TRAFFIC ENGINEERING IN DATA CENTER NETWORKS BASED ON SOFTWARE-DEFINED NETWORKING AND MACHINE LEARNING

CARLOS FELIPE ESTRADA SOLANO

Doctoral Thesis in Telematics Engineering

Advisor:
Oscar Mauricio Caicedo Rendón
Ph.D. in Computer Science

Co-Advisor:
Nelson Luis Saldanha da Fonseca
Ph.D. in Computer Engineering

University of Cauca
Faculty of Electronic and Telecommunications Engineering
Department of Telematics
Line of Research in Advanced Services of Telecommunications
Popayan, April 2022

CARLOS FELIPE ESTRADA SOLANO

TRAFFIC ENGINEERING IN DATA CENTER NETWORKS BASED ON SOFTWARE-DEFINED NETWORKING AND MACHINE LEARNING

Thesis presented to the Faculty of Electronic
and Telecommunications Engineering of the
University of Cauca to obtain the degree of

Doctor of Philosophy in:
Telematics Engineering

Advisor:
Prof. Dr. Oscar Mauricio Caicedo Rendón

Co-Advisor:
Prof. Dr. Nelson Luis Saldanha da Fonseca

Popayan
2022

The final version replaces this page with a copy of the act of public defense signed by the advisors and the evaluation panel members.

I dedicate this thesis to everyone that I love, that supported and loved me through distance and time to come this far.

To my wife, Silvana, whose unconditional love and support have given me the strength to work every day.

To my parents, Tina Adriana and Luis Carlos, whose dedication and sacrifice have been the base for my progress.

And to my furry pets, Lucas and Chata (in memory), whose follies have reduced the stress and bad mood caused by research.

It does not matter how slowly you go,
as long as you do not stop.

(Confucius)

Acknowledgements

I want to thank my family, whose endless love and support have been with me every day, even in the distance and in adversity. To my wife, Silvana, my parents, Tina Adriana and Luis Carlos, and my mother-in-law, María Inés, for their good-natured forebearance with the process and for their pride in this accomplishment. It was a team effort.

I am very grateful to the distinguished faculty members, Professor Oscar Mauricio Caicedo Rendon and Professor Nelson Luis Saldanha da Fonseca. As my advisors, they shared a detailed guidance and encouragement throughout the course of preparing for and conducting the research. Their belief that it was, indeed, possible to finish kept me going. Certainly, today I am a better researcher because of their influence.

I thank the distinguished faculty member, Professor Raouf Boutaba, and his research team at the Cheriton School of Computer Science of the University of Waterloo, Canada, for their support, patience, encouragement, and insightful comments about my investigation during and after my research internship.

My thanks to the Department of Telematics of the University of Cauca, the Institute of Computing of the University of Campinas, and the Cheriton School of the University of Waterloo, to all the professors and employees who helped me and gave me the opportunity of developing my doctoral studies.

I am also grateful to the distinguished faculty members who served as evaluators of my thesis, Professor Wilmar Campo, Professor Jéferson Nobre, and Professor Edmundo Madeira. They used their valuable time and expertise to provide helpful comments on short notice and quickly continue with the review process of my research.

Thanks to all my friends in the 116 IPET room at the University of Cauca, in the Computing Research Laboratory at the University of Campinas, and in the Davis Centre at the University of Waterloo, for all those off-topic discussions, jokes, and laughs during the stressful days.

Finally, I thank to the Administrative Department of Science, Technology and Innovation of Colombia COLCIENCIAS under Grant 647-2014, the São Paulo Research Foundation FAPESP of Brazil under Grant #2019/04914-3, and the Government of Canada under the ELAP scholarship program, for having financed my research.

Structured abstract

Background. Data Centers Networks (DCN) represent the critical infrastructure for running Internet-based applications and services that demand colossal computing and storage resources. However, the most prevalent multipath routing mechanism in DCNs, Equal Cost Multiple-Path (ECMP), may degrade the performance of these applications and services while using low network capacity due to the traffic characteristics of flows in DCNs (mice and elephants). Novel multipath routing approaches tackle this problem by leveraging Software-Defined Networking (SDN) for detecting and rescheduling the elephant flows. Some SDN-based approaches have also incorporated Machine Learning (ML) techniques to improve elephant detection and predict elephant traffic characteristics. However, SDN-based multipath routing still requires finding the best trade-off between prompt elephant detection, traffic overhead, data collection accuracy, and network modifications. Moreover, SDN-based multipath routing algorithms call for finer granularity traffic characteristics of elephant flows for improving rescheduling decisions.

Aims. This thesis focuses on developing a multipath routing mechanism based on ML and SDN for improving the routing function in DCNs. This objective divides into three tasks: *(i)* to design a multipath routing reference architecture that incorporates the capabilities of ML and SDN for improving the routing function in DCNs, *(ii)* to construct and evaluate a mechanism based on ML that predicts, in a fine-granularity way, flow characteristics in DCNs; and *(iii)* to construct and evaluate a routing mechanism based on SDN that uses predicted flow characteristics for improving the routing function in DCNs.

Methods. This thesis proposes a multipath routing mechanism that leverages both SDN and ML to improve the routing function in DCNs. Three major components form the proposed multipath routing mechanism. First, a flow detection method, called Network Elephant Learner and anaLYzer (NELLY), incorporates incremental learning at the server-side of SDN-based DCNs (SDDCN) to accurately and timely identify elephant flows at low traffic overhead while enabling continuous model adaptation under limited memory resources. Second, a Pseudo-MAC-based Multipath (PM2) routing algorithm supports transparent host migration across the whole network while reducing the number of rules installed on SDN switches, decreasing the delay introduced to flows (mainly mice) traversing the SDDCN. Third, a flow rescheduling method at the controller-side of SDDCNs, called intelligent Rescheduler of IDentified Elephants (iRIDE), improves network throughput and traffic completion time by using deep incremental learning to predict the rate and duration of elephants for computing and installing the best path across the network.

Results. An extensive evaluation shows that NELLY achieves high accuracy with a short classification time when using adaptive decision trees algorithms. Moreover, NELLY reduces traffic overhead, elephant detection time, and switch table occupancy compared to other ML-based flow detection methods. On the other hand, an analytical comparison corroborates that PM2 installs much fewer rules than other multipath routing algorithms that support transparent host migration across a large network area (other than the same switch). Finally, an extensive evaluation demonstrates that iRIDE achieves a low prediction error of the flow rate and flow duration when using deep neural networks with regularization and dropout layers. Moreover, iRIDE enables intelligent elephant rescheduling algorithms that efficiently use the available bandwidth, generating higher throughput and shorter traffic completion time than conventional ECMP.

Conclusions. Incremental learning builds accurate and efficient models for classifying and predicting flow traffic characteristics in DCNs. In fact, incremental learning reduces memory consumption by continuously updating the models from constantly generated data that is temporarily persisted. Furthermore, incorporating incremental learning into an SDN-based multipath routing mechanism improves the network traffic routing function in DCNs by reducing traffic overhead and switch table occupancy, and by supporting intelligent elephant rescheduling algorithms that efficiently use the available bandwidth.

Keywords: software-defined networking, machine learning, data center networks, multipath routing, incremental learning

Resumen estructurado

Antecedentes. Las Redes de Centros de Datos (DCN, Data Center Network) representan la infraestructura clave para ejecutar aplicaciones y servicios basados en la Internet que demandan grandes cantidades de recursos de procesamiento y almacenamiento. Sin embargo, el mecanismo de enrutamiento multicamino más predominante en las DCNs, Equal Cost Multiple-Path (ECMP), puede degradar el rendimiento de estas aplicaciones y servicios cuando se utiliza poca capacidad de red debido a las características de tráfico de los flujos en las DCNs (ratones y elefantes). Nuevas propuestas de enrutamiento multicamino abordan este problema aprovechando las Redes Definidas por Software (SDN, Software-Defined Networking) para detectar y redirigir los flujos elefante. Algunas propuestas basadas en SDN también han incorporado técnicas de Aprendizaje Automático (ML, Machine Learning) para mejorar la detección de elefantes y predecir características de tráfico de elefantes. Sin embargo, el enrutamiento multicamino basado en SDN aún requiere encontrar el mejor balance entre detección temprana de elefantes, sobrecarga de tráfico, precisión de la recolección de datos y modificaciones de red. Adicionalmente, los algoritmos de enrutamiento multicamino basado en SDN demandan características de tráfico más específicas de los flujos elefante para mejorar las decisiones de enrutamiento.

Objetivos. Esta tesis se enfoca en desarrollar un mecanismo de enrutamiento multicamino basado en ML y SDN para mejorar la función de enrutamiento en las DCNs. Este objetivo se divide en tres partes: *(i)* diseñar una arquitectura de referencia de enrutamiento multicamino que incorpore capacidades de ML y SDN para mejorar la función de enrutamiento en las DCNs, *(ii)* construir y evaluar un mecanismo basado en ML para predecir, de forma muy fina, las características de los flujos en las DCNs; y *(iii)* construir y evaluar un mecanismo de enrutamiento basado en SDN que utilice las predicciones de las características de los flujos para mejorar la función de enrutamiento en las DCNs.

Métodos. Esta tesis propone un mecanismo de enrutamiento multicamino que aprovecha tanto SDN como ML para mejorar la función de enrutamiento en DCNs. Tres componentes principales componen el mecanismo de enrutamiento multicamino propuesto. Primero, un método de detección de flujos, llamado Aprendiz y Analizador de Elefantes de Red (NELLY, Network Elephant Learner and anaLYzer), incorpora aprendizaje incremental del lado del servidor de las DCNs basadas en SDN (SDDCN, SDN-based DCN) para identificar los flujos elefante de forma precisa y oportuna con baja sobrecarga de tráfico y soportando una adaptación continua del modelo con recursos de memoria limitados. Segundo, un algoritmo de enrutamiento Multicamino basado en Pseudo-MAC (PM2, Pseudo-MAC-based Multipath) soporta la migración transparente de máquinas a través de toda la red y reduce el número de reglas instaladas en los conmutadores SDN, reduciendo el retardo introducido a los flujos (principalmente, ratones) que viajan a través de la SDDCN. Tercero, un método de redirección del lado del controlador de las SDDCNs, llamado Redireccionador Inteligente de Elefantes Identificados (iRIDE, intelligent Rescheduler of IDentified Elephants), mejora el rendimiento de la red y el tiempo de finalización del tráfico utilizando aprendizaje incremental profundo para predecir la tasa de envío y la duración de elefantes para calcular e instalar el mejor camino a través de la red.

Resultados. Una evaluación extensiva demuestra que NELLY alcanza alta precisión y un tiempo corto de clasificación cuando utiliza algoritmos de árboles de decisión adaptativos. Adicionalmente, NELLY reduce la sobrecarga de tráfico, el tiempo de detección de elefantes y la ocupación de las tablas de enrutamiento en comparación a otros métodos de detección de flujos basados en ML. Por otra parte, una comparación analítica corrobora que PM2 instala muchas menos reglas que otros algoritmos de enrutamiento multicamino que soportan migración transparente de máquina a través de un área grande de la red (otra que en el mismo switch). Finalmente, una evaluación extensiva demuestra que iRIDE alcanza un error bajo de predicción tanto de la tasa de envío como de la duración del flujo cuando utiliza redes neuronales profundas con regularización y capas de abandono (*dropout*). Además, iRIDE soporta algoritmos inteligentes de redirección de elefantes que utilizan de forma eficiente el ancho de banda disponible, generando un rendimiento más alto y un tiempo de finalización de tráfico más corto que el tradicional ECMP.

Conclusiones. El aprendizaje incremental construye modelos precisos y eficientes para clasificar y predecir características de tráfico de los flujos en las DCNs. De hecho, el aprendizaje incremental reduce el consumo de memoria al adaptar continuamente los modelos utilizando datos generados de forma constante que son almacenados temporalmente. Adicionalmente, incorporar aprendizaje incremental en un mecanismo de enrutamiento multicamino basado en SDN mejora la función de enrutamiento de tráfico de red en las DCNs al reducir la sobrecarga de tráfico y la ocupación de las tablas de enrutamiento, y al soportar algoritmos inteligentes de redirección de elefantes que utilizan el ancho de banda disponible de forma eficiente.

Palabras clave: redes definidas por software, aprendizaje automático, redes de centros de datos, enrutamiento multicamino, aprendizaje incremental

Contents

List of Figures vii

List of Tables ix

List of Algorithms xi

Acronyms xiii

1 Introduction 15

1.1 Problem statement . . . 15

1.2 Hypothesis . . . 17

1.3 Objectives . . . 18

1.3.1 General objective . . . 18

1.3.2 Specific objectives . . . 18

1.4 Contributions . . . 18

1.5 Scientific production . . . 19

1.6 Methodology and organization . . . 21

2 Background and state-of-the-art 23

2.1 Software-defined networking . . . 23

2.1.1 SDN Architecture . . . 24

2.1.2 Management plane . . . 26

2.2 Machine learning for networking . . . 28

2.2.1 Incremental learning . . . 30

2.2.2 Evolution of machine learning techniques . . . 32

2.2.3 Cognitive networking . . . 41

2.3 Traffic engineering in data center networks . . . 43

2.3.1 Data center network . . . 43

2.3.2 Traffic engineering . . . 44

2.3.3 Multipath routing in data center networks . . . 45

2.4 Final remarks . . . 52

3 Flow detection using online incremental learning at the server-side of software-defined data center networks 55

3.1 Architecture of NELLY . . . 56

3.1.1 Analyzer 58
3.1.2 Learner 62
3.2 Evaluation 63
3.2.1 Datasets 64
3.2.2 Accuracy metrics 65
3.2.3 Experiment setup 67
3.2.4 Performance analysis 68
3.3 Comparative analysis 73
3.4 Final remarks 78

4 Multipath routing based on software-defined networking and machine learning for data center networks 79
4.1 Pseudo-MAC-based multipath routing in software-defined data center networks 79
4.1.1 Motivation 81
4.1.2 Pseudo-MAC-based multipath routing 83
4.1.3 Implementation 91
4.1.4 Analytic evaluation 92
4.2 Rescheduling of elephants in software-defined data center networks using deep incremental learning 96
4.2.1 Architecture of iRIDE 97
4.2.2 Prediction 100
4.2.3 Rescheduling 112
4.2.4 Implementation 114
4.2.5 Evaluation 116
4.3 Final remarks 119

5 Conclusions 121
5.1 Answers for the fundamental questions 122
5.2 Future work 123

Bibliography 125

A Scientific Production 153

List of Figures

1.1 Thesis phases . . . 21

2.1 High-level SDN architecture . . . 24
2.2 SDN vertical management plane . . . 27
2.3 Problem categories that benefit from ML paradigms . . . 29
2.4 The constituents of ML-based solutions . . . 30
2.5 The evolution of machine learning techniques with key milestones . . . 32
2.6 Cognitive control loop for network management . . . 42
2.7 Tree-based DCN with fourth degree and multipath routing . . . 44

3.1 Architecture of NELLY . . . 57
3.2 Structure of flow records in FlowRepo . . . 60
3.3 Example of how flow records are created and updated in FlowRepo . . . 60
3.4 Confusion matrix for binary classification . . . 66
3.5 Classification accuracy of NELLY when using a single weight and inverse weights for the UNI1 and UNI2 datasets . . . 68
3.6 Classification accuracy of NELLY with the ARF and AHOT algorithms when varying the labeling threshold (θ_L) for the UNI1 and UNI2 datasets . 72
3.7 Classification accuracy of NELLY with the ARF and AHOT algorithms when varying the inverse weights of elephant flows (W_E) for the UNI1 and UNI2 datasets . . . 73
3.8 Accuracy of NELLY with the ARF and AHOT algorithms when varying the labeling threshold (θ_L) for all the IPv4 flows in UNI1 and UNI2 data traces 76
3.9 Accuracy of NELLY with the ARF and AHOT algorithms when varying the inverse weights of elephant flows (W_E) for all the IPv4 flows in UNI1 and UNI2 data traces . . . 77

4.1 RTT measurement experimental scenarios . . . 81
4.2 Average RTT per number of flow rules installed in the switch of each experimental scenario with $T = 30$ and $N = 30$ . . . 82
4.3 PM2 process to install routing rules . . . 84
4.4 ARP request interception in PM2 . . . 87
4.5 PMAC table in PM2 . . . 89
4.6 Implementation of PM2 . . . 92

4.7 Number of rules installed per switch for each layer in a fat-tree topology with size $k = 48$ and only one host in each server ($h_s = 1$) 95

4.8 Architecture of iRIDE 98

4.9 Transformations of flow rate in UNI1 and UNI2 104

4.10 Transformations of flow duration in UNI1 and UNI2 104

4.11 NN structure 108

4.12 Mean RMSE over a sliding window of instances for DNN regression models with different mini-batch sizes (n_B) in an incremental learning setting . 111

4.13 Mean RMSE over mini-batch sizes (n_B) for DNN regression models in an incremental learning setting 112

4.14 Mean prediction time per flow over mini-batch sizes (n_B) for DNN regression models in an incremental learning setting 113

4.15 Implementation of iRIDE 115

4.16 PCAP parsing for the traffic generator in iRIDE 117

4.17 Throughput over time in the bisection links of a fat-tree topology of size $k = 4$ when using PM2 and iRIDE, with both LC and WF+BTF, for routing UNI1 traffic 118

4.18 Mean throughput in the bisection links of a fat-tree topology of size $k = 4$ and traffic completion time when using PM2 and iRIDE, with both LC and WF+BTF, for routing UNI1 traffic 119

List of Tables

2.1 Summary of the limitations of distributed multipath routing 48
2.2 Summary of the limitations of SDN-based multipath routing 53

3.1 Symbols in the architecture of NELLY 58
3.2 Details of real packet traces and extracted IPv4 flows 64
3.3 Header types defined during feature engineering 65
3.4 Classification performance of NELLY with different incremental algorithms 69
3.5 Comparison of NELLY with related approaches 74

4.1 Fat-tree topology size 85
4.2 Comparison of PM2 routing with related approaches 95
4.3 Symbols in the architecture of iRIDE 98
4.4 Learning algorithm and data preprocessing with the best accuracy performance in a batch learning setting 106
4.5 Data preprocessing with the best accuracy performance for MLP in a batch learning setting 107
4.6 Accuracy performance of MLP using feasible incremental data preprocessing techniques in a batch learning setting 107
4.7 DNN structures with the lowest training errors in a batch learning setting . 109
4.8 DNN structures and regularization methods with the lowest validation errors in a batch learning setting 110

List of Algorithms

3.1 NELLY Analyzer 59
3.2 NELLY Learner 62

4.1 Handling intercepted ARP messages in PM2 88

Acronyms

$AHOT$	Adaptive Hoeffding Option Tree
AI	Artificial Intelligence
ARF	Adaptive Random Forest
ARP	Address Resolution Protocol
AS	Autonomous System
CIM	Common Information Model
$CVFDT$	Concept-adapting Very Fast Decision Tree
DCN	Data Center Network
DL	Deep Learning
DNN	Deep Neural Network
$DSCP$	Differentiated Services Code Point
$ECMP$	Equal Cost Multiple-Path
$FCAPS$	Fault, Configuration, Accounting, Performance, and Security
FPR	False Positive Rate
HTN	Hierarchical Task Network
IAT	Inter-Arrival Time
$IIoT$	Industrial Internet of Things
$iRIDE$	intelligent Rescheduler of IDentified Elephants
kNN	k-Nearest Neighbors
MAC	Media Access Control
MCC	Matthews Correlation Coefficient

ML	Machine Learning
MLP	Multi-Layer Perceptron
MOA	Massive Online Analysis
NB	Naïve Bayes
$NELLY$	Network Elephant Learner and anaLYzer
NFV	Network Function Virtualization
NN	Neural Network
$OAUE$	Online Accuracy Updated Ensemble
PA	Passive-Aggressive
PAW	Probabilistic Adaptive Windowing
PC	Physical Computer
PCP	Priority Code Point
$PM2$	Pseudo-MAC-based Multipath
$PMAC$	Pseudo-MAC
RL	Reinforcement learning
RTT	Round-Trip Time
$SDDCN$	Software-Defined Data Center Network
SDN	Software-Defined Networking
SGD	Stochastic Gradient Descent
$SMOGN$	Synthetic Minority Over-sampling technique for regression with Gaussian Noise
SVM	Support Vector Machines
ToR	Top-of-Rack
TPR	True Positive Rate
$VFDR$	Very Fast Decision Rules
$VFDT$	Very Fast Decision Tree
VM	Virtual Machine
$YANG$	Yet Another Next Generation

Chapter 1

Introduction

1.1 Problem statement

Nowadays, the ever-growing Internet-based applications and services demand a huge amount of computing and storage resources. Data centers represent the key infrastructure that supports and provides such resources as a large number of servers interconnected by a specially designed network, called Data Center Network (DCN) [1]. The goal of DCN is to provide significant bandwidth capacity in order to achieve high throughput[1] and low-latency[2].

Everyday, DCN managers are looking for solutions that allow optimizing these performance requirements (i.e., high bandwidth, high throughput, and low latency) without the need to add more capacity to the network. Traffic engineering represents a great opportunity in this realm. Particularly, load-balancing is a desirable feature for reducing network congestion while improving network resource availability and application performance [2, 3]. A well-known technique for implementing load-balancing in DCNs is multipath routing, which distributes traffic over multiple concurrent paths such that all the links are optimally loaded [4]. Hereinafter, when this dissertation mentions routing in DCN, it is particularly referring to intra-DCN routing.

The most prevalent multipath routing solution in DCNs is Equal Cost Multiple-Path (ECMP) [5, 6]. Usually, ECMP uses a hash function in every switch to assign each incoming flow to one of the equal-cost forwarding paths maintained by the switch for reaching a destination [7]. However, traffic in DCNs presents a broad distribution of flow sizes: from small, short-lived flows (i.e., mice) to large, long-lived flows (i.e., elephants) [8–11]. This wide dispersion of flow sizes causes *hot-spots* in DCNs based on ECMP routing, i.e., some links are highly utilized while others are underutilized.

For example, if two mouse flows and two elephant flows arrive at the same ECMP-

[1] Total number of packets processed per second.

[2] Average processing time used for a single packet.

enabled switch, it is possible that this switch assigns the two mouse flows to one of the forwarding paths and the two elephant flows to one of the other forwarding paths. Therefore, the link transporting the two mouse flows is going to present less load and be free much faster than the link transporting the two elephant flows, causing an under-using and overloading of links, respectively. Facebook's Altoona [12] propose to prevent the degrading of elephant flows by making the network multi-speed. However, this is not an efficient approach. Rather than adding more capacity, the issue is selecting a routing mechanism for drawing traffic effectively.

For this reason, recent multipath routing mechanisms have been proposed for improving ECMP. Broadly, these mechanisms can be categorized as distributed and centralized multipath routing. Distributed multipath routing maintains routing decisions at switches or servers and tackles ECMP limitations by *(i)* using different levels of granularity for traffic splitting (i.e., packet-level [13, 14] and sub-flow-level [15]), *(ii)* adding weights to the paths [15, 16]; or *(iii)* incorporating congestion information for making routing decisions [17]. Distributed multipath routing mechanisms that combine sub-flow-level traffic splitting and congestion-awareness—local congestion [18, 19] or global congestion [20–22]—have provided great results for load-balancing in DCNs. However, these solutions require specialized hardware implementation, potentially cause packet reordering, and lack a global view of traffic for making routing decisions.

Centralized multipath routing have leveraged Software-Defined Networking (SDN) to face the ECMP limitations; DCNs using SDN are referred to as Software-Defined Data Center Networks (SDDCNs). SDN allows a logically centralized controller to dynamically make and install routing decisions on the basis of a global view of the network [23, 24]. Hereinafter, this dissertation uses the term SDN-based multipath routing to refer to centralized mechanisms. SDN-based multipath routing reschedules elephant flows, while handling mouse flows by employing default routing, such as ECMP. Early SDN-based mechanisms proposed reactive flow detection methods to discriminate elephants from mice by using static thresholds either at the controller-side [25, 26], switch-side [27], or server-side [28, 29] of SDDCNs. However, reactive methods are not suitable for SDDCNs since hot-spots may occur before the elephant flows are detected.

Novel SDN-based multipath routing have incorporated Machine Learning (ML)-based flow detection methods for proactively identifying elephants. However, ML-based methods train their classification models at the controller-side of SDDCNs, requiring the central collection of either per-flow data [30–32] or sampling-based data [33–35]. The central collection of per-flow data, however, causes problems such as heavy traffic overhead and poor scalability. Sampling-based data, on the other hand, tends to provide delayed and inaccurate flow information. Moreover, sampling techniques that mitigate the problem rely on non-standard SDN specifications.

Another gap in multipath routing is that SDN-based mechanisms, both with reactive and ML-based flow detection methods, merely identify elephant flows (i.e., binary classification) and lack fine-grained information for making routing decisions. Therefore, their routing algorithm reschedules all the elephants using the same approach regardless of the different traffic characteristics that elephant flows exhibit in DCNs. This elephant-oblivious routing, however, may cause hot-spots in SDDCNs, reducing the performance of the network. Only a few SDN-based mechanisms introduce ML-based methods that classify flows into more than two categories (i.e., multiclass classification) [31, 33]. However, the routing algorithms in such SDN-based mechanisms use only a part of the information given by their flow detection methods. Moreover, such flow detection methods employ a reduced number of classification categories (up to five) that still fall short to cover the broad distribution of elephant flows in DCNs.

Based on these statements, SDN-based multipath routing still requires finding the best trade-off between prompt elephant detection, traffic overhead, data collection accuracy, and network modifications. Besides, SDN-based multipath routing algorithms call for finer granularity traffic characteristics of elephant flows for improving rescheduling decisions. Therefore, this thesis project focused on solving the following research question:

How to carry out multipath routing in DCNs for enabling high throughput and low delay while maintaining efficient use of resources?

1.2 Hypothesis

To address the research question, this thesis raised the following hypothesis: **using ML for fine-granularity prediction of flow characteristics and SDN for dynamic control of flow scheduling would allow building a multipath routing mechanism for DCNs that improves[3] the routing function.**

The following fundamental questions, associated with the hypothesis, guided the investigation conducted in this thesis.

- What is the accuracy and efficiency, in terms of time and memory, of ML techniques for predicting flow characteristics of network traffic from DCNs?
- Does incorporating ML techniques to an SDN-based multipath routing mechanism improve network traffic routing, in terms of throughput and delay, in DCNs?

[3] In terms of high throughput and low delay while efficient use of resources

1.3 Objectives

1.3.1 General objective

To develop a multipath routing mechanism based on ML and SDN for improving the routing function in DCNs.

1.3.2 Specific objectives

- To design a multipath routing reference architecture that incorporates the capabilities of ML and SDN for improving the routing function in DCNs.
- To construct and evaluate[4] a mechanism based on ML that predicts, in a fine-granularity way, flow characteristics in DCNs.
- To construct and evaluate[5] a routing mechanism based on SDN that uses predicted flow characteristics for improving the routing function in DCNs.

1.4 Contributions

The scientific research process conducted during this thesis led to building a multipath routing mechanism that leverages ML techniques for predicting traffic flow characteristics and SDN capabilities for differentiated and dynamic control of traffic flows aiming to improve the routing function in DCNs. Three major components form our multipath routing mechanism.

- A flow detection method that incorporates incremental learning at the server-side of SDDCNs to accurately and timely identify elephant flows while generating low traffic overhead and adapting to varying traffic characteristics under limited memory resources.
- A Pseudo-MAC (PMAC)-based multipath routing algorithm for steering traffic flows (mainly, mice) in SDDCNs that supports transparent host migration across the whole network while reducing the number of rules installed on SDN switches, decreasing the delay introduced to flows traversing the network.
- A flow rescheduling method at the controller-side of SDDCNs that applies deep incremental learning for predicting traffic characteristics of elephant flows to compute and install the best path per elephant flow across the network.

[4] In terms of accuracy (e.g., true/false positives/negatives) and processing requirements (e.g., training data, training time, run-time).

[5] In terms of traffic performance (e.g., throughput, delay) and resource utilization (e.g., links load)

Moreover, collaborations with other researchers (i.e., student advisory and research internships) during this thesis led to the following contributions.

- An SDN management architecture based on Hierarchical Task Network (HTN) and Network Function Virtualization (NFV) that provides an automated, workable, and flexible approach for monitoring, configuring, and controlling SDN resources.
- A vertical Management Plane for SDN that considers management tasks involving more than one Autonomous System (AS).
- A set of data models for the SDN architecture based on the Yet Another Next Generation (YANG) language to support integrated management in a technology-agnostic and heterogeneous SDN environment.

  These three contributions are built on the reference architecture for SDN integrated management and the Common Information Model (CIM)-based information model proposed in the author's master thesis [36].
- A cognitive control loop framework for autonomic network management that incorporates ML at every function of the closed-loop and each of the Fault, Configuration, Accounting, Performance, and Security (FCAPS) management areas. A discussion about the opportunities and challenges pertaining to using ML to manage autonomic networks complements this cognitive framework.
- A comprehensive body of knowledge on ML techniques in support of networking. Particularly, this body of knowledge comprehends the following contributions.
  - A generic approach for designing ML-based solutions in networking.
  - A brief history of ML focused on the techniques that have been applied in networking.
  - A literature review about the advances made in the application of ML in different networking areas, including traffic prediction, classification, and routing, which are fundamental in traffic engineering for optimizing network performance.
  - Prominent challenges and open research opportunities on the feasibility and practicality of ML in current and future networks.

1.5 Scientific production

Two published papers (one in a highly ranked journal and one in a renowned conference) and one journal paper in construction report to the scientific community the major contributions achieved during this thesis.

- “NELLY: Flow Detection Using Incremental Learning at the Server Side of SDN-based Data Centers,” published in IEEE Transactions on Industrial Informatics, 2020 [37]. Ranking: JCR Q1, SJR Q1, Publindex A1, Qualis A1. Contribution: the elephant flow detection method using incremental learning.

- “An Efficient Mice Flow Routing Algorithm for Data Centers based on Software-Defined Networking,” published in the proceedings of 2019 IEEE International Conference on Communications (ICC) [38]. Ranking: H5-index 56, Qualis A1, CORE B. Contribution: the PMAC-based multipath routing algorithm for SDDCNs.

- “iRIDE: Rescheduling of Elephant Flows in SDN-based Data Centers Using Incremental Deep Learning to Predict Traffic Characteristics,” in construction. Contribution: the flow rescheduling method using deep incremental learning.

Furthermore, six papers published in renowned journals and conferences report to the scientific community the contributions achieved in collaboration with other researchers. These papers are listed in chronological order.

- “A Framework for SDN Integrated Management based on a CIM Model and a Vertical Management Plane,” published in Computer Communications, 2017 [39]. Ranking: JCR[6] Q2, SJR[7] Q2, Publindex[8] A1, Qualis[9] A2. Contribution: the reference architecture for the SDN integrated management and the CIM-based information model.

- “SDN Management Based on Hierarchical Task Network and Network Functions Virtualization,” published in the proceedings of the 2017 IEEE Symposium on Computers and Communications (ISCC) [40]. Ranking: H5-index[10] 20, Qualis A3, CORE[11] B. Contribution: the HTN- and NFV-based architecture for managing SDN.

- “A YANG Model for a vertical SDN Management Plane,” published in the proceedings of the 2017 IEEE Colombian Conference on Communications and Computing (COLCOM) [41]. Ranking: H5-index 9. Contribution: the YANG data model for the SDN Management Plane.

- “Machine Learning for Cognitive Network Management,” published in IEEE Communications Magazine, 2018 [42]. Ranking: JCR Q1, SJR Q1, Publindex A1,

[6] Quartile from Journal Citation Reports (JCR)

[7] Quartile from SCImago Journal Rank (SJR)

[8] Bibliographic index from COLCIENCIAS, Colombia

[9] Bibliographic index from CAPES, Brazil

[10] H-index from Google Scholar Metrics

[11] Rank from COmputing Research and Education (CORE), Autralia

Qualis A1. Contribution: the cognitive control loop framework for autonomic network management.

- “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities,” published in Journal of Internet Services and Applications, 2018 [43]. Ranking: SJR Q2, Publindex A2, Qualis A2. Contribution: the body of knowledge on ML in networking.
- “An Approach based on YANG for SDN Management,” published in International Journal of Communication Systems, 2021 [44]. Ranking: SJR Q2, Publindex A2, Qualis A3. Contribution: the Management Plane with multiple ASs support and the YANG data models for the SDN architecture.

Appendix A lists the eight published papers in chronological order.

1.6 Methodology and organization

The research process that guided the development of this thesis is based on a typical scheme of the scientific method [45]. Figure 1.1 depicts the phases that form this research process: Problem Statement, Hypothesis Construction, Experimentation, Conclusion, and Publication. Problem Statement, for identifying and establishing the research question. Hypothesis Construction, for formulating the hypothesis and the associated fundamental questions. In addition, this phase aimed to define and carry out the conceptual and technological approaches. Experimentation, for testing the hypothesis and analyzing the evaluation results. Conclusion, for outlining conclusions and future works. Note that Hypothesis Construction had feedback from Experimentation and Conclusion. Publication, for submitting and publishing papers for renowned conferences and journals. The writing of this dissertation document also belongs to this last phase.

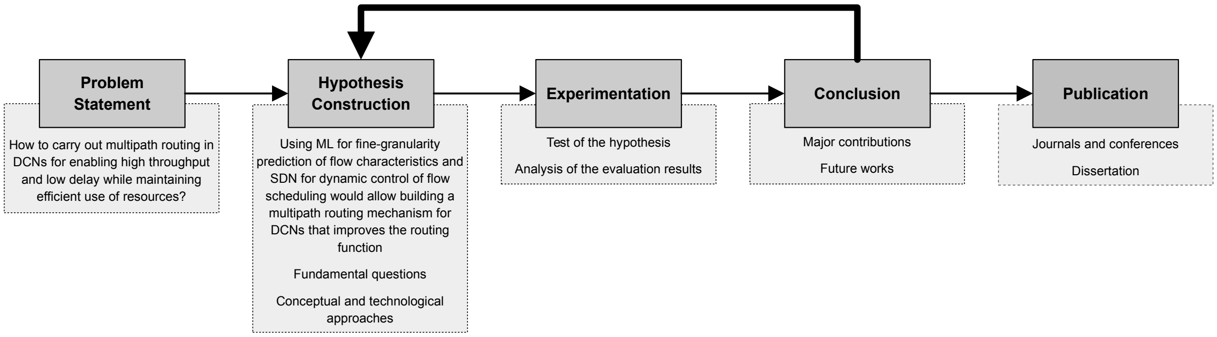

Problem Statement
How to carry out multipath routing in DCNs for enabling high throughput and low delay while maintaining efficient use of resources?
Hypothesis Construction
Using ML for fine-granularity prediction of flow characteristics and SDN for dynamic control of flow scheduling would allow building a multipath routing mechanism for DCNs that improves the routing function
Fundamental questions
Conceptual and technological approaches
Experimentation
Test of the hypothesis
Analysis of the evaluation results
Conclusion
Major contributions
Future works
Publication
Journals and conferences
Dissertation

Figure 1.1: Thesis phases

The organization of this document reflects the phases of the methodology.

- This introductory chapter presents the problem statement, delineates the hypothesis, exposes the objectives, summarizes the contributions, lists the scientific production, and describes the overall structure of this dissertation.

- **Chapter 2** reviews the main concepts and research related to SDN management, ML for networking, and traffic engineering in DCNs.

- **Chapter 3** introduces the server-side flow detection method for SDDCNs based on incremental learning.

- **Chapter 4** details the multipath routing mechanism based on ML and SDN for DCNs. Section 4.1 describes the multipath routing algorithm for steering mice in SDDCNs.

- **Chapter 5** presents conclusions about the hypothesis and the fundamental questions as well as research directions.

Chapter 2

Background and state-of-the-art

This chapter presents the background of the main research topics encompassed in this thesis. In this way, the first section introduces a bottom-up description of the typical SDN architecture followed by a detailed explanation of our view of a management plane for SDN. The second section provides a primer of ML for networking, discussing different categories of ML-based techniques, their essential constituents, and their evolution. Moreover, this section reviews the notion of cognitive networking, focusing on our proposal for realizing a cognitive control loop for autonomic networking. Finally, this chapter contextualize the concepts of DCN and traffic engineering, and provides a literature review about multipath routing for load-balancing in DCNs, focusing on the seminal works that use SDN and ML for addressing such a challenge.

2.1 Software-defined networking

SDN represents one of the most accepted and attractive trends, in research and industry, for defining the architecture of future networks [46,47]. From a general aspect, SDN decouples the control and forwarding planes for enabling a simpler network operation from a logically centralized software program, usually known as the *controller* [24]. The *control plane* (i.e., the controller) compiles decision policies and enforces them on the *data plane* (i.e., switches and routers) through a vendor independent protocol. OpenFlow [48] is the most well-known open SDN protocol and a *de facto* standard because of its widespread use by vendors and research.

SDN provides four major advantages for operating networks [49]: *(i)* a centralized global view about the network state (e.g., resource capabilities and dynamic status) and the deployed applications (e.g., QoS requirements), *(ii)* a dynamic programmability of multiple forwarding devices (e.g., allocating resources to prevent congestion and improve performance), *(iii)* open interfaces for handling the forwarding plane (e.g., OpenFlow) and for developing the applications (e.g., Application Programming Interfaces (APIs) based on protocols and programming languages); and *(iv)* a flexible flow management (e.g., multiple flow tables in OpenFlow). These unique features lead the SDN architecture to emerge as a promising scenario for efficiently and intelligently implementing management techniques, particularly for traffic engineering.

2.1.1 SDN Architecture

Multiple standardization bodies, such as the Linux Foundation [50] and Open Network Foundation (ONF) [51], focus on encouraging and normalizing open SDN frameworks. Also, various private networking vendors, such as Cisco [52] and Juniper [53], offer proprietary SDN deployments. In turn, several research surveys [23, 54] work on improving architectural aspects of SDN. These open, proprietary, and research proposals establish a typical SDN architecture composed of three horizontal planes (i.e., data, control, and application) and three interfaces (i.e., southbound, northbound, and east/westbound), as depicted in Figure 2.1.

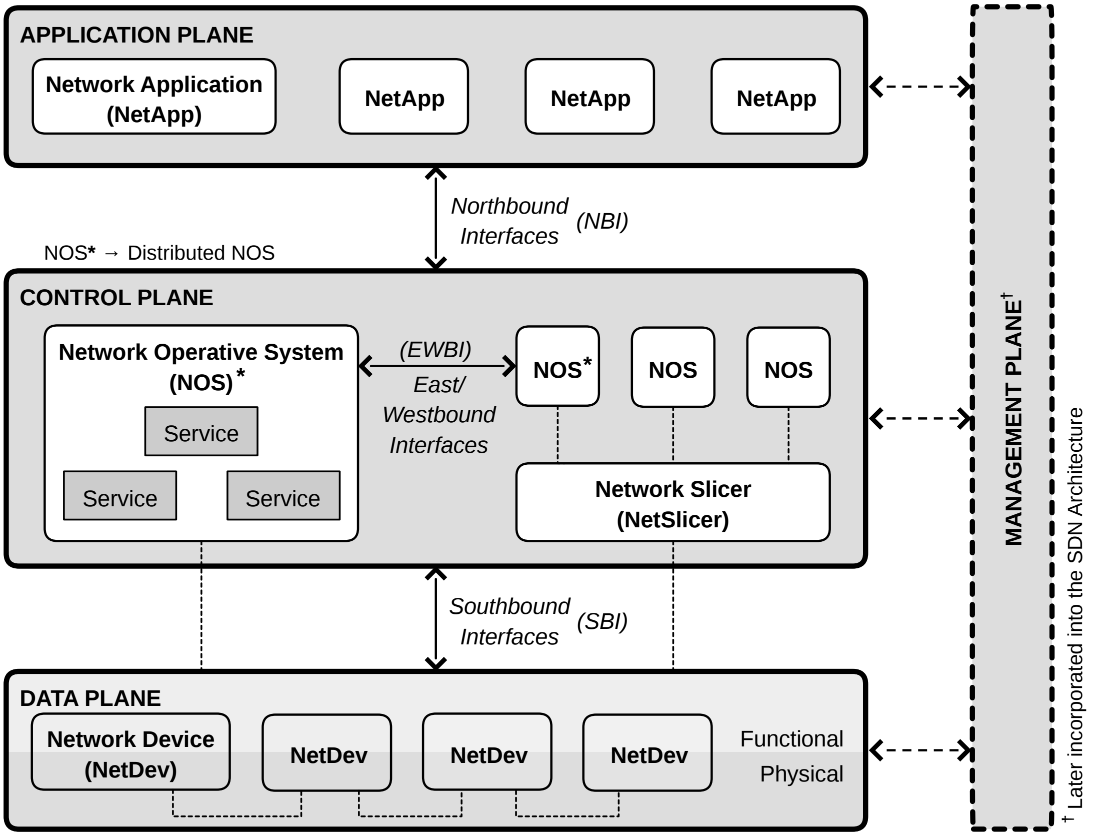

APPLICATION PLANE
Network Application (NetApp)
NetApp
NetApp
NetApp
Northbound Interfaces (NBI)
NOS* → Distributed NOS
CONTROL PLANE
Network Operative System (NOS)*
Service
Service
Service
(EWBI) East/ Westbound Interfaces
NOS*
NOS
NOS
Network Slicer (NetSlicer)
Southbound Interfaces (SBI)
DATA PLANE
Network Device (NetDev)
NetDev
NetDev
NetDev
Functional
Physical
MANAGEMENT PLANE†
† Later incorporated into the SDN Architecture

Figure 2.1: High-level SDN architecture

At the bottom of the SDN architecture, the *data plane* (a.k.a. forwarding plane) deploys the network infrastructure formed by interconnected Network Devices (NetDevs), such as switches and routers, that perform forwarding operations. A NetDev consists of a *physical* and a *functional* part. The former comprises hardware elements, such as ports, storage, processor, and memory. The latter defines a collection of software-based forwarding functions executed by NetDevs. Regarding this functional part, a NetDev ranges from *dumb* to *custom*. A dumb NetDev merely carries out simple forwarding functions, such as longest prefix match. For example, OpenFlow-only switches [55] just forward packets using the rules installed in their flow tables—updated by an OpenFlow controller. On the other hand, a custom NetDev relies on programmable platforms [56]—e.g., Protocol-Independent Switch Architecture (PISA) and Field-Programmable Gate Array (FPGA)—to integrate more complex forwarding functions, such as load balancing [57] and in-band network telemetry [58]. For example, P4 [59] provides a target- and protocol-independent language that allows programming packet processing functionality.

In the middle, the *control plane* compiles the network logic and enforces decision policies on the data plane through SouthBound Interfaces (SBIs). Each SBI defines the set of instructions and the communication protocols to allow the interaction between components in the control and data planes. The OpenFlow protocol [48] is the most well-known open standard SBI because its widespread use by vendors and research [46]. Other SBI proposals are Forwarding and Control Element Separation (ForCES) [60], Protocol-Oblivious Forwarding (POF) [61], and P4Runtime [62].

The control plane comprises Network Slicers (NetSlicers) and Network Operating Systems (NOSs). A NetSlicer divides the underlying network infrastructure into several isolated logical network instances (a.k.a. slices), assigning their control to specific NOSs. NetSlicers may employ SBIs to communicate with NOSs. For example, FlowVisor [63] acts as an OpenFlow proxy between switches and controllers, redirecting messages according to flow parameters, such as TCP ports and IP addresses. An NOS instructs the underlying data plane and provides generic services (e.g., topology discovering and host tracking) and NorthBound Interfacess (NBIs) to the *application plane*, facilitating to integrate custom Network Applications (NetApps). The possibility to add these NetApps in an easier way is the key advantage of SDN to encourage innovation on the Internet. OpenFlow controllers [55] and ForCES Control Element (CE) [60] represent NOS instances. It is important to highlight that a lot of frameworks exist to develop and deploy OpenFlow controllers, including open source projects like NOX [64] for C++, POX [65] and Ryu [66] for Python, Floodlight [67] and OpenDaylight [68] for Java, and Trema [69] for Ruby. Also, the control plane defines East/WestBound Interfacess (EWBIs) to deploy distributed NOSs. For example, SDNi [70] and ForCES CE-CE interface [60].

At the top of the SDN architecture, the application plane contains NetApps that deploy and orchestrate business logic and high-level network functions, such as routing policies and access control. As aforementioned, NetApps communicate with the control plane through NBIs provided by NOSs. NBIs encompass common APIs based on protocols (e.g., Floodlight REST API [71]), programming languages (e.g., ad-hoc, Pyretic [72], and Procera [73]), file systems (e.g., YANC [74]), among others. NetApps run either locally or remotely regarding NOSs. Local NetApps prefer NBIs based on programming languages, whereas remote NetApps usually employ protocol-based NBIs.

2.1.2 Management plane

As depicted in Section 2.1.1, the traditional SDN architecture lacked an integrated and standardized framework for managing the virtual[1], dynamic[2], and heterogeneous[3] SDN environment. Later SDN approaches [75–78] considered a vertical *management plane* in the SDN architecture (see Figure 2.1) for carrying out different Operation, Administration, and Maintenance (OAM) functions. For example, assigning the data plane resources to the corresponding control components, and configuring the policies and Service Level Agreements (SLAs) of the control and application planes. Although NOSs may implement some OAM functions, flooding the control plane with a lot of managing tasks may cause low network performance.

These SDN approaches exposed a very high-level of their management component. Therefore, we extended and detailed such management plane aiming to facilitate integrated control and monitoring of heterogeneous SDNs [39]. This approach originally lacked inter-domain communication management; hence, we later incorporated *inter-AS* elements (i.e., repository, adapter, interface, and agent) for supporting management tasks involving more than one domain [41, 44]. Figure 2.2 depicts the proposed vertical management plane comprising different elements: *data repositories*, *managers*, *adapters*, *agents*, and *management interfaces*.

Two data repositories coexist, each holding a Resource Representation Model (RRM) that handles metadata to provide an abstract, technology-neutral characterization of SDN resources. Data repositories also serve managers for storing *instance data*, which represents execution-specific data whose structure follows such an RRM. Particularly, the inter-AS data repository focuses on information relevant from other ASs for enabling management tasks in an inter-domain environment. On the other hand, managers orchestrate and deploy *management services* to carry out different SDN management functions. These management services expose user interfaces to enable *network administrators* to interact with managers. Managers also interact with agents via adapters, which enable a protocol-agnostic communication using data type rendering, protocol translation, and well-defined management interfaces. Note each management inter-

[1] Capability for sharing network resources from a same physical infrastructure among several virtual network instances.

