer to the data source within the network, offering low-latency and high-throughput ML services [218]. In-network ML has a divergence in service provision compared to server-based approaches, which serves as a promising complement to server-based ML systems. However, offloading ML models for inference on programmed network devices presents challenges [214]. In this chapter, I provide an overview of programmable network devices (§2.1), discuss the motivations of in-network ML (§2.2-2.4), provide a review of state-of-the-art researches (§2.5-2.6), and show the existing gaps (§2.7). These aspects serve as the foundation and starting point of this research. The content of this chapter was published in [239, 243].

2.1 Programmable Network Devices

SDN [134, 57] realises centralised network operation primarily by separating the data plane and control plane and using a controller to jointly manage a set of network data planes. This vision started with OpenFlow [136], which employs a standardised communication interface for data plane reconfigurability. While OpenFlow facilitates network programmability, its initial focus was on modifying forwarding and routing rules or policies, offering a limited selection of data plane functionality [256]. The subsequently introduced reconfigurable match-action table (RMT) model further enhances data plane programmability [30], empowering users such as network operators to dynamically alter their data plane's functionality. Currently, with RMT-based network devices, network applications can be executed in an "in-network" manner [109]. Specifically, the program is offloaded from servers, and is implemented and executed within the data plane. This method allows in-network applications to operate at line rate, do the data processing in band, and achieve low latency. With the help of in-network computing, applications can adapt to increasing cloud network infrastructure demands. As the base of in-network computing, in this section, I provide an overview of programmable network devices and show their programming workflow.

2.1.1 Protocol-Independent Switch Architecture

A programmable device architecture defines the way to make network devices programmable. Many data plane architectures are similar to and even originate from protocol independent switch architecture (PISA) [135]. Figure 2.1 shows

PISA, a basic pipeline architecture for programmable data planes evolving from the RMT model [30]. PISA allows network users to tailor packet processing logic within the data plane without hardware modifications, making them independent of vendor-provided, fixed sets of protocols. Other architectures also exist, such as those based on disaggregated RMT (dRMT) [44], which have a similar concept of a match-action (M/A)-based pipeline with RMT-based architectures. Given its widespread adoption, this study applies to PISA-related architectures.

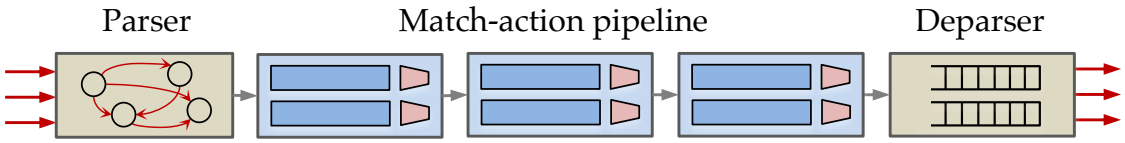


Figure 2.1: Protocol-Independent Switch Architecture (PISA).

The PISA architecture has three key building blocks: a parser, a deparser, and a M/A pipeline. When there is an incoming packet, it first goes through the parser. The parser is a state machine that extracts a sequence of fields from the packet, called packet header vector (PHV). The PHV contains fields from both packet headers (e.g., Ethernet, IP, VLAN, TCP/UDP, and other header fields defined by users) and intrinsic metadata (e.g., ingress and egress ports). After the extraction, the packet's payload is stored in a buffer outside the pipeline. The extracted vector (in PHV) is processed within the pipeline in a sequence of logical stages by using M/A tables. Each logical stage allows a fixed number of M/A operations, where a key (an input field from PHV or metadata) undergoes a lookup in a table, leading to a corresponding action. This enables the processing of packets in a predetermined way. Finally, the deparser reconstructs PHV fields, and assembles them with the 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) [149], portable NIC architecture (PNA) [11], SimpleSumeSwitch [95], v1model [9], and Tofino native architecture (TNA) [4]. Besides the PSA and v1model, most other architectures are tailored for specific targets and incorporate vendor-specific features. All these mentioned targets are similar in general, with distinctions primarily focusing on the number and order of building blocks (Figure 2.1) employed. This study is mainly based on programmable network devices with the v1model or TNA architecture. The target devices of these architectures are detailed in the following Section 2.1.3.

2.1.2 Data Plane Programming Language

Programming protocol-independent packet processors (P4) is a domain-specific language used for network packet processing [29]. P4 provides a flexible and customisable approach to packet parsing, matching, forwarding behaviours, and processing methodologies on network devices. A key feature of the P4 language is protocol independence, enabling the use of the same P4 program on different network devices with minimal or no code modifications. Furthermore, the P4 language supports interaction and control 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 with PISA-based architecture are programmed by 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., behavioral model version 2 (BMv2), exist. The open source p4c [5] 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 file that is executable on a specific target, based on a given data plane architecture. Specifically, P4 targets are programmable network infrastructures, P4 architectures are the pipeline structures that define how P4 is applied to the hardware, and P4 is the language that defines the actual packet processing logic. Till now, P4 have two versions. The $P4_{16}$ is currently the commonly used version of P4, and $P4_{14}$ is deprecated.

2.1.3 P4 Targets

The P4 language supports a variety of packet processing targets (programmable network devices), including SmartNICs, data processing units (DPUs), FPGAs, hardware switch-ASICs, and software switches [81]. P4-based programmable hardware switches are available from multiple equipment vendors, such as Intel Tofino, NVIDIA Spectrum, and Cisco Silicon-One. DPU and SmartNICs vendors include NVIDIA BlueField [3], Intel IPU [1], AMD Pensando [7], Netronome NFP [2] 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. BMv2 [8] 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 [150]. In the following, I discuss some commonly used P4 platforms of different types.

Switch-ASIC. Switch-ASICs are specialised devices designed specifically for packet forwarding in network switches, utilising an application-specific integrated circuit (ASIC) to achieve high-speed and low-latency switching. Pro-

grammable switch-ASICs introduce programmability into the switch pipeline without compromising on performance. They are characterised by high throughput and low latency. Current switch-ASICs exceed 50Tbps and can process tens of billions of packets per second [139], with sub-microsecond latency. Using Intel's Tofino switch-ASIC [4] as an example, there are multiple *pipes* (each composed of an *ingress* and an *egress* pipeline), with multiple ports associated with each pipe. To provide guaranteed throughput and latency performances, ensure independence, and avoid conflicts and concurrent access issues, state is not shared between pipes. For instance, register values in one pipe (either ingress or egress pipeline) cannot be read by programs running in other pipes. 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 limited number of *stages*, which can execute operations such as 1) using M/A to lookup keys in tables and take corresponding actions, 2) utilising counters, meters, or registers, and 3) computing values using arithmetic logic units (ALUs). Each stage has limited hardware resources, which is the biggest challenge for offloading novel network functions into programmable targets as described in most existing works [245, 23, 210, 252].

FPGA. FPGAs are configurable integrated circuits that can be programmed to perform specific functions, providing flexibility and reconfigurability for implementing custom hardware designs. FPGAs were early demonstration targets for P4-based network devices, with works such as P4FPGA [206]. Other example targets include NetFPGA [254] running P4→NetFPGA [95] and AMD Alveo running OpenNIC [32]. 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 hun-

dreds of Gbps, lower than high-end switch-ASIC but higher than CPU-based targets. Different from previous targets, FPGAs allow users to design their own P4 architecture. For example, P4→NetFPGA [95] uses the SimpleSumeSwitch architecture, which uses only a single pipeline, without separation to Ingress and Egress.

Software Switch. Software switches realise programmable network pipeline virtually on standard CPUs, rather than on specialised hardware. To overcome performance barriers of CPUs, kernel bypass techniques are often employed. For instance, T4P4S compiles a P4 program (with v1model architecture) and loads the compiled program to DPDK accelerated data planes [205]. Similarly, compiled P4 programs using PSA architecture can run on software targets such as PSA-eBPF, P4 DPDK, and Open vSwitch (OVS) [156]. As a model used to explore P4 functionality, the P4 behavioral model (BMv2) is widely used. It supports both v1model and PSA architectures. Another hybrid target is P4Pi [115], which enables P4 execution on a Raspberry Pi. P4Pi supports T4P4S and BMv2, and offers an affordable hardware target suitable for educational and research purposes. Although software switches typically exhibit lower performance compared to hardware targets, their widespread availability makes them a common choice for design verification.

2.1.4 Control Plane

Control plane manages the runtime behaviour of P4 targets via an application programming interface (API). The API is supported by a device driver or an equivalent software component. P4Runtime [6] is a common control plane specification that allows to control or configure the P4 program running on the data

plane of programmable network devices. Figure 2.2 illustrates the main control

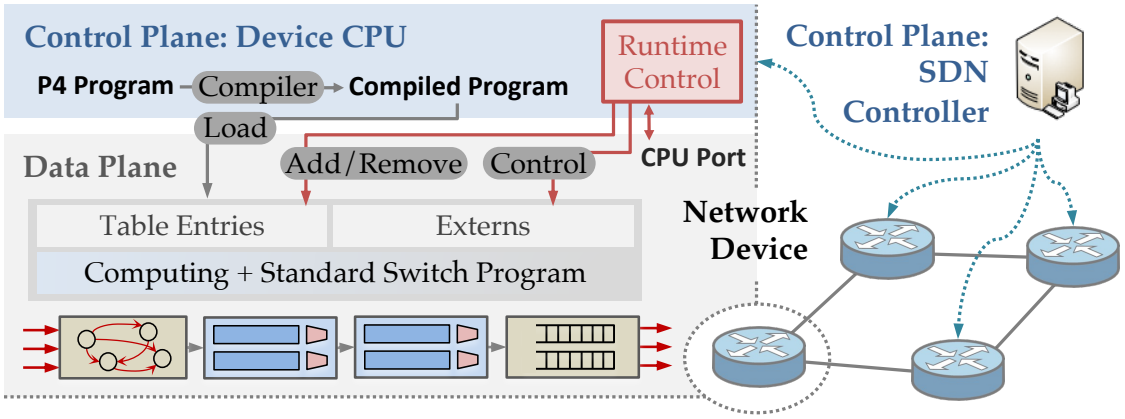


Figure 2.2: Control plane and data plane interaction.

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. The runtime operations can be realised on the device level control plane (e.g., switch CPU). The control plane of all programmable network devices can be further linked to a centralised controller (e.g., SDN controller) for joint management.

2.2 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 [201]. Realisation of in-network computing allows networks to become part of available computing resources. It provides better in-

tegration of communication and computing resources when diverse application requirements need to be addressed [202]. Microsoft Azure highlighted the potential of in-network computing for telecommunication workloads [18], 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 [111].

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 [103] 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 [51] 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 centres. NetChain [102] uses switch-ASIC to store data and process queries in-band (within the data plane), which provides scale-free sub-RTT coordination in data centres. These in-network computing applications are mainly implemented on a standalone device, yet they can further co-deploy services with CPU [169] or other programmable network devices [193].

2.3 Scope of In-network Machine Learning

The combination of machine learning and networking can be classified into three forms: *general ML*, *network-assisted ML*, and *in-network ML*. These forms can be

primarily distinguished based on the location of the ML inference decision, as depicted in Figure 2.3. The detailed definition of each form is listed as follows:

- *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 & preprocessing (more detailed features collected inside 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.

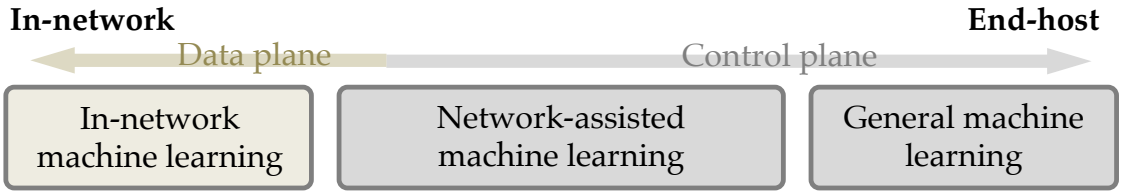


Figure 2.3: The difference between general ML, network-assisted ML and in-network ML (The arrow indicates where the *ML inference decision* is made).

This thesis explores the field of in-network ML. Network-assisted ML techniques such as in-network aggregation [171, 116] and feature extraction [128, 19, 21, 122] are outside this scope.

In-network ML is suitable for a range of applications. For example, it can be used for cyber security applications, where in-network ML classifies traffic in real time and detects network attacks. In this high-throughput application example, in-network ML can classify the traffic at line rate and drop any malicious

packets. A second example is using in-network ML for latency-sensitive applications, such as stock price movement prediction. In this example, the switch can forecast whether the price will rise or fall and recommend a buy, sell, or hold operation directly within the network to reduce latency, improving overall profitability. Note that the outcomes of this research are not confined to these two examples and have broad applicability.

2.4 Motivation: the 3-Ls

The benefits of running in-network ML using programmable network devices can be summarised as the 3-Ls: **L**ocation, **L**atency, and **L**oad.

M1: Location. Any cloud-processed user-generated data goes through the network first. This means that any information that needs to be classified, is already processed by switches. Extending this processing to include classification is a natural step. Network switches are located at every point of the network (e.g., edge, data centre, point of presence), providing early access to data as well as visibility into the aggregation of data sources. In addition, network switches are already part of the infrastructure carrying user data and do not need to be newly added. There are no cost or space overheads, beyond existing network requirements, unlike other accelerators (e.g., GPUs, middleboxes).

M2: Latency. The latency from a data-generating node to a processing node is always higher than the latency to any network devices along the path between these nodes. Within a data centre, every hop avoided through in-network ML saves hundreds of nanoseconds to microseconds [255]. In WAN, propagation delay can be in the order of milliseconds, therefore, classifying next to the end-user or at the edge can significantly reduce latency. This is important for time-

sensitive applications, such as financial transactions, industrial control [114], smart transportation systems, and latency-critical IoT applications [173]. As discussed later, automatically converting and loading trained ML models to (local and remote) network switches, can speed up further the reaction to events in the network and shorten the time for detection and mitigation.

M3: Load. Network switches can process billions of transactions per second and do so while providing high power efficiency [201]. The rate of classification decisions by an end-host or an accelerator is bounded by the data rate of the attached network device. A fully realised in-network ML offers both the high throughput of a network switch and the reduction of the load on the backend. In-network ML can significantly reduce the amount of traffic to servers, and that requires further processing. In some use cases, such as mitigating distributed denial of service (DDoS), dropping malicious traffic close to the source can dramatically reduce both network and server loads.

2.5 State-of-the-Art of In-network Machine Learning

Since 2017, researchers have been exploring implementing the inference process of ML algorithms directly within programmable network devices. Work prior to this research can be primarily categorised into three types: 1) tree-based models, 2) neural networks (NNs), and 3) other traditional ML models. All in-network ML works to date are listed in Table 2.1, roughly by type and based on their development timeline. There are two noteworthy points to highlight in this table: 1) The volume of publications in this domain has consistently grown since 2018; 2) Existing works predominantly support only a single ML model (> 75%) as opposed to multiple models. To better understand the background of in-network

No	Scheme	Algorithm	Platform
1	IIsy (Short) [218]	DT, KM, SVM, and NB	NetFPGA-SUME and BMv2
2	pForest [33]	RF	Tofino
3	SwitchTree [119]	RF	BMv2
4	BACKORDERS [46]	RF	BMv2
5	N2Net [183]	BNN	RMT-like Switch Pipeline
6	BaNaNa Split [169]	BNN	SmartNIC
7	toNIC [186]	BNN	Netronome SmartNIC
8	Qiaofeng et al. [160]	BNN (Federated learning)	Netronome SmartNIC and BMv2
9	N3IC [185, 184]	BNN	Netronome SmartNIC, NetFPGA, and BMv2
10	Taurus [196, 195]	M-RA^2: DNN, SVM, KM, and LSTM	Modified ASIC
	Works after The Initial Work of This Study		
11	Clustreams [66]	*k*-NN Clustering	Spectrum-3 switch
12	Mousika [215]	DT	Tofino
13	pHeavy [232]	DT	BMv2 and Tofino
14	INC [64]	DT	Tofino
15	Bruno et al. [213]	RF	Netronome SmartNIC and BMv2
16	MAP4 [214]	DT, RF	Netronome
17	NetPixel [177, 178]	DT, CNN	BMv2
18	IOI [248]	NN	Modified ASIC
19	Paolucci et al. [152]	NN	BMv2
20	Homunculus [197]	DT, KM, SVM, and NN	Modified ASIC
21	OPaL [180, 181]	Temporal-difference RL algorithms (SARSA)	Netronome SmartNIC
	Works Resulting from This Study		
22	Planter [247, 245, 246]	SVM, NB, KM, DT, RF, XGB, IF, KNN, PCA, AE, and BNN	Tofino, Tofino2, BMv2, T4P4S, P4Pi, and FPGA
23	IIsy [243, 244]	DT, RF, XGB, KM, SVM, and NB	NetFPGA-SUME, Tofino, and BMv2
24	DINC [242]	Distributed NB, SVM, DT, RF, and XGB	BMv2 and Tofino
25	QCMP [241]	Q-Learning	Tofino and BMv2
26	Linnet [90]	NB, DT, RF, XGB	BMv2
27	LOBIN [91]	KM, KNN, DT, RF, XGB	BMv2, Tofino, and Tofino2
28	P4Pir [228, 226]	DT, RF	P4Pi
29	FLIP4 [227]	Federated XGB	P4Pi

Table 2.1: Summary of in-network ML Algorithms until early 2023. All the used acronyms are listed in Abbreviations.

ML, this section previews an overview of previous works, outlines the process of their evolution, and discusses their inherent limitations. A more thorough explanation of all in-network ML works can be found in our survey paper [239].

2.5.1 Tree Based Ensemble Models

The deployment of decision tree (DT) and tree-based ensemble models on programmable network devices was introduced by IIsy [218] and pForest in 2019 [33], and is followed by many other works such as Planter, SwitchTree, and BACKORDERS [247, 119, 46].

Within these efforts, the implementation of the base tree model can be categorised into two technical routes, as shown in Figure 2.4. The first route is using a table per depth. This route is led by pForest [33] which hierarchically realises decision tree and random forest, employing a M/A stage for each depth (level) in the tree. All branches of the tree at the same depth are stored in a common M/A table. pForest validated this mapping approach on both BMv2 and Tofino platforms using a flow classification use case. Subsequent works, such as SwitchTree [119] and BACKORDERS [46], adopted this solution for the model on anomaly detection use cases. Evaluation results show that this method has low memory consumption but requires a substantial number of pipeline stages, which constrains the model size when deployed on commodity hardware. The second route is using a table per feature. IIsy [218] lead this second route, which implemented decision tree through feature mapping. This approach applies an M/A table per feature to transform input features into codes. Subsequently, a decision table is utilised to map these codes to the output class. The IIsy-based solution, while less intuitive than pForest, offered significant reductions in stage

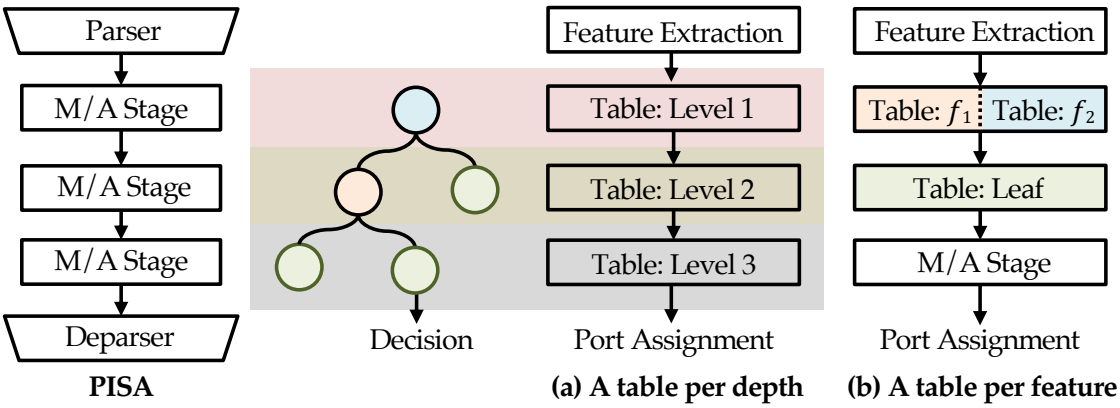


Figure 2.4: Two technical routes of implementing tree models: (a) using a table per depth [33] and (b) using a table per feature [218].

consumption. However, IIsy only provided an FPGA-based solution for decision tree, without extending decision tree to ensemble tree models. Moreover, the memory-intensive nature of the decision table and feature tables used for mapping may potentially result in table size explosions, especially when applied to switch ASIC.

2.5.2 BNN Based Models

The implementation of neural networks in programmable network devices involves the utilisation of binary neural networks (BNNs) to transform the neural network into a format compatible with pipeline operations. N2Net was the first work to provide a solution to map the forward path of the BNN to RMT switches [183]. However, due to model complexity, this model is challenging to deploy on commodity hardware.

Two approaches appear to mitigate this challenge. The first approach uses targets that support more stages. The software switch such as BMv2 has no stage limit and is suitable for BNN implementation. Several existing works such as

Qiaofeng et al. [160], toNIC [186], and N3IC [185] apply BMv2 in their functionality evaluations. However, this method is at the expense of system performance such as throughput and latency. The second approach uses specialised hardware that allows complex operations to reduce stage consumption. Specialised hardware components, such as micro-engines (MEs) in SmartNICs [185, 169] or customised MapReduce (MR) blocks using FPGA, are used to conduct complex operations in neural network [195]. In this work, I mainly focus on using commodity off-the-shelf devices. The design and application of specialised hardware or externs for neural network is not our focus.

2.5.3 Other ML Models

Traditional ML algorithms, including support vector machine (SVM), naïve Bayes (NB), and *k*-Means (KM) were initially proposed by IIsy [218]. To avoid complex operations in the algorithm, after model training, intermediate results for each potential input are stored in M/A tables. IIsy made trade-offs between accuracy and feasibility, so as to realise 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 in the table generation stage. The evaluation on BMv2 and NetFPGA verified the functionality of the generated models. Regarding resource utilisation, this approach requires a greater number of pipeline stages compared to the maximum number of tables utilised for storing intermediate results, due to interdependency (when intermediate results from all tables are employed in each formula). The memory consumption depends on the needed number of inputs for each intermediate result calculation and the value range per input feature.

For IIsy-based solutions, the consumption of stages cannot be ignored, particularly when it coexists with other network functions. Moreover, the possibility of table size explosion when reducing stages hinders its applicability in use cases.

2.6 Deployment Scenarios

In-network ML can be used in different deployments, such as data centre networks [33], WAN [119], and edge computing [169]. Many of the works focus on the technology, it can be expected 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 centre or the edge (where computation time dominates over propagation time). I divide these works into three types of deployment [243] (to be clear that this is already our work): native switch, endpoint accelerator, and SmartNIC.

Switch/Router. 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.

Endpoint Accelerator. A programmable network device 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 host 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.

SmartNICs. 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 lower throughput than a switch.

2.7 The Gap in In-network Machine Learning

Researchers have attempted to effectively port ML models to programmable network devices. However, these attempts were preliminary and limited in practical application. The gap in existing solutions can be summarised into the following five points:

Resource Restricted Hardware. Many previous approaches mainly prototyped their mappings on software targets, which require significant changes or reduced model sizes when deployed on hardware targets [33, 243, 119]. The design of hardware programmable network devices, especially off-the-shelf switches, primarily focuses on high-speed processing and switching of packets, resulting in constrained general computing capabilities compared to software targets, FPGAs, and SmartNICs. For instance, the current mainstream switch-ASICs, have

1) limited data types, where floating-point numbers are not directly supported; 2) limited arithmetic operations, where multiplication and division are not allowed; 3) limited stages, where loop operations are not allowed; and 4) limited memory, in the order of tens of megabytes. The combination of these limitations poses challenges to the algorithm mapping of in-network ML.

Lack of Algorithm Mapping. Current in-network ML implementations are relatively simple and limited in scope. Specifically, some common problems include 1) limited supported models [160, 183], 2) restricted mapping options for each model [119], 3) table size explosion (such as applying a single M/A table to directly map input features to labels) [218], and 4) pipeline stage explosion (typically including algorithms implemented with a similar logic as it is in the server) [33]. These problems will be further magnified when running in-network ML on commercial switches, leading to problems such as restricted model and mapping selection, limited model size with poor scalability, and suboptimal performance. The combination of these challenges highlights the need for developing general and efficient algorithm mapping solutions. Simultaneously, to achieve good scalability and performance under constrained resources, realising the optimised algorithm mapping for in-network ML to balance and even reduce memory and stage consumption is essential.

Inefficient Model Implementation. The deployment of in-network ML algorithms is still a challenge to existing works. There is a lack of exploration in 1) the rapid deployment of individual models, 2) the seamless transitioning of models across different architectures and target devices, and 3) the swift integration of algorithms with use cases. Existing works mainly employ manually written data plane code [218, 119]. The absence of deployment frameworks significantly complicates ML deployment and impedes comparisons among various models and

mapping solutions. Moreover, without such a framework, it is hard to realise the swift comparison of in-network ML algorithms across diverse hardware deployments for various use cases. These gaps make the adoption of in-network ML notably complex in real applications.

Constrained System Performance. Existing In-network ML solutions fall short of achieving performance comparable to ML on dedicated servers with large model sizes [33, 46]. Despite continuous improvements in the deployment techniques that may help narrow this performance gap, the inherent resource constraints of each individual in-network device ultimately limit the scalability and performance of in-network ML. It is challenging to build an in-network ML system that meets the stringent demands for the classification performance of current applications. This is as the in-network ML system needs to achieve comparable classification performance to server-based large models while maintaining the advantage of high throughput and low latency.

Limited Use Cases. While in-network ML can provide ML services with high throughput and low latency within the network along the data path, it remains an emerging field and has only been applied to a limited range of use cases [119, 33, 46, 218]. The current research primarily focuses on algorithmic and networking implementations. To make in-network ML an effective ML service, it is critical to propose and validate more robust use cases that demonstrate its effectiveness. Migration of use cases from traditional ML to in-network ML is not an easy task. On one hand, the use case should be carefully chosen to match the location and performance of in-network ML. On the other hand, use cases may need to be implemented inside the network on different architectures and programming languages. Identifying applicable use cases and implementing them on programmable network devices is challenging.

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

- **Decision tree and tree-based ensemble models:** Although tree-based models were considered the most mature ones in terms of data plane implementation compared to other models, existing solutions have limitations. The *using a table per depth* approach [33, 119] exhibits poor scalability on hardware targets and sensitivity to the number of input features. However, this approach is not sensitive to the number of branches and leaf nodes. In most cases, the deployed decision tree with this approach consumes a small amount of memory, but a large number of stages. The *using a table per feature* approach [247, 243] can accommodate a large number of features, with generally lower stage consumption. However, this method only supported decision tree. Additionally, in scenarios where tree depth is large, the number of branches and leaf nodes increases and can consume significant memory.
- **BNN based models:** While previous works presented solutions 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 [185, 186]. Also, early works [183] did not mention the performance matrix of their solutions compared to other end-host deployed ML benchmarks.
- **Other ML models:** Some works [218, 196] have proposed preliminary general solutions and ideas for ML deployment, but only 5 ML algorithms have been demonstrated. Moreover, the existing mappings were on FPGA rather than switch-ASIC. The mapping of more ML algorithms remains unexplored, 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.

2.8 Summary

In this chapter, I introduced the background relevant to this thesis, in particular, the existing programmable network devices and the state-of-the-art in-network ML research. Programmable network devices are often based on PISA-based architectures and can be programmed using the P4 language (§2.1). The appearance of these devices created opportunities for in-network computing (§2.2). This computing approach can benefit from the distinctive deployment location and inherent high throughput and low latency characteristics of network devices, offering an alternative solution complementary to the existing server-based computing models. ML workloads, as increasingly popular computational tasks, prompted the expected deployment of ML algorithms within networks, forming in-network ML (§2.3). Similar to in-network computing, in-network ML has the benefit of 3-Ls, which are location, load, and latency (§2.4). While several ML algorithms are already mapped to programmable network devices (§2.5-2.6), many limitations remain in terms of supported model types, model adaptability, model scalability, deployment techniques, and deployment strategy (§2.7). The practical and massive adoption of in-network ML is restricted due to these gaps. In the following chapters, I introduce solutions and frameworks to bridge gaps and realise practical and effective in-network ML services.

CHAPTER 3

IN-NETWORK MACHINE LEARNING MAPPING

The preceding chapter discussed the works of previous research that implements in-network ML algorithms. Owing to the complexity of ML inference algorithms, the existing mappings mainly cover a small number of (relatively simple) models [33, 46]. Moreover, existing efforts have predominantly opted for programmable network devices with relaxed resource constraints, high programmability, and lower throughput, rather than achieving better performance on resource-constrained commodity hardware devices [119, 218]. To enable a broader application of in-network ML across various services, there is a need for more available ML models, diverse algorithm mapping approaches, and universal support on different target devices. In this chapter, I build upon the early concepts in [218] and propose three mapping methodologies for in-network ML algorithms (§3.2). Each methodology can be applied to a group of ML algorithms with specific characteristics. Building upon these methodologies, I support and realise the mapping of a wide range of ML algorithms (§3.3-3.5), with multiple mappings realised for some of the models. Most of these mappings are designed to support multiple software and hardware programmable network devices, particularly resource-constrained commodity targets. The mapping methodologies introduced in this chapter enrich the available models, alleviating the scarcity of in-network ML algorithms, which were published in [245, 247, 243, 241, 239].

3.1 Objective

This research aims to narrow the gap of in-network ML mapping and sets the following design goals:

1. General mapping methodology. Fundamental characteristics of ML inference algorithms should be identified and their data plane mapping should be simplified. This can be used as a guidance for in-network ML realisation and drive the implementation of new models and their variations.

2. Optimised models. ML inference models mapped to programmable data planes should provide high ML and system performance, with minimum resource overheads. As programmable network devices are primarily designed for packet processing, they have limited resources, and support a constrained set of mathematical operations. Mapped ML algorithms need to trade off model size and performance to fit on a network device. Thus, this work should support a wide range of predefined mapped ML models, with optimised resource efficiency, that can co-exist with mandatory network functionality.

3. Theoretical analysis. ML models mapped by the same mapping methodology have similar characteristics in stage and memory (M/A table entry) consumption. A theoretical analysis can help us better understand each proposed mapping and the benefits of its selection and adoption. Due to the dependency on implementation, practical evaluations are discussed in the following Chapter 4.

3.2 Generalising Mapping Methodologies

Mapping ML inference models to the data plane can be challenging, therefore many previous works [119, 33, 66, 160, 186, 184] have focused on a single model. While a single-model approach has benefits, it also limits the agility and adaptability to use cases, creating a barrier to adoption of new algorithmic solutions. This research tackles this challenge by proposing three general ML model map-

ping methods, based on similarities in models' structures.

A model's mapping needs to attend to the constraints of programmable network devices, such as pipeline architectures, resources (e.g., stages and memory), and limited mathematical operations. To enable the mapping of multiple different models under these constraints, I propose three general mapping methodologies: direct-mapping (DM), encode-based (EB), and lookup-based (LB). Some models have a clear sequential and simple inference process, where a direct-mapping solution can be used (§3.3). However, for algorithms that have relatively complex inference processes, direct-mapping solutions may result in high resource consumption and can not be applied. At this time, if the algorithm inference process can be treated as the splitting of the feature space, encode-based solutions can be applied (§3.4). Similarly, if the inference model has a form of polynomials, we can utilise a lookup-based solution to support them (§3.5). The following section describes the three mapping methodologies with leading examples per method in detail.

3.3 Direct-Mapping Methodology

Researchers have identified early on, that some ML algorithms have a structure relatively similar to a data plane [218], simplifying their mapping. Their inference process can be converted to a sequence of match actions using the direct-mapping approach. However, to realise them on programmable network devices, some adaptations are still required, including the utilisation of alternative operations or approximations to replace complex operations that lack support.

3.3.1 Decision Tree (DT)

Decision tree is one of the classical supervised learning algorithms, which can solve both classification and regression problems [141]. A decision tree model has a tree-like structure that includes a root node, several internal nodes, and several leaf nodes [166]. 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 decision tree, 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 calculation logic of a simple switch pipeline is similar to a tree structure, which makes decision tree a good application prospect [218]. However, the iterative node-based comparisons performed by decision tree during inference pose a primary challenge in realising the algorithm on programmable network devices [218].

Direct-mapping decision tree (DT_{DM}) [119, 33, 213], as shown in Figure 2.4 (a) (or Tree 2 in Figure 3.1), hierarchically maps the tree model into the M/A pipeline, using N_{depth} tables. The model is executed within the pipeline layer by layer (depth by depth), until reaching the leaf node. The model starts by extracting the required features from incoming data. In each layer of the tree, 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 as there is a dependency between the tree's depth and the number of stages. Another 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 [213] and does not save stages. Both these two methods mainly fit targets that are not sensitive to stages.

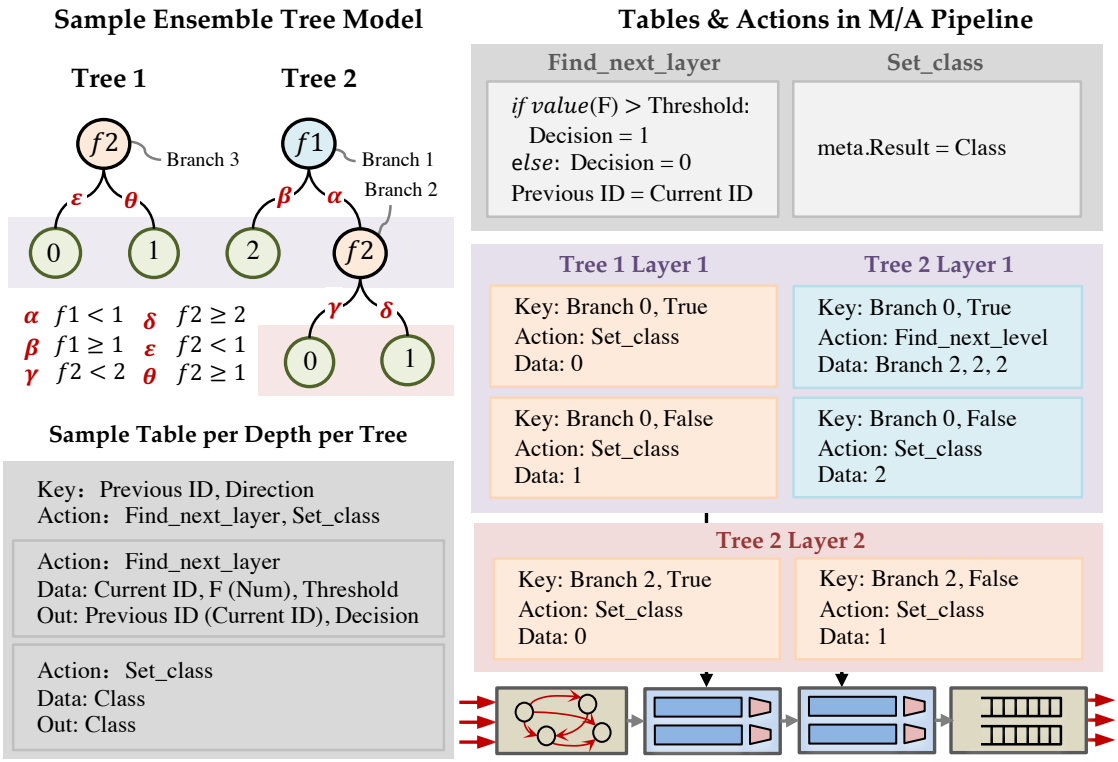


Figure 3.1: Ensemble trees workflow using direct-mapping methodology.

3.3.2 Random Forest (RF)

Random forest is an ensemble model built from a set of decision tree models [85]. The direct-mapping random forest (RF_{DM}) uses a similar mapping process as decision tree [33, 119], with an extra decision table used to conclude the labelling based on all results from each base tree model. As shown in Figure 3.1, to map a p-depth (layer) model, a single DT_{DM} uses p tables. Each table uses the result from the previous layer as the key. The workflow checks the current branch ID, its threshold, and the used feature. After the lookup, a comparison is done between the matched value and a threshold. The comparison's result and the current branch ID are used as the keys in deeper layers. Direct-mapping ensemble

trees consume relatively low memory, but the logic operations are complex and lookups need to be executed sequentially due to the strong dependency between parent and child nodes, which is stage-consuming and latency-consuming. This research optimises the implementation in [119] by enabling parallel tree placement, allowing each pipeline stage to concurrently process branches at a certain depth across all trees.

3.3.3 Neural Network (NN)

Neural network stacks simple classifiers that operate in parallel to model the complex relationship between input and output from historical data. These simple classifiers, such as perceptrons, are shown in Figure 3.2. Due to its layer-based structure, this research supports the direct-mapped binary neural network, as proposed in [162, 184]. With binary inputs and weights, the three key steps in the forwarding path of a perceptron in a neuron network are shown in Figure 3.2, 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 [183, 169, 186, 185, 160]. M/A is another alternative to realise these three steps in perception with the trade-off between memory (M/A) and stages (Popcount & XNOR) but usually requires a large amount of memory and many stages. Since in current programmable network devices, the M/A-based method does not bring benefits in terms of saving stages and memory, this subsection focuses on the Popcount (Hamming weight) and XNOR-based method.

BNN is a stack of binarised perceptions, its workflow within the data plane is shown in Figure 3.3. This example requires three input features, employs three

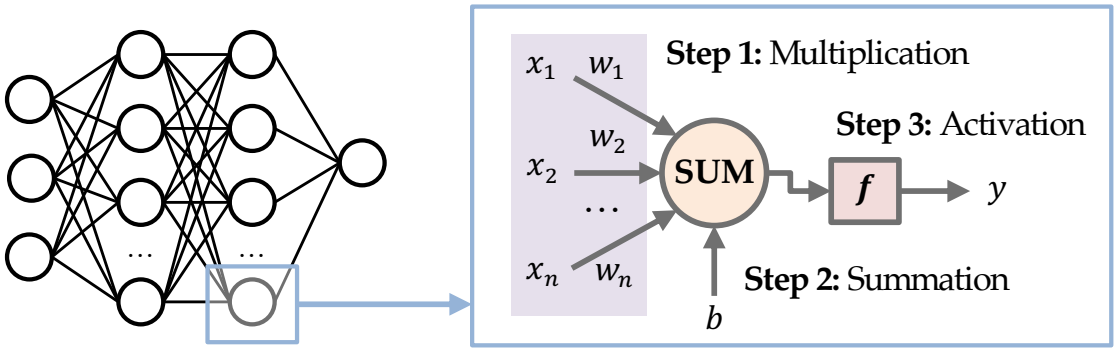


Figure 3.2: A fully connected neural network constructed by many perceptrons.

neurons in the middle layer, and has one neuron in the output layer. The workflow starts by extracting selected features from headers of incoming packets or local memory, such as meters, counters, and registers. 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 3.3) 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. The workflow iterates previous steps for each layer until the last layer. There is a notation change as shown in Figure 3.3. Different from Figure 3.2, the weights in each layer are equal to the number of neurons in the next layer. A simple example of a basic element of the workflow is shown in the bottom right corner of Figure 3.3.

The applied model and backpropagation method should ensure that the network weights are close to the range [-1, 1], which helps reduce the loss of information when applying the binarisation technique. Benefiting from register-stored weights, binary neural networks can apply the online update without

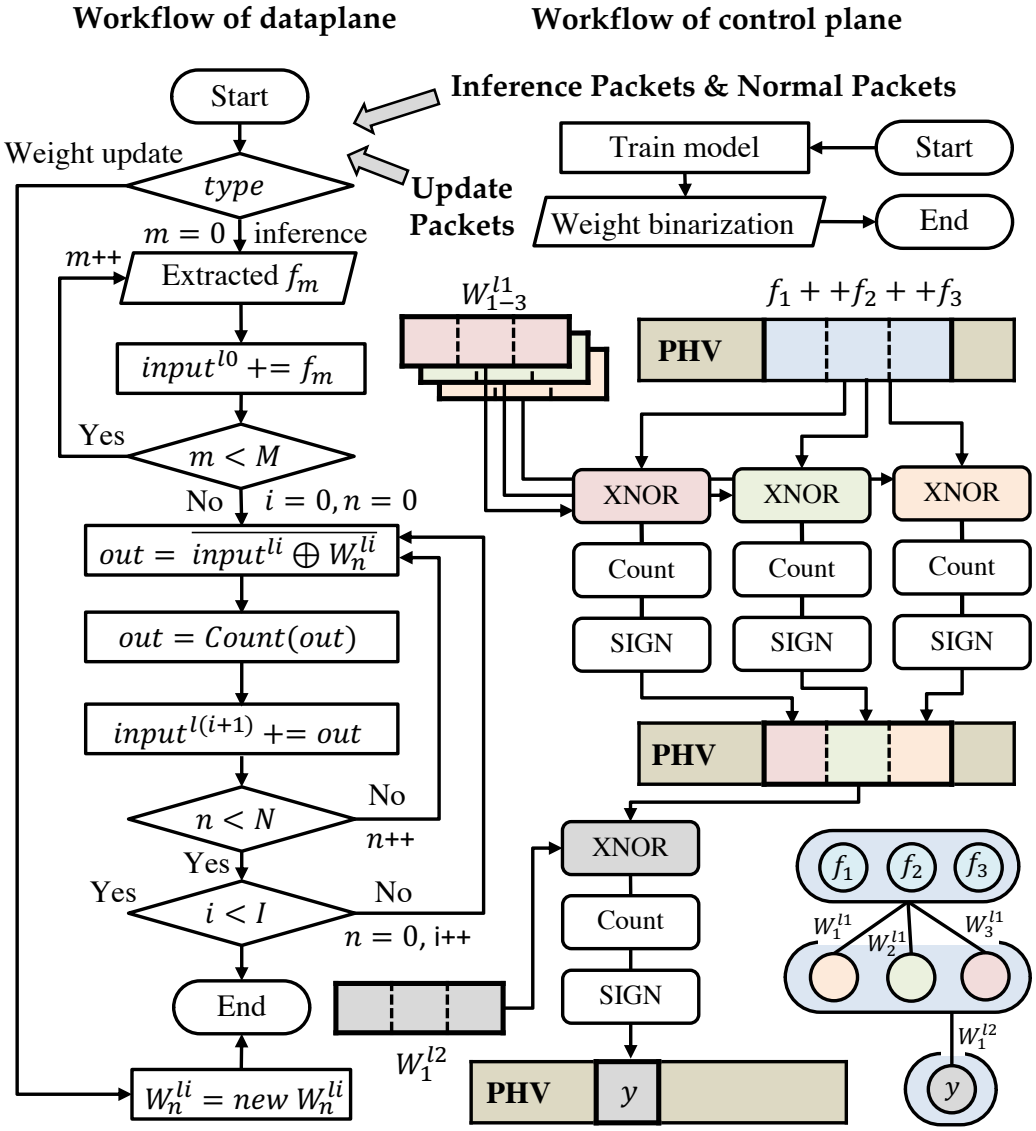


Figure 3.3: Workflow of the binary neural network implementation within a programmable network device.

stopping the device [160]. The trained binary weights will be packed and transferred to target data plane devices and trigger the weight update workflow.

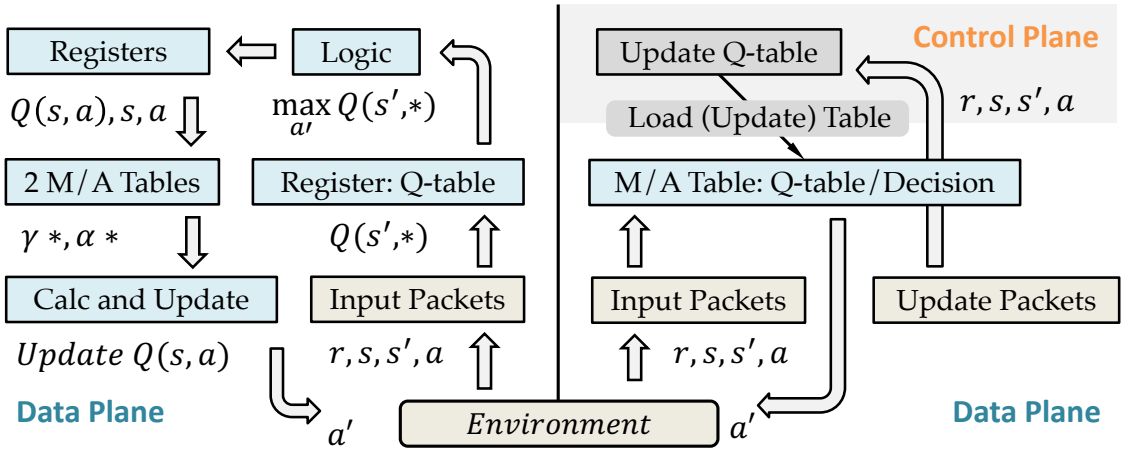
3.3.4 Q-Learning

Q-learning [209] is a popular reinforcement learning algorithm. Unlike other reinforcement learning algorithms that are deep neural-network and hard to implement in a data plane [245], Q-Learning is suitable for direct-mapping methodology. As shown in Algorithm 1, the algorithm tries to solve the Markov decision process by learning an optimal policy (Q-table) through value iteration. It achieves this by iteratively updating the Q-value of state-action pairs (s-a and s'-a' are the current and next state-action pairs). To this end, it uses observed rewards (r) and the maximum expected future rewards (Algorithm 1 line 6, where α is the learning rate and γ is a discount factor). The algorithm explores the environment using an ϵ-greedy algorithm to balance exploration and exploitation, gradually converging towards an optimal policy that maximises the expected cumulative reward over time. During the exploration process, each time Q-table update means a step and each time environment restart means an episode. The look-up mechanism used by Q-learning's Q-table fits the M/A table in the data plane well, and the limited action space inside the Q-table reduces memory requirements.

