UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Enabling knowledge-defined networks: deep reinforcement learning, graph neural networks and network analytics

by

José Rafael Suárez-Varela Macià

Enabling Knowledge-Defined Networks:

Deep Reinforcement Learning, Graph Neural Networks and Network Analytics

A thesis submitted for the degree of

Doctor of Philosophy in Computer Architecture

by

José Rafael Suárez-Varela Maciá

Advisors:

Dr. Pere Barlet Ros

Dr. Albert Cabellos Aparicio

March 2020

Barcelona Neural Networking Center (BNN)
Departament d'Arquitectura de Computadors (DAC)
Universitat Politècnica de Catalunya (UPC)

Acknowledgments

This dissertation represents the work I have done during the last four years of my life. All this period has been an extremely enriching experience for me, both at the professional and the personal levels. During this amazing journey, I have met lots of people that have contributed –directly or indirectly– to this thesis. This is only a humble attempt to show my deep gratitude to some of them who deserve a special mention, for advising, supporting me, working with me, or just being there when I needed them most.

First and foremost, I would like to express my sincere gratitude, and also admiration, to my advisors: Prof. Pere Barlet, and Prof. Albert Cabellos. As exceptional researchers with large expertise, they have offered me the opportunity to work in a first-class research environment. Their always clever advice, clear vision, infinite motivation, and tireless perseverance have been fundamental pillars for the development of this thesis. This dissertation would not be the same without the countless, and sometimes endless, meetings that have guided my research along all this period. Today, I am very proud to say that they always tried to make the best of me. I will be eternally grateful for their firm support. *Thank you!*

During the elaboration of this thesis, I had the opportunity to meet many excellent researchers around the world. Specially, I would like to thank Prof. Franco Scarselli for his warm welcome during my research stay at the University of Siena. It was a very fruitful experience, both professionally and personally. I appreciate a lot all the interesting meetings where he shared with me his vast knowledge on Graph Neural Networks. Also, many thanks to the other members of his lab, the Siena Artificial Intelligence Lab (SAILab), for inviting me to their interesting seminars as well as to other scientific and social events. I will always remember this lovely trip where Prof. Scarselli took me to watch some of the most beautiful landscapes in the Tuscany region.

I would also like to express my profound gratitude to all the extraordinary researchers with whom I have collaborated in different works developed for this thesis. First, I would like to thank Dr. Albert Mestres and Dr. Krzysztof Rusek for sharing their extensive knowledge and expertise in machine learning, and for introducing me novel concepts of this field. Also, thanks to the other members of our current research team. Specifically, I truly thank Albert López and Paul Almasan, who have offered me an invaluable support for the development of some works during the last part of my thesis. I would like to highlight their tenacity and their determination to do their work rigorously. In general, thanks to all the team members for doing a great job and making teamwork so easy. Moreover, I am very grateful to Arnau Badia, Sergi Carol, and Prof. Marta Arias for their helpful collaboration in some works related to Graph Neural Networks. My gratitude also extends to Dr. Valentín Carela, with whom I had the pleasure of working in the early stages of this thesis. Many thanks also

to Junlin Yu, Dr. Li Kuang, and Haoyu Feng for their smart advice in the contributions related to Deep Reinforcement Learning.

My sincere appreciation also to the members of my PhD defense committee (Dr. David Carrera, Dr. Pedro Casas, Dr. Jérôme François, Dr. Jordi Perelló, and Dr. José Núñez), the external reviewers (Dr. Pedro Casas, and Dr. Jérôme François), and the pre-defense committee (Dr. Josep Solé-Pareta, Dr. Jordi Perelló, and Dr. David Carrera) for reading the manuscript and providing useful feedback.

Let me also acknowledge the great support from the members of the Broadband Communications Systems and Architectures (CBA) research group and the Computer Architecture Department in general. Here, I would like to make a special mention to Prof. Josep Solé-Pareta for following my progress during all the development of the thesis and providing me with wise advice that has contributed to improve the quality of this work.

Beyond the ones that have contributed to my research, there are many other colleagues who have shared my daily life in the D6-008 office, making my work more enjoyable, sharing the lunch time, helping me generously in what they could, and encouraging me in the most difficult moments. In this part, I could write a never-ending list of people. At least, I would like to acknowledge those with whom I have shared a long period and very relevant moments of my life in the office: Albert López, Albert Mestres, Arnau Badia, Giorgio Stampa, Hamidreza Taghvaee, Faezeh Zarrinkhat, Ismael Castell, Jordi Paillissé, Kurdman Rasol, Paul Almasan, and Sergi Abadal. Jordi, I wish you the best in the last stage of your thesis, and I hope we will continue having many interesting conversations while drinking coffee. To the newcomers, like Guillermo Bernárdez and Miquel Ferriol, I hope they will enjoy at least as much as I did this unique moment in their lives. *Thank you all!*

Lastly, I would like to acknowledge the Spanish Ministry of Economy and Competitiveness, particularly to the SUNSET project (ref. TEC2014-59583-C2-2-R), for funding my research during the four years I have been working for this thesis with a FPI grant (ref. BES-2015-073711). Also, thanks to the ALLIANCE project (ref. TEC2017-90034-C2-1-R), and the Generalitat de Catalunya (ref. 2017SGR-1037) for their support to the CBA research group.

A nivel personal, me gustaría agradecer a mi familia por su apoyo incondicional. A mis padres y a Joaquina, por ofrecerme todo lo necesario para crecer en un entorno de afecto, seguridad y bienestar, por inculcarme la curiosidad y la confianza necesarias para embarcarme en la realización de esta tesis. A mi hermana Marta, por su cariño, comprensión y sus inestimables consejos a lo largo de esta tesis y el resto de mi vida, tanto en lo personal como en lo profesional. Gracias a todos por vuestro apoyo, generosidad y consejos en mis experiencias vitales más importantes. Por último, pero no por ello menos importante, quisiera agradecer enormemente a Naile por acompañarme en momentos clave de esta tesis y de mi vida en Barcelona, por su cariño, por apoyarme, comprenderme y animarme, por estar ahí cuando más lo necesitaba. *¡Gracias!*

Abstract

Significant breakthroughs in the last decade in the Machine Learning (ML) field have ushered in a new era of Artificial Intelligence (AI). Particularly, recent advances in Deep Learning (DL) have enabled to develop a new breed of modeling and optimization tools with a plethora of applications in different fields like natural language processing, or computer vision.

In this context, the Knowledge-Defined Networking (KDN) paradigm highlights the lack of adoption of AI techniques in computer networks and – as a result – proposes a novel architecture that relies on Software-Defined Networking (SDN) and modern network analytics techniques to facilitate the deployment of ML-based solutions for efficient network operation.

This dissertation aims to be a step forward in the realization of Knowledge-Defined Networks. In particular, we focus on the application of AI techniques to control and optimize networks more efficiently and automatically. To this end, we identify two components within the KDN context whose development may be crucial to achieve self-operating networks in the future: (*i*) the *automatic control module*, and (*ii*) the *network analytics platform*.

The first part of this thesis is devoted to the construction of efficient automatic control modules. First, we explore the application of Deep Reinforcement Learning (DRL) algorithms to optimize the routing configuration in networks. DRL has recently demonstrated an outstanding capability to solve efficiently decision-making problems in other fields. However, first DRL-based attempts to optimize routing in networks have failed to achieve good results, often under-performing traditional heuristics. In contrast to previous DRL-based solutions, we propose a more elaborate network representation that facilitates DRL agents to learn efficient routing strategies. Our evaluation results show that DRL agents using the proposed representation achieve better performance and learn faster how to route traffic in an Optical Transport Network (OTN) use case. Second, we lay the foundations on the use of Graph Neural Networks (GNN) to build ML-based network optimization tools. GNNs are a newly proposed family of DL models specifically tailored to operate and generalize over graphs of variable size and structure. In this thesis, we posit that GNNs are well suited to model the relationships between different network elements inherently represented as graphs (e.g., topology, routing). Particularly, we use a custom GNN architecture to build a routing optimization solution that – unlike previous ML-based proposals – is able to generalize well to topologies, routing configurations, and traffic never seen during the training phase.

The second part of this thesis investigates the design of practical and efficient network analytics solutions in the KDN context. Network analytics tools are crucial to provide the control plane with a rich and timely view of the network state. However this is not a

trivial task considering that all this information turns typically into big data in real-world networks. In this context, we analyze the main aspects that should be considered when measuring and classifying traffic in SDN (e.g., scalability, accuracy, cost). As a result, we propose a practical solution that produces flow-level measurement reports similar to those of NetFlow/IPFIX in traditional networks. The proposed system relies only on native features of OpenFlow – currently among the most established standards in SDN – and incorporates mechanisms to maintain efficiently flow-level statistics in commodity switches and report them asynchronously to the control plane. Additionally, a system that combines ML and Deep Packet Inspection (DPI) identifies the applications that generate each traffic flow.

Resumen

La evolución del campo del Aprendizaje Máquina (ML) en la última década ha dado lugar a una nueva era de la Inteligencia Artificial (AI). En concreto, algunos avances en el campo del Aprendizaje Profundo (DL) han permitido desarrollar nuevas herramientas de modelado y optimización con múltiples aplicaciones en campos como el procesado de lenguaje natural, o la visión artificial.

En este contexto, el paradigma de Redes Definidas por Conocimiento (KDN) destaca la falta de adopción de técnicas de AI en redes y, como resultado, propone una nueva arquitectura basada en Redes Definidas por Software (SDN) y en técnicas modernas de análisis de red para facilitar el despliegue de soluciones basadas en ML.

Esta tesis pretende representar un avance en la realización de redes basadas en KDN. En particular, investiga la aplicación de técnicas de AI para operar las redes de forma más eficiente y automática. Para ello, identificamos dos componentes en el contexto de KDN cuyo desarrollo puede resultar esencial para conseguir redes operadas autónomamente en el futuro: (*i*) el módulo de control automático y (*ii*) la plataforma de análisis de red.

La primera parte de esta tesis aborda la construcción del módulo de control automático. En primer lugar, se explora el uso de algoritmos de Aprendizaje Profundo por Refuerzo (DRL) para optimizar el encaminamiento de tráfico en redes. DRL ha demostrado una capacidad sobresaliente para resolver problemas de toma de decisiones en otros campos. Sin embargo, los primeros trabajos que han aplicado DRL a la optimización del encaminamiento en redes no han conseguido rendimientos satisfactorios. Frente a dichas soluciones previas, proponemos una representación más elaborada de la red que facilita a los agentes DRL aprender estrategias de encaminamiento eficientes. Nuestra evaluación muestra que cuando los agentes DRL utilizan la representación propuesta logran mayor rendimiento y aprenden más rápido cómo encaminar el tráfico en un caso práctico en Redes de Transporte Ópticas (OTN). En segundo lugar, se presentan las bases sobre la utilización de Redes Neuronales de Grafos (GNN) para construir herramientas de optimización de red. Las GNN constituyen una nueva familia de modelos de DL específicamente diseñados para operar y generalizar sobre grafos de tamaño y estructura variables. Esta tesis destaca la idoneidad de las GNN para modelar las relaciones entre diferentes elementos de red que se representan intrínsecamente como grafos (p. ej., topología, encaminamiento). En particular, utilizamos una arquitectura GNN específicamente diseñada para optimizar el encaminamiento de tráfico que, a diferencia de las propuestas anteriores basadas en ML, es capaz de generalizar correctamente sobre topologías, configuraciones de encaminamiento y tráfico nunca vistos durante el entrenamiento.

La segunda parte de esta tesis investiga el diseño de herramientas de análisis de red eficientes en el contexto de KDN. El análisis de red resulta esencial para proporcionar al plano de control una visión completa y actualizada del estado de la red. No obstante, esto no es una tarea trivial considerando que esta información representa una cantidad masiva de datos en despliegues de red reales. Esta parte de la tesis analiza los principales aspectos a considerar a la hora de medir y clasificar el tráfico en SDN (p. ej., escalabilidad, exactitud, coste). Como resultado, se propone una solución práctica que genera informes de medidas de tráfico a nivel de flujo similares a los de NetFlow/IPFIX en redes tradicionales. El sistema propuesto utiliza sólo funciones soportadas por OpenFlow, actualmente uno de los estándares más consolidados en SDN, y permite mantener de forma eficiente estadísticas de tráfico en conmutadores con características básicas y enviarlas de forma asíncrona hacia el plano de control. Asimismo, un sistema que combina ML e Inspección Profunda de Paquetes (DPI) identifica las aplicaciones que generan cada flujo de tráfico.

Contents

List of Figures

Page

List of Tables

Page

List of Acronyms

ACER Actor Critic with Experience Replay

AI Artificial Intelligence

API Application Programming Interface

AVI Audio Video Interleave

A3C Asynchronous Advantage Actor Critic

BU Bandwidth Unit

BGP Border Gateway Protocol

CDF Cumulative Distribution Function

CNN Convolutional Neural Network

CPU Central Processing Unit

DDPG Deep Deterministic Policy Gradients

DHCP Dynamic Host Configuration Protocol

DL Deep Learning

DNN Deep Neural Network

DNS Domain Name System

DoS Denial-of-Service

DPI Deep Packet Inspection

DRL Deep Reinforcement Learning

EB exabytes

EON Elastic Optical Network

FEC Forward Error Correction

FQDN Fully Qualified Domain Name

FSD Flow Size Distribution

GAE Generalized Advantage Estimator

GAN Generative Adversarial Network

GBN German Backbone Network

GNN Graph Neural Network

GRU Graph Recurrent Unit

GPU Graphics Processing Unit

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IDS Intrusion Detection System

IETF Internet Engineering Task Force

IoT Internet of Things

IP Internet Protocol

IPFIX Internet Protocol Flow Information Export

ISP Internet Service Provider

KDN Knowledge-Defined Networking

KPI Key Performance Indicator

LDP Lowest Delay Path

LISP Locator/ID Separation Protocol

LSTM Long Short-Term Memory

MARL Multi-Agent Reinforcement Learning

MDP Markov Decision Process

ML Machine Learning

MPNN Message-Passing Neural Network

MSE Mean Squared Error

M2M Machine-to-Machine

NN Neural Network

NFV Network Function Virtualization

NLP Natural Language Processing

NOS Network Operating System

NSF National Science Foundation

ODU Optical Data Unit

ONF Open Network Foundation

OSPF Open Shortest Path First

OTN Optical Transport Network

OTU Optical Transport Unit

PCL Path Consistency Learning

PPO Proximal policy optimization

P2P Peer-to-Peer

QoS Quality of Service

QUIC Quick UDP Internet Connection

RAM Random Access Memory

RL Reinforcement Learning

RMSA Routing, Modulation and Spectrum Assignment

RNN Recurrent Neural Network

ROADM Reconfigurable Optical Add-Drop Multiplexer

RTP Real-time Transport Protocol

RTT Round-Trip Time

SAP Shortest Available Path

SDN Software-Defined Networking

SELU Scaled Exponential Linear Unit

SGD Stochastic Gradient Descent

SLA Service-Level Agreement

SMTP Simple Mail Transfer Protocol

SNI Server Name Indication

SNMP Simple Network Management Protocol

SSH Secure Shell

SSL Secure Sockets Layer

SVM Support Vector Machines

TCAM Ternary Content-Addressable Memory

TLS Transport Layer Security

TRPO Trust Region Policy Optimization

TCP Transmission Control Protocol

UDP User Datagram Protocol

VNI Visual Networking Index

VoIP Voice over Internet Protocol

VXLAN Virtual Extensible Local Area Network

WAN Wide Area Network

WMRD Weighted Mean Relative Difference

Chapter 1

Introduction

Since its inception, the Internet has become increasingly complex and hard to manage, and this trend will be reinforced by the gradual adoption of emerging service paradigms such as big data, Internet of Things (IoT), virtualized cloud computing, or multimedia content delivery. One fundamental problem of the nowadays' Internet is that it undergoes a severe "ossification" problem [2, 3]. During decades, networks have relied heavily upon the well-established TCP/IP stack, and deficiencies have been solved with incremental *ad hoc* changes over this protocol stack [4]. This strategy – albeit effective to adapt to the rapid growth of the Internet – has brought many limitations to deploy new standards, or even to add extra functionality to the existing ones (e.g., TCP, UDP). One example of this is the massive use of middleboxes, which has frustrated later efforts to deploy new protocols that do not rely on native features of TCP or UDP. As a result, the current "ossified" Internet architecture imposes a great barrier to adopt disruptive technology that may help satisfy the upcoming operational demands of networks [3].

The next few years will see an ever-increasing growth in the scale of the Internet. According to the Cisco Visual Networking Index (VNI) of 2019 [5], the overall IP traffic is expected to reach 396 exabytes (EB) per month by 2022, up from 122 EB – per month – in 2017. Particularly, IoT-related applications including smart meters, video surveillance, healthcare monitoring, or transportation will play an important role. The same Cisco VNI report estimates that, by 2022, there will be around 14.6 billion Machine-to-Machine (M2M) connections, which will involve roughly 51% of the worldwide devices and connections. In this vein, the Internet will witness the emergence of more and more high-demand applications with a great variety of stringent Quality of Service (QoS) requirements, ranging from ultra-low latency (e.g., immersive online gaming) to extremely high reliability (e.g., telesurgery, emergency response) [6]. All this presents a grand challenge to networking equipment manufacturers and service providers, who advance serious difficulties to meet the demands of future applications with the current network technology.

In order to reverse this situation, the Software-Defined Networking (SDN) paradigm [7, 8] proposed a decade ago a novel network architecture intended to add more flexibility in network management and troubleshooting tasks. Inspired by the evolution of the software field, SDN propounds a logical separation between the control and data planes of networks, and defines a standard API to communicate these two "layers". In this well-established architecture, the data plane forwards the user traffic over the network topology, while the control plane is in charge of the network operation. Particularly, the control plane comprises a set of SDN controllers that work in collaboration and run software to perform different network operation tasks. Likewise, the data plane contains programmable network devices that can be dynamically configured from the control plane, which offers a great flexibility to implement and combine diverse networking policies. This – unlike in traditional networks – enables to deploy easily new standards just by updating the software in the centralized controllers, without requiring any hardware modification.

Since SDN was first proposed – around 2008 –, it has gained a lot of support from both the academia and industry, and nowadays networks are experiencing a relentless paradigm shift towards software-defined architectures embracing also concepts like Network Function Virtualization (NFV) [9] and cloud networking. This is – for instance – reflected in the Cisco VNI report [5], which forecasts that SDN deployments for Wide Area Networks (SD-WAN) will increase 5-fold from 2017 to 2022, and are expected to involve around 29% of the global WAN traffic by the end of 2022.

While computer networks have witnessed the so-called "softwarization" process, a series of breakthroughs in the Machine Learning (ML) field during the last decade have marked the beginning of a new era of Artificial Intelligence (AI). More specifically, recent advances in Deep Learning (DL) [10] have enabled to produce a new breed of modeling tools with many revolutionary applications in fields like Natural Language Processing (NLP) [11, 12], computer vision [13, 14], biomedicine [15, 16], information retrieval [17, 18], or automated control (e.g., self-driving cars [19, 20]). Also, DL models in combination with optimization strategies, such as Reinforcement Learning (RL) algorithms, have opened the possibility to efficiently solve complex decision-making and automated control problems [21, 22]. All this has attracted a lot of interest from the networking community, which has recently started to investigate how to build cost-efficient ML-based solutions to address network-related problems like routing optimization, performance prediction, or traffic classification [23, 24].

However, despite the much enthusiasm to adopt AI technology in networks, at the time of this writing ML-based solutions have been barely deployed in real-world networks. In such a context, the Knowledge-Defined Networking (KDN) paradigm emerged in 2016 in order to invigorate the adoption of ML techniques applied to network control and operation. In this work, the authors expose the main reasons for the lack of adoption of AI in traditional networks, and claim that the recent rise of SDN together with the latest advances in network analytics may be leveraged to deploy efficiently ML-based systems in networks.

For this purpose, KDN revisits the concept of a knowledge plane for networks – proposed by D. D. Clark *et al.* in 2003 –, and presents a renewed architecture that unifies SDN, network analytics, and AI. With this architecture, KDN facilitates the construction of a functional knowledge plane running AI-based solutions to operate networks in a more efficient and automatic way, all this with the ultimate goal of achieving self-driving networks in the future. Since it was conceived, the networking community has received KDN with increasing interest and – in line with its objectives – it has already served as a reference paradigm for later research endeavors to deploy AI techniques for network automation [25–29].

The objective of this dissertation is to go further in the realization of future Knowledge-Defined Networks. To this end, we first make a thorough analysis of the main components within the KDN architecture and identify some technologies that may be key enablers to achieve self-operating networks in the future. Thereafter, we present a series of contributions that – we believe – will represent a significant advance for the construction of several key components of future knowledge-based networks.

To put the reader in context, the remainder of this introduction chapter is structured as follows. Section 1.1 provides essential background on SDN and its evolution towards the KDN proposal. Then, in Section 1.2 an analysis over the KDN architecture is made to describe the principal network components involved in automatic network operation. Lastly, Section 1.3 highlights the main contributions of this thesis and outlines the content in the remainder of the document.

1.1. Background

This section provides an overview about the inception of the SDN paradigm and how it has evolved over time. Likewise, it serves as a motivation to introduce subsequently the KDN paradigm as an extended architecture inherited from SDN that is specifically designed to embrace AI technology in networks.

1.1.1. Software-Defined Networking

The Software-Defined Networking (SDN) paradigm has its roots in some research works like RCP (Routing Control Platform) [30], 4D [31] or Ethane [32], which posited the idea of separating the control and data planes of networks as a practical way to experiment with new protocols. However, many authors determine its origin in 2008, with the proposal of the OpenFlow protocol [2], which has become its major driver. Strictly, the usage of the term "SDN" itself was coined in 2009 in an article about the OpenFlow project [33].

From its inception, SDN has gained a lot of attention from academia and industry, being supported by big players of the Internet like Google, Cisco, HP, Juniper, or NEC and by standardization organizations like the Open Network Foundation (ONF) or the Internet Engineering Task Force (IETF). As a result, computer networks are progressively shifting

towards SDN-based architectures. In this context, we have already witnessed the construction of some large-scale SDN deployments for Wide Area Networks (WANs) and data center networks that have had a great impact among the networking community [34–36].

In the OpenFlow paper [2], the authors talk about the commonly held belief that the Internet was "ossified", and highlight that traditional networks with distributed management are inflexible due to the hardwired implementation of forwarding rules in the networking devices. This has turned into a continuous frustration for network operators and networking equipment vendors, who have needed to apply incrementally *ad hoc* solutions to increase flexibility in network management and support widely demanded use cases [37]. To reverse this situation, the authors of OpenFlow proposed a novel network architecture that decouples logically and physically the control and data planes of networks, and defines a standard interface that uses the OpenFlow protocol to communicate both planes.

Figure 1.1 depicts an scheme of the SDN architecture. In this new paradigm, the data plane contains programmable forwarding devices, while the control plane is logically centralized and managed by some entities called SDN controllers, that work together to operate the network. Likewise, it creates an abstraction of the forwarding devices and provides a standard interface – the Southbound API – to access them from the control plane.

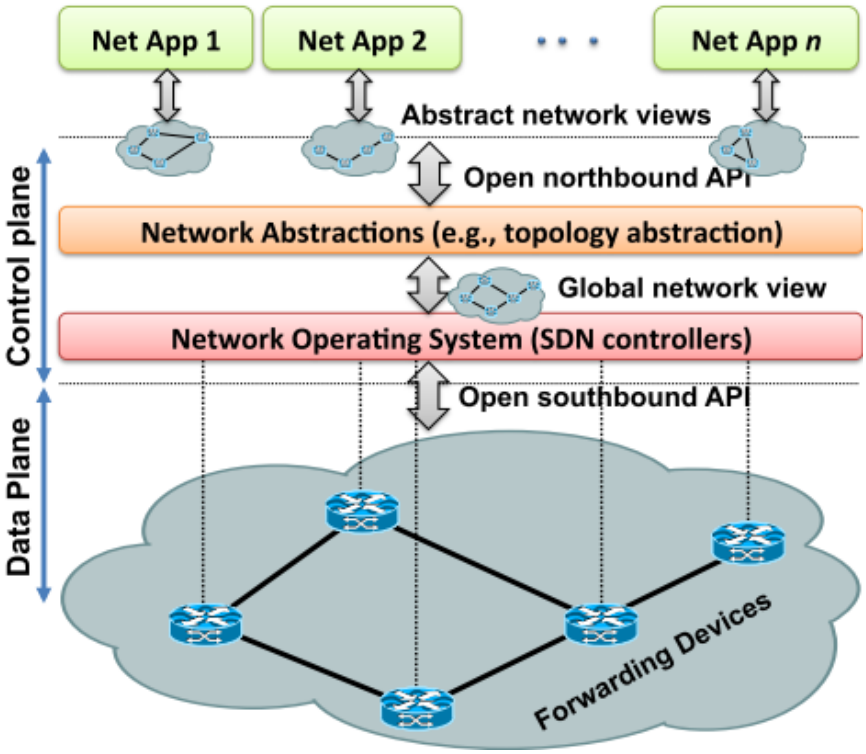


Figure 1.1: Software-Defined Networking architecture.
Source: D. Kreutz *et al.*, "Software-Defined Networking: A comprehensive survey" Proceedings of the IEEE, vol. 103, no. 1, pp. 14-76, 2015.

The decoupling of the two planes introduces the benefits of a centralized approach for network operation, and enables to modify the network policy directly from the control plane. In traditional networks – also referred to as "legacy" networks – forwarding devices need to be configured individually using vendor-specific codes. In contrast, in SDN, network administrators can apply new high-level policies from SDN controllers, and these controllers are in charge of translating the new policies into forwarding rules that are eventually installed in the data-plane devices. This – in analogy with the software field – allows network operators to develop applications for different networking tasks being oblivious of the characteristics of the devices in the underlying infrastructure. To this end, SDN controllers run a Network Operating System (NOS) that supplies a Northbound API with high-level network abstractions to applications running on top it. The NOS concept was first introduced with the NOX controller [38], and thereafter a plethora of proposals such as OpenDaylight [39], ONOS [40], POX [41], Ryu [42], Floodlight [43], Beacon [44], Onix [45] or Maestro [46] have appeared in the literature. In this extensive catalog of operating systems, we can find a wide variety of designs specifically optimized for different purposes (e.g., scalability, performance, fault tolerance, fast prototyping). Additionally, some high-level programming languages have been already proposed to achieve practical Northbound APIs. For instance, Frenetic [47] and Procera [48] propose declarative languages where network operators define high-level policies that can be automatically translated into OpenFlow rules for the data-plane devices.

The forwarding equipment in SDN is target-independent, which means that data-plane devices are not limited to perform specific networking tasks (e.g., router, L2 switch, firewall). It opens the possibility to reconfigure – via software – the topology and dynamically change the roles of the data-plane devices at runtime. Moreover, SDN provides support to maintain traffic statistics in data-plane devices and collect them in SDN controllers. This enables to maintain a timely and accurate global view of the network state in the control plane, which can be very beneficial to achieve centralized and efficient network management. As for the Southbound API, since OpenFlow [2] was proposed, multiple contemporary proposals have been made to define alternative standards for the communication between the control and data planes (e.g., P4 [49], LISP [50], BGP [51]).

In summary, software-defined networks offer a level of flexibility in network control and operation never seen before in legacy networks. This enables to perform fine-grained network management in order to deal with the operational demands of today's networks, which need to satisfy applications with very diverse QoS requirements and adapt to adverse conditions such as highly fluctuating traffic.

1.1.2. Knowledge-Defined Networking

The Knowledge-Defined Networking (KDN) paradigm [52] emanates from the idea of fostering the deployment of AI techniques in networks. To this end, KDN restates the concept

of a knowledge plane for networks already proposed by D. D. Clark *et al.* in 2003 [53]. As it was initially proposed, the knowledge plane is a construct that combines AI tools and cognitive systems to control and operate the network efficiently. It should incorporate mechanisms that respond to changes in the network in better-than-human timescales and optimize the resource use in complex network environments where humans and analytical solutions fail to achieve good performance.

Despite the knowledge plane concept has aroused much interest from the networking community along its history, at the time of this writing no functional prototype or real-world deployment has been implemented yet. According to the authors of KDN, one of the main limitations to deploy AI techniques in nowadays' networks is that they are intrinsically distributed. It means that network devices have only a partial view of the network state, and their actions have an impact only on a small portion of the network – typically on their neighborhood. One example of this is routing in legacy networks, where forwarding devices are limited to select the next hop based on their local state. In this vein, KDN highlights the recent rise of two technologies that may act as a catalyst to construct a functional and efficient knowledge plane: (*i*) the SDN paradigm and (*ii*) modern network analytics techniques. First, in SDN the control plane permits to gather knowledge about the network state in a single centralized point. This may bring many advantages for modern ML-based network operation solutions [23, 24], which can leverage the global state information to make decisions over the network as a whole. Second, data-plane devices in SDN offer improved computing and storage capabilities with respect to traditional networking equipment. This paves the way to develop a new breed of monitoring techniques – also known as telemetry [54, 55] – that enable to maintain measurements in data-plane devices and collect timely information about the network state in a centralized platform. At this point, network analytics techniques are essential to collect, structure, process, and maintain efficiently the network state information, which in real-world networks typically turns into big data. Moreover, KDN introduces recent advances in the AI field – and specifically in Deep Learning – as the third pillar to construct the knowledge plane. Thus, they state that AI techniques, together with the centralized control capability enabled by SDN, and the rich global view supplied by network analytics, may enable to operate networks more efficiently and automatically.

In light of the above, KDN proposes a novel paradigm that accommodates and leverages SDN, network analytics and AI in an unified architecture. As a result, they present the operational loop depicted in Figure 1.2. This loop can be exploited for network control and operation in two different ways – closed loop and open loop – depending on whether there is human intervention or not in the process. The closed-loop mode is intended to automate network operation tasks (i.e., recognize-act). This includes automatic decision making and real-time optimization of the network configuration (e.g., routing, security policy). Alternatively, the open-loop operation mode is aimed at the implementation of

recommendation systems that can be leveraged by network administrators to make better decisions (i.e., recognize-explain-suggest). This – for instance – permits to perform what-if analysis, estimate performance metrics (e.g., loss, delay) given a network state snapshot, or validate a configuration before implementing it in the network. In both operation modes, the use of ML-based techniques such as supervised and unsupervised Deep Learning [10], or Reinforcement Learning [21,22] may be a fundamental component to produce either recommendations for network operators (open loop) or a set of actions that can be directly applied on the network (closed loop).

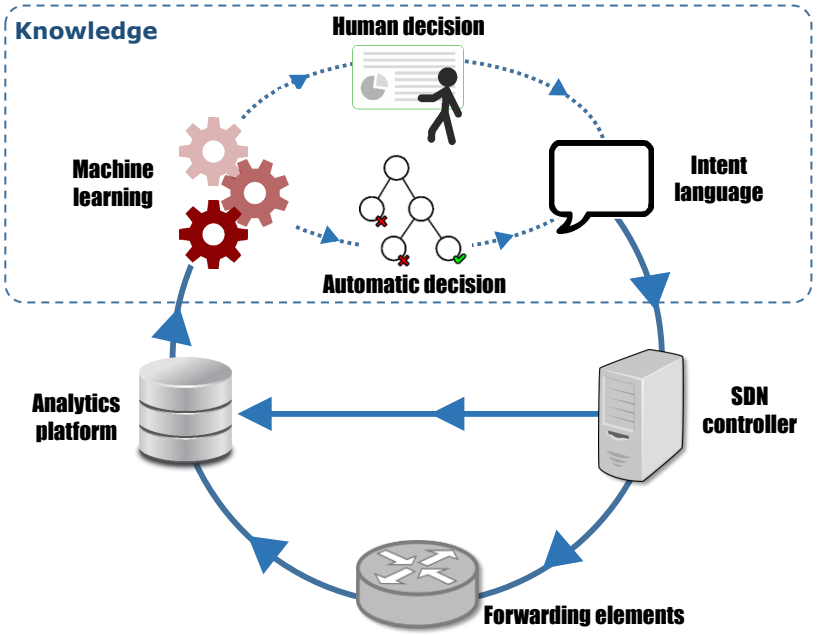


Figure 1.2: KDN operational loop.
Source: A. Mestres *et al.*, "Knowledge-Defined Networking"
ACM SIGCOMM Computer Communication Review, vol. 47, no. 3, pp. 2-10, 2017.

1.2. Motivation: Enabling automation in Knowledge-Defined Networking

This dissertation puts the spotlight on enabling automation in Knowledge-Defined Networks. For this purpose, the operational loop presented in the original KDN paper (Fig. 1.2) is analyzed and redefined to consider only those network components that are involved in automatic network operation processes (recognize-act). As a result, the simplified scheme in Figure 1.3 is presented. As we can observe, this workflow consists of four main network components: (*i*) a network analytics platform, (*ii*) an automatic control module, (*iii*) a cluster of network actuators, and (*iv*) the network infrastructure. This section describes

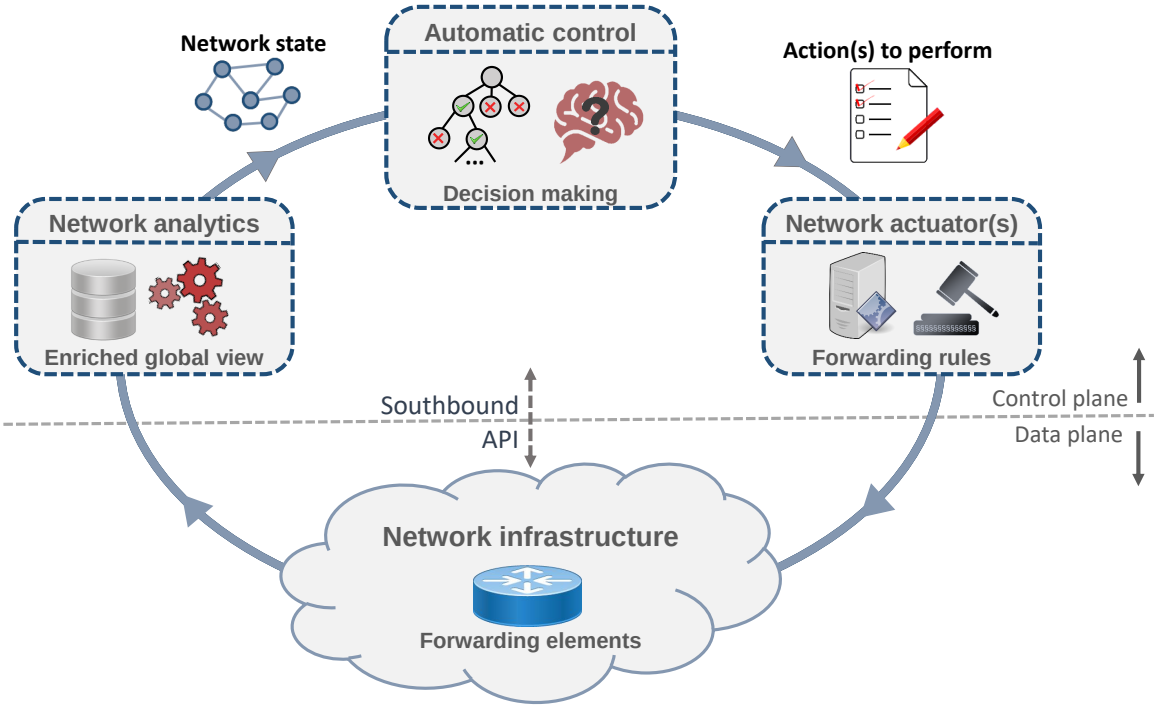


Figure 1.3: Operational loop for automatic control in Knowledge-Defined Networks.

all these components and outlines some relevant research avenues that may be particularly useful to address in order to achieve self-operating networks in the future.

Network analytics platform:

Network analytics encompasses a variety of network monitoring and big data processing techniques that are intended to provide an enriched view of the network state, all this with the aim of assisting network operation and troubleshooting tasks. This enables to automate and improve networking tasks such as fine-tuning the network configuration, predicting traffic and application trends, or preventing potential problems like performance degradation or security breaches.

The huge scale and diversity of nowadays' networks makes it difficult to measure and maintain accurate and timely statistics of the network state. In this context, the SDN paradigm offers the advantage of a data plane populated by devices with enhanced computing and storage capabilities, as well as a centralized control plane that permits to gather all the state information maintained in data-plane devices. For instance, OpenFlow [2] offers support to maintain flow-level traffic measurements in forwarding devices (e.g., traffic volume, flow duration) and provides an API to report this information to SDN controllers.

However, despite SDN solves some classical problems of network measurement in distributed environments, it brings new challenges to address. The decoupling of the control and data planes adds new implications that need to be identified and considered for the

design of efficient network analytics solutions. For instance, this separation introduces a latency in the communication between the control and data planes (i.e., between SDN controllers and forwarding devices). Thus, this latency is not only affected by the delay of the connection itself, but also by other factors like the current workload of the devices and their availability. Likewise, the fact that SDN controllers are centralized entities that typically manage many forwarding devices, makes them prone to become bottlenecks. This adds the need to prevent possible scalability issues by carefully selecting the tasks that are processed in controllers and those that may be devolved to data-plane devices.

In addition, the use of modern ML and big data processing techniques enables to provide a deep insight into all the information collected from the network. In this vein, one application that may be particularly beneficial for network operation is to classify the traffic by applications. This can be done by using supervised ML methods such as decision trees, Support Vector Machines (SVM), or Deep Neural Networks, which can be trained with labeled data to then classify the applications from a limited view of the network traffic. For instance, some works leverage basic flow-level traffic measurements (e.g., traffic volume) to discover – with ML – the applications in the network [56, 57]. Other applications where ML can provide an added value for network analytics are the following: anomaly detection to prevent security problems (e.g., DoS attacks) [58], root cause analysis to quickly restore the normal state of the network [59], or traffic prediction to devise long-term strategies for network planning and optimization [60]. In the latter case, unsupervised ML methods can also be applied to find correlations on the data, that may – for instance – help forecast the organic growth of the network traffic.

Automatic control module:

This module probably represents the cornerstone for achieving automatic operation in networks. An efficient implementation of such a component should be able to make real-time management decisions autonomously and modify the configuration to adapt to changes in the network state. As an example, it could serve to perform automatically tasks like adaptive traffic routing, failure recovery (e.g., when a link fails), or optimal resource allocation in Network Function Virtualization (NFV) scenarios. To this end, KDN envisions the use of optimization algorithms together with ML techniques to build self-driving solutions for network control and operation. Naturally, the performance of this module will also depend largely on the quality of the data provided by the network analytics platform.

