Traffic representations for network measurements

Raphael Azorin

SORBONNE UNIVERSITÉ

DOCTORAL THESIS

Traffic Representations for Network Measurements

Raphael AZORIN

A thesis submitted in 2024 in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Computer Science, Telecommunications and Electronics Doctoral School
Sorbonne University (ED130) - EDITE de Paris
EURECOM Data Science Department

Committee in charge

Pietro Michiardi	EURECOM	Advisor
Massimo Gallo	Huawei Research	Co-advisor
Marco Mellia	Politecnico di Torino	Reviewer and Jury President
Marco Fiore	IMDEA	Reviewer
Chadi Barakat	Inria	Examiner
Melek Önen	EURECOM	Examiner

SORBONNE UNIVERSITY (ED130) - EDITE DE PARIS

Abstract

Doctor of Philosophy

Traffic Representations for Network Measurements

by Raphael AZORIN

Measurements are essential to operate and manage computer networks, as they are critical to analyze performance and establish diagnosis. In particular, per-flow monitoring consists in computing metrics that characterize the individual data streams traversing the network. To develop relevant traffic representations, operators need to select suitable flow characteristics and carefully relate their cost of extraction with their expressiveness for the downstream tasks considered. In this thesis, we propose novel methodologies to extract appropriate traffic representations. In particular, we posit that Machine Learning can enhance measurement systems, thanks to its ability to learn patterns from data, in order to provide predictions of pertinent traffic characteristics.

The first contribution of this thesis is a framework for sketch-based measurements systems to exploit the skewed nature of network traffic. Specifically, we propose a novel data structure representation that leverages sketches' under-utilization, reducing per-flow measurements memory footprint by storing only relevant counters. The second contribution is a Machine Learning-assisted monitoring system that integrates a lightweight traffic classifier. In particular, we segregate large and small flows in the data plane, before processing them separately with dedicated data structures for various use cases. The last contributions address the design of a unified Deep Learning measurement pipeline that extracts rich representations from traffic data for network analysis. We first draw from recent advances in sequence modeling to learn representations from both numerical and categorical traffic data. These representations serve as input to solve complex networking tasks such as clickstream identification and mobile terminal movement prediction in WLAN. Finally, we present an empirical study of task affinity to assess when two tasks would benefit from being learned together.

Keywords: traffic measurements – machine learning – network data representation – networked systems – learned sketches – sequence modeling – multitask learning

Acknowledgements

First and foremost, I am deeply grateful to both my advisors, Prof. Pietro Michiardi and Dr. Massimo Gallo, for the invaluable lessons they taught me throughout this PhD study. I could not have undertaken this journey without their continuous support and persistent patience. They constantly demonstrated me the tremendous value of their expertise and experience in conducting thorough science. In particular, they taught me how important it is in research to ask the right questions. I truly thank them for the great amount of time they took to train me.

Second, I would like to express my sincere thanks to my collaborators and co-authors, from Huawei Paris Research Center and Sapienza University in Rome, for their treasured support and technical expertise. During this PhD, I have been extremely fortunate to be surrounded by talented researchers that have always been excited to discuss ideas and available to provide guidance. They have been deeply influential in shaping my experience and skills during this journey, and I truly thank them for making these years a wonderful time.

Last, but not least, I would like to offer my special thanks to my family and to my friends. In particular, I would like to express my profound gratitude to my partner, who does not know much about data science, but knows a lot about me. Without her immense support and encouragement, I could not have completed this study.

Contents

List of Figures

List of Tables

List of Abbreviations

ANN	**A**rtificial **N**eural **N**etwork
AP	**A**ccess **P**oint
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
CHT	**C**uckoo **H**ash **T**able
qCHT	**q**uotiented **C**uckoo **H**ash **T**able
CMS	**C**ount **M**in **S**ketch
CNN	**C**onvolutional **N**eural **N**etwork
CV	**C**omputer **V**ision
DCN	**D**ata **C**enter **N**etwork
DDoS	**D**istributed **D**enial **o**f **S**ervice
DDSketch	**D**istributed **D**istribution **Sketch**
DL	**D**eep **L**earning
ES	**E**lastic **S**ketch
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GS	**G**radient **S**imilarity
GT	**G**radient **T**ransference
HLL	**H**yper **L**og **L**og
IAS	**I**nput **A**ttribution **S**imilarity
IAT	**I**nter-**A**rrival **T**ime
IBLT	**I**nvertible **B**loom **L**ookup **T**able
pIBLT	**p**erfect **I**nvertible **B**loom **L**ookup **T**able
IP	**I**nternet **P**rotocol
ISP	**I**nternet **S**ervice **P**rovider
LI	**L**abel **I**njection
MAT	**M**atch-**A**ction **T**able
ML	**M**achine **L**earning
MTL	**M**ulti-**T**ask **L**earning
NIC	**N**etwork **I**nterface **C**ard
NLP	**N**atural **L**anguage **P**rocessing
P4	**P**rogramming **P**rotocol-independent **P**acket **P**rocessors
QoE	**Q**uality **o**f **E**xperience
QoS	**Q**uality **o**f **S**ervice
RF	**R**andom **F**orest
RSA	**R**epresentation **S**imilarity **A**nalysis
STL	**S**ingle **T**ask **L**earning
TD	**T**axonomical **D**istance
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
W2V	**W**ord **2** **V**ec
WLAN	**W**ireless **L**ocal **A**rea **N**etwork

List of Symbols

a	Scalar
$\mathbf{a}$	Vector
$\mathbf{A}$	Matrix
$\mathcal{D}_N$	Dataset of N samples
$\mathbf{X}$	Dataset inputs (matrix of N rows, one for each instance)
$\mathbf{Y}, \mathbf{y}$	Dataset outputs (matrix, vector of N rows, one for each instance)
$\mathbf{x}_i, x_i$	i-th input instance (vector, scalar)
$\mathbf{y}_i, y_i$	i-th output instance (vector, scalar)
$\hat{\mathbf{y}}_\mathbf{i}, \hat{y}_i$	i-th prediction of a model (vector, scalar)
$\mathcal{L}$	Loss
$f : \mathcal{X} \to \mathcal{Y}$	Function f with domain $\mathcal{X}$ and range $\mathcal{Y}$
$f(x)$	Evaluation of function f at location x
T_j	Table j
$T_j[x]$	Bucket from table j at location x
$\mathbb{N}$	Natural numbers
$\mathbb{Z}$	Integer numbers
$\mathbb{R}$	Real numbers
$Card(S)$	Cardinality of the multiset S
$\mathbb{E}[\cdot]$	Expectation
$\mathcal{O}(\cdot)$	Big-O
$\lvert \cdot \rvert$	Absolute value function
$\lfloor \cdot \rfloor$	Floor function
$\lceil \cdot \rceil$	Ceiling function
$\lVert \mathbf{a} \rVert_1$	L_1 norm of vector $\mathbf{a}$

To all the teachers I had the good fortune to learn from…

Chapter 1

Introduction

1.1 The Growing Complexity of Networks

Since their inception in the early 1960s, computer networks have dramatically evolved in terms of scale, speed and capabilities. At its origin, networking was intended to deal with the cooperation of time-shared and expensive computers (Marill and Roberts, 1966). The pioneering ARPANET was the first network to connect remote machines that interacted by exchanging information. Built on the concept of open-architecture, networking has then fostered the interconnection of different and independent computer networks, governed by fundamental communication principles, including addressing, routing, and flow control protocols. In the last decades, as computing technology became more and more accessible, and information sharing took a prominent place in society, networking activity has spread across organizations and countries. In contrast to the early days of networking, modern networks operate at unprecedented scale and speed. Analysts from global manufacturer Cisco estimate that there are 30 billion networked devices in the world, operating at an average 110 Mbps broadband speed, which has more than doubled just in the last five years (Cisco, 2020). This aggregate growth, mostly supported by the consumer segment, should not obfuscate the considerable evolution of enterprise networks. IT departments are transforming their infrastructures to support the increasing adoption of novel networked applications, e.g., based on cloud computing, Internet-of-Things, Big Data, or artificial intelligence. In particular, the growing rate of data collection and processing stresses the importance of Data-Center Networking (DCN), with its own architectural paradigms and protocols (Alizadeh et al., 2010). In terms of speed, already ten years ago, it was not unusual for data centers to carry aggregate traffic at 100 Tbps over 100K servers (Zhu et al., 2015). In addition, the following years have witnessed the global data center traffic grow over 25% year-on-year (Cisco, 2018). Consequently, service providers are adapting their resources and offers to cope with enterprises' expanding requirements, e.g., in terms of bandwidth and latency demands, but also in terms of flexibility. In networking, processes are abstracted into logically separated planes of existence. Two planes are usually referenced: the control plane, that defines and controls how data is forwarded from one device to another, and the data plane, that corresponds to the actual data forwarding process (Yang et al., 2004). By separating these two planes, novel automation opportunities have emerged with Software-Defined Networking (SDN), that provides a programming interface for a controller to modify the configuration of the underlying devices' data plane.

Yet, this unparalleled growth of networks comes with complex challenges in Operations, Administration and Management (OAM). The steep increase in speed, volume, and diversity of network traffic translates into novel problems for operators to master their networks. From 2013 to 2019, the number of active flows has been estimated in the order of 100K for each Gb of forwarded traffic (Scazzariello et al., 2023). Simultaneously, novel requirements have emerged in terms of reliability, latency, or throughput to serve highly diverse applications, e.g., in DCN (Xia et al., 2017). In turn, this stresses the need for operators to optimize network design (e.g., topology, linking) and operation (e.g., energy consumption). With the advent of programmable networks since the early 2010s, the number of roles and dependencies between network devices keeps augmenting. Virtualization and SDN open new avenues for network management, giving more capabilities to the data plane in addition to its forwarding role, e.g., telemetry (Ben Basat et al., 2020b), congestion control (Li et al., 2019), or even Machine Learning (ML) (Xiong and Zilberman, 2019). However, these functionalities compete for scarce resources and put further pressure on operators to optimize their networks. In this context, the monitoring of network behaviors becomes challenging. Consequently, configuring networks to implement critical changes such as fault resolution or load balancing, is more demanding for engineers. Last, networking's mass adoption across individuals and companies is also accompanied by the rapid growth of cybersecurity threats and malicious activity. These include ransomware, worms, botnets or spam attacks, whose size and frequency keep increasing. For example, the worldwide number of Distributed Denial of Service (DDoS) attacks has almost doubled in the past five years (Cisco, 2020). To protect networks, gaining visibility thanks to measurements and monitoring has become a key priority in IT teams defense strategies, in particular for Internet Service Providers (ISPs) (Netscout, 2023). As networks grow in scale, speed, and diversity, developing a precise understanding of networks operations is both increasingly critical and challenging. In this regard, network measurements constitute a prerequisite for operators wishing to apprehend network behavior, optimize performance and ensure operational security.

1.2 The Importance of Traffic Monitoring

Traffic monitoring is the process of capturing and analyzing traffic to provide insights on both the infrastructure status and the data flowing through the network. A monitoring system is composed of probes or analyzers, which are hardware or software apparatus programmed to collect measurements, either passively or actively. In particular, passive monitoring systems are used to observe the network in a non-intrusive manner, without injecting traffic nor interfering with it. Measurement systems are crafted to extract specific traffic characteristics (e.g., throughput, losses) that facilitate various networking tasks, including performance evaluation, traffic engineering, and security management. These selected traffic features aim at bridging the "semantic gap" between users' needs and the raw data supplied by the network (Varghese and Estan, 2004). As a result, many metrics and methodologies have been proposed to realize such systems, ranging from aggregate monitoring to detailed traffic dissection (D'Alconzo et al., 2019; Li et al., 2013; Yassine et al., 2015; Chaudet et al., 2005). In a nutshell, network measurements provide operators with a view of the network state by extracting knowledge from raw traffic data.

In particular, the ability to isolate and monitor subsets of the traffic is paramount to debug several critical events and performance anomalies, e.g., switch faults (Zhu et al., 2015), routing loops (Kučera et al., 2020) and blackholes (Zhou et al., 2020). By collecting statistics on specific flows of packets that share a common property designated as the flow key, per-flow measurements support fine-grained analyses of network traffic. Typical flow keys include the 5-tuple, used e.g., in traffic classification, that is composed of the source and destination IP addresses, ports, and transport protocol; or the source IP, used e.g., in super-spreader detection. The flow characteristics selected for monitoring (e.g., size, latency) capture the traffic information that is deemed useful by operators to solve OAM tasks. Hence, per-flow monitoring deals with the characterization of individual traffic data streams. This level of granularity is key to detect and investigate important network events such as congestion (Lee et al., 2015; Li et al., 2019), packet drops (Li et al., 2016a; Li et al., 2016b), anomaly detection (Sekar et al., 2008), or forensics investigation (Xie et al., 2005). Furthermore, as modern networks typically support a wide variety of interdependent applications and services, understanding their dependencies is essential for network maintenance and optimization. In this respect, per-flow measurements enable network operators to infer the structure and relationships of the applications running on their networks (Popa et al., 2009). In this quest to gain a complete understanding of their networks, operators aim at monitoring as many flows as possible.

However, the large amount of active flows in high-speed networks makes the accurate monitoring of *all* the flows impractical at scale. In practice, passive measurement functions are typically implemented inside network devices (e.g., switches, routers) to cope with their fast-forwarding pace. Although programmable devices provide custom per-packet logic, their memory constraints, in the dozens of MBs (Zeng et al., 2022), forbid the accurate monitoring of all the flows at scale. For example, considering a Tbps forwarding device in a data center, monitoring the millions of active flows present with just a handful of accurate per-flow measurements would require several GBs of memory (Scazzariello et al., 2023). To address this overhead issue, measurement systems trade off memory for accuracy or coverage. For example, NetFlow and sFlow rely on packets sampling, by capturing only a fraction of the flow packets to estimate the metrics of interest (Jang et al., 2020). Another option consists in restricting monitoring to a subset of the active flows, i.e., filtering or sampling flows. While these techniques successfully reduce the processing overhead, they discard a non-negligible share of the available information, risking to produce biased or incomplete analysis, e.g., missing critical packets or flows. Alternatively, sketches are a family of probabilistic data structures employed to aggregate and summarize large amounts of stream data with a small memory footprint (Han et al., 2022). However, as they usually share memory across several flows, sketches summaries suffer from collisions which produce estimation errors. As such, sketches can be considered as lossy compression data structures to gain approximate knowledge about traffic behaviors. To sum up, it is challenging for network operators to extract approximate but sufficient knowledge from their traffic. Segregating traffic into flows, described by a set of hand-picked metrics, constitutes a first level of abstraction to represent network activity. This process of developing suitable flow representations, biased to accommodate various cost-expressiveness trade-offs, is key in the design of efficient network measurement systems. Hence, the selection, extraction, and processing of traffic representations is a crucial research topic.

1.3 The Promises of Traffic Representation Learning

Unlike other data modalities, e.g., images, extracting the right representation of network traffic is still an open question in the research community. First, the raw traffic under monitoring may be represented at various levels of granularity. Depending on the intended task, the aggregated traffic observed from a vantage point may be dissected into individual flows, further characterized up to packet-level measurements. For instance, identifying if a link is congested relies on aggregated throughput information, while pinpointing the specific application responsible for congestion requires flow-level measurements. Second, diverse attributes may be selected to describe the traffic under monitoring, which represent the characteristics believed to be valid for a corresponding task. From coarse-grained properties (e.g., traffic protocol or class) to fine-grained measurements (e.g., inter-arrival times, packet sizes), there is a wide spectrum of options for operators to extract traffic features. Finally, the collection and storage of traffic measurements are limited by the resources available on network devices. Operators wishing to monitor a large amount of traffic typically compromise by trading off accuracy for memory, e.g., using sampling, sketches, or exact counting for a subset of the traffic. The measurement approximations stemming from these trade-offs impact the final characterization of the traffic. For example, describing flows by their exact length is a costly characterization, but it may be justified for sensitive use cases monitoring a portion of the traffic, e.g., debugging flooding attacks (Kim et al., 2004). Alternatively, imprecise information on flow length, e.g., elephants-mice classification, might be sufficient for scheduling (Mitzenmacher, 2021). Ultimately, traffic representations aim at best describing the characteristics believed to be valid for a given task under resource constraints. Thus, we reject the vision of a silver bullet representation that would fit all networking tasks.

Thanks to its ability to extract patterns from data, ML stands as a promising candidate to elicit traffic representations. In particular, ML holds promises to provide *a priori* knowledge of flow characteristics learned from previous observations, i.e., *predicting* measurements. For example, flow length prediction would benefit many network management tasks (Đukić et al., 2019). ML algorithms have already been applied to a large range of networking tasks, including e.g., traffic prediction with time-series forecasting, or attack detection with clustering (Boutaba et al., 2018). Yet, these approaches, mostly tested offline, face significant challenges to make their way into production, namely robustness and overhead concerns. Indeed, networks are dynamic environments with non-stationary data distributions, meaning that traffic patterns are assumed to be temporally correlated and to change continuously (O'Reilly et al., 2014). Therefore, predictive network models are expected to either evolve, e.g., with online learning, or drift and degrade over time. Furthermore, performing in-network Machine Learning at line rate requires implementing models to operate in the data plane, hence competing for scarce resources against critical forwarding functionalities. Several works have addressed the implementation of lightweight models, e.g., Random Forests, in programmable network switches (Xiong and Zilberman, 2019; Zhang et al., 2021b; Busse-Grawitz et al., 2022; Akem et al., 2022). However, we argue that considerable efforts are still needed to demonstrate the benefits of ML-based traffic characterization over non-learned baselines, especially relating the overhead of the end-to-end task performance over time.

With the recent rise of Deep Learning (DL) to extract rich representations from data to solve complex tasks, researchers are starting to envision "self-driving networks" (Feamster and Rexford, 2017). Intelligent networks able to drive themselves will rely on a holistic representation of their state, upon which models may extract measurements and translate high-level objectives into actions. However, network activity is captured into various modalities, including but not limited to traffic. For example, logs (e.g., text), topologies (e.g., graphs), or configuration files (e.g., structured data) also contain information on the network. As a result, next-generation network models and digital twins are anticipated to integrate various data sources to produce a complete representation of the network state (Zeydan and Turk, 2020; Hui et al., 2023; Behringer et al., 2021). In this regard, thanks to their ability to automatically extract features as an integral part of the end task, Deep Learning algorithms are gaining attention from the networking community. Several networking problems have been modeled with DL approaches, including e.g., Graph Neural Networks

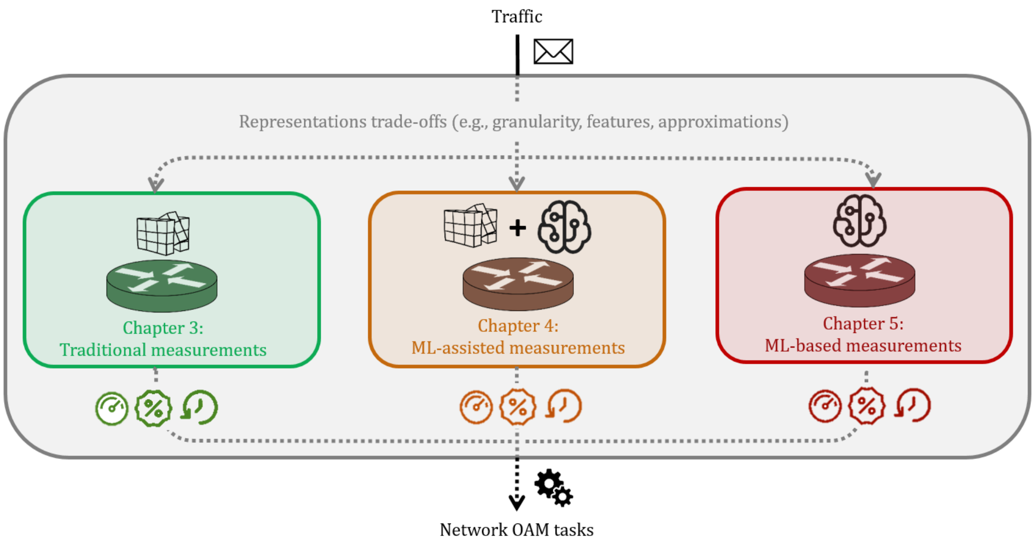


FIGURE 1.1: Illustrative approaches for traffic representations extraction, with varying degrees of Machine Learning integration.

for path delay, jitter and losses estimation (Rusek et al., 2020) or Recurrent Neural Networks for DCN performance modeling (Zhang et al., 2021a). Interestingly, some networking tasks may be related to one another or have mutual dependencies. Thus, they may benefit from a shared representation of their input. For example, the same model might simultaneously provide predictions for flow completion time, along with path delay and throughput (Wang et al., 2022b). Similarly, a single model backbone may learn flow features that are useful to solve several classification tasks at the same time, such as identifying encapsulation technique, traffic type and application name (Nascita et al., 2023). In this regard, Multi-Task Learning (MTL) is a learning paradigm that aims at sharing the extraction of common representations across related tasks. In pursuit of a holistic view of the network state used as a basis for various tasks, Deep Learning in general, and MTL in particular, constitute attractive representation learning approaches for the networking community to explore. Nonetheless, the challenges already mentioned for traditional ML are only more stringent when considering DL for networking. In addition to heavy computational requirements, Deep Learning algorithms' lack of interpretability further hinders their adoption in practice (Zhang et al., 2022)

Ultimately, developing appropriate traffic representations for network measurements is a challenging process. It involves the selection of suitable characteristics, carefully related to their cost of extraction and their expressiveness, that are specific for the considered downstream tasks. In this thesis, we consider three approaches to improve this process, with varying degrees of Machine Learning involvement, as depicted in Figure 1.1. We propose several methodologies to facilitate knowledge extraction from traffic, starting with the design of tailored data structures for *traditional* sketch-based measurements, followed by the prototyping of an *ML-assisted* measurement system, concluding with *ML-based* representation learning procedures.

1.4 Thesis Outline and Contributions

Outline

In order to address the challenges previously mentioned, we organize the contributions of this thesis into the following chapters:

- Chapter 2 serves as a background for the rest of the thesis. It starts with a review of the data structures used for network measurements (e.g., sketches), then, it introduces important Machine Learning concepts and describes the algorithms used throughout the remainder of the thesis.

- Chapter 3 presents the first contribution of the thesis, positioned in the scope of *traditional measurements*. In particular, we address the problem of sparsity in sketch-based monitoring systems and introduce novel representations to better exploit the skewed nature of network data. We note that many sketches are designed for extreme scenarios and are over-provisioned in practice, leaving most counters empty. We propose a framework to store only non-zero counters, thanks to a novel flow-to-counters mapping in place of the traditional flow-to-sketch mapping. The additional overhead induced by this indexing is compensated by the fewer counters to maintain, which depends on the sparsity of the sketch. We evaluate our framework on four real traffic traces, using three per-flow sketches: DDSketch (Masson et al., 2019) for Inter-Arrival Times quantiles estimation, HLL (Flajolet et al., 2007) for cardinality estimation and ElasticSketch (Yang et al., 2018) for flow size estimation. Our experiments show that our framework achieves from $2\times$ to $11\times$ memory requirements reduction wrt. state-of-the-art, while preserving the same accuracy.

- Chapter 4 introduces *ML-assisted measurements* where a Machine Learning model improves the usage of classic monitoring data structures. This chapter presents the second contribution of the thesis, in which we consider the practical implementation of a ML model in the data plane to provide timely classification of incoming flows. Specifically, we observe that coarse-grained flow length predictions (i.e., elephant or mice) would benefit several networking tasks including flow scheduling, packet inter-arrival time estimation, and flow size estimation itself. We develop and integrate a lightweight Random Forest to classify flows as early as the fifth packet arrival. These predictions are then leveraged by three downstream tasks at line rate. The effectiveness of our system is demonstrated through extensive experiments on three real traffic traces (CAIDA, MAWI, and UNI). We compare the tasks' end-to-end performance, including overhead, against state-of-the-art baselines, and our results show that our system is robust to traffic patterns changes and outperforms non-learned approaches.

- Chapter 5 presents *ML-based measurements*, where we develop Deep Learning models that extract traffic representations to fuel downstream network measurements applications. This chapter contains the last two contributions of the thesis. We first introduce a methodology to learn rich flow representations from both numerical and categorical traffic data. In particular, we remark that traffic data is typically composed of quantities (numerical variables such as measurements) related to entities (categorical variables such as ports, domain names or access points). We revisit recent advances in sequence modeling from Natural Language Processing (Mikolov et al., 2013b) and adapt them to the networking domain to encode sequences of entities based on their co-occurrences. We demonstrate the benefit of considering categorical data in addition to numerical measurements on two network analysis tasks: clickstream identification for ISPs and terminal movement prediction in a WLAN. Our results show that embedding network entities is not only profitable when coupled to measurements, but can also outperform quantities-only approaches. Finally, we motivate the design of a Multi-Task Learning pipeline that extracts flow representations shared across related tasks. Specifically, to facilitate the grouping of cooperative tasks together, we conduct a benchmark of six task affinity metrics, either borrowed, revisited, or novel. Our empirical campaign reveals how, even in a small-scale scenario, task affinity scoring does not correlate well with actual MTL performance. Yet, some metrics can be more indicative than others
- Chapter 6 concludes the thesis by commenting on the contributions above and discussing their potential impact in the networking and Machine Learning communities. Additionally, it discusses perspectives for future research on traffic representations for network measurements.

Publications

This thesis is based on publications that have been peer-reviewed by technical program committees in international conferences and workshops, mainly in the networking community and for some part in the Machine Learning community.

Chapter 3 is based on the following journal publication, that was presented at ACM CoNEXT 2023. It was a collaboration with Sapienza University in Rome that contributed its expertise in P4 for prototyping:

- Andrea Monterubbiano, **Raphael Azorin**, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi. "SPADA: A Sparse Approximate Data Structure Representation for Data Plane Per-Flow Monitoring". In *Proceedings of the ACM on Networking* (2023). DOI: 10.1145/3629149

Chapter 4 is based on the following paper accepted for publication in PACMNET, which will be presented at ACM CoNEXT 2024. It was another collaboration with Sapienza University that contributed its expertise in data plane programmability:

- **Raphael Azorin**, Andrea Monterubbiano, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi. "Taming the Elephants: Affordable Flow Length Prediction in the Data Plane". In *Proceedings of the ACM on Networking* (2024). DOI: 10.1145/3649473

This paper is an extension of the following extended abstract:

- Andrea Monterubbiano, **Raphael Azorin**, Gabriele Castellano, Massimo Gallo, and Salvatore Pontarelli. "Learned Data Structures for Per-Flow Measurements". In *Proceedings of the 3rd International ACM CoNEXT Student Workshop* (2022). DOI: 10.1145/3565477.3569147

Part of this work was also presented in this conference poster:

- Andrea Monterubbiano, **Raphael Azorin**, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi. "Memory-Efficient Random Forests in FPGA SmartNICs". In *Proceedings of the 19th International Conference on Emerging Networking EXperiments and Technologies. CoNEXT Posters.* (2023). DOI: 10.1145/3624354.3630089

Chapter 5's main idea of a unified Deep Learning measurements pipeline has been introduced in the following extended abstract:

- **Raphael Azorin**, Massimo Gallo, Alessandro Finamore, Maurizio Filippone, Pietro Michiardi, and Dario Rossi. "Towards a Generic Deep Learning Pipeline for Traffic Measurements". In *Proceedings of the 2nd ACM CoNEXT Student Workshop* (2021). DOI: 10.1145/3488658.3493785

The single-task learning approach has been concretized in the following international conference publication:

- Zied Ben Houidi, **Raphael Azorin**, Massimo Gallo, Alessandro Finamore, and Dario Rossi. "Towards a Systematic Multi-Modal Representation Learning for Network Data". In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks (HotNets)* (2022). DOI: 10.1145/3563766.3564108

This has been further developed in the following conference workshop paper for the multi-task learning approach:

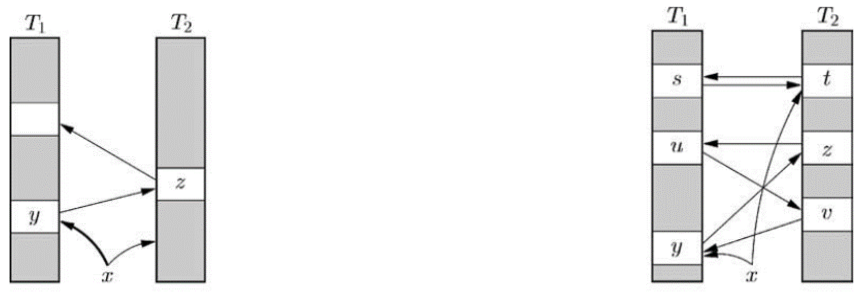
- **Raphael Azorin**, Massimo Gallo, Alessandro Finamore, Dario Rossi, and Pietro Michiardi. "It's a Match! A Benchmark of Task Affinity Scores for Joint Learning". In *2nd International Workshop on Practical Deep Learning in the Wild* during the 37th AAAI Conference on Artificial Intelligence (2023). ARXIV: 2301.02873

The technical chapters of this thesis include materials reformatted from these publications. Although parts of these papers are reproduced in extenso, we omit the quotation marks for readability.

Chapter 2

Background

In this chapter, we lay the background that supports the rest of the thesis. First, we introduce the data structures considered for network measurements in this work, starting with advanced key-value stores and following with popular sketches for frequency, cardinality and distribution estimations over data streams. Then, we present key principles and algorithms in Machine Learning, introducing Random Forests, Word2Vec and Multi-Task Learning. The remainder of the thesis is built on the measurements and models introduced in this chapter.



(A) (B)

FIGURE 2.1: Cuckoo Hashing insertion examples with two tables. **(left)** x is inserted with success by moving y and z. **(right)** x cannot be successfully inserted and a re-hash is required. Figures from (Pagh and Rodler, 2004).

2.1 Measurements Data Structures

In this section, we present the various data structures used throughout the thesis for network measurements. We first introduce two sophisticated key-value mapping data structures: Cuckoo Hash Tables with quotienting and Invertible Bloom Lookup Tables. Then, we detail state-of-the-art sketches used for network measurements.

2.1.1 Key-Value Storage

For per-flow network monitoring, many use cases require mapping a flow to some measurements, stored as `<key,value>` pairs. This mapping is typically implemented in a network device for monitoring at line-rate. Here, we detail two particular data structures used for such mappings.

Cuckoo Hash Tables with quotienting. A Cuckoo Hash Table (CHT, Pagh and Rodler, 2004) can be seen as a dictionary with constant lookup time. It is composed of d tables $T_0 \dots T_{d-1}$ with r buckets each, and d hash functions $h_0 \dots h_{d-1}$ are used to map each key x to the range $[0, r-1]$. A `<key,value>` item with key x is stored in only one of the d buckets $T_i[h_i(x)]$. To insert x, the first position $T_0[h_0(x)]$ is located among the d designated buckets. If a free cell is found, x is inserted there with its value and the procedure ends. If not, x is still inserted in this bucket, replacing the old key y previously stored there[1] and y is reinserted in one of its other available d buckets, i.e., $T_j[h_j(y)]$ for $j \neq i$. The procedure is repeated until all elements are stored, as illustrated in Figure 2.1a with $d = 2$ tables. Note that this procedure may loop, as depicted in Figure 2.1b. Thus, the number of loop iterations is bounded by an external threshold specified by the user, and if this threshold is reached, the keys are re-hashed with new hash functions. Insertions are performed in worst-case $\mathcal{O}(n)$ time, where n denotes the number of items, although a constraint on the maximum

[1]In the original CHT paper, insertion is biased towards the first table for faster lookups. Some implementations favor faster insertion instead, by checking for an empty bucket across all $T_i[h_i(x)]$ locations to insert x before moving any of the already stored `<key,value>` pairs.

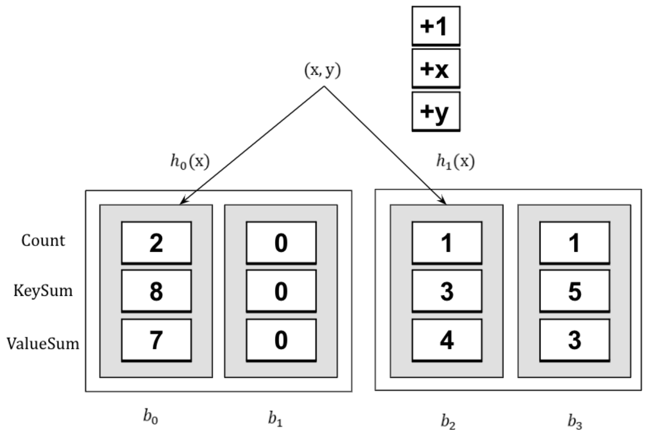


FIGURE 2.2: Insertion of the pair (x, y) in an Invertible Bloom Lookup Table with $m = 4$ buckets and $d = 2$ hash functions. The new item increments the three fields in each of the buckets designated by $h_i(x)$.

table load factor enables amortized constant insertion time (Pagh and Rodler, 2004). The lookup operation is performed in $\mathcal{O}(1)$ time as the key x is found in either one of the fixed d locations indexed by $T_i[h_i(x)]$. Variants of cuckoo hashing proposed to use additional hash functions (Fotakis et al., 2005) or to use buckets of size larger than one (Panigrahy, 2004) to achieve higher load factors.

Quotienting is a technique, originally proposed by (Knuth, 1973), to reduce the memory needed to store keys: consider a universe $\mathcal{U}$ comprising 2^u elements of u bits each and d bijective functions $m_i : \mathcal{U} \to \mathcal{U}$. With quotienting, we use the r least significant bits of $m_i(x)$ as the hash function $h_i(x)$, and store only the remaining $u - r$ bits of $m_i(x)$ in $T_i[h_i(x)]$, i.e., the quotient of $m_i(x)$. This method enables unambiguous identification of the items stored in a cuckoo table while reducing the memory cost of each key from u to $u - r$. Note that quotienting provides significant savings when $u \approx r$, i.e., when a significant fraction of the universe is stored in the qCHT.

Invertible Bloom Lookup Tables. An Invertible Bloom Lookup Table (IBLT, Goodrich and Mitzenmacher, 2011) is a randomized data structure that enables storing `<key, value>` pairs with constant insertion time. Similar to a Bloom Filter (Bloom, 1970), an Invertible Bloom Lookup Table (IBLT) uses a single lookup table T of m buckets, and d hash functions $h_0 \dots h_{d-1}$ to identify the d buckets $T[h_i(x)]$ where a pair (x, y) is placed. Note that, to ensure that the hashes point to distinct locations, a standard approach consists in splitting the m buckets into d tables of size m/d. To solve key collisions, each bucket contains three fields: a counter for the number of colliding keys, a field to sum (or XOR) the colliding keys, and a values-store summing (or XORing) all the values associated with the colliding keys. The insertion of a new item updates these three fields for each of the d buckets it is placed into, as illustrated in Figure 2.2. This operation is supported in $\mathcal{O}(1)$ time as it only depends on the number of hash functions used, which is fixed. The lookup for a key x is also performed in $\mathcal{O}(1)$ time, following a procedure similar to a membership test in a Bloom

Filter: each of the d buckets designated by the hash locations $h_i(x)$ is inspected. If one of them has its keys-counter set to zero, then x is not present in the IBLT. If one of them has its keys-counter set to one, then either its keys-sum field is equal to x and the corresponding value is returned from its values-sum field, otherwise x is not present in the IBLT. Finally, if the previous tests have failed for all d buckets, then the lookup procedure fails and x may or may not be in the IBLT. Lookup failures have low probability similar to the false-positive rate of a Bloom Filter (Goodrich and Mitzenmacher, 2011).

A specificity of the IBLT is that is allows listing its content with a decoding procedure that performs a "peeling process". Intuitively, the peeling process operates by first identifying the buckets where the keys-counter is equal to 1, i.e., only one value was stored there. Then, this value is removed from all the d associated buckets, and their keys-counter, keys-sum and values-sum fields are decremented accordingly. Hopefully, then other buckets end up having their keys-counter equal to one, hence the process continues iteratively. This procedure can retrieve all the `<key,value>` pairs if the load factor of the IBLT is below a certain peeling threshold.

2.1.2 Frequency estimation

In network measurements, flow size corresponds to the number of packets or bytes composing a flow. When considered in terms of packets, flow size measurement corresponds to a particular application of frequency estimation on a stream of items. Flow size distributions are typically skewed and heavy-tailed (Han et al., 2022): a small fraction of the flows with very large sizes (the elephants) account for most of the traffic, while the majority of flows are composed of a small number of packets (the mice). Because of their impact on the network, large flows are of particular interest for many applications, e.g., Quality of Service (QoS, Pan et al., 2003), traffic engineering (Feldmann et al., 2000), accounting and billing (Duffield et al., 2001). Hence, flow size is at the basis of important network measurement tasks, including heavy-hitters and elephants detection (Tang et al., 2019; Ben Basat et al., 2017) or top-k flows identification (Yang et al., 2019). At first sight, recording the exact length of a flow only requires to maintain a counter. However, due to the high speed of modern networks, this approach does not scale to monitor millions of active flows, in terms of memory requirements in network devices. Instead of using exact counters or sampling techniques, approximate hash-based data structures (sketches) have been developed to estimate flow size at a reasonable memory cost, by sharing counters across flows.

Count-Min Sketch (CMS) is used to estimate frequency in a stream of positive items (Cormode and Muthukrishnan, 2005). A CMS is composed of L hash tables, also called rows, of W counters each, also called buckets. These counters are initialized at zero, and are addressed by L independent hash functions $\{h_i\}_{i=0}^{L-1}$. In the context of flow size estimation, one observes a stream of `<flowkey, value>` pairs, each item

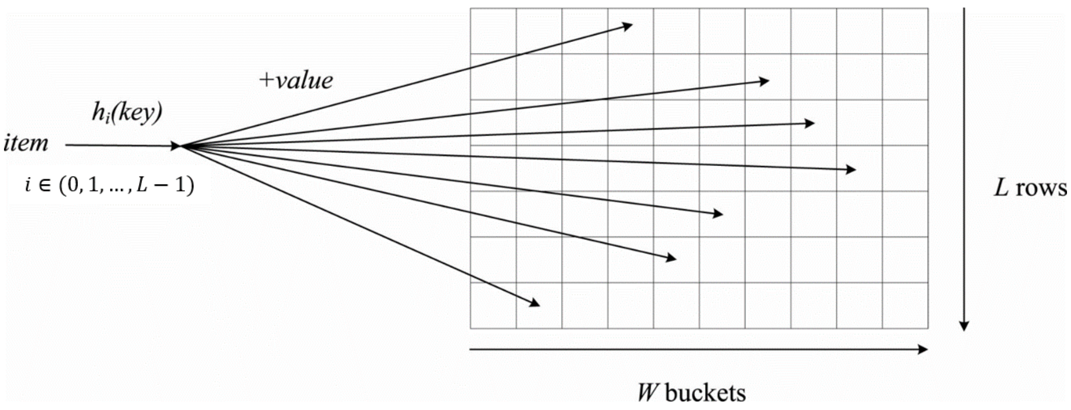


FIGURE 2.3: Count-Min Sketch insertion. Each item increments one counter in each row. Figure adapted from (Han et al., 2022)

indicating a new packet for a given flow. When estimating flow size in terms of packets, the `value` is set to 1, alternatively `value` may be set to the packet size. At each item arrival, the `flowkey` is hashed L times and the resulting hash codes are used as indexes to increment the corresponding counters by the `value`, as depicted in Figure 2.3. To estimate the size of a given flow, the CMS is queried by, first, hashing the `flowkey` L times to identify its corresponding counters, second, returning the minimum of the L counters thus identified. Hence, the CMS reduces the memory required to monitor the sizes of all the active flows by sharing counters across them. However, since the space used by the CMS is usually much smaller than the space required to represent the exact stream frequencies, there is an approximation: hashed keys collisions may produce an overestimation of the flow size. The estimation error depends on W and L, which represent the memory-accuracy trade-off of the sketch, and also depends on the actual frequencies in the stream.

More formally, let us consider a vector $\mathbf{s} = [s_1, ..., s_i, ..., s_n]$ that represents the exact frequencies observed so far in a stream of n active flows under monitoring. Then, considering a CMS with $L = \lceil \ln \frac{1}{\delta} \rceil$ rows composed of $W = \lceil \frac{e}{\epsilon} \rceil$ buckets, the estimated flow size $\hat{s}_i$ is bounded by[2]:

$$\hat{s}_i \leq s_i + \epsilon \|\mathbf{s}\|_1, \tag{2.1}$$

with probability at least $1 - \delta$, where $\|\mathbf{s}\|_1 = \sum_{i=1}^{n} |s_i|$. Thus, the fixed hyper-parameters δ and ϵ define the probability of error as well as the space and time requirements of the CMS. For instance, setting $\delta = \epsilon = 0.0001$ yields a CMS composed of $L = 10$ rows of $W = 27,183$ counters each, which requires $\approx$ 1MB when considering 4-byte counters. The CMS performs insertions and queries in $\mathcal{O}(1)$ time as it only depends on L that is fixed. In particular, this CMS keeps the frequency estimation error within 0.01% of the total count of all items, with a probability over 99.99%.

[2]Proof in (Cormode and Muthukrishnan, 2005).

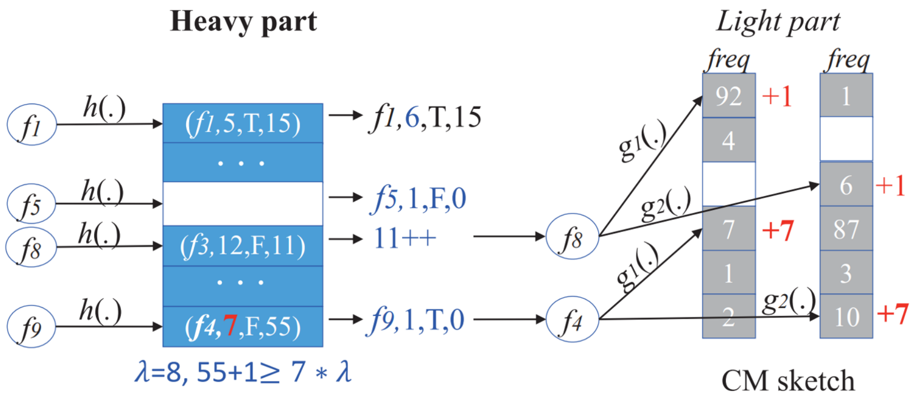


FIGURE 2.4: ElasticSketch insertions considering a threshold $\lambda = 8$ and a light part CMS with $L = 2$ rows. Flow f_4 gets ostracized from the heavy part. Figure from (Yang et al., 2018)

ElasticSketch (ES) is an adaptive data structure that improves frequency estimation accuracy over the plain CMS (Yang et al., 2018). In a CMS, hash collisions with high-count items may produce serious frequency over-estimations. For example, in the context of network measurements, a few large flows co-live with a majority of short flows, which may suffer from high relative estimation errors. ElasticSketch is based on the idea of segregating high-frequency items from the rest of the items stored in the CMS. By counting these high-frequency items separately, the CMS pollution is reduced to collisions among lower-frequency items, with lesser impact on the final estimation error. Therefore, an ES data structure is composed of two parts: a heavy part (e.g., a hash table with exact counters) and a light part (e.g., a CMS). Note that the heavy part can be dynamically sized in order to adapt to a varying and unknown number of high-frequency items in a data stream, hence the name "elastic". To separate items between these two parts, ES introduces an *ostracism* mechanism. The insertion of incoming items is first attempted in the heavy part. If the corresponding hash table bucket is empty or already occupied by their own key, the incoming item increments a counter of positive votes. Otherwise, if the bucket is already occupied by a different key, the incoming item increments a separate counter recording negative votes, and it is stored in the light part instead. When an incoming item's vote makes the negative counter exceed a threshold, it expels the current key to the light part and replaces it.

More formally, let us consider a heavy part $\mathcal{H}$ composed of a hash table of B buckets addressed with a hash function h. Each bucket of $\mathcal{H}$ stores four fields: the `flowkey`, a counter for positive votes (i.e., number of packets for this flow), a flag indicating if the light part contains positive votes for this flow, and a counter for negative votes (i.e., number of packets from other flows that collide to this bucket). Let us define an ostracism threshold λ. Additionally, let us consider a light part $\mathcal{G}$ that is a CMS composed of L rows of W counters each. The insertion procedure is depicted in Figure 2.4. To insert an incoming item `<flowkey, 1>`, first, its corresponding bucket $\mathcal{H}[h(\texttt{flowkey})]$ in the heavy part is selected. If the bucket is empty, we insert $(\texttt{flowkey}, 1, F, 0)$ into it, where the flag F (false) indicates that no eviction has happened yet. If the bucket is already occupied, let us denote its fields by $(\texttt{key}, \mathit{vote}^+, \mathit{flag}, \mathit{vote}^-)$. If $\texttt{flowkey} = \texttt{key}$, then vote^+ is incremented by 1. Otherwise, if $\frac{\mathit{vote}^-}{\mathit{vote}^+} < \lambda$ after incrementing vote^- by 1, then `<flowkey, 1>` is inserted in the CMS. If instead $\frac{\mathit{vote}^-}{\mathit{vote}^+} \geq \lambda$ after incrementing vote^- by 1, then the current `key` is ostracized: the `key` is evicted to the light part by incrementing the mapped L counters by vote^+, and the heavy bucket fields are set to $(\texttt{flowkey}, 1, T, 1)$. The flag T (true) is necessary to record that some packets from `flowkey` may have been previously inserted in the light part. At query time, if a flow is not present in the heavy part, its frequency is estimated from the CMS. However, if a flow is present in the heavy part, either its flag is set to false and its size is the corresponding vote^+ counter, or its flag is true and the number of packets vote^+ is added to the query result from the CMS. Following the notation introduced earlier for the CMS, ElasticSketch's frequency estimation $\hat{s}_i$ is bounded by[3]:

$$\hat{s}_i \leq s_i + \epsilon \|\mathbf{s}_{\mathcal{G}}\|_1 < s_i + \epsilon \|\mathbf{s}\|_1, \tag{2.2}$$

with probability at least $1 - \delta$, where $\mathbf{s}_{\mathcal{G}}$ denotes the sizes vector of the sub-stream that has been recorded by the light part.

2.1.3 Cardinality estimation

Cardinality refers to the number of distinct items in a data collection (Flajolet and Martin, 1985). In network measurements, host cardinality, also called spread, consists in counting the number of distinct hosts connected to a specific source or destination. Knowledge of cardinality distributions is particularly useful for service providers to derive communication patterns between hosts (Liu et al., 2016). More importantly, unusually large cardinalities indicate abnormal and suspicious network events, such as DDoS attacks (high source cardinality, Zhao et al., 2006) or network scanning from super-spreaders (high destination cardinality, Venkataraman et al., 2005). An exact method to count cardinality consists in maintaining a per-flow list of elements without replication and return its size. However, this trivial approach comes at a prohibitive cost in terms of memory requirements and number of accesses, which makes it impractical to compute cardinalities in network devices at high-speed.

To estimate cardinality at a small memory cost, Flajolet and Martin exploit probabilistic counting with the Flajolet-Martin (FM) algorithm in 1985, that probably constitutes the first sketch proposed. To estimate the cardinality of a multiset S, the FM algorithm relies on encoding each element x into a binary string $\mathit{hash}(x)$ of length L, with a sufficiently uniformly distributed hash function. If the values are uniformly

[3]Proof in (Yang et al., 2018).