Algorithm 1: Q-learning

Initialize: The Q-table $Q(s,a)$ arbitrarily

1 **Repeat** // for each episode
2 Initialize s;
3 **Repeat** // for each step of episode
4 $a \leftarrow Q(,)$ and s using policy e.g., ϵ-greedy;
5 Take action $a \rightarrow$ observe r and s';
6 $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$;
7 $s \leftarrow s'$;
8 **while** *step is not terminal*;
9 **while** *episode is not terminal*;

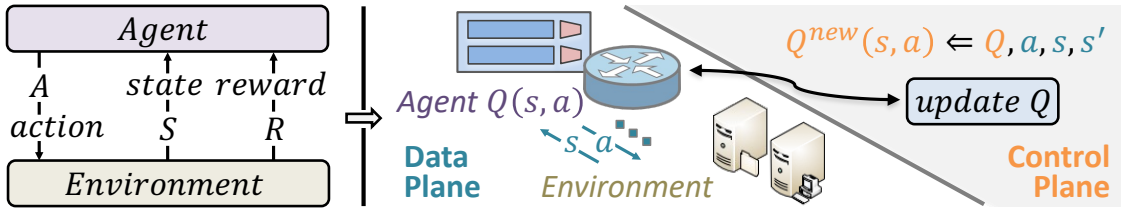


(a) Q-learning fully on Data Plane (b) Q-learning partially on Data Plane

Figure 3.4: In-network Q-learning solutions. (a) Register-based Q-learning and (b) M/A table-based Q-learning.

The main challenge to implementing in-network Q-learning is updating the Q-table. This includes where to store the Q-table, how to calculate Q-value, and how to maintain history action, state, and reward information. To address these challenges, I introduce two new solutions: one purely in the data plane, and one combining data and control planes.

The first solution implements Q-learning entirely in the data plane using registers, as shown in Figure 3.4 (a). In this solution, the Q-table and some parameters (e.g., previous state, action, reward) are stored in register arrays. The workflow of this register-based Q-learning is as follows. When a new packet arrives (Input Packets in Figure 3.4 (a)), it is associated with environment's information including new state (s') and reward (r) (past state s and action a are optional), reflected in line 5 of Algorithm 1. The rewards $Q(s', *)$ are read from the Q-table stored in registers (shown in Figure 3.4 (a) step "Register: Q-table"). The new action (a') is calculated using logic based on an ϵ-greedy algorithm. The previous Q-value ($Q(s,a)$), state (s), and action (a) are read from registers and the registers are updated with the new values ($Q(s',a')$, s', a') (line 7 in Algorithm 1). The reg-



(a) General RL (b) In-network RL

Figure 3.5: Comparison between traditional Q-learning and M/A-based in-network Q-learning in Figure 3.4 (b).

isters in this step can be changed to packet headers if the input packet contains these values. The Q-value update is done in the step "Calc and Update" (line 6 in Algorithm 1). For the update calculation, the multiplication results between all intermediate values and the learning rate or discount factor are pre-computed, quantised, and stored in two M/A tables, removing the need for multiplication which is unsupported within the data plane. After the new Q-value is computed and the Q-table is updated, the packet (or another object or function) will return the new action (a') to the environment (line 4 in Algorithm 1). This procedure will iterate, with the agent taking the new action and getting the reward from the environment.

The second solution moves complex operations from the data plane to the control plane, as shown in Figure 3.4 (b) and Figure 3.5. The workflow of this M/A table-based Q-learning is as follows. Like the first solution, an input packet is associated with the previous state & action (s & a) and the new (current) state & reward (s' & r). The reward of all actions is read from the Q-table, implemented as a M/A table, and the new action (a') is selected based on an ϵ-greedy algorithm. The new action (a') is sent back to the environment to guide the agent's behaviour. Different from the registers used in the first solution, M/A tables can not be updated in-band. Instead, the control plane (e.g., the switch CPU) needs to update the entries. The current reward (r) and state (s') are brought to the con-

trol plane using update packets. The update packets can be generated from the switch CPU, sent by the environment, or generated (mirrored) by the switch. As the update process happens in the control plane, there is no need to calculate the new Q-value in the data plane. The new Q-value can be calculated in the control plane before the table update. After the Q-table has been changed, the control plane updates the changed table entry in the data plane.

The two Q-learning solutions presented in this section introduce trade-offs: the register-based solution will react faster to changes (sub-μs vs tens of μs) and can process updates at line rate. The M/A-table approach, on the other hand, can handle a larger state space, requires fewer resources, and can support more complex reinforcement learning implementations. Further aspects of implementation are discussed in Section 7.4.

3.3.5 Potential Extensions

This thesis currently supports four direct-mapping models, which apply alternative operations or approximations to overcome constraints on programmable network devices. Additionally, other ML models with a format close to the M/A pipeline are also candidates for direct-mapped mappings, such as DNN [146], CNN [26] and LSTM [86].

3.4 Encode-Based Methodology

Many classification algorithms aim to find borders in either the original feature space or the mapped feature space. The area confined by a set of borders (par-

titions) is labelled as a class. Algorithms use different methods to define their borders. Some use complex functions, while others use linear functions for approximation. The proposed encode-based mapping typically uses linear borders to slice the feature space with a code to represent a certain area within the space.

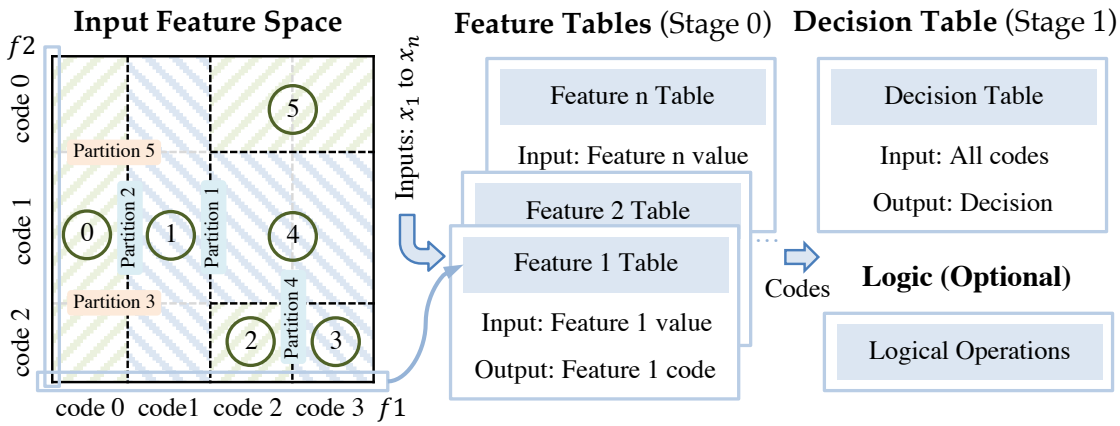


Figure 3.6: Mapping methodology of the encode-based solutions.

In a general encode-based model, the mapping to the data plane starts with slicing input feature space into classes, using feature tables and a decision table. As shown in Figure 3.6, feature space (e.g., two-dimensional space) is sliced into 6 areas (i.e., area ⓪ to area ⑤) by 5 partitions (i.e., partition Partition 1 to partition Partition 5). Mapping this ML model to a M/A pipeline requires two feature tables, recording the mapping from feature values to codes, where code pairs represent an area (e.g., area ③ coded as $f1$-code 3 & $f2$-code 2). The mapping from codes to labels is stored in a decision table. In an n dimensional feature space, the model similarly needs n feature tables and 1 decision table. Encode-based models vary depending on how the feature space is split. The realisation details of supported models are provided below.

3.4.1 Decision Tree (DT)

Decision tree uses a top-down decision process, splitting the feature space at each branch (node) until reaching the leaf nodes [212]. Figure 3.7 shows a sample decision tree model and a two-dimensional input feature space split by its branches. The similarity between Figures 3.6 and 3.7 indicates that the general encode-based solution fits the decision tree model. For an n features input space, an encode-based decision tree (DT_{EB}) requires n feature tables, encoding each feature value. The encoded feature space is mapped using a decision table to labels. All feature tables share a single pipeline stage (within target limitations) and the entire mapping requires only two logical stages.

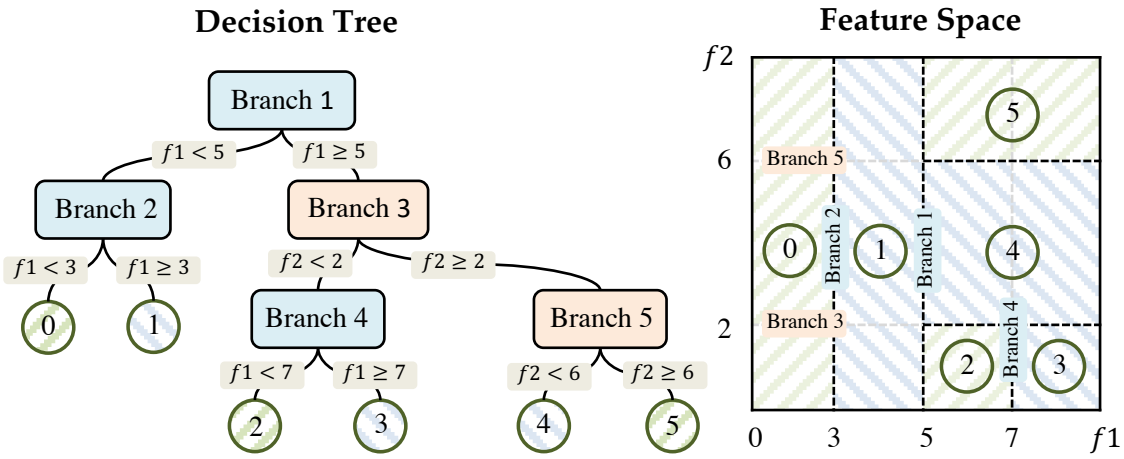


Figure 3.7: Feature space split in tree-based models.

To realise DT_{EB} mapping, I use four steps, as shown in Figure 3.8 for Tree 2. The input to the process is a trained decision tree model. In the step titled "Find feature splits", the algorithm collects all the branches related to each feature. The feature values are encoded (mapping an area to a code word) and saved as a feature table in the step "Generate feature table". The encoding is determined by the splitting conditions of the branches. The algorithm associates each area in the feature space with a leaf node, and determines the range of values it in-

cludes. Finally, "Generate the tree table" (also named code or decision table) links mapping from leaf nodes to codes pointing to their areas.

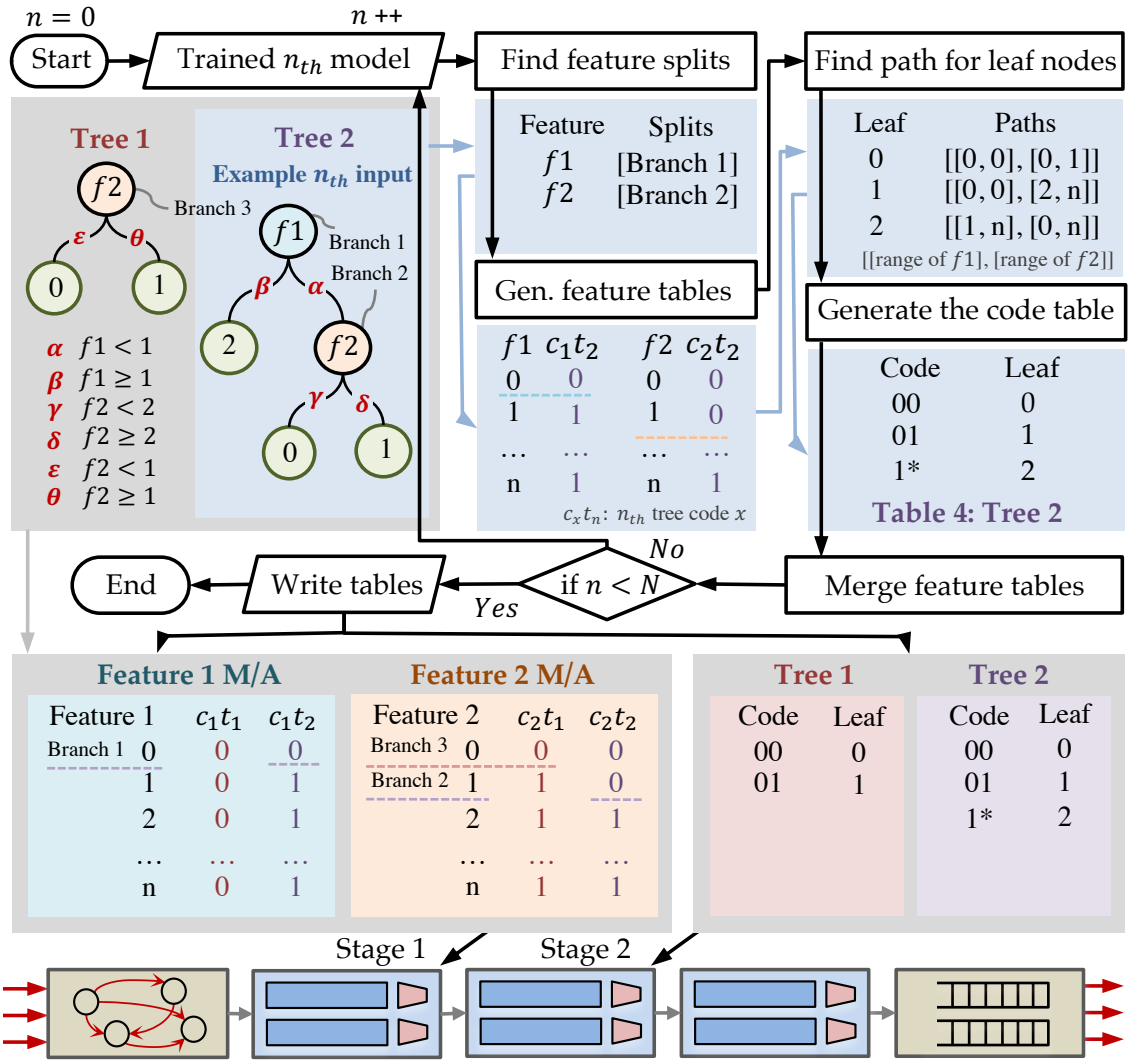


Figure 3.8: Ensemble trees (Tree 1&2) and Decision Tree (Tree 2) models' mapping workflow using the encode-based methodology. The upper part of the flowchart mainly illustrates the M/A tables generation process of Tree 2.

To further improve its performance, in this research, DT_{EB} uses default actions in tree tables (M/A table in the pipeline) to store the most common label, along with using the ternary match, longest prefix match (LPM), or range match in all tables. These reduce the number of table entries, saving significant memory resources (§4.4.2).

3.4.2 Random Forest (RF)

Random forest is one of the popular models generated by the bagging technique [120], 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. In random forest, different trees are trained on different data and every tree model has the same importance. By applying the encode-based methodology to random forest, I introduce a new RF_{EB} algorithm mapping, which allocates trees in parallel and decides the label (classification decision) using a voting table, as shown in Figure 3.9. In the voting table, RF_{EB} model makes the decision based on DT_{EB} votes. Figure 3.8 shows our random forest workflow and a toy example of mapping a two-tree random forest model to M/A format. For a random forest model with n input features and m trees, the mapped model uses n feature tables and m tree tables. In each feature table, the codes $(c_i t_1, c_i t_2, \ldots, c_i t_m | i \in n)$ for all trees are stored as actions. As noted in Section 3.4.1, n feature tables can share a stage. Similarly, m decision tables, one per tree, can share a stage and be looked up in parallel. The voting table requires a third logical stage. Compared with if/else logic [213] and depth-based solution [119], the proposed RF_{EB} is scalable and saves stages (shown in §4.4.2).

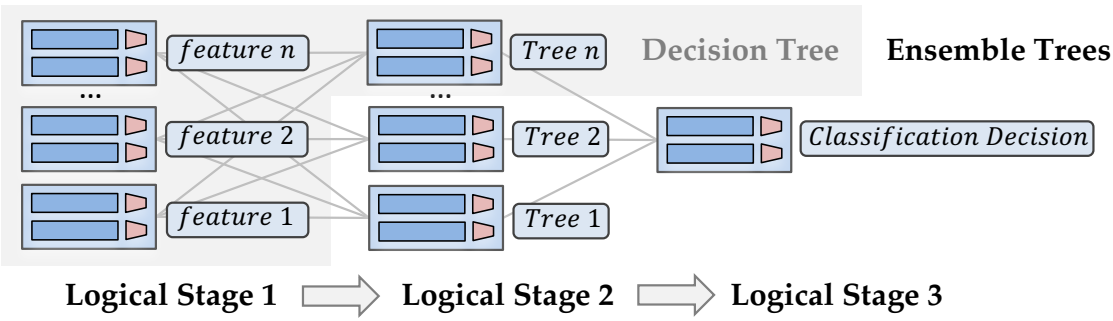


Figure 3.9: Differences between decision tree and ensemble trees in M/A table usage. Figure 3.8 shows table generation workflow.

3.4.3 XGBoost (XGB)

XGBoost is a different type of ensemble tree model. Based on boosting, in XGBoost, a serial learning sampling technique samples data under the distribution based on the learning results of the last iteration [172]. Each subsequent tree is trained to estimate the errors of the previous trees. A primary difference between XGBoost and random forest is that XGBoost accumulates probabilities from each tree's leaf nodes to make the final decision [38]. However, calculating probabilities within an M/A pipeline is non-trivial or costly in resources. Instead, I introduce a new XGBoost (XGB_{EB}) solution, encoding the probabilities in each tree and storing them in M/A tables (tree table in Figure 3.8). Specifically, each code (c_*t_*) in the feature table represents a unique probability or a small range of probabilities. To create the decision (codes-to-label) table, the XGBoost's mapping workflow calculates the cumulative probabilities and expected output label in advance for each code combination. XGBoost's probabilities addition and comparison operations are therefore replaced by simple codes-to-label look-ups in the final decision process. Multiple discrete probabilities are mapped to the same code if they lead to the same label mapping, thus saving resources. XGB_{EB} uses the same amount of logical stages as RF_{EB}.

3.4.4 Isolation Forest (IF)

Isolation forest (IF) is an unsupervised ensemble model based on random forest [126] and is mainly used for outlier detection. In an isolation forest, to make the decision, the total number of branches used in the forest decision is compared to an anomaly threshold, and the fewer branches (shorter path length) used, the more likely the sample is an outlier. Equation 3.1 shows this process, where x

is one input, $h(x)$ is the path length, t is the total number of training inputs, γ is Euler's constant, and $E(h(x))$ is the average $h(x)$ of a collection of trees.

$$E(h(x)) \leq -(2(ln(t-1)+\gamma) - 2(t-1)/t)log_2 0.5 \tag{3.1}$$

In the data plane, recording the path length is difficult. Consequently, the path length for all leaf nodes in the forest is precomputed. To realise Equation 3.1 in the M/A pipeline and store these values, our newly proposed isolation forest (IF_{EB}) adopts a method similar to XGB_{EB}. Specifically, the number of passed branches $h(x)$ can be first encoded with user-defined granularity, which can balance resource consumption and accuracy. These encoded values are stored in tree (code) tables (as shown in Figure 3.8 Tree Table 1 & 2). The mapping between the number of passed branches per tree and the final label is calculated in advance and stored as a M/A table named code-to-label. This substitution of threshold operations with lookup tables saves stages in IF_{EB}.

3.4.5 *k*-Nearest Neighbor (KNN)

The *k*-nearest neighbors (KNN) first finds *k* closest training samples to the testing input in the feature space. Then, in a classification task, the algorithm categorises the testing samples into the label with the largest number of closest training samples according to the majority voting rule. This process can be treated as splitting the feature space by distance according to its *k*-nearest neighbours. A tree data structure provides a feature space slicing approximation and labelling [167, 66]. This research newly maps *k*-nearest neighbor to the data plane using the encode-based methodology and tree data structure. Figure 3.10 shows an example of applying Quadtree to do feature space slicing. Specifically, in a higher dimensional feature space, at each layer of the tree, the input n dimensional feature space is

divided and labelled into 2^n equal parts in the same order. k-nearest neighbor requires a code of $d \times n$ bits to represent each area when the maximum depth is d. The feature space is split continuously until the tree reaches the maximum depth or all vertices of the current unit belong to the same class. This tree-like splitting approach enabled storing all codes in a ternary table (Figure 3.10) or several feature tables and a decision table (Figure 3.6), thereby no need to store any testing data and reducing memory consumption. This method can be also applied in the following k-means (§3.4.6).

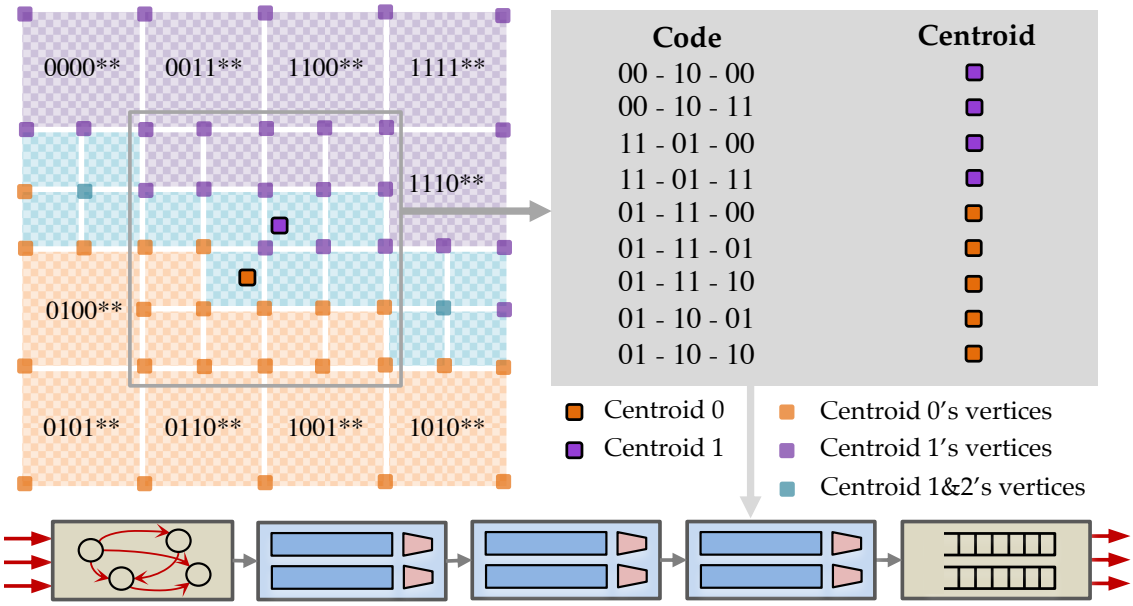


Figure 3.10: KNN and KM workflow using encode-based solutions.

3.4.6 k-means (KM)

Encode-based k-means (KM_{EB}) labels the input based on the distance between the data point and each centroid [62]. k-means is a typical classification method based on feature space slicing. This research implements KM_{EB} using n-dimensional tree splitting [167], building upon Clastream [66], a solution predating k-nearest neighbor. KM_{EB} labels the input based on the distance between the

data point and each centroid [62]. The input feature space thus can be divided into small pieces. Figure 3.10 shows a KM_{EB} toy example, a variation on Clustreams's [66] solution, which encodes the feature space (2 dimensional) by using a Quadtree [167]. For this two-dimensional input toy example in the figure, this method uses fine-grained squares to describe the boundaries and uses coarse-grained squares to represent spaces inside. The level of detail in its boundary depiction 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 3.10, uses consecutive codes to index (encode) each input feature after assigning each block to a class (Figure 3.6). 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 method uses the Quadtree index only, as shown in Clustreams' work [66]. In this method, the feature space is easier to convert to the TCAM table but requires pre-processing before the packet is sent to programmable network devices.

3.4.7 Potential Extensions

This thesis currently supports six encode-based models. This methodology demonstrates high generality, making it applicable to the majority of ML models. However, among all ML models, those employing feature space partitioning with a limited number of linear slicing, are especially well-suited. In addition to the models previously discussed, CatBoost [159], LightGBM [106], and AdaBoost [63] are examples of additional models that can potentially be implemented using encode-based methodology.

3.5 Lookup-Based Methodology

Many ML algorithms require mathematical operations on input features to decide a label, commonly too complex to implement in a hardware data plane. The lookup-based methodology uses M/A tables to store intermediate results of these operations and thus enable realising in-network ML. As shown in Figure 3.11, any ML algorithms with a *Decision Process* (a set of polynomials) can use lookup-based solutions. In lookup-based solutions, feature tables store the mapping between each input feature value and intermediate results. These intermediate values are then used for the remaining operations, typically addition and comparison, as a final logic stage. There are alternative design choices (variations) for table design to store intermediate results, mainly differentiated by the granularity of the intermediate results. For instance, one approach involves utilising all features $x_1, x_2, \ldots, x_n$ as the input and directly output the summation of all intermediate results $IR_i + IR_2 + \ldots + IR_n$ for equations 1 to k. However, the variation introduced in this section strikes the optimal balance between memory usage and processing stages, aligning most effectively with the resource characteristics of programmable network devices. The methodology in this section is inspired by the utilisation of lookup tables to implement ML algorithms on

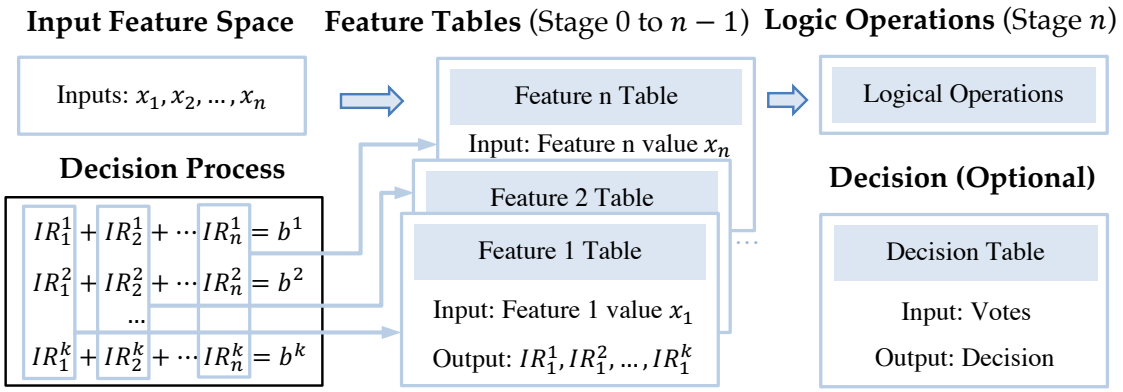


Figure 3.11: Mapping methodology of lookup-based solutions.

FPGAs in IIsy [218].

3.5.1 Naïve Bayes (NB)

As a statistical classification method, naïve Bayes is the general term for algorithms based on Bayes' theorem [22] shown in Equation 3.2, which is one of the simple classical Bayes models.

$$P(x_i|y) = \frac{P(y|x_i)P(x_i)}{P(y)} \tag{3.2}$$

Based on the posterior probability introduced by Bayes' theorem, the algorithm tests the probability of each set of inputs with each label, under the assumption that each input feature is independent. As shown in Equation 3.3, where x is the input and y is the label, the label with the highest probability is the prediction result.

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i \mid y) \tag{3.3}$$

One lookup-based naïve Bayes mapping solution, proposed in IIsy [218], used features as the direct input to M/A tables, outputting the respective posterior probabilities of all their classes. This method only works when the range of input values is narrow. Otherwise, the table will be excessively large, as all intermediate results $P(x_i \mid y)$ are multiplied. In this research, I introduce an enhanced lookup-based naïve Bayes mapping, which uses logarithms to convert multiplication into addition, as shown in Equation 3.4.

$$\hat{y} = \arg\max_{y} [map(log_2 P(y)) + \sum_{i=1}^{n} map(log_2 P(x_i \mid y))] \tag{3.4}$$

This mapping fits the standard lookup-based methodology, shown in Figure 3.11, using n feature tables (e.g., Input: x_i, Output $map(log_2 P(x_i \mid y_1)), map(log_2 P(x_i \mid y_2)), \ldots, map(log_2 P(x_i \mid y_k))$) for any k classes inference task.

The logarithm operation both reduces operation complexity and lowers memory consumption. As in all models realised using look-up methodology, these mapped intermediate results are quantised and then stored in M/A tables. In the final logic, all intermediate results for each class are summed up and compared (replacing arg max) to get the output label. Based on this method, this work also supports variations (such as GaussianNB and BernoulliNB [154]), catering to diverse types of input variables.

3.5.2 Autoencoder (AE)

Autoencoder (AE) has an unsupervised neural network architecture, typically used for representation learning through data encoding and decoding [145]. 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. Due to the resource constraints on programmable network devices, this research only focuses on the one-layer autoencoder used for dimensional reduction. Autoencoder's workflow is composed of an encoder and a decoder [125].

$$X_{new} = XW + Bias \tag{3.5}$$

The forward path of the single-layer encoder network is interpreted as Equation 3.5, where X is the high dimensional original input and X_{new} are the outputs with lower dimensions. The encoder's equation has a similar format as polynomials depicted in Decision Process (black box in Figure 3.11), enabling this research to introduce a new mapping for autoencoder, based on the lookup-based

methodology. Specifically, when X_{new} has k dimensions, W is a $n \times k$ weight matrix. The mapping uses n feature tables to store the intermediate results of all output feature dimensions $(x_i w_1^i, x_i w_2^i, \ldots, x_i w_k^i)$ under the corresponding feature i. The final logic sums all intermediate results in each output dimension and the *Bias* to get the value of new features. This mapping avoids multiplication operations in autoencoder and can achieve a similar computational overhead as naïve Bayes.

3.5.3 Principal Component Analysis (PCA)

Principal component analysis (PCA) is a statistical method that is commonly utilised to reduce the dimensionality of large datasets while preserving most of the data variation [211]. PCA finds a new axis with a predefined dimension that can best represent the feature space [67] and it computes the cumulative projection of each component on each data point onto new components to conduct dimension reduction. As shown in Equation 3.6, the forward path of a trained PCA has two main steps: shift and map.

$$X_{new} = (X - X_{means})Components \tag{3.6}$$

In this equation, the input X is the array $[x_1, x_2, ..., x_n]$ with n input features, and $X_{means} = [x^1_{means}, x^2_{means}, ..., x^n_{means}]$ is the mean value of each feature. $Components$ is a transformation matrix with n rows and m columns. The output X_{new} is the array $[x^1_{new}, x^2_{new}, ..., x^m_{new}]$ with m output features. This matrix multiplication process can be represented as polynomials, fitting the mapping of lookup-based methodology. Based on this, the newly proposed lookup-based PCA requires n feature tables. In each feature table, all intermediate results related to that feature are stored. For example, all results related to feature i, including $IR_i^1 = (x_i - x^i_{means})w_i^1$, $IR_i^2 = (x_i - x^i_{means})w_i^2$, $\ldots$, $IR_i^m = (x_i - x^i_{means})w_i^m$, are stored in the action data field

of feature table i. The final logic of PCA is similar to the autoencoder, which is the summation of intermediate results of each new dimension and the construct of new feature values. Due to a similar structure in the M/A pipeline, PCA can achieve comparable memory and stage consumption as naïve Bayes.

3.5.4 Support Vector Machine (SVM)

Support vector machine (SVM) is a popular classification algorithm, which performs well in solving nonlinear dataset problems and high dimensional pattern recognition problems [49] with a small sample size. The SVM model projects data into hyperspace and it aims to find hyperplanes that 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) [35], etc [153].

$$\begin{cases} w_1^1 x_1 + w_1^2 x_2 + \ldots + w_1^n x_n + d_1 = 0 \\ w_2^1 x_1 + w_2^2 x_2 + \ldots + w_2^n x_n + d_2 = 0 \\ \qquad\qquad \ldots \\ w_m^1 x_1 + w_m^2 x_2 + \ldots + w_m^n x_n + d_m = 0 \end{cases} \tag{3.7}$$

For instance, Equation 3.7 shows how an SVM model is constructed by the linear kernel for a k classification task with n dimensional input data $X = \{x_1, x_2, \ldots, x_n\}$. Each line of the equation represents a hyperplane as the border of two sub-classes. The model will generate m hyperplanes where $m = k(k-1)/2$. The output label can be determined by counting the votes from all hyperplanes and using logic or a decision table [49]. This process is similar to the Decision Process in Fig-

ure 3.11. Lookup-based SVM is supported in this research based on the proposal in [243]. To achieve scalability, n feature tables will be used to store the intermediate results from all hyperplanes (e.g., Input: x_i, Output $w_1^i x_i$, $w_2^i x_i$, $\ldots w_m^i x_i$). All feature tables belong to a single logical stage and use the addition operation for hyperplanes (initialised by bias b_i). This method saves memory and stages compared with other approaches [218].

3.5.5 *k*-means (KM)

The k-means workflow labels inputs according to their distance to the trained k centroid [62], as described in Equation 3.8. A lookup-based k-means (KM_{LB}) mapping, as in IIsy [218, 243] and Figure 3.11, uses n feature tables to store intermediate results in parallel.

$$D_i = \sqrt{(x_1 - c_1^i)^2 + (x_2 - c_2^i)^2 + \ldots (x_n - c_n^i)^2} \tag{3.8}$$

Equation 3.8 shows how to calculate the distance between each set of input to each cluster, where $X = \{x_1, x_2, \ldots, x_n\}$ is the input and $C^i = \{c_1^i, c_2^i, \ldots, c_n^i\}$ is the centroids of cluster i. The complex mathematical operations, including square and square root, make it challenging to fit on programmable network devices. Due to the monotonically increasing nature (when the value under the square root is positive) and low curvature, the square root operation in Equation 3.8 can be removed from the data plane implementation with minor accuracy influence. To replace the square operation, KM_{LB} solution uses $map(.)$ operation and constructs feature tables with input x_i, output $map(x_i - c_i^1), map(x_i - c_i^2), \ldots, map(x_i - c_i^k)$. The $map(.)$ function maps all input values to a domain $\{1 : 2^{n_{bits}}/n\}$, where n_{bits} is the width of each action data. With this $map(.)$ operation, the distance values are mapped to a proper range, lowering the accuracy loss.

3.5.6 Potential Extensions

This research currently supports five lookup-based models. Additional ML algorithms that can be converted to a set of polynomials, such as Lasso [200, 65], linear regression [144], and polynomial regression [189], can potentially be implemented using lookup-based mapping, with a similar methodology.

3.6 Analysis and Discussion of Table Size and Stage

Assessing the resource consumption of algorithm mappings mentioned in the preceding sections is important, as it significantly impacts the scalability and performance of these models on programmable network devices. While this research has only manually validated the correctness of these mappings in either emulation or software environments at the stage of this chapter, resource consumption of these in-network ML algorithms is briefly analysed on a theoretical basis, categorised by mapping methodology. The detailed experimental results are presented in the subsequent Chapter 4, whose results align with this analysis.

Direct-Mapping Models. As the model realised by direct-mapping methodology lacks a standardised template and the algorithm mapping using this methodology is realised mainly case-by-case, this analysis is primarily oriented towards some common characteristics. In terms of table entry overhead, the direct-mapped algorithms are usually efficient. These algorithms usually do not employ mappings to replace complex logic and operations (while if lookups are used, the memory requirements may increase significantly). Instead, these algorithms mainly apply alternative operations that are supported by programmable network devices to substitute (e.g., replacing matrix multiplication with Pop-

count & XNOR). However, implementing these alternative operations usually consumes pipeline stages. Together with the stages used to realise the full-size unmodified (or slightly modified) inference workflow, these direct-mapped algorithms commonly require a large number of pipeline stages.

Encode-Based Models. Encode-based algorithms follow a standard structure, utilising a sequence of feature tables and a decision table. In this context, consider a use case with a feature space containing n inputs ranging from 1 to f_{max}. Following model training, each feature i is divided into k_i partitions within the sliced feature space. After encode-based mapping to the data plane, each feature table contains f^i_{max} entries. The decision table has a size of $\prod_{i=0}^{n} k_i$. As there is no interdependence among feature tables, they can be accommodated in a single logical pipeline stage. The overall resource consumption for this algorithm is $\sum_{i=0}^{n} f^i_{max} + \prod_{i=0}^{n} k_i$ table entries and 2 pipeline stages. If an LPM match is applied to all feature tables, this value can be reduced to $\sum_{i=0}^{n} e f^i_{max} + \prod_{i=0}^{n} k_i$, where e denotes the efficiency factor of LPM mapping. This can significantly reduce table entry consumption, particularly when the feature range k is large. For the encode-based methodology, while the model can effectively control the usage of pipeline stages, the table entry consumption may rise significantly when the slicing of the feature space is overly complex, primarily due to the $\prod_{i=0}^{n} k_i$ term.

Lookup-Based Models. For algorithms supported by lookup-based mapping methodology, they have a similar structure consisting of a sequence of feature tables, followed by logical operations. Consider a use case with n input features, having a range of $(0, f^i_{max}]$ for a given feature i. Similar to the encode-based methodology, the lookup-based algorithm requires $\sum_{i=0}^{n} f^i_{max}$ table entries (just remove the table entry in the decision table). However, since the number of different intermediate result values is usually large, the LPM mapping is not that

effective in this case. In terms of stage consumption, due to the interdependency of intermediate results from each feature table, the algorithm requires $1 + l$ logical stages, where l is the number of stages used in final logic (typically requires $\lceil log_2 n \rceil$ to $n + 1$ stages for a use case with two classes). For models implemented using lookup-based methodology, the required table size is relatively controllable and proportional to the number of features. The consumption of stages heavily depends on the number of features and classes from the use case.

Discussion. This research finds that for direct-mapping in-network ML algorithms, memory consumption is generally low, but the complexity of computations and alternative operations demands high requirements for stage usage and special operations. Regarding encode-based in-network ML algorithms, they benefit from parallelism and manage stage consumption effectively, yet excessive model sizes can still lead to significant memory consumption. In the case of lookup-based in-network ML algorithms, both memory and stage consumption scale linearly with setups, but are more susceptible to use-case characteristics.

3.7 Summary

This chapter presents a comprehensive implementation of various in-network ML algorithms. By abstracting deployment approaches of in-network ML into three mapping methodologies (§3.2), this research extends the scope of supported algorithms and provides diverse choices for algorithm implementation. Specifically, the chapter details the mapping of four in-network ML algorithms using the direct-mapping methodology (§3.3), six algorithm mappings employing the encode-based methodology (§3.4), and five algorithm mappings utilising the lookup-based methodology (§3.5). Additionally, I theoretically analysed the

data plane resource consumption for each mapping methodology (§3.6). While specific experiments were not conducted in this chapter; all experimental results will be addressed in the subsequent chapter with the help of an automated framework.

CHAPTER 4

AUTOMATED DEPLOYMENT FRAMEWORK

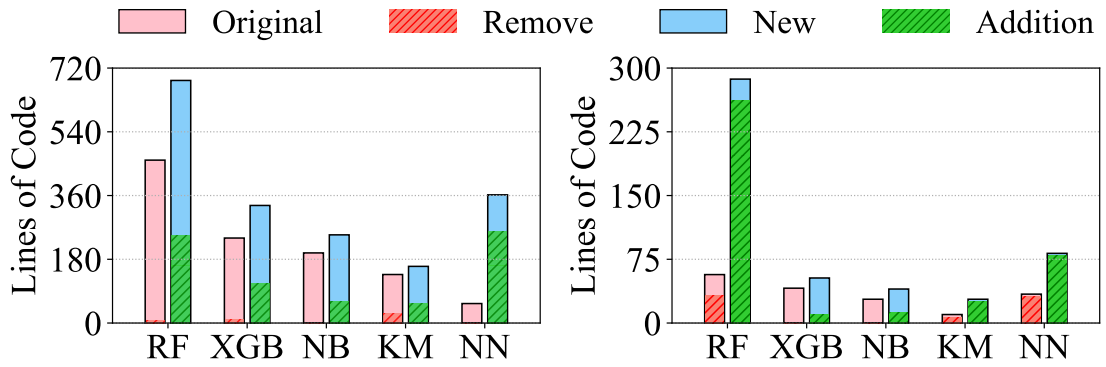
In-network ML inference provides high throughput and low latency. It has better power efficiency, can be deployed along the datapath, and improves applications' performance. Despite its advantages, implementing and updating in-network ML models is cumbersome and requires significant expertise in programmable data planes, which hinders the wide adoption of in-network ML algorithms (§4.1). In this chapter, I present Planter: a modular and efficient open-source framework for rapid prototyping of in-network ML models across a wide range of platforms and pipeline architectures (§4.2). Building upon the general mapping methodologies for ML algorithms in Chapter 3, a wide spectrum of in-network ML algorithms and their variations are supported. This framework is implemented using Python and P4, supports multiple architecture and target programmable network devices, and has already been further extended by users to new fields and applications (§4.3). With the help of the proposed Planter framework, in this chapter, I evaluate the in-network ML algorithms described in the previous chapter (§4.4). The evaluation results show that the proposed new mappings improve ML performance compared with previous model-tailored works, while significantly reducing resource consumption and co-existing with network functions. I also show that these algorithms can run at line rate on unmodified commodity hardware, providing billions of inference decisions per second. The contents of this chapter are based on framework design and evaluation-related materials published in [243, 244, 245, 246].

4.1 Motivation

In practical in-network ML deployment, model implementation is not a one-time task, particularly in the pursuit of optimal performance. Model comparison, tuning, or retraining is unavoidable, but conducting these procedures for in-network ML models is cumbersome. Figure 4.1 demonstrates the effect of ML model changes in terms of LOC. Common changes like increasing model size, adding more features, or moving between targets require changing hundreds of lines of code (LOC) in the data plane, the same scale as the entire original code. Changes also affect M/A table rules, added or removed through the control plane. A framework for rapid prototyping can save endless debugging and improve the efficiency of model deployment. The absence of a framework further hinders efficient model comparison, selection, and replacement, as well as swift migration among use cases. This chapter aims to narrow the gap to production deployment of in-network ML, and sets the following design goals:

Rapid Deployment. Mapping ML models to programmable network devices using P4 should be easy. The challenges of algorithm mappings have been addressed in the previous chapter. However, the deployment of in-network ML models is also complex. New or changed targets, architectures, and models may result in significant changes. An easy-to-operate solution is needed to handle the deployment process and flexibly adapt to changes (§4.2.1).

Extensibility and Portability. Extending and porting different inference solutions should require minimum effort. ML algorithms are emerging rapidly and new programmable network targets are reaching the market. The framework should be able to support new ML algorithms, architectures, and targets. It should be easy to add new use case scenarios and to port designs between differ-



(a) Data Plane (b) Control Plane

Figure 4.1: LOC changes as a result of model changes (UNSW dataset [140]). RF - depth of 2 to 5, XGB - 2 to 6 trees, NB - 2 to 5 features, KM - *v1model* to *TNA*, NN - 2 to 5 layers. All the used acronyms are listed in Abbreviations.

ent devices. This calls for a modular design with elements easily added, updated, or replaced, independently of other components (§4.2.2 - §4.2.3).

Automated Configuration. Not all in-network ML users have ML expertise. It is important to have a tool to handle compilation failures and automatically select the best hyperparameters under constraints. An automated frontend is needed to drive the in-network ML generation framework in realising models (§4.2.4).

4.2 Framework Design

Chapter 3 presented new and more efficient in-network ML mapping methodologies. Using the Planter framework, those can be rapidly prototyped on different targets. This section addresses four aspects in the design of Planter as an automated and "one-click" framework:

1. What should be the functionality supported by the framework, and what should be the respective framework architecture?

2. How to generate efficient data plane designs for different deployments?
3. How to create a modular framework that is extendable to different models, targets, architectures, and use cases?
4. How to provide ease of use, handle failures and tune hyper-parameters?

By addressing these aspects in this section (§4.2.1-4.2.4), I provide a framework that goes beyond the state-of-the-art and enables wide adoption of in-network ML (§4.4).