[2] Flexibility for adding, modifying, migrating, and removing network resources.

[3] Independence of the technology deployed by network resources.

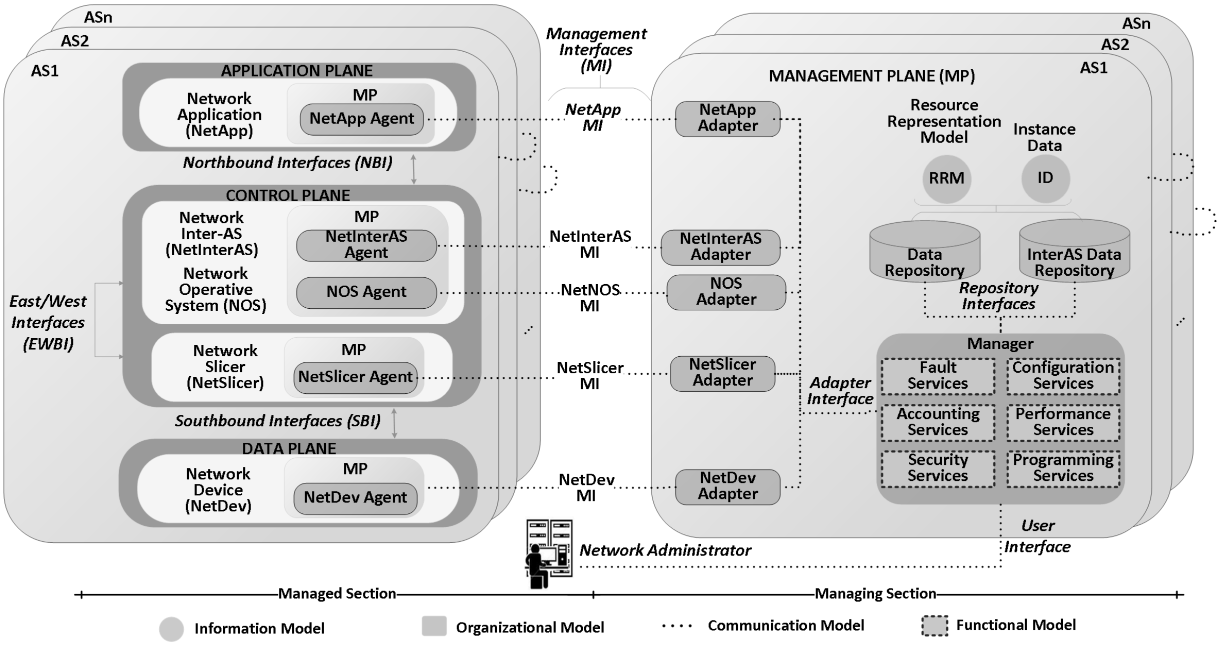


Figure 2.2: SDN vertical management plane

face connects an adapter with its corresponding agent. Finally, agents inside managed SDN resources act on behalf of managers.

Figure 2.2 also describes our management plane referencing the four Open System Interconnection (OSI) network management submodels [79]. First, the *organizational model* specifies roles and collaboration forms of the managing entities (i.e., managers and adapters) and managed entities (i.e., agents). Second, the *communication model* delineates the exchange of management data (e.g., operations, queries, events) and the enabling technologies for the user and adapter interfaces (e.g., JSON [80] over HTTP [81]), repository interfaces (e.g., XML [82] over HTTP), and management interfaces (e.g., OVSDB [83], NETCONF [84], SNMP [85]). Third, the *functional model* structures the management services referencing the five OSI management functional areas (i.e., FCAPS [86]) along with a novel programmability function (i.e., FCAPS+P) introduced by SDN. Fourth, the *information model* establishes a shared abstraction of SDN resources for achieving an integrated and technology-independent management.

The whole operation of our management plane is based on RRM, which implements the information model and originally leveraged CIM[4] for representing the SDN architecture as a conceptual model [39]. We later implemented the CIM model using YANG data models[5] since the latter provides a more human-readable and easier-to-learn language

[4] CIM is an open standard aimed at assisting the management of devices, services, and computer networks by facilitating their modeling [87].

[5] Information models represent managed objects at a conceptual level, independent of any specific protocols used to transport the data. Whereas, data models define managed objects at a lower level of

than the former, while also being technology-agnostic [41, 44]. Note that different SDN solutions from the industry and academia increasingly use YANG to build new management solutions [89–91]. This is because YANG emerged from a widely adopted network management protocol (i.e., NETCONF) [92, 93] and structures data under a hierarchical tree topology using modules and submodules that allow description easily network devices and their relationships [94].

We also defined our management plane in the context of network automation by introducing an SDN management architecture based on HTN and NFV [40]. This architecture provides an automated, workable, and flexible approach for monitoring, configuring, and controlling SDN resources. To achieve this goal, our management plane instantiates the three NFV MANagement and Orchestrator (MANO) functional blocks [95]. The two managers (i.e., virtual function and virtualized infrastructure) enable the communication between the managing and managed entities. Whereas, the orchestrator leverages the automated planning capability from HTN [96] to facilitate composing network management tasks. This allows overcoming low automation management tasks, such as reconfiguring a broken connection, with minimal human intervention.

2.2 Machine learning for networking

Machine learning is a branch of artificial intelligence whose foundational concepts were acquired over the years from contributions in the areas of computer science, mathematics, philosophy, economics, neuroscience, psychology, control theory, and more [97]. In 1959, Arthur Samuel coined the term “Machine Learning”, as *“the field of study that gives computers the ability to learn without being explicitly programmed.”* However, ML goes beyond simply learning or extracting knowledge, to utilizing and improving knowledge over time and with experience [98]. Broadly, ML can be divided into three paradigms, based on how the *learning* is achieved [97, 99]: *supervised*, *unsupervised*, and *reinforcement* learning.

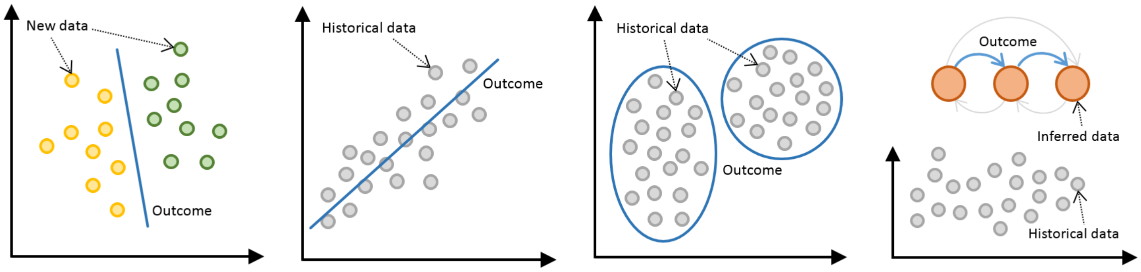
Supervised learning uses labeled training datasets to create models that map inputs to their corresponding outputs. Then, this learning approach requires labeling methods for establishing the *ground truth* in datasets and “learns” to identify patterns or behaviors in the “known” training datasets. Typically, supervised learning solves *classification* and *regression* problems that pertain to predicting discrete or continuous valued outcomes, respectively (see Figures 2.3(a) and 2.3(b)). For example, a classification problem can be to identify mice and elephant flows. Whereas, a regression problem can be to predict the size of each flow.

Unsupervised learning uses unlabeled training datasets to create models that find

abstraction, including implementation- and protocol-specific details. Multiple data models can be derived from a single information model [88].

dominating structure or patterns in the data. This approach is most suited for *clustering* problems (see Figure 2.3(c)). For instance, outliers detection and density estimation problems in networking, can pertain to grouping different instances of attacks based on their similarities. Between supervised and unsupervised learning resides *semi-supervised* learning to face partial knowledge. That is, having incomplete labels or missing labels for training data.

Reinforcement learning (RL) uses an agent that interacts with the external world to learn by exploring the environment and exploiting the knowledge. The actions are rewarded or penalized. Therefore, the training data constitutes a set of state-action pairs and rewards (or penalties). The agent uses feedback from the environment to learn the best sequence of actions or "policy" to optimize a cumulative reward. Hence, this learning approach is best suited for making cognitive choices, such as decision making, planning, and scheduling [100]. For example, rule extraction from the data that is statistically supported and not predicted (see Figure 2.3(d)).



(a) Classification (b) Regression (c) Clustering (d) Rule extraction

Figure 2.3: Problem categories that benefit from ML paradigms

Though there are different categories of problems that enjoy the benefits of ML, there is a generic approach to building ML-based solutions. Figure 2.4 illustrates the key constituents in designing ML-based solutions for networking. *Data collection* pertains to gathering, generating, and defining the set of data and the set of classes of interest. Depending on the applied ML paradigm, the collected data might be labeled for establishing the *ground truth*. *Feature engineering* is used to reduce dimensionality in data and identify discriminating features that reduce computational overhead and increase accuracy. *Model learning* refers to training one or multiple models using ML techniques, which carefully analyze the complex inter- and intra-relationships in data to yield the outcome. Finally, *model validation* regards defining the accuracy metrics that measure the performance of the trained models.

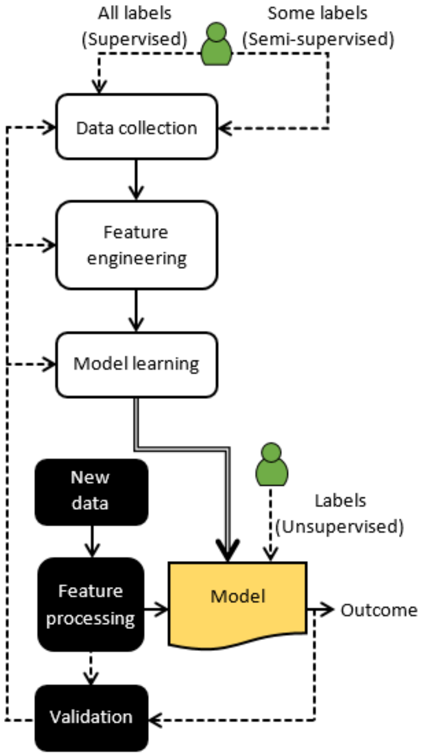


Figure 2.4: The constituents of ML-based solutions

2.2.1 Incremental learning

Most ML approaches, mainly based on supervised and unsupervised learning, implement the classical *batch* learning: all data needed to generate an inference is collected before training and is simultaneously accessed [101]. Usually, batch learning divides the data into *training*, *validation* (also called development), and *test* sets [102]. The training set is leveraged to fit the parameters of an ML model (e.g., weights). Whereas, the validation set is used to choose the suitable hyperparameters of an ML model (e.g., architecture, learning rate, regularization), or choose a model from a pool of ML models. Finally, the test set is used to assess the unbiased performance of the selected model. Note, batch learning considers that both the data and its underlying structure are static.

A common batch setting decomposition of the dataset can conform to $60/20/20$% among training, validation, and test datasets, or $70/30$% in case validation is not required [102]. These rule-of-thumb decompositions are reasonable for datasets that are not very large. However, in the era of big data, where a dataset can have millions of entries, other extreme decompositions, such as $98/1/1$% or $99.8/0.1/0.1$%, are also valid. Several ML studies and practitioners consider that validation and test sets with sizes on the order of tens of thousands of instances are sufficient. In batch learning, validation and testing usually follows one of two methods[6] [103]: holdout or k-fold cross-validation. In the holdout method, part of the available dataset is set aside and used as a validation (or testing) set. Whereas, in the k-fold cross-validation, the available dataset is randomly divided into k equal subsets. Validation (or testing) process is repeated k times, with $k - 1$ unique subsets for training and the remaining subset for validating (or testing) the model, and the outcomes are averaged over the rounds.

Batch learning has also incorporated some big data upgrades, such as external storage and data subsets, that enable processing large but static datasets [103]. However, the more data available, the less performance to simply output a final static model. Moreover, batch learning fails to handle a continuous supply of changing data (i.e., sequential data or data stream), which is characteristic of the networks and its high dynamicity. Since static models cannot continuously integrate new information, they have to be constantly reconstructed to ensure their validity over time. However, re-training a model from scratch is computationally expensive, time-consuming, and leads to potentially outdated models [104]. An interesting research direction is to achieve *incremental* learning, where the model is re-trained with only the new data.

Incremental learning[7] refers to continuously updating a model using sequential data (i.e., constantly arriving data with no specific order) without re-processing the data already used [105]. In fact, many datasets, although static, are so large that they would be dealt with as sequential data. Since sequential data can become endless in some domains, including computer networks, incremental learning aims for bounding model complexity and processing time, enabling lifelong learning and a constantly updated model under limited memory resources. Therefore, four fundamental aspects characterize an incremental learning algorithm [103]: *(i)* it processes an instance at a time in the order that arrives, inspecting it as input only once[8], *(ii)* it uses a limited amount of memory, even when processing data much more extensive than available memory, *(iii)* it restricts the runtime to a limit, which is particularly pivotal for algorithms aimed at real-time applications (e.g., networking); and *(iv)* it can provide a prediction anytime, no matter the number of instances used for training.

The evaluation of incremental learning algorithms follows one of two methods [103]: holdout and interleaved test-then-train. Holdout represents a natural extension from batch learning, where a single, independent, and sufficiently large (i.e., tens of thousands of instances) test set can provide a valid accuracy measurement. The model

[6] Other evaluation practices for batch learning exist, such as leave-one-out and bootstrap, but the ML community warns about using them.

[7] The definition of incremental learning is not always consistent in the literature and involves certain ambiguity regarding related terms, including online learning, data stream mining, stream learning, and incremental online learning. The definition presented in this thesis aims to cover the commonalities among such related concepts.

[8] An algorithm may store some instances internally in the short term. However, at some point, it needs to discard some of them to meet memory and time limits.

can be evaluated on the test set either after the last training or periodically to track its performance over time. In scenarios where the data statistics change over time (a.k.a. concept change), the test set cannot be static but must be constantly collected using new instances not yet used for training. On the other hand, interleaved test-then-train (a.k.a. prequential) refers to using each individual instance for testing the model before training. Note this method does not need a holdout set, making maximum use of the collected data for both testing and training. In interleaved test-then-train, early mistakes from a poorly trained model (a.k.a. cold start) punish the actual accuracy that incremental algorithms might achieve. Although this effect diminishes over time, evaluation techniques like pretraining and sliding windows help to correct the accuracy measurement. Incremental learning algorithms are usually evaluated using interleaved test-then-train as it easily handles a potentially infinite sequence of instances arriving one after another.

2.2.2 Evolution of machine learning techniques

Research efforts during the last 75 years have given rise to a plethora of ML techniques [97–99, 106]. This section provides a brief history of ML, focusing on the techniques that have been particularly applied in the area of computer networks, including this thesis (see Figure 2.5).

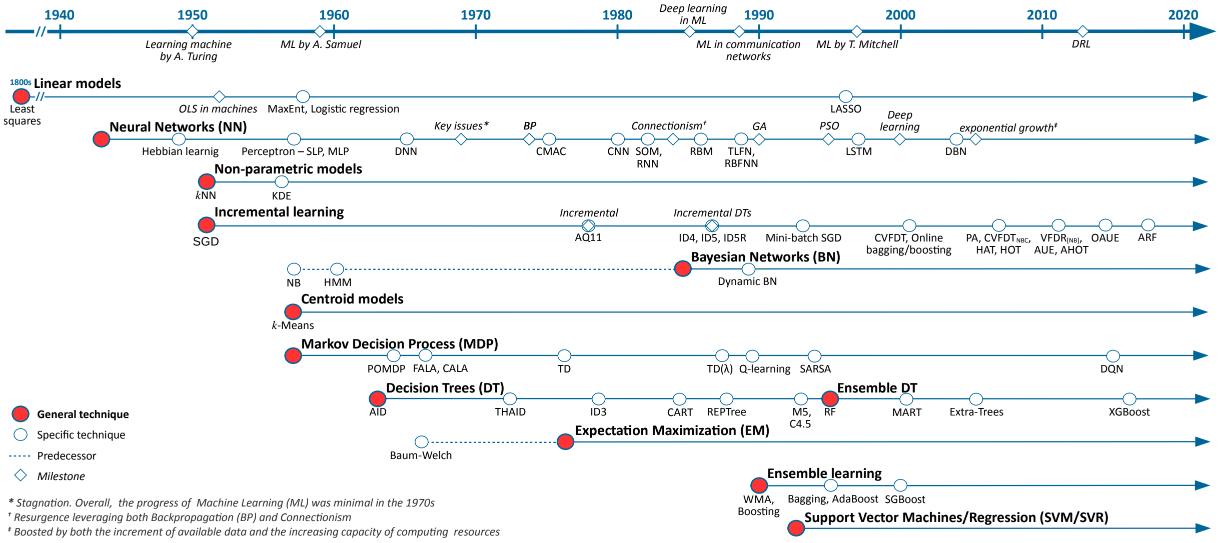


Figure 2.5: The evolution of machine learning techniques with key milestones

The beginning of ML dates back to 1943, when the first mathematical model of Neural Network (NN) for computers was proposed by McCulloch [107]. This model introduced a basic unit called artificial neuron that has been at the center of NN development to this day. However, this early model required to manually establish the correct weights of the connections between neurons. This limitation was addressed in 1949 by Hebbian learning [108], a simple rule-based algorithm for updating the connection weights of the early NN model. Like the neuron unit, Hebbian learning greatly influenced the progress of NN. These two concepts led to the construction of the first NN computer in 1950, called SNARC (Stochastic Neural Analog Reinforcement Computer) [97]. In the same year, Alan Turing proposed a test—where a computer tries to fool a human into believing it is also human—to determine if a computer is capable of showing intelligent behavior. He described the challenges underlying his idea of a *"learning machine"* in [109]. These developments encouraged many researchers to work on similar approaches, resulting in two decades of enthusiastic and prolific research in the ML area.

In the 1950s, the simplest linear regression model called Ordinary Least Squares (OLS)—derived from the least squares method [110,111] developed around the 1800s—was used to calculate linear regressions in electro-mechanical desk calculators [112]. To the best of our knowledge, this is the first evidence of using OLS in computing machines. Following this trend, two linear models for conducting classification were introduced: Maximum Entropy (MaxEnt) [113,114] and logistic regression [115].

Different ML techniques derived from pattern recognition and root-finding problems appeared during this decade too. Research trends centered on pattern recognition exposed two non-parametric models (i.e., not restricted to a bounded set of parameters) capable of performing regression and classification: k-Nearest Neighbors (kNN) [116, 117] and Kernel Density Estimation (KDE) [118], also known as Parzen density [119]. The former uses a distance metric to analyze the data, while the latter applies a kernel function (usually, Gaussian) to estimate the probability density function of the data. On the other hand, research on root-finding optimization introduced the stochastic approximation algorithm [120,121], which uses successive approximations to find the unique root of a regression function. This algorithm was later referred to as Stochastic Gradient Descent (SGD) [122] since it approximates the true gradient (computed using the whole dataset) by iteratively adding a scaled gradient estimate over every single instance of the dataset—a later proposed variant that iterate over instance subsets, called *mini-batches*, can improve performance and convergence [123]. SGD has become a well-known incremental (a.k.a. online) learning method for training various ML models, such as linear regression and NNs, facing large-scale datasets [124].

The 1950s also witnessed the first applications of the Naïve Bayes (NB) classifier in the fields of pattern recognition [125] and information retrieval [126]. NB, whose foundations date back to the 18th and 19th centuries [127,128], is a simple probabilistic classifier that applies Bayes' theorem on features with strong independence assumptions. NB was later generalized using KDE, also known as NB with Kernel Estimation (NBKE), to estimate the conditional probabilities of the features. In the area of clustering, Steinhaus [129] was the first to propose a continuous version of the to be called k-Means algorithm [130], to partition a heterogeneous solid with a given internal mass distribution into k subsets. The proposed centroid model employs a distance metric to partition the data into clusters where the distance to the centroid is minimized.

By the end of the 1950s, the Markov model [131, 132] (elaborated 50 years earlier) was leveraged to construct a process based on discrete-time state transitions and action rewards, named Markov Decision Process (MDP), which formalizes sequential decision-making problems in a fully observable, controlled environment [133]. MDP has been essential for the development of prevailing RL techniques [106]. Research efforts building on the initial NN model flourished too: the modern concept of perceptron was introduced as the first NN model that could learn the weights from input examples [134]. This model describes two NN classes according to the number of layers: Single-Layer Perceptron (SLP), an NN with one input layer and one output layer, and Multi-Layer Perceptron (MLP), an NN with one or more hidden layers between the input and the output layers. The perceptron model is also known as feedforward NN since the nodes from each layer exhibit directed connections only to the nodes of the next layer. Finally, the term *"Machine Learning"* was coined and defined for the first time by Arthur Samuel (see Section 2.2), who also developed a checkers-playing game that is recognized as the earliest self-learning program [135].

ML research continued to flourish in the 1960s, giving rise to a novel statistical class of the Markov model, named Hidden Markov Model (HMM) [136]. An HMM describes the conditional probabilities between hidden states and visible outputs in a partially observable, autonomous environment. The Baum-Welch algorithm [137] was proposed in the mid-1960s to learn those conditional probabilities. At the same time, MDP continued to instigate various research efforts. The Partially Observable MDP (POMDP) approach to finding optimal or near-optimal control strategies for partially observable stochastic environments, given a complete model of the environment, was first proposed by Cassandra *et al.* [138] in 1965, while the algorithm to find the optimal solution was only devised five years later [139]. Another development in MDP was the learning automata—officially published in 1973 [140]—an RL technique that continuously updates the probabilities of taking actions in an observed environment, according to given rewards. Depending on the nature of the action set, the learning automata is classified as Finite Action-set Learning Automata (FALA) or Continuous Action-set Learning Automata (CALA) [141].

In 1963, Morgan and Sonquis published Automatic Interaction Detection (AID) [142], the first regression tree algorithm that seeks sequential partitioning of an observation set into a series of mutually exclusive subsets, whose means reduces the error in predicting the dependent variable. AID marked the beginning of the first generation of Decision Tree (DT) models. However, the application of DTs to classification problems was only initiated a decade later by Morgan and Messenger's THeta AID (THAID) [143] algorithm.

In the meantime, the first algorithm for training MLP-NNs with many layers [9]—also known as Deep NN (DNN) in today's jargon—was published by Ivakhnenko and Lapa in 1965 [145]. This algorithm marked the commencement of the Deep Learning (DL) discipline, though the term only started to be used in the 1980s in the general context of ML, and in the year 2000 in the specific context of NNs [146]. By the end of the 1960s, Minsky and Papertkey's *Perceptrons* book [147] drew the limitations of perceptrons-based NN through mathematical analysis, marking a historical turn in Artificial Intelligence (AI) and ML in particular, and significantly reducing the research interest for this area over the next several years [97].

Although ML research was progressing slower than projected in the 1970s [97], this decade was marked by milestones that greatly shaped the evolution of ML, and contributed to its success in the following years. These include the Back-Propagation (BP) algorithm [148], the Cerebellar Model Articulation Controller (CMAC) NN model [149], the Expectation Maximization (EM) algorithm [150], the to-be-referred-to as Temporal Difference (TD) learning [151], the Iterative Dichotomiser 3 (ID3) algorithm [152], and the AQ11 learning system [153].

Werbos's application of BP—originally a control theory algorithm from the 1960s [154–156]—to train NNs [148] resurrected the research in the area. BP is to date the most popular NN training algorithm and comes in different variants [157], such as SGD (the *de facto* standard algorithm), conjugate gradient, one step secant, Levenberg-Marquardt, and resilient BP. Though, BP is widely used in training NNs, its efficiency depends on the choice of initial weights. In particular, BP has been shown to have slower speed of convergence and to fall into local optima. Over the years, global optimization methods have been proposed to replace BP, including Genetic Algorithms (GA), simulated annealing, and ant colony algorithms [158]. In 1975, Albus proposed CMAC, a new type of NN as an alternative to MLP [149]. Although CMAC was primarily designed as a function modeler for robotic controllers, it has been extensively used in RL and classification problems for its faster learning compared to MLP.

In 1977, in the area of statistical learning, Dempster *et al.* proposed EM, a generalization of the previous iterative, unsupervised methods, such as the Baum-Welch algorithm, for learning the unknown parameters of statistical HMM models [150]. At the same time, Witten developed an RL approach to solve MDPs, inspired by animal behavior and learning theories [151], that was later referred to as TD in Sutton's work [159, 160]. In this approach, the learning process is driven by the changes, or differences, in predictions over successive time steps, such that the prediction at any given time step is updated to bring it closer to the prediction of the same quantity at the next time step. Towards the end of the 1970s, the second generation of DT models emerged as the ID3 algorithm was released. The algorithm, developed by Quinlan [152], relies

[9] By 1971, the learning algorithm Group Method of Data Handling was capable of training an 8-layer MLP [144]

on a novel concept for attribute selection based on entropy[10] maximization. ID3 is a precursor to the popular and widely used C4.5 and C5.0 DT algorithms. In addition, Michalski and Larson developed AQ11 [153], a learning system that incrementally generates new rules using the existing ones and new training instances, later discarded after learning. AQ11 is the first evidence of using the term "incremental" in an ML technique, to the best of our knowledge.

The 1980s witnessed a renewed interest in ML research, and in particular in NNs. In the early 1980s, three new classes of NN models emerged, namely Convolutional Neural Network (CNN) [161], Self-Organizing Map (SOM) [162], and Hopfield network [163]. CNN is a feedforward NN specifically designed to be applied to visual imagery analysis and classification, and thus require minimal image preprocessing. Connectivity between neurons in CNNs is inspired by the organization of the animal visual cortex—modeled by Hubel in the 1960s [164,165]—where the visual field is divided between neurons, each responding to stimuli only in its corresponding region. Similarly to CNN, SOM was also designed for a specific application domain; dimensionality reduction [162]. SOMs employ an unsupervised competitive learning approach, unlike traditional NNs that apply error-correction learning (such as BP with gradient descent).

In 1982, the first form of Recurrent Neural Network (RNN) was introduced by Hopfield. Named after the inventor, Hopfield network is an RNN where the weights connecting the neurons are bidirectional. The modern definition of RNN, as a network where connections between neurons exhibit one or more than one cycle, was introduced by Jordan in 1986 [166]. Cycles provide a structure for internal states or memory allowing RNNs to process arbitrary sequences of inputs. As such, RNN are found particularly useful in time series forecasting, handwriting recognition, and speech recognition.

Several key concepts emerged from the 1980s' *connectionism movement*, one of which is the concept of *distributed representation* [167]. Introduced by Hinton in 1986, this concept supports the idea that a system should be represented by many features and that each feature may have different values. Distributed representation establishes a many-to-many relationship between neurons and *(feature,value)* pairs for improved efficiency, such that a *(feature,value)* input is represented by a pattern of activity across neurons as opposed to being locally represented by a single neuron. The second half of the 1980s also witnessed the increase in popularity of the BP algorithm and its successful application in training DNNs [168, 169], as well as the emergence of new classes of NNs, such as Restricted Boltzmann Machines (RBM) [170], Time-Lagged Feedforward Network (TLFN) [171], and Radial Basis Function (RBF) NN (RBFNN) [172].

Originally named Harmonium by Smolensky, RBM is a variant of Boltzmann machines [173] with the restriction that there are no connections within any of the network layers, whether it is visible or hidden. Therefor, neurons in RBMs form a bipartite graph.

[10] Measure of the uncertainty about a source of messages

This restriction allows for more efficient and simpler learning compared to traditional Boltzmann machines. RBMs are found useful in a variety of application domains such as dimensionality reduction, feature learning, and classification, as they can be trained in both supervised and unsupervised ways. The popularity of RBMs and the extent of their applicability significantly increased after the mid-2000s as Hinton introduced in 2006 a faster learning method for Boltzmann machines, called Contrastive Divergence [174], making RBMs even more attractive for DL [175]. Interestingly, although the use of the term "deep learning" in the ML community dates back to 1986 [176], it did not apply to NNs at that time.

As aforementioned, TLFN—an MLP that incorporates the time dimension into the model for conducting time series forecasting [171]—and RBFNN—an NN with a weighted set of RBF kernels trained in supervised or unsupervised ways [172]—joined the growing list of NN classes. Indeed, any of these NNs can be employed in a DL architecture, either by implementing a larger number of hidden layers or by stacking multiple simple NNs.

In addition to NNs, several other ML techniques thrived during the 1980s. Among these techniques, Bayesian Network (BN) arose as a Directed Acyclic Graph (DAG) representation for the statistical models in use [177], such as NB and HMM—the latter considered as the simplest dynamic BN [178, 179]. Two DT learning algorithms, similar to ID3 but developed independently, referred to as Classification And Regression Trees (CART) [180], were proposed to model classification and regression problems. Another DT algorithm, under the name of Reduced Error Pruning Tree (REPTree), was also introduced for classification. REPTree aimed at building faster and simpler tree models using information gain for splitting, along with reduced-error pruning [181]. DT also experienced its earliest incremental learning algorithms built upon batch models, mainly ID3 (e.g., ID4 [182], ID5 [183], and ID5R [184]), which greatly influenced the incremental DTs in the new millennium.

Towards the end of the 1980s, two TD approaches were proposed for RL: TD(λ) [160] and Q-learning [185]. TD(λ) adds a discount factor ($0 \leq \lambda \leq 1$) that determines to what extent estimates of previous state-values are eligible for updating based on current errors, in the policy evaluation process. For example, TD(0) only updates the estimate of the value of the state preceding the current state. Q-learning, however, replaces the traditional state-value function of TD by an action-value function (i.e., Q-value) that estimates the utility of taking a specific action in specific states. As of today, Q-learning is the most well-studied and widely-used model-free RL algorithm. By the end of the decade, the application domains of ML started expending to the operation and management of communication networks [186–188].

In the 1990s, significant advances were realized in ML research, focusing primarily on NNs and DTs. Bio-inspired optimization algorithms, such as GA and Particle Swarm Optimization (PSO), received increasing attention and were used to train NNs for improved performance over the traditional BP-based learning [189, 190]. Probably one of the most important achievements in NNs was the work on Long Short-Term Memory (LSTM), an RNN capable of learning long-term dependencies for solving DL tasks that involve long input sequences [191]. Today, LSTM is widely used in speech recognition as well as natural language processing. In DT research, Quinlan published the M5 algorithm in 1992 [192] to construct tree-based multivariate linear models analogous to piecewise linear functions. One well-known variant of the M5 algorithm is M5P, which aims at building trees for regression models. A year later, Quinlan published C4.5 [193], that builds on and extends ID3 to address most of its practical shortcomings, including data overfitting and training with missing values. C4.5 is to date one of the most important and widely used algorithms in ML and data mining.

Several techniques other than NNs and DTs also prospered in the 1990s. Research on regression analysis propounded the Least Absolute Selection and Shrinkage Operator (LASSO), which performs variable selection and regularization for higher prediction accuracy [194]. Another well-known ML technique introduced in the 1990s was Support Vector Machines (SVM). SVM enables plugging different kernel functions (e.g., linear, polynomial, RBF) to find the optimal solution in high-dimensional feature spaces. SVM-based classifiers find a hyperplane to discriminate between categories. A single-class SVM is a binary classifier that deduces the hyperplane to differentiate between the data belonging to the class against the rest of the data, that is, *one-vs-rest*. A multi-class approach in SVM can be formulated as a series of single class classifiers, where the data is assigned to the class that maximizes an output function. SVM has been widely used primarily for classification, although a regression variant exists, known as Support Vector Regression (SVR) [195]. In addition, SVM can learn incrementally using SGD, though this applies only to models using a linear kernel function. By 1999, an incremental learning variant of SVM appeared for handling perceived and real concept drifts [196].

In the area of RL, State-Action-Reward-State-Action (SARSA) was introduced as a more realistic, however less practical, Q-learning variation [197]. Unlike Q-learning, SARSA does not update the Q-value of an action based on the maximum action-value of the next state, but instead it uses the Q-value of the action chosen in the next state.

A new emerging concept called *ensemble learning* demonstrated that the predictive performance of a single learning model can be be improved when combined with other models [97]. As a result, the poor performance of a single predictor or classifier can be compensated with ensemble learning at the price of (significantly) extra computation. Indeed the results from ensemble learning must be aggregated, and a variety of techniques have been proposed in this matter. The first instances of ensemble learning include Weighted Majority Algorithm (WMA) [198], boosting [199], bootstrap aggregating (or bagging) [200], and Random Forest (RF) [201]. RF focused explicitly on tree models and marked the beginning of a new generation of ensemble DT. In addition, some variants of the original boosting algorithm were also developed, such as Adaptive Boosting (AdaBoost) [202] and Stochastic Gradient Boosting (SGBoost) [203].

These advances in ML facilitated the successful deployment of major use cases in the 1990s, particularly, handwriting recognition [204] and data mining [205]. The latter represented a great shift to data-driven ML, and since then it has been applied in many areas (e.g., , retail, finance, manufacturing, medicine, science) for processing huge amounts of data to build models with valuable use [98]. Furthermore, from a conceptual perspective, Tom Mitchell formally defined ML: “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E” [206].

The 21st century began with a new wave of increasing interest in SVM and ensemble learning, and in particular ensemble DT. Research efforts in the field generated some of the the most widely used implementations of ensemble DT as of today: Multiple Additive Regression Trees (MART) [207], extra-trees [208], and eXtreme Gradient Boosting (XGBoost) [209]. MART and XGBoost are respectively a commercial and open source implementation of Friedman’s Gradient Boosting Decision Tree (GBDT) algorithm; an ensemble DT algorithm based on gradient boosting [203, 207]. Extra-trees stands for *extremely randomized trees*, an ensemble DT algorithm that builds random trees based on k randomly chosen features. However instead to computing an optimal split-point for each one of the k features at each node as in RF, extra-trees selects a split-point randomly for reduced computational complexity.

At the same time, the popularity of DL increased significantly after the term “deep learning” was first introduced in the context of NNs in 2000 [146]. However, the attractiveness of DNN started decreasing shortly after due to the experienced difficulty of training DNNs using BP (e.g., vanishing gradient problem), in addition to the increasing competitiveness of other ML techniques (e.g., SVM) [98]. Hinton’s work on Deep Belief Networks (DBN), published in 2006 [210], gave a new breath and strength to research in DNNs. DBN introduced an efficient training strategy for DL models, which was further used successfully in different classes of DNNs [211, 212]. The development in ML (particularly, in DNNs) grew exponentially with advances in storage capacity and large-scale data processing (i.e., big data) [98]. This wave of popularity in DL has continued to this day, yielding major research advances over the years. One approach that has recently received tremendous attention is Deep RL (DRL), which incorporates DL models into RL for solving complex problems. For example, Deep Q-Networks (DQN)—a combination of DNN and Q-learning—was proposed for mastering video games [213]. Although the term DRL was coined recently, this concept was already discussed and applied 25 years ago [214, 215].

The 2000s have also been fruitful for the incremental learning research community. Although well-known ML techniques, such as NN and kNN, fit naturally to incremental learning, several algorithms emerged during the last 20 years, particularly for incremental DT. In 2001, the Very Fast Decision Tree (VFDT) algorithm [216] (a.k.a. Hoeffding tree) exposed an incremental DT that selects a root node from the first training instances and grows the tree from the leaf nodes. However, VFDT aimed for static concepts; hence, an extension of this algorithm, called Concept-adapting VFDT (CVFDT) [217], addresses concept drift by maintaining a set of alternate DTs and storing instances over a window of time. CVFDT experienced further refinements during the subsequent years. $\text{CVFDT}_{\text{NBC}}$ [218] incorporated NB classifiers in the leaf nodes of the tree, which demonstrated better accuracy than its predecessor. Hoeffding Adaptive Tree (HAT) [219] introduced ADaptive WINdowing (ADWIN) for monitoring the accuracy of the DT branches, which are replaced with more accurate branches when their performance decreases. Hoeffding Option Tree (HOT) [220] refers to an ensemble method that uses additional option nodes to build a single structure with multiple CVFDTs as separate paths. Adaptive Hoeffding Option Tree (AHOT) [103] extends HOT by storing in each leaf the error estimation, computed using Exponentially Weighted Moving Average (EWMA). Finally, Adaptive Random Forest (ARF) [221] provides another incremental ensemble DT, which adapts the widely used RF ensemble. In contrast to RF, ARF comes with a drift and warning detector per base DT that enables selective resets and tree replacements—when the warning becomes a drift.

Other non-strictly-DT incremental learning techniques also appeared during and after the 2000s. In 2001, Oza and Russell [222] developed incremental variants of the bagging and boosting ensemble algorithms, simply referred to as online bagging and online boosting, respectively. Both incremental ensembles allow using different learners as the base model, such as NB and any incremental DT, though recent implementations commonly use $\text{CVFDT}_{\text{NBC}}$. Five years later, the online Passive-Aggressive (PA) algorithms [223] characterized a family of margin-based incremental learning techniques for solving classification and regression problems. These algorithms (*passively*) keep the same weights when no loss is present but (*aggressively*) update them when the loss is positive. In 2011, two new approaches emerged: Very Fast Decision Rules (VFDR) [224] and Accuracy Updated Ensemble (AUE) [225]. VFDR introduced a rule-based incremental algorithm aiming at more interpretability and flexibility than DTs. This single-pass algorithm continuously learns ordered and unordered rule sets and classifies using the majority class strategy. A variant called VFDR_{NB} [224] incorporates NB classifiers that improve the accuracy. On the other hand, AUE presented an adaptive block-based ensemble for incremental learning, which selects and updates the base classifiers regarding the current distribution in a block series. An improvement to AUE, called Online Accuracy Updated Ensemble (OAUE) [226], introduced a new error-based weighting function to incrementally train and evaluate the base classifiers. Similar to online bagging and boosting, recent implementations of both AUE and OAUE commonly use $\text{CVFDT}_{\text{NBC}}$ as the base learner.

It is important to mention that the evolution in ML research has enabled improved learning capabilities which were found useful in several application domains, ranging from games, image and speech recognition, network operation and management, to self-driving cars [227].

2.2.3 Cognitive networking

In theory, ML can be used for automating network operations and management as it allows extracting knowledge from data. However, application of ML for incorporating intelligence and autonomy in networking is a non-trivial task. Prohibiting factors include the distributed control and vendor-specific nature of legacy network devices, lack of available data, and cost of compute and storage resources. Several technological advances have been made in the last decade to overcome these limitations. The advent of network softwarization and programmability through SDN and NFV offers centralized control and alleviates vendor lock-in. The advances in ML along with the proliferation of new sources of data and big data analytics platforms provide abundant data and extract knowledge from them. Furthermore, the availability of seemingly *infinite* storage and compute resources through the cloud overcomes the cost of resources. These together provide the environment to realize the vision of cognitive networks.

Knowledge-Defined Networking (KDN) [228] depicts a recent initiative for cognitive networking. This approach revisits the inclusion of a *knowledge plane* proposed initially 15 years before for the Internet [229]. KDN defines the knowledge plane at the top of the SDN architecture (see Section 2.1), replacing the application plane; hence, it can interact with the management and control planes to obtain a rich view and control over the network. The knowledge plane enables learning from the network behavior by processing operation and management data collected by the other SDN planes. Via ML techniques, such data become knowledge aimed at providing recommendations and making network decisions—either automated or human intervention.

In the context of autonomic systems and networks, IBM's autonomic computing architecture [230] is to date the most influential reference model. It comprises several layers of autonomic managers. The behavior of each manager is governed by the MAPE control loop that consists of four functions; *Monitor*, *Analyze*, *Plan*, and *Execute*. As shown in Figure 2.6, the *Knowledge* source is orthogonal to every MAPE function. Functions can retrieve data from and/or log created knowledge to the Knowledge source. For example, the Analyze function obtains information about the historical behavior of a managed resource and stores the ML models and the analytics it generates in the Knowledge source.

In [230], we observe that cognition has been restricted to the Analyze function, which inhibits the ability to achieve closed-loop cognitive network management. In [42], we proposed to incorporate cognition at every function in the loop. For example, the Monitor function should be able to determine the what, when, and where to monitor.

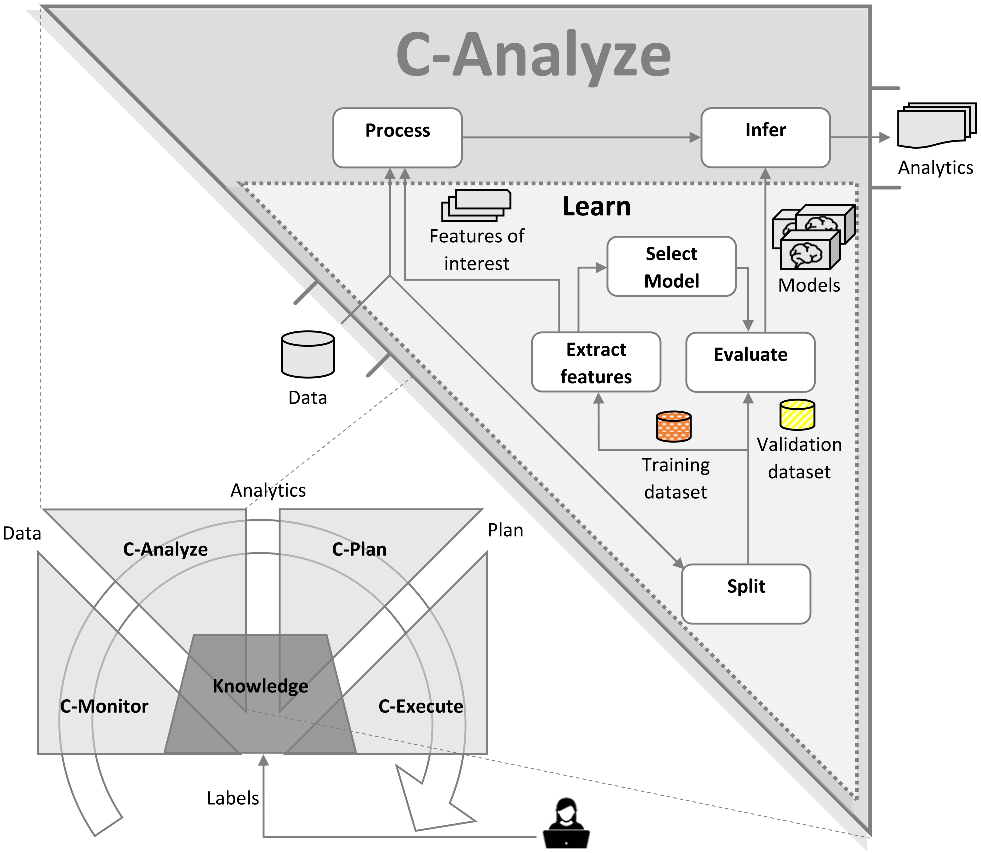


Figure 2.6: Cognitive control loop for network management

ML can be leveraged to build this cognition in every function and allow each function to operate in full autonomy. Therefore, we extend IBM's MAPE control loop into a cognitive control loop that we denote as C-MAPE. As illustrated in Figure 2.6, cognition is achieved by introducing learning and inference in every function.

- *C-Monitor* function refers to the cognitive monitor that performs intelligent probing. For instance, when the network is overloaded, the C-Monitor function may decide to reduce the probing rate and instead perform regression for data prediction.
- *C-Analyze* function is responsible for detecting or predicting changes in the network environment (e.g., faults, policy violations, frauds, performance degradation, and attacks). ML has been leveraged to address some of these challenges in each of the FCAPS management areas, as discussed in [42].
- *C-Plan* function can leverage ML to develop an intelligent Automated Planning (AP) engine that reacts to changes in the network by selecting or composing a change plan. In the last decade, AP systems have been applied to real-world problems and have been relying on ML for automating the extraction and organization of knowledge (e.g., plans, execution traces), and for decision making [231].

- *C-Execute* function can use ML to schedule the generated plans and determine the course of action should the execution of a plan fail. These tasks lend themselves naturally to RL where the C-Execute agent could *exploit* past successful experiences to generate optimal execution policies, and *explore* new actions should the execution plan fail.

Closing the control loop is achieved by monitoring the state of the network to measure the impact of the change plan.

2.3 Traffic engineering in data center networks

This section first reviews the concepts of DCN and traffic engineering to contextualize the approach of this dissertation. Then, it provides a literature review about multipath routing for load-balancing in DCNs, focusing on the seminal works that use SDN and ML for addressing such a challenge.

2.3.1 Data center network

DCN encompasses the communication infrastructure of data centers that aim to interconnect a large number of servers with significant bandwidth capacity in order to achieve high throughput and low-latency [1]. Different DCN topologies have been designed to meet such performance requirements. In general, these DCN topologies can be classified into three categories based on the routing and switching equipment used to forward or process network traffic [232–234]: *switch-centric*, *server-centric*, and *hybrid*. Switch-centric topologies consider switches as the only relay nodes (i.e., routing decisions) and servers as mere endpoints, such as VL2 [235] and Jellyfish [236]. Server-centric topologies define servers as both endpoints and relaying nodes, such as BCube [237] and DCell [238]. Hybrid topologies use a combination of electrical, optical, and/or wireless equipment to add extra bandwidth capacity to the DCN, such as FireFly [239] and Helios [240].

Currently, most of the DCN deployments follow the switch-centric topology, particularly the tree-based design, such as Facebook's Altoona [8] and Google's Jupiter [241]. In switch-centric, the switches are interconnected in a hierarchical model with multiple layers and the servers are connected to the switches of the lowest layer, known as *edge* or Top-of-Rack (ToR). The tree-based design is an instance of the Clos network with a degree defined according to the network scale indicating the number of layers. For example, VL2 [235] and Fat-tree [242] arrange low-cost commodity switches in a tree-based topology with fourth-degree, namely, from bottom-up, one layer of servers and three layers of switches: *edge*, *aggregation*, and *core* (see Figure 2.7). Differing from this tree-based design, Jellyfish [236] uses a random regular graph to interconnect the switches at the edge layer. Although Jellyfish provides some benefits over the tree-based topologies, such as higher capacity, shorter paths, and more resilience to failures and wiring errors, it is more challenging and complex in terms of cabling, management, and routing.