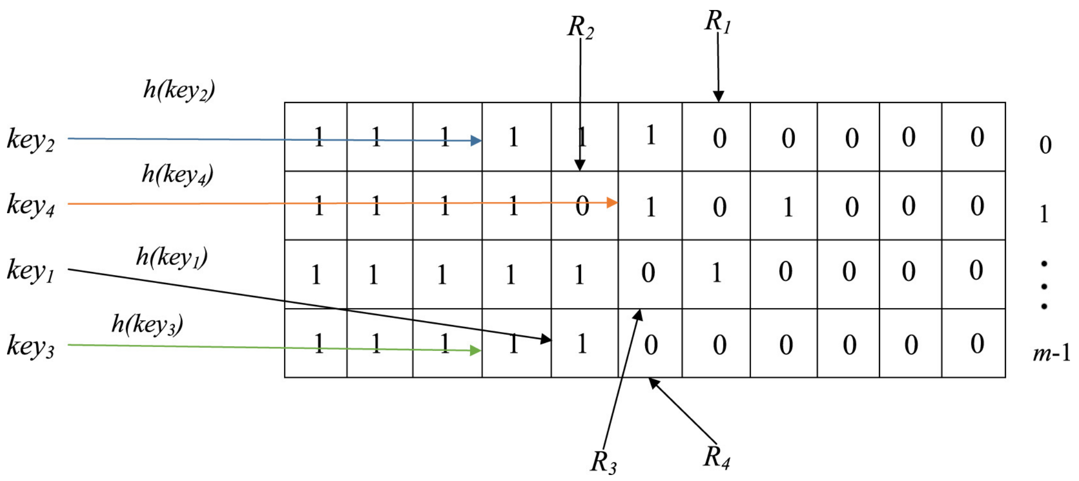


FIGURE 2.5: Flajolet-Martin Sketch with Probabilistic Averaging. Each item's trailing zeros pattern is recorded into one of m bitmaps. Figure from (Han et al., 2022)

distributed, the probability of observing k trailing zeros in the binary representation of $hash(x)$ is $1/2^{k+1}$. Intuitively, observing a large number of trailing zeroes is less likely and indicates a greater cardinality. Hence, by hashing all the elements contained in S, and recording the occurrences of trailing zeros patterns, one can estimate the number of distinct elements in the multiset S. Namely, if $Card(S) = n$, then sequences of k trailing zeros have been observed approximately $n/2^{k+1}$ times. More formally, when scanning S, let us record in a vector bitmap $B[0, ..., L-1]$ the position of the least significant 1-bit observed for each $hash(x)$. The FM algorithm uses the position of the leftmost zero in B, denoted R, as an indicator of $log_2(n)$.

However, as empirically shown by the authors, this estimate has a predictable bias towards larger cardinalities, which can be corrected by taking into account the expectation $\mathbb{E}(R)$ and standard deviation σ of R under the assumption of uniformly hashed values, i.e. $\mathbb{E}[R] \approx log_2(\varphi n)$ and $\sigma \approx 1.12$ where $\varphi \approx 0.77351$ is a correction factor[4]. Also, to address the variability of R, the elements are distributed across m bitmap vectors $\{B_i\}_{i=1}^{m}$, addressed by the same hash function. In details, for each incoming item x, its corresponding bitmap vector is selected by $\alpha = hash(x)mod(m)$ and is indexed with $hash(x)div(m)$. This FM sketch with stochastic averaging is depicted in Figure 2.5. The results $\{R_i\}_{i=1}^{m}$ from the m bitmaps are averaged into:

$$A = \frac{R_1 + R_2 + ... + R_m}{m}, \tag{2.3}$$

Finally, the cardinality of S is estimated by $n \approx \frac{m}{\varphi} 2^A$ with $\varphi/\sqrt{m}$ relative accuracy.

The LogLog algorithm (Durand and Flajolet, 2003) improves this idea with quotienting. The multiset S is separated into $m = 2^b$ subsets, such that the first b bits of $hash(x)$ are used to select the corresponding counter B_i, while the remaining $L - b$

[4]Proof in (Flajolet and Martin, 1985).

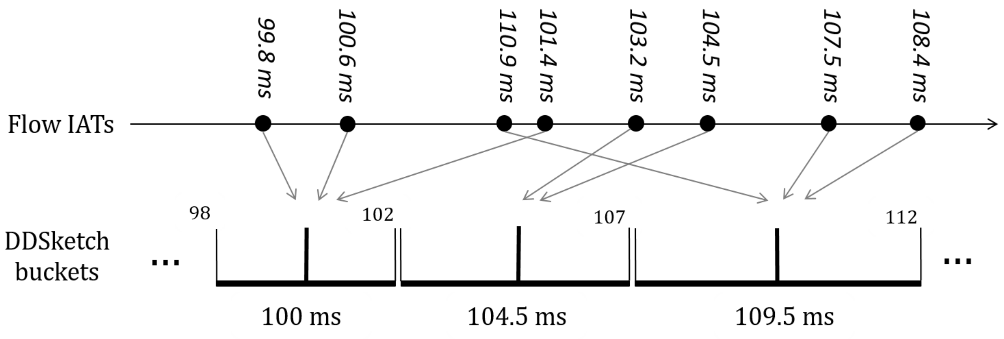


FIGURE 2.6: DDSketch insertions. Each value is mapped to a single bucket, whose range is maximized to maintain relative-error guarantees at the lowest memory footprint. Buckets bounds and representative values are rounded.

bits are used to estimate cardinality, i.e., to record in the counter the maximum number of trailing zeroes that have been observed. Finally, the HyperLogLog (HLL, Flajolet et al., 2007) further enhances this algorithm by discarding the largest 30% outliers among the m results from the counters, before averaging them with a normalized version of the harmonic mean. In this way, HLL reduces the relative error to $1.04/\sqrt{m}$ while maintaining constant $\mathcal{O}(m)$ time complexity at a memory cost in $\mathcal{O}(m \log_2 \log_2 n)$. For instance, hashing on $L = 32$ bits, setting $b = 11$ prefix bits, and maintaining $m = 2048$ buckets of 5-bit each (enough to record up to 32 trailing zeroes), produces a HLL sketch of 1.3 KB. This HLL can estimate cardinalities over 10^9 with a standard error of $\approx 0.02\%$. Note that, in the context of network monitoring, a dedicated HLL is allocated for each flow (e.g., each destination host) and its contacting hosts (e.g., its sources) constitutes the multiset S.

2.1.4 Quantiles estimation

Quantiles, sometimes referred to as percentiles when highlighting their partitioning in hundredths, are compact measures to represent the distribution of a collection of values. As a reminder, the q-quantile item x_q ($0 \leq q \leq 1$) of a multiset S of n real items is defined as the item $x \in S$ whose rank $R(x)$ is $\lfloor 1 + q(n-1) \rfloor$ in the sorted multiset. Particular quantiles include the minimum ($q = 0$), the median ($q = 0.5$) and the maximum ($q = 1$). In network measurements, quantiles are particularly useful to express QoS metrics. For example, one typical metric of interest is latency, captured e.g., through the distribution of flow Inter-Arrival Times (IATs) or Round-Trip Times (RTTs). Latency distributions are typically skewed, with the highest quantiles being the most critical. As a consequence, instead of traditional aggregates such as the average, network operators are more interested by the extreme latencies experienced in the network, e.g., computing the 95th percentile of IATs to represent the 5% cases with the worst latency. Unfortunately, computing exact quantiles is challenging as it requires to store and sort *all* the values in the multiset (Cranor et al.,

Algorithm 1 : Quantile(q). From (Masson et al., 2019)

```
Require: $0 \leq q \leq 1$
  $i_0 \leftarrow \min(\{j : B_j > 0\})$;
  $count \leftarrow B_{i_0}$;
  $i \leftarrow i_0$;
  while $count \leq q(n-1)$ do
      $i \leftarrow \min(\{j : B_j > 0 \wedge j > i\})$;
      $count \leftarrow count + B_i$;
  end while
  return $2\gamma^i/(\gamma+1)$;
```

2003). Thus, this would translate into expensive memory requirements if the full set of IATs needs to be retained to perform the exact computation.

The Distributed Distribution Sketch (DDSketch) is a state-of-the-art lossy data structure employed to estimate the quantiles of a stream of positive real values (Masson et al., 2019). Instead of computing exact quantiles, the DDSketch stores approximate values and sort these approximate values to compute approximate quantiles. Concretely, a DDSketch is an array of buckets of fixed length, that divide the universe of possible values into ranges. At ingestion, each bucket keeps tracks of the values falling within its range by incrementing a counter. Each bucket defines a representative value to speak for the range it covers, as depicted in Figure 2.6. To estimate a given quantile, the counters are summed up in order, until the bucket containing the quantile value is found and its representative value is returned. Note that a DDSketch is allocated on a per-flow basis, e.g., a single DDSketch would estimate the IATs quantiles of a single flow.

Unlike other quantiles summaries data structures that provide rank-error guarantees (e.g., the GK Sketch Greenwald and Khanna, 2001), the DDSketch guarantees relative-error bounds. For example, a rank-error guarantee of 0.02 implies that the estimated 95th percentile is between the actual 93th and 97th percentile. Instead, a relative-error guarantee of 0.02 implies that the estimate lies within $\pm$ 2% of the actual percentile *value*. DDSketch enforces these guarantees by defining buckets whose relative width is twice the relative error. More formally, to design a DDSketch that guarantees an α-accurate estimation $\hat{x}_q$ of an actual q-quantile x_q, i.e., $|\hat{x}_q - x_q| \leq \alpha x_q$, let us first define:

$$\gamma = (1+\alpha)/(1-\alpha), \tag{2.4}$$

Each bucket B_i, indexed by $i \in \mathbb{Z}$, counts the number of values falling in $]\gamma^{i-1}, \gamma^i]$. When an incoming value x is observed, it is assigned to the bucket indexed by $\lceil \log_\gamma(x) \rceil$. To estimate a q-quantile[5], the DDSketch is queried using Algorithm 1. In practice, the number of buckets m is not unbounded and is instead limited according

[5]Proof of α relative accuracy in (Masson et al., 2019).

to the range of values that the DDSketch must cover. This is implemented by collapsing the buckets outside of the monitored range, which guarantees logarithmic space complexity for non-degenerate data (Masson et al., 2019). For instance, setting $\alpha = 0.02$ for a 2% relative error, a DDSketch covering the IATs range from 1 millisecond to 1 minute only requires 275 buckets, which translate to 2.2 KB with 4-byte counters. Interestingly, covering the IAT range from 1 nanosecond to 1 day with the same relative error only requires 802 buckets, i.e., 6.4 KB, that is a $\approx 3\times$ increase in memory. Besides, DDSketch answers queries in constant $\mathcal{O}(m)$ time. Finally, the DDSketch is particularly popular in network measurements because of its flexibility, as DDSketches are mergeable and offers various implementations options.

2.2 Machine Learning Models

Some of the methodologies proposed in this thesis rely on Machine Learning models to automate pattern-matching and features extraction from network traffic. In this section, we detail the paradigms and algorithms used throughout the thesis.

2.2.1 Random Forests

Random Forests (RFs) represent a popular supervised learning procedure based on the aggregation of multiple decision trees predictors. They have been theoretically characterized in (Breiman, 2001) after several earlier works (Amit and Geman, 1997; Ho, 1998; Dietterich, 2000). In a nutshell, a RF is an ensemble of several decision trees, each trained on a randomized fraction of the data, which are then aggregated to produce the final RF prediction. Beside the simplicity of their bagging approaches, RFs are recognized for their ability to handle high-dimensional features with minimal pre-processing, for their interpretability and for their parallelization (Biau and Scornet, 2016). Although several flavors of RFs have been proposed, the bootstrap-aggregating or bagging paradigm is at the core of the algorithm.

To introduce the RF algorithm more formally, let us consider a binary supervised classification task. Given an observed random variable $\mathbf{X} \in \mathbb{R}^p$, the goal is to predict the response random variable $\mathbf{Y}$ that takes values in $\{0, 1\}$. We have access to pairs of samples drawn from the random variables $\mathbf{X}$ and $\mathbf{Y}$, that constitute the training dataset $\mathcal{D}_n = ((\mathbf{X} = \mathbf{x_1}, \mathbf{Y} = \mathbf{y_1}), ..., (\mathbf{X} = \mathbf{x_n}, \mathbf{Y} = \mathbf{y_n}))$. A classifier f_n is a function of $\mathbf{X}$ and $\mathcal{D}_n$ that attempts to estimate the label $\mathbf{Y}$. For example, a classification decision tree recursively partitions the feature set into subsets, such that samples with the same label are grouped together in successor children. This partitioning is implemented by if-then-else rules based on input features and guided by a splitting criterion such as Gini impurity minimization (Breiman, 2017) or information gain maximization (Quinlan, 1986). The recursion goes on until a stopping criterion is met (e.g., the node is pure).

A particular trait of decision trees is that the trees grown very deep tend to overfit their training set. Random Forests aim at reducing this high variance by introducing randomness in the construction of the classifier. The first source of randomness comes from the aggregation of several decision trees trained on various subsets of the data set $\mathcal{D}_n$. This bootstrap sampling aims at de-correlating the individual trees, such that their average is less sensitive to noise in the training data. In this context, a RF is a ensemble of m randomized classification trees $h_1, ..., h_m$, each trained on a random sample drawn with replacement from $\mathcal{D}_n$. In the original RF algorithm, the final classification rule for an example $\mathbf{x}$ is obtained by a majority vote among the individual trees (Biau and Scornet, 2016):

$$RF(\mathbf{x}; \mathcal{D}_n) = f_n(\mathbf{x}; h_1, ..., h_m, \mathcal{D}_n) = \begin{cases} 1 \text{ if } \frac{1}{m} \sum_{j=1}^{m} h_j(\mathbf{x}; \mathcal{D}_n) > \frac{1}{2} \\ 0 \text{ otherwise} \end{cases} \tag{2.5}$$

Alternative aggregations techniques have been proposed, such as the weighted average of the individual trees class probability prediction, which is implemented in the popular Scikit-Learn Python package (Pedregosa et al., 2011). To further decrease the variance of RFs estimators, feature bagging introduces an additional source of randomness by considering only a random subset of the features at each candidate split node. This process also aims at de-correlating the m individual trees that compose the ensemble. Note that Random Forest algorithms are sensitive to several hyper-parameters, including the number of trees m, the number of features to consider at each candidate split or the minimum number of examples required to perform a split. These parameters are typically cross-validated to optimize a success metric, e.g., accuracy in a balanced classification setting. Finally, alternative tree-based ensembles have been developed, such as Gradient-Boosted Decision Trees algorithms (Chen and Guestrin, 2016) which iteratively fit weak learners to minimize a cost function by predicting its negative gradients.

Decision trees and Random Forests have been of particular interest for in-network Machine Learning implementation. Their simplicity and parallelization make them attractive candidates for deployment with low-resources consumption. Indeed, at inference, the trees composing the forest can be independently queried in parallel to speed up prediction time. For example, programmable switches' Match-Action pipelines have been adapted to implement decision trees traversals in (Xiong and Zilberman, 2019; Busse-Grawitz et al., 2022; Zheng and Zilberman, 2021). Last, RFs implementation in re-configurable hardware is particularly appealing, e.g., to deal with frequent model updates (Owaida et al., 2017).

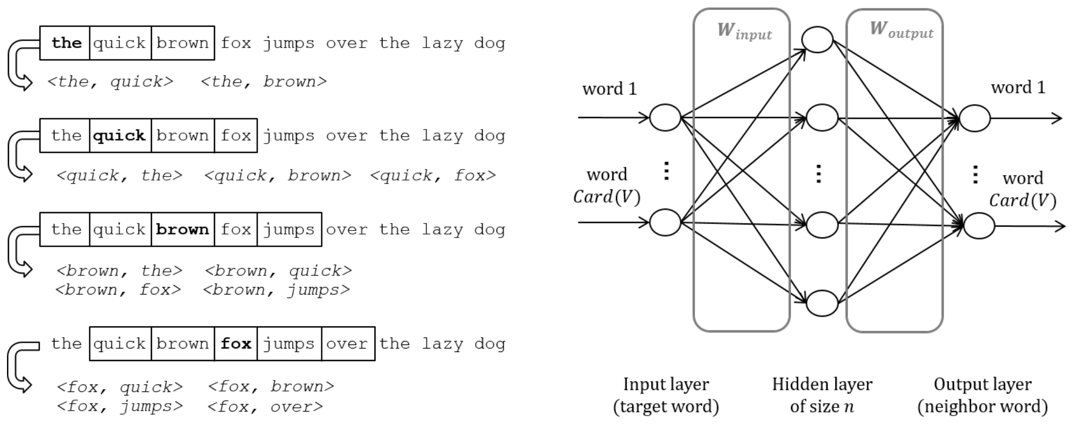

the quick brown fox jumps over the lazy dog
<the, quick> <the, brown>
the quick brown fox jumps over the lazy dog
<quick, the> <quick, brown> <quick, fox>
the quick brown fox jumps over the lazy dog
<brown, the> <brown, quick>
<brown, fox> <brown, jumps>
the quick brown fox jumps over the lazy dog
<fox, quick> <fox, brown>
<fox, jumps> <fox, over>
W_{input}
W_{output}
word 1
word $Card(V)$
word 1
word $Card(V)$
Input layer (target word)
Hidden layer of size n
Output layer (neighbor word)

FIGURE 2.7: **(left)** Skip-Gram samples with a context window of size $c = 2$. **(right)** Word2Vec architecture with a vocabulary V.

2.2.2 Word2Vec

Word2Vec (W2V) corresponds to a Deep Learning methodology introduced in the context of Natural Language Processing by (Mikolov et al., 2013b) to extract vector representations of words. The W2V algorithm captures syntactic and semantics words relationships based on their co-occurrences in sequences of text, i.e., phrases. W2V refers a family of to two-layer Artificial Neural Networks (ANNs) instrumented to model natural language. These models take as input a corpus of text and output a vector space in which each unique word of the corpus is associated to a particular vector. Word2Vec training objective is to learn word representations that facilitates the prediction of nearby words. The approach comes in two flavors, Skip-Gram or Continuous Bag-of-Words, that both consider individual words and their context, materialized by a sliding window. During training, the model is tasked to predict neighboring words from the current word (or the opposite). This objective aims at embedding words that share common context (i.e., that co-occur frequently) into vectors that are close in the output space.

More formally, let us consider a vocabulary V which is the set of words present in a text corpus. Word2Vec aims at learning a vector $v_w \in \mathbb{R}^n$ for each word w in the vocabulary. In this section, we only detail the Skip-Gram approach. Given a sequence of words $w_1, w_2, ..., w_T$, the training objective is for each *target* word to maximize the probability of predicting the set of *neighbor* words in a context window of size c, i.e., maximizing the average log probability (Mikolov et al., 2013b):

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} \log P(w_{t+j}|w_t), \tag{2.6}$$

In this approach, Skip-Gram training samples are pairs of target-neighbor words $< w_i, w_j >$ extracted from the corpus according to the sliding window's size as illustrated in Figure 2.7 (left), and words themselves are one-hot encoded. A W2V model, as depicted in Figure 2.7 (right), is a neural network whose architecture resembles an Auto-Encoder with a single hidden layer. It is composed of an input layer of size $Card(V)$, one fully connected layer (without activation) of size n, represented by its weights matrix $\mathbf{W}_{input}$, and one output layer of size $Card(V)$ with its weights matrix $\mathbf{W}_{output}$. Given that each word is one-hot encoded, the rows of $\mathbf{W}_{input}$ act as a look-up table and constitute the word vectors, or word embeddings, that are learned throughout training via back-propagation. The probability $P(neighbor|target)$ is defined using the softmax function on the output layer, i.e., similar to a classification layer:

$$P(neighbor|target) = \frac{exp(\mathbf{v'}_{neighbor}^{\top}\, \mathbf{v}_{target})}{\sum_{w=1}^{Card(V)} exp(\mathbf{v'}_{w}^{\top}\, \mathbf{v}_{target})}, \tag{2.7}$$

where $\mathbf{v}_w \in \mathbb{R}^n$ is the vector representation of word w in the input layer, and $\mathbf{v'}_w \in \mathbb{R}^n$ is its output layer representation. Instead of optimizing all weights for the precise Skip-Gram objective presented above, Word2Vec typically approximately maximizes it, using negative sampling for computational efficiency (Mikolov et al., 2013b).

The Word2Vec learning procedure depends on various parameters, namely the dimension of the word vector space (n) and the size of the context window (c). Increasing the dimension n of the vector space translates into higher word embeddings quality, although diminishing returns have been reported after some point (Mikolov et al., 2013b). Larger context windows produce more training examples, which can result in higher word embeddings quality, but at a higher training cost. We note that the Word2Vec methodology has already been applied to networking, e.g., with IP2Vec (Ring et al., 2017) or DarkVec (Gioacchini et al., 2021) to learn IP addresses embeddings.

2.2.3 Multi-Task Learning

In Deep Learning, Multi-Task Learning (MTL) is a paradigm that seeks to improve generalization performance by learning multiple *related* tasks simultaneously (Caruana, 1997). MTL is based on the assumption that the domain-specific information of related tasks can be leveraged by learning them in parallel and sharing common representations. In the traditional Single-Task Learning (STL) approach, an inductive learner is provided with training data for a specific task and a space of hypotheses from which it must select (i.e., learn) the one hypothesis that minimizes error over the underlying distribution of the data (i.e., generalize). During learning, any basis for the learner to prefer some hypothesis over others is referred to as an inductive bias (Mitchell, 1980). MTL argues that the training signals of related tasks can be used as an inductive bias, causing the learner to favor hypotheses that solve more than a single task (Caruana, 1997). During MTL training, the error from several tasks is aggregated and back-propagated to update hidden representations, hence developing features that would not have been extracted in a STL approach. As such, Multi-Task Learning constitutes an inductive transfer mechanism between

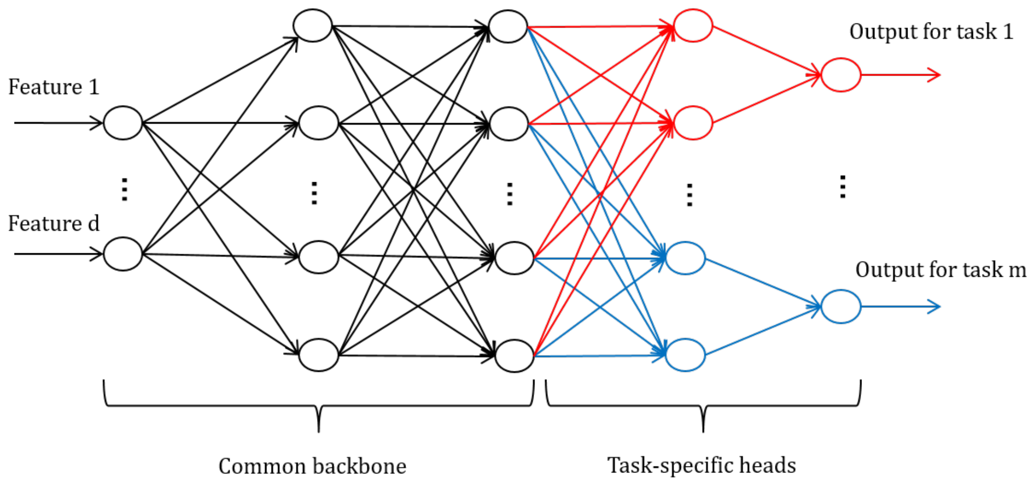


FIGURE 2.8: Multi-Task Feed-Forward Neural Network. Representations are shared across tasks in the model backbone, but private in the task-specific heads. Figure adapted from (Zhang and Yang, 2022)

tasks. However, in practice, tasks interference may degrade performance if their respective updates become unaligned or contradictory during simultaneous learning. This negative transfer phenomenon constitutes a key challenge in MTL, which is often observed to make single-task models superior without appropriate mitigation mechanisms (Standley et al., 2020). The Multi-Task Learning paradigm is also appealing in terms of cost-performance trade-off. Given a set of tasks to solve and a limited computational budget for training and inference, MTL offers a flexible way to assign tasks to models in order to maximize the overall tasks set performance.

The study of Multi-Task Learning comes into various flavors, that may be categorized under three main questions: (i) *what to share*, (ii) *when to share*, and (iii) *how to share* (Zhang and Yang, 2022). First, (i) concerns the object of knowledge sharing, which can be feature-based, instance-based or parameter-based. Feature-based sharing focuses on extracting higher-level representations of the input, that are common to several tasks. This approach corresponds to the initial feed-forward MTL architecture proposed in (Caruana, 1997) that shares hidden representations across multiple related tasks. In addition, instance-based MTL modulates knowledge sharing by identifying the relevant training instances to be shared on a task basis. For example, when few samples are available for a task but a related task has more samples at its disposal, re-sampling weights may reflect this discrepancy for fair MTL training (Bickel et al., 2008). Finally, parameter-based MTL deals with learning relations across tasks, by instrumenting the parameters of task-specific models. For instance, one may quantify tasks relatedness by characterizing the relationships between STL models parameters(Ando et al., 2005) or by clustering tasks (Thrun and O'Sullivan, 1996). In a learning problem involving several tasks, (ii) corresponds to modeling choices from a task perspective, deciding which tasks should be learned together (MTL) and which tasks should be learned alone (STL), based on their affinity. This partitioning of the task space to group beneficial tasks together and separate competing tasks apart is also referred to as task grouping (Fifty et al., 2021). Last, (iii) deals with the actual training process to share knowledge among related tasks, implemented through model design. In this thesis, we study homogeneous feature-based supervised MTL, i.e., learning hidden representations extracted from the same input, and common to multiple related tasks. More formally, we consider a set of m tasks $\mathcal{T} = \{t_1, t_2, \cdots, t_m\}$ and a dataset of N instances $\mathcal{D}_N = \{\mathbf{x}_i, \mathbf{y}_i^j\}_{i=1}^N$, where $\mathbf{y}_i^j$ corresponds to the label of the i^{th} instance for task t_j. In the remainder of this subsection, we present the main MTL approaches falling into this category.

Feature-based MTL assumes that related tasks benefit from similar features. Hence, by sharing the knowledge extraction process across tasks, internal representations learned for one task may be used by other tasks in the hidden layers (Caruana, 1997). A straightforward implementation to concretize this approach consists in adding several tasks (i.e., several outputs) to a back-propagation neural network. As depicted in Figure 2.8, such model is composed of a shared backbone, whose final bottleneck layer serves as a feature map plugged to multiple per-task heads. Note that while this architecture is illustrated with simple feed-forward layers, the backbone may contain more sophisticated layers, e.g., Convolutional Neural Networks. The loss function to optimize is typically a linear combination of the individual tasks losses, i.e.:

$$\mathcal{L} = \sum_{j=1}^{m} \alpha_j \mathcal{L}_j(\mathbf{X}, \mathbf{Y}^j, \mathbf{W}_B, \mathbf{W}_j), \tag{2.8}$$

where α_j and $\mathcal{L}_j$ denote the coefficient and loss function used for task t_j resp., where $\mathbf{X}$ denotes the input samples with corresponding labels $\mathbf{Y}^j$ for task t_j, and where $\mathbf{W}_B$ and $\mathbf{W}_j$ are the model backbone and head weights for task t_j resp. Therefore, during the backward pass, the task-specific error is back-propagated in each particular head, while the combined error is back-propagated throughout the shared backbone, hence sharing training signals among tasks. Additionally, several works proposed more sophisticated loss functions to control tasks interactions dynamically throughout training (Leang et al., 2020; Pascal et al., 2021). A more flexible feature-based MTL approach considers that tasks require related but different representations. For example, the Cross-Stitch architecture (Misra et al., 2016) duplicates the model backbone per-task, and defines feature-level relationships across tasks, materialized by learnable parameters. These parameters are used to tune interactions between task-specific representations during training.

Chapter 3

Sparse Sketches Representations for Per-flow Monitoring

In this chapter, we propose a sparse representation to improve the memory footprint of *traditional measurements* systems based on sketches. Accurate per-flow monitoring is critical for precise network diagnosis, performance analysis, and network operation and management in general. However, the limited amount of memory available on modern programmable devices and the large number of active flows force practitioners to monitor only the most relevant flows with approximate data structures, limiting their view of network traffic. We argue that, due to the skewed nature of network traffic, such data structures are, in practice, heavily underutilized, i.e., sparse, thus wasting a significant amount of memory. This chapter proposes a Sparse Approximate Data Structure (SPADA) representation that leverages sketches' sparsity to reduce the memory footprint of per-flow monitoring systems in the data plane while preserving their original accuracy. SPADA representations can be integrated into a generic per-flow monitoring system and are suitable for several measurement use cases. We test our approach on four real network traces over three different monitoring tasks. Our results show that SPADA achieves $2\times$ to $11\times$ memory footprint reduction with respect to the state-of-the-art while maintaining the same accuracy, or even improving it.

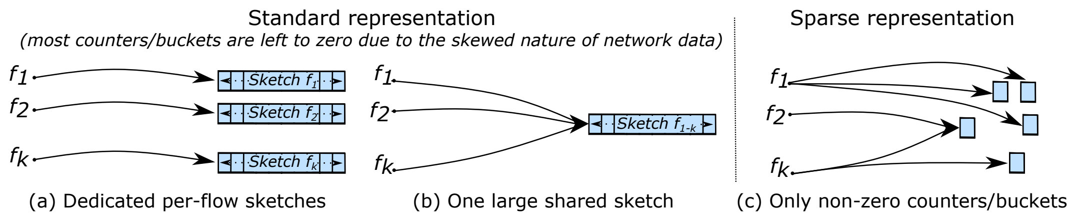
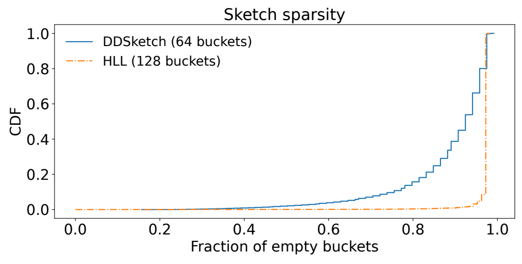

Standard representation
(most counters/buckets are left to zero due to the skewed nature of network data)
Sparse representation
f1
f2
fk
Sketch f1
Sketch f2
Sketch fk
Sketch f1-k
(a) Dedicated per-flow sketches
(b) One large shared sketch
(c) Only non-zero counters/buckets

FIGURE 3.1: Per-flow monitoring using standard **(a)**, **(b)** and sparse **(c)** representations.

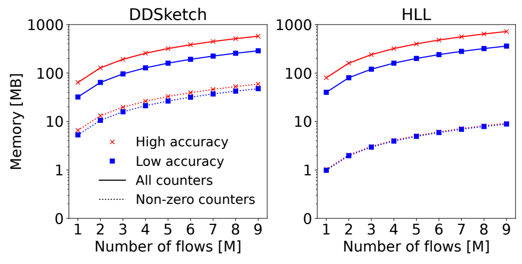
3.1 Introduction

Monitoring network traffic on a per-flow basis requires measuring several quantities related to the packets traversing network devices. These accurate measures provide network operators the necessary data for fine-grained Operations, Administration, and Management (OAM) algorithms such as responsive diagnosis (Schlinker et al., 2019; Chen et al., 2016), precise fault localization (Li et al., 2016b; Arzani et al., 2018), traffic engineering (Benson et al., 2011), network accounting (Estan and Varghese, 2003), anomaly detection (Zhang, 2013), and many others. However, collecting per-flow metrics requires a considerable amount of resources, especially at high speed when the number of active flows might be very high. Considering that recent studies estimate the number of active flows in the order of 100K per Gbps of traffic (Scazzariello et al., 2023), accurately monitoring Tbps of traffic might require several GBs for a few per-flow metrics. Programmable ASIC devices feature memories in the order of dozens of MBs to accommodate all network applications, including L2/L3 forwarding, among others. Consequently, the amount of memory allocated for monitoring is but a fraction of the total one, making accurate per-flow monitoring impractical due to its high memory requirements. Similarly, in FPGA-based SmartNICs, the amount of memory available for per-flow monitoring is scarce since the use of large memories such as Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM) or High Bandwidth Memory (HBM) is limited by their high access latency, i.e., tens of clock cycles, and the required resource-hungry cache memory hierarchy.

To better understand the per-flow monitoring problem, Figure 3.1ab illustrates at a high level how flows f_{1-K} are typically mapped to dedicated or shared arrays of counters (sketches). In particular, we consider a few concrete use cases: *super spreader detection*, *per-flow quantiles*, and *flow size* estimation. The goal of super spreader detection algorithms (Jia et al., 2020; Wang et al., 2021; Han et al., 2022) is to estimate, for any given source IP (sIP), its "cardinality", i.e., the number of destination IPs (dIPs) it contacts. This task is often achieved using the HyperLogLog (HLL) (Flajolet et al., 2007) sketch, allocating one HLL data structure for each sIP. Unfortunately, achieving good accuracy with HLL requires a considerable amount of memory. For this reason, practitioners usually limit the number of monitored sIP to reduce memory occupancy, or decrease the accuracy, e.g., vHLL (Jia et al., 2020; Xiao et al., 2015).



(A) Sketch counters sparsity.



(B) Memory requirement.

FIGURE 3.2: Flow sparsity and memory requirements analysis using one sketch per flow from a CAIDA trace.

Similarly, monitoring per-flow quantiles (Choi et al., 2007; Ivkin et al., 2019) of relevant flow properties, e.g., packet Inter-Arrival Time (IAT), packet size, etc., requires processing the stream of all packets belonging to the same unidirectional 5-tuple using histogram-based data structures such as DDSketch (Masson et al., 2019). However, using a dedicated sketch for each monitored flow makes quantile estimation challenging due to the high memory requirements. It is worth noting that even for basic measurements such as flow size estimation, the amount of memory required to monitor hundreds of thousands of flows is not negligible with ElasticSketch (Yang et al., 2018). We remark that the above-mentioned monitoring algorithms, as well as other existing ones (Lall et al., 2006; Chen et al., 2020; Song et al., 2020; Hartmann and Schlossnagle, 2020; Cormode and Garofalakis, 2005; Papapetrou et al., 2015; Arasu and Manku, 2004), require the allocation of a sketch composed by an array of shared or dedicated counters (also called buckets, bins) for each monitored flow (cf. Figure 3.1ab): this frequently translates into significant memory requirements making it difficult to deploy them even for only a fraction of the active flows. Therefore, we assert the need to reduce the memory requirements of per-flow network monitoring algorithms, enabling them to coexist with other critical network functions in the data plane.

Contributions. In this chapter, we show that sizeable gains can be attained by *reducing the amount of unnecessarily allocated memory without compromising the monitoring accuracy*. We remark that per-flow sketches are designed for extreme scenarios and hence underutilized in most cases. To solve this problem, we propose a data structure representation built around the simple concept of only storing counters that are actually used, as depicted in Figure 3.1c. To provide a quantitative idea of the under-utilization of sketches in typical measurement tasks, Figure 3.2a reports the CDF of the fraction of empty buckets in high accuracy DDSketch and HLL (64 and 128 buckets) on a CAIDA trace for 700K flows. In both cases, per-flow sketches are highly underutilized, i.e., 80% of per-flow DDSketches feature at least 80% of empty buckets, even more for HLL. This results in a huge waste of memory as observed in Figure 3.2b, which contrasts the memory occupied by the sketches with the one required by only non-zero counters: the picture shows one-to-two orders of magnitudes of potential memory savings, which holds even for low accuracy DDSketch and HLL (32 and 64 buckets). We remark that similar sparsity issues arise in different use cases where sketch-based data structures are employed, (Lall et al., 2006; Chen et al., 2020; Song et al., 2020; Yang et al., 2018; Hartmann and Schlossnagle, 2020; Cormode and Garofalakis, 2005; Papapetrou et al., 2015; Arasu and Manku, 2004). Thus, it can be argued that sparsity is a property common to many measurement tasks.

Sparse monitoring data structure representations are not trivial to implement in networking because it is not possible to know a priori which flow will lead to sparse data. We acknowledge that sparse data representation is a well-known technique widely used in many fields ranging from signal processing to machine learning (Reginald P., 1973; Yuster and Zwick, 2005; Zhang et al., 2015; Ediger et al., 2012; Winter et al., 2017; Busato et al., 2018) – yet, to the best of our knowledge, it has never been explored in the context of network monitoring. To support sparse monitoring data structures in the data plane we propose a Sparse Approximate Data Structure (SPADA) representation, which stores only relevant, i.e., non-zero, sketch counters. One key challenge of implementing sparse representations in the data plane is storing `<key, value>` pairs for the non-zero counters, taking into account hardware limitations and without a priori knowledge of flow sparsity. To address this challenge, we propose two alternative solutions: *(i)* a qCHT, which can accommodate any kind of data at the price of non-constant insertion time, and *(ii)* a novel data structure, pIBLT, featuring constant insertion time but only suitable for counter-based sketches. Our main contributions are as follows:

1. we introduce a sparse representation in the context of a generic data plane monitoring system which is beneficial for several use cases by reducing memory footprint with no accuracy penalty;
2. we design a novel data structure pIBLT, which improves over a traditional IBLT removing false positives at the cost of a small bitmap;
3. we provide simulation code and results with the CAIDA and MAWI datasets on three use cases, assessing memory reduction from $2\times$ to over $11\times$ with respect to the state-of-the-art;

Organization. In Section 3.2, we review state-of-the-art sketches that can benefit from SPADA and detail three use cases we use as examples throughout the chapter. In Section 3.3, we introduce the SPADA representation within a standard per-flow monitoring system and provide its memory occupancy analytical model. Then, we discuss SPADA memory sizing in detail in Section 3.4. Trace-based simulation results are reported in Section 3.5. Finally, Section 3.6 discusses the related work, and Section 3.7 concludes the chapter.

Measure	**Sketch name**	**Description**	**Note**
Cardinality (use case ⓐ)	HLL (Flajolet et al., 2007) ✓	Array of counters	Skewness similar to the HLL sketch used as reference for use case ⓐ.
	PCSA (Flajolet et al., 1985)	Array of bit-vectors	
	KVM (Bar-Yossef et al., 2002)	Stores up to k minimum values	
	Fast-AGMS (Cormode et al., 2005)	Array of counters	
	BeauCoup (Chen et al., 2020)	Bitvector	
Quantile (use case ⓑ)	DDSketch (Masson et al., 2019) ✓	Array of counters	Skewness similar to DDsketch used as reference for use case ⓑ. High-level KLL compactors not fully allocated.
	Circllhist (Hartmann et al., 2020)	Array of counters	
	KLL (Karnin et al., 2016)	Array of compactors	
Flow Size (use case ⓒ)	Count-Min Sketch (Cormode et al., 2005)	Array of counters	Sparse when high accuracy is needed.
	ECM-sketches (Papapetrou et al., 2015)	Count-min sketch over sliding windows	
	ElasticSketch (Yang et al., 2018) ✓	Split heavy and light flows	Similar to CMS, the light part can be sparsified for high accuracy.
	LearnedSketch (Hsu et al., 2019)	Split heavy and light flows with ML	
Entropy	EntropySketch (Lall et al., 2006)	Up to k counters for stream entropy estimation.	

TABLE 3.1: Sketches that feature sparse data. Marked (✓) ones are used as reference in our analysis.

3.2 Background

In this section, we first identify a set of sparse monitoring data structures, then we detail state-of-the-art sketches used for three use cases: ⓐ super spreader detection with HLL, ⓑ per-flow packet IAT distribution with DDSketch, and ⓒ flow size estimation with ElasticSketch. Given their popularity and generality, we use them to showcase the benefits of SPADA throughout the rest of the chapter.

3.2.1 Sparse monitoring data structures

We report in Table 3.1 a list of sketches that can exploit sparsity. Note that the actual benefit of the sparse representation depends on several factors, such as the number of flows under monitoring, the required accuracy, and the traffic skewness. Generally speaking, a sparse representation of a sketch can be beneficial when it is allocated per-flow (i.e., a sketch for each monitored flow) since, due to the natural skewness of traffic, many sketches will be significantly sparse as motivated before (e.g., use cases ⓐ, ⓑ). Another scenario in which a sparse representation is useful is when the sketch must provide high accuracy and thus reduce the collision probability (e.g., use case ⓒ) by increasing the sketch size. We note that with appropriate sampling strategies (Ben Basat et al., 2020b), any sketch can be sparsified if a small loss in accuracy is acceptable.

3.2.2 Monitoring use cases

ⓐ **Super spreader detection.** HLL (Flajolet et al., 2007) is a data structure for cardinality estimation based on the probabilistic counting method developed in (Flajolet and Martin, 1985) and already introduced in Chapter 2. On the one hand, using a large number of HLL counters m reduces the estimation error. On the other hand, the use of many counters leads to sparse HLLs, as the number of distinct flow counters updated is proportional to the cardinality itself, which is typically small. For example, on a CAIDA trace, less than 10% of sIP have a cardinality higher than 3 (90% of the HLLs only use three out of the m allocated counters). However, knowing in advance which flows will have high cardinality is challenging. Given this large number of unused counters, moving to a sparse HLL representation can significantly reduce the memory footprint of a super spreader detection system.

ⓑ **Packet IAT distribution.** DDSketch (Masson et al., 2019) is a data structure used to estimate the quantiles for a set of real positive values as presented in Chapter 2. DDSketch can be modified to accept a limited number of bins m depending on the desired accuracy α. It is worth highlighting that in per-flow IAT monitoring, most DDSketch counters are left to zero, as most flows consist of only a few packets, and samples, e.g., IAT, tend to cluster around a few values. Thus, an efficient sparse DDSketch representation would significantly reduce its memory requirements.

ⓒ **Flow size estimation.** ElasticSketch (Yang et al., 2018) is among the state-of-the-art for flow size estimation. It comprises a *heavy* and a *light* part, storing elephant and mouse flows respectively, as introduced in Chapter 2. The heavy part comprises multiple hash tables with per-flow counters, while the light part is a Count-Min Sketch (CMS, Cormode and Muthukrishnan, 2005). One of the effects of segregating elephants in the heavy part is that the CMS can be composed of a single row ($d = 1$) of 8-bit counters, thanks to the reduced number of flows and their size. To achieve good accuracy, the CMS still needs to be dimensioned to keep the amount of collisions considerably low. This means budgeting a large number of counters, even though most of them are left to zero, making the CMS sparse.

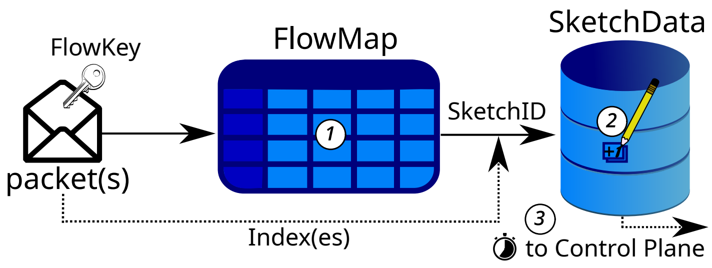


FIGURE 3.3: Generic data plane monitoring system.

3.3 Per-flow Monitoring System Design

We provide a high-level overview of a generic per-flow monitoring data plane pipeline depicted in Figure 3.3. First ①, a flow key (e.g., IP address or 5-tuple) is extracted at packet arrival and mapped to a `SketchID` via a *Flow Map* (FM). Second ②, the sketch counters corresponding to that flow are updated in a *Sketch Data* (SD) store. Note that a wide set of network monitoring systems (Yang et al., 2018; Chen et al., 2020; Xiao et al., 2015; Ben Basat et al., 2020a; Zhao et al., 2021) can be cast into this two-stage monitoring system data plane. Finally ③, the measurements are exported or used locally for traffic management: as the focus of this section is on the design and implementation of the data plane, the control plane is limited to the collection of sketches at the end of a measurement epoch.

SPADA in a nutshell. A SPADA representation consists of a compression of the *Sketch Data* by exploiting its sparsity. To showcase the generality of SPADA, we consider three popular measurement use cases and two alternative SPADA configurations summarized in Table 3.2 (right). Considered use cases are ⓐ super spreader detection, ⓑ per-flow packet IAT quantile estimation, and ⓒ flow size estimation, which can be performed using state-of-the-art sketches such as HLL (Flajolet et al., 2007), DDSketch (Masson et al., 2019) and ElasticSketch (Yang et al., 2018) respectively. For each use case, we consider two SPADA implementations, where flows are inserted in the *Flow Map* either statically by the control plane (*static*), hence monitoring a predefined set of flows, or dynamically by the data plane (*dynamic*), as new flows are received.

In the remainder of this section, we first describe the architectural components of a generic monitoring system data plane, cf. Figure 3.3. We then detail and contrast two approaches: a baseline that allocates one full sketch per monitored flow, and its corresponding SPADA representation that only stores non-zero counters. We propose two alternative SPADA representations that improve over the baseline in terms of memory footprint with different trade-offs. Finally, we discuss the pros and cons of each representation and analyze SPADA memory sizing.

Use case	Data structure[†]	FM	+	SD	Recirc.
ⓐ	HLL - static	MAT	+	qCHT	✓
	HLL - dynamic	Cuckoo Hash Table (CHT)	+	qCHT	✓
ⓑ	DDSketch - static	MAT	+	pIBLT	×
	DDSketch - static	MAT	+	qCHT	✓
	DDSketch - dynamic	CHT	+	pIBLT	✓
	DDSketch - dynamic	CHT	+	qCHT	✓
ⓒ	ES - dynamic	ES Heavy	+	pIBLT	✓
	ES - dynamic	ES Heavy	+	qCHT	✓

[†] Flows are either statically inserted by the control plane in a simple Match-Action Table (MAT), or dynamically inserted by the data plane.

TABLE 3.2: Summary of SPADA configurations.

3.3.1 Architectural components

Flow Map. In the data plane, a first component performs a `FlowToSketch` mapping to associate incoming flow packets to a specific sketch in the system. This *Flow Map* takes as input a flow key (e.g., the packet TCP/IP 5-tuple) and outputs a `SketchID`. Note that the way the mapping is performed might depend on the monitoring use case. The mapping can be *direct*, i.e., a flow stored in the *Flow Map* is associated with a specific sketch (cf. Figure 3.1a), or *indirect*, i.e., if a flow is not in the *Flow Map*, then it is associated with a default sketch as in (Yang et al., 2018) (cf. Figure 3.1b). Finally, flows stored in the *Flow Map* can be associated with additional flow metadata, e.g., last packet timestamp.

Sketch Data. The `SketchID` retrieved from the *Flow Map* is used to access the measurement counters that need to be updated for a particular flow. These counters are stored in the second component, which we refer to as *Sketch Data*. Based on the monitoring task, this component may store different things. For instance, in the case of ⓑ per-flow IAT quantile estimation, the `SketchID` is used to access a dedicated per-flow DDSketch. Aside from the specific way information is structured within the *Sketch Data*, from a high-level perspective it is an architectural component that stores one or multiple sketches and provides access to the monitoring counters associated with a specific flow, sometimes shared with other flows.

Monitoring routine. In a measurement epoch, the system data plane updates the counters stored in the *Sketch Data* based on incoming packets, as depicted in Figure 3.3. First, at packet arrival, the flow key (e.g., 5-tuple) is extracted and the *Flow Map* is queried to retrieve the associated `SketchID`, optionally updating any flow metadata. Second, the `SketchID` is used to access the *Sketch Data* and locate the sketch for the flow. Finally, depending on the monitoring task, the measurement associated with the last packet is used to determine one index *within* the sketch and modify the corresponding counter. For ⓑ IAT quantile estimation for example, the last IAT value is mapped to a sketch bucket with the DDSketch algorithm, and the counter therein is incremented. At the end of the epoch, the *Flow Map* and *Sketch Data* are read by the control plane, which reconstructs per-flow sketches by looking up all counters belonging to a flow and computes the required metrics.

Notations. SPADA design is based on the assumption, justified by our experiments, that among m sketch counters, the ratio p of non-zero ones is typically small. In order to quantify the memory footprint of the monitoring system data plane, we define $n_u = p \cdot m$ as the average number of sketch counters different than zero, i.e., the lower n_u, the higher the sparsity. With SPADA, we provide a series of memory reduction techniques whose efficiency is inversely proportional to n_u. To analytically estimate the efficiency of SPADA, we derive their memory footprint by using the notation reported in Table 3.3.

Symbol	Meaning	Symbol	Meaning
n_s	number of sketches to store (e.g., one per flow)	s_c	size (in bits) of a single sketch counter
s_k	size (in bits) of the flow key	p	ratio of useful (non-0) counters in a sketch
m	number of sketch counters (buckets)	n_u	average number of non-0 counters in a sketch

TABLE 3.3: Summary of main notations.

3.3.2 Baseline representation

Flow Map. A simple *direct* `FlowToSketch` mapping can be realized using a MAT to provide a unique `SketchID` for every incoming flow. Given the huge number of entries required, this approach is not suitable for handling previously unknown flows. Instead, it requires populating the *Flow Map* statically from the control plane with a known set of flows to monitor. Another solution consists of using a hash of the flow key as `SketchID`. Although this simple approach suffers from collisions, it can be used when approximate measurements are acceptable. Between these two simple options, there is a wide range of possibilities trading off accuracy for memory efficiency.