4.2.1 Functionality, Workflow, and Main Components

The goal of Planter is to provide as simple as possible in-network ML development environment for users. For example, model developers should be able to focus on model mapping without worrying about use cases, while use case practitioners can develop just the use cases and pick from a given set of models and targets. To achieve this goal, in Planter functions are strictly partitioned according to their type and are connected using standard interfaces. In turn, an in-network ML application is divided into four types of elements: ML model-related, use case-related, architecture-related, and target (test)-related. The main functions of Planter's backend workflow are shown in Figure 4.2. *Use case related functions (amber)* include a Data Loader ❶ for training & testing purposes and a Common P4 ❸ to generate use case-specific P4 code (code for feature extraction from data, e.g., `U` in Algorithm 4.1). *Model related functions (red)* are formed by a Model Trainer & Converter ❷ for training and converting a model to M/A format and a Dedicated P4 ❸ for generating model related P4 code (e.g., `M` in Algorithm 4.1)). *Architecture related functions (purple)* includes a Standard P4 ❸

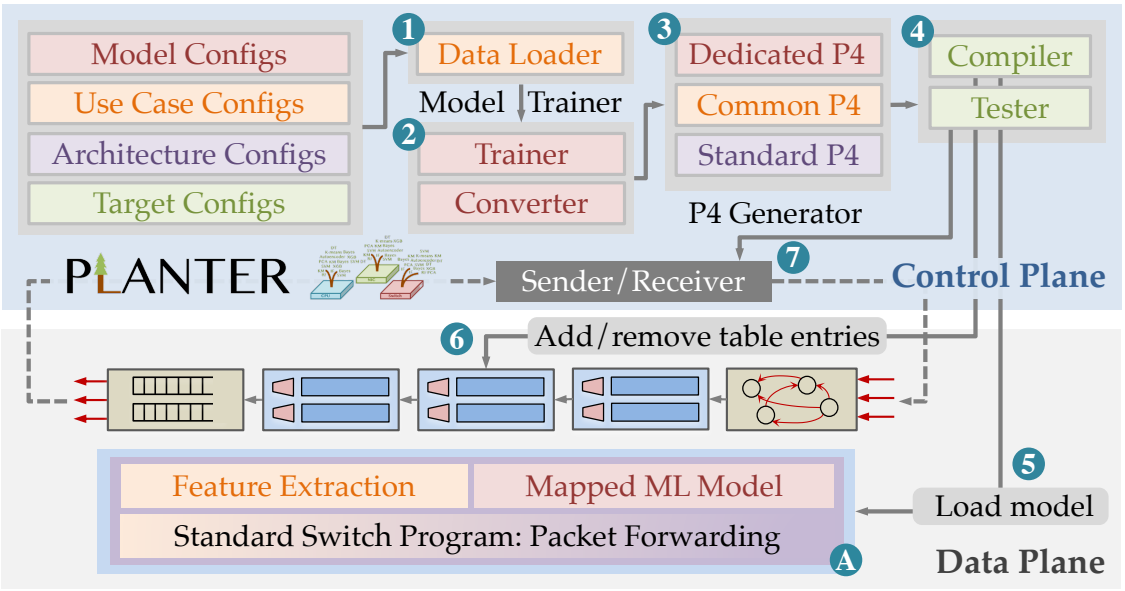


Figure 4.2: The Planter framework backend components and workflow steps (❶ to ❼).

generator that generates architecture-related P4 code and combines the outputs from Common and Dedicated P4 to generate the complete P4 program. *Target related functions (green)* include a Compiler ❹ that compiles and runs the generated in-network ML inference program and a Tester ❹ for validating the functionality of the program.

The workflow of Planter's backend, shown in Figure 4.2, has seven steps. In the first two steps, Planter loads a dataset ❶ and trains it ❷. The model is mapped to P4 ❸ using selected architecture and target. The generated P4 code is compiled ❹ and loaded to the target's data plane ❺. In step ❻, table entries and registers are loaded through the control plane. Last, in step ❼, an auto-generated functionality test is run on the target. The generated data plane, shown in Figure 4.2 Ⓐ, has three components: standard switching functionality, ML feature extraction, and ML inference. The ML feature extraction and inference are in parallel to the standard functionality, while the P4 parser operation is merged.

```
A #include <core.p4>
A #include <tna.p4>
U const bit<16> const1 = 0x01;
M const bit<16> const2 = 0x02;
A struct header_t{ // Header
U     header_U_h    header_U;
M     header_M_h    header_M;
A    }
A struct metadata_t{ // Metadata
U     bit<10> meta_data_U;
M     bit<10> meta_data_M;
A    }
A parser SwitchParser(...){ // Parser
U     state parse_header_U{...}
M     state parse_header_M{...}
A    }
A control SwitchIngress(...){ // Ingress
U     ... // Use case related Tables, Actions, Registers ...
U     ... // (for feature extraction)
M     ... // ML model related Tables, Actions, Registers ...
M     ... // (for inference)
A     apply{
U         ... // Use case model related logic
U         ... // (for feature extraction)
M         ... // ML related logic
M         ... // (for inference)
A        }}
A control SwitchEgress(...){ // Egress
A U M ...}
A Switch(...) main; // Main
```

Algorithm 4.1: Sample in-network ML P4 code. `A`: architecture-related code generated by Standard P4 in Figure 4.2 ❸. `U`: use case-related code generated by Common P4 in Figure 4.2 ❸. `M`: ML model code generated by Dedicated P4 in Figure 4.2 ❸.

4.2.2 Code Generator

The data plane's P4 code is mainly determined by the selected ML model, P4 architecture, and use case (three out of four function groups in § 4.2.1). As shown in Algorithm 4.1, when Planter generates P4 code, architecture-related (`A`) code forms the skeleton of the program, and use case-related (`U`) and model-related (`M`) code snippets are added into the skeleton. To simplify the coding experience, Planter provides commonly used architecture templates as skeletons in

the Standard P4 file (Figure 4.2 ❸). P4's sequential execution can be a problem when assembling the code snippets. Planter ensures the correct assembly order by utilising the Standard P4 to coordinate the Common & Dedicated P4 parts (Figure 4.2 ❸). This is achieved by alternately calling specific functions that generate ML (inference logic)-related code from Common P4 and use case (feature extraction)-related code from Dedicated P4 at each designated place (such as *ingress apply* block) to accurately merge the model and use case-related codes. This approach enables Planter's extensibility, fitting different models and use cases into the Standard P4 generator.

4.2.3 Modular Framework Design

Planter proposes a modular framework to enhance extensibility. The framework isolates the code within individual modules for each ML model, architecture, target, or use case. Modules are independent and can be flexibly and easily replaced through configuration. Additionally, Planter provides a set of common functions for all modules, such as exact-to-LPM/Ternary table conversion. Planter supports generating a dependency graph, showing all the used functions, and their dependencies. This simplifies debugging and informs modules' swapping.

Python	Architecture	Target	Model	Use Case	Data	Function
Min LOC	123	291	293	54	16	6
Avg LOC	137	542	477	91	88	29
Max LOC	145	793	665	127	148	130
Modules	6	6	52	10	20	18

Table 4.2: The number of LOC and modules (including module variations) in Planter modular design.

To illustrate the modularity of Planter, Table 4.2 presents the average and

maximum LOC of different modules in Planter. ML models require an average of 477 LOC, and at most 665 LOC. Supporting a new P4 architecture requires less than 150 LOC, and supporting a target requires less than a thousand LOC. This lightweight implementation of architectures, targets, and models is a key advantage of Planter. Shared framework functionality ("Function") requires 377 LOC in total. Combined, till now, Planter has 52 implemented ML modules, 6 targets and architecture models, 10 use case modules, and 20 datasets.

4.2.4 Planter Frontend

Although the Planter backend (§4.2.1-4.2.3) enables the realisation of ML models within the data plane, fitting the models on resource-constrained (commodity) hardware with ML performance goals needs to be addressed. The key challenge is hyperparameter selection, combining the desired inference accuracy and minimum resource consumption. To address this challenge, Planter provides a frontend optimiser that automates hyperparameter tuning, and fixes compilation failures.

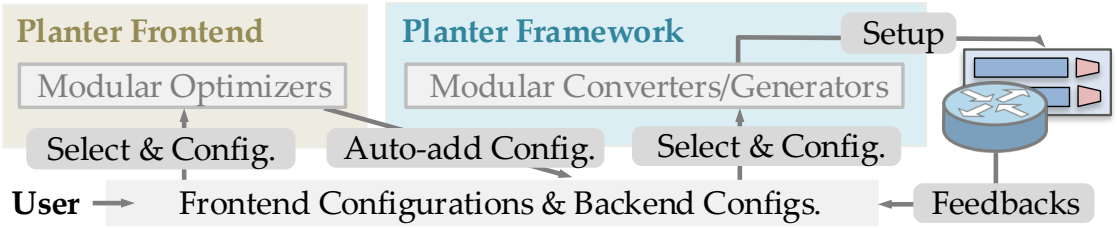


Figure 4.3: The combined Planter framework.

The frontend applies modular optimisation models, such as Bayesian Optimisation, to optimise parameters based on a predefined objective function. The optimisation objective can be expressed as metrics such as Accuracy (*acc*), F1 Score ($F1$), or the difference between the target accuracy (*target*) and the model

accuracy $n^{-|acc-target|}$ ($n > 1$). As illustrated in Figure 4.3, Planter requires users' input for frontend configurations, including datasets and model constraints. Based on these configurations and the selected objective function, the frontend identifies optimal hyperparameter settings. Subsequently, as shown in Figure 4.3, the frontend automatically generates backend configurations, utilises the backend (Figure 4.2) to produce data plane code & setup, and performs compilation and testing to verify the settings. The frontend receives the test results and triggers a subsequent training round if the outcomes are not satisfactory. For example, in case of a compilation failure, the frontend will regenerate a more moderately sized model. Planter's frontend is also part of the modular design in §4.2.3. While the backend supports experienced developers with customising model details, the frontend is designed to help users with limited ML experience to conduct intent-based development.

4.2.5 Planter Supported Algorithm Mappings

Planter incorporates models and mapping introduced in Chapter 3. Table 4.3 lists distinct implementations of ML models in Planter, categorised according to the three methodologies. As shown in the table, Planter has realised seven completely new mappings and four improved mappings proposed by this research as well as supports four existing mappings. Each implemented mapping in Planter has several variations, e.g., encode-based decision tree using Ternary or LPM table.

	Supported			Improved			New Mapping					
Types	SVM	KM	NN	NB	DT	RF	XGB	IF	KNN	PCA	AE	QL†
EB		✧$_2$			❙$_6$	✓$_{11}$	✓$_6$	✓$_4$	✓$_2$			
LB	✧$_2$	✧$_2$		❙$_4$						✓$_2$	✓$_2$	
DM			✧$_3$		❙$_2$	❙$_4$						✓$_2$

† As a reinforcement learning algorithm, QL is primarily implemented manually.

Table 4.3: Different ML models, and their implementation using the three proposed methodologies. Notation: ✓new, ❙ improved, ✧ supported. ✧$_n$, ❙$_n$ or ✓$_n$ indicates n supported variations. The QL refers to Q-Learning.

4.3 Implementation

The Planter framework is implemented in more than 57k LOC in Python, and is available at Planter's GitHub repository [238]. The framework trains a model using a python-based learning framework, currently PyTorch and scikit-learn (Sklearn). Training parameters are set by the user, Planter frontend, or using parameter tuning tools [93, 197]. A trained model is mapped into M/A format, and the framework saves table entries and generated weights (for neural networks) into `JSON/txt` (target dependent) files. Data plane P4 code is consequently automatically generated. Planter further generates `Bash` scripts to interact with the target for model deployment and verification. The provided control plane support is target-dependent, e.g., loading tables using P4Runtime. The targets currently supported include Intel Tofino and Tofino2 (switch-ASIC), AMD Alveo U280 (FPGA) over Open-NIC [216], P4Pi-BMv2 and P4Pi-T4P4S on Raspberry Pi [115], and BMv2. Pipeline architectures include v1model [155], Intel TNA [97], PSA [150], AMD xsa [217] and NVIDIA spectrum. Supported ML models and their variations are listed in Table 4.3, and use cases in §4.4.1.

4.4 Evaluation

The evaluation in this chapter focuses on the framework and mappings (Chapter 3), with practical application (e.g., complex feature extraction) in Chapter 5 and use cases in Chapter 7. The detailed hyperparameter setup used in this evaluation for each model is summarised in Table 4.4 and 4.5. Hyperparameters related to the converted model size on programmable network devices are defined in a gradient scale to differentiate the small (S)/medium (M)/large (L) model size. Huge (H) model size refers to setups on servers with nearly unlimited resources, primarily employed to explore optimal model performance. Other hyperparameters for each model remain as the default values defined in the *scikit-learn* package [154].

4.4.1 Methodology and Testbed Setup

Workloads: Planter has been applied in several works for various scenarios [238, 247, 245, 241, 244, 91, 228, 227, 225, 37], as discussed later in Chapter 7. Some example use cases and datasets include attack detection (using AWID3 [36], CICIDS 2017 [175], KDD99 [191], and UNSW-NB15 [140]), finance (NASDAQ TotalView-ITCH [142] and Jane Street Market Prediction [76]), QoE (Requet [79]) and flowers classification (Iris [60]) for brevity. This evaluation presents only the results for attack detection (throughput and latency-sensitive, using CICIDS 2017 and UNSW-NB15 dataset) and finance (latency-sensitive, using Jane Street Market Prediction dataset). Attack detection uses 5 features, commonly used for traffic classification [160, 100]: Source IP (first 8 bits), Destination IP (first 8 bits), Source Port, Destination Port, and protocol and classifies traffic as

either normal or malicious. The first octet of the IP address is a coarse indicator of source/destination network. In the Jane Street Market Prediction dataset, five features (42, 43, 120, 124 and 126) from 130 anonymised real stock market data are used to predict buy or sell for each trading opportunity [76]. An additional evaluation of these two use cases is provided in Chapter 5 and results for other use cases are included in [245] and Chapter 7.

Testbed setup: The testbed uses two servers for traffic generation and monitoring (ESC4000A-E10, AMD EPYC 7302P CPUs, 256GB RAM, Ubuntu 20.04LTS, ConnectX-5 NICs). PTP with timestamping in the NICs is used for latency measurements. The evaluated platforms are a 1) Tofino switch (APS-Networks BF6064X, SDE 9.6.0) using a snake configuration for throughput tests. 2) AMD Alveo U280 FPGA. 3) NVIDIA BlueField-2 DPU. In addition, 4) P4Pi [115] running on Raspberry Pi 4 Model B with 8GB RAM is evaluated twice: using v1model over BMv2 and using T4P4S [205]. The P4Pi testbed is connected to a server with an Intel Xeon W-2133 CPU and 64 GB RAM. Planter-generated in-network ML models were also deployed on Dell IoT Gateway [53] and have been evaluated in [226].

Parameter settings: Mapped in-network ML models are explored using four different model sizes: small (S), medium (M), large (L), and huge (H), with detailed setups of model parameters provided in Table 4.4 and 4.5. The model size refers to the converted data plane model size, which is a function of both training and conversion parameters. Small to large in-network ML models are expected to fit on the target data plane. Huge models represent the maximum inference potential of a model (per dataset) running on a server.

Evaluation metrics: The following metrics are used in the evaluation:

Small (S) Model (for in-network ML)							
	Action Bits	Depth	Num Trees	Max Leafs	lr	Batch Size	Epoch
SVM	8						
DT_{EB}		4		1000			
DT_{DM}		4		1000			
RF_{EB}		4	6	1000			
RF_{DM}		4	6	1000			
XGB		4	6	1000			
IF			3		128 (Num Instance†)		
NB	8						
KM_{LB}	8						
KM_{EB}		2					
KNN		2			5 (Num Neighbors‡)		
NN	1	1(16)			0.01	100	50
PCA	8						
AE	8				0.01	100	50
Medium (M) Model (for in-network ML)							
	Action Bits	Depth	Num Tree	Max Leaf	lr	Batch Size	Epoch
SVM	16						
DT_{EB}		5		1000			
DT_{DM}		5		1000			
RF_{EB}		5	9	1000			
RF_{DM}		5	9	1000			
XGB		5	9	1000			
IF			9		128 (Num Instance†)		
NB	16						
KM_{LB}	16						
KM_{EB}		3					
KNN		3			5 (Num Neighbors‡)		
NN	1	1(32)			0.01	100	50
PCA	16						
AE	16				0.01	100	50

† Num Instance indicates the number of samples drawing from the training data used to train each base estimator, which is t in Equation 3.1.
‡ Num Neighbors indicates the number of neighbours used to determine the classification of a specific query point.

Table 4.4: Parameters settings for (S)mall and (M)edium model on data plane device/server. F - Full precision. lr - learning rate. This setting is used in all evaluation processes in this chapter to unify the evaluation setups.

Large (L) Model (for in-network ML)							
	Action Bits	Depth	Num Trees	Max Leafs	lr	Batch Size	Epoch
SVM	32						
DT_{EB}		6		1000			
DT_{DM}		6		1000			
RF_{EB}		6	12	1000			
RF_{DM}		6	12	1000			
XGB		6	12	1000			
IF			12		128 (Num Instance†)		
NB	32						
KM_{LB}	32						
KM_{EB}		4					
KNN		4			5 (Num Neighbors‡)		
NN	1	1(48)			0.01	100	50
PCA	32						
AE	32				0.01	100	50
Huge (H) Model (for server-based ML)							
	Action Bits	Depth	Num Tree	Max Leaf	lr	Batch Size	Epoch
SVM	F						
DT_{EB}		30		100000			
DT_{DM}		30		100000			
RF_{EB}		30	200	100000			
RF_{DM}		30	200	100000			
XGB		30	200	100000			
IF			200		1280 (Num Instance†)		
NB	F						
KM_{LB}	F						
KM_{EB}		F					
KNN		F			5 (Num Neighbors‡)		
NN	F	1(48)			0.01	100	50
PCA	F						
AE	F				0.01	100	50

† Num Instance indicates the number of samples drawing from the training data used to train each base estimator, which is t in Equation 3.1.
‡ Num Neighbors indicates the number of neighbours used to determine the classification of a specific query point.

Table 4.5: Parameters settings for (L)arge model on data plane device/server and (H)uge model size on server. F - Full precision. lr - learning rate. This setting is used in all evaluation processes in this chapter to unify the evaluation setups.

1. ML performance: *Accuracy* and *F1 score* are used to evaluate ML inference performance.
2. Scalability performance: *Memory utilisation*, *Table entries*, and *Number of stages* are used to evaluate scalability.
3. System performance: *Throughput* and *Latency* (measured by ptp4l) are used to evaluate the system performance of mapped models.
4. Framework performance: *Model training time* and *trained model conversion time* are used to assess Planter's runtime performance.

On Tofino, following non-disclosure agreement (NDA) guidelines, I report the latency relative to Intel's *switch.p4* reference program. *switch.p4* is an L2/L3 switch program for Tofino, including 10 network functions such as load balancing, tunnelling, firewall, and statistics.

4.4.2 ML Performance

This section focuses on evaluating ML-related performance, especially when compared with server-based benchmarks and state-of-the-art works.

ML Inference Accuracy: The inference performance evaluation checks if the mapped in-network ML models have a similar inference accuracy as running an identical inference task on a server with the same setup, and how the size of the model affects the accuracy. The results are presented in Table 4.6 and 4.7. It is not suggested to compare *between* ML models in Table 4.6 and 4.7 as the preference of models varies among different use cases.

The upper part in both tables shows previously proposed mappings [218, 243,

		CICIDS				UNSW					
		Switch (M)		Sklearn (M)		Switch (M)		Sklearn (M)		Server (H)	
Work	Model	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1
SwitchTree [119]	DT_{DM}	99.92	99.92	99.92	99.92	99.40	**94.53**	99.40	**94.53**	99.40	94.31
pForest [33]	RF_{DM}	99.80	99.79	99.80	99.79	99.38	94.44	99.38	94.44	99.42	94.51
IIsy [218]	KM_{LB}	58.40	56.80	58.40	56.80	71.28	41.88	71.28	41.88	71.28	41.88
Clustreams [66]	KM_{EB}	56.92	55.75	58.40	56.80	72.69	42.37	71.28	41.88	71.28	41.88
IIsy [218]	SVM	59.24	37.20	95.04	94.94	97.31	49.32	99.23	93.51	99.23	93.51
N3IC [184]	NN‡	92.09	92.00	99.96	99.96	98.33	85.68	99.25	93.67	99.25	93.68
Planter	DT_{EB}	99.92	99.92	99.92	99.92	99.40	**94.53**	99.40	**94.53**	99.40	94.31
Planter	RF_{EB}	99.80	99.79	99.80	99.79	99.37	94.41	99.38	94.44	99.42	94.51
Planter	XGB	**99.98**	**99.98**	**99.98**	**99.98**	**99.42**	**94.53**	**99.42**	**94.53**	**99.43**	**94.59**
Planter	IF	44.89	35.35	37.90	31.08	84.86	58.90	63.83	45.07	86.33	55.05
Planter	NB	98.99	98.95	98.99	98.96	99.25	93.68	99.25	93.68	99.25	93.68
Planter	KNN	69.33	60.63	99.38	99.36	87.51	31.55	99.30	93.17	99.30	93.17
Planter	PCA*	76.12	74.92	76.19	75.00	97.45	65.42	97.89	67.73	97.89	67.73
Planter	AE*	99.92	99.92	99.92	99.92	99.24	93.55	99.24	93.55	99.28	93.53

Bold and underline indicate the best and second-best results among all models respectively.

‡ NN is trained with PyTorch instead of Sklearn.

* Results of PCA and AE are the accuracy of (S)mall DT using new features.

Table 4.6: Model Accuracy (ACC) and F1 Score (F1) on CICIDS and UNSW datasets, by using (S)mall, (M)edium, (L)arge, and (H)uge models.

247, 33, 66, 184] implemented in Planter, where these mappings were not implemented before on a switch-ASIC. The lower part in both tables shows Planter's new & improved mappings. As shown in Table 4.6, for the same model size, all the models have a similar accuracy performance on the programmable switch as on Sklearn or a baseline server, verifying Planter's mapping has no or minor accuracy loss. The accuracy is checked using 10-fold cross-validation, with a standard deviation of less than 0.05% for top-performing models, indicating its statistical significance.

Table 4.7 compares the resource performance for different sizes of models. As model size increases, some models achieve slightly higher accuracy with more switch resources required. All small and medium-sized models proposed or up-

		UNSW										
		ACC (Switch)		Memory (%)			Latency (Relative)			Stages (Tofino)		
Work	Model	S	L	S	M	L	S	M	L	S	M	L
SwitchTree [119]	DT_{DM}	99.34	99.41	1.46	1.72	1.98	81.16	88.36	88.36	11	13†	15†
pForest [33]	RF_{DM}	99.25	99.39	7.71	14.01	NF	88.36	89.04	NF	11†	14†	NF
IIsy [218]	KM_{LB}	71.55	71.28	3.13	3.96	5.78	21.58	21.58	21.58	7	7	7
Clustreams [66]	KM_{EB}	77.21	71.30	0.40	3.16	NF	**19.52**	**19.52**	NF	2	2	NF
IIsy [218]	SVM	97.31	99.23	2.81	3.23	4.12	26.37	35.27	35.30	9	9	9
N3IC [184]	NN‡	98.33	97.50	NF	NF	NF	NF	NF	NF	NF	NF	NF
Planter	DT_{EB}	99.34	99.41	1.18	**1.34**	**1.34**	26.37	26.37	26.37	2	2	**2**
Planter	RF_{EB}	99.25	99.39	1.81	2.59	3.94	39.04	39.40	45.89	3	4	4
Planter	XGB	**99.40**	**99.45**	1.70	6.65	NF	33.22	45.78	NF	3	5	NF
Planter	IF	81.74	NF	2.01	9.01	NF	36.30	43.33	NF	5	5	NF
Planter	NB	99.25	99.25	3.28	4.22	6.20	28.77	28.77	28.77	8	8	8
Planter	KNN	78.24	92.73	**0.23**	1.89	21.01	20.74	20.74	22.22	**1**	**1**	5
Planter	PCA*	97.29	97.47	5.78	5.78	5.78	20.89	20.89	**20.89**	6	6	6
Planter	AE*	99.23	99.28	5.89	5.89	5.89	21.58	21.58	21.58	7	7	7

Bold and underline indicate the best and second-best results among all models respectively.

‡ NN is trained with PyTorch instead of Sklearn.

* Results of PCA and AE are the accuracy of (S)mall DT using new features.

Table 4.7: Model Accuracy (ACC), resources and latency relative to *switch.p4*. Using (S)mall, (M)edium, (L)arge and (H)uge models. Some models are not feasible (NF) on Tofino but are feasible (†) on Tofino2.

graded by Planter are feasible on commodity hardware with less than 8 stages, 9.1% memory, and 45.78% of relative latency. The optimised mapping methods of Planter can achieve comparable accuracy with existing mapping solutions, but with significantly lower resource consumption. For example, a large decision tree (DT) model in Planter can reach the same accuracy as a large SwitchTree model [119], while reducing memory by 30%, latency by 70%, and stages by 87%. Resource consumption varies between Planter targets. For example, deploying a decision tree model on U280 FPGA has resource usage as low as 15% LUTs and 11% registers.

Planter was integrated with *switch.p4*, an Intel reference L2/L3 switch. The *switch.p4* functionality coexists with Planter's new or upgraded models with no

or minimal cost in stages and latency, but with higher resource utilisation.

Comparison with State-of-the-Art: I compare Planter with IIsy's proposed decision tree (DT), support vector machine (SVM), naïve Bayes (NB), and *k*-Means (KM) [218], SwitchTree/pForest decision tree and random forest [33, 119], Clustreams *k*-Means [66], N3IC's neural network (NN) [184], and Homunculus/Taurus's neural network [195, 197] in terms of inference performance and scalability.

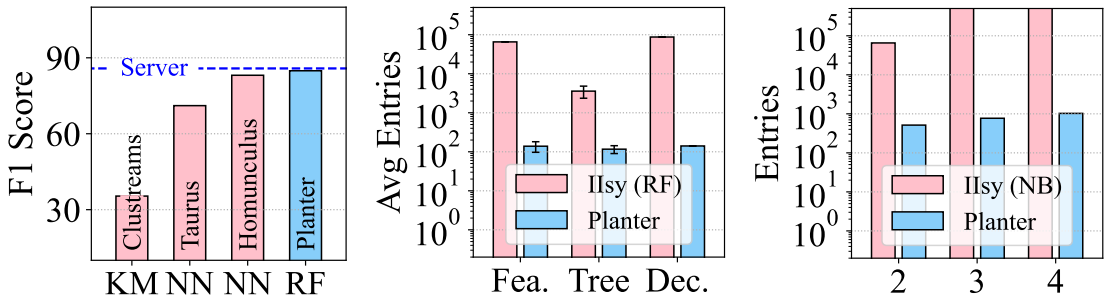


Figure 4.4: Comparison with state-of-the-art in F1 score (a) and table entries (b) & (c). UNSW dataset is used.

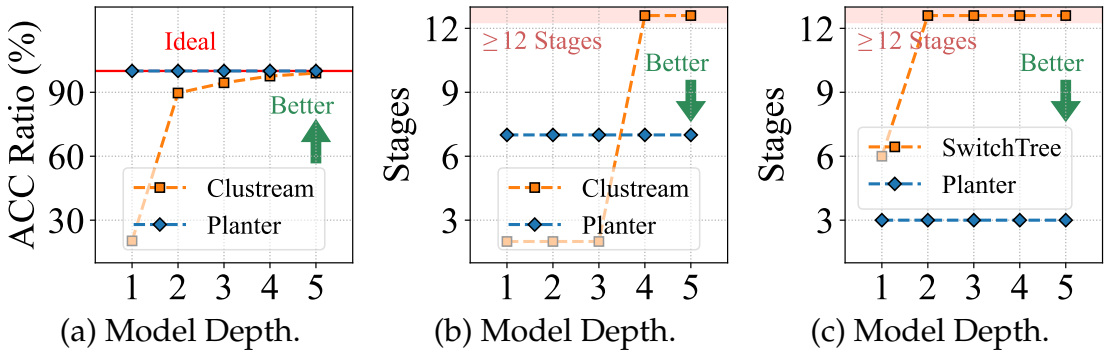
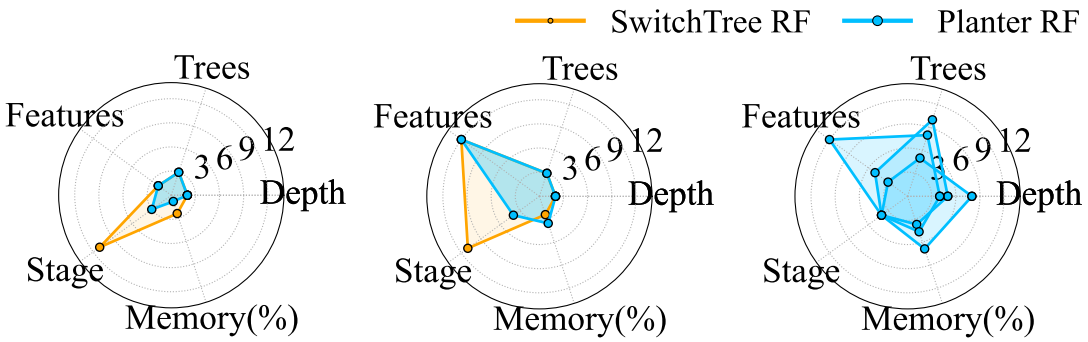


Figure 4.5: Comparison with state-of-the-art in accuracy (a) and stage consumption (b) & (c). UNSW dataset is used.

Figure 4.4 (a) compares the accuracy performance of Planter on Tofino and existing solutions where Clustreams is deployed on Tofino, and Taurus and Homunculous are implemented on Taurus backend using the modified ASIC (results from [195, 197]). Planter achieves a higher F1 score (84.88%) than Clustreams (35.4%), Taurus (71.1%) and Homunculus (83.1%), and similar to server



(a) Tiny Model. (b) Max. SwitchTree. (c) Max. Planter.

Figure 4.6: Comparison with state-of-the-art in stages and memory consumption, for (a) tiny model, (b) maximum (Max.) SwitchTree, and (c) maximum (Max.) Planter. Finance dataset is used.

performance. Compared with IIsy on Tofino, for the same model setup, Planter achieves lower memory consumption as shown in Figure 4.4 (b) & (c). Planter reduces the number of table entries in all types of tables for random forest (RF) and prevents table entry explosion by turning multiplications into additions for naïve Bayes (NB).

Figures 4.5 (a)-(c) compare Planter with Clustreams using k-Means and SwitchTree using decision tree. Clustreams has limited scaling capability, as table entries may explode when seeking better accuracy, compared with Planter's accurate and more resource efficient solution for larger models. Similarly, Planter maintains a constant number of stages for tree models, compared with the increasing number of stages in SwitchTree. Planter demonstrates better scalability in commodity hardware by parallelising the inference processing within a pipeline to avoid exceeding the stage limitation as model size scales up.

Figures 4.6 (a)-(c) compare the resource consumption and scalability of Planter's ensemble tree model (RF_{EB}) with SwitchTree (also pForest) RF_{DM} using ditrect-mapping [119, 33] under the finance use case. When using a tiny model

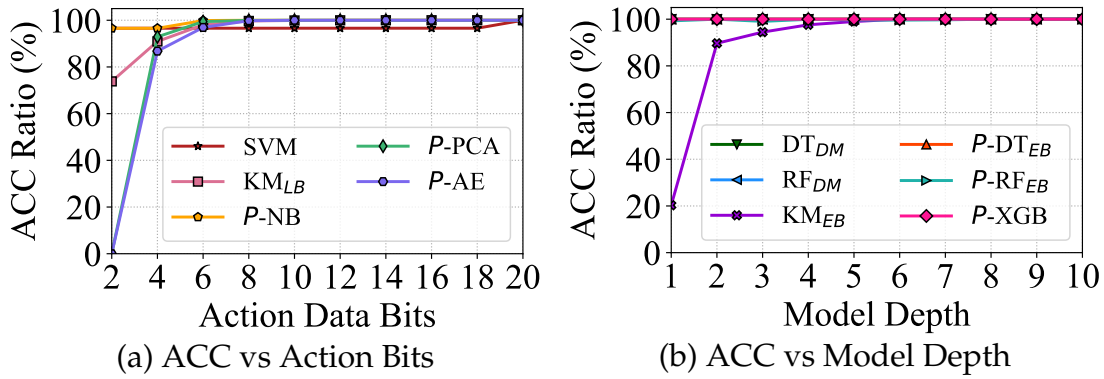
size (3 trees, depth of 2), as shown in Figure 4.6 (a), Planter requires only 3 stages and 0.82% memory, compared with 11 stages and 2.4% memory in SwitchTree. The maximum size of a SwitchTree RF model fitting on Tofino, as shown in Figure 4.6 (b), is again 3 trees and a depth of 2, but using 12 features. For the same model, Planter requires 1.5% more memory but uses only 4 stages, 7 stages less than SwitchTree. Figure 4.6 (c) shows combinations of hyperparameters settings of Planter, all too big for SwitchTree.

Combined, Figures 4.4 to 4.6 demonstrate that Planter outperforms prior works in both accuracy and resource utilisation. This is due to the optimised mapping methods introduced in Chapter 3.

4.4.3 Scalability Performance

Scalability and Relative Accuracy: I examine whether Planter can perform effective algorithm mapping with various hyperparameters and adapt to different resource constraints. Specifically, this evaluation studies the effect of action data bits (action field's width, which can control quantisation accuracy) and model depth on models' relative accuracy, comparing the switch's output with Sklearn's result on a server.

Figure 4.7 shows the switch accuracy relative to server accuracy. For lookup-based models in Figure 4.7 (a), the relative accuracy increases as the number of action data bits increases (more accurate intermediate results are stored). Among solutions, Planter has a negligible accuracy loss with just 4 action data bits requirements, which can save 40% memory in practice (more action data bits means higher memory consumption). In Figure 4.7 (b), compared to server-based models with the same model size, the accuracy of Planter's models re-



(a) ACC vs Action Bits (b) ACC vs Model Depth

Figure 4.7: The ratio of accuracy (ACC), data plane model relative to the server. KM$_{LB}$ & SVM refers to IIsy [218]; DT$_{DM}$ & RF$_{DM}$ refers to SwitchTree [119] & pForest [33]; KM$_{EB}$ refers to Clustreams [66]. *P-* refers to Planter proposed or upgraded model.

mains stable as model depth changes. It indicates that Planter's solutions can consistently achieve high mapping accuracy across a range of model sizes, effectively adapting to different resource constraints. Compared with KM$_{EB}$ (Clustreams [66]) and DT/RF$_{DM}$ (SwitchTree/pForest [119, 33]) that can easily lead to table entry or stage explosion when the model depth grows, Planter's solutions improve the mapping precision when under identical memory consumption.

Resources Scalability: Both training & mapping configurations (model depth, action data bits, and number of trees) and use case inputs (number of features and unique feature values) influence the applicability and scalability of Planter's in-network ML models. The resource scalability of each model is evaluated in two dimensions: the number of table entries and the number of pipeline stages. Table entries indicate the potential memory requirements from the switch, and the number of stages indicates remaining M/A stages for model growth and non-parallel functionality.

Figures 4.8 (a) & (b) show that as a model's depth increases, more table entries are required in all encode-based models and direct-mapping tree-based mod-

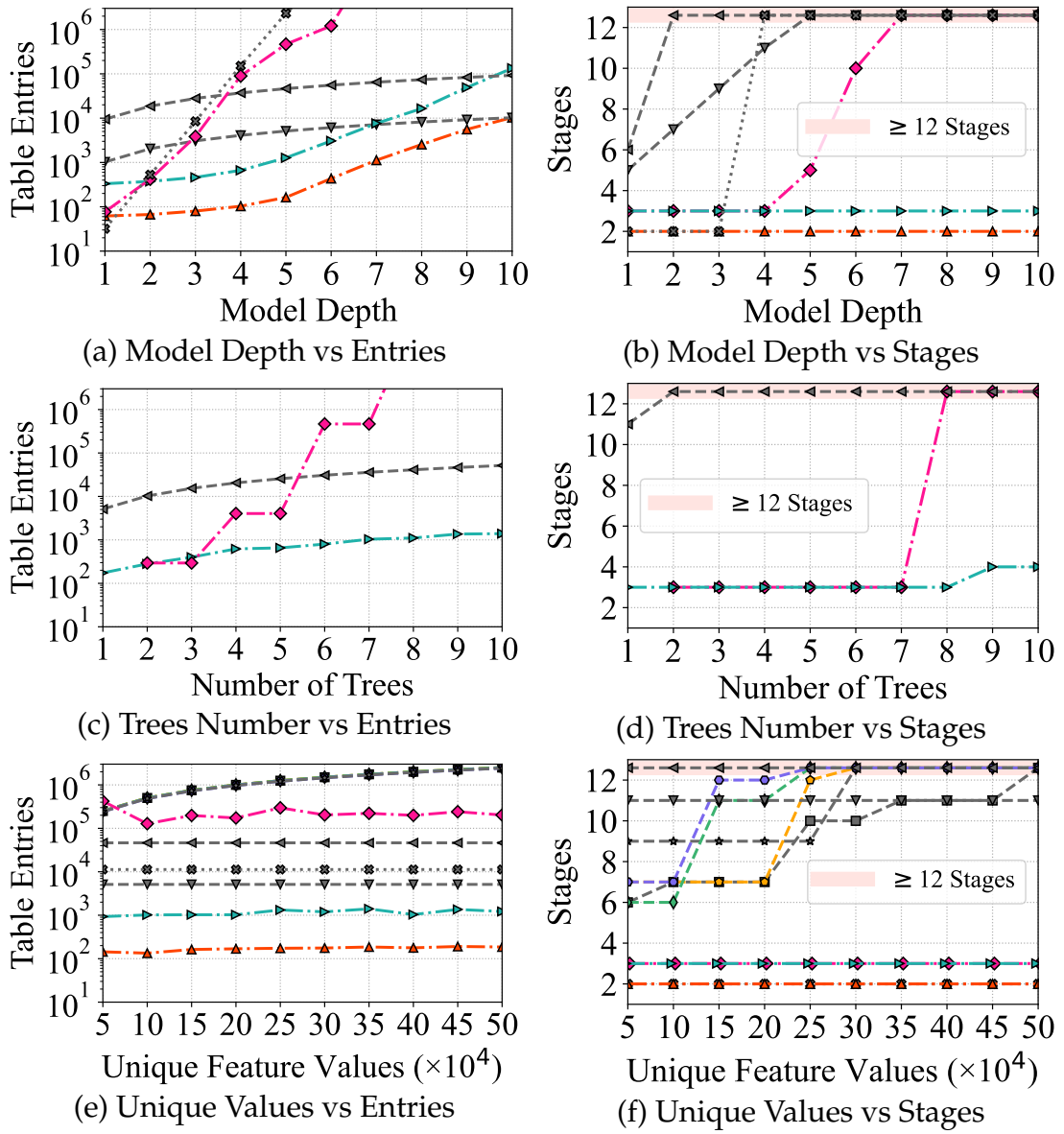


Figure 4.8: Memory and stage scaling with hyperparameters and feature properties (UNSW dataset). Flat or gentle curves indicate better resource scalability with model parameters. Model name references are same as Figure 4.7.

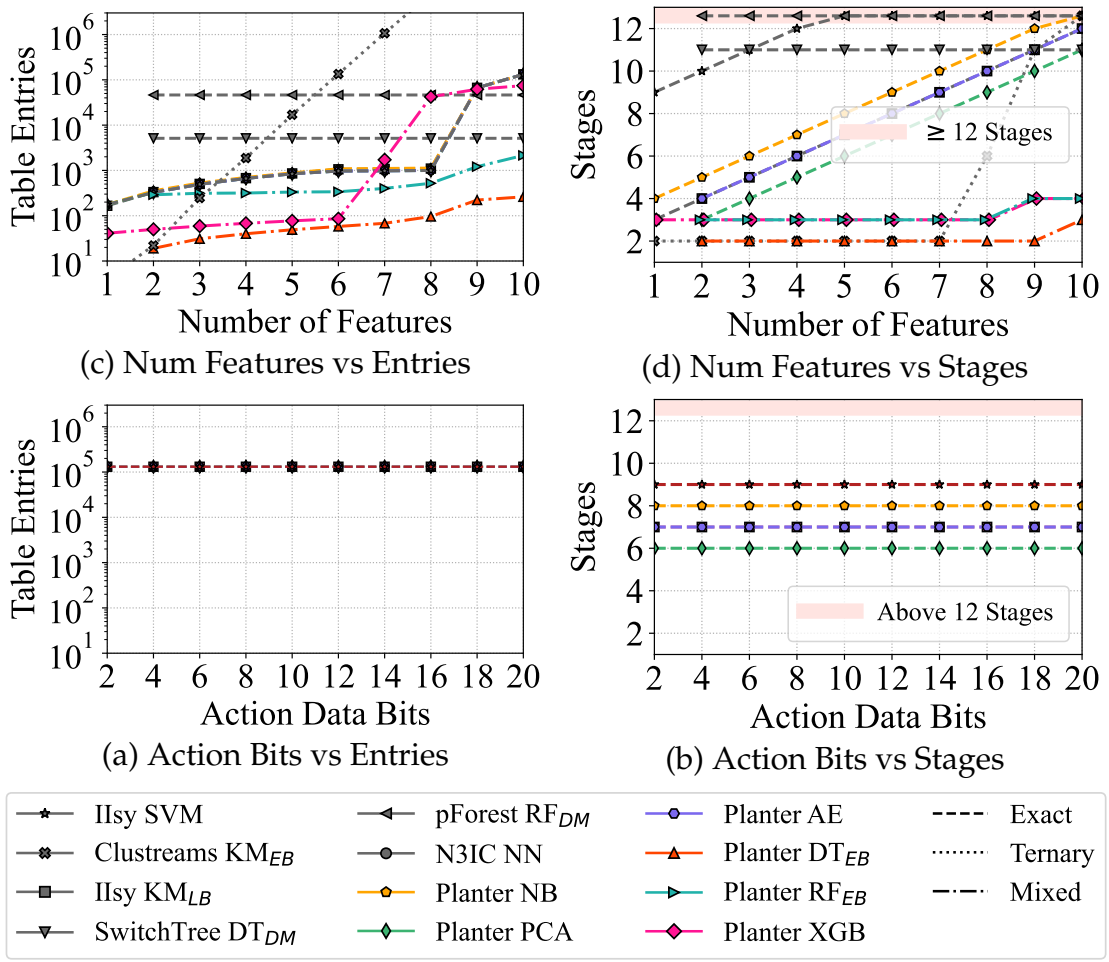


Figure 4.9: Memory and stage scaling with mapping setups (action data bits) and feature properties (UNSW dataset). Flat or gentle curves indicate better resource scalability with model parameters. Model name references are same as Figure 4.7.

els. Among them, direct-mapping solutions have a comparatively slower increment in table entries. Encode-based tree models are more stable in terms of stage consumption. Figures 4.8 (c) & (d) show that as the number of trees increases, direct-mapping tree models require 8 more stages than encode-based tree models, except in cases of excessive table entries. In Figures 4.8 (e) & (f), the feature range, which is the number of unique feature values per feature, only influences lookup-based models' stage and memory consumption. Figures 4.9 (a) & (b) show that except for direct-mapping tree-based models, models consume more

table entries as the number of features increases. In terms of stage consumption, only lookup-based models have a strong correlation to the number of features. Figures 4.9 (c) & (d) show that the number of action data bits does not influence the required number of table entries and the required number of stages. Note that the evaluated models are only those where action bits are a parameter. However, the number of used action data bits has an impact on calculation & inference errors as well as memory consumption. Figure 4.10 shows the relative error of a result (e.g., hyperplane equation in SVM) calculated on programmable network devices (e.g. Tofino switch), compared with the same equation calculated on a server. While this error is small (less than 0.001%), the more important result is the misclassification due to calculation error: zero for SVM and *k*-Means, and 0.00003% for naïve Bayes when using action width of 16 bit. This error is due to extremely low probabilities, and can be eliminated by encoding the results of naïve Bayes calculations, rather than normalising values. As shown in Figure 4.10, increasing the number of bits in an action has a minor effect on overall memory consumption, but can significantly reduce calculation errors.