One fundamental issue in optimization problems in general is the need to have a model capable of understanding the input variables of the environment in which it operates and how they relate to the resulting performance. For example, in the networking context, to find the optimal routing configuration that minimizes the delay, it is necessary a model able to predict how the delay relates to other network characteristics like the topology, the traffic, or the routing configuration. In this line, ML techniques – and in particular

DL models – seem to be a suitable solution to build cost-efficient network models. The outstanding capability of DL models to abstract deep knowledge from data makes them appropriate to discover the complex relationships between different network characteristics and learn how they affect to different Key Performance Indicators (KPIs) of networks (e.g., delay, jitter, packet loss).

Likewise, automatic network optimization can be achieved by combining network models with any optimization strategy. For instance, one straightforward approach to build an optimizer is to combine the network model with a classic optimization algorithm (e.g., hill climbing). Alternatively, recent advances in the AI field have opened the possibility of combining Reinforcement Learning (RL) algorithms with DL models in order to create a new breed of optimization techniques commonly known as Deep Reinforcement Learning (DRL). These methods have already demonstrated an unprecedented capability to efficiently solve complex decision-making problems in other fields [21, 22]. However, despite some early attempts to perform DRL-based optimization in classic networking problems (e.g., routing optimization [61,62]), we are still far from achieving practical and reliable DRL agents that can be deployed in real-world networks.

Additionally, the use of modern unsupervised DL techniques like Generative Adversarial Networks (GANs) [63] or auto-encoders [64] may play an important role in assisting network optimization tasks in future knowledge-based networks.

Last but not least, some efforts should be made in the direction of creating high-level abstractions of this module. In this line, declarative languages (e.g., Procera [48], NeMo [65]) may be suitable to define an API for network operators to select the target policy. Then, the module should incorporate some mechanisms to translate automatically high-level policies into well-defined (multi-objective) optimization problems.

Network actuator(s):

In the architecture presented in Figure 1.3, network actuators are entities that act as intermediaries between the automatic control module and the data plane. In particular, their main purpose is to process the actions generated by the control module and translate them into sets of forwarding rules to be installed in data-plane devices. To this end, they rely on a southbound API (e.g., OpenFlow [2], P4 [49]) that allows them to communicate with the SDN-enabled devices of the data plane. Besides, developing a standard API for the communication with the control module may help create an abstraction of network actuators implementing different southbound APIs. This, in turn, would enable to make cross-platform implementations for automatic control modules.

Note that, the scheme in Figure 1.3 presents four logically independent network components. However, the network analytics platform, the automatic control module, and network actuators are part of the control plane and, thereby, may be implemented within SDN controllers.

Network infrastructure:

Lastly, the network infrastructure comprises a set of interconnected forwarding devices that shape the network topology. As such, it is ultimately responsible for routing the user traffic along the network. The design of the internal architecture of forwarding devices is crucial in two different ways: (*i*) to achieve high performance in traffic forwarding, and (*ii*) to provide an appropriate forwarding abstraction accessible from the control plane. Note that the design of this architecture is tightly coupled to the definition of the Southbound API, and typically requires a painstaking designing and standardization process until a functional and well-established proposal is achieved. So far, only a few architectural designs deserve to be mentioned. The first remarkable switch architecture came with the proposal of the OpenFlow protocol [2], where the authors propose to use flow tables as an abstraction of data-plane devices. This permitted to create a flexible Southbound API – implemented by the OpenFlow protocol – that manages traffic at a flow-level granularity. More recently, proposals like PISA (Protocol Independent Switch Architecture) [66] and its successor PSA (Portable Switch Architecture) [67] were presented within the context of the P4 protocol [49], and represent an ongoing effort to achieve more flexible and transparent abstractions of the data plane.

1.3. Contributions and Outline of the Thesis

This section presents a summary of the scope and main contributions of this thesis, as well as a brief description of the remaining chapters of the document.

The work presented in this dissertation has been developed under the framework of the Knowledge-Defined Networking (KDN) paradigm. In particular, our efforts are directed towards enabling full automation on network control and operation tasks. To this end, the operational loop originally presented in KDN was reformulated to consider only the components involved in network automation processes. As a result, the scheme in Figure 1.3 was presented along with a description of all its network components and some key challenges that may be particularly practical to address in order to achieve self-operating Knowledge-Defined Networks (see Section 1.2).

This thesis focuses specifically on the development of two network components that – we believe – represent the main enablers for achieving automatic operation in KDN: (*i*) the *automatic control module*, and (*ii*) the *network analytics platform*. The automatic control module represents in itself the automated execution of network operation tasks, while the network analytics platform is essential to provide the control module with a complete and meaningful view of the network state, which paves the way to achieve efficient network operation. These two components give name to the two core parts in which this document is divided. Note that we leave out of the scope of this thesis the design of practical and efficient network actuators and switch architectures. However, despite the progress

made in this direction [2, 49, 67], much remains to be done to achieve – and standardize – flexible implementations of the data plane able to meet the operational demands that future applications and services will require.

The remainder of this thesis is structured as follows:

Part I: Enabling Automatic Control

This part delves into the construction of a functional and efficient automatic control module. In particular, we divided it in two chapters that respectively represent two main contributions where we borrow some cutting-edge AI technologies with remarkable applications in other fields and adapt them to address classic network optimization problems. This has led to achieve efficient optimization tools with outstanding performance in different well-known network optimization use cases, such as traffic routing, network planning, or failure recovery.

Chapter 2 - Deep Reinforcement Learning-based Network Optimization

This chapter explores the application of modern *Deep Reinforcement Learning* (DRL) techniques [21, 68] to address network optimization problems. Particularly, we focus on optimizing the routing configuration in networks. In this context, existing DRL-based routing optimization solutions fail to achieve good performance, often being outperformed by traditional heuristics. Thus, we propose a novel representation of the network state that allows DRL agents to achieve better performance and learn faster how to route the traffic in networks. This chapter includes an extensive evaluation against state-of-the-art DRL-based proposals in a routing use case in Optical Transport Networks (OTNs).

Chapter 3 - Applying Graph Neural Networks to Network Optimization

In this chapter, we investigate the application of *Graph Neural Networks* (GNNs) [69, 70] for the automatic operation of computer networks. First, it introduces the fundamentals of GNN, a recently proposed family of Deep Learning models specifically tailored to operate over graph-structured data. Thereafter, we present a custom GNN architecture that is utilized to address different classic network optimization problems. The GNN model presented is able to accurately predict the per-source-destination delay and jitter given an input topology, a routing configuration, and a traffic matrix. Thus, we integrated this model into an optimizer and evaluated it in five different network optimization use cases, including delay-aware routing optimization, failure recovery, and budget-constrained network upgrade. In this vein, our work pioneers the use of GNNs to build network optimization tools that – unlike previous ML-based proposals – are able to successfully generalize to variable-size topologies, routing configurations, and traffic unseen in advance by the model.

Part II: Network Analytics

The second part of this thesis delves into the realization of a practical network analytics platform within the KDN context. Network analytics is crucial to provide the control plane with a complete and timely picture of the network state. To this end, it is necessary to count on efficient mechanisms to process all the state information collected from the network, as it typically involves a massive amount of data in large-scale real-world network deployments.

Chapter 4 - Flow-level Measurement and Classification in SDN

This chapter analyzes the main aspects that should be considered to achieve efficient traffic measurement and classification systems in SDN (e.g., scalability, accuracy, cost). In this context, we propose a practical network monitoring solution that produces flow-level measurement reports as those provided by NetFlow/IPFIX in legacy networks. The proposed system uses only features supported by OpenFlow, and provides mechanisms to maintain flow-level statistics in the switches and report them asynchronously to SDN controllers. Moreover, for the sake of scalability, this system implements two different traffic sampling methods depending on the OpenFlow features available in the switches. Additionally, we combine ML and Deep Packet Inspection (DPI) techniques to classify the traffic also at the flow-level granularity, with special attention on the applications within web and encrypted traffic. Finally, we implemented the proposed system in the OpenDaylight SDN controller [39], and made an extensive evaluation of it with real-world network traffic.

Chapter 5 - Conclusions and Future Work

This last chapter concludes the dissertation and summarizes some paths that could be explored for future work.

Additionally, **Appendix A** provides a complete list of all the publications derived from the work done during the elaboration of this thesis.

Part I

Enabling Automatic Control

Chapter 2

Deep Reinforcement Learning-based Network Optimization

Recent advances in Deep Reinforcement Learning (DRL) are providing a dramatic improvement in decision-making and automated control problems. As a result, we are witnessing a growing number of research works that are proposing ways of applying DRL techniques to network-related problems such as routing optimization. However, existing proposals fail to achieve good results, often under-performing traditional routing techniques. We argue that successfully applying DRL-based techniques to networking requires finding good representations of the network parameters: *feature engineering*. DRL agents need to represent both the state (e.g., link utilization) and the action space (e.g., changes to the routing policy). In this chapter, we show that existing approaches use straightforward representations that lead to poor performance. We propose a novel representation of the *state* and *action* that outperforms existing ones. We made an evaluation of this representation in two different uses cases: (*i*) routing in Optical Transport Networks (OTNs), and (*ii*) a simple QoS-aware routing scenario in IP networks. The evaluation results show that using our representation, DRL agents achieve better performance and learn how to route traffic significantly faster compared to state-of-the-art DRL-based network representations.

2.1. Introduction

In the last few years, we have witnessed significant advances in DRL that are revolutionizing the way we can resolve decision-making and automated control problems [21, 22]. As a result, the networking community has recently started to investigate the potential of such techniques to solve network-related problems, such as IP routing [71], routing in optical

networks [61], Quality of Service (QoS) provisioning [72] or job scheduling [73]. These recent proposals aim to build self-driving networks [52] by designing agents based on Machine Learning (ML) that operate the network autonomously.

Network routing is a particularly interesting problem for the application of DRL techniques. Indeed, the configuration of optimal routes in a network is arguably one of the most fundamental and well studied problems in the field of networking. This problem has been addressed using traditional traffic engineering techniques [74], which rely on complex optimization algorithms with simple delay models. In contrast, DRL and modern Deep Learning (DL) techniques enable to operate under complex and non-linear models. However – at the time of this writing – existing DRL-based proposals still fail to achieve good results given their lack of generalization capability. This means that they are not able to make correct decisions when facing network scenarios not explored during the training phase. As a result, they often under-perform traditional routing techniques based on simple heuristics. In this chapter, we argue that the main reason behind this poor performance is that these proposals apply DRL as a black-box using *straightforward representations* for both the observation and the action space.

In DRL algorithms, two main elements must be defined: (*i*) the *observation space* and (*ii*) the *action space*. The observation space describes the state of the environment (i.e., the current state of the network in our case). The action space, on the other hand, describes the modifications that the DRL agent makes over the environment. In our context, the action represents changes to be applied to the routing configuration. The network state is typically represented as a matrix containing the per-link utilization [61]. Given the high dimensionality of the output (i.e., possible routing configurations), existing proposals usually limit the action space by using straightforward representations, such as the per-link weights for link-state routing algorithms [e.g., Open Shortest Path First (OSPF)] [29, 71].

We argue that networking problems are fundamentally different from other recent problems where DRL techniques have been successfully applied (e.g., [22]). We show that, in order to outperform traditional algorithms in the networking context, it is not enough to leverage recent advances in DRL algorithms as done in previous works, but it is even more important to carefully design good representations of the state and action spaces that can better represent the singularities of network topologies and simplify the learning process to the DRL agent. We refer to this design process as *feature engineering*.

An important limitation of existing representations is that they do not achieve good generalization capability. The power of modern DRL techniques relies on their ability to make good decisions in scenarios where they have not been trained in advance. A good representation is crucial to simplify the learning process by reducing the level of abstraction required by the DL model. In other words, the more advanced the representation is, the easier and faster is the learning process.

In this chapter, we propose a novel representation for the network state (observations) that captures both the overall utilization of the network and the critical inter-dependencies among the paths resulting from the network topology in a way that is simple and that can be more easily exploited by the agent to learn better and faster. In addition, one of the main advantages of this representation is that it is flexible enough to be applied to a wide set of networking use cases.

We experimentally evaluate the proposed representation against state-of-the-art DRL-based solutions and traditional heuristics in two separate routing use cases to show its flexibility. First, we make an extensive evaluation of the performance of the DRL agent routing traffic demands in OTNs. This evaluation includes a profound hyperparameter evaluation and a reverse engineering analysis of the routing policy learned by the agent. And second, we test the agent on a QoS-aware routing scenario in IP networks including traffic with different levels of requirements (e.g., bandwidth, end-to-end delay).

2.2. DRL-based Routing Scenario

The objective of a DRL agent is to learn the policy that leads to a maximum *cumulative reward*. The learning is achieved by iteratively exploring the state and action spaces. In this chapter, we focus on optimizing the routing policy in a network. In particular, we define the routing problem as devising a certain strategy to route new source-destination traffic demands (flows) with the aim of making the best of the network resources in the long-term. Note that this is a challenging task for the DRL agent since it should learn during the training phase some singularities of network topologies such as potential bottlenecks as well as understand the underlying dependencies among end-to-end paths. Moreover, this learning process is also hampered by the uncertainty in the generation of future traffic demands, which is stochastic.

This problem can be modeled as a Markov Decision Process (MDP). A MDP is defined by the tuple $\{s, a, T, r, \gamma, s_0\}$. The *state* ($s$) must represent the environment unambiguously according to the action space defined. In our scenario, it must represent the current network state and some information about the traffic demand(s) to be routed. The *action* (a) stands for the set of actions the agent can apply (i.e., the changes to the routing configuration). The *transition distribution* $[T(s, a, s')]$ defines how the environment, the network in our case, evolves after applying an action. In our scenario, the transition function models the stochastic behavior of changes in the network state as well as the generation of new traffic demands to be routed. The *reward* (r) is the incentive that the agent obtains after making a decision. Its purpose is to steer the learning process towards the achievement of the optimization goal. Then, the objective is to find a policy $\pi(s)$ mapping input states to actions that maximizes the *discounted cumulative reward* (R):

$$R = \sum_{t=0}^{T} \gamma^t r(s_t, a_t) \tag{2.2.1}$$

The actions can be deterministic or stochastic (i.e., probability distribution over actions), and the *discount factor* [$\gamma \in [0, 1)$] defines the importance of the reward obtained in future decisions. In our scenario, γ must be considerably high, since our objective represents a long-term planning strategy. Finally, the initial state (s_0) represents an empty network with a first traffic demand to be routed.

Solving the MDP requires to evaluate all the possible combinations of state-action pairs, and this is computationally very expensive when the state has high dimensionality. In our scenario, the number of possible network states is proportional to the number of links, their capacity and the granularity considered for their utilization. For standard networks, this is typically an extremely high number of states. For instance, in some of the experiments we perform in Section 2.5 (Experimental Evaluation), the number of possible states is above 10^{100}.

An alternative to solve the MDP is using Reinforcement Learning (RL) together with sophisticated generalization techniques. This makes it possible to extract knowledge from visited states that can be used for unexplored states. In order to achieve this level of generalization, recent DRL algorithms propose the use of Deep Neural Networks (DNNs). Thus, with a proper training, such neural networks are able to model how to act successfully in regions not explored in advance.

Figure 2.1 represents the basic operation of the DRL agent in a network routing scenario. We assume that the DRL agent has access to telemetry information of the network, which is aligned with current architectural trends, such as Software-Defined Networking (SDN) [7] or Overlay Networking. Thus, when there is a new traffic demand to be routed, this is communicated to a network controller (step 1). Then, the controller generates a new state representation that will be the input of the DRL agent (step 2). This representation includes information about the network state and the new traffic request. With this input, the DRL agent selects an action that involves making a routing decision (step 3). Lastly, the controller translates the resulting action to a particular set of forwarding rules that are installed into some network devices (step 4). During the training phase, the agent explores different routing strategies and receives a reward after applying every action. This enables to learn a routing policy that leads to maximize the cumulative reward in the long-term.

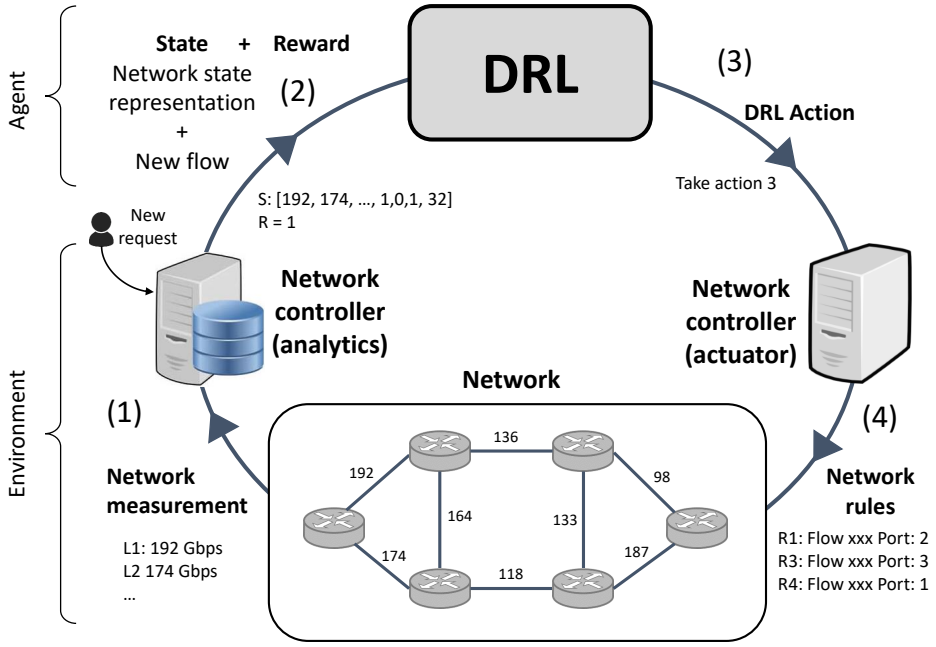


Figure 2.1: Schematic representation of the DRL-based routing scenario.

2.3. Proposed Representation

In this section, we propose a novel representation for DRL agents to perform online routing optimization. The design of this representation involves both how to define the network state of the agent (i.e., the *observation space*) and the set of actions it can apply (i.e., the *action space*).

2.3.1. State-of-the-Art Representations

Before describing our representation, we review the brief related work addressing network routing based on DL techniques and the representations they proposed. Some works, like [29, 52, 71], opt for using directly the traffic matrix (i.e., the traffic sent between each source-destination pair) as the representation of the network state. This information allows the agent to define a global routing policy considering the overall traffic demand in the network. Then, the action of the agent is to select the link weights of an external algorithm (e.g., softmin routing [71], OSPF-like [29]) that defines the final routing policy. Although these representations obtain reasonable performance in simple routing problems (e.g., link-weight selection), they exhibited poor results when applied to more complex problems, such as flow-based routing, even in some cases falling behind more classical routing algorithms.

Other approaches propose making routing decisions for every new traffic demand considering the current state of the network. Thus, in [75] the network state is represented

as a matrix that contains the traffic demand aggregated in every router for a number of time intervals. Alternatively, [61] represents the network state with the links' utilization. Particularly, they model links as binary arrays with a number of slots that can be either available (1) or occupied (0). Both proposals, [75] and [61], define a discrete action space for the agent where each option represents the selection of a path among a number of candidate paths. The main drawback of these representations is that the DRL agent must abstract knowledge from the link-level features represented in the observation space to the path-level options present in the action space.

We claim that, in networking scenarios, it is not feasible to achieve good performance by using only straightforward representations of the network state, such as the links' utilization or the traffic matrix. For example, these representations do not include information about the network topology and the interdependencies between the links that form an end-to-end path, which is critical to routing. Note that the alternative of including this information as an adjacency matrix would not solve the problem, as it would be very difficult for the DRL agent to learn these relationships from a raw matrix (e.g., the network paths and the links they share).

2.3.2. Description of the Representation

In contrast to state-of-the-art proposals, our approach is to do *feature engineering* to achieve a more elaborated state-action representation that facilitates the agent to learn how to efficiently route the traffic. In other words, our representation should help the DRL agent achieve generalization.

A key aspect to consider in the design of such a representation is that, in network scenarios, we can provide a simple estimate of how the network state will change after routing a new traffic demand. For instance, if we assume that we know the bandwidth request of an incoming demand (as in the routing scenario in Sec. 2.2), it is easy to estimate the resulting link utilization after allocating that demand to a specific end-to-end path. This means that we can leverage this information and provide this knowledge directly to the agent. This considerably simplifies the problem, because otherwise the agent would have to learn these relationships from exploration. However, there is still the challenge of how to choose the best routing policy considering the uncertainty of future traffic requests, which typically follow a stochastic generation process. In this context – after proper training – the agent should acquire some knowledge from the network and learn routing strategies that effectively deal with such uncertainty in the traffic. For instance, it may detect potentially critical links/paths and try to prevent bottlenecking them.

In the light of the above, we propose the following representation for the DRL agent. Instead of considering link-level statistics as in previous works (e.g., link utilization [61]), we propose the use of statistics at the level of *end-to-end paths*. This way, the agent does not need to infer knowledge from the link level to the path level. Regarding the action

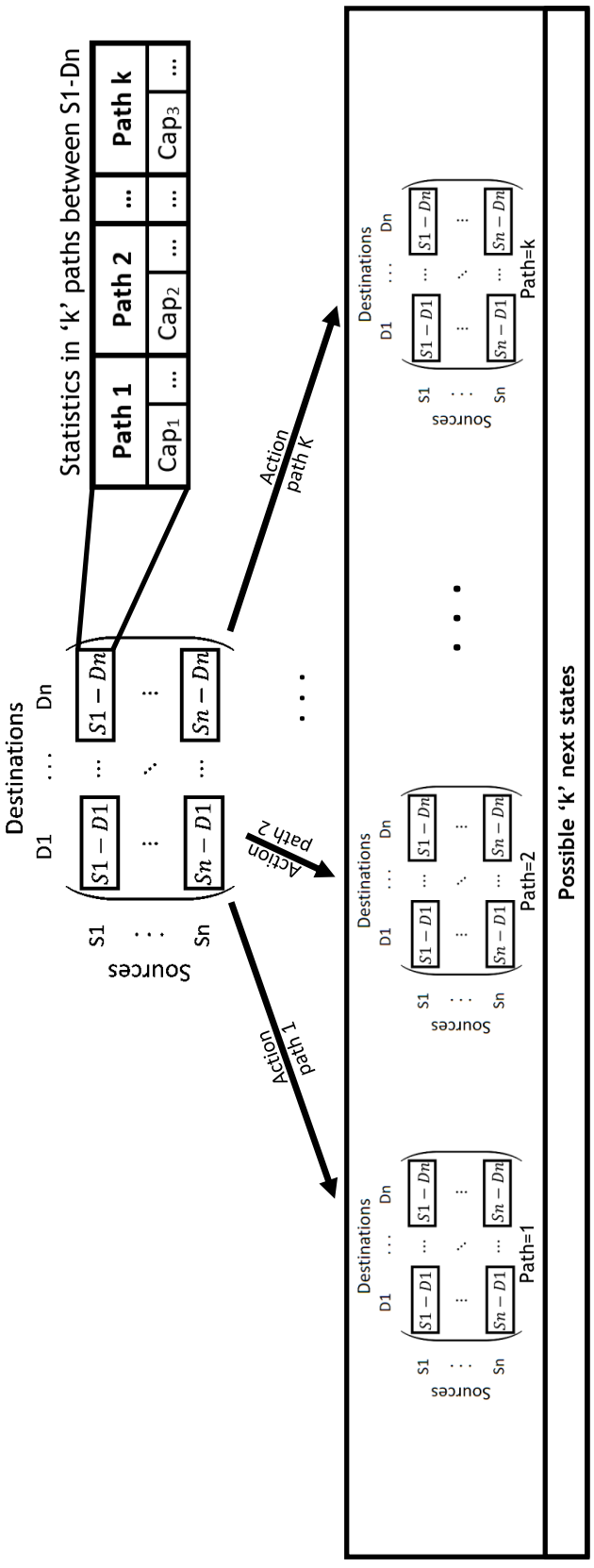


Figure 2.2: Scheme of the state representation proposed.

space, we propose a set of discrete actions where each action corresponds to the selection of a specific end-to-end path. Particularly, we consider that, for each new traffic demand $\{source,\ destination,\ bandwidth,\ \dots\}$, the agent can select one path among a list of "k" candidate paths (e.g., "k" shortest paths) that connect the source and the destination of such a demand. With respect to the state representation, the agent is provided with relevant statistics of the "k" candidate paths of all the source-destination pairs in the network. In Figure 2.2, the top matrix represents a scheme of the current network state. This matrix contains the current statistics of the "k" end-to-end paths for each source-destination pair. With this representation, it is possible to compute all the next states that can be reached after applying every possible action in the current action set and provide them to the DRL agent. That is, in each epoch the input of the DRL agent will be "k" matrices (as shown in the bottom of Fig. 2.2), where each matrix represents the estimated path statistics (e.g., available capacity) after allocating the current traffic demands to each of the "k" candidate paths. In other words, the proposed representation provides the agent with a set of matrices that describe the consequences of applying every possible action. Note that the cost to compute the input state is proportional to the number of possible actions considered. However, limiting the actions to "k" paths allows us to control the dimensionality and the cost to compute the state.

This representation is intended to be flexible and be used in a wide variety of routing-related problems. Note that, depending on the particular problem to address, the path-level statistics present in the state representation may vary (e.g., available capacity, end-to-end delay). Additionally, it may be required to include some information about the current traffic request (e.g., bandwidth, delay constraints).

2.4. Description of the DRL-based Solution

This section describes the complete DRL solution integrating the representation proposed in Section 2.3.2. The DRL agent receives as input the current network state including a new traffic demand, and the objective is to select a discrete action $a(t) \in [0..k]$ that represents a specific end-to-end path for the demand. Note that the number of candidate paths (k) and the criteria to select them (e.g., shortest paths) may have an impact on the final performance achieved by the agent. Formally, the DRL agent aims at finding a policy $\pi_\theta(s|a)$ modeled by a DNN with some weights and biases (θ) that are updated during training to maximize the discounted cumulative reward.

Besides the design of a good state-action representation for the agent, it is also important the selection of a proper DRL algorithm that well suits the nature of the problem. This involves for instance the exploration bias of the algorithm, which controls the tradeoff between the performance that can be potentially achieved and the training time to converge to the solution. The more biased is the exploration, the faster should the agent converge. However, having high exploration bias may result in policies whose performance is very

far from the optimal solution of the MDP problem. Note that the optimal MDP solution represents an upper-bound of the performance that DRL agents can achieve.

In order to select a proper DRL algorithm for our agent, we made some preliminary experiments with different agents implementing the following algorithms: Trust Region Policy Optimization (TRPO) [68], Proximal policy optimization (PPO) [76], Deep Deterministic Policy Gradients (DDPG) [77], Path Consistency Learning (PCL) [78], Asynchronous Advantage Actor Critic (A3C) [79] and Actor Critic with Experience Replay (ACER) [80]. To this end, we used the default implementations of these algorithms in ChainerRL (v0.3.0) [81]. Lastly, we found that the TRPO algorithm clearly outperformed the other algorithms in terms of performance and time to converge to the solution in the routing scenario presented in Section 2.2. Note that it does not necessarily mean that the other algorithms tested may not potentially achieve similar or even better performance than TRPO after a fine-tuning process. TRPO is a recent RL algorithm based on the classic Natural Policy Gradient algorithm [82]. Unlike primary policy gradients (e.g., REINFORCE [83]), TRPO introduces a number of sophisticated mechanisms that provide stability to the training and avoids the well-known vanishing or exploding gradient problems. This algorithm can be applied to both continuous and discrete action spaces. In our case, we consider that the agent aims to learn a stochastic policy $\pi_\theta(s|a) = P[a|s;\theta]$ based on the discrete action space proposed in Section 2.3.2 (i.e., k candidate paths). Additionally, the implementation used in this work includes two more mechanisms that further contribute to stabilize the training and, consequently, achieve better performance: (i) it implements an *actor-critic model* where the policy and the value estimates are modeled by separate Neural Networks (NNs), and (ii) it uses the Generalized Advantage Estimator (GAE) [84] as advantage function, which enables to reduce the variance of policy gradient estimates at the expense of some bias.

Algorithm 1 describes the DRL agent's training process. Firstly, a number of episodes is executed (line 2) following the current policy (π_θ) modeled by the actor NN. Thus, for each timestep in these episodes a tuple (s_t, a_t, r_t, s_{t+1}) is stored in a buffer ($\mathcal{D}_i$). Subsequently, advantage estimates are computed for each timestep sample in $\mathcal{D}_i$ (line 3) using the GAE $\left[\hat{A}_t^{GAE} = \sum_{l=0}^{\infty} (\lambda\gamma)^l \delta_t^V\right]$. Where $\lambda \in [0, 1]$ adjusts the bias-variance tradeoff of the estimator, $\gamma \in [0, 1)$ is the discount factor and δ_t^V is the temporal difference error that can be computed with the following expression: $\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$. At this point, the value estimates of the critic NN $V_\phi(s)$ are used to compute δ_t^V. Then, using these advantage estimates the actor NN (π_ϕ) is updated via the TRPO update algorithm [68] (lines 4-8). Finally, the critic is updated by minimizing the Mean Squared Error (MSE) between the current value predictions V_{ϕ_i} and the values updated with the new advantage estimates $V_{\phi_i} + \hat{A}_i^{GAE}$. To this end, it uses an Adam optimizer [85], which is an extension of the classic Stochastic Gradient Descent (SGD) method that incorporates adaptive learning rate.

Input: Initial parameters for policy π_ϕ and value V_ϕ
for $i = 1,\ 2,\ 3,\ \ldots$ *until convergence* **do**
  - Collect trajectories $\mathcal{D}_i$ on policy $\pi_i = \pi(\phi_i)$
  - Compute advantages $\hat{A}_i^{GAE}$ using the critic estimates V_{ϕ_i}
  - Compute the policy gradient $\hat{g}_i$ using $\hat{A}_i^{GAE}$
  - Compute the KL-divergence Hessian-vector product function:
$$f(V) = \hat{H}_i v$$
  - Apply conjugate gradient method to calculate:
$$x_i \approx \hat{H}_i^{-1} \hat{g}_i$$
  - Compute the proposed policy update step:
$$\Delta_i \approx \sqrt{\frac{2\delta}{x_i^T \hat{H}_i x_i}} x_i$$
  - Backtracking line search to obtain the final policy update (actor):
$$\theta_{i+1} = \theta_i + \alpha^j \Delta_i$$
  - Update the critic neural network (ϕ_{i+1}) using $\hat{A}_i^{GAE} \in \mathcal{D}_i$ with an Adam optimizer
end

Algorithm 1: TRPO Actor-critic training process.

2.5. Experimental Evaluation

In this section we evaluate the DRL-based solution described in Section 2.4 – which includes the state-action representation proposed in Section 2.3.2 – to route traffic in networks. To this end, we present a routing use case in Optical Transport Networks (OTNs). In this use case, we make an extensive evaluation of some relevant hyperparameters to optimize the performance achieved by the DRL agent, evaluate it against state-of-the-art DRL-based solutions and traditional heuristics, and finally make a reverse engineering analysis of the routing policy learned by the agent. Additionally, we test the performance of the agent in a simple QoS-aware routing in IP networks.

2.5.1. OTN Routing Use Case

In this use case, the objective of the DRL agent is to efficiently route traffic demands in OTNs. Figure 2.3 illustrates the DRL-based routing scenario in the context of OTN. In this scenario, we assume that the agent operates over a logical topology, where nodes are Reconfigurable Optical Add-Drop Multiplexer (ROADM) devices and edges are predefined lightpaths connecting the ROADM nodes. Then, the role of the DRL agent is to route new traffic demands – as they arrive – through particular sequences of lightpaths, which form end-to-end paths. Each traffic demand is defined by the tuple $\{source, destination, bandwidth\}$ and must be allocated in an end-to-end path connecting its source and destination endpoints. Since the agent works at the electrical domain, traffic demands are considered requests of Optical Data Unit (ODU) signals defined in the ITU-T Recommendation G.709 [86]. These ODUk signals – that may belong to different clients – are then multiplexed into Optical Transport Units (OTUs), which are data frames including

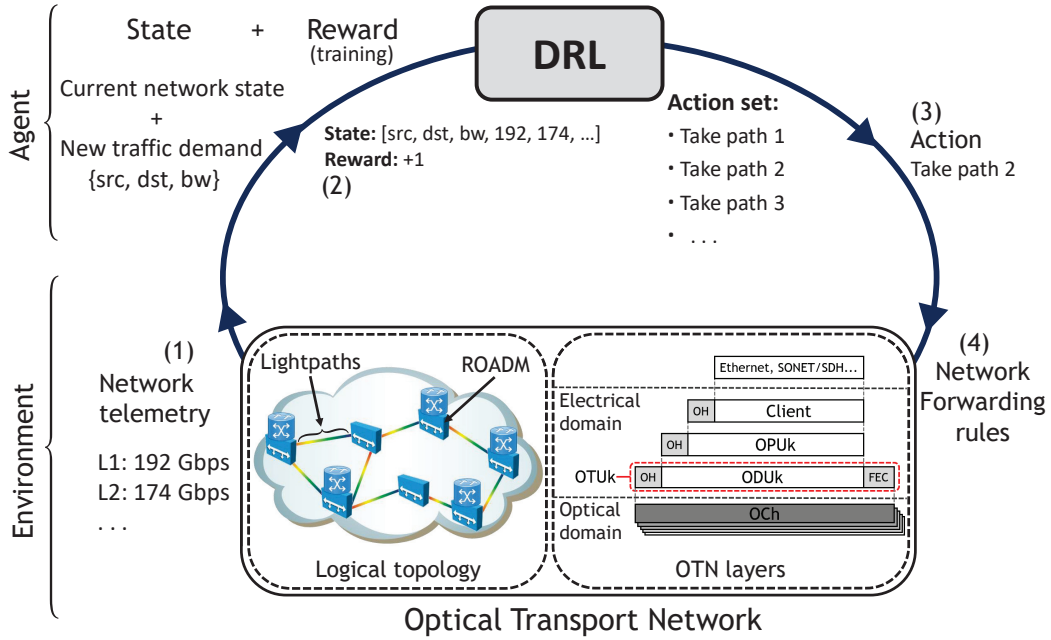


Figure 2.3: Schematic representation of the OTN routing scenario.

Forward Error Correction (FEC). The OTUk frames are finally transmitted over sequences of lightpaths (within optical channels) in the OTN. Note that in this scenario the DRL agent acts oblivious of the mechanisms related to the optical domain (e.g., physical impairments, modulation, wavelength assignment).

For the sake of simplicity, we consider 5 different types of traffic demands (ODU0 to ODU4) whose bandwidth requirements are expressed in terms of multiples of ODU0 signals. According to the ITU-T Recommendation G.709 [86], we define the bandwidth requirements as follows:

- ODU1 = 2 ODU0 Bandwidth Units (BUs)
- ODU2 = 8 ODU0 BUs
- ODU3 = 32 ODU0 BUs
- ODU4 = 64 ODU0 BUs

We consider that a demand is properly allocated if there was enough available capacity in all the lightpaths forming the end-to-end path selected. Traffic demands do not expire during an episode, hence episodes end when a demand do not fit into the path selected. Likewise, in order to maximize the total bandwidth allocated in the network, we define the immediate reward of the agent as the bandwidth (in ODU0 BUs) of the current traffic demand if it was properly allocated, otherwise the reward is 0.

In our experiments, we train the DRL agent described in Section 2.4. We used two independent fully-connected NNs respectively for the actor and critic models of the agent.

Each NN has two hidden layers, each one with 64 units. We made some experiments increasing the number of layers and units, but we did not see any relevant performance improvement. Note that the number of units in the input and output layers varies depending on the size of the observation and action spaces in the different experiments we perform. For the discount factor $\gamma \in [0, 1)$, we selected a value of γ=0.995. Note that γ represents the importance of the rewards obtained in future decisions (Sec. 2.2). Since the optimization objective in our scenario represents a long-term planning strategy, γ must be considerably high. For the DRL agent with our representation (Sec. 2.3.2), we pre-compute the k shortest paths (by number of hops) of all the source-destination pairs in the network. Thus, each step the action space contains the k shortest paths that connect the source and the destination of the new traffic demand to be routed. In all the cases, we use one-hot encoding vectors to represent the source, destination and ODUk type (i.e., bandwidth requirement) of traffic demands.

2.5.2. Hyperparameters Evaluation

We perform an evaluation of some relevant hyperparameters of the DRL implementation to optimize the performance of the agent specifically for our problem environment (Sec. 2.5.1). Particularly, we consider the following three hyperparameters:

(i) Number of paths considered in the action space
(ii) λ parameter used in the DRL algorithm
(iii) Update interval of the actor and critic networks

We evaluate – with a custom-built simulator – the DRL agent using our state-action representation in the 14-node National Science Foundation (NSF) network topology [87] (Fig. 2.4), where the topology edges represent lightpaths with a capacity of 200 ODU0 BUs in both directions. We generate new traffic demands with an uniform distribution for the source, destination and ODUk type.

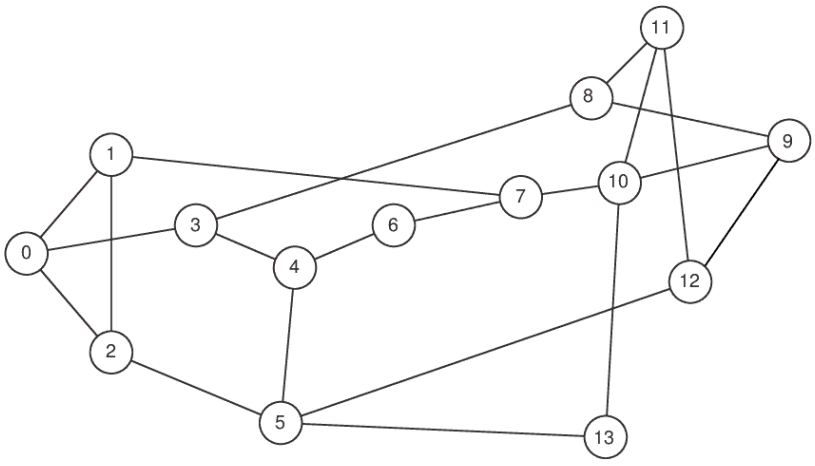


Figure 2.4: 14-node NSF network topology (image extracted from [1]).