A unique, direct, `FlowToSketch` mapping can be realized with a CHT. A CHT provides a constant lookup time and a high memory utilization, thus is widely used to implement this kind of per-flow mapping. However, this solution requires a careful design: whereas cuckoo hashing is feasible in programmable data planes, it relies on several stages and has non-constant insertion time which is handled by packet recirculation that may severely impact the system performance. In Appendix A, we provide P4 implementation details and show how to reduce recirculation impact. Regardless of the *Flow Map* implementation, whenever a new `FlowToSketch` mapping is needed, i.e., when a packet from a previously unseen flow key is received, a new `SketchID` can be obtained from a free ID counter. Then, the `<flow key, SketchID>` mapping is stored in the CHT. Note that using a counter to generate unique `SketchIDs` does not constitute a limitation since the system is designed for epoch-based measurements and the counter is reinitialized at the beginning of each epoch. Finally, we note that an indirect `FlowToSketch` mapping can be realized using the *Flow Map* as a filter. This means that flows stored in the *Flow Map* are monitored using their metadata while the remaining ones are monitored via a separate sketch, as it is done in ElasticSketch (Yang et al., 2018).

Sketch Data. A simple way of storing measured values is to allocate enough memory for all the counters required by every per-flow sketch. Hence, in the baseline implementation, the *Sketch Data* component is a list of full sketches, indexed through the `SketchID` that corresponds to the location of the first bucket of the sketch as depicted in Figure 3.4. For instance, in the case of ⓑ per-flow IAT quantile estimation,

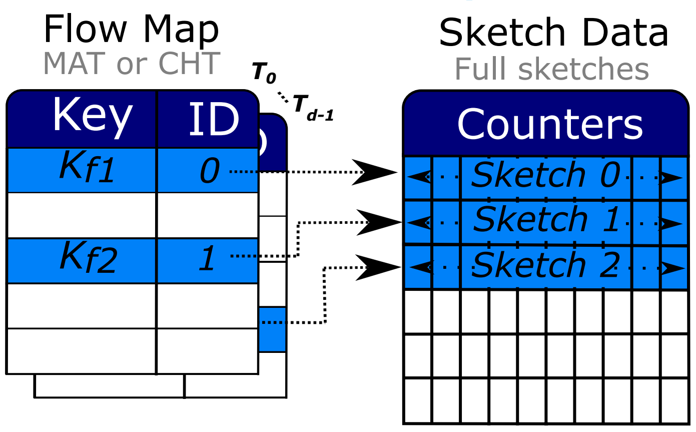


FIGURE 3.4: Baseline *Flow Map* and *Sketch Data* representations in a traditional per-flow monitoring data plane

the `SketchID` is used to locate the first bin of the DDSketch dedicated to a particular flow. At that point, the bin to be modified is retrieved simply as an offset from the `SketchID` (i.e., `SketchID` + bin). Note that, depending on the specific task, a dedicated per-flow sketch might not be required, e.g., ElasticSketch only requires a single Count-Min Sketch.

Memory footprint. We derive the memory needed by the baseline described above assuming that, within one epoch, the monitoring system requires n_s different sketches, i.e., one sketch per flow. We note that the CHT cannot be filled up to 100% (Kirsch et al., 2010) and the expected number of flows needs to be overestimated in the *Flow Map*. Considering a CHT *Flow Map*, the memory requirement of this baseline is:

$$Memory = n_s \cdot (Row_{FM-CHT} + Row_{SD-Base}), \tag{3.1}$$

where Row_{FM-CHT} and $Row_{SD-Base}$ are the memory required to store an entry in the *Flow Map* and a sketch in the *Sketch Data* respectively.

Row_{FM-CHT} and $Row_{SD-Base}$ can be defined as:

$$Row_{FM-CHT} = s_k + \lceil log_2(n_s) \rceil, \qquad Row_{SD-Base} = m \cdot s_c, \tag{3.2}$$

where s_k and $\lceil log_2(n_s) \rceil$ are the flow key and the `SketchID` bit sizes respectively, while m and s_c are the number of sketch counters and their size in bits. Note that independently from the *Sketch Data*, the *Flow Map* size could be further decreased by saving key fingerprints i.e., less than s_k bits.

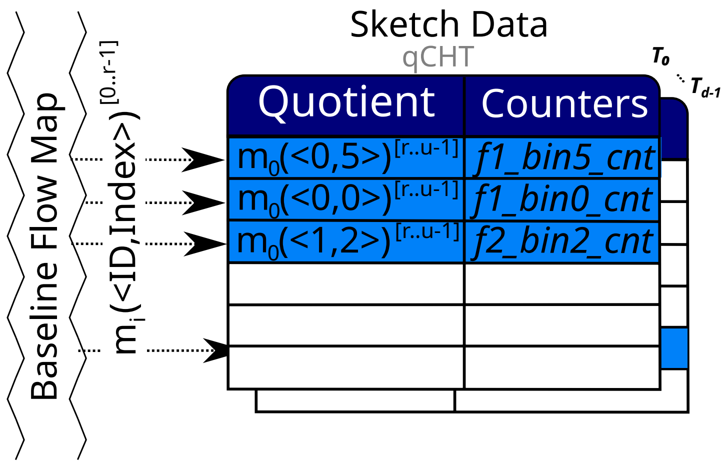


FIGURE 3.5: SPADA representation of the *Sketch Data* with qCHT

3.3.3 Sparse sketches representation (SPADA)

In this section, we describe the SPADA representation for the *Sketch Data*. The main idea is to replace per-flow sketches with a series of non-zero counters, addressable with the pair `<SketchID, index>` where `index` is the sketch counter position in the logical per-flow sketch. The key difference with respect to the baseline is that the memory required by each sketch depends on the number of its non-zero elements. Indeed, while the baseline *Sketch Data* statically reserves an entire sketch upon receiving the first packet of a flow, SPADA creates a "virtual sketch" by reserving a unique `SketchID` whenever a new flow is received, and dynamically assigns pre-reserved counters whenever a new `<SketchID, index>` pair is required. We design two possible implementations of the sparse *Sketch Data*: the first one uses a qCHT, while the second one uses a modified version of the IBLT, namely perfect IBLT (pIBLT). While the former has the drawback of requiring data plane recirculation, it provides additional flexibility compared to the pIBLT, as the latter can only be used with sketches whose update operation consists of a linear increase.

Sparse Sketch Data with qCHT. This version of the *Sketch Data* is based on a Cuckoo Hash Table (CHT), a key-value data structure with constant lookup time. As shown in Figure 3.5, we use `<SketchID,index>` pairs (of u bits) as CHT keys and use it to store non-zero counters. A CHT comprises d tables ($d = 4$ in our settings), hence `<SketchID,index>` pairs are hashed d times to generate one index per table. The CHT stores both key and value, and key conflicts are resolved by moving entries across the various tables; this comes at the price of a non-constant insertion time. To reduce the size of keys in the table, SPADA uses a CHT with quotienting (qCHT). A full description of the CHT and of the quotienting technique has been provided in Chapter 2. As a reminder, a qCHT relies on d bijective functions m_i to hash the keys, uses the least r significant bits of the hashes to index the tables, and only stores the remaining $u - r$ bits for conflict resolutions instead of the whole key of u bits.

Before analytically deriving the memory required by a qCHT *Sketch Data*, let us first analyze the simpler case of a sparse *Sketch Data* using a simple CHT:

$$Memory = n_s \cdot (Row_{FM-CHT} + n_u \cdot Row_{SD-CHT}), \tag{3.3}$$

where the first part of the equation is the memory required for a CHT *Flow Map* as for the baseline while the second part is the size of one entry in the sparse *Sketch Data* with CHT, multiplied by the expected number of non-zero sketch counters n_u. We can then express Row_{SD-CHT} as follows:

$$Row_{SD-CHT} = (\lceil log_2(n_s \cdot m) \rceil) + s_c, \tag{3.4}$$

where $\lceil log_2(n_s \cdot m) \rceil$ is the size of each key stored in the CHT, i.e., `<SketchID, index>`, and s_c the size of each counter in bits. Now that we have derived the memory requirement of the sparse *Sketch Data* using a simple CHT, let us detail the case when using a qCHT. Since we use a qCHT composed of 4 tables, each table contains up to $(n_s \cdot n_u)/4$ elements. Therefore, it can be addressed by using only $r = \lceil log_2(n_s \cdot n_u/4) \rceil$ bits instead of $u = \lceil log_2(n_s \cdot m) \rceil$, i.e., the full length of the key. To detect collisions, the remaining $u - r$ quotient bits are stored in the table:

$$u - r = \lceil log_2(n_s \cdot m) \rceil - \lceil log_2(n_s \cdot n_u/4) \rceil \approx 2 + log_2(m/n_u) = 2 + log_2(1/p), \tag{3.5}$$

The memory requirements for the sparse *Sketch Data* using qCHT can be summarized as:

$$Row_{SD-qCHT} \approx (2 + log_2(1/p)) + s_c, \tag{3.6}$$

It is worth noting that $u - r$ is small since the sparse *Sketch Data* stores a significant fraction of the items in the `<SketchID, index>` universe. This is a quite different setting compared to the *Flow Map* one, as in the latter case the key size may exceed 100 bits, leading to negligible savings.

To provide a rough idea of SPADA-qCHT memory saving, we report here a concrete monitoring use case for ⓑ IAT quantile estimation. We use the 5-tuple as flow key, and a DDSketch with $m = 64$ counters, i.e., bins, of size $s_c = 8$ bits. We assume a conservative bin usage ratio of $p = 0.15$ and a small $n_s = 100K$ number of sketches. Note that sparsity factors extracted from real traffic traces later used in the evaluation section are between 0.003 and 0.16. The baseline implementation requires 7.9 MB overall, of which 6.4 MB for the *Sketch Data* store. SPADA-qCHT requires the same 1.5 MB *Flow Map* and a 1.5 MB sparse *Sketch Data*, for 3 MB overall, i.e., 62% memory saving. Note that these savings result from both the quotienting technique and sparse representation. Indeed, a sparse *Sketch Data* using CHT without quotienting would lead to an overall memory footprint of 5.2 MB, i.e., 35% memory saving.

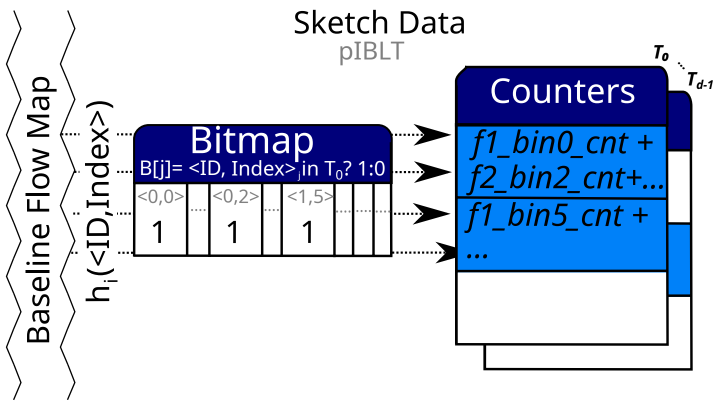


FIGURE 3.6: SPADA representation of the *Sketch Data* with pIBLT

SPADA-qCHT has two main drawbacks. First, each non-zero counter requires more memory with respect to its baseline counterpart. Indeed, we store the quotient of the key `<SketchID, index>` in addition to the counter itself. However, this overhead is greatly compensated by the fact that only non-zero counters are stored. Memory saving thus depends on the sparsity factor p: the lower p, the more negligible this per counter overhead, leading to higher memory savings. A quantitative analysis of this effect can be appreciated in Section 3.4 where we show memory savings at different sparsity factors. The second drawback is that CHTs in programmable data planes are challenging as they require a non-constant number of memory accesses at insertion time, which might not be acceptable on constrained hardware. In Appendix A.1 we address this challenge and propose a CHT implementation compatible with programmable data planes.

Sparse Sketch Data with pIBLT. To overcome the limitations of qCHT, we propose an alternative SPADA representation based on the Invertible Bloom Lookup Table (IBLT), a structure that provides key-value storage with a fixed number of memory accesses, as introduced in Chapter 2. As a reminder, IBLT aggregates multiple keys within the same bucket, while each key is stored in d separate buckets addressed with d hash functions h_i. Using IBLT introduces two challenges. First, the per-bucket overhead is higher than the qCHT, as each bucket requires: a counter maintaining the number of colliding keys stored in the bucket, the sum (or XOR) of the colliding keys, and the sum (or XOR) of the values associated to such keys. Second, to extract stored values, the IBLT uses a peeling procedure named `ListEntries`, that imposes a strict upper bound to the load factor, i.e., 82% achieved using $d = 3$ (Walzer, 2021).

We propose an improved IBLT that uses a bitmap B to keep track of `<SketchID, index>` pairs, i.e., B features $2^u = n_s \cdot m$ bits, one for each possible pair. We call this modified structure perfect IBLT (pIBLT). Differently from a IBLT, our pIBLT does not need to count the number of colliding keys per bucket, not does it require to store summed (or XORed) keys to take note of which ones contribute to the corresponding bucket. Instead, such information is retrieved from the bitmap B. Therefore, in

Algorithm 2 pIBLT `ListEntries`

```
Require: $B, T_0 \dots T_{d-1}$
1: Initialize: all $a_{ij} = 0$; initialize $b_i$ values from counters in $T_0 \dots T_{d-1}$
2: for $j \in [0 \dots 2^u - 1]$ do
3:     if B[j] == 1 then
4:         save key $key_j$ (i.e., a pair <SketchID, index>)
5:         for all $i \in \{h_0(key_j), \dots, h_{d-1}(key_j)\}$, set $a_{ij} = 1$
6:     end if
7: end for
8: solve the linear system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$
9: return pairs $(key_j, x_j)$ for all saved keys
```

pIBLT each bucket only sums the values of its associated keys. Besides, the bitmap removes false positives and does not require peeling to retrieve entries, that can be derived by solving a system of linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, where $a_{ij} \in \{0, 1\}$ indicates whether key_j contributed to bucket b_i, and x_j denotes the value associated with key_j. We detail the new `ListEntries` in Algorithm 2. Note that this `ListEntries` procedure is only executed in the control plane after a measurement epoch. Therefore, the time needed to solve the linear system does not introduce any overhead in the data plane monitoring pipeline.

The *Sketch Data* implemented using pIBLT is illustrated in Figure 3.6. We use a pIBLT with $d = 4$ tables, each indexed using a different hash function and featuring $n_s \cdot n_u / 4$ locations. Each location simply stores the sum of its associated sketch counters. At each update, the d buckets associated with the `<SketchID, index>` key are incremented, and the corresponding bit in the bitmap is set. It can be proven (Dubois and Mandler, 2002; Dietzfelbinger et al., 2010; Pittel and Sorkin, 2016) that $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ has a unique solution with high probability when the load factor is below a threshold c_k. For $d = 4$, $c_k = 0.97$, which is higher than the IBLT peeling threshold i.e., 0.82. Even if the probability that $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ does not have a unique solution is small, we remark that in this case, it is possible to derive an approximate resolution. For example, in PR-Sketch (Sheng et al., 2021) an iterative method provides the solution with minimum ℓ_2-norm. FlowLidar (Monterubbiano et al., 2023a) presents an alternative method based on an initial peeling phase removing dependent rows from $\mathbf{A}$.

Memory footprint. For the pIBLT, the memory requirements can be expressed as:

$$Memory = n_s \cdot (Row_{FM-CHT} + n_u \cdot s_c) + (n_s \cdot m), \tag{3.7}$$

where the space occupied by the CHT *Flow Map* is the same as the baseline, $n_u \cdot s_c$ is the space occupied to store the counters of a single sketch, and $n_s \cdot m$ is the size of the bitmap B. Similarly to the qCHT version, the advantages of using a pIBLT *Sketch Data* in place of the baseline diminishes with increasing values of p, i.e., with an increasing number of non-zero counters.

We now highlight the SPADA-pIBLT memory footprint referring to the same use case ⓑ as previously for SPADA-qCHT. As the *Flow Map* is the same, its required memory remains 1.5MB. Concerning the *Sketch Data*, we need 800KB for the bitmap, that is, one bit for each possible `<SketchID, index>` pair, while the 4 tables require $n_s \cdot n_u \cdot s_c = 960$KB. Therefore the total memory is around 3.26MB, which is similar to the case of the qCHT, i.e., 60% smaller compared to the baseline.

We remark that pIBLT supports only additive counters, e.g., used in CMS or DDSketch, hence it is not suitable for HLL since it requires reading values at update time to write the maximum. Furthermore, with a qCHT *Sketch Data*, increasing m, i.e., the number of buckets of each sketch, has negligible impact on the memory size. This is not true for the pIBLT *Sketch Data*, as the size of the bitmap storing the non-zero `<SketchID, index>` pairs grows linearly with the number of buckets.

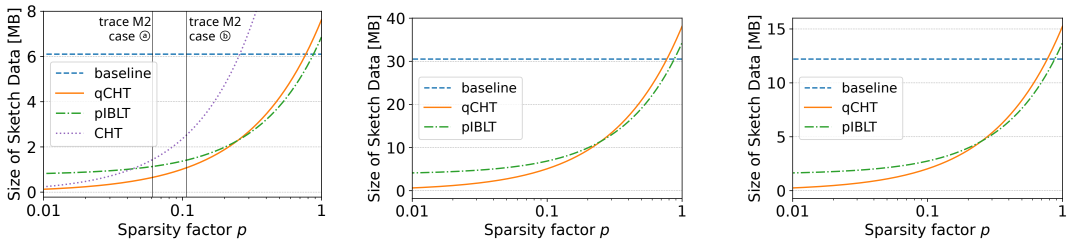
3.4 System Memory Sizing

In this section, we analyze the relationships between sparsity, the number of flows to be monitored, and the required memory size. We detail the trade-offs of using SPADA in place of the baseline, comparing the two proposed solutions based on qCHT and pIBLT in multiple settings. This synthetic analysis breaks down the advantages that SPADA can bring in practice and provides an insight into proper system configuration based on the available memory and the number of flows to monitor.

3.4.1 Sparsity factor sensitivity

The baseline *Sketch Data* memory footprint depends on *(i)* the number of flows n_s to monitor, i.e., number of sketches in most use cases, and *(ii)* the desired accuracy, i.e., m buckets per sketch. Additionally, SPADA *Sketch Data* memory footprint depends on a worst-case sparsity assumption, i.e., average ratio of non-zero counters per sketch p. Despite allowing flexible bucket assignments across different flows, our solution requires fixing the total number of non-zero buckets in practice. If the average p of the data is higher than the expected one, the *Sketch Data* reaches its critical load factor, making new insertions impossible.

Figure 3.7a depicts the *Sketch Data* memory footprint for the three implementations varying p, assuming $n_s = 100K$ sketches with $m = 64$ buckets each, and a bucket size $s_c = 8$ bits. The plot also reports the sparsity factors for two reference use cases, i.e., ⓐ super spreader and ⓑ IAT quantiles estimation, extracted from the worst-case trace used in our evaluation (cf. trace M2 in Table 3.4 for further details). We remark that the pIBLT bitmap introduces a constant cost that is relatively high when p is small, i.e., when data is highly sparse. This disadvantage diminishes for higher values of p, as the pIBLT does not use extra memory to store the key of each bucket like the qCHT. For $p > 0.25$ the disadvantage of the bitmap disappears and



(A) n_s=100K, m=64 (B) n_s=500K, m=64 (C) n_s=100K, m=128

FIGURE 3.7: Relationship between data sparsity p and memory requirements of SPADA data structures

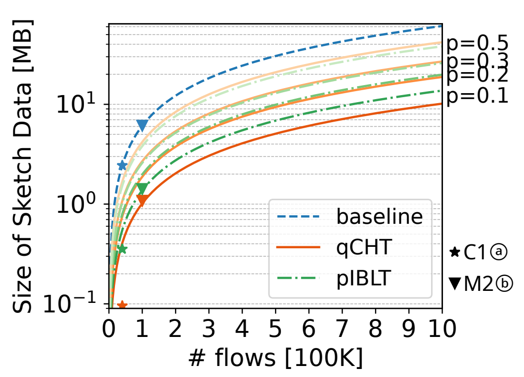
the qCHT requires slightly more memory. In general, both solutions greatly outperform the baseline even for very conservative sparsity assumptions (up to $p \approx 0.75$). Finally, we remark that the vanilla CHT is not a good solution for the *Sketch Data* implementation in most cases, as it occupies more memory than the baseline when $p \approx 0.3$. This also showcases the impact of quotienting, that makes the per-counter overhead of CHT more and more detrimental against qCHT as p increases (i.e., as more `<SketchID, index>` pairs are inserted in the *Sketch Data*). Additionally, Figures 3.7b and 3.7c also report the size of the *Sketch Data* according to p, but with $\times 5$ more flows under monitoring and $\times 2$ more buckets respectively.

We note that, in extreme cases when the worst-case p exceeds expectations, SPADA can stop adding new counters or start allowing sharing them across different flows at the cost of losing accuracy with respect to the original per-flow sketch, as it would happen for non-SPADA structures with more memory. Nevertheless, we remark that, as highlighted in Figure 3.7a, the memory trade-off provided by SPADA is better when compared to a non-sparse implementation, even for very conservative sparsity assumptions (up to $p = 0.7$).

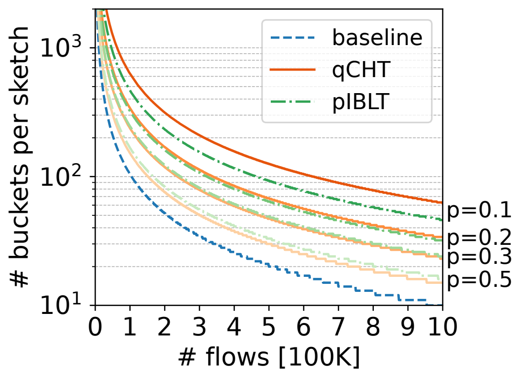
3.4.2 Monitoring trade-offs

By exploiting sparsity, SPADA reduces the memory footprint of the *Sketch Data* under conservative ratios p. Here, we express these memory gains in terms of additional monitoring capacity provided to the system thanks to SPADA. Specifically, based on the system requirements, i.e., considering a fixed memory budget or a desired accuracy, we provide insights on how to properly set up SPADA.

First, Figure 3.8a contrasts the required *Sketch Data* memory with the number of monitored flows. For instance, assuming that 10 MB are pre-allocated for the *Sketch Data*, a baseline implementation would allow to monitor $\approx 150K$ flows, while SPADA can monitor $\approx 400K$ assuming a worst-case $p = 0.3$. Second, Figure 3.8b shows the trade-off between the number of monitored flows and the desired monitoring accuracy, i.e., number of buckets per sketch m, for 10 MB of available memory. Based on the number of desired flows, one can adjust the expected sparsity factor p to reach the desired monitoring precision: with 500K flows, setting $p = 0.3$ only grants $m \approx 50$



(A) Memory vs. number of flows for various values of p (m=64).



(B) Max number of buckets per sketch m in 10 MB of *Sketch Data*.

FIGURE 3.8: Monitoring trade-offs

counters per sketch, while $p = 0.2$ brings m to ≈ 70. We highlight that properly selecting p, hence sizing the system accordingly, requires additional considerations based on historical knowledge, traffic predictions, and the purpose of the monitoring system itself.

3.5 System Evaluation

In this section, we evaluate SPADA on reference use cases using CAIDA 2016 (*The CAIDA Anonymized Internet Traces Dataset* 2016) and MAWI 2019 datasets (*The MAWI Working Group Traffic Archive* 2019) and a custom software simulator available at `https://github.com/cpt-harlock/SPADA`. We compare SPADA against state-of-the-art approaches in terms of memory requirements and accuracy (when affected). Additionally, we provide in-depth analysis of SPADA's memory requirements based on sparsity. Evaluation of the FPGA implementation and discussion of its feasibility in real systems is deferred to Appendix A.

3.5.1 Experimental protocol

To evaluate the efficiency of SPADA in a realistic scenario, we feed the simulator with 1-hour **C**AIDA traces and 5-minute **M**AWI traces. In particular, we consider:

- **C1** 2016-01-21 13:00 – 14:00
- **C2** 2016-03-17 14:00 – 15:00
- **M1** 2019-04-09 22:15 – 22:20
- **M2** 2019-04-09 13:15 – 13:20

For CAIDA traces, we consider 1-second epochs, while for MAWI traces we use 1-minute epochs since they feature fewer packets. We consider TCP traffic only from all traces.

Use case	**Parameters**						**Sparsity factor**			
	s_k	m	n_s C1	n_s C2	n_s M1	n_s M2	p_{C1}	p_{C2}	p_{M1}	p_{M2}
ⓐ	32	64	36K	11K	5.4K	7.6K	0.020	0.028	0.051	0.061
	32	128	36K	11K	5.4K	7.6K	0.010	0.015	0.028	0.034
ⓑ	104	32	59K	31K	41K	100K	0.068	0.078	0.139	0.162
	104	64	59K	31K	41K	100K	0.040	0.048	0.092	0.107
ⓒ	104	0.5M	59K	31K	41K	100K	0.085	0.025	0.039	0.124
	104	4M	59K	31K	41K	100K	0.011	0.003	0.005	0.016

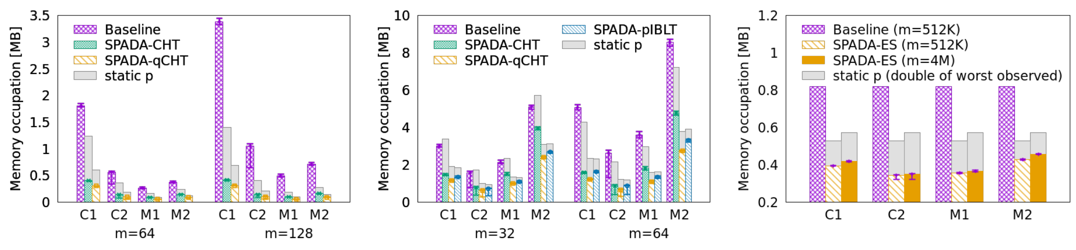
TABLE 3.4: Data structure parameters for each use case and their sparsity values for traces C1, C2 and M1, M2.

For ⓐ super spreader detection, we use source IPs as flow keys and $m = 64$ or $m = 128$ counters for the HLL sketches (error 13% and 9% respectively). For ⓑ IAT quantile estimation, we use 5-tuples as flow keys and DDSketches with $m = 32$ or $m = 64$ counters each (relative error $\alpha = 0.28$ and $\alpha = 0.14$ respectively). Finally, for ⓒ flow size estimation, we build a baseline ElasticSketch (ES) of 818KB, divided into a heavy part of 318KB and a light part of 500KB (i.e., a CMS composed of $m = 512$K 8-bit counters). SPADA-ES allocates the same amount of memory to the heavy part (*Flow Map*), uses a qCHT to store the light part (*Sketch Data*), and is tested using ES open source simulator (*Elastic Sketch source code* 2023). Additionally, we evaluate a more accurate SPADA-ES "virtually" increasing the CMS size ($m = 4$M). In our simulations both *Flow Map* and *Sketch Data* are sized to keep the load factor below 90%, and all (q)CHTs feature a stash with 16 additional buckets (Kirsch et al., 2010). Table 3.4 lists the main SPADA parameters for uses cases ⓐ, ⓑ and ⓒ, also reporting the sparsity recorded for the various traces, expressed as the average ratio of non-zero sketch counters p_{Ci}, ranging from 0.003 to 0.162.

3.5.2 Experimental results

Memory occupancy. Figures 3.9a, 3.9b and 3.9c contrast the memory occupied by the baseline and SPADA monitoring system. Colored histograms average values across the epochs, with error bars for min and max values when available. Overall, SPADA reduces memory occupancy from 2× for ⓑ (DDSketch), 2.5× for ⓒ (ElasticSketch), to 11× for ⓐ (HLL). More precisely, the sparser the baseline sketches, the higher the memory saving. As previously analyzed, SPADA-pIBLT requires more memory than SPADA-qCHT due to the additional bitmap. However, the memory saving with respect to the baseline is still significant with the advantage of avoiding recirculation in the *Sketch Data* component.

Additionally, Figures 3.9a, 3.9b and 3.9c show over-dimensioned SPADA memory footprint using a conservative fixed value of p (grey bars), where we set p to the double of the highest value observed in each experiment — cf. Table 3.4. We remark that with qCHT and pIBLT *Sketch Data*, memory reduction is still sizeable despite



(A) Super spreader detection. (B) IAT quantile estimation. (C) Flow size estimation.

FIGURE 3.9: Memory footprint for the reference use cases comparing SPADA with SOTA baselines.

conservative settings. Conversely, as expected for the CHT, the per-counter overhead is too high, and hence its memory occupancy is in practice similar, sometimes even higher, than the baseline (for $p \approx 0.3$).

Processing time for pIBLT lookup. At the end of the measurement epoch, the control plane dumps the pIBLT content and solves the linear system associated with it. Note that, even though this computation does not affect the data plane, the system needs to be solved before the end of the next epoch to avoid overloading the resolution system. Since the resolution time of a linear system is superlinear with respect to the number of equations, we use a first-level hash function to split the rows of the linear system into a set of smaller disjoint linear systems. This reduces the computation time and enables the use of multiple cores. In particular, with two threads of an Intel i7-10700K CPU clocked at 3.80 GHz, the linear system of the pIBLT is solved in less than one second (960 ms). We achieved this result by splitting the 36K rows of the linear system into 36 independent linear systems of 1K rows each. The time needed for solving each of these systems is approx. 15 ms.

Flow size estimation accuracy. While for use cases ⓐ (super spreader) and ⓑ (IAT quantiles), the estimation accuracy is not impacted, use case ⓒ (flow size estimation) requires additional consideration. As shown in Figure 3.9c, with SPADA we can increase the CMS size m from 512K to 4M with minimal impact on memory. We now compare the accuracy of the baseline ElasticSketch (ES) with our enhanced SPADA-ES. Table 3.5 reports Average Relative Error (ARE), Average Absolute Error (AAE), and the fraction of flows for which the sketch provides the exact result (ER). We note that, despite the much smaller memory footprint, SPADA-ES always provides better accuracy than the baseline ES. In particular, SPADA-ES achieves one order of magnitude better AAE and ARE than ES. Furthermore, SPADA-ES provides the exact count for more than 98% of flows in *all* traces, whereas the standard ES tops at 98% only in C2 which is the trace featuring fewer flows.

Metric		m	C1	C2	M1	M2
ARE	ES	0.5M	0.10	0.01	0.05	0.30
	SPADA-ES	4M	0.01	2E-3	6E-3	0.03
AAE	ES	0.5M	0.16	0.02	0.12	0.97
	SPADA-ES	4M	0.02	3E-3	5E-3	0.12
ER	ES	0.5M	0.93	0.98	0.97	0.89
	SPADA-ES	4M	0.99	0.99	0.99	0.98

TABLE 3.5: ES and SPADA-ES accuracy for traces C1, C2, M1, M2.

3.6 Related Work and Discussion

Several Flow-to-ID mapping techniques have been proposed in the literature (Pontarelli et al., 2019; Zhao et al., 2021; Barbette et al., 2020). Since SPADA main goal is to compact the *Sketch Data*, we do not directly review them but we remark that such techniques could be used in SPADA to further reduce the memory footprint.

In the context of cardinality estimation, HLL (Flajolet et al., 2007) and BeauCoup (Chen et al., 2020) are popular and efficient methodologies proposed in the literature. HLL enhancements (Jia et al., 2020; Xiao et al., 2015) try to exploit sparsity to reduce the data structure memory footprint. Unlike SPADA however, these approaches employ a counter-sharing mechanism that affects the measurement accuracy. BeauCoup instead, performs distinct counting through "coupons" collection in bit-vectors whose size trades off memory occupancy and accuracy e.g., for $m = 256$ bits for each vector. BeauCoup error is comparable with HLL using the same memory while $m = 32$ increases the error by $\approx 3\times$ for $\approx 70\%$ less memory. Unlike BeauCoup, SPADA provides significant memory saving without sacrificing accuracy. It is worth mentioning that SPADA can also be applied to BeauCoup, as the bit-vectors are affected by sparsity ($p \approx 0.1$). Finally, SPADA can reduce the size of a series of other data structures in the context of cardinality estimation, e.g., PCSA (Flajolet and Martin, 1985), KVM (Bar-Yossef et al., 2002), and Fast-AGMS (Cormode and Garofalakis, 2005). Similar considerations apply to the quantile estimation use case. In particular, Circllhist (Hartmann and Schlossnagle, 2020) and KLL (Karnin et al., 2016) feature sparse arrays similar to DDSketch and can be implemented using SPADA representations. Other approaches rely on a compact data representation by restricting the data collection to a subset of more relevant flows. For instance, SQUAD (Shahout et al., 2023) combines the problem of quantile estimation with the one of heavy hitter detection, hence dynamically restricting quantile estimation only to the most frequent flows. Instead, SPADA does not need to rely on a prediction mechanism and significantly reduces the memory footprint without the drawback of restricting monitoring to a few flows. The key difference between SPADA and the aforementioned approaches is that it relies on *sparse data representation* to mitigate memory footprint. This leads to a generic technique that can be applied to a variety of other use cases as summarized in Table 3.1.

Existing sparse representations are typically restricted to static data structures and include well-known techniques such as Compressed Sparse Row, Coordinate Format, etc. Dynamic sparse representations such as STINGER (Ediger et al., 2012), AIM (Winter et al., 2017), and HORNET (Busato et al., 2018) have been recently proposed in the context of graph and matrix representations for parallel processing units, e.g., GPU. However, such solutions are deployed on hardware where memory access is not constrained, unlike programmable switches. Finally, sparse representation directly relates to Compressive Sensing (Fornasier and Rauhut, 2015). In the context of sketching, NZE (Huang et al., 2021) uses Compressive Sensing to design specific sketches that can approximately reconstruct the desired measurement, e.g., flow size estimation. Although NZE goal is similar to SPADA, we note that the reconstruction complexity in our case is negligible, whereas for NZE it exponentially increases with the number of flows.

3.7 Conclusion

This chapter presented SPADA, a Sparse Approximate Data Structure representation. SPADA is a method to reduce the data plane memory occupancy of per-flow monitoring systems without affecting accuracy. SPADA exploits the observation that, due to the skewed nature of network traffic, only a handful of sketch counters are used for most flows. This leads to heavily underutilized data structures in a number of monitoring use cases. SPADA has been designed to efficiently represent such sparse data by only storing non-zero sketch buckets and relies on (q)CHT and on a novel data structure pIBLT. We performed extensive simulations and experiments on real traces for three popular monitoring use cases, achieving a memory reduction between 2× and 11× while maintaining the same accuracy and introducing limited computational overhead.

Related publication

Andrea Monterubbiano, Raphael Azorin, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi (Nov. 2023c). "SPADA: A Sparse Approximate Data Structure Representation for Data Plane Per-Flow Monitoring". In: *Proceedings of the ACM on Networking* 1.CoNEXT3. DOI: 10.1145/3629149

Chapter 4

Affordable Flow Size Representation with Machine Learning

In this chapter, we develop a *Machine Learning-assisted measurements* system that provides coarse flow size representations to improve the usage of monitoring data structures. We first note that early flow size prediction would be beneficial for a wide range of networking use cases. However, implementing an ML-enabled system is a challenging task due to network devices' limited resources. While previous works have demonstrated the feasibility of running simple ML models in the data plane, their integration in a practical end-to-end system is not trivial. Additional challenges in resources management and model maintenance need to be addressed to ensure the network task(s) performance improvement justifies the system overhead. To this regard, we propose DUMBO, a versatile end-to-end system to generate and exploit flow size hints at line rate. Our system seamlessly integrates and maintains a simple ML model that offers early classification of elephants and mice flows in the data plane. We evaluate the proposed system on flow scheduling, per-flow packet inter-arrival time distribution, and flow size estimation using three real traffic traces. Our results show that DUMBO outperforms traditional state-of-the-art approaches by equipping network devices data planes with a lightweight ML model. Code is available at https://github.com/cpt-harlock/DUMBO.

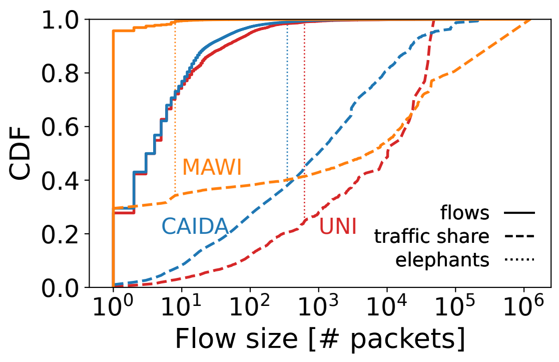


FIGURE 4.1: Flow sizes distribution and traffic share on three network traces.

4.1 Introduction

Flow size prediction is an important yet challenging problem in the networking community. Accurately predicting flow size provides valuable information for network administrators to improve network management, as acknowledged in (Đukić et al., 2019). Flow size distributions exhibit a heavy-tailed nature, with the majority of flows being short (referred to as *mice*) and a small fraction of them being significantly larger (known as *elephants*). To provide a concrete example, Figure 4.1 reports the flow size distribution from three different traffic traces: CAIDA (*The CAIDA Anonymized Internet Traces Dataset* 2016), MAWI (*The MAWI Working Group Traffic Archive* 2019), and UNI (Benson et al., 2010). While elephants represent only a tiny fraction of all flows, e.g., the top 1%, they correspond to a substantial portion of the overall volume. Consequently, elephants significantly impact network management and monitoring tasks (Đukić et al., 2019; Hsu et al., 2019). Due to the dynamic nature of network traffic, precise flow size prediction is a difficult operation to perform, especially in the data plane.

We argue that early elephant flows identification is a more attainable objective, which still offers a significant advantage for network management and monitoring tasks. Our broad vision encompasses a networking system that seamlessly integrates machine learning (ML) models directly into the data plane, enabling the provision of timely flow size *hints*. Downstream networking tasks can then leverage these hints to enhance their overall performance and efficiency. One notable family of tasks that can greatly benefit from traffic predictions is *network management*: this includes congestion control, scheduling, and routing, wherein several studies have explored the usage of measurements or forecast-based hints (Lee et al., 2015; Li et al., 2019; Kumar et al., 2020; Alizadeh et al., 2013; Gao et al., 2019; Perry et al., 2014; Đukić et al., 2019; Poupart et al., 2016; Sacco et al., 2020). Another family that would benefit from traffic predictions is *network monitoring*, in which approximate data structures are typically used to reduce system memory footprint. By leveraging ML hints, the

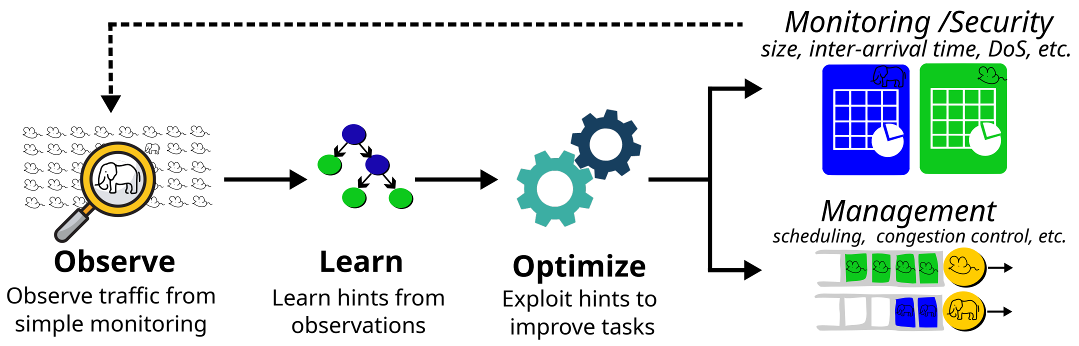

Monitoring /Security
size, inter-arrival time, DoS, etc.
Management
scheduling, congestion control, etc.
Observe
Observe traffic from simple monitoring
Learn
Learn hints from observations
Optimize
Exploit hints to improve tasks

FIGURE 4.2: Synopsis of DUMBO.

monitoring system can dedicate more memory to elephant flows, which are naturally more important than the others cf. Figure 4.1. In turn, this segregation helps reduce estimation errors, thereby improving the reliability of various monitoring tasks (Karnin et al., 2016; Masson et al., 2019; Yang et al., 2018) as demonstrated by (Hsu et al., 2019; Du et al., 2021).

A high-level view of an ML-enabled data plane pipeline is illustrated in Figure 4.2. In a nutshell, the system collects lightweight observations (or measurements) which are used to train an ML model providing hints that are then exploited to improve downstream tasks. Despite its potential, several challenges must be addressed to realize such a system. One significant challenge pertains to the limited number of available operations and memory scarcity in network devices, e.g., SmartNICs, switches, etc. Introducing an ML model in network devices incurs overhead which effectively competes for resources with existing network functionalities. Therefore, the ML-enabled data plane benefits must outweigh the associated costs and apply to multiple network tasks, while ensuring that the imprecision of the model does not degrade the performance of the tasks it is supposed to support.

Contribution. This chapter presents DUMBO, a comprehensive system that provides traffic characteristics hints in the form of simple binary classifications, elephants or mice flows. Our proposal is based on three key observations: *(i)* simple hints can be learned by a constrained machine learning model, *(ii)* whose implementation in the data plane can be realized with modern programmable hardware, and *(iii)* that holds significant value for multiple downstream networking tasks. While previous work has addressed some of these aspects individually (Hsu et al., 2019; Du et al., 2021; Zhang et al., 2021b; Xiong and Zilberman, 2019; Akem et al., 2022), we conduct an end-to-end analysis of the system, spanning from design to implementation and experimentation. DUMBO leverages a single ML model to enhance various networking tasks, carefully considering the trade-offs between performance gains and overhead. By encompassing a holistic approach, we address the entire system lifecycle, examine its design choices, and evaluate its benefits on the final tasks performance metrics. In contrast to existing literature, this approach enables us to accurately assess the capabilities and the benefits of our system.

Organization. Section 4.2 introduces the scientific background and main motivations of this work, presenting the use cases we consider to showcase DUMBO: flow scheduling, packet inter-arrival time distribution, and flow size estimation. Section 4.3 presents the system design and Section 4.4 investigates the development of a suitable ML model, including a thorough analysis of model performance, size, and update. Section 4.5 presents the details of the system implementation. Section 4.6 validates the system end-to-end using real traffic traces, also providing a performance trade-off analysis. Section 4.7 concludes the chapter.

4.2 Background and Motivation

In this section, we first motivate our ML-based data plane pipeline and position our contribution with respect to related work. We then introduce the three use cases we use as a reference to evaluate the end-to-end effectiveness of our approach.

4.2.1 Machine Learning for networked systems

Predicting flow size. Previous studies have demonstrated the advantages of early flow size prediction in various networking scenarios, such as flow scheduling (Đukić et al., 2019), routing (Poupart et al., 2016; Sacco et al., 2020), congestion control (Lee et al., 2015), and flow measurements (Hsu et al., 2019; Du et al., 2021). While statistics-based heuristics can be used efficiently (Sivaraman et al., 2017; Ben Basat et al., 2020a), they often necessitate a long detection time, thereby diminishing the utility of the provided hints. Therefore, researchers have turned to ML techniques to train models that can approximate flow sizes in advance. Most studies have focused on offline analysis, proposing complex and computationally expensive Deep Learning models, such as Recurrent Neural Networks (Hsu et al., 2019; Du et al., 2021).

The possibility of obtaining accurate flow size predictions has been questioned in (Đukić et al., 2019), leading researchers to investigate the implications of acquiring such information. However, in the context of job scheduling, (Mitzenmacher, 2021) recently provided theoretical evidence that a simple binary hint of whether a job is "big" or "small" can lead to significant gains in the scheduling task, even when the hint is not accurate. We believe that this observation is true also for many networking tasks that identify flows using TCP/IP five-tuple. Hence, we consider a simple Random Forest model for a classification task, similarly to (Hsu et al., 2019; Du et al., 2021; Zhang et al., 2021b), i.e., predict "elephants" or "mice" flows based on the first few packets. We define elephants as the top-1% (cf. Figure 4.1) and mice as the rest. In (Hsu et al., 2019; Du et al., 2021), authors only use features from the flow five-tuple, thus being able to perform classification using only the first packet. Because such an approach is highly vulnerable to traffic variations, pHeavy (Zhang et al., 2021b) proposes to wait for the first 5—20 packets to extract additional features, hence providing a later prediction. Compared to pHeavy, we limit the DUMBO pipeline to the first 5 packets and develop a model that can be run at the beginning of the flow and is 3.5× more precise.

Deployment trade-off. Despite the rich literature, several challenges remain unresolved for practical flow size prediction in network devices data planes with a limited amount of resources e.g., Smart NICs, switches. We argue that prior works that address the ML deployment aspect of the problem (Xiong and Zilberman, 2019; Akem et al., 2022; Siracusano et al., 2022; Zheng and Zilberman, 2021; Akem et al., 2023; Zhang et al., 2021b; Busse-Grawitz et al., 2022) are far from demonstrating an end-to-end pipeline that can be adopted in practice. The traditional ML pipeline, involving the selection of the best model on a validation dataset and its subsequent use in the target environment, has known limitations (D'Amour et al., 2022; Arp et al., 2022; Amodei et al., 2016). This approach overlooks the properties of the target system and focuses solely on classification performance rather than evaluating the system end-to-end. Indeed, a critical aspect of data plane ML integration pertains to the trade-off between the model benefits and its deployment overhead. Although this trade-off has been partially studied (Xiong and Zilberman, 2019; Akem et al., 2022; Zheng and Zilberman, 2021; Akem et al., 2023; Busse-Grawitz et al., 2022), we posit that considerable effort is still needed. We highlight two main problems that affect the analysis done by existing works. On one hand, *(i)* the effectiveness of a model should be defined by its ability to consistently provide valuable hints for downstream tasks and the penalties that may result from incorrect predictions, and not by an offline performance metric. On the other hand, *(ii)* the operational overhead should be characterized by all the additional components needed for the model to operate in the data plane.

When designing DUMBO, we take into account these aspects and evaluate the deployment trade-off by accounting for the memory and processing overhead and relate the model performance metric with the actual end-to-end performance of the tasks. For the deployment to be practical, DUMBO should impose limited memory and processing overhead, and outperform its non-learned counterparts which operate without any hint but have access to more resources. Finally, we include in

Family	Use case	References	Description
Network management	Congestion control	FACC (Lee et al., 2015), HPCC (Li et al., 2019), ORCA (Abbasloo et al., 2020), Swift (Kumar et al., 2020)	*Elephant* flows are treated separately e.g., lower priority.
	Scheduling and routing	pHost (Gao et al., 2019), pFabric (Alizadeh et al., 2013), MLRouting (Poupart et al., 2016), Blaster (Sacco et al., 2020),	
Security and Monitoring	Heavy hitter, DoS	Elastic Sketch (Yang et al., 2018) and CMS-like	*Elephant* flows are allocated more accurate data structures.
	Quantile estimation	KLL (Karnin et al., 2016), DDSketch (Masson et al., 2019)	

TABLE 4.1: Network use cases that can exploit elephants/mice hints.

our pipeline a mechanism that periodically updates the model using newly sampled data to maintain the system performance stable over time. To the best of our knowledge, this is the first work proposing a complete data plane ML pipeline *(i)* detailing all its components, their deployment overhead, and a system prototype, *(ii)* evaluating the end-to-end pipeline performance on the downstream tasks, and *(iii)* demonstrating its long-term deployment effectiveness.

4.2.2 Use cases

As suggested by (Đukić et al., 2019; Mitzenmacher, 2021), getting early access to hints about the flow size may enhance a series of networking tasks which we survey in Table 4.1. In this chapter, we reference three use cases described below to support our claims with some concrete examples.

Flow scheduling. Flow scheduling refers to the process of efficiently allocating network resources to different flows (e.g., identified by TCP/IP 5-tuple). It involves determining the next packet to be forwarded and possibly prioritizing some flows. Leveraging flow size estimation has emerged as a promising approach to optimize flow scheduling algorithms in recent years. For instance, pFabric (Alizadeh et al., 2013) aims at minimizing the average flow completion time by prioritizing small flows over huge ones. However, it requires end-hosts to communicate flow residual length, which requires, in turn, applications and network devices modifications. An alternative strategy is pHost (Gao et al., 2019), which strives for optimal performance akin to pFabric, but without requiring network device modification. Unlike pHost and pFabric, DUMBO does not require the involvement of end-hosts to provide flow size hints. Yet, such a system could still prioritize smaller flows over larger ones, thereby offering scheduling performance similar to pFabric and pHost.