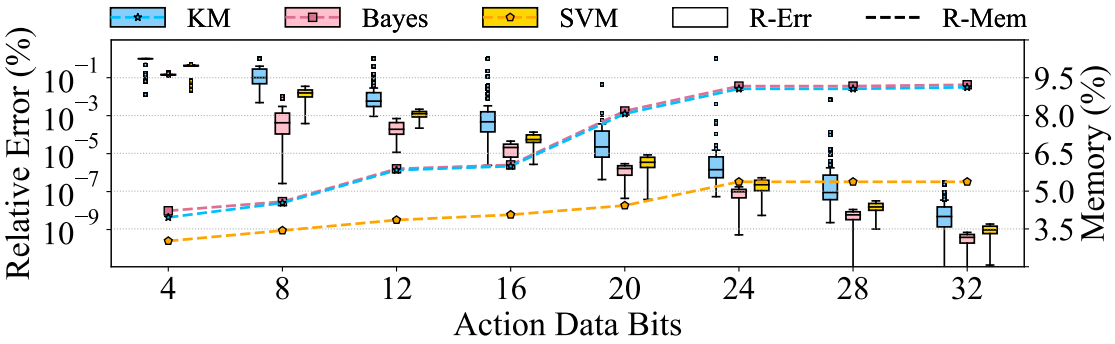


Figure 4.10: Calculation error in SVM, Bayes and K-means.

Some insights based on this evaluation are:

1. The scalability of encode-based models is mostly affected by the models' parameters (e.g., number of trees/model depth) and less by the use case

(e.g., range/number of features).

2. Lookup-based models' resources change with use case properties (e.g., range/number of features), and typically not with model parameters.
3. Direct-mapping models' scalability is model-dependent. They are usually bounded by stages, and less by memory.

This evaluation also illustrates Planter's widely supported model types and their characters. This diversity can make Planter adaptable to a wide range of use cases. More details related to the use cases are discussed in Chapters 5 and 7.

4.4.4 General System Performance

In terms of system performance, the throughput evaluation of different models is shown under the attack detection use case (UNSW), which is a volumetric use case. Latency is shown using price movement prediction use case (Jane Street Market Prediction), which is latency sensitive.

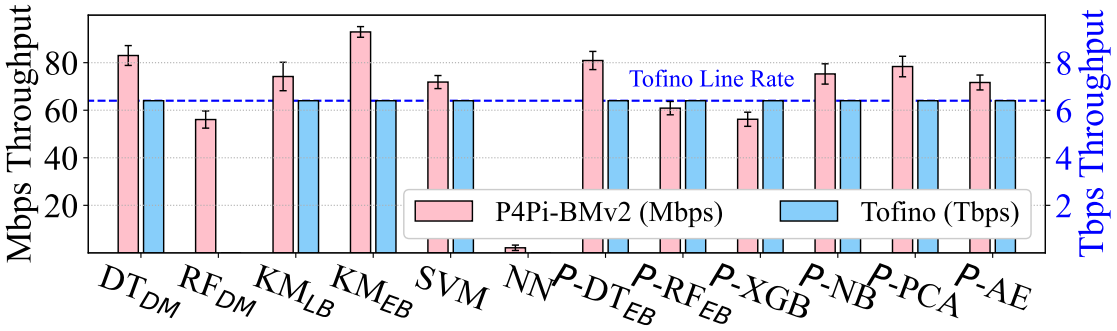


Figure 4.11: Throughput of ML algorithms for attack detection on Tofino (in Tbps) and P4Pi (in Mbps). DT_{DM} and RF_{DM} refers to SwitchTree [119] & pForest [33]; KM_{LB} & SVM refers to IIsy [218]; KM_{EB} refers to Clustreams [66], NN refers to N3IC [184].

Throughput: Throughput tests record the throughput of each in-network ML algorithm on a Tofino switch and P4Pi, as shown in Figure 4.11. The baseline

throughput of basic forwarding is 6.4Tbps on Tofino and 94Mbps on P4Pi. On a Tofino switch, full 6.4Tbps is achieved for all feasible models (Table 4.6 and 4.7). On P4Pi, which essentially runs a software switch on a CPU, the results vary between models. Seven of the models achieve more than 80% of the baseline throughput. Ensemble models (RF_{EB}, RF_{DM}, and XGB) and NN have degraded throughput on P4Pi, due to their increased use of resources (e.g., pipeline stages, table entries, logical operations, and especially registers). The throughput of in-network ML on other targets can be found in Section 4.4.5.

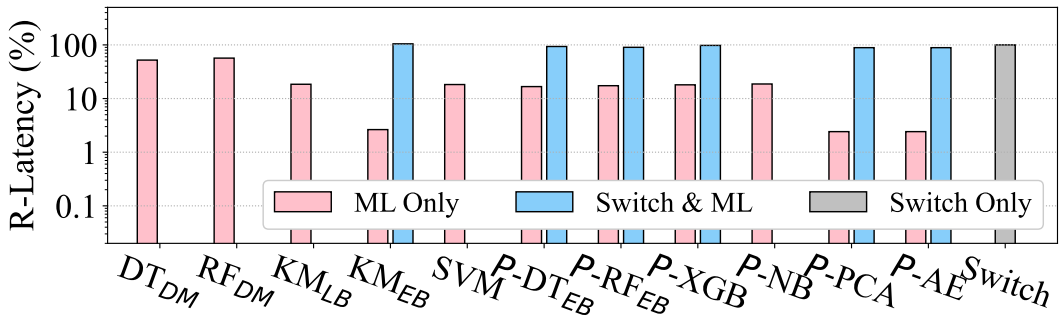


Figure 4.12: The relative latency (R-Latency) on Tofino in the financial prediction use case, measured for standalone ML, ML combined with *switch.p*4, and standalone *switch.p*4.

Latency: Latency tests are conducted with finance use cases (Jane Street Market Prediction dataset) where Planter's models can achieve sub-microsecond latency. In compliance with Intel NDA, I report the relative latency in Figure 4.12. The baseline is the latency of *switch.p4*, an L2/L3 reference switch with 10 network functions. Due to resources occupied by the reference switch, some models failed to be deployed on Tofino when coexisting with these network functions, which shows zero in the figure. For those not running out of resources, when only the ML models are deployed without additional functions, the latency in most models is less than 22% of *switch.p4*. When the ML models are combined with *switch.p*4, there is an overhead of less than 4.7% for all feasible algorithms. Compared with previous works, Planter's mappings require less logic for sim-

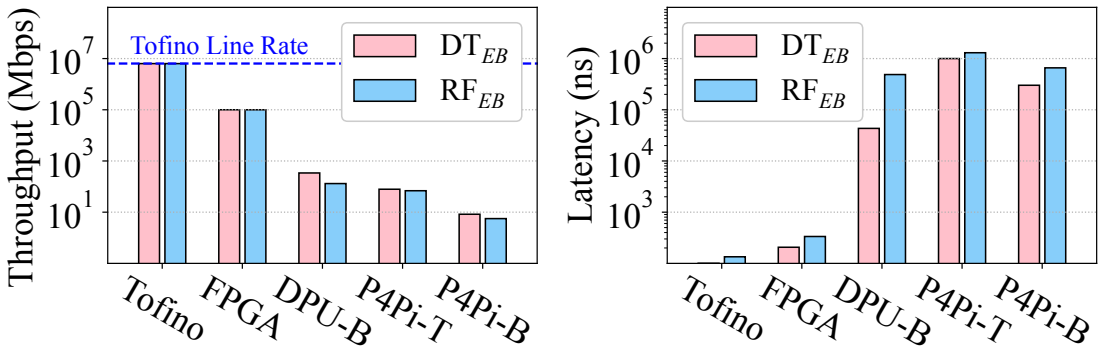
ilar models and are more compatible with other switch functions in resource-constrained targets. The latency of in-network ML on other targets can be found in Section 4.4.5.

4.4.5 System Performance on Different Targets

The system performance of two sample in-network ML algorithms, DT_{EB} and RF_{EB}, is evaluated on different targets[1]. As illustrated in Figure 4.13, hardware targets, such as Tofino and FPGA, achieve line rate throughput (6.4Tbps and 100Gbps, correspondingly). In contrast, software switches (BMv2 or T4P4S), running on the ARM cores of P4Pi and DPU, reach a throughput in the range of tens to hundreds of Mbps. Similarly, hardware targets achieve microsecond-scale latency, whereas software targets achieve sub-millisecond latency. Furthermore, on software targets the complexity of the model impacts system performance; the more complex random forest algorithm has lower throughput and higher latency on these targets. The model complexity does not have a notable impact on the performance of hardware targets. To provide more information about each of the targets:

FPGA. Planter's FPGA support is evaluated using AMD Alveo U280. Vitis Networking P4 is used to compile the generated P4 code to an IP block with standard AXI interfaces for the OpenNIC shell. Taking DT_{EB} and RF_{EB} (encode-based) as examples, the baseline latency through a forwarding-only program is about a microsecond. The latency added by a decision tree model is 170 nanoseconds, and random forest latency is approximately 320 nanoseconds, aligned with compiler prediction. Both In-network ML models achieve 100Gbps, full line rate.

[1]Thank Liam Perreault for helping in configuring Planter on FPGA, and Mingyuan Zang for helping in evaluating Planter on P4Pi.



(a) Targets vs Throughput.

(b) Targets vs Latency.

Figure 4.13: Throughput and latency of encode-based decision tree (DT_{EB}) and random forest (RF_{EB}) on different target devices. -B refers to BMv2 and -T refers to T4P4S.

DPU. Planter currently supports in-network ML on NVIDIA BlueField-2 DPU using BMv2 running on its ARM cores. P4C is used to compile the P4 code with v1model architecture to a BMv2 software switch. Under this setup, compared to the baseline of simple forwarding, DT_{EB} introduces an additional latency of approximately 43 µs, and RF_{EB}'s latency is approximately 487 µs. In terms of throughput, using BMv2 performance configuration [47], decision tree can reach about 360 Mbps and random forest is approximately 131 Mbps.[2]

P4Pi-T4P4S. T4P4S [205] is an open-source compiler that generates a target-agnostic software switch using Data Plane Development Kit (DPDK). In this scenario, T4P4S is running on top of P4Pi [115]. Taking the same DT_{EB} and RF_{EB} models, the baseline latency is around 1 ms with basic forwarding functions. When DT_{EB} and RF_{EB} are enabled, the latency increases to 2 ms and 2.3 ms, correspondingly. The baseline switch throughput is 100 Mbps. When DT_{EB} and RF_{EB} are deployed, the throughput decreases to 78.7 Mbps and 68.8 Mbps.

P4Pi-BMv2. P4Pi is also evaluated using v1model over BMv2 software switch,

[2]NVIDIA's P4 compiler for DPU, currently not generally available, will enable higher performance.

using BMv2 performance configuration [47] and the same testbed as P4Pi-T4P4s. The DT_{EB} and RF_{EB} models deployed on BMv2 achieve a throughput of 80 Mbps and 60 Mbps. The latency results are 2.1 ms and 2.5 ms when DT_{EB} and RF_{EB} are deployed, with a baseline latency of 1.1 ms.

4.4.6 Framework Performance

Framework Execution Time: I measure the time required to load a dataset, train a model, convert the trained model, test table entries, compile the mapped model to a target, and load the generated tables. Among these, I focus on training and conversion time, the two time-consuming components in Planter's operation. Based on the results (shown in Figure 4.14), for small models (Figure 4.4) using UNSW dataset under the Anomaly Detection use case, most of the small models' training time (except SVM, NN, and AE) and all of the models' conversion time are less than $10s$, which shows Planter can rapidly prototype in-network ML.

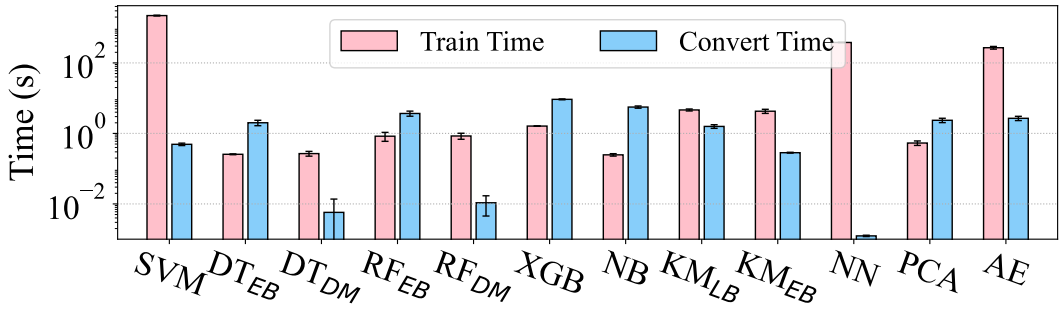


Figure 4.14: Algorithms' train & convert time (UNSW dataset).

User Experience: The efficiency of rapid prototyping was explored by checking the implementation time of an in-network ML inference prototype by two undergraduate students with no P4 knowledge and two graduates with basic P4

knowledge[3]. All test users successfully compiled and configured an in-network ML prototype on a programmable target within 10 minutes. The graduates estimated it would have taken them 2-3 months to study ML prerequisites and debug the code without Planter. These results demonstrate the clear advantage of rapid prototyping for users of all skill levels.

4.5 Discussion

ML Performance. ML models mapped by Planter provide in-network ML accuracy similar to running the same model on a host, as the evaluation shows. However, model size and inference performance present a trade-off. Sometimes, a large model can achieve higher accuracy for additional switch resources. Planter can handle this trade-off and find the optimal mapping setup using the framework frontend to fit on a switch and achieve high accuracy. Additionally, ML performance may vary across different hardware deployments depending on the low-level hardware features and restrictions.

Neural Network (NN) vs Traditional Models. In previous sections, NN is shown to be supported in Planter based on prior work [184]. Evaluation results indicate that NN is feasible for certain model sizes and hardware targets. For smaller NN sizes, there is a decrease in accuracy/F1 performance. While NN is extremely powerful, especially in training, research to date has shown that PISA-based ASIC is less suitable for NN (e.g. [169]), and this work does not challenge this claim. Instead, Planter shows that a range of other inference models are feasible and powerful.

[3]Approved by an institutional compliance team (institutional review board equivalent).

Pipeline Stages. The number of stages required by a model relates both to the type of mapped model and its size. For the UNSW dataset, at least 2 stages are consumed, and some models do not fit. Planter's encode-based solutions outperform previous direct-mapping solutions. Planter shows that stages can be shared with standard switch functionality. Some designs can be hand-crafted to reduce stages, e.g., where network and ML functions have similarities. The experience is that 2-3 stages can be saved through manual optimisation, by improving the decision logic and forcing stage allocation.

Agnostic Targets. Planter is not target-specific. It currently supports a range of P4 targets, such as Intel Tofino and Tofino 2, BMv2, P4Pi [115] using either T4P4S over DPDK or BMv2, Alevo U280 FPGA over OpenNIC Shell [216], and all P4 architectures required by these targets. Planter is open to new targets and will continuously expand its support for emerging targets, keeping in-network ML vibrant. Owing to the framework's modular design, adding more targets to Planter is straightforward, primarily involving the inclusion of scripts pertaining to the target's compiler and testing environment. Detailed guidance can be found in the Planter repository [238].

Support & Use Cases. While this chapter mainly focuses on the implementation of algorithm mappings, Planter provides one-click support for many emerging use cases. Early users of Planter have explored, for example, smart IoT gateways (e.g., P4Pir [228, 225] and FLIP4 [227]), anomaly detection (e.g., IIsy [243]), e-commerce bot detection (e.g., INCS [84]), financial market prediction (e.g., LOBIN [91]), and load balancing (e.g., QCMP [241]). The implementation and evaluation of these use cases are discussed in Chapters 5 and 7. I believe that the wide adoption of in-network ML requires a suitable framework. Planter aims to be to in-network ML what CUDA was to GPUs [143]: the enabler for wide

adoption on programmable targets, leading to a proliferation of use cases.

Benefits to Community. Planter serves as a rapid prototyping solution for in-network ML development. It empowers researchers and developers in the community to rapidly validate design ideas, conduct benchmark experiments, and evaluate the performance of the design. Apart from its primary function as a prototyping tool, Planter also serves as an educational resource, helping students understand concepts of in-network computing and gain hands-on experience. Given that in-network computing is a recent research area, Planter can play a vital role in expediting tests, validation, and standardisation process. Furthermore, it encourages diverse exploration in network and interdisciplinary domains, accelerating research and development.

4.6 Summary

In this chapter, I introduced Planter, a modular framework for rapid implementation of in-network ML algorithms. Planter's modular design enables the integration of new ML models, architectures, targets, and use cases (§4.2). Planter implements a wide range of in-network ML algorithms proposed and discussed in Chapter 3 (§4.3). The evaluation shows that Planter accurately maps trained models to a switch, can achieve high inference accuracy and line rate throughput, and can be integrated with *switch.p4* without consuming additional stages (§4.4). The evaluation indicates that the new mappings based on proposed methodologies in Chapter 3 can scale better than multiple previous works. As an open-source platform, Planter is the enabler for in-network ML research, and is available at [238].

CHAPTER 5

HYBRID IN-NETWORK MACHINE LEARNING

Offloading ML tasks becomes more practical and promising using proposed mapping methodologies and the rapid prototyping framework introduced in previous chapters. However, as shown in Chapter 4, network devices are resource-constrained, and lack support for some large in-network ML models, which limits the overall inference performance of the in-network ML system. In this chapter, I present IIsy, **I**n-network **I**nference made ea**sy**, building upon an FPGA-based early work in [218]. This chapter addresses aspects of in-network ML that were not previously discussed, such as feature extraction and model retraining. It then introduces a hybrid deployment ML model that can achieve close to optimum ML performance while still benefiting from the performance of in-network computing. The chapter is organised as follows: it first describes the hybrid system (§5.2-5.3), before moving to address the differences in mapping IIsy models (in-network ML models used for hybrid deployment) to switches (§5.4). It expands on feature extraction (§5.5) and model updates (§5.6), which are beyond the support of Planter. Finally, this section presents the implementation (§5.7) and evaluation of IIsy (§5.8), showing its ML and system performance. The content of this chapter was published in [244].

5.1 Motivation

Using in-network ML offers system performance and deployment location benefits, such as the 3-Ls (location, latency, and load) introduced in §2.4. However, these benefits would only be made possible if the inference performance of in-network ML meets the service requirements. The limited resources on pro-

grammable switches impose constraints on the size of models that can be deployed [245, 239], which affects the accuracy of its classification services. Although the mapping solutions proposed in Chapter 3 scale in-network model size beyond previous works, their scalability remains limited compared to the large models running on server-based setups. However, the performance demands for practical services, such as classification accuracy, are usually non-negotiable. This means that an in-network ML system with high classification performance is required. While updating hardware with more resources can support larger models for improved ML accuracy, this approach introduces additional costs, contradicting the original intent of in-network ML. Therefore, this section aims to design an inference system with high accuracy using restricted-size in-network ML models. Such a system, ensuring reliable performance, would allow in-network ML to meet diverse service requirements of use cases, providing tangible benefits to applications.

5.2 Hybrid Deployment

In many in-network ML models, such as random forest and XGBoost, the ML model can provide a classification with a corresponding confidence level – the probability that the classification is correct [224]. This allows IIsy to adopt the concept of a hybrid ML deployment [204], by implementing a small in-network model (e.g., by limiting the number of base models in an ensemble or using a subset of features [198]), and running a large ML model at the backend.

The small model within the network is necessarily a model where the training process can provide confidence scores for classification outputs, e.g. tree-based ensembles. The large model at the backend operates independently from the

small model, and they do not need to be of the same type. The selection of the large model at the end-point primarily considers the processing ability of the backend system, as well as the ML performance of the large model. Advanced models, such as deep neural networks (DNNs) [83], and sophisticated training techniques, like adversarial training [69], can be utilised.

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

Previous pure ML works [204] have shown that most of the queries in a given dataset can be classified by a small ML model with a high confidence level. Hence, a hybrid ML deployment reduces both the classification latency and the load on the backend servers (by forwarding only "hard-to-inference" queries for re-classification), as compared to a monolithic ML model deployment at the backend, where all queries are processed by the end-point. In §5.8, I validate these assumptions and demonstrate the benefits using two use cases from different domains, cyber-security, and finance.

5.3 IIsy Architecture

The architecture of IIsy, shown in Figure 5.1, has four components: ML training, a mapping tool from a trained model to a target network device, a data plane implementation on a hardware target, and a control plane component for populating table entries. Additionally, the figure shows the integration with a server backend, for hybrid deployments.

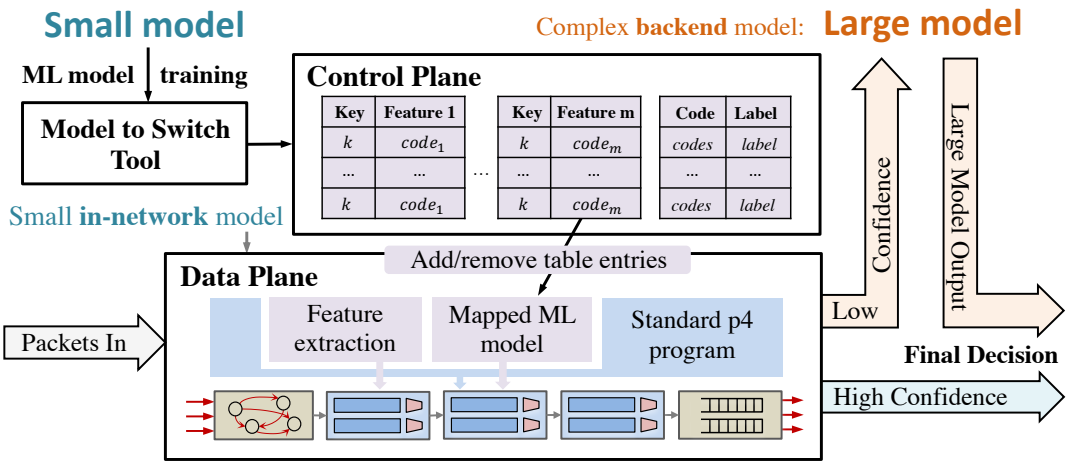


Figure 5.1: The high-level architecture of IIsy.

IIsy uses host-based ML training based on standard ML frameworks and allows for model updates over time. For the small in-network ML model, as shown in Figure 5.1, IIsy modifies Planter as the mapping tool (§5.4). It takes the output of the ML framework, the trained model, and maps it to a switch-ASIC target. The tool generates two components: an implementation of the network switch data plane (P4 based), and the table entries loaded by the control plane. The data plane components combine standard network switch functionality (provided by the user), a more practical feature extraction process from IIsy, and the in-network ML code generated by the modified mapping tool. The control plane component is responsible for the configurations and updates of the network device. Beyond Planter, this control plane also supports runtime updates on the

deployed classification model. As such, it combines the standard user-defined control plane and the table configurations generated by the mapping tool and required for in-network ML. IIsy's support for hybrid deployment means that an additional data plane component is needed, which considers the confidence level of a classified transaction, and accordingly decides if to forward the transaction to its destination (high confidence) or route it to the backend for classification by the large model (low confidence). The confidence threshold is configurable, and confidence levels are programmed through the control plane. The type of the model on the switch and at the backend do not need to be identical, e.g., XGBoost on the switch and a neural network at the backend.

5.4 Mapping Models to Switches

The implementation challenges of in-network ML models have been mostly mitigated by the algorithm mappings and framework introduced in the previous chapters, which is the basis of IIsy's algorithm mapping. For the small in-network model on programmable network devices, IIsy uses ensemble tree models. Ensemble methods improve ML prediction results by combining multiple learning models [253], and most importantly are also able to output the confidence of each classification decision for hybrid deployment. The decision confidence of an ensemble model is typically calculated using one of the following two methods: mean approach, which involves averaging the confidence scores of selected output labels across all models, or percentage approach, which represents the confidence as the proportion of models with the same output label. In this chapter, IIsy applies the mean approach.

IIsy considers two tree-based types of ensemble methods: bagging (random

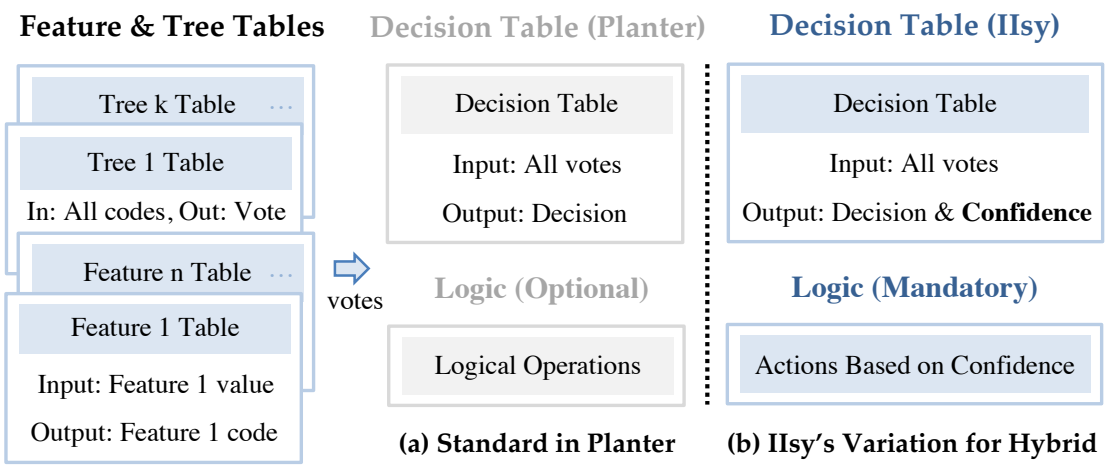


Figure 5.2: The variation of the ensemble tree model used in IIsy (b) compared to the standard version of algorithm mapping in Figure 3.8 using Planter (a).

forest) and boosting (XGBoost). Based on the evaluation result in §4.4.2, IIsy applies encode-based mapping methodology to map them. Encode-based ensemble tree models require three logical pipeline stages: the first stage is used for features lookup; the second stage conducts the independent votes lookup of all the models in parallel; and the third stage makes a classification decision based on the votes of the ensemble, as shown in Figure 5.2 (a). While this process provides the classification result, to get the confidence of each classification decision in-network for hybrid deployment, IIsy changes the mapping detail. As shown in Figure 5.2 (b), to include confidence output, the decision table will map the vote of all trees into not only a class but also a confidence level. This value can be stored together with the class or using a separate action data field. Since action data has a small impact on memory, as shown in Figure 4.10 in §4.4.3, IIsy applies two fields to store them. The action in the final logic for the small in-network ML model is not fixed and is usually tailored based on the use case. One typical option is to forward all the low-confidence traffic to the complex model at the back end. More detailed final logic configurations for use cases can be found in §5.8.2.

5.5 Feature Extraction

Network devices are designed to extract headers from packets. However, the research community has already gone beyond packet headers for applications ranging from telemetry [110] to in-network classification [33]. While previous chapters have incorporated certain feature extraction, their primary emphasis lies in algorithmic mapping and the implementation of in-network ML models. The feature extraction supported therein is limited to fundamental per-packet features. In this section, I further discuss how features can be extracted from data on different levels of granularity. The implementation and evaluation of feature extraction processes on specific use cases are discussed later in §5.8.3-5.8.5.

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

Flow level features. Flow-level features, such as flow size and flow duration, are stateful. Information is collected and stored across multiple packets [188]. IIsy supports two types of flow-level features: counted features (e.g., flow size, packets count), and time-related features (e.g., flow's start time, inter-packet gap).

Aggregate level features. Aggregate level features consider a group of flows, the aggregation of traffic (e.g., from/to port X) or the network as a whole. Examples of features useful for ML purposes include traffic volume from a group of subnets, inter-arrival time toward a specific application or a histogram of source and destination ports [174]. Implementing aggregate-level features is mostly similar to flow-level features, however, additional operations may be required, such as

mapping flow identifiers to an aggregated-feature identifier.

File level features. Supporting feature extraction from files can be categorised into two scenarios. In the first scenario, a single packet per file, all necessary features can be found within each packet. The extraction process for this type of file resembles that of packet-level features. It is relatively intuitive and can be realised by parsing the header and examining the payload. The second scenario involves files transmitted across multiple packets, which is more complex than in the previous scenario [24]. In this case, it necessitates not only verifying the packet sequence but also identifying the files' start and end points. Additionally, if the packet size exceeds the capacity of the programmable data plane's bus, it may require recirculation (dependent on the target) to accommodate the packets pertaining to the file.

5.6 Retraining and Updates

ML models often need to be retrained, e.g., due to data skew, and the resulting classification model needs to be updated. This section considers such updates and realises the process by only updating table entries, without changes to the deployed program. Specifically, the generated P4 program (ensemble trees) depends on a set of user definitions: the features that need to be extracted (not necessarily used by the ML model), the type of the model, and constraints on the model (e.g., number of trees). While retraining will result in a different ML model, as long as the definitions above are kept the P4 program will not change. Changes will only happen to the action field in the features tables, code-to-classification tables, and decision table (as in Figure 3.8), which manifest as table entries changes rather than changes to the P4 code. These can be loaded

through table updates, a common management operation. During updating entries to the table, packets may be falsely classified due to the incomplete table entries. A shadow runtime update mechanism is designed to mitigate this effect and is further discussed in this thesis through a case study in §7.2.1. Changes to some hyperparameters require generating new P4 code and are not supported in runtime. In a hybrid deployment, traffic can be directed to the backend during updates, to avoid misclassification. Data for retraining can be collected through sampling and using in-network telemetry [110] and will be affected by the location of a switch (e.g., edge vs data centre).

5.7 Implementation

The IIsy's ML algorithms are trained offline on a server [237] using established frameworks like *scikit-learn* [154]. IIsy automatically maps trained classification models to programmable network devices, and in particular to off-the-shelf switch ASIC, for hybrid deployment. To further extend its reusability, this process is implanted in Planter (Chapter 4) as a variation module of the in-network ML. This IIsy variation uses a similar model training process as in Planter but applies a modified mapping process to include confidence prediction in its output to data plane code and control plane code. The switch implementation of the mapped ML model mainly runs on Intel's Barefoot Tofino (ASIC) in this chapter for brevity. On Tofino, packet-level, flow and aggregate features are supported, with a further focus on files. Data is extracted from text files, both with fixed and unknown feature sizes (e.g., words separated by delimiters). The prototype supports features of up to 30 ASCII (32-bit) and 59 numerical characters (restricted by pipeline hardware resources). In addition, it supports features split

between packets and features implemented deep within the packet (§5.8.2). The data plane and the control plane are auto-generated, using Python scripts and a configuration file. A user defines in a configuration file design constraints, such as the maximum number of trees, and the tool takes the output of the training stage (pickle file) and uses it to generate both the data plane (P4 files) and the control plane (table entries). Further information is provided in [244].

5.8 Evaluation

This section first introduces the evaluation setup (§5.8.1), two used use cases (§5.8.2), and their feature extraction processes (§5.8.3). The evaluation starts with the performance of small in-network ML models (§5.8.4). Then, this section presents the limitation of model scalability and shows the benefits of a hybrid deployment (§5.8.5). Finally, this section evaluates the inference accuracy and system performance of hybrid deployment for both use cases (§5.8.6).

5.8.1 Testbed Setup

The training of the IIsy's models uses Python with packet scikit-learn 0.24.1 and XGBoost 1.3.3, running over a `c4.8xlarge` AWS EC2 instance with 36 vCPUs and 60 GB RAM running Ubuntu 16.04 LTS. The system test environment uses $64 \times 100G$ ports Barefoot Tofino, APS-Networks BF6064X. Four servers with 100G NVIDIA ConnectX-5 NICs are used to send and receive traffic from the switch. To test full throughput, I use a snake configuration, where traffic is looped from each port to the following one, enabling traffic across all 64 ports, which is a common practice [51]. As a baseline, I measure 6.4Tbps on the switch when

running simple forwarding.

5.8.2 Use Cases

This evaluation is driven by two use cases: network anomaly detection using the UNSW-NB15 dataset [140], and time-sensitive financial market prediction using the Jane Street Market Prediction dataset [76].

I. Anomaly detection - Reducing backend resource consumption

Anomaly detection, such as intrusion detection and prevention, is typically done at the backend and can consume significant compute or acceleration resources [235]. All network traffic toward certain application servers needs to be examined, and malicious traffic needs to be filtered. Our goal is to provide a scalable solution, whereby normal traffic is admitted by the switch, and anomaly traffic is either dropped in the switch or sent to the backend (in a hybrid mode). In the hybrid mode evaluation, any traffic that is classified as anomalous or with low confidence is sent to the backend for deeper inspection. In this manner, the switch does not block (drop) legitimate traffic and offloads significant processing from the backend, as most traffic is normal. The dataset used, UNSW-NB15 [140], contains a mix of normal traffic and different types of attacks. This use case is focused on *load reduction* benefits, where in-network ML saves resources compared with host-based solutions while also scaling with the network's bandwidth.

From the ML perspective, random forest is suitable for this use case [158], as it offers low variance in its classifications. This leads to a more predictable fraction of the traffic that is correctly classified as normal (unless the traffic distribution

changes dramatically – which requires retraining the model). Other ML models are evaluated for feasibility purposes. The ML training uses 80% of the data and the rest 20% is used for testing. The model running on the backend is using a random forest of 200 trees (estimators) and 10,000 leaf nodes (at most), and all the features in the dataset.

II. Financial market prediction - Reducing latency

Low latency financial transactions, such as algorithmic trading, are very sensitive to latency. For top 10% financial traders, the decision latency is less than 42 microseconds [20] from a passive order to an active transaction. In algorithmic trading, a data feed from the stock market provides live information using an unencrypted protocol, such as NASDAQ ITCH [142]. Typically, a large backend is used to provide real-time classification for all market transactions. In this use case, the switch can identify and tag high-priority transactions, while other transactions are sent to the backend for fine-grain classification. The tagged high-priority and high-confidence transactions can be forwarded to a different server for immediate execution. Moreover, tagged queries can be prioritised over dedicated link(s), avoiding congestion. Assigning time-sensitive high-priority and high-confidence transactions to a special fast processing path may bring significant financial benefits with low resource consumption. Any misclassified high-priority transactions will simply undergo the regular classification path. This is an example of *latency* benefits for time-sensitive applications, while the change to the backend's load is small. To demonstrate this use case, the Jane Street Market Prediction dataset [76] is used. Each entry in the dataset contains 130 anonymised features, representing real market data, and two output values ('weight' and 'resp') representing the trade's return. Using these two output val-

ues, we label the transactions by recommended actions: 'Strong sell or buy', and 'Sell/Hold/Buy'. The transactions are typically a feed of individual trade instructions from the stock exchange. The Jane Street dataset is recent and open information available from a trading company, presenting pre-processed transactions. All incoming transactions are assumed to go through the switch, so any classification by the switch has an additive latency of close to zero[1]. In this evaluation, for clarity, all packets with a low confidence level will be forwarded to the backend and all packets with a high confidence level will be forwarded to the special fast processing path.

In terms of ML performance, while I evaluate with different models, the preference for this use case is XGBoost, commonly used in financial applications as boosting offers a controlled bias that is more suitable for identifying minorities. The ML model in this use case is trained using 80% of the dataset and the rest 20% is used for testing. The model running on the backend is using all 130 features, with XGBoost of 100 trees (estimators) and a maximum depth of 8 (XGBoost trees tend to be shallow).

5.8.3 Feature Extraction

The IIsy for the anomaly detection use case supports packet-level features (e.g., source and destination port, protocol, service, and ports equivalence) and flow-level features (e.g., duration, flow size in bytes, and packets in each direction). Flow level features sometimes improve the quality of the prediction, but cost two stages: to hash the flow ID, and to update a register holding the value of the feature (e.g., flow size). Choosing between the two options requires weighting

[1]Use of L1 switches, e.g., Cisco Nexus 3550 is a different scenario.

also other considerations, such as if flow ID is needed for "standard" networking purposes. Our resource consumption evaluation (Table 5.1) uses source and destination port, protocol, and service features, and the study of hybrid deployment (Table 5.3) uses in addition the feature is_sm_ips_ports (is same source and destination?) and the stateful feature "source bytes" (sbytes). The effect of flow-level features on accuracy is shown in Table 5.1, where the feature "service" is swapped with "sbytes". As shown in the accuracy (Acc.) of "sbytes" row, the application of this flow level feature can improve most of the average model accuracy from 83.16% to 92.83% and reduce the standard deviation among all models' inference performance from 15.1 to 4.8.

The Jane Street dataset contains 130 numerical features, which I evaluate twice: using packets containing numerical values, and in a csv format. For ease of exploration, the csv file is reformatted as columns of eight characters, though other IIsy implementations are not of fixed size or known delimiter location. The features ranked most important and used are features number 42, 43, 45, 124, and 126. Both numerical and csv format processing use these features, thereby demonstrating feature extraction from deep within the packet (using the parser), successfully extracting without recirculation. As financial transactions are typically a feed of individual trade instructions (§5.8.2), and the size of an entry in the Jane Street dataset, with 130 columns, barely fits within an maximum transmission unit (MTU) packet (1522B), each transaction is sent individually.

5.8.4 Performance of Small Model

Table 5.1 and Table 5.2 summarise the resource consumption of anomaly detection and financial transactions, respectively. The tables show, for each model, the

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

Table 5.1: Anomaly Detection - Latency on Tofino relative to *switch.p*4 reference program. The row named accuracy (Acc.) sbytes shows the IIsy performance of using flow-level features "sbytes".

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

Table 5.2: Financial Transactions - Latency on Tofino relative to *switch.p*4 reference program.

size of the model (not maximum size) that fits within Tofino's ingress pipeline using the features noted above. The ensemble models use a small model of 6 trees with a depth of 4. In both tables, the memory indicates the proportion of overall utilisation, while the latency is assessed relative to Tofino's *switch.p*4 reference design. As the results show, the memory requirements are low in comparison with *switch.p*4. For anomaly detection, all the models consume less than 9.3% of the memory, with decision tree (DT) and random forest (RF) requiring less than 1.9%. In the financial use case, all the models require less than 6.7% of memory. This result shows that small size in-network ML models can fit in commodity programmable network devices well.

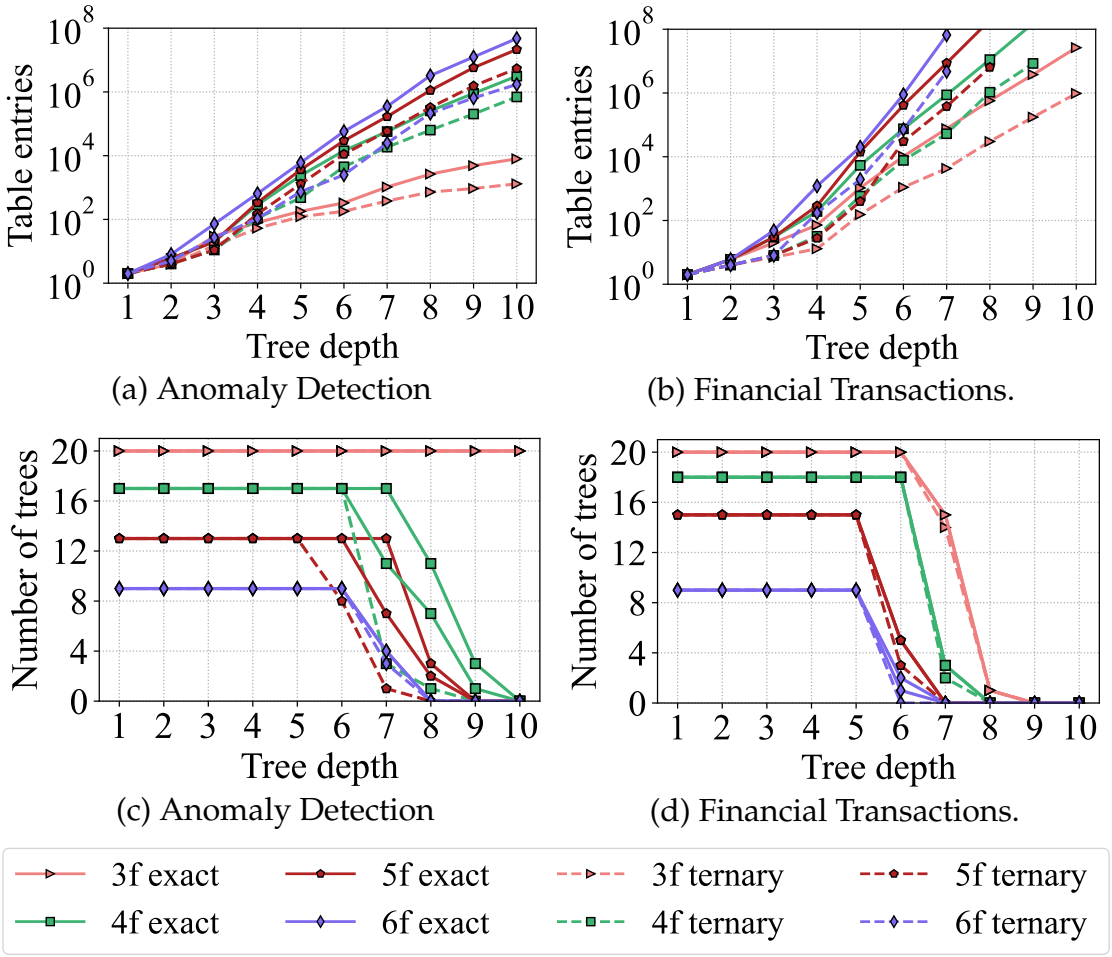


Figure 5.3: Ensemble scaling of table entries (a,b) and maximum number of trees (c,d) with tree depth and features. *f* refers to the number of used features.

5.8.5 Model Scalability

The size of a model fitting within a switch depends on the type of model, model hyperparameters, mapping configurations, the dataset, and its features. The influence of these parameters on the table size and consumption of pipeline stages for each in-network ML model is illustrated in Figures 4.8-4.9 in Chapter 4. This section presents a more detailed scalability result of in-network ML models. Specifically, Figures 5.3 (a) & (b) show how memory requirements of a decision tree scale with the number of features and the depth of the tree, using exact match or ternary feature tables. In the finance use case, all the features are

similar, and adding features increases memory requirements in a roughly consistent manner. In the anomaly detection use case, features vary significantly in their memory requirement. For example, the protocol type requires significantly fewer entries than the source or destination port. Consequently, the anomaly detection use case requires less memory than the finance use case for the same model size. As Figures 5.3 (c) & (d) show, using up to 6 features, one can fit up to 20 trees. Increasing tree depth means that fewer trees can fit within the switch, due to the size of the decision table.

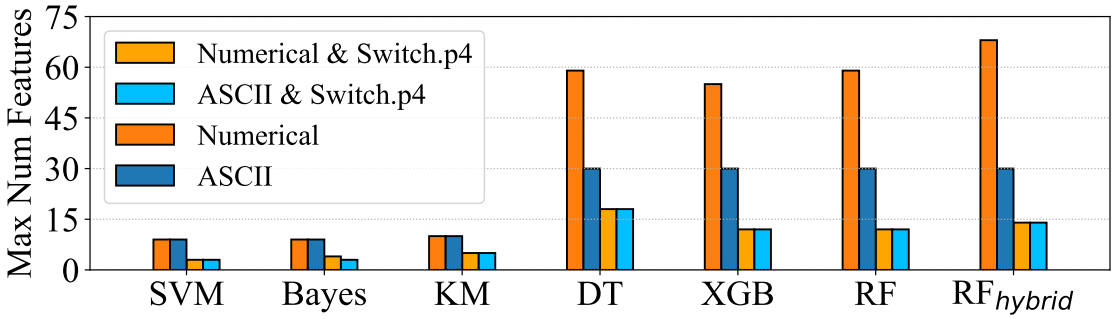


Figure 5.4: The maximum number of features that can fit on a switch in the financial transactions use case.

To explore the maximum number of features that can be supported, four types of implementations are evaluated: in-network ML using numerical features, in-network ML using ASCII (from `csv`) features, in-network ML integrated with *switch.p4* and using numerical features, and in-network ML integrated with *switch.p4* and using ASCII features. This is applied to the financial use case, supporting both types of features. Figure 5.4 shows the maximum number of features feasible under the four variations. Tree-based models can fit more features compared to classic models due to stage sharing. For example, decision tree and random forest can fit up to 59 numerical features or 30 ASCII features (due to PHV size), while XGBoost fits 55 numerical features and 30 ASCII features. Classical models support 9-10 numerical or ASCII features and are limited

by the number of stages required for logical operations. The integration with *switch.p4* limits the resources available for feature tables, leading to 12-18 features allowed for tree-based models. From this result, using the maximum number of supported features as an example, we can see that even though in-network ML can scale, it is still small compared to the large mode on the server (almost unlimited number of features), and at a cost of high resource consumption in the data plane.

5.8.6 Hybrid Performance

This section explores IIsy's ML performance, with a focus on ensemble models in a hybrid deployment. Although SVM and NB achieve an accuracy of 0.88–0.92, this is as the anomaly detection dataset is biased, with most of the traffic benign (which has a macro average F1 score of 0.48–0.51). Using ensemble tree models, I correctly identify anomalies. The baseline for ML performance comparison is the full ensemble model running on backend servers. I implement on the switch a small model (Table 5.3), that classifies a subset of the traffic, and forwards to the backend all low-confidence or anomalous traffic. A confidence level is set in the switch to determine the threshold for on-switch classification.