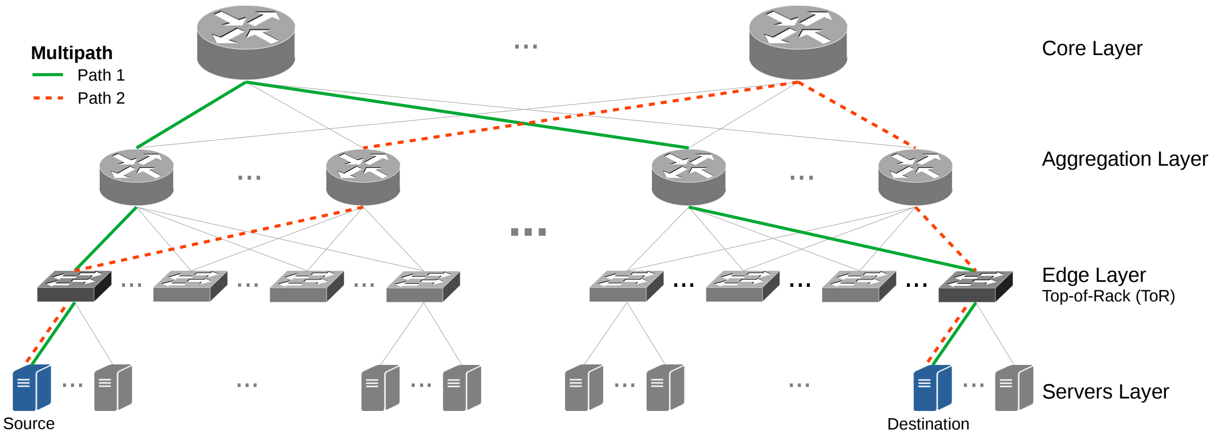


Figure 2.7: Tree-based DCN with fourth degree and multipath routing

In addition, several research works have analyzed the traffic characteristics of switch-centric tree-based DCNs, converging to similar patterns [8, 9, 11]: (i) 10% to 25% of links are hot-spots, varying over time; (ii) less than 25% of capacity is utilized, even with over-subscription; (iii) around 80% of flows (i.e., mice) carry less than 10% of total bytes, last less than 10 seconds, and transmit less than 10KB, while 20% of flows (i.e., elephants) carry almost 80% of total bytes, last up to 500 seconds, and transmit up to 5GB; (iv) almost every flow at the ToR switch presents an inter-arrival time less than 100 milliseconds; and (v) the number of active flows can be up to 10,000 flows per second.

2.3.2 Traffic engineering

Traffic engineering involves the methods for measuring and managing network traffic to optimize the performance of the network [3, 243]. This optimization requires providing appropriate traffic requirements (e.g., throughput, delay, packet loss) while efficiently—in terms of cost and reliability—utilizing network resources (e.g., bandwidth).

Traffic engineering encompasses four main dimensions [49]: flow management, fault-tolerance, topology update, and traffic analysis. Flow management is about mapping and controlling the traffic flows in the network for optimizing the routing function to steer traffic (from ingress nodes to egress nodes) in the most effective way. Fault-tolerance refers to ensuring network reliability by providing mechanisms that enhance network integrity and by embracing policies emphasizing network survivability. Topology update involves managing the capacity of the network in order to carry out planned changes, such as network policy modifications. Traffic analysis deals with monitoring the performance of the network and verify the compliance with network performance goals to evaluate and debug the effectiveness of the applied traffic engineering methods. Each dimension may operate at multiple levels of temporal resolution, ranging from a few nanoseconds to possibly years. For example, topology update works at very coarse temporal levels, from days to years, while flow management operates at finer levels of temporal resolution, from microseconds to hours.

Load-balancing is one of the most well-known traffic engineering methods. As part of the flow management dimension, load-balancing controls and optimizes the routing function to minimize the maximum load across the links [2]. The goal is mapping traffic flows from the heavily loaded paths to the lightly loaded paths for avoiding congestion (i.e., hot-spots) and increasing network throughput and resource utilization. *Multipath routing* has shown to effectively achieve load-balancing by distributing traffic over multiple concurrent paths such that all the links are optimally loaded [4]. Figure 2.7 depicts two disjoint[11] paths in a tree-based DCN. In practice, multipath routing protocols may split the traffic at different levels of granularity, such as per-flow, per-sub-flow, and per-packet. In addition, these protocols may run at distinct layers of the TCP/IP model. For example, ECMP [7] and valiant load balancing [244] work at the network layer, while Multi-Path TCP (MPTCP) [245] operates at the transport layer.

2.3.3 Multipath routing in data center networks

From a general perspective, multipath routing in DCNs divides into two categories: distributed and SDN-based (i.e., centralized). Initially, distributed approaches were the standard for multipath routing. However, with the emergence of centralized network programmability, SDN-based multipath routing has gained the same level of attention. Recent investigations that have adopted the SDN-based approach aim to optimize the routing function by using ML techniques for predicting flow traffic characteristics.

Distributed multipath routing

Distributed multipath routing mechanisms place routing decisions at either the switches or servers of a DCN. These routing decisions can be oblivious to traffic conditions or based on feedback information from the network (e.g., congestion). In addition, traffic splitting in distributed multipath routing is conducted at different levels of granularity (e.g., flow, packet, sub-flow). The following paragraphs describe the seminal works focused on distributed multipath routing.

[11] No common nodes or links except for source and destination.

ECMP [6,7] represents the state-of-the-art distributed mechanism for multipath routing in DCNs. ECMP is oblivious to traffic conditions and splits the traffic at the level of flows. Generally, ECMP applies a hash function at every switch on selected packet headers to assign each incoming flow to an output port. The output port belongs to one of the several equal-cost forwarding paths maintained by the switch for reaching a destination. A key limitation is that the broad distribution of flow sizes in DCNs (i.e., mice and elephants) cause hot-spots in ECMP-based routing [9, 11]. This is because two or more elephant flows can collide on their hash and end up on the same output port. Since ECMP does not account for either current network utilization or flow size, the resulting collisions overwhelm switch buffers and degrade overall switch and link utilization. In addition, the performance of ECMP degrades significantly during asymmetric topologies caused by link failures.

Despite ECMP limitations, it remains as the standard multipath routing mechanism for load-balancing in today's DCNs [5]. For example, Facebook's DCN in Altoona [12] employs BGP to populate the routing tables and ECMP for routing through the equal-cost paths.

Motivated by the drawbacks of ECMP's flow splitting, Random Packet Spraying [13] and Digit Reversal Bouncing (DRB) [14] split the traffic at the level of packets through the different equal-cost paths. However, these works rely on an ideal symmetry of the network to avoid the adverse effects of packet reordering[12] on TCP. To deal with the asymmetry caused by failures, DRB includes a topology update mechanism, though the distributed nature of this mechanism is not suitable for large-scale DCNs.

Other approaches have also addressed asymmetry by proposing topology-dependent weighing of paths and using distinct granularity levels for splitting traffic. For example, Weighted Cost Multiple-Path (WCMP) [16] extends ECMP to support weighted flow-level splits at switches by repeating the same next-hop multiple times. Whereas, Presto [15] implements weights at the servers and splits flows at the level of TCP Segmentation Offload (TCO) units, termed as *flowcells*. However, WCMP and Presto rely on static weights that are generally sub-optimal with asymmetry, particularly for dynamic traffic workloads. In terms of traffic splitting, WCMP inherits the elephant collision problem from ECMP, while the sub-flow splitting defined by Presto is prone to packet reordering. Moreover, neither of these distributed multipath routing mechanisms is aware of traffic conditions, causing degraded performance during link failures.

In light of these gaps, some works have used knowledge of congestion on different paths to make routing decisions. LocalFlow [18] and Flare [19] rely on local measurements of traffic congestion to balance the load on the switch ports. However, they lack taking global congestion information into account, yielding sub-optimal results for high varying traffic. Therefore, further approaches use feedback from the network to

[12] Packet reordering can cause TCP to unnecessarily reduce the sender's rate, leading to severe throughput inefficiencies.

gather path-wise congestion information and shift traffic to less-congested paths. The congestion feedback can be collected either at the servers (e.g., FlowBender [17], LetFlow [20], MPTCP [245]) or switches (e.g., HULA [21], CONGA [22], DeTail [246]) of the network. These global congestion-aware mechanisms achieve better throughput and delay results in DCNs, yet at the cost of significant implementation complexity or specialized hardware support—software implementation produces sub-optimal results—at the servers or switches of the network.

On the other hand, the level of granularity for splitting traffic in congestion-aware multipath routing is variable. FlowBender works at flow-level, DeTail at packet-level, and the rest at sub-flow-level. Particularly, LocalFlow conducts a *spatial* flow splitting based on TCP sequence numbers, while MPTCP splits a TCP flow into multiple sub-flows by varying port numbers. Flare, CONGA, HULA, and LetFlow rely on the concept of *flowlets* [247], defined as a burst of packets from a flow separated by enough time gaps. As discussed before, the flow splitting carries the elephant collision problem, while the packet and sub-flow splitting potentially cause packet reordering. The latter because sub-flow splitting lack an exhaustive study about the appropriate space or time between sub-flows for achieving the best performance without causing packet reordering.

Table 2.1 outlines the limitations of distributed multipath routing. To summarize, the distributed multipath routing mechanisms based on global congestion-awareness and sub-flow splitting (particularly, flowlets) provide great results for load-balancing traffic in DCNs. However, they require the implementation of specialized hardware in the network, increasing the capital and operational expenditure for deploying a DCN. Moreover, these distributed multipath routing mechanisms rely on a static separation between sub-flows, which is not suitable for the varying traffic in DCNs.

Multipath routing based on software-defined networking

SDN-based multipath routing mechanisms leverage the potential of network programmability for enabling a centralized controller to make routing decisions in a DCN. The controller has a global view of the network status and instructs the switches for packet forwarding. The routing decisions rely on *flow detection* methods that classify flows, mostly into two classes: mice and elephants. Such classification occurs either at the controller-side (e.g., by pulling or sampling traffic statistics), switch-side, or server-side of SDDCNs. Moreover, some flow detection methods have leveraged ML techniques to classify flows proactively (i.e., prediction). In the following, the seminal works centered on Software-Defined Networking-based multipath routing are discussed.

Hedera [26] represents the state-of-the-art Software-Defined Networking-based mechanism for multipath routing in DCNs. Hedera defines a centralized controller that periodically pulls flow statistics from ToR-switches to discriminate elephant flows from mouse flows (i.e., binary classification). By default, Hedera assumes all flows as mice, forward-

Table 2.1: Summary of the limitations of distributed multipath routing

Perspective	Approach	Gaps
Traffic conditions awareness	Oblivious [7, 13–16]	- Performance degradation due to asymmetry of network topology caused by link failures - Routing decisions not based on traffic conditions
	Local congestion [18, 19]	- Routing decisions not based on global congestion data - Require specialized hardware implementation
	Global congestion [17, 20–22, 245, 246]	- Significant implementation complexity - Require specialized hardware implementation
Traffic splitting granularity	Flow-level [7, 16, 17]	- Performance degradation due to hot-spots caused by collisions of large, long lived flows (i.e., elephant flows)
	Packet-level [13, 14, 246]	- Potential packet reordering - Rely on the symmetry of the network for avoiding packet reordering
	Sub-flow-level [15, 18–22, 245]	- No guarantee that two distinct sub-flows take different paths - Potential packet reordering - Unclear about the spatial or time space between sub-flows for avoiding packet reordering

ing them using ECMP, until a pre-defined threshold rate (10% of bandwidth, $100Mbps$ in implementation) is reached. Hedera then executes a simple routing algorithm that dynamically computes a suitable path for the detected elephant flow and installs the corresponding rules on the switches along that path. PMCE [248] provides an enhanced routing algorithm for improving the performance of Hedera.

However, the short inter-arrival time of flows at ToR-switches ($< 100ms$) [8, 9, 11] requires a high rate of statistics pulling for achieving good performance (e.g., $< 500ms$ to perform better than ECMP [245]). This high rate along with the huge amount of active flows in ToR-switches (up to $10,000$ flows) greatly increases traffic overhead and controller processing, negatively affecting the performance of the network traffic. Moreover, since Hedera installs a flow rule per each active flow in the ToR-switches, the limited memory capacity of switches restricts its scalability. Furthermore, Hedera's binary classification is too simple for the wide distribution of flow sizes in DCNs (i.e., tens of bytes to hundreds of megabytes), causing the routing algorithms to make inaccurate decisions regarding the traffic volume across a forwarding path.

To address the traffic overhead shortcoming of Hedera, Devoflow [27] reduces the number of interactions between the controller and switches by introducing a rule cloning action for wildcard OpenFlow rules. Each cloned wildcard rule includes a trigger (i.e., a counter and a comparator) based on a pre-defined threshold ($128KB$, $1MB$, or $10MB$) that enables the switches to identify elephant flows. The detected elephant flows are sent to the controller, which calculates the least congested path. The mouse flows are locally handled by the switch through multipath and rapid re-routing actions (included as *select* and *fast failover* group types in OpenFlow v1.5.1 [55], respectively). A key limitation is that the rule cloning action is not supported in OpenFlow, therefore, DevoFlow requires a specialized switch hardware implementation. In addition, although the latest version of OpenFlow (i.e., 1.5.1) support threshold-based triggers, setting a trigger for each flow would limit the scalability and impose a lot of processing to the switch. On the other hand, the limitations of conducting a simple binary classification of flow sizes remain in Devoflow.

Similarly, Mahout [29] tackles the traffic overhead shortcoming of Hedera by monitoring and detecting elephant flows at the server-side of SDDCNs through a shim layer in the operating system. When an elephant flow is detected, the server marks the subsequent packets of the detected elephant flow. ToR-switches recognize these marked packets and send the first of each flow to the controller. The controller then computes and installs a path for the marked packets. As in Hedera, the packets without marks (i.e., mice) are routed using ECMP. Mahout greatly reduces traffic overhead, however, at the cost of software modifications in the servers of SDDCNs. Moreover, inaccurate routing decisions also appear in Mahout since it relies on a simple binary classification of flow sizes. MiceTrap [28] employs the same mechanism of Mahout for identifying and handling elephant flows, while proposing an aggregation module for managing mice flows. This mice aggregation module reduces the number of rules installed on switches for handling mice flows. Nevertheless, the shortcomings of Mahout persist in MiceTrap.

The Elephant Sensitive Hierarchical Statistics Pulling (ESHSP) approach [249] proposes an iterative process to detect elephant flows by decomposing the flow space until an elephant flow is isolated from the others. ESHSP uses a combination of aggregate and individual statistics messages of OpenFlow to reduce the traffic overhead generated by Hedera-like approaches. However, the traffic overhead produced by ESHSP is significantly higher than switch-side and server-side approaches, though it does not require hardware or software modifications in SDDCNs. While ESHSP does not perform further actions with the detected elephant flows, the routing algorithms defined in works like Hedera and Mahout can be easily integrated to it. In addition, ESHSP also suffer from the problems of relying on a simple binary classification of flow sizes.

Sampling is another method that has been used for collecting data to detect elephant flows without requiring hardware or software modifications. Sampling-based DevoFlow [27] and TinyFlow [25] adopt the packet sampling mechanism from sFlow [250] to identify elephant flows in DCNs. SDEFIX [251] has also integrated sFlow for detecting elephant flows, though in Internet exchange points. In particular, DevoFlow exposes sampling as an alternative to the threshold-based triggers for cloned wildcard rules. However, this sampling-based DevoFlow rely on a flow-level simulation that does not simulate actual packets but assume a distribution of packets as reported by earlier works [252]. Therefore, the evaluation of the sampling-based DevoFlow is biased and might not represent the limitations of using sampling for detecting and scheduling elephant flows.

In the same context of SDDCNs, TinyFlow routes all flows using ECMP by default until a pre-defined threshold is exceeded. Once an elephant flow is detected through sampling, the TinyFlow controller installs a new rule on the ToR-switch that monitors the byte count of that flow. When the byte count exceeds a pre-defined threshold, the switch resets the byte count and chooses a different egress port to route the elephant flow through a different equal-cost path. A key limitation of TinyFlow is that it imposes processing overhead on the switch for monitoring each elephant and changing the egress port accordingly, which was not evaluated. On the other hand, the elephant detection based on sampling generates a traffic overhead lower than pulling statistics, though still significantly higher than switch-side and server-side approaches. In addition, since only a small fraction of packets are sampled (typically, 1 in 1000), the elephant detection is not as accurate as in the other approaches.

A common problem of these Software-Defined Networking-based multipath routing mechanisms is that they can only detect elephant flows reactively when their traffic characteristics (e.g., flow size or flow rate) surpass a pre-defined static threshold. This is not ideal since hot-spots may occur until traffic characteristics are detected and new paths are chosen by the routing algorithms. For this reason, recent works have incorporated ML techniques at the controller-side of SDDCNs to predict traffic characteristics and to make routing decisions proactively.

Early ML-based approaches implement flow size prediction methods that learn from centralized data collected by periodically pulling flow statistics from switches (similar to Hedera). Xiao *et al.* [253] introduces a cost-sensitive C4.5 DT classifier. The accuracy of this classifier heavily depends on the choice of the costs for identifying the class of a flow (i.e., mouse or elephant). Experiments conducted on two real datasets, from a trans-Pacific line [254] and a private data center edge link, demonstrated that incorporating cost-sensitive to C4.5 improves its accuracy for elephant detection. Similarly, the Online Flow Size Prediction (OFSP) approach [32] explores different ML techniques for elephant flow detection. These include gaussian regression, BN, and NN. Experiments were performed on three public real datasets from two university DCNs [255] and an academic building at Dartmouth College [256] with over three million flows each. In contrast to the previous work, OFSP employs the predicted flow size for making routing decisions. Presumably mouse flows are routed through ECMP, whereas elephant flows are routed through the least congested path. However, both approaches assume that the data is centralized, requiring the centralized controller to periodically pull flow statistics from the switches. This collection of data increases traffic overhead, which exponentially grows as these approaches require per-packet headers. In addition, as other non-ML approaches, the problems of a static threshold for simple binary classification of flow sizes remain.

Other ML-based approaches applied similar learning algorithms but on data collected by sampling. The Efficient Sampling and Classification Approach (ESCA) [35] uses sampling for collecting data and proposes a two-phase elephant flow detection. In the first phase, the approach improves sampling efficiency by estimating the arrival interval of elephant flows and filtering out redundant samples using a filtering flow table, which requires modifications in the OpenFlow specification. In the second phase, the approach classifies samples with a new supervised classification algorithm based on the C4.5 DT. Once classified a flow, a differentiated scheduling approach called DIFFERENCE [34] searches for the best path using a specific algorithm for each flow class: a blocking-island-based algorithm for elephants and a weighted multipath algorithm for mice. Extensive experiment results demonstrated that ESCA can provide accurate detection with less sampled packets and shorter detection time than sFlow-based approaches (e.g., DevoFlow and TinyFlow). However, the proposed sampling method depends on non-existing SDN specifications, hence, requiring custom-made switch hardware.

FlowSeer [33] also leverages sampling and DTs for performing a two-phase cooperative classification for elephant detection. In the first phase, the controller applies a simple DT algorithm (i.e., C4.5) for identifying potential elephant flows. The statistics for this phase are collected by an unsupervised flow sampling method based on wildcard rules. Then, the switches are instructed to forward the headers of the first five packets of the potential elephant flows to the controller. In the second phase, the controller use an incremental DT algorithm (i.e., Hoeffding tree) to detect true elephant flows from the potential ones and further classify them into five categories regarding their size and duration. The median of the classified range is used by the routing algorithm as the predicted demand for computing the best path. Although a five-class classification is much better than a binary classification, a finer granularity prediction is desirable for improving the decisions of the routing algorithm. In addition, FlowSeer carries the limitations of sampling statistics collection, including inaccurate elephant detection and moderate traffic overhead.

All these mechanisms for detecting elephant flows are pre-configured with a fixed threshold value, which might cause high detection error rates when working with the frequently changing traffic of DCNs. Motivated by this shortcoming, Liu *et al.* [257] introduce an adaptive approach for elephant flow detection by adopting a dynamical traffic learning algorithm to configure the threshold value. Although the results show improvement compared to other methods, this work relies on pulling flow statistics, which generates high traffic overhead, and conducts a simple binary classification of flow sizes.

Rather than flow detection methods, some SDN-based multipath routing approaches have used the short-term and partial predictability of the traffic matrix to make routing decisions. MicroTE [258] proposes software modifications into each server (similar to Mahout) for incorporating a monitoring component that collects network traffic. Only one server per rack is responsible for aggregating, processing, and summarizing the network statistics for the entire rack. This designated server then determines the matrix traffic predictability and communicates this information to the controller. The controller computes paths for predictable traffic, while the unpredictable traffic is routed in a weighted form of ECMP. Similarly, Nie *et al.* [259] predicts and estimates the traffic matrix in DCNs but applying a DL technique (i.e., DBN). The authors demonstrate through simulations that the proposed method can capture the short time scale property of traffic flows faithfully. However, the bursty nature of the traffic in DCNs makes traffic matrix prediction questionable. Moreover, the short time scale predictability of the traffic matrix $(1-2s)$ might not be enough for the whole process: sending information to the controller, making routing decisions, and installing the computed paths on the switches. This concern is more severe for the DBN approach than for MicroTE due to the time and processing requirements for training DL techniques. Further investigation is required to clarify these questions.

Table 2.2 summarizes the limitations of the reviewed SDN-based multipath routing mechanisms. In general, SDN-based multipath routing have demonstrated improvements over the state-of-the-art multipath routing (i.e., ECMP). However, the existing mechanisms lack finding the best trade-off between traffic overhead, data collection accuracy, and network modifications. In addition, there is still room for improving the routing decisions of SDN-based multipath routing by analyzing finer traffic characteristics.

2.4 Final remarks

First, this chapter detailed the traditional three-plane architecture for deploying an SDN-based network as well as the later inclusion of the management plane in the architecture. We extended such a management plane for integrated control and monitoring of heterogeneous SDNs in one or more domains. Our management plane included an information model that leveraged either CIM or YANG to represent the SDN architecture. We also defined our management plane in the context of network automation by introducing an SDN management architecture based on HTN and NFV. Subsequently, the chapter provided a primer on ML, which discussed different categories of ML-based approaches, including supervised, unsupervised, reinforcement, batch, and incremental learning. This primer also included a brief history of ML techniques used in networking and the vision of cognitive networks towards automating operations and management. The latter included our C-MAPE approach that incorporates cognition in every function of IBM's autonomic computing control loop. Finally, this chapter exposed the concepts

Table 2.2: Summary of the limitations of SDN-based multipath routing

Perspective	Approach	Gaps
Traffic analysis location	Centralized by pulling [26, 32, 248, 249, 253, 257, 259]	- High traffic overhead - High controller processing - Low resource scalability
	Centralized by sampling [25, 27, 33–35, 251]	- Moderate traffic overhead - Moderate controller processing - Inaccurate collection of traffic statistics
	Distributed at switches [27]	- Specialized hardware - Low resource scalability - High switch processing
	Distributed at servers [28, 29, 258]	- Software modifications
Traffic characteristics	Binary classification of flow sizes - *non-ML* [25–29, 248, 249, 251] - *ML-based* [32, 34, 35, 253]	- Static threshold to identify elephants - Coarse classification of flow sizes (two classes) - Reactive detection of traffic characteristics *(non-ML approaches)*
	Multiclass classification of flow sizes based on ML [33]	- Moderate classification of flow sizes (four to five classes)
	Adaptive binary classification of flow sizes based on ML [257]	- Coarse classification of flow sizes
	Prediction of traffic matrix - *non-ML* [258] - *ML-based* [259]	- Not suitable for the bursty traffic in DCNs - Short time predictability of traffic matrix

and a literature review related to multipath routing for load-balancing in DCNs, focusing on the seminal works that use SDN and ML for addressing such a challenge. From the literature review, we can conclude that SDN-based multipath routing have demonstrated improvements over the prevalent ECMP. However, the existing SDN-based mechanisms lack finding the best trade-off between traffic overhead, data collection accuracy, and network modifications. Furthermore, finer traffic characteristics is desirable for improving the routing decisions of SDN-based multipath routing.

Chapter 3

Flow detection using online incremental learning at the server-side of software-defined data center networks

Data centers provide significant bandwidth capacity for a large number of servers interconnected by a specially designed network, called DCN [1, 260]. This bandwidth capacity can be optimized by using *multipath routing*, which distributes traffic over multiple concurrent paths [4]. Nowadays, ECMP is the default multipath routing mechanism for DCNs [5]. ECMP uses in every router a hash function on packet headers to assign each incoming flow to one of the equal-cost forwarding paths for reaching a destination. However, ECMP can degrade the performance of DCNs due to the coexistence of many small, short-lived flows (i.e., mice) and few large, long-lived flows (i.e., elephants), since ECMP can assign more elephant flows to the same path, generating *hot-spots* (i.e., some links overused while others underused). Flows traversing hot-spots suffer from low throughput and high latency.

Recent multipath routing mechanisms have leveraged SDN to face the ECMP limitations; DCNs using SDN are referred to as SDDCNs. SDN allows a logically centralized controller to dynamically make and install routing decisions on the basis of a global view of the network [24]. SDN-based multipath routing dynamically reschedules elephant flows, while handling mouse flows by employing default routing such as ECMP [5] and our pseudo-MAC-based approach (see Section 4.1). Reactive *flow detection* methods, which are at the heart of SDN-based mechanisms, discriminate elephants from mice by using static thresholds [26, 27, 29]. However, reactive methods are not suitable for SDDCNs since hot-spots may occur before the elephant flows are detected.

Novel SDN-based flow detection methods incorporate ML for proactively identifying elephant flows. However, ML-based methods operate at the *controller-side* of

SDDCNs, requiring the central collection of either *per-flow* data [32] or *sampling-based* data [33,35]. The central collection of per-flow data, however, causes problems such as heavy traffic overhead and poor scalability. Sampling-based data, on the other hand, tends to provide delayed and inaccurate flow information. Moreover, sampling techniques that mitigate the problem rely on non-standard SDN specifications. Using ML on either the *switch-side* or *server-side* represents a potential solution to the controller-side problems since these locations enable prompt and per-flow data with low traffic overhead. Switch-side flow detection methods based on ML are impractical because they require specialized hardware and put a heavy processing load on the switches. Conversely, ML-based flow detection methods at the server-side require only software modifications in the servers; nonetheless, these methods have not been fully explored.

In this chapter, we propose a novel flow detection method denominated Network Elephant Learner and anaLYzer (NELLY), which applies online incremental learning at the server-side of SDDCNs for accurately and timely identifying elephant flows while generating low control overhead. Incremental learning allows NELLY to constantly train a flow size classification model from continuous and dynamic data streams (i.e., flows), providing a constantly updated model and reducing time and memory requirements. Thus, NELLY adapts to the variations in traffic characteristics and performs endless learning with limited memory resources. We extensively evaluate NELLY using datasets extracted from real packet traces and incremental learning algorithms. Quantitative evaluation demonstrates that NELLY is efficient in relation to accuracy and classification time when adaptive decision trees algorithms are used. Analytic evaluation corroborates that NELLY is scalable, causes low traffic overhead, and reduces detection time, yet it is in conformance with SDN standards.

The remainder of this chapter is as follows. Section 3.1 introduces the architecture of NELLY. Section 3.2 presents a quantitative evaluation of NELLY using incremental learning algorithms and real packet traces. Section 3.3 compares NELLY to other related work. Section 3.4 concludes the chapter.

3.1 Architecture of NELLY

Figure 3.1 introduces NELLY, a flow detection method that applies online incremental learning at the server-side of SDDCNs to identify elephant flows accurately in a reasonable time while generating low control overhead. NELLY operates as a software component either in the kernel of the host operating system or in the hypervisor of servers in the SDDCN with the aim of monitoring all packets sent by the applications, containers, and virtual machines. Since NELLY detects elephant flows at their origin, a small overhead is demanded.

The architecture of NELLY (see Figure 3.1) presents two subsystems: *Analyzer*

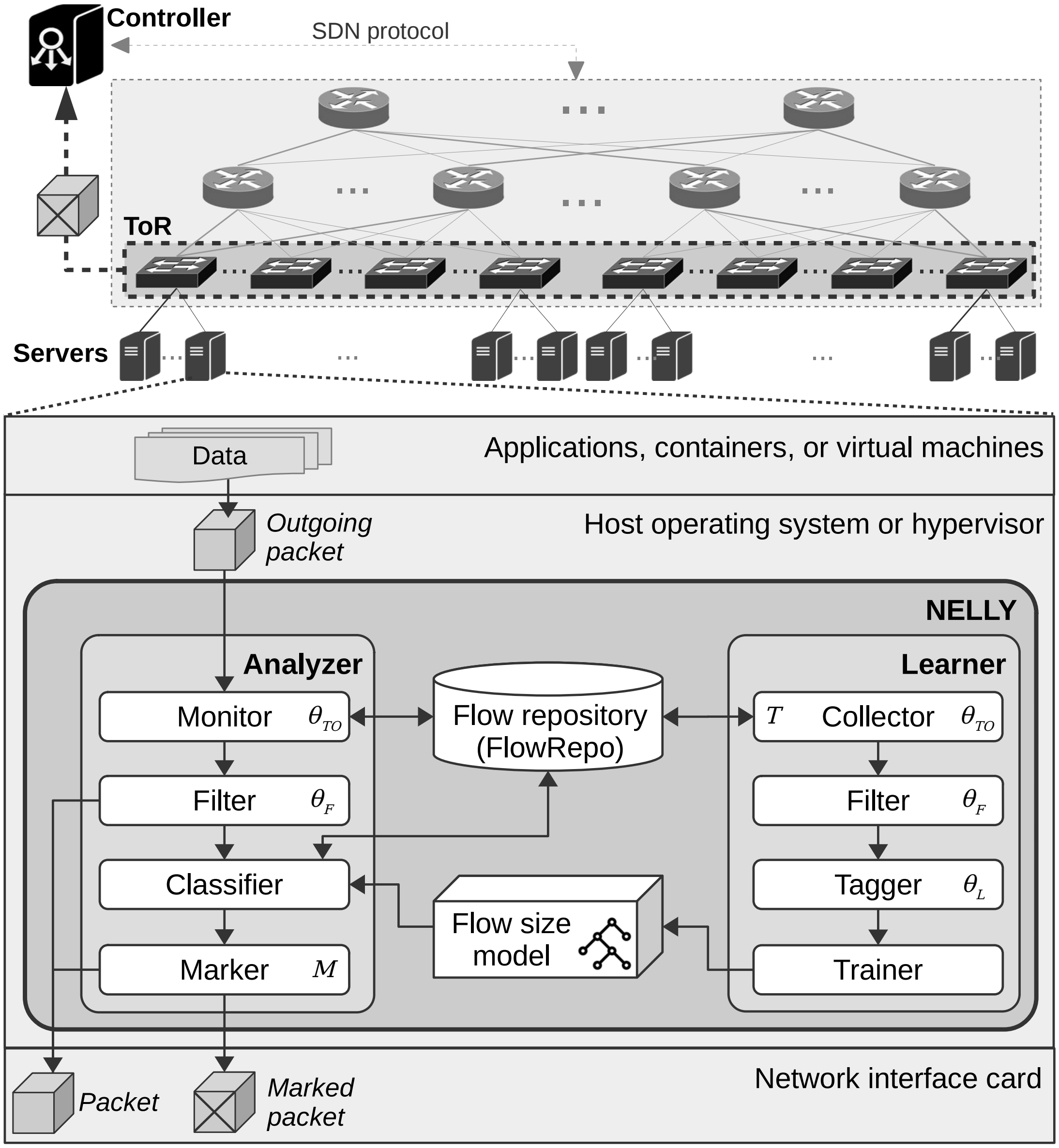


Figure 3.1: Architecture of NELLY

and *Learner*. The Analyzer applies a flow size classification model for detecting and marking elephant flows on the fly. The Learner then applies an incremental learning algorithm for building and updating the flow size classification model. This model maps online features (i.e., features extracted from the first few packets of a flow) onto the corresponding class of flows (i.e., mice or elephants). The processes of the Analyzer and the Learner run concurrently as depicted in Algorithms 3.1 and 3.2, respectively. Moreover, for the sake of readability, Table 3.1 lists and describes the symbols defined in the architecture of NELLY.

NELLY is conceived for recognizing and handling elephant flows in real SDN imple-

Table 3.1: Symbols in the architecture of NELLY

Symbol	Name	Description
θ_{TO}	Timeout threshold	Time limit above which both the Monitor and the Collector acknowledge a flow has terminated
θ_F	Filter threshold	Flow size limit below which either the Analyzer sends the packets without further processing or the Learner discards the flow records
M	Packet marking	Number of subsequent packets in an elephant flow to be marked
T	Collection rate	Time interval at which the Collector looks for terminated flows in FlowRepo
θ_L	Labeling threshold	Flow size limit above which the Tagger labels the flows as elephants

mentations. NELLY can run on any host operating system or hypervisor. In the control plane, any OpenFlow-compliant controller (e.g., OpenDaylight , ONOS, Ryu) can be used since NELLY operates at the server-side. In the data plane, OpenFlow-compliant switches (e.g., Open vSwitch) can be employed since NELLY requires only that the ToR switches include a pre-configured routing rule to forward elephant flows to the controller. The controller then can install specific routing rules per elephant flow based on the best path computed by a rescheduling algorithm, as discussed in Chapter 4.

3.1.1 Analyzer

As illustrated in Figure 3.1, the Analyzer consists of four modules: *Monitor*, *Filter*, *Classifier*, and *Marker*. The process of each module is detailed in Algorithm 3.1. As shown in lines 1–2, the Monitor keeps track of flows by extracting the header, size, and timestamp of each outgoing packet. A flow consists of subsequent packets sharing the same value for certain header fields, and separated by a time-space shorter than a threshold timeout (θ_{TO}). NELLY enables a flexible configuration of these flow parameters, namely, flow header fields and θ_{TO}. For example, the flow header fields can be set as the well-known 5-tuple: source IP, source port, destination IP, destination port, and IP protocol. These flow header fields can also include MAC addresses and VLAN ID. On the other hand, the configuration of θ_{TO} is discussed in Section 3.1.2.

The Analyzer manages a flow record in the Flow Repository (FlowRepo) for each observed flow. As illustrated in Figures 3.2 and 3.3, the flow record includes the Flow IDentifier (FlowID), start time, last-seen time, packet header (e.g., 5-tuple), flow size, the size and Inter-Arrival Time (IAT) of the first N packets, as well as the identified class

Algorithm 3.1: NELLY Analyzer

```
input  : outgoing packet $p$ with header $h_p$, size $s_p$, and timestamp $t_p$, and flow size classification
         model $m$
output: either packet $p$ or packet marked $p^*$
data   : timeout threshold $\theta_{TO}$, filtering flow size threshold $\theta_F$, and number of first packets $N$
begin on receiving $p$
    // Monitor
    get $h_p$, $s_p$, and $t_p$ from $p$;
    $fid \leftarrow$ compute FlowID using the flow header fields from $h_p$;
    if $fid \notin$ FlowRepo then
        $f \leftarrow$ call CREATE_FLOW($fid$, $h_p$, $s_p$, $t_p$)
    else
        $F \leftarrow$ fetch the last flow $f \in$ FlowRepo such that $f$.id = $fid$;
        if (currentTime $-$ $f$.lastSeenTime) $> \theta_{TO}$ then
            $f \leftarrow$ call CREATE_FLOW($fid$, $h_p$, $s_p$, $t_p$)
        else
            $f \leftarrow$ call UPDATE_FLOW($f$, $s_p$, $t_p$)
        end
    end
    // Filter
    if $f$.size $< \theta_F$ then return $p$;
    // Classifier
    if $\nexists$ $f$.class then
        $f$.class $\leftarrow$ $m$.CLASSIFY($f$);
        update $f \rightarrow$ FlowRepo;
    end
    // Marker
    if $f$.class $=$ “Elephant” then
        $p^* \leftarrow$ mark $p$;
        return $p^*$;
    end
    return $p$;
end
function CREATE_FLOW($fid$, $h_p$, $s_p$, $t_p$):
    $f \leftarrow$ initialize a new flow with FlowID $fid$;
    $f$.headerFields[] $\leftarrow$ array of flow header fields from $h_p$;
    $f$.startTime $\leftarrow$ $f$.lastSeenTime $\leftarrow$ $t_p$;
    $f$.size $\leftarrow$ $f$.sizePackets[0] $\leftarrow$ $s_p$;
    create $f \rightarrow$ FlowRepo;
    return $f$
end
function UPDATE_FLOW($f$, $s_p$, $t_p$):
    $n \leftarrow$ current number of packets of $f$;
    if $n \leq N$ then
        $f$.sizePackets[$n$] $\leftarrow$ $s_p$;
        $f$.iatPackets[$n$] $\leftarrow$ $t_p - -f$.lastSeenTime;
    end
    $f$.size $\leftarrow$ $f$.size $+$ $s_p$;
    $f$.lastSeenTime $\leftarrow$ $t_p$;
    update $f \rightarrow$ FlowRepo;
    return $f$
end
```

(i.e., mice or elephants). Note that the IAT of the first packet is not included because it does not provide distinctive flow information (i.e., the IAT is always zero for the first packet of every flow).

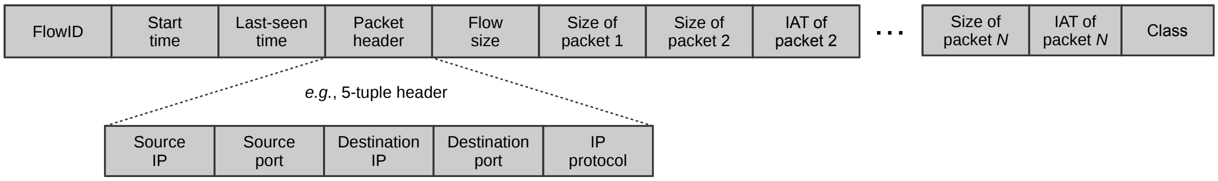
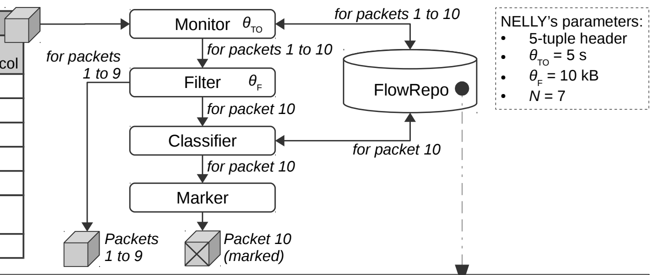


Figure 3.2: Structure of flow records in FlowRepo

As depicted in lines 3–13 in Algorithm 3.1, the Monitor then generates a FlowID from the flow header fields of each packet and checks to see if it exists in the FlowRepo. If this FlowID is missing (e.g., for packets 1 and 3 in Figure 3.3), or if the time since the last update of an existing record with this FlowID is longer than θ_{TO} (e.g., for packet 4), the Monitor creates a new record in the FlowRepo (Algorithm 3.1, lines 25–32). Otherwise, the Monitor fetches and updates the flow record (Algorithm 3.1, lines 33–43) using the FlowID stored in the FlowRepo (e.g., for packets 2 and 5 through 10 in Figure 3.3). When multiple flow records sharing the same FlowID exist in the FlowRepo, the Monitor always works with the most recent one (e.g., for packets 5 to 10).

Outgoing packets

Packet #	Time (ms)	Size (bytes)	Source IP	Source port	Destination IP	Destination port	IP protocol
1	100	1500	1.1.1.1	2000	2.2.2.2	20	6
2	900	1500	1.1.1.1	2000	2.2.2.2	20	6
3	2000	175	1.1.1.1	3000	3.3.3.3	80	6
4	6000	1500	1.1.1.1	2000	2.2.2.2	20	6
5	6010	1500	1.1.1.1	2000	2.2.2.2	20	6
6 to 9	6020 to 6050 (every 10ms)	1500	1.1.1.1	2000	2.2.2.2	20	6
10	6060	1500	1.1.1.1	2000	2.2.2.2	20	6



NELLY's parameters:
- 5-tuple header
- θ_{TO} = 5 s
- θ_F = 10 kB
- *N* = 7

Flow records in FlowRepo

For packet #	FlowID	Start time	Last-seen time	Source IP	Source port	Destination IP	Destination port	IP protocol	Flow size	Size of packet 1	Size of packet 2	IAT of packet 2	. . .	Size of packet 7	IAT of packet 7	Class
1	**1.1.1.1_2.2.2.2-2000_20-6**	**100**	**100**	**1.1.1.1**	**2000**	**2.2.2.2**	**20**	**6**	**1500**	**1500**	**-**	**-**	**···**	**-**	**-**	**-**
2	1.1.1.1_2.2.2.2-2000_20-6	100	**900**	1.1.1.1	2000	2.2.2.2	20	6	**3000**	1500	**1500**	**800**	···	-	-	-
3	1.1.1.1_2.2.2.2-2000_20-6	100	900	1.1.1.1	2000	2.2.2.2	20	6	3000	1500	1500	800	···	-	-	-
	1.1.1.1_3.3.3.3-3000_80-6	**2000**	**2000**	**1.1.1.1**	**3000**	**3.3.3.3**	**80**	**6**	**175**	**175**	**-**	**-**	**···**	**-**	**-**	**-**
4	1.1.1.1_2.2.2.2-2000_20-6	100	900	1.1.1.1	2000	2.2.2.2	20	6	3000	1500	1500	800	···	-	-	-
	1.1.1.1_3.3.3.3-3000_80-6	2000	2000	1.1.1.1	3000	3.3.3.3	80	6	175	175	-	-	···	-	-	-
	1.1.1.1_2.2.2.2-2000_20-6	**6000**	**6000**	**1.1.1.1**	**2000**	**2.2.2.2**	**20**	**6**	**1500**	**1500**	**-**	**-**	**···**	**-**	**-**	**-**
5	1.1.1.1_2.2.2.2-2000_20-6	100	900	1.1.1.1	2000	2.2.2.2	20	6	3000	1500	1500	800	···	-	-	-
	1.1.1.1_3.3.3.3-3000_80-6	2000	2000	1.1.1.1	3000	3.3.3.3	80	6	175	175	-	-	···	-	-	-
	1.1.1.1_2.2.2.2-2000_20-6	6000	**6010**	1.1.1.1	2000	2.2.2.2	20	6	**3000**	1500	**1500**	**10**	···	-	-	-
6 to 9	The Monitor updates the fields **last-seen time**, **flow size**, and the **size** and **IAT of packet N** of the last flow record with FlowID 1.1.1.1_2.2.2.2-2000_20-6 (i.e., third row)															
10	1.1.1.1_2.2.2.2-2000_20-6	100	900	1.1.1.1	2000	2.2.2.2	20	6	3000	1500	1500	800	···	-	-	-
	1.1.1.1_3.3.3.3-3000_80-6	2000	2000	1.1.1.1	3000	3.3.3.3	80	6	175	175	-	-	···	-	-	-
	1.1.1.1_2.2.2.2-2000_20-6	6000	**6060**	1.1.1.1	2000	2.2.2.2	20	6	**10500**	1500	1500	10	···	**1500**	**10**	**Elephant***

*In bold the values created or updated by the Monitor; * only the values of Class are updated by the Classifier.*
Time values are in milliseconds (ms) and size values are in bytes.

Figure 3.3: Example of how flow records are created and updated in FlowRepo

Using the updated flow record, the Filter (Algorithm 3.1, line 14) avoids the introduction of a delay in the classification of a large number of mouse flows (usually latency-sensitive [27, 29]) by sending the packets of flows with a size below a certain threshold (θ_F) directly to the SDDCN without further processing (e.g., for packets 1 to 9 in Figure 3.3). The Filter also ensures that the Classifier receives all the required online features for making the classification. The online features refer to flow data extracted from the first N packets of a flow. The Filter then guarantees the size and IAT of the first N packets of a flow since the maximum value of N depends on θ_F. For example, $\theta_F = 10$ kB would require an $N \leq 7$ over Ethernet, otherwise, data from some packets would be missed. Consequently, the Classifier operates once the Monitor has processed packets that increment the size of flows over θ_F (e.g., for packet 10).

The Classifier (Algorithm 3.1, lines 15–18) applies the flow size classification model to the online features to identify flows as either mice or elephants. This model results from an incremental learning algorithm, which maps the online features to the corresponding class of flows used as training data. After applying the flow size classification model, the Classifier stores the identified class in the FlowRepo for each flow record with a flow size greater than θ_F (e.g., elephant for flow of packet 10 in Figure 3.3). Therefore, when processing a packet of a previously identified flow, the Classifier checks the fetched class from the FlowRepo to avoid any delay from the classification. The Classifier then reports to the Marker the class of the flow for each packet. We discuss in Section 3.1.2 how the Learner collects the training data for building and updating the flow size classification model.

The Marker (Algorithm 3.1, lines 19–23) forwards the packets of flows classified as mice without changes but marks those classified as elephants (e.g., packet 10 in Figure 3.3). To mark a packet, the Marker sets a predefined value in a code point header field supported by SDN switches. For example, OpenFlow switches support matching in two code point header fields. The first of these is the 6-bit Differentiated Services Code Point (DSCP) field of the IPv4 header. This DSCP reserves a code point space for experimental and local usage (i.e., $****11$, where $*$ is 0 or 1). The second is the 3-bit 802.1Q Priority Code Point (PCP) field of the Ethernet header. In practice, NELLY can rely on either one of these fields, since it is improbable that a data center use both DSCP and PCP simultaneously [29].

The Marker can be extended by enabling a flexible configuration of the number of subsequent packets in an elephant flow to be marked (M), thus enabling a trade-off between reliability and latency. For instance, as M increases, the lesser the probability that the controller will miss elephant flows due to losses of marked packets in the SDDCN. However, a higher M introduces a delay in the Marker for a higher number of packets of elephant flows. Once the controller has installed a higher priority routing rule for handling a specific elephant flow across the SDDCN, the subsequent marked packets of this flow are not forwarded to the controller.

3.1.2 Learner

As depicted in Figure 3.1, the Learner consists of four modules: *Collector*, *Filter*, *Tagger*, and *Trainer*. The process of each module is detailed in Algorithm 3.2. As shown in lines 1–4, the Collector fetches terminated flows from the FlowRepo at every interval T. A flow is considered terminated if it remains idle for longer than θ_{TO}. Therefore, the Collector recognizes terminated flows by checking that a time longer than θ_{TO} has passed since the last-seen time of the FlowID records in the FlowRepo. Note that the Collector relies on the FlowID records updated by the Monitor for the recognition of the terminated flows, so their actual size can be obtained.