Flow packets IAT distribution estimation. A popular network measurement task for Service-Level Agreement is the estimation of packet inter-arrival time (IAT) quantiles. Precise IAT distribution monitoring is impractical given the amount of memory it would require, which calls for approximate measurement techniques. A well-known sketch for distribution estimation is DDSketch (Masson et al., 2019), which is allocated once per each flow under measurement. As previously described in Chapter 2, a DDSketch contains multiple buckets (more buckets lead to more accurate estimation) that correspond to different ranges of IAT values. Mice are composed of a few packets, which yield only a handful of IAT values to be inserted into the corresponding DDSketch buckets, thus leaving the majority of the buckets empty. Early knowledge of flow size can help dimension the number and the size of buckets to allocate per flow. Therefore, DUMBO employs flow size hints to use more and/or bigger buckets for elephants. To the best of our knowledge, we are the first to propose a learned DDSketch using flow size hints. Other sketches such as KLL (Karnin et al., 2016), and GKsketch (Greenwald and Khanna, 2001) can similarly benefit from flow size hints.

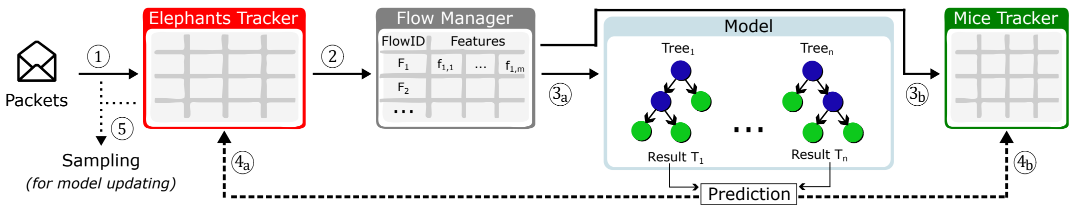


FIGURE 4.3: DUMBO system architecture.

Flow size estimation. Flow size estimation is a popular network monitoring task that is typically addressed by employing probabilistic data structures like the Count-Min Sketch (CMS, Cormode and Muthukrishnan, 2005). As introduced in Chapter 2, in a CMS, flow keys are hashed to identify flow counters that are shared across multiple flows. Intuitively, the estimation error of mice flows that share counters with elephant flows is particularly impacted. Recent variants of the CMS attempt to segregate elephants from mice and count them in separate data structures. For instance, ElasticSketch (ES, Yang et al., 2018), also presented in Chapter 2, uses a voting algorithm to decide whether a flow should be counted in the CMS or in a dedicated bucket. Parallel to this line of research, (Hsu et al., 2019; Du et al., 2021) explore the use of an ML model to make such a decision. These works show promising results evaluating the learned CMS offline, yet they do not take system implementation into account.

4.3 System Design

Our system is built around the concept that coarse flow size predictions in the data plane benefit several network tasks. For this reason, we aim at classifying flows based on their size, with reasonable confidence, and as early as possible within the first few packets of the flow. Late predictions would reduce the benefits of the ML-enabled data plane pipeline. In this section, we present the system design for integrating such ML hints in the data plane. Figure 4.3 presents the high-level DUMBO system design with its four components: *Elephant Tracker*, *Flow Manager*, *Model*, and *Mice Tracker*. In the following, we first introduce the processing pipeline, and then we detail the specific design of each component.

4.3.1 Pipeline overview

The first component in the processing pipeline (cf. Figure 4.3) is the *Elephant Tracker*, whose role is twofold. First, it collects precise monitoring data (e.g., packet count, timestamps, etc.) for the elephant flows. In addition to that, it serves as a cache for elephant predictions from the model. Upon receiving a packet (step ①), the system performs a lookup in the Elephant Tracker. If the flow is found therein, i.e., the flow was already predicted as an elephant, it is updated, and the processing pipeline ends. Otherwise, the packet is sent to the *Flow Manager* (step ②), which detects if the packet belongs to a new flow, e.g., using the SYN flag or relying on a bloom filter (Bloom, 1970). The primary role of the Flow Manager is to collect the features that need to be extracted from the first k packets of each flow. Once these features have been collected, they are sent to the *Model* (step ③a), triggering a prediction. If the Model predicts an elephant flow, the packet is sent to the Elephant Tracker, e.g., by packet recirculation in modern programmable pipelines, together with the information collected so far, and a new entry for this flow is created (step ④a). If the model predicts a mouse, the packet is sent to the *Mice Tracker* component instead (step ④b), where monitored metrics are stored in a compact probabilistic data structure, e.g., with sketches. In addition, the Flow Manager serves as a negative cache for flows predicted as mice: if a packet that does not belong to a new flow, and has no entry in the Elephant Tracker nor in the Flow Manager, it is directly sent to the Mice Tracker (step ③b). Similarly, any entry that gets evicted from the Flow Manager before collecting k packets is directly treated as a mouse without triggering any prediction. Note that the system continuously samples packets to trigger model updates when needed (step ⑤).

4.3.2 System components

Elephant Tracker The Elephant Tracker keeps track of flows that are predicted as elephants by the model. It can be implemented with a simple multi-level[1] and multi-entry[2] hash table. This hash table stores 6-byte flow fingerprints[3] and precise flow monitoring data for the task at hand (e.g., 3-byte counters recording flow length). Hash table and fingerprint collisions can be handled using existing Flow to ID mapping solutions such as (Zhao et al., 2021; Pontarelli et al., 2019; Barbette et al., 2020; Sengupta et al., 2022) that fit in modern programmable hardware like FPGA-based smart NICs and Tofino switches.

The Elephant Tracker is accessed whenever a new incoming packet is processed by the system. We first extract the flow fingerprint from the packet, then perform a lookup on the hash table for the flow entry, that is, we check if the flow was already classified as an elephant by the Model. If so, the data stored in the entry is updated according to the metrics extracted from the last packet (e.g., increment the corresponding counter recording flow length). The data stored in each flow entry might significantly vary based on the monitoring tasks performed by the system. For instance, in the case of measuring packets IAT, each entry of the Elephant Tracker features the timestamp of the last seen packet and a DDSketch to provide accurate quantile estimation. In this use case, the system updates the latest timestamp and uses the previous one to compute the IAT, then it updates the DDSketch accordingly. Insertions of new entries in the Elephant Tracker are dictated by the

[1]Multi-level refers to a hash table with h successive hash functions used to address h separate tables.
[2]Multi-entry refers to a hash table with multiple entries for a single addressable bucket.
[3]An efficient fingerprinting algorithm can be derived by hashing the 5-tuple flow identifier.

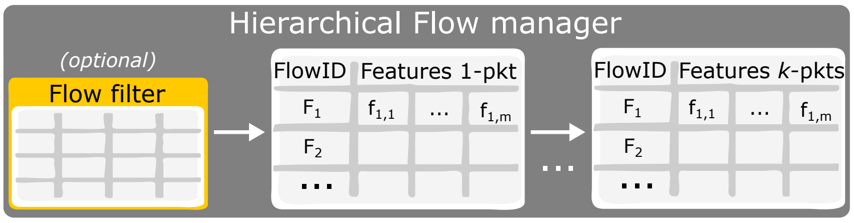


FIGURE 4.4: Architecture of a hierarchical Flow Manager. Flow entries move to the next stage when a new packet is collected.

Model, i.e., the system occupies a pre-reserved entry whenever a new flow is predicted to be an elephant. Notice that, due to the use of a hash table, such insertion may fail (that is, the entry in the hash table is already allocated to another flow). In this case, the system treats the new flow as a mouse and forwards its packets and metadata to the Mice Tracker. We note that the Model decision needs to be back-propagated to the Elephant Tracker; however, this operation is not possible in programmable switches, because the Elephant Tracker is the first component of the processing pipeline. Therefore, we need to recirculate a certain fraction of packets. Since this is only needed at insertion time, i.e., one packet per inserted flow, and since elephant flows account for only 1% of all the flows, we expect a negligible recirculation overhead due to the Elephant Tracker.

Flow Manager. The Flow Manager is responsible for collecting flow features from the first *k* packets of each flow that are necessary to get hints from the Model. Once the required features have been accumulated, the Flow Manager forwards them to the Model for inference. Similarly to the Elephant Tracker, the Flow Manager is organized as a multi-entry hash table, using the same 6-byte fingerprint as a key. The Flow Manager is accessed only when there is a miss in the Elephant Tracker. If this is the case, the system first checks if the packet belongs to a flow that has never been seen so far. To do so, a possible strategy is to exploit packet headers in case of TCP flows i.e., SYN, SYN-ACK flags. Alternatively, to accommodate non-TCP traffic, another strategy is to query an optional *Flow Filter* component as illustrated in Figure 4.4, which can be implemented with a Bloom Filter (Bloom, 1970). If the flow is new, it is inserted in the hash table in a free bucket or evicting an old flow. The new entry is initialized with the features extracted from the first packet and its timestamp. If the flow is not new and there is a match in the Flow Manager, the corresponding entry is updated with the latest feature values. When the *k*-th packet of the flow arrives, the system packages the collected features from the updated counters values, extracts the 5-tuple features from the packet header (TCP/IP ports and addresses), and forwards them to the Model for inference, also freeing the entry in the Flow Manager. Finally, if the flow is not new but is not in the Flow Manager, it is considered a mouse and directly forwarded to the Mice Tracker for further processing. This situation can occur in two cases: *(i)* the flow has been previously predicted as a mouse by the model and then evicted from the Flow Manager, or *(ii)* the flow

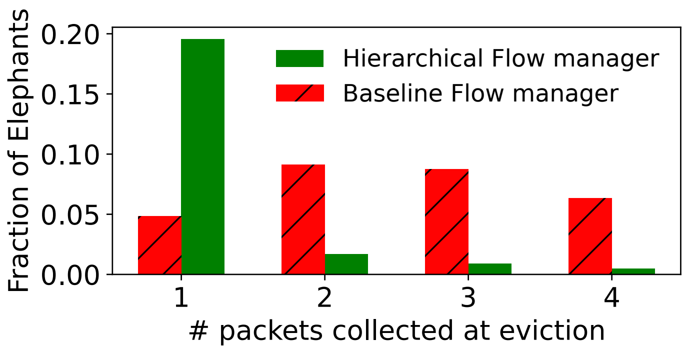


FIGURE 4.5: Simple vs. Hierarchical Flow Manager elephants evictions. Elephants discarded before reaching $k = 5$ packets.

has been evicted from the Flow Manager before reaching k packets. We remark that flow eviction occurs when there is a collision in the hash table and no free slots are available to store the new flow. In this case, we evict the least recently used flow entry in the bucket. In practice, after detecting the entry with the smallest timestamp within the bucket and its position, its place is taken by the newly inserted flow: the flow evicted is sent to the Mice Tracker and considered as mouse thereupon.[4]

The Flow Manager described above has the drawback of possibly evicting flows close to reaching k packets, i.e., just before model inference. To address this problem we organize the Flow Manager hierarchically as illustrated in Figure 4.4. The first table keeps track of flows that have accumulated one packet so far, the second table those that have accumulated two packets, and so on. Eviction is restricted among flows with the same amount of packets. In this way, flows close to reaching k packets are implicitly given priority over the others. Note that this hierarchical structure is compatible with programmable pipelines because when a hit occurs, the flow simply needs to be moved to the next level. Figure 4.5 compares a hierarchical Flow Manager with a simple one, using a CAIDA traffic trace with $k = 5$ and a fixed memory budget. In this simple example, the hierarchical version reduces early evictions of the elephants to 22% (32% with simple Flow Manager).

Model. The model is responsible for producing hints in the form of a binary elephant/mice classification by taking as input the flow features collected by the Flow Manager. We recall that, if a flow is evicted from the Flow Manager before accumulating k packets, it will never get to the Model and thus never get a chance to be correctly classified. Therefore, the Flow Manager plays a key role in the overall quality of the hints provided by the Model. Hence, we highlight the importance of carefully co-designing the Flow Manager and Model components to get the best performance.

[4]This can be done without recirculation in FPGAs by adopting a two-stage pipelined memory access: the first stage for reading the memory, the second for performing the update. This approach would be hard to implement in other popular programmable architecture, like Intel(R) Tofino, due to its constrained memory model, hence requiring packet recirculation.

Mice Tracker. The role of the Mice Tracker is to approximately monitor flows predicted as mice. Simply disregarding these flows would be inaccurate because some of them are actually elephants mispredicted as mice, while others did not get a model prediction due to early eviction from the Flow Manager. As a result, the Mice Tracker serves as a backup monitoring data structure, similar to the approach used in (Yang et al., 2018; Hsu et al., 2019). In our system, the Mice Tracker comprises one or multiple approximate data structures such as Count-Min Sketch for flow size estimation or compact DDSketches (Masson et al., 2019) for IAT distribution estimation.

4.4 Machine Learning Model

The goal of the DUMBO ML model is to classify flows according to their size as quickly as possible while maintaining high performance and low memory overhead. In this section, we first introduce the data, metrics and baselines used to compare our approach, and then we detail model design and sizing. Finally, we present an automatic update mechanism to ensure model robustness over time.

4.4.1 Benchmark setup

Datasets. To motivate our choices, we run model micro-benchmarks using the following traffic traces:

- CAIDA on 2016-01-21 13:00 – 13:59
  (*The CAIDA Anonymized Internet Traces Dataset* 2016)
- MAWI on 2019-04-09 18:45 – 19:44
  (*The MAWI Working Group Traffic Archive* 2019)
- UNI on 2010-01-12 20:00 – 22:29
  (Benson et al., 2010)

As (Hsu et al., 2019; Du et al., 2021), we label flow sizes above the 99th percentile as elephants (positive class) and the rest as mice (negative class). Similarly, we consider minute-defined measurement epochs to extract uni-directional flows from each trace. In Figure 4.6 we report the amount of flows and their size distributions for each trace, broken down by protocol. This analysis motivates us to deliberately exclude ICMP traffic from our dataset. Indeed, ICMP traffic features smaller flows and would only make the early classification task less challenging on MAWI. On the other hand, this exclusion would have almost no impact on CAIDA and UNI traces as they feature almost no ICMP flows. Therefore, we focus on TCP and UDP traffic in the remainder of this chapter, which represent $\approx$1M, $\approx$500K, and $\approx$5K flows per epoch for CAIDA, MAWI, and UNI respectively. Still, we report results including ICMP traffic in Appendix B.1.

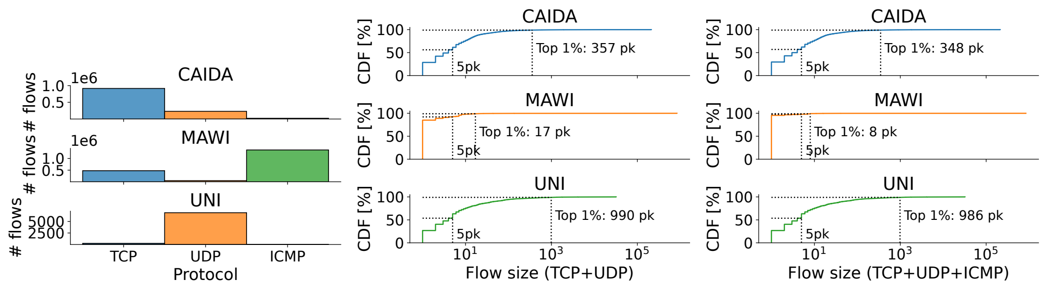


FIGURE 4.6: Traffic analysis for the 50th minute of each trace.

Metrics. The elephants/mice classification task is heavily imbalanced by design resulting in unequal consequences for mispredictions. It is important to note that binary classifiers typically output the *probability* that a sample belongs to the positive class. The classification result depends on a fixed threshold applied to this output. Tuning this probability threshold corresponds to selecting a trade-off between Precision and Recall. As a reminder, precision P and recall R take their values in the interval $[0,1]$ (higher is better) and are defined as:

$$P = \frac{T_p}{T_p + F_p}, \quad (4.1) \qquad\qquad R = \frac{T_p}{T_p + F_n}, \quad (4.2)$$

where T_p are true positives, F_p false positives and F_n false negatives. To ensure a fair comparison between classifiers irrespective of this arbitrary threshold, an appropriate metric is the Average Precision (AP) score. It is calculated as:

$$\text{AP} = \sum_n (R_n - R_{n-1}) \cdot P_n, \tag{4.3}$$

where P_n and R_n are the precision and recall for the n-th threshold. In a nutshell, the AP score is the average of precisions at each threshold, weighted by the increase in recall from the previous threshold. Thus, the AP score summarizes the trade-offs achievable with a given model by considering all its possible probability thresholds. It can be interpreted as the area under the Precision-Recall curve of a binary classifier.

Baselines. In our benchmark campaign, we compare against pHeavy (Zhang et al., 2021b) that consists in a pipeline of successive decision trees queried at various packet arrivals (e.g., 5-20) in order to filter out mice progressively until the model makes a final prediction. Hence, pHeavy can only predict elephants when the last model stage is reached (e.g., 20th packet). Additionally, pHeavy optimizes for high recall and high specificity (true negative rate) to guide model training. However, these metrics are insensitive to class imbalance, which is inherent to elephants/mice classification (i.e., 1:99 imbalance ratio by design in our settings).

4.4.2 Model design

Packet features. We choose Random Forests (RF) (Breiman, 2001) models which are typically preferred over more sophisticated algorithms (e.g., Deep Learning) for data plane implementation, due to their simplicity and interpretability (Busse-Grawitz et al., 2022; Lee and Singh, 2020; Xiong and Zilberman, 2019; Zheng et al., 2022b; Zheng and Zilberman, 2021; Zheng et al., 2022a; Akem et al., 2022). Ideally, the RF model should provide a prediction as early as possible i.e., at the first packet. In this case, the flow 5-tuple is the only feature fed to the model. As in (Hsu et al., 2019; Du et al., 2021), the 5-tuple is transformed into 97 binary features: 32 + 32 features for source and destination IPs, 16 + 16 features for source and destination ports and 1 feature for protocol (TCP or UDP in our case). Binary features have the advantage of simplifying the RF deployment in the data plane. Similarly, it is also desirable for a data plane implementation to have binary leaves only. In vanilla RF, each leaf is assigned an impurity value, which in practice equals the proportion of one class over the other among training samples that map to that leaf. Such values are provided as output by every tree at inference time and then averaged across the forest to output a class probability. This probability of the sample belonging to the positive class is then thresholded to output the final elephant vs mouse classification result. To simplify data plane implementation and avoid costly floating point operations, we fine-tune an appropriate probability threshold within the training pipeline, encode it directly in each tree leaf, and use them as votes for class prediction.

Flow features. While obtaining timely predictions from the first packet is attractive, this can limit the model's long-term performance. Therefore, we integrate additional flow features to better characterize their behavior. After analysis, we select four *aggregated features* collected from the first k packets of the flow, namely the *mean* and *standard deviation* of *packet sizes* and *inter-arrival times*. To do so, the Flow Manager stores a 2-byte timestamp of the last packet and four 2-byte counters for the aggregated features per each flow. These counters store the sum of the measured values $S = \sum_i x_i$ and the sum of their square $Q = \sum_i {x_i}^2$. After k packets, we compute the mean $\mu = S/k$ and standard deviation $\sigma = \sqrt{(kQ - S^2)/(k(k-1))}$. In conclusion, the k-packets model takes as input 101 features from a flow: 97 binary features for the 5-tuple and 2 + 2 aggregated features.

Evaluation. To validate our model design, we train the Random Forest on the first 5 minutes of traffic in CAIDA, MAWI, and the first 95 minutes in UNI (as this trace features fewer flows), and use the last 55 minutes for testing. The model hyper-parameters (number of trees, max depth, etc.) are obtained through random search with 2-fold cross-validation on the training minutes, optimizing for Average Precision (AP) score. This first training phase does not consider any model size constraint; we consider this model as an upper bound in terms of achievable performance with

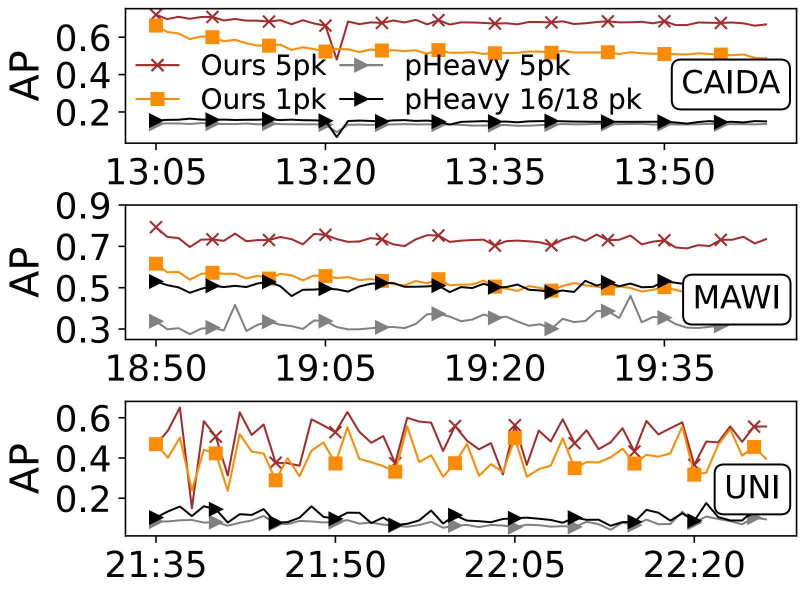


FIGURE 4.7: Model performance across time.

RF. Figure 4.7 compares models' performance on the test set constituted by the remaining 55 minutes of each trace. We remark that, as expected, adding aggregated features (with $k = 5$) leads to stable performance within the full test set. We also tested $k > 5$ but obtained stable performance starting from $k = 5$, hence, in the remainder of this chapter we use the 5-packet model. We remark that using only aggregated features (no 5-tuple) does not provide accurate predictions (AP score $\approx$20% lower, not reported in the figure) as the model cannot correlate flow characteristics with its source and destination. Finally, we remark that classification on MAWI appears easier than on the two other traffic traces. This is partly explained by their different flow size distributions: in MAWI $\approx$90% of the flows have less than five packets, while this represents only $\approx$50% of the flows in CAIDA and UNI. This also explains the higher performance of pHeavy on MAWI: when using its last pipeline stage at the 18th packet arrival in the model we trained, most mice have already been filtered out. The results on UNI indicate comparatively lower and less stable performance. This can be attributed to the limited amount of flows available for training and testing (i.e., $\times$100 less than CAIDA). To finalize our model for the ML-enabled data plane pipeline integration, we then use the last minute of the training set to empirically tune the probability threshold for elephants prediction. This threshold has a direct impact on the model Recall and Precision: the lower the threshold, the larger the number of flows predicted as elephants thereby favoring Recall over Precision and vice versa. Interestingly, this threshold enables us to control the fraction of flows that the model classifies as elephants, which is a crucial parameter for appropriately sizing the Elephant Tracker. Once the appropriate threshold is selected, it is encoded in all trees' leaves.

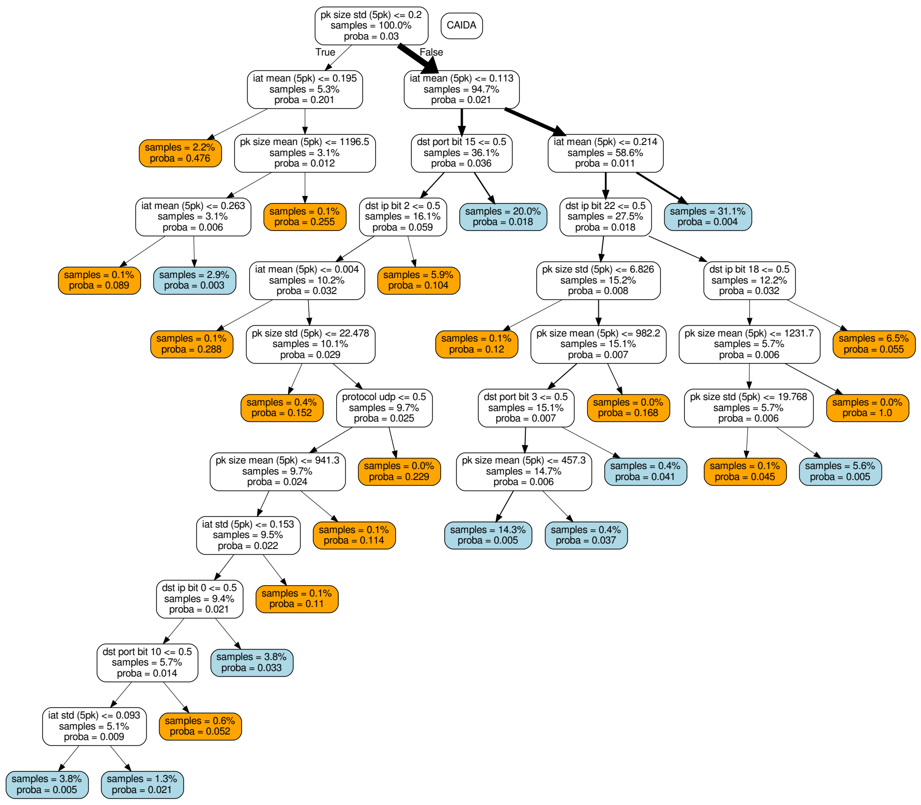


FIGURE 4.8: Simplified representation of the explainable tree extracted with TrusteeML (Jacobs et al., 2022) on CAIDA. Probability threshold is 0.043 for ≈20K elephants. Samples proportions may not sum to 100% because of top-k branches pruning.

Explainability. To interpret the trained model, we use the TrusteeML library (Jacobs et al., 2022) to extract a high-fidelity and low-complexity tree that explains the decisions of the model. TrusteeML is a framework that aims at identifying model under-specification issues such as shortcut learning or spurious correlations. It takes a model-agnostic approach to iteratively train various decision trees (students) that best mimic the model's decisions (teacher). Students are trained on various samples of the training set using bagging, and the misclassified samples are used to augment the initial training set at each iteration. The student with the highest fidelity is selected for top-k branch pruning to make it easily interpretable at the expense of performance. Each branch represents a decision rule that accounts for a certain fraction of the input samples, and only the top-k branches are kept after pruning. This overall process is repeated several times to select as final explainer the pruned student that is the most stable across runs (i.e., the highest mean agreement across pruned students). Figure 4.8 shows a simplified explainable tree extracted with TrusteeML. Note that this is a heavily-pruned version of the tree, that aims at interpretability at the expense of performance. When analyzing the model, we note that the four aggregated features we introduce rank first in terms of feature importance (measured by the number of samples the feature is used to classify).

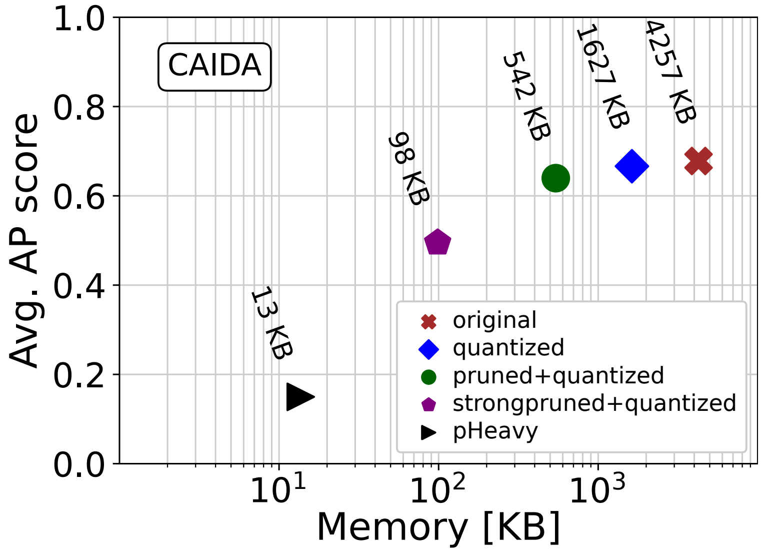


FIGURE 4.9: Model performance against size. In addition to features quantization, we further reduce the model size by iteratively removing the least performing trees (forest pruning).

4.4.3 Model size

In the data plane, memory is often limited, making it crucial to limit the size of the model size to minimize system overhead. Alternative approaches for elephant flow detection, such as pHeavy (Zhang et al., 2021b), employ tiny models but require ≈5 MB per 100K flows to store the required features because of late predictions e.g., at 18th packet. We argue that reserving this amount of memory is problematic in programmable network devices. Data structures dedicated to flow-size monitoring and management are typically allocated less than 1 MB of memory. For this reason, and to limit the system overhead, we target much smaller memory footprints for the full system, i.e., in the order of 1 MB. To evaluate the amount of memory required for the RF, as in (Monterubbiano et al., 2023b) we consider three alternative decision tree implementation approaches: *Match Action Table* (MAT), which exploits MAT available in programmable network devices, *full tree*, which involves storing fully-grown binary trees (including nodes not used by the actual decision tree), and *hybrid*, which consists in storing fully-grown trees up to a certain depth, and individual tree nodes from that level downward. We defer the reader to Section 4.5 for additional details about model system implementation and size estimation. Here we report only the model memory occupation obtained using the best strategy among the three.

As depicted in Figure 4.9, the original 5-packets model we obtain by training on the first 5 minutes of the CAIDA trace requires over 4 MB (red cross). It achieves an average AP score of 0.68 in the remaining 55 test minutes. This is influenced by unconstrained model parameters such as *(i)* the memory used to store feature values, and *(ii)* the number of trees composing the forest. To reduce the model size and meet the memory constraints of the data plane, we employ simple strategies with training speed and future model updates in mind. These strategies include quantization of feature values and forest pruning. For *(i)*, we discretize the aggregated floating point features using quantiles. This pre-processing step is part of the automated model training process. It creates an input feature vector with only binary values, making it easier to integrate the model into the data plane. Aggregated features quantization provides a $\times 2.6$ reduction in size for a 1.7% decrease in performance (blue diamond). For *(ii)*, we prune the forest by removing the worst-performing trees until a given size constraint is respected ($\approx$500 KB in our settings). This procedure allows precise control over the size-performance trade-off of the final model and is also part of the model training routine. We select a conservative final model of size 542 KB (green circle), i.e., achieving over $\times 7.8$ size reduction for an AP score of 0.64 (5.7% performance decrease). While stronger pruning led to a 98 KB model with a 0.50 AP score (purple pentagon), we stick with the 542 KB model to provide conservative results. We observed similar model size reduction on MAWI and UNI.

4.4.4 Model update

Active learning. Model maintenance is needed to account for drifts in traffic behaviors and ensure that the quality of the predictions is sufficient for benefiting downstream tasks. Since traffic patterns are dynamic, we expect model performance to degrade over time, for which we need to periodically update the model. In our case, the classification task requires new training samples i.e., input features describing the flow and the corresponding label, to learn and adapt to the newest patterns. The challenge lies in the sampling policy implemented to acquire fresh data points. On the one hand, we should not distort the flow size distribution to keep the training set representative of the real data. On the other hand, we need to constrain sampling to incur a reasonably low overhead. Following these guidelines, we draw from active learning and uncertainty sampling (Di Cicco et al., 2023; Settles, 2011) to determine which flows to sample. During each measurement epoch, the model itself is used to identify challenging flows that should be sampled. By aggregating votes from individual RF trees, we can quantify the uncertainty of the model prediction. Hence, we select for sampling flows with more than $k = 5$ packets that do not reach a voting agreement threshold, e.g., 0.6 and send them to the control plane. It is important to note that the agreement threshold is different than the one for model prediction. This sampling strategy has the advantage of selecting challenging flows with limited overhead, but it may also distort the actual data distribution. To alleviate this shortcoming, we additionally randomly sample 1% of the flows that reach the model, i.e., 1% of the flows with more than $k = 5$ packets. In our experiments, combining both uncertainty and random sampling leads to an overall sampling rate below 3% in the worst case. Sampled flows collected at the controller are appended to the initial training dataset to constitute a retraining buffer. To keep a stable re-training cost, the training buffer is a FIFO queue of a fixed size equal to the initial training dataset (e.g., 2.5M flows with at least 5 packets for CAIDA). Finally, we implement a drift

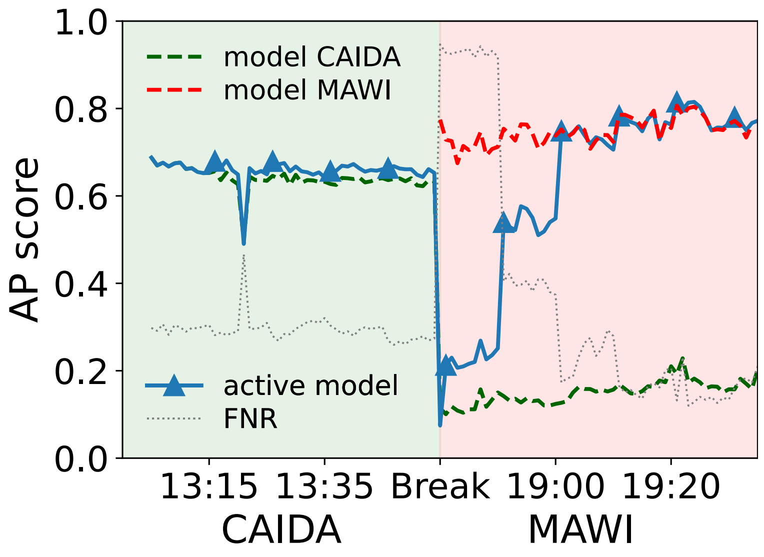


FIGURE 4.10: Model update stress-test.

detector to adapt quickly to sudden traffic changes. This detector triggers when the flow features sampled by the controller drift abruptly from one epoch to another, resets the retraining buffer, and triggers frequent model updates.

Stress test. Starting from a model trained on CAIDA, Figure 4.10 shows a 10-minute periodic active model update strategy in a worst-case scenario where, after 50 minutes of CAIDA test traffic (corresponding to a slow drift), the traffic abruptly changes to MAWI for the next 50 minutes (corresponding to out-of-distribution). Clearly, due to significant differences (e.g., IAT, IP addresses, flow size distribution), we expect a static model trained on CAIDA to be unfit to perform flow classification on MAWI traffic (dashed green line). However, even in such an extreme scenario, the active model update strategy (solid blue) quickly recovers to match the performance of a static model trained exclusively on MAWI data (dashed red). We also report False Negative Rate (FNR, i.e., the rate of elephants mispredicted as mice) over time, and show that it only requires two model updates to return to reasonable levels. We remark that model re-training does not need to be frequent and periodic, but can be triggered when performance on randomly sampled flows becomes unsatisfactory, i.e., after a long drift or an abrupt change. Finally, we highlight that the training time for a single tree on such a large dataset is ≈30 s on a single core from an Intel(R) Xeon(R) Platinum 8164 2.00GHz CPU, i.e., ≈1 minute training with 15 cores for a forest of 30 trees. Subsequent updates of the model weights on the FPGA target do not introduce any considerable extra overhead (≈1 ms in our case for rewriting the trees in the FPGA memory), hence allowing for line-rate updates in our system.

4.5 System Implementation

In this section, we first motivate a FPGA-based deployment scenario for DUMBO. Then, we detail three methodologies to implement a binary Random Forest in an FPGA-based SmartNIC and analytically derive the corresponding memory requirements. The full system prototype and evaluation in a real FPGA prototype is deferred to Appendix B.2.

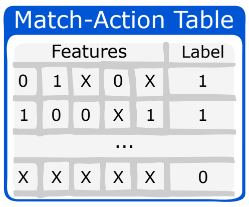
4.5.1 Deployment scenario

The DUMBO system leverages machine learning techniques to enhance the deployment of various network applications in high-speed networks such as the ones connecting data center servers. The introduction of SmartNICs and programmable switches has opened up opportunities to offload an increasing number of network applications from end-host CPUs (precious to cloud providers) to networking devices. While programmable switches could potentially incorporate the proposed ML-enabled data plane, we believe that FPGA-based SmartNICs are ideal candidates for implementing this technology due to their rapid adoption and flexibility. The first public example of the deployment of FPGA-based SmartNICs was presented in 2018 (Firestone et al., 2018), where a fleet of more than 1 million hosts is equipped with an ad-hoc SmartNIC. Today, key players such as Alibaba, Amazon, Baidu, Huawei, and Tencent now expose FPGAs to application developers in their data center infrastructures or as part of their service offerings. A comprehensive survey discussing these and other advances (Michel et al., 2021; Pontarelli et al., 2019; Ibanez et al., 2019; Rivitti et al., 2023) can be found in (Bobda et al., 2022). We argue that the processing power of FPGA-based SmartNICs, already prevalent in large data centers, can be efficiently utilized for ML inference.

4.5.2 Random Forest implementation

When implementing a RF in a programmable pipeline, i.e., FPGA in our case, it is critical to meet stringent throughput and memory size constraints. In particular, achieving one inference at each clock cycle is desirable for packet processing tasks as it greatly simplifies implementation. This calls for a pipeline architecture that can be realized using a *Match-Action Table (MAT)* implementation, a *full tree* representation, or a *hybrid* implementation. As previously mentioned, we restrict the analysis to binary features and binary leaves.

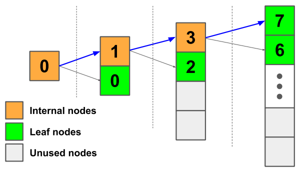
MAT implementation. MATs are commonly employed in the processing pipeline of network devices. Packet-processing tasks are implemented by a sequence of MATs, each one operating based on the result of the previous one. In (Lee and Singh, 2020), MATs were used to implement a decision tree by representing each level of the tree by a match-action stage. Alternatively, (Xiong and Zilberman, 2019) implements a decision tree using one stage per feature, to match it with all its potential values and



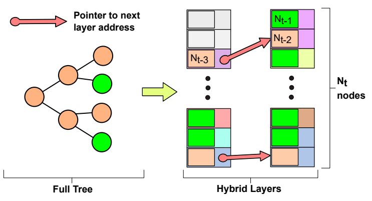
Match-Action Table

Features					Label
0	1	X	0	X	1
1	0	0	X	1	1
…					
X	X	X	X	X	0

(A) MAT tree optimized encoding.



(B) Full tree encoding.



(C) Hybrid tree encoding.

FIGURE 4.11: Visualization of different techniques for encoding a binary tree in the data plane.

output a feature code word that indicates a branch. An additional last stage is then required to match the concatenated features codes to the resulting leaf of the tree. This approach scales well for Random Forests (Zheng and Zilberman, 2021), where each additional tree in the forest only requires an extra MAT. In our binary classification case with binary input features, a decision tree can be implemented using a single MAT featuring one rule for each leaf (i.e., each traversal) of the tree. In each row, the matching key corresponds to the concatenation of the feature bits, while the action corresponds to the predicted label. The memory required (in number of bits) for a Random Forest RF implemented with MATs is then:

$$Mem_{MAT} = \sum_{t \in RF} L_t(2F + 1), \tag{4.4}$$

where t represents a tree from the Random Forest, L_t the number of leaves for t, F the number of binary features, and the factor of two relates to the encoding of a ternary value (i.e., "0", "1", or "do not care") for each feature, that requires two bits. We depict this approach in Figure 4.11a. A simple optimization consists of encoding only the leaves that correspond to one of the two classes. If one of the two classes is under-represented in the tree's leaves, this leads to a decrease in memory, since only the leaves corresponding to the less frequent class are stored in the MAT.

Full tree implementation. In contrast to the MAT implementation, the full tree approach reserves a separate memory block for each level of the tree. Each node in the tree is numbered sequentially starting from 1 to $2^{D_t} - 1$, where D_t is the tree depth, as depicted in Figure 4.11b. Consequently, traversing the tree for inference is straightforward: given any node i, its children in the next layer are found at the memory offset $2i$ and $2i + 1$ depending on the result of the parent split condition. However, this approach potentially leads to considerable memory overhead if the tree is sparse, as it assumes each tree to be fully grown. In our binary classification case with binary input features, each node is only required to store (i) one flag bit to distinguish between the two types of nodes (split or leaf); and (ii) the index of the feature to compare against (if it is a split node) or the predicted class label (if it is a leaf node). The memory required in bits for a RF is thus:

$$Mem_{FT} = \sum_{t \in RF} (2^{D_t} - 1)(\lceil log_2(F) \rceil + 1), \tag{4.5}$$

where D_t is the tree depth. We remark that, while this implementation can be more efficient in terms of memory occupancy than MAT in some cases, it has the drawback of requiring more memory accesses.

Hybrid implementation. The full tree representation may waste a significant amount of memory in the case of a sparse tree that is far from being a full binary tree. In our case, we observe that the trees composing the RF are often full only up to a certain level. Hence, the vanilla full tree implementation would lead to significant memory waste for the deepest layers that contain very few nodes (e.g., last layer in Figure 4.11b). We propose a hybrid approach that uses the full tree representation for the top layers and an *indexed encoding* for the bottom ones. We identify the maximum number of nodes N in any layer of any tree in the forest. We encode the first $M = \lfloor log_2(N) \rfloor + 1$ layers using the full tree implementation. For the remaining deeper and sparser layers, we only allocate N nodes. For each of these nodes, we store a pointer to their left child in the next layer, cf. Figure 4.11c. The right child is thus located at this address plus 1. The memory required in bits for a Random Forest RF can then be expressed as:

$$Mem_{Hyb} = \sum_{t \in RF} \underbrace{(2^M - 1)(\lceil log_2(F) \rceil + 1)}_{\text{full tree implementation}} + \underbrace{N(D_t - M)(\lceil log_2(F) \rceil + 1 + \lceil log_2(N) \rceil)}_{\text{indexed encoding}}, \tag{4.6}$$

4.6 Experimental Results

In this section, we detail the memory allocation for the system components and evaluate DUMBO on three tasks: flow scheduling, flow packets IAT distribution estimation, and flow size estimation. First, for every use case, we compare the end-to-end performance of DUMBO with state-of-the-art baselines and with a competitor ML pipeline based on pHeavy hints (Zhang et al., 2021b) obtained within the first 5-packets. Then, we analyze the deployment trade-off between model performance and pipeline memory overhead. We run our tests on a custom packet-level simulator using the last ten minutes of traffic traces from CAIDA, MAWI, and UNI datasets. We highlight that UNI features ≈5K flows per minute, which is significantly lower than CAIDA and MAWI with ≈1M and ≈500K flows per minute, respectively. However, we still include results on UNI, also used in (Zhang et al., 2021b), as it enables us to estimate the applicability of our approach on datacenter-like traffic.

4.6.1 Memory setup

We categorize memory allocation into *system*-related, and network *task*-related. This distinction stems from the observation that, while system-related memory is shared across the various network tasks that benefit from ML hints, task-related memory may significantly vary based on the use case.

System-related (DUMBO). We allocate 1 MB in total for Model, Flow Manager, and Elephant Tracker. We consider a conservative model size of 542 KB. The size of the Flow Manager depends on the number of simultaneous flows. We use the CAIDA training traces, which feature the highest amount of flows across our evaluation traces, to estimate this quantity. Accordingly, we allocate a hierarchical Flow Manager whose stages are broken down following the observed flow size distribution; this amounts to 290 KB, each entry requiring 6 bytes for the flow fingerprint, 2 bytes for the timestamp, and four 2-byte counters to collect the aggregated features. The size of the Elephant Tracker depends on the number of flows predicted as elephants, which is controlled in DUMBO by tuning the model probability threshold. The training pipeline uses the last minute of the training trace to empirically determine a threshold that sends 2% of the flows to the Elephant Tracker (i.e., ≈20K flows on CAIDA with a 20% overprovisioning). This amounts to 117 KB, each entry requiring 6 bytes for the flow fingerprint.

System-related (pHeavy). Our implementation of the pHeavy model only takes 13 KB. However, flow management requires much more memory due to extra features and eviction strategy (based on (Zhang et al., 2021b), it would take ≈50 MB on CAIDA traces, 1M flow per minute). To make comparison feasible, we replace flow tuples with our same fingerprints and apply a more aggressive eviction strategy that replaces flows after 5 seconds of inactivity, thus constraining pHeavy flow management to ≈1.6 MB in the worst case. For the Elephant Tracker, we use the same configuration as in DUMBO and similarly tune a model probability threshold.

Task-related. The base system is sufficient to run the scheduling use case. The two other use cases (IAT distribution and flow size estimation) however, require additional memory for measurements for both Mice and Elephant Trackers. In our experiments, we allocate 33 MB of memory for IAT distribution and 2 MB for flow

Component	System	Scheduling	IAT est.	Flow size est.
Model	542 KB	n.a.	n.a.	n.a.
Flow Manager	290 KB	n.a.	n.a.	n.a.
Elephant Tr.	117 KB	n.a.	1.26 MB	59 KB
Mice Tr.	n.a.	n.a.	Remaining mem.	Remaining mem.
TOTAL	**1 MB**	**n.a.**	**33 MB**	**2 MB**

TABLE 4.2: Memory allocation for DUMBO.

size estimation. Detailed memory allocations can be found in Table 4.2. In particular, the Elephant Tracker for flow size estimation requires one additional 3-byte counter for measurements, while the IAT distribution estimation use case requires a 2-byte timestamp and a DDSketch. Finally, the remaining available memory from the use case budget is fully allocated to the Mice Tracker. This consists of a Count-Min Sketch for flow size estimation or an array of coarse DDSketches for IAT distribution estimation. In the case of pHeavy, we remove from this remaining budget the extra memory it needs for flow management.

4.6.2 End-to-end performance

Flow scheduling. For the flow scheduling use case, we evaluate the performance of the system in terms of average normalized Flow Completion Time (FCT) as defined in (Alizadeh et al., 2013; Gao et al., 2019), i.e.:

$$normalized\ FCT = actual\ FCT / ideal\ FCT, \tag{4.7}$$

The ideal FCT is the time required to transmit a flow when the whole network fabric is composed of a single 10 Gbps link and there are no other flows, while the actual FCT is the measured flow completion time. We design a scheduling policy based on model predictions, assuming that the Flow Manager collects measurements from the first $k = 5$ packets of each flow. Each packet is assigned one out of three priorities from 0 (highest) to 2 (lowest). We assign priorities as follows: if the packet is one of the first $k - 1$, it is assigned to the highest priority queue; if the packet has no entry in the Elephant Tracker nor in the Flow Manager, we assume it is a mouse and assign it to the medium priority queue; if the packet belongs to a flow predicted as elephant, it is assigned to the lowest priority queue. Our priority queues schema is much less fine-grained than pFabric, where the exact residual flow size is used to determine packet priority. Yet, we argue that residual flow size is hard to obtain and that the proposed coarse-grained priority definition is easier to implement and sufficient to reduce FCT.

We evaluate our approach using YAPS (Kumar et al., 2016), a packet-based network simulator used in (Gao et al., 2019) to evaluate pHost and pFabric, two state-of-the-art baselines we compare against. We use the same network topology as in pFabric: the fabric interconnects 144 hosts through 9 leaf switches (top-of-rack) connected to 4 spine switches in a full mesh. Each leaf switch has sixteen 10 Gbps downlinks, and four 40 Gbps uplinks. The simulator randomly generates 1M flows based on user-provided CDFs, which we extract from CAIDA, MAWI, and UNI traces. To simulate ML hints (DUMBO or pHeavy), we use the model confusion matrix computed on the respective trace's test minutes. Then we instrument the confusion matrix to simulate authentic predictions, i.e., mimicking the model performance one could expect on the trace. This process allows us to reasonably evaluate ML-based approaches

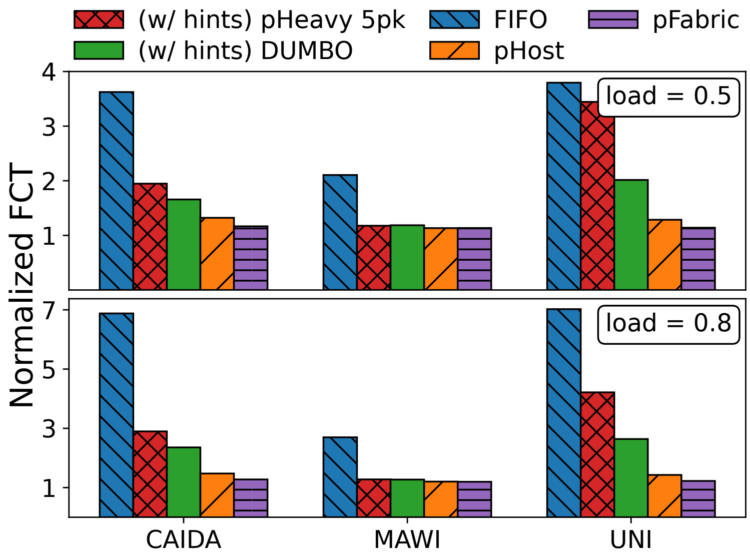


FIGURE 4.12: Flow scheduling slowdown.

at various loads (i.e., number of concurrent flows), even on the UNI dataset that contains fewer flows than the other traces. Results reported in Figure 4.12 show that pFabric and pHost perform near-optimally with an average slow down with respect to the ideal one within 1.13-1.28×, and 1.13-1.48× respectively. We remark that these two best-performing policies are often impractical, as both of them assume exact knowledge of the flow size by modifying end-hosts (this often requires changes to all the applications involved in the system). Instead, DUMBO only relies on hints extracted from the first $k = 5$ packets to assign scheduling priorities. Nonetheless, our system often provides performance close to pFabric and pHost (average slow down 1.19-2.64× with respect to the ideal), while operating without any dependence on end hosts and with much fewer priority queues. We note that both ML-enabled systems largely outperform host-agnostic FIFO scheduling. Comparatively, pHeavy scores an average slowdown 1.18-4.22×. We note that DUMBO outperforms pHeavy on all traces but MAWI, where all approaches perform on par. This is due to the flow sizes distribution of MAWI, which features over 90% of flows with less than 5 packets and 99% of flows with less than 16 packets, hence making it a less challenging trace for scheduling. Finally, we remark that DUMBO could manage more priority queues by leveraging the model prediction confidence, i.e., the number of trees votes in agreement. This optimization is impractical for pHeavy because it features a single tree per stage.