Table 5.3 explores the effect of ensemble size on ML performance, showing both native switch deployment and hybrid deployment. The results are compared to fully-grown ensemble models running on a backend (§5.8.2). As the table shows, the size of a model has a limited effect on its ML performance (except for small RF in anomaly detection), and negligible effect in a hybrid deployment. Furthermore, the results of the hybrid deployment are almost identical to the baseline, showing that a hybrid deployment using a small model on the

Anomaly Detection, Random Forest, confidence threshold 0.7				
	Small	Medium	Large	Baseline
Features	4	5	6	25
Trees	6	10	14	200
Max Depth	4	5	6	—
Accuracy	97.05	97.17	97.78	99.51
Precision	98.06	98.12	98.60	99.67
Recall	88.55	89.04	91.36	99.75
F1 score	92.60	92.94	94.58	98.88
Hybrid Accuracy	98.58	98.94	99.31	—
Hybrid F1	96.64	97.53	98.41	—
Financial Market Prediction, XGBoost, confidence threshold 0.7				
Features	4	5	6	130
Trees	6	10	14	200
Max Depth	4	5	6	–
Accuracy	72.48	72.65	73.73	77.34
Precision	68.48	68.76	70.05	74.43
Recall	66.51	65.69	68.09	72.76
F1 score	67.16	65.51	68.78	73.43
Hybrid Accuracy	77.31	77.30	77.26	—
Hybrid F1	73.41	73.43	73.40	—

Table 5.3: Scalability and ML performance of ensemble models.

switch allows for high ML performance and little resources consumption.

Figure 5.5 (a, c, e) shows the hybrid deployment in the anomaly detection use case using random forest. The evaluation includes the accuracy of the switch and server, the fraction of traffic offloaded by the switch (switch fraction), and the corresponding misclassification rate, as a function of the switch classification confidence threshold. The baseline result is a misclassification rate of 0.49% and

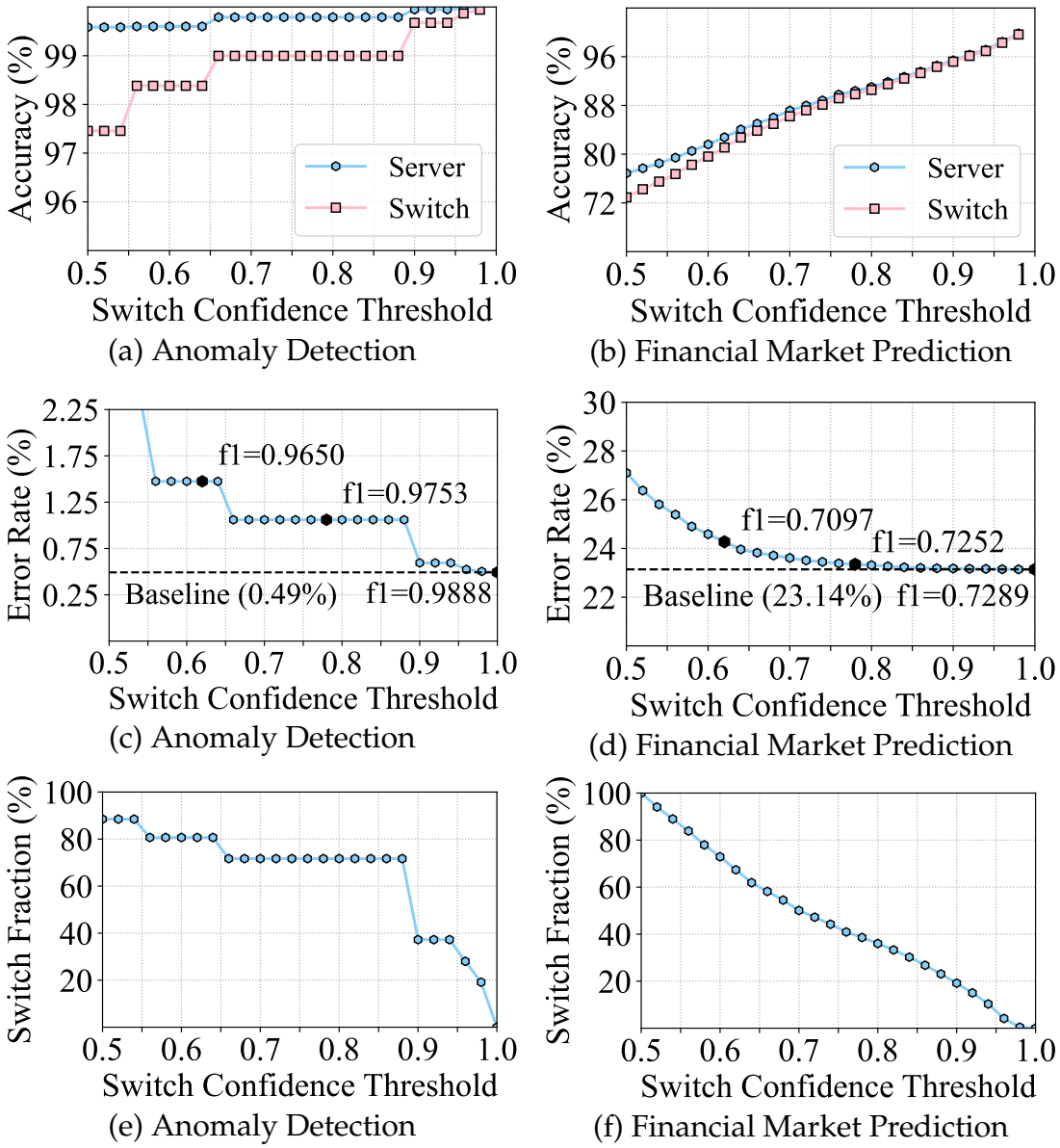


Figure 5.5: Anomaly Detection (Random Forest) and Financial Market Prediction (XGBoost) in a hybrid deployment - accuracy, error rate, and fraction of traffic handled by the switch.

an F1 score of 0.9888. In comparison, with a confidence threshold of 0.7, meaning only traffic with classification confidence lower than 0.7 will be forwarded to the backend, 71.7% of the traffic is handled by the switch, achieving a misclassification rate of 1.35% and F1 score of 0.975. Meanwhile, the inference performance of this hybrid system improves as the confidence threshold increases, but the

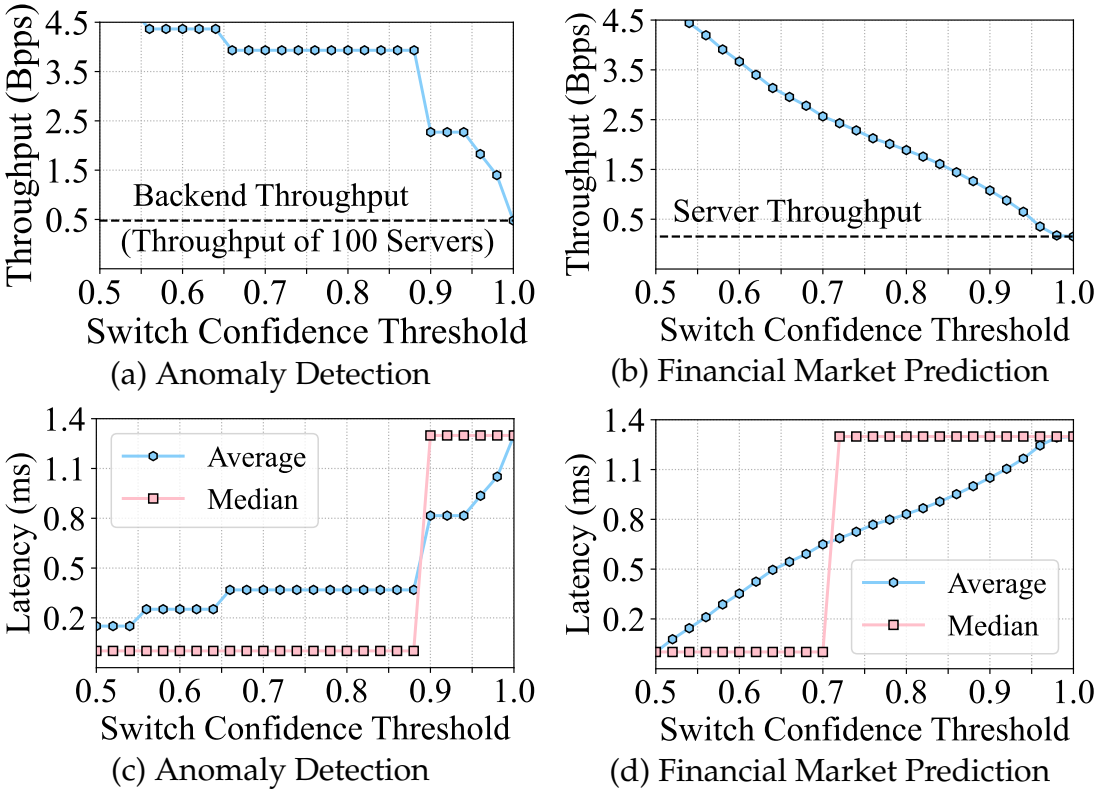


Figure 5.6: Anomaly Detection (Random Forest) and Financial Market Prediction (XGBoost) in a hybrid deployment - throughput and latency

fraction of traffic handled by the switch decreases. For the same scenario, Figure 5.6 (a, c) shows the throughput and latency of hybrid deployment, where the throughput follows the fraction of offloaded while latency is the opposite. Switch's and server's performance match the results in §4.4.4, and the backend uses 100 servers.

Figure 5.5 (b, d, f) presents the effect of confidence threshold on the fraction of traffic offloaded by the switch and ML performance of financial market prediction. Figure 5.5 (b) shows the error rate for classifications done by the switch compared with the error rate for the same transactions if done by the host. As the graph shows, transactions that have low confidence (below 0.8) on the switch, are less likely to be misclassified by the full-grown model running on the server. In fact, starting 0.8 confidence threshold (where 36.01% of deci-

sions are being served by the switch model) the error rate difference between the server and the switch is very small, and some traders may even find that the error difference of 0.7 is still small enough to provide higher transactions rate for 50.07% of the transactions. For the same use case, Figure 5.5 (d) shows that the baseline achieves an error rate of 0.231. In comparison, the hybrid model achieves an error rate of 0.271 with a confidence threshold of 0.5. Increasing the confidence level to 0.7 reduces the error rate to 0.236. However, there is a trade-off here, shown in Figure 5.5 (f): with a confidence threshold of 0.6, 72.91% of transactions are classified by the switch, whereas at 0.7 confidence, 50.07% of the transactions are classified by the switch. Figure 5.6 (b, d) shows the throughput and latency of hybrid deployment in financial market prediction. This use case is under the same setup as in anomaly detection. As shown in these figures, the trend of throughput and latency versus confidence threshold in financial market prediction is similar to that in anomaly detection. When confidence threshold is 0.7, the hybrid deployment can reduce the median latency by 50% and offload more than 50% load from backend servers to the switch.

5.9 Discussion

Scope. This chapter does not focus on the methodology of mapping trained ML models to network devices (covered in Chapter 3). This chapter does not seek to improve the quality of training ML models, nor to contribute to a specific use case. Applying the methodology to certain applications, such as congestion control, is beyond the scope of the chapter. My choice of ensemble models is primarily as they provide confidence level and have the best results for the example use cases.

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

Model Update. To address data drift and the dynamic network environment, supported models can be updated while ensuring uninterrupted normal traffic flow through the control plane. We further address the effect of ML model updates in §7.2.1 [226].

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

5.10 Summary

This chapter introduced a hybrid in-network ML system used for in-network classification, which enables similar inference performance as a server-based large model while benefiting from the system performance of the small in-network model (§5.2). In the hybrid in-network ML system, a large ML algorithm is deployed at the backend to assist the small ensemble tree model on programmable network devices (§5.3). In the small in-network ML model, a confidence level is employed to determine whether assistance from the large ML model at the backend is needed (§5.4). Meanwhile, to enhance the applicability of in-network ML systems in practical scenarios, this chapter also supports complex feature extraction (§5.5) and runtime model update (§5.6) mechanism for the small in-network model, beyond algorithm mappings and implementation in previous chapters. IIsy is implemented on Tofino (§5.7) and evaluated on two use cases (§5.8). Evaluation results show that IIsy reduces the load on the backend, and achieves high throughput, low latency, and near-optimal classification results while coexisting with standard switch functionality. These performance benefits make IIsy an important step toward the practical use of in-network ML.

CHAPTER 6

DISTRIBUTED IN-NETWORK COMPUTING

In the previous chapter, I demonstrated that the hybrid deployment system can assist in-network ML to provide services with high inference and system performance. While hybrid in-network ML is promising and practical, its ML performance can still be affected by the size of the small in-network model. IIsy did not address the scalability of these models. Even though Planter in Chapter 4 can generate efficient mappings, in-network ML still faces constraints on the size of each model based on the resources of programmable network devices. In this chapter, I aim to overcome this limitation, and further scale in-network ML algorithms by realising a distributed in-network computing (DINC) framework. DINC disaggregates a large in-network computing program into segments and deploys them on multiple network devices in a distributed manner (§6.3-6.4). Its planner supports any-to-any routing and can distribute and deploy program segments across network devices while providing full & correct functionality (§6.5). Information sharing between segments or nodes is transparent to users (§6.6). To streamline the process, DINC is designed to be a scalable, flexible, and easy-to-deploy framework (§6.7). The introduction of DINC allows the direct deployment of large-scale computing (including in-network ML) algorithms on programmable network devices (§6.8-6.9). The content of this chapter was published in [242].

6.1 Introduction

Scaling in-network services is hard, as programmable network devices are intended for high-efficiency packet processing, with limited resources (e.g., mem-

ory, operations, and stages) compared with CPUs. One approach is to optimise algorithms' design for a single-device deployment, yet resource constraints remain a limitation. An alternative solution is moving to *distributed* in-network computing, jointly utilising resources of programmable network devices.

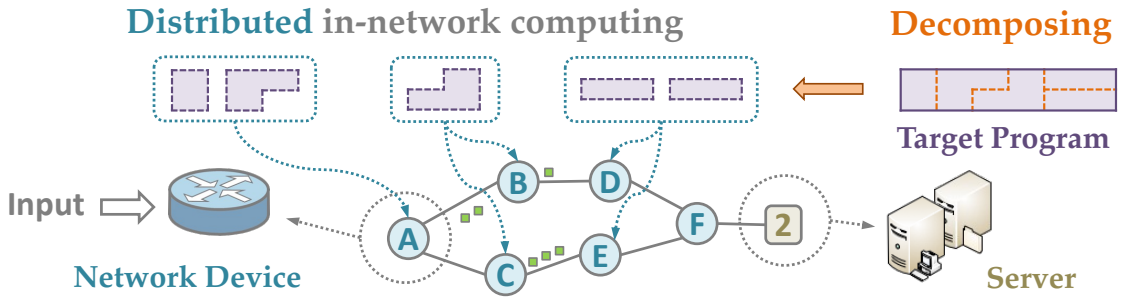


Figure 6.1: Illustration of distributed in-network computing paradigm.

Distributed in-network computing raises multiple implementation challenges, especially where resource-heavy applications are considered, such as large ML models. Figure 6.1 illustrates the challenges: (1) **Decomposing** a single program into multiple *segments*. (2) **Distributing** the program's segments across multiple devices without affecting the correctness of its functionality. (3) Satisfying the program's and network's set of **constraints**, such as latency and resource constraints. (4) Providing the program's functionality for any set of paths within the network without routing rule changes. This last challenge is possibly the hardest, as in a network, packets may travel from any node to any node, and operators may use different routing optimisation methods. In a software-defined network, a controller has a centralised view of the network, including device information, and potential routes (including multi-path routes) through the network. Building upon this information, a controller can be designed to provide a joint resource provisioning plan for distributed in-network computing. It can utilise unused network resources, splitting a single in-network computing service across several devices.

Network service chains provide heuristic solutions for segment placement and distributed planning [208]. However, there are intrinsic and significant differences between it and distributed in-network computing, both in terms of the type of devices used and their location. Unlike CPU- and GPU-based service chains, deployed at the server level, in-network computing service segments are deployed physically within the network, on switch-ASIC [30], FPGA [95], or NICs [201]. The architecture, resource constraints, communication models and performance requirements are inherently different to traditional service chains (see §6.2). Preliminary efforts toward distributed in-network computing [127, 39, 193, 40] focused on the distribution problem. Some of these works [39, 40] distributed multiple programs, rather than slicing a single program, or partitioning M/A tables using manual directives [127]. Flightplan [193] was the only one to disaggregate and place a distributed program, on rack/pod level. Yet little effort has attended to the challenge of routing through a large network, from any node to any node, without routing rules modifications or at the scale of a WAN. This chapter aims to bridge this gap and provide such distributed in-network computing services to further improve the performance of in-network computing services.

6.2 Challenges

To better understand the challenges in distributed in-network computing, I start with an exploration of available resources in the network. Programmable network devices are resource-constrained due to their performance-driven design logic [87]. For example, the Intel Tofino switch [78] can guarantee Tbps-scale throughput but has only 12 processing stages and around tens of Mb memory.

These resource constraints are a main challenge for offloading applications to the network. Still, in-network computing prototypes (e.g., caching [103], aggregation [171, 116], ML inference [218, 245], resource scheduling [27], and consensus [51]) demonstrated real-time processing, offloading computing tasks from servers and achieving high performance.

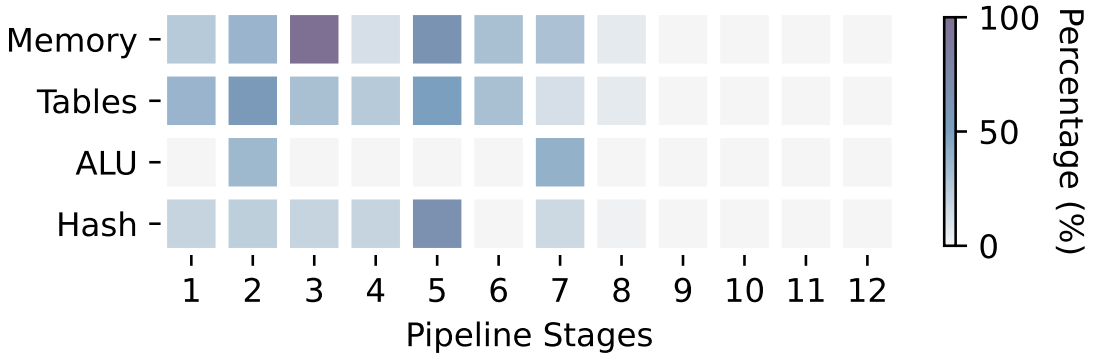
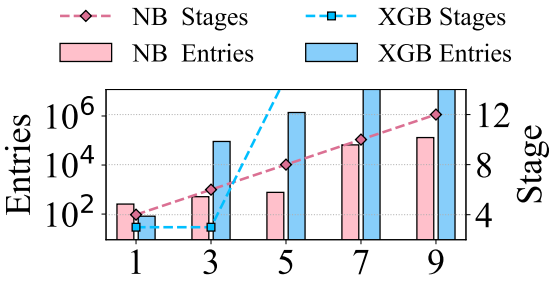


Figure 6.2: Resource consumption of a basic functionality RARE Router on Intel Tofino.

While the performance on a single device is high, resource contention between in-network computing applications and network functionality is a barrier for adoption. To illustrate the challenge, Figure 6.2 shows the resource consumption of the RARE open source router [130] when deployed on Tofino. As shown in Figure 6.2, the router consumes more than half of the pipeline stages with only basic functionality. Using more features of the RARE router exhausts and even exceeds Tofino's resources. Deploying an in-network computing program on the same device as RARE, is hard or impossible without stage sharing. Figure 6.3(a) shows the feasibility of deploying example ML models (using Planter [245]) on Tofino, standalone and coexisting with RARE. As shown in the figure, most in-network ML models are able to be standalone deployed, but will run out of resources when coexisting with other network functions. Even for standalone deployment models, as a model's size increases to improve its performance, resource consumption increases too, as shown in Figure 6.3(b).

Types	SVM	NB	RF	XGB
Stage	9	8	6	15
Entries	0.1M	0.1M	0.3M	1.4M
Alone	✓	✓	✓	✗
Coexist	✗	✗	✗	✗

(a) Alone vs Coexist w/ RARE



(b) Parameter vs Resource

Figure 6.3: ML models resource consumption and scaling with model hyper-parameters changes. Hyper-parameter is depth for XGB and features for NB.

Consider the case where different ML models, as shown in Figure 6.3, are used to implement a cyber-security service, detecting and dropping malicious traffic. In a WAN, such malicious traffic may come from any user, pass through any switch, and go to any destination. The cyber-security service will need to be deployed in a manner that guarantees that no matter the path taken through the network, malicious packets will be detected and dropped.

To support *both* normal network functionality, and the ML-based service, distributing programs across several devices is needed. However, it is not easy to distribute an in-network application across multiple devices, for the following reasons: 1. There is no agreed model of the network used for distributed in-network computing. Intuitively, in-network computing should not affect existing network functions, nor the routing rules used to forward packets. 2. After program segmentation, there are parameters shared across segments of the application, and the data passing model is undefined. 3. Application splitting and coexistence with network functions is error-prone, as well as guaranteeing segments' execution order on multiple nodes. To address these challenges, this work explores efficient methods for deploying in-network computing services in a distributed manner.

6.3 The Concept of DINC

To explain how in-network computing can be distributed and deployed, I first discuss common properties of algorithms using an example in-network computing application, and then demonstrate the many-paths nature of the network using a sample network topology.

6.3.1 In-network Computing Example

In-network computing provides application functionality by mapping computation tasks to programmable data planes on network devices. These data planes use match-action pipelines, where values are looked up in a table, and the result of the lookup is an action. Typically, every match-action pair consumes a processing stage within the pipeline. Sequential dependencies between operations lead to a series of stages used on the device, with metadata used to pass shared information between stages (metadata is stored in a PHV [30], which is initialised per packet). Despite the high performance of existing work [103, 23, 245, 51, 91, 227, 226, 116, 241], resource limitations remain a constraint. As demonstrated in Figure 6.3(b), scaling up computing complexity leads to exhaustion of resources. I use naïve Bayes as an example of a classical ML classification algorithm that uses typical in-network ML mapping and faces resource constraints.

Equation 6.1 shows the naïve Bayes mapping used by §3.5.1 [245]. Different from traditional naïve Bayes, the log(#) operation converts multiplication into addition (as multiplication is not supported on a switch). The map(#) operation

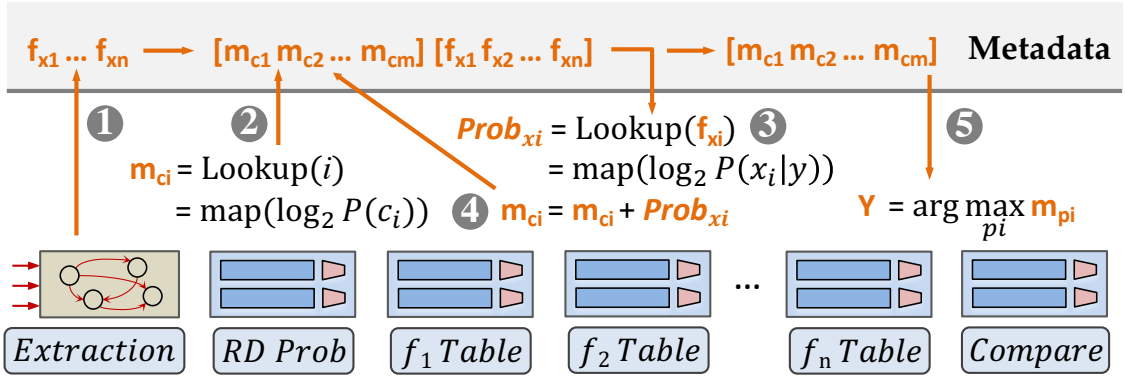


Figure 6.4: Data plane implementation of Bayes based on [245] and Equation 6.1.

ensures that used intermediate values are covered with minimal accuracy loss.

$$\hat{y} = \arg\max_{y}[\text{map}(\log_2 P(y)) + \sum_{i=1}^{n} \text{map}(\log_2 P(x_i \mid y))] \tag{6.1}$$

Figure 6.4 shows the data plane realization of the above equation. Step ❶ shows the extraction of n features (fields) from the packet header, and storing them in metadata fields as f_{x1} to f_{xn}. Next, the probability of each class map($\log_2 P(y)$) is read from a table called *read probability (RD Prob)* into metadata fields as m_{c1} to m_{cm} (given a classification problem with m classes) in Step ❷. For every input feature i, Steps ❸ and ❹, look up an intermediate value map($\log_2 P(f_{xi} \mid cj)$) and add it to its respective class j. The prediction probability of all m classes will be m_{c1} to m_{cm}, and the final pipeline stage (marked *compare*) finds the class with the maximum probability through comparison, and sets it as the output label (Step ❺).

While mapping details vary between applications, this example is representative in terms of its common stage-based structure and metadata passing. The distributed deployment strategy of this example on a sample network topology, and the challenges it faces, will be explained in Section 6.3.3.

6.3.2 Network Scenario

In this chapter, I consider a distributed in-network computing deployment scenario in a network with many-paths, using multiple ingress and egress nodes. Data can originate from any ingress node and terminate at one or more egress destinations. Beyond this any-to-any or many-to-many connectivity, techniques such as load balancing or routing redundancy mean packets from the same service may be routed through multiple paths, for a given source-destination pair.

Distributed in-network computing should not affect existing network protocols (e.g., IS-IS) nor services (e.g., load balancing). Given a deployment scenario and underlying routing rules, distributed in-network computing should find best-effort service deployment within given network constraints, without changes to routing rules. In this manner, packets from any source node should be fully processed before reaching their destination along any possible path.

6.3.3 Distributed Deployment Example

DINC enables the deployment of large in-network computing programs, as its framework is able to slice a program into segments and deploy these segments within network devices given resource constraints (e.g., memory, operations, stages). Figure 6.5 shows a sample deployment of a Bayes classification algorithm (explained in Figure 6.4) using three-features on a Folded-Clos topology (assuming all network devices are programmable). Two inputs (core switches) and four outputs (edge switches) are assumed[1]. Packets can go through any downstream path with minimal changes. Segments on each device are shared

[1]This is a simplified scenario, traffic is presumably generated outside this network.

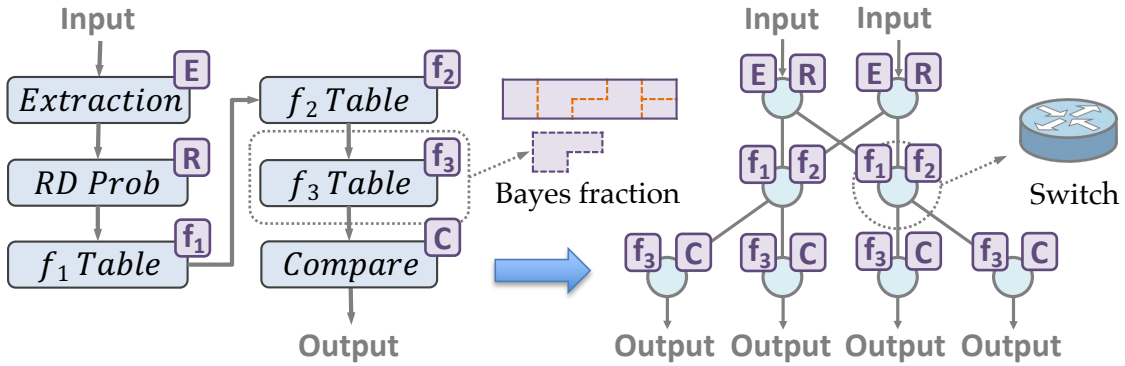


Figure 6.5: An example deployment of an in-network algorithm (Bayes) on a network topology (Folded-Clos).

among multiple paths flowing through the node. This figure shows an ideal example of a distributed deployment, where *Extraction (E)* and *read probability (RD Prob)* are deployed in the first hop (core switches), *feature tables 1 & 2* (f_1, f_2) in the second hop, and *feature tables 3* (f_3) and *Compare (C)* in the last hop (edge switches). The deployment may be obvious, as the network is symmetric and well structured. For larger or more complex networks and programs, the deployment is complicated, as demonstrated in §6.9.

6.3.4 Related Work

NFV service chain planning. Service distribution is common in Network function virtualization (NFV) [220, 98, 101, 42, 41, 99] but different from DINC. NFV-based work mainly focused on traditional processing, where the network serves just as a medium, and latency and throughput are the main objectives. In DINC, the network is both the processing element and the medium, and in-network services are deployed across many paths. DINC's deployment augments network functionality without changes to routing.

Distributed in-network computing. As shown in Table 6.1, previous research

Work	Slice Code	ILP Planner	Any-to-Any	Code Gen.
Hermes [40]	✗	✓	✗	✗
SRA [127]	✓	✗	✗	✓
SPEED [39]	✗	✓	✗	✗
ClickINC [221]	✗	✗	Partial	✓
Flightplan [193]	✓	✗	Partial	✓
DINC	✓	✓	✓	✓

Table 6.1: Comparison between related works. Flightplan supports any to any while changing routing rules.

efforts [40, 127, 39] primarily focused on distributing programs across devices to support a single path through the network. Notably, Hermes [40] and SPEED [39] did not support slicing of a given program. In SRA [127], Match-Action tables were disaggregated, and routing rules were introduced to reach the next table. None of these studies addressed the many-paths, any-to-any routing problem. ClickINC [221] required program translation into a new language and mainly supported symmetric topologies. Flightplan [193] demonstrated P4 program disaggregation, however, it is designed for rack/pod scale and its BSP planner is less suitable for large-scale complex topologies. Importantly, Flightplan also introduces changes to routing rules and does not explore coexistence with network functionality. In contrast, DINC does not modify routing rules, scales to large networks, and coexists with basic network functions.

6.4 DINC Overview

We first provide an overview of the DINC framework's operation, shown in Figure 6.6. DINC is given an in-network computing program with both data plane and control plane code components (shown in ❶), and a network topology

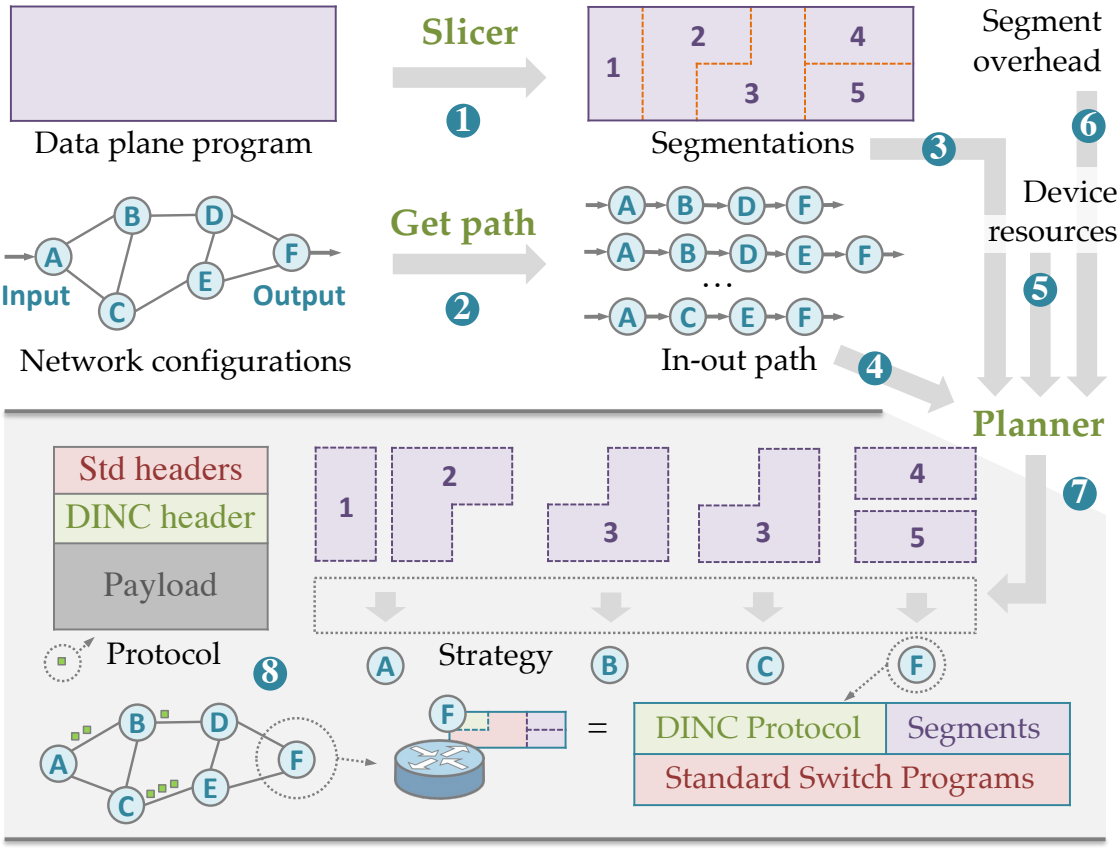


Figure 6.6: DINC workflow overview (steps ❶ to ❽).

(shown in ❷). DINC's P4 slicer (§6.7.1) extracts the program resource requirements (shown in ❻), dependencies (shown in ❸), and metadata information. The network controller provides the routing table, with all paths identified either by the controller or DINC (shown in ❹) and the resources available on each network device (shown in ❺). DINC's planner (§6.5) uses the outputs from steps ❸ to ❻ to craft an integer linear programming (ILP) problem and outputs a deployment strategy in step ❼. This guides the P4 generator (§6.7.2) for data plane and control plane codes generation in step ❽. To provide an ideal deployment of in-network computing program segments on complex network topologies, DINC answers three key questions:

Q1: *How to plan and distribute segments across multiple paths in the network?* (§6.5)

Q2: *How to ensure functionality when programs are distributed inside a network?* (§6.6)

Q3: *How to make distributed in-network computing easy to deploy?* (§6.7)

6.5 Planning

Most previous distributed in-network computing works focused on single-planned paths [48, 104, 207]. In this section, I address the absence of distributed planning for in-network computing across *multiple paths*, without influencing the routing rules of the original network. To tackle this, the problem is formulated within the DINC planner as an ILP problem and I introduce a solution with reduced complexity. In §6.5.1, I introduce the network model and algorithm partition and formally define a deployment strategy. §6.5.2 and 6.5.3 model the deployment optimisation mathematically into an integer linear programming problem. An ILP solver is discussed in the §6.5.4.

6.5.1 Network Model

We focus on the in-network computing tasks planning problem on programmable devices and hence ignore undeployable (unprogrammable) nodes after edge contraction. A set of network devices with order and without duplication along a data trace connecting the input and output nodes is referred to as an *In-Out Path*. A network with N_d deployable devices can be represented by tuple $(\mathcal{D}, \mathcal{P})$, where $\mathcal{D} := \{1, \cdots, N_d\}$ is the set of devices and $\mathcal{P}$ contains $N_p := |\mathcal{P}|$ paths. Each element $P_i \in \mathcal{P}, \forall i = 1, \cdots, N_p$ is an ordered set of size l_i, i.e., $P_i = \{p_i^1, \cdots, p_i^{l_i}\}$. The chain $p_i^1 \rightarrow \cdots \rightarrow p_i^{l_i}$ represents a path from an input

device p_i^1 to an output device $p_i^{l_i}$, where l_i is the total number of devices in path i. There are N_r types of resources in the programmable device networks (e.g., storage), and we use R_d^r to denote the available resource type $r \in [N_r]$ on device $d \in \mathcal{D}$. Assume that the target in-network computing algorithm can be decomposed into N_e elements. Let $\mathcal{E} := \{1, 2, \cdots, N_e\}$ be the set of all the algorithm's element indices. For the DINC planner, let $X_{e \to d} \in \{0, 1\}, \forall e, d$ be the deployment decision (strategy), where $X_{e \to d} = 1$ indicates algorithm element e is deployed on device d. If $X_{e \to d} = 1$, by running element e, device d will cost O_e^r units of resource r (e.g., CPU running time or storage). The goal is to design a *deployment strategy*, i.e., $\{X_{e \to d}\}$, that can achieve application objectives (e.g., low latency) and a small number of duplicated deployed segments while consuming little resources on the programmable devices.

6.5.2 Constraints

A successful deployment strategy should typically satisfy dependency, integrity, and resource constraints, which are explained as follows:

Dependency. Assume that the N_e elements have to be completed in order among each *in-out path*. For every path, any elements e in the in-network computing algorithm should appear at least once before its successor e'. Mathematically, if element e' is deployed on device p_i^j, i.e., the j-th device of the i-th path, the deployment decision variable $X_{e \to p_i^j} = 1$ and element e has to be deployed on at least one node in set $\{p_i^1, \cdots, p_i^{j-1}\}$, i.e.,

$$\sum_{k=1}^{j-1} X_{e \to p_i^k} \geq X_{e' \to p_i^j}, \forall e < e', i \in [N_p], j \in [l_i]. \tag{6.2}$$

Integrity. All the segments should be executed on each *in-out path* to satisfy the integrity of the algorithm. Therefore, on every path i, every element $e \in \mathcal{E}$ should

appear at least once, i.e.,

$$\sum_{k=1}^{l_i} X_{e \to p_i^k} \geq 1, \forall i \in [N_p]. \tag{6.3}$$

Resource Constraints. The type r resource on each device d must be sufficient for all deployed elements. Therefore,

$$\sum_{\mathrm{e} \in E} O_e^r X_{e \to d} \leq R_d^r . \forall d \in [N_d], r \in [N_r]. \tag{6.4}$$

Other Potential Constraints. The above constraints can be extended, encompassing additional and customised requirements, such as limitations on hops and latency for specific services, constraints on throughput, and variations in resource constraints using heterogeneous devices. The specific constraint conditions applied can be adjusted based on the specific deployment scenario.

6.5.3 Objectives

We expect our deployment strategy to optimise the following three major objectives in distributed in-network computing: minimise the resource consumption, minimise the computation delay and minimise duplicate deployed segments. Quantifying the aforementioned objectives is as follows:

Resource Consumption. A good *deployment strategy* should minimise the total resource consumption of all types of resource r. Let $\mathbb{O}_{\mathrm{R},r}$ be the total type r resource cost in the network. Recall that O_e^r is the type r resource overhead if element e is deployed, for each deployment strategy $\{X_{e \to d}\}$, then $\mathbb{O}_{\mathrm{R},r}$ can be computed as follows:

$$\mathbb{O}_{R,r} := \sum_{d=1}^{N_d} \sum_{e=1}^{N_e} O_e^r X_{e \to d}. \tag{6.5}$$

Latency. Assume that the transmission delay on each path is fixed regardless of the execution task. I then focus (as an example objective) on minimising the execution latency. The execution latency on each in-out path $j \in [N_p]$ can be computed by accumulating the execution time of each element e. Suppose L_d^e is the execution time of element e deployed on device d. The execution delay $\mathbb{O}_{L,i}$ on path i can be computed by:

$$\mathbb{O}_{L,i} = \sum_{j=1}^{l_i} \sum_{e=1}^{N_e} L_e^{p_i^j} X_{e \to p_i^j} \tag{6.6}$$

Multi-objective Optimisation. We aim to design a deployment strategy that can optimise resource consumption and performance objectives (e.g., latency) at the same time. Such a problem can be formulated as a multi-objective optimisation problem, where there may exist multiple Pareto optimal points (i.e., the execution latency cannot be minimised without consuming fewer resources). To find such Pareto optimal points, we took a linear scalarisation approach [94]. The ultimate objective function is a weighted linear combination of the execution latency on all in-out paths and all types of resource consumption. i.e.,

$$\mathbb{O} := w_R \sum_{r=1}^{N_r} \mathbb{O}_{R,r} + w_L \sum_{i=1}^{N_p} \mathbb{O}_{L,i}, \tag{6.7}$$

where $w_R, w_L \in \mathbb{R}^+$ are the weights of resource consumption and execution latency, respectively. The above optimisation objectives can be flexibly adjusted based on varying service requirements and deployment environments.

6.5.4 ILP Solver

Recall that deployment decision $X_{e \to d}$ are binary decision variables. Minimising the scalarised objective function (6.7) under constraints (6.2), (6.3) and (6.4) can

be rewritten as an ILP problem. The constraints (§ 6.5.2) and objectives (§ 6.5.3) in this section are the most common ones and can be amended or replaced (§ 6.7) for different applications.

Searching for the optimum solution of an ILP is NP-hard. We can tackle this problem using the branch-and-cut method [151] that combines the cutting plane method and brand-and-bound method. The solving details using standard library `SciPy` can be found in DINC's GitHub repository [236].

6.6 Distributing Segments to Nodes

The previous section presented a theoretical solution for planning the program distribution strategy, while questions still remain on how to effectively deploy the program element into a programmable network topology based on the *deployment strategy*. In order to clearly address these questions, I provide a brief introduction to how typical in-network computing programs are executed. Figure 6.7 shows a sample in-network computing program. When packets come in, features f_1 and f_2 are extracted from the packet header to PHV as metadata. Together with m_1, m_2, and f_3, all metadata convey intermediate results between segments e_1, e_2, and e_3, where the output is feature f_3. The output f_3 from the last segment e_3 is put back into the packet header as a result. Through this process, PHV will be cleared between packets and metadata will reset. For the distributed deployment of such a sample program, each segmentation should be sequen-

Figure 6.7: In-network computing program, encode-based ML model as an example [245], and the sliced segments of the program.

tially executed and each segment requires metadata from dependent segments in order to function correctly.

To realise this, two questions should be addressed. 1. *How to encode and pass metadata between segments?* 2. *How to ensure all program elements are being executed in order and without duplication?* These two questions are solved by using a DINC dedicated header (§6.6.3).

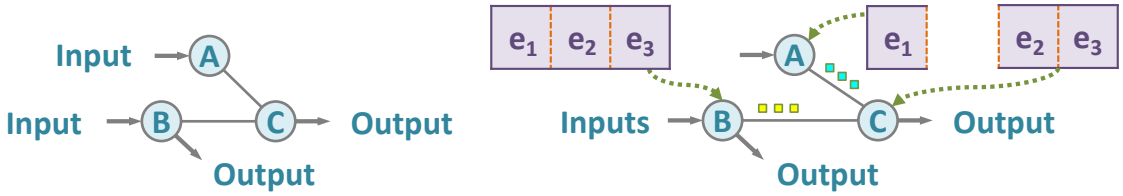


Figure 6.8: The sample in-network computing programs deployment on the programmable network with two *in-out paths*.

A simple toy example shown in Figure 6.7 and Figure 6.8 is used to demonstrate these two problems. Consider given constraints when deploying the shown algorithm in programmable network devices. One possible *deployment strategy* ($X_{e \to d}$ in §6.5) is plotted in Figure 6.8. In this case, there are three *in-out paths* within the topology $\{\{B\}, \{B \to C\}, \{A \to C\}\}$. The two problems addressed in this case are:

1. Along path $\{A \to C\}$, metadata from segment e_1 on device A should be encoded and passed to device C for segments e_2 and e_3. At the same time, besides the input and output of segment e_1, the system should recognise f_2 (input of e_2 on device C) as part of the input and output on device A. (§6.6.1)
2. Along path $\{B \to C\}$, after packets execute e_1, e_2, and e_3 on device B, the segments e_2 and e_3 on device C should not be activated. While these two segments should be activated when it receives the packet along the path $\{A \to C\}$ from the device A. (§6.6.2)

6.6.1 Metadata Passing

Any used metadata should be well encoded and passed to the target segment from its predecessor segment. There are several options to convey information from one node to the other. The first option is to mirror and send the packet independently [16]. However, this option is not realistic without synchronising arrival time and will not give a guaranteed metadata delivery. Another option is to pass metadata through the control plane, the control plane cannot guarantee conveying metadata and has a limited bandwidth for data transfer. In DINC, I choose to store metadata in a predefined header. Although a larger header will increase the average packet size and influence the packet rate, the evaluation shows the influence is minor and the program generated by DINC is able to reach the full line rate. Most importantly, it can guarantee the runtime metadata passing. Once the metadata is marked as the output of the segment, DINC will break and store it in several 32 bits chunks. In the parser of the following dependent segment, the auto-generated encapsulation logic will help to restore the metadata. To guarantee the consistency of metadata passing over segments, and ensure transmission reliability when it is not used in some of the segments, DINC designed a checking process to auto-complement missing metadata on manually added markers.

6.6.2 Segment Traversing

In the distributed scenario, it is possible that segments are being duplicated especially when multiple paths join together as shown in Figure 6.8. To ensure every segment is executed once and only once, a bit map is held by each packet. Before the execution of any segment, it will check if the packet is fully processed

by the predecessor-dependent segments. Once passing this check, along with the execution of the segment, the bitmap will be updated with the current segment. With the help of this per-segment bitmap checking process, no matter which path the packet comes from or which set of segments the packet has been passed, they can share the same set of segments without being duplicated and executed. All the checking processes and segment bitmap IDs are auto-generated by the DINC framework.