Algorithm 3.2: NELLY Learner

```
   input  : flow size classification model $m$
   output: either actual $m$ or updated $m$
   data   : learning time interval $T$, flow timeout threshold $\theta_{TO}$, filtering flow size threshold $\theta_F$, and
            labeling flow size threshold $\theta_L$
 1 begin every $T$
       // Collector
 2     $F \leftarrow$ fetch flows $f \in$ FlowRepo;
 3     for $f \in F$ do
 4         if currentTime $-$ $f$.lastSeenTime $> \theta_{TO}$ then
 5             delete $f \rightarrow$ FlowRepo;
               // Filter
 6             if $f$.size $\geq \theta_F$ then
                   // Tagger
 7                 if $f$.size $\geq \theta_L$ then  $f$.class $\leftarrow$ "Elephant" ;
 8                 else  $f$.class $\leftarrow$ "Mouse" ;
                   // Trainer
 9                 $m \leftarrow m$.UPDATE($f$.headerFields[], $f$.sizePackets[], $f$.iatPackets[], $f$.class);
10             end
11         end
12     end
13     return $m$;
14 end
```

The Collector avoids increasing memory consumption in NELLY by removing terminated flows from the FlowRepo (Algorithm 3.2, line 5). The actual size of terminated flows can also be further used to provide fixed-memory probability distributions that support autonomous configuration of flow size thresholds [257]. Memory requirements in the FlowRepo thus depend on both T and θ_{TO}. T provides a trade-off between memory and processing. As T decreases, the Collector removes the terminated flows from the FlowRepo more quickly, consuming less memory, but leading to more processing. In turn, θ_{TO} directly affects the number of FlowID records stored in memory. As θ_{TO} increases, the FlowRepo retains FlowID records for a longer time. θ_{TO} is related to the *inactive timeout* configuration of flow rules in SDN-enabled switches, which provides a trade-off between flow table occupancy and miss-rate (i.e., when the packet IAT is greater than the timeout) [261].

The Filter of the Learner (Algorithm 3.2, line 6) receives the terminated flows from the Collector and reports to the Tagger only those with size greater than θ_F. The terminated flows are then used by the Trainer to build the flow size classification model. Since the Classifier operates only with flows of a size greater than θ_F, the Filter of the Learner prevents the introduction of noise to the model.

The Tagger (Algorithm 3.2, lines 7–8) compares the actual size of the filtered flows to a labeling threshold (θ_L) so that they can be tagged as either mice or elephants. θ_L will vary (e.g., 100 kB or 1 MB) as a function of the traffic characteristics and performance requirements of SDDCNs. Labeled flows provide the Trainer (Algorithm 3.2, line 9) with the *ground truth* for building a supervised learning model for flow size classification (see Section 2.2). This classification model maps online features (i.e., packet header, size, and IAT of the first N packets) onto the corresponding class (i.e., mice or elephants). Recall that the Classifier relies on the flow size classification model to identify elephant flows.

Since flows represent continuous and dynamic data streams, the Trainer uses an incremental learning algorithm (e.g., Hoeffding tree and online ensembles) for building the flow size classification model. Incremental learning enables updating the flow size classification model as the Trainer receives labeled flows over time, rather than retraining from the beginning (see Section 2.2.1). Therefore, NELLY adapts to varying traffic characteristics and performs continuous learning with limited memory resources. There is no need for the Trainer to maintain labeled flows in memory. This is an important characteristic of NELLY, since it helps to reduce the consumption of resources in all the servers of the SDDCN.

3.2 Evaluation

This section presents the evaluation of NELLY in relation to classification accuracy and time by using real packet traces and incremental learning algorithms. The generic approach for designing ML-based solutions in networking (see Figure 2.4) is used to describe and conduct the evaluation of NELLY: *(i)* data collection to gather the raw packet traces and generate the flow size datasets, *(ii)* feature engineering to extract and format the online features of the flow size datasets, *(iii)* establishing the ground truth to label each flow in the datasets using the flow size and the classes of interest (i.e., mice and elephants), *iv* model validation to define the accuracy metrics that measure the performance of the trained models; and *(v)* model learning to train different models by using a variety of incremental learning algorithms.

3.2.1 Datasets

Two real packet traces, UNI1 and UNI2, captured in university data centers [255], were employed to evaluate NELLY (Table 3.2 summarizes their characteristics). These two traces are shorter than three hours long, but their mice and elephants distributions are similar to those found in non-public traces collected at different periods along the day [9, 11]. On the other hand, to the best of our knowledge, neither traces nor datasets of IPv6 traffic in DCNs are publicly available. In line with that, NELLY was evaluated using IPv4 traffic only which represents over 99% of the packets in UNI1 and UNI2.

Table 3.2: Details of real packet traces and extracted IPv4 flows

Packet traces [255]		**UNI1**		**UNI2**	
Duration		65 min		158 min	
Packets		19.85 M		100 M	
IPv4 % of total traffic		98.98% *(mostly TCP)*		99.9% *(mostly UDP)*	
IPv4 flows		1.02 M *(TCP and UDP)*		1.04 M *(mostly UDP)*	
Details of IPv4 flows	**Flow size**	**% of IPv4 flows**	**% of IPv4 traffic**	**% of IPv4 flows**	**% of IPv4 traffic**
	$\geq$ **10 kB**	7.16%	95.06%	5.91%	98.81%
	$\geq$ **100 kB**	0.83%	83.71%	1.93%	96.86%
	$\geq$ **500 kB**	0.14%	73.14%	0.76%	93.52%
	$\geq$ **1 MB**	0.07%	69.52%	0.48%	90.83%
	$\geq$ **5 MB**	0.01%	60.33%	0.17%	81.34%

IPv4 flows obtained using the 5-tuple header and a threshold timeout $\theta_{TO} = 5$ s

Only the following parameters needed to be defined to generate the datasets: the flow header fields, θ_{TO}, and N. Firstly, the 5-tuple header (i.e., source IP, destination IP, source port, destination port, and IP protocol) as the flow header fields since it sufficiently characterizes IPv4 flows; hereinafter, they are referred just as flows. Secondly, $\theta_{TO} = 5$ s was established on the basis of the break-even point analysis between the flow table occupancy and the miss-rate in OpenFlow switches for DCNs considered by [261]. Then, since the maximum value of N depends on θ_F, $N = 7$ was set as the maximum for $\theta_F = 10$ kB. As shown in Table 3.2, the selected θ_F encompasses all the potential elephants (i.e., flows carrying more than 95% of the traffic) and avoids the introduction of the classification delay to mice (for more than 93% of the flows). Using these parameters, the UNI1 and UNI2 data traces were processed to generate the corresponding flow size datasets, each containing somewhat more than a million flows (see Table 3.2). Since NELLY only classifies flows greater than θ_F, those smaller than $\theta_F = 10$ kB were removed from both datasets. Therefore, the UNI1 and UNI2 datasets consisted of approximately 70,000 and 60,000 flows, respectively.

The datasets [262] included the following flow information: start time, end time, 5-tuple header, size and IAT of the first 7 packets, as well as flow size. The start and end times enabled a more realistic evaluation (see Section 3.2.3). The 5-tuple header and the size and IAT of the first 7 packets represented the online features for the flow size classification model. The flow size was compared to different θ_L (e.g., 100 kB, 500 kB, 1 MB, and 5 MB) to label the flows as mice or elephants (i.e., classes of interest). Unless otherwise stated, the datasets with $\theta_L =$ 100 kB were used. Labeled flows represented the ground truth for learning and validating the flow size classification model.

For complementing feature engineering, various different data types were considered for the online features, particularly for the 5-tuple header. Certainly, the size and IAT of the first 7 packets (13 features, since the IAT of the first packet is not included) indicate a measurement, hence, numeric data, whereas the 5-tuple header contains two IP addresses in dotted-decimal notation (i.e., categorical data) and three numeric codes (i.e., nominal data). However, the huge set of possible categories for IP addresses (i.e., 2^{32}) hinders a real implementation. To address this problem, the IP addresses were divided into four octets, resulting in a total of 11 nominal features for the 5-tuple header. To handle these 11 nominal features as numeric data, a *Numeric* (Num) header type was defined. These features were then transformed into binary digits (bits), generating 104 features for the 5-tuple header. Considering these binary features, two more header types were defined: *Binary-Numeric* (BinNum) to treat binary features as numeric data (i.e., a value between zero and one) and *Binary-Nominal* (BinNom) to handle binary features as nominal data (i.e., zero or one). Table 3.3 illustrates the features included by each header type. Unless otherwise stated, the datasets with BinNom-header were used.

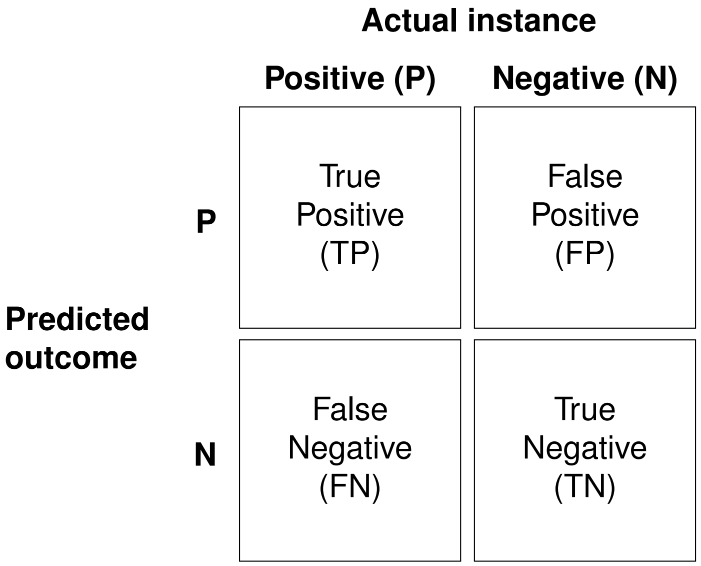
Table 3.3: Header types defined during feature engineering

Header type	**Features of the header type** *(example)*						**# of features**
5-tuple	Source/Destination IP				Source/Destination Port	IP Protocol	5
	41.177.26.55 *244.3.160.248*				*80* *43521*	*6*	
Num	Source/Destination IP				Source/Destination Port	IP Protocol	11
	Octet 1	Octet 2	Octet 3	Octet 4			
	41 *244*	*177* *3*	*26* *160*	*55* *248*	*80* *43521*	*6*	
BinNum **BinNom**	Source/Destination IP				Source/Destination Port	IP Protocol	104*
	8 bits	8 bits	8 bits	8 bits	16 bits	8 bits	
	00101001 *11110100*	*10110001* *00000011*	*00011010* *10100000*	*00110111* *11111000*	*0000000001010000* *1010101000000001*	*00000110*	

** Each bit represents one feature*

3.2.2 Accuracy metrics

Metrics derived from the confusion matrix (see Figure 3.4) were used, including the True Positive Rate (TPR) and the False Positive Rate (FPR), thus avoiding the over-optimism of the conventional accuracy metric caused by an imbalance of classes [263]. In the datasets, the imbalance between mice and elephants depends on θ_L. For example, assuming $\theta_L = 100$ kB, only 12% of flows above 10 kB in the UNI1 dataset represent the elephant class (see Table 3.2). The imbalance grows as θ_L increases.



		Actual instance	
		Positive (P)	**Negative (N)**
Predicted outcome	**P**	True Positive (TP)	False Positive (FP)
	N	False Negative (FN)	True Negative (TN)

Figure 3.4: Confusion matrix for binary classification

Consider that each row in the confusion matrix, illustrated in Figure 3.4, represents a predicted outcome and each column represents the actual instance. In this manner, True Positive (TP) is the intersection between correctly predicted outcomes for the actual positive instances. Similarly, True Negative (TN) is when the classification model correctly predicts an actual negative instance. Whereas, False Positive (FP) and False Negative (FN) describe incorrect predictions for negative and positive actual instances, respectively. Note, that TP and TN correspond to the true predictions for the positive and negative classes, respectively.

Recall that flows classified as elephants are forwarded to the controller for further processing, thus introducing transmission and processing delays. Therefore, NELLY aims at detecting as many elephants while negatively affecting as few latency-sensitive mice as possible. Considering elephants as the positive condition, the TPR describes the proportion of detected elephants (see Equation 3.1) whereas the FPR provides the ratio of negatively affected mice (see Equation 3.2). Both TPR and FPR range between 0 and 1. Furthermore, the Matthews Correlation Coefficient (MCC) was used to analyze the balance between the TPR and the FPR. The MCC takes all values from the confusion matrix to provide a measure between 1 and -1 (see Equation 3.3). As the MCC gets closer to 1, the difference of the TPR over the FPR increases, leading to a more accurate classifier. An MCC between 0 and -1 means that TPR $\leq$ FPR, which would be less accurate than a random classifier. In our experiment, the MCC values were always greater than 0, hence, we use a range between 0 and 1 to plot TPR, FPR, and MCC in Figures 3.5, 3.6, and 3.7.

$$TPR = \frac{TP}{TP + FN} \tag{3.1}$$

$$FPR = \frac{FP}{FP + TN} \tag{3.2}$$

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.3}$$

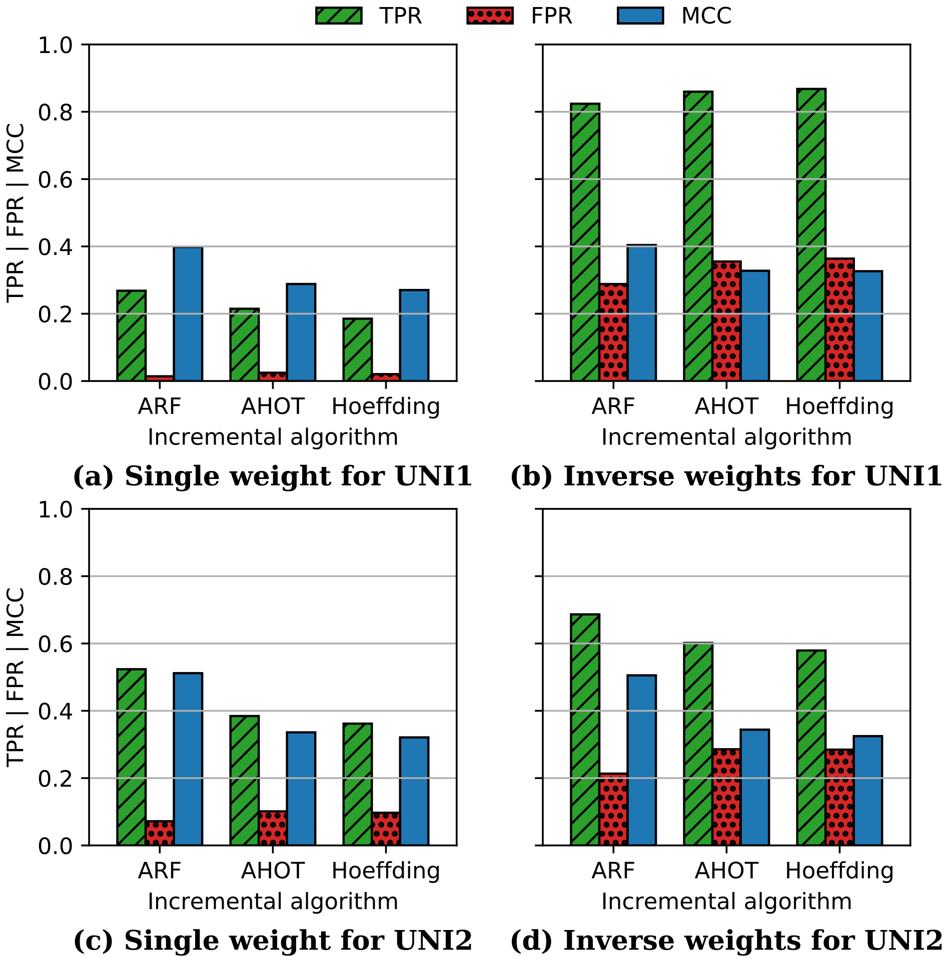
The MCC metric is employed in the performance analysis because it is recommended for imbalanced datasets (like UNI1 and UNI2) [264]. The MCC score is only high when the classification algorithms are doing well in both the positive and negative elements (i.e., elephants and mice, respectively). The ROC curve has also proven to be useful for imbalanced datasets but it is more appropriate to analyze classification algorithms that output a real value [265]. Thus, we preferred the MCC because the output of the incremental learning classification algorithms employed in this paper is a single class value (either mouse or elephant) rather than a real value.

3.2.3 Experiment setup

Incremental learning algorithms are commonly evaluated using the interleaved test-then-train approach [266]. This approach refers to going through each flow to classify it first by working only with the online features and then use its actual class for training the flow size classification model. However, since flows start and end over time, some order of the flows must be established. Moreover, under real conditions, some flows start before a classified flow ends, whereas others end before a new flow starts. Therefore, the flows are classified at the start time and the model is trained at the end time, so the performance evaluation will be based on more realistic conditions.

The imbalance of classes in the UNI1 and UNI2 datasets was addressed by training the flow size classification model using *inverse weights*, as in [32], i.e., weights (between 0 and 1) inversely proportional to the ratio of training instances previously encountered by the model for each class. If the model is trained with a *single weight* (i.e., 1 by default in the Massive Online Analysis (MOA) tool [266]), it would tend to classify all flows as mice due to the imbalance of classes.

To corroborate the weighting decision, the accuracy was evaluated when training the model with the single weight and inverse weights for three incremental learning algorithms available in MOA, namely, Adaptive Random Forest (ARF), Adaptive Hoeffding Option Tree (AHOT), and Hoeffding trees (a.k.a. CVFDT)—as later discussed in Section 3.2.4, NELLY achieves the best performance by using these algorithms. Figure 3.5 shows that the three algorithms achieve a higher elephant detection rate (i.e., TPR) for both datasets with the inverse weights than with the single weight. These gains in the TPR come at a sacrifice to the FPR; however, the three algorithms maintain a similar MCC. Therefore, in the performance evaluation, inverse weights were used to improve the TPR while maintaining the trade-off between the TPR and FPR.



(a) Single weight for UNI1 **(b) Inverse weights for UNI1**

(c) Single weight for UNI2 **(d) Inverse weights for UNI2**

Figure 3.5: Classification accuracy of NELLY when using a single weight and inverse weights for the UNI1 and UNI2 datasets

3.2.4 Performance analysis

To determine the consideration for the best classification performance of NELLY, the UNI1 and UNI2 datasets were used with different header types (i.e., Num, BinNum, and BinNom), as well as 50 incremental learning classification algorithms available in MOA. The performance evaluation included the accuracy metrics (i.e., TPR, FPR, and MCC) and the classification time per flow (T_C). The algorithms were executed with their default settings (except for the training weights) and without previous model initialization.

For the sake of brevity, Table 3.4 presents ten algorithms, namely, AHOT, ARF, Hoeffding tree, k-Nearest Neighbors (kNN) with Probabilistic Adaptive Windowing (PAW), Naïve Bayes (NB), Online Accuracy Updated Ensemble (OAUE), online bagging, online boosting, Stochastic Gradient Descent (SGD) for Support Vector Machines (SVM), and Very Fast Decision Rules (VFDR) with NB classifiers (VFDR$_{\text{NB}}$). These algorithms were selected on the basis of the best performance results between algorithms with a similar learning approach. Furthermore, Table 3.4 includes only the best results of each algorithm, taking into account both accuracy and classification time for a specific header type. The BinNom headers were found to enable the best performance of the majority of the algorithms for the UNI1 and UNI2 datasets. This was due to the fact that most algorithms achieved greater accuracy using the BinNom headers than the Num headers for a comparable classification time. The use of the BinNum headers is strongly discouraged; although similar or slightly better accuracy results were obtained, there was a significant increase in the classification time (up to 4x).

Table 3.4: Classification performance of NELLY with different incremental algorithms

Algorithms	UNI1					UNI2				
	TPR (%)	**FPR (%)**	**MCC**	**T_C (μs)**	**Header type**	**TPR (%)**	**FPR (%)**	**MCC**	**T_C (μs)**	**Header type**
AHOT	**85.97**	35.52	**0.327**	**4.07**	BinNom	**60.16**	28.58	**0.304**	**10.17**	BinNom
ARF	**82.39**	28.82	**0.359**	**12.01**	BinNom	**68.65**	21.33	**0.460**	**17.39**	BinNom
Hoeffding tree	**86.79**	36.38	**0.326**	**3.18**	BinNom	57.92	28.46	0.284	**4.64**	BinNom
kNN-PAW	25.30	2.99	0.311	473.1	Num	40.29	10.22	0.302	454.1	Num
NB	74.76	35.69	0.254	**4.76**	BinNom	49.74	23.18	0.267	**4.82**	BinNom
OAUE	**86.79**	33.63	**0.347**	25.58	BinNom	**63.28**	28.65	**0.332**	33.06	BinNom
Online bagging	**87.78**	37.11	**0.327**	23.98	BinNom	**64.17**	31.13	**0.314**	36.61	BinNom
Online boosting	75.88	29.93	0.307	**11.56**	BinNom	**64.62**	32.22	**0.307**	**16.82**	BinNom
SGD-SVM	16.76	10.21	0.067	**0.81**	Num	38.69	30.99	0.076	**0.8**	Num
VFDR$_{\text{NB}}$	74.44	33.77	0.267	18.47	BinNom	54.83	29.15	0.248	18.59	BinNom

In bold the top five results of TPR and MCC, and the T_C results shorter than 17.5 μs, for both UNI1 and UNI2

The accuracy results show that no single algorithm achieves the best values of the TPR and MCC for the UNI1 and UNI2 datasets. This is due to the fact that the flow size distribution and the features of the elephant and mouse flows were specific for each dataset. Therefore, the top five results were used to analyze the accuracy performance. Regarding the T_C, most algorithms introduced a classification delay per flow shorter than 17.5 μs, but this represents only a small percentage (7%) of the Round-Trip Time (RTT) in DCNs (i.e., 250 μs in the absence of queuing [267]).

Both Hoeffding tree and NB represent the state-of-the-art in incremental learning algorithms. Their simplicity and low computational cost enabled a very short delay ($T_C < 5$ μs) that accounts for only 2% of the RTT in DCNs. However, only the Hoeffding tree represents a valid alternative for the traffic similar to that of UNI1 because its TPR and MCC were among the top five results for the UNI1 dataset. The Hoeffding tree in MOA uses NB classifiers on the leaves (i.e., CVFDT$_{\text{NBC}}$), which improves the accuracy without compromising the computational cost.

The ARF, OAUE, online bagging, and online boosting are ensemble-based algorithms that combine multiple Hoeffding trees (ten in our evaluation) for improving the accuracy at the expense of increasing the computational cost. As described in Section 2.2.2, ensemble learning aims at improving the accuracy performance of a single learning model by combining it with other models. As a result, the poor performance of a single classifier can be compensated with ensemble learning at the price of extra computation. Indeed the results from ensemble learning must be aggregated, and a variety of techniques have been proposed in this matter, including random forest, block-weighting, bagging, and boosting.

Therefore, the ARF and online boosting algorithms introduced a T_C shorter than 7% of the RTT in DCNs. ARF provided the best MCC and a TPR among the top five for the UNI1 and UNI2 datasets. Online boosting can be seen as an option for the traffic similar to that of UNI2 since it was in the top five accuracy results only for the UNI2 dataset. In contrast, although the OAUE and online bagging algorithms also provided good accuracy results (particularly for the UNI1 dataset), they introduced a T_C twice longer than the T_C of ARF and online boosting. This long T_C is because OAUE and online bagging rely on ensemble methods (block-weighting and bagging, respectively) that demand more computation than those used by ARF and online boosting (random forest and boosting, respectively).

Similar to ARF, the AHOT algorithm figured in the top five accuracy results for both datasets. Moreover, AHOT only introduced a T_C shorter than 2% (5 μs) and 4% (10 μs) of the RTT in DCNs for the UNI1 and UNI2 datasets, respectively. AHOT is capable of improving the accuracy of the Hoeffding tree algorithm without demanding too much computation by providing an intermediate solution between a single Hoeffding tree and an ensemble of Hoeffding trees. AHOT uses additional option paths (five maximum in our evaluation) to build a single structure that efficiently represents multiple Hoeffding trees.

The implementations in MOA of the VFDR$_{\text{NB}}$, SGD-SVM, and kNN-PAW algorithms are strongly discouraged. The VFDR$_{\text{NB}}$ presented accuracy results outside the top five for both datasets and a T_C slightly longer than 7% of the RTT in DCNs. This is because rule-based algorithms focus on building more interpretable models than does the Hoeffding tree algorithm, which increases the computational cost but not necessarily improves the accuracy. The SGD-SVM algorithm introduced the shortest classification delay ($T_C < 1$ μs) but it produced the worst values in the TPR and MCC metrics. The reason for these values is that MOA implements a very simple SGD-SVM algorithm that uses a linear kernel which is not sufficient to model different patterns in flows of packets. The kNN-PAW provided the second-worst TPR for both datasets and a very long classification delay ($T_C >$ 450 μs), which increased up to 3,000 μs with the BinNum and BinNom headers (i.e., 12x the RTT in DCNs). This long T_C value is a consequence of the computation of a distance metric by the algorithms based on kNN every time the classification is performed.

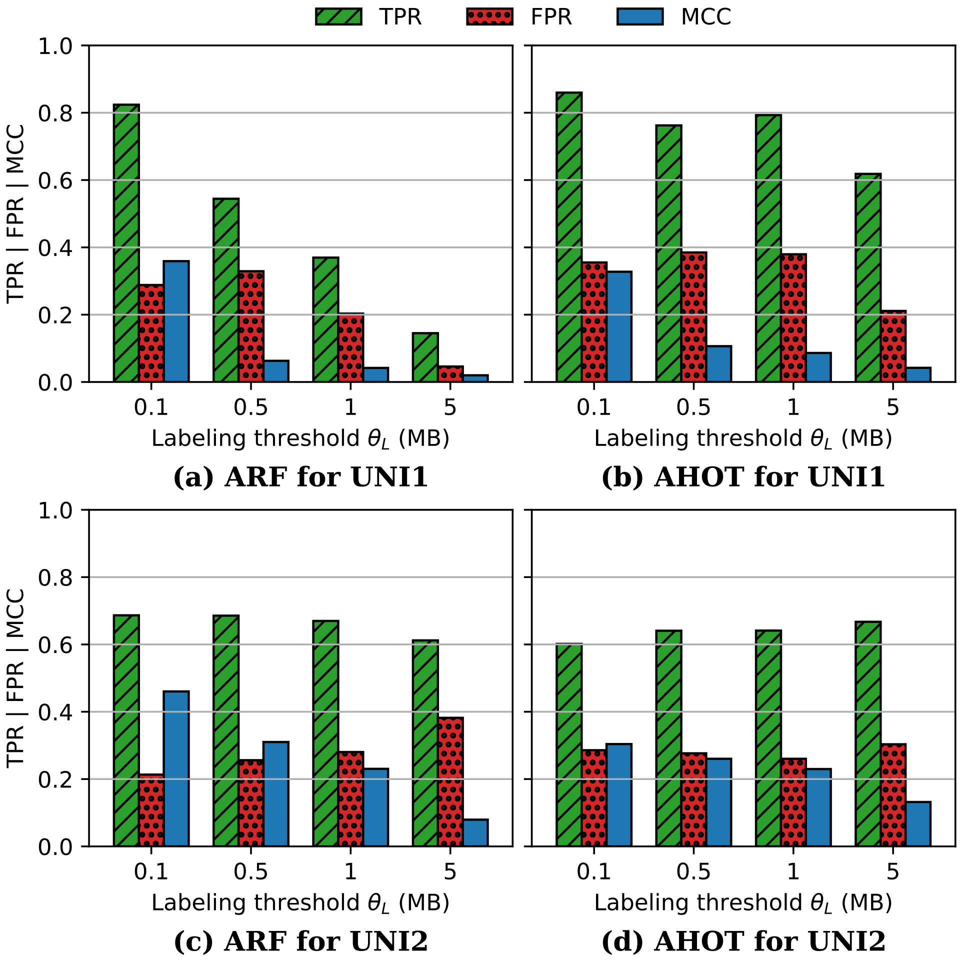
In conclusion, NELLY achieves the best classification performance by using the BinNom headers along with the following incremental learning algorithms:

- The ARF is good for any type of traffic and if the RTT is flexible. It achieved the best MCC for the UNI1 and UNI2 datasets, and it was also the fifth- and second-best for the TPR while introduced a T_C lesser than 7.5% of the RTT in DCNs.

- The AHOT is good for any type of traffic and a strict RTT. The TPR and MCC ranked among the top five for both datasets while the T_C was shorter than that of the ARF, especially for the UNI1 dataset.

- The Hoeffding tree (CVFDT$_{\text{NBC}}$) is good for traffic similar to that of UNI1 and if the RTT is very strict. The TPR was the second-best and the MCC was the fifth (quite close to the AHOT) for the UNI1 dataset while introduced a very short T_C. When the RTT constraint takes precedence over the accuracy, this would be a good option for traffic similar to that of UNI2 because a very short T_C was maintained while provided the sixth-best TPR and MCC for such traffic.

The classification accuracy of NELLY was also evaluated with the ARF and AHOT algorithms for different values of θ_L since this threshold may vary as a function of traffic and routing requirements. Both ARF and AHOT ranked among the top five in accuracy for both datasets with a T_C shorter than 7.5% of the RTT in DCNs. As shown in Figure 3.6, the MCC results of both algorithms were degraded as θ_L increased, especially for the UNI1 dataset. This is because the difference between the features of the elephants and mice becomes less significant as θ_L increases. In contrast, the TPR remained very similar as θ_L increased, except that the ARF suffered from a significant reduction in the TPR for the UNI1 dataset. Therefore, the AHOT was more robust to variations in θ_L for traffic similar to that of UNI1, although the performance of both algorithms was similar for the UNI2 dataset. Based on this summary, NELLY with the AHOT algorithm enables a flexible configuration of θ_L while providing great elephant flows detection in data centers regardless the type of traffic. For traffic similar to that of UNI2, both ARF and AHOT represent valid alternatives for the use of NELLY and the flexible configuration of θ_L is possible since they perform similarly in terms of elephants detection.



(a) ARF for UNI1

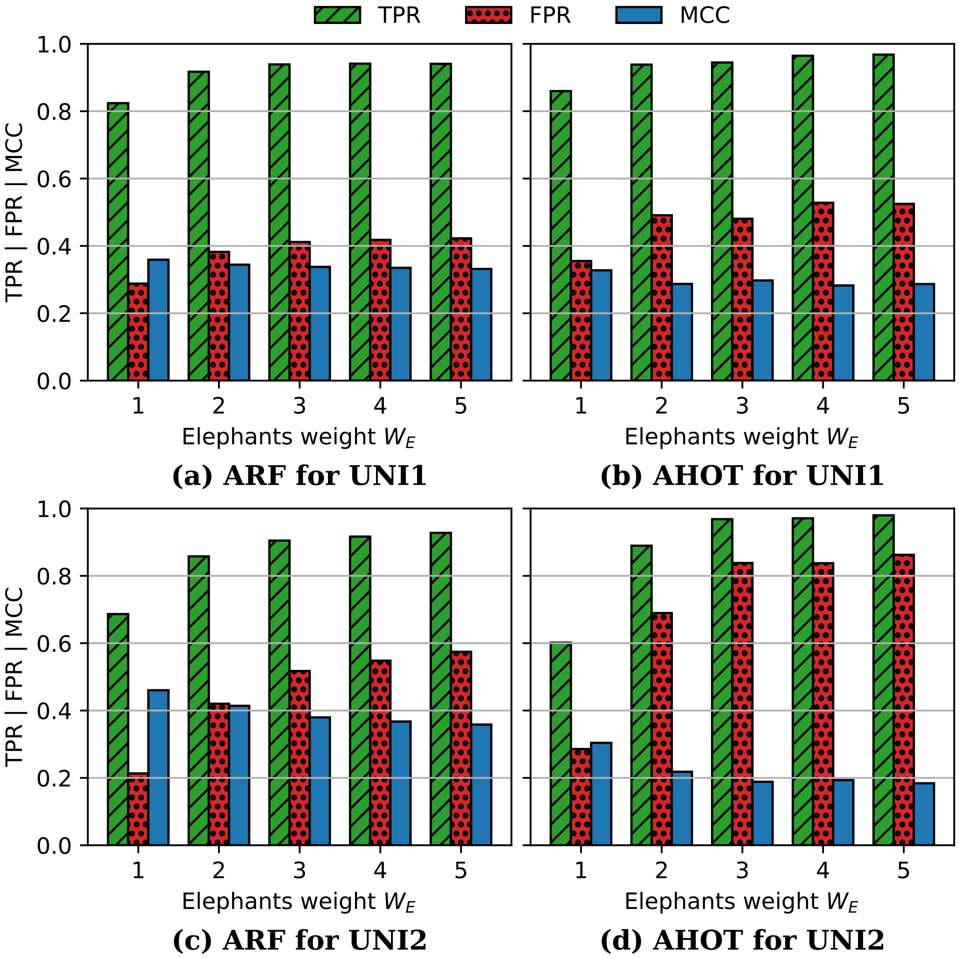
(b) AHOT for UNI1

(c) ARF for UNI2

(d) AHOT for UNI2

Figure 3.6: Classification accuracy of NELLY with the ARF and AHOT algorithms when varying the labeling threshold (θ_L) for the UNI1 and UNI2 datasets

Finally, the effect of the handling of different ranges of the inverse weights in the two classes on the classification accuracy of NELLY with the two algorithms (ARF and AHOT) was analyzed. The weights of the mice were maintained between 0 and 1, whereas the weights of the elephants ranged from 0 to W_E, where W_E varied from 1 to 5. Figure 3.7 shows that both the ARF and AHOT algorithms achieved a higher TPR for both datasets as W_E increased (up to 94% and 98% of elephants detection, respectively). These results were expected since establishing greater weights for the elephant class makes the learning algorithms increment the influence of the features of the elephant flows in the classification model. Moreover, the trade-off between the TPR and FPR (i.e., MCC) remained quite similar for UNI1-type traffic whereas that of UNI2 was degraded as W_E increased. This is due to the greater differences between the elephants and mice for the UNI1 dataset than for UNI2 when $\theta_L = 100$ kB. Therefore, as W_E increased for the UNI2 dataset, the increment of mouse flows wrongly classified as elephants (i.e., FPR) was greater than that of elephant flows correctly classified (i.e., TPR). In conclusion, NELLY supports a flexible configuration of inverse weights for meeting different accuracy requirements.



(a) ARF for UNI1

(b) AHOT for UNI1

(c) ARF for UNI2

(d) AHOT for UNI2

Figure 3.7: Classification accuracy of NELLY with the ARF and AHOT algorithms when varying the inverse weights of elephant flows (W_E) for the UNI1 and UNI2 datasets

3.3 Comparative analysis

NELLY was compared with OFSP [32], ESCA [35], FlowSeer [33], and Mahout [29]. OFSP, ESCA, and FlowSeer incorporate ML at the controller-side of SDDCNs for proactively detecting elephant flows, whereas Mahout performs reactive detection at the server-side. The results reported by each work for the UNI1 dataset were used to compare them in relation to: learning approach, elephants detection, false elephants, table occupancy, control overhead, detection time, network modifications, and performance factors. The seminal works involving Hedera [26] and DevoFlow [27] were not considered. These approaches perform reactive flow detection and their limitations hinder real implementation. Hedera causes large control traffic overhead and has poor scalability, whereas Devoflow requires custom-made switch hardware and imposes a heavy burden on switches. Devoflow also presents an alternative based on sFlow [250], however, ESCA revealed and outperformed the inaccurate and late elephant detection suffered by the sFlow-based DevoFlow.

Table 3.5 summarizes the comparative analysis. Overall, NELLY achieved a better balance between the comparative features than the other approaches, namely, a very high elephant detection rate with a very short detection time, while significantly reducing traffic overhead, without demanding switch table occupancy, and only software modifications and resources in servers were required. The following paragraphs outline the comparison of NELLY with the other approaches.

Table 3.5: Comparison of NELLY with related approaches

FEATURE	OFSP [32]	ESCA [35]	FlowSeer [33]	Mahout [29]	NELLY
Learning approach	Incremental	Batch	Batch and incremental	None	Incremental
Elephants detection	Very high	High	Very high	Perfect	Very high
False elephants	Very low	Very low	High	None	Very low
Table occupancy	Very high	Medium	Low	None	None
Control overhead	High	Medium	Medium	Very low	Very low
Detection time	Very short	Medium	Medium	Long	Very short
Network modifications*	None	Hardware of switches	None	Software in servers	Software in servers
Performance factors	Controller and ToR	Controller and ToR	Controller and ToR	Servers	Servers

* *Assuming OpenFlow-based networks [48]*

Learning approach. ML algorithms used for detecting elephant flows can involve batch or incremental learning. Batch learning refers to the use of training models based on static datasets (i.e., all training data are simultaneously available). However, batch learning requires the storage of unprocessed data to cope with traffic variations in DCNs, so the models must repeatedly work from scratch. This is time-consuming and prone to outdated models. Conversely, incremental learning continuously adapts the ML models on the basis of streams of training data, enabling constantly updated models and reducing time and memory requirements (see Section 2.2.1). ESCA relies on batch learning whereas NELLY and OFSP rely on incremental learning for detecting elephant flows. FlowSeer is a mixed approach using batch learning for the identification of potential elephants and incremental learning for the classification of the potential ones. Mahout has no learning approach, since it performs reactive elephants detection.

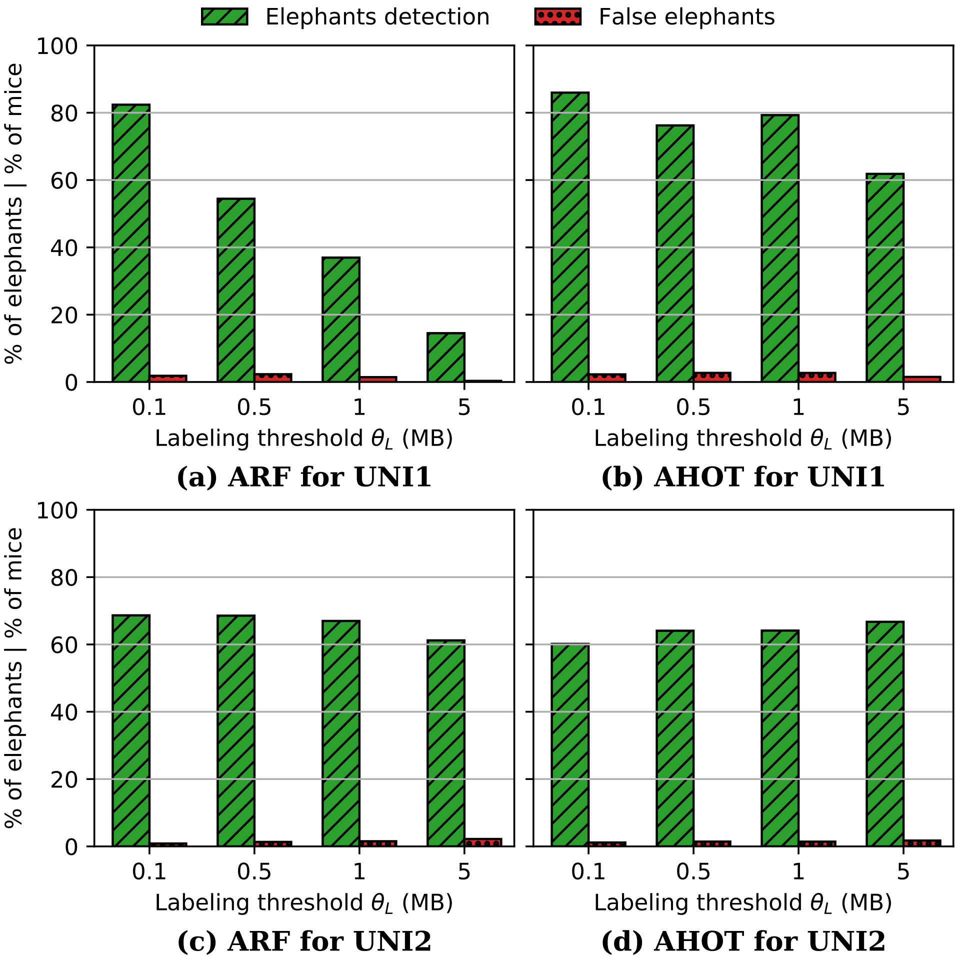
Elephants detection. The main goal of flow detection methods is to identify elephant flows (i.e., TPR). NELLY, OFSP, and FlowSeer all proactively detected more than 95% of elephant flows, whereas ESCA detected a maximum of 88.3%. Mahout provides perfect detection, although this is reactive.

False elephants. Mouse flows mistakenly identified as elephants (i.e., FPR) are needlessly forwarded to and processed by the controller. For achieving the highest elephants detection rate, FlowSeer informed the controller of 29% of mice as potential elephants, whereas OFSP and ESCA only reported around 2%. No mouse flow is reported to the controller by Mahout since detection is reactive. NELLY yielded an FPR of 40%, but this was computed using only 7% of the flows (i.e., $\theta_F = 10$ kB, in Section 3.2.1). NELLY sent the other 93% of the flows (corresponding to mice) directly to the SDDCN without further processing (see Section 3.1.1). NELLY thus forwards only 2.5% of mice to the controller for achieving the highest elephants detection rate (see Figure 3.9(b)).

Figures 3.8 and 3.9 depict the elephants detection and false elephants results that NELLY achieves when considering all the IPv4 flows from the UNI1 and UNI2 data traces, including the portion of mice sent directly to the SDDCN without further processing. Figure 3.8 shows the results for different values of θ_L (cf. Figure 3.6), while Figure 3.9 sets $\theta_L = 100$ kB and varies the range of the inverse weights of elephant flows (W_E) (cf. Figure 3.7).

Table occupancy. Controller-side flow detection methods install flow table entries in ToR switches for centrally collecting flow data. The smaller the number of flow table entries, the more efficient is the resource utilization. OFSP requires one entry per flow, thus constraining its scalability because of the limited memory in SDN switches. ESCA and FlowSeer install wildcard entries for sampling packets of flows. They reported 236 and 50 flow table entries, respectively, for achieving their highest detection rate in the UNI1 dataset. Conversely, NELLY and Mahout do not require flow table entries for collecting data since they operate at the server-side.

Control overhead. Flow detection methods require ToR switches to send control packets to the controller, either for the collection of flow data or for the reporting of detected elephant flows. The smaller the control overhead, the lower are the link utilization and the impact on the controller performance (since it has to process fewer control packets). The overhead of this control was computed by assuming no loss in the network and a control packet of 64 bytes. OFSP collects information from the first three packets of each flow, generating a control overhead of 402 kbps. FlowSeer collects information from the first five packets of sampled flows (i.e., 30% of the flow data) and potential elephants, yielding a control overhead of 288 kbps. ESCA reduces the



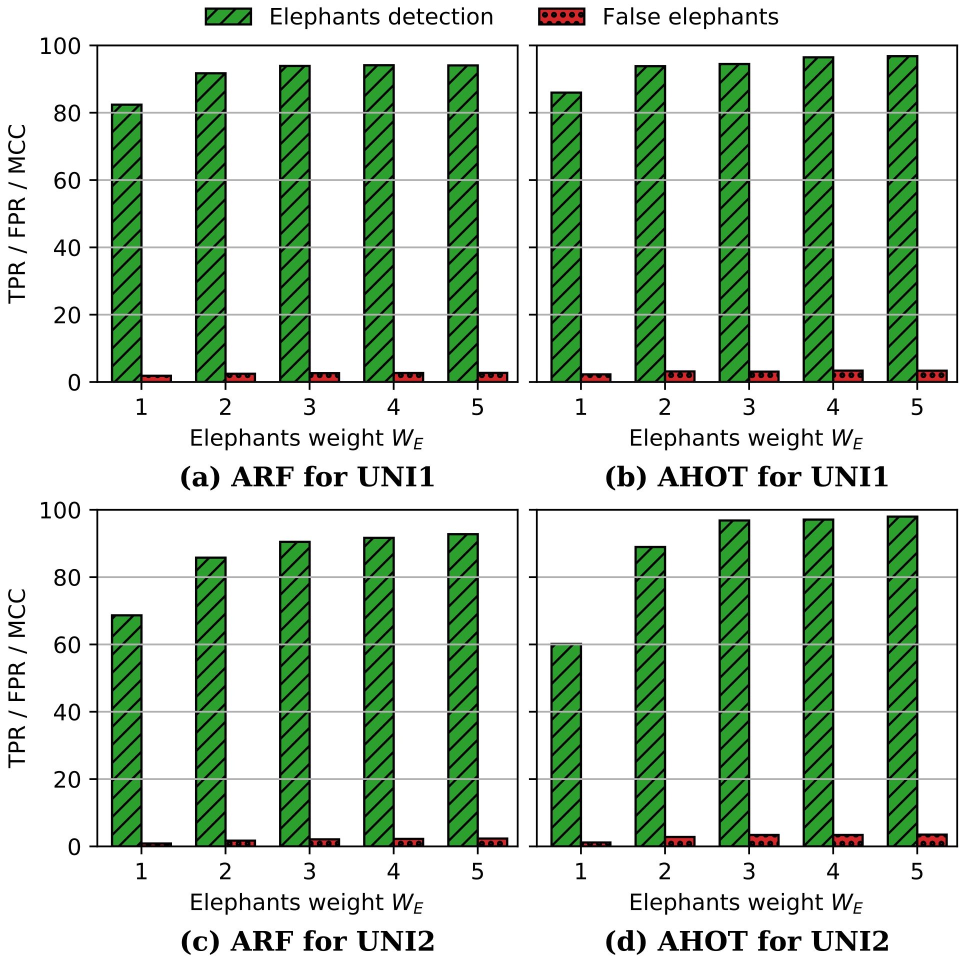
(a) ARF for UNI1 **(b) AHOT for UNI1**

(c) ARF for UNI2 **(d) AHOT for UNI2**

Figure 3.8: Accuracy of NELLY with the ARF and AHOT algorithms when varying the labeling threshold (θ_L) for all the IPv4 flows in UNI1 and UNI2 data traces

control overhead to 215 kbps by using a sampling method that only reports information from the first packet. In contrast, NELLY and Mahout merely require that ToR switches send information of flows marked as elephants, greatly reducing the control message overhead to 4.4 kbps and 1.1 kbps, respectively.