Flow IAT distribution estimation. To evaluate DUMBO performance on the IAT distribution estimation use case, we measure the 75th and 95th IAT quantiles and defer results on other quantiles to Appendix B.3. We compare against baselines that use the same 33 MB total memory budget to estimate the IAT distribution of all flows. The first baseline allocates one DDSKetch of 32×1-byte buckets for each flow, and the second baseline a DDSketch of 16×2-byte buckets per flow. For DUMBO and

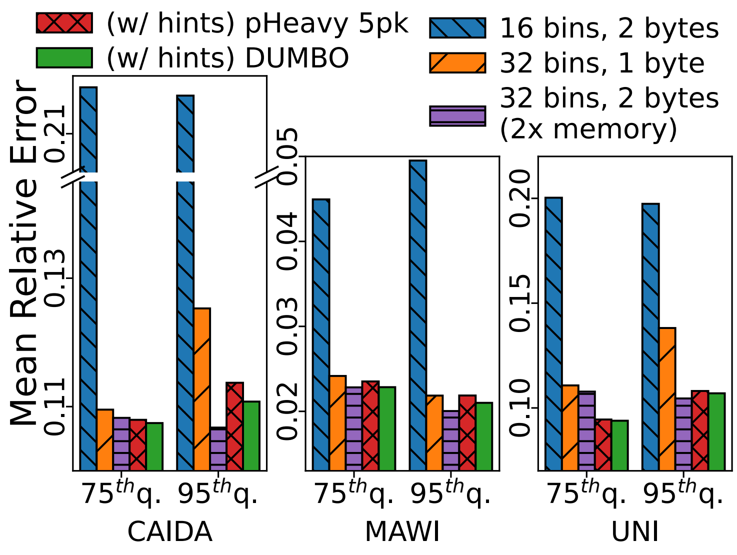


FIGURE 4.13: IAT quantile estimation error.

the pHeavy-based solution we use DDSketches of two different sizes based on the predicted flow class: 32×2 bytes buckets for elephants (high-accuracy DDSketches), and 31×1 byte buckets for mice (low-accuracy DDSketches). For these two hint-based approaches, we allocate 2^{15} high-accuracy DDSketches and 2^{20} low-accuracy ones, hence using the same amount of memory as for the baselines (33 MB). The rationale behind this strategy is that most flows do not require a high number of buckets for accurately estimating IAT, hence wasting memory with oversized DDSketches. We argue that longer flows i.e., Elephants, naturally lead to a higher number of IAT measurements, thus to a higher chance of overflowing counters.

The custom packet-level simulator we use to evaluate IAT and flow-size distributions (*DUMBO Simulator* 2024) replays the test traffic traces, and simulates the full set of DUMBO system components. When a flow exits the Flow Manager, model predictions are gathered from an ML model written in Python and wrapped to the simulation using ONNX (*ONNX. Open Neural Network Echange* 2017). Results in Figure 4.13 assess the advantages of hint-based approaches over both baseline configurations, in terms of mean relative estimation error, plotting the median over the 10 test minutes. As a reference, we also report an "ideal" performance when allocating accurate DDSketches for all the flows (32 2-byte buckets — i.e., doubling the memory). Remarkably, DUMBO provides errors close to this ideal setup while halving the memory requirements. Finally, the pHeavy-based solution reports a higher error compared to DUMBO, although it achieves decent end-to-end performance partly because of the heavy-lifting operated by the Flow Manager: the information it stores allows for exact quantile computation for a fraction of the flows with less than $k = 5$ packets. This explains why in some cases DUMBO and pHeavy even outperform the "ideal" baseline.

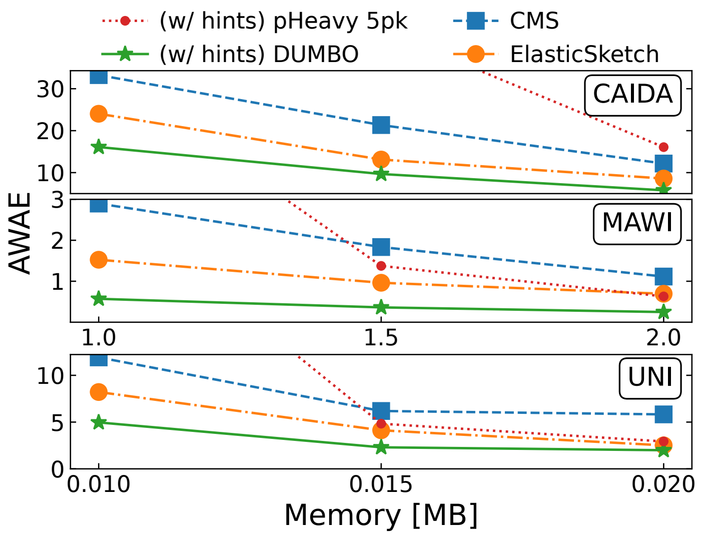


FIGURE 4.14: Flow size estimation error.

Flow size estimation. Finally, we evaluate the flow size estimation use case in terms of average weighted absolute estimation error (AWAE) as in (Hsu et al., 2019; Du et al., 2021). We size the ML-enabled pipelines and the two baselines to fit three different memory budgets, namely 1, 1.5, and 2 MB. The first baseline is a plain Count-Min Sketch (CMS) with 2 rows of 3-byte buckets, and the second is ElasticSketch (ES) (Yang et al., 2018), a popular sketch for flow size estimation. For DUMBO and the pHeavy-based solution, the Elephant Tracker is augmented with an additional 3-byte counter for each entry, to monitor exact flow sizes for the elephants. The approximate monitoring of short flows is delegated to the Mice Tracker which consists of a CMS of 2 rows, with 3-byte buckets in the first row and 2-byte buckets in the second row. This allows for a wider second row, while the first one serves as a backup in case of overflow and mitigates the effect of model misprediction. Similar to our hint-based approach, ES also dedicates exact counters for monitoring the size of elephants. However, ES does not use a model to provide hints but rather uses a dynamic mechanism called *ostracism* to determine which flows should be removed from the exact counters and moved to the mice CMS instead. For ES, we evaluate different repartitions for the memory allocated between the elephant exact counters and the mice CMS, reporting results with the best memory partitioning. Finally, due to the lack of flows in the UNI traces, we downsize the memory setup by 100× when simulating with this dataset.

As for the previous use case, we use our custom simulator (*DUMBO Simulator* 2024) with test traffic traces. Results averaged over the 10 test minutes are shown in Figure 4.14. We remark that, for all considered memory budgets, DUMBO outperforms both the plain CMS and state-of-the-art ElasticSketch. On CAIDA, where the number of flows to measure is substantial, the pHeavy-based solution exhibits poor performance at low memory budgets. This is explained by pHeavy misclassifying many mice (low precision) and saturating the Elephant Tracker; this leaves no space for legitimate elephants that would then pollute the mice CMS. The lower the memory budget, the smaller the mice CMS and the greater the impact of low-precision hints. Notably, on MAWI (or UNI), the relative (or absolute) small amount of flows with more than 5 packets leads to lower penalization for mice flows misclassified as elephants by both hint-based approaches. Additionally, pHeavy is impractical to deploy in a low-memory regime because it extracts more features from incoming packets, hence requiring much more memory for the Flow Manager (≈1.6 MB on CAIDA, exceeding the 1.0 MB system-related memory of DUMBO by ≈600 KB that are detracted from the task-related memory).

4.6.3 Impact of model performance on downstream tasks

In this subsection we analyze DUMBO end-to-end performance with respect to memory overhead and model quality. First, we investigate the impact of model size to identify the maximum memory overhead that can be tolerated. Second, we characterize the impact of model mispredictions on the performance of the downstream tasks. To do so, we use our custom simulator (*DUMBO Simulator* 2024) run with CAIDA traces and the system configuration introduced in the previous section.

Memory overhead. Figure 4.15 shows the impact of model size on the flow size estimation task when 1 MB of memory is available for the task itself. The figure plots the size of the ML model alone (top axis) and the overhead of the whole ML pipeline (bottom axis). DUMBO is denoted with a black star (542 KB model, 949 KB pipeline, 19 AWAE). The impact on the end-to-end performance also depends on the ML hints quality, here characterized in terms of the percentage of misclassified elephants (colored solid lines). Note that these misprediction rates are simulated by artificially modifying the model confusion matrix, thus, the resulting predictions do not depend on actual flow features but are instead stochastic. Our findings assess that the ML pipeline overhead greatly impacts the deployment feasibility and effectiveness on the downstream task. For instance, a model that wrongly classifies 20%

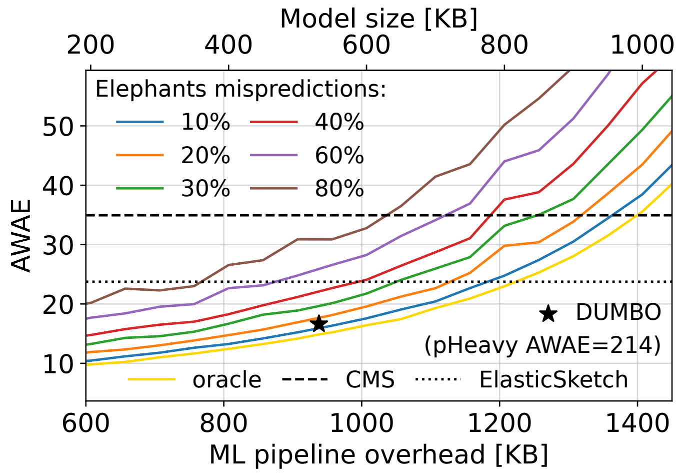
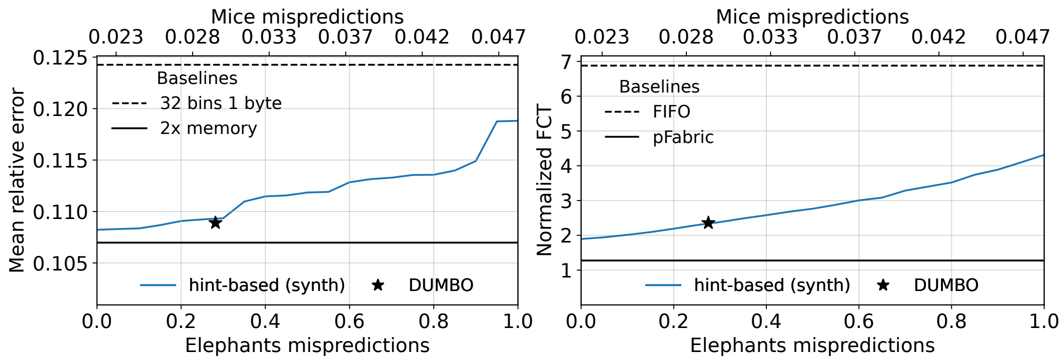


FIGURE 4.15: Impact of model size on flow size estimation (CAIDA).

of elephants as mice still brings more than 20% benefits in terms of AWAE compared to ElasticSketch as long as the ML pipeline fits 1 MB. However, for an ML pipeline overhead of 1.2 MB, this benefit disappears as the end-to-end error exceeds ElasticSketch's. Finally, with such hints quality, a pipeline requiring more than 1.35 MB is not beneficial even when compared to a simple CMS. Moreover, the plot shows that the end-to-end performance deterioration due to pipeline overhead is more severe than the deterioration due to hints degradation. For instance, a pipeline that fits 1 MB is better than ElasticSketch with up to ≈ 40% elephants mispredictions. Last, Figure 4.15 also reports oracle performance (i.e., a model without any mispredictions). Interestingly, even an oracle would not bring any benefit over the ElasticSketch and CMS baselines if the ML pipeline exceeds ≈1.2 MB and ≈1.4 MB respectively. In additional experiments in Appendix B.3, we notice that, counter-intuitively, the mice misprediction rate plays a more impactful role on the end-to-end performance: for a 1 MB pipeline, a model featuring 3.5% (4.5%) mice mispredictions performs worse than the ES (CMS) baseline. This result is due to the imbalanced nature of traffic: even a slight increase in mice misprediction rate translates into a huge overall number of mice flows being wrongly classified as elephants, hence preventing legitimate elephants from entering the Elephant Tracker.

Mispredictions costs. Figure 4.16a analyzes the impact of misprediction rates on the IAT estimation task. As per previous sections, for this use case, the memory allocated to the task itself is one order of magnitude larger than the pipeline size: therefore, we do not show how the trade-off changes varying the pipeline size, as effects are negligible. The plot reports mean relative error on the 95-th quantile (other quantiles in Appendix B.3) varying elephants misprediction rate from 0 to 1 and mice misprediction rate consequently, i.e., so that ≈20K flows are classified as elephants. Results show that a model featuring low elephant mispredictions provides end-to-end performance similar to the baseline that uses twice the same amount of memory i.e., allocating high precision DDSketches for all the flows. Notably, for this use case, pHeavy reports good performance, as its memory overhead has relatively



(A) IAT estimation, 95th quantile.

(B) Flow scheduling.

FIGURE 4.16: Impact of mispredictions on use cases end-to-end performance (CAIDA).

less impact on the IAT estimation task. In general, our tests show that up to ≈30% of elephants misprediction i.e., ≈3% of mice mispredictions, a hint-based system provides a relative error only ≈2% higher compared to the ideal configuration that uses twice the memory budget.

Last, Figure 4.16b analyzes the impact of mispredictions on the scheduling use case, simulating 1M flows with a load of 0.8. Similar to the previous use cases, the plot reports the normalized FCT when varying the elephants misprediction rate between 0 and 1 (and mice misprediction rate accordingly to classify ≈20K flows as elephants). We recall that our system considers only three priority queues (one for each predicted class and one for the Flow Manager), in contrast to pFabric which considers each flow's residual size to assign fine-grained priorities. As a consequence, even a perfect binary hint-based system that correctly classifies all elephants cannot reach the performance of pFabric. Yet, DUMBO only increases FCT by 1.85× compared to pFabric (solid black line), without any end-host involvement. Additionally, we note that the model ability to detect elephants (recall) is more important than its precision in this three-queues setting. This explains the FCT gap between pHeavy and DUMBO on CAIDA (+28% recall), which is further visible on UNI (+56% recall).

4.7 Conclusion

In this chapter, we presented DUMBO, an end-to-end system design to generate and benefit from approximate flow size predictions in the data plane. These predictions are generated using a lightweight ML model based on custom Random Forests. We investigated the model training and update routine to enable accurate and stable classification of incoming flows into elephants or mice, using both packet-level and flow-level features aggregated from the first k packets. Furthermore, we carefully studied the data plane implementation feasibility of the components at play in an ML-enabled pipeline and analyzed the various trade-offs to consider. Finally, we evaluated the proposed system on three network tasks (flow scheduling, inter-arrival time estimation, and flow size estimation). Thanks to its good hints quality and careful data plane pipeline design, DUMBO is more efficient compared to both classic and ML-enabled data plane solutions in the state of the art.

Related publications

Raphael Azorin, Andrea Monterubbiano, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi (Mar. 2024). "Taming the Elephants: Affordable Flow Length Prediction in the Data Plane". In: *Proceedings of the ACM on Networking* 2.CoNEXT1. DOI: 10.1145/3649473

Andrea Monterubbiano, Raphael Azorin, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi (2023b). "Memory-Efficient Random Forests in FPGA SmartNICs". In: *Companion of the 19th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT Posters '23. Paris, France: Association for Computing Machinery, 55–56. ISBN: 9798400704079. DOI: 10.1145/3624354.3630089

Andrea Monterubbiano, Raphael Azorin, Gabriele Castellano, Massimo Gallo, and Salvatore Pontarelli (2022). "Learned Data Structures for Per-Flow Measurements". In: *Proceedings of the 3rd International CoNEXT Student Workshop*. CoNEXT Student Workshop '22. Rome, Italy: Association for Computing Machinery, 42–43. ISBN: 9781450399371. DOI: 10.1145/3565477.3569147

Chapter 5

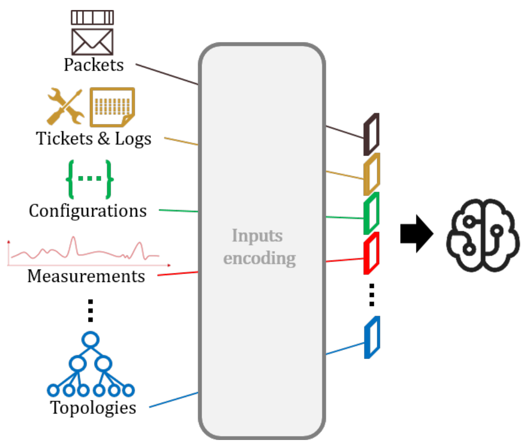
Traffic Representation Learning for Network Measurements

In this chapter, we envision *ML-based measurements* systems, using Deep Learning approaches to extract rich traffic representations. We organize our contributions to address (i) the encoding of network data into representations that facilitate knowledge extraction, and (ii) the design of a unified Deep Learning model that serves several tasks simultaneously. First, we remark that networking activity is captured by a mixture of various modalities, e.g., numerical data from measurements, free text logs history or categorical hosts identities. This makes networking tasks particularly interesting from a representation learning perspective. Hence, we propose a systematic multi-modal approach to learn representations from network data. We exploit recent advances in language modeling to encode network *entities* and *quantities*. We demonstrate the superiority of this bi-modal approach on two toy examples: ISP clickstream identification and terminal movement prediction in WLAN. Second, we advocate for a Multi-Task Learning paradigm to exploit potential synergies among networking tasks that are related to one another, e.g. measurements. To approach this challenge, we develop task affinity metrics aiming at grouping cooperative tasks together while separating competing tasks, *a priori*. Through an extensive empirical campaign, we assess the relationship between actual Multi-Task Learning performance and six task affinity estimation techniques. As a first step, we conduct this benchmark in the Computer Vision domain, taking advantage of well-established tasks, datasets and models. While no affinity metric is perfectly predictive of Multi-Task Learning performance, some can be more indicative than others to hint at beneficial task groupings, leading to increased overall performance and reduced model costs by mutualizing computation.

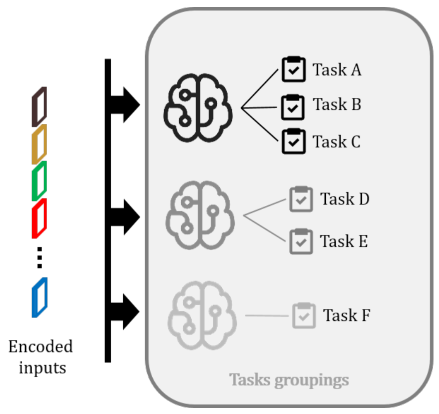
5.1 Introduction

Traffic measurements are key for network management as testified by the rich literature from both academia and industry. At their foundation, measurements rely on transformation functions $f(x) = y$, mapping input traffic data x to an output performance metric y. Yet, common practices adopt a bottom-up design (i.e., metric-based) which leads to invest a lot of efforts into (re)discovering how to perform such mapping and create specialized solutions. For instance, sketches are a compact way to extract traffic properties (heavy-hitters, super-spreaders, etc.) but require analytical modeling to offer correctness guarantees, and careful engineering to enable in-device deployment and network-wide measurements. Rather than relying on network experts domain knowledge, Deep Learning offers algorithms to learn $f(x) = y$ mappings. Artificial Neural Networks (ANNs) act as universal functions approximators as they can learn complex input-output data relationships with automatic feature extraction. Thus, contextualized to networking tasks, Deep Learning (DL) could empower a top-down design (i.e., model-based), potentially fostering automation and generalization. Yet, we claim that its application to networking is still in its infancy and we identified two research questions that require more work in the community.

First, considering the nature of network traffic, (i): *What is the appropriate representation of network traffic to facilitate knowledge extraction?* A DL model approximates a function mapping an input x to an output y. Under the hood, this mapping is operated via a double transformation. First, the raw data x is projected into a latent space by the model backbone, which does most of the knowledge extraction. Second, points in the latent space are transformed into the final outputs y thanks to the model head, which is typically a simple fully connected layer. These two transformations compose the $f(x) = y$ mapping. Conceptually, the latent representation is expected to capture latent properties. Despite being automatic, the effectiveness of the feature extraction operated by DL algorithms highly depends on the input representation. Specifically, the input should be presented in a way that stresses the salient characteristics deemed helpful for the mapping. For example, images used in Computer Vision (CV) are represented as static quantities grids of fixed size, allowing Convolutional Neural Networks to extract patterns at different scales in the picture (low-level, mid-level and high-level features by exploiting spatial locality). Words in Natural Language Processing (NLP) are embedded in a vectorial space to capture their semantic and syntactic relationships, and encoded sequentially for further processing by an ANN. Similarly, in networking, traffic patterns may emerge when capturing properties across several packets or events presented in sequences. Thus, we need to adapt the DL workflow to the specific characteristics of traffic data (Wang et al., 2018). In networking, data is typically abundant and captured in various modalities. For example, from raw traffic (e.g., PCAP), network activity may be characterized by quantitative measurements (i.e., time-series), related to



(A) Challenge (i)



(B) Challenge (ii)

FIGURE 5.1: Challenges for a unified Deep Learning measurements pipeline. **(left)** Encoding various data sources and modalities to facilitate knowledge extraction. **(right)** Sharing representations across related and cooperative tasks.

various entities (e.g., flows, devices, links). Alternatively, network activity may be recorded into logs and tickets (i.e., text) and network infrastructure may be described by topologies (i.e., graphs) and configuration files (i.e., structured data). This calls for a systematic multi-modal representation learning strategy in order to harness the potential of all data sources available to solve network tasks. We illustrate this challenge in Figure 5.1a. Thus, question (i) calls for a deeper investigation of the design space.

Second, considering generalization, (ii): *How to design a unified pipeline that serves several networking tasks?* This suggests that parts of the learning process might be shared across different networking tasks, e.g., measurements or their applications. The baseline DL approach consists in a single pair of model backbone and model head, trained by a loss minimization objective between a single target (e.g., a measurement) and its prediction. To this extent, this kind of supervised ANNs act as feature extractors for Single-Task Learning (STL). CV and NLP demonstrated that it is possible to reuse ANNs knowledge across related tasks, e.g., with transfer learning. Alternatively, the feature extraction process performed by ANNs can be designed to learn several mappings (i.e., several measurements) at the same time with Multi-Task Learning (MTL). Network operators may be interested in mutualizing the learning process to amortize training, inference and maintenance costs across several related measurement tasks. MTL would enable operators to derive multiple measurements from a single common representation of the input traffic, thus reducing the burdens of managing several individual models. To accommodate this need, a traditional Multi-task Learning process considers several model heads plugged on the same model backbone. Each head takes care of a specific target mapping and the latent space construction is guided by concurrent supervised learning objectives. The intuition behind this approach is that one task may benefit from the knowledge learned during the training of other related tasks. However, this is not a guarantee and, as MTL success is dependent on various factors, this knowledge transfer can even be detrimental to some tasks (Caruana, 1997). In particular, to prevent negative transfer across tasks, it is critical to group cooperative tasks together and separate competing tasks (Standley et al., 2020). This particular challenge is depicted in Figure 5.1b. Hence, question (ii) calls for actionable methods to uncover optimal tasks groups in order to reap the benefits of MTL.

Contributions In this chapter, we first advocate for a multi-modal network data representation, modeling sequences of corresponding *entities* and *quantities*. We draw from recent NLP advances by repurposing the Word2Vec methodology to extract entities meaning from their co-occurrence in sequences. We demonstrate the superiority of this multi-modal approach over the traditional unimodal one on two challenging examples: ISP clickstream identification and terminal movement prediction in WLAN. Second, we empirically analyze synergies between tasks learned together, aiming at grouping beneficial tasks and separating detrimental ones. We present an empirical comparison of six task affinity metrics and introduce an original proposal. Unfortunately, beyond traffic classification (Wang et al., 2022a; Aceto et al., 2019; Draper-Gil et al., 2016), network modeling lacks large public datasets of real network activity for multi-task learning. Instead of turning to simulated data, we conduct this benchmark on the reputed Taskonomy Computer Vision Multi-Task dataset. We show that the actual MTL performance does not correlate well with any of the metrics surveyed, while some techniques can still yield insights prior to joint training.

Organization In Section 5.2, we first deal with question (i), by introducing a systematic multi-modal representation learning methodology for networking data. We detail and evaluate this strategy against standard uni-modal approaches, i.e., learning from network quantities alone. Then, in Section 5.3, we tackle question (ii), and expose the details of the task grouping problem for Multi-Task Learning as a first step. Finally, we present several task affinity estimation techniques for joint learning and we benchmark them in an empirical campaign.

5.2 Representation Learning for Network Data

In this section, we propose a first approach to address research question *(i) What is the appropriate representation of network traffic to facilitate knowledge extraction?* Learning the right representations from complex input data is the key ability of successful Deep Learning models. The latter are often tailored to a specific data modality. For example, Recurrent Neural Networks (RNNs) were designed for sequences of words, while Convolutional Neural Networks (CNNs) were designed to exploit spatial correlation in images. Unlike Natural Language Processing and Computer Vision, each of which targets a single well-defined modality, network ML problems often have a mixture of data modalities as input. Yet, instead of exploiting such abundance, practitioners tend to rely on sub-features thereof, reducing the problem to a single modality for the sake of simplicity. In this section, we advocate for exploiting all the modalities naturally present in network data. As a first step, we observe that network data systematically exhibits a mixture of *quantities* (e.g., measurements), and *entities* (e.g., IP addresses, domain names, etc.). Whereas the former are generally well exploited, the latter are often underused or poorly represented (e.g., with one-hot encoding). We propose to systematically leverage language models to learn entity representations whenever significant sequences of such entities are historically observed. Through two diverse use-cases, we show that such entity encoding can benefit and naturally augment classic quantity-based features.

We first motivate our multi-modal approach in Section 5.2.1, then we abstract our bimodal approach in Section 5.2.2. We formalize our processing pipeline in Section 5.2.3 and apply it to two illustrative use cases in Section 5.2.4. Finally, we show supporting examples from the literature in Section 5.2.5 and discuss future opportunities in Section 5.2.6.

5.2.1 Motivation

In this subsection we first motivate the use of Deep Learning for automatically extracting representations from data, drawing example of its success in various domains (e.g., images, text). Then, we argue that this approach can be adapted to the particular multi-modality of network data.

Representation learning. Deep Learning's success is mainly due to its ability to learn good representations from complex unstructured data. Such ability is a fundamental aspect of intelligent agents, both artificial and biological. The representation learning ubiquity is perhaps best witnessed by the striking similarities between features learned by Artificial Neural Networks and biological brains. Two representative examples are visual and spatial representations. Decades after the discovery of simple and complex cells by 1983 Nobel prize winners Hubel and Wiesel (Hubel and Wiesel, 1968; Hubel and Wiesel, 1959), it was found that ANNs learn similar

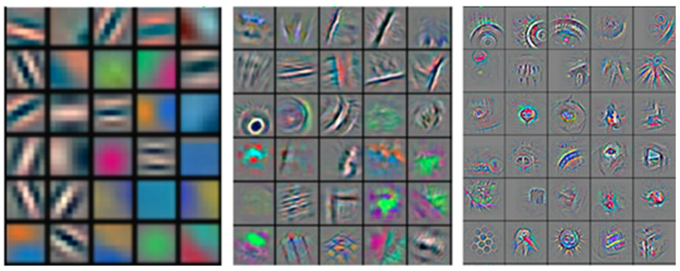


FIGURE 5.2: Visualization of features in a fully-trained CNN model. Adapted from (Zeiler and Fergus, 2014).

simple-to-complex representations (Zeiler and Fergus, 2014; Kruger et al., 2012; Cichy et al., 2016) as illustrated in Figure 5.2. The same applies for the 2014 Nobel prize winning discovery of Place and Grid cells (Hafting et al., 2005; Franzius et al., 2007), for which some neurons encode places and space representations. Similar representations were found in artificial agents that learn how to navigate (Banino et al., 2018; Cueva and Wei, 2018). Advances in NLP also corroborate the importance of learning good representations, by first pre-training neural networks on large unlabeled datasets with self-supervised tasks, followed by per-task fine tuning with few labels – which is behind the success of GPT-3 in few-shot learning (Brown et al., 2020). A similar process proved crucial for few-shot image classification, where learning good representations or embeddings, followed by training linear classifiers outperformed state-of-the-art few-shot methods (Tian et al., 2020).

Casting these observations to networking, to fully exploit DL potential, it seems necessary to put more focus on representation learning for network data. This is all the more important, given the abundance of unlabeled data generated and collected by networks. Such appeal is however immediately moderated by the complexity of network data, namely its *multi-modality*. Indeed, the most prominent advances in DL have been obtained on classic single modalities. From the perspective of input data, NLP takes sequences of categorical variables as input and CV takes as input pixel values stored in fixed-size matrices. Additionally, within the same language, words have a coherent meaning across contexts and corpora. The same applies to visual features which are universal across domains to some extent. This is far from being the case in network data which is way more dynamic and heterogeneous (including multi-variate timeseries, flow and system logs, topologies, routing events, etc.) and where identifiers may have a more local significance. Lacking a universal network data representation, DL has been applied to network problems in a rather opportunistic way focusing on a specific modality, or handcrafting input features. On the opposite side, each classic modality in mainstream DL tasks has its own research community. For example, CV heavily relies on variations of CNNs, e.g., AlexNet (Krizhevsky et al., 2017), ResNet (He et al., 2016) or MobileNet (Howard et al., 2017), to handle images tasks such as classification or segmentation. Modern NLP models instead take as input words and sub-words vector representations, pre-trained using word embedding techniques, e.g., Word2Vec (Mikolov et al., 2013b) or ELMo (Peters et al., 2018), on large corpora of raw text. Sequence to sequence models (Long-Short Term Memory, Hochreiter and Schmidhuber, 1997 and Transformers, Vaswani et al., 2017) are then often applied on such sequences of vector representations to solve a language task, e.g., classification or translation.

Network data bi-modality. As such, a legitimate yet challenging question emerges: what is the representation learning strategy that is best fit for the various network data modalities? It is exactly to answer this question that we call for research arms in this section. We believe that in order to take full advantage of emerging DL techniques, the networking community must rethink its "retina" i.e., input data format, and "visual cortex" i.e., representation learning strategy, used to extract knowledge from the input. Even assuming that for each modality there exists a different learning strategy, it is unlikely that each of them requires a new DL discipline. Alternatively, and more realistically, one could map each of the existing network modalities to the best-fit existing representation learning technique.

Taking a first principled step beyond uni-modality, we remark the existence of a natural dichotomy in network data, where we identify two network data types: *quantities* which are measured features such as numbers of packets, bytes, etc. and *entities* (or categorical data in ML terminology) which instead range from the named objects that relate to these measurements e.g., source IP, user id, to various attributes or events' names e.g., "interface down". We argue that language model pre-training is an appropriate tool to learn a representation for such entities, emphasizing on the similarity between sequences of co-occurring network traffic entities and sequences of words in natural language. Indeed, similar to natural language, the order and context in which network entities co-appear in network logs is often not arbitrary, hence patterns could be learned from it using appropriate language models (Mikolov et al., 2013b; Peters et al., 2018). Accordingly, we propose to leverage language model pre-training to learn vector representations, also known as *embeddings*, whenever (i) significant sequences of such entities are historically observed and (ii) these entities are consistently named across time and space.

Throughout this section, we refer to this network data dichotomy as entity-quantity *bi-modality*, that we explore as a first principled step towards network data multi-modality. In particular, we advocate for the need to use language model pre-training, such as word embeddings, to learn rich entities representations. The latter can then be simply concatenated with quantities or their representations before performing a DL task. We illustrate our proposal with two toy cases: (i) clickstream identification, where entities are sequences of domain names that carry a semantic meaning, and (ii) WLAN terminal movement prediction, where entities are access points identifiers that are not expected to have any semantic.

5.2.2 Network data representation with word embeddings

As a first step, we narrow the scope to a representative family of network data. We then provide some background on word embeddings used as a language model pre-training method, illustrating why and under which conditions they are suitable to deal with network data.

Entities and quantities in network data. While producing a thorough taxonomy of network data types is a challenging and useful target, it is outside the scope of this chapter. Instead, as a first step, we simply notice the difference between two main families of data types, for which a unified representation learning methodology could be devised. As argued earlier, network data include a mixture of entities and quantities. Quantities represent telemetry derived by measurement apparatus, while entities are abstract objects often related to them. The latter are typically described by names assigned by humans e.g., trouble tickets, IP addresses, domain names, host identifiers, etc., hence carrying a semantic meaning. We further argue that sequences of entities carry precious information encoded in the non-arbitrary order in which the elements appear in the sequence i.e., the "context". We advocate that such sequences must be systematically leveraged, and that NLP self-supervised pre-training is an appropriate representation learning technique. We also acknowledge that not all network data is sequential or measured over time i.e., network topologies. However such data often pertains to entities e.g., node names or identifiers, for which sequence data is abundant e.g., routing advertisement, in which case our proposed representation learning guidelines are still applicable to some extent.

Word and character embeddings. Oversimplifying, the closest problem in DL is sequence modeling in NLP – words are nothing more than sequences of entities that follow each other. To perform a DL task, words must be first transformed into a numerical representation. This can be naively done using integer or one-hot encoding. Modern NLP models instead, either build or take as input word vector representations obtained through self-supervised pre-trained models created from large text corpora. A well known word embedding technique, which we use in this section for its simplicity, is Word2Vec (W2V, Mikolov et al., 2013b). W2V transforms each word into a high dimensional vector, hence embedding it into an hyperspace. In practice, word embeddings are sometimes complemented with sub-word or character level embeddings (Kim et al., 2016; Chen et al., 2015), i.e., with separate vector representations for handling out-of-vocabulary words, misspellings, etc.

As presented in Chapter 2, W2V is a simple neural network with one hidden layer (whose dimension is the embedding vectors size), trained to predict a target word from its neighboring context, or vice versa. Word2Vec thus does not need expensive or human-made labels, but rather cheaply and automatically builds labels to supervise the training from the sequence data itself, thus it is self-supervised. First, all words are encoded using one-hot encoding, resulting in a one-hot vector of the size of the vocabulary in which each position represents one word. The neural network is then trained on large amounts of sequences of words from which `<target, neighbor>` pairs are extracted for training. At the end of the training, for each position in the one-hot encoding, the learned weights of the hidden layer are used to form the vector representing the corresponding word. Two main parameters hence influence the learned representations: the size of the embedding layer, and the size of the context window from which pairs are built.

Although trained to predict the missing words in a sequence, vector representations learned by word embeddings exhibit interesting properties. A well known example is the ability to extract semantic relationships by doing simple arithmetic operations on vectors e.g., King - Men + Woman = Queen. Another popular example is that vector representations of different languages exhibit strikingly similar structures, such that with few adjustments one can observe that words with similar meaning fall in the same positions in each language vector space (Mikolov et al., 2013a). This observation opened the way later to self-supervised language translation using only single-language corpora (Lample et al., 2017).

Conditions to apply language model pre-training. In NLP, Word2Vec and language model pre-training do not impose particular conditions the language must satisfy. However, moving away from natural language to any arbitrary sequence of named entities, we believe that at least two conditions must be satisfied. The foremost is the *(i) naming consistency*. Like words in natural language, network entities are expected to keep the same meaning. We note however that exceptions may exist, e.g., the word "set". As such, contextual word embeddings like ELMo (Peters et al., 2018) have been devised to solve this problem. Moreover, we require corpora *(ii) stability*: while this is true in natural language as adding a new word to the vocabulary is infrequent, observing a new entity in network data is rather frequent. Drawing the proper conclusion from the above conditions, we infer that sequences of entities containing, e.g., non-consistently anonymized IP addresses, are not suitable for entity embedding. Instead, entities that are named consistently and relatively stable over time are good candidates.

5.2.3 Bi-modal pipeline for network entities and quantities

In this subsection we sketch a generic bimodal pipeline, consisting of four steps namely *Pre-training*, *Sample selection*, *Training* and *Inference*, as shown in Figure 5.3.

Pre-training. Similarly to self-supervised pre-training, the first phase of the proposed pipeline consists in leveraging huge amounts of unlabeled data to learn relevant representations. As shown in the leftmost part of Figure 5.3, the pipeline takes sequences of unlabeled network data as input. Note that this pre-training phase is segmented by data type, i.e., entities and quantities, each being encoded with the

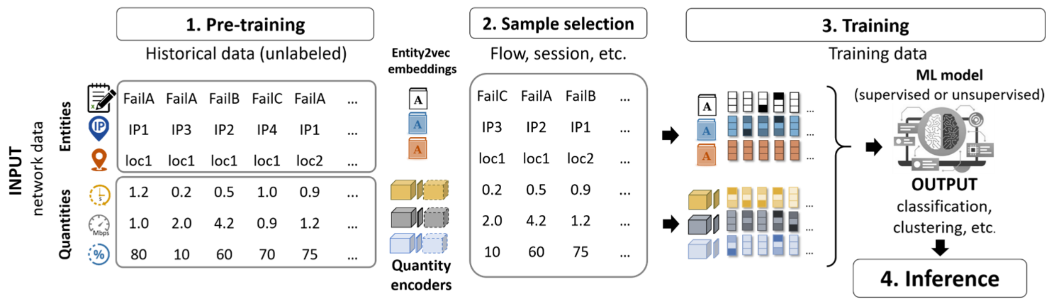
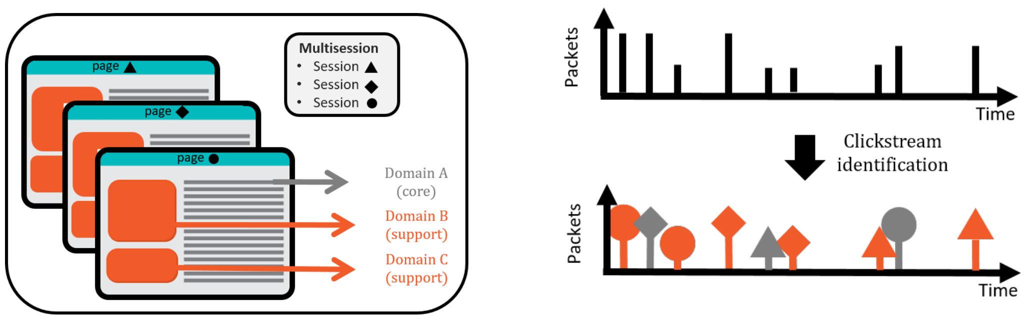


FIGURE 5.3: Generic bi-modal pipeline for network data representation learning.

most suitable representation learning model. Entities representations are learned using Word2Vec, per entity type. For example, sequences of IP addresses are used to train an *IP2Vec* model that embeds each IP address in a vector space. Similarly, sequences of access points are fed to an *AP2Vec* model to learn a representation of each access point, etc. Quantities representations may instead be learned with (Variational) Auto-Encoders (VAEs, Kingma and Welling, 2014) or other self-supervised learning methods. At the end of the pre-training phase, we have at our disposal entities and quantities encoders, learned by exploiting large amount of historical unlabeled data.

Sample selection. Once encoders have been pre-trained, the next step is to define the input samples for the downstream tasks. By input sample, we mean the individual subject that the task(s) will take as input. Unlike classic DL tasks whose input samples are often clear, e.g., a matrix of normalized pixel values for CV or a sequence of words embeddings for NLP, network-related tasks may have a variety of input samples. For example, if the goal is to classify IP addresses (or flows) as malicious or benign, then the input sample should be a vector representation of the IP address (or the flow). Alternatively, the input sample could be the first k packets of a flow, or a sequence of flows, etc. Once the input sample defined, its fixed-size vector representation is created by combining (i) the corresponding entities embeddings, and (ii) the corresponding quantities (or their encoded representations). For example, if the input samples is a sequence of k packets from five-tuple flows, its vector representation could be composed of the flow source and destination addresses embeddings, the flow source and destinations ports embeddings and the flow protocol embedding entities, combined with the sequence of k (encoded) packets size quantitie. For the sake of simplicity, in the remainder of this section, entities and quantities representations are combined by simple concatenation.

Training. When pre-training and sample selection are done, training a downstream task is rather straightforward. In an unsupervised use-case, one can simply cluster the unlabeled vector representations in order to group similar input samples together. Alternatively, in a supervised use-case, one can first associate labels to the input samples and then feed pairs of vector representations and labels to train a



(A) Users' view (B) ISP's view

FIGURE 5.4: Challenges for clickstream identification **(left)** Each web page downloads objects corresponding to the core domain content, or to support domains resources. **(right)** An ISP collects entangled traffic flow logs, from which it desires to identify the corresponding pages and domain type (core or support).

classifier or a regressor. Notice that, the more robust representations learned, the fewer labeled samples required for the downstream task (Tian et al., 2020; Brown et al., 2020). For example, in the clickstream toy case presented hereafter, we use a relatively small labeled dataset leveraging page visits of top 1000 Alexa ranking websites.

Inference. The last step is to use the trained model to perform inference for the downstream task. Classic ML challenges on how to keep the models up-to-date also apply here, but are out of the scope of this chapter.

5.2.4 Experimental results

This subsection illustrates the advantages of embedding network entities, as opposed to using only quantities, with two use-cases. In showcasing our approach, we only focus on categorical data embeddings. Otherwise stated, we use Word2Vec pre-training for entities and leave raw unprocessed quantities as they are. While a better representation of quantities might be learned (e.g., using auto-encoders), we leave it for future work and focus our experiments on characterizing the benefits of embedding network entities.

Clickstream identification In modern Web traffic, a single page corresponds to the download of tens of objects, retrieved from tens of different locations, respectively 70 and 50 in the top 1000 Alexa ranking websites. Since the advent of encryption, an Internet Service Provider (ISP) collecting web traffic flow logs cannot infer (i) which flow belongs to which page, nor (ii) which flow queries the domain of the main page i.e., *core domain* and which queries a necessary resource to render the page, i.e., *support domains*. These challenges, referred to as clickstream identification, are illustrated in Figure 5.4. Such knowledge could be useful for ISPs to estimate per-page

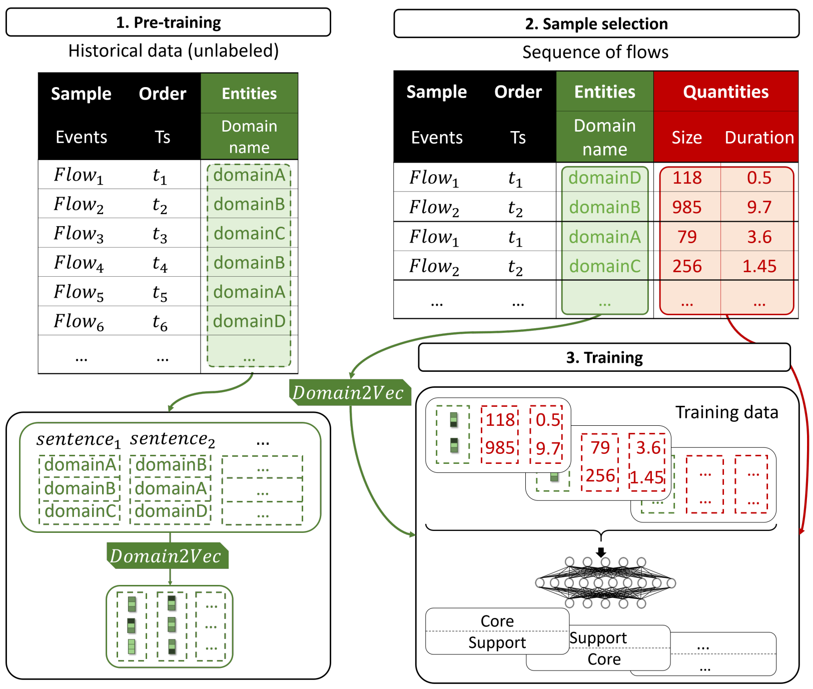
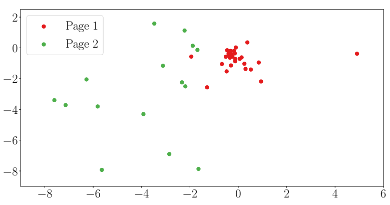


FIGURE 5.5: Bi-modal pipeline for clickstream identification.

Web quality of experience metrics from flow logs. Beyond the usefulness of the scenario itself, the two tasks above come with a number of methodological challenges that we believe are well suited to illustrate the proposed bi-modal representation learning scheme.

Let us consider a network vantage point that collects per-flow size & duration measurements (quantities) and domain names (entities). Following our guidelines, the first step is pre-training. In this case, we learn a *Domain2Vec* model from historical sequences of domain names in our dataset. Later, this *Domain2Vec* model will be used to embed each domain name in each input sample. As mentioned earlier, domain names can be embedded either with word or both word and character embeddings. With character embedding, words like `cdn`, `cdn21`, and `cdn22` will have similar embeddings even if they never co-occur in similar contexts. For our evaluation, we consider a Web traffic dataset of 20K domains retrieved by downloading 10 times each of the Web pages from the top 1,000 Alexa ranking, and recording flow logs. The pre-training dataset is then constructed by synthetically generating 100K multisessions of 3 to 10 (median of 6) simultaneous page visits. Using a context window of 200 flows, we create pairs of `<target, neighbor>` domain names from this long historical sequence of multisessions. *Domain2Vec* is then trained with a hidden size of 200 to generate domain names vector representations, as shown in the left part of Figure 5.5.