6.6.3 DINC Header Design

No matter what *deployment strategy* is employed, if segments are allocated on different devices, information (e.g., intermediate results and segment ID) needs to pass between devices. Thus, a standard DINC protocol header is designed to meet requirements and allow services like intra-metadata encoding and passing (§6.6.1) and segment traversing (§6.6.2).

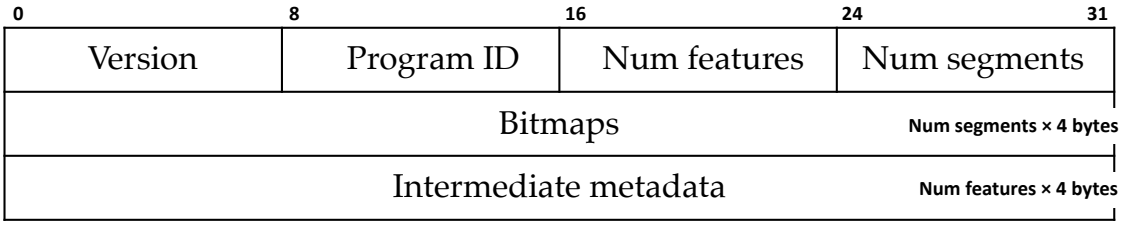


Figure 6.9: The DINC header design. Fied program ID, number features, and number segments reflect the information of the deployed in-network computing program. Field bitmaps marks all processed segments. Field intermediate metadata holds all feature and variable information that needs to be passed.

As shown in Figure 6.9, beyond the DINC version, the protocol contains three types of information: 1. program related, 2. metadata related, and 3. segment related. The program-related information shows which one or several in-network computing programs are currently executed and included in the packet, mainly

based on the Program ID field. The metadata-related contains information on the number of intermediate features included and their values. The bitmap information indicates the set of segments already executed. The overhead of DINC is limited to 16B if the program is sliced into 32 segments or less, and the metadata traversing between two segments is within 8B. Given the header in Figure 6.9 was designed with redundancy for future extensions (e.g., version, number of features, number of segments, 32 bits bitmap), the used DINC header can be smaller. The header size can be further minimised by adding constraints and objectives in the planner [40]. Thus, the overhead of the DINC header is usually not significant (§6.9.2 DINC overheads).

6.7 DINC Framework Design

The deployment of distributed in-network computing is challenging for two reasons. The first is the complexity of the data plane program. It is hard to correctly extract information from a given data plane program according to user expectations. The second is the complexity of data plane program generation. Reconstruction of a valid data plane program under a certain architecture, coexisting with its use case, and with an embedded DINC header is difficult, especially when the slice is complex.

The DINC framework is designed to tackle these two challenges. DINC contains a data plane program Slicer, a strategy Planner, a code Generator, and a Tester, allowing the generation of sliced data plane programs and their deployment on target hardware nodes. The detailed workflow of the framework shown in Figure 6.10 is as follows:

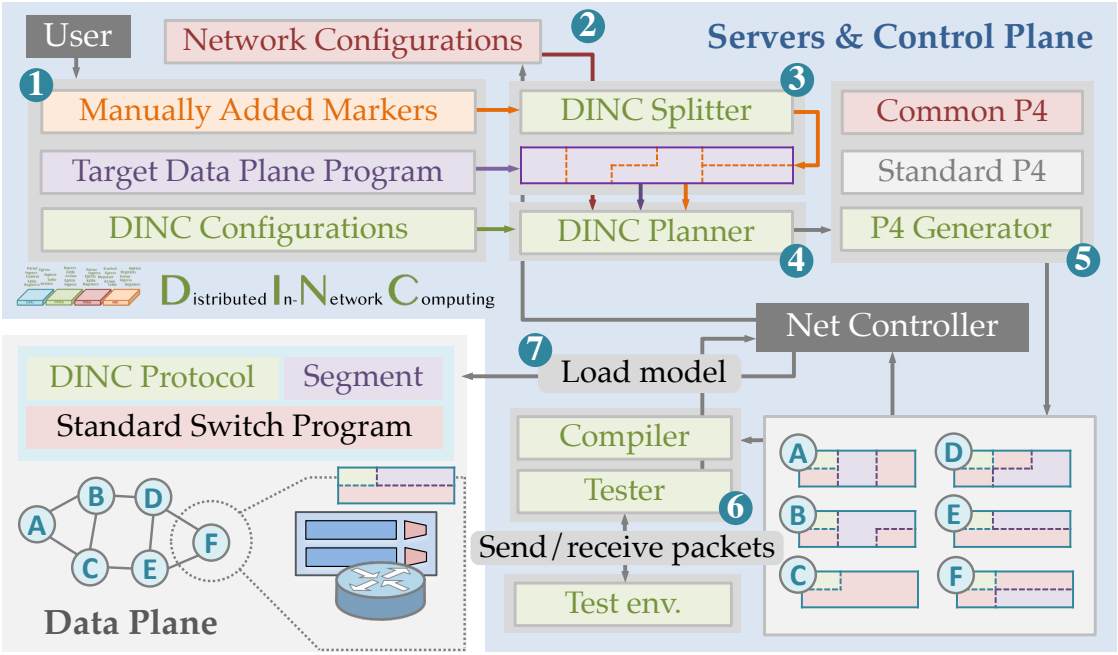


Figure 6.10: The DINC framework components and workflow steps (❶ to ❼).

❶ User input: The target data plane program and slicer markers are required as user inputs for DINC configurations.

❷ Network configuration: The network topology and its related resources are required and stored in DINC configurations, which can generate directly from the network controller or emulate by using the DINC framework.

❸ DINC slicer: The network slicer slices the target program based on the markers, builds the dependency between each segment, and digs the resource information of each segment.

❹ DINC planner: The planner is used to generate deployment strategies for the target program. Based on the input network topology, segment dependency, remaining resources for each network device, and the resource overhead of each segment from previous steps, the planner applies the ILP (§6.5) to formulate the plan.

❺ P4 generator: The code generator generates data plane codes for all programmable network devices. To generate the codes, the generator jointly

combines the selected architecture and use case based on the DINC configurations, and combines the program segments of the input program from the deployment strategy.

6. Synthetic test: Before loading the generated code to the real hardware, a synthetic test is implemented by the framework. The framework will activate a test environment with the same topology as the input network, and load the generated model for each node. The test can then be done within the environment while the content varies by use cases, e.g., the functionality of the normal network functions and in-network computing program. Common tests include the functionality verification of the network protocol and in-network computing programs. P4 debugging tools can also be used in this field to trace and correct bugs in each generated segment [251, 190].
7. Segment loader: When the test is finished, the generated data plane programs with sliced segments will be sent to the controller for deployment.

From the seven key components mentioned above, the DINC planner has been introduced in Section 6.5. The other two key components that simplify the process of distributing segments to nodes are the DINC specialised P4 code slicer (§6.7.1) and P4 code generator (§6.7.2). The remaining four components are used as auxiliaries for the evaluation and functionality tests. The testing results of this framework will be presented in §6.9.

6.7.1 Code Slicer

The DINC slicer is used to segment the input data plane program and extract information from it. There are two design options, the first is a compiler-like mod-

ule that is able to auto-detect and extract information. Despite its advantages, this design needs extra configuration efforts, limits the scope of input programs and information types and can lead to errors. The second, used by DINC's slicer, is to use manually added markers to slice the program and extract the information. This solution is lightweight and flexible.

A functional slicer needs to support several properties of the program: 1) The code is between different parts of the pipeline (e.g., parser, ingress, egress). 2) The dependency of each segment is complex. 3) The required resources may change between cases.

Manually added markers enable solving these challenges. I use a fragment of a sample-sliced program in Algorithm 6.2 to show how it works. In the program, markers @! and !@ are used to notify the slicer where the information should be extracted. The marker design is flexible and can be customised in the framework. The markers are written in comments and do not affect the execution of the original program.

Multiple types of information among markers are used to indicate the DINC slicer the segment identifier and the position. For example, in Algorithm 6.2 lines 13 and 15, the Slice identify the segment ID is 0 and the End shows the end of this segment. Position in markers shows the position of that segment is control apply (line 13), while the segment in control block is marked as control (line 2). All segments with same ID and position will be merged and saved for future use.

The dependency information is embedded in the marker Previous (Algorithm 6.2 line 13, 16, and 19). It indicates if this segment has any prerequisite segments (e.g., `Pre,0,1` in line 19 means that segment 2 depends on results from segments 0 & 1). The prerequisite segment can be none, single, or multiple.

```
control Ingress(...){
    // @!S,0!@ @!P,control!@
    table e_1{...}
    action e_1(...){...}
    // @!E,0!@
    // @!S,1!@ @!P,control!@
    table/action...
    // @!E,1!@
    // @!S,2!@ @!P,control!@
    table/action e_3...
    // @!E,2!@
    apply{
        // @!S,0!@ @!Pre,none!@ @!P,control-apply!@
        e_1.apply();
        // @!E,0!@
        // @!S,1!@ @!Pre,none!@ @!P,control-apply!@
        e_2.apply();
        // @!E,1!@
        // @!S,2!@ @!Pre,0,1!@ @!P,control-apply!@
        e_3.apply();
        // @!E,2!@
    }
}
```

Algorithm 6.2: Sample in-network ML P4 code (DT) with markers to show a sliced pattern as in Figure 6.7. Segment information like End, Slice, Previous, and Position are included between markers @! and !@.

The required resource information, metadata in/out, and any other required information are used in a similar way as a dependency to be added to the marker. It will be totally flexible for the DINC slicer to add or remove required resources, if the current resource marker is missing for a specific segment, it will be marked as zero. The resource with the same segment ID will be added together without being influenced by the marker position.

This is not the only option for slicer design and the DINC's modular framework allows personalised slicer design that is different from the current design (§6.7.3).

6.7.2 Code Generator

The design objective for the code generator is to generate the data plane program that can be directly applied to each node, and with maximise reused resources. To meet this, the DINC generator is designed based on a controller plus multiple architectures and uses case building blocks. Before generating any node, the controller will invoke the selected architecture and use case block based on the network controller or the DINC configurations. Besides writing the skeleton of the data plane program, the selected architecture folder will call the respective use case function for writing use case codes at each position of the program. The deployment strategy will also be applied in the architecture block to drive the regeneration of the sliced segment at the right position. Functions that can auto-generate P4 codes dealing with metadata embedding and extraction as well as bitmap checks are applied and can be called by any nodes or segments during the generation process.

6.7.3 Modular Framework Design

As new architectures and target devices appear frequently, a design that allows extension is required. The DINC framework applies a modular design, where a centralised controller is used to call corresponding modules (e.g., related to solver, topology, or use case) according to the input DINC configurations. When a module is selected, all the functions under this module will be loaded to the controller. With this modular framework design, DINC can be adapted to the required case easily and with strong customised capacity.

6.7.4 Multi-program Deployment and Resource Fairness

DINC is primarily designed to distribute a single large program across multiple devices when it cannot fit on a single device. However, it does not currently support the automatic distribution of multiple programs provided incrementally; it only supports iterative deployment. To ensure fairness, limitations should be imposed within the planner, such as allocating available resources evenly or adding upper constraints on the total resources for each resource type.

This subsection discusses the runtime fairness of DINC's deployment of algorithms from two perspectives. 1. *Fairness in terms of planning*: Network infrastructure owners (or service providers) have the flexibility to ensure fairness in the deployment through various strategies, such as setting constraints on the maximum resource usage per device per resource category or imposing constraints on the overall resource usage. Alternatively, they may opt for more aggressive strategies, such as a first-come, first-served approach. The choice of strategy depends on the characteristics and requirements of the network operator. 2. *Fairness in terms of services*: DINC, particularly in scenarios devoid of intricate operations like multicast or recirculate, inherently provides fairness in network services. This is because DINC does not disrupt the load balancing and forwarding of the regular network, which is one of its distinguishing features. If absolute fairness is required, maximum throughput limits for individual services can be configured on switches. Nonetheless, services that may introduce additional traffic require the addition of constraints in the planner to mitigate potential impacts.

6.8 Implementation

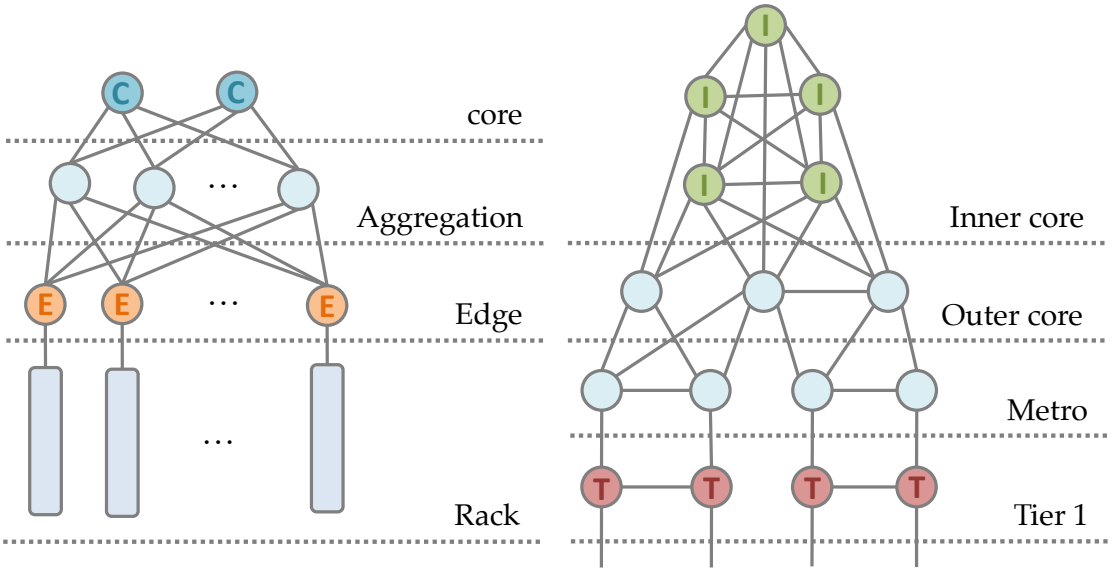
The DINC framework is implemented in Python 3.10. The ILP solver is based on *milp*, a mixed-integer linear programming solver in *SciPy v1.10.0*. The topology is stored using *NetworkX 3.0*. The framework is open and available on the DINC's GitHub repository [236]. DINC supports a range of predefined modules, including topologies, solvers, slicers, architectures, targets, and testers. Under topologies, DINC currently supports Fat-Tree, Folded-Clos, and BT WAN. The P4 slicers currently support manual configuration inputs, where inputs come from manual marking or are auto-generated by a data plane automation framework like Planter. DINC supports two architectures: v1model [9] and TNA [97] and coexists with simple forwarding, RARE [130], and Intel's *switch.p4*.

6.9 Evaluation

Our evaluation focuses on three key questions: (i) does DINC enable deployment of large in-network computing algorithms that were not previously feasible (§ 6.9.2)? (ii) is DINC suitable for different network topologies with various configurations and scales (§ 6.9.3)? and (iii) is DINC's performance sensitive to network configurations and parameter tuning (§ 6.9.4-§ 6.9.5)?

6.9.1 Evaluation Setup and Datasets

1. Topologies. DINC is evaluated using two common network scenarios — a Folded-Clos based data center network topology (shown in Figure 6.11(a) with 3 core, 6 aggregation level, and 24 edge switches, connecting around ten thousand



(a) Clos Data Center Topology **(b) BT ISP Network Topology**

Figure 6.11: Network topology used for evaluation.

servers), and a large Internet service provider (ISP) backbone network - British Telecom (BT) (shown in Figure 6.11(b) with more than a thousand nodes: 8 inner core, 12 outer core, 63 metro, and 925 tier 1 switches) [168]. An aerial view of the BT topology is shown in [242].

Using the two topologies, Table 6.3 shows a subset of potential ingress/egress switch setups. A total of five setups are presented, and are combinations of two dimensions: communication direction and deployment view. The direction of communication in a network may depend on a service requesting or responding, going between Core (C) and Edge (E) switches in Figure 6.11(a) or Inner Core (I) and Tier 1 (T) switches in Figure 6.11(b). Peer-to-peer communication between end servers is considered between Edge (E) switches or Tier 1 (T) switches.

In terms of application deployment, a service may target all traffic going through a network, or a subset of users. In the first case, the network operator is the one that defines and controls the deployment. In the second, which

fits e.g., campus networks, only a subset of the network might be used and have proprietary constraints.

2. Workloads. I use several popular in-network computing programs from different domains: in-network telemetry (INT) using PINT [23], load balancing using Pegasus [121], and in-network ML inference based on Planter (Chapter 4) with 5 ML models. These applications represent advanced in-network computing services, yet they require significant resources and need optimisation when co-deployed with other data plane forwarding functions.

3. Metrics. The efficiency of distributed in-network computing is evaluated for the following: *1. Number of used nodes:* The number of nodes required to deploy an in-network computing program. This number depends on the total number and connectivity of available nodes. *2. Hops per path:* The number of hops required to complete a given in-network computing program. Hop number is directly proportional to the latency, and hops are used to represent latency, and to ensure the evaluation is unbiased in terms of network setups and equipment selection. Cumulative distribution function (CDF) of hops provides further insights into the distribution of required hops per path. *3. Duplication of segments:* In topologies with multiple in-out paths, duplicate segments may exist on different paths. The amount of duplication is relative to the total number of nodes and shows the efficiency of the deployment strategy. *4. Execution time:* Job completion time of the ILP solver is critical to its application. The ILP solver's execution time depends on multiple factors, ranging from the topology to the program, and the solver will be generally scalable if the time increases in a linear manner.

4. Environment. Both large-scale simulation and small-scale hardware tests are conducted. The simulation is based on Mininet and BMv2, and hardware tests run on Tofino using APS-Networks BF6064X-T ($64 \times 100G$, SDE 9.6.0) and two

NetBerg Aurora 710 ($32 \times 100G$, SDE 9.9.0) switches. Tofino compiler is used for feasibility testing.

6.9.2 Functionality

Simulation: DINC is evaluated both on the Folded-Clos data centre network and BT topology, using the five setups and seven workloads. I measure the resource consumption and number of hops on both single and distributed deployments for all workloads, as summarised in Table 6.3 and 6.4.

Program	Stand Alone			Seg.	Setup		
	Stage	Alone	Coexist		No	Clos Topology	BT Topology
INT-PINT [23]	7	✓	✗	5	4	In: all C/Out: one E	In: all I/Out: one T
LB-Pegasus [121]	8	✓	✗	4	1	In: all E/Out: all C	In: all T/Out: all I
ML-NB [245]	8	✓	✗	2	4	In: all C/Out: one E	In: all I/Out: one T
ML-SVM [243]	9	✓	✗	3	5	In: one E/Out: one E	In: one T/Out: one T
ML-DT [218]	2	✓	✓	2	3	In: one E/Out: all C	In: one T/Out: all I
ML-XGB [243]	6	✓	✗	4	2	In: all C/Out: all E	In: all I/Out: all T
ML-RF [243]	6	✓	✗	3	1	In: all E/Out: all C	In: all T/Out: all I

Table 6.3: Sample programs and setups on both topologies. Segmentation (Seg.) details can be found in DINC repository [236]. Duplication (Dup.) - number of duplicated segments. ✓/✗ Deployment feasibility. Distribution (Dis.) - distribution feasibility. Nodes - nodes used/total. Hops - average used hops/path length. C/E/I/T refers to the node in Figure 6.11.

DINC is capable of processing distributed deployment problems at data centre level with about 10000 servers or at ISP level with around 1000 switches. As Table 6.4 shows, all workloads, including those that cannot coexist with RARE switch functionality (✗ in Coexist column) are feasible in a distributed deployment, coexisting with network functionalities (Dis. column). As some programs, e.g., ML-DT, coexist with other switch functions, DINC selects the best node

Program	Folded-Clos Topology					BT ISP Topology				
	Path	Dis.	Nodes	Hops	Dup.	Path	Dis.	Nodes	Hops	Dup.
INT-PINT [23]	18	✓	10/33	3/3	9	36	✓	3/1008	1.42/3.92	8
LB-Pegasus [121]	432	✓	30/33	2/3	56	26512	✓	400/1008	3.11/3.81	796
ML-NB [245]	18	✓	9/33	2/3	7	36	✓	6/1008	2.92/3.92	4
ML-SVM [243]	6	✓	8/33	3/3	5	12	✓	7/1008	3/5	4
ML-DT [218]	18	✓	1/33	1/3	0	36	✓	1/1008	1/3.92	0
ML-XGB [243]	432	✓	9/33	2/3	11	26512	✓	410/1008	2.76/3.81	422
ML-RF [243]	432	✓	30/33	2/3	33	26512	✓	400/1008	3.11/3.81	405

Table 6.4: The resource utilisation of distributed deployed sample programs supported by DINC. Segmentation (Seg.) details can be found in [236]. Setups refer to Table 6.3 and Figure 6.11. Duplication (Dup.) - number of duplicated segments. ✓/✗ Deployment feasibility. Distribution (Dis.) - distribution feasibility. Nodes - nodes used/total. Hops - average used hops/path length.

along the path to optimally utilise resources and minimize latency. Such resource optimisation advantages can be found in all distributed deployed programs.

In the comparison between standalone and distributed deployment, as illustrated in Table 6.3 (in the same row), consider the example of ML-RF/XGB. Under setup 2 shown in Table 6.3 and Figure 6.11, with a Folded-Clos data centre configuration featuring 3 core, 6 aggregation, and 24 edge switches (equivalent to 1000 servers), the solver successfully resolves 4 segments. In this setup, more than 400 distinct data paths require service deployment. DINC's implemented algorithm deploys segments on only 9 out of 33 devices, achieving service completion with an average of 2 (out of 3) hops. Additionally, there are only 9 secondary segments duplicated, compared to deploying on each individual data path (which would require thousands of repeated segments) or alternatively routing all flows requiring service through a single path or a subset of paths. This significantly enhances deployment efficiency and reduces resource consumption.

In the comparison between distributed deployment under different path setups (among rows in Table 6.3 and 6.4), workloads in BT topology using all-to-all scenarios (setups 1 & 2) require using less than 40% of the nodes, and each path requires on average less than one duplicated segment. For one-to-one and one-to-all scenarios (setups 3-5), not all the devices are needed for complete task execution. In the data centre setting, the number of all-to-all deployments requires one-third of nodes when data is arriving from the core (e.g., incoming to the data centre). Tasks for outward flows may require all nodes if a latency optimisation is applied (comparing ML-RF and ML-XGB). This shows that DINC's ILP solver can generate planning with efficient node utilisation and limited duplication under different topologies and traffic path conditions for both use cases.

DINC overheads: 1. *Communication Overhead*. Considering the scenarios listed in Table 6.3 and 6.4, the maximum DINC header size needed is 20B. This translates to 3.77% traffic overhead in the ML scenarios for packet classification. Using the ML models for flow-level classification using the first 30 packets per flow [33, 140], this overhead can be further reduced to 1.18%. For in-network telemetry applications, specifically under the Hadoop workload [165, 23], the communication overhead is as low as 0.08%. 2. *Resource Overheads*. DINC's distributed deployment does not introduce additional memory overheads, as illustrated in Figure 6.12(a). However, DINC's distributed deployment does require additional pipeline stages. In comparison to a non-distributed program (the ideal case, which is calculated from the unsegmented program), the deployment of sliced segments requires 2 additional stages overhead in total, as shown in Figure 6.12(b). The two nodes indicate the two consecutive connected network devices used for distributed deployment. These extra stages are primarily used to ensure the sequential execution of segments, as discussed in Section 6.6.2. Due to this design characteristic, other programs in Table 6.3 and

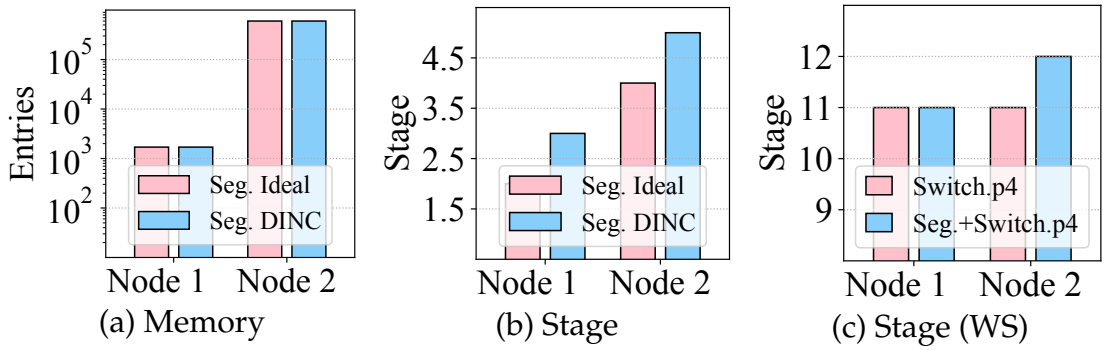
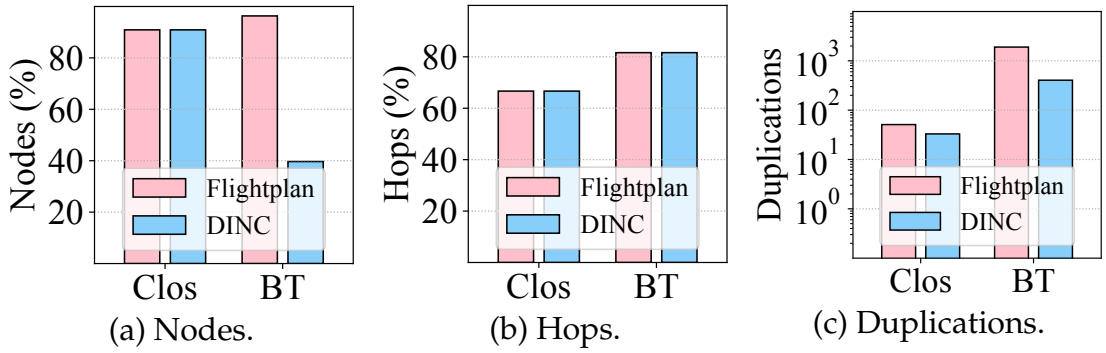


Figure 6.12: Overhead of DINC on distributed deploying ML-RF [243] on two nodes, same as in the last row of Table 6.3 and 6.4. Seg. Ideal - resource consumption of segments calculated from the unsegmented program, WS - with switch functionality (*switch.p4*).

6.4 entail the same resource overhead. Based on the stage consumption results in Figure 6.12(b), Figure 6.12(c) presents the co-deployment together with the L2/L3 switch with 15 network features (*switch.p4*), which is an advanced version of the RARE router [130]. Compared to RARE in Figure 6.2, *switch.p4* consumes more resources, which is used here for a stress test. The figure shows that a 3-stage segment in the first network device (Node 1) can be co-deployed without any stage overheads, while a 5-stage segment requires one extra stage overhead on the following Node 2.

Comparison with state-of-the-art: Flightplan [193] is a state-of-the-art solution for P4 program disaggregation, which focuses on rack/pod scale topology, mainly targets at heterogeneous resource usage, and affects the routing rules by assigning the output port. Detailed comparisons between DINC and related works are in §6.3.4. In this evaluation, I apply the Flightplan planner with objectives in terms of latency, throughput, and hops. I compare a deployment of ML-RF [243] on Clos (similar to the topology used in the Flightplan [193]) and BT topologies using both DINC and Flightplan. As Figure 6.13 shows, given the same program segmentation, in the Clos topology (33 nodes/switches), DINC



(a) Nodes. (b) Hops. (c) Duplications.

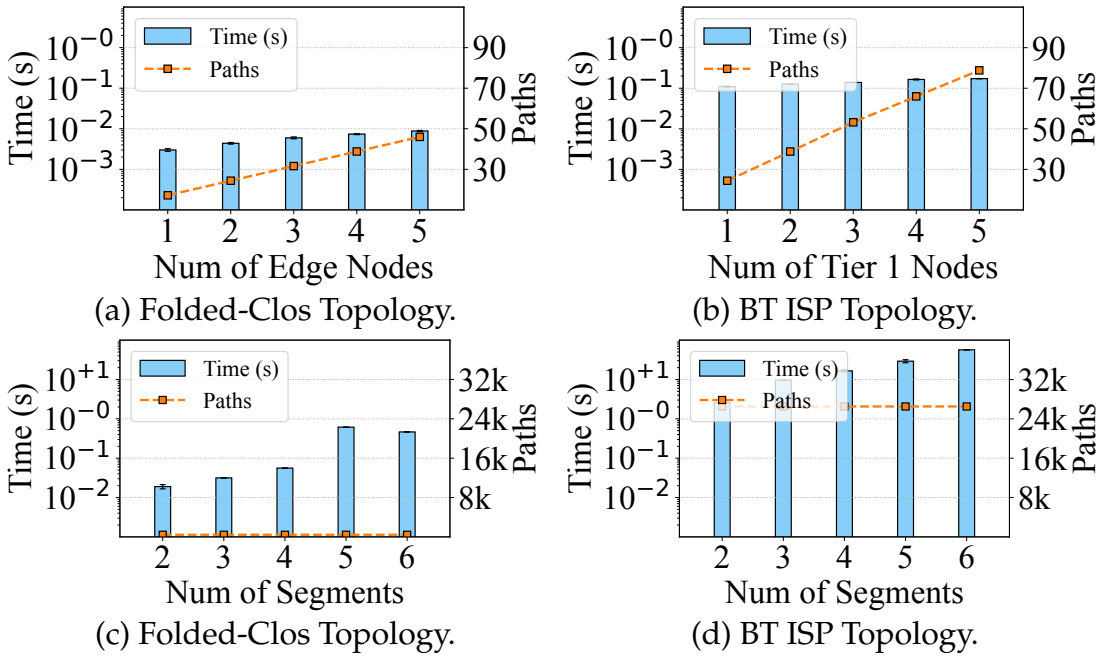
Figure 6.13: Resource consumption comparison between Flightplan [193] and DINC on deploying ML-RF [243] on both BT ISP and Folded-Clos topology with setup 1, same as in the last row of Table 6.3.

and Flightplan use the same number of nodes and require the same number of hops for program completion (classification result). This is a result of a symmetric topology. Benefiting from the ILP planner, DINC requires 35.3% less segment duplications, saving 11.3% of memory resources. The benefits of DINC are demonstrated on the larger BT wide area network (1008 nodes), where DINC uses only 39.7% of the nodes, compared with Flightplan's 96.3%, and reduces the number of duplicate segments from 1901 to 405, saving 54.5% memory resources in total while maintaining the same number of hops. Compared to the heuristics used in Flightplan, DINC's ILP Planner is able to handle complex environments better and place segments efficiently.

Hardware test: DINC is evaluated on a small-scale hardware setup using the distributed RF-ML program on Point-to-Point and Fat-Tree topologies constructed using three Tofino switches, covering the majority of scenarios in Table 6.3 - 6.4. RF-ML is chosen as Planter provides a functionality validation test. The result shows the model functions correctly with the same accuracy as a standalone deployment.

6.9.3 Scalability

Using ML-RF as a leading example, I explore the scalability of the DINC solver on both topologies in terms of two key factors: the number of paths in the topology and the number of segments in the in-network computing program. Despite the exponential increase in solving time for complex topologies, our tests on medium and large networks show that DINC solver efficiently handles the problem within a reasonable timeframe. It achieves millisecond-level planning for the core to single switch deployment (network setup 4 in Table 6.3) for both Folded Clos data centre topology (which is able to support around 10000 servers) and BT ISP topology (for the whole UK). Network-level deployment for a 6-segment ML-RF program across all cores and edge switches is solved in 1 second for the Folded-Clos topology (400 paths) and 1 minute for the BT ISP topology (26k paths). Looking into the trend of time consumption, as reported in Figure 6.14(a) and (b), a setup where input is all Core/Inner core switches and output is an increased number of Edge/Tier 1 switch. When the number of Edge/Tier1 nodes increases, the number of paths also increases linearly, while the time required by the DINC solver increases not as steeply as the number of paths increases. When testing the number of segments, I apply the same network setup 4 for chasing a relatively large usage of paths. In Figure 6.14(c) and (d), when the number of segments of in-network computing programs increases, the time consumption increases exponentially. However, this is acceptable as the number of segments in an in-network computing program is usually small (the good places to "cut" the programme is limited).



(a) Folded-Clos Topology. (b) BT ISP Topology.

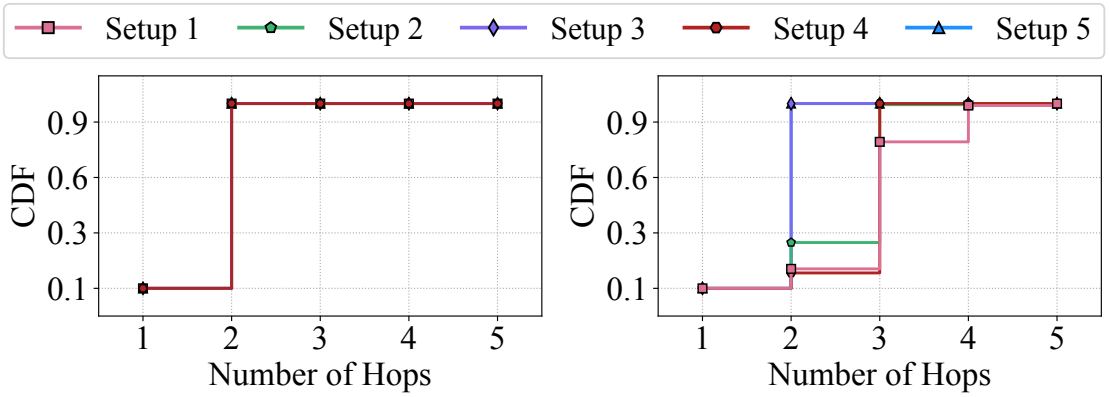
(c) Folded-Clos Topology. (d) BT ISP Topology.

Figure 6.14: DINC ILP solver runtime and number of *In-Out Paths* scaling with coverage area (the number of edge/tier 1 switches) or number of program segments.

6.9.4 Performance

We use random forest (ML-RF in Table 6.3 - 6.4) as a leading example to show how DINC performs under different network scales. The random forest program requires 6 stages and can be deployed within a single path with consecutive switches. Figure 6.15 shows CDFs of hops along all 5 sets of in-out paths generated by 5 different setups. As reported in Figure 6.15(a), random forest can be executed within two hops on all paths on Folded-Clos topology due to its relatively simple and strongly symmetric topology, for all setups.

In Figure 6.15(b), for BT ISP topology, setups 3 and 5 exhibit the best performance in terms of latency as they process the traffic from the user level (input only comes from one switch). Under this circumstance, there is no conflict among the objectives of lower latency, memory, and the number of segments (slicing),



(a) Folded-Clos Topology. (b) BT ISP Topology.

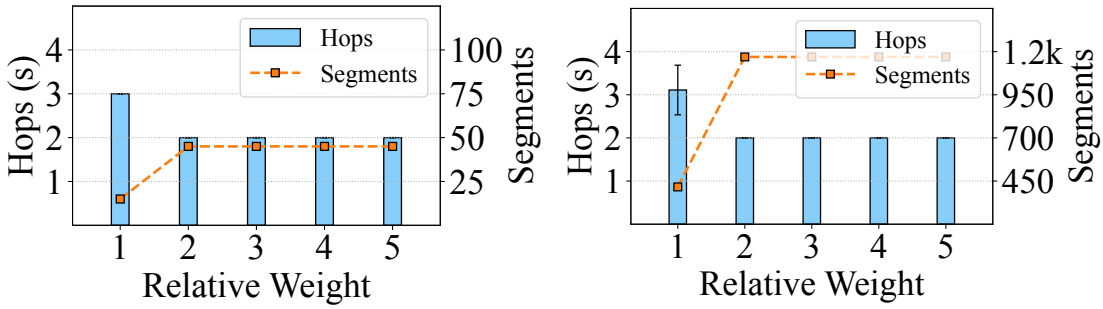
Figure 6.15: CDF of hops needed to complete a program. Setups are in Table 6.3 and Figure 6.11.

allowing for closer-to-source deployment. Setups 2 and 4 have a similar performance and share a relatively larger number of hops than the optimised solution. This is because their tasks come from network inside and the input switches are inner core switches. These setups contain more in-out paths but the traffic comes in at the places where each path is easier to share segments. Setup 1 has the worst latency performance and it requires 4 hops to ensure all paths finish the processing of the program. This is because setup 1 represents the deployment of services from all tier 1 edge switches to the inner core. The shared switches, usually at the inner part of the network, are at the end of each route. It will be an extreme waste of resources if we deploy duplicated segments on all paths for lower latency. For this reason, even though both setups 1 & 2 have the same number of paths and just swap input and output switches, they have different CDF results.

In conclusion, we observe that DINC performs effectively in the distributed deployment of programs, whether in common data centre networks composed of 33 network devices or in large-scale ISP networks with up to 1008 network devices.

6.9.5 Sensitivity

We test the sensitivity and the trade-offs of the DINC solver to the average number of hops and the required number of duplicated segments, as the relative weights between latency and other objectives are changed. Figure 6.16(a) and (b) show that for both two topologies when we increase the relative weight of latency above a threshold, the average number of hops will decrease. Yet, to ensure functionality, the provided strategy requires more duplicate segments. This implies that DINC users have the flexibility to adjust deployment strategies based on the defined objective function and its associated weights. Note that DINC is relatively robust and the deployment strategy will not change abruptly when relative weights change.



(a) Folded-Clos Topology.

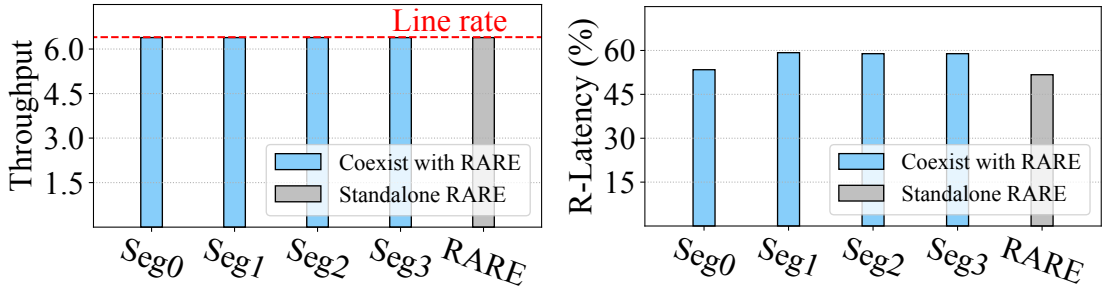
(b) BT ISP Topology.

Figure 6.16: How relative objective weight of w (Equation 6.7) can influence the DINC deployment strategy.

The robustness of DINC's distribution planner is further explored for ML-RF. When changing the number of segments from 2 to 6, DINC consistently generates identical distribution results. This consistency can be attributed to the fact that a certain level of finer algorithmic segmentation remains insufficient to better utilise remaining resources along the path.

6.9.6 Throughput & Latency

This section shows the evaluation of the throughput and latency of different decomposed RF model segments on an Intel Tofino switch $64 \times 100G$, coexisting with the RARE router [130]. In compliance with Intel NDA, I report the relative latency of each segment+RARE with *switch.p4* (an L2/L3 reference switch).



(a) Throughput (Tbps). (b) R-Latency.

Figure 6.17: Throughput and latency on hardware.

For the sample distributed RF segments, As shown in Figure 6.17, all segments can coexist with the RARE router program and can achieve a full line rate of 6.4Tbps bit rate. In terms of latency, the coexistence of RARE & in-network computing application segment will increase 10% of the number of clock cycles through the pipeline. However, even when coexisting with RARE, all deployed segments have a relatively low latency, which is lower than 60% of the reference design (*switch.p4*). The full line rate shown in Figure 6.17(a) proves that the DINC-added bitmap and metadata passing mechanism do not limit the throughput of the system. The relative latency in Figure 6.17(b) shows that the coexistence of network functions when using DINC does not significantly increase the latency through the pipeline.

6.10 Discussion and Limitations

DINC is an important step toward distributed in-network computing and improved utilization of data plane resources.

Scope of DINC. Not all in-network computing programs are suitable for a distributed deployment. DINC is well-suited for stateless programs without packet recirculation. In stateful programs, the limitation stems from potential multi-path routing, rather than DINC's distributed deployment. Packets going through different paths will experience different states (e.g., counter value). This can be resolved by setting DINC to generate single-path results. In programs requiring recirculation, DINC cannot guarantee that the segmented program can resend packets to the switch of the first deployed segment. Instead, recirculation can be transformed to longer programs (as in loop unrolling) before applying DINC.

DINC prerequisites. To run DINC, users need to provide a program, the program's segmentation, and resource requirements for each segment, and it is assumed users have a certain level of understanding of their programs. Automation of the segmentation and resource tasks can be achieved by adding new DINC modules (e.g., [112]). Additionally, the Planner requires the network's topology, routing paths (any or planned), and available resources at each node. The architecture and base program running on each network device are needed to auto-generate the distributed code.

Stage sharing. Although switch functionality consumes considerable resources in the pipeline, it is still possible to share stages with an in-network application. Currently, DINC does not exhaust these spaces during the planning process and leaves them as elastic "spaces". They are sometimes used for operations like

assigning metadata to header and bitmap checking.

Slicing applications in other languages. In general, DINC is designed for P4 code decomposition and distribution on programmable network devices. To support other languages, e.g. NPL or Domino [187], the P4 Generator of DINC should be replaced with the relevant languages' generator.

Partitioning characteristics. Different program partitioning can impact usability. Finer-grain segmentation yields better results from the planner. However, increasing segmentation also increases the computational burden on the planner. Additionally, the placement of slices can impact the amount of metadata used, thereby influencing communication overheads. This consideration can be incorporated into optimisation constraints and objective functions.

Runtime updates. The control plane can drive runtime control and updates based on the planner's results. However, when adding or altering services, or when there are changes to network topology, it is necessary to rerun DINC and update affected nodes. For certain modifications, such as adding new routing paths, it is sufficient to run DINC on the added routing paths rather than the entire topology. While this incremental approach may not yield the optimal result, it avoids non-critical updates and can significantly reduce computational overhead. However, to find the global optimal result, I have to run DINC over the entire topology.

Failed DINC compilation. Three reasons can lead to a compilation failure. 1. *Insufficient Resources for Deployment*: If a path contains less resources than the minimum needs of a given program, or if the network resources cannot jointly satisfy the minimal deployment requirement across multiple paths, no valid planning solution exists. 2. *Inaccurate resource estimation*: An overestimation of device re-

sources by the central controller, an underestimation of required resources by the user, or an underestimation of DINC's code merging overheads can lead to a failed compilation. As these issues are reported during compilation, they can be fixed by adjusting the corresponding resource constraints and re-running DINC. 3. *Inadequate Code Slicing*: In certain instances, excessively coarse-grain code partitioning can result in deployment failures. Very large code blocks may exceed a single device's resources, requiring a finer-grain partitioning, followed by a re-execution of DINC.

Network failures. A device failure does not necessarily lead to a program failure. If multiple paths exist between the source and destination (prior to the failure), operations continue without disruption. This is due to the handling of link failures by the network's inherent load balancing and routing mechanisms, underscoring DINC's advantage of preserving the original routing and network functionalities. In the case of a device failing along a single route, a new route needs to be found and DINC should be incrementally run for this route.

Stages consumption In the evaluation, I use models and configurations that are sometimes intentionally larger than in the original publication. For example, for ML-RF [245], it is possible to deploy a small 3-stage model, with a reduced level of accuracy, while we use a larger model that consumes 6 stages. This is as one of DINC's aims is to improve the scalability of in-network programs, and ML model sizes in particular.

Planning objectives and constraints. The constraints and objectives above are provided as an example and are suitable for most in-network computing workloads. More objectives, such as latency or throughput constraints, variations of use-case and resources constraints, and minimizing packet overheads [40] can be specified in a planner module (§ 6.7). DINC also supports heterogeneous devices

by adjusting constraints on the corresponding resources.

DINC Applications. DINC is designed to execute a single large in-network computing program across multiple network devices in a distributed manner. It supports various in-network computing applications, such as telemetry [23], aggregation [116], inference [247], and load balancing [121]. Furthermore, by relaxing planner constraints, DINC can potentially offer distributed deployment strategies for a broader range of computing tasks (e.g., [123, 58, 68]).

6.11 Summary

This chapter presented DINC, a distributed in-network computing framework to further scale in-network computing services, including in-network ML. DINC decomposes in-network computing programs and generates planning strategies to distribute decomposed segments onto devices within the network (§6.3-6.4). DINC leverages ILP to yield a planning strategy (§6.5), and a custom header to flexibly handle intermediate metadata while traversing segments (§6.6). To streamline the process, DINC is implemented with an automated and modular framework design, facilitating ease of adaptation to new requirements in diverse use cases (§6.7). Experimental results show that large-size in-network computing programs can be deployed within the network using efficient decomposing and optimal planning (§6.8-6.9). DINC boosts applications' performance by utilising network resources without compromising functionality.