In order to optimize each hyperparameter, we perform independent parametric evaluations using the following initial values:

- Number of candidate paths = 4
- $\lambda = 0.97$
- Update interval = 5,000 steps

The selection of these initial values is based on some preliminary experiments we made in a simpler scenario with a 6-node network topology. Note that despite each hyperparameter is optimized independently, there may be some co-dependencies among them. In this process, the selection of proper initial values is important to avoid large variations on some parameters during their independent fine-tuning process.

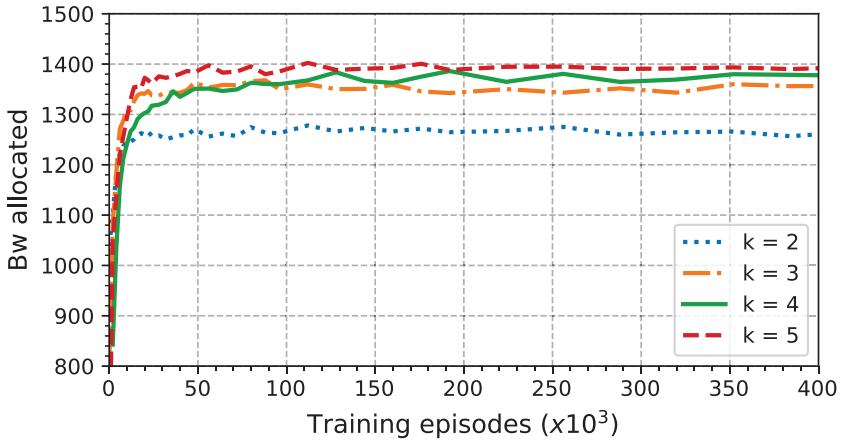
(*i*) Number of candidate paths

We vary the number of "k" shortest paths considered in the action space. Intuitively, the more paths available, the more flexibility has the DRL agent to allocate the traffic. However, it is important to maintain a reduced number of paths since more paths involves larger dimensionality in the state, and this implies more processing cost to update the input state and higher Random Access Memory (RAM) consumption. Also, it may imply a more complex learning process as there is a larger action space to explore.

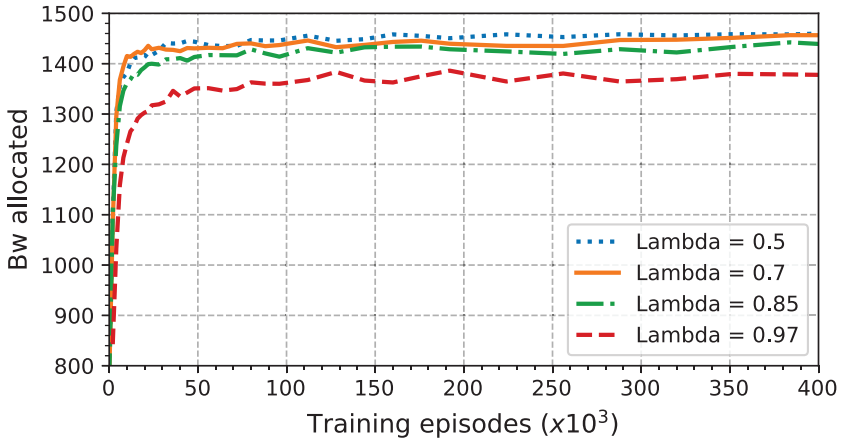
Figure 2.5a presents the results of this evaluation. The y-axis shows the average amount of bandwidth (in ODU0 BUs) allocated properly by the agent over 4,000 evaluation episodes with different seeds to generate the traffic. The x-axis indicates the number of training episodes of the DRL agent. Here, we can observe that the curves saturate after few training episodes, which means that the agent converges fast to its best strategy in all the cases. It is noteworthy that, in this particular case, the learning process does not necessarily slow down as the number of candidate paths grows. Based on these results, we consider that a value of k=4 paths is sufficient, since further increasing the number of paths implies to enlarge the state (more processing cost) and it does not improve significantly the performance. Note that the optimal value of "k" depends on some graph-level properties of the topology where the DRL agent operates (e.g., size, connectivity).

(*ii*) λ parameter of TRPO

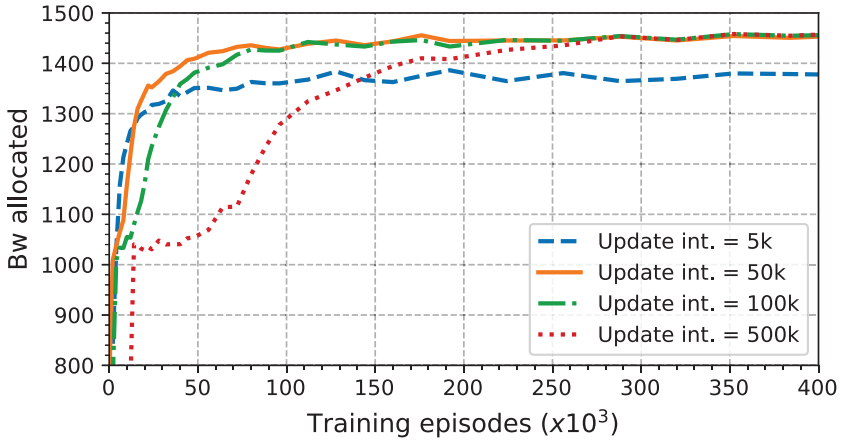
As mentioned earlier in Section 2.4, our DRL agent uses the GAE to compute the advantage estimates for the TRPO policy and value updates. It enables to reduce considerably the variance of the gradient estimates, but this comes at the expense of some bias. To control the tradeoff between bias and variance in the estimates there is a tunable parameter $\lambda \in [0, 1]$ in the GAE calculation $\left[\hat{A}_t^{GAE} = \sum_{l=0}^{\infty} (\lambda\gamma)^l \delta_t^V\right]$. Thus, λ=1 provides an unbiased estimate but introduces high variance. Conversely, λ=0 produces much lower variance, but may induce a lot of bias.



(a) Number of paths



(b) λ exploration parameter of TRPO



(c) Update interval

Figure 2.5: Hyperparameter evaluation in the NSF network. The y-axis represents the avg. bandwidth allocated over 4,000 evaluation episodes.

Figure 2.5b shows the results of our parametric evaluation of λ. Here, we observe that reducing the λ value from 0.97 to 0.7 both improves the performance and slightly accelerates the learning process. Likewise, decreasing the value beyond λ=0.7 does not further improve the performance.

(*iii*) **Update interval**

The update interval defines the number of training steps that are executed and maintained in memory before the DRL agent updates the policy and the value functions (i.e., the actor and critic NN models). The larger this interval is, the more data has to be stored until the NNs are updated. Consequently, this typically implies higher consume of RAM.

Figure 2.5c depicts the evaluation results varying the update interval. From these results, we can infer that the optimal value is an update interval of 50,000 steps in our case. With this value the agent achieves the same performance than using higher values (100,000 and 500,000). However, it learns faster and consumes less RAM.

2.5.3. Evaluation Against State-of-the-Art Representations

In this section, we compare the performance of our DRL agent with respect to other DRL agents using different state-action representations commonly present in the state-of-the-art [29, 61, 71] and a traditional Shortest Path routing policy.

We evaluate the agent in two real-world network topologies: the 14-node NSF network (Fig. 2.4) and the 17-node German Backbone Network (GBN) [88] (Fig. 2.6). As in the previous experiments, we consider that every edge in both topologies represents lightpaths with capacity for 200 ODU0 demands on both directions.

In our evaluation, we consider the following state-of-the-art representations:

(a) *Links and k-paths:* This is the simplest representation. It uses the available capacity of the lightpaths (edges in the logical topology) as the state representation and a discrete action space with k=4 candidate shortest paths (as in the DRL agent with our representation). This representation is similar to the one they use in Deep-RMSA [61], although they address the Routing, Modulation and Spectrum Assignment (RMSA) problem in Elastic Optical Networks (EONs).

(b) *Links and weights:* This representation uses also the available capacity of the lightpaths to represent the network state, but the actions consist of defining weights for the lightpaths; then, the path with lowest weight is selected (as in OSPF). This second representation uses the same action space as [29, 71].

We train the agent in two scenarios with different traffic profiles. In the first scenario, we used an uniform distribution for sources, destinations and ODUk types (as in Sec. 2.5.2).

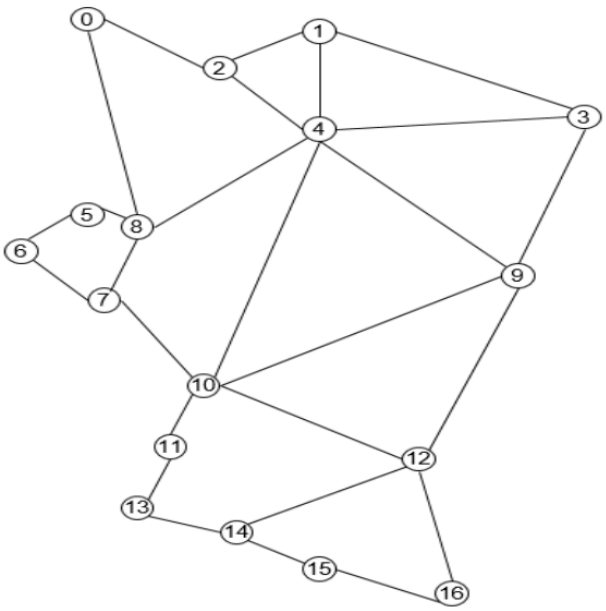
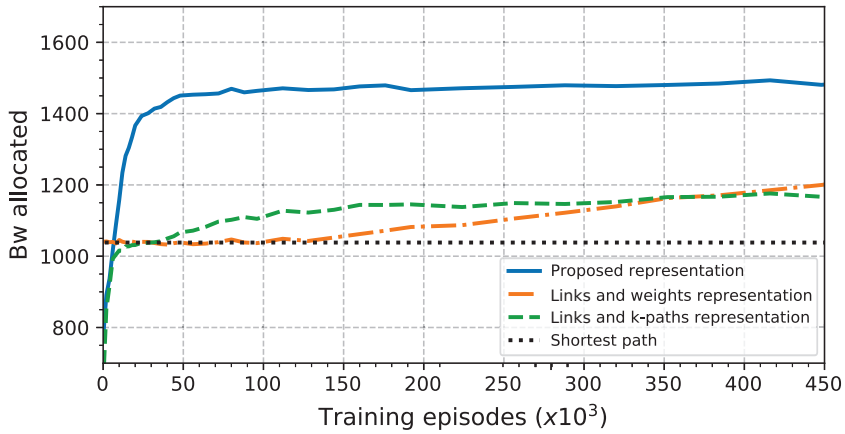


Figure 2.6: 17-node GBN topology (image extracted from [1]).

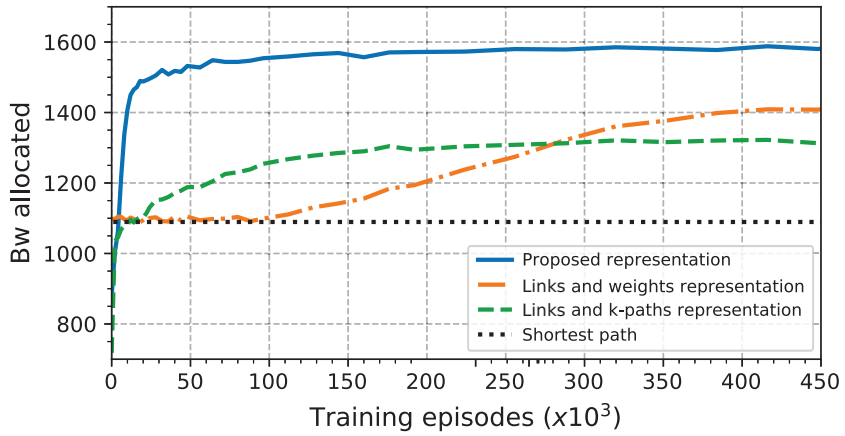
This represents the most challenging case for the DRL agent, given that it cannot exploit particular characteristics from the traffic to direct the exploration during training. The second scenario represents a more realistic traffic distribution similar to the one we can find in real-world networks. We generate traffic with a bimodal distribution [89], in which 20% of nodes generate 80% of the traffic. Also, the distribution of the ODUk requests follows an elephant-mice distribution [90], where there is a high number of low bandwidth requests and the bulk of the traffic is generated by a reduced number of big traffic demands.

Based on the hyperparameters evaluation (Sec. 2.5.2), we select a value of λ=0.7 and an update interval of 50,000. For those representations with discrete path-based actions, we use k=4.

Figure 2.7 shows the average bandwidth properly allocated w.r.t. the number of training episodes for the two traffic scenarios in the NSF network topology. Each figure depicts the performance achieved by our representation, the two representations based on the state-of-the-art and the application of a traditional Shortest Path routing policy. In the scenario with realistic traffic (Fig. 2.7b), the agent achieves slightly higher performance compared to the scenario with uniform traffic because there are more low bandwidth requests, which are easier to be allocated properly. Likewise, for both traffic distributions, we observe a similar behavior: (*i*) the simplest representation (*Links and k-paths*) outperforms the shortest path policy but its performance is quite poor compared to our representation. The proposed representation is able to allocate approximately 26% more bandwidth with an uniform traffic distribution and 22% with realistic traffic. (*ii*) the *Links and weights* representation is able to surpass the shortest path policy and *Links and k-paths*, but it learns much slower than



(a) Uniform traffic distribution



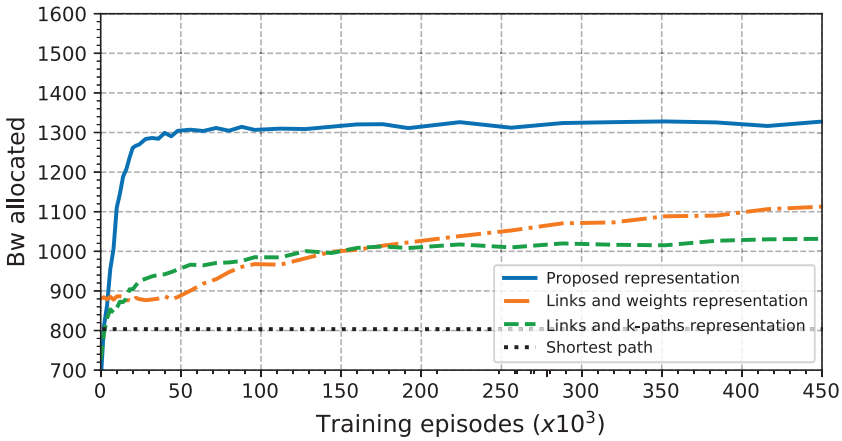
(b) Realistic traffic distribution

Figure 2.7: Evaluation against state-of-the-art representations in the NSF network. The y-axis represents the avg. bandwidth allocated over 4,000 evaluation episodes.

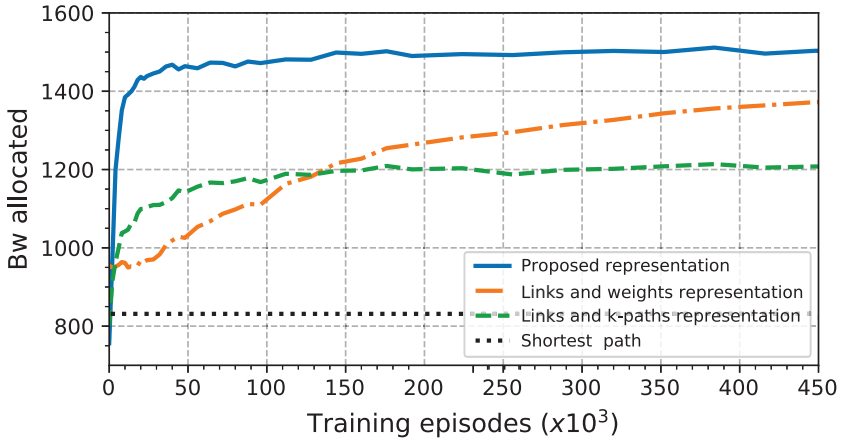
using the other representations. For instance, in the case with realistic traffic, the proposed representation needs only 10,000 episodes to reach the same performance achieved by the *Links and weights* representation after 450,000 episodes.

Figure 2.8 shows the same experiments for the GBN topology. In this scenario, we can observe a similar behavior for the three representations, but also an increase in performance compared to the results obtained by the shortest path policy. This can be explained by the different distributions of the betweenness centrality of edges (i.e., lightpaths) in both topologies. For example, in the GBN topology, we observe that some edges are included in a high number of shortest paths connecting different source-destination pairs, and this makes these edges more prone to become congested. We further discuss this issue in Section 2.5.4.

In all the evaluations we performed with link-based state representations (i.e., "Links and k-paths" and "Links and weights"), the agent achieves better performance when it



(a) Uniform traffic distribution



(b) Realistic traffic distribution

Figure 2.8: Evaluation against state-of-the-art representations in GBN. The y-axis represents the avg. bandwidth allocated over 4,000 evaluation episodes.

applies actions to select the weights on the lightpaths ("Links and weights") than when it selects directly end-to-end paths ("Links and k-paths"). This suggests that, when representing the state with link-level features, it may be more beneficial la using also link-level actions (e.g., links weights) than applying path-level actions. Additionally, we observe that the learning process of the agent is much faster when it uses discrete path-level actions than when it defines weights at the link-level. In this context, our representation uses path-level features for both the state and action spaces, which avoids the agent to abstract knowledge from the link to the path-level. What is more, the state includes explicit information about the resulting states after applying all the possible actions, which makes the problem less complex for the agent.

2.5.4. Evaluation in an Adverse Scenario for Shortest Path Routing

In Section 2.5.3, we discussed that in the GBN scenario the DRL agent achieves better performance with respect to the shortest path policy than in the NSF network scenario. In order to further investigate this issue, in this section we analyze these topologies from a graph theory perspective and make an evaluation in a scenario that is specifically adverse to the application of the shortest path policy.

To this end, we first introduce the concept of *betweenness centrality*. This is a metric of centrality used in graph theory based on the configuration of shortest paths. Particularly, the betweenness centrality of an edge is the ratio of shortest paths that pass through that edge with respect to the total number of node pairs in the graph. Thus, in network topologies using shortest path routing, edges with high betweenness centrality are more likely to become saturated if their capacity is not scaled to that factor. That is the case of the GBN topology in Section 2.5.3, where all the lightpaths have the same capacity and some of them (the top 5) are included in [8.8-14.3%] of all the shortest paths connecting all the node pairs in the topology. In contrast, in the NSF network topology, the top 5 lightpaths are within the range [6.5-8.2%] of betweenness centrality.

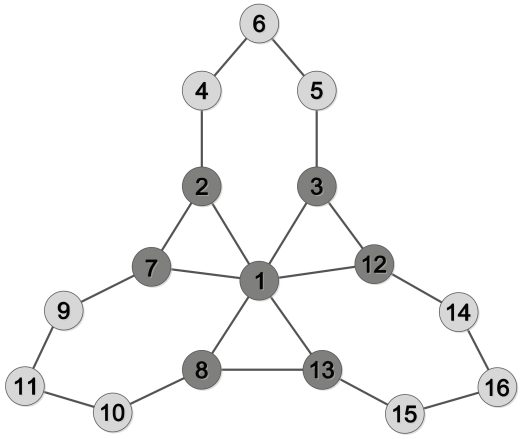


Figure 2.9: Adverse network topology for shortest path routing.

To show the ability of our DRL-based solution to adapt to these adverse scenarios to the shortest path policy, we make an evaluation in the topology depicted in Figure 2.9. This topology is inspired by widely deployed real-world networks that form rings to connect some regions (e.g. country-level networks) and are inter-connected by a central core. Particularly, this topology contains three identical rings that are connected via a central node and has some alternative links that permit to offload traffic through the closest nodes of the other rings (e.g., from node 2 to 7). Note that all the lightpaths have a capacity of 200 ODU0 demands. In this topology, the top 5 lightpaths have a betweenness centrality in the range [9.1-10%].

For the evaluation, we maintain the same hyperparameter values used in Section 2.5.3 and train the DRL agent in two scenarios with different traffic profiles. In the first scenario, we generate traffic following an uniform distribution for sources, destinations and ODUk types. In the second scenario, we further exacerbate the effect of lightpaths with high betweenness centrality by using a realistic bimodal traffic distribution in which 30% of node pairs generate 80% of the total traffic volume. In this case, the set of node pairs generating more traffic contains nodes directly connected to the top 5 lightpaths with higher betweenness centrality. This further hampers the possibility of the shortest path policy to succeed. Particularly, these pairs are (in both directions):

$$\{(1\text{-}2),\ (1\text{-}3),\ (1\text{-}7),\ (1\text{-}8),\ (1\text{-}12), (1\text{-}13), (2\text{-}3), (7\text{-}8),\ (12\text{-}13)\}$$

In this latter scenario, ODUk demands follow an elephant-mice distribution [90] as in Section 2.5.3.

Figure 2.10 shows the evaluation results. We can observe that, with uniform traffic the DRL agent achieves ≈40% more bandwidth allocated than the shortest path policy. Likewise, as we expected the DRL agent further outperforms the shortest path policy in the scenario with realistic traffic by allocating ≈52% more bandwidth. In this latter scenario, we analyzed the behavior of the DRL agent to get some hints of the strategy learned. Thus, we could observe that in some cases the agent decides to take longer paths to avoid using those lightpaths with high betweenness centrality, given that they are more prone to be congested. For instance, when there is a new traffic demand between nodes 2 and 3, the agent always selects the path [2-4-6-5-3] instead of following the shortest path [2-1-3], which traverses two lightpaths with higher betweenness centrality. This is a smart strategy given that the amount of traffic generated among node pairs in the set {2,4,6,5,3} is considerably lower in this scenario.

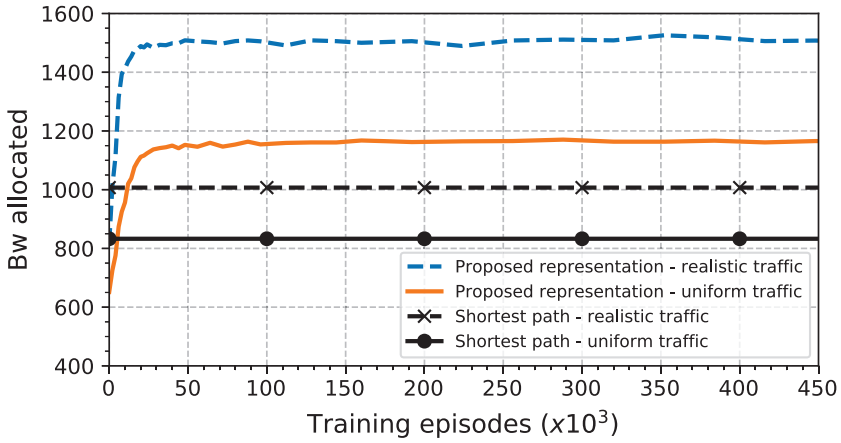


Figure 2.10: Evaluation in an adverse scenario for shortest path routing. The y-axis represents the avg. bandwidth allocated over 4,000 evaluation episodes.

2.5.5. Evaluation Against Current Shortest Available Path

In this section, we evaluate the DRL agent using our representation against the application of a more sophisticated heuristic we call "current Shortest Available Path" (hereafter SAP). In particular, it consists of dynamically filtering from the set of "k" candidate shortest paths those with enough capacity to support the new traffic demand and select from this subset the path with lower number of hops. This policy typically represents a performance very close to the optimal MDP solution in our OTN routing scenario. Note that it does not necessarily represents an efficient strategy in other related problems considering optical-level constraints (e.g., wavelength continuity).

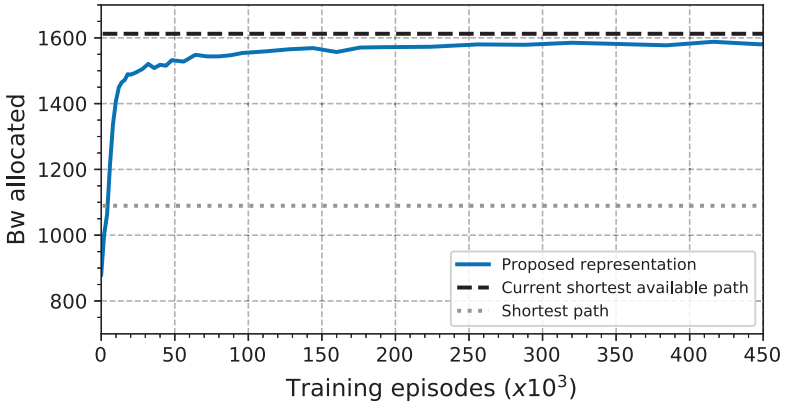
We make the evaluation in the setup described in Section 2.5.1 using the same hyper-parameter configuration as in Section 2.5.3 for our DRL-based solution. For a fair comparison, we use the same action set with k=4 candidate shortest paths for the DRL agent and the SAP policy. The evaluation is made in the three network topologies used previously: (*i*) NSF network topology (Fig. 2.4), (*ii*) GBN topology (Fig. 2.6), and (*iii*) adverse topology for the shortest path policy (Fig. 2.9).

In our experiments, we use realistic traffic distributions. For the NSF network and GBN topologies we generate traffic with the bimodal distribution described in Section 2.5.3, whereas in the adverse topology for shortest path routing we use the bimodal distribution described in Section 2.5.4. Figs. 2.11a, 2.11b and 2.11c show the evaluation results respectively in the three different topologies. As we can observe, the DRL agent achieves similar performance to SAP in the NSF network topology. However, it clearly allocates more bandwidth than SAP in the GBN topology (≈10%) and the adverse topology for shortest path (≈6%). This evidences that the DRL agent devised in these cases a smarter strategy than SAP by exploiting some knowledge acquired from topology singularities and traffic distributions.

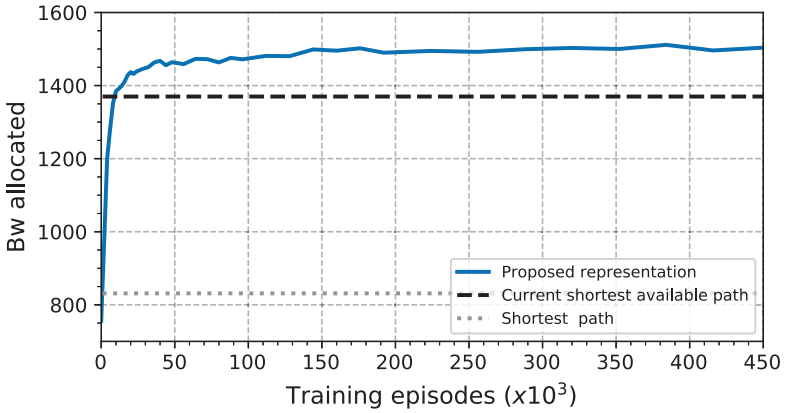
In order to better understand the performance achieved by the SAP policy, we calculate the optimal solution in a simple scenario. Particularly, we solve the MDP problem by exploring all the possible states and compare the performance achieved with respect to SAP and the DRL agent using our representation.

Since the calculation of the MDP solution is computationally very expensive, we made the evaluation in a simple scenario with the 6-node topology used in Deep-RMSA [61]. Here, the edges correspond to lightpaths with capacity for 3 ODU0 demands in both directions. For the traffic generation, we consider that all the traffic demands are ODU0 and the sources and destinations follow an uniform distribution.

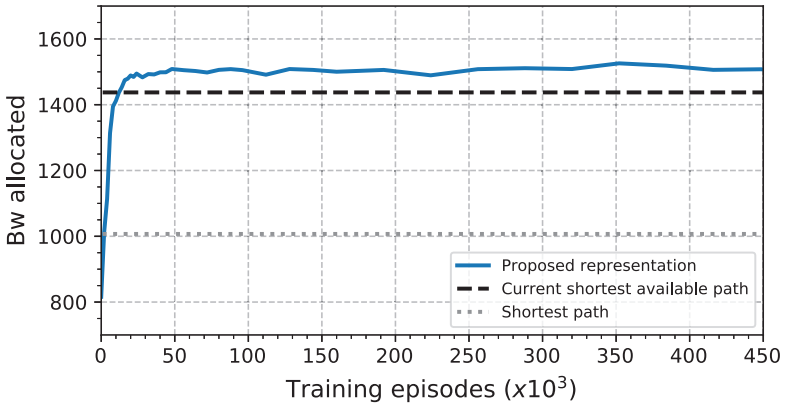
Figure 2.12 depicts the evaluation results. As we expected, the SAP policy obtains practically the same performance as the optimal MDP solution. This figure also evidences that, in this scenario the DRL agent with our representation achieves a performance very



(a) NSF network topology



(b) GBN topology



(c) Adverse topology for shortest path

Figure 2.11: Evaluation of the DRL-based solution proposed against the SAP policy with realistic traffic distributions. The y-axis represents the avg. bandwidth allocated over 4,000 evaluation episodes.

close to the optimal policy. Likewise, we observe that the shortest path policy is quite far from the other strategies. Particularly, it allocates ≈50% less bandwidth on average.

Note that, in these latter experiments traffic demands follow an uniform distribution, hence it is not possible to exploit any meaningful information from the traffic distribution. However, in the presence of non-uniform distributions (e.g., bimodal) the optimal MDP solution may achieve better performance with respect to SAP. We do not provide results for this latter case given that the computation of the MDP solution turns to be considerably more costly.

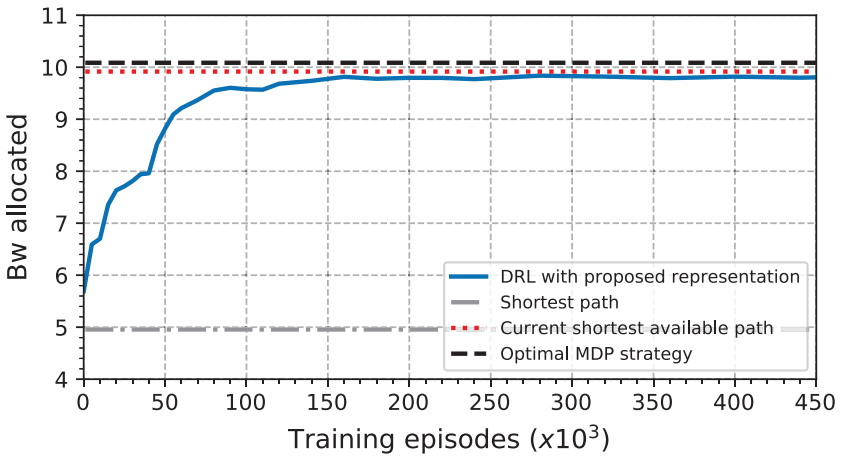


Figure 2.12: Evaluation of the current SAP policy w.r.t. the optimal MDP solution in a simple scenario.

2.5.6. Reverse Engineering of the Routing Policy Learned by the DRL Agent

Reverse engineering of DRL-based solutions may have two main advantages: (*i*) it enables to make more efficient implementations of the policy learned by DRL agents (e.g., new heuristics), and (*ii*) having an implementation with deterministic behavior avoids facing the uncertainty that characterizes ML-based solutions when acting over input states not considered during training and evaluation.

In this section we aim to gain insight into the policy learned by our DRL agent. As we could observe in Section 2.5.5, there are some cases where the DRL agent clearly outperformed the SAP policy, particularly in the GBN scenario and the adverse topology for shortest path. In this context, our approach is to focus on these scenarios and analyze the cases where the actions of our DRL agent differ from SAP. To this end, we make experiments in the adverse topology for shortest path and use the DRL agent already trained for 450,000 episodes in the scenario with realistic traffic (Fig. 2.11c).

For the evaluation, we run 10,000 episodes and collect some relevant statistics considering the cases where the DRL agent differs from SAP. We mainly focus on analyzing the long-term strategy of the agent. Accordingly, we only analyze routing scenarios where the network is not excessively congested so that the agent has enough flexibility to allocate the new traffic demand in multiple paths. Otherwise, when the network becomes considerably congested, decisions are not meaningful to infer the long-term strategy as there are often very few paths with enough available capacity. Thus, in our experiments we consider that episodes begin with an empty network and end when there is at least one lightpath that cannot support an ODU4 demand (i.e., minimum available capacity of 64 ODU0 BUs).

Table 2.1 summarizes some statistics we extracted from the experiments. The first remarkable point is that the DRL agent took on average 59.94% of actions different from SAP. This suggests that the strategy followed by the agent may not be similar to the SAP policy. Additionally, we can observe that the DRL agent selected a longer path (in number of hops) than SAP in 56.97% of the cases. In other words, 95.04% of the times that the DRL agent differs from SAP, it selects a longer path (see Table 2.1, #2). Considering the significance of this statistic, we further analyze these cases. We then compute the following combinations of occurrences: (*i*) longer path with higher available capacity and (*ii*) longer path with lower weighted betweenness (see Table 2.1 #5 and #6). Note that we define the weighted betweenness of a link as the potential number of end-to-end paths that traverse such link weighted by the amount of traffic that may carry all these paths. Here, we can observe that in ≈40% of the cases the agent selects a larger path that also has higher available capacity or lower weighted betweenness.

Based on these results, we compute a more elaborated statistic that considers the ratio between the available capacity and the weighted betweenness (see Table 2.1, #7). Then, we can see that in 53.96% of the cases where the DRL agent differ from SAP, it selects a path with higher minimum (*av. cap/wgt. betweenness*) ratio over all the links of the path. This suggests that the behavior of the DRL agent may be partially explained by a policy that selects the path that maximizes the minimum (*av. cap/wgt. betweenness*) ratio over the k candidate paths. We show below an analytic expression describing such policy:

$$Path = \max_{P_k \in \mathcal{P}} \left(\min_{l \in P_k} \left(\frac{Av.\ cap\ (P_k(l))}{wgt.\ btw\ (P_k(l))} \right) \right) \tag{2.5.1}$$

Where $\mathcal{P}$ is the the set with k candidate paths and l indexes all the links on the k-th path (P_k).

Note that this policy implicitly includes a prediction of the potential traffic that may carry each link to compute the weighted betweenness. An alternative method that does not involve traffic prediction mechanisms would be to consider instead the classic (non-weighted) betweenness metric (i.e., considering only the number of paths that may traverse a link). This would be equivalent to apply the policy in Equation 2.5.1 and assume that the

Table 2.1: Statistics summary of the reverse engineering analysis.

#	Statistic	$\frac{\text{\% of actions}}{\text{Total actions}}$	$\frac{\text{\% of actions}}{\text{Different actions}}$	Description
1	Different actions	59.94%	-	% of the DRL agent's actions that differ from the SAP policy (w.r.t. total number of actions over 10,000 episodes)
2	Longer path	56.97%	95.04%	% of cases where the DRL agent selects a longer path (in number of hops) than the selection of SAP
3	More available capacity	27.95%	46.62%	% of cases where the DRL agent selects a path with more available capacity than in the selection of SAP
4	Lower weighted betweenness	24.30%	40.54%	% of cases where the DRL agent selects a path with lower maximum weighted betweenness than in the selection of SAP
5	Longer path and more av. cap.	26.06%	43.47%	% of cases where the DRL agent selects a path that is longer and has more available capacity than in the selection of SAP
6	Longer path and lower wgt. btw.	22.62%	37.74%	% of cases where the DRL agent selects a path that is longer and has lower weighted betweenness than in the selection of SAP
7	Higher $min\left(\frac{av.cap}{wgt\_betweenness}\right)$	32.34%	53.96%	% of cases where the DRL agent selects a path whose minimum $av.\ cap/wgt.\ betweeness$ (over all its links) is higher than in SAP

traffic is uniformly distributed over all the source-destination pairs in the network. In order to evaluate the performance of this latter heuristic, we run 5,000 evaluation episodes in the three topologies used in previous sections with the realistic traffic distributions. Table 2.2 shows the average bandwidth allocated (in ODU0 BUs) by the heuristic and compares it with the performance achieved by the SAP policy over the same experiments. As we can observe, our heuristic inspired by the actions of the DRL agent outperforms in all the cases the SAP policy. The best case is in the GBN topology, where it allocated 15.07% more bandwidth than SAP.

Table 2.2: Evaluation of the routing heuristic based on our DRL agent.

Average bandwidth allocated (5,000 episodes)	NSF network	GBN	Adverse topology to shortest path
Heuristic	1642.28	1574.52	1536.81
SAP	1611.71	1368.36	1470.15
Improvement	1.90%	15.07%	4.53%

2.5.7. Evaluation in a QoS-aware Routing Use Case

In this section, we test our DRL agent (Sec. 2.4) with the state-action representation proposed in Section 2.3.2 to perform routing in IP networks. Particularly, this use case is focused on Quality of Service (QoS) provisioning. The objective of the DRL agent is to efficiently route traffic flows so that their QoS requirements are met. In this context, we consider a scenario with 4 different types of applications that generate flows with various bandwidth and end-to-end delay requirements (Table 2.3).

Table 2.3: Types of applications in the QoS-aware routing use case.

Application	**Bandwidth**	**Maximum tolerable delay**
App 1	300 kbps	10 ms
App 2	500 kbps	15 ms
App 3	1.5 Mbps	20 ms
App 4	3 Mbps	30 ms

Note that this scenario is aware of end-to-end delays in paths. For this purpose, we extended the custom-built simulator used in previous evaluation sections to model also the delay. Particularly, we compute the delay on links with a non-linear function based on the link utilization (Fig. 2.13).

We trained the agent in the 6-node topology depicted in Figure 2.14, where all the links have a capacity of 50 Mbps. An episode starts with an empty network. Then, the simulator generates new flows with random source, destination and application type that must be

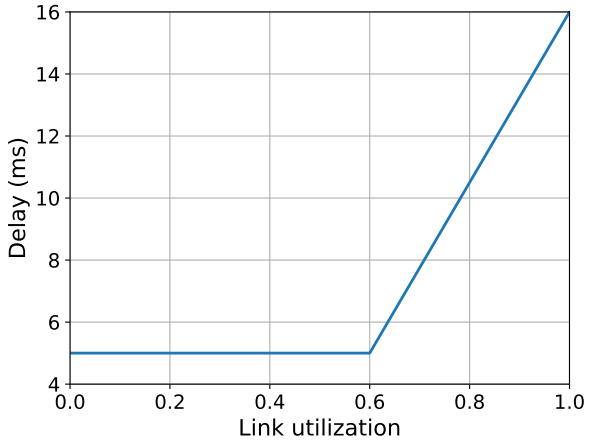


Figure 2.13: Delay function in the QoS-aware routing use case.

routed by the DRL agent. Finally, an episode ends when the average link utilization of the network is above 85%.