Detection time. Timely detection of elephant flows enables the controller to make early decisions to improve routing. OFSP, ESCA, FlowSeer, and NELLY enable a short detection time by proactively detecting elephant flows. ESCA reported a detection time of 1.98 s for achieving the highest detection rate. OFSP and NELLY detect elephants in a shorter time since they rely on the first N packets. On average, the detection time was 0.5 s for OFSP ($N = 3$) and 0.8 s for NELLY ($N = 7$). Further experimentation is needed to evaluate the detection time of FlowSeer. Nevertheless, the detection time of the latter would be slightly greater than for ESCA, since it is also based on sampling and considers the first five packets (ESCA considers only one packet). In contrast, Mahout



(a) ARF for UNI1 **(b) AHOT for UNI1**

(c) ARF for UNI2 **(d) AHOT for UNI2**

Figure 3.9: Accuracy of NELLY with the ARF and AHOT algorithms when varying the inverse weights of elephant flows (W_E) for all the IPv4 flows in UNI1 and UNI2 data traces

relies on a reactive mechanism that detects elephant flows after their corresponding socket buffer in a server surpasses a certain threshold. Assuming a small threshold of 100 kB, the average detection time of Mahout is 3.8 s. However, unlike ML-based flow detection methods, the detection time of Mahout becomes longer as the threshold increases, which may cause hot-spots before the traffic carried by elephant flows reaches the threshold.

Network modifications. ESCA proposes a sampling method that depends on non-existing SDN specifications, hence, requiring custom-made switch hardware. In contrast, OFSP, FlowSeer, NELLY, and Mahout rely on OpenFlow [48], therefore enabling the use of commercial switches. Essentially, NELLY and Mahout require the installation of additional software in the servers, which need only to be done once with further configuration possible on the basis of a policy manager or autonomously. This installation can be carried out by using DevOps automation tools, such as Puppet and Chef, that support the distribution of software components to the operating systems of servers [268]. Moreover, virtualization platforms, such as VMWare and Xen, support software distribution to the servers as updates to the hypervisor without interrupting running virtual machines (either by live-migration or live-updating) [269].

Performance factors. Depending on the location of the flow detection method, different factors may affect its performance. Controller-side methods (i.e., OFSP, ESCA, and FlowSeer) rely on the resources available at the controller and ToR switches. The controller should be powerful enough for detecting all the elephants and processing the control packets sent by the ToR switches in the DCN. Similarly, the ToR switches should have enough memory for installing the required flow table entries. Moreover, the accuracy of the controller-side methods can be negatively affected if the ToR switches drop some of the first packets of the elephant flows. On the other hand, NELLY and Mahout operate at the server-side, so they depend on servers resources. As NELLY is based on ML, it requires more resources than does Mahout. Both server-side methods detect the elephants generated by each server (i.e., distributed operation). Note that servers should be able to monitor the first packets of the elephant flows for avoiding a decrease in accuracy.

3.4 Final remarks

ECMP, which is the default routing technique in DCNs, can degrade the network performance when handling mouse and elephant flows. Novel techniques for scheduling the elephant flows can alleviate this problem. Recently, several approaches have incorporated ML techniques at the controller-side of SDDCNs to detect elephant flows. However, these approaches can produce heavy traffic overhead, low scalability, low accuracy, and high detection time. In this chapter, we introduced NELLY to deal with this limitations. NELLY performs continuous learning and requires limited memory resources by virtue of using incremental learning. An extensive evaluation based on real packet traces and various incremental learning algorithms demonstrated the high accuracy and speed of NELLY when used with the ARF and AHOT algorithms. Moreover, an analytical comparison to seminal related works corroborated the scalability of NELLY as well as its generation of low traffic overhead and the fact that no modifications in SDN standards are required.

Chapter 4

Multipath routing based on software-defined networking and machine learning for data center networks

The majority of the flows in DCNs are mice (i.e., , small, short-lived flows), whereas only very few are elephants (i.e., , large, long-lived flows) [8–11]. Mouse flows represent latency-sensitive and bursty network traffic, such as search results [260], and elephant flows depict massive data traffic, such as server migrations [270]. These traffic characteristics negatively impact the performance of mice in DCNs as elephants tend to utilize most bandwidth, introducing delay to mouse flows sharing the same links.

To tackle this problem, first, this chapter proposes a PMAC-based multipath routing algorithm for steering traffic flows (mainly, mice) in SDDCNs that supports transparent host migration across the whole network while reducing the number of rules installed on SDN switches, decreasing the delay introduced to flows traversing the network. Second, this chapter introduces a flow rescheduling method at the controller-side of SDDCNs that applies incremental deep learning for predicting traffic characteristics of elephant flows to compute and install the best path per elephant flow across the network.

4.1 Pseudo-MAC-based multipath routing in software-defined data center networks

As described in Section 2.3.3, several SDDCN (i.e., DCNs using SDN) multipath routing mechanisms focus on dynamically rescheduling paths for identified elephants from the controller while relying on a default multipath routing algorithm, like ECMP [7], for handling the rest of the flows (potentially, mice). However, most of these SDN-based mechanisms do not specify how to implement such a default multipath routing algorithm in an SDN enviroment. The simplest SDN implementation would dynamically compile and install the path for each flow from the controller. However, this implementation introduces a delay (approximately, 10 ms[1]) when edge switches (a.k.a. ToR) send the first packet of each flow to the controller [27, 271]. This delay negatively affects the latency-sensitive mouse flows [272]. Moreover, the massive number of flows in DCNs leads to scalability issues due to a large occupancy of the narrow flow tables in SDN switches and a significant overload to the controller, which increases the processing delay [273].

Implementing the default multipath routing algorithm (e.g., ECMP) by installing static flow rules overcomes the controller-related drawbacks (i.e., overload and first packets delay). However, this implementation usually relies on multipath routing algorithms that install a high number of source-destination (either Media Access Control (MAC) or IP) static flow rules. As aforementioned, an SDN switch with many flow rules installed creates scalability issues and increases the delay while the switch matches the rule for a specific flow. Some approaches have proposed algorithms for reducing the number of flow rules installed, particularly for mice [15, 28, 274]. However, these approaches do not address the complexity of continuously updating the static flow rules in case of dynamic changes in the network state. Therefore, an efficient implementation of the default multipath routing algorithm should avoid sending the first flow packet to the controller, install the less possible number of flow rules in switches, and update these rules as the network state changes.

This section introduces an SDN-based multipath routing algorithm, named Pseudo-MAC-based Multipath (PM2), which performs efficient routing of flows (mainly, mice) in DCNs following the fat-tree topology [242]. Unlike other proposals that use addresses (either MAC or IP) from hosts, PM2 identifies each switch's layer (i.e., edge, aggregation, core) and position for generating Pseudo-MAC (PMAC) prefixes, which allow installing routing flow rules with wildcards to save space in the flow tables. PM2 then intercepts Address Resolution Protocol (ARP) messages at the controller to generate a PMAC address for each host (virtual and physical) and install the corresponding flow rules in edge switches for both parsing MAC addresses and reaching destination. Results reveal that PM2 significantly reduces the number of rules installed in switches while supporting transparent host migration across the whole SDDCN.

The remainder of this section is as follows. Section 4.1.1 demonstrates the impact of the number of switch installed flow rules on the routing delay. Section 4.1.2 describes the algorithm and procedures of PM2. Section 4.1.3 presents a proof of concept of PM2

[1] Assuming that only the first packet goes to the controller. If multiple packets arrive (i.e., bursty traffic) before installing the flow rule, more packets will bear the cost.

in an emulated scenario. Section 4.1.3 compares PM2 with feasible routing approaches in SDDCNs regarding the number of rules installed.

4.1.1 Motivation

Figure 4.1 shows three experimental scenarios that we use to explain how the number of flow rules installed in a switch table impact the delay of flows traversing that switch. All three scenarios deploy a Ryu controller [66] running an algorithm developed in Python that simply installs static flow rules into the flow table of the corresponding switch. The *emulated* scenario uses the network emulation tool Mininet 2.2.2 [275] to deploy two virtual hosts[2] (h) and an OpenFlow switch in a single Virtual Machine (VM). The Ryu controller runs in the same Physical Computer (PC) hosting the VM. The *virtual* scenario deploys two virtual hosts and an Open vSwitch 2.5.4 [277] using different VMs interconnected through virtual Ethernet interfaces [278]. The Ryu controller and the three VMs run in the same PC. The *physical* scenario deploys an HP 2920 OpenFlow switch (a.k.a. Aruba 2920) [279] that interconnects two PCs (physical hosts) through OpenFlow VLAN interfaces. The Ryu controller runs in an independent PC. The PCs for all three scenarios shared the following system specifications: Lubuntu 16.04 operating system, 2.40 GHz Intel Core i5 processor, and 3 GB RAM.

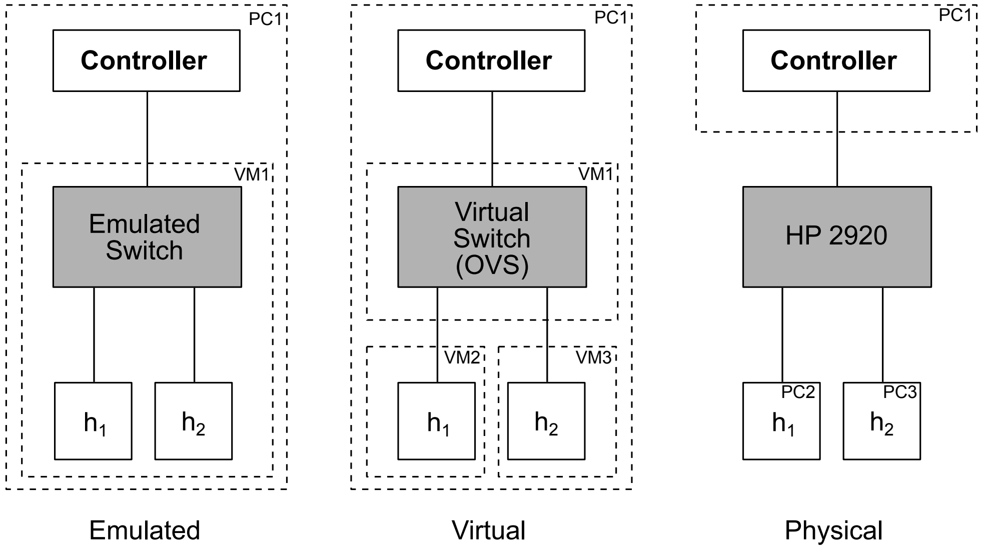


Figure 4.1: RTT measurement experimental scenarios

We measured the RTT for all three scenarios while varying the number and arrangement of flow rules installed in the corresponding switch. RTT represents the total time a packet takes to go to the destination and back to the source. The number of installed flow rules (r) ranged from 1000 to 16000 in steps of 1000 ($r \in R =$

[2] Mininet virtual hosts are processes running in their own Linux network namespace [276].

$\{1000, 2000, ..., 16000\}$), where only two rules matched and forwarded the packets sent between the hosts (*routing rules*) while the remaining occupied the flow table memory (*filling rules*). For each $r \in R$, the arrangement of the flow rules varied by placing the routing rules at the beginning, middle, and end of the filling rules. We installed each flow rule arrangement T times in the switch and took the RTT N times for each $t \in T$. Each experiment was about sending a ping request from the first host to the second one and then measuring the RTT in the first host after receiving the corresponding ping reply from the second host.

Figure 4.2 depicts the results for each scenario with $T = 30$ and $N = 30$. These results reveal that the RTT for the emulated scenario remains around 0.1 ms regardless of the number of installed flow rules and the position of the routing rules. Similarly, the RTT for the virtual scenario persists as the number and arrangement of the installed flow rules vary, though it approximates to 1.3 ms. Such an increment in the RTT value is because the virtual scenario deploys more virtual Ethernet interfaces between the hosts than the emulated scenario. In contrast, the RTT for the physical scenario increases from 0.8 ms up to 2.1 ms as the number of installed flow rules grows and as the position of the routing rules moves to the end of the filling rules. For example, for routing rules placed at the beginning of the filling rules, the RTT grows from 0.8 ms for $r = 1000$ to 1.1 ms for $r = 16000$. Whereas, for $r = 16000$, the RTT increases to 1.6 ms and 2.1 ms when placing the routing rules in the middle and end of the filling rules, respectively.

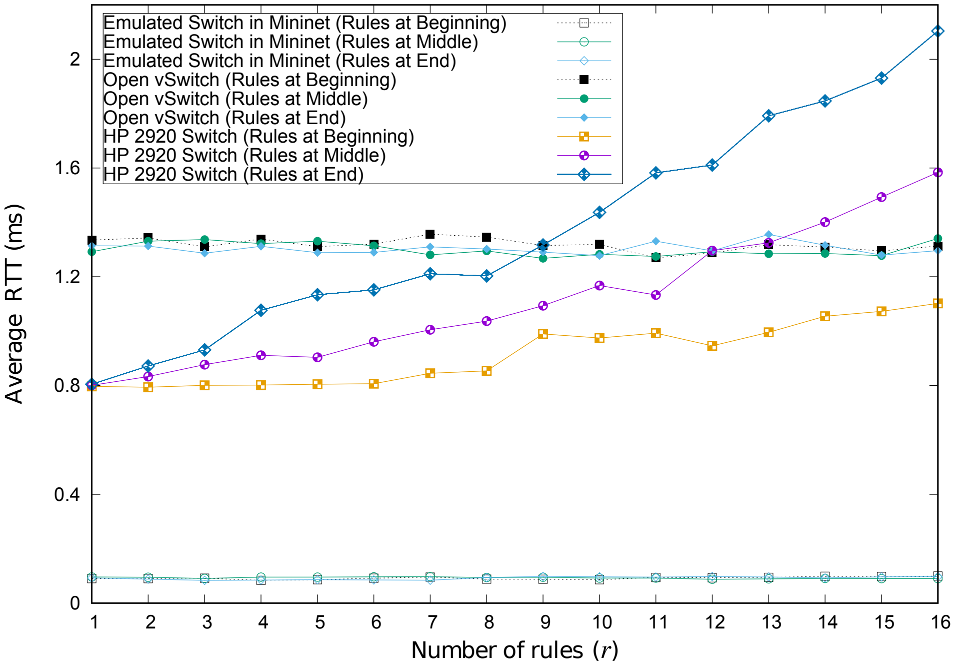


Figure 4.2: Average RTT per number of flow rules installed in the switch of each experimental scenario with $T = 30$ and $N = 30$

We learned from these experiments that the RTT in the emulated and virtual scenarios remains nearly constant while varying the number of installed flow rules. Second, in the physical scenario, the limited memory in switches causes the number of installed flow rules to impact the delay introduced to the packets of flows traversing the switch. Thus, reducing the number of flow rules installed in SDN switches represents an opportunity to lower the delay introduced to packets of flows (mainly, mice) traveling in SDDCNs.

4.1.2 Pseudo-MAC-based multipath routing

PM2 provides a multipath routing algorithm for steering flows (mainly, mice) in a fat-tree SDDCN. PM2 focuses on supporting transparent host migration across the whole SDDCN while reducing the number of rules installed in the switches, decreasing the delay introduced to the packets of flows traversing the network. To do so, PM2 reuses the key concept of Pseudo-MAC (PMAC), which defines a hierarchical MAC address assigned to the physical and virtual hosts in a fat-tree DCN topology [280].

PM2 generates and assigns 48-bit PMACs using the form $lia{:}pod{:}pos{:}port{:}vmid$, where lia, pod, pos, and $port$ represent eight-bit fields, whereas $vmid$ depicts a 16-bit field. lia defines PMACs as local-individual addresses (e.g., 0x$0a$ or 000010**10**[3]). pod depicts the number of the pod containing the edge switch. pos describes the position of the edge switch within the pod. $port$ details the switch port number connected with the physical host. $vmid$ identifies different virtual hosts using a bridge adapter in the same physical host (0x00 if no virtual hosts using bridge adapters).

Figure 4.3 presents the process executed by PM2 for installing the routing rules. This process involves four main tasks: (*i*) generate PMAC addresses, (*ii*) define the set of routing rules based on the PMAC addresses, (*iii*) install the set of routing rules in the corresponding switches; and (*iv*) update the routing rules when network changes occur. Note the PM2 process runs after either discovering the topology or intercepting an ARP message.

Topology discovery

PM2 requires an overview of the fat-tree DCN topology, including the position of the switches and their connections to the other switches, for generating the PMAC addresses and installing the required routing rules. The fat-tree topology represents a k-ary network that consists of k-port switches distributed into three layers: core, aggregation, and edge, top-down. There are k pods interconnected by $\frac{k^2}{4}$ core switches.

[3] A MAC address with the second-least-significant bit of the first octet set to **one (1)** represents a locally administered address, whereas setting to **zero (0)** the least-significant bit of the first octet defines an individual address meant for unicast communication [281].

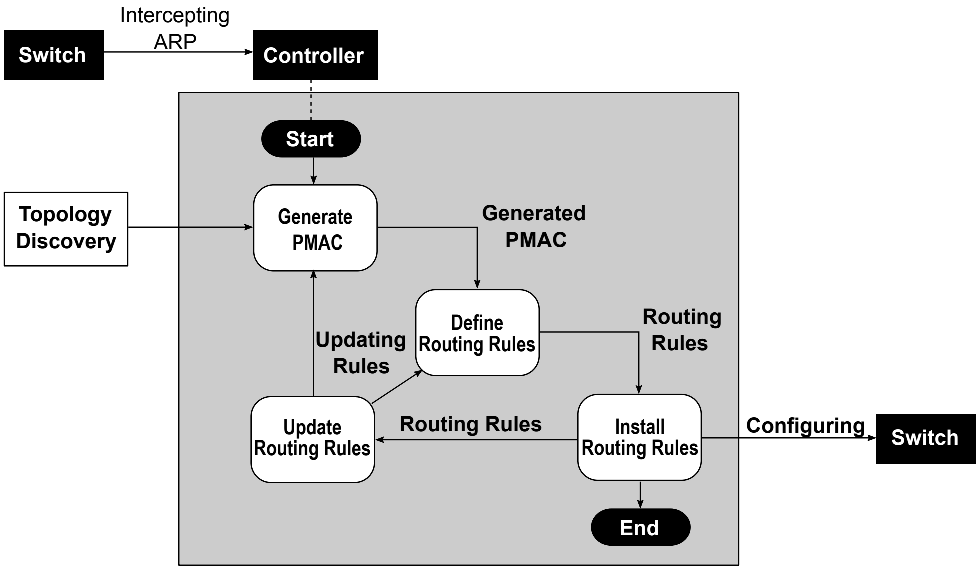


Figure 4.3: PM2 process to install routing rules

Each pod contains $\frac{k}{2}$ aggregation switches (upper pod) and $\frac{k}{2}$ edge switches (lower pod). Aggregation switches connect to the core switches. Each aggregation switch connects to each edge switch in the same pod. Each edge switch connects to $\frac{k}{2}$ physical hosts. The fat-tree topology provides a high degree of available path diversity: between any source and destination host pair from different pods, $\frac{k^2}{4}$ equal-cost paths exist, each corresponding to a core switch, although the paths are not link-disjoint. Table 4.1 depicts the fat-tree topology size, in terms of hosts, switches, and links, based on the typical number of ports in switches.

For *topology discovery*, PM2 relies on the Link Layer Discovery Protocol (LLDP) to obtain the connections among the switches. To do so, first, PM2 installs a routing rule on every switch for redirecting any received LLDP packet to the controller, and then frequently instructs every switch for sending LLDP packets through all the ports. The switch connections allow PM2 determining the position of the switches in the fat-tree topology. Assuming the network is fully connected, PM2 identifies as edge switches those ones that failed to receive LLDP packets from half of their ports (i.e., ports connected to physical hosts). In case the network is not fully connected, PM2 might rely on the switch identifier (ID) to identify edge switches. In OpenFlow, for example, the switch ID is a 64-bit field known as datapath ID: the 48 Least Significant Bits (LSB) correspond to the switch MAC, whereas the 16 Most Significant Bits (MSB) depend on the switch implementation (vary among models). Most OpenFlow switch implementations allow defining a custom MSB, enabling network administrators to set a specific value in the datapath ID of edge switches that PM2 identifies when the switches connect to the controller. Finally, switches connected to edge switches become aggregation switches, and those connected to aggregation switches become core switches.

Table 4.1: Fat-tree topology size

Order* **(k)**	**Pods**	**Hosts**	**Edge switches**	**Aggregation switches**	**Core switches**	**Total switches**	**Links**
4	4	16	8	8	4	20	48
8	8	128	32	32	16	80	384
10	10	250	50	50	25	125	750
12	12	432	72	72	36	180	1296
16	16	1024	128	128	64	320	3072
24	24	3456	288	288	144	720	10368
28	28	5488	392	392	196	980	16464
48	48	27648	1152	1152	576	2880	82944
52	52	35152	1352	1352	676	3380	105456

** Based on the typical number of ports in switches*

Having determined the position of the switches in the fat-tree topology, PM2 generates PMAC prefixes for the pods and edge switches. PMAC prefixes for the pod and edge switches have the form $lia{:}pod{:}*$ and $lia{:}pod{:}pos{:}*$, respectively, where $*$ represents the remaining wildcard bits (32 bits and 24 bits, respectively). For example, the pod two in the fat-tree topology would have the PMAC prefix $0a{:}02{:}*$, whereas the edge switch in the position one of the pod two would have the PMAC prefix $0a{:}02{:}01{:}*$. Note that all the hosts (virtual and physical) under the same pod and edge switch share the same PMAC prefix.

After generating the PMAC prefixes, PM2 defines and installs the wildcard routing rules in the core and aggregation switches for enabling the top-down communication (i.e., from core to aggregation and from aggregation to edge). Core switches only need to match the PMAC prefix of a pod for sending a packet to a host that is under that pod. For example, if the port two of a core switch connects to the pod two, the core switch would have installed a wildcard routing rule that forwards through the port two every packet whose destination MAC address matches the PMAC prefix $0a{:}02{:}*$. Similarly, aggregation switches only need to match the PMAC prefix of an edge switch for sending a packet to a host that is connected to that edge switch. For example, if the port one of an aggregation switch in the pod two connects to the edge switch in position one, the aggregation switch would have installed a wildcard routing rule that forwards through the port one every packet whose destination MAC address matches the PMAC prefix $0a{:}02{:}01{:}*$. These wildcard routing rules based on PMAC prefixes enable reducing the number of flow rules installed on switches at the core and aggregation layers.

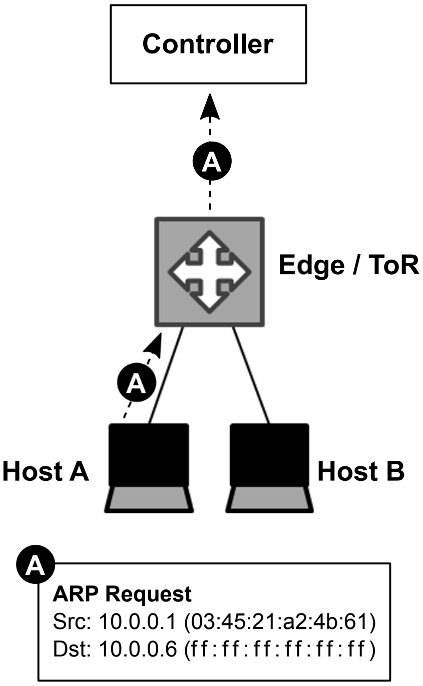
PM2 also defines and installs group routing rules at the edge and aggregation switches for enabling the bottom-up communication (i.e., from edge to aggregation and from aggregation to core). These group routing rules represent low priority rules that switches apply when none of the PMAC-based routing rules, with a higher priority, match the packet. Therefore, if a packet from a source host arrives at an edge switch but the destination host is not connected to the same edge switch, this switch applies a group routing rule to select one of the ports connected to the aggregation switches to forward the packet. Similarly, if a packet from an edge switch arrives at an aggregation switch but the destination host is not in the same pod, the aggregation switch applies a group routing rule select one of the ports connected to the core switches to forward the packet. These group routing rules enable multipath routing between the equal-cost paths in the fat-tree topology. For example, OpenFlow [55] provides a group table that contains group entries, each consisting of a list of action buckets (e.g., forward through port two, forward through port three, etc.). OpenFlow switches apply the actions in one or more action buckets depending on the group type. Particularly, the select group type executes one of the action buckets in the group. The selection of the action bucket in the group depends on the switch implementation. Open vSwitch, for instance, applies a hash function on the packet header for selecting the action bucket in the group.

The last routing rules that PM2 defines using the topology discovery and installs on the switches are for handling ARP messages. PM2 installs a routing rule on every edge switch for intercepting all the ARP requests and replies. To do so, the ARP intercepting routing rule matches all the packets with the EtherType field set to the ARP protocol (0x0806) and forwards them to the controller. At the core and aggregation layer, PM2 simply instructs the switches to drop all the ARP messages. Note the ARP routing rules present the highest priority among the installed routing rules.

Finally, PM2 triggers the update of routing rules when discovering changes in the topology, such as broken links or switches down. That way, PM2 avoids forwarding packets through disconnected ports, minimizing the rate of droppped packets. To do so, PM2 generates a new configuration of routing rules and compares it with the current configuration to enforce only the modifications. Note this updating task might imply generating new PMAC prefixes, installing new routing rules, and deleting existing routing rules.

ARP interception

PM2 also operates when intercepting ARP messages. As shown in Figure 4.4, let's assume that a source host A, using the IP 10.0.0.1, wants to communicate with a destination host B, using the IP 10.0.0.6. The source host A initially ignores the IP 10.0.0.6 is served by the destination host B, so host A sends an ARP request to ask "who has 10.0.0.6?" and "what is you MAC address?" in order to get the MAC address from host B. Since host A ignores the MAC address from host B, the ARP request uses the broadcast MAC address (ff:ff:ff:ff:ff:ff) for the destination MAC. This ARP request matches the intercepting routing rule at the edge switch, which forwards it to the controller.



ARP Request
Src: 10.0.0.1 (03:45:21:a2:4b:61)
Dst: 10.0.0.6 (ff:ff:ff:ff:ff:ff)

Figure 4.4: ARP request interception in PM2

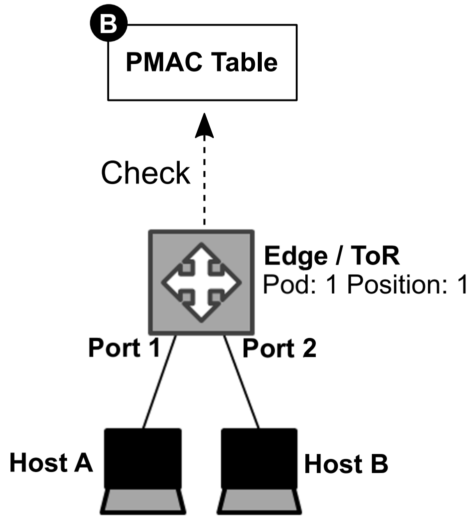
As depicted in Algorithm 4.1, PM2 receives as input the intercepting edge switch data (sw) and the intercepted ARP message (A). The intercepting edge switch data includes the switch ID (id) and the input port number the edge switch received the ARP packet in (in_port). Note the intercepting edge switch data is not part of the ARP message but of the messages sent to the controller by SDN protocols (e.g., OpenFlow packet-in message). The intercepted ARP message contains the fields $oper$, ip_src, ip_dst, eth_src, and eth_dst, which represent the ARP operation code (e.g., request or reply), the source IP address, the destination IP address, the source MAC address, and the destination MAC address. In the example from Figure 4.4, the value of the $oper$ field is 0x01 since the ARP message corresponds to an ARP request (this field is 0x02 for ARP reply).

Algorithm 4.1 shows that PM2, first, generates the PMAC address for the source host using the switch ID ($sw.id$), the input port ($sw.in\_port$), the source IP ($A.ip\_src$), and the source MAC ($A.eth\_src$). Then, PM2 inserts the host generated PMAC into the PMAC table, associating it with the source IP address ($A.ip\_src$). Note that the switch ID ($sw.id$), the input port ($sw.in\_port$), and the source MAC ($A.eth\_src$) are also stored in the PMAC table record. In Figure 4.5, for example, the PMAC table associates the IP

Algorithm 4.1: Handling intercepted ARP messages in PM2

```
data: $sw$        edge switch that intercepted ARP message
      $A$         intercepted ARP message data
      $T_H$       PMAC table for hosts (virtual or physical)
      $R$         installed PMAC routing rules
      $E$         set of edge (ToR) switches
      $\theta_S$  ARP stale time threshold
// Function to handle intercepted ARP messages
function HANDLE_ARP($sw$, $A$):
    // Generate and store PMAC
    $pmac \leftarrow$ GENERATE_PMAC($sw.id$, $sw.in\_port$, $A.ip\_src$, $A.eth\_src$);
    $T_H[A.ip\_src] \leftarrow \{pmac, sw.id, sw.in\_port, A.eth\_src\}$;
    // Install PMAC routing rules
    if $\{sw.id, A.eth\_src, pmac\} \notin R$ then
        INSTALL_PMAC_RULES($sw.id$, $A.eth\_src$, $pmac$, $sw.in\_port$);
        $R \leftarrow \{sw.id, A.eth\_src, pmac\}$;
    end
    // Update ARP data
    $A.eth\_src \leftarrow pmac$;
    UPDATE_TIME($A.dst\_ip$);
    // Check if ARP request
    if $A.oper = 1$ then
        // Check if destination host is known
        if $A.dst\_ip \notin T_H$ then
            $actions \leftarrow [flood]$;
            for each $e \in E$ do
                $e$.SEND_PACKET($A$, $actions$);
            end
        else
            // Check if stale time is shorter than threshold
            if STALE_TIME($A.dst\_ip$) $> \theta_S$ then
                CHECK_CONNECTION($A.dst\_ip$);
            else
                // Send ARP reply with source MAC set to destination PMAC
                $dst\_pmac \leftarrow T_H[A.dst\_ip][pmac]$;
                $reply \leftarrow$ ARP_REPLY($A$);
                $reply.eth\_src \leftarrow dst\_pmac$;
                $actions \leftarrow [A.in\_port]$;
                $sw$.SEND_PACKET($reply$, $actions$);
            end
        end
    else
        // Forward ARP reply
        $dst\_swid, dst\_port \leftarrow T_H[A.dst\_ip]$;
        $dst\_sw \leftarrow$ GET_SWITCH($dst\_swid$);
        $actions \leftarrow [dst\_port]$;
        $dst\_sw$.SEND_PACKET($A$, $actions$);
    end
end
```

10.0.0.1 with the PMAC $0a:01:01:01:00:00$, which was generated using the ARP request sent by host A, whose MAC is $03:45:21:a2:4b:61$ and connects to the edge switch on port one.



B

IP	PMAC	MAC	Port
10.0.0.1	0a:01:01:01:00:00	03:45:21:a2:4b:61	1
10.0.0.6	0a:01:01:02:00:00	08:d4:6c:59:23:1e	2

Figure 4.5: PMAC table in PM2

Continuing with Algorithm 4.1, PM2 defines and installs on the intercepting edge switch the routing rules associated with the generated PMAC, in case it has not been done yet. The PMAC routing rules consist of two routing rules per host on the edge switch the host connects to. The first rule updates the source MAC of any packet received from any of the hosts connected to the edge switch. This rule parses the actual MAC of the host to the PMAC generated for that host. For example, in Figure 4.5, this rule parses the source MAC from $03:45:21:a2:4b:61$ to $0a:01:01:01:00:00$ for any packet host A sends to the edge switch. The second rule updates the destination MAC of any packet targeting any host connected to the edge switch and forwards the packet though the corresponding port. This rule parses the PMAC generated for the host to the actual MAC of that host. For example, in Figure 4.5, this rule, first, parses the destination MAC from $0a:01:01:01:00:00$ to $03:45:21:a2:4b:61$ for any packet targeting host A, and then forwards the packet through port one.

Next, Algorithm 4.1 depicts that PM2 updates the source MAC ($A.eth\_src$) of the ARP message by setting the generated PMAC. This enables each host to associate the IPs to the PMACs generated for the other hosts instead of using their actual MACs. Therefore, when the hosts send packets targeting any IP, they will set the source MAC with their own actual MAC addresses and the destination MAC with the corresponding

PMAC that the controller reported for such IP. In our example, PM2 would parse the source MAC of the ARP request from $03{:}45{:}21{:}a2{:}4b{:}61$ to $0a{:}01{:}01{:}01{:}00{:}00$. Following, PM2 checks the operation code of the intercepted ARP message ($A.oper$) to discriminate ARP requests and ARP replies. For intercepted ARP requests ($A.oper = 1$), if the PMAC table ignores the IP destination address ($A.dst\_ip$), PM2 instructs all the edge switches to flood the updated ARP request (i.e., send the packet through all ports) to find the host with the destination IP. The destination host then receives the updated ARP request and responds with an ARP reply to inform that the destination IP is at the corresponding MAC. For example, in Figure 4.5, host B would respond that IP 10.0.0.6 is at MAC $08{:}d4{:}6c{:}59{:}23{:}1e$. Note the ARP reply matches the intercepting routing rule at the edge switch, which forwards it to the controller.

As show in Algorithm 4.1, PM2 also leverages ARP replies for generating host PMACs, which are further stored in the PMAC table and used for defining and installing the PMAC routing rules on the intercepting edge switch. In Figure 4.5, for example, the PMAC table associates the IP 10.0.0.6 with the PMAC $0a{:}01{:}01{:}02{:}00{:}00$, which was generated using the ARP reply sent by host B, whose MAC is $08{:}d4{:}6c{:}59{:}23{:}1e$ and connects to the edge switch on port two. Subsequently, PM2 updates the source MAC ($A.eth\_src$) by setting the generated PMAC and recognizes the ARP message is an ARP reply ($A.oper = 2$), so it forwards the updated ARP reply to the destination host via the corresponding edge switch and port. In our example, PM2 would parse the source MAC of the ARP reply from $08{:}d4{:}6c{:}59{:}23{:}1e$ to $0a{:}01{:}01{:}02{:}00{:}00$ and forward it through port one towards host A.

When validating if the PMAC table knows the IP destination from the ARP request ($A.dst\_ip$), in case it does but the controller has not intercepted an ARP message (request or reply) from that IP for longer than a stale time threshold (θ_S), PM2 checks the connection by sending the updated ARP request directly to the known destination host. When the destination host replies, PM2 forwards the updated ARP reply to the source host, reporting that the destination IP is at the generated PMAC. If the controller does not intercepts an ARP reply from the destination host for longer than a removal time threshold (θ_R), PM2 removes the record associated with the destination IP from the PMAC table. On the other hand, in case the PMAC table knows the destination IP from the ARP request ($A.dst\_ip$) and the time since the controller intercepted an ARP message from that IP is shorter than θ_S, PM2 generates and sends an ARP reply to the source host, reporting that the destination IP is at the generated PMAC. Note that θ_S and θ_R enable recognizing hosts down to avoid filling the network with traffic (e.g., UDP) that has no destination.

Finally, PM2 supports transparent migration of hosts across the whole SDDCN. Transparent migration refers to moving a host (virtual or physical) while keeping its IP and being able to communicate with that host without performing manual configurations. In PM2, when changing the position of a host, any source host communicating with it stops receiving replies for the direct ARP requests. Source hosts will receive an ARP reply either when one of them sends a broadcast ARP request or when the migrated host sends an ARP request for communicating with any other host. These ARP messages allow discovering the new position of the migrated host, for which PM2 updates the PMAC address and the corresponding routing rules.

4.1.3 Implementation

Figure 4.6 depicts the proof of concept we implemented for evaluating the feasibility of PM2. We deployed this implementation on a server at the Computer Networks Laboratory (LRC, Laboratório de Redes de Computadores) of the University of Campinas. This server runs Debian 8.11 server in a 2.66 GHz Intel Xeon Processor X5650 machine, with 12 physical cores, 24 logical cores, and 40 GB RAM. On the server, we deployed a VM for running Mininet [275], a well-known tool for emulating OpenFlow networks [48]. We installed Mininet 2.3.0 and Open vSwitch 2.5.9 [277], which Mininet uses for deploying the OpenFlow switches in the emulated network. Open vSwitch 2.5.9 fully supports up to OpenFlow 1.3 and OpenFlow 1.5 with missing features. The Mininet VM runs an Ubuntu 16.04 server with 6 logical cores and 20 GB RAM. Using the Mininet Python API, we developed a custom fat-tree topology that accepts as input a parameter k for setting the size of the fat-tree network (see Table 4.1). Our custom fat-tree topology implementation is available in [282]. We executed our custom fat-tree topology on the Mininet VM, varying the size of the network.

On the server, we also deployed the Ryu SDN framework [66] as the controller for managing the OpenFlow switches in Mininet. We installed Ryu 4.34, which fully supports up to OpenFlow 1.5 [55]. Using the Python libraries from the Ryu SDN framework, we developed a network application that implements PM2, including the topology discovery, the ARP management, and the process for generating PMAC addresses, defining the routing rules, and installing them on the OpenFlow switches. Note we assumed a fully connected network, so our PM2 implementation identifies as edge switches those ones that failed to receive LLDP packets from half of their ports. Our Ryu network application that implements PM2 is available in [282]. We executed our PM2 network application on the Ryu SDN framework.

To test that PM2 installs the appropriate routing rules, we performed a ping test between all the hosts in the emulated fat-tree network, validating that all of them were able to communicate with each other. Due to resource limitations, we were able to test the communication for a fat-tree topology of size 16, which deploys 1024 hosts, 320 switches (128 at edge layer, 128 at aggregation layer, and 64 at core layer), and 3072 links (see Table 4.1).

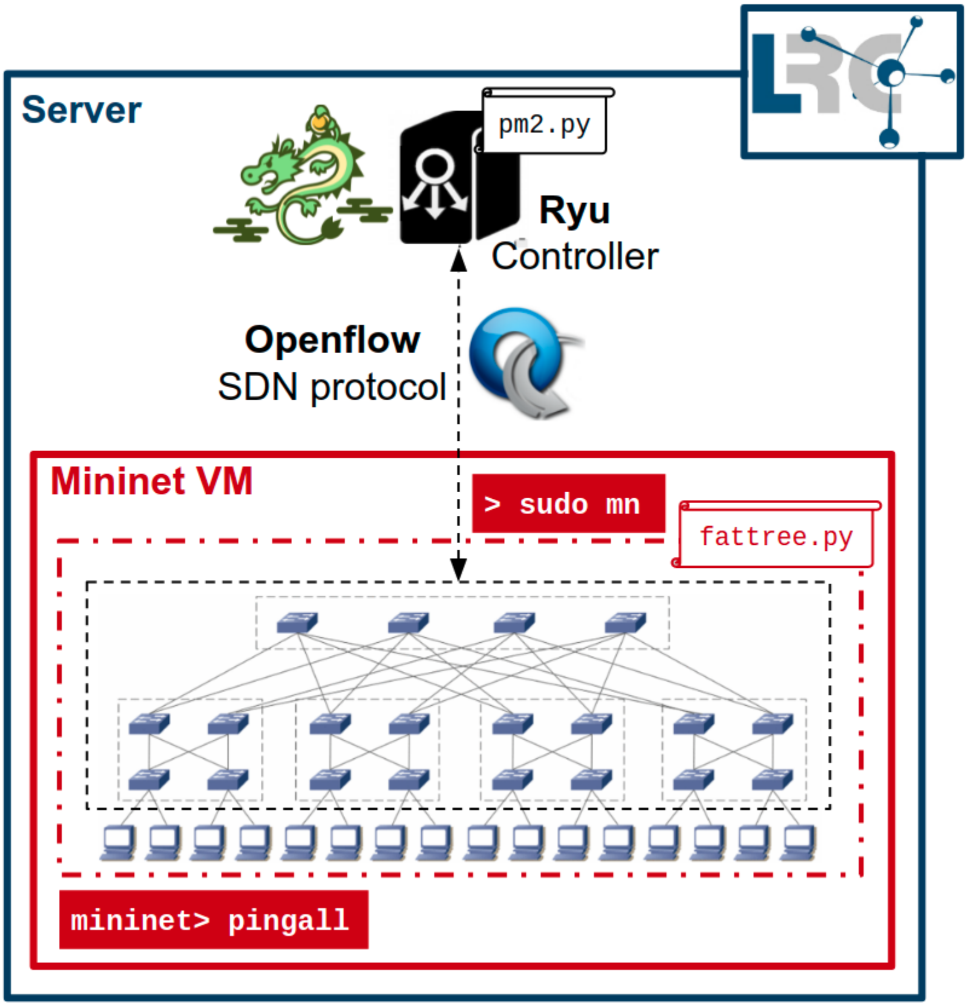


Figure 4.6: Implementation of PM2

4.1.4 Analytic evaluation

PM2 enforces a PMAC-based routing for reducing the number of routing rules installed on the switches. PM2 is similar to PortLand [280] in that both assign PMAC addresses to the hosts according to their position in the fat-tree topology. However, for topology discovery, PortLand depends on a custom Location Discovery Protocol (LDP), whereas PM2 relies on the widely-used LLDP. Moreover, unlike PortLand, PM2 defines and installs the routing rules for supporting multipath routing (i.e., group routing rules). PortLand simply assumes a standard technique such as flow hashing in ECMP. Finally, PM2 validates the time that hosts have been in communication with others (i.e., θ_S and θ_R) to identify hosts down, avoiding filling the network with traffic (e.g., UDP) that has no destination. PortLand does not account for hosts down.

We further evaluated PM2 by analyzing the number of rules installed per switch for an all-to-all communication in an OpenFlow fat-tree topology, in comparison with other three feasible approaches: *(i)* Layer 2 (L2), which uses MAC addresses for communication, *(ii)* Hierarchical Layer 3 (HL3), which divides the network using IP prefixes; and *(iii)* a hybrid approach between L2 and HL3 (L2-HL3), which uses MAC addresses from the aggregation layer down and divides the pods using IP prefixes. Hereinafter, k denotes the order (i.e., size) of the fat-tree topology (see Table 4.1).

First, we compute the number of rules that PM2 installs on the switches of an OpenFlow fat-tree topology. Equation 4.1 describes the number of rules that PM2 installs per edge switch. The four (4) rules are the ARP intercepting rule, the group rule, a miss destination MAC rule that redirects to the group rule, and a miss source MAC rule that handles packets coming from the aggregation layer. Moreover, PM2 installs two PMAC routing rules per each host h in each server machine $s \in 1..\frac{k}{2}$ connected to the edge switch. The first PMAC rule parses the source MAC, whereas the second rule parses the destination MAC and forwards the packet through the corresponding port. Note that h includes the physical host and the virtual hosts using a bridge adapter to connect to the network. Virtual hosts using NAT adapters communicate to the network using the IP and MAC addresses of the physical host.

$$4 + 2 \cdot \sum_{s=1}^{k/2} h_s \tag{4.1}$$

Equation 4.2 depicts the number of rules that PM2 installs per aggregation switch. The three (3) rules are the ARP dropping rule, the group rule, and a miss destination MAC rule that redirects to the group rule. In addition, PM2 installs a wildcard routing rule per each of the $\frac{k}{2}$ edge switches connected to the aggregation switch. Note these wildcard rules use the PMAC prefixes generated for the edge switches (i.e., $lia{:}pod{:}pos{:}*$).

$$3 + \frac{k}{2} \tag{4.2}$$

Equation 4.3 presents the number of rules that PM2 installs per core switch. One (1) ARP dropping rule and a wildcard routing rule per each of the k pods connected to the core switch. These wildcard rules use the PMAC prefixes generated for the pods ($lia{:}pod{:}*$).

$$1 + k \tag{4.3}$$

Next, we calculate the number of rules installed on the switches by the other three OpenFlow-feasible approaches, namely, L2, HL3, and L2-HL3. Equation 4.4 denotes the number of rules that these three routing approaches install per edge switch. The three (3) rules are an ARP management rule, the group rule, and the miss destination MAC rule that redirects to the group rule. Furthermore, these approaches install a routing rule per each host h in each server machine $s \in 1..\frac{k}{2}$ connected to the edge switch.

$$3 + \sum_{s=1}^{k/2} h_s \tag{4.4}$$

Equation 4.5 describes the number of rules that L2 and L2-HL3 install per aggregation switch. The three (3) rules are an ARP management rule, the group rule, and the miss destination MAC rule that redirects to the group rule. Moreover, L2 and L2-HL3 install a routing rule per each host h in each server machine $s \in 1..\frac{k}{2}$ connected to each edge switch $e \in 1..\frac{k}{2}$ connected to the aggregation switch. In contrast, the number of rules that HL3 installs per aggregation switch is given by Equation 4.2. Similar to PM2, HL3 leverages address prefixes for reducing the number of rules at the aggregation layer.

$$3 + \sum_{e=1}^{k/2} \sum_{s_e=1}^{k/2} h_{s_e} \tag{4.5}$$

Lastly, Equation 4.6 depicts the number of rules that L2 installs per core switch. One (1) ARP management rule and a routing rule per each host h in each server machine $s \in 1..\frac{k}{2}$ connected to each edge switch $e \in 1..\frac{k}{2}$ belonging to each pod $p \in 1..k$. In contrast, the number of rules that HL3 and L2-HL3 install per core switch is shown in Equation 4.3. Similar to PM2, HL3 and L2-HL3 leverage address prefixes for reducing the number of rules at the core layer.

$$1 + \sum_{p=1}^{k} \sum_{e_p=1}^{k/2} \sum_{s_{e_p}=1}^{k/2} h_{s_{e_p}} \tag{4.6}$$

For facilitating the analytic comparison, we make some assumptions to compute the number of rules all these approaches install per switch for each layer in a fat-tree topology. Table 4.2 simplifies Equations 4.1-4.6 by assuming only one host (i.e., the physical host) in each server machine (i.e., $h_s = 1$). Furthermore, Figure 4.7 shows the number of rules each approach (PM2, L2, HL3, L2-HL3) install per switch for each layer in a fat-tree topology of size $k = 48$ and only one host in each server machine (i.e., $h_s = 1$). Note that a fat-tree topology of size $k = 48$ consists of 27648 hosts, 2880 switches (1152 at edge layer, 1152 at aggregation layer, and 576 at core layer), and 82944 links (see Table 4.1). Moreover, the Y axis of Figure 4.7 is in logarithmic scale.