Given the two tasks (*i*) and (*ii*) described above, an input sample is a series of consecutive flows corresponding to the simultaneous query of an unknown number of



Approach	Precision [%]			Recall [%]		
	Q1	Q2	Q3	Q1	Q2	Q3
Naive	100	100	100	14	16	20
Quantities	66	75	100	50	60	75
Word emb.	75	100	100	33	50	66
Word+Char emb.	80	100	100	58	77	87

FIGURE 5.6: **(left)** PCA visualization of domain names embeddings for two pages. **(right)** Representation learning comparison on click-stream identification.

web pages. Each sample is thus a sequence of flows, each represented by its corresponding domain name embedding as well as its size and duration measurements, as depicted in the right part of Figure 5.5. We do not encode quantities in this experiment and instead keep raw flow measurements, that we concatenate with their domain name embeddings. The first ML task (i) aims to "disentangle" flows by associating them to their Web page. First, we qualitatively show how the entity-based representation helps solving this task. Figure 5.6 (left) plots a 2 component PCA representation of the domain names embeddings from flows that belong to two distinct web pages. Interestingly, without additional feature learning, the domain embeddings of different pages are already "disentangled", i.e., flows of different pages cluster in different regions, thus hinting that entity-based embeddings extract useful features. The subsequent task (ii) is to classify each flow in a sequence as either *core* or *support*. The supervised learning dataset consists of 60K similarly synthesized multisessions, i.e., 60K sequences of entangled flows corresponding to simultaneous page visits. Each page load corresponds to several flows, one of which is labeled as core because it queries the domain of the main page, and the remaining flows are labeled as support because they query external domain names resources. Hence, the task is to predict a sequence of binary labels for each multisession (i.e., each input sample). We select around 900 pages for training and validation sets, which corresponds to 50K multisessions. We test on 10K multisessions composed by the remaining *unseen* 100 pages which queried more than 1.2K unseen support domains. All in all, training/validation and test sessions queried respectively 15K and 2K domains. For this task, we consider as a baseline a naive predictor which systematically tags the first domain as core and the subsequent ones as support. Hence, this baseline correctly classifies the first flow, but misses the other core domains in case of a multisession. We compare this baseline against Gated Recurrent Unit (GRU, Cho et al., 2014) models fed with different flow representations: quantity-only features (flow size and byte progression), word embedding only, and word & character embeddings. In the following, we quantitatively show how the entity-based representation helps solving this task. Figure 5.6 (right) presents results in terms of precision and recall of the minority class, i.e., core domains. For each multisession, we compute precision and recall in predicting the core domains of the sequence. We show the

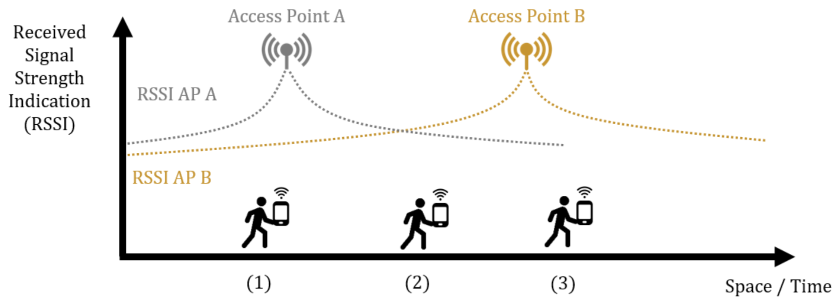


FIGURE 5.7: Illustrative example of Wi-Fi roaming. (1) the user terminal is associated to *AP A*. (2) the terminal receives weak signal from *AP A* and starts probing nearby APs. (3) the terminal has compared response signals and shifts to *AP B*.

distribution of these metrics across all multisessions, i.e, across all test samples, using quartiles. Despite the difficulty of the task, all models performed better than the naive baseline. Surprisingly, entity-based encoding outperformed the quantity-based one. Also, character embedding adds value compared to word embedding only. Note that, in this case, one-hot encoding the entities is not a viable solution because the test set contains unseen domain names.

Movement prediction in Wireless LANs A Wireless LAN deployment typically involves several Access Points (AP) providing network connectivity to mobile terminals, e.g., cellphones, laptops. In this context, an important problem is to predict whether a terminal is going to move away from its access point, reconnecting to another one. We depict this situation in Figure 5.7. Predicting terminal movement allows the network operator to proactively steer the terminal to roam before its signal actually degrades.

For this use case, let us consider network data composed of series of received signal strength indicators (RSSI) (quantities) and the set of APs traversed over time (entities). In other words, each terminal or user has a current associated AP and a signal strength towards it. For pre-training, we consider a dataset of real network data with approximately 2K mobile terminals and 240K movement events across 80 different APs located in a canteen WLAN. Following our bi-modal pipeline, the pre-training phase aims at learning to encode entities (and quantities) from historical sequences. This time, the historical RSSI and AP sequences are grouped by user. Sequences of per-user APs are then used to train an *AP2Vec* embedding model as depicted in the left part of Figure 5.8. Similar to the previous use case, we keep raw unprocessed quantities as they are. The *AP2Vec* pre-training is executed with context window size of 10 APs and generates an embedding vector of size 20. At the end of the pre-training phase, we have at our disposal a trained model for AP embedding.

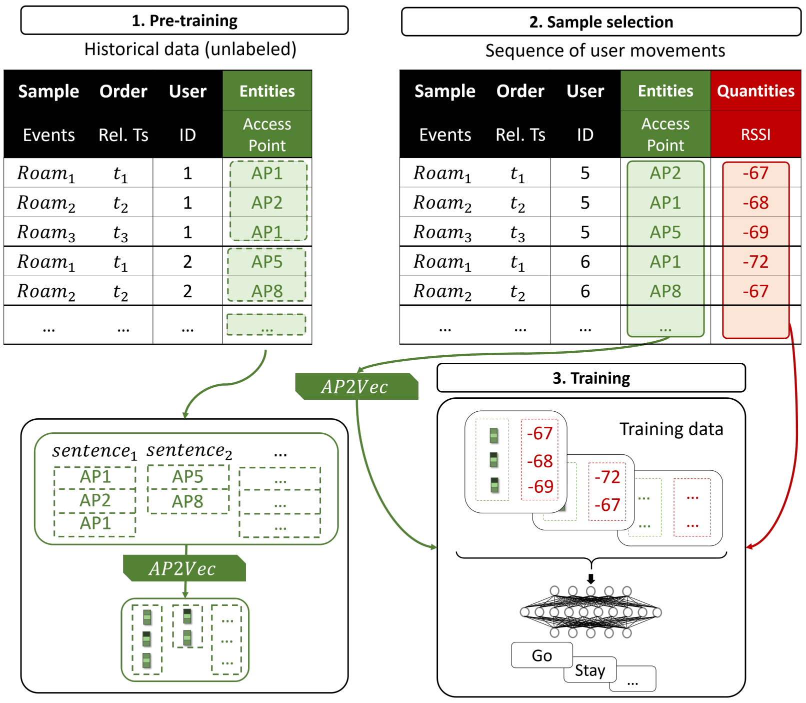


FIGURE 5.8: Bi-modal pipeline for terminal movement prediction.

For the terminal movement prediction downstream task, each input sample is a sequence of 10 consecutive RSSI experienced by a user during 10 seconds, concatenated with its last AP vector representation. Each sequence is labeled according to the action that the user will take in the next 20 seconds: either *stay* associated with its last AP or *move* to a different AP. We train a 1D CNN classifier on 3 days of these input samples and test the model on 2 other days of data. Figure 5.9 compares two input representation approaches to solve this task: RSSI+*AP2Vec* embedding (i.e., using quantities and entity) versus RSSI-only (i.e., using only quantities). The figure plots the precision and recall obtained over 10 runs of each approach. From the results it is clear that the bi-modal network data representation helps the ML task to reach a more accurate movement prediction. In particular, Figure 5.10 illustrates the disadvantage of the quantities-only approach (bottom) for a specific user by plotting

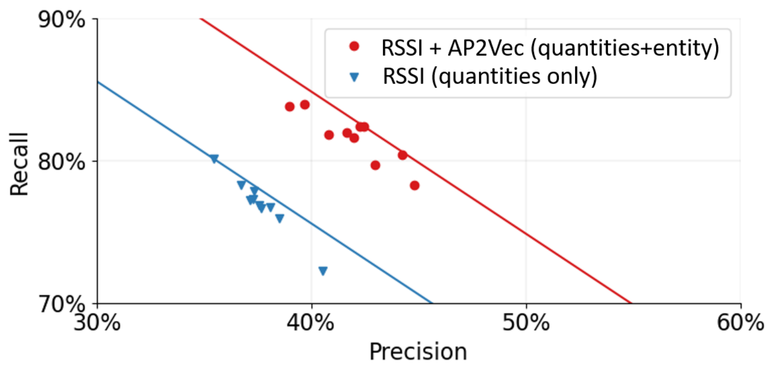


FIGURE 5.9: Approaches comparison on movement prediction.

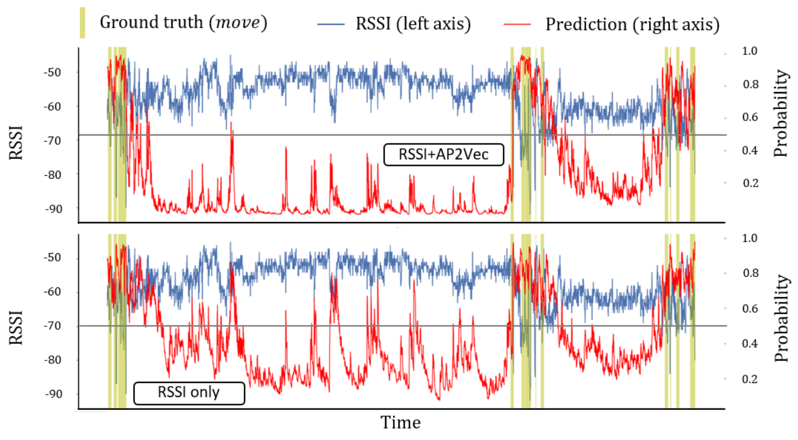


FIGURE 5.10: Comparison of model predictions for a specific user.

RSSI and corresponding model predictions over time. The quantities-only approach fails to identify RSSI degradations that do not lead to departure from the AP, hence producing more false positives than the model that uses quantities and entity representation. It is worth mentioning that for this use case, given the limited number and stability of entities (i.e., 80 fixed APs), one-hot encoding is also a viable solution as embedding technique. Although finding the optimal embedding technique is out of the scope of this work, we report that *AP2Vec* led to a slightly better model in our experiments.

5.2.5 Related work

Recent work started learning alternative representations for entities. One of the first efforts is IP2Vec (Ring et al., 2017) which embeds source and destination IP addresses and ports with the objective of identifying IP addresses with similar behaviors. However, the embedding training is limited only to the 5-tuples, and hence does not exploit the the historical sequences of entities. With the goal of identifying malicious behaviors, DANTE (Cohen et al., 2020) also leverages Word2Vec, but to embed ports that are sequentially tried by attackers. Similarly to DANTE, Darkvec (Gioacchini et al., 2021) uses word embeddings to project potential attackers, identified by IP, and grouped by service, identified by ports, into a latent space with the goal of clustering senders with similar behaviour. Another example (Gonzalez et al., 2021), close to the clickstream use case, leverages Word2Vec to build user profiles from browsing historical data.

An alternative to learning representations is feature engineering, but the latter often ends up in using a single modality. Searching for the best representation, Traffic Refinery (Bronzino et al., 2021) seeks the right balance between a feature-selection that is effective, i.e., accurate, and feasible, i.e., deployable at line rate. nPrintML (Holland et al., 2021) takes an orthogonal approach to network data representation with respect to the one proposed in this section, by representing packets headers in a one-hot encoding format that is then used to feed classical ML models. While replacing feature engineering from network data, such approach is extremely costly and fails to identify relevant patterns. Finally, with a few exceptions (Atmaja et al., 2022), we could not find examples in other domains beyond the classic (image, text) bi-modality. It could be a matter of time before this issue is similarly tackled in other areas.

5.2.6 Concluding remarks

As a first step towards multi-modality, we proposed a bi-modal network data representation of *entities* and *quantities*, in which historical sequences of entities are systematically transformed into vector representations using word embeddings. We showed the effectiveness of such representation through two toy examples as well as recent examples from the literature. As networks and systems in general are rife with sequential events, we believe the scope for potential applications of bi-modal data representation learning are broader than the illustrative toy cases. Other relevant use cases in networking include sequences of IP addresses, BGP advertisements, routing events, alarms, etc. With the widespread of data-driven decision taking (e.g. finance, manufacturing), applications beyond networks are likewise numerous. Yet, network data is more complex than co-occuring sequences of events and quantities illustrated in this section. Although useful as a conceptual framework, our bi-modal representation is only the starting point of the journey towards modeling more general multi-modality network data. For instance, as entities often exhibit complex relationships that can be represented by time-evolving graphs, Graph Neural Networks (GNNs) and graph embedding techniques seem to be another necessary piece in the quest towards multi-modality. Incorporating these pieces in the bigger puzzle of network data representation remains an interesting open research question for the networking community as a whole.

5.3 Task Groupings for Multi-Task Learning

In this section, we take a first step to tackle research question *(ii) How to design a unified pipeline that serves several networking measurements?* Network measurements fundamentally rely on transformation functions, that map input traffic data to an output metric or label. Traditionally, this mapping is implemented with specialized algorithms and data structures designed by network domain experts (e.g., sketches). Alternatively, Deep Learning algorithms propose instead to learn such input-output mapping by automatically extracting features from data. While they come with various challenges (e.g., implementation, robustness, generalization), neural networks' ability to extract rich representations from complex data make them appealing candidates for modeling traffic measurements. For example, Deep Learning-based representations have already been successfully adapted to model flow completion time, throughput or round-trip-time (Zhang et al., 2021a), as well as delay, jitter and losses (Rusek et al., 2020). Interestingly, some networking tasks and measurements might be related or dependent on one another. This relatedness constitutes an opportunity to exploit synergies by learning multiple tasks simultaneously to extract traffic representations common to several downstream tasks. For example, (Nascita et al., 2023) presents a CNN model that shares the same extracted features to solve three traffic classification tasks at once. (Wang et al., 2022b) uses the common traffic representations learned from a single GNN to provide predictions for flow completion time, path delay and throughput simultaneously. Alternatively, (Collet et al., 2023) learns to predict and aggregate intertwined forecasts to solve a global performance objective (e.g., QoE). Thus, we argue that Multi-Task Learning (MTL) is an appropriate paradigm for related networking tasks to benefit from a shared representation of their input.

While the promises of MTL are attractive, e.g., improving generalization (Caruana, 1997) or reducing training, inference as well as maintenance costs (Standley et al., 2020), characterizing the conditions of its success is still an open problem in Deep Learning. Some tasks may benefit from being learned together while others may be detrimental to one another. From a task perspective, grouping cooperative tasks while separating competing tasks is paramount to reap the benefits of MTL. While task relatedness plays a decisive role, recent work suggests that the training conditions themselves, e.g., model capacity, learning rate or training set size, have a significant impact on the outcomes of MTL (Standley et al., 2020; Fifty et al., 2021). Therefore, estimating task affinity for joint learning is a key endeavor. Yet, the literature is lacking of a benchmark to assess the effectiveness of tasks affinity estimation techniques and their relation with actual MTL performance. In this section, we take a first step in recovering this gap by *(i) defining a set of affinity estimation techniques* by both revisiting contributions from previous literature as well as presenting new ones and *(ii) benchmarking them* on a large public dataset. Unfortunately, there are not many network traffic datasets that are public and fit for MTL (Draper-Gil et al., 2016;

Barut et al., 2021; Sharafaldin et al., 2018; Ren et al., 2018). Furthermore, when considering task affinity assessment in particular, these datasets either contain too few tasks (i.e., up to three tasks) to properly evaluate numerous combinations of tasks groups, or contain tasks that are too closely related by design (e.g., hierarchical traffic classification, creating tasks by varying class granularity from traffic type to application name). Thus, instead of relying on simulated traffic data and hand-crafted tasks for this empirical campaign, we take a detour into the Computer Vision domain to benefit from Taskonomy (Zamir et al., 2018), an established dataset that offers a collection of 25 diverse tasks. Our findings reveals how, even in a small-scale scenario, task affinity estimation does not correlate well with actual MTL performance. Yet, some metrics can be more indicative than others.

In Section 5.3.2, we review the state of the art on MTL affinity characterization. In Section 5.3.3, we present the affinity metrics selected for benchmarking and detail our evaluation protocol. Then, we introduce our experimental setup and present our results in Section 5.3.4. Finally, we discuss the advantages and limitations of these metrics in Section 5.3.5.

5.3.1 Motivation

For more than two decades since its inception (Caruana, 1997), Multi-Task Learning has been extensively studied by the Deep Learning community. For practitioners interested in the best strategy to learn a collection of tasks, the promises of MTL are numerous and attractive. First, learning to solve several tasks simultaneously can be more cost-efficient from a model development and deployment perspective. Second, if the tasks learned together cooperate, MTL can even outperform its Single-Task Learning counterpart for the same computational cost (Standley et al., 2020). However, MTL potential advantages are tempered by the difficulty of estimating *task affinity*, i.e., identify tasks benefiting from joint learning, without testing all combinations of tasks. This calls for *task affinity metrics*, to quantify a priori and at a cheap computational cost the potential benefit of learning tasks together. The quest for the perfect affinity estimation technique is further exacerbated by MTL performance's strong dependency on the learning context, i.e., the data and models used for training. For instance, tasks cooperating in one learning context can result in competition when using slightly different data or models (Standley et al., 2020). Recent works (Fifty et al., 2021; Standley et al., 2020) have integrated this context-dependency when designing task grouping strategies. While these approaches avoid a complete search across all task combinations, they still require training and comparing some MTL models for the final network selection. Furthermore, those studies show that even in a small-scale scenario, MTL performance cannot be accurately predicted without actually performing MTL.

Despite providing assessment of task affinity, previous literature lacks of a broader comparison of the associated metrics. In this work, we take a first step in recovering this gap by presenting an empirical comparison of several task affinity estimation techniques. Some of these techniques are inspired by previous literature ranging from Transfer Learning to Multi-Task Learning: `taxonomical distance` (Zamir et al., 2018), `input attribution similarity` (Song et al., 2019), `representation similarity analysis` (Dwivedi and Roig, 2019), `gradient similarity` (Zhao et al., 2018) and `gradient transference` (Fifty et al., 2021). We benchmark an additional technique which is an original proposal: `label injection`. We evaluate all of them on the public Taskonomy dataset (Zamir et al., 2018) which is a well-known large benchmark spanning several Computer Vision tasks. Note that our objective is not to present a novel state-of-the-art MTL architecture but rather an objective benchmark of task affinity estimation techniques. More specifically we aim to understand if task affinity estimation can (i) be used as proxy for true MTL performance and (ii) suggest the best partner task to improve the performance of a target task. These metrics and their discussion aim at helping practitioners gauge the benefit of MTL for their own set of tasks.

5.3.2 Background and related work

In this section, we first review relevant work on MTL and task grouping, briefly present the Taskonomy dataset, and finally introduce task affinity characterization.

Multi-Task Learning. As introduced in Chapter 2, the promises of MTL are based on the assumption that cooperative tasks benefit from inductive transfer during joint learning. By being learned together, tasks are encouraged to share, at least partially, common representations, e.g., the extracted feature vector at the model's bottleneck, depending on the model architecture. The intuition is that some tasks might exhibit compatible goals and help one another during training through synergies, i.e., positive transfer. However, tasks interference can still degrade performance if their respective updates become unaligned or contradictory during simultaneous learning i.e., negative transfer through competition.

To mitigate these effects, two complementary lines of research both aim at reducing task interference and increasing task synergies. The first direction focuses on *model design*, hence crafting the model such that it is adapted to learn a certain set of tasks. In this case, the task set is fixed while the model is adapted to fit all the tasks under consideration. Through hard parameter sharing, task weights can be adapted during training in order to balance their impact on the combined loss (Leang et al., 2020; Pascal et al., 2021). Alternative approaches focus on tuning gradients to mitigate task interference during MTL training (Chen et al., 2018; Yu et al., 2020; Kendall et al., 2018). In soft parameter sharing instead, parameters are segregated by task and the model is guided, during MTL learning, to only share information when it is

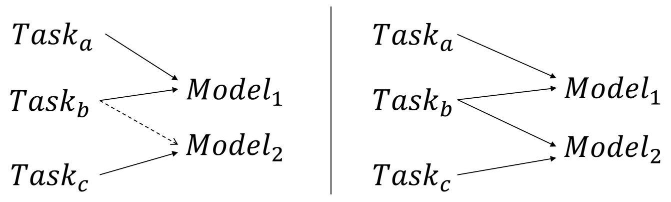


FIGURE 5.11: **(left)** A valid task grouping. Model$_1$ is assigned both Task$_a$ and Task$_b$ for inference. Model$_2$ is assigned Task$_c$ for inference and it uses Task$_b$ as a cooperative task only during training. **(right)** An invalid task grouping. Model$_1$ and Model$_2$ are both assigned to solve Task$_b$ at inference time.

beneficial (Sun et al., 2020; Misra et al., 2016). The second research direction is more recent and focuses on *task grouping strategies* by identifying cooperative tasks that can be grouped to be profitably learned together. In this case, the model design is fixed while the task set is adapted i.e., split into potentially overlapping subsets. Recent works from (Fifty et al., 2021) and (Standley et al., 2020) show promising results as they succeed in stimulating positive transfer by combining only tasks that are beneficial to one another. While these two research directions are complementary, our work is more in the scope of the latter as we benchmark affinity metrics that should indicate if tasks benefit one another when learned together.

Task grouping. Task grouping strategies aim at assigning tasks to models (that can be STL or MTL) in order to maximize the total performance of all the tasks under consideration, given a computational budget. More formally, let us consider the following:

- a set of n tasks $\mathcal{T} = \{t_1, t_2, ..., t_n\}$ that need to be solved;
- a total computational budget of β Multiply-Add operations;
- a set of $k \leq n$ models $\mathcal{M} = \{m_1, m_2, ..., m_k\}$, each one associated with its respective amount of Multiply-Adds operations $\mathcal{C} = \{c_1, c_2, ..., c_k\}$.

Task grouping aims at constructing $\mathcal{M}$ such that for all $t_i \in \mathcal{T}$ there exists exactly one model $m_j \in \mathcal{M}$ assigned to solve task t_i at inference time, while respecting the computational budget $\sum_{j=1}^{k} c_j \leq \beta$. Thus, any model from $\mathcal{M}$ can learn an arbitrary number of tasks as long as each task is assigned to one and only one model at inference time. To illustrate this point, we present valid and invalid task groupings in Figure 5.11. The final objective of task grouping is to maximize the aggregated test performance:

$$P = \sum_{t_i \in \mathcal{T}} P(t_i|\mathcal{M}), \tag{5.1}$$

where $P(t_i|\mathcal{M})$ denotes the performance[1] of task t_i using its assigned model from $\mathcal{M}$. It is worth mentioning that the task grouping problem differs from simple model selection as (i) the objective (aggregated performance) and constraints (total cost) concern all tasks and models simultaneously and (ii) any model can learn an arbitrary number of tasks.

The optimal task grouping is typically obtained by testing all task combinations within the computational budget. Therefore, to be as efficient as possible, grouping strategies rely on *task affinity estimates* that guide the search of a solution in the task groups space. This approach might only identify sub-optimal task groups but it has a much a lower cost than an exhaustive search. Sophisticated task grouping strategies are studied in (Fifty et al., 2021) in terms of performance and runtime. Such strategies include Higher-Order Approximation from (Standley et al., 2020), gradients cosine similarity maximization and task transference approximation from (Fifty et al., 2021). Our work complements this benchmark of grouping strategies as we are interested in assessing the strengths and weaknesses of the *underlying* affinity metrics. This includes an evaluation of the predictive quality of such metrics. Indeed, the perfect affinity estimation technique should not only identify the best partner tasks, but also be a proxy of the true MTL performance. Overall, we aim for a broader view of affinity metrics qualities w.r.t. what provided in the literature.

Taskonomy – the reference framework. From a Transfer Learning perspective, Taskonomy (Zamir et al., 2018) has been a successful attempt at clarifying transfer synergies between visual tasks. From an MTL perspective, (Standley et al., 2020) performs a broad empirical campaign on the same dataset to identify which visual tasks should be trained together with MTL. In particular, they evaluate if learning a target task with a partner task could outperform learning the target task alone. Thus, this framework quantifies task affinity as the performance gained on a task learned in MTL versus STL. First, the authors show that cooperation between tasks is not symmetrical, as one task may benefit from another but not necessarily the opposite. Second, by comparing MTL performance gains for the same pairs of tasks but learned in various settings i.e., different dataset size or different MTL model capacity, they unveil the impact of the training context itself on task cooperation. Based on this framework, (Fifty et al., 2021) monitors the evolution of task affinities during MTL training. Their experimentation on the CelebA dataset (Liu et al., 2018) suggests that task cooperation evolves throughout training. Furthermore they also show that hyper-parameters such as the learning rate or the batch size can also affect cooperation. Those works provide an in-depth view of relevant MTL training dynamics. Our work complements these findings with an in-breath view across several affinity estimation techniques that integrate, at varying degrees, data, model and hyper-parameters dependencies.

[1]Minimizing the model loss, or using a task-specific metric such as Intersection over Union for semantic segmentation.

Task affinity. As previously mentioned, (Standley et al., 2020) showed that task affinity is not symmetric, hence, in the remainder of this section, we refrain from using the term "metric" that carries mathematical properties. Instead, we group the methods aiming at quantifying task affinity for MTL under the term **affinity scores** for short, and break them down into three main categories depending on their requirements for computation.

Model-agnostic affinity scores are computed using solely the data at hand. This may be accomplished using nomenclatures or taxonomies to loosely relate tasks. For example, Object classification and Semantic segmentation are both considered to be semantic tasks while Depth estimation is a 3-D task (Zamir et al., 2018). This can also be more sophisticated and make use of information theory to quantify how dependent two tasks are, e.g., using labels entropy as in (Bingel and Søgaard, 2017).

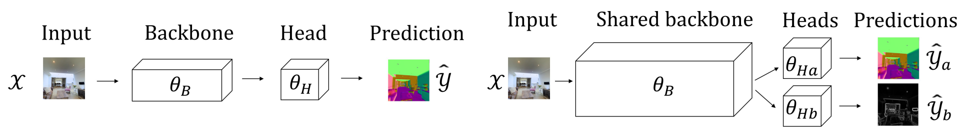
STL-based affinity scores make use of STL models and compare them to estimate affinity between tasks. Common approaches include comparing the STL models latent representations using e.g., the Representation Similarity Analysis (Dwivedi and Roig, 2019). Another option is to compare the STL models attribution maps assuming cooperative tasks use the same features (Caruana, 1997). Also, drawing from Meta-Learning, (Achille et al., 2019) estimates affinity as the distance between tasks in an embedded space that encodes task complexity.

MTL-based approaches estimate task affinity during the training of surrogate MTL model(s). Such computations need to be more efficient than testing all tasks combinations, otherwise it would defeat its very purpose of efficiently quantifying task affinity. (Fifty et al., 2021) proposes an affinity extraction method by simulating the effects that task-specific updates of the model parameters would have on other tasks. (Standley et al., 2020) extends pairwise MTL performance gain to higher-order task combinations i.e., groups of three or more tasks. Also, both (Fifty et al., 2021) and (Zhao et al., 2018) propose to compute the cosine similarity between task-specific gradient updates as a way to estimate task affinity during MTL training.

5.3.3 Task affinity scores benchmark methodology

Based on the assumption that grouping cooperative tasks together is a key success factor of MTL, we are interested in quantifying task affinity through several scores. In this section, we motivate the affinity scores selected and we detail the evaluation protocol implemented to benchmark them.

Affinity scores To simplify reasoning on task cooperation and competition, we restrict ourselves to *pairwise* task affinity estimation, i.e., affinity scores for 2-task MTL. We depict typical STL and pairwise-MTL architectures in Figures 5.12a and 5.12b. Considering two tasks $t_1 = a$ and $t_2 = b$ and a batch of examples $\mathcal{X}$, we denote:



(A) STL model architecture. (B) MTL model schematic architecture.

FIGURE 5.12: Schematic comparison of traditional STL and MTL architectures for two tasks $t_1 = a$ and $t_2 = b$. θ_B denotes the backbone weights, θ_H the single head weights, while θ_{Ha} and θ_{Hb} denote the separate heads weights.

- their respective loss functions $\mathcal{L}_a$ and $\mathcal{L}_b$
- their respective STL models STL_a and STL_b with losses
  - $\mathcal{L}_{STL_a} = \mathcal{L}_a(\mathcal{X}, STL_a)$
  - $\mathcal{L}_{STL_b} = \mathcal{L}_b(\mathcal{X}, STL_b)$
- their joint MTL model $MTL_{(a,b)}$ with loss
  - $\mathcal{L}_{MTL_{(a,b)}} = \mathcal{L}_a(\mathcal{X}, MTL_{(a,b)}) + \mathcal{L}_b(\mathcal{X}, MTL_{(a,b)})$

We consider six task affinity scores that we further describe in the remainder of this section. Their detailed computations are available in Appendix C.1. Some scores are symmetric, i.e., assessing how much two tasks a and b help each other regardless of direction; others instead are asymmetric, i.e., assessing how much a target task a benefits from being learned with a partner task b. For each metric we report its category (model-agnostic, STL-based or MTL-based) and contribution (borrowed from literature, revisited from literature or novel).

`Taxonomical distance (TD)` *Model-agnostic – borrowed*: A natural way of assessing affinity between tasks from a human perspective is to organize them through a hierarchical taxonomy. For example, classification datasets such as (Van Horn et al., 2018) or (Wah et al., 2011) provide hierarchical class granularity that can be used to group similar tasks together as in (Achille et al., 2019). In our case, we used the tasks similarity tree from (Zamir et al., 2018). This symmetric affinity score is computed as the distance between a and b in the tree.

`Input attribution similarity (IAS)` *STL-based – revisited*: (Caruana, 1997) defines related tasks as tasks that use the same features. Following this definition we assess how tasks relate to one another in terms of input attribution similarity using InputXGradient (Shrikumar et al., 2017) to compute attribution maps for STL_a and STL_b. The affinity score is then obtained via the cosine similarity of the attribution maps (Song et al., 2019). Therefore this score is symmetric.

`Representation similarity analysis (RSA)` *STL-based – revisited*: RSA, a well-known method in computational neuro-sciences (Dwivedi and Roig, 2019), relies on the assumption that, if tasks are similar, they learn similar representations, i.e., a given input should be projected in similar locations in the latent space. Referring to

Figure 5.12a, this score compares the latent representations structures between the respective backbones θ_B of STL_a and of STL_b. In a nutshell, RSA uses the Spearman correlation of Representation Dissimilarity Matrices. This is a symmetric score.

`Label injection (LI)` *STL-based – novel*: Another way to estimate task affinity is to measure the performance gained from adding the target label of another task to the input. For example, a task a targeting the classification of handwritten digits could be paired with a task b targeting the prediction of even and odd digits. Since the two tasks are (clearly) related, "injecting" the label of task b, i.e., providing it as complementary input when training task a, could lead to performance increase for task a. The performance of label injection can be considered as a proxy of task affinity. This affinity score is asymmetric. It is computed as the performance gain between the standard STL model for task a and the b-injected STL model for a denoted by $STL_{a \leftarrow b}$, i.e., using the test losses from the fully trained models:

$$\frac{\mathcal{L}_{STL_a} - \mathcal{L}_{STL_{a \leftarrow b}}}{\mathcal{L}_{STL_{a \leftarrow b}}}, \tag{5.2}$$

`Gradient similarity (GS)` *MTL-based – borrowed*: This task affinity score relies on the assumption that cooperative tasks yield similar i.e., non-contradictory, weights updates to the model backbone during MTL training. This score, which we borrow from (Fifty et al., 2021; Zhao et al., 2018), is symmetric. It is computed as the cosine similarity between gradients from each task loss with respect to the MTL model common backbone weights. Using the notation from Figure 5.12b, we the compute following cosine similarities at each epoch, and average them throughout training:

$$S_{cos}\left(\frac{\partial \mathcal{L}_a(\mathcal{X}, \theta_B, \theta_{Ha})}{\partial \theta_B}, \frac{\partial \mathcal{L}_b(\mathcal{X}, \theta_B, \theta_{Hb})}{\partial \theta_B}\right), \tag{5.3}$$

`Gradient transference (GT)` *MTL-based – borrowed*: During MTL training, by simulating task-specific updates to the common backbone, one can estimate how it would impact the other task's performance. This corresponds to the losses lookahead ratio defined in (Fifty et al., 2021). This asymmetric score is computed comparing the loss of task a after updating the common backbone according to b, and the loss of task a before this simulated update. Referring to the notation from Figure 5.12b, we denote the b-specific update of the common backbone by $\theta_{B|b}$. Thus, we compute the following ratios at each epoch and average them throughout training:

$$\frac{\mathcal{L}_a(\mathcal{X}, \theta_{B|b}, \theta_{Ha})}{\mathcal{L}_a(\mathcal{X}, \theta_B, \theta_{Ha})}, \tag{5.4}$$

Evaluation We evaluate these affinity scores against the true MTL performance. Moreover, we evaluate the scores across three levels by progressively relaxing the constraint of the analysis.

True performance: MTL gain. As in (Standley et al., 2020), we quantify MTL success as the relative gain between STL and MTL performance in terms of test loss. MTL gain for a target task a when using a partner task b is defined as:

$$\mathcal{G}(a|b) = \frac{\mathcal{L}_{STL_a} - \mathcal{L}_{MTL_{(\bar{a},b)}}}{\mathcal{L}_{MTL_{(\bar{a},b)}}}, \tag{5.5}$$

where $\mathcal{L}_{STL_a}$ is the test loss for task a in a STL configuration, and $\mathcal{L}_{MTL_{(\bar{a},b)}}$ is the test loss for task a in a MTL configuration using tasks a and b for joint learning. Note that the contribution of task b to the MTL loss is not considered when computing the gain, yet is considered during training. We perform an exhaustive search through all possible pairs of tasks to compute the "ground truth" affinities. These serve as baseline against which each affinity score is evaluated.

Level 1: predictive power. As previously stated, an ideal affinity score should be a proxy of the actual MTL gain: higher/lower score should imply large/small benefit from joint training. This is a stringent requirement, yet easy to quantify by mean of Pearson's correlation. Specifically, for each target task a and affinity scoring technique, we compute the correlation between the MTL gain across all partner tasks (the true performance) and the affinity score across the same partners (the proxy of the performance). It follows that affinity scoring techniques with correlation values close to -1 (perfect negative correlation) and $+1$ (perfect positive correlation) have strong predictive power; correlation values close to zero imply no predictive power.

Level 2: partners ranking. To relax the previous requirement, we define as acceptable an affinity score capable to successfully rank potential partner tasks by decreasing order of MTL gain. More formally, for a target task a, and a set of partner tasks $\mathcal{P}$, we want an affinity score δ such that:

$$\forall\, t_i \in \mathcal{P},\ \text{rank}(\,\delta(a, t_i)\,) = \text{rank}(\,\mathcal{G}(a|t_i)\,), \tag{5.6}$$

To evaluate the agreement between the ranking given by the affinity score and the actual ranking by MTL gain obtained by exhaustive search, we use Kendall's correlation coefficient (Kendall, 1948) that ranges from -1 (opposite rankings) to $+1$ (same rankings).

Level 3: best partner identification. In case only pairs of tasks are considered for MTL, one is essentially interested in finding the best partner. This means that we can further relax the previous constraint and for a target task a, we want an affinity score δ such that:

$$\arg\max_{t_i \in \mathcal{P}} \delta(a, t_i) = \arg\max_{t_i \in \mathcal{P}} \mathcal{G}(a|t_i), \tag{5.7}$$

To evaluate this, we report the MTL gain obtained when choosing the top partner according to the affinity score and compare it with the maximum MTL gain obtained when choosing the actual best partner.

5.3.4 Experimental results

Dataset. In this work, we select a portion of the Taskonomy medium-size split. This constitutes a representative dataset of Computer Vision tasks, composed of labeled indoor scenes from 73 buildings whose list is available in Appendix C.2. The whole dataset amounts to 726,149 input images which represent approximately 1.2 TB including the various labels. We select the same five tasks as (Standley et al., 2020; Fifty et al., 2021) to conduct our experiments, namely:

- Semantic segmentation (*SemSeg*)
- 2D SURF keypoints identification (*Keypts*)
- Edges texture detection (*Edges*)
- Depth Z-Buffer estimation (*Depth*)
- Surface normals estimation (*Normal*)

A detailed description of the tasks can be found in the supplementary material from (Zamir et al., 2018).

Models definition. We build on the work of (Standley et al., 2020) to train five STL models for the five aforementioned tasks and ten pairwise MTL models. Models are variants of the Xception architecture (Chollet, 2017), composed of a backbone that learns a latent representation of the input and a head. In the case of the STL models, the backbone output is forwarded to a single head that produces the final prediction, cf. Figure 5.12a. In the case of the pairwise MTL models, the shared backbone output is forwarded to two disjoint heads, one for each task under consideration by the MTL model, cf. Figure 5.12b. In this work, as well as in (Standley et al., 2020; Fifty et al., 2021), we only consider hard parameter sharing for the MTL backbone. While this approach simplifies reasoning about shared representations and weights updates, it does not incorporate task interference mitigation strategies.

In terms of model capacity, we replicate the Xception17 models design from prior work in (Standley et al., 2020), allowing each STL model only half of the capacity i.e., number of Multiply-Add operations, of a pairwise MTL model. This constraint is implemented by reducing the number of channels in the CNN blocks composing the backbone. Therefore, STL and MTL models use the same architecture but with varying capacity. Each model is trained for 50 epochs with a decreasing learning rate, selecting the best-performing epoch on the validation set as final model. Finally, hyper-parameters are set to default values from (Standley et al., 2020) with no further tuning.

In the remainder of this section, we report our evaluation based on the methodology described previously. The detailed values of each affinity score are instead reported in Appendix C.3.

Trained with	MTL gain on					Avg.
	SemSeg	Keypts	Edges	Depth	Normal	
SemSeg	-	-11.81	-10.22	-0.55	+0.95	-5.41
Keypts	-6.70	-	-8.67	-9.87	-13.88	-9.78
Edges	-22.01	+1.26	-	-8.24	+2.18	-6.70
Depth	+18.02	-3.81	+16.69	-	-6.37	+6.13
Normal	+50.24	+29.56	+78.05	-0.45	-	+39.35

TABLE 5.1: *True performance: MTL gain.* Ground-truth MTL gain for each target task (column) and each partner task (row). E.g., the task Edges performs 78.05% better than learned alone in STL when trained with Normal as partner.

MTL gain. In Table 5.1, we report the ground truth MTL gain for each pair of tasks. We reiterate that these results serve as reference for evaluating the affinity scores. Furthermore, recall that MTL gains are tightly related to the specific training conditions of our experiment i.e., the data, models and hyper-parameters used, and they may vary if computed in another setting. From this table, we note that some tasks are more helpful than others. For example, *Normal* is a helpful partner task, but fails to be significantly assisted by any other task. Overall, we find MTL gains to be highly asymmetric. Nonetheless, almost all tasks would benefit from being learned with their best partner. This is in line with the findings of (Standley et al., 2020).

Predictive power. Table 5.2 shows the Pearson correlation between the MTL gain and each individual affinity score. Each row considers a separate target task, while the last row labeled as *all-at-once* reports the correlation computed using all pairs of all target tasks together. Starting from such an aggregate scenario, we can see that no scoring technique strongly correlates with the MTL gains. Only `Label injection (LI)` moderately correlates with MTL gain across all tasks pairs (Pearson corr. $=$ 0.47). This invalidates the predictability property desired for an ideal affinity score. Interestingly, when considering a single target task at a time, some affinity scores successfully predict MTL performance. For example, *Depth*'s MTL gains can be predicted using `Input attribution similarity (IAS)`, with a corr. $=$ 0.98. Yet, no scoring provides a stable correlation across *all* tasks pairs.

Partners ranking. In Table 5.3, we evaluate each affinity scoring technique in terms of its ability to correctly rank potential partner tasks according to the MTL gains they provide. For a given target task, we compare the rank obtained from the affinity score with the rank obtained from the MTL gains by mean of the Kendall rank correlation. As in the predictive power evaluation table, each row reports on the correlation for each target task separately while in this case the last line summarizes the overall performance using the average of the rank correlations across target tasks. Starting from the aggregate view, we observe that no score-based ranking correlates strongly with true ranking. Only `Label injection (LI)` and `Gradients`

	Model agnostic	**STL-based**			**MTL-based**	
Task	`TD`	`IAS`	`RSA`	`LI`	`GS`	`GT`
SemSeg	0.4	0.99	0.81	0.99	0.79	0.76
Keypts	-0.03	-0.06	-0.37	0.95	0.22	-0.08
Edges	-0.34	-0.44	-0.68	0.90	-0.37	-0.66
Depth	0.90	0.98	0.96	0.64	0.69	0.97
Normal	0.60	0.38	0.20	-0.11	-0.19	0.40
All-at-once	0.08	0.08	-0.15	**0.47**	-0.08	-0.02

TABLE 5.2: *Level 1: predictive power.* Affinity scores correlation with MTL gain. E.g., using `Label injection (LI)` to estimate affinities for the target task SemSeg, its output strongly correlates with the actual MTL gains (Pearson corr. = 0.99).

`similarity (GS)` show a moderate and positive correlation (average Kendall corr. = 0.47 and 0.4 resp.). Differently from before, when considering specific targets tasks, the correlation does not necessarily improve. For instance, *Keypts* and *Normal* STL-based scores completely fail, yet MTL-based scores are not necessarily better. Still considering *Keypts* target task, notice how `Label injection (LI)` showed significantly higher Pearson correlation, while the Kendall correlation shows that half of the partner tasks are wrongly ranked according to the affinity score.

Best partner identification. Table 5.4 shows the top-1 partner according to each affinity scoring technique. This is to be compared with the maximum MTL gain that can be achieved using the actual best partner. `Label Injection (LI)` correctly identifies the best partner for four out of five tasks. However, not a single affinity score is capable of correctly identifying *Normal*'s best partner for MTL. Furthermore, *Keypts* and *Edges* seem to be particularly difficult tasks for best partner identification. All scores but `Label injection (LI)` recommend choosing either one as best partner for the other, while the actual best choice is *Normal* for both of them.

	Model agnostic	**STL-based**			**MTL-based**	
Task	`TD`	`IAS`	`RSA`	`LI`	`GS`	`GT`
SemSeg	0.0	1.0	0.33	1.0	0.67	0.67
Keypts	0.0	0.0	0.0	0.0	0.67	0.33
Edges	-0.33	-0.33	-0.33	0.67	0.0	-0.33
Depth	1.0	0.67	1.0	0.67	1.0	0.67
Normal	0.33	0.0	0.0	0.0	-0.33	0.0
Average	0.2	0.27	0.2	**0.47**	0.4	0.27

TABLE 5.3: *Level 2: partners ranking.* Comparison of partner tasks ranking by affinity score versus by MTL gain. E.g., `Label injection (LI)` perfectly ranks partners for the target task SemSeg (Kendall corr.= 1).

Task	Expected partner	Model-agnostic	STL-based			MTL-based	
		`TD`	`IAS`	`RSA`	`LI`	`GS`	`GT`
SemSeg	Normal	Normal (0)	Normal (0)	Depth (-32.2)	Normal (0)	Depth (-32.2)	Depth (-32.2)
Keypts	Normal	Edges (-28.3)	Edges (-28.3)	Edges (-28.3)	Normal (0)	Edges (-28.3)	Edges (-28.3)
Edges	Normal	Keypts (-86.7)	Keypts (-86.7)	Keypts (-86.7)	Normal (0)	Keypts (-86.7)	Keypts (-86.7)
Depth	Normal	Normal (0)	SemSeg (-0.1)	Normal (0)	Normal (0)	Normal (0)	SemSeg (-0.1)
Normal	Edges	SemSeg/Depth (-4.9)	SemSeg (-1.2)	Depth (-8.6)	Depth (-8.6)	Depth (-8.6)	Depth (-8.6)

TABLE 5.4: *Level 3: best partner identification*. Comparison of best partner selection by affinity score. In parenthesis, we report the difference of MTL gain between the actual best and the selected partner. E.g., for the target task Keypts the actual best partner is Normal and all scores but `Label injection (LI)` select Edges leading to 1.26 - 29.56 = -28.3 points decrease in performance gain.

5.3.5 Discussion

A perfect affinity score should be both predictive of the actual MTL gain and cheap to compute. As prior work hints that the training conditions themselves impact MTL gain, it seems particularly tough to reconcile these properties as we also verify throughout our experimental campaign. In this section, we benchmark various affinity scoring techniques that incorporate data, model and hyper-parameters dependencies at varying degrees: from model-agnostic scores that do not take these into account, through STL-based scores that try to include them, to MTL-based scores that are supposed to be the closest to the actual MTL learning conditions. Unfortunately, none of the selected scores, not even the MTL-based ones that are close to MTL training, can accurately predict MTL gain across all pairs of tasks. However, `Label injection (LI)`, the original affinity score we introduce, appears useful for predicting the gains corresponding to potential partners given a target task. We also observe that, surprisingly, MTL-based scores are not necessarily better than STL-ones, i.e., not even quantifying affinity during the actual MTL training seems sufficient to link affinity to performance.

From a cost perspective, except for `Taxonomical distance (TD)`, all the scoring techniques we benchmark require some model training. We quantify the computational cost of an affinity score by the total amount of training it requires to estimate affinities across all pairs of tasks. Let us consider n tasks and a standard half-capacity STL model with its respective number of Multiply-Add operations denoted by c_s. Using this notation, we report the computational cost associated with each affinity score in Table 5.5. While `Input attribution similarity (IAS)` and `Representation similarity analysis (RSA)` only require one STL model per task, `LI` requires to train an additional STL-injected model for each ordered pair. We note that, in some scenario, the STL models may be readily available, such that the costs associated with `IAS` and `RSA` can be amortized. Regarding MTL-based scores, both `Gradient similarity (GS)` and `Gradient transference (GT)` require to train a full-capacity MTL model for each unordered pair of tasks. Note that we neglect the cost of the simulated task-specific update during `GT` training. Finally, while `TD`

	Model agnostic	**STL-based**			**MTL-based**	
Cost	TD	IAS	RSA	LI	GS	GT
# of Multiply-Adds	0	$n \cdot c_s$	$n \cdot c_s$	$n \cdot c_s + 2\binom{n}{2} \cdot c_s$	$\binom{n}{2} \cdot 2c_s$	$\binom{n}{2} \cdot 2c_s$

TABLE 5.5: *Affinity scores costs.* Comparison of training costs considering all pairs across n tasks, where c_s denotes the amount of Multiply-Add operations for a standard half-capacity STL model.

may appear cost-efficient, it has been established using a Transfer Learning-based taxonomy that itself requires STL models training, cf. (Zamir et al., 2018).

From a practical perspective, LI can correctly identify the best partner for most tasks, except for *Normal*, for which none of the other scores succeeded. As in (Standley et al., 2020), we find that *Normal* is different from the other tasks: it benefits others but it is better learned alone. We conjecture that the high complexity of this task makes it a good partner for sharing knowledge during joint learning, but prevents it from being helped by easier tasks. To further corroborate this hypothesis, task complexity need to be incorporated in the affinity scoring design. We believe that Task2Vec from (Achille et al., 2019) is a first step towards this direction as it establishes a distance metric between tasks incorporating task difficulty from a Transfer Learning perspective. Unfortunately, Task2Vec cannot be directly used in our context as it has only been defined for homogeneous tasks, i.e., from the same domain. Indeed, in (Achille et al., 2019), the tasks are defined using coarse or fine-grained classification variations from the same hierarchy. We leave the exploration of this research direction as future work.

While this empirical campaign provides a better understanding of the challenges to take up when designing task affinity scores, it is not conclusive given the high variability coming from data, models, and tasks used for MTL. In other words, while some results are encouraging, more research is required to make those mechanisms actionable for an actual model design and operation. In this direction, we identify some limitations that we intend to tackle in future work. First, this analysis is limited to five Computer Vision tasks. Some model-agnostic affinity scores such as TD might not be trivially adapted to other task domains. Second, the affinity scores we defined can only estimate pairwise task affinity. While this is a reasonable starting point, various effects may be at play when learning more than two tasks simultaneously. (Zhang et al., 2021c) propose a new perspective on a sample-wise basis to quantify task transfer and interference separately. However, their metrics are defined for classification tasks only and their adaptation to heterogeneous tasks is an open question. Third, the MTL architecture we selected features hard parameter sharing and a static combined loss with equal weights. Although this design choice is consistent with prior work and facilitates reasoning on tasks cooperation, it does not take advantage of the recent advances in task interference mitigation techniques for MTL training (Chen et al., 2018; Yu et al., 2020; Kendall et al., 2018). Indeed, tasks may be affine but still interfere during joint learning if no mechanism is implemented to attenuate it, which is why MTL architecture design and task grouping strategies are complementary lines of research.

5.3.6 Concluding remarks

Based on the assumption that identifying cooperative tasks to be learned together is a key success factor of MTL, we borrowed, adapted and designed various task affinity scores for this purpose. We benchmarked these scores for pairs of tasks on a public Computer Vision dataset to discuss their strengths and weaknesses. Although no score is perfectly predictive of MTL gain, some of them still hold value for practitioners, by being able to identify the best partner for a given target task. This empirical campaign offers a better understanding of the conditions that allow MTL to be superior to STL and sheds light on the challenges to be met when predicting it. We leave the study of porting these findings to the networking domain as future work, once a large and public multi-task networking dataset is established.