For this use case, the proposed state representation (Sec. 2.3.2) includes as path-level statistics the available capacity and the end-to-end delay measured on paths. Similarly to the previous use case, we represent new flows using one-hot encoding vectors to describe the source, destination and application type (1-4). Moreover, the action set consists of selecting one among the k=4 shortest paths for the specific source-destination pair of the new flow. We calculate the immediate reward (r) of the agent as the ratio of flows whose delay requirements are satisfied with respect to the number of flows routed at the current time (eq. 2.5.2).

$$r = \frac{flows\ satisfied}{flows\ routed} \tag{2.5.2}$$

Figure 2.15 shows the evaluation results of the agent as a function of the training episodes (x-axis). The score represents the maximum number of flows satisfied that the agent was able to route during an evaluation episode. Particularly, each evaluation point (y-axis) shows the average score achieved by the agent over 120 episodes with different flow sets randomly selected. Also, we compare these results with the score achieved by a QoS-

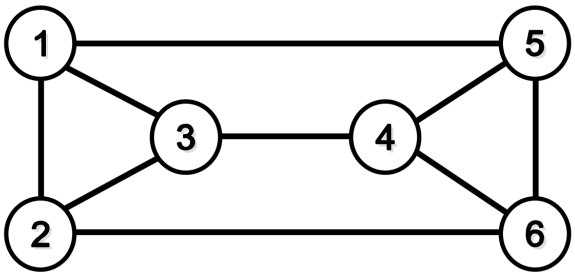


Figure 2.14: 6-node topology used in the QoS-aware routing use case.

aware routing heuristic we call Lowest Delay Path (LDP). We define the LDP policy as selecting greedily the current path with lowest delay among the k=4 candidate paths. As we can observe, after 2,000 training episodes the DRL agent beats the LDP policy and, by episode 25,000, it achieves around 14% better performance.

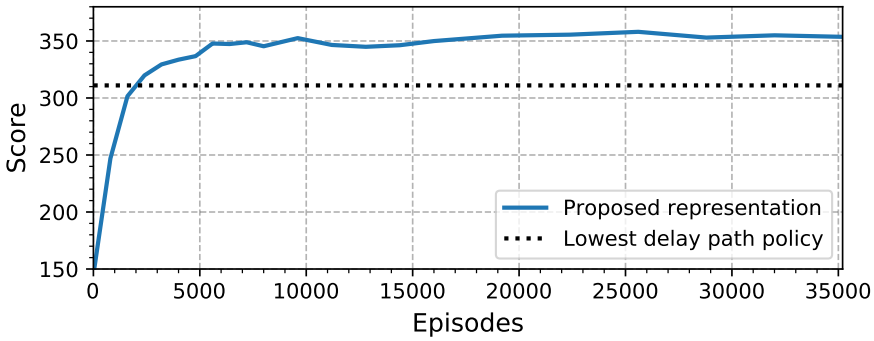


Figure 2.15: Training of the QoS-aware routing use case.

In order to further analyze the behavior of the DRL agent with our representation, we show in Figure 2.16 a step-by-step evaluation during an example episode. This figure represents the number of flows whose delay requirements were satisfied (y-axis) with respect to the total number of flows already routed by the agent (x-axis). It is noteworthy that, at the beginning of the episode – when the network is empty – all the flows can be satisfied regardless of the path selected by the agent. However, there is a point around flow #170 where the agent begins to outperform the LDP baseline. This suggests that the DRL agent planned from the beginning a better routing strategy in the long-term. Likewise, we observe that when the network suffers from severe congestion (around flow #355), the DRL agent is still able to minimize the impact of the congestion and maintain a higher rate of flows satisfied than LDP.

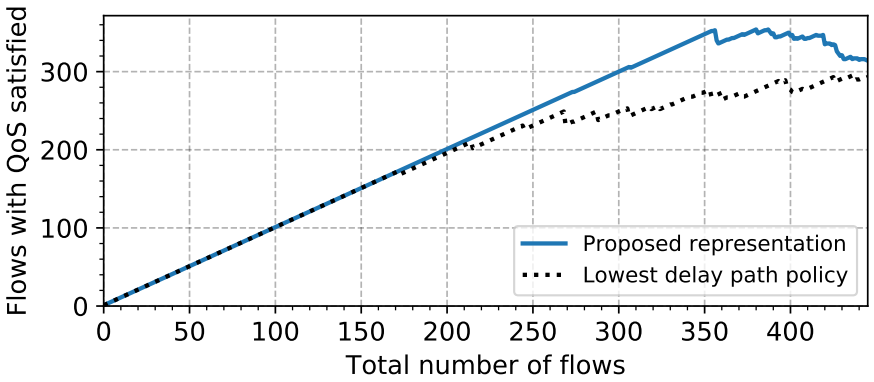


Figure 2.16: Step-by-step evaluation during an episode.

2.6. Chapter Summary

In this chapter, we discussed the challenges involved in the application of DRL to networking. Recent efforts in the computer networks field went in the direction of applying DRL as a black-box using simple representations of the observation/action space. We argue that the key to the success of DRL in networking is a more careful representation of the network state (i.e., *feature engineering*). This is explained by the complexity of describing link interdependencies and the stochastic nature of the network traffic.

As a result, we proposed an alternative representation of the network state that still has reasonable dimensionality, but can better capture the crucial relationships among the links that form the network topology. This more "engineered" representation allows the agent to learn more easily and faster. Our evaluation results, including different network topologies and traffic profiles, show that our representation significantly outperforms previous DRL-based proposals in two paradigmatic use cases: routing in Optical Transport Networks, and a simple QoS-aware routing scenario in IP networks.

Chapter 3

Applying Graph Neural Networks to Network Optimization

Network modeling is a critical component for building self-operating networks, particularly to find optimal network configurations that meet the goals set by administrators. However, existing modeling techniques do not meet the requirements to provide accurate estimations of relevant performance metrics such as delay and jitter. In this chapter, we present a novel Graph Neural Network (GNN) model able to understand the complex relationships between topology, routing, and input traffic to produce accurate estimates of the per-source-destination mean delay and jitter. GNNs are tailored to learn and model information structured as graphs. As a result, the GNN model presented in this chapter is able to generalize over arbitrary topologies, routing schemes, and variable traffic intensity. In the evaluation, we show that this model provides accurate estimates of delay and jitter (worst case $R^2 = 0.86$) in scenarios with topologies (up to 50 nodes), routing schemes, and traffic not seen during training. In addition, we present the potential of the model for network operation by presenting several use cases that show its effective use in different delay and jitter-aware network optimization scenarios.

3.1. Introduction

Network optimization is a well-known and established topic with the fundamental goal of operating networks efficiently. In the context of the Knowledge-Defined Networking (KDN) paradigm [52], network optimization is achieved by incorporating two components to SDN controllers: (i) a network model and (ii) an optimization algorithm (e.g, [91]). Typically, the network administrator configures the network policy (goals) in the optimization algorithm that uses the network model to obtain the configuration that meets the goals.

In this traditional and well-known architecture the model is responsible for predicting the performance (e.g, per-link utilization) of the network (e.,g topology) for a particular configuration (e.g, routing). Then the optimization algorithm is tasked to explore the configurations to find one that meets the goals of the network administrator. An example of this is Traffic Engineering, where the goal is finding a routing configuration that keeps the per-link utilization below the per-link capacity. Since the dimensionality of the configuration is typically very large, efficient optimization strategies reduce them by using expert knowledge. The networking community has developed over decades a large set of network models and optimization strategies [74].

One of the fundamental characteristics of network optimization is that *we can only optimize what we can model*. For example, in order to optimize the jitter of the packets traversing the network we need a model able to understand how jitter relates to other network characteristics. In the field of fixed networks many accurate network models have been developed in the past, particularly using Queuing Theory [92]. However, such models make some simplifications like assuming some non-realistic properties of real-world networks (e.g., generation of traffic with Poisson distribution, probabilistic routing). Moreover, they do not work well for networking problems involving multi-hop routing (i.e., multi-point to multi-point queueing) and estimation of end-to-end performance metrics [62]. As a result, they are not accurate for large networks with realistic routing configurations and as such, delay, jitter and losses remain as critical performance metrics for which no practical model exists.

Recent advances in Artificial Intelligence (AI) [21,22] have led to a new era of Machine Learning (ML) techniques such as Deep Learning (DL) [10]. This has attracted the interest of the networking community to try to take advantage of these novel techniques to develop a new breed of models, particularly focused on complex network behavior and/or metrics.

In this context, relevant research efforts are being devoted into this new field. Researchers are using neural networks to model computer networks [23, 24] and using such models for network optimization [75], in some cases in combination with advanced strategies based on Deep Reinforcement Learning (DRL) [61, 62, 71].

Such proposals [93, 94] typically use well-known Neural Network (NN) architectures like fully-connected NN, Convolutional NN (extensively used for image processing), Recurrent NN (used for text processing) or Variational Auto-Encoders. However, computer networks are fundamentally represented as graphs, and such types of NN are *not designed to learn information structured as graphs*. As a result, the models trained are strongly limited: they provide limited accuracy and are unable to generalize in terms of topologies or routing configurations. This is one of the main reasons why ML-based network optimization techniques have – at the time of this writing – failed to meet its expectations and clearly outperform traditional techniques.

To address these issues we present RouteNet, a pioneering network model based on Graph Neural Network (GNN) [69]. Our model is able to understand the complex relationship between topology, routing and input traffic to produce accurate estimates of the per-source-destination pair mean delay and jitter. GNN are tailored to learn and model information structured as graphs and as a result our model is able to generalize over arbitrary topologies, routing schemes and variable traffic intensity. To the best of our knowledge, this is the first work to address such fundamental networking problem using ML-based techniques able to learn and *generalize*.

GNN models have grown in popularity in recent years and are particularly designed to operate on graphs with the aim of achieving relational reasoning and combinatorial generalization. In other words, GNNs facilitate the learning of relations between entities in a graph and the rules for composing them (i.e., they have a strong *relational inductive bias* [70]). Specifically, our model is inspired by Message-Passing Neural Network (MPNN) [95], such models are already used in chemistry to develop new compounds. With this framework we design a new model that captures *meaningfully* traffic routing over network topologies. This is achieved by modeling the relationships of the links in topologies with the source-destination paths resulting from the routing schemes and the traffic flowing through them.

In order to test the accuracy of our model we train it with a dataset generated using a per-packet simulator (OMNet++ [96]) resulting in high estimation accuracy of delay and jitter (worst case $R^2 = 0.86$) when testing against topologies, routing and traffic not seen during training. More importantly, we verify that our model is able to generalize and for instance, when training the model with samples of a 14-node topology the model is able to provide accurate estimates in a never seen 24-node network.

Finally, and in order to showcase the potential of our model we present a series of use cases applicable to a SDN architecture:

(a) **Routing Optimization:** We first show that our model can be used to find routing schemes that minimize per-source-destination average delay and/or jitter. We benchmark it against traditional utilization-aware models (e.g., OSPF) achieving improvements up to 43.5%. We show that this model can be also used for SLA optimization, where delay or jitter SLA is maintained for a set of source-destination pairs even when the overall traffic volume increases.

(b) **Link failures:** In order to show the generalization capabilities of our model we show that it is able to produce estimates of delay and jitter in the presence of link failures. That is, changes in the topology and the routing.

(c) **What-if Scenarios:** Finally we show that the model can be used to reason in what-if scenarios answering the following questions: What will happen to the delay/jitter of the network if a new user is added? And, how should I upgrade the network to significantly reduce the overall delay and jitter?

The remainder of this chapter is structured as follows: first, Section 3.2 presents the foundations of GNN, including some background and a detailed description of how a basic GNN is structured. Then, Section 3.3 describes a generic optimization scenario within the KDN context. In Section 3.4, we describe RouteNet – the GNN model used to perform network optimization –, and Section 3.5 makes an extensive evaluation of the accuracy and generalization capabilities of this GNN model, including networks up to 50 nodes. Thereafter, we use RouteNet to perform network optimization in five relevant use cases (in Section 3.6). Lastly, we summarize some related work in Section 3.7 and conclude with a chapter summary in Section 3.8.

3.2. Graph Neural Networks

This section introduces the foundations of GNN and the applications they may potentially have to a plethora of problems where the information is fundamentally structured in the form of graphs. We include a detailed description of how a basic GNN architecture is constructed from input graphs and explain the main aspects that make GNNs unique DL models to generalize over graph-structured data with respect to other well-known neural network architectures vastly used in the literature (e.g., fully-connected NN, convolutional NN, recurrent NN, auto-encoders).

3.2.1. Background

A graph is a data type that describes a set of objects (vertices) that are somehow related between them (by edges) and is a fundamental way to represent a plethora of elements present in the nature. Some examples of this are molecules and compounds in the chemistry field, physical systems with interactions between objects (e.g., gravitational attractions in a group of celestial objects), user relations in social networks, or relations between elements in computer network topologies (e.g., forwarding devices, links, routing paths). In general, graphs enable to provide structured, dense and self-contained representations of systems composed by heterogeneous elements that may have arbitrary relations between them.

Other examples of data types that are vastly used to structure the data conveniently are: (*i*) sequences, which are used to represent sets of objects ordered on a temporal basis (e.g., a sequence of words defines a sentence), or (*ii*) arrays with grid elements ordered according to a spatial reference (e.g., digital images).

In the Deep Learning context [10], a lot of effort has been devoted to design neural network architectures well suited to understand and abstract deep knowledge from different widely used data structures as those described above. For instance, Recurrent Neural Networks (RNNs) [97] have demonstrated to be effective models to understand and keep memory of the temporal relations present in data structured as sequences. RNNs are able to learn and model sequential dependencies in data (e.g., sentences) and can apply this

knowledge to other sequences unseen before. For instance, in the field of language processing, RNNs have enabled to build DL models able to learn the syntax by themselves and perform efficient multi-language translation. Other, well-known neural network architectures with lots of applications in the field of image processing are Convolutional Neural Networks (CNNs) [98]. These neural networks are based on convolutional operations that are applied over images – more generically, arrays of grid elements spatially ordered – and are learned from "a clean slate" during the training phase. CNNs are invariant to spatial translations, and as a result they can apply their previous knowledge to images where the position and orientation of certain objects have been modified. For instance, CNNs have enabled to build DL models with an unprecedented capacity to find patterns at different scales on images and classify them.

The main essence behind these two neural network architectures (RNN and CNN) is that they are specifically tailored to understand and learn from the relational information contained in the data structure itself (i.e., temporal relations in sequences and spatial relations in images). RNNs and CNNs are respectively invariant to temporal and spatial translations and – as a result – they offer a strong relational inductive bias [70] that provides the models with an outstanding ability to generalize over data structured as sequences or images. This, for instance explains the capacity of some DL models based on Generative Adversarial Networks (GANs) that use CNNs (or RNNs) to generate new "fake" images (or text) that may resemble completely real.

In this context, GNNs [69, 70] is a family of NNs recently proposed with the aim of building DL models specifically tailored to operate and generalize over graph-structured data. Particularly, GNNs have a modular neural network architecture that represents explicitly the elements (nodes) in graphs and the relations (edges) between them. Table 3.1 shows a comparison of the properties offered by GNN in contrast to other well-known neural networks. GNNs offer support for input graphs of variable size and with arbitrary relations between elements and, what is more important, they are invariant to node and edge permutation. As a result, they offer a strong relational inductive bias over graph-structured information [70]. This has enabled to construct a new breed of DL models that are specifically tailored to abstract complex knowledge about the relations between elements in graphs and apply this knowledge to make inference in other graphs never seen before. Particularly, GNNs have already been successfully applied to many problems of different fields such as chemistry (e.g., molecular properties prediction [95]) or physics (e.g., trajectory prediction in n-body systems [99]) and are very promising for a plethora of applications (see Table 3.1).

In the computer networks field there are some few recent attempts that show the feasibility to construct efficient GNN-based network models able predict performance metrics in networks [100, 101] and combine them with optimization algorithms to address network-related problems such as job scheduling optimization [102].

Table 3.1: Properties of well-known neural network families and main applications.

Type	Relations	Invariance	Applications
Fully connected	All-to-all	-	**Feature-based classifiers** (e.g., user profile modeling, medical diagnosis, signature-based malware detection), etc
Recurrent	Sequential	Time translation	**Text Processing** (e.g., translation, Natural Language Processing, voice processing), **Time series analysis** (e.g., stock price prediction), etc
Convolutional	Local	Spatial translation	**Computer Vision** (e.g., face recognition, object identification, self-driving cars), **Augmented Reality** (e.g., video games), etc
Graph	Edges	Node, edge permutations	**Computer Science** (e.g., computer networks, job scheduling, web page ranking), **Chemistry** (e.g., compound generation, drug discovery), **Biology** (protein-protein interactions prediction), **Physics** (e.g., motion prediction and center of mass computation in n-body systems), **Social Sciences** (e.g., social networks), etc

3.2.2. Message-passing Architecture

One characteristic that make GNNs different from other well-known families of NNs is that their architecture is not fixed, instead it depends on the structure of the input graph. The main concept behind GNNs is an iterative message passing process where every node in the input graph has a hidden state (h_i) that is combined with the neighboring nodes according to the graph structure (Fig. 3.1). Every message passing iteration, the hidden state of a node i – typically represented by a n-element vector – is updated with the hidden states received from its neighbors, and this process is executed a number of times until the state of nodes converge to fixed values.

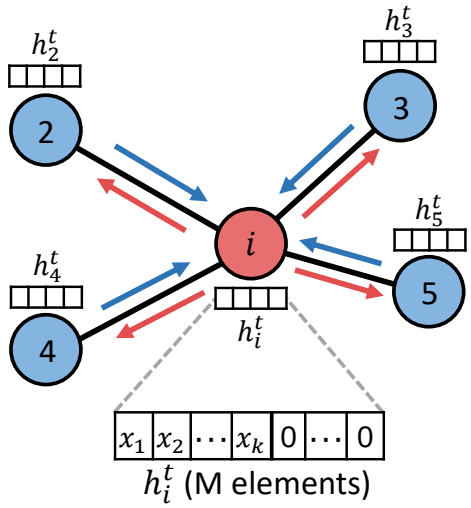


Figure 3.1: Message-passing scheme over a graph.

In this section, we describe in detail a generic message-passing GNN architecture according to the definition in [95].

First of all, the input representation for the GNN must be initialized based on the data provided in the input (graph-structured) sample. The hidden states of elements in the graph (i.e., the vertices) may be initialized with some features (x_i) describing the node. For instance, in the case of molecules a label may be used to identify the element type corresponding to each atom (e.g., hydrogen, nitrogen). Additionally, edges (e_{ij}) may contain some information describing the relation type (e.g., ionic, covalent bonds between atoms). Since the hidden states (h_i) are represented by n-element vectors that typically are larger than the initial feature vector (x_1-x_K), they are typically padded with zeros to fill the vector (Fig. 3.1). The vector size of h_i represents the amount of information that may be potentially encoded in the hidden state of each graph element.

Once all the hidden states are initialized, it is possible to execute the forward pass of the GNN, which may be divided in two phases: (i) the massage-passing phase and (ii) the readout phase.

Message-passing Phase

During this phase, an iterative process (message-aggregation-update) is executed over the input graph for T iterations. Figure 3.2 illustrates an iteration of the message passing process over one node of the graph. Note that this process may run in parallel for all the graph nodes. In this phase three main functions are applied over the graph elements:

(a) *Message function* (M): This function encodes in a message (m_{ij}) some information about the pair-wise relations between elements in graphs. Particularly, a message (m_{ij}) is a n-element vector that results from applying the M function to every node pair connected in the graph. This function may use as input the hidden states of two nodes connected (h_i and h_j) and some additional properties describing the relation type (e_{ij}) between them [i.e., $m_{ij} = M_{ij}(h_i, h_j, e_{ij})$].

(b) *Aggregation function* ($Aggr$): After all the messages (m_{ij}) are computed for every node in the graph with its corresponding neighbors, the messages received on each node are combined by using an aggregation function ($Aggr$) that results in an output fixed-size vector (m_i^*). The most common aggregation function is an element-wise summation, although other more elaborated functions such as gated summations may be used [103].

(c) *Update function* (U): This function is tasked to update the node hidden states at the end of a message passing iteration. Particularly, the function U is applied individually to the hidden state of every node (h_i^t) and the new message computed for such node during the current message passing iteration (m_i^{*t}). Intuitively, it produces an

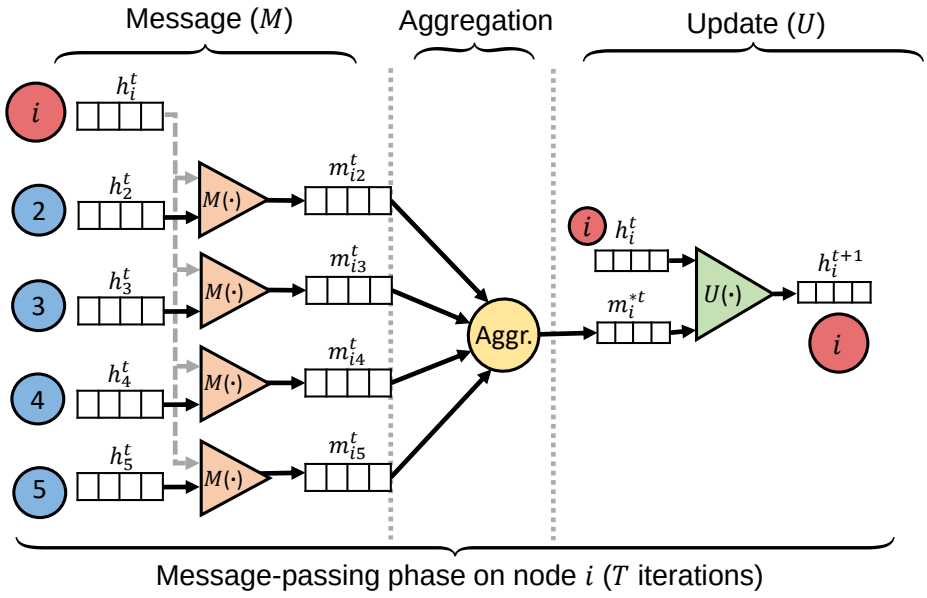


Figure 3.2: Message-passing phase over one graph node.

updated hidden state for the node (h_i^{t+1}) by combining the current state (h_i^t) and some information from the neighboring nodes encoded in the message vector (m_i^*). Then, the updated hidden states will be used in the next message passing iteration or during the readout phase after the T message-passing iterations are completed (see more details below).

Note that during each message-passing iteration a node receives only information from its neighbors. Then, to ensure that every node potentially receives information from all the other nodes in the graph, we need to run a number of iterations at least equal than the longest shortest path between all the node pairs in the graph (i.e., the graph diameter). This may be used as a reference to define the total number of message-passing iterations (T) needed for the hidden states to converge.

Readout phase

After the message-passing phase is completed, the resulting hidden states – which should have converge on some fixed values – are passed through the readout phase. This last phase is responsible for converting the encoded information in the hidden states to the output values or labels that the GNN eventually produces. Before continuing with this, it is important to note that GNNs may produce typically two different types of output: (i) node-level outputs and (ii) global outputs. For instance, following with the example of molecules, it may be desirable to predict properties individually on every atom of the molecule (e.g., energy level) or inferring global properties of the molecule (e.g., toxicity, flammability). In the first case, node-level outputs are typically obtained by applying a readout function (R_n) individually to the hidden state of every node in the graph (Fig. 3.3, top), while global outputs (Fig. 3.3, bottom) are typically computed by first aggregating the hidden states of all the nodes and then applying a global readout function (R_n). In this latter case, the application of an aggregation function allows the GNN to operate over an arbitrary number of nodes in the readout phase, i.e., it provides the GNN with the capacity to operate over variable-size graphs.

In the end, a GNN architecture is a combination of the four functions described above (M, $Aggr$, U and R). These functions are unique and are replicated over the GNN architecture based on the input graph structure. For instance, the message function (M) must be the same to compute the message over all the node pairs. Typically, M, U and R are functions approximated by three independent neural networks (e.g., fully-connected), while the $Aggr$ function is often a mathematical operation defined in the GNN design (e.g. element-wise summation).

Note that GNNs model end-to-end differentiable functions and thereby, it is possible to train them as end-to-end structures by applying a conventional backpropagation algorithm. This enables to make supervised training by applying samples labeled with some

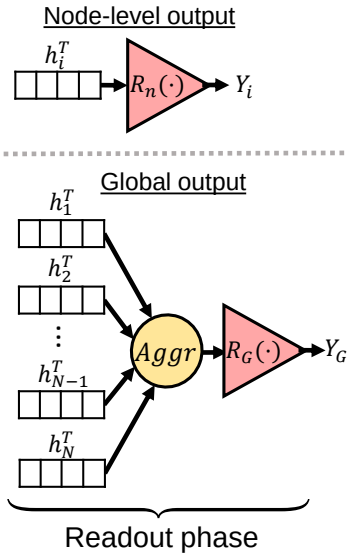


Figure 3.3: Readout phase.

global or node-level output features and learn the optimal parameters in those functions approximated by neural networks (i.e., M, U and R). Also note that in a GNN architecture there are multiple instances of the M, U and R functions that are concatenated based on the structure of the input graph, and these functions should be unique. Then, during the training phase – when the backpropagation algorithm is applied – all the instances of the same function share their parameters (i.e., weights and biases) and are jointly optimized. This forces these functions to be invariant to node and edge permutation.

Lastly, note that, although GNN is a family of NNs with a wide range of architectural variants (e.g., [69, 99, 104–106]), all of them are based on the principle of message passing between graph elements described above (i.e., message-aggregation-update).

3.3. Network Optimization Scenario

Network modeling enables the control plane to further exploit the potential of Software-Defined Networking (SDN) to perform fine-grained management. This permits to evaluate the resulting performance of what-if scenarios without the necessity to modify the state of the data plane. It may be profitable for a number of network management applications such as optimization, planning or fast failure recovery. For instance, in Figure 3.4 we show an architecture of a use case that performs network optimization within the context of the KDN paradigm [52]. In this case, we assume that the control plane receives timely updates of the network state (e.g., traffic matrix, delay measurements). This can be achieved by means of "conventional" SDN-based measurement techniques (e.g., OpenFlow [2], OpenSketch [107]) or more novel telemetry proposals such as INT for P4 [54] or iOAM [55]. Likewise, in the knowledge plane there is an optimizer whose behavior is defined by a given target policy. This policy, in line with intent-based networking, may be defined by a declarative language

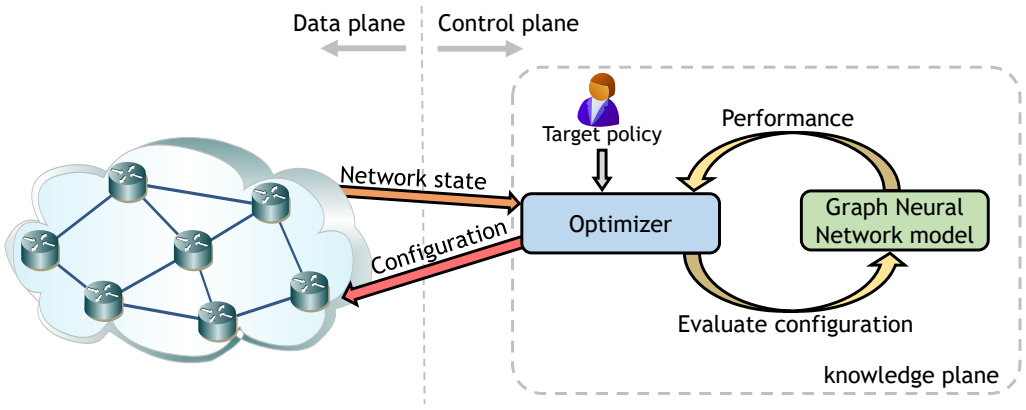


Figure 3.4: Network optimization architecture.

such as NeMo [65] and finally being translated to a (multi-objective) network optimization problem. At this point, an accurate network model can play a crucial role in the optimization process by leveraging it to run optimization algorithms (e.g., hill climbing) that iteratively explore the performance of candidate solutions in order to find the optimal configuration. We intentionally leave out of the scope of this architecture the training phase.

To be successful in scenarios like the one proposed above, the network model should meet two main requirements: (*i*) providing accurate results, and (*ii*) having a low computational cost to allow network optimizers to operate in short time scales. Moreover, it is essential for optimizers to have enough flexibility to simulate what-if scenarios involving different routing schemes, changes in the topology and variations in the traffic matrix. To this end, we rely on the capability of GNN to efficiently operate and generalize over environments represented as graphs. The GNN-based model described in this chapter, RouteNet, inspired by the MPNN model [95] used in the chemistry field, is able to propagate any routing scheme throughout a network topology and abstract meaningful information of the current network state. Figure 3.5 shows a schematic representation of the model.

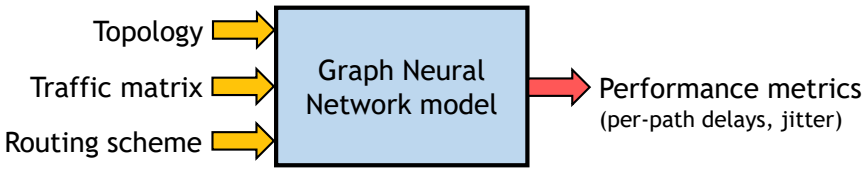


Figure 3.5: Scheme of RouteNet.

More in detail, RouteNet takes as input (*i*) a given topology, (*ii*) a source-destination routing scheme (i.e., relations between end-to-end paths and links) and (*iii*) a traffic matrix (defined as the bandwidth between each pair of nodes in the network), and finally produces performance metrics according to the current network state (e.g., per-path delays or jitter). To achieve it, RouteNet uses fixed-dimension vectors that encode the states of paths and

links and propagate the information among them according to the routing scheme. In Section 3.6, we provide some relevant use cases with experiments that exhibit how we can benefit from this GNN model in different network-related problems.

3.4. Network Modeling with GNN

In this section, we provide a detailed mathematical description of RouteNet, a GNN-based model designed specifically to operate in networking scenarios.

3.4.1. Notation

A computer network can be represented by a set of links $\mathcal{N} = \{l_i\}, \quad i \in (0, 1, \ldots, n_l)$, and the routing scheme in the network by a set of paths $\mathcal{R} = \{p_k\} \quad k \in (0, 1, \ldots, n_p)$. Each path is defined as a sequence of links $p_k = (l_{k(0)}, \ldots, l_{k(|p_k|)})$, where $k(i)$ is the index of the i-th link in the path k. The properties (features) of both links and paths are denoted by $\mathbf{x}_{l_i}$ and $\mathbf{x}_{p_i}$.

3.4.2. Message Passing on RouteNet

Let us consider the delay on path $p_k = (l_{k(0)}, l_{k(1)}, l_{k(2)} \ldots)$. The state of every link in this path and consequently, the associate delays, depend on all the traffic traversing these links. If packet loss is negligible, the order of links in the path does not matter. Then, the delay could be computed as $\sum_i d(l_{k(i)})$, where $d(l_j)$ represents the delay on the j-th link. However, the presence of links with losses introduces sequential dependence between the link states.

Let the state of a link be described by $\mathbf{h}_{l_i}$, which is an unknown hidden vector. Similarly, the state of a path is defined by $\mathbf{h}_{p_i}$. We expect the link state vector to contain some information about the link delay, packet loss rate, link utilization, etc. Likewise, the path state is expected to contain information about end-to-end metrics such as delays or total losses. Considering these assumptions, we can state the following principles:

1) The state of a path depends on the states of all the links in the path.

2) The state of a link depends on the states of all the paths including the link.

These principles can be matematically formulated with the following expressions:

$$\mathbf{h}_{l_i} = f(\mathbf{h}_{p_1}, \ldots, \mathbf{h}_{p_j}), \quad l_i \in p_k, k = 1, \ldots, j \tag{3.4.1}$$

$$\mathbf{h}_{p_k} = g(\mathbf{h}_{l_{k(0)}}, \ldots, \mathbf{h}_{l_{k(|p_k|)}}) \tag{3.4.2}$$

where f and g are some unknown functions.

It is well-known that neural networks can work as universal function approximators. However, a direct approximation of functions f and g is not possible in this case given that:

(i) Equations (3.4.1) and (3.4.2) define an implicit function (a nonlinear system of equations with the states being hidden variables), (ii) these functions depend on the input routing scheme, and (iii) the dimensionality of each function is very large. This would require a vast set of training samples.

Our goal is to achieve a structure for f and g being invariant for the routing scheme but still being aware of it. For this purpose, we present RouteNet as an effective GNN architecture to model these functions.

Algorithm 2 describes the forward propagation (and the internal architecture) of the network. In this process, RouteNet receives as input the initial path and link features $\mathbf{x}_p$, $\mathbf{x}_l$ and the routing description $\mathcal{R}$, and outputs inferred per-path metrics ($\hat{\mathbf{y}}_p$). Note that we simplified the notation by dropping sub-indexes of paths and links.

RouteNet's architecture enables dealing with the circular dependencies described in equations (3.4.1) and (3.4.2), and supporting arbitrary routing schemes (which are inherently represented within the architecture). This is all thanks to the ability of GNNs to

Input: $\mathbf{x}_p$, $\mathbf{x}_l$,$\mathcal{R}$
Output: $\mathbf{h}_p^T$, $\mathbf{h}_l^T$, $\hat{\mathbf{y}}_p$
// Initialize states of paths and links
foreach $p \in \mathcal{R}$ **do**
  $\mathbf{h}_p^0 \leftarrow [\mathbf{x}_p, 0 \ldots, 0]$
end
foreach $l \in \mathcal{N}$ **do**
  $\mathbf{h}_l^0 \leftarrow [\mathbf{x}_l, 0 \ldots, 0]$
end
for $t = 0$ *to* $T-1$ **do**
  // Message passing from links to paths
  foreach $p \in \mathcal{R}$ **do**
    foreach $l \in p$ **do**
      $\mathbf{h}_p^t \leftarrow RNN_t(\mathbf{h}_p^t, \mathbf{h}_l^t)$
      $\tilde{\mathbf{m}}_{p,l}^{t+1} \leftarrow \mathbf{h}_p^t$
    end
    $\mathbf{h}_p^{t+1} \leftarrow \mathbf{h}_p^t$
  end
  // Message passing from paths to links
  foreach $l \in \mathcal{N}$ **do**
    $\mathbf{m}_l^{t+1} \leftarrow \sum_{p:l \in p} \tilde{\mathbf{m}}_{p,l}^{t+1}$
    $\mathbf{h}_l^{t+1} \leftarrow U_t\left(\mathbf{h}_l^t, \mathbf{m}_l^{t+1}\right)$
  end
end
// Readout function
$\hat{\mathbf{y}}_p \leftarrow F_p(\mathbf{h}_p)$

Algorithm 2: Internal architecture of RouteNet.

address problems represented as graphs and solve circular dependencies by making an iterative approximation to fixed point solutions.

In order to address the circular dependencies, RouteNet repeats the same operations over the state vectors T times (loop from line 7). These steps represent the convergence process to the fixed point of a function from the initial states $\mathbf{h}_p^0$ and $\mathbf{h}_l^0$.

Regarding the issue of routing invariance (more generically known as topology invariance in the context of graph-related problems). This requires the use of a structure that is able to represent graphs of different topologies and sizes. In our case, we aim at representing different routing schemes in a uniform way. One state-of-the-art solution for this problem [100] proposes using neural message passing architectures that combine both: a representation of the topology as a graph, and vectors to encode the link states. In this context, RouteNet can be interpreted as an extension of a vanilla message passing neural network that is specifically suited to represent the dependencies among links and paths given a routing scheme [Equations (3.4.1) and (3.4.2)].

In Algorithm 2, the loop from line 9 and the line 16 represent the *message-passing* operations that exchange mutually the information encoded (hidden states) among links and paths. Likewise, lines 11 and 17 are *update* functions that encode the new collected information into the hidden states. The path update (line 11) is a simple assignment, while the link update (line 17) is a trainable neural network. In general, the path update could be also a trainable neural network.From a computational point of view, the loops over links and paths are the most expensive parts of the algorithm. An upper bound estimate of complexity is $O(n^3)$, where n is the number of nodes in the network. This assumes the worst case scenario, where all the paths have length n. Typically, the expected diameter in real networks is around $\log(n)$ (e.g., in Erdös-Rényi random graphs), thereby the complexity would decrease to $O\left(n^2 \log(n)\right)$.

This architecture provides flexibility to represent any source-destination routing scheme. This is achieved by the direct mapping of $\mathcal{R}$ (i.e., the set of end-to-end paths) to specific message passing operations among link and path entities that define the architecture of RouteNet. Thus, each path collects messages from all the links included in it (loop from line 9) and, similarly, each link receives messages from all the paths containing it (line 16). Given that the order of paths traversing the same link does not matter, we used a simple summation for the path-level message aggregation. However, in the case of links, the presence of packet losses may imply sequential dependence in the links that form every path. Consequently, we use a Recurrent Neural Network (RNN) for the link-level message aggregation. Note that RNNs are well suited to capture dependence in sequences of variable size (e.g., text processing). This allows us to model losses in links and propagate this information through all the paths. For an input sequence $\mathbf{i}_1, \mathbf{i}_2, \ldots$ and an initial hidden state $\mathbf{s}_0$, the output of a RNN is defined as:

$$(\mathbf{o}_t, \mathbf{s}_t) = RNN(\mathbf{s}_{t-1}, \mathbf{i}_t). \tag{3.4.3}$$

In our case, we use a simple version of a RNNs, where $o_t = s_t$.

Moreover, the use of these message aggregation functions (RNN and summation) enables to significantly limit the dimensionality of the problem. The purpose of these functions is to collect an arbitrary number of messages received in every (link or path) entity, and compress this information into fixed-dimension arrays (i.e., hidden states). Note that the size of the hidden states of links and paths are configurable parameters. In the end, all the hidden states in RouteNet represent an explicit function containing information of the link and path states. This enables to leverage them to infer various features at the same time. Given a set of hidden states $\mathbf{h}_p^T$ and $\mathbf{h}_l^T$, it is possible connect readout neural networks to estimate some path and/or link-level metrics. This can be typically achieved by using ordinary fully-connected networks with some layers and proper activation functions. In Algorithm 2, the function F_p (line 20) represents a readout function that predicts some path-level features ($\hat{\mathbf{y}}_p$) using as input the path hidden states $\mathbf{h}_p$. Similarly, it would be possible to infer some link-level features ($\hat{\mathbf{y}}_l$) using information from the link hidden states $\mathbf{h}_l$.