The results show that PM2, HL3, and L2-HL3 installs much less rules per core switch than L2. Note the former three leverage address prefixes (either PMAC or IP) for reducing the number of rules at the core layer. At the aggregate layer, PM2 and HL3 install much fewer rules than L2 and L2-HL3. In this case, only PM2 and HL3 leverage address prefixes (PMAC and IP, respectively) for reducing the number of rules

Table 4.2: Comparison of PM2 routing with related approaches

FEATURE	L2	HL3	L2-HL3	PM2
Rules per edge (ToR) switch*	$3+\frac{k}{2}$	$3+\frac{k}{2}$	$3+\frac{k}{2}$	$4+k$
Rules per aggregation switch*	$3+\frac{k^2}{4}$	$3+\frac{k}{2}$	$3+\frac{k^2}{4}$	$3+\frac{k}{2}$
Rules per core switch*	$1+\frac{k^3}{4}$	$1+k$	$1+k$	$1+k$
Transparent host migration	Network	Edge switch	Pod	Network

**Equations obtained assuming only one host in each server* ($h_s = 1$)

**k denotes the order (i.e., size) of the fat-tree topology*

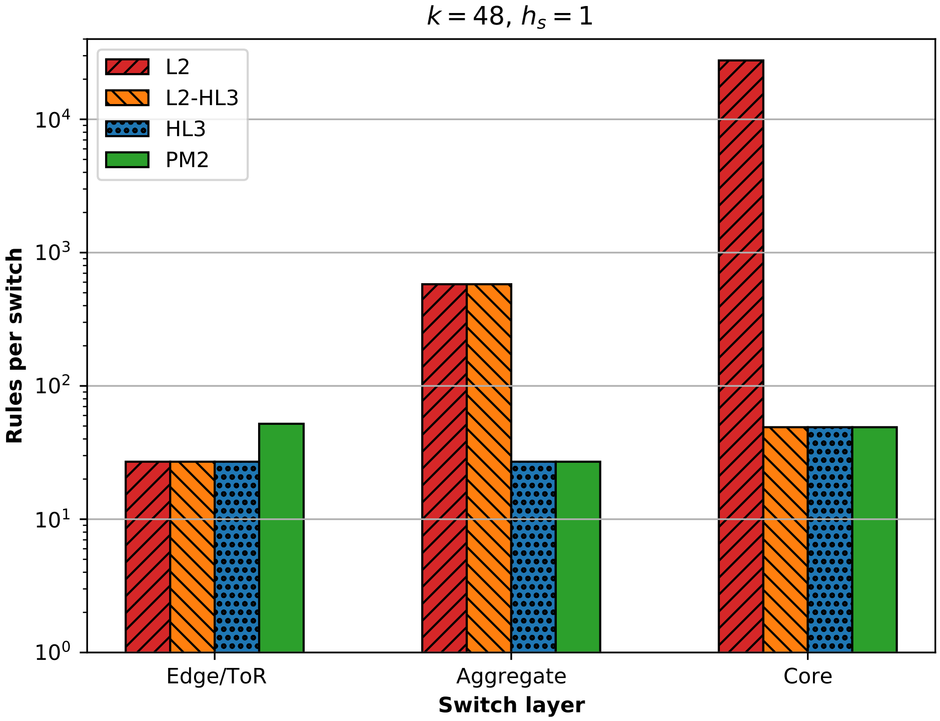


Figure 4.7: Number of rules installed per switch for each layer in a fat-tree topology with size $k = 48$ and only one host in each server ($h_s = 1$)

per aggregate switch. Lastly, PM2 installs a little more rules per edge switch than the other approaches. This is because PM2 requires the MAC-PMAC address parsing. Overall, PM2 installs much fewer rules than L2 and L2-HL3. In contrast, PM2 installs a little more rules than HL3.

However, as described in Table 4.2, HL3 limits the transparent host migration to the edge switch. Note that transparent host migration requires migrated hosts to keep their IPs since services running on other hosts might depend on it. As HL3 divides the whole network using IP prefixes, each edge switch supports only a specific range of

IPs. Therefore, using HL3, a host cannot be migrated to another edge switch without changing its IP. On the other hand, L2-HL3 extends the transparent host migration scope to the pod since it uses IP prefixes only for dividing the pods. Whereas, PM2 and L2 enable transparent host migration across the whole fat-tree network.

To sum up, PM2 installs much fewer rules than the other OpenFlow-feasible approaches that support transparent host migration across a fat-tree topology area larger than the same edge switch. Such reduction in the number of rules installed on switches allows decreasing the delay introduced to flows (mainly mice) traversing the fat-tree topology network.

4.2 Rescheduling of elephants in software-defined data center networks using deep incremental learning

Recall that SDDCN multipath routing mechanisms (see Section 2.3.3) deploy a controller that dynamically reschedules paths for identified elephants while relying on a default multipath routing algorithm, like PM2 (see Section 4.1 and ECMP [7], for handling the rest of the flows (potentially, mice). These SDDCN multipath routing mechanisms depend on flow detection methods that discriminate elephants from mice, either reactively by using thresholds or proactively by incorporating ML (see NELLY, introduced in Chapter 3). However, these flow detection methods merely indicate which flows are elephants (i.e., binary classification) but do not provide specific traffic characteristics (e.g., size, rate, duration) of such flows.

Therefore, most SDDCN multipath routing mechanisms handle all the elephants flows in the same way, that is, all with the same traffic characteristics. This is not suitable for covering the broad distribution of traffic characteristics of elephant flows in DCNs [8–11]. Only FlowSeer [33] perform a multiclassification for dividing the rate of the elephants into five classes. Although a five-class classification is much better than a binary classification, finer granularity traffic characteristics are desirable for improving the decisions of the multipath routing algorithm. Moreover, FlowSeer relies on sampling-based data collection, which increases the traffic overhead and the detection time while reducing the accuracy. As described in Section 3.3, FlowSeer reports to the controller 29% of mice as potential elephants, which can negatively affect the performance of the multipath routing algorithm.

In this section, we introduce a flow rescheduling method denominated intelligent Rescheduler of IDentified Elephants (iRIDE), which applies incremental learning at the controller-side of SDDCNs for predicting traffic characteristics of flows identified as elephants to compute and install the best path per flow across the network. Incremental learning allows iRIDE adapting to the variations in traffic characteristics and performing endless learning with limited memory resources. Quantitative evaluation demonstrates that iRIDE achieves low prediction error of flow rate and flow duration when using DNNs with regularization and dropout layers. Moreover, iRIDE enables intelligent elephant rescheduling algorithms that efficiently use the available bandwidth, generating higher throughput and shorter traffic completion time than conventional ECMP.

The remainder of this section is as follows. Section 4.2.1 introduces the architecture of iRIDE. Section 4.2.2 presents a quantitative evaluation of the prediction components in iRIDE using real data and incremental learning algorithms in both batch and incremental settings. Section 4.2.3 describes practical heuristics for implementing two rescheduling algorithms for iRIDE. Section 4.2.4 exposes the implementation of iRIDE, including the prediction and rescheduling components, for evaluating the networking performance. Section 4.2.5 discusses the networking performance results.

4.2.1 Architecture of iRIDE

Figure 4.8 introduces iRIDE, a flow rescheduling method at the controller-side of SDDCNs that applies incremental learning for predicting traffic characteristics of elephant flows to compute and install the best path per elephant flow across the network. iRIDE relays on flow detection methods, such as NELLY, for sending marked packets that identify elephant flows traversing the network.

As illustrated in Figure 4.8, iRIDE consists of five modules: *Catcher*, *Predictor*, *Rescheduler*, *Monitor*, and *Trainer*. For the sake of readability, Table 4.3 lists and describes the symbols defined in the architecture of iRIDE. The Catcher installs rules on edge switches (a.k.a. ToR) that forward the marked packets to the controller (i.e., catching rules). This installation can be conducted once the controller knows a switch belongs to the edge layer. The Catcher can receive this information from a *topology discovery* application (see Section 4.1). The catching rule can look for a predefined value in a code point header field supported by SDN switches. For example, OpenFlow switches support matching in two code point header fields. The first of these is the 6-bit DSCP field of the IPv4 header. This DSCP reserves a code point space for experimental and local usage (i.e., $****11$, where $*$ is 0 or 1). The second is the 3-bit 802.1Q PCP field of the Ethernet header. In practice, iRIDE can rely on either one of these fields, since it is improbable that a data center use both DSCP and PCP simultaneously [29].

As soon as the catching rule matches a marked packet in an edge switch, the marked packet is sent to the controller. The catcher receives the marked packet and extracts the flow information from it, commonly, the header (e.g., 5-tuple) to identify the elephant flow. This marked packet might also include the size and IAT of the first N flow packets that flow detection methods, such as NELLY, usually collect for classifying flows as mice and elephants (see Chapter 3). Note that NELLY requires a small modification in the Marker module of the Analyzer subsystem (see Section 3.1.1) to be able to communicate this extra information to iRIDE. Instead of marking the packets

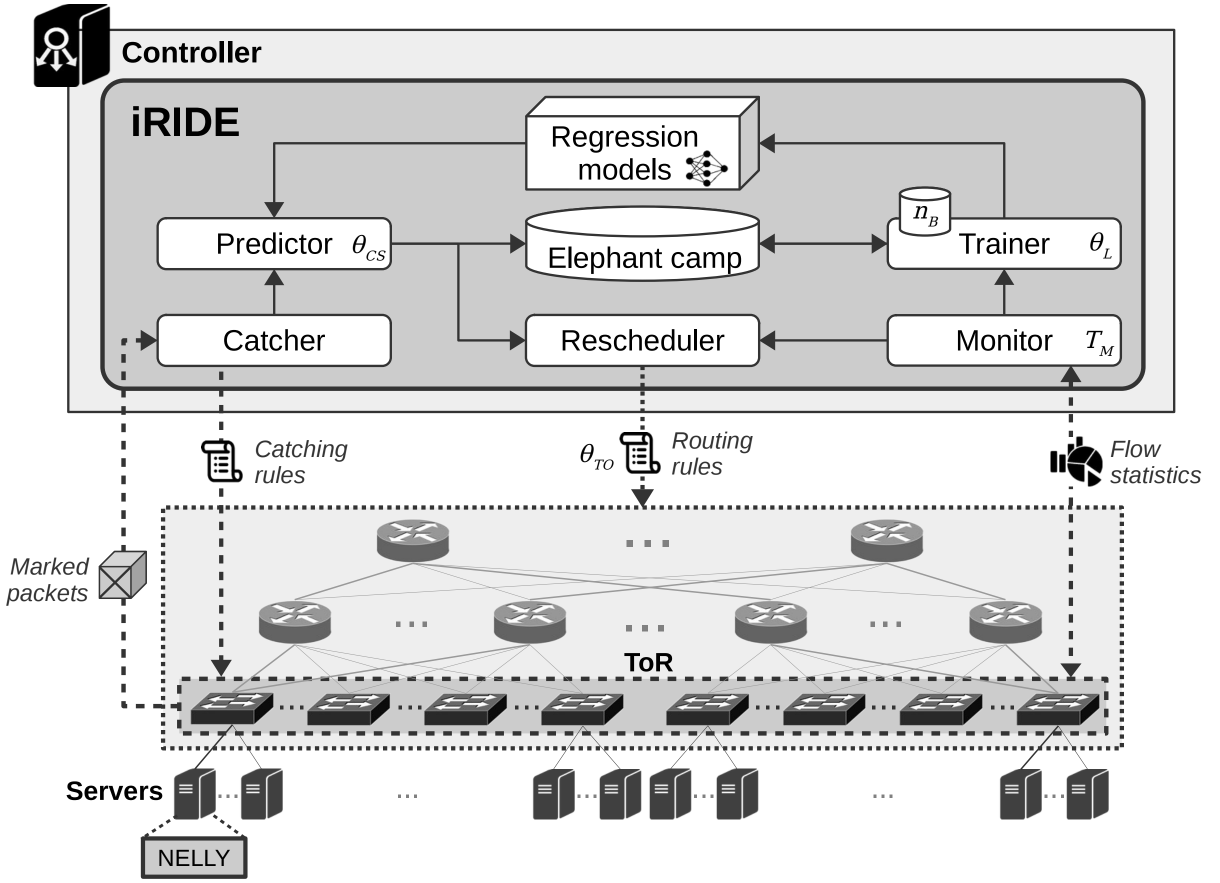


Figure 4.8: Architecture of iRIDE

Table 4.3: Symbols in the architecture of iRIDE

Symbol	Name	Description
θ_{CS}	Cold-start threshold	Value (e.g., number of training instances) above which the Predictor uses the regression models for predicting the traffic characteristics
θ_{TO}	Timeout threshold	Time limit above which switches acknowledge inactive flows as terminated for removing the corresponding routing rule
θ_L	Labeling threshold	Flow size limit below which the Trainer discards flows incorrectly classified as elephants
n_B	Mini-batch size	Number of elephant instances for training the regression models
T_M	Monitoring rate	Time interval at which the Monitor requests flow characteristics from the network

of flows classified as elephants, the Marker forwards the packets without changes and generates an extra marked packet for each elephant flow. Such an extra marked packet uses the same flow header (e.g., 5-tuple) and includes the size and IAT of the first N packets in its payload. In this case, the parameter M of the Marker would represent the number of extra marked packets generated for an elephant flow, thus enabling a trade-off between reliability and overhead. As M increases, the lesser the probability that the controller will miss elephant flows due to losses of extra marked packets in the SDDCN, but the more extra packets are sent to edge switches and to the controller. Once the controller has installed a higher priority routing rule for handling a specific elephant flow across the SDDCN, the subsequent extra marked packets of this flow are not required but they are still forwarded to the controller, which increases traffic overhead.

After the Catcher extracts the flow information from marked packets, it passes that data to the Predictor, which uses the regression models to predict the rate and duration of the flows identified by the marked packets. If the flow information includes the size and IAT of the first N packets, the Predictor's configuration can be extended by including a cold start threshold (θ_{CS}) that defines if estimating or predicting the flow traffic characteristics (i.e., rate and duration). For example, if the number of instances (i.e., flows) used to train the model is less than θ_{CS}, the Predictor computes the rate using the first N packets data and assigns a default value to the duration. When the number of trains reaches θ_{CS}, the Predictor starts using the regression models to predict the traffic characteristics. θ_{CS} allows warming up the regression models to avoid predictions about which they have not yet gathered sufficient information.

The Predictor stores all the flow information in the *elephant camp* (a temporal repository for elephant flows) and communicates the predicted traffic characteristics to the Rescheduler. The Rescheduler then can install specific routing rules per elephant flow across the network based on a path computed by a rescheduling algorithm that uses the predicted traffic characteristics. Section 4.2.3 describes two rescheduling algorithms that use either the predicted flow rate or flow duration to compute the path and install the routing rules for rerouting elephants in a fat-tree DCN topology.

Each routing rule includes a threshold timeout (θ_{TO}) that instructs SDN-enabled switches to remove the routing rule as soon as the corresponding flow has been inactive for θ_{TO} (i.e., the switch has not received a packet that matches the flow rule). Note that θ_{TO} is related to the flow definition in NELLY (see Section 3.1.1). Edge switches additionally include a mechanism that reports to the controller *flow statistics* of the removed routing rules due to time-out. Such a report must include the flow header, flow size, and flow duration of the routing rule. For example, OpenFlow allows inserting the flag OFPFF_SEND_FLOW_REM into the installed flow rules so when the OpenFlow switch removes one of them, it reports the removed flow rule to the controller, including the match header, removal cause (e.g., idle time-out), byte count, and duration in seconds. P4 enables a flexible data plane programming for easily implementing this reporting mechanism into P4 switches.

The Monitor receives the report of the removed routing rule from the edge switch and extracts the flow statistics: flow header, flow size, and flow duration. The Monitor then communicates these flow statistics to the Trainer, which discards those flows that were incorrectly marked as elephants, that is, flows whose size (e.g., byte count) is less than an elephant size threshold (θ_L). In NELLY, θ_L represents the labeling threshold for tagging flows as either mice or elephants (see Section 3.1.2). The Trainer computes the rate and duration (*ground truth*) of the non-discarded flows (i.e., true elephants) and use them to train the regression models. Each regression model maps online features (i.e., packet header, size, and IAT of the first N packets) onto the corresponding value (i.e., rate and duration). Recall that the Predictor relies on the regression models to predict flow traffic characteristics. The Trainer avoids increasing memory consumption in iRIDE by removing all timed-out flows (both discarded and used for training) from the elephant camp. Instead of using only one elephant instance, the Trainer might hold a mini-batch of elephant instances of size n_B for training the regression models.

Since flows represent continuous and dynamic data streams, the Trainer uses an incremental learning algorithm for building the regression models. Incremental learning enables updating the regression models as the Trainer receives timed-out flows over time, rather than retraining from the beginning (see Section 2.2.1).

Note that the Rescheduler operates depending on the predicted traffic characteristics that the Predictor communicates. However, the Monitor might also support the Rescheduler by implementing a mechanism that frequently requests flow characteristics (e.g., rate) from the network. This mechanism should keep a low traffic overhead. Therefore, every T_M, the Monitor requests traffic characteristics from elephant flows from edge switches and passes that information to the Rescheduler. For example, the Monitor can compute the rate of each requested elephant flow and pass it to the Rescheduler to have an updated view of the network traffic for taking routing decisions.

4.2.2 Prediction

This section presents the evaluation of the ML modules in iRIDE (i.e., the Predictor and the Trainer) in relation to prediction accuracy and time by using real packet traces and incremental learning algorithms. We used both batch and incremental settings for evaluating these learning algorithms. Furthermore, we used the generic approach for designing ML-based solutions in networking (see Figure 2.4) to describe and conduct this evaluation: data collection, feature engineering, establishing the ground truth, model validation, and model learning.

Datasets

We reused the two datasets [262], UNI1 and UNI2, generated for evaluating NELLY (see Section 3.2.1). We selected the datasets with BinNom-header as they enabled the best performance of the majority of the algorithms (see Section 3.2.4). BinNom-header provides a total of 117 online features. Moreover, since iRIDE only reschedules flows marked as elephants, we removed those flows smaller than θ_L = 100 kB from both datasets. Therefore, the newly generated UNI1 and UNI2 datasets consisted of approximately 8,500 and 20,000 flows, respectively.

The new datasets included the following flow information: start time, end time, 5-tuple header, size and IAT of the first 7 packets, as well as flow size. The start and end times allowed computing the duration of each flow. The 5-tuple header and the size and IAT of the first 7 packets represented the online features for the two regression models: flow rate and flow duration. The flow size was divided by the duration of the flow to compute its rate. The rate and duration of each flow represent the target value to predict and provide the ground truth for learning and validating the corresponding regression model.

To complement feature engineering, we converted to negative one (-1) the binary zero (0) values from the online features corresponding to the 5-tuple header since some ML techniques (particularly, NNs) perform better when the input values are centered around zero rather than ranging between 0 and 1 [283, 284]. We also transformed the numeric values in the online features (i.e., size and IAT of first 7 packets) using different feature scaling methods, as discussed later in the experiment setup.

Accuracy metrics

We used two accuracy metrics commonly used in the literature to report the performance of regression models: the coefficient of determination (R^2) and the Root Mean Square Error (RMSE). R^2 provides a goodness-of-fit score that measures how well the regression models fit the observed data (i.e., ground truth) [285]. We rely on the common R^2 definition that uses the first equation of Kvålseth (see Equation 4.7), which provides scores usually between 0 and 1. As the R^2 score gets closer to 1, the better the regression predictions ($\hat{y}$) approximate to the observed values (y). Note that R^2 scores below 0 might occur, which represent that the regression fit performs worse than a horizontal line [286]. R^2 enables comparisons across different types of data as it does not depend on the scale of the values. However, R^2 by itself cannot indicate if a regression model is adequate. Therefore, we used RMSE to complement the R^2 scores.

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \text{ where } \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{4.7}$$

As shown in Equation 4.8, RMSE measures the differences between the values predicted by the regression models ($\hat{y}$) and the observed values (y). RMSE provides a non-negative value that expresses a higher accuracy as the error gets smaller. We preferred RMSE over other similar error metrics often used to gauge the accuracy of regression models, such as MAE and MSE [286]. The three error metrics disregard the direction of under- and over-estimations in the predictions. Moreover, unlike R^2, these metrics depend on the scale of the data so they cannot be used to compare the accuracy of regression models working with different types of data. However, MSE and RMSE are more useful than MAE for heavily penalizing large errors and outliers. Additionally, in contrast to MSE, RMSE expresses the standard deviation of the error, which is in the same units as the quantity being predicted. Our RMSE results display Megabits per second (Mbps) for flow rate and seconds (s) for flow duration.

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}} \tag{4.8}$$

At the end of the evaluation process, we also analyzed the residual plots of the selected regression models to check no bias nor problematic patterns exist in the residuals.

Experiment setup

Incremental learning algorithms are commonly evaluated using the interleaved test-then-train approach [266]. However, for selecting the data preprocessing techniques (i.e., feature scaling, target transformation, imbalance correction) and the regression model (i.e., learning algorithm and hyperparameter tuning), we followed the common $60/20/20$% batch decomposition for dividing the datasets into training, validation, and test sets, respectively (see Section 2.2.1). This batch decomposition allows using the validation set for selecting the best model and evaluate it on the test set to get an unbiased estimate of the model's performance [287]. Since our datasets are in the order of the tens of thousands, we applied the batch holdout method for validation and testing. Note the result from the test set represents the optimal performance to which the model would tend when using an incremental evaluation method (either holdout or interleaved test-then-train) [288].

Feature scaling is an essential data preprocessing step for ML algorithms that compute a distance function between input features, such as kNN [283]. Distance functions (e.g., Euclidean distance) depend heavily on the variability of the features and are biased towards numerically larger values. For example, the online features in our datasets contain binary values [-1, 1] for the 5-tuple packet header and numeric values for the size and IAT of the first 7 packets. The packet size ranges from 60 bytes up to 1522 bytes, whereas the packet IAT can go from a microsecond up to five million of microseconds (i.e., $\theta_{TO} = 5$s). Therefore, our non-scaled numeric values (particularly, large IATs) could bias ML algorithms based on distance functions. Moreover, feature scaling for ML algorithms using gradient descent, such as NNs, might not be strictly required but can make their training to converge faster and with less chances of sticking in a local optima [283]. Conversely, DT algorithms (e.g., Hoeffding trees) are insensitive to feature scaling.

We analyzed different feature scaling methods on the numeric values of the online features [289]. The *min-max zero-center scaler* (see Equation 4.9) is a simple method that rescales the features to the range [-1, 1] by using the minimum and maximum values of each feature. In our datasets, we defined 60 and 1522 (bytes) as the minimum and maximum values for the packet size, as well as one and five million (microseconds) for the packet IAT. The *standard scaler*, also called Z-score normalization (see Equation 4.10), removes the mean and scales the values to unit variance. The *robust scaler* removes the median and scales the data according to the interquartile range (i.e., range between the first and third quartiles). The *quantile transformer* provides a non-linear transformation that maps the features to follow either a *uniform* or a *normal* distribution. The *power transformer* applies either the *Box-Cox* transform (see Equation 4.11) or the *Yeo-Johnson* transform (see Equation 4.12). Both non-linear transformations use the maximum likelihood to estimate the optimal parameter λ that maps data to follow a Gaussian-like distribution. The *normalizer* performs the Euclidean norm (a.k.a. L2 norm) to scale features individually to unit form.

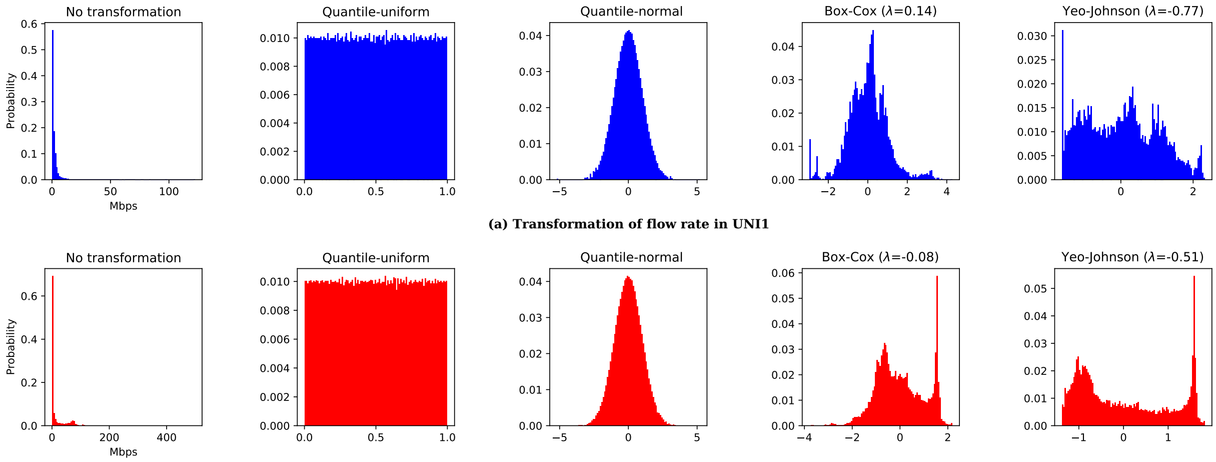
$$X_i' = \frac{2 \cdot (X_i - min(X))}{max(X) - min(X)} - 1 \tag{4.9}$$

$$X_i' = \frac{X_i - \bar{X}}{\sigma}, \text{ where } \bar{X} \text{ is the mean and } \sigma \text{ is the standard deviation} \tag{4.10}$$

$$X_i^{(\lambda)} = \begin{cases} \frac{X_i^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \ln(X_i) & \lambda = 0 \end{cases} \tag{4.11}$$

$$X_i^{(\lambda)} = \begin{cases} \frac{(X_i+1)^\lambda - 1}{\lambda} & \lambda \neq 0, X_i \geq 0 \\ \ln(X_i + 1) & \lambda = 0, X_i \geq 0 \\ -\frac{(-X_i+1)^(2-\lambda) - 1}{2-\lambda} & \lambda \neq 2, X_i < 0 \\ -\ln(-X_i + 1) & \lambda = 2, X_i < 0 \end{cases} \tag{4.12}$$

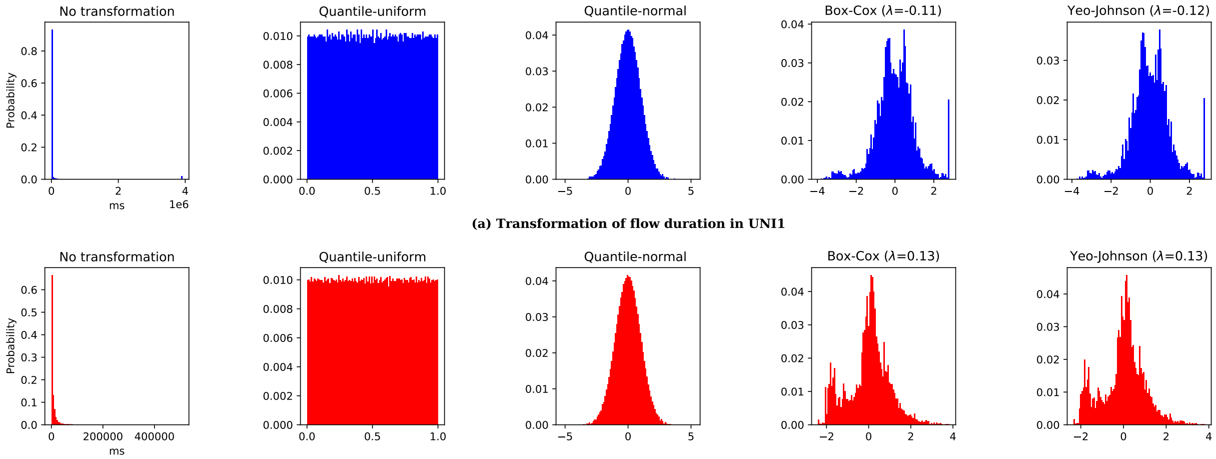
We also analyzed different target variable transformations by applying some of these feature scaling methods: *quantile-uniform*, *quantile-normal*, *Box-Cox*, and *Yeo-Jhonson*. Figures 4.9 and 4.10 depict these transformations on the two target variables, flow rate and flow duration, respectively, in both datasets, UNI1 and UNI2. Note both target variables present an imbalanced domain when no transformation has been applied to the data. Therefore, we further included the Synthetic Minority Oversampling technique for regression with Gaussian Noise (SMOGN) [290] into our experiments for analyzing imbalance correction (i.e., under-sampling and over-sampling) in our datasets. SMOGN combines three methods proposed for addressing regression imbalance: random under-sampling [291], Synthetic Minority Over-sampling TEchnique for Regression (SMOTER) [292], and introduction of Gaussian Noise (GN) [293]. Random under-sampling enables removing less interesting instances, whereas SMOTER and GN introduction generate new synthetic data from close and distant instances, respectively.



(a) Transformation of flow rate in UNI1

(b) Transformation of flow rate in UNI2

Figure 4.9: Transformations of flow rate in UNI1 and UNI2



(a) Transformation of flow duration in UNI1

(b) Transformation of flow duration in UNI2

Figure 4.10: Transformations of flow duration in UNI1 and UNI2

Finally, we used the interleaved test-then-train approach [266] for evaluating the selected regression model in an incremental setting. Similar to NELLY (see Section 3.2.3, the prediction of flow characteristics (i.e., flow rate and flow duration) takes place at the flow start time, while retraining the regression models at the flow end time. Moreover, we analyzed different mini-batch sizes (n_B) for training the model incrementally.

Performance analysis

To determine the regression model with the best accuracy performance, first, we used the batch decomposition to evaluate different learning algorithms that can operate in an incremental approach. Three regression algorithms from scikit-learn [294]: Stochastic Gradient Descent (SGD) for Ordinary Least Squares (OLS), Passive-Aggressive (PA), and Multi-Layer Perceptron (MLP) (i.e., NN); plus four regression algorithms from scikit-multiflow [295]: k-Nearest Neighbors (kNN), Hoeffding tree, Hoeffding Adaptive Tree (HAT), and Adaptive Random Forest (ARF). The algorithms were executed with their default settings and without previous model initialization. We only modified the MLP structure to two hidden layers: the first one with 175 units and the second one with 117 units (i.e., respectively, $3/2$ and 1 times the number of input units, which is the number of online features). Moreover, we evaluated all the feature scaling methods and target transformations (both explained in the experiment setup) for each learning algorithm.

For the sake of brevity, Table 4.4 presents the learning algorithm and data preprocessing techniques that achieved the best accuracy performance (i.e., the highest R^2 and the lowest RMSE) for predicting each target variable (i.e., flow rate and flow duration) in the training and validation sets of UNI1 and UNI2. The results show that MLP achieved the best accuracy for predicting the two target variables in both evaluation sets of UNI1 and UNI2, though the associated feature scaling methods and target transformations varied. Note Table 4.4 presents the second best accuracy performance for predicting both target variables in the validation set of UNI1. Although kNN achieved the best accuracy in the validation set, its accuracy results performed really bad in the training set. In fact, kNN presented negative R^2 values in the training set, representing that the regression fit performed worse than a horizontal line. Therefore, the kNN results in the validation set of UNI1 were not reliable.

The results in Table 4.4 also depict that MLP fits better the rate (i.e., higher R^2) in UNI1 than in UNI2, achieving a lower error (i.e., RMSE) for the first one. In contrast, although MLP fits better the duration in UNI1 than in UNI2, the error in the first one is higher than in the other. This is because UNI1 presents some flow duration values that greatly differ from the others, which are highly penalized by the RMSE metric. Therefore, hereinafter, we make decisions based on RMSE results.

We further applied SMOGN to our datasets for evaluating the effect of imbalance correction (i.e., under-sampling and over-sampling) on the accuracy performance. In

Table 4.4: Learning algorithm and data preprocessing with the best accuracy performance in a batch learning setting

Dataset	Target variable	Evaluation set	Learning algorithm	Feature scaling	Target transform	R^2	RMSE*
UNI1	Rate	Train	MLP	Box-Cox	None	0.9417	1.42
		Validation†	MLP	Robust	Yeo-Johnson	0.3123	5.61
	Duration	Train	MLP	Quantile-normal	Quantile-normal	0.9678	85.64
		Validation†	MLP	Min-max	None	0.9119	197.92
UNI2	Rate	Train	MLP	Quantile-normal	Quantile-normal	0.5424	23.07
		Validation	MLP	Robust	None	0.4367	24.81
	Duration	Train	MLP	Quantile-normal	Box-Cox	0.7378	14.28
		Validation	MLP	Min-max	Quantile-normal	0.5215	19.34

* *RMSE units depend on target variable: Megabits per second (Mbps) for rate and seconds (s) for duration*

† *Showing the second best result, disregarding kNN*

this experiment, we only used MLP as the learning algorithm as it performed the best in the results depicted in Table 4.4. However, we kept analyzing all the feature scaling methods and target transformations. MLP preserved the same NN structure (i.e., two hidden layers of 175 and 117 units), as well as the rest of default settings and no previous model initialization.

Table 4.5 shows the data preprocessing techniques (imbalance correction, feature scaling, and target transformation) with which MLP achieved the best accuracy performance in the validation set for predicting both target variables in UNI1 and UNI2. To summarize, for predicting the rate in UNI1, MLP achieved the best accuracy by using the under-sampling imbalance corrective, the robust feature scaler, and the Yeo-Johnson target transformation. Whereas, for the duration in UNI1, MLP performed the best by using over-sampling, the min-max scaler, and no target transformation. On the other hand, for predicting both target variables in UNI2, MLP required no imbalance corrective nor target transformation but used the feature normalizer to achieve the best performance. However, many data preprocessing techniques that enabled the best accuracy have been specially designed for batch learning (i.e., operate with the whole training set), including under-sampling and over-sampling, robust scaler, normalizer, and Yeo-Johnson transformer. Implementing these techniques in an incremental setting might require huge memory resources and it is still an open research challenge [296].

Therefore, we evaluated a *feasible incremental* approach by combining MLP with those data preprocessing techniques that can operate in an incremental setting, namely, min-max feature scaler, no target transformation, and no imbalance correction. Note the min-max scaler only requires defining the minimum and maximum values of the features. In our datasets, 60 and 1522 (bytes) as the minimum and maximum values for the packet size, as well as one and five million (microseconds) for the packet IAT. Table 4.6 presents the accuracy performance of MLP using the feasible incremental

Table 4.5: Data preprocessing with the best accuracy performance for MLP in a batch learning setting

Dataset	Target variable	Imbalance corrective	Feature scaling	Target transform	Training		Validation	
					R^2	RMSE[*]	R^2	RMSE[*]
UNI1	Rate	None[†]	Robust	None	0.9722	1.12	0.4849	4.27
		Under-sampling	Robust	Yeo-Johnson	0.9745	1.62	0.4944	4.23
		Over-sampling	Yeo-Johnson	None	0.9801	1.75	0.4649	4.35
	Duration	None[†]	Standard	None	0.9979	24.82	0.8899	192.24
		Under-sampling	Min-max	None	0.9976	44.89	0.8885	193.46
		Over-sampling	Min-max	None	0.9978	49.58	0.9023	181.14
UNI2	Rate	None[†]	Normalizer	None	0.5167	23.45	0.4308	25.28
		Under-sampling	Normalizer	Yeo-Johnson	0.7081	24.72	0.3971	26.01
		Over-sampling	Robust	None	0.7532	23.09	0.4139	25.65
	Duration	None[†]	Normalizer	None	0.6335	17.01	0.4936	21.77
		Under-sampling	Min-max	None	0.6476	22.05	0.4647	22.38
		Over-sampling	Min-max	Yeo-Johnson	0.6786	22.15	0.4522	22.64

[*] *RMSE units depend on target variable: Megabits per second (Mbps) for rate and seconds (s) for duration*

[†] *Original dataset, with no under-sampling nor over-sampling*

data preprocessing techniques. As expected, MLP achieved worse validation errors in comparison with using the best data preprocessing techniques (see Table 4.5). RMSE in the validation set increased by 0.6 (Mbps) and 0.3 (seconds) when predicting the flow rate in both datasets and the flow duration in UNI2, respectively. Moreover, the flow duration prediction in UNI1 represents the worst case, incrementing the validation RMSE by 21 (seconds). Nevertheless, MLP achieved similar RMSE values in the training set, opening an opportunity to improve the validation error.

Table 4.6: Accuracy performance of MLP using feasible incremental data preprocessing techniques in a batch learning setting

Dataset	Target variable	Training		Validation	
		R^2	RMSE[*]	R^2	RMSE[*]
UNI1	Rate	0.9532	1.45	0.3259	4.88
	Duration	0.9969	29.77	0.8778	202.57
UNI2	Rate	0.4990	23.88	0.4032	25.88
	Duration	0.5832	18.14	0.4781	22.09

[*] *RMSE units depend on target variable: Megabits per second (Mbps) for rate and seconds (s) for duration*

Aiming at improving the accuracy performance on the validation set, first, we focused on building NN structures that reduce the training errors by using Tensorflow [297] and Keras [298]. We set typical hyperparameters for DNNs [299–301], including the Rectified Linear Unit (ReLU) [302] as the activation function in the units of the hidden layers, He normal [303] to initialize the NN weights, Adam optimization [304] to improve SGD, and the default mini-batch size of 32 instances per gradient update [305]. For tuning the NN structure (see Figure 4.11), we varied the number of hidden layers (L_h) from one up to ten in steps of one (i.e., $L_h \in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$), and the number of units per each hidden layer ($n^{[l_h]}\ \forall\ l_h \in L_H$) from 15 up to 600 in steps of specific multiples of 15 (i.e., $n^{[l_h]} \in [15, 30, 60, 120, 240, 360, 480, 600]$). Note the data preprocessing techniques remained as the feasible incremental approach (i.e., min-max feature scaler, no target transformation, and no imbalance correction).

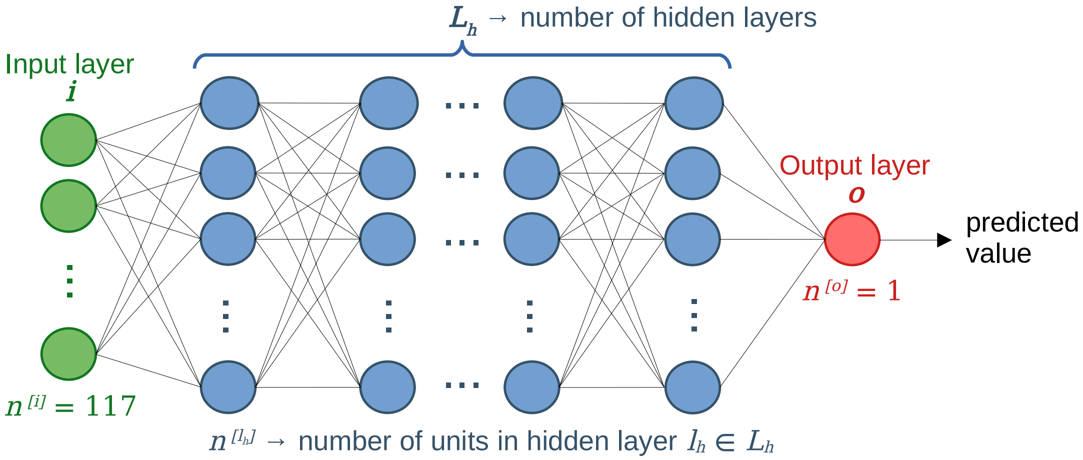

L_h → number of hidden layers
Input layer
i
$n^{[i]} = 117$
Output layer
o
$n^{[o]} = 1$
predicted value
$n^{[l_h]}$ → number of units in hidden layer $l_h \in L_h$

Figure 4.11: NN structure

Table 4.7 describes the NN structures that achieved the lowest RMSE values in the training sets of both datasets, UNI1 and UNI2, for predicting the two target variables. In general, the results show that for reducing the training errors, NN requires deep structures (i.e., DNN) from five up to nine hidden layers (L_h) and from 360 up to 600 units per each hidden layer ($n^{[l_h]}$). In comparison to the results of MLP using the feasible incremental data preprocessing techniques (see Table 4.6, the reduction of the training RMSE was good for predicting the rate and duration in UNI1 ($\sim$33% and $\sim$24%, respectively), moderate for the duration in UNI2 ($\sim$11%), and minimal for the rate in UNI2 ($\sim$4.6%).

The problem about focusing on reducing the training error is that it causes overfitting [306], which means that the model would not be able to generalize in unseen data (i.e., poor accuracy performance on the validation and test sets). To tackle this problem, we incorporated a combination of two regularization methods to reduce the error on the validation sets. First, L2 regularization [307] (see Equation 4.13), for which we varied the regularization parameter (λ) from 0 up to 0.1 in steps of 1 thousand (i.e.,

Table 4.7: DNN structures with the lowest training errors in a batch learning setting

Dataset	**Target variable**	**Hidden layers (L_h)**	**Hidden units ($n^{[l_h]}$)**	**Training RMSE***
UNI1	Rate	5	480	0.97
	Duration	8	360	22.53
UNI2	Rate	8	600	22.78
	Duration	9	480	16.22

** RMSE units depend on target variable: Megabits per second (Mbps) for rate and seconds (s) for duration*

$\lambda \in [0, 10^{-5}, 10^{-4}, 10^{-3}, 0.01, 0.1]$). Second, a dropout regularization layer [308] for each hidden layer, for which we varied the dropout rate from 0% up to 75% in steps of 25% (i.e., $[0, 25, 50, 75]\%$).

$$L2 = \frac{\lambda}{2m} \parallel w^{[l]} \parallel_F^2 \quad \text{, where } \lambda \text{ is the regularization parameter, } m \text{ is the number of training instances, and } \parallel w^{[l]} \parallel_F^2 \text{ is the squared Frobenius norm of the weights in layer } l \tag{4.13}$$

Table 4.8 depicts the DNN structures and regularization methods (i.e., L2 and dropout layers) that achieved the lowest RMSE values in the validation sets of both datasets, UNI1 and UNI2, for predicting the two target variables. For predicting the rate, the results show the DNN structures achieved the lowest validation errors by using a combination of the two regularization methods, with $\lambda = 0.1$ and a dropout rate of 50% for UNI1, and $\lambda = 10^{-4}$ and a dropout rate of 25% for UNI2. Whereas, for predicting the duration, using only L2 regularization with $\lambda = 10^{-5}$ provided the best validation results. In comparison to the results of MLP with the feasible incremental data preprocessing techniques (see Table 4.6), the trained DNNs reduced the validation RMSE values, except for the rate in UNI2, where no improvement nor degradation was observed. In fact, in comparison to the results of MLP with the best data preprocessing techniques (see Table 4.5), the DNNs reduced the validation errors for predicting the duration in UNI1 and UNI2. Although the validation RMSE for predicting the rate in UNI1 is higher for the trained DNN than for the best MLP, this error is still lower than for MLP with the feasible incremental data preprocessing techniques.

To conclude the evaluation in a batch learning setting, we evaluated the obtained DNN regression models in the test set of both datasets, UNI1 and UNI2, for reporting an unbiased estimate of the model's accuracy performance when predicting the flow rate and the flow duration (see Table 4.8).

Table 4.8: DNN structures and regularization methods with the lowest validation errors in a batch learning setting

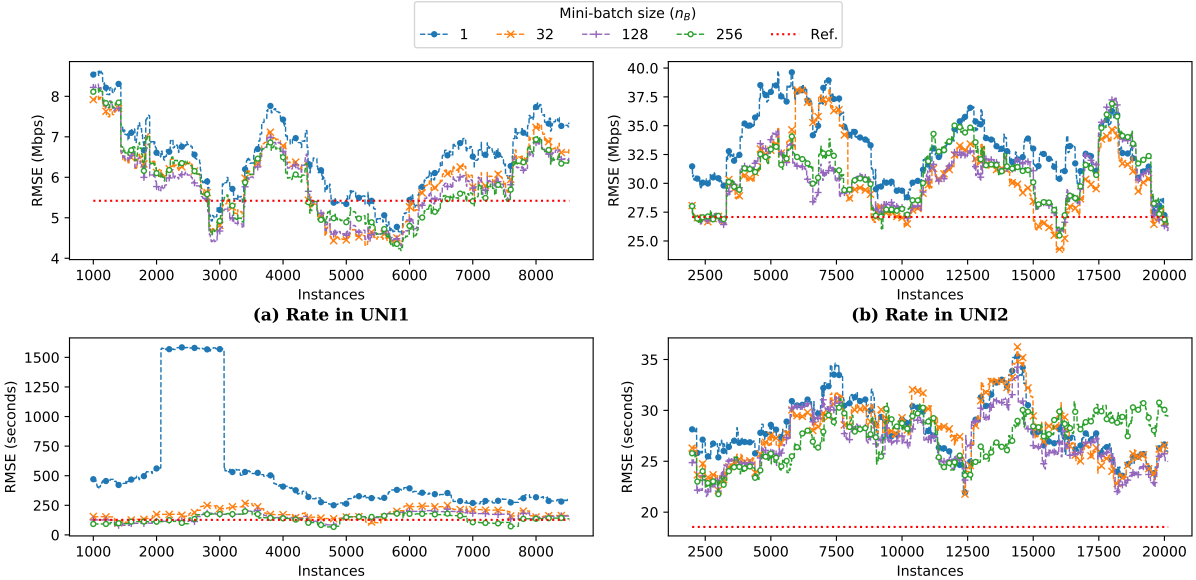
Dataset	Target variable	Structure*	Regularization parameter (λ)	Dropout rate	Training RMSE†	Validation RMSE†	Testing RMSE†
UNI1	Rate	$L_h = 5, n^{[l_h]} = 480$	0.1	50%	4.61	4.34	5.42
	Duration	$L_h = 8, n^{[l_h]} = 360$	10^{-5}	0%	30.14	173.97	126.89
UNI2	Rate	$L_h = 8, n^{[l_h]} = 600$	10^{-4}	25%	25.51	25.88	27.08
	Duration	$L_h = 8, n^{[l_h]} = 360$	10^{-5}	0%	17.51	21.52	18.55

* *Defines the number of hidden layers (L_h) and the number of units per each hidden layer ($n^{[l_h]}$)*

† *RMSE units depend on target variable: Megabits per second (Mbps) for rate and seconds (s) for duration*

Continuing our prediction evaluation, we used the interleaved test-then-train approach for evaluating the tuned DNN regression models in an incremental learning setting. Figure 4.12 shows the mean RMSE over a sliding window of approximately 10% of the instances (1000 in UNI1 and 2000 in UNI2). Note that we also analyzed mini-batch sizes (n_B) other than one, varying the number of training instances from 8 up to 256 in doubling steps (i.e., $n_B \in [1, 8, 16, 32, 64, 128, 256]$). Moreover, the reference dotted-red line (Ref.) in the figure represents the RMSE values reported for the test sets in the batch learning evaluation (see Table 4.8). The results show, first, the DNN regression models incrementally tend to the test errors from the batch learning evaluation, except for the duration in UNI2. Second, the models suffer from higher error as new traffic characteristics appear in the data. However, the DNN regression models incrementally adapt to new traffic characteristics, lowering the RMSE values back. Third, using a large n_B generally provides no significant improvement over time. In fact, using $n_B =$ 256 for predicting the duration in UNI2 might cause overfitting, producing a higher error than using a smaller n_B. This is not the case for the duration in UNI1, which greatly benefits from using $n_B > 1$ to reduce the negative impact of the flow duration outliers on the prediction error.