Related publications

Raphael Azorin, Massimo Gallo, Alessandro Finamore, Maurizio Filippone, Pietro Michiardi, and Dario Rossi (2021). "Towards a Generic Deep Learning Pipeline for Traffic Measurements". In: *Proceedings of the CoNEXT Student Workshop*. CoNEXT-SW '21. Virtual Event, Germany: Association for Computing Machinery, 5–6. ISBN: 9781450391337. DOI: 10.1145/3488658.3493785

Zied Ben Houidi, Raphael Azorin, Massimo Gallo, Alessandro Finamore, and Dario Rossi (2022). "Towards a Systematic Multi-Modal Representation Learning for Network Data". In: *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. HotNets '22. Austin, Texas: Association for Computing Machinery, 181–187. ISBN: 9781450398992. DOI: 10.1145/3563766.3564108

Raphael Azorin, Massimo Gallo, Alessandro Finamore, Dario Rossi, and Pietro Michiardi (2023). "It's a Match! A Benchmark of Task Affinity Scores for Joint Learning". In: *Association for the Advancement of Artificial Intelligence (AAAI). 2nd International Workshop on Practical Deep Learning in the Wild*. Washington D.C., USA. URL: https://arxiv.org/pdf/2301.02873.pdf

Chapter 6

Conclusion

In this thesis, we discussed the challenges of appropriately representing traffic for various network measurements and their applications. We proposed several solutions to address these challenges, in particular by leveraging Machine Learning (ML) algorithms. In this chapter, to conclude this thesis, we first recapitulate the main contributions. Then we discuss the impact and limitations of the research conducted. Finally, we detail future research challenges and opportunities in traffic representation learning for network measurements.

6.1 Summary of Contributions

As discussed throughout this thesis, traffic measurements are a prerequisite for many network operations and management tasks. Measurements provide operators with a view of their network activity, which is essential to evaluate performance, establish diagnosis, and address issues. In particular, per-flow monitoring deals with the characterization of individual data streams flowing through the network. Through the selection, collection, and analysis of specific metrics, measurements describe (i.e., *represent*) network activity at different levels of granularity. This process of extracting a traffic representation is both critical and challenging. It requires carefully relating the cost and feasibility of obtaining specific traffic characteristics, with their expressiveness to fuel various downstream networking tasks. In this thesis, we proposed several solutions to improve this process, segregated by their degree of Machine Learning integration.

6.1.1 Traditional network measurements

In Chapter 3, we introduced **sparse sketches representations for per-flow monitoring**. Sketches are dimensioned for extreme scenarios, while only a minority of flows require high-accuracy and memory-intensive sketches. Hence, a majority of per-flow sketches end up under-utilized, wasting precious memory. Based on this observation, our solution SPADA consists in storing only non-zero sketch counters, which requires to address various data plane implementation challenges, i.e., storing <key, value> pairs for the relevant counters, taking into account hardware limitations and without a priori knowledge of flow sparsity. Our solution comes with two implementation flavors, corresponding to two key-value data structures for counters update (quotiented Cuckoo Hash Tables and perfect Invertible Bloom Lookup Tables), with distinct advantages and drawbacks in terms of flexibility and overhead. Compared to traditional approaches that reserve a full sketch of fixed size as soon as the first packet of a new flow arrives, our solution instead dynamically assigns pre-allocated per-flow counters when needed. Thus, it requires maintaining fewer counters to monitor active flows. We evaluated the overhead of SPADA due to the additional counter indexes on three monitoring use cases using real traffic data. We recorded from $2\times$ to $11\times$ memory footprint reduction with respect to the state-of-the-art while maintaining the same accuracy.

Open questions. The memory savings achievable by SPADA depend on the actual sparsity of the sketches. When configuring our framework, the system's Sketch Data is sized according to a worst-case average sparsity estimation, computed from historical traffic records. Under-estimating sparsity leads to reduced memory savings but over-estimating it risks overflowing the data structure. Although we demonstrated the advantage of our solution even taking into account conservative sparsity factors (twice the observed sparsity on real traces), sketches' under-utilization is dependent on traffic patterns, which are dynamic. Hence, an interesting question is how to extend this framework to adapt to sudden changes in sparsity. Also, for sketch algorithms that require non-additive counters update rules (e.g., HyperLogLog for cardinality estimation), our solution works only in its quotiented Cuckoo Hash Table flavor, which triggers some packet re-circulations. Even if worst-case evaluations (i.e., with a load factor ≈90%) on real traces assess that SPADA's re-circulation rate remains acceptable (below 5%), further research efforts could improve this aspect of the proposed system.

6.1.2 ML-assisted network measurements

In Chapter 4, we presented a **machine learning solution to provide early and affordable flow size representation**. In particular, we integrated a custom Random Forest classifier in the data plane to provide timely and coarse-grained prediction of flow size (i.e., the flow is going to be an elephant or a mouse). While imprecise, such early knowledge of a key flow characteristic proved valuable to improve various networking use cases. The proposed system, DUMBO, has been tuned to meet stringent memory-performance trade-offs. Additionally, DUMBO is equipped with an active learning mechanism to adapt its lightweight classifier to traffic pattern changes. We demonstrated the superior performance of our system over state-of-the-art non-learned approaches for flow scheduling, per-flow packet inter-arrival time distribution, and flow size estimation, using real traffic traces.

Open questions. While useful in several use cases, this approach based on ML hints might be extended to enhance more networking tasks. In DUMBO, a single binary hint on the flow's expected length (i.e., elephants/mice classification) is leveraged across three networking tasks. This information could be useful to additional use cases, e.g., congestion control. However, distinct tasks might exhibit different sensitivity to mispredictions, which makes it challenging to tune model performance (i.e., precision and recall in our case). For example, scheduling queues could tolerate lower hint precision than what is acceptable for flow size exact counters before being saturated by mispredicted mice. Hence, an interesting research question pertains to model performance tuning to best satisfy diverse use cases. Alternatively, other kind of hints could bring benefits to additional tasks, e.g., recent work proposed to predict packet drops to improve buffer management (Addanki et al., 2024). Finally, the implementation of more sophisticated and higher-performing ML models in the data plane is a promising research direction. For example, providing finer-grained classification, flow size regression (Hsu et al., 2019) or even inferring multiple flow metrics from the same model seems to be achievable with higher-capacity ML architectures (e.g., neural networks). In this regard, weights quantization initiatives for hardware accelerators (e.g., Umuroglu et al., 2017 for Binarized Neural Networks) are attractive but require further evaluation to ensure their implementation is profitable in a data plane monitoring system.

6.1.3 ML-based network measurements

In Chapter 5, we proposed **traffic representation learning methodologies for network measurements**. We first motivated the use of Deep Learning algorithms to extract rich representations from complex traffic data in order to solve multiple downstream networking tasks. Then, we introduced a systematic bi-modal representation learning methodology, tapping into network categorical data by leveraging word embeddings. In particular, we proposed to exploit the information contained in the non-arbitrary ordering of network entities that occur in sequences. We demonstrated the added benefit of incorporating such knowledge on two toy networking use cases: clickstream identification and WLAN terminal movement prediction. Finally, we advocate for a Multi-Task Learning (MTL) approach to serve several network measurements simultaneously. As a first step towards this objective, we addressed the task grouping problem that deals with the identification of cooperative and competing tasks. Specifically, we benchmarked six task affinity metrics on an established MTL dataset. While this empirical campaign shed more light on the conditions for MTL to outperform single-task learning from a task perspective, no metric was perfectly predictive of task cooperation.

Open questions. In the first part of our work on ML-based measurements, our approach considers network entities and quantities separately during pre-training. We used Word2Vec pre-training to embed network entities' co-occurrence patterns in a vector space, and then concatenate these representations with quantities to serve as input to fine-tune a model for a specific downstream task. While successful in two use cases, this embed-then-concatenate approach may be sub-optimal to capture relationships between entities and quantities, as these are only learned in the final training phase. More recent architectures, e.g., Transformers (Vaswani et al., 2017), are now ubiquitous in various domains (e.g., speech, images, text), featuring superior attention-based mechanisms to produce rich embeddings. Interestingly, Transformers have been applied beyond language categorical data, e.g., learning embeddings from images numerical pixel data (Dosovitskiy et al., 2020). Hence, future traffic embedding techniques may consider self-attention to relate network entities *and* quantities to one another, within the same pre-training procedure rather than in isolation. Regarding the simultaneous learning of multiple tasks in the second section of this chapter, our work only scratched the surface of MTL for network measurements. Indeed, we only addressed this research direction from a task perspective, and in an orthogonal domain (computer vision). While our benchmark was not conclusive in this scenario, we conjecture that task difficulty plays an important role in characterizing affinity. Further work may integrate this dimension into affinity estimation techniques, e.g., using a framework such as Task2Vec (Achille et al., 2019). Finally, we leave the adaptation of our approach to the networking domain as future work, to bridge traffic sequence modeling with Multi-Task Learning.

6.2 Perspectives

One of the main research hypotheses of this thesis is that *Machine Learning can provide traffic representations that enhance network measurement tasks*. We broke down this hypothesis into ML-assisted and ML-based measurements and showed that it can outperform traditional non-learned measurements in some conditions. In the following, we reflect on the challenges faced and the lessons learned along this journey. Finally, we suggest avenues to explore in order to foster future research in the field.

First, we underscore that the transition from a theoretically sound concept to the actual deployment of an ML-enabled system within constrained devices is a complex and iterative process. While based on previous theoretical works (Mitzenmacher, 2021; Hsu et al., 2019; Du et al., 2021), we faced numerous engineering challenges and trade-off decisions to propose an affordable yet effective ML networked system. This journey required a deep understanding of both the underlying protocols and systems (e.g., which tasks can exploit which hints), and of the practical constraints involved (e.g., strike a balance between model performance and overhead). The main challenge we faced was to simplify the tasks and models previously considered while retaining sufficient hint value to benefit a real measurement system. In particular, it involved careful model design to develop a predictor that fits the stringent memory and computation constraints of network devices. We downgraded a difficult flow size regression task into a simpler binary classification and developed a custom Random Forest with low space and computation complexity for data plane implementation. Additionally, the dynamic nature of traffic patterns, too often overlooked in prior works on learned-sketches, prompted us to design a model update mechanism based on active learning (Settles, 2011). Also, the complex interplay between system components required us to characterize various memory-accuracy trade-offs, in order to guide their sizing and satisfy diverse use cases. To foster research in this direction and stimulate capitalization on our work, we made our model, system and evaluation code publicly available at (*DUMBO Simulator* 2024). We note that ML-assisted networked systems are gaining more traction, e.g., with the recent Credence (Addanki et al., 2024) or QCLIMB (Li et al., 2024) system proposals. As various types of affordable and valuable traffic hints emerge (e.g., predictions on flow class, flow size, packet drops), an interesting future direction concerns the prediction of diverse flow characteristics to serve numerous use cases simultaneously. In our work, we concentrated our efforts on fully exploiting the value of a single prediction (e.g., flow size). Future work may instead study the implementation of multiple or multi-task models in constrained network devices.

Second, we highlight the inherent difficulty of learning from network data, which is still a relatively new modality to the broader field of representation learning. Instead of developing hand-crafted models to predict selected traffic characteristics (e.g., flow size), we envisioned the automatic extraction of relevant traffic features thanks to Deep Learning algorithms (i.e., representation learning). Inspired by successful methodologies in other domains (Mikolov et al., 2013b; Fifty et al., 2021), we proposed to adapt and tailor recent advances in representation learning to automatically extract knowledge from network data. In this journey, we acknowledged the complexity of learning representations from network data. Compared to the more mature Natural Language Processing and Computer Vision communities, networking data representation learning is still in its infancy. For instance, no clear consensus has emerged yet regarding the correct encoding of input data to facilitate knowledge extraction. Indeed, some approaches favor ingesting packets or flow features, some consider multivariate time series (e.g., packets inter-arrival times and direction) while others rely on payload features (Yang et al., 2021). Instead, (Holland et al., 2021) advocates for one-hot encoding packet headers. In addition, downstream tasks, which should guide architecture and input encoding choices, are also harder to define. From this wide array of input options and end-tasks, we recognize the unfeasibility of a silver bullet representation. Navigating this uncharted space from a representation learning point of view may seem daunting at first. Given the abundance of network data in practice, it is unfortunate that large, public, and recent-enough network datasets are still so few in number from a practical research standpoint. Additionally, we note that a critical specificity of network data is that it is dynamic. This property limits the robustness of the representations that may be learned from historical (i.e., offline) traffic data. To make an analogy with language, while a single word may have various meanings depending on its context, it is uncommon for these meanings to change over time or across data sets. On the contrary, the meaning carried by a network flow's five-tuple is not stable across time or space. Finally, network activity is captured into various modalities (e.g., traffic, textual logs, graph topologies, structured data). Hence, developing methodologies to summarize disparate data sources and efficiently represent the network state is a key endeavor for the community (Feamster and Rexford, 2017; Behringer et al., 2021). In this regard, neural compression approaches constitute an interesting research direction to encode and compress network data.

Appendix A

Appendix for Chapter 3

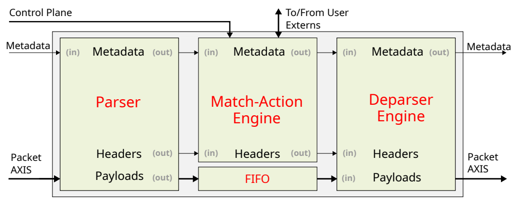
In this appendix, we provide details on the implementation of a SPADA pipeline in P4 and benchmark it on a Xilinx FPGA-based SmartNIC target. We deferred this implementation work here because it has not been conducted as part of the PhD program and was the responsibility of external collaborators from Sapienza University, Rome.

A.1 System Implementation

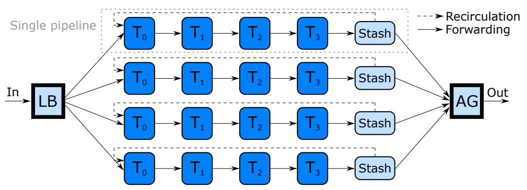
We implement SPADA using the Xilinx Vitis Networking P4 (AMD, 2023). The framework translates P4 code into Intellectual Property blocks for AMD-Xilinx FPGAs. Such blocks can then be integrated into a wrapper, as the AMD Xilinx OpenNIC Shell (*AMD OpenNIC Project* 2023) or the NetFPGA-PLUS (NetFPGA publisher, 2023), which provide basic networking functionalities, and deployed in an FPGA-based SmartNIC. Our implementation exploits the NetFPGA-PLUS reference NIC using the Xilinx Alveo U280 (2×100Gbps QSFP, 8GB DDR, 41MB SRAM, 1M Lookup Tables, and 2M Flip-Flops) as target board. All the processing pipelines are clocked at 180 MHz, which is the standard frequency for the NetFPGA-PLUS datapath. In the following, after a brief Vitis Networking P4 architecture overview, we detail basic SPADA data structures implementations, and two full-fledged SPADA monitoring pipelines synthesized for the Xilinx Alveo U280 FPGA, contrasting them with baseline designs.

A.1.1 The Vitis Networking P4 architecture

The Vitis Networking P4 architecture (cf. Figure A.1a) is composed of three main blocks: *(i)* a parser, *(ii)* a match action engine, and *(iii)* a deparser. Extern blocks are directly implemented in Verilog HDL and can be added to the P4 pipeline using a configurable interface. In our prototype, we use externs to deploy data plane accessible registers, similar to those in Banzai (Sivaraman et al., 2016) or Tofino (Agrawal and Kim, 2020).



(A) Vitis Networking P4 Architecture.



(B) P4 pipeline for CHT and qCHT.

FIGURE A.1: Details of P4 FPGA programming framework and CHT and qCHT implementation pipelines.

A.1.2 SPADA building blocks

SPADA monitoring pipelines can be built using two of the following building blocks: MAT, (q)CHT, and pIBLT. Here, we detail their implementation in Vitis Networking P4 while the required hardware resources with the Xilinx Alveo U280 FPGA-based Smart-NIC as target are deferred to Appendix A.2.3.

Match Action Table. The MAT is the basic programming unit offered by the P4 language abstraction. Each MAT is defined by: what data to match on, a list of possible actions, and an optional number of properties, e.g., size, default action, etc. In SPADA, the MAT can be used to implement the *Flow Map* when `FlowToSketch` mappings are statically pushed from the control plane. Its implementation in the Vitis P4 framework is directly supported by the compiler and requires specifying the matching key, the algorithm (exact or ternary), the action, and the table length. The resulting block offers an AXI4 interface that can be wired to the control plane to populate the MAT. We implement two different MATs: one matching on source IP for cardinality estimation (use case ⓐ), and one matching on TCP/IP 5-tuple for IAT quantile estimation (use case ⓑ). Both MAT actions consist of writing in the packet metadata an integer `SketchID`.

Cuckoo Hash Table with quotienting. In SPADA the CHT, optionally with quotienting when needed, can be used for the *Flow Map* or the *Sketch Data*. Its P4 implementation requires d hash functions (implemented with externs in the Vitis P4 framework) to compute the table indexes. Unlike the MAT, implementing a CHT in P4 is challenging due to its non-constant insertion time. Indeed, when all designated CHT indexes for an item are occupied, in our implementation, one of them is randomly chosen, replaced, and reinserted elsewhere. This would require accessing the same memory multiple times in a pipeline, which is not allowed in P4. Hence, CHT insertion may force recirculating keys and values (as we are monitoring, packets can be forwarded normally), potentially impacting the processing rate of the forwarding pipeline. To mitigate this problem, we implement a CHT with four tables, i.e., $d = 4$, augmented with a small memory called *stash* (Kirsch et al., 2010) that enables fixed insertion time via lazy recirculation. The high-level architecture of the simple P4 pipeline is depicted in Figure A.1b (top). At key insertion, the packet first traverses the different pipeline modules one by one, each implementing a single table T_r. If a free position is found, then the item is inserted there; otherwise, the key is stored in the next free slot in the stash. Our system is based on a per-epoch measurement approach in which elements are deleted only at the end of an epoch. Hence it is safe to insert a key in the first available slot without checking if the key is already present in a subsequent table. We note that the same key might be stored twice in the stash, hence possibly recirculated in two separate tables. However, at the end of the measurement epoch, any duplicated value is reconciled by the control plane resolving potential conflicts. We expect this phenomenon to be limited since CHTs are reset at each epoch and packets of the same flow may fall in separate counters. The recirculation process is triggered whenever a stash insertion makes it exceeds a predefined threshold. In this case, we randomly pick a table T_r and insert the evicted item into it, replacing any previously stored item. The replaced item then traverses the pipeline starting from block r, looking for an empty memory slot among its other designated positions in other tables. If an empty slot is found, the item is inserted there, and the recirculation process ends; otherwise, it is stored in the stash, triggering another recirculation.

This solution has the drawback of only recirculating one item at a time. To overcome this limitation, we stack up to four (smaller) CHT that operate on parallel datapaths fed by a hash-based load balancer. Each datapath features its own stash, and recirculation is triggered on all datapaths at the same time. We call this mechanism "batch recirculation" since a single recirculation step moves more than one item at a time. Note that different recirculation policies might be implemented, e.g., recirculate when one (aggressive) or all (conservative) stashes reach the threshold. Finally, an output aggregator is responsible for recomposing a single output stream of packets.

Perfect Invertible Bloom Look-up Table. Due to its simple insertion routine, implementing the pIBLT in P4 does not present particular challenges. Similarly to the CHT, the P4 code employs four externs for $d = 4$ hashes used to identify the indexes within each table, and another one for both the bitmap and the buckets.

Use case	**Pipeline**	**LUT** [K - %]	**LUTRAM** [K - %]	**FF** [K - %]	**BRAM** [# - %]
-	NetFPGA	125.6 - 9.6	17.1 - 2.8	201 - 7.7	279 - 13.8
ⓐ	HLL Baseline	401.7 - 30.8	186.1 - 30.9	225.2 - 8.6	354 - 17.5
	SPADA-HLL (static, 4 datapaths)	206.9 - 15.8	67.3 - 11.1	329 - 12.61	358 - 17.7
ⓑ	DDSketch Baseline	388.7 - 30	187 - 31	237.5 - 9.1	390 - 19.3
	SPADA-DDSketch (static)	183.1 - 14	57.4 - 9.5	241.7 - 9.3	390 - 19.3
ⓒ	ES Baseline	387.2 - 29.7	193.8 - 32.2	227.3 - 8.7	282 - 13.9
	SPADA-ES	185.9 - 14.2	66.8 - 12	248.1 - 9.5	282 - 13.9

TABLE A.1: Prototypes resources utilization.

A.1.3 SPADA-enabled monitoring pipelines

In this section, we use the building blocks described above to implement pipelines for use cases ⓐ, ⓑ, and ⓒ and wrap them in the NetFPGA-Plus architecture. The pipelines feature three P4 blocks: *(i)* a MAT or CHT for the *Flow Map*, *(ii)* a middle block that computes the bucket index, and *(iii)* a pIBLT or qCHT for storing the sparse *Sketch Data*. Table A.1 contrasts the basic NetFPGA-Plus reference NIC hardware requirements to the ones for HLL, DDSketch, and ElasticSketch (with $m = 32$ and $s_c = 16$ bits) wrapped in this reference NIC. It also details resource usage for static and dynamic versions as well as monitoring system baselines. In general, we remark that the additional hardware requirements to deploy our monitoring data plane on top of the NetFPGA-Plus reference NIC architecture are marginal, i.e., $\approx +10\%$. Additionally, we note that baseline implementations require $\approx 2\times$ hardware resources with respect to their SPADA counterparts and would be able to accurately monitor fewer flows in the data plane.

HLL. We implement the HLL sketching algorithm for super spreader detection (use case ⓐ) in two steps within the second and third P4 blocks above. The middle block relies on a single extern to compute the hash of the destination IP and uses the output to *(i)* identify the index of the HLL sketch bucket, and *(ii)* compute the value to be stored in the bucket, i.e., the number of consecutive leading zeroes. The bucket index is concatenated to the `SketchID` to build the key `<SketchID, index>`, and both key and value are attached to the packet as metadata. Finally, the block implementing the *Sketch Data* is responsible for checking whether the newly computed value is greater than the one currently stored, and replacing it if so. For HLL we use smaller $s_c = 8$ bits counters.

DDSketch. For IAT quantile estimation (use case ⓑ), the *Flow Map* stores the `SketchID` and the timestamp of the last packet for each flow. The latter is used to compute the IAT upon reception of a new packet. The IAT value is attached to the packet as metadata and used by the middle block to identify the relevant DDSketch bin `index` through a small MAT table. In particular, we select the longest prefix matching between the current IAT value and precomputed delimiters of DDSketch buckets. The key is built using the `<SketchID, index>` pair and written in the packet metadata. The last block is responsible for incrementing the counter stored in the corresponding bucket.

ElasticSketch. For flow size estimation with ElasticSketch (use case ⓒ), the first P4 block implements the *Flow Map*: four hash tables constituting the heavy part that stores elephant flows. In particular, every flow key is associated with an exact packet counter and positive and negative votes to implement the ostracism mechanism. Flows identified as mice by the ostracism are dynamically evicted. Mice flows are stored in a separate *Sketch Data*: a single-row Count-Min Sketch (CMS). The second

P4 block computes a hash on the flow key for mice flows, thus identifying the corresponding bucket within the CMS, and attaches the `index` as user-defined metadata. Finally, the last block is responsible for incrementing the relevant CMS bucket. Note that, as there is only one sketch, in this case, the key for the *Sketch Data* is composed of the sole bucket `index`.

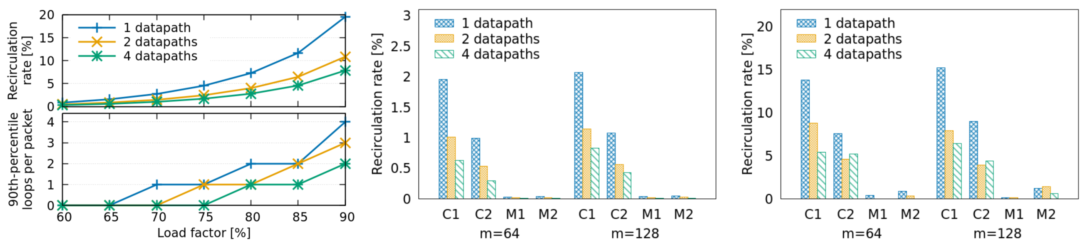
A.2 Prototype Evaluation

In this section we first evaluate the (q)CHT recirculation overhead and discuss its feasibility in real systems. Then, we evaluate latency and throughput of the FPGA implementation, both via real prototype experiments and with accurate Verilog simulations.

A.2.1 Cuckoo Hash Table recirculation overhead

In this part, we evaluate the computational overhead of SPADA-qCHT due to CHT recirculations. For simplicity, we assume asynchronous recirculation loops in our simulations, hence we never experience insertion failures due to stashes overload. In this set of tests, we fix the number of CHT slots to 2^{16}, with 16 additional slots for the stash, equally split among the available datapaths and trigger recirculation when *all* stashes reach a threshold of 50%, unless otherwise specified.

Synthetic keys. We first run a stress test that consists of inserting random keys in the CHT until a target load factor is reached. Figure A.2a (top) shows the average recirculation rate when starting from an empty CHT. We observe that when using a single datapath, the recirculation rate to reach a load factor of 90% is 20%. This drastically improves when using four datapaths thanks to batch recirculation. Note that additional experiments with a more aggressive policy, i.e., recirculate when at least one stash reaches the threshold, reported up to 26% more recirculations w.r.t. the conservative policy when using 4 datapaths. This is due to recirculations triggered when some stashes are still empty, thus wasting available datapaths. Figure A.2a (bottom) provides the 90th percentiles loop length when starting from non-empty CHT, i.e., we count recirculations that occur when the CHT is at the target load in order to evaluate the recirculation overhead in the worst case. We observe that, at 90% load, most packets trigger less than 4 recirculations (90th percentile) when using a single datapath. Recirculation is halved when using 4 datapaths, so that 90% of the packets at 90% load trigger at most 2 recirculations. Further experiments showed that average recirculation is much lower: at 90% load, we measured 1.6 (resp. 0.6) loops per packet with 1 (resp. 4) datapath(s), meaning that every insertion triggered less than one recirculation on average. Thus, in most cases, the recirculation overhead is negligible and does not affect the system performance.



(A) w/ uniform random keys. (B) Overall — w/ real traffic. (C) Worst case — w/ real traffic.

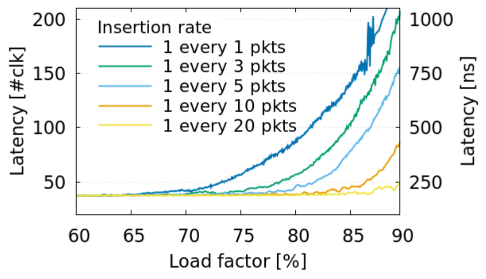
FIGURE A.2: **(a)** CHT recirculation rate (top) and 90th percentile loop length (bottom) vs. load factor with random keys. **(b)** Overall and **(c)** worst-case recirculation rate on real traces for super spreader detection.

Real traces. We then evaluate SPADA-qCHT recirculation overhead on real traffic. Unlike the previous analysis, we remark that on real traffic a large fraction of packets only update already occupied `<SketchID, index>` pairs, thus not triggering any recirculation. Figure A.2b reports the recirculation rate with respect to the overall number of packets. This set of results refers to the HLL use case ⓐ with $m = 64$ and $m = 128$. Simulations are performed using 1, 2, and 4 datapaths and the CHT is dimensioned to reach 90% load factor. We observe an overall recirculation rate below 2% for a single datapath and around 0.5% when using 4 datapaths with CAIDA traces. Recirculation rate drops drastically with MAWI traces as they feature much fewer flows, i.e., sIP in the HLL use case ⓐ. Our findings assess the feasibility of the recirculation approach in real scenarios as the extra bandwidth required to recirculate is negligible. Figure A.2c reports worst-case values, which correspond to a fully loaded table (around 90%). In this case, the recirculation rate is much higher but stays below 5% when 4 datapaths are used. Other data plane applications that exploit recirculation exhibit similar, e.g., (Sonchack et al., 2021) with 2% on average, or higher, e.g., (Sengupta et al., 2022) with 16% in worst-case, recirculation rates.

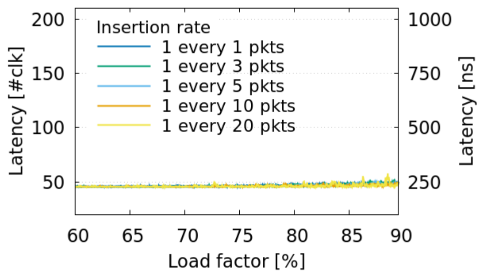
A.2.2 FPGA implementation evaluation

Finally, we evaluate throughput and latency of our SPADA-qCHT FPGA implementation through real FPGA experiments and clock cycle-accurate Verilog simulations. The latter enables better visibility of latency degradation since it usually corresponds to a few clock cycles over the 960 ns of the plain Open NIC Shell. Note that SPADA is a passive monitoring system, meaning that monitoring logic does not delay incoming packets or influence forwarding decisions. Thus, any additional latency introduced by SPADA is due to *(i)* the arbiter that multiplexes input packets and *(ii)* processing of recirculated packets.

Figure A.3 shows CHT per-packet latency of the SPADA-qCHT Verilog simulation at maximum throughput, varying the insertion rate, that is, how many packets trigger the insertion of a new key. Note that the latency in the figure does not take into account the latency overhead of the Xilinx Open NIC Shell used to host the system.



(A) Single datapath.



(B) 2 datapaths.

FIGURE A.3: FPGA insertion latency vs. load factor at different insertion rates (average over 100 runs).

With a single datapath, latency is ≈ 40 clock cycles (200 ns) for a low CHT load factor, i.e., $<65\%$. At 90% load, latency increases to more than 200 clock cycles (1000 ns) in the worst case that every packet triggers a new CHT insertion. However, the latency increase is much more limited at lower insertion rates, and becomes negligible when performing 1 insertion every 20 packets. Note that in CAIDA and MAWI traces, worst case insertion rate is 1 every 38 packets. As depicted in Figure A.3b, when using 2 datapaths, we do not observe latency increase even at very high load factors and insertion rates (4 datapaths not shown as results are similar).

Concerning throughput, our simulations show no throughput degradation and no-failure insertions up to 85% load factor for a single datapath, and up to 89.0% for 2 datapaths, and 4 datapaths also in the worst case of 1 insertion at every packet. This is mainly because we clock the system at 180 MHz, and the maximum throughput with minimum size packets (64B) at 100 Gbps is 144 Mpps thereby leaving 36 Mpps for packet recirculations. To confirm the Verilog simulations we tested our FPGA prototype with minimum-sized packets at maximum throughput. The experiments confirm no throughput degradation, while the measured latency is 1160ns, that is 200 ns latency due to SPADA-qCHT plus 960 ns overhead given by the Xilinx Open NIC Shell.

A.2.3 FPGA resource requirements

Table A.2 reports the hardware resource requirements for SPADA building blocks namely MAT, CHT, qCHT, and pIBLT. Data structures are dimensioned for 2^{14} entries in the *Flow Map*, each with 2^{16} buckets of 16 bits in the *Sketch Data*, assuming "virtual sketches" with $m = 32$ buckets unless otherwise specified. The table reports the number of LUTs (expressed in thousands, K, and in percentage, %) used as logic, those used as distributed RAMs (LUTRAM), and the number of Flip-Flops (FF) and Block RAMs (BRAM). It is worth mentioning that due to the Vitis P4 framework architecture, the MAT Table A.2 (top) is mainly mapped to the BRAM memory element and that, as expected, the 5-tuple one requires more resources as it has a bigger flow key.

Datapaths	**Building Block**	**LUT** [K - %]	**LUTRAM** [K - %]	**FF** [K - %]	**BRAM** [# - %]
-	MAT (Src IP)	7.3 - 0.57	2.3 - 0.4	12.2 - 0.47	73 - 3.62
-	MAT (5-tuple)	10.3 - 0.8	2.8 - 0.5	18.4 - 0.71	109 - 5.41
1	CHT	53.2 - 4.09	43.7 - 7.27	41.6 - 1.6	1 - 0.05
	qCHT	37.3 - 2.87	29.3 - 4.88	32.2 - 1.24	1 - 0.05
2	CHT	57.8 - 4.44	43.3 - 7.21	58.2 - 2.32	3 - 0.15
	qCHT	48.3 - 3.71	34.9 - 5.81	57.8 - 2.22	3 - 0.15
4	CHT	80 - 6.14	53.8 - 8.96	109.5 - 4.20	5 - 0.25
	qCHT	70.9 - 5.44	46.2 - 7.70	108.8 - 4.17	5 - 0.25
-	pIBLT	40.5 - 3.11	35.4 - 5.9	8.8 - 0.34	1 - 0.05

TABLE A.2: MAT, (q)CHT, and pIBLT resources utilization.

Table A.2 (middle) details the FPGA hardware resources required by CHT and qCHT with 1, 2, and 4 datapaths, for $m = 32$. We recall that the qCHT is a CHT that stores a quotient instead of the full key, hence saving a significant amount of memory as the table highlights. The extra memory (mainly BRAM) for multiple datapaths is due to the load balancing and interconnection overhead. This overhead is fixed, i.e., does not depend on the hash table size, and is also related to the specific synthesizer optimizations. In particular, we observed that the memory required to synthesize a single datapath diminishes by increasing the number of datapaths. With multiple datapaths, such memory reduction may compensate for the extra memory required by the load balancer.

Table A.2 (bottom) details the memory requirements for the pIBLT with 2^{16} counters and a bitmap B of size 2^{19} bits, assuming "virtual sketches" with $m = 32$. It is worth mentioning that in this case pIBLT occupies fewer resources with respect to qCHT. This is mainly due to the limited amount of sketch counters m. Increasing m would lead to a bigger bitmap B and hence bigger pIBLT.

Appendix B

Appendix for Chapter 4

B.1 Additional Model Analysis

Here we provide additional analysis on the data used for training and evaluation and on the model itself. In particular, in Figure B.1 we analyze the flow size distributions of each trace, broken down by protocol. As expected, ICMP traffic features shorter flows. We report model performance including ICMP traffic in Figure B.2, while we based our evaluation on combined TCP and UDP traffic in the chapter.

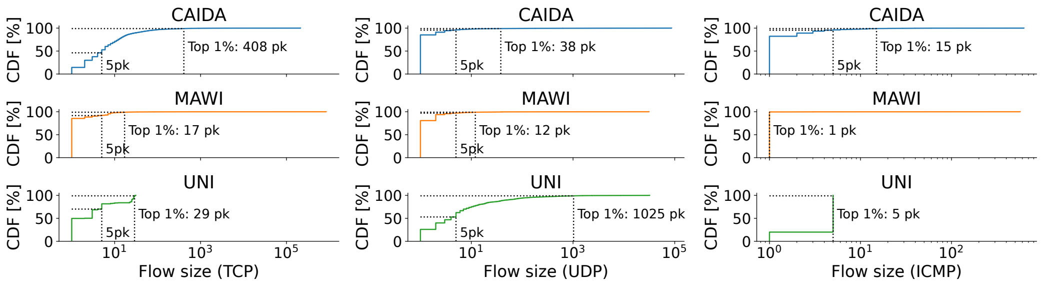


FIGURE B.1: Flow sizes distributions by protocol for the 50th minute of each trace.

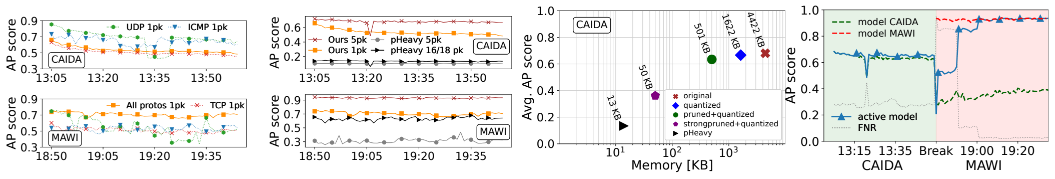


FIGURE B.2: Model evaluation including all three protocols (i.e., TCP, UDP and ICMP) on CAIDA and MAWI. The large volume of ICMP flows makes the classification task easier, especially on MAWI.

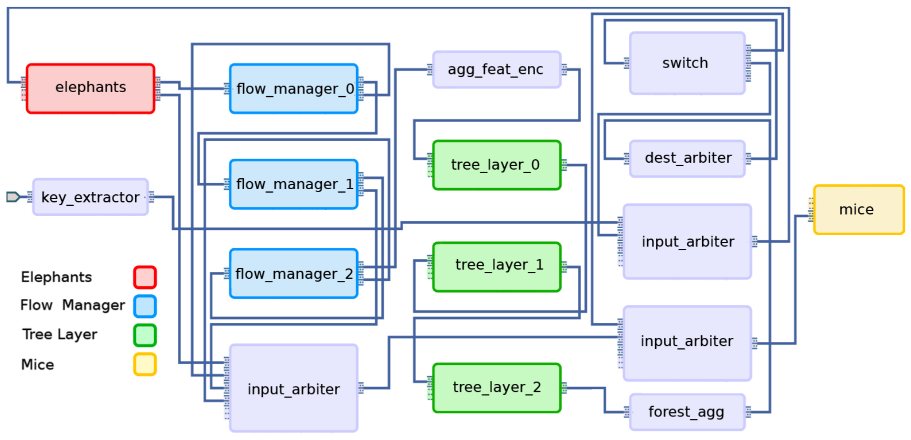


FIGURE B.3: Xilinx Vivado block diagram of the DUMBO pipeline for a simplified 3-layer RF model and 3-stage Flow Manager.

B.2 System Prototype

In this section, we provide details on the implementation of DUMBO on a Xilinx FPGA-based SmartNIC target. We deferred this implementation work here because it has not been conducted as part of the PhD program and was the responsibility of external collaborators from Sapienza University, Rome.

We realize a fully working prototype wrapping the DUMBO architecture into the Xilinx OpenNIC shell (*AMD OpenNIC Project* 2023), providing a system able to process a 100 GbE link. We consider the flow size estimation use case, where the Mice Tracker is a Count-Min Sketch with 4×2^{14} buckets of 2 bytes and the Elephant Tracker is a hash table with 4×2^{12} slots containing exact counters. The Flow Manager is composed of a total of 2^{15} slots. For the RF hybrid implementation, we implement a model composed of 33 trees with a maximum depth $max_{t \in RF}(D_t) = 23$ levels and a maximum number of nodes per layer of $N = 940$. Thus, we fix the threshold to switch from the full tree implementation to the indexed encoding at $M = 10$. All the syntheses have been carried out targeting the AMD-Xilinx Alveo U280 and an operating frequency of 180 MHz, which is more than the one needed to sustain the full throughput of 144 Mpps, i.e., the maximum rate for a 100 GbE link.

Figure B.3 depicts the main DUMBO building blocks where, for the sake of clarity, we only show a 3-layer tree instead of the full RF model and a 3-stage Flow Manager. Packets flow from left to right, neglecting the light grey blocks which contain minor glue logic. First, packets traverse the Elephant Tracker, then three levels of Flow Manager, three layers of decision tree, and are finally directed to the Mice Tracker or the Elephant Tracker based on the prediction. Finally, packet and byte counters are disseminated into the OpenNIC shell to monitor the datapath and identify the location of possible packet drops.

Table B.1 reports the FPGA resource consuption of the full system and of the DUMBO pipeline (neglecting the OpenNIC shell). The full system requires less than 15% of

	DUMBO	Full system
CLB LUTs	93878 (7.20%)	187419 (14.38%)
CLB Registers	242467 (9.30%)	369083 (14.16%)
Block RAM	717 (35.57%)	860.5 (42.68%)

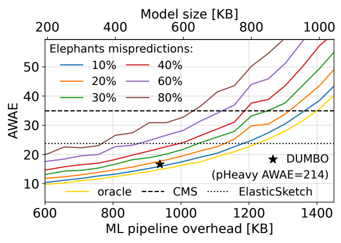
TABLE B.1: DUMBO prototype resources utilization.

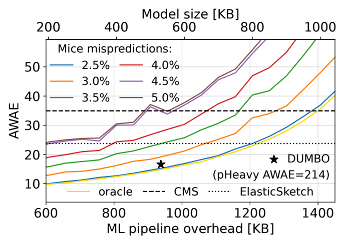
available logic resources and ≈40% of memory resources. Note that a significant fraction of logic resources is due to the fixed overhead of the OpenNIC infrastructure. Instead, if needed, DUMBO resources can be augmented in terms of memory for Elephant and Mice Trackers and a larger RF Model.

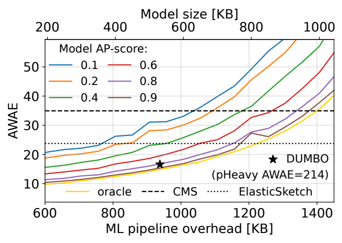
For performance, we measured FPGA throughput and latency. The full system is able to sustain the full 100 GbE throughput regardless of packet size. To assess the latency of the prototype, we test both the full system included in the shell and the plain OpenNIC shell without the DUMBO pipeline. The baseline OpenNIC shell has a latency of around 960 ns. For the whole system, as we consider the flow size estimation use case which is a pure monitoring application, the forwarding decision is taken independently from the counting procedure. Thus, there is no latency degradation due to the Machine Learning pipeline. If instead we consider the scheduling use case, the packet is classified after traversing the Elephant Tracker and Flow Manager blocks, inducing an additional latency of 14 clock cycles (≈70 ns).

B.3 Additional Trade-offs Analysis

Here we provide additional results on the CAIDA trace concerning the trade-offs between memory overhead and end-to-end performance, and how these are impacted by the model mispredictions. Similar to the analyses presented in the technical chapter, we simulate various misprediction rates taking into account that ≈20K flows are classified as elephants. Note that these rates are simulated by tweaking the confusion matrix, thus, the resulting predictions do not depend on actual flow features but are instead stochastic. Figure B.4ab show the impact of elephants and mice mispredictions varying memory overhead on combined TCP and UDP traffic (Figure B.4a also appears in the technical chapter), while Figure B.5ab show results for TCP-only traffic. Additionally, we show in Figure B.4c and Figure B.5c how the offline AP-score metric translates into end-to-end average weighted absolute error for combined TCP and UDP traffic, and TCP-only respectively. Last, in Figure B.6 and Figure B.7 we show how changing the misprediction rates affects the Mean Relative Error (MRE) when computing multiple quantiles for the IAT estimation use case, respectively on TCP+UDP and TCP-only traffic (Figure B.6c also appears in the main chapter). Note that the DUMBO marker (black star) corresponds to the actual model performance while solid lines refer to simulated performance trade-offs by artificially modifying the model confusion matrix.

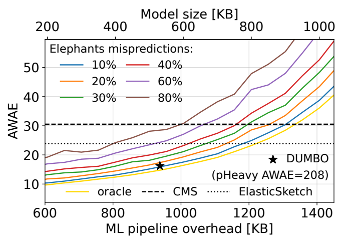


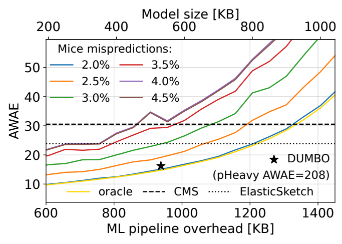


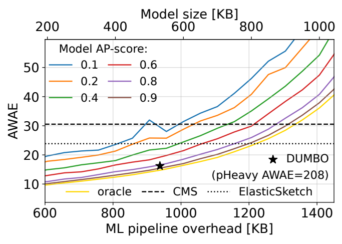


(A) Impact of FNR. (B) Impact of FPR. (C) Impact of AP score.

FIGURE B.4: Impact of mispredictions on flow size estimation use case (TCP and UDP).

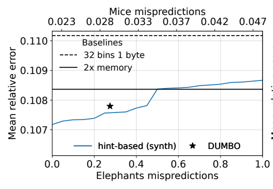


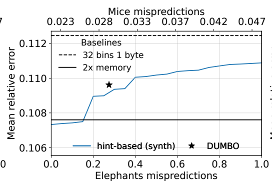


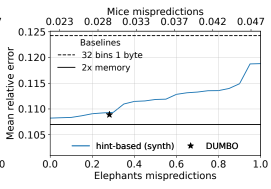


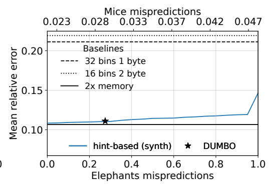
(A) Impact of FNR. (B) Impact of FPR. (C) Impact of AP score.

FIGURE B.5: Impact of mispredictions on flow size estimation use case (TCP only).



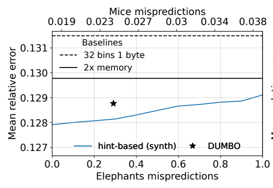


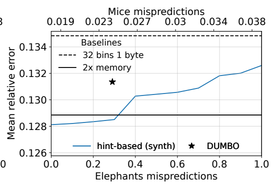


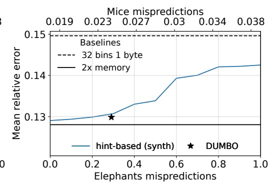


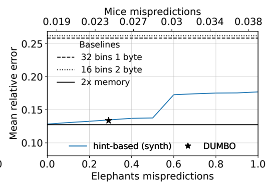
(A) 75-th quantile. (B) 90-th quantile. (C) 95-th quantile. (D) 99-th quantile.

FIGURE B.6: Impact of mispredictions on IAT use case (TCP and UDP).









(A) 75-th quantile. (B) 90-th quantile. (C) 95-th quantile. (D) 99-th quantile.

FIGURE B.7: Impact of mispredictions on IAT use case (TCP only).

Appendix C

Appendix for Chapter 5

C.1 Affinity Scores Computation

In Table C.1, we detail the computation of each selected affinity score. Considering two tasks $t_1 = a$ and $t_2 = b$ and a batch of examples $\mathcal{X}$, we denote:

- their resp. losses functions $\mathcal{L}_a$ and $\mathcal{L}_b$
- their resp. STL models STL_a and STL_b with losses
  - $\mathcal{L}_{STL_a} = \mathcal{L}_a(\mathcal{X}, STL_a)$
  - $\mathcal{L}_{STL_b} = \mathcal{L}_b(\mathcal{X}, STL_b)$
- their joint MTL model $MTL_{(a,b)}$ with loss
  - $\mathcal{L}_{MTL_{(a,b)}} = \mathcal{L}_a(\mathcal{X}, MTL_{(a,b)}) + \mathcal{L}_b(\mathcal{X}, MTL_{(a,b)})$

Note that if the score is symmetric, it assesses how much the two tasks help each other regardless of direction. If it is asymmetric, it considers how much the target task a benefits from being learned with the partner task b. While all scores could not be constrained to lie in the same range, higher always means more affinity.

Affinity scoring	**Type**	**Computation**	**Comment**	**Range**
`Taxonomic. distance (TD)`	Model-agn.	Distance between tasks in a taxonomy tree.	Symmetric. Taxonomy borrowed from (Zamir et al., 2018). Multiplied by -1 for consistency i.e., higher is better.	$]-\infty, 0]$
`Input attr. similarity (IAS)`	STL-based	$\frac{1}{\lvert\mathcal{X}\rvert}\sum_{x\in\mathcal{X}} S_{cos}(Attr(STL_a,x), Attr(STL_b,x)),$ (C.1) where S_{cos} is cosine similarity, $\mathcal{X}$ denotes a batch of examples and $Attr$ the attribution method used.	Symmetric. Revisited from (Song et al., 2019). Computed on a subset of the test set (2,048 images) using InputXGradient attribution (Shrikumar et al., 2017).	$[-1,+1]$
`Repr. similarity analysis (RSA)`	STL-based	$RSA(\theta_{Ba},\theta_{Bb},\mathcal{X}),$ (C.2) where RSA denotes the Representation Similiarity Analysis, $\mathcal{X}$ a batch of examples, θ_{Ba} and θ_{Bb} the backbone weights of the STL models for tasks a and b resp.	Symmetric. Revisited from (Dwivedi and Roig, 2019). Computed on a subset of the test set (2,048 images).	$[-1,+1]$
`Label injection (LI)`	STL-based	$\frac{\mathcal{L}_{STL_a}-\mathcal{L}_{STL_{a\leftarrow b}}}{\mathcal{L}_{STL_{a\leftarrow b}}},$ (C.3) where $STL_{a\leftarrow b}$ represents the STL model for task a, modified to ingest the corresponding label from task b in addition to the input.	Asymmetric. Novel proposal. Computed using test losses.	$]-\infty,+\infty[$
`Gradient similarity (GS)`	MTL-based	$\frac{1}{N}\sum_{i=1}^{N} S_{cos}(\frac{\partial\mathcal{L}_a(\mathcal{X},\theta_B^i,\theta_{Ha}^i)}{\partial\theta_B^i}, \frac{\partial\mathcal{L}_b(\mathcal{X},\theta_B^i,\theta_{Hb}^i)}{\partial\theta_B^i}),$ (C.4) where N denotes the number of training epochs, S_{cos} the cosine similarity, $\mathcal{X}$ a batch of examples, θ_B^i the weights of the common MTL backbone at the i^{th} epoch, θ_{Ha}^i and θ_{Hb}^i the weights of the heads for a and b at the i^{th} epoch.	Symmetric. Borrowed from (Fifty et al., 2021; Zhao et al., 2018).	$[-1,+1]$
`Gradient transf. (GT)`	MTL-based	$\frac{1}{N}\sum_{i=1}^{N} 1-\frac{\mathcal{L}_a(\mathcal{X},\theta_{B\mid b}^{i+1},\theta_{Ha}^i)}{\mathcal{L}_a(\mathcal{X},\theta_B^i,\theta_{Ha}^i)},$ (C.5) where N denotes the number of training epochs, $\mathcal{X}$ a batch of examples, $\theta_{B\mid b}^{i+1}$ the weights of the common MTL backbone updated using the loss of task b at the epoch $i+1$, θ_{Ha}^i and θ_{Hb}^i the weights of the heads for a and b at the i^{th} epoch.	Asymmetric. Borrowed from (Fifty et al., 2021).	$]-\infty,+\infty[$

TABLE C.1: Tasks affinity scores description and computation considering two tasks $t_1 = a$ and $t_2 = b$.