3.4.3. Delay Model

RouteNet is a general neural architecture capable of modeling various network performance metrics. In order to apply it to particular problems, the following design decisions may be considered: (*i*) the size of the hidden states for both paths ($\mathbf{h}_p$) and links ($\mathbf{h}_l$), (*ii*) the number of message passing iterations (T), and (*iii*) the neural network architectures for RNN, U, and Fp. The last decision may be the most complex one, since there are multiple types of neural networks and possible configurations. In our particular case, where we use RouteNet to model per-path delays, we decided to use Graph Recurrent Units (GRUs) for both U and RNN. The reason behind this, is that GRUs are simpler than Long Short-Term Memory (LSTM) networks, since there is no output gate and, *a priori*, can achieve comparable performance [108]. GRUs are recurrent units that have an internal structure that by design reuses weights (i.e., weight tying), which considerably simplifies the model.

We modeled the readout function (Fp) with a fully-con-nected neural network and use Scaled Exponential Linear Unit (SELU) activation functions in order to achieve desirable scaling properties [109]. These hidden layers are interleaved with two dropout layers.

The dropout layers play two important roles in the model. During training, they help to avoid overfitting, and during the inference, they can be used for Bayesian posterior approximation [110]. This allows us to asses the confidence of the network predictions and avoid the issue of adversarial examples [111]. Typically, when a neural network is optimized to minimize the error for a particular output, the solution may be too optimistic. Thus,

repeating an inference multiple times with random dropout can provide a probabilistic distribution of results, and this distribution (e.g., the spread) can be used to estimate the confidence of the predictions.

3.4.4. Jitter Model

In order to assess the ability of RouteNet to generalize to different network metrics, we took a model in an early training stage that produces delay estimates and trained it to produce per-path jitter estimates. The main difference between these two metrics is in the scaling factor, since they are closely related but the jitter spans on different range than the delay.

3.5. Evaluation of the Accuracy of the GNN Model

In this section, we evaluate the accuracy of RouteNet (Sec. 3.4) to estimate the per-source-destination mean delays and jitter in different network topologies and routing schemes.

3.5.1. Simulation Setup

In order to build a ground truth to train and evaluate the GNN model, we implemented a custom-built packet-level simulator with queues using OMNeT++ (version 4.6) [96]. In this simulator, the delay and jitter modeled in each queue are related to the bandwidth capacity of the corresponding egress links. For each simulation, we measure the average end-to-end delay and jitter experienced during 16k units of time by every pair of nodes. We model the traffic matrix ($\mathcal{TM}$) for each S-D pair in the network as:

$$\mathcal{TM}(S_i, D_j) \; = \; \mathcal{U}(0.1, 1) * TI/(N-1) \quad \forall \, i, j \in nodes, i \neq j \tag{3.5.1}$$

Where $\mathcal{U}(0.1, 1)$ represents a uniform distribution in the range [0.1, 1], TI represents a parameter to tune the overall traffic intensity in the network scenario and N is the number of nodes in the network.

To train and evaluate the model, we used the 14-node and 21-link NSF network topology [87] (Fig. 2.4 of Chapter 2). Moreover, we use the 24-node Geant2 topology (Fig. 3.6) and the 17-node German Backbone Network (GBN) [88] – depicted in Fig. 2.6 of Chapter 2 – only for evaluation purposes. For the sake of simplicity, we consider the same capacity for all the links in these networks and vary the traffic intensity in each scenario.

3.5.2. Training and Evaluation

We implemented the RouteNet models of delay and jitter in TensorFlow. The source code and all the training/evaluation datasets used in this chapter are publicly available at [112].

We trained both models (delay and jitter) on a collection with 260,000 training samples from the NSF network generated with our packet-level simulator. Despite this dataset only

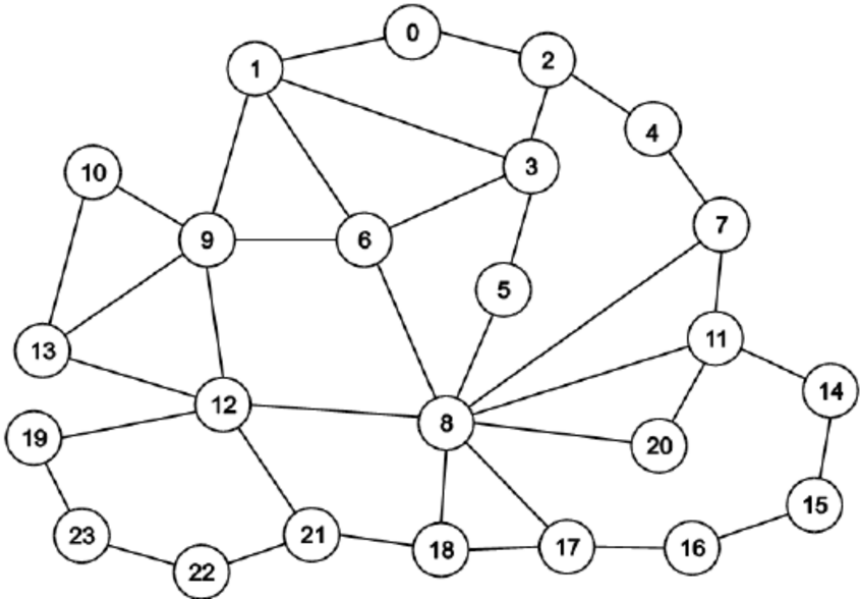


Figure 3.6: 24-node Geant2 topology (image extracted from [1]).

contains samples from single topology, it includes around 100 different routing schemes and a wide variety of traffic matrices with different traffic intensity. For the evaluation, we use 30,000 samples.

In our experiments, we select a size of 32 for the path's hidden states ($\mathbf{h}_p$) and 16 for the link's hidden states ($\mathbf{h}_l$). The initial path features ($\mathbf{x}_p$) are defined by the bandwidth that each source-destination path carries (extracted from the traffic matrix). In this case, we do not add initial link features ($\mathbf{x}_l$). Note that, for larger networks, it might be necessary to use larger sizes for the hidden states. Moreover, every forward propagation we execute T=8 iterations. The Dropout rate is equal to 0.5, this means that each training step we randomly deactivate half of neurons in the readout neural network. This also allows us to make a probabilistic sampling of results and infer the confidence of the estimates.

During the training we minimized the Mean Squared Error (MSE) between the prediction of RouteNet and the ground truth plus the $L2$ regularization loss ($\lambda = 0.1$). The loss function is minimized using an Adam optimizer with an initial learning rate of 0.001. This rate is decreased to 0.0003 after 60,000 training steps approximately.

We executed the training over 300,000 batches of 32 samples randomly selected from the training set. In our testbed with a GPU Nvidia Tesla K40 XL, this took around 96 hours ($\approx$ 27 samples per second). Figure 3.7 shows the loss during the training process. Here, we observe that the training is stable and the loss drops quickly.

Table 3.2 shows a summary of all the experiments we made in the three different network topologies. We report two statistics: (i) the Pearson correlation ρ and (ii) the percentage of variance explained by the model (R^2). For the Geant2 and GBN topologies,

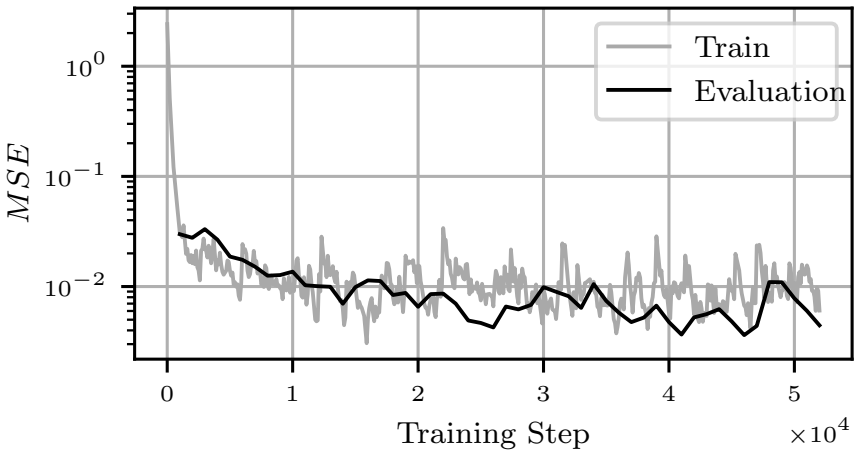


Figure 3.7: Exponentially smoothed MSE for the delay RouteNet model.

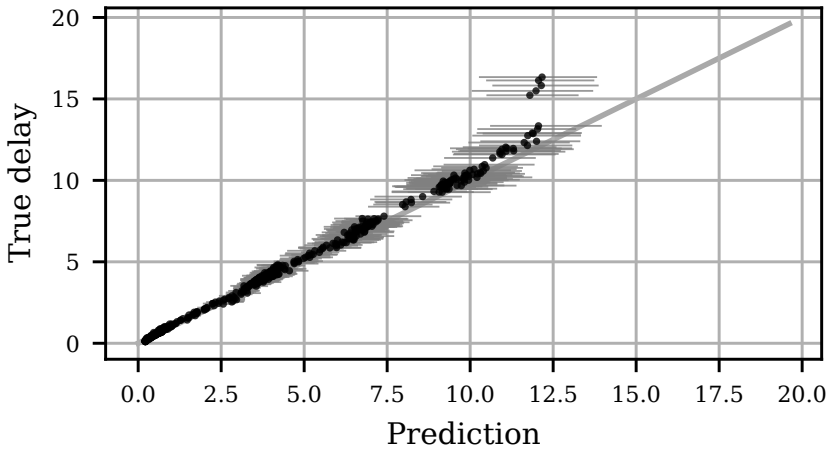
we tested the accuracy over a dataset with 100,000 samples. For the NSF network, we utilize the same 30,000 samples used for evaluation during the training process. To calculate the statistics in Table 3.2, we compute for each sample in the evaluation dataset 50 independent predictions using random dropout and take the median value. Note that *the Geant2 and GBN networks were never included in the training.* The model was only trained with samples from the NSF network (14 nodes). The high accuracy on Geant2 (24 nodes) and GBN (17 nodes) networks reveals the ability of RouteNet to well generalize even to larger networks.

Table 3.2: Summary of the accuracy results of RouteNet.

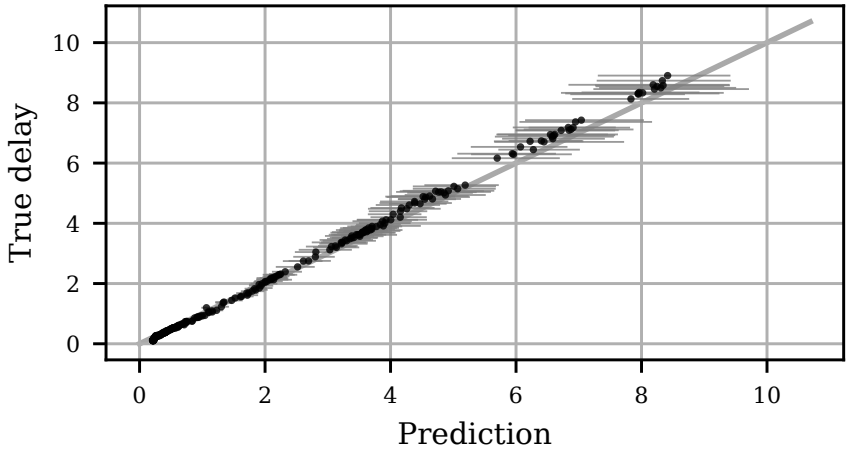
	NSF		**Geant2**		**GBN**	
	Delay	**Jitter**	**Delay**	**Jitter**	**Delay**	**Jitter**
R^2	0.99	0.98	0.97	0.86	0.99	0.97
ρ	0.998	0.993	0.991	0.942	0.997	0.987

This generalization capability is partly thanks to the Bayesian nature of the network (i.e., the use of layers with random dropout). Figure 3.8 shows an example of the probabilistic prediction for a single sample of the dataset of Geant2 (Fig. 3.8a) and GBN (Fig. 3.8b). The dots represent the median value of the predictions of RouteNet, while gray lines show the range containing 95% of the results.

Statistics like ρ or R^2 provide a good picture of the general accuracy of the model. However, there are more elaborated methods that offer a more detailed description of the model behavior. One alternative to gain insight into prediction models is to provide regression plots with the evaluation results. Thus, in Figure 3.8 we present relevant regression plots for specific scenarios of the evaluation in Geant2 (Fig. 3.8a) and GBN (Fig. 3.8b). However, it is not functional showing a regression plot with all the points predicted by RouteNet in all the evaluation scenarios (dozens of millions of points). Hence, we focus



(a) Geant2 network topology



(b) GBN topology

Figure 3.8: Regression plots with uncertainty.

on the distribution of residuals (i.e., the error of the model). Particularly, we present a Cumulative Distribution Function (CDF) of the relative error (Fig. 3.9) over all the evaluation samples. This allows us to provide a comprehensive view of the whole evaluation in a single plot. In these results, we can observe that the prediction error in general is considerably low. Moreover, we see that the jitter model is more biased compared to the delay model. This can be explained by the fact that this model was trained from a model previously trained for the delay (Sec. 3.4.4), while the delay model was optimized from the beginning to predict delays. Nevertheless, this shows the feasibility to adapt pre-trained models optimized for a specific metric to predict different network performance metrics (i.e., transfer learning [113]).

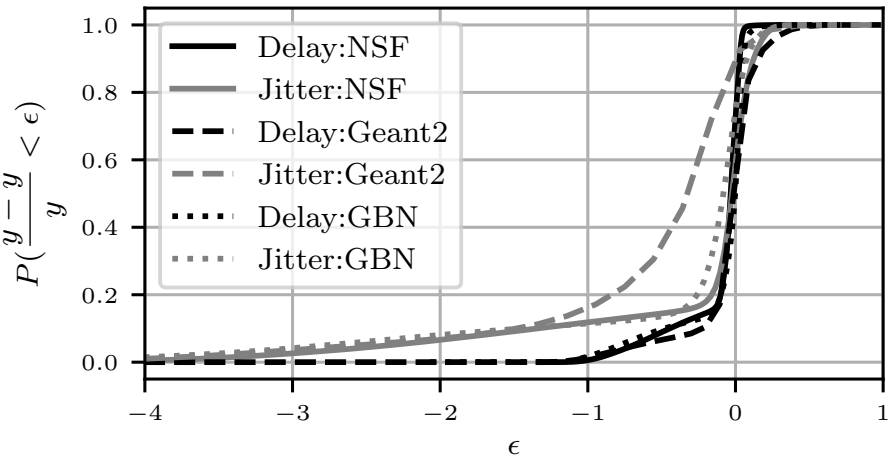


Figure 3.9: CDF of the relative error. Solid line for NSF, dashed line for Geant2, dotted line for GBN. y is the true value and $\hat{y}$ denotes the model prediction.

3.5.3. Challenging the Generalization Capabilities of RouteNet

This section further analyzes the generalization capabilities offered by RouteNet. To this end, more experiments are made including topologies up to 50 nodes. After this, we add a brief reflection about the generalization properties and the limitations of the RouteNet models evaluated in this chapter.

For this evaluation, we adapt RouteNet to support network topologies with variable link capacities. This is done by adding the link capacity as an initial feature of the links' hidden states. Also, we have optimized a set of hyperparameters to adapt the model to scenarios with larger topologies and more complex routing schemes. Thus, we trained RouteNet to estimate delays on a dataset with 480,000 samples generated by the custom-built packet-level simulator also used in the previous experiments (see Sec. 3.5.1). This includes samples from two topologies: (*i*) the 14-node NSF network topology and (*ii*) a 50-node synthetically-generated topology (depicted in Fig. 3.10). Every topology was simulated with a wide variety of routing schemes and traffic matrices with different traffic intensity.

After the training, we test the model on a different dataset with samples not present during training. Particularly, this evaluation dataset contains 120,000 unseen samples simulated in the two topologies where RouteNet was trained (14-node and 50-node). Additionally, to test the capability of RouteNet to generalize to topologies of variable size, we made a separate evaluation over a collection of 300,000 samples simulated in a third topology, the 24-node Geant2 (Fig. 3.6). Note that while the datasets used in the previous section considered the same capacity on all the links, these new datasets include variable link capac-

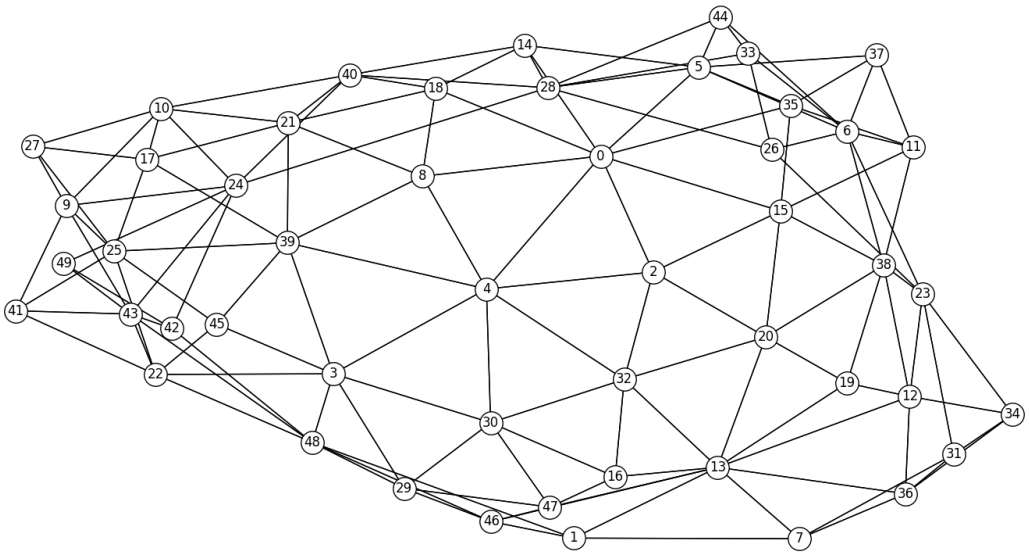


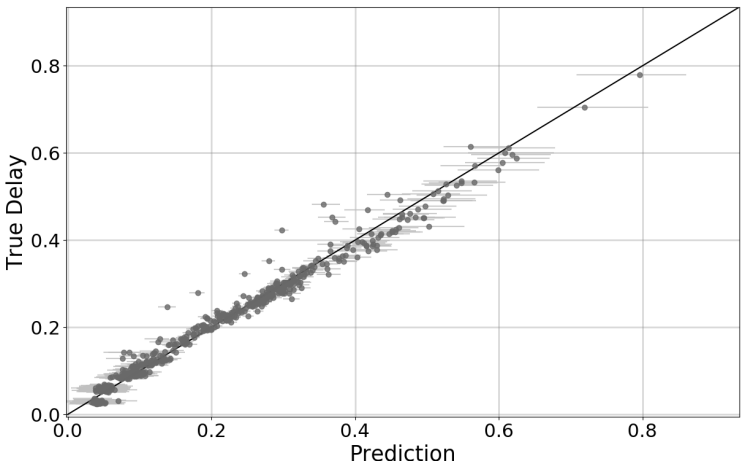
Figure 3.10: 50-node synthetically-generated topology (image extracted from [1]).

ities on the three different topologies simulated. The source code, the delay model already trained, and the training/evaluation datasets used are publicly available at [114].

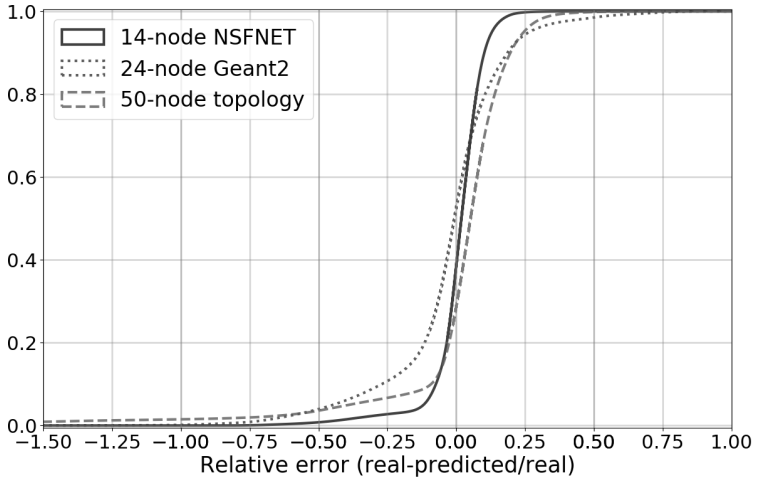
As a result, we observe that RouteNet still produces accurate estimates even in unseen topologies with variable link capacity. For instance, Figure 3.11a shows a regression plot of the delays predicted by RouteNet in a sample scenario of the Geant2 topology. Likewise, in Figure 3.11b we provide the CDF of the relative error between RouteNet's predictions and the real delay values in our evaluation datasets. Particularly, this plot includes the relative error over all the evaluation samples of the three different topologies.

After these evaluation results, we discuss the generalization capabilities and limitations of RouteNet. As in all ML-based solutions, RouteNet is expected to provide more accurate inference as the distribution of the input data is closer to the distribution of the training samples. In our case, it involves topologies with similar number of nodes and distribution of connectivity, routing schemes with similar patterns (e.g., variations of shortest path) and similar ranges of traffic intensities. Nevertheless, the previous evaluations have shown an extraordinary capability to accurately model networks from 14 to 50 nodes. In order to expand the generalization capabilities of RouteNet, an extended training set could be used including more variability on the input elements (i.e., toplogies, routing, traffic).

RouteNet's architecture is built to estimate path-level metrics using information from the output path-level hidden states. However, it is relatively easy to change the architecture and use information encoded in the link-level hidden states to produce link-related metrics inference (e.g., congestion probability on links).



(a) Regression plot in a sample scenario of Geant2



(b) CDF of the relative error

Figure 3.11: Evaluation results of accuracy in the extended evaluation.

3.6. Network Optimization Use Cases

In this section, we present some use cases to illustrate the potential of RouteNet (Section 3.4) to be used in relevant network optimization tasks. In these use cases, we use the delay (Section 3.4.3) and jitter (Section 3.4.4) models of RouteNet to evaluate the resulting performance after applying different network configurations. Particularly, we limit the optimization problem to evaluate a given set of candidate configurations (e.g., routing schemes) and select the one that results in better performance according to a given target policy. We compare the performance achieved by our optimizer using the delay/jitter estimates of RouteNet to the results obtained by the same optimizer using measurements of the links' utilization. As a reference, we also provide the results obtained when applying a

traditional Shortest Path routing policy. Note that more elaborated state-of-the-art optimization strategies (e.g., [115]) could – and most likely will – result in better performance than these baselines and also could be combined with RouteNet to further improve the resulting performance. However, we leave the analysis of such techniques as future work, since the purpose of this section is to show the added value of using the lightweight and accurate delay/jitter models provided by RouteNet to perform delay/jitter-aware network optimization.

In this context, using state-of-the-art delay/jitter models to perform online network optimization is typically unfeasible since these models often result into inaccurate estimation (e.g., theoretical models) and/or prohibitive processing cost (e.g., packet-level simulators). All the evaluation in this section is carried in the NSF network topology.

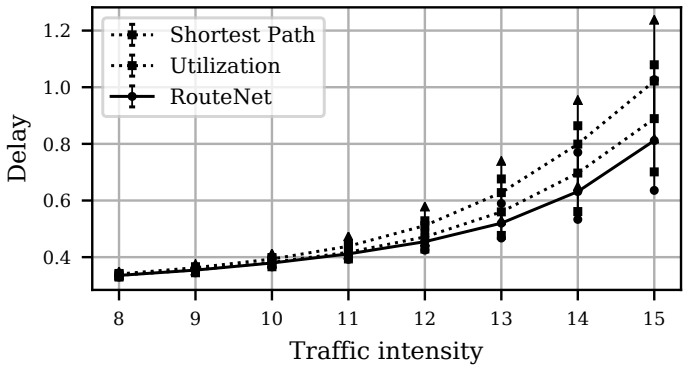
3.6.1. Delay and Jitter-based Routing Optimization

In this use case, the objective is to optimize multiple Key Performance Indicator (KPI) of the network. In particular, we made different experiments where the optimizer aims to: (*i*) minimize the mean end-to-end delay and jitter, and (*ii*) minimize the maximum delay and jitter experienced among all the source-destination pairs.

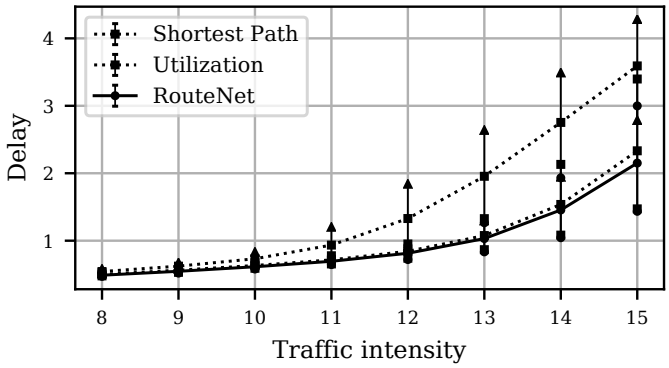
We compare the optimal routing policy obtained by Route-Net with two traditional approaches: (*i*) Shortest Path (SP) routing, where we compute different SP schemes using the Dijkstra algorithm, and (*ii*) a more elaborated routing optimizer whose objective is to minimize the bandwidth utilization. This latter strategy represents an upper-bound of the results that could be obtained by traditional routing optimizers based on links' utilization. In particular, for the case of minimizing the mean delay/jitter, we select the routing scheme with minimum mean utilization. In the case of minimizing the maximum delay/jitter, we select the scheme that minimizes the utilization of the link more loaded.

We evaluated the performance obtained by all the routing approaches varying the traffic intensity. Moreover, for a fair comparison, all of them perform the optimization over the same set with 100 different routing schemes randomly generated.

Figure 3.12a shows the minimum mean delay obtained w.r.t. the traffic intensity. Note that traffic intensities (in the x-axis) are in TI units according to the expression in Section 3.5.1 to generate traffic matrices ($\mathcal{T}$). Moreover, for each traffic intensity, we randomly generated 100 different traffic matrices (TMs) with various per-source-destination traffic distributions. The lines show the average results over the experiments (with those 100 TMs) and the error bars represent the 20/80 percentiles. Likewise, in Figure 3.12b we show the results for the use case where all routing techniques aim at minimizing the maximum end-to-end delay.



(a) Optimization of mean delay



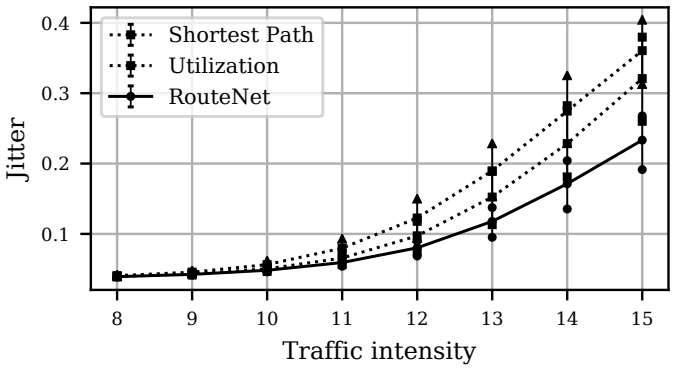
(b) Optimization of maximum delay

Figure 3.12: Delay-based optimization.

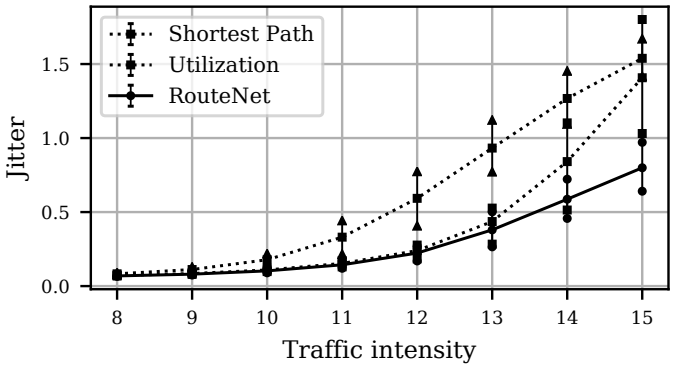
The same experiments were made to evaluate the results optimizing the mean (Fig. 3.13a) and the maximum (Fig. 3.13b) jitter experienced by the source-destination pairs in the network.

Considering these results, we can see that, as expected, the performance achieved by the different routing techniques does not differ with low traffic intensity (TI$\leq$9). However, the optimizer based on RouteNet delay estimations begins to achieve better performance with medium traffic intensity (TI=10-13) and, for high traffic intensity (TI=13-15), it achieves considerable higher performance. Particularly, with TI=15, it obtains the following results:

- When optimizing the mean delay and jitter, the RouteNet-based optimizer achieves a reduction of 20.87% in delay and 35.27% in jitter with respect to the SP policy. Likewise, it achieves 12.18%/27.21% lower delay/jitter than the utilization-based optimizer.
- When optimizing the maximum delay/jitter, the RouteNet-based optimizer achieves 40.08%/48.09% lower delay/jitter than the SP policy, and 8.11%/43.53% lower delay/jitter than the utilization-based optimizer.



(a) Optimization of mean jitter



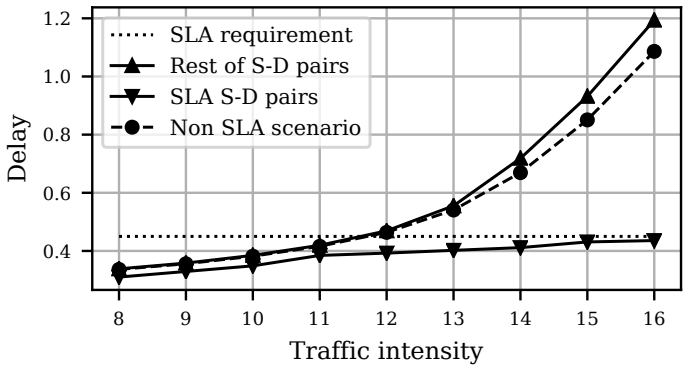
(b) Optimization of maximum jitter

Figure 3.13: Jitter-based optimization.

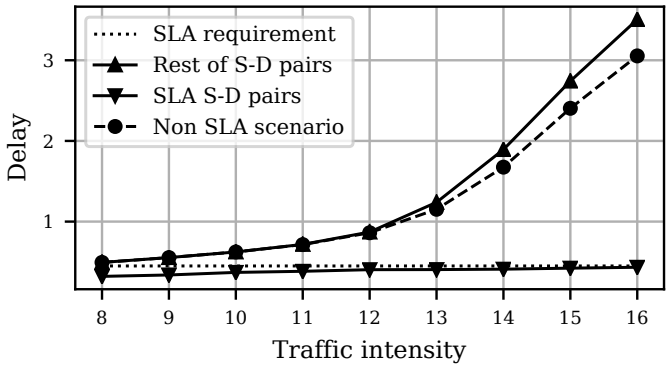
3.6.2. SLA Optimization

This use case represents a network scenario where the routing optimizer must comply a Service-Level Agreement (SLA) for some specific clients, while minimizing the impact on the performance of the rest of users in the network. In particular, we consider 4 source-destination pairs to have specific delay requirements. We made the experiments in the NSF network and selected the following source-destination pairs (S-D pairs) that must comply a certain delay requirement: (0,3) (3,4) (3,5) (3,6). Then, the objective is to optimize the routing configuration to guarantee that the traffic among those sources and destinations is below the target delay.

Figure 3.14a shows the results in the case that the RouteNet-based optimizer aims to optimize the mean delay experienced in the network, while Figure 3.14b shows the results for the case of minimizing the maximum delay for all the source-destination pairs. In these figures, the dashed line (labeled as "Non SLA scenario") represents the results if the optimizer does not distinguish between different traffic classes, and the solid lines represent the results after applying the optimal routing scheme that complies the SLA of the 4 S-D pairs. The dotted line represents the delay requirement of the S-D pairs with



(a) Optimization of mean delay



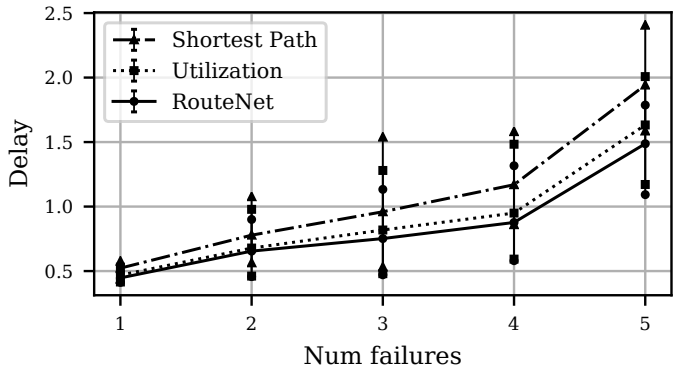
(b) Optimization of maximum delay

Figure 3.14: Delay optimization under SLA guarantees.

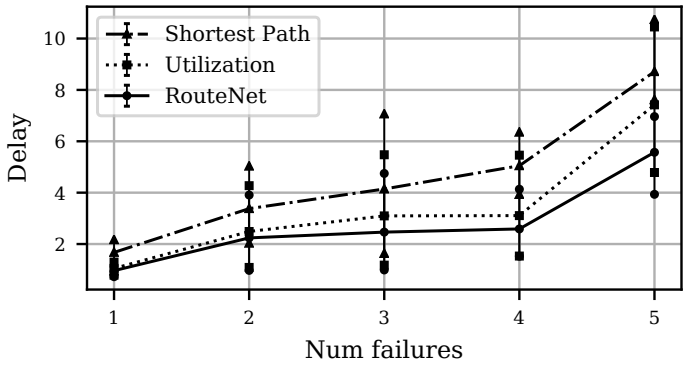
SLA, which is an input parameter of the optimizer. Then, we can observe that for the optimization case that considers the SLAs, the delay experienced by the 4 S-D pairs with SLA (labeled as "SLA S-D pairs") fulfills the delay requirements (dotted line) even with high traffic intensities (TI=13-16). Moreover, we observe that the rest of S-D pairs without SLA requirements (labeled as "Rest of S-D pairs") do not experience a great increase in the mean/maximum delays compared to the "non SLA scenario". For instance, with high traffic intensity (TI=15), in the case of optimizing the mean delay, the rest of the traffic only experiences an increase of 9.9% in the average delay (14.8% in the case of optimizing maximum delay).

3.6.3. Robustness Against Link Failures

In this use case, we show how our model is able to generalize in the presence of link failures. When a certain link fails, it is necessary to find a new routing that avoids this link to reroute the traffic. As the number of links failures increases, less paths are available and the network becomes more saturated.



(a) Mean delay



(b) Maximum delay

Figure 3.15: Delay optimization under the presence of different link failures.

We evaluate the performance of the aforementioned methods under the presence of link failures following the same methodology than in the first use case (see Sect. 3.6.1). The initial network state is a low traffic intensity scenario (TI=8).

Figure 3.15 shows the optimized mean delay (Fig. 3.15a) and the optimized max delay (Fig. 3.15b). Each point in the graph corresponds to the optimal delay obtained under 10 random possible links failures. We observe that, as shown in the first use case, the mean and the maximum delays increase as the network is more congested and, in these scenarios, the RouteNet-based optimization mechanism outperforms traditional approaches.

3.6.4. What-if Scenarios

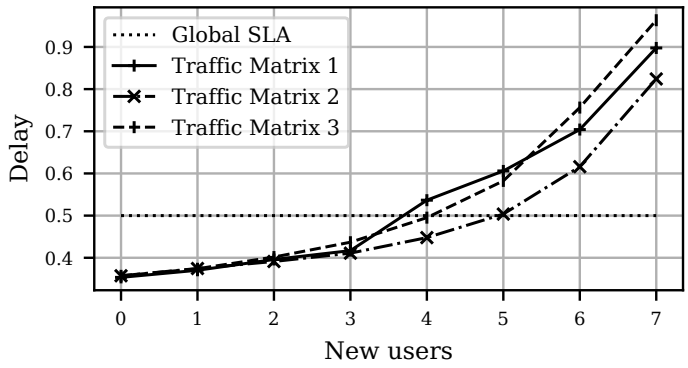
One application of interest of network modeling is that network operators can simulate hypothetical what-if scenarios to evaluate the resulting performance before making strategic decisions. These decisions, for instance, include making agreements to route a considerable bulk of traffic from other network (e.g., BGP peering agreements) or finding a network upgrade that results more beneficial given a limited budget.

Adding new users

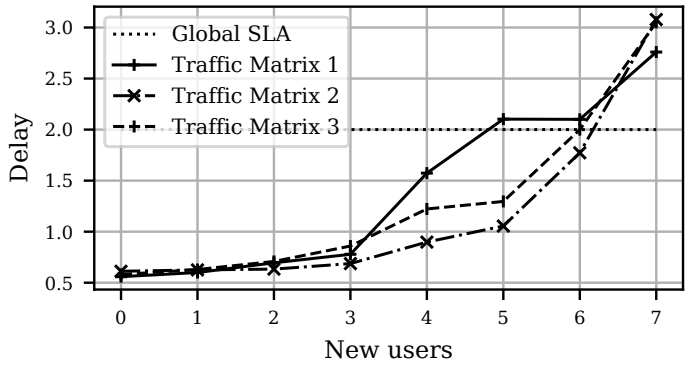
The objective of this use case is to evaluate the performance of the network under the presence of potential new users. Each new user in the network increases the amount of traffic that it has to support, and consequently the average and the maximum delay are increased.

Specifically, we explore when certain delay requirements cannot be fulfilled as the number of users with high bandwidth requirements increases. We model these new users as follows: each user multiplies by 2.5 the existing bandwidth demand in a certain node, the first user is connected to node 10, the second one to node 2, the third one to node 8, the fourth one to node 5, the fifth one to node 12, the sixth one to node 1, the seventh one to node 7 and the last one to node 0. We repeat this process under 3 different traffic matrices with initial low traffic intensity (TI=8).

Figure 3.16 shows the mean and maximum delay as new users are subscribed to the network. The dotted line represents the delay requirement, whereas the other lines represent the delay obtained with these different traffic matrices. We observe that the RouteNet



(a) Mean delay



(b) Maximum delay

Figure 3.16: Delay optimization as a function of the number of new users.

model is able to predict the future performance of the network and to know "a priori" when the delay requirements will not be accomplished. For example, we observe that a network operating with TM_1 will require an update before than the networks operating under the other traffic matrices.

Budget-constrained network upgrade

In this final use case, we address a common problem in networking, how to optimally upgrade the network by adding a new link between two nodes. For this, we take advantage of the RouteNet-based model to explore different options to place this new link to select the one that minimizes the mean delay.