Figure 4.13 corroborates the impact of n_B on the prediction errors by presenting the mean RMSE over different values of n_B for the tuned DNN regression models. Similarly, the dotted-striped-red bar represents the RMSE values reported for the test sets in the batch learning evaluation (see Table 4.8). The results show that using $n_B > 1$ for predicting the rate in both UNI1 and UNI2 reduces RMSE by a small amount (maximum 9.5%), achieving a minimum value that is around 11% over the test errors from the batch learning evaluation. Note the RMSE reduction is less significant for $n_B > 32$. Regarding the duration in UNI1, using $n_B > 1$ greatly reduces the prediction error by a maximum of 79%, achieving the lowest RMSE when using $n_B = 256$, which is only 7% over the test error from the batch learning evaluation. Similarly, the error reduction is minor for $n_B > 32$. In contrast, for predicting the duration in UNI2, the results show that using



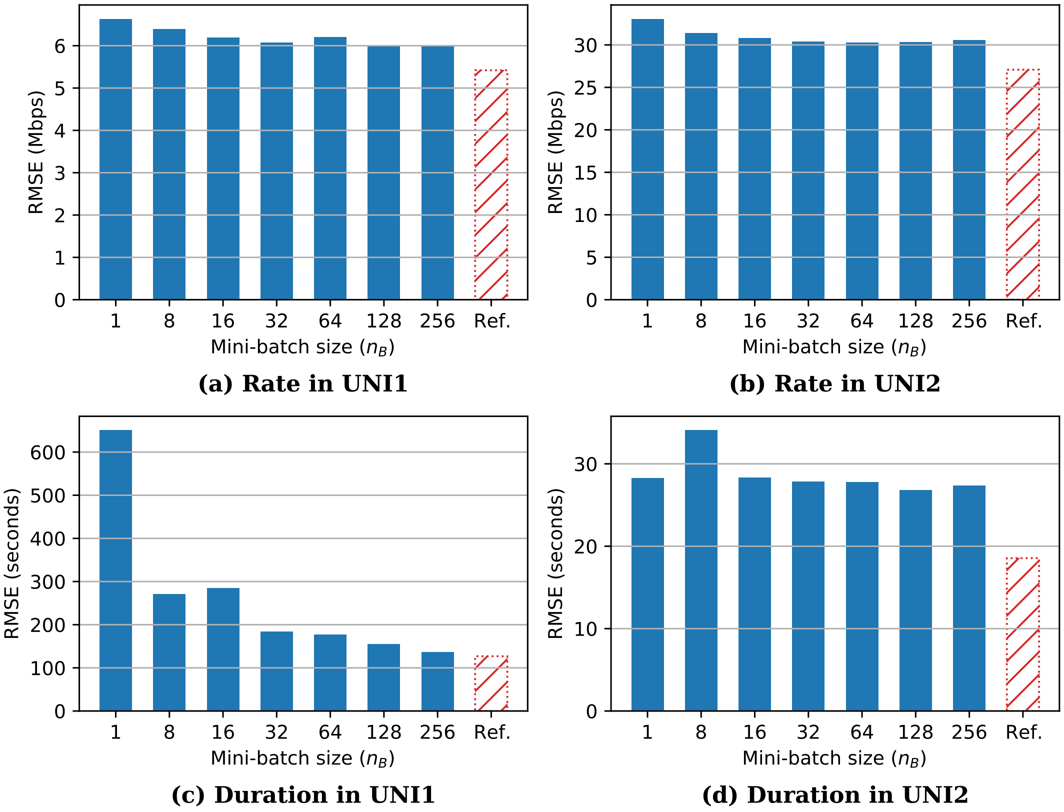
(a) Rate in UNI1 **(b) Rate in UNI2**

(c) Duration in UNI1 **(d) Duration in UNI2**

Figure 4.12: Mean RMSE over a sliding window of instances for DNN regression models with different mini-batch sizes (n_B) in an incremental learning setting

$n_B > 1$ provides no perceptible RMSE reduction. In fact, when $n_B = 8$, the RMSE grows up to 27% over the rest of n_B values. Note the lowest RMSE is 44% over the test error from the batch learning evaluation.

Lastly, Figure 4.14 depicts the mean prediction time per flow over different values of n_B for the tuned DNN regression models. Note that the time for predicting the two target variables, rate and duration, in both datasets UNI1 and UNI2, achieves the minimum value when $n_B = 1$. This prediction time per flow grows as n_B increases, achieving a maximum value when $n_B = 32$, which is between 1.8 and 3.4 milliseconds (ms) over the lowest prediction time. Based on the results of both the prediction time per flow and the mean RMSE (see Figure 4.13), we recommend using only one training instance at a time (i.e., $n_B = 1$) when predicting the rate in UNI1 and UNI2 as the error reduction is small (up to 9.5%) in comparison to the increment of the prediction time (> 2 ms) when using $n_B > 1$. Similarly, stick to $n_B = 1$ when predicting the duration in UNI2 since no error reduction is perceptible when using $n_B > 1$ while the prediction time does increment by at least 2 ms. In contrast, for predicting the duration in UNI1, we recommend using $n_B = 256$ because the error is greatly reduced (up to 79%) while generating a prediction time per flow that is only 0.7 ms above the prediction time for $n_B = 8$.



(a) Rate in UNI1 **(b) Rate in UNI2**

(c) Duration in UNI1 **(d) Duration in UNI2**

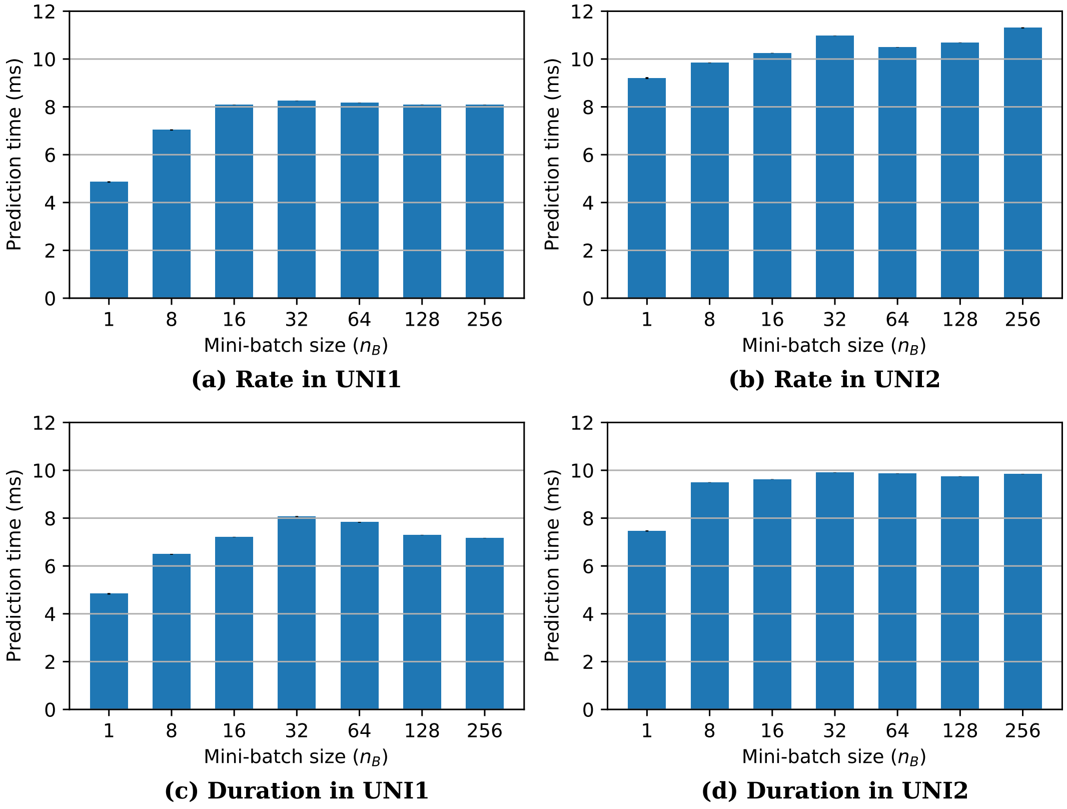
Figure 4.13: Mean RMSE over mini-batch sizes (n_B) for DNN regression models in an incremental learning setting

4.2.3 Rescheduling

As described in Section 4.8, the Rescheduler module of iRIDE implements a rescheduling algorithm that uses the predicted traffic characteristics, reported by the Predictor, to compute a path and install specific routing rules per elephant flow across the network. The problem about finding the path for different flows while not exceeding the bandwidth capacity of any link is known as Multi-Commodity Flow, which is NP-complete [26]. To the best of our knowledge, no polynomial time algorithm exists for simultaneous flow routing in realistic DCN topologies, such as the 3-tier fat-tree topology (i.e., 5-stage Clos network) [26]. Therefore, this section describes practical heuristics for implementing two rescheduling algorithms that consider the fat-tree DCN topology, as seminal related works have done (e.g., simulated annealing [26] and increasing first-fit [29]).

Least Congested (LC) path

In a fat-tree topology, multiple equal-cost paths exist between any pair of hosts not connected to the same edge switch. When the Rescheduler receives a flow identified



(a) Rate in UNI1

(b) Rate in UNI2

(c) Duration in UNI1

(d) Duration in UNI2

Figure 4.14: Mean prediction time per flow over mini-batch sizes (n_B) for DNN regression models in an incremental learning setting

as elephant, the Least Congested (LC) path algorithm linearly searches all the paths between the source and destination hosts to select the one whose link components carry the less traffic load. Note that only one path exists if the pair of hosts are connected to the same edge switch. The Rescheduler then places the flow on the selected path. First, the Rescheduler uses the predicted flow rate, reported by the Predictor, to reserve the bandwidth capacity for the flow on the links corresponding to the selected path. Second, the Rescheduler installs routing rules for bottom-up communication (i.e., from edge to aggregation and from aggregation to core) in the corresponding edge and aggregation switches. The top-down communication (i.e., from core to aggregation, from aggregation to edge, and from edge to destination host) relies on the default routing algorithm, such as PM2 (see Section 4.1). The Rescheduler then maintains the reserved bandwidth capacity for every link in the network to determine which paths carry the less traffic load for placing new flows identified as elephants. When receiving the notification from the Monitor that a flow has expired (i.e., a flow has been inactive for θ_{TO}), the Rescheduler clears the corresponding reservations of bandwidth capacity.

Worst-Fit Badwidth-Time-Fit (WF+BTF)

In contrast to LC, the Worst-Fit Badwidth-Time-Fit (WF+BTF) algorithm uses the two predicted traffic characteristics reported by the Predictor, namely, flow rate and flow duration. To do so, WF+BTF operates in two steps. First, when the Rescheduler receives a flow identified as elephant, WF+BTF linearly searches all the paths to find the one with the less traffic load whose link components can all accommodate the predicted rate of that flow. Note that this corresponds to the Worst-Fit (WF) algorithm from the bin packing problem [309]; WF places an item (flow) in a feasible bin (path) with most free space (bandwidth). When the network traffic load is light, finding such a path that can accommodate the predicted flow rate is likely to be easy. However, as the network traffic load grows and links become congested, WF does not guarantee that all flows identified as elephants will be placed in a path. When no path among the many existing ones between the pair of hosts can accommodate the predicted flow rate, WF+BTF moves to the second step, the Badwidth-Time-Fit (BTF) algorithm. BTF searches for the path with the maximum harmonic mean (see Equation 4.14) between the relative scores of the available bandwidth in the path (b_p) and the time the path would take to fit the predicted rate (t_p). BTF computes the relative scores b_p and t_p for each path by comparing the corresponding values of the path to the best values (i.e., the maximum free bandwidth and the minimum time to fit, respectively) among the many possible paths between the pair of hosts. Note that the time to fit the predicted flow rate is computed using the predicted duration of the flows.

$$\begin{aligned} H = \frac{2 \times b_p \times t_p}{b_p + t_p} \text{ , where } b_p &= \frac{\text{available bandwidth in path}}{\text{maximum available bandwidth among paths}} \\ t_p &= \frac{\text{minimum time to fit among paths}}{\text{time to fit in path}} \end{aligned} \tag{4.14}$$

4.2.4 Implementation

As depicted in Figure 4.15, we reused the Mininet VM deployed for PM2 (see Section 4.1.3) for running our custom fat-tree topology to evaluate iRIDE. However, instead of executing a simple ping test between the hosts as in PM2, we installed the traffic generator Tcpreplay 4.3.4 [310] on the Mininet VM for the hosts to inject real traffic into the network. Tcpreplay supports replaying network traffic captured in the format of PCAP files. Moreover, we installed Ifstat 1.1 [311] on the Mininet VM for capturing the activity of the switch interfaces in our fat-tree emulated network. Ifstat reports interface statistics such as the incoming and outgoing bandwidth traffic. Note that all the hosts in our fat-tree emulated network concurrently execute a Tcpreplay process for generating traffic, which requires more resources (around a logical processor per process) than a simple ping test. Therefore, we redeployed our Mininet VM on to a more powerful server at LRC than the one used for the proof of concept of PM2. The new server also runs Debian 8.11 server but in a 2.00 GHz Intel Xeon Processor ES-2660 machine, with 28 physical cores, 56 logical cores, and 226 GB RAM. Then, we increased the resources of the Mininet VM to 18 logical cores and 100 GB RAM.

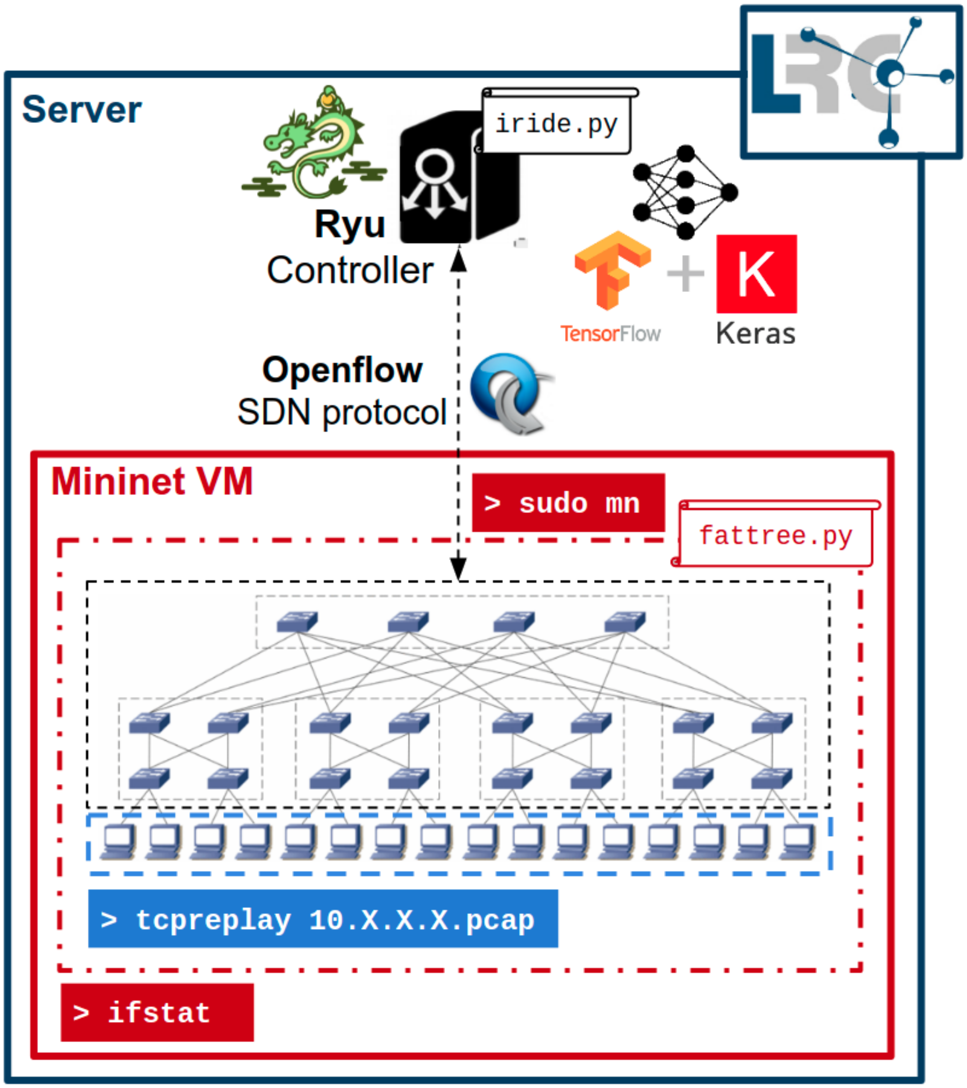


Figure 4.15: Implementation of iRIDE

Similar to PM2, we installed Ryu 4.34 [66] on the new server for deploying the OpenFlow controller that manages the OpenFlow switches. Furthermore, we installed Tensorflow 2.3.0 [297] and Keras 1.1.2 [298] for training (incrementally) and inference of DNNs. Using the Python libraries from Ryu, Tensorflow, and Keras, we developed a network application that implements the five modules of iRIDE: Catcher, Predictor, Rescheduler, Monitor, and Trainer. The Catcher uses the value 001111 in the 6-bit DSCP field of the IPv4 header for installing the catching rules that match and forward to the controller the marked packets reporting the identified elephant flows. The Predictor implements no cold start threshold ($\theta_{CS} = 0$) for predicting the flow traffic characteristics. The Rescheduler sets the timeout threshold $\theta_{TO} = 5$ seconds for instructing OpenFlow switches to remove idle rules installed for routing identified elephant flows. Moreover, the Rescheduler inserts the OpenFlow flag OFPFF_SEND_FLOW_REM into the routing rules installed on the edge switches that report the elephant flows (i.e., forward the marked packets). That way, when a rule times out, the OpenFlow switch removes it and reports the flow rule statistics to the controller. The Monitor performs no frequent requesting of flow characteristics from the network ($T_M = \infty$). The Trainer sets the labeling threshold $\theta_L = 100$ kB for filtering out flows incorrectly marked as elephants. In addition, the Trainer implements a mini-batch size $n_B = 1$. Finally, we used the DNN structures and regularization methods from Table 4.8 for implementing the incremental regression models that the Trainer builds and the Predictor applies for inference. Our Ryu network application that implements iRIDE is available in [282]. We executed our iRIDE network application on the Ryu SDN framework.

Each host in our fat-tree emulated network runs a Tcpreplay process for replaying the traffic from a PCAP file. Figure 4.16 describes the process we followed to build a PCAP file for each host from public real packet traces. We used the Scapy 2.4.5 Python library to develop a PCAP parser that hashes the MAC and IP addresses, adds payload, and generates marked packets. Note the two real packet traces [255], UNI1 and UNI2, consist of different PCAP files. For security reasons, these PCAP files come anonymized, which changes the source and destination MAC and IP addresses, remaps the transport layer ports, and truncates the payload of the captured packets [312]. Our PCAP parser then hashes the source and destination MAC and IP addresses for distributing the packets among the hosts in our fat-tree network. We use the source IP for assigning each packet to the corresponding PCAP file (e.g., 10.0.0.1.pcap). The PCAP parser also reads the packet length for padding the payload of each packet with empty data. Lastly, recall that iRIDE relies on a flow detection method, such as NELLY (see Chapter 3), that reports the elephant flows using marked packets. Therefore, the PCAP parser reads the classification results from NELLY for generating the marked packets that report those flows classified as elephants. The PCAP parser inserts the value 001111 into the DSCP field of the marked packets, so the catching rules will match and forward them to the controller. Moreover, the marked packets include the size and IAT of the first 7 packets of the flow into their payload.

4.2.5 Evaluation

Due to resource limitations, we executed our evaluation using a fat-tree topology of size $k = 4$, which deploys 16 hosts, 20 switches (8 at edge layer, 8 at aggregation layer, and

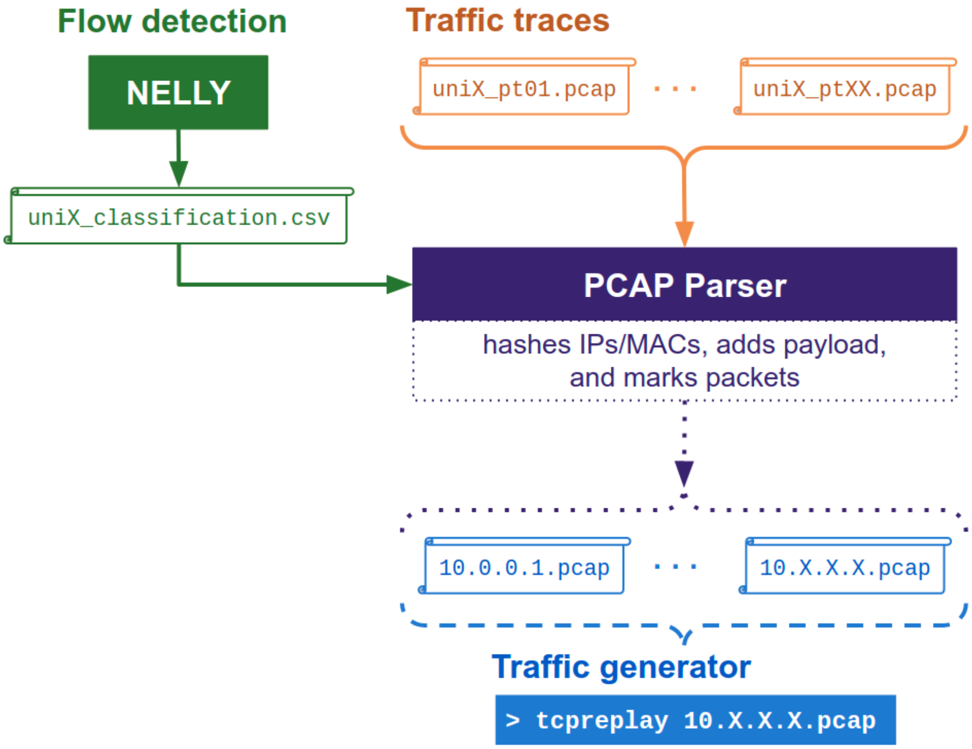


Figure 4.16: PCAP parsing for the traffic generator in iRIDE

4 at core layer), and 48 links (see Table 4.1). Note that the number of logical cores in the server is insufficient for running a fat-tree topology of size $k = 8$ with all the 128 hosts concurrently executing a Tcpreplay process. Since the total bandwidth in our Mininet VM (obtained using the *iperf* test) was restricted to a maximum of 20 Gbps, we limited each link to a bandwidth of 100 Mbps, for a total of 4.8 Gbps. Moreover, we replayed only the traffic from the UNI1 packet trace, for which the PCAP parser generated the PCAP files with a total size of 12 GB. We were not able to generate the PCAP files for the UNI2 packet trace due to disk space limitations in the server. The hosts in our fat-tree emulated network replayed the traffic from UNI1 at top-speed.

For the evaluation, we measured the throughput over time in the links of the fat-tree emulated network when using iRIDE, with both rescheduling algorithms LC and WF+BTF, and when using PM2, which provides an ECMP implementation for OpenFlow networks. Figure 4.17 depicts the throughput over time in the bisection links of the fat-tree topology from the first run of the evaluation. The bisection links of a network is represented by “the minimum number of links to be removed to disconnect the network into two halves of equal size” [313]. It is noteworthy that we also measured the throughput on links from other areas of the fat-tree topology, including the links from the edge layer to the aggregation layer and from the aggregation layer to the core layer (the links from the edge layer to the hosts are not of our interest as they do not provide multiple paths). However, we observed the same pattern as in the bisection links, so we decided to present only the throughput in the bisection links for simplicity. The results show that iRIDE is able to use more efficiently the bisection bandwidth than PM2. Moreover, iRIDE using the rescheduling algorithm WF+BTF is able to use more efficiently the bisection bandwidth than using LC. Note that as better the bandwidth efficiency, the faster to complete the traffic.

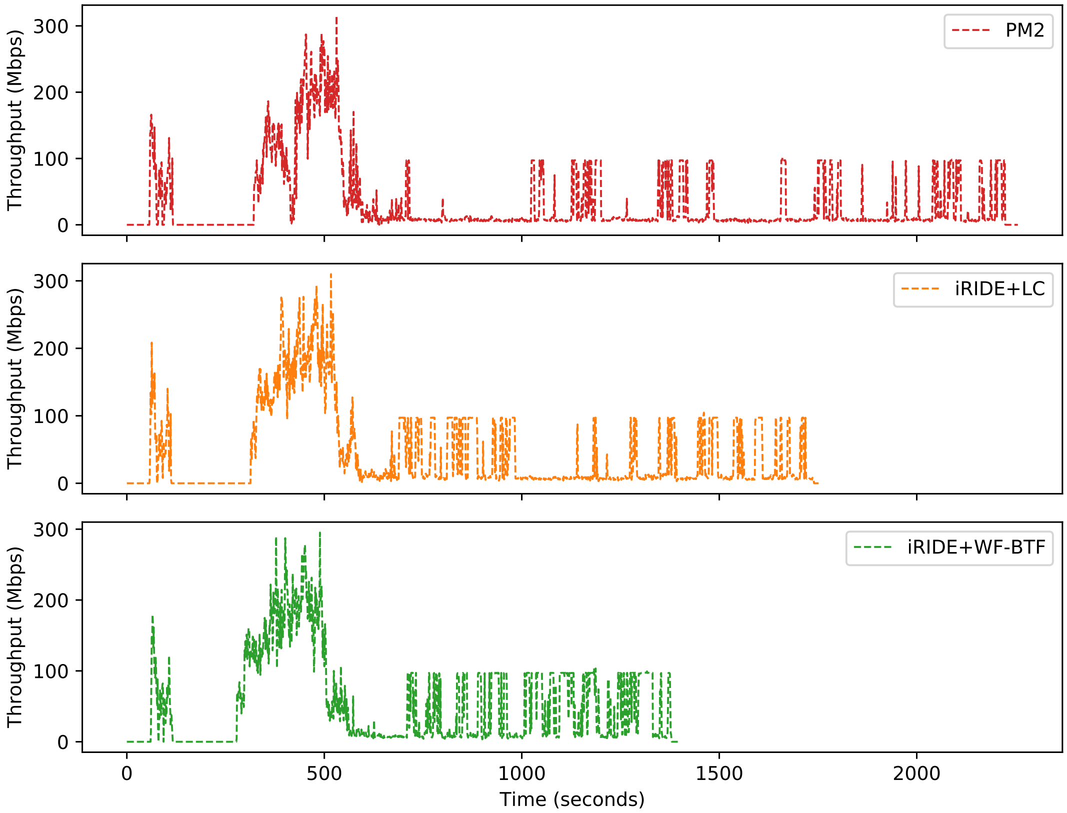
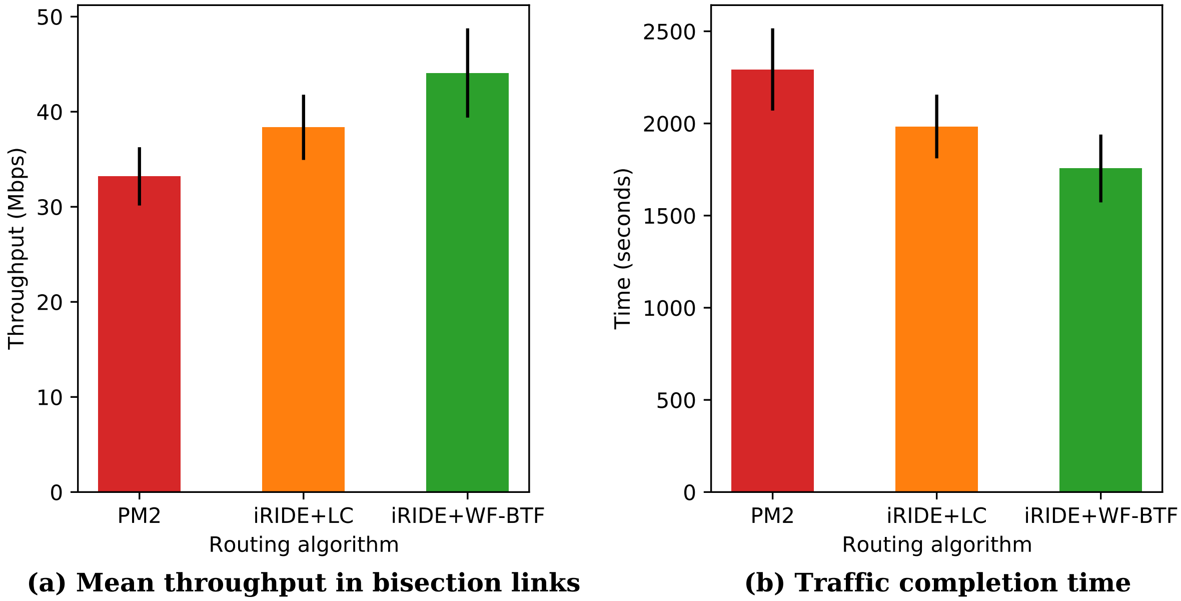


Figure 4.17: Throughput over time in the bisection links of a fat-tree topology of size $k = 4$ when using PM2 and iRIDE, with both LC and WF+BTF, for routing UNI1 traffic

Finally, Figure 4.18(a) presents the mean throughput in the bisection links of the fat-tree topology, whereas Figure 4.18(b) depicts the traffic completion time. Both figures display the average values and the corresponding standard deviation from ten runs for each routing algorithm. Figure 4.18(a) shows that PM2 achieves a mean throughput of 33 Mbps in the bisection links of the fat-tree topology, whereas iRIDE increments that throughput by 5 and 11 Mbps when using LC and WF+BTF, respectively. Conversely, Figure 4.18(b) depicts that all the hosts in the fat-tree topology completed the communication in about 38 minutes when using PM2 only, whereas iRIDE reduced such a traffic completion time by 5 and 9 minutes when using LC and WF+BTF, respectively. These results confirm that iRIDE is able to generate more throughput and to complete the traffic faster than PM2, particularly, when using the rescheduling algorithm WF+BTF, which uses the flow rates and flow durations predicted using the incremental DNNs.



(a) Mean throughput in bisection links

(b) Traffic completion time

Figure 4.18: Mean throughput in the bisection links of a fat-tree topology of size $k = 4$ and traffic completion time when using PM2 and iRIDE, with both LC and WF+BTF, for routing UNI1 traffic

4.3 Final remarks

A relevant problem affecting the overall performance of multipath routing in SDDCNs is the coexistence of mice and elephant flows. Aiming at overcoming this problem, this chapter introduced PM2, a multipath routing algorithm for steering flows (mainly mice) in a fat-tree DCN topology. PM2 supports transparent host migration across the whole network while reducing the number of rules installed on SDN switches, decreasing the delay introduced to flows (mainly mice) traversing the network. An analytical comparison corroborated that PM2 installs much fewer rules than other OpenFlow-feasible multipath routing algorithms that support transparent host migration across a topology area greater than the same edge switch. Futhermore, this chapter proposed iRIDE, a flow rescheduling method that applies incremental learning at the controller-side of SDDCNs for predicting traffic characteristics of flows identified as elephants to compute and install the best path per flow across the network. An extensive evaluation based on real packet traces and various incremental learning algorithms demonstrated the low error for predicting the flow rate and duration of iRIDE when using DNNs with regularization and dropout layers. Furthermore, the evaluation results show the high throughput and short traffic completion time of iRIDE when implementing a rescheduling algorithm that uses the two predicted traffic characteristics.

Chapter 5

Conclusions

This chapter starts summarizing the research work carried out in this thesis. Then, it provides the answers for the fundamental questions that guided the verification of the hypothesis defended in this thesis. The last section outlines directions for future work.

This thesis presented the investigation carried out to verify the hypothesis: **using ML for fine-granularity prediction of flow characteristics and SDN for dynamic control of flow scheduling would allow building a multipath routing mechanism for DCNs that improves[1] the routing function.** Based on the hypothesis, this work proposed a multipath routing mechanism that leverages both SDN and ML to improve the routing function in DCNs. Three major components form the proposed multipath routing mechanism: NELLY, PM2, and iRIDE.

NELLY introduced a flow detection method that incorporates incremental learning at the server-side of SDDCNs to accurately and timely identify elephant flows at low traffic overhead while enabling continuous model adaptation under limited memory resources. An extensive evaluation based on real packet traces and various incremental learning algorithms demonstrated the high accuracy and speed of NELLY when used with the ARF and AHOT algorithms. Moreover, an analytical comparison to seminal related works corroborated the scalability of NELLY as well as its generation of low traffic overhead and the fact that no modifications in SDN standards are required.

PM2 provided a multipath routing algorithm that supports transparent host migration across the whole network while reducing the number of rules installed on SDN switches, decreasing the delay introduced to flows (mainly mice) traversing the SDDCN. A prototype implementation serves as a proof of concept for demonstrating the feasibility of PM2 in a fat-tree DCN topology. Moreover, an analytical comparison corroborated that PM2 installs much fewer rules than other OpenFlow-feasible multipath routing algorithms depending on either MAC or IP addresses and supporting transparent host migration across a topology area greater than the same edge switch.

[1] In terms of high throughput and low delay while efficient use of resources

iRIDE proposed a flow rescheduling method at the controller-side of SDDCNs that improves network throughput and traffic completion time by using incremental learning to predict the rate and duration of elephants for computing and installing the best path across the network. An extensive evaluation based on real packet traces and various incremental learning algorithms demonstrated the low error of iRIDE for predicting the flow rate and flow duration when using DNNs with regularization and dropout layers. Furthermore, the evaluation results show the high throughput and short traffic completion time of iRIDE when implementing the rescheduling algorithm WF+BTF, which uses both predicted traffic characteristics, flow rate and flow duration.

5.1 Answers for the fundamental questions

Two fundamental questions guided the investigation about using ML for fine-granularity prediction of flow characteristics and SDN for dynamic control of flow scheduling aiming at building a multipath routing mechanism for DCNs that improves the routing function. This section reviews and answers such questions.

Fundamental question I: *What is the accuracy and efficiency, in terms of time and memory, of ML techniques for predicting flow characteristics of network traffic from DCNs?*

In this work, NELLY focused on predicting the size of flows by classifying them as mice or elephants. The evaluation results demonstrated that using incremental learning algorithms for performing such a classification achieves high elephant detection with short classification time. In particular, NELLY achieved the best classification performance, in terms of accuracy and time, by using the BinNom headers along with the following adaptive decision trees algorithms. ARF provides the best classification accuracy for UNI1 and UNI2 traffic with a classification time less than 17 μs (i.e., less than 7.5% of the RTT in DCNs). AHOT is also good for UNI1 and UNI2 traffic, with a minor classification accuracy than ARF but reducing the classification time to less than 10 μs. Finally, the Hoeffding tree is only good for traffic similar to that of UNI1 but achieves a classification accuracy similar to that of AHOT with a classification time less than 3 μs.

Similarly, iRIDE focused on predicting the rate and duration of flows by using regression models. The evaluation results revealed that iRIDE achieved the lowest prediction errors of flow rate and flow duration when using DNNs with L2 regularization and dropout layers. In particular, the most accurate DNNs required deep structures from five up to nine hidden layers and from 360 up to 600 units per each hidden layer. Moreover, for predicting the flow rate, the DNN structures achieved the lowest errors by using a combination of L2 regularization and dropout layers, whereas, for predicting the duration, the most accurate DNN structures only required L2 regularization.

Finally, note that incremental learning reduces memory consumption by continuously updating the models from constantly generated data that is temporarily persisted. This enabled both NELLY and iRIDE, which rely on incremental learning algorithms, to adapt to the variations in traffic characteristics and perform endless learning with limited memory resources.

Fundamental question II: *Does incorporating ML techniques to an SDN-based multipath routing mechanism improve network traffic routing, in terms of throughput and delay, in DCNs?*

Recall that NELLY incorporates incremental learning at the server-side of SDDCNs for proactively identifying elephant flows at low traffic overhead. An analytical comparison to seminal related works corroborated that NELLY reduces the switch table occupancy, the traffic overhead, and the flow detection time. In particular, NELLY requires no flow table entries on the SDN switches for collecting data since NELLY operates at the server-side of SDDCNs, having local access to the data. Regarding the traffic overhead, NELLY merely requires that edge (ToR) switches send control packets of flows marked as elephants, greatly reducing the control traffic overhead (to 4.4 kbps if assuming a control packet of 64 bytes). Lastly, NELLY detects elephant flows in a very short time as it relies on the first N packets (0.8 seconds when using the first 7 packets).

On the other hand, iRIDE incorporated incremental learning at the controller-side of SDDCNs to predict the rate and duration of flows. These predicted flow traffic characteristics enabled constructing intelligent elephant rescheduling algorithms, such as LC and WF+BTF. The results from a quantitative evaluation demonstrated that iRIDE efficiently uses the available bandwidth, generating higher throughput and shorter traffic completion time than conventional ECMP. In particular, when replaying the traffic from the UNI1 packet trace in a fat-tree emulated network, iRIDE with WF+BTF incremented the bisection throughput by 11 Mbps and reduced the traffic completion time by 9 minutes in comparison with PM2. Note that WF+BTF uses the flow rates and flow durations predicted using the incremental DNNs.

5.2 Future work

During the development of this thesis, we observed interesting opportunities for further research. These opportunities are outlined as follows.

- Implement NELLY as an in-kernel software component for evaluating its impact cost to server resources, including processing and memory consumption. This implementation would enable to evaluate NELLY in an emulated SDDCN by installing the software component into micro virtual machines connected to Open vSwitch instances.

- Although this paper has proven that incremental learning algorithms are efficient to detect elephant flows in DCNs, there is still no consistent and accepted method for defining the threshold value that discriminates between mice and elephants in DCNs. In this thesis, we evaluated different thresholds but did not specify how to select the appropriate threshold value for the traffic and routing requirements. RL algorithms can be useful for selecting a threshold that maximizes DCN routing performance (e.g., throughput and delay) for specific traffic conditions.

- A future analysis of network traffic using IPv6 or at other layers of IIoT systems would help to analyze if such traffic characteristics can also benefit from incremental learning for either classifying flows (NELLY) or predicting flow features (iRIDE). For example, fog layers are formed by micro data centers that analyze data that require a rapid return (low latency). Moreover, it is expected that IIoT systems introduce more diversity of network traffic (from elephant flows to mouse flows). Therefore, there is a need to create publicly available datasets of network traffic with different characteristics (e.g., IPv6, fog layers) to evaluate the performance improvement of ML-based methods on such datasets.

- Extending iRIDE to achieve a full cognitive networking, such as the C-MAPE loop that we proposed in collaboration with other researchers (see Section 2.2.3. Note that the cognitive operation in iRIDE (i.e., predicting flow traffic characteristics) belongs to the C-Analyze function. However, the statistics collection, the path selection, and the rescheduling algorithm can be extended using an ML-based approach, providing cognition in the Monitor, Plan, and Execute functions, respectively. Different works [40, 314–316] in our research group have already explored some of these cognitive approaches but independently.

Bibliography

[1] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, “Data center network virtualization: A survey,” *IEEE Communications Surveys Tutorials*, vol. 15, pp. 909–928, Second 2013.

[2] R. K. Singh, N. S. Chaudhari, and K. Saxena, “Load balancing in IP/MPLS networks: A survey,” *Communications and Network*, vol. 4, pp. 151–156, May 2012.

[3] N. Wang, K. H. Ho, G. Pavlou, and M. Howarth, “An overview of routing optimization for internet traffic engineering,” *IEEE Communications Surveys Tutorials*, vol. 10, pp. 36–56, First 2008.

[4] S. K. Singh, T. Das, and A. Jukan, “A survey on internet multipath routing and provisioning,” *IEEE Communications Surveys Tutorials*, vol. 17, pp. 2157–2175, Fourthquarter 2015.

[5] M. Chiesa, G. Kindler, and M. Schapira, “Traffic engineering with equal-cost-multipath: An algorithmic perspective,” *IEEE/ACM Transactions on Networking*, vol. 25, pp. 779–792, April 2017.

[6] G. Detal, C. Paasch, S. van der Linden, P. Mérindol, G. Avoine, and O. Bonaventure, “Revisiting flow-based load balancing: Stateless path selection in data center networks,” *Computer Networks*, vol. 57, no. 5, pp. 1204 – 1216, 2013.

[7] C. Hopps, “Analysis of an Equal-Cost Multi-Path Algorithm,” RFC 2992, Internet Engineering Task Force, Nov. 2000.

[8] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM ’15, (New York, NY, USA), pp. 123–137, ACM, 2015.

[9] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC ’10, (New York, NY, USA), pp. 267–280, ACM, 2010.

[10] T. Benson, A. Anand, A. Akella, and M. Zhang, “Understanding data center traffic characteristics,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 92–99, Jan. 2010.

[11] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, “The nature of data center traffic: Measurements & analysis,” in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC ’09, (New York, NY, USA), pp. 202–208, ACM, 2009.

[12] A. Andreyev, “Introducing data center fabric, the next-generation Facebook data center network.” Facebook Engineering, Nov. 2014. https://code.facebook.com/posts/360346274145943/ (accessed Oct. 19, 2021).

[13] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, “On the impact of packet spraying in data center networks,” in *2013 Proceedings IEEE INFOCOM*, pp. 2130–2138, April 2013.

[14] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. A. Maltz, “Per-packet load-balanced, low-latency routing for clos-based data center networks,” in *CoNEXT*, 2013.

[15] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, “Presto: Edge-based load balancing for fast datacenter networks,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM ’15, (New York, NY, USA), pp. 465–478, ACM, 2015.

[16] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, “Wcmp: Weighted cost multipathing for improved fairness in data centers,” in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys ’14, (New York, NY, USA), pp. 5:1–5:14, ACM, 2014.

[17] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, “Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks,” in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’14, (New York, NY, USA), pp. 149–160, ACM, 2014.

[18] S. Sen, D. Shue, S. Ihm, and M. J. Freedman, “Scalable, optimal flow routing in datacenters via local link balancing,” in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’13, (New York, NY, USA), pp. 151–162, ACM, 2013.

[19] S. Kandula, D. Katabi, S. Sinha, and A. Berger, “Dynamic load balancing without packet reordering,” *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 51–62, Mar. 2007.

[20] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, “Let it flow: Resilient asymmetric load balancing with flowlet switching,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, (Boston, MA), pp. 407–420, USENIX Association, 2017.

[21] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, “Hula: Scalable load balancing using programmable data planes,” in *Proceedings of the Symposium on SDN Research*, SOSR ’16, (New York, NY, USA), pp. 10:1–10:12, ACM, 2016.

[22] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, “Conga: Distributed congestion-aware load balancing for datacenters,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM ’14, (New York, NY, USA), pp. 503–514, ACM, 2014.

[23] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, pp. 14–76, Jan 2015.

[24] H. Kim and N. Feamster, “Improving network management with software defined networking,” *IEEE Communications Magazine*, vol. 51, pp. 114–119, February 2013.

[25] H. Xu and B. Li, “Tinyflow: Breaking elephants down into mice in data center networks,” in *2014 IEEE 20th International Workshop on Local Metropolitan Area Networks (LANMAN)*, pp. 1–6, May 2014.

[26] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks,” in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI’10, (Berkeley, CA, USA), pp. 19–19, USENIX Association, 2010.

[27] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, “Devoflow: Scaling flow management for high-performance networks,” in *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM ’11, (New York, NY, USA), pp. 254–265, ACM, 2011.

[28] R. Trestian, G. M. Muntean, and K. Katrinis, “Micetrap: Scalable traffic engineering of datacenter mice flows using openflow,” in *2013 IFIP/IEEE International*

Symposium on Integrated Network Management (IM 2013), pp. 904–907, May 2013.

[29] A. R. Curtis, W. Kim, and P. Yalagandula, “Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection,” in *2011 Proceedings IEEE INFOCOM*, pp. 1629–1637, April 2011.

[30] J. Lu, W. Liu, Y. Zhu, S. Ling, Z. Chen, and J. Zeng, “Scheduling mix-flow in sd-dcn based on deep reinforcement learning with private link,” in *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*, pp. 395–401, 2020.

[31] W.-X. Liu, J. Cai, Y. Wang, Q. C. Chen, and J.-Q. Zeng, “Fine-grained flow classification using deep learning for software defined data center networks,” *Journal of Network and Computer Applications*, vol. 168, p. 102766, 2020.

[32] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin, “Online flow size prediction for improved network routing,” in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pp. 1–6, Nov 2016.

[33] S.-C. Chao, K. C.-J. Lin, and M.-S. Chen, “Flow classification for software-defined data centers using stream mining,” *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 105–116, 2019.

[34] H. Zhang, F. Tang, and L. Barolli, “Efficient flow detection and scheduling for sdn-based big data centers,” *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, pp. 1915–1926, 2019.

[35] F. Tang, L. Li, L. Barolli, and C. Tang, “An efficient sampling and classification approach for flow detection in sdn-based big data centers,” in *IEEE AINA*, (Taipei, Taiwan), pp. 1106–1115, Mar. 2017.

[36] C. F. Estrada Solano, “Software-defined networking management based on web 2.0 and web 3.0 technologies,” Master’s thesis, Telematics Dept., Univ. of Cauca, Popayan, Colombia, Jan. 2016. Available: http://repositorio.unicauca.edu.co:8080/xmlui/handle/123456789/1358.

[37] F. Estrada-Solano, O. M. Caicedo, and N. L. S. Da Fonseca, “Nelly: Flow detection using incremental learning at the server side of sdn-based data centers,” *IEEE Transactions on Industrial Informatics*, vol. 16, no. 2, pp. 1362–1372, 2020.

[38] F. Amezquita-Suarez, F. Estrada-Solano, N. L. S. da Fonseca, and O. M. C. Rendon, “An efficient mice flow routing algorithm for data centers based on software-defined networking,” in *IEEE ICC*, (Shanghai, China), pp. 1–6, May 2019.

[39] F. Estrada-Solano, A. Ordonez, L. Z. Granville, and O. M. Caicedo Rendon, “A framework for sdn integrated management based on a cim model and a vertical management plane,” *Computer Communications*, vol. 102, pp. 150 – 164, Apr. 2017.