C.2 Taskonomy Buildings

We split our subset of the Taskonomy dataset into train, validation and test sets, on a per-building basis.

Train set These buildings amount to 603,437 input images.

- adairsville
- airport
- albertville
- anaheim
- ancor
- andover
- annona
- arkansaw
- athens
- bautista
- bohemia
- bonesteel
- bonnie
- broseley
- browntown
- byers
- scioto
- nuevo
- goodfield
- donaldson
- hanson
- merom
- klickitat
- onaga
- leonardo
- marstons
- newfields
- pinesdale
- lakeville
- cosmos
- benevolence
- pomaria
- tolstoy
- shelbyville
- allensville
- wainscott
- beechwood
- coffeen
- stockman
- hiteman
- woodbine
- lindenwood
- forkland
- mifflinburg
- ranchester
- springerville
- swisshome
- westfield
- willow
- winooski
- hainesburg
- irvine
- pearce
- thrall
- tilghmanton
- uvalda
- sugarville
- silas

Validation set These buildings amount to 82,345 input images.

- corozal
- collierville
- markleeville
- darden
- chilhowie
- churchton
- cauthron
- cousins
- timberon

- wando

Test set These buildings amount to 40,367 input images.

- ihlen
- muleshoe
- noxapater
- mcdade

C.3 Affinity Scores Raw Values

In Table C.2 to C.7, we report the raw affinities estimations for all tasks, using each affinity scoring technique. Results are rounded at the second decimal.

	Affinities estimations				
with	**SemSeg**	**Keypts**	**Edges**	**Depth**	**Normal**
SemSeg	-	-8	-6	-8	-5
Keypts	-8	-	-4	-12	-9
Edges	-6	-4	-	-10	-7
Depth	-8	-12	-10	-	-5
Normal	-5	-9	-7	-5	-

TABLE C.2: `Taxonomical distance (TD)`. Distance between tasks in the similarity tree from (Zamir et al., 2018). Multiplied by -1 for consistency (i.e., higher means more affinity).

	Affinities estimations				
with	**SemSeg**	**Keypts**	**Edges**	**Depth**	**Normal**
SemSeg	-	0.25	0.23	0.31	0.45
Keypts	0.25	-	0.52	0.18	0.23
Edges	0.23	0.52	-	0.18	0.22
Depth	0.31	0.18	0.18	-	0.29
Normal	0.45	0.23	0.22	0.29	-

TABLE C.3: `Input attribution similarity (IAS)`. Cosine similarity between STL models attribution maps.

	Affinities estimations				
with	**SemSeg**	**Keypts**	**Edges**	**Depth**	**Normal**
SemSeg	-	0.33	0.37	0.46	0.46
Keypts	0.33	-	0.66	0.04	0.05
Edges	0.37	0.66	-	0.12	0.13
Depth	0.46	0.04	0.12	-	0.69
Normal	0.46	0.05	0.13	0.69	-

TABLE C.4: `Representation similarity analysis (RSA)`. Representation similarity analysis using the STL models backbones output.

	Affinities estimations				
with	**SemSeg**	**Keypts**	**Edges**	**Depth**	**Normal**
SemSeg	-	-2.93	-3.50	-3.07	+3.70
Keypts	-8.47	-	+4.97	-8.31	+1.42
Edges	-15.93	-4.20	-	-9.58	+2.42
Depth	+4.04	-3.34	-1.26	-	+20.29
Normal	+25.68	+60.29	+23.79	+66.30	-

TABLE C.5: `Label injection (LI)`. Performance gain (%) when incorporating the label from the partner task in the STL model's input, relative to standard STL.

	Affinities estimations				
with	**SemSeg**	**Keypts**	**Edges**	**Depth**	**Normal**
SemSeg	-	0.51	0.39	1.93	1.54
Keypts	0.51	-	1.89	0.75	1.0
Edges	0.39	1.89	-	0.92	0.59
Depth	1.93	0.75	0.92	-	8.40
Normal	1.54	1.0	0.59	8.40	-

TABLE C.6: `Gradient similarity (GS)`. Cosine similarity between task-specific gradient updates on the MTL backbone. Averaged across all training epochs. Multiplied by 100.

	Affinities estimations				
with	**SemSeg**	**Keypts**	**Edges**	**Depth**	**Normal**
SemSeg	-	+0.02	+0.25	+1.69	+0.74
Keypts	-0.03	-	+0.38	-0.01	+0.01
Edges	-0.20	+0.71	-	+0.19	+0.27
Depth	+0.47	+0.01	+0.15	-	+0.90
Normal	+0.27	+0.03	+0.16	+1.26	-

TABLE C.7: `Gradient transference (GT)`. Look-ahead ratio simulating the effect of applying task-specific updates to the MTL backbone for the other task. Averaged across all training epochs.

Bibliography

Abbasloo, Soheil, Chen-Yu Yen, and H. Jonathan Chao (2020). “Classic Meets Modern: A Pragmatic Learning-Based Congestion Control for the Internet”. In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’20. Virtual Event, USA: Association for Computing Machinery, 632–647. ISBN: 9781450379557. DOI: 10.1145/3387514.3405892.

Aceto, Giuseppe et al. (2019). “MIRAGE: Mobile-app Traffic Capture and Ground-truth Creation”. In: *2019 4th International Conference on Computing, Communications and Security (ICCCS)*, pp. 1–8. DOI: 10.1109/CCCS.2019.8888137.

Achille, Alessandro et al. (2019). “Task2vec: Task embedding for meta-learning”. In: *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 6430–6439.

Addanki, Vamsi, Maciej Pacut, and Stefan Schmid (2024). “Credence: Augmenting Datacenter Switch Buffer Sharing with ML Predictions”. In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI’24)*.

Agrawal, Anurag and Changhoon Kim (2020). “Intel Tofino2 – A 12.9Tbps P4-Programmable Ethernet Switch”. In: *2020 IEEE Hot Chips 32 Symposium (HCS)*, pp. 1–32. DOI: 10.1109/HCS49909.2020.9220636.

Akem, Aristide Tanyi-Jong, Michele Gucciardo, and Marco Fiore (2023). “Flowrest: Practical Flow-Level Inference in Programmable Switches with Random Forests”. In: *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*.

Akem, Aristide Tanyi-Jong et al. (2022). “Henna: Hierarchical machine learning inference in programmable switches”. In: *Proceedings of the 1st International Workshop on Native Network Intelligence*.

Alizadeh, Mohammad et al. (2010). “Data center tcp (dctcp)”. In: *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 63–74.

Alizadeh, Mohammad et al. (2013). “pfabric: Minimal near-optimal datacenter transport”. In: *ACM SIGCOMM Computer Communication Review* 43.4, pp. 435–446.

AMD (2023). *Xilinx Vitis Networking P4, https://www.xilinx.com/products/intellectual-property/ef-di-vitisnetp4.html*. AMD Inc. (Visited on 06/22/2023).

AMD OpenNIC Project (2023). https://github.com/Xilinx/open-nic.

Amit, Yali and Donald Geman (1997). “Shape quantization and recognition with randomized trees”. In: *Neural computation* 9.7, pp. 1545–1588.

Amodei, Dario et al. (2016). “Concrete problems in AI safety”. In: *arXiv preprint arXiv:1606.06565*.

Ando, Rie Kubota, Tong Zhang, and Peter Bartlett (2005). “A framework for learning predictive structures from multiple tasks and unlabeled data.” In: *Journal of Machine Learning Research* 6.11.

Arasu, Arvind and Gurmeet Singh Manku (2004). “Approximate Counts and Quantiles over Sliding Windows”. In: *Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’04. Paris, France: Association for Computing Machinery, 286–296. ISBN: 158113858X. DOI: 10.1145/1055558.1055598.

Arp, Daniel et al. (2022). “Dos and don’ts of machine learning in computer security”. In: *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3971–3988.

Arzani, Behnaz et al. (2018). “007: Democratically Finding the Cause of Packet Drops”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, pp. 419–435. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/arzani.

Atmaja, Bagus Tris, Akira Sasou, and Masato Akagi (2022). “Survey on bimodal speech emotion recognition from acoustic and linguistic information fusion”. In: *Speech Communication*.

Azorin, Raphael et al. (2021). “Towards a Generic Deep Learning Pipeline for Traffic Measurements”. In: *Proceedings of the CoNEXT Student Workshop*. CoNEXT-SW ’21. Virtual Event, Germany: Association for Computing Machinery, 5–6. ISBN: 9781450391337. DOI: 10.1145/3488658.3493785.

Azorin, Raphael et al. (2023). “It’s a Match! A Benchmark of Task Affinity Scores for Joint Learning”. In: *Association for the Advancement of Artificial Intelligence (AAAI). 2nd International Workshop on Practical Deep Learning in the Wild*. Washington D.C., USA. URL: https://arxiv.org/pdf/2301.02873.pdf.

Azorin, Raphael et al. (Mar. 2024). “Taming the Elephants: Affordable Flow Length Prediction in the Data Plane”. In: *Proceedings of the ACM on Networking* 2.CoNEXT1. DOI: 10.1145/3649473.

Banino, Andrea et al. (2018). “Vector-based navigation using grid-like representations in artificial agents”. In: *Nature* 557.7705, pp. 429–433.

Bar-Yossef, Ziv et al. (2002). “Counting Distinct Elements in a Data Stream”. In: *Randomization and Approximation Techniques in Computer Science*. Ed. by José D. P. Rolim and Salil Vadhan. RANDOM 2002. Springer, pp. 1–10. ISBN: 978-3-540-45726-8.

Barbette, Tom et al. (Feb. 2020). “A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency”. In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, pp. 667–683. ISBN: 978-1-939133-13-7.

Barut, Onur et al. (2021). “Multi-task hierarchical learning based network traffic analytics”. In: *ICC 2021-IEEE International Conference on Communications*. IEEE, pp. 1–6.

Behringer, Michael H. et al. (May 2021). *A Reference Model for Autonomic Networking*. RFC 8993. DOI: 10.17487/RFC8993. URL: https://www.rfc-editor.org/info/rfc8993.

Ben Basat, Ran et al. (2017). “Optimal elephant flow detection”. In: *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, pp. 1–9.

Ben Basat, Ran et al. (2020a). “Designing heavy-hitter detection algorithms for programmable switches”. In: *IEEE/ACM Transactions on Networking* 28.3, pp. 1172–1185.

Ben Basat, Ran et al. (2020b). “PINT: Probabilistic In-Band Network Telemetry”. In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’20. Virtual Event, USA: Association for Computing Machinery, 662–680. ISBN: 9781450379557. DOI: 10.1145/3387514.3405894.

Benson, Theophilus, Aditya Akella, and David A Maltz (2010). “Network traffic characteristics of data centers in the wild”. In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280.

Benson, Theophilus et al. (2011). “MicroTE: Fine Grained Traffic Engineering for Data Centers”. In: *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*. CoNEXT ’11. Tokyo, Japan: Association for Computing Machinery. ISBN: 9781450310413. DOI: 10.1145/2079296.2079304.

Biau, Gérard and Erwan Scornet (2016). “A random forest guided tour”. In: *arXiv preprint arXiv:1511.05741*.

Bickel, Steffen et al. (2008). “Multi-task learning for HIV therapy screening”. In: *Proceedings of the 25th International Conference on Machine Learning*. ICML ’08. Helsinki, Finland: Association for Computing Machinery, 56–63. ISBN: 9781605582054. DOI: 10.1145/1390156.1390164.

Bingel, Joachim and Anders Søgaard (2017). “Identifying beneficial task relations for multi-task learning in deep neural networks”. In: *arXiv preprint arXiv:1702.08303*.

Bloom, Burton H (1970). “Space/time trade-offs in hash coding with allowable errors”. In: *Communications of the ACM* 13.7, pp. 422–426.

Bobda, Christophe et al. (2022). “The future of FPGA acceleration in datacenters and the cloud”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15.3, pp. 1–42.

Boutaba, Raouf et al. (2018). “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities”. In: *Journal of Internet Services and Applications* 9.1, pp. 1–99.

Breiman, Leo (2001). “Random Forests”. In: *Machine Learning* 45.

– (2017). *Classification and regression trees*. Routledge.

Bronzino, Francesco et al. (2021). “Traffic Refinery: Cost-Aware Data Representation for Machine Learning on Network Traffic”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 5.3. DOI: 10.1145/3491052.

Brown, Tom et al. (2020). “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33, pp. 1877–1901.

Busato, Federico et al. (2018). “Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs”. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7. DOI: 10.1109/HPEC.2018.8547541.

Busse-Grawitz, Coralie et al. (2022). *pForest: In-Network Inference with Random Forests*. arXiv: 1909.05680 [cs.NI].

Caruana, Rich (1997). “Multitask learning”. In: *Machine learning* 28.1, pp. 41–75.

Chaudet, Claude et al. (2005). “Optimal positioning of active and passive monitoring devices”. In: *Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pp. 71–82.

Chen, Haoxian et al. (2016). “Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis”. In: *Proceedings of the Symposium on SDN Research*. SOSR ’16. Santa Clara, CA, USA: Association for Computing Machinery. ISBN: 9781450342117. DOI: 10.1145/2890955.2890971.

Chen, Tianqi and Carlos Guestrin (2016). “Xgboost: A scalable tree boosting system”. In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794.

Chen, Xiaoqi et al. (2020). “BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time”. In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’20. Virtual Event, USA: Association for Computing Machinery, 226–239. ISBN: 9781450379557. DOI: 10.1145/3387514.3405865.

Chen, Xinxiong et al. (2015). “Joint learning of character and word embeddings”. In: *Twenty-fourth international joint conference on artificial intelligence*.

Chen, Zhao et al. (2018). “Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks”. In: *International conference on machine learning*. PMLR, pp. 794–803.

Cho, Kyunghyun et al. (2014). “On the properties of neural machine translation: Encoder-decoder approaches”. In: *arXiv preprint arXiv:1409.1259*.

Choi, Baek-Young et al. (2007). “Quantile sampling for practical delay monitoring in Internet backbone networks”. In: *Computer Networks* 51.10, pp. 2701–2716.

Chollet, François (2017). “Xception: Deep learning with depthwise separable convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258.

Cichy, Radoslaw Martin et al. (2016). “Comparison of deep neural networks to spatio-temporal cortical dynamics of human visual object recognition reveals hierarchical correspondence”. In: *Scientific reports* 6.1, pp. 1–13.

Cisco (2018). *Cisco Global Cloud Index (2016–2021) White Paper*. Tech. rep. URL: https://newsroom.cisco.com/c/r/newsroom/en/us/a/y2018/m02/global-cloud-index-projects-cloud-traffic-to-represent-95-percent-of-total-data-center-traffic-by-2021.html.

– (2020). *Cisco Annual Internet Report (2018–2023) White Paper*. Tech. rep. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

Cohen, Dvir et al. (2020). “DANTE: A Framework for Mining and Monitoring Darknet Traffic”. In: *Computer Security – ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I*. Springer-Verlag, 88–109.

Collet, Alan et al. (2023). “AutoManager: a Meta-Learning Model for Network Management from Intertwined Forecasts”. In: *IEEE International Conference on Computer Communications*.

Cormode, Graham and Minos Garofalakis (2005). "Sketching Streams through the Net: Distributed Approximate Query Tracking". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 13–24. ISBN: 1595931546.

Cormode, Graham and Shan Muthukrishnan (2005). "An improved data stream summary: the count-min sketch and its applications". In: *Journal of Algorithms* 55.1, pp. 58–75.

Cranor, Chuck et al. (2003). "Gigascope: A stream database for network applications". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 647–651.

Cueva, Christopher J and Xue-Xin Wei (2018). "Emergence of grid-like representations by training recurrent neural networks to perform spatial localization". In: *arXiv preprint arXiv:1803.07770*.

D'Amour, Alexander et al. (2022). "Underspecification presents challenges for credibility in modern machine learning". In: *The Journal of Machine Learning Research* 23.1, pp. 10237–10297.

Di Cicco, Nicola et al. (2023). "Poster: Continual Network Learning". In: *Proceedings of the ACM SIGCOMM 2023 Conference*, pp. 1096–1098.

Dietterich, Thomas G (2000). "Ensemble methods in machine learning". In: *International workshop on multiple classifier systems*. Springer, pp. 1–15.

Dietzfelbinger, Martin et al. (2010). "Tight Thresholds for Cuckoo Hashing via XORSAT". In: *Automata, Languages and Programming*. Ed. by Samson Abramsky et al. Springer, pp. 213–225. ISBN: 978-3-642-14165-2.

Dosovitskiy, Alexey et al. (2020). "An image is worth 16x16 words: Transformers for image recognition at scale". In: *arXiv preprint arXiv:2010.11929*.

Draper-Gil, Gerard et al. (2016). "Characterization of encrypted and vpn traffic using time-related". In: *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP)*, pp. 407–414.

Du, Elbert, Franklyn Wang, and Michael Mitzenmacher (2021). "Putting the "Learning" into Learning-Augmented Algorithms for Frequency Estimation". In: *38th International Conference on Machine Learning*. PMLR.

Dubois, O. and J. Mandler (2002). "The 3-XORSAT threshold". In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* IEEE, pp. 769–778. DOI: 10.1109/SFCS.2002.1182002.

Duffield, Nick, Carsten Lund, and Mikkel Thorup (2001). “Charging from sampled network usage”. In: *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pp. 245–256.

DUMBO Simulator (2024). URL: https://github.com/cpt-harlock/DUMBO (visited on 02/28/2024).

Durand, Marianne and Philippe Flajolet (2003). “Loglog counting of large cardinalities”. In: *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*. Springer, pp. 605–617.

Dwivedi, Kshitij and Gemma Roig (2019). “Representation similarity analysis for efficient task taxonomy & transfer learning”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12387–12396.

D’Alconzo, Alessandro et al. (2019). “A survey on big data for network traffic monitoring and analysis”. In: *IEEE Transactions on Network and Service Management* 16.3, pp. 800–813.

Ediger, David et al. (2012). “STINGER: High performance data structure for streaming graphs”. In: *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, pp. 1–5. DOI: 10.1109/HPEC.2012.6408680.

Elastic Sketch source code (2023). https://github.com/BlockLiu/ElasticSketchCode.

Estan, Cristian and George Varghese (Aug. 2003). “New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice”. In: *ACM Trans. Comput. Syst.* 21.3, 270–313. ISSN: 0734-2071. DOI: 10.1145/859716.859719.

Feamster, Nick and Jennifer Rexford (2017). “Why (and how) networks should run themselves”. In: *arXiv preprint arXiv:1710.11583*.

Feldmann, Anja et al. (2000). “Deriving traffic demands for operational IP networks: methodology and experience”. In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’00. Stockholm, Sweden: Association for Computing Machinery, 257–270. ISBN: 1581132239. DOI: 10.1145/347059.347554.

Fifty, Chris et al. (2021). “Efficiently identifying task groupings for multi-task learning”. In: *Advances in Neural Information Processing Systems* 34, pp. 27503–27516.

Firestone, Daniel et al. (2018). “Azure Accelerated Networking: SmartNICs in the Public Cloud”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 51–66.

Flajolet, Philippe and G Nigel Martin (1985). “Probabilistic counting algorithms for data base applications”. In: *Journal of computer and system sciences* 31.2, pp. 182–209.

Flajolet, Philippe et al. (June 2007). “HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm”. In: *AofA: Analysis of Algorithms*. DMTCS Proceedings DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07). Ed. by Philippe Jacquet, pp. 137–156. DOI: 10.46298/dmtcs.3545.

Fornasier, Massimo and Holger Rauhut (2015). “Compressive Sensing.” In: *Handbook of mathematical methods in imaging* 1, pp. 187–229.

Fotakis, Dimitris et al. (2005). “Space efficient hash tables with worst case constant access time”. In: *Theory of Computing Systems* 38.2, pp. 229–248.

Franzius, Mathias, Henning Sprekeler, and Laurenz Wiskott (2007). “Slowness and sparseness lead to place, head-direction, and spatial-view cells”. In: *PLoS computational biology* 3.8, e166.

Gao, Peter X et al. (2019). “phost: Distributed near-optimal datacenter transport over commodity network fabric”. In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pp. 1–12.

Gioacchini, Luca et al. (2021). “DarkVec: automatic analysis of darknet traffic with word embeddings”. In: *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. ACM.

Gonzalez, Roberto et al. (2021). “User Profiling by Network Observers”. In: *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. ACM.

Goodrich, Michael T. and Michael Mitzenmacher (2011). “Invertible bloom lookup tables”. In: *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, pp. 792–799. DOI: 10.1109/Allerton.2011.6120248.

Greenwald, Michael and Sanjeev Khanna (2001). “Space-efficient online computation of quantile summaries”. In: *ACM SIGMOD Record* 30.2, pp. 58–66.

Hafting, Torkel et al. (2005). “Microstructure of a spatial map in the entorhinal cortex”. In: *Nature* 436.7052, pp. 801–806.

Han, Hui et al. (2022). “Applications of sketches in network traffic measurement: A survey”. In: *Information Fusion* 82, pp. 58–85.

Hartmann, Heinrich and Theo Schlossnagle (2020). *Circllhist – A Log-Linear Histogram Data Structure for IT Infrastructure Monitoring*. DOI: 10.48550/ARXIV.2001.06561.

He, Kaiming et al. (2016). “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*.

Ho, Tin Kam (1998). “The random subspace method for constructing decision forests”. In: *IEEE transactions on pattern analysis and machine intelligence* 20.8, pp. 832–844.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long Short-Term Memory”. In: *Neural Computation* 9.8, pp. 1735–1780.

Holland, Jordan et al. (2021). “New Directions in Automated Traffic Analysis”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. DOI: 10.1145/3460120.3484758.

Houidi, Zied Ben et al. (2022). “Towards a Systematic Multi-Modal Representation Learning for Network Data”. In: *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. HotNets ’22. Austin, Texas: Association for Computing Machinery, 181–187. ISBN: 9781450398992. DOI: 10.1145/3563766.3564108.

Howard, Andrew G et al. (2017). “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861*.

Hsu, Chen-Yu et al. (2019). “Learning-Based Frequency Estimation Algorithms”. In: *International Conference on Learning Representations*.

Huang, Qun et al. (Apr. 2021). “Toward Nearly-Zero-Error Sketching via Compressive Sensing”. In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, pp. 1027–1044. ISBN: 978-1-939133-21-2.

Hubel, David H and Torsten N Wiesel (1959). “Receptive fields of single neurones in the cat’s striate cortex”. In: *The Journal of physiology* 148.3, p. 574.

– (1968). “Receptive fields and functional architecture of monkey striate cortex”. In: *The Journal of physiology* 195.1, pp. 215–243.

Hui, Linbo et al. (2023). “Digital Twin for Networking: A Data-Driven Performance Modeling Perspective”. In: *IEEE Network* 37.3, pp. 202–209. DOI: 10.1109/MNET.119.2200080.

Ibanez, Stephen et al. (2019). “The p4 to netfpga workflow for line-rate packet processing”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 1–9.

Ivkin, Nikita et al. (2019). “QPipe: Quantiles Sketch Fully in the Data Plane”. In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. CoNEXT ’19. Orlando, Florida: Association for Computing Machinery, 285–291. ISBN: 9781450369985. DOI: 10.1145/3359989.3365433.

Jacobs, Arthur S. et al. (2022). “AI/ML and Network Security: The Emperor has no Clothes”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’22. Los Angeles, CA, USA: Association for Computing Machinery.

Jang, Rhongho et al. (2020). “Sketchflow: Per-flow systematic sampling using sketch saturation event”. In: *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, pp. 1339–1348.

Jia, Peng et al. (2020). “Accurately Estimating User Cardinalities and Detecting Super Spreaders over Time”. In: *IEEE Transactions on Knowledge and Data Engineering* 34.1, pp. 92–106.

Karnin, Zohar, Kevin Lang, and Edo Liberty (2016). “Optimal quantile approximation in streams”. In: *2016 ieee 57th annual symposium on foundations of computer science (focs)*. IEEE, pp. 71–78.

Kendall, Alex, Yarin Gal, and Roberto Cipolla (2018). “Multi-task learning using uncertainty to weigh losses for scene geometry and semantics”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7482–7491.

Kendall, Maurice George (1948). *Rank correlation methods.*

Kim, Myung-Sup et al. (2004). “A flow-based method for abnormal network traffic detection”. In: *2004 IEEE/IFIP Network Operations and Management Symposium (IEEE Cat. No.04CH37507)*. Vol. 1, 599–612 Vol.1. DOI: 10.1109/NOMS.2004.1317747.

Kim, Yoon et al. (2016). “Character-aware neural language models”. In: *Thirtieth AAAI conference on artificial intelligence*.

Kingma, Diederik P. and Max Welling (2014). “Auto-Encoding Variational Bayes”. In: *2nd International Conference on Learning Representations, ICLR*.

Kirsch, Adam, Michael Mitzenmacher, and Udi Wieder (2010). “More Robust Hashing: Cuckoo Hashing with a Stash”. In: *SIAM Journal on Computing* 39.4, pp. 1543–1561. DOI: 10.1137/080728743.

Knuth, Donald Ervin (1973). *The art of computer programming: sorting and searching*, pp. 723–723.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2017). “Imagenet classification with deep convolutional neural networks”. In: *Communications of the ACM* 60.6, pp. 84–90.

Kruger, Norbert et al. (2012). “Deep hierarchies in the primate visual cortex: What can we learn for computer vision?” In: *IEEE transactions on pattern analysis and machine intelligence* 35.8, pp. 1847–1871.

Kučera, Jan et al. (2020). “Detecting routing loops in the data plane”. In: *Proceedings of the 16th International Conference on emerging Networking Experiments and Technologies*, pp. 466–473.

Kumar, Gautam, Akshay Narayan, and Peter Gao (2016). *YAPS Network Simulator*. URL: https://github.com/NetSys/simulator (visited on 06/22/2023).

Kumar, Gautam et al. (2020). “Swift: Delay is Simple and Effective for Congestion Control in the Datacenter”. In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’20. Virtual Event, USA: Association for Computing Machinery, 514–528. ISBN: 9781450379557. DOI: 10.1145/3387514.3406591.

Lall, Ashwin et al. (2006). “Data streaming algorithms for estimating entropy of network traffic”. In: *ACM SIGMETRICS Performance Evaluation Review* 34.1, pp. 145–156.

Lample, Guillaume et al. (2017). “Unsupervised machine translation using monolingual corpora only”. In: *arXiv preprint arXiv:1711.00043*.

Leang, Isabelle et al. (2020). “Dynamic task weighting methods for multi-task networks in autonomous driving systems”. In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, pp. 1–8.

Lee, Chunghan et al. (2015). “Flow-Aware Congestion Control to Improve Throughput under TCP Incast in Datacenter Networks”. In: *2015 IEEE 39th Annual Computer Software and Applications Conference*. Vol. 3, pp. 155–162. DOI: 10.1109/COMPSAC.2015.225.

Lee, Jong-Hyouk and Kamal Singh (2020). “Switchtree: in-network computing and traffic analyses with random forests”. In: *Neural Computing and Applications*, pp. 1–12.

Li, Bingdong et al. (2013). “A survey of network flow applications”. In: *Journal of Network and Computer Applications* 36.2, pp. 567–581.

Li, Wenxin et al. (2024). “Flow Scheduling with Imprecise Knowledge”. In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI’24)*.

Li, Yuliang et al. (2016a). “FlowRadar: A Better NetFlow for Data Centers”. In: *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, pp. 311–324.

– (2016b). “LossRadar: Fast Detection of Lost Packets in Data Center Networks”. In: *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. CoNEXT ’16. Irvine, California, USA: Association for Computing Machinery, 481–495. ISBN: 9781450342926. DOI: 10.1145/2999572.2999609.

Li, Yuliang et al. (2019). “HPCC: High Precision Congestion Control”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM ’19. Beijing, China: Association for Computing Machinery, 44–58. ISBN: 9781450359566. DOI: 10.1145/3341302.3342085.

Liu, Weijiang, Chao Liu, and Shuming Guo (2016). “A hash-based algorithm for measuring cardinality distribution in network traffic”. In: *International Journal of Autonomous and Adaptive Communications Systems* 9.1-2, pp. 136–148.

Liu, Ziwei et al. (2018). “Large-scale celebfaces attributes (celeba) dataset”. In: *Retrieved August* 15.2018, p. 11.

Marill, Thomas and Lawrence G Roberts (1966). “Toward a cooperative network of time-shared computers”. In: *Proceedings of the November 7-10, 1966, fall joint computer conference*, pp. 425–431.

Masson, Charles, Jee E. Rim, and Homin K. Lee (Aug. 2019). “DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees”. In: *Proc. VLDB Endow.* 12.12, 2195–2205. ISSN: 2150-8097. DOI: 10.14778/3352063.3352135.

Michel, Oliver et al. (2021). “The programmable data plane: Abstractions, architectures, algorithms, and applications”. In: *ACM Computing Surveys (CSUR)* 54.4, pp. 1–36.

Mikolov, Tomas, Quoc V Le, and Ilya Sutskever (2013a). “Exploiting similarities among languages for machine translation”. In: *arXiv preprint arXiv:1309.4168*.

Mikolov, Tomas et al. (2013b). “Distributed representations of words and phrases and their compositionality”. In: *Advances in neural information processing systems* 26.

Misra, Ishan et al. (2016). “Cross-stitch networks for multi-task learning”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3994–4003.

Mitchell, Tom M (1980). *The need for biases in learning generalizations*.

Mitzenmacher, Michael (2021). “Queues with small advice”. In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, pp. 1–12.

Monterubbiano, Andrea et al. (2022). “Learned Data Structures for Per-Flow Measurements”. In: *Proceedings of the 3rd International CoNEXT Student Workshop*. CoNEXT Student Workshop ’22. Rome, Italy: Association for Computing Machinery, 42–43. ISBN: 9781450399371. DOI: 10.1145/3565477.3569147.

Monterubbiano, Andrea et al. (2023a). “Lightweight Acquisition and Ranging of Flows in the Data Plane”. In: *Proceedings of the ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, Vol. 7, No. 3, Article 44, December 2023*. SIGMETRICS ’24. Association for Computing Machinery. DOI: 10.1145/3626775.

Monterubbiano, Andrea et al. (2023b). “Memory-Efficient Random Forests in FPGA SmartNICs”. In: *Companion of the 19th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT Posters ’23. Paris, France: Association for Computing Machinery, 55–56. ISBN: 9798400704079. DOI: 10.1145/3624354.3630089.

– (Nov. 2023c). “SPADA: A Sparse Approximate Data Structure Representation for Data Plane Per-Flow Monitoring”. In: *Proceedings of the ACM on Networking* 1.CoNEXT3. DOI: 10.1145/3629149.

Nascita, Alfredo et al. (2023). “Improving Performance, Reliability, and Feasibility in Multimodal Multitask Traffic Classification with XAI”. In: *IEEE Transactions on Network and Service Management* 20.2, pp. 1267–1289. DOI: 10.1109/TNSM.2023.3246794.

NetFPGA publisher (2023). *NetFPGA-PLUS, https://netfpga.org/NetFPGA-PLUS.html*. (Visited on 06/22/2023).

Netscout (2023). *Netscout DDoS Threat Intelligence Report - Issue 11*. Tech. rep. URL: https://www.netscout.com/threatreport.

ONNX. Open Neural Network Echange (2017). https://github.com/onnx/onnx.

O’Reilly, Colin et al. (2014). “Anomaly Detection in Wireless Sensor Networks in a Non-Stationary Environment”. In: *IEEE Communications Surveys & Tutorials* 16.3, pp. 1413–1432. DOI: 10.1109/SURV.2013.112813.00168.

Owaida, Muhsen et al. (2017). “Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms”. In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, pp. 1–8.

Pagh, Rasmus and Flemming Friche Rodler (2004). “Cuckoo hashing”. In: *Journal of Algorithms* 51.2, pp. 122–144. ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.12.002.

Pan, Rong et al. (2003). “Approximate fairness through differential dropping”. In: *ACM SIGCOMM Computer Communication Review* 33.2, pp. 23–39.

Panigrahy, Rina (2004). “Efficient hashing with lookups in two memory accesses”. In: *arXiv preprint cs/0407023*.

Papapetrou, Odysseas, Minos Garofalakis, and Antonios Deligiannakis (June 2015). “Sketching Distributed Sliding-Window Data Streams”. In: *The VLDB Journal* 24.3, 345–368. ISSN: 1066-8888. DOI: 10.1007/s00778-015-0380-7.

Pascal, Lucas et al. (2021). “Maximum Roaming Multi-Task Learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.10, pp. 9331–9341.

Pedregosa, F. et al. (2011). “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Perry, Jonathan et al. (2014). “Fastpass: A centralized" zero-queue" datacenter network”. In: *Proceedings of the 2014 ACM conference on SIGCOMM*, pp. 307–318.

Peters, ME et al. (2018). “Deep contextualized word representations. arXiv 2018”. In: *arXiv preprint arXiv:1802.05365* 12.

Pittel, Boris and Gregory B. Sorkin (2016). “The Satisfiability Threshold for k-XORSAT”. In: *Combinatorics, Probability & Computing* 25.2, pp. 236–268. DOI: 10.1017/S0963548315000097.

Pontarelli, Salvatore et al. (Feb. 2019). “FlowBlaze: Stateful Packet Processing in Hardware”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, pp. 531–548. ISBN: 978-1-931971-49-2.

Popa, Lucian et al. (2009). “Macroscope: End-point approach to networked application dependency discovery”. In: *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pp. 229–240.

Poupart, Pascal et al. (2016). “Online flow size prediction for improved network routing”. In: *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pp. 1–6. DOI: 10.1109/ICNP.2016.7785324.

Quinlan, J. Ross (1986). “Induction of decision trees”. In: *Machine learning* 1, pp. 81–106.

Reginald P., Tewarson (1973). *Sparse Matrices*. Mathematics in science and engineering: a series of monographs and textbooks. Academic Press. ISBN: 9780126856507.

Ren, Jingjing et al. (2018). “A longitudinal study of pii leaks across android app versions”. In: *Network and Distributed System Security Symposium (NDSS)*. Vol. 10.

Ring, Markus et al. (2017). “IP2Vec: Learning Similarities Between IP Addresses”. In: *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, 657–666. DOI: 10.1109/ICDMW.2017.93.

Rivitti, Alessandro et al. (2023). “eHDL: Turning eBPF/XDP Programs into Hardware Designs for the NIC”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 208–223.

Rusek, Krzysztof et al. (2020). “Routenet: Leveraging graph neural networks for network modeling and optimization in sdn”. In: *IEEE Journal on Selected Areas in Communications* 38.10, pp. 2260–2270.

Sacco, Alessio, Flavio Esposito, and Guido Marchetto (2020). “A Federated Learning Approach to Routing in Challenged SDN-Enabled Edge Networks”. In: *2020 6th*

IEEE Conference on Network Softwarization (NetSoft), pp. 150–154. DOI: 10.1109/NetSoft48620.2020.9165506.

Scazzariello, Mariano et al. (2023). "A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 1237–1255.

Schlinker, Brandon et al. (2019). "Internet Performance from Facebook's Edge". In: *Proceedings of the Internet Measurement Conference*. IMC '19. Association for Computing Machinery, 179–194. ISBN: 9781450369480. DOI: 10.1145/3355369.3355567.

Sekar, Vyas et al. (2008). "CSAMP: A System for Network-Wide Flow Monitoring". In: *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. San Francisco, CA: USENIX Association.

Sengupta, Satadal, Hyojoon Kim, and Jennifer Rexford (2022). "Continuous In-Network Round-Trip Time Monitoring". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM '22. Amsterdam, Netherlands: Association for Computing Machinery, 473–485. ISBN: 9781450394208. DOI: 10.1145/3544216.3544222.

Settles, Burr (2011). "From theories to queries: Active learning in practice". In: *Active learning and experimental design workshop in conjunction with AISTATS 2010*. JMLR Workshop and Conference Proceedings, pp. 1–18.

Shahout, Rana, Roy Friedman, and Ran Ben Basat (2023). "Together is Better: Heavy Hitters Quantile Estimation". In: *Proceedings of the ACM on Management of Data* 1.1, pp. 1–25.

Sharafaldin, Iman, Arash Habibi Lashkari, and Ali A Ghorbani (2018). "Toward generating a new intrusion detection dataset and intrusion traffic characterization." In: *ICISSP* 1, pp. 108–116.

Sheng, Siyuan et al. (June 2021). "PR-Sketch: Monitoring per-Key Aggregation of Streaming Data with Nearly Full Accuracy". In: *Proc. VLDB Endow.* 14.10, 1783–1796. ISSN: 2150-8097. DOI: 10.14778/3467861.3467868.

Shrikumar, Avanti, Peyton Greenside, and Anshul Kundaje (2017). "Learning important features through propagating activation differences". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 3145–3153.

Siracusano, Giuseppe et al. (Apr. 2022). "Re-architecting Traffic Analysis with Neural Network Interface Cards". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, pp. 513–533. ISBN: 978-1-939133-27-4.

Sivaraman, Anirudh et al. (2016). "Packet Transactions: High-Level Programming for Line-Rate Switches". In: *Proceedings of the 2016 ACM SIGCOMM Conference*.

SIGCOMM ’16. Florianopolis, Brazil: Association for Computing Machinery, 15–28. ISBN: 9781450341936. DOI: 10.1145/2934872.2934900.

Sivaraman, Vibhaalakshmi et al. (2017). “Heavy-hitter detection entirely in the data plane”. In: *Proceedings of the Symposium on SDN Research*, pp. 164–176.

Sonchack, John et al. (2021). “Lucid: A Language for Control in the Data Plane”. In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM ’21. Virtual Event, USA: Association for Computing Machinery, 731–747. ISBN: 9781450383837. DOI: 10.1145/3452296.3472903.

Song, Cha Hwan et al. (2020). “FCM-Sketch: Generic Network Measurements with Data Plane Support”. In: *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT ’20. Barcelona, Spain: Association for Computing Machinery, 78–92. ISBN: 9781450379489. DOI: 10.1145/3386367.3432729.

Song, Jie et al. (2019). “Deep model transferability from attribution maps”. In: *Advances in Neural Information Processing Systems* 32.

Standley, Trevor et al. (2020). “Which tasks should be learned together in multi-task learning?” In: *International Conference on Machine Learning*. PMLR, pp. 9120–9132.

Sun, Ximeng et al. (2020). “Adashare: Learning what to share for efficient deep multi-task learning”. In: *Advances in Neural Information Processing Systems* 33, pp. 8728–8740.

Tang, Lu, Qun Huang, and Patrick PC Lee (2019). “Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams”. In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, pp. 2026–2034.

The CAIDA Anonymized Internet Traces Dataset (2016). https://www.caida.org/catalog/datasets/passive_dataset/.

The MAWI Working Group Traffic Archive (2019). http://mawi.wide.ad.jp/mawi/.

Thrun, Sebastian and Joseph O’Sullivan (1996). “Discovering structure in multiple learning tasks: The TC algorithm”. In: *ICML*. Vol. 96. Citeseer, pp. 489–497.

Tian, Yonglong et al. (2020). “Rethinking few-shot image classification: a good embedding is all you need?” In: *European Conference on Computer Vision*. Springer, pp. 266–282.

Đukić, Vojislav et al. (2019). “Is advance knowledge of flow sizes a plausible assumption”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 565–580.

Umuroglu, Yaman et al. (2017). “FINN: A Framework for Fast, Scalable Binarized Neural Network Inference”. In: *Proceedings of the 2017 ACM/SIGDA International*

Symposium on Field-Programmable Gate Arrays. FPGA '17. Monterey, California, USA: Association for Computing Machinery, 65–74. ISBN: 9781450343541. DOI: 10.1145/3020078.3021744.

Van Horn, Grant et al. (2018). "The inaturalist species classification and detection dataset". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8769–8778.

Varghese, George and Cristian Estan (2004). "The measurement manifesto". In: *ACM SIGCOMM Computer Communication Review* 34.1, pp. 9–14.

Vaswani, Ashish et al. (2017). "Attention is all you need". In: *Advances in neural information processing systems* 30.

Venkataraman, Shobha et al. (2005). "New streaming algorithms for fast detection of superspreaders." In: *NDSS*. Vol. 5, pp. 149–166.

Wah, Catherine et al. (2011). *The caltech-ucsd birds-200-2011 dataset*.

Walzer, Stefan (2021). "Peeling Close to the Orientability Threshold: Spatial Coupling in Hashing-Based Data Structures". In: *Proceedings of the Thirty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '21. Virtual Event, Virginia: Society for Industrial and Applied Mathematics, 2194–2211. ISBN: 9781611976465.

Wang, Chao et al. (2022a). "AppClassNet: a commercial-grade dataset for application identification research". In: *SIGCOMM Comput. Commun. Rev.* 52.3, 19–27. ISSN: 0146-4833. DOI: 10.1145/3561954.3561958.

Wang, Haibo et al. (2021). "Randomized error removal for online spread estimation in data streaming". In: *Proceedings of the VLDB Endowment* 14.6, pp. 1040–1052.

Wang, Mowei et al. (2018). "Machine Learning for Networking: Workflow, Advances and Opportunities". In: *IEEE Network* 32.2, pp. 92–99. DOI: 10.1109/MNET.2017.1700200.

Wang, Mowei et al. (2022b). "xNet: Improving Expressiveness and Granularity for Network Modeling with Graph Neural Networks". In: *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pp. 2028–2037. DOI: 10.1109/INFOCOM48880.2022.9796726.

Winter, Martin, Rhaleb Zayer, and Markus Steinberger (2017). "Autonomous, independent management of dynamic graphs on GPUs". In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7. DOI: 10.1109/HPEC.2017.8091058.

Xia, Wenfeng et al. (2017). "A Survey on Data Center Networking (DCN): Infrastructure and Operations". In: *IEEE Communications Surveys & Tutorials* 19.1, pp. 640–656. DOI: 10.1109/COMST.2016.2626784.

Xiao, Qingjun et al. (June 2015). "Hyper-Compact Virtual Estimators for Big Network Data Based on Register Sharing". In: *SIGMETRICS Perform. Eval. Rev.* 43, 417–428.

Xie, Yinglian et al. (2005). "Worm origin identification using random moonwalks". In: *2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE, pp. 242–256.

Xiong, Zhaoqi and Noa Zilberman (2019). "Do switches dream of machine learning? toward in-network classification". In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pp. 25–33.

Yang, Lily et al. (2004). *Forwarding and Control Element Separation (ForCES) Framework*. RFC 1654. RFC Editor. URL: https://www.rfc-editor.org/in-notes/rfc3746.txt.

Yang, Lixuan et al. (2021). "Deep Learning and Zero-Day Traffic Classification: Lessons Learned From a Commercial-Grade Dataset". In: *IEEE Transactions on Network and Service Management* 18.4, pp. 4103–4118. DOI: 10.1109/TNSM.2021.3122940.

Yang, Tong et al. (2018). "Elastic Sketch: Adaptive and Fast Network-Wide Measurements". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 561–575. ISBN: 9781450355674. DOI: 10.1145/3230543.3230544.

Yang, Tong et al. (2019). "HeavyKeeper: an accurate algorithm for finding Top-k elephant flows". In: *IEEE/ACM Transactions on Networking* 27.5, pp. 1845–1858.

Yassine, Abdulsalam, Hesam Rahimi, and Shervin Shirmohammadi (2015). "Software defined network traffic measurement: Current trends and challenges". In: *IEEE Instrumentation & Measurement Magazine* 18.2, pp. 42–50.

Yu, Tianhe et al. (2020). "Gradient surgery for multi-task learning". In: *Advances in Neural Information Processing Systems* 33, pp. 5824–5836.

Yuster, Raphael and Uri Zwick (July 2005). "Fast Sparse Matrix Multiplication". In: *ACM Transaction on Algorithms* 1.1, 2–13. ISSN: 1549-6325. DOI: 10.1145/1077464.1077466.

Zamir, Amir R et al. (2018). "Taskonomy: Disentangling task transfer learning". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3712–3722.

Zeiler, Matthew D and Rob Fergus (2014). "Visualizing and understanding convolutional networks". In: *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*. Springer, pp. 818–833.

Zeng, Chaoliang et al. (Apr. 2022). "Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, pp. 1345–1358. ISBN: 978-1-939133-27-4.

Zeydan, Engin and Yekta Turk (2020). "Recent Advances in Intent-Based Networking: A Survey". In: *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pp. 1–5. DOI: 10.1109/VTC2020-Spring48590.2020.9128422.

Zhang, Qizhen et al. (2021a). "MimicNet: fast performance estimates for data center networks with machine learning". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pp. 287–304.

Zhang, Tianzhu et al. (2022). "Interpreting AI for Networking: Where We Are and Where We Are Going". In: *IEEE Communications Magazine* 60.2, pp. 25–31. DOI: 10.1109/MCOM.001.2100736.

Zhang, Xiaoquan et al. (2021b). "pHeavy: Predicting heavy flows in the programmable data plane". In: *IEEE Transactions on Network and Service Management* 18.4, pp. 4353–4364.

Zhang, Ying (2013). "An Adaptive Flow Counting Method for Anomaly Detection in SDN". In: CoNEXT '13. Santa Barbara, California, USA: Association for Computing Machinery, 25–30. ISBN: 9781450321013. DOI: 10.1145/2535372.2535411.

Zhang, Yipeng, Tyler L Hayes, and Christopher Kanan (2021c). "Disentangling Transfer and Interference in Multi-Domain Learning". In: *arXiv preprint arXiv:2107.05445*.

Zhang, Yu and Qiang Yang (2022). "A Survey on Multi-Task Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 34.12, pp. 5586–5609. DOI: 10.1109/TKDE.2021.3070203.

Zhang, Zheng et al. (2015). "A survey of sparse representation: algorithms and applications". In: *IEEE access* 3, pp. 490–530.

Zhao, Qi, Jun Xu, and Abhishek Kumar (2006). "Detection of super sources and destinations in high-speed networks: Algorithms, analysis and evaluation". In: *IEEE Journal on Selected Areas in Communications* 24.10, pp. 1840–1852.

Zhao, Xiangyun et al. (2018). "A modulation module for multi-task learning with applications in image retrieval". In: *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 401–416.

Zhao, Zongyi et al. (2021). "Efficient and Accurate Flow Record Collection With HashFlow". In: *IEEE Transactions on Parallel and Distributed Systems* 33.5, pp. 1069–1083.

Zheng, Changgang and Noa Zilberman (2021). “Planter: seeding trees within switches”. In: *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*, pp. 12–14.

Zheng, Changgang et al. (2022a). “Automating in-network machine learning”. In: *arXiv preprint arXiv:2205.08824*.

Zheng, Changgang et al. (2022b). “IIsy: Practical in-network classification”. In: *arXiv preprint arXiv:2205.08243*.

Zhou, Yu et al. (2020). “Flow event telemetry on programmable data plane”. In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pp. 76–89.

Zhu, Yibo et al. (2015). “Packet-level telemetry in large datacenter networks”. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 479–491.