Table 3.3 shows the optimal new placement in the NSF network topology under 10 different traffic matrices with high traffic intensity (TI=15). For each, we also show the average delay before placing the link, the obtained delay with the new optimal link and the delay reduction achieved. We observe that we can achieve an important reduction on the average delay by properly choosing between which nodes this new link is deployed. Note that the optimal placement for the new link depends on the traffic conditions in the network.

Table 3.3: Analysis of the optimal link placement under different traffic matrices.

Traffic matrix	**Optimal link placement**	**Previous delay**	**Delay with new link**	**Delay reduction**
TM_1	(1, 9)	0.732	0.478	35.7 %
TM_2	(2, 13)	0.996	0.464	53.4 %
TM_3	(1, 9)	1.179	0.516	56.2 %
TM_4	(2, 11)	0.966	0.518	46.37 %
TM_5	(1, 11)	0.908	0.509 2	44.7 %
TM_6	(0, 13)	0.811	0.484	40.3 %
TM_7	(1, 12)	0.842	0.485	42.4 %
TM_8	(1, 11)	0.770	0.431	44.0 %
TM_9	(1, 9)	1.009	0.492	51.2 %
TM_{10}	(2, 11)	1.070	0.491	54.1 %

3.7. Related Work

Network modeling with deep neural networks is a recent topic proposed in the literature [24, 52] with few pioneering attempts. The closest works to our contribution are first Deep-Q [93], where the authors infer the QoS of a network using the traffic matrix as an input using Deep Generative Models. And second [94], where a fully-connected feed-forward neural network is used to model the mean delay of a set of networks using as input the traffic matrix, the main goal of the authors is to understand how fundamental network characteristics (such as traffic intensity) relate with basic neural network parameters (depth of the neural network). RouteNet is also able to produce accurate estimates of performance

metrics -delay and jitter-, but it does not assume a fixed topology and/or routing, rather it is able to produce such estimates with arbitrary topologies and routing schemes not seen during training. This enables RouteNet to be used for network operation, optimization and what-if scenarios.

Finally, an early attempt to use GNN for computer networks can be found in [101]. In this case the authors use a GNN to learn shortest-path routing and max-min routing using supervised learning. While this approach is able to generalize to different topologies it cannot generalize to different routing schemes beyond the ones for which has been specifically trained. In addition the focus of the paper is not to estimate the performance of such routing schemes.

3.8. Chapter Summary

SDN has brought an unprecedented degree of flexibility to network management, which allows the network controller to configure the network behavior up to the flow-level granularity. This flexibility combined with the information provided by network telemetry opens many possibilities for online network optimization.

However, existing network modeling techniques based on analytic models cannot handle this huge complexity. As a result, current optimization approaches are limited to improve a global performance metric, such as network utilization. Although DL is a promising solution to handle such complexity and to exploit the full potential of the SDN paradigm, previous attempts to apply DL to networking problems still offer limited capabilities to generalize to network scenarios different from those seen during the training.

This chapter has presented RouteNet, a new type of Graph Neural Network (GNN) that is specifically designed for modeling computer networks. RouteNet is inspired by the Message-Passing Neural Network (MPNN) previously proposed to address a problem in the field of quantum chemistry. The main innovation behind RouteNet is a novel message-passing process that allows the GNN to capture the complex relationships between the paths and links that form a network topology and the routing configuration.

We used RouteNet to model the per-source-destination delay and jitter of networks. Our results show that RouteNet is able to generalize to other network topologies (up to 50 nodes), routing configurations, and traffic matrices that were not present in the training set. Finally, we presented some relevant use cases that show the potential of RouteNet to be applied for network optimization in SDN. Particularly, we showed that RouteNet can be used in KDN-based networks to optimize multiple Key Performance Indicators (KPIs) and to guarantee the Service-Level Agreement (SLA) of a particular set of flows, as well as to analyze different what-if scenarios.

Part II

Network Analytics

Chapter 4

Flow-level Measurement and classification in Software-Defined Networks

Obtaining flow-level measurements similar to those provided by NetFlow/IPFIX, is a challenging task in OpenFlow-based Software-Defined Networks, as it requires to install per-flow entries in the flow tables. This approach does not scale well, since the number of entries in the flow tables is limited and small. Moreover, labeling the flows with the applications that generate the traffic would greatly enrich these reports, with numerous potential applications on different network optimization and security-related tasks. In this chapter, we present a scalable flow monitoring solution fully compatible with current off-the-shelf OpenFlow switches. In the proposed system, measurements are maintained in the switches and are asynchronously sent to SDN controllers. Additionally, flows are classified using a combination of Deep Packet Inspection (DPI) and Machine Learning (ML) techniques, with special focus on the identification of web and encrypted traffic. For the sake of scalability, we designed two different traffic sampling methods depending on the OpenFlow features available in the switches. We implemented our monitoring solution within the OpenDaylight SDN controller and evaluated it in a testbed with Open vSwitch, using also some DPI and ML tools. Particularly, our experiments are directed to find the best tradeoff between accuracy and performance. The evaluation results – using real-world traffic from three different networks – show that the proposed measurement and classification systems can achieve good accuracy and still maintain a reasonable deployment cost.

4.1. Introduction

Traffic monitoring is a cornerstone for network operation and optimization, as it provides essential information for different tasks such as traffic engineering, security, or troubleshooting. With the advent of the Software-Defined Networking (SDN) paradigm, it is even more important to perform fine-grained monitoring to optimally exploit the possibilities offered by a centralized control plane, which can make decisions with a global view of the network.

Nowadays, one of the most deployed solutions in legacy networks for network monitoring is NetFlow/IPFIX. There are plenty of tools in the market based on NetFlow that harness the information provided by flow-level measurement reports to infer some meaningful information about the network. These tools also include a wide variety of services such as bandwidth monitoring, traffic classification, or anomaly detection.

In the SDN context, the OpenFlow protocol [2] has become a dominant standard for the communication in the Southbound interface (i.e., between the control and data planes). This protocol enables to maintain records with flow statistics in the switches, and includes an interface that permits to collect this information passively or actively in the control plane. These features can be leveraged to generate flow-level measurement reports as those of NetFlow.

An inherent issue in SDN is its scalability. Thus, for a proper design of a monitoring system, it is of paramount importance to consider the network and processing overheads to store and collect the flow statistics. First, given that controllers typically manage a large amount of switches in the network, it is important to reduce their processing load as much as possible. Second, the most straightforward way of implementing per-flow monitoring with OpenFlow is by maintaining an entry for each flow in a table of the switch. Thus, monitoring all the flows in the network results in a great constraint, since nowadays OpenFlow commodity switches do not support a large number of flow entries due to their limited hardware resources (i.e., the number of TCAM entries and processing power) [116].

Additionally, flows in the measurement reports have often been labeled (e.g., by protocol) using port-based classification techniques. However, these techniques are becoming increasingly obsolete, since they are not well suited to discover new applications generating significant traffic volumes on the Internet. Nowadays, it is becoming more frequent to find very diverse applications sharing the same port (e.g., web-based applications) or using non well-known ports to avoid being detected (e.g., P2P applications). Particularly, we can find behind the HTTP and HTTPS ports a wide variety of applications ranging from some services that are very sensitive to delays (e.g., VoIP) to other services whose performance relies on the average bandwidth (e.g., cloud storage). This reflects the necessity of a more comprehensive level of classification where the system can provide information about the specific applications generating the traffic (e.g., Netflix).

Regarding the latest trends in the research area of traffic classification, two main lines can be mainly remarked. On the one hand, techniques based on Deep Packet Inspection (DPI) analyze the packets' payload to identify the applications in the traffic. Typically, they can perform an exhaustive classification (e.g., at the application layer) achieving high accuracy levels. Nevertheless, performing DPI over every packet in the traffic is resource consuming and not feasible in all the network environments. Alternatively, other solutions based on Machine Learning (ML) where proposed to alleviate the burden of classification systems. These techniques are able to achieve similar accuracy to DPI tools when classifying the traffic by application-level protocols (e.g., SMTP, DNS). However, they cannot accurately identify the names of the applications (e.g., Gmail, YouTube) that generate traffic over the same application protocol (e.g., HTTP). In these particular cases, DPI techniques typically far outperform ML-based classifiers.

In light of the above, we present a scalable monitoring solution with OpenFlow that implements flow sampling and performs flow-level traffic classification with special emphasis on the identification of web and encrypted traffic. As in NetFlow/IPFIX, for each flow sampled, we maintain a flow entry in the switch that records the duration, and the amount of packets and bytes. Moreover, our system efficiently combines some state-of-the-art traffic classification techniques used in legacy networks to label every flow with the application that generated it. In more detail, our monitoring system has the following novel features:

Scalable: We address the scalabity issue in two different dimensions: (*i*) to alleviate the overhead for the controller, and (*ii*) to reduce the number of entries required in the flow tables of the switches. To this end, we designed two sampling methods which depend on the OpenFlow features available in the switches of the network. We implement flow sampling because it is easier to provide without requiring modifications to the OpenFlow specification, although we also plan to provide a packet sampling implementation in a future work. Note that our methods only require to initially install some rules in switches, which will operate autonomously to discriminate randomly the traffic to be sampled. To the best of our knowledge, there are no solutions in line with this approach. For example, iSTAMP [116] proposes to sample traffic based on an algorithm that "stamps" the most informative flows. However, this solution specifically addresses the detection of particular flows like *heavy hitters*, while our solution provides a generic report of the flows in the network.

Fully compliant with OpenFlow: Our monitoring system implements flow sampling using only native features present since OpenFlow v1.1.0. This makes our proposal more pragmatic and realistic for current SDN deployments, which strongly rely on OpenFlow. Furthermore, for backwards compatibility, we also propose a less effective monitoring scheme that is compatible with OpenFlow v1.0.0, further increasing the targets that can benefit from this solution. We could check that many SDN-enabled switches (e.g., HP, or NEC) do not implement NetFlow, so the proposed solution would be a good alternative for these

devices, since it provides reports with flow-level statistics as in NetFlow. Moreover, unlike NetFlow, OpenFlow enables to independently monitor specific slices of the network, which can be highly interesting in emerging SDN/NFV scenarios. We also found in the literature some monitoring proposals for SDN that rely on different protocols than OpenFlow. For instance, OpenSample [117] performs traffic sampling using sFlow, which is more commonly present than NetFlow in current off-the-shelf SDN switches. However, we consider that sFlow has a high resource consumption, as it sends every sampled packet to an external collector and maintains there the statistics. In contrast, the proposed system maintains the statistics directly in the switches. Alternatively, some authors propose architectures specifically designed for network monitoring tasks. For example, OpenSketch [107] enables to combine different sketches depending on the measurement tasks to perform. However, in favor of our proposal, some works like [118] highlight the importance of making an OpenFlow compatible monitoring solution, as it is cheaper to implement and does not require standardization by a larger community. Note that despite the advances in the OpenFlow standard, at the time of this writing the protocol does not provide direct support for flow sampling yet.

Transparent: The flow entries used by our system are independent from the entries installed by other modules performing different network functions (e.g., forwarding). To ensure this, we make use of the pipeline processing feature with multiple tables of OpenFlow. It follows a similar approach to Omniscient [119], where they propose to use separate rules to monitor specific flows that are already tagged by end-hosts and store their statistics in a separate OpenFlow table.

Asynchronous collection of flow statistics: Our system aggregates the flow-level statistics directly in the switch, and reports these statistics to SDN controllers when the flow entries expire (either by an idle or hard timeout). In FlowSense [120], they propose a similar mechanism to collect statistics for the entries in the switches and estimate the link utilization. However, in this solution the statistics of flows with large timeouts are typically collected too late. This makes it an impractical solution to obtain a timely view of the traffic in the network. In our solution, since the measurement module is completely decoupled from the others, we have the control to define the most adequate timeouts to obtain accurate and timely measurements. Our solution also support the use of efficient algorithms to conveniently select the timeouts, such as those proposed in PayLess [121] or OpenNetMon [118], where they design adaptive schedule algorithms to collect the statistics more efficiently.

Traffic classification: Our system performs flow-level classification combining some state-of-the-art techniques. In particular, we apply specific DPI and ML techniques to different types of traffic in order to achieve a good tradeoff between accuracy and deployment cost. Similarly to [122] and [123], we use information within the HTTP headers and certificates of SSL/TLS connections to unveil the applications generating respectively web

and encrypted traffic. Moreover, we process the DNS traffic as a supplementary source of information to discover the domain names associated to different flows. This, in turn, can help identify the application names. For the different classification techniques we evaluate, we individually measured the accuracy against a ground truth generated from real-world traffic and estimate the cost to deploy them in OpenFlow-based SDN environments.

The remainder of this chapter is structured as follows: in Section 4.2, we provide an OpenFlow overview focusing on the features and messages involved in the proposed solution. Section 4.3 shows the architecture of the proposed flow monitoring system. Section 4.4 describes the flow measurement system within this architecture. Section 4.5 describes the traffic classification system. In Section 4.6, we implement the system within the OpenDaylight [39] SDN controller and evaluate the accuracy and cost in a testbed with Open vSwitch, also combining different traffic classification techniques. Section 4.7 summarizes the related work. Lastly, Section 4.8 concludes this chapter and outlines some aspects for future work.

4.2. OpenFlow Background

Nowadays, there is a growing trend among vendors to adopt OpenFlow in their data-plane devices. Some of them opt for OpenFlow-only devices, while others offer hybrid solutions, where both traditional network protocols and OpenFlow coexist. At the time of this writing, it is quite unusual to find commodity switches with higher support than OpenFlow v1.3.0.

In this section, we particularly focus on the OpenFlow v1.1.0 specification, since it is the first version fully compatible with our solution. This is because from this version it is possible to make use of multiple tables, which enable us to decouple our monitoring system from others. Nevertheless, we propose an alternative solution with some limitations for switches with OpenFlow v1.0.0 support (more details are explained in Section 4.4.2). It is also worth mentioning that everything described for our solution can be applied to IPv6 traffic from OpenFlow v1.2.0 onwards, since previous versions have only support for IPv4.

Regarding the monitoring solution proposed in this chapter, we provide below a summary of the main elements and messages involved.

4.2.1. Multiple Flow Tables and Groups

Multiple flow tables and groups are both available from OpenFlow v1.1.0. The support of multiple tables enables to decouple the sets of entries of modules with different network functions operating in different tables.

Packets begin their processing pipeline in the first table of the device and can be directed to other tables. In this way, as it goes through the pipeline, a packet can both execute an action and continue the processing in the next table or accumulate the actions

and apply them at the end of the pipeline. In order to resolve possible conflicts between overlapping rules in the same flow table, each entry has a priority field.

Groups are abstractions that allow to represent a set of actions for all packets matching an entry in a flow table. Each group table contains a number of buckets which, in turn, are composed of a set of actions. Thus, if a bucket is selected, all its actions are applied to the packet. OpenFlow devises four different mechanisms to select the bucket to be applied to packets reaching the group table: (*i*) All (e.g., for multicast), (*ii*) Select (e.g., for multipath), (*iii*) Indirect, and (*iv*) Fast Failover (e.g., to use first live port). Our system leverages the *select* mechanism to implement one of the flow sampling methods described in Section 4.4.1. In a group of type *select*, packets are processed by a single bucket and, thereby, only actions within the selected bucket are applied. This bucket selection depends on the algorithm implemented in the switch. According to the OpenFlow specification, this algorithm should perform equal or weighted load sharing among buckets.

4.2.2. Adding New Flow Entries and Groups

When a packet matches a flow entry whose action is "*output to controller*", a portion of this packet is encapsulated in an OFPT_PACKET_IN message and forwarded to a SDN controller. Once the packet has been processed, the controller may send an OFPT_FLOW_MOD message to the switch to install a new flow entry with a set of instructions to be applied to the following packets of the flow. This is how OpenFlow enables to install reactively new flow entries. When adding a new flow entry, it is possible to set two timeouts (idle and hard) to define when the flow will be removed from the switch. The idle timeout defines the maximum time interval between two consecutive packets matching the entry, while the hard timeout is the maximum lifetime since the entry was installed.

In order to add a new group, the controller may send an OFPT_GROUP_MOD message to the switch. This message defines the group type (all, select, indirect or fast failover), a set of buckets with their corresponding actions, and a unique identifier (32 bits) for this group. Note that a group table does not contain match fields, but only actions within buckets that can be applied to the packets directed to the group. In order to forward packets to a group table, it is necessary to add an entry in a flow table defining an action of type OFPAT_GROUP, and this action must include the unique identifier of the group.

4.2.3. Statistics Retrieval Methods

To retrieve flow measurements, two different approaches deserve to be highlighted. First, pull-based mechanisms consist of making active measurements. To this end, SDN controllers may send queries (OFPT_MULTIPART_REQUEST message) to switches to collect the statistics of some flows. Then, the switch responds with an OFPT_MULTIPART_REPLY message including a summary of the flow (duration in seconds and nanoseconds, packet

count, and byte count). Alternatively, push-based mechanisms consist of collecting measurements asynchronously. In this case, when adding a new flow entry, the idle and/or hard timeouts are defined. Then, when a flow entry expires, the switch sends to the controller an OFPT_FLOW_REMOVED message with the flow statistics. This message also informs with flags that indicate if the expiration was caused by either the idle or the hard timeout. To receive asynchronously this message, the controller has to explicitly mark the OFPFF_SEND_FLOW_REM flag in the OFPT_FLOW_MOD message when adding a new flow.

4.3. Architecture of the Proposed Monitoring System

Figure 4.1 illustrates the architecture of our flow monitoring system, which is implemented in the control plane (i.e., in SDN controllers). We divided it in two different logical subsystems: (*i*) the *Flow measurement system*, and (*ii*) the *Flow classification system*. The former one is in charge of maintaining accurate and timely traffic statistics at a flow-level granularity. Particularly, flow measurements are maintained in switches and, when a flow entry in a switch expires, the statistics of the corresponding flow (packets and bytes counts, and duration) are reported to the control plane. Note that we identify the flows by its 5-tuple. At the same time, the flow classification system manages to identify the applications that generated each flow in the traffic. To this end, this system combines different DPI and ML techniques for traffic classification and eventually provides reports with application labels associated to each flow. Thus, our monitoring system combines the outputs of these two subsystems to finally provide a report that includes, for each flow in the traffic, its associated measurements and a label identifying the application that generated it.

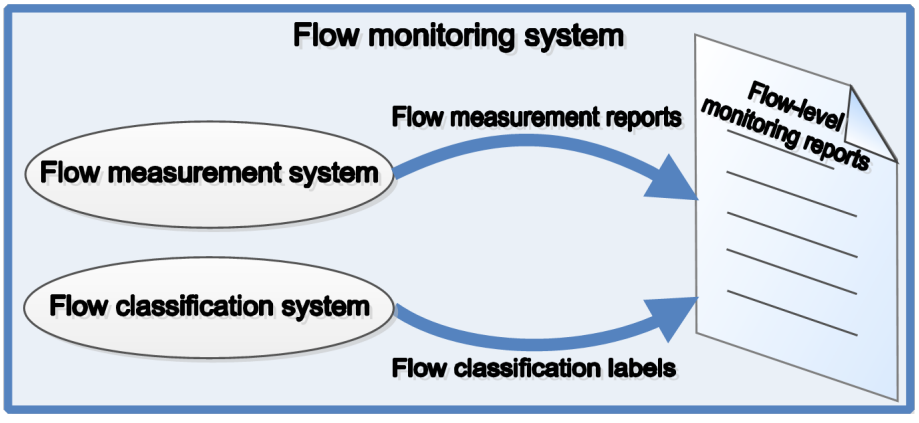


Figure 4.1: Architecture of our flow monitoring system.

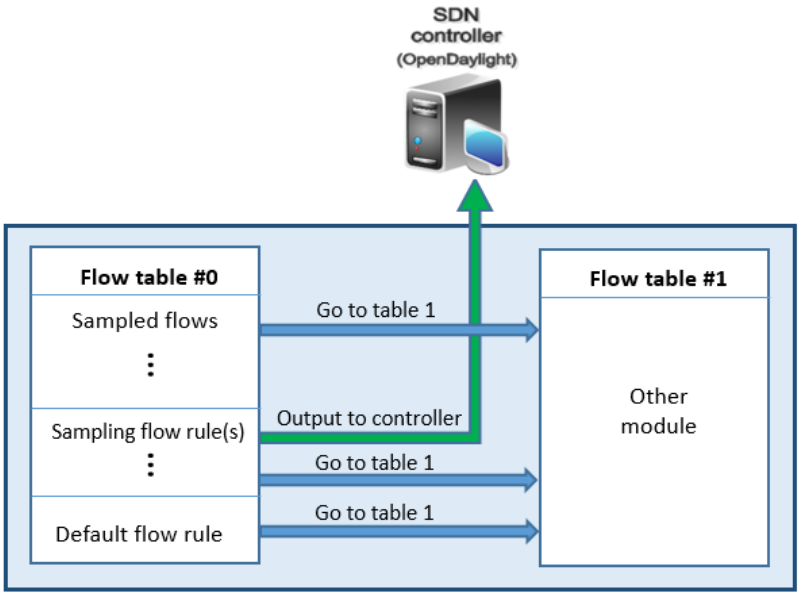
The following two sections of this chapter provide a detailed description of the flow measurement system (Section 4.4) and the flow classification system (Section 4.5) that compose this monitoring system.

4.4. Flow Measurement System

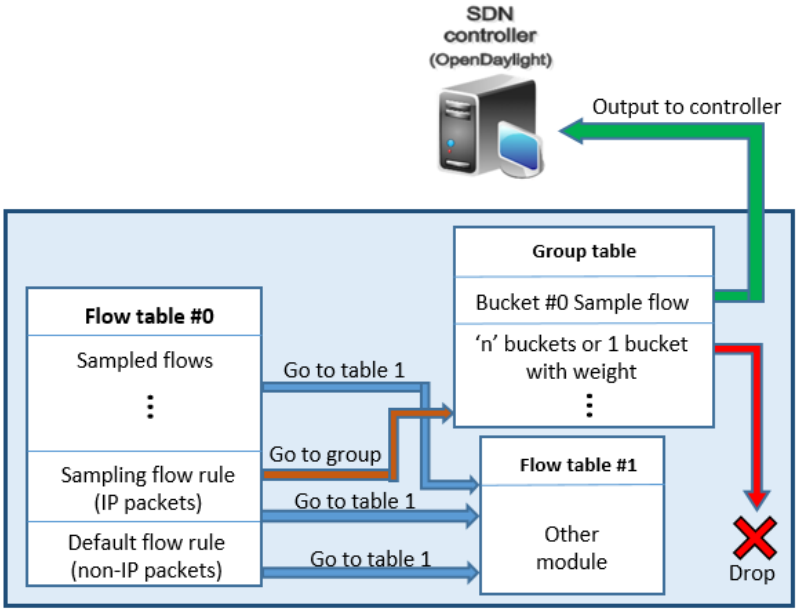
The proposed flow measurement system relies completely on the OpenFlow specification to obtain flow-level measurements similar to those of NetFlow/IPFIX in legacy networks. This is not new in SDN, since some works, such as [124], followed a similar approach earlier. However, to the best of our knowledge, no previous works proposed OpenFlow-based measurement methods that implement traffic sampling and provide reports in a NetFlow/IPFIX style. Particularly, our system samples randomly the traffic and maintains per-flow statistics in separated records, which are eventually reported to a collector (i.e., a SDN controller). Since we are aware that OpenFlow has many features that are classified as "*optional*" in the specification, we designed two different sampling methods with different levels of requirements of features available in the switch. These methods, in summary, consist of installing a set of entries in the switch that enable to discriminate the traffic to be sampled. Thus, we only send to the controller the first few packets of those flows to be monitored, and the controller is in charge of installing reactively specific flow entries to maintain the measurements for the following packets of the flow.

Our measurement system makes use of multiple flow tables linked, which is supported from OpenFlow v1.1.0. Nevertheless, we propose an alternative solution with some limitations for switches with OpenFlow v1.0.0 support (see more details in Section 4.4.2). The support of multiple tables permits to decouple the flow entries installed by different modules performing various network tasks in the same SDN controller.

Before going into the details of our sampling methods, we first describe the generic structure of the OpenFlow tables used in our system, which is illustrated in Figure 4.2a. In the two proposed sampling methods, the monitoring system operates in the first table of the switch, where the pipeline process starts. In this way, our system installs in this table some entries to sample the traffic and maintain records of the monitored flows. All the entries in the first table have at least one instruction to direct the packets to another table, where other modules can install entries with different purposes (e.g., traffic forwarding). Focusing on the table where our system operates, three different blocks of entries can be differentiated by their priority field. There is a first block of flow-level entries – identified by the 5-tuple – that maintain individual records of the monitored flows. Then, a block of entries with lower priority defines the packets to be sampled. And lastly, we add a default entry with the lowest priority which directs to the next table the packets that did not match any previous entries. The key point of this system lies in the second block of entries, where the sampling methods (described below) install rules to define which flows are monitored. The operation mode when a new packet arrives to the switch is to check first if it is already in one of the flow monitoring entries of the first block. If it matches any of these entries, the packets and bytes counters are updated, and the packet is directed to the next table. Otherwise, it goes through the second block of entries, which defines whether it should be sampled or not. If it matches one of the rules in this latter block, the packet is forwarded



(a) Sampling based on IP suffixes



(b) Sampling based on hash function

Figure 4.2: Scheme of OpenFlow tables of the proposed flow measurement system.

to the next table and to the controller – with an OpenFlow *Packet In* message. Then, the controller adds a specific entry in the first block of the table to maintain measurements for the following packets of the flow. Lastly, if the packet does not match any of the previous entries, it is simply directed to the next table.

4.4.1. Proposed Sampling Methods

This section presents the two sampling methods devised for our measurement system and discuss the OpenFlow features required for each of them. One is based on hash functions, and performs flow sampling very accurately. The other one – based on IP suffixes – is proposed as a fallback mechanism when it is not possible to implement the previous one. Our sampling mechanisms match the definition of the Packet Sampling (PSAMP) Protocol Specification [125], which is compatible with the IPFIX protocol specification. According to the PSAMP terminology, the first method matches the definition of hash-based filtering. While the second method can be classified as property match filtering, where a packet is selected if specific fields within the packet headers are equal to a predefined set of values. These solutions assume that switches have support for OpenFlow v1.1.0 onwards, since they need to use multiple flow tables in switches. However, in Section 4.4.2, we make some comments about how to implement an alternative solution with OpenFlow v1.0.0.

Sampling based on IP suffixes

This method performs traffic sampling based on IP addresses. For this purpose, the controller adds proactively one entry with match fields for particular IP address ranges. Traditionally, IP addresses are aggregated by prefixes (e.g., prefix-based routing). In contrast, we consider to apply a mask that checks the last n bits of the IPs (i.e., the IP suffixes). In this way, we sample a more representative set of flows, since we monitor traffic from different subnets (i.e., different IP prefixes). To implement this, it is only necessary to use wildcarded entries that filter the IP suffixes of specific groups of source and/or destination IP addresses. In order to control the number of flows sampled, we make a rough consideration that, in average, flows are homogeneously distributed along the whole IP range (we later analyze this assumption with real traffic in Section 4.6.1). As a consequence, for each bit fixed in the mask, the number of flows sampled will be divided by two with respect to the total number of flows arriving to the switch. We acknowledge that typically there are some IPs that generate much more traffic than others, but this method somehow permits to control the number of flows monitored. Furthermore, if we consider source-destination IP pairs for the selection, instead of individual (source or destination) IP suffixes, we can control better this effect. In this case, if we sample an IP address of a host that generates a large number of flows, only those flows that match both, the source and destination IP suffixes, are sampled. Thus, our sampling rate can be defined by the following expression:

$$sampling\,rate = \frac{1}{2^m \cdot 2^n} \tag{4.4.1}$$

Where "m" is the number of bits checked for the source IP suffix and "n" the number of bits checked for the destination IP suffix.

This method is similar to host-based (or host-pair-based) sampling, as we are using IP addresses to select the packets sampled. However, host-based schemes typically provide statistics of the traffic sent and received aggregated on each host. In contrast, we sample the traffic by IP suffixes, but provide individual statistics at a flow-level granularity. Note that, to avoid bias in the sampling process, the IP suffixes can be periodically modified by simply replacing the sampling rule(s) in the OpenFlow table.

To implement this method, the only optional requirement of OpenFlow is the support of arbitrary masks to check IP suffixes. Note that there are some devices that only support prefix-based IP masks. Additionally, we have presented and evaluated an alternative method based on matching on port numbers for those switches that do not support suffix-based IP masks. However, this method requires a larger number of entries to sample the traffic. The results of this evaluation are reported in [126].

Hash-based flow sampling

This method consists of computing a hash function on the traditional 5-tuple fields of the packet header and selecting it if the hash value falls in a particular range. To implement this method, we make use of the group tables feature of OpenFlow. In OpenFlow, a group table contains a number of buckets which, in turn, are composed by a set of actions. Thus, when a bucket is selected, all its actions are applied to the packet. For the implementation of this method, we leverage the use of the *select* mechanism to balance the load between different buckets within a group. The implementation of the bucket selection algorithm is external to the OpenFlow specification. It depends on the load balancing algorithm implemented in the switch, which should perform equal or weighted load sharing among the buckets installed. In Figure 4.2b, we can see the table structure designed for this method. In this case, all the IP packets are directed to the next table as well as to a group table. In the group table, only one bucket is configured to send the packet to the controller, and eventually monitor the following packets of the flow. The remaining buckets in the group table drop the packet. Thus, it is possible to control the sampling rate by selecting a weight for each of these buckets. This method controls much better the sampling rate, as we can assume that a hash function is homogeneous along all its range for all the 5-tuples seen in the switch. In contrast to the previous method (based on IP suffixes), this approach accurately follows the definition of flow sampling (i.e., sample the packets of a set of flows with some probability).

This method requires the use of group tables with *select* buckets as well as having an accurate algorithm in the switch to balance the load among buckets.

4.4.2. Modularization of the System

The proposed measurement system leverages the support of multiple tables to isolate its operation from other modules performing other network functions. Thus, we can see our monitoring system as an independent module in the controller that does not interfere with other

modules operating in other tables. In the controller we can filter and process the *Packet In* messages triggered by entries of our module, since these messages contain the table ID of the entry which forwarded the packet to the controller. Note that, in order to process packets of the monitoring system before other modules, it is typically necessary to select properly the priority of the *Packet In* messages received by the different modules running in the network. Additionally, our system can be deployed in a network using a hypervisor (e.g., CoVisor [127]) to run network modules in a distributed manner in different controllers.

Note that, for those switches that have only support for OpenFlow v1.0.0, it is feasible to implement the sampling method based on IP suffixes. Thus, the flow entries installed by the monitoring system can be combined with all the entries of other modules in the same table. However, this would imply loosing the isolation of our monitoring system from the remaining modules.

4.4.3. Statistics Retrieval

Our system implements a push-based approach to retrieve the flow statistics. It means that, when a new flow entry is added, it includes some predefined idle and hard timeouts. Then, when the entry expires, the switch sends to the controller an OFPT_FLOW_REMOVED message including the flow statistics. Given that our system installs flow entries independently from the other networking modules, it can conveniently select the timeouts. Thus, we overcome the issue of other push-based solutions like FlowSense [120], which reports the statistics of flows with large timeouts after too long a time.

4.5. Flow Classification System

The proposed classification system produces labels that identify the applications generating the different flows that were sampled by the measurement system (Sec. 4.4). Particularly, it operates in the the control plane (i.e., in SDN controllers), and combines efficiently different classification strategies already used in legacy networks. However, we adapt these techniques to implement them efficiently in SDN-based environments. Our system classifies the traffic at two different levels of detail: (*i*) it classifies all the flows at the level of application protocols (e.g., RTP, DHCP, SSH), and (*ii*) applies specific DPI techniques to discover the names of the applications generating web and encrypted traffic (e.g., Netflix, Facebook, YouTube).

In OpenFlow-based networks, when installing flows reactively, packets belonging to the same flow are sent to the controller until a specific entry is installed for them in the switch. As a consequence, the SDN controller can receive more than one packet for each flow to be monitored. In particular, this occurs during the time interval between the arrival of the first packet of a flow at the switch, and the time when the corresponding flow entry is installed in the switch. This time interval is mainly the result of the following factors: (*i*)

the time needed by the switch to process an incoming packet of a *new* sampled flow and forward it to the controller, (*ii*) the Round-Trip Time (RTT) between the switch and the controller, (*iii*) the time in the controller to process the *Packet In* message and send an order to the switch to install a new flow entry, and (*iv*) the time in the switch to install the flow entry. The first and fourth factors depend on the processing power and the workload of the switch. The RTT depends on some aspects like the distance between the switch and the controller, or the utilization of the link that connects them. The second factor depends on the processing power and the workload of the controller and, of course, its availability. Thus, our classification system leverages this issue by applying DPI only to those packets received in SDN controllers, which often contain meaningful information to classify the traffic. This permits to achieve accurate traffic classification without producing large additional overheads.

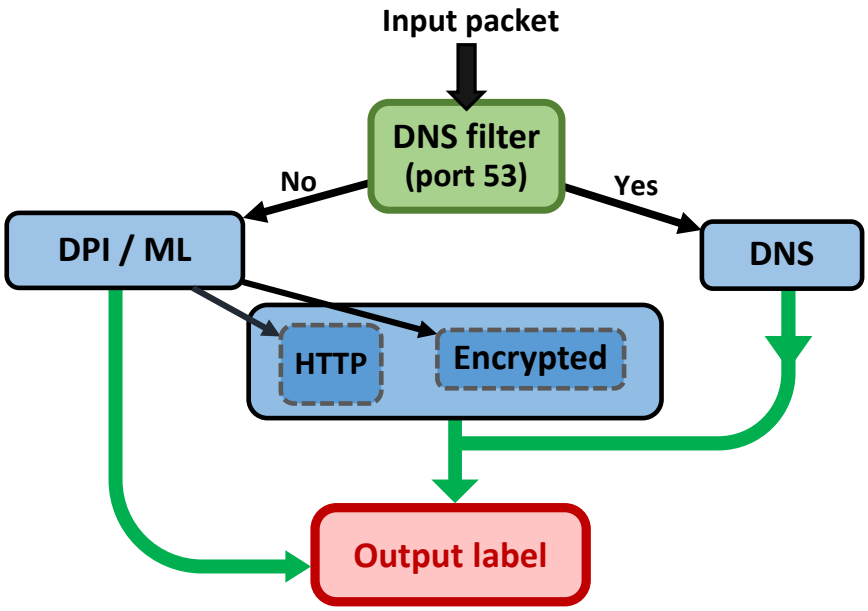


Figure 4.3: Scheme of the proposed flow classification system.

Figure 4.3 shows the operational scheme of our classification system when a new packet arrives to the SDN controller. First, it discerns whether the packet contains traffic from the Domain Name System (DNS) protocol or not. We consider that the traffic is DNS if the packet header matches port 53. In this case, the packet is forwarded to our "DNS module" to extract some data from the records in DNS queries. This enables to then apply the method described in [123], which associates domain names to the server IPs of web and encrypted flows. Otherwise, if the packet is considered as non-DNS traffic, it is processed by our "DPI/ML module", which executes either a DPI tool or a supervised ML model. This module is intended to provide a label that classifies the flow at the level of the application protocol (e.g., SMTP). Following the scheme in Figure 4.3, if the "DPI/ML module" labels the flow as HTTP or encrypted traffic, the packet is accordingly directed to our "HTTP"

or "Encrypted" modules. In these modules, we apply the DPI techniques proposed in [123] and [122] to extract the hostnames associated to flows either from the HTTP headers or the SSL/TLS certificates. If the "HTTP" or "Encrypted" modules manage to obtain the hostname, the system outputs the classification labels produced by these modules. Otherwise, we use the information collected in the "DNS module" to infer a domain name associated to the server IP address of the flow.

We provide below a more detailed description of the the four classification modules that compose our system (Fig. 4.3):

DPI/ML module: The aim of this module is to classify the monitored flows with labels that identify their associated application-level protocols (e.g., BGP, SNMP). To this end, we implemented within this module two alternative solutions to classify the traffic: one based on a commercial DPI tool, and another based on supervised ML (c5.0 decision tree). We then evaluate these two classifiers (in Section 4.6.2) using real-world traffic to show the tradeoffs between accuracy and resource consumption when applying each of them. Note that, except for those flows that are classified as HTTP or encrypted traffic, the output label of our classification system is the one generated by this module.

DNS module: In this module, we implemented the proposal described in [123], where they use information in DNS queries to then associate domain names to the HTTP(S) flows in the traffic. For this purpose, they rely on the basic assumption that, prior to execute an HTTP(S) request, the client application typically resolves the IP address of the server by sending a DNS query. Hence, monitoring the DNS traffic enables to discover the domain names associated to the server IP addresses of the HTTP(S) flows. Note that, as they highlight in [122], the domain name information is very meaningful, as it often permits to unveil the name of the application that generated a specific flow. In order to implement this technique, this module should receive the DNS traffic traversing the monitored switches and maintain the information extracted from DNS records until their expiration time. To this end, we proactively add a flow entry in the switches that redirects to the controller all the traffic matching port 53. We then evaluate (in Section 4.6.2) the cost of deploying this technique in a SDN controller and show that the amount of traffic processed by this module is quite reduced in our evaluation scenario using real-world traffic.

HTTP module: In this module, we implement the technique they propose in [123] to extract the hostname from the *host* field in the HTTP headers. In this way, inspecting the first few packets of an HTTP connection is typically sufficient to find the hostname associated to the HTTP flow. This, in turn, provides useful information to discover the applications generating different flows of HTTP traffic. Note that this module only processes the packets that are classified by the "DPI/ML module" as HTTP traffic.

Encrypted module: In this module we implement the technique proposed in [122], where they perform DPI to extract the Server Name Indication (SNI) fields of the SSL/TLS certificates exchanged during the handshake prior to establish an encrypted connection.

Similarly to the previous module, this information is typically present in the first few packets of flows and is useful to unveil the applications generating encrypted traffic. Note that this module only processes the packets classified as encrypted traffic by the "DPI/ML module".

We provide in Section 4.6.2 a description of the tools and features involved in the implementation of each of these modules within the SDN controller.

In a nutshell, our approach is to combine different classification techniques and apply them only to specific traffic types regarding the tradeoff between accuracy and performance. This allows us to achieve high levels of accuracy considerably saving the processing power needed in the SDN controller to perform such a comprehensive classification. Remark that our system provides a deep insight of the traffic, as it not only classifies the traffic by application protocols, but also provides the hostname information to discover the applications that generate web and encrypted traffic.

4.6. Experimental Evaluation

This section includes an evaluation of the two subsystems that compose our Flow Monitoring System (Sec. 4.3). We first evaluate the Flow Measurement System (Sec. 4.4). Then, we evaluate the Flow Classification System (Sec. 4.5) with special focus on measuring the classification accuracy and the cost to deploy the system.