[40] M. A. Gironza-Ceron, W. F. Villota-Jacome, A. Ordonez, F. Estrada-Solano, and O. M. Caicedo Rendon, “Sdn management based on hierarchical task network and network functions virtualization,” in *2017 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1360–1365, 2017.

[41] A. I. Montoya-Munoz, D. M. Casas-Velasco, F. Estrada-Solano, A. Ordonez, and O. M. Caicedo Rendon, “A yang model for a vertical sdn management plane,” in *2017 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pp. 1–6, 2017.

[42] S. Ayoubi, N. Limam, M. A. Salahuddin, N. Shahriar, R. Boutaba, F. Estrada-Solano, and O. M. Caicedo Rendon, “Machine learning for cognitive network management,” *IEEE Communications Magazine*, vol. 56, no. 1, pp. 158–165, 2018.

[43] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities,” *Journal of Internet Services and Applications*, vol. 9, p. 16, June 2018.

[44] A. I. Montoya-Munoz, D. Casas-Velasco, F. Estrada-Solano, O. M. Caicedo Rendon, and N. L. Saldanha da Fonseca, “An approach based on yet another next generation for software-defined networking management,” *International Journal of Communication Systems*, vol. 34, no. 11, p. e4855, 2021.

[45] S. Crawford and L. Stucki, “Peer review and the changing research record,” *J. Am. Soc. Inf. Sci.*, vol. 41, pp. 223–228, Mar. 1990.

[46] N. Feamster, J. Rexford, and E. Zegura, “The road to sdn: An intellectual history of programmable networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–98, Apr. 2014.

[47] P. Lin, J. Bi, H. Hu, T. Feng, and X. Jiang, “A quick survey on selected approaches for preparing programmable networks,” in *Proceedings of the 7th Asian Internet Engineering Conference*, AINTEC ’11, (New York, NY, USA), pp. 160–163, ACM, 2011.

[48] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: Enabling innovation in campus networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[49] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, “A roadmap for traffic engineering in sdn-openflow networks,” *Comput. Netw.*, vol. 71, pp. 1–30, Oct. 2014.

[50] L. Efremova and D. Andrushko, “What’s in opendaylight?.” Mirantis, Apr. 2015. https://www.mirantis.com/blog/whats-opendaylight/ (accessed Oct. 19, 2021).

[51] ONF, “Software-defined networking: The new norm for networks,” white paper, Open Network Foundation, Apr. 2012.

[52] S. Kiran and G. Kinghorn, “Cisco open network environment: Bring the network closer to applications,” White Paper C11-728045-03, Cisco, Sept. 2015.

[53] A. Singla and B. Rijsman, *Day One: Understanding OpenContrail Architecture*. Juniper Networks Books, Nov. 2013.

[54] M. Casado, N. Foster, and A. Guha, “Abstractions for software-defined networks,” *Commun. ACM*, vol. 57, p. 86–95, Sept. 2014.

[55] ONF, “Openflow switch specification version 1.5.1,” Technical Specification TS-025, Open Networking Foundation, Mar. 2015.

[56] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, “The programmable data plane: Abstractions, architectures, algorithms, and applications,” *ACM Comput. Surv.*, vol. 54, May 2021.

[57] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, “Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’17, (New York, NY, USA), p. 15–28, Association for Computing Machinery, 2017.

[58] C. Kim, A. Sivaraman, N. P. K. Katta, A. Bas, A. A. Dixit, and L. J. Wobker, “In-band network telemetry via programmable dataplanes,” in *ACM SIGCOMM Symposium on SDN Research (SOSR) ’15 Demos*, June 2015.

[59] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, p. 87–95, July 2014.

[60] L. Yang, R. Dantu, T. Anderson, and R. Gopal, “Forwarding and control element separation (ForCES) framework,” RFC 3746, Internet Engineering Task Force, Apr. 2004.

[61] H. Song, “Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN ’13, (New York, NY, USA), p. 127–132, Association for Computing Machinery, 2013.

[62] P4.org API Working Group, “P4runtime specification.” P4.org, July 2021. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html (accessed Oct. 19, 2021).

[63] R. Sherwood, G. Gibb, K. kiong Yap, M. Casado, N. Mckeown, and G. Parulkar, “Flowvisor: A network virtualization layer,” Technical Report OpenFlow-tr-2009-1, OpenFlow, Oct. 2009.

[64] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “Nox: Towards an operating system for networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 38, p. 105–110, July 2008.

[65] NOX Repo, “The POX network software platform.” GitHub. https://github.com/noxrepo/pox (accessed Oct. 19, 2021).

[66] R. Kubo, T. Fujita, Y. Agawa, and H. Suzuki, “Ryu sdn framework-open-source sdn platform software,” *NTT Technical Review*, vol. 12, 08 2014.

[67] Project Floodlight, “Floodlight controller.” Project Floodlight. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller (accessed Oct. 19, 2021).

[68] OpenDaylight Project, “Welcome to opendaylight documentation.” OpenDaylight Documentation. https://docs.opendaylight.org/ (accessed Oct. 19, 2021).

[69] Trema, “Trema: Full-stack openflow framework in ruby.” GitHub. https://github.com/trema/trema (accessed Oct. 19, 2021).

[70] H. Yin, H. Xie, T. Tsou, D. Lopez, P. Aranda, and R. Sidi, “SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains.” Internet Draft, June 2012.

[71] R. Izard and Q. Wang, “Floodlight REST API.” Project Floodlight, Mar. 2018. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343539/Floodlight+REST+API (accessed Oct. 19, 2021).

[72] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, “Modular sdn programming with pyretic,” *USENIX Login*, vol. 38, pp. 128–134, 10 2013.

[73] A. Voellmy, H. Kim, and N. Feamster, “Procera: A language for high-level reactive network control,” in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN ’12, (New York, NY, USA), p. 43–48, Association for Computing Machinery, 2012.

[74] M. Monaco, O. Michel, and E. Keller, “Applying operating system principles to sdn controller design,” in *In Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, (New York, NY, USA), p. 2:1–2:7, ACM, 2013.

[75] J. A. Wickboldt, W. P. De Jesus, P. H. Isolani, C. B. Both, J. Rochol, and L. Z. Granville, “Software-defined networking: management requirements and challenges,” *IEEE Communications Magazine*, vol. 53, no. 1, pp. 278–285, 2015.

[76] E. Haleplidis, J. Hadi Salim, S. Denazis, and O. Koufopavlou, “Towards a network abstraction model for sdn,” *Journal of Network and Systems Management*, vol. 23, no. 2, pp. 309–327, 2015.

[77] ONF, “Sdn architecture,” Technical Reference TR-502, Open Network Foundation, June 2014. Issue 1.

[78] ITU, “Framework of software-defined networking,” Recommendation Y.3300, International Telecommunication Union, June 2014.

[79] ISO, “Information technology – Open Systems Interconnection – Systems management overview,” ISO/IEC 10040:1998, International Organization for Standardization, Nov. 1998.

[80] T. Bray, “The JavaScript Object Notation (JSON) data interchange format,” Standards Track RFC 7159, Internet Engineering Task Force, Mar. 2014.

[81] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1,” Standards Track RFC 2616, Internet Engineering Task Force, June 1999.

[82] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible Markup Language (XML) 1.0 (fifth edition),” recommendation, World Wide Web Consortium, Nov. 2008.

[83] B. Pfaff and B. Davie, “The open vswitch database management protocol,” Informational RFC 7047, Internet Engineering Task Force, Dec. 2013.

[84] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, “Network Configuration Protocol (NETCONF),” Standards Track RFC 6241, Internet Engineering Task Force, June 2011.

[85] D. Harrington, R. Presuhn, and B. Wijnen, “An architecture for describing Simple Network Management Protocol (SNMP) management frameworks,” Standards Track RFC 3411, Internet Engineering Task Force, Dec. 2002.

[86] ITU, “TMN management functions,” Recommendation M.3400, International Telecommunication Union, Feb. 2000.

[87] DMTF, “Common Information Model (CIM) Infrastructure v2.7.0,” Specification DSP0004, Distributed Management Task Force, Apr. 2012.

[88] A. Pras and J. Schoenwaelder, “On the difference between information models and data models,” Informational RFC 3444, Internet Engineering Task Force, Jan. 2003.

[89] A. Afanasyev and S. K. Ramani, “Ndnconf: Network management framework for named data networking,” in *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6, 2020.

[90] S. Popić, B. Majstorović, M. Vuleta, Đ. Sarić, and B. M. Todorović, “Efficient usage of resources in sdn by modifying yang modules in linux-based embedded systems,” in *2019 27th Telecommunications Forum (TELFOR)*, pp. 1–4, 2019.

[91] G. Parladori, G. Gasparini, F. Ruggi, A. D. Broi, V. Simone, and F. Nicassio, “Yang modelling of optical nodes,” in *20th Italian National Conference on Photonic Technologies (Fotonica 2018)*, pp. 1–4, 2018.

[92] M. Bjorklund, “The YANG 1.1 data modeling language,” Standards Track RFC 7950, Internet Engineering Task Force, Aug. 2016.

[93] J. Schönwälder, M. Björklund, and P. Shafer, “Network configuration management using netconf and yang,” *IEEE Communications Magazine*, vol. 48, pp. 166–173, 2010.

[94] S. Wallin, “Uml visualization of yang models,” in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, pp. 1129–1134, 2011.

[95] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, “Network function virtualization: State-of-the-art and research challenges,” *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[96] C. Ullrich and E. Melis, “Pedagogically founded courseware generation based on htn-planning,” *Expert Systems with Applications*, vol. 36, pp. 9319–9332, 07 2009.

[97] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2009.

[98] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[99] E. Alpaydin, *Introduction to Machine Learning*. The MIT Press, 2nd ed., 2010.

[100] G. Tesauro, “Reinforcement learning in autonomic computing: A manifesto and case studies,” *IEEE Internet Computing*, vol. 11, pp. 22–30, Jan 2007.

[101] M. Bramer, *Principles of Data Mining*. Springer Publishing Company, Incorporated, 2nd ed., 2013.

[102] A. Ng, *Machine Learning Yearning*. Online Draft, 2017.

[103] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, *Data Stream Mining A Practical Approach*. The University of Waikato.

[104] V. Losing, B. Hammer, and H. Wersing, “Incremental on-line learning: A review and comparison of state of the art algorithms,” *Neurocomputing*, vol. 275, pp. 1261–1274, 2018.

[105] A. R. T. Gepperth and B. Hammer, “Incremental learning algorithms and applications,” in *ESANN*, 2016.

[106] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 2nd ed., 2016.

[107] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, Dec 1943.

[108] D. Hebb, *The Organization of Behavior: A Neuropsychological Theory*. A Wiley book in clinical psychology, Wiley, 1949.

[109] A. M. Turing, “Computing machinery and intelligence,” *Mind*, vol. 59, no. 236, pp. 433–460, 1950.

[110] A. Legendre, *Nouvelles méthodes pour la détermination des orbites des comètes*. Nineteenth Century Collections Online (NCCO): Science, Technology, and Medicine: 1780-1925, F. Didot, 1805.

[111] S. M. Stigler, “Gauss and the invention of least squares,” *Ann. Statist.*, vol. 9, pp. 465–474, 05 1981.

[112] A. S. Goldberger, “Econometric computing by hand,” *Journal of Economic and Social Measurement*, vol. 29, no. 1-3, pp. 115–117, 2004.

[113] E. T. Jaynes, “Information theory and statistical mechanics,” *Phys. Rev.*, vol. 106, pp. 620–630, May 1957.

[114] E. T. Jaynes, “Information theory and statistical mechanics. ii,” *Phys. Rev.*, vol. 108, pp. 171–190, Oct 1957.

[115] D. R. Cox, “The regression analysis of binary sequences,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 20, no. 2, pp. 215–242, 1958.

[116] E. Fix and J. L. Hodges, “Discriminatory analysis-nonparametric discrimination: consistency properties,” Report No. 4, Project 21-49-004, USAF School of Aviation Medicine, Feb. 1951.

[117] C. Stanfill and D. Waltz, “Toward memory-based reasoning,” *Commun. ACM*, vol. 29, pp. 1213–1228, Dec. 1986.

[118] M. Rosenblatt, “Remarks on some nonparametric estimates of a density function,” *Ann. Math. Statist.*, vol. 27, pp. 832–837, 09 1956.

[119] E. Parzen, “On estimation of a probability density function and mode,” *Ann. Math. Statist.*, vol. 33, pp. 1065–1076, 09 1962.

[120] H. Robbins and S. Monro, “A Stochastic Approximation Method,” *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400 – 407, 1951.

[121] J. Kiefer and J. Wolfowitz, “Stochastic Estimation of the Maximum of a Regression Function,” *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462 – 466, 1952.

[122] C. Darken and J. Moody, “Note on learning rate schedules for stochastic optimization,” in *Advances in Neural Information Processing Systems* (R. P. Lippmann, J. Moody, and D. Touretzky, eds.), vol. 3, Morgan-Kaufmann, 1991.

[123] M. Moller, “Supervised learning on large redundant training sets,” in *Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop*, pp. 79–89, 1992.

[124] D. Newton, R. Pasupathy, and F. Yousefian, “Recent trends in stochastic gradient descent for machine learning and big data,” in *2018 Winter Simulation Conference (WSC)*, pp. 366–380, 2018.

[125] C. K. Chow, “An optimum character recognition system using decision functions,” *IRE Transactions on Electronic Computers*, vol. EC-6, pp. 247–254, Dec 1957.

[126] M. E. Maron, “Automatic indexing: An experimental inquiry,” *J. ACM*, vol. 8, pp. 404–417, July 1961.

[127] M. Bayes and M. Price, “An essay towards solving a problem in the doctrine of chances. by the late rev. mr. bayes, f. r. s. communicated by mr. price, in a letter to john canton, a. m. f. r. s.,” *Philosophical Transactions (1683-1775)*, vol. 53, pp. 370–418, 1763.

[128] P. S. Laplace, *Théorie analytique des probabilités*. Paris, France: Courcier, 1812.

[129] H. Steinhaus, “Sur la division des corp materiels en parties,” *Bull. Acad. Polon. Sci*, vol. 1, pp. 801–804, 1956.

[130] J. MacQueen, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, vol. 1, (Berkeley, CA, USA), pp. 281–297, University of California Press, 1967.

[131] A. A. Markov, “An example of statistical investigation in the text of eugene onegin illustrating coupling of tests in chains,” in *Proceedings of the Royal Academy of Sciences of St. Petersburg*, vol. 1, (St. Petersburg, Rusia), p. 153, 1913.

[132] P. Gagniuc, *Markov Chains: From Theory to Implementation and Experimentation*. Wiley, 2017.

[133] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1 ed., 1957.

[134] F. Rosenblatt, “The perceptron, a perceiving and recognizing automaton,” Report No. 85-460-1, Project PARA, Cornell Aeronautical Laboratory, Jan. 1957.

[135] A. L. Samuel, “Some studies in machine learning using the game of checkers,” *IBM J. Res. Dev.*, vol. 3, pp. 210–229, July 1959.

[136] R. L. Stratonovich, “Conditional markov processes,” *Theory of Probability & Its Applications*, vol. 5, no. 2, pp. 156–178, 1960.

[137] L. E. Baum and T. Petrie, “Statistical inference for probabilistic functions of finite state markov chains,” *Ann. Math. Statist.*, vol. 37, pp. 1554–1563, 12 1966.

[138] K. Astrom, “Optimal control of markov processes with incomplete state information,” *Journal of Mathematical Analysis and Applications*, vol. 10, no. 1, pp. 174 – 205, 1965.

[139] E. J. Sondik, *The Optimal Control of Partially Observable Markov Decision Processes*. PhD thesis, Stanford University, California, 1971.

[140] M. Tsetlin, *Automaton Theory and Modeling of Biological Systems*. Automaton Theory and Modeling of Biological Systems, Academic Press, 1973.

[141] K. S. Narendra and M. A. L. Thathachar, "Learning automata - a survey," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-4, pp. 323–334, July 1974.

[142] J. N. Morgan and J. A. Sonquist, "Problems in the analysis of survey data, and a proposal," *Journal of the American Statistical Association*, vol. 58, no. 302, pp. 415–434, 1963.

[143] R. Messenger and L. Mandell, "A modal search technique for predictibe nominal scale multivariate analys," *Journal of the American Statistical Association*, vol. 67, no. 340, pp. 768–772, 1972.

[144] A. G. Ivakhnenko, "Polynomial theory of complex systems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-1, pp. 364–378, Oct 1971.

[145] A. Ivakhnenko, V. Lapa, and P. U. L. I. S. O. E. ENGINEERING., *Cybernetic Predicting Devices*. Purdue University School of Electrical Engineering, 1965.

[146] I. N. Aizenberg, N. N. Aizenberg, and J. P. Vandewalle, *Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[147] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, 1972.

[148] P. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1975.

[149] J. S. Albus, "A new approach to manipulator control: the cerebellar model articulation controller (cmac," *Journal of Dynamic Systems, Measurement, and Control*, vol. 97, pp. 220–227, 1975.

[150] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 39, no. 1, pp. 1–38, 1977.

[151] I. H. Witten, "An adaptive optimal controller for discrete-time markov environments," *Information and Control*, vol. 34, no. 4, pp. 286 – 295, 1977.

[152] J. Quinlan, “Discovering rules form large collections of examples: A case study,” *Expert Systems in the Micro Electronic Age. Edinburgh Press: Edingburgh*, 1979.

[153] R. Michalski and J. Larson, “Selection of most representative training examples and incremental generation of vl1 hypotheses: The underlying methodology and the description of programs esel and aq11,” Technical Report 867, University of Illinois, Urbana, May 1978.

[154] H. J. Kelley, “Gradient theory of optimal flight paths,” *ARS Journal*, vol. 30, no. 10, pp. 947–954, 1960.

[155] A. E. Bryson, “A gradient method for optimizing multi-stage allocation processes,” in *Proc. Harvard Univ. Symposium on digital computers and their applications*, p. 72, April 1961.

[156] A. Bryson and Y. Ho, *Applied optimal control: optimization, estimation, and control*. Blaisdell book in the pure and applied sciences, Blaisdell Pub. Co., 1969.

[157] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, (Berlin, Heidelberg), p. 9–50, Springer-Verlag, 1998.

[158] Y. Zhu, G. Zhang, and J. Qiu, “Network traffic prediction based on particle swarm bp neural network.,” *JNW*, vol. 8, no. 11, pp. 2685–2691, 2013.

[159] R. S. Sutton and A. G. Barto, “A temporal-difference model of classical conditioning,” in *Proceedings of the ninth annual conference of the cognitive science society*, pp. 355–378, Seattle, WA, 1987.

[160] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[161] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biological Cybernetics*, vol. 36, pp. 193–202, Apr 1980.

[162] T. Kohonen, “Self-organized formation of topologically correct feature maps,” *Biological Cybernetics*, vol. 43, pp. 59–69, Jan 1982.

[163] J. J. Hopfield, “Neural networks and physical systems with emergent collective computational abilities,” *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[164] D. H. Hubel and T. N. Wiesel, “Receptive fields of single neurones in the cat’s striate cortex,” *The Journal of Physiology*, vol. 148, no. 3, pp. 574–591, 1959.

[165] D. H. Hubel and T. N. Wiesel, “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex,” *The Journal of Physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[166] M. I. Jordan, “Serial order: A parallel distributed processing approach,” Tech. Rep. ICS Report 8604, University of California, San Diego, 1986.

[167] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart, “Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1,” in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1* (D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, eds.), ch. Distributed Representations, pp. 77–109, Cambridge, MA, USA: MIT Press, 1986.

[168] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1,” in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1* (D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, eds.), ch. Learning Internal Representations by Error Propagation, pp. 318–362, Cambridge, MA, USA: MIT Press, 1986.

[169] Y. Le Cun, *Learning Process in an Asymmetric Threshold Network*, pp. 233–240. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986.

[170] P. Smolensky, “Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1,” in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1* (D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, eds.), ch. Information Processing in Dynamical Systems: Foundations of Harmony Theory, pp. 194–281, Cambridge, MA, USA: MIT Press, 1986.

[171] A. S. Lapedes and R. M. Farber, “How neural nets work,” in *NIPS*, 1987.

[172] D. S. Broomhead and D. Lowe, “Radial basis functions, multi-variable functional interpolation and adaptive networks,” Memorandum No. 4148, Royal Signals and Radar Establishment Malvern (United Kingdom), 1988.

[173] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, “A learning algorithm for boltzmann machines,” *Cognitive science*, vol. 9, no. 1, pp. 147–169, 1985.

[174] G. E. Hinton, “Training products of experts by minimizing contrastive divergence,” *Training*, vol. 14, no. 8, 2006.

[175] R. Salakhutdinov and G. Hinton, “Deep boltzmann machines,” in *Artificial Intelligence and Statistics*, pp. 448–455, 2009.

[176] R. Dechter, “Learning while searching in constraint-satisfaction-problems,” in *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI’86, pp. 178–183, AAAI Press, 1986.

[177] J. Pearl, “Bayesian networks: A model of self-activated memory for evidential reasoning,” in *Proceedings of the 7th Conference of the Cognitive Science Society*, (University of California, Irvine), pp. 329–334, Aug. 1985.

[178] T. Dean and K. Kanazawa, “A model for reasoning about persistence and causation,” *Computational Intelligence*, vol. 5, no. 2, pp. 142–150, 1989.

[179] P. Dagum, A. Galper, and E. J. Horvitz, *Temporal Probabilistic Reasoning: Dynamic Network Models for Forecasting*. Knowledge Systems Laboratory, Medical Computer Science, Stanford University, 1991.

[180] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series, Taylor & Francis, 1984.

[181] J. R. Quinlan, “Simplifying decision trees,” *Int. J. Man-Mach. Stud.*, vol. 27, pp. 221–234, Sept. 1987.

[182] J. C. Schlimmer and D. Fisher, “A case study of incremental concept induction,” in *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI’86, p. 496–501, AAAI Press, 1986.

[183] P. E. UTGOFF, “Id5: An incremental id3,” in *Machine Learning Proceedings 1988* (J. Laird, ed.), pp. 107–120, San Francisco (CA): Morgan Kaufmann, 1988.

[184] P. E. Utgoff, “Incremental induction of decision trees,” *Mach. Learn.*, vol. 4, p. 161–186, Nov. 1989.

[185] C. J. Watkins, *Models of Delayed Reinforcement Learning*. PhD thesis, Psychology Department, Cambridge University, 1989.

[186] A. Jennings, “A learning system for communications network configuration,” *Engineering Applications of Artificial Intelligence*, vol. 1, no. 3, pp. 151 – 160, 1988.

[187] K. J. Macleish, “Mapping the integration of artificial intelligence into telecommunications,” *IEEE Journal on Selected Areas in Communications*, vol. 6, pp. 892–898, Jun 1988.

[188] L. Bernstein and C. M. Yuhas, “How technology shapes network management,” *IEEE Network*, vol. 3, pp. 16–19, July 1989.

[189] D. J. Montana and L. Davis, “Training feedforward neural networks using genetic algorithms,” in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI’89, (San Francisco, CA, USA), pp. 762–767, Morgan Kaufmann Publishers Inc., 1989.

[190] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, pp. 1942–1948 vol.4, Nov 1995.

[191] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.

[192] J. R. Quinlan, “Learning with continuous classes,” in *Proceedings of Australian Joint Conference on Artificial Intelligence*, pp. 343–348, World Scientific, November 1992.

[193] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[194] R. Tibshirani, “Regression shrinkage and selection via the lasso,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[195] B. E. Boser, I. M. Guyon, and V. N. Vapnik, “A training algorithm for optimal margin classifiers,” in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT ’92, (New York, NY, USA), pp. 144–152, ACM, 1992.

[196] N. A. Syed, H. Liu, and K. K. Sung, “Handling concept drifts in incremental learning with support vector machines,” in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’99, (New York, NY, USA), p. 317–321, Association for Computing Machinery, 1999.

[197] G. A. Rummery and M. Niranjan, “On-line Q-learning using connectionist systems,” CUED/F-INFENG/TR 166, Cambridge University Engineering Department, Sept. 1994.

[198] N. Littlestone and M. K. Warmuth, “The weighted majority algorithm,” in *30th Annual Symposium on Foundations of Computer Science*, pp. 256–261, Oct 1989.

[199] R. E. Schapire, “The strength of weak learnability,” *Mach. Learn.*, vol. 5, pp. 197–227, July 1990.

[200] L. Breiman, “Bagging predictors,” *Mach. Learn.*, vol. 24, pp. 123–140, Aug. 1996.

[201] T. K. Ho, “Random decision forests,” in *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR ’95, (Washington, DC, USA), pp. 278–, IEEE Computer Society, 1995.

[202] Y. Freund and R. E. Schapire, “Experiments with a new boosting algorithm,” in *Proceedings of the Thirteenth International Conference on International Conference on Machine Learning*, ICML’96, (San Francisco, CA, USA), pp. 148–156, Morgan Kaufmann Publishers Inc., 1996.

[203] J. H. Friedman, “Stochastic gradient boosting,” *Comput. Stat. Data Anal.*, vol. 38, pp. 367–378, Feb. 2002.

[204] S. N. Srihari and E. J. Kuebert, “Integration of hand-written address interpretation technology into the united states postal service remote computer reader system,” in *Proceedings of the 4th International Conference on Document Analysis and Recognition*, ICDAR ’97, (Washington, DC, USA), pp. 892–896, IEEE Computer Society, 1997.

[205] ACM Special Interest Group on KDD, “KDD cup archives.” KDD. http://www.kdd.org/kdd-cup (accessed Oct. 19, 2021).

[206] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 1997.

[207] J. H. Friedman, “Greedy function approximation: A gradient boosting machine.,” *Ann. Statist.*, vol. 29, pp. 1189–1232, 10 2001.

[208] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely randomized trees,” *Machine Learning*, vol. 63, pp. 3–42, Apr 2006.

[209] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” *CoRR*, vol. abs/1603.02754, 2016.

[210] G. E. Hinton, S. Osindero, and Y. W. Teh, “A fast learning algorithm for deep belief nets,” *Neural Computation*, vol. 18, pp. 1527–1554, July 2006.

[211] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, “Greedy layer-wise training of deep networks,” in *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS’06, (Cambridge, MA, USA), pp. 153–160, MIT Press, 2006.

[212] M. Ranzato, C. Poultney, S. Chopra, and Y. LeCun, “Efficient learning of sparse representations with an energy-based model,” in *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS’06, (Cambridge, MA, USA), pp. 1137–1144, MIT Press, 2006.

[213] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[214] G. Tesauro, “Practical issues in temporal difference learning,” *Machine Learning*, vol. 8, pp. 257–277, May 1992.

[215] L.-J. Lin, *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.

[216] P. Domingos and G. Hulten, “Mining high-speed data streams,” in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’00, (New York, NY, USA), p. 71–80, Association for Computing Machinery, 2000.

[217] G. Hulten, L. Spencer, and P. Domingos, “Mining time-changing data streams,” in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’01, (New York, NY, USA), p. 97–106, Association for Computing Machinery, 2001.

[218] S. Nishimura, M. Terabe, K. Hashimoto, and K. Mihara, “Learning higher accuracy decision trees from concept drifting data streams,” in *Proceedings of the 21st International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems: New Frontiers in Applied Artificial Intelligence*, IEA/AIE ’08, (Berlin, Heidelberg), p. 179–188, Springer-Verlag, 2008.

[219] A. Bifet and R. Gavaldà, “Adaptive learning from evolving data streams,” in *Advances in Intelligent Data Analysis VIII* (N. M. Adams, C. Robardet, A. Siebes, and J.-F. Boulicaut, eds.), (Berlin, Heidelberg), pp. 249–260, Springer Berlin Heidelberg, 2009.

[220] B. Pfahringer, G. Holmes, and R. Kirkby, “New options for hoeffding trees,” in *Proceedings of the 20th Australian Joint Conference on Advances in Artificial Intelligence*, AI’07, (Berlin, Heidelberg), p. 90–99, Springer-Verlag, 2007.

[221] H. M. Gomes, A. Bifet, J. Read, J. P. Barddal, F. Enembreck, B. Pfahringer, G. Holmes, and T. Abdessalem, “Adaptive random forests for evolving data stream classification,” *Machine Learning*, vol. 106, pp. 1–27, 10 2017.

[222] N. Oza, “Online bagging and boosting,” in *2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 3, pp. 2340–2345 Vol. 3, 2005.

[223] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, "Online passive-aggressive algorithms," *J. Mach. Learn. Res.*, vol. 7, p. 551–585, Dec. 2006.

[224] J. a. Gama and P. Kosina, "Learning decision rules from data streams," in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, p. 1255–1260, AAAI Press, 2011.

[225] D. Brzeziński and J. Stefanowski, "Accuracy updated ensemble for data streams with concept drift," in *Hybrid Artificial Intelligent Systems* (E. Corchado, M. Kurzyński, and M. Woźniak, eds.), (Berlin, Heidelberg), pp. 155–163, Springer Berlin Heidelberg, 2011.

[226] D. Brzezinski and J. Stefanowski, "Combining block-based and online methods in learning ensembles from concept drifting data streams," *Information Sciences*, vol. 265, pp. 50–67, 2014.

[227] P. Domingos, *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. Basic Books, Basic Books, 2015.

[228] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, *et al.*, "Knowledge-defined networking," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, pp. 2–10, 2017.

[229] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski, "A knowledge plane for the internet," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 3–10, 2003.

[230] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart, "An architectural approach to autonomic computing," in *International Conference on Autonomic Computing, 2004. Proceedings.*, pp. 2–9, May 2004.

[231] T. Zimmerman and S. Kambhampati, "Learning-assisted automated planning: looking back, taking stock, going forward," *AI Magazine*, vol. 24, no. 2, pp. 73–96, 2003.

[232] W. Xia, P. Zhao, Y. Wen, and H. Xie, "A survey on data center networking (dcn): Infrastructure and operations," *IEEE Communications Surveys Tutorials*, vol. 19, pp. 640–656, Firstquarter 2017.

[233] K. Chen, C. Hu, X. Zhang, K. Zheng, Y. Chen, and A. V. Vasilakos, "Survey on routing in data centers: insights and future directions," *IEEE Network*, vol. 25, pp. 6–10, July 2011.

[234] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica, “A cost comparison of datacenter network architectures,” in *Proceedings of the 6th International COnference*, Co-NEXT ’10, (New York, NY, USA), pp. 16:1–16:12, ACM, 2010.

[235] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, “Vl2: A scalable and flexible data center network,” in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM ’09, (New York, NY, USA), pp. 51–62, ACM, 2009.

[236] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, “Jellyfish: Networking data centers randomly,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI’12, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2012.

[237] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, “Bcube: A high performance, server-centric network architecture for modular data centers,” in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM ’09, (New York, NY, USA), pp. 63–74, ACM, 2009.

[238] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, “Dcell: A scalable and fault-tolerant network structure for data centers,” in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM ’08, (New York, NY, USA), pp. 75–86, ACM, 2008.

[239] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer, “Firefly: A reconfigurable wireless data center fabric using free-space optics,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM ’14, (New York, NY, USA), pp. 319–330, ACM, 2014.

[240] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, “Helios: A hybrid electrical/optical switch architecture for modular data centers,” in *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM ’10, (New York, NY, USA), pp. 339–350, ACM, 2010.

[241] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, “Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM ’15, (New York, NY, USA), pp. 183–197, ACM, 2015.

[242] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM ’08, (New York, NY, USA), pp. 63–74, ACM, 2008.

[243] D. O. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao, “Overview and Principles of Internet Traffic Engineering,” RFC 3272, Internet Engineering Task Force, May 2002.

[244] R. Zhang-Shen and N. McKeown, “Designing a predictable internet backbone with valiant load-balancing,” in *Proceedings of the 13th International Conference on Quality of Service*, IWQoS’05, (Berlin, Heidelberg), pp. 178–192, Springer-Verlag, 2005.

[245] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, “Improving datacenter performance and robustness with multipath tcp,” in *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM ’11, (New York, NY, USA), pp. 266–277, ACM, 2011.

[246] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, “Detail: Reducing the flow completion time tail in datacenter networks,” in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM ’12, (New York, NY, USA), pp. 139–150, ACM, 2012.

[247] S. Sinha, S. Kandula, and D. Katabi, “Harnessing TCPs Burstiness using Flowlet Switching,” in *3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, (San Diego, CA), November 2004.

[248] Z. Liu, D. Gao, Y. Liu, and H. Zhang, “An enhanced scheduling mechanism for elephant flows in sdn-based data center,” in *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall)*, pp. 1–5, Sept 2016.

[249] C. Y. Lin, C. Chen, J. W. Chang, and Y. H. Chu, “Elephant flow detection in datacenters using openflow-based hierarchical statistics pulling,” in *2014 IEEE Global Communications Conference*, pp. 2264–2269, Dec 2014.

[250] P. Phaal, S. Panchen, and N. McKee, “Inmon corporation’s sflow: A method for monitoring traffic in switched and routed networks,” RFC 3176, Internet Engineering Task Force, Sept. 2001.

[251] L. A. Dias Knob, R. P. Esteves, L. Z. Granville, and L. M. R. Tarouco, “Sdefix — identifying elephant flows in sdn-based ixp networks,” in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pp. 19–26, 2016.

[252] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, “Identifying elephant flows through periodically sampled packets,” in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC ’04, (New York, NY, USA), pp. 115–120, ACM, 2004.

[253] P. Xiao, W. Qu, H. Qi, Y. Xu, and Z. Li, “An efficient elephant flow detection with cost-sensitive in sdn,” in *2015 1st International Conference on Industrial Networks and Intelligent Systems (INISCom)*, pp. 24–28, 2015.

[254] MAWI Working Group, “Packet traces from WIDE backbone.” WIDE Project. http://mawi.wide.ad.jp/mawi/ (accessed Oct. 19, 2021).

[255] T. Benson, “Data set for IMC 2010 data center measurement.” University of Wisconsin-Madison. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html (accessed Oct. 19, 2021).

[256] D. Kotz, T. Henderson, I. Abyzov, and J. Yeo, “The dartmouth/campus dataset (v. 2009-09-09).” CRAWDAD, Sept. 2009. http://crawdad.org/dartmouth/campus/20090909 (accessed Oct. 19, 2021).

[257] Z. Liu, D. Gao, Y. Liu, H. Zhang, and C. H. Foh, “An adaptive approach for elephant flow detection with the rapidly changing traffic in data center network,” *International Journal of Network Management*, vol. 27, no. 6, pp. e1987–n/a, 2017. e1987 nem.1987.

[258] T. Benson, A. Anand, A. Akella, and M. Zhang, “Microte: Fine grained traffic engineering for data centers,” in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT ’11, (New York, NY, USA), pp. 8:1–8:12, ACM, 2011.

[259] L. Nie, D. Jiang, L. Guo, S. Yu, and H. Song, “Traffic matrix prediction and estimation based on deep learning for data center networks,” in *2016 IEEE Globecom Workshops (GC Wkshps)*, pp. 1–6, Dec 2016.

[260] N. L. S. da Fonseca and R. Boutaba, *Cloud Services, Networking, and Management*. Hoboken, NJ, USA: John Wiley & Sons, 1st ed., 2015.

[261] A. Zarek, “Openflow timeouts demystified,” Master’s thesis, Department of Computer Science, University of Toronto, ON, Canada, 2012.

[262] F. Estrada-Solano, “NELLY datasets.” GitHub. https://github.com/festradasolano/nelly/tree/master/nelly-ml/analysis/datasets (accessed Oct. 19, 2021).

[263] D. Chicco and G. Jurman, “The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation,” *BMC Genomics*, vol. 21, pp. 6–, Jan. 2020.

[264] S. Boughorbel, F. Jarray, and M. El-Anbari, “Optimal classifier for imbalanced data using matthews correlation coefficient metric,” *PLOS ONE*, vol. 12, pp. 1–17, June 2017.

[265] D. Chicco, “Ten quick tips for machine learning in computational biology,” *BioData Mining*, vol. 10, pp. 1–17, Dec. 2017.

[266] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, “MOA: massive online analysis,” *Journal of Machine Learning Research*, vol. 11, pp. 1601–1604, Aug. 2010.

[267] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 63–74, Aug. 2010.

[268] M. Miglierina, “Application deployment and management in the cloud,” in *SYNASC*, (Timisoara, Romania), pp. 422–428, Sept. 2014.

[269] F. F. Brasser, M. Bucicoiu, and A.-R. Sadeghi, “Swap and play: Live updating hypervisors and its application to xen,” in *ACM CCSW*, (Scottsdale, AZ, USA), pp. 33–44, Nov. 2014.

[270] M. Afaq, S. U. Rehman, and W.-C. Song, “A Framework for Classification and Visualization of Elephant Flows in SDN-Based Networks,” *Procedia Computer Science*, vol. 65, pp. 672–681, 2015.

[271] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, “Can the production network be the testbed?,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI’10, (USA), p. 365–378, USENIX Association, 2010.

[272] B. Nunes Astuto, M. Mendonça, X. Nam Nguyen, K. Obraczka, and T. Turletti, “A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks,” *HAL Archive*, vol. 16, no. 3, pp. 1617–1634, 2014.

[273] P. Song, Y. Liu, T. Liu, and D. Qian, “Controller-proxy: Scaling network management for large-scale SDN networks,” *Elsevier Computer Communications*, vol. 108, pp. 52–63, 2017.

[274] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, “Efficient traffic splitting on commodity switches,” *CoNEXT*, pp. 1–13, 2015.

[275] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *ACM SIGCOMM*, Hotnets-IX, (New York, NY, USA), pp. 19:1–19:6, 2010.

[276] M. Kerrisk, “ip-netns(8).” Linux manual page, Aug. 2021. https://man7.org/linux/man-pages/man8/ip-netns.8.html (accessed Oct. 19, 2021).

[277] Open vSwitch, “What is open vswitch?.” Linux Foundation Collaborative Projects. https://docs.openvswitch.org/en/latest/intro/what-is-ovs/ (accessed Oct. 19, 2021).

[278] M. Kerrisk, “veth(4).” Linux manual page, Aug. 2021. https://man7.org/linux/man-pages/man4/veth.4.html (accessed Oct. 19, 2021).

[279] Aruba, “Data sheet aruba 2920 switch series.” Aruba Networks. https://www.arubanetworks.com/assets/ds/DS_2920SwitchSeries.pdf (accessed Oct. 19, 2021).

[280] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, “PortLand : A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric,” *ACM SIGCOMM*, pp. 39–50, 2009.

[281] IEEE Standards Association, “Standard group mac addresses: A tutorial guide.” IEEE. https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/tutorials/macgrp.pdf (accessed Oct. 19, 2021).

[282] F. Estrada-Solano, “iRide Ryu SDN applications.” GitHub. https://github.com/festradasolano/iride/tree/master/ryu (accessed Dec. 13, 2021).

[283] W. Sarle, “Should i normalize/standardize/rescale the data?.” faqs.org, Mar. 2014. http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html (accessed Dec. 19, 2021).

[284] W. Sarle, “Why not code binary inputs as 0 and 1?.” faqs.org, Mar. 2014. http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-8.html (accessed Dec. 19, 2021).

[285] T. O. Kvalseth, “Cautionary note about r2,” *The American Statistician*, vol. 39, no. 4, pp. 279–285, 1985.

[286] scikit-learn developers, “Metrics and scoring: quantifying the quality of prediction.” scikit-learn, 2021. https://scikit-learn.org/stable/modules/model_evaluation.html (accessed Dec. 19, 2021).

[287] DeepLearning.AI, “Train/dev/test sets.” YouTube, Aug. 2017. https://www.youtube.com/watch?v=1waHlpKiNyY (accessed Dec. 19, 2021).

[288] J. a. Gama, R. Sebastião, and P. P. Rodrigues, “Issues in evaluation of stream learning algorithms,” in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’09, (New York, NY, USA), p. 329–338, Association for Computing Machinery, 2009.

[289] scikit-learn developers, “Preprocessing data.” scikit-learn, 2021. https://scikit-learn.org/stable/modules/preprocessing.html (accessed Dec. 19, 2021).

[290] P. Branco, L. Torgo, and R. P. Ribeiro, “SMOGN: a pre-processing approach for imbalanced regression,” in *Proceedings of First International Workshop on Learning with Imbalanced Domains: Theory and Applications* (P. B. Luís Torgo and N. Moniz, eds.), vol. 74 of *Proceedings of Machine Learning Research*, pp. 36–50, PMLR, 22 Sep 2017.

[291] L. Torgo, P. Branco, R. P. Ribeiro, and B. Pfahringer, “Resampling strategies for regression,” *Expert Systems*, vol. 32, no. 3, pp. 465–476, 2015.

[292] L. Torgo, R. P. Ribeiro, B. Pfahringer, and P. Branco, “Smote for regression,” in *Progress in Artificial Intelligence* (L. Correia, L. P. Reis, and J. Cascalho, eds.), (Berlin, Heidelberg), pp. 378–389, Springer Berlin Heidelberg, 2013.

[293] P. Branco, R. P. Ribeiro, and L. Torgo, “Ubl: an r package for utility-based learning,” 2016.

[294] 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,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[295] J. Montiel, J. Read, A. Bifet, and T. Abdessalem, “Scikit-multiflow: A multi-output streaming framework,” *Journal of Machine Learning Research*, vol. 19, no. 72, pp. 1–5, 2018.

[296] H. M. Gomes, J. Read, A. Bifet, J. P. Barddal, and J. a. Gama, “Machine learning for streaming data: State of the art, challenges, and opportunities,” *SIGKDD Explor. Newsl.*, vol. 21, p. 6–22, nov 2019.

[297] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané,

R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[298] F. Chollet *et al.*, "Keras." Keras, 2015. https://keras.io (accessed Dec. 19, 2021).

[299] DeepLearning.AI, "Weight initialization in a deep network." YouTube, Aug. 2017. https://www.youtube.com/watch?v=s2coXdufOzE (accessed Dec. 19, 2021).

[300] DeepLearning.AI, "Adam optimization algorithm." YouTube, Aug. 2017. https://www.youtube.com/watch?v=JXQT_vxqwIs (accessed Dec. 19, 2021).

[301] DeepLearning.AI, "Understanding mini-batch gradient dexcent." YouTube, Aug. 2017. https://www.youtube.com/watch?v=-_4Zi8fCZO4 (accessed Dec. 19, 2021).

[302] A. F. Agarap, "Deep learning using rectified linear units (relu)," 2018.

[303] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015.

[304] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[305] F. Chollet *et al.*, "Model training apis." Keras, 2015. https://keras.io/api/models/model_training_apis/ (accessed Dec. 19, 2021).

[306] DeepLearning.AI, "Bias/variance." YouTube, Aug. 2017. https://www.youtube.com/watch?v=SjQyLhQIXSM (accessed Dec. 19, 2021).

[307] DeepLearning.AI, "Regularization." YouTube, Aug. 2017. https://www.youtube.com/watch?v=6g0t3Phly2M (accessed Dec. 19, 2021).

[308] DeepLearning.AI, "Dropout regularization." YouTube, Aug. 2017. https://www.youtube.com/watch?v=D8PJAL-MZv8 (accessed Dec. 19, 2021).

[309] J. Boyar, L. Epstein, L. M. Favrholdt, J. S. Kohrt, K. S. Larsen, M. M. Pedersen, and S. Wøhlk, "The maximum resource bin packing problem," *Theoretical Computer Science*, vol. 362, no. 1, pp. 127–139, 2006.

[310] F. Klassen and AppNeta, "Tcpreplay - pcap editing and replaying utilities." Tcpreplay. https://tcpreplay.appneta.com/ (accessed Dec. 19, 2021).

[311] G. Roualland, "ifstat(1) - linux man page." die.net. https://linux.die.net/man/1/ifstat (accessed Dec. 19, 2021).

[312] X. Mertens, “Anonymous packet capture.” /dev/random, May 2008. https://blog.rootshell.be/2008/05/01/anonymous-packet-capture/ (accessed Dec. 19, 2021).

[313] B. Schmidt, J. González-Domínguez, C. Hundt, and M. Schlarb, “Chapter 2 - theoretical background,” in *Parallel Programming* (B. Schmidt, J. González-Domínguez, C. Hundt, and M. Schlarb, eds.), pp. 21–45, Morgan Kaufmann, 2018.

[314] D. M. Casas-Velasco, O. M. C. Rendon, and N. L. S. da Fonseca, “Drsir: A deep reinforcement learning approach for routing in software-defined networking,” *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.

[315] E. F. Castillo, O. M. C. Rendon, A. Ordonez, and L. Zambenedetti Granville, “Ipro: An approach for intelligent sdn monitoring,” *Computer Networks*, vol. 170, p. 107108, 2020.

[316] W. Villota, M. Gironza, A. Ordoñez, and O. M. Caicedo Rendon, “On the feasibility of using hierarchical task networks and network functions virtualization for managing software-defined networks,” *IEEE Access*, vol. 6, pp. 38026–38040, 2018.

Appendix A

Scientific Production

The research work presented in this thesis was reported to the scientific community through paper submissions to renowned conferences and journals. The process of doing research, submitting paper, gathering feedback, and improving the work helped to achieve the maturity hereby presented. The list of published papers to date are listed below in chronological order.

1. “A Framework for SDN Integrated Management based on a CIM Model and a Vertical Management Plane,” published in Computer Communications, 2017.

2. “SDN Management Based on Hierarchical Task Network and Network Functions Virtualization,” published in the proceedings of the 2017 IEEE Symposium on Computers and Communications (ISCC).

3. “A YANG Model for a vertical SDN Management Plane,” published in the proceedings of the 2017 IEEE Colombian Conference on Communications and Computing (COLCOM).

4. “Machine Learning for Cognitive Network Management,” published in IEEE Communications Magazine, 2018.

5. “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities,” published in Journal of Internet Services and Applications, 2018.

6. “An Efficient Mice Flow Routing Algorithm for Data Centers based on Software-Defined Networking,” published in the proceedings of the 2019 IEEE International Conference on Communications (ICC).

7. “NELLY: Flow Detection Using Incremental Learning at the Server Side of SDN-based Data Centers,” published in IEEE Transactions on Industrial Informatics, 2020.

8. “An Approach based on YANG for SDN Management,” published in International Journal of Communication Systems, 2021.

There is other paper that is still under construction.

1. “iRIDE: Rescheduling of Elephant Flows in SDN-based Data Centers Using Incremental Deep Learning to Predict Traffic Characteristics,” in construction.

The published papers are available in the next pages.