CHAPTER 7

IN-NETWORK MACHINE LEARNING APPLICATIONS

After exploring algorithm mappings as well as deployment techniques in preceding chapters, this section focuses on the use cases of in-network ML. Previous efforts predominantly applied in-network ML to anomaly detection, mainly for the classification of packets or flows [218, 33, 119, 84, 89]. This use case has been covered and assessed in Chapters 4 - 6. This chapter first summarises the anomaly detection use cases from previous chapters and extends their scope to include the detection of caching and financial attacks (§7.1) [245, 243, 244, 242]. Next, this chapter explores in-network ML applications in IoT traffic classification (§7.2) [228, 226, 227]. Different from attack detection in §7.1, this use case mainly focuses on the runtime model update solution and privacy-preserving in-network ML mechanism, building on top of Planter. Likewise, based on Planter and IIsy, this chapter further discusses the application of in-network ML in financial market prediction (§7.3) [90, 91]. This chapter also demonstrates the application of in-network ML to load-balancing, based on an in-network reinforcement learning algorithm proposed in Chapter 3 (§7.4) [241]. Note that this chapter does not intend to cover all potential in-network ML use cases. The purpose is to offer some leading examples of in-network ML applications, demonstrating the use of the proposed tools from previous chapters, and inspiring the adoption of in-network ML in other practical contexts. While I was not the lead on all use cases, I contributed as an "expert assistant" and research collaborator. Specifically, I helped define the use cases and their in-network solutions, adapted the mapping framework and strategies to all the use cases, and conducted part of the evaluations.

7.1 Anomaly Detection

ML has demonstrated its efficiency in detecting anomalies across various systems [50, 70], especially networks [52]. Anomaly behaviours often exhibit distinctive features such as high volume and rapid dissemination [17, 12], necessitating swift detection to mitigate its impact. Conventional server-based detection methods tend to be less responsive [201]. Additionally, the utilisation of telemetry data often exhibits a coarse granularity [128], while inspecting all packets places significant pressure on both network capacity and servers' computational ability [96, 15]. In-network ML-based approaches, running on programmable network devices, have high throughput and can categorise packets in real-time within the network. Immediate actions can be taken after categorisation, such as dropping or limiting those packets identified as malicious, or routing them to servers for additional processing. Such measures facilitate the early termination of suspicious traffic, thereby protecting the system and alleviating the burden on both networks and backend servers. Other in-network ML based anomaly detection works, e.g., Mousika [215], LEGO [124], and NetBeacon [249], were in parallel or cited our early works [247, 245, 243].

7.1.1 Network Anomaly Detection

Prior chapters have shown our exploration of using in-network ML for detecting network anomalies and intrusions, such as Denial of Service, Remote-to-Local and User-to-Root attacks [117]. Under this scenario, our detection system can directly take measures, such as dropping or limiting the data rate of packets/flows, based on the classification results of in-network ML on network devices, aim-

ing to safeguard the servers and networked system. To realise this, I proposed algorithm mappings (Chapter 3) and a fast-tailored framework (Chapter 4) to implement various ML algorithms on programmable network devices. I also focused on improving the classification performance of these inference algorithms by introducing hybrid (Chapter 5) and distributed (Chapter 6) in-network ML systems. These novel techniques have been applied to detect traditional network anomalies using datasets such as UNSW [140], AWID3 [36], CICIDS [175], and KDD [191], showing more than 90% accuracy and F1 score, achieving line rate classification with sub-microsecond latency [245]. Moreover, these methods can offload around 70% of the load from servers to network devices, achieving a median latency reduction of approximately 70% with negligible compromise on accuracy. This is achieved at the cost of only 3 pipeline stages and less than 5% of memory on programmable network devices. These results have shown the promise in utilising these in-network ML solutions to detect anomaly behaviours in data centres [244], edge systems [37], and WAN [242].

7.1.2 Caching Attack Detection

The exponential expansion of e-commerce necessitates the accurate identification and swift response to popular items matching users' interests, whether from the perspective of e-shoppers or the platform itself. In-network caching represents a technique that enables rapid responses to user requests by retrieving cached values stored within network devices. However, implementing in-network caching within the backend of an e-commerce system introduces potential security vulnerabilities. The proliferation of automated bots engaging in fraudulent activities has been increasingly observed in this context [55, 192, 230], which has the potential to falsify the popular items stored in the caching systems (Figure 7.1).

Studies have illustrated that even a slight enhancement in the hit rate, by just 1%, can lead to a significant reduction of over 35% in application layer latency [45]. This slower response time can significantly affect sales within the field of e-commerce [157, 219].

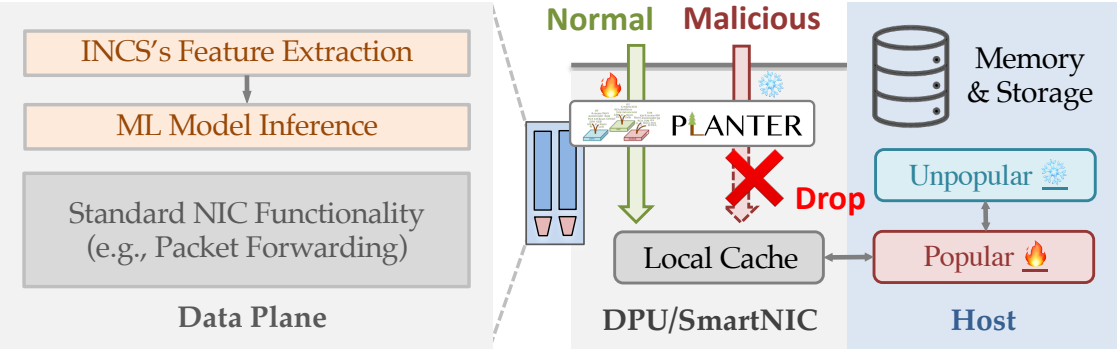


Figure 7.1: General deployment scenario of INCS (source: [84], originally drawn by me).

To mitigate attacks targeting e-commerce systems, we propose INCS [84] (work led by Masoud Hemmatpour), aiming to protect its caching system. As shown in Figure 7.1, INCS employs ML inference on DPU or SmartNIC between caching units (either host-based [61] or in-network caching [103]) and networks. For the inference algorithms, INCS utilises in-network ML to classify incoming requests by extracting requests' features such as crucial time intervals between consecutive requests and the specific items these requests pertain to. For those traffic classified as aggressive bots, INCS limits its rate or immediately drops them. This approach enables the detection, response, and defence against queries generated by bots.

To implement INCS, we introduced a new "Data Loader" module and a "Common P4" module in Planter (Figure 4.2) for realising INCS's specialised feature extraction process (Figure 7.1). In terms of ML models' deployment, INCS directly utilises Planter, implementing in-network ML models on the NVIDIA

BlueField-2 DPU with BMv2 [8] software switch.

We evaluate INCS's performance in distinguishing varying levels of bot activity. The synthetic dataset employed in this assessment is derived from real bot traffic patterns observed on e-commerce websites [84]. The results demonstrate that in-network ML solutions can achieve detection accuracy of 80.42%, 87.36%, and 94.72% for moderate, intense, and aggressive bot activities, respectively. These findings show that INCS is effective in protecting the caching system against bot attacks.

7.1.3 Transaction Fraud Detection

Fraudulent activities are widespread in financial transactions nowadays [163]. Transactional fraud yields substantial and widespread adverse effects, leading to potential financial losses amounting to hundreds of billions for individuals, businesses, and society [133]. Hence, in averting and minimising potential harm, it is important to detect fraudulent activities. In previous works, ML algorithms are extensively employed to detect fraudulent patterns within real-time transaction data streams [25]. However, the surge in transaction volume per second poses a challenge to server-based ML models, impeding their efficiency in reducing detection time. This overload on servers impedes their capacity to promptly prevent the processing and completion of fraudulent transactions.

To address this challenge, we realise an in-network ML-based transaction fraud detection system [89] (work led by Xinpeng Hong), to replace conventional server-based solutions. This system is tailored for deployment within the data systems of financial institutions. As shown in Figure 7.2, the new system directly relocates traditional server-based ML services to programmable network

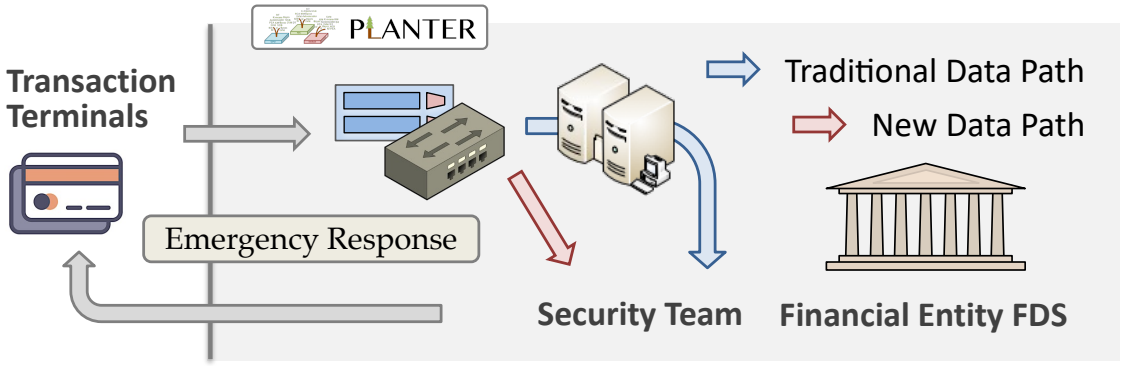


Figure 7.2: General deployment scenario of in-network transaction fraud detection (source: [89], partially contributed by me).

devices. It facilitates real-time transaction fraud detection using in-network ML inference via the Planter framework within the programmable data plane. This in-network fraud detection system specialises in feature engineering for transaction fraud. Specifically, upon receiving a new transaction record, the system operates within the data plane to select, manipulate, and convert raw data into appropriate ML features. These features are then passed to in-network ML models for inference to classify whether the transaction is fraudulent or legitimate.

Similar to INCS, the in-network ML based transaction fraud detection system is implemented using Planter. This system added a new "Data Loader" and "Common P4" module to realise its specialised feature extraction process in-network. Using Planter, this framework can be deployed on both the software target (BMv2, running on DPU) and hardware target (Intel Tofino).

We evaluate this system using several financial fraud detection datasets [129, 176, 182]. In contrast to server-based machine-learning solutions, the in-network system implemented on hardware switches significantly boosts transaction processing rate by over ×800 per second and substantially reduces latency by over ×1300 per transaction. This significantly reduces the time for each transaction consumed in the fraud detection system. Additionally, the system achieves a

relative accuracy of 99.94% to server-based benchmarks and maintains a relative F1-score of 93.66% to server-based ML, with only a minimal decline in classification performance.

7.2 IoT Traffic Classification

The previous discussion focused on the development of in-network anomaly detection solutions. While IoT traffic can be treated as a "special case" of anomaly detection, this separated section mainly focuses on dynamic traffic classification and privacy-preserving in-network ML systems, using IoT as a background. In recent years, there has been widespread adoption of IoT devices. Among the network components facilitating the connection between IoT devices and the core network, IoT gateways play a crucial role by enabling functions like data routing, aggregation, and segregation. With the evolving deployment of IoT devices, there is an increase in security risks. IoT gateways are required to offer traffic analysis and serve as the primary defence against various sources of traffic, as noted in [222]. The distinctive characteristic of ultra-reliable low-latency communications (URLLC) in 5G, coupled with the pervasive and dynamic nature of interconnected IoT devices, amplifies the potential for dynamic threats to rapidly propagate within IoT networks.

Various conventional ML algorithms like random forest, XGBoost, and naïve Bayes have proven their efficiency in IoT device identification and traffic classification [113]. The use case transitions from traditional server-based ML approaches to implementing classification within the network, leveraging techniques from earlier chapters, particularly the Planter framework. However, employing an ML-based solution requires data collection and continual model up-

dates to sustain effectiveness. The model update during runtime remains challenging, and data collection raises privacy concerns. This section investigates strategies for efficiently performing runtime updates of in-network ML models and explores integrating federated learning with in-network ML to establish a privacy-preserving ML inference system.

7.2.1 Runtime Model Update

Given the dynamic nature of IoT systems and the fluctuating patterns within IoT data streams, regular model updates become crucial to counter data drift. Several systems, like OASW [223], have been proposed to adapt to these pattern changes in data streams for conventional server-based ML systems. However, there remains a lack of discussion on performing model updates specifically for in-network ML-based classification within IoT networks.

To realise continuous learning and consistent updates for in-network ML, we introduced P4Pir [226] (work led by Mingyuan Zang) based on Planter. P4Pir incorporates runtime traffic parsing and model updates at the IoT gateway, enabling real-time multi-protocol data collection, in-network ML-based attack mitigation, and seamless runtime ML updates. As illustrated in the P4Pir workflow (Figure 7.3), Step ❶ represents the typical workflow of Planter-based in-network ML detection (Chapter 4), where a model is trained and integrated into M/A table rules and P4 code to analyse incoming traffic. Building upon this existing in-network inference solution, P4Pir not only identifies and blocks suspicious traffic (Step ❷) but also logs this information to the control plane by encapsulating extracted features in a digest (Step ❸). The digest only encapsulates the required features in a digest structure instead of the whole packet. This can reduce

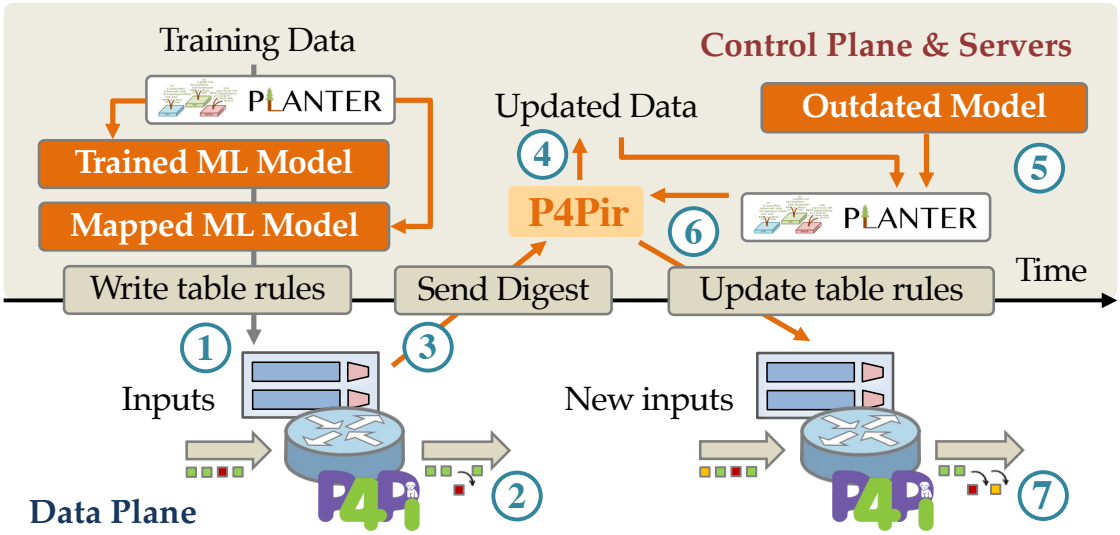


Figure 7.3: System design of P4Pir (source: [228], partially contributed by me).

the extra overhead by forwarding shorter messages, only carrying the features needed for analysis. Utilising these digests, P4Pir retrains the model to adapt to the new traffic pattern (Steps ❹ & ❺). Planter is then used to map the the new model, generating a set of new rules. These updated rules are implemented into the data plane, while outdated rules are removed (Step ❻). With this updated mechanism, P4Pir continually learns from incoming traffic, thereby mitigating abnormal traffic continuously in a dynamic environment (Step ❼).

P4Pir is developed based on the Planter framework using Python 3. The main control of the P4Pir framework is independent of Planter, which calls the functions in the Planter to train ML models and map the trained model to the data plane. During this process, the data plane code is generated in P4 language utilising either the v1model architecture for the BMv2 software switch [8] running on P4Pi (a Raspberry Pi-based platform providing data plane programmability) [115] and Dell Edge Gateway (Dell EMC Edge Gateway 5200) [53].

We evaluate P4Pir using publicly available IoT datasets, *EDGE-IIOTSET* [59] and *CICIDS 2017*. The evaluation shows that P4Pir's runtime model update

mechanism can efficiently learn and adapt model parameters to new attacks. Specifically, P4Pir enhances the accuracy of the decision tree by 50% when the attack transitions from "SCAN" to "DOS" in the *EDGE-IIOTSET* dataset. Similarly, in the *CICIDS 2017* dataset, P4Pir enhances the accuracy of the random forest by over 30% when the attack changes from "SYN" to "SCAN". The update process takes approximately 0.05s for model retraining and about 0.45s for updating new rules in the data plane. The runtime overhead of P4Pir is an additional ~10% temperature and ~21% utilisation in CPU on P4Pi [115].

7.2.2 Privacy Preserving Federated Learning

When considering a larger network setup constructed by several swarms [37], multiple switches serve as IoT gateways between IoT end devices in each swarm and a cloud-based remote server. Each IoT end device is susceptible to various network attacks like Scanning, man-in-the-middle (MITM), and DDoS, with attackers potentially exploiting these end devices for protocol attacks or forming botnets. Training an in-network ML model based on data and traffic collected from each gateway while maintaining privacy presents a challenge.

Federated learning is an ML learning technique designed to train an aggregated model across decentralised devices while preserving privacy and reducing data transmission. Using federated learning, ML models can be locally trained on distributed nodes, and their parameters are sent to a server for consolidation, resulting in a global model. ML algorithms like ensemble tree models have demonstrated effectiveness in implementing FL. In the context of federated learning, differential privacy has been explored to increase training privacy. Differential privacy incorporates random noise during shared models, thereby min-

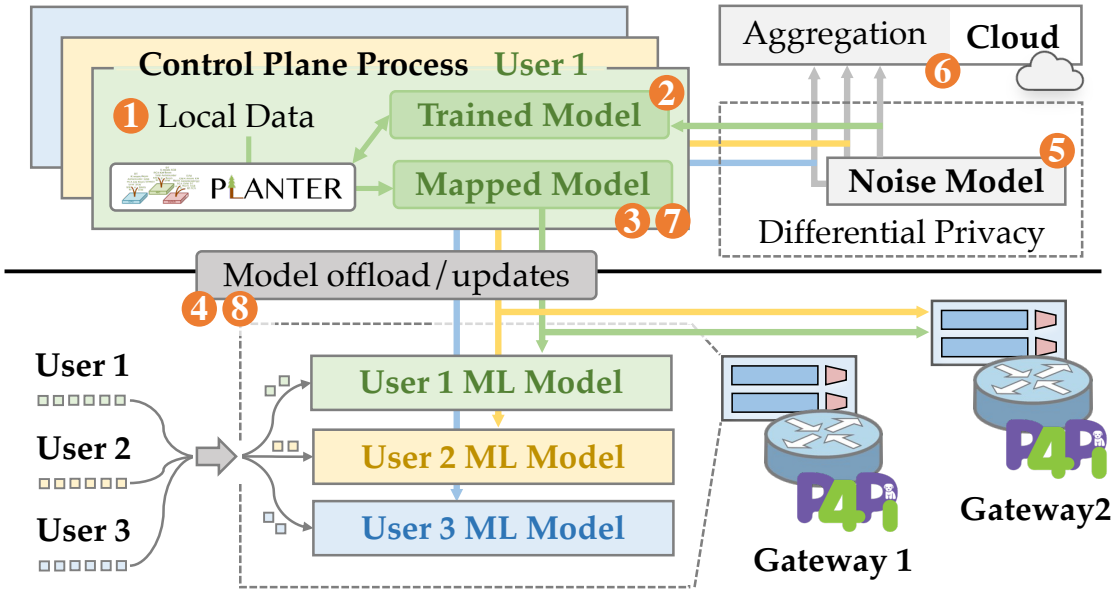


Figure 7.4: System design of FLIP4 (source: [227], partially contributed by me).

imising the risk of attackers intercepting sensitive information related to shared trained models.

In this use case, based on the Planter and P4Pir frameworks, we introduce FLIP4 [227] (led by Mingyuan Zang), which enables federated learning-based privacy-preserving in-network analysis. The FLIP4 process includes the steps outlined in Figure 7.4. Initially, a gateway (represented as a switch) is initialised with in-network ML using the Planter framework trained by the local traffic dataset (Steps ❶ to ❹). The local model undergoes periodic updates facilitated by the P4Pir workflow. Concurrently, on the server side, a global model is maintained to aggregate individual local models from multiple gateways, with the parameters of each trained model transmitted to the server under differential privacy (Step ❺). Upon receiving updated information from all gateways, the aggregator employs federated learning techniques to average parameters and update the global model. This global model's parameters are then sent back to each IoT gateway for local model updates (Step ❻). Planter on each gateway

maps the updated model to table rules (Step ❼), and P4Pir inserts these new rules into the data plane pipeline at runtime (Step ❽). Within this framework, IoT gateways efficiently handle traffic with minimal delay and a high throughput. Edge servers are responsible for managing the storage and training of localised data. The models trained locally are consolidated in the cloud to create a global model, and to address privacy concerns, noise is introduced to any uploaded models' weights.

FLIP4 is implemented by extending the design presented in the federated learning model [132] and P4Pir [226] (built upon Planter [245]). The FLIP4 prototype can operate on Raspberry Pi using P4Pi-v.0.0.3, Dell Edge Gateway (Dell EMC Edge Gateway 5200) utilising BMv2, and APS-Networks BF6064X equipped with Intel Tofino chipset using Barefoot's SDE 9.6.0. The controller and server functionalities are implemented in Python.

In the ML performance evaluation of this prototype, Mininet is used to emulate FLIP4 performance on multiple nodes (3 nodes in this case). Two publicly available IoT datasets, *CICIDS 2017* [175] and *IoT Sentinel* [137], are used in this evaluation. This evaluation involved comparing the inference performance of FLIP4's XGBoost with the state-of-the-art work [160], implemented in an in-network manner using BNN. The evaluation results demonstrate that the aggregated global model of in-network XGBoost within FLIP4 provides better accuracy in both datasets compared to previous federated learning methodologies [160, 131]. In the *IoT Sentinel* dataset, the in-network XGBoost model outperforms both decision tree-based [131] and BNN-based [160] works, achieving a 2%-3% higher accuracy than these approaches. Similarly, in the *CICIDS 2017* dataset, the FLIP4-based XGBoost has a similar performance compared to the BNN-based design [160], while demonstrating a 17% higher accuracy than the

decision tree-based federated learning method [131]. We also show that differential privacy in FLIP4 leads to a 4% decrease in accuracy loss, presenting an accepted accuracy cost for privacy-preserving transmission of model weights.

7.3 Financial Market Prediction

In addition to the previously discussed applications, and beyond ordering the priority of transactions in §5.8.2, in-network ML can be directly applied to predict future stock price movement in High-frequency trading (HFT). HFT is a form of algorithmic trading that involves placing a large number of orders swiftly while promptly reacting to evolving market conditions [74, 73]. The emergence and development of artificial intelligence have attracted the widespread adoption of ML algorithms into the field of HFT and show advanced performance [107, 92]. One example is the effective use of ML to predict forthcoming price shifts through the analysis of market microstructure signals extracted from Market by Order (MBO) data streams [108, 148, 234, 203]. Nonetheless, the complexity of ML models and the location of server-based deployment introduces additional processing and transmission time, which significantly impacts the generated profits, thereby generating a need for reducing latency across the current ML-based trading process. With the emergence of network programmability and in-network computing, ML models can be realised within the data path by conducting inference on the programmable network devices with reduced latency and enhanced throughput [169, 218]. In this section, we explore the utilisation of in-network ML inference to reduce latency within this time-sensitive financial application.

7.3.1 Prediction of Future Stock Price Movement

MBO data constitutes an order-based data stream containing trade instructions for stocks [233]. It encompasses an order's timestamp, unique identifier, action (such as adding a new order, cancelling an existing one, or modifying price or quantity), side (buy or sell for the given security), price, and quantity, as shown in Figure 7.5. Our initial investigations reveal that utilising only stateless MBO features in the ML model results in a limited accuracy of approximately 36.7% [90, 91]. To enhance this prediction performance, better features should be explored. Limit order books (LOBs), derived implicitly from MBO data, encompass an assortment of unmatched limit orders awaiting execution at predetermined or superior price levels [75], which are commonly used in ML algorithms to forecast future trends in stock prices [233, 147]. However, constructing LOBs in programmable network devices for ML classification is challenging.

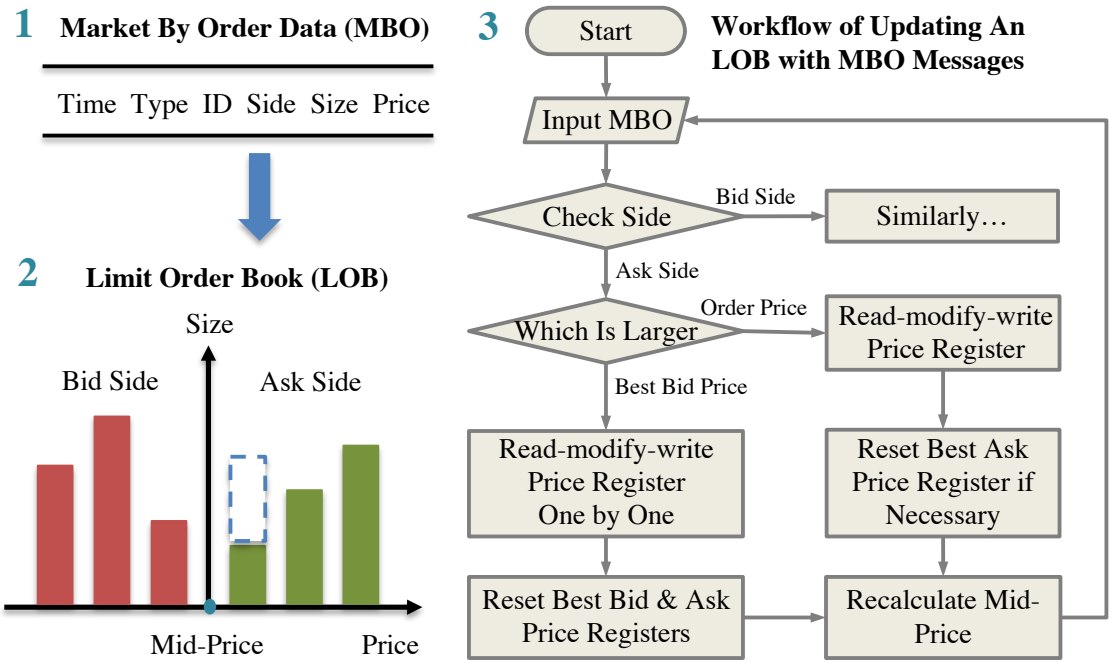


Figure 7.5: 1. Market by Order (MBO) data fields, 2. graphical representation of a limit order book (LOB), and 3. workflow of updating a LOB with MBO messages (source: [90], partially contributed by me).

In this use case, we proposed LOBIN [91] (also referred to as Linnet [90], work led by Xinpeng Hong), which can accelerate the prediction of stock price movements by dynamically constructing and updating LOBs within the programmable data plane. Specifically, as shown in Figure 7.5, upon receiving a new MBO message for a particular stock, LOBIN swiftly updates the corresponding LOB. It then extracts features by gathering data on price and quantity from various levels on both the bid and ask sides of the LOB. These features serve as inputs for in-network ML to forecast trends in future stock prices. Figure 7.6 illustrates the proposed workflow of LOBIN. The workflow firstly constructs LOB using historical market data feeds and trains the ML model (Step ❶). The trained model is then mapped to the data plane, and table entries are created based on the Planter framework (Step ❷). This process involves generating a P4 program that encompasses both the integrated ML inference model and logic related to LOB operations: construction and updating LOB and feature extraction (Step ❸). Subsequently, based on Planter, the generated P4 program is compiled and loaded onto the programmable data plane, while table entries are loaded via the control plane (Step ❹ and ❺). Based on the trained in-network ML model and in-network constructed LOBs, the LOBIN workflow realises the direct future stock price prediction within the data plane.

To further enhance LOBIN's performance, the hybrid deployment system can be applied [88], as elaborated in Chapter 5. Under the hybrid deployment, the prediction of price movements within the switch is labelled only when the confidence level of the prediction is high. Conversely, if the confidence level is low, the MBO data would be forwarded to a server for prediction using a larger and more complex model.

LOBIN is implemented based on the Planter and IIsy framework. By incorpo-

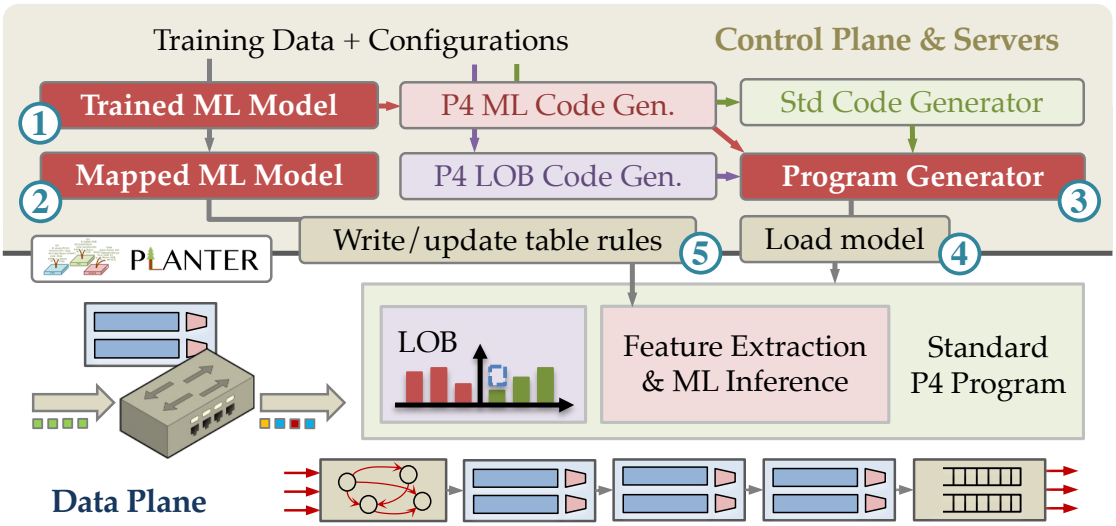


Figure 7.6: System design of LOBIN (source: [91], partially contributed by me).

rating a newly designed LOBIN-oriented `Common P4` module, we integrate LOB code with ML inference code generated through Planter [245]. In the data plane, to update the LOB for each incoming MBO, LOBIN employs registers to keep prices. On certain hardware targets, registers can be accessed multiple times in the pipeline, with reading and writing each constituting an access (resulting in two accesses). To implement price updates, LOBIN uses recirculation, involving one pass for register reading and a subsequent pass for writing. LOBIN is prototyped on both an APS-Networks BF6064T-X Intel Tofino switch running SDE 9.4.0 and a software environment through SDE 9.9.0 deployed on a server (for Tofino 2 emulation). The servers are equipped with an AMD EPYC 7302P CPU, 256GB DDR4 RAM, operating on Ubuntu 20.04 LTS. The switch and server are interconnected via NVIDIA ConnectX-5 100G NICs using direct-attach cables.

We assess LOBIN's performance using NASDAQ's Historical TotalView-ITCH sample data feeds [10]. Specifically, MBO messages for various stocks on the 27th of March 2019 are extracted, representing the most recent available data in this source. Within this dataset, three stocks from distinct sectors,

PYPL (Financials Sector), PEP (Consumer Staples Sector), and GOOGL (Communication Sector), are chosen for evaluation due to their significant market capitalisation [91]. Our evaluation reveals that across diverse models and stocks, LOBIN demonstrates an average accuracy decline of 16.47% (51.04%←67.53%) and an average F1-score decrease of 15.60% (42.28%←57.87%) when operating on Tofino compared to server benchmarks. LOBIN achieves microsecond-level latency, showing over a 10% latency improvement compared to the NASDAQ order-matching server benchmark [28].

7.4 Load Balancing

Previous use cases predominantly focus on classification. However, there is a scarcity of adoption and evaluation of in-network ML for other types of tasks, such as control. This section extends the type of applied in-network ML algorithm to reinforcement learning and introduces it to a load-balancing use case.

Network traffic is dynamic by nature. The increasing use of cloud computing and the introduction of more and more networked services mean that traffic patterns are less predictable and more imbalanced than in the past [231]. Furthermore, network topologies are also complex and evolve over time, as a network grows, nodes fail and connectivity changes. The dynamism of the network is a special challenge when trying to satisfy capacity demands and maintain service objectives. In networks where multiple paths exist between source and destination, load balancing aims to assign flows to different paths in a manner that optimises network utilisation [231], and consequently user experience.

The classic load balancing algorithm, equal-cost multi-path (ECMP) [199],

evenly distributes traffic to all available paths using a hash function. While ECMP is simple and feasible within switches, it may suffer imbalance due to hash collisions and varying data rates [231], or global traffic imbalancing in networks with asymmetric topologies [71]. Network telemetry, and in particular in-band telemetry, and the rise of software defined networks, enabled new load balancing solutions such as CONGA [14] and HULA [105]. These solutions gained real-time insights into network utilisation, using a variety of in-network techniques. However, they still use fixed load balancing policies.

Unlike fixed-strategy policies, reinforcement learning has the ability to dynamically adapt to unknown environments. It improves policies by interacting with the environment to find the optimal policy. When an environment changes, reinforcement learning can gradually discover new optimal strategies, adjusted based on trial and error and environmental feedback. Its applicability to complex problems, interactive learning, and autonomous decision-making capabilities make it a promising solution for load balancing challenges. Previous works have applied reinforcement learning to host-based load balancing [161] and controller-based load balancing [250], but the exploration of reinforcement learning for switch-based load balancing remains limited [231].

7.4.1 In-network Q-Learning for Distributed Load Balancing

In this use case, we proposed a solution named QCMP [241], a reinforcement learning-based distributed load balancing solution for network switches. QCMP applies in-network Q-learning, as introduced in Chapter 3.3.4, to adjust the weight of each path, balancing the load across the network. QCMP uses INT to collect congestion and utilisation information across the network, and uses

Q-learning to make decisions. QCMP's distributed switch-based load balancing ensures quick reactions and high scalability, overcoming the limitations of centralised controllers. The reinforcement learning-based decision ensures continuous policy updates by QCMP and adjusting to diverse environments. The implementation of QCMP follows the structure outlined in Figure 3.4, implementing in-network reinforcement learning with the Q-table in the data plane. Most data centres use hardware switches for high throughput and low latency traffic forwarding. On hardware targets, the register-based solution is harder to implement and has limited scalability. Therefore, we focus on M/A-based Q-learning (Figure 3.4 (b) in Chapter 3.3.4) as a leading example for QCMP.

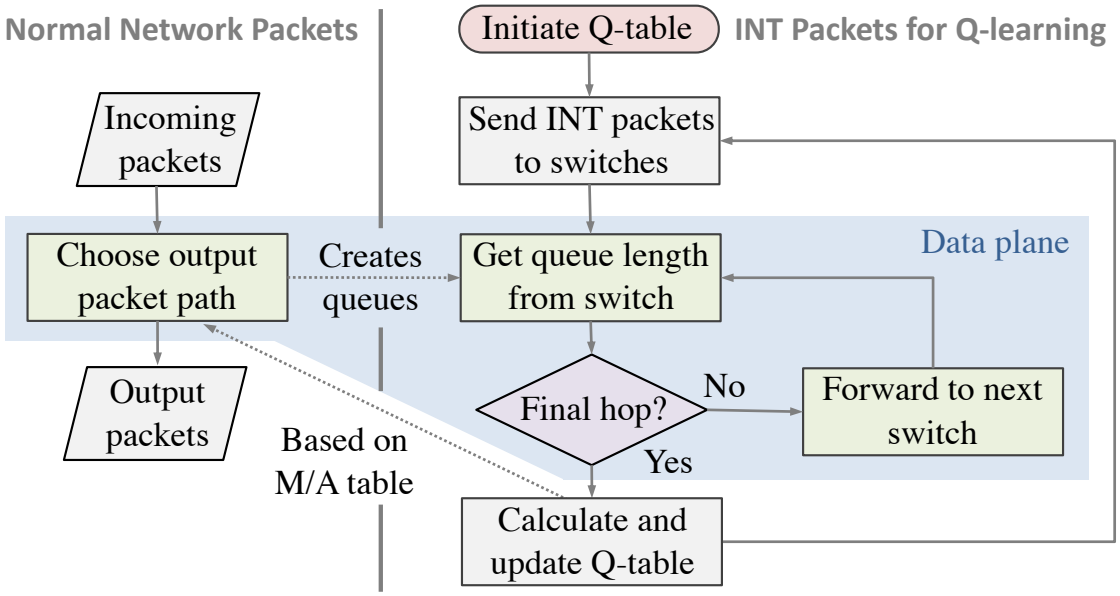


Figure 7.7: System design of QCMP.

A high-level view of QCMP's implementation is shown in Figure 7.7. The operation distinguishes between two types of packets: normal traffic and INT messages[1]. Normal network traffic, on the left of Figure 7.7, is forwarded based on path weights. Parsed packets are forwarded to ports based on routing tables (routing-related) and weights (load balancing-related). INT packets, on the right

[1]Use of dedicated INT packets is for illustration purposes

of Figure 7.7, contain queue lengths information, used to update path weights. This information is sent to the control plane for Q-value updates. The Q-table and the match-action table which stores path weights, are initially set to equal weight. When INT packets pass through a switch, the switch updates queue length in the INT header. For coverage, we assume that source routing is used, as in [194]. INT packets are mirrored to the CPU and the controller calculates a new Q-value and updates the Q-table in the control plane. Based on the queue lengths from INT packets, path weights are updated in the M/A table.

Both in-network Q-learning strategies are implemented in BMv2 software switch using P4 with v1model architecture. The M/A table-based Q-learning approach is also implemented on an Intel Tofino switch-ASIC (APS-Networks BF6064X, SDE 9.5.0) using TNA architecture. Emulation of QCMP uses Mininet [240]. The switch control plane is written in Python and supports P4Runtime. QCMP hardware performance test is run on Intel Tofino. System level operation is evaluated using BMv2, and performance is compared with ECMP. QCMP is evaluated on a 3-tier Fat-Tree topology with two spine switches connected to a pod. Within a pod, two aggregation switches are connected to two top-of-rack (ToR) switches. In the evaluation, the queue length rate on the simulator ranges from [0, $N_{queue} = 100$] and we set $N_q = 10$. In the evaluated network topology, each switch is connected to two other switches, which means $N_p = 2$. The path changes weight k is initialised to 5 and the constant x in the reward equation is set as 50. The model starts with a high learning rate and exploration rate for rapid explorative learning but reduces to lower values for a more exploitative approach that converges accurately to the optimal policy. The exploration rate is limited to a minimum of 0.1 to overcome non-optimal states, a discount factor of 0.25 to lay more emphasis on immediate reward and less on long-term reward.

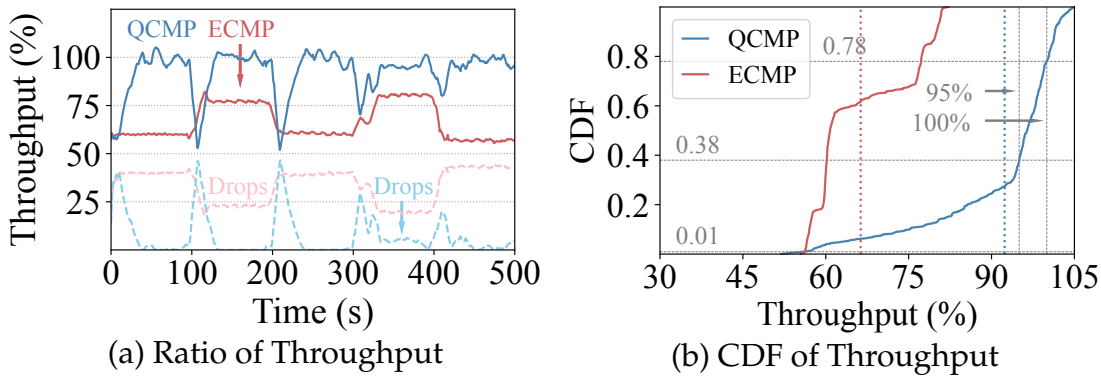


Figure 7.8: System performance comparison between QCMP and ECMP.

Figure 7.8 shows the performance of QCMP compared to ECMP over a 500-second episode, with the output port queue rates changing every 100 seconds. A 10-second moving average is used for both sets of results in order to limit the effect of noise due to the random variations of the hash function. As shown in Figure 7.8 (a), QCMP's throughput starts at the same level as ECMP, with around 60%, but it learns the optimal path weights to achieve a throughput of 100% within 30 seconds. At 100 seconds, when the queue rates are switched, the throughput suddenly drops to around 50%, which is slightly worse than ECMP for a short time. Note that this happens only due to the significant artificial change in port rates in our experiment. At 400 seconds, the optimal queue weights are re-found within 14 seconds. This is shorter than the first 3 times of learning because the Q-table already contains a trained policy[2]. These show that QCMP adapts to network conditions changes.

CDFs of QCMP and ECMP throughput are shown in Figure 7.8 (b). The average throughput of QCMP is 92.6% (including rate change events), whilst ECMP achieves 66%, showing significant improvement. Our experiments show that the gap in performance grows as networks become more complex. QCMP achieves over 95% of input packet rate at 62% of the time, and matches 100% input packet

[2]The scale of seconds is intentionally set, to avoid fluctuations.

rate at 22% of the time, outperforming ECMP 99% of the time.

QCMP's implementation on Tofino is tested using a snake configuration, utilising $64 \times 100GE$ ports. Our measurement shows that QCMP achieves full line rate on all 64 ports, with no packet drops. The resources used on the switch are negligible, less than 1% of memory resources and two pipeline stages.

7.5 Summary

In-network ML is increasingly recognised as a viable computational paradigm for ML-based applications. This chapter applied in-network ML to four distinct use cases. These applications include not only anomaly detection (§7.1) and traffic classification (§7.2) but also extend to services like financial market prediction (§7.3) and load balancing (§7.4). These applications show that in-network ML models have a unique deployment location, good inference performance, high throughput, low latency, and the capability for early traffic termination, making it more advent than conventional solutions. The evaluation of these use cases has shown the potential of applying proposed in-network ML in data-intensive and time-sensitive use cases.

CHAPTER 8
CONCLUSION

This chapter summarises the contributions of this research (§8.1). These efforts are a large step toward in-network ML as a practical service. Additionally, this chapter discusses the limitations (§8.2) & future work of in-network ML (§8.3), and ends with concluding remarks (§8.4).

8.1 Summary of Contributions

In this thesis, I have thoroughly addressed the challenges in mapping in-network ML to programmable network devices. These efforts, using a bottom-up approach, contribute to the practical in-network ML:

In-Network ML Mapping. To address the challenge of algorithm mapping, I proposed general methodologies, derived from implementation experiences. These mapping methodologies serve as a guidance for in-network ML realisation and drive the implementation of new models. By applying these methodologies, I realised over ten in-network ML models with more than fifty variations. Experimental results demonstrate that these mappings maintain accuracy with no or only minor accuracy loss. Compared to existing work, the new mappings can achieve up to two orders of magnitude reduction in table entries and 50% fewer pipeline stages, as well as maintain good inference accuracy and high system performance on multiple programmable network devices. This offers abundant and efficient in-network ML algorithm mappings for practical applications.

Rapid Prototyping Framework. To realise the proposed in-network ML mappings, I introduced Planter framework for rapid prototyping. Planter integrates

the workflow of ML model training, data and control plane code generation, and mapped model verification. The modular design of Planter allows plug-and-play deployment and enables support for new hardware targets, algorithms, and use cases without the need for redesigning the whole framework. Using this framework, users can implement the in-network ML algorithm mapping within minutes. To date, Planter has already been used by five other researchers, showing good applicability.

Hybrid Deployment System. To overcome the limitation of inference performance for in-network ML models, I applied the concept of hybrid deployment to IIsy, which employs a small in-network ML model on the network device and large ML models over the backend server. The high-confidence decisions will be directly made in the network and low-confidence samples will be forwarded to servers for further processing. With the help of hybrid deployment, the in-network ML system can achieve close to optimal classification results, with minimal resource overhead on programmable network devices, while reducing the load on the backend server by 70% for the data-intensive use case, and saving 50% latency on average for the time-sensitive application.