4.6.1. Evaluation of the Flow Measurement System

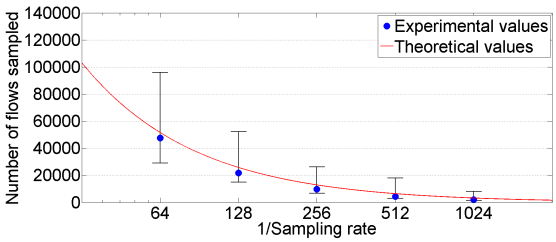
For the evaluation of our monitoring system, we implemented it in a module within the OpenDaylight SDN controller [39]. We conducted experiments in a small testbed with (*i*) a virtual switch (Open vSwitch), (*ii*) a host that injects traffic into the switch, and (*iii*) another host that acts as a sink for all the traffic injected. All the experiments make use of real-world traffic from three different network scenarios. We provide in Table 4.1 a description of the traces we used. These traces were filtered to keep only the TCP and UDP traffic.

Accuracy of the Sampling Methods

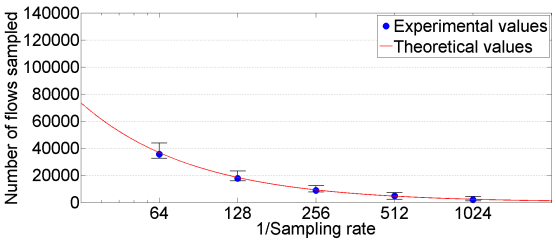
We conducted experiments to evaluate the accuracy of our flow measurement system. Thus, we first assess if the sampling rate is applied properly and if the selection of flows is random enough when using the proposed sampling methods. All our experiments were separately done for the MAWI, CAIDA and UNIVERSITY-1 traces described in Table 4.1 and repeated applying sampling rates of 1/64, 1/128, 1/256, 1/512, and 1/1024. For the method based on IP suffixes, we considered two different alternatives: matching only a source IP suffix, or matching both the source and destination IP suffixes. For each alternative, we execute 500 experiments selecting randomly IP suffixes, and repeat the evaluation for every traffic trace and every sampling rate.

Table 4.1: Summary of the real-world traffic traces used in the experiments.

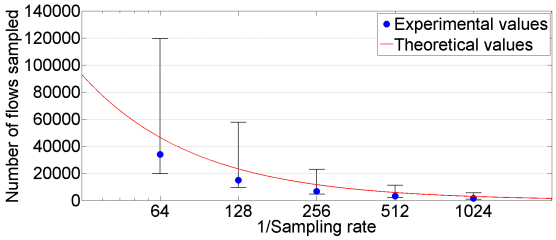
Trace dataset	**# of flows**	**# of packets**	**Description**
MAWI [128] 15th July 2016	3,299,166 (total flows) 2,653,150 (TCP flows) 646,016 (UDP flows)	54,270,059	1 Gbps transit link from the WIDE network to the network of the Internet Service Provider (ISP). Trace from the samplepoint-F. Average traffic rate: 507 Mbps
CAIDA [129] 18th February 2016	2,353,413 (total flows) 1,992,983 (TCP flows) 360,430 (UDP flows)	51,368,574	10 Gbps backbone link of a Tier1 ISP (direction A - from Seattle to Chicago). Average traffic rate: 2.9 Gbps
UNIVERSITY-1 25th November 2016	2,972,880 (total flows) 2,349,677 (TCP flows) 623,203 (UDP flows)	75,585,871	10 Gbps access link of a large Spanish university that connects about 25 schools and 40 departments (geographically distributed in 10 campuses) to the Internet through the Spanish Research and Education network (RedIRIS). Average traffic rate: 2.41 Gbps
UNIVERSITY-2 17th March 2017	4,679,374 (total flows) 3,712,431 (TCP flows) 966,943 (UDP flows)	298,860,479	This trace was captured from the same vantage point than the trace labeled as "UNIVERSITY-1". Average traffic rate: 3.17 Gbps



(a) MAWI trace



(b) CAIDA trace



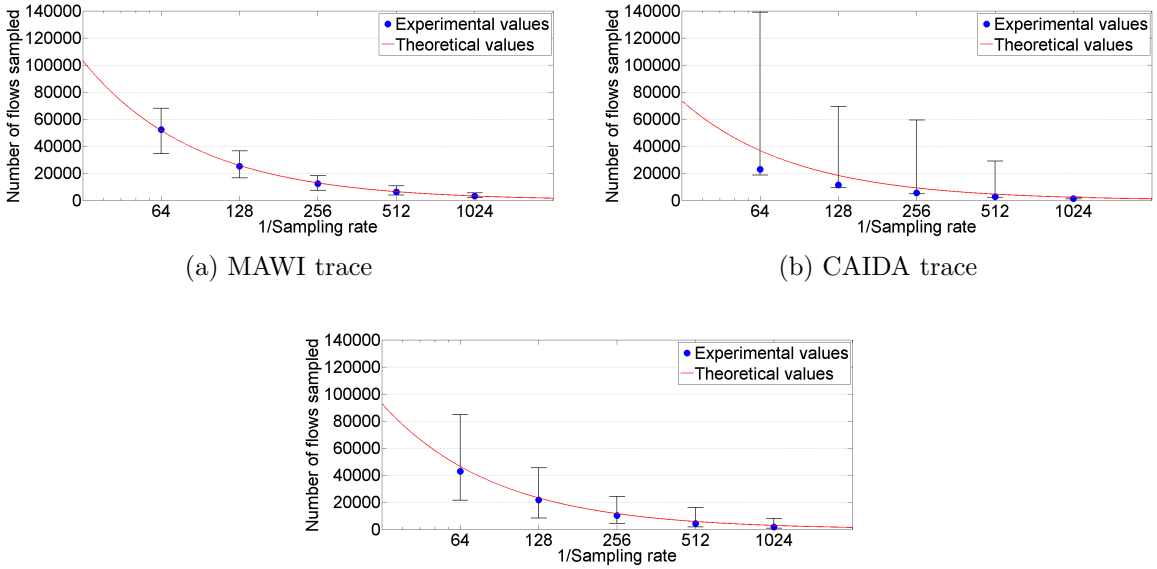
(c) UNIVERSITY-1 trace

Figure 4.4: Evaluation of the sampling rate for methods based on source IP suffixes.

To analyze the accuracy in the application of the sampling rate, we evaluate the number of flows sampled by our methods and compare it with the theoretical number of flows if we used a perfectly random selection function. We calculated these theoretical numbers by simply multiplying the total number of flows in the traces by the sampling rate applied. Figure 4.4 shows the results for the method based only on source IP suffixes for the three traces. Each point in these plots displays the median value of the number of flows sampled in the 500 experiments executed. Likewise, the bars show the interval between the 5^{th} and the 95^{th} percentiles along the 500 experiments. The x-axis represents the sampling rate applied. Additionally, in Figure 4.5, we show the same results for the case that considers pairs of source and destination IP suffixes. Given these results, we can see that the median values obtained are quite close to the theoretical values (i.e., in the average case these methods apply properly the sampling rate established). However, we can observe a high variability among experiments. This means that, depending on the IP suffixes selected, we can over- or under-sample. Regarding Equation 4.4.1, we show in Table 4.2 the number of bits we checked for the source and destination IP suffixes to define the desired sampling rates in the different experiments we made.

Table 4.2: Number of bits checked for the source and destination IPs.

IP-based sampling methods		**Number of bits checked**				
		SR = 1/64	**SR = 1/128**	**SR = 1/256**	**SR = 1/512**	**SR = 1/1024**
Based on src IP suffixes	bits src IP suffix (m)	6	7	8	9	10
Based on src-dst IP pairs	bits src IP suffix (m)	3	4	4	5	5
	bits dst IP suffix (n)	3	3	4	4	5



(a) MAWI trace

(b) CAIDA trace

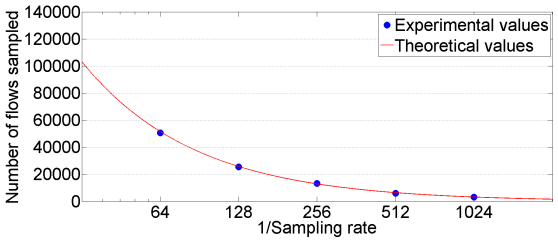
(c) UNIVERSITY-1 trace

Figure 4.5: Evaluation of the sampling rate for methods based on pairs of IP suffixes.

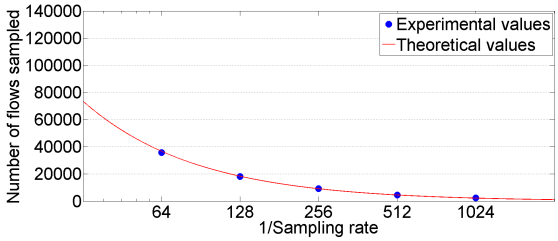
Next, we evaluate the hash-based sampling method making use of the load balancing algorithm for group tables included in Open vSwitch. In line with the scheme in Figure 4.2b, we installed in the switch a group table with two buckets of type *select* (with actions "*output to controller*" and "*drop*" respectively) and defined their weights to send to the controller the desired amount of flows according to the sampling rate applied. The results in Figure 4.6 show that this method can better control the sampling rate with respect to the previous one. Not only it samples a number of flows very close to the theoretical one, but also it does not experience significant variability among experiments, as it is based on a deterministic selection function. Furthermore, it achieves good results for the three traces, which demonstrates that it is a robust and generalizable method for different network scenarios.

In order to evaluate the randomness in the selection of our sampling methods, we compare our results with those obtained with a perfect implementation of flow sampling that selects completely random the sampled flows. Thus, if our implementation is close to the perfect flow sampling method, the Flow Size Distribution (FSD) should remain unchanged after applying the sampling, i.e., the distribution of the flow sizes (in number of packets) must be very similar for the original and the sampled data sets. We acknowledge that this property is not completely preserved for the IP-based method, but we follow this approach to measure how random is the flow selection of this method and compare it with the hash-based method.

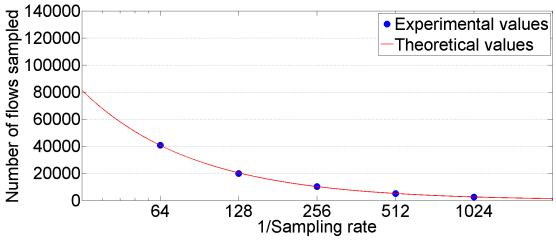
We quantify the randomness of the sampling method by calculating the difference between the FSDs of the original and the sampled traffic. For this purpose, we use the



(a) MAWI trace



(b) CAIDA trace

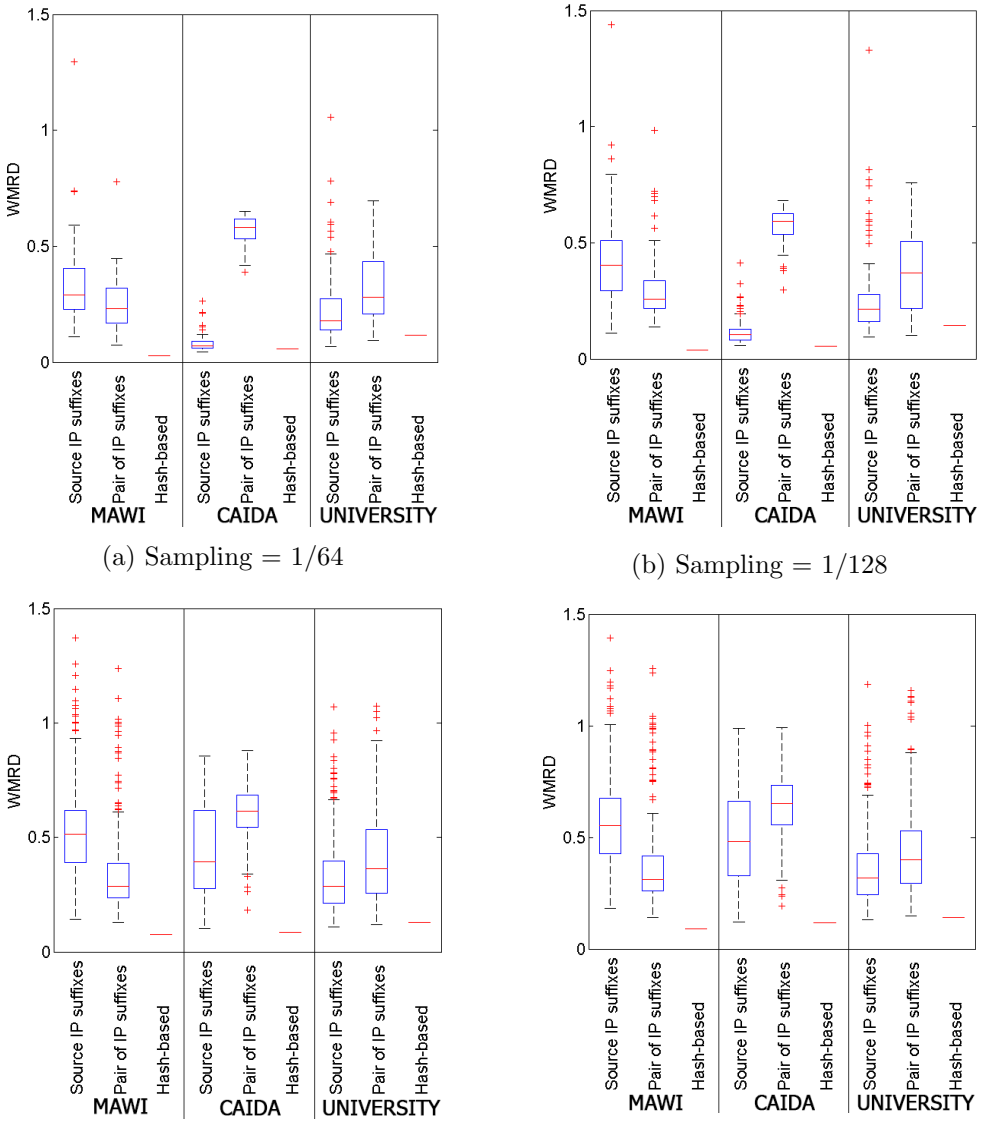


(c) UNIVERSITY-1 trace

Figure 4.6: Evaluation of sampling rate for the hash-based method.

Weighted Mean Relative Difference (WMRD) metric proposed in [130]. Thus, a small WMRD means that the flow selection is quite random. In Figure 4.7, we present boxplots with the results of our proposed methods for sampling rates of 1/64, 1/128, 1/512, and 1/1024. We can observe that these results are in line with the previous results about the accuracy controlling the sampling rate. The method that shows better results is the hash-based one. Regarding the methods based on IP suffixes, we observe that the method based on pairs of IP addresses achieves more random flow selection in the MAWI trace. While for the CAIDA and UNIVERSITY-1 traces, the method based on source IP suffixes achieves better results. Note that we chose the FSD to compare the randomness of the sampling methods because this function is known to be robust against flow sampling.

All the previous experiments were done using traffic traces captured in edge nodes of large networks, which aggregate great amounts of traffic from different subnets in the network. However, we find also interesting to evaluate how our sampling methods behave when they are deployed in switches that forward traffic from few subnets. To this end, we repeated the experiments with a filtered trace obtained from the UNIVERSITY-2 trace (in Table 4.1) that keeps only traffic belonging to a specific department in the university network. In particular, this trace contains traffic from three different IP prefixes with mask lengths of 24 bits. The resulting trace contains a total of 82,183 flows. In this case, we only test the methods based on IP suffixes. Since the total number of flows in the trace is quite low, we made only experiments applying sampling rates of 1/64, 1/128 and 1/256, as lower sampling rates result in an excessively small amount of flows sampled. Figures 4.8a and 4.8b show the results regarding the accuracy applying the sampling rate respectively



(a) Sampling = 1/64

(b) Sampling = 1/128

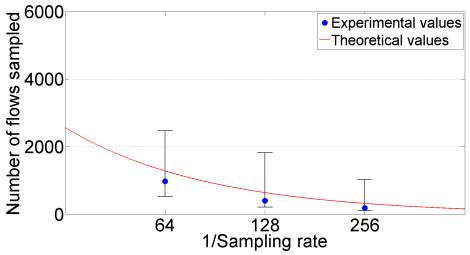
(c) Sampling = 1/512

(d) Sampling = 1/1024

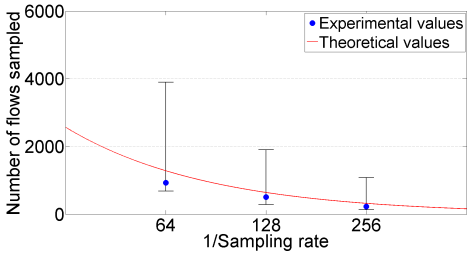
Figure 4.7: Weighted Mean Relative Difference (WMRD) between FSDs.

for the methods based on source IP suffixes and pairs of src-dst IP suffixes. Additionally, we show in Figure 4.8c the results regarding the randomness of these two methods applying a sampling rate of 1/128.

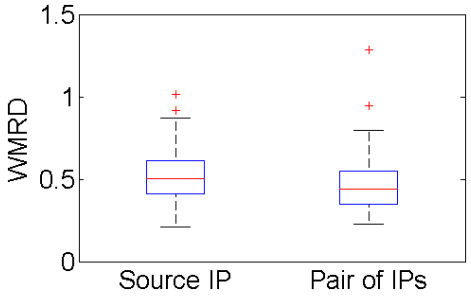
Note that our system permits to consistently sample the traffic along the whole network by installing rules in all the monitored switches using the same IP suffixes or, for the hash-based method, sampling traffic that matches the same range of hash values. This, enables to perform Trajectory Sampling [131], which is very useful to observe the trajectories of different flows traversing the network.



(a) Source IP suffixes



(b) Pair of IP suffixes



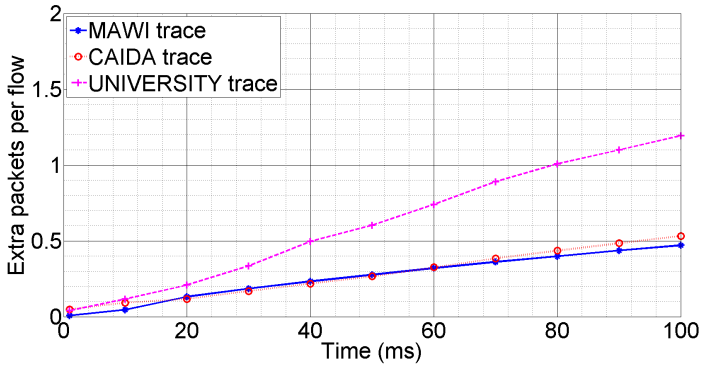
(c) WMRD with SR=1/128

Figure 4.8: Results with traffic from a specific department in trace UNIVERSITY-2.

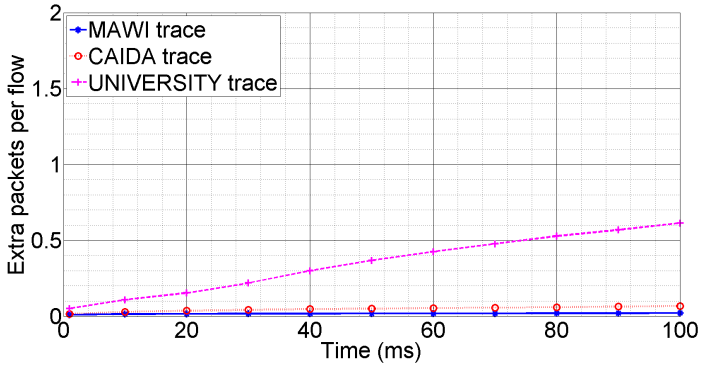
Overhead of the Flow Measurement System

An inherent problem in OpenFlow is that, when we install flows reactively, packets belonging to the same flow are sent to the controller until a specific entry for them is installed in the switch. As a consequence, in our system we can receive in the controller more than one packet for each sampled flow. Particularly, this occurs during the time interval between the reception of the first packet of a flow in the switch, and the time when a specific entry for this flow is installed in the switch. The factors that contribute to the duration of this time interval were already summarized in Section 4.5.

In order to analyze all the different bottlenecks in a single metric, we measure the number of packets that are sent to the controller for each sampled flow before the switch installs specific rules to maintain the measurements. This is the amount of additional packets of the same flow that our measurement system has to process. In the remainder of this section we refer to these additional packets as "extra packets". We consider a scenario with a range from 1 ms to 100 ms for the elapsed time to install a new flow entry. As a reference, in [40] they observe a median value of 34.1 ms for the time interval to add a new flow entry with the ONOS controller in an emulated network with 206 software switches and 416 links. Thus, we simulate this range of time intervals for the MAWI, CAIDA and UNIVERSITY-1 traces (in Table 4.1). We use the packet timestamps to calculate how many packets are within this interval and, thereby, would be sent to the controller. We analyze separately the overhead for TCP and UDP, as their results may differ due to their different traffic patterns. We show the results in Figure 4.9. As we can observe, the average



(a) TCP traffic



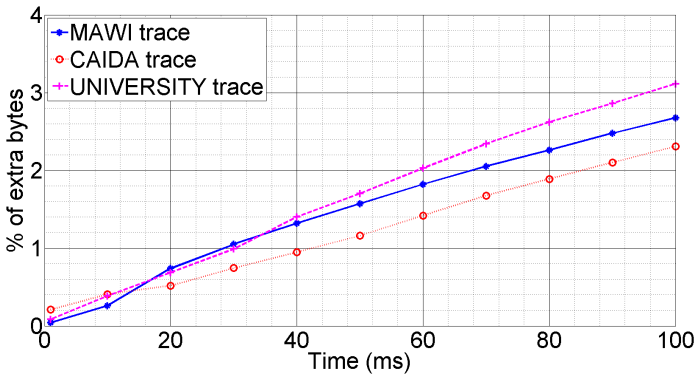
(b) UDP traffic

Figure 4.9: Average number of extra packets per flow.

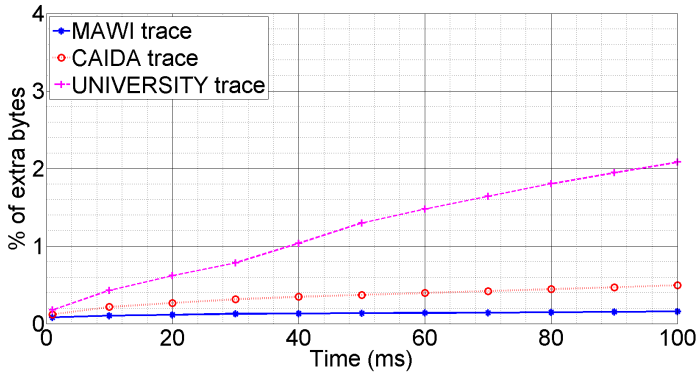
number of extra packets varies from less than 0.2 packets for delays below 20 ms, to roughly 1.2 packets for an elapsed time of 100 ms with TCP traffic.

Likewise, in Figure 4.10 we show the results in terms of average percentage of extra bytes sent to the controller. We observe that the percentage of extra bytes ranges from less than 0.8% for elapsed times below 20 ms to 3.1% in the worst case with an elapsed time of 100 ms and TCP traffic. These results show that the amount of extra traffic sent to the controller is significantly smaller than if we implemented the trivial approach of forwarding all the traffic to the controller or a NetFlow probe instead of maintaining the per-flow statistics in switches.

These results also reflect that for UDP traffic the number of extra packets and bytes per flow is significantly smaller than for TCP traffic. Among other reasons, this is due to the fact that typically many UDP flows are single-packet (e.g., DNS requests or responses). Note that these results are not applicable to any scenario, as we can find networks using protocols such as VXLAN, LISP, or QUIC that typically generate large flows over UDP. For these cases, the overhead contribution of the UDP flows would be closer to the case of TCP traffic. This is the case of the results with the UNIVERSITY-1 trace, where we could



(a) TCP traffic



(b) UDP traffic

Figure 4.10: Percentage of extra bytes.

observe that UDP flows had a larger average number of packets, as it is reflected in Figures 4.9b and 4.10b.

From these results, we can estimate the CPU cost of running our monitoring system in a SDN controller, as the processing cost per packet can be considered constant. In particular, the controller only needs to maintain a hash table to keep track of those packets that are sent to the controller and thus are not accounted for in switch (i.e., the extra packets shown in Fig. 4.9).

As for the memory overhead in the switch, we implement sampling methods that provide mechanisms to control the number of entries installed. With our solution it is necessary to maintain a flow entry for each individual sampled flow. Thus, there are three main factors that determine the amount of memory needed in the switch to maintain the statistics: (*i*) the flow rate per time unit (5-tuple flows), (*ii*) the sampling rate selected, and (*iii*) the idle and hard timeouts selected for the entries to be maintained. The first factor depends specifically on the nature of the network traffic, i.e., the rate of new flows arriving to the switch. It is a parameter fixed by the network environment where we operate. However, as in NetFlow, the sampling rate and the timeouts (idle and hard) are configurable parameters

that affect the memory requirements in the switch. Thus, Equation (4.6.1) permits to make a rough estimate of the average amount of concurrent flow entries maintained in the switch.

$$\begin{aligned} Avg.\,entries &= R_{flows/s} \cdot sampling\,rate \cdot E[t_{out}] \\ sampling\,rate &\in (0,1] \quad t_{out} \in [t_{idle}, t_{hard}] \end{aligned} \tag{4.6.1}$$

Where "$R_{flows/s}$" denotes the average rate of flows per time unit, "sampling rate" is the ratio of flows we expect to monitor, and $E[t_{out}]$ is the expected time that a flow entry will be maintained in the switch.

In order to configure a specific sampling rate, for the method based on IP suffixes we can set the number of bits to be checked for the IP suffix(es) according to (4.4.1). Likewise, for the hash-based method, we can set the portion of flows to be sampled by configuring the weights of the buckets. Regarding the timeouts, the controller can set the values of the idle and hard timeouts when adding a new flow entry in the switch to record the statistics (with the OFPT_FLOW_MOD message).

To conclude this section, we propose some different scenarios and estimate the average number of concurrent flow entries to be maintained in the switch. The purpose of this analysis is to have a picture of the approximate memory requirement of the proposed flow measurement system. To this end, we rely on (4.6.1). In our scenarios we consider the three different real-world traces MAWI, CAIDA and UNIVERSITY-1 described in Table 4.1. Thus, to calculate "$R_{flows/s}$" for each trace, we divide their respective total number of flows by their duration. Furthermore, we consider two different sampling rates: 1/128 and 1/1024. For the configuration of the timeouts, we envision a typical scenario using the default values defined in NetFlow: 15 seconds for the idle timeout, and 30 minutes for the hard timeout. Regarding the average time that a flow remains in the switch ($E[t_{out}]$), we know that it ranges from the idle timeout to the hard timeout. In this way, we consider these two extreme values and some others in the middle. The case with the lowest memory consumption will be when $E[t_{out}]$ is equal to the idle timeout, and the case with the highest consumption, when $E[t_{out}]$ is equal to the hard timeout. The amount of memory for each flow entry strongly depends on the OpenFlow version implemented in the switch. The total amount of memory of a flow entry is the sum of the memory of its match fields, its action fields and its counters. For example, in OpenFlow v1.0 there are only 12 different match fields (269 bits approximately), while in OpenFlow v1.3 there are 40 different match fields (1,261 bits).

Table 4.3 summarizes the results for all the cases described above. As a reference, in [132] they noted that modern OpenFlow switches have support for 64,000 to 512,000 flow entries. To these flow entries estimated, we must add the additional amount of memory of the implementation of the sampling methods described in Section 4.4.1. For both methods,

Table 4.3: Estimation of the average flow entries used in the switch.

Sampling rate	**Trace dataset**	$\mathbf{R_{flows/s}}$	**Avg. number of flow entries**						
			E[t]=15 s	**E[t]=60 s**	**E[t]=300 s**	**E[t]=600 s**	**E[t]=900 s**	**E[t]=1,200 s**	**E[t]=1,800 s**
1/128	UNIVERSITY-1	9,916	1,162	4,648	23,241	46,481	69,722	92,963	139,444
	MAWI	3,665	429	1,718	8,590	17,180	25,770	34,359	51,539
	CAIDA	21,672	2,540	10,159	50,794	101,588	152,381	203,175	304,763
1/1024	UNIVERSITY-1	9,916	145	581	2,905	5,810	8,715	11,620	17,430
	MAWI	3,665	54	215	1,074	2,147	3,221	4,295	6,442
	CAIDA	21,672	317	1,270	6,349	12,698	19,048	25,397	38,095

the switch must allocate an additional table to maintain the sampled flows as well as the entries that determine the flows to be sampled. For the method based on IPs, it uses an additional wildcarded flow entry that defines the IP suffix(es) to be sampled. For the hash-based method, it uses an additional entry to redirect the packets to a group table, as well as the group table with its respective buckets. We do not provide an estimate of this memory contribution since we consider it is too dependent on the OpenFlow implementation in the switch. Nevertheless, we assume that this amount of memory is negligible compared to the amount of memory allocated for the entries that record the statistics of the sampled flows.

Note that our system could be attacked by malicious agents sending malformed packets or messages at a high rate in order to cause congestion in the controller, as it would receive all the first few packets from sampled flows. Also the communication in the Southbound interface could be interrupted because the control channels were compromised. All these are inherent problems of OpenFlow-based networks, since the data plane typically relies too much on the decision making in the controller. In this case, it would be useful to count on a system able to detect and mitigate these attacks and guarantee a reliable connection between the control and data planes.

4.6.2. Evaluation of the Flow Classification System

We implemented our classification system combining different classification tools. For the 'DPI/ML module', we implemented the classifier based on DPI using a distribution of "nDPI" (version 1.8-stable). A list of the protocols supported by this tool is available at [133]. The implementation of our ML classifier was done using the well-known C5.0 decision tree [134], whose code is under the GNU GPL license. We made the ML feature selection based on the work in [56], where they use as inputs for the model some flow-level NetFlow features. In particular, we included the following features: (*i*) source and destination ports, (*ii*) the transport protocol (e.g., TCP, UDP), and (*iii*) the size of the first packets of the flow (with a maximum of 6 packets). As possible output classes, we use the application protocols included in the list of classification labels provided by nDPI [133]. To this end, we only selected from this list those labels whose prefix is "NDPI_PROTOCOL". In total, our ML model has 169 different classes. However, note that nDPI also provides additional labels for web-based traffic that identify the applications (e.g., Facebook) or the content type (e.g., AVI contents). In total, it supports 63 different labels of this type. We did not consider these labels in our ML system as we used more accurate DPI techniques to specifically unveil the applications generating web and encrypted traffic. For the implementation of the HTTP, Encrypted and DNS modules, we used respectively the HTTP, SSL and DNS scripts of the open-source tool Bro IDS, which permits to extract the hostname from HTTP headers and SSL/TLS certificates, and the domain names from DNS queries.

In this Section, we evaluate the different techniques implemented in our classification system in order to better understand the tradeoff between accuracy and performance when deploying them in SDN environments.

Ground truth

In order to evaluate our flow classification system, we created a ground truth using real-world traffic. Our dataset includes a collection of 4,679,374 flows from the trace labeled as UNIVERSITY-2, which corresponds to a large university network. More details about this trace are described in Table 4.1.

We built our ground truth using the open-source tools nDPI [133] and Bro IDS [135], which also performs DPI. We processed the whole UNIVERSITY-2 trace with both tools to obtain extensive information about the traffic in the trace. Our ground truth includes all the flows in the trace (identified by their 5-tuple) along with associated labels that classify each of them. As for the selection of the labels, we follow the operational scheme of our classification system (Fig. 4.3). First, we select the label provided by nDPI, which classifies the flows by application protocols. In the case that nDPI classifies a flow as HTTP or SSL/TLS traffic, we substitute this label for the server hostname extracted either from the HTTP headers or the SSL/TLS certificates. We consider that a hostname is valid if it meets the following two conditions: (*i*) it has at least one character, and (*ii*) it does not correspond to an IP address. This latter condition was imposed because we could observe some flows with certificates where the SNI field contained an IP address, and it does not provide any valuable information to unveil the application generating the flow. That way, our ground truth includes a collection of labels selected by nDPI except for flows with web or encrypted traffic, where we use the Fully Qualified Domain Name (FQDN) of the server associated to the connection.

Note that this selection of labels allows us to properly evaluate our classification system for all the techniques it includes. First, we can compare our ML classifier with the results achieved by DPI, as the ML model produces only output labels included in the list of supported protocols of nDPI [133]. Furthermore, we can evaluate the server domain names obtained by the DNS module comparing them with the hostnames generated by the HTTP and Encrypted modules.

Accuracy of the Flow Classification System

As we discussed in Section 4.5, in our monitoring system, the controller can receive packets from the same sampled flow during the time interval between the reception of the first packet in the switch, and the time when a specific flow entry is installed in the switch. That way, our classification system leverages the reception of all these packets to perform traffic classification.

In this evaluation, we consider the same scenario than in Section 4.6.1, with a range from 1 ms to 100 ms for the elapsed time to install a flow entry. Thus, we simulate this range of time intervals using the UNIVERSITY-2 trace (described in Table 4.1) and analyze the accuracy and the resource consumption of our classification system. In particular, we analyze these factors separately for each of the modules within the classification system.

In order to evaluate the accuracy achieved by our system, we used the ground truth described in Section 4.6.2. This ground truth represents the results that could be achieved if all the traffic was mirrored to the SDN controller and we applied the DPI techniques in our "DPI/ML" (using nDPI), "HTTP" and "Encrypted" modules to classify every flow. However, this approach is resource consuming and not feasible in most real-world network environments. That way, we consider a scenario where we perform DPI only to those first few packets of the flows that the controller receives. Note that, in our scenario, the longer it takes to install a flow entry in the switch, the more packets from the same flow will be received in the controller. Likewise, a larger number of packets processed by our classification system will potentially lead to higher accuracy. Alternatively, we also consider the use of our supervised ML classifier (within the DPI/ML module) that collects flow-level features (described at the beginning of Section 4.6.2) to perform traffic classification at the level of application protocols. This allows us to compare the results obtained by nDPI with those achieved by ML specifically when classifying the traffic by application protocols. To measure the accuracy of our system, we consider that it classifies a flow correctly if it produces the same label than the ground truth. That is, the application protocol labels produced by nDPI for all the flows except for web and encrypted flows, which are labeled with the server hostname associated. Remark that, for the evaluation of the techniques implemented in our "HTTP", "Encrypted" and "DNS" modules, we only consider the second-level domains of the hostnames. For example, if the label in the ground truth is *www.google.com*, we consider that our system succeeds if it provides the label *google*.

In Figure 4.11, we show the accuracy results individually achieved by our classification modules as well as different combinations of them. Particularly, the y-axis represents the percentage of flows well classified, while the x-axis represents the elapsed time to install the flow entries. Note that in our "DPI/ML" module we achieve almost the same accuracy either using nDPI or using the ML model. Furthermore, the results achieved for low time intervals show that both techniques are able to classify the traffic even when the controller receives a small number of packets. Regarding the operation of nDPI, we do not have specific details of its implementation, but typically DPI tools perform classification not only considering the well-known ports, but also inspecting the payload content. Note that in this particular case, both techniques achieve similar accuracy due to the classification is done at the level of application protocols. Similarly, in [56] we can observe that they achieve very good results using a C4.5 decision tree to classify the traffic with a comparable complexity level. Nevertheless, we note that performing a comprehensive classification by

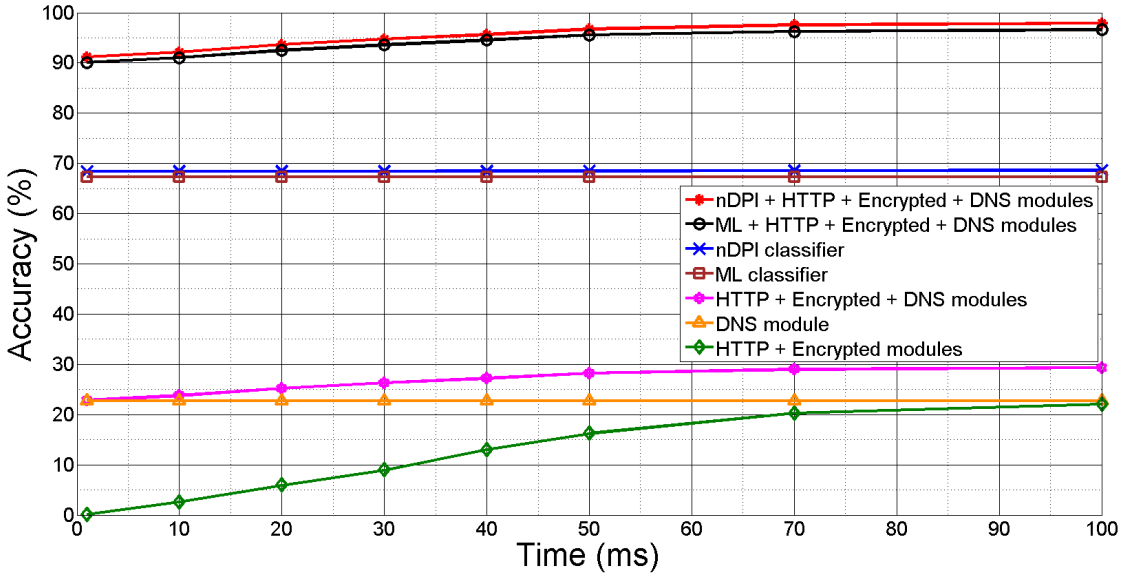


Figure 4.11: Evaluation of the accuracy of the Traffic classification system.

application names is not feasible using ML classifiers without considerably sacrificing the accuracy results. Moreover, in Figure 4.11 we also see that the HTTP and Encrypted modules significantly improve their accuracy as the time interval becomes longer. We also remark that, in terms of false positives, the DNS module provides wrong labels for around 21.4% of web and encrypted flows. However, the HTTP and Encrypted modules do not produce any false positive. This is the reason why we prioritize the labels produced by the HTTP and Encrypted modules, and use the DNS module as a fallback classifier if the other modules did not achieve an output label.

Regarding the applicability of our classification system for security purposes, we note here some issues that would be addressed as future work. First, malicious hosts could avoid being monitored if they use for example uncommon ports that nDPI or the ML classifier cannot properly detect. Moreover, the use of encrypted tunnels (e.g., SSH) prevents our system from classifying the different flows that they carry inside.

As final remark, note that it is possible to control the time interval fixed by a network scenario by delaying in the controller the execution of the order to install the flow entry in the switch. This would allow to further improve the accuracy achieved by the classifier at the expense of higher processing overhead in the controller, as it would have to process more packets.

Evaluation of the Accuracy for Specific Applications within Encrypted Traffic

In order to further analyze our system, we evaluated separately the accuracy of the system classifying specific applications within encrypted traffic, which may be of particular interest given the increasing importance of this traffic in networks and the limitations when applying

DPI-based techniques. In particular, we consider the domain names associated to the applications shown in Table 4.4. To this end, we use again the ground truth used in the previous evaluation, which has a total of 505,505 HTTPS flows with valid labels.

Table 4.4: Applications considered for the evaluation of the classification accuracy.

Application name	Domain name
Google Drive	drive.google.com
Google Gmail	mail.google.com
Netflix	netflix.com
Whatsapp web	web.whatsapp.com
YouTube	youtube.com

In figure 4.12, we show the accuracy results achieved by our classification system for the different applications. The y-axis represents the percentage of flows well classified for each application with respect to the total number of flows generated by each of them in our ground truth. It is noteworthy that the results follow different patterns depending on the application. Thus – for instance – our system achieves a high accuracy to identify flows from Netflix even in scenarios with short time intervals. However, other applications such as YouTube or Google Drive are quite more sensitive to the time interval.

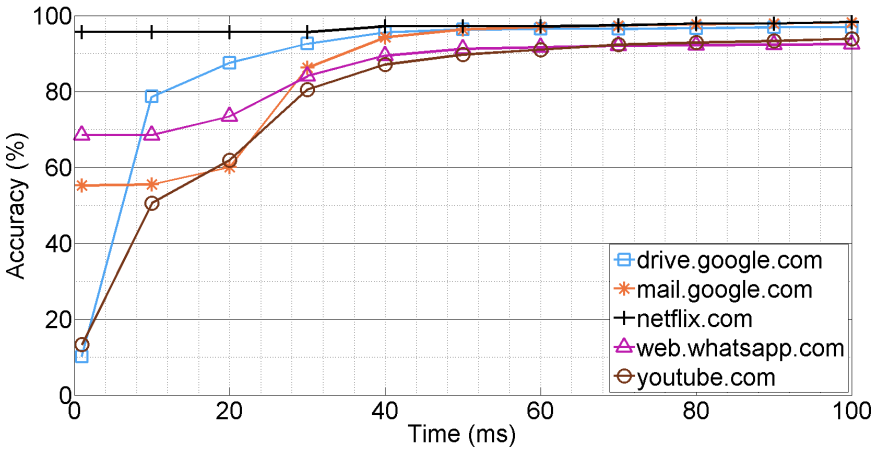


Figure 4.12: Accuracy achieved per application by the classification system.

Overhead of the Flow Classification System

Regarding the processing power required to deploy our classification system in SDN controllers, we made an evaluation of the execution cost of the different classification modules. We measure the cost by calculating the average execution time per packet. To this end, we execute some experiments in a machine with an Intel i7 quad-core processor and 8 GB of RAM memory. We process the whole UNIVERSITY-2 traffic trace with each classification

module and divide the processing time by the total number of packets in the trace. Note that each module does not process all the packets in the traffic, but only the incoming packets to the module. For example, the DNS module only processes the traffic matching port 53 (more details are described in Section 4.5). Figure 4.13 shows the results of the per-packet execution times for the different modules. From these results, we infer that the ML module performs much better than the DPI module in terms of processing overhead, as we expected. Also note that the accuracy and the execution times of our DNS module do not depend on the elapsed time (x-axis) to install the flow entries, since it is always fed by all the DNS traffic independently of this time interval. The reduced per-packet execution time of the DNS module can be explained by the low amount of DNS packets that can be typically found in real-world traffic. For example, in the trace we used in our experiments, only 0.487% of the total number of packets correspond to DNS traffic.

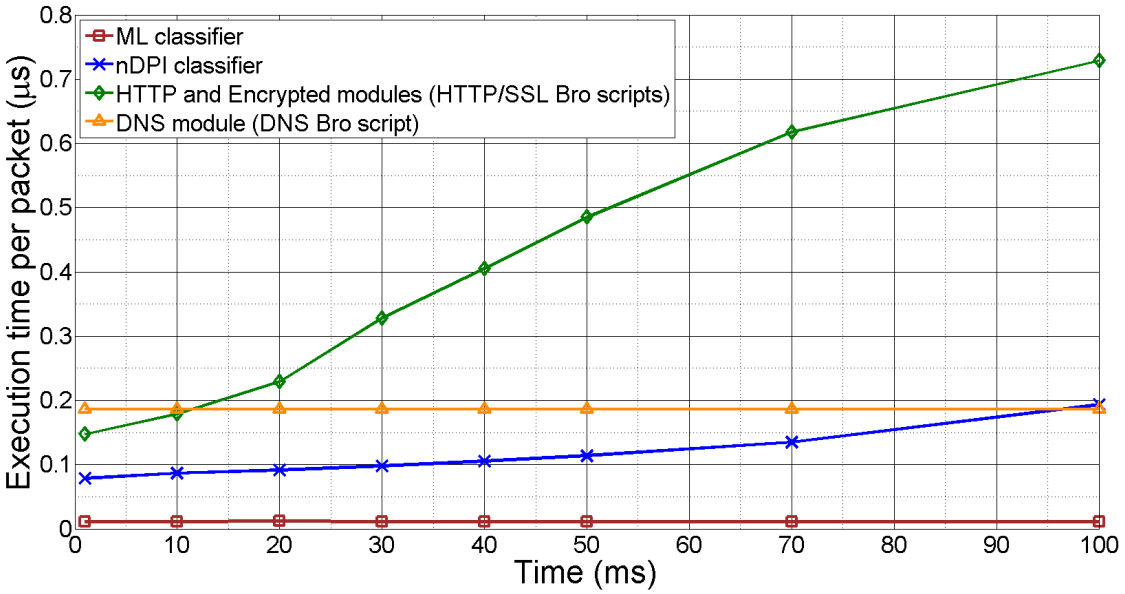


Figure 4.13: Evaluation of the overhead of the Traffic classification system.

We remark that the purpose of this evaluation is to compare in relative terms the cost of the different classification modules. Nevertheless, note that the execution times we provide can be inflated if we compare them with the computational time that they may have in real deployments, as our values include the time to read the traffic trace from the hard-disk memory of the computer.

4.7. Related Work

In this section, we provide an overview of the state-of-the-art regarding traffic measurement and traffic classification in SDN.

There are many efforts in the literature to perform traffic measurement in the SDN paradigm. However, this is an open research line that still has a lot of room for improve-

ment. For example, in [124] they use the measurement features of OpenFlow to maintain per-flow statistics in the switches. However, their approach is not scalable as they do not perform traffic sampling and it requires to install an entry in the flow tables for every flow in the traffic. In iSTAMP [116], they perform flow-based sampling by making use of a multi-armed-bandit algorithm to "stamp" some flows. However, this solution specifically addresses the detection of particular flows like *heavy hitters*, while our solution provides a generic set of the flows in the network. Alternatively, some authors suggest to make use of different architectures specifically designed for monitoring tasks. Thus, in [107], they propose using OpenSketch, where some sketches can be defined and dynamically loaded to perform traffic measurement. We also found some proposals that rely on different protocols than OpenFlow. For instance, OpenSample [117] performs traffic sampling using sFlow. Nevertheless, we consider sFlow has a high resource consumption as it sends every sampled packet to an external collector to maintain there the statistics. Other authors propose distributed solutions to address the scalability issue in SDN. Thus, in OpenNetMon [118], they design an scheme to monitor flows in edge switches and make measurements of throughput, packet loss, and delay. However, these solutions may produce high overhead in the controller, which has to calculate the paths traversed by the flows and balance as much as possible the flow entries installed in the edge switches.

Regarding traffic classification, at the time of this writing we could not find in the literature many contributions specifically addressing the SDN paradigm. We highlight for instance the architecture they proposed in [136] to select flow features for traffic classification in OpenFlow-based networks. However, for traditional networks, there are plenty of proposals to classify the traffic using different techniques ranging from DPI to ML. Thus, in our classification system we used some techniques such as the C5.0 decision tree [134] they used in [56] to classify the traffic, or those methods proposed in [122] [123], where they use specific DPI techniques addressing web, encrypted and DNS traffic.

4.8. Chapter Summary

This chapter presents a flow monitoring system for OpenFlow that provides flow-level measurement reports like those of NetFlow/IPFIX in traditional networks. Moreover, these reports are enriched with labels identifying the applications generating each flow. In order to reduce the overhead in the controller and the number of flow entries required in the switch, we proposed two traffic sampling methods that can be implemented in off-the-shelf OpenFlow switches. For traffic classification, we efficiently combined some DPI and ML techniques with special focus on the identification of web and encrypted traffic. To this end, we leverage the possibility with OpenFlow to receive in SDN controllers only the first few packets of specific flows. We implemented our system in the OpenDaylight SDN controller and evaluated its accuracy and overhead using real traffic from three different networks.

Chapter 5

Conclusions and Future Work

The coming years will see a constant emergence of applications and services that will rely on network communications with inflexible Quality of Service (QoS) requirements, such as guaranteed ultra-low latency or extremely high reliability. This, in combination with the persistent growth of the Internet traffic, is raising serious concerns among network operators, which find severe deficiencies to satisfy the upcoming operational demands of networks. In response to this, great endeavors have been made during the last decade to push networks towards software-defined architectural designs intended to gain more flexibility in network operation. We are thus witnessing the so-called "*softwarization*" process of networks.

In this line, the Knowledge-Defined Networking (KDN) paradigm was conceived – in 2016 – with the aim of empowering network control with the uptake of modern Artificial Intelligence (AI) technology. To this end, KDN exposes a novel architecture that leverages Software-Defined Networking (SDN) and recent network analytics techniques (e.g., telemetry) to facilitate the deployment of AI mechanisms. All this with the goal of optimizing and automating network operation. Since its inception, KDN has been warmly welcomed by the networking community, and nowadays it represents a flagship initiative towards achieving AI-assisted network operation in the future.

This dissertation aims to be a step forward in the realization of Knowledge-Defined Networks with autonomous operation. To this end, we revisited the conceptual design of KDN and identified two network components that – we envision – are the main enablers to achieve automation in networks: (*i*) the *automatic control module*, and (*ii*) the *network analytics platform*. The first component is tasked to perform automatically network control and optimization tasks, while network analytics provides mechanisms to process and augment the data collected from the network to eventually support a wide range of network operation functions. Thus, this thesis has presented – and validated through extensive evaluation – several fresh contributions that can be applicable to the construction of these two network components.

According to the thesis objectives, we divided the document in two parts that respectively present contributions related to the two aforementioned network components. We summarize the main contributions of this dissertation below:

Part I: Enabling Automatic Control

This part has explored the feasibility to perform automatic network operation using two recent Machine Learning (ML) techniques with revolutionary applications in other fields: (*i*) *Deep Reinforcement Learning*, and (*ii*) *Graph Neural Networks*.

Deep Reinforcement Learning DRL

Chapter 2 has investigated the application of DRL techniques to network optimization. In particular, we have focused on the problem of optimizing the routing configuration in networks. First, we argue that existing DRL-based routing solutions are limited due to the simple representations of networks that they present to DRL agents. This has led to the proposal of a more elaborate representation for the state and action spaces of DRL that makes it easier for the agents to learn how to route the traffic in networks. This representation shows more explicitly the mutual relationships between the links in network topologies and the end-to-end paths resulting from the routing configuration. To validate the properties of the proposed representation, we have made an extensive evaluation including experiments with several real-world topologies and traffic profiles, and testing the solution in two well-known routing optimization use cases: routing in Optical Transport Networks, and QoS-aware routing in IP networks. As a result, we have observed that DRL agents incorporating our representation achieve better performance and learn faster how to route the traffic with respect to other DRL-based solutions using state-of-the-art network representations.

Graph Neural Networks (GNN)

Chapter 3 has delved into the application of GNN for network optimization. First, we have presented the foundations of GNN, which is a new family of Deep Learning models specifically designed to operate over graph-structured data. Then, we have introduced RouteNet, a GNN model that is specifically tailored to learn the relationships between different network elements and estimate how they relate to different performance metrics of networks (delay and jitter). Particularly, we have used this model to produce accurate predictions of the per-source-destination mean delay and jitter given: (*i*) a network topology, (*ii*) a routing configuration, and (*iii*) a traffic matrix. The internal architecture of RouteNet uses a message passing function to exchange the information between the links in the topology, and the paths resulting from the routing scheme. As a result, it is able to learn the reciprocal dependencies between the states of paths and links, and produce accurate predictions of delay and jitter. More in detail, the evaluation results show the capability of the model to generalize over variable-size topologies, routing schemes, and traffic never seen during the training phase. To train the model, a packet-level simulator was used to generate different

datasets including topologies up to 50 nodes. Thereafter, we have used a trained RouteNet model to show its potential applicability to perform network optimization in different use cases where the target is to minimize multiple Key Performance Indicators (KPI) of networks, such as delay or jitter. Thus, we have built a routing optimizer that uses RouteNet to predict the delay and jitter for candidate routing configurations, and have evaluated it in five different relevant use cases, including delay and jitter-aware routing optimization and link failure recovery. The evaluation results show that using RouteNet may be beneficial for network optimization tasks with respect to the application of state-of-the-art optimization strategies based on link utilization. For instance, in a scenario with high traffic intensity, the proposed RouteNet-based optimizer has achieved a reduction of 12.1% for the mean delay and 27.21% for the mean jitter compared to a traditional utilization-based optimizer. Last but not least, we have publicly released all the simulation datasets used to train and evaluate the GNN model [1], which – we believe – may attract much interest from those members of the networking community interested in the application of ML for network modeling and optimization.

Part II: Network Analytics

The second part of this dissertation has investigated the design of functional and efficient solutions to collect, process, and maintain a rich and timely view of the network state. All the work presented in this part has been developed from the optic of the SDN paradigm, which offers data-plane devices with enhanced computing and storage capabilities, and a flexible Southbound API to retrieve statistics from the data plane (e.g., traffic measurements) and build a global picture of the network state in the control plane.

Flow-level Measurement and Classification in SDN

Chapter 4 has studied the design of functional and scalable solutions to perform traffic monitoring in Software-Defined Networks. As a result of this investigation, we have presented a flow-level monitoring system that is fully compliant with OpenFlow – which is one of the dominant protocols for the Southbound API in SDN. Our system provides flow-level measurement reports as those of NetFlow/IPFIX in traditional networks. To this end, it leverages the flow tables abstraction of data-plane devices in OpenFlow in order to collect and maintain flow-level measurements within switches. Similarly to NetFlow, we use some timeouts to report the traffic statistics asynchronously to the control plane. This design prioritizes scalability. For this purpose, we implemented two traffic sampling methods that have different levels of requirements of OpenFlow features supported by switches. Our evaluation results show that the use of these sampling mechanisms permit to adapt the sampling rate to the resources available in the network, which are mainly limited by: (*i*) the storage capacity in switches, and (*ii*) the processing power of SDN controllers. On top of this, we have implemented a classification system that runs in SDN controllers and classifies the traffic at the application level. This system includes a ML model – the c5.0

decision tree – that uses as input flow-level traffic measurements to classify flows at the level of the application protocol (e.g., SSH, SMTP). Moreover, we leverage the possibility to receive selectively the first few packets of some flows in SDN controllers to perform Deep Packet Inspection (DPI). This has enabled to improve the classification accuracy and also use some information from the DNS traffic and SSL/TLS certificates to identify the applications generating web and encrypted traffic. Lastly, we have implemented the proposed solution within the OpenDaylight SDN controller and made an extensive evaluation of the two modules that it integrates: (*i*) the measurement, and (*ii*) classification systems. Our experiments put special emphasis on finding the tradeoffs between the accuracy achieved by these modules and the cost to deploy them considering some configurable parameters (e.g., sampling rate, packets processed by DPI). To this end, along this chapter we have evaluated these tradeoffs using real-world traffic traces from three different large-scale networks.

Before concluding, our ambition with this dissertation is twofold. We do not only expect that the contributions presented in this thesis will represent itself significant advancements towards the achievement of the long-desired self-driving networks, but also will open up new research paths to be explored. Although some future work has already been outlined along previous chapters, we summarize below some avenues of research that may be interesting to investigate in the future:

- Build an optimizer based on DRL and GNN: We believe that this may be potentially the most interesting line for future work. In virtue of the generalization capabilities of GNNs to model network-related information structured as graphs, the integration of such models into DRL agents may enable to build more generic optimization solutions for networks. In this vein, the GNN model can help the agent acquire deeper knowledge about networks and learn what are the most meaningful features for a particular optimization problem. This would enable to generalize better to other scenarios different from those that the agent observed during the training phase.

- Extend DRL to other network optimization use cases: In this thesis we mainly explored the case of optimizing the routing configuration. However, DRL-based solutions can be applied to many other network-related problems. Some examples of this are the well-known problem of virtual network function placement in NFV scenarios, or optimizing the radio spectrum use in wireless networks.

- Design distributed DRL-based solutions for network optimization: The application of recent DRL-based techniques based on Multi-Agent Reinforcement Learning (MARL) may be an interesting avenue to investigate the design of network optimization solutions where multiple DRL agents cooperate to maximize a common optimization goal.

- Use of advanced Deep Learning (DL) models for traffic classification: Although traditional ML-based techniques such as decision trees or Support Vector Machines (SVM) are able to achieve accurate results in many traffic classification use cases, we believe that the use of modern neural network models may open up the possibility to expand the horizon of current classification techniques. Particularly, in specific cases where state-of-the-art techniques still fail to classify well the traffic. This is the case – for instance – of malware detection, or traffic classification within covert channels.

Appendices

Appendix A

List of publications

A.1. Related Publications

Journal Papers:

- José Suárez-Varela, Albert Mestres, Junlin Yu, Li Kuang, Haoyu Feng, Albert Cabellos-Aparicio, and Pere Barlet-Ros, "Routing in Optical Transport Networks with Deep Reinforcement Learning." ***IEEE/OSA Journal of Optical Communications and Networking***, vol. 11, pp. 547-558, 2019.
- José Suárez-Varela, and Pere Barlet-Ros, "Flow monitoring in Software-Defined Networks: Finding the accuracy/performance tradeoffs," ***Computer Networks***, vol. 135, pp. 289-301, 2018.

Conference Papers:

- José Suárez-Varela, Albert Mestres, Junlin Yu, Li Kuang, Haoyu Feng, Pere Barlet-Ros, and Albert Cabellos-Aparicio, "Feature engineering for Deep Reinforcement Learning based routing" in *Proceedings of the IEEE International Conference on Communications* ***(ICC)***, Shanghai, China, May 2019.
- Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio, "Unveiling the potential of Graph Neural Networks for network modeling and optimization in SDN," in *Proceedings of the ACM Symposium on SDN Research* ***(SOSR)***, San Jose, CA, USA, Apr. 2019, pp. 140-151.
- José Suárez-Varela, Albert Mestres, Junlin Yu, Li Kuang, Pere Barlet-Ros, and Albert Cabellos-Aparicio, "Routing based on Deep Reinforcement Learning in Optical Transport Networks" in *Proceedings of the Optical Fiber Communication Conference* ***(OFC)***, San Diego, CA, USA, Mar. 2019.
- José Suárez-Varela, and Pere Barlet-Ros, "Towards accurate classification of HTTPS traffic in Software-Defined Networks," in *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference* ***(I2MTC)***, Houston, TX, USA, May 2018.

- José Suárez-Varela, and Pere Barlet-Ros, "Towards a NetFlow implementation for OpenFlow Software-Defined Networks," in *Proceedings of the International Teletraffic Congress **(ITC)***, Genoa, Italy, Sept. 2017.
- José Suárez-Varela, Pere Barlet-Ros, and Valentín Carela-Español, "A NetFlow/IPFIX implementation with OpenFlow," In *TERENA Networking Conference **(TNC)***, Linz, Austria, May 2017.

Demo papers:

- José Suárez-Varela, Sergi Carol-Bosch, Krzysztof Rusek, Paul Almasan, Marta Arias, Pere Barlet-Ros, and Albert Cabellos-Aparicio, "Challenging the generalization capabilities of Graph Neural Networks for network modeling," In *Proceedings of the **ACM SIGCOMM Conference Posters and Demos***, Beijing, China, Aug. 2019.
- José Suárez-Varela, and Pere Barlet-Ros, "SBAR: SDN flow-Based monitoring and Application Recognition," in *Proceedings of the ACM Symposium on SDN Research **(SOSR)***, Los Angeles, CA, USA, Mar. 2018.

Technical reports:

- José Suárez-Varela, and Pere Barlet-Ros, "Reinventing NetFlow for OpenFlow Software-Defined Networks," arXiv preprint arXiv:1702.06803, 2017.

A.2. Other Publications

Journal papers:

- **(under revision)** Krzysztof Rusek, José Suárez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio, "RouteNet: Leveraging Graph Neural Networks for network modeling and optimization in SDN," *IEEE Journal on Selected Areas in Communications **(JSAC)***.

Conference papers:

- Arnau Badia-Sampera, José Suárez-Varela, Paul Almasan, Krzysztof Rusek, Pere Barlet-Ros, Albert Cabellos-Aparicio, "Towards more realistic network models based on Graph Neural Networks," in *Proceedings of the **ACM CoNEXT Student Workshop***, Orlando, FL, USA, Dec. 2019.
- Jordi Zayuelas, José Suárez-Varela, and Pere Barlet-Ros, "Detecting cryptocurrency miners with NetFlow/IPFIX network measurements," in *Proceedings of the IEEE International Symposium on Measurements and Networking **(M&N)***, Catania, Italy, July 2019.

A.3. Other merits

- Doctoral fellowship from the Spanish Ministry of Economy and Competitiveness, FPI grant (ref. BES-2015-073711) under the SUNSET project (ref. TEC2014-59583-C2-2-R), May 2016.
- 3-month research stay at the University of Siena: working in collaboration with Prof. Franco Scarselli to investigate the application of Graph Neural Networks for computer network modeling and optimization, *Università degli Studi di Siena*, from 2019-04-25 to 2019-07-26.
- TMA PhD school travel grants ('16, '17, and '18): awarded to selected students to attend the PhD schools on Traffic Monitoring and Analysis (TMA) in Louvain La Neuve, Belgium (2016), Dublin, Ireland (2017), and Vienna, Austria (2018).
- ACM SOSR travel grant '18: awarded to selected students to attend the ACM Symposium on SDN Research (SOSR) in San Jose, CA, USA, April 2018.

Bibliography

[1] "Knowledge-defined networking - GitHub," https://github.com/knowledgedefinednetworking, 2019.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[3] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K. J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic *et al.*, "De-ossifying the internet transport layer: A survey and future perspectives," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 619–639, 2016.

[4] J. S. Turner and D. E. Taylor, "Diversifying the Internet," in *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*, vol. 2, 2005, pp. 6–14.

[5] "Cisco visual networking index: Forecast and trends, 2017-2022 white paper," https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html, Accessed: 2019-12-23.

[6] M. A. Lema, A. Laya, T. Mahmoodi, M. Cuevas, J. Sachs, J. Markendahl, and M. Dohler, "Business case and technology analysis for 5G low latency applications," *IEEE Access*, vol. 5, pp. 5917–5935, 2017.

[7] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[8] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[9] 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, 2015.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[11] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Computational intelligence magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[12] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 3111–3119.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2012, pp. 1097–1105.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[15] D. C. Cireşan, A. Giusti, L. M. Gambardella, and J. Schmidhuber, "Mitosis detection in breast cancer histology images with deep neural networks," in *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2013, pp. 411–418.

[16] G. Litjens, C. I. Sánchez, N. Timofeeva, M. Hermsen, I. Nagtegaal, I. Kovacs, C. Hulsbergen-Van De Kaa, P. Bult, B. Van Ginneken, and J. Van Der Laak, "Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis," *Nature scientific reports*, vol. 6, no. 26286, 2016.

[17] P. S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, "Learning deep structured semantic models for web search using clickthrough data," in *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, 2013, pp. 2333–2338.

[18] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil, "Learning semantic representations using convolutional neural networks for web search," in *Proceedings of the 23rd International Conference on World Wide Web (WWW)*, 2014, pp. 373–374.

[19] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, 2019.

[20] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue *et al.*, "An empirical evaluation of deep learning on highway driving," *arXiv preprint arXiv:1504.01716*, 2015.

[21] 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, no. 7540, p. 529533, 2015.

[22] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv:1712.01815*, 2017.

[23] Z. M. Fadlullah, F. Tang, B. Mao, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "State-of-the-art deep learning: Evolving machine intelligence toward tomorrows intelligent network traffic control systems," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2432–2455, 2017.

[24] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine learning for networking: Workflow, advances and opportunities," *IEEE Network*, vol. 32, no. 2, pp. 92–99, 2017.

[25] J. Hyun, N. Van Tu, and J. W. K. Hong, "Towards knowledge-defined networking using in-band network telemetry," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2018, pp. 1–7.

[26] H. Fang, W. Lu, Q. Li, J. Kong, L. Liang, B. Kong, and Z. Zhu, "Predictive analytics based knowledge-defined orchestration in a hybrid optical/electrical datacenter network testbed," *Journal of Lightwave Technology*, vol. 37, no. 19, pp. 4921–4934, 2019.

[27] Q. Li, H. Fang, D. Li, J. Peng, J. Kong, W. Lu, and Z. Zhu, "Scalable knowledge-defined orchestration for hybrid optical-electrical datacenter networks," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 12, no. 2, pp. A113–A122, 2019.

[28] Z. Zhu, S. Liu, B. Li, and W. Lu, "AI-assisted knowledge-defined multilayer optical networks," in *Proceedings of the IEEE Photonics Society Summer Topical Meeting Series*, 2018, pp. 127–128.

[29] G. Stampa, M. Arias, D. Sanchez-Charles, V. Muntés-Mulero, and A. Cabellos, "A deep-reinforcement learning approach for software-defined networking routing optimization," *arXiv preprint arXiv:1709.07080*, 2017.

[30] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. Van Der Merwe, "The case for separating routing from routers," in *Proceedings of the ACM SIGCOMM workshop on future directions in network architecture*, 2004, pp. 5–12.

[31] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 41–54, 2005.

[32] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 1–12, 2007.

[33] K. Greene, "TR10: Software-defined networking," *Technology Review (MIT)*, 2009.

[34] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[35] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," *ACM SIGCOMM computer Communication Review*, vol. 45, no. 4, pp. 183–197, 2015.

[36] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson *et al.*, "Network virtualization in multi-tenant datacenters," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 203–216.

[37] M. Casado, N. McKeown, and S. Shenker, "From ethane to SDN and beyond," *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 5, pp. 92–95, 2019.

[38] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[39] "OpenDaylight," http://www.opendaylight.org/, Accessed: 2020-01-09.

[40] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proceedings of the workshop on Hot topics in Software Defined Networking (HotSDN)*, 2014, pp. 1–6.

[41] "POX controller," https://github.com/noxrepo/pox/, Accessed: 2020-01-04.

[42] "Ryu SDN framework," https://osrg.github.io/ryu/, Accessed: 2020-01-04.

[43] "Floodlight OpenFlow controller," http://www.projectfloodlight.org/, Accessed: 2020-01-04.

[44] D. Erickson, "The beacon OpenFlow controller," in *Proceedings of the 2nd ACM SIGCOMM workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013, pp. 13–18.

[45] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 10, 2010, pp. 1–6.

[46] Z. Cai, A. L. Cox, and T. Ng, "Maestro: A system for scalable OpenFlow control," https://hdl.handle.net/1911/96391, Tech. Rep., 2010.

[47] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[48] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

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

[50] A. Rodriguez-Natal, M. Portoles-Comeras, V. Ermagan, D. Lewis, D. Farinacci, F. Maino, and A. Cabellos-Aparicio, "LISP: a southbound SDN protocol?" *IEEE Communications Magazine*, vol. 53, no. 7, pp. 201–207, 2015.

[51] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (BGP-4)," Internet Requests for Comments, IETF, RFC 4271, 2006. [Online]. Available: https://tools.ietf.org/html/rfc4271

[52] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, G. Estrada, K. Ma'ruf, F. Coras, V. Ermagan, H. Latapie, C. Cassar, J. Evans, F. Maino, J. Walrand, and A. Cabellos, "Knowledge-defined networking," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, pp. 2–10, 2017.

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

[54] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM conference Posters and Demos*, 2015.

[55] "In-situ operations, administration, and maintenance (iOAM)," https://github.com/CiscoDevNet/iOAM, Accessed: 2020-01-06.

[56] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, and J. Solé-Pareta, "Analysis of the impact of sampling on NetFlow traffic classification," *Computer Networks*, vol. 55, no. 5, pp. 1083–1099, 2011.

[57] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Mobile encrypted traffic classification using deep learning," in *Proceedings of the Network Traffic Measurement and Analysis Conference (TMA)*, 2018, pp. 1–8.

[58] T. Shon and J. Moon, "A hybrid machine learning approach to network anomaly detection," *Information Sciences*, vol. 177, no. 18, pp. 3799–3821, 2007.

[59] J. M. N. Gonzalez, J. A. Jimenez, J. C. D. Lopez *et al.*, "Root cause analysis of network failures using machine learning and summarization techniques," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 126–131, 2017.

[60] Y. Lv, Y. Duan, W. Kang, Z. Li, and F. Y. Wang, "Traffic flow prediction with big data: a deep learning approach," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 865–873, 2014.

[61] X. Chen, J. Guo, Z. Zhu, R. Proietti, A. Castro, and S. Yoo, "Deep-RMSA: A deep-reinforcement-learning routing, modulation and spectrum assignment agent for elastic optical networks," in *Proceedings of the Optical Fiber Communications Conference (OFC)*, 2018.

[62] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2018, pp. 1871–1879.

[63] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2014, pp. 2672–2680.

[64] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[65] "NeMo: An application's interface to intent-based networks," http://nemo-project.net/, Accessed: 2020-01-08.

[66] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[67] "P416 portable switch architecture (PSA). Version 1.1," https://p4.org/p4-spec/docs/PSA-v1.1.0.html, Accessed: 2020-01-09.

[68] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015, pp. 1889–1897.

[69] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[70] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[71] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to route," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[72] S. C. Lin, I. F. Akyildiz, P. Wang, and M. Luo, "QoS-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach," in *Proceedings of the IEEE International Conference on Services Computing (SCC)*, 2016, pp. 25–33.

[73] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with Deep Reinforcement Learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets)*, 2016, pp. 50–56.

[74] J. Rexford, "Route optimization in IP networks," in *Handbook of Optimization in Telecommunications*, 2006, pp. 679–700.

[75] B. Mao, Z. M. Fadlullah, F. Tang, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "Routing or computing? the paradigm shift towards intelligent computer network packet transmission based on deep learning," *IEEE Transactions on Computers*, vol. 66, no. 11, pp. 1946–1960, 2017.

[76] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[77] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[78] O. Nachum, M. Norouzi, K. Xu, and D. Schuurmans, "Bridging the gap between value and policy based reinforcement learning," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 2775–2785.

[79] V. Mnih, A. P. Badia *et al.*, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 2016, pp. 1928–1937.

[80] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *arXiv preprint arXiv:1611.01224*, 2016.

[81] "ChainerRL," https://github.com/chainer/chainerrl, Accessed: 2019-12-20.

[82] S. M. Kakade, "A natural policy gradient," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2002, pp. 1531–1538.

[83] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2000, pp. 1057–1063.

[84] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[85] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[86] *ITU-T Recommendation G.709/Y.1331: Interface for the Optical Transport Network*, http://www.itu.int/rec/T-REC-G.709/, Std., 2016.

[87] H. Beyranvand and J. A. Salehi, "A quality-of-transmission aware dynamic routing and spectrum assignment scheme for future elastic optical networks," *IEEE/OSA Journal of Lightwave Technology*, vol. 31, no. 18, pp. 3043–3054, 2013.

[88] J. Pedro, J. Santos, and J. Pires, "Performance evaluation of integrated OTN/DWDM networks with single-stage multiplexing of optical channel data units," in *Proceedings of the 13th IEEE International Conference on Transparent Optical Networks (ICTON)*, 2011, pp. 1–4.

[89] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot, "Traffic matrix estimation: Existing techniques and new directions," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4, pp. 161–174, 2002.

[90] L. Guo and I. Matta, "The war between mice and elephants," in *Proceedings of the 9th International Conference on Network Protocols. (ICNP)*. IEEE, 2001, pp. 180–188.

[91] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks," *Computer Networks*, vol. 71, pp. 1–30, 2014.

[92] F. Ciucu and J. Schmitt, "Perspectives on network calculus: no free lunch, but still good value," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 311–322, 2012.

[93] S. Xiao, D. He, and Z. Gong, "Deep-Q: Traffic-driven QoS inference using deep generative network," in *Proceedings of the Workshop on Network Meets AI & ML (NetAI)*, 2018, pp. 67–73.

[94] A. Mestres, E. Alarcón, Y. Ji, and A. Cabellos-Aparicio, "Understanding the modeling of computer network delays using neural networks," in *Proceedings of the ACM SIGCOMM Workshop on Big Data Analytics and Machine Learning for Data Communication Networks (Big-DAMA)*, 2018, pp. 46–52.

[95] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the International Conference on Machine Learning (ICML)*, vol. 70, 2017, pp. 1263–1272.

[96] A. Varga, "The OMNeT++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference (ESM)*, 2001.

[97] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[98] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[99] P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende *et al.*, "Interaction networks for learning about objects, relations and physics," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 4502–4510.

[100] K. Rusek and P. Chołda, "Message-passing neural networks learn littles law," *IEEE Communications Letters*, vol. 23, no. 2, pp. 274–277, 2018.

[101] F. Geyer and G. Carle, "Learning and generating distributed routing protocols using graph-based deep learning," in *Proceedings of the ACM SIGCOMM Workshop on Big Data Analytics and Machine Learning for Data Communication Networks (Big-DAMA)*, 2018, pp. 40–45.

[102] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019, pp. 270–288.

[103] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[104] D. Raposo, A. Santoro, D. Barrett, R. Pascanu, T. Lillicrap, and P. Battaglia, "Discovering objects and their relations from entangled scene representations," *arXiv preprint arXiv:1702.05068*, 2017.

[105] X. Wang, R. Girshick, A. Gupta, and K. He, "Non-local neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 7794–7803.

[106] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola, "Deep sets," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 3391–3401.

[107] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," *Proceedings of the USENIX symposium on Networked Systems Design and Implementation (NSDI)*, vol. 13, pp. 29–42, 2013.

[108] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[109] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 971–980.

[110] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016, pp. 1050–1059.

[111] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[112] "Knowledge-Defined Networking - Unveiling the potential of GNN for network modeling and optimization in SDN [GitHub]," https://github.com/knowledgedefinednetworking/Unveiling-the-potential-of-GNN-for-network-modeling-and-optimization-in-SDN, 2019.

[113] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[114] "Knowledge-Defined Networking - Challenging the generalization capabilities of graph neural networks for network modeling [GitHub]," https://github.com/knowledgedefinednetworking/demo-routenet, 2019.

[115] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois, "A declarative and expressive approach to control forwarding paths in carrier-grade networks," *ACM SIGCOMM computer Communication Review*, vol. 45, no. 4, pp. 15–28, 2015.

[116] M. Malboubi, L. Wang, C. N. Chuah, and P. Sharma, "Intelligent SDN based traffic (de)aggregation and measurement paradigm (iSTAMP)," *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, pp. 934–942, 2014.

[117] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A low-latency, sampling-based measurement platform for commodity SDN," *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pp. 228–237, 2014.

[118] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in openflow software-defined networks," in *Proceedings of the IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–8.

[119] D. A. Popescu and A. W. Moore, "Omniscient : Towards realizing near real-time data center network traffic maps," *ACM CoNEXT Student Workshop*, 2015.

[120] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "FlowSense: Monitoring network utilization with zero measurement cost," in *Proceedings of the International Conference on Passive and Active Network Measurement (PAM)*, 2013, pp. 31–41.

[121] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in *Proceedings of the IEEE Network Operations and Management Symposium (NOMS)*, 2014.

[122] M. Trevisan, I. Drago, M. Mellia, and M. M. Munafò, "Towards web service classification using addresses and DNS," in *Proceedings of the International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2016, pp. 38–43.

[123] T. Mori, T. Inoue, A. Shimoda, K. Sato, S. Harada, K. Ishibashi, and S. Goto, "Statistical estimation of the names of HTTPS servers with domain name graphs," *Computer Communications*, vol. 94, pp. 104–113, 2016.

[124] L. Hendriks, R. D. O. Schmidt, R. Sadre, J. A. Bezerra, and A. Pras, "Assessing the quality of flow measurements from OpenFlow devices," *Proceeedings of the International Workshop on Traffic Monitoring and Analysis (TMA)*, 2016.

[125] B. Claise, "Packet sampling (PSAMP) protocol specifications," 2009.

[126] J. Suárez-Varela and P. Barlet-Ros, "Reinventing NetFlow for OpenFlow software-defined networks (Technical report)," *arXiv preprint arXiv:1702.06803*, 2017.

[127] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A compositional hypervisor for software-defined networks," *Proceedings of the USENIX symposium on Networked Systems Design and Implementation (NSDI)*, pp. 87–101, 2015.

[128] "MAWI Working Group traffic archive - [15/07/2016]," http://mawi.wide.ad.jp/mawi/.

[129] "The CAIDA UCSD anonymized internet traces 2016 - [18/02/2016]," http://www.caida.org/data/passive/passive_2016_dataset.xml.

[130] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," *IEEE/ACM Transactions on Networking*, vol. 13, no. 5, pp. 933–946, 2005.

[131] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Transactions on Networking*, vol. 9, no. 3, pp. 280–292, 2001.

[132] "Can OpenFlow scale?" https://www.sdxcentral.com/articles/contributed/openflow-sdn/2013/06/, Accessed: 2017-07-14.

[133] "nDPI - List of supported protocols," https://github.com/ntop/nDPI/blob/1.8-stable/src/include/ndpi_protocol_ids.h, Accessed: 2017-12-05.

[134] "Data mining tools see5 and c5.0," https://www.rulequest.com/see5-info.html.

[135] "The Bro network security monitor," https://www.bro.org/, Accessed: 2018-05-15.

[136] A. Santos, C. C. Machado, R. V. Bisol, L. Z. Granville, and A. Schaeffer-filho, "Identification and selection of flow features for accurate traffic classification in SDN," *Proceeding of the IEEE International Symposium on Network Computing and Applications (NCA)*, 2015.