Distributed Deployment Framework. Even though hybrid deployment provides high inference performance, the in-network ML is still limited in size. To further scale the size and performance of the in-network model, I focused on distributed in-network computing and proposed a DINC framework. DINC employs an integer linear programming model (strategy planner) for deployment optimisation on multi-path networks with resource-constrained network devices. DINC also includes a data plane program slicer, a code generator, and a tester, which automates the slicing and generation of data plane programs for targeted hardware nodes. By jointly utilising resources among network devices,

larger available resources can be accessed and better service performance can be provided. It is shown that DINC is an important first step towards the efficient utilisation of data plane resources through distributed in-network computing.

In-network ML Applications. The thesis also explored the application of proposed solutions in four use cases, showing the benefit of applying in-network ML. Moreover, some in-network ML solutions in these applications go beyond the above designs. For example, one incorporated features like a continuous update mechanism in Planter to address data shift and model updates, and the other proposed federated in-network ML based on Planter to scale the solution while safeguarding local data privacy. These applications have demonstrated the effectiveness of proposed mappings and frameworks, indicating the potential of applying in-network ML algorithms across diverse use cases.

8.2 Limitations

Beyond the contributions, there are two main limitations (scope):

1. Not all ML models are suitable for running on programmable network devices at this stage. While this study mitigates resource constraints through novel mappings and deployment strategies, it does not claim universal support for all ML models. One example is deep learning, which can even exhaust resources easily (e.g., Video RAM) on a GPU.
2. This thesis aims to implement ML algorithms utilising P4 on programmable network devices with PISA architecture. Other potential network targets, whether present or future, may employ P4 or different programming languages, following similar or distinct architectures. The scope

of this thesis does not directly extend to implementing in-network ML algorithms on alternative architectures, such as those not RMT-based or using languages like C and Domino [187]. However, the outcomes of this research can contribute to this endeavour, by applying the introduced concepts and frameworks.

8.3 Future Work

In the future, based on outcomes from this research and the above limitations, several directions can be further researched:

Training In-network ML Models. Current training of in-network ML algorithms mainly relies on standard libraries such as scikit-learn and PyTorch. However, there's room for further optimisation of ML structures and their training processes to better align with the data plane architecture and data structures. While some related works have explored training methods, such as the binary decision tree [34], federated learning [132], and knowledge distillation [43], many other models and training enhancements remain unexplored. One example lies in the exploration of leveraging deep learning-assisted tree model training to enhance accuracy and reduce resource consumption. These improvements can enhance the performance and scalability of in-network ML while leveraging existing mapping techniques.

In-network ML Models Mapping. The types and complexities of ML models continue to advance, and it is the fact that there is an increased demand for sophisticated models in various use cases. Exploring efficient solutions for mapping various emerging ML models, especially deep neural networks, to network

devices is an ongoing work and is a future direction. For example, how to efficiently offload a large language model to programmable network devices is still unsolved. The support of these models can empower in-network ML with high inference performance, and attract broader use cases to apply in-network ML.

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. Even though several in-network ML use cases have been explored in Chapter 7, compared to traditional ML, the range of in-network ML use cases is still very narrow, leading to limited adoption in practice. For example, it is worth exploring the application of in-network ML in enhancing IoT devices with low-latency ML services in smart cities and in improving the performance of next-generation cellular (6G) & satellite networks. Applying in-network ML beyond the networking domain, to fields such as natural language processing and computer vision, is also worth exploring. At a certain point, the emergence of new applications will also, in turn, drive in-network ML improvements and stimulate novel ideas.

In-network ML Targets. Currently, in-network ML algorithms are predominantly deployed on devices such as programmable switches, SmartNICs, and FPGA, and prototyped on software targets such as BMv2 and T4P4s. However, hardware targets exhibit various limitations in terms of resources such as memory, operations, and pipeline stages. Designing future programmable network devices with expanded functionality, without compromising network performance, presents a significant challenge. The enhancement of hardware capabilities can significantly broaden the range of feasible models and their scalability & performance. However few works have explored modified ASICs for in-network ML, and further exploration in this area is needed.

Power and Carbon Measurement. Current in-network ML measurements primarily focus on metrics such as inference accuracy (e.g., accuracy and F1 score), resource efficiency (e.g., memory usage and stage consumption), and system performance (e.g., throughput and latency). However, there has been limited measurement or modelling of power consumption and carbon emissions of in-network ML models, despite known power efficiency of in-network computing [201]. Power consumption is important to the cost-effectiveness of in-network ML models and can significantly influence their commercial value. Additionally, modelling carbon emissions (which are location dependent) can help identify preferred deployment locations for in-network ML, minimising greenhouse gas emissions and promoting environmental sustainability.

Multi-model and Multi-application. Current in-network ML research mainly focuses on scenarios where a single model is deployed on a network device or system for a specific task and user. While previous chapters (e.g. Chapter 6) touched on some related topics, further research is needed to develop in-network ML systems and mechanisms for multi-model and multi-application deployments. For example, combining multiple models based on their strength handling specific features to form ensemble models beyond the ones presented in this thesis. Additionally, supporting concurrent applications requires extending current solutions to manage virtualised applications and their tenancy. These enhancements would improve both the performance and cloud environment's applicability of in-network ML.

Security of In-network ML. Although in-network ML has been widely used in cybersecurity applications, the security of in-network ML itself remains largely unexplored. Examples of some areas worth exploring include enhancing models' explainability, improving system resilience to security attacks, and developing

robust methods for managing in-network ML failure modes. Further research of these aspects can strengthen the security and reliability of in-network ML, and increase users' confidence in its practical deployment.

8.4 Concluding Remarks

Looking back on this work, the thesis offers solutions for offloading ML algorithms to programmable network devices. The proposed three mapping methodologies enable a wide range of algorithm mappings. The designed Planter framework simplifies the deployment workflow and easily fits changes and new setups. The introduced hybrid and distributed deployment strategies scale the model size and improve ML performance. The applications of in-network ML showcase the benefit of proposed solutions to practical use cases. Moreover, these solutions are becoming the enabler of in-network ML to a larger community through open source. At the end of the thesis, I hope this research can open the door for in-network ML, make in-network ML a practical computing paradigm, and push in-network ML to be an option in parallel to current server-based ML.

BIBLIOGRAPHY

[1] Intel Infrastructure Processing Unit (Intel IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html. Accessed Jan 24, 2024.

[2] Netronome Flow Processor Product Brief. https://www.netronome.com/media/documents/PB_NFP-6000-7-20.pdf. Accessed Dec 25, 2023.

[3] NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. Accessed Dec 25, 2023.

[4] Open Tofino. https://github.com/barefootnetworks/Open-Tofino. Accessed Dec 25, 2023.

[5] p4c. https://github.com/p4lang/p4c. Accessed Dec 25, 2023.

[6] P4Runtime Specification. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html. Accessed Dec 25, 2023.

[7] The Pensando Distributed Services Platform. https://pensando.io/our-platform/. Accessed Dec 25, 2023.

[8] The Reference P4 Software Switch. https://github.com/p4lang/behavioral-model. Accessed Dec 25, 2023.

[9] Version 1.0 Switch Architecture Model. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4. Accessed Dec 25, 2023.

[10] Nasdaq ITCH Data Source. 2020. https://emi.nasdaq.com/ITCH/Nasdaq%20ITCH/. Accessed Dec 25, 2023.

[11] P4 Portable NIC Architecture (PNA), May 2021. https://p4.org/p4-spec/docs/PNA.html. Accessed Dec 25, 2023.

[12] Hamad Al-Mohannadi, Qublai Mirza, Anitta Namanya, Irfan Awan, Andrea Cullen, and Jules Disso. Cyber-Attack Modeling Analysis Techniques: An Overview. In *2016 IEEE 4th international conference on future internet of things and cloud workshops (FiCloudW)*, pages 69–76. IEEE, 2016.

[13] Abdullah Alanazi. Using Machine Learning for Healthcare Challenges and Opportunities. *Informatics in Medicine Unlocked*, 30:100924, 2022.

[14] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, et al. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[15] Ahamed Aljuhani. Machine Learning Approaches for Combating Distributed Denial of Service Attacks in Modern Networking Environments. *IEEE Access*, 9:42236–42264, 2021.

[16] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. NanoTransport: A Low-Latency, Programmable Transport Layer for NICs. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, SOSR '21, page 13–26, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Zein Ashi, Laila Aburashed, Mohammad Al-Fawa'reh, and Malek Qasaimeh. Fast and Reliable DDoS Detection using Dimensionality Reduction and Machine Learning. In *2020 15th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 1–10. IEEE, 2020.

[18] Victor 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/`. Accessed Dec 25, 2023.

[19] Jiasong Bai, Menghao Zhang, Guanyu Li, Chang Liu, Mingwei Xu, and Hongxin Hu. FastFE: Accelerating ML-based Traffic Analysis with Programmable Switches. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, pages 1–7, 2020.

[20] Matthew Baron, Jonathan Brogaard, Björn Hagströmer, and Andrei Kirilenko. Risk and Return in High-Frequency Trading. *Journal of Financial and Quantitative Analysis*, 54(3):993–1024, 2019.

[21] Diogo Barradas, Nuno Santos, Luıs Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. 2021.

[22] FRS Bayes. An Essay Towards Solving A Problem in The Doctrine of Chances. *Biometrika*, 45(3-4):296–315, 1958.

[23] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael 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*, pages 662–680, 2020.

[24] Antoine Bernabeu and Noa Zilberman. In-network Computing - Towards Language Processing within Network Devices. Report: End of Study Internship, 2020.

[25] Siddhartha Bhattacharyya, Sanjeev Jha, Kurian Tharakunnel, and J Christopher Westland. Data Mining for Credit Card Fraud: A Comparative Study. *Decision support systems*, 50(3):602–613, 2011.

[26] Christopher M Bishop and Nasser M Nasrabadi. *Pattern Recognition and Machine Learning*, volume 4. Springer, 2006.

[27] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. Switches for HIRE: Resource Scheduling for Data Center In-Network Computing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 268–285, 2021.

[28] Julius Bonart and Martin D Gould. Latency and Liquidity Provision in a Limit Order Book. *Quantitative Finance*, 17(10):1601–1616, 2017.

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

[30] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[31] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications and Research Opportunities. *Journal of Internet Services and Applications*, 9(1):1–99, 2018.

[32] Gordon Brebner. Expanding the P4 universe. Website, May 2022.

https://opennetworking.org/wp-content/uploads/2022/05/Expanding-the-P4-universe.pdf. Accessed Dec 25, 2023.

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

[34] Sung-Hyuk Cha and Charles C Tappert. A Genetic Algorithm for Constructing Compact Binary Decision Trees. *Journal of pattern recognition research*, 4(1):1–13, 2009.

[35] Yin-Wen Chang, Cho-Jui Hsieh, Kai-Wei Chang, Michael Ringgaard, and Chih-Jen Lin. Training and Testing Low-degree Polynomial Data Mappings via Linear SVM. *Journal of Machine Learning Research*, 11(4), 2010.

[36] Efstratios Chatzoglou, Georgios Kambourakis, and Constantinos Kolias. Empirical Evaluation of Attacks Against IEEE 802.11 Enterprise Networks: The AWID3 Dataset. *IEEE Access*, 9:34188–34205, 2021.

[37] Hongyi Chen, Damu Ding, Changgang Zheng, Rana Abu Bakar, Filippo Cugini, et al. SmartEdge. pages 1–57, 2024. D4.1 Design of Dynamic & Secure Swarm Networking, GA 101092908.

[38] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[39] Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. SPEED: Resource-Efficient and High-Performance Deployment for Data Plane Programs. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2020.

[40] Xiang Chen, Hongyan Liu, Qingjiang Xiao, Kaiwei Guo, Tingxin Sun, Xiang Ling, Xuan Liu, Qun Huang, Dong Zhang, Haifeng Zhou, et al. Toward Low-Overhead Inter-Switch Coordination in Network-Wide Data Plane Program Deployment. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 370–380. IEEE, 2022.

[41] Yang Chen. *Network Update and Service Chain Management in Software Defined Networks*. Temple University, 2020.

[42] Yang Chen and Jie Wu. NFV Middlebox Placement with Balanced Set-up

Cost and Bandwidth Consumption. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018.

[43] Jang Hyun Cho and Bharath Hariharan. On the Efficacy of Knowledge Distillation. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4794–4802, 2019.

[44] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 1–14, 2017.

[45] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392. USENIX Association, 2016.

[46] Bruno Loureiro Coelho. Detecting DoS Attacks Utilizing Random Forests in Programmable Data Planes. Bachelor Thesis, Federal University of Rio Grande do Sul, 2020.

[47] P4 Language Consortium. Performance of BMv2, 2020. `https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md`. Accessed Dec 25, 2023.

[48] Ryan A Cooke and Suhaib A Fahmy. A Model for Distributed In-Network and Near-Edge Computing with Heterogeneous Hardware. *Future Generation Computer Systems*, 105:395–409, 2020.

[49] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine learning*, 20(3):273–297, 1995.

[50] Lei Cui, Youyang Qu, Longxiang Gao, Gang Xie, and Shui Yu. Detecting False Data Attacks Using Machine Learning Techniques in Smart Grid: A Survey. *Journal of Network and Computer Applications*, 170:102808, 2020.

[51] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, 2020.

[52] Dipankar Dasgupta, Zahid Akhtar, and Sajib Sen. Machine Learning in

Cybersecurity: A Comprehensive Survey. *The Journal of Defense Modeling and Simulation*, 19(1):57–106, 2022.

[53] Dell Technologies. Dell EMC Edge Gateway 5200 Software User's Guide, 2022. https://www.dell.com/support/manuals/en-ae/dell-edge-gateway-5200/egw-5200-software-users-guide. Accessed Mar 5, 2024.

[54] Victor Dey and Louis Columbus. AI and ML: The new frontier for data center innovation and optimization, 2023. https://venturebeat.com/data-infrastructure/ai-and-ml-the-new-frontier-for-data-center-innovation-and-optimization/. Accessed Jan 21, 2024.

[55] Rabiyatou Diouf, Edouard Ngor Sarr, Ousmane Sall, Babiga Birregah, Mamadou Bousso, and Sény Ndiaye Mbaye. Web Scraping: State-of-the-Art and Areas of Application. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 6040–6042. IEEE, 2019.

[56] Matthew F Dixon, Igor Halperin, and Paul Bilokon. *Machine Learning in Finance*, volume 1170. Springer, 2020.

[57] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, April 2014.

[58] Joaquín Delgado Fernández, Sergio Potenciano Menci, Chul Min Lee, Alexander Rieger, and Gilbert Fridgen. Privacy-Preserving Federated Learning for Residential Short-Term Load Forecasting. *Applied energy*, 326:119915, 2022.

[59] Mohamed Amine Ferrag, Othmane Friha, Djallel Hamouda, Leandros Maglaras, and Helge Janicke. Edge-IIoTset: A New Comprehensive Realistic Cyber Security Dataset of IoT and IIoT Applications: Centralized and Federated Learning. *IEEE Access*, 10:40281–40306, 2022.

[60] Ronald A Fisher. The Use of Multiple Measurements in Taxonomic Problems. *Annals of eugenics*, 7(2):179–188, 1936.

[61] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[62] Evelyn Fix and Joseph Lawson Hodges. Discriminatory Analysis. Non-

parametric Discrimination: Consistency Properties. *International Statistical Review/Revue Internationale de Statistique*, 57(3):238–247, 1989.

[63] Yoav Freund and Robert E Schapire. A Decision-Theoretic Generalization of On-Line Learning and An Application to Boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[64] Kurt Friday, Elie Kfoury, Elias Bou-Harb, and Jorge Crichigno. INC: In-Network Classification of Botnet Propagation at Line Rate. In *European Symposium on Research in Computer Security*, pages 551–569. Springer, 2022.

[65] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of statistical software*, 33(1):1, 2010.

[66] Roy Friedman, Or Goaz, and Ori Rottenstreich. Clustreams: Data Plane Clustering. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, SOSR '21, page 101–107, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Karl Pearson F.R.S. LIII. On Lines and Planes of Closest Fit to Systems of Points in Space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[68] Alex Galakatos, Andrew Crotty, and Tim Kraska. *Distributed Machine Learning*, pages 1196–1201. Springer New York, New York, NY, 2018.

[69] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-Adversarial Training of Neural Networks. *Journal of machine learning research*, 17(1):2096–2030, 2016.

[70] Joseph Gardiner and Shishir Nagaraja. On the Security of Machine Learning in Malware C&C Detection: A Survey. *ACM Computing Surveys (CSUR)*, 49(3):1–39, 2016.

[71] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 350–361, 2011.

[72] Leonardo Reinehr Gobatto, Pablo Rodrigues, Mateus Saquetti Pereira de Carvalho Tirone, Weverton Luis da Costa Cordeiro, and José Ro-

drigo Furlanetto Azambuja. Programmable Data Planes meets In-Network Computing: A Review of the State of the Art and Prospective Directions. *Journal of Integrated Circuits and Systems*, 16(2):1–8, 2021.

[73] Michael A Goldstein, Pavitra Kumar, and Frank C Graves. Computerized and High-Frequency Trading. *Financial Review*, 49(2):177–202, 2014.

[74] Peter Gomber and Martin Haferkorn. High Frequency Trading. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 1–9. IGI Global, 2015.

[75] Martin D Gould, Mason A Porter, Stacy Williams, Mark McDonald, Daniel J Fenn, and Sam D Howison. Limit Order Books. *Quantitative Finance*, 13(11):1709–1742, 2013.

[76] Jane Street Group. Jane Street Market Prediction, 2020. `https://www.kaggle.com/c/jane-street-market-prediction`. Accessed Dec 25, 2023.

[77] Renjie Gu, Chaoyue Niu, Fan Wu, Guihai Chen, Chun Hu, Chengfei Lyu, and Zhihua Wu. From Server-Based to Client-Based Machine Learning: A Comprehensive Survey. *ACM Computing Surveys (CSUR)*, 54(1):1–36, 2021.

[78] Vladimir Gurevich and Andy Fingerhut. P4-16 Programming for Intel Tofino Using Intel P4 Studio. In *2021 P4 Workshop*, 2021.

[79] Craig Gutterman, Katherine Guo, Sarthak Arora, Xiaoyang Wang, Les Wu, Ethan Katz-Bassett, and Gil Zussman. Requet: Real-Time QoE Detection for Encrypted YouTube Traffic. In *Proceedings of the 10th ACM Multimedia Systems Conference*, MMSys '19, page 48–59, New York, NY, USA, 2019. Association for Computing Machinery.

[80] Hasanin Harkous, Michael Jarschel, Mu He, Rastin Pries, and Wolfgang Kellerer. P8: P4 With Predictable Packet Processing Performance. *IEEE Transactions on Network and Service Management*, 18(3):2846–2859, 2020.

[81] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. *Journal of Network and Computer Applications*, 212:103561, 2023.

[82] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril,

Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[83] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[84] Masoud Hemmatpour, Changgang Zheng, and Noa Zilberman. E-Commerce Bot Traffic: In-Network Impact, Detection, and Mitigation. In *7th Conference on Innovation in Clouds, Internet and Networks (ICIN)*. 2024.

[85] Tin Kam Ho. Random Decision Forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[86] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 9(8):1735–1780, 1997.

[87] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular Switch Programming Under Resource Constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 193–207, Renton, WA, April 2022. USENIX Association.

[88] Xinpeng Hong, Changgang Zheng, Joshua Lilley, Stefan Zohren, and Noa Zilberman. Accelerating Machine Learning for Trading Using Programmable Switches. In *27th European Conference on Artificial Intelligence (ECAI)*. IOS Press, 2024.

[89] Xinpeng Hong, Changgang Zheng, and Noa Zilberman. In-network machine learning for real-time transaction fraud detection. In *27th European Conference on Artificial Intelligence (ECAI)*. IOS Press, 2024.

[90] Xinpeng Hong, Changgang Zheng, Stefan Zohren, and Noa Zilberman. Linnet: Limit Order Books Within Switches. In *Proceedings of the SIGCOMM'22 Poster and Demo Sessions*, pages 37–39. 2022.

[91] Xinpeng Hong, Changgang Zheng, Stefan Zohren, and Noa Zilberman. LOBIN: In-Network Machine Learning for Limit Order Books. In *2023 IEEE 24th International Conference on High Performance Switching and Routing (HPSR)*, pages 159–166. IEEE, 2023.

[92] Boming Huang, Yuxiang Huan, Li Da Xu, Lirong Zheng, and Zhuo Zou. Automated Trading Systems Statistical and Machine Learning Methods and Hardware Implementation: A Survey. *Enterprise Information Systems*, 13(1):132–144, 2019.

[93] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated Machine Learning: Methods, Systems, Challenges*. Springer Nature, 2019.

[94] C-L Hwang and Abu Syed Md Masud. *Multiple Objective Decision Making — Methods and Applications: A State-of-the-Art Survey*, volume 164. Springer Science & Business Media, 2012.

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

[96] Wan Nur Hidayah Ibrahim, Syahid Anuar, Ali Selamat, Ondrej Krejcar, Rubén González Crespo, Enrique Herrera-Viedma, and Hamido Fujita. Multilayer Framework for Botnet Detection Using Machine Learning Algorithms. *IEEE Access*, 9:48753–48768, 2021.

[97] Intel. P4_16 Intel® Tofino™ Native Architecture – Public Version, 2021. `https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf`. Accessed Dec 25, 2023.

[98] Maryam Jalalitabar, Evrim Guler, Danyang Zheng, Guangchun Luo, Ling Tian, and Xiaojun Cao. Embedding Dependence-Aware Service Function Chains. *Journal of Optical Communications and Networking*, 10(8):C64–C74, 2018.

[99] Maryam Jalalitabar, Yang Wang, and Xiaojun Cao. Branching-Aware Service Function Placement and Routing in Network Function Virtualization. In *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–6. IEEE, 2019.

[100] Weirong Jiang and Viktor K Prasanna. Scalable Packet Classification on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(9):1668–1680, 2011.

[101] Yu Jiang, Xiaolong Xu, Kunda Lin, and Weihua Duan. TSC-ECFA: A trusted service composition scheme for edge cloud. In *2021 IEEE 27th In-*

ternational Conference on Parallel and Distributed Systems (ICPADS), pages 58–65. IEEE, 2021.

[102] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[103] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[104] Matthews Jose, Kahina Lazri, Jérôme François, and Olivier Festor. NetREC: Network-wide in-network REal-value Computation. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 189–197. IEEE, 2022.

[105] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[106] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *Advances in neural information processing systems*, 30, 2017.

[107] Michael Kearns and Yuriy Nevmyvaka. Machine Learning for Market Microstructure and High Frequency Trading. *High Frequency Trading: New Realities for Traders, Markets, and Regulators*, 2013.

[108] Alec N Kercheval and Yuan Zhang. Modelling High-Frequency Limit Order Book Dynamics with Support Vector Machines. *Quantitative Finance*, 15(8):1315–1329, 2015.

[109] Somayeh Kianpisheh and Tarik Taleb. A Survey on In-Network Computing: Programmable Data Plane and Technology Specific Applications. *IEEE Communications Surveys & Tutorials*, 2022.

[110] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band Network Telemetry via Programmable Dataplanes. In *ACM SIGCOMM*, volume 15, pages 1–2, 2015.

[111] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 90–106, 2020.

[112] Suriya Kodeswaran, Mina Tahmasbi Arashloo, Praveen Tammana, and Jennifer Rexford. Tracking P4 Program Execution in the Data Plane. In *Proceedings of the Symposium on SDN Research*, pages 117–122, 2020.

[113] Rakesh Kumar, Mayank Swarnkar, Gaurav Singal, and Neeraj Kumar. IoT Network Traffic Classification Using Machine Learning Algorithms: An Experimental Analysis. *IEEE Internet of Things Journal*, 9(2):989–1008, 2021.

[114] Sándor Laki, Csaba Györgyi, József Pető, Péter Vörös, and Géza Szabó. In-Network Velocity Control of Industrial Robot Arms. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 995–1009, 2022.

[115] Sándor Laki, Radostin Stoyanov, Dávid Kis, Robert Soulé, Péter Vörös, and Noa Zilberman. P4Pi: P4 on Raspberry Pi for Networking Education. *ACM SIGCOMM Computer Communication Review*, 51(3):17–21, 2021.

[116] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761, 2021.

[117] Majd Latah and Levent Toker. Towards an Efficient Anomaly-Based Intrusion Detection for Software-Defined Networks. *IET networks*, 7(6):453–459, 2018.

[118] Changhun Lee and Chiehyeon Lim. From technological development to social advance: A review of Industry 4.0 through machine learning. *Technological Forecasting and Social Change*, 167:120653, 2021.

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

[120] Tae-Hwy Lee, Aman Ullah, and Ran Wang. Bootstrap Aggregating and Random Forest. In *Macroeconomic Forecasting in the Era of Big Data*, pages 389–429. Springer, 2020.

[121] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[122] Qian Li, Jiao Zhang, Tian Pan, Tao Huang, and Yunjie Liu. Data-driven Routing Optimization based on Programmable Data Plane. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2020.

[123] Yiran Li, Kevin Gao, Xin Jin, and Wei Xu. Concerto: Cooperative Network-wide Telemetry with Controllable Error Rate. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 114–121, 2020.

[124] Jiaye Lin, Qing Li, Guorui Xie, Yong Jiang, Zhenhui Yuan, Changlin Jiang, and Yuan Yang. In-Forest: Distributed In-Network Classification with Ensemble Models. In *2023 IEEE 31st International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2023.

[125] Cheng-Yuan Liou, Wei-Chen Cheng, Jiun-Wei Liou, and Daw-Ran Liou. Autoencoder for Words. *Neurocomputing*, 139:84–96, 2014.

[126] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation Forest. In *2008 eighth ieee international conference on data mining*, pages 413–422. IEEE, 2008.

[127] Hongyan Liu, Xiang Chen, Qun Huang, Haifeng Zhou, Dong Zhang, and Chunming Wu. SRA: Switch Resource Aggregation for Application Offloading in Programmable Networks. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[128] Mingyuan Liu, Deyun Gao, Gang Liu, Jingchao He, Lu Jin, Chunliang Zhou, and Fucong Yang. Learning Based Adaptive Network Immune Mechanism to Defense Eavesdropping Attacks. *IEEE Access*, 7:182814–182826, 2019.

[129] Edgar Lopez-Rojas, Ahmad Elmir, and Stefan Axelsson. PaySim: A Financial Mobile Money Simulator for Fraud Detection. In *28th European Modeling and Simulation Symposium, EMSS, Larnaca*, pages 249–255. Dime University of Genoa, 2016.

[130] Fréderic Loui, Csaba Mate, Alexander Gall, and Maxime Wisslé. Project Overview: Router for Academia Research & Education (RARE). 2022.

[131] Heiko Ludwig et al. IBM Federated Learning: an Enterprise Framework White Paper V0.1, 2020.

[132] Samuel Maddock, Graham Cormode, Tianhao Wang, Carsten Maple, and Somesh Jha. Federated Boosted Decision Trees with Differential Privacy. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2249–2263, 2022.

[133] Nick Maynard. Online Payment Fraud: Market Forecasts, Emerging Threats & Segment Analysis 2022-2027. *Juniper Research*, 2022.

[134] Nick McKeown. Software-Defined Networking. *28th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, 17(2):30–32, 2009.

[135] Nick McKeown. *PISA: Protocol Independent Switch Architecture*, 2015. P4 Workshop.

[136] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[137] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, N. Asokan, Ahmad-Reza Sadeghi, and Sasu Tarkoma. IoT SENTINEL: Automated Device-Type Identification for Security Enforcement in IoT. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2177–2184, 2017.

[138] John Minnix. Global Data Center Statistics, 2024. `https://www.cloudzero.com/blog/cloud-service-providers/`. Accessed Jan 21, 2024.

[139] Timothy Prickett 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/`. Accessed Dec 25, 2023.

[140] Nour Moustafa and Jill Slay. UNSW-NB15: A Comprehensive Data set for Network Intrusion Detection systems (UNSW-NB15 Network Data Set). In *2015 military communications and information systems conference (MilCIS)*, pages 1–6. IEEE, 2015.

[141] Anthony J Myles, Robert N Feudale, Yang Liu, Nathaniel A Woody, and Steven D Brown. An Introduction to Decision Tree Modeling. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 18(6):275–285, 2004.

[142] NASDAQ OMX PSX. NASDAQ OMX PSX TotalView-ITCH 5.0, 2014. `http://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/PSXTVITCHSpecification_5.0.pdf`. Accessed Dec 25, 2023.

[143] Vincent Natoli. Kudos for CUDA, 2010. `https://www.hpcwire.com/2010/07/06/kudos_for_cuda/`. Accessed Dec 25, 2023.

[144] John Ashworth Nelder and Robert WM Wedderburn. Generalized Linear Models. *Journal of the Royal Statistical Society: Series A (General)*, 135(3):370–384, 1972.

[145] Andrew Ng et al. Sparse Autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[146] Michael A Nielsen. *Neural Networks and Deep Learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[147] Paraskevi Nousi, Avraam Tsantekidis, Nikolaos Passalis, Adamantios Ntakaris, Juho Kanniainen, Anastasios Tefas, Moncef Gabbouj, and Alexandros Iosifidis. Machine Learning for Forecasting Mid-Price Movements Using Limit Order Book Data. *IEEE Access*, 7:64722–64736, 2019.

[148] Adamantios Ntakaris, Martin Magris, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Benchmark Dataset for Mid-Price Forecasting of Limit Order Book Data with Machine Learning Methods. *Journal of Forecasting*, 37(8):852–866, 2018.

[149] P4 Language Consortium. P4_16 Portable Switch Architecture (PSA). 2018.

[150] The P4.org Architecture Working Group. P4_16 PSA Specification (v1.1), 2017. `https://p4.org/p4-spec/docs/PSA-v1.1.0.html`. Accessed Dec 25, 2023.

[151] Manfred Padberg and Giovanni Rinaldi. A Branch-and-cut Algorithm for the Resolution of Large-scale Symmetric Traveling Salesman Problems. *SIAM review*, 33(1):60–100, 1991.

[152] Francesco Paolucci, Lorenzo De Marinis, Piero Castoldi, and Filippo Cugini. Demonstration of P4 Neural Network Switch. In *2021 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3. IEEE, 2021.

[153] Arti Patle and Deepak Singh Chouhan. SVM Kernel Functions for Classification. In *2013 International Conference on Advances in Technology and Engineering (ICATE)*, pages 1–9. IEEE, 2013.

[154] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[155] Larry Peterson, Carmelo Cascone, Brian O'Connor, Thomas Vachuska, and Bruce Davie. *Software-Defined Networks: A Systems Approach*. Systems Approach, LLC, 2021.

[156] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[157] Nicolas Poggi, David Carrera, Ricard Gavalda, Eduard Ayguadé, and Jordi Torres. A Methodology for the Evaluation of High Response Time on E-commerce Users and Sales. *Information Systems Frontiers*, 16:867–885, 2014.

[158] Rifkie Primartha and Bayu Adhi Tama. Anomaly Detection using Random Forest: A Performance Revisited. In *2017 International conference on data and software engineering (ICoDSE)*, pages 1–6. IEEE, 2017.

[159] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. CatBoost: unbiased boosting with categorical features. *Advances in neural information processing systems*, 31, 2018.

[160] Qiaofeng Qin, Konstantinos Poularakis, Kin K Leung, and Leandros Tassiulas. Line-Speed and Scalable Intrusion Detection at the Network Edge via Federated Learning. In *2020 IFIP Networking Conference (Networking)*, pages 352–360. IEEE, 2020.

[161] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance

and Robustness with Multipath TCP. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.

[162] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[163] Arjan Reurink. Financial Fraud: A Literature Review. *Journal of Economic Surveys*, 32(5):1292–1325, 2018.

[164] Rortune Business Insights. Machine Learning Market Size, Share & COVID-19 Impact, By Enterprise Type, By Deployment, By End-Use Industry, and Regional Forecast, 2023. `https://www.fortunebusinessinsights.com/machine-learning-market-102226`. Accessed Jan 21, 2024.

[165] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside The Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[166] S Rasoul Safavian and David Landgrebe. A Survey of Decision Tree Classifier Methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[167] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[168] SamKnows. 21CN Overview - More Details About BT Wholesale 21CN. `https://availability.samknows.com/broadband/exchanges/21cn_overview`. Accessed Feb 15, 2023.

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

[170] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 150–156, 2017.

[171] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis,

Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808, 2021.

[172] Robert E Schapire. A Brief Introduction to Boosting. In *Ijcai*, volume 99, pages 1401–1406. Citeseer, 1999.

[173] Philipp Schulz, Maximilian Matthe, Henrik Klessig, Meryem Simsek, Gerhard Fettweis, Junaid Ansari, Shehzad Ali Ashraf, Bjoern Almeroth, Jens Voigt, Ines Riedel, et al. Latency Critical IoT Applications in 5G: Perspective on the Design of Radio Interface and Network Architecture. *IEEE Communications Magazine*, 55(2):70–78, 2017.

[174] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 137–150, 2002.

[175] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. *the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, 1:108–116, 2018.

[176] Kartik Shenoy. Credit Card Transactions Fraud Detection Dataset, 2022. `https://www.kaggle.com/datasets/kartik2112/fraud-detection`. Accessed Mar 5, 2024.

[177] Hisham Siddique. Towards In-Network Image Classification for Latency-Critical IoT Applications. Master Thesis, Dalhousie University, 2021.

[178] Hisham Siddique, Miguel Neves, Carson Kuzniar, and Israat Haque. Towards Network-accelerated ML-based Distributed Computer Vision Systems. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 122–129. IEEE, 2021.

[179] Thomas M Siebel. *Digital Transformation: Survive and Thrive in an Era of Mass Extinction*. RosettaBooks, 2019.

[180] Kyle A. Simpson and Dimitrios P. Pezaros. Online RL in the Programmable Dataplane with OPaL. In *17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21)*. 2021.

[181] Simpson, Kyle A and Pezaros, Dimitrios P. Revisiting the Classics: On-

line RL in the Programmable Dataplane. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–10. IEEE, 2022.

[182] Sanskar Singh. Fraudulent Transaction Detection Dataset, 2023. `https://www.kaggle.com/datasets/sanskar457/fraud-transaction-detection`. Accessed Mar 5, 2024.

[183] Giuseppe Siracusano and Roberto Bifulco. In-network Neural Networks. *arXiv preprint arXiv:1801.05731*, 2018.

[184] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 513–533, 2022.

[185] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. Running Neural Networks on the NIC. *arXiv preprint arXiv:2009.02353*, 2020.

[186] Giuseppe Siracusano, Davide Sanvito, Salvator Galea, and Roberto Bifulco. Deep Learning Inference on Commodity Network Interface Cards. In *Proc. Workshop Syst. ML Open Source Softw. NeurIPS*, 2018.

[187] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 15–28, New York, NY, USA, 2016. Association for Computing Machinery.

[188] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.

[189] Kirstine Smith. On the Standard Deviations of Adjusted and Interpolated Values of an Observed Polynomial Function and its Constants and the Guidance they give Towards a Proper Choice of the Distribution of Observations. *Biometrika*, 12(1/2):1–85, 1918.

[190] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 Programs with Vera. In *Proceedings of the*

2018 Conference of the ACM Special Interest Group on Data Communication, pages 518–532, 2018.

[191] S.J. Stolfo, Wei Fan, Wenke Lee, A. Prodromidis, and P.K. Chan. Cost-based Modeling for Fraud and Intrusion Detection: Results from the JAM Project. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 130–144 vol.2, 2000.

[192] Grażyna Suchacka, Alberto Cabri, Stefano Rovetta, and Francesco Masulli. Efficient On-the-Fly Web Bot Detection. *Knowledge-Based Systems*, 223:107074, 2021.

[193] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 571–592, 2021.

[194] Carl A Sunshine. Source Routing in Computer Networks. *ACM SIGCOMM Computer Communication Review*, 7(1):29–33, 1977.

[195] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle 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*, pages 1099–1114, 2022.

[196] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, and Kunle Olukotun. Taurus: An Intelligent Data Plane. *arXiv preprint: 2002.08987*, 2020.

[197] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle 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*, pages 329–342, 2023.

[198] Jiliang Tang, Salem Alelyani, and Huan Liu. Feature Selection for Classification: A Review. *Data classification: Algorithms and applications*, page 37, 2014.

[199] Dave Thaler and C Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. Technical report, 2000.

[200] Robert Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

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

[202] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. LaKe: The Power of In-Network Computing. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2018.

[203] Avraam Tsantekidis, Nikolaos Passalis, Anastasios Tefas, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Using Deep Learning for Price Prediction by Exploiting Stationary Limit Order Book Features. *Applied Soft Computing*, 93:106401, 2020.

[204] Shay Vargaftik and Yaniv Ben-Itzhak. Efficient Multiclass Classification with Duet. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, pages 10–19, 2022.

[205] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[206] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research*, pages 122–135, 2017.

[207] Shuhe Wang, Zili Meng, Chen Sun, Minhu Wang, Mingwei Xu, Jun Bi, Tong Yang, Qun Huang, and Hongxin Hu. SmartChain: Enabling High-Performance Service Chain Partition between SmartNIC and CPU. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2020.

[208] Yang Wang, Gaogang Xie, Zhenyu Li, Peng He, and Kavé Salamatian. Transparent Flow Migration for NFV. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–10, 2016.

[209] Christopher JCH Watkins and Peter Dayan. Q-Learning. *Machine learning*, 8:279–292, 1992.

[210] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P^{2}GO: P4 profile-guided optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 146–152, 2020.

[211] Svante Wold, Kim Esbensen, and Paul Geladi. Principal Component Analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[212] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. Top 10 Algorithms in Data Mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[213] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. Programmable Switches for in-Networking Classification. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[214] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. MAP4: A Pragmatic Framework for In-Network Machine Learning Traffic Classification. *IEEE Transactions on Network and Service Management*, 2022.

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

[216] Xilinx. Xilinx OpenNIC Shell, Accessed on 04/15/2023. `https://github.com/Xilinx/open-nic`. Accessed Dec 25, 2023.

[217] Xilinx. Vitis Networking P4 User Guide, Accessed on 04/16/2023. `https://docs.xilinx.com/r/en-US/ug1308-vitis-p4-user-guide/Target-Architecture`. Accessed Dec 25, 2023.

[218] Zhaoqi Xiong and Noa Zilberman. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*, pages 25–33, 2019.

[219] Haitao Xu, Zhao Li, Chen Chu, Yuanmi Chen, Yifan Yang, Haifeng Lu, Haining Wang, and Angelos Stavrou. Detecting and Characterizing Web Bot Traffic in a Large E-commerce Marketplace. In *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23*, pages 143–163. Springer, 2018.

[220] Qi Xu, Deyun Gao, Taixin Li, and Hongke Zhang. Low Latency Security Function Chain Embedding Across Multiple Domains. *IEEE Access*, 6:14474–14484, 2018.

[221] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, et al. ClickINC: In-network Computing as a Service in Heterogeneous Programmable Data-center Networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 798–815, 2023.

[222] Qiao Yan, Wenyao Huang, Xupeng Luo, Qingxiang Gong, and F. Richard Yu. A Multi-Level DDoS Mitigation Framework for the Industrial Internet of Things. *IEEE Communications Magazine*, 56(2):30–36, 2018.

[223] Li Yang and Abdallah Shami. A Lightweight Concept Drift Detection and Adaptation Framework for IoT Data Streams. *IEEE Internet of Things Magazine*, 4(2):96–101, 2021.

[224] Bianca Zadrozny and Charles Elkan. Transforming Classifier Scores into Accurate Multiclass Probability Estimates. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 694–699, 2002.

[225] Mingyuan Zang, Changgang Zheng, Lars Dittmann, and Noa Zilberman. Advanced Threat Defense with In-Network Traffic Analysis for IoT Gateways. *UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK)*, 2023.

[226] Mingyuan Zang, Changgang Zheng, Lars Dittmann, and Noa Zilberman. Towards Continuous Threat Defense: In-Network Traffic Analysis for IoT Gateways. *IEEE Internet of Things Journal*, 2023.

[227] Mingyuan Zang, Changgang Zheng, Tomasz Koziak, Noa Zilberman, and Lars Dittmann. Federated Learning-Based In-Network Traffic Analysis on IoT Edge. *Security for IoT Networks and Devices in 6G (Sec4IoT), 2023 IFIP Networking Conference (IFIP Networking)*, 2023.

[228] Mingyuan Zang, Changgang Zheng, Radostin Stoyanov, Lars Dittmann, and Noa Zilberman. P4Pir: In-Network Analysis for Smart IoT Gateways. In *Proceedings of the SIGCOMM'22 Poster and Demo Sessions*, pages 46–48. 2022.

[229] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning

Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

[230] Fuzhi Zhang, Yueqi Qu, Yishu Xu, and Shilei Wang. Graph Embedding-Based Approach for Detecting Group Shilling Attacks in Collaborative Recommender Systems. *Knowledge-Based Systems*, 199:105984, 2020.

[231] Jiao Zhang, F Richard Yu, Shuo Wang, Tao Huang, Zengyi Liu, and Yunjie Liu. Load Balancing in Data Center Networks: A Survey. *IEEE Communications Surveys & Tutorials*, 20(3):2324–2352, 2018.

[232] Xiaoquan Zhang, Lin Cui, Fung Po Tso, and Weijia Jia. pHeavy: Predicting Heavy Flows in the Programmable Data Plane. *IEEE Transactions on Network and Service Management*, 2021.

[233] Zihao Zhang, Bryan Lim, and Stefan Zohren. Deep Learning for Market by Order Data. *Applied Mathematical Finance*, 28(1):79–95, 2021.

[234] Zihao Zhang, Stefan Zohren, and Stephen Roberts. DeepLOB: Deep Convolutional Neural Networks for Limit Order Books. *IEEE Transactions on Signal Processing*, 67(11):3001–3012, 2019.

[235] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100, 2020.

[236] Changgang Zheng. DINC's GitHub Repository, 2023. `https://github.com/In-Network-Machine-Learning/DINC`.

[237] Changgang Zheng. IIsy's GitHub Repository, 2023. `https://github.com/In-Network-Machine-Learning/IIsy`.

[238] Changgang Zheng. Planter's GitHub Repository, 2023. `https://github.com/In-Network-Machine-Learning/Planter`.

[239] Changgang Zheng, Xinpeng Hong, Damu Ding, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. In-Network Machine Learning Using Programmable Network Devices: A Survey. *IEEE Communications Surveys & Tutorials*, pages 1–1, 2023.

[240] Changgang Zheng and Benjamin Rienecker. QCMP's GitHub Repository,

2023. https://github.com/In-Network-Machine-Learning/QCMP.

[241] Changgang Zheng, Benjamin Rienecker, and Noa Zilberman. QCMP: Load Balancing via In-network Reinforcement Learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, 2023.

[242] Changgang Zheng, Haoyue Tang, Mingyuan Zang, Xinpeng Hong, Aosong Feng, Leandros Tassiulas, and Noa Zilberman. DINC: Toward Distributed In-Network Computing. *Proceedings of the ACM on Networking*, 1(CoNEXT3):1–25, 2023.

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

[244] Changgang Zheng, Zhaoqi Xiong, Thanh T. Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Hybrid In-Network Classification Using Programmable Switches. *IEEE/ACM Transactions on Networking*, pages 1–16, 2024.

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

[246] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Liam Perreault, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Planter: Rapid Prototyping of In-Network Machine Learning Inference. In *ACM SIGCOMM Computer Communication Review*. 2024.

[247] Changgang Zheng and Noa Zilberman. Planter: Seeding Trees Within Switches. In *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*, pages 12–14. 2021.

[248] Zhizhen Zhong, Weiyang Wang, Manya Ghobadi, Alexander Sludds, Ryan Hamerly, Liane Bernstein, and Dirk Englund. IOI: In-network Optical Inference. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Optical Systems*, pages 18–22, 2021.

[249] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. An Efficient Design of Intelligent Network Data Plane. In *32nd USENIX Security Symposium (USENIX Security 23). Anaheim, CA: USENIX Association*, 2023.

[250] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, et al. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[251] Yu Zhou, Jun Bi, Cheng Zhang, Bingyang Liu, Zhaogeng Li, Yangyang Wang, and Mingli Yu. P4DB: On-the-fly Debugging for Programmable Data Planes. *IEEE/ACM Transactions on Networking*, 27(4):1714–1727, 2019.

[252] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing 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*, pages 76–89, 2020.

[253] Zhihua Zhou. *Ensemble Methods: Foundations and Algorithms*. CRC press.

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

[255] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W Moore. Where Has My Time Gone? In *Passive and Active Measurement: 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings 18*, pages 201–214. Springer, 2017.

[256] Noa Zilberman, Philip M Watts, Charalampos Rotsos, and Andrew W Moore. Reconfigurable Network Systems and Software-Defined Networking. *Proceedings of the IEEE*, 103(7):1102–1124, 